

07. Plánování procesů Deadlock

ZOS 2014, L. Pešička



Plánování procesů

- v **dávkových** systémech
 - v **interaktivních** systémech
 - Příklad – Windows 2000 (NT/XP/Vista/7)
 - Ve **víceprocesorových** systémech
 - V systémech **reálného času**
 - Plánování procesů x plánování vláken
-



Plánování procesů v interaktivních systémech

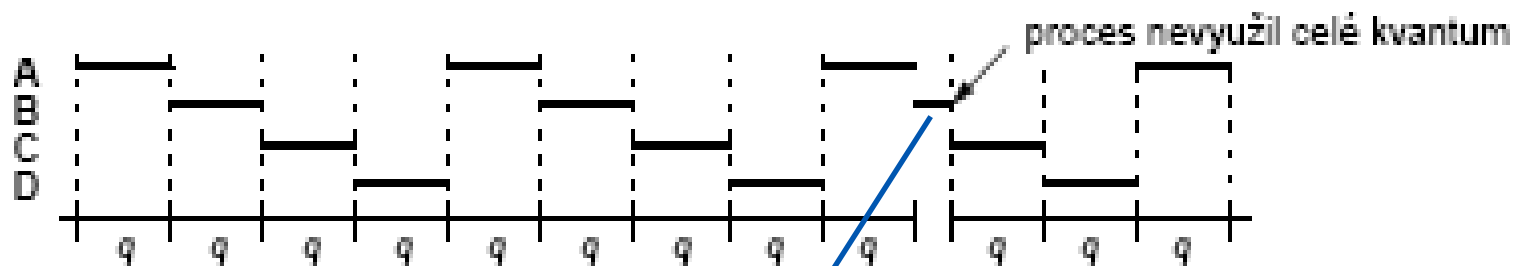
- potřeba docílit, aby proces neběžel „příliš dlouho“
 - možnost obsloužit další procesy
- každý proces – jedinečný a nepredikovatelný
 - nelze říct, jak dlouho poběží, než se zablokuje (nad I/O, semaforem, ...) – jak dlouhý bude CPU burst
- vestavěný systémový časovač v počítači
 - provádí pravidelně přerušení (tiky časovače, clock ticks)
 - vyvolá se obslužný podprogram v jádře
 - rozhodnutí, zda proces bude pokračovat, nebo se spustí jiný (preemptivní plánování) – po několika tikách časovače



Algoritmus cyklické obsluhy – Round Robin (RR)

- jeden z nejstarších a nejpoužívanějších
- každému procesu přiřazen **časový interval**
 - **časové kvantum**, po které může běžet
- proces běží na konci kvanta
 - **preemce**, naplánován a spuštěn další připravený proces
- proces skončí nebo se zablokuje před uplynutím kvanta
 - **stejná akce** jako v předchozím bodě ☺

Round Robin



Pokud proces nevyužije celé časové kvantum,
okamžitě se naplánuje další proces, na nic se nečeká
(je třeba max. využít procesor)

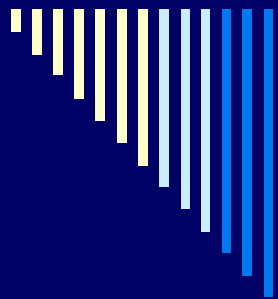


Round Robin

- jednoduchá implementace plánovače
 - plánovač udržuje seznam připravených procesů
 - Při **vypršení kvanta** nebo **zablokování**
→ vybere další proces

Procesu je
nedobrovolně
odebrán procesor,
přejde do stavu
připravený

Proces žádá I/O
dobrovolně se
vzdá CPU, přejde
do stavu
blokováný



Obslužný program přerušení časovače

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



1 kvantum – odpovídá více přerušením časovače

- Časovač může proces v průběhu časového kvanta přerušit vícekrát.
- přerušení 100x za sekundu (příklad)
 - 10 ms mezi přerušeními
- pokud kvantum 50 ms
 - přeplánování každý pátý tik



vhodná délka časového kvanta

□ krátké

- přepnutí procesů chvíli trvá (uložení a načtení registrů, přemapování paměti, ...)
- přepnutí kontextu 1ms, kvantum 4ms – 20% velká režie

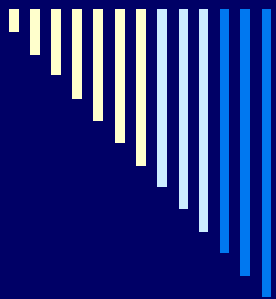
□ dlouhé

- vyšší efektivita; kvantum 1s – menší režie
- pokud kvantum delší než průměrná doba držení CPU procesem – preempce je třeba zřídka
- problém interaktivních procesů – 10 uživatelů stiskne klávesu, odezva posledního procesu až 10s



vhodná délka kvanta - shrnutí

- ❑ **krátké** kvantum – snižuje efektivitu (režie)
- ❑ **dlouhé** – zhoršuje dobu odpovědi na interaktivní požadavky
- ❑ kompromis 😊
- ❑ pro algoritmus cyklické obsluhy obvykle 20 až 50 ms
- ❑ kvantum **nemusí** být **konstantní**
 - změna podle zatížení systému
- ❑ pro algoritmy, které se lépe vypořádají s interaktivními požadavky lze kvantum delší – 100 ms



Problém s algoritmem cyklické obsluhy

- v systému výpočetně vázané i I/O vázané úlohy
- **výpočetně** vázané – většinou kvantum spotřebují
- **I/O** vázané – pouze malá část kvanta se využije a zablokují se
- **výpočetně** vázané – získají nespravedlivě vysokou část času CPU

- **modifikace VRR** (Virtual RR, 1991)
 - procesy po dokončení **I/O** mají **přednost** před ostatními



Prioritní plánování

- předpoklad RR: všechny procesy stejně důležité
 - ale:
 - vyšší priorita zákazníkům, kteří si „připlatí“
 - interaktivní procesy vs. procesy běžící na pozadí (odesílání pošty)
 - prioritu lze přiřadit staticky nebo dynamicky:
 - **staticky**
 - při startu procesu, např. Linux – nice
 - **dynamicky**
 - přiřadit I/O větší prioritu, použití CPU a zablokování
-



Priorita

priorita = statická + dynamická

- obsahuje obě složky – výsledná jejich součtem
 - **statická** (při startu procesu)
 - **dynamická** (chování procesu v poslední době)
-
- kdyby pouze statická složka a plánování jen podle priorit – běží pouze připravené s nejvyšší prioritou
 - plánovač **snižuje dynamickou** prioritu běžícího procesu při každém tiku časovače; klesne pod prioritu jiného - **přeplánování**



Dynamická priorita

- V kvantově orientovaných plánovacích algoritmech:
- dynamická priorita např. dle vzorce $1 / f$ (!)
- f – velikost části kvanta, kterou proces naposledy použil
- zvýhodní I/O vázané x CPU vázaným

Pokud proces nevyužil celé kvantum, jeho dynamická priorita se zvyšuje, např. pokud využil posledně jen 0.5 kvanta, tak $1/0,5 = 2$, pokud celé kvantum využil $1/1=1$



Spojení cyklického a prioritního plánování

- **prioritní třídy**
 - v každé třídě procesy se stejnou prioritou
- **prioritní plánování** mezi třídami
 - Bude obsluhována třída s nejvyšší prioritou
- **cyklická obsluha** uvnitř třídy
 - V rámci dané třídy se procesy cyklicky střídají
- obsluhovány jsou pouze připravené procesy v nejvyšší neprázdné prioritní třídě

A kdy se dostane na další fronty?

