B4B35OSY: Operační systémy

Lekce 4. Procesy a vlákna

Petr Štěpán
stepan@fel.cvut.cz

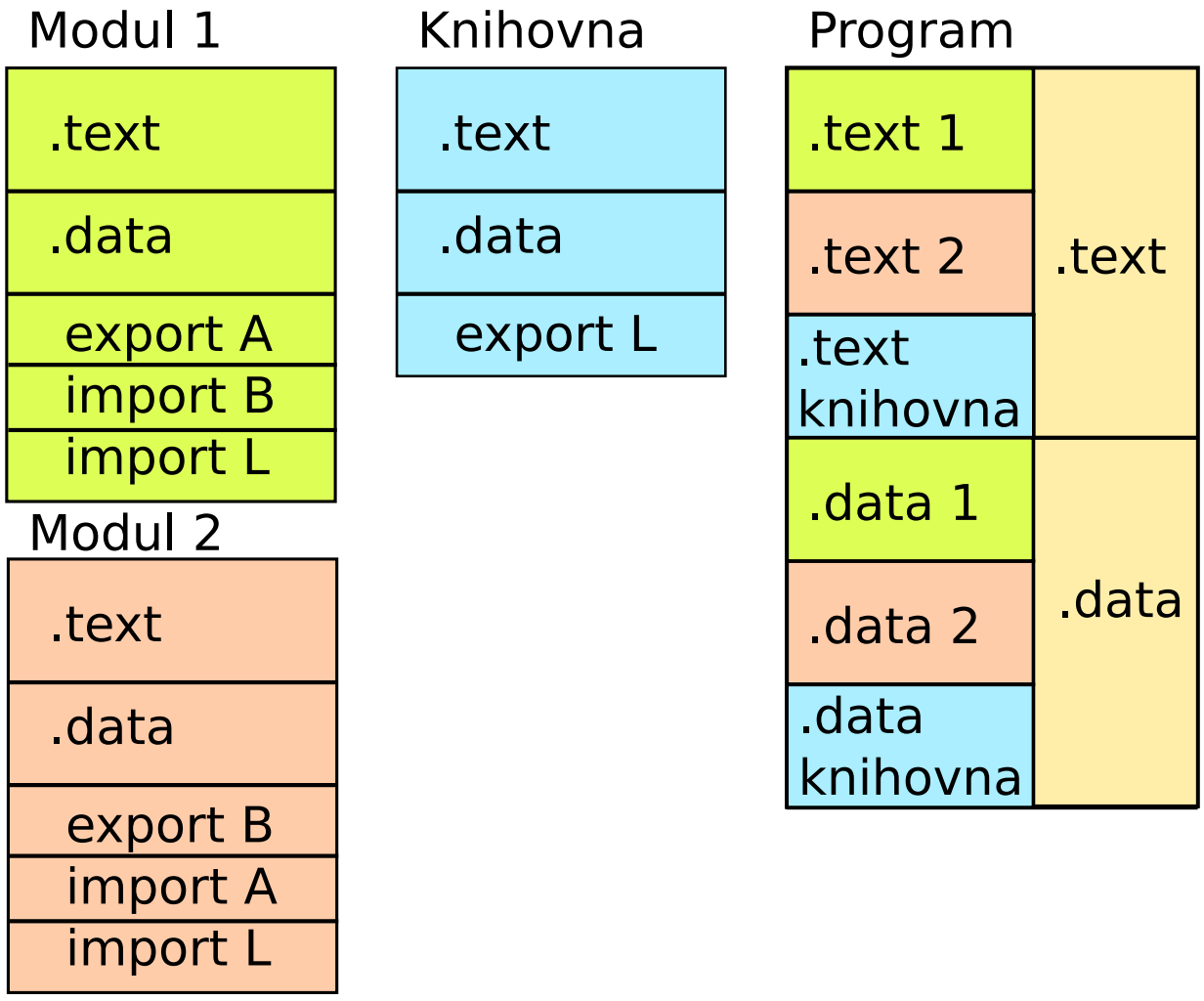
6. října, 2022

Outline

Statické knihovny

- Knihovna je vlastně balík binárních objektových modulů
- Podle symbolů požadovaných moduly programu se hledají moduly v knihovnách, které tyto symboly exportují
- Nový modul z knihovny může vyžadovat další symboly z dalších modulů
- Pokud po projití všech modulů programu i všech modulů knihovny není symbol nalezen, je ohlášena chyba a nelze sestavit výsledný program

Externí symboly - statická knihovna



Dynamické knihovny

- Sestavovací program pracuje podobně jako při sestavování statickém, ale dynamické knihovny nepřidává do výsledného programu.
- Odkazy na symboly z dynamických knihoven je nutné vyřešit až při běhu programu. Existují v zásadě dva přístupy:
  - Vyřešení odkazů **při zavádění programu** do paměti – všechny nepropojené symboly extern se propojí před spuštěním programu
  - **Opožděné sestavování** – na místě nevyřešených odkazů připojí sestavovací program malé kousky kódu (zvané stub), které zavolají systém, aby odkaz vyřešil. Při běhu pak „stub“ zavolá operační systém, který zkontroluje, zda potřebná dynamická knihovna je v paměti (není-li zavede ji do paměti počítače), ve virtuální paměti knihovnu připojí tak, aby ji proces viděl. Následně stub nahradí správným odkazem do paměti a tento odkaz provede.
    - Výhodné z hlediska využití paměti, neboť se nezavádí knihovny, které nebudou potřeba.

