Last updated: 8. máj 2019

# POS - poznamky + otazky

**Další zdroje:** [http://wiki.fituska.eu/index.php?title=Fulltextov%C3%A9_ot%C3%A1zky_POS](http://wiki.fituska.eu/index.php?title=Fulltextov%C3%A9_ot%C3%A1zky_POS)

**Starý sdílený dokument** (informace z něj byly vypsány do tohoto dokumentu - spíše pro zajímavost):
[POS priprava na semestralku]

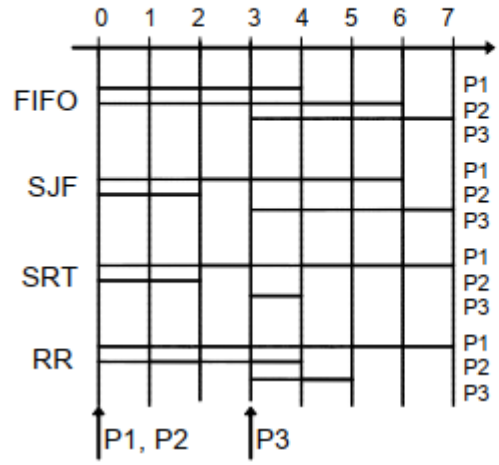
**Výtah:**

pos_vytah.pdf
208.6 KB

#ganttuv diagram

**Simulace**

**Ganttův diagram**




```
r1=4
r2=6 R=14 s=4,7
r3=4
r1=6
r2=2 R=12 s=4
r3=4
r1=7
r2=2 R=10 s=3,3
r3=1
r1=7
r2=4 R=13 s=4,3
r3=2
```

R - celková doba strávená v systému (všech procesů)
s - střední doba strávená v systému

**Operační systém (OS)**

• most mezi hardwarem a běžícím programem
• most mezi hardwarem a uživatelem
• vytváří prostředí pro běh programů (procesů, vláken)

**Architektura počítačů**

• jeden nebo více procesorů

**Počet procesoru a organizace paměti (synchronizace):**
• jednoprocesorové
• víceprocesorové:
- jedna sdílená paměť (UMA),
- rozprostřená sdílená paměť (NUMA),
- distribuované systémy (bez sdílené paměti)

**Typy operačních systémů, požadavky na technické vybavení**

**Typy operačních systémů**
**Plánování:**
• dávkové (batch),
• sdílení času (timesharing),
• systémy reálného času (real-time)
**Použití:**
• univerzální
• specializované (souborový server, databázový server, RT)
**Počet uživatelů:**
• jednouživatelské,
• víceuživatelské

**A) Monoprogramové - MS-DOS:**
• aktivní pouze jeden proces,
• jednodušší implementace - nenastává souběžnost provádění,
• využití zdrojů slabé, obvykle jen jeden uživatel.
**B) Multiprogramové (multitasking, multiprogramming):**
• aktivních více procesů současně,
• efektivnější využití prostředků,
• jednodušší implementace vyšších vrstev (GUI, síť. rozhraní),
• nutnost pro víceuživatelský systém.
**B-1) Jednoprocesorové (uniprocessor, UP)**
**B-2) Víceprocesorové, paralelní (multiprocessor, MP)**
**B-2-a) Symetrické multiprocesorové systémy (SMP)**
• kód jádra i kód procesů je prováděn na všech procesorech,
• procesory mají rovnocenný přístup k operační paměti, V/V zařízením a přerušovacímu systému.
**B-2-b) Nesymetrické multiprocesorové systémy**
Jak realizovat současný běh více procesů než je počet procesorů?
**Ochrana operačního systému:**
1. znemožnění modifikace kódu a datových struktur jádra OS,
2. zabránění provádění V/V operací,
3. zabránění přístupu do paměti mimo přidělený prostor,
4. odolnost vůči chybám (odebrání procesoru při zacyklení).

**Nutná podpora na úrovni hardware:**
1. Dva režimy činnosti procesoru,
2. privilegované instrukce povolené pouze v systémovém režimu (V/V, změna režimu, zpracování přerušení),
3. ochrana paměti – definuje přístupné úseky adresového prostoru pro běžící proces,
4. přerušovací systém, přerušení převede procesor do systémového režimu,

**systémech? Jaký to má vliv na synchronizaci?**

**1) symetrický multiprocesorový systém** - kód jádra i kód procesu je prováděn na všech procesorech. Procesory mají rovnocenný přístup k paměti i k V/V zařízením i přerušovacímu systému.
**2) nesymetrický multiprocesorový systém** - jádro OS běží na dedikovaném procesoru; zpravidla prvním (někdy označovaném jako bootovací), na ostatních procesorech běží uživatelské procesy v uživatelském módu. Toto řešení není příliš efektivní proto se dnes prakticky nepoužívá.

**Techniky strukturování jádra operačních systémů**

**1. Monolitické jádro (monitor)**
• bez vnitřní struktury (big mess)
• volání mezi moduly voláním podprogramů
• žádné omezení volání a vztahů mezi moduly
• uživatelský proces = podprogram jádra
• často bez rozdělení na systémový/uživatelský režim
**2. Jádro (kernel)**
• jádro běží v systémovém režimu
• procesy běží v uživatelském režimu a volají jádro (jádro je pasivní), striktní rozhraní mezi procesy a jádrem
• jádro vytváří pro proces abstrakci virtuálního počítače
**3. Mikrojádro (Mach)**
• Služby jádra částečně v systémovém režimu (mikrojádro), částečně v uživatelském režimu (systémové procesy)

*Minimální jádro - úkoly:*
- přepínání kontextu
- přidělování paměti
- ochrana paměti, nastavení adresového prostoru

*Ostatní služby řešeny samostatnými procesy nad mikrojádrem:*
- prostředí procesů, spouštění procesů
- autentizace, autorizace, účtování
- virtualizace paměti, odkládání, zavádění
- V/V
- síťové vrstvy
- systém souborů

**4. Exokernel**
Služby jsou poskytovány jako podprogramy uvnitř uživatelského procesu. Jádro v systémovém režimu je volno pouze pro synchronizaci a přidělování prostředků.

**5. Virtuální počítač (Virtual Machine)**
• jádro běží zcela v uživatelském režimu
• v systémovém režimu běží pouze monitor virtuálního počítače - zachytává a emuluje privilegované instrukce
• plně virtualizuje všechny prostředky

Princip - vrstvy s definovanou funkcí, volání pouze podřízených vrstev.
1. virtuální procesor - přepínání kontextu, synchronizace
2. přidělování paměti, uvolňování
3. plánování - přidělování procesoru, zastavení, synchronizace na vyšší úrovni

7. spouštění a ukončování procesů
8. systém souborů
9. virtualizace paměti
10. rozhraní jádra, prostředí procesu

- Makrojádro
- Makrojádro s moduly
- Mikrojádro se službami

**Režimy činnosti procesoru, přechody mezi režimy, volání jádra**

Uživatelský režim – provádění uživatelských procesů
Systémový režim – provádění kódu jádra OS

Přechody mezi režimy:
**a) Přerušení** – vždy do systémového režimu, na definovanou adresu
**b) Návrat z obsluhy přerušení** – návrat zpět (IRET)
**c) Volání jádra** – privilegovaná instrukce, přechod do systémového režimu, na pevnou adresu, parametry obvykle na **zásobníku v uživatelském adresovém prostoru**
**d) Návrat z volání jádra** – privilegovaná instrukce pro přechod do uživatelského režimu a nastavení programového čítače (PC)

**Volání jádra**
• volání jádra je realizováno speciální instrukcí (SVC, lcall, int, trap),
• jediný styk procesu s okolním prostředím,
• vše je zprostředkováno jádrem operačního systému,
• proces je zapouzdřen, operační systém pro něj vytváří iluzi virtuálního počítače

**Jak se resi kritická sekce v jadre a v podprogramech pri preruseni. Lze k tomu vyuzit semafor**

- zakázat přerušení
- zakázat preemptivní přepínání
- vzájemné vyloučení zamykaním datových struktur a povolení preemptivního přepínání jádra (jsem v KS, je mi odebrán CPU, ale zámek je pořád zamčený - nikdo nic nezmění)

**Porovnejte rezii v prepinani kontextu v ruznych implementacich vlaken a procesu. Ve ktere z implementace lze vlakna pouzit spolu s procesy.**

**Proces** je instance programu v paměti (vlastní adresový prostor), která se vykonává. Má jednoznačnou identifikaci (PID). Může být více procesů pro jeden program.
*Vztahy*: nové procesy vznikají duplikací běžícího rodičovského procesu (fork()) existuje vztah otec-syn. Nejvyšším prarodičem je proces init. Při ukončení otce se synové přesouvají k init. Při ukončení syna si otec vybere stav. Pokud otec na stav nečeká, stav visí v paměti a ze syna se stává zombie.
**Vlákno** je samostatně prováděná část programu v rámci jednoho procesu. Takto může jeden proces běžet na více procesorech paralelně. Vlákna jednoho procesu sdílí logický adresový prostor a systémové prostředky. Registry, zásobník a stav provádění programu se uchovává pro každé vlákno samostatně.
*Použití*: I/O vlákno vedle výpočtů, GUI, u více procesorů nutnost pro výkon.
*Výhody*: rychlejší než fork(), sdílení celého adresového prostoru (bez ochrany), pro zásobník nemusí řešit vícenásobný přístup.

