KIV/ZOS 2003/2004
Přednáška 7

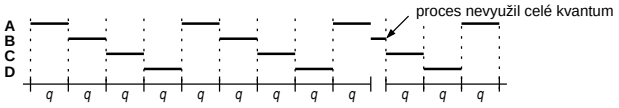
Plánování procesů v interaktivních systémech

Dále si uvedeme základní mechanismy plánování procesů v interaktivních systémech. Uvedené algoritmy mohou být použity také pro plánovač přidělení procesoru v dávkových systémech.

* základní problém - každý proces je jedinečný a nepredikovatelný, nedá se říci, jak dlouho poběží než se zablokuje např. při I/O, nad semaforem apod.
* aby proces neběžel příliš dlouho, má počítač vestavěné hodiny, které provádějí pravidelně přerušení (”tiky časovače”, clock ticks)
* při přerušení se vyvolá obslužný podprogram přerušení v jádře
* OS rozhodne, zda dovolí procesu pokračovat nebo zda dá CPU jinému procesu (preemptivní plánování).

Algoritmus cyklické obsluhy

* angl. Round Robin (RR)
* jeden z nejstarších a nejpoužívanějších algoritmů (v různých variantách)
* každému procesu přiřazen časový interval = časové kvantum, po které může běžet
* pokud běží ještě na konci kvanta, je provedena preempce a je naplánován a spuštěn další připravený proces
* pokud proces skončil nebo se zablokoval ještě před spotřebováním časového kvanta, je také naplánován a spuštěn další připravený proces



Poznámka (obslužný podprogram přerušení)

Přerušení od časovače nastává typicky 50 až 100x za sekundu. Při přerušení je vyvolán obslužný podprogram v jádře - ten má v systému několik funkcí:

- nastavuje interní časovače systému,
- shromažďuje statistiky systému (např. kolik času CPU využíval který proces)
- po uplynutí časového kvanta (resp. v případě potřeby) zavolá plánovač

Časovač tedy může proces v průběhu jeho časového kvanta přerušit vícekrát. Například nastává-li přerušení od časovače 100x za sekundu, je čas mezi přerušeními 10 ms. Při délce kvanta 50 ms bychom prováděli přeplánování každý pátý tik.

[]

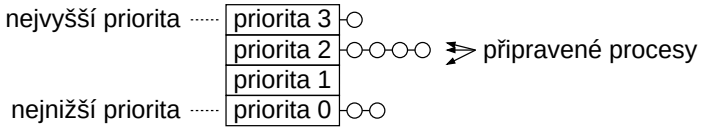
* jednoduchá implementace plánovače
 - plánovač udržuje seznam připravených procesů
 - po vypršení kvanta/zablokování na další se vybere další připravený proces v seznamu

Otázka - jaká je vhodná délka časového kvanta?

