1 Jádro operačního systému

Operační systém je most mezi HW a (uživatelským) programem, vytváří prostředí pro běh programů. Poskytuje tak abstrakce pro přístup k HW (čti soubor namísto instrukcí `in`).

Operační systém = jádro OS + systémové programy; Dle rozdělení klasifikujeme (mikrojádro, dávkový systém apod.).

Přínosy OS:

1. *Abstrakce* – rozšíření instrukčního repertoáru o vyšší operace
2. *Virtualizace* – sdílení prostředků, izolace, iluze samostatnosti
3. *Správa prostředků* – optimalizace výkonu, spravedlnost, řešení konfliktů

Služby operačního systému:

- Systémové služby
- Komunikace (síť, sériová linka, ...)
- Autentizace a účtování
- Rozhraní pro uživatele

Jádro OS je složitý asynchronní paralelní program, zaveden při startu a celou dobu v paměti, řídí hardware a volá systémové programy.

Typy OS:

1. *Monoprogramové* – aktivní jen jeden proces, není souběžnost, většinou jeden uživatel, MS-DOS
2. *Multiprogramové* – více procesů současně, nutné pro víceuživatelský OS, lepší využití prostředků
   (a) Jednoprocesorové
   (b) Víceprocesorové
      i. Symetrické (rovnocenné procesory)
      ii. Nesymetrické (procesory pro různé účely)

Režim sdílení času (timesharing) je způsob emulace paralelního běhu na jednom procesoru. V jeden okamžik se vykonává jediný proces, ale běží pouze chvíli (timeslice) a pak se řízení předá dalšímu. V delším intervalu je výsledek stejný, jako by běžely paralelně. OS musí umět proces pozastavit a uložit stav.

OS a ochrana proti procesům – OS nesmí být procesy modifikován, je třeba procesy omezit:

- *Privilegované intrukce*
- *Omezen přístup k paměti*
- *Nemožnost provádění I/O operací přímo*
- *Ochrana proti chybám* – nucené odebrání procesoru, je-li třeba

Potřebná podpora v HW:

- *Přerušovací subsystém* – po přerušení se procesor přepne do režimu jádra
- *Dva režimy procesoru* – privilegované instrukce jen v režimu jádra

- *Ochrana paměti* – přístup mimo meze vyvolá přerušení
- *Generátor přerušení* – časovač
- *DMA pro I/O*

Přerušovací systém je potřebný kvůli malé rychlosti I/O zařízení. Jádro musí umět přerušení zpracovat (zablokovat další přerušení, pozastavit běh, obsloužit a znovu spustit):

1. *Klasické jádro* – obsluha přerušení je v jádře, pokud již běželo, je potřeba spinlock (krátkodobý zámek)
2. *Mikrojádro* – obsluhou je proces, stačí jej odblokovat

1. *Návrat do uživatelského režimu* – jednoduchý, navíc může jádro ještě chvíli pracovat na svém
2. *Návrat do jádra* – buď na stejné místo (nepreemptivní jádro) nebo i jinam (preemptivní), ale je to náročné

Přechod mezi režimy (systémový a uživatelský) probíhá přerušením a návratem z obsluhy nebo voláním služby jádra a návratem. Volání jádra je jediný způsob kontaktu mezi uživatelským režimem a jádrem (implementován jako přerušení).

Rozhraní jádra definuje způsob volání jádra – binárně (číslo IRQ, položky zásobníku) nebo zdrojově (volání funkce). Binární je závislé na HW, zdrojové je nyní nezávislé, ale nebylo tomu tak vždy, byla potřeba sjednotit rozhraní – dnes IEEE/ISO POSIX 1003.1

POSIX 1003.1 je standardizované rozhraní jádra operačního systému pro jazyk C. Mělo více revizí, přidávaly se vlastnosti (vlákna).

Správa procesoru se stará o vytváření a rušení procesů, jejich pozastavení, spuštění a přepnutí kontextu, synchronizaci a komunikaci. Proces využívá *virtuální procesor* sám pro sebe.

Stav procesoru uložený při přepnutí zahrnuje všechny registry a virtuální adresový prostor (opět iluze izolovanosti).

Přepnutí kontextu nastává při vyčerpání časového kvanta, volání jádra (zahrnuje čekání na I/O nebo explicitní pozastavení `sleep` a `wait`) nebo má-li běžet proces s vyšší prioritou.

Správa paměti se stará o vytváření virtuální paměti pro procesy a překlad adres na adresu fyzickou, také stránkování nebo segmentaci, swapování a ochranu paměti.

Monolitické jádro (monitor) je jeden program, kde nejen služby OS, ale i *procesy* jsou procedury jednoho programu. Většinou neexistuje rozdíl mezi uživatelským a jaderným režimem.

Klasické jádro obsahuje služby jako podprogramy; rozhraní mezi procesy a jádrem je striktní, obsluha událostí, které nejsou spjaty s procesy je problémová (provádí se před návratem z přerušení jakoby "vloženě").

Mikrojádro má moduly částečně v uživatelském a částečně v jaderném režimu; jádro obsahuje jen přepínání kontextu, přidělování paměti a ochranu, zbytek obsluhují procesy (tedy i služby jádra); komunikace s částmi jádra není volání podprogramu, ale zasílání zpráv → *problém*.

Virtuální stroj je speciálně upravené jádro, které umí pouštět jiný OS jako uživatelský proces. V systémovém režimu běží monitor virtuálního stroje, který odchytává přerušení a zpracovává je již normální proces (hostovaný OS).

2 Procesy a vlákna

Program je statický kód s daty uložený v souboru.

Proces je instance programu v paměti, která se vykonává. Má jednoznačnou identifikaci (PID).