- *N:N (1:1)* – vlákna na úrovní OS,
  - (+) volání je přímé a rychlé, plné využití multiprocessingu.
  - (-) evidence všech vláken v paměti jádra, větší režie během přepínání.
- *N:1* – OS vidí jen procesy, vlákna jsou v userspace pomocí knihoven.
  - (+) nízká režie vlákna (paměťová i časová), plná kontrola plánování (nezasahuje OS).
  - (-) blokující volání jádra se zapouzdřují, nejde použít pro více procesorů (CPU, dostane celý proces)
- *N:M* – kombinovaný přístup, OS vidí M vláken z OS.
  - (+) OS vidí jen potřebná vlákna (většinou podle počtu CPU), čekající vlákna už OS nevidí (bez režie), zároveň lze přepínat kontext i v userspace.
  - (-) velmi problematická implementace.

**Implementace vláken *(podrobněji):*** (pocet_procesu : pocet_vlaken)

• N:N (1:1) - na úrovni jádra systému, vlákna na úrovni uživatelské jsou reprezentována v jádře (OS/2, AIX < 4.2, Windows/NT, LinuxThreads, NPTL – clone()):

− režie přepínání kontextu
− jádro musí evidovat všechna vlákna (datové struktury v jádře)
+ plné zakázat přerušení, zakázat přerušení využití více procesorů v jednom programu
+ volání jádra přímá (nemusí být zapouzdřena)

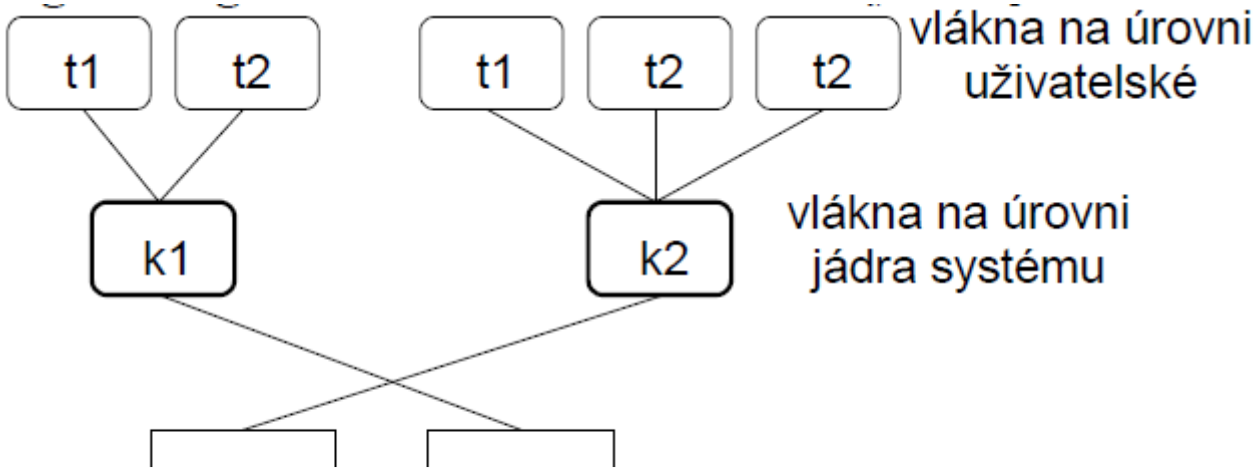
• N:1 - na úrovni knihoven, vlákna jsou plně implementována v rámci uživatelského procesu, jádro o nich nic neví (DCE, FreeBSD < 5.x)

+ nízká režie jádra, plná kontrola nad plánováním
− všechna blokující volání jádra musí být zapouzdřena
− nelze využít více procesorů v jednom programu, jednotkou přidělování času procesoru je proces

• N:M (N≥M≥1) - kombinovaný přístup, důvody:

• prováděná vlákna musí být reprezentována v jádře pro správu procesorů, nicméně nemá smysl reprezentovat všechna běžící, stačí tolik, kolik je procesorů,
• čekající (pozastavená) a připravená vlákna nemusí být reprezentována datovými strukturami v jádře, jádro o nich vůbec nemusí vědět - menší režie
• přepínání kontextu přes jádro má větší režii než v rámci uživatelského procesu – dokud lze využít přidělený procesor, probíhá běh v režimu sdílení času pro všechna aktivní vlákna v daném procesu.
• lze simulovat N:1 až N:N nastavením max. počtu vláken na úrovní jádra (thr_setconcurrency())
SOLARIS (LWP), AIX, Irix 6.5, FreeBSD 5.x (KSE) - vlákna na úrovni uživatelské lze vázat dynamicky nebo pevně na vlákna jádra systému
Problém modelů N:N a N:M – podpora vláken na úrovni uživatelské nemá dostatečné informace o akcích na straně jádra, plánování je problém.

**Light Weight Process** - vlákno na úrovni jádra systému



Implementace modelu M:N je značně složitá, LWP mají v jádře obdobnou režii jako procesy (bez adresového prostoru). LWP jsou z hlediska jádra jednotkami přidělování procesorů. Klasický proces pak běží jako 1 LWP vlákno.

**Současnost, souběžnost, atomické operace, synchronizace**

Přepínání kontextu → **souběžný** běh více procesů
**Def**.: **Atomická operace** - nedělitelná operace, nemůže být přerušena uprostřed.
**Význam**: Pokud jsou prováděny všechny operace nad sdílenou datovou strukturou atomickými operacemi, zůstává stav struktury konzistentní i při paralelním přístupu

**Synchronizace**: zajištění kooperace mezi paralelně (souběžně) prováděnými procesy.

**Vzájemné vyloučení, podmínky, které musí splňovat obecné řešení, vztah k přidělování procesoru**

**Synchronizace -** zajištění kooperace mezi paralelně (souběžně) prováděnými procesy
**Vzájemné vyloučení (mutual exclusion)** – základní úloha synchronizace: pouze jeden proces může provádět danou operaci. Vytváříme tím složitější atomickou operaci (nedělitelná operace, nemůže být přerušena uprostřed)
**Kritická sekce** – kód, jehož provádění je vzájemně vyloučené.
**Omezující podmínky:**
1. Sekce start a výpočet mohou být libovolně dlouhé, trvat libovolnou dobu, včetně nekonečné (proces zde může skončit).
2. Kritická sekce je provedena vždy v konečném čase (proces zde nesmí zůstat čekat, ani skončit).
3. Proces musí mít zaručen nekonečně krát vstup do kritické sekce v konečné době (liveness).
4. Přidělování procesoru je nestranné, spravedlivé (weak fairness).
**Fairness (spravedlnost) přidělování procesoru:**
• **unconditional fairness (nepodmíněná)** - každý aktivní nepodmíněný atomický příkaz bude někdy proveden (může být dlouhodobě prováděn pouze jeden proces)
• **weak fairness** - unconditional fairness + každý aktivní podmíněný atomický příkaz bude proveden za předpokladu, že podmínka nabude hodnoty TRUE a nebude se měnit.
• **strong fairness** - unconditional fairness + každý podmíněný atomický příkaz, jehož podmínka se nekonečně častokrát mění, bude nekonečně krát proveden (prakticky nerealizovatelný plánovací algoritmus, musel by střídavě provádět po jedné atomické instrukce jednotlivých procesů).
**Formální požadavky na hledané řešení:**
• bezpečnost (safeness): v daném případě zaručuje vyloučení
• živost (liveness):
- nedochází k uváznutí (deadlock)
- nedochází k blokování (blocking)
- nedochází k stárnutí (starving)

**Živost, uváznutí, blokování a stárnutí (hladovění)**

**Živost (liveness)** - algoritmus je živý, pokud je bezpečný a nedochází k uváznutí, blokování a stárnutí **(je zaručeno jeho dokončení v konečné době).**

**Stárnutí (starving)** - proces může čekat v synchronizaci na stav, který nemusí být nikdy pravdivý v okamžiku testování. Není striktně omezena horní mez čekání (jinak jako blokování). V praxi se obvykle toleruje, závisí na plánovacím algoritmu.