* krátké:
 - problém - přepnutí mezi procesy trvá nějakou dobu (uložení a načtení registrů, přemapování paměti apod.)
 - např. přepnutí kontextu 1 ms, kvantum 4 ms => 20% režie

```
* dlouhé:
 - vyšší efektivita; např. 1 s => režie pouze 1%
 - navíc pokud je kvantum delší než průměrná doba držení CPU procesem,
   preempce je třeba zřídka
 - problém - např. pokud 10 uživatelů stiskne klávesu, poslední proces by
   mohl přijít na řadu až za 10 s
* závěr:
 - krátké časové kvantum snižuje efektivitu (vysoká režie)
 - dlouhé může zhoršovat dobu odpovědi na interaktivní požadavky
 - proto volíme kompromis
 - pro algoritmus cyklické obsluhy obvykle 20 až 50 ms
 - kvantum nemusí být konstantní (může se měnit podle zatížení systému)
 - pro algoritmy které se umějí lépe vypořádat s interaktivními
   požadavky než RR bývá kvantum delší, např. 100 ms

Problém algoritmu cyklické obsluhy:

* pokud jsou v systému jak výpočetně vázané tak I/O vázané úlohy, výpočetně
  vázané spotřebují kvantum většinou, celé zatímco I/O vázané spotřebují pouze
  malou část svého časového kvanta a zablokují se
* výpočetně vázané tak získají nespravedlivě vysokou část času procesoru
* (Haldar & Subramanian 1991) navrhli vylepšení - nazývají VRR (Virtual RR)
 - procesy po dokončení I/O mají přednost před ostatními

Prioritní plánování
...................

* algoritmus RR předpokládal, že všechny procesy jsou stejně důležité
* to ale nemusí být pravda, např.:
 - na superpočítači mohu dát prioritu zákazníkům, kteří si zaplatí více
 - interaktivní procesy mohou mít vyšší prioritu než procesy běžící na
   pozadí (viz uvedený příklad - interaktivní práce vs. odesílání pošty)

Z výše uvedených dvou příkladů vidíme, že priorita může být procesům
přiřazena staticky nebo dynamicky:

* staticky - např. při startu procesu; důležité procesy můžeme spouštět
             s vyšší statickou prioritou (v Linuxu příkazem nice(1))
* dynamicky - např. pokud mají I/O-vázané procesy vyšší prioritu,
              budou se moci po krátkém použití CPU opět zablokovat.

Obvykle má priorita statickou složku (určenou při startu procesu) a
dynamickou složku (určenou chováním procesu v poslední době); jako
výsledná priorita se chápe součet statické a dynamické priority.

Poznámka:

 Kdyby byla priorita určena pouze staticky a kdyby bylo plánování jen
 podle priorit, běžely by pouze připravené procesy s nejvyšší
 prioritou; proto např. plánovač snižuje dynamickou prioritu běžícího
 procesu při každém tiku časovače; pokud priorita procesu klesne pod
 prioritu jiného procesu, nastane přeplánování.

 V kvantově orientovaných plánovacích algoritmech je nejjednodušší
 přiřadit procesům dynamickou prioritu 1/f, kde f je velikost části
 kvanta, kterou proces naposledy využil. Tím se zvýhodní I/O-vázané
 oproti CPU-vázaným procesům.

[]

Často je výhodné spojit cyklické a prioritní plánování:

* procesy rozdělíme do prioritních tříd - v každé třídě jsou procesy se
  stejnou prioritou
* prioritní plánování mezi třídami
* cyklická obsluha uvnitř třídy, obsluhovány jsou pouze připravené
  procesy v nejvyšší neprázdné prioritní třídě

Příklad:
```

* 4 prioritní třídy, 0 .. 3



* dokud jsou procesy v prioritní třídě 3, spusť cyklicky každý na 1 kvantum
* pokud je prioritní třída 3 prázdná, totéž pro prioritní třídu 2 atd.
* jednou za čas se priority přepočítají, procesům které využívaly CPU se sníží priorita

[]

Výše uvedený algoritmus s dynamickým přiřazováním priority (podle využívání času procesoru v poslední době, priorita procesu se snižuje při jeho běhu a zvyšuje při jeho nečinnosti) a s cyklickým střídáním procesů v nejvyšší prioritními třídě se používá například v systémech typu UNIX (30-50 prioritních tříd). Popis plánovacího algoritmu původního UNIXu lze nalézt např. v (Bach 1986).

Plánovač spravedlivého sdílení

* např. v algoritmu RR je přidělován čas každému procesu nezávisle
* pokud má uživatel více procesů, dostane celkově více času

* ”spravedlivé sdílení” - přidělovat čas každému uživateli (nebo jinak definované skupině procesů) proporcionálně bez ohledu na to, kolik má procesů
* například pokud je přihlášeno N uživatelů, každý dostane přibližně 1/N času

* implementace jednoduchá:
 - pro každého uživatele zavedeme novou položku - ”priorita skupiny spravedlivého sdílení”
 - obsah položky se započítává do priority každého procesu příslušného uživatele
 - hodnota položky průběžně odráží poslední využití procesoru všemi procesy uživatele

Poznámka (jak určit prioritu skupiny spravedlivého sdílení)

Jako příklad zde uvedu jednu možnou implementaci.

Pro každého uživatele zavedeme novou položku g. Obslužný podprogram přerušení zvětší při každém tiku časovače o 1 položku g toho uživatele, jehož proces právě využívá procesor. Aby položka g odrážela poslední využití času procesoru, je jednou za sekundu proveden rozklad: g:=g/2.

Pokud procesy uživatele využívaly v poslední době procesor (tj. mají vysokou hodnotu pole g), jejich priorita má být nízká, proto bude hodnota g prioritu procesu snižovat: P(p,g)=p-g.

