Sistemas Operativos
Cap. V
Escalonamento de Threads

Prof. José Rogado

jose.rogado@ulusofona.pt

Universidade Lusófona

Escalonamento do CPU

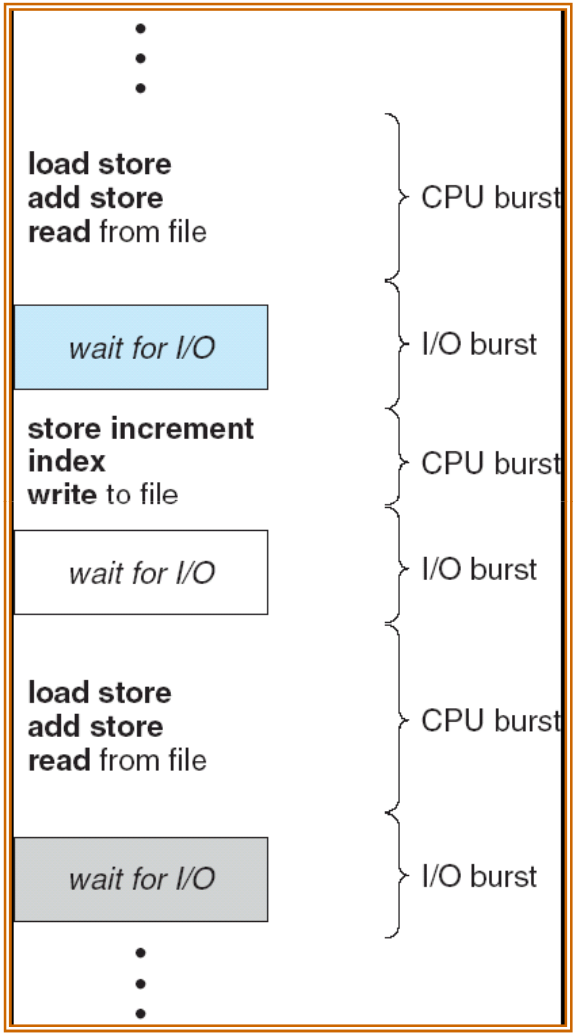
- Conceitos Básicos
- Critérios de Escalonamento
- Algoritmos de Escalonamento
- Escalonamento Multi-processador
- Escalonamento em Tempo Real
- Escalonamento de Threads
- Exemplos de escalonamento em Sistemas Operativos
- Escalonamento de Threads
- Avaliação de Algoritmos

Conceitos Básicos

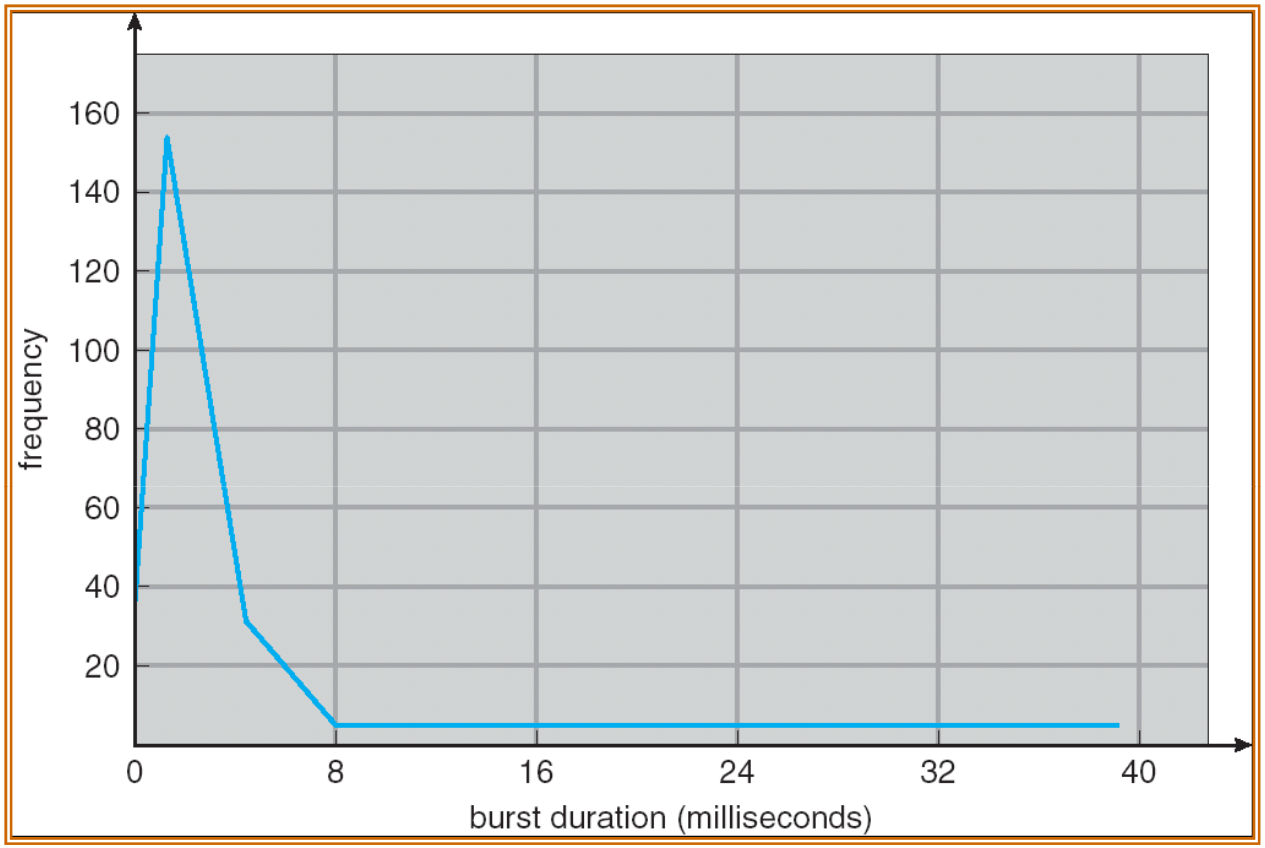
- Objectivos do Scheduler
  - Obter a melhor utilização possível do CPU em multiprogramação

Baseia-se no seguintes factos:

- Padrão de Execução de Processos
  - Alternância de ciclos de picos de utilização de CPU e I/O
- Distribuição dos picos de utilização de CPU
  - Segue um padrão idêntico na maioria dos sistemas



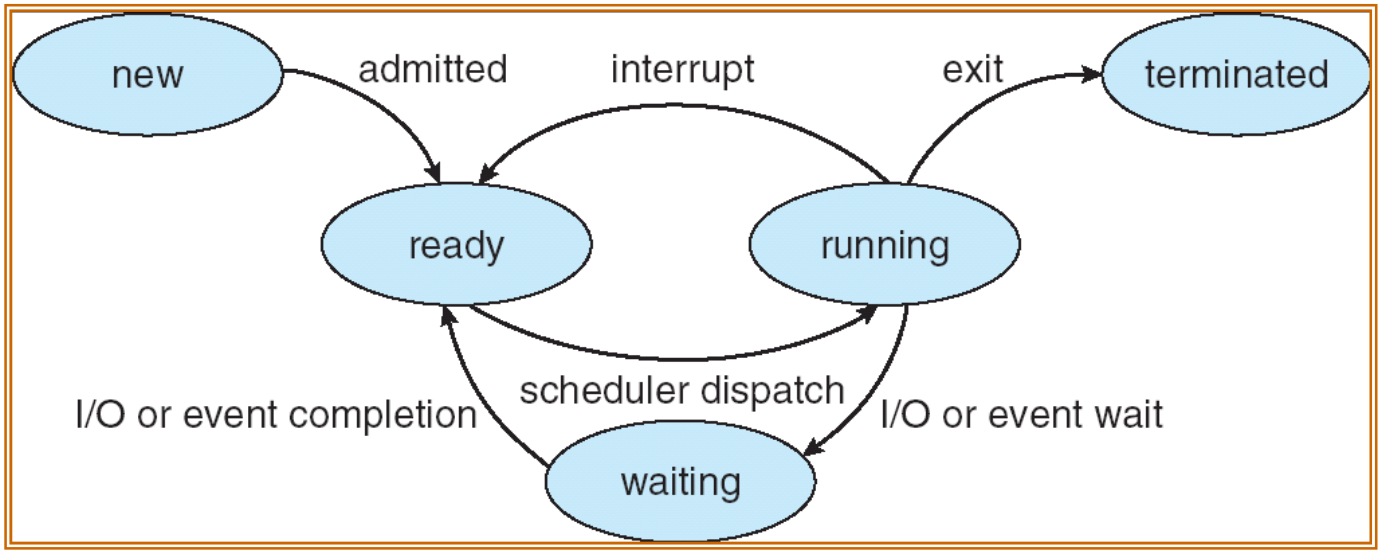
Histograma da utilização do CPU



- As utilizações curtas do CPU tendem a ser mais frequentes
  - Muitas utilizações curtas
  - Poucas utilizações longas