**Uváznutí při přidělování prostředku, podmínky uváznutí, možnosti řešení**

Uváznutí - čekání na událost, která nemůže nastat díky čekání || stav, kdy dochází k čekání na událost/prostředek, která nikdy nenastane/se nepřidělí.

Máme dva druhy prostředků:

- SR (Serially Reusable): opakovaně použitelné
- CR (Consumable Resources): jednorázově použitelné

**Nastává při:**

- Zamykání semaforů, zámků, souborů, …
  - P1: lock(S1); lock(S2) … P2: lock(S2); lock(S1)
- Přidělování paměti
  - pokud P1 a P2 chtějí dvě jednotky z M, kdy M má právě dvě jednotky
- Zasílání zpráv
  - Všechny procesy čekají na zprávu, kterou nikdo nepošle kvůli čekání

**Příčiny:**

- Pouze jeden proces může využívat prostředek
- Proces se nevzdá přidělených prostředků při neúspěšné alokaci dalších prostředků
- Proces získává prostředky sekvenčně
- Prostředek nemůže být preemptivně odebrán, odevzdává jej proces

**Vzájemné vyloučení u jednoprocesorových systémů**

**Atomický kód** – úsek kódu, kde nemůže dojít k přepnutí kontextu (multiprogramování) nebo přerušení (souběžný běh ovladače a procesů)

**Vzájemné vyloučení mezi:**

1. **procesy/vlákny a obsluhou přerušení** (zakázání přerušení). zaručuje, že obsluha přerušení nenaruší KS, komunikace s řadičem trvá moc dlouho (vypnutí a zapnutí má vysokou režii)
2. **různými procesy/vlákny v jádře** (zakázání přepnutí kontextu): **synchronně** (zahájení čekání, spuštění jiného procesu); **asynchronně**, na externí událost (přerušení od časovače, apod.)
   a) zakázat přerušení – blokuje přepínání kontextu, ale i I/O.
   b) zakázat preemptivní přepínání kontextu v jádře - funguje explicitní přepínání, ale není výkonově vhodné pro systém.
   c) vzájemné vyloučení zamykáním datových struktur (bin. semafor) a povolení preemptivního přepínání jádra
3. **uživatelskými procesy.** futex (fast user-space mutex). dříve používaly nástroje v jádře – semafory, apod.

ostatní procesory běží současně a mohou také provádět kód jádra).

**1.) Krátkodobé vzájemné vyloučení (spin_lock).**

Může střežit pouze kritické sekce, které jsou krátké, neblokující a bez preempce (proces/vlákno nesmí být pozastaveno, nesmí na nic zahájit čekání). Aktivní čekání je v tomto případě přijatelné, protože pak může být kritická sekce obsazena pouze procesem běžícím na jiném procesoru a ten ji brzy uvolní. Pozastavení procesu by bylo náročnější než krátké aktivní čekání (a vyžadovalo by opět vzájemné vyloučení). Krátkodobé vyloučení je nutné pro implementaci synchronizačních nástrojů (mutex, semafor, atd.).

**a) Implementace pouze čtením/zápisem**

• elegantní algoritmy pouze pro malý počet procesů, složitost

• dostupnost lepších speciálních atomických instrukcí

**b) Speciální atomické instrukce**

Nutná atomická instrukce nedělitelného čtení a zápisu (RMW) do paměti (musí korektně fungovat ve víceprocesorovém systému se sdílenou pamětí!)

**Problémy**

cyklus test&set zatěžuje pam. sběrnici (stále čte a zapisuje)

u HT se blokuje druhý kontext aktivním čekáním (také možnost stárnutí) -> řešením je využít cache a pro HT spec. instrukce

**2.) Dlouhodobé vyloučení.** Využívá bin. zámek (první sekce zamkne a odemkne při odchodu). Impl. pomocí test&set.

## Řešení problému nežádoucí zátěže paměti

Data se čtou z cache, která je při zápisu jiným procesorem automaticky invalidována - jen za těch okolností se během čekání čte z paměti.

```
while(test_and_set(&lk->lock)){
   while (lk->lock != 0); // při čekání se pouze čte cache
}
```

**Inverze priority, rekurzivní zámky**

Inverze priority - zvýšení priority procesu v kritické sekci. Inverze priority je treba kdyz proces s nižší prioritou blokuje provádění procesu s vyšší prioritou. Proces s vyšší prioritou (h) nemůže vstoupit do kritické sekce, protože je v kritické sekci proces s nižší prioritou (l) a neběží, protože jsou v systému procesy s prioritou p > l (pokud p < h, jedná se o inverzi priority těchto procesů proti h).

**Řešení**

**Dědění priority ( *priority inheritance)***

- při vstupu procesu do KS je zvýšena priorita tohoto procesu na úroveň max. priority čekajících procesů na KS
- + priorita se nemění, pokud je proces osamocen
- - potřeba přepočítat max. prioritu při každém blokujícím čekání

**Horní mez priority (*priority ceiling)***

**Použito pro…**

- **Ano:** bin. semafor, mutex, monitor
- **Ne:** obecný semafor, condition (není jasné, kdo blokuje)

**Rekurzivní zámek** - složité knihovny, např. standardní V/V pro C:
printf() - musí zamknout stdout. printf() volá putchar() - musí zamknout stdout, nastává uváznutí → pokud je zámek zamčen stejným procesem, pouze se inkrementuje čítač úrovně rekurze, při odemykání se zmenšuje, zámek se odemkne až při 0

**Klasické synchronizační úlohy, řešení různými nástroji**

**1. Vzájemné vyloučení (Mutual Exclusion)**

**2. Producent/konzument (Producer/Consumer, Bounded Buffer)**

- producenti produkují data do sdílené paměti, konzumenti je z ní odebírají
- konzumenti musí čekat, pokud nic není vyprodukováno
- producenti musí čekat, pokud je paměť plná
- operace s pamětí musí být synchronizovány

```
semaphore_t empty, full;
mutex_t mutex;
DATA v;
shared buffer[N];
shared int get, put;

init(empty, 0);
init(full, N);
init(mutex, 0);
...
producent                    konzument
while (1) {                  while(1) {
    v = produkuj data;           down(empty);
    down(full);                  lock(mutex);
    lock(mutex);                 v = buffer[put];
    buffer[get] = v;             put = (put+1)%N;
    get = (get+1)%N;             unlock(mutex);
    unlock(mutex);               up(full);
    up(empty);                   zpracuj data v;
}                            }
```

**3. Čtenáři/písaři (Readers/Writers)**

- přístup ke sdíleným datům
- čtenář pouze čte data
- písař čte a zapisuje
- vzájemné vyloučení všech je příliš omezující:
- více čtenářů současně
- pouze jeden písař

```
mutex_t read, write;
int readers;

init(read, 0);
init(write, 0);
...
čtenář                       písař
while (1) {                  while(1) {
    lock(read);
    if (++readers == 1)
        lock(write);             lock(write);
```

```
    if (--readers == 0)
        unlock(write);          unlock(write);
    unlock(read);
}                           }
```

**4. Pět filozofů (Dining philosophers)**

5 filozofů, 5 vidliček, 5 talířů

Problémy:

• vyhladovění – je třeba zajisti, že když chce jíst, dostane v konečném čase najíst a nebude systematicky předbíhán jinými filozofy

• uváznutí – všichni přijdou ke stolu a uchopí levou vidličku, nikdo nemůže jíst

**Bez uváznutí:**

```
mutex_t mutex;
while (1) {
    myslí
    lock(mutex);    // atomické získání vidliček
    down(fork[i]);
    down(fork[(i+1)%5];
    unlock(mutex);
    jí
    up(fork[i]);
    up(fork[(i+1)%5];
}
```

Pokud je mutex silný, nenastává hladovění, ale je neefektivní (může být dostatek vidliček pro jiného filozofa u stolu, ale *mutex* zůstane zamčený a tím brání ostatním jíst, např. vidl. 2/3 zrovna jí, vidl. 1 dostanu, na přidělení vidl. 2 zůstanu čekat, vidl. 4/5 jsou volné, ale nebudou použity, dokud se neuvolní vidl.2)

**Lepší řešení – nepustíme ke stolu více než 4:**

```
semaphore_t fork[5], total;
init(fork[*], 1)
init(total, 4);     // povolí max. 4 přidělení
while (1) {
    myslí
    down(total);
    down(fork[i]);
    down(fork[(i+1)%5];
    jí
    up(fork[i]);
    up(fork[(i+1)%5];
    up(total);
}
```

Pokud jsou semafory silné, řeší i hladovění, jinak je třeba doplnit (použít sdílené proměnné).

