Plánování
Scheduling

Main Entry:
[1]sched·ule
Pronunciation:
\‘ske-(,)jül, -jəl, *Canada also* ‘she-, *British usually* ‘she-(,)dyül\

Zdroj: //www.merriam-webster.com/

procesor(y) je sdílen více procesy
proces má přidělen procesor na časové kvantum (*time quantum*) – sdílení času (*time sharing*)
pokud neskončí, procesor je přidělen novému procesu – přepnutí procesů

plánování pojednává o tom, kdy vykonat přepnutí procesů, a který proces vybrat

plánovací strategie

> pravidla na určení kdy vykonat přepnutí procesů, a který proces vybrat

implementace plánování

> algoritmy a datové struktury na implementaci plánovací strategie

plánovací strategie

v systémech UNIX/Linux tradičně vychází ze sdílení času

v systému souběžně může běžet několik aplikací

odpovídající procesy podle jejich požadavků můžeme rozdělit na:

- interaktivní procesy
  čekají na stisk klávesy, kliknutí myší, po vstupu se však čeká rychlá reakce, krátká doba odpovědi (*response time*), průměrně mezi 50-150 ms, ale i s omezeným rozptylem, příkladem je shell, editory, grafické aplikace

- dávkové procesy
  nevyžadují bezprostřední interakci s výpočtem, často jsou vykonávány v pozadí, měřítkem efektivnosti plánování je propustnost systému (*troughput*), příkladem jsou kompilátory, vyhledávání v databázích, vědecké výpočty

- procesy reálného času
  mají velmi přísné požadavky na plánování, zaručující krátkou dobu odezvy s minimálním rozptylem, například při zpracování video informace může být upřednostněno zobrazování konstantního počtu 15 snímků za sekundu, namísto zobrazování 10 až 30 snímků za sekundu, s vyšším průměrem 20 snímků za sekundu

alternativní klasifikace tradičně dělí aplikace na:

- vědecko-výzkumné výpočty
  mají vysoké nároky na čas procesoru („*CPU-bound*“)

- zpracování hromadných údajů
  potřebují množství V/V operací a velkou část času stráví čekáním na jejich vykonání („*I/O bound*“)

v systémech UNIX/Linux mají procesy zvláštní třídu(y) plánování (*scheduling class*) pro procesy reálného času a zvláštní třídu(y) plánování pro ostatní procesy

na druhé straně není nikde specifikováno, které procesy požadují především čas procesoru, a které většinou čekají na dokončení V/V operací

plánovací strategie je založena na pořadí, ve kterém procesy požadují o přidělení procesoru a na prioritách, které mají přednost

- procesy reálného času
  statické priority

- ostatní
  dynamické priority
  procesům, které dlouho nečerpají své časové kvantum prioritu zvýšíme a opačně procesům, které jsou dlouho běžící penalizujeme

  interaktivní aplikace budou mít dobrou dobu odezvy

volba časového kvanta

- krátké trvání způsobuje vysokou režii
  je-li např. doba přepnutí mezi procesy 10 ms a časové kvantum je rovněž 10 ms, režie je nejmíň 50%

- příliš dlouhé trvání způsobuje ztrátu zdání souběžného vykonávání procesů
  je-li časové kvantum 4 sekundy, a **`n`** dalších procesů je připraveno, pak bude typicky naplánován za 4***`n`** sekund

- dlouhé trvání časového kvanta nemusí způsobit zvýšení doby odpovědi, procesy které mají interaktivní charakter mají vysokou dynamickou prioritu a rychle vykonají preempci dávkových procesů a to bez ohledu na délku časového kvanta (editor vs. překladač)

