KIV/ZOS 2003/2004
Přednáška 2

Základní koncepce OS

OS poskytuje základní abstrakce -- procesy, soubory, uživatelské rozhraní.

Procesy

* proces = program, který běží
* má adresový prostor, ve kterém běží (MMU mu zajišťuje soukromí)
* v adresním prostoru má kód spustitelného programu, data, zásobník
* s procesem sdruženy registry (čítač instrukcí (PC), SP, univerzální registry CPU atd.) a další informace potřebné k běhu programu = stavové informace, nutné v okamžiku spuštění pozastaveného procesu

Např. v systémech se sdílením času periodické přepínání procesů, časem bude znovu spuštěn přesně ve stavu, ve kterém byl pozastaven.

* základní služby systému související s procesy:
 - vytvoření nového procesu (fork v UNIXu, CreateProcess ve Win32)
 - skončení procesu (exit v UNIXu, ExitProcess ve Win32)
 - čekání na dokončení potomka (wait v UNIXu, WaitForSingleObject ve Win32)
* další služby související s procesy:
 - alokace a uvolnění paměti procesu,
 - komunikace mezi procesy (interprocess communication, IPC)
* ve víceuživatelských systémech nutná identifikace
 - každému uživateli je předělen identifikátor uživatele (UID), každý proces běží s UID uživatele, který ho spustil
 - uživatelé mohou být členové skupin, např. pracovní týmy; skupiny GID (např. v systémech typu UNIX jsou UID a GID čísla)

* problém uvíznutí poměrně důležitý problém v OS - příklad:
 - představme si křižovatku, kde auta mohou jet pouze dopředu, protože za nimi stojí další
 - pokud by přijela na křižovatku ve stejné době a pokusila se projet, uvíznutí
 - totéž se může stát skupině procesů - pokud každý člen skupiny čeká na něco, co může (resp. nemůže, protože je také uvízlý) způsobit jiný člen skupiny
 - např. 3 procesy, každý čeká na zprávu, aby jí poslal dalšímu (uvidíme ještě spoustu lepších příkladů)

Soubory

* hlavní fcí zakrytí podrobností o discích a dalších I/O zařízeních, poskytnutí ”hezké” abstrakce soubor
* související systémová volání: vytvoření souboru, zrušení souboru, čtení ze souboru, zápis do souboru
* před čtením nebo zápisem musí být soubor otevřen, po ukončení práce zavřen (open, close)
* ”místo” kde je soubor -- adresář pro logické sdružování souborů, položkou adresáře je adresář nebo soubor => strom adresářů (nejčastěji)
* soubory a adresáře chráněny přístupovými právy souboru -- jejich kontrola při otevření souboru; není-li přístup => chyba

* připojitelnost souborových systémů:
 - v DOSu & Windows je disk určený prefixem (D:), co když potřebuji dát fyzicky jinam? = přesně typ závislosti na zařízení, který chceme v OS eliminovat
 - v UNIXu připojení kamkoli do adresářového stromu.

Uživatelské rozhraní

* uživatelské rozhraní původně řádkové (Command Line Interface, CLI)
* pro uživatele-laiky vyvinuta grafická uživatelská rozhraní (GUI)

a) b)

* původně uživatelské rozhraní součástí jádra (a)
* v moderních systémech je jedním z programů (b), výhodou je možnost výměny

* UNIX a Linux:
  - CLI - příkazový interpret je program, který se spustí po úspěšném přihlášení
  - GUI - systém X Window (zobrazování grafiky) + grafické prostředí (správce oken atd.) jsou programy v uživatelské režimu
* Windows NT
  - GUI - rozděleno na grafickou část (v jádře) a logickou část (v uživatelském režimu)

Procesy a vlákna

* nejdůležitější koncepce v OS: proces je abstrakce běžícího programu
* vše ostatní v OS s procesy souvisí => je zapotřebí se s tím seznámit co nejdříve

* dále předpokládáme systémy kde může běžet ”několik procesů souběžně”
* dvě možnosti:
  - skutečný paralelismus - pro každý proces je k dispozici CPU
  - pseudoparalelismus - CPU rychle přepíná mezi procesy a vytváří tím iluzi, že procesy jsou vykonávány paralelně (např. v systémech se sdílením času)
* pro snazší zacházení se zavádí koncepční model sekvenčního procesu

Proces jako abstrakce

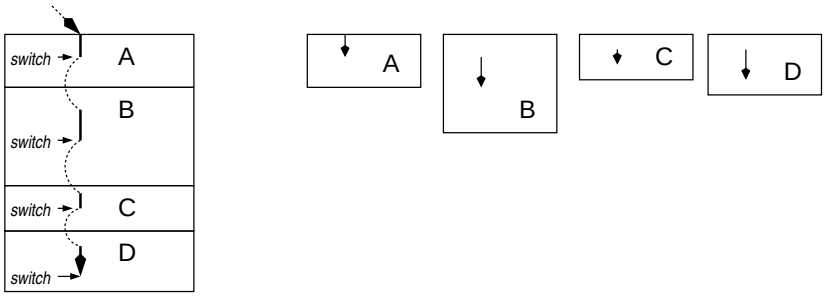
* všechen SW běžící na počítači (někdy i včetně OS) organizován jako {množina sekvenčních procesů}, krátce jen {procesů}
* proces = běžící program včetně obsahu čítače instrukcí, registrů, proměnných; běží ve vlastní paměti
* koncepčně má každý proces vlastní {virtuální CPU}

Ve skutečnosti reálný procesor přepíná mezi procesy (multiprogramování), ale pro porozumění mnoha problémům je snazší představa množiny procesů běžících (pseudo)paralelně.