**Přidělování procesoru, plánování, stavový diagram procesů**

• **plánování** (scheduler) - volba strategie a řazení procesů
    • sdilený plánovač
    • samostatný plánovač

• **přidělování** (dispatcher) - přepínaní kontextu na základě naplánování

**Cíle plánování:**

• Minimalizace doby odezvy

• Efektivní využití prostředků

• Spravedlivé dělení času procesoru

• Doba zpracování

• Průchodnost (počet úloh/čas)

**Přidělování procesoru** = přepínání kontextu podle plánování

Založen na prioritě procesů, tedy procesor je přidělen procesu s nejlepší prioritou. Definován třemi charakteristikami:

1. **Interval rozhodování**
   a. Nepreemptivní - proces běží do ukončení nebo čekání
   b. Preemptivní - procesu může být odebrán procesor v časových kvantech, odblokování procesu s vyšší prioritou nebo příchod nového procesu
   c. Selektivní preempce - některé procesy mohou přerušovat nebo být přerušeny
2. **Prioritní funkce**, priorita určena na základě několika parametrů:
   a. Paměťové požadavky
   b. Spotřebovaný čas procesoru
   c. Doba čekání na procesor
   d. Doba strávená v systému
   e. Externí priorita
   f. ...
3. **Výběrové pravidlo**
   Výběr z více procesů se stejnou prioritou pomocí výběrového algoritmu

**Plánovací algoritmy pro dávkové systémy**

prioritní funkce P(a,r,t,d)

vysvětlivky:

- a - spotřevovaný čas cpu
- r - čas strávený v systému
- t - celkový čas procesoru
- d - perioda opakování procesu

algoritmy:

1. FIFO: P(r) = r
2. LIFO: p(r) = -r
3. SJF (Shortest Job First): P(t) = -t
4. SRT (Shortest Remaining time): p(a,t) = a - t
5. Statická priorita: P(i) = konstanta i dle procesu
6. RR (Round Robin): P() = konstanta (vždycky následuje ten následující, např. 1->2, 2->3, 3->1)
7. MLF (MultiLevel Feedback): prasárna
8. RM (Rate Monotonic): P(d) = -d
9. EDF (Earliest Deadline First): P(r,d) = -(d - r%d)

**Deterministický paralelní systém, nezávislost, interference, Bernsteinovy podmínky**

Bude výsledek paralelního systému při paralelním provádění vždy stejný bez ohledu na posloupnost provádění?
Pokud ano, pak nazýváme paralelní systém časově nezávislým, deterministickým.

Def.: Paralelní systém je deterministický, jestliže pro daný
počáteční stav s0 je Vx(α) = Vx(α'), 1 ≤ x ≤ m, pro všechny

Def.: Bernsteinovy podmínky neinterference:
Dva procesy Pi a Pj jsou neinterferující, jestliže platí:
1. Pi < Pj nebo
2. Pj < Pi nebo
3. R(Pi) ∩ W(Pj) = W(Pi) ∩ R(Pj) = W(Pi) ∩ W(Pj) = ∅
Věta: Paralelní systém skládající se ze vzájemně neinterferujících procesů je deterministický.

Nezávislé procesy mají R(Pi) ∩ W(Pj) ≠ ∅

**Binární semafor, operace, implementace, použití**

Operace

- init(sem, v) – inicializace semaforu sem na hodnotu v = 0, 1
- lock(sem) – zamčení, čekání na nulovou hodnotu a nastavení v na 1
- unlock(sem) – odemčení, nastavení v na 0 a odblokování procesů čekajících v lock()

nelze číst hodnotu (může se změnit)
čekání v lock(sem) je pasivní
lock(sem) a unlock(sem) jsou atomické
odemykat může jiný proces než zamknul (předávání zámku)

silný/slabý semafor – nepodléhá/podléhá stárnutí
Mutex (mutual exclusion) – speciální binární semafor určený pouze pro vzájemné vyloučení, při zamčení má identifikovaného vlastníka, pouze vlastník ho může odemknout (nutné pro řešení inverze priority)

**Použití**
a) vzájemné vyloučení

```
init(sem, 0); /* volný */
while (1) {
   lock(sem); ENTRY
   *kritická sekce*
   unlock(sem); EXIT
   výpočet
}
```

b) signalizace událostí
**nevhodné -** řešeno obecným semaforem

**Obecný semafor, operace, implementace, použití**

Počáteční hodnota určuje „kapacitu“ semaforu – kolik jednotek zdroje chráněného semaforem je k dispozici. Jakmile se operací down() zdroj vyčerpá, jsou další operace down() blokující, dokud se operací up() nějaká jednotka zdroje neuvolní.

- interní hodnotou je celé číslo
- pokud je nula, je zamčen a čeká se na navýšení hodnoty

**Operace**
init(sem, v) – inic. sem. sem na hodnotu v >= 0

pamatuje si počet up() a down()

**Použití**
a) vzájemné vyloučení - ekvivalentní bin. semaforu
b) signalizace událostí

```
init(sem, 0);
P1:      P2:
...      up(sem);
down(sem);   /* čeká až P2 provede up() */
```

**oproti bin. sem. bezpečné (žádná událost se nemůže ztratit)**

c) hlídání zdroje s definovanou kapacitou N:

```
init(sem, N);
Pi:
down(sem); /* pokud je volno, pokračujeme dále */
... /* nejedná se o kritickou sekci, je zde až N procesů současně! */
up(sem); /* uvolníme místo */
```

**Implementace** v UNIXových sys. není -> používá se monitor. Obecně implementován pomocí spinlocku a testování hodnoty čítače pozastavené procesy se přidávají na konec fronty.

**Simulace**

- binární sem. lze simulovat dvěma číselnými se sdílenými proměnnými
- obecný sem. lze simulovat třemi binárními a sdílenou hodnotou

**Monitory, čekání, řešení uvolnění čekajících, srovnání se semafory**

**Monitor** je abstraktní datový typ, který zajistí vzájemné vyloučení operací nad monitorem = všechny operace monitoru jsou atomické. Skládá se z dat, ke kterým je potřeba řídit přístup, a množiny funkcí, které nad těmito daty operují. Odděluje čekání a operace se sdílenými proměnnými.

**Podmíněné proměnné**
slouží k čekání uvnitř monitoru (proces v monitoru čeká na splnění nějaké podmínky)
Když funkce monitoru potřebuje počkat na splnění podmínky, vyvolá operaci ***wait*** na podmíněné proměnné, která je s touto podmínkou svázána. Tato operace proces zablokuje, zámek držený tímto procesem je uvolněn a proces je odstraněn ze seznamu běžících procesů a čeká, dokud není podmínka splněna. Jiné procesy zatím mohou vstoupit do monitoru (zámek byl uvolněn). Pokud je jiným procesem podmínka splněna, může funkce monitoru „signalizovat“, tj. probudit čekající proces pomocí operace ***notify***. Operace notify budí jen ty procesy, které provedly wait na stejné proměnné.
**Srovnání**
Používá se namísto obecných semaforů v UNIXu (vyloučení i signály). Snazší používání (vyšší úroveň abstrakce), z pohledu programátora transparentní (jen volá funkce, zamykání řeší monitor).

```
resource sem:.value: integer;
procedure down;
        begin
          region sem when value > 0 do value:=value-1;
        end;
procedure up;
        begin
          region sem do value:=value+1;
        end;
```

**Implementace pomocí semaforů**

```
lock(mutex); /* binární/obecný */
++waiting;
while (!C) {
  unlock(mutex);
  down(block); /* obecný */
  lock(mutex);
}
--waiting;
KS;
for (i = 0; i < waiting; i++) up(block);
unlock(mutex);
```

**Bariéra** - čekání na n procesů

**Implementace pomocí semaforů:**
• čítač s počátečním stavem N
• proces v bariéře dekrementuje čítač a pokud není nula, čeká na obecný blokující semafor
• až přijde poslední, čítač je nula, musí odblokovat všechny čekající (N-1 krát operace up())
**Problém:**
• blokovaný proces nemusí stihnout provést down(b), další odblokuje bariéru cyklem up() a bude tam o jednu signalizaci více, takže vstup do následující bariéry nebude blokující!