[]

Plánování pomocí loterie

* angl. Lottery Scheduling (Waldspurger & Weihl, 1994)
* cílem poskytnout procesům příslušnou (volitelnou) proporci času CPU

* základní myšlenka:
 - procesy obdrží tickety (česky losy)

```
 - plánovač vybere náhodně jeden ticket
 - vítězný proces obdrží cenu - 1 kvantum času na CPU
 - důležitější procesy mohou obdržet více ticketů, aby se zvýšila
   jejich šance na výhru

Např. pokud je celkově 100 losů a proces jich má 20, má 20% šanci na
výhru, tj. v dlouhodobém průměru dostane 20% času CPU.

Plánování pomocí loterie umožňuje řešit problémy, jejichž řešení je v
jiných plánovacích algoritmech obtížné:

* spolupracující procesy si mohou předávat losy
 - např. pokud klient posílá zprávu serveru a blokuje se, může
   serveru propůjčit všechny své tickety
 - po vykonání požadavku server tickety vrátí
 - pokud nejsou požadavky, server žádné tickety nepotřebuje

* dobře využitelný pokud potřebujeme rozdělit čas procesoru mezi
  procesy v určitém poměru
 - to neplatí u prioritního plánování - ve skutečnosti nevíme, co bude
   znamenat přidělení priority 30
 - jakmile je jsou procesu přiděleny tickety, má šanci vyhrát čas na
   CPU - algoritmus dobře reaguje na změny

Plánování pomocí loterie je relativně nový algoritmus, proto ho zatím
používá málo reálných systémů.

Poznámka (charakteristiky algoritmů pro plánování procesů)
..........................................................

algoritmus  rozhodovací mód            prioritní fce    rozhodovací pravidlo
RR          preemptivní (vypršení kvanta) P()=1         cyklicky
prioritní   preemptivní (P jiného > P)    viz text      náhodně nebo cyklicky
spravedlivé preemptivní (P jiného > P)    P(p,g)=p-g    cyklicky
loterie     preemptivní (vypršení kvanta) P()=1         podle výsl. loterie

[]



Příklad skutečného plánovacího algoritmu - plánování v systému Windows 2000:
.....................................................................

* 32 prioritních úrovní, očíslováno od 0 do 31
* implementováno jako pole 32 položek, každá položka obsahuje ukazatel
  na seznam připravených procesů na příslušné úrovni

* plánovací algoritmus prohledává pole od položky 31 do 0
 - pokud nalezne neprázdnou frontu, naplánuje první proces ve frontě a
   nechá ho běžet po dobu 1 kvanta
 - po uplynutí kvanta je proces zařazen na konec fronty na příslušné
   prioritní úrovni

* priority jsou rozděleny do skupin:
 -  0 ..... nulování stránek pro správce paměti
 -  1 .. 15 pro obyčejné procesy
 - 16 .. 31 pro systémové procesy

* 0 - pokud není nic jiného na práci

* obyčejné procesy (1-15)
 - základní (bázová) priorita = minimální prioritní úroveň, může určit
   uživatel např. voláním SetPriorityClass
 - aktuální priorita - pohybuje se mezi bázovou a maximální prioritou 15
 - aktuální priorita se mění se s historií procesu podle následujících
   pravidel:
  . dokončení I/O zvyšuje prioritu o určenou hodnotu (typicky 1 - disk,
    2 - sériový port, 6 - klávesnice, 8 - zvuková karta)
  . vzbuzení po čekání na semafor, mutex apod. zvýší prioritu o 2 pokud
```

```
    je proces na popředí (tj. řídí okno, do kterého je posílán vstup z
    klávesnice), jinak o 1
  . pokud proces využil celé kvantum, sníží se priorita o 1