Funcionalidades do Scheduler

- Selecciona um dos processos em memória que esteja pronto para execução (*Ready Queue*) e atribui-lhe o CPU
- As decisões do scheduler podem ter lugar quando um processo
  1. Passa do estado running para waiting
  2. Passa do estado running para ready
  3. Passa do estado ready para running
  4. Termina
- O scheduling no caso 2 é *preemptivo*
- Nos outros pontos é *não-preemptivo*



Dispatcher

- O Dispatcher é o módulo que se encarrega de entregar o controlo do CPU ao processo escolhido pelo scheduler, o que envolve:
  - Mudança de contexto
  - Mudança para modo utilizador
  - Executar o processo a partir do endereço apropriado
- *Dispatch latency* – o tempo que o dispatcher leva a parar um processo e recomeçar um outro
  - Tempo que demora a mudança de contexto
  - Envolve inúmeras operações de manipulação de registos CPU, unidade de gestão de memória (MMU), pilhas, etc…

Critérios de Scheduling

- Utilização do CPU - manter o CPU o mais ocupado possível
- Débito de processamento - nº de processos executados por unidade de tempo
- Tempo de execução - tempo total que um dado processo leva a ser executado (***Turnaround Time***)
- Tempo de espera - tempo que um processo espera na *ready queue* antes de ser activado
- Interactividade - tempo que leva um determinado pedido a ser tomado em conta (tempo de resposta)
- Optimizações possíveis:
  - Máxima Utilização CPU
  - Máximo Débito de Processamento
  - Mínimo tempo de execução
  - Mínimo tempo de espera
  - Máxima interactividade

Algoritmos de Scheduling

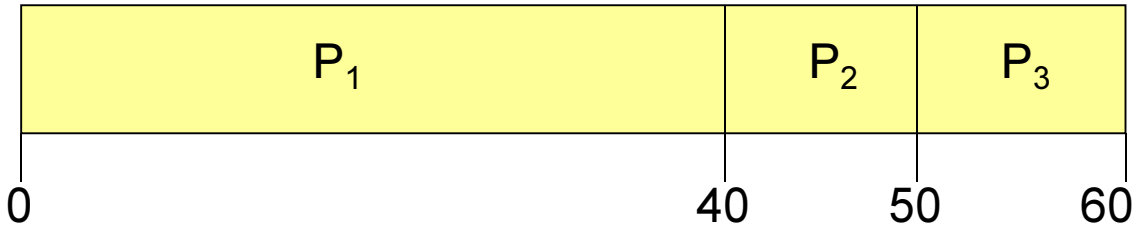
- Existem numerosos algoritmos de escalonamento que correspondem a vários objectivos da gestão de processos num determinado sistema operativo
  - Um dado sistema pode utilizar um ou múltiplos algoritmos, de forma a melhor se adaptar às circunstâncias de funcionamento
- O algoritmo de scheduling determina qual o processo que irá ser executado num dado instante, em função de vários parâmetros
  - Tempo de utilização do CPU, prioridade, etc..
- Os algoritmos utilizados mais frequentemente são os seguintes:
  - First-Come First Served (FCFS) ou FIFO
  - Shortest-Job-First (SJF)
  - Algoritmos preemptivos e não preemptivos
  - Round-Robin com ou sem prioridade
  - Multilevel
  - Etc.

Algoritmo First-Come, First-Served (FCFS)

Exemplo:

Processo	Tempo de CPU
P_1	40
P_2	10
P_3	10

- Supondo que os processos são criados pela ordem: P_1 , P_2 , P_3
  O mapa de Gantt para o *scheduling* é:



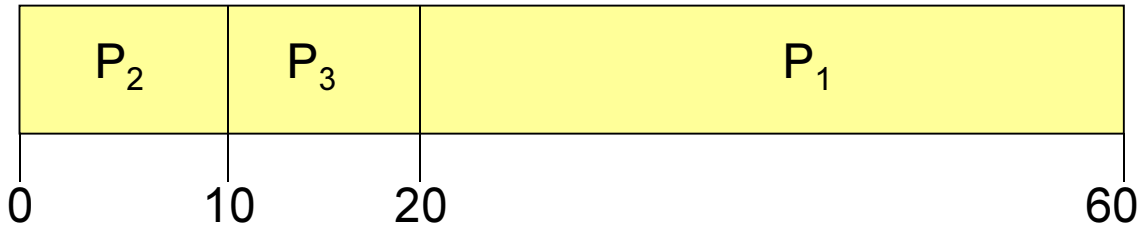
- Tempos de espera: P_1 = 0; P_2 = 40; P_3 = 50
- Tempo de espera médio: (0 + 40 + 50)/3 = 30
- Implementação simples: lista FIFO

FCFS Scheduling (Cont.)

Supondo agora que os mesmos processos são criados pela ordem

$$P_2 , P_3 , P_1$$

- O mapa de Gantt para o *scheduling* é:



- Tempos de espera: $P_1 = 20$; $P_2 = 0$; $P_3 = 10$
- Tempo de espera médio: (20 + 0 + 10)/3 = 10
- Muito melhor do que no caso anterior !
- Conclusão: para optimizar o tempo de espera convém executar os processos de duração curta primeiro
  - Evita o efeito do comboio

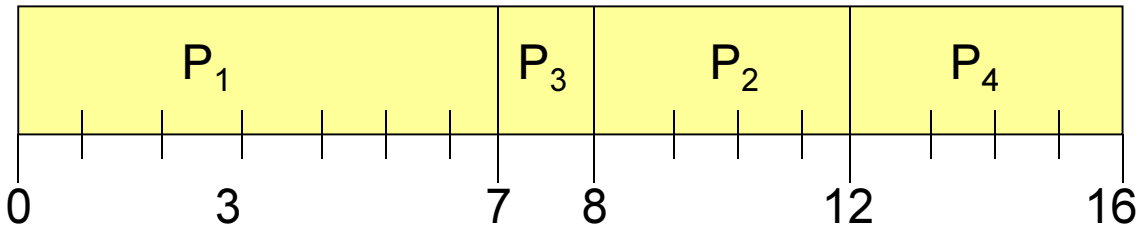
Scheduling Shortest-Job-First (SJF)

- Associa-se a cada processo a duração expectável do seu próximo ciclo de utilização do CPU (***service time***).
  - O processo com o valor mais baixo é activado primeiro
- Duas possibilidade:
  - Não-preemptivo: uma vez o CPU é atribuído a um processo, este não é interrompido até ao fim do seu ciclo de CPU
  - Preemptivo: se chega um novo processo com um ciclo de CPU mais curto do que o tempo que falta ao processo corrente, este é interrompido.
    - É conhecido por Shortest-Remaining-Time-First (SRTF)
- O escalonamento SJF é o óptimo do ponto de vista do tempo de espera médio
  - Minimiza o tempo de espera média para um dado conjunto de processos

Exemplo de SJF não preemptivo

Processo	Criação	Service Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (não preemptivo)

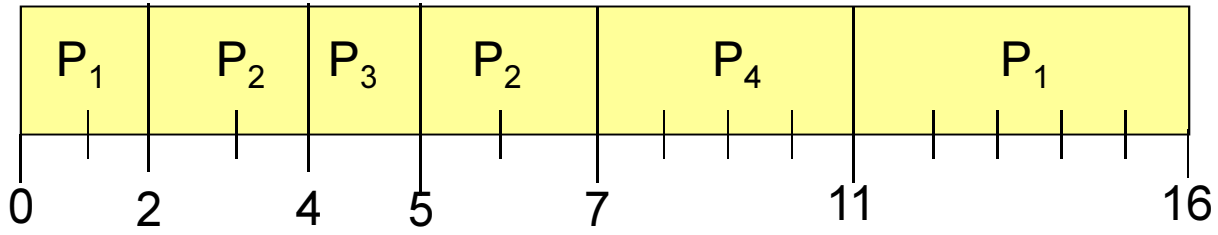


- Tempo de espera médio = (0 + 6 + 3 + 7) / 4 = 4

Exemplo de SJF preemptivo

Processo	Criação	Service Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptivo)



- Tempo de espera médio = (9 + 1 + 0 +2) / 4 = 3

Determinação da Duração do Ciclo de CPU

- A determinação da duração do ciclo de utilização de CPU só pode ser feita por estimativa
  - Um processo utiliza o CPU em vários ciclos sucessivos que constituem o historial do processo
- Assim, a estimativa utiliza a duração dos ciclos anteriores, calculando a média exponencial à medida da execução do processo
- Para isso define-se:
  1. t_n = duração do ciclo de CPU *n*
  2. τ_{n+1} = valor estimado para o próximo ciclo
- O valor estimado é:
  1. $\tau_{n+1} = \alpha\, t_n + (1- \alpha)\, \tau_n$
  2. Sendo $0 <= \alpha <= 1$
- Valores limites
  - Se $\alpha = 0$ então $\tau_{n+1} = \tau_n$ (valor previsto para o ciclo anterior)
  - Se $\alpha = 1$ então $\tau_{n+1} = t_n$ (valor real efectivo do ciclo anterior)