Příklad:

Čtyři procesy, každý z nich má vlastní ”bod běhu” (tj. vlastní čítač instrukcí), každý běží nezávisle na ostatních.

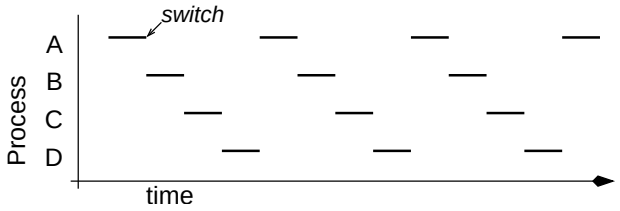
Na obr. (a) pseudoparalelní běh, (b) paralelní běh čtyř procesů, používáme jako koncepční model pro nezávislé sekvenční procesy.



a) One CPU b) Four CPUs

[]

Při pseudoparalelním běhu je v jednu chvíli aktivní pouze jeden proces. Po nějakém čase se OS rozhodne pozastavit běh procesu a spustit běh dalšího (např. proto, že proces běžel ”dlouho”). Po dostatečně dlouhém čase všechny procesy vykonají část své činnosti.



Přirovnání - kuchař peče koláče, má recept (kód programu) a potřebné ingredience (vstupní data). Přijde šéf a chce udělat minutku; kuchař si založí kde skončil s koláčem (uloží stav aktuálního procesu) a začne připravovat minutku podle příslušného receptu.

! Pozor, rychlost běhu procesu není konstantní! Většinou není ani reprodukovatelná.

=> Procesy {nesmějí} mít vestavěné předpoklady o časování.

Např. čtení pásky v jednoprogramovém systému: spuštění, čekací smyčka (rozjezd), čtení prvního záznamu. V multiprogramovém systému - pokud CPU zatím přepnut na jiný proces, záznam procesu může utéci - bude již za čtecí hlavou, až se zase přepne zpátky. Takovéto časově kritické záležitosti musí obvykle zařizovat jádro (nebo RT procesy).

Procesy také {neběží stejně rychle}; příchod události s vysokou prioritou (příchod šéfa).

Rozdíl mezi programem a procesem:

* program: v analogii recept, algoritmus vyjádřený ve vhodné podobě
* proces: aktivita, v analogii vaření; má program, vstup, výstup a stav (analogie: recept, ingredience, koláč, stav).

Stavy procesu
.....

Příklad - procesy často potřebují komunikovat s dalšími procesy:

```
$ ls -l | more
```

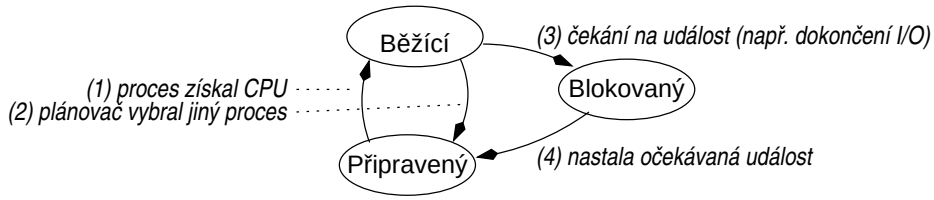
Příkaz ls vypíše adresář, příkaz more zobrazí jednu obrazovku a čeká na vstup uživatele. Na začátku bude ”more” připraven běžet, ale nemá žádný vstup => musí se {zablokovat} dokud vstup nedostane.

* blokování procesu - proces nemůže pokračovat, protože čeká na zdroj (vstup, zařízení, paměť) které zatím není dostupné

Proces může také být koncepčně připraven pokračovat, ale CPU vykonává jiný proces - odlišná situace: v prvním případě nemůže logicky pokračovat (např. nemůže zpracovat vstup který ještě neexistuje).

=> proces může být ve třech základních stavech:

1. běžící (running) - skutečně využívá CPU
2. připraven (ready, runnable) - dočasně pozastaven, aby jiný proces mohl běžet
3. blokován (blocked, waiting) - neschopný běhu, dokud nenastane externí událost.



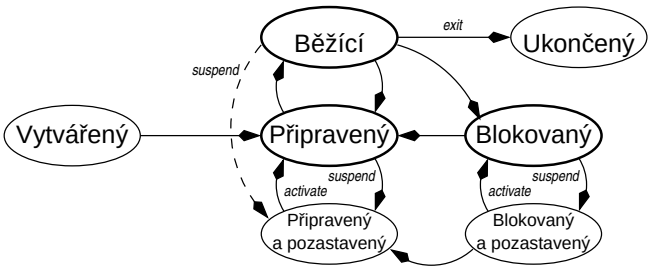
Přechody nastanou:
1. Plánovač vybere tento proces
2. Proces je pozastaven, plánovač (= část OS která vybírá procesy) vybere jiný proces
3. Proces se zablokuje, protože čeká na událost (na zdroj; zablokování provede systém např. pokud proces chce číst z klávesnice a není žádný vstup)
4. Nastala očekávaná událost, např. zdroj se stal dostupný.

=> můžeme mít představu o systému:

* jádro OS obsahuje plánovač, ten určuje, který proces bude běžet
* nad OS řada procesů, střídají se o CPU.