- v některých případech však zvýšení doby odpovědi při dlouhém trvání časového kvanta může nastat, například, dva rovnocenní uživatelé **`A`** a **`B`** zadají příkazy, přičemž **`A`** zadal interaktivní a **`B`** zadal dávkový příkaz, shell pro každého z nich vytvoří proces a předpokládejme u nich po vytvoření stejnou prioritu, operační systém neví který je který a naplánuje-li jako první dávkový příkaz, interaktivní proces bude do začátku jeho vykonávání čekat celé časové kvantum

praktické pravidlo:

zvolte trvání časového kvanta co nejdelší pří zachování dobrého času odpovědi systému

implementace plánování

technické vybavení má hodiny, časovač (*clock, timer*), který periodicky generuje přerušení, jejich frekvence je typicky 100Hz, tedy tikne každých 10ms

zpracování přerušení od časovače

- aktualizuje čas od začátku práce systému
- aktualizuje čas a datum
- aktualizuje statistiky o využívání prostředků
- určuje jak dlouho běžel okamžitý proces
- kontroluje jestli neuplynul čas sdružený s každým programovým časovačem a když ano vyvolá odpovídající funkci, využíváno jádrem i procesy

jádro například vypne mechaniku pružného disku, pokud se k němu jistou dobu nepřistupovalo

uživatel může voláním **`setitimer()`** a **`alarm()`** způsobit, aby procesu byly periodicky nebo jednotlivě zasílány signály po uplynutí stanoveného času

Tradiční plánování

vybere se proces ve stavu připraven s nejvyšší prioritou
má-li nejvyšší prioritu více procesů, vybere se ten, který je připraven nejdéle, cyklické plánování (*round robin*)

Main Entry: **round-rob·in**
Pronunciation: `'raund-"rä-b&n`
Function: *noun*
Etymology: from the name *Robin*
Date: 1730
1 a : a written petition, memorial, or protest to which the signatures are affixed in a circle so as not to indicate who signed first **b :** a statement signed by several persons **c :** something (as a letter) sent in turn to the members of a group each of whom signs and forwards it sometimes after adding comment

každý proces má ve svém **`proc`** záznamu položku s prioritou pro plánování, nastavenou při vytvoření procesu

větší číselná hodnota znamená nižší prioritu

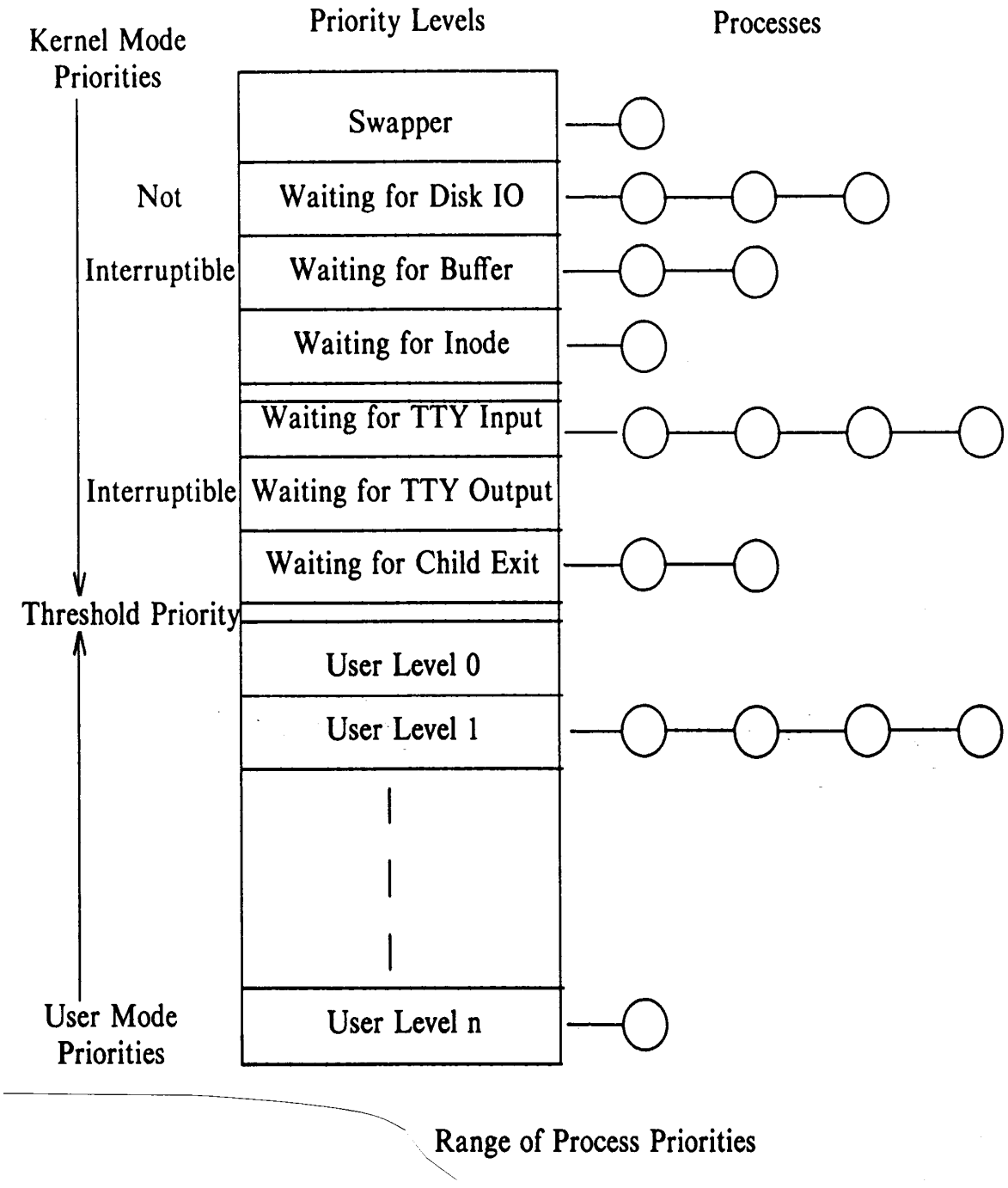
priority v módu jádro
- nepřerušitelné
- přerušitelné