Dynamické knihovny PLT/GOT

V předcházejícím případě jsme použili dynamickou knihovnu glibc.
Začátek programu (zkráceno)

```
080482f0 <_start>:
 80482f0:       31 ed                   xor    %ebp,%ebp
 ...
 8048307:       68 03 84 04 08          push   $0x8048403
 804830c:       e8 cf ff ff ff          call   80482e0 <__libc_start_main@plt>
```

Funkce `__libc_start_main@plt` je zodpovědná za dynaminckou funkci start_main z knihovny libc

```
080482e0 <__libc_start_main@plt>:
 80482e0:       ff 25 10 a0 04 08       jmp    *0x804a010
 80482e6:       68 08 00 00 00          push   $0x8
 80482eb:       e9 d0 ff ff ff          jmp    80482c0 <_init+0x2c>
```

V okamžiku kompilace a spuštění je v paměti na adrese `0x804a010` hodnota `0x80482e6`

```
Disassembly of section .got.plt:
0804a000 <_GLOBAL_OFFSET_TABLE_>:
        ...
 804a010:       e6 82 04 08
```

- Po zavedení knihovna dojde k přepsání GOT položky pro tuto funkci na správnou adresu funkce v paměti
- Při dalším volání tedy již instrukce `jmp *0x804a010` skočí přímo do funkce `__libc_start_main`

PIC a DLL

Dynamické knihovny jsou sdíleny různými procesy. Buď musí být na stejné pozici/relokovatelná (dll) nebo musí být na pozici nezávislé (PIC)

- PIC = Position Independent Code
  - Překladač generuje kód nezávislý na umístění v paměti
  - Skoky v kódu a odkazy na data jsou buď relativní vůči IP, nebo podle GOT – Global offset table
  - Pokud nelze k adresaci použít registr IP, je nutné zjistit svoji polohu v paměti:

```
        call .tmp1
.tmp1:  pop %edi
        addl $_GLOBAL_OFFSET_TABLE - .tmp1, %edi
```

  - pro x86_64 lze použít pro adresaci registr `RIP` (číslo `0x2009db` je posunutí GOT od prováděné instrukce, spočítáno překladačem)

```
mov  $0x2009db(%rip), %rax
```

  - kód je sice obvykle delší, avšak netřeba cokoliv modifikovat při sestavování či zavádění
  - užívá se zejména pro dynamické knihovny
  - v poslední době se využívá i pro programy

DLL

- DLL – knihovna je na stejném místě pro všechny procesy
  - pokud se zavádí nová knihovna pro další process, pak musí být na volném místě
  - pokud není volné místo tam, kam je připravena, musí se relokovat – posunout všechny vnitřní pevné odkazy (skoky a data)
  - MS přiděluje místa v paměti na požádání vývojářů, aby minimalizoval možnost kolize
  - Vaše knihovna bude pravděpodobně v kolizi a bude se proto přesouvat – pomalejší provedení programu s touto knihovnou

Zavaděč

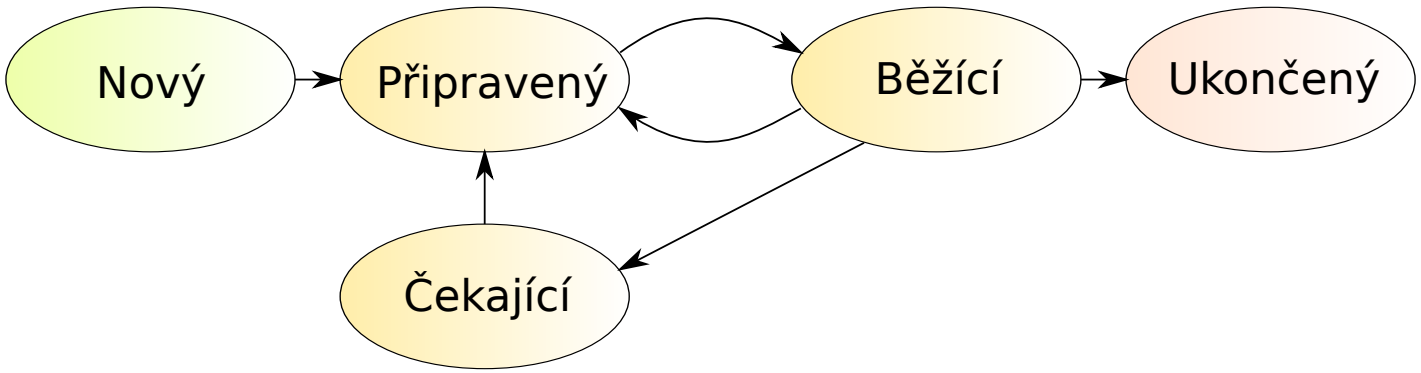
- Zavaděč – loader – je zodpovědný za spuštění programu
- V POSIX systémech je to vlastně obsluha služby „execve“
- Úkoly zavaděče
  - vytvoření „obrazu procesu“ (memory image) v odkládacím prostoru na disku a částečně i v hlavní paměti v závislosti na strategii virtualizace, případné vyřešení nedefinovaných odkazů
  - sekce ze spustitelného souboru se stávají segmenty procesu (pokud správa paměti nepodporuje segmentaci, pak stránkami)
  - segmenty získávají příslušná „práva“ (RW, RO, EXEC, ...)
  - inicializace „registrů procesu“ v PCB
    - např. ukazatel zásobníku a čítač instrukcí
  - předání řízení na vstupní adresu procesu

