# ZOS

Verze 2010-srpen-31, v3

L. Pešička

- Základní znalosti vhodné k opakování
- Rozhodně prosím neberte jako jediný materiál na učení ke zkouškám / státnicím – obsahuje jen vybrané části

- Jde zde o zopakování, zda rozumíte dané problematice, tj. nejen fakta, ale i jak dané věci fungují

- Průběžně doplňováno, sledovat číslo verze na titulní straně

# Témata

1. Systémové volání
2. Přerušení
3. Program, proces, vlákno
4. Komunikace procesů / vláken
5. Životní cyklus procesů
6. Plánování procesů

# Téma I.

Systémové volání

Co to je?

Jak se dostaneme do režimu jádra?

Jak sdělím jádru bliží informace o tom, co chci?

# 2 základní režimy OS

- **Uživatelský režim**
  - V tomto režimu běží aplikace (word, kalkulačka,..)
  - Nemůžou vykonávat všechny instrukce, např. přímý přístup k zařízení (tj. zapiš datový blok *x* na disk *y*)
    - Proč? Jinak by škodlivá aplikace mohla např. smazat disk
    - Jak se tomu zabrání? Aplikace musí požádat jádro o službu, jádro ověří, zda aplikace má na podobnou činnost oprávnění
- **Režim jádra**
  - Zde jsou povoleny všechny instrukce
  - Běží v něm jádro, které mj. vykonává služby (systémová volání), o které je aplikace požádá

# Jak se dostat z uživatelského režimu do režimu jádra?

- Jde o přepnutí „mezi dvěma světy“, v každém z nich platí jiná pravidla

- Softwarové přerušení – instrukce INT 0x80
  - Stejně jako při hardwarovém přerušení (např. stisk klávesy) se začne vykonávat kód přerušení a vykoná se příslušné systémové volání
- Speciální instrukce (sysenter)
  - Speciální instrukce mikroprocesoru

# Systémové volání

- Pojem **systémové volání** znamená vyvolání služby operačního systému, kterou by naše uživatelská aplikace nemohla sama vykonat, např. již zmíněný přístup k souboru na disku.

- Aplikace může volat systémové volání přímo (*open, creat*), nebo prostřednictvím knihovní funkce (v C např. fopen), která následně požádá o systémové volání sama.

- Výhodou knihovní funkce je, že je na různých platformách stejná, ať už se vyvolání systémové služby děje různým způsobem na různých platformách.

# Systémové volání - příklad

1. Do nějakého registru uložím číslo služby, kterou chci vyvolat
   - Je to podobné klasickému číselníku
   - Např. služba 1- vytvoření procesu, 2- otevření souboru, 3- zápis do souboru, 4- čtení ze souboru, 5- výpis řetězce na obrazovku atd.
2. Do dalších registrů uložím další potřebné parametry
   - Např. kde je jméno souboru který chci otevřít
   - Nebo kde začíná řetězec, který chci vypsat
3. Provedu instrukci, která mě přepne do světa jádra
   - tedy INT 0x80 nebo sysenter
4. V režimu jádra se zpracovává požadovaná služba
   - Může se stát, že se aplikace zablokuje, např. čekání na klávesu
5. Uživatelský proces pokračuje dále

# Příklad

Jen ideový, v reálném systému se příslušné registry mohou jmenovat jinak


```
LD CX, 5        .. Budeme volat službu 5 (tisk řetězce)
LD BX, 100      .. Od adresy 100 bude uložený řetězec
LD AX, 10       .. Délka řetězce, co se bude tisknout
INT 0x80        .. Vyvolání sw přerušení, přechod do světa jádra
                .. Vykonání systémového volání
…               .. Kód naší aplikace pokračuje dále
```