Propriedades da Média Exponencial

Interpretação dos valores limites:

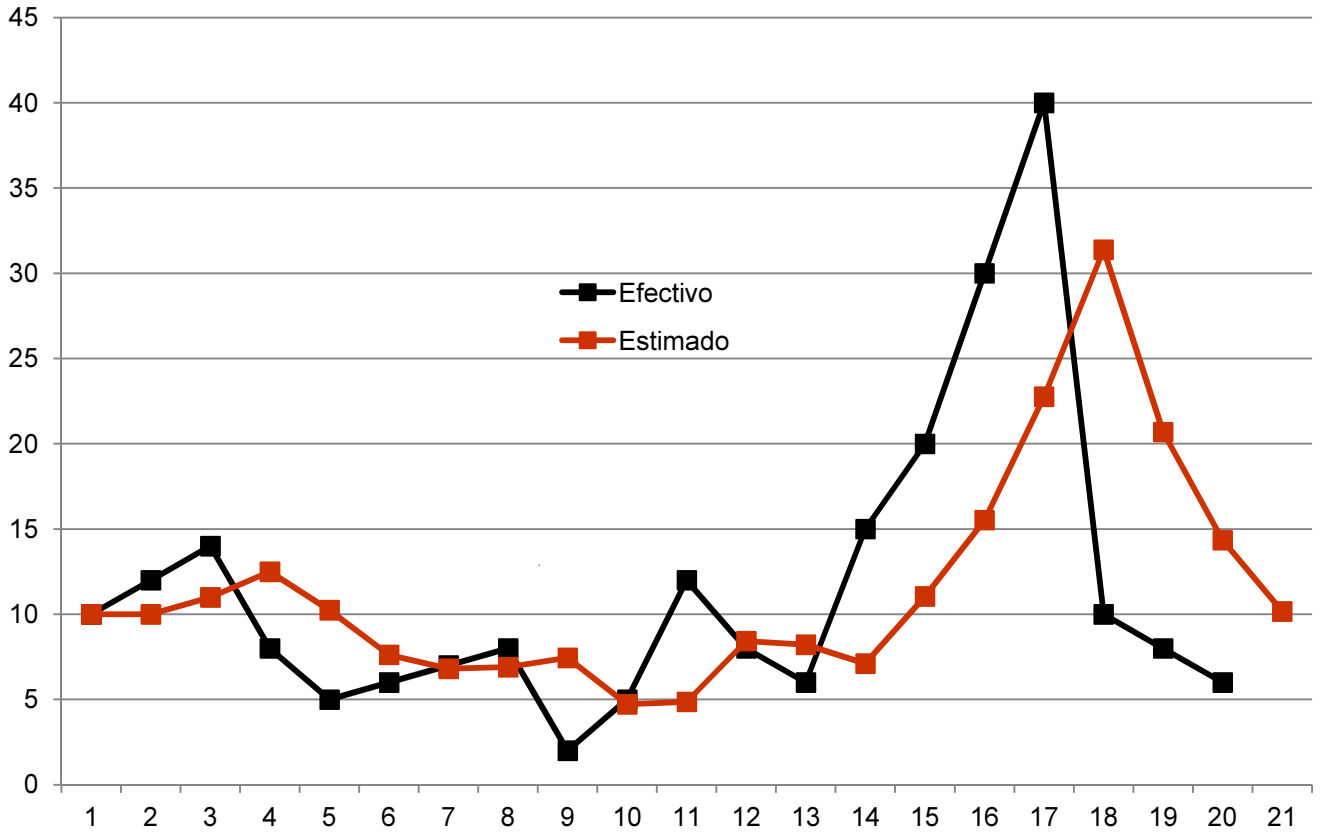
- Se $\alpha = 0$ então $\tau_{n+1} = \tau_n$
  - Valor previsto para o ciclo anterior: o valor efectivo anterior não entra em linha de conta, só é considerado o valor estimado
- Se $\alpha = 1$ então $\tau_{n+1} = t_n$
  - Valor real efectivo do ciclo anterior: só é considerada a duração do ciclo anterior
- Expandindo a fórmula, obtém-se:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$$

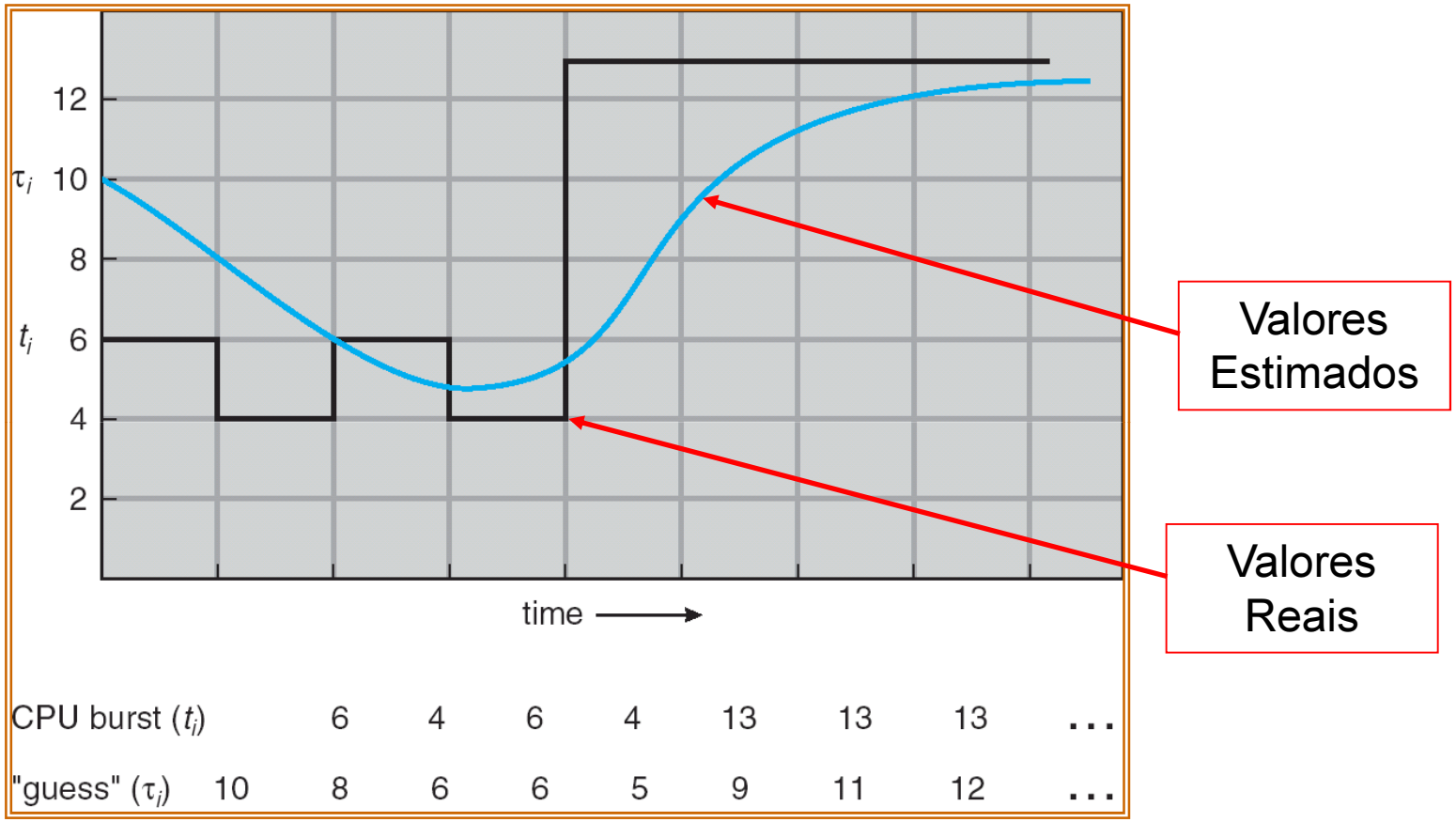
- Como α e $1 - \alpha$ são inferiores ou iguais a 1, a contribuição dos termos passados vai diminuindo com o tempo
- Na prática utiliza-se $\alpha = 0.5$ dando igual peso à história anterior e à imediata

Exemplo Numérico

Efectivo		Estimado	
T0	10	E0	10,0
T1	12	E1	10,0
T2	14	E2	11,0
T3	8	E3	12,5
T4	5	E4	10,3
T5	6	E5	7,6
T6	7	E6	6,8
T7	8	E7	6,9
T8	2	E8	7,5
T9	5	E9	4,7
T10	12	E10	4,9
T11	8	E11	8,4
T12	6	E12	8,2
T13	15	E13	7,1
T14	20	E14	11,1
T15	30	E15	15,5
T16	40	E16	22,8
T17	10	E17	31,4
T18	8	E18	20,7
T19	6	E19	14,3
		E20	10,2



