1. NIS

1.1. ASWI

1.1.1. Základní modely životního cyklu software, softwarový proces, metodika.

Proces – systematická série akcí vedoucí k určitému výsledku

Životní cyklus (meta proces?) – proces od zahájení vývoje až po vyřazení z provozu

Metodika

Definovaný proces pro konkrétní účel, tj. definuje fáze, aktivity, role, artefakty, milníky atd. jsou dobře popsány metodikami:

- Booch method
- SSADM
- Rational Unified Process
- SCRUM
- …

- souhrn doporučených praktik a postupů, pokrývajících celý životní cyklus vytvářené aplikace
- UML není metodika!

Softwarový proces

Systematická série fází a **aktivit**, souvisejících **rolí** a **artefaktů** vedoucí k vyšší pravděpodobnosti úspěšného vytvoření potřebného software.

- Výsledek = kvalitní software
- Členění: fáze, **aktivity**
  - ***Technické aktivity****: komunikace, plánování, modelování, konstrukce, nasazení*
  - ***Podpůrné aktivity****: řízení, kontrola kvality, správa konfigurace, dokumentace*

- Mezivýsledky: **artefakty**
  - ***Technické****: specifikace, dokumentace, testy, modely, ...*
  - ***Komunikační****: specifikace, plán*
  - ***Obchodní****: plán, rozpočet, produkt*

- Činitelé: **role**
  - ***Technické role:*** *analytik, architekt, vývojář, tester, databázista, ...*
  - ***Manažerské role****: team leader, šéf vývojářů, šéf projektů, CEO, CIO, ...*
  - ***Podpůrné role****: poradce, lektor, uživ. podpora, dokumentace, ...*

Varianty a příklady procesů:

Společná snaha = snížení rizika chaotického postupu

- **Řízené plánem** (sledovat plán – hra na jistotu)
  - *Vodopád, V-model*
  - Kontext neměnný, zadání a technologie zřejmé, rozsah malý

- **Řízené riziky** (omezovat rizika)
  - *Průzkumník/prototypování, Spirálový model*
  - Kontext zřejmý, zadání a/nebo technologie nejasné

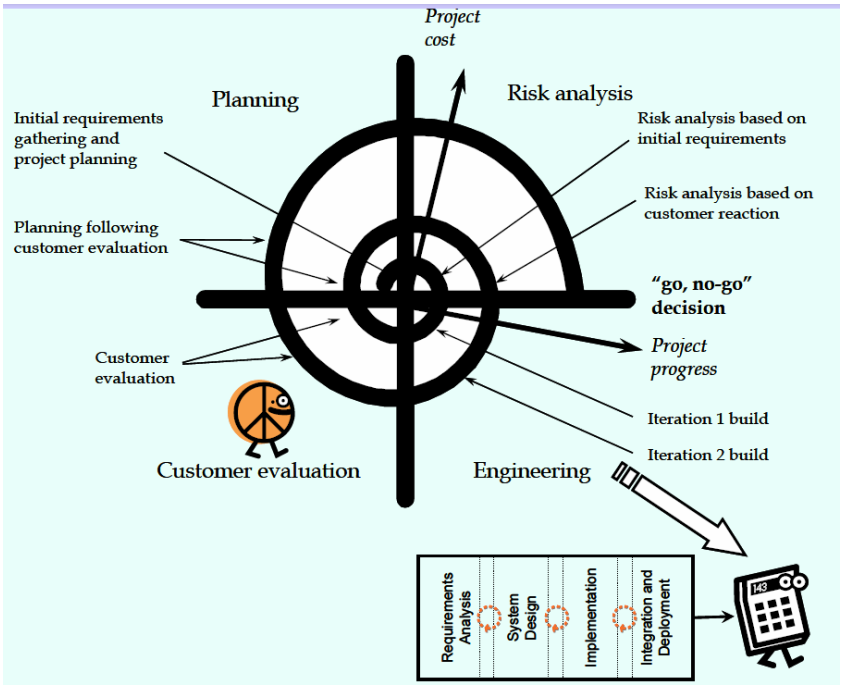
- **Řízené změnou** (adaptace na změnu)
  - *Iterativní – RUP, agilní – SCRUM*
  - Kontext proměnlivý, zadání a/nebo technologie nejasné

Základní modely životního cyklu

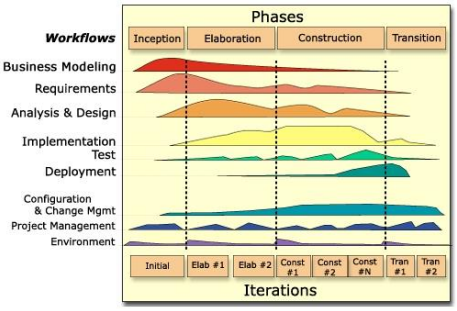
- **Sekvenční** - *velký třesk* (vztažené na celý produkt, naplánované pro celý projekt, oddělené meziprodukty)
  - *Vodopádový model*

- **Cyklický** - opakování technických aktivit, přírůstky (roste znalost, funkcionalita, kvalita, …)

  - ***Spirálový model***



- ***RUP***

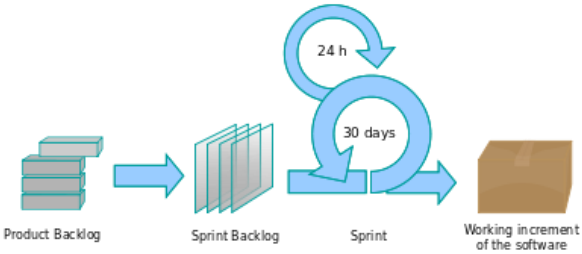


- *Inkrementální (přírůstek je nějaká část (funkcionalita) produktu, produkt je funkční až na konci vývoje)*
- *Iterativní (na konci každé iterace je funkční produkt, např. Na počátku jen se základní funkcionalitou, ale v dalších iteracích se funkcionalita postupně rozšiřuje)*

- **Agilní** - evoluce

  - **Extrémní programování *XP***

  - ***SCRUM***



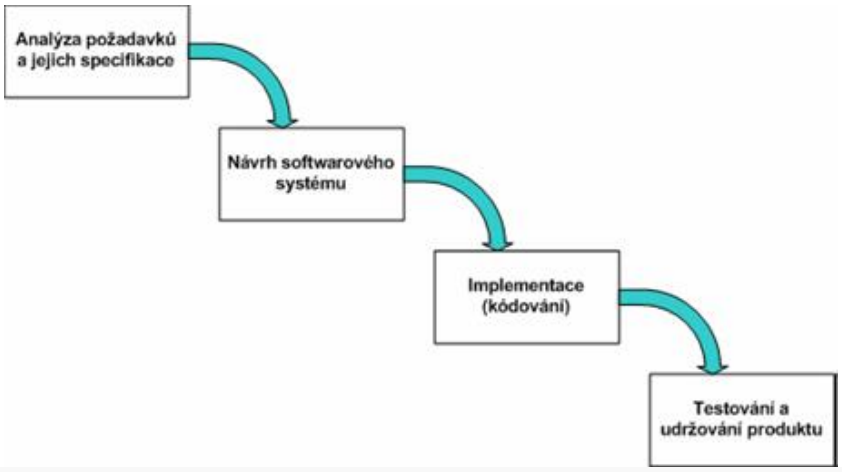
  - Důraz na manifest:
    - Jednotlivci a interakce před procesy a nástroji
    - Fungující software před vyčerpávající dokumentací
    - Spolupráce se zákazníkem před vyjednáváním o smlouvě
    - Reagování na změny před dodržováním plánu

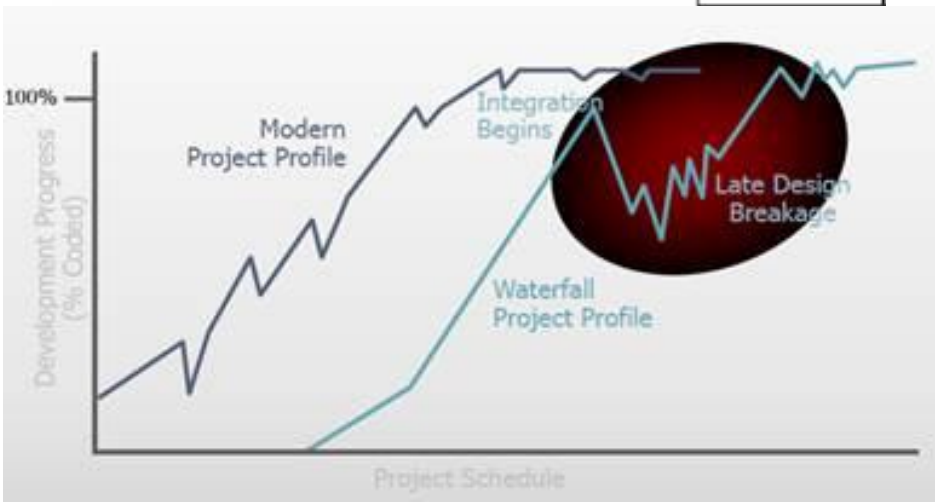
1.1.2. Sekvenční a iterativní přístup k vývoji software, výhody, nevýhody, důsledky, způsoby dodávky produktu.

Sekvenční přístup

- vztažené na **celý** produkt- *velký třesk*
- naplánované pro celý projekt, nebo pro oddělené meziprodukty
- vhodné pouze pro malé projekty s malou mírou neznáma (change management, náklady na změny, ...)
- předání až na konci celého projektu

- **Vodopádový model**
  - Životní cyklus projektu je rozdělen na 4 základní fáze:
    - Analýza požadavků a jejich specifikace
    - Návrh softwarového systému
    - Implementace (kódování)
    - Testování a udržování vytvořeného produktu
  - Následující množina činností spjatá s danou fází nemůže započít dříve než skončí předchozí





Výhody:

- snadné k pochopení
- dobrá možnost řízení a sledování postupu řešení (milníky (milestones) – voleny po činnostech)
- klade důraz na dokumentaci - specifikace, design, analýza

Nevýhody:

- vyžaduje mít na počátku přesně a úplně definované požadavky (uživatel často nedokáže stanovit předem)
- provozuschopnost verze vidí zákazník až v závěrečných fázích řešení, případné závažné nedostatky jsou odhaleny velmi pozdě.
- během vývoje se mohou měnit požadavky a výsledkem je, že dodaný produkt není to, co zákazník chtěl
- během implementace se zjistí, že design není v pořádku a je třeba ho změnit

Iterativní přístup

- Vývoj rozdělen na malé části, miniaturní úplné projekty s cca vodopádovým modelem, tzv. Iterace. (Charakteristika a vlastnosti / průběh iterace viz. Otázka 4.)
- Inkrementální rozšiřování produktu z původní hrubé formy do výsledné podoby -> Umožňuje postupné upřesňování požadavků na cílový produkt
- Řízen riziky a prioritami uživatele na funkčnost produktu
- Zaměření na architekturu produktu
- Požadavky mají zásadní vliv na návrh a implementaci produktu
- Řízení a plánování iterativně vedeného softwarového projektu viz. Otázka 5.

Výhody:

- Menší časové úseky v dodávkách produktu -> zvýšení úspěšnosti produktu díky zpětné vazbě
- Snížení rizik
- Snazší řízení změn na základě zpětné vazby uživatele
- Vyšší míra znovuvyužitelnosti
- Projektový tým se může učit během procesu vývoje
- Vyšší kvalita produktu

Nevýhody:

- Průběžné změny mohou způsobit porušení původní systémové struktury což vede k náročnější údržbě softwaru
- Vyžaduje přísnější management

Způsoby dodávky produktu

Velký třesk

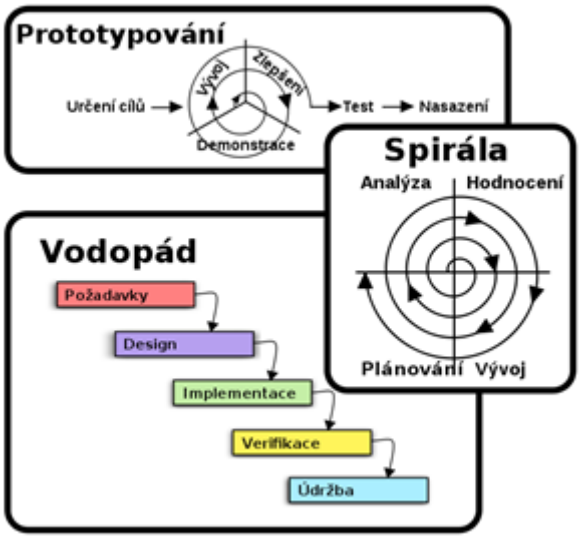
- Jednorázová dodávka hotového produktu
- malé projekty, jasné požadavky

Přírůstkově

- určení přírůstků -> plán -> postupné dodávky
- zpětná vazba, ale úpravy projektu obtížné

Evolučně

- cyklus: určení cíle -> dodávka -> zpřesnění ("growing sw")



1.1.3. Základní charakteristiky iterativních a agilních metodik.

[aswi 01b-iterativni.pdf, 02a-zahajeni.pdf]

Průběh iterace

1. Plánování cíle iterace (funkčnost)
2. Doplnění a zpřesnění požadavků (základ: plán projektu, vize, předchozí feedback)
3. Dotváření návrhu
4. Implementace přírůstku funkčností
5. Integrace přírůstku (ověření, otestování)
6. Předání do provozu (validace zákazníkem)
7. Zhodnocení

Počet a pravidla iterací

Počet: zavisí na charakteru projektu a fázích vývoje, obvykle alespoň 3 celkem

Pevné datum ukončení: plánováno nejpozději na začátku iterace

Běžící iterace uzavřena změnám zvenčí (nutné pro stabilitu projektu), potřebuje dobré změnové a projektové řízení. Zdroje tlaku na změnu: *čas, funkčnost, postup*

Délka iterace

Krátká je lepší – blízký cíl, menší složitost/riziko, rychlá adaptace, vysoká produktivita

- 1 - 4 týdny pro malé, 3 - 6 týdnů velké projekty, zřídka měsíce

Vždy pevné datum ukončení

Timeboxované iterace = délka známa předem

Milníky

1. **LCO** (Lifecycle Objectives) – definování terče – vize produktu
2. **LCA** (Lifecycle Architecture) - určení způsobu řešení – Architektura technického řešení
3. **IOC** (Initial Operational Capability) – schopnost efektivně „vyrobit“ řešení – beta verze, all features, unit a funkční testy
4. **GA** (General Availability) – uvést produkt do rutinního provozu – „krabice“ s produktem website launch

Charakter iterací dle fáze

Základní schéma pevné, mění se činnosti a artefakty

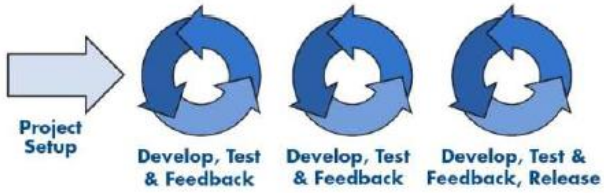
1. **Zahájení** – analytické činnosti, validace vize zákazníkem (1-2 iterace)
2. **Projektování** – analytické a designérské činnosti, ověřování prototypy, implementace (2+ iterací)
3. **Konstrukce** – designérské a programátorské činnosti, změnové řízení, testování a ověřování (N iterací)

Nasazení – integrační a konzultační činnosti, ověřování provozem, náběh uživatelské podpory (1-2 iterace)

1.1.4. **Vlastnosti iterace, její průběh.**

Iterace →

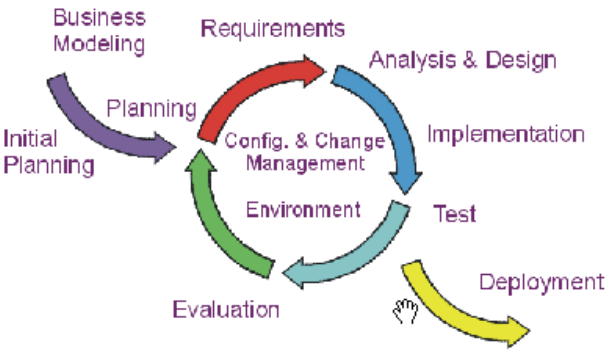
- miniaturní úplný projekt s cca vodopádovým modelem, prolínání aktivit.
- cílem je iterační release (otestovaný, funkční ale funkčně neúplný produkt).
- cyklické opakování: Develop, Test, Feedback (pro celý projekt viz obr. níže)



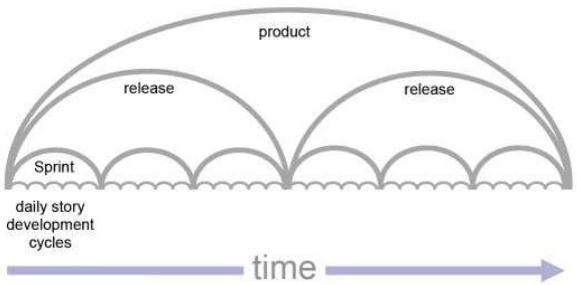
- min 3 iterace na projekt (závisí na projektu, velikosti týmu,…)
- datum konce iterace se volí vždy na začátku – *timeboxované iterace* (XP 2 týdny, SCRUM 30 dní)
- iterace je uzavřená změnám zvenčí (pro stabilitu projektu)
- když se nestíhá je možné omezení plánované funkčnosti, ale ne přesčasy, nehotový release, měnit datum
- předávání po částech (konce iterace) → artefakty, demo, retrospektiva (!)
- každá iterace končí vytvořením spustitelného kódu

Průběh iterace

- Plánování cíle iterace (funkčnost)
- Doplnění / zpřesnění požadavků
  - Základ: plán projektu, vize, předchozí feedback
- Dotváření návrhu
- Implementace přírůstku funkčnosti
- Integrace přírůstku
  - Ověření, otestování
- Předání do provozu
  - Validace zákazníkem
- Zhodnocení



Kontext iterace v procesu vývoje



(pozn.: sprint = iterace, resp. Sprint je iterace v terminologii SCRUMu)

1.1.5. Plánování a řízení iterativně vedeného softwarového projektu.

Globální řízení iterativního projektu

- Výchozí bod: **vize** produktu
- Oddělené sekvenční **fáze** reprezentující „klasické“ inženýrské disciplíny. Každá fáze má jasné rozdělení cílů a výsledků. Skládá se z 1 až N iterací.

Přehled a cíle fází

Zahájení (Inception)

- Nápad → poptávka → nabídka → zformování vize || kontrakt → zahájení projektu
- „To achieve concurrence among all stakeholders on the lifecycle objectives for the project“
    - **vize produktu** – co se má vytvořit
    - **business case** – zdůvodnění, že se to vyplatí
    - **technický koncept** – ověření proveditelnosti
- (1 – 2 iterace), analytické činnosti, validace vize zákazníkem
- Končí milníkem **LCO**

Projektování (Elaboration)

- Zahájený projekt (vize) → sběr a analýza požadavků → návrh architektury technického řešení → ověření návrhu → příprava na vývoj
- „To baseline the architecture of the system to provide a stable basis for … the design and implementation effort “
    - **rozumně kompletní DSP** – všechny klíčové/kritické požadavky
    - **architektura** – základní rysy technického řešení
    - **infrastruktura** – prostředí pro realizaci
- Architecture shall be executable
- (2+ iterace), analytické a designérské činnosti, ověřování prototypy, implementace
- Končí milníkem **LCA**

Konstrukce (Construction)

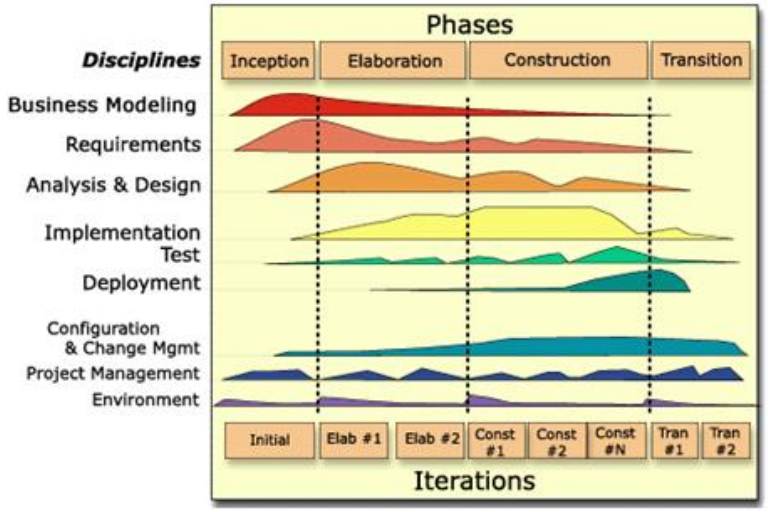
- (Známý cíl a technická architektura) → dovyrobit produkt + ujasnit přitom zbylé/příslušné požadavky
- „In some sense a manufacturing process, emphasis is on managing resources and optimizing costs, schedules, and quality.“
    - dosáhnout nasaditelných verzí v dobré kvalitě co nejrychleji
    - připravit nasazení do provozu (produkt, prostředí, uživatelé)
- (N iterací), designérské a programátorské činnosti, změnové řízení, testování a ověřování
- Končí milníkem **IOC**

Předání (Transition)

- (Hotová beta verze produktu) → dát produkt k dispozici uživatelům
  - … triviální až extrémně složitá fáze

- „Fine tuning“ a dodávka
  - příprava release
  - field testing a korekce
  - konfigurace, použitelnost; pokud funkčnost pak je problém
  - konverze dat, příprava překlopení (cutover)
  - školení uživatelů, podklady marketingu a výrobě
  - akceptace, zhodnocení produktu dle vize

- „By the end of the Transition Phase, lifecycle objectives should have been met“ → uzávěrka projektu
- (1 – 2 iterace), Integrační a konzultační činnosti, ověřování provozem, náběh uživatelské podpory



Globální plánování:

Milníky voleny na základě stupních přesnosti (produktu) a míře rizika

Milníky:

1. LCO (Lifecycle Objectives)

- srozumění s rozsahem, cenou, harmonogramem
- souhlas s požadavky a jejich klíčovostí
- navrhovaný postup vývoje souhlasí
- rizika identifikována a řešení známo

Artefakty

- Vize produktu, Business case
- Seznam rizik a strategie jejich řešení
- Slovník pojmů a přehled klíčových požadavků
- Koncept technického řešení (architektura + prototypy)
- Plán projektu
- Popis procesu a infrastruktury

2. LCA (Lifecycle Architecture)

- Vize a klíčové požadavky jsou stabilní
- testy ověřily, že architektura řeší rizikové požadavky/faktory
- jsou přesnější odhady pracnosti, na nich postavené plány
- nástroje a postupy pro realizaci jsou v provozu
- stakeholders: vize realizovatelná, spotřebované zdroje adekvátní

Artefakty

- Vize produktu (aktualizace), Specifikace požadavků
- Seznam rizik a strategie jejich řešení (aktualizace)
- Popis architektury, validační testy
- Plán projektu, Popis infrastruktury

3. IOC (Initial Operational Capability)

- Je hotová „beta“ verze produktu
- Je hotová první verze plánu nasazení
- Implementace je dokumentovaná, existují používané testy
- Je rozpracována uživatelská dokumentace
- Jsou aktualizovány popisy návrhu, datového modelu, požadavků

Artefakty

- Plán nasazení (první verze)
- Testovací sady + reporty
- Architektura (aktualizovaná), popisy implementace
- Uživatelská příručka a podpůrné materiály (draft)

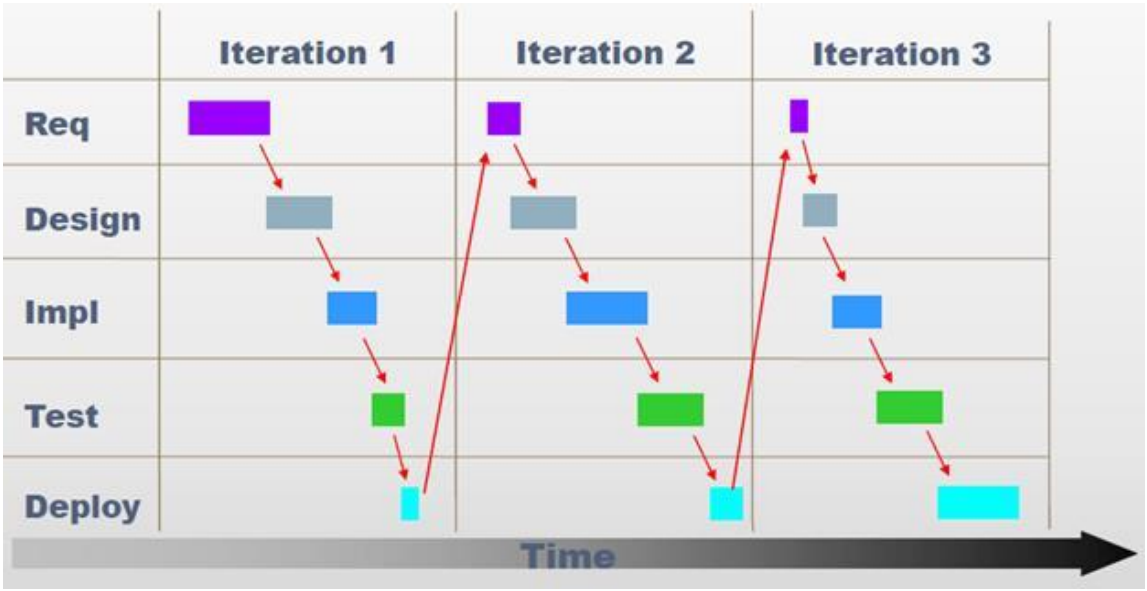
4. GA (Product Release)

- „Is the result of the customer reviewing and accepting the project deliverables“
- Uživatel je spokojen s produktem
- Stakeholders jsou spokojeni s projektem (čas, peníze)
- Uvedení produktu do reálného provozu

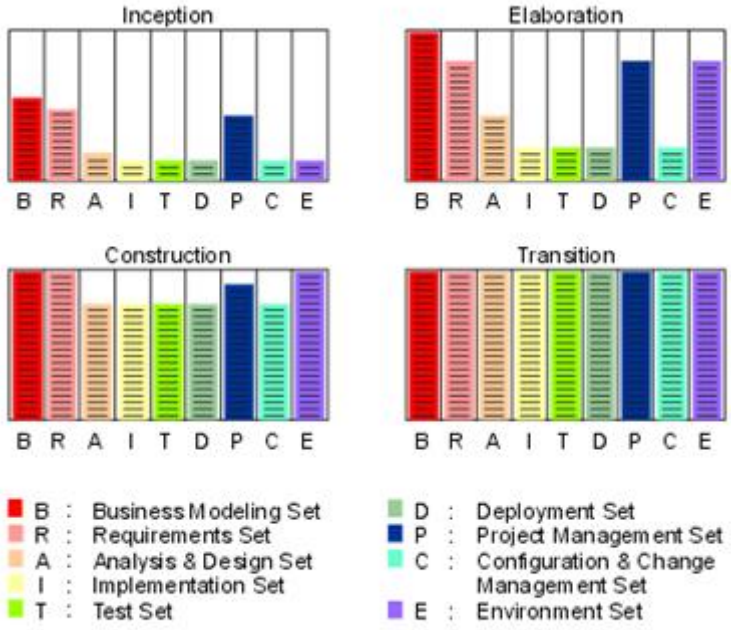
Artefakty

- Release produktu
- Podpůrné materiály („uživatelská dokumentace“)
- Baseline kompletní konfigurace release
- … dle povahy produktu

Charakter iterací dle fáze



Artefakty dle fáze:



Information set evolution over the development phases.

1.1.6. Požadavky na software – typy požadavků, formy popisu, úrovně detailu a jejich vztah k procesu vývoje.

Co je to požadavek?

- ***požadavek* = schopnost nebo vlastnost**, kterou má sw mít, aby jej uživatel mohl použít k vyřešení problému nebo dosažení cíle, který vedl k zadání, nebo aby splnil podmínky stanovené smlouvou, standardem nebo jinou specifikací.
- vlastnosti požadavku: **úplný, bezesporný**
- požadavkem není to, co uživatel nepotřebuje

Typy požadavků

- **Funkční požadavky = funkce**
  - popisují funkce nebo služby, které jsou od systému očekávány
  - příklady: požadavky na univerzitní knihovní systém
- **Mimofunkční požadavky = vlastnosti**
  - netýkají se funkcí systému, ale vlastností jako je spolehlivost, čas odpovědi, obsazené místo na disku nebo v paměti, aj.
  - často kritičtější než jednotlivé funkční požadavky (např. pokud je řídící systém letadla nespolehlivý, je nepoužitelný)
  - někdy dané vnějšími faktory, tj. legislativní požadavky (př. zákon na ochranu osobních údajů apod.)
  - př. veškerá komunikace mezi uživatelem a systémem by měla být vyjádřitelná ve znakové sadě ISO 8859-2
- **Business požadavky**
  - Vize a rozsah projektu
  - Smluvní záležitosti
  - Náklady, TCO
- **Právní a další**
  - Např. různé právní systémy dvou zemí a cloud

Způsob formulace požadavku

- uživatelská specifikace
- vysokoúrovňový popis funkčních a mimofunkčních požadavků zákazníka
- musí být srozumitelné pro uživatele, kteří nemají technické znalosti
- systémová specifikace
- podrobnější specifikace uživatelských požadavků pro vývojáře
- slouží jako výchozí bod pro design systému

Formy popisu

- textový popis
- shopping list
- strukturovaný text
- grafické vyjádření
- use case diagramy
- ERA, UML
- implementace
- popis ve formě prototypu a uživatelské příručky

Úrovně detailu v rámci procesu vývoje

- **Zahájení projektu:** strategické, klíčové, obrysy
- **Projektování:** podstatné, úplnost
- **Konstrukce:** podrobnosti

Konkrétněji:

Fáze zahájení projektu

- přesně cíl/vize projektu
- seznam klíčových aktérů, jejich cíle
- seznam/diagram podstatných funkčností (dle cíle)
- stručný popis klíčových funkčností, znalost klíčových vlastností

Fáze projektování

- kompletní seznam aktérů, popis důležitých
- kompletní specifikace, 80 - 100% funkčnosti
- přesné popisy důležitých funkčností, stručné povědomí u všech
- přesný popis mimo funkčních vlastností

Úroveň detailu a agilní metodiky

- Cíl: zachytit věci v danou chvíli nejpodstatnější (viz Manifest), detaily se dohodnou až bude potřeba
  - Zákazník musí dát podklad pro odhad a plánování
- S každou iterací zpřesnění
  - User stories
  - Tasky v backlogu
  - Jednotlivé úkoly

Dokument specifikace požadavků (DSP)

- konečný výsledek analýzy požadavků
- kompletní popis chování systému
- zahrnuje případy užití popisující všechny interakce uživatele se SW -- funkční požadavky
- technický dokument, oficiální vyjádření o tom, co se od vyvíjeného systému očekává (dohoda mezi zákazníkem a dodavatelem, co má zadaný sw dělat a jak to má vypadat)
- základ pro pozdější ověření správnosti - důraz na jednoznačnost, ověřitelnost, reálnost, srozumitelnost, úplnost, přesnost a správnost, modifikovatelnost, konzistenci
- měl by specifikovat pouze externí chování systému, tj. snaha vyloučit design z DSP
- strukturován tak, aby v něm bylo snadné provádět změny (modifikovatelnost)
- měl by specifikovat omezení implementace – mimofunkční požadavky
- měl by charakterizovat přijatelné odpovědi na nežádoucí události

Jednodušeji:

Jednoznačnost, úplnost, srozumitelnost, modifikovatelnost, přesnost, ověřitelnost, reálnost, specifikace pouze chování - NE jak to udělat

1.1.7. **Postupy pro sběr požadavků.**

Problematika získávání požadavků

- Uživatelé nerozumí tomu, co chtějí, nebo uživatelé nemají jasnou představu o svých požadavcích
- Uživatelé neschválí seznam sepsaných požadavku jako finální
- Uživatelé trvají na nových požadavcích i po zafixovaní nákladů a časového harmonogramu
- Komunikace s uživateli je pomalá
- Uživatelé se často nepodílejí na kontrolách, nebo jsou neschopní to udělat
- Uživatelé jsou technicky nevzdělaní
- Uživatelé nerozumí procesu vývoje
- Uživatelé neví o současné technologii
- Je potřeba předem počítat s různou úrovní počítačové gramotnosti. To může vést k situaci, kdy uživatelé průběžně mění své požadavky, i když systém nebo vývoj produktu byl zahájen.

Způsoby sběru požadavků

- **Neinteraktivní**
    - *analýza existujícího systému*
        - inspirujme se tím, jak funguje stávající systém
    - Studium dokumentace
    - Hlášení problémů
    - Analýza trhu
    - Konkurenční systémy

- **Interaktivní**
    - *interview*
        - předem připravený rozhovor, který vede moderátor (klade otázky, dává slovo)
        - nedoporučuje se více než 2 hodiny
        - předem si připravit scénář, které okruhy se budou probírat, v jakém pořadí, scénář se snažit nenásilně dodržovat
    - *pozorování, práce s uživateli*
        - pozorování prací u zákazníka (účast analytiků)
    - *dotazníky*
        - vhodnými otázkami zjistíme od uživatelů, co potřebují
    - *prototypování* – tvorba prototypů, podle kterých si zákazník ujasní své požadavky
        - stačí na papír nebo skutečné programové prototypy
    - *studium hlášení problémů*

Způsoby vyjádření

- *přirozený jazyk*
    - výhodou je srozumitelnost pro uživatele
    - nevýhodou – spoléhá se na to, že autoři používají stejná slova pro stejný koncept (stejná věc se dá říci mnoha různými způsoby). Obtížná modularizace - kterých všech dalších požadavků se změna dotkne.

- *formuláře*
    - pro vyjádření požadavku se nadefinuje jeden nebo více typů formulářů
    - měl by obsahovat:
        - popis specifikované funkce nebo entity
        - popis vstupů, odkud se berou
        - popis výstupů, kam putují
        - jaké další entity specifikovaná funkce nebo entita používá
        - případné pre/post conditions (co platí při vstupu do funkce a co při výstupu z ní)
        - pokud vznikají postranní efekty, pak jejich popis
- *pseudokódy*
    - v přirozeném jazyce těžko vyjádřitelné vnořené podmínky nebo smyčky
    - jazyk s abstraktními konstrukcemi, které právě potřebujeme
    - vnoření konstrukcí je vyjádřeno odsazením
    - vyhýbáme se syntaktickým konstrukcím cílového programovacího jazyka (popisujeme požadovaný záměr, nikoli jak to bude v cílovém jazyce)
    - na druhou stranu musí umožňovat téměř automatickou konverzi do kódu
- *Obrázky, protoyp GUI*

Kontrola požadavků

- musíme zjistit, zda jsou požadavky úplné, konzistentní a zda odpovídají tomu, co zadavatel chce
- vstupem je úplný **Dokument specifikace požadavků**
- metody:
- přezkoumání (reviews) – požadavky jsou systematicky kontrolovány týmem, manuální proces
- prototypování – zákazníkovi předvedeme spustitelný model systému
- generování testovacích případů – vytvoříme testy požadavků, pokud je obtížné vytvořit test, bude požadavek obtížně implementovatelný
- automatická analýza konzistence – pokud byly požadavky specifikovány jako model ve formální nebo strukturované notaci

Management požadavků

- požadavky na systém se stále mění
- měl by začít plánováním, ve kterém se rozhodne:
    - **způsob identifikace požadavků** – každý požadavek by měl mít jednoznačné ID
    - **proces změny požadavků** – definujeme proces, abychom se ke změnám požadavků chovali konzistentním způsobem
    - **sledovatelnost**
        - zdroj požadavku – kdo požadavek navrhnul, důvod; abychom se mohli zdroje zeptat na podrobnosti
        - vztahy mezi požadavky – kolika požadavků se změna dotkne
    - **nástroje** – co se použije pro uchování informací o požadavcích (malé projekty – obvyklé prostředky(textové nástroje, EXCEL, databáze, aj), velké projekty – CASE nástroje)

1.1.8. Analýza požadavků a tvorba objektového návrhu – postup, použité modely a diagramy.

Analýza požadavků obsahuje tři typy aktivit:

- **Sběr požadavků:** komunikace se zákazníky a uživateli za účelem získání jejich požadavků na systém.
- **Analýza požadavků:** identifikování nejasných požadavků, nekompletních, nejasných, nebo protichůdných a následně řešení těchto nesrovnalostí.
- **Zaznamenání požadavků:** dokumentování požadavků v různých formách, jako běžný textový dokument, případy užití (use case), nebo specifikace procesů → dokument specifikace požadavků

Analýza a klasifikace požadavků

- Identifikace zúčastněných stran ("Stakeholderů")
- Případy užití (Use Case)
- XP (Extrémní programování) – user stories
- Funkční
- Mimofunkční
    - Výkonnostní
    - Designové
    - Zákonný rámec

Metoda FURPS – model klasifikace funkčních a mimofunkčních požadavků

- F (**functionality**) – funkčnost
- U (**usability**) – užitečnost
- R (**reliability**) – spolehlivost
- P (**performace**) – výkon
- S (**supportability**) – rozšiřitelnost

Podle úhlu pohledu

- **Stakeholder**
    - **Sponsor**
    - **Uživatel**
    - **Oponent**
- **Vývojář**

Další metody analýzy

Agilní metodiky

Agilní metodiky nevyžadují potřebu přesného popisu, kategorizace a charakterizace požadavků. Vývoj softwaru považují za pohyblivý cíl. Uživatelské příběhy (user stories) a akceptační testy

Diagram aktivit (UML)

- **akce** – atomické dále nedělitelné kroky
- **vnořené aktivity** – volání jiných procesů (aktivit), tyto aktivity mohou být reprezentovány dalším **diagramem aktivit**.
  Sekvenci jednotlivých kroků v diagramu aktivit určuje řídicí tok.

Strukturální model

- Postup se zaměřuje na data a jejich transformaci pomocí procesů systému,
- hlavními nástroji jsou tedy DFD (Data Flow Diagram) popisující **procesy a toky dat** a ERD (Entity Relationship Diagram) popisující **data a vztahy mezi nimi**
- Zaměřuje se na vytvoření logického modelu nového navrhovaného systému (**esenciální model**) a následně přizpůsobení implementačním požadavkům (**implementační model**).

- Model **prostředí**
  - Definuje hranice systému a okolí
  - Obsahuje kontextový diagram a seznam událostí (event list)

- Model **chování**
  - Definuje vnitřní chování systému, tak aby plnil požadavky okolí
  - Používané modely (diagramy) DFD, ERD, DD (datový slovník), SP (specifikace procesů), případně STD (stavový diagram)

Objektový model

- Často se používá **modelovací jazyk UML**
  - Model případů užití
  - Doménový model
  - Diagram tříd
  - Stavové diagramy, sekvenční diagramy a další

- **CRC karty** (Class-Responsibility-Collaboration cards)
  - Umožňuje zvládnout návrh i složitých a velkých systémů
  - Není na první pohled vidět kudy vedou vztahy, musí se vyčíst z karet (barevné rozlišení, čáry na nástěnce…)

- Hierarchie objektů – sdružovány podle logických souvislostí do balíků a podbalíků
- Přirozený přechod od analýzy k návrhu

Doménová analýza

- Doménová analýza nepřihlíží podstatně k jednomu účelu a způsobu použití (kontextu použití), nýbrž shromažďuje pojmy a vazby pro doménu podstatné.
- Hledají se objekty, operace a vazby, které znalci z problémové oblasti pokládají za důležité (často používají jejich názvy apod.)

1.1.9. Architektura softwarových systémů, význam a součásti architektury, formy popisu architektury, architektonické styly.

Význam

- Architektura definuje konceptuální integritu systému.
- Systém má vždy právě jednu architekturu (může integrovat více stylů)
- Definice architektury je první krok návrhu
- Umožňuje myšlenkové pochopení návrhu velmi složitých systémů
- Stanovuje základní kameny návrhu a základní směry vývoje a údržby

Součásti

- konvence a politiky (pravidla pro návrh, dodržují všichni vývojáři)
- **Funkční, procesní, datová, aplikační**
- členění, doménová analýza:
- **logické členění** (např. do balíků)
  - balík – skupina souvisejících tříd, tvořící organizační celek, mapování do jazyka (balík vytváří jmenný prostor), hierarchické vnořování
  - třídy v balíku funkčně příbuzné
  - vhodné protože bude přehled o systému a snadné rozdělení implementace mezi členy týmu
  - analytický model tříd je příliš rozsáhlý -> lepší jej členit
- **funkční členění do subsystémů**
  - subsystém = skupina souvisejících balíků a/nebo tříd tvořící funkční celek
  - vhodné, protože monolitická aplikace není praktická
  - jak najít subsystémy?
    - buď je to dopředu zřejmé (jednoduché, architektonické styly)
    - na základě objektového modelu (nutno vidět všechny třídy a vazby, pak shluk těsně vázaných tříd je kandidátem)
    - na základě případů užití

Formy popisu architektury

Modely - UML, Layer diagramy, …

UML, **Unified Modeling Language** je v softwarovém inženýrství grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. UML nabízí standardní způsob zápisu jak návrhů systému včetně konceptuálních prvků jako jsou business procesy a systémové funkce, tak konkrétních prvků jako jsou příkazy programovacího jazyka, databázová schémata a znovupoužitelné programové komponenty.

UML podporuje objektově orientovaný přístup k analýze, návrhu a popisu programových systémů. UML neobsahuje způsob, jak se má používat, ani neobsahuje metodiku(y), jak analyzovat, specifikovat či navrhovat programové systémy.

Standard UML definuje standardizační skupina Object Management Group (OMG).

Architektonické styly

- **Vrstvení** - funkce jsou uspořádány do několika vrstev tak, že funkce vyšší vrstvy mohou využívat pouze funkcí podřízených vrstev.
  - **Monolitická**
  - **Dvouvrstvá (tenký/tlustý klient)**
  - **Třívrstvá** je nejběžnějším případem vícevrstvé architektury.
    - **Prezentační vrstva**
    - **Aplikační vrstva (též Business Logic)**
    - **Datová vrstva**

  - **MVC (Model-View-Controller)**
    Je softwarová architektura, která rozděluje datový model aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých component tak, že modifikace některé z nich má jen minimální vliv na ostatní.

  - **Porovnání třívrstvé s architekturou MVC**
    Model-view-controller má trojúhelníkovou topologii (ne třívrstvou) – pohled je obnovován (aktualizován) přímo modelem, na příkaz řadiče.

  Architektura distribuovaných systémů
  - Klient-server
  - Peer-to-peer

Filosofický přístup k architektuře/ jednotící architektura: **Service-Oriented Architecture (SOA)**

Příklad vrstvené architektury v síťové komunikaci:

(spíš tak na okraj)

ISO/OSI - 7 vrstev (Fyzická-Linková-Síťová-Transportní-Spojová-Presentační-Aplikační)

TCP/IP - 4 vrstvy (Network Access=Ethernet/FDDI - Internet=IP - Transportní=TCP/UDP - Aplikační)

1.1.10. Konfigurační management, jeho součásti a role ve vývoji software, základní postupy.

Konfigurační management

„Proces identifikování a definování prvků systému, řízení změn těchto prvků během životního cyklu, zaznamenávání a oznamování stavu prvků a změn, a ověřování úplnosti a správnosti prvků.“ (IEEE)

= jak vytvářet, sestavovat a vydávat produkt, identifikovat jeho části a verze, a sledovat změny

- Administrační a manažerský aspekt Softwarového procesu

Prvek konfigurace

Configurable Item (CI)

- konstituující složka systému (konfigurace se sestává z prvků konfigurace)
- jsou atomické z hlediska změn a označování verzí, jednoznačně identifikovatelné
- př.: dokument, zdrojový soubor, knihovna, skript, spustitelný soubor, testovací data, …
- Je spravován SCM - ví se o jeho existenci, vlastníkovi, změnách, umístění v produktu…
- Je atomický z hlediska identifikace, změn
- Jednoznačně identifikovatelný - např. MSW/WS/IFE/SD/01.2
    - (typ prvku, označení projektu, název prvku, identifikátor verze…)

Konfigurace

SW konfigurace – souhrn prvků konfigurace reprezentující určitou podobu daného SW systému

- V konfiguraci musí být vše, co je potřebné k jednoznačnému opakovatelnému vytvoření příslušné verze produktu (včetně překladačů, build scriptů, inicializačních dat, dokumentace)o
- Konzistentní konfigurace – konfigurace, jejíž prvky jsou navzájem bezrozporné (tj. zdrojové soubory lze přeložit, knihovny přilinkovat, …)

Role ve vývoji SW

Určení a správa konfigurace

- určení (identifikace) prvků systému, přiřazení zodpovědnosti za správu
- identifikace jednotlivých verzí prvků
- kontrolované uvolňování (release) produktu
- řízení změn produktu během jeho vývoje

Zjišťování stavu systému

- udržení informovanosti o změnách a stavu prvků
- zaznamenávání stavu prvků konfigurace a požadavků na změny
- poskytování informací o těchto stavech
- statistiky a analýzy (např. dopad změny, vývoj oprav chyb)

Správa sestavení (build) a koordinace prací

- určování postupů a nástrojů pro tvorbu spustitelné verze produktu
- ověřování úplnosti, konzistence a správnosti produktu
- koordinace spolupráce vývojářů při zpracování, zveřejňování a sestavení změn

Role ve správě změn

Change Control Board

- Skupina členů projektu, která má zodpovědnost za změnové řízení
  - Vyhodnocování a schvalování hlášení problémů
  - Rozhodování o požadavcích na změny
  - Sledování hlášení a požadavků při jejich zpracování
  - Koordinace s vedením projektu
  - Složení: jedinec - vývojář, QA osoba; tým - technické i manažerské role

Základní postupy

- Identifikace konfigurace: stanovení výchozího bodu (baseline, třeba každá major verze), známá kvalita (např. seznam bugů dané konfigurace, kompletní testování před stanovením výchozího bodu), kompletně opakovatelná
- Řízení konfigurace: rozhodování o způsobu změny konfigurace a koordinace změny konfigurace po schválení změny změnovým managementem (zm. mgmt schválí, chg. mgmt zavádí)
- Sledování stavu konfigurace: dokumentování stavu konfigurace každého vydání a změn konfigurace mezi vydáními (průběh změn mezi jednotlivými výchozími body)
- Auditování konfigurace: ověření, že nový výchozí bod implementuje všechny plánované a schválené změny, že nová verze je kompletní, a že dodávka je kompletní vč. právních opatření, dokumentace a dat.

! Role konfigurace ve vývoji vychází z postupů nebo naopak

ITIL: Konfigurační management IT infrastruktury podniku

- zaznamenávání informačních aktiv podniku, nastavení organizace a jejích služeb
- poskytování přesných informací o nastavení procesů a jejich dokumentace,
- poskytovat základ pro Incident Management, Problem Management, Change management a Release Management
- ověření konfiguračních záznamů oproti skutečnosti a jejich sladění.

Standardy

Standardů pro *Configuration management* existuje poměrně velké množství. Drtivá většina standardů je založena na metodice **ITIL**. Příklady některých standardů jsou: IEEE, ISO, ANSI, NATO standards.

1.1.11. Správa verzí, možnosti verzování, typické situace při správě verzí (větvení, značkování), nástroje pro správu verzí, vazba na správu změn.

Správa verzí (SCM)

Správa verzí je součást úlohy konfiguračního managementu – identifikace konfigurace

Účelem je udržení přehledu o podobách prvků konfigurace

- verze popisuje jednu konkrétní podobu
- v úložišti jsou skladovány všechny verze

Druhy verzí

- evoluční = revize (př. Word 6.0)
- alternativní podoba = varianta (př. Word pro Macintosh)

Určení konkrétní verze

- verzování podle stavu (verze prvku) – identifikují se pouze prvky
- verzování podle změn (identifikace změny prvku) – identifikují se také změny prvků, výsledná verze prvku vznikne aplikací změn

Granularita

- Jednotlivé prvky (verzování komponent)
  - Konfigurace nemá verzi
- Celé konfigurace (úplné verzování)
  - Verze konfigurace indikuje verze prvků
- Verze produktu

Popis verze

- extenzionální verzování: každá verze má jednoznačné ID
  - major.minor + build schéma- např. 6.0.2800.1106 (MSIE 6)
  - kódové jméno: One Tree Hill (= Firefox 0.9)
  - marketingový: Windows 95
- intenzionální verzování: verze je popsána souborem atributů
  - např. OS=DOS and UmiPostscript = YES
  - C preprocesor umožňuje intenzionální stavové verzování - např. chceme variantu foo.c pro případ OS=DOS and UmiPostscript=YES

Možnosti verzování

- Rychlý přístup k jakékoli historické nebo alternativní verzi
- Možnost vytvoření branch, tagu => částečná izolace ale s možností aplikace vývoje v trunku
- Pojmenovávání milestounů
- Celý tým má přístup k aktuálnímu stavu vývoje
- Aktuální stav vývoje je jednoznačně určen
- Soukromý pracovní prostor v rámci nejnovější nebo vybrané verze
- Možnost testování lokální změny a commitu až funkční a otestované součásti

- Možnosti pro řešení konfliktů
- Některé verzovací systémy jsou inherentně zálohovací (GIT)

Informace o verzi

- Identifikátor verze (extenzionální) - klíčovým požadavkem je jedinečnost
- "major.minor.micro + build" - např. 6.0.2800.1106 (MSIE6)
- Záleží na použitém nástroji a vztahuje se na prvky konfigurace
- Marketingové jméno se pak dává celému produktu (Longhorn)
- Další meta-data prvku jsou datum/čas vytvoření, autor, stav prvku/konfigurace, předchůdci…

Prostředí pro verzování: úložiště

- Úložiště (databáze projektu, repository) = sdílený datový prostor, kde jsou uloženy všechny prvky konfigurace projektu
  - Zdrojové kódy
  - Knihovny (přeložené) a kód třetích stran
  - Konfigurační soubory, datové soubory
  - Scripty pro build, testování a instalace
  - Dokumentace, modely, prototypy
  - Odpadkový koš
- Řízený přístup (udržení konzistence)

Práce s úložištěm

- Základní operace
  - Inicializace - vytvoření úložiště, naplnění bootstrap verzí projektu
  - Check-out - kopie prvku do lokálního pracovního prostoru
  - Check in (commit) - uložení změněných prvků do úložiště
  - Zjišťování stavu - sledování změn z úložišti versus v pracovním prostoru
- Přístup k zamykání při check in/check out
  - Read-only pro všechny
  - Pesimistický: read-write kopie prvku jen pro pověřeného
  - Optimistický: read-write pro kohokoli, řešené konfliktů

Pracovní prostor

- Workspace = soukromý datový prostor, v němž je možno provádět změny prvků konfigurace, aniž by byla ovlivněna jejich oficiální podoba používaná ostatními vývojáři

Typické situace při správě verzí

Trunk: Hlavní vývojová větev

Branch: Větve – „soukromý“ vývojový prostor

Merge: Sloučení větve a do kmene a řešení konfliktů

Tag: Značkování – označování milestounů, release apod.

Kolize a konflikty – diff, sloučení dvou konfliktních commitů…

Postup vývoje a verzování

- Check out výchozí verze
- Vývoj
- Lokální testování a opravy
- Check in nové verze
- Integrační testy a opravy
- Check in nové baseline

Codeline (vývojová linie)

- Je to série podob (verzí) množiny prvků konfigurace tak, jak se mění v čase
- Má přiřazena pravidla práce s codeline (kdy a jak je možno provádět změny…)
- Vrchol codeline obsahuje nejčerstvější verzi (head)

Tag (label)

- Označení konfigurace symbolickým jménem

Baseline

- Konzistentní konfigurace tvořící stabilní základ pro produkční verzi nebo další vývoj
    - Příklad: milník "stabilní architektura", beta verze aplikace
    - Stabilní: vytvořená, otestovaná a schválená managementem
    - Změny prvků baseline jen podle schváleného postupu
    - Při problémech návrat k baseline

Paralelní práce na stejné konfiguraci

- Důvod: velké úpravy, release, spekulativní vývoj, varianty…
- Cíl: vzájemná izolace paralelních prací tak, aby ukládané změny během nich neovlivnily ostatní (oddělení paralelních vývojových linií) -> cena za to je následné řešení konfliktů

Delta, diff

- Delta = množina změn prvku konfigurace mezi dvěma po sobě následujícími verzemi
- V někerých systémech jednoznačně identifikovatelná
- Changeset: delta + důvod
- Diff a patch = rozdíl mezi verzemi (text, binární), aplikace rozdílu na verzi - viz changeset

Větvení a spojování

- Kmen (trunk, master) - hlavní vývojová linie
- Větev (brach) - paralelní vývojová linie - operace vytvoření větve = branch off, split
- Spojení (merge) - sloučení změn na větvi od kmene
    - Slučuje se delta od branch off nebo posledního merge
    - Řešení konfliktů: automatizace vhodná, ale ne vždy možná
    - 2-way a 3-way merge

Distribuovaný vývoj

- Geograficky distribuovaný tým, v čase distribuovaný tým, offline práce se synchronizací, experimentální lokální úložiště
- Možnosti: centrální úložiště s privátními větvemi, distribuovaný verzovací systém

Nástroje pro správu verzí

Druhy verzovacích nástrojů:

- **Základní** - správa verzí souborů
  - Obvykle extenzionální verzování modulů
  - Centrální úložiště
  - Ukládání všech verzí v zapouzdřené úsporné formě
  - Příklad nástrojů: rcs, cvs, subversion…

- **Distribuované**
  - Více úložišť, synchronizace
  - Flexibilnější postupy
  - Příklad nástrojů: SVK, git, Mercurial

- **Pokročilé** - integrace do CASE
  - Obvykle kombinace extenzionálního a intenzionálního verzování
  - Automatická podpora pro check in/check out prvků z repository do nástrojů
  - Příklad nástrojů: ClearCase, Adele

Verzovací systémy:

- **Centralizované**
  - **RCS**: revision control systém
    - Pesimistický přístup
    - Pracuje s jednotlivými soubory, nepodporuje projekty
    - Historie všech změn vč. autorů
    - Ukládá rozdíly
    - Umožňuje zamykání
  - **CVS**: current versioning systém
    - Práce s celými konfiguracemi a projekty najednou
    - Optimista – slučování změn
  - **SVN**: subversion
    - Velmi podobný CVS (následník)
    - Verzuje celé úložiště (inkrementální číslování revizí)
    - Souborová struktura

- **Decentralizované**
  - Každý uživatel má kompletní lokální kopii repozitáře (klony)
  - Lokální commity, na centrální server lze nahrát víc commitů najednou
  - **GIT** – jádro linuxu
    - Velmi nelineární vývoj, recenzování a začleňování
    - Nelze měnit historické verze
  - **Mercurial** – Netbeans, OpenJDK, Symbian OS
  - **Bazaar** – Ubuntu

Co nástroj má umět

- Operace s úložištěm
- Verzování
- Podpora týmu a procesu - vzdálený přístup, konfigurovatelné zamykání a přístupová práva, automatické oznamování, spouštění scriptů při operacích, Integrace do IDE, řádkové a webové rozhraní

Rcs

- Správa verzí pro jednotlivé textové soubory
- Ukládá historii všech změn v textu souboru
  - Informace o autorovi, datu a času změn
  - Textový popis změny zadaný uživatelem
  - Další info
- Používá diff(1) pro úsporu místa - poslední revize je uložena celá, předchozí jen pomocí diff
- Funkce: zamykání souborů, symbolická jména revizí, návrat k předchozím verzím, možnost větvení a merge

CVS

- Concurrent Versioning Systém
- Práce s celými konfiguracemi (projekty) najednou
- Sdílené úložiště + soukromé pracovní prostory
- Optimistický přístup ke kontrole paralelního přístupu (zkopíruj modifikuj sluč)
- Zjišťování stavu prvků, rozdílů oproti repository
- Integrace do mnoha IDE a CASE nástrojů

Subversion

- Následník CVS
- Bez omezení předchůdce - přejmenování, verzování adresářů, atomický commit, http přístup
- Nové možnosti - binární diff, meta-data, abstraktní síťová vrstva (DAV), čisté API
- Způsob práce a příkazy velmi podobné CVS
- Identifikace verzí - globální kontinuální celočíselné identifikátory - číslují commit

Vazba na správu změn

- Vazba revize na ticket/change request
- Možnost požadavků na opravu / update konkrétních verzí (např. long-term support)

1.1.12. Typy požadavků na změny, postup jejich zpracování, nástroje pro podporu řízení změn, vazba na správu verzí.

Typy požadavků na změny

- Požadavek na novou funkci/vlastnost
- Bug

Postup zpracování změny

- vytvoření/přijetí (přidělí se ID)
- vyhodnocení (možná řešení, jejich dopady a odhad pracnosti)
- rozhodnutí
  - způsob vyřízení (vyřešit/odmítnout/duplikát/odložit)
  - závažnost (kritická chyba/problém/vada na kráse/vylepšení)
  - priorita (vyřídit okamžitě/urgentní/vysoká/střední/nízká)
- přidělení odpovědné osobě / teamu
- zpracování
- uzavření
  - build: ověření konzistence; verzování: vytvoření nové baseline
  - Informovat zadavatele hlášení a další zájemce

Nástroje pro podporu řízení změn

- Bug tracking (BT) systémy
  - evidence, archivace požadavků (Ticket systém)
  - sledování stavu požadavku (BT, Ticket systém)
  - přehled, reporty, grafy, statistiky
  - realizace: emailové, webové, klientské
  - př. **Flyspray, Redmine, Mantis**
    - Jednoduché, snadná instalace, webové rozhraní, emailová notifikace
  - Př. **Bugzilla, Jira**
    - Robustní, pro velké projekty, konfigurovatelná
- Struktura projektu v BT syst0mu
  - Název, kategorie hlášení, úrovně závažnosti, priority, verze produktu, operační systém, odhad a realita pracnosti…

Change Control Board (CCB)

- skupina členů projektu, která má zodpovědnost za změnové řízení
  - vyhodnocování a schvalování hlášení problémů
  - rozhodování o požadavcích na změny (může významně ovlivňovat podobu a chod projektu)
  - sledování hlášení a požadavků při jejich zpracování
  - koordinace s vedením projektu
- složení
  - jedinec – vývojář, QA osoba
  - tým – technické i manažerské role (vhodné, pokud má změna mít velký dopad)

Vazba na správu verzí

- Vazba ticketu/change requestu na verzi
- Vytvoření nové verze s opravou

1.1.13. Sestavení produktu, postup sestavení a jeho varianty, nástroje pro sestavení.

[aswi 04b]

Sestavení produktu = build

Řízení sestavení

Aktivity provádějící transformaci zdrojových prvků konfigurace na odvozené, zejména sestavení celého produktu

Cíl: **vytvořit systematický a automatizovaný postup**

Pojmy: **build** (**integration**, **proces sestavení**, **sestavení**)

Postup při vytváření sestavení

Build process

- míra formálnosti
- míra preciznosti

Kroky:

- příprava
- check-out
- preprocessing, překlad, linkování
- nasazení
- spuštění
- testování
- značkování, check-in
- informování

Vlastnosti sestavení

- **Jedinečnost a identifikovatelnost** – identifikátor jednoznačný, čitelný; vytvořitelný a zpracovatelný automaticky (schema pro id)
- **Úplnost** – tvoří kompletní systém, obsahuje všechny komponenty
- **Konzistence** – vzniklo ze správných verzí správných komponent – tj. konzistentní konfigurace
- **Opakovatelnost** – možnost opakovat build daného sestavení kdykoli v budoucnu (se stejným výsledkem)
- **Dodržujte pravidla vývojové linie** – build odpovídající baseline, zejména release, má striktní pravidla

- **Základní postupy** – soukromé sestavení (private systém build) + sdílení součástí, integrační sestavení (integration build), release
- **Podpůrné aktivity** – smoke test, regression test, archvace prostřední, packaging
- Obecný cíl: odchytit co nejdříve okamžik kdy „se to rozbilo“

Součásti prostředí pro sestavení

- Pravidla (neměnit) – vývojová linie, součásti a vlastnosti sestavení
- Scripty – check-out, značkování, check-in;preprocessing, překlad, linkování; nasazení, spouštění, testování; informování vývojářů, vytváření statistik; vytvoření distribuční podoby (packaging)
- Vyhrazený stroj a workspace – „build machine“

Varianty sestavení

Typy sestavení

- Co je použito pro sestavení (ušetřit čas na překladu) – čistý, úplný, přírůstkový (inkrementální) build
- Účel sestavení (lokální/neoficiální komponenty povoleny) – soukromý, integrační (oficiální), release build

Soukromé sestavení

- Cíl: ověřit si konzistenci konfigurace – produkt lze sestavit pro mnou provedených změnách, předh check-in (problémy řeším já x všichni)
- Postup: sestavit produkt v soukromém prostoru
- Urychlení průběhu – použit inkrementální sestavení tam kde je to vhodné, vynechat postupy pro balení, vkládání info o verzi, pomoci si sdílením odvozených prvků (shared version cache)

Integrační sestavení

- Cíl: spolehlivě ověřit, že produkt jde sestavit – soukromý build nestačí (složité závislosti, specifika ve workspace, zjednodušení pro zrychlení)
- Postup: **celý produkt (vč. Závislostí) sestaven centrálně, automatizovaným a opakovatelným procesem**
  - postup co nejpodobnější sestavení pro release
  - maximální automatizace – typicky běží přes noc
  - mechanismy zaznamenání chyb a informování o nich
  - úspěšné sestavení může být označkováno ve verzovacím systému

Release build

- Význačné integrační sestavení: dodáno zákazníkovi (interní zákazník, např QA)
- Náležitosti release:
  - revize/verze konfigurace použité pro sestavení
  - datum vytvoření
  - identifikátor sestavení
  - další metadata: zodpovědná osoba, zdrojová znakča konfigurace (zverzovacího systému), jakými prošlo testy (a výsledky), cesta k logům překladu (a testů)
  - „marketingová verze“ např Open cms 7.5

Diskuze o sestavení

- Celý proces automatizovat, plánovač spouštění buildu, vytváření čísel/identifikátorů sestavení, ukládání metadat do databáze a do verzování.
- Frekvence intragračního sestavení – čím častěji tím lépe – snažší nalezení chyb), kompromis trvání buildu x frekvence změn x velikost změn
- Samotné sestavení nestačí

Kusovník

- Kompletní seznam prvků sestavení
  - Reprodukovatelnost sestavení kdekoli, kdykoli
  - Zejména při distribuovaném nebo jinak složitém buildu
- Samoidentifikijící konfigurace pomůže
  - Znalost verzí bez přístupu k verzovacímu systému

Archivace prostředí

- Správa verzí objektů, které nejsou v úložišti
  - Nástroje, platformy, hardware, prostředí - identifikovat sestavení
- Klíčové pro dlouho žijící software (např. povinné v letectví)

Nejlepší praktiky: SCM + QA

- ověřené postupy sestavení pro největší zisk (zejména iterativní a přírůstkový vývoj)
  - **Statické kontroly kódu**
    - **Ověření formální správnosti + dodržování pravidel + metriky**
    - **Nástroje – překladač a jeho hlášení, C: lint, Java: pmd,findbugs**
    - **Postupy – programming by Contract, review, párové programování, automatický build- výběr**
  - **Jednotkové testy (unit testy)**
  - **Pravidla pro codeline aktivního vývoje** (active development line)
  - **Denní sestavení a zkouška těsnosti** (Daily build and smoke test)
    - Integrační sestavení + zkouška těsnosti – pravidelně 1xdenně (nočně)
    - Výsledky okamžitě reflektovány
    - Výhody: zvladatelné množství změn během denních check-in
    - Cena: trocha disciplíny, trocha automatizace
    - Smoke test = ověřit, že sestavení vytvořilo funkční produkt, vytvořit testy ověřující základní funkčnost, bez nároku na kompletní otestování
  - **Regresní testy (regression test)**
    - **Cíl: zajistit, aby nové funkce a vylepšení nesnižovaly již hotové kódu**
    - **Postup: ověřit build produktu pomocí testů, kterými již dříve prošel**
    - **Zdroj testů: chyby objevené QA, při validaci, zákazníkem**
  - **Soustavná integrace (Continuous integration)**
    - **Dotažení do dokonalosti (nebo do extrému)**
    - **Klíčová je automatizace**

Nástroje pro podporu sestavení

- **Scriptovací:**
  - shell, perl, python, php

    Buildovací:
  - make:
    - build (překlad a sestavení) projektu na základě popisu závislostí typu zdrojový - odvozený
    - makefile: definice pravidel (deklarace závislostí, příkazy pro překlad)

    Maven:
    - deklarativní build
    - popis struktury projektu
    - build „automaticky“

    ANT

    Hudson (Jenkins):
    - automatický build a průběžná integrace (vyhrazený stroj)
    - spuštění buildu
    - konfigurace buildu
    - informace

    CruiseControl

    Verifikace sestavení
  - xUnit(JUnit apod, Cactus)
  - testovací roboti

1.1.14. Způsoby prevence chyb v software, metriky a oponentury.

[http://www.robertdresler.cz/2012/02/techniky-pro-prevenci-softwarovych-chyb.htm]
[aswi přednáška 04b.pdf, aswi 06-kvalita.pdf]

Způsoby prevence chyb

- Cíl: **zabránit vzniku a dalšímu šíření chyby**
- Využití Racionální proces a best practices
- Kontroly a měření meziproduktů (častější v úvodních fázích)+
  - **Automatizované testy**
    - Základní kontrola kvality kódu
      - Typicky unit testy
  - **Prověření meziproduktu nezávislým oponentem** (dříve než se z něj začne vycházet v další práci)
  - **Technická oponentura a podobné techniky (Faganovská inspekce)**
    - Viz oponentury.
  - **Párové programování, refactoring**
    - Párové programování
      - Metafora řidič (udává směr, vysvětluje, naslouchá) + navigátor (dohledává, kontroluje, pomáhá) - svědomí páru (společný cíl, plné nasazení, komunikace)
    - Refactoring
      - Změna interní struktury software, která jej činí srozumitelnějším a snáze upravitelným, aniž by změnila jeho vnější chování
      - Detekce “zapáchajícího” kódu
      - Změna designu, oprava
  - **Strukturované procházení**
    - Podobné Faganovské inspekci, menší důraz na formálnost
  - **Peer review**
    - Kontrola nezaujatým čtenářem
    - Autor prochází kód a vysvětluje
    - Kolega hledá problémy a komentuje
- Měření
  - Kvantitativní ukazatele pomáhají najít slabiny kvality
  - Přesnost a dokazatelnost, možnost statistik
  - GQM přístup, FURPS

• Detekční a opravné techniky

- Cíl: najít a opravit již existující chybu
- Testování a ladění (typické v koncových fázích, tzv. výstupní kontrola)

• Psaní čistého kódu

- snižuje riziko "ukrytí" zákeřných programových chyb už v prvotní fázi psaní kódu
- podporuje efektivnější ladění

• Refaktorizace kódu

= změna struktury kódu, která nemá vliv na jeho celkovou funkčnost (přejmenování proměnné/metody, vyjmutí kódu do samostatné metody,..)

- snížení složitosti kódu
- Zpřehlednění

• Revize kódu (Code Review)

- Prováděna ideálně během nebo těsně po vlastní implementaci
- Revize kódu je jedním ze základních aspektů párového programování (Pair Programming)
- Efektivnost závisí na zkušenostech vývojáře a "revizora"
- Začínající vývojář (revizorem zkušený pracovník) X zkušený vývojář (revizorem posluchač získavající zkušenosti a praktické rady)
- Jsou různé techniky revizí kódu, liší se svojí formálností, obsazením revizního týmu a způsobem evidence nalezených defektů

• Statická analýza (kontrola) kódu

= analýza kódu, která je prováděna bez nutnosti spouštění programu – soubor preventivních technik. Ověření formální správnosti + dodržování pravidel + metriky

- Nástroje: překladač a jeho hlášení
- Postupy: párové programování

• Dynamická analýza kódu

• Volba technologií

• Automatizované testování

- Dynamická analýza kódu
- Podle TDD (Test-Driven Development) by mělo předcházet vlastní implementaci
- Testy jsou indicatory chybových stavů
- Unit testy
    - Volání metod instance testované třídy s určitými vstupy a validace výstupů
    - Pokud výstupy neodpovídají předpokladům, test selže a je nutno hledat chybu v implementaci.
    - Musí být správně cílené na problémové situace
    - White-box testování – test "vidí" do implmentace
        - Pokrytí cest provádění
    - Black-box testování – testovaný kód není pro test přístupný
        - zosobňuje naivní přístup klientské strany, který může přinést nečekané způsoby volání
- Integrační a systémové testy
    - Validují interakci vice objektů a funkcionalitu větších celků
- Testy by měly být vzájemně nezávislé

- **Ruční testování**
- **Automatizace buildů**
- **Prototypování**

Prototyp je funkčně zjednodušený základ (předobraz, demoverze) vyvíjeného systému. Měl by vzniknout relativně rychle a slouží k průběžné revizi požadavků. Prototyp můžete předvést investorovi a získat zpětnou vazbu. Prototyp je vyvíjen v cyklech. V každém cyklu jsou zapracovány získané připomínky.

- **Revize výstupů v předimplementačních fázích**
- **Motivace lidí**

Metriky

Klíčový pojem: fitness for purpose: vhodnost pro zamýšlený účel použití

➔ primární měřítko jsou uživatelské testy

Softwarové metriky:

- **Cyclomatic Complexity** = složitost části programu (např. metody) co do množství větvení a cyklů. Tyto programové konstrukce zvyšují počet možných cest provádění programu. Vysoké číslo indikuje komplikovaně napsané složité metody, které je problematické pokrývat testy. Řešením je metodu rozbít do více menších metod.
- **Line Count** (SLOC) vyjadřuje množství řádků kódu v metodách. Dlouhé metody jsou nepřehledné a dělají zřejmě více než jednu věc (pro danou úroveň abstrakce). Použitelnost takových metod je problematická, stejně jako pokrytí testy. Řešením je opět refaktorizace do více metod.
- ***Objektové metriky*** jako **Weighted Method Count** (celková složitost metod ve třídě), **Depth of Inheritance Tree** (počet předků třídy) a **Coupling Between Objects** (provázanost mezi objekty) mohou indikovat problémy v objektovém návrhu.

Z přednášek aswi 06-kvalita.pdf:

Formální verifikace

- Matematické důkazy správnosti
  - Návrhu
  - Implementace
- Model checking
  - Formální model systému - Pteriho sítě, algebry (CSP) (a-priory nebo získaný analýzou kódu)
  - Model checker -> deadlock-free, liveliness, …
  - Soulad s implementací

Statistické kontroly

- Základ: metriky
  - Indikátor někde je něco špatně
- Stanovení očekávaných/správných hodnot
- Průběžné monitorování
- Korekce procesu, komponent při odchylkách

Z přednášek aswi 05-metriky.pdf:

Proč měřit

- Kvantitativní ukazatele
  - Pomáhají najít slabiny -> zlepšení kvality, přesnosti, efektivity…
  - Dávají přehled a kontrolu nad projektem-produktem - plán, kvalita, splnění požadavků…
  - Kalibrují odhady
- Výhody
  - Přesnost a dokazatelnost
  - Možnost statistik a vizuální prezentace

Metrika, měření

- Metrika = měřitelná charakteristika nějaké entity
- Je získána na základě dat (primitivních metrik)
- Měřený objekt - entita: produkt nebo proces
- Metriky samy o sobě "k ničemu" -> měření
- Plán měření - pro projekt
  - Co měřit, proč měřit, jak měřit
  - Jak s daty pracovat
- Organizational focus - záměr zlepšovat kvalitu
  - Vede k potřebě mít informace

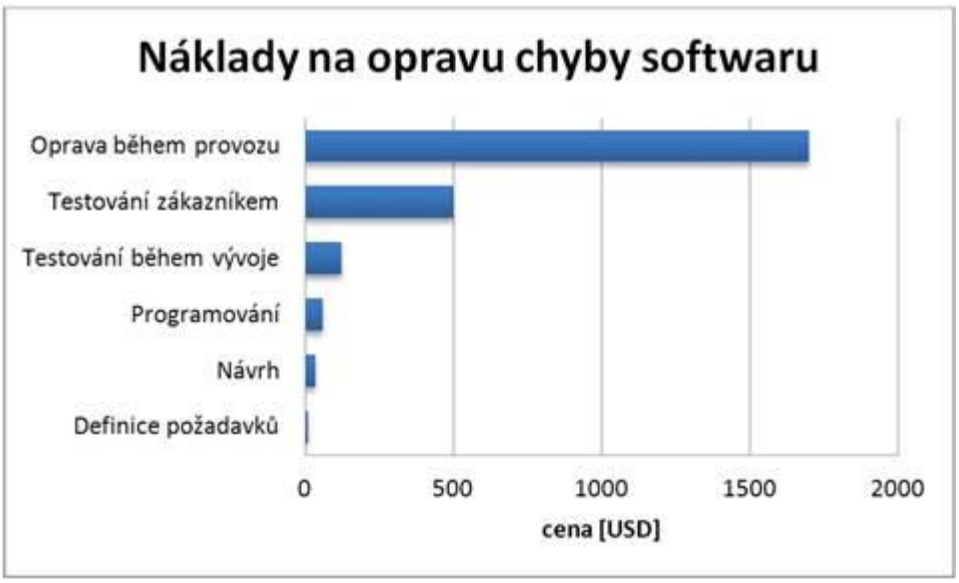
Metriky produktu

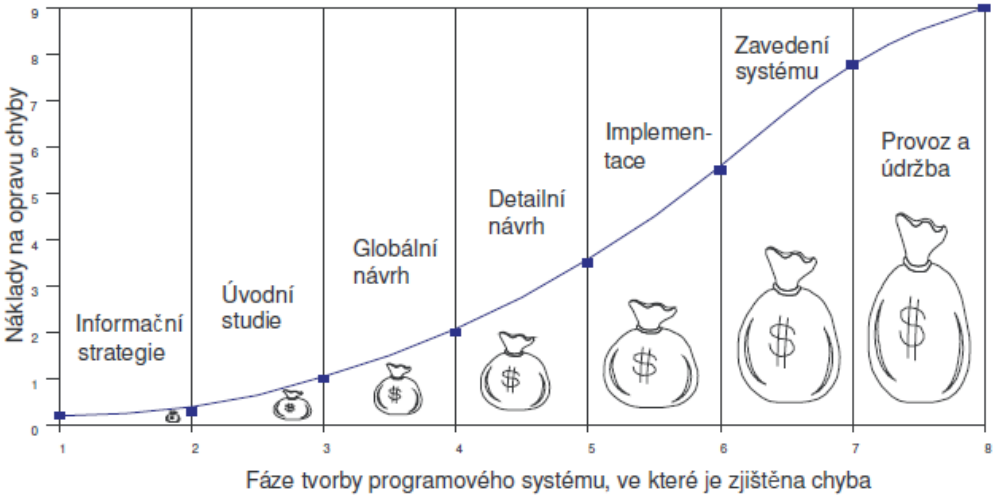
- Složitost, přehlednost
  - McCabe cyclomatic complexity
  - Fan-in / fan-out (afferent / efferent coupling) => stabilita
  - Weighted method per class
  - Lack of cohesion
- Velikost
  - Počet UC, funkčních bodů
  - Možná někdy případně i také LOC
    - SLOC, DSLOC, CBLOC, TLOC

- Metriky produktu (2)
  - Spolehlivost
    - MTBF = MTTF + MTTR
    - Dostupnost je pak (MTTF / MTBF) * 100
  - Kvalita (nepřímé metriky)
    - Pokrytí testy - kódu, požadavků
    - Charakteristiky defektů - hustota, výskyt
    - Kvalita zdrojového kódu
  - Nástroje
    - PMD, FindBugs, …
    - IDE pluginy
- Projektové a procesní metriky
  - Postup
    - Project velocity / burndown
    - Jitter - change requesty a jejich zpracování, staff turnover, změny postupu a plánu
  - Kvalita
    - Breakage = průměrná váha změnu LOC / CR (lines of code / change request)
    - Pracnost celkem, přepočtená na Change Requesty
    - Defect discovery rate, defect removal (zpracování, trendy)
- Jak měřit
  - **Top-down**
    - Definovat cíl měření
    - Zvolit metrik
  - **Goal-Question-Metric**
  - **Bottom-Up**
    - Jaké metriky?
- **Goal-Question-Metric**
  - Přístup k definování metrik
  - Rámec pro systém zaměřený na konkrétní problémy
  - **Goal** = problém + cíl měřícího programu
  - **Question** = měřené objekty a způsob měření
  - **Metric** = konkretizují získávaná data
- Nástroje pro měření
  - Spreadsheet, kalendář
  - Bugtracker
  - Statsvn
  - Junit a corbetura
  - Databáze
- Řízení měření
  - Plán měření
    - RUP template
    - GQM přístup
    - Definice metrik, jejich význam a zpracování
    - Způsob získání dat
  - Sledování projektu a produktu
    - Automatické získávání a vyhodnocování
    - Sledování (management)
    - Korektivní akce

Oponentura

- **Technická oponentura**
  - Též Faganovská inspekce
  - Skupinová technika, cílem je odhalit chyby v návrhu/kódu/dokumentu, sledování standardů, vzdělávání
  - Ne: dělat potíže autorovi (neúčast vedení), hledat nápravu chyb
- **Role ve skupině**
  - Moderátor - řídí diskuzi
  - Průvodce - předkládá dílo
  - Autor - vysvětluje nejasnosti
  - Zapisovatel - zaznamenává nalezené problémy
  - Oponenti - hledají chyby, obvykle podle seznamů otázek
- **Postup**
  - Příprava
    - Distribuce díla (moderátor), projití a hledání problémů (oponenti) - několik dní předem, cca 2 hodiny práce
  - Schůzka
    - Sekvenční procházení díla (průvodce či moderátor)
    - Vznášení připomínek
    - Zapisování nálezů (chyb a otevřených otázek)
      - Nejvýše 2 hodiny
      - Nepřipouštět dlouhé diskuze, řešení chyb (moderátor)
      - Možná následná schůzka pro vyřešení otázek
  - Závěry
    - Verdikt: v pořádku/drobné chyby/nutné přepracování/nová oponentura
    - Autor odstraní chyby dle nálezů, moderátor zkontroluje
      - **Dokument: Nálezy oponentury**
- **Zhodnocení technické oponentury**
  - Použitelné ve všech fázích životního cyklu
  - Velmi dobrá detekce chyb (až 75%)
  - Výsledkem jsou nižší náklady na vývoj a vyšší produktivita
  - Nároky
    - Náročné na čas
    - Je třeba zkušenost





Obrázek A 10 Náklady na změny v projektu

1.1.15. Měření software, produktové a procesní metriky, význam pro sledování kvality a řízení postupu.

Metrika = způsob stanovení velikosti

Metriky software

- **Složitost, přehlednost**
  - počet možných cest skrz zdrojový kód
  - Fan-in / fan-out (afferent / efferent coupling) => stabilita
    - strukturální metrika, která měří poměr počtu modulů, které volají daný modul ku počtu modulů, které volá daný modul
  - Weighted Methods per Class
    - součet složitosti všech metod ve třídě
  - Lack of cohesion
    - nedostatek soudržnosti - jedna metoda dělái více funkcí (které se ve svém "smyslu" liší)
- **Velikost**
  - Počet Use Cases, funkčních bodů
  - Lines of Code
    - SLOC (Source Lines of Code),
    - DSLOC (Delivered Source Lines of Code),
- **Kvalita (nepřímé metriky)**
  - Pokrytí testy – kódu, požadavků
  - Charakteristika defektů – hustota, výskyt
  - Kvalita zdrojového kódu
- **Spolehlivost**
  - Střední doba mezi poruchami
  - dostupnost

Produktové a procesní metriky - viz předchozí otázka

Metriky produktu

- počet use case,
- počet podsystémů, modulů, tříd…,
- složitost modulů,
- počet řádků,
- datová velikost (ubuntu na 1 CD),
- počet odhalených chyb v jednotlivých modulech při testování,
- složitost dat modulu (funkční body),
- náklady na vývoj,
- člověkohodiny apod.

Metriky procesu

- Postup projektu
  - Rychlost vývoje
  - Change requesty a jejich zpracování,
  - Staff turnover (fluktuace zaměstnanců),
  - změny postupu/plánu
  - …
- Kvalita
  - Breakage = průměrná váha změny (LOC (Lines of Code) / CR (Change Rate))
  - Pracnost celkem, přepočtená na CR (Change Rate)
    - Množství chyb (procenta) odhalených před odesláním zákazníkovi.

Řízení postupu

- Plán měření
  - RUP template
- GQM (Goal Question Metric) přístup
  - Definice metrik, jejich význam a zpracování
- Sledování projektu a produktu
  - Automatické získávání a vyhodnocování
  - Sledování (management)
  - Korektivní akce

Význam pro sledování kvality a řízení postupu

- Lines of Code nic neznamená pro řízení kvality, ale třeba se dá odhadovat postup
- Je nutné sledovat kvalitu a upravovat vůči ní proces vzhledem k nákladům na zdroje, čas. Navíc pokud mineme chybu a vyjde do produkce, kromě řádově vyšších nákladů můžeme poškodit jméno společnosti

1.1.16. Způsoby detekce chyb v software, metody testování, vztah k sestavení produktu.

Způsoby detekce chyb

- Chyby ve zdrojáku – odhaleny při překladu
- Statické / Dynamické
- Ladění
- Testování
- Inspekce kódu
- Formální verifikace – automatické ověření zda systém splňuje požadavek

Metody testování

- Whitebox
  - máme k dispozici zdrojové kódy program => testování zaměřené na programovou logiku
  - **Unit testy** (testování malých částí programů, jako jsou podprogramy nebo třídy)
  - **Integrační testy** (jsou testovány komponenty a jejich interakce na základě rozhraní)
- Blackbox
  - Metoda testování bez znalosti kódu softwaru. Máme tedy k dispozici specifikaci softwaru a samotný software v podobě „černé skříňky", tzn. že se nemůžeme podívat dovnitř, jak funguje.
  - Řeší jiné typy chyb
    - Nesprávné nebo zcela chybějící funkce
    - Chyby rozhraní
    - Chyby ve struktuře dat nebo externích databázích
    - Neočekávané chování
    - Chyby při inicializaci nebo ukončení
  - **Smoke test** (jestli to vůbec naběhne)
  - **Zátěžový test** (jestli se to sesype)
  - **Systémový test** (funkčnost v kontextu systému a interakce s jinými systémy)
  - **Hraniční testy** (vstupní data velmi blízko nebo na hranici akceptovatelnosti, v praxi je to totiž nejčastější zdroj problémů)
  - **Akceptační testy** (smoke test nového buildu a test zákazníkem po kompletním otestování)
  - **Usability testing** – testování uživatelem, hodnocení přívětivosti, intuitivnosti, …

Beta testing sem nespadá – může se skládat z akceptačních testů, usability testů atd. poté, co se dosáhlo beta milestonu a produkt se v této fázi testuje

Vztah k sestavení produktu

Popsáno v jednotlivých metodách – různé typy v průběhu vývoje, před sestavením, po sestavení a před předáním, test zákazníkem

1.2. SI

1.2.1. Strategické řízení firem, poslání a role IT v organizaci, strategie IT/IS

Strategické řízení firem

Strategie firmy – dlouhodobé určování směru rozvoje podniku. Zohledňuje historický vývoj podniku a character organizace.

Strategické řízení – dlouhodobé plánování a směřování organizace / určuje dlouhodobé cíle a záměry přizpůsobuje se podmínkám prostředí.

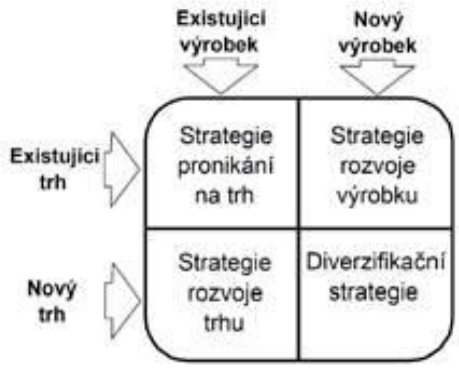
Postup: analýza (vnitřní,vnější, zájmové ksupiny) -> výběr strategie -> implementace strategie -> zpětná vazba

Strategii firmy je možné chápat jako **komplot**, neboli **plánovaný manévr**, nebo **model chování organizace** ve vztahu k jeho historickému vývoji nebo jako pozici, vyzdvihující význam výrobků dodávaných na specifický trh a konečně jako **charakter organizace**.

Strategie je **koncept**, **abstrakce** v myslích zainteresovaných stran.

Strategie je **perspektiva sdílení všemi členy organizace** (jedná se o kolektivní mysl, sjednocení jednotlivců ke společnému způsobu myšlení a jednání).

Typologie strategií (podle Ansoffa)



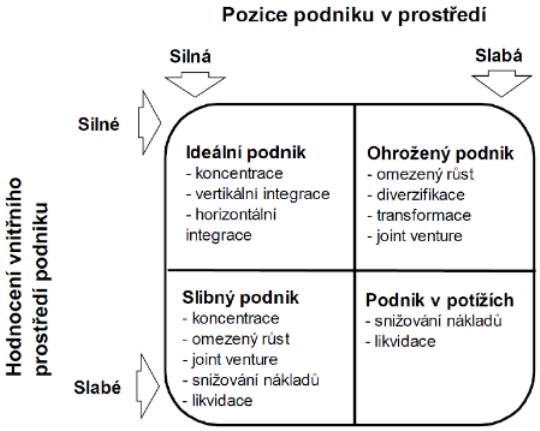
Strategické řízení je **vrcholovým řízením rozvoje podniku** jako celku v **delším časovém rozmezí**:

Strategické řízení = dlouhodobé plánování a směřování organizace

Proces určení dlouhodobých cílů a záměrů, přizpůsobení se podmínkám prostředí a alokace zdrojů organizace ve vztahu ke stanoveným cílům

Zaměření na rozsah činností podniku v dlouhodobém horizontu, které v ideálním případě vytvářejí soulad mezi podnikovými zdroji a měnícím se vnějším prostředím – zvláště trhem a zákazníkem

Integrovaný model strategických alternativ



Záměry

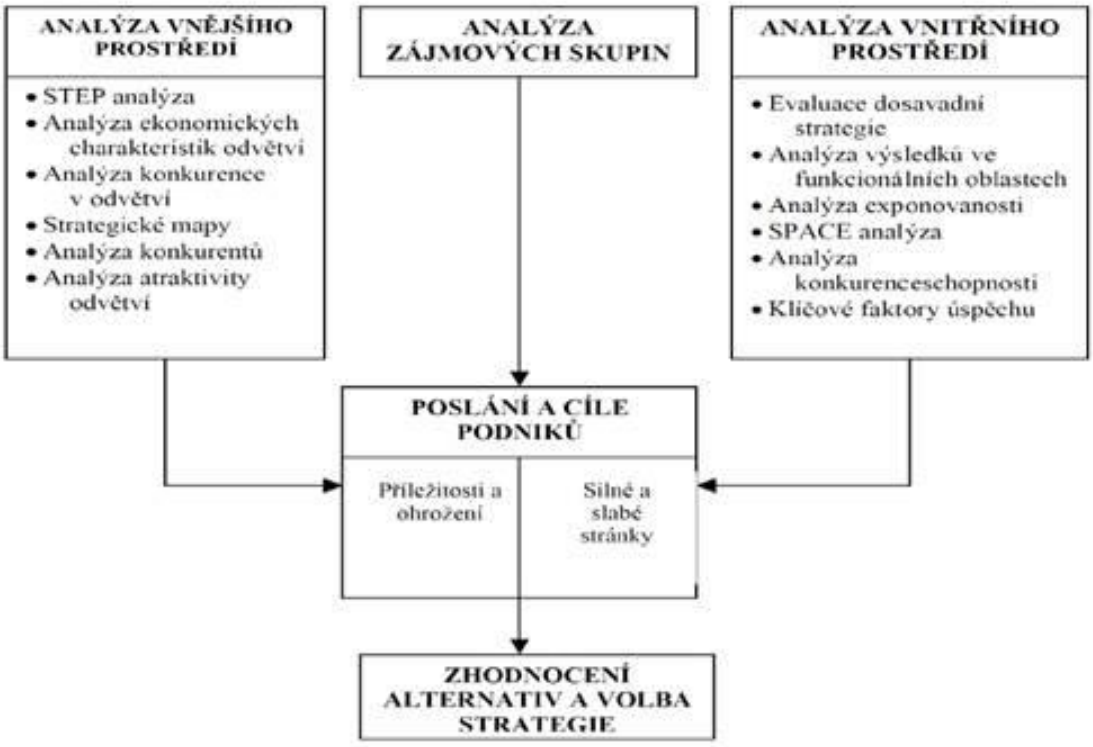
- Finanční i nefinanční zájmy různých zájmových skupin
- Umožňují a podporují zdůvodněné kompromisy
- Kompromisy u protichůdných cílů
- Motivující, ale dosažitelné
- Jdou napříč funkcionálními oblastmi

Cíle

- Operativní vymezení záměrů
- Vyjadřují, čeho chce podnik dosáhnout krátkodobě i dlouhodobě
- V souladu se zaměřením podniku

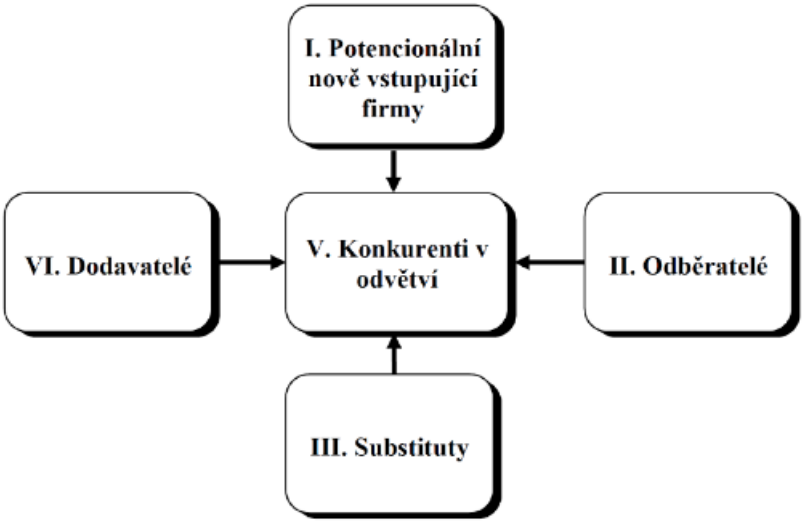
Analýza

Proces formulace podnikové strategie:



Analýza vnějšího prostředí

Porterova analýza:



STEP analýza

- **S**polečenská (úroveň vzdělání, distribuce příjmů, životní styl…)
- **T**echnologická (vládní výdaje za vědu a výzkum, nové vynálezy…)
- **E**konomická (trend vývoje HDP, inflace, nezaměstnanost…)
- **P**olitická (Stabilita vlády, daňová politika, ochrana životního prostředí…)

Ptáme se přitom na otázky:

1. Které z vnějších faktorů mají vliv na podnik?
2. Jaké jsou možné účinky těchto faktorů?
3. Které z nich jsou v blízké budoucnosti nejdůležitější?