Poznámka (stav procesu ”pozastavený”)

Mnoho operačních systémů si vystačí s výše uvedenými třemi základními stavy. V některých systémech může být proces operačním systémem nebo jiným procesem pozastaven nebo aktivován. Ve stavovém diagramu pak přibudou dva nové stavy.



[]

Implementace procesu
.....

Zatím zjednodušeně - jak se zařídí, aby každý proces měl vlastní paměť apod. se dozvíme postupně.

OS udržuje tabulku nazývanou {tabulka procesů} - každý proces v ní má položku, nazývanou PCB (Process Control Block).

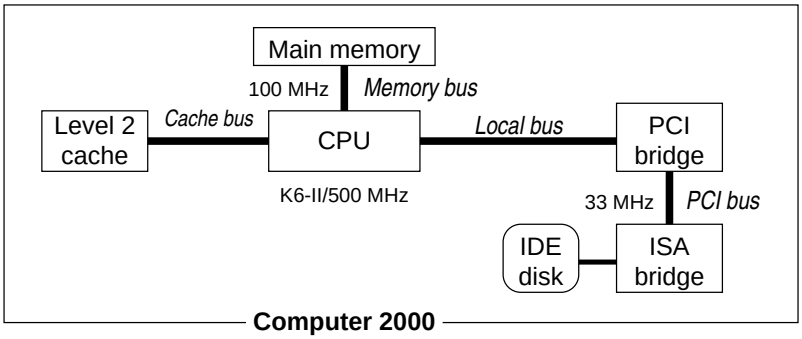
PCB obsahuje všechny informace, které musejí být uchovány, je-li proces přepnut ze stavu ”běžící” do ”připraven” nebo ”blokován” - tak aby bylo proces možné znovu spustit.

Konkrétní obsah PCB se liší mezi systémy, ale většina obsahuje pole týkající se správy procesů, správy paměti a správy souborů:

```
* správa procesů
 - identifikátory (bývají obvykle číselné)
  . identifikátor procesu (číslo jednoznačně určující proces)
  . identifikátor uživatele
 - stavová informace procesoru
  . obsah univerzálních registrů CPU (někdy je v PCB jen koncepčně, tj.
    fyzicky uloženo jinde)
  . ukazatel na další instrukci (obsah čítače instrukcí, PC)
  . stav CPU (Program Status Word, PSW - např. zda je zakázáno přerušení apod.)
  . ukazatel zásobníku
 - stav procesu (běžící, připraven nebo blokován)
 - plánovací parametry procesu (plánovací algoritmus, priorita apod.)
 - odkazy na rodiče a potomky
 - účtovací informace
  . čas spuštění procesu
  . čas CPU spotřebovaný procesem
 - nastavení meziprocesové komunikace (nastavení signálů, zprávy apod.)

* správa paměti
 - (popis paměti = ukazatel, velikost, přístupová práva)
  . úsek paměti s kódem (instrukcemi) programu
  . data (hromada - Pascal - new/release, C - malloc/free)
  . zásobník (návratové adresy, parametry fcí a procedur, lokální proměnné)

* správa souborů
 - prostředí (nastavení)
  . aktuální pracovní adresář, ...
 - otevřené soubory
  . způsob otevření - pro čtení / zápis
  . pozice v otevřeném souboru


Jak probíhá přepnutí procesu - viz také stará cvičení na webu:

* systém nastaví časovač, ten pak provádí pravidelně přerušení (přerušení ale
  přicházejí také od I/O zařízení, např. od disku po dokončení operace)
* na předem definovaném místě je adresa obslužného podprogramu
  přerušení
* po příchodu přerušení CPU:
 - uloží čítač instrukcí (PC) do zásobníku
 - načte do PC adresu obslužného podprogramu přerušení
 - přepne do režimu jádra
* vyvolaná obslužná procedura přerušení:
 - uloží obsah registrů do zásobníku
 - nastaví nový ”prozatímní” zásobník
* plánovač nastaví proces jako ready, vybere nový (= ”nejdůležitější”)
  proces
* ”přepnutí kontextu”
 - nastaví mapu paměti nového procesu
 - nastaví zásobník
 - načte obsah registrů
 - provede ”návrat z přerušení” (instrukce typu RET) = do PC adresu ze
   zásobníku, přepne do uživatelského režimu

Poznámka: Detaily výše uvedeného se mohou mezi konkrétními OS a CPU lišit,
zde uveden pouze základní princip.

Jak se procesy plánují si povíme téměř až na konci povídání o procesech.


Poznámka (problémy s pamětí)

 * CPU:
  - rychlost udávána v počtu instrukcí za sekundu
  - obvykle nejrychlejší komponenta v systému (hodinová frekvence)
  - skutečný počet vykonaných instrukcí závisí na rychlosti, jakou lze
    instrukce+data přenášet z a do hlavní paměti.
```

* hlavní paměť:
 - rychlost udávána v paměťových cyklech (= operace čtení nebo zápisu)
 - je cca o řád pomalejší než CPU
 - snaha řešit rozdíl rychlostí
  . CPU obsahuje rychlé registry = zápisníková paměť, obvykle 32x32 nebo 64x64 bitů; žádné zpoždění při přístupu
  . cache - malá paměť s vysokou rychlostí, obsahující poslední použitá data nebo instrukce; pokud je požadované slovo v cache (”cache hit”), dostaneme za 2 tiky hodin

Rychlosti sběrnic můžeme vidět na obrázku (”typický počítač z r. 2000”):