**Řešení - dva čítače a dva blokující semafory, jednou použít jeden pár, pak druhý pár (nebo jeden pro vstup, druhý pro výstup)**

## Zasílání zpráv, adresace, použití pro synchronizaci

**synchronní** - odesílatel čeká na přijetí zprávy příjemcem (CSP)
**asynchronní** - bez čekání, proces může pokračovat
**Adresace:**
- **explicitní (přímá):** send(p, msg), receive(q, msg) - p, q jsou procesy
- **implicitní:** send(msg), receive(msg) - komukoli, od kohokoli
- **nepřímá**: send(m, msg), receive(m, msg) - m je schránka, port

**Problémy implementace:**
- priorita zpráv a výběr podle priority
- velikost zpráv a efektivní kopie mezi adresovými prostory
- vyrovnávací paměť (0 = rendezvous)

```
    receive(lock, msg);
    KS;
    send(lock, msg);
    ...
}
```

**Zápis paralelních systémů a paralelismu, příklad zápisu paralelního systému.**

**1. P/S**

P(P1, P2) paralelní provádění P1 a P2
S(P1, P2) sériové provádění P1, P2

**2. cobegin/coend**

Dijkstra: parbegin/cobegin
cobegin co
P1|P2|P3; P1 // P2 // P3;
coend oc
Lamport: < A; B; C; > atomická operace

**3. fork/join**

fork label spuštění procesu od návěští
quit ukončení procesu
join m,label m=m-1, když m==0 skok na návěští

Priklad zapisu : S(P(S(P(t1=a*b, t2=c*d), t4=t1+t2), t3=a/e), r=t4+t3)

**Metody verifikace paralelních sysémů, stavový prostor paralelního systému, konstrukce stavového prostoru a ověřování živosti ve stavovém prostoru**

**Stavový prostor**

Kripkeho model (S, R, L) - stavový prostor paralelního systému
S - konečná množina stavů
R - množina přechodů mezi stavy S
L - značení, definuje hodnotu atomických tvrzení pro každý stav
cesta - posloupnost σ = s1, s2, s3, ... (si, si+1) ∈ R
Problém: počet stavů je exponenciální - redukce

**Značení - (příkaz, příkaz, stav flag1, stav flag2)**

**Konstrukce** - Začneme od počátečního stavu a přidáváme všechny stavy, do kterých se může systém dostat, přičemž stavy se stejným značením tvoří jeden uzel. Z každého uzlu vede cesta dvěma směry (máme 2 procesy, pro více vícerozměrný prostor), směr vpravo = postup Q, dolů = postup P.
**Bezpečnost** = neexistuje cesta do stavu označeného EE**
**Živost** = neexistuje nekonečný cyklus obsahující alespoň jeden vertikální a současně alespoň jeden horizontální přechod a neprocházející kritickou sekcí v původním grafu, ani v grafu vzniklém po zablokování jednoho procesu v sekci výpočet (reprezentujeme změnou přechodů C->D*** na C->C***).
**Uváznutí a stárnutí** = nekonečný cyklus neprocházející

**Organizace paměti, společný/oddělené adresové prostory**

A. organizace LAP
1. monoprogramové systémy
2. multiprogramové systémy
    a. společný LAP
    b. oddělený LAP

B. mapování LAP na FAP
    1. úseky
    2. segmenty
    3. stránky

**A1) Jeden souvislý úsek paměti**

LAP(Pi) = FAP, jeden program v paměti

- Monoprogramování bez ochrany paměti - jeden adresový prostor, program je zaváděn na pevnou nebo proměnnou adresu.
- Jádro systému - pevně přidělena část paměti, zbytek dostupný pro procesy (MS-DOS), ochrana mezním registrem (HP RTE-II)
- Segmentování (překryvné segmenty, overlay) - zavádění programu po částech, podle potřeby

**A2a) Společný adresový prostor procesů**

LAP(Pi) = LAP(Pj) = FAP pro všechny programy v paměti
Jeden adresový prostor, více programů v paměti na různých adresách, lze kombinovat s následujícími technikami zobrazování LAP do FAP
- vyžaduje dynamickou relokaci programů na danou adresu
- omezený sdílený LAP (ale pro 64bitové procesory nevadí)
+ jednoduchá správa
+ jediná tabulka stránek při kombinaci se stránkováním
+ přidělování paměti viz přidělování úseků, stránkování

**Multiprogramování** - nutná ochrana paměti:
- ochrana uživatelských programů navzájem
- ochrana jádra systému

**Implementace ochrany:**
- mezní registry
- chráněný režim činnosti procesoru

**A2b) Oddělené adresové prostory procesů**

LAP(Pi) != LAP(Pj) != FAP - zobrazení, mapování
Každý proces má k dispozici celý logický adresový prostor. Logický adresový prostor je tranformován (mapován) na fyzický některou z následujících transformací. Nutná podpora hardware (převod logické adresy na fyzickou je nutný pro každou instrukci).

**Adresový prostor jádra:**

a) oddělený - nutný přepočet adres parametrů volání jádra, přístup do jiného adresového prostoru pro čtení/zápis parametrů a dat (bufferů) každého volání jádra
b) sdílený s procesem, který volá jádro - horní část LAP rezervovaná pro jádro, v uživatelském režimu chráněná (i proti čtení), v systémovém režimu zde běží jádro (Unix, Windows/NT)
- problém 32bitového adresového prostoru, rezerva pro jádro omezuje maximální velikost uživatelských dat, jádro nemá moc prostoru pro buffery (obvykle 512-1024 MB)

**Správa úseků proměnné velikosti**

Zdroj má omezenou kapacitu, je přidělován po úsecích proměnné velikosti. Při uvolňování vznikají volné úseky.

**Cíle:**

 Efektivní využití zdroje o omezené kapacitě – pokud není zdroj zcela využit, musí být volné úseky použitelné k uspokojení požadavků.

 Rychlá alokace zdroje – minimální čas nutný pro nalezení dostatečně velkého volného úseku.

Použití – přidělování systémové paměti, swap, LAP, apod.

**Problém**: Který volný úsek použít pro uspokojení požadavku o velikosti N? Pokud je větší, přidělit jej celý nebo rozdělit?

**Přidělování:**

**First Fit** - přiděluje první postačující úsek

**Next Fit** - přiděluje první postačující úsek, příště pokračuje od místa posledního přidělení

**Best Fit** - přiděluje minimální postačující volný úsek, minimální interní fragmentace

**Worst Fit** - přiděluje vždy největší volný úsek, menší externí fragmentace malých úseků, ale nepříjemná fragmentace velkých úseků

Při uvolňování vznikají prázdné úseky o různých velikostech - problém evidence volných úseků:

- seznam řazený podle pořadí uvolnění – FIFO, LIFO (nejjednodušší)
- společný seznam organizovaný podle adres (vhodné pro spojování)
- samostatný seznam volných bloků
- seznam organizovaný podle velikosti (náročné zařazování)

Spojování volných úseků:

- nikdy
- při alokaci
- při uvolnění
- při neúspěšném přidělení (garbage collection)

Vnitřní struktura souboru (na úrovni rozhraní V/V)

- pole alokačních bloků
- alokační blok = několik souvislých sektorů (512 byte) na disku

Alokace:

- souvislá
- lineární seznam
- Index-sekvenční (pevné alokační bloky)
- Index-sekvenční (proměnné alokační bloky)

Volba velikosti alokačního bloku, měření viz popis BSD FFS:

- 512 B - 95% využití diskového prostoru
- 1 KB - 90 %, pomalé čtení/zápis 20-50 KB/s
- 2 KB - 80 %
- 4 KB - 60 %, rychlé 140-220 KB/s

Optimalizace:

**Stránkování - princip, funkce, organizace tabulky stránek, TLB**

**Organizace tabulky stránek**

Problém realizace tabulky stránek v HW

1. Transformace LAP na FAP se musí uskutečnit při každé instrukci a adresaci operandu (načtení instrukce, operandy)

– rychlost provádění instrukcí < 1 ns

– rychlost přístupu do paměti = 50 ns

– adresu nelze transformovat indexováním a čtením tabulky stránek z operační paměti!

2. Zobrazení plného 32bitového adresového prostoru - 1M položek (4 MB pro stránku 4 KB)

– malý program by vyžadoval plnou tabulku stránek (kód na začátku adresového prostoru, zásobník na konci)

– takto velkou tabulku stránek nelze umístit do rychlé interní paměti na procesoru (nehledě na problém jejího nastavování po přepnutí kontextu)

– pro 64bitový adresový prostor nereálné i uložení v paměti

**Volba optimální velikosti stránky**

cílem je minimalizovat režii interní fragmentace

s - priemerna velkost procesu

e - velkost polozky tabulky

p = sqrt(2se)

Možné organizace:

- standardní jednoúrovňová tabulka stránek - Položkou tabulky pro danou stránku je přímo adresa do FAP, kde stránka začíná.
- víceúrvňová tabulka stránek - Adresa stránky je rozdělena na části. První část ukazuje to tabulky stránek 1. úrovně, ta obsahuje adresu, kde se nachází tabulka stránek druhé úrovně. V tabulce druhé úrvně se podle druhé části adresy najde vlastní fyzická adresa stránky.
- inverzní (hashovaná) tabulka stránek - tabulka není indexovaná adresou stránky, ale adresou rámce. Kde se daná stránka nachází, se zjistí prohledáním tabulky.