Stav procesu zahrnuje registry, virtuální paměťový prostor, zásobník a systémové prostředky (deskriptory apod.).

Hierarchie – jelikož procesy vznikají duplikací běžícího procesu, 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 syn se stává zombie.

Vlákno je samostatně prováděná část programu v rámci jednoho procesu. Takto může jeden proces běžet na více procesorech paralelně.

Proces s vlákny je pak jen obalová jednotka pro vlákna, která jsou "procesy" z pohledu procesoru. Proces tak pouze uchovává vnější stav (PID, deskriptory apod.).

Rychlost: spuštění vlákna je 10-100x rychlejší než `fork()` (jinak by neměla vlákna moc smysl).

Prostředky: vlákna sdílejí celý adresový prostor (procesy jen kód popř sdílenou paměť), zásobníky jsou sice odděleny, ale jen jiným offsetem (mohou se přepsat). Zásobník je jediné místo, kde vlákno nemusí řešit vícenásobný přístup.

V UNIXu je tedy proces jednotkou přidělování prostředků a vlákno jednotkou přidělování procesoru.

HyperThreading přináší více kontextů na jednom procesoru, ale běh je stále bez paralelizace, procesor se pak tváří jako 2 CPU.

Přidělování procesoru vláknům může být *globální* (každé vlákno je nezávislé) nebo *lokální* (obalový proces rozhoduje o přidělování vláknům).

Pthreads je implementace vláken podle POSIX. Vlákno se spouští voláním funkce, která obsahuje kód vlákna (volající je pak pokračuje dál jako původní vlákno). Volá se pomocí `pthread_create()`, je možné nastavit atributy vlákna. Ukončuje se `pthread_join()` a je identifikováno `pthread_t`.

Atributy pthreads zahrnují lokální/globální plánování, plánovací algoritmus, možnost ukončení bez čekání na stav, velikost zásobníku apod.

Problémy implementace – vlákna sdílejí paměť, starší konstrukce pak kolidují (errno, localtime, strtok), nutnost definovat `_REENTRANT`.

Typické použití: I/O vlákno vedle výpočtů, GUI, u více procesorů nutnost pro výkon, ...

3 Implementace vláken

Možné implementace:

1. *N:N (1:1)* – všechna vlákna jsou na úrovni OS: OS/2, WinNT, LinuxThreads, NPTL
   + \+ volání jádra je přímé a rychlé
   + \+ plné využití multiprocessingu

- jádro musí všechna vlákna evidovat, zabírají jadernou paměť
- při přepnutí vlákna větší režie přepnutí (jde přes jádro)

2. *N:1* – OS vidí jen procesy, vlákna jsou v userspace pomocí knihoven: FreeBSD < 5.x
   + 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)
3. *N:M* – kombinovaný přístup, OS vidí M vláken z N: Solaris, AIX, Irix, FreeBSD
   + OS vidí jen potřebná vlákna (většinou podle počtu CPU)
   + čekající vlákna už OS nevidí, bez režie
   + zároveň lze přepínat kontext i v userspace
   + lze simulovat N:N i N:1
   - velmi problematická implementace

LWP (Lightweight Process) je implementace M:N (Solaris). Jedno vlákno na úrovni procesoru obsahuje staticky několik uživatelských. Plánování je problém, userspace neví moc o tom, co se děje v jádře s LWP. LWP jsou pro jádro normálními procesy (chová se k nim jako ke starým procesům).

KSE (Kernel Scheduler Entities) je nová implementace pomocí konceptu *Scheduler Activations* (FreeBSD > 5.x).

- Procesor se přiděluje skupinám KSE.
- Skupina obsahuje 1 až (počet CPU) aktivních KSE.
- Vlákna, která je potřeba provádět, dostanou přiděleno jedno KSE.
- Pokud je vlákno pozastaveno, probudí se jiné a je to ohlášeno userspace.

4 Základní pojmy pro souběžnost

Nedeterminismus se projevuje v paralelně běžících procesech (a vláknech), při provádění není jasná posloupnost akcí.

Synchronizace je zajištění kooperace pomocí synchronizačních okamžiků (procesy čekají na ostatní).

Race condition chyba vznikající kvůli různé rychlosti paralelních procesů, je potřeba zavést synchronizaci.

Atomická operace je taková, která se provede celá najednou nebo vůbec. Zajišťuje konzistenci dat.

Atomické operace v CPU:

- `read`,`write` – atomické pro slovo (dáno architekturou) zarovnané na hranici přístupu (většinou stránka)
- *Instrukce s jedním operandem v paměti* – opět pozor na zarovnání
- *Speciální instrukce*

Vzájemné vyloučení užívá synchronizaci k tomu, aby danou operaci mohl provádět jediný proces.

Kritická sekce je úsek programu, jehož provádění je vzájemně vyloučené.

- Výpočty před a po KS mohou trvat libovolně dlouho.
- KS musí být provedena v konečném čase (ošetření ukončení procesu).
- Proces může do KS vstoupit nekonečně krát.
- Pokud v KS nikdo není, musí do ní jít vstoupit okamžitě.
- Přidělování procesoru je spravedlivé.

Spravedlnost (fairness) přidělení CPU:

- *Unconditional* – každý nepodmíněný atomický příkaz bude někdy proveden
- *Weak* – také každý podmíněný atomický příkaz bude proveden, pokud nabyde hodnoty TRUE a nebude se měnit
- *Strong* – provedou se i příkazy, jejichž hodnoty podmínek se nekonečně krát změní, pak se také příkaz nekonečně krát provede

Bezpečný (safe) algoritmus zaručuje vzájemné vyloučení.