Obsah

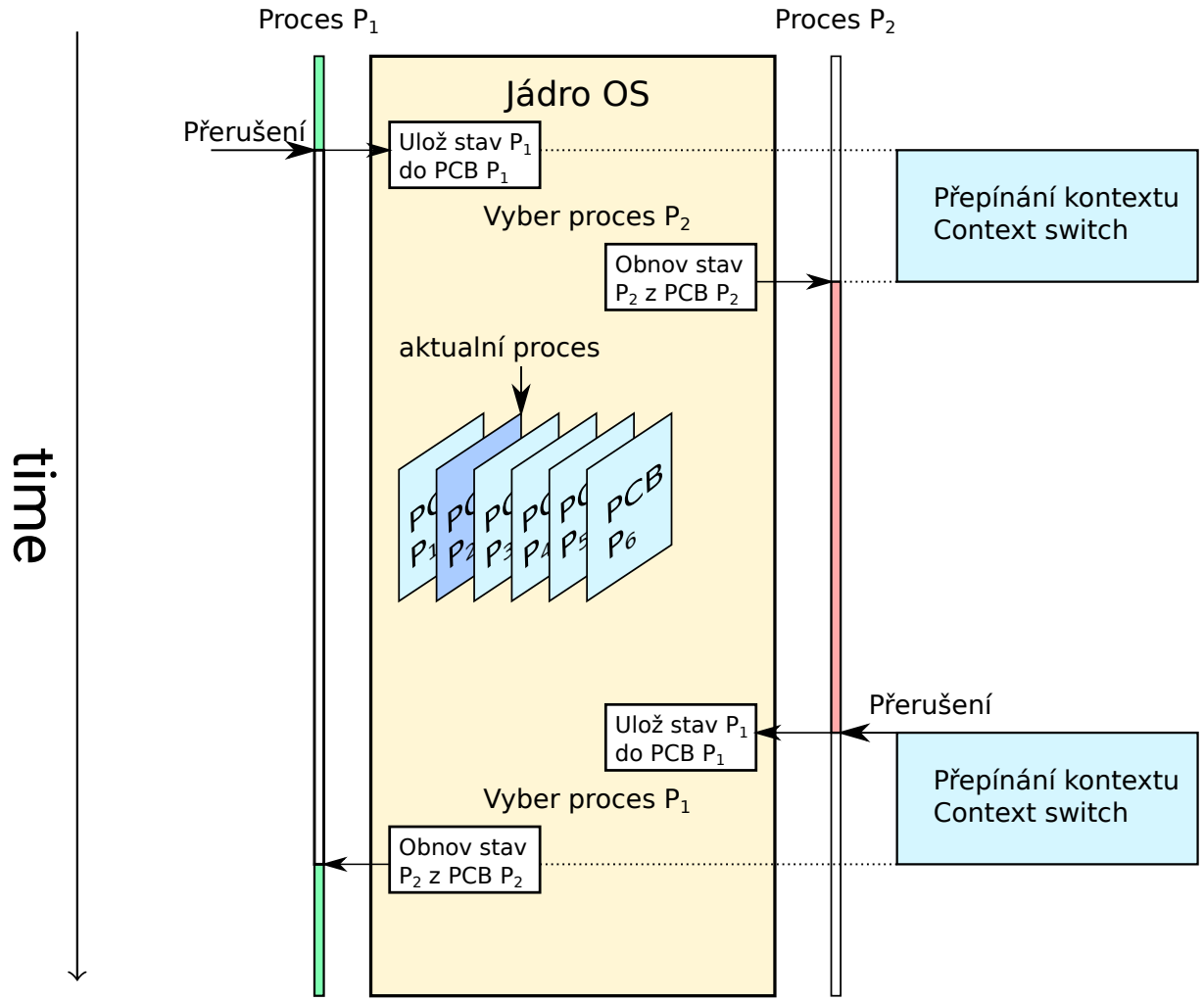
Stavy procesu

Proces se za dobu své existence prochází více stavy a nachází se vždy v jednom z následujících stavů:

- Nový (new) – proces je právě vytvářen, ještě není připraven k běhu, ale již jsou připraveny některé části
- Připravený (ready) – proces čeká na přidělení procesoru
- Běžící (running) – instrukce procesu jsou právě vykonávány procesorem, tj. interpretovány některým procesorem
- Čekající (waiting, blocked) – proces čeká na událost
- Ukončený (terminated) – proces ukončil svoji činnost, avšak stále ještě vlastní některé systémové prostředky