Transformace LAP na FAP při každé instrukci, což je velmi náročné. Používá se proto cache naposledy hledaných stránek TLB (Translation Lookaside Buffer).

viz opora strana 161-164

- redukce prostorove režie - různé způsoby alokace (související bloky vedle sebe a načítat dopředné načítání), poskládání bloků za sebou, indexace
- redukce casove rezie - ve funkcích v cyklu, nesmí se zbytečně odkládat často používané stránky (nevhodné LIFO)

**V jaké formě muze byt rozhrani (binarni a ... , to bylo v prvni prednasce)**

- **binarni** - assembler, HW zavislé; standardy: iBCS-2 (Intel), MIPS ABI (SGI)

**Definice rozhraní jádra systému, standardizace**

Definice rozhraní jádra

A) Na úrovni binární:

```
# na zásobníku fd, buf, length
read: lea $0x3,%eax
        lcall $7,$0 (Linux - int $0x80)
        jb error
        ret
error: movl %eax,_errno
        movl $-1,%eax
        ret
```

Binární rozhraní je systémově závislé (procesor, verze systému)

Standardy: iBCS-2 (Intel), MIPS ABI (SGI), apod.

B) Na úrovni zdrojové:

```
read(fd, buffer, length); /* pro C/C++ */
```

Historicky také systémově závislé (např. datový typ length)

**Standardiazce**

Sjednocování rozhraní řeší *přenositelnost* – signály, terminály, synchronizace,…

**SVID** – UNIX System V Interface Definition

- definice rozhraní originálního komerčního UNIXu z AT&T

**X/Open (Open Group) –** X/Open Portability Guide (XPG)

- sdružení výrobců, širší definice rozhraní jádra, bez omezení na konkrétní verzi UNIXu

**IEEE POSIX 1003.1**

- rozhraní jádra systému pro jazyk C
- součást standardů POSIX
- mělo více revizí, přidávání vlastností (vlákna)

**Vlákna POSIX 1003.1c - vytváření vláken, ukončení, přebrání stavu**

```
int global; /* sdílená proměnná */
void *vlakno(void *arg)
{
        int local; /* privátní prom.vlákna */
        static global; /* sdílená proměnná */
        ... /* kód vlákna */
        return stat; /* ukončení vlákna */
        /* pthread_exit(void *stat); /* ekvivalentní*/
}
```

Vytvoření a spuštění vlákna – pthread_create():
#include <pthread.h>
int pthread_create(pthread_t *thread,
const pthread_attr_t *attr, /* atributy */
void *(*func)(void *), /* vlákno */
void *arg); /* parametr */

Čekání na ukončení vlákna a převzetí stavu ukončení:
int pthread_join(pthread_t thread, void **st);

**#WIKI FITUSKA !!!**

## 3. Porovnání paralelního programování na úrovni vláken a procesů v Unixu, co jsou jednotky paralelního provádění, jak vznikají a zanikají.

na úrovni vláken:

- jednotkou je vlákno
- možnost využít na výpočet více procesorů (v rámci 1 procesu)
- vlákna jednoho procesu sdílí haldu a statická data, nesdílí zásobník
- vzniká fcí pthread_create
- zaniká po návratu z funkce vlákna, nebo po volání pthread_exit, nebo při skončení celého procesu

na úrovni procesů:

- jednotkou je proces
- možnost využít jenom 1 procesor (v rámci 1 procesu)
- procesy mezi sebou sdílí jen vybrané segmenty:
  - implicitní – sdílení kódu procesů, sdílených knihoven
  - explicitní – shm_open, mmap
- vzniká syscallem fork()
- zaniká návratem z main(), voláním exit, obdržením SIGKILL apod.

## 4. Proč nelze použít binární semafory k obecnému čekání na událost? Jak to lze obejít – uveďte příklad řešení (operace čekej(), signalizuj()).

Teoreticky by šlo použít binární semafory s nulovou počáteční hodnotou. Čekání na událost by se řešilo zamknutím semaforu (down()), zaslání události pak jeho odemčením z jiného procesu (up()).

Pokud by ale událost odemčení přišla ve chvíli, kdy by zrovna nikdo nečekal, událost by v případě binárního semaforu byla ztracena.

Řešením je použití obecného semaforu. V případě příchodu dvou událostí pak nabude semafor hodnoty 2 a následná dvě čekání na signál jsou odbavena ihned.

```
semaphore S1,S2,S3;
int value;

init:
```

```
  lock(S1);
  value--;
  if (value < 0) { /* musime cekat */
    unlock(S1);
    lock(S2); /* čekání na signál */
  } else {
    unlock(S1);
  }
  unlock(S3);

up: /* signalizuj() */
  lock(S1);
  value++;
  if (value <= 0){ /* někdo čeká */
    unlock(S2);
  }
  unlock(S1);
```

## 6. Princip segmentace a stránkování, jaký typ fragmentace přitom nastává, jak se řeší.

**Stránkování - první způsob mapování LAP na FAP (stejně velké stránky)**

- L pam. dělena na stránky, logická adresa rozdělena na dvě části: <číslo stránky;offset>
- F pam. dělena na rámce (1-16KB)
- FA = f(proces, LA >> log2(velikost_ramce)) + LAoffset
- transformace FA na LA tabulkou stránek (cachována do TLB)
- paměť přidělována po rámcích, lze přidělit libovolné rámce, nemusí být vedle sebe
- nenastává externí fragmentace
- interní nastává, ale průměrně jen 1/2 stránky (maximální je velikost stránky mínus 1 byte)

**Segmentace - druhý způsob mapování LAP na FAP (různě velké segmenty)**

- FA = f(proces, segment) + LAoffset
- přiděluje 1 procesu více různě dlouhých úseků (segmentů)
- LA se skládá z segmentu, offsetu - explicitní (každá LA obsahuje adr. segmentu nebo implicitní (segment někde v registru)
- tabulka segmentů - mapuje seg. na úseky fyz. pam. (báze - FA, délka)- lokální (pro každý proces), globální
- program je pak soubor segmentů
- problém externí fragmentace
- řeší se pomocí "Setřásání"

## 7. K čemu slouží sdílení kódu procesů, jak je řešeno v systémech se stránkováním, jakou vyžaduje podporu a jaké výhody (nevýhody) má.

Sdílení kódu procesů

- kód musí být přístupný pouze pro čtení – reentrantní (nesmí být za běhu modifikován).
- procesy řízené stejně relokovaným programem
- vliv na stránkování – vícenásobné použití stránek
- datové struktury – správa sdílených částí LAP

## 9. Co je přepnutí kontextu, jak probíhá, rozdíl mezi preemptivním a nepreemptivním přepínáním kontextu.

- Přepnutí kontextu – pozastavení prováděného procesu a pokračování v provádění jiného pozastaveného procesu = předání procesoru mezi procesy:
- Preemptivní – bez spoluúčasti procesů, procesu může být procesor odebrán
- Nepreemptivní – kooperativní předání procesoru, proces se procesoru vzdá

Kdy nastává přepnutí kontextu:

- vyčerpání časového kvanta (plánování v režimu sdílení času)
- zahájena blokující operace (čtení, zápis)
- proces je pozastaven (sleep, wait)
- má běžet proces s vyšší prioritou

Kdy se přepíná kontext v jádře (jednoprocesorovém)?

- synchronně (zahájení čekání, spuštění jiného procesu) – není problém, datové struktury mohou být v konzistentním stavu
- asynchronně, na externí událost (přerušení od časovače, apod.) – pokud zabráníme, je kód jádra nepreemptivní (nemůže se spustit jiný proces, dokud se předchozí synchronně nezastaví).

Přepínání kontextu - jiný adresový prostor, nutné vyprázdnění TLB → doplnění čísla procesu (adresového prostoru) do položky TLB (SPARC, Intel IA-32e).

**Průběh bude asi podobný jako u přerušení:**

Zpracování přerušení:

- dokončení právě prováděné instrukce
- zablokování dalších přerušení (všech nebo menší priority)
- uložení stavu procesoru (registry, stav, adresový prostor)
- skok do obslužného podprogramu přerušení v systémovém režimu činnosti procesoru:
- pevná adresa
- vektor podle zdroje přerušení
- obsluha přerušení (může být přerušena přerušením vyšší prio)
- obnova stavu procesoru (pokračování v přerušené práci)

## 12. Metody prevence uváznutí při přidělování SR prostředků (uveďte alespoň 3).

1. Povolit sdílené použití prostředků (spooling) – porušení podmínky výlučného používání

2. Přidělit prostředky jen jednomu procesu (silně omezující) – nemůže vzniknout cyklus v grafu alokace prosředků

3. Přidělovat vždy všechny požadované prostředky najednou jedním požadavkem (zbytečné blokování prostředků, které nejsou využity po celý běh procesu) - dtto

4. Vzdání se přidělených prostředků při požadavku na další (aplikovatelné u zámků v databázích, ale ne u prostředků typu tiskárna) – dtto

6. Žádat prostředky vzdy ve vzrůstajícím pořadí očíslovaných tříd prostředků (nejpoužívanější metoda, nemůže vzniknout cyklus, locking order)

## 14. Synchronní a asynchronní provádění V/V v Unixu.

Synchronní V/V - po dobu V/V je proces pozastaven

Asynchronní V/V – proces může mít více zahájených V/V operací, dokončení lze testovat nebo je oznámeno signálem. Lze simulovat pomocí neblokující operace (POSIX).