Živý (live) algoritmus je bez uváznutí (deadlock), blokování (blocking) a stárnutí/vyhladovění (starving)

Uváznutí je stav, kdy všechny procesy čekají na odemčení vstupu do KS, ale proces který by odemkl čeká také. Obtížná detekce, nenastává deterministicky.

Blokování je stav, kdy proces čeká na hodnotu, kterou musí nastavit proces jiný a ten ji nenastaví i když je KS volná. Druhý proces tak blokuje vstup.

Stárnutí je čekání na splnění podmínky vstupu do KS, ale nenastává, jelikož v ní neustále někdo je (např. předbíhání). Může být tolerováno, pokud je vyloučeno praktickým nasazením.

Synchronizační nástroje: zámky, semafory, monitory, zprávy

Synchronizační techniky: čtení/zápis, zákaz přerušení, instrukce (implementují nástroje)

5 Vzájemné vyloučení pro 1 CPU

Atomický kód je úsek kódu, při kterém nemůže dojít k přerušení nebo přepnutí kontextu.

Zakázání přerušení zaručuje, že obsluha přerušení nenaruší KS. Problém je, že komunikace s řadičem trvá moc dlouho, takže vypnutí a zapnutí má vysokou režii. Lze obejít tak, že se zaznamená blokování a pokud IRQ přijde, tak se neobslouží, ale uloží k obsloužení později.

Zakázání přepnutí kontextu v jádře lze pomocí zákazu přerušení, ale tak se zakáže i I/O. Lepší je zakázat preemptivní přepínání kontextu, pak stále funguje přepínání explicitní, ale to není výkonově výhodné pro systém. Je tedy lepší ponechat preempci a používat binární semafory (zámky).

Uživatelské procesy dříve používaly nástroje v jádře (semafory apod.), ale volání jádra má příliš velkou řežii, proto se používá *futex* (Fast Userspace Mutex), který jádro volá pouze, pokud je již zámek uzamčen, jinak se pouze atomicky testuje hodnota.

6 Vzájemné vyloučení pro více CPU

Víceprocesorové prostředí přináší další problémy. Vypnutí přerušení nezabrání běhu procesů na ostatních procesorech.

Krátkodobé vyloučení je speciální pro kritické sekce, které jsou *krátké, neblokující a bez preempce* (nesmí se přerušit její běh). Je možné i *aktivní čekání*, protože přepnutí by mělo vysokou režii. Je potřeba pro implementaci dalších nástrojů.

1. *Implementace čtením a zápisem* – pro více procesorů moc složité
2. *Speciální atomické instrukce* – instrukce nedělitelného čtení a zápisu
   - Test and Set (XCHG, Compare&Swap)
   - Load Linked & Store Conditional

Problémy: instrukce zatěžují paměťovou sběrnici (stále něco testují), u HT se blokuje druhý kontext aktivním čekáním, také možnost stárnutí. Řešením je využití cache a pro HT speciální instrukce

Dlouhodobé vyloučení využívá binární zámek (první sekce zamkne a odemkne při odchodu). Implementace pomocí Test&Set.

Přístup do paměti:

1. *Silná konzistence* – Přístupy v pořadí, v jakém procesory žádaly je pomalé.
2. *TSO* (Total Store Ordering) – Pořadí zachováno jen na jednom procesoru.
3. *PSO* (Partial Store Ordering) – Proházeno i na procesoru.

Bariéra je místo, na kterém čekají procesy, dokud se nesejdou všechny.

7 Binární semafor a mutex

Binární semafor je zámek střežící kritickou sekci.

- Operace `init()`, `lock()` a `unlock()`.
- Nelze číst jeho hodnotu (nemá smysl, mohla by se změnit potom).
- Pasivní čekání ve funkci `lock()`.
- Operace `lock()` a `unlock()` jsou atomické.
- Odemykat může i jiný proces než ten, co zamknul (předávání zámku).
- Silný nebo slabý semafor (má nebo nemá stárnutí).
- Použití jak pro vzájemné vyloučení tak signály (nevhodné).

Mutex je speciální binární semafor pro vzájemné vyloučení, který nelze předávat.

Implementace: `pthread_mutex_t` a funkce `..._init(m,v)`, `..._destroy(m)`, `..._lock(m)`, `..._unlock(m)` a `..._trylock(m)`

8 Obecný semafor

Obecný semafor obsahuje celočíselnou hodnotu, pokud je nula, je zamčen a čeká se na navýšení hodnoty.

- Operace `down()` se čeká na hodnotu větší než 0 a pak ji sníží a může pokračovat.
- Operace `up()` zvýší hodnotu a tím případně odblokuje další proces.
- Používá pasivní čekání ve funkci `down()`.
- Také definováno jako `P()` a `V()` nebo `wait()` a `signal()`.
- Použití pro vzájemné vyloučení (inicializace na 1) nebo signály (udrží i počet neobsloužených).

Implementace v UNIXových systémech není, používá se monitor. Obecně je implementován pomocí spinlocku a testování hodnoty čítače. Pozastavené procesy se přidávají na konec fronty.

Zamykání v jednom procesu (např. `putchar()` v `printf()`) se implementuje jako čítač rekurze. Pokud zámek zamkl stejný proces tak se pouze mění tento čítač a ne zámek.

Simulace: binární semafor lze simulovat dvěma číselnými se sdílenými proměnnými; obecný lze simulovat třemi binárními a sdílenou hodnotou

9 Inverze priority při synchronizaci

Inverze priority nastává v systémech s plánováním s prioritou. Je-li v KS proces s nižší prioritou a proces s vyšší prioritou chce vstoupit, nemůže to udělat díky zamčení KS. Zároveň proces s nižší prioritou nemůže běžet a odemknout, protože proces s prioritou vyšší má právo na procesor.