Prioritní třídy

Máme zde
priority, třídy i časová kvanta



4 prioritní třídy

dokud procesy v třídě 3 – spustit **cyklicky** každý na **1 kvantum**

pokud třída 3 prázdná – obsluhujeme třídu 2

(prázdná => žádný proces danou prioritu nemá, nebo je ve stavu blokováný, čeká např. na I/O)

jednou za čas – přepočítání priorit

procesům, které využívaly CPU se sníží priorita



Prioritní třídy

- **dynamické přiřazování priority**
 - dle využití CPU v poslední době
 - priorita procesu
 - snižuje se při běhu
 - zvyšuje při nečinnosti
- **cyklické střídání** procesů
- OS typu Unix
 - Mají 30 až 50 prioritních tříd



Plánovač spravedlivého sdílení

□ problém:

- čas přidělován každému procesu nezávisle
- Pokud uživatel má více procesů než jiný uživatel
-> dostane více času celkově

□ spravedlivé sdílení

- přidělovat čas každému **uživateli** (či jinak definované skupině procesů) **proporcionálně**, bez ohledu na to, kolik má procesů
- máme-li N uživatelů, každý dostane $1/N$ času

= spravedlnost vůči uživateli



Spravedlivé sdílení

- nová položka **priorita skupiny spravedlivého plánování**
 - Zavedena pro každého uživatele
- obsah položky
 - započítává se do priority každého procesu uživatele
 - odráží poslední využití procesoru všemi procesy daného uživatele

Má-li uživatel Pepa procesy p1, p2, p3
a pokud proces p3 bude využívat CPU hodně často, budou touto
položkou penalizovány i další procesy uživatele Pepa



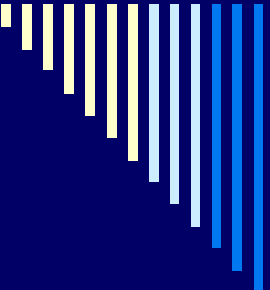
Spravedlivé sdílení - implementace

- každý uživatel – položka g
- obsluha přerušení – inkrementuje g uživatele, kterému patří právě běžící proces
- jednou za sekundu rozklad: $g=g/2$
 - Aby odrážel chování **v poslední době**, vzdálená minulost nás nezajímá
- priorita $P(p,g) = p - g$
- pokud procesy uživatele využívaly CPU v poslední době – položka g je vysoká, vysoká penalizace



Plánování pomocí loterie

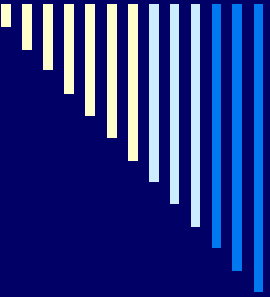
- Lottery Scheduling (Waldspurger & Weihl, 1994)
- cílem – poskytnout procesům příslušnou proporcí času CPU
- základní princip:
 - procesy obdrží **tikety (losy)**
 - plánovač **vybere náhodně** jeden tiket
 - **vítězný** proces obdrží cenu – 1 **kvantum** času CPU
 - důležitější procesy – **více tiketů**, aby se zvýšila šance na výhru (celkem 100 losů, proces má 20 – v dlouhodobém průměru dostane 20% času)



Loterie - výhody

řešení problémů, v jiných plán. algoritmech obtížné

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



Loterie - výhody

- rozdělení času mezi procesy **v určitém poměru**
 - to bychom těžko realizovali u prioritního plánování, co znamená, že má proces prioritu např. 30?
 - proces – tickety – šance vyhrát

- zatím spíše experimentální algoritmus

Shrnutí

Algoritmus	Rozhodovací mód	Prioritní funkce	Rozhodovací pravidlo
RR	Preemptivní vyprší kvantum	$P() = 1$	cyklicky
prioritní	Preemptivní $P_{\text{jiný}} > P$	Viz text	Náhodně, cyklicky
spravedlivé	Preemptivní $P_{\text{jiný}} > P$	$P(p,g)=p-g$	cyklicky
Ioterie	Preemptivní vyprš. kv.	$P() = 1$	Dle výsledku Ioterie



Příklad – Windows 2000/XP/...

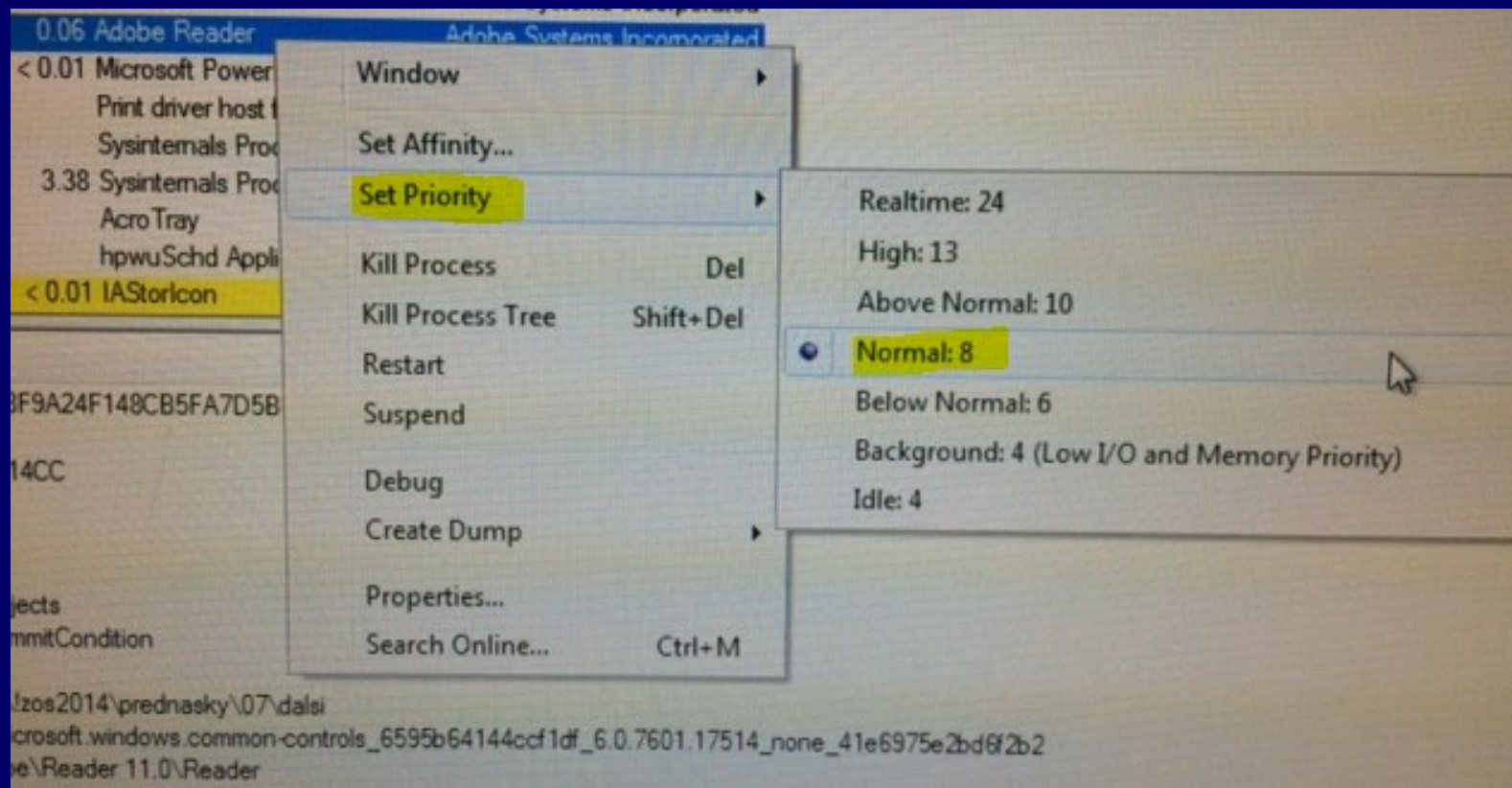
- 32 prioritních úrovní, 0 až 31 (nejvyšší)
- pole 32 položek
 - každá položka – ukazatel na seznam připravených procesů
- plánovací algoritmus – prohledává pole od 31 po 0
 - nalezne neprázdnou frontu
 - naplánuje první proces, nechá ho běžet 1 kvantum
 - po uplynutí kvanta – proces na konec fronty na příslušné prioritní úrovni