Exemplo de Estimativas



Valores obtidos com $\alpha = 0.5$ e $\tau_0 = 10$

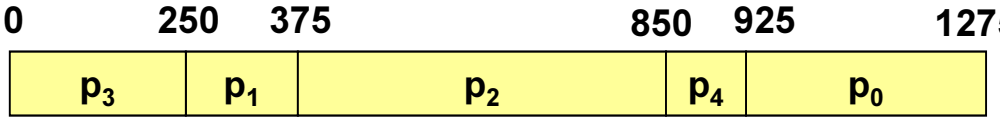
Scheduling Baseado na Prioridade

- A prioridade é representada por um valor inteiro associado ao processo (geralmente valor menor ≡ prioridade maior)
- O CPU é alocado ao processo da *ready queue* com maior prioridade
- O scheduling por prioridades pode ser ou não preemptivo
  - Preemptivo: um processo com maior prioridade toma o lugar de um com menor prioridade
  - Não Preemptivo : o processo em curso é executado até finalizar o seu ciclo CPU
- O scheduling SJF pode ser considerado um scheduling de prioridade em que a medida da prioridade é o valor previsto do próximo ciclo CPU
- O scheduling por prioridade pode trazer o problema de que certos processos com baixa prioridade nunca sejam executados (***Starvation***)
- A solução é utilizar a técnica de ***Aging*** que aumenta a prioridade dos processos em espera à medida que o tempo aumenta.

Priority Scheduling

i	$t(p_i)$	Pri
0	350	5
1	125	2
2	475	3
3	250	1
4	75	4

- *Reflecte a utilização do sistema*
- Pode causar *starvation*
- A *starvation* resolve-se com *aging*



$T_{TRnd}(p_0) = t(p_0)+t(p_4)+t(p_2)+t(p_1))+t(p_3) = 350+75+475+125+250 = 1275$

$T_{TRnd}(p_1) = t(p_1)+t(p_3) = 125+250 = 375$

$T_{TRnd}(p_2) = t(p_2)+t(p_1)+t(p_3) = 475+125+250 = 850$

$T_{TRnd}(p_3) = t(p_3) = 250$

$T_{TRnd}(p_4) = t(p_4)+ t(p_2)+ t(p_1)+t(p_3) = 75+475+125+250 = 925$

$W(p_0) = 925$
$W(p_1) = 250$
$W(p_2) = 375$
$W(p_3) = 0$
$W(p_4) = 850$

$Wavg = (925+250+375+0+850)/5 = 2400/5 = 480$

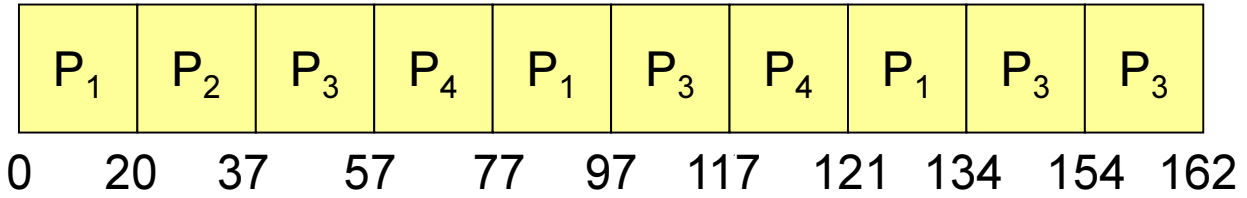
Scheduling em Round Robin (RR)

- Cada processo recebe uma pequena fracção do tempo de CPU (*time quantum*), da ordem dos 10-100 milissegundos
- Depois de esgotado esse tempo, o processo é interrompido e passado para o fim da fila de espera (*Ready Queue*)
- Se houver ***n*** processos na *Ready Queue* e o quantum de tempo for ***q***, então cada processo recebe **1/*n*** do tempo de CPU, em fracções de ***q*** unidades de tempo.
  - Nenhum processo espera mais de ***(n-1) x q*** unidades de tempo
- Performance: a relação entre q e o tempo de duração média τ do ciclo de CPU dos processos é determinante:
  - Se $q > \tau \Rightarrow$ degenera em FIFO
  - Se $q < \tau \Rightarrow q$ deve ser suficientemente grande em relação ao tempo médio da mudança de contexto, ou então o *overhead* torna-se demasiado elevado.

Exemplo de RR com Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- O mapa de Gantt é:

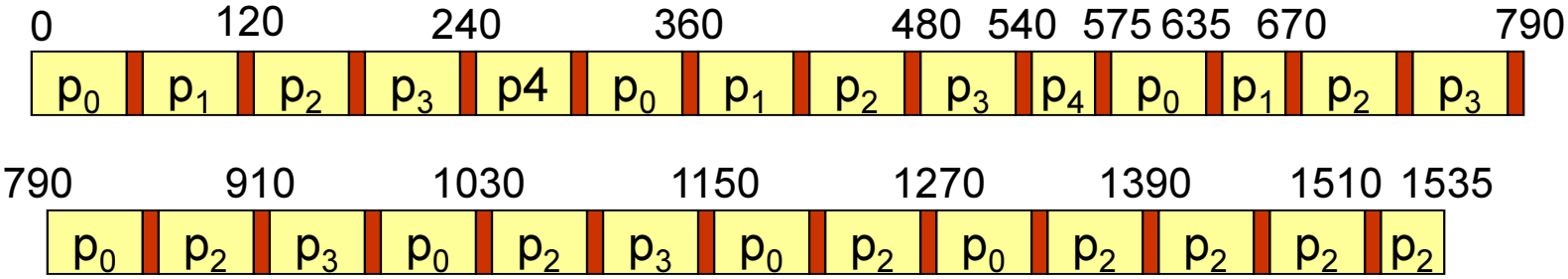


- Tipicamente o tempo de execução médio é maior que em SJF mas obtém-se maior interactividade (menor tempo de resposta)

RR com Overhead=5 (TQ=50)

i	$t(p_i)$
0	350
1	125
2	475
3	250
4	75

- O Overhead da comutação de processos deve ser tomado em conta



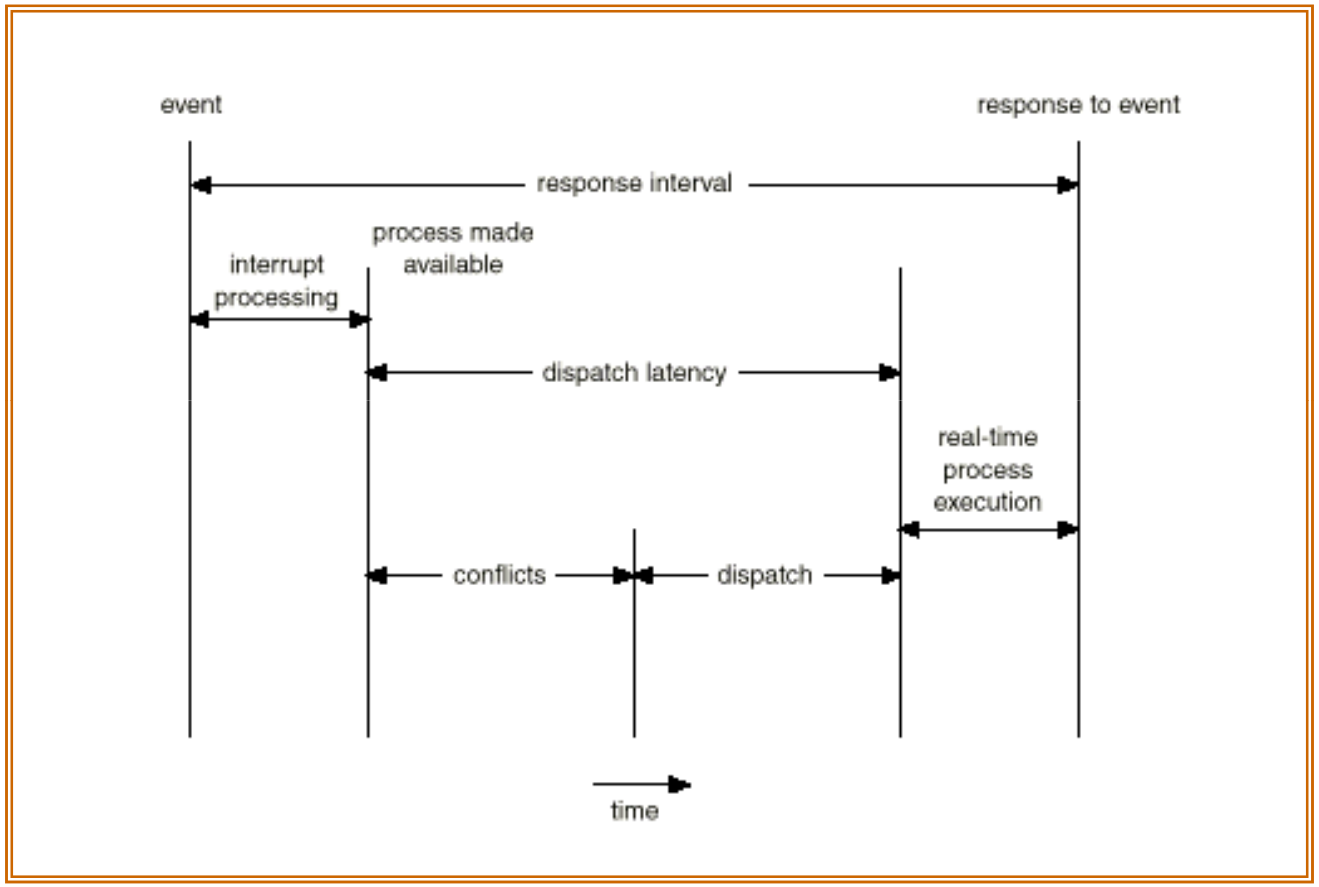
$T_{TRnd}(p_0) = 1320$
$T_{TRnd}(p_1) = 660$
$T_{TRnd}(p_2) = 1535$
$T_{TRnd}(p_3) = 1140$
$T_{TRnd}(p_4) = 565$