Přepínání procesů



Přepínání procesů

- Přechod od procesu P_1 k P_2 zahrnuje tzv. přepnutí kontextu
- Přepnutí od jednoho procesu k jinému nastává výhradně v důsledku nějakého přerušení (či výjimky)
- Proces P_1 přejde do jádra operačního systému, který provede přepnutí kontextu $\rightarrow$ spustí se proces P_2
  - Nejprve OS uschová stav původně běžícího procesu P_1 v $PCBP_1$
  - jádru OS rozhodne, který proces poběží dál – P_2
  - Obnoví se stav procesu P_2 z PCB P_2
- Přepnutí kontextu představuje režijní ztrátu
  - během přepínání systém nedělá nic užitečného, nepoběží žádný proces
  - časově nejnáročnější je správa paměti dotčených procesů
- Doba přepnutí závisí na hardwarové podpoře v procesoru
  - minimální hardwarová podpora je implementace přerušení:
    - uchování IP a FLAGS
    - naplnění IP a FLAGS ze zadaných hodnot
  - lepší podpora:
    - ukládání a obnova více/všech registrů procesoru jedinou instrukcí
    - vytvoří otisk stavu procesoru do paměti a je schopen tento otisk opět načíst (pusha/popa)

NOVA – přepínání procesů – vstup do jádra

Vstup do jádra

- Prohlédněte si `kern/src/entry.S`

```
;; System-Call Entry
.align 4, 0x90
.globl entry_sysenter
entry_sysenter:
    cld
    pop   %esp
    lea   -44(%esp), %esp
    pusha
    mov   $(KSTCK_ADDR + PAGE_SIZE), %esp
    jmp   syscall_handler
```

- pop esp - nastav zásobník podle hodnoty ze systémového zásobníku
- pusha - ulož všechny registry do struktury uchovávající data procesu
- mov $(KSTCK_ADDR + PAGE_SIZE), %esp - nastav stack pro volání funkcí uvnitř obsluhy systémového volání
- jmp syscall_handler - spusť funkci syscall_handler z `kern/src/ec_syscall.cc`

NOVA – přepínání procesů – návrat z jádra

Návrat z jádra

- Prohlédněte si `kern/src/ec.cc`

```
void Ec::ret_user_sysexit() {
    asm volatile (
      "lea %0, %%esp;"
      "popa;"
      "sti;"
      "sysexit"
      : : "m" (current->regs)
      : "memory");
    UNREACHED;
}
```

- lea %0, esp - nastav zásobník na current->regs
- popa - obnov všechny registry
- sti - povol přerušení
- sysexit - vrať se ze systémového volání

Popis procesů

Process Control Block (PCB), v NOVĚ třída Ec – Execution context

- Obsahuje veškeré údaje o procesu – Linux `task_struct` – include/linux/sched.h
- Datová struktura obsahující:
  - Identifikátor procesu (PID) a rodičovského procesu (PPID)
  - Globální stav (process state)
  - Místo pro uložení všech registrů procesoru
  - Informace potřebné pro plánování procesoru/ů
  - Priorita, historie využití CPU
  - Informace potřebné pro správu paměti
  - Informace o právech procesu, kdo ho spustil
  - Stavové informace o V/V (I/O status)
  - Otevřené soubory
  - Proměnné prostředí (environment variables)
  - $\cdots$ (spousta dalších informací)
  - Ukazatelé pro řazení PCB do front a seznamů

Fronty procesů pro plánování

- Fronta připravených procesů
  - množina procesů připravených k běhu čekajících pouze na přidělení procesoru
- Fronta na dokončení I/O operace
  - samostatná fronta pro každé zařízení
- Seznam odložených procesů
  - množina procesů čekajících na přidělení místa v hlavní paměti, FAP
- Fronty související se synchronizací procesů
  - množiny procesů čekajících synchronizační události
- Fronta na přidělení prostoru v paměti
  - množina procesů potřebujících zvětšit svůj adresní prostor
- ... (další fronty podle potřeb)
- Procesy mezi různými frontami migrují