# Další možnosti uložení parametrů

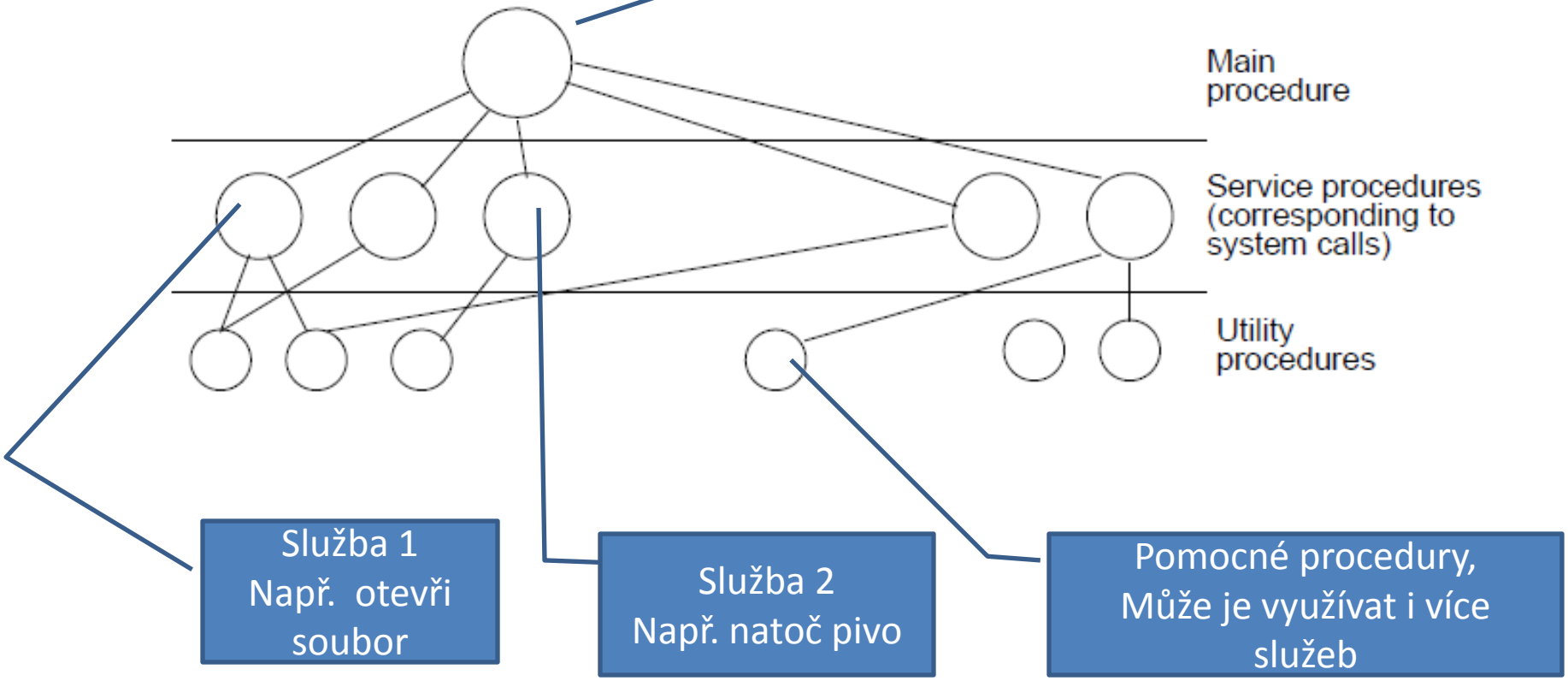
- Musím nějak jádru říci, kterou službu chci a další parametry

- Informaci můžeme uložit
  - Do registrů
  - Na zásobník
  - Na předem danou adresu v paměti
  - Kombinací uvedených principů

- Příklad informace:
  *chci po OS službu 2 (natočení piva) číslo služby uložím do CX, do registru AX uložím velikost piva (malé), do registru BX stupeň (desítka)…*
- *Jádro ví, že když je zavolána služba 2, tak jak má interpretovat obsah registrů AX a BX*

# Jak jádro implementuje jednotlivé služby?

Ukážeme si na monolitickém jádru:



# Téma II. – přerušení (interrupt)

- Přerušení patří k základním mechanismům používaným v OS

- **Asynchronní** obsluha události, procesor **přeruší** vykonávání sledu instrukcí (části kódu, které se právě věnuje), **vykoná** obsluhu přerušení (tj. instrukce v obslužné rutině přerušení) a **pokračuje** předchozí činností

- Analogie:
  - vařím oběd (vykonávám instrukce běžného procesu),
  - zazvoní telefon (přijde přerušení, je to asynchronní událost – kdykoliv)
  - Vyřídím telefon (obsluha přerušení)
  - Pokračuji ve vaření oběda (návrat k předchozí činnosti)
- Některé systémy mají víceúrovňová přerušení
  - (telefon přebije volání, že na někoho v sousedním pokoji spadla skříň)

# Druhy přerušení (!!)

- **Hardwarové přerušení (vnější)**
  - Přichází z I/O zařízení, např. stisknutí klávesy na klávesnici
  - **Asynchronní** událost – uživatel stiskne klávesu, kdy se mu zachce
  - Vyžádá si pozornost procesoru bez ohledu na právě zpracovávanou úlohu
  - Doručovány prostřednictvím **řadiče přerušení** (umí stanovit prioritu přerušením,aj.)
- **Vnitřní přerušení**
  - Vyvolá je sám procesor
  - Např. pokus o dělení nulou, výpadek stránky paměti (!!)
- **Softwarové přerušení**
  - Speciální strojová instrukce (např. zmiňovaný příklad INT 0x80)
  - Je **synchronní**, vyvolané záměrně programem (chce službu OS)
    volání služeb operačního systému z běžícího procesu (!!)
    uživatelská úloha nemůže sama skočit do prostoru jádra OS, ale má právě k tomu softwarové přerušení

- Velmi doporučuji přečíst:
  *http://cs.wikipedia.org/wiki/P%C5%99eru%C5%A1en%C3%AD*

# Kdy v OS použiji přerušení? (to samé z jiného úhlu pohledu)

- **Systémové volání** (volání služby OS)
  - Využiji softwarového přerušení a instrukce INT
- **Výpadek stránky paměti**
  - V logickém adresním prostoru procesu se odkazuji na stránku, která není namapovaná do paměti RAM (rámec), ale je odložená na disku
  - Dojde k přerušení – výpadek stránky
    - Běžící proces se pozastaví
    - Ošetří se přerušení – z disku se stránka natáhne do paměti (když je operační pamět plná, tak nějaký rámec vyhodíme dle nám známých algoritmů ☺)
    - Pokračuje původní proces přístupem nyní už do paměti RAM
- **Obsluha HW zařízení**
  - Zařízení si žádá pozornost
  - Klávesnice:  stisknuta klávesa
  - Zvukovka : potřebuju poslat další data k přehrání
  - Síťová karta: došel paket