$W(p_0) = 0$
$W(p_1) = 60$
$W(p_2) = 120$
$W(p_3) = 180$
$W(p_4) = 240$

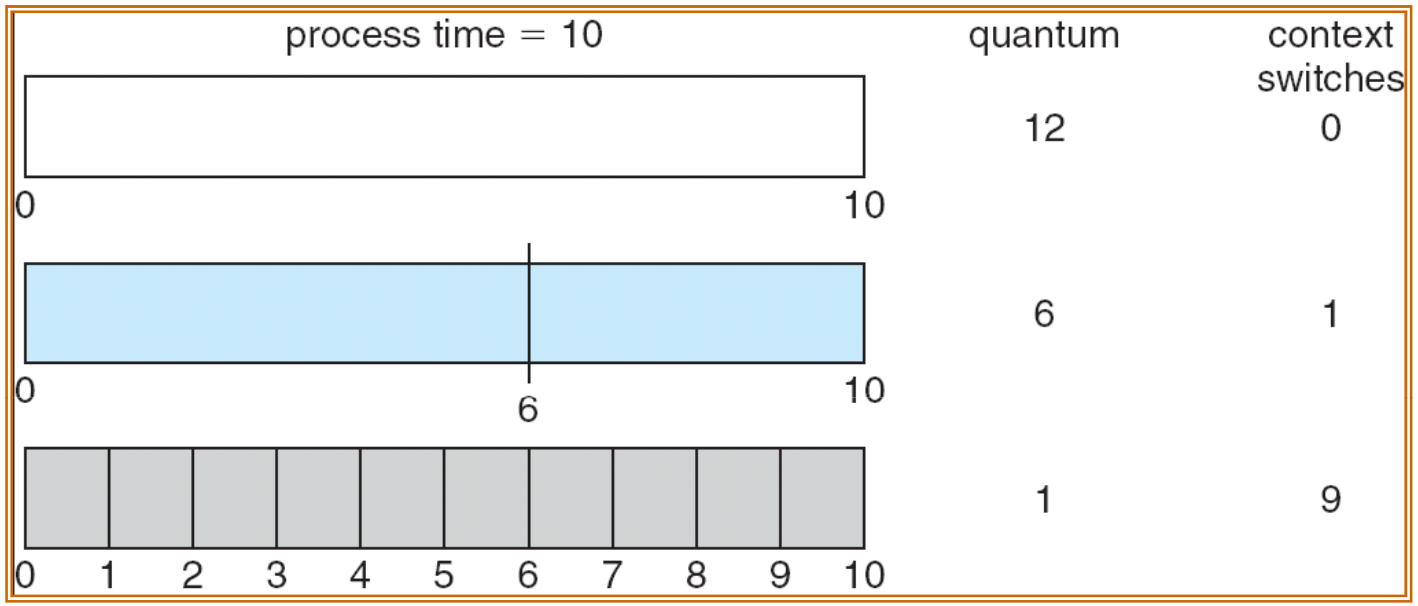
$$T_{TRnd\_avg} = (1320+660+1535+1140+565)/5 = 5220/5 = 1044$$