priority v uživatelském módu

- procesům, které přecházejí do stavu spící, jádro přiradí pevné priority podle příčiny přechodu, priorita nezávisí na charakteru procesu

- při přechodu z módu jádra do módu uživatel, jádro upraví prioritu na uživatelskou

schéma rozsahu priorit
(nejvyšší priorita je 0)



Zdroj: Bach, M.: The Design of UNIX Operating System. Prentice-Hall 1986

- pro plánování procesů v uživatelském módu jádro pro běžící proces po každém tiknutí časovače zvyšuje hodnotu položky **`nedávné_použití_CPU`** záznamu **`proc`**

- jádro každou sekundu (SVR3, 4.3BSD) přepočítá priority všech procesů v uživatelském módu

sníží hodnotu **`nedávné_použití_CPU`** funkcí **`pokles`** (*decay*)

```
nedávné_použití_CPU = pokles(nedávné_
                                použití_CPU)
```

a prioritu vypočte ze vztahu

```
priorita = nedávné_použití_CPU/2 +
    základní_uživatelská_priorita
```

- proces, který nedávno čerpal hodně času procesoru, bude mít nízkou prioritu

- funkce **`pokles`** postupně zabezpečuje růst priority procesům, které čerpali čas procesoru dávno

Příklad

Mějme 3 procesy **`A, B, C`**, které byly vytvořeny současně se začáteční prioritou 60, nejvyšší uživatelská priorita je 60 a frekvence tiknutí je 60 s^{-1} a jiné procesy nejsou. Funkce **`pokles`** je

```
pokles(nedávné_použití_CPU) =
                    nedávné_použití_CPU / 2
```

Time	Proc A		Proc B		Proc C	
	Priority	Cpu Count	Priority	Cpu Count	Priority	Cpu Count
0	60	0 1 2 ⋮ 60	60	0	60	0
1	75	30	60	0 1 2 ⋮ 60	60	0
2	67	15	75	30	60	0 1 2 ⋮ 60
3	63	7 8 9 ⋮ 67	67	15	75	30
4	76	33	63	7 8 9 ⋮ 67	67	15
5	68	16	76	33	63	7