Dědění priority při vstupu do KS přidělí procesu prioritu nejvyššího procesu, který na KS čeká.

+ Priorita se nemění, pokud je proces osamocen.
- Je potřeba přepočítávat maximální prioritu při každém blokujícím zamykání.

Horní mez priority přidělí procesu v KS prioritu pevnou.

+ Jednoduchá implementace, jediná hodnota.
- Priorita se zvyšuje vždy, i když to není nutné.

Použito pro binární semafor, mutex, monitor; obecný semafor a condition přesně neurčují, kdo blokuje

10 Monitor

Monitor je abstraktní datový typ, který odděluje čekání a operace se sdílenými proměnnými. Semafory jsou příliš obecné a mají složité problémy.

- Sdílené proměnné dostupné pouze operacemi monitoru.
- Operace jednoho monitoru (`up()`,`down()`) jsou vzájemně vyloučeny, atomické.
- Čekání je pasivní, monitor obsahuje frontu *condition*.
- Operace nad frontou `wait()` se monitoru vzdá, `signal()` odblokuje čekající proces.

- Používá se namísto obecných semaforů v UNIXu (vyloučení i signály).

Problém: proces, který odblokoval je ještě v monitoru – buďto se pozastaví a je odblokován až další proces dokončí práci a nebo existuje operace `notify()`.

11 Synchronizační nástroje dle POSIX 1003.1/1996

Pthreads: `pthread_cond_t` je typ pro condition (čekání na situaci), který je *vždy* použit se vzájemným vyloučením typu `pthread_mutex_t` → spolu tvoří *monitor*.

Operace `wait()`: `pthread_cond_wait(condition, mutex)` atomicky uvolní mutex a čeká na splnění podmínky.

Příklad:

```
pthread_mutex_lock(&mutex);
while (suma == 0) { /* podmínka splněna? */
   pthread_cond_wait(&cond, &mutex); /* ne, čekat */
}
/* kritická sekce */
pthread_mutex_unlock(&mutex);
```

Ostatní operace: `pthread_cond_signal(cond)`, `pthread_cond_broadcast(cond)`.

Implementace semaforu je jednoduchá, `init()` inicializuje condition a mutex, `down()` je předchozí příklad, kde v KS je snížení sdílené hodnoty a `up()` zamkne mutex, zvýší hodnotu, signalizuje změnu podmínky a odemkne mutex. Opačný postup (semafor na monitor) je mnohem složitější.

12 Signály v Unixu

Signály jsou ve standardu ISO jazyka C, původem z UNIXu.

Implicitní chování po příjmu signálu je většinou nastaveno na ukončení procesu, kromě signálů SIGKILL a SIGSTOP lze předefinovat.

Zpracování signálu:

1. Signál je odeslán pomocí volání `kill()`.
2. Pokud je proces v jádře a volání bylo přerušitelné, končí chybově (EINTR) a proces přechází zpět do uživatelského režimu.
3. Pokud volání nelze přerušit, signál se pouze zaznamená do doby přechodu.
4. Pokud proces čeká na přidělení procesoru nebo běží, signál se zaznamená do doby přechodu ze systémového do uživatelského režimu (přepnutí kontextu).
5. Pokud při přechodu do uživatelského režimu jsou čekající signály, ošetří se:
   (a) *Implicitně* – většinou ukončení procesu
   (b) *Ignorování* – lze nastavit, některé impicitně
   (c) *Ošetřující funkce* – součást uživatelského programu

Ošetřující funkce ANSI C:

- Tvar `void (*sighandler_t)(int);`, speciální ukazatele `SIG_DFL` a `SIG_IGN`.
- Nastavena funkcí `sighandler_t signal(int signo, sighandler_t func);`.
- Před voláním funkce se nastavení *obnoví* na `SIG_DFL`!
- Funkce pak musí znovu nastavit ošetření – *race condition*.
- Proces se pozastaví a návratová adresa se nastaví na ošetřující funkci.
- Návrat probíhá standardně pomocí `return` nebo `exit()`, `abort()`, `longjmp()`.
- Funkce standardní knohovny *nesmí* být použity (kromě uvedených výše).
- Nelze blokovat příjem signálu po dobu ošetření, takže se může funkce volat paralelně.
- Nejsou reentrantní.
- V ošetření signálu může tento přijít před znovunastavením.
- Čekání na příchod lze pouze přes sdílenou proměnnou (kterou nastaví ošetření signálu) a `sleep()`, což není bezpečné.

Ošetřující funkce POSIX:

- Tvar `void (*sa_handler)(int);`, alternativně `void (*sa_sigaction)(int, siginfo_t*, void*)` (1003.1b).
- Nastvena funkcí `int sigaction(int sig, const struct sigaction* act, struct sigaction* oldact);`.
- Původní nastavení je oznámeno přes `struct sigaction* oldact`.
- Nastavení se nevrací na `SIG_DFL`.
- Po dobu vykonávání je signál blokován a případně uložen do fronty. Je možné blokovat i další signály.
- Je možné blokovat množinu signálů i jindy – hlavně při nastavení ošetření je to nutnost.
- Bezpečné čekání na množinu signálů pomocí `int sigsuspend(const sigset_t* sigset);`.
- Operace nad bitovou množinou signálů, nastavení blokování `sigprocmask()`.

13 Přidělování procesoru

Přidělování procesoru znamená přepnutí kontextu mezi dvěma procesy, vykonává dispatcher.

Plánování (scheduling) je volba strategie a řazení procesů do fronty.