  . pokud proces neběžel ”dlouhou dobu”, je mu na 2 kvanta zvýšena
    priorita na 15 (aby se zabránilo prioritní inverzi)

* procesy plánovány přísně podle priorit, tj. obyčejné procesy pouze
  pokud není žádný systémový proces připraven

[]
```

```
Poznámka pro zajímavost (plánování na víceprocesorových strojích)
..................................................................

 V praxi se nejspíše setkáte s těsně vázanými symetrickými multiprocesory -
 tj. se systémy se společnou hlavní pamětí, kde jsou si procesory rovné.

 V takových systémech musí plánovací algoritmus řešit navíc následující
 problémy:

 * přiřazení procesů procesorům - jedním extrémem je permanentní přiřazení,
   druhým společná fronta připravených procesů plánovaných na kterýkoli volný
   procesor
  - permanentní přiřazení má menší režii, ale některé CPU mohou být nevyužity
  - v praxi často afinita procesu k procesoru, na kterém běžel naposledy
  - někdy může být nutné přiřadit procesoru jediný proces (RT procesy)
 * plánování vláken - některé paralelní aplikace mají podstatně vyšší
   výkonnost, pokud jejich vlákna běží současně (zkrátí se vzájemné čekání
   vláken apod.)
  - v současnosti předmět intenzivního výzkumu v OS

[]

Poznámka pro zajímavost (plánování v systémech reálného času)

 * pro připomenutí:
  - RT procesy buď řídí nebo reagují na události nastávající ve vnějším světě
  - správnost závisí nejen na výsledku, ale i na čase, ve kterém je výsledek
    vyprodukován
  - s každou ”podúlohou” procesu je obvykle možné sdružit deadline, tj. čas
    ve kterém musí být podúloha spuštěna nebo dokončena
  - hard RT = času musí být dosaženo, soft RT = dosažení deadline je žádoucí

 * podúlohy procesu, resp. události na které úlohy reagují, mohou být
   aperiodické (nastávají nepredikovatelně) nebo periodické (v pravidelných
   časových intervalech)
 * v závislosti na množství času potřebném pro zpracování události nemusí být
   možné všechny včas zpracovat; pokud to možné je, říkáme že systém je
   plánovatelný (schedulable)
 * plánovací algoritmy v RT systémech jsou statické nebo dynamické:
  - statické - plánovací rozhodnutí se provede před spuštěním systému, to
    předpokládá že je předem znám dostatek informací o vlastnostech procesů
  - dynamické - za běhu; některé algoritmy provedou analýzu plánovatelnosti,
    nový proces je přijat pouze pokud je výsledek plánovatelný

 * typické vlastnosti současných RT systémů:
  - malá velikost OS (s tím souvisí omezená funkčnost)
  - snaha spustit RT proces co nejrychleji, s tím souvisí:
   . rychlé přepínání mezi procesy nebo vlákny
   . rychlá obsluha přerušení + minimalizace intervalů, kdy je přerušení
     zakázáno
  - multitasking + meziprocesová komunikace (semafory, signály, události...)
  - primitiva pro zdržení procesu o zadaný čas, čítače časových intervalů
  - někdy rychlé sekvenční soubory (viz budoucí přednáška o souborech)
```

```
Poznámka (plánování procesů a plánování vláken)
...............................................

* plánování procesů je vždy součástí OS

* naproti tomu plánování vláken - 2 možné způsoby implementace:
 - běh vláken plánuje OS (kernel-level threads)
 - plánování vláken je součástí uživatelského procesu, OS o existenci
   vláken nic neví (user-level threads)
```