# Vektor přerušení

- I/O zařízení signalizuje přerušení *(něco potřebuji)*
- Přerušení přijde na nějaké lince přerušení
  (IRQ, můžeme si představit jeden drát ke klávesnici, jiný drát k sériovému portu, další k časovači atd.)
- Víme číslo drátu (např. IRQ 1), ale potřebujeme vědět, na jaké adrese začíná obslužný program přerušení
- Kdo to ví? … vektor přerušení
- **Vektor přerušení** je vlastně index do pole, obsahující adresu obslužné rutiny, vykonané při daném typu přerušení

# Téma III. – program, proces, vlákno

- **Program**
  - Zápis algoritmu v nějakém jazyce (strojový kód, C, Java,..)
  - Zdrojový kód – obsahuje např. zdrojový kód v jazyce C
  - Binární kód – přeložený zdrojový kód spustitelný na dané platformě, např. soubor *calc.exe* , *notepad.exe* pro MS Windows
- **Proces**
  - Instance spuštěného programu
  - Dle jednoho programu můžeme spustit více procesů
  - Např. spustíme 2x calc.exe, na obrazovce 2 kalkulačky (dva procesy dle jednoho stejného programu), když se podíváme task managerem, budou mít různé PID (identifikátor procesu)
  - Každý proces má svůj vlastní PID (process id) – identifikátor v rámci operačního systému

# Program, proces, vlákno

- Program je statický soubor uložný např. na disku
- Proces je instance programu
  - Je dynamický, např. jeho pamětový prostor se může v čase měnit
  - Vyžaduje paměťový prostor, čas procesoru, …
- **Vlákno**
  - Každý proces má minimálně jedno vlákno
  - *Registr PC* (program counter) procesoru ukazuje na další instrukci, která bude vykonávána – bod běhu, při přeplánování na jiný proces (či vlákno) bude registr PC procesoru ukazovat zas na jinou instrukci (jiný bod běhu)
  - Pokud má náš proces více vláken, má více bodů běhu
  - Příklad textového editoru
    - Jedno vlákno čeká na vstup z klávesnice a zadaný znak zobrazí v editoru
    - Druhé vlákno na pozadí kontroluje pravopis napsaného textu
    - Třetí vlákno zobrazuje animaci (sponku co poulí očima)

# Multivláknový vs. multiprocesní

- Např. webový server můžeme udělat jako **multivláknový** (více vláken v jednom procesu) nebo **multiprocesní** (více samostatných procesů)
- Obě řešení jsou možná, každý přístup má své výhody a nevýhody, záleží na povaze aplikace, zde např. protože vlákna sdílejí stejný adresní prostor, tak chyba v jednom vlákně může položit i celý proces, multiprocesní řešení může být více odolné, ale zas náročnější na systémové zdroje

- **Vlákna**
  - sdílejí společný adresní prostor v rámci stejného procesu
- **Procesy**
  - každý má svůj vlastní adresní prostor (větší izolace)
  - vytvořit proces je obecně **náročnější** (na systémové zdroje) než vytvořit vlákno

# Vlákna v rámci jednoho procesu

- Něco sdílejí a něco mají jen pro sebe
- **Sdílejí**
  - Adresní prostor
  - Otevřené soubory
- **Soukromé**
  - Obsah registrů
    - Např. čítač instrukcí PC, tj. jaká další instrukce bude vykonávána
  - Zásobník (můžou mít tedy vlastní lokální proměnné)

# Procesy x vlákna

- **fork()**
  - Systémové volání, vytvoří nový proces
  - Náš proces je rodičem nově vytvořeného procesu
- **pthread_create()**
  - Knihovní funkce, vytvoří nové vlákno v rámci daného procesu
  - Knihovna pthreads  - POSIXová vlákna (přenositelnost mezi různými systém)

# Process Control Block (PCB) (!!)

- Operační systém si vede informace o tom, jaké procesy v daném systému existují a v jakém stavu se nacházejí – v datové struktuře PCB, každý proces svůj záznam

- OS má v PCB uloženy všechny potřebné informace, aby byl schopen zastavený proces znovu spustit (přidělit mu procesor a aby běžel z místa a stavu, ve kterém skončil) ( !!!!)

- Položky PCB – konkrétně se liší systém od systému
  - Pro správu procesů
  - Pro správu paměti
  - Pro správu souborů

# PCB – pro správu procesů

- **Identifikátory**
  - **PID** (process ID) – identifikátor procesu
  - **UID** (user ID) – identif. uživatele, pod kterým běží daný proces
    („proces má právo na to, na co jeho uživatel“)
  - **PPID** (parent PID) – id rodiče procesu (jen v některých systémech)
    (v Linuxu strom procesů s jedním společným prapředkem)
    (i ve Windows mají procesy svého rodiče!)
- **Stavové informace procesoru**
  - Obsah univerzálních registrů
  - Obsah registru PC (program counter – ukazatel na další instrukci)
  - Ukazatel zásobníku – registr SP
  - Stav CPU (PSW – různé příznaky, zda je povolené přerušení aj.)