- Minimalizuje odezvu programů (u desktopových systémů menší časová kvanta).
- Efektivní využití prostředků (u serverů delší kvanta a priority).
- Spravedlivé přidělování.
- Zvýšení průchodnosti (transakcí v čase).

Univerzální plánovač je založen na prioritě procesů.

- *Interval rozhodování* určuje interval mezi voláním plánovače. V tomto intervalu se nemůže měnit pořadí procesů.
  1. Nepreemptivní – proces běží do ukončení nebo čekání, jednoduché, malá režie, ale špatná odezva, nepoužitelné pro realtime OS

2. Preemptivní – procesu je odebrán procesor po uplynutí kvanta, příchodu nového procesu nebo odblokování prioritního procesu
3. Selektivní preempce – některé procesy mohou být přerušeny, některé mohou přerušovat, individuální nastavení

- *Prioritní funkce* určuje priortu procesu dle parametrů.
  - Paměťové požadavky,
  - spotřebovaný čas procesoru,
  - doba čekání na procesor,
  - celková doba v systému,
  - externí priorita,
  - zátěž systému, ...
- *Výběrové pravidlo* vybírá z více procesů se stejnou prioritou.
  - Náhodně
  - Cyklicky
  - FIFO

Algoritmus FIFO (FCFS) je nepreemptivní, nechá běžet vždy první proces, který vstoupil.

+ jednoduchý, malá režie přepnutí
+ deterministická odezva
- krátké procesy čekají než starší dlouhý skončí
- dlouhá odezva a doba zpracování

Algoritmus LIFO je nepreemptivní, nechá běžet nejnovější proces.

- ohromné množství stárnutí

Algoritmus SJF (Shortest Job First) je nepreemptivní, nechá běžet proces s nejkratší dobou zpracování.

+ celková doba zpracování je malá
+ fronta čekajících procesů je malá
- musí se odhadovat doba zpracování
- dlouhé procesy stárnou

Algoritmus SRT (Shortest Remaining Time) je preemptivní. nechá běžet proces s nejkratší *zbývající* dobou zpracování.

+ minimální celková doba zpracování
- musí se odhadovat (v praxi nejde) doba zpracování

Algoritmus statické priority je preemptivní (kvanta) i nepreemptivní, typický pro realtime OS, dvě úrovně priority (vyšší je nepreemptivní)

Algoritmus Round Robin je preemptivní (kvanta), cyklicky a stejnoměrně přiděluje procesor.

+ dobrá odezva

+ naprosto spravedlivý
- dlouhá doba zpracování
- problém nastavení kvanta (malé – velká režie; velké – dlouhá odezva)

Algoritmus MLF (Multilevel Feedback) je preemptivní (kvanta), má několik úrovní priorit a na nich pracuje cyklicky.

- Úroveň s vyšší prioritou má kratší kvantum.
- Při vypršení kvanta se procesu priorita sníží (delší kvantum příště).
- Při překročení úrovně buď konec nebo znovu (nic moc řešení).
- Dlouho běžící procesy mají vždy nízkou prioritu (není dobré).
- Základ pro další plánovače (podobný ve FreeBSD 4.4, SVR4).

14 Uváznutí při přidělování prostředků

Přidělování prostředků se zabývá obecnými prostředky (uplatní se na paměť, I/O, systémové prostředky apod.).

Uváznutí je čekání na událost, která nenastane. V tomto kontextu tedy na prostředek, který nebude nikdy přidělen.

Prostředky jsou opakovatelně použitelné (SR) nebo jednorázově použitelné (CR).

Kdy nastává uváznutí:

1. Zamykání zámků, semaforů, souborů apod. – `P1: lock(1);lock(2);  P2: lock(2);lock(1);`
2. Přidělování paměti (nebo bufferů) – pokud je málo paměti a procesy mající půlku paměti chtějí další, uváznou.
3. Zasílání zpráv – všechny procesy čekají na zprávy, které vzájemně pošlou až poté.

Příčiny uváznutí:

- Pouze jeden proces může prostředek používat, ostatní čekají.
- Proces, který má prostředky přiděleny se při neúspěšné alokaci dalších se těchto nevzdá.
- Proces získává prostředky sekvenčně.
- Prostředek nemůže být preemptivně odebrán, musí jej odevzdat proces.

Stav systému udává stav alokace prostředků, je měněn procesy při požadavku, získání a uvolnění prostředku.

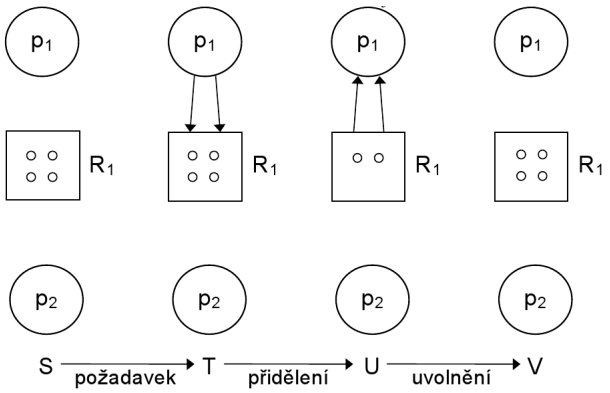
- Proces je blokovaný, pokud nemůže měnit stav systému (čeká na prostředek).
- Proces uváznul, je-li blokovaný a žádné změny stavu jej neodblokují.
- Omezením přechodů mezi stavy tak, aby nedošlo k uváznutí se uváznutí řeší.
- Konstrukce grafu není v běžícím systému možná, jelikož nejsou předem známy požadavky.