$$W_{avg} = (0+60+120+180+240)/5 = 600/5 = 120$$

Dispatch Latency

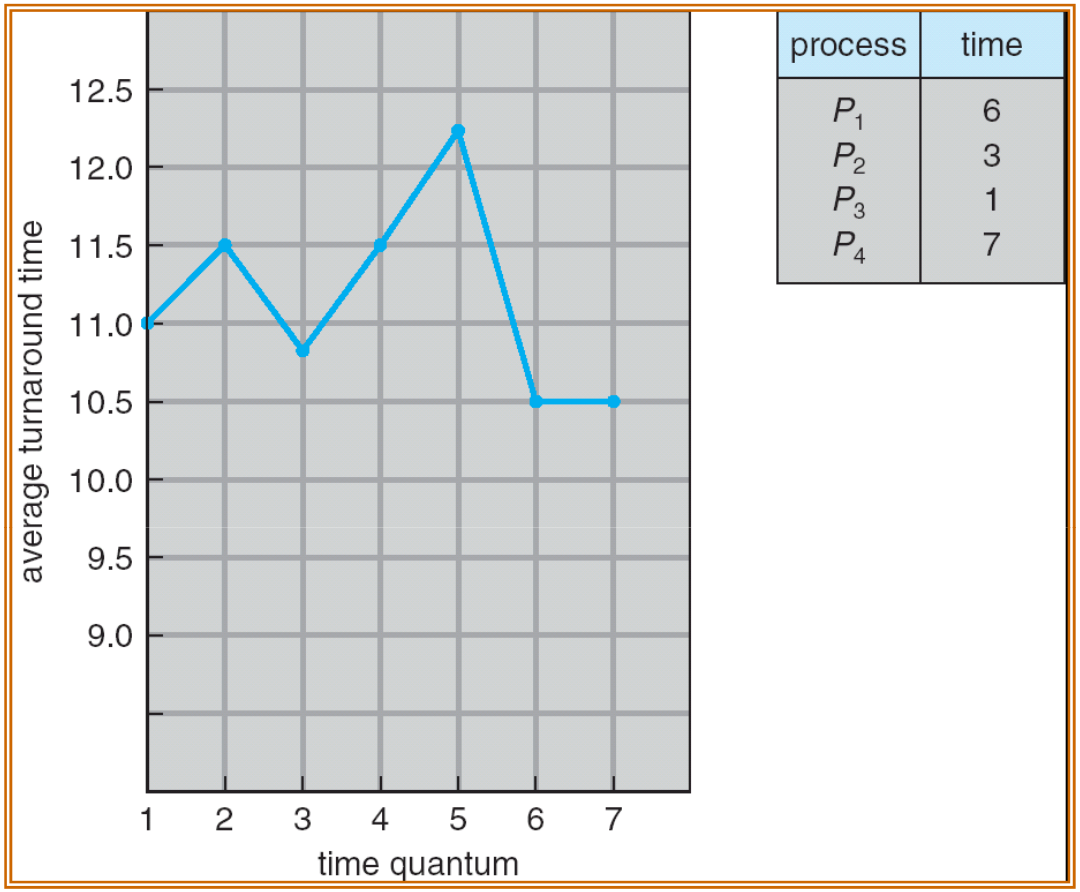


Quantum e Mudança de Contexto



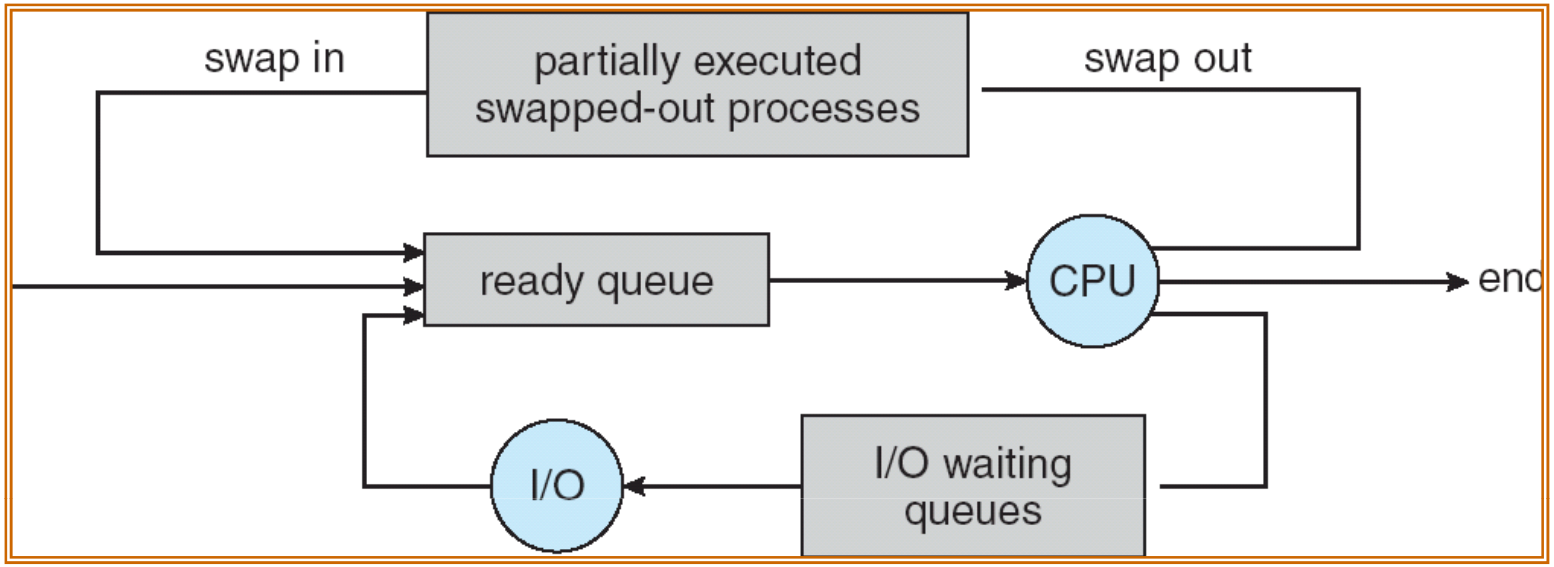
- À medida que o time quantum diminui, aumenta o número de mudanças de contexto durante a execução do processo
- Típicamente, o tempo de comutação de contexto é da ordem dos 10 µs, ou seja um milésimo do valor do quantum habitual (10-100 ms)

Tempo de Resposta vs. Time Quantum



- O tempo de resposta médio não melhora necessariamente com o aumento do Time Quantum
- Em geral, deve-se ter 80 % dos ciclos de CPU inferiores ao Time Quantum

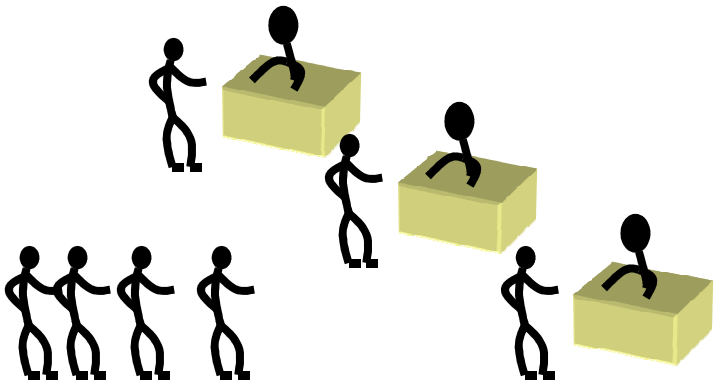
Noção de Swapping



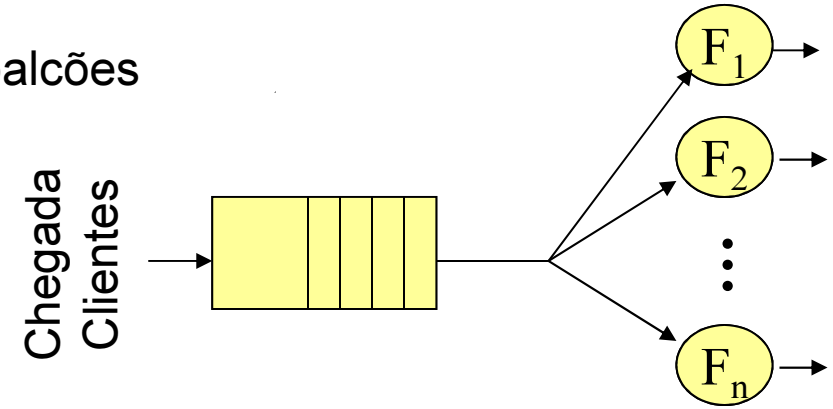
- Quando demasiados processos estão a competir pela utilização do CPU, certos sistemas realizam ***swapping***
  - São tirados processos da wait queue (que não estão a ser executados) e enviados para uma zona em disco (ex: *swap partition*)
  - Mais tarde podem voltar para a ready queue quando o número de processos em memória diminui