Zdroj: Bach, M.: The Design of UNIX Operating System. Prentice-Hall 1986

systémové volání **`nice()`**

- umožňuje změnit prioritu procesu o hodnotu argumentu volání **`nice()`**

```
priorita = nedávné_použití_CPU/2 +
    základní_uživatelská_priorita +
    hodnota_argumentu_nice
```

- jenom privilegovaný uživatel může zadat zápornou hodnotu a tím zvýšit prioritu procesu
- neprivilegovaný uživatel, může jenom snížit prioritu svých procesů, je tedy **nice** k ostatním

SVR3 nemá plánovací třídu pro procesy reálného času, jádro je nepreemtivní

SVR4

plánovací třídy:	priority:
reálný čas	100 – 159
systém	60 – 99
sdílení času	0 – 59

vysoká hodnota = vysoká priorita

plánovací třída pro sdílení času

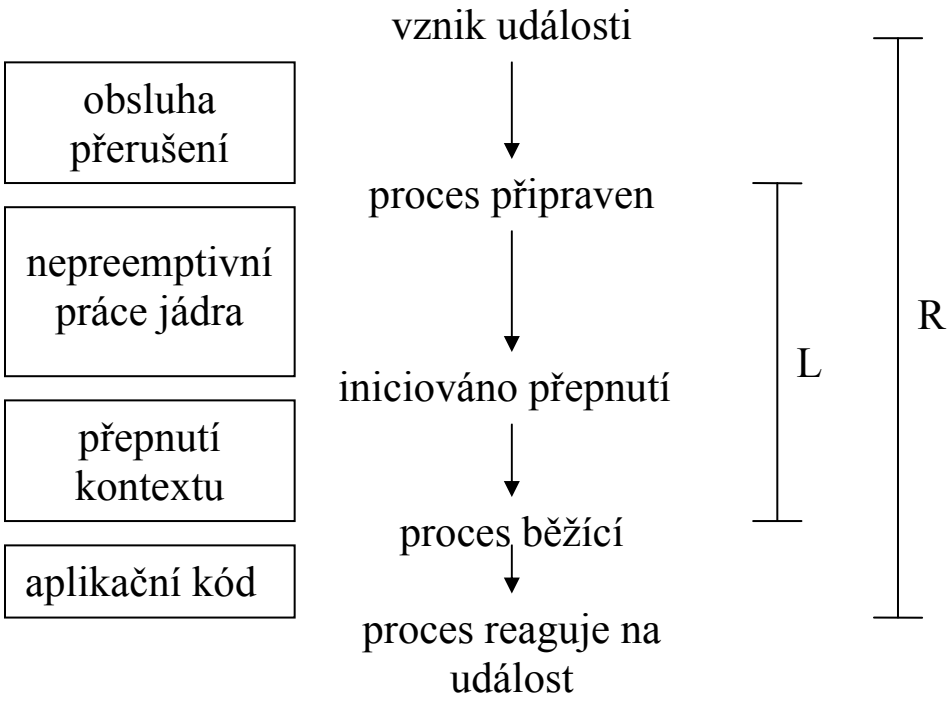
- dynamicky mění priority a pro procesy stejné priority používá cyklické plánování
- pro každou prioritu je definováno časové kvantum a další parametry
- pro nízké priority je časové kvantum delší
- změna priorit pro plánování řízena událostmi v systému, vyčerpání časového kvanta, obvykle přepočet priority pro jeden proces

plánovací třída pro reálný čas

- statické priority a pevné časové kvantum
- omezená
    - plánovací latence (*dispatch latency*)
    - doba odpovědi (*response time*)

při vzniku události, na kterou musí proces reálného času reagovat, je běžící proces přerušen a na konci obsluhy odpovídajícího přerušení je proces reálného času převeden do stavu připraven

pokud byl proces přerušen při vykonávání systémového volání, protože jádro je nepreemtivní může být naplánován při návratu do uživatelského módu