# PCB – pro správu procesů 2

- **Stav procesu**
  - Běžící ☺, připravený (chci běžet), blokovaný (na něco čekám)
- Plánovací parametry procesu
  - Např. priorita a další příznaky, které využije plánovač procesů (různé příznaky pro různé plánovací algoritmy)
- Odkazy na rodiče, potomky, sourozence
  - Záleží opět na konkrétním systému, jaké všechny odkazy uchovává
- Účtovací informace
  - Nejen kvůli placení, ale pro statistiky
  - Čas spuštění procesu
  - Čas CPU spotřebovaný procesem (v jádře, uživatelském režimu)

# PCB – pro správu paměti

- Jaké části paměti má proces přidělený
  - I více úseků
  - Každý  úsek popsaný (ukazatel na poč. adresu, velikost, příst. práva)

  - Úsek paměti s kódem (code segment)
  - Úsek paměti pro data (hromada, alokace malloc/free v C)
  - Zásobník (stack segment)
    - Zásobník obsahuje lokální proměnné funkcí, parametry fcí a procedur, návratové adresy

# PCB – pro správu souborů

- Nastavení prostředí
  - Jaký mám aktuální pracovní adresář
    (tj. když vytvořím soubor ahoj.txt bez cesty, tak kam se uloží)

- Otevřené soubory
  - Jaký soubor
  - Způsob otevření (pro čtení – zápis)
  - Pozice v otevřeném souboru (na začátku, na 100 bajtu..)

# Komunikace procesů

- 2 základní způsoby, jak spolu mohou procesy komunikovat

- Sdílená paměť
  - Např. 2 **vlákna stejného** procesu, sdílejí paměť
  - Pro 2 různé **procesy** lze namapovat úsek společné paměti(!)

- Zasílání zpráv
  - Např. 2 procesy běžící na různých uzlech distribuovaného systému
  - Ale můžou být i na stejném uzlu, pokud nesdílejí paměť

# Problém se sdílenou pamětí

- Problémem je souběžný přístup, kdy se více procesů může snažit manipulovat se stejným místem v paměti, proměnnou, což může vést k různým výsledkům

- Tedy např. přístup ke **sdílené** proměnné x

- Úsek, kde přistupujeme ke sdílené proměnné, ke které by mohlo přistupovat více procesů/vláken najednou, se nazývá KRITICKÁ SEKCE

- Kritických sekcí může být více,
  např. KS1 – proměnná x, KS2 – proměnná y
  ochrana jedné KS1 nemá vliv na druhou KS2

# Ošetření kritické sekce

- V kritické sekci se může nalézat nanejvýš jeden proces (resp. vlákno)
- Jak toho docílit?
  - Ošetřit synchronizačním mechanismem
    - Mutex
    - Semafor
    - Monitor
- Jak to funguje?
  - Např. u semaforu před vstupem do kritické sekce voláme P()
  - Pokud je někdo jiný v kritické sekci („na semaforu svítí červená“), tak se proces ve volání P() zablokuje – přejde ze stavu běžící do stavu blokovaný, dokud se semafor neuvolní a nevrátí se z operace P()

# Semafory, monitory, mutexy co je třeba o každém znát?

- **O každém mechanismu znát základní pojmy – definice (co to je), použití (k čemu to je dobré), implementace (jak bychom daný mechanismus implementovali prostřednictvím toho, co nám systém dává k dispozici)**
- **Definice**
  - Např.: semafor je synchronizační primitivum tvořené celočíselnou hodnotou semaforu s a operacemi P() a V(), které fungují ….
- **Použití**
  - K ošetření kritické sekce:
    *s = 1; … P(s); KS; V(s)*
  - K synchronizaci:
    hlídání bufferu omezené velikosti např. v případě čtenáře a písaře
- **Implementace**
  - Např. jak bychom mohli semafor v operačním systému realizovat

# Producent - konzument

- Buffer omezené velikosti N
- Je potřeba hlídat:
  - Když je prázdný
    - Hlídá ho **semafor e** (empty) ; počáteční hodnota **N** (vše volno)
    - Konzument nemůže konzumovat položku z bufferu
  - Když je plný
    - Hlídá jej **semafor f** (full) ; počáteční hodnota **0** (nic zaplněno)
    - Producent nemůže ukládat položku do bufferu
  - Vlastní operace s bufferem
    - Hlídá ho **binární semafor m** ; počáteční hodnota **1** (volná krit. sekce)
    - Nemusí být jen pole, takže obecně vlastní vložení a výběr z bufferu ošetřit jako přístup do kritické sekce
    - Zde stačí binární semafor – kritická sekce je 0 obsazeno, 1 volno ☺

# Producent - konzument

```
Semaphore  e = N;
Semaphore  f=  0;
Semaphore  m = 1;

// kod producenta
While (1) {
    .. Namerim treba teplotu vzduchu
   p(e);    // je vůbec nějaká prázdná položka, abych měl kam ukládat?
     p(m);  vlozim_zaznam;   v(m);
  v(f);      // zvýším počet plných položek
}
```

# Producent - konzument

```
// kod konzumenta

While (1) {
p(f);       // je tam vůbec nějaký záznam v bufferu? Snížím počet plných
    p(m);   vyberu_zaznam;   v(m);
 v(e);      // zvýším počet prázdných položek
}
```