Druhy plánovačů

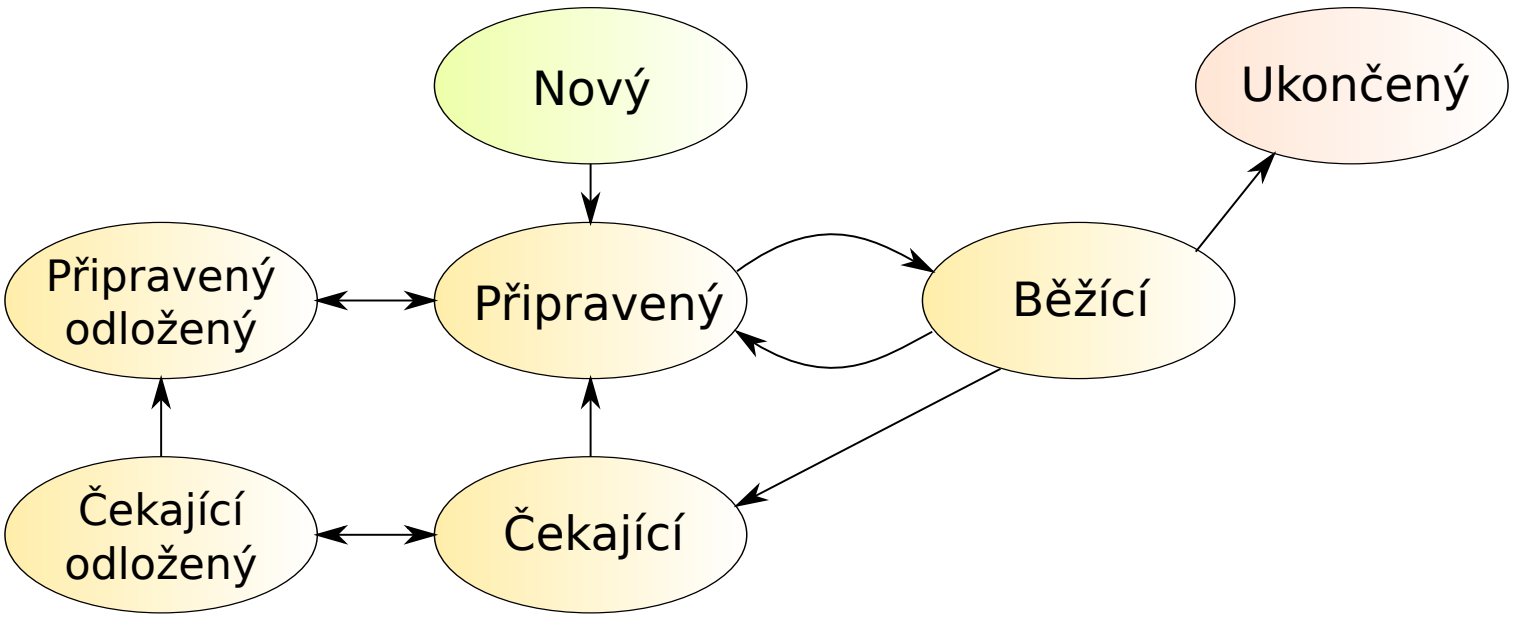
- Krátkodobý plánovač (operační plánovač, dispečer):
  - Základní správa procesoru/ů
  - Vybírá proces, který poběží na uvolněném procesoru přiděluje procesu procesor (CPU)
  - vyvoláván velmi často, musí být extrémně rychlý
- Střednědobý plánovač (taktický plánovač)
  - Úzce spolupracuje se správou 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).



Stavy vláken

- Vlákna podléhají plánování a mají své stavy podobně jako procesy
- Základní stavy
  - běžící
  - připravené
  - čekající
- Všechna vlákna jednoho procesu sdílejí společný adresní prostor
- Vlákna se samostatně neodkládají na disk(process swap), odkládá je jen proces
- Ukončení (havárie) procesu ukončuje všechna vlákna existující v tomto procesu

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)

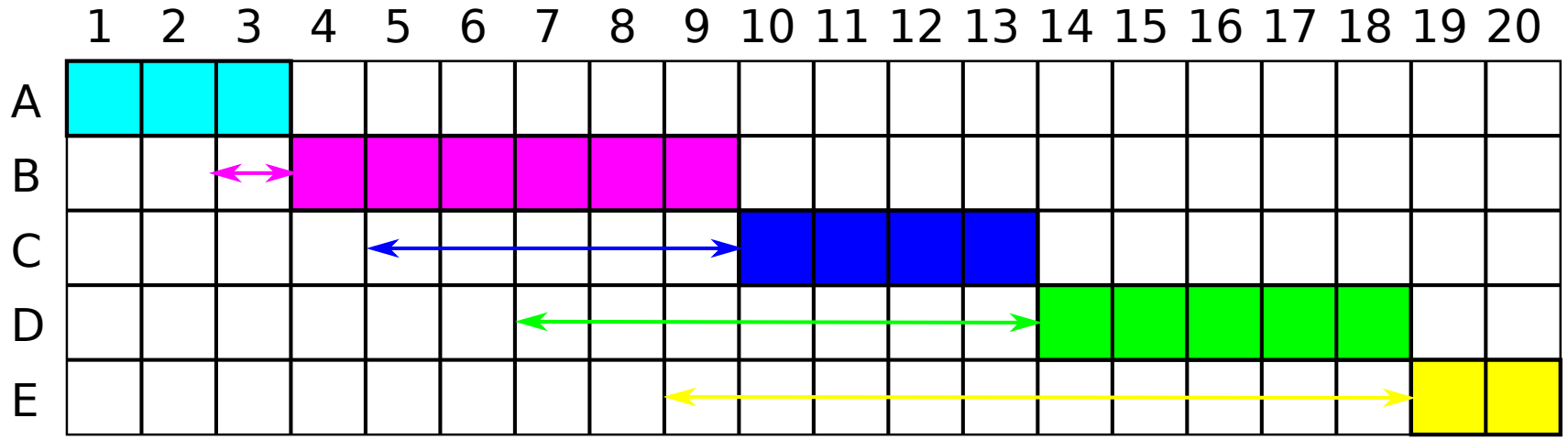
Příklad pro ilustraci algoritmů:

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:



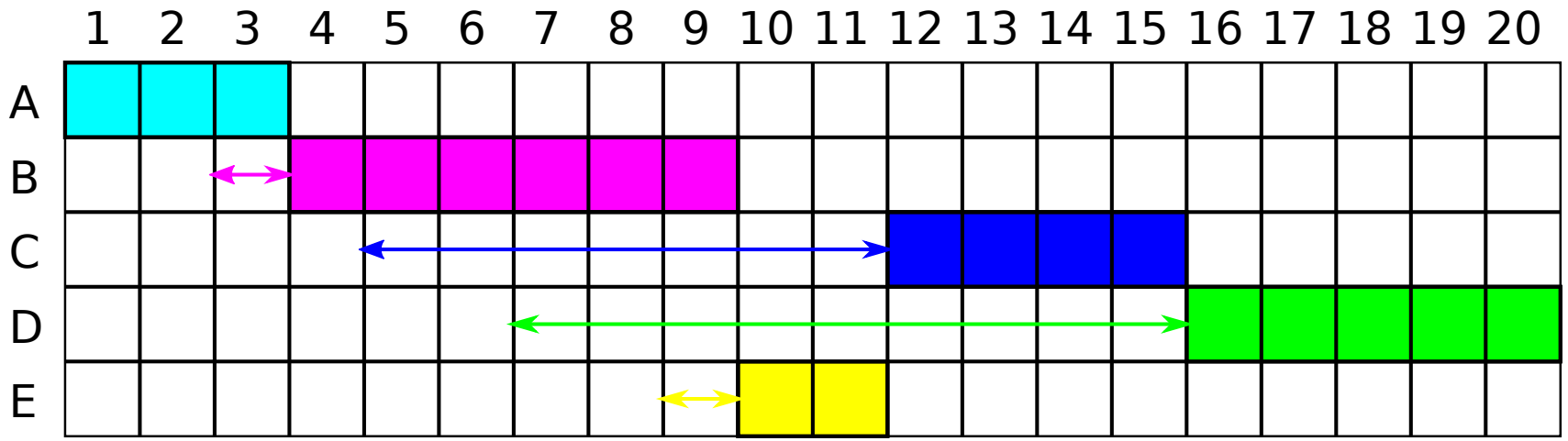
- Průměrné čekání $T_{Avg} = \frac{0+1+5+7+10}{5} = 4.6$
- Průměrné čekání bychom mohli zredukovat, pokud by proces E běžel hned po B.

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

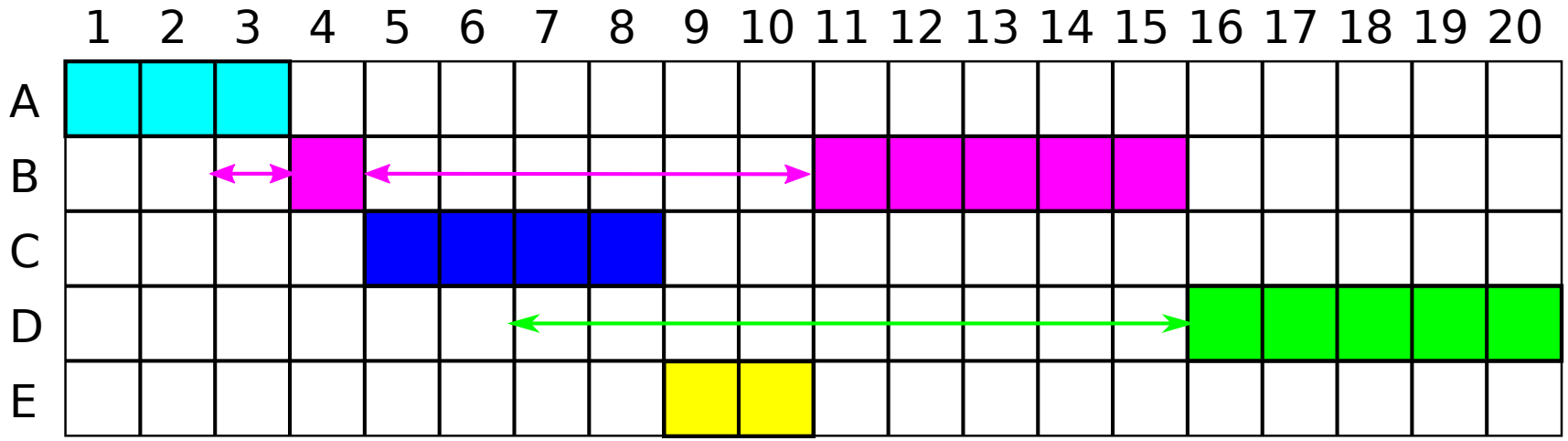
- SPN = Shortest Process Next (nejkratší proces jako příští); též nazýváno SJF = Shortest Job First
  - Opět nepreemptivní
  - Vybírá se připravený proces s nejkratší příští dávkou CPU
  - Krátké procesy předbíhají delší, nebezpečí stárnutí dlouhých procesů
  - Je-li kritériem kvality plánování průměrná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

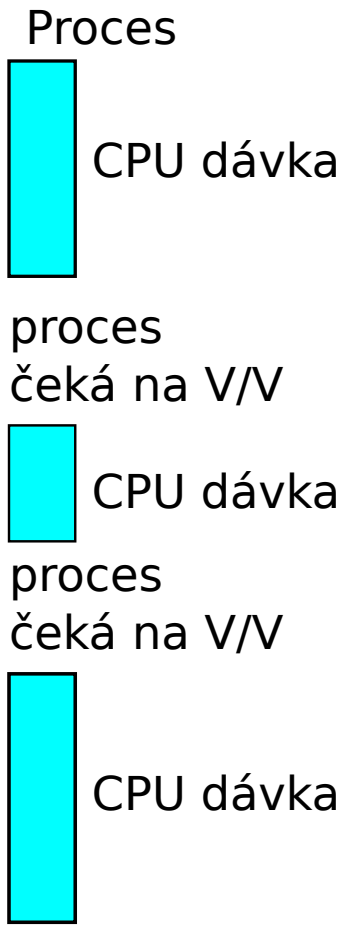
- SRT = Shortest Remaining Time (nejkratší zbývající čas)
- Preemptivní varianta SPN
- CPU dostane proces, který potřebuje nejméně času do svého ukončení
- Jestliže existuje proces, kterému zbývá k jeho dokončení čas kratší, než je čas zbývající do skončení procesu běžícího, dojde k preempci
- Může existovat 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
  - $\alpha, 0 \leq \alpha \leq 1$ 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