L - plánovací latence

R - doba odpovědi

pro snížení plánovací latence jádro SVR4 definuje body preempce, jsou to místa v kódu jádra, kdy jsou všechny údaje v konzistentním stavu a jádro se pouští do delšího výpočtu

po dosažení bodu preempce jádro zjišťuje, jestli není proces reálného času ve stavu **`připraven`** a je-li tomu tak vykoná preempci běžícího procesu

Linux 2.4

čas procesoru je rozdělen do epoch (*epoch*)

v jedné epoše, každý proces má specifikováno časové kvantum vypočteno na jejím začátku

v jedné epoše proces může využívat své časové kvantum po částech

epocha končí, když všechny běhu schopné procesy vyčerpali svá časová kvanta, kdy jsou přepočítána časová kvanta všech procesů (ne jenom běhu schopných) a začne nová epocha

každý proces má základní časové kvantum
potomek zdědí základní časové kvantum rodiče

uživatel může změnit základní časové kvantum voláním **`nice()`** a **`setpriority()`**(pro skupinu procesů)

proces **`0`** má nastavenou prioritu na **`DEF_PRIORITY`**

```
#define DEF_PRIORITY (20*HZ/100)
```

`HZ` je frekvence tiknutí, 100 Hz, tedy hodnota základního časového kvanta je 20 tiknutí

deskriptor procesu obsahuje pro plánování položky:

`policy`

plánovací třída, povolené hodnoty

SCHED_FIFO

procesy reálného času, proces s nejvyšší prioritou má přidělen procesor jak dlouho chce

SCHED_RR

cyklické plánovaní procesů reálného času

SCHED_OTHER

konvenční procesy se sdílením času

`rt_priority`

statická priorita procesů reálného času

`priority`

základní časové kvantum

`counter`

počet tiknutí, které procesu zůstávají do vypršení časového kvanta, na začátku je rovno trvání časového kvanta procesu, snižuje se po každém tiknutí v obsluze přerušení od časovače

když je proces vytvořen, hodnota **`counter`** se nastaví

```
rodič->counter >>= 1;
potomek->counter = rodič->counter;
```

počet tiknutí se rozdělí na polovinu

need_resched

příznak pro volání funkce **schedule()**

vhodnost procesu pro jeho naplánování jako dalšího běžícího počítá funkce **goodness()** s parametry **prev-**ukazatel na předcházející běžící proces a **p-**ukazatel na vyhodnocovaný proces

```
if (p = &init_task)
      return -1000;
if (p->policy != SCHED_OTHER)
      return 1000 + p->rt_priority;
if (p->counter == 0)
      return 0;
if (p->mm == prev->mm)
      return p->counter + p->priority + 1;
return p->counter + p->priority;
```

p získá malý bonus, sdílí-li adresový prostor s předcházejícím běžícím procesem

plánovač realizuje funkce **schedule()**

její volání funkcemi v jádře může být přímé (*direct*) nebo odložené (*lazy*)

přímé volání nastane, když běžící proces se musí stát spícím

odložené volání nastane, když běžící proces při návratu do uživatelského módu zjistí, že má příznak **need_resched** nastaven

příznak **need_resched** je nastaven:

- když proces vyčerpal své časové kvantum

```
if (běžící->pid) {
    běžící->counter -= tiknutí;
    if (běžící->counter < 0) {
        běžící->counter = 0;
        běžící->need_resched = 1;
    }
}
```

proces **pid = 0** nesmí být plánován pro sdílení času

táto část obsluhy přerušení od hodin je odložená (*bottom half*), proto při snižování hodnoty **counter** se použije lokální proměnná **tiknutí**

- když vhodnost vzbuzeného procesu je vyšší než běžícího

```
if (goodness(běžící, p) >
            goodness(běžící, běžící))
    běžící->need_resched = 1;
```

- když je zavoláno systémové volání **sched_setscheduler()** nebo **sched_yield()**, které umožňují nastavit strategii a prioritu nebo vzdát se procesoru procesům reálného času