# Opakování

- Jakých hodnot může nabývat semafor s?
  - Celočíselných, 0, 1, 2, 3, ..  ; pokud je binární, tak jen 0 a 1
  - Hodnota 0 znamená, že semafor je blokovaný
  - Nenulová hodnota značí, kolikrát se může zavolat operace P() než se semafor zablokuje
- Jak vypadá typické ošetření kritické sekce?
  - Semaphore s = 1;  // na začátku musí být semafor volný
  - P(s);  // chrání vstup do KS
  - KS…..
  - V(s);  // uvolním vstup do KS
- Můžu kritickou sekci ochránit monitorem?
  - Ano, v monitoru může být aktivní pouze jeden proces, tedy na vstupu monitor pustí pouze jeden proces, ostatní čekají a až tento proces opustí monitor, tak může do něj vstoupit další (pozor – funkcionalitu monitoru lze rozšířit operacemi wait a signal)

# Opakování

- K čemu slouží v monitoru podmínková proměnná, wait(), signal()?
  - Aby se mohl proces v monitoru zablokovat, v té chvíli je monitor volný a může do něj další proces
  - Volání wait(c1) zablokuje náš proces nad podmínkou c1
  - Volání signal(c1) signalizuje jednomu procesu spícímu nad c1, aby se vzbudil
  - Podmínková proměnná c1 – představuje frontu procesů spících nad podmínkou c1
    bud je prázdná – nikdo není nad danou podmínkou zablokovaný
    nebo je neprázdná – obsahuje např. id procesů, které nad danou podmínkou spí
  - Je třeba dát pozor, aby po zavolání signal(c1) nebyli v monitoru aktivní dva procesy
    řeší Hoare – když náš proces zavolá signal, sám jde spát do speciální fronty
    řeší Hansen – volání signal musí být poslední akce před opuštěním monitoru

# Zasílání zpráv

- Operace **send**, **receive**
- Operace může být blokující x neblokující
  - Blokující – z funkce se vrátím, až např.  přijmu zprávu
  - Neblokující – z funkce se vrátím okamžitě, např. hned jak OS převezme zprávu k odeslání
- Když zasíláme zprávu, tak spíše než přímo proces (identifikovaný svým id) adresujeme schránku (např. číslo portu)
  - Důvod? Procesy často vznikají, zanikají, mění se jejich pid
  - Adresa schránky zůstává stejná
  - Znát rozdíl mezi mailboxem a portem

# Zasílání zpráv

- Výhoda – mohou komunikovat i procesy na různých uzlech, není potřeba sdílená paměť

- Může se (převážně u distribuovaných systémů) stát, že se zpráva ztratí, zduplikuje, přijde ve špatném pořadí.. Na lokálním systému (1 uzel) toto většinou nenastává

- RPC
  vzdálené volání procedur
  např. procedura sečti dvě čísla se provede na jiném uzlu
  na lokálním uzlu se zavolá spojka klienta, ta se spojí se spojkou serveru a na serveru se provede výpočet

# Životní cyklus procesů

- Kolik procesů může být ve **stavu běžící**?
  - Tolik, kolik je procesorů, nebo jader procesorů
  - Pokud jeden jednojádrový procesor, tak pouze 1 proces
- Jak se proces dostane do stavu blokovaný?
  - Běžící proces zavolá požadavek na I/O operaci, např. prostřednictvím systémového volání (resp. Knihovní funkce), např. read()
  - Běžící proces zavolá volání semaforu P() a semafor je blokovaný
- Jak dlouho setrvá proces ve stavu běžící?
  - Dokud neskončí, dokud se nezablokuje na I/O operaci
  - Nebo u systému se sdílením času, dokud mu nevyprší časové kvantum (tj. preemptivní systémy) – je tam šipka ze stavu běžící do připravený

# Životní cyklus procesů

- Jak je organizována fronta připravených procesů?
  - Záleží na plánovací strategii
    - RoundRobin – FIFO
    - Podle priorit – i více front a bere se z neprázdné fronty s nejvyšší prioritou
    - Jiné strategie (losování, …)
- Jak se může projevit deadlock?
  - Například tak, že několik procesů je ve stavu blokovaný a nedostanou se z něj, ale ostatní procesy  můžou být plánovány
  - Např. semafor s5 má hodnotu 0, a procesy P1,P2 zavolají oba operaci P(s5) – oba zůstanou viset ve stavu blokovaný, pokud žádný jiný proces se semaforem s5 nebude manipulovat

# Plánovač

- **Krátkodobý**
  - Je vždy
  - Plánování procesů ev. vláken, jak je známe z PC…
- **Střednědobý**
  - Pokud by v systému bylo moc procesů najednou, některé z nich odloží na později
    procesy si překážejí, ubírají pamět, perou se o zdroje, stejně jako když mezi lidmi vznikne tlačenice, pokud pouštíme procesy do systému ne okamžitě, jak je vytvořen, ale po menších dávkách..
- **Dlouhodobý**
  - Plánování úloh v dávkových systémech
  - Krátká úloha může předběhnout dlouhou, aby se zmenšila průměrná doba obrátky
    (analogie: pacient na 5 min předběhne v čekárně 2h pacienta)

# Plánovač vs. dispatcher

- **Plánovač** rozhoduje, jak budou procesy organizovány ve frontě připravených procesů

- **Dispatcher** provede vlastní přepnutí na proces zvolený plánovačem (tj. CPU začne vykonávat instrukce tohoto vítězného procesu)