* vnější paměť - nejčastější disky
 - jsou mechanické a proto ještě o 3 řády pomalejší než RAM
 - zato jsou o 2 řády levnější/bit než RAM
 - kapacita obvykle o dva řády větší než RAM

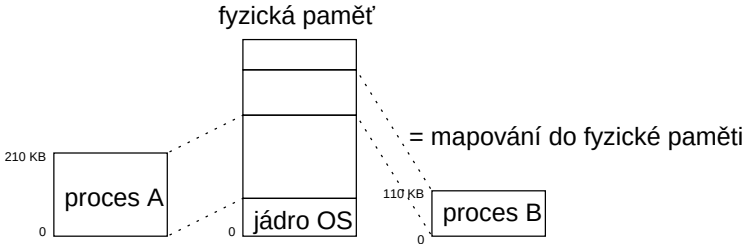
Můžeme považovat za hierarchii - nejmenší a nejdražší typ paměti je nejrychlejší, největší a největší nejlevnější (cena vzhledem k množství uchovávaných dat) je nejpomalejší.

Poznámka (úloha MMU)

Máme-li více programů v paměti, je zapotřebí zajistit

1) aby každý program měl paměť pro sebe, např. od adresy 0 (program je ve skutečnosti ”někde v paměti” - relokace)
2) aby si programy nemohly navzájem zasahovat do paměti nebo do jádra (ochrana).

Řešení - mezi CPU a pamětí je koncepčně umístěna jednotka správy paměti (Memory Management Unit, MMU). Program pracuje s tzv. virtuálními adresami, MMU je převede na fyzické adresy.



[Uživatelské programy a data jsou mapovány do fyzické paměti.]

Má důsledky pro výkonnost:

* pokud program nějakou dobu běží, budou v cache jeho data a instrukce, dobrá výkonnost - po přepnutí na jiný proces převážně přístup do hlavní paměti
* nastavení MMU se musí změnit.

Přepnutí mezi úlohami i přepnutí do jádra (např. při volání služby systému) je relativně drahé (ve smyslu ”stojí čas”).

[]

Základní služby pro práci s procesy
.....

* nejjednodušší systémy: všechny potřebné procesy jsou spuštěny při startu systému (např. zapouzdřených systémech)
 - běží celou dobu běhu systému => žádné služby nepotřebujeme

* OS UNIX a Linux:
 - služba fork() vytvoří přesnou kopii rodičovského procesu
 - návratová hodnota rozliší mezi rodičem a potomkem (potomkovi vrací fork() nulu)

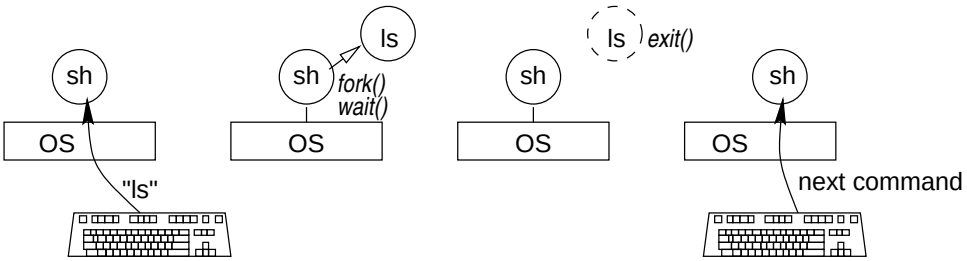
```
pid = fork();
if (pid == 0)
   jsem potomek
else
   jsem rodič
```

 - potomek může svou činnost ukončit pomocí exit(), rodič může na potomka čekat pomocí wait()

 - potomek může místo sebe spustit jiný program voláním execve(), které nahradí ”obsah paměti” volajícího procesu procesem spuštěným ze zadaného souboru
 - příklad použití:

```
if (fork() == 0)
      execve(”/bin/ls”, argv, envp);
else
      wait(NULL);
```

Např. příkazový interpret spouští příkaz => vytvoří nový proces, čeká na jeho dokončení. Proces se ukončí voláním služby systému.

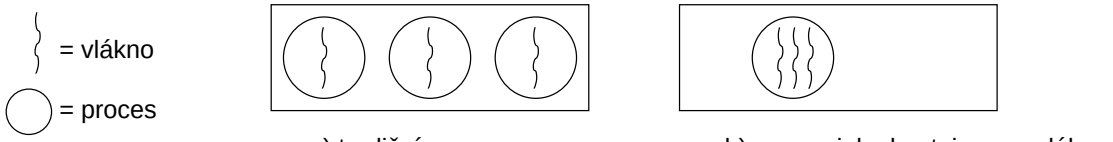


[Uživatel zadá ”ls”, shell spustí ”/bin/ls” a čeká na dokončení příkazu, ls končí, shell pokračuje.]

* Win32
 - vytvoření procesu službou CreateProcess
 - má mnoho parametrů specifikujících vlastnosti vytvářeného procesu
 - ve Win32 není koncept rodič/potomek - oba procesy jsou si rovné

Procesy a vlákna
---

* v tradičních OS každý proces svůj vlastní adresový prostor a místo kde běží (”bod běhu”) (a)
* v mnoha situacích výhodné, pokud více bodů běhu (jako by byly samostatné procesy), ale ve stejném adresovém prostoru (b)
* bod běhu nazýván vlákno (thread, někdy také lightweight process)
* je-li dovoleno více vláken ve stejném procesu - multithreading



a) tradiční procesy b) proces jako kontejner na vlákna

* vlákna v procesu sdílejí adresní prostor, otevřené soubory atd.
  (atributy procesu)
* vlákna mají soukromý čítač instrukcí, obsah registrů a soukromý zásobník
 - díky soukromému obsahu registrů a soukromému zásobníku mohou mít soukromé
   lokální proměnné

Původně byla vlákna využívána zejména pro VT výpočty na multiprocesorech (každé vlákno vlastní CPU, výpočet nad daty ve společné paměti).