Scheduling Multilevel

Analogia com o processo de atendimento com vários balcões

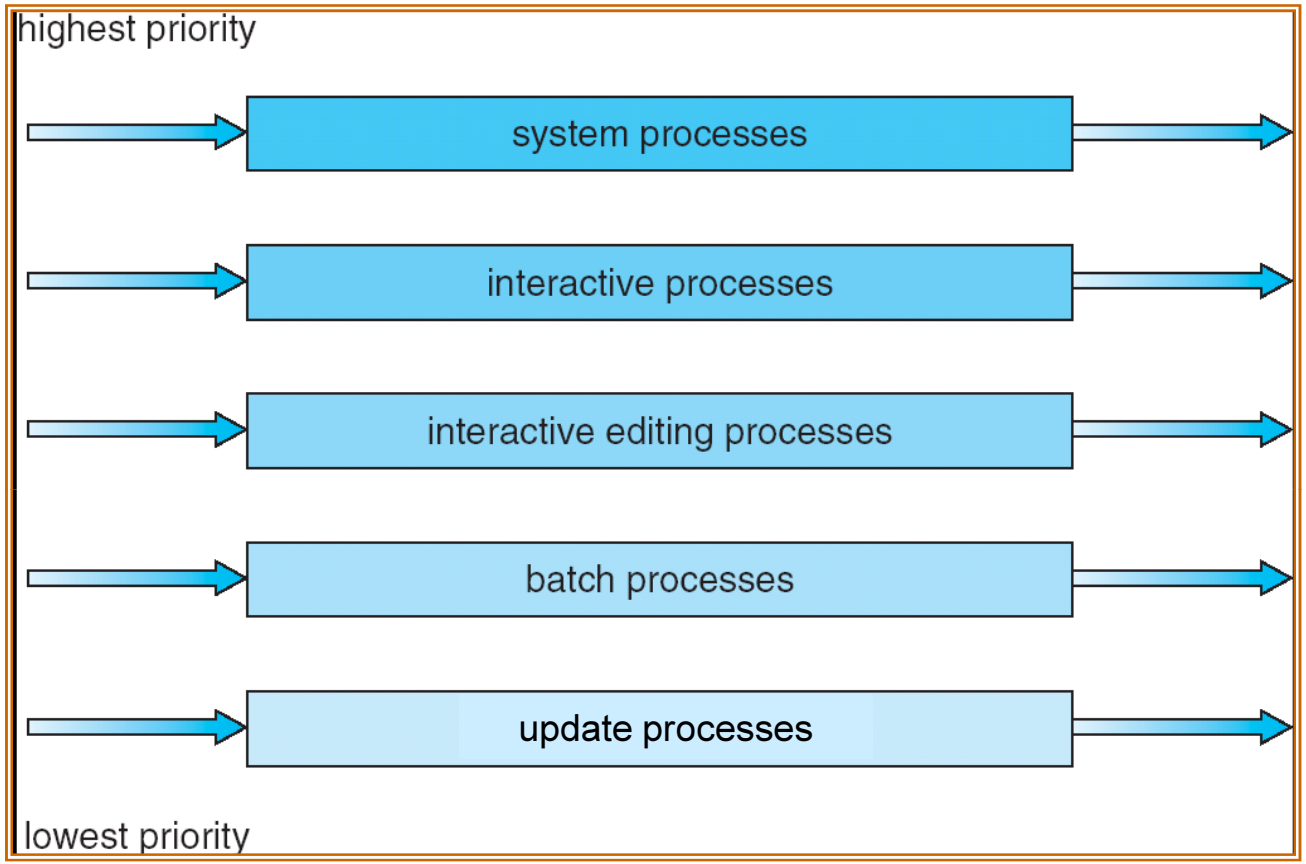


Fila de espera para múltiplos balcões



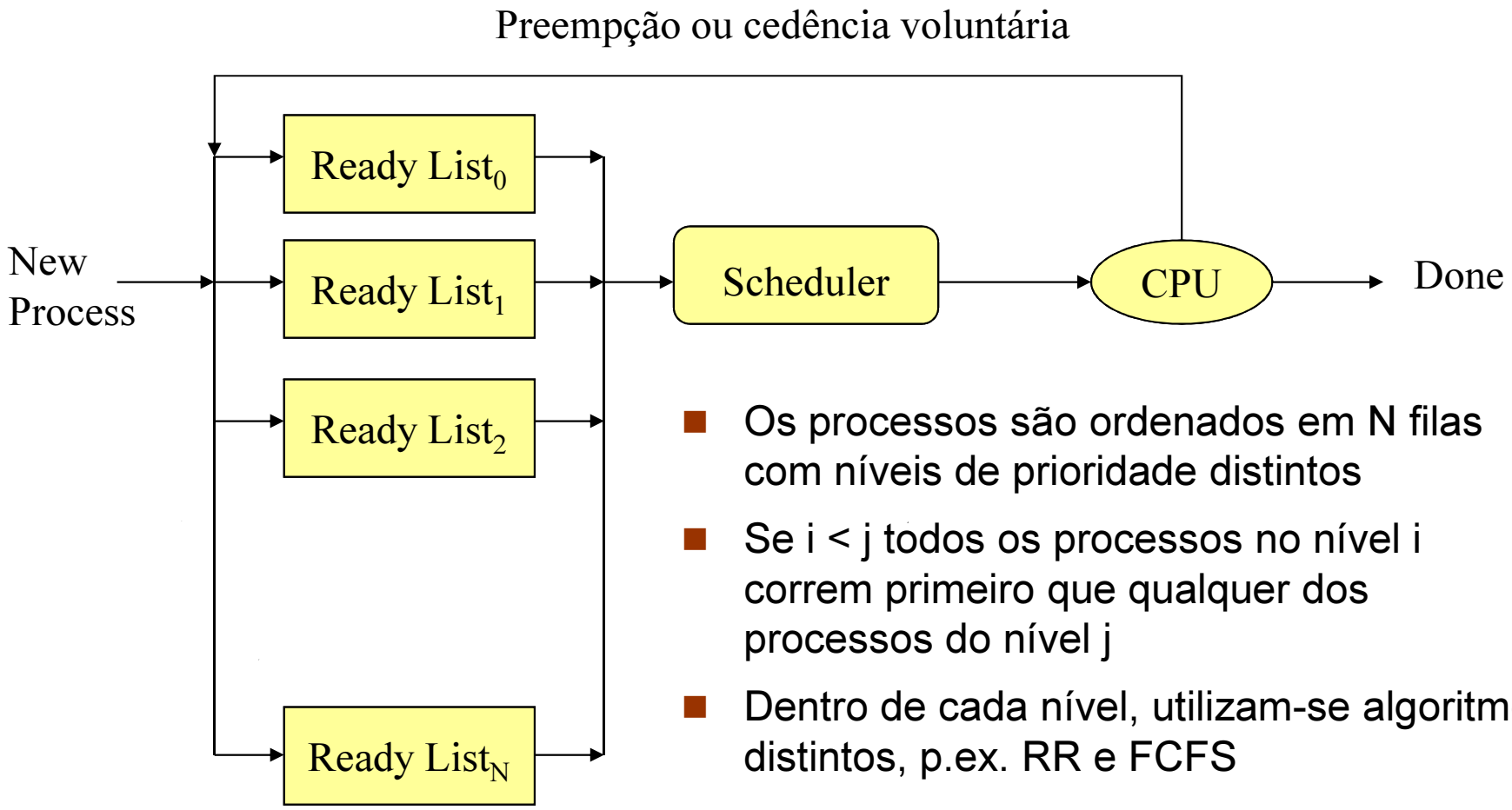
Modelo de fila de espera

Scheduling Multilevel



A Ready Queue é devidida em várias filas distintas com algoritmos de scheduling diferentes

Filas Multi-Level



- Os processos são ordenados em N filas com níveis de prioridade distintos
- Se i < j todos os processos no nível i correm primeiro que qualquer dos processos do nível j
- Dentro de cada nível, utilizam-se algoritmos distintos, p.ex. RR e FCFS

Exemplo de Scheduling Multilevel

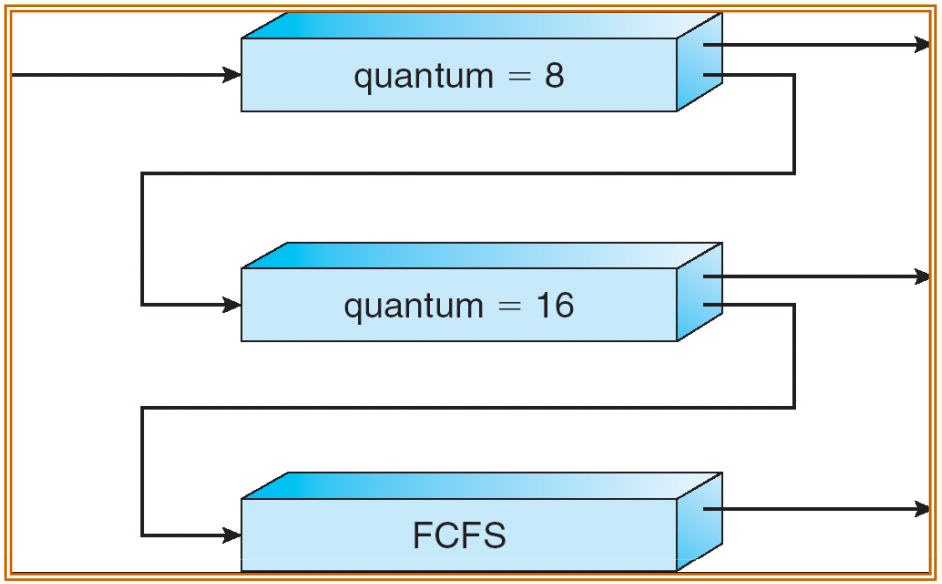
- A Ready Queue é dividida em várias filas separadas, p.ex.:
  - Foreground (interactiva)
  - Background (batch)
- Cada fila tem o seu próprio algoritmo de scheduling
  - Foreground - RR
  - Background - FCFS
- Tem de haver scheduling entre as várias filas
  - Scheduling de prioridade fixa
    - Primeiro foreground depois background ≡ starvation.
  - Time slice – cada fila recebe uma certa percentagem de CPU que é dividida entre os seus processos, p.ex.:
    - 80% para foreground em RR
    - 20% para background em FCFS

Multilevel com Feedback

- Evolução: um processo pode passar de umas filas para as outras
  - É uma forma de implementar o processo de *aging*
- O scheduling Mutilevel com feedback é definido pelos seguites parâmetros
  - Número de filas de espera
  - Algoritmos de scheduling da cada fila
  - Métodos utilizado para mover processos entre filas
    - *Upgrade* (promoção) de processos
    - *Demote* (despromoção) de processos
  - Método utilizado para determinar em que fila entra um processo quando passa para o estado Ready

Exemplo de Gestão de Filas Multilevel

- Três filas:
  - Q_0 - RR com time quantum de 8 milisegundos
  - Q_1 - RR com time quantum de 16 milisegundos
  - Q_2 - FCFS

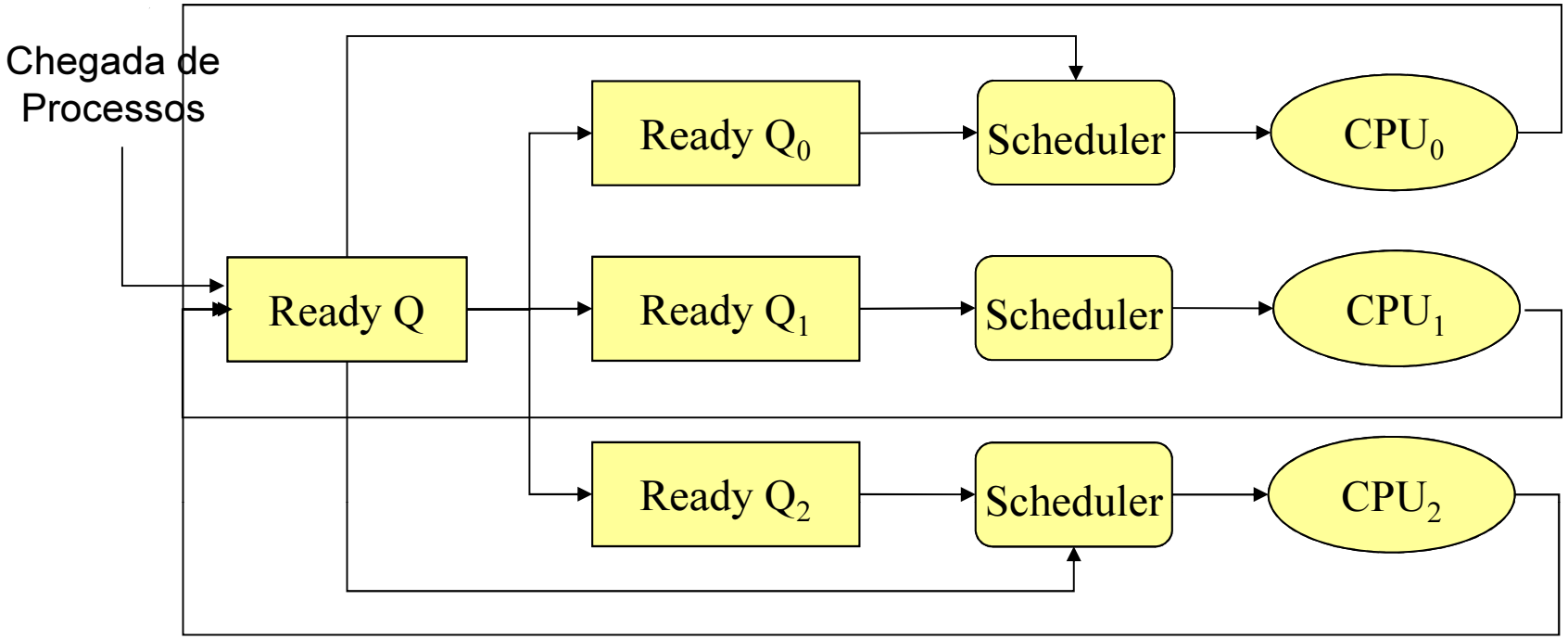


- Scheduling
  - Um novo processo entra na fila Q_0.
    - Quando recebe o CPU corre durante 8 ms.
    - Se não acaba dentro desse período para a fila Q_1
  - Na fila Q_1 recebe mais 16 ms de CPU
    - Se não acaba ainda dentro desse período, é movido para a fila Q_2
    - O processo termina quando chegar a sua vez, nos 20% de tempo CPU que são atribuídos à fila FCFS

Scheduling Multi-Processador

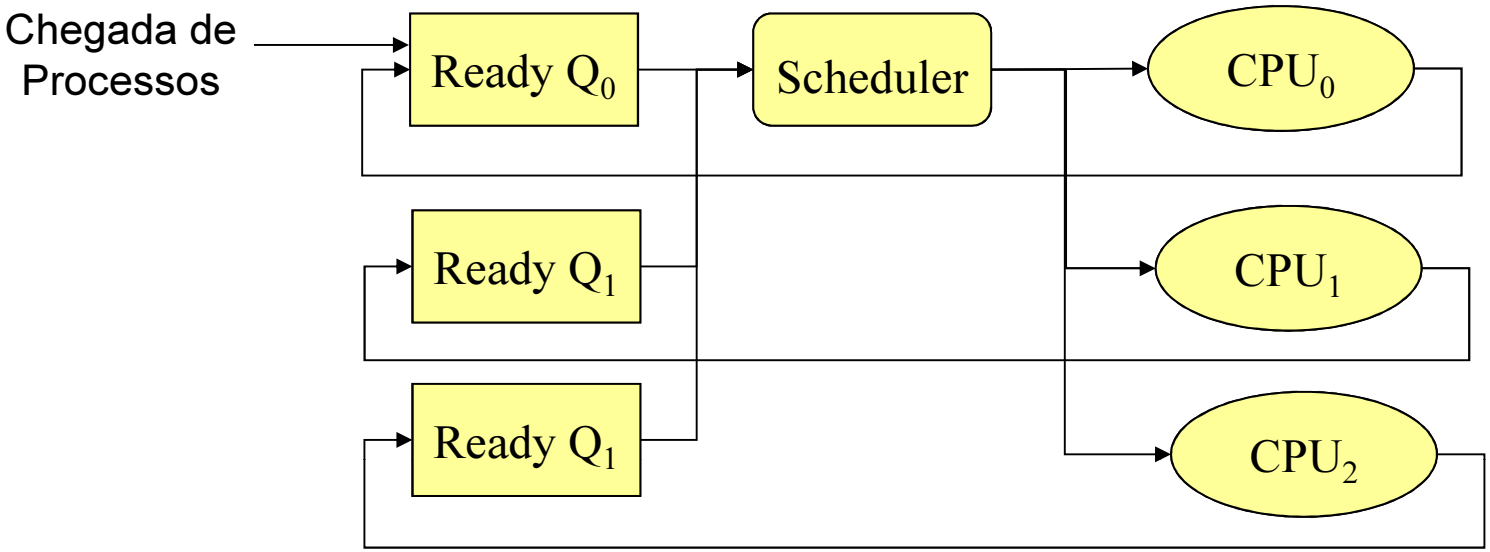
- O scheduling torna-se mais complexo quando se dispõe de vários CPUs. Vários casos são possíveis:
- Processadores indiferenciados ou simétricos (SMP)
  - Uma *ready queue* única*:* cada processador retira processos da fila sempre que fica livre
    - Problemas de afinidade (cache, MMU, etc)
  - Uma *ready queue* por processador: os processos tendem a ficar sempre no mesmo processador
    - só migram quando houver grandes assimetrias
- *Load Balancing*
  - Permite distribuir a carga de forma homogénea por todos os processadores
  - Técnica de *push*: o scheduler global examina a carga de cada processador periodicamente e distribui os processos
  - Tecnica de *pull*: o scheduler de cada processador vai buscar processos sempre que a sua *ready queue* fica vazia

Multiprocessador Simétrico



- Processadores indiferenciados ou simétricos
- Uma Ready Q_i por processador, mas cada processador também pode ir buscar à Ready Q comum
- Permite distribuir a carga de forma homogénea por todos os processadores
- Necessidade de exclusão mútua na execução dos algoritmos de *scheduling* para garantir coerência

Multiprocessador Assimétrico



- Processadores diferenciados ou assimétricos
  - Processadores com especificidades diferentes
  - I/O, Gráfico (GPU), Aritmético (FPU)
- A atribuição do CPU é feita de acordo com a necessidade de cada processo em termos do recurso específico

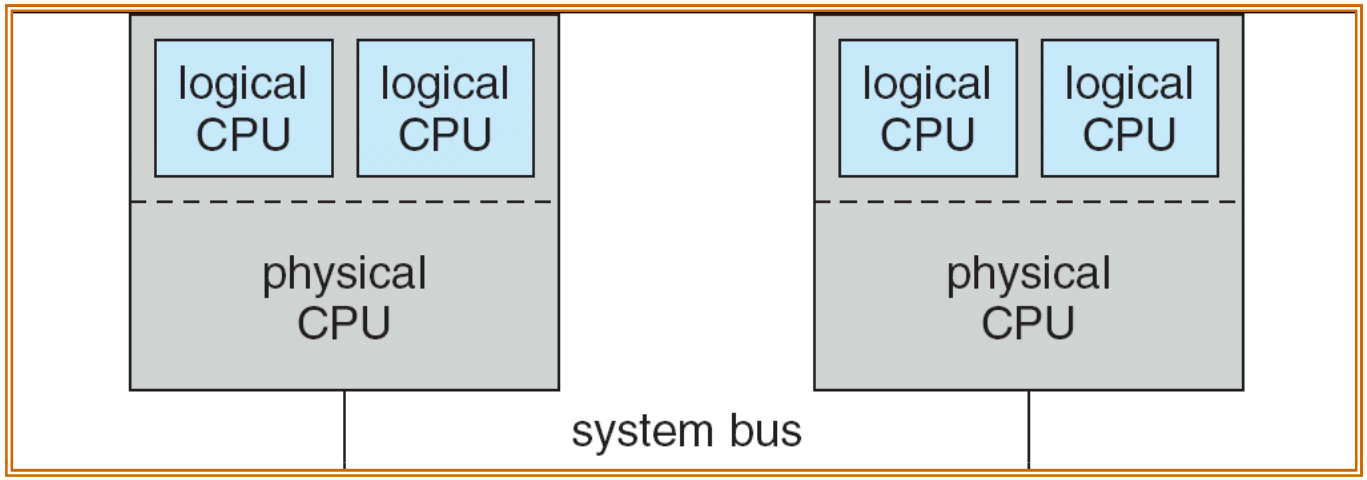
Scheduling Real-Time

- *Sistemas Hard Real-Time* – necessitam de completar uma tarefa crítica dentro de um intervalo de tempo garantido
- *Sistemas Soft Real-Time* – requerem que processos críticos tenham prioridade sobre outros menos importantes

Scheduling de Threads

- Segundo a implementação do package de threads pode haver dois tipos de scheduling
- Local: implementado na biblioteca de threads, para decidir que thread irá correr em modo utilizador associada às threads kernel disponíveis
- Global: implementado no kernel, e corresponde aos algoritmos descritos até agora
  - Scheduling Threads ≡ Scheduling Processos

Symmetric Multithreading (SMT)



- Alguns CPUs fornecem a possibilidade de criar processadores lógicos dentro de cada CPU
  - Designado por HyperThreading no mercado
- Cada CPU lógico permite executar uma thread distinta sem mudança de contexto
- Em SMP-SMT, o scheduler precisa de saber se são processadores reais ou lógicos para poder optimizar a alocação

Exemplos de Scheduling

- Alguns Exemplos de Algoritmos de Scheduling
  - Scheduling Linux
  - Scheduling Windows
  - Scheduling Solaris