# Plánování procesů
# Round Robin

- Časové kvantum
  - Max. množství času, po které náš proces může jednorázově využívat procesor
  - Jakmile uběhne, je přeplánováno na jiný proces (pokud nějaký chce počítat)
  - Nemusí je celé využít – může dojít k
    - Dokončení procesu
    - Zablokování nad I/O operací (velmi časté !)
  - Pokud proces celé časové kvantum nevyužije, nečeká se zbytečně na dokončení tohoto kvanta, okamžitě se naplánuje jiný proces a pro něj běží jeho vlastní časové kvantum (!!!)
  - Analogie: parkovací lístek, zaplatíme si parkování na hodinu, ale když po 45minutách odjedeme, okamžitě může volné místo obsadit jiné auto se svým vlastním parkovacím lístkem, tj. parkovací místo nezůstane 15 minut prázdné !

# krátké vs. dlouhé časové kvantum

- Krátké časové kvantum
  - Rychlejší odezva systému (interaktivita)
  - Větší režie – pořád se přepíná mezi procesy
- Dlouhé časové kvantum
  - Menší režie
  - Dlouhá doba odezvy (vadí u interaktivních procesů, které komunikují s uživatelem)
    *představte si systém, kde běží 60 procesů a velikost kvanta by byla 1 sekunda – odezva systému jednou za minutu…*

# I/O based a CPU based proces

- I/O vázaný proces
  - Velké množství I/O operací, nevyužije tolik celé časové kvantum, většinou se často během něj zablokuje
  - Typicky interaktivní proces (textový editor)
- CPU vázaný proces
  - Často využije kvantum až do konce
  - Typicky matematický výpočet bez interakce s uživatelem
  - Stráví na procesoru daleko víc času
- **Jak zvýhodnit I/O vázané procesy?**
  - Jedním z modifikací bylo, že po dokončení I/O operace šel proces na začátek fronty, nikoliv na konec fronty připravených procesů

# Prioritní plánování

- Vždy dvě složky
  - Statická priorita
    - Určí se při startu procesu, je stálá
    - Vyjadřuje jak důležitý je proces
  - Dynamická priorita
    - Mění se průběžně (dynamicky)
    - Např. procesu, který dlouho čekal ve frontě je zvýšena dynamická složka priority
    - Zabraňuje vyhladovění procesu (aby nebyl proces s nízkou prioritou neustále předbíhán)
- Výsledná priorita = statická + dynamická

# Prioritní fronty

- Procesy mohou být řazeny do front dle priority (prioritní třídy)
- Obsluhuje se vždy nejvyšší neprázdná fronta
  (fronta s co největší prioritou, která není prázdná)
- Uvnitř jedné fronty se procesy klasicky střídají FIFO
- Počet front může být velký, např 30-50
- Jednou za čas se priority přepočítají (mění se dynamická složka)

# Plánovač spravedlivého sdílení

- Spravedlnost vůči uživateli

- Pokud jeden uživatel u1 si pustí 100 procesů
  a
  druhý uživatel u2 si pustí 1 proces

  měl by být poměr využití času pro uživatele
  1:1 (zohledňuje se uživatel)
  a ne
  100:1 (plánují se procesy bez ohledu na uživatele)

  (penalizuje se uživatel, kterému běží hodně procesů)