Analýza vnitřního prostředí podniku

Analýza výsledků v jednotlivých funkcionálních oblastech: výroba, finance, marketing, úroveň řízení a lidské zdroje, výzkum a vývoj

Portfolio metody

Vytvoření matice portfolia, zmapování konkurenčního prostředí pro každou podnikatelskou činnost a vyvození závěrů o aktivitě všech položek portfolia, ohodnocení konkurenceschopnosti jednotlivých aktivit v portfoliu, hlubší proniknutí do situace podniku, určení potřeby finančních prostředků a dalších podnikových zdrojů na podporu strategií jednotlivých aktivit, porovnání aktivit z hlediska ziskovosti a přitažlivosti odvětví s následujícím roztříděním investičních priorit, kontrola s cílem vyhodnotit vyváženost portfolia, zjištění zda je portfolio v souladu s podnikovou strategií

Bostonská matice

Ukazuje spojitosti mezi tempem růstu obchodů a konkurenční pozicí společnosti, slouží především manažerům společností jako pomoc při řízení a dělání rozhodnutí ohledně zdrojů, dále ukazuje v oblasti skladového hospodářství v závislosti na financích, zajímavosti jako je prodej zboží na trhu, možnosti nárůstu či poklesu skladových zásob. 4 kvadranty:

Otazníky: výrobky zaváděné na trh vyžadují značné finanční vstupy, ale jsou šancí do budoucna, průzkum trhu rozhodne, jestli do nich dále investovat nebo je stáhnout.

Hvězdy: produkty s nejlepšími obchodními výsledky, udržení těchto výsledků je finančně náročné, ale výsledkem je vysoký zisk.

Dojné krávy: hlavní finanční opora firmy, přinášejí vysoké zisky bez větších finančních vkladů.

Bídní psi: produkty na konci prodeje, zvážení podniků, jak dlouho se vyplatí příslušný produkt udržovat na trhu a podporovat jejich prodej zesílenou marketingovou politikou.

Analýza zájmových skupin

- Kulturní kontext – porozumění hodnotám, které společnost uznává
- Politický kontext – posuzujeme, jak různá očekávání jednotlivců nebo skupin mohou ovlivnit účel podniku. Ten se vyjadřuje v jeho poslání a cílech, na jejichž formulaci se podílí dominantní zájmová skupina.
- Etický kontext – týká se vlivu chování jednotlivců na hodnoty sdílené společností

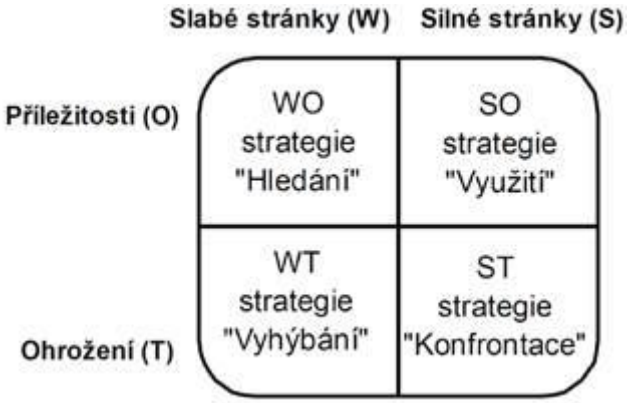
Zájmové skupiny: akcionáři, věřitelé, zaměstnanci, zákazníci, dodavatelé, vlády, odbory, konkurenti, široká veřejnost

Spojením tří předchozích můžeme sestavit swot analýzu

SWOT

SWOT analýza je základní metodou pro posouzení silných a slabých stránek podniku a příležitostí a ohrožení, která jsou závislá na vlivu vnějšího prostředí podniku. SWOT = Strengths, Weaknesses,

Opportunities, Threats. Účelem analýzy je zaměřit se jen na ty stránky, které mají nějaký strategický význam.



Volba strategie

- Generování (vytváření) strategických alternativ
- Určení rámce problému
- Generování souboru

Metody pro podporu generování alternativ

- Generování scénářů
- Generování konfliktů
- Brainstorming
- Teorie chaosu
- Systémy podporující týmovou práci
- Zúžení souboru alternativ

Porovnání a hodnocení strategických alternativ

Hodnocení ve vztahu k následujícím krit.:

- Přijatelnost
- Vhodnost
- Realizovatelnost
- Poskytnutí výhody

Výběr alternativy jako budoucí strategie

Rozhodovací analýza, pro snížení chybovosti se využívá skupinové rozhodování

Kategorie alternativ

- **Zřejmé**, jasné alternativy
- **Kreativní** alternativy
- **Nemyslitelné** alternativy

Poslání

Vymezuje účel a smysl proč podnik existuje

Poslání je integrální součástí strategického zaměření podniku, které vymezuje účel a smysl, kvůli kterému podnik existuje

V obecné rovině je to vize a mise podniku, v konkrétnějším vyjádření pak záměr a cíle.

Vize = vyjadřuje to, čím by podnik měl být – aspirace, zaměření do budoucnosti.

Mise = poslání = zformulovaná a napsaná vize + pohled do minulosti, proč firma vznikla

Efektivně formulované poslání zohledňuje – tržní orientace (vymezení podniku ve vztahu k trhu), realizovatelnost (optimální vymezení předmětu činnosti), motivace (zesilování pocitu zaměstnanců že jejich úsilí je významné a prospívá společnosti), specifikace (vyjádření hodnotového systému podniku, vztahu k zákazníkům, dodavatelům…)

Role IT v organizaci

core operational constituent that can improve business performance and increase shareholder value.

CIO je součástí boardu, ovlivňuje rozhodnutí společnosti.

http://www.iso.com/Research-and-Analyses/ISO-Review/The-Role-of-IT-in-the-Modern-Corporate-Enterprise.html

CTO = chief technology officer = kouká na to z pohledu technologie

CIO = kouká na to z pohledu procesu

IT (IS, Management, Procesní management…) má za cíl podporovat hlavní činnost podniku (projektu), přispívat k jeho úspěchu, umožňovat reagovat na hrozby a využívat (vytvářet) příležitosti. Řídí se strategií definovanou firemním / projektovým managementem, jeho cíle jsou vždy podřízeny cílům podniku/projektu.

Strategie IT/IS

Správně fungující IT prostředí, resp. informační systém, již v dnešní době není pouze konkurenční výhodou, ale je to klíčová podmínka pro business aktivity každé fungující společnosti. Důsledkem toho je krátkodobé i dlouhodobé plánování rozvoje IT prostředí, kterému zpravidla předchází stanovení strategie IT prostředí, ruku v ruce s business strategií společnosti.

Přistupovat strategicky k aktivitám, které mají závažnější dopad na rozvoj a provoz IT, nevyžaduje zpravidla časově náročné studie, bádání a nekonečné hledání. Je třeba mít na paměti, že lze využít řadu již vyzkoušených, ověřených a úspěšně dokončených scénářů. A to je i směr, který naše společnost preferuje – pochopit potřeby zákazníka a díky dlouholetým zkušenostem v různých IT prostředích, a znalosti vývoje trendů v IT, navrhovat směry rozvoje IT, které zákazníkovi budou přinášet hmatatelný užitek již nyní.

V hodnocení a návrzích strategie rozvoje IT se soustředíme na tyto priority:

- Zvyšování produktivity uživatelů
- Snižování nákladů na provoz IT (TCO)
- Zvyšování bezpečnosti zpracovávaných dat

1.2.2. Komponenty podnikévého IT, přehled oborových a technických standardů

Komponenty podnikového IT

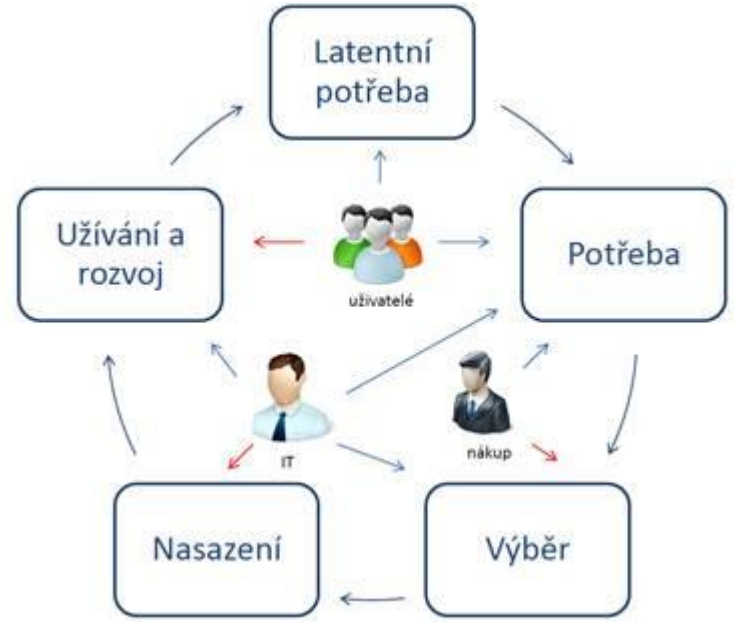
- **Hardware**
  - Server, Workstation, Storage, Mobile, Monitoring, Backup, Asset Management (inventarizace)
- **Networking**
  - Connectivity, VPN, FireWall, Intranet, Extranet, Web, VoIP, Mobile device
- **Information**
  - Databáze, ETL (Extract Transform Load), MDM (Master Data Management), ECM (Enterprise Content Management), DMS (Document Managemtn System), DW (Data Warehouse), Business Inteligence, Business Analytics
- **Enterprise Applications**
  - ERP (Enterprise Resource Planing), CRM (Customer Relation Management), SCM (Supply Chain Management), HR (Human Resources), Helpdesk, Spisová služba, Accouting, Asset Management, CPM (Corporate Performance Management), Bussiness monitoring, Dashboardy
- **Security**
  - Identity management, Authentication, Autorization, CA (Certification Authority), Certifikáty, SSO (Single Sign-On), Antivir, Antispam
- **Colaboration**
  - E-mail, IM (Instant Messiging), Groupware, Calendar, Resource planning, DMS/CMS/ECM, Workflow, BPM (Business Process Management), Portal, Mash-up, Social network
- **Middleware**
- Application infrastructure (aplikační server, cluster, high availability, dissater recovery), ESB, messaging, IT governance, procesní servery, Business Rules, adaptéry, konektory, B2B gateways
- **Development**
- Requirements definition and management, Analysis, Design, Construction tools, Deployment, release management, testing, QA, Project management, Portfolio management

Přehled standardů

- **Technické**
- SQL, BPEL (Business Process Execution Language), BPMN(Business Process Model Notation), TCP/IP, DNS, SOA (Service Oriented Architecture)
- **Procesní**
- PMBOK (Project Management Body of Knowledge), ITIL (Information Technology Infrastructure Library), CMMI (Capability Maturity Model Integration), ISO
- TOGAF (The Open Group Architecture Framework) – komplexní přístup k návrhu, plánování, implementaci a dohledu enterprise architektury
- ISO 9000 (kvalita)

1.2.3. Životní cyklus IS, dodávka IS, process akvizice IS/IT systému

Životní cyklus IS



Plánování – identifikace potřeb, vnitřní vlivy (strategie), vnější vlivy (legislativa), plánování financí, plánování zdrojů, koordinace projektů, způsob pořízení (krabicové řešení, na klíč, na míru…), Return of investment (ROI)

Analýza – detailní analýza potřeb, funkcí, parametrů; stakeholdeři, konzultační firmy, prováděna formou dokumentu, ustanovení realizačního týmu, pracovní skupiny, identifikace omezení (finance, lidé, čas)

Výběr řešení a dodavatele – poptávka, výběrové řízení (tender), Proof of Technology (**PoT**) = ukázka, demo; Proof of Concept (**PoC**) = pilot, placené, na míru společnosti; porovnávání řešení, vyhodnocování řešení, faktické vlastnosti, míra uspokojení požadavků, ekonomické faktory (cena, splátky, Return of Investment – **ROI** = celkový zisk/náklady, Total Cost of Ownership – **TOC** = cena včetně údržby apod.)

Dodávka IS

Implementace – může trvat dny až roky; dělí se na více fází, vyžaduje součinnost organizace;

možnost customizace, konfigurace, integrace na stávající systémy, nové a změněné komponenty – HW, SW, procesy, migrace dat, Quality Assurance (QA), testování, zaškolení lidí, předání

Postimplementační podpora – doladění systému, trvá obvykle týdny až měsíce

L1 podpora – uživatelská (helpdesk)

L2 podpora – systémová (admin)

L3 podpora – aplikační (změna kódu)

Vnější a vnitřní zdroje

Provoz, podpora – řádově roky, implementace dílčích změn, aktualizace, záplatování, sledování provozních parametrů, helpdesk, servicedesk, ITIL, reporting a sledování nákladů.

Ukončení, migrace – tzv. sunseting, dožití. Zmrazení investic a zahájení přípravy akvizice nového systému.

Proces akvizice IS/IT systému

Plánování – Analýza – Výběr řešení a dodavatele – dodávka a implementace – postimplementační podpora – provoz, podpora - dožití

1.2.4. Proces výběrového řízení, poptávka a nabídka, výběr a nákup řešení, stude proveditelnosti, PoC, PoT, poptávkové řízení (RFI, RFP, RFQ)

Proces výběrového řízení

Zadavatel nejdříve udělá RFI, aby zjistil, jaké jsou možnosti apod. Odpovědí na RFI je obvykle FS (Feasibility Study). Na základě toho zadavatel zpracuje poptávku (RFP, RFQ) a jako odpověď dostane nabídku. Z přijatých nabídek dle hodnoticích kritérií vybere zadavatel tu, která nejlépe splňuje požadavky uvedené v poptávce. Svou volbu pak oznámí ostatním účastníkům výběrového řízení.

Poptávka

- Žádost o kompletní nabídku
- Zadavatel chce realizovat nákup
- Má:
  - Formální část
  - Věcná část
  - Finanční část
- Desítky stran, někdy i stovky, přílohy…

Struktura

- Zadavatel
- Kontakty pro dotazy
- Harmonogram procesu
- Zadávací dokumentace
- Požadavky na uchazeče
  - Kvalifikační předpoklady
  - Reference
- Požadavky na nabídku
  - Formální – struktura jak má nabídka vypadat
  - Variantní řešení
  - Popis řešení
  - Strukturovaná cena

Nabídka

- Identifikace nabídky
- Disclaimer
- Kontakty na dodavatele (kdo napsal nabídku)
  - Obchodní
  - Technické
  - Statutární
- Informace o dodavateli
  - Přehled
  - Certifikace
  - Organizační struktura
  - Obraty
- Executive summary
- Popis variant řešení
  - Koncepce, architektura

        - Produkty
        - Služby
        - Customizace
        - Rozšiřitelnost
    - Výhody řešení
    - Diskuze požadavků zadavatele
    - Zkušenost dodavatele, reference
    - Navrhovaný harmonogram realizace
    - Součinnost zákazníka
    - Cenová nabídka
        - Co je a co není v ceně
        - Na pořízení
        - Na vlastnictví X let -> TCO (Total Cost of Ownership)
        - Platební kalendář
    - Odkazy
        - Dokumentace použitých komponent třetích stran
        - Licenční ujednání
    - Návrh smlouvy
    - Složení projektového týmu včetně CV

Výběr a nákup řešení

Rámcový proces výběru a nákupu:

- Latentní potřeba
- Potřeba
- Studie proveditelnosti, analýza
- Vize řešení
- Požadavky
- RFI
- RFP – poptávka, zadávací dokumentace
- Výběr, shortlist
- RFQ – cenová nabídka
- Podpis smlouvy

Studie proveditelnosti

Zabývá se:

- Současný stav
- Seznam-analýza požadavků
- Možné přístupy řešení
- Popis variant
    - Funkcionalita, technologie, architektura, API, dokumentace
    - Zhodnocení variant
    - SWOT
- Analýza a mitigace rizik
    - Mitigace (co s nimi): ignorovat, přijmout, protiopatření, delegace
- Odhad nákladů a přínosů
- Provedené zkoušky, testy, prototypy
    - PoC, PoT

Obecná struktura

- Obsah

- Historie změn
- Úvod
- Účel
- Rozsah
- Reference
- Executive summary
- Obsahová část
- Shrnutí
- Přílohy

PoC

Proof of concept = pilotní verze řešení – placené, v součinnosti se zákazníkem

- Ověření vhodnosti řešení pro konkrétního zákazníka
- Ověření předpokladů TCO, ROI
- Identifikace problémových míst a rizik implementačního projektu
- Nastavení základu pro odhady pracnosti a složitosti
- Upřesnění požadavků zákazníka
- Data zákazníka
- Součinnost zákazníka
- Trvání dny až týdny
- Obvykle placené

PoT

Proof of technology = ukázka řešení, demo = zdarma většinou

- Technologické nebo produktové demo
- Ukázka funkčnosti (kompletního) řešení
- Ověření pro dané prostředí
  - Technické (OS, integrace…)
  - Tržní (segment, jazyk, velikost zákazníka…)
- Připravené dodavatelem
- Generická data
- Trvání hodiny – dny
- Obvykle zdarma

Poptávkové řízení

Rozdíl mezi poptávkovým z výběrovým řízením je, že v poptávkovém řízení oslovíme jen vybrané firmy, zatímco v tom výběrovém mají šanci všichni.

RFI

Request for information = business proces, jehož cílem je zjistit informace o možnostech dodavatelů, cílem je obvykle získání dostatečného množství informací, na jejichž základě je možné provést kvalifikované rozhodnutí.

Zadavatel hledá řešení, jedná se o první fázi RFP, odpovědí může být studie proveditelnosti, ceny jsou pouze orientační. Struktura RFI: zadavatel, harmonogram procesu RFI, RFP, definice problému, rámcové požadavky, omezení, cenová představa, kritéria výběru, časová představa – harmonogram realizace, požadavky na uchazeče – kvalifikační předpoklady a reference, struktura odpovědi. (z přednášek)

..

RFP

Request for proposal = proces, jehož cílem je vyžádání nabídky od dodavatelů. U RFP klient neví přesně co chce a nebo není schopen to, co chce dostatečně přesně specifikovat. Jinými slovy klient ví, jakou potřebu chce řešit, ale neví jak na to. Ještě jinak formulováno cílem je zjistit JAK by dodavatel řešil daný problém a KOLIK by si účtoval za implementaci svého řešení.

RFQ

Request for quotation = proces, jehož cílem je vyžádání nabídky od dodavatelů. Na rozdíl od RFP zde zadavatel přesně ví, co chce a také ví, že existuje více dodavatelů, od nichž si tu samou věc může koupit. Cílem je obvykle získání co nejnižší ceny. Ještě jinak formulováno cílem je zjistit pouze KOLIK by si dodavatel účtoval za konkrétní řešení.

1.2.5. Projektové a multiprojektév řízení, projektová kancelář, PMBOK

Projektové a multiprojektové řízení

Co je podstatou projektového řízení? Tento výraz vznikl z anglického termínu project management, kterým se rozumí řízení projektu s jasně stanoveným cílem, který musí být dosažen ve stanoveném čase, nákladech a kvalitě. Projekty jsou často rozhodující součástí strategického řízení podniku. Mohou být zaměřeny na inovace výrobků, zavádění nových technologií, vývoj softwaru, modifikaci procesů a postupů, realizaci stavebních či investičních akcí, zavádění systémů řízení jakosti či realizaci podnikatelských záměrů. Jednoduše řečeno, projekt je organizované úsilí směřující k dosažení určitého cíle. Vyznačuje se jasně stanoveným konkrétním cílem, termínem, omezenými zdroji a především specifikací přínosů jeho realizace. Projektem naopak není periodicky se opakující práce či každodenní kontrolní činnosti.

Projektové řízení je komplexní proces, který vyžaduje profesionální přístup a podporu vrcholového managementu organizace. K řízení projektů se používají specifické nástroje a techniky. Všechny projekty se vyznačují společnými základními postupy a životním cyklem. Projekt je dynamický systém, který v sobě zahrnuje fázi iniciace a přípravy, plánování, realizaci i controlling projektu. Vyznačuje se samozřejmě i určitými specifickými riziky a v jeho průběhu se může vyskytnout řadu problémů. Jistá specifika mají i projekty v multiprojektovém prostředí, tedy v prostředí, ve kterém se řídí více projektů, dochází ke sdílení zdrojů a na projektové manažery jsou kladeny vyšší nároky. Neodmyslitelnou součástí profesionální projektového řízení a podmínkou úspěchu je i týmová spolupráce, protože právě lidský faktor se může stát příčinou mnoha komplikací nebo naopak zdrojem pozitivních synergických efektů.

http://www.systemonline.cz/clanky/aktualni-otazky-projektove-rizeni.htm

Projektová kancelář

Ve velkých společnostech se setkáváme s útvarem projektové kanceláře. Projektová kancelář hraje roli interního zákazníka ve společnosti.

Když stavíme projektový tým, musí v něm existovat duální hierarchie projektové organizace. Je zde strana zákazníka, jako příjemce dodávky a oponenta kvality dodávky, která se snaží minimalizovat finanční náklady a maximalizovat přínosy z ní plynoucí. Vůči ní vystupuje strana dodavatele, která se snaží rovněž minimalizovat své náklady a maximalizovat tím zisk, nebo tržby. Je to model klasického obchodního vztahu. Má dvě strany. Dodavatele, který se snaží rychle dodat, akceptovat a utéct a zákazníka, jehož role je kontrolovat, přebírat a platit.

Projektová struktura je virtuální. Vzniká a zaniká společně s projektem a je naplněna z interních liniových struktur zákazníka a dodavatele. Toto pravidlo se musí týkat všech projektů. Každý musí mít svého zákazníka a dodavatele, jinak by nemohl být úspěšný.

Pokud je ve společnosti rozhodnuto o tom, že bude implementován nový informační systém, nebo proběhne jiná velká dodávka od externího dodavatele, musí za sebe společnost postavit někoho, kdo odřídí projekt na straně zákazníka. V tomto případě není pochyb o tom, že je ideální, aby tuto roli

sehrála projektová kancelář, neboť role projektového manažera vyžaduje potřebné kapacity a kompetence pro řízení projektu.

Dalším typickým případem, který může nastat, je realizace interního projektu. Stanovíme dodavatele, kterým je organizační jednotka, nebo tým složený z několika organizačních jednotek společnosti současně. Zde to svádí k tomu, postavit do čela týmu dodavatele projektovou kancelář. Jenže kdo v tom případě sehraje roli zákazníka? Právě do této role by se měla projektová kancelář posunout. Projektového manažera dodavatele je nutné hledat mezi manažery se znalostí dodávaného produktu a zkušeností s vedením dodávek. Projektová kancelář pak má za úkol dohlížet na průběh projektu v roli zákazníka a odběratele, hodnotit jeho kvalitu a přejímat a kontrolovat výstupy.

A máme tu poslední případ. Společnost dodává svůj produkt externímu zákazníkovi. Jakou roli zde hraje projektová kancelář? Vážení přátelé, ideálně žádnou!

http://www.systemonline.cz/rizeni-projektu/role-projektove-kancelare-ve-spolecnosti.htm

A Project Management Office (PMO) is a group or department within a business, agency or enterprise that defines and maintains standards for project management within the organization. The PMO strives to standardize and introduce economies of repetition in the execution of projects. The PMO is the source of documentation, guidance and metrics on the practice of project management and execution. In some organisations this is known as the **Program Management Office** (sometimes abbreviated to **PgMO** to differentiate); the subtle difference is that program management relates to governing the management of several related projects.
Traditional PMOs base project management principles on industry-standard methodologies such as PRINCE2 or guidelines such in PMBOK.

From <http://en.wikipedia.org/wiki/Project_management_office>

PMBOK

Project Management Body Of Knowledge, je metodika a příručka pro projektové řízení vyvíjena neziskovou organizací zaměřující se na projektové řízení PMI (Project Management Institute). Základem je shromažďování nejlepších praxí z oboru a uvedení jich ve standard pro řízení projektů.

- Řízení integrace projektu
- Řízení rozsahu projektu
- Řízení času v projektu
- Řízení nákladů v projektu
- Řízení kvality projektu
- Řízení lidských zdrojů projektu
- Řízení komunikací v projektu
- Řízení rizik v projektu
- Řízení obstarávání v projektu

1.2.6. Provoz IS/IT (dodávka a podpora IT služeb), řízení změn, ITIL

Provoz IS/IT

POZN. Stejné jako 1.2.3???

Implementace – může trvat dny až roky; dělí se na více fází, vyžaduje součinnost organizace;

možnost customizace, konfigurace, integrace na stávající systémy, nové a změněné komponenty – HW, SW, procesy, migrace dat, Quality Assurance (QA), testování, zaškolení lidí, předání

Postimplementační podpora – doladění systému, trvá obvykle týdny až měsíce

L1 podpora – uživatelská (helpdesk)

L2 podpora – systémová (admin)

L3 podpora – aplikační (změna kódu)

Vnější a vnitřní zdroje

Provoz, podpora – řádově roky, implementace dílčích změn, aktualizace, záplatování, sledování provozních parametrů, helpdesk, servicedesk, ITIL, reporting a sledování nákladů.

Ukončení, migrace – tzv. sunsetting, dožití. Zmrazení investic a zahájení přípravy akvizice nového systému.

Řízení změn

POZN. ITIL – change management?

Vždy když se má zavézt změna do procesu/produktu/... musí se tyto změny zaznamenat – co se mění, jak, odpovědná osoba, očekávaný přínos... ITIL coby knihovna best-practices věnuje svou část v knize Service Support právě návrhu procesů Change managementu (tj. spadá to pod Operativní plánování)

- Zajistit hladkou a nákladově efektivní implementaci pouze schválených změn
- Minimalizovat vznik incidentů resultujících z provedených změn v infrastruktuře
- Change management odpovídá za:
  - Řízení oběhu Request for Change (žádosti o změnu),
  - Schvalování a plánování změn
  - Koordinaci implementace změn
- Change management svou činností zajišťuje FLEXIBILITU infrastruktury

ITIL

Co to je

Information Technology Infrastructure Library = soubor best practices pro IT service management = rozsáhlý, konzistentní a procesně orientovaný rámec pro IT service management. Je to knihovna řešící definici procesů, jejich I/O, stanovení rolí a odpovědností, měření kvality poskytovaných služeb, vazby mezi procesy, zásady pro implementaci procesů, přínosy procesu, náklady, critical success factors, zásady řízení a bezpečnosti ICT infrastruktury

Verze 2

Verze 3

Co řeší

Vydefinování procesů potřebných pro zajištění ITSM:

Stanovení cílů, vstupů a aktivit každého procesu

- Stanovení rolí a jejich odpovědností v daném procesu
- Způsob měření kvality poskytovaných IT služeb a účinnosti ITSM procesů

- Vzájemné vazby mezi jednotlivými procesy
- Postupy auditu a zásady reportingu pro každý proces
- Zásady pro implementaci procesů ITIL:

Přínosy každého procesu

- Critical success factors, možné problémy a vhodná protiopatření
- Náklady na implementaci a následný provoz
- Zásady pro řízení podpůrné ICT infrastruktury
- Zásady bezpečnosti ICT infrastruktury

Neřeší: konkrétní podobu organizační struktury, podobu a obsah pracovních procedur, projektovou metodiku implementace ITSM

Publikace = oblasti

Service Support a Service Delivery – základní, nejznámější, knihy o řízení, dodávce a podpoře IT služeb

Service Support

Service desk

Zajišťuje na denní bázi aktivní kontakt se zákazníky, uživateli, pracovníky vlastní organizace a pracovníky externí podpory, tzv. single point of contact pro uživatele a zákazníky
Zajišťuje obnovu standardní dodávky služby s minimálním dopadem na zákazníky, a to v mezích dohodnuté úrovně služby a podle obchodních priorit

Configuration management

Podporuje ostatní procesy poskytováním věrohodných informací o konfiguračních položkách infrastruktury
Stará se o konfigurační databázi CMDB

Incident management

Obnovuje normální provoz služby a to co nejrychleji při současné minimalizaci důsledků výpadku na provoz
Odpovědný za včasnou detekci problémů, jejich zaznamenávání a řízení jejich životního cyklu
Nezkoumá, proč k problémům dochází, jen hledá nejrychlejší řešení

Problém management

Zabránit opakování incidentů
Analyzuje incidenty, hledá příčiny, nápravu
Zajišťuje stabilitu celé infrastruktury

Change management

Zajišťuje hladkou a nákladově efektivní implementaci změn
Minimalizuje vznik incidentů plynoucích z provedených změn
Schvalování, plánování, koordinace a implementace změn

Release management

Zajistit hladký a kontrolovaný průběh nasazení nových verzí hardware a software do produkčního prostředí

Service delivery

Service level management

Udržování a zlepšování kvality IT služeb
Vyjednávání o obsahu a uzavírání Service Level Agreements, Operation Agreements…
Klíčový článek ITSM, spojuje poskytovatele a odběratele

Capacity management

Zajistit optimální kapacitu ICT infrastruktury
Hledání rovnováhy mezi existující kapacitou a náklady na upgrade

Availability management

Zajišťuje nákladově optimální dostupnost IT služeb, která bude v souladu s obchodními potřebami
Plánování, měření a sledování dostupnosti IT služeb

IT service continuity management

Obnova funkčnosti infrastruktury po vážném výpadku ve schválených mezích
Zpracování analýzy obchodních dopadů globálního výpadku

Financial management for IT services

Poskytuje nákladově efektivní správcovství ICT majetku a zdrojů
Sestavuje rozpočet ICT

ICT Infrastructure Management - Kniha aspektů řízení ICT infrastruktury od identifikace obchodních požadavků přes nabídkové řízení až po testování, instalaci, nasazení a následnou pravidelnou údržbu a podporu ICT komponent a IT služeb. Kniha popisuje hlavní procesy týkající se řízení všech oblastí souvisejících s technologiemi.

Application Management - Procesy celého životního cyklu aplikačního softwaru od prvotní studie proveditelnosti, přes vývoj, testování, vytváření aplikační dokumentace a školení uživatelů, implementaci do produkčního prostředí, provoz aplikace, změnová řízení během provozu aplikace až po stažení aplikace z používání.

Business Perspective - Určena zejména vedoucím pracovníkům obchodních a provozních úseků podniku. Jsou zde představeny základní prvky a principy řízení ICT infrastruktury, IT Service Managementu a Application Managementu, které jsou nezbytné pro podporu obchodních procesů.

Planning to Implement Service Management - Popisuje aktivity, úkoly a problémy související s plánováním, implementací a zlepšováním procesů IT Service Managementu v podnikovém prostředí. Je určena především členům implementačních týmů

Security Management - Popis organizace a řízení bezpečnosti ICT infrastruktury z pohledu IT manažera, a popis procesu plánování a řízení definované úrovně bezpečnosti informací a IT služeb včetně všech aspektů souvisejících s reakcí na bezpečnostní incidenty.

Software Asset Management - Popis procesů řízení, kontroly a ochrany softwarového majetku ve všech stadiích jeho životního cyklu

CCMDB – change and configuration management DB, info o všech konfigurovatelných Položkách

1.2.7. Integrace na datové vrstvě, MDM, ETL

Integrace na datové vrstvě

Přenos souborů

Jeden z nejjednodušších způsobů integrace na datové vrstvě je export, přenos a importu souborů. Rozhraní mezi systémy je v tomto případě realizováno samotným souborem, který má určitý definovaný formát, používaný jak zdrojovou, tak i cílovou aplikací. Výhodou je zejména skutečnost, že ukládání dat do souborů podporují všechny operační systémy a pro realizaci není nutné využívat další technologie.

Sdílená databáze

U tohoto stylu integrace používá více informačních systémů či aplikací jednu databázi pro společné ukládání provozních dat. Jednotlivé systémy tedy využívají společnou databázi a společný datový model. Výhodou oproti předchozímu stylu integrace pomocí přenosu souborů je aktuálnost dat, kdy nedochází k časovému zpoždění a veškerá data jsou vždy aktuální.

Sdílené soubory

V případě integrace pomocí sdílených souborů je architektura a princip podobný předchozímu stylu, tj. integraci pomocí sdílené databáze. Zde se však jedná o využití společného diskového úložiště a společných souborů obsahujících provozní data. Jedná se o jednoduchý způsob integrace na datové vrstvě, jehož problémem je zejména zajištění a ošetření souběžného přístupu k souborům.

Replikace dat

MDM

Master Data Management, je přístup, který pomáhá jednoznačně identifikovat a integrovat klíčová data. Obsahuje procesy a nástroje pro definici a správu “master” dat (údaje z CRM,ERP, Data Warehouse).

MDM tvoří jeden z pilířů datové kvality. MDM je součástí Data Governance (data quality, data management, data policies). Cílem MDM (jako i celé DG) je zajistit kontrolované a konzistentní vytváření Master data a jejich kvalitu.

ETL

Extract Transform Load, mechanizmus získávání dat z provozních systémů podniku (ekonomika, skladové hospodářství, výroba, odbyt atd.), jejich následné zpracování a poskytnutí aplikacím pro podporu rozhodování (decission support systémy, datové sklady, business intelligence) - hlavně jde o uložení do cílového úložiště, LOAD do data warehouse typicky. Úkolem ETL nástrojů (IBM InfoSphere DataStage a QualityStage) je maximálně zjednodušit a zefektivnit implementaci a současně provoz ETL procesů.

Extraction – přenos dat ze zdroje (S-FTP, SCP, DB LINK – ODBC…)
Transformation – konverze dat, normalizace dat, unifikace (eliminace redundantních dat), historizace, agregace (při transformacích ve vyšších vrstvách skladů)
Loading – uložení do cílového úložiště

1.2.8. Integrace na aplikační vrstvě, SOA

Integrace na aplikační vrstvě

Služba

Služba je dobře definovaná a vymezená funkcionalita, která je zcela zapouzdřená a nezávislá na svém okolí (stavu ostatních služeb).

Web services

- Univerzální a platformně nezávislý způsob propojení na bázi XML.
- SOAP = **Simple Object Access Protocol** = nejběžnější způsob, jakým mezi sebou služby komunikují
- WSDL = **Web Services Description Language** – popis rozhraní služby
- Ws-gateways, ESB = metodiky propojování služeb

ESB

Enterprise Service Bus = koncept sběrnice služeb, místo propojování všeho se vším vytvoříme sběrnici, ke které vše připojíme. Komponenty (služby) totiž obvykle komunikují jeden s jedním a tvorba dvoubodových spojení pak vede časem k chaosu. ESB vytváří jakousi P2P síť. Jedná se o protokolově nezávislý způsob, jak vyvolat službu. Přijímá požadavky od WS klientů, zjistí, co s nimi má dělat, kam je předat, postará se o implementační detaily => ESB je implementací SOA.

Přínosy

- Úspora nákladů, dle statistik 30-40%, ale ne hned. Umožnění podnikům flexibilně reagovat na změnu, lépe zarovnat potřeby IT a business – k tomu nestačí jen web services, ale je potřeba mít i jiné prvky infrastruktury (middleware). ESB jako propojení služeb, Portál jako vhodné místo pro interakci uživatelů se službami.
- Služby jsou platformově i technologicky nezávislé
- Zjednodušuje využití ICT
- Umožňuje inkrementální nasazení
- Má schopnost rychle adoptovat změny (rychlost = úspora)
- Podporuje podnikání v reálném čase
- Jako vedlejší efekt přináší znovupoužitelnost služeb

SOA

Servisně orientovaná architektura (SOA) je soubor služeb, které jsou nějak spolu propojeny a vzájemně komunikují.

IS komunikují s jinými IS, spolupracují mezi sebou – aby mohly být používány rozumně, musí spolupracovat podobně jako služby reálného světa, tj. asynchronně reagovat na požadavky z různých zdrojů a být použity jako černé skříňky. SOA je navržena aby propojila mezi sebou libovolné služby.

Vývojový cyklus SOA: Posun od kódování ke skládání, Model -> Assemble -> Deploy -> Manage

1.2.9. Integrace na prezentační vrstvě, portály, mashupy, web 2.0

Integrace na prezentační vrstvě

Portály, mashupy.

Portály

Funkce

Enterprise portál webové místo, kde je pro určité skupiny uživatelů cíleně připraven nějaký obsah a funkcionalita (aplikace). = vylepšený webserver s novými funkcemi. Kombinuje různé aplikace a informační zdroje do jediné ucelené prezentace (AGREGACE). Uživatelé v různých rolích vidí odlišný obsah dle svých přístupových oprávnění (PERSONALIZACE). Uživatelé si mohou obsah sami přizpůsobit (CUSTOMIZACE).

Portlet

Stavební kameny stránek, jsou to vlastně kukátka do aplikací = rozšiřující zásuvné moduly.

Agregační princip

Kombinuje různé aplikace a informační zdroje do jediné ucelené prezentace (AGREGACE).

Mashupy

Aplikace, které kombinují výstupy z více různých služeb třetích stran (datových zdrojů) do jedné nové centrální aplikace, kde jsou tato data zobrazena. Touto na první pohled jednoduchou kombinací dat ze dvou zdrojů se získá snadno použitelný nástroj, který dává přidanou hodnotu datům z obou zdrojů. Jsou jedním z velkých fenoménů webu 2.0

Mashupy pro zaměstnance kombinují data z firemních znalostních databází jako jsou např. wiki spolu s dalšími vnitrofiremními aplikacemi a nebo s některou venkovní aplikací (mapy a jiné). Vznikají tak nové nástroje, které ulehčují orientaci v datech. Klientské mashupy mohou být

postavené úplně stejně, ale budou kombinovat pouze data, která jsou veřejná a mohou být přístupná komukoli.

Web 2.0

Označení pro etapu vývoje webu, v níž byl pevný obsah webových stránek nahrazen prostorem pro sdílení a společnou tvorbu obsahu.

Termín "Web 2.0" označuje vývojovou fázi webu, kde se z počátků internetu, statického sdílení dat, vyvíjí dynamický web vytvářený samotnými uživateli.

Ke statickým HTML/CSS stránkám se přidávají dynamické programovací jazyky na serveru PHP, JSP, ASP a vznikají následující technologie:

- Wiki
- Sociální sítě
- Blogy
- Sdílení videa a fotografií

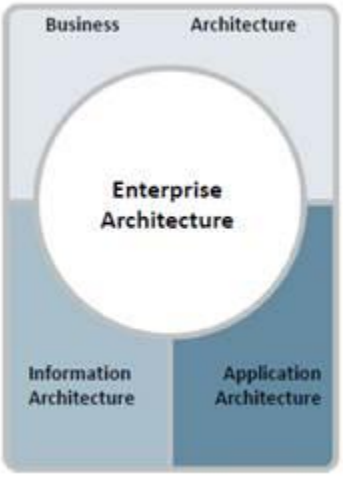
Při integraci Webu 2.0 do obchodních procesů podniků se hovoří o Enterprise 2.0.

1.2.10. Enterprise architektura, IT governance

Enterprise architektura

Modelová situace - zaměstnanec potřebuje spojit data z více aplikací - přijde vývojář, napíše program/skript. Pak zaměstnanec zjistí, že by bylo dobrý tohle opakovat denně tak přijde admin a nastaví to aby se to spouštělo každý den. No a pokud se tohle zopakuje u hodně zaměstnanců, dostaneme něco čemu se říká "hairball architecture" chuchvalec vlasů. Problém nastane až se nějaký systém na který je tahle změť vazeb napojená zhroutí/přestane být podporován - pak je problém s tím cokoli udělat, protože nikdo neví kde co změnit. A tohle by měla řešit Enterprise Architektura. Ta by měla zjistit aktuální stav, a vytvořit plán, jak to vše přeorganizovat do nějakých smysluplných bloků, se kterými bude možné v budoucnu manipulovat - to je to "mapování IT na potřeby business". Nějaký plánovači to tedy naplánujou, pak přijde architecture review board a ti to musejí zkouknout že je to tak opravdu ok. No a pak se to implementuje. Cílem je umožnit změnu v podniku někdy v budoucnu bez zásadního dopadu na IT infrastrukturu.

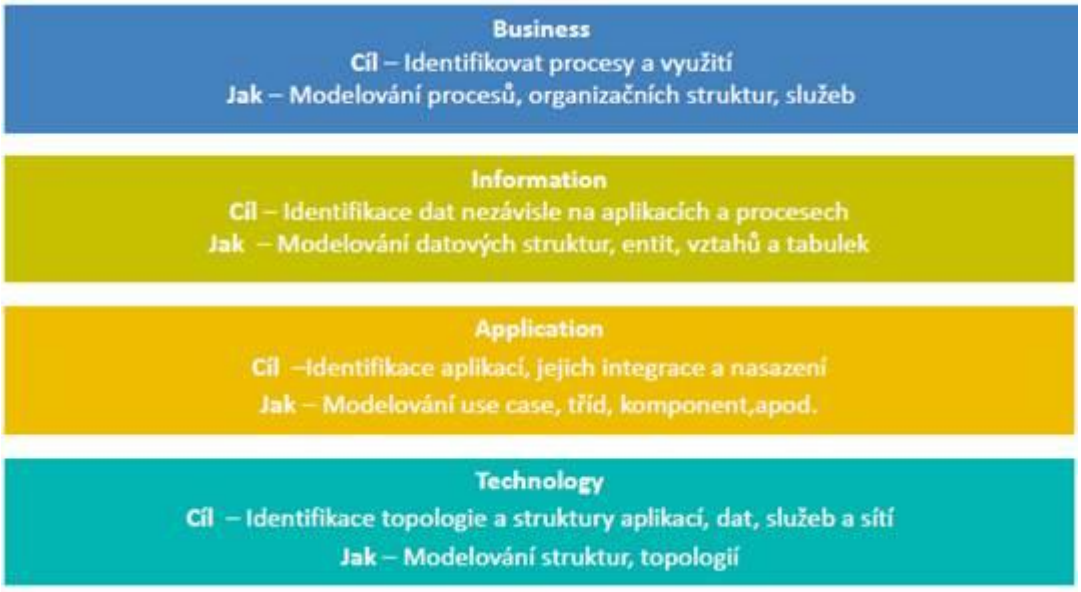
Enterprise architektura – modelování vztahu mezi organizací, byznysem a IT, koncepčně řízený rozvoj tohoto vztahu



EA pomáhá napasovat IT na potřeby business. Dále pomáhá aplikacím homogenních řešení, tzn. Opakovatelná řešení nějakých problémů – proč tisíckrát vymýšlet jak stavět strop když na to je postup který se dá použít vždy.

Součásti:

- Metodologie a metodiky
- Správa metadat
- Standardy
- Plánování IT, řízení projektů
- Modelování



Způsoby popsání Enterprise architektury:

- Zachmann Framework – taxonomie pro popis architektury systémů na enterprise úrovni

Zachman framework

	Why	How	What	Who	Where	When
Contextual	Goal List	Process List	Material List	Organizational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Organizational Unit & Role Rel. Model	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location details	Event Details

- TOGAF (The Open Group Architecture Framework) – komplexní přístup k návrhu, plánování, implementaci a dohledu enterprise architektury
- IBM EA Consulting Method – metodika IBM podporující kompletní řešení enterprise architektury a poskytující standardní výstupy popisující vlastní architekturu, dohled a koordinaci na programové i projektové úrovni, ohled a koordinaci realizace změn architektury

Složky

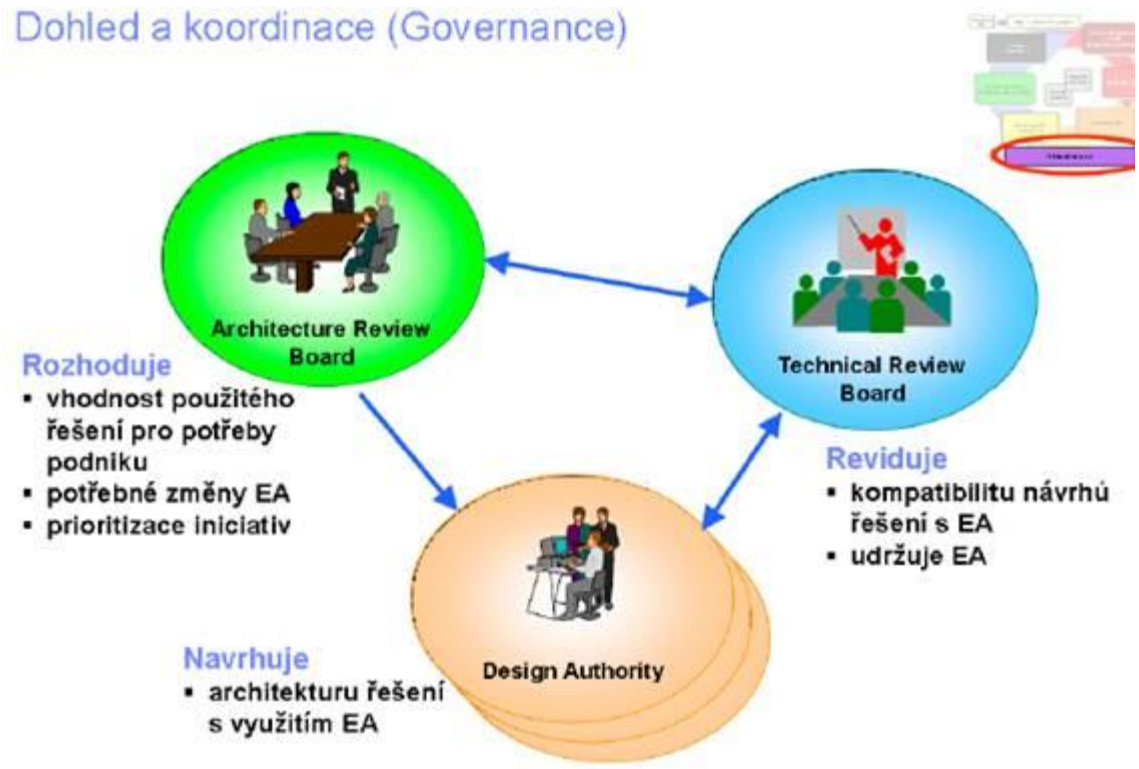
- Strategic capabilities network (SCN)

    - Identifikace kapacit zdrojů potřebných pro dosažení a naplnění strategických cílů
- Komponentní model (CBM)
    - Funkční model podniku
    - Podnik je popsán jako sada vzájemně propojených komponent
    - Komponenty jsou navrženy tak, aby byly schopné fungovat samostatně
    - „black box“ pohled – důležitá jsou rozhraní (poskytované služby, potřebné vstupy, vytvářené výstupy)
- Procesní model (BPM)
    - Procesní model popisující základní entity a vztahy mezi nimi
    - Události, aktivity, role (uživatelé) a data

IT Governance

Information technology governance is a subset discipline of corporate governance focused on information technology (IT) systems and their performance and risk management. The rising interest in IT governance is partly due to compliance initiatives, for instance Sarbanes-Oxley in the USA and Basel II in Europe, but more so because of the need for greater accountability for decision-making around the use of IT in the best interest of all stakeholders.
IT capability is directly related to the long term consequences of decisions made by top management. Traditionally, board-level executives deferred key IT decisions to the company's IT professionals. This cannot ensure the best interests of all stakeholders unless deliberate action involves all stakeholders. IT governance systematically involves everyone: board members, executive management, staff and customers. It establishes the framework (see below) used by the organization to establish transparent accountability of individual decisions, and ensures the traceability of decisions to assigned responsibilities.

From <http://en.wikipedia.org/wiki/Corporate_governance_of_information_technology>



Pěkná prezentace od Profinitu o EA (alespoň jsem z ní lépe pochopil ZACHMAN) str37

1.2.11. Spolupráce a komunikace ve firmách – ECM, BPM, workflow, social business

Spolupráce a komunikace ve firmách

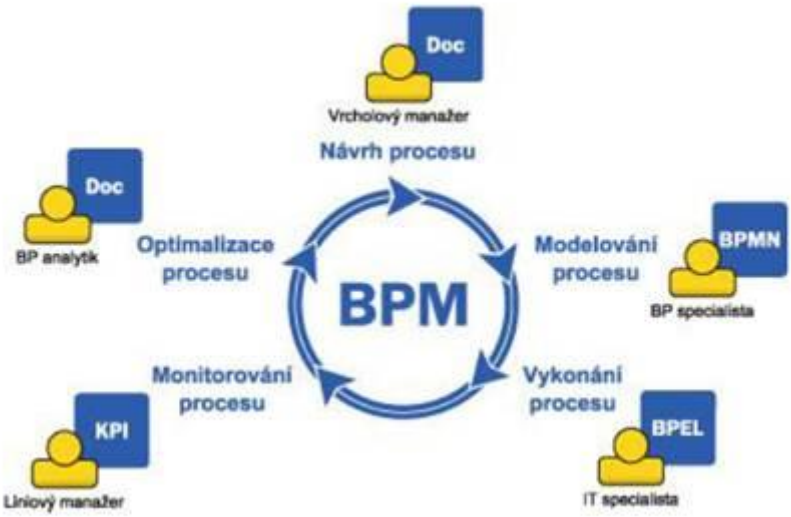
ECM

Enterprise content management, Řešení pro zpracování i nestrukturovaného obsahu (emaily,směrnice,podnikové znalosti). Smyslem je sdílení informací. Konverze dokumentů.

BPM

BPM – Business Process Management, správa podnikových procesů, systematický přístup ke zlepšování procesů v organizaci. Pomáhá zjednodušení a urychlení zavádění procesů v organizaci a jejich změn.

BP - Business Process. Je to abstraktní popis nějaké činnosti – běží dlouho, lidská interakce, platné stavy – nereálné provádět uvnitř aplikací, jež jsou odladěné a fungují jako černé skříňky, BP navrhují Business lidé a IT je implementují. Má jen jednoho vlastníka, který je za proces zodpovědný.

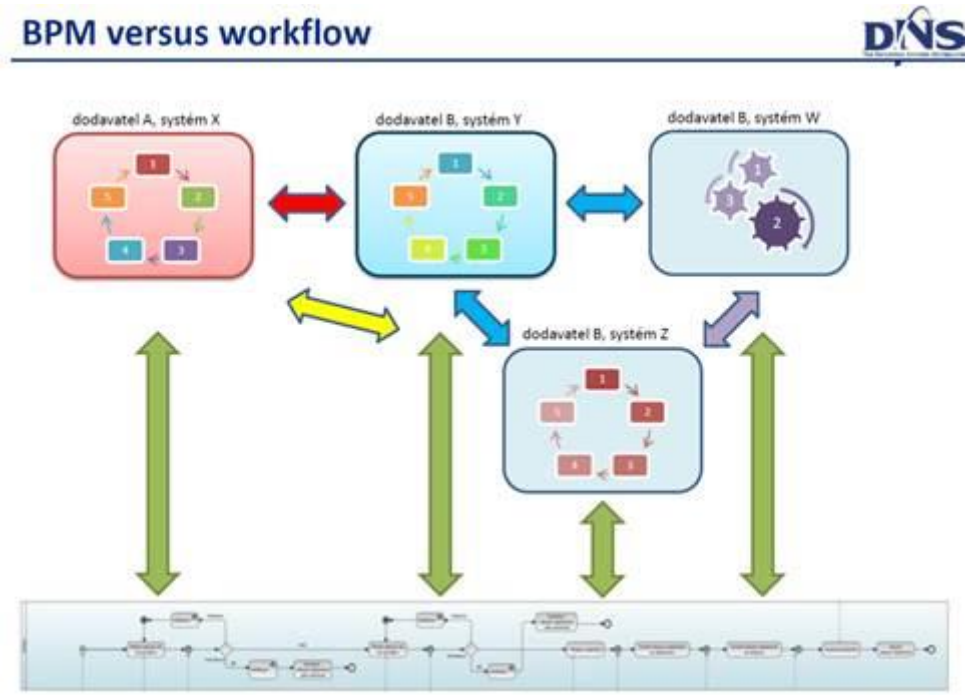


Principy BPM:

- Efektivní implementace a nasazení procesů ve firmě
- Přehledný diagram procesů
- Monitoring procesů
- Optimalizace procesů
- Zefektivňování procesů

Workflow

Workflow znamená automatizaci celého nebo části podnikového procesu, během kterého jsou dokumenty, informace nebo úkoly předávány od jednoho účastníka procesu k druhému podle sady procedurálních pravidel tak, aby se dosáhlo nebo přispělo k plnění celkových/globálních podnikových cílů.



Social business

Social Business využívá principy sociálních médií aby interně zlepšil organizaci (podnik) tak, aby byla více čiperná, průhledná a zapojená (nimble, transparent, engaged). Organizace, která implementuje moderní technologie (web 2.0) společně s organizačními, kulturními a procesními změnami, aby zvýšila svůj výkon a propojení s globálním ekonomickým prostředím.

Sociální podnik

- Hlubší vztahy se zákazníky, zaměstnanci, partnery, dodavateli
- Větší organizační transparentnost a agilitu
- Větší produktivitu a spokojenost zaměstnanců
- Větší zapojení a zpětnou vazbu od zákazníků
- Zrychlené inovace
- **KOMPETITIVNÍ VÝHODU**

Social business

- Naslouchání trhu, hledání advokátů (marketing, péče o zákazníky)
- Social je součástí procesů, propojení uvnitř i vně podniku (vývoj produktů a služeb)
- Vytváření komunit, rychlé drobné reakce (lidské zdroje, provoz, kancelář)

Tři roviny social business

- Spolupráce (nástroje social media)
  - Zapojení zaměstnanců, partnerů, zákazníků
  - Zrychlené generování nápadů
  - Rychlejší a lepší rozhodování
  - Lepší spolupráce
- Pochopení (analytické nástroje)
  - Směrování pozornosti, filtrování, polarizace
  - Pochopení vzorů chování, nálad
  - Metriky adopce a chování
- Transformace (nástroje procesní integrace)
  - Efektivita a zrychlování procesů
  - Rychlejší zapojení lidí
  - Podniková kultura inovace

1.2.12. Outsourcing IT, ITaaS, cloud

Outsourcing IT

Znamená, že firma vyčlení různé podpůrné a vedlejší činnosti a svěří je smluvně jiné společnosti čili sub-kontraktorovi, specializovanému na příslušnou činnost fáze outsourcingového procesu:

1) rozhodnutí o outsourcingu (outsourcovat ty činnosti, které nejsou pro podnik činnostmi strategickými),
2) detailní analýza části podniku určené pro outsourcing (slouží pro porovnání současných vlastních nákladů a dosavadní úrovně služeb s parametry nabízenými externí firmou – tzv. interní audit),
3) definice rozhraní podnik/poskytovatel (konkretizace požadované služby a určení návaznosti procesů na externě zajišťované činnosti)
4) výběr dodavatele.

ITaaS

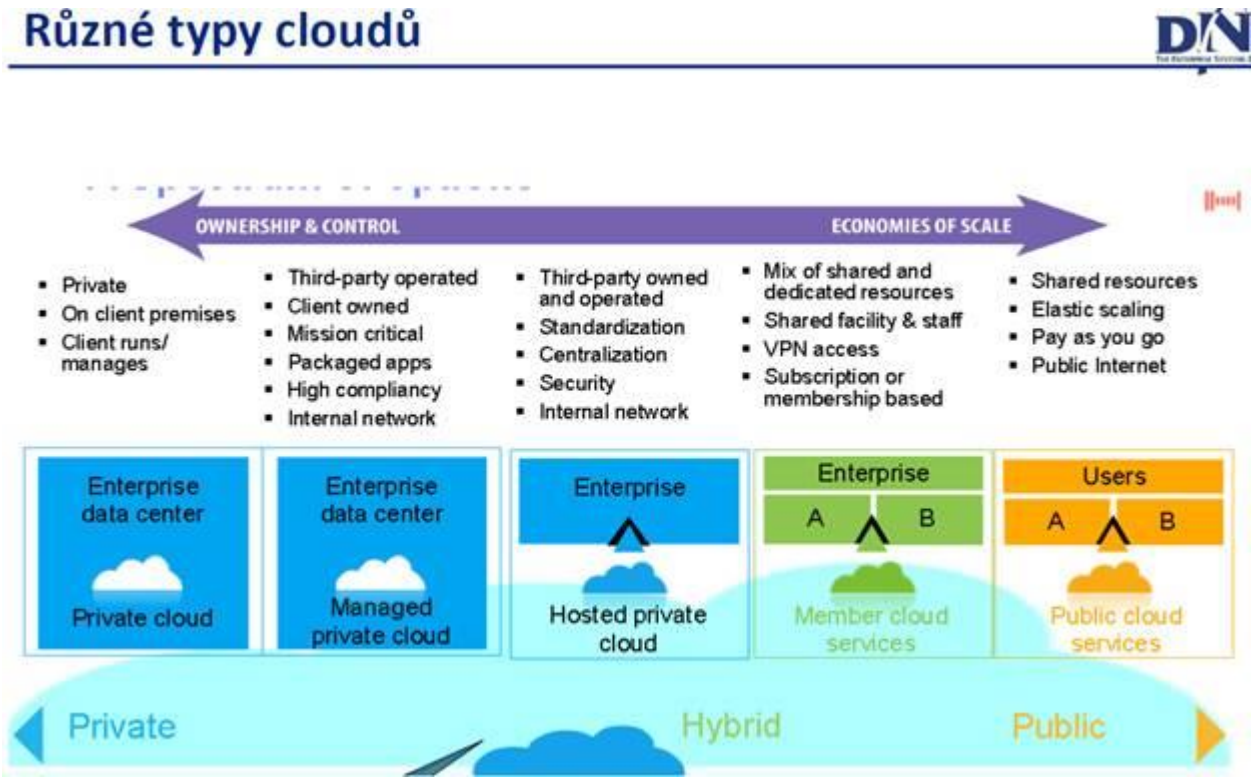
IT as a Service – outsourcing IT, IT jako služba – je umožněno díky cloudu. Všeobjímající termín, který zahrnuje všechny modely uvedené níže.

Cloud

Na Internetu založený model vývoje a používaní počítačových technologií

Distribuční model. Model se zabývá tím, co je v rámci služby nabízeno, obvykle software nebo hardware či jejich kombinace.

- **IAAS** — infrastruktura jako služba (z "Infrastructure as a Service") — v tomto případě se poskytovatel služeb zavazuje poskytnout infrastrukturu. Typicky se jedná o virtualizaci. Hlavní výhodou tohoto přístupu je to, že se o veškeré problémy s hardwarem stará poskytovatel. Na druhou stranu je někdy velice těžké toto akceptovat vzhledem k tomu, že hardware se bere jako něco, co vlastníme, na co můžeme sáhnout a jsme za to zodpovědní. IAAS je vhodné pro ty, kteří vlastní software (či jejich licence) a nechtějí se starat o hardware.
- **PAAS** — platforma jako služba (z "Platform as a Service") — poskytovatel v modelu PAAS poskytuje kompletní prostředky pro podporu celého životního cyklu tvorby a poskytování webových aplikací a služeb plně k dispozici na Internetu, bez možnosti stažení softwaru. To zahrnuje různé prostředky pro vývoj aplikace jako IDE nebo API, ale také např. pro údržbu. Nevýhodou tohoto přístupu je proprietární uzamčení, kdy může každý poskytovatel používat např. jiný programovací jazyk. Příkladem poskytovatelů PAAS jsou Google App Engine nebo Force.com (Salesforce.com).
- **SAAS** — software jako služba (ze "Software as a Service") — aplikace je licencována jako služba pronajímaná uživateli. Uživatelé si tedy kupují přístup k aplikaci, ne aplikaci samotnou. SaaS je ideální pro ty, kteří potřebují jen běžné aplikační software a požadují přístup odkudkoliv a kdykoliv. Příkladem může být známá sada aplikací Google Apps, nebo v logistice známý systém Cargopass.



Veřejný cloud (Public cloud): Cloud tak, jak je dnes nejčastěji chápán – tedy jako poskytování služeb IT (IaaS, PaaS, SaaS) prostřednictvím internetu třetí stranou, přičemž je zajištěna vysoká škálovatelnost a účtování podle využívaných zdrojů. Sdílen navzájem nesouvisejícími uživateli.

Soukromý cloud (Private cloud): Infrastruktura poskytující stejné služby jako veřejný cloud, ale pouze jedné organizaci. Je organizací většinou vlastněn a administrován. Aby bylo možno infrastrukturu označit za soukromý cloud, musí splňovat podmínku vysoké škálovatelnosti; někteří výrobci kladou důraz rovněž na schopnost účtování využívaných zdrojů jednotlivým složkám organizace.

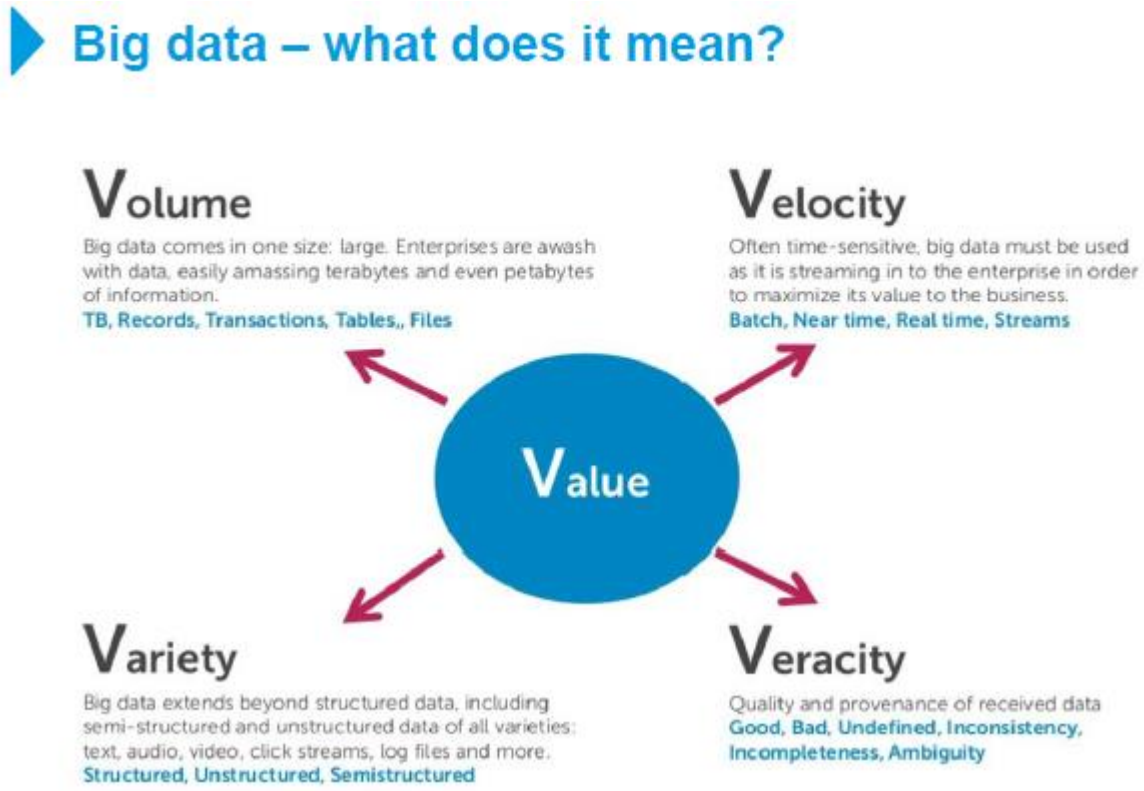
Komunitní cloud (Community cloud): Cloud využívaný definovanou komunitou, například spolupracujícími firmami, komunitou vývojářů určitého projektu apod.

Hybridní cloud: Cloud složený z více různých cloudů, např. několika veřejných a soukromého (jejich **propojení**). Organizace může typicky využívat různé typy vzájemně propojených služeb od různých cloudových poskytovatelů.

1.2.13. Aktuální témata podnikového IT – gidata, social, mobile, analytics

Bigdata

- Např. Twitter (12TB tweetů) nebo Monitoring telefonních hovorů v Číně
- Dat je moc, nelze udělat globální dotaz
- Nejde analyzovat všechna data
- Analýzu nelze provést v reálném nebo rozumném čase



Volume

- Dat je mnoho a jejich zpracování trvá příliš dlouho

Velocity

- Obrovské množství dat je potřeba analyzovat v krátkém čase

Variety

- Data mají různou strukturu a jsou nestrukturovaná: video, senzorická data, text…

Veracity

- Problematická důvěra v data

Využití Big Data

- Marketing
- Social network
- Analýza
- Skladování (dlouhodobé)

NoSQL databáze

- Obecný název pro řadu databázových aplikací zaměřených na vysoký výkon a škálovatelnost pro zpracování velikých objemů dat
- Jednoduché použití
- Horizontální škálování
- Hlavní kategorie:
    1. Key-value
    2. Columns
    3. Graph
    4. Document
- noSQL databáze pracují na odlišném principu než RDBMS – např. agregace dat do dokumentů pomocí JSON (Java Script Object Notation)
- škálování standardních databází má své limity, noSQL databáze škálují jako skutečný distribuovaný systém
- wide column store – Hadoop
- document store – MongoDB, Crouchbase…
- key-value – Berkley DB

ACID

- Atomicity, Consistency, Isolation, Durability

Social

POZN. Viz Social Business???

Trend Web 2.0…

Social Business využívá principy sociálních médií aby interně zlepšil organizaci (podnik) tak, aby byla více čiperná, průhledná a zapojená (nimble, transparent, engaged). Organizace, která implementuje moderní technologie (web 2.0) společně s organizačními, kulturními a procesními změnami, aby zvýšila svůj výkon a propojení s globálním ekonomickým prostředím.

Sociální podnik

- Hlubší vztahy se zákazníky, zaměstnanci, partnery, dodavateli
- Větší organizační transparentnost a agilitu
- Větší produktivitu a spokojenost zaměstnanců
- Větší zapojení a zpětnou vazbu od zákazníků
- Zrychlené inovace
- **KOMPETITIVNÍ VÝHODU**

Social business

- Naslouchání trhu, hledání advokátů (marketing, péče o zákazníky)
- Social je součástí procesů, propojení uvnitř i vně podniku (vývoj produktů a služeb)
- Vytváření komunit, rychlé drobné reakce (lidské zdroje, provoz, kancelář)

Tři roviny social business

- Spolupráce (nástroje social media)
    - Zapojení zaměstnanců, partnerů, zákazníků
    - Zrychlené generování nápadů
    - Rychlejší a lepší rozhodování
    - Lepší spolupráce
- Pochopení (analytické nástroje)

- Směrování pozornosti, filtrování, polarizace
- Pochopení vzorů chování, nálad
- Metriky adopce a chování

- Transformace (nástroje procesní integrace)
  - Efektivita a zrychlování procesů
  - Rychlejší zapojení lidí

Podniková kultura inovace

Mobile

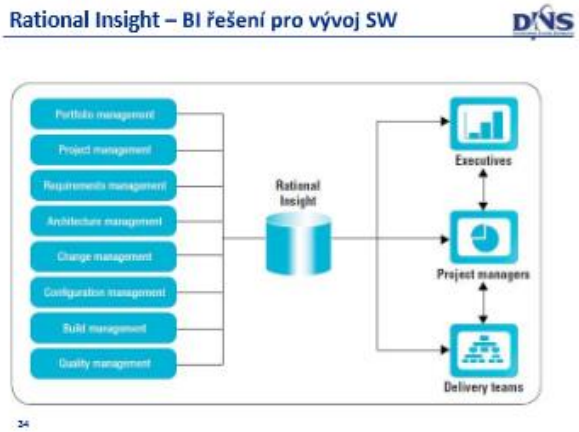
BYOD – bring your own device – uživatel smí ve firmě využít vlastní zařízení
Důraz na SOA, zejména na RESTful služby (RESTful = webová služba splňující požadavky REST protokolu)
Vytváření mobilních aplikací pro firemní použití -> např. i analytics nástroje

Analytics

Potřeba analyzování sebraných statistik podle provozu procesů -> grafy, přehledy -> např. kokpit, nástroje BI (BI metriky, BA souhrn schopností, dovedností a znalostí), dashboardy (KPI = Key Performance Indicators)

BI

(Business Inteligence) - dovednosti, znalosti, technologie, aplikace, kvalita, rizika, bezpečnostní otázky a postupy používané v podnikání pro získání lepšího pochopení chování na trhu a obchodních souvislostech



Datamining

Data mining ([dejta majnyn], angl. *dolování z dat* či *vytěžování dat*) je analytická metodologie získávání netriviálních skrytých a potenciálně užitečných informací z dat. Někdy se chápe jako analytická součást *dobývání znalostí z databází* (Knowledge Discovery in Databases, KDD),[1] jindy se tato dvě označení chápou jako souznačná.

Data mining se používá v komerční sféře (například v marketingu při rozhodování, které klienty oslovit dopisem s nabídkou produktu), ve vědeckém výzkumu (například při analýze genetické informace) i v jiných oblastech (například při monitorování aktivit na internetu s cílem odhalit činnost potenciálních škůdců a teroristů).

2. DB

2.1. DB2

2.1.1. „Vnitřní“ programovací konstrukce (Embedded SQL) - procedurální prostředky v rámci jazyka SQL

Embedded SQL

Přímý přístup programovacího jazyka do databázových struktur – **jazyk (překladač) je obohacený o konstrukce pro SQL**

- SQL dotazy jsou psány přímo ve zdrojovém kódu. Syntaxe SQL je přizpůsobena syntaxi daného programovacího jazyka.
- Před kompilací programu se musí zdrojový kód **zpracovat preprocesorem Embedded SQL**, kdy SQL dotazy jsou nahrazeny odpovídajícím kódem programovacího jazyka.
- Nejčastěji v kombinaci s jazyky C/C++.

Podpora v databázích

Podporují klasický velké databázové systémy

- IBM DB2 (C/C++, Java, Cobol)
- Oracle (Cobol, *Pro*C* - embedded SQL Oraclu pro C/C++)
- PostgreSQL (C/C++)

Nepodporují

- MySQL
- Microsoft SQL Server (starší verze podporovaly C)

Příklad syntaxe

Oracle Embedded SQL v jazyce C:

```
{
    int a;
    /* ... */
    EXEC SQL SELECT salary INTO :a
             FROM Employee
             WHERE SSN=876543210;

    printf("The salary is %d\n", a);
}
```

Procedurální prostředky (rozšíření) SQL

- SQL bylo původně navrženo pro získávání dat z relačních databází. SQL je **deklarativní jazyk** a ne imperativní, jako například C nebo Java.

- Dodavatelům DB systémů možnosti SQL nedostačovaly a tak si začali implementovat vlastní procedurální rozšíření SQL.
- Příklad: kurzor

Motivace

- Šetření komunikačního kanálu
- Menší množství odesílaných povelů
- v jednom povelu je větší množství příkazů
- Podstatně menší objem přenesených dat - Data se zpracují na serveru bez přenosu na klienta
- Odlehčení klienta - Možnost ukládat a vykonávat kód na serveru

Na co si dát při návrhu pozor

- Hlídat na úrovni databáze všechny manipulace s daty, které ohlídat jdou
  - Cokoli jde zadat uživatelem špatně, bude zadáno špatně
- Integritní omezení, triggery
  - Později je čištění nekonzistentních dat namáhavé a opravy často nemožné
  - Lépe ohlídat vše

Procedurální rozšíření v SQL:1999

SQL:1999 standardizuje procedurální rozšíření. Rozšíření se jmenuje *SQL/PSM - SQL/Persisted Stored Modules*. **Nikdo to moc nedodržuje a všichni si dělají vlastní standardy/implementace**

- funkce a procedury - lze zapsat v SQL i v hostitelském programovacím jazyce
- řídící konstrukce - cykly, větvení, přiřazení

Podpora v databázích

SQL:1999 (SQL/PSM)

- IBM DB2
- MySQL
- PostgreSQL

Vlastní (proprietární) rozšíření

- Oracle (PL/SQL)
- PostgreSQL (PL/pgSQL)
- Microsoft (T-SQL)
- Sybase (T-SQL)

2.1.2. SQL, jazyk PL/SQL.

SQL - *Structured Query Language* (Strukturovaný dotazovací jazyk) je standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích

Rozdělení standardního SQL

SQL příkazy se dají rozdělit do několika kategorií podle toho co provádíte

Data definition language (DDL) neboli příkazy určené pro práci se strukturou databázových objektů. Nejčastěji tabulek.

CREATE - vytvoření

ALTER - změně

DROP - odstranění

RENAME - přejmenování

TRUNCATE - smazání, aniž by se data ukládala do koše

COMMENT - přidání komentáře

Data manipulation language (DML) neboli příkazy určené pro manipulaci s daty

SELECT - vybrání dat z databáze

INSERT - vložení

UPDATE - úprava nebo také editace či změna

DELETE - smazaní

MERGE - sloučení

Řízení transakcí. Jednotlivé příkazy DML můžete slučovat do transakcí, ale nemusíte.

COMMIT - slouží k potvrzení veškerých změn

ROLLBACK - provede rollback veškerých změn

SAVEPOINT - vytvoří časovou značku ke které se můžete vracet.

Data control language (DCL) neboli příkazy sloužící k přídání či odebrání opravnění k databázi a objektů v ní.

GRANT - přiřazení

REVOKE = odebrání

PL/SQL

PL/SQL přidává k jazyku SQL konstrukce procedurálního programování.

PL/SQL (Procedural Language/Structured Query Language) je procedurální nadstavba jazyka SQL od firmy Oracle založená na programovacím jazyku Ada.

PL/SQL je rozšíření jazyka SQL o procedurální rysy. Je specifické pro produkty firmy Oracle, procedurální rozšíření SQL produktů jiných firem se zpravidla navzájem liší. Výjimkou je ŠRBD DB2 společnosti IBM, který podporuje jak vlastní procedurální jazyk SQL PL, tak je plně kompatibilní s jazykem PL/SQL včetně datových typů. Základním stavebním kamenem PL/SQL je tzv. **PL/SQL blok**, který může být buď tělem triggeru, procedry a funkce, nebo samostatný. Struktura PL/SQL bloku je viz "základní konstrukce"

Základním stavebním kamenem v PL/SQL je **blok**. Program v PL/SQL se skládá z bloků, které mohou být vnořeny jeden do druhého. Obyčejně každý blok spouští jednu logickou akci v programu. Blok má následující strukturu:

Základní konstrukce

```
DECLARE
 deklarace
BEGIN
 výkonná část
EXCEPTION
 ošetření výjimek
END;
```

```
Komentáře:

-- komentář do konce řádky
/* komentář od do */
```

Pouze výkonná sekce je povinná, ostatní jsou doporučené. Jediné příkazy jazyka SQL, které jsou ve výkonné sekci **povolené, jsou SELECT, INSERT, UPDATE, DELETE** a několik dalších pro manipulaci s daty a pro kontrolu transakcí. **Definiční příkazy jazyka SQL jako CREATE, DROP nebo ALTER nejsou povoleny**. Avšak použití těchto příkazů lze pomocí direktivy EXECUTE IMMEDIATE "PŘÍKAZ". PL/SQL není citlivé na velikost písmen a mohou být použity komentáře ve stylu jazyka C.

Kurzor

- Kurzor je pracovní oblast obsahující data (výsledná množina, tzv. result set), které lze dále využívat prostřednictvím operací nad kurzory
- Existují implicitní a explicitní kurzory
    - Implicitní jsou jednořádkové SQL (INTO)
    - Explicitní můžeme deklarovat v části DECLARE pomocí klíčového slova CURSOR
- Práce s kurzory se podobá souborům

Procedury + funkce

Bloky příkazů jazyka PL/SQL lze pojmenovat a uložit ve spustitelné formě do databáze = procedury + funkce

- Jsou uloženy ve zkompilovaném tvaru v databázi.
- Mohou volat další procedury či funkce, či samy sebe.
- Lze je volat ze všech prostředí klienta.

Funkce, na rozdíl od procedury, vrací jedinou hodnotu (procedura může vracet hodnot více, resp. žádnou).

Triggery

- uživatelsky definovaný blok PL/SQL sdružený s určitou tabulkou. Je implicitně spuštěn (proveden), jestliže je nad tabulkou prováděn aktualizační příkaz

Anonymous Blocks (jenom BEGIN až END bez názvu, takže zapomenout EXEC)

Packages

- Programový balík je sdružením řady funkcí a procedur s vlastním jmenným prostorem a vlastním persistentním prostorem pro proměnné v rámci jedné session
- Umožňuje uchovávat hodnoty v rámci session pro řadu procedur a funkcí
- Ne náhodná analogie s objekty

Nested Tables

- Představují pole - množina (set, bag) v některých programovacích jazycích
- Lze ukládat tyto tabulky v databázových tabulkách (oboustranná kompatibilita)
- Deklarace:
  DECLARE TYPE ntable IS TABLE
  OF element_type;
- Typ prvku může být lib. PL/SQL typ mimo odkazu (REF) a kurzoru (CURSOR)

Varrays

- variable-size array (dynamické pole) – tzv. varrays odpovídají klasickým dynamickým polím, uchovávají definovaný počet hodnot,
- pomalejší přístup SQL nástroji než k nestedtables

Loops (FOR/WHILE)

```
LOOP
pocet:= pocet +1
IF pocet =100 THEN EXIT;
END IF;
END LOOP;
```

IF

```
IF podmínka THEN příkazy_1;
ELSIF podmínka THEN příkazy_2;
.
.
.
ELSE příkazy_n;
END IF;
```

CASE

```
CASE proměnná
WHEN výraz_1 THEN příkazy_1;
WHEN výraz_2 THEN příkazy_2;
WHEN výraz_n THEN příkazy_n;
ELSE příkazy_n+1
END CASE
```

2.1.3. Kurzory – definice, klasifikace, použití kurzorů.

Definice

- Kurzor je abstraktní datový typ umožňující procházet záznamy vybrané dotazem, který je s kurzorem spojen.
- Prostředek pro získání informace z databáze a předání do programu v jazyce PL/SQL
- SELECT co vrací více řádků, přes kurzor je možné výsledky procházet

Klasifikace

- explicitní kurzor
  - nutno deklarovat, otevřít, načíst data a uzavřít DECLARE CURSOR <jméno kurzoru>IS <dotaz>; OPEN <jméno kurzoru>; FETCH <jméno kurzoru>INTO <jméno proměnné1>, <jméno proměnné2>, …; CLOSE <jméno kurzoru>;
- implicitní kurzor
  - je deklarován a prováděn přímo v těle programu
  - v tomto typu kurzoru jsou povoleny pouze příkazy SQL, které vrací jednotlivé řádky nebo nevrací žádné řádky,
  - příkazy SELECT, UPDATE, INSERT a DELETE obsahují implicitní kurzory
  - musí se shodovat datové typy sloupců a proměnných
  - implicitní kurzor SELECT musí vracet pouze jeden řádek (SELECT INTO !) SELECT <jméno sloupce 1>, <jméno sloupce 2> INTO <jméno proměnné 1>, <jméno proměnné 2> FROM ... ;

Atributy kurzoru

- **cursor%FOUND** obsahuje záznamy?
- **cursor%NOTFOUND** neobsahuje záznamy?
- **cursor%ISOPEN** je otevřený?
- **cursor%ROWCOUNT** dosud zpracováno ř.

Použití kurzorů

Když je potřeba iterovat přes hromadu položek a pro každou položku něco provést (tedy ne pro všechny najednou, ale pro každou zvlášť)

- v triggerech
- v uložených procedurách

příklad:

```
DECLARE
  tmp osoby%ROWTYPE;
  CURSOR plist IS SELECT * FROM osoby;
BEGIN
  OPEN plist;
  LOOP
    FETCH plist INTO tmp;
    EXIT WHEN plist%NOTFOUND;
    dbms_output.put_line(plist%ROWCOUNT||'.       '||tmp.jmeno||'
'||tmp.prijmeni);
  END LOOP;
  CLOSE plist;
END;
```

- **Příklad implicitního kurzoru:**

```
DECLARE
       Sum NUMBER;
BEGIN
   SELECT SUM(salary) INTO Sum FROM Employee
END;
```

- Dále např. automatické číslování v SŘBD Oracle (Autoinkrement)

Jaká konstrukce se používá pro zpracování víceřádkových SELECT v PL/SQL a jaké jsou její dva typy?

Používá se konstrukce CURSOR – předává programu PL/SQL info z databáze

Explicitní kurzor - nutno deklarovat, otevřít, načíst data a uzavřít (vše dělá programátor)

Implicitní kurzor - je automaticky deklarován Oraclem a prováděn přímo v těle programu při použití SQL dotazu. Implicitní je proto, že ho uživatel nemusí nijak deklarovat. **Pokud např. Select vrací více řádku, se kterými chceme dále pracovat, je nutné použít explicitní kurzor**

2.1.4. Uložené procedury, funkce a balíky procedur a funkcí, kompilace, spouštění.

Podprogram je pojmenovaný blok, který může být opakovaně volán a může přebírat aktuální parametry. Typy podprogramů jsou **funkce a procedury**. Procedury a funkce lze sdružovat do logických celků – balíků (package).

Všechny anonymní bloky PL/SQL lze natrvalo uložit do databáze, a to buď ve tvaru procedury, nebo funkce.

Uložené podprogramy – procedury a funkce

Procedury a funkce mohou být trvale uloženy do DB a mohou být použity jakoukoli aplikací, která s databázovým strojem komunikuje. Jednou přeložený a uložený program patří mezi databázové objekty a může být referován libovolným počtem aplikací.

Uložené podprogramy mohou být samostatné, nebo součástí balíku.

Podprogramy lze volat z:

- DB triggerů
- Jiných uložených podprogramů
- Aplikačních programů zapsaných ve vyšším programovacím jazyce, pro nějž existuje předkompilátor (ORACLE Pro)
- Interaktivně (SQL*Plus) : EXECUTE jmeno_procedury(parametry)

Uložené procedury a funkce mohou mít parametry, které mohou být:

- IN - vstupní
- OUT - výstupní
- IN OUT – vstupní i výstupní

Uložená procedura

Procedura se vstupními parametry:

```
CREATE OR REPLACE PROCEDURE nastav_plat (id IN NUMBER, novy_plat IN
NUMBER) AS
   zam_plat NUMBER(5,2);
   nema_plat EXCEPTION;
BEGIN
   SELECT plat INTO zam_plat FROM osoby WHERE os_cislo = id;
   IF zam_plat IS NULL THEN
      RAISE nema_plat;
   END IF;
EXCEPTION
   WHEN nema_plat THEN
      UPDATE osoby SET plat = novy_plat WHERE os_cislo = id;
      COMMIT;
END;
```

Pro zrušení (smazání) uložené procedury použijeme příkaz:

```
DROP PROCEDURE nastav_plat;
```

Uložená funkce

Způsob vytvoření uložené funkce (s parametry) ukazuje následující příklad:

```
CREATE OR REPLACE FUNCTION secti(a IN INTEGER, b IN INTEGER) RETURN
INTEGER AS
BEGIN
      RETURN (a + b);
END;
```

Pro zrušení (smazání) uložené funkce použijeme příkaz:

```
DROP FUNCTION secti;
```

Uložené balíky

Uložené balíky procedur a funkcí slouží ke sdružení logicky spolu souvisejících procedur a funkcí, ale i typů a objektů. Mohou obsahovat i globální proměnné, jejichž platnost je omezena délkou aktuálního spojení s databází.

Definice balíku představuje definici rozhraní balíku (deklarace typů, proměnných, konstant, podmínek definujících nestandardní stavy, kurzorů, podprogramů dostupných zvenčí)(co tam je za funkce parametry, ale bez těla) a těla balíku (implementuje specifikaci)(normálně napsané procedury včetně implementace).

Př. rozhraní:

```
CREATE OR REPLACE PACKAGE arithmetic AS
      usage INTEGER;

      FUNCTION add(a IN INTEGER, b IN INTEGER) RETURN INTEGER;

      FUNCTION sub(a IN INTEGER, b IN INTEGER) RETURN INTEGER;

      PROCEDURE inc(a IN OUT INTEGER);
END;
```

Př. Těla:

```
CREATE OR REPLACE PACKAGE BODY arithmetic AS

      FUNCTION add(a IN INTEGER, b IN INTEGER) RETURN INTEGER IS
      BEGIN
            usage := usage + 1;
            RETURN (a + b);
      END;

      FUNCTION sub(a IN INTEGER, b IN INTEGER) RETURN INTEGER IS
      BEGIN
            usage := usage + 1;
            RETURN (a - b);
      END;

      PROCEDURE inc(a IN OUT INTEGER) IS
      BEGIN
            usage := usage + 1;
            a := a + 1;
      END;

BEGIN
      usage := 0;
END;
```

Kompilace

- Pokud je volána procedura/funkce ve stavu INVALID, kompiluje se automaticky.
    - Pokud se kompilace nezdaří, dojde k výjimce.
- Ruční kompilace
- `ALTER PROCEDURE[FUNCTION] jmproc COMPILE;`
    - Pokud se kompilace nezdaří, dojde k výjimce
- SQL*Plus a chyby kompilace – pomocí něj můžeme zjistit, jaké chyby se vyskytly při kompilaci. Pokud uložení procedury/funkce neproběhlo bez chyb, nelze ji používat a je nutné ji opravit. Pomocí SQL*Plus příkazů:
    - SHOW ERROR – vypíše popis poslední chyby, na kterou při ukládání (kompilaci) narazil
    - SHOW ERR typ jméno – např. SHOW ERR FUNCTION F – pro uvedený objekt
    - Pomocí dotazu na tabulku USER_ERRORS

Spouštění

Proceduru můžeme zavolat (spustit různými způsoby:

- Pomocí direktivy EXEC – EXEC nastav_plat(123, 10000);
- V těle jiného PL/SQL bloku

```
BEGIN
...
Nastav_plat(123, 10000);
...
END;
```

Funkci můžeme volat také dvěma způsoby:

- V příkazu SELECT – SELECT secti(2, 3) FROM dual;

- V těle jiného PL/SQL bloku

```
DECLARE
      a INTEGER;
      b INTEGER;
BEGIN
      a := 5;
      b := secti(a, 2);
      dbms_output.put_line('Vysledek : '||b);
END;
```

Při volání funkcí/procedur z balíků používáme tečkovou notaci (package.funkce()) pro kvalifikaci jejich jména.

Zabalený podprogram lze volat z db triggeru, jiného uloženého programu, aplikace napsané pro některý z předkompilátorů, standardních klientských nástrojů (SQL*Plus)

Standardní balík ***STANDARD*** = definuje prostředí PL/SQL.

2.1.5. Aktivní databáze – Oracle triggery, klasifikace a spouštění triggerů.

V řadě IS, které běží nad nějakou databází, potřebujeme v případě vzniku nějaké události (např. modifikujeme řádek v nějaké tabulce) automaticky spustit příkaz, který provede nějaké operace. K tomuto účelu slouží triggery (spouštěče). Trigger je speciální typ uložené procedury, která se aktivuje při splnění nějaké podmínky na serveru (aktualizace dat, události spojené s DB nebo session).

Triggery jsou vlastně procedury, které automaticky volá (spouští) SŘBD při definované události. Touto událostí může být buď vložení (INSERT), rušení (DELETE), nebo aktualizace (UPDATE) záznamu v tabulce. Triggery bývají obvykle volány buď:

- Před specifikovanou událostí – **BEFORE**
- Po specifikované události – **AFTER**
- **INSTEAD OF** - místo specifikované události

Od uložených procedur a funkcí se odlišují tím, že jsou spuštěny *při modifikaci tabulky*, definují se pouze pro db tabulky a nepřijímají argumenty a lze je spustit jen při zmiňovaných DML příkazech.

U triggeru lze rovněž specifikovat podmínku, kdy má být vykonáno jeho tělo (PL/SQL blok) a to použitím klauzule WHEN.

Triggery jsou plně zkompilované po spuštění příkazem CREATE TRIGGER a po uložení procedurálního kódu v systémovém katalogu.

Trigger lze také

- deaktivovat (zakázat) - ALTER TRIGGER jmeno DISABLE;
- aktivovat – ALTER TRIGGER jmeno ENABLE;
- zrušit – DROP TRIGGER jmeno;

Výhody triggerů jsou:

- nepovolí neplatné datové transakce, zajišťují komplexní bezpečnost, zajišťují referenční integritu přes všechny uzly db, zajišťují audit (sledování), spravují synchronizaci tabulek, zaznamenávají statistiku často modifikovaných tabulek.

Klasifikace triggerů

Příkazové triggery (statement level)

Triggery se spustí nad tabulkou bez ohledu na to, kolik tabulka obsahuje řádek. Např. pokud chci logovat změny provedené v DB, která obsahuje tabulky PRACOVISTE, ZAMESTNANCI do tabulky LOGY. Po každé operaci I, U, D nad tabulkami PRAC a ZAM se do tabulky LOGY vloží záznam o modifikaci tabulky a typu modifikace.

```
CREATE TRIGGER tai_pracoviste
  AFTER INSERT ON pracoviste
BEGIN
  INSERT INTO logy VALUES ( 'PRACOVISTE', 'I');
```

```
END;
```

Řádkové triggery – FOR EACH ROW

Trigger se spouští jednou pro každý řádek tabulky. Např. z předchozího příkladu chci mít u každého záznamu informaci, kdy byl zadán a kdo jej zadal.

```
CREATE TRIGGER trbi_pracoviste
  BEFORE INSERT ON pracoviste
  FOR EACH ROW
BEGIN
  :new.zadal := user;
  :new.datum := sysdate;
END;
```

Vlastní obsluhu události lze nadefinovat v PL/SQL bloku. Uvnitř PL/SQL bloku (a také klauzuli WHEN) řádkového triggeru se lze odkazovat na původní a nový záznam pomocí pseudoproměnných **:new** (obsahuje vkládaný záznam) a **:old** (původní záznam). Je zřejmé, že při vkládání nového záznamu není definována :old a při mazání :new. Jejich jména lze předefinovat v klauzuli **REFERENCINGOLD AS.**

Business rules triggery

Triggery jsou také často používány při realizaci tzv. business rules, tj. integritních omezení specifických pro danou oblast použití. Např. použijeme – studenti si mohou zapsat max 20 kreditů za semestr. Pokud při vkládání dat do tabulky ZAPIS překročíme maximální povolenou hodnotu, dostaneme chybové hlášení – a to zařídí business trigger.