## 15. Metadata v systémech souborů – co obsahují, problém odolnosti vůči výpadku a způsoby řešení.

Obsahují (na BSD):

- typ souboru
- práva, majitel, skupina
- počet odkazů (hard link)
- velikost
- čas posledního přístupu, změny, změny stavu
- indexy alokačních bloků

Odolnost proti výpadku:

- synchroní zápis metadat - prostě se vždycky zapisuje všechno naráz
- soft metadata update - zápis diskových bloků v tom pořadí, jak to vyžaduje V/V vyrovnávání, ale se zaručenou konzistencí metadat na disku dle grafu závislosti metadat uložených v paměti
- transakční zpracování - klasické transakce jak v DB, informace o změnách v žurnálu

## 16. Algoritmy plánování u dávkových systémů, popsat pomocí univerzálního

### Univerzální plánovač je definován třemi charakteristikami

1. interval rozhodování (časové okamžiky, ve kterých je aktivován plánovač a počítá se priorita)
2. prioritní funkce = P(a,r,t,d)
3. výběrové pravidlo (co dělat, když mají shodnou prioritu)

Vysvětlivky:

- a - spotřebovaný čas CPU
- r - čas strávený v systému = spotřebovaný (a) + doba čekání (w)
- t - celkový čas procesoru
- d - perioDa opakování procesu

### Algoritmy

3. náhodný výběr

**+** deterministická odezva (dostanou na řadu jak přišly - žádný nemůže být opomíjen a trčet v systému až nekonečně dlouho)

**−** krátké procesy musí čekat na dříve zahájené dlouhé, delší celková doba zpracování, odezva

2. **LIFO**: Zpracování vždy posledního přišlého procesu
    1. nepreemptivní, v čase příchodu procesu (odblokování)
    2. P(r) = -r (nejkratší dobu v systému)
    3. náhodný výběr

        **−** stárnutí (pokud pořád přibývají nové, ty staré se nemusí nikdy dostat na řadu)
3. **SJF (Shortest Job First)**: Zpracovat vždy nejkratší proces (nutná znalost t)

    Používá se tam, kde opakovaně spouštíme nějaké úlohy a víme, jak dlouho budou trvat.

    1. nepreemptivní, v čase příchodu procesu
    2. P(t) = -t (jak dlouho bude trvat)
    3. náhodný výběr nebo podle r

        **+** kratší celková doba zpracování

        **+** malý počet čekajících procesů

        **-** odhad t (dávkové systémy)

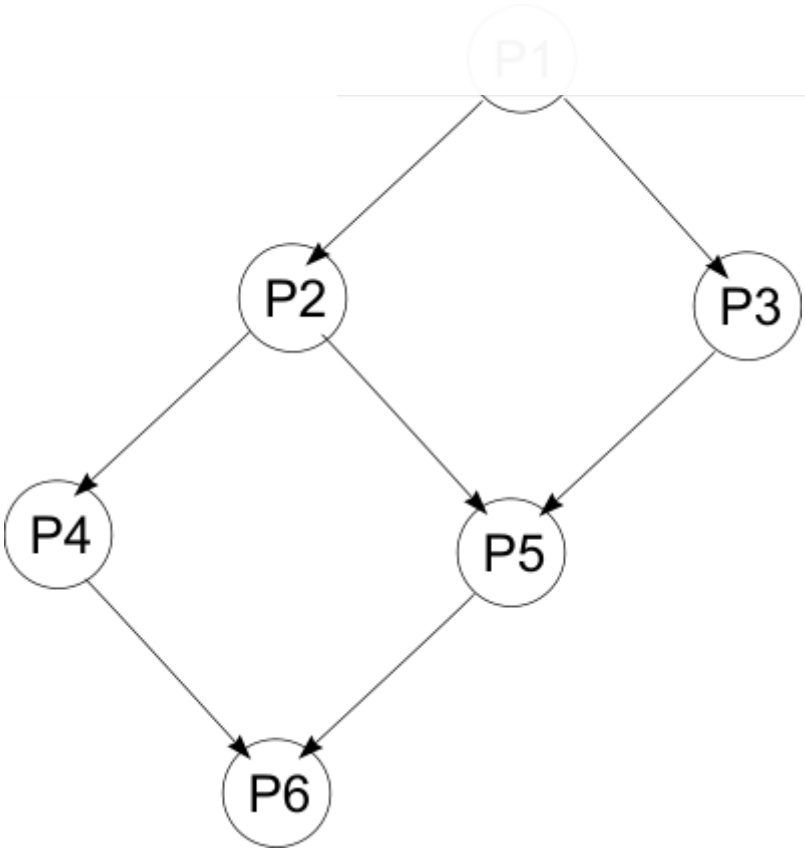
        **-** stárnutí dlouhých procesů při neustálém příchodu krátkých
4. **SRT (Shortest Remaining time)**: Zpracovat vždy nejkratší proces k dokončení (nutná znalost t)
    1. preemptivní, v čase příchodu procesu
    2. P(a, t) = a - t (kolik mu zbývá)
    3. náhodný výběr nebo podle r

        **+** minimální celková doba zpracování

        **-** odhad t (dávkové systémy)
5. Statická priorita: P(i) = konstanta i dle procesu
6. RR (Round Robin): P() = konstanta (vždycky následuje ten následující, např. 1->2, 2->3, 3->1)
7. MLF (MultiLevel Feedback): prasárna
8. RM (Rate Monotonic): P(d) = -d
9. EDF (Earliest Deadline First): P(r,d) = -(d - r%d)

## 17. Zakresli graf pokrytí paralelního systému ({P1, P2, P3, P4, P5, P6}, {(1,2), (1,3), (2,4), (2,5), (3,5), (4,6), (5,6)}*). Lze tento paralelní systém zapsat pomocí P a S? Zapište tento paralelní systém pomocí binárních semaforů (procesy jsou počátečně všechny spuštěny, provedení příkazů P1 – P6 musí být střeženo semafory).

- P1: proces, unlock(S2), unlock(S3)
- P2: lock(S2), proces, unlock(S4), unlock(S5a)
- P3: lock(S3), proces, unlock(S5b)
- P4: lock(S4), proces, unlock(S6a)
- P5: lock(S5a), lock(S5b), proces, unlock(S6b)




## 19. Zapiste algoritmus vzájemného vyloučení při víceprocesorové systémy s využitím atomické instrukce swap(&var1,&var2). Algoritmus nesmí blokovat společnou sběrnici při čekání [editovat]

```
int swap(int *v, int new) //instrukce Swap(mem, reg);
atomic {
        int old = *v;
        (*) v = new;
        return old;
}
```

```
typedef struct QNODE {
   volatile sig_atomic_t lock; /* zámek */
   volatile struct QNODE *next; /* další čekající */
} qnode_t;
/* zámek je reprezentován hlavičkou fronty */
static volatile qnode_t *slock;
/* položka do seznamu, lokální pro každý proces*/
auto qnode_t mynode;
/* zámek položka do seznamu */
void spin_lock(qnode_t **lk, qnode_t *my)
{
   qnode_t *prev;
   my->next = NULL;
   // zakázání přerušení
   if ((prev = swap(lk, my)) != NULL) { // 1
      my->lock = 1;                     // 2
      prev->next = my;                  // 3
      while (my->lock == 1) ;           // 4
   }
}

void spin_unlock(qnode_t **lk, qnode_t *my)
{
   if (my->next == NULL) { /*na konci (lk==my)*/
      if (compare_and_swap(lk, my, NULL)) {
         // povolení přerušení
         return;
      }
      /* vstup je mezi krokem 1. a 3., počkáme až provede krok 3. */
      while (my->next == NULL) ;
   }
   my->next->lock = 0;
}
```

### Alternativní varianta

```
volatile sig_atomic_t lock;
```

```
void spin_unlock(sig_atomic_t *lk)
{
  *lk = 0;
}
```

## 20. Logická a fyzická organizace souborů v systémech souborů

Logická:

- bez struktury - pole slabik (Unix)
  write(fd, buf, len)
- logické záznamy pevné délky
  + rychlý náhodný přístup k záznamům
  + lze aktualizovat záznamy uvnitř souboru
  - využití (záznam s délkou jeden 1B zabírá jako záznam s max. délkou)
- logické záznamy proměnné délky
  - složité vystavení na záznam - sekvenční průchod
  - záznamy lze aktualizovat, pouze pokud se nezmění jejich délka
- index-sekvenční soubory (záznamy s klíčem, B strom)

Fyzická:

- pole alokačních bloků
- alokační blok = nekolik souvislých sektorů (512B) na disku
- alokace:
  - souvislá
  - alok. bloky pevné délky
  - alok.bloky proměnné délky

## 21. Disková hlava je na pozici 80, fronta obsahuje požadavky s adresami 100,180,30,120,10,130,70,50. Uveďte, o kolik se posune hlava pro algoritmy FSDS, SSTF a SCAN (momentální směr pohybu nahoru)

FSDS (FIFO)

- pořadí vystavení: (80), 100, 180, 30, 120, 10, 130, 70, 50

- vystavování o diference: 20 + 80 + 150 + 90 + 110 + 120 + 60 + 20 = 650 cyl.