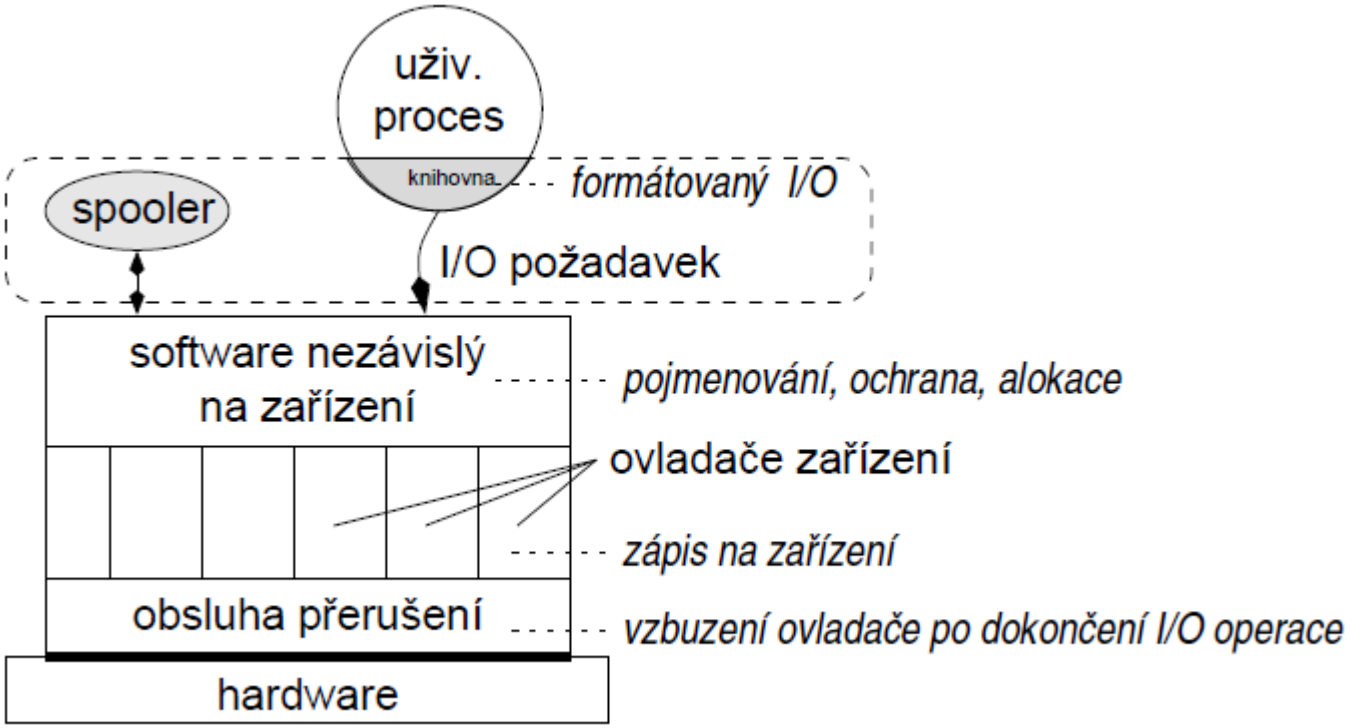
# Plánovač spravedlivého sdílení

- Vědět mechanismus jak funguje..
- Používá se zde rozklad g = g / 2  jednou za sekundu
  - Proč?
  - Aby odrážel chování systému v poslední době
    (nedávná minulost má větší váhu než dávná)
    tedy pokud by uživateli běželo 100 procesů před hodinou, má to menší vliv, než když před 10 sekundami

# Princip I/O softwaru

1. Uživatelský I/O SW
   - Např. vyvolání funkce printf(“”);
2. SW vrstva OS nezávislá na zařízení
   - Společné fce pro zařízení daného druhu
   - Pojmenování zařízení (/dev/tty1)
   - Přístupová práva
   - Vyrovnávací paměti (blok, znak, ..)
3. Ovladač zařízení (kód závislý na zařízení)
   - Často od výrobce HW, bez ovladače zařízení nepoužitelné
4. Obsluha přerušení (nejnižší úroveň v OS)
   - Ve chvíli dokončení I/O požadavku (např. data pro zvukovou kartu)

