

07.

# Plánování procesů Deadlock

---

ZOS 2016, 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 – na každého se dostalo

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 **přerušování** (tiky časovače, clock ticks)
- vyvolá se **obslužný podprogram v jádře**
- **rozhodnutí**, zda proces bude pokračovat, nebo se spustí jiný (**preemptivní plánování**) – po několika tikech časovače

# Algoritmus cyklické obsluhy – Round Robin (RR)

---

jeden z nejstarších a nejpoužívanějších

každému procesu přiřazen

**časový interval** = **časové kvantum**, po které může proces běžet

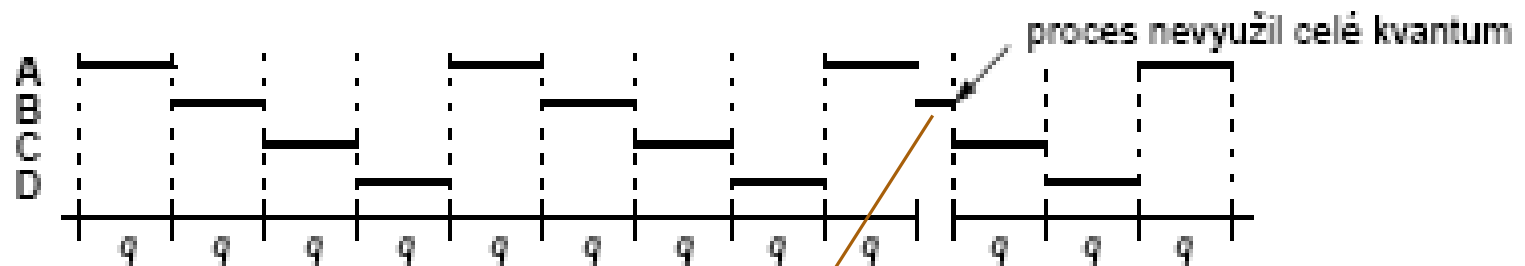
**proces běží na konci svého kvanta**

- **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 časové kvantum – odpovídá více přerušením časovače

---

Časovač může proces v průběhu časového kvanta přerušit vícekrát.

Pokud bude přerušení 100x za sekundu, tj. každých 10ms  
a časové kvantum bude mít hodnotu 50ms

=> 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 – kvantum 1s, 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
  - jeden z možných návrhů řešení

# Prioritní plánování

---

- předpoklad RR: všechny procesy **stejně důležité**
- ale:
  - vyšší priorita zákazníkům, kteří si „připlatí“
  - interaktivní procesy vs. procesy běžící na pozadí
- prioritu lze přiřadit staticky nebo dynamicky:
- **staticky**
  - při startu procesu, např. Linux – příkaz **nice**
- **dynamicky**
  - přiřadit I/O procesům 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 má, ale je ve stavu blokový, 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 typicky cca 30 až 50 prioritních tříd (ale i více)

# 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 časovače – 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 (např. celkem 100 losů, proces má 20 – v dlouhodobém průměru dostane 20% času)

# Loterie - výhody

---

řešení problémů, které jsou v jiných plán. algoritmech obtížné

- **spolupracující procesy – mohou si předávat losy**

- klient posílá zprávu serveru a blokuje se
- klient může serveru **propůjčit** všechny **své tikety**
- server běží s prioritou (počtem losů) daného klienta
- 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 jeden proces má prioritu např. 30 a jiný 10?
  - proces – množství ticketů – velikost šance vyhrát

zatím spíše experimentální algoritmus

# Loterie - nevýhody

---

- není deterministický
- Nemáme zaručeno, že budou naše losy „vylosovány“ v konečném čase.
- Nelze použít tam, kde jsme na determinismu závislí  
-> řídicí aplikace.



# 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
loterie	Preemptivní vyprší kvant.	$P() = 1$	Dle výsledku loterie

# Interaktivní systémy (!!!!)

---

- Základem je **round robin**

- Pojem **časové kvantum**

## **Prioritní plánování**

- Statická a dynamická složka priority

Spojení RR + priority => **prioritní třídy**

## **Spravedlivé sdílení**

- Modifikace plánovače pro spravedlnost vůči uživatelům

## **Loterie**

- Experimentální, zajímavé vlastnosti
  - Nelze použít pro řídicí systémy - nedeterminismus

# 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

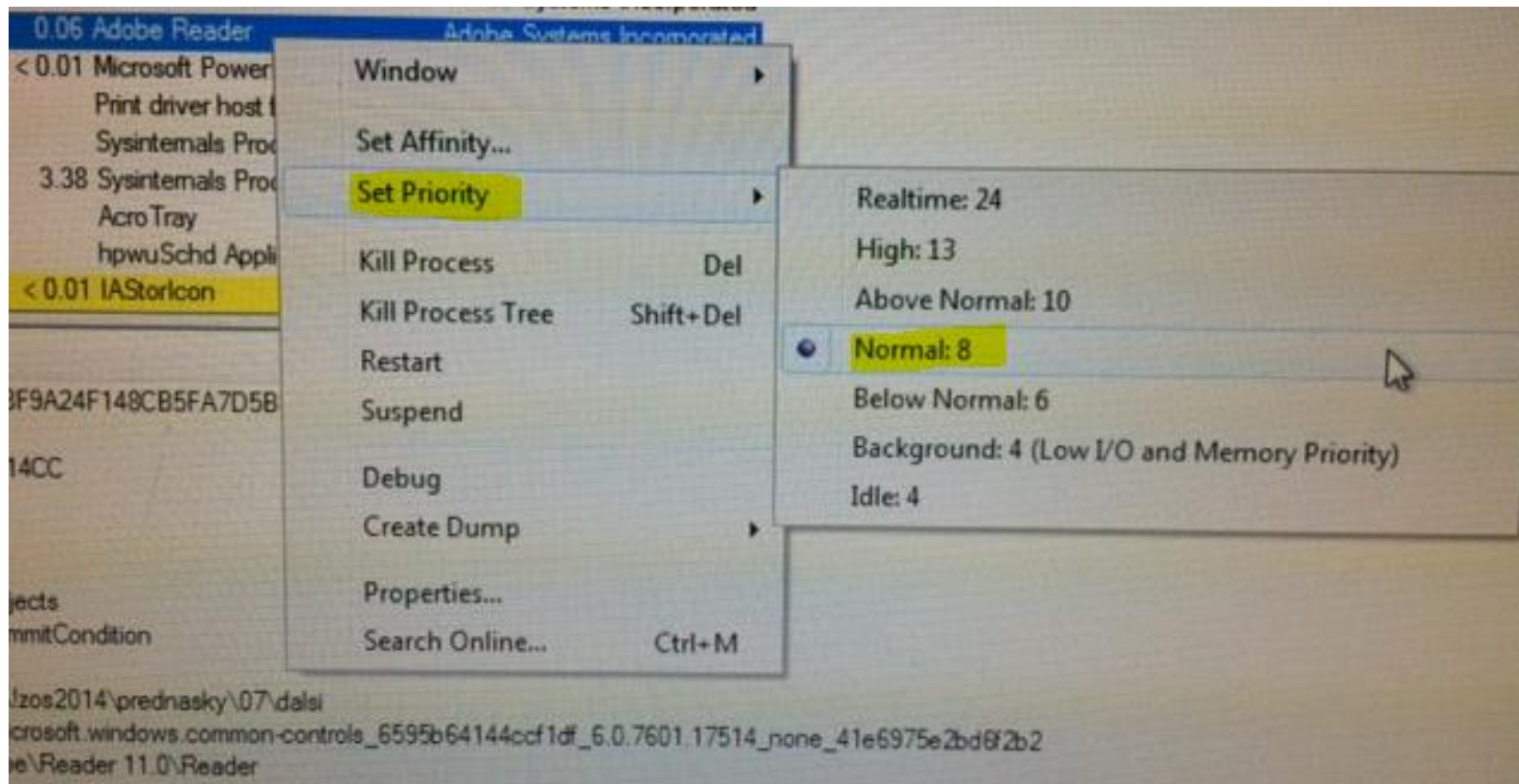
- nalezne **neprázdnou** frontu
- naplánuje první proces, nechá ho běžet **1 kvantum**
- po uplynutí kvanta – proces na konec fronty na příslušné prioritní úrovni

# Windows – skupiny priorit

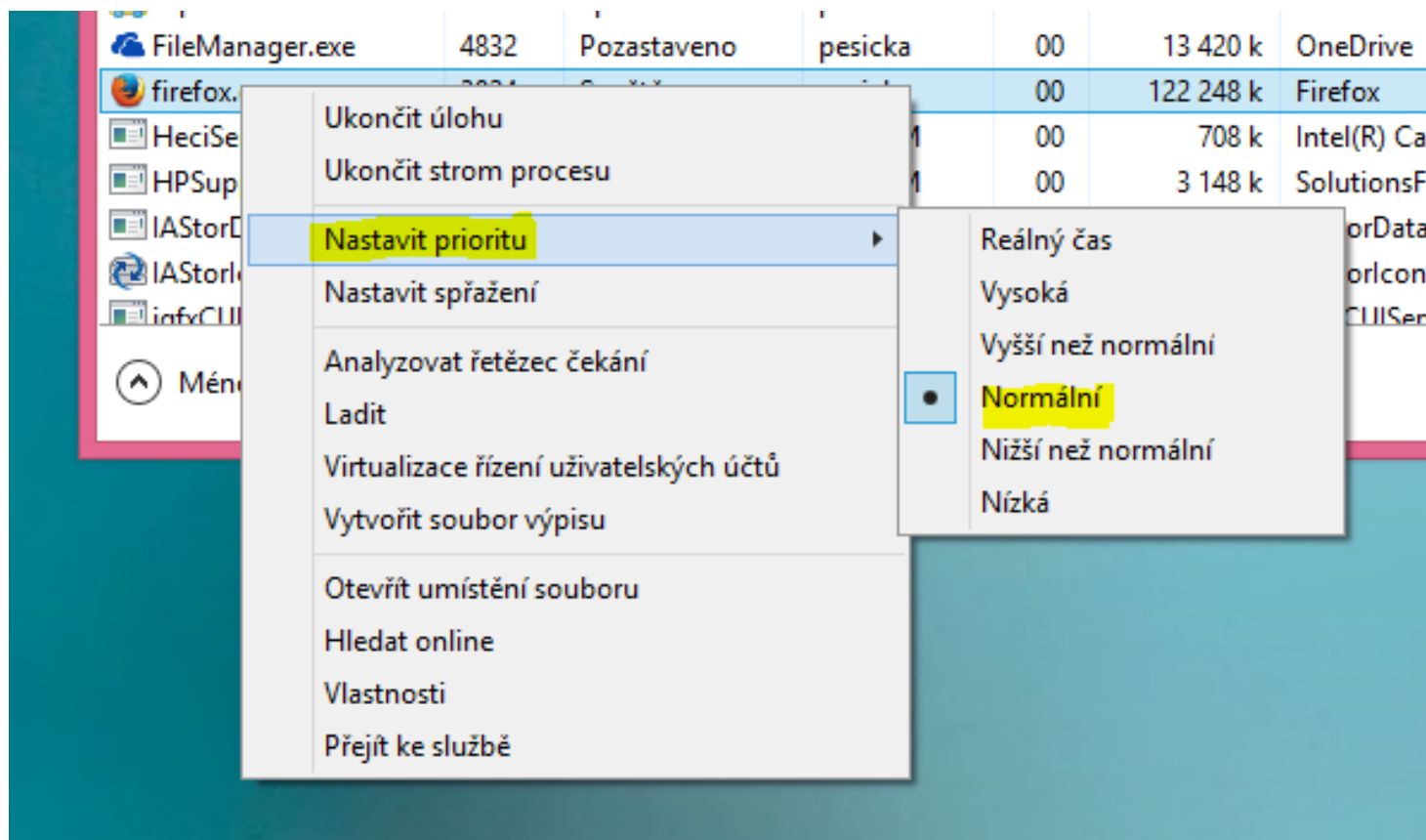
---

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

# Process explorer ze sysinternals sady programů



# Windows 8/10 správce úloh



# Windows - priority

---

- 0 .. pokud není nic jiného na práci
- 1 .. 15 – obyčejné procesy
- aktuální priorita – <bázová, 15>
- **bázová** 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 rovnoměrně 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 dřívějšíka“, aby se nedostal k informacím, ke kterým se dostat nemá

# Kvantum, stretching

---

- kvantum stretching
  - maximum 6 tiků (3x proti základu)
  - middle 4 tiky (2x proti základu)
  - none 2 tiky (základ)
- Na **desktopu** je defaultní kvantum **2 ticky** u vlákna 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

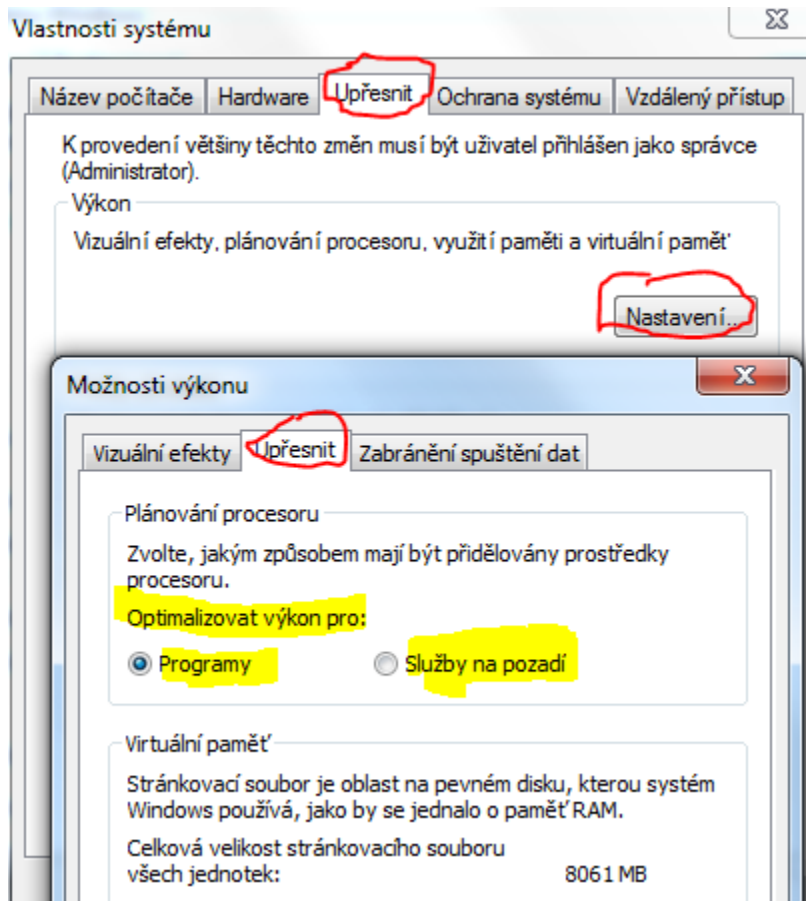
---

Program **clockres** ze sady Sysinternal

```
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
```

# Windows 7/10

## Systém – 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)

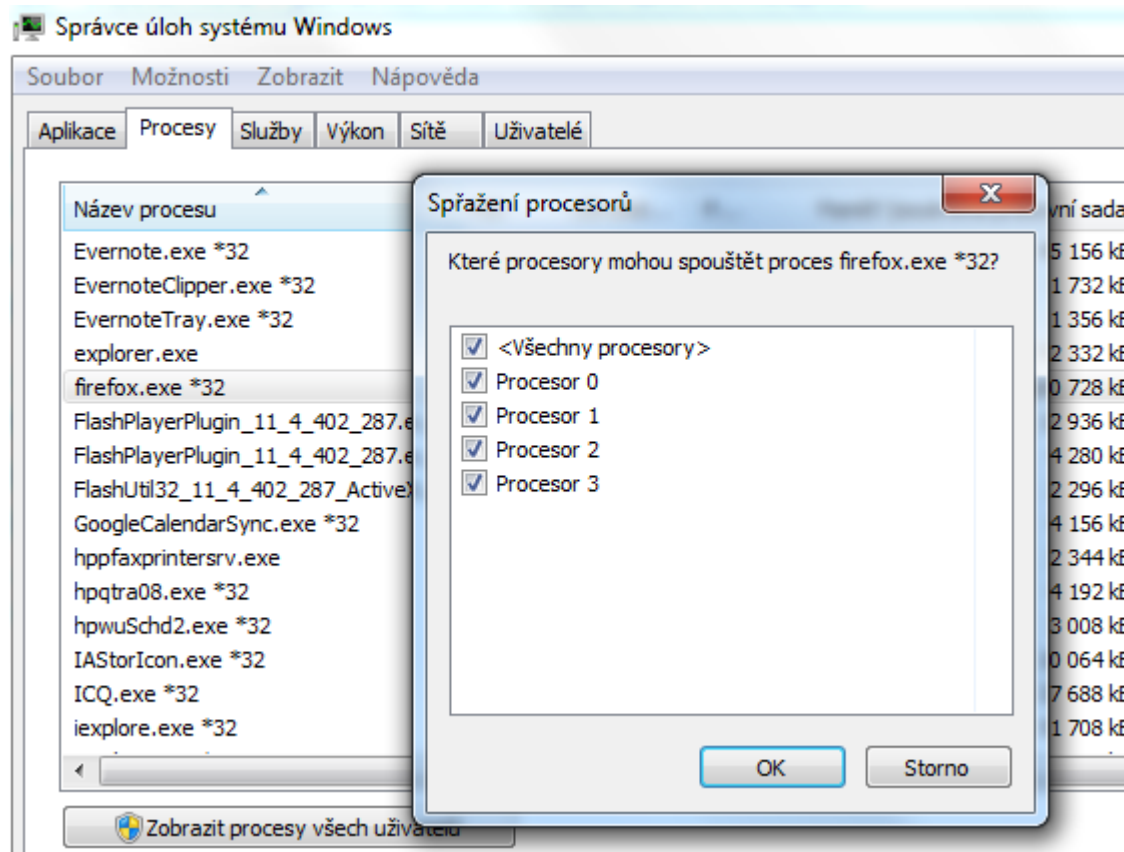
---

- kromě vláken i fibers
- fibers plánuje vlastní aplikace, nikoliv centrální plánovač jádra
- vytvoření fiberu: `CreateFiber`
- nepreemptivní plánování – odevzdá řízení jinému vlákenku 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



# Windows Vista – modifikace plánovače

---

- používá **cycle counter registr** moderních CPU
  - Počítá, kolik CPU cyklů vlákno vykonalo (než jen používat intervalový časovač)
- prioritní plánování I/O fronty
  - Defragmentace neovlivňuje procesy na popředí

# Přečtěte si...

---

[http://cs.wikipedia.org/wiki/Plánování\\_procesů](http://cs.wikipedia.org/wiki/Plánování_procesů)

[http://en.wikipedia.org/wiki/Scheduling\\_%28computing%29](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/Preempce_%28informatika%29)

[http://cs.wikipedia.org/wiki/Změna\\_kontextu](http://cs.wikipedia.org/wiki/Změna_kontextu)

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

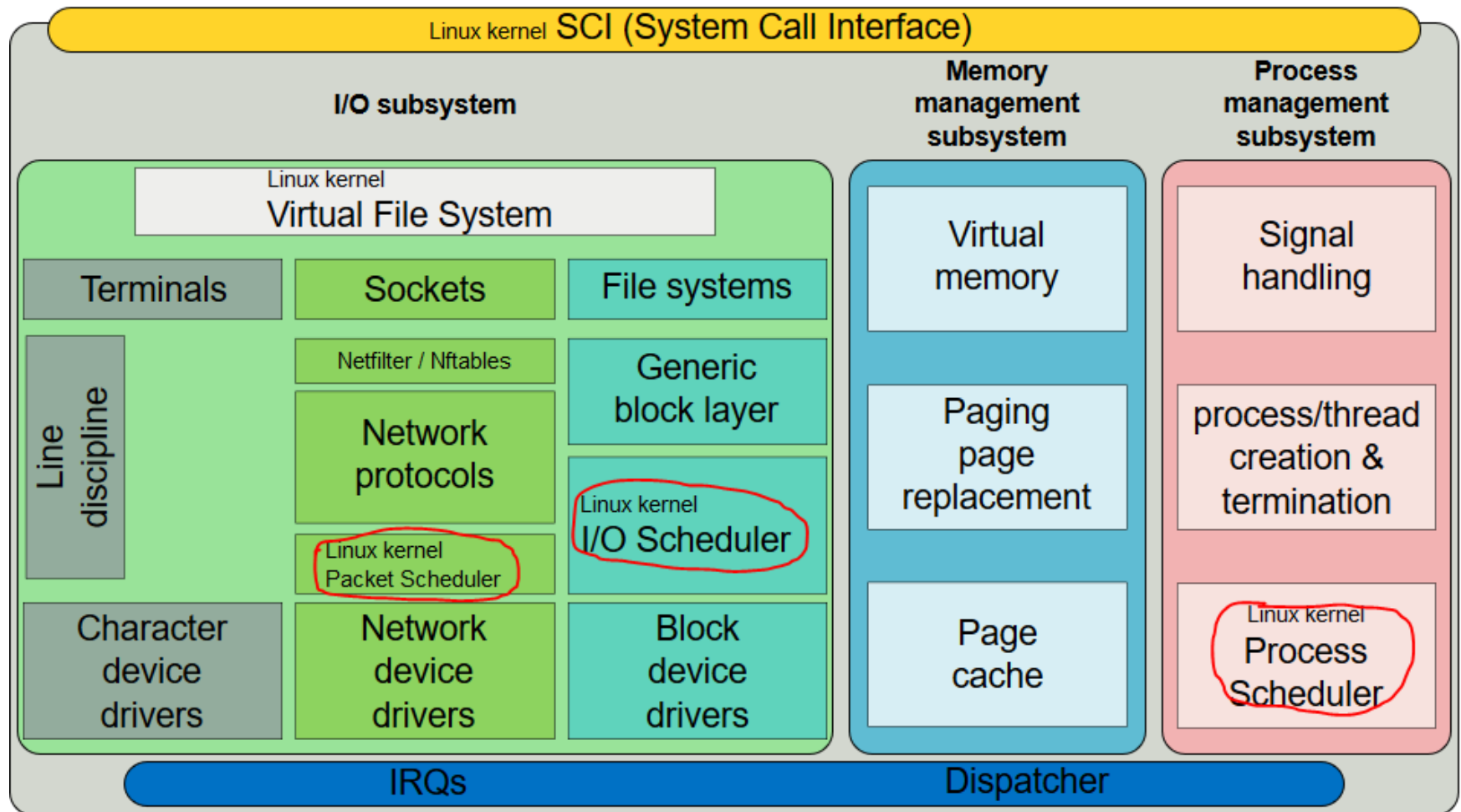
[http://cs.wikipedia.org/wiki/Round-robin\\_scheduling](http://cs.wikipedia.org/wiki/Round-robin_scheduling)

[http://cs.wikipedia.org/wiki/Priority\\_scheduling](http://cs.wikipedia.org/wiki/Priority_scheduling)

[http://cs.wikipedia.org/wiki/Earliest\\_deadline\\_first](http://cs.wikipedia.org/wiki/Earliest_deadline_first) (RTOS)

[http://cs.wikipedia.org/wiki/Completely\\_Fair\\_Scheduler](http://cs.wikipedia.org/wiki/Completely_Fair_Scheduler) (CFS)

# Linux



# Linux – vývoj plánovače

---

- jádro 2.4
  - $O(n)$  plánovač
  - Globální runque
    - úloha může být na libovolném CPU
    - (dobré pro balancování, ale ne pro cache)
- jádro 2.6
  - $O(1)$  plánovač
  - 5 integerů – bitmapa – ve které frontě je úloha k běhu
  - Čas najít úlohu nezávisí na počtu úloh ale na počtu priorit (140)
- jádro 2.6.23 a výše
  - CFS plánovač – jiný přístup, nejsou pole úloh atd.

# Linux

---

vlastní jádro

(dříve nonpreemptivní, dnes 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 plánování

---

- Většina uživatelských procesů **SCHED\_OTHER**
- V případě soft-realtimových:
  - SCHED\_FIFO (FCFS)
    - Není časové kvantum
    - Přeruší se pouze tehdy, přijde-li prioritnější
  - SCHED\_RR
    - Procesy na stejné úrovni se střídají po určitém časovém kvantu

# Linux scheduler

---

do verze jádra 2.6

multilevel feedback queue (pozor, trochu jiný než z dávkových)

procesy mají time slice

priority 0-139

- 0 – 99 real-time úlohy, kvantum 200ms
- 100-139 uživatelské úlohy, kvantum 10ms

dvě fronty

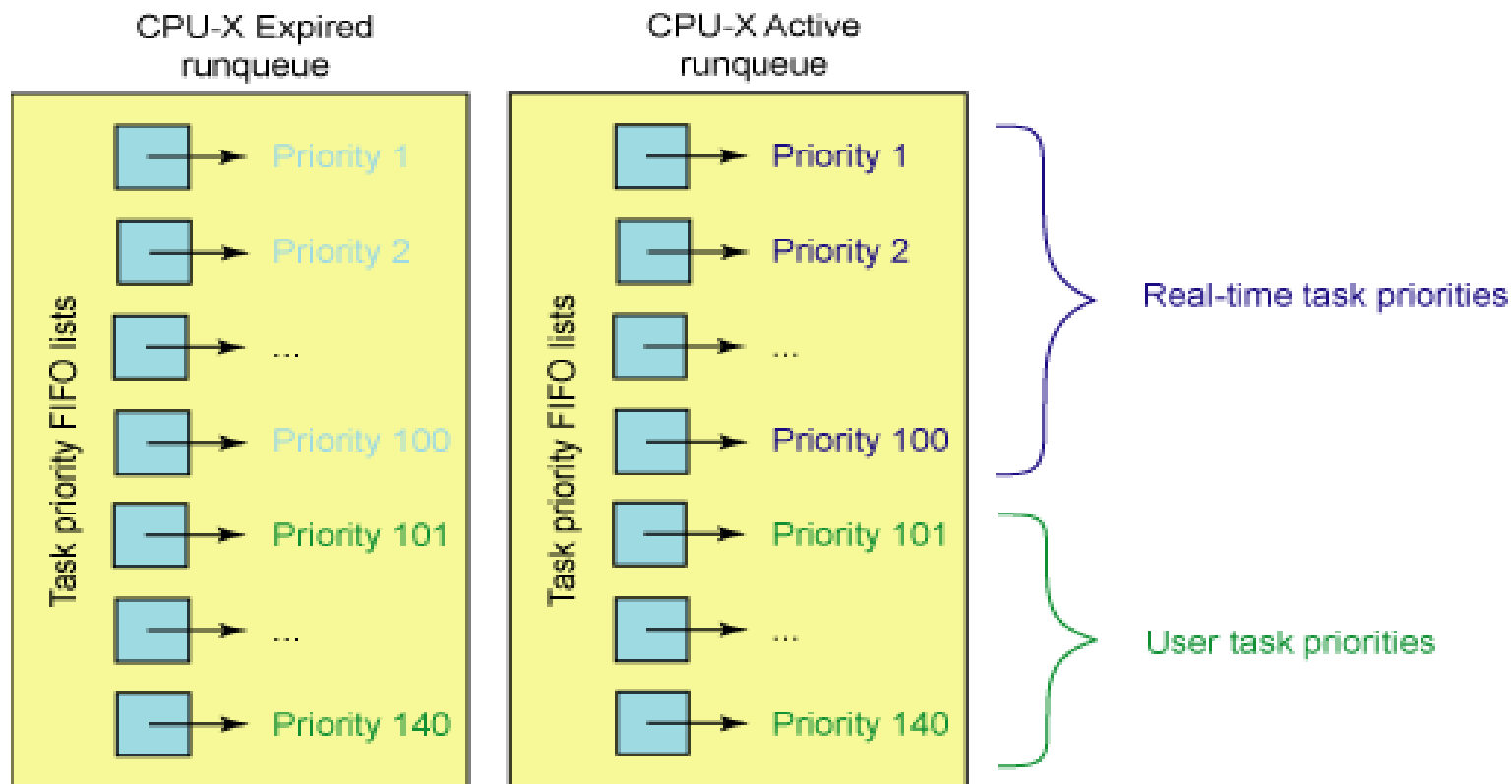
- active queue
  - když je prázdná, vymění se jejich role
- expired queue
  - sem přijde proces, když vyčerpá celý svůj time slice



Různá velikost  
kvanta



Příkaz nice



Active fronta, Expired fronta

$\text{PRIO} = \text{MAX\_RT\_PRIO} + \text{NICE} + 20$

Na stejné prioritě – round robin

zdroj obrázku:

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

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**
- každý procesor si vybírá ze své fronty
- Bitmapa 5 integerů – snadno poznáme, na jaké prioritě je task, co by chtěl běžet
- může se stát, že nějaký procesor bude chvíli idle, zatímco u jiného budou stát ve frontě
- procesy – bitová maska, na jakém CPU mohou běžet (afinita)
- maska je děděná child procesy a vlákny
- pravidelné vybalancování front (speciální kernel thread)
- pole active, expired ; v active nic → nová epocha

# Interaktivita

---

- dynamicky škáluje prioritu úlohu podle interaktivity
- Interaktivní úlohy dostávají bonus
- Aktuální bonus / penalty podle porovnání sleep average oproti konstantnímu maximum sleep average
- Netýká se RT úloh
- když úloha nevyčerpá svůj timeslice
  - Interaktivní znovu do pole [Aktivní úlohy](#)
  - Neinteraktivní – novou prioritu v poli [Expired úlohy](#) (započítá se jim, že nevyužili celý timeslice)

# Completely Fair Scheduler (CFS) !!

---

- od verze jádra 2.6.23 do současnosti
- **red-black tree** místo front
  - klíč: spent processor time (vruntime)  
(kolik času na CPU již spotřeboval proces)
  - účtovací čas v nanosekundách
- rovnoměrné rozdělení času procesům
- žádný idle procesor, pokud je co dělat
  - Na každém CPU migration thread
  - Přesunout task z jednoho CPU na jiné
  - Periodicky nebo když je potřeba

# Poznámka

---

Vruntime říká, jak moc si úloha zaslouží běžet, oproti ostatním úlohám na stejném procesoru.

Bere v úvahu celou řadu věcí – prioritu procesu, kolik času už proces na CPU strávil atd.

Nižší číslo vruntime – zasloužil by si více běžet

Organizováno ve stromu, vybere se vždy nejlevější prvek, tedy s nejnižší hodnotou vruntime.

# CFS

---

- Místo priorit se používá decay faktor (do češtiny „zahnívání“)
- Jak rychle se zmenšuje čas pro běh tasku
- Tasky s vysokou prioritou
  - Zvyšují vruntime pomaleji, potřebují více CPU času
- Tasky s nízkou prioritou
  - Vruntime se zvyšuje rychleji

# CFS pokračování

---

1. Naplánuj task s nejnižším vruntime  
(spent processor time, uzel nejvíce vlevo)
2. Task běží
3. Update vruntime
4. Znovu vložíme do stromu

# Red-black tree

viz wikipedia

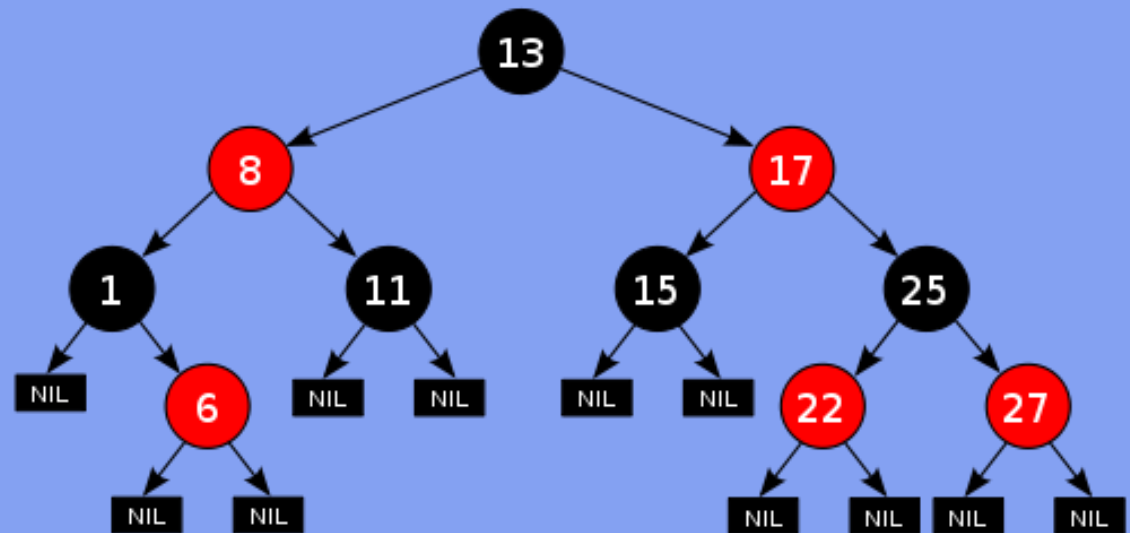
self-balancing binary search tree

(žádná cesta ve stromu není dvojnásobek jiné cesty)

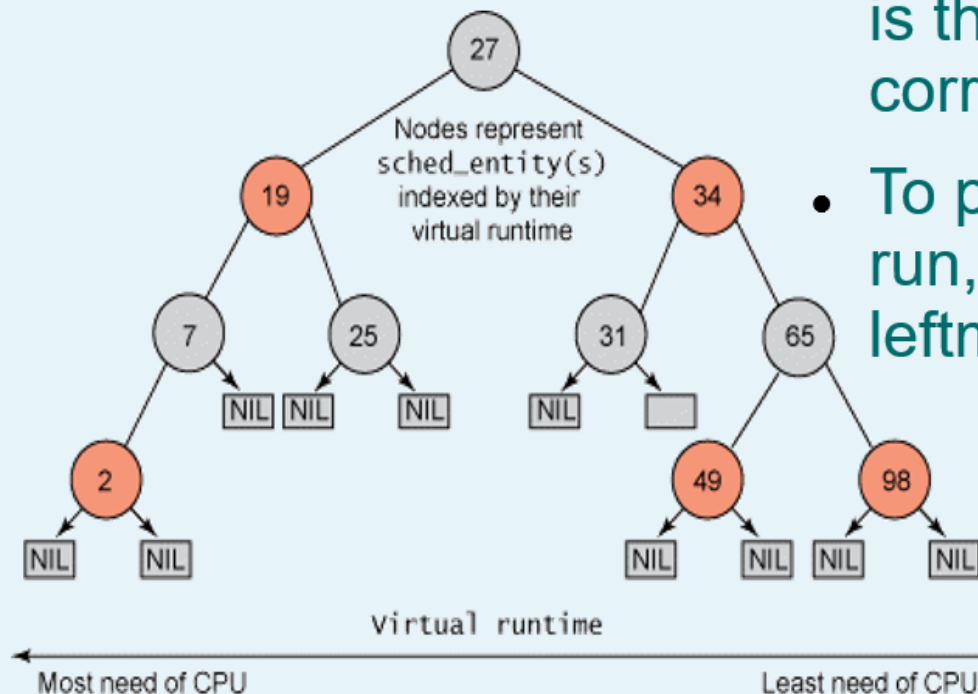
uzel je červený nebo černý, kořen je černý

všechny listy jsou černé

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



# The CFS Tree



- The key for each node is the vruntime of the corresponding task.
- To pick the next task to run, simply take the leftmost node.

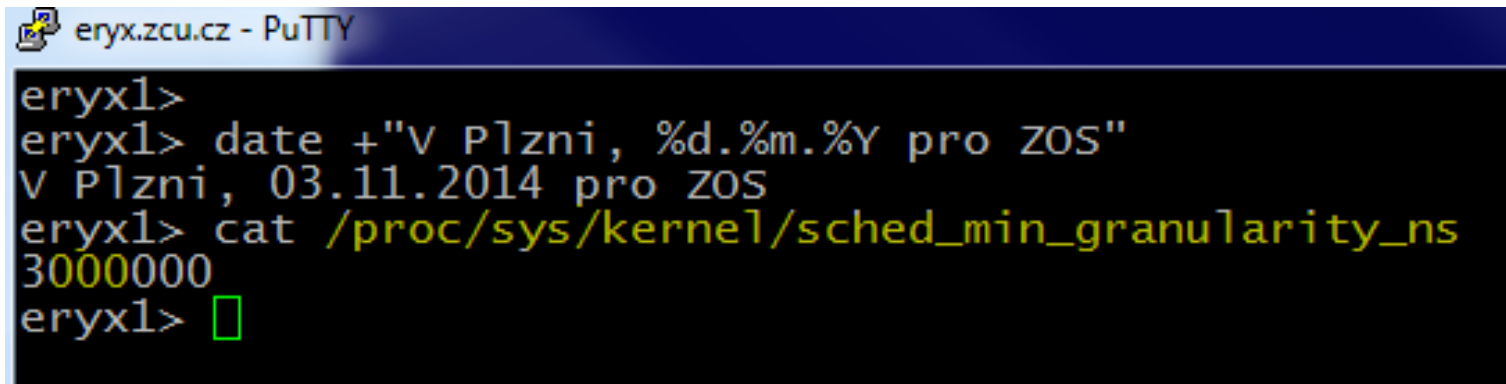
<http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>



# CFS - poznámky

---

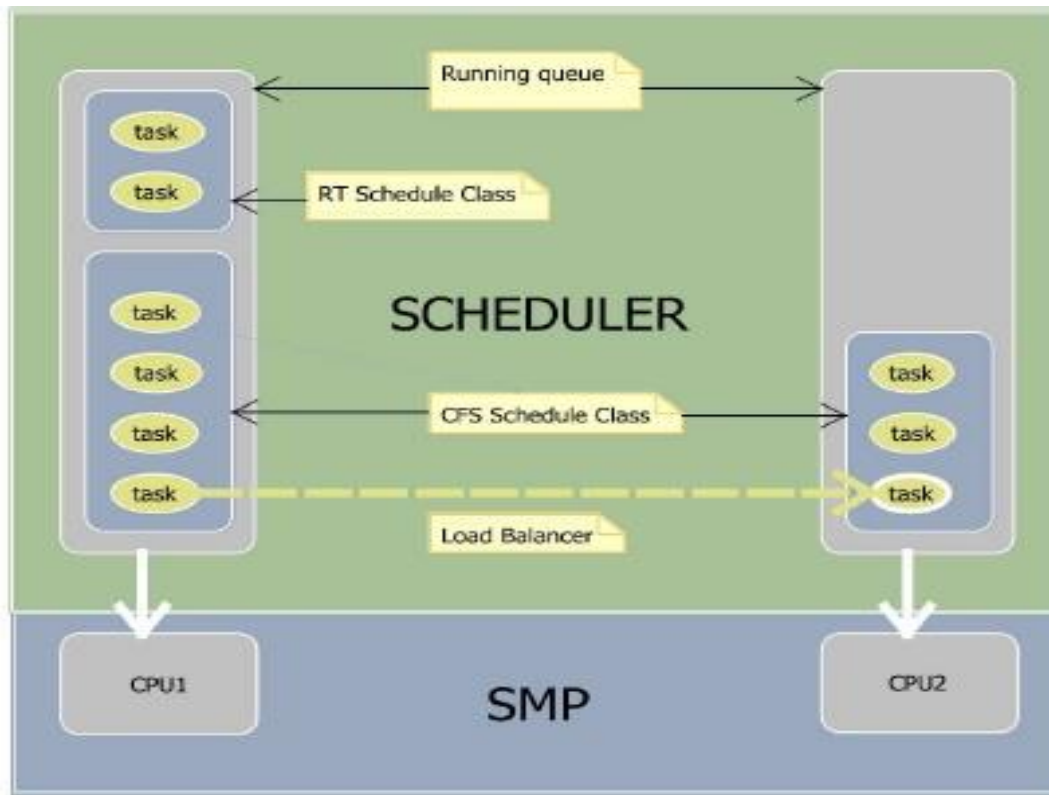
cat /proc/sys/kernel/sched\_min\_granularity\_ns



```
eryx.zcu.cz - PuTTY
eryx1>
eryx1> date +"V Plzni, %d.%m.%Y pro ZOS"
V Plzni, 03.11.2014 pro ZOS
eryx1> cat /proc/sys/kernel/sched_min_granularity_ns
3000000
eryx1> █
```

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

# Linuxový plánovač



Run queue  
pro každý procesor

Schedule class  
(plánovací třídy)  
- CFS  
- RT

Load Balancer  
přesun úloh z vytíženého  
CPU na nevytížené CPU

# 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/cgi/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



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

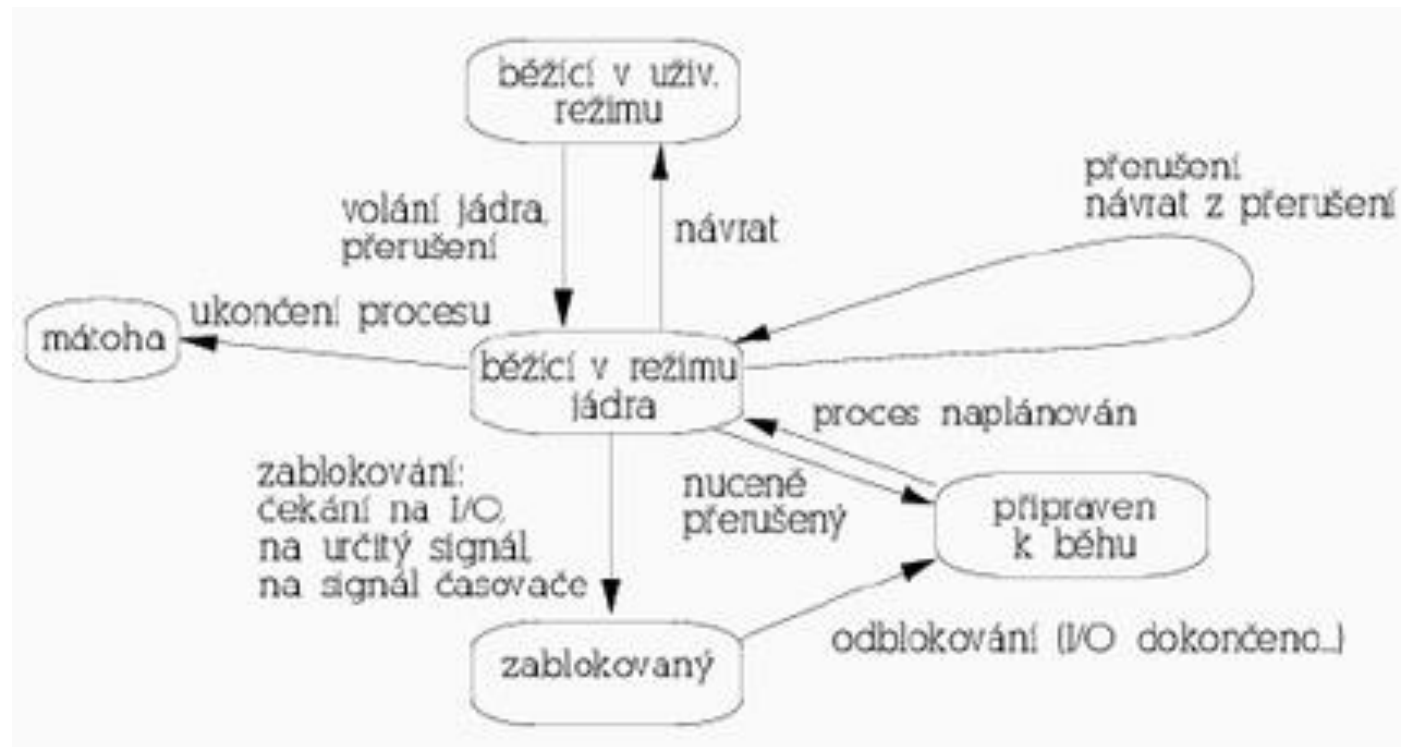
answered Apr 8 '13 at 12:15



Michael Foukarakis

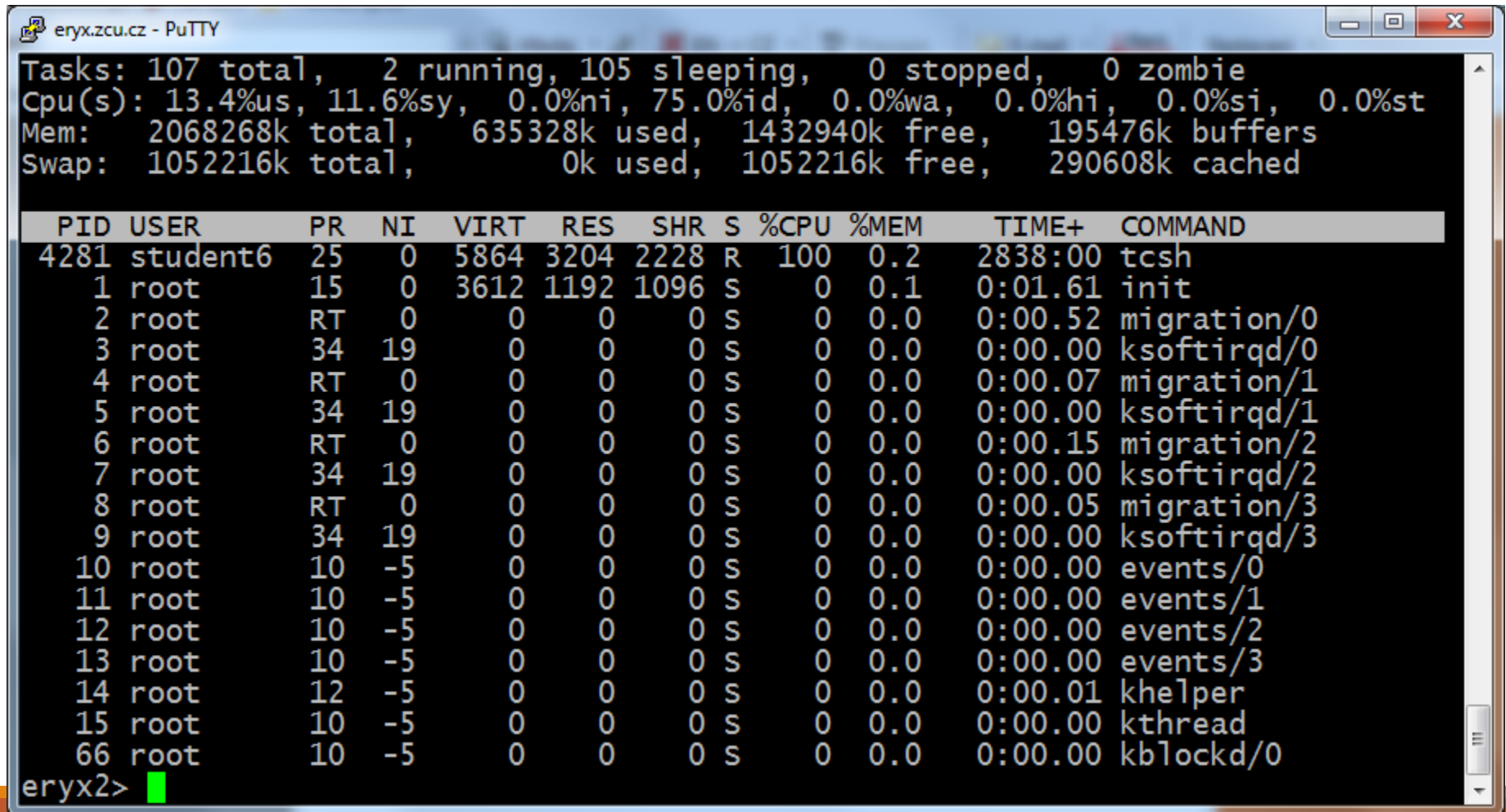
15.4k ● 2 ● 38 ● 64

# Linux – stavy procesů



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

# Linux – příkaz top



```
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 procesem
11. TIME – celkový procesorový čas
12. COMMAND - příkaz

# Příkaz nice

---

## Změna priority procesu

- běžný uživatel: 0 až +19, tedy pouze zhoršovat prioritu
- 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: syntaxe záleží i na shellu, který používáme.*



# 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 – prováděný syscall se zruší – návrat do userspace
    - obsluha signálu
    - znovu zavolá přerušené systémové 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

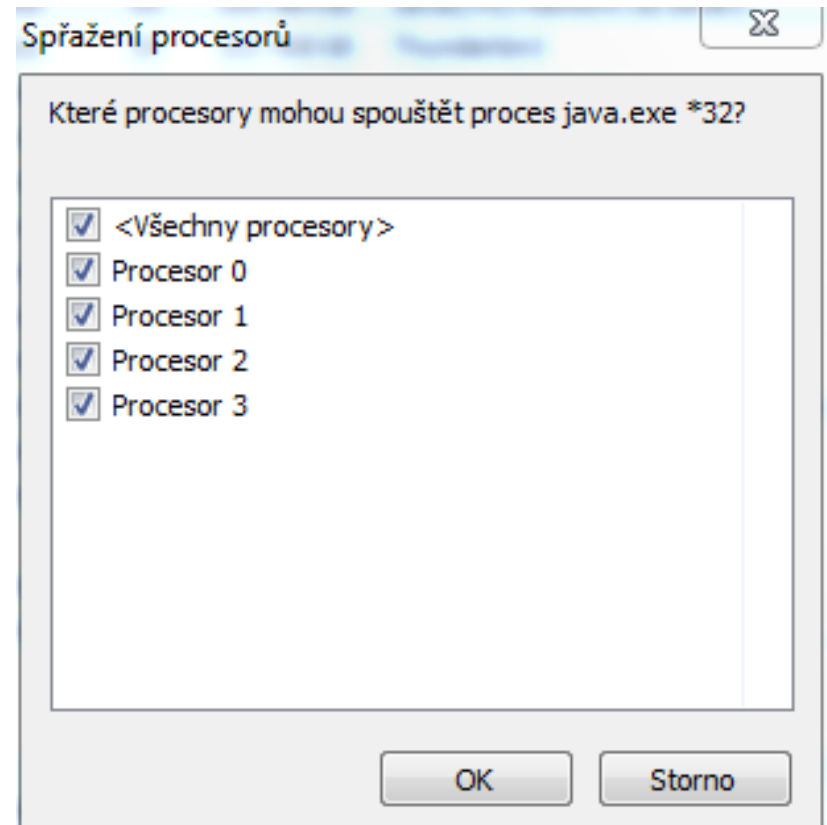
## Přiřazení procesů procesorům

- **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
  - Častější

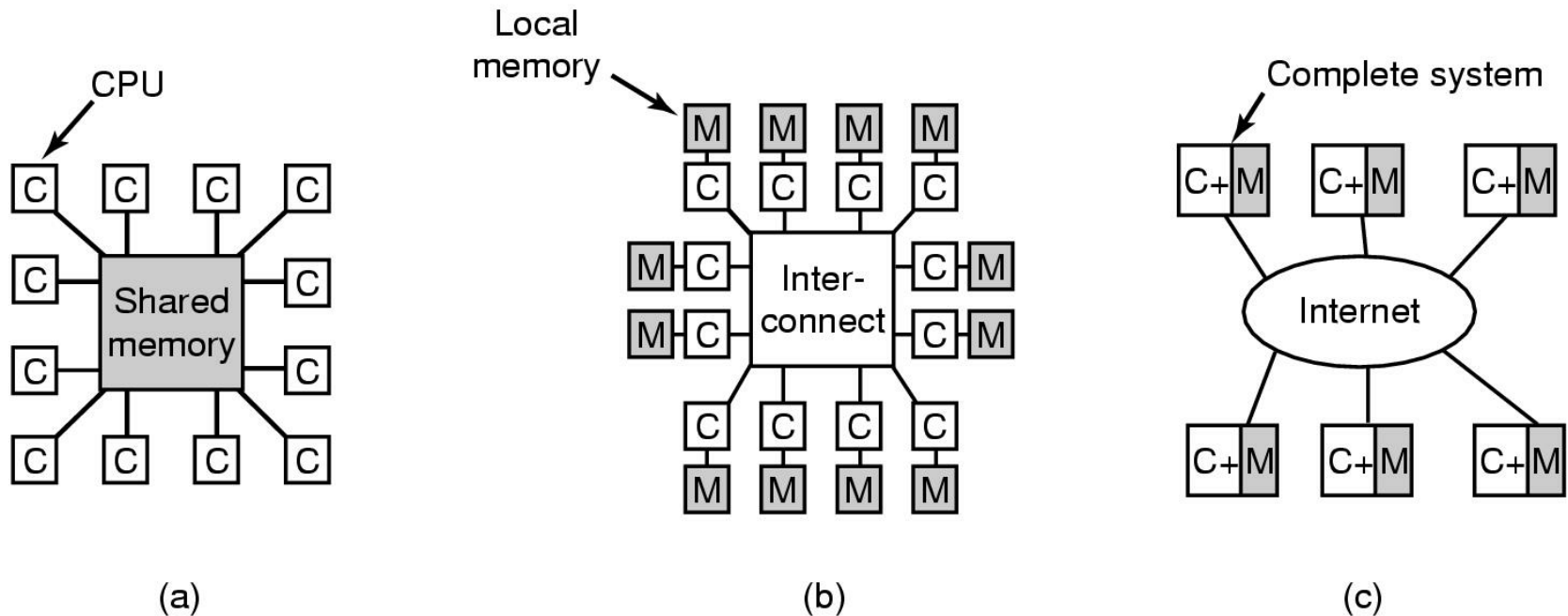
# afinita

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**

# Systémy RT

---

- Události, na které reálný proces reaguje
  - **Aperiodické** – nastávají **nepredikovatelně**
  - **Periodické** – v pravidelných **intervalech**

V systému běží procesy, které je třeba periodicky plánovat, aby splnily své deadliny (doba, do které zareagují).

Stejně tak musí být systém připraven reagovat na aperiodickou událost.

- Zpracování události vyžaduje čas
- Pokud je možné všechny události včas zpracovat  
=> systém je **plánovatelný (schedulable)**



# Příklad

---

V RT systému poběží dva procesy:

P1 – potřebuje běžet každé dvě sekundy, doba jeho běhu je půl sekundy

P2 – potřebuje běžet každou sekundu, doba jeho běhu je 0.1 s

Je takový systém možný, aby byl schopen obsloužit i další události?

Co když bychom chtěli spustit další proces

P3 – potřebuje běžet každou sekundu, doba jeho běhu je 0.5s

Pustíme ho do systému? Odůvodněte!

# 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

- Plánovací rozhodnutí **před spuštěním** systému
- Předpokládá **dostatek informací** o vlastnostech procesů
- Speciální použití na míru danému systému

## ■ 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 vláken

---

## Vlákná plánována OS

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

# Plánování vláken

---

## Vlákna plánována uvnitř procesu

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

# Dispatcher

---

## Dispatcher

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

### Provede:

- 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

multiprocessor – pokud lze, nestřídat procesory

nastavení priority uživatelem

# Poznámka - simulace

---

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

# Uváznutí - definice

---

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

Všichni čekají – nikdo událost nevygeneruje, nevzbudí jiný proces

Obecný termín zdroj – označuje zařízení, záznam, aj.

# 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

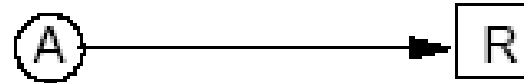
hrany

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

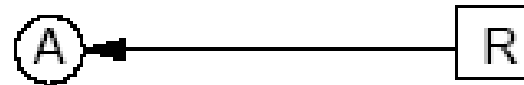
# Modelování uvíznutí

---

proces A čeká na zdroj R:



zdroj R je držen procesem A:



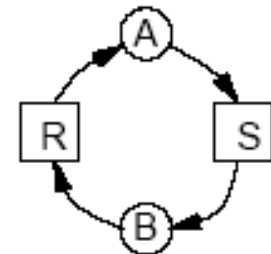
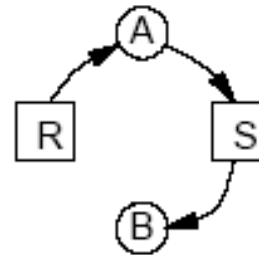
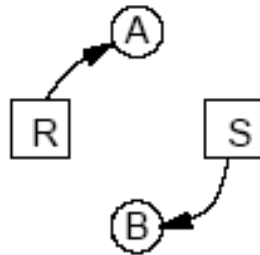
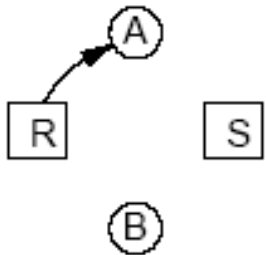
**Cyklus** v grafu → nastalo **uvíznutí**

**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





# Uvívnutí - poznámky

---

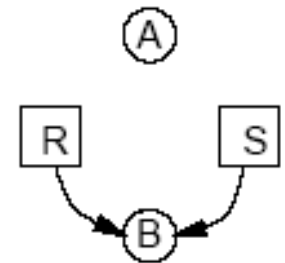
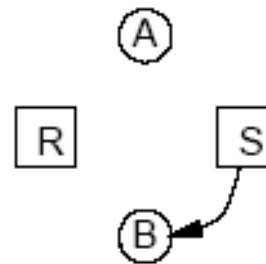
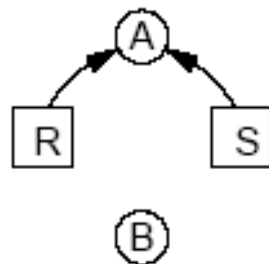
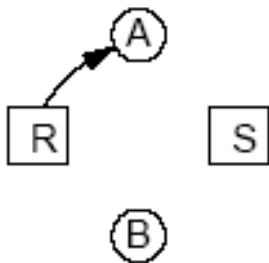
- Cyklus v grafu – nutnou a postačující podmínkou pro vznik uvívnutí.
- Závisí na pořadí vykonávání instrukcí procesů.
- Pokud nejprve alokace a uvolnění zdrojů procesu A, potom B => uvívnutí nenastane.

# Uváznutí – ne vždy musí nastat

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