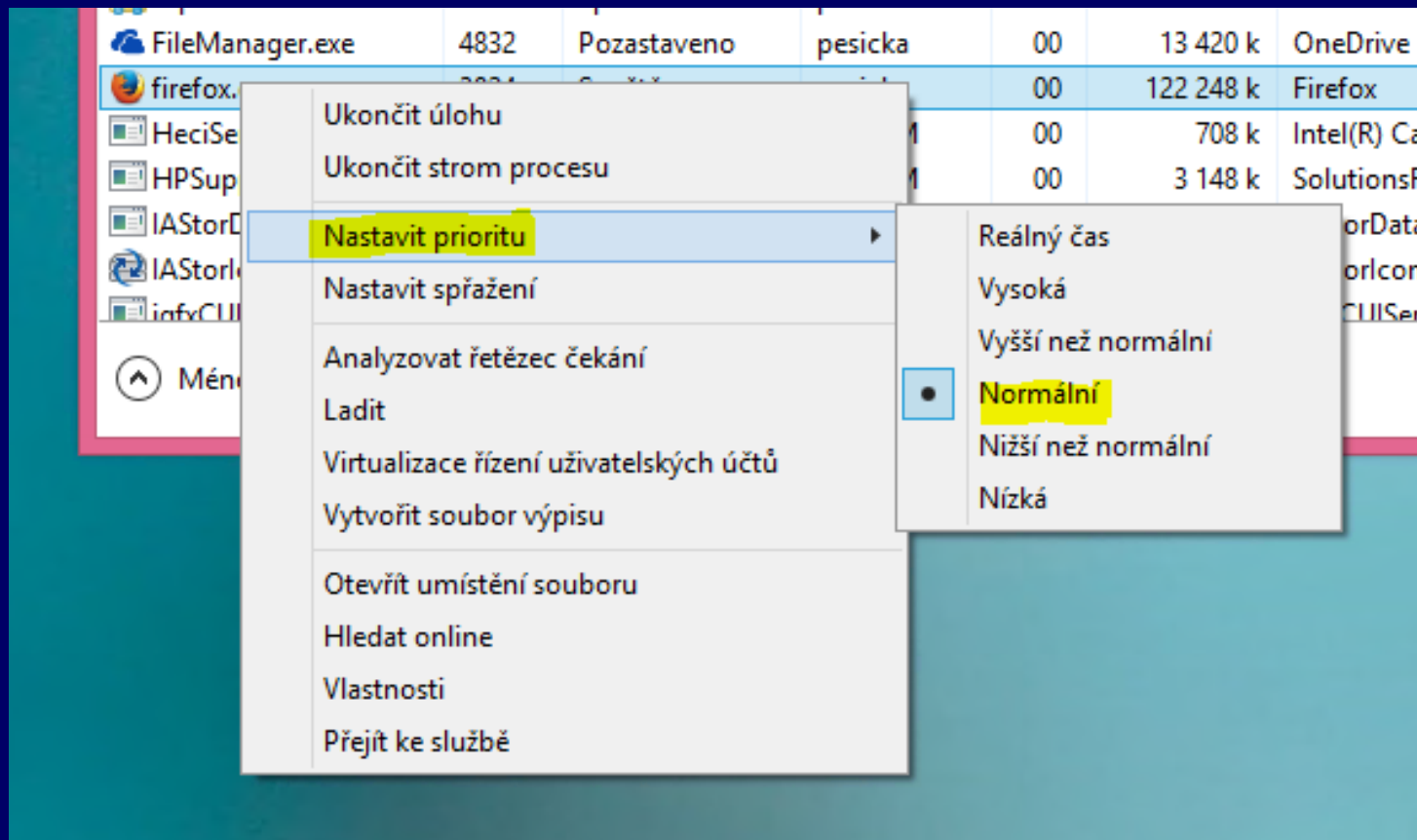
Windows – skupiny priorit

priorita	popis
0	Nulování stránek pro správce paměti
1 .. 15	Obyčejné procesy
16 .. 31	Systémové procesy

Process explorer ze sysinternals sady programů



Windows 8 správce úloh





Windows - priority

- 0 .. pokud není nic jiného na práci
- 1 .. 15 – obyčejné procesy
- aktuální priorita – <bázová, 15>
- базová priorita – základní, může ji určit uživatel voláním *SetPriorityClass*
- aktuální priorita se mění – viz dále
- procesy se plánují přísně podle priorit, tj. obyčejné pouze pokud není žádný systémový proces připraven



Windows – změna akt. priority

- dokončení I/O zvyšuje prioritu o
 - 1 – disk, 2 – sériový port, 6 – klávesnice, 8 – zvuková karta
- vzbuzení po čekání na semafor, mutex zvýší o
 - 2 - pokud je proces na popředí
(řídí okno, do kterého je posílán vstup z klávesnice)
 - 1 – jinak
- proces využil celé kvantum
 - sníží se priorita o 1
- proces neběžel dlouhou dobu
 - na 2 kvanta priorita zvýšena na 15 (zabránit inverzi priorit)



Windows – plánování na vláknech

proces A = 10 spustitelných vláken

proces B = 2 spustitelná vlákna

předpokládáme - stejná priorita

každé vlákno cca 1/12 CPU času

NENÍ 50% A, 50% B

nedělí ferově mezi procesy, ale mezi vlákna



Idle threads

- Jeden pro každý CPU
 - „pod prioritou 0“
 - účtování nepoužívaných clock threadů
 - umožní nastavit vhodný power management
 - volá **HAL** (hardware abstraction layer)



Zero page thread

- Jeden pro celý systém
- Běží na úrovni priority 0
- Nuluje nepoužívané stránky paměti

Bezpečnostní opatření, když nějakému procesu přidělíme stránku paměti, aby v ní nezůstali data jiného procesu „z dřívější“, aby se nedostal k informacím, ke kterým se dostat nemá



Kvantum, stretching

□ kvantum stretching

- maximum 6 tiků (3x)
- middle 4 tiky (2x)
- none 2 tiky

- Na **desktopu** je defaultní kvantum **2 ticky** vlákn na popředí – může být stretching
- na **serveru** je kvantum vždy **12 ticků**, není kvantum stretching
- standardní **clock tick** je **10** nebo **15** ms



Zjištění hodnoty časovače

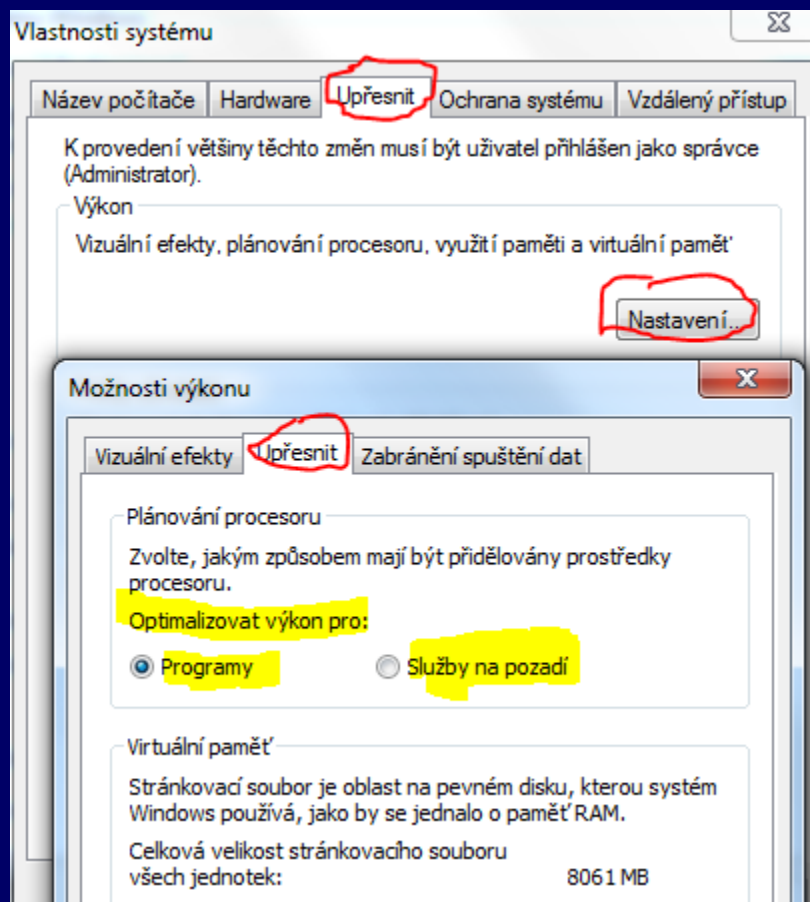
Program **clockres** ze sady Sysinternals

```
C:\!zos2012\prednasky\07\dalsi\SysinternalsSuite>clockres
```

```
ClockRes v2.0 - View the system clock resolution  
Copyright (C) 2009 Mark Russinovich  
SysInternals - www.sysinternals.com
```

```
Maximum timer interval: 15.600 ms  
Minimum timer interval: 0.500 ms  
Current timer interval: 10.000 ms
```

Win 7 – vlastnosti systému – upřesnit – optimalizovat výkon pro



registrový klíč:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\PriorityControl

Win32PrioritySeparation 2

6bitů: XX XX XX

kvantum

- krátké, dlouhé
- proměnné, pevné
- navýšení pro procesy na popředí: 2x, 3x)

viz

<http://technet.microsoft.com/library/Cc976120>



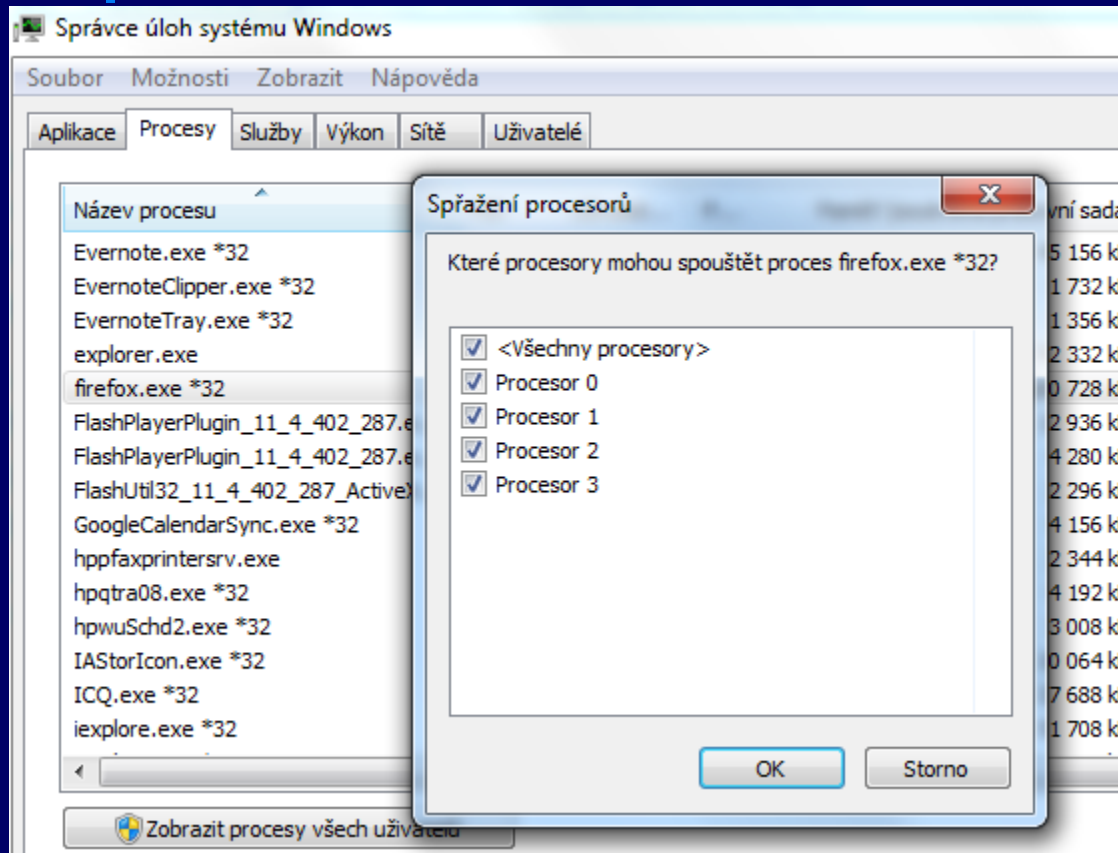
Windows: vlákénka (fibers)

- ❑ kromě vláken i fibers
- ❑ fibers plánuje vlastní aplikace, nikoliv centrální plánovač jádra
- ❑ vytvoření fiberu: **CreateFiber**
- ❑ nepreemptivní plánování – odevzdá řízení jinému vlákénku přes **SwitchToFiber**

příklad:

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms686919%28v=vs.85%29.aspx>

Windows - Afinity



afinita

určení CPU (jádra CPU), na kterých může proces běžet

hard afinity

seznam jader

soft afinity

vlákno přednostně plánováno na procesor, kde běželo naposledy



Přečtěte si...

http://cs.wikipedia.org/wiki/Plánování_procesů

http://en.wikipedia.org/wiki/Scheduling_%28computing%29

shrnutí – vhodné pro zopakování

http://cs.wikipedia.org/wiki/Preempce_%28informatika%29

http://cs.wikipedia.org/wiki/Změna_kontextu

<http://cs.wikipedia.org/wiki/Mikrojádru>

http://cs.wikipedia.org/wiki/Round-robin_scheduling

http://cs.wikipedia.org/wiki/Priority_scheduling

http://cs.wikipedia.org/wiki/Earliest_deadline_first (RTOS)

http://cs.wikipedia.org/wiki/Completely_Fair_Scheduler (CFS)



Linux

- vlastní jádro
 - (nepreemptivní, dobrovolně preemptivní, preemptivní)
 - epocha
 - čas přidělený procesu
 - když jej všechny procesy **po částech** spotřebují, začíná nová epocha, tedy dostanou nový přidělený čas
 - plánovače (nastavitelné per proces)
 - SCHED_FIFO – pro RT úlohy bez přerušení
 - SCHED_RR (RoundRobin) – RT úlohy, preemptivně
 - SCHED_BATCH – pro dávkové úlohy
 - SCHED_OTHER – běžné úlohy (nice, dynamické priority)
-



Linux scheduler

verze do 2.6

multilevel feedback queue (pozor, trochu jiný)

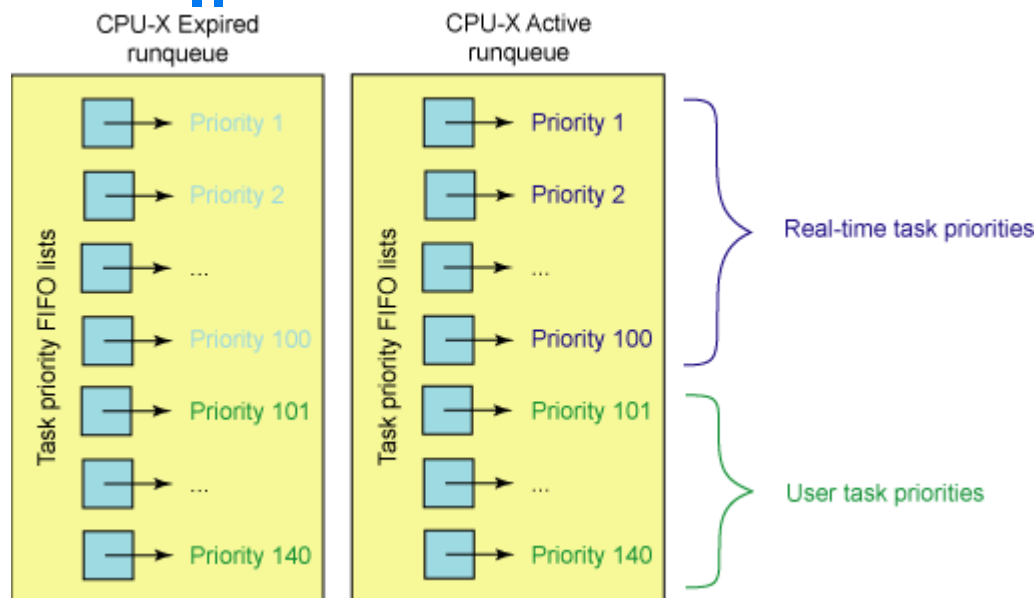
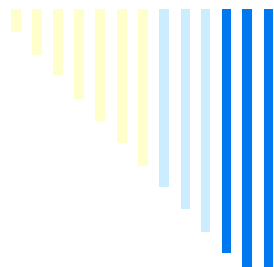
procesy mají time slice

priority 0-140

- 0 – 99 real-time úlohy, kvantum 200ms
- 100-140 nice tasks level, kvantum 10ms

dvě fronty

- active queue
 - když je prázdná, vymění se jejich role
 - expired queue
 - sem přijde proces, když vyčerpá celý svůj time slice
-



Active RunQueue,
Expired RunQueue

zdroj obrázku:
<http://www.ibm.com/developerworks/linux/library/l-scheduler/index.html>

statická priorita 0..99, k běhu vybrán s nejvyšší statickou prioritou

statická priorita 0 (SCHED_BATCH, SCHED_OTHER)

statická priorita >0 (SCHED_FIFO, SCHED_RR)

dynamická priorita (-20 až 19, viz nice)

přečíst:

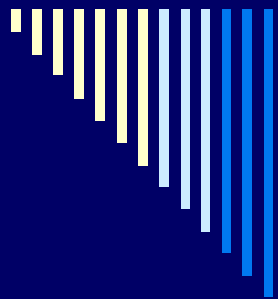
<http://www.root.cz/clanky/pridelovani-procesoru-procesum-a-vlaknum-v-linuxu/>



Linux scheduler

□ O(1) scheduler

- verze 2.6-2.6.23
- fronta připravených pro každý procesor
- pole active, expired ; v active nic – nová epocha



Completely Fair Scheduler (CFS) !!

- od verze jádra 2.6.23 do současnosti
- **red-black tree** místo front
 - klíč: spent processor time
(*kolik času na CPU již spotřeboval proces*)
 - nanosekundy pro accounting
- rovnoměrné rozdělení času procesům

Red-black tree

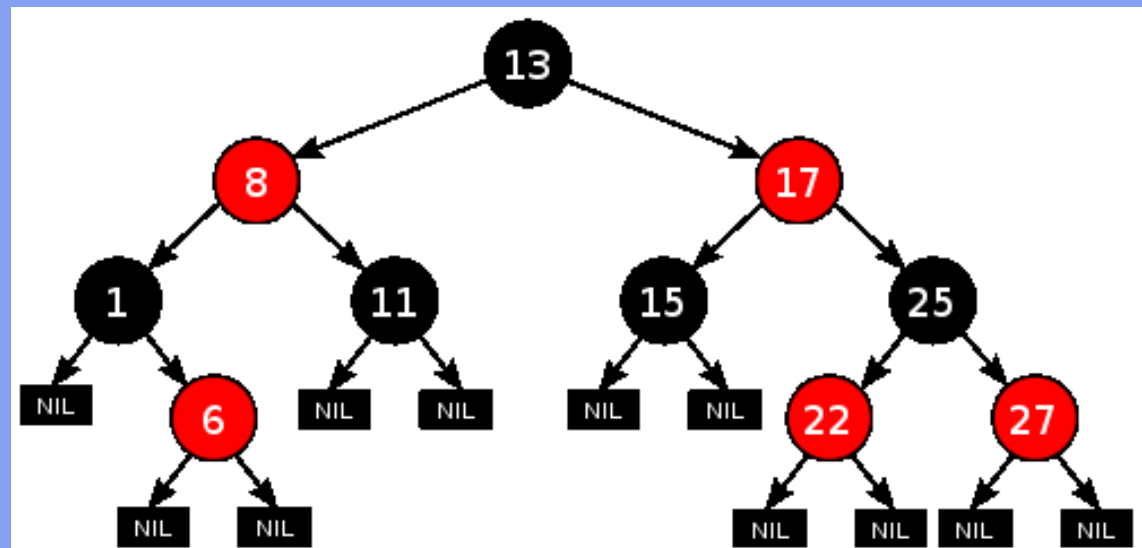
viz wikipedia

self-balancing binary search tree

uzel je červený nebo černý, kořen je černý

všechny listy jsou černé

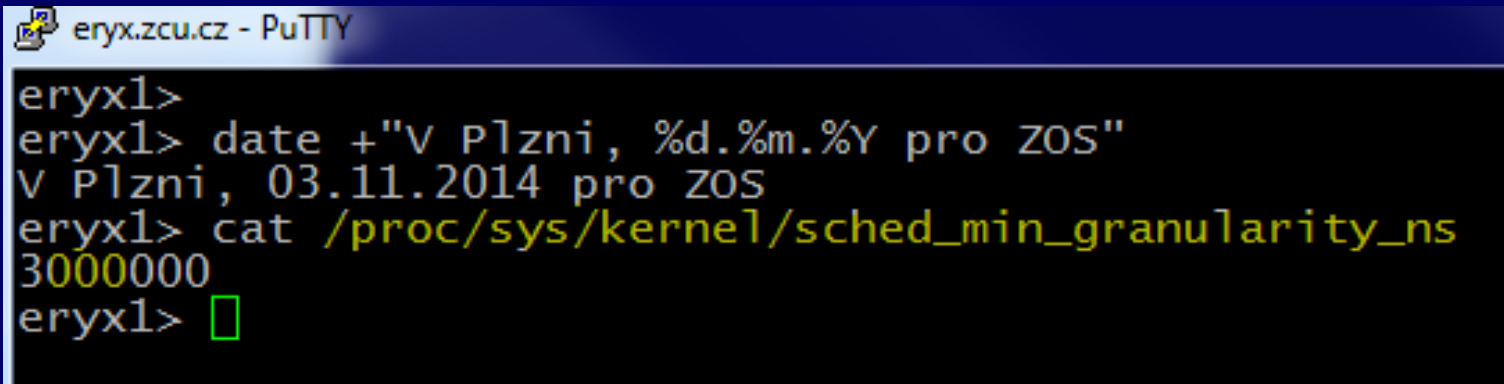
každá jednoduchá cesta z uzlu do listu obsahuje stejný počet černých uzlů





CFS - poznámky

cat /proc/sys/kernel/sched_min_granularity_ns



```
eryxl>
eryxl> date +"V Plzni, %d.%m.%Y pro ZOS"
V Plzni, 03.11.2014 pro ZOS
eryxl> cat /proc/sys/kernel/sched_min_granularity_ns
3000000
eryxl> 
```

nastavit plánovač od desktopu (nízké latence) k serveru
(větší dávky)



CFS – plánovací politiky

□ SCHED_NORMAL

- tradiční SCHED_OTHER
- pro běžné úlohy (tasky)

□ SCHED_BATCH

- preempce po delším čase, tj. dávkové úlohy
- lepší využití cache x interaktivita

□ SCHED_IDLE

- ještě slabší než **nice 19**, ale není idle time scheduler – vyhne se problémům s inverzí priority
-

CFS - poznámky

podrobnější popis:

<http://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/tree/Documentation/scheduler/sched-design-CFS.txt>

jaký plánovač používá Linux Kernel (3.0+)?

<http://stackoverflow.com/questions/15875792/scheduling-mechanism-in-linux-kernel-3-0>

1 Answer

active

oldest

votes



5



Linux is currently using the CFS (Completely Fair Scheduler) scheduler. You can read about it in the kernel documentation. It also contains a real-time scheduler which is disabled by default.

For a very short summary, CFS maintains a time-ordered red-black tree, where all runnable tasks are sorted by the amount of work the CPU has already performed (accounting for wrap-arounds). CFS picks the task with the least amount of work done and "sticks to it". More details are available in the documentation.

share | improve this answer

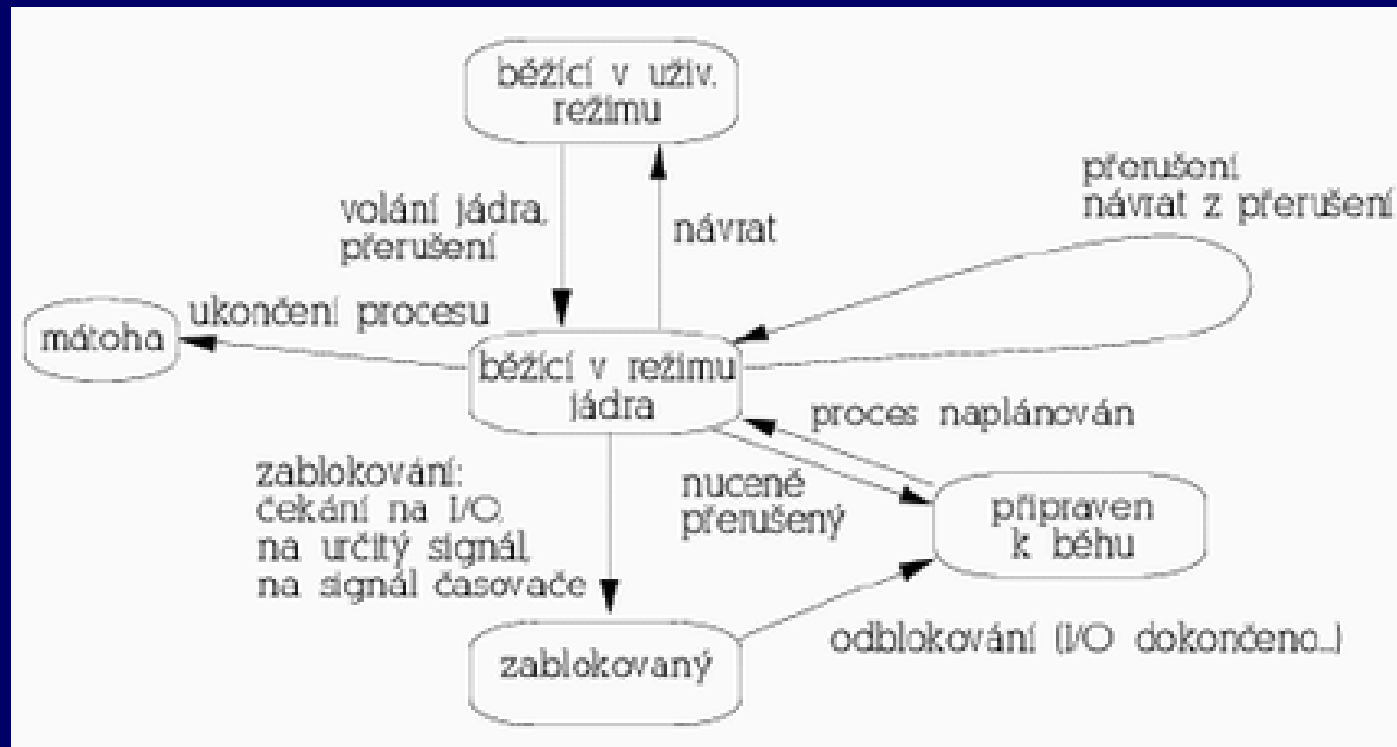
answered Apr 8 '13 at 12:15



Michael Foukarakis

15.4k ● 2 ● 38 ● 64

Linux – stavy procesů



obrázek z: <http://www.linuxzone.cz/index.phtml?ids=9&idc=252>

Linux – příkaz top

```
eryx.zcu.cz - PuTTY

Tasks: 107 total,  2 running, 105 sleeping,  0 stopped,  0 zombie
Cpu(s): 13.4%us, 11.6%sy,  0.0%ni, 75.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2068268k total,  635328k used, 1432940k free, 195476k buffers
Swap: 1052216k total,    0k used, 1052216k free, 290608k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
  4281 student6  25   0   5864  3204 2228  R   100   0.2   2838:00 tcsh
    1 root      15   0   3612  1192 1096  S    0   0.1    0:01.61 init
    2 root      RT    0     0     0     0  S    0   0.0    0:00.52 migration/0
    3 root      34  19     0     0     0  S    0   0.0    0:00.00 ksoftirqd/0
    4 root      RT    0     0     0     0  S    0   0.0    0:00.07 migration/1
    5 root      34  19     0     0     0  S    0   0.0    0:00.00 ksoftirqd/1
    6 root      RT    0     0     0     0  S    0   0.0    0:00.15 migration/2
    7 root      34  19     0     0     0  S    0   0.0    0:00.00 ksoftirqd/2
    8 root      RT    0     0     0     0  S    0   0.0    0:00.05 migration/3
    9 root      34  19     0     0     0  S    0   0.0    0:00.00 ksoftirqd/3
   10 root      10  -5     0     0     0  S    0   0.0    0:00.00 events/0
   11 root      10  -5     0     0     0  S    0   0.0    0:00.00 events/1
   12 root      10  -5     0     0     0  S    0   0.0    0:00.00 events/2
   13 root      10  -5     0     0     0  S    0   0.0    0:00.00 events/3
   14 root      12  -5     0     0     0  S    0   0.0    0:00.01 khelper
   15 root      10  -5     0     0     0  S    0   0.0    0:00.00 kthread
   66 root      10  -5     0     0     0  S    0   0.0    0:00.00 kblockd/0

eryx2>
```

1. PID procesu
2. USER – identita uživatele
3. PRI – aktuální priorita daného procesu
4. NICE – výše priority příkazem nice
 - ▣ Záporné číslo – vyšší priorita
 - ▣ Kladné číslo – sníží prioritu (běžný uživatel)
5. VIRT – celková velikost procesu
 - ▣ Kód + zásobník + data
6. RES – velikost použité fyzické paměti
7. SHR – sdílená paměť
8. STAT – stav procesu
9. %CPU – kolik procent CPU nyní využívá
10. %MEM – procento využití fyzické paměti daným proc.
11. TIME – celkový procesorový čas
12. COMMAND - příkaz



Příkaz nice

Změna priority procesu

- Běžný uživatel 0 až +19, tedy pouze snižovat
- root: -20 (nejvyšší) až +19 (nejnižší)

```
eryx2> /bin/bash
```

```
eryx2> nice -n -5 sleep 10
```

```
nice: cannot set niceness: Permission denied
```

```
eryx2> nice -n +5 sleep 10
```

Pozn: záleží i na shellu, který máme



Příkaz renice

Změna priority běžícího procesu

Běžný uživatel

- může měnit jen u svých procesů
- opět pouze snižovat

eryx2> renice +10 32022

32022: old priority 5, new priority 10



Proces – stav blokováný (Unix)

- čeká na událost – ve frontě
 - **přerušitelné signálem** (terminál, sockety, pipes)
 - procesy označené **S**
 - signál – syscall se zruší – návrat do userspace
 - obsluha signálu
 - znovu zavolá přerušené syst. volání (pokud požadováno)
 - **nepřerušitelné**
 - procesy označené **D**
 - operace s diskem – skončí v krátkém čase
 - plánovač mezi nimi nerozlišuje
-



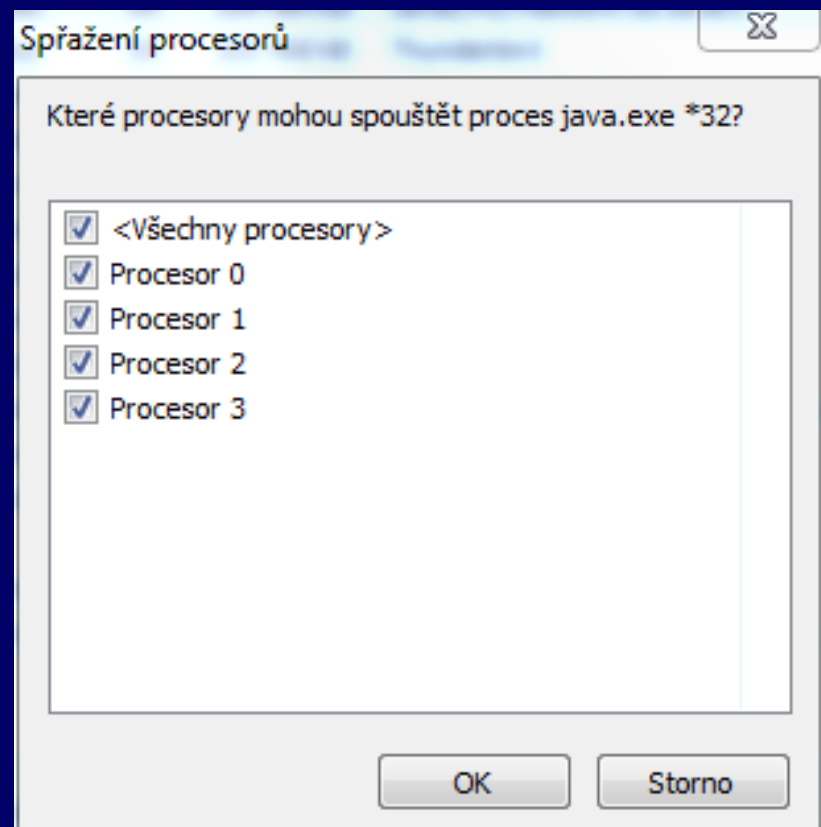
Plánování – víceprocesorové stroje

- nejčastější architektura
 - těsně vázaný symetrický multiprocessor
 - procesory jsou si rovné, společná hlavní paměť
- Přirazení procesů procesorům – ukázka
 - Permanentní přiřazení
 - Menší režie, některá CPU mohou zahálet
 - Afinita procesu k procesoru, kde běžel naposledy
 - Někdy procesoru přiřazen jediný proces – RT procesy
 - Společná fronta připravených procesů
 - Plánovány na libovolný procesor

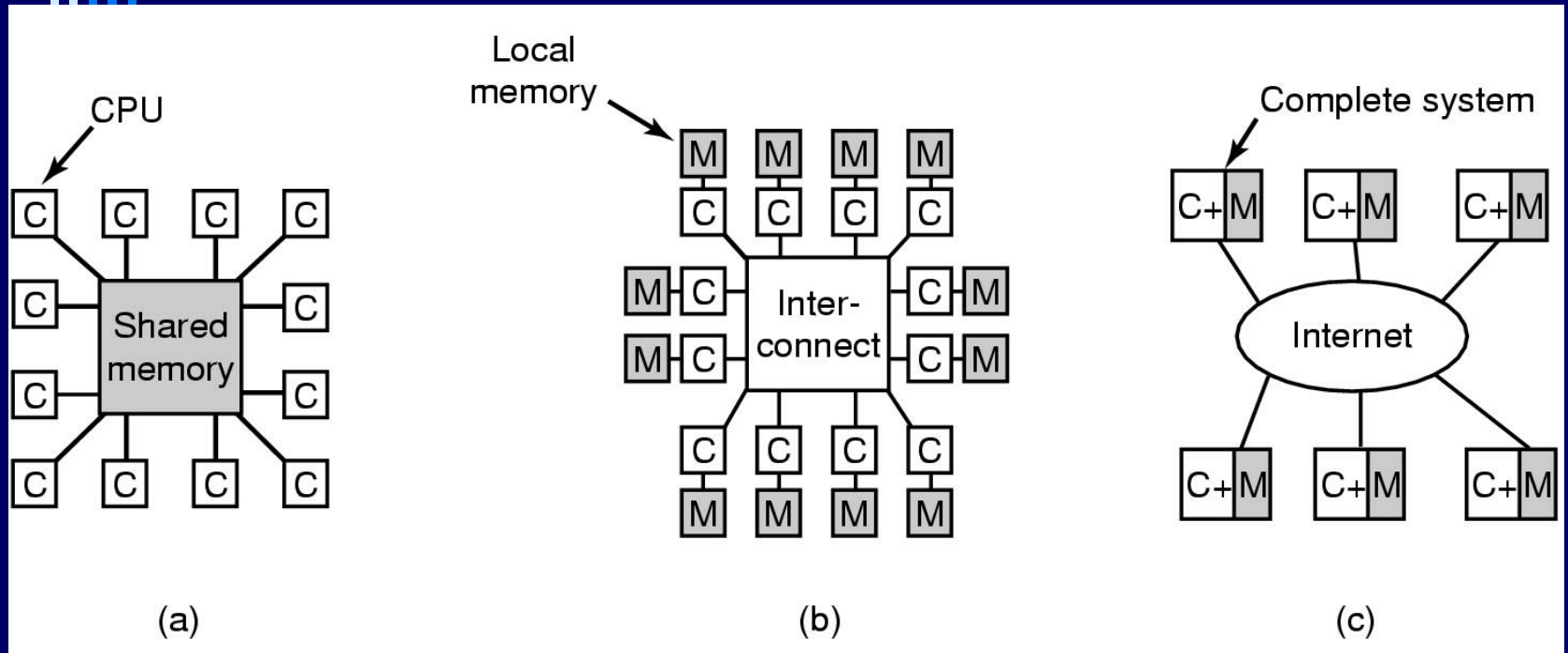
afinita

na jakých CPU může daný proces běžet

správce úloh systému
Windows – procesy –
vybrat proces – pravá myš
– nastavit spřažení

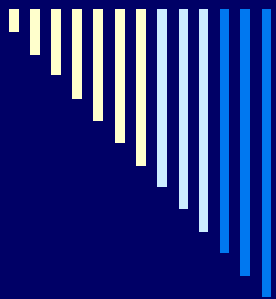


Multiprocessorové systémy



Architektury:

- shared memory model (sdílená paměť)
- message passing multiprocessor (předávání zpráv)
- wide area distributed system (distribuovaný systém)

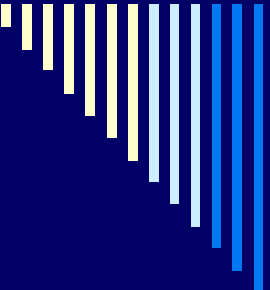


Víceprocesorové stroje

Plánování vláken

Paralelní aplikace – často podstatně větší výkonnost, pokud jejich vlákna běží současně

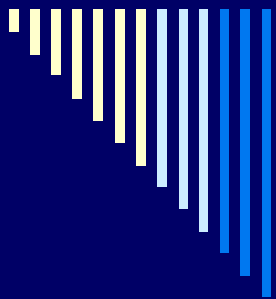
Zkrátí se vzájemné čekání vláken



Plánování v systémech reálného času

Charakteristika RT systémů

- RT procesy **řídí** nebo **reagují** na události ve vnějším světě
- Správnost závisí nejen na **výsledku**, ale i na **čase**, ve kterém je výsledek vyprodukován
- S každou podúlohou – sdružit **deadline** – čas kdy musí být spuštěna nebo dokončena
- **Hard RT** – času **musí** být dosaženo
- **Soft RT** – dosažení deadline je **žádoucí**



Systemy RT

Podúlohy procesu (události, na které se reaguje)

- **Aperiodické** – nastávají **nepredikovatelně**
- **Periodické** – v pravidelných **intervalech**

Zpracování události vyžaduje čas

Pokud je možné všechny včas zpracovat

=> systém je **plánovatelný (schedulable)**



Plánovatelné RT systémy

□ Je dáno

- m – počet periodických událostí
- výskyt události i s periodou P_i vyžadující C_i sekund

□ Zátěž lze zvládnout, pokud platí:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$



Plánovací algoritmy v RT

□ Statické nebo dynamické

□ Statické

- Plánovací rozhodnutí **před spuštěním** systému
- Předpokládá dostatek informací o vlastnostech procesů

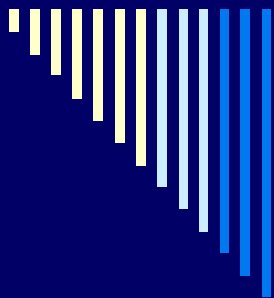
□ Dynamické

- **Za běhu**
- Některé algoritmy provedou analýzu plánovatelnosti, nový proces přijat pouze pokud je výsledek plánovatelný

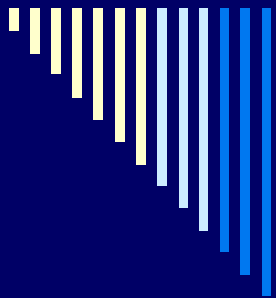


Vlastnosti současných RT

- **Malá velikost** OS → omezená funkčnost
 - Snaha **spustit** RT proces **co nejrychleji**
 - Rychlé přepínání mezi procesy nebo vlákny
 - Rychlá obsluha přerušení
 - Minimalizace intervalů, kdy je přerušení zakázáno
 - Multitasking + meziprocesová komunikace (semaforey, signály, události)
 - Primitiva pro zdržení procesu o zadaný čas, čítače časových intervalů
 - Někdy rychlé sekvenční soubory (viz později)
-



Zpátky obecně k plánování procesů



Plánování procesů a vláken

- Plánování **procesů** – vždy součást OS

- Plánování **vláken**
 - **Běh vláken plánuje OS**
 - Kernel-level threads
 - **Běh vláken plánován uživatelským procesem**
 - User-level threads
 - OS o existenci vláken nic neví



Plánování vláken

□ Vlákna plánována OS

- Stejné mechanismy a algoritmy jako pro plánování procesů
- Často plánována bez ohledu, kterému procesu patří (proces 10 vláken, každé obdrží časové kvantum)



Plánování vláken

□ Vlákna plánována uvnitř procesu

- Běží v **rámci času**, který je přidělen **procesu**
- Přepínání mezi vlákny – **systémová knihovna**
- Pokud OS neposkytuje procesu pravidelné “přerušení”, tak pouze nepreemptivní plánování
- Obvykle algoritmus RR nebo prioritní plánování
- Menší režie oproti kernel-level threads, menší možnosti

□ Windows 2000> a Linux – vlákna plánována jádrem

□ Některé varianty UNIXu – user-level threads



Dispatcher

□ Dispatcher

- Modul, který předá řízení CPU procesu vybraným **short-term** plánovačem

□ Proveďte:

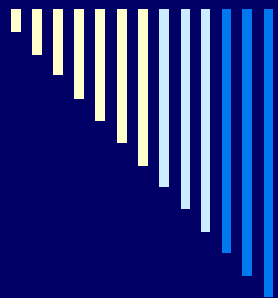
- Přepnutí kontextu
- Přepnutí do uživatelského modu
- Skok na danou instrukci v uživatelském procesu

□ Co nejrychlejší, vyvolán během každého přepnutí procesů



Scheduler – protichůdné požadavky

- příliš časté přepínání procesu – velká režie
- málo časté – pomalá reakce systému
- čekání na diskové I/O, data ze sítě – probuzen a brzy (okamžitě) naplánován – pokles přenosové rychlosti
- multiprocesor – pokud lze, nestřídat procesory
- nastavení priority uživatelem



Poznámka – vyhladovění procesu

V roce 1973 na MITU shut down stroje IBM 7094
Nalezen proces, který nebyl spuštěn od roku 1967



Poznámka - simulace

- **Trace tape** – monitorujeme běh reálného systému, zaznamenáváme posloupnost událostí
- Tento záznam použijeme pro řízení simulace
- Lze využít pro porovnávání algoritmů
- Trace tape – nutno uložit velké množství dat



Uvíznutí (deadlock)

- Příklad:
 - Naivní večeřící filozofové – vezmou levou vidličku, ale nemohou vzít pravou (už je obsazena)
 - Uvíznutí (deadlock); zablokování
-



Uvíznutí – alokace I/O zařízení

Výhradní alokace I/O zařízení

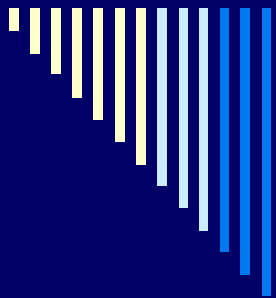
zdroje:

Vypalovačka CD (**V**), scanner (**S**)

procesy:

A, **B** – oba úkol naskenovat dokument a zapsat na vypalovačku

1. **A** žádá **V** a dostane, **B** žádá **S** a dostane
2. **A** žádá **S** a čeká, **B** žádá **V** a čeká -- **uvíznutí !!**



Uváznutí – zamykání záznamů v databázi, semaforey

- Dva procesy A, B požadují přístup k záznamům R,S v databázi
- A zamkne R, B zamkne S, ...
- A požaduje S, B požaduje R

Vymyslete příklad deadlocku s využitím semaforů



Zdroje

- **přeplánovatelné** (preemptable)
 - lze je odebrat procesu bez škodlivých efektů
- **nepřeplánovatelné** (nonpreemptable)
 - proces zhavaruje, pokud jsou mu odebrány



Zdroje

□ Sériově využitelné zdroje

- Proces zdroj **alokuje, používá, uvolní**

□ Konzumovatelné zdroje

- Např. zprávy, které **produkuje jiný proces**
- Viz producent – konzument

Také zde uvíznutí:

1. Proces A: ... receive (B,R); send (B, S); ..
2. Proces B: ... receive (A,S); send (A, R); ..

Dále
budeme
povídat o
sériově
využitelných
zdrojích,

problémy
jsou stejné



Více zdrojů stejného typu

Některé zdroje – **více exemplářů**

Proces žádá zdroj **daného typu** – **jedno** který dostane

Např. bloky disku pro soubor, paměť, ...

□ Př. 5 zdrojů a dva procesy A,B

1. A požádá o dva zdroje, dostane (zbydou 3)
2. B požádá o dva zdroje, dostane (zbude 1)
3. A žádá o další dva, nejsou (je jen 1), čeká
4. B žádá o další dva, nejsou, čeká – nastalo uvíznutí

Zaměříme se na situace, kdy 1 zdroj každého typu



Práce se zdrojem

□ Žádost (request)

- Uspokojena bezprostředně nebo proces čeká
- Systémové volání

□ Použití (use)

- Např. tisk na tiskárně

□ Uvolnění (release)

- Proces uvolní zdroj
 - Systémové volání
-



Uvíznutí - definice

- Obecný termín **zdroj** – zařízení, záznam, ...

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

- Všichni čekají – nikdo událost nevygeneruje, nevzbudí jiný proces
-



Podmínky vzniku uvíznutí (!!!)

Coffman, 1971

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
-



Podmínky vzniku uvíznutí

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



Vznik uvíznutí - poznámky

- Pro vznik uvíznutí – musejí být **splněny všechny 4 podmínky**
 - 1. až 3. předpoklady, za nich je definována 4. podmínka
- Pokud jedna z podmínek **není splněna**, uvíznutí **nenastane**
- Viz příklad s CD vypalovačkou
 - Na CD může v jednu chvíli zapisovat pouze 1 proces
 - CD vypalovačku není možné zapisovacímu procesu odejmout



Modelování uvíznutí

Graf alokace zdrojů

□ 2 typy uzlů

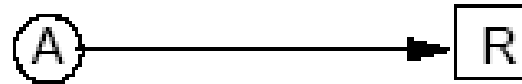
- **Proces** – zobrazujeme jako kruh
- **Zdroj** – jako čtverec

□ hrany

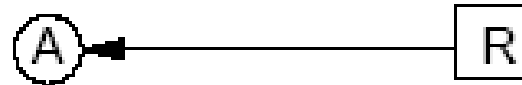
- **Hrana od zdroje k procesu:**
 - zdroj držen procesem
- **Hrana od procesu ke zdroji:**
 - proces blokován čekáním na zdroj

Modelování uvíznutí

proces A čeká na zdroj R:



zdroj R je držen procesem A:



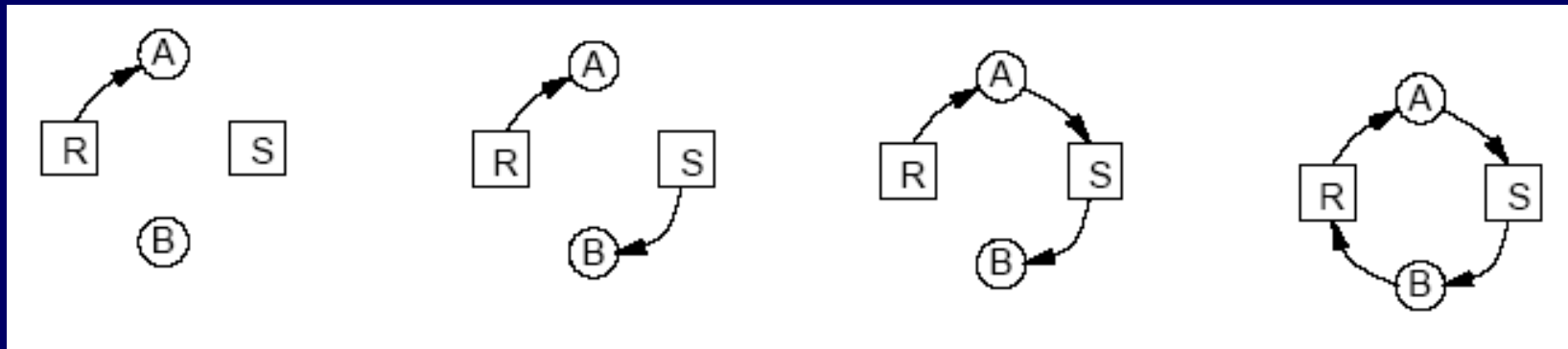
Cyklus v grafu – nastalo **uvíznutí**

Uvíznutí se týká procesů a zdrojů v cyklu

Uváznutí

zdroje: Rekorder R a scanner S; procesy: A,B

1. A žádá R dostane, B žádá S dostane
2. A žádá S a čeká, B žádá R a čeká - uváznutí



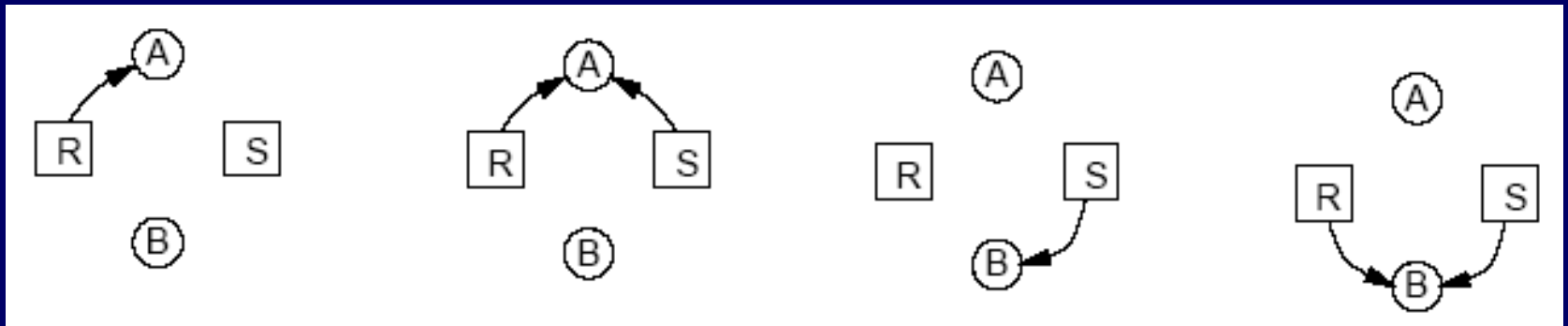


Uváznutí - poznámky

- **Cyklus v grafu** – **nutnou a postačující** podmínkou pro vznik uváznutí
- Závisí **na pořadí** vykonávání instrukcí procesů
- Pokud nejprve alokace a uvolnění zdrojů procesu A, potom B => uváznutí **nenastane**

Uváznutí - poznámky

1. A žádá R a S, oba dostane, A oba zdroje uvolní
 2. B žádá S a R, oba dostane, B oba zdroje uvolní
- Nenastane uváznutí
 - Při některých bězích nemusí uváznutí nastat – hůře se hledá chyba



Uváznutí – pořadí alokace

- Pokud bychom napsali procesy A,B tak, aby oba žádaly o zdroje R a S **ve stejném pořadí** – uváznutí **nenastane**
1. A žádá R a dostane, B žádá R a čeká
 2. A žádá S a dostane, A uvolní R a S
 3. B čekal na R a dostane, B žádá S a dostane