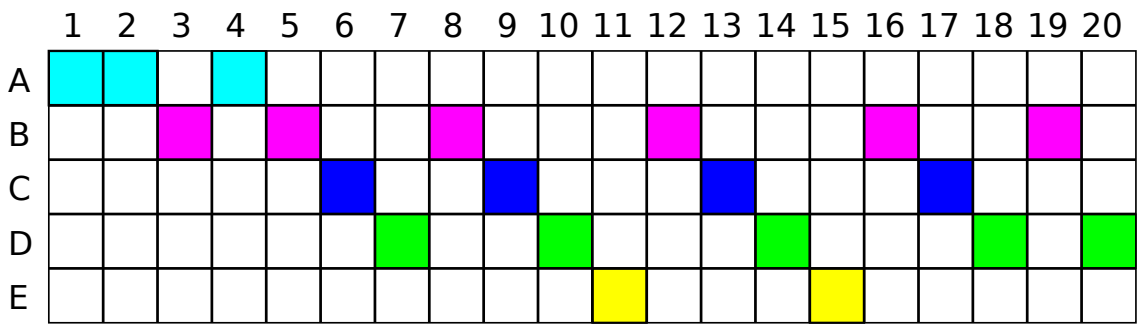
- Problém stárnutí (starvation):
  - Procesy s nízkou prioritou nikdy nepoběží; nikdy na ně nepřijde řada
    - Údajně: Když po řadě let vypínali v roce 1973 na M.I.T. svůj IBM 7094 (jeden z největších strojů své doby), našli proces s nízkou prioritou, který čekal od roku 1967.
- Řešení problému stárnutí: 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 $q \cdot (n-1)$

Cyklické plánování

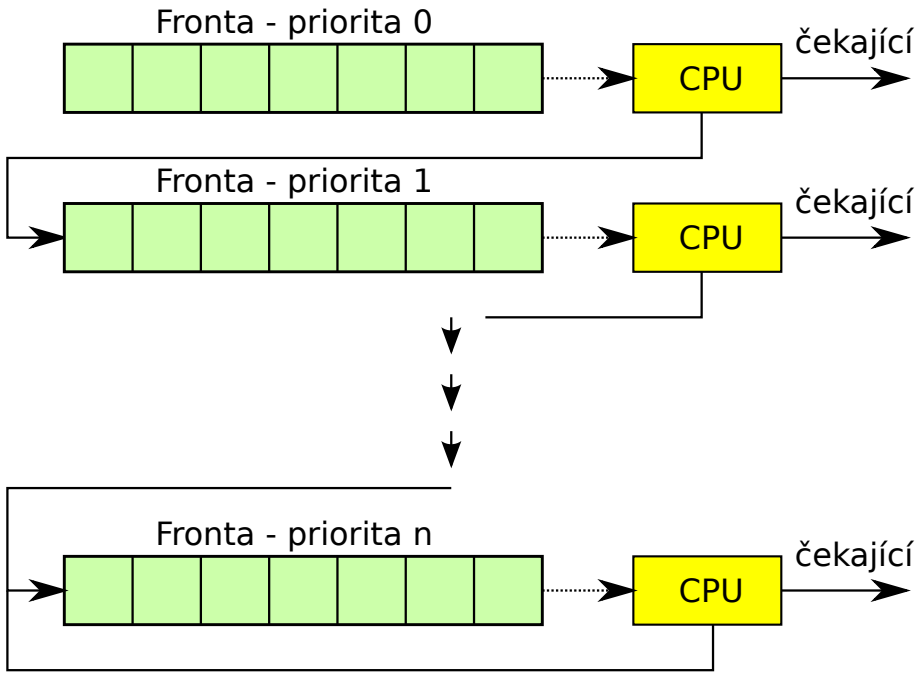
- Efektivita silně závisí na velikosti kvanta
- Veliké kvantum – blíží se chování FCFS
  - Procesy dokončí svoji CPU dávku dříve, než jim vyprší kvantum.
- Malé kvantum – časté přepínání kontextu
  - značná režie
- Dosahuje se průměrné doby obrátky delší oproti plánování SRT
  - Průměrná doba obrátky 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 CFS (Completely Fair Scheduler) v roce 2007
- 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