`schedule()` vybere nejvhodnější proces, kterému bude přidělen procesor

proměnná **`next`** bude obsahovat ukazatel na deskriptor nejvhodnějšího procesu (s nejvyšší prioritou)

napřed inicializujeme **`next`** na proces, ke kterému začneme vyhodnocování a **`c`** na jeho vhodnost

```
if (prev->stav == BĚŽÍCÍ) {
    next = prev;
    if (prev->policy & SCHED_YIELD) {
        prev->policy &= ~SCHED_YIELD;
        c = 0;
    } else
        c = goodness(prev, prev);
} else {
    c = -1000;
    next = &init_task;
}
```

jestli se běžící vzdal procesoru byl jeho příznak **`SCHED_YIELD`** nastaven a jeho vhodnost je 0

začneme procesem, který byl dosud běžící nebo procesem 0 (**`init_task`**)

projdeme kruhový seznam běhu schopných procesů a najdeme nejvhodnější proces

```
p = init_task.next_run;
while (p != &init_task) {
    váha = goodness(běžící, p);
    if (váha > c) {
        c = váha;
        next = p;
    }
    p = p->next_run
}
```

je vybrán první proces s maximální váhou, předcházející běžící je upřednostněn před jinými běhu schopnými procesy se stejnou váhou

je-li **`c = 0`** všechny běhu schopné procesy vyčerpali svoje časové kvantum a začíná nová epocha a **`schedule()`** přidělí všem procesům nové časové kvantum

```
if (!c) {
    pro_každý_proces(p)
       p->counter = p->priority +
                          (p->counter >> 1);
```

spícím a zastaveným procesům se tedy zvyšuje dynamická priorita

SMP a Linux 2.4 plánovač

- jedna globální fronta připravených procesů

    plánujeme jenom pro jeden procesor

    nová časová kvanta se počítají na konci epochy
    pokud zůstal jeden proces s nevyčerpaným kvantem,
    ostatní procesory stojí

    proces, který na procesoru vyčerpal své kvantum
    nepokračuje, a procesor bude přidělen jinému procesu,
    který mohl běžet na jiném procesoru a nutno zabezpečit
    konzistenci cache paměti