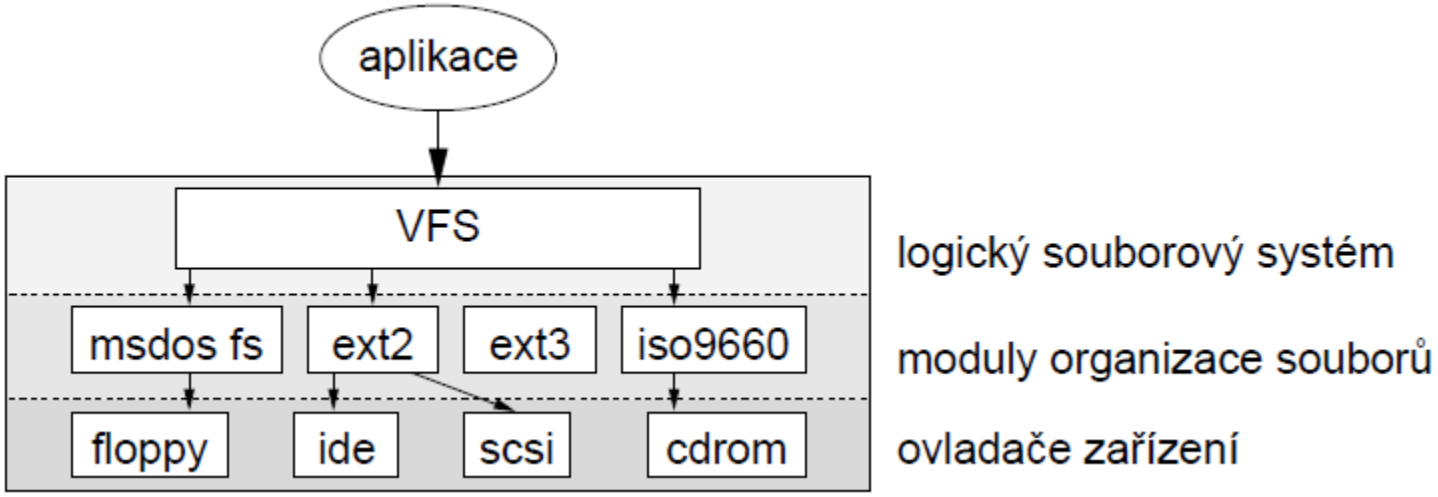
# Princip I/O SW



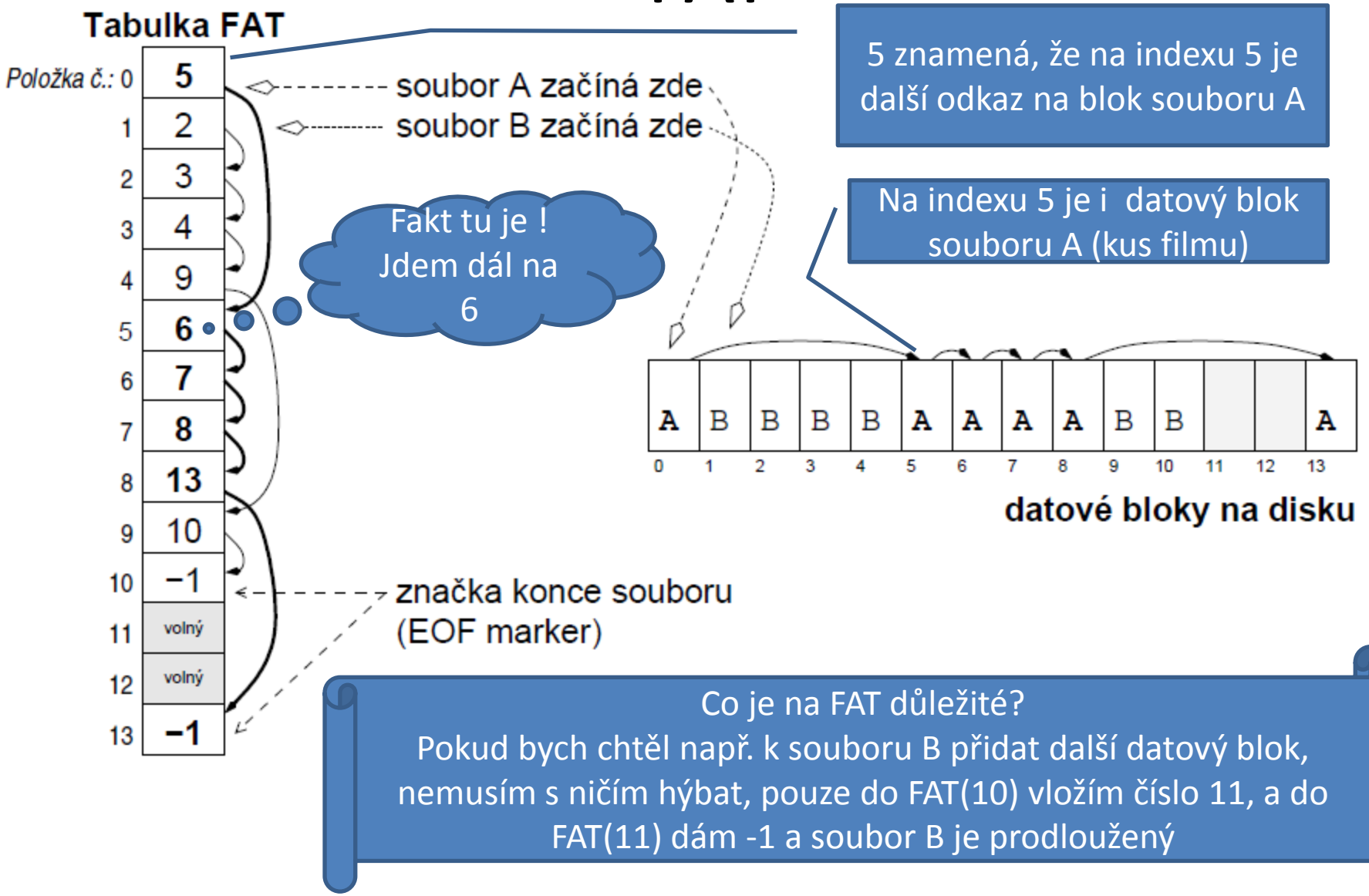
# Implementace souborových systémů

1. Virtuální souborový systém (VFS)
   - Kód společný pro všechny typy FS
   - Volán aplikacemi
2. Modul organizace souborů
   - Konkrétní souborový systém: FAT, NTFS, ext3 aj.
   - Řeší správu datových struktur daného filesystému (vzpomeňte FAT)
   - Řeší správu volného místa (volných bloků)
   - Čtení / zápis datových bloků konkrétního souboru
   - Převod čísla bloku souboru na diskovou adresu
3. Ovladače zařízení
   - Interpretuje požadavek přečti blok 5235 ze zařízení 3
   - Z pevného disku, CD, přes síť, …

# Implementace souborových systémů



# FAT




Co je na FAT důležité?
Pokud bych chtěl např. k souboru B přidat další datový blok, nemusím s ničím hýbat, pouze do FAT(10) vložím číslo 11, a do FAT(11) dám -1 a soubor B je prodloužený

# FAT

- FAT je ukázka implementace souborového systému, kde v jedné části máme datové bloky (obsahující např. části jednoho filmu)
  a v druhé části máme indexy, které nám říkají, pod jakým číslem se nalézá další odkaz
- Výhodou je, že s určitým souborem můžeme manipulovat, zrušit ho, prodloužit, atd., aniž bychom ovlivnili pozici ostatních souborů na disku

# Příklady filesystémů (!!!)

- **FAT**
  - Starší verze Windows, paměťové karty
  - Nepoužívá ACL – u souborů není žádná info o přístupových právech
- **NTFS**
  - Používá se ve Windows XP/Vista/7
  - Používají ACL: k souboru je přiřazen seznam uživatelů, skupin a jaká mají oprávnění k souboru (!!!!)
- **Ext2**
  - Použití v Linuxu, nemá žurnálování
  - Nepoužívá ACL – jen standardní nastavení (vlastní,skupina, others), což ale není ACL (to je komplexnější)
- **Ext3**
  - Použití v Linuxu, má žurnál (rychlejší obnova konzistence po výpadku)