15 Uváznutí u SR prostředků

SR prostředky (Serially Reusable):

- Prostředek se skládá z konstantního počtu stejných jednotek
- Jednotka je buďto volná nebo přidělená.
- Proces může uvolnit jednotku, pokud ji má přidělenu.

Graf alokace SR prostředků ukazuje stav systému pro SR prostředky. Prostředek může být přidělen do maximální kapacity a lze žádat maximálně o jeho celou kapacitu.



Detekce uváznutí hledá procesy, které nejsou zablokovány a mohou přivést systém do jiného stavu. Procesy jsou zablokovány, pokud jejich požadavek nemůže být uspokojen ani nemohou uvolnit prostředky.

Redukce grafu nezablokovaným procesem p odstraňuje hrany z/do p přidělením požadovaných prostředků a následně jejich uvolněním.

Neredukovatelný graf nelze žádným procesem redukovat – stav uváznutí.

Plně redukovatelný graf je možné naprosto oprostit od hran. Všechny sekvence ostraňování jsou ekvivalentní.

Algoritmus detekce: postupně prochází graf a redukuje jej s využitím znalosti počtu požadovaných a přidělených prostředků, složitost $O(mn)$.

Systém s okamžitým přidělením přiděluje prostředky ihned, graf tak obsahuje pouze požadavky, pak stačí hledat knot grafu.

Systém s prostředky kapacity 1 uvázne pouze při existenci cyklu grafu.

Systém s jednotkovými požadavky lze také ověřovat existencí knotu.

Zotavení z uváznutí – násilným ukončením procesu nebo odebráním prostředků. Proces se vybere podle ceny (priorita, cena znovuprovedení, typ procesu). Po odebrání se změní stav (graf) a přepočítá se.

Odebrání prostředků:

1. *Přímé* – blokující požadavek na prostředky je ukončen chybou "prostředky odebrány"
2. *Nepříme* – návrat k předchozímu stavu (rollback)

Prevence proti uváznutí omezuje přechody tak, aby se uváznutí vyhnula:

1. Sdílení prostředků (ale to porušuje podmínku výlučnosti).
2. Přidělit prostředky jedinému procesu (ale příliš omezující).
3. Jediný požadavek od procesu, dostane vždy všechny jednotky (neefektivní).
4. Při požadavku další jednotky se nejprve všech vzdát a žádat najednou (ne vždy možné).
5. Požadavek je blokující pouze, pokud žádné jednotky proces nevlastní, jinak neblokující a při neúspěchu se musí vzdát starých, aby mohl zkusit znovu.
6. Prostředky žádat očíslovaně ve vzrůstajícím pořadí, pak nemůže vzniknout blokace (nejpoužívanější).

Bankéřův algoritmus omezuje maximální počet požadavků

16 Organizace paměti

Jeden úsek paměti – LAP=FAP, OS má přidělenu část, zbytek jednomu programu (MS-DOS), není ochrana, monoprogramování, posléze zavedeny segmenty

Společný adresový prostor – všechny programy mají stejný prostor mapovaný na fyzickou paměť, pouze jsou zavedeny na jiná místa. Bez ochrany, programy se mohou přepisovat, je potřeba relokace, ale jednoduchá správa a přidělování paměti.

Ochrana paměti při multiprogramování ve společných LAP – mezní registr a chráněný režim OS (programy mají ale omezený LAP).

Oddělené adresové prostory – každý program má k dispozici celý LAP, ochrana oddělením při mapování, ale mapování složitější (podpora hardware).

Adresový prostor jádra je oddělený LAP (volání jádra musí přepočítat adresy), ale je také sdílený s volaným programem (chráněný).

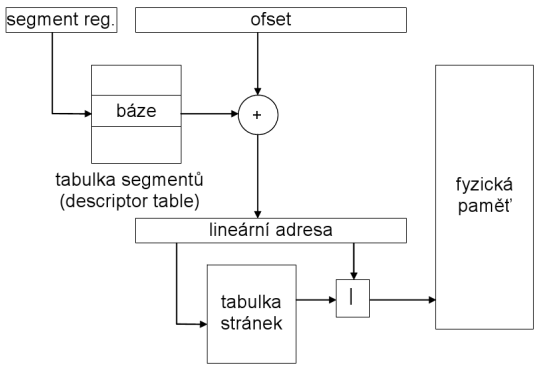
Úseky pevné velikosti dělí FAP na pevné úseky, kam se programy musejí vlézt. Fronty čekajících procesů na úsek, interní fragmentace.

Úseky proměnné velikosti mění velikost dynamicky (spojování sousedních volných nebo dělení při zavedení programu), externí fragmentace.

Segmentace přiděluje každému procesu úseky proměnné velikosti, které mají adresu báze a limit. Tabulka segmentů může být globální nebo pro každý proces zvlášť. Umožňuje různou ochranu segmentů, ale stále externí fragmentace.

Stránkování dělí FAP na *rámce* a LAP na *stránky* stejné velikosti. Tabulka stránek pak určuje mapování, logická adresa určuje číslo stránky. Je zamezeno externí fragmentaci (v LAP), ochrana na úrovni stránek.

Segmentace se stránkováním – segmenty jsou stránkovány



Segmentace a stránkování Intel >= 80386

17 Stránkování

Tabulka stránkek obsahuje mj. číslo rámce, flagy zápisu, systémové stránky, označení *dirty* a přítomnosti v paměti.

Virtualizace – ne všechny stránky jsou v paměti, pokud se použije nepřítomná stránka, jedná se o *výpadek stránky*.

Výpadek stránky vyvolá přerušení a systém musí tabulku načíst z disku (swap), poté se instrukce, která výpadek vyvolala restartuje.