* jsou-li vlákna plánována OS, obvykle se používají stejné mechanismy a algoritmy jako pro plánování procesů (často jsou vlákna plánována bez ohledu na to, kterému procesu patří - např. má-li proces 10 vláken, každé vlákno obdrží časové kvantum)

* jsou-li vlákna plánována uvnitř procesu
 - vlákna běží v rámci času, který je přidělen procesu
 - přepínání mezi vlákny implementuje obvykle systémová knihovna
 - pokud OS neumí poskytovat procesu pravidelné ”přerušení”, je možné pouze nepreemptivní plánování
 - obvykle se používá algoritmus RR nebo prioritní plánování
 - má menší režii oproti kernel-level threads, ale také menší možnosti

Příklad - ve Windows 2000 a v Linuxu jsou vlákna plánována jádrem, některé varianty systému UNIX používají user-level threads.

[]

Uvíznutí (deadlock)

* představme si ”naivní” večeřící filozofy - vezmou pravou vidličku, ale nemohou vzít levou (protože tato je již obsazena sousedem)
* česky nejčastěji název uvíznutí (deadlock), někteří používají název ”zablokování” (Mrázek)

* nejčastější příklad v učebnicích: výhradní alokace I/O zařízení
 - například mějme vypalovačku CD (nazveme R jako recorder) a scanner (S)
 - dva procesy A a B chtějí nascanovat dokument a zapsat na CD ROM
 - A žádá R a dostane, B žádá S a dostane
 - A žádá S a čeká, B žádá R a čeká => uvíznutí
* také zamykání záznamů v databázi:
 - dva procesy A a B požadují přístup k záznamům R a S v databázi
 - A zamkne R, B zamkne S, pak ...
* totéž se může stát se semafory: P(sem) jako alokace zdroje, V(sem) jako uvolnění
 - mějme dva semafory R a S, R=1 a S=1
 - A provede P(R), B provede P(S), pak ...

Poznámka (sériově využitelné a konzumovatelné zdroje)

* vše výše uvedené jsou ”sériově využitelné zdroje”, tj. proces zdroj alokuje, používá, uvolní
* poněkud odlišný případ nastává, pokud procesy očekávají (požadují) ”konzumovatelné zdroje”, např. zprávy, které může produkovat jiný proces (situace typu producent/konzument)
* také může nastat uvíznutí
* příklad: mějme dva procesy A a B komunikující pomocí zpráv:
 - proces A: .... receive(B, R); send(B, S); ....
 - proces B: .... receive(A, S); send(A, R); ....
* z časových důvodů budeme hovořit pouze o sériově využitelných zdrojích

[]

Poznámka (více zdrojů stejného typu)

 * některé zdroje existují ve více exemplářích, proces žádá o zdroj daného typu - je jedno, který dostane
 * nejčastější příklad bloky disku pro soubor, paměť apod.
 * například mějme celkově 5 zdrojů, a dva procesy A a B
  . proces A požádá o dva zdroje, dostane (zbudou 3)
  . proces B požádá o dva zdroje, dostane (zbude 1)
  . proces A požádá o další dva, nejsou (je pouze 1), čeká
  . proces B požádá o dva, nejsou, čeká, tj. nastalo uvíznutí
 * z časových důvodů budeme se zaměříme převážně na případ, kdy existuje pouze jeden zdroj každého typu (například procesy žádají o

konkrétní tiskárnu a není jedno, která se přiřadí)

[]

* zařízení, záznam, atd. budeme nazývat obecným termínem ”zdroj”

Uvíznutí (def.):

V množině procesů nastalo uvíznutí, jestliže každý proces množiny čeká na událost, kterou může způsobit pouze jiný proces množiny.

* protože všechny čekají, žádný z nich událost nevygeneruje, nevzbudí jiný proces

Podmínky vzniku uvíznutí (Coffman et al. 1971)
.....

Hovoří o ”zdrojích” (zdrojích systému), protože na nich se to nejlépe ukazuje.

1. vzájemné vyloučení: každý zdroj je buď dostupný nebo je výhradně přiřazen právě jednomu procesu
2. ”hold and wait”: proces držící výhradně přiřazené zdroje může požadovat další zdroje
3. nemožnost odejmutí: jednou přiřazené zdroje nemohou být procesu násilně odejmuty (proces je musí sám uvolnit)
4. cyklické čekání: musí být cyklický řetězec 2 nebo více procesů, kde každý z nich čeká na zdroj držený dalším členem

* pro vznik uvíznutí musejí být splněny všechny 4 podmínky - podmínky 1. až 3. jsou předpoklady, za kterých lze definovat 4. podmínku, téměř totožnou s definicí uvíznutí
* jinými slovy, pokud jedna z nich není splněna, uvíznutí nemůže nastat

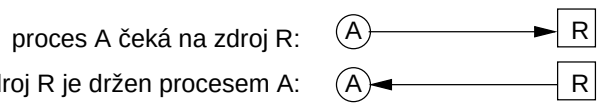
Poznámka:

Podmínky 1 a 3 platí např. pro první uvedený příklad. Na CD může v jednu chvíli zapisovat pouze 1 proces (podmínka 1) a CD recorder není možné zapisujícímu procesu odejmout.