Zápis

```
CREATE OR REPLACE TRIGGER jméno
  BEFORE | AFTER | INSTEAD OF
  DELETE | INSERT | UPDATE OF cols
  ON tabulka
  [ způsob odkazování ]
  [ FOR EACH ROW ]
  [ WHEN ( podmínka ) ]
AS pl/sql kód
```

Omezení triggerů

- BEFORE a AFTER triggery nelze specifikovat nad pohledy
- V BEFORE triggerech není možné zapisovat do :old záznamů
- V AFTER triggerech nelze zapisovat do :old ani do :new záznamů
- INSTEAD OF triggery pracují jen s pohledy, mohou číst :old i :new, ale nemohou zapisovat ani do jednoho
- Nelze kombinovat INSTEAD OF a UPDATE
- Nelze definovat trigger nad LOB atributem

Dvě zásadní omezení

- Nelze použít transakce, pokud je zpracovávána jiná transakce, tedy prakticky nelze použít transakce vůbec
- Není možné sledovat ani modifikovat data v tabulce, která způsobila vyvolání DML triggeru -> jediné známé řešení je zrcadlení tabulek

Sémantika Oracle Triggerů

- Spouštění probíhá okamžitě při události, nelze je spustit explicitně (např. uživ. Příkazem)
- Vnořené spouštění triggerů - činnost triggeru může vyvolat jiný trigger - kontext aktuálního triggeru se uloží, začne se vykonávat nový trigger, pak se zas obnoví a pokračuje ten původní
- Maximální hloubka zanoření je 32, pak to hodí výjimku

Doplňující pojmy

události (events)

- změna stavu databáze
- časové události
- externí, definované aplikací

podmínky (conditions)

- databázový predikát
- databázový dotaz

akce (actions)

- libovolná manipulace s daty

- transakční příkazy
- pravidla zpracování
- externí procedury

Spouštění triggerů

- **Vykonání triggeru**
  - Okamžité (immediate)
    - *Před událostí*
    - *Po události*
    - *Namísto události*
  - Odložené (deferred)
    - *Na konci transakce*
    - *Po uživatelském příkazu*
    - *Následkem uživatelského příkazu*
  - Oddělené (detached)
    - *V kontextu samostatné transakce vypuštěné z počáteční transakce poté, co nastala událost*
    - *Možné kauzální závislosti počáteční a oddělené transakce*

- **Vykonání akce**
  - Okamžité (immediate)
    - *Následuje ihned po vyhodnocení podmínky*
  - Odložené (deferred)
    - *Akce je odsunuta na konec transakce*
    - *Akci vyvolá uživatelský příkaz*
  - Oddělené (detached)
    - *Probíhá v kontextu samostatné transakce vypuštěné z počáteční transakce ihned po vyhodnocení podmínky*
    - *Možné kauzální závislosti počáteční a oddělené transakce*

2.1.6. Transakce, dvoufázový uzamykací protokol, detekce uváznutí.

Transakce

Databázové transakce musí splňovat tzv. vlastnosti ***ACID***

- ***A*** *- Atomicity*
  Databázová transakce je jako operace dále nedělitelná (atomická). Provede se buď jako celek, nebo se neprovede vůbec (a daný databázový systém to dá uživateli na vědomí, např. chybovou hláškou).
- ***C*** *- Consistency - konzistentnost*
  Při a po provedení transakce není porušeno žádné integritní omezení.
- ***I*** *- Isolation - izolovanost*
  Operace uvnitř transakce jsou skryty před vnějšími operacemi. Vrácením transakce (ROLLBACK) není zasažena jiná transakce, jinak i tato musí být vrácena. V důsledku tohoto chování může dojít k tzv. řetězovému vrácení (cascading rollback).
- ***D*** *- Durability - trvalost*
  Změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi a již nemohou být ztraceny.

Globální vs. lokální

- Lokální transakce probíhá pouze na jediném uzlu.
- Globální (distribuovaná) transakce přesahuje rozsah jednoho uzlu.

Stavy transakce

- Aktivní - od počátku provádění transakce
- Částečně potvrzený - stav po provedení poslední operace transakce
- Chybný - nelze pokračovat v normálním průběhu transakce
- Zrušený - nastane po skončení operace ROLLBACK
- Potvrzený - po úspěšném vykonání COMMIT

Prováděné operace

Pro práci s transakcemi je nutné zavést následující operace:

- BEGIN - začátek transakce
- COMMIT - ukončení transakce a uložení dosažených výsledků do databáze
- ROLLBACK - odvolání změn - není-li definován savepoint, (místo, po které lze provedené změny vrátit zpět) tak návrat do stavu před započetím vykonávání transakce

Optimistické vs. pesimistické zamykání

- U ***pesimistického zpracování*** se v jeho průběhu změny zaznamenávají do dočasných objektů (například a nejčastěji: do řádků tabulek s příznakem dočasných dat, platných jen po dobu transakce) a teprve po přesunu/změně dat se odznačí příznak dočasnosti a data se stanou platnými. (Tento způsob se dá přibližně připodobnit přepisu souboru, při kterém se nejdříve nová verze souboru nakopíruje pod dočasným jménem a teprve poté se tento soubor přejmenuje za starý a tím ho nahradí.)
- U ***optimistického zpracování*** se (optimisticky) předpokládá, že při transakci nenastane chyba a nebude třeba ji vrátit zpět (přestože tato možnost je zachována). Měněné záznamy v tabulkách jsou při optimistickém zpracování transakce zapisovány „natvrdo“, současně s tím se však vytváří tzv. rollback log coby seznam SQL příkazů, které dokáží prováděné změny vrátit zpět. V případě, že při transkaci dojde k nějaké nezotavitelné chybě, tento log se provede a transakce (aby dodržela pravidlo atomicity) skončí ve výchozím stavu s chybou. Naopak, na konci transakce, při které k žádné takové chybě nedošlo, se rollback log maže.

Žurnály

Jsou záznamy, které uchovávají informace o průběhu transakcí a slouží k zotavení po vzniklé chybě. Žurnály musí být v každém uzlu a obsahují záznamy o historii každé transakce.

Dvoufázový protokol (2PL)

Dvoufázová transakce v první fázi zamyká vše co je potřeba a od prvního odemknutí (druhá fáze) již jen odemyká co měla zamčeno (již žádná operace LOCK). Tedy transakce musí mít všechny objekty uzamčeny předtím, než nějaký objekt odemkne. Dá se dokázat, že pokud jsou všechny transakce v dané množině transakcí dobře formované a dvoufázové, pak každý jejich legální rozvrh je uspořádatelný. Dvoufázový protokol zajišťuje uspořádatelnost, ale ne zotavitelnost ani bezpečnost proti kaskádovému rušení transakcí nebo uváznutí.

Striktní dvoufázový protokol (S2PL)

Problémy 2PL jsou nezotavitelnost a kaskádové rušení transakcí. Tyto nedostatky lze odstranit pomocí striktních dvoufázových protokolů, které uvolňují zámky až po skončení transakce (COMMIT). Zřejmá nevýhoda je omezení paralelismu. 2PL navíc stále nevylučuje možnost deadlocku.

Konzervativní dvoufázový protokol (C2PL)

Rozdíl oproti 2PL je ten, že transakce žádá o všechny své zámky, ještě než se začne vykonávat. To sice vede občas k zbytečnému zamykání (nevíme co přesně budeme potřebovat, tak radši zamkneme víc), ale stačí to již k prevenci uváznutí (deadlocku).

Detekce uváznutí a zotavení

- Uváznutí se detekuje pomocí čekacího grafu
  - Vrcholy jsou transakce Ti
  - Orientovaná hrana Ti → Tj značí, že Ti čeká, až Tj odemkne datovou položku
  - Je-li v čekacím grafu cyklus, došlo k uváznutí
- Hledá se takový plán transakcí, aby se co nejmíň kryly a tak, aby byl dodržen princip ACID (hlavně Isolation), když se takový plán povede najít, nazývá se uspořadatelný
- ***Well formed transakce*** (správně zamykat a odemykat)
- ***Když se zjistí uváznutí***
  - Je nutno nalézt obětní transakci a vnutit jí abort (a tím i obnovu dat). Obětuje se obvykle nejmladší transakce, tj. ta, která ještě neudělala mnoho změn
  - Transakce mohou stárnout, bude-li za oběť vybírána vždy nejmladší transakce. Proto je vhodné do kritéria výběru obětí zahrnout i počet transakcí provedených návratů.
  - Která data se ale mají obnovovat?
    - ***Totální obnova*** transakci úplně zruší, data se vrátí do počátečního stavu, a transakce se restartuje. To může být velmi nákladné
    - Efektivnější je, když se transakce "vrací postupně" do stavu, kdy uváznutí zmizí. Tento postup je ale náročný na evidenci kroků a změn transakcí provedených: metoda kontrolních bodů (***checkpointing***) – konzistentní mezistavy

Zajištění uspořadatelnosti pomocí pořadových čísel transakcí

Neboli také metoda časových razítek.

Transakce vyvolá write W(x):

- TSR(x) > TS(t): zápis do „později přečtené“ paměti > ROLLBACK
- TSW(x) > TS(t): zápis do „později přepsané“ paměti > ROLLBACK
- jinak proveď zápis

Transakce vyvolá read R(x):

- TSW(x) > TS(t): čtení z „později zapsané“ paměti > ROLLBACK
- jinak read

Využití časových razítek - time stamp viz stará přednáška 5

2.1.7. Optimalizace dotazu, jednotlivé přístupy (např. Cost Based optimalizace (CBO)), podstata optimalizátoru, přínos optimalizace.

- SQL velmi flexibilní – dvěma i více různými dotazy je možné obdržet stejná data, ovšem rychlost dotazů nemusí být stejná
- důvodem optimalizace je minimalizace nákladů na:
  - strojový čas
  - kapacitu paměti či prostoru
  - programátorskou práci
  - přenesená data
- u malých databází optimalizace nepatrná, projeví se až u objemných, nebo u často navštěvovaných webů může při špatně formulovaných dotazech vzrůst trafic v obou směrech

zpracování příkazů se skládá z následujících komponent:

- parser
- optimalizátor
- generátor řádkových zdrojů
- vlastní provádění SQL

SŘBD Oracle již sám používá optimalizačních technik pro vyhodnocení jakéhokoli dotazu. Těmito technikami jsou:

- Rule based optimaliaztion (RBO)
- Cost based optimalization (CBO) – od verze Oracle 9i je preferovaný

Jak zjistit způsob provedení příkazu? Abychom mohli zjistit, jak ve skutečnosti daná optimalizace vyhodnocení dotazu funguje, potřebujeme vytvořit tabulku **PLAN_TABLE** (podle skriptu *utlxplan.sql),* kam optimalizátor ukládá své vítězné plány právě vyhodnoceného dotazu. Pro vysvětlení (tj. zjištění optimálního plánu) vyhodnocení dotazu použijeme příkaz **EXPLAIN_PLAN.**

```
explain plan for select p.NAZEV,m.ZKRATKA, ...
```

Vykonáním tohoto příkazu se vítězný plán uloží do tabulky PLAN_TABLE v podobě několika záznamů. Informace vyčteme s použitím dotazu:

```
select plan_table_output from table(dbms_xplan.display());
```

CBO/RBO

Oracle doporučuje používat pouze CBO, který je stále vylepšován a RBO je implementován hlavně kvůli zpětné kompatibilitě.

RBO (Rule Based Optimization)

- Starší přístup, dnes často deprecated (Oracle),
- Odvozuje plán ze syntaxe příkazu a existence indexů
  řídí se předem sestavenou sadou pravidel, která nezohledňují např:
  - velikost tabulky -- ***Malá tabulka*** (obsahuje 5 řádků a vejde se do jednoho datového bloku), je rychlejší jí přečíst celou než hledat podle indexu (1 IO operace vs. čtení bloku indexů a pak dat)
  - Možnost špatného výběru použitého indexu - Pokud existuje více neunikátních indexů na jedné tabulce, nemusí optimalizátor vybrat ten nejlepší. Použití určitého indexu je možné optimalizátoru znemožnit použitím výrazu v dotazu.

  Ceny přístupu k podmnožině řádek v tabulce v klesajícím pořadí:

- *Plný přístup (Full scan)* – prochází se celá tabulka – všechny záznamy, u každé řádky se ověří podmínka. Vhodné, pokud procento vyhovujících řádek bude velké.
- *Index-Range-Scan* – vyhledání intervalu v indexu. Ověření ostatních podmínek v odkazovaných řádcích
- *Unique-Index-Scan* – vyhledání jediné možné vyhovující řádky podle unikátního indexu
- *ROWID-Scan* – vyhledání řádky na základě známé hodnoty jejího fyzického identifikátoru v DB.

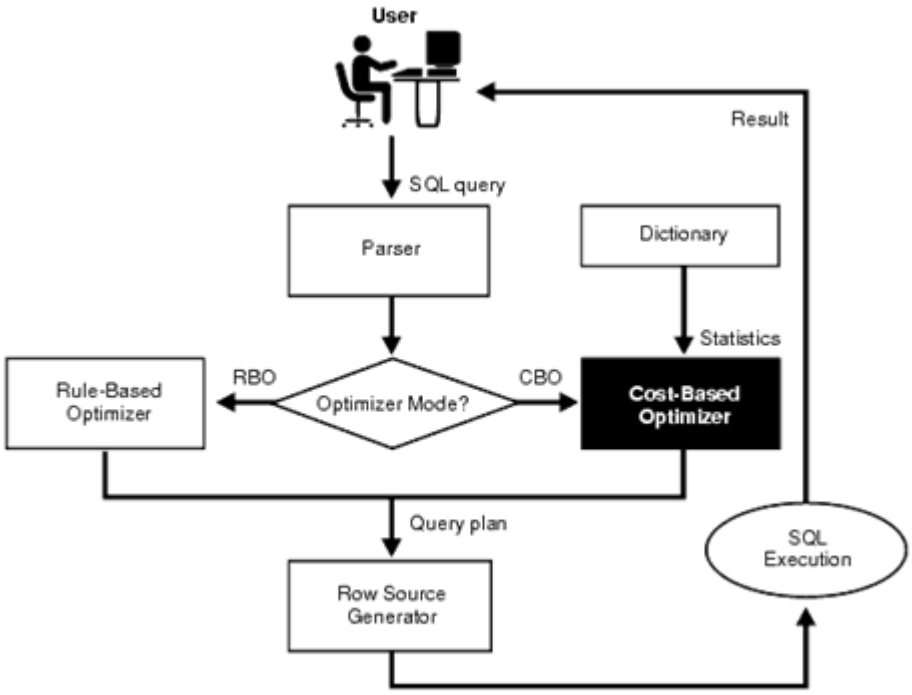
- Z důvodu kompatibility je možné jej však aktivovat odpovídajícím hintem RULE.

CBO (Cost Based Optimization)

- Hledá plán s nejmenšími náklady pomocí statistik. Optimalizace je založena na vyhodnocení kvantitativního využívání zdrojů v průběhu zpracování dotazu (CPU, paměť, I/O,…). Využívá **statistiky** o tabulkách a datech, které jsou uloženy v Data Dictionary:
- Počet různých hodnot ve sloupci, histogramy rozložení hodnot ve sloupci, počet řádek v tabulce, průměrná délka jedné řádky.
- Některé statistiky jsou přístupné i pro uživatele db pomocí pohledů, či tabulek
- Aktualizují se výpočtem nebo odhadem
- **Typy statistik**:
  - **Údaje o tabulkách** – počet řádků, bloků, délka záznamu
  - **Údaje o sloupcích** – počet unikátních hodnot a NULL, histogram
  - **Údaje o indexech** – počet listových bloků, clustering
- Dokáže rozlišit plány i pro různé typy konstant v dotazu.
- Použití CBO je doporučeno firmou Oracle. Vybírá se plán s nejnižší váženou cenou.

Podstata optimalizátoru a přínos

Podstata: stejná data lze z databáze získat různými dotazy (SELECT * vs. SELECT col1, col2…), výsledek bude stejný, ovšem zpracování se může lišit potřebným časem a systémovými nároky.



Výstupem optimalizátoru je plán vykonávání (execution plan), který určije:

- Přístupové cestu k jednotlivým tabulkám používaných dotazem a pořadí jejich spojování (JOIN order)

Hints

Hint = podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu. Hinty se aplikují na blok dotazu, ve kterém se vyskytují.

V CBO lze využít pro optimalizaci i nápovědu – Hints. Prostřednictvím ní mohu optimalizátoru vnutit některou operaci, protože si myslím, že její užití přispěje k lepší optimalizaci. Tato nápověda se zapisuje jako komentář specifického tvaru.

```
explain plan for select /*+ INDEX (predmety predmety_index1) */
p.NAZEV,…
```

Indexy

- Tvorba indexů není v SQL-92 standardizována
- Jednotlivé databázové systémy řeší tvorbu indexů svými prostředky, které jsou navzájem více či méně podobné
- Může se lišit syntaxe, podpora různých typů indexů, jejich použití/nepoužití pro daný dotaz

- **B-tree indexy**
  - Obvykle redundantní B+ stromy
    - Hodnoty v listech

    - Listy oboustranně linkované pro snadný sekvenční průchod
    - Vhodné pro sloupce s vysokou selektivitou (počtem různých hodnot ve sloupci)
    - Vícesloupcové (složené) indexy mohou zvýšit selektivitu
    - Nad jednou tabulkou v jednom dotazu nelze obvykle kombinovat více B-tree indexů. Dotaz se vyhodnocuje s použitím jednoho z indexů a ostatní podmínky se dopočítávají.
- **Bitmapové indexy**
    - Pro každou hodnotu sloupce/výrazu vytvořen binární řetězec obsahující 1 právě pro řádky s danou hodnotou
        - Vhodné pro sloupce s nízkou selektivitou
        - Lze kombinovat více bitmapových indexů nad jednou tabulkou pro zvýšení selektivity
        - Kombinací více bitmap se zvyšuje selektivita indexu
- Indexy nepomohou
    - Pokud je procento vyhovujících záznamů velké (zvýšená režie s přístupem k řádkům v nesekvenčním pořadí daném indexem
    - Při dotazech na hodnotu null
        - V indexech se běžně neukládá
- Indexy pomohou
    - V dotazech na rovnost sloupce s konstantou
    - V dotazech na to, zda je hodnota v intervalu
- Indexy jsou automaticky vytvářeny
    - Pro primární klíče
    - Pro sloupce s UNIQUE (kandidátní klíče)
- **Vždy vytvářet indexy pro cizí klíče!!!**
    - Zrychlení odezvy při manipulaci s nadřízenou tabulkou
    - Průchod přes index najde efektivně všechny existující závislé řádky bez nutnosti čtení celé tabulky

Výběr typu optimalizace

Je dalším krokem, kterým můžeme ovlivnit optimalizaci. Jedná se o změnu optimalizačního typu optimalizátoru SŘBD Oracle. Typy optimalizace jsou:

- **CHOOSE** – výběr podle (ne)přítomnosti statistik nejsou-li k dispozici, potom RBO, jinak CBO.
- **ALL_ROWS** – vždy CBO, minimalizuje se cena za získání všech řádek odpovědi. Vhodné pro dávkové zpracování.
- **FIRST_ROWS** – vždy CBO, minimalizuje se cena za získání prvních řádek odpovědi. Vhodné pro interaktivní zpracování.
- **RULE** – vždy RBO.

Typ optimalizace se vybírá příkazem:

```
ALTER SESSION SET OPTIMIZER_GOAL = ALL_ROWS;
```

Další možnost ladění je změna módu optimalizátoru. Dotazy mohou být optimalizovány na:

- Nejlepší průchodnost – ALL_ROWS
- Nejrychlejší odezvu – FIRST_ROWS_1 – zkrátí čas vyhodnocení dotazu.

Př. nastavení módu:

```
ALTER SESSION SET optimizer_mode = all_rows;
```

Obecná pravidla pro psaní SQL dotazů

Vyplývají z technik optimalizace:

- V selectu nepoužívat v seznamu sloupců *, protože ve většině případů nepracujeme se všemi.
- Používat co nejméně klauzuli LIKE, IN, NOT IN (vhodnější je WHERE a WHERE NOT EXISTS)
- Používat klauzule typu LIMIT
- Používat hinty (podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu)
- Na začátek dávat obecnější podmínky (takové, po kterých vypadne co nejvíc záznamů)
- Výběr vhodného pořadí spojení
- Nastavit indexy

Přínos:

- zdrojový čas
- kapacitu paměti či prostoru
- programátorskou práci
- přenesená data

2.1.8. Postrelační databáze – výhody a nevýhody, mapování, RDB, ORDB, OODB.

Postrelační databázový systém je relační databázový systém rozšířený o nějakou specializaci na databázové úrovni, jelikož aplikační řešení by bylo nedostačující

Příklady postrelačních DB systémů

- *Prostorové databáze* — rozšířeny o práci s prostorovými objekty a vztahy mezi nimi
- *Objektově orientované databáze* — rozšířeny o objektový model dat a vazby
- *Deduktivní databáze* — rozšířeny o funkce pro analýzu dat
- *Temporální databáze* — rozšířeny o temporální logiku
- *Multimediální databáze* — rozšířeny o funkce pro práci s multimediálním obsahem
- *Aktivní databáze* — rozšířeny o aktivní pravidla

V současnosti všechny používané databázové systémy jsou postrelační, jelikož obsahují nějaká rozšíření oproti původnímu relačnímu schématu (např. triggery).

- **RDB** (RSŘBD) - relational database
- **ORDB** (ORSŘBD) - object-relational database
- **OODB** (OOSŘBD) - object oriented database

Jedná se o všechny současné databáze - jde o relační databáze doplněné o nějakou "funkci" navíc, např. aktivní databáze (triggery) už jsou postrelační databáze.

Relační databáze

Technologie relačních databází byla původně navržena E.F.Coddem a později ji implementovala IBM a jiní. Standard je popsán ANSI a ISO normou, časteji se na ni ovšem odvoláváme jako na SQL + číslo verze. Poslední je tedy SQL2. Novější verze SQL3 obsahuje navíc některá objektová rozšíření.

- Datový model

  RDB uchovává data v databázi skládající se z řádků a sloupců. Řádek odpovídá záznamu (record, tuple); sloupce odpovídají atributům (polím v záznamu). Každý sloupec má určen datový typ. Datových typů je omezené množství, typicky 6 nebo víc (např. znak, řetězec, datum, číslo...). Každý atribut (pole) záznamu může uchovávat jedinou hodnotu. Vztahy nejsou explicitní, ale spíše plynou z hodnot ve speciálních polích, tzv. cizí klíče (foreign keys) v jedné tabulce, který se rovná hodnotám v jiné tabulce.

- Dotazovací jazyk

  Pohled (view) je podmnožina databáze, která je výsledkem vyhodnocení dotazu. V RDB je pohled tabulka. RDB využívá SQL pro definici dat, řízení dat a přístupu a získávání dat. Data jsou získávána na základě hodnoty v určitém poli záznamu.

- Výpočetní model

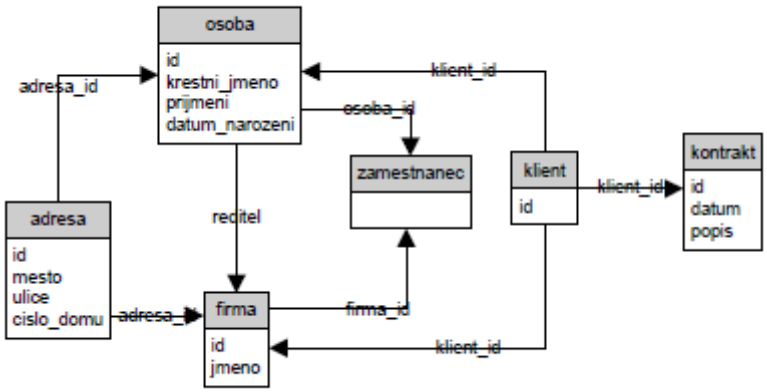
  Veškeré zpracováváni je založeno na hodnotách polí záznamů. Záznamy nemají jednotné identifikátory, které jsou neměnné po dobu existence záznamu. Neexistují žádné odkazy z jednoho záznamu na jiný. Vytvoření výsledku je prováděno pod kontrolou kurzoru, který

umožňuje uživateli sekvenčně procházet výsledek po jednotlivých záznamech. Totéž platí pro update.

Výhody:

- Výkonné OLTP
- Dostupnost dat
- Utajení
- Prostředky pro správu dat
- Standardní jazykové rozhraní
- Řízení paměti
- Souběžné zpracování dat
- Integrita

Příklad:

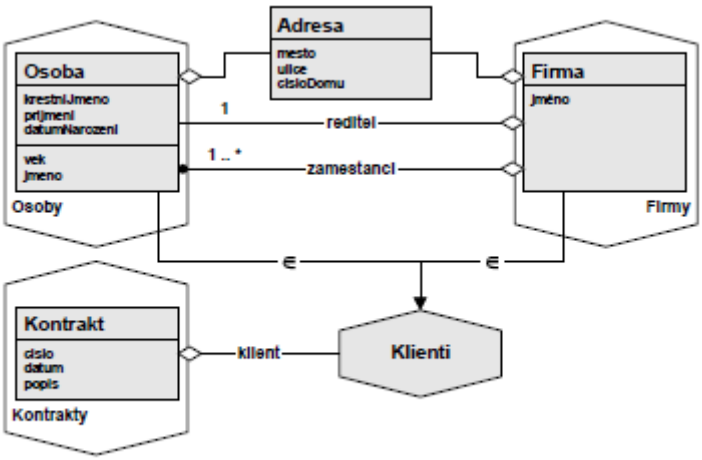


obr. č. 2. - relační implementace databáze

Objektová databáze

Vedle relačních databází se začal vyvíjet nový typ databázových systémů, založených na principech objektového programování. Co nového objektové databáze přináší? Tak jako jsme mohli vnímat přechod od strukturálního programování k objektovému programování (např. klasický Turbo Pascal a Delphi), tak můžeme vnímat i přechod z relačních databází na objektové databáze. Základem OO databáze není tentokrát tabulka, ale objekt. Každý objekt má atributy/vlastnosti (zde je vidět analogie se sloupci v tabulce) a metody, které nějakým způsobem manipulují s hodnotami vlastností. Jednotlivé "záznamy" jsou instance objektu s konkrétními hodnotami (v relačních databázích - 1 řádek). Lze zde využít všech výhod dědičnosti (a to i mnohonásobné), zapouzdřenosti a polymorfismu. Díky tomu OO databáze výrazně rozšiřují možnosti tvorby databázových aplikací.

Příklad oproti relační:



obr. č. 3. - objektová implementace databáze

Pro objektové databáze neexistuje žádný oficiální standard. Standardem je de facto kniha Morgana Kaufmana The Object Database Standard: ODMG-V2.0. Důraz ODB je na přímou korespondenci mezi následujícími:

- Objekty a objektové vztahy v aplikaci napsané v OO jazycích
- jejich uchovávání v databázi.

- Objektově orientované databáze (OODB) využívají objektových principů jako jsou abstraktní datové typy, zapouzdření, inheritance, polymorphismus apod. Struktura objektu je (i, c, v) = (unique id, constructor, stav objektu). Unique Object identifiers, …

  OQL = Object Query Language = deklarativní jazyk, přidaná flexibilita

- Datový model

  Objektové databáze využívají datového modelu, který má objektově orientované aspekty jako třídy s atributy a metodami a integritními omezeními; poskytují objektové identifikátory (OID) pro každou trvalou instanci třídy; podporují zapouzdření (encapsulation); násobnou dědičnost (multiple inheritance) a podporují abstraktní datové typy.

  Objektové databáze kombinují prvky objektově orientovaného programování s databázovými schopnostmi. Rozšiřují funkčnost objektových programovacích jazyků (C++, Smalltalk, Java) a poskytují plnou schopnost programování databáze. Datový model aplikace a datový model databáze se ve výsledku hodně shodují a výsledný kód se dá mnohem efektivněji udržovat.

- Dotazovací jazyk

  Objektově orientovaný jazyk (C++, Java, Smalltalk) je jazykem jak pro aplikaci, tak i pro databázi. Poskytuje těsný vztah mezi objektem aplikace a uloženým objektem. Názorně je to vidět v definici a manipulaci s daty a v dotazech.

- Výpočetní model

  V RDB rozumíme dotazovacím jazykem vytváření, přístup a aktualizaci objektů, ale v ODB, ačkoliv je to stále možné, je toto prováděno přímo pomocí objektového jazyka (C++, Java, Smalltalk) využitím jeho vlastní syntaxe. Navíc každý objekt v systému automaticky obdrží identifikátor (OID), který je jednoznačný a neměnný během existence objektu. Objekt může mít buď vlastni OID, nebo může ukazovat na jiný objekt.

Výhody:

- Operace na složitých objektech
- Rekurzivní struktury
- Abstraktní datové typy
- Rozhraní k OO jazyku
- Složité transakce

Objektově-relační databáze

"Rozšířená relační" a "objektově-relační" jsou synonyma pro databázové systémy, které se snaží sjednotit rysy jak relačních, tak objektových databází. ORDB je specifikována v rozšíření SQL standardu — SQL3. Do této kategorie patří např. Informix, IBM, Oracle a Unisys.

- Datový model

  ORDB využívají datový model tak, že "přidávají objektovost do tabulek". Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu, nazývanou abstraktní datové typy (ADT). ADT je datový typ, který vznikne zkombinováním základních datových typů. Podpora ADT je atraktivní, protože operace a funkce asociované s novými datovými typy mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu. ORDB jsou nadmnožinou RDB a pokud nevyužijeme žádné objektové rozšíření jsou ekvivalentní SQL2. Proto má omezenou podporu dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem.

- Dotazovací jazyk

  ORDB podporuje rozšířenou verzi SQL — SQL3. Důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod. ORDB je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření, pracuje právě s nimi.

- Výpočetní model

  Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s databází. Přímá podpora objektových jazyků stále chybí, což nutí programátory k překladu mezi objekty a tabulkami.

  Dva přístupy:

- univerzální paměť, kdy všechny druhy dat jsou řízeny SŘBD), jde o integraci (různými způsoby!) ⇒ univerzální servery
- univerzální přístup, kdy všechna data jsou ve svých původních (autonomních) zdrojích

Technika: middleware

- brány (min. dva nezávislé servery)
- zobrazení schémat, transformace dotazů
- objektové obálky: Persistence Software, Ontologic, HP,
- Next, ... (problémy: výkon)
- DB založené na Web

Shrnutí

Relační model je jednoduchý a elegantní, ale je naprosto rozdílný od objektového modelu. Relační databáze nejsou navrhovány pro ukládání objektů a naprogramování rozhraní pro ukládání objektů v databázi je velmi složité. Relační databázové systémy jsou dobré pro řízení velkého množství dat, vyhledávání dat, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. Jazyky jako SQL umožňují tabulky propojit za běhu, aby vyjádřily vztah mezi daty.

Naproti tomu **objektově orientovaný model** je založen na objektech, což jsou struktury, které kombinují daný kód a data. Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty "v databázi"; tj. místo věčného přeskakování mezi jazykem aplikace (např. C) a dotazovacím jazykem (např. SQL) může programátor jednoduše používat objektový jazyk k vytváření a přístupu k metodám. Krátce řečeno, ODB jsou výborné pro manipulaci s daty.

Hlavní rozdíl je v přístupu ke vztahům

- v OO databázích jsou vztahy reprezentovány pomocí OIDs, což zlepšuje přístup k datům
- v relačních databázích jsou vztahy mezi n-ticemi specifikovány atributy se stejnou doménou.

Nevýhody OO

- Chabý výkon (ORM 15-20% slabší než samotný JDBC driver). Ve srovnání s relačními jsou optimalizátory pro OO DB velmi složité.
- Problémy se škálovatelností, neschopnost podporovat rozsáhlé systémy.

2.1.9. **ANSI/ISO normy SQL – objektové vlastnosti jazyka SQL99.**

ANSI/ISO normy SQL

Vývoj standardů SQL:
SQL86
SQL89
SQL92
SQL/Call Level Interface 95
SQL/Persistent Stored Module Language Interface 96
SQL/Java
SQL99
SQL/Object Language Bindings 2000
SQL/Management External Data 2000
SQL/OLAP
SQL/temporal
SQL/Schemata
SQL/XML
SQL/MM

Pracovní názvy:

- SQL1 (>SQL86)
- SQL2 (>SQL92),
- SQL3 (>SQL99),
- SQL4

SQL-86 (SQL 87)
První standard formalizovaný ANSI

SQL-92 (SQL2)

Standard je rozdělen na tři úrovně: *entry*, *intermediate* a *full*. Někdy je také uváděn mezistupeň mezi entry a intermediate jako *transitional*. Úrovně slouží k tomu, aby mohlo být u implementací standardu (jednotlivých databází) uvedeno do jaké míry splňují daný standard.

Změny možno klasifikovat jako:

- Entry
  Jen formální změny oproti SQL-86
- Transitional
  - Podpora různých druhů spojení jako NATURAL JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN
  - Podpora nových datových typů DATE, TIME, TIMESTAMP and INTERVAL, including various datetime and interval features (excluding time zones)
- Intermediate
  - Podpora dlouhých identifikátorů (do 128 znaků)
  - Podpora kurzorů a směru získávání dat příkazem FETCH
  - Podpora definice a používání znakových sad
- Full
  - Podpora dočasných tabulek
  - Možnost výběru přesnosti u datových typů TIME a TIMESTAMP
  - Možnost testování pravdivostních hodnot pomocí TRUE, FALSE or UNKNOWN
  - Vnořené tabulky ve FROM
  - Podpora UNION JOIN a CROSS JOIN
  - Podpora pro collations znakových sad
  - Vylepšené udělování práv

SQL:1999 (SQL3)

Jedná se o standard pro relačně-objektový dotazovací jazyk (na rozdíl od předchozích verzí, které byly pouze relační)

Jedná se o standard pro relačně-objektový dotazovací jazyk (na rozdíl od předchozích verzí, které byly pouze relační)

- Regulární výrazy
- Rekurzivní dotazy
- Triggery
- Procedurální rozšíření (Příkazy řízení běhu - LOOP, IF...)
- Objektové rozšíření
- nové typy STRING, BOOLEAN, REF, ARRAY, typy pro full-text, obrázky, prostorová data

Existují i novější standardy

- **SQL:2003** (Představeny XML-vázané funkce, window funkce, standardizované sekvence a sloupce s automaticky generovanými hodnotami)

- **SQL:2006** (SQL může být použito ve spojení s XML - možnost importu a skladování XML dat v SQL databázi, manipulaci s nimi a publikace dat v XML formě. Možnost využití XQuery)
- **SQL:2008** (ORDER BY mimo definici kurzoru, INSTEAD OF triggery, přidán TRUNCATE příkaz)
- Jednotlivé databázové servery ne vždy dodržují ANSI normu – obvykle pouze SQL-92 Entry
- Čím vice se při vývoji aplikace využijí rysy vyšší než SQL-92 Entry, tím je menší šance, že aplikace bude provozuschopná i na jiné databázi

většina z těchto rozšíření lze najít v jiných otázkách, proto zde nebudou více rozepsána.

2.1.10. Objektově relační databáze.

Za posledních 25 let je mohutný trend přechodu od strukturovaného k objektově orientovanému programování, a to i v oblasti zpracování dat a databází. Objektově orientované DB se objevili v 90. Letech minulého století a kladou si za cíl urychlit a ulehčit práci s daty. Situace ovšem není jednoduchá, protože relační a objektový přístup je od základu rozdílný. Existuje tak mnoho výhod i mnoho nevýhod pro relační i objektové DB. Na současném trhu existují 3 základní typy:

- **Relační DB** (Relational Database Management System, RDBMS) – výkonné na tradičních datech. Př. Oracle 7.x, DB2.
- **Objektově-relační** (Object Relational Database Management t System, ORDBMS) – Poskytuje programátorské pohodlí, rychlý a udržitelný vývoj aplikací. Př. Oracle 8.x, 9.x, 10.x.
- **Objektové DB** (Object Database Management system, ODBMS) – Výkonné na netradičních datech, slabší v databázových rysech. Př. Jasmine, Gemstone, O2.

Objektově relační databáze

Objektově relační DB se snaží přinést objektové rysy do relačních databází. ORDBMS je specifikována v rozšíření SQL standardu – SQL3. Do této kategorie patří např. Informix, IBM, Oracle a Unisys.

Objektově relační mapování přináší vrstvy mezi OO aplikací a SQL databází. Charakteristiky jsou:

Myšlení v objektech, ev. Cache objektů, zodpovědné za persistenci.

Datový model ORDBMS

Rozšiřují datový model tak, že přidávají objektovost do tabulek. Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu (porušení 1. NF), nazývanou abstraktní datové typy, tzv. ADT:

- ADT je datový typ, který vznikne kombinací základních datových typů
- Podpora dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem je omezená
- Funkce a operace jsou asociované s novými datovými typy, mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu.

Dotazovací jazyk

ORDBMS podporují rozšířenou verzi SQL – SQL3 (SQL 99), důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). ORDBMS je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření. Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod.

Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s DB.

Objektové vlastnosti SQL99 (SQL3)

Objektové rozšíření standardizované v SQL99 zahrnuje:

- **Strukturované uživatelské typy** – Oracle od verze 9.i včetně jednoduché dědičnosti. Mohou být organizovány do hierarchie s děděním. Chování uživatelem definovaných typů je

realizováno pomocí procedur a funkcí a (metod u ADT). Jedná se o řádkové typy, ADT, odlišující typy.

- **Pole s proměnnou délkou (VARRAY)** – CREATE TYPE typ AS VARRAY(5) OF VARCHAR(15)
- **Hnízděné tabulky** – typ TABLE
- **Typ REF** – odkaz ukazatel – jeho obsahem je OID nějakého záznamu, nelze s ním manipulovat jako s hodnotou, ale jako s odkazem. Zajišťuje objektovou identitu.
    - Výhoda - pro sdílení objektů (nejsou zbytečně kopírována data a změna se provádí na jednom místě)

- CREATE TYPE TypHerec AS (jmeno CHAR(30), nejlepsiFilm REF (FilmTyp))
- Dereference př. SELECT Film->titul FROM hrajev WHERE herec->jmeno='Chaplin';

SQL99 je kompatibilní s existujícími jazyky a další vlastnosti jsou:

- **Řádkové typy** (jsou typem relace)
    - Vytvoření řádkových typů:

```
CREATE ROW TYPE typadresa (ulice CHAR VARYING(50), mesto CHAR
VARYING(20));
```

    - Příklad tvorby tabulky s řádkovým typem:

```
CREATE TABLE FilmovyHerec OF typherec;
CREATE TABLE FilmovyHerec (
     jmeno CHAR VARYING(30),
     adresa ROW (
          ulice CHAR VARYING (50),
          mesto CHAR VARYING(20)
     )
);
```

- Tvorba dotazů:

```
SELECT FilmovyHerec.jmeno, FilmovyHerec.adresa.ulice
FROM FilmovyHerec WHERE FilmovyHerec.adresa.mesto = 'Plzeň';
```

- **Lze definovat i podtypy** a **podtabulky** CREATE TYPE typSekretarka UNDER typZamestnanec AS(...

- Abstraktní datové typy (jsou typem atributu relace) – umožňují zapouzdření atributů a operací (na rozdíl od řádkových typů). Hodnoty jejich typů mohou být umístěny do sloupců tabulek.

    - Př. (Oracle)

      ```
      CREATE TYPE typZamestnanec AS (
      c_zam INTEGER,
      METHOD mzda() RETURNS DECIMAL);
      CREATE METHOD mzda ... FOR typZamestnanec
      BEGIN ... END
      ```

    - Instance vznikají konstruktorem jmenoTypu(), operátorem NEW jmeno hodnota, příkazem INSERT INTO osoby VALUES(...)

    - Funkce a procedury vyjádřeny v SQL/PSM (Persistent stored module), nebo C/C++, Java, ADA.... Jsou svázány s ADT. Metody jsou uložené ve schématu typu definovaného uživatelem. Metody se dědí. Metody i funkce mohou být polymorfní (liší se způsobem výběru). CREATE PROCEDURE zjisti_cenu ...; CALL zjisti_cenu(...);

- Odlišující typy (musí být FINAL) – emulace domén – strong typing
- OID – záznamy mají/mohou mít OID (v relačních DB mohou být použity jako primární klíče), které zajišťují objektovou identitu. V jiných záznamech atribut typu REF – odkaz ukazatel. Zpřístupnění klauzulemi REF IS SYSTEM GENERATED a REF IS USER GENERATED.

2.1.11. Vlastnosti objektově orientovaného datového modelu.

HDM, SDM a RDM = záznamově orientované modely. V 90. letech se začaly objevovat první objektově orientované SŘBD (OOSŘBD), které umožňují pracovat s datovou abstrakcí na úrovni objektů.

- Výhody
  - snadnější aktualizace dat
  - přímé vyjádření složitých objektů modelované reality v databázi (odpadají mezikroky převodu objektů do normalizovaných tabulek relační databáze. Stejně tak je zjednodušen i opačný krok načítání objektů z databáze do aplikace)
  - součástí uložených objektů je také jejich chování (metody atd.)
- Nevýhoda
  - Vývoj a návrh objektově orientovaných modelů v jejich univerzálnosti a komplexnosti je velmi složitý proces ⇒ menší uplatnění tohoto typu modelu v reálných aplikacích.

Objektově orientované programování (OOP)

Koncepce objektově orientovaných databází vychází z principů používaných v OOP - základem je objekt (prvek typu třída). Data se nazývají atributy, funkce metody (služby). Metoda je aktivována příchodem zprávy do jiného objektu.

Hlavní vlastnosti OOP: zapouzdření dat, dědičnost, polymorfismus

Objektově orientované modelování dat

Postup objektově orientovaného modelování dat lze stručně shrnout do následujících bodů:

- Vyhledají se objekty jako nositelé aktivit (např. metodou gramatické inspekce: podstatná jména představují objekty nebo třídy, přídavná jména představují hodnoty atributů a slovesa představují většinou aktivity).
- Identifikují se třídy zobecňování objektů se stejnými atributy (+ zkoumá se možnost uspořádat třídy hierarchicky podle dědičnosti) a vztahy.
- Stanoví se integritní omezení na hodnoty jednotlivých atributů a případně na typy atributů.
- Vyhotoví se seznam nabízených a požadovaných služeb pro všechny metody (přehled toku zpráv)
- Implementují se metody (teprve po jednoznačném vymezení všech funkcí systému)

Hlavní rozdíly mezi RDM a ODM:

	RDM	ODM
1	• relační tabulka • jeden záznam • manipulace s atributy záznamu	• množina objektů • jeden objekt • přenos a zpracování zpráv
2	normalizace relací (dekompozice) vede k rozptýlení popisu vlastností složitého objektu do mnoha tabulek	spojuje jednotlivé složky pomocí odkazů
3	záznamy relací jsou omezeny na jednoduché datové typy	složité strukturované datové entity - objekty, které lépe vystihují prvky reálného světa
4	manipulace s hodnotami atributů záznamů	operace posílání zpráv poskytuje větší možnosti
5	každá tabulka musí mít identifikační klíč (ten nemusí odrážet požadavky zadání)	zabezpečuje identifikaci objektů vlastními systémovými prostředky (OID)
6	při zpracování dotazů dochází často k získávání údajů z několika tabulek ⇒ narůstá čas potřebný k vyhodnocení dotazu	ke spojování množin dochází v daleko menší míře; dotazovací konstrukce lze díky polymorfismu aplikovat i na množiny obsahující různé typy objektů

Pozn.: RDM za určitých podmínek představuje zvláštní případ ODM.

Produkty: Objectivity, Versant, POET, CACHÉ, db4o, Ozone, GOODS, XL2, ZODB, Prevayler

Bariéry rozšíření objektových databázových systémů:

- neochota vývojářů a jejich klientů k přechodu od tradičního relačního přístupu k objektovému
- nedostatek kvalifikovaných vývojářů
- nízká podpora standardů
- nízká podpora dotazovacích jazyků
- neexistence mechanismu pro řízení přístupu k datům

Objektový (konceptuální model)

- popisuje z čeho je realita složena a jaké jsou základní (podstatné/statické) složky (objekty) a vazby mezi nimi.
- Akce (metody) a algoritmy, vázané k objektům v jejich životních cyklech, zde mají význam též statický (jsou podřízeny statickému - pohledu).
- K popisu lze využít specifický diagram – Diagram tříd (Class Diagram - základní diagram jazyka UML).
- Model (entitních, business, analytických) objektů (podstata struktury reality) sleduje základní stavební kameny, z nichž se realita (problémová doména) skládá.

Třída, Objekt

- **Třída** - popis množiny objektů sdílejících stejné vlastnosti (atributy), chování (operace/metody) a vztahy.
- **Objekt** - instance třídy (chybně se pojem třída a objekt volně zaměňují).

Definice – J. Rumbauh: objekt je diskrétní entita s jasně definovaným rozhraním, které zapouzdřuje stav a chování.

- Třídu si můžeme představit jako razítko, objekty jsou pak otisky tohoto razítka, které vidíme na papíře.
- Při návrhu třídy neuvažujeme o konkrétním naplnění atributů, pouze určíme jejich název a typ. Teprve při vzniku instance objektu se atributům přiřadí skutečné hodnoty.
- Třída je jednoznačně určena svým názvem (v příslušném názvovém prostoru – balíčku). Pro třídu je možno definovat vlastnosti - atributy (Attribute) a chování - operace (Operation).
- Hledání tříd, jejich atributů a kompetencí - vyberme z reality objekty, kandidáty pro zobecnění na třídy a prověřme jejich vhodnosti:
    - Potenciální třída je smysluplná, pokud je nezbytná pro funkci systému.
    - Potenciální třída je dostatečně stabilní a invariantní vůči vnějším změnám např. technologie, legislativy apod.

Atributy

Definice atributů

Atribut určuje vlastnosti objektu, je nositelem informace o objektu.

Atributy popisují hodnoty (stavy) udržované v jednotlivých objektech.

Objekty jsou vymezeny (popsány) množinou atributů.

Atributy popisují vlastnosti objektů, které potřebujeme k dosažení daného cíle. Reálný objekt ve své nekonečné složitosti nelze vymezit omezenou množinou atributů. Základní problém analýzy IS - výběr rozumného množství relevantních atributů.

Vazby – relace mezi třídami

Vazba asociace (Association)

Vazba asociace mezi třídami je vyjádřením abstraktního vztahu mezi objekty (instancemi tříd). Asociace říká, že objekty mají mezi sebou přímý vztah, že o sobě ví.

Zaměstnanec pracuje v daném oddělení, mohu se ptát: V jakém oddělení pracuje zaměstnanec, mohu získat seznam všech zaměstnanců v oddělení. Vazba je nositelem významu – sémantiky, odpovídá – mapuje požadavky kladené na systém. Je trvalejšího charakteru.

Asociace je společný typ vazby pro:

- agregaci, vyjadřující vztah mezi celkem a částí
- prostou asociaci, vyjadřující prostou objektovou referenci.

Vazba asociace je specifikována řadou vlastností, z nichž některé jsou vázány přímo k vazbě asociace (například název), ostatní k zakončením vazby (například role). Podrobné určení

vlastností až v okamžiku návrhu – specifikace návrhových tříd a jejich vazeb má „implementační“ důsledky.

Vazbu asociace lze zavézt jako orientovanou (Navigability), přičemž neorientovaná vazba je považována za obousměrnou (dva jednosměrné vztahy).

Třídy v asociaci mohou vůči sobě vystupovat v rolích (Role) (Objednávka – Zaměstnanec, Zaměstnanec vystupuje ve vztahu k objednávce v roli Prodejce). Každá strana asociace má své jméno – roli. Role popisuje vlastnost, funkci třídy „viděné“ z druhé strany.

V asociaci lze určit násobnost vazby, (kardinalitu) multiplicitu, která vyjadřuje počet možných vazeb objektů tříd v asociaci (0, 0..1, 0..*, 1, 1..*, *, M..N, …).

Násobnost vazby definuje, kolik může k jednomu objektu, tj. k jedné instanci třídy A na jedné straně vztahu existovat minimálně (parcialita) a maximálně (kardinalita) objektů ze třídy B na druhé straně vztahu a obráceně.

- Standardně je vazba asociace implementována zavedením atributu třídy - role pro zachycení objektové reference (množiny referencí pro parcialitu 0..*) na objekty druhé třídy. Implementace vazby – objekt si sebou nese reference na asociované objekty.
- S vazbami je třeba šetřit.
- Na rozdíl implementace v relačním datovém modelu se jedná o explicitní vyjádření vazby. V relačním modelu se pro vyjádření vazby používají tzv. cizí klíče – implicitní implementace vazby.

Další vlastnosti vazeb související s implementací vazby (mimo UML, CASE):

Každý konec asociace, vazby se nazývá role. Pro roli můžeme definovat řadu vlastností.

- Asociativní třída (Association Class) je vazba asociace, která je rozšířena přiřazením třídy pro zachycení informací nutných pro úplnou specifikaci této vazby.
- Asociativní třída se používá například v případě oboustranně násobné vazby N:M. Asociativní třída nemá vlastní identitu, identitu přejímá od „asociovaných“ tříd
- Vztah mezi třemi a více prvky popisují vícenásobné asociace (N-ary Association). Pro vyjádření vícenásobné asociace se používá element modelu asociativní třída.

Vazba agregace (Aggregation)

Agregace je vyjádřením abstrakce vztahu mezi objekty (instancemi tříd), který odpovídá vztahu celku a části .

Agregace je speciálním případem asociace. Jazyk UML rozlišuje mezi dvěma typy agregací:

- prostou agregací (Simple Aggregation)
- kompozicí (Composition).

Vazba kompozice je silnější než vazba prosté agregace, jedná o vlastnictví částí celkem. Pokud je celek existenčně (tj. svou logikou) závislý na částech a nemůže bez nich fungovat, jedná se o kompozici – pokud se bez nich obejde, jedná se o agregaci.

Agregace: Profesoři - Katedra - zruším katedru, profesoři zůstanou, mohou fungovat bez katedry

Kompozice: Fakulta - katedra, zruším fakultu, katedra je bezvýznamná

Vazba generalizace (Generalization)

Vazba generalizace je vyjádřením vztahu mezi obecným elementem (Parent) a specifickým elementem (Child), který je konzistentní s obecným elementem a přidává k jeho definici další informace, je tedy bližší specifikací (specializací) obecného elementu.

Vazba generalizace mezi třídami je vyjádřením vlastnosti dědičnosti, jedné ze základních vlastností objektově orientovaného přístupu.

Jazyk UML povoluje vyjádřit vícenásobnou dědičnost zavedením více vazeb generalizace.

Vazba závislosti (Dependency)

- umožňuje znázornit jistou závislost mezi elementy modelu.
- je určena svým názvem a obvykle se používá s určitým stereotypem, který blíže specifikuje formu závislosti, zavádí její typ.
- je znázorněna orientovanou přerušovanou čárou, kde orientace je vyjádřena šipkou ve směru závislosti.

Závislost obvykle vzniká pouze dočasně pro potřeby poskytnutí služby klientskému objektu a poté tato vazba zaniká (implementační rozdíl od asociace).

Změna jednoho (nezávislého) elementu ovlivní druhý (závislý) element.

Objektové databáze

Stejně tak jako lidé postoupili od strukturálního programování k objektovému, tak si řekli, že by nebylo od věci neukládat data do tabulek a relací, ale do objektů tak, jak s nimi pracujeme přímo v programu. Bylo by přeci velice pěkné, když bych mohl objekt tak, jak ho mám, prostě uložit do databáze a o nic víc se nemuset starat.

Nemusel bych přemýšlet nad strukturami tabulek (tak aby dodržovaly „dobré mravy“ dané normami) a tvořit ruční INSERTy a SELECTy, jen bych databázovému stroji přes nějaké API řekl, načti mi uživatele s číslem 451, a dostal bych ho se všemi atributy naplněnými.

A přesně takto objektové databáze fungují. Místo tabulek jsou zde uloženy přímo objekty, včetně svých vlastností, a místo řádků se ukládají samotné instance objektů. Každý takto vložený objekt je jednoznačně identifikován svým OID, které na logické úrovni odpovídá ukazateli do virtuální paměti počítače a stejně tak se chová (při přesunu v paměti se změní i OID). Není tedy potřeba vytvářet primární klíče na objektech ani normalizovat databázi.

Objektové databáze také nabízejí využití možností vícenásobné dědičnosti, zapouzdření a polymorfizmu. Navíc vlastnosti (datové hodnoty) objektů nemusí být jen primitivního typu, ale mohou být dále strukturované jako například objekt (pomocí reference), množina nebo seznam.

Pojem zapouzdření znamená, že každý objekt obsahuje nejen datové hodnoty (vlastnosti), ale i funkce, které definují, jak je možné s těmito vlastnostmi zacházet.

Polymorfizmus umožňuje objektům zastupovat své potomky (ve smyslu dědičnosti) při volání metod. Program nemusí znát přesný typ objektu, který volá, ale ten se zjistí až za běhu a zavolá se metoda na správné třídě.

Pro vytváření nových tříd v databázi je definován nový speciální jazyk ODL (Object Definition Language). Nicméně v dnešní době se využívá vlastností pokročilých programovacích jazyků, jako je reflexe. Například v db4o stačí zavolat metodu set na objektu databáze, předat jí jako parametr obyčejný objekt a zbytek už si zařídí knihovna sama.

```
void storePilot (string pilotName, int pilotsPoints)
{
  Pilot pilot1 = new Pilot(pilotName, pilotsPoints);
  db.Set(pilot1);
}
```

Pro načítání dat z objektové databáze existuje jazyk OQL (Object Query Language). Jak je vidět už podle názvu, jeho syntaxe je velice blízká SQL. V následujícím příkladu si povšimněte, že odpadla potřeba spojovat tabulky, protože vše je dostupné přes objektové vazby:

```
SELECT o.customer_id.get_name(), o.room_id.id
FROM orders o
WHERE o.check_day(o.room_id.id, datefrom, dateto) = 1;
```

Další velice zajímavou možností je vytvořit dotaz pomocí QBE (Query By Example). Princip tohoto přístupu spočívá v tom, že databázi předáme částečně naplněný objekt a ta nám ho dohledá ve svém

zdroji a doplní ostatní vlastnosti. Přesněji řečeno vrátí nějaký kontejner naplněný objekty, které původnímu objektu odpovídají. Uvedu zde jeden ilustrační příklad:

```
Pilot retrievePilotByName (string pilotName)
{
  Pilot proto = new Pilot("Michael Schumacher", 0);
  IObjectSet result = db.Get(proto);
  if (result.HasNext())
    return (Pilot)result.Next();
  else
    return null;
}
```

2.1.12. „Vnější“ programování (přes rozhraní/knihovny) – rozhraní ODBC, JDBC, rozhraní podporující objektově- relační mapování (Java Hibernate).

ODBC

Open DataBase Connectivity .Je standardizované softwarové API pro přístup k databázovým systémům (DBMS). Snahou ODBC je poskytovat přístup nezávislý na programovacím jazyku, operačním systému a databázovém systému. Je to čistě C-čkové API, které nemá žádný objektový základ.

Navrženo Microsoftem, proto primárně přístupné pouze přes C/C++. Založeno na specifikaci X/Open a ISO: SQL Call Level Interface (SQL/CLI)

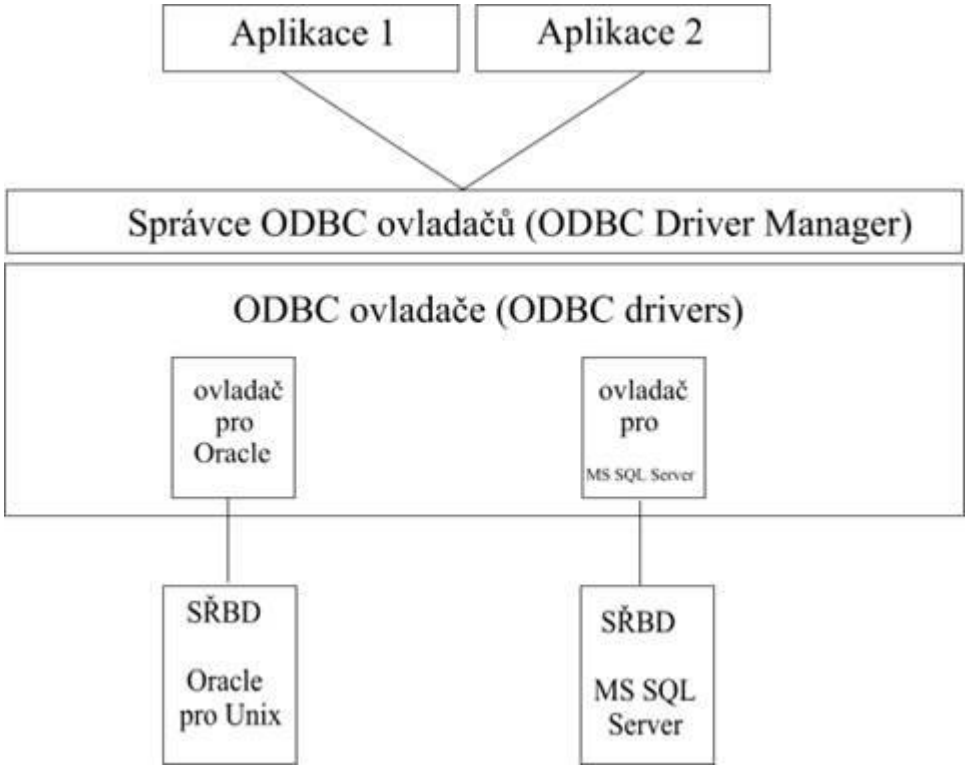
Model struktury ODBC se dá znázornit pomocí čtyř vrstev:

- V první nejvrchnější vrstvě se nachází samotná **aplikace**. Ta v případě, že potřebuje data, provede volání ODBC funkcí (ve formě SQL dotazu).

- Druhou vrstvou je tzv. **"Správce ODBC ovladačů"** (ODBC Driver Manager). Úkolem správce ovladačů je zajistit propojení mezi aplikací a příslušným ODBC ovladačem (ODBC ovladače tvoří třetí vrstvu modelu, podrobněji viz dále). Jakmile aplikace potřebuje data, správce ovladačů vyhledá a nahraje příslušný ovladač. (ve formě DLL knihovny). Správce ovladačů také zjistí, jaké konkrétní funkce jsou podporovány jednotlivými ovladači, a uschová si jejich adresy v paměti do tabulky. V případě, že aplikace volá konkrétní funkci, správce souborů zjistí, ke kterému ovladači funkce patří a zavolá ji. Tímto způsobem může být prováděn souběžný přístup k více ovladačům, což se hodí v případě programování aplikací přistupujících souběžně k několika zdrojům dat.

- Třetí vrstvou zde již zmíněnou vrstvou jsou **ODBC ovladače**. Ty provedou zpracování volané ODBC funkce, přeložení požadavku do SQL pro příslušný SŘBD (DBMS) a jeho následné poslání.

- Poslední vrstvou je **SŘBD**, který provede zpracování operace požadované ODBC ovladačem a výsledky této operací mu vrátí.



Open Database Connectivity (ODBC) is Microsoft's strategic interface for accessing data in a heterogeneous environment of relational and non- relational database management systems. Based on the Call Level Interface specification of the SQL Access Group, ODBC provides an open, vendor- neutral way of accessing data stored in a variety of proprietary personal computer, minicomputer, and mainframe databases.

ODBC alleviates the need for independent software vendors and corporate developers to learn multiple application programming interfaces. ODBC now provides a universal data access interface. With ODBC, application developers can allow an application to concurrently access, view, and modify data from multiple, diverse databases.

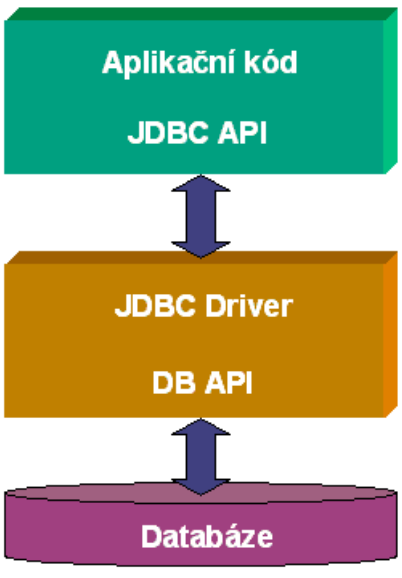
ODBC is a core component of Microsoft Windows Open Services Architecture. Apple has endorsed ODBC as a key enabling technology by announcing support into System 7 in the future. With growing industry support, ODBC is quickly emerging as an important industry standard for data access for both Windows and Macintosh applications.

JDBC (Java Database Connectivity)

- jeho API poskytuje základní rozhraní pro unifikovaný přístup k databázím, aplikační programátor je tak odstíněn od specifického API databáze a může se naučit pouze jednotné rozhraní JDBC
- lze použít i mimo databáze – pro přístup k datům ve forme tabulek (CSV, XLS, ...)
- ovladače jsou k dispozici pro většinu databázových systémů

inspirováno rozhraním ODBC:

- objektové rozhraní
- strukturovanější a přehlednější
- možnost spolupráce s ODBC



JDBC ovladač

- zprostředkovává komunikaci aplikace s konkrétním typem databáze
- implementován obvykle výrobcem databáze
- dotazovací jazyk – SQL
    - předá se databázi
    - ovladač vyhodnotí přímo
- reprezentován specifickou třídou
    - sun.jdbc.odbc.JdbcOdbcDriver
    - com.mysql.jdbc.Driver

Typy JDBC ovladačů

Typ 1:

- využívá ODBC (pres JDBC-ODBC bridge)
- Obtížně konfigurovatelné

Typ 2:

- komunikace s nativním ovladačem nainstalovaným na počítači

Typ 3:

- komunikuje s centrálním serverem (Network Server) sítovým protokolem
- pro rozsáhlé heterogenní systémy, velmi efektivní i díky poolingu připojení

Typ 4:

- založen ciste na jazyce Java
- prímý prístup do databáze

Java Hibernate

Hibernate je framework napsaný v jazyce Java, který umožňuje tzv. ORM – Objektově-Relační mapování. Usnadňuje řešení otázky zachování dat z objektů i po ukončení běhu aplikace. Provádí podobné věci jako např. JPA – Java Persistence API.

Co dělá hibernate

Hibernate poskytuje způsob, pomocí nějž je možné zachovat stav objektů mezi dvěma spuštěními aplikacemi. Říkáme tedy, že udržuje data persistentní. Dosahuje toho pomocí ORM, což znamená, že mapuje Javovské objekty na entity v relační databázi. K tomu používá tzv. mapovací soubory, ve kterých je popsáno, jakým způsobem se mají data z objektu transformovat do databáze a naopak, jakým způsobem se z databázových tabulek mají vytvořit objekty. Druhý způsob, jak mapovat objekty, je použít anotace místo mapovacích souborů. V Hibernate tedy pracujete se svými normálními business objekty, pouze pro každý atribut přidáte get/set metody a metody hashCode() a equals(). Nutno podotknout, že nelze použít EJB(viz.Java Bean), ale pouze tzv. POJO(Plain Old Java Object). Poté, co máte objekty uložené v databázi se na ně můžete dotazovat jazykem HQL (Hibernate Query Language), který je odvozen z SQL a je mu tedy velice podobný.

Objektovou struktura mého programu předhodím Hibernatu a základě tohoto objektového modelu (a označením objektů které tam požaduju) si Hibernate vytvoří vlastní schéma, aby věděl kde jsou data uložena.

Výhody používání Hibernate

Hibernate, framework pro perzistentní vrstvu, usnadňuje programátorovi práci tím, že nemusí transformovat objekty do relací ručně, ale přenechá to perzistentní vrstvě. Zároveň jsou tím odstíněna specifika jednotlivých databází – programátor používá API Hibernate.

Objektově relační mapování

S relačními databázemi přichází i potřeba podpory **objektově relačního mapování**, což je proces překladu objektů na tabulkovou reprezentaci a překlad vazeb a vztahů mezi těmito objekty do dodatečných tabulek

2.1.13. Distribuované databáze – koncepce distribuovaného databázového systému, replikace a fragmentace dat, distribuovaná správa transakcí.

Distribuovaná DB je množina databází, která je uložena na několika počítačích. Uživateli se může jevit jako jedna velká databáze. V DB neexistuje žádný centrální uzel nebo proces odpovědný za vrcholové řízení funkcí celého systému. Výrazně to zvyšuje odolnost systému proti výpadkům jeho částí. Data i funkce rozděleny mezi více počítačů

Charakteristické vlastnosti

Distribuovaná DB je charakterizována:

- Transparentností – z pohledu klienta se zdá, že data jsou zpracována na jednom serveru v lokální databázi. Uživatel si rozdělení nemusí být vědom.
- Rozšiřitelností – zvýšení výkonu přidáním dalších počítačů.
- Robustností – výpadek jednoho počítače neovlivní funkci ostatních
- Jsou syntakticky shodné příkazy pro lokální i vzdálená data, nespecifikuje se místo uložení dat (řeší to distribuovaný SŘBD)
- Autonomností – s každou lokální bází dat zapojenou do distribuované databáze je možno pracovat nezávisle na ostatních databázích.
- Lokální Db je funkčně samostatná, připojení do jiné části distribuované db se v případě potřeby zřizují dynamicky.
- Nezávislost na počítačové síti – jsou podporovány různé typy architektur lokálních i globálních počítačových sítí (LAN, WAN)
- V distribuované databázi mohou být zapojeny počítače i počítačové sítě různých architektur, pro komunikaci se používá jazyk SQL.

Distribuované databáze jsou výhodné kvůli:

- Lokální autonomie – odpovídají struktuře decentralizovaných organizací. Data jsou uložena v místě nejčastějšího využití a zpracování – zlevnění provozu
- Zvýšení výkonu (rozdělení zátěže na více počítačů)
- Spolehlivosti (replikace dat, degradace služeb při výpadku uzlu, přesunutí na jiný uzel)
- Rozšiřitelnosti
- Schopnosti sdílet informace integrací podnikových zdrojů
- Agregaci informací – z více bází dat lze získat informace nového typu.

Naopak Distribuované Db mohou způsobit i pro ně specifické problémy:

- Složitost – distribuce db, distribuce zpracování dotazu a jeho optimalizace, složité globální transakční zpracování, distribuce katalogu, paralelismus a uvíznutí, složité zotavování z chyb
- Cena (komunikace je navíc)
- Bezpečnost
- Obtížný přechod – neexistuje automatický konverzní prostředek z centralizovaných DB na DDB

Problémy replikace a fragmentace dat

U fragmentace a replikace bychom měli respektovat hlediska:

- Rozdělit relace do lokálních serverů tak, aby aplikace zatěžovali servery stejnoměrně.
- Přístupnost a spolehlivost - Replikací zlepšíme spolehlivost a read-only dostupnost.
- Lokality zpracování (maximalizovat lokální).
- Dostupnosti a ceny paměti v jednotlivých uzlech

Replikace dat

Replikační transparentnost je neobtěžovat uživatele skutečností, že pracuje s daty existujícími ve více kopiích = uživatel neví o replikách.

Při replikaci dat se nachází kopie množiny objektů v každém uzlu, ve kterém je využívaná. V systému tedy existuje několik kopií každého objektu. Výhodou tohoto řešení je kvalitní dostupnost, rychlý přístup ke každému objektu a menší nároky na komunikaci mezi uzly. Nevýhodou je problém duplicity, díky níž je nutné trvale zajišťovat konzistenci všech kopií. Je nutné implementovat systém, který určí správné pořadí provedených operací a při replikaci rozdistribuje správnou kopii. Také je nutné zabránit současné modifikaci dvou kopií objektu.

Správné pořadí provedených operací je určováno většinou časovými razítky, současné modifikaci dvou kopií jednoho objektu mohou zabránit klasické paralelní synchronizační prostředky (zámky, semafory apod.)

Fragmentace dat

Fragmentační transparentnost = uživatelův dotaz je specifikován na celou relaci, ale musí být vykonán na jejím fragmentu = uživatel neví o fragmentech.

Fragmentace dat se dělí na:

- Horizontální – dle selekční podmínky rozdělíme tabulku na 2 části horizontálním řezem,tedy např. na knihy s ISBN nižším než 122 a na knihy s ISBN větším nebo rovným 122.
- Odvozená horizontální – dochází k rozdělení tabulky na 2 a více částí horizontálním řezem, v tomto případě však je fragmentace založena na jiné relaci. Např. DODAVATELE a KNIHY. DODAVATELE rozdělíme do fragmentů a na základě těchto fragmentů provedeme fragmentaci v tabulce KNIHY, která je spojena s DODAVATLE relací.
- Vertikální – rozdělení tabulky podle sloupců na 2 a více částí. V jedné skupině jsou jedny sloupce, v druhé jiné.
- Smíšená – tabulku např. rozdělíme dle sloupců a následně v jednotlivých částech provedeme horizontální fragmentaci.

2.1.14. Temporální databáze, porovnání klasických a temporálních databází, modely času, vztah událostí a času (snapshot), temporální SQL.

Temporální databáze jsou databáze určitým způsobem podporujícím čas. Čas potřebujeme v databázích např. ve studijním informačním systému, účetních a bankovních systémech, docházkových systémech. Hlavní cíl temporálního DM by měl být zachytit sémantiku dat měnící se v čase. V praxi existuje mnoho nekompatibilních datových modelů s mnoha dotazovacími jazyky.

Temporální databáze (temporální databázový systém) je databáze (databázový systém) zohledňující časové vlastnosti ukládaných dat

Jméno	Plat	Funkce	Datum narození	Platí_od	Platí_do
Pepa	60000	Vrátný	1945-04-09	1995-01-01	1995-06-01
Pepa	70000	Vrátný	1945-04-09	1995-06-01	1995-10-01
Pepa	70000	Vrchní vrátný	1945-04-09	1995-10-01	1995-02-01
Pepa	70000	Ředitel bezpečnosti	1945-04-09	1996-02-01	1997-01-01

Porovnání klasických a temporálních DB

Klasický DB systém

Zachycuje stav systému v aktuálním časovém okamžiku. Problém: co dělat se starými daty. V případě, že se systém v čase vyvíjí, změny se v DB projeví přidáním nových informací a mazáním starých. Klasické DB neobsahují informaci o čase. V případě, že požadujeme uchování historie změn, či alespoň předchozího stavu, je nutné do DB doplnit informace o čase. Aktualizaci a operace s časem musí zajistit uživatel, což není triviální.

- Neobsahují informaci o čase
- V databázi zachycen pouze aktuální stav systému. V případě, že se v čase systém vyvíjí, změny se v databázi projeví přidáváním nových informací a mazáním starých.
- V případě, že požadujeme uchovávání historie změn, či alespoň předchozího stavu, je nutné do databáze doplnit informaci o čase. Aktualizaci a operace s časem musí zajistit uživatel. Což (jak ilustrujeme dále) není triviální.
- Jako příklad poslouží SIS, kde chceme uchovávat informace o předcházejících semestrech. Řešením je přidání sloupce, který identifikuje konkrétní semestr. Nevýhodou tohoto řešení je, že s touto informací musí manipulovat uživatel sám.

Temporální DB

Databáze určitým způsobem podporující čas. Poskytuje vhodný dotazovací jazyk zahrnující práci s časem – výhodou jsou jednodušší dotazy, v nichž se vyskytuje čas, což přináší méně chyb v aplikačním kódu a zajišťuje jednodušší udržování aplikací.

Temporální projekce – jako projekce v klasických databázích (z celé relace jsou vybrány hodnoty podle zadaných atributů), navíc bere v úvahu čas. V případě, že dvě n-tice výsledku mají stejné hodnoty všech svých atributů a překrývají se nebo dotýkají se časem, srostou tyto dvě n-tice s časem odpovídajícím sjednocení obou n-tic.

Temporální spojení – stejné jako spojení (JOIN) v klasických DB (zadáme sloupce a podmínku, která říká, kdy jsou dva řádky tabulky spojeny), navíc bere v úvahu čas události. V Temporálních DB nemusíme zadávat sloupce uchovávající čas, pouze podmínku na spojení pro čas.

- Konkrétní podporu času uvidíme později
- Vhodný dotazovací jazyk zahrnující práci s časem
- Výhodou jsou jednodušší dotazy v nichž se vyskytuje čas, což přináší méně chyb v aplikačním kódu

Modely času

Temporální logika: čas je libovolná množina okamžiků s daným uspořádáním. Modely času se rozlišují podle:

Dle uspořádání

- **Lineární** – čas roste od minulosti k budoucnosti lineárně
- **Větvený** (čas možných budoucností) – lineární minulost až do teď, pak se větví do několika časových linií reprezentujících možný sled událostí. Každá linie se může dále větvit.
- **Cyklický** – opakující se procesy. Př. týden, každý den se opakuje po sedmi dnech.

Dle hustoty

- Diskrétní – spolu s lineárním uspořádáním. Každý okamžik má právě jednoho následníka.
- Hustý – Mezi každými dvěma okamžiky existuje nějaký další
- Spojitý – každé reálné číslo odpovídá bodu v čase.

- Omezenost času – Omezený – nutnost zejména kvůli reprezentaci v počítači, Neomezený.
- Absolutní /relativní čas – Absolutní se vyjádří hodnotou, také ale potřebuje počátek. Relativní vyžaduje nějaký počátek, čas se pak vyjádří jako vzdálenost a směr od počátku.

Datové typy pro čas jsou:

- Časový okamžik (instant) – DATE, TIME, TIMESTAMP
- Časový úsek (time period) – doba mezi dvěma časovými okamžiky (15:30-16:00)
- Časový interval (interval) – doba o specifikované délce, ale bez konkrétních krajních bodů (30 minut)
- Množina časových okamžiků (instant set)

- Množina časových úseků (temporal elements)

Vztah událostí a času (snapshot)

Čas platnosti (valid time)

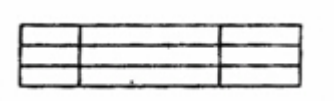
- Čas, kdy byla událost pravdivá v reálném světě. Může být v minulosti přítomnosti i budoucnosti.
- Reprezentace času platnosti – časový okamžik, doba, časový úsek, množina okamžiků
- Čas platnosti může být přidružen k atributům, množině atributů, celé n-tici nebo objektu

Transakční čas (transaction time)

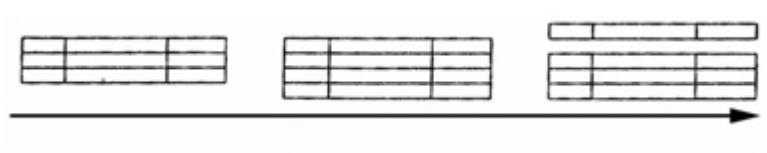
- Čas, kdy byl fakt reprezentován v DB. Nabývá pouze aktuální hodnoty. Monotónně roste.
- Reprezentace transakčního času – časový okamžik (nová n-tice se stejným klíčem – logické odstranění původní), časový úsek (teď, dokud nezměněno), tři časové okamžiky (čas zaznamenání začátku v reálném světě, konce události v reálném světě, logického odstranění události z db), množina časových úseků

Snapshot

Datový model nepodporující čas platnosti ani transakční čas. Je to klasický relační model. Každá n-tice je fakt platný v reálném světě. Při změně reálného světa jsou do relace prvky přidávány nebo z ní odebírány.

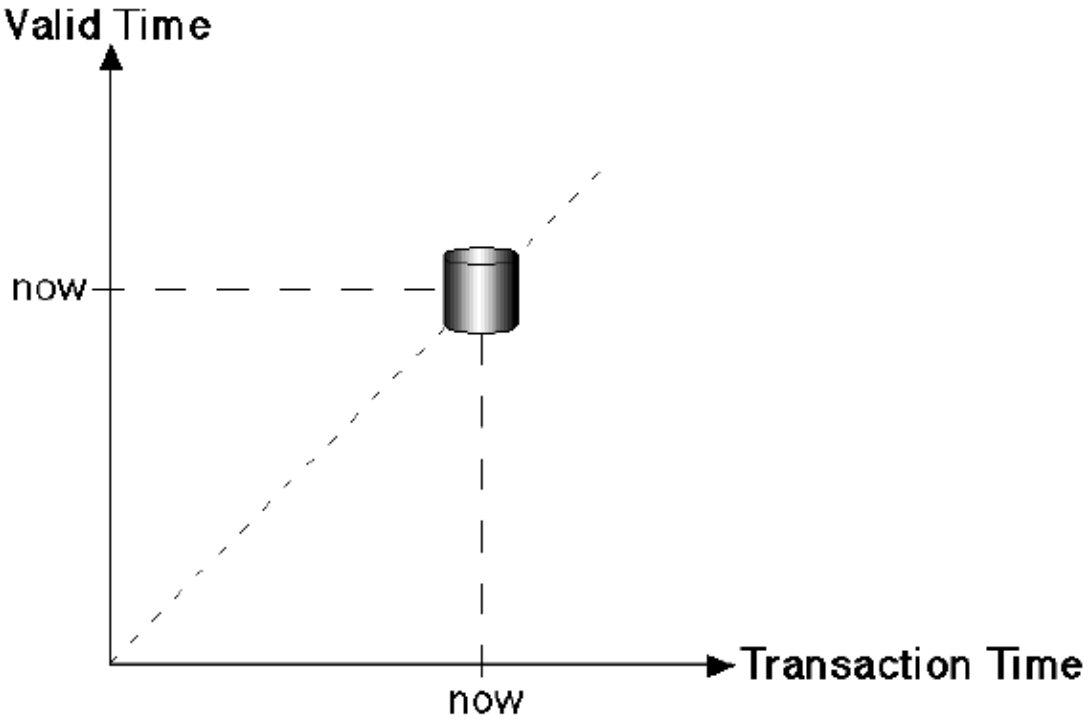


Další datové modely mohou být **valid-time** relace (podporuje čas platnosti, umožňuje klást dotazy o faktech z minulosti i budoucnosti), **transaction-time** relace (podporuje pouze transakční čas, umožňuje získat informaci ze stavu db v nějakém okamžiku v minulosti), bitemporální, temporální.

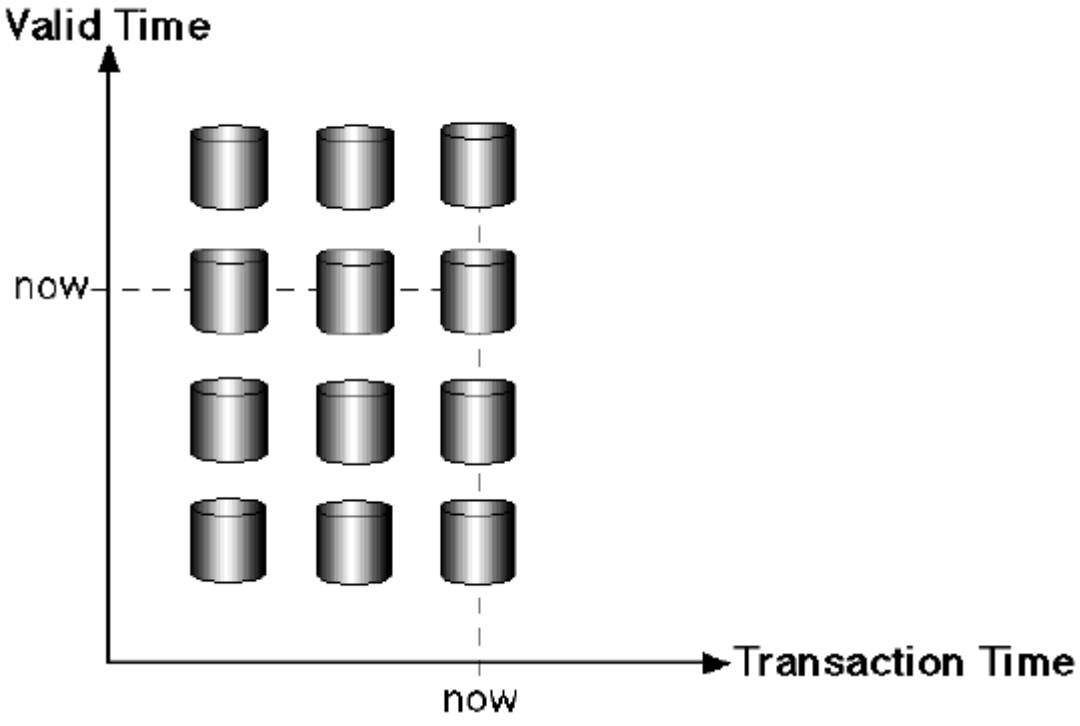


Obr – transaction-time

Snapshot database:



Temporal database:



Temporální SQL

Temporální datový model obsahuje objekty s přesně danou strukturou, omezení pro dané objekty a operace na daných objektech (temporální dotazovací jazyky). Temporálních dotazovacích jazyků je velké množství. Nejčastěji jsou založené na SQL. Typy jsou:

- Relační – HQL, HSQL, TDM, TQuel, TSQL, TSQL2
- Objektově orientované – Matisse, OSQL, OQL, TMQL

TSQL2

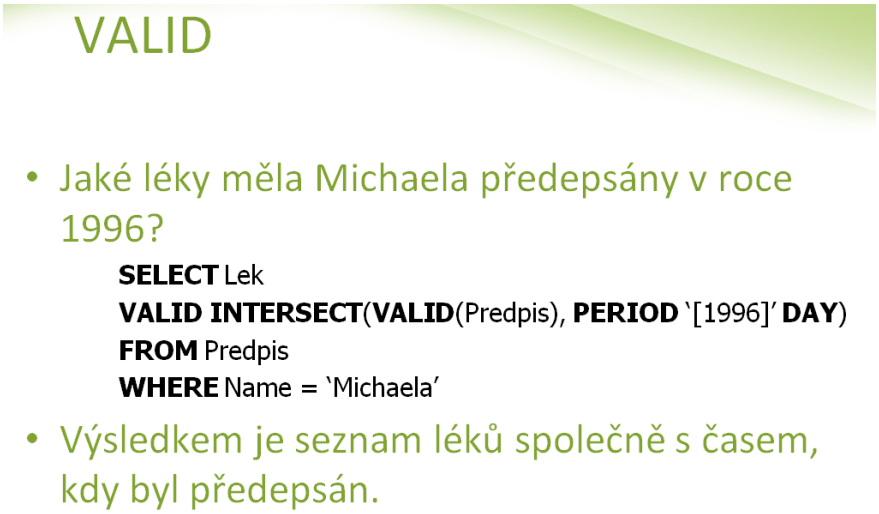
Temporal SQL 2 – měl sjednotit přístupy k temporálním datovým modelům. Je to nadmnožina SQL92.

V TSQL2 je časová osa na obou koncích omezena, ale dostatečně daleko (18 miliard let). U časových údajů jsou možné různé granularity. Časové typy: DATE, TIME, TIMESTAMP, INTERVAL, PERIOD.

Datový model je bitemporální. Řádek je orazítkován množinou bitemporálních chrononů. Bitemporální chronon je dvojice (chronon transakčního času, chronon času platnosti). Př. Relace ZAMESTNANEC – umístění lidí v odděleních určitého podniku. Schéma (Jméno, Oddělení) + časové razítko.

Př. SELECT – komu byl předepsán nějaký lék – výsledek bez podpory času

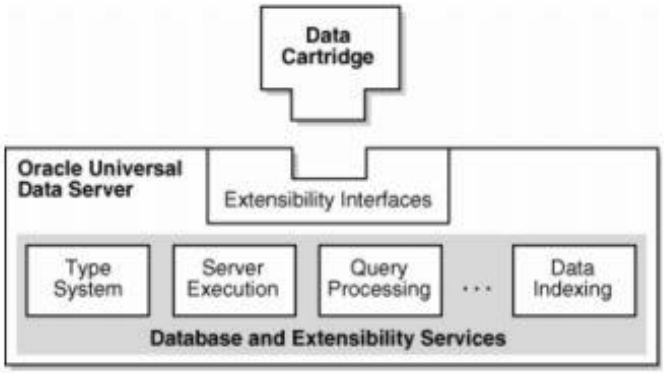
```
SELECT SNAPSHOT Jmeno FROM Predpis
```



2.1.15. Uživatelské rozšíření databázových systémů – data cartridge, příklady použití.

- rozšiřitelnost = možnost přidávání datových typů a programů (funkcí) zabalených do speciálního modulu
- uživatelsky definované
  - typy = UDT
  - funkce = UDF
- problém: zapojení do relačního SŘDB včetně SQL
  - DB/2 = relační extendery
  - Informix = DataBlades
  - Sybase = Component Integration Layer
  - Oracle = cartridges

Oracle Extensibility Framework



- **Data cartridge**
  - způsob uživatelského rozšíření databázového serveru Oracle
  - rozšíření je pouze na straně serveru
  - jsou integrované se serverem přes rozhraní
  - jsou v podobně balíků = instalují se jako celek
- **Extensibility interface**
  - rozhraní, které umožňuje vytvářet cartridge jednotnm způsobem
  - poskytuje přesně definovaný způsob, jak se serverem komunikovat
  - zpřístupňuje jednotlivé standardní služby, které lze v serveru rozšířit
- **Database and extensibility services**
  - sada standardních služeb, které poskytuje server
  - lze je využívat a rozšiřovat pomocí cartridge
  - jednotlivé služby jsou:
    - **Extensible type system**
      - podpora pro uživatelské typy, kolekce, reference (REF), LOBy
    - **ExtensibleServer execution environment**
      - podpora vlastních procedru a funkcí
    - **Extensible Indexing**
      - vlastní způsob indexování = tzv. doménové indexování pro doménově specifická data

        - **Extensible Optimiser**
            - podpora pro vytváření uživatelských funkcí a indexů pro vlastní sběr statistik pro CBO
    - Pracuje na principu tzv. cartridge
        - způsob uživatelského rozšíření databáze Oracle
            - integruje se pomocí sady rozhraní pro jednotlivé přístupy (k datům, indexům,…)
        - rozšíření může definovat
            - nové datové typy(standardní, pole, hnízděné tabulky, LOBy) a jejich funkce
            - nové typy indexů
            - nové operátory
        - proč implementovat DataCartridge
            - nutnost zpracování komplexních dat, která neodpovdají standardním
            - relačním informacím
            - nutnost snadné nanipulace s takovými daty
        - příklady
            - mulitoborové = datové, statiskické výpočty, prostorové databáze, multimédia
            - specializované, finanční a právní systémy
    - typická struktura cartridge
        - definice nových objektových typů
        - implementace těl typů, balíky, procedury, funkce
        - případné DLL knohovny s implementací v C
        - operátory
        - doménové indexy pro porporu operátorů
    - př. standardní cartridge: podpora indexování a vyhledávání v textech

LOB - Large Objects

- standardní typy pro ukládání objemných dat na serveru (až 4 GB)
- typy
    - Externí
        - BFILE = samostatný binární soubor uložený vně databáze
    - Interní
        - CLOB = zankový typ v UTF
        - NCLOB = znakový typ v národní sadě
        - BLOB = binární typ
- ve sloupci tabulky uložen pouze deskritpro odkazující na samotná data
    - hodnoty pro xLOB slupce = NULL, EMPTY_CLOB(), EMPTY_BLOB(), EMPTY_NCLOB()
- manipulace
    - oracle nabízí balík DBMS_LOB s řadou funkcí a procedru pro standardní manipulaci s daty
    - provádí se po částech pomocí bufferů
- indexace
    - není standardně indexováno, ale je možné implementovat svá vlastní rozhraní pro indexaci

Rozšíření serveru

- server dovoluje psát implementace procedru v řadě jazyků
    - nativním PL/SQL
    - Javě
    - v čemkolikv s konvecní jazyka C a kompilací DLL

2.1.16. Dokumentografické systémy, fulltextové vyhledávání, filtrace, disambiguace, lemmatizace, indexy, tezaury, dotazování.

Dokumentografické systémy (DIS)

- vznik 50. léta 20. stol. za účelem automatizace postupů používaných v knihovnictví
- Nyní samostatná podčást IS
    - Faktografický IS - informace s definovanou vnitřní strukturou (nejčastěji tabulky)
    - Dokumentografický IS - informace v podobě textu v přirozeném jazyce bez pevné vnitřní struktury
- Práce s DIS:
- *Zadání dotazu*
- *Porovnání*
- *Získání seznamu odpovídajících dokumentů*
- *Ladění dotazu*
- *Vyžádání dokumentu*
- *Obdržení textu*
- Struktura DIS:
- Systém zpřístupnění dokumentu - sekundární informace o dokumentu (Autor, Název, ...)
- Systém dodání dokumentu - někdy není řešeno pomocí SW

Vyhodnocení dotazu

- přímé porovnávání náročné na čas
- nutné vytvořit model dokumentu
- ztrátový proces, založený na identifikaci slov v dokumentech
- výsledkem strukturovaná data vhodná pro porovnávání
- dotaz se upraví do odpovídající podoby a porovná se s modelem dokumentů

Text

Předzpracování

- vyhledávání nad modelem efektivnější, ale lze použít jen informace z modelu
- cíl: vytvořit model, zachovávající nejvíce info z původního textu
- problem: nejednoznačností (ambiguity)
- dosud neřešené nároky na encyklopedické i asociativní znalosti

Porozumění textu

- Homonymie slov
    - jedno slovo může mýt stejný tvar pro různé pády a další gram. jevy
        1. kontroly: 1.p.m.č., 2.p.j.č. - není zřejmé jestli více kontrol nebo jedna kontrola
    - jeden tvar může mít různý význam
        1. hnát - sloveso, podst. jm.
        2. pět - číslovka, sloveso

- přiřazení je závislé na osobě, která dokument píše nebo čte
    - dva lidé mohou jednomu slovu přikládat zcela nebo částečně jiný význam
    - dva lidé si i pod stejným významem mohou představit jiný konkrétní předmět nebo množinu
        1. máma, pokoj, ...
- výsledkem situace, kdy dva různí čtenáři nemusí přečtením získat stejnou informaci jako autor, ani navzájem.
- Homonymie a nejednoznačnosti narůstají při přechodu od slov k větám.