Earliest Eligible Virtual Deadline First

- Základní nevýhoda CFS přidělování času procesoru je, že existují krátce běžící procesy, které je potřeba spustit velmi rychle, nejlépe do předem určené doby (deadline).
  - Tato skutečnost se v CFS řešila různými záplatami, které se snažili zajistit předbíhání vybraných procesů.
- Motivací pro tento algoritmus je algoritmus dlouho používaný pro systémy reálného času, který vybírá proces s nejbližším deadlinem (Earliest Deadline First)
  - V systémech reálného čase se plánuje pro periodické procesy, pro které se zadává deadline, do kdy musí výsledek spočítat.
- V Linux od verze 6.6 je EEVDF hlavním plánovačem od roku 2023

Earliest Eligible Virtual Deadline First

- Podobně jako CFS počítá pro každý proces rozdíl kolik času by měl proces využít mínus kolik času skutečně využil
  - k výpočtu kolik času by měl proces využít se používá virtuální čas, který zohledňuje počet připravených procesů (podobně jako CFS)
  - tento čas se nazývá lag
  - proces, který má záporný lag není způsobilý (eligible) pro běh
  - pro každý proces se počítá, kdy v budoucnu bude jeho lag nulový a tato doba se nazývá eligible time.
- Virtuální deadline procesu se spočte tak, že se k eligible time přičte velikost posledního přiděleného času
  - procesy které jsou citlivé na zpoždění (latency) mají malé přidělené časy, ostatní procesy mají větší časy
  - díky tomu jsou automaticky preferované procesy s menšími přidělenými časy, protože jejich virtuální deadline je menší
- Žádný proces nemůže zestárnout, všechny procesy mají vypočtený eligible time a po této době se dostanou k běhu.

Kvíz

Nejhorší vlastnost plánovače je, že v něm může proces stárnout. Který z následujících plánovačů má problém se stárnutím procesů?

A - Víceúrovňové zpětnovazební fronty - Multilevel feedback queue

B - Cyklický plánovač - Round robin

C - O(1) plánovač

D - Zcela férový plánovač - Completely Fair Scheduler

Plánování v multiprocesorech

Přiřazování procesů (vláken) procesorům:

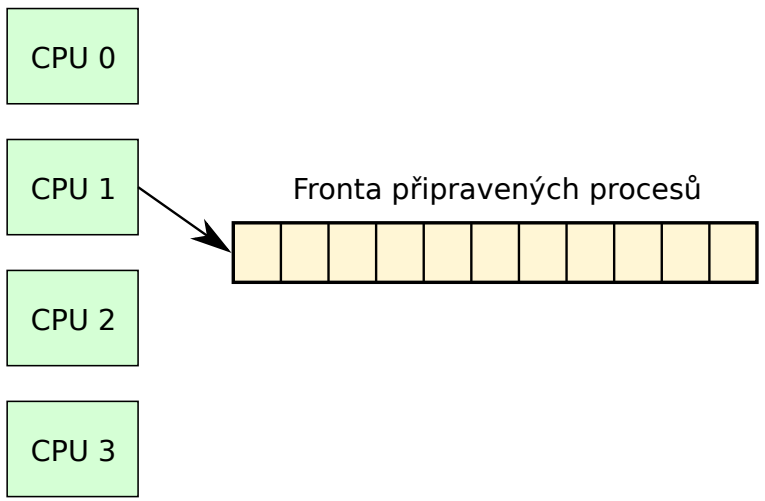
- Architektura „master/slave“
  - Klíčové funkce jádra běží vždy na jednom konkrétním procesoru
  - Master odpovídá za plánování
  - Slave žádá o služby mastera
  - Nevýhoda: dedikace
    - Přetížený master se stává úzkým místem systému
- Symetrický multiprocesing (SMP)
  - Všechny procesory jsou si 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:

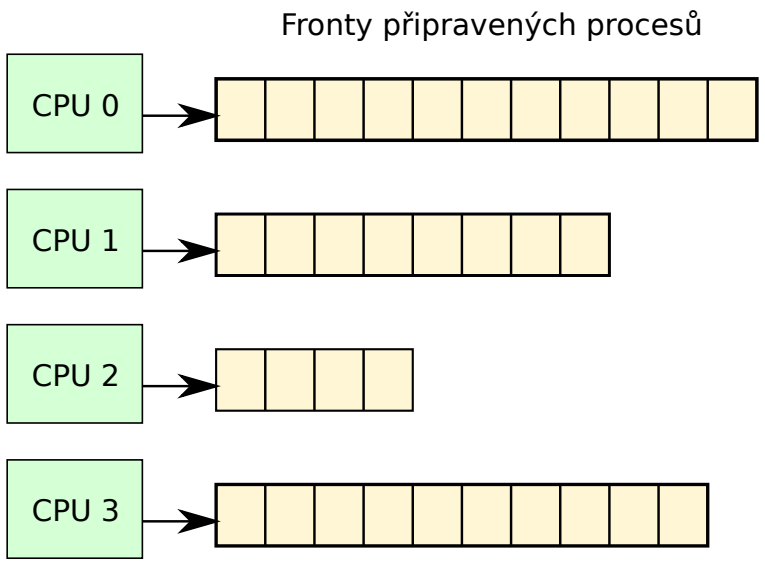
- Jedna společná fronta pro všechny procesory
  - Fronta může být víceúrovňová dle priorit
  - Problémy:
    - Jedna centrální fronta připravených sledů vyžaduje používání vzájemného vylučování v jádře
    - Kritické místo v okamžiku, kdy si hledá práci více procesorů
    - Předběhnutá (přerušená) vlákna 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