B4B35OSY: Operační systémy

Lekce 4. Plování a synchronizace

Petr Štěpán
stepan@fel.cvut.cz

10. září, 2020

Outline

Obsah

Druhy plánovačů

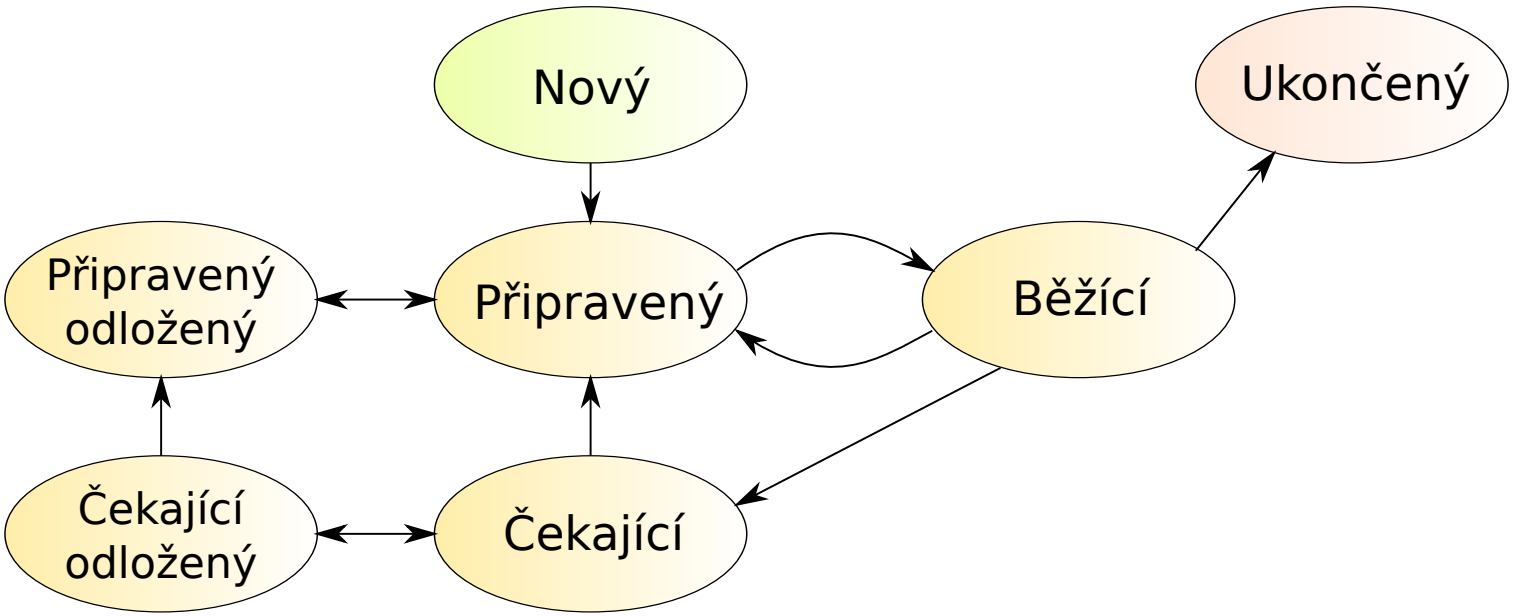
- Krátkodobý plánovač (operační plánovač, dispečer):
  - Základní správa procesoru/ů
  - Vybírá proces, který poběží na uvolněném procesoru přiděluje procesu procesor (CPU)
  - vyvoláván velmi často, musí být extrémně rychlý
- Střednědobý plánovač (taktický plánovač)
  - Úzce spolupracuje se správou hlavní paměti
  - Taktika využívání omezené kapacity fyzické paměti při multitaskingu
  - Vybírá, který proces je možno zařadit mezi odložené procesy
  - uvolní tím prostor zabíraný procesem v fyzické paměti
  - Vybírá, kterému odloženému procesu lze znovu přidělit prostor v paměti počítače
- Dlouhodobý plánovač (strategický plánovač, job scheduler)
  - Vybírá, který požadavek na výpočet lze zařadit mezi procesy, a definuje tak stupeň multiprogramování
  - Je volán zřídka (jednotky až desítky sekund), nemusí být rychlý
  - V interaktivních systémech se používá velmi omezeně, např. plánování aktualizací

Stavy procesu

Nové stavy spojené s odkládáním procesu na disk při nedostatku fyzické paměti:

- Odložený připravený
- Odložený čekající

Moderní OS většinou neprovádí odkládání celých procesů, ale při nedostatku paměti pak hrozí thrashing (podrobněji probereme při stránkování).



Dispečer

- Dispečer pracuje s procesy, které jsou v hlavní paměti a jsou schopné běhu, tj. připravené (ready)
- Existují 2 typy plánování
  - nepreemptivní plánování (kooperativní plánování, někdy také plánování bez předbíhání)
    - běžícímu procesu nelze „násilně“ odejmout CPU, proces se musí procesoru vzdát, nebo ho nabídnout
    - historické operační systémy, kdy nebyla od systému podpora preempce
    - nyní se používá zpravidla jen v „uzavřených systémech“, kde jsou předem známy všechny procesy a jejich vlastnosti. Navíc musí být naprogramovány tak, aby samy uvolňovaly procesor ve prospěch procesů ostatních
  - preemptivní plánování (plánování s předbíháním),
- procesu schopnému dalšího běhu může být procesor odňat i „bez jeho souhlasu“, tedy kdykoliv
- plánovač rozhoduje v okamžiku:
  1. kdy některý proces přechází ze stavu běžící do stavu čekající nebo končí
  2. kdy některý proces přechází ze stavu čekající do stavu připravený
  3. přijde vnější podnět od HW prostřednictvím přerušení, nejčastěji od časovače
- První případ se vyskytuje v obou typech plánování
- Další dva jsou použity pouze pro plánování preemptivní

Kritéria plánování

Kritéria plánování

- Uživatelsky orientovaná
  - čas odezvy
    - doba od vzniku požadavku do reakce na něj
  - doba obrátky
    - doba od vzniku procesu do jeho dokončení
  - konečná lhůta (deadline)
    - požadavek dodržení stanoveného času dokončení
  - předvídatelnost
    - Úloha by měla být dokončena za zhruba stejnou dobu bez ohledu na celkovou zátěž systému
    - Je-li systém vytížen, prodloužení odezvy by mělo být rovnoměrně rozděleno mezi procesy
- Systémově orientovaná
  - průchodnost
    - počet procesů dokončených za jednotku času
  - využití procesoru
    - relativní čas procesoru věnovaný aplikačním procesům
  - spravedlivost
    - každý proces by měl dostat svůj čas (ne „hladovění“ či „stárnutí“)
  - vyvažování zátěže systémových prostředků
    - systémové prostředky (periferie, hlavní paměť) by měly být zatěžovány v čase rovnoměrně

Základní plánovače

Ukážeme plánování:

- FCFS (First-Come First-Served)
- SPN (SJF) (Shortest Process Next)
- SRT (Shortest Remaining Time)
- cyklické (Round-Robin)
- zpětnovazební (Feedback)