Přesnost a úplnost

- důsledek nejednoznačnosti: žádný existující DIS nedává ideální výsledky
- Pro zobrazení odpovědi na dotaz lze určit
    - Nv (počet vrácených dokumentů) - O nich si DB myslí, že jsou relevantní, odpovídající dotazu
    - Nvr (počet vrácenýcy relevantních dok.) - o nich si tazatel myslí, že uspokojí jeho požadavky
    - Nr (počet všech relevantních dok. v DB) - problematické u velkých DB
- Kvalita výsledné množiny se měří na základě:
    - Přesnost (Precision): P = Nvr / Nv
        1. pravděpodobnost, že dokument zařazený v odpovědi je skutečně relevantní
    - Úplnost (Recall): R = Nvr / Nr
        1. pravděpodobnost, že skutečně relevantní dokument je zařazený v odpovědi
- koeficienty jsou opět závislé na subjektivním názoru tazatele
- dokument vrácený na výstupu může uspokojovat požadavky dvou uživatelů, kteří položili stejný dotaz, různou měrou
- ideální případ: P == R == 1

Kritéria

- **Kritérium predikce**
    - při formulaci dotazů je třeba uhádnout, které termy (slova) byly v dokumentu autorem použity pro vyjádření dané myšlenky
        1. problémy způsobují
            1. synonyma - autor používá synonyma, které si tazatel nemusí při dotazu uvědomit
            2. překrývající se význam slov
            3. opisy jedné situace jinými slovy
    - částečné řešení - zařazení tezauru, který obsahuje
        1. hierarchie slov a jejich významů
        2. synonyma slov
        3. asociace mezi slovy
    - tazatel může tezaurus využít při formulaci svých dotazů
    - při ladění dotazů má uživatel tendenci postupovat konzervativně
        1. v dotazu často zůstávají ty části, které uživatele napadly na začátku a mění se jen podružné části, které nekvalitní výsledek nemusí zásadně ovlivnit
    - vhodné je uživateli pomoci s odstraněním nevhodných částí dotazu, které nepopisují relevantní dokumenty a naopak s přidáváním formulací, které relevantní dokumenty popisují

- **Kritérium maxima**
    - tazatel obvykle není schopen (ochoten) procházet příliš mnoho dokumentů do té míry, aby se rozhodl, zda jsou pro něj relevantní nebo ne
    - obvykle 20-50 podle velikosti
    - potřeba nejen dokumenty rozlišovat na odpovídající/neodpovídající dotazu, ale řadit je na výstupu podle míry předpokládané relevance
    - v důsledku kritéria maxima se při ladění dotazu uživatel obvykle snaží zvýšit přesnost
        1. malé množství dokumentů v odpovědi, obsahující co největší poměr relevantních dokumentů
    - některé oblasti použití vyžadují co nejvyšší přesnost i úplnost
        1. např. právnictví

Modely dokumentografických systémů

Úrovně modelů:

- rozlišují (ne)přítomnost slov v dokumentech
- rozlišují frekvence výskytů slov
- rozlišují pozice výskytů slov v dokumentech

Boolský model

- vznik 50. léta 20. stol., automatizace postupů používaných v knihovnictví
- Databáze obsahuje dokumenty, dokumenty popisovány pomocí termů, reprezentace dokumentu pomocí množiny termů (obsažených v dokumentu, popisujících význam dokumentu)
- **Indexace**
    - Přiřazení množiny termů, které jej popisují ke každému dokumentu
        1. Ruční - nekonzistence
        2. Automatická - konzistentní, ale bez porozumění textu
        3. Řízená - předem daná množina termů
        4. Neřízená - množina termů se mění s přibývajícími dokumenty
    - Tezaurus - vnitřně strukturovaná množina termů
        1. Synonyma s preferovanými termy
        2. Hierarchie užších/širších termů
        3. Příbuzné termy
        4. ...
    - Stop-list - nevýznamová slova
    - Příliš obecná slova nejsou pro identifikaci dokumentů vhodná, příliš specifická slova také ne
    - dotaz vyjádřen logickým výrazem: AND, OR, NOT
    - Příklad dotazu: počítač AND NOT osobní
    - Víceslovné termy: počítač AND NOT osobní počítač
    - Organizace indexu:
        1. Invertovaný seznam - pro každý term je seznam dokumentů, ve kterých se vyskytuje
        2. Zpracování dokumentů na vstupu - vnikne posloupnost dvojic <dok_id,term_id>

3. Setřídění dle term_id,dok_id

- **Nevýhody**
  - formulace dotazů je spíše uměním než vědou
  - nemožnost ohodnotit vhodnost vystupujících dokumentů ¨
  - všechny termy v dotazu i v identifikaci dokumentu jsou chápány jako stejně důležité
  - nemožnost řízení velikosti výstupu
  - některé výsledky neodpovídají intuitivní představě

Vektorový model

- vznik 70. léta 20. stol., cca o 20 let mladší než Boolské DIS
- snaha minimalizovat nebo odstranit nevýhody Boolských DIS
- Struktura:
  - databáze obsahuje dokumenty
  - dokument popisován pomocí množiny termů
  - term je slovo nebo sousloví
  - reprezentace dokumentu pomocí vektoru vah termů

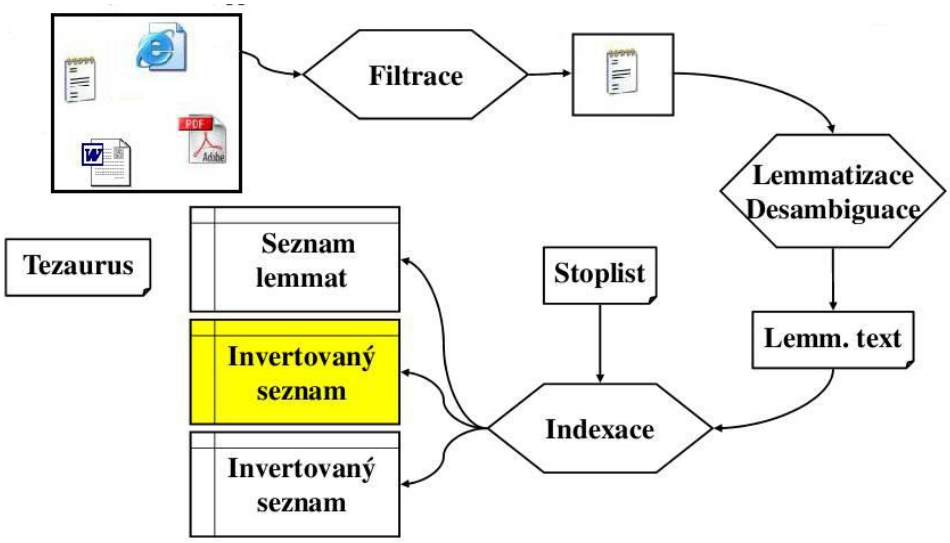
Fulltextové vyhledávání

Filtrace, Disambiguace, Lemmatizace, Indexy, Tezaury, Dotazování

- Odlišné od principů běžného vyhledávání
  - neprohledávají se striktně strukturovaná data, kde má každý sloupec každé tabulky předem daný význam
  - prohledávají se volně psané texty, kde může být stejná událost popsána více autory rozdílně - různá slova stejného významu, různé slovní obraty a opisy
- DB systémy využívají svých prostředků rozšiřitelnosti a dodávají standardně prostředky, které vyhledávání v textových datech umožňují
- Rozdílné přístupy a možnosti
  - neexistuje objektivně nejlepší řešení
  - výsledky navíc podléhají subjektivním názorům tazatelů
- Samotná formulace dotazu, který by vrátil všechny dokumenty, které tazatele zajímají a žádné jiné, obvykle nelze zformulovat
  - spolu s vyhovujícími - relevantními - odpověďmi se obvykle vrací i odpovědi nerelevantní
- **Problémy**
  - Hononyma - ptá se tazatel dotazem "koruna" na finanční, lesnické či panovnické dokumenty?
  - Synonyma
    1. Vyhovuje dokument o "krychlích" dotazu na dokumenty o "kostkách"?
    2. Vyhovuje dokument o "stromech" dotazu na "souvislé grafy bez cyklů"?
  - Hierarchie významů
    1. Zvíře - Savec - Šelma - Medvěd
    2. Tiskovina - Časopis
  - Ohebnost slov - Jít, Jde, Jdu, Jdou, ...
- Striktní boolská logika není pro formulaci dotazů příliš vhodná
  - Dokument buďto vyhovuje nebo nevyhovuje

- Dotazování v textech vyžaduje třídit odpovědi podle předpokládané vhodnosti pro tazatele - je potřebné mít možnost definovat míru shody dotazu s dokumentem

Předzpracování

- Databáze obvykle používají některý z boolských modelů reprezentace dokumentů
  - nejlépe odpovídá běžným dotazům
  - relativně snadno se implementuje
  - dotazy jsou ve formě boolských formulí, ve kterých operandy tvoří jednotlivá slova - řada různých modifikací
- 

Jednotlivé kroky:

- *Filtrace* - odstraní formátovací značky a nechá čistý ASCII text
- *Desambiguace* - určí význam slova podle kontextu
  - "pět chválu" ... sloveso pět
  - "pět vozidel" ... číslovka pět
- *Lemmatizace* - určí základní tvar slova a gramatický tvar v dokumentu, často nahrazen pomocí stemmeru, který hledá kmen slova
- *Indexace* - vytvoří pomocné seznamy lemmat a dokumentů a invertovaný soubor
  - dvojice *[id_dok, id_lemmatu]* setříděné dle id_lemmatu a zbavené duplicit
  - dnes obvykle více informací, např. pětice *[id_dok,č_odstavce,č_věty,č_slova,id_lemmatu]* - dovoluje vyhodnocování tzv. proximitních omezení na vzdálenost slov v dokumentu

Large OBjects (LOB)

- pro podporu vyhledávání je potřeba nad textovým sloupcem vytvořit index - invertovaný soubor
- běžné textové sloupce jsou pro tyto účely krátké a nevyhovující
  - obvykle se takto indexují sloupce některého z LOB (Large OBject) typů
- LOBy
  - standardní typy pro ukládání objemných dat na serveru, definováno v SQL-92 Full
  - až 4GB dat
    1. BLOB - standardní binární typ

2. CLOB - znakový typ v univerzální znakové sadě
3. NCLOB - znakový typ v národní znakové sadě

- v Oracle navíc externí typ BFILE
  1. pouze pro čtení
  2. samostatný binární soubor uložený vně databáe v OS
- v MS SQL
  1. Image - binární data do velikosti 2 GB
  2. Text - textová data do velikosti 2 GB
  3. NText - textová data v národní znakové sadě do vel. 1 GB

- Ve sloupcích tabulky je uložen pouze deskriptor (tzv. LOB lokátor), odkazující na samostatně uložená data

Oracle fulltext

- Filtrování vstupních dokumentů
  - NULL_FILTER - pro textové dokumenty TXT, HTML, XML
  - INSO_FILTER - pro binární dokumenty
  - CHARSET_FILTER - pro konverzi získaných dokumentů do znakové sady databáze
- Druhy fulltextových indexů
  - CONTEXT
    1. základní typ indexu pro vyhledávání v textových datech
    2. vhodný pro větší dokumenty
    3. synchronizace indexu s daty je nutno provést explicitně
  - CTXCAT
    1. vhodný pro menší dokumenty a jejich úryvky
    2. může být zkombinován s dalšími netextovými sloupci pro kombinované dotazování
    3. synchronizace indexu s daty se provádí automaticky se změnami v tabulce
  - CTXRULE
    1. postaven na množině předdefinovaných dotazů
    2. slouží pro klasifikaci dokumentů do skupin podle toho, kterým dotazům vyhovuje
- Uložení dokumentů
  - NORMAL_DATASTORE - text je v jednom sloupci jednoho rádku
  - MULTI_COLUMN_DATASTORE - text ve více sloupcích jednoho řádku
  - URL_DATASTORE - text je na internetu, dostupný přes URL ve sloupci
- Příklad vytvoření indexu nad textovým sloupcem:

```
CREATE INDEX myindex ON doc(htmlfile)
INDEXTYPE IS ctxsys.context
PARAMETERS('datastore ctxsys.default_datastore
filter ctxsys.null_filter
section group ctxsys.html_section_group');
```

- Spolu s novými typy indexů databáze implementují nové operátory pro porovnávání dotazu s textem
- Operátory vrací číslo - očekávanou míru shody obsahu textu s tazatelovými požadavky

2.1.17. Možnosti tvorby datových skladů a metody dolování znalostí.

Základní problémy u běžných transakčních databázových systémů:

- nedosažitelnost dat skrytých v transakčních systémech
- dlouhá odezva při plnění komplikovaných dotazů
- složitá, uživatelsky nepříjemná rozhraní k databázovému softwaru
- cena v administrativě a složitost v podpoře vzdálených uživatelů
- soutěžení o počítačové zdroje mezi transakčními systémy a systémy podporujícími rozhodování

Cesta k řešení těchto problémů = datové sklady, tzv. Data Warehouse – DW

Datawarehouse

- samostatný informační systém postaven na již pořízených datech, určen především k jejich analýze
- architektura založená na relačním SŘBD, která se používá pro údržbu historických dat získaných z databází operativních dat, jenž byla sjednocena a zkontrolována před jejich použitím v databázi DW
- data z DW jsou aktualizována v delších časových intervalech, jsou vyjádřena v jednoduchých uživatelských pojmech a jsou sumarizována pro rychlou analýzu
- DW je obrovská databáze obsahující data za dlouhé časové období
- často slučuje data z více rozdílných zdrojů, které mohou obsahovat data různé kvality nebo používat nejednotné formáty a reprezentace
- objemově zabírá stovky GB až několik TB
- nemusí být databází v běžném smyslu, tj. pro přesné provádění transakcí
- je určen pro rychlé vyhledávání
- nejsou kladeny nijak důrazné požadavky na správnost a úplnost dat

Charakteristika

- data jsou uložena na různých místech ve formě relačních tabulek
    - uživatelé mohou tabulky jen číst
    - zapisovat může aktualizační program pravidelně udržující tabulky
- dotazy jsou většinou komplexní
    - podporují tzv. on-line analytické zpracování (OLAP)
        - výrazně se liší od on-line transakčního zpracování (OLTP)
        - operační databáze je přizpůsobena pro podporu OLTP
            - složité OLAP dotazy by vyústily do nepřijatelné odezvy
        - typické OLAP operace
            - roll-up (zvýšení stupně agregace)
            - drill-down (snížení stupně agregace)
            - slice-and-dice (selekce a projekce)
            - pivot (přeorientování vícerozměrného pohledu na data)

- na základě dotazu se pospojují potřebná data do vícerozměrné tabulky (nebo více tabulek), do kterých lze klást SQL dotazy
- pro častější dotazy si uchovávají předem připravené vícerozměrné tabulky
- zátěž je většinou způsobena složitými dotazy, jež přistupují k miliónům záznamů a provádějí množství operací
- data bývají modelována vícerozměrně
    - v obchodním data warehouse mohou těmito rozměry být např. čas prodeje, místo prodeje, prodavač, výrobek, …
    - rozměry mohou být i hierarchické např. čas prodeje jako den-měsíc-čtvrtletí-rok, zboží jako výrobek-kategorie-průmysl
    - spojení více tabulek pomocí odkazu na řádky jednotlivých tabulek
    - používají speciální organizaci dat, přístupové a implementační metody, jež obecně nejsou v komerčních databázových systémech určených pro OLTP podporovány

Databázový systém – OLTP (Online Transaction Processing Systems)

- zákaznicky orientovaný
- aktuální data -- lze považovat i za slabinu, při výpadku (chybě), vznikají ztráty pro byznys
- ER schéma
- sofistikované atomické transakce i přes několik systémů(bank, po síti,...)
- velikost DB až několik GB
- jednoduché a efektivní
- příkladem je bankomat

DataWarehouse – OLAP (Online analytical Processing)

- orientovaný na trh, rychlé (oproti OLTP) získání výsledků na analytické dotazy
- historická data, multidimenzionální datový model
- agregovaná data (nenormalizovaná=redundantní)
- schéma hvězdy či vločky
- převážně pouze čtení
- velikost až TB
- použití: byznys reporty o prodeji, marketing, management reporty, rozpočty, finanční předpovědi a reporty

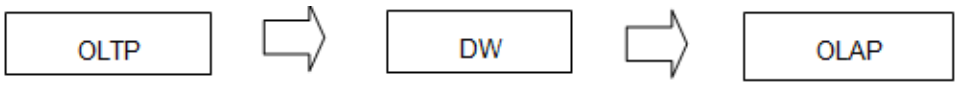
Použití DW

- prezentace dat
- testování hypotéz
- objevování nových informací

Architektura DataWarehouse

- tři úrovně:
- klient

- OLAP server (MOLAP/ROLAP server)
- databázový server DW
- data lze organizovat v tzv. multidimenzionálním datovém modelu
    - odlišný od modelu relačního
    - odpovídá mu specializovaný software, multidimenzionální SŘBD (MDD)
    - model připomíná techniku spreadsheet ve více než dvou rozměrech
    - data jsou implementována pomocí vícerozměrných polí, jejichž dimenze odpovídají dimenzím podnikání organizace
- navržení a vytvoření DW je proces skládající se z následujících bodů:
- definovat architekturu, umístění a rozčlenění dat a fyzickou organizaci
- naplánovat kapacitu, vybrat OLAP servery a nástroje
- spojit servery, klientské nástroje, zdroje přes gatewaye, drivery ODBC, ...
- navrhnout schéma a pohledy, přístupové metody, některé složité dotazy
- mít skripty pro získávání, čištění, transformaci, ukládání a aktualizaci dat
- vytvořit koncové uživatelské aplikace
- spustit data warehouse i aplikace
- vytvoření je složitý proces trvající mnohdy i několik let
- mnoho organizací proto používá Data Mart umožňující rychlejší práci



Datová tržiště (Data Mart)

- DW slouží jako základna pro extrakci množin dat, resp. jejich agregaci do dílčích (replikovaných) MDD (Multidimenzionální DB)
    - MDD může pro DW sloužit ve dvou rolích
        - ”front-end” pro DW a poskytovat uživateli služby pro realizaci analytického zpracování (DW/OLAP)
        - “front-end” jednomu (několika) systémům OLTP - alternativa za DW, tj. poskytnout uživateli s OLTP data analytickým způsobem (OLTP/OLAP) – jde vlastně o datové tržiště

Systém OLAP (OnLine Analytical Processing)

- na databázové stroje jsou kladeny specifické požadavky
- objem zpracovávaných dat
- transakční systém o velikosti gigabajtů dosáhne použitím jen jedné dimense velikosti desítek či stovek gigabajtů
- rychlost odezvy analytického systému je důležitá
- počet uživatelů současně pracujících s databází není zajímavý
    - počet pracovníků vyššího managementu je omezen
    - pro pracovníky nižších stupňů bývají údaje z datových skladů převedeny do menších specializovaných databází – datových tržišť
- s těmito omezeními se vyrovnává dvojím způsobem
- uzpůsobení stávajících systémů pro práci s vícerozměrovými daty
- přidáním modulu, který to zajišťuje a prostředků pro jeho ovládání

- v lepším případě mění způsob uložení dat, v horším“překládá” operace s vícedimenzionálními daty na operace s daty relačními
- vytvoření speciálního systému správy dat, určeného pouze pro OLAP
- umožňuje provést maximum optimalizací vzhledem k nárokům, jež jsou kladené analytickým způsobem práce - převažující způsob

Programy pro vytváření a plnění databáze (ETL - viz SI)

- převodní programy
    - načtení data z několika databází, či souborů a udělat z nich novou databázi, agregace se musí naprogramovat
- systémy znázorňující převodu dat graficky a administrátor dat namapuje zdrojová data do struktur vytvářeného datového skladu
    - výsledkem jsou buď programy (scripty) nebo přímo vykonání funkce
- moduly pro plánování jednotlivých akcí

Nástroje pro práci s daty - poslední trendy v architektuře klient/server

- nabízejí variantu tenkého klienta v podobě HTML prohlížeče

Reporting, monitorování, ad-hoc dotazy

- programy umožňující kladení dotazů a formátování odpovědí
    - nejčastěji jde o vizuální dotazovací nástroje
    - makra v tabulkovém procesoru
    - uživatelské rozhraní různě propracované:
        - zadání seskupení výsledku podle různých kriterií
        - formální kontrola dotazů
        - vytváření slovníků a metadat

MOLAP - Multidimenzionální OLAP

- datová krychle (obsahuje fakta)
- hierarchické dimenze (částečné či totální uspořádání)
    - vločkové schéma -- hlavní tabulka faktů je v relaci s dimezionálními tabulkami, přes cizí klíče, dimenzionální tabulky mohou být také v relaci s dalšími subdimenzionálními tabulkami podobně jako hlavní tabulka faktů; vytváří hierarchie dimenzí
    - hvězdové schéma -- je speciální případ vločkového, dimenzionální tabulky již nejsou v relaci s dalšími subdimenzionálními tabulkami; žádné hierarchie, jednodušší

ROLAP – Relační OLAP

- na relační architektuře založený model DW strukturou propojených DB tabulek - Relační OLAP (ROLAP) – pomalejší zpracování než MOLAP
- užívá relační nebo rozšířený relační DBMS, např server METACUBE Informix, pracuje s relačními tabulkami uspořádanými do hvězdy/vločky, adresuje pomocí klíče, data jsou neagregovaná)

2.2. PSDS

2.2.1. Informační Systémy, jejich základní vlastnosti a typy

Informační systém (IS) je systém pro sběr, udržování, zpracování a poskytování informací.

Informační systém je komplexní systém pro sběr, udržování, zpracování a poskytování informací, vždy za jistým utilitárním (prospěšným, ekonomickým) cílem.
Informační systémy jsou založené na informačních a komunikačních technologiích (ICT).

Funkce informačního systému
Konkrétní procesy (činnosti) podporující základní cíle informačního systému:

- získávání informací
- zpracování informací (evidence, organizace – pořádání, kategorizace, konverze –změna média, třídění, vyhledávání, agregace, odvozování nových informací, dolování znalostí)
- uložení informací (zaznamenávání a archivace dat, datová úložiště a datové sklady)
- přenos informací (v rámci počítačových sítí)
- zpřístupnění informací (tisk, zobrazení, vizualizace, šíření…)

Následující body vystihují vlastnosti, které by kvalitní IS s maximální výkonností měl splňovat.

- Musí obsahovat nutné informace, které uchovává, analyzuje a s potřebnou rychlostí předává procesům. Dané informace se týkají zejména vlastní činnosti firmy jako je výroba, evidence zákazníků, zásob, zaměstnanců, finance, stav a vývoj vlastních výrobků.
- Musí obsahovat informace o konkurenci, světovém trhu, trendech výroby, optimalizaci výrobních procesů, o místech působnosti firmy, o strategických cílech a podobně.
- Musí obsahovat moduly pro zjednodušení a urychlení výroby, čímž je míněno hlavně urychlení a zefektivnění návrhu výrobků, technologická příprava výroby a její řízení.
- Musí umožňovat rychlou komunikaci pracovníků firmy, jednotlivých pracovních úseků, ale musí také zahrnovat komunikaci se světem.
- Musí umožňovat z dostupných informací zpracovávat cíle a strategie firmy, koordinovat činnost různých procesů a tím přispívat k zefektivnění činnosti firmy.
- Musí nabízet rychlou komunikaci se zákazníkem přes počítačovou síť.
- Musí obsahovat další nutné moduly k vedení firmy jako jsou statistiky, mzdy, účetnictví, kompletní personalistika, sklad, oblast manažer – marketing, výroba a další.

Životní cyklus IS

Zkušenosti ukazují, že je účelné **tvorbu (analýzu, návrh a realizaci) IS organizovat do posloupnosti etap**, mluvíme o tzv. **životním cyklu IS**.

Životní cyklus IS není statická sekvence činností. Popisuje dynamiku vývoje, měnící se vnější podmínky (změny legislativy), postup od obecného ke speciálnímu. Životní cyklus IS začíná prvotní představou o systému a končí vyřazením systému z provozu. Samozřejmě není znám přesný a úplný (matematický) model životního cyklu. Nepřesné modely ŽC jsou však nepostradatelné pro řízení projektů.

Základní fáze životního cyklu IS:

- Stanovení globálních cílů, specifikace požadavků, specifikace vlastností, které by měl budoucí systém realizovat (implementovat).
- Analýza systému, tvorba analytického, logického modelu systému, model požadovaných vlastností systému, stanovení obsahu IS.
- Návrh (design), specifikace způsobu, jak požadované vlastnosti implementovat, jaké zvolit technické řešení
- Implementace systému, převedení navrženého systému do funkční aplikace
- Testování vytvořeného kódu
- Nasazení systému, provoz, údržba

Typy informačních systémů

Rozdělení dle technologie zpracování dat

*Jedním z kriterií je rozdělení IS dle technologie zpracování dat na **transakční** (operativní, provozní)* - OLTP (Online Transaction Processing) a ***analytické*** - OLAP (Online Analytical Processing).

- OLTP - umožňují skupině uživatelů vykonávat bezprostředně (online) velké množství transakcí
  - relační databáze
- OLAP - analýza velkého množství údajů, většinou jen pro čteni, nadstavba OLTP
  - např. BI

Cílem OLTP systémů je umožnit skupině uživatelů vykonávat bezprostředně (online) velké množství transakcí, tj. umožnit co nejsnadnější a nejbezpečnější modifikaci uložených dat ve více-uživatelském prostředí.

Jedná se např. o podnikové, bankovní, obchodní a výrobní systémy.

Data v OLTP systémech jsou obvykle uloženy v relačních databázových systémech, tj. v mnoha normalizovaných, atomizovaných, relačně svázaných tabulkách. Relační databáze jsou optimalizovány na současné vykonávání velkého množství transakcí.

Cílem OLAP systémů je analýza velkého množství uložených údajů. Výsledky analýzy jsou souhrny a reporty, obecně informace získané z uložených dat, které slouží jako podklady pro rozhodování v oblasti řízení firem, či ekonomických a technologických procesů. Pro získání těchto informací je potřeba vykonat velké množství rozsáhlých výpočtů a agregací a to pružně a rychle (online).

Data v OLAP systémech jsou obvykle uložena ve struktuře, která umožňuje efektivní analýzu a dotazování, např. v multidimenzionálních databázích, OLAP krychlích, obecněji v datových skladech.

Data uchovávaná v OLTP databázovém systému jsou periodicky zpracovávána (agregována) a poté ukládána do datového skladu, nad nímž se posléze provádí analýzy. Datový sklad je na rozdíl od OLTP databáze určen výhradně ke čtení dat pro potřeby nejrůznějších analýz.

Rozdělení IS dle funkce

Systémy ERP (Enterprise Resources Planning)
ERP je IS, který umožňuje účelně a efektivně řídit všechny klíčové podnikové zdroje.
Systémy na komplexní podporu provozu ekonomických subjektů - firem a organizací.
ERP systémy pokrývají základní, každodenní činnosti podniku, jako jsou především finance a účetnictví, personalistika, řízení a plánování výroby, obchodní aktivity apod..

ERP systémy integrují veškerá data a procesy podniku do unifikovaného celku.
Typický ERP systém využívá k dosažení integrace množství softwarových komponent (modulů) a hardwarové infrastruktury.

V současnosti se hovoří o **systémové integraci**, jedná se o komplexní službu s cílem vytvořit a provozovat IS v podniku metodou integrace různých produktů a služeb.

Klíčovou „ingrediencí" většiny ERP systémů je použití unifikované (relační) databáze k ukládání veškerých dat. Tuto společnou databázi pak využívá široká škála jednotlivých modulů systému.

Technologický princip: aplikační software, OLTP (Online Transaction Processing), relační databáze.

Systémy pro podporu plánování, řízení a rozhodování.
Úkolem těchto systémů je poskytování informací pro plánování a řízení obvykle ekonomických subjektů. Existuje řada systémů na různé úrovni taktického a strategického rozhodování a řízení.

Systémy na podporu rozhodování
MIS - Management information system
EIS –Executive information system,
DSS – Decision support system,

Základem těchto systémů není realizace transakcí, ale prohledávání a analýza velkých objemů dat, vyhledávání informací v „shromážděných" datech,

Technologický princip: OLAP (Online Analytical Processing), datové sklady (data warehouse), dolování dat (data mining)

Systémy na podporu plánování činností
APS – Advanced Planning and Scheduling: systémy na podporu vnitropodnikového (dílenského) plánování,
SCM – Supply Chain Management: plánování dodavatelských logistických řetězců
HR – Human Resources – řízení lidských zdrojů

Systémy řízení vztahů se zákazníky
CRM – Customer relationship management: shromažďování, zpracování a využití informací o zákaznících firmy za účelem poznat, pochopit a předvídat potřeby, přání a obchodní zvyklosti zákazníků. Operativní podpora komunikace mezi firmou a jejími zákazníky.

Systémy pro tvorbu a správu dokumentů
DTP – desktop publishing
DMS – document management system
Systémy umožňující efektivní práci s elektronickými dokumenty a jejich obsahem v průběhu celého jejich životního cyklu.

Typickými procesy jsou tvorba, schvalování, evidence, digitalizace, prohlížení, editace, publikování, komunikace, sdílení, uložení, vyhledání, archivace, skartace apod.
Obvykle je zahrnuta i skupinová spolupráce, workflow management a propojení dokumentů s informacemi v ostatních (např. provozních) informačních systémech.

Technologický princip: aplikační software, obsahující nástroje pro tvorbu, publikování, fulltextové vyhledávání, řízení přístupu k elektronickým dokumentům, správu verzí, sledování historie použití a změn.

Knihovní systémy

Systémy určené k automatizaci procesů realizovaných v knihovně. Obvykle mají modulární strukturu; typické moduly jsou akvizice, katalogizace, výpůjčky apod. Zpravidla obsahuje i nástroje pro zapojení do sítě knihoven a pro komunikaci s externími zdroji.

Technologický princip: aplikační software provozního (transakčního) typu,

Geografické informační systémy (GIS)

Prostorově orientované informační systémy, provozované za podpory informačních a komunikačních technologií. Datovou základnu tvoří digitální geografické informace ve formě záznamů nebo objektů (tzv. geoprvky), s nimiž specializovaný software umožňuje provádět manipulaci (zápis a editace údajů, uložení, vyhledávání, propojování, transformace a vizualizace), lokalizaci (určení polohy), geografické analýzy a modelování (např. trojrozměrný model terénu).

Expertní systémy

Počítačové aplikace nebo systémy simulující poznávací a rozhodovací činnost experta při řešení složitých úloh s cílem dosáhnout ve zvolené problémové oblasti kvality rozhodování na úrovni experta.

Technologický princip: základní součásti tvoří báze znalostí, báze dat (faktů) k řešeným případům a řídící mechanismus (inferenční neboli odvozovací stroj, rozhodovací jádro), tj. program pro práci s těmito bázemi využívající technik umělé inteligence. Tyto základní součásti obvykle doplňuje modul pro komunikaci s uživatelem

Systémy na podporu návrhu a projekční činnosti

Jde o velkou rychle se rozvíjející oblast IT, která zastřešuje širokou škálu činnost navrhování a kreativní lidské činnosti s pomocí počítače.

- CAD (computer-aided design): počítačem podporované projektování.
- CAM (computer-aided manufacturing): strojírenství
- AEC (Architecture-Engineering-Construction), BIM (Buliding Information Model), CAAD (Computer-aided architectural design): stavebnictví a architektura
- CASE (Computer-aided software engineering): počítačem podporované procesy tvorby software. Budeme se jim podrobněji věnovat

Rozdělení IS dle skupin uživatelů

Veřejné informační systémy - Informační systémy, které jsou dostupné široké veřejnosti a poskytují veřejné informační služby. V tomto smyslu se jedná o jakékoli informační systémy bez ohledu na jejich provozovatele, obsah, typ, formu a příp. cenu poskytovaných informací a služeb.

Veřejné informační systémy Informační systémy, které jsou dostupné široké veřejnosti a poskytují veřejné informační služby. V tomto smyslu se jedná o jakékoli informační systémy bez ohledu na jejich provozovatele, obsah a příp. cenu poskytovaných informací a služeb. Opakem jsou tzv. privátní, uzavřené, neveřejné informační systémy (např. podnikové informační systémy, systémy zajišťující obranu státu, osobní informační systémy apod.).

Státní informační systém (eGovernment)
Systém, jehož účelem je podporovat činnosti provozované při výkonu veřejné správy, tj. státní správy a samosprávy a poskytovat veřejné informační služby včetně informací o subjektech veřejné správy. Představuje komplex navzájem propojených subsystémů, členěných z hlediska věcného, resortního i regionálního.

Nejdůležitější součást datové základny tvoří evidence (registry) základních skutečností nezbytných pro výkon veřejné správy: evidence obyvatel, evidence ekonomických subjektů, evidence území a územních jednotek.

IS ve zdravotnictví (eHealth)
Elektronické zdravotnictví je souhrnný název pro řadu nástrojů založených na informačních a komunikačních technologiích, které podporují a zlepšují prevenci, diagnostiku i léčbu pacientů a sledování a řízení zdravotního systému.

Privátní, uzavřené, neveřejné informační systémy (např. podnikové informační systémy, systémy zajišťující obranu státu, osobní informační systémy ad.).

Další členění informačních systémů

Veřejné informační systémy

Informační systémy, které jsou dostupné široké veřejnosti a poskytují veřejné informační služby. V tomto smyslu se jedná o jakékoli informační systémy bez ohledu na jejich provozovatele, obsah, typ, formu a příp. cenu poskytovaných informací a služeb. Opakem jsou tzv. privátní, uzavřené, neveřejné informační systémy (např. podnikové informační systémy, systémy zajišťující obranu státu, osobní informační systémy ad.).

Státní informační systém, informační systém veřejné správy

Systém, jehož účelem je podporovat činnosti provozované při výkonu veřejné správy, tj. státní správy a samosprávy, a poskytovat veřejné informační služby včetně informací o subjektech veřejné správy. Představuje komplex navzájem propojených subsystémů, členěných z hlediska věcného, resortního a regionálního.

Nejdůležitější součást datové základny tvoří evidence (registry) základních skutečností nezbytných pro výkon veřejné správy: evidence obyvatel, evidence ekonomických subjektů, evidence území a územních jednotek.

eGovernment

Moderního, přátelského a efektivního úřadu

eHealth

Elektronické zdravotnictví (eHealth) je souhrnný název pro řadu nástrojů založených na informačních a komunikačních technologiích, které podporují a zlepšují prevenci, diagnostiku, léčbu, sledování a řízení zdraví a životního stylu.

eLibrary

Computer aided technologie (CAD, CAM, CIM, CASE...)

Podstata: počítačová podpora (automatizace) některých procesů (návrh, výroba ap.)

CAD (computer-aided design) počítačem podporované projektování. Jde o velkou oblast IT, která zastřešuje širokou činnost navrhování.

Ve strojírenství CAM (computer-aided manufacturing) CAE (computer-aided engineering)

Stavebnictví a architektura – AEC (Architecture-Engineering-Construction), BIM (Buliding Information Model), CAAD (Computer-aided architectural design)

2.2.2. Analýza informačních systemů (IS), role modelování a metodik při tvorbě IS

Informační systém

(IS) je systém pro sběr, udržování, zpracování a poskytování informací.

Informační systémy jsou založené na informačních a komunikačních technologiích. V poslední době se používá zkratka ICT (Information and Communication Technologies).

Vztah mezi daty, znalostmi a informacemi

- *Data* - jakékoli vyjádření (reprezentace) skutečnosti, schopné přenosu, uchování, interpretace či zpracování. Umožňují přenášet a zpracovávat odraz skutečnosti.
- *Znalosti* - výsledek poznávacího procesu, předpoklad uvědomělé činnosti. To, co jednotlivec ví po osvojení dat a po jejich začlenění do souvislostí. Účel znalostí - porozumět realitě.
- *Informace* - je definovaná pomocí dat a znalostí - data, která mají smysl (význam), sdělitelné (komunikovatelné) znalosti.

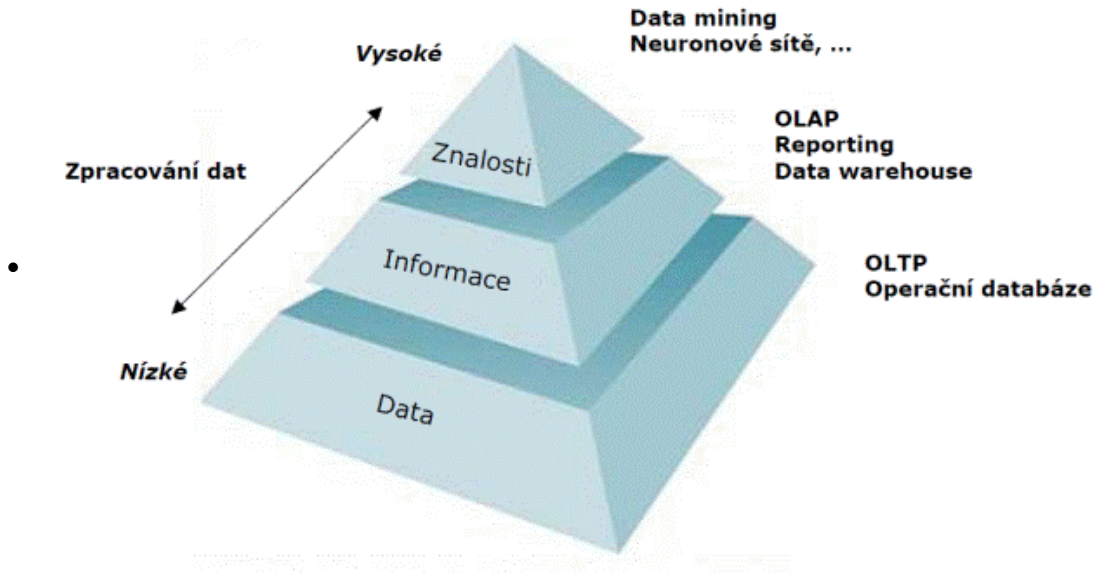
Vztah trochu jinak dle KIV/ZIM:

Znalost x Informace x Data

- **Data** = symboly, objektivní hodnoty, fakta vnímatelné smysly
- **Informace** = data interpretovaná s ohledem na význam a důležitost
- **Znalost** = interakce dat a informací se zkušeností, dovednostmi, hodnotami, myšlením člověka, ostatními fakty, informacemi a znalostmi
  – „informace + X“

Pokud subjekt (příjemce) rozumí významu, který je ukrytý v datech a má o těchto datech jisté znalosti, znamenají pro něj tato data také nějakou informaci. Znalosti jsou potřebné pro konverzi dat do informace.

Informace představuje vypovídací schopnost dat, vzniká zpracováním dat a je cílem tohoto zpracování.



Role modelování a metodiky při tvorbě IS

Složitost systému se promítá do složitosti jeho návrhu a realizace. (tedy i do modelování)

Fenomém složitosti
Se složitostí uspořádání světa roste i **složitost používaných informačních systémů**. V důsledku rychlého vývoje technologií a stále rostoucí složitosti IS tradiční postupy návrhu selhávají. Složitost systému se promítá do složitosti jeho návrhu a realizace, ale i vlastního řízení projektů.

Složitost versus jednoduchost: pokud má být IS pro uživatele srozumitelný, jednoduše, intuitivně použitelný, je vnitřně složitý a naopak. (Připravit si stručnou přednášku je velmi pracné).

Role modelování a metodiky je taková, že odstraňuje tyto problémy (asi)

- Systém dělá něco jiného než by měl
- Systém řeší problémy lokálně. Důsledkem je, že jedna a tatáž věc je řešena na různých místech několikrát, po každé jinak.
- Opravy a změny systému jsou velmi obtížné a drahé. (Jestliže opravujeme chybu na základě lokálních znalostí, tak vlastně opravujeme výskyt chyby, ale ne její příčinu.)
- Systém nelze realizovat několika skupinami současně, paralelně. Bez plánu vznikají komunikační problémy.

Metodiky

Chceme-li se vyhnout potížím s lokálním rozhodováním, musíme postupovat **metodicky (ne chaoticky)**, strukturálně, dle „dobrých“ osvědčených vzorů. Návod jak postupovat nám dávají **ověřené postupy** – vypracované metodiky.

Metodiky odrážejí určité náhledy na „realitu“, říkají **„jaké“** kroky učinit v jakém pořadí a **„jak“** je provádět. Dobré metodiky nám říkají i **„proč“** to tak má být.

Metodiky jsou konzervovanou zkušeností několika generací programátorů a projektantů. Zobecnění principů, zásad, které se osvědčily, viz historie UML.

Modely

Místo abychom se snažili popsat systém jako celek, vytváříme na něj **jednotlivé pohledy** – jeho jednotlivé, dílčí modely. Díváme se na systém postupně z jednotlivých „míst pozorování“, z jednotlivých perspektiv.

Díváme-li se na systém z jednoho místa, opomíjíme vlastnosti z tohoto místa „neviditelné“, nepodstatné a tím si práci zjednodušíme tak, že je mentálně zvládnutelná. Jednotlivé pohledy jsou jednodušší, zvládnutelné. Opomíjené vlastnosti se neztratí, jsou hlavními vlastnostmi v jiných pohledech - modelech.

Pohledy musíme volit tak, že postupně popíšeme všechny relevantní vlastnosti systému. Postupně popíšeme vše, co potřebujeme k dosažení stanoveného cíle.

Z jednotlivých pohledů lze zpětně zrekonstruovat celý systém (počítačová tomografie).

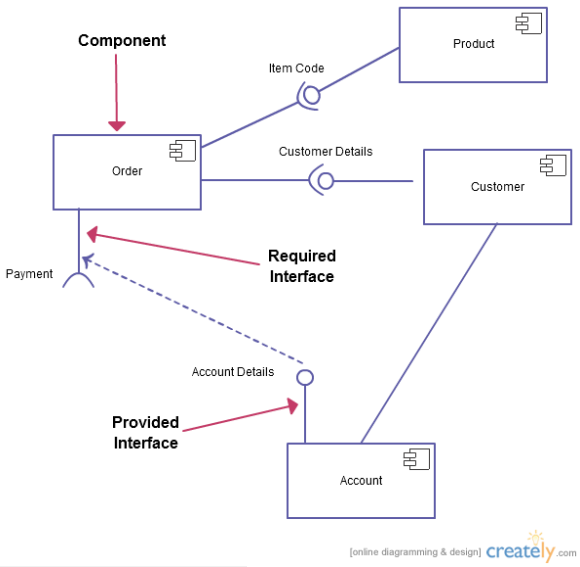
Pro tvorbu různých pohledů jsou obvykle **využity diagramy** – grafické objekty, jejichž kombinací lze tyto pohledy vytvářet.

Diagram je **graficky znázorněný model**. Diagram popisuje jistou část modelu pomocí grafických symbolů.

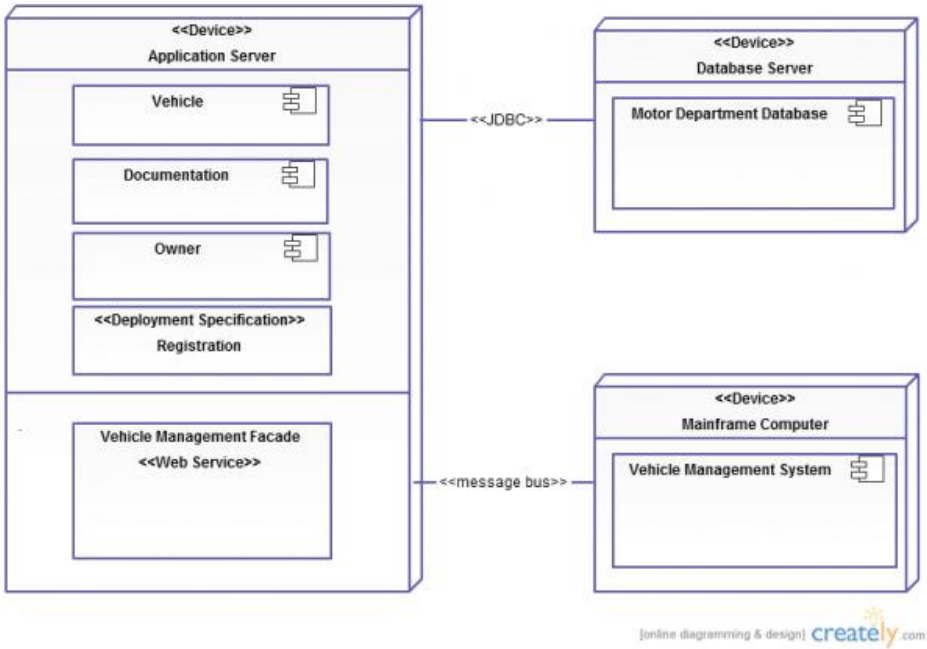
Tento přístup lze přirovnat k modelu stavby, který je tvořen syntézou dílčích stavebních plánů odpovídajících specifickým pohledům na stavbu – plán hrubé stavby, plánu rozvodů elektřiny, plánu rozvodů vody, … V každém z těchto plánů jsou zobrazeny pouze elementy modelu podstatné pro daný pohled, od ostatních elementů modelu je abstrahováno. **Pohledy nejsou nezávislé**, dohromady tvoří konzistentní pohled na systém, tedy konzistentní model.

Pro tvorbu diagramů systému, jejichž syntézou bude model, definuje např. **UML devět typů diagramů.**

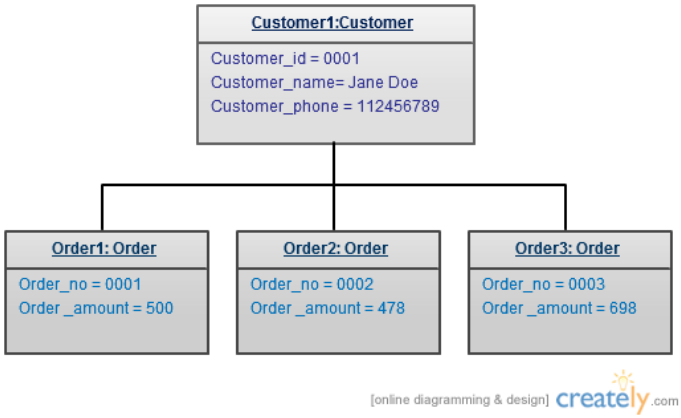
1. Class Diagram
2. Component Diagram



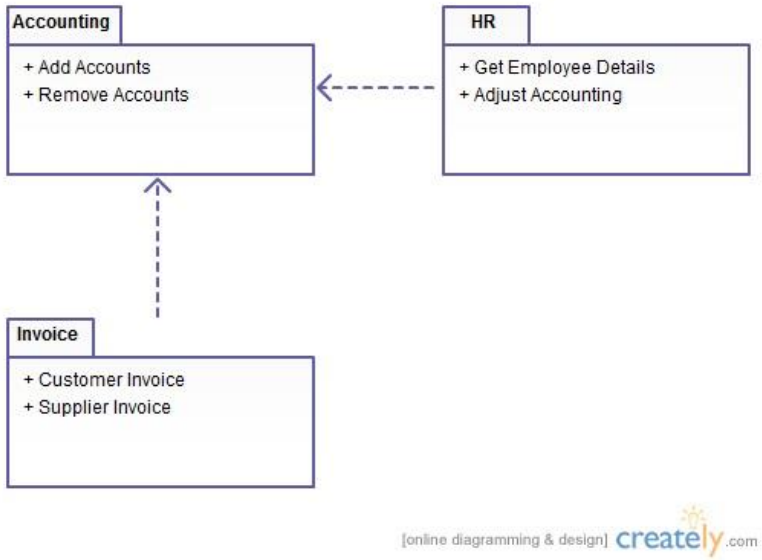
3. Deployment Diagram



4. Object Diagram (Instance diagram)

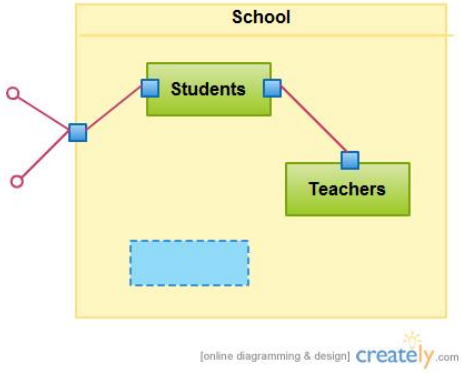


5. Package Diagram

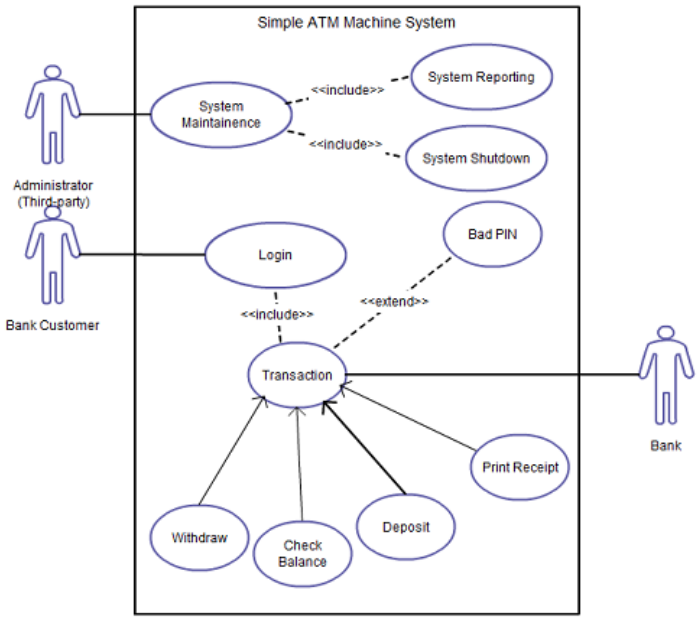


6. Profile Diagram (? Asi ignorovat, od UML 2)

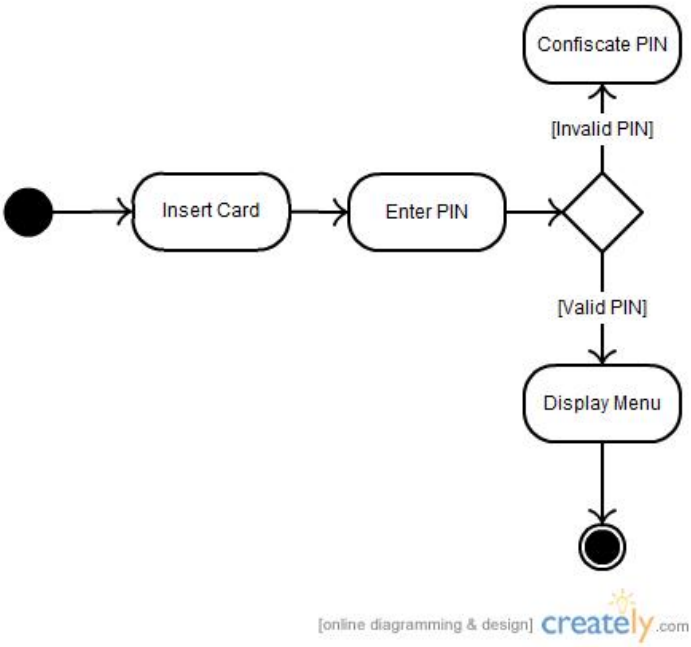
7. Composite Structure Diagram (popis vnitřní struktury třídy)



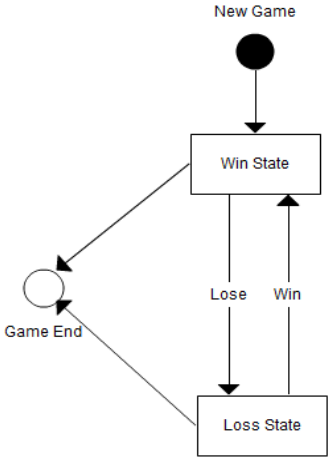
8. Use Case Diagram



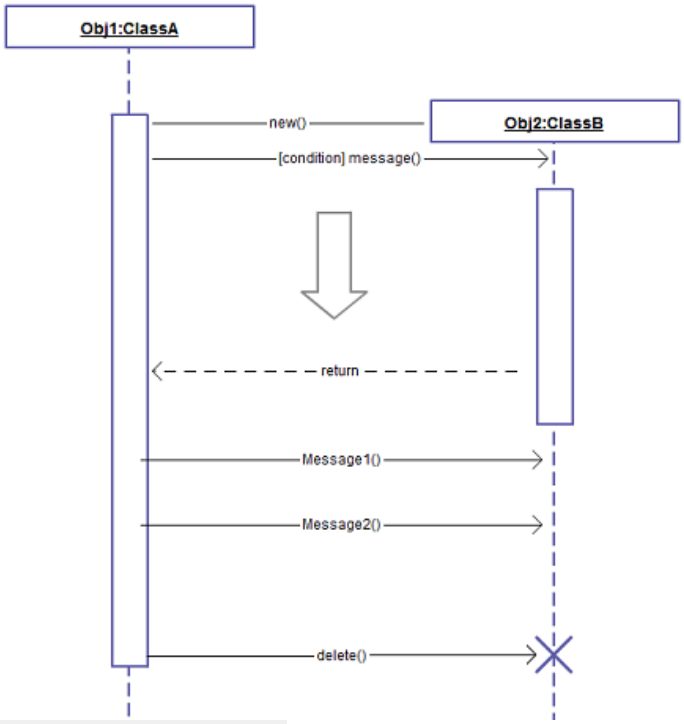
9. Activity Diagram



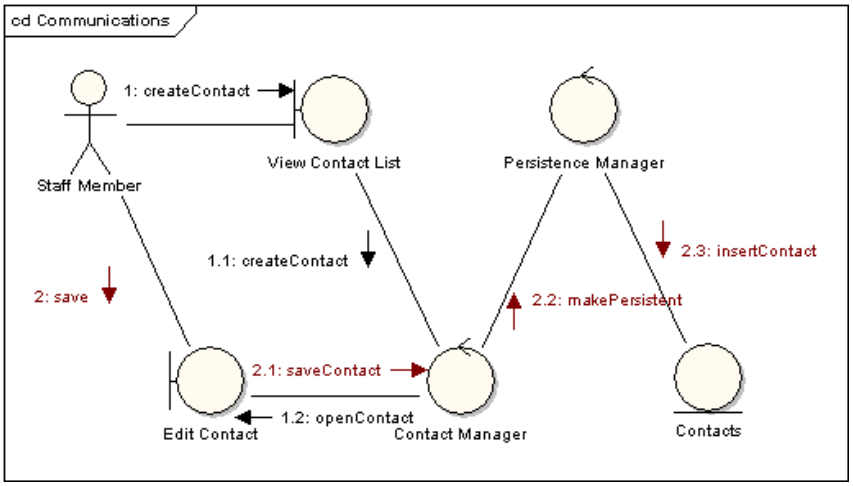
10. State Machine Diagram



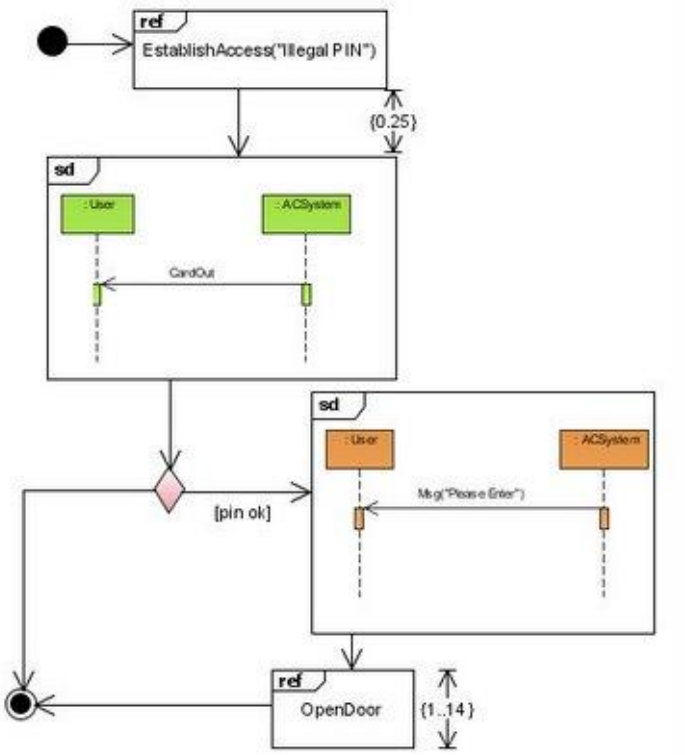
11. Sequence Diagram



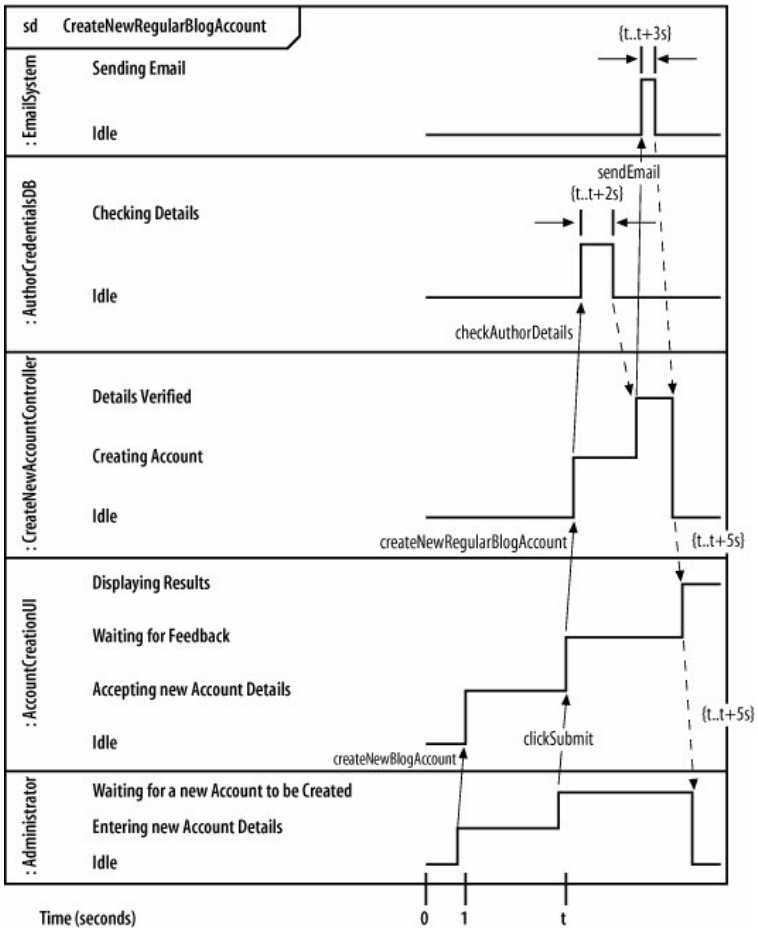
12. Communication Diagram



13. Interaction Overview Diagram



14. Timing Diagram (asi ignore)



Životní cyklus IS

Zkušenosti ukazují, že je účelné tvorbu (analýzu, návrh a realizaci) IS organizovat do posloupnosti etap, mluvíme o tzv. životním cyklu IS.

Životní cyklus IS není statická sekvence činností. Popisuje dynamiku vývoje, měnící se vnější podmínky (změny legislativy), postup od obecného ke speciálnímu. Životní cyklus IS začíná prvotní představou o systému a končí vyřazením systému z provozu. Samozřejmě není znám přesný a úplný (matematický) model životního cyklu. Nepřesné modely ŽC jsou však nepostradatelné pro řízení projektů.

Základní fáze životního cyklu IS:

- Stanovení globálních cílů, specifikace požadavků, specifikace vlastností, které by měl budoucí systém realizovat (implementovat).
- Analýza systému, tvorba analytického, logického modelu systému, model požadovaných vlastností systému, stanovení obsahu IS.
- Návrh (design), specifikace způsobu, jak požadované vlastnosti implementovat, jaké zvolit technické řešení
- Implementace systému, převedení navrženého systému do funkční aplikace
- Testování vytvořeného kódu
- Nasazení systému, provoz, údržba

Sekvenční mode - model vodopád

Pro řízení a vývoj IS by bylo ideální, kdyby po úplném ukončení jedné fáze, etapy ŽC následovala další a k předchozí etapě by nebylo nutné se již vracet. Existuje systematický postup, jak dojít od zadání k cíly. Používá se např. ve stavebnictví (ve dne šije, v noci párá).

Model vodopád je složen z posloupnosti vymezených činností.

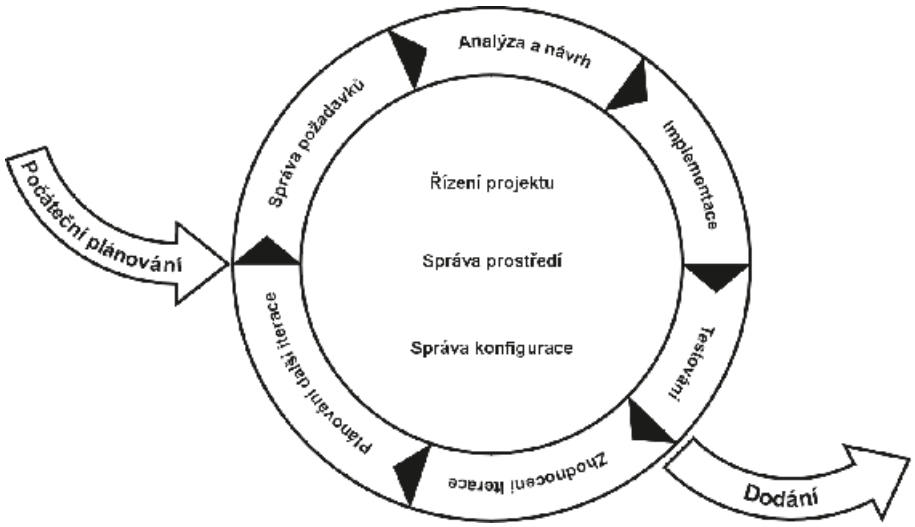
Realita je však díky své složitosti jiná. Jednotlivé fáze, etapy ŽC se překrývají, viz metodika RUP (Rational Unified Process – komerční verze metodiky Unified Process od IBM, původně Rational Corporation).

Tak jak postupujeme v ŽC, postupně upřesňujeme výchozí poznatky a musíme se vracet k předchozím etapám. V průběhu práce se také mění výchozí požadavky uživatelů a vnější (legislativní, technologické prostředí).
Jednotlivé fáze ŽC jako sběr požadavků, analýza, návrh, implementace i testování se v čase překrývají.

Iterativní model

Je možné se vracet do předchozích fází vývoje za účelem zpřesnění. Používané varianty - prototypový model a spirálový model.

Iterace – opakování (dokola)



Prototypový model

Tento model se začal prosazovat v 80. letech. Jeho hlavním cílem je zlepšení vývoje IS využitím prototypů.

Prototyp můžeme chápat jako zjednodušenou implementaci celého systému nebo jako plnou implementaci části systému.

Prototyp je proveden v co nejkratším čase a v takové funkčnosti, která umožní ověřit, otestovat požadované vlastnosti (např. umožňuje zákazníkovi reagovat na výsledky). Na základě vyhodnocení vlastností prototypu jsou upřesňovány požadavky a modifikován další vývoj.

Spirálový model

Tento model vytvořil B.W.Boehm v roce 1988 a je kombinací prototypového přístupu a analýzy rizik.

Základem celého modelu je neustálé **opakování vývojových kroků** tak, že v každém dalším kroku se na již ověřenou část systému přibalují části na vyšší úrovni.

Postup vývoje v jednotlivých krocích se skládá z následujících částí:

- Specifikace cílů a určení plánu řešení.
- Vyhodnocení alternativ řešení a analýza rizik s daným řešením souvisejících (je daná alternativa vyhovující, únosná).
- Vývoj prototypu dané úrovně a jeho předvedení a vyhodnocení.
- Revize požadavků neboli validace (testování zda prototyp pracuje tak jak má).

Verifikace, neboli ověření zda celkový výstup daného kroku je v souladu se zjištěnými požadavky.

RUP (Rational Unified Process)

Rational Unified Process (RUP) je rozsáhlá, propracovaná objektově orientovaná iterativní metodologie vývoje softwaru. RUP náleží do skupiny. tzv. přístupů řízených případy použití (use-casedriven approach), což znamená, že jako základní element je chápán případ použití definovaný jako posloupnost akcí prováděných systémem či uvnitř systému, která poskytuje určitou hodnotu uživateli systému (v nejobecnějším smyslu).

RUP zavádí čtyři základní fáze vývoje, přičemž každá z nich je organizována do několika iterací. Před započetím nové iterace musí být splněna definovaná kritéria předchozí iterace.

- ve fázi zahájení (inception) musí vývojáři definovat účel a rozsah projektu a jeho obchodní kontext,

- ve fázi projektování (elaboration) je úkolem vývojářů analyzovat potřeby projektu a zákazníka a definovat základy architektury,
- třetí fáze, realizace (construction), je zaměřena na vývoj aplikace a tvorbu zdrojových kódů,
- poslední fázi, ve fázi předání (transition), dochází k předání projektu – buď zákazníkovi nebo do dalšího vývojového cyklu.
-

Evoluční model

Celý systém vzniká postupně, přírůstkově i podle toho jak se během vývoje mění požadavky na systém. Variantami tohoto přístupu je inkrementální programování a agilní metodiky obecně.
Nevýhodou tohoto přístupu je velmi obtížné řízení projektu.
Rozpor mezi rigorózními a agilními přístupy (AP).

S evolučním modelem souvisí i nové principy managementu, řízení softwarových firem:

- Každý má možnost se vyjádřit,
- Schopnosti jsou důležitější než diplomy,
- Pravomoc může přecházet a závisí na vytvářené hodnotě,
- Společenství jsou sebeomezující se,
- Soutěžící myšlenky mají rovné příležitosti k uplatnění.

Metodiky životního cyklu IS

Dnes existují desítky metodik, které **pokrývají** jednotlivé fáze životního cyklu IS. Je zřejmé, že žádná metodika není a ani nemůže být v praxi striktně vynucována a dodržována ve všech svých detailech.

Metodika je určitou kostrou poskytující základní rámec pro vývoj systému, poskytuje návod jak postupovat v jednotlivých fázích životního cyklu informačního systému.

Všechny části metodiky mohou být a pravděpodobně budou uživateli upravovány podle specifických potřeb. Firma volí své firemní metodiky.

Metodiky jsou obsahově naplňovány jednotlivými:
- **metodami** a s nimi souvisejícími
- **technikami** a k tomu potřebnými
- **nástroji**

2.2.3. Metodika návrhu a realizace informačního systému – strukturální a objektová analýza

Existuje v zásadě několik metodik, které popisují analýzu, návrh a realizaci informačních systémů

Jedná se o více méně podrobný popis postupu, který vede návrháře jasně definovanými fázemi krok po kroku při vytváření IS

Metodiky jsou často obecné a každá firma si je může přizpůsobit pro vlastní potřebu a prostředí

Mezi nejpoužívanější patří Strukturální analýza (ta je nejstarší), SSADM (vznik 80. léta ve Velké Británii) a Objektová analýza (konec 80. let, 90. léta)

Metodiky návrhu a realizace informačního systému

Fenomén úhlu pohledu
Chceme-li se vyhnout potížím s lokálním rozhodováním, musíme postupovat metodicky (ne chaoticky), strukturálně, dle „dobrých“ osvědčených vzorů.

Návod jak postupovat nám dávají ověřené postupy – vypracované metodiky.

Metodiky odrážejí určité náhledy na „realitu“, říkají „jaké“ kroky učinit v jakém pořadí a „jak“ je provádět. Dobré metodiky nám říkají i „proč“ to tak má být.

Metodiky jsou konzervovanou zkušeností několika generací programátorů a projektantů. Zobecnění principů, zásad, které se osvědčily, viz historie UML.

Místo abychom se snažili popsat systém jako celek, vytváříme na něj jednotlivé pohledy – jeho jednotlivé, dílčí modely. Díváme se na systém postupně z jednotlivých „míst pozorování“, z jednotlivých perspektiv.

Díváme-li se na systém z jednoho místa, opomíjíme vlastnosti z tohoto místa „neviditelné“, nepodstatné a tím si práci zjednodušíme tak, že je mentálně zvládnutelná. Jednotlivé pohledy jsou jednodušší, zvládnutelné.

Opomíjené vlastnosti se neztratí, jsou hlavními vlastnostmi v jiných pohledech - modelech.

Pohledy musíme volit tak, že postupně popíšeme všechny relevantní vlastnosti systému. Postupně popíšeme vše, co potřebujeme k dosažení stanoveného cíle.

Z jednotlivých pohledů lze zpětně zrekonstruovat celý systém (počítačová tomografie).

Pro tvorbu různých pohledů jsou obvykle využity diagramy – grafické objekty, jejichž kombinací lze tyto pohledy vytvářet.

Diagram je graficky znázorněný model. Diagram popisuje jistou část modelu pomocí grafických symbolů.

Tento přístup lze přirovnat k **modelu stavby**, který je tvořen **syntézou dílčích stavebních plánů** odpovídajících specifickým pohledům na stavbu

– plán hrubé stavby, plánu rozvodů elektřiny, plánu rozvodů vody, … V každém z těchto plánů jsou zobrazeny pouze elementy modelu podstatné pro daný pohled, od ostatních elementů modelu je abstrahováno. Pohledy **nejsou nezávislé**, dohromady tvoří **konzistentní pohled na systém**, tedy konzistentní model.

Pro tvorbu modelu systému, respektive pro **tvorbu pohledů na s**ystém, jejichž syntézou bude model, definuje např. UML **devět typů** diagramů.

Zkušenosti ukazují, že je účelné tvorbu (návrh a realizaci) IS organizovat do posloupnosti etap, mluvíme o tzv. životním cyklu IS.

Životní cyklus IS není statická sekvence činností. Popisuje dynamiku vývoje, měnící se vnější podmínky (změny legislativy), postup od obecného ke speciálnímu. Životní cyklus IS začíná prvotní představou o systému a končí vyřazením systému z provozu. Samozřejmě není znám přesný a úplný (matematický) model životního cyklu. Nepřesné modely ŽC jsou však nepostradatelné pro řízení projektů.

Základní fáze životního cyklu IS:

- Stanovení globálních cílů, specifikace požadavků, specifikace vlastností, které by měl budoucí systém realizovat (implementovat).
- Analýza systému, tvorba analytického, logického modelu systému, model požadovaných vlastností systému.
- Návrh (design), specifikace způsobu, jak požadované vlastnosti implementovat
- Implementace systému, převedení navrženého systému do spustitelného kódu
- Testování vytvořeného kódu
- Nasazení systému, provoz, údržba

V rámci životního cyklu IS řešíme i řadu organizačních a ekonomických otázek:
proč, pro koho, termíny, cena, pracovní tým, situace na trhu, návratnost investice, řízení pracovního týmu, hardwarové prostředky, dokumentace, reakce na změny.

Model vodopád
Pro řízení a vývoj IS by bylo ideální, kdyby po úplném ukončení jedné fáze, etapy ŽC následovala další a k předchozí etapě by nebylo nutné se již vracet.

Model vodopád je složen z posloupnosti vymezených činností.

Realita je však díky své složitosti jiná. Jednotlivé fáze, etapy ŽC se překrývají. Tak jak postupujeme v ŽC, postupně upřesňujeme výchozí poznatky a musíme se vracet k předchozím etapám. V průběhu práce se také mění výchozí požadavky uživatelů a vnější (legislativní, technologické prostředí).

Prototypový model
Tento model se začal prosazovat v 80.letech. Jeho hlavním cílem je urychlení vývoje IS využitím prototypů.

Prototyp můžeme chápat jako zjednodušenou implementaci celého systému nebo jako plnou implementaci části systému.

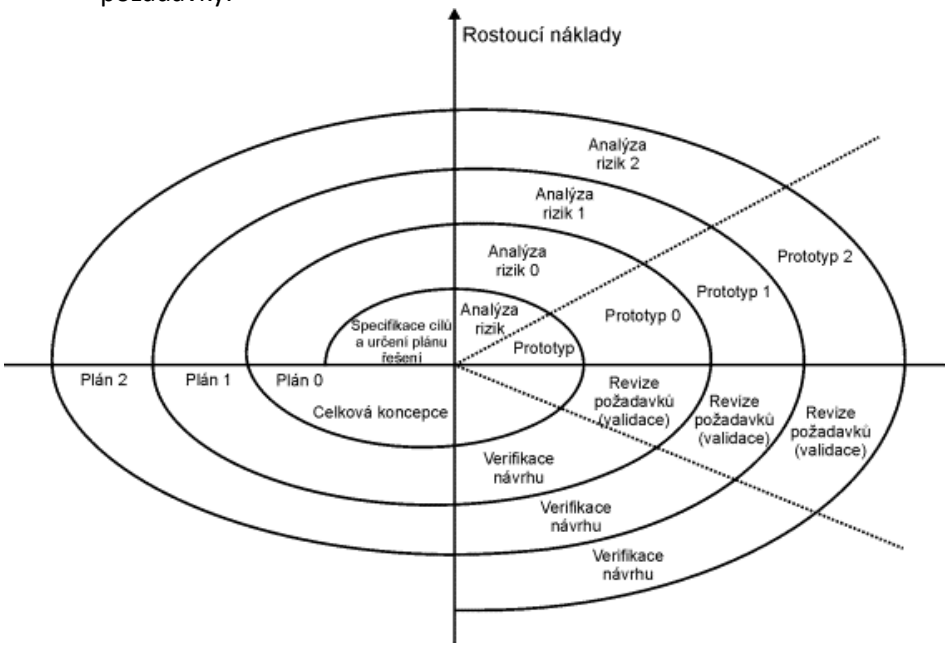
Prototyp je provedena v co nejkratším čase a v takové funkčnosti, která umožní ověřit, otestovat požadované vlastnosti (např. umožňuje zákazníkovi reagovat na výsledky). Na základě vyhodnocení vlastností prototypu jsou upřesňovány požadavky a modifikován další vývoj.

Spirálový model
Tento model vytvořil B.W.Boehm v roce 1988 a je kombinací prototypového přístupu a analýzy rizik.

Základem celého modelu je neustálé opakování vývojových kroků tak, že v každém dalším kroku se na již ověřenou část systému přibalují části na vyšší úrovni.

Postup vývoje v jednotlivých krocích se skládá z následujících částí.

- Specifikace cílů a určení plánu řešení.
- Vyhodnocení alternativ řešení a **analýza rizik s daným řešením souvisejících** (je daná alternativa vyhovující, únosná).
- Vývoj prototypu dané úrovně a jeho předvedení a vyhodnocení.
- Revize požadavků neboli validace (testování zda prototyp pracuje tak jak má).
- Verifikace, neboli ověření zda celkový výstup daného kroku je v souladu se zjištěnými požadavky.



Úhlová dimenze modelu reprezentuje postup prací v čase, radiální dimenze pak rostoucí náklady. Nevhodné prototypy jsou opuštěny. Každý nový průchod cyklem rozvíjí nejlepší prototyp.

Nevýhodou modelu je špatný odhad termínu dokončení projektu a jeho celková cena. Proces vývoje je jakoby otevřen.

Strukturální a objektová analýza

Strukturální analýza

Původní strukturální přístup k analýze IS spočíval **v rozdělení systému na funkční a datovou část** (např. DFD diagramy a ER model).

Přínosem byla funkční hierarchická dekompozice – psaní programu shora dolů a datové konceptuální modelování.

Rostoucí složitost systémů (magická hranice 1000 entit a 10000 funkcí) znemožňuje soudržnost datové a funkční vrstvy. Konstruovali se matice, kde řádky jsou datové entity a sloupce funkce systému. Koexistence se označovala křížkem – možnost dekompozice systém – diagonální matice.

Objektová analýza

Objektový model – diagram tříd

Objektový přístup čelí kromě jiného složitosti systému tím, že třída - objekt jako nositel (funkční) odpovědnosti – dovednosti, plně **odpovídá** za svá data.

Objekt má svou identitu, vlastnosti, chování a **odpovědnost**. Síla odpovědnosti spočívá v tom, že je **nedělitelná** – žádný jiný objekt nemůže odpovědnost sdílet – dělit se o ni, plést se do ní.

Modelování tříd a objektů je **klíčová aktivita** objektově orientovaného vývoje.

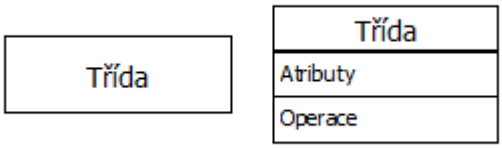
Třída je popisem **množiny objektů** sdílejících stejné vlastnosti - atributy, chování – operace (metody) a vztahy.
Objekt je instancí třídy (chybně se pojem třída a objekt volně zaměňují).

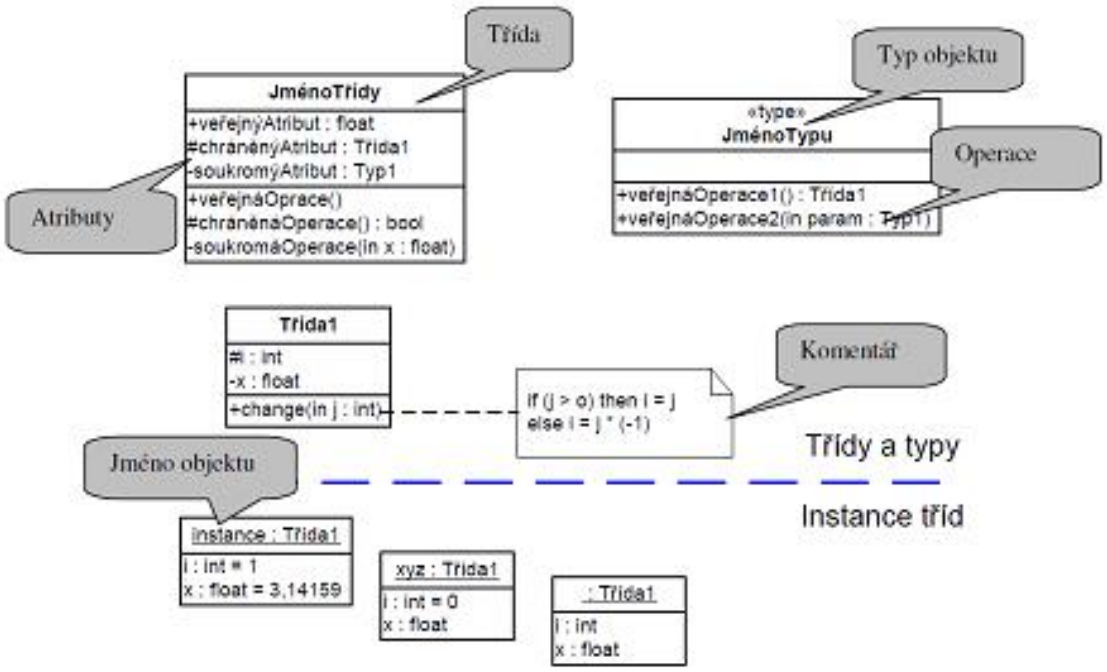
Definice – J. Rumbauh: objekt je diskrétní entita s jasně definovaným rozhraním, které zapouzdřuje stav a chování.

Třídu si můžeme představit jako razítko, objekty jsou pak otisky tohoto razítka, které vidíme na papíře.

Při návrhu třídy neuvažujeme o konkrétním naplnění atributů, pouze určíme jejich název a typ. Teprve při vzniku instance objektu se atributům přiřadí skutečné hodnoty.

Třída je jednoznačně určena svým názvem (v příslušném názvovém prostoru – balíčku). Pro třídu je možno definovat vlastnosti - atributy (Attribute) a chování - operace (Operation). Vizuálním elementem:





Hledání tříd, jejich atributů a kompetencí - vyberme z reality objekty, kandidáty pro zobecnění na třídy a prověřme jejich vhodnosti:

- Potenciální třída je smysluplná, pokud je nezbytná pro funkci systému.
- Potenciální třída je dostatečně stabilní a invariantní vůči vnějším změnám např. technologie, legislativy apod.

Hledání tříd na základě analýzy podstatných jmen a sloves.

Analyzujeme jazyk problémové domény, např. text sebraných požadavků. Podstatná jména a jejich spojení mohou označovat třídy nebo atributy.
Slovesa mohou označovat odpovědnosti, chování tříd.

Pozor na skryté, utajené třídy, které nejsou v textu uvedeny.

2.2.4. Datové modelování, perzistence objektů, konceptuální a fyzický datový model

Datové modelování

- jedna ze základních funkcí IS je ukládání a následné zpracování informací ve formě dat, proto se provádí při návrhu IS také datové modelování
- jeho úkolem je zvolit jaké objekty, jako nositele informace, potřebujeme ukládat do trvalé paměti a jaké jejich vlastnosti a vztahy mezi nimi chceme uchovávat
- při datovém modelování obvykle vytváříme konceptuální a fyzický datový model a také určujeme způsob perzistence objektů

- Cílem datového modelování je navrhnout „kvalitní“ datovou strukturu a databázový systém pro konkrétní IS.

Perzistence objektů

- zabývá se kam a jakým způsobem budou objekty trvale uloženy
- objekty se obvykle ukládají do relační databáze ve formě datových záznamů v tabulkách
- pro programový přístup do databáze se často používá standardizované softwarové API pro databáze jako ODBC nebo JDBC
- pro usnadnění programátorské práce při vytváření perzistentní vrstvy můžeme využít objektově-relační mapování, které nám zajistí automatickou transformaci ukládaných objektů do záznamů relační databáze - např. framework Hibernate pro Java aplikace

Konceptuální datový model

"Konceptuální modelování má své kořeny v počátcích datového modelování a úzce souvisí i například s teorií relačních dat. Jedná se o modelování reality prostřednictvím základních pojmů (konceptů) a jejich vzájemných souvislostí. Konceptuální modelování je dnes široce rozvinutý samostatný obor s propracovanými nástroji a metodami, jejichž principy se odráží i v pravidlech modelování tříd objektů pomocí UML."

- vyjadřuje jaké objekty a jejich atributy budeme ukládat a vztahy mezi nimi
- pro grafické vyjádření se používají ERA modely (diagramy)

Při datovém modelování vytváříme nejprve logický - konceptuální datový model. Konceptuální datový model představuje určité zobecnění oproti implementaci datové struktury v konkrétní relační databázi.

> **Konceptuální model – fáze návrhu, volby technologie, nezávislý na konkrétní databázi**

Fyzický datový model

- odvozuje se z konceptuálního modelu a vyjadřuje navíc jak přesně budou data v konkrétní databázi uložena
- také se popisuje ERA modely, které obsahují i přesné databázové typy atributů tabulek

Zvolíme-li konkrétní databázi (např. Oracle) mluvíme o fyzickém datovém modelu, na který je konceptuální model převeden.

Fyzický model – fáze implementace pro konkrétní databázi, konkrétní implementace. Z fyzického modelu můžeme generovat SQL skripty, případně se napojit přímo na databázi. Na fyzické úrovni můžeme psát uložené procedury a triggery.
Dva přístupy:

- **Při tvorbě konceptuálního datového modelu vycházíme z diagramu analytických, resp. návrhových tříd s jasnou představou o požadavcích na perzistenci.Postup od objektové analýzy, přes návrhové třídy k implementaci.**

Primárně pracujeme s konceptuálním modelem, objektová nástavba je sekundární.

ERA modely

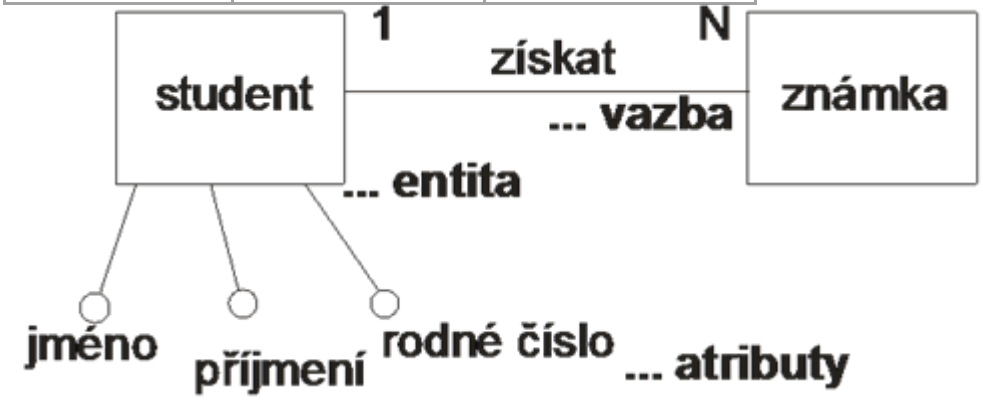
viz DB1 - vypsáno z DB1

Entita – model z reálného světa

Relation – podchycuje vztahy mezi entitami

ERA modely = modelová analýza

Datový model	Datová struktura	Souborový přístup
Entitní množina	Tabulka	Soubor
Entita	Řádka	Záznam
Atribut	Název sloupce	Položka



Vazby

1:N - vyjadřuje, že jedné entitě E1 může příslušet více entit z entitní množiny E2

jedné entitě z E2 přísluší jen jedna entita z E1

nejlépe 1:N(0)

př.: student – známka

1:1 - na vazbě se podílí jej jedna entita z E1 a jedna z E2

vždy se ptát, proč je to rozdělené, proč to není jedna entita

vazba je zajímavá, pokud alespoň jeden konec volný (nepovinný)

př.: student – známka (student nemusí mít známku)

N:N - jedné entitě z E1 přísluší více entit z E2

jedné entitě z E2 přísluší více entit z E1

př.: student – rozvrhová akce

ER modely

- primární – minimální množina atributů, která jednoznačně určuje entitu
- na vazbu se díváme jako na entitu – lze k ní přidat atributy (čtenář – výpůjčka – exemplář)
- vazba 1:N se při realizaci vytvoří tak, že do "podřízené" tabulky přenesu klíč (cizí klíč) z "nadřazené" tabulky
- vazba M:N nelze realizovat, nelze vyřešit pomocí cizích klíčů = musí se provést rozklad vazby
- mezi dvěma entitními množinami může existovat více vazeb
- vazba nemusí být binární, ale může být n-ární
- vazba může být i unární (sama na sebe)

Některé datové modely rozlišují entitní množiny regulární a slabé

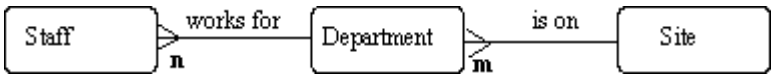
Slabá entitní množina je taková, u níž nelze určit, či nemůžeme zjistit nadřazenou množinu

FAN

Problém: 2 vazby 1:N se větví z jedné entity

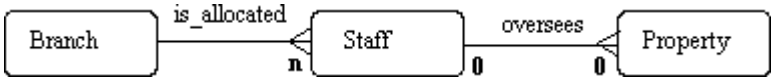
datové položky ve dvou složkách nejsou v přímém vztahu, ale mají vazbu založenou na datových položkách ve třetí složce

Řešení:

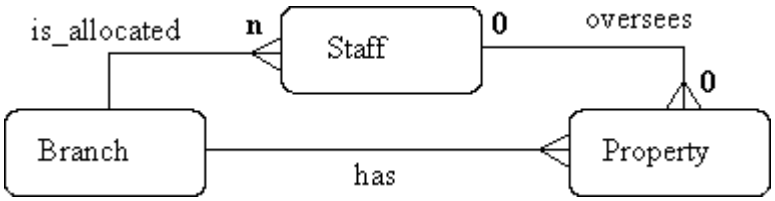


CHASM trap

Problém: Existuje vztah mezi entitami, ale chybí vazba.



Řešení:



Objektově relační mapování - Perzistence

Objektově relační mapování – O/R slouží k tomu, aby bylo možné snadno používat relační databáze v prostředí objektově orientovaných programovacích jazyků. Vzhledem k tomu, že objektově orientovaný návrh dat není jednoznačně převoditelný na relační databáze a opačně, používají se různé formy mapování.

Mapování má za účel načítat data z relační databáze a naplnit jimi příslušné datové položky objektů včetně vazeb mezi objekty, případně naopak datové položky objektů ukládat do databáze.
Snahou ORM je co nejlepší využití obou zmíněných technologií

- objekty by měly reprezentovat objekty reálného světa, jak to požadují principy OOP
- na straně databáze bychom zase měli využít všech možností relačních databází – indexy, pohledy, primární klíče, triggery a uložené procedury.

Alternativou k O/R mapování je použití objektové databáze, která je navržená přímo pro ukládání objektů. Použití takové databáze eliminuje potřebu převádět data z objektové podoby do relační. Data jsou uložena přímo ve své objektové reprezentaci. Objektové databáze zatím nejsou příliš rozšířené.
V současnosti je nejpoužívanějším nástrojem pro ORM produkt od firmy JBOSS Hibernate.
Hibernate je O/R mapovací nástroj pro jazyk Java. Jde o volně šiřitelný open source software.
Nabízí prostředí pro mapování objektového modelu na tradiční relační schéma.

Perzistentní třídy musí splňovat jisté vlastnosti, obsahovat

- konstruktor bez parametrů
- getter a setter metody pro perzistentní položky

Konstruktor bez parametrů Hibernate používá při načítání objektu z databáze. Jeho zavoláním v paměti vytvoří prázdný objekt, jehož položky následně nastaví podle hodnot uložených v databázi pomocí setter metod. Getter metody Hibernate používá při čtení položek při ukládání objektu do databáze.

Základní mapování - mapování tříd na tabulky

Perzistentní (entitní, bussines) třídy, resp. jejich instance - objekty odpovídají entitám konceptuálního datového modelu, resp. řádkům tabulek fyzického modelu. Atributy třídy se stanou sloupci tabulek.

Mapovací soubory pro Hibernate se píší v jazyce XML nebo se využívá tzv. anotací Javy (zápis metadat přímo do kódu). Každá třída se mapuje na jednu tabulku. Každá tabulka musí obsahovat primární klíč.
Mapovací soubor obsahuje výčtem elementů <property>, které reprezentují položky, které mají být ukládány a jak mají být ukládány.

Způsob uložení položek lze ovlivnit velkým množstvím atributů. Několik nejpoužívanějších popisuje následující výčet.

- column="column_name" - určuje název sloupce. Implicitně se používá název proměnné.
- type="typename" - určuje typ konverze mezi Java typem a SQL typem.
- not-null="true|false" určuje zda je možné do daného sloupce uložit NULL hodnotu.

Mapování atributů musí odpovídat konverzi datových typů, norma SQL – 92 definuje standardy datových typů (opakem jsou transientní třídy).

Mapování vztahu

Mapováním vztahů zajišťujeme, že se může mezi objektovým modelem a databází současně „přenášet“ síť vzájemně provázaných – asociovaných objektů. Základní výhoda ORM
Při použití Hibernatu jako ORM vrstvy se vazby dělí ještě na jednosměrné a obousměrné.
Rozdíl je v tom, že u jednosměrné vazby má referenci jen jedna entita. Druhá tedy žádnou referenci nemá, kdežto u obousměrné vazby mají reference obě entity.
Vztah se v mapovacím souboru mapuje pomocí jednoho z elementů

- <one-to-one>
- <one-to-many>
- <many-to-one>
- <many-to-many>

podle kardinality daného vztahu. Jediným povinným atributem je name, ostatní atributy jsou nepovinné. Nepovinné atributy jsou podobné jako u elementu <property>.

Mapování dědičnosti

Protože cílový fyzický datový model nepřipouští dědičnost tabulek, viz norma SQL 92, existují tři možnosti:

- mapování 1:1 – každá třída se mapuje do samostatné tabulky, jedna instance objektu je rozložena po více tabulkách, řada nevýhod.
- zahrnutí do nadtřídy – atributy podtříd jsou zahrnuty do nadtřídy, z třídy a jejich podtříd vnikne jedna tabulka. Vhodné v případě malého počtu podtříd.
- rozpuštění do podtříd – všechny atributy nadtřídy jsou přeneseny do tabulek pro všechny podtřídy. Počet tabulek odpovídá počtu podtříd. Vhodné pro velký počet podtříd.
- Viz CASE

CASE (Computer Aided Software Engineering) jsou systémy (programy) určené na **podporu vývoj informačních systémů**.

Používání CASE nástrojů umožňuje analytikům, programátorům, testerům i manažerům (tedy všem, kteří se na vývoji systému podílejí) mít **společný náhled** na to, jak projekt vypadá jako celek, jak jsou jeho jednotlivé části v detailu a jaký je jeho stav v jednotlivých fázích vývoje.

2.2.5. OLAP systémy, jejich význam a oblasti využití, základními principy, dimenze, agregace, extrakce a transformace dat, srovnání transakčních a analytických systémů (OLAP a OLTP technologií).