Nahrazovací algoritmus určuje, která stránka se má z paměti ostranit, aby bylo místo pro žádanou stránku.

Organizace tabulky stránek:

1. *Rychlost* – překlad při každé instrukci → podpora hardware (TLB)
2. *Velikost* – pro 32b adresy a 4kB stránky má tabulka 1M položek → úrovně

Translation Look-Aside Buffer (TLB) je vyrovnávací paměť v HW pro překlad adres, poslední adresy jsou uloženy v rychlé (SRAM) asociativní paměti. Přístup je zredukován z 50ns na 5ns. Čím větší úspěšnost TLB, tím rychlejší průměrný přístup. Při přepnutí kontextu se TLB vyprázdní (SPARC ne, přidává položkám i číslo procesu).

Víceúrovňová organizace stránek – adresa obsahuje indexy tabulek, položky tabulek pak vybírají bázi následující tabulky. Intel – Page Directory a Page Table, AMD64 má 4 úrovně.

Inverzní tabulka stránek indexuje podle rámců a obsahuje, komu je rámec přidělen a číslo stránky, kterou proces vidí. Mnohem jednodušší tabulka a malá režie, ale hledání je komplikované a sdílení paměti ještě více.

Stránkovací algoritmus udává, kdy a kolik stránek se zavede, které rámce se jim přiřadí a případně, které stránky mají být odstraněny. Snaha minimalizovat výpadky (nahrazením nepoužívaných apod.).

Výběr zaváděných stránek je možné provádět prefetchingem (dopředu), používá se zavádění na žádost (při výpadku).

Počet zaváděných stránek: celý LAP (neefektivní), jediná stránka (neefektivní), stránky a okolí (předvídá se přístup na sousední stránky).

Umísťování stránek nemá vliv na výpadky, ale na rychlost odkládání (souvislý kus se odloží rychleji).

Swapping je umisťování vyřazených stránek na disk do swapu (swap se nemusí použít, ale pak nelze stránky vyřadit a dojde paměť).

Thrashing je stav, kdy počet výpadků překračuje přípustnou mez, většinu výkonu spotřebuje režie stránek (nejen algoritmus, ale i I/O swapu).

18 Nahrazovací algoritmy

Nahrazovací algoritmus určuje, které stránky vyřadit z paměti, snaha o minimalizaci následných výpadků. Problém je, že nezná následující sled stránek, takže pouze odhaduje z minulosti.

Klasifikace:

1. *Lokální* – vyřazuje pouze stránky procesu, který nyní chce stránku zavést
2. *Globální* – nerozlišuje stránky podle procesu

1. *Pevný počet rámců* – počet stránek v paměti je stejný
2. *Proměnný počet rámců* – počet stránek v paměti se mění

Princip lokality

- *Prostorová lokalita* – při běhu programu, nejsou najednou potřeba všechny stránky LAP, ale je vysoká pravděpodobnost, že následující adresa bude směřovat do stejné nebo sousední stránky (sekvenční kód, struktury a pole, proměnné jsou u sebe).
- *Časová lokalita* – při běhu programu se některé proměnné používají opakovaně (cykly, funkční proměnné)

Neplatí-li princip lokality (optimalizace kódu a dat), sebelepší používaný algoritmus nebude fungovat.

19 Nahrazovací algoritmy s pevným počtem rámců

Optimální algoritmus:

- Vybírá stránku, která bude nejdéle nepoužívána.
- Vyžaduje znalost budoucnosti (není reálný).
- Minimální možný počet výpadků stránek.

LRU (Least Recently Used):

- Vybírá stránku, která byla nejdéle nepoužita.
- Aproximuje optimální algoritmus (princip lokality).
- Implementován jako zásobník, při použití přesun nahoru, vyhazuje se nejspodnější.

NRU (Not Recently Used):

- Aproximuje LRU jedním bitem.
- Při zavedení a použití je bit nastaven.
- Bit je nulován po uplynutí intervalu.

LFU (Least Frequently Used):

- Odstraněna nejméně používaná stránka.
- Může ale nahradit i právě použitou stránku.
- V případě rovnosti použije LRU.

FIFO:

- Odstraní stránku, která byla nejdéle v paměti.
- Mohla tam být proto, že je neustále používána.

LIFO

- Odstraní stránku, která je v paměni nejkratší dobu.
- Dojde do stavu, kdy se pouze vyhazuje poslední stránka a ostatní zůstávají.

Clock:

- Cyklický seznam stránek s jednobitovým příznakem.
- Ukazatel na aktuální pozici.
- Při hledání se testuje bit.
- Je-li nastaven, vynuluje se a posune ukazatel.
- Pokud není nastaven, našla se stránka (a posune se ukazatel).

Second Chance NRU:

- Založen na Clock, ale má kromě příznaku použití i příznak modifikace.
- Není-li bit užití nastaven a je nastaven zápisový, stránka se pouze zapíše na disk, bit se vynuluje a posune se ukazatel.

20 Nahrazovací algoritmy s proměnným počtem rámců

VMIN:

- Optimální nerealizovatelný algoritmus.
- Udržuje pouze stránky, které budou potřeba (znalost budoucnosti).

WS (Working Set):

- V paměti je pouze pracovní množina stránek, tedy stránky, které byly použity v daném intervalu.
- Složitá implementace, při každém odkazu stránky (nejen výpadku) se musí aktualizovat.

Page Fault Frequency:

- Volí velikost pracovní množiny podle frekvence výpadků.
- Při vysoké frekvenci výpadků je možné přidat další prvek množiny.
- Při nízké frekvenci se odeberou všechny stránky nepoužité od posledního výpadku.