Příklady dnes obvyklého využití:

* pokud úloha provádí rozsáhlejší výpočet a rozsáhlejší vstup/výstup, může
  být výhodné je provádět paralelně (urychlení úlohy)
* interaktivní procesy: jedno vlákno komunikuje s uživatelem, zatímco další
  vlákno provádí činnost v pozadí
 - WWW prohlížeč: jedno vlákno příjem dat, druhé vlákno zobrazování a
   interakce s uživatelem
 - textový procesor: jedno vlákno interakce s uživatelem, druhé vlákno
   přeformátování textu (např. po zrušení řádku v rozsáhlém souboru)
* servery (WWW server, databázový server...) jedno vlákno pro každého klienta

Multithreading podporují téměř všechny moderní OS (např. Linux, Windows NT), často i moderní jazyky (např. Java).

Proces začíná svůj běh obvykle s jedním vláknem, ostatní vytváří za běhu programově (konstrukce typu ”vytvoř vlákno”).

Poznámka:

V případě snahy paralelizovat výpočet apod. je třeba brát v úvahu režii na vytvoření/zrušení vlákna. V mnoha systémech je vytvoření/zrušení vlákna cca 100x rychlejší než vytvoření/zrušení procesu, ve srovnání s trváním nějaké operace provedené sériově však tento čas nemusí být zanedbatelný.

[]

Poznámka terminologická:

V mnoha případech pro nás není podstatné, zda máme jeden proces obsahující více vláken nebo zda máme více procesů, které část paměti sdílejí. Proto v následující části přednášek, týkající se synchronizace procesů a synchronizace vláken, nebudeme vždycky přesně rozlišovat ”procesy sdílející paměť” a ”vlákna”.

Ve starší literatuře týkající se OS se oba případy popisují jako ”procesy” - této terminologie se přidržím i já.

[]

Programové konstrukce pro vytváření vláken

OS a jazyky podporující multithreading musejí poskytovat způsob, jak potřebná vlákna vytvořit.

* statické: proces obsahuje deklaraci pevné množiny podprocesů, všechny
  jsou spuštěny při spuštění procesu (implementace např. tabulkou - v
  procesu tabulka podprocesů)
* dynamické: procesy mohou vytvářet potomky dynamicky (implementace -
  volání služby systému ”vytvoř proces” apod.; totéž i pro vlákna).

V následujícím textu uvedeme několik možností dynamického vytváření

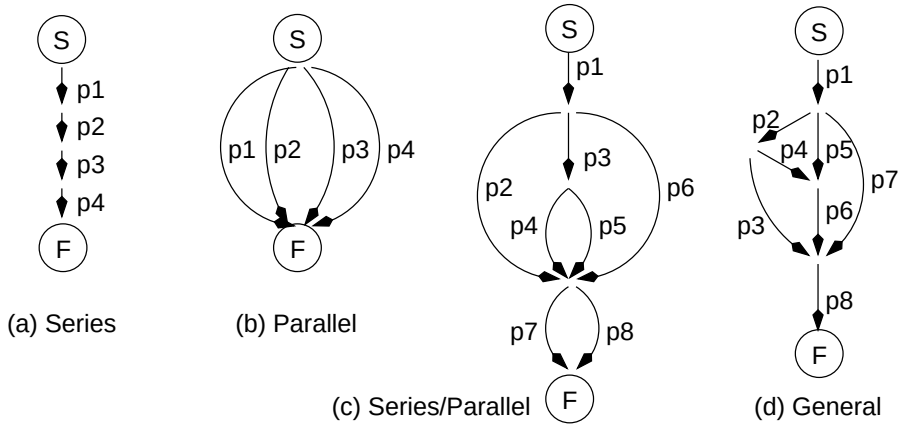
procesů; pro popis si zavedeme precedenční grafy.

Precedenční grafy

* grafy se hodí pro popis - konstrukce pro vytváření/rušení procesu
  musí být schopna vyjádřit různé {precedenční relace mezi procesy}

Za předpokladu, že systém má společný start a konec procesů možno
znázornit pomocí {precedenčního grafu procesů} (process flow graph):

* acyklický orientovaný graf
* běh procesu p_i je vyjádřen orientovanou hranou grafu
* vztahy mezi procesy (sériové nebo paralelní spojení)
  znázorněny spojením hran.



Abstraktní primitiva fork, join a quit

* Conway, asi 1963
* jeden z prvních mechanismů, možnost obecného popisu paralelních aktivit
* funkce primitiv:

```
     fork X;   provedení primitiva ”fork x” způsobí spuštění nového vlákna
               od příkazu označeného návěštím x; nové vlákno bude běžet
               paralelně s původním vláknem.

       quit;   provedení primitiva ”quit” ukončí vlákno.

  join t, Y;   atomicky (tj. nedělitelně) provede:

               t:=t-1; if t=0 then goto Y;

               Tj. pokud je hodnota t<>1, provede pouze t:=t-1; pokud je
               hodnota t=1, provede t:=0 a skočí na návěští Y.
```

Příklad použití:

Běh procesů odpovídající precedenčnímu grafu (a)

bychom mohli zapsat pomocí fork/join/quit takto:

```
     n:=3;                  // nejdříve musíme přiřadit hodnotu proměnné
     fork L2;               // spustíme vlákno od L2
     fork L3;               // spustíme vlákno od L3
     p1; join n, L4; quit;  // první vlákno pokračuje zde
 L2: p2; join n, L4; quit;
 L3: p3; join n, L4; quit;
 F:  ....                   // zde bude pokračovat jenom poslední,
                            // ostatní vlákna zaniknou
```

Zde vidíme typické použití ”join t, Y” - do t vložíme počet vláken, která se mají ”spojit” na Y a za join umístíme ”quit”.