Základní problémy u běžných transakčních databázových systémů:

- nedosažitelnost dat skrytých v transakčních systémech
- dlouhá odezva při plnění komplikovaných dotazů
- složitá, uživatelsky nepříjemná rozhraní k databázovému softwaru
- cena v administrativě a složitost v podpoře vzdálených uživatelů
- soutěžení o počítačové zdroje mezi transakčními systémy a systémy podporujícími rozhodování

Cesta k řešení těchto problémů = datové sklady, tzv. Data Warehouse – DW

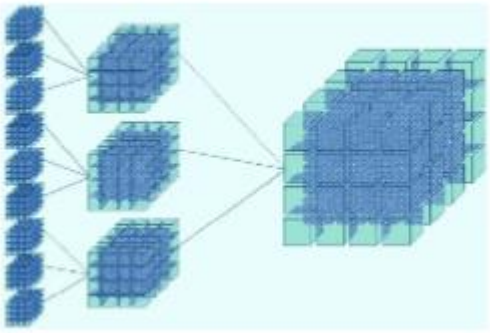
Datawarehouse

- samostatný informační systém postaven na již pořízených datech, určen především k jejich analýze
- architektura založená na relačním SŘBD, která se používá pro údržbu historických dat získaných z databází operativních dat, jenž byla sjednocena a zkontrolována před jejich použitím v databázi DW
- data z DW jsou aktualizována v delších časových intervalech, jsou vyjádřena v jednoduchých uživatelských pojmech a jsou sumarizována pro rychlou analýzu
- DW je obrovská databáze obsahující data za dlouhé časové období
- často slučuje data z více rozdílných zdrojů, které mohou obsahovat data různé kvality nebo používat nejednotné formáty a reprezentace
- objemově zabírá stovky GB až několik TB
- nemusí být databází v běžném smyslu, tj. pro přesné provádění transakcí
- je určen pro rychlé vyhledávání
- nejsou kladeny nijak důrazné požadavky na správnost a úplnost dat

Charakteristika

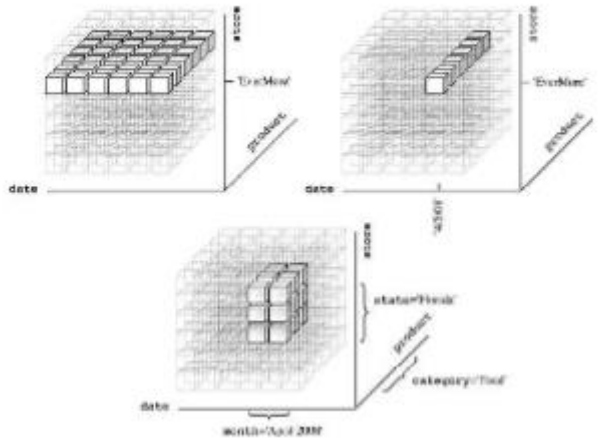
- data jsou uložena na různých místech ve formě relačních tabulek
  - uživatelé mohou tabulky jen číst
  - zapisovat může aktualizační program pravidelně udržující tabulky
- dotazy jsou většinou komplexní
  - podporují tzv. on-line analytické zpracování (OLAP)
    - výrazně se liší od on-line transakčního zpracování (OLTP)
    - operační databáze je přizpůsobena pro podporu OLTP
      1. složité OLAP dotazy by vyústily do nepřijatelné odezvy
    - typické OLAP operace
    - **roll-up** (sumarizace dat napříč dimenzí)

Roll-up



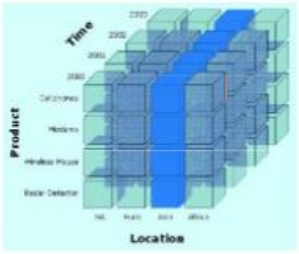
- **drill-down** (zanoření se do nižší úrovně dat pro získání více detailů)
- Drill-up (opak drill down, přechod o level výše pro skrytí detailů a získání lepšího celkového přehledu o datech)

Drill down



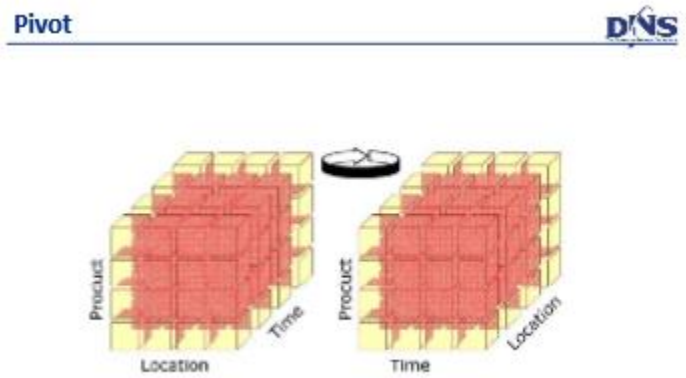
- **slice-and-dice** (selekce a projekce)
- **Slice** - v jednom rozměru krychle zvolíme pouze jednu hodnotu, což vytvoří novou krychli, která je "řezem" té původní
- **Dice** - vybereme konkrétní hodnoty v každé dimenzi krychle, vznikne menší krychle

Slice

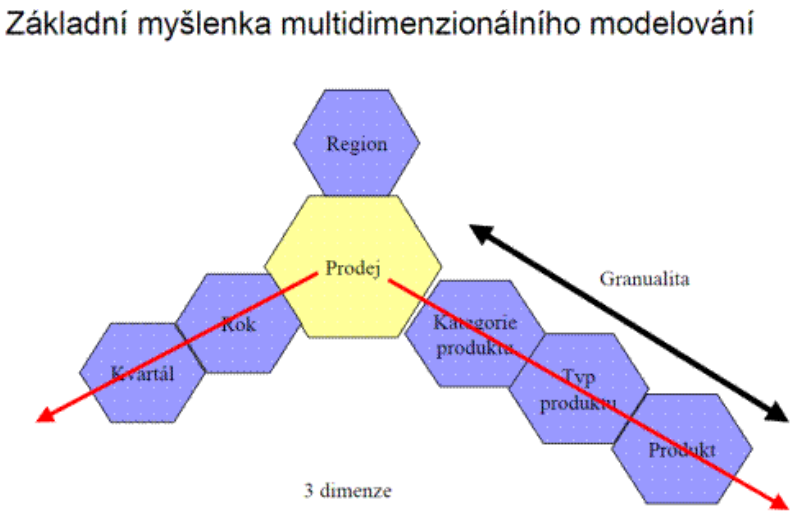


Dice

- **Pivot** (přeorientování vícerozměrného pohledu na data, prostě prohození os)



- na základě dotazu se pospojují potřebná data do vícerozměrné tabulky (nebo více tabulek), do kterých lze klást SQL dotazy
- pro častější dotazy si uchovávají předem připravené vícerozměrné tabulky
- zátěž je většinou způsobena složitými dotazy, jež přistupují k miliónům záznamů a provádějí množství operací
- data bývají modelována vícerozměrně
  - v obchodním data warehouse mohou těmito rozměry být např. čas prodeje, místo prodeje, prodavač, výrobek, …
  - rozměry mohou být i hierarchické např. čas prodeje jako den-měsíc-čtvrtletí-rok, zboží jako výrobek-kategorie-průmysl
  - spojení více tabulek pomocí odkazu na řádky jednotlivých tabulek
  - používají speciální organizaci dat, přístupové a implementační metody, jež obecně nejsou v komerčních databázových systémech určených pro OLTP podporovány



Databázový systém – OLTP (Online Transaction Processing Systems)

- zákaznicky orientovaný
- aktuální data -- lze považovat i za slabinu, při výpadku (chybě), vznikají ztráty pro byznys
- ER schéma
- sofistikované atomické transakce i přes několik systémů(bank, po síti,...)
- velikost DB až několik GB
- jednoduché a efektivní
- příkladem je bankomat

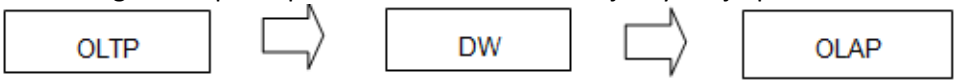
DataWarehouse – OLAP (Online analytical Processing)

- orientovaný na trh, rychlé (oproti OLTP) získání výsledků na analytické dotazy
- historická data, multidimenzionální datový model
- agregovaná data (nenormalizovaná=redundantní)
- schéma hvězdy či vločky
- převážně pouze čtení
- velikost až TB
- použití: byznys reporty o prodeji, marketing, management reporty, rozpočty, finanční předpovědi a reporty

Použití DW

- prezentace dat
- testování hypotéz
- objevování nových informací

Architektura DataWarehouse

- tři úrovně:
  - klient
  - OLAP server (MOLAP/ROLAP server)
  - databázový server DW
- data lze organizovat v tzv. multidimenzionálním datovém modelu
  - odlišný od modelu relačního
  - odpovídá mu specializovaný software, multidimenzionální SŘBD (MDD)
  - model připomíná techniku spreadsheet ve více než dvou rozměrech
  - data jsou implementována pomocí vícerozměrných polí, jejichž dimenze odpovídají dimenzím podnikání organizace
- navržení a vytvoření DW je proces skládající se z následujících bodů:
  - definovat architekturu, umístění a rozčlenění dat a fyzickou organizaci
  - naplánovat kapacitu, vybrat OLAP servery a nástroje
  - spojit servery, klientské nástroje, zdroje přes gatewaye, drivery ODBC, …
  - navrhnout schéma a pohledy, přístupové metody, některé složité dotazy
  - mít skripty pro získávání, čištění, transformaci, ukládání a aktualizaci dat
  - vytvořit koncové uživatelské aplikace
  - spustit data warehouse i aplikace
- vytvoření je složitý proces trvající mnohdy i několik let
- mnoho organizací proto používá Data Mart umožňující rychlejší práci
- 

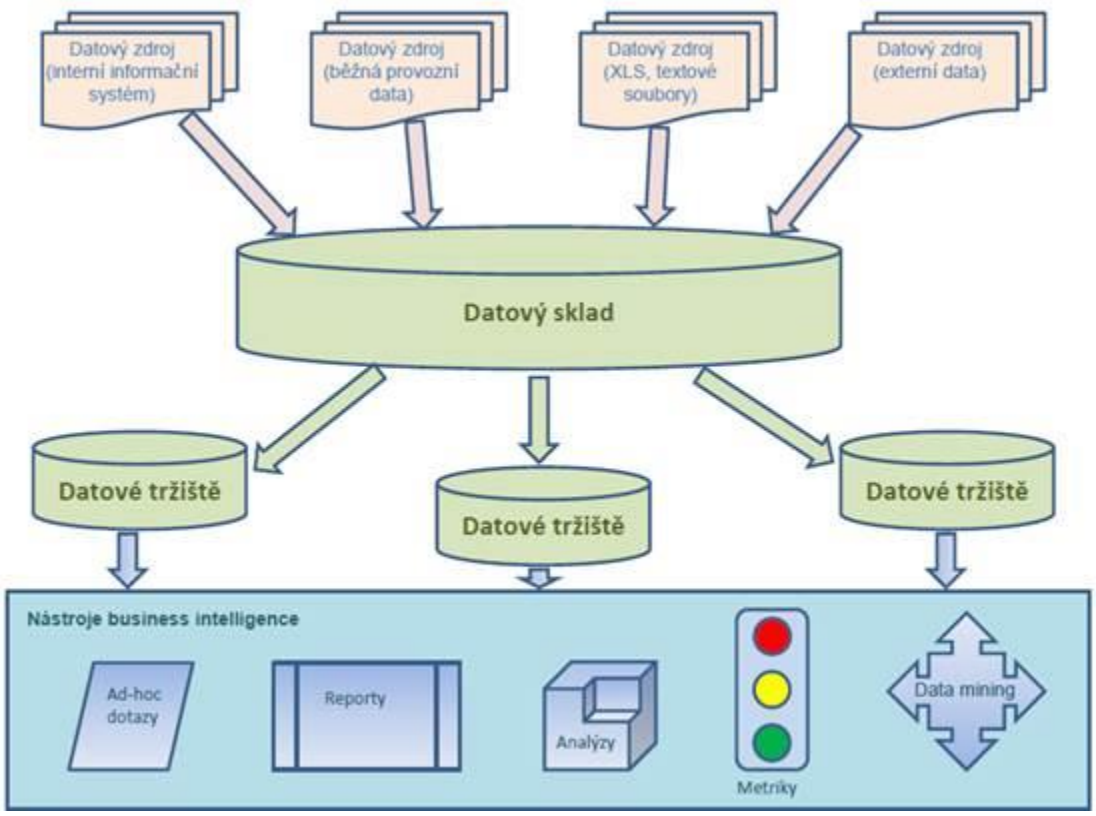
Datová tržiště (Data Mart)

- DW slouží jako základna pro extrakci množin dat, resp. jejich agregaci do dílčích (replikovaných) MDD (Multidimenzionální DB)
  - MDD může pro DW sloužit ve dvou rolích
  - ”front-end” pro DW a poskytovat uživateli služby pro realizaci analytického zpracování (DW/OLAP)
  - “front-end” jednomu (několika) systémům OLTP - alternativa za DW, tj. poskytnout uživateli s OLTP data analytickým způsobem (OLTP/OLAP) – jde vlastně o datové tržiště

Systém OLAP (OnLine Analytical Processing)

- na databázové stroje jsou kladeny specifické požadavky
  - objem zpracovávaných dat
    - transakční systém o velikosti gigabajtů dosáhne použitím jen jedné dimense velikosti desítek či stovek gigabajtů

    - rychlost odezvy analytického systému je důležitá
    - počet uživatelů současně pracujících s databází není zajímavý
        - počet pracovníků vyššího managementu je omezen
        - pro pracovníky nižších stupňů bývají údaje z datových skladů převedeny do menších specializovaných databází – datových tržišť
- s těmito omezeními se vyrovnává dvojím způsobem
    - uzpůsobení stávajících systémů pro práci s vícerozměrovými daty
        - přidáním modulu, který to zajišťuje a prostředků pro jeho ovládání
        - v lepším případě mění způsob uložení dat, v horším "překládá" operace s vícedimenzionálními daty na operace s daty relačními
    - vytvoření speciálního systému správy dat, určeného pouze pro OLAP
        - umožňuje provést maximum optimalizací vzhledem k nárokům kladeným analytickým způsobem práce - převažující způsob



Programy pro vytváření a plnění databáze

- převodní programy
    - načtení data z několika databází, či souborů a udělat z nich novou databázi, agregace se musí naprogramovat
- systémy znázorňující převodu dat graficky a administrátor dat namapuje zdrojová data do struktur vytvářeného datového skladu
    - výsledkem jsou buď programy (scripty) nebo přímo vykonání funkce
- moduly pro plánování jednotlivých akcí

Nástroje pro práci s daty - poslední trendy v architektuře klient/server

- nabízejí variantu tenkého klienta v podobě HTML prohlížeče

Reporting, monitorování, ad-hoc dotazy

- programy umožňující kladení dotazů a formátování odpovědí
  - nejčastěji jde o vizuální dotazovací nástroje
  - makra v tabulkovém procesoru
  - uživatelské rozhraní různě propracované:
    - zadání seskupení výsledku podle různých kriterií
    - formální kontrola dotazů
    - vytváření slovníků a metadat

MOLAP - Multidimenzionální OLAP

- datová krychle (obsahuje fakta)
- hierarchické dimenze (částečné či totální uspořádání)
  - vločkové schéma -- hlavní tabulka faktů je v relaci s dimezionálními tabulkami, přes cizí klíče, dimenzionální tabulky mohou být také v relaci s dalšími subdimenzionálními tabulkami podobně jako hlavní tabulka faktů; vytváří hierarchie dimenzí
  - hvězdové schéma -- je speciální případ vločkového, dimenzionální tabulky již nejsou v relaci s dalšími subdimenzionálními tabulkami; žádné hierarchie, jednodušší

Pozn: dle mého názoru do MOLAP patří jen multidimenzionální krychle (proto MOLAP). Vločka a hvězda jsou ROLAP.

ROLAP – Relační OLAP

- na relační architektuře založený model DW strukturou propojených DB tabulek - Relační OLAP (ROLAP) – pomalejší zpracování než MOLAP
- užívá relační nebo rozšířený relační DBMS, např server METACUBE Informix, pracuje s relačními tabulkami uspořádanými do hvězdy/vločky, adresuje pomocí klíče, data jsou neagregovaná)

Srovnání OLAP a OLTP

Znak	OLTP	OLAP
Charakteristika	Provozní zpracování	Informační zpracování
Orientace	Transakční	Analytická
Uživatel	Běžný uživatel, databázový administrátor	Znalostní pracovník (manažer, analytik)
Funkce	Každodenní operace	Dlouhodobé informační požadavky, podpora rozhodování
Návrh databáze	Entitně-relační základ, aplikačně orientovaný	Hvězda/sněžná vločka, věcná orientace
Data	Současná, zaručeně aktuální	Historická
Sumarizace dat	Základní, vysoká podrobnost dat	Shrnutá, kompaktní
Náhled	Detailní	Shrnutý, multidimenzionální
Jednotky práce	Krátké, jednoduché transakce	Komplexní dotazy
Přístup	Číst, pořizovat a aktualizovat	Pouze číst
Zaměření	Vkládání dat	Získávání informací
Počet dostupných záznamů	Desítky	Miliony
Počet uživatelů	Stovky – tisíce.	Desítky – stovky.
Velikost databáze	100 MB až GB	100 GB až TB
Přednosti	Vysoký výkon, vysoká přístupnost	Vysoká flexibilita, nezávislost koncového uživatele
Míry hodnocení	Propustnost transakcí	Propustnost dotazů a doba odezvy

2.2.6. Vlastnosti a typy CASE nástrojů a jejich význam v analyze a návrhu informačních system

CASE - obecně
CASE – Computer Aided Software Engineering – jsou programy určené k tomu, aby podporovaly vývoj informačních systémů.

Společný pohled - používání CASE nástrojů umožňuje designerům, programátorům, testerům a manažerům (tedy všem, kteří se na vývoji systému podílejí) mít společný náhled na to, jak projekt vypadá jako celek, jak jeho jednotlivé části v detailu a jaký je jeho stav v jednotlivých fázích vývoje.

CASE:

- **pomáhá** zajistit to, že proces vývoje projektu je řízený, řiditelný a kontrolovatelný.
- **čelí složitosti** systému, která by bez jejich pomoci byla těžko zvládnutelná.
- **zajišťuje kvalitu procesů** vývoje softwaru (díky použitým metodikám a odhalování chyb při jejich použití)**.**
- **zajišťuje** značnou **úsporu času** (a tedy nákladů) potřebného k vývoji systému.
- **slouží jako úložiště** projektové dokumentace.

Některé CASE nástroje jsou přímo integrovány do vývojových prostředí.

Odhady hovoří o tom, že použití CASE (přes počáteční zpomalení) nástrojů představuje úspory okolo **50 až 70 procent** v dalších etapách životního cyklu softwaru.

CASE nástroje jsou založeny na dvouvrstvé architektuře.
Základ každého z nich tvoří tzv. „repository“, kam se ukládají veškeré informace o navrhovaném systému (jedná se o databázi, která automaticky udržuje data v konzistentním stavu).
Nad společným repository pracuje druhá modelová vrstva, která zpřístupňuje informace uložené v repository.

Každá z modelových vrstev se opírá o jistou metodiku a reprezentuje jistý pohled na informace uložené ve společném repozitury. Jednotlivé modely jsou díky společnému repozitury na sebe vzájemně převoditelné (např. z diagramu tříd můžeme vygenerovat fyzický datový model).

Vývoj a typy CASE nástrojů

- CASE systémy vznikly v **sedmdesátých letech dvacátého století**, v situaci, kdy začala prudce narůstat složitost IS.

- CASE se na trhu začaly výrazně prosazovat zhruba *v* **polovině osmdesátých let**.
- Tyto systémy vznikly v okamžiku dosažení kritické kvality v metodách, organizaci práce a technologiích, potřebných při vývoji informačních systémů. Od podpory čistě vývojových fází životního cyklu IS (analýzy a konstrukce systému) k podpoře strategických rozhodování na počátku projektu a operativních činností souvisejících s provozem systému a řízením jeho změn a rozvoje.

- To, co dalo podnět ke vzniku CASE nástrojů a určuje také směry dalšího vývoje, je **metodikou tvorby informačních systémů – strukturální a objektové metodiky.**

- Postupem doby a s měnícími se požadavky dnes existuje celá řada CASE nástrojů. Je to díky podporovaným metodikám a samozřejmě také tím, v **jaké fázi vývoje je nástroj používán.**

- **Cílem je využít CASE ve všech fázích životního cyklu IS od s**pecifikace požadavků, analýzu, návrh a kódování po údržbu IS.

Kategorie CASE nástrojů

Podle podporovaných fází životního cyklu systému lze CASE nástroje rozdělit do dvou základních kategorií:

- **Integrované CASE**. Zaměřují se na podporu celého životního cyklu vývoje IS.

- **Specializované CASE**. Tyto nástroje jsou orientované na určité specifické etapy.

Specializované CASE nástroje

Nástroje používané v různých etapách se liší. Většinou pokrývají jen určité činnosti. Podle životního cyklu vývoje lze CASE nástroje rozdělit dle na:

- ***Pre CASE*** – Tyto nástroje jsou určeny pro tvorbu celkové strategie IS.

- ***Upper CASE*** –Nástroje této kategorie podporují plánování, specifikaci požadavků, modelování organizace podniku a celkovou analýzu IS**.**

  Hlavním úkolem je analýza organizace, zachycení všech procesů, definice klíčových toků a dokumentace zjištěných požadavků. Použití je pro specifikaci cílů a počátečních požadavků a řízení projektu.

- ***Middle CASE*** –Tento druh CASE nástrojů je základem všech komerčně dodávaných nástrojů.

- **Prostředky této kategorie slouží pro podrobnou specifikaci požadavků analýzu a návrh systému.** Používají se také pro dokumentaci a vizualizaci systému.

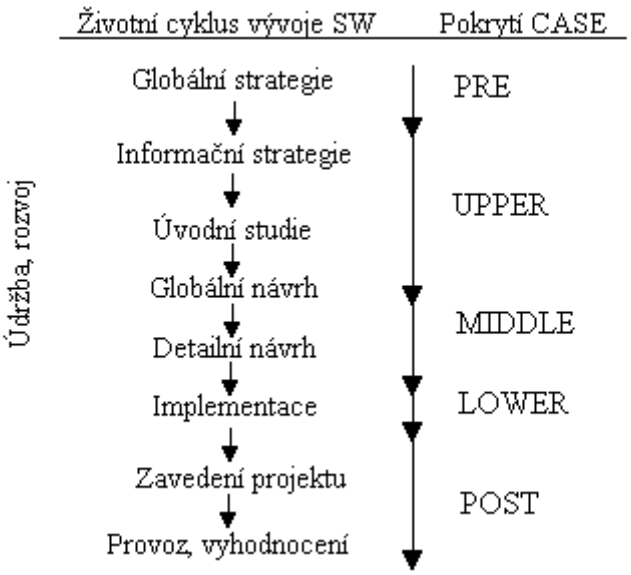
- ***Lower CASE*** – Tyto nástroje slouží především jako podpora kódování, testování a údržby. Jejich součástí jsou i **nástroje forward engineeringu**[1], tedy generátory kódu z modelu (ty mohou generovat kostru nebo podstatnou část výsledného kódu, programátor poté doplňuje jen nutné detaily a algoritmy). Dále pak jde o prostředky pro **reverse engineering**[2], které umožňují získat model z již existující aplikace, prostředky pro plánování a zjišťování kvality SW (sběr informací o testování, vyhodnocení testů, řízení testování), pro správu konfigurace, prostředky pro sledování a vyhodnocování práce systému. Funkce CASE nástrojů této kategorie se často překrývají s funkcemi obecných vývojových prostředí.

- ***Post CASE* –** Tento druh CASE nástrojů podporuje organizační činnosti jako zavedení, údržbu a rozvoj IS.

Působnost takto rozdělených CASE nástrojů se překrývá, protože jimi podporované činnosti se mohou vyskytovat v různých fázích životního cyklu vývoje IS.

Fáze životního cyklu IS a CASE nástroje

Vztah CASE nástrojů a fází životního cyklu IS je zachycen na následujícím obrázku:



Pravdivé a mylné představy o CASE nástrojích

Pravdivé představy:

- Hlavním přínosem těchto nástrojů je vytváření úplných podkladů pro programování aplikací.
- CASE jsou nástroje, které mohou zlepšit produktivitu práce, efektivita práce vždy závisí na osobních kvalitách jednotlivých pracovníků.
    - Mohou generovat části kódu, ale nenahrazují programovací jazyky.
    - Praxe ukázala, že CASE nástroje často selhávají právě díky nedisciplinovanosti uživatelů.
- Automatizací „chaosu“ vznikne automatizovaný „chaos“.
    - Na počátku práce je nutné vykonat velmi mnoho činností, jejichž výsledek není dlouho vidět.
    - Dostanou-li stejný CASE dva systémoví analytici, dospějí k dvěma naprosto odlišným řešením.

Mylné představy:

- CASE nástroje slouží jako náhrada programovacích jazyků.
- Všechny CASE nástroje pracují podobně (poskytují stejné výstupy).
- Užívání CASE nástrojů zlepší práci manažerů organizace využívající výsledný produkt.
- CASE odstraňuje potřebu disciplíny a přísného vývoje aplikací IT.
- Od CASE nástrojů se často očekává jako výstup tvorba aplikačního programového vybavení.
- Produktivita dosažená pomocí CASE je okamžitě zřejmá.
- Užívání CASE zaručí konzistenci výstupů.

3. SP

3.1. OS

3.1.1. Zavadení a struktura operačního systému

Zavedení OS

BIOS

- program přítomný ve vestavěné paměti HW (většinou na základní desce)
- provádí testy a nastavení HW
- vybere zaváděcí jednotku
- načte první sektor (MBR), kde je umístěn program zavaděče a provede skok na adresu jeho programu, čímž mu předá řízení

Zavaděč (bootstrap program)

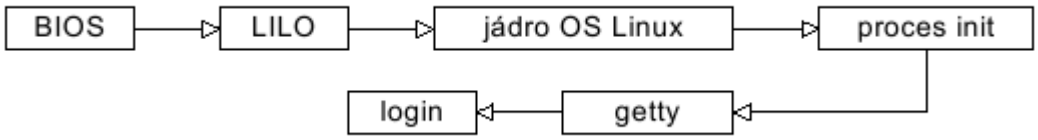
- pro Linux LILO ((LInux LOader) je všeobecně použitelný zavaděč (boot loader) pro Linux) nebo GRUB
- dává možnost zvolit startující OS
- načte jádro operačního systému do paměti a spustí ho

Jádro OS

- detekuje hardware a odpovídajícím způsobem nastaví ovladače zařízení
- připojí kořenový svazek pro čtení a provede kontrolu souborového systému
- spustí na pozadí proces init

Proces init

- proces init se konfiguruje pomocí souboru /etc/inittab
- inicializuje operační systém
- spuštěn po celou dobu běhu operačního systému a ošetřuje některé události (úklid v adresáři /tmp)
- spustí služby - Démony
- nakonec spustí program getty pro terminály a virtuální konzoli a v ní program *login*
- nastavením parametru jádra tzn. runlevel lze upravit chování systému



Úrovně běhu systému – runlevel

- runlevel 0 – zastavení systému - halt
- runlevel 1 – jednouživatelský režim Single user mode
- runlevel 2 – víceuživatelský režim bez podpory sítě Multiuser, without NFS
- runlevel 3 – víceuživatelský režim s podporou sítě Full multiuser mode

- runlevel 4 – není použit unused
- runlevel 5 – víceuživatelský režim s podporou sítě a XFree X11
- runlevel 6 – restart systému reboot

Zavádění systému

- nejprve proběhne úspěšný test zavádění systému – POST
  - kontroluje HW v zařízení
  - série testů ke zjištění, zda HW pracuje správně
  - zjištěné chyby jsou uloženy nebo oznámeny – blikáním LED/série pípnutí
  - Po dokončení je řízení předáno bootovací sekvence volající ovládací SW nebo zavaděč OS
- **Bootování**
  - Najde a zavede (vygeneruje se přerušení 19h) se tzv. Bootovací sector (boot sector)
  - **Boot sector -** oblast 512 bajtů na záznamovém médiu, které je jako první nastavené v paměti BIOSu
    - Bootsektor se nachází na prvním sektoru záznamového média (v případě pevných disků je to válec 0 hlava 0 stopa 0 sektor 1)
  - BIOS se snaží najít na tomto sektoru Master Boot Record (MBR) – hlavní spouštěcí záznam.
    - Ten nahraje do paměti na adresu 0000:7C00 a v případě úspěchu mu předá řízení.
    - V případě chybného MBR se bootovací proces přeruší pomocí softwarového přerušení 18h – vygeneruje chybové hlášení (např. NO ROM BASIC – SYSTEM HALTED)
    - Správnost MBR BIOS zjišťuje pomocí kontrolní hodnoty umístěné na posledních dvou bajtech sektoru - **AA55h** (zápis je uložen ve formátu little endian)
    - MBR – ze dvou částí – **partition loader** a **partition tabulky**
    - MBR uchovává záznamy o rozdělení disku (oddílech) a určuje, ze kterého z nich se má bootovat.
    - Pokud MBR OK – řízení se předá partition loaderu
    - **Partition loader** v Partition tabulce vyhledá oddíl, který je označen jako aktivní a přejde na první sektor tohoto oddílu. MBR sám sebe překopíruje na jiné místo v paměti a na své původní místo zkopíruje tento první sektor a předá mu řízení

Spuštění počítače

Po zapnutí PC jsou všechny procesory v reálném režimu
Náhodně se vybere jedno jádro -> bootstrap processor (BSP)
ostatní CPU jsou nyní application processors AP - pozastavené, dokud je nezapne kernel

- nyní je BSP v tzv. real mode, vypnuté stránkování (BSP simuluje staré 8086 ze let okolo '78)
- v tomto stavu je adresováno pouze 1MB paměti bez ochrany (lze v ní spustit cokoliv)
- u Intel CPU je hack, kdy se nastaví bázová adresa (jako offset) na tzv. reset vector – 0xFFFFFFF0 (konec 4GB paměti - 16B)
- na adrese reset vectoru je jump na adresu, kde je namapovaný BIOS entry point (zajišťuje základní deska)
- tento skok vymaže Intelí hack bázovou adresu
- oblasti v paměti jsou zaplněna správnými daty díky memory map v chipsetu
- nyní BSP spustí BIOS -> Power-On self test (POST) -> error = pípání PC speakeru nebo zombie PC
- po POST bootování systému, umístění volitelné (disketa, DVD ROM, HDD, ...)
- BIOS nyní přečte první sektor umístění (HDD) o velikosti 512B (zero sector) = Master Boot Record (MBR)
- MBR obsahuje:
  1) malý zavaděč na začátku MBR specifický pro OS

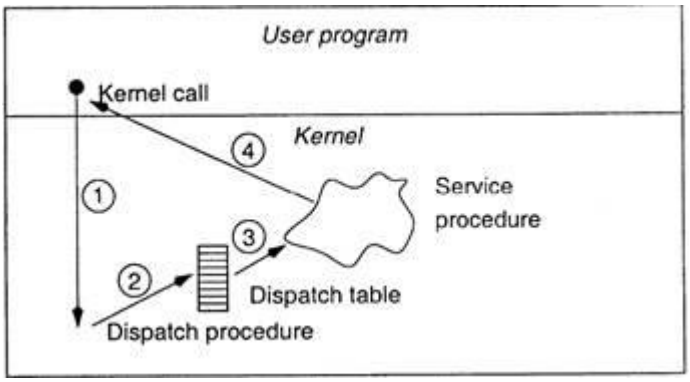
2) tabulka partition
obsah MBR je načten do adresy 0x7c00 a skočí na začátek kódu v MBR (zavaděče)

bootujeme

Viz přednáška PPR d_multithreading.pdf.

Systémové volání (služba jádra)

- volání vstupního bodu jádra OS s přepnutím do privilegovaného režimu
- zjištění čísla požadované služby
- volání obslužné procedury
- návrat s přepnutím do neprivilegovaného režimu



Monolitické jádro

- hlavní program, který spouští obslužnou proceduru
- Uvnitř moduly pro jednotlivé funkce
- množina obslužných procedur pro systémová volání
- podpůrné procedury pro vykonání obslužných procedur

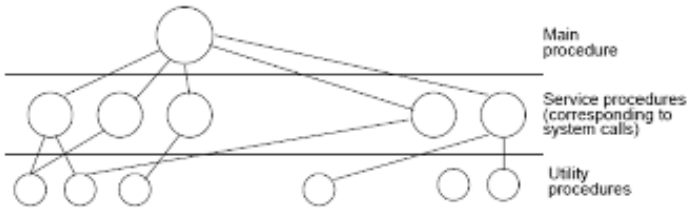
Monolitické jádro

- Jeden spustitelný soubor
- Uvnitř moduly pro jednotlivé funkce
- Jeden program, řízení se předává voláním podprogramů
- Příklady: UNIX, Linux, MS DOS

Typickou součástí jádra je např. souborový systém

Linux je monolitické jádro OS, s podporou zavádění modulů za běhu systému

Monolitické jádro (!)



Main procedure – vstupní bod jádra, na základě čísla služby zavolá servisní proceduru
Service procedure – odpovídá jednotlivým systémovým voláním (zobrazení řetězce, čtení ze souboru, aj.)
Service procedure volá pro splnění svých cílů různé pomocné utility procedures (lze je opakovaně využít v různých voláních)

- mají tendenci extrémně narůstat
- Monolit akumuluje moduly, které by potenciálně mohli být potřebné
- Těžce se ladí

Např. Linux, MS DOS

Vrstvené jádro

Vrstvené jádro

- Výstavba systému od nejnižších vrstev
- Vyšší vrstvy využívají primitiv poskytovaných nižšími vrstvami
- Hierarchie procesů
  - Nejníže vrstvy komunikující s HW
  - Každá vyšší úroveň poskytuje abstraktnější virtuální stroj
  - Může být s HW podporou – pak nelze vrstvy obcházet (obdoba systémového volání)
- Příklady: THE, MULTICS

Vrstvené systémy

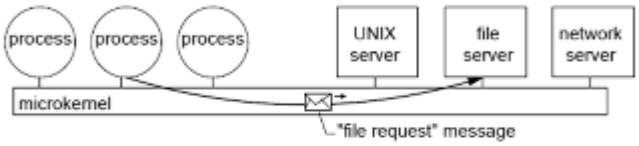
- Hierarchie vrstev poskytujících služby
- Programy vyšší vrstvy využívají služeb nižších vrstev
- Holý počítač je nejnižší vrstva
- Aplikační program je nejvyšší vrstva
- Princip vrstev umožňuje systematickou tvorbu programů a jejich testování
- Princip vrstev je možné použít pro monolitický model i pro systémy

Mikrojádro

Mikrojádro (!)

- Model klient – server
- Většinu činností OS vykonávají samostatné procesy mimo jádro (servery, např. systém souborů)
- Mikrojádro
  - Poskytuje pouze nejdůležitější nízkoúrovňové funkce
    - Nízkoúrovňová správa procesů
    - Adresový prostor, komunikace mezi adresovými prostory
    - Někdy obsluha přerušení, vstupy/výstupy
  - Pouze mikrojádro běží v privilegovaném režimu
    - Méně pádů systému
- Výhody
  - vynucuje modulární strukturu
  - Snadnější tvorba distribuovaných OS (komunikace přes síť)
- Nevýhody
  - Složitější návrh systému
  - Režie
- Příklady: QNX, Hurd, OSF/1, MINIX, Amoeba

Mikrojádro



Mikrojádro – základní služby, běží v privilegovaném režimu
1. proces vyžaduje službu
2. mikrojádro předá požadavek příslušnému serveru
3. server vykoná požadavek

Snadná vyměnitelnost serveru za jiný
Chyba serveru nemusí být fatální pro celý operační systém (není v jádře)
Server může event. běžet na jiném uzlu sítě (distribuov. syst.)

- Vrstva nad holým strojem, která obsahuje minimální množinu abstrakcí, tak aby ostatní funkce OS mohly být implementovány nad ním
- Tyto funkce OS nemusí být vykonávány v privilegovaném režimu
- Jenom mikrojádro musí být vykonáváno v privilegovaném režimu
- Typická množina abstrakcí implementována mikrojádrem:
  - Přerušení
  - Vlákna
  - Správa paměti
  - Meziprocesová komunikace
  - Procesy
- Ostatní funkce – soubory, adresáře, síťové služby – jsou programy vykonávané v uživatelském režimu

Hybridní jádro

- Kombinuje vlastnosti monolitického a mikrojádra
- Část kódu součástí jádra (monolitické)
- Jiná část jako samostatné procesy (mikrojádro)

- Příklady
  - Windows NT (Win 2000, Win XP, Windows Server 2003, Windows Vista,..)
  - Windows CE (Windows Mobile)
  - BeOS

základní rozdělení

- monolitcké systémy - hlavní program, obslužné procedury, podpůrné procedury
- vrstvené systémy - hierarchie vrstev, nejnižší je holý počítač, nejvyšší je aplikační program

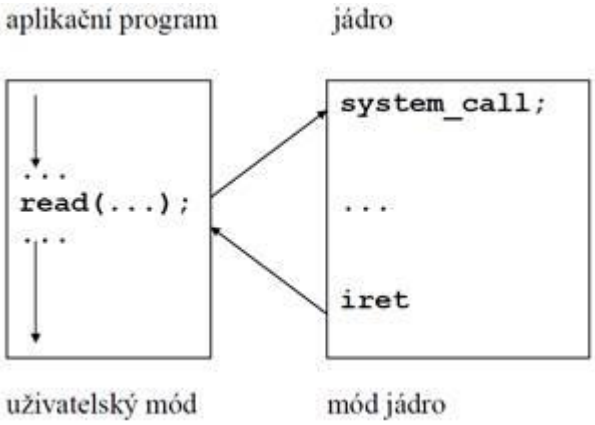
funkční hierarchie - někdy je problém rozdělit do vrstev podle úrovně abstrakce, proto dělení do vrstev podle funkčnosti

- klient-server - obsahuje mikrojádro, které poskytuje pouze základní funkce, většinu práce dělají servery, které jsou oddělené od jádra
- objektově orientovaná struktura - jádro spravuje řadu objektů (zastupují soubory, HW zařízení, ...), mezi objekty jsou tzv. capability = odkaz na objekt + množina práv definujících operace

3.1.2. Jádro operačního systému – monolitické, hybridní a mikrojádro

Jádro operačního systému

Jádro je speciální program zavedený do hlavní paměti při startu systému přímo vykonávaný HW. Jádro má přístup k adresovému prostoru procesů. Jádro je reentrantní, každý proces má svůj zásobník jádra, často přímo v adresovém prostoru procesu – chráněný, spravovaný jádrem. (procesy se navzájem neovlivňujou)

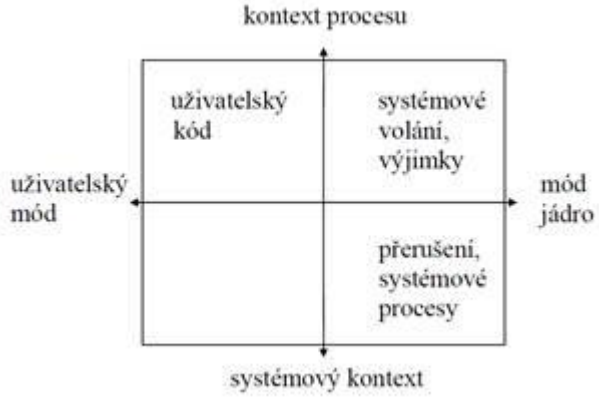


Jádro

- Vykonává služby
- Zpracovává výjimky
- Zpracovává přerušení od periferních zařízení
- Vykonává systémové procesy (správa paměti, přepočítávání priorit procesů)

Jádro pracuje

- V kontextu procesu
- V procesu v systémovém kontextu



Kontextem procesu rozumíme jeho stav, tj. množinu informací nutnou k obnovení činnosti procesu poté, co byl přerušen.

Je-li OS vykonáván jistý proces, říkáme o něm, že běží v kontextu procesu. Rozhodne-li se OS vykonávat jiný proces, provede **přepnutí kontextu**. OS dovoluje přepnout kontext jen za určitých předpokladů. Při připínání kontextu si jádro uchovává dostatek informací k pozdějšímu obnovení činnosti procesu.

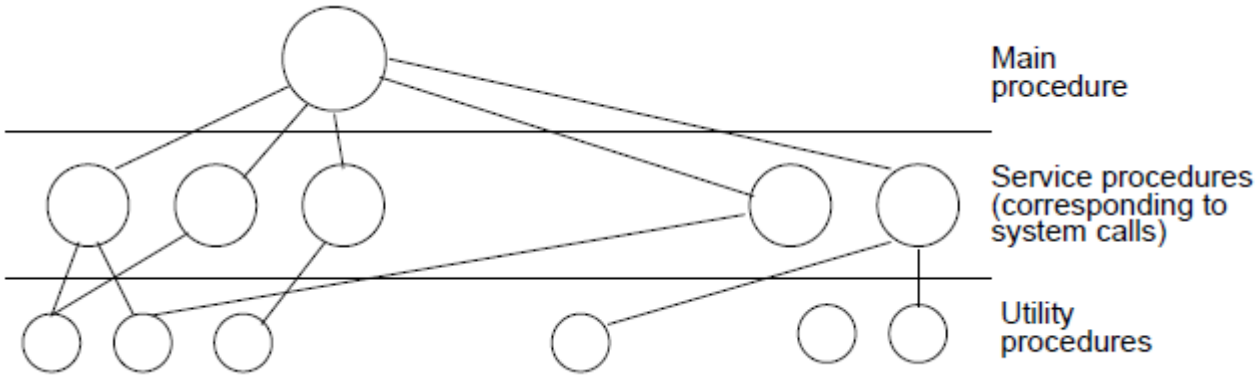
Obdobně při přechodu z uživatelského režimu do režimu jádra (zde jde o změnu režimu, nikoli změnu kontextu) si jádro uchová dostatek informací k návratu do uživatelského režimu.
Přerušení je obsluhováno v kontextu přerušeného procesu (pro obsluhu přerušení není vytvářen zvláštní proces), ačkoli tento proces nemusel přerušení způsobit.

Monolitické

- Jeden spustitelný soubor
- Uvnitř moduly pro jednotlivé funkce (filesys, procesy)
- Jeden program, řízení se předává voláním podprogramů
- Horší na údržbu
- Příklady: UNIX, Linux, MS DOS

Typickou součástí jádra je např. souborový systém

Linux je monolitické jádro OS, s podporou zavádění modulů za běhu systému



Mikrojádro

- Model klient – server
- Většinu činností OS vykonávají samostatné procesy mimo jádro (servery, např. systém souborů)
- Poskytuje pouze nejdůležitější nízkoúrovňové funkce
    - Nízkoúrovňová správa procesů
    - Adresový prostor, komunikace mezi adresovými prostory
    - Někdy obsluha přerušení, vstupy/výstupy
- Pouze mikrojádro běží v privilegovaném režimu
    - Méně pádů systému

Výhody

- vynucuje modulární strukturu
- Snadnější tvorba distribuovaných OS (komunikace přes síť)

Nevýhody

- Složitejší návrh systému
- Režie

Příklady: QNX, **Hurd**, OSF/1, MINIX, Amoeba

Hybridní

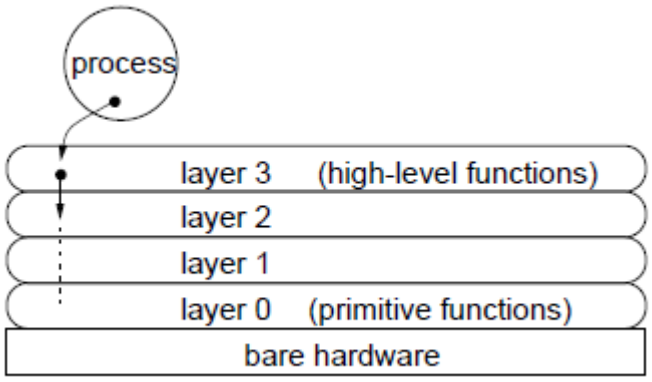
Hybridní jádro je v informatice označení pro jádro operačního systému, které kombinuje vlastnosti monolitického jádra a mikrojádra za účelem získání výhod obou vyhraněných řešení. Hybridní jádro je podobné mikrojádru, ale má některé vlastnosti monolitického jádra, kvůli vyššímu výkonu. Na rozdíl od monolitického jádra nedokáže hybridní jádro za běhu samo zavádět moduly. V jaderném prostoru hybridního jádra běží některé služby (např. implementace síťového protokolu nebo souborový systém), aby se dosáhlo nižší režie v porovnání s mikrojádry, ostatní kód jádra (ovladače zařízení), běží v uživatelském prostoru a označují se jako servery.

- WinNT

http://cs.wikipedia.org/wiki/Hybridn%C3%AD_j%C3%A1dro

Vrstvené

Výstavba systému od nejnižších vrstev Vyšší vrstvy využívají primitiv poskytovaných nižšími vrstvami Hierarchie procesů
–Nejníže vrstvy komunikující s HW
–Každá vyšší úroveň poskytuje abstraktnější virtuální stroj
–Může být s HW podporou – pak nelze vrstvy obcházet (obdoba systémového volání) Příklady: THE, MULTICS

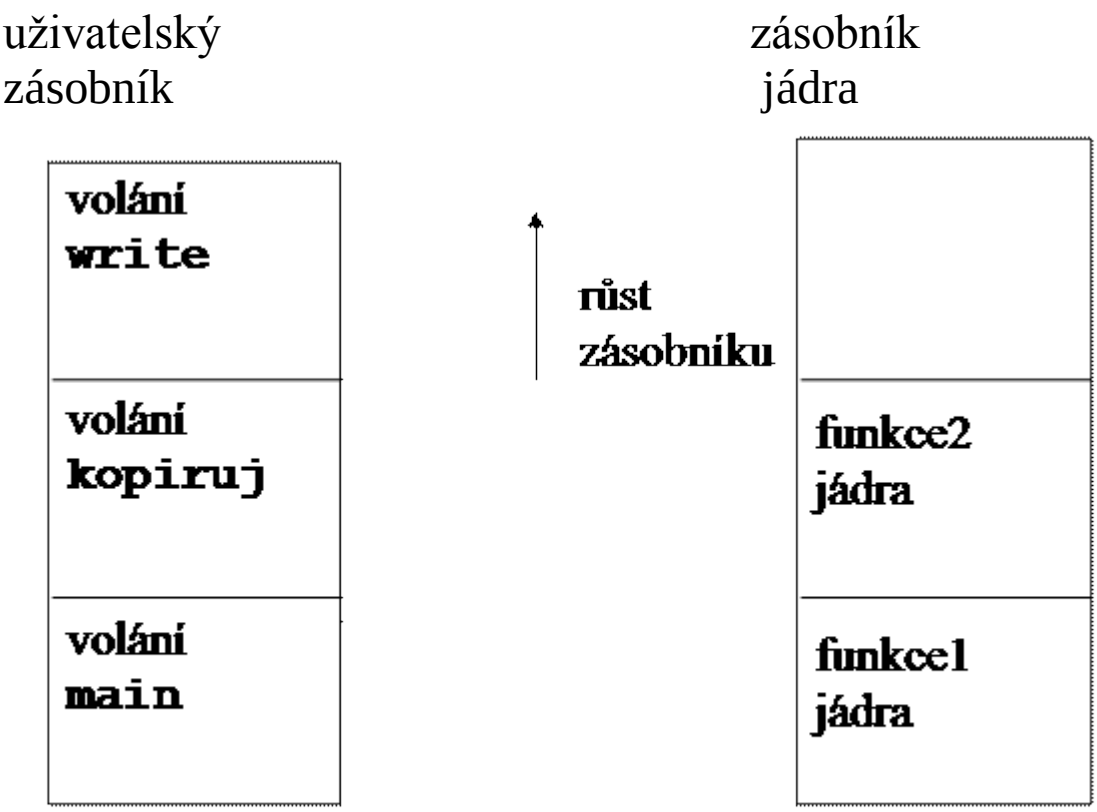


Zásobník jádra:

proces v uživatelském módu má přístup ke svému adresovému prostoru, k systémovému prostoru voláním sytem_call()

jádro má přístup k adresovému prostoru procesů
proces používá dva zásobníky

- uživatelský
- jádra



jádro je reentrantní

každý proces má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem

každý proces má položku v tabulce procesů **`proc`**
záznam a u (*user*) oblast

u oblast – údaje potřebné když je proces vykonávaný

- tabulku deskriptorů souborů otevřených souborů
- okamžitý adresář
- kořenový adresář
- často zásobník jádra procesu

3.1.3. Virtuální adresový proctor

ZOS

Viz přednáška 09_2012 ZOS. Program větší než dostupná fyzická paměť => mechanismus překrývání (overlays). Jako řešení tohoto problému se dnes nejčastěji používá právě virtuální paměť.

Překrývání (overlays)

Program – rozdělen na moduly tak, aby se daly postupně zavádět jednotlivé části do paměti, která je menší než celková paměť potřebná pro běh aplikace. (analogie s koberci)

Start – spuštěna část 0, při skončení zavede část 1…

Časté zavádění některých modulů

- Více překryvných modulů + data v paměti současně
- Moduly zaváděny dle potřeby (nejen 0, 1, 2…)
- Mechanismus odkládání (jako odkládání procesů)

Zavádění modulů zařizuje OS

Rozdělení programů i dat na části – navrhuje programátor (Např. vytváření DLL)

- Vliv rozdělení na výkonnost, komplikované
- Pro každou úlohu nové rozdělení

Virtuální paměť

- Potřebujeme rozsáhlý adresový prostor
- Ve skutečné paměti je pouze část adresového prostoru
  - Jinak by to bylo příliš drahé
- Zbytek může být odložen na disku

Virtuální adresy

- Fyzická paměť slouží jako cache virtuálního adresního prostoru procesů
- Proces – používá virtuální adresy
- Pokud požadovaná část VA prostoru je ve fyzické paměti, tak MMU převede VA => FA, přístup k paměti
- Pokud požadovaná část není ve fyzické paměti, OS si ji musí přečíst z disku I/O operace – přidělení CPU jinému procesu
- Většina systémů virtuální paměti používá stránkování

Mechanismus stránkování (paging)

- Program používá virtuální adresy
- Musíme rychle zjistit, zda je požadovaná adresa v paměti – pokud ano, převedeme VA na FA
- Musí být co nejrychlejší, děje se při každém přístupu do paměti

VAP – stránky (pages) pevné délky. Délka je obvykle mocnina 2, nejčastěji se jedná o 4KB, běžně 512B – 8KB

Fyzická paměť – rámce (page frames) stejné délky. **Rámec** může obsahovat **PRÁVĚ JEDNU stránku**. Na **známém místě v paměti** pak musí být **TABULKA STRÁNEK**. Ta poskytuje mapování virtuálních stránek na rámce.

Tabulka stránek

- Součástí PCB (tabulka procesů) – určuje, kde leží jeho tabulka stránek
- Velikost záznamu v tabulce stránek je 32 bitů kde vyšších 20 bitů určuje číslo stránky a nižších 12 bitů určuje offset
- Číslo rámce má pak 20 bitů (takže max. 2^20 stránek)

Výpočet adresy

Velikost stránky = 4096B.

Je dána VA(p1)=100. Určete FA. Tabulka stránek je:

Číslo stránky	Rámec
0	**1**
1	**2**
2	**--**
3	**0**

Máme-li více procesů, každý má svou vlastní tabulku stránek.

Virtuální adresu rozdělíme na číslo stránky a offset.

Str = VA div 4096 (dělení)

Offset = VA mod 4096 (zbytek po dělení)

Převod pomocí tabulky stránek – převedeme číslo stránky na číslo rámce

- tab_str[0] = 1 (pro stránku 0 je číslo rámce 1)
- tab_str[1] = 2
- tab_str[2] = -- stránka není namapována
- tab_str[3] = 0
- Pro VA = 100 je stránka 0, offset 100 => tedy rámec 1

Z čísla rámce a offsetu sestavíme fyzickou adresu:

- FA = rámec * 4096 + offset
- FA = 1 * 4096 + 100
- FA = 4196 v daném případě
- V reálném systému dělení znamená rozdělení na vyšší a nižší bity adresy (proto mocnina dvou)
- Nižší bity – offset
- Vyšší bity – číslo stránky

Výpadek stránky

- stránka není mapována
- Výpadek stránky způsobí výjimku, zachycena OS pomocí přerušení
- OS iniciuje zavádění stránky a přepne na jiný proces
- Po zavedení stránky OS upraví mapování (tabulku stránek)
- Proces může pokračovat
- Pokud daná stránka procesu není namapována na určitý rámec ve fyzické paměti a chceme k ní přistoupit, dojde k výpadku stránky – vyvolání přerušení operačního systému. Operační systém se postará o to, aby danou stránku zavedl do nějakého rámce ve fyzické paměti, nastavil mapování a poté může přístup proběhnout.

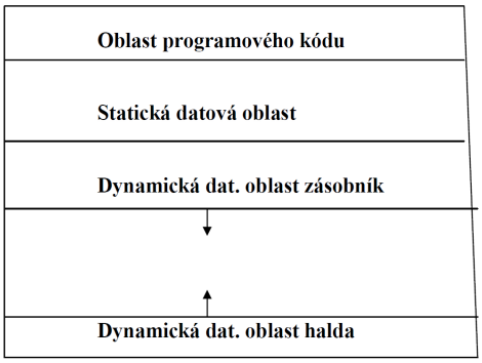
- Vnitřní fragmentace (část přidělené paměti je nevyužita), vnější fragmentace (souvislý paměťový prostor mapován do nesouvislých částí paměti)…
- Tabulka stránek procesu – mapuje číslo stránky na číslo fyzického rámce, obsahuje i další informace jako např. příznaky ochrany.
- Relokace = mapování VA na FA
- Ochrana – v tabulce stránek jsou pouze ty stránky, ke kterým má proces přístup. Při přepnutí na jiný proces přepne MMU na jinou tabulku stránek.
- Problémy
  - velikost tabulky stránek – pomůže víceúrovňová struktura
  - rychlost převodu VA -> FA – pomůže TLB (Transaction Look-aside Buffer)
  - HW cache
  - dosáhneme zpomalení jen 5 až 10%
  - Přepnutí kontextu na jiný proces – problém (vymazání cache, …); než se TLB opět zaplní – pomalý přístup

Invertovaná tabulka stránek – řešení problému velikosti celé tabulky, obsahuje položky pro každý fyzický rámec. Omezený počet – dán velikostí RAM – VA je 64 bitů, 4KB stránky, 256MB RAM – 65536 položek. Forma položky: (id procesu, číslo stránky).

Při výpadku stránek nastupují algoritmy nahrazování stránek: FIFO, …
OPT, LRU, NRU, second chance, aging, clock

Each active process must have a Page Directory assigned to it. However, there is no need to allocate RAM for all Page Tables of a process at once; it is more efficient to allocate RAM for a Page Table only when the process effectively needs it.
 The physical address of the Page Directory in use is stored in a control register named **cr3** .
 The Directory field within the linear address determines the entry in the Page Directory that points to the proper Page Table. The address's Table field, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page. The Offset field determines the relative position within the page frame (see Figure 2-7). Because it is 12 bits long, each page consists of 4096 bytes of data.

Z FJP o rozdělení paměti:



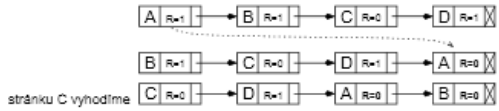
3.1.5. **Algoritmy nahrazování stránek**


```
KIV/ZOS 2003/2004
Přednáška 10

Algoritmus Not-Recently-Used (NRU, NUR)
.......................................

* OS se snaží zjistit, které stránky se používají a nepoužívané vyhazovat
* systémy s VM poskytují HW podporu - stavové bity Referenced (R) a Dirty
  (zde M jako Modified) v tabulce stránek
* bity nastavované HW podle způsobu přístupu ke stránce
 - bit R - nastaven na 1 při čtení nebo zápisu do stránky
 - bit M - nastaven na 1 při zápisu do stránky; označuje, že
           se stránka má při vyhození zapsat na disk
 - po nastavení bitu zůstane na 1 dokud ho SW nenastaví zpět na 0

* algoritmus NRU:
 - na začátku mají všechny stránky nastaveny R=0, M=0
 - bit R je OS nastavován periodicky na 0 (např. při přerušení
   časovače) - tím se rozliší, které stránky byly referencovány
   v poslední době
 - OS rozlišuje 4 kategorie stránek:

   třída 0: R=0, M=0
   třída 1: R=0, M=1  ;; vznikne z třídy 3 po tiku, který nastaví R=0
   třída 2: R=1, M=0
   třída 3: R=1, M=1

 - algoritmus NRU vyhodí stránku z nejnižší neprázdné třídy, výběr mezi
   stránkami ve stejné třídě je náhodný

* algoritmus předpokládá, že je lepší vyhodit modifikovanou stránku
  která nebyla použita 1 tik než nemodifikovanou stránku, která se
  právě používá

* výhody algoritmu NRU:
 - jednoduchost, srozumitelnost
 - efektivně implementovatelný

* nevýhody:
 - výkonnost (jsou i lepší algoritmy)

Pokud by HW neměl bity R a M, můžeme je simulovat následujícím způsobem:

* při startu procesu se všechny jeho stránky označí jako nepřítomné
  v paměti
* při odkazu na stránku nastane výpadek stránky - OS interně nastaví
  R=1 a nastaví mapování v režimu READ ONLY
* při pokusu o zápis do stránky nastane výjimka - OS výjimku zachytí, nastaví
  M=1 a změní režim přístupu do stránky na READ/WRITE.

Algoritmy "Second Chance" a "Clock"
...................................

* algoritmy "Second Chance" a "Clock" vycházejí z algoritmu FIFO

Obchod: V algoritmu FIFO jsme vyhazovali zboží, které bylo zavedeno před
nejdelší dobou (bez ohledu na to, jestli ho někdo chce nebo ne). V algoritmu
"Second Chance" začneme evidovat, jestli zboží někdo v poslední době koupil
(pokud ano, prohlásíme ho za čerstvě zavedené zboží).

* jak modifikovat FIFO, abychom zabránili vyhození často používané stránky?
* algoritmus (Second Chance - vyhledání stránky pro vyhození):
 - podívat se na bit R nejstarší stránky
 - pokud R=0, stránka je nejstarší a zároveň nepoužívaná => vyhodíme
 - pokud R=1, nastavíme R na 0 a přesuneme na konec seznamu stránek
   (jako by byla nově zavedena)
```

Příklad:

Stránky uchováváme v seznamu uspořádaném podle času příchodu. V paměti budeme mít např. stránky A, B, C, D (viz obrázek); algoritmus "Second Chance" bude probíhat v následujících krocích:

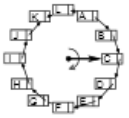
1. krok: Nejstarší je A; má R=1 => nastavíme R na 0 a přesuneme na konec seznamu;
2. krok: Druhá nejstarší je B; má také R=1 => nastavíme R na 0 a opět přesuneme na konec seznamu;
3. krok: Další nejstarší je C, R=0 => vyhodíme jí.



[]

* algoritmus "Second Chance" vyhledává nejstarší stránku, která nebyla referencována "v poslední době"
* pokud byly všechny referencovány, degeneruje na čisté FIFO:
 - postupně všem stránkám nastavíme bit R na 0 a přesuneme je na konec seznamu
 - dostaneme se opět na stránku A, tentokrát má R=0 => vyhodíme jí
 => algoritmus končí nejvýše po $počet_rámců + 1$ krocích

* algoritmus "Clock" - optimalizace datových struktur algoritmu Second Chance:
 - stránky udržovány v kruhovém seznamu
 - ukazatel na nejstarší stránku ("ručička hodin")

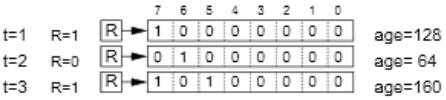


* výpadek stránky -> vyhledáváme stránku k vyhození
 - stránka kam ukazuje "ručička":
  . má-li R=0, stránku vyhodíme a ručičku posuneme o 1 pozici
  . má-li R=1, nastavíme R na 0, ručičku posuneme o 1 pozici; opakujeme dokud nenalezneme stránku s R=0
* od algoritmu Second Chance se liší pouze implementací
* varianty algoritmu Clock používají např. systémy BSD UNIX

Softwarová aproximace LRU
.....

* algoritmus LRU vždy vyhazuje nejdéle nepoužitou stránku

* algoritmus Aging:
 - každá položka v tabulce stránek má pole "stáří" age, N bitů (např. N=8)
 - na počátku age=0
 - při každém přerušení časovače pro každou stránku:
  . posun pole "stáří" o 1 bit vpravo
  . zleva se přidá hodnota bitu R
  . nastavení R na 0



* to odpovídá následujícímu kódu (v Turbo Pascalu):

```
    age := age shr 1;           { posun o 1 bit vpravo }
    age := age or (R shl N-1);  { zleva se přidá hodnota bitu R }
    R := 0;                     { nastavení R na 0 }
```

* při výpadku stránky se vyhodí stránka, jejíž pole age je má nejnižší hodnotu

Dva rozdíly od LRU:

* několik stránek může mít stejnou hodnotu pole age a nevíme která stránka byla odkazovaná dříve (u LRU to víme vždy)
 - rozlišení je "hrubé" (= po tících časovače)
* pole age se může snížit na 0 - nevíme, zda stránka byla naposledy odkazovaná před 9 nebo před 1000 tiky časovače
 - uchovává pouze omezenou historii
 - v praxi není problém: pokud je tik časovače po 20 ms a N=8, nebyla odkazována 160 ms => nejspíš není tak důležitá, můžeme jí vyhodit
* pokud se musíme rozhodovat mezi dvěma stránkami se stejnou hodnotou age, vybíráme náhodně

Shrnutí algoritmů pro nahrazování stránek
.....

* optimální algoritmus (MIN čili OPT)
 - není implementovatelný, ale je užitečný pro srovnání
* FIFO
 - vyhazuje nejstarší stránku
 - jednoduchý, ale je chopen vyhodit důležité stránky a trpí Beladyho anomálií
* LRU (Least Recently Used)
 - výborný
 - implementace vyžaduje speciální HW, proto prakticky používán zřídka
* NRU (Not Recently Used)
 - rozděluje stránky do 4 kategorií podle bitů R a M
 - efektivita nic moc, přesto občas používán
* "Second chance" a "Clock"
 - vycházejí z FIFO, před vyhozením zkontrolují zda se stránka používala
 - mnohem lepší než FIFO
 - používané algoritmy (např. některé varianty UNIXu)
* Aging
 - dobře aproximuje LRU => efektivní
 - často prakticky používaný algoritmus

Ostatní problémy stránkované virtuální paměti

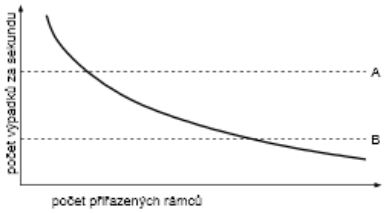
Alokace fyzických rámců
.....

* 2 základní metody - globální a lokální alokace:
 - globální alokace - pro vyhození se uvažují všechny rámce
 - lokální alokace - pro vyhození se uvažují pouze rámce alokované procesem (tj. obsahující stránky procesu, jehož výpadek stránky se obsluhuje)
  . počet stránek alokovaných pro proces se nemění
  . program se vzhledem ke stránkování chová přibližně stejně při každém běhu
 - u globální alokace vybírá ze všech rámců
  . lepší průchodnost systému - proto globální alokace častější
  . na běh procesu má vliv chování ostatních procesů

* při lokální alokaci - kolik rámců dát kterému procesu?
 - nejjednodušší - všem procesům dáme stejně, ale potřeby procesů mohou být různé
 - proporcionální - dáme každému proporcionální díl podle velikosti procesu

- nejlepší metoda - podle frekvence výpadků stránek (Page Fault Frequency, PFF)

Pro většinu rozumných algoritmů se PFF snižuje s množstvím přidělených rámců:



PFF se snažíme udržet v rozumných mezích:

* pokud je PFF větší než A, přidáme procesu rámce
* pokud je PFF menší než B, proces má asi příliš paměti, rámce mu mohou být odebrány

Zahlcení
.....

* proces pro svůj rozumný běh potřebuje pracovní množinu stránek
* pokud se pracovní množiny stránek aktivních procesů nevejdou do paměti, nastane tzv. zahlcení (angl. trashing)

* jak to vypadá:
 - v procesu nastane výpadek stránky
 - paměť je plná (není volný rámec) => je třeba některou stránku vyhodit
 - stránka pro vyhození bude ale brzy zapotřebí, takže se bude muset vyhodit jiná používaná stránka
 - z uživatelského hlediska se to projeví tak, že systém pracuje intenzivně s diskem a běh procesů řádově zpomalí (stráví víc času stránkováním než během)

* řešení - při zahlcení snížit úroveň multiprogramování (zahlcení lze detekovat pomocí PFF)

Zhodnocení mechanismu virtuální paměti
.....

Virtuální paměť má podstatné výhody oproti předchozím mechanismům:

* rozsah virtuální paměti (např. 2 GB pro proces - NT nebo Linux na i386)
 - adresový prostor úlohy není omezen velikostí fyzické paměti
 - multiprogramování (= počet procesů) není zásadně omezeno rozsahem fyzické paměti
* efektivnější využití fyzické paměti
 - není vnější fragmentace paměti
 - nepoužívané části adresového prostoru úlohy nemusejí být ve fyzické paměti

Nevýhody:

* režie při převodu virtuálních adres na fyzické adresy
* režie procesoru (údržba tabulek stránek a rámců, výběr stránek pro vyhození, plánování I/O)
* režie I/O při čtení/zápisu stránky
* paměťový prostor pro tabulky stránek
* vnitřní fragmentace

```
Segmentace
----------

* dosud diskutovaná virtuální paměť byla jednorozměrná:
  od adresy 0 do nějaké maximální virtuální adresy
* pro mnoho programů by bylo výhodnější mít víc samostatných virtuálních
  adresových prostorů
* příklad - mám několik tabulek a chci, aby jejich velikost mohla růst
=> paměť nejlépe mnoho nezávislých adresových prostorů = segmenty

* segment = logické seskupení informací
* každý segment lineární posloupnost adres, začínající od adresy 0
* programátor o segmentech ví, používá explicitně (adresuje konkrétní
  segment)
* příklad - překladač Pascalu může používat samostatné segmenty pro:
  - kód přeloženého programu
  - globální proměnné
  - hromadu
  - zásobník návratových adres
  - je možné i jemnější dělení (segment pro každou proceduru/fci)
* často se používá také pro implementaci:
  - přístupu k souborům (1 soubor = 1 segment)
    . není třeba open, read...
  - sdílených knihoven:
    . dnešní programy využívají rozsáhlé knihovny - knihovnu potřebuje prakticky každý
      program
    . myšlenka vložit knihovnu do segmentu a sdílet mezi více programy

* každý segment je logická entita - má smysl, aby měl samostatnou ochranu

Čistá segmentace
................

* každý odkaz do paměti se skládá z dvojice: (selektor, offset)
 - selektor: číslo segmentu, určuje segment
 - offset: relativní adresa v rámci segmentu
* technické prostředky musí přemapovat dvojici (selektor, offset) na fyzickou
  (= lineární) adresu
* k tomu slouží tabulka segmentů, každá položka tabulky obsahuje:
  - počáteční adresu segmentu (bázi)
  - rozsah segmentu (limit)
  - příznaky ochrany segmentu (nejčastěji čtení, zápis, provádění = rwx)
* postup při převodu na fyzickou adresu:
  - PCB obsahuje odkaz na tabulku segmentů procesu
  - odkaz do paměti má tvar (selektor, offset)
    např. v důsledku instrukce LD R, selektor:offset
  - selektor = index do tabulky segmentů
  - zkontroluje se zda je offset<limit; ne => chyba porušení ochrany paměti
  - zkontroluje se, zda dovolen způsob použití; ne => chyba porušení
    ochrany paměti
  - adresa = báze + offset
* často možnost sdílet segment mezi více procesy

* mnoho věcí podobných jako přidělování paměti po sekcích, ale rozdíl:
  - po sekcích - pro procesy
  - segmenty - pro části procesu
* stejné problémy jako přidělování paměti po sekcích: externí
  fragmentace paměti, mohou zůstávat malé díry (tj. dále již prakticky
  nepoužitelné)

Segmentace na žádost
....................

* segment může být zavedený v paměti nebo odložený na disku
* pokus o adresování segmentu, který není v paměti způsobí výpadek segmentu
* OS zavede segment do paměti
* není-li místo, je některý jiný segment odložen na disk

* HW podpora - v tabulce segmentů bity:
```

3.1.6. Obsluha přerušení, výjimek a systémových volání

Přerušení

metoda pro asynchronní obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí, vykoná obsluhu přerušení a pak pokračuje.

- **vnitřní** - vyvolané procesorem (problém se zpracováním instrukcí, dělení 0, výpadek stránky)
  - Také nazývané **synchronní**, a to proto, že CPU jej vyvolá až po dokončení aktuálně vykonávané instrukce, Intelovský manuál je nazývá **Exception**
- **vnější** - hardwarové, přichází z V/V zařízení, mají přidělena čísla IRQ (Interrupt Request)
  - pro signalizaci přerušení– *kanály přerušení*, reprezentovány čísly IRQ, na jednom IRQ kanálu může být napojeno více zařízení (= sdílený kanál, sdílené IRQ)
  - Také nazývané **asynchronní**, a to proto, že závisí na ticích časovače a může nastat v době kdy CPU právě vykonává nějakou instrukci, Intelovský manuál je nazývá **Interrupt**
- **softwarové** - speciální instrukce INT 0x80 nebo SYSENTER (SYSCALL)

Další dělení je na maskovatelná a nemaskovatelná

Interrupts:
 Maskable interrupts: All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts . A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.
 Nonmaskable interrupts: Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts . Nonmaskable interrupts are always recognized by the CPU.

- Přerušení mají priority - obsluha přerušení může být přerušena přerušením s vyšší prioritou :) - pozn. Priority z hlediska kernelu jsou jen **low** a **high**, tedy přerušitelné a nepřerušitelné. APIC (Advanced Programmable Interrupt Controller ale priority podporuje, takže priority přerušení jako takové Linux nikde neřeší, ale nechává to na hardware (APIC je v každém CPU).
- Přerušení je obecně asynchronní vzhledem k přerušenému procesu
- Zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící
- čas zpracování přerušení je započítán přerušenému procesu, při zpracování se tedy přistupuje do jeho záznamu proc

Obsluha:

1) uloží IRQ (Interrupt ReQuest) a obsah registrů

2) pošle potvrzení PIC (Programmable Interrupt Controller), který zajišťuje provoz přerušení

3) *modul pro obsluhu přerušení* (Interrupt Handler) vykoná obsluhu přerušení

4) ukončí skokem na ret_from_intr()

Výjimky

- synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)

Zpracování:

- Stejné jako u přerušení, 80x86 procesory mají přibližně 20 definovaných výjimek, každá je zpracována přiřazeným exception handlerem - ten většinou nedělá nic jiného než že pošle signál procesu, který výjimku způsobil, kernel musí poskytnout exception handler pro každou definovanou výjimku, příklady výjimek i s jejich číslem: 0 - Divide Error = dělení nulou, 1 - Debug, 4 - Overflow = přetečení…

1) uloží obsah registrů

2) zpracuje výjimku (funkce v jazyku C)

- pošle signál procesu
- zpracuje žádost o stránku

3) ukončí se voláním funkce ret_from_exception()

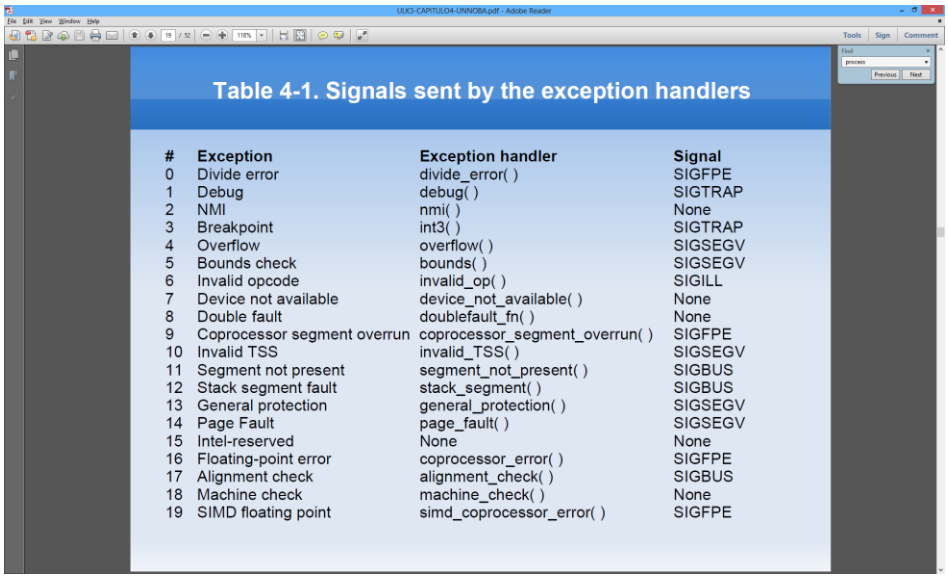


Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
0	Divide error	divide_error( )	SIGFPE
1	Debug	debug( )	SIGTRAP
2	NMI	nmi( )	None
3	Breakpoint	int3( )	SIGTRAP
4	Overflow	overflow( )	SIGSEGV
5	Bounds check	bounds( )	SIGSEGV
6	Invalid opcode	invalid_op( )	SIGILL
7	Device not available	device_not_available( )	None
8	Double fault	doublefault_fn( )	None
9	Coprocessor segment overrun	coprocessor_segment_overrun( )	SIGFPE
10	Invalid TSS	invalid_TSS( )	SIGSEGV
11	Segment not present	segment_not_present( )	SIGBUS
12	Stack segment fault	stack_segment( )	SIGBUS
13	General protection	general_protection( )	SIGSEGV
14	Page Fault	page_fault( )	SIGSEGV
15	Intel-reserved	None	None
16	Floating-point error	coprocessor_error( )	SIGFPE
17	Alignment check	alignment_check( )	SIGBUS
18	Machine check	machine_check( )	None
19	SIMD floating point	simd_coprocessor_error( )	SIGFPE

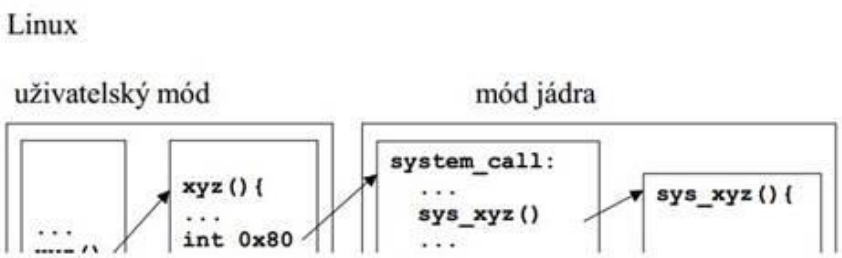
Vektor přerušení

- 0-31 vnitřní přerušení (výjimky), 32 - 255 IRQ přerušení

Vektor přerušení u x86 se nazývá IDT (Interrupt Descriptor Table) - tabulka, která přiřazuje obslužnou rutinu ke každému přerušení

Systémové volání

- mechanismus pro volání funkcí operačního systému aplikacemi
- v standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá softwarovým přerušením procedruře jádra syscall()
- system_call - jedna pro všechny služby, požadovaná služba se odlišuje parametrem procedury, který se nazývá číslo systémového volání



Obsluha:

system_call:

1) uloží obsah registrů (HW kontext)

2) zavolá odpovídající funkci (v jazyku C)

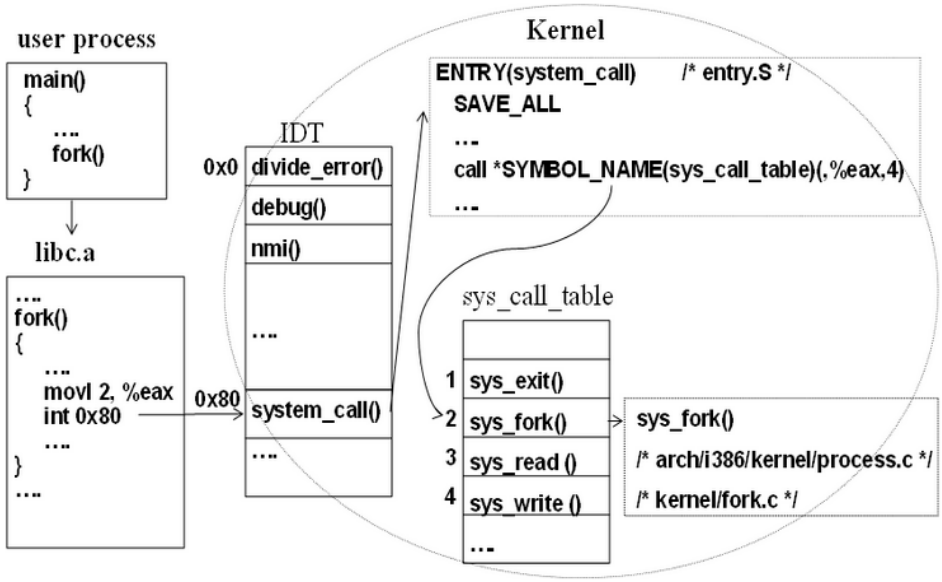
3) ukončí se voláním ret_from_sys_call()

Int 0x80 je legacy instrukce, funguje jen u 32 bit systému a dnes se nepoužívá

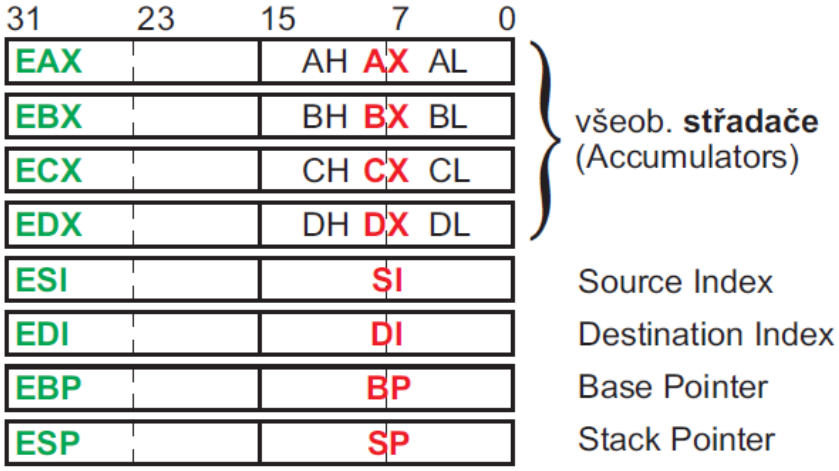
- `syscall` is default way of entering kernel mode on `x86-64`. This instruction is not available in 32 bit modes of operation *on Intel processors*.
- `sysenter` is an instruction most frequently used to invoke system calls in 32 bit modes of operation. It is similar to `syscall`, a bit more difficult to use though, but that is kernel's concern.
- `int 0x80` is a legacy way to invoke a system call and should be avoided.
  Preferable way to invoke a system call is to use VDSO, a part of memory mapped in each process address space that allow to use system calls more efficiently (for example, by not entering kernel mode in some cases at all). VDSO also takes care of more difficult, in comparison to the legacy `int 0x80` way, handling of `syscall` or `sysenter` instructions.

  From <http://stackoverflow.com/questions/12806584/what-is-better-int-0x80-or-syscall>

  VDSOs (**V**irtual **D**ynamically linked **S**hared **O**bjects) are a way to export kernel space routines to user space applications, using standard mechanisms for linking and loading (i.e. standard ELF format).
  It helps to reduce the calling overhead on simple kernel routines, and also can work as a way to select the best system call method on some architectures.
  An advantage over other methods is that such exported routines can provide proper DWARF debugging information.
  Implementation generally implies hooks in the dynamic linker to find the **VDSO**s.



Všeobecné registry procesoru x86 (od 80386 dále)



- tyto registry jsou obecně použitelné v programu k dočasnému ukládání dat (až na ESP - s tím opatrně)
- naplňují se instrukcí `MOV` *reg*, *hodnota*, např. `MOV BL, 5`

3.1.7. Implementace režimu jádra a uživatelského režimu

Výpočet v módu jádro – v důsledku událostí:

Popsat rozdíl mezi uživatelským módem a režimem jádra.

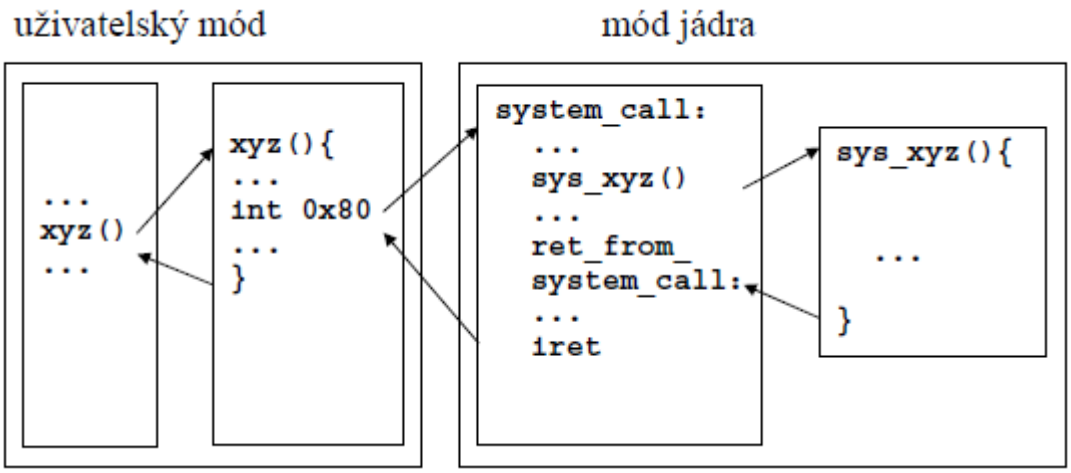
- Přerušení
- Výjimky
- Softwarové přerušení

Řízení se předá na proceduru pro ošetření odpovídající události. Část stavu přerušeného procesu potřebná pro jeho vykonávání po skončení obsloužení události (počítadlo instrukcí, PSW (**Processor Status Word**) se uloží do zásobníku jádra přerušeného procesu.

Systémové volání

V standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá software přerušením proceduře jádra, která se nazývá syscall(), system_call. Protože je jen jedna pro všechny služby, požadovaná služba je identifikována parametrem procedury, který se nazývá číslo systémového volání.

Linux



Aplikační program volá službu xyz, obálková procedura uloží číslo služby do registru **eax** před vykonáním **int 0x80**.

System_call

- Uloží obsah registrů (hardwarový kontext)
- Zavolá odpovídající funkci (v jazyce C)
- Ukončí se voláním ret_from_sys_call()

Výjimky se zpracují obdobně

- jsou synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)
- Procedury pro jejich zpracování mají obdobnou strukturu jako procedura pro systémová volání **systém_call**

Linux

Procedura pro zpracování výjimky

- Uloží obsah registrů
- Zpracuje výjimk (funkce v jazyku C)
  - Pošle signál procesu
  - Zpracuje žádost o stránku
- Ukončí se voláním funkce ret_from_exception()

Zpracování přerušení

Přerušení je obecně asynchronní vzhledem k přerušenému procesu – proces čeká na přenos dat, po dokončení přenosu je přerušen úplně jiný proces. Zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící. Čas zpracování přerušení je započítán přerušenému procesu, při zpracování přerušení se tedy přistupuje do jeho záznamu **proc**.

Obsloužení přerušení:

- Uloží IRQ (Interrupt ReQuest) a obsah registrů
- Pošle potvrzení PIC (Programmable Interrupt Controller)
- Vykoná obslužní proceduru přerušení
- Ukončí se skokem na ret_from_intr()

Vzájemné vnoření systémového volání, výjimek a přerušení

Předpokládejme odladěné jádro

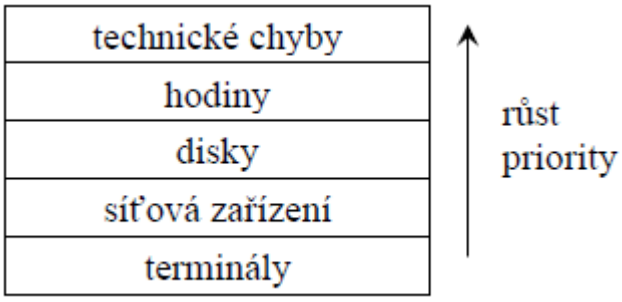
1. Zpracování systémového volání
   - Může vzniknout výjimka při žádosti o stránku (výpadek stránky)
   - Může dojít k přerušení
2. Zpracování výjimky
   - Může dojít k přerušení
3. Zpracování přerušení
   - Může dojít k přerušení

Při každém odkladu zpracování některé z uvedených událostí musíme nejdříve uložit odpovídající HW kontext.

- Vytváří se kontextové vrstvy zásobníku jádra přerušeného procesu
- Existuje globální zásobník přerušení

Prioritní schéma

- Přerušení mají přiřazené prioritní úrovně (interrupt priority level)

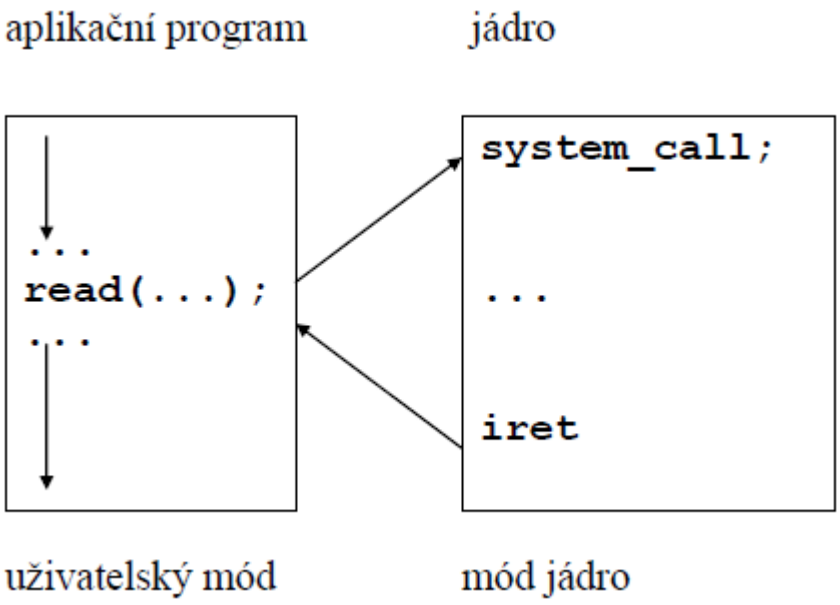


  Ve stavovém registru procesoru je nastavena okamžitá prioritní úroveň zpracovávaného přerušení.

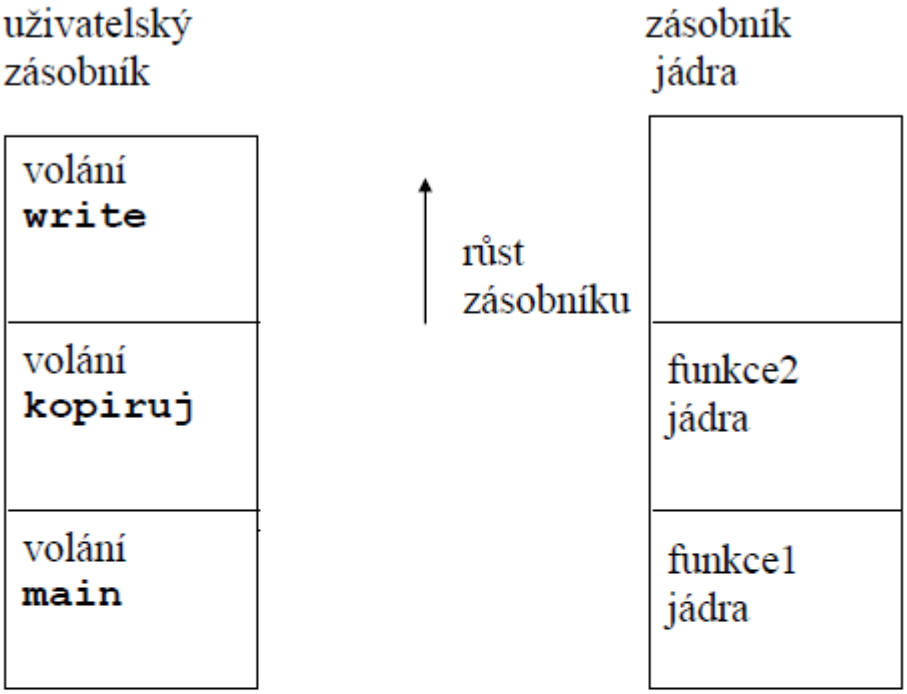
Vznikne-li přerušení přerušení s nižší nebo stejnou prioritní úrovní, je uloženo a jeho obsluha je odložena.
Vznikne-li přerušení s vyšší prioritní úrovní, uloží se HW kontext, na tuto vyšší prioritní úroveň se nastaví hodnota okamžitého přerušení ve stavovém registru procesoru a přerušení se zpracuje.

Při skončení zpracování přerušení se z uloženého PSW obnoví okamžitá prioritní úroveň přerušení.

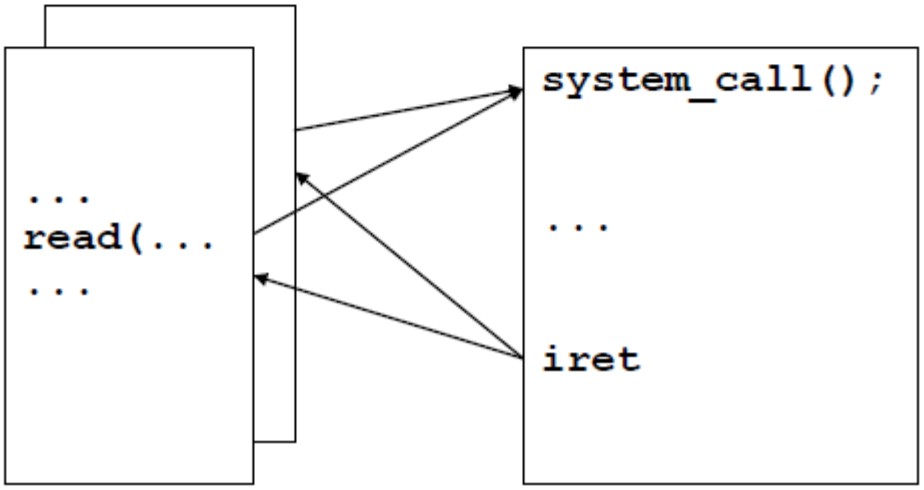
Proces a jádro



- uživatelský
- jádra



více procesů



Jádro je reentrantní – každý proces má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem. Každý proces má položku v tabulce procesů: **proc** záznam a **u** (user) oblast.

Z PPR

Princip – MSDOS compatible, uniprocesor

V reálném režimu: Pomocí služby DOSu se nainstaluje handler přerušení 8, které generují hodiny. Když je volána obsluha přerušení, v zásobníku jsou uloženy registry CS, IP a Flags kódu, jehož vykonávání bylo přerušeno. Obsluha přerušení uloží stávající registry, plánovač vybere novou úlohu a zapíše do zásobníku její hodnoty uvedených registrů (CS, IP a Flags). Obsluha přerušení obnoví zbývající registry procesoru pro plánovačem vybranou úlohu. Provede se instrukce iret, kterou se spustí naplánovaný proces díky přepsání hodnot v zásobníku.

3.1.8. Proces a thread –stavy, implementace, plánování, synchronizace

Wiki: Thread je nejlehčí jednotka plánování, TXKoutný: Proces - *největší* výpočetní entita plánovače.

Je potřeba rozlišovat thread programátorský a thread z hlediska plánovače – dvě úplně odlišné věci, ale v literatuře se oboje jmenuje thread a pak jsou z toho zmatky.