[]

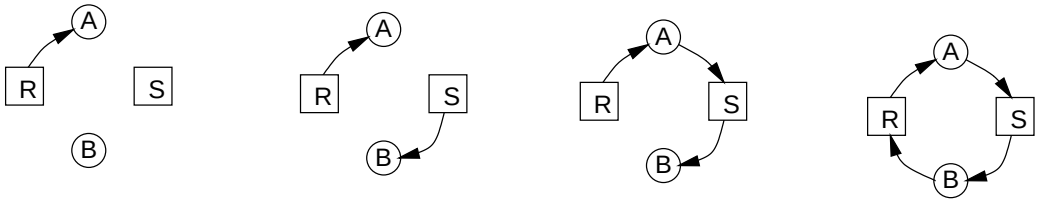
Modelování uvíznutí
.....

* tyto 4 podmínky mohou být modelovány pomocí orientovaných grafů = graf alokace zdrojů
* graf má 2 typy uzlů
  - proces - zobrazujeme jako kruh O
  - zdroj - zobrazujeme jako čtverec []
* význam hran
  - hrana od zdroje k procesu: zdroj držen procesem
  - hrana od procesu ke zdroji: proces je blokován čekáním na zdroj



Příklad:

* CD (R), scanner (S), dva procesy A a B:
  - A žádá R a dostane, B žádá S a dostane
  - A žádá S a čeká, B žádá R a čeká => uvíznutí
* cyklus v grafu znamená, že je uvíznutí (uvíznutí se týká procesů a zdrojů v cyklu)



Poznámka:

Za výše uvedených podmínek je cyklus v grafu nutnou a postačující podmínkou pro vznik uvíznutí.

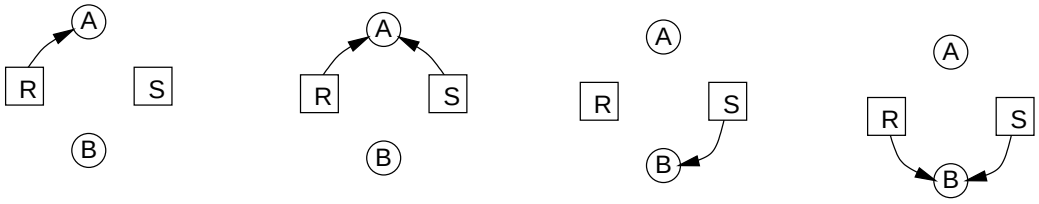
[]

Poznámka (vznik uvíznutí závisí na pořadí vykonání instrukcí procesů)

Pokud by plánovač způsobil, že by byly nejprve vykonány všechny (pro nás zajímavé) instrukce alokace a uvolnění zdrojů procesu A a potom instrukce procesu B, uvíznutí nenastane:

* například dva procesy A a B žádají zdroje R a S
 - A žádá R a S a oba dostane, A oba zdroje uvolní
 - B žádá S a R a oba dostane, B oba zdroje uvolní
 - uvíznutí nenastalo

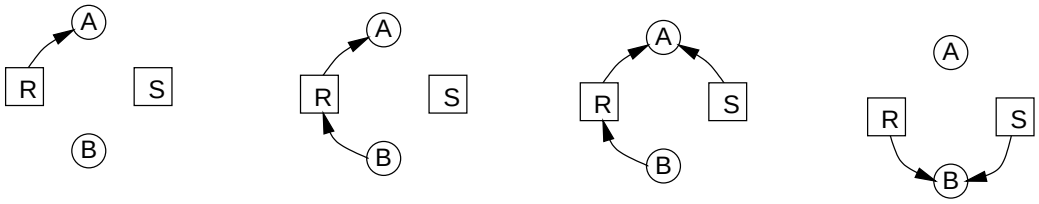
Tj. při některých bězích výše zmíněných dvou procesů nemusí uvíznutí nastat (o to hůře se nám bude hledat případná chyba).



[]

Poznámka (vznik uvíznutí závisí na pořadí alokace zdrojů)

* pokud bychom napsali procesy A a B tak, aby oba žádaly o zdroje R a S ve stejném pořadí (např. napřed R a pak S), uvíznutí nenastane:
 - A žádá R a dostane, B žádá R a čeká
 - A žádá S a dostane, A uvolní R a S
 - B čekal na R a dostane, B žádá S a dostane
 - uvíznutí nemohlo nastat



[]

❄