Poznámka pro zajímavost:

Někteří autoři uvádějí pro tato primitiva poněkud odlišnou sémantiku: primitivum quit se neuvádí, primitivum join má význam join + quit. Tato modifikace je méně obecná a vychází z pozorování, že join a quit se často užívají společně.

[]

Zápis pomocí fork/join/quit je už na první pohled špatně čitelný, proto budeme hledat strukturovanější zápis.

Správně vnořené precedenční grafy

V případě grafů (a) a (b) z předchozího obrázku jsou všechny komponenty správně vnořené.

Definice (správně vnořené precedenční grafy)

Nechť:

* S(a, b) označuje sériové spojení procesů (za procesem a následuje proces b),
* P(a, b) označuje paralelní spojení procesů a a b.

Precedenční graf je {správně vnořený} pokud může být popsán kompozicí fcí S a P.

[]

Vlastnost správného vnoření je podobná ”správnému vnoření” v blokově strukturovaných jazycích (např. Pascal).

Příklad:

První tři grafy mohou výt zapsány jako:

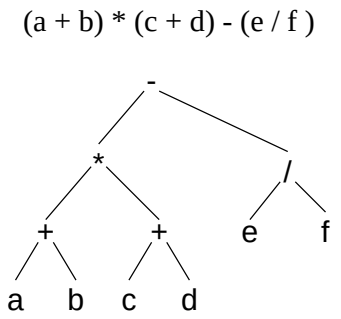
(a) S(p1, S(p2, S(p3, p4)))
(b) P(p1, P(p2, P(p3, p4)))
(c) S(p1, S(P(p2, P(S(p3, P(p4, p5)), p6)), P(p7, p8)))

Čtvrtý graf není správně vnořený, nemůžeme popsat jako kompozici fcí S a P: sériově spojené procesy (např. p2, p3) mají další proces (p4) který začíná v uzlu mezi nimi, ale nelze jej popsat jako S(pi, pj) nebo P(pi, pj).

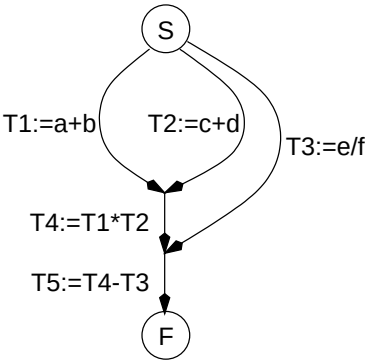
[]

Příklady paralelismu, při kterém vznikají správně vnořené procesy:

* vyhodnocení aritmetického výrazu:
  např. (a+b) * (c+d) - (e/f)

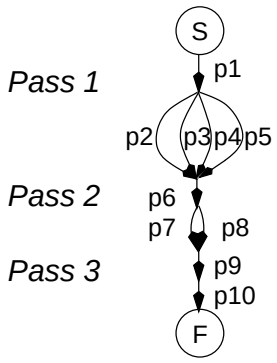


a) expression tree



b) process flow graph

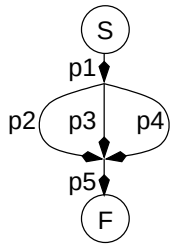
```
* řazení, např. dvoucestným slučováním:
  dvojice setříděných souborů velikosti 2^{i-1} jsou sloučeny do
  seznamu velikosti 2^i
```




```
* maticové násobení:
  při maticovém násobení A = B x C mohou být všechny prvky A počítány
  paralelně.


Abstraktní primitiva cobegin a coend
....................................

* původně parbegin, parend (Dijkstra 1968)
* explicitně specifikuje sekvenci programu, která má být spuštěna
  paralelně

Poznámka:

  Edsger Wybe Dijkstra (1930-2002) je autorem mnoha základních
  koncepcí, např. termínů "strukturované programování" nebo "zásobník".
  Za svůj život publikoval na 13 tis. článků...
  (http://www.cs.utexas.edu/users/EWD)

* "abstraktní" cobegin má formát:

   cobegin
      C1 || C2 || ... || Cn
   coend

 kde každé Ci je autonomní segment kódu (blok).

* výsledkem je vytvoření samostatného vlákna pro všechna Ci
* každé Ci běží nezávisle na ostatních vláknech v konstrukci cobegin / coend
* program pokračuje za coend až po skončení posledního Ci

Tj. kód

   begin
      C1;
```

```
      cobegin
         C2 || C3 || C4
      coend
      C5
   end.
```

odpovídá precedenčnímu grafu



Jak odpovídá cobegin/coend fcím P a S?

* každé Ci (segment kódu) může být dekomponováno na sekvenci příkazů p_i, můžeme psát:

  S(p_i1, S(p_i2, ...))

* konstrukce cobegin C1 || C2 || ... coend odpovídá následujícímu vnoření fcí P:

  P(C1, P(C2, ...)).