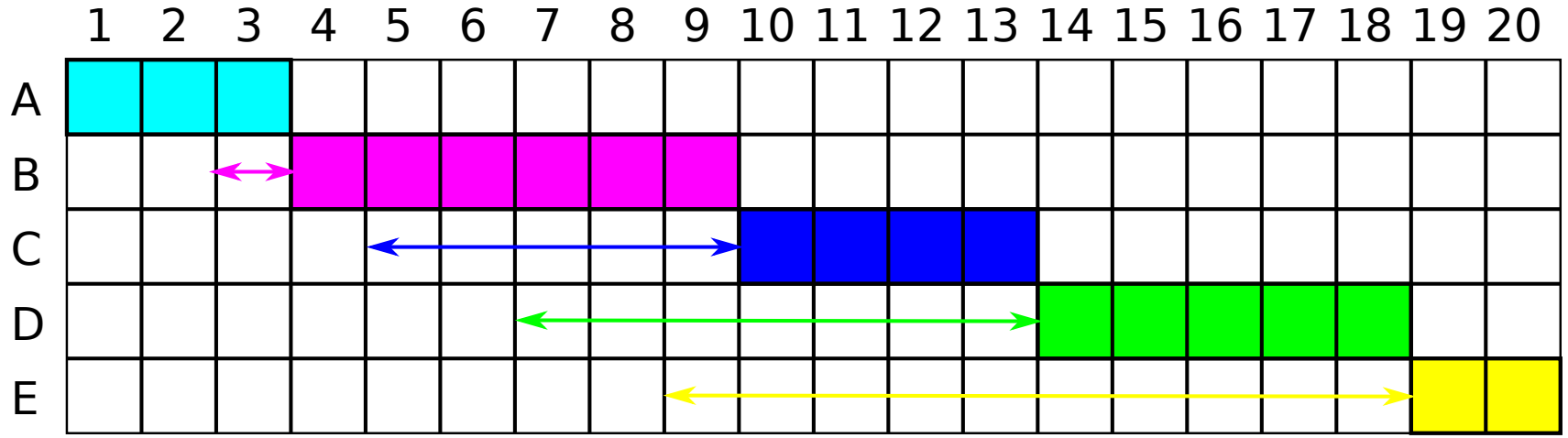
Příklad pro ilustraci algoritmů:

Proces	Čas příchodu	Potřebný čas
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

FCFS

- FCFS = First Come First Served – prostá fronta FIFO
- Nejjednodušší nepreemptivní plánování
- Nově příchozí proces se zařadí na konec fronty
- Průměrné čekání může být velmi dlouhé

Příklad:



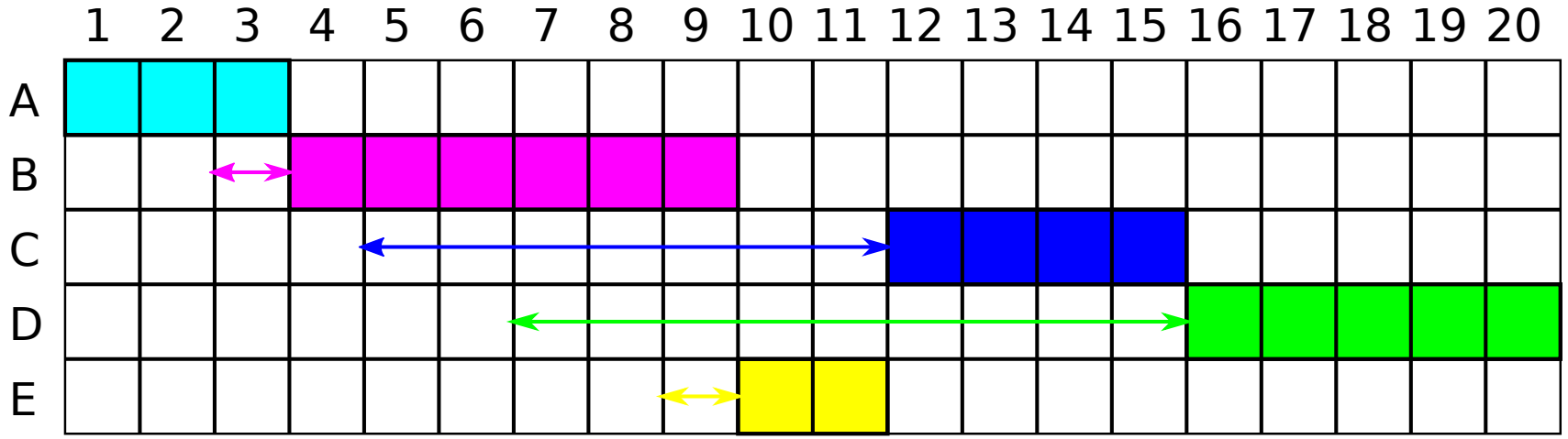
- Průměrné čekání $T_{Avg} = \frac{0+1+5+7+10}{5} = 4.6$
- Průměrné čekání bychom mohli zredukovat, pokud by proces E běžel hned po B.

FCFS – vlastnosti

- FCFS je primitivní nepreemptivní plánovací postup
- Průměrná doba čekání T_{Avg} silně závisí na pořadí přicházejících dávek
- Krátké procesy, které se připravily po dlouhém procesu, vytváří tzv. konvojový efekt
  - Všechny procesy čekají, až skončí dlouhý proces
- Pro krátkodobé plánování se FCFS samostatně fakticky nepoužívá.
  - Používá se pouze jako složka složitějších plánovacích postupů

SPN

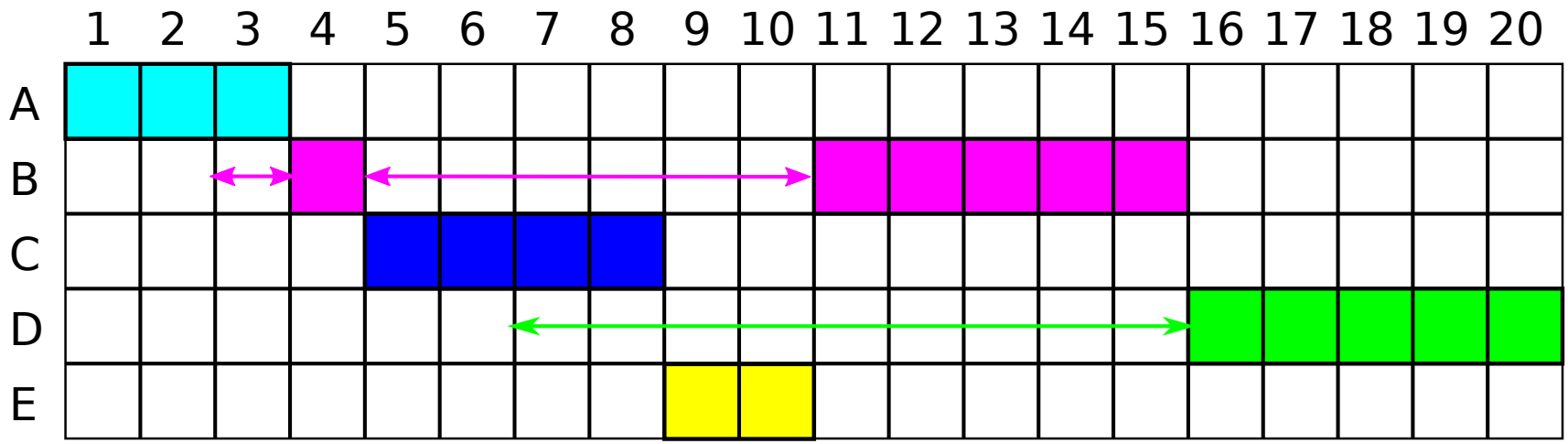
- SPN = Shortest Process Next (nejkratší proces jako příští); též nazýváno SJF = Shortest Job First
  - Opět nepreemptivní
  - Vybírá se připravený proces s nejkratší příští dávkou CPU
  - Krátké procesy předbíhají delší, nebezpečí stárnutí dlouhých procesů
  - Je-li kritériem kvality plánování průměrná doba čekání, je SPN optimálním algoritmem, což se dá exaktně dokázat



