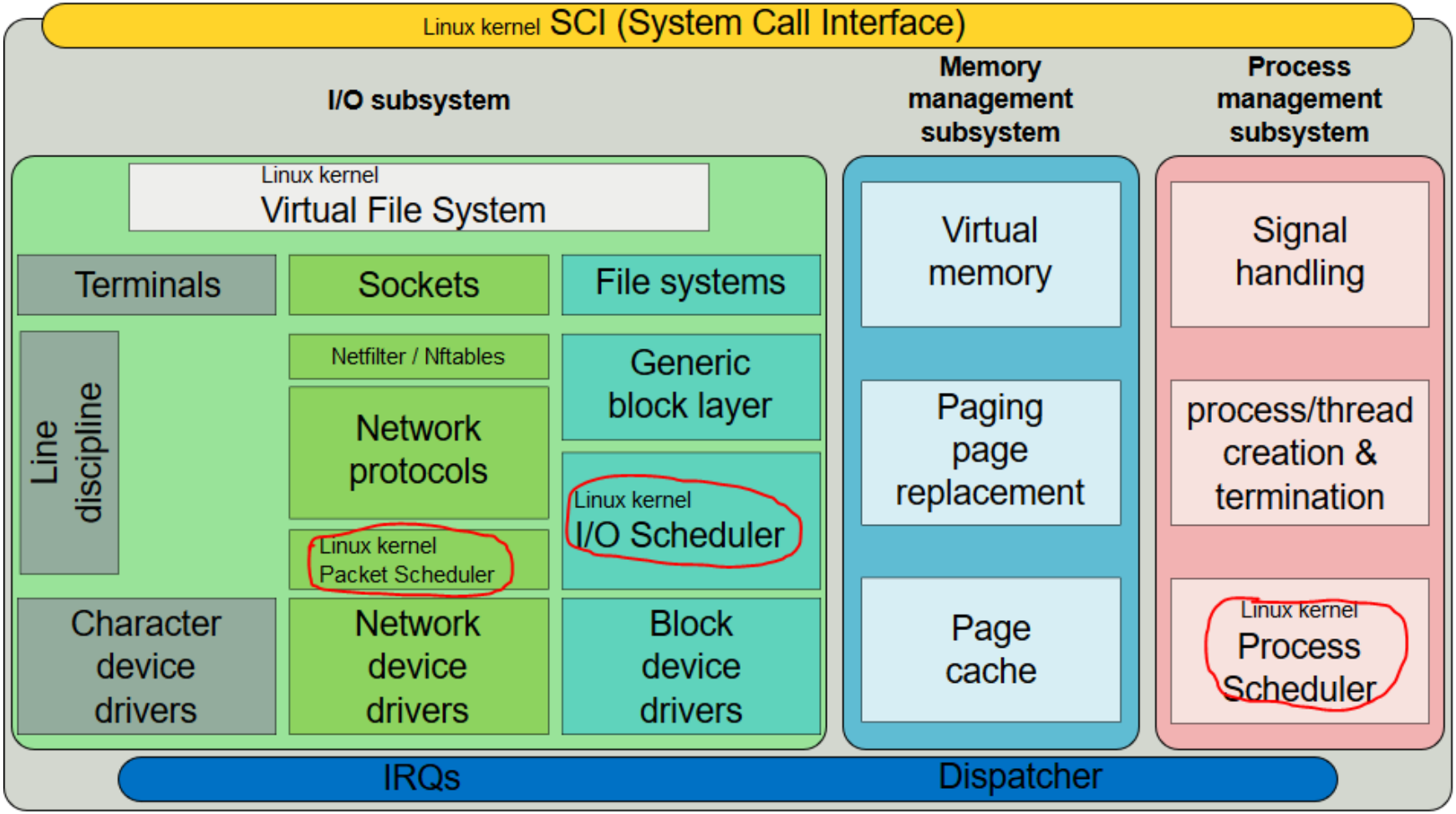
09. Plánování procesů Deadlock

ZOS 2024, L. PEŠIČKA

Linux

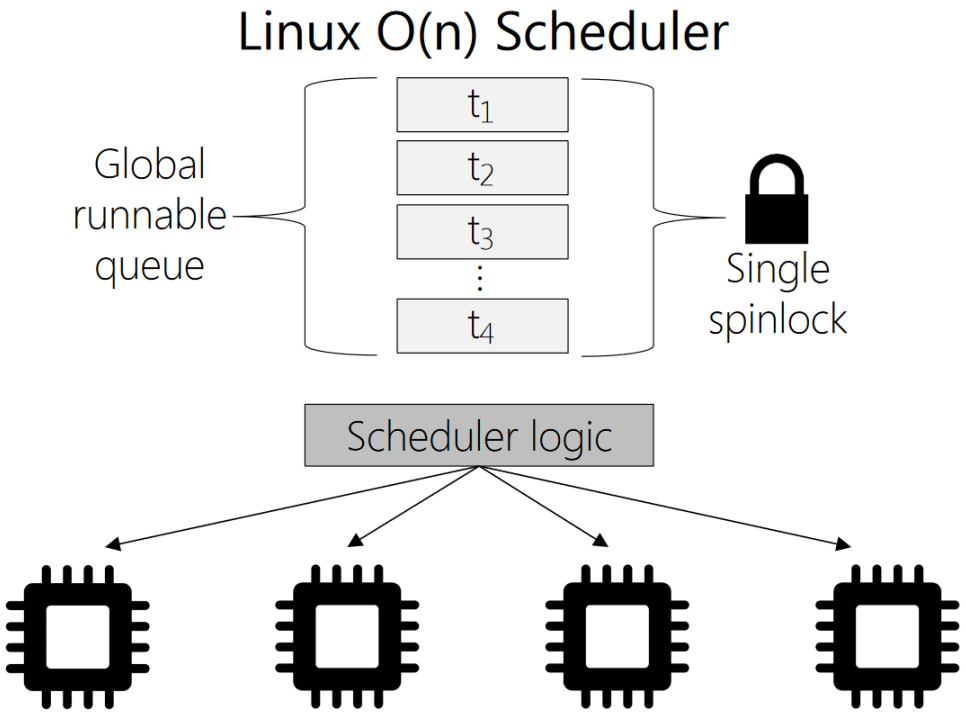


Zdroj obrázku: prezentace na Internetu

Linux – vývoj plánovače

- jádro 2.4
  - **O(n) plánovač**
  - Globální runque (fronta připravených)
  - úloha může být na libovolném CPU (vybalancování využití CPU ok, ale ne pro cache)
  - procházíme frontu úloh
- jádro 2.6 nižší verze
  - **O(1) plánovač**
  - Runque pro každé CPU
  - pole 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** – O(log n), jiný přístup, nejsou pole úloh atd.

Linux – plánovač O(n)



Procházíme frontu připravených úloh a vybereme nejvhodnější k běhu (s nejlepší hodnotou „goodnes“)

Pokud všechny tasky mají goodnes == 0, začne nová epocha (dostanou čas v nové epoše, přepočet)

Zdroj: http://www.eecs.harvard.edu/~cs161/notes/scheduling-case-studies.pdf

Priority úlohy

- Statická priorita
  - Základní priorita
    - Pro realtimové úlohy (sched priority 1 až 99): fronta 0 až 99
    - Pro normální úlohy (sched priority 0): fronta 100 až 139 (defaultně fronta 120)
  - Nice priorita
    - Defaultně má hodnotu 0 (fronta 120)
    - Lze nastavit -20 až 19 (příkaz nice), vede na fronty 100..139
- Dynamická priorita
  - Odvozena ze statické priority
  - Pro normální úlohy dle následujícího vzorce:
  - Dynamická = max (100, min (staticka – bonus+5,139))
  - Bonus je 0 až 10, <5 penalty, 5 stejně, >5 zlepší prioritu
  - Bonus odvozen od average sleep time

Plánovač vybere realtimovou úlohu s nejvyšší prioritou, pokud žádná není, tak normální úloha

Celkem máme 140 front, 100 pro realtimové úlohy, 40 pro běžné úlohy

Epocha

Epocha

- Plánovač dělí čas na epochy
- Dva seznamy – active, expired epocha
- Na začátku epochy dostane proces přidělený time slice
- Time slice – zbývající CPU čas v rámci dané epochy
- Když vyčerpá přidělený time slice – přejde z Active do Expired
- Když v Active nikdo nezbývá, z Active se stane Expired a naopak
- Když jej všechny procesy po částech spotřebují, začíná nová epocha, tedy dostanou nový přidělený čas
- Dlouhé time slice se čerpají po částech (např. po 20ms)

Linux

- 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)
- Většina uživatelských procesů SCHED_OTHER

Tři typy úloh

Realtimové – systémové úlohy, co mají přednost před vším
Dávkové úlohy – dlouhé výpočty bez uživ. interakce
Other – všechny naše běžné interaktivní úlohy

Linux plánování

- V případě soft-realtimových (fronty 0 až 99):
  - SCHED_FIFO (FCFS)
    - Není časové kvantum
    - Přeruší se pouze tehdy, přijde-li úloha s vyšší prioritou
  - SCHED_RR
    - Procesy na stejné úrovni se střídají po určitém časovém kvantu (např. 100 ms)

Linux scheduler

- procesy mají time slice (přidělený čas v epoše)
- fronty
  - 0 – 99 real-time úlohy
  - 100-139 uživatelské úlohy
- Dvě sady 140 front (O1 plánovač – fronty pro každé CPU)
  - active queue
    - Používá se
    - Když je prázdná, vymění se jejich role s expired
  - expired queue
    - Sem přijde proces, když vyčerpá celý svůj time slice

Výpočet time slice

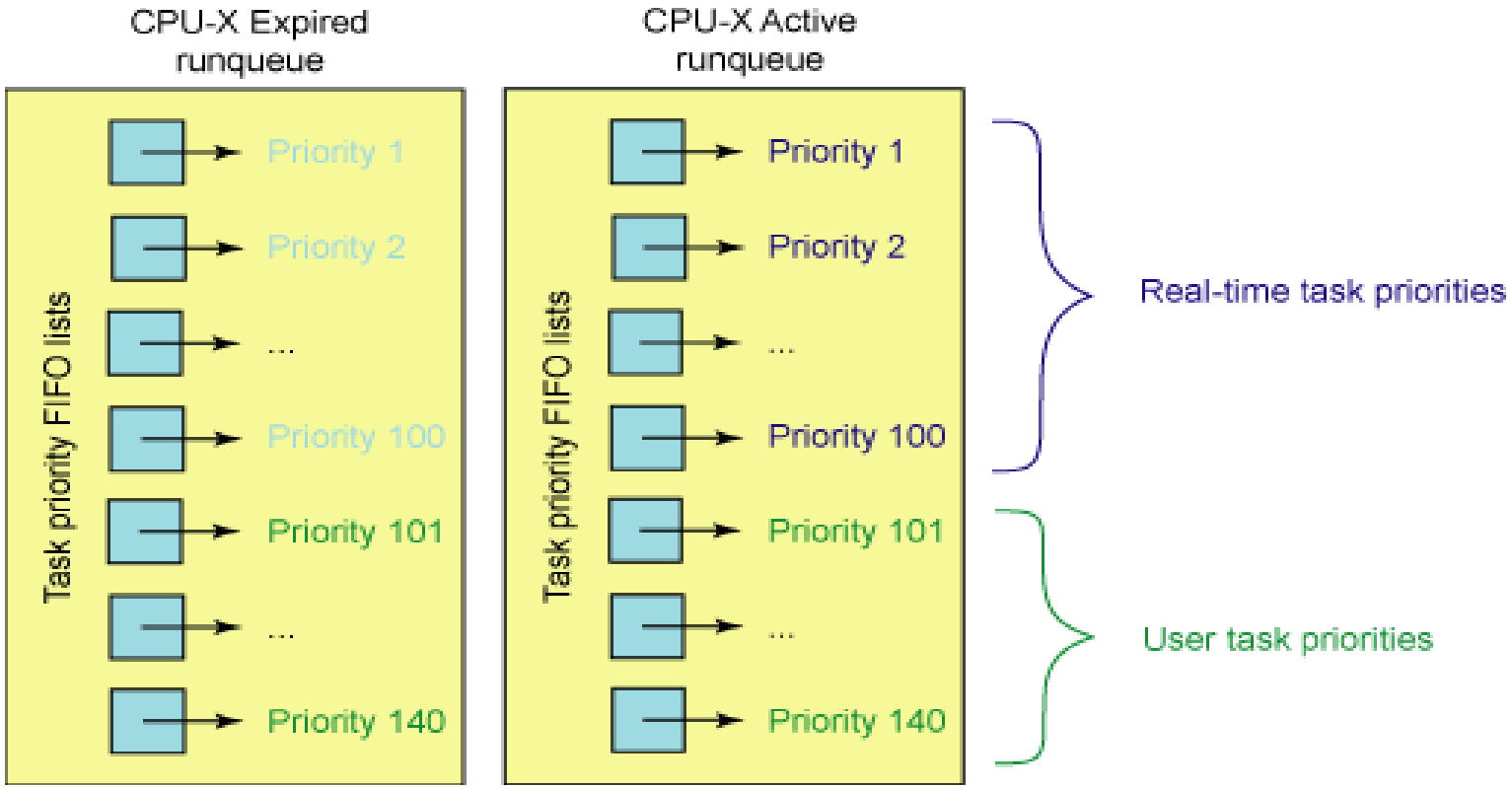
- uživatelské procesy 100 – 139
- SP = statická priorita
- příkaz nice
  - modifikace priority, defaultně 0
  - vylepšit může root (až -20), zhoršit může i běžný uživatel (až +19)

```
if (SP < 120)
        Time slice = (140-SP)*20;
else if (SP>= 120)
        Time slice = (140-SP)*5
```

Priorita	SP	Nice	Time slice
Nejvyšší	100	-20	800 ms
Vysoká	110	-10	600 ms
Normální	120	0	100 ms
Nízká	130	+10	50 ms
Nejnižší	139	+19	5 ms

Dynamická priorita

- Nejen nice ovlivňuje prioritu, ale i bonus
  - Dynamická = max (100, min (staticka – bonus+5,139))
  - Bonus je 0 až 10, <5 penalty, >5 zlepší prioritu
  - Aktuální bonus / penalty podle porovnání
    sleep average oproti konstantnímu maximum sleep average
    (předpoklad: spí dlouho -> čeká na vstup -> interaktivní)
- Systém zohlední dobu, jak dlouho proces neběžel
  - Interaktivní úlohy dostávají bonus
  - Úloha, která více spí – zlepší její prioritu (tj. nižší výsledné číslo)
  - „Čekám dlouho? Asi budu interaktivní úloha, čekám na stisk klávesy. Pak očekává uživatel rychlou reakci.“
- Složité heuristiky plánovače



- Active fronta, Expired fronta
  - 0..139, zde na obrázku označené 1 až 140
  - Na stejné prioritě – round robin

zdroj obrázku:
http://www.ibm.com/developerworks/linux/library/l-scheduler/index.html
další popis:
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 (dlouhá fronta – přesun procesů na jiné CPU)
- pravidelné vybalancování front (řeší speciální vlákno jádra)
  - procesy – bitová maska, na jakém CPU mohou běžet (afinita)
  - maska je děděná child procesy a vlákny
- pole active, expired úloh; když v active nic –> nová epocha

Preempce – zabránit neinteraktivitě

Timeslice distribution:

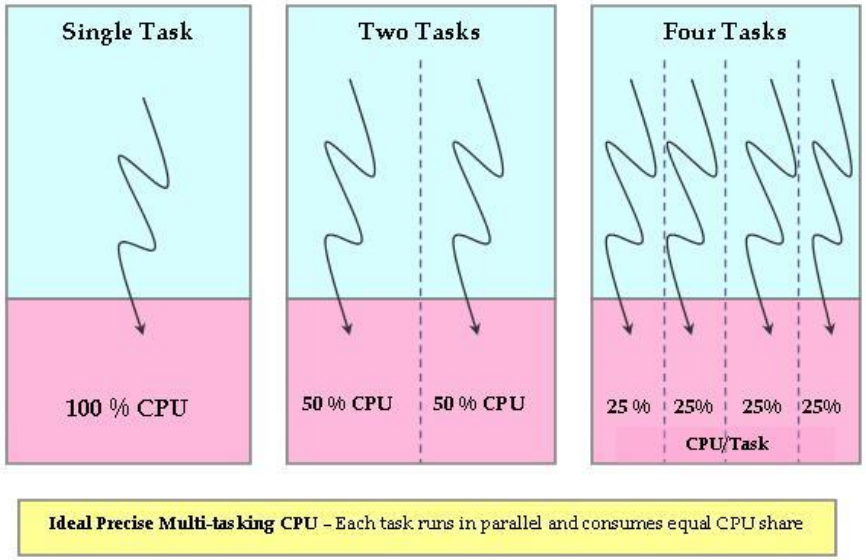
- Priority is recalculated only after expiring a timeslice
- Interactive tasks may become non-interactive during their LARGE timeslices, thus starving other processes
- To prevent this, time-slices are divided into chunks of 20ms
- A task of equal priority may preempt the running task every 20ms
- The preempted task is requeued and is round-robined in it's priority level.
- Also, priority recalculation happens every 20ms

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

Model ideálního CPU

- CFS se snaží vytvořit model ideálního CPU, aby úlohy dostaly spravedlivý podíl výkonu CPU



Zdroj: https://www.linuxjournal.com/node/10267

Vruntime

- **Vruntime** říká, jak moc si úloha zaslouží běžet, oproti ostatním úlohám na stejném procesoru(„kolik toho už napočítal“).
- Nižší číslo vruntime
  –> zatím tolik nenapočítal, zasloužil by si více běžet

- Bere v úvahu celou řadu věcí – prioritu procesu, kolik času už proces na CPU strávil atd.
- Organizováno ve stromu, vybere se vždy nejlevější prvek, tedy s nejnižší hodnotou vruntime.

Příklad

Mějme 2 úlohy:

- Task 1 – renderování videa – neinteraktivní, CPU bound
- Task 2 – textový editor - interaktivní, většinou čeká na stisk klávesy, do fronty připravených se dostane jednou za čas

- Task 1 – je často na CPU, zvyšuje se mu vruntime - množství času, které již napočítal

- Task 2 – většinou čeká, má daleko nižší vruntime – když se dostane mezi připravené úlohy, dostane přednost před taskem 1

CFS – jak řešit priority?

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

CFS – kdy se spouští?

CFS je volán v době:

- Vytvoření nové úlohy
- Probuzení stávající úlohy
- Usnutí úlohy
- V obsluze časovače přerušení

Co udělá:

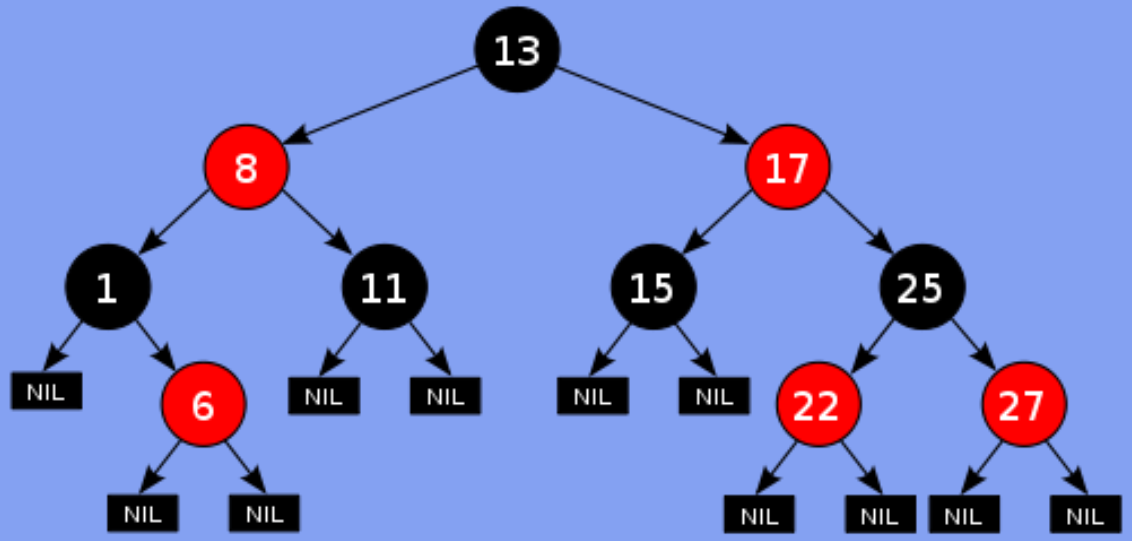
- Spočítá čas právě strávený úlohou na CPU a vynásobí jej koeficientem priority
- Přičte jej do vruntime položky úlohy
- Pokud hodnota již nemá nejmenší vruntime (s jistou granularitou), vystřídá ji úloha nejvíce nalevo v red-black tree

CFS

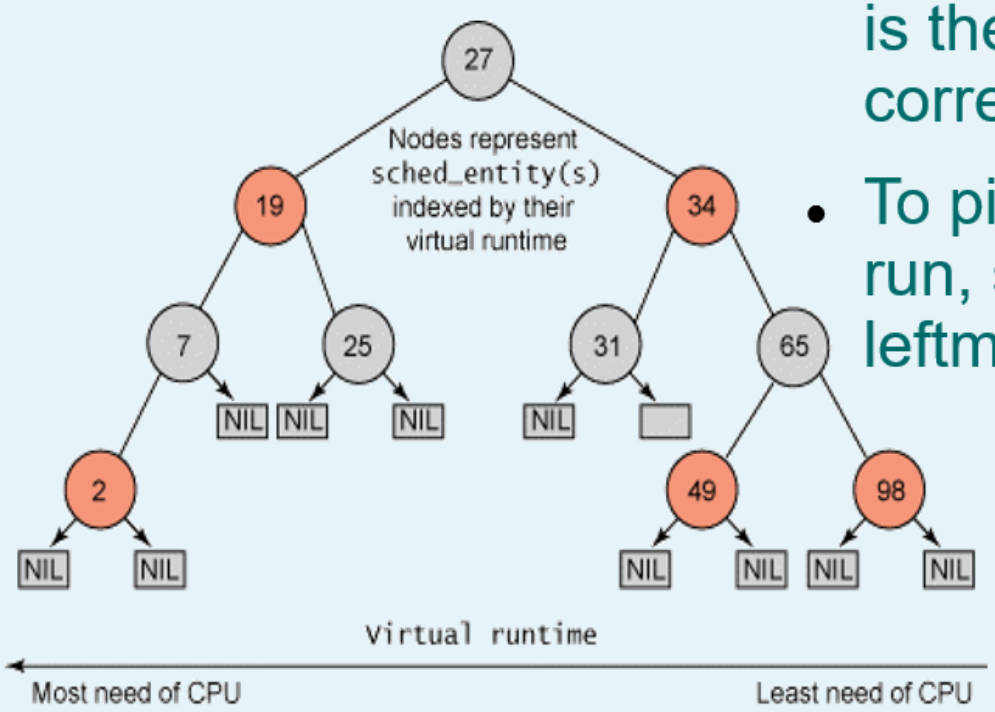
1. Naplánuj task s nejnižším vruntime
   (spent processor time, uzel nejvíce vlevo)
2. Task běží
3. Update vruntime
4. Znovu vložíme do stromu

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



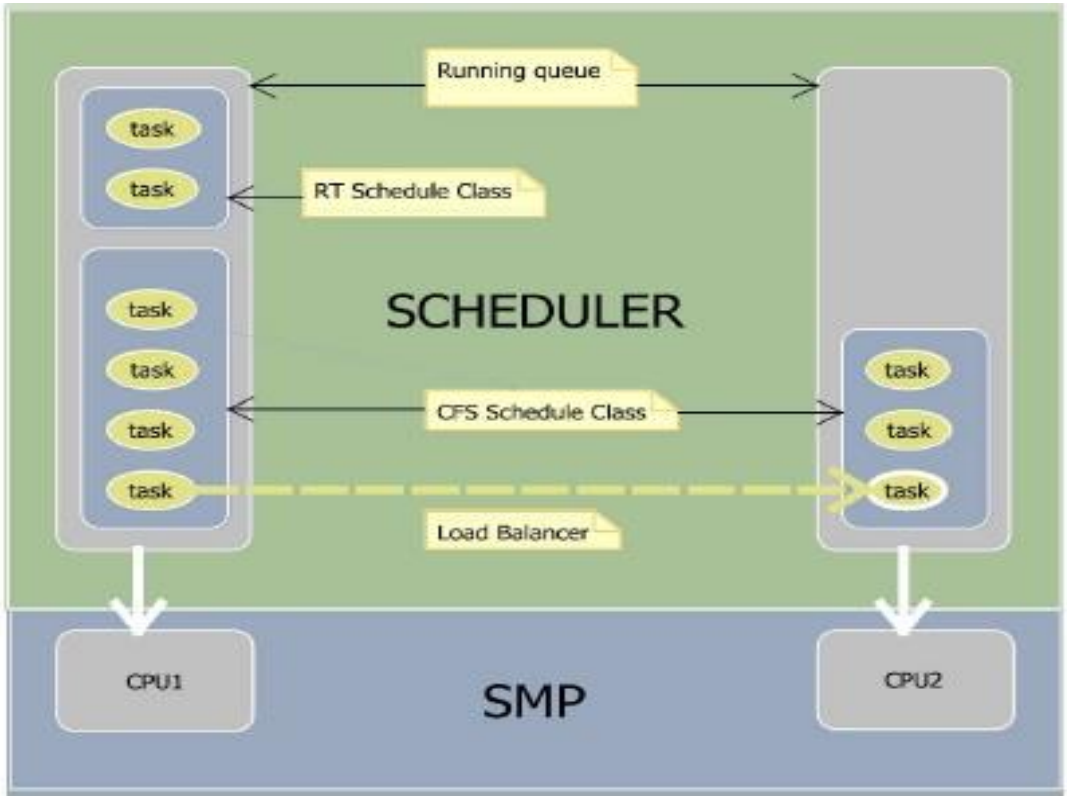
The CFS Tree



- 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/

Linuxový plánovač



Run queue
pro každý procesor

Schedule class
(plánovací třídy)
- CFS
- RT

Load Balancer
přesun úloh z vytíženého CPU na nevytížené CPU

Zdroj obrázku: prezentace na Internetu

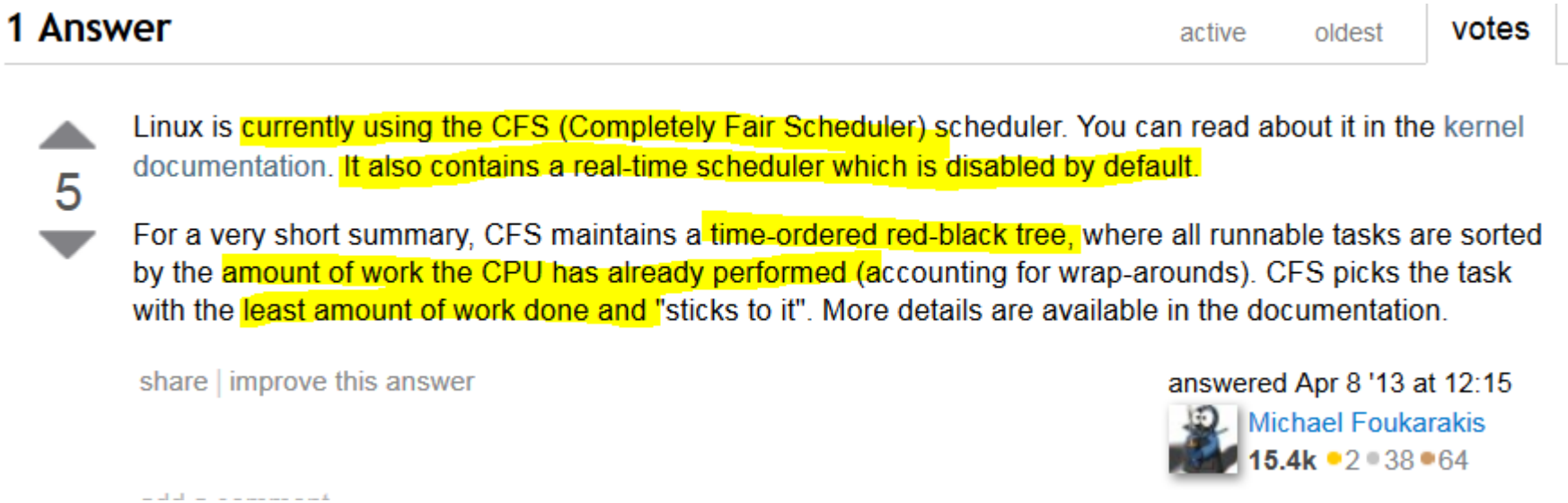
CFS – plánovací politiky

- **SCHED_NORMAL**
  - to co se dříve označovalo jako 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

zkuste v Linuxu: man sched

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

Příkaz nice

Změna priority procesu

- běžný uživatel: 0 až +19, tedy pouze zhoršovat prioritu
- root: -20 („vylepšuje prioritu“) až +19 (nejnižší, „zhoršuje prioritu“)

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

Plánování – víceprocesorové stroje

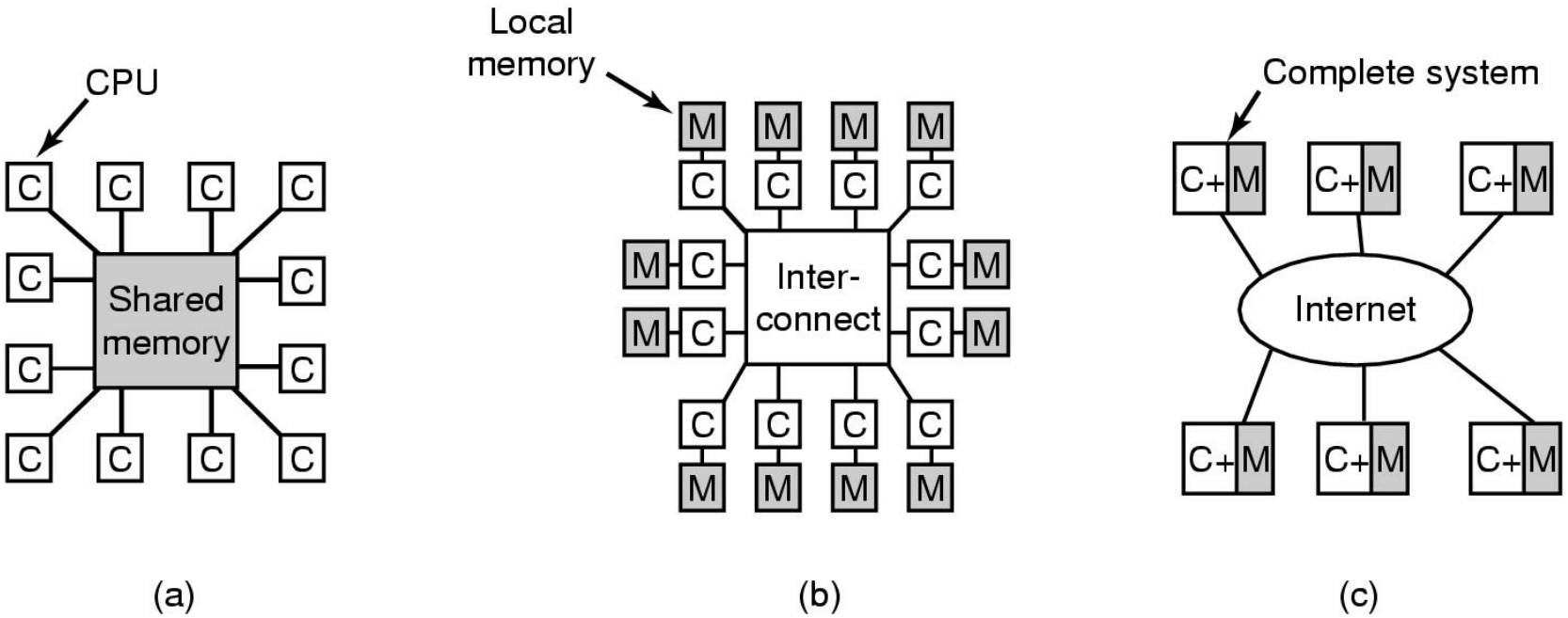
nejčastější architektura

- těsně vázaný symetrický multiprocesor
- 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
- Balancování
  - Přiřazení na CPU a možnost převést k jinému CPU

Multiprocesorové systémy



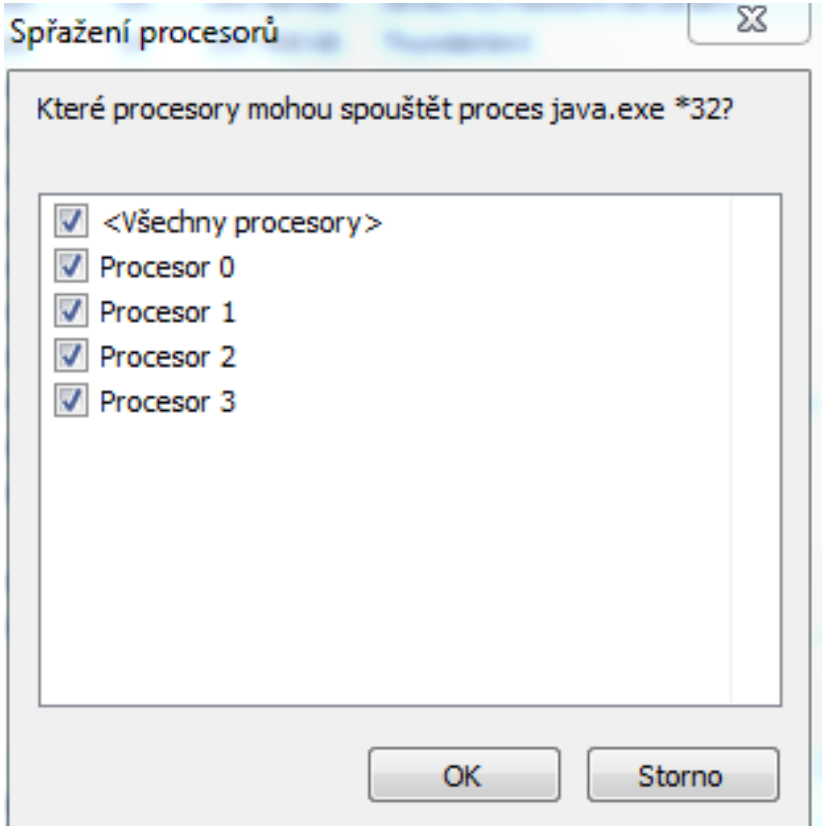
Architektury:

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

afinita

na jakých CPU může daný proces běžet

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



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 realtimový 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

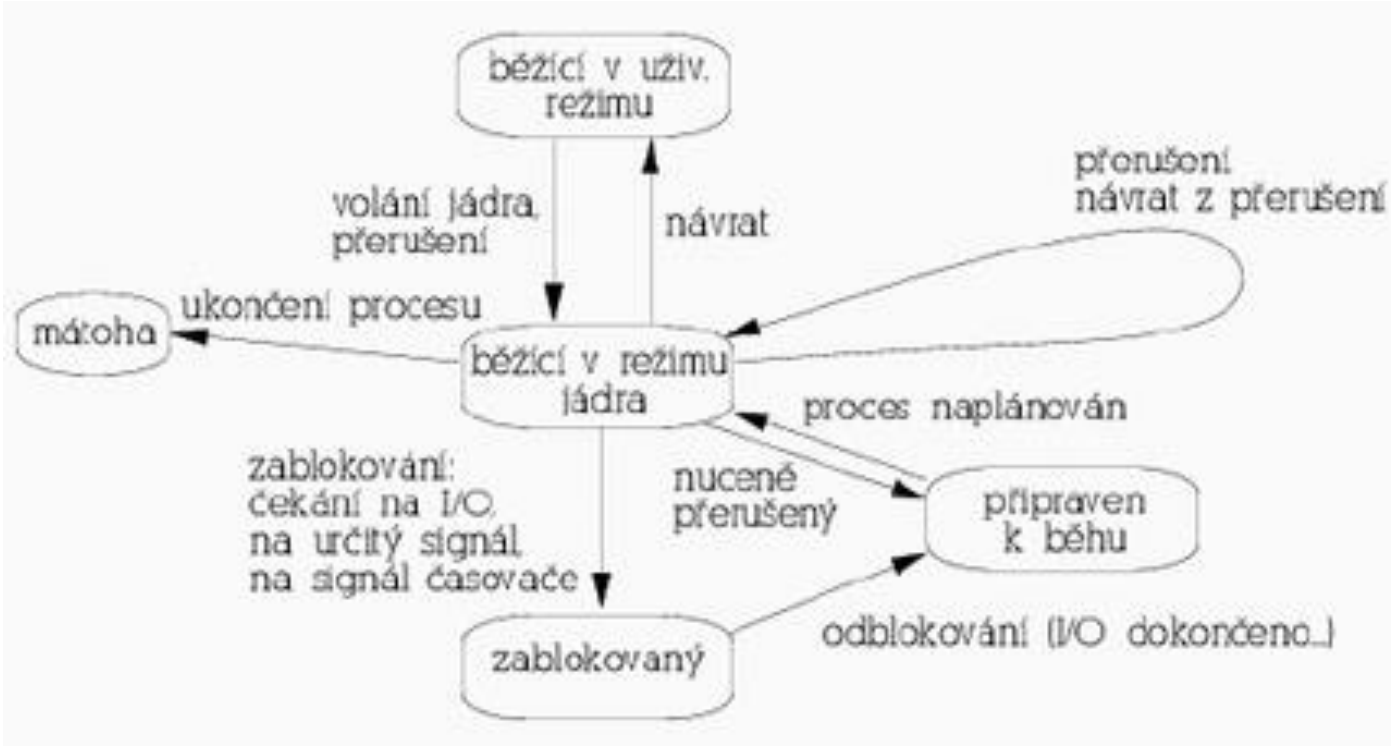
- Statické
  - Plánovací rozhodnutí před spuštěním systému
  - Předpokládá dostatek informací o vlastnostech procesů
  - Speciální použití na míru danému systému
- Dynamické
  - Za běhu
  - Některé algoritmy provedou analýzu plánovatelnosti, nový proces přijat pouze pokud je výsledek plánovatelný

Vlastnosti současných RT

- Malá velikost OS –> omezená funkčnost
- Snaha spustit RT proces co nejrychleji
  - Rychlé přepínání mezi tasky (procesy, vlákna)
  - 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 používají jen sekvenční soubory (rychlejší)

Zpátky obecně k plánování procesů

Linux – stavy procesů detailně



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

Proces – stav blokovaný (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 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)
- Používá Linux, Windows

Plánování vláken

Vlákna plánována uvnitř procesu

- Vlákna v user space, OS o nich neví
- Běží v rámci času, který je přidělen procesu
- Přepínání mezi vlákny – systémová knihovna
- Pokud OS neposkytuje procesu pravidelné “přerušení“, tak pouze nepreemtivní plánování
- Obvykle algoritmus RR nebo prioritní plánování
- Menší možnosti

Dispatcher

Dispatcher

- Modul, který předá řízení CPU vybranému procesu 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

multiprocesor – 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 !!**

Uvíznutí – zamykání záznamů v databázi, semafory

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 (preemtable)

- lze je odebrat procesu bez škodlivých efektů

nepřeplánovatelné (nonpremeptable)

- 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

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

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

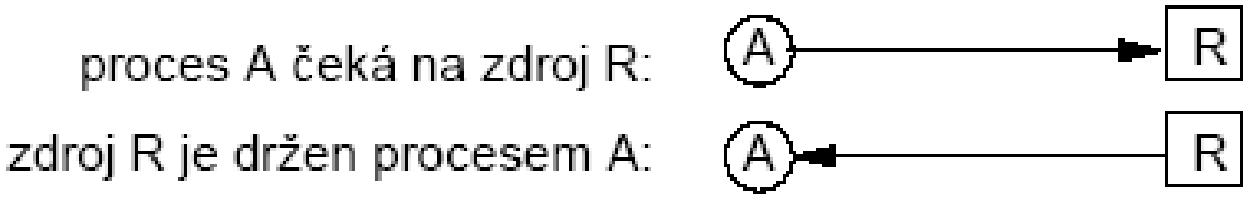
2 typy uzlů

- 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



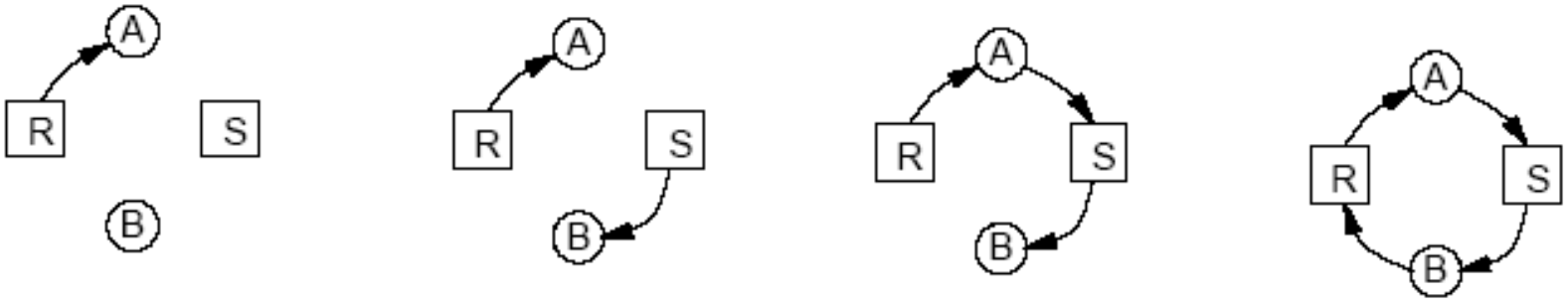
Cyklus v grafu –> nastalo uvíznutí

Uvíznutí se týká procesů a zdrojů v cyklu.

Uvíznutí

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 - uvíznutí



Uvíznutí - poznámky

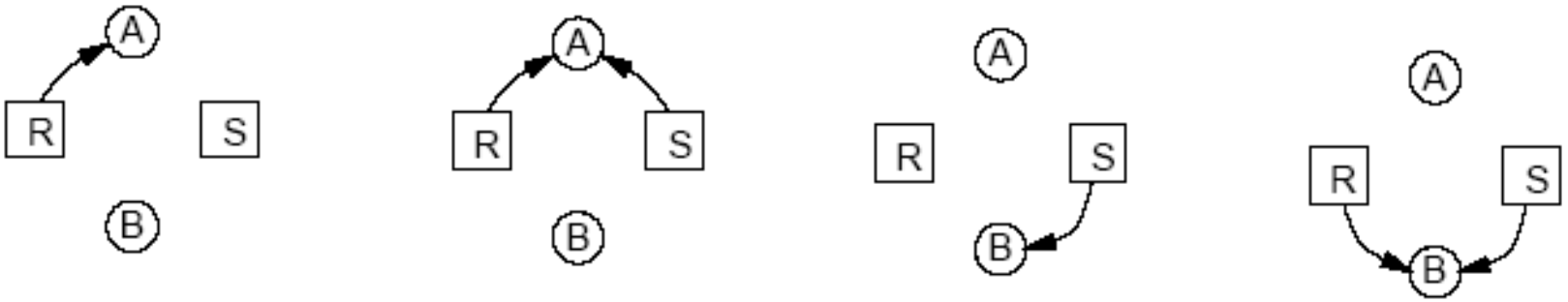
- Cyklus v grafu – nutnou a postačující podmínkou pro vznik uvíznutí.
- Závisí na pořadí vykonávání instrukcí procesů.
- Pokud nejprve alokace a uvolnění zdrojů procesu A, potom B => uvíznutí nenastane.

Uvíznutí – 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 uvíznutí

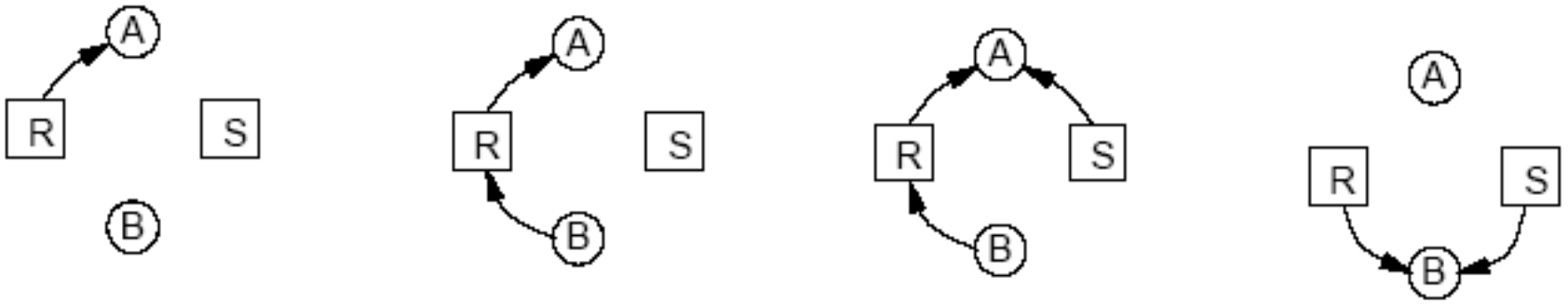
Při některých bězích nemusí uvíznutí nastat – hůře se hledá chyba.



Uvíznutí – pořadí alokace

Pokud bychom napsali procesy A,B tak, aby oba žádaly o zdroje R a S ve stejném pořadí – uvíznutí 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



Jak se vypořádat s uvíznutím

1. Problém uvíznutí je zcela ignorován
2. Detekce a zotavení
3. Dynamické zabránění pomocí pečlivé alokace zdrojů
4. Prevence, pomocí strukturální negace jedné z dříve uvedených nutných podmínek pro vznik uvíznutí

1. Ignorování problému

- Předstíráme, že problém neexistuje ☺
  - „pštrosí algoritmus“
- Proč neřešit? vysoká cena za eliminaci uvíznutí
  - Neexistuje žádné univerzální řešení
  - Např. činnost uživatelských procesů by byla omezena
- Žádný ze známých OS se nezabývá uvíznutím uživatelských procesů
  - Snaha o eliminaci uvíznutí pro činnosti jádra

U uživatelských procesů uvíznutí neřešíme, snažíme se, aby k uvíznutí nedošlo v jádře OS

2. Detekce a zotavení

- Systém se nesnaží zabránit vzniku uvíznutí
- Detekuje uvíznutí
- Pokud nastane, provede akci pro zotavení

Samotná detekce uvíznutí nemusí být snadná

- Detekce pro 1 zdroj od každého typu
  - Při žádostech o zdroj OS konstruuje graf alokace zdrojů
  - Detekce cyklu – pozná, zda nastalo uvíznutí
  - Různé algoritmy detekce cyklu (teorie grafů)
    - Např. prohledávání do hloubky z každého uzlu, dojdeme-li do uzlu, který jsme již prošli - cyklus

Zotavení z uvíznutí – odebrat zdroj?

Zotavení pomocí preempce

- Vlastníkovi zdroj dočasně odejmout
- Závisí na typu zdroje – často obtížné či nemožné
  - Tiskárna – co by to obnášelo:
    po dotištění stránky proces zastavit, ručně vyjmout již vytištěné stránky, odejmout procesu a přiřadit jinému
    (tedy prakticky nepoužitelné)

Zotavení z uvíznutí – zrušení změn

Zotavení pomocí zrušení změn (rollback)

- **Častá uvíznutí** – checkpointing procesů
  = zápis stavu procesů do souboru, aby proces mohl být v případě potřeby vrácen do uloženého stavu

- **Detekce uvíznutí** – nastavení na dřívější checkpoint, kdy proces ještě zdroje nevlastnil (následná práce ztracena)

- Zdroj přiřadíme uvízlému procesu – zrušíme deadlock
- Proces, kterému jsme zdroj odebrali – pokusí se ho alokovat, ale má jej jiný - usne

Zotavení z uvíznutí – zrušení procesu

Zotavení pomocí zrušení procesu

- Nejhorší způsob – zrušíme jeden nebo více procesů
- Zrušit proces v cyklu
  - Pokud nepomůže zrušit jeden, zrušíme i další

- Často alespoň snaha zrušit procesy, které je možné spustit od začátku.

3. Dynamické zabránění

- Ve většině systémů procesy žádají o zdroje po jednom
- Systém rozhodne, zda je přiřazení zdroje bezpečné, nebo hrozí uvíznutí
- Pokud bezpečné – zdroj přiřadí, jinak pozastaví žádající proces
- Stav je bezpečný, pokud existuje alespoň jedna posloupnost, ve které mohou procesy doběhnout bez uvíznutí
- I když stav není bezpečný, uvíznutí nemusí nutně nastat

Bankéřův algoritmus pro jeden typ zdroje

- Předpokládáme více zdrojů stejného typu
  - Např. N magnetopáskových jednotek

- Algoritmus plánování, který se dokáže vyhnout uvíznutí (Dijkstra 1965)
- Bankéř na malém městě, 4 zákazníci – A, B, C, D
- Každému garantuje půjčku (6, 5, 4, 7) = 22 dohromady
- Bankéř ví, že všichni zákazníci nebudou chtít půjčku současně, pro obsluhu zákazníků si ponechává pouze 10

Bankéřův algoritmus

Zákazník	Má půjčeno	Max. půjčka
A	1	6
B	1	5
C	2	4
D	4	7

Bankéř má volných prostředků: 10 – (1+1+2+4) = 2

Stav je **bezpečný**, bankéř může obsloužit všechny požadavky v nějaké posloupnosti - pozastavit všechny požadavky kromě C:

Dá C 2 jednotky, C skončí a uvolní 4, může použít pro D nebo B atd. až obslouží VŠECHNY.

Bankéřův algoritmus (B o 1 více)

Zákazník	Má půjčeno	Max. půjčka
A	1	6
B	2	5
C	2	4
D	4	7

Dáme B o jednotku více; zůstane nám volných prostředků: 1

Stav není bezpečný – pokud všichni budou chtít maximální půjčku, bankéř nemůže uspokojit žádného – nastalo by uvíznutí
Uvíznutí nemusí nutně nastat, ale s tím bankéř nemůže počítat …

Rozhodování bankéře

Zkusí „jako by“ přidělit zdroj a zkoumá, zda je nový stav bezpečný

- U každého požadavku – zkoumá, zda vede k bezpečnému stavu:
- Bankéř předpokládá, že požadovaný zdroj byl procesu přiřazen a že všechny procesy požádaly o všechny bankéřem garantované zdroje.
- Bankéř zjistí, zda je dostatek zdrojů pro uspokojení některého zákazníka; pokud ano – předpokládá, že zákazníkovi byla suma vyplacena, skončil a uvolnil (vrátil) všechny zdroje
- Bankéř opakuje předchozí krok, pokud mohou všichni zákazníci skončit, je stav **bezpečný.**

Vykonání požadavku

1. Proces požaduje nějaký zdroj

2. Zdroje jsou poskytnuty pouze tehdy, pokud požadavek vede k bezpečnému stavu

3. Jinak je požadavek odložen na později
   – proces je pozastaven

Bankéřův algoritmus pro více typů zdrojů

zobecnění pro více typů zdrojů:

používá dvě matice
(sloupce – třídy zdrojů, řádky – zákazníci)

- matice přiřazených zdrojů (current allocation matrix)
  - který zákazník má které zdroje
- matice ještě požadovaných zdrojů (request matrix)
  - kolik zdrojů kterého typu budou procesy ještě chtít

	Zdroj R	Zdroj S	Zdroj T
Zák. A	3	0	1
Zák. B	0	1	0
Zák. C	1	1	1
Zák. D	1	1	0

Matice přiřazených zdrojů

	Zdroj R	Zdroj S	Zdroj T
Zák. A	1	1	0
Zák. B	0	1	1
Zák. C	3	1	0
Zák. D	0	0	1

Matice ještě požadovaných zdrojů

zavedeme vektor A volných zdrojů (available resources)

např. A = (1, 0, 1) znamená jeden volný zdroj typu R, 0 typu S, 1 typu T

Určení, zda je daný stav bezpečný

1. V matici ještě požadovaných zdrojů hledáme řádek, který je menší nebo roven A.
   Pokud neexistuje, nastalo by uvíznutí.

2. Předpokládáme, že proces obdržel všechny požadované zdroje a skončil. Označíme proces jako ukončený a přičteme všechny jeho zdroje k vektoru A.

3. Opakujeme kroky 1. a 2., dokud všechny procesy neskončí (tj. původní stav byl bezpečný), nebo dokud nenastalo uvíznutí (původní stav nebyl bezpečný)

Bankéřův algoritmus & použití v praxi

- publikován 1965, uváděn ve všech učebnicích OS
- v praxi v podstatě nepoužitelný
  - procesy obvykle nevědí dopředu, jaké budou jejich maximální požadavky na zdroje
  - počet procesů není konstantní (uživatelé se přihlašují, odhlašují, spouštějí procesy, ...)
  - zdroje mohou zmizet (tiskárně dojde papír ...)
- nepoužívá se v praxi pro zabránění uvíznutí
- odvozené algoritmy lze použít pro detekci uvíznutí při více zdrojích stejného typu

4. Prevence uvíznutí

jak skutečné systémy zabraňují uvíznutí?

viz 4 Coffmanovy podmínky vzniku uvíznutí

1. vzájemné vyloučení – výhradní přiřazování zdrojů
2. hold and wait – proces držící zdroje může požadovat další
3. nemožnost zdroje odejmout
4. cyklické čekání

pokud některá podmínka nebude splněna
–> uvíznutí strukturálně nemožné

P1 – Vzájemné vyloučení

- prevence – zdroj nikdy nepřiřadit výhradně
- problém lze řešit pro některé zdroje (tiskárna)
- spooling
  - pouze daemon přistupuje k tiskárně
  - nikdy nepožaduje další zdroje – není uvíznutí
  - převádí soutěžení o tiskárnu na soutěžení o diskový prostor – soutěžení o zdroj, „kterého je více“
  - pokud ale 2 procesy zaplní disk se spool souborem, žádný nemůže skončit
- spooling není možný pro všechny zdroje (záznamy v databázi)

P2- Hold and wait

- proces držící výhradně přiřazené zdroje může požadovat další zdroje
- požadovat, aby procesy alokovaly všechny zdroje před svým spouštěním
  - většinou nevědí, které zdroje budou chtít
  - příliš restriktivní
  - některé dávkové systémy i přes nevýhody používají, zabraňuje deadlocku
- Modifikace:
  pokud proces požaduje nové zdroje, musí uvolnit zdroje které drží a o všechny požádat v jediném požadavku.

P3 – Nemožnost zdroje odejmout

- odejímat zdroje je velmi obtížné
- proces může zanechat zdroj v nekonzistentním stavu

Př: můžete přepisovat dokumentaci k programu na novou verzi, musíte jít od PC a necháte rozepsaný odstavec – v tomto stavu ji nemůžete odevzdat k zápočtu

Př2.: proces bude měnit obsah datové struktury zdroje, a v okamžiku odejmutí zdroje provede jen polovinu plánovaných změn

P4 – Cyklické čekání

- Proces může mít jediný zdroj, pokud chce jiný, musí předchozí uvolnit – restriktivní, není řešení ☹

- Všechny zdroje očíslovány, požadavky musejí být prováděny v číselném pořadí
  - Alokační zdroj nemůže mít cykly
  - Problém – je těžké nalézt vhodné očíslování pro všechny zdroje
  - Není použitelné obecně, ale ve speciálních případech výhodné (jádro OS, databázový systém, …)

Př. Dvoufázové zamykání

- V DB systémech
- První fáze
  - Zamknutí všech potřebných záznamů v číselném pořadí
  - Pokud je některý zamknut jiným procesem
    - Uvolní všechny zámky a zkusí znovu
- Druhá fáze
  - Čtení & zápis, uvolňování zámků
- Zamyká se vždy v číselném pořadí, uvíznutí nemůže nastat

Shrnutí přístupu k uvíznutí (!)

1. Ignorování problému – většina OS ignoruje uvíznutí uživatelských procesů
2. Detekce a zotavení – pokud uvíznutí nastane, detekujeme a něco s tím uděláme (vrátíme čas – rollback, zrušíme proces …)
3. Dynamické zabránění – zdroj přiřadíme, pouze pokud bude stav bezpečný (bankéřův algoritmus)
4. Prevence – strukturálně negujeme jednu z Coffman. podmínek
   - Vzájemné vyloučení – spooling všeho
   - Hold and wait – procesy požadují zdroje na začátku
   - Nemožnost odejmutí – odejmi (nefunguje)
   - Cyklické čekání – zdroje očíslujeme a žádáme v číselném pořadí

Vyhladovění

- Procesy požadují zdroje – pravidlo pro jejich přiřazení
- Může se stát, že některý proces zdroj nikdy neobdrží
  - I když nenastalo uvíznutí !

Př. Večeřící filozofové

- Každý zvedne levou vidličku, pokud je pravá obsazena, levou položí
- Vyhladovění, pokud všichni zvedají a pokládají současně

Vyhladovění 2

Př. Přiřazování zdroje strategií SJF

- Tiskárnu dostane proces, který chce vytisknout nejkratší soubor
- 1 proces chce velký soubor, hodně malých požadavků – může dojít k vyhladovění, neustále předbíhán

1. řešení – použít jinou strategii (FIFO)

2. řešení – označíme požadavek časem příchodu a při překročení povolené doby setrvání v systému bude obsloužen

Terminologie

- Blokovaný (blocked, waiting), někdy: čekající
  - Základní stav procesu v diagramu stavů procesu
- Uvíznutí, uváznutí, deadlock, někdy: zablokování
  - Neomezené čekání na událost
- Vyhladovění, starvation někdy: umoření
  - Procesy běží, ale nemohou vykonávat požadovanou činnost
- Aktivní čekání (busy wait), s předbíháním (preemptive)

Bernsteinovy podmínky

$R(p) \cap W(q) = \emptyset$

$W(p) \cap R(q) = \emptyset$

$W(p) \cap W(q) = \emptyset$

Procesy p, q

Množina dat, kterou daný proces čte nebo zapisuje

Dodatek ke kritickým sekcím
Souběžné čtení je OK

Windows – ukázky funkcí

Správa vláken

```
CreateThread()
SuspendThread(), ResumeThread()
ExitThread()                    // ukončení vlákna
TerminateThread()               // ukončí jiné vlákno

WaitForSingleObject()           // čeká na jeden
WaitForMultipleObjects()        // čeká na 1  nebo všechny
CloseHandle()
```

Windows - synchronizace

Kritické sekce

InitializeCriticalSection()

DeleteCriticalSection()

EnterCriticalSection()

LeaveCriticalSection()

Stejné vlákno může zavolat EnterCriticalSection n-krát, potom ale musí n-krát volat LeaveCriticalSection

viz dokumentace:

kritickou sekci mohou využít pouze vlákna stejného procesu (na rozdíl od mutexu)

optimalizovanější

Windows - synchronizace

Mutexy

Mohou použít vlákna různých procesů

```
CreateMutex()                    // vytvoří mutex
OpenMutex()                      // v jiném procesu otevře
WaitForSingleObject()            // čekáme na mutex
WaitForMultipleObjects()
ReleaseMutex()                   // uvolníme  mutex
CloseHandle()
```

Windows - semafory

Semafory

CreateSemaphore(), // inic. hodnota, max. hodnota

OpenSemaphore(),

WaitForSingleObject(), // operace P()

WaitForMultipleObjects()

ReleaseSemaphore(), // operace V()

CloseHandle()

http://msdn.microsoft.com/en-us/library/windows/desktop/ms686946%28v=vs.85%29.aspx

Windows - synchronizace

Eventy

CreateEvent()

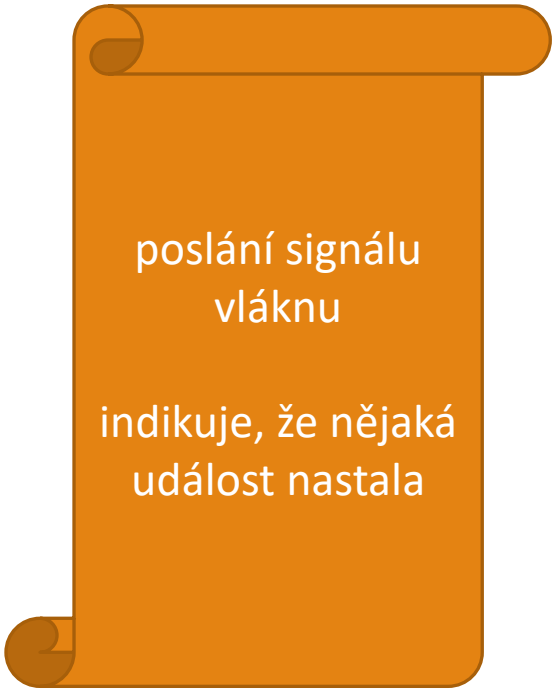
SetEvent()

ResetEvent()

WaitForSingleObject()

WaitForMultipleObjects()

CloseHandle()



Windows

Atomické operace

```
InterlockedIncrement()          // inkrementuje o 1
InterlockedDecrement()          // sníží o 1
InterlockedExchange()           // nastaví novou hodnotu
                                // a vrátí původní
```

Windows

priorita vláken

SetThreadPriority()

GetThreadPriority()

The priority of each thread is determined by the following criteria:

- The priority class of its process
- The priority level of the thread within the priority class of its process

The priority class and priority level are combined to form the *base priority* of a thread.

Jak se určuje priorita?
http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100%28v=vs.85%29.aspx

Windows – podrobný popis

Zde najdete podrobné vysvětlení a použití funkcí:

http://msdn.microsoft.com/en-us/library/windows/desktop/ms685129%28v=vs.85%29.aspx

Windows – Context Switches

Kroky přepnutí kontextu:

1. Save the context of the thread that just finished executing.
2. Place the thread that just finished executing at the end of the queue for its priority.
3. Find the highest priority queue that contains ready threads.
4. Remove the thread at the head of the queue, load its context, and execute it.

Důvody přepnutí kontextu:

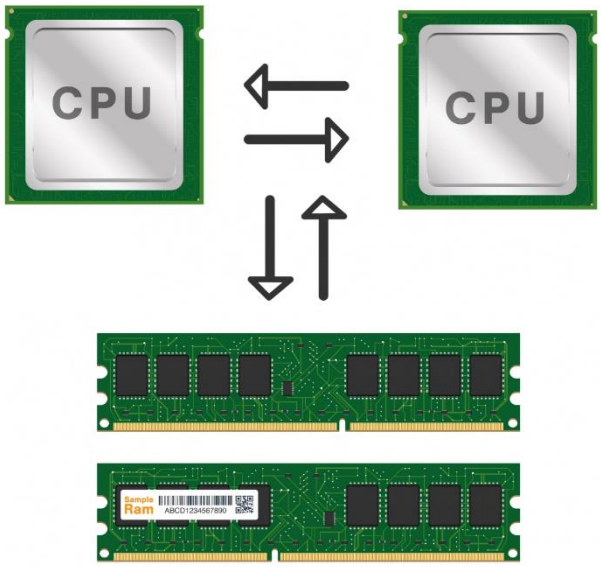
- The time slice has elapsed.
- A thread with a higher priority has become ready to run.
- A running thread needs to wait.

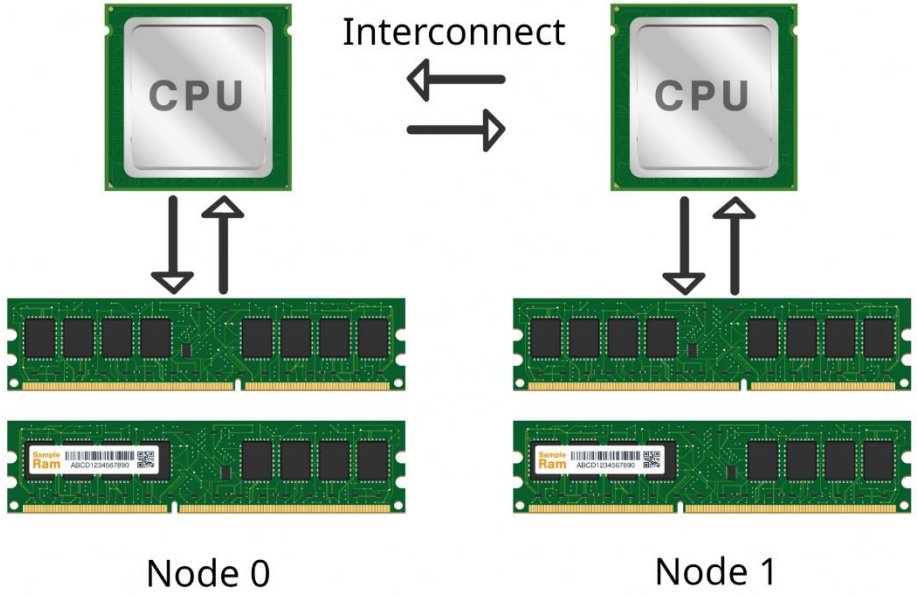
Zajímavý odkaz
https://msdn.microsoft.com/en-us/library/windows/desktop/ms682105(v=vs.85).aspx

Více procesorů

- Symetrický multiprocesor (SMP)
  - Dva nebo více identických procesorů (nebo jader CPU) jsou připojeny k jedné hlavní paměti.
  - Libovolné vlákno lze přidělit libovolnému procesoru.
  - Lze ovlivnit: thread affinity, thread ideal processor
- Non-uniform memory access (NUMA)
  - Každý procesor má blíž k jedné části paměti než ostatní procesory.
  - Systém se pokusí plánovat vlákno na procesoru, který je blízko používané paměti.

UMA vs. NUMA





Zdroj obrázků:
root.cz

NUMA – např. motherboard s více fyzickými procesory (dvě patice na CPU)

Thread Ideal Processor

Když určíme Thread ideal Processor, plánovač k tomuto nastavení přihlédne, pokud je to možné – ale negarantuje naplánování na daném procesoru.

(Srovnejte s affinitou).

Funkce SetThreadIdealProcessor