* tj. primitiva cobegin/coend jsou rozšířením fcí S a P se stejnou vyjadřovací silou => omezené na správně vnořené precedenční grafy

[]

Příklad použití:

```
  begin { (a+b) * (c+d) - (e/f) - viz precedenční graf uvedený výše }
       cobegin
              begin
                   cobegin
                          T1 := a+b || T2 := c+d
                   coend;
                   T4 := T1*T2
              end
              ||  T3 := e/f
       coend;
       T5 := T4 - T3
  end
```

Poznámka, resp. domácí úkol:

Je v předchozím příkladu maximálně využito paralelismu? Zkuste si převést aritmetický výraz a + b*c + d - e/f na paralelní program s cobegin/coend tak, aby bylo maximálně využito paralelismu.

Maximálním využitím paralelismu je zde míněno to, že část výpočtu spustím paralelně ”jakmile mohu”, tj. na nějakou část výpočtu čekám pouze pokud potřebuji její mezivýsledek.

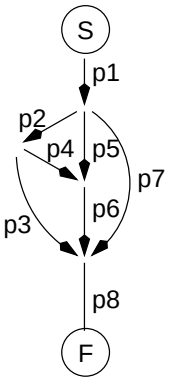
[]

Poznámka:

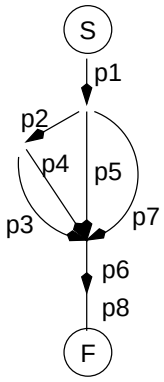
* pomocí cobegin/coend umíme vyjádřit správně vnořené precedenční grafy procesů, ale co pokud máme obecný precedenční graf?

* někdy mohou být obecné precedenční grafy převedeny na správně vnořené pomocí čekání:

- 1 nebo více procesů musí čekat (u výše uvedeného grafu např. zpozdíme p6 do skončení p3 a p7)
- nemusí to být vždy možné, např. pokud p6 musí běžet paralelně s p3 a p7 (např. kvůli komunikaci: pozdržet spuštění p6 nebude možné, pokud p3 a p7 musí běžet současně s p6, aby si s ním vyměnily zprávy)



(a) General



(b) "properly nested"

[]

Z předchozích příkladů vidíme, že strukturovaný zápis pomocí primitiv cobegin/coend je podstatně čitelnější, než zápis pomocí nestrukturovaného fork/join/quit. Například výpočet výrazu (a+b)*(c+d)-(e/f) by se pomocí fork/join/quit zapsal takto:

```
            n := 2;
            fork L3;
            m := 2;
            fork L2;
            t1 := a + b;   join m, L4; quit;
        L2: t2 := c + d;   join m, L4; quit;
        L4: t4 := t1 * t2; join n, L5; quit;
        L3: t3 := e/f;     join n, L5; quit;
        L5: t5 := t4-t3;
```

Na druhou stranu fork/join/quit postačuje pro popis libovolného precedenčního grafu a může být použito kdekoli v programu, např. uvnitř smyček. To lze použít např. pro popis iterací:

```
           for i:=1 to m do
              for j:=1 to n do
                  fork E;
           quit;
        E: A[i][j]:= ...
           join t, R;
           quit;
        R: ...
```

Poznámka:

Ve vláknech je často potřeba soukromých kopií některých proměnných rodičovského vlákna - např. ve výše uvedeném fragmentu kódu potřebuje každé vlákno vytvořené pomocí ”fork E” soukromou kopii proměnných i a j (jinak by se i a j měnilo v průběhu cyklu).

Pro označení proměnných, pro něž se má vytvořit ve chvíli vzniku nového vlákna kopie, se používají deklarace typu ”private”.

[]

Poznámka:

Popis obecného grafu na obr. (d) bude vypadat následovně:

```
            t6 := 2; t8 := 3;
            p1; fork L2; fork L5; fork L7; quit;
        L2: p2; fork L3; fork L4; quit;
        L5: p5; join t6, L6; quit;
        L7: p7; join t8, L8; quit;
        L3: p3; join t8, L8; quit;
        L4: p4; join t6, L6; quit;
        L6: p6; join t8, L8; quit;
        L8: p8; quit;
```

[]

Protože zápis pomocí cobegin/coend je čitelnější, budeme ho dalším textu používat pro zápis synchronizačních úloh apod.

Konstrukce pro vytváření vláken v současných systémech obvykle adaptují kombinace myšlenek cobegin/coend a fork/join/quit. V moderních jazycích se snaží být strukturované (podobně jako cobegin/coend), ale mít možnost vytvářet vlákna podle potřeby programově (jako fork/join/quit).

Explicitní deklarace podprocesu - Ada

Některé jazyky (např. Ada - US DoD, 1981) poskytují mechanismus pro označení úseku kódu jako samostatný podproces, jehož spuštění je možné řídit za běhu.

* statická deklarace podprocesu:
 - klíčové slovo process označuje segment kódu mezi begin a end jako samostatnou jednotku běhu
 - v deklaracích mohou být další definice podprocesů, tyto podprocesy jsou automaticky spuštěny při spuštění podprocesu p

```
process p
   deklarace ...
begin
   ...
end
```

* dynamická deklarace
 - místo ”process” je uvedeno ”process type”
 - definuje template, jehož instance mohou být vytvářeny dynamicky za běhu pomocí ”new”

Např.:

```
process type p2
   deklarace ...
begin
   ...
end

begin
   ...
   q = new p2;
   ...
end
```

Rozdíly od dříve popsaných primitiv cobegin/coend a fork/join/quit:

* new podobné fork a end podobné quit
* neumožňuje čekání na dokončení jiného procesu - chybí join nebo coend
* proto jsou pro vzájemnou interakci nutné další mechanismy.