- Průměrné čekání $T_{Avg} = \frac{0+1+7+9+1}{5} = 3.6$

SRT

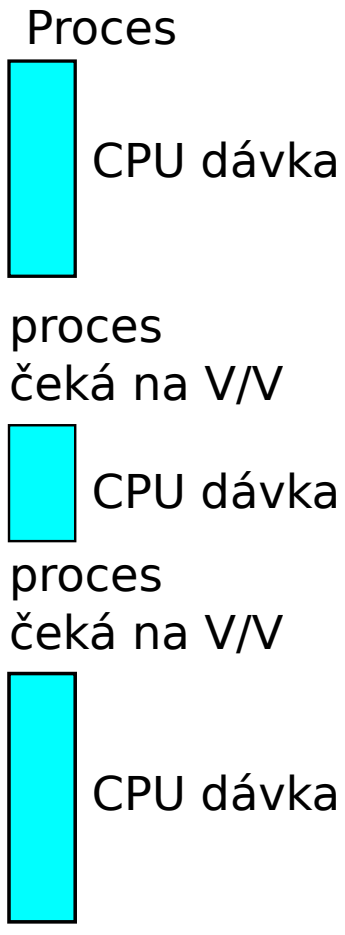
- SRT = Shortest Remaining Time (nejkratší zbývající čas)
- Preemptivní varianta SPN
- CPU dostane proces, který potřebuje nejméně času do svého ukončení
- Jestliže existuje proces, kterému zbývá k jeho dokončení čas kratší, než je čas zbývající do skončení procesu běžícího, dojde k preempci
- Může existovat více procesů se stejným zbývajícím časem, a pak je nutno použít „arbitrážní pravidlo“, např. vybrat první z fronty



- Průměrné čekání $T_{Avg} = \frac{0+7+0+9+0}{5} = 3.2$

Jak nejlépe využít procesor

- Maximálního využití CPU se dosáhne uplatněním multiprogramování
- Jak ?
- Běh procesu = cykly alternujících dávek
  - CPU dávka
  - I/O dávka
- CPU dávka se může v čase překrývat s I/O dávkami dalších procesů



Odhad délky běhu

- Délka příští dávky CPU skutečného procesu je známa jen ve velmi speciálních případech
  - Délka dávky se odhaduje na základě nedávné historie procesu
  - Nejčastěji se používá tzv. exponenciální průměrování
- Exponenciální průměrování
  - t_n skutečná změřená délka n-té dávky CPU
  - τ_{n+1} odhad délky příští dávky CPU
  - α, $0 \leq \alpha \leq 1$ parametr vlivu historie
  - $\tau_{n+1} = \alpha \cdot t_n + (1\alpha)\tau_n$
  - Příklad:
    - $\alpha = 0.5$
    - $\tau_{n+1} = 0.5 \cdot t_n + 0.5 \cdot \tau_n = 0.5 \cdot (t_n + \tau_n)$
    - τ_0 se volí jako průměrná délka CPU dávky v systému nebo se odvodí z typu nejčastějších programů

Prioritní plánování

- Každému procesu je přiřazeno prioritní číslo
  - Prioritní číslo – preference procesu při výběru procesu, kterému má být přiřazena CPU
  - CPU se přiděluje procesu s nejvyšší prioritou
  - Nejvyšší prioritě obvykle odpovídá (obvykle) nejnižší prioritní číslo
    - Ve Windows je to obráceně
- Existují opět dvě varianty:
  - Nepreemptivní
    - Jakmile se vybranému procesu procesor předá, procesor mu nebude odňat, dokud se jeho CPU dávka nedokončí
  - Preemptivní
    - Jakmile se ve frontě připravených objeví proces s prioritou vyšší, než je priorita právě běžícího procesu, nový proces předběhne právě běžící proces a odejme mu procesor
- SPN i SRT jsou vlastně případy prioritního plánování
  - Prioritou je predikovaná délka příští CPU dávky
  - SPN je nepreemptivní prioritní plánování
  - SRT je preemptivní prioritní plánování

Prioritní plánování – problémy

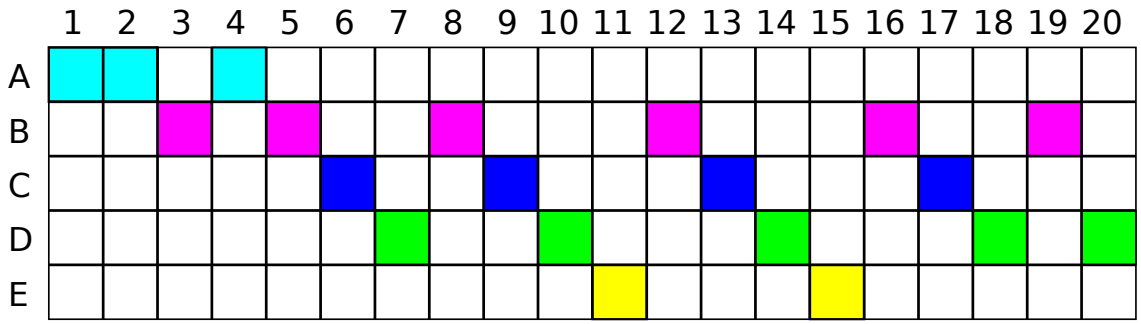
- Problém stárnutí (starvation):
  - Procesy s nízkou prioritou nikdy nepoběží; nikdy na ně nepřijde řada
    - Údajně: Když po řadě let vypínali v roce 1973 na M.I.T. svůj IBM 7094 (jeden z největších strojů své doby), našli proces s nízkou prioritou, který čekal od roku 1967.
- Řešení problému stárnutí: zrání procesů (aging)
  - Je nutno dovolit, aby se procesu zvyšovala priorita na základě jeho historie a doby setrvávání ve frontě připravených
    - Během čekání na procesor se priorita procesu zvyšuje

Cyklické plánování

- Cyklická obsluha (Round-robin) – RR
- Z principu preemptivní plánování
- Každý proces dostává CPU periodicky na malý časový úsek, tzv. časové kvantum, délky **q** (desítky ms)
- V „čistém“ RR se uvažuje shodná priorita všech procesů
- Po vyčerpání kvanta je běžícímu procesu odňato CPU ve prospěch nejstaršího procesu ve frontě připravených a dosud běžící proces se zařazuje na konec této fronty
- Je-li ve frontě připravených procesů n procesů, pak každý proces získává $\frac{1}{n}$ doby CPU
- Žádný proces nedostane 2 kvanta za sebou (samozřejmě pokud není jediný připravený)
- Žádný proces nečeká na začátek přidělení CPU déle než $q(n1)$

Cyklické plánování

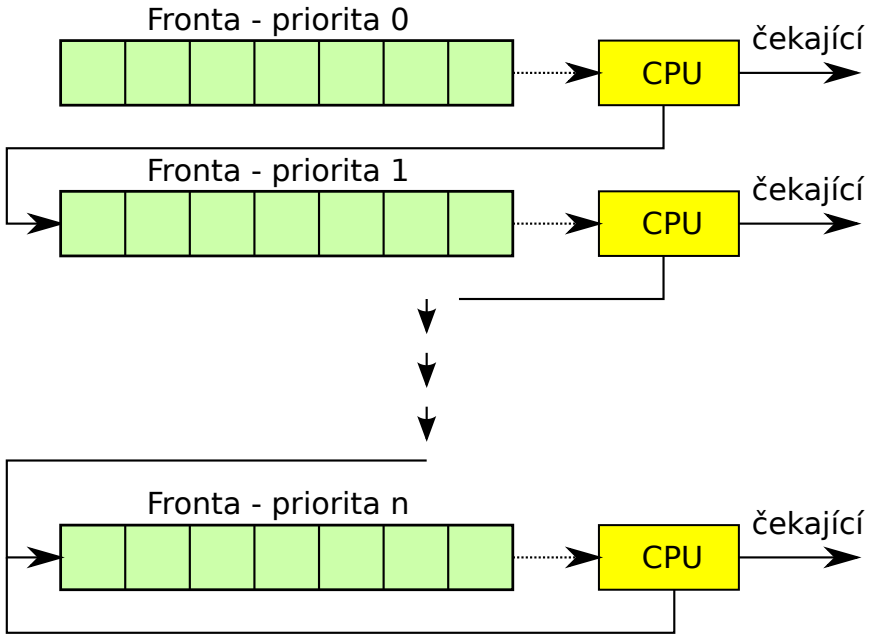
- Efektivita silně závisí na velikosti kvanta
- Veliké kvantum – blíží se chování FCFS
  - Procesy dokončí svoji CPU dávku dříve, než jim vyprší kvantum.
- Malé kvantum – časté přepínání kontextu
  - značná režie
- Dosahuje se průměrné doby obrátky delší oproti plánování SRT
  - Průměrná doba obrátky se může zlepšit, pokud většina procesů se době **q** ukončí
  - Empirické pravidlo pro stanovení **q**: cca 80% procesů by nemělo vyčerpat kvantum
- Výrazně lepší je čas odezvy