21 Stránkování v praxi

Paměť jádra je potřebná okamžitě (např. při I/O), nemůže se čekat na nahrazení.

Nahrazování probíhá už dríve než nastanou výpadky (většinou 75%). Také se udržuje seznam aktivních a neaktivních stránek.

Zamykání stránkek je nutné pro DMA a realtime OS. Privilegované funkce `mlock()` a `munlock()`.

Spouštění procesu voláním `fork()` vytváří *kopii* procesu a jeho LAP, prostředků apod. Většinou ale následuje volání `exec()`, které LAP přepíše načteným kódem programu – zbytečná kopie.

Volání `vfork()`: pozastraví rodiče do volání `exec()` a sdílí jeho LAP. Nebezpečné, pokud se špatně použije.

Copy on Write je kopie stránky taková, že ukazuje na stejné místo v paměti, ale pokud dojde k zápisu, vytvoří se kopie na místě jiném a teprve tam se zapíše. Umožňuje sdílet celé tabulky stránek (efektivita).

Sdílení stránek:

- *Sdílení kódu* – stránky jen pro čtení, instance programů sdílejí, ušetří mnoho paměti
- *Pozičně nezávislý kód* – adresy jsou relativní podle GOF (Global Offset Table), využívá EBX jako bázovou adresu, Wndows nemají, více instancí na různých relokovalných adresách.
- *Explicitní sdílení* – na požádání procesu, využito pro komunikaci, různá rozhraní

SVR3: sdílená paměť označená klíčem `shmget()`, `shmat()`, omezená velikost.

BSD 4.4: mapování souborů do LAP, ale sdílení pouze s `fork()` soubory, `mmap()`.

POSIX 1003.1b: mapování přes soubory, ale lze vytvořit sdílenou paměť klíčem pomocí `shm_open()`.

Virtualizace souborů – použitím `mmap()` lze soubor namapovat jako součást LAP, pak se chová jako swapovací místo dané části paměti. Ušetří se systémová volání `read()` a `write()`. Výhodné pro náhodný přístup a přístup po malých částech, ale zabírá více paměti. Lze mapovat jen pro čtení (`MAP_PRIVATE`) a pak se swapuje do swapu.

22 Systémy souborů

Soubor je abstraktní jednotka obsahující data, která má jednoznačnou identifikaci.

Fyzická abstrakce souboru: I/O po sektorech disku, adresa fyzického sektoru, bez práv a ochran

Logická abstrakce souboru: I/O po bajtech (nebo záznamech), identifikace souboru, práva souboru

Systém souborů poskytuje: strukturu souborů, alokaci diskového prostoru, souborové operace, ochranu

Logická struktura souboru je na úrovní jádra OS

1. *Bez struktury* – pouhé pole bajtů (Unix)
2. *Logické záznamy s pevnou délkou* – rychlý náhodný přístup, lehké změny, nevyužit prostor
3. *Logické záznamy s proměnnou délkou* – pomalý přístup, špatná aktualizace, plné využití
4. *Indexsekvenční soubory* – záznamy proměnné délky a index podle klíče

Vnitřní struktura souboru je na úrovni I/O

- Soubor je pole alokačních bloků.
- Alokační blok je několik sektorů.
- Je potřeba překlad z logického adresování na adresy bloků.
- Velikost alokačního bloku ovlivňuje rychlost a využití místa.

Souvisle alokované soubory:

+ rychlý přístup k celému souboru i při náhodném přístupu
- nutná prealokace místa
- při alokaci po úsecích velká interní fragmentace
- problém vkládání dat jinam než na konec

Soubory jako lineární seznamy:

+ jednoduché vkládání, přidávání, rušení dat
- náhodný přístup příliš pomalý, nutné procházení
- obrovská externí fragmentace

Indexsekvenční uložení souboru:

- Index obsahuje seznam alokačních bloků a jejich offset.
- Tabulka může být velká...
- Globální tabulka (FAT)
- Víceúrovňová tabulka (UFS) je rychlejší, ale nevýhodná pro malé soubory.
- Pro malé soubory se index nepoužije (složitější implementace).
- S proměnnou alokační jednotkou – XFS, JFS

Alokace diskového prostoru v OS musí nejen volit umístění a velikost jednotek, ale také mít přehled o volném místě a rozložení suborů.

- *Volné místo* – souvislá alokace nebo alokace pevných bloků eviduje volné bloky; alokace proměnné délky musí evidovat volné úseky a spojovat je nebo dělit
- *Okamžik alokace* – prealokace omezuje fragmentaci souboru, ale nešetří místo; dynamická alokace při zápisu může vyvolat konflikt s jiným souborem; odložená alokace čeká až na uložení stránky paměti
- *Velikost jednotky* – 512B využije prostor na 95%, ale větší režie

Organizace souborů bývá hierarchická, je nutný (víceúrovňový) adresář souborů (mapuje jméno na identifikaci). Vyhledávání podle seznamu, stromu nebo hash tabulky.

Operace se soubory:

- *Zpřístupnění* – vyhledání souboru, čtení metadat
- *Čtení a zápis* – na základě identifikace
- *Manipulace* – změny metadat

Umístění metadat může být přímo v adresáři nebo jak zvláštní datová jednotka (i-node).

Odolnost FS proti výpadku je vynucena existencí více míst, kde je nutná změna (smazání souboru musí změnit adresář, seznam volných bloků a seznam i-node).

- *Synchronní změny* – v pořadí, při výpadku lze detekovat a opravit, hodně pomalé
- *Soft metadata update* – pokud I/O vykoná něco asynchronně, změní se ostatní data tak, aby to reflektovala
- *Transakční zpracování* – vyžaduje žurnálování