Práce s vlákny v systému UNIX a jazyce C

* operační systémy typu UNIX poskytují knihovní fce pro vytvoření a ukončení vlákna (knihovna libpthread)

- jako vlákno se spustí určený podprogram, návratem z podprogramu vlákno zanikne
- protože názvy knihovnách fcí jsou dlouhé (všechny začínají pthread_*) a fce mají několik argumentů, uvedu pouze základní typy fcí a příklad
- knihovna poskytuje služby typu create f, join t, exit x, detach t, cancel t

. t = create f - podprogram f se spustí jako vlákno, vrací identifikátor vlákna
. exit x - odpovídá primitivu quit, navíc pomocí ní může končící vlákno předat hodnotu tomu, kdo vyvolá join
. x = join t - čeká na dokončení vlákna t, vrací hodnotu předanou končícím vláknem pomocí fce exit x
. detach(t) - oznámí systému, že na dokončení vlákna se nebude čekat pomocí join
. cancel(t) - zruší jiné vlákno uvnitř stejného procesu

- konkrétní názvy fcí jsou delší a argumentů je více např. vytvoření vlákna by vypadalo:

```
result = pthread_create(&id_vlákna, NULL, podprogram, argumenty);
```

- příklad programu:

```
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void *vlakno(void *m) /* podprogram pro vlákno */
{
        int i;

        for (i=0; i<10000; i++)
                write(1, m, 1);
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t th1, th2;

        pthread_create(&th1, NULL, vlakno, "*"); /* vytvoří vlákno */
        pthread_create(&th2, NULL, vlakno, ".");
        pthread_join(th1, NULL);        /* čeká na dokončení vlákna */
        pthread_join(th2, NULL);

        return 0;
}
```

Práce s vlákny v programovacím jazyce Java

* Java - vestavěná třída java.lang.Thread
 - programátor vytvoří podtřídu s vlastní metodou run() implementující činnost vlákna
 - vlákno se spustí vytvořením instance této podtřídy a spuštěním její metody start():

```
MyThread t = new MyThread();
t.start();
```

Poznámka (BACI a cvičení ze ZOS)

Abychom si vyzkoušeli taje vícevláknového programování, budeme používat emulátor BACI, kde je k dispozici cobegin a coend; jeho syntaxe i sémantika se od "abstraktního" cobegin a coend poněkud liší:

- v BACI se může konstrukce cobegin/coend objevit pouze v hlavním programu
- v BACI musejí být bloky Ci v cobegin / coend procedury

```
  - bloky v cobegin/coend nemohou být vnořeny
  - na rozdíl od "abstraktního" popisu se jako oddělovač používá
    znak středník, např.:

    begin { main }
       ...
       cobegin
          proc1(...); proc2(...); ... ; procN(...)
       coend;
       ...
    end.
```

[]

Časový souběh aneb problém kritické sekce

Pokud procesy sdílejí společnou paměť, kterou čtou a zapisují, může nastat časový souběh (race condition). Uvažme například hypotetický příklad, kdy dva procesy budou asynchronně zvětšovat společnou proměnnou x:

```
cobegin
  ...
  x := x + 1;
  ...
  ||
  ...
  x := x + 1;
  ...
coend
```

Předpokládejme, že vysokoúrovňový příkaz x := x+1 bude přeložen jako tři strojové instrukce:

1. načtení hodnoty x do registru procesoru (LD R, x)
2. zvýšení hodnoty x (INC R)
3. zápis nové hodnoty do paměti. (LD x, R)

Pokud oba procesy provedou příkaz sekvenčně, bude mít x správnou hodnotu x := x+2:

```
Proces 1:               Proces 2:

  LD R, x                  ...
  INC R                    ...
  LD x, R                  ...
  ...                      LD R, x
  ...                      INC R
  ...                      LD x, R
```

Pokud procesy běží pseudoparalelně a přepnutí kontextu nastane v nevhodném okamžiku, můžeme obdržet po vykonání obou sekvencí chybnou hodnotu x := x + 1:

```
Proces 1:
  LD R, x  // x = 0, R = 0
  (přepnutí na proces 2)
                          Proces 2:
                            LD R, x  // x = 0, R = 0
                            INC R    // x = 0, R = 1
                            LD x, R  // x = 1, R = 1
                            (přepnutí na proces 1)
Proces 1:  // x = 1, R = 0
  INC R    // x = 1, R = 1
  LD x, R  // x = 1, R = 1
```

Výsledek je zřejmě chybný, protože by se mělo provést každé zvětšení

hodnoty x (příklad - obsazování sedadel v rezervačním systému).

Tentýž problém nastane i pokud máme 2 CPU, např:

```
Proces 1:               Proces 2:

  LD R, x                  ...
  INC R                    LD R, x
  LD x, R                  INC R
  ...                      LD x, R
```

Poznámka, resp. domácí úkol (časový souběh)

Představte si dva procesy, provádějící přístup do databáze. Jeden proces chce provést operaci

účet:=účet+20 000

a druhý - téměř současně běžící - chce provést

účet:=účet-15 000

Jaký má být správný výsledek a jakého výsledku se můžeme dočkat, nastane-li časový souběh?

[]

❇