Zpětnovazební plánování

- Základní problém:
  - Neznáme předem časy, které budou procesy potřebovat
- Východisko:
  - Penalizace procesů, které běžely dlouho
- Řešení:
  - Dojde-li k preempci přečerpáním časového kvanta, procesu se snižuje priorita
  - Implementace pomocí víceúrovňových front
    - pro každou prioritu jedna
    - Nad každou frontou samostatně běží algoritmus určitého typu plánování, obvykle RR s různými kvanty a FCFS pro frontu s nejnižší prioritou

Víceúrovňové zpětnovazební fronty



- Proces opouštějící procesor kvůli vyčerpání časového kvanta je přeřazen do fronty s nižší prioritou
- Fronty s nižší prioritou mohou mít delší kvanta
- Problém stárnutí ve frontě s nejnižší prioritou
  - Řeší se pomocí zrání (aging) – v jistých časových intervalech (≈ 10 s) se zvyšuje procesům priorita přemístěním do „vyšších“ front

O(1) plánovač – Linux 2.6.22

- O(1) – rychlost plánovače nezávisí na počtu běžících procesů – je rychlý a deterministický
- Dvě sady víceúrovňových front
  - Na začátku první sada obsahuje připravené procesy, druhá je prázdná
  - Při vyčerpání časového kvanta je proces přeřazen do druhé sady front do nové úrovně
  - Vzbuzené procesy jsou zařazovány podle toho, zda ještě nevyužily celé svoje časové kvantum do aktivní sady front, nebo do druhé sady front
  - Pokud je první sada prázdná, dojde k prohození první a druhé sady front procesů
- Heuristika pro odhad interaktivních procesů a jejich udržování na nejvyšších prioritách s odpovídajícími časovými kvanty

Zcela férový plánovač

- Linux od verze 2.6.23 (Completely Fair Scheduler)
- Nepoužívá fronty, ale jednu strukturu, která udržuje všechny procesy uspořádané podle délky již spotřebovaného času a délky čekání
  - kritérium = spotřebovaný_čas - férový_čekací_čas
  - férový_čekací_čas je reálný čas dělený počtem čekajících procesů na jeden procesor
  - ideálně všechny procesy mají kritérium 0
- Pro rychlou implementaci se používá vyvážený binární červeno-černý strom, zaručující složitost úměrnou log(n) počtu připravených procesů
- Nepotřebuje složité heuristiky pro detekci interaktivních procesů
- Jediný parametr je časové kvantum:
  - pro uživatelské PC se volí menší
  - pro serverové počítače větší kvanta omezují režii s přepínáním procesů a tím zvyšuje propustnost serveru
- Žádný proces nemůže zestárnout, všechny procesy mají stejné podmínky

Plánování v multiprocesorech

Přiřazování procesů (vláken) procesorům:

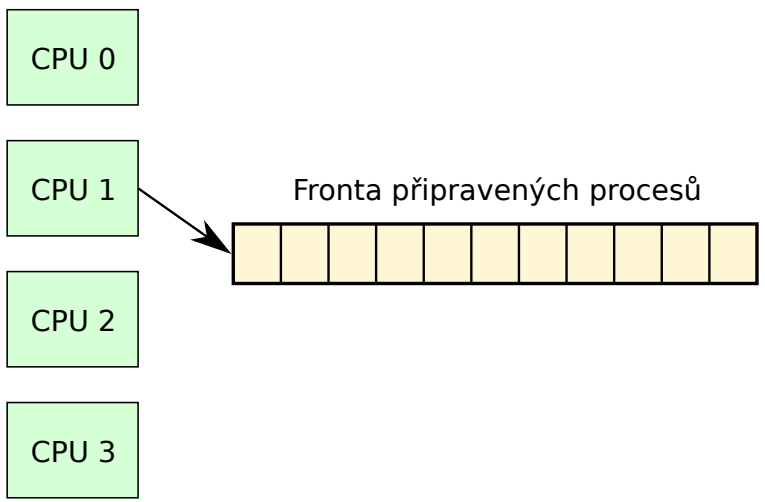
- Architektura „master/slave“
  - Klíčové funkce jádra běží vždy na jednom konkrétním procesoru
  - Master odpovídá za plánování
  - Slave žádá o služby mastera
  - Nevýhoda: dedikace
    - Přetížený master se stává úzkým místem systému
- Symetrický multiprocesing (SMP)
  - Všechny procesory jsou si navzájem rovny
  - Funkce jádra mohou běžet na kterémkoliv procesoru
  - SMP vyžaduje podporu vláken v jádře
  - Proces musí být dělen na vlákna, aby SMP byl účinný
- Aplikace je sada vláken pracujících paralelně do společného adresního prostoru
- Vlákno běží nezávisle na ostatních vláknech svého procesu
- Vlákna běžící na různých procesorech dramaticky zvyšují účinnost systému

? používá většina OS: Windows, Linux, Mac OS X, Solaris, BSD4.4

SMP

Dvě řešení SMP:

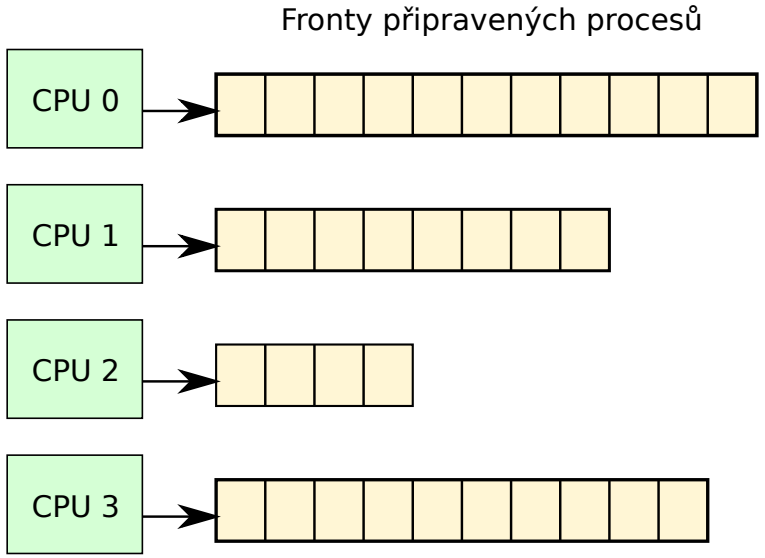
- Jedna společná fronta pro všechny procesory
  - Fronta může být víceúrovňová dle priorit
  - Problémy:
    - Jedna centrální fronta připravených sledů vyžaduje používání vzájemného vylučování v jádře
    - Kritické místo v okamžiku, kdy si hledá práci více procesorů
    - Předběhnutá (přerušená) vlákna nebudou nutně pokračovat na stejném procesoru – nelze proto plně využívat cache paměti procesorů



SMP

Druhé řešení SMP:

- Každý procesor má svojí frontu a občasná migrace vláken mezi procesory má za úkol udržovat fronty přibližně stejně dlouhé
  - Každý procesor si sám vyhledává příští vlákno
  - Přesněji: instance plánovače běžící na procesoru si je sama vyhledává
  - Problémy – některé fronty jsou kratší:
    - Heuristická pravidla, kdy frontu změnit



SMP optimalizace

- Používají se různá (heuristická) pravidla (i při globální frontě):
  - Afinita vlákna k CPU – použij procesor, kde vlákno již běželo (možná, že v cache CPU budou ještě údaje z minulého běhu)
  - Afinita vlákna k CPU při globální frontě – neber první proces z fronty, ale prozkoumej více procesů na začátku fronty a hledej proces, který běžel na daném procesoru
  - Použij nejméně využívaný procesor
- Mnohdy značně složité
  - při malém počtu procesorů (4) může přílišná snaha o optimalizaci plánování vést až k poklesu výkonu systému, výběr se dělá při každém rozhodování, kdo poběží
    - Tedy aspoň v tom smyslu, že výkon systému neporoste lineárně s počtem procesorů
  - při velkém počtu procesorů dojde naopak k „nasycení“, neboť plánovač se musí věnovat rozhodování velmi často (končí CPU dávky na mnoha procesorech)
    - režie tak neúměrně roste

Obsah

Jak to bude fungovat?

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

volatile int a;
void *fce(void *n) {
   int i;
   for (i=0; i<10000; i++) {
      a+=1;
   }
   printf("a=%i\n", a);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   pthread_t tid1,tid2;
   a=0;
   pthread_create(&tid1, NULL, fce, NULL);
   pthread_create(&tid2, NULL, fce, NULL);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   return 0;
}
```

Problém synchronizace

- Souběžný přístup ke sdíleným datům může způsobit jejich nekonzistenci
  - nutná koordinace procesů
- Synchronizace běhu procesů
  - Čekání na událost vyvolanou jiným procesem
- Komunikace mezi procesy (IPC = Inter-process Communication) – příští přednáška
  - Výměna informací (zpráv)
  - Způsob synchronizace, koordinace různých aktivit
- Sdílení prostředků – problém soupeření či souběhu (race condition)
  - Procesy používají a modifikují sdílená data
  - Operace zápisu musí být vzájemně výlučné
  - Operace zápisu musí být vzájemně výlučné s operacemi čtení
  - Operace čtení (bez modifikace) mohou být realizovány souběžně
  - Pro zabezpečení integrity dat se používají kritické sekce

Producent konzument

Ilustrační příklad

- Producent generuje data do vyrovnávací paměti s konečnou kapacitou (bounded-buffer problem) a konzument z této paměti data odebírá
- Zavedeme celočíselnou proměnnou count, která bude čítat platné položky v bufferu. Na počátku je count $= 0$
- Pokud je v poli místo, producent vloží položku do pole a inkrementuje count
- Pokud je v poli nějaká položka, konzument při jejím vyjmutí dekrementuje count

Producent a konsument

Sdílená data

```
#define BUF_SIZE = 20
typedef struct { /* data */ } item;
item buffer[BUF_SIZE];
int count = 0;
```

Producent

```
void producer() {
    int in = 0;
    item nextProduced;
    while (1) {
        /* Vygeneruj novou položku do
           proměnné nextProduced */
        while (count == BUF_SIZE);
          /* čekání nedělej nic */
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
        count++;
    }
}
```

Konzument

```
void consumer() {
    int out = 0;
    item nextConsumed;
    while (1) {
        while (count == 0);
          /* čekání nedělej nic */
        nextConsumed =  buffer[out];
        out = (out + 1) % BUF_SIZE;
        count--;
        /*  Zpracuj položku z
          proměnné nextConsumed */
    }
}
```

Kde je problém?

Problém soupeření

- count ++ bude obvykle implementováno:

P_1:	**count** → registr	`mov count, %eax`
P_2:	registr+1 → regist	`add 1, %eax`
P_3:	registr → **count**	`mov %eax, count`

- count - - bude obvykle implementováno:

K_1:	**count** → registr	`mov count, %eax`
K_2:	registr-1 → registr	`sub 1, %eax`
K_3:	registr → **count**	`mov %eax, count`

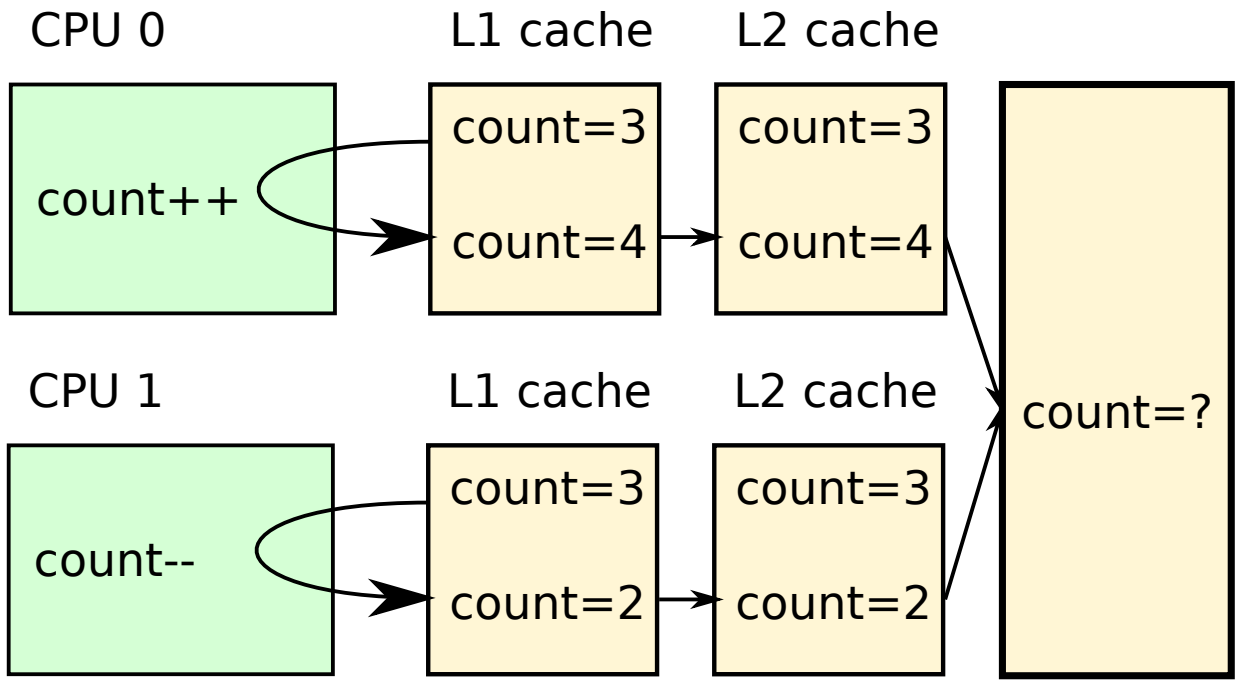
Může nastat následující paralelizace procesů konzument a producent:

akce	běží	akce	výsledek
P_1:	producent	**count** → registr	eax = 3
P_2:	producent	registr+1 → registr	eax = 4
K_1:	konzument	**count** → registr	eax = 3
K_2:	konzument	registr-1 → registr	eax = 2
K_3:	konzument	registr → **count**	count = 2
P_3:	producent	registr → **count**	count = 4

- Na konci může být count roven 2 nebo 4, ale správně je 3 (což se většinou podaří)
- Je to důsledkem nepředvídatelného prokládání procesů/vláken vlivem možné preempce

Problém soupeření – cache

- Problém soupeření je i při vícejádrových procesorech
- Jedna proměnná je uložena na více místech cache úrovně L1, úrovně L2 a pouze jednom místě úrovně L3 a paměti RAM



- Výsledek zápisu je určen kdo přijde dříve a kdo později, ale pouze hodnota 2 nebo 4

Kritická sekce

- Problém lze formulovat obecně:
  - Jistý čas se proces zabývá svými obvyklými činnostmi a jistou část své aktivity věnuje sdíleným prostředkům.
  - Část kódu programu, kde se přistupuje ke sdílenému prostředku, se nazývá kritická sekce procesu vzhledem k tomuto sdílenému prostředku (nebo také sdružená s tímto prostředkem).
- Je potřeba zajistit, aby v kritické sekci sdružené s jistým prostředkem, se nacházel nejvýše jeden proces
  - Pokud se nám podaří zajistit, aby žádné dva procesy nebyly současně ve svých kritických sekcích sdružených s uvažovaným sdíleným prostředkem, pak je problém soupeření vyřešen.
- Modelové prostředí pro řešení problému kritické sekce
  - Předpokládá se, že každý z procesů běží nenulovou rychlostí
  - Řešení nesmí záviset na relativních rychlostech procesů

Požadavky na kritickou sekci

- Vzájemné vyloučení – podmínka bezpečnosti (Mutual Exclusion)
  - Pokud proces P_i je ve své kritické sekci, pak žádný další proces nesmí být ve své kritické sekci sdružené s týmž prostředkem
- Trvalost postupu – podmínka živosti (Progress)
  - Jestliže žádný proces neprovádí svoji kritickou sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené se tímto zdrojem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho.
- Konečné čekání – podmínka spravedlivosti (Fairness)
  - Proces smí čekat na povolení vstupu do kritické sekce jen konečnou dobu.
  - Musí existovat omezení počtu, kolikrát může být povolen vstup do kritické sekce sdružené se jistým prostředkem jiným procesům než procesu požadujícímu vstup v době mezi vydáním žádosti a jejím uspokojením.

Řešení kritických sekcí

Základní struktura procesu s kritickou sekcí:

```
do {
    enter_cs();
       // critical section
    leave_cs ();
       // non-critical section
} while (TRUE);
```

Klíčem k řešení celého problému kritických sekcí je korektní implementace funkcí enter_cs() a leave_cs().

- Čistě softwarová řešení na aplikační úrovni
  - Algoritmy, jejichž správnost se nespoléhá na další podporu
  - Základní (a problematické) řešení s aktivním čekáním (busy waiting)
- Hardwarové řešení
  - Pomocí speciálních instrukcí CPU
  - Stále ještě s aktivním čekáním
- Softwarové řešení zprostředkované operačním systémem
  - Potřebné služby a datové struktury poskytuje OS (např. semafory)
  - Tím je umožněno pasivní čekání – proces nesoutěží o procesor
  - Podpora volání synchronizačních služeb v programovacích systémech/jazycích (např. monitory, zasílání zpráv)

Řešení na aplikační úrovni

Zaveďme proměnnou lock, jejíž hodnota určuje, zda je kritická sekce obsazená

```
while(TRUE) {
    while(lock!=0);
       /* čekej */
    lock = 1;
    critical_section();
    lock = 0;
    noncritical_section();
}
```

Je zde nějaký problém?

Řešení na aplikační úrovni

Zaveďme proměnnou lock, jejíž hodnota určuje, zda je kritická sekce obsazená

```
while(TRUE) {
    while(lock!=0);
       /* čekej */
    lock = 1;
    critical_section();
    lock = 0;
    noncritical_section();
}
```

Je zde nějaký problém?

- **Je to úplně špatně!**
- Protože mezi otestováním proměnné lock a jejím nastavení je možné, že proběhne další otestování jiným vláknem.
- Neřeší tedy základní podmínku exkluzivity kritické sekce

Řešení na aplikační úrovni

Striktní střídání dvou procesů nebo vláken.

- Zaveďme proměnnou turn, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce.
- Je-li `turn == 0`, do kritické sekce může P_0,
- je-li `turn == 1`, pak P_1.

P_0

```
while(TRUE) {
        while(turn!=0);
           /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1

```
while(TRUE) {
        while(turn!=1);
           /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

Je zde nějaký problém?

Řešení na aplikační úrovni

Striktní střídání dvou procesů nebo vláken.

- Zaveďme proměnnou turn, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce.
- Je-li `turn == 0`, do kritické sekce může P_0,
- je-li `turn == 1`, pak P_1.

P_0

```
while(TRUE) {
        while(turn!=0);
           /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1

```
while(TRUE) {
        while(turn!=1);
           /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

Je zde nějaký problém?

- P_0 proběhne svojí kritickou sekcí velmi rychle, turn $= 1$ a oba procesy jsou v nekritických částech. P_0 je rychlý i ve své nekritické části a chce vstoupit do kritické sekce. Protože však turn $== 1$, bude čekat, přestože kritická sekce je volná.
- Je porušen požadavek Trvalosti postupu
- Navíc řešení nepřípustně závisí na rychlostech procesů

Petersonovo řešení

- Petersonovo řešení střídání dvou procesů nebo vláken
- Řešení pro dva procesy P_i (i = 0, 1) – dvě globální proměnné:
  - `int turn;`
    - Proměnná turn udává, který z procesů je na řadě při přístupu do kritické sekce
  - `boolean interest[2];`
    - V poli interest procesy indikují svůj zájem vstoupit do kritické sekce; (interest[i]==TRUE) znamená, že P_i tuto potřebu má
    - Prvky pole interest nejsou sdílenými proměnnými.

```
j = 1 - i;
interest[i] = TRUE;
turn = j;
while (interest[j] && turn == j) ;        /* čekání */
/*  KRITICKÁ SEKCE                    */
interest[i] = FALSE;
/*  NEKRITICKÁ ČÁST PROCESU           */
```

- Náš proces bude čekat jen pokud druhý proces je na řadě a současně má zájem do kritické sekce vstoupit
- Všechna řešení na aplikační úrovni obsahují aktivní čekání, nebo používají funkci sleep/usleep

Petersonovo řešení

```
int a;
volatile int turn;
volatile int interest[2];

void *fce(void *n) {
  int i;
  int j=*(int*)n;
  for (i=0; i<1000000; i++) {
    interest[j]=1;
    __sync_synchronize();     /* memory barrier */
    turn = (1-j);
    while (interest[1-j]==1 && turn==(1-j));  /* repeat and wait */
    a+=1;
    interest[j]=0;
  }
  printf("a=%i\n", a);
  pthread_exit(NULL);
}
```

Memory barrier

- Většina moderních CPU umí měnit pořadí dvou po sobě jdoucích instrukcí kvůli zrychlení přístupu do paměti.
- Pro Petersonovo řešení je pořadí zápisu do proměnných `turn` a `interest` klíčové
- `__sync_synchronize` memory barrier pro překladač gcc (visual studio má funkci `MemoryBarrier`)
- memory barrier umožní i synchronizaci cache pamětí

```
interest[i] = TRUE;
__sync_synchronize(); /* memory barrier */
turn = j;
while (interest[j] && turn == j) ;        /* čekání */
```

- Nyní je všechno v pořádku a řešení funguje

Petersonovo řešení

Obecné řešení pro N procesů

- je již daleko více komplikovanější, tím je náchylnější k implementační chybě
- proměnné `level` charakterizují, kdo čeká na kritickou sekci
- proces, který dospěje až do nejvyšší úrovně (level), tak získá kritickou sekci

```
int level[N]
int last_to_enter[N-1]
for (l=0; l<N-1; l++)
  level[i] = l
  last_to_enter[l] = i
  while (last_to_enter[l] == i and exists k != i; level[k] >= l)
    wait;
```

- kontrukce exist k je zkratka za