SSTF

- pořadí vystavení: (80), 70, 50, 30, 10, 100, 120, 130, 180

- vystavování o diference: 10 + 20 + 20 + 20 + 90 + 20 + 10 + 50 = 240 cyl.

SCAN (nahoru)

- pořadí vystavení: (80), 100, 120, 130, 180, 70, 50, 30, 10

## 24. Využití virtuální paměti pro zobrazení diskových souborů, jak se přitom pracuje s obsahem souborů, kdy je vhodné využít tohoto přístupu k souborům

Myšlenka: zamapování souboru do adresového prostoru procesu a pro přístup k němu využít virtuální paměť.

Módy mapování souboru do virtuální paměti:

- **MAP_SHARED** - změny jsou ukládány zpět do souboru, mapovaný soubor vlastně funguje pro danou část LAP jako odkládací prostor (swap).
- **MAP_PRIVATE** – změny se neukládají zpět do souboru (při nahrazování stránek se použije normální swap).

Výhody:

- Rychlejší přístup k datům, může odpadnout kopie přes systémové vyrovnávací paměti (záleží na implementaci V/V).
- Pracovní množina procesu tvoří zároveň vyrovnávací paměť (pokud je dostatek volné paměti, může být celý soubor trvale v paměti).
- Bezprostřední sdílení změn dat ve více procesech (pokud mapují stejný soubor, sdílí stejné fyzické stránky, do kterých jsou zavedeny části souboru).

Kdy je vhodné využít tohoto přístupu k souborům? - Pokud chceme využít některou z výhod asi? 😉

## 25. Organizace adresového prostoru z hlediska zobrazení procesů a jádra systému

- Jeden úsek paměti
  - Monoprogramové systémy bez ochrany paměti
  - LAP = FAP
  - Jádro má pevně přidělenou část paměti, zbytek je dostupný pro procesy
- Společný adresový prostor procesů
  - Multiprogramové systémy
  - LAP(Pi) = LAP(Pj) = FAP
  - Jeden adresový prostor, více programů v paměti na různých adresách
- Oddělený adresový prostor procesů
  - Multiprogramové systémy
  - LAP(Pi) != LAP(Pj) != FAP
  - Každý program má svůj LAP
  - LAP se musí mapovat na FAP

Adresový prostor jádra:

- Oddělený
  - Musí se přepočítávat adresy parametrů volání jádra
- Sdílený s procesem, který volá jádro

## 26. Proč musí být pthread_cond_wait() spojena s mutexem a implementována atomicky? Jaký je rozdíl mezi condition a semaforem z pohledu zasílání událostí

Mutex musí vždy střežit proměnné, které jsou testovány v podmínce cyklu volání pthread_cond_wait(). Jejich testování bez mutexu by bylo nebezpečné. Použití bez opakovaného testování podmínky v cyklu je také špatně - musí bránit případnému samovolnému probuzení.

Pokud čeká nějaké vlákno na cond, je odblokováno (případně všechny právě čekající pro broadcast). Monitor je dále střežen mutexem, takže odblokované vlákno čeká, až bude volný (až se podaří znovu zamknout mutex, který je v pthread_cond_wait()).

Pokud nikdo nečeká, je signalizace prázdnou operací. Pokud čekají všechna vlákna na stejný stav (podmínku) a stačí probudit libovolné z nich, je vhodný pthread_cond_signal(), jinak pthread_cond_broadcast() (probuzení všech najednou).

**Rozdíl mezi semafory a condition:**

- obecný semafor si pamatuje historii - počet down() a up()
- condition ne! c.signal na prázdnou frontu je prázdná operace

Pokud vlákno čeká na condition variable, je blokováno, dokud jiné vlákno nesignalizuje condition variable. Condition variable (na rozdíl od semaforu) není čítač, vlákno na ni musí čekat před tím, než je signalizována. Jinak je signál ztracen a vlákno čeká na další.

Tzn. pokud semafor nastaví up() a odpálí tak událost, je tato událost zpracována až někdo zavolá down(). To může nastat hned nebo kdykoliv budoucnu. Jde tedy o čítač počtu odpálených událostí. Na rozdíl od vláken, kde se žádná "historie událostí" neukládá. Událost odpálím a ti co čekají, ji zpracují. Ti co na ni budou čekat v budoucnu, už neví, jestli v minulosti někdy byla odpálena a musí čekat na její nové odpálení.

---

## 27. Průběh příkazu fork() z hlediska správy paměti a zmenšení jeho režie

*Fork()* vytváří kopii volajícího procesu. To znamená, že se kopíruje i celý adresový prostor původního procesu do prostoru nového procesu. Pokud však dětský proces následné zavolá *exec()*, přepíše se jeho adresový prostor kódem a daty spouštěného programu -> předchozí kopírování bylo zbytečné.

Řešení:

- **vfork() (BSD 4.3)**

  Dětský proces sdílí adresový prostor rodiče; rodič je pozastaven, dokud dětský proces nezavolá *exec()* nebo *exit()*.

  Omezení:

  - Dětský proces nesmí změnit sdílený adresový prostor
  - Nedefinované chavání ve vícevláknových programech

- **Využití stránkování - Copy-On-Write (COW)**

  Oba procesy mohou mít stejně naplněné tabulky stránek. Zakáže se zápis do všech platných stránek. Zápisy do tohoto prostoru vyvolají výpadek stránky, ale zjistí se, že jde o legální zápis do oblasti COW. Vytvoří se kopie sdílené stránky a procesům se povolí zápis do obou kopií.

  Výhody

```
if ((id = fork()) == 0) {  /* dětský proces */
    execl("/prog", ARG0, ARG1, ...);
} ...
```

- sémantika *fork()* – musí vzniknout kopie procesu = kopie obsahu adresového prostoru do vytvářeného procesu,
- pokud následuje *exec()*, přepíše se adresový prostor kódem a daty spuštěného programu, kopie ve *fork()* je zbytečná!

**Řešení:**
**BSD 4.3 -** *vfork()*
Dětský proces používá (sdílí) adresový prostor rodiče, rodič je pozastaven do té doby, než potomek provede *exec()* nebo skončí.

**Omezení:** vzniklý proces nesmí změnit obsah sdíleného adresového prostoru (např. návrat z funkce volající *vfork()* by mohl poškodit obsah zásobníku, volání knihovny std. C změnit statická data v knihovnách, apod.). Není definováno chování ve vícevláknovém programu (obvykle skončí špatně)!

**Využití stránkování** – Copy-On-Write (COW)
- oba procesy mohou sdílet počátečně stejně naplněnou tabulku stránek,
- zakáže se zápis do všech platných stránek,
- první pokus o zápis způsobí výpadek paměti – zjistí se, že se jedná o legální zápis do oblasti COW, vytvoří se kopie stránky a povolí zápis do originálu a kopie v obou procesech.

**Výhody:**
- data se kopírují až v okamžiku, kdy je třeba,
- lze sdílet i celé tabulky stránek (u víceúrovňových).

---

## 28. Optimalizace diskových operací na úrovni V/V, redukce doby provádění

- Disk si sám optimalizuje frontu požadavků nebo
- Optimalizace fronty požadavků na úrovni systému:
  - FIFO
  - SSTF
  - SCAN
  - C-SCAN
  - N-step-SCAN

- FSCAN
- CFQ



**značení volných alokačních bloků na disku**

• lineární seznam volných bloků (UFS) – nelze
• bitová mapa – pro celý disk je příliš velká, nelze udržovat trvale v paměti, ani prohledávat na disku
• skupiny cylindrů, pro každý samostatná bitová mapa a sumární statistika – jedna bitová mapa rozumné velikosti, lze v ní hledat optimální skupinu volných AB (UFS velikost rovna AB*8)
• index-sekvenční organizace – volnou část disku popsat jako soubor se seřazenými volnými úseky podle čísla AB (B-strom)