- složitost

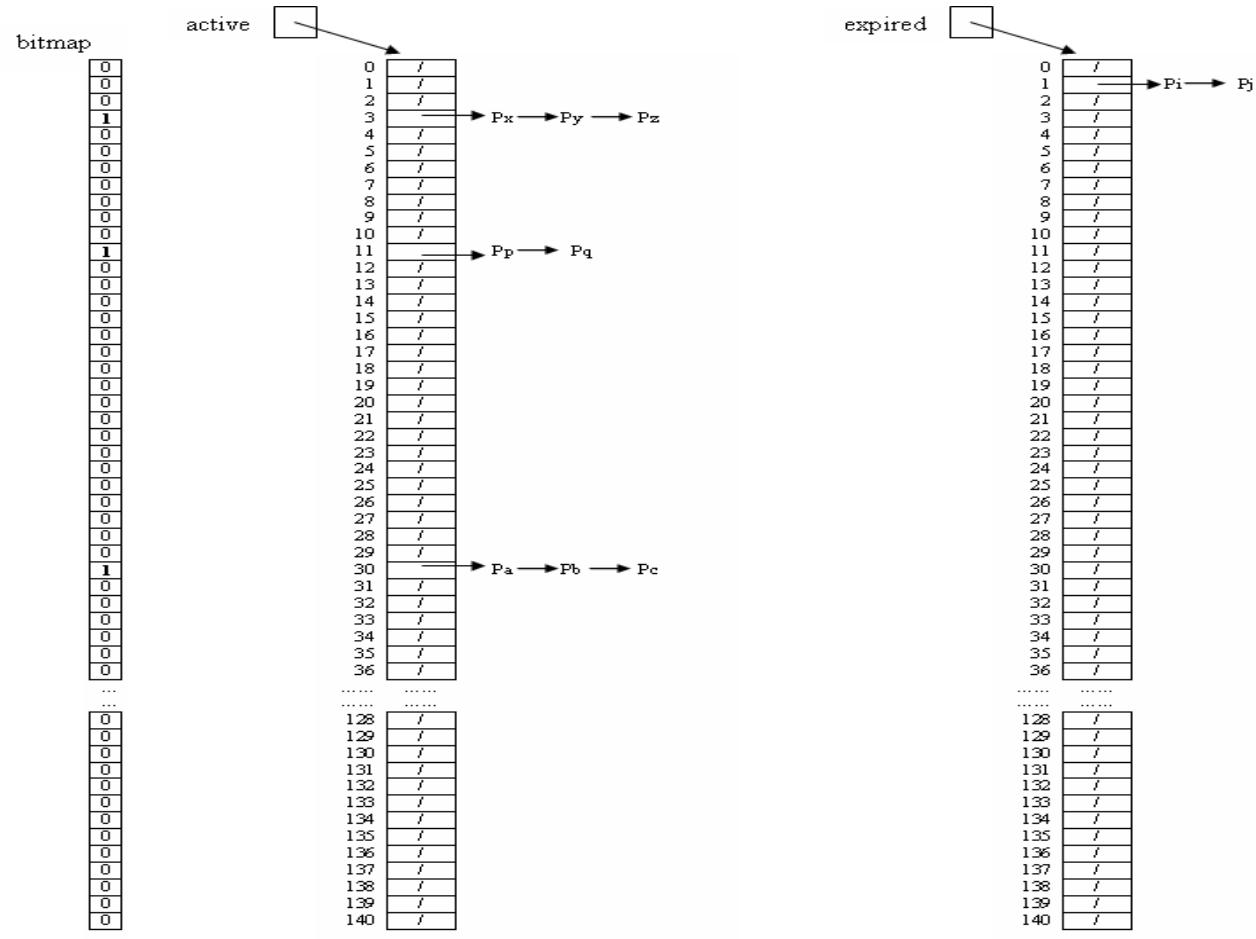
    při plánování procesů se prochází celá fronta – *O(n)*
    na konci epochy se přepočítavají časová kvanta – *O(n)*

Linux 2.6 – plánovač *O(1)* / Ingo Molnar

- fronta připravených procesů pro každý procesor
- fronta je složena ze dvo polí o velikosti počtu priorit

`active` – procesy s ještě nevyčerpaným časovým kvantem
prioritní plánování
RR v rámci priority

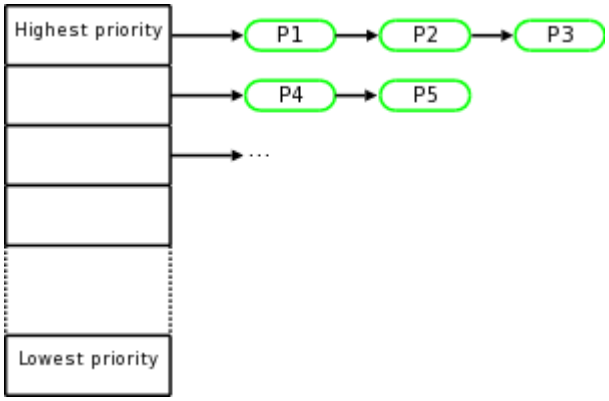
`expired`- procesy s vyčerpaným časovým kvantem



nové časové kvantum a priorita se vypočte ihned po vyčerpání předcházejícího kvanta *O(1)* a proces se zařadí do fronty procesů s novou prioritou do pole **`expired`** na index odpovídající prioritě

je-li pole **`active`** prázdné „konec epochy“ pole se přepnou

RSDL (Rotating Staircase Deadline Scheduler) plánovač / Con Kolivas



- procesy nejvyšší priority jsou plánovány RR
- vyčerpá-li proces své kvantum je přeřazen do nižší priority
- každá priorita má vlastní kvótu a je-li vyčerpána, všechny procesy jsou přeřazeny do nižší priority
- z nejnižší priority jsou přeřazeny do **`expired`**
- nejsou-li procesy v poli **`active`**, pole se přepnou