```
set = False
for (k=0; k<N; k++) {
  if (k!=i && level[k] >= l)
    set = True;
```

HW podpora

- Využití zamykací proměnné je rozumné, avšak je nutná atomicita
- Jednoprocesorové systémy mohou vypnout přerušení, při vypnutém přerušení nemůže dojít k preempci
  - Nelze použít na aplikační úrovni (vypnutí přerušení je privilegovaná akce)
  - Nelze jednoduše použít pro víceprocesorové systémy
- Moderní systémy nabízejí speciální nedělitelné (atomické) instrukce
  - Tyto instrukce mezi paměťovými cykly „nepustí“ sběrnici pro jiný procesor
  - Instrukce TestAndSet atomicky přečte obsah adresované buňky a bezprostředně poté změní její obsah (tas – MC68k, tsl – Intel)
  - Instrukce Swap (xchg) atomicky prohodí obsah registru procesoru a adresované buňky
  - Např. IA32/64 (I586+) nabízí i další atomické instrukce
  - Prefix „LOCK“ pro celou řadu instrukcí typu read-modify-write (např. ADD, AND, ... s cílovým operandem v paměti)

HW podpora

- tas např. Motorola 68000

```
enter_cs:   tas  lock          ; nastav lock na 1 a otestuj starou hodnotu
            jnz enter_cs       ; byla stará hodnota nenulová?
            ret

leave_cs:   mov $0, lock       ; vynuluj lock pro uvolnění kritické sekce
            ret
```

- xchg – IA32

```
enter_cs:   mov $1, %eax       ; připrav hodnotu 1 pro výměnu
            xchg  lock, %eax   ; eax obsahuje nyní starou hodnotu
            jnz enter_cs       ; byla stará hodnota nenulová
            ret

leave_cs:   mov $0, lock       ; vynuluj lock pro uvolnění kritické sekce
            ret
```

Xchg řešení

```
volatile int a;
volatile int turn;
void *fce(void *n) {
    int i, tmp;
    for (i=0; i<1000000; i++) {
        tmp=1;
        asm volatile ("xchg%z0  %2, %0;"
          : "=g" (turn), "=r" (tmp): "1" (tmp) : );
        while (tmp!=0) {
            asm volatile ("xchg%z0  %2, %0;"
              : "=g" (turn), "=r" (tmp): "1" (tmp) : );
        }
        a+=1;
        turn=0;
    }
    printf("a=%i\n", a);
    pthread_exit(NULL);
}
```

Synchronizace bez aktivního čekání

- Aktivní čekání mrhá strojovým časem
- Může způsobit i nefunkčnost při rozdílných prioritách procesů
- Např. vysokoprioritní producent zaplní pole, začne aktivně čekat a nedovolí konzumentovi odebrat položku (samozřejmě to závisí na metodě plánování procesů a na to navazující dynamicky se měnící priority)
- Blokování pomocí systémových atomických primitiv
  - suspend() místo aktivního čekání – proces se zablokuje
  - wakeup(process) probuzení spolupracujícího procesu při opouštění kritické sekce

```
void producer() {
    while (1) {
        /* Vygeneruj položku do proměnné nextProduced */
        if (count == BUFFER_SIZE)  suspend();      // Je-li pole plné, zablokuj se
        buffer[in] = nextProduced;  in = (in + 1) % BUFFER_SIZE;
        count++ ;
        if  (count == 1) wakeup(consumer);        // Bylo-li pole prázdné, probuď konzumenta
   }
}

void consumer() {
    while (1) {
         if (count == 0) suspend();               // Je-li pole prázdné, zablokuj se
         nextConsumed =  buffer[out];  out = (out + 1) % BUFFER_SIZE;
         count-- ;
         if (count == BUFFER_SIZE-1)              // Bylo-li pole plné, probuď producenta
             wakeup(producer);
         /*  Zpracuj položku z proměnné nextConsumed */
    }
}
```

Problém s čekáním

- Předešlý kód není řešením – zůstalo konkurenční soupeření – count je opět sdílenou proměnnou:
  - Konzument přečetl count == 0 a než zavolá suspend(), je mu odňat procesor
  - Producent vloží do pole položku a count == 1, načež se pokusí se probudit konzumenta, který ale ještě nespí!
  - Po znovuspuštění se konzument domnívá, že pole je prázdné a volá suspend()
  - Po čase producent zaplní pole a rovněž zavolá suspend() – spí oba!
  - Příčinou této situace je ztráta budícího signálu
- Lepší řešení:
  - Jednině OS umí uspat a vzbudit procesy – Semafory, mutexy

Semafor

- Obecný synchronizační nástroj (Edsger Dijkstra, NL, [1930–2002])
- Systémem spravovaný objekt
- Základní vlastností je celočíselná proměnná (obecný semafor, nebo také čítající semafor)
- Dvě standardní atomické operace nad semaforem
  - sem_wait(S) [někdy nazývaná lock(), acquire() nebo down()]
  - sem_post(S) [někdy nazývaná unlock(), release() nebo up()]

```
sem_wait(S) {
    while (S <= 0);
    S--;
}
```

```
sem_post(S) {
    S++;
    // Čeká-li jiný proces před
    // semaforem, pusť ho dál
}
```

- Tato sémantika stále obsahuje aktivní čekání
- Skutečná implementace však aktivní čekání obchází tím, že spolupracuje s plánovačem CPU, což umožňuje blokovat a reaktivovat procesy (vlákna)

Implementace semaforů

Struktura semaforu

```
typedef struct {
    int value;                  // „Hodnota" semaforu
    struct process *list;               // Fronta procesů stojících „před semaforem"
} sem_t;
```

Operace nad semaforem jsou pak implementovány jako nedělitelné s touto sémantikou

```
void sem_wait(sem_t *S) {
    S->value= S->value - 1;
    if (S->value < 0)           // Je-li třeba, zablokuj volající proces a zařaď ho
        block(S->list);         //   do fronty před semaforem (S.list)
}

void sem_post(sem_t *S) {
    S->value= S->value + 1
    if (S->value <= 0) {
        if (S->list != NULL) {      // Je-li fronta neprázdná
            //   vyjmi proces P z čela fronty
            wakeup(P);          //    a probuď P
        }
    }
}
```

Implementace semaforů

- Záporná hodnota S.value udává, kolik procesů „stojí“ před semaforem
- Fronty před semaforem:
  - Většinou FIFO bez uvažování priorit procesů, jinak vzniká problém se stárnutím
  - Systémy reálného času (RTOS) většinou prioritu uvažují
- Operace wait(S) a post(S) musí být vykonány atomicky
- OS na jednom procesoru nemá problém, OS rozhoduje o přepnutí procesu
- OS na více jádrech:
  - Jádro musí používat atomické instrukce či jiný odpovídající hardwarový mechanismus na synchronizaci skutečného paralelizmu
  - Instrukce xchg, tas, či prefix lock musí umět zamknout sběrnici proti přístupu jiných jader, či zamknout a aktualizovat cache systémem cache snooping

Mutex

- Mutex – speciální rychlejší semafor, hodnoty pouze 1,0 binární semafor
- Implementace musí zaručit:
- Operace lock() (odpovídá funkci wait() u semaforu) a unlock() (odpovídá funkci post()) musí být atomické stejně jako u semaforů
- Aktivní čekání není plně eliminováno, je ale přesunuto z aplikační úrovně (kde mohou být kritické sekce dlouhé) do úrovně jádra OS pro implementaci atomicity operací se semafory
- Mutex definuje koncept "vlastníka mutexu" a díky tomu jej lze například zamykat rekurzivně z jednoho vlákna nebo lze implementovat mechanismus pro zabránění uváznutí.

Užití:

```
void *fce(void *n) {
  int i;
  for (i=0; i<100000; i++) {
    pthread_mutex_lock(&mutex);
    a+=1;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}
```

Producent – konzument

Tři semafory

- **mutex** s iniciální hodnotou 1 – pro vzájemné vyloučení při přístupu do sdílené paměti
- **used** – počet položek v poli – inicializován na hodnotu 0
- **free** – počet volných položek – inicializován na hodnotu BUF_SIZE

```
void producer() {
    while (1) {     /* Vygeneruj položku do proměnné nextProduced */
        sem_wait(&free);
        sem_wait(&mutex);
        buffer [in] = nextProduced;  in = (in + 1) % BUF_SZ;
        sem_post(&mutex);
        sem_post(&used);
    }
}

void consumer() {
    while (1) {
        sem_wait(&used);
        sem_wait(&mutex);
        nextConsumed =  buffer[out];  out = (out + 1) % BUF_SZ;
        sem_post(&mutex);
        sem_post(&free);
        /*  Zpracuj položku z proměnné nextConsumed */
    }
}
```

Producent – konzument jen s mutexy

Pro korektní uspání potřebujeme podmínkové proměnné

- čekání na podmínku se provádí funkcí `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- časově omezené čekání na podmínku se provádí funkcí `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- při čekání na podmínku se spojí podmínka s mutexem, který se čekáním uvolní
- probuzení vlákna čekajícího na podmínku se provede funkcí `int pthread_cond_signal(pthread_cond_t *cond);`
- probuzení všech vláken čekající na konkrétní podmínku se provede funkcí `int pthread_cond_broadcast(pthread_cond_t *cond);`
- při probuzení vlákna, čekajícího na podmínku se opět mutex obsadí (tedy vlákno počká na jeho uvolnění)
- čekání na podmínku musí být vždy uvnitř kritické sekce mutexu
- probuzení cizího vlákna nastavením podmínky, by mělo být také uvnitř kritické sekce mutexu

Prodmínkové proměnné

```
void *producer(void *i) {
   char str[256], *s;
   int wr_ptr=0;

   while ((s=fgets(str, 250, stdin))!=NULL) {
     pthread_mutex_lock(&mutex);
     while (glob_free<=0) {
       printf("Wait full\n");
       pthread_cond_wait(&full, &mutex);
     }
     memcpy(global[wr_ptr], s, 256);
     glob_free--;
     pthread_cond_signal(&empty);
     pthread_mutex_unlock(&mutex);

     wr_ptr=(wr_ptr+1)%8;
  }
  glob_end=1;
  pthread_mutex_lock(&mutex);
  pthread_cond_signal(&empty);
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

```
void *consumer(void *i) {
  int rd_ptr=0;
  while (!glob_end) {
    pthread_mutex_lock(&mutex);
    while(glob_free>=8 && !glob_end) {
      printf("Wait empty\n");
      pthread_cond_wait(&empty, &mutex);
    }
    if (glob_free<8) {
      printf("Zadano: %s\n", global[rd_ptr]);
      glob_free++;
    }
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
    rd_ptr=(rd_ptr+1)%8;
    sleep(1);
  }
  return NULL;
}
```

Čtenáři a písaři

- Úloha: Několik procesů přistupuje ke společným datům
- Některé procesy data jen čtou – čtenáři
- Jiné procesy potřebují data zapisovat – písaři
- Souběžné operace čtení mohou čtenou strukturu sdílet – libovolný počet čtenářů může jeden a tentýž zdroj číst současně
- Operace zápisu musí být exklusivní, vzájemně vyloučená s jakoukoli jinou operací (zápisovou i čtecí)
  - v jednom okamžiku smí daný zdroj modifikovat nejvýše jeden písař
  - Jestliže písař modifikuje zdroj, nesmí ho současně číst žádný čtenář
- Dva možné přístupy
  - Přednost čtenářů
    - Žádný čtenář nebude muset čekat, pokud sdílený zdroj nebude obsazen písařem. Jinak řečeno: Kterýkoliv čtenář čeká pouze na opuštění kritické sekce písařem.
    - Písaři mohou stárnout
  - Přednost písařů
    - Jakmile je některý písař připraven vstoupit do kritické sekce, čeká jen na její uvolnění (čtenářem nebo písařem). Jinak řečeno: Připravený písař předbíhá všechny připravené čtenáře.
    - Čtenáři mohou stárnout

Priorita čtenářů

- Sdílená data
  - semaphore wrt, readcountmutex;
  - int readcount
- Inicializace
  - wrt = 1; readcountmutex = 1; readcount = 0;

Čtenář:

```
sem_wait(readcountmutex);
readcount++;
if (readcount==1) sem_wait(wrt);
sem_post(readcountmutex);

   // čtení sdíleného zdroje

sem_wait(readcountmutex);
readcount--;
if (readcount==0) sem_post(wrt);
sem_post(readcountmutex);
```

Písař:

```
sem_wait(wrt);
   // písař modifikuje zdroj
sem_post(wrt);
```

Priorita písařů

- Sdílená data
  - semaphore wrt, rdr, readcountmutex, writecountmutex;
  - int readcount, writecount;
- Inicializace
  - wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
  - readcount = 0; writecount = 0;

Písař:

```
sem_wait(writecountmutex);
writecount++;
if (writecount==1) sem_wait(rdr);
sem_post(writecountmutex);
sem_wait(wrt);
   // písař modifikuje zdroj
sem_post(wrt);
sem_wait(writecountmutex);
writecount--;
if (writecount==0) sem_post(rdr);
sem_post(writecountmutex);
```

Čtenář:

```
sem_wait(rdr);
sem_wait(readcountmutex);
readcount++;
if (readcount == 1) sem_wait(wrt);
sem_post(readcountmutex);
sem_post(rdr);
   // čtení sdíleného zdroje
sem_wait(readcountmutex);
readcount--;
if (readcount == 0) sem_post(wrt);
sem_post(readcountmutex);
```

Monitor

- Monitor je synchronizační nástroj vyšší úrovně
- Umožňuje bezpečné sdílení libovolného datového typu
- Na rozdíl od semaforů, monitor explicitně definuje která data jsou daným monitorem chráněna
- Monitor je jazykový konstrukt v jazycích „pro paralelní zpracování“
- Podporován např. v Concurrent Pascal, Modula-3, C, ...
- V Javě může každý objekt fungovat jako monitor (viz metoda Object.wait() a klíčové slovo synchronized)
- Procedury definované jako monitorové procedury se vždy vzájemně vylučují

```
monitor monitor_name {
    int i;                      //  Deklarace sdílených proměnných
    void p1(...) { ... }        //  Deklarace monitorových procedur
    void p2(...) { ... }
    {
        // inicializační kód
    }
}
```

Synchronizace v Javě

- Java používá pro synchronizaci Monitor
- Uživatel si může nadefinovat semafor následovně:

```
public class CountingSemaphore {
  private int signals = 1;

  public synchronized void sem_wait() throws InterruptedException{
    while(this.signals <= 0) wait();
    this.signals--;
  }

  public synchronized void sem_post() {
    this.signals++;
    this.notify();
  }
}
```

Případně lze použít i efektivnější java.util.concurrent.Semaphore

Spin-lock

- Spin-lock je obecný (čítající) semafor, který používá aktivní čekání místo blokování
- Blokování a přepínání mezi procesy či vlákny by bylo časově mnohem náročnější než ztráta strojového času spojená s krátkodobým aktivním čekáním
- Používá se ve víceprocesorových systémech pro implementaci krátkých kritických sekcí
- Typicky uvnitř jádra
- Např. při obsluze přerušení, kde není možné blokování (přerušení není součástí žádného procesu, jedná se o hardwarový koncept)
- Další použití je pro krátké kritické sekce, např. zajištění atomicity operací se semafory (ale to se většinou řeší efektivnějšími atomickými instrukcemi)
- Užito např. v multiprocesorových Windows 2k/XP/7 i Linuxu