- Viz přednáška 6 OS

Typy vláken dle OS
- jádrová
- lehké procesy
- uživatelská

Process

- Má svůj vlastní kontext a adresní prostor
- Může mít jedno nebo více vláken v jednom (svém) adresním prostoru
- Vlákna uvnitř procesu sdílejí prostředky procesu (otevřené soubory)

Thread

- Kernel thread: jednotka plánování plánovače jádra (činnost plánovače je založená na přepínání kernel threadů), jejich počet je nezávislý na počtu jader procesoru nebo počtu procesorů
  - pouze jádrová věc
- User thread: uživatelem vytvořený thread v rámci prostředků programovacího jazyka
  - Bez podpory plánovače jádra: fiber (100% user space)
  - S podporou plánovače jádra: lightweight proces (lehký protože nemá vlastní adr. prostor, ukazatele na soubory apod.) – fiber namapovaný na kernel thread
- Sdílená paměť a Thread-local storage (může mít a nemusí, v rámci adresního prostoru procesu)

Fiber

- Běží v uživatelském prostoru a při přepnutí fiberů se nepřepíná kontext – rychlé a levné
- Spolupracují kooperativně (ne preemptivně) – musí udělat yield, jsou **implicitně synchronizované**
- Méně problematická thread-safety: nejsou potřeba spinlocky a atomické operace
- Vyžaduje menší podporu od OS, nemusí požadovat vůbec žádnou (třeba podle výhodnosti plánování –OS může a nemusí mít „lepší“ plánovač). Podporují je Unixové systémy, Microsoft, Symbian…
- Nemohou využívat víc procesorů (všechny fibery jsou v jediném kernel threadu)
- Sdílená paměť a Fiber-local storage

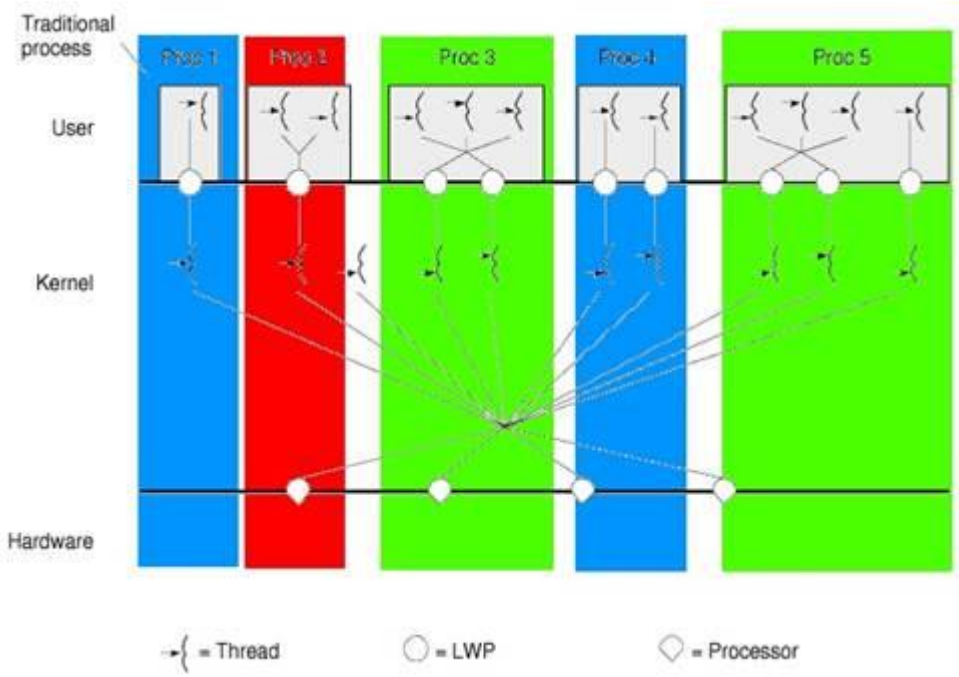
LWP (Light-Weight Process)

- Linux: http://www.linuxforu.com/2011/08/light-weight-processes-dissecting-linux-threads/
  - Kernel thread = LWP
  - Obsluha user-thread (u Linux = process) na LWP 1:1
- Na Windows: http://www.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/03-ThreadScheduling/ThreadScheduling.pdf
  - Windows plánuje vlákna, ne procesy

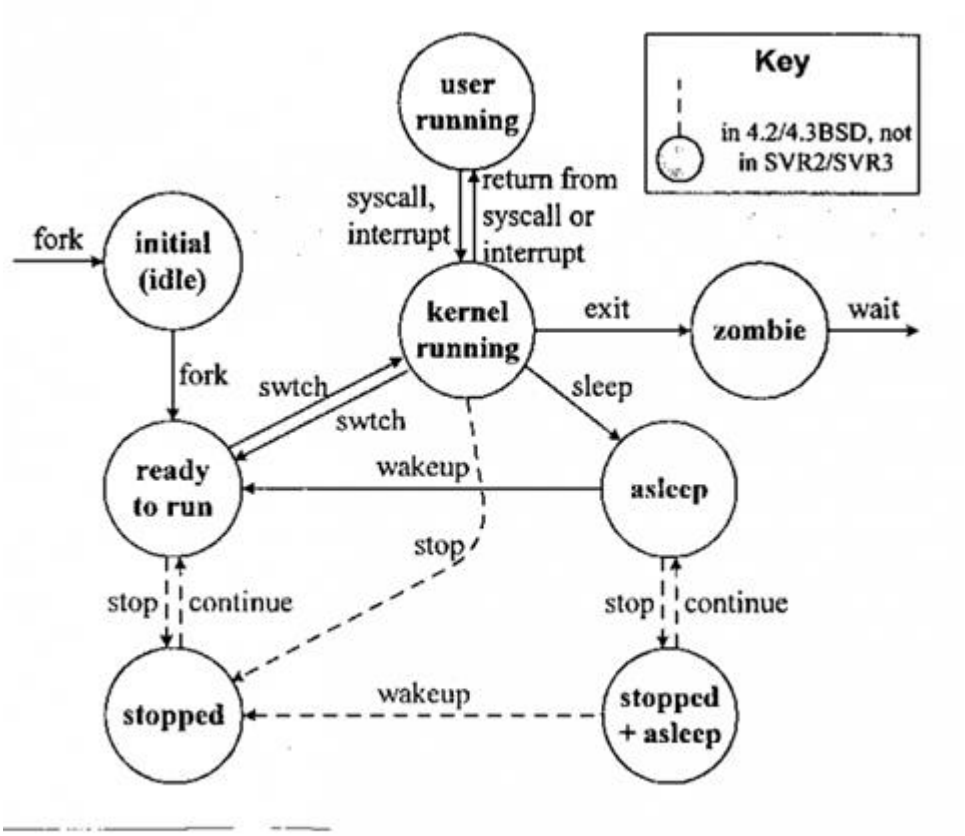
- Plánování je preemptivní, založeno na prioritě a používá round-robin pro nejvyšší priority
- Každý thread má aktuální a základní prioritu - základní priorita je inicializována při spuštění procesu, aktuální pak závisí na chování threadu - priorita se snižuje, pokud thread vždy vyčerpá přidělené kvantum

Vláknové modely

- **1:1 (kernel vlákna)** Vlákna vytvořená uživatelem odpovídají počtem 1:1 entitám, které plánuje jádro. Nejjednodušší přístup, ale je třeba uvážit preempci a thread-safety
- **N:1 (uživatelská vlákna)** všechna aplikační vlákna jsou namapována na jedno jádrové vlákno, jádro nemá tušení o tom, že aplikace běží vláknově.
- **M:N (hybridní)** komplexní na implementaci (vyžaduje změny v jádře i uživ. prostoru) ale umožňuje zvýšení efektivity výpočtu (například přiřazení více uživatelských vláken více vláknům jádra – proces může běžet na více procesorech).
-



Stavy procesu

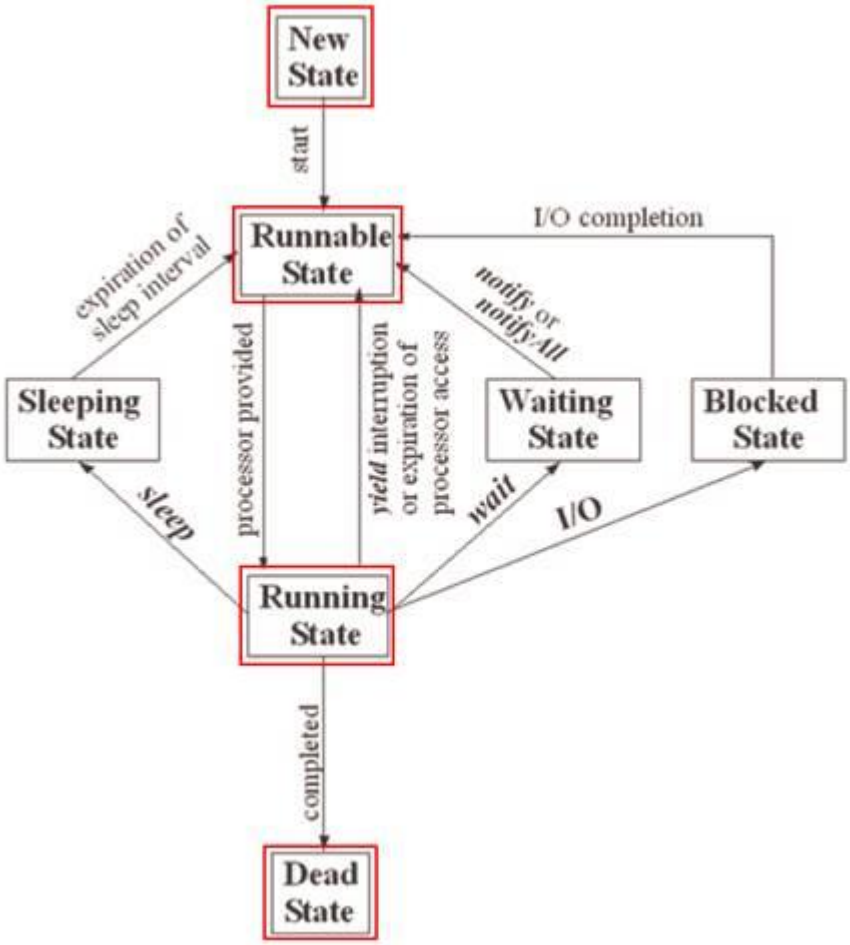


Process states and state transitions.

- **Počáteční** (initial/idle) – fork začal vytváření procesu.
- **Připraven na vykonání** (ready to run) – fork dokončil vytváření procesu, proces čeká až bude naplánován na přidělení procesoru.
- **Běžící v jádře** (kernel running) – byl naplánován, vykoná se přepnutí kontextu, procedura jádra swtch() uloží HW kontext do registrů.
- **Běžící uživatelsky** (user running) – proces vykonává svůj programový kód. Může přejít do stavu běžící v jádře v důsledku volání služby jádra nebo přerušení, po skončení obsluhy se vrátí.
- **Spící** (asleep) – při vykonávání systémového volání se může stát, že je nutno čekat na nějakou událost nebo prostředek, proces (v jádře) proto zavolá proceduru sleep(). Když událost nastane, jádro vzbudí proces, proces se stane připraven na vykonání a po naplánování pokračuje obsluha systémového volání ve stavu běžící v jádře.

Připraven na vykonání se může stát také, je-li běžící a uplyne mu přidělené časové kvantum - vykoná se preempce běžícího procesu a to ve stavu běžící uživatelsky nebo při návratu do něj. Jádro je nepreemptivní. Přerušení se může vyskytnout i ve stavu běžící v jádře, kdy po skončení obsluhy přerušení proces pokračuje ve stavu běžící v jádře.

- **Mátoha** (zombie) - proces končí voláním exit() anebo v důsledku signálu, dokud rodič nevykoná wait().
- **Zastaven** (stopped) - do něj přejde proces, je-li běžící nebo spící (+ asleep), po stop signálech:
  - SIGSTOP zastav proces
  - SIGTSTP CTRL-Z
  - SIGTTIN tty čtení procesu v pozadí
  - SIGTTOU tty psaní procesu v pozadí
  - SIGCONT převede proces do stavu připraven na vykonání nebo do stavu spící.



Plánování, plánovací třídy, inverze priority

Víceúlohové systémy:

- systémy reálného času
- interaktivní systémy
- dávkové systémy

Ve víceúlohových systémech sdílení času je tradiční problém „vhodně“ a spravedlivě přidělovat čas jednotlivým procesům (adekvátně k typu – rt/dávkový) a zajišťovat vysokou průchodnost (tyhle tři požadavky jsou často v kontradikci). Je třeba hlídat aby nedošlo k vyhladovění a k inverzi priorit.

Problém velikosti časového kvanta: v interaktivních systémech je třeba přepínat často, aby byl vytvořen dojem okamžité odezvy, ale přepínání HW kontextu je náročné a zdržuje = čím kratší čas, tím větší režie systému.

Typy plánování:

- preemptivní – s předbíháním, proces je možno přerušit zvenčí, je potřeba zavést synchronizační primitiva
- nepreemptivní – proces se musí vzdát procesoru sám (fiber)

Plánování:

- **FIFO** – nejjednodušší, neadaptabilní, málokdy spravedlivý. Když nějaký proces nedoběhne, může dojít k vyhladovění

- **Shortest proces (job) first -** *nepreemptivní*, u dávkových úloh, předpokládá se znalost dob trvání procesů, vezme se proces/job s nejkratší předpokládanou dobou běhu, po jeho skončení se z fronty bere další proces; optimalizuje dobu obrátky
- **shortest remaining time –** *preemptivní,* vybere úlohu, jejíž zbývající doba běhu je nejkratší - když je plánovaná úloha s 10 min do jejího dokončení a přijde úloha trvající 1 minutu, systém provede přeplánování a začne běžet ta nová úloha; dobrý pro krátký (hl. I/O vázaný) úlohy, může ale dojít k vyhladovění (v systému je hodně „krátkých" úloh, na ty dlouhodobější se nedostane řada)
- **Round robin** – procesy obdrží každý stejný čas a střídají se dokola, nedojde k vyhladovění protože není zavedena priorita
- **Prioritní plánování** – Každý proces má svou prioritu (procesy jádra nejvyšší, interaktivní vysokou, dávkové nízkou), procesy jsou podle priority roztříděny do tříd, ve kterých cyklují metodou Round Robin (vždy nejdřív první neprázdná třída od nejvyšší priority). Může dojít k vyhladovění procesů s nízkou prioritou (řešením je zvyšovat prioritu dlouho nenaplánovaným procesům).
    - priorita **statická** (při startu procesu) a **dynamická** (chování procesu v poslední době, snižuje ji u běžícího procesu při každém tiku plánovače; = 1/f, *f* je velikost částí kvanta, kterou proces naposledy použil)
- Prakticky ve všech dnešních OS, mnoho různých metod:
- **Fair Share** - Férové rozdělení času mezi uživatele, nikoli procesy. Rekurzivní aplikace Round Robin na každé úrovni abstrakce - nejprve na skupiny, pak na uživatele, pak na procesy.
- **Loterie** - plánovač má k dispozici pevný počet tiketů, každý proces obdrží určitý počet tiketů a plánovač pak vybere jeden tiket - **náhodně** vybere tiket a přidělí časové kvantum procesu, který ten tiket vlastní
- **Epochy** - čas procesoru je rozdělen do epoch
    a. každý proces má specifikováno časové kvantum v rámci epochy
    b. v jedné epoše proces může využívat své časové kvantum po částech
    c. epocha končí, když všechny běhu schopné procesy vyčerpaly svá časová kvanta
    d. Délka časového kvanta v epoše závisí na prioritě procesu.

Jaký je rozdíl mezi NoRealTime, SoftRealTime a HardRealTime

NoRealTime – Nemají žádný deadline.

SoftRealTime – překročení termínu se toleruje, systém reaguje zhoršenou kvalitou poskytovaných služeb – např. vypadne pár snímků, nebo přílet letadla se dozvíte s několikasekundovým zpožděním.

HardRealTime – Dokončení výpočtu po termínu se považuje za chybu a výsledek za bezcenný – strict deadline. Nedodržení termínu může vést k celkovému selhání systému (airbag, jaderná elektrárna…).

- Soft RT
- Hard RT

Viz níž.

Inverze priority

Inverzí priorit rozumíme situaci, která nastane, pokud vlákno s vyšší prioritou požaduje přístup k systémovým zdrojům, které v danou chvíli právě exkluzivně drží vlákno s nižší prioritou. V tomto případě dojde k preempci do vlákna s vyšší prioritou, které však nemůže běžet díky zablokovanému systémovému zdroji. To je velice nepříjemná situace, zejména v RTOS. Jediným řešením je umožnit vláknu, které systémový zdroj drží co nejrychleji doběhnout a umožnit tak i jiným vláknům pokračovat v jejich činnosti. **K vyřešení této situace se používá systém inverze priorit**, který umožní vláknu s nižší prioritou zdědit prioritu kritického vlákna, rychle vykonat potřebné operace až do

chvíle uvolnění požadovaného systémového zdroje a dále pak nechat pokračovat v práci kritické vlákno.

Synchronizace

Kernel space (viz kapitola 9), vs. user space. Eventy (WinAPI v user space) a další (?)

Process Control Block (PCB)

Udržuje informace spojené s každým procesem, je využit při znovu rozběhnutí přerušeného procesu.

- Stav procesu
- Program counter
- CPU registry
- Informace o CPU plánování
- Info o správě paměti
- Info o status I/O (File descriptory, sockety atd.)

3.1.9. **Signály**

- Jen u Linuxu, Windows mají "alternativu" - message
- Signály jsou přerušení generována softwarem, která jsou zaslána procesu, pokud nastane nějaká událost

Unix systémy používají signály, aby daly **vědět procesu, že nastala určitá událost** (proces je pak např. vzbuzen, přerušen, …). Oznamují se jen čísla signálu, žádné parametry.

Signál vs. přerušení

Hezky podané na:

- http://stackoverflow.com/questions/13341870/signals-and-interrupts-a-comparison
- Na *přerušení* jde pohlížet jako na prostředek komunikace *mezi procesorem a jádrem OS*
- *Signály* mohou být brány jako prostředek komunikace *mezi jádrem OS a procesy*
  - Mohou být započaty jádrem OS (SIGSEGV, SIGIO) nebo procesem (*kill()*)
  - Jsou nakonec spravovány jádrem OS, které je doručí do cílového procesu/vlákna a spustí buď nějakou obecnou akci (ignorovat, ukončit, ukončit + dump core) a nebo obsluhu signálu, kterou poskytl proces

Synchronní a asynchronní signály

Signály mohou být buď **synchronní nebo asynchronní**, záleží na zdroji a důvodu, proč byla událost signalizována.

Mezi synchronní signály patří např. neoprávněný přístup do paměti a dělení nulou. Pokud běžící program provede některou z těchto akcí, vygeneruje se signál. Synchronní signály jsou doručeny stejnému procesu, který provedl operaci, která ten signál způsobila (což je důvod, proč jsou považovány za synchronní).

Když je signál generován událostí, která je externí vzhledem k běžícímu procesu, tak ten proces přijme ten signál asynchronně. Příklady takových signálů jsou ukončení procesu pomocí konkrétní klávesové kombinace (třeba <control><C>) a vypršení časovače. Asynchronní signál je typicky zaslaný jinému procesu.

Obsluha signálu

Jakmile byl signál generován výskytem nějaké události (např. dělením nulou, neoprávněným přístupem do paměti, uživatel stisknul CTRL+C = SIGINT signál), signál je dopraven procesu, kde musí být zpracován. Proces, který přijme signál, jej může zpracovat různými způsoby:

- Ignorování signálu (kromě SIGKILL a SIGSTOP signálů)
- Použití defaultního (výchozího) signal handleru (obsluhy signálu/funkce na zpracování signálu dle Košičana)
- Poskytnutí vlastní signal-handling funkce = vlastní obsluha signálu (kromě SIGKILL a SIGSTOP).

Každý signál má svou **výchozí obsluhu signálu (*default signal handler*),** která běží v jádře, když je signál zpracováván **(???**that is run by the kernel when handling that signal). Tato výchozí akce může být přepsána **uživatelem definovanou obsluhou signálu**, který je volán pro obsluhu daného signálu. Signály mohou být obslouženy různými způsoby. Některé signály (jako změna velikosti okna) mohou být jednoduše ignorovány; jiné (jako třeba neoprávněný přístup do paměti) mohou být obslouženy ukončením programu.

Signály mohou být obslouženy nastavením určitých proměnných v C struktuře `struct sigaction` a pak předáním této struktury `sigaction()` funkci. Signály jsou definovány v include souboru `/usr/include/sys/signal.h`. Např. signál SIGINT reprezentuje signál pro ukončení programu pomocí <Control> <C>. Výchozí obsluha signálu (signal handler) pro SIGINT je ukončit program.

Další možností je, že v programu může být nastavena vlastní funkce pro obsluhu signálu, a to nastavením `sa_handler` proměnné ve struktuře `struct sigaction` na název funkce, která obslouží ten signál, a pak zavoláním funkce `sigaction()`. Té se předají jako parametry (1) signál, pro který nastavujeme obsluhu, a (2) pointer na `struct sigaction`.

Fáze signálů

Všechny signály, ať už synchronní nebo asynchronní, mají stejný životní cyklus:

1. Signál je vygenerován a **odeslán** poté, co nastane nějaká událost.
   - PM (*Process manager*) nejprve zjistí, které procesy mají obdržet signál
     - V tabulce procesů je pro každý proces několik `sigset_t` proměnných (=bitmapy, definují ignorované, zachycované signály)
   - Pro procesy zachycující daný signál:

     1) jádro zaznamená v záznamu proc (deskriptoru procesu) cílového procesu odeslání nového signálu.

     2) Jádro přeruší standardní provádění posloupnosti instrukcí cílového procesu, uloží informace o stavu procesu, aby se pak mohl opět pokračovat v běhu. Informace jsou uloženy na zásobníku toho procesu (kterému má dorazit signál) + kontrola, že je dost místa na zásobníku (dělá PM).
   - PM pak volá system task in jádře pro uložení informací do zásobníku. System task také manipuluje s program counterem procesu, aby proces mohl být spuště v kódu obsluhy.
2. Vygenerovaný signál je **doručen – přijat -** procesu.
3. Jakmile je signál doručen, musí být zpracován.
   - Když obsluha skončí, je provedeno systémové volání `sigreturn`. Prostřednictvím tohoto volání se PM i jádro podílejí na obnově kontextu signálu a registrů „signalizovaného“ procesu, aby mohl pokračovat v normálním běhu.
   - Pokud není signál zachycen (nemá def. handler), podnikne se defaultní akce, která se může týkat volání souborového systému pro vytvoření **core dumpu** (zápis obrazu paměti procesu do souboru, který může být prozkoumán debuggerem) či zabití procesu, pro něž je třeba zapojit PM, souborový systém a jádro.
   - PM řídí jednu nebo více opakování akcí výše - podle toho, jestli je signál doručen jednomu procesu nebo skupině procesů.

Fáze vypořádání se signály jinak:

1. Příprava (Preparation) – kód programu se připraví pro možný signál
   - Několik systémových volání, která lze nastavit jako odpověď na signál
   - `sigaction` → co má proces dělat se signálem: ignorovat/zachytit/nastavit defaultní reakci
   - `sigprocmask()` → blokování signálu; signál bude zařazen do fronty či se jím bude řídit až jej process později odblokuje
   - `sigsuspend(sigmask)` → nastaví se blokované signály podle `sigmask` a process přejde do stavu *čekající* až do zaslání signálu, který není blokován/ignorován. Dle *[Košičan, prednaska07, slide 13]* není to samý, co `sigprocmask()` a `sleep()`, ???nechápu
2. Odpověď (Response) – **signál je přijat** a příslušná akce je vykonána
3. Vyčištění (Cleanup) – obnova normální operace procesu

- Viz bod 3 předchozího rozfázování

Příklady scénářů synchronního a asynchronního signálu

Synchronní:

- výjimka (dělení nulou, nedovolená instrukce,...) způsobí přechod do módu jádro
- jádro vykoná její obsluhu a zašle se odpovídající signál běžícímu procesu
- při návratu z obsluhy proces najde signál

Asynchronní:

- uživatel stiskne **CTRL-C**
- generuje se přerušení (jako u každého stisknutí klávesy)
- ovladač rozpozná, že jde o kombinaci generující signál, a odešle signál **SIGINT** procesu v popředí
- když je proces naplánován jako běžící při návratu do uživatelského módu anebo byl-li běžící při návratu z přerušení, proces najde signál

Signály a vícevláknové procesy

Obsluha signálů v jednovláknových programech je přímočará; signály jsou vždy doručeny procesu. Doručení signálů je však komplikovanější u vícevláknových programů, kde proces může mít několik vláken. Kam se má potom signál doručit?

Obecně existují následující možnosti:

1. Doručit signál vláknu, ke kterému se signál vztahuje.
2. Doručit signál každému vláknu v procesu.
3. Doručit signál určitým vláknům v procesu.
4. Pověřit jedno konkrétní vlákno, aby přijímalo všechny signály pro proces.

Metoda pro doručení signálu záleží na typu generovaného signálu. Například synchronní signály je třeba doručit vláknu, které ten signál způsobilo a už ne ostatním vláknům procesu. U asynchronních signálů to ale není tak jasný. Některé asynchronní signály, třeba signál, který ukončuje proces (např. <control><C>), by měly být poslány všem vláknům.

Většina vícevláknových verzí UNIXu umožňuje vláknu určit, které signály bude přijímat a které blokovat. V některých případech proto může být asynchronní signál doručen pouze těm vláknům, která jej neblokují. Protože však signály musí být obslouženy pouze jednou, signál je obvykle doručen prvnímu nalezenému vláknu, které jej neblokuje.

Standardní UNIXová funkce pro doručení signálu je `kill (aid_t aid, int signal);` uvádíme zde proces (aid), kterému bude příslušný signál doručen. POSIX Pthreads taky ještě poskytují funkci `pthread_kill(pthread_t tid, int signal`, která umožňuje doručit signál konkrétnímu vláknu (tid.)

Windows APC

Ačkoliv Windows neposkytuje přímo podporu signálů, mohou být emulovány pomocí **asynchronních volání procedur** - ***asynchronous procedure calls*** **(APCs).** APC umožňuje uživatelskému vláknu (user thread) uvést funkci, která má být zavolána, když tomu user threadu přijde oznámení o určité události. Jak už název napovídá, APC je zhruba to samý co asynchronní signál v UNIXu. Zatímco se však UNIX musí potýkat s pořešením signálů ve vícevláknovém prostředí, možnost APC je přímočařejší, protože APC je doručeno konkrétnímu vláknu a ne procesu.

Příklady signálů

SIGINT – signál pro přerušení od terminálu (stisknutím Ctrl+C)

SIGQUIT – ukončí proces + core dump (záznam stavu pracovní paměti procesu do souboru, často při abnormálním ukončení)

SIGKILL – zabije proces, nemůže být zachycen ani ignorován + proces nemůže po jeho přijetí provést úklid

Proč může dojít ke zpoždění vyřízení signálu?

- signál je nevyřízen (*pending*), byl-li odeslán, ale nebyl přijat
- jenom jeden signál každého typu může být nevyřízen

Reakci na signál vykonává proces, kterému je signál zaslán, včetně ukončení procesu. To znamená, že musí být aspoň plánován stát se běžícím

Má-li nízkou prioritu, může mezi odesláním signálu a jeho přijetím, kdy se vykoná odpovídající akce, uplynout dosti dlouhá doba

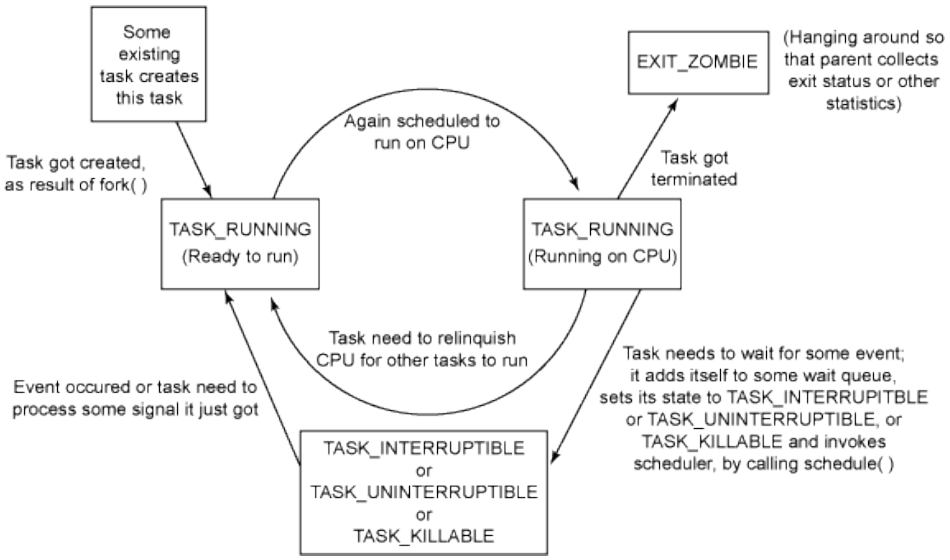
Další prodlení může způsobit, je-li proces v čase odeslání signálu ve stavu **zastaven** nebo **spící**

Co se má stát, když je odeslán signál spícímu procesu?

činnost jádra záleží na tom, proč proces přešel do stavu **spící**

- čeká-li **na událost, která zakrátko nastane**, např. dokončení diskové V/V operace, je spící v kategorii nepřerušitelný a signál je pouze zaznačen jako nevyřízen
- čeká-li **na událost, o které nevíme kdy nastane** nebo dokonce nemusí nastat vůbec, např. skončení potomka, vstup z terminálu, je spící v kategorii přerušitelný, je jádrem vzbuzen a přejde do stavu připraven

Linux nemá stav spící, ale stavy úloha_přerušitelná (***TASK_INTERRUPTIBLE***) a úloha_nepřerušitelná (***TASK_UNINTERRUPTIBLE***)



Spolehlivé a nespolehlivé signály

Nespolehlivé = můžou se ztratit; proces někdy zachytí signál, jindy ho ztratí; kvůli resetu signálu na implicitní akci.

Neumožňují ignorovat signál v daný moment, ale pamatovat si, že nastal, aby jej šlo blokovat a zpracovat později (třeba po skončení důležitého výpočtu).

Pokud to chceme emulovat (nastavit nějakou akci pro následující signál), musíme ji opět instalovat (= volat funkci `signal(sig, function)`) - vznik nedeterminismu, někdy se to stihne, někdy ne

Spolehlivé = perzistentní obslužné funkce signálů; blokování signálu, např. při obsluze signálů → nevznikne hnízdění

3.1.10. Meziprocesová komunikace, roury, sdílená paměť, semafory, zasílání zpráv – implementace

Řada aplikací se skládá z mnoha spolupracujících procesů. Ty mezi sebou komunikuje a sdílejí informace.

Jádro OS musí poskytovat mechanismy, které to umožní – nazýváme je *prostředky meziprocesové komunikace*. Jejich **účelem je**:

- Přenos údajů
- Sdílení dat
- Oznámení vzniku událostí
- Sdílení prostředků
- Sledování a sdílení běhu procesu, např. při ladění programu

1. **Signály**
   - Umožňují oznámení procesům o asynchronní události
   - Viz otázka *„Signály“*
2. **Roury (pipes)**
   - Zápis dat na konec roury, čtení dat od začátku

   a. Nepojmenované roury

   - Vytvoření systémovým voláním **`pipe()`** – vrací dva deskriptory, jeden pro čtení, druhý pro zápis
   - Při vytváření procesů jsou deskriptory roury děděné
   - do roury může zapisovat i číst z ní více procesů, data jsou čtena v pořadí, v jakém byla zapsána
   - Procesy mohou **komunikovat** prostřednictvím roury, **pokud byla vytvořena společným předchůdcem**
   - Po skončení všech předchůdců přestává roura existovat

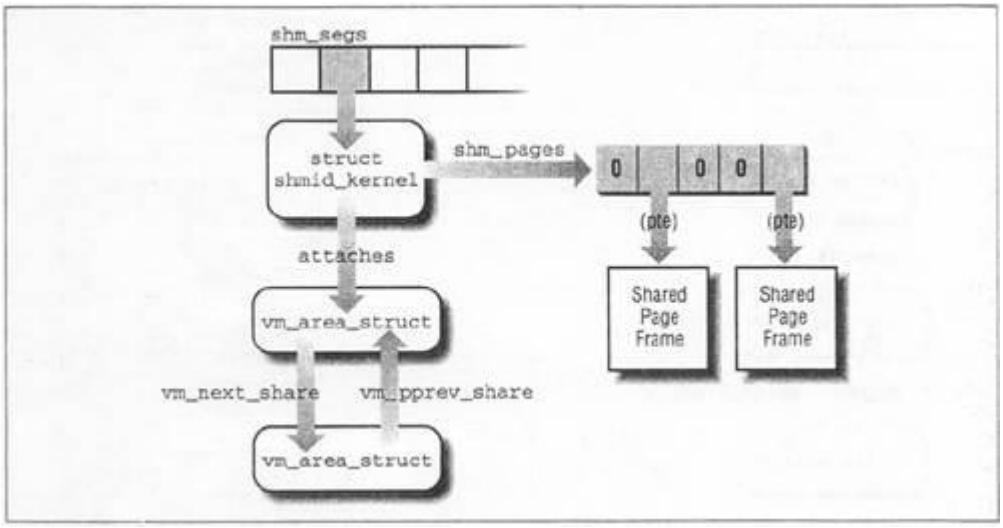
   b. Pojmenované roury, FIFO roury

   - **Perzistentní,** existují jako soubory, i když je nepoužívají žádné procesy
   - FIFO musí být explicitně zrušen, jako obyč soubory – **`unlink`**
   - Oproti obyč souborům jsou přečtená data odstraněna a z pohledu komunikace mají stejnou sémantiku jako nepojmenované roury
   - Vytvoření FIFO souboru: **`mknode(cesta, mód, zařízení)`**, **`mkfifo(cesta, mód)`**
     - Mód = obvyklá oprávnění
     - Zařízení = pro vytváření speciálních souborů pro zařízení
   - FIFO jsou pak otevřena sys voláním **`open()`** – vrací deskriptor souboru
   - do FIFO jde zapisovat sys voláním **`write()`** nebo z něj číst sys voláním **`read()`**

   Následující způsoby mají podobnou implementaci – jsou identifikovány IPC (inter-process comunication) klíčem (obdoba cesty k souboru) a zpřístupňovány identifikátory IPC (obdoba deskriptoru souboru) – ty nejsou vázány na proces a nemění se v průběhu života objektu; získání identifikátoru IPC voláním `shmget(), semget(), msgget()`

3. **Sdílená paměť**
   - Oblast paměti, která je sdílená více procesy
   - Sdílená data jsou v paměti fyzicky uložena jen jednou, procesy je mají namapované do vlastního virtuálního paměťového prostoru (např. Používají DLL)
   - Proces ji **vytvoří/získá** voláním **`shmid = shmget(klíč, velikost, příznak);`**

- Shared memory id, shared memory get
- Proces **připojí oblast** na virtuální adresu voláním:
  `adr = shmat(shmid, shmadr, shmpříznak);`
  - „shared memory at“
  - `shmadr` je návrh adresy pro připojení oblasti; pokud je `shmadr` nula, jádro adresu vybere
  - skutečná adresa je návratová hodnota
- **Odpojení** oblasti:
  - `shmdt(shmaddr);`



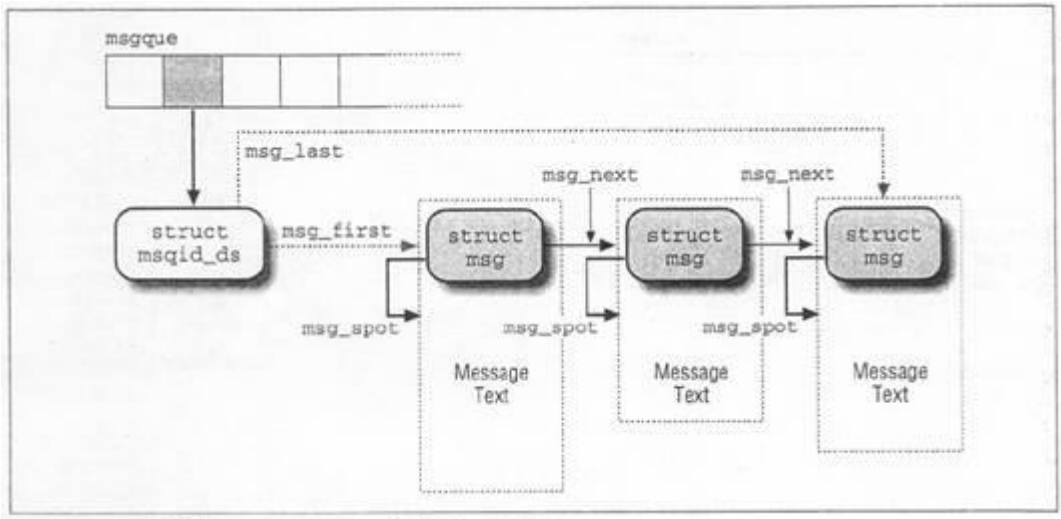
4. **Semafory** - synchronizační primitivum s celočíselným čítačem a metodami P() a V()
   - **`semid = semget(klíč, počet, příznak)`**
     - vrátí identifikátor pole semaforů o velikost počet
   - **`stav = semop(semid, sops, nsops)`**
     - atomicky vykoná operace nad polem semaforů
     - `sops` je ukazatel na pole s `nsops` prvky typu `sembuf`

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

5. **Zasílání zpráv**
   - Procesy komunikují prostřednictvím zpráv
   - Zpráva vytvořená procesem je zaslána do fronty zpráv, je tam, dokud ji jiný proces nepřečte
   - Zpráva obsahuje 32bitový typ a data zprávy
   - Typ zprávy umožňuje selektivní výběr zpráv z fronty
   - Synchronní/asynchronní, blokující/neblokující
   - Proces **získá/vytvoří zprávu** voláním:
     - **`msgqid = msgget(klíč, příznak);`**
     - „message queue id“
   - Zpráva se **uloží do fronty** voláním:
     - **`msgsnd(msgqid, msgp, počet, příznak)`**
     - msgp = pointer na zprávu obsahující typ zprávy, který je následován
     - počet = velikost zprávy včetně typu v bytech
   - zprávy jsou ve frontě v pořadí jejich příchodu
   - **výběr zpráv** z fronty voláním:

- **počet = msgrcv(msgqid, msgp, maxpct, msgtyp, příznak)**
- je-li čtená zpráva delší než maxpct, je oseknutá
- msgtyp = 0 → vrátí se první zpráva z fronty
- msgtyp kladný → vrátí první zprávu typu msgtyp
- msgtyp záporný → vrátí se první zpráva nejnižšího typu než je abs hodnota `msgtyp`

- Na Windows:
  - SendMessage(HWND okna, int message, WPARAM. LPARAM) - blokující, čeká na vyzvednutí zprávy
  - SendNotifyMessage/PostMessage - neblokující
  - Každé okno má svou smyčku zpráv while(GetMessage(..)) - blokující
  - PeekMessage - neblokující



IPC message queue data structures

3.1.11. Synchronizace v jádře, symetrikcý multiprocesor, atomické operace

Problém kritické sekce - uvedení dat do nekonzistentního stavu nebo přerušení v okamžiku výpočtu

Kód jádrových funkcí se vykonává:

- Při systémovém volání
- Při obsluze výjimek a přerušení

Synchronizace v jádře

- Atomické operace
  - Atomická instrukce, jen jeden procesor aktivní v době jejího vykonání
- Spinlock
  - Klasický spinlock, u uniprocesoru řešen jen zákazem preempce jádra, uvnitř spinlocku je preempce jádra též zakázána, u multiprocesoru v čekací smyčce je preemplce jádra povolena a kernel tak místo toho může naplánovat jiný proces
- Read/Write spinlock
  - Oproti klasickému spinlocku zde readeři mohou číst neustále, writer musí ale čekat až všichni vše přečtou, pak to zamknout a zapsat; to samé u readerů, kteří všichni musejí čekat až jim to writer zas uvolní
- Seqlock

  When using read/write spin locks, requests issued by kernel control paths to perform a read_lock or a write_lock operation have the same priority: readers must wait until the writer has finished and, similarly, a writer must wait until all readers have finished.
  Seqlocks introduced in Linux 2.6 are similar to read/write spin locks, except that they give
  a much higher priority to writers:
  in fact a writer is allowed to proceed even when readers are active. The good part of this strategy is that a writer never waits (unless another writer is active);
  the bad part is that a reader may sometimes be forced to read the same data several times until it gets a valid copy.
  Each seqlock is a seqlock_t structure consisting of two fields:
  a lock field of type spinlock_t and an integer sequence field.
  This second field plays the role of a sequence counter. Each reader must read this sequence counter twice, before and after reading the data, and check whether the two values coincide.
  In the opposite case, a new writer has become active and has increased the sequence counter, thus implicitly telling the reader that the data just read is not valid.
- Kernel Semaphore

  However, whenever a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It becomes runnable again when the resource is released.
  Therefore, kernel semaphores can be acquired only by functions that are allowed to sleep:
  interrupt handlers and deferrable functions cannot use them.

- BKL (Big Kernel Lock)
- http://www.ibm.com/developerworks/linux/library/l-linux-synchronization/index.html

Systémová volání sdílejí jednotlivé struktury v jádře OS – snaha o zabránění soutěže nad systémovými daty prostřednictvím:

Nepreemptivnost procesů jádra

Jádrový proces nemůže být přerušen a nahrazen procesem s vyšší prioritou (pokud samotný proces neudělá yield). I tak ale může být přerušen výjimkou nebo přerušením, a obsluha přerušení může být zase přerušena přerušením.

Problém blokujících operací – možnost deadlocku -> uvolnit procesor před blokující operací

Atomické operace

Instrukce provádějící celou v jedné instrukci (inc, dec...)

Nemůže vzniknout nekonzistence dat – není přerušení

Pouze pro jednoprocesorový stroj, pro multiprocesorový je speciální instrukce zamykající sběrnici

Zákaz přerušení

Kritická oblast začíná zákazem přerušení a končí obnovením přerušení

KS musí být krátká a rychlá -> blokování I/O a procesoru

Velmi nebezpečné a nefunguje na víceprocesorových systémech

Nesmí se čekat na oznámení události přerušením

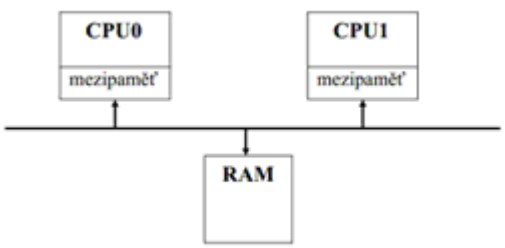
Zamykání

Pro víceprocesorové systémy velmi nákladné (režie)

Semafory v jádru, spinlock

Zabránění uvíznutí kdy více procesů žádá více zdrojů (přiřazení A->1, 2 B-> 2, 1) – prostředky se VŽDY alokují v pořadí adres semaforů

Symetrický multiprocesing SMP



- Procesory (identické) a sdílená hlavní pamět jsou připojeny ke společné sběrnici, tu je nutno zamykat pro výhradní přístup jednoho procesoru (aby nemohly 2 procesy zapisovat na stejné místo v paměti naráz).
- Klasické atomické instrukce (dec, inc) nejsou atomické, nutná speciální instrukce zamykající sběrnici
- Nutná synchronizace mezipaměti (cache) procesorů: když procesor modifikuje svou mezipamět, musí kontrolovat jestli stejná data nejsou v mezipaměti jiného procesoru a když ano, musí je aktualizovat nebo zneplatnit.
- Úloha může být zpracovávána postupně různými procesory, je umístěna ve sdílené hlavní paměti
- Na více procesorech naráz mohou být současně provedena systémová volání
- Obsluha přerušení – obsluhuje procesor, který má k dispozici nejvíce volných prostředků (obsluhuje proces s nejnižší prioritou)

Nutno řešit pokročilým způsobem synchronizaci jádra – semafory + spinlock

spinlock – efektivní pro multiprocesory – blokující skončí na jiném procesoru a nemusí se nic přepínat

spinlock nesmí chránit data přístupná při obsluze přerušení - přerušení modifikuje data – problém řešený zákazem přerušení v KS

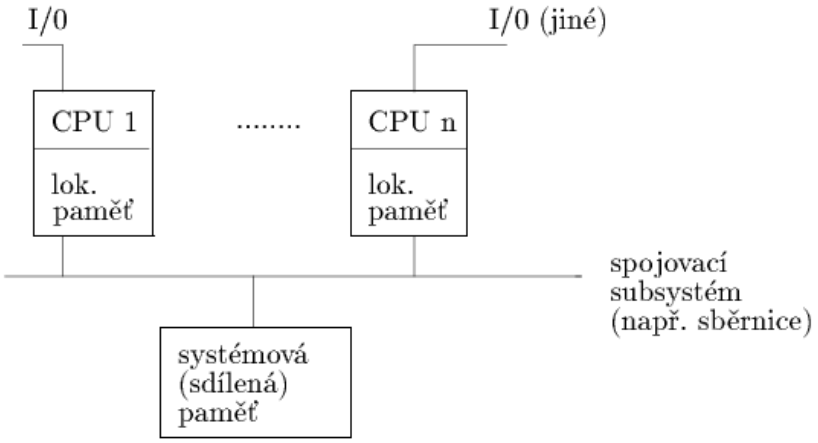
Procesy mohou využívat zámky dvěma způsoby:

sdílené - pro čtení (lepší propustnost)

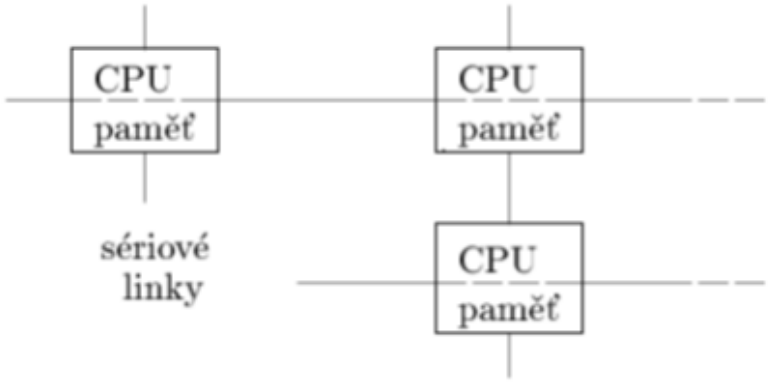
výhradní – pro zápis

Kromě SMP existují další technologie pracující na podobném principu:

- **asymetrické multiprocesory ASMP** - navíc vlastní lokální paměti a I/O připojení u procesorů, které tak mohou mít různé instrukční sady – každý procesor má speciální funkce – nelze libovolný proces na libovolný CPU



- **multiprocesor s distribuovanou pamětí** - procesory mají jen lokální paměti, není sdílená paměť, komunikace zasíláním zpráv, topologie 2D mřížky nebo N-rozměrné krychle, výhoda odstranění společné sběrnice jako úzkého hrdla



Atomické operace

- operace vykonávaná procesorem v jedné instrukci (inc, dec)
- problém u SMP – instrukce již není atomická v rámci procesů – nutno navíc uzamknout sběrnici instrukcí lock*
- atomická instrukce pro podporu semaforů a spinlocku – TSL (TestAndSetLock)

```
mov edx, DWORD(-1) //-1 zamčeno
//otestujeme stav zámku, 0 = odemčeno
spin: mov eax, [lockState]
test eax, eax
jnz spin
//zkusíme ho zamknout s -1
lock cmpxchg [lockState], edx
//nepředběhl nás jiný procesor?
//původní lockState je v eax
test eax, eax
jnz spin

Neoptimální verze:
while(!acquire_lock())
{ Sleep( 0 ); }
do_work();
release_lock();


spin_lock:
    TSL R, lock ;; atomicky provede R:=lock a lock:=1
    CMP R, 0 ;; byla v lock 0?
    JNE spin_lock ;; pokud nebyla (R<>0), byl zámek nastaven ->
    cyklus
    RET ;; návrat, tj. vstup do KS

spin_unlock:
    LD lock, 0 ;; ulož hodnotu 0 do lock
    RET
```

3.1.12. Virtuální souborový systém, Extended File Systém

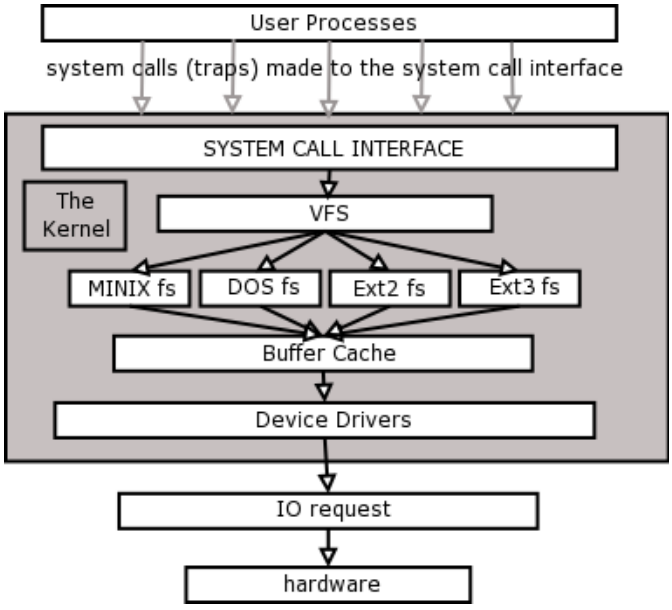
Abstrakční vrstva(vrstva abstraktního API) mezi OS a konkrétním souborovým systémem.

Na jedné straně je univerzální rozhraní pro operační systém a na druhé straně konkrétní ovladače pro všechny podporované souborové systémy

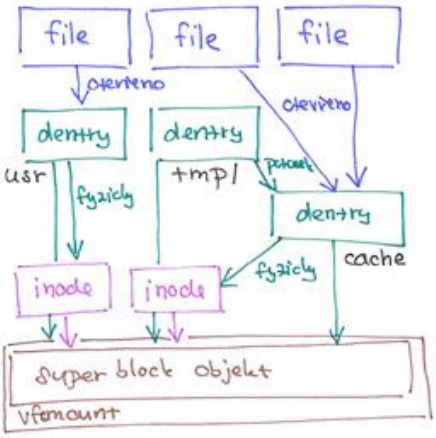
Lze jednoduše přidávat podporu pro různé FS jenom novým ovladačem pro VFS bez potřeby upravovat jádro.

Umožňuje zajistit dopřednou kompatibilitu OS s novými FS

rozhraní stejné pro všechny FS – aplikace nemusí znát podrobnosti o FS, jen využívá poskytnuté rozhraní



Struktura VFS Linux



File

- struktura obsahující informace o otevřeném souboru (procesem)

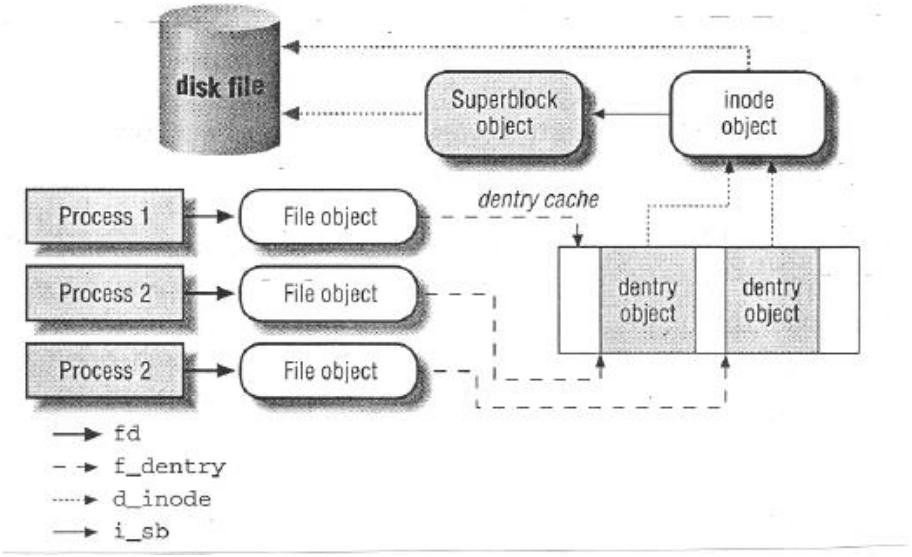
- ukazatel na dentry = otevřený soubor (spojuje i-node – jeho číslo se jménem souboru)
- mód otevření, pozice v souboru, operace (read/write/seek)
- počet procesů sdílejících File (příkazem *dup*)

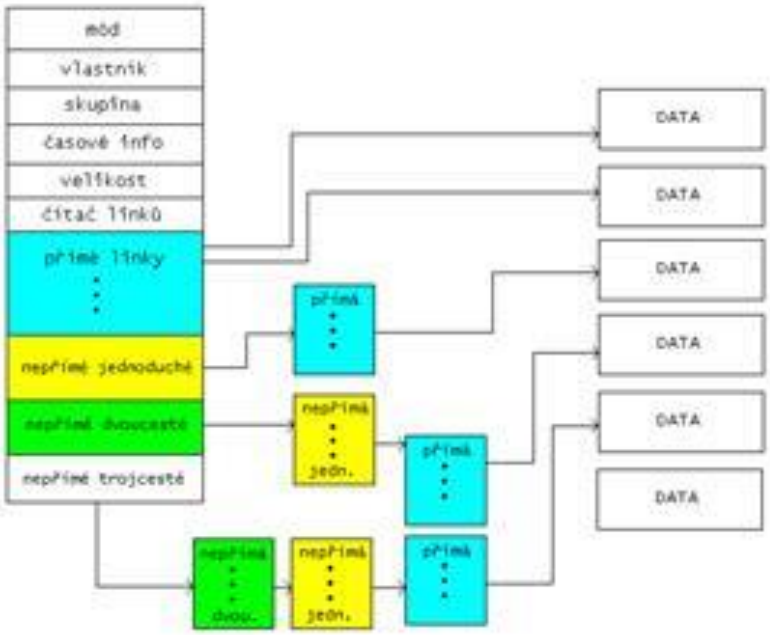
Dentry

- položka hierarchické struktury, umožňuje vytvářet stromy v moderních FS
- rodič (root nemá, je jen jeden)
- potomci (adresáře + soubory)
- pouze virtuální (neexistuje na disku)
- sdružený inode objekt se samotnými daty a informacemi
- využití hash tabulky pro urychlení přístupu k položkám

Inode

- obsahuje veškeré informace pro VFS o souboru (souborem je i adresář, blokové zařízení)
  - Jen metadata, na datové bloky odkazuje
- typ souboru, vlastník, skupina, práva, ACL
- velikost souboru, data přidání, modifikace, mazání
- počet odkazů (soft/hard link)
- odkazy na i-uzly (jeho číslo) – přímé/nepřímé
- stav i-uzlu (dirty/clean)
- pro prohledávání je inode ve zřetězených seznamech: nepoužívané, používané, dirty
-





Superblok

- globální informace o fyzickém FS
- kořenový adresář, typ FS, seznamy i-uzlů
- všechny superbloky jsou ve spojovém seznamu

Přípojení FS

- = registrace + připojení (začlenění do souborového systému)
- registrace zavedením modulu – OS má k dispozici funkce a informace pro připojení FS → body připojení = adresář, kde je FS připojen
- kořenový souborový systém = kořen souborového systému OS
- OS přečte superblock z disku a připojí kořenový adresář do cílového bodu připojení → původní adresář je zakrytý novým FS
- přidá do seznamu připojených souborových systémů

Odpojení FS:

- nelze odpojit kořenový FS
- nelze odpojit používaný FS (jeho soubory)
- odpojení způsobí flush na disk (zápis dirty souborů)

EXT (Extended File System)

- Vychází z UFS, navržen a vytvořen pro Linux

- EXT, EXT2 až EXT4
- Různé typy souborů: obyčený soubor, blokové a znakové zařízení
- Pevné odkazy, symbolické odkazy

Ext3

- žurnálovací (3 způsoby žurnálování)
- aktivně předchází fragemntaci (nelze jej defragmentovat když je připojený)
- bezpečnější mazání (složitější obnova smazaných souborů)
- zpětně kompatibilní s ext2

Ext4

- Zpětně kompatibilní s ext3 (fork a přidání funkčnosti) krom extentů
- Extent: nahrazení tradičního blokového rozdělení, zmenšuje fragmentaci a zlepšuje výkon při práci s velkými soubory (alokační jednotka o velikosti až 128 MB) – při použití nelze mountnout jako ext3
- Delayed allocation – alokování až při zápisu (zmenšuje fragmentaci)
- Rychlejší kontrola (přeskakování nealokovaného místa)
- Nanosekundový timestamp

FAT

Viz Kapitola/Záložka Bakalář

NTFS

3.1.13. Správa V/V zařízení

Též I/O zařízení nebo periferie.
V/V (input/output) zařízení je hw zařízení které zprostředkuje kontakt počítače s okolím,

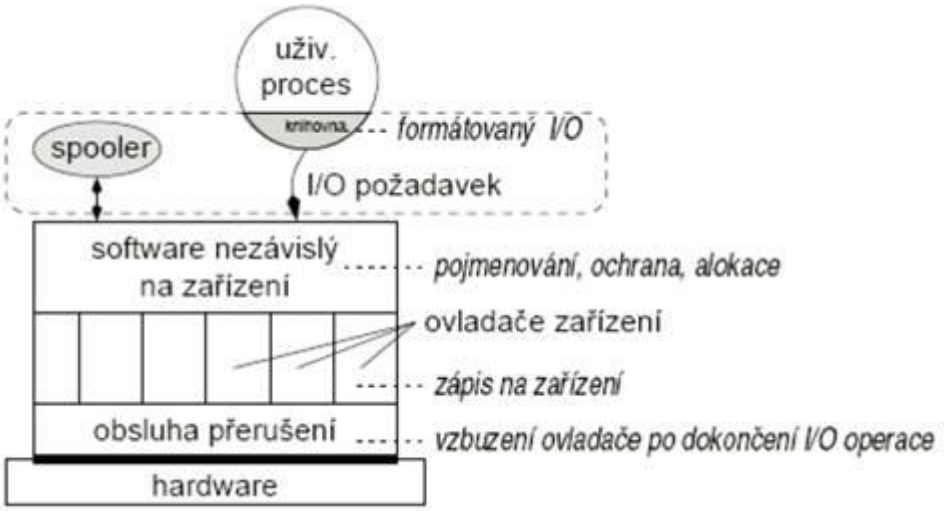
- Klávesnice/myš, čtečka kódů
- Obrazovka, tiskárna
  případně pseudo zařízení
- /dev/null
- /dev/random - generátor náhodných čísel

Principy I/O Software

- Všechny I/O instrukce jsou privilegované instrukce, a tedy probíhají v režimu jádra, operační system pak musí poskytovat prostředky pro provádění I/O

The I/O system se skládá z:

- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices



1.-3. – režim jádra:

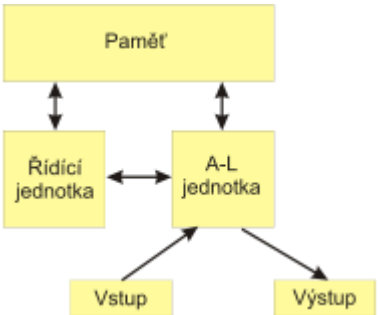
1. **Obsluha přerušení**
   - Řadič vyvolá přerušení ve chvíli **dokončení** I/O požadavku
   - Snaha, aby se přerušením nemusely zabývat vyšší vrstvy
   - Ovladač zadá I/O požadavek, **usne** (p(sem))
   - Po příchodu přerušení ho obsluha přerušení **vzbudí** (v)
   - Časově kritická obsluha přerušení – co nejkratší
2. **Ovladače zařízení**
   - Obsahují veškerý kód závislý na I/O zařízení = způsob komunikace s řadičem zařízení + zná detaily (ví o sektorech a stopách na disku, pohybech diskového raménka,…)
   - Ovládá všechna zařízení daného druhu nebo třídu podobných zařízení (ovladač SCSI disků → všechny SCSI disky
   - Ovladači je předán příkaz vyšší vrstvou (zapiš data do bloku n) → nový požadavek zařazen do fronty (může ještě být obsluhován předchozí) → ovladač zadá příkazy řadiči (až přijde požadavek na řadu; např. Přečtení sektoru) → zablokuje se do vykonávání požadavku (při rychlých operacích jako zápis do registru se neblokuje) → vzbuzení obsluhou přerušení (dokončení operace) + kontrola, zda nenastala chyba → pokud OK –

předá výskledek (status + data) vyšší vrstvě → bere další požadavky ve frontě (1 vybere a spustí)
- Ovladače často vytvářeny výrobci HW (dobře def. rozhraní mezi OS a ovladači)
- Ovladače podobných zařízení – stejná rozhraní (síťové karty, zvukové karty,…)

3. **SW vrstva OS nezávislá na zařízení**
   - Poskytuje I/O funkce společné pro všechna zařízení daného druhu (např. Společné fce pro všechna bloková zařízení)
   - Definuje rozhraní s ovladači
   - Poskytuje jednotné rozhraní uživatelskému SW
   - **Funkce:**
     1. Pojmenování zařízení (LPT1 x /dev/lp0)
     2. Ochrana zařízení (přístupová práva)
     3. Alokace a uvolnění (v 1 chvíli použitelná jen 1 procesem – tiskárna, plotter, …)
     4. Vyrovnávací paměti (blok. zařízení – bloky pevné délky, pomalá zařízení – čtení/zápis s využitím bufferu)
     5. Hlášení chyb
     6. Jednotná velikost bloku pro bloková zařízení
   - V Linuxu se zařízení jeví jako objekty v souborovém systému
4. **I/O SW v uživatelském režimu**
   - Programátor používá v programech **I/O funkce** nebo **příkazy jazyka**
   - `printf` v C, `writeln` v Pascalu | knihovny sestavené s programem | formátování – printf | často vlastní vyrovnávací paměť na jeden blok
   - **Spolling –** impl. pomocí procesů běžících v uživatelském režimu, = způsob obsluhy vyhrazených I/O zařízení
   - *příklad spoolingu*: přístup k tiskárně má pouze 1 speciální process – daemon lpd (má přístup do spooling directory, odkud vezme připravený soubor k vytisknutí, vytiskne ho a zruší)

Procesor komunikuje s V/V zařízeními pomocí registrů (můžou sloužit jako vyrovnávací paměť)

- **Izolované (port-mapped)**: Přístupné pomocí speciálních instrukcí
- **Vnitřní (IO mapped)**: namapovaná paměť (namapuju si CD-ROMku do paměti),
  - adresované jako paměť,
  - přístupné pomocí běžných paměťových instrukcí
  - např. DMA, velmi rychlé protože data nemusí do paměti přenášet procesor
  - Porušuje Von Neumannovu architekturu:



V/V zařízení si vyžádá obsluhu procesorem pomocí přerušení – 1 bitový kanál, který pouze upozorňuje procesor, že je třeba věnovat se V/V

Využití V/V zařízení v programu vyžaduje systémová volání (procesor musí běžet v privilegovaném režimu, do kterého se uživatelský program nesmí přepnout).

- Speciální konstrukce jazyka – např. proudy v C (stdio)

Soubor typu zařízení

Unixové OS mapují V/V zařízení do souborového systému (protože se snaží do soub. Systému mapovat úplně všechno)

Soubor se vytváří systémovým voláním **`mknod()`** s parametry

- Jméno
- Druh – blokové, znakové, roura
- Hlavní číslo – identifikuje skupinu (USB)
- Vedlejší číslo – konkrétní zařízení (třetí port zprava)

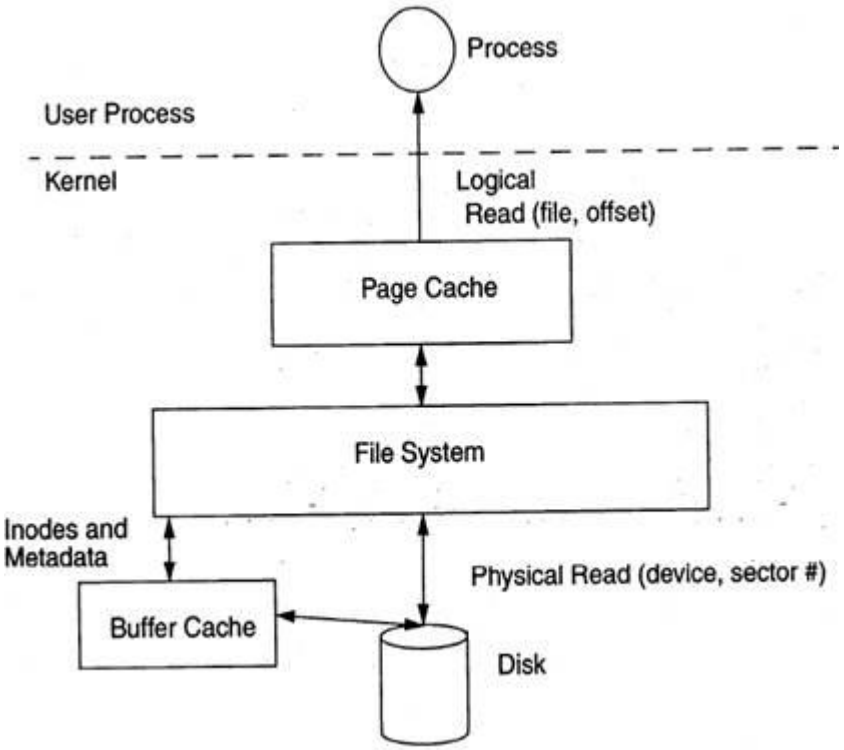
- **Bloková zařízení:** je potřeba zapisovat a číst vždy celý blok určené velikosti (video, zvuková karta)
  - **HDD**
  - Obsluhují se V/V operacemi s mezipamětí nebo stránkovými V/V operacemi
  - **Libovolný (random) přístup** k blokům zařízení
- **Znaková zařízení:** zapsán a čten je vždy jeden znak (terminál, COM a LPT porty)
  - **Klávesnice, myš**
  - Protože jde o jeden znak, nemají mezipaměť → data jsou přenášena primo do uživatelského adresového prostoru
  - Mechanizmus proudu – duplexní zpracování
  - **Sekvenční přístup** k datům

Síťová zařízení nemají svůj soubor.

Obsluha blokových zařízení

Přístup k blokovým zařízením musí být řádně plánován a synchronizován. K tomu účelu obvykle slouží modul jádra zvaný *I/O Scheduler* (*I/O plánovač*). Tento modul spravuje fronty požadavků na bloková zařízení a s použitím těchto front posílá požadavky ovladačům zařízení.

5. V/V operace s vyrovnávací pamětí
   - Ve vyrovnávací paměti je uložen diskový blok
   - Vyrovnávací paměti bloků jsou organizovány v mezipaměti vyrovnávacích pamětí bloků (*block buffer cache*) pomocí hlaviček bloků
6. Stránkové V/V operace
   - Data se přenášejí po blocích, adresový prostor procesu je množina stránek
   - V/V operace pro obyčejné soubory jsou vykonávány a ukládány po stránkách v mezipaměti stránek


Process
User Process
Kernel
Logical
Read (file, offset)
Page Cache
File System
Inodes and
Metadata
Physical Read (device, sector #)
Buffer Cache
Disk

3.2. PPR

Základy

Základní pojmy

- Program, programový kód
  - statický popis výpočetního postupu, tj. co bude počítač provádět
  - nemá stav

- Assembler
  - Překladač do strojového kódu cílového procesoru
  - I když formálně nesprávně, používá se i k označení jazyka symbolických adres, ve kterém budou některé ilustrativní příklady
    - x86/x86-64
  - dokud neuvidíte, do čeho vlastně procesor nutíte, neznáte skutečné náklady kódu ve vyšším jazyku

- Vlákno (fiber, thread, task)
  - vykonávaný programový kód
  - tj. aktivita v čase, která má svůj stav, který je určen
    - aktuálním místem v programu (CS:RIP),
    - obsahem registrů procesoru
    - a obsahem zpracovávaných dat
  - vlákno v jednom časovém okamžiku běží pouze na jednom procesoru

- Proces
  - Dynamická kolekce vláken => má stav daný vlákny
  - Vždy obsahuje alespoň jedno vlákno, které vytvořil operační systém/interpret (např. Java)
  - Vlastní prostředky, které operační systém/interpret přidělil důsledkem činnosti některého vlákna
  - Vlákna jednoho procesu mohou běžet současně, je-li k dispozici více než jeden procesor
- Distribuovaná aplikace
  - Několik spolupracujících procesů
  - Obvykle jsou distribuovány na uzly počítačové sítě
    - Jeden uzel má *n* procesorů
  - V extrémním případě mohou být na jednom uzlu
    - Dříve, kdy ještě platilo 1 proces = 1 vlákno, se spouštělo několik instancí jednoho programu

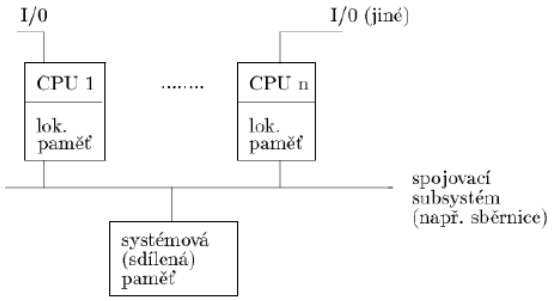
- Paralelní výpočet
  - Buď vícevláknový program,
  - Nebo distribuovaná aplikace
  - Výpočet je dynamicky strukturován na procesy a vlákna

- Paralelní program
  - Kód je staticky strukturován na podprogramy vláken
  - Má deklarovaná sdílená data

- Interakce
  - Spolupráce na úrovni vláken uvnitř procesů
  - Spolupráce na úrovni procesů distribuované aplikace
  - Výměna informací

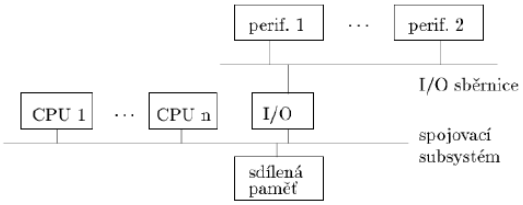
- Synchronizace
  - Forma interakce
  - Zajištění správné návaznosti operací

Asymetrický multiprocesor (ASMP)



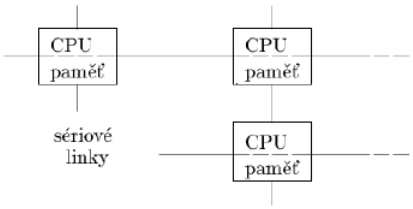
- Kromě sdílené paměti, každý procesor má vlastní lokální paměť a vlastní připojení I/O
- Každý procesor může mít jinou instrukční sadu
  - v extrémním případě na každém z nich běží jiný OS
- OS nemůže alokovat libovolný proces na libovolný procesor
- Jednotlivé procesory mohou vykonávat specifické úkoly
- Sdílená paměť může obsahovat pouze minimum dat
  - Minimalizace úzkého hrdla sběrnice

Symetrický multiprocesor (SMP)



- Všechny procesory jsou identické
- Vlákno může být alokováno na libovolný procesor
- Pro plánovač OS je to jednodušší, než kdyby se procesory významně lišily
- Procesory sdílejí jednu paměť – úzké hrdlo je sběrnice
  - Řešením je spojit jednotlivé procesy přímo – tzv. point-to-point

Multiprocesor s distribuovanou pamětí



- Každý procesor má svou lokální paměť
- Sdílná paměť není
- Každý uzel je v podstatě počítač s omezeným I/O
- Komunikuje se zasíláním zpráv
- Typické topologie jsou 2D cyklicky uzavřené mřížky, nebo n-rozměrná krychle
- Výhodou je odstranění jedné sběrnice jako úzkého hrdla
- Nevýhodou je však malá univerzálnost – výkonnost záleží na způsobu alokace procesů na uzly a použitém komunikačním schématu
- Vhodné např. pro tzv. pipe-lines
- Transputery - Occam

Distribuovaný systém

- De facto počítačová síť, která se může tvářit jako jeden stroj
- Dědí problémy multiprocesoru s distribuovanou pamětí
- Každý uzel je plnohodnotný počítač, na kterém může běžet několik procesů zároveň
- Např. cluster
  - Počítače propojené velmi rychlou lokální sítí

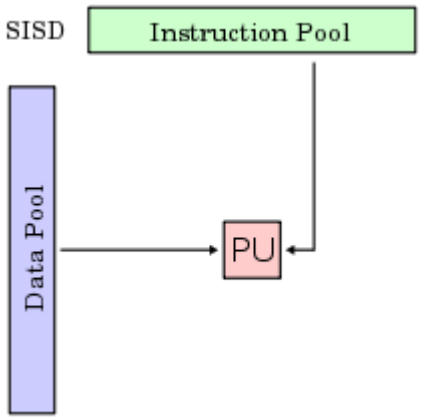
Flynnova taxonomie

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

3.2.1. Modely SIMD, SPMD. MPSD, MPMD.

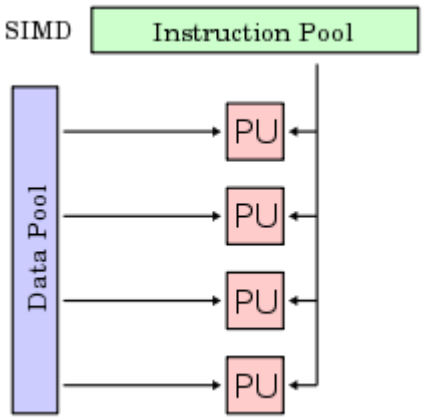
SISD (Single Instruction Single Data)

Sekvenční výpočet, žádný paralelismus, např. MSDOS



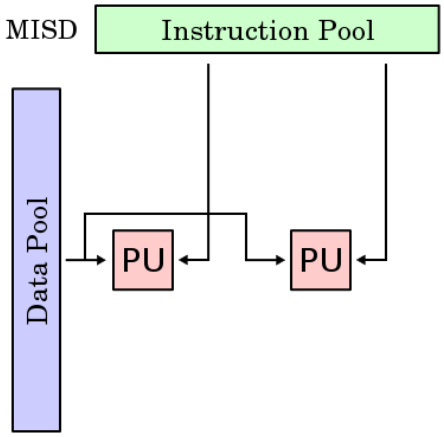
SIMD (Single Instruction Multiple Data)

Datový paralelismus, např. vektorový paralelní počítač, maticové a vektorové výpočty na GPU (operace pro úpravu kontrastu v obrázku apod.), vektorová instrukce = jedna instrukce zpracuje několik dat najednou



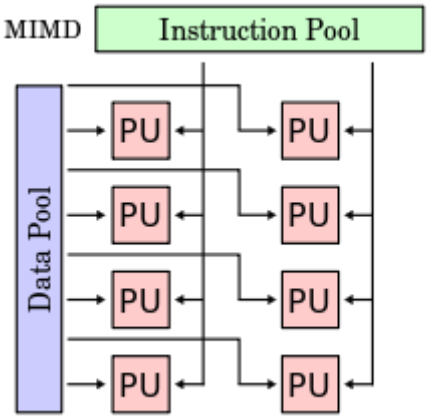
MISD (Multiple Instruction Single Data) = MPSD (Multiple Program Single Data)

Používáno pro výpočty odolné proti poruchám, několik systémů zpracovává ta samá data a musejí se shodnout na výsledku – např. letadla. Používá se také v pipeline architektuře, kde několik procesů zpracovává data v jednom datovém proudu, např. montážní linka. Urychlení u pipeline je N (rovno počtu fází).



MIMD (Multiple Instruction Multiple Data)

Několik procesorů zároveň vykonává různé instrukce nad několika různými daty. Procesory pracují buď asynchronně. Procesory buď mají sdílenou paměť – o zajištění integrity se stará OS, nebo distribuovanou paměť – má lepší škálovatelnost.



SPMD (Single Program Mutiple Data)

- Několik procesorů autonomně vykonává jeden program nad různými daty. Bod vykonávání programu nemusí být na všech procesorech stejný, označuje se též jako dekompozice dat.
- Používá se ke zpracování velkých objemů dat, k procesů běží podle stejného programu a zpracovává strukturou stejné, ale hodnotově různé části dat.
- Příklady použití: Monte Carlo, iterační numerická řešení

MPMD (Multiple Program Multiple Data)

- Několik procesorů autonomně vykonává více než jeden program nad různými daty. Např. farmer-worker, kdy jeden proces úkoluje ostatní. Nemusí jít nutně pouze o urychlení výpočtu – např. distribuované simulace, systém spolupracujících komponent.

3.2.2. **Paralelizace cyklů.**

[3b_shared_spmd.pdf]

- Lze provést dekompozici dat, můžeme cykly paralelizovat a tím urychlit výpočet => datový paralelismus SPMD
- Máme jeden programový kód, ale můžeme ho vykonávat na několika procesorech
- Příklady: nalezení minima/maxima v poli, součet prvků v poli

Rozdělení proměnných v paralelním výpočtu

Pro vykonávání smyček paralelně je třeba určit:

- **Lokální proměnné** – inicializovány uvnitř smyčky
- **Sdílené proměnné –** hdnoty se přenáši mezi jednotlivými iteracemi
  - **Nezávislé** – pokud jsou využívány jen pro čtení, nebo pokud v případě pole jde pouze o jeden prvek, se kterým je pracováno v jedné iteraci
  - **Závislé** – (je třeba synchronizovat) dále se dělí na:
    - **Redukční** – nejprve se čte a pak zapisuje (vše v jedné iteraci) – např. suma
    - **Uzamykané** – může být čtena i zapisována v několika iteracích, může být čtena I zapoisována několikrát po sobě v jedné iteraci , nezáleží na pořadí iterací – např. max
    - **Uspořádané** – správného výsledku je dosaženo pouze thedy jsou-li iterace vykonávány pouze ve stanoveném pořadí

Paralelizaci součtu pole lze provést snadno, protože se v cyklu vyskytuje nejvýše redukční proměnná, kterou stačí jen ošetřit mutexem.

Paralelizace cyklu se závislou uspořádanou proměnnou

S = urychlení = čas_neparalelizovaný/čas_paralelní

E = účinnost = urychlení/počet_cpu

Paralelizace sčítání prefixů (výsledek pole součtů) již nelze provést jednoduchým rozdělením iterací vláknům, protože se v cyklu nachází uspořádaná proměnná.

Uspořádaná proměnná nám brání zapisovat do pole v jiném než určeném pořadí. Proto zavedeme ještě jednu kopii pole. S i-1 pak přistupujeme do pole, do kterého se v cyklu již nezapisuje (zrušeno uspořádání). Poté můžeme přeuspořádat pořadí počítání prefixů do stromu s vrcholem uprostřed sčítané posloupnosti. Jestliže lze paralelizovat výpočet posledního prvku, lze paralelizovat i výpočet ostatních prvků. Pomocí úprav a optimalizací se můžeme dostat až na složitost O(n/m) u paralelní verze.

Loop Unrolling

- Nápověda pro překladač s autovektorizací
- V nejhorším dojde k většímu využití pipelines procesoru

3.2.3. **Amdahlův a Gustafsonův zákon, Karp-Flattova metrika.**

http://homen.vsb.cz/~roz0015/pa/ref/pa_ref.htm#gustavson_low

- Urychlení
  - Speedup
  - Existuje několik způsobů, jak ho vyjádřit, následující je nejrozšířenější/nejznámější způsob
  - Poměr doby výpočtu referenčního algoritmu a porovnávaného algoritmu
    - Např. nejlepšího známého sekvenčního algoritmu a paralelního algoritmu na témže (paralelním) počítači
      - Lze ovšem porovnat i urychlení referenčního sekvenčního algoritmu oproti provedené optimalizaci
      - Anebo porovnat dva různé paralelní algoritmy
  - *S(p) = E(1) / E(p)*
    - >1 znamená urychlení
  - Perfektní urychlení
    - Poměr je přesně roven počtu procesorů
      - Asi těžko ho dosáhnete :-)

- Účinnost
  - Efficiency
  - Urychlení dělené počtem procesorů
    - Uvažujeme urychlení proti sekvenčnímu algoritmu
  - Sekvenční výpočet trvá 10s, paralelní algoritmus na 4 procesorech trvá 5s
    - Urychlení: 2
    - Účinnost: 0,5

Výpočetní čas = čas strávený výpočtem a čas strávený čekáním

Amdahlův zákon

Amdahlův zákon zkoumá možnost maximálního urychlenéí výpočtu pevně dané úlohy.
Vychází z doby vykonávání sériového programu a z ní vypočítáváme očekávané urychlení.

$$E(p) = G + \frac{H}{p}$$

$$S(p) = \frac{(G+H)}{G+\frac{H}{p}}$$

$$S(p) \leq \frac{1}{f+\frac{1-f}{p}} \leq \frac{1}{f}$$ (f = sekvenčně prováděná část kódu)

E- účinnost
S-speed-up zrychlení

- Amdahlův zákon
  - Nelze dosáhnout perfektního urychlení, protože vždy bude nějaká část výpočtu provedena sekvenčně
  - *G, H* – čas strávený sekvenčně provedeným výpočtem
  - *G*
    - čas strávený vykonáváním neparalelizovatelného kódu, tj. sekvenčně
    - Unavoidably Serial
  - *H*
    - čas strávený vykonáváním paralelizovaného kódu, ale sekvenčně
    - Serialized-Parallel
  - $E(p) = G + H/p$
  - $S(p) = (G+H)/(G+H/p)$
  - Pro velké p platí $S(p) = 1+H/G$
    - To by znamenalo, že dosažení perfektního urychlení je možné, pokud se můžeme vyhnout kódu, který nelze paralelizovat
      - Má vliv na *H*
    - Ale – čím větší počet procesorů, tím např. větší režie jejich komunikace => takže přece jenom nepůjde…
      - Režie OS (plánování, I/O, atd.) se o to také postará
  - Lze ho také zapsat s f (sekvenčně prováděná část kódu) $$S \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Předpokládá, že **velikost problému** zůstane při paralelizaci **stejná**

- Předpokládá tedy problém o konstantní velikosti
- Tzn. že s rostoucím počtem CPU se nemění rozsah úlohy
- Nevyužívá tak plně výpočetní sílu, která je k dispozici při navýšení počtu procesorů

Nelze dosáhnout perfektního urychlení, protože vždy bude nějaká část výpočtu provedena sekvenčně. G, H – čas strávený sekvenčně prováděným výpočtem. G je čas strávený vykonáváním neparalelizovatelného kódu. H je čas strávený výpočtem paralelizovatelného kódu, ale sekvenčně. Pro velké p platí S(p)=1 + H/G. To by znamenalo, že perfektní urychlení je možné, pokud se lze vyhnout kódu, který nelze paralelizovat. Ale čím větší počet procesorů, tím větší bude režie jejich komunikace, takže to přece jen nepůjde (režie OS je další faktor).

Nebere v potaz load balancing, režii komunikace

- Pro problém pevné velikosti s narůstajícím počtem CPU narůstá režie, což od jistého počtu CPU vede k pomalejšímu běhu než při menším počtu CPU

Anomalní urychlení.

Distribuce rozsáhlých dat u distribuované aplikace může omezit nutnost stránkovat RAM. S dostatečně rychlými komunikačními kanály pak dojde k rychlejšímu vykonávání programového kódu, protože odpadá čekání na zpomalující IO operace provázející stránkování, včetně obsluhy příslušných přerušení. Např. paralelizované vyhledávací algoritmy mohou mít větší než lineární urychlení – lze rychleji upřesnit výběrová kritéria.

Superlinearní urychlení.

Je možné, aby S vyšlo lépe než perfektní urychlení. Nejedná se o chybu ve výpočtu, zejména, pokud porovnáváme proti nejlepšímu sekvenčnímu algoritmu. Vliv na to má **cache procesoru**, především případ, kdy nastane mnohem častější cache-hit než je obvyklé. Výpočet je pak prováděn mnohem rychleji, než když se programový kód dostává k procesoru z pomalejší paměti. Záleží na konkrétním programovém kódu, zda se ho bude dostatečné množství nalézat právě v lokální cache jednotlivých procesorů. Analogicky to platí i pro datovou cache.

Gustavsonův zákon

Říká, že výpočty nad velkou množinou dat mohou být efektivně paralelizovány.

Gustavson: nyní se tedy nebudeme snažit zkracovat čas výpočtu, místo toho se pokusíme v daném čase vypočítat co největší úlohu. U většiny paralelizovatelných úloh s rostoucí velikostí úlohy roste velikost čistě sekvenční části řádově pomaleji, než celková velikost.

Změna oproti Amdahlovi - místo konstatního rozsahu **se uvažuje konstantní doba běhu**.

- Pokud jde něco paralelizovat, vyřeší se za stejnou dobu větší problémy ("vypočte se toho víc")

Pokud bude sériová část zanedbatelná a budeme mít k dispozici spoustu procesoru s distribuovanou pamětí, Amdahl nám nepomůže. a(n)=čas výpočtu sériově, b(n) = čas výpočtu paralelně na p procesorech.

$$a(n) + b(n) = 1 \text{ výpočet na p CPU}$$

$$a(n) + p * b(n) = 1 \text{ počet na jednom CPU}$$

$$S = \frac{a(n)+p*b(n)}{a(n)+b(n)}$$

S = na jednom cpu / na p cpu

Jinak vyjádřeno:

$$S(P) = P - \alpha \cdot (P - 1)$$

- S = urychlení
- P = počet procesorů
- Alfa = neparalelizovatelná část každého paralelního procesu

Karp-Flattova metrika

$e = \frac{\frac{1}{y} - \frac{1}{p}}{1 - \frac{1}{p}} =$

určuje podíl, kolik kódu se provedlo sériově - y (má to být psí) určuje urychlení na p procesorech.

- Např. z tabulky s počtem využitých procesorů (první řádka) a dosaženého urychlení (druhá řádka) lze vypočítat pro jednotlivé počty CPU sériově prováděný podíl - *e*
  - Pokud *e* zůstává stejné s přibývajícím počtem CPU - hlavním důvodem je omezená možnost paralelismu (a urychlení *psí* narůstá čím dál méně)
  - Pokud *e* s přibývajícím počtem CPU roste, důvodem k slabému urychlení je režie paralelismu (buď se týká nastartování procesů, komunikace, synchronizace nebo omezení architektury)
  - Pokud *e* postupně klesá = ideál, čím dál větší část výpočtu se provádí paralelně

- $e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$
  - e – metrika, podíl, kolik kódu se provedlo sériově
  - p – počet procesorů
  - ψ - urychlení na p procesorech
    - Určí se z naměřených časů
- $T(p) = Ts + \frac{Tp}{p}$
  - Čas výpočtu na p procesorech
  - Ts – čas po který kód běžel sériově
  - Tp – čas po který kód běžel paralelně
- $T(1) = Ts + Tp$
- $e = \frac{Ts}{T(1)}$
  - Dosadí se do T(p)
- $\psi = \frac{T(1)}{T(p)}$
- $\frac{1}{\psi} = e + \frac{1-e}{p}$
- A úpravou dostaneme *e*

3.2.4. Faktory ovlivňující rychlost vykonávání programového kódu.

Motivace

- State of the Art schopností procesorů
    - Superskalární architektura
    - CISC instrukce se překládají na mikrokód, aby i procesory s CISCovou instrukční sadou mohly těžit z výhod RISCové architektury
    - Pipelining
    - Vykonávání instrukcí mimo pořadí
    - Přejmenovávání registrů
    - Spekulativní vykonávání kódu dopředu
    - Odhadování výsledků podmínek skoků
    - Vektorové instrukce

Ve výsledku může jedno jádro procesoru vykonávat celé instrukce, nebo alespoň části instrukcí, paralelně

- Tj. co programátor píše jako kód jednoho vlákna, se ve skutečnosti vykonává paralelně
- Tj. uvedené optimalizace spadají do rámce PPR
    - Více vláken = principiální pohled na PPR
    - +Optimalizace = reakce na aktuální stav
    - Co bývalo výhodné paralelizovat, to dnes může rychleji spočítat „sériový kód“
- Dobrý překladač může vytvořit takový strojový kód, který umí využít paralelizačních schopností procesoru
    - Proto se liší výkonnost kódu v závislosti na překladači a zvolené míře optimalizace
    - Ale ani ten nejlepší překladač neumí odhadnout vše
- Proto je třeba mu umět pomoci takovým zápisem kódu, ze kterého toho pozná co nejvíce
    - A proto je třeba znát, jak to vypadá z pohledu procesoru
    - Low-level techniky, pokud pro to není zvláštní důvod, se vyplatí nechat na překladači
    - Ale algoritmy na vyšší úrovni jsou záležitosti pro programátora

Základní myšlenky

- Co lze, to se vypočítá pouze jednou
- Co lze, to se vypočítá paralelně
    - Je však třeba zvolit správnou granularitu výpočtu
    - Paralelizace totiž není vždy výhodná
- V některých případech je sériový kód rychlejší, má menší režii – nevytváří a nesynchronizuje vlákna
- Paměť se alokuje co nejméně, využívá se již alokovaná paměť
- Program se píše tak, aby operační systém co nejméně swappoval
- Redukovat náklady na vytvoření a synchronizaci vláken
- Běžet co nejvíce v uživatelském adresovém prostoru
- Ale hlavně, psát efektivní kód už samotného vlákna

- Paralelizovaný neefektivní kód bude s největší pravděpodobností pomalejší než efektivně napsaný sériový kód

Skoky

- Eliminace skoků má zásadní vliv na zvýšení výkonu (jump je hodně náročnej)
- Využit instrukce cmov – bytecode pro to nemá ekvivalent
    - Cílem je uspořádat podmínky, aby si procesor správně tipnul, která instrukce bude další
- Při adresování paměti přes pointery *m musí překladač předpokládat, že *m může ukazovat kamkoli a nemá moc možností, jak optimalizovat. Proto vzhledem k optimalizacím je dobré použít zápis přes indexy, kdy takový zápis obsahuje jakousi nápovědu pro překladač, jak to celé optimalizovat. (= místo pointru který ukazuje na začátek pole – přičtení adresy a dereferencování, využít zápis formou pole pole[i] –jednoznačné pro překladač)

Eliminace nepotřebných sekvencí Store-Load

- Používání dočasných proměnných, které mohou být realizovány s pomocí registru
- Programátor by měl pomoci překladači jejich použitím – namísto co nejkratšího zápisu kódu

Konverze operandů

- Nemíchat operandy různé velikosti, konverze to zpomaluje

Loop Unrolling

- Nápověda pro překladač s autovektorizací
- V nejhorším dojde k většímu využití pipelines procesoru

Naivně	Lépe

`double a[100], sum;`
`int i;`
 	
`sum = 0.0f;`
 	
`for (i = 0;`
`i < 100; i++) {`
`sum += a[i];`
`}`
`//Překladač to může,`
`//ale i také nemusí`
`//správně pochopit.`
 	
`//Loop-unrolling`
`//také snižuje počet`
`//porovnávání, tj. i`
`//případných skoků.`
	`double a[100], sum;`
	`double sum1, sum2, sum3, sum4;`
	`int i;`
	
	`sum1 = 0.0f;`
	`sum2 = 0.0f;`
	`sum3 = 0.0f;`
	`sum4 = 0.0f;`
	
	`for (i = 0; i < 100; i + 4) {`
	`sum1 += a[i];`
	`sum2 += a[i+1];`
	`sum3 += a[i+2];`
	`sum4 += a[i+3];`
	`}`
	`sum = (sum4 + sum3) +`
	`(sum1 + sum2);`

Snaha omezit režii na obsluhu cyklů tím, že v rámci jednoho cyklu se vykoná několik operací po sobě “najednou”v jednom cyklu. Závisí to na faktoru který si určím, tak by to bylo ještě výhodné.

Vyhodnocování podmínek

- Podmínky jsou obvykle vyhodnocovány ve zkrácené formě – cílem je seřadit podmínky tak, aby při jejich vyhodnocování bylo zapotřebí provést co nejméně úkonů, tzn. Bylo co nejrychleji vyřazeno co nejvíce možností a aby byly co nejmenší nároky na vyhodnocení podmínky.
- Sekvence AND končí prvním FALSE, sekvence OR končí prvním TRUE.

Branch prediction – předpovídání skoků

- procesor obsahuje tzv. Branch prediction, kdy spekulativně vykonává kód dopředu podle toho, jaký předpokládá výsledek skoku. Uvedená konverze nemění stav branch-prediction, tj. Neovlivňuje urychlování volající funkce, a kdyby se alokovala paměť, např. Pro nový string, tak už to stav branch-prediction ovlivní.

Kopírování paměti

- kopírování celého bloku paměti místo prvek po prvku
- efektivnější je použít zkopírovaní celého bloku paměti, kdy se nebude kopírovat podle velikosti prvku pole, ale podle velikosti slvoa procesoru (např. Funkce: memcpy)

Garbage Collector

- Programátor si nedělá starosti s drahými operacemi přidělování/odebírání paměti
- způsobuje náhlé, nedeterministické vytěžování systému

Synchronizace

- jinak než kritickou sekcí, např. Pomocí InterlockedIncrement apod. Kritická sekce má velkou režii.

Využití cache

Viz superlineární urchlení

- výkonnosti program lez pomoci, pokud se data zpracovávají po takovýc hčástech, aby sůstala v RAM, nebo a by se kód případně data, dostala do cache procesoru
- programátor neovlivní, kolik z jeho kódu bude v cache procesoru, ale může zvýšit pravděpodonobst malým kódem, schopným samostatné činnosti

Využití registrů

Počet proměnných na procedure/funkci

- čím méně, tím větší šance, že budou v registrech

Synchronizace jinak než kritickou sekcí

- kritická sekce má ve skutčnosti velkou režii oproti celočíselné proměnné (lze do ní zapsat ID vláken jak jsou plánovány)

Přínos paralelizace

Kratšího výpočetního času paralelizací dosáhne programátor tehdly:

- Není-li režie použití vice vláken/procesů natolik velká, aby ve výsledku běžela paralelní verze déle než s jedním vláknem

Profiling

Profiling (jen poznámky z pohledu PPR)

- Staticky
  - Analýza programového kódu, co by se kde dalo naprogramovat lépe, aniž by se program spustil
    - Relevantní je právě znalost, jak psát efektivní programový kód
    - A jak funguje systém, pro který se programový kód píše

- Dynamicky
  - Až je hotová statická analýza, provede se dynamická
  - Za běhu programu se měří, který kód se vykonává nečastěji
    - Je pak předmětem dalších optimalizací
      - Jsou-li ještě nějaké možné
    - Anebo je předmětem optimalizace nadřazený programový kód, aby zbytečně často nevolal pomalý programový kód

Optimalizace ??

3.2.5. Programové prostředky pro multithreading - Java, POSIX, WinAPI, OpenMP.

Co je to vlákno?

Hierarchie z pohledu operačního systému:

- Proces
  - největší výpočetní entita plánovače
  - vlastní prostředky, paměť a další zdroje
  - v závislosti na OS možnost preemptivního multitaskingu
- Vlákno Thread
  - každý proces má alespoň jeden, primární, thread
  - jeden proces může mít několik vláken
  - vlákna sdílí adresový prostor procesu a jeho zdroje
  - každé vlákno má svůj vlastní kontext (id, registry, prioritu, atd.)
  - v závislosti na OS možnost preemptivního multithreadingu
- Vlákno Fiber
  - Fiber je analogií threadu k procesu
  - Fiber plánuje některý thread procesu, ne plánovač operačního systému
  - Fiber běží v kontextu threadu, který ho naplánoval
  - Má pouze stav, zásobník a specifická data – např. prioritu má pouze thread
  - Fiber může ukončit běh vlákna v jehož kontextu běží

Java

Java

- Plánovač OS vs. JVM => Java jako vysokoúrovňový prostředek
  - JVM může, v závislosti na implementaci (není jenom JVM od Sunu), převzít úplnou kontrolu nad plánováním javovských vláken, nebo využije služeb OS
- Přímá podpora synchronizace klíčovým slovem synchronized
- Třída Thread
- Rozhranní Runnable

Základní konstrukce Javy

- Rozhraní Runnable
  - Jakákoliv třída, která má být spuštěna ve vláknu musí implementovat toto rozhraní
  - Lze využít, pokud nechceme vytvářet potomky třídy Thread
  - Spustí se instancí třídy thread, které se implementované rozhraní zadá jako parametr
  - Nemělo by být používáno, pokud chcete měnit chování i jiných metod než je run

- Třída Thread
  - Implementuje rozhraní Runnable
  - S využitím dědičnosti lze rovnou vytvořit vlákno s definovaným chováním popsaným metodou run
  - Lze použít, pokud chcete změnit chování dalších metod třídy Thread

- synchronized u metod
  - Klíčové slovo používané v deklaraci metody
    - public synchronized void doSomething()
  - Maximálně jedno jediné vlákno může vykonávat (ne, být v kritické sekci) takto synchronizovanou metodu

- V programovacím jazyce Java jsou objekty a metody potřebné pro práci s vlákny dostupné ve standardních knihovnách.
- Základem je třída Thread

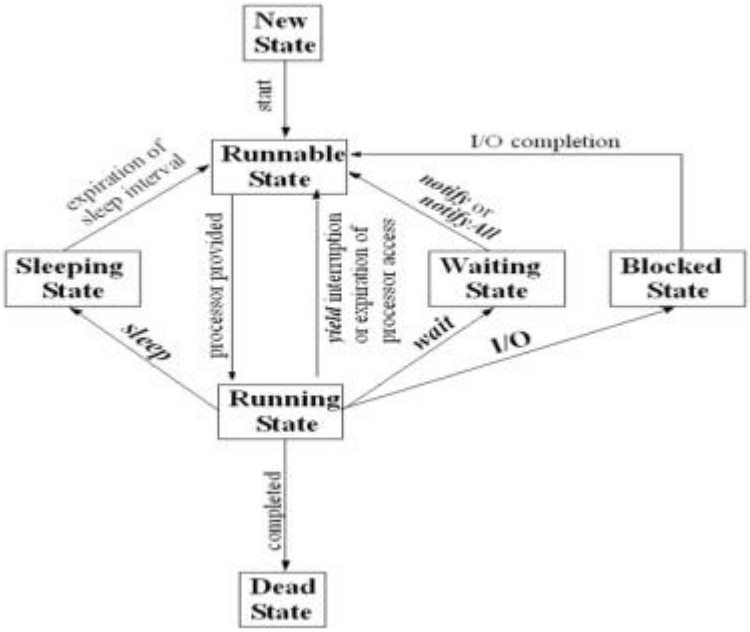
Thread

Třída Thread

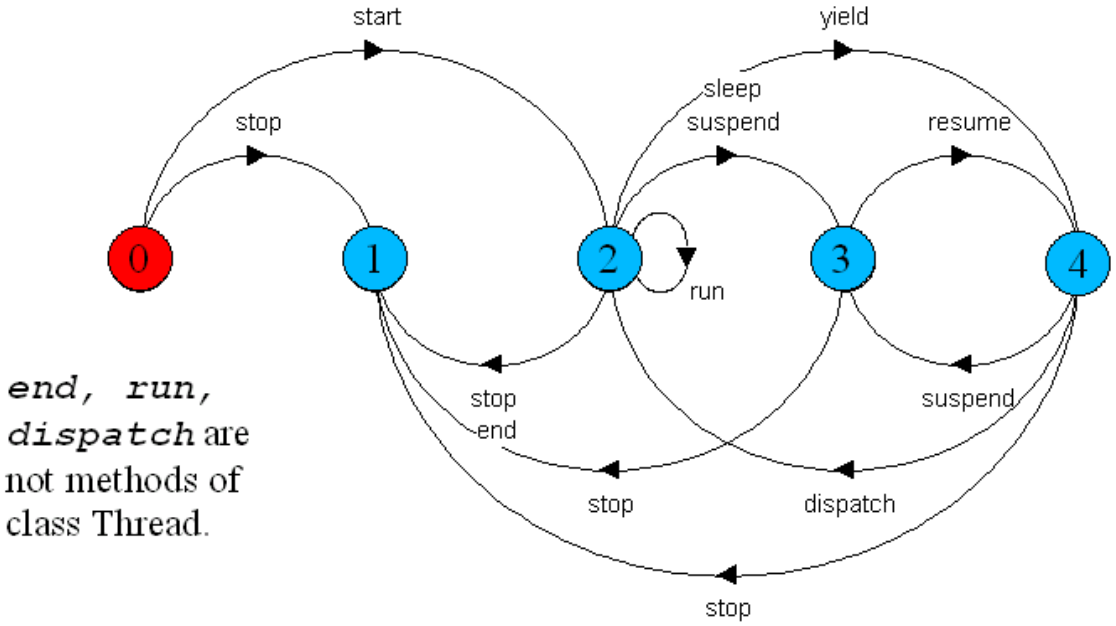
- Start
  - Zavolá metodu run – tj. spustí vlákno
- Stop
  - Deprecated
  - Zastaví běh vlákna a uvolní všechny zámky
- Wait
  - Pozastaví vlákno na monitoru objektu, dokud pro něj jiné vlákno nezavolá notify/all a nezíská opět monitor
  - Lze specifikovat, na jak dlouho se má vlákno uspat
    - Ve skutečnosti minimální dobu, na kterou se má uspat
    - Může být spuštěno o něco déle
- Notify
  - Vzbudí vlákno, které je pozastavené na příslušném monitoru
- NotifyAll
  - Vzbudí všechny vlákna, které jsou pozastavené na příslušném monitoru
  - Oproti notify má větší režii
    - není třeba ho používat, není-li to nezbytné

- tvoří základ všech paralelních programů v Javě
- pro jednoduché programy stačí oddědit od této třídy a překrýt metodu *run()*, do které se napíše výkonný kód vlákna.
- protože někdy je potřeba, aby naše třída dědila od jiné a zároveň měla vlastnosti vlákna, existuje ještě rozhraní *Runnable*, které stačí implementovat a třída rovněž získá vlastnosti vlákna
- pro napsání paralelního programu stačí vytvořit potřebné třídy, napsat jejich výkonné kódy do metod *run(),* a pak vlákna spustit (vlákna se spouští metodou *start()*
- *Stop, resume, suspend* - ***deprecated***
  - Stop - po jejím volání vlákno uvolní všechny držené monitory --> může zanechat uvolněné objekty v nekonzistentním stavu

Stavy vlákna (obecně):



Stavy vlákna v Javě:



end, *run*, *dispatch* are not methods of class Thread.

States 0 to 4 correspond to `CREATED`, `TERMINATED`, `RUNNING`, `NON-RUNNABLE`, and `RUNNABLE` respectively.

Monitory v Javě

- komunikace vláken je řešena přes sdílenou paměť
- kritické sekce jsou ošetřeny klíčovým slovem ***synchronized*** (monitor)
- monitor je součástí každého objektu
- pro implementaci monitorů jsou uvnitř objektu monitoru skryté atributy:
  - **1 zámek** – pro všechny synchronizované metody
  - **1 fronta** – pro blokovaná vlákna čekající na vstup do synchronizované metody (kritická sekce)

- nad frontou fungují privátní metody monitoru (objektu) *wait()*, *notify()* a *notifyAll()*
- kromě těchto implicitních monitorů objektu lze v metodě ještě vytvořit synchronizační blok, který může použít i jiný zámek, než je implicitní zámek objektu, což dovoluje jemnější členění vzájemného vyloučení (např. jen na část metody)

POSIX

Struktury v posixu jsou reprezentovány pomocí *handle*. Struktury jsou **pthread_t, pthreat_mutex_t** a **pthread_cond_t**. Objekty mají sadu atributů, které lze měnit pthread_attr_t, pthread_mutexattr_t, pthread_condattr_t. Funkce pro manipulaci s objekty buď vracejou 0 jako OK, nebo jinou hodnotu jako chybový kód.

Vlákna

Program vlákna je vlastně funkce C s typem `void* prog_name(void* arg)`

vlákno se vytvoří následujícím způsobem:

```
int status; //0 uspech, jinak chyba
status = pthread_create(&worker, NULL, prog_name,…);
```

- vlákno běží hned po vytvoření
- stavy vlákna jsou **ready, running, waiting** a **terminated**
- vlákno se ruší pomocí detach
- Atributy:
    - **způsob plánování**: FIFO – nejprioritnější kategorie (nejdříve se plánují vlákna tétokategorie), RR – round-robin, FG – foreground, implicitní kategorie, střídání vláken, ty s vyšší prioritou mají více času, BG – background, všechna vlákna se střídají, ale dostávají méně času než FG
    - **priorita**
    - **rozměr zásobníku**
    - **hlídač zásobníku** – jak daleko se můžeme od konce zásobníku dostat, jinak vznikne výjimka
    - **konec vlákna** - vlákno dojde na konec svého programu - na toto ukončení se lze synchronizovat z jiného vlákna pomocí pthread_join(na_koho_se_čeká, kam_přijde_výsledek) nebo zabito z vnějšku – pokud možno nepoužívat, základní funkce pro likvidaci pthread_cancel(oběť); oběť se může bránit pthread_setcancelstate(…), k likvidaci nemůže dojít kdekoliv v kódu vlákna, ale jen v předem připravených místech (volání blokující funkce, volání pthread_testcancel()), popisovaný způsob je synchronní, existuje i asynchronní)

Mutex

Je to synchronizační primitivum, umožňuje implementaci vzájemného vyloučení v kritické sekci. Jen jedno vlákno vlastnící mutex může být v kritické sekci, ostatní vlákna čekají.

```
Pthread_mutex_lock(mutex_handle)
Pthread_mutex_unlock(mutex_handle)
State = pthread_try_lock(mutex_handle)
```

Podmínková proměnná

Vlákno se uspí do té doby, dokud se nesplní nějaká podmínka, která se pak signalizuje pomocí podmínkové proměnné. Podmínková proměnná je svázána s mutexem.

`Pthread_cond_init`
`Pthread_cond_wait` `- blokující operace, vlákno se převede do stavu` *`waiting`*`, a odemkne se zámek`
`Pthread_cond_timedwait`
`Pthread_cond_signal`
`Pthread_cond_broadcast - jako notifyAll()`
`Pthread_cond_destroy`

WINAPI

Process

- Má virtuální paměťový prostor, systémové zdroje, bezpečnostní kontext (kdokoliv nemůže provádět cokoliv), jedinečný identifikátor, spustitelný kód
- Prioritní třída – vlákna pak mohou mít prioritu pouze v rozsahu prioritní třídy procesu.
- Má alespoň jedno vlákno (primární - to hlavní), které může vytvářet další vlákna – threads a fibers

Thread

- Entita v rámci procesu, kterou **plánuje OS**. Všechny vlákna sdílí paměťový prostor a zdroje svého procesu. Vlastní handler výjimek. Priorita: OS zajišťuje inverze priorit, dynamic boost, foreground/input včetně realtime.
- Jedinečný identifikátor
- TLS – thread local storage, data, která jsou specifická/lokální pro daný thread. Možnost, jak si thread může zabezpečit jedinečný přístup ke svým datům. Každý proces má k dispozici několik TLS slotů, které mohou být použity jeho thready. Možnost využití ke zpracování výjimek.

Fiber

- Běží v kontextu vlákna, **plánuje ho thread procesu**, jeden thread může naplánovat několik fibers. FLS – fiber local storage =analogie k TLS, FLS je asociováno s threadem.
- Má malý kontext (v porovnání s kontextem threadu) – zásobník, podmnožinu registrů a inicializační data.
- Má přístup do TLS threadu, v jehož kontextu běží.
- Nemá prioritu.

Synchonizační objekty WinApi

Mutex – vyžaduje přepnutí do režimu jádra, CreateMutex, OpenMutex, ReleaseMutex

Event – mutex v user space, stavy signaled/nonsignaled, může být buď pulsní, tj, překlopí se do nonsignaled jakmile propustí jednoho čekajícího, nebo zůstane signaled. CreateEvent, SetEvent, ResetEvent.

Semafor – klasický semafor, CreateSemaphore, OpenSemaphore, ReleaseSemaphore

Kritická sekce – Critical section, má velkou režii, EnterCriticalSection (blokující), TryEnterCriticalSection (neblokující), LeaveCriticalSection

Na signalizaci objektů lze čekat pomocí WaitForSingleObject nebo WaitForMultipleObjects.

Knihovny pro tvorbu paralelních aplikací bez nutnosti tvoření vláken v kódu:

OpenMP

Idea je taková, že vezmu sériový program a jen s minimem změn a s pomocí direktiv preprocesoru určím, které úseky kódu se budou vykonávat paralelně. Pokud se překladač vyvolá s direktivou OpenMP přeloží se jako paralelní program, jinak jako sériový. Pomocí direktiv pak lze překladači říci, které proměnné jsou sdílené a vyžadují tedy synchronizaci (kritickou sekci) a které nikoli.

Programování „bez vláken“

- „Tradiční“ přístup je vytvoření vícevláknového programu, kde se řekne, co má které vlákno dělat
- Ale jde to i jinak, lze vytvořit program tak, že se řekne, co se má udělat paralelně
- OpenMP, Intel Thread Building Blocks, MS Concurrency Runtime

OpenMP

- Open Multi-Processing
- Idea je, že se vezme sériový program a za použití speciálních knihoven se pomocí direktiv překladače řekne, co se má provádět paralelně
- Např. pokud se překladač vyvolá s direktivou openmp, program se přeloží jako paralelní, bez ní jako sériový

```
#pragma omp parallel
{
  for (int i=0; i<10; i++) {
        DoSomething();
    }
}
```

- Pokud se provádí paralelní výpočet, obvykle je nutná nějaká sdílená/redukční/uzamykaná proměnná
- Překladač se je může pokusit uhodnout, ale to se vždy nemusí podařit správně
  - Je lepší mu to prozradit pomocí speciálních direktiv
  - shared, private, reduction, default…
- To samé platí i pro synchronizaci a plánování

Intel Thread Building Blocks (TBB)

TBB knihovna jako taková využívá vláken OS, ale programátor s nimi již nepracuje. Místo toho jsou v TBB tzv. Tasky – knihovna pak vykonává jednotlivé úlohy paralelně. Lze si napsat vlastní plánovač

tasků a protože TBB nespoléhá jen na direktivy překladače, lze paralelizovat i takové úlohy, které by se s OpenMP paralelizovaly těžko.

3.2.6. **Intel Thread Building Blocks**

//pseudokod

Intel Thread Building Blocks

- Open Source, podpora více platforem
  - Stejně jako OpenMP
- Ačkoliv si je TBB vědoma vláken poskytovaných OS, myšlenka je takový, že programátor už s vlákny nepracuje
- Namísto vláken specifikuje úlohy, tasks, a jejich návaznosti

- Knihovna vykonává úlohy paralelně, jak nejlépe to jde
  - Obsahuje různé optimalizace v jakém pořadí úlohy vykonávat
  - Ne vždy platí, že FIFO je nejlepší strategie
    - Využívá se technika „task stealing“
  - Lze si napsat vlastní plánovač

- Stejně jako STL, i TBB intenzivně používá C++ šablony
  - Tj. nelze ji použít v C jako OpenMP

- Základní konstrukce TBB jsou
  - parallel_for, parallel_reduce, parallel_scan
  - parallel_while, parallel_do,pipeline, parallel_sort
  - paralelní kontainery, které jsou v STL
  - mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, recursive mutex
  - fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store

- TBB dokáže použít vlastní paměťový manažer, který je optimalizovaný na paradigma výpočtu úloh

- TBB je náročnější se naučit, ale zase se to vyplatí na výkonu, pokud se začíná psát nová aplikace, než např. paralelizovat s OpenMP
  - Navíc TBB se nespoléhá na direktivy pro překladač, takže podmínka if OpenMP se dá realizovat libovolně složitá, nebo se dají udělat konstrukce, který by šli s OpenMP udělat jen velmi obtížně – např. cancellation výpočtu v OpenMP není

- Např. chceme-li spustit na pozadí několik výpočetně náročných úloh paralelně

Případová studie použití TBB

- Uvažujme výpočetně náročnou aplikaci nad rozsáhlými daty s GUI
- Jedno vlákno bude obsluhovat GUI
- Jedno vlákno bude obsluhovat I/O
  - Naivní přístup je jedno vlákno pro zápis a jedno pro čtení
  - Lepší je jenom jedno vlákno a použití asynchronních I/O operací
    - OS si je uspořádá sám pro lepší výkon
    - Např. disky mají Native Command Queuing
  - Ale co s výpočetní částí?
    - Určitě v tom bude alespoň jedno vlákno, aby výpočet běžel na pozadí a GUI bylo responsive

- Přístup s psaním vláken by znamenal postarat se
  - Vytváření a rušení vláken, což je další režie pro OS
  - Jejich správnou synchronizaci, což je náročné, jakmile se program stává složitější
  - Výkonnostně je rozdíl, jestli se synchronizuje v kernel-mode, nebo v user-mode address space
  - Výkonnostně hraje roli, zda se k datům přistupuje ze stejného procesoru – thread affinity
  - Optimální počet vláken odpovídá počtu procesorových jader v systému, což ale není přístup, který je vždy možný při psaní vláken
    - Např. počet vláken může odpovídat tomu, jaká je metoda výpočtu – problém škálovatelnosti
      - S TBB se taková vlákna nahradí úlohami

- S TBB
  - O výše uvedené problémy se TBB postará
  - Programátor napíše jedno vlákno, ve kterém pak spustí výpočet úloh TBB
  - Nicméně programátor si stále musí být vědom toho, jak se píší paralelní programy s vlákny, aby mohl správně používat synchronizační primitiva TBB
    - Optimálně se naprogramují pouze úlohy s tím, že každá úloha po dokončení vrátí další úlohu, která se má vykonat jako navazující
    - Jako bariéra na dokončení více úloh se pak použije task list
    - Úlohy však mohou sdílet proměnné, a proto je třeba znát teorii o psaní vláken
      - Aby byla představa, že 2 úlohy mohou, ale i nemusí, běžet paralelně, např. když se má správně použít mutex, podmínka...

Concurrency Runtime

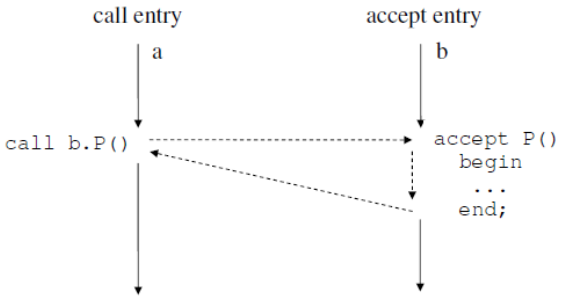
- Proprietární technologie MS
- Ideově podobná TBB, je to novější knihovna

- Spuštění více úloh

```
void Task1() {}
void Task2() {}


void Main() {
    task_group tg;

   tg.run(&Task1);
   tg.run(&Task2);

   tg.wait();
}
```

3.2.7. **Rendez-Vous, vč. select v Adě, a jeho porovnání s monitorem.**

- Rendez-Vous



  - Synchronní – call a accept jsou blokující
  - Propojí se adresní prostory obou procesů
  - /vláken – vždyť paměť je přece psaná na proces?!
    - Každé vlákno má svůj zásobník a adresy aktuálních lokálních proměnných funkcí se liší podle aktuálního vnoření volání funkcí
  - Sdružuje se synchronizace a výměna dat
  - Náročná implementace v distribuovaném prostředí
  - Accept entry je atomická operace se vzájemným vyloučením
    - Tj. může být aktivní pouze jeden – volající
      - Tj. s r-v lze implementovat monitor

```
select
    accept P1()
 or accept P2()
    …
 or accept Pn()
end;
```

  - r-v nejsou jenom samá pozitiva
    - Ada (bude později)
      - Všechny Tasky jsou aktivní, paralelně běžící vlákna
      - Monitor soupeří o zdroje s „normálním“ vláknem
      - Více přepínání kontextu, což je drahé
      - Minimálně se vynutí dvě přepnutí při každém zavolání (a opuštění) r-v

- Ada je objektově orientovaný jazyk se silnou verifikací typů (nelze implicitně přetypovat datové typy – např. void pointer)
- nepoužívá interpretr
- nepodporuje fibres
- paralelní části výpočtu se označují jako tasky (ne thready)
- mohou být prováděny na jednom procesoru, více procesorech nebo více počítačích, není to však vyjádřeno v kódu programu
- pro synchronizaci tasků se používá asymetrické synchronní rendezvous (zasílání zpráv)

Tasky

Konstrukce *task* představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy.

Deklarace je následující:

```
task jméno is
deklarace jmen komunikačních typů
end jméno;
task body jméno is
lokální deklarace a příkazy
end jméno;
```

- pro ukončení procesu lze použít příkaz *abort jméno;*, ale jeho použití by mělo být výjimečné
- k ukončení tasku se používá příkaz *terminate*

```
Select
    Accept e
Or
    Terminate
End select
```

Rendez-vous

Pro interakci procesů používá ADA principu asymetrického rendezvous, kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje tak elegantní konstrukci monitorů.

- prostředek pro synchronizaci úkolů (tasks)
- dva úkoly spolu komunikují pomocí rendez-vous - Meeting point, entry calls
- task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat

```
task Simple_Task is
entry Start(Num : in Integer);
entry Report(Num : out Integer);
end Simple_Task;
task body Simple_Task is
Local_Num : Integer;
begin
//čeká na vložení čísla – entry call
accept Start(Num : in Integer) do
Local_Num := Num;
end Start;
//normálně pokračuje v běhu
Local_Num := Local_Num * 2;
//čeká na vyzvednutí spočítané hodnoty
accept Report(Num : out Integer) do
Num := Local_Num;
end Report;
end Simple_Task;
```

- uvedený příklad stačí, pouze pokud potřebujeme jen jedno vlákno běžící podle uvedeného kódu
- průběh dostaveníčka - accept:
    - klient zavolá server
    - server si převezme parametry
    - server provede výpočet, klient spí

- server předá výsledky

Select - může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) – pokaždé na jiné dle okolností – tj. ne v předem určeném pořadí

```
//Vynutíme si inicializaci a další se
//už pak může vykonávat v libovolném
//pořadí.
accept Init(Item : in Integer) do
    Local_Item := Item;
end Init;
loop
    select
        accept Stop;
                exit;
        or
        when podmínka = > //může i nemusí být
                accept Put(Item : in Integer) do
                    Local_Item := Item;
                end Put;
                Local_Item := Local_Item * 2;
        else
                Put_Line("No entry call at this time");
    end select;
    delay 0.01;
end loop;
```

Protected Objects, Protected Types

- tasky mohou sdílet objekty
- objekt je instance typu – klíčové slovo *type*
- klíčové slovo *protected* zajistí exkluzivní přístup k chráněnému objektu
- jsou tři operace nad chráněnými objekty:
  - **Procedury** – mění stav objektu, aniž by pro to musela být splněna podmínka; překladač se stará, aby měly exkluzivní přístup k objektu
  - **Entry calls** – stejné jako procedury, ale pro vykonání *entry call* je třeba navíc splnit podmínku
  - **Funkce** – pouze vrací stav a nic nemění, a proto nemusí mít exkluzivní přístup k objektu

3.2.8. Výpočetní prostředí s distribuovanou pamětí.

MPMD

Systém pro paralelní výpočet s distribuovanou pamětí se skládá z výpočetních uzlů a komunikačních kanálů. Uzel je prvek, na kterém probíhá nějaká cast výpočtu – typicky počítač v síti, komunikační kanál je prve, který přenáší data mezi sousedními uzly např switch

- Univerzální počítačová síť (softwarový multipočítač, SETI)
- Univerzální paralelní počítač (stavěný přímo pro paralelní počítač, Cluster)
- Jednoúčelový paralelní počítač (jedna konkrétní aplikace s maximální optimalizací)

Protože neexistuje sdílená paměť, používá se pro komunikaci mezi procesy především zasílání zpráv

Protože systémy s distribuovanou pamětí nemají žádný úzký profil ve formě sběrnice, přes kterou by procesory přistupovaly ke sdílené paměti, hodí se pro úlohy vyžadující tzv. masivní paralelismus (stovky až tisíce procesorů)

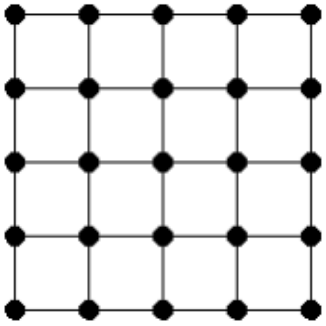
Obecně systém s distribuovanou pamětí umožňuje větší urychlení než systém se sdílenou pamětí díky paralelizaci komunikace

- Zatímco se data přenášejí kanálem, uzel může počítat
- Urychlení ovšem závisí na dalších parametrech
  - Objemu interakce
  - Celkovém objemu zpracovávaných dat
  - Konkrétní hw architektura
  - Jak dalece je použitý programový kód optimální pro danou architekturu

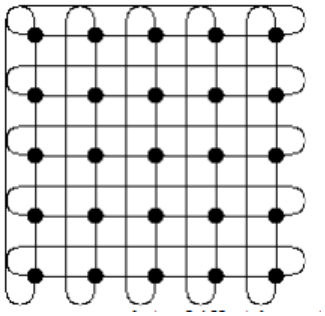
HW pohled

- Topologie obecně
  - Pravidelná: krychle, mřížka, hvězda
  - Nepravidelná: Internet
- Směrování
  - Pevná topologie – vzkaz lze poslat jen sousedovi
  - Podle příjemce – jak to známe z IP
  - Podle odesílajícího – odesilatel si určí cestu

- Fyzická topologie
    - Pevná: procesory jsou propojeny komunikačním kanálem
        - **Adresa může reflektovat polohu v síti**
        - Každý s každým
        - 2d mřížka ($d_{max} = 2(n - 1)$)



        - Toroid ($d_{max} = n - 1$)



        - 3d mřížka, n-rozměrné krychle…

    - Flexibilní: přepínání okruhů, přepínání paketů

Parametry

- N: Celkový počet uzlů v síti
- d_{ij}: vzdálenost mezi dvěma uzly (sousedé mají 1)
- d_{max}: nejhorší varianta, kolika uzly musí projít zpráva, než je doručena (nejdelší cesta zprávy v celém systému)
- počet sousedů: s kolika dalšími uzly je daný uzel spojen přímo
- přenosová kapacita:
    - agregovaně – kolik uzlů může najednou posílat zprávu
    - odolnost proti chybám – kolik komunikačních kanálů musí selhat, než se z jedné sítě stanou dvě

Snahou je dosáhnout

- Co největšího počtu uzlů v síti
    - Škálovatelnost
- Co nejmenší komunikační vzdálenosti – dmax
    - Tj. omezit komunikační zpoždění
- Co nejmenší počet sousedů
    - Aneb, i komunikační kanál něco stojí
- Dosáhnout co největší přenosové rychlosti

- Cílem je namapovat virtuální topologii na síťovou tak, aby docházelo k co nejmenšímu zpoždění
    - Ideálně se fyzická, síťová i virtuální topologie shodují 1:1
    - „Úplně ideálně“ jde zároveň i o optimální metodu výpočtu (který používá danou topologii)
    - Např. mřížka na mřížku o stejných, nebo větších, rozměrech
    - Úlohy tzv. geometrické dekompozice v rovině
    - Model Farmer-worker/Master-slave
        - Farmer úkoluje workery
        - Virtuální topologie je typu hvězda

SW pohled

- **Alokování uzlů**
    - 1 proces na 1 uzel
        - Např. pevně daná u paralelního počítače
        - 1 proces dokáže plně využít celý uzel, takže nemá smysl jich na jednom uzlu spouštět několik
        - OS uzlu neumí spustit více jak jeden proces najednou
    - Potenciálně nula až několik procesů na jeden uzel
    - Přidělení celé sítě pro jeden výpočet
        - Celkový čas výpočtu je pak dán
            - Dobou k zavedení programů, spuštění procesů a distribuce dat do uzlů
            - Vlastním výpočetem
            - Získáním výsledků z uzlů
    - Přidělení části sítě jednomu výpočtu
    - Několik paralelně běžících výpočtů
        - Na jednom uzlu může běžet několik procesů
        - Nelze se spoléhat na odvozená urychlení, protože ta nepočítala se zátěží, kterou vygeneruje neznámý kód
        - Nehodí se pro synchronní/lockCstepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku
- **Identifikace procesů**
    - Jedinečná ID procesů
    - Interakce send/receive (vše ostatní je na nich postaveno)
    - Podle přidělení na uzly:
        - 1 uzel – 1 proces
        - Více procesů na uzlu
        - Více procesů na uzlu a procesy mohou migrovat (tabulka umístění procesů)
- **Komunikační schéma**
    - Fyzická topologie = *jak je to sdrátováno*
    - Síťová topologie = *jak vidí fyzickou topologii software*
    - Virtuální topologie = *komunikační vazby procesů*

    - Ideálně 1:1 (aby docházelo k nejmenším zpožděním)

3.2.9. Rozdíly mezi PVM a MPI.

- PVM a MPI se používají v prostředí s distribuovanou pamětí

Hlavní rozdíly

- *PVM* - vznik v prostředí *heterogenní* sítě, *MPI* navržen spíše pro clustery - *homogenní* prostředí (u obou je ale běh možný v heteregonním prostředí)
- MPI - jednodušší a efektivnější abstrakce na vyšší úrovni
- MPI nabízí bohatší možnosti pro přenos zpráv (např. plně duplexní *sendrecv,* perzistentní komunikace, každý pošle každému - *MPI_Alltoall*)
- MPI se už v návrhu snaží o omezení kopírování paměťových bloků
- PVM se nestará o topologii, MPI podporuje logické komunikační topologie
- MPI nemá žádný config jako PVM
- MPI se vyhýbá nízkoúrovňovým rutinám kvůli přenositelnosti
- MPI - možnost definovat vlastní datové typy pro snížení režie při posílání velkého množství dat (vs. PVM - vícenásobné volání *pvm_pack*)
- MPI - podpora souborových operací (soubor se rozdělí na 1-N bloků, čtení a zápis jako zasílání zpráv)

PVM (Parallel Virtual Machine)

- Univerzální výpočetní model pro **heterogenní** distribuované výpočetní prostředí
- v PVM se musí provádět operace PVM_initsend, PVM_PK* apod.
- PVM umožňuje především provádět distribuované výpočty v heterogenním prostředí (různé architektury)
- PVM se nestará o topologii, MPI podporuje logické komunikační topologie.
- Programátor PVM může využít funkci PVM_Config aby např. určil, kde spustit další proces – MPI nic takového nemá.
- PVM programátorovi umožňuje, aby se do systému napojil pomocí nízko úrovňových rutin, MPI se tomu vyhýbá kvůli vyšší přenositelnosti.

Základní funkce

- Probíhá pomocí zasílání zpráv

pvm_spawn(char *task, char **argv, int flag,char *where, int ntask, int *tids)

- Spustí několik procesů podle zadaného programu
- Procesy se po vytvoření neznají -> proces, který je vytvořil je musí seznámit
- **numt** – počet úspěšně spuštěných procesů (*návratová hodnota)*
- **task** – spustitelný soubor
- **argv** – parametry
- **flag** – možnosti spuštění

    - **PvmTaskDefault** – PVM si rozhodne, kde spustit
    - **PvmTaskHost** – where bude obsahovat adresu, kde spustit
    - **PvmTaskArch** – where bude obsahovat typ architektury/platformy
    - Existují i další jako *PvmTaskDebug*, *PvmTaskTrace*, *PvmMppFront*, *PvmHostCompl*
- **where** – kde spustit proces, ignoruje se, pokud nejsou příslušně nastaveny možnosti spuštění
- **ntask** – počet procesů ke spuštění
- **tids** – identifikátory spuštěných procesů

pvm_initsend(int encoding) – nastavení kódování (defaultně se používá PvmDataDefault)

int pvm_pkint(int *ip, int nitem, int stride) – vloží data do bufferu

int pvm_send(int tid, int msgtag) – pošle data procesu (TID)

int pvm_recv(int tid, int msgtag) – přijme data od procesu (TID)

pvm_upkint(int *ip, int nitem, int stride) – rozbalí data

MPI (Message Passing Interface)

- MPI je postaveno na PVM
- Knihovna pro podporu paralelních výpočtů v systémech s distribuovanou pamětí, vazby (interface) má pro programovací jazyky C a Fortran.

- Proti PVM se jedná o programovací prostředek vyšší úrovně, vztah je asi jako mezi jazykem symb. adres (odpovídá PVM) a vyšším programovacím jazykem (odpovídá MPI), tedy:
- v PVM jde naprogramovat "skoro cokoliv", ale dá to velkou práci a program pak nejspíš nebude přenositelný,
    - dominantní aplikací pro MPI jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný do jiné instalace MPI

- MPI je primárně míněno (na rozdíl od PVM) pro **homogenní** výpočetní prostředí (tj. cluster stanic se společnou administrací, superpočítač jako N-Cube, ap.), takže poskytuje prostředky i pro "synchronní" algoritmy (tj. takové, kde se předpokládá přibližně stejná rychlost běhu jednotlivých procesů) a odpovídající statické rozdělení práce mezi procesy.

- MPI primárně využívá SPMD model paralelního výpočtu, tj. vyrobí se (na rozdíl od PVM) jen jeden "exe" soubor programu a ten se zavede do zvoleného počtu (dále N) procesorů. V každém procesoru běží jen jeden proces. Všech N procesů tudíž běží podle téhož programu, zjistí si své číselné ID a podle toho odliší svou činnost. V zásadě je tudíž realizovatelný i MPMD model výpočtu (třeba ve verzi farmer-workers, přičemž v programu je přepínač podle ID, jednu větev realizuje proces s číslem třeba 0 - farmer, druhá větev (jiná čísla než 0) je pro procesy typu worker).

- Komunikace procesů je asynchronní message-passing s přímým adresováním přes číselné ID procesu. Existuje mechanismus skupin procesů (viz dále tzv. komunikátory) a možnost

broadcastu zprávy ve skupině procesů. Zprávy není na rozdíl od PVM třeba pracně "pakovat". MPI má svoje "primitivní datové typy" a z nich lze skládat "strukturované typy", sloužící ovšem jen pro účely komunikace (tj. zjednodušený popis toho, co má přijít do zprávy - lze srovnat s náročností "pakování" zprávy v PVM).

- Existuje sada funkcí pro tzv. "globální operace", tj. operace nad daty, jejichž instance jsou "rozprostřeny" ve všech procesech výpočtu. Realizace takových operací v PVM se musí pracně rozepsat do primitivnějších operací pvm_send() a pvm_recv().

Základní funkce

Je jich 6, přičemž už umožňují napsat jednodušší aplikaci. První čtyři z nich je třeba použít v každém MPI programu. Všechny funkce vrací celočíselný kód úspěšnosti provedené operace, přičemž symbolická hodnota MPI_SUCCESS znamená úspěch. Přehled základních funkcí v C-syntaxi:

int MPI_Init (int* argc, char* argv)**

- Inicializace MPI výpočtu, argc a argv jsou argumenty hlavního programu.

int MPI_Comm_size (MPI_Comm comm, int* adr_size)

- Zjištění počtu procesů, počet se dosadí do proměnné odkazované parametrem adr_size. MPI_Comm je typ "komunikátor", pokud při volání dosadíme za parametr comm hodnotu MPI_COMM_WORLD, zjišťujeme počet všech vytvořených procesů aplikace N.

int MPI_Comm_rank (MPI_Comm comm, int* adr_rank)

- Zjištění čísla procesu v rámci "komunikačního světa" comm. Čísla jsou v rozmezí 0 až M-1, kde M je "rozměr komunikačního světa" reprezentovaného komunikátorem comm (M = N, pokud dosadíme za comm hodnotu MPI_COMM_WORLD).

int MPI_Finalize (void)

- Ukončení výpočtu v MPI, provádí každý proces.

int MPI_Send (void* adr_buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- Odeslání zprávy s typem tag v komunikačním světě comm procesu dest. Odesílá se zpráva z bufferu buf obsahující count položek typu datatype.
- Čili buffer pro zprávu je na rozdíl od PVM kdekoliv v datech programu (zadá se adresa příslušného pole). Za datatype se dosazují buď primitivní typy MPI, například MPI_INT, MPI_DOUBLE, MPI_CHAR, nebo strukturované typy vytvořené z primitivních (viz dále).

int MPI_Recv (void* adr_buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *adr_status)

- Blokující příjem zprávy. Parametry mají analogický význam jako u MPI_Send. Navíc je parametr adr_status, odkazující kam se má uložit status příjmu zprávy (výstupní parametr). Status obsahuje položky: status. MPI_SOURCE - od koho zpráva přišla a status.MPI_TAG - jakého typu zpráva přišla. Dosadí-li se za source hodnota MPI_ANY_SOURCE a za tag MPI_ANY_TAG, přijme

se jakákoliv zpráva a z uvedených položek výstupního parametru status se dá zjistit co to vlastně přišlo.

Globální operace MPI

- Globální operace jsou operace, do kterých jsou zapojeny všechny procesy patřídí do téhož "komunikačního světa" (tj. jedním parametrem funkcí pro globální operace je příslušný komunikátor). Funkci globální operace volají všechny zúčastněné procesy (což je pochopitelné, protože typicky procesy běží podle téhož programu), přičemž jejich činnost se v obecném případě liší podle čísla procesu.
- Globální operace v PVM nejsou (kromě broadcastu) a musely by se pracně rozepsat do posloupnosti operací send() a receive().

Globální operace MPI lze rozdělit na tři skupiny - synchronizace, přesuny dat a redukční operace.

Synchronizace

Každý komunikátor mj. realizuje bariéru, na které se mohou všechny jeho procesy synchronizovat voláním funkce

int MPI_Barrier (MPI_Comm comm)

- Volání této funkce je tedy blokující a výpočet každého procesu pokračuje následujícím příkazem až poté, kdy se na bariéře "sejdou" všechny procesy patřící do "komunikačního světa" comm.

Přesuny dat

- Jedná se o operace s charakterem broadcastu, shromáždění či rozptýlení dat a tzv. redukční oprace.

  int MPI_Bcast (void* adr_buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- Operace broadcast. Má v zásadě stejné parametry jako MPI_Send(). Proces s číslem root vysílá, ostatní zprávu přijímají (čili root používá buffer jako vysílací, ostatní jako přijímací). Proti send() chybí tag - pro "globální" komunikační operaci nemá význam - všichni aktéři komunikace prochází tímtéž bodem programu a tudíž vědí, "o co při komunikaci jde".

  int MPI_Gather (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)

- Proces s číslem root shromažďuje data od všech procesů včetně sebe. Čili všechny procesy vysílají data (count položek typu intype) z bufferu inbuf a proces root je zapisuje do bufferu outbuf - samozřejmě je zapisuje v pořadí podle čísel vysílajících procesů (tj. nikoliv tak, jak zprávy došly). Parametr outbuf (a též outcnt a outtype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

  int MPI_Scatter (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)

- Inverzní operace k Gather(), tj. proces s číslem root "rozptyluje" data z bufferu inbuf všem ostatním procesům včetně sebe. Vstupní buffer musí obsahovat M částí dat (obecně různých, jedna část je pole count položek typu intype), přičemž i-tá část se pošle do bufferu outbuf procesu s číslem i. Parametr inbuf (a též incnt a intype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

Redukční operace

Redukční operace má M operandů umístěných ve vstupních bufferech komunikujících procesů. Výsledek operace se zapisuje do výstupního bufferu buď jednoho určeného procesu (root) nebo všech procesů.

int MPI_Reduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Za parametr op se dosazuje typ realizované operace, kde použitelné symbolické hodnoty typu jsou

- MPI_MAX, MPI_MIN (maximum, minimum),
- MPI_SUM, MPI_PROD (součet, součin),
- MPI_LAND, MPI_LOR, MPI_LXOR (logické operace)
- MPI_BAND, MPI_BOR, MPI_BXOR (logické operace se všemi bity)

Operandy jsou ve vstupních bufferech procesů, výsledek je ve výstupním bufferu procesu root.

int MPI_Allreduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- Funguje jako předchozí operace, ale výsledek dostanou do výstupního bufferu všechny zúčastněné procesy, tudíž chybí parametr root, který pak nemá smysl.

Komunikátor

Komunikátor je interní objekt MPI (typ MPI_comm, jedná se o typ tzv. "handle", čili jakéhosi zobecněného ukazatele reprezentujícího komunikátor).

Komunikátor reprezentuje skupinu komunikujících procesů a komunikační kontext této skupiny.

Dále komunikátor slouží pro paralelní členění programu, či jinak řečeno pro realizaci modelu MPMD (funkční paralelismus), kdy se procesy aplikace rozdělí na skupiny (reprezentované různými komunikátory) a každá skupina "dělá něco jiného" a její procesy "se vybavují jen mezi sebou". Čili uvnitř skupiny procesů (komunikátor) funguje datový paralelismus, kdežto mezi skupinami procesů funguje funkční paralelismus.

Základní funkce nad komunikátory jsou (podrobnosti viz literatura):

MPI_Comm_rank()

- vrací počet procesů komunikátoru, základní funkce MPI - viz dříve v části 2

MPI_Comm_dup()

- vytváří duplikát komunikátoru

MPI_Comm_split()

- "štěpí" kpomunikátor na dva jiné, čili rozděluje jednu skupinu procesů na dvě jiné

MPI_Comm_free()

- uvolňuje (likviduje) komunikátor (samozřejmě nikoliv procesy, které komunikátor reprezentuje)

MPI_Intercomm_create()

vytváří tzv. "interkomunikátor" jakožto prostředek komunikace mezi dvěma skupinami procesů.

3.2.10. Přidělování práce v prostředí s distribuovanou pamětí, možnosti urychlení výpočtu a přiřazení procesů na jednotlivé uzly.

Faktory ovlivňující rychlost výpočtu

- Virtuální topologie, komunikační schéma, distribuované aplikace
- Prezentovaná síťová topologie
- Fyzická topologie sítě
- Výkonnost jednotlivých uzlů v síti
- Výpočetní model distribuované aplikace
    - Každý proces může běžet podle vlastního programu

Řešení obecně

- Umístit procesy na uzly sítě takovým způsobem, aby běžely co nejrychleji a zároveň bylo komunikační zpoždění co nejmenší
    - Může se jednat i o protichůdné požadavky
- Zatížení a dostupnost uzlů se může měnit v čase
- Jsou tři typy úloh
    - S konečným časem výpočtu – distribuovaná aplikace má jenom něco spočítat a pak skončí
    - S teoreticky nekonečným časem výpočtu – distribuovaná aplikace poskytuje službu
        - Neplatí, že uzel musí být zcela vytížen výpočtem po celou dobu
    - Processing While in Transit – výpočet se nad daty provádí během jejich přenosu Např. Active Network

Load-Sharing

=přerozdělování paralelismu v distribuovaném systemu – začne rozdělovat až se uzel master vytíží

- Předpokládá se prostředí pracovních stanic, které nemusejí být vždy plně vytíženy
- Jeden uzel se vyhradí jako master, kde se spustí aplikace
    - Ostatní uzly se označují termínem slave
- V okamžiku, kdy je master vytížen na maximum, zkusí se vyhledat nevytížený uzel

Červ

- jednotlivé procesy se mohou replikovat na nevytížených uzlech
- červ se skládá z několika segmentů – procesů
- počet segmentů je buď pevně stanoven, nebo se při pokročilejší implementaci stanovuje dynamicky podle okolností
- nápadně připomíná šíření virů
- červ musí být věrohodný pro uživatele pracovních stanic
- komponenty červa
    - inicializační kód ke spuštění master
    - inicializační kód ke spuštění slave

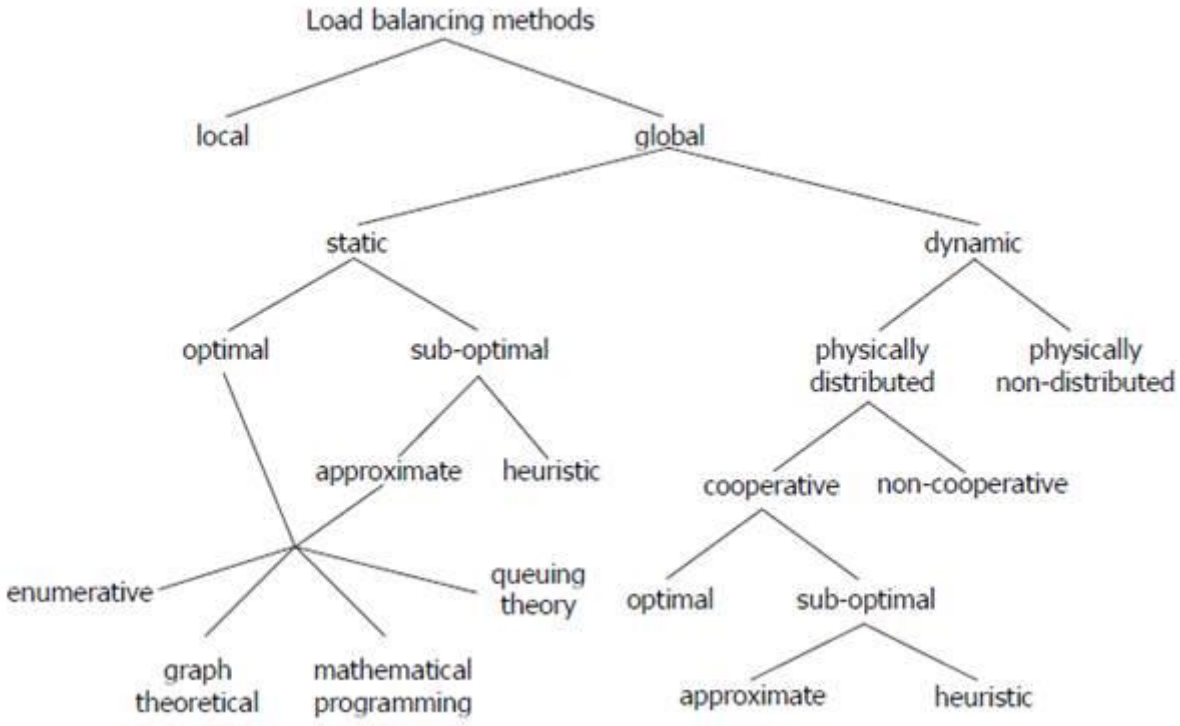
    - výpočetní program
    - ze života červa
        - nalezení nevytíženého uzlu
    - žádný uzel nezná globální stav sítě, a proto se každý segment musí postarat o nalezení volné stanice sám za sebe
    - lze hledat například pomocí broadcast hodila by se synchronizace, protože několik segmentů může soupeřit o stejný uzel
        - uvolnění uzlu
    - segment se musí postarat, aby uzel byl zase viditelný jako dostupný i pro ostatní
        - kontrola růstu
    - čím větší červ, tím vyšší rychlost výpočtu, ale vznikají další problémy
- rychlost nemusí růst lineárně
- synchronizace
- stabilita

Condor

- snaží se o fér využívání všech uzlů
- pokud některý uzel selže, proces se restartuje jinde
- arbitr, který rozhoduje o tom, kde se spustí proces
    - sám hledá použitelné uzly
    - centralizované místo
- možnost selhání
- checkpoints
    - procesy lze relokovat za běhu distribuované aplikace
- používá se ukládání obrazu procesu v paměti,
- ne rekonstrukce stavu
- uzly musejí být identické
- co s otevřenými soubory?
    - checkpoint se ukládá periodicky
- pokud selže výpočet, použije se poslední
- checkpoint
    - pre-emptivnost procesů je implementována pomocí checkpointů

Load-Balancing

- na rozdíl od load-sharingu se předpokládá, že celá síť je dostupná pro výpočet
- existuje několik různých metod, které se liší použitelností, účinností, náročností na zdroje (paměť, procesor, …), přesností, spolehlivostí, …



- **statické**
  - výpočet přiřazení procesů na uzly je proveden ještě před spuštěním distribuované aplikace
    - výpočet může běžet libovolně dlouho, abychom dosáhli požadované přesnosti předpovědi – pokud ji metoda umožňuje dosáhnout
  - nelze reagovat na dynamické změny v prostředí
  - vyžadují předem spoustu informací o chování sítě a aplikace (např. kom. zpoždění, doby běhu procesů)
    - nereálné požadavky nelze splnit
      - vliv na přesnost a tedy i rychlost výpočtu
- **dynamické**
    - výpočet přiřazení procesů na uzly sítě se provádí za běhu distribuované aplikace
    - výpočet se odehrává v reálném čase a nemůže si proto dovolit konzumovat příliš mnoho zdrojů
      - umí se vyrovnat s dynamickými změnami
      - procesy musí umět pre-empci
      - potřebné informace lze zjistit až za běhu aplikace, nebo si jich část vyžádat předem
- **pre-emptivní**
    - procesy lze přerušit během výpočtu a přemístit je na jiný uzel, aby bylo možné kompenzovat změny v síti
    - např. některý z procesů mohl skončit svoji činnost, nebo se odebral do dlouhodobého wait-stavu
    - pokud tuto vlastnost procesy nemají, na změny lze reagovat až při vytváření
  - centralizované

        - mají jeden centrální prvek, arbitr, který rozhoduje o rozdělování zátěže na jednotlivé uzly
        - centrální prvek je slabé místo, co se stane, když selže?
            - centralizované správa vyžaduje komunikaci jednoho uzlu se všemi
        - možnost přetížení
        - arbitr má přehled o známé síti, a proto lze očekávat, že dokáže zátěž rozdělovat celkem efektivně bez rizik, která jsou jinak spojena s distribuovaným principem
- **distribuované**
        - rozhodování o rozdělování zátěže provádí několik až všechny procesy
        - mohou být distribuovány na několik uzlů
        - když jeden selže, nic se neděje, pokud ho výpočetní model aplikace nutně nepotřebuje k životu
        - procesy, které provádějí rozhodování mohou být buď specialisté, anebo to mají jako „vedlejšák“ ke své hlavní činnosti
- **adaptivní**
        - síť prochází změnami během výpočtu – mění se stav uzlů
        - adaptivní metody berou do úvahy i několik předchozích stavů při rozhodování o přidělení zátěže
- **kooperativní**
        - každý proces se může rozhodovat buď sám za sebe, nebo může na rozhodnutí spolupracovat s ostatními
        - přímo – procesy spolupracují nad konkrétním rozhodnutí
        - nepřímo – procesy dávají informace o svých rozhodnutích k dispozici ostatním a ty je použijí při svých rozhodnutích
- **sender-initiated**
        - v okamžiku, kdy je uzel zatížen přes určitou mez, začne vyhledávat jiné uzly, kam by přemístil část své zátěže
        - je to režie navíc, protože čas potřebný na vyhledávání nových uzlů mohl být použit na běh procesů
- **receiver-initiated**
        - v okamžiku, kdy zátěž uzlu klesne pod určitou mez, začne vyhledávat jiné uzly, odkud by mohl převzít jejich zátěž
        - režie se projeví zvýšenou komunikací, uzel má dost volného výpočetního času, který může alokovat pro vyhledávání zátěže

Load Redistribution

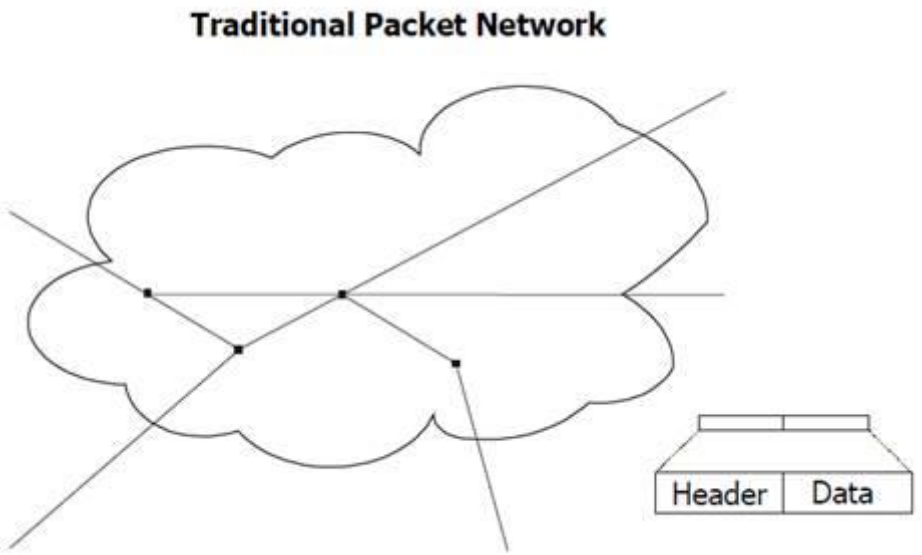
- load-balancing tradičně měří zátěž v počtu procesů, což není zrovna to nejlepší
- následující text bude o Load-Redistribution Method in Distributed Environment
- metoda vyžaduje pokročilou síťovou architekturu jako jsou Aktivní sítě

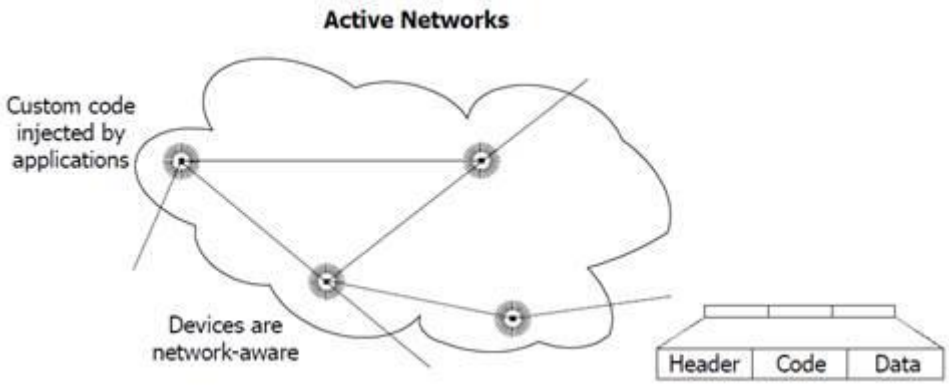
Aktivní sítě

- stvořil Pentagon pro vyřešení nedostatků IP protokolu
- např. *Any-Cast* = metodologie pro adresování a routování, kdy jsou datagramy od jediného odesilatele routovány uzlu, který je topologicky nejblíže v dané skupině potenciálních příjemců

(ačkoliv může být zasláno vícero uzlům, pokud mají stejnou adresu). Rozdíl od multicastu, který posílá všem ze skupiny najednou (a vždy), broadcastu, který zasílá jednoduše všem.

- Any-Cast je až v IPv6, aktivní sítě mají PAMcast – Programmable Any-Multicast – služba pro doručování zpráv, která generalizuje jak anycast tak multicast , která doručuje zprávy M z N příjemců, kde 1 <= M <= N
- IP
  - Dvojice vysílající a příjemce
  - Vysílající odešle paket na konkrétní adresu, včetně portu, kde předpokládá příjemce
  - Pokud tam není, paket se zahodí a vysílající zjistí chybu až timeoutem
- Aktivní síť
  - Paket, nazývaný capsule, je asociován s kódem, který se spustí na každém uzlu, kterým capsule prochází
    - Vždy je přítomný příjemce
  - Kód může manipulovat s daty, vlastnostmi (cíl, TTL, atd.) a vykonávat další uživatelsky definované činnosti
  - Proces se označuje termínem aktivní aplikace
    - Distribuovaná aktivní aplikace se skládá z několika aktivních aplikací, které mohou injektovat capsule a zároveň capsule může injektovat aktivní aplikace





- Komunikační model sám-sobě

- Je možné injektovat capsuli do sítě, aby nasbírala potřebná data a pak je předala procesu, který ji injektoval
- U IP by bylo nutné mít dopředu na každém uzlu, který by capsule mohla navštívit, spuštěný specializovaný proces

- Migrace procesů
  - Migrující proces změní svoji síťovou adresu, ale ještě ji nedal na vědomí ostatním procesům
  - Informaci o své nové síťové adrese zanechal na uzlu, odkud migroval
  - Capsule, která má doručit data, dorazí na uzel, odkud proces odmigroval, tam ho nenajde, ale použije svůj kód, aby si přečetla novou adresu a pouze změní svůj cíl
  - Ostatní procesy si mohou aktualizovat záznamy až později – lazy update, u IP je nutné vyřešit předem
- V aktivní síti je zapotřebí standardizace pouze dvou věcí
  - Programového kódu
    - Kód vykonává Execution Environment (EE), na jednom uzlu může být několik EE
  - Code distribution protocol
  - Vše ostatní je pak už aplikačně specifické

- Při přerozdělování zátěže (load redistribution) se procesy rozhodují samy za sebe, periodicky sledují své okolí (jako kolonie organismů)
  - Dosažení vyváženého stavu ne hned, ale postupně

- Capsule zjistí síťové okolí uzlu z hlediska topologie, výkonnosti, zatížení, komunikačního zpoždění apod.
  - Využije se při hledání výhodnějšího uzlu pro odmigrování

- **Rizika:**
  - *Masová migrace* - více procesů si vybere stejný uzel, ten se stane přetíženým
  - *Oscilace* - proces se může pohybovat po síti, aniž by něco počítal
    - řešení - zavedení kreditů, od určitého počtu může migrovat
  - *Zbytečná migrace*

3.3. FJP

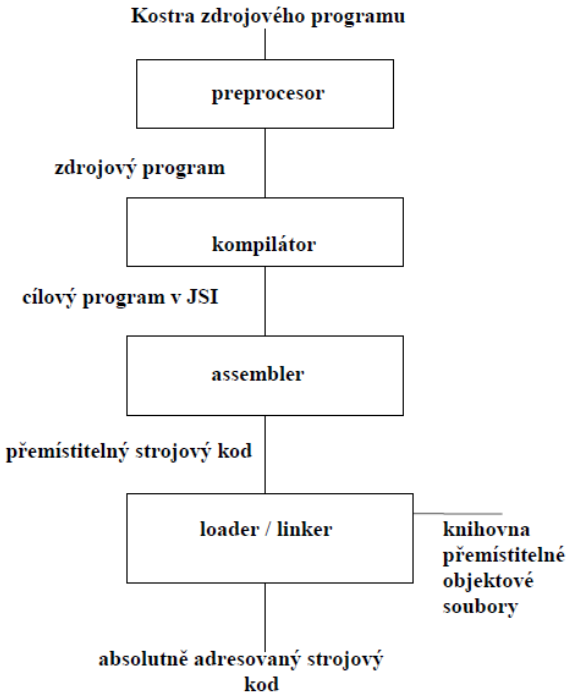
3.3.1. Překladače – typy, struktura a princip činnosti

Formálně je překladač **zobrazením ze zdrojového jazyka do cílového jazyka**.

Typy

Postup při tvorbě cílového spustitelného programu:

Zdrojový program → [preprocesor] → upravený zdrojový program → [kompilátor]→cílový program v jazyce symbolických instrukcí → [assembler] → relokovatelný strojový kód & zvenku: knihovní soubory a další relokovatelné objektové soubory → [linker/loader] → cílový strojový kód



Obr. Systém zpracování jazyka

Preprocesory
Realizují vnořování částí programu do hostitelského jazyka. Např. expandují makra, přidají include <něco.h> apod. Jeho úkolem je posbírat zdrojový program.

Kompilátory
Generují z vyššího programovacího jazyka kód – strojový/symbolický/jiný jazyk (1. Fortran IBM, 50. léta). Assembly language (jazyk symbolických adres) je jednodušší vyplivnout jako výstup a je i jednodušší pro debugování.

Interprety
Namísto přeložení celého programu provádí příkaz za příkazem operace uvedené ve zdrojovém programu nad vstupními daty. Proto jsou interprety obecně pomalejší. Obvykle ale díky spouštění programu příkaz za příkazem lépe diagnostikují chyby než kompilátory.(vyšší programovací jazyk + data => výsledky)

Java – kombinace kompilace a interpretace
Zdrojový program je nejdříve zkompilován do *bytecode* a ten je pak interpretován virtuálním strojem.

Výhoda – bytecode může být zkompilován na jednom stroji a interpretován na jiném. Aby to bylo rychlejší, některé Java kompilátory (*just-in-time* kompilátory) převádí bytecode do strojového jazyka těsně předtím, než proběhne intermediate (vnitřní) program pro zpracování vstupních dat.

Zdrojový program → [Translator] → intermediate program + vstup → [Virtual Machine] → výstup

Assemblery
Překládají z jazyka symbolických instrukcí (JSI) do strojového kódu. Přeložený strojový kód může být buďto *absolutní binární kód* nebo *přenositelný binární kód.*

Hlavní problémy, které řeší, je adresace symbolických jmen a makra.

Linker a loader
Větší programy jsou často kompilované po částech, takže vytvořený relokovatelný strojový kód (= lze jej umístit do libovolného místa v paměti) je potřeba spojit s dalšími relok. objektovými soubory a knihovními soubory. O to se stará **Linker.** Řeší adresy externí paměti, kde kód v jednom souboru může odkazovat na místo v jiném souboru. **Loader** pak narve všechny spustitelné objektové soubory do paměti pro spuštění.

Typy překladačů

Formátory textu
jde o úpravu textu podle požadavků uživatele, např. TeX. Např. syntax highlighting, nebo přeformátování kódu – odsazení apod. Takový překladač, který ze vstupního souboru definovaného určitým jazykem vygeneruje výstup. (překladače pro sazbu textu třeba Tex -> DVI)

Silikonový překladač
Pro návrh integrovaných obvodů. Proměnné nereprezentují místo v paměti, ale logickou proměnnou obvodu. Výstupem je návrh obvodu.

Dávkový překladač
Dávkové zpracování

Inkrementální překladač
Je interaktivní a překládá po úsecích

Křížový překladač
Překládá na jiném procesoru než na kterém se program (přeložený kód) spouští (např. zabudované – embedded – systémy)

Kaskádní překladač
Máme již překlad z jazyka A do jazyka B, chceme ale A→C. Pak vytvoříme kompilátor z jazyka B do jazyka C, pokud je to snazší než vytvořit kompilátor z jazyka A do jazyka C. Jazyk B je vnitřním jazykem, a pokud je to standardní všeobecně používaný jazyk, pak programy v jazyce A budou snadno přenositelné.

Nevýhodou je však, že oba překladače produkují chybové zprávy. Chybové zprávy druhého překladače jsou cizí pro uživatele jazyka A, protože jsou orientovány na jazyk B. Chybová hlášení výpočtu tak budou pomíchaná.

Paralelizující překladač
Zjišťuje nezávislost úseků programu

Optimalizující překladač
Možnosti ovlivnění optimalizace času či paměti programátorem.

Konverzační překladač
interaktivní

Struktura, princip činnosti

Překladače jsou dva druhy: kompilátory a interprety

Struktura Kompilátoru
všechny příkazy překládá najednou, program lze spustit až po ukončení celého překladu (Pascal, C, Fortran, Ada, …)

ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **generování kódu**, cílový program

Všechny části spolupracují s pracovními tabulkami překladače. Základní tabulkou kompilátoru i interpretu je **tabulka symbolů.** Obsahuje záznamy o názvech proměnných, jejich typu, rozsahu, názvy procedur společně s věcmi jako počet a typy argumentů, metoda předání jednotlivých argumentů (odkazem nebo hodnotou) a návratový typ.

Výhodou kompilátoru je rychlá exekuce programu.

Struktura Interpreta

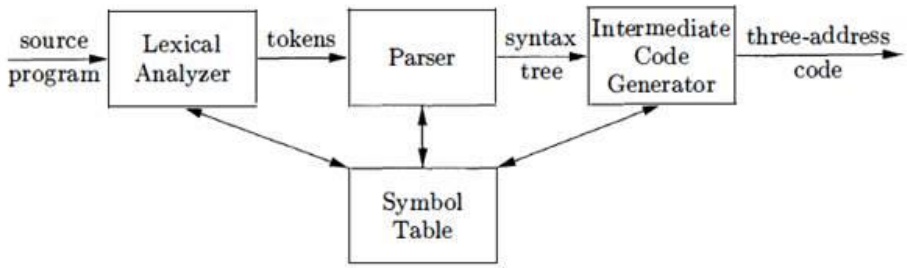
zpracovává příkazy jednotlivě a každý provede okamžitě po jeho přeložení (Python, Perl, JavaScript, Ruby, …)

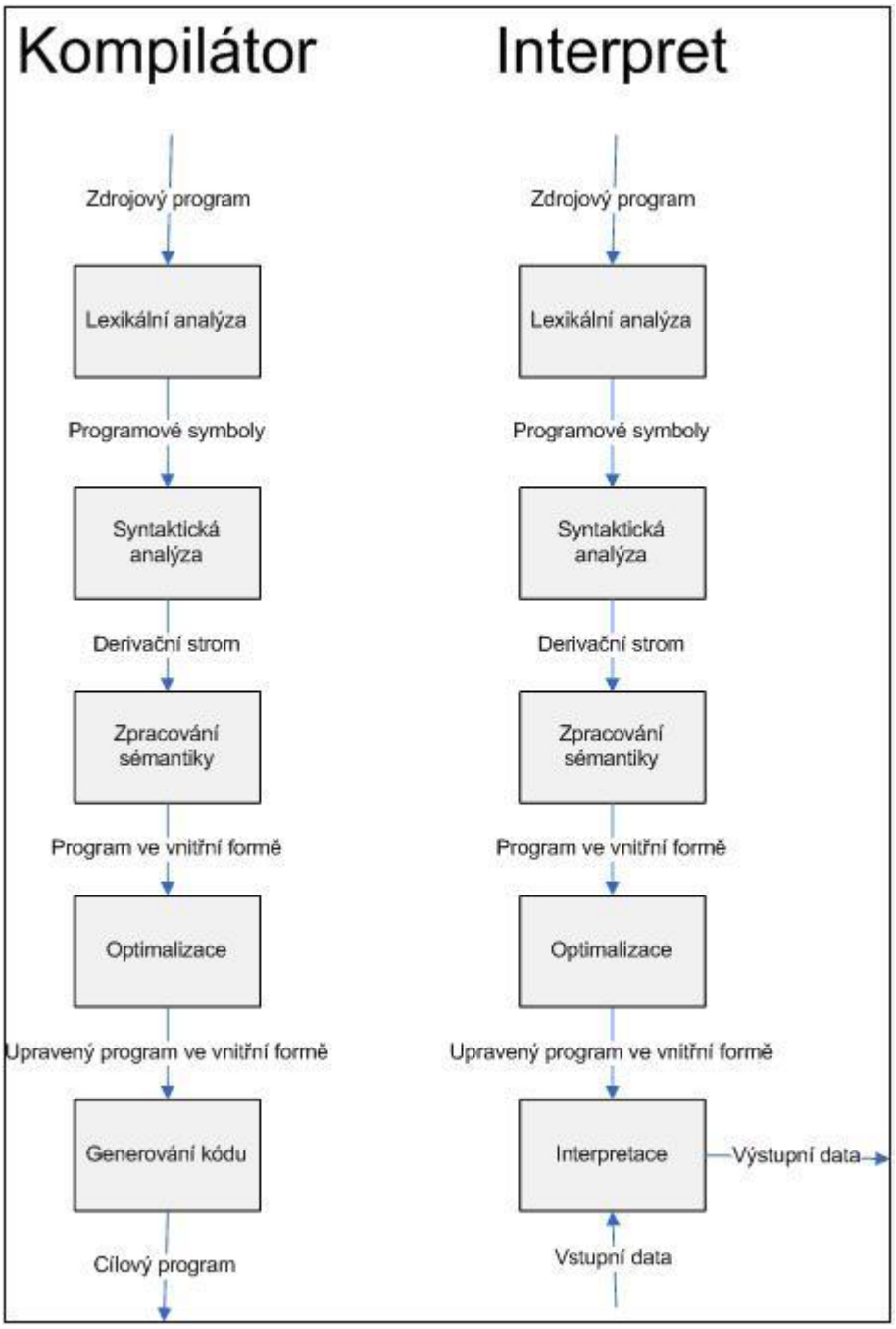
ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **interpretace**, pracuje se vstupními daty, aby vygeneroval výstupní data

Výhodou interpretu je:

- Eliminace kroků cyklu „editace → překlad → sestavení → exekuce“
- Snazší realizace ladících mechanismů (zachování původních jmen symbolů)

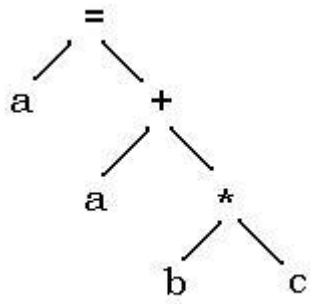




Lexikální analýza
zdrojový kód vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní *lexikální symboly* jako konstanty, identifikátory, klíčová slova nebo operátory. Je založena na regulárních gramatikách. Výsledkem je posloupnost symbolů, např. je na vstupu rozeznáno klíčové slovo begin a do posloupnosti lexikálních symbolů bude zařazen nový symbol reprezentující právě toto klíčové slovo. Tyto symboly jsou programem snadno použitelné a dále zpracovatelné. V této fázi se odstraňují veškeré komentáře.

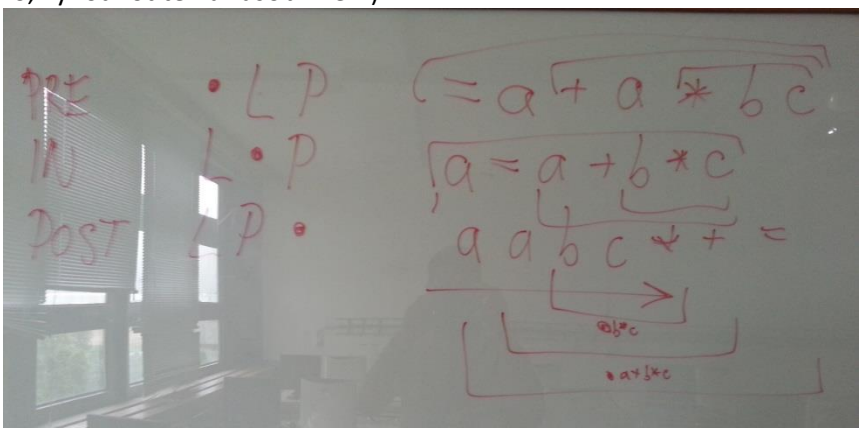
Syntaktická analýza
Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury (vnitřní jazyk překladače), které mají jako celek svůj vlastní význam, např. výrazy, příkazy, deklarace nebo

program. Programy jsou psány většinou v infixové notaci (a=a+b*c) => analyzujeme a vytváříme hierarchické uspořádání derivačního stromu:



Notace vnitřního jazyka překladače:

- Prefixová (nemá závorky, operátory bezprostředně *předchází* operandy a pořadí operandů je zachováno)
- Infixová
- Postfixová (nemá závorky, operátory bezprostředně *následují* operandy a pořadí operandů je zachováno, vyhodnotitelná zásobníkem)



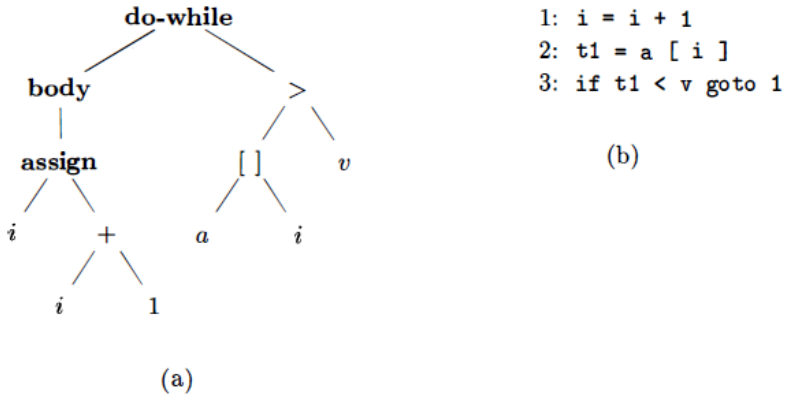


Figure 2.4: Intermediate code for "`do i = i + 1; while (a[i] < v);`"

Sémantická analýza

Provádějí se některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (např. kontrola deklarací, typová kontrola, apod.). Např. kontrola, jestli index pole je integer.

Typická reprezentace programu ve *vnitřní formě* (intermediate code) je sekvence trojic nebo čtveřic (3- nebo 4-adresových instrukcí)

Optimalizace
Optimalizátor kódu zajišťuje, aby se používalo co nejméně pomocných proměnných pro mezivýpočty, aby se v cyklu zbytečně několikrát nevyhodnocoval tentýž výraz, jestliže hodnota jeho prvků zůstává bez změny a vyhodnocení stačí provést jednou před cyklem, apod. Optimalizací prochází program obvykle v intermediálním tvaru – intermediální kód je již podobný cílovému programu, má však strukturu vhodnější pro optimalizaci. Může to být zápis podobný assembleru nebo třeba dynamická struktura (dynamický seznam stromů představujících jednotlivé příkazy).

Generování kódu
poslední fází překladače je generování cílového kódu, což je obvykle přemístitelný kód nebo program jazyka asembleru. Všem proměnným použitým v programu se přidělí místo v paměti. Potom se instrukce mezikódu překládají do posloupnosti strojových instrukcí, které provádějí stejnou činnost.

Vícefázový a víceprůchodový překladač

Fáze = logicky dekomponovaná část (může obsahovat více průchodů, např. optimalizace)

Průchod = čtení vstupního řetězce, zpracování, zápis výstupního řetězce – může obsahovat více fází

Jednoprůchodový překladač = všechny fáze probíhají v rámci jediného čtení zdrojového textu programu

- Omezená možnost kontextových kontrol
- Omezená možnost optimalizace
- Lepší možnosti zpracování chyb a ladění (tedy dobré pro výuku)

Na strukturu překladače mají vliv:
- Vlastnosti zdrojového a cílového jazyka
- Vlastnosti hostitelského počítače
- Rychlost/velikost překladače
- Rychlost/velikost cílového kódu
- Ladicí schopnosti (detekce chyb, zotavení)
- Velikost projektu, prostředky, termíny

Testování a údržba překladače
Díky formální specifikaci jazyka je možné automatické provádění testů

Systematického testování lze dosáhnout regresními testy, což je sada testů doplňovaná o testy na odhalené chyby. Po každé změně v překladači se provedou všechny testy a jejich výstupy se porovnají s předešlými.

3.3.2. Regulární gramatiky, regulární výrazy a konečné automaty

Obecně k jazykům a gramatikám
ABECEDA = neprázdná množina, její prvky se nazývají PÍSMENA
ŘETĚZEC = SLOVO = libovolná konečná posloupnost písmen abecedy
JAZYK = libovolná množina řetězců (= slov) nad abecedou

ABECEDA A={a,b,c} → SLOVO (ŘETĚZEC) aaab, abc, cb,.. →JAZYK L= {aaab, abc, cb}

GRAMATIKA definuje jazyk jako množinu všech řětězců (slov), které lze v gramatice odvodit.

Chomskeho hierarchie (místo „regulární“ má být asi „lineární“)

Regulární gramatiky

Gramatika G je čtveřice (N,Σ,P,S), kde:

- N je konečná množina neterminálních symbolů (neterminálů).
- Σ je konečná množina terminálních symbolů tak, že žádný symbol nepatří do N a Σ zároveň (jsou disjunktní).
- P je konečná množina odvozovacích pravidel. Každé pravidlo je tvaru

$$(\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

"cokoli poskládaný ze všech možnejch symbolů na cokoli"; S je prvek z N nazývaný počáteční symbol.

Lineární gramatika = bezkontextová gramatika, která má nanejvýš jeden neterminál na pravé straně. Regulární gramatika je speciálním případem lineární gramatiky, kdy všechny neterminály jsou na levém konci/straně (levá lineární = levá regulární) nebo ekvivalentně pro pravou stranu.

Regulární gramatika – je to gramatika typu 3 = lineární, navíc převedená do regulárního tvaru (podle Chomského hierarchie). Pravidla těchto lineárních gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se u pravé regulární gramatiky skládá z jednoho terminálu (u lineární i z více), který může být následován jedním neterminálem, tedy:

$$X \to wY$$
$$X \to w,$$

kde X, Y jsou neterminály a w je řetězcem terminálů. Regulární gramatiky se také nazývají **pravé lineární gramatiky**. Obdobně se definují i **levé regulární gramatiky**, které obsahují pravidla typu:

$X \to Yw$
$X \to w$

Pravé a levé gramatiky jsou ekvivalentní. Jazyky generované regulárními (=lineárními) gramatikami jsou právě jazyky rozpoznatelné konečným automatem.

Lineární gramatika = má na pravé straně právě jeden neterminál

Regulární gramatika = gramatika, která popisuje regulární jazyk, přesně definovaný tvar pravidel (B -> a, B -> aC, B -> e pro pravou regulární gramatiku)

Regulární gramatika je tedy buď jen levá lineární gramatika nebo jen pravá lineární gramatika. Čistě lineární (levo-pravá) gramatika je pak taková gramatika, která sestává z pravých i levých pravidel současně.

Regular languages are also characterized by special grammars called regular grammars whose productions take the following form, where w is a string of terminals.
A → wB or A → w.
Example. A regular grammar for the language of a*b* is
S → Λ | aS | T
T → b | bT.

Regulární výrazy

Regulární výrazy umožňují algebraické manipulace s regulárními množinami - umožňují **vyjádření regulárních množin**. Třída regulárních výrazů nad abecedou Σ je definována takto:

- *e* a $\varnothing$ jsou regulární výrazy
- každé písmeno (symbol - znak) $\sigma \in \Sigma$ je regulární výraz nad Σ
- jsou-li R_1 a R_2 regulární výrazy nad Σ, pak i $(R_1 + R_2)$, $(R_1 . R_2)$ a R_1^* jsou regulární výrazy nad Σ

Daná množina je regulární množina nad Σ, právě když může být popsána vhodným regulárním výrazem nad Σ. Každý regulární výraz **U** popisuje jistou množinu **Ũ** slov nad **Σ**: **Ũ** $\subseteq$ **Σ***

Regulární množiny se vhodně charakterizují přechodovými grafy. Přechodový graf T nad abecedou Σ je konečný orientovaný graf, jehož každá hrana je pojmenována jistým slovem $w \in \Sigma^*$;alespoň jeden uzel je počáteční.

Množinu všech slov akceptovaných konečným automatem **A** označíme **Ã**. Množina je regulární nad **Σ** právě když je akceptována vhodným automatem nad **Σ**

Regulární výraz je řetězec popisující celou množinu řetězců (slov), konkrétně regulární jazyk. Používají se nejčastěji v počítačových programech a skriptovacích jazycích pro vyhledávání a úpravu textu. V případě, že uživatel chce v textu vyhledat nějaký řetězec, který nezná přesně nebo který může mít více variant, může zadat regulární výraz, který postihne všechny chtěné varianty. Program tak nalezne všechny části textu, které danému výrazu odpovídají.

Každý z regulárních výrazů označuje jistý regulární jazyk.

Konečné automaty

Lidsky:
-konečný počet stavů
-konečný počet vstupů

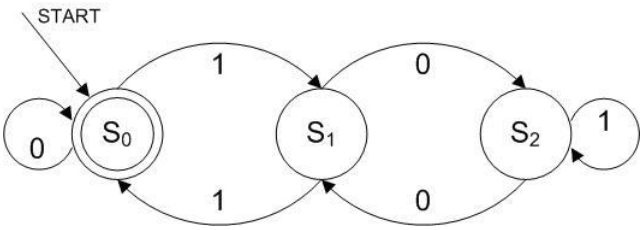
-jednosznačně určený následující stav
-jednoznačně určený počáteční stav

3 typy konečných automatů: Rozpoznávací (akceptuje / ne), Klasifikační, S výstupní funkcí, dale se mluví asi jenom o tom rozpoznávacím (finite automaton)

Formálně je konečný automat definován jako **uspořádaná pětice (S,Σ,P,s,F)** , kde:
S je konečná množina stavů.
Σ (*velké sigma*) je konečná množina vstupních symbolů nazývaná abeceda.
P je tzv. přechodová funkce (též přechodová tabulka), formálně zobrazení $\delta: S \times \Sigma \to S$, popisující pravidla přechodů mezi stavy. Přechod je určen stavem ve kterém se automat nachází a symbolem, který přichází na vstup (nebo který je čten na vstupu)
s je počáteční stav (s naleží S)
F je množina koncových (přijímacích) stavů (F je podmnožinou S)

Znázornit lze: Tabulkou, Stavovým diagramem (to jsou ty stavy a hrany), Stavovým stromem

Popis činnosti automatu: Na počátku se automat nachází v definovaném počátečním stavu. Dále v každém kroku přečte jeden symbol ze vstupu a přejde do stavu, který je dán hodnotou, která v přechodové tabulce odpovídá aktuálnímu stavu a přečtenému symbolu. Poté pokračuje čtením dalšího symbolu ze vstupu, dalším přechodem podle přechodové tabulky, atd. Podle toho, zda automat skončí po přečtení vstupu ve stavu, který patří do množiny koncových stavů, platí, že automat buď daný vstup přijal nebo nepřijal. Množina všech řetězců, který daný automat přijme, tvoří regulární jazyk.



Meze regulárních gramatik
Jak určit zdali je nějaký jazyk možné rozpoznat regulárním výrazem (konečným automatem, regulární gramatikou)? K tomuto lze použít tzv. Pumping teorém [wiki] nebo česky [abclinuxu].

Teorém vlastně říká, že v dostatečně dlouhém slově *w* daného regulárního jazyka můžeme nalézt tři části — *x*, *y* a *z*, přičemž nejdůležitější část *y* může zahrnovat i celé slovo. Aby byl tento jazyk regulární, musí platit, že část *y* můžeme ze slova vyjmout, nebo jí libovolně zopakovat, a přitom stále zůstáváme v rámci stejného jazyka.

3.3.3. Ekvivalence konečných automatů a regulárních gramatik

Lexikální symboly (léxémy anglicky tokens) jsou jsou regulární jazyk.

Regulární jazyk lze definovat gramatikou typu 3 nebo konečným automatem nebo regulárním výrazem.

Každou lineární gramatiku lze převést na regulární tvar.

Regulární gramatiky popisují všechny *regulární jazyky* a v tomto smyslu (ve schopnosti popisu jazyka) jsou ekvivalentní s konečnými automaty a regulárními výrazy. Regulární gramatika ja buď pravá regulární (neterminály jsou vpravo) nebo levá regulární (neterminály jsou vlevo).

Regulární jazyk je formální jazyk (množina (i nekonečná) slov složených z omezené abecedy), který:

- může být akceptován deterministickým/nedeterministickým konečným stavovým automatem
- lze popsat regulárním výrazem
- lze ho generovat regulární gramatikou

Příkladem **ne**regulárního jazyka je $a^n b^n$, kde n > 1 (alespoň jedno *a* následované stejným počtem *b*), gramatika pro palindromy apod. - *lze určit na základě Nerodovy věty, která se užívá v důkazech, že nějaký jazyk není rozpoznatelný konečným automatem*

Každý regulární jazyk je rozpoznatelný konečným automatem; každý jazyk rozpoznatelný konečným automatem je regulární.

Kleenova věta: Libovolný jazyk je regulární, právě když je rozpoznatelný konečným automatem. Přechodový graf je T nad **S** je konečný orientovaný graf, jehož každá hrana je pojmenována jistým slovem w∈**S***. Alespoň jeden z uzlů grafu je počáteční a některé uzly jsou koncové. Ke každému přechodovému grafu T nad abecedou S existuje regulární výraz R nad S takový, že

$$\tilde{R} = \tilde{T}$$

a ke každému regulárnímu výrazu R nad S existuje konečný automat A takový, že

$$\tilde{A} = \tilde{R}$$

Postup převodu gramatiky na konečný automat

Potřebujeme získat gramatiku typu 3 ve standardní formě.

Regulární gramatika je ve **standardní formě**, jestliže obsahuje pouze pravidla tvaru $X \rightarrow aY$ a $X \rightarrow a$, $X \rightarrow e$ kde *X,Y* jsou neterminály, *a* je právě jeden terminál, *e* je prázdný symbol. Toho dosáhneme takto:

- Původní gramatika typu 3 (lineární): G = (N, T, S, P)
- Požadovaná regulární gramatika: G' = (N', T, S, P')
- Požadovaná gramatika G' bude mít stejné terminální symboly a stejný počáteční stav.
- Konstrukce přechodů *P'*:
  - do *P'* zařadíme všechna pravidla z *P* ve tvaru $X \rightarrow aY$ a $X \rightarrow e$
  - za každé pravidlo $X \rightarrow x1\ x2\ x3\ Y$ zařadíme do *P'* soustavu pravidel:

- $X \rightarrow x1\ X1$
- $X1 \rightarrow x2\ X2$
- $X2 \rightarrow x3\ Y$
    - za každé pravidlo $X \rightarrow z1\ z2$, zařadíme do P' soustavu:
- $X \rightarrow z1\ Z1$
- $Z1 \rightarrow z2\ Z2$
- $Z2 \rightarrow e$

- Místo pravidel tvaru $X \rightarrow Y$ musíme zajistit to, aby z každého stavu X pro který máme $X \rightarrow Y$, bylo možné odvodit všechny řetězce, které lze odvodit z Y.
- N' vznikne obohacením N o všechny nově vytvořené neterminální symboly

Zkonstruujeme automat z nově vytvořené gramatiky

- stavy budou odpovídat neterminálním symbolům
- vstupy budou odpovídat terminálním symbolům
- přechodovou funkci zkonstruujeme na základě analogií
    - $X \rightarrow aY \leftrightarrow$ přechod ze stavu X do stavu Y při vstupu symbolu a
- počáteční stav bude odpovídat počátečnímu symbolu
- množinu koncových stavů určíme z pravidel $X \rightarrow e$

Tímto jsme získali **nedeterministický konečný automat**, který lze převést na **deterministický konečný automat.**

Regulární atributované a překladové gramatiky

Atributovaná gramatika AG = (G, Atributy, Sémantická pravidla) Atributy jsou přiřazeny symbolům gramatiky a sémantická pravidla jednotlivým přepisovacím pravidlům. Při aplikaci přepisovacího pravidla se provedou příslušná sémantická pravidla a vypočtou hodnoty atributů. Atributy vyhodnocované průchodem derivačním stromem zdola nahoru nazýváme syntetizované, shora dolů nazýváme dědičné.

Překladová gramatika PG = (N, T u D, P, S) Obsahuje disjunktní množiny T a D, vstupních a výstupních terminálních symbol

3.3.4. Nedeterministický a deterministický konečný automat

Deterministický konečný automat

Lidsky:
-konečný počet stavů
-konečný počet vstupů
-jednosznačně určený následující stav
-jednoznačně určený počáteční stav

Je uspořádaná pětice (S,Σ,P,s,F), kde:

- S je konečná množina stavů.
- Σ je konečná množina vstupních symbolů nazývaná abeceda.
- P je tzv. přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy.
- s je počáteční stav (s naleží S)
- F je množina koncových stavů (F je podmnožinou S)

Nedeterministický konečný automat

Nedeterministickým konečným automatem (NKA) *bez výstupu* nazýváme každou pětici

A = (Q, Σ, δ, S, F), kde:

- Q je konečná, neprázdná, množina stavů
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- δ (přechodová funkce) je zobrazení δ: QxΣ → P(Q). Kde P(Q) je potenční množina (množina všech podmnožin množiny Q včetně prázné množiny *e*)
- S je množina počátečních stavů (S náleží Q) - není jednoznačně určen počáteční stav
- F je množina koncových stavů (F náleží Q)

Oborem hodnot přechodové funkce jsou všechny podmnožiny množiny stavů

Formálně je definován podobně jako DKA, ale obsahuje prvky nedeterminismu:

1. nejednoznačně určený počáteční stav (může jich být více)
2. nejednoznačné přechody (při přijetí stejného vstupu lze přejít do více stavů)
3. e - přechody (přechod do stavu bez přijetí vstupního symbolu)

- chování NKA lze popsat sekvencí pozic (množina stavů, ve kterých se automat může nacházet), z nichž každá jednoznačně definuje, zda je zpracovaný řetězec akceptován či zamítnut
- pozic je konečný počet
- přechody mezi pozicemi jsou jednoznačné
- Nejdůležitější rozdíl mezi DKA a NKA je v tom, že výsledkem přechodové funkce není pouze jeden stav, ale množina stavů, která může být i prázdná
- to vše jsou vlastnosti DKA a proto **ke každému NKA existuje ekvivalentní DKA**

V případě nedeterministického konečného automatu (NKA) je vstupní slovo akceptováno (rozpoznáno,) pokud toto slovo může automat převést do některého z koncových stavů (množina F) z některého z počátečních stavů (množina S).

Převod NKA na DKA

1. Lineární gramatiku nejprve převedeme na regulární tvar (postup viz otázka [*Ekvivalence konečných automatů a regulárních gramatik*].
2. Pak zkonstruujeme nedeterministický konečný automat a z něho nakonec deterministický (jak viz dále).

   A) Hlavní myšlenka je taková, že **každý stav vytvořeného DFA odpovídá množině stavů NFA.**

   B) Nebo: Z nedeterministického automatu se vytváří **strom**, který již popisuje deterministický automat, popisující tentýž problém.

Postup převodu

Samotný převod stojí na myšlence, že pokud lze ze vstupního uzlu S přejít do uzlu A a do uzlu B, tak vytvoříme nový uzel, říkejme mu $[A,B]$. Tento uzel bude mít stejné vstupy a výstupy jako sjednocení uzlů A a B. Nyní tabulka převedeného automatu obsahuje dva uzly $\{S,[A,B]\}$. Postup opakujeme pro uzel $[A,B]$. Takto postupně projdeme všechny stavy nově vytvářeného deterministického automatu.

Koncovými uzly převedeného deterministického automatu budou takové uzly, které jsou nadmnožinou koncových uzlů původního automatu (měl-li původně automat výstupní uzel A, tak uzel $[A,X]$, který vznikl jako sjednocení uzlu A a uzlu X, bude také výstupní).

Tento postup zároveň eliminuje všechny stavy, do kterých se deterministická verze automatu nemůže vůbec dostat. Zároveň ale mohou vzniknout uzly, které mají totožné vlastnosti (vstupní a výstupní uzly, konečnost, vlastnost *být počátečním uzlem*). Tyto uzly můžeme po doběhnutí algoritmu ztotožnit.

Příklad

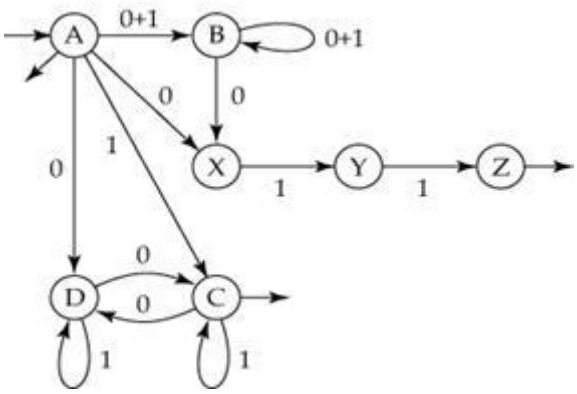
Zadaná pravá lineární gramatika:

```
A --> B | C
B --> 0B | 1B | 011
C --> 0D | 1C | e
D --> 0C | 1D
```

Pravá regulární gramatika:

```
A --> 0B | 1B | 0X | 0D | 1C | e
B --> 0B | 1B | 0X
X --> 1Y
Y --> 1Z
Z --> e
C --> 0D | 1C | e
D --> 0C | 1D
```

Nedeterministický konečný automat:



Deterministický konečný automat:

Přechodovou tabulku deterministického konečného automatu vytvoříme z přechodového diagramu nedeterministického kon. automatu takto:

- Do prvního řádku tabulky zapíšeme počáteční stav automatu a postupně zjistíme, do jakých množin stavů se nedeterministický automat může dostat z tohoto stavu přijmutím jednotlivých symbolů jeho vstupní abecedy.
- Z nalezených množin s více než jedním stavem vytvoříme tzv. *kompozitní* stavy det. automatu. Ty pak použijeme do přechodové tabulky det. automatu jako výstupy přechodové funkce pro počáteční stav a odpovídající vstupní symboly.
- Vzniklé kompozitní stavy (a případně i normální stavy) také využijeme v dalších řádcích přechodové tabulky a případně doplňujeme nové kompozitní stavy, do kterých se můžeme dostat z množin původních stavů každého kompozitního stavu přes vstupní symboly.
- Takto postupně vytvoříme celou přechodovou tabulku ekvivalentního deterministického automatu.
- Kompozitní stavy, zahrnující původní koncové stavy, můžeme označit také jako koncové.

	stav	0	1
<-->	A	BXD	BC
	BXD	BXC	BYD
<--	BC	BXD	BC
<--	BXC	BXD	BYC
	BYD	BXC	BZD
<--	BYC	BXD	BZC
<--	BZD	BXC	BD
<--	BZC	BXD	BC
	BD	BXC	BD

Nové stavy jsou A,BXD, BC, BXC, atd. Přechody 0,1.

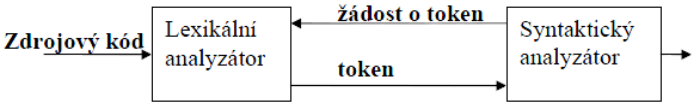
3.3.5. Lexikální analýza, princip činnosti

Úkoly lexikálního analyzátoru

- Čtení zdrojového textu,
- Nalezení a rozpoznání lexikálních symbolů ve volném formátu textu, včetně případného rozlišení klíčových slov a identifikátorů. Vyžaduje spolupráci s SA.
- Vynechání mezer a komentářů,
- Interpretace direktiv překladače,
- Uchování informace pro hlášení chyb,
- Zobrazení protokolu o překladu.

Lexikální analyzátor je samostatnou částí pro jednodušší návrh překladače, zlepšení efektivity překladu a lepší přenositelnost.

Lexikální analyzátor rozpoznává a zakóduje lexikální symboly jazyka = lexémy (anglicky tokens)



Lexikální symboly jsou regulárním jazykem.

Lexikální analýza

Je prováděna **lexikálním analyzátorem**, který je vstupní a nejjednodušší částí překladače. Čte znaky zdrojového programu, a jeho výstupem jsou **tokeny**. Vstupní posloupnost znaků - program - je slučována do lexikologicky smysluplných mnohoznakových jednotek, tzv. ***lexémů*** *(*např. if, foo123bar). Tokeny pak symbolicky reprezentují lexémy (např. *if* pro lexém klíčové slovo *if*, *id* pro identifikátor *foo123bar*) a lexémy jsou tak vlastně jejich instance. Podoba lexémů reprezentujících jednotlivé tokeny je vymezena **vzorem** (**pattern**), typicky regulárním výrazem.
Kromě toho je jeho úkolem odstranění komentářů a eliminace přebytečných bílých znaků.

Token Name	**popis (v podstatě pattern)**	**příklad lexémů**	**hodnota atributu**
if	znaky i, f	if	-
else	zn. e, l, s, e	else	-
id	písmeno násl. písm./číslicí	foo, score, myId	pointer do tab. ~~Záznamů~~ symbolů
number	jakákoliv číselná konstanta	3.14, 10	pointer do tab. záznamů
literal	vše v uvozovkách	“hello world”	pointer do tab. záznamů
relop	<, <=, >, >=, =, <>	<, <=, >, >=, =, <>	LT, LE, GT, GE, EQ, NE

Token je tvořen dvěma částmi – názvem tokenu (token name) a hodnotou atributu (attribute value). Názvy tokenu jsou často abstraktní symboly, které jsou pak použity parserem pro syntaktickou

analýzu. Jde např. o nějaké klíčové slovo nebo o soubor znaků představujících identifikátor. Operátory, klíčová slova a další ve skutečnosti atributové hodnoty nepotřebují. Pokud má token hodnotu atributu, jde o pointer do tabulky symbolů, která obsahuje dodatečné informace o tokenu, které nejsou součástí gramatiky.

Proud tokenů je předán parseru pro syntaktickou analýzu. Lexikální analyzátor také obvykle používá tabulku symbolů, do které ukládá objevené lexémy a ze kterých bere informace, aby mohl parseru podstrčit správný token. Jak název tokenu (typ - id, číslo,…), tak jeho atribut (číslo 0/1,…) ovlivňují rozhodování ve fázi parsování a pozdějších fázích. Parser proto potřebuje od analyzátoru dostat další token ke zpracování včetně informací z tabulky symbolů (viz obrázek níže).

V lexikální analýze mohou nastat nejednoznačnosti, pokud je jeden symbol prefixem jiného symbolu (== apod.). Pak se hledá nejdelší symbol a je vyžadována nápověda od syntaktického analyzátoru.

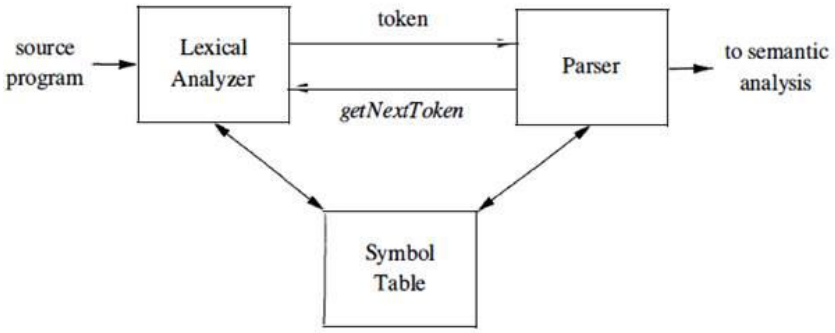


Figure 3.1: Interactions between the lexical analyzer and the parser

Často je potřeba dopředu skenovat vstup, aby se zjistilo, kde následující lexém končí. Proto lex. analyzátory typicky bufferují vstup.

Princip činnosti

Princip lexikálního analyzátoru – nalezení a rozpoznání lexikálního symbolu

Třídy symbolů:

- Identifikátory
- Klíčová slova (rezervované identifikátory)
- Celá čísla
- Jednoznakové omezovače
- Dvouznakové omezovače

?Nějaký jiny postup z přednášky

- Zpracování začíná prvým dosud nezpracovaným znakem ze vstupu,
- Zpracování končí, je-li automat v koncovém stavu a pro další vstupní znak již neexistuje žádný přechod
- Pro každou kategorii předpokládáme samostatný koncový stav,
- Neohodnocená větev se vybere, pokud vstupujícímu znaku neodpovídá žádná z ohodnocených větví

Nejednoznačnost v lexikální analýze

Nastává v případě, kdy jeden symbol je prefixem jiného symbol (== apod)

Pravidlo “hledej nejdelší symbol”

3.3.6. Konstruktory lexikálních analyzátorů

Lex

Program LEX (Lexical Analyzer Generator) slouží k tvorbě jiných programů, které mají cosi udělat se vstupním (textovým) souborem za pomoci lexikální analýzy. Tím se myslí analýza struktur, které se dají zapsat lineárními gramatikami, konečnými automaty nebo regulárními výrazy. Typické použití LEXu je dvojí: vytvořený program pracuje samostatně, nebo slouží jako vstupní filtr pro jiný (syntaktický) analyzátor, např. bison či yacc.

Lex umožňuje vytvořit lexikální analyzátor uvedením regulárních výrazů, které popisují vzory (patterns) pro tokeny. Vstupní notace pro Lex se nazývá *Lex language* a samotný nástroj je *Lex compiler*. Kompilátor Lexu transformuje vstupní vzory do přechodového diagramu (Jádrem toho všeho je *konečný automat*.) a generuje kód do souboru lex.yy.c, ve kterém je simulován přechodový diagram.

Vstupem Lexu je soubor, obvykle s koncovkou .l, např. *lex.l*, je napsaný v jazyce Lexu a popisuje lexikální analyzátor k vygenerování (rozpoznávaná slova a akce, které se mají po jejich rozpoznání provést). Slova (tokeny) se popisují regulárními výrazy, akce v cílovém programovacím jazyku. Výstup LEXu je zdrojový kód hotového programu, který se potom musí běžným způsobem přeložit. Pokud tedy používáme variantu LEXu, která generuje výstup v jazyku C, musí být i akce zapsané v jazyku C.

Lex kompilátor překlopí lex.l do programu v C, který je vždy uložen v souboru *lex.yy.c*. Pak je tento soubor zkompilován vždy do *a.out*. Výstupem je fungující lexikální analyzátor, který bere proud vstupních znaků a vytváří z nich proud tokenů.

Hodnoty atributů (= numerický kód/pointer do tabulky symbolů/nic) jsou umístěny v globální proměnné `yylval`, kterou sdílí lexikální analyzátor a parser.

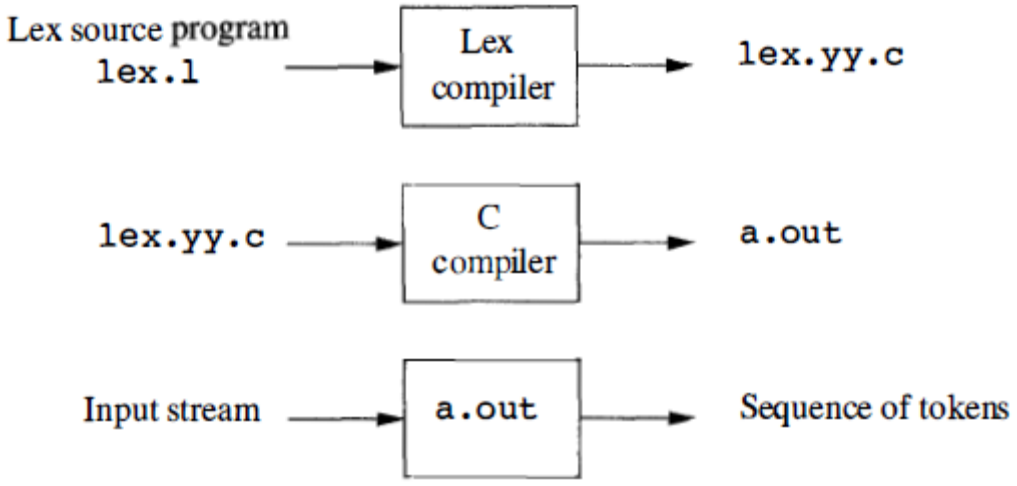


Figure 3.22: Creating a lexical analyzer with Lex

Struktura programu v Lexu (tohle je nějak zvláštně)

```
deklarace, definice
%%
popis slov a akcí
```

```
%%
další funkce (zapsané v cílovém jazyku)
```

Příklad: program, který ve vstupním souboru nahradí všechny identifikátory slovem "IDENTIFIKATOR"

```
%%
[a-zA-Z_][0-9a-zA-Z_]*    printf("IDENTIFIKATOR");
%%
int main(void)
{
  yylex();
  return 0;
}
```

V popisu rozpoznávaných slov lze používat následující konstrukce:

x	znak "x"
[xy]	znak "x" nebo "y"
[x-z]	všechny znaky od "x" až k "z"
[^x]	jakýkoliv znak vyjma "x"
.	jakýkoliv znak, až na novou řádku
^x	znak "x", pokud se nachází na začátku řádky
x$	znak "x", pokud se nachází na konci řádky
x*	libovolný počet znaků "x"
x+	alespoň jeden znak "x"
x?	jeden nebo žádný znak "x"
x{m,n}	M až n výskytů znaku "x"
x|y	znak "x" nebo "y"
(x)	znak "x"
x/y	znak "x", je-li následován znakem "y"
{DEF}	doplnění definice z úvodní sekce
<y>x	znak "x", je-li splněna podmínka y

Obecný tvar vstupního souboru pro Lex (z přednášky)

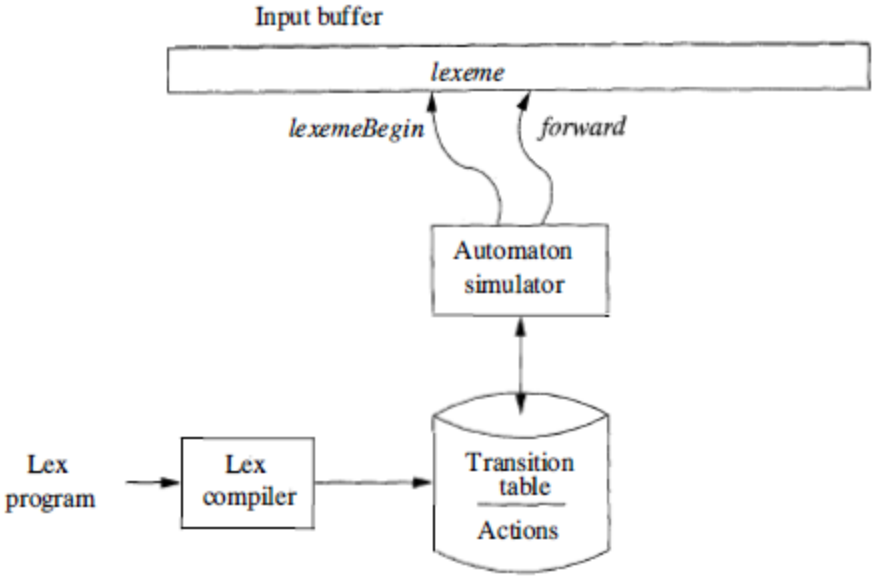
```
Obecný tvar vstupního souboru pro Lex:
      {definice použité v regulárních výrazech a C deklarace}
      %%
      {pravidla v podobě regulárních výrazů a příslušných akcí}
      %%
      {doplňkové procedury}

-Definice - zahrnují deklarace  proměnných,
                                konstant,
                                regulárních definic.

-Pravidla mají tvar -     p1     {akce1 v C notaci}
                          P2     {akce2   "        }
                                 …
                          pn     {akceN   „        }

            pi jsou regulární výrazy
            {akcei} jsou programové fragmenty

-Doplňkové procedury jsou pomocné, mohou obsahovat C rutiny
volané akcemi
```

Regulární výrazy Lexu (z přednášky)

```
Regulární výrazy v pravidlech mohou mít podobu:

je-li       c     jeden znak
            r     reg. výraz
            s     řetězec
            i     identifikátor
pak
výrazu      odpovídá                          např.
-------------------------------------------------
c           libov. neoperátorový znak c       a
\c          znak c literálně                  \*
"s"         řetězec s literálně               "***"
 .          libov. znak mimo nový řádek       a.*b
^           začátek řádky                     ^abc
$           konec řádky                       abc$
[s]         libov. znak z s                   [abc]
[x-z]       znaky x, y, . . . z.              [0-9]
[^s]         "    "  "  není-li z s           [^abc]
r*          nula nebo více r                  a*
r+          jeden nebo více r                 a+
r?          nula nebo jeden r                 a?
r{m,n}      m až n výskytů r                  a{1,5}
r1r2        r1 pak  r2                        ab
r1|r2       r1 nebo r2                        a|b
(r)         r                                 (a|b)
r1/r2       r1 je-li n sledováno r2           abc/123
{i}         překlad i z definiční sekce       {PISMENO}
-------------------------------------------------
```

```
yylval      proměnná pro předání tokenu do Yacc (ten provádí synt. analýzu)
yytext      proměnná  obsahující text odpovídajícího reg.výrazu
yyleng      "              "   počet znaků    "
yyless(n)   ubere n znaků z yytext[]
yymore()    přidá  k obsahu yytext[] další koresp. část textu
REJECT      přejde na další pravidlo bez změny obsahu yytext[]
```

Architektura lexikálního analyzátoru generovaného Lexem



Lex z definovaných regulárních výrazů ze vstupního souboru → NKA → DKA; pravidlo: v případě konfliktů přiřazuje lexém vzoru dle nejdelšího prefixu.

Další varianty

- Flex – volně dostupná implementace Lexu, pro C.
- JLex – volně dostupná implementace Lexu, pro Javu.
- C# LEX - varianta JLex pro C#.
- PLY - implementace Lexu v Pythonu

3.3.7. Bezkontextové gramatiky a zásobníkové automaty, formální popis, ekvivalence

Bezkontextové gramatiky

Bezkontextové gramatiky (BKG) jsou gramatiky typu 2 podle Chomského hierarchie.

Gramatika je bezkontextová, **tvoří-li levou stranu všech přepisovacích pravidel právě jeden neterminální symbol**

Skládají se tedy z pravidel $A \to \gamma$ kde A je právě jeden neterminál a γ je řetězec terminálů a neterminálů. Pravidlo $S \to e$ je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. jazyky generované touto gramatikou jsou **rozpoznatelné nedeterministickým zásobníkovým automatem**.

Skládají se z **terminalů, neterminálů, počátečního symbolu a přepisovacích (produkčních) pravidel; G = (N,T,P,S)**.

- T = Terminály – jde o názvy tokenů (klíčová slova *if*, *else*, symboly „(“, „)“ atd.)
- N = Neterminály – syntaktické proměnné, pomáhají definovat jazyk generovaný gramatikou; zavádějí hierarchickou strukturu jazyka, která je klíčová pro syntaktickou analýzu
- S = Počáteční symbol – jeden z neterminálů
- P = přepisovací pravidla - ve tvaru

  $A \to \gamma$, kde $A \in N$ a $\gamma \in N \cup T$

Příklad

Jazyk

$$L = \{0^n 1^n\}$$

pro

$$n \geq 0$$

, takovýto jazyk není rozpoznatelný konečným automatem, zásobníkovým ano („konečný automat neumí počítat“). U programovacích jazyků by to třeba znamenalo, že není možné závorkovat (vnořovat kód) do libovolné úrovně.

Pro tento jazyk by platilo:

$$N = \{S\}$$
$$T = \{0,1\}$$
$$P = \{S \to 0S1, S \to e\}$$
$$S = \{S\}$$

Zásobníkový automat

Formálně je zásobníkový automat definován jako **uspořádaná sedmice (Q,T,G,δ, q_0,z_0,F)**, kde:

- Q je konečná množina vnitřních stavů,
- T je konečná vstupní abeceda,
- G je konečná abeceda zásobníku,
- δ je tzv. přechodová funkce, popisující pravidla činnosti automatu (jeho program), je definováno jako zobrazení
  $Qx(T \cup \{e\})xG^* \text{ do } QxG^*$
- q_0 je počáteční stav,
- z_0 popisuje symboly uložené na počátku v zásobníku,
- F je množina přijímajících stavů,
  $F \subseteq Q$

.

Je vidět, že zásobníkový automat se v podstatě skládá z konečného automatu, který má navíc k dispozici potenciálně nekonečné množství paměti ve formě zásobníku. Obsah tohoto zásobníku ovlivňuje činnost automatu tím, že vstupuje jako jeden z parametrů do přechodové funkce.

Zásobníkový automat se od konečného automatu liší ve dvou směrech:

1. Využívá vršek zásobníku při rozhodování jaký přechod provést.
2. Může manipulovat se zásobníkem jako součást provádění přechodu.

Popis činnosti automatu

Na počátku se automat nachází v definovaném počátečním stavu a zásobník obsahuje pouze počáteční symboly. Dále v každém kroku podle aktuálního stavu, symbolů na vrcholu zásobníku a symbolu na vstupu provede přechod, při kterém může vyjmout ze zásobníku několik symbolů, vložit místo nich jiné a na vstupu přečíst další symbol. Toto se opakuje.

Po dokončení činnosti (po přečtení celého vstupu, pokud do té doby nedojde k chybě) je rozhodnuto, jestli automat vstupní řetězec přijal. K tomu mohou sloužit dvě kritéria:

- stav, ve kterém se na konci automat nachází, patří do množiny přijímajících stavů, nebo
- zásobník je na konci prázdný.

Obě definice jsou ekvivalentní, automaty na sebe lze vzájemně převádět (u druhé možnosti je možno z definice automatu zcela vypustit množinu přijímajících stavů).

Konfigurace automatu se dá popsat uspořádanou trojicí (q, w, alfa), kde q je vnitřní stav, w dosud nezpracovaná část vstupu a alfa obsah zásobníku. Na počátku práce je automat v konfiguraci (q0, w, z0).

Příklad akceptace řetězce zásobníkovým automatem: *viz cvičení 10 (LL gramatiky) na courseware FJP*

Vztah bezkontextových gramatik a zásobníkových automatů

Zásobníkové automaty jsou ekvivalentní bezkontextovým gramatikám: pro každou bezkontextovou gramatiku existuje zásobníkový automat, který generuje (akceptuje) identický jazyk generovaný touto gramatikou a naopak.

Pro danou BKG gramatiku W=(N, T, P, S) můžeme sestrojit zásobníkový automat P takový, že L(W)=L(P). Jsou dvě varianty:

1. Konstrukce zásobníkového automatu, který je modelem **syntaktické analýzy shora dolů**:
   - Q = {q} (automat má jen jeden vnitřní stav),
   - T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
   - G = N+T, tj. v zásobníku se může vyskytnout jakýkoliv symbol rozpoznávané gramatiky,
   - δ je dáno rozkladovou tabulkou,
   - q0 = q, počáteční stav automatu je q, neboť automat jiné stavy nemá,
   - z0 = S, tj. na počátku je v zásobníku startovací symbol gramatiky
   - F = {}, což se interpretuje jako "automat akceptuje vyprázdněním zásobníku".
   - *LL(k) gramatiky*
2. **Analýza zdola nahoru** je obecnější a vyžaduje trochu složitější automat:
   - Q = {q, r}, stav q je "pracovní", stav r "akceptační",
   - T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
   - G je v nejjednodušším případě rovno N+T+{#}, tj. sjednocení symbolů gramatiky a speciálního symbolu "#"; deterministický automat může mít množinu G složitější
   - δ je dáno rozkladovou tabulkou,
   - q0 = q,
   - z0 = #,
   - F = {r}.
   - *gramatiky: LR, SLR, LALR*

3.3.8. Nedeterministický syntaktický analyzátor

Při syntaktické analýze konstruujeme derivační strom. Podle toho, jak je konstruován derivační strom věty, rozlišujeme dvě základní metody syntaktické analýzy:

1. **metoda shora dolů:**

   derivační strom konstruujeme od kořene k listům a zleva doprava (provádíme levou derivaci)

2. **metoda zdola nahoru**:

   postupujeme od listů směrem ke kořeni, ale také zleva doprava (provádíme pravou derivaci)

K syntaktické analýze se využívají zásobníkové automaty (ZA), které jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

- Deterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
- Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický (pohled do zásobníku nebo dále do vstupního řetězce).

Obecný popis nedeterminismu a determinismu

Základem **nedeterminismu** je tedy vždy problém **výběr správného pravidla**, ať už analýzou shora dolů nebo zdola nahoru. Pokud se nemá analyzátor podle čeho rozhodnout, prostě **prochází prostor všech řešení** buď do šířky nebo do hloubky (backtracking) a hledá to správné řešení. Tato metoda je tedy značně **neefektivní**, protože v nejhorším případě může projít všechny možnosti a nenajít žádné správné řešení, tedy správnou množinu pravidel, jejichž expanzí/redukcí lze dosáhnout požadovaného výsledku.

V případě, že chceme analyzovat vstup **deterministicky**, musíme analyzátor **zdokonalit**. Ten musí mít přesnou informaci o tom, jaké pravidlo gramatiky v danou chvíli použít. Automat může využít informaci o **dosud provedené částečné derivaci** a také o **vstupu**, který ještě nebyl zpracován. Zásobníkovému automatu, který je abstraktním modelem syntaktického analyzátoru, tedy připravíme rozkladovou tabulku (lookup table), ve které bude určeno, jaké pravidlo má analyzátor použít podle vstupu a stavu zásobníku.

Metoda shora dolů

Derivační strom konstruujeme od kořene (ohodnoceného startovním symbolem) dolů k listům, zleva doprava podle levé derivace. Jedná se o zásobníkový automat LL. Počáteční konfigurace automatu se dá popsat uspořádanou trojicí **(q, w, alfa)**, kde:

q =vnitřní stav
w = dosud nezpracovaní část vstupu
alfa = obsah zásobníku
Na počátku práce je automat v konfiguraci **(q_0, w, z_0)**, napr. (q, abaaab, s).

Pokud jen generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Naším úkolem však bývá spíše analýza již existující věty. Zde již náhoda nepřipadá v úvahu, protože posloupnost pravidel pro levou derivaci již nemusí být jednoznačná. Potřebujeme automat, který tuto analýzu provádí, a tento automat musí mít možnost jednoznačně vybírat mezi pravidly to správné.

Existují dva postupy analýzy (nedeterministická a deterministická):

- **analýza s návratem** (nedeterministická)
    - postupně zkoušíme vhodná pravidla. Nejdřív první, pokračujeme dále ve výpočtu, a když se ukáže, že pravidlo nevyhovuje (dostaneme se do slepé uličky), vrátíme se zpátky a vyzkoušíme druhé pravidlo atd. Tato metoda je sice účinná. ale zbytečně pomalá.
    - Rekurzivní sestup.
- **deterministická analýza**
    - při výběru pravidla se řídíme dalšími informacemi. Může to být *pohled do budoucnosti*, kdy se díváme dále do vstupní posloupnosti symbolů a řídíme se tím, co později dostaneme na vstupu. Nebo například kontrolujeme obsah zásobníku (nestačí nám pouze vidět ten symbol, který ze zásobníku vyjímáme, ale i další, které jsou pod ním).
    - LL parsery

Metoda zdola nahoru

- Konstruujeme derivační strom zdola od listů nahoru ke kořeni, přičemž postupujeme zleva doprava.
- Stejně jako u první metody i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci - je to proces nalezení pravého rozkladu věty (LR gramatika).
- I zde musíme rozhodovat, která pravidla chceme použít. Tentokrát však nejde o pravidla se stejnou levou stranou (pro stejný neterminál), ale rozhodujeme se mezi pravidly, která mají podobnou pravou stranu a jsou proto použitelná pro tentýž podřetězec větné formy.

Řešíme to podobně jako u předchozí metody:

- **analýza s návratem** (nedeterministicky)
    - Vybereme ve větné formě jeden podřetězec (jako první vybíráme ten, který začíná nejvíc nalevo, je co nejdelší a je shodný s pravou stranou některého pravidla), přepíšeme neterminálem na pravé straně pravidla a pokračujeme v konstrukci derivačního stromu.
    - Pokud zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec atd. Tato metoda je příliš časově náročná.
- **deterministická analýza (LALR, SLR)**
    - Využíváme další informace získané při překladu, např. obsah nepřečtené části vstupního kódu nebo obsah zásobníku.

3.3.9. Derivace a derivační strom, víceznačnost gramatiky

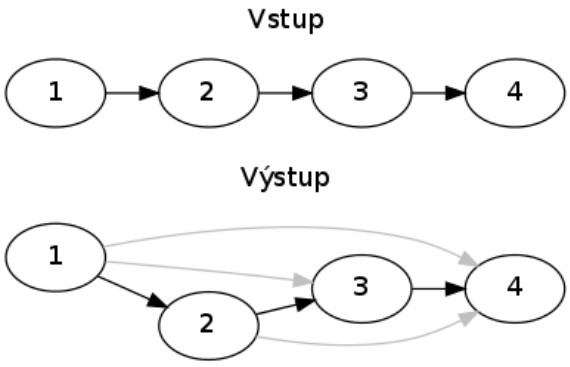
Derivace

- Posloupnost kroků odvození terminálu pomocí přepisovacích pravidel gramatiky
- Derivační pohled odpovídá konstrukci parsovacího (syntaktického) stromu shora dolů (top-down).
- Parsování zdola nahoru (bottom-up) je spjato s pravými derivacemi.
- Podle toho, který neterminál nahradit v každém kroku derivace, se rozlišují **levá a pravá derivace**.

DERIVACE řetězce α je posloupnost kroků odvození α pomocí přepisovacích pravidel gramatiky

$S = \alpha_1 => \alpha_2 => \ldots => \alpha_n = \alpha$

Dtto $S =>^* \alpha$ pozn.: $=>^*$ je uzávěr relace => (všechny přechody kam se dá transitivně dostat)



PŘÍMÁ DERIVACE: α A β => α γ β ,kde A -> γ ϵ P (pozn.: P je množina pravidel, pomocí kterých lze odvodit jazyk.)

Derivační strom

Derivační strom (parse tree) je orientovaný acyklický graf a je grafickou reprezentací, která říká, v jakém pořadí byla přepisovací pravidla uplatňována na neterminály, tedy jak vznikla věta jazyka.

- Kořen stromu je označen startovacím symbolem gramatiky
- Každý vnitřní uzel je ohodnocen neterminálními symboly.
- Listy jsou ohodnoceny terminálními symboly.
- Listy se čtou zleva doprava a dávají větu (generovanou gramatikou).
- Jestliže uzly $n_1, n_2, \ldots, n_k$ jsou bezprostřední následníci uzlu n, jsou ohodnoceny symboly $A_1, A_2, \ldots, A_k$ a uzel *n* je ohodnocen A, pak v množině pravidel gramatiky existuje pravidlo $A \rightarrow A_1A_2 \ldots A_k$.
- Není třeba značit orientaci hran.

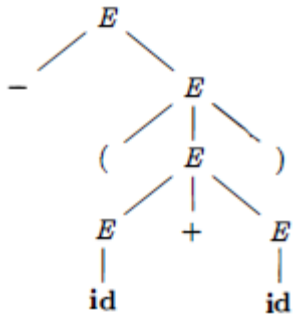
Jiná definice jazyka generovaného gramatikou je množina vět, které mohou být vytvořeny derivačním stromem. Proces hledání derivačního stromu pro danou větu (řetězec terminálů) se nazývá **parsování** tohoto řetězce

Derivační strom ignoruje variace v pořadí, v jakém jsou symboly přepisovány. Proto je mezi derivacemi a derivačními stromy vztah 1:N – např.:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Jsou různé derivace, jejich derivační strom ale vypadá stejně:



Pro získání jednoznačného derivačního stromu pro derivaci se proto užívá buď pravá a nebo levá derivace.

Víceznačnost gramatik

Gramatika, která generuje větu, pro níž lze sestavit aspoň dva různé derivační stromy, je víceznačná. Jinak řečeno: je to taková gramatika, která produkuje více než jednu levou nebo víc než jednu pravou derivaci pro tutéž větu.

Příklad: Pro gramatiku

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

umožňuje vytvořit dvě levé derivace pro id + id * id (násobení není upřednostněno před sčítáním):

$$\begin{aligned} E &\Rightarrow E+E \\ &\Rightarrow \mathbf{id}+E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E*E \\ &\Rightarrow E+E*E \\ &\Rightarrow \mathbf{id}+E*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*E \\ &\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id} \end{aligned}$$

- Nutnou podmínkou jednoznačnosti gramatiky je, aby pro žádný neterminální symbol neexistovalo jak pravidlo rekurzivní zprava, tak i pravidlo rekurzivní zleva
- Problém nejednoznačnosti bezkontextových jazyků je algoritmicky nerozhodnutelný.

Je potřeba buďto vytvořit jednoznačné gramatiky pro kompilaci aplikací, nebo u nejednoznačných gramatik zavést dodatečná pravidla, která řeší případné nejednoznačnosti.

Odstranění levé rekurze

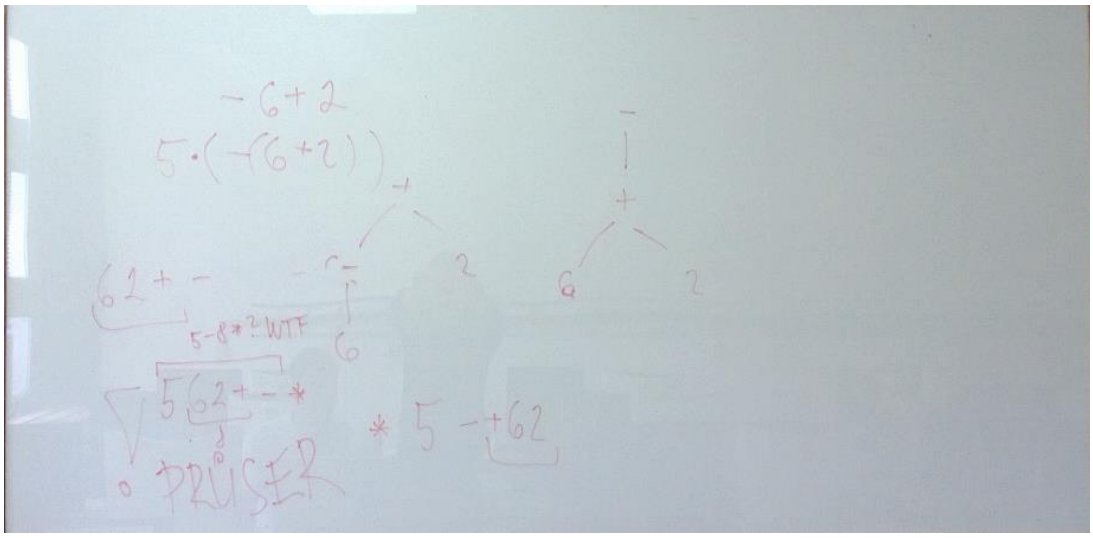
- **Levorekurzivní gramatiku nelze použít k analýze shora dolů**

Odstranění pravidla rekurzivního zleva:

- Nechť je dána BKG G = (N, T, P, S), ve které,
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
    jsou všechna A pravidla v P a žádné z β nezačíná A.

- Pak G' = (N ∪ { A'}, T, P', S), kde P' obsahuje místo uvedených pravidel pravidla:
  - $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

  $A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$



3.3.10. **Deterministická syntaktická analýza**

Je k ní zapotřebí **prediktivní parser** = rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Deterministická syntaktická analýza využívá další informace získané při překladu – obsah zásobníku a obsah nepřečtených vstupních tokenů. Na základě těchto informací umožňují následující funkce vhodný výběr přepisovacích pravidel, a tím prediktivní parsování:

FIRST(A) = množina terminálů, kterými mohou začínat řetězce odvozené z A **(terminály na začátku A)**

Funkce first zjišťuje, co vznikne přepsáním jednotlivých neterminálů na levé straně všech pravidel.

- A => bxayB → b náleží FIRST(A)

Algoritmus výpočtu

- FIRST(A) pro A = terminál nebo *e:* je terminál/*e*
- Když je A neterminál:
  - Je to první terminál ve všech přepisovacích pravidlech s A na levé straně
  - Pokud jsou v přepisovacím pravidle na P straně jen neterminály, hledá se FIRST prvního neterminálu na pravé straně
  - Pokud lze nějaký z těchto neterminálů přepsat na prázdný řetězec *e*, je potřeba se podívat na *first* neterminálu následujícího po něm v některém z přepisovacích pravidel

FOLLOW(A) = množina terminálů, které mohou následovat za A v některé větné formě v derivacích **(terminály hned za A)**

- S => bxCAyZ → y náleží FOLLOW(A)

Algoritmus výpočtu

1. Polož FOLLOW (A) = θ

2. Je-li A počáteční symbol G, přidej e do FOLLOW(A)

3. Pro všechny pravé strany pravidel z G tvaru α A β přidej FIRST (β) do FOLLOW (A), nepřidávej ale e.

4. Je-li v G pravidlo L → α A nebo L → α A β , kde FIRST (β)

obsahuje e, pak přidej do FOLLOW (A) množinu FOLLOW (L)

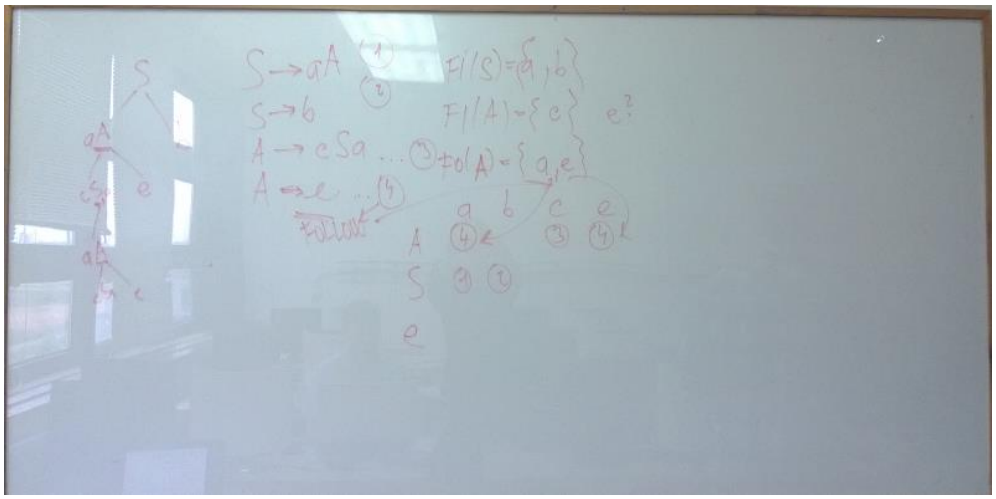
Vytváříme vždy pro všechny neterminály zároveň!

$FIRST_k(A)$, $FOLLOW_k(A)$ = zobecnění na množiny terminálních řetězců o délce nejvýše *k*

Tyto funkce slouží k vytvoření **rozkladové tabulky**, která nahrazuje přechodovou funkci. V první řádce jsou uvedeny všechny možné vstupy, v prvním sloupci všechny možné stavy vrcholu zásobníku (vč. dna zásobníku #). Má stavy:

- **Srovnání (pop)** – na vstupu i na vrcholu zásobníku jsou stejné hodnoty
- **Přijetí (accept)** – bylo dosaženo dna zásobníku a přijímá se prázdný symbol *e*
- **expanze (expand)** – aplikace přepisovacího pravidla, které je uvedené v buňce tabulky určené vstupem (který sloupec) a vrcholem zásobníku (který řádek)
- **chyba (error)** – pokud pro vstup a hodnotu na vrcholu zásobníku je buňka v tabulce prázdná → vstupní řetězec není větou jazyka

Syntactic Analysis in terms of **bottom up algorithms: LR, SLR, LALR**



3.3.11. Rekurzivní sestup

Rekurzivní sestup nebo také **rekurzivní sestupný parser** postupuje shora dolů a je sestaven ze vzájemně se volajících procedur. Každá taková procedura obvykle implementuje jedno přepisovací pravidlo gramatiky. (Kromě něj do top-down parserů patří prediktivní parsery založené na LL(k) gramatikách.)

Prediktivní parser je rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Rekurzivní sestup s návratem je technika určování použitého produkčního pravidla zkoušením všech pravidel. Není limitován na LL(k) gramatiky, ale nemá zaručeno skončit, pokud gramatika není LL(k). Může vyžadovat exponenciálni čas pro svůj běh.

Princip

Hlavní myšlenka je taková, že pro každý neterminál gramatiky je implementována příslušná funkce v programu.

- každému neterminálnímu symbolu *A* odpovídá procedura *A*
- tělo procedur je dáno pravými stranami pravidel pro *A*
- pravé strany musí být rozlišitelné na základě symbolů vstupního řetězce
- je-li rozpoznána pravá strana, pak v případě neterminálního symbolu vyvolá *A* proceduru pro rozpoznaný neterminální symbol, v případě terminálního symbolu, ověří *A* jeho přítomnost ve vstupním řetězci a zajistí přečtení dalšího znaku ze vstupu
- rozpoznané pravidlo analyzátor oznámí (např. jeho číslo)
- chybnou strukturu vstupního řetězce oznámí chybovým hlášením

Terminál na pravé straně je porovnán s dalším vstupním symbolem. Pokud se shodují, přejde se na další vstupní symbol a na další symbol na pravé straně. V opačném případě je nahlášena chyba.

O neterminál na pravé straně je postaráno voláním příslušné funkce. Po jejím vykonání se pokračuje dalším symbolem na pravé straně.

Pokud na pravé straně už nejsou žádné symboly, funkce končí (function returns).

Takto se postupně volají všechny funkce, až se nakonec opět ocitneme ve funkci pro startovací symbol, která byla zavolána jako první. Ta také oznamuje úspěšný průběh, pokud se prošel celý vstupní řetězec.

```
    void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Pseudokód výše je nedeterministický, protože začíná volbou A-přepisovacího pravidla, které se používá blíže nepopsaným způsobem.

Příklad: http://info.lu2.name/soubory/prekl_06_604.pdf

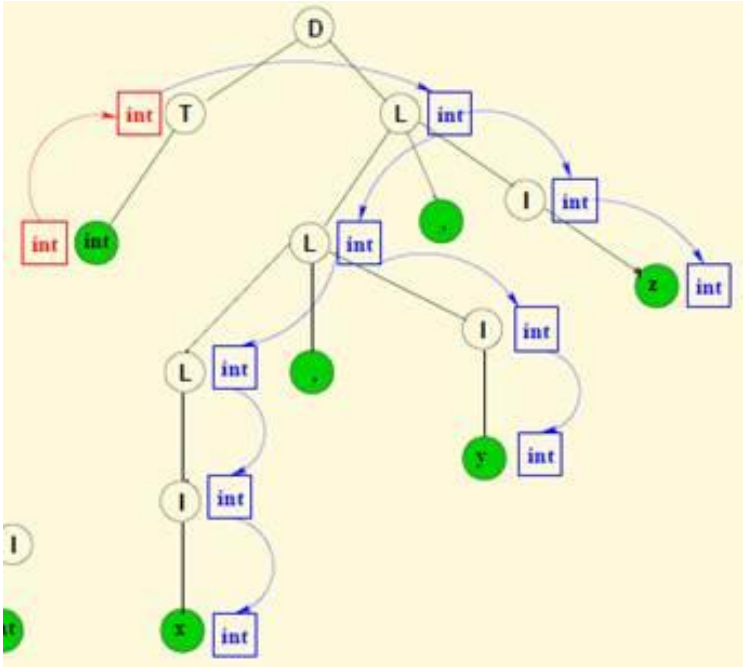
Obecně může rekurzivní sestup potřebovat backtracking, tzn. někdy je třeba se vrátit a opakovaně číst vstup. Backtracking je však potřeba zřídkakdy. Kód výše neumožňuje backtracking, bylo by je nutno modifikovat – v řádce 7 se pak vrátit na řádku 1 a zvolit jiné pravidlo, popř. nahlásit chybu, když už žádné další nejde použít.

Sémantické zpracování

Při rekurzivním sestupu se může provádět také sémantické zpracování. Sémantické zpracování zahrnuje vyhodnocení atributů symbolů v derivačním stromu. Atributy = vlastnosti gramatických symbolů nesoucí sémantickou informaci (hodnota, adresa, typ, scope, spojitost mezi formálními a skutečnými parametry apod.).

Způsoby vyhodnocení:

1. procházením stromem od listů ke kořenu = **syntetizované atributy**
2. procházením stromem od rodiče k potomkovi, od staršího bratra k mladšímu = dědičné atributy (např vícenásobné deklarace v C – int x,y,z)



Je nutné doplnit procedury lex. analýzy (LA) i syntakt. analýzy (SA) takto:

- LA bude předávat s přečteným vstupním symbolem i jeho atributy.

- procedury SA pro neterminály doplnit o:
  - vstupní parametry odpovídající dědičným atributům
  - výstupní parametry odpovídající syntetizovaným atributům
  - zavést lokální proměnné pro uložení atributů pravostranných symbolů
  - před vyvoláním procedury korespondujícího neterminálu z pravé strany vypočítat hodnoty jeho dědičných atributů
  - na konec procedury popisující pravou stranu pravidla zařadit příkazy vyhodnocující syntetizované atributy

Vlastnosti

- pro metodu rekurzivního sestupu, tj. analýza shora dolů, se používají *LL* gramatiky
- jednoduchá *LL* gramatika je taková gramatika, kde levou stranu tvoří právě jeden neterminální symbol a kde každá pravá strana začíná terminálním symbolem
- navíc musí platit, že např. pro pravidla *A->...* jsou počáteční symboly různé
- obecná *LL* gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

3.3.12. Principy a podmínky LL analýzy

LL-gramatika

LL gramatika je jakákoliv gramatika, z níž se dá udělat rozkladová tabulka pro LL parser. **LL(k) parser** se kouká při parsování věty na následujících *k* tokenů, aby věděl, co dál. Pokud takový parser může být použit pro nějakou gramatiku, aniž by se musel použít backtracking, jedná se o **LL(k) gramatiku**.

Aby se ze vstupní gramatiky dala udělat LL(1) gramatika – eliminace levé rekurze, levá faktorizace (eliminace překrývajících se množin FIRST –

př: STAT => if EXP then STAT | if EXP then STAT else → STAT => if EXP then STAT ElsePart; ElsePart => else STAT | e)

Podmínky

- Nesmí být přítomna levá rekurze.
- Nesmí dojít k first-follow (u neterminálu, který se přepisuje na "e") kolizi, first-first kolizi

Třídy jazyků LL(k)

L = Left to right -> vstupní text (soubor) čteme zleva doprava

L = Left parse -> vytváříme levý rozklad

K = při rozhodování mezi pravidly potřebujeme vidět nejvýše *k* znaků z nepřečtené části vstupu

Tzn.: **LL(k)** gramatika provádí deterministický rozbor *čtením textu z* ***L****eva doprava*, s *použitím* ***L****evé derivace* a *prohlédnutí* ***k*** *dalších symbolů vstupního textu*.

- gramatika je typu **LL(k)**, jestliže ji lze použít pro deterministickou syntaktickou analýzu metodou shora dolů (tj. vytváříme levý rozklad) a při rozhodování mezi pravidly potřebujeme znát nejvýše **k** symbolů ze vstupu.
- jazyk je typu **LL(k)**, pokud je generován některou **LL(k)** gramatikou

LL(0) gramatika

- lze určit správné pravidlo aniž bychom předem potřebovali vidět nějaký znak na vstupu
- každý neterminál musí mít jen jednu jedinou pravou stranu (jen jedno přepisovací pravidlo)
- neumožňuje rekurzi
- prostě jen určují, jestli sekvence patří do jazyka nebo ne, žádné rozhodování není potřeba.
- LL(0) gramatiky jsou nevhodné pro popis programovacích jazyků, protože zde není možná rekurze a pro každý neterminál existuje právě jedno pravidlo (důsledek faktu, že gramatika se nemůže rozhodnout podle následujícího vstupního symbolu), tudíž mohou generovat jen jazyk s jediným slovem.
  From <http://cs.wikipedia.org/wiki/LL_syntaktick%C3%BD_analyz%C3%A1tor>

- Příklad:
```
G == id name lastname
id == [0-9]+
name == string
```

```
lastname == string
string == <unicode>+
```

LL(1) gramatika

- pro danou gramatiku *G* se vystačí při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky 1 -> proto LL(1) je **gramatika silná**
- **jednoduchá LL (1) gramatika** je taková bezkontextová gramatika jestliže platí:
  - pravá strana každého pravidla začíná terminálním symbolem např. A -> aB
  - pokud mají 2 pravidla stejnou levou stranu, pak pravé strany začínají různými terminálními symboly např. A -> aB, A->bB
  - to znamená, že v každém políčku rozkladové tabulky bude právě jeden element
- **Obecná LL(1):**
  - gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Mohutnosti gramatik

Typy analýzy

- shora (top-down)
- sdola (bottom-up) (vyžadují LR gramatiku, takže se netýkají této otázky)

Analýza shora = analýza top-down

- Při hledání derivace začínáme počátečním symbolem a snažíme se dostat k hledanému slovu
- LL analýza: hledáme levou derivaci, vstupní slovo analyzujeme zleva
  - Přesně určuje volbu pravidel při analýze a umožňuje jednoznačný postup při odvození
  - Gramatika, která je jednoznačná a lze ji takto analyzovat: LL gramatika
  - Využívá se zásobníkový automat

- LL(k) označení gramatiky pro LL analýzu, číslo k určuje, kolik následujících symbolů na vstupu je nutné znát pro analýzu slova
- LL(1): nejpoužívanější gramatika, stačí znát jeden následující symbol
- LL(0): umožňuje jen jazyky s konečným počtem slov
- LL gramatiky s k>1 lze převést na LL gramatiky s k = 1
    - Existují přesné popisy, jak jednotlivá pravidla nahrazovat (přidávají se neterminály a pravidla se upravují, aby při analýze stačilo znát jeden další symbol)

LL parsery = parsery s analýzou top-down

LL parsery používají parsing **shora dolů**, zpracovávají vstup zleva doprava a konstruují nejlevější derivaci. Proto se také nazývá L (left-to-right) L(leftmost derivation). Občas se setkáváme s označením LL(k), kde k značí počet tokenů, které potřebujeme znát při rozhodování o průběhu další analýzy bez toho, aby bylo třeba používat backtracking (= prediktivní parser). Také se v této souvislosti používá pojem look-ahead. Prakticky do nedávné doby se tyto gramatiky příliš nepoužívaly, ovšem na počátku 90. let minulého století došlo ke změně přístupu.

Syntaktická analýza LL gramatik

Budeme se zabývat algoritmem syntaktické analýzy, který vytváří derivační strom analyzovaného řetězce směrem shora dolů. Základní princip syntaktické analýzy můžeme v tomto případě formulovat takto:

Je dána bezkontextová gramatika G= (N,T,P,S) a řetězec w = a_1 a_2 … a_n, který je větou z L(G). Pak existuje levá derivace

S = $\gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n$ = w.

Vzhledem k tomu, že derivace je levá, má každá větná forma γi tvar:

γi = a_1 a_2 … a_j A_i β_i,

kde a_1, a_2 …, a_j jsou terminální symboly, A_i je neterminální symbol, β_i je řetězec terminálních a neterminálních symbolů. Přitom řetězec a_1a_2 … a_j je předponou věty w, j ≥ 0.

Podmínky LL analýzy

Předpokládejme, že A → α_1 | α_2 | … | α_n jsou všechna pravidla v P s neterminálním symbolem A na levé straně. Pak základní problém syntaktické analýzy metodou shora dolů spočívá v nalezení toho pravidla A → α_k, jehož aplikací dostaneme z větné formy γ_i větnou formu γ_i + 1.

Pro výběr pravidla A → α_k, je možno použít:

1. informaci o dosavadním průběhu (historii) analýzy,
2. informaci o dosud nepřečtené části vstupního řetězce (dopředu prohlíženém řetězci omezené délky).

Pokud tyto informace vždy stačí k jednoznačnému výběru pravidla A → α_k, pak se gramatika G nazývá *LL gramatika*. Název je odvozen od toho, že při čtení vstupního řetězce zleva je vytvářen levý rozklad. Při syntaktické analýze LL gramatik jsou do zásobníku ukládány řetězce, které odpovídají levým větným formám nebo takovým jejich příponám, které vzniknou odejmutím předpony tvořené řetězcem terminálních symbolů.

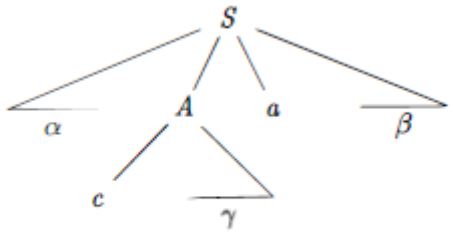
Základními operacemi syntaktického analyzátoru pro LL gramatiky (LL analyzátoru) jsou:

- **Expanze** – neterminální symbol na vrcholu zásobníku je nahrazen pravou stranou vybraného pravidla
- **Srovnání** – terminální symbol na vrcholu zásobníku se ze zásobníku vyloučí, jestliže je shodný se symbolem, který byl ze vstupního řetězce přečten.
- **Přijetí** – vstupní řetězec je přečten a zásobník je prázdný.
- **Chyba** – ve všech ostatních případech.

Pokud pro danou gramatiku G vystačíme při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky nejvýše *k*, pak se gramatika G nazývá *silná LL(k) gramatika*. Při analýze silných LL(k) gramatik jsou do zásobníku ukládány přímo symboly gramatiky a syntaktický analyzátor je řízen rozkladovou tabulkou.

Funkce FIRST a FOLLOW

Konstrukce jak top-down, tak bottom-up parserů používá dvě funkce, FIRST a FOLLOW, spojené s gramatikou G. Při parsování shora dolů nám FIRST a FOLLOW říkají, které přepisovací pravidlo uplatnit v závislosti na dalším vstupním symbolu. Během zotavení z chyby při panic módu mohou být množiny tokenů získané pomocí FOLLOW použity jako synchronizační tokeny.



Terminal c is in FIRST(A) and a is in FOLLOW(A)

Popis LL gramatik a LL analyzátoru

Algoritmus
Výpočet funkce FOLLOW

Vstup: Bezkontextová gramatika $G=(N,T,P,S)$ a neterminální symbol A
Výstup: FOLLOW(A).
Metoda:

1. Vytvoříme množinu $Ne = \{ B : B \Rightarrow^* e, B \in N\}$, tj. neterminálních symbolů, ze kterých je možno generovat prázdné řetězce.

2. Vytvoříme množinu F takto:
   a) Vytvoříme fiktivní pravidlo $A \rightarrow A$ a $F := \{ A \rightarrow A.\}$.
   b) Jestliže v množině F je položka, ve které je tečka na konci pravidla, tj. položka $B \rightarrow \gamma.$, vložíme do F nové položky vytvořené tak, že vezmeme všechna pravidla z P, ve kterých se na pravých stranách vyskytuje symbol B a tečku v nich umístíme právě za tento symbol B:
   $F := F \cup \{ C \rightarrow \alpha B. \beta : B \rightarrow \gamma. \in F, C \rightarrow \alpha B \beta \in P \}$.
   c) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol, který patří do množiny Ne, přidáme do F další položku, kterou vytvoříme z uvažované položky posunutím tečky o jeden symbol doprava:
   $F := F \cup \{ A \rightarrow \alpha B. \beta : A \rightarrow \alpha .B \beta \in F, B \in Ne \}$.
   d) Kroky b) a c) opakujeme tak dlouho, dokud je možno do F přidávat další prvky.
   e) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol B, přidáme do množiny F všechna pravidla z P se symbolem B na levé straně a tečku umístíme před první symbol pravé strany:
   $F := F \cup \{ B \rightarrow . \alpha : C \rightarrow \gamma. B \beta \in F, B \in N \; B \rightarrow \alpha \in P \}$.
   f) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol, který patří do množiny Ne, přidáme do F další položku, kterou vytvoříme z uvažované položky posunutím tečky o jeden symbol doprava:
   $F := F \cup \{ A \rightarrow \alpha B. \beta : A \rightarrow \alpha. B \beta \in F, B \in Ne \}$.
   g) Kroky e) a f) opakujeme tak dlouho, dokud je možno do F přidávat další prvky.

3. Množinu FOLLOW(A) vytvoříme tak, že do ní vložíme všechny terminální symboly, které se vyskytují bezprostředně za tečkou v některém prvku množiny F. Jestliže je v množině F prvek, ve kterém se vyskytuje tečka na konci pravidla a na levé straně je symbol S (tj. počáteční symbol gramatiky), přidáme do FOLLOW(A) prázdný řetězec:
   FOLLOW(A) := $\{a : a \in T, B \rightarrow \alpha . a \beta \in F\} \cup \{e : S \rightarrow \alpha . \in F\}$.

3.3.13. Vnitřní jazyky překladačů – druhy, použití v jednotlivých fázích překladu, překlad jednoduchých jazykových konstrukcí

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní intermediální reprezentaci zdrojového programu (mezikód). Intermediální reprezentaci můžeme považovat za program pro nějaký abstraktní počítač. Tato reprezentace by měla mít dvě důležité vlastnosti: měla by být jednoduchá pro vytváření a jednoduchá pro překlad do tvaru cílového programu. Intermediální kód slouží obvykle jako podklad pro optimalizaci a generování cílového kódu. Může však být také konečným produktem překladu v interpretačním překladači, který vygenerovaný mezikód přímo provádí. Intermediální reprezentace mohou mít různé formy.

Postfixová notace

operátory následují ihned za operandy

- A B C * D + - => A – (B * C + D)
- efektivní zpracování pomocí zásobníku, musíme vědět prioritu operátorů

Prefixová notace

operátory a pak operandy

- + A B + C D => (A + B) * (C + D)

Tříadresový kód

Abstraktní forma mezikódu sestávající ze sekvence příkazů ve tvaru **x := y op z**, kde x, y a z jsou jména, konstanty nebo dočasné proměnné, op je nějaký operátor. Na levé straně je **adresa**, na pravé **instrukce.** Adresou může být název (ze zdrojového programu, je pak nahrazen pointerem do jeho tabulky symbolů), konstanta, kompilátorem generovaná dočasná proměnná (užitečné pro optimalizaci).

Jde o linearizovanou podobu syntaktického stromu.

Překlad výrazu x+y*z na tříadresový kód:

t1 := y * z

t2 := x + t1

Další formy tříadresových instrukcí: s unárním operátorem (x = -y), *copy* instrukce (x = y), indexované copy instrukce (x = y[0]), nepodmíněný skok (goto L), podmíněný skok (if x goto L), volání procedur (call p, n; předtím uvedeno *n* parameterů).

Trojice a čtveřice

Implementací tříadresového kódu jsou záznamy se třemi nebo čtyřmi poli: trojice resp. čtveřice.
Následující příklady budou ukázány na výrazu: a := b * (-c) + d [b]

Čtveřice

Záznam má čtyři položky nazývané **op, arg1, arg2 a res**. Tříadresový příkaz ve tvaru x := y op z je reprezentován umístěním op do op, y do arg1, z do arg2 a x do res. Některé tříadresové příkazy nepotřebují všechny položky (např. x := y).

Výhoda čtveřic oproti trojicím – v optimalizaci kompilátoru, kdy jsou instrukce často přemísťovány; přesun čtveřic je ok (nová pozice se dá hned určit podle dočasných proměnných), u trojic je při posunu třeba změnit reference na výsledky, protože jsou určeny svou pozicí.

Trojice

Jestliže se chceme vyhnout generování dočasných proměnných, je možné použít formu trojic. Trojice obsahuje op, arg1 a arg2. Místo dočasných proměnných jsou indexy do pole trojic (jejich pozice).

Příklady

Ukažme si tříadresový kód, čtveřice a trojice na příkladě výrazu:

a := b * (- c) + d [b]

Tříadresový kód

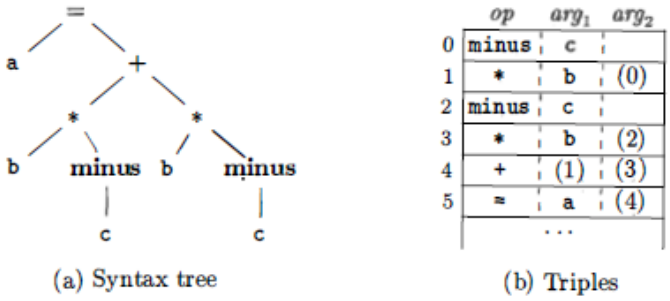
```
t1 := - c
t2 := b * t1
t3 := d [ b ]
t4 := t2 + t3
a := t4
```

Čtveřice

	op	arg1	arg2	res
(1)	uminus	c		t1
(2)	*	b	t1	t2
(3)	loadidx	d	b	t3
(4)	+	t2	t3	t4
(5)	:=	t4		a

Trojice

	op	arg1	arg2
(1)	uminus	c	
(2)	*	b	(1)
(3)	loadidx	d	b
(4)	+	(2)	(3)
(5)	:=	a	(4)



(a) Syntax tree (b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

3.3.14. Tabulka symbolů – obsah, způsob manipulace při vytváření a využívání při překladu

Jakmile syntaktický analyzátor najde určitou konstrukci symbolů, tedy frázi, je třeba této konstrukci přiřadit význam. Součástí syntaktického analyzátoru bývá procedura (nebo více procedur či funkci), která je postupně pro každou frázi volaná a jejím úkolem je doplnit údaje do tabulky symbolů nebo do interního kódu.

OBSAH

Do tabulky symbolů (tabulky objektů) **ukládáme postupně všechny objekty** - pojmenované **identifikátory** (které nejsou klíčovými slovy), **proměnné** nebo **konstanty**, **uživatelské datové typy**, **funkce**, **procedury**, návěští apod., na které v kódu narazíme. Pojem objekt zde budeme chápat obecněji než je obvyklé v teorii programování, bude to **prostě jakýkoliv identifikátor**, **který není klíčovým slovem** a lexikální analýza ho proto odlišila od jiných identifikátorů.

Zapisujeme zde obvykle název, typ, adresu, případně počáteční hodnotu objektu, počet a typ parametrů funkce a další informace potřebné při dalším překladu, ale také při provádění programu.

Tabulka symbolů může vypadat takto:

Název	Typ	Délka	Deklarováno	Adresa	Použito
`delky`	integer array 10	40 B	A	.....	N
`I`	byte	1 B	A	.....	A
`pocet`	integer	4 B	A	.....	N
`x1`	real	6 B	A	.....	N
`z1`	*nedefinováno*	0	N	0	A

V tabulce vidíme objekty délky (pole o délce 10 prvků, prvky jsou celá čísla), I, pocet a x1, které již byly deklarovány a objekt I také použit. Objekt z1 ještě nebyl deklarován, ale už je v kódu použit. V jazyce, který umožňuje pracovat pouze s deklarovanými proměnnými, se jedná o sémantickou chybu.

U každého typu objektu potřebujeme uchovávat různé druhy informací. Například u proměnné je to název, adresa, datový typ, velikost potřebné paměti apod., u funkce název, adresa, návratový typ, počet a typ jednotlivých parametrů, příp. zda jsou volány hodnotou nebo odkazem (jestliže jsou volány odkazem, musí sémantický analyzátor navíc ošetřit, aby ve volání funkce byly jako skutečné parametry použity pouze názvy proměnných a nikoli například výrazy nebo konstantní hodnoty), u dalších typů objektů to budou opět jiné údaje. Řádky tabulky mohou být navzájem závislé (jeden uživatelský datový typ může využívat deklaraci již dříve uvedeného, popř. proměnná je typu deklarovaného dříve, . . .), nesmí se však jednat o kruhovou závislost.

Tato tabulka nám slouží k mnoha účelům. **Využívá ji zejména sémantický analyzátor** (kontroluje, zda proměnná použitá v kódu je deklarovaná a zda její datový typ odpovídá jejímu použití, jestli u funkce souhlasí počet a typ argumentů, atd.), **používá se také u generování cílového kódu** (překladač musí vědět, kolik místa v paměti má vyhradit pro jednotlivé symboly).

Při interpretaci obvykle není nutné uchovávat informaci o adrese, samotná tabulka symbolů může sloužit jako úschovna symbolů, se kterou pak neustále pracujeme.

Způsob manipulace při vytváření a využívání při překladu

Tabulka symbolů může být **vytvářena již lexikálním analyzátorem**, ten však má **omezené možnosti** při zjišťování některých údajů, proto je v mnoha případech vhodnější přenechat tuto práci syntaktickému nebo sémantickému analyzátoru. Často používaný postup je vytváření tabulky lexikálním analyzátorem (kdykoliv narazí na identifikátor, který není klíčovým slovem, uloží ho do tabulky) s tím, že **další části překladače doplňují zbývající informace o vlastnostech uloženého identifikátoru**.

Otázkou je, **jak** vlastně **řadit** jednotlivé objekty v tabulce. Důležitým kritériem je rychlost vyhledávání, protože k tabulce symbolů přistupuje zejména sémantický analyzátor velmi často. U jednodušších jazyků je možné tabulku automaticky řadit **podle abecedy**, u složitějších jazyků řešíme **indexací**, kdy zároveň s tabulkou vytváříme indexový seznam (příp. soubor), ve kterém jsou odkazy na objekty seřazené podle abecedy.

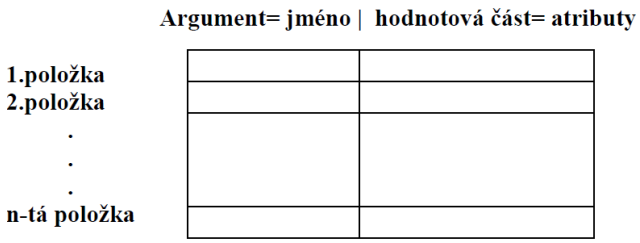
Speciální implementaci vyžaduje tabulka symbolů **pro jazyk s blokovou strukturou**, jako je třeba Pascal. Rozlišují se zde **lokální a globální objekty** a přístupnost lokálních je omezena. Každá proměnná je viditelná v tom bloku, ve kterém je deklarovaná, a také ve všech blocích vnořených. Když v určitém bloku použijeme proměnnou, hledáme informace o ní nejdřív v tom bloku, ve kterém se nacházíme. Při neúspěchu se posouváme do nadřízeného bloku a tak postupujeme, dokud ji nenajdeme. Pokud neuspějeme ani v hlavním bloku, znamená to, že byla použita proměnná, která není deklarovaná, jde o sémantickou chybu. **Každý blok má svoji vlastní tabulku**. S celou strukturou **se pracuje jako s** klasickým **zásobníkem**. Každá z tabulek má svou vlastní organizaci a je z ní přístupná nadřízená tabulka. „Aktivní tabulka je na vrcholu zásobníku, kde také začínáme prohledávat. Při vyhodnocení konce bloku se aktivní tabulka ze zásobníku odstraní. Po jejím odstraněni se sem přesune nadřízená tabulka. Tabulka hlavního bloku zůstává v zásobníku až do konce vyhodnocování programu, je odstraněna až jako poslední po vyhodnocení celého programu.

Tabulka symbolů

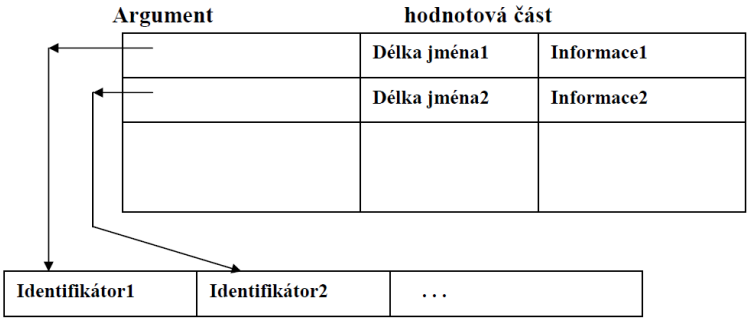
- uchovává informace o objektech
- umožňuje kontextové kontroly
- umožňuje operace
    a. inicializaci informace pro standardní jména
    b. vyhledání jména
    c. doplnění informace ke jménu
    d. přidání položky pro nové jméno
    e. vypuštění položky či skupiny položek

Struktura tabulky symbolů

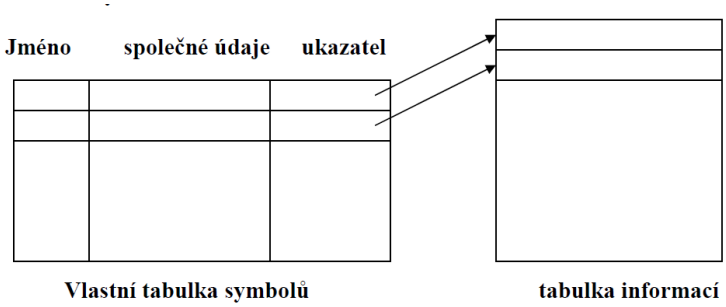
- s jednoduchou strukturou



- s oddělenou tabulkou identifikátorů

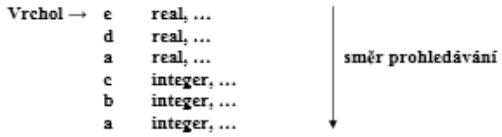


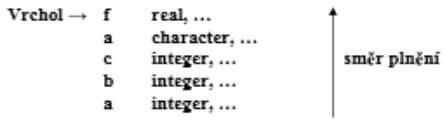
- s oddělenou tabulkou informací



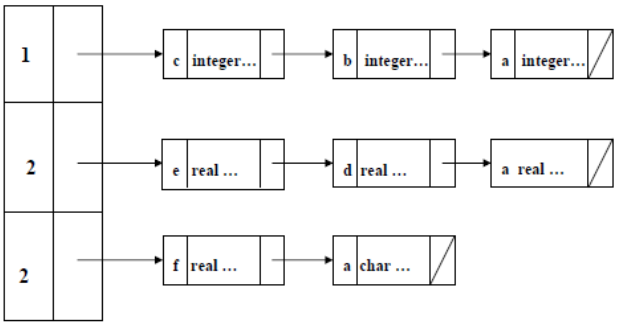
- uspořádané do podoby zásobníku

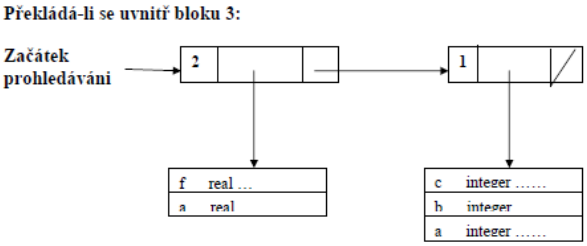
Tabulka symbolů uspořádaná do podoby zásobníku (pro jazyky s blokovou strukturou).
Rozsahová jednotka je blok, modul, funkce, balík,…
Respektuje zásady lokality





- s blokovou strukturou





Implementace tabulky symbolů

- **Vyhledávací netříděné tabulky** (jen pro krátké programy)
  - prostá struktura
  - lineární seznam
- **Vyhledávací setříděné tabulky**
  - průběžné setřiďování
  - setřídění po zaplnění
- **Frekvenčně uspořádané tabulky**
- **Binární vyhledávací stromy**
- **Tabulky s rozptýlenými položkami**

Ukládání polí a struktur

Pole i struktury mají pevnou adresu začátku pole a pro přístup k jednotlivým prvkům se výsledná adresa dopočítává. Pole mohou být v paměti uložena buď po řádcích nebo po sloupcích. Tomu musí odpovídat mapovací funkce, která vypočítává relativní adresu prvků. K této adrese musí být připočtena adresa začátku pole.

3.3.15. **Princip přidělování paměti překladačem**

Přeložený program dostane od operačního počítače k dispozici blok paměti, který obecně může být rozdělen na následující části:

- Vygenerovaný cílový kód
- Statická data
- Řídící zásobník
- Hromada

Základní způsoby přidělování:

- **Statické** (přidělení paměti v čase překladu)
- **Dynamické** (přiděleno v run time) **- v zásobníku** nebo **na haldě**

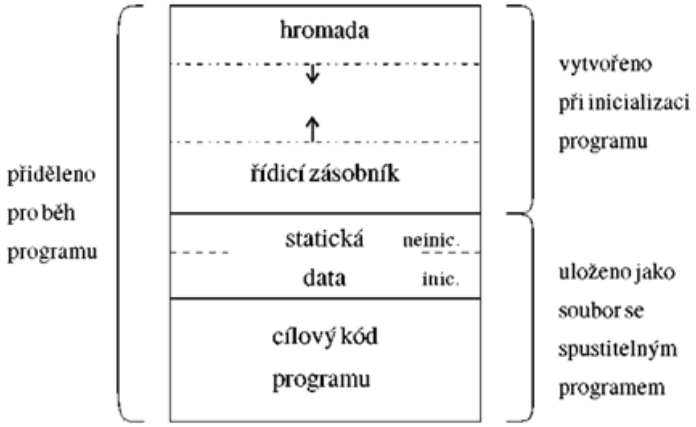
Velikost vygenerovaného kódu je známa již v době překladu, takže jej může překladač umístit **do staticky definované oblasti** (*Code*/*cílový kód programu*), obvykle na začátek přiděleného paměťového prostoru.

Rovněž velikost **statických datových objektů** může být známa již v době překladu a překladač je může **umístit za program** nebo uložit dokonce jako součást programu (to lze pouze u těch programovacích jazyků, které neumožňují rekurzivní volání procedur – Fortran).

Jazyky umožňující rekurzi (Pascal, C, …) využívají pro aktivace podprogramů řídicího **zásobníku (stack)**, do kterého se ukládají jednotlivé ***aktivační záznamy*** (AZ jsou generovány při voláních procedur).

Pro účely **dynamického přidělování paměti** (explicitně vyžadovaného voláním příslušných funkcí nebo implicitně při přidělování paměti například pro pole s dynamickými rozměry) se používá zvláštní část paměti zvaná **hromada (heap)**.

Vzhledem k tomu, že se velikosti použité části paměti pro zásobník a hromadu v průběhu činnosti programu mohou značně měnit, je výhodné pro obě části využít opačné konce společné části paměti – viz obrázek. **Zásobník roste směrem k nižším adresám, hromada směrem k vyšším**. Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti.



Pro zmíněné datové oblasti se používají následující hlavní metody přidělování paměti:

- **Statické** přidělování paměti v době překladu

- **Dynamické** přidělování paměti za běhu programu:
  - Přidělování paměti na *zásobníku*
  - Přidělování paměti z *hromady*

Statické přidělování paměti v době překladu

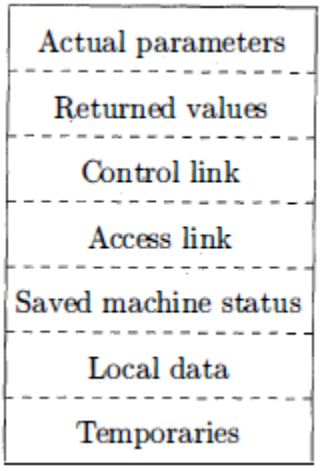
Při statickém přidělování paměti jsou všem objektům v programu **přiděleny adresy již v době překladu**. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné vždy na stejném místě, což umožňuje zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu. Statická alokace proměnných však klade na zdrojový jazyk určitá omezení. Údaje o velikosti a počtu všech datových objektů musejí být známy již v době překladu, **rekurzivní podprogramy mají velmi omezené možnosti**, neboť všechny aktivace podprogramu sdílejí tytéž proměnné, a konečně nelze vytvářet dynamické datové struktury.

Přidělování na zásobníku

Přidělování paměti pro aktivační záznamy na zásobníku se používá běžně u jazyků, které **umožňují rekurzivní volání podprogramů** nebo které **používají staticky do sebe zanořené podprogramy**. **Paměť pro lokální proměnné je přidělena při aktivaci podprogramu vždy na vrcholu zásobníku a při návratu je opět uvolněna**. To ale zároveň znamená, že hodnoty lokálních proměnných se **mezi dvěma aktivacemi podprogramu nezachovávají**.

Aktivace procedur při běhu programu jde znázornit ***aktivačním stromem***. Co uzel, to jedna aktivace, kořen je aktivací hlavní procedury, která je volána po spuštění programu; potomci uzlu *p* = volání procedur z procedury *p*. Kořen aktivačního stromu je na dně zásobníku, poslední aktivace má svůj záznam na vrcholu zásobníku.

Každá „živá“ aktivace má **aktivační záznam**. Obsah aktivačního záznamu se liší podle implementovaného jazyka.

1. 

dočasné hodnoty – vypadnou po vyhodnocení výrazů, jsou tu, pokud nemohou být udržovány v registrech

2. ***lokální data*** – patřící k dané proceduře s příslušným aktivačním záznamem
3. ***uložený strojový status*** – info o stavu stroje před voláním procedury. Typicky jde o:
   - *návratovou adresu* (= hodnota program counteru, kam se má pak procedura vrátit) a o
   - *obsah registrů* použitých procedurou (musí být obnoveny po návratu z procedury)

4. ***access link = statický ukazatel*** – pro lokaci dat, která procedura potřebuje, ale která se nachází v jiném aktivačním záznamu

5. ***control link*** – ukazuje na aktivační záznam volajícího (caller)

6. ***vrácené hodnoty*** – prostor pro návratovou hodnotu volané funkce (kvůli rychlosti lepší dávat do registru)

7. ***vlastní parametry*** – parametry použité volající procedurou; pokud je to možné, jsou umístěny radši v registrech kvůli výkonnosti.

Při implementaci přidělování paměti na zásobníku bývá **jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrcholu zásobníku**. Relativně k tomuto registru se pak počítají všechny adresy datových objektů, které jsou umístěny v aktivačním záznamu. Naplnění registru a přidělení nového aktivačního záznamu je součástí volací posloupnosti, obnovení stavu před voláním se provádí během návratové posloupnosti.

Volací (a návratové) posloupnosti se od sebe v různých implementacích liší. Jejich činnost bývá rozdělena mezi volající a volaný program. Obvykle volající program určí adresu začátku nového aktivačního záznamu (k tomu potřebuje znát velikost záznamu vlastního), přesune do něj předávané argumenty a spustí volaný podprogram zároveň s uložením návratové adresy do určitého registru nebo na známé místo v paměti. Volaný podprogram nejprve uschová do svého aktivačního záznamu stavovou informaci (obsahy registrů, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokračuje zpracováním svého těla. Při návratu opět volaný podprogram uloží hodnotu výsledku do registru nebo do paměti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si převezme návratovou hodnotu a tím je volání podprogramu ukončeno.

přístup k nelokálním proměnným při statickém =lexikálním rozsahu platnosti jmen. To řeší tzv. **řetězec statických ukazatelů** (*access links*). Pro zrychlení přístupu k nelokálním proměnným se zavádí vektor ukazatelů – **displej**. Zamezí se tak průchod aktivačními záznamy pro hluboko zanořené podprogramy.

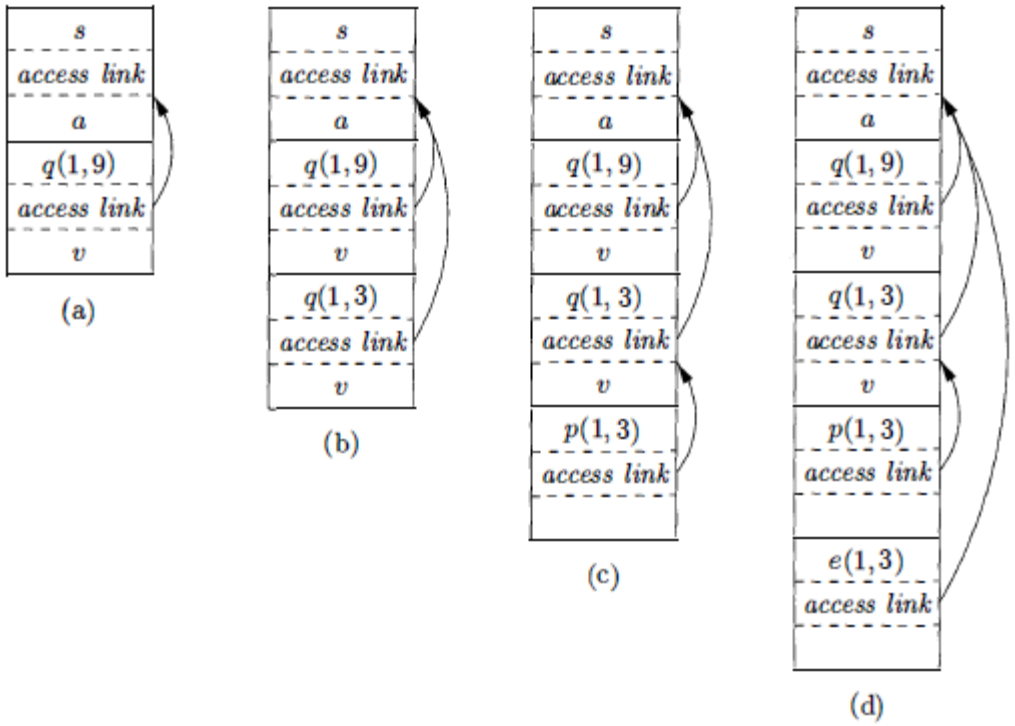


Figure 7.11: Access links for finding nonlocal data

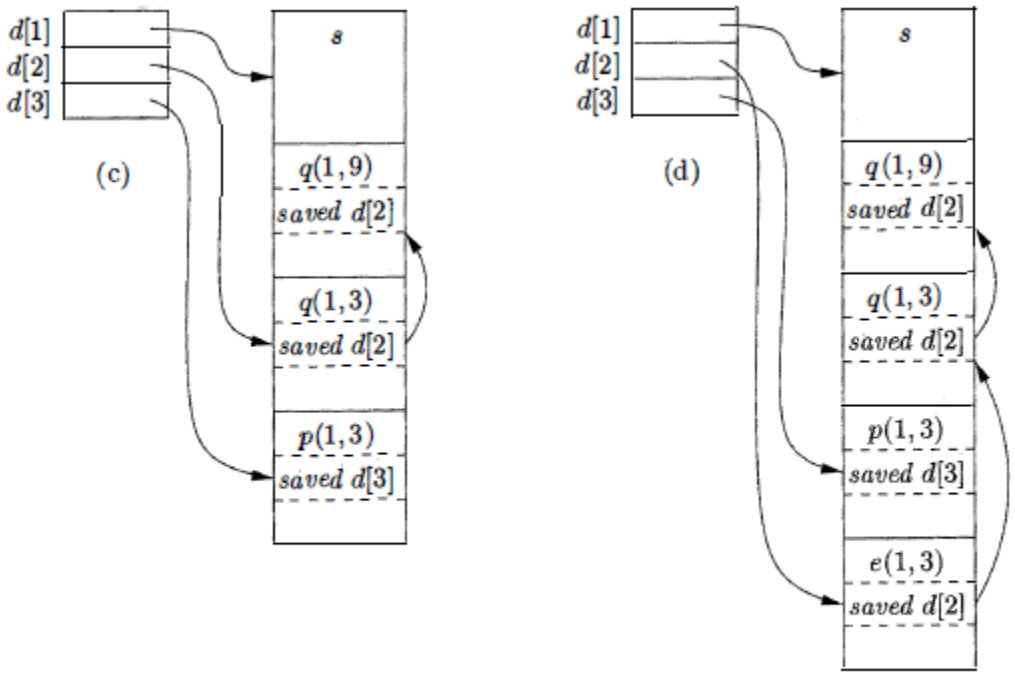


Figure 7.14: Maintaining the display

Přidělování z hromady

Strategie přidělování na zásobníku je nepoužitelná, pokud mohou hodnoty lokálních proměnných přetrvávat i po ukončení aktivace, případně pokud aktivace volaného podprogramu může přežít aktivaci volajícího. V těchto případech přidělování a uvolňování aktivačních záznamů se mohou překrývat, takže nemůžeme paměť organizovat jako zásobník. Aktivační záznamy se mohou v těchto nejobecnějších situacích přidělovat z volné oblasti paměti (hromady), která se jinak používá pro

dynamické datové struktury vytvářené uživatelem. Přidělené aktivační záznamy se uvolňují až tehdy, pokud se ukončí aktivace příslušného podprogramu nebo pokud už nejsou lokální data potřebná.

Při použití této strategie se pro vlastní přidělování a uvolňování paměti používají stejné techniky jako pro dynamické proměnné.

Správce paměti (memory manager) alokuje a dealokuje místo na heapu. V důsledku toho může dojít k fragmentaci heapu (vznik malých, nesouvislých míst = *děr*). Strategie *best fit* = alokuj nejmenší vhodnou a dostupnou díru.

Garbage collection hledá místo na heapu, které se už nepoužívá a může být proto realokované pro uchovávání dalších dat (Java, C#). Automaticky uvolňuje již nepoužívané objekty z paměti.

3.3.16. Vlastnosti jazykovýc hkonstrukcí pro statický a pro dynamický způsob přidělování paměti

http://service.felk.cvut.cz/courses/X36PJP/Skripta_prednasky.pdf !!!

Důležitá hlediska jazykových konstrukcí:

- Dynamické typy
- Dynamické proměnné
- Rekurze
- Konstrukce pro paralelní výpočty

Podstatný je rovněž způsob:

- Omezování existence entit v programu
- Předávání parametrů funkce (hodnotou, odkazem)
- Určování přístupu k nelokálním entitám
    - na základě statického vnořování rozsahových jednotek,
    - na základě dynamického vnoření rozsahových jednotek.

Statické přidělování paměti:

- Globální proměnné
- Statické proměnné
- Proměnné jazyka bez rekurze (i s blokovou strukturou) (možno staticky na zásobníku)

Dynamické přidělování paměti (na hromadě):

- Dynamické typy a proměnné (překladač neví v době překladu kolik jich bude potřebovat)
- Proměnné předávané odkazem
- Pointery v C pro dynamicky alokované proměnné

Předávání parametrů podprogramům

- hodnotou (C, C++, Java, C#) formální parametr podprogramu je lokální proměnnou (tohoto podprogramu) do níž se předá hodnota (proměnná je zkopírována do zásobníku podprogramu)
- odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# parametry označené ref) předá informaci o umístění skutečného parametru (předána adresa na proměnnou) - u Javy se všechno předává prostřednictvím hodnoty (pass-by-value) tj. v případě instancí tříd (objektů) dochází k předávání **adresy** objektu ???

- výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu

Dynamické přidělování v zásobníku

Aktivační záznam obsahuje místo pro:

- Lokální proměnné
- Parametry
- Návratovou adresu
- Funkční hodnotu (je-li podpr. funkcí)
- Pomocné proměnné (pro mezivýsledky)
- Další informace potřebné k uspořádání aktivačních záznamů

Statická typová kontrola – referenční prostředí podprogramů je definováno staticky, tj. při překladu zdrojového programu. Pro každou deklaraci je staticky vymezen rozsah platnosti, tj. část zdrojového kódu, ve kterém lze deklarované jméno použít. V podprogramu pak kromě lokálních jmen lze použít ta nelokání jména, do jejichž rozsahu platnosti je definice podprogramu vnořena. Statické referenční prostředí je často založeno na blokové struktuře programu.

Statické referenční prostředí je při překladu reprezentováno tabulkou symbolů. Překlad deklarace znamená rozšíření tabulky symbolů o nový záznam, při dosažení místa konce platnosti deklarace se záznam odstraní nebo skryje. Při překladu těla podprogramu překladač na základě tabulky symbolů pro každé jméno rozhodne, zda je či není v daném místě použitelné a jaký datový objekt (nebo jiný programový prvek) označuje. Tím se dosáhne vyšší bezpečnosti programu (nepoužitelné jméno je odhaleno již při překladu) i vyšší efektivity cílového programu.

Dynamická typová kontrola – např. Lisp, referenční prostředí podprogramů je definováno dynamicky. Dynamicky definované referenční prostředí **se nevytváří ani nekontroluje při překladu, ale až při provádění programu**. Při spuštění programu se vytvoří referenční prostředí tvořené vazbami jmen definovaných jazykem. Při každém vstupu do podprogramu se referenční prostředí rozšíří o vazby lokálních jmen podprogramu, při návratu z podprogramu se jeho lokální prostředí odstraní. Při provádění příkazů se pro každé jméno hledá jeho vazba.

Dyn. Definované ref. Prostředí snižuje bezpečnost programu i jeho efektivitu (význam jména se hledá při provádění programu). Jeho výhodou je jednoduchá sémantika – nenajde-li se jméno v lokálním prostředí podprogramu A, hledá se v lokálním prostředí podprogramu B, ze kterého byl podprogram A vyvolán, případně v lokálním prostředí podprogramu C, z čehož byl vyvolán podprogram B apod.