CFS (Complemetly Fair Scheduler) / Ingo Molnar

Linux 2.6.23

- ne pole, ale časově uspořádaný RB strom, operace jsou O(log n)

- modeluje rovnoměrné rozdělení času procesoru, n procesům, každý 1/n času

- skupinové plánování

Reálný čas

efektivní plánovač pro reálny čas musí kromě plánovací latence řešit i další problémy

nereemtivní jádro Linuxu neposkytuje krátkou plánovací latenci a jeho využití pro reálny čas je omezené

možná řešení:

zavedení bodů preempce (SVR4)
vytvoření preemtivního jádra (Solaris, Linux 2.6)

Inverze priority (*priority inversion*)

proces vysoké priority musí čekat na uvolnění prostředku, který vlastní proces nízké priority

mohou být plánovány procesy „středních“ priorit !

Mars Pathfinder, 1997

vesmírnou sondu řídil jeden procesor připojený na sběrnici, na které byly další přístroje - rádio, kamera, a rozhraní k sběrnici 1553

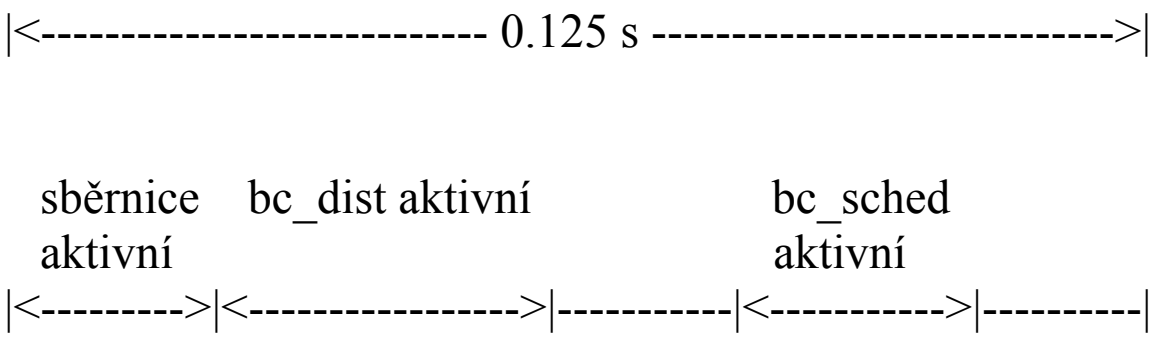
sběrnice 1553 byla připojena k části "letové" a "přistávací"

v přistávací části byl připojen meteorologický přístroj ASI/MET

na sběrnici 1553 byly aktivity plánovány s frekvencí 8 Hz

sběrnice a přístroje byly obsluhovány dvěma procesy

bc_sched řídil nastavení sběrnice
bc_dist distribuoval data



kontrolovalo se, jestli proces skončil než následující začal, když ne chyba, která vyvolala "reset"

bc_sched měl nejvyšší prioritu
bc_dist třetí nevyšší
následovaly další procesy
ASI/MET proces měl velmi nízkou prioritu

bc_dist i ASI/MET proces používali pro přenos dat IPC (*inter process communication*) mechanizmus založený na rourách

přístup k výběru roury byl řízen mutex semaforem

1. ASI/MET proces získal mutex
2. vznikla preempce a běželo několik procesů střední priority
3. bc_dist byl aktivován, požádal o mutex a stal se spícím
4. další procesy střední priority běžely
5. byl aktivován bc_sched a zjistil, že bc_dist neskončil, indikoval chybu a inicializoval se reset

řešení:

3. ... ASI/MET proces zdědí vysokou prioritu (*priority inheritance*) bc_dist procesu požadujícího prostředek
4. pokračuje ASI/MET proces a uvolní semafor, kdy je mu snížena priorita, a vykoná se preempce
5. pokračuje bc_dist a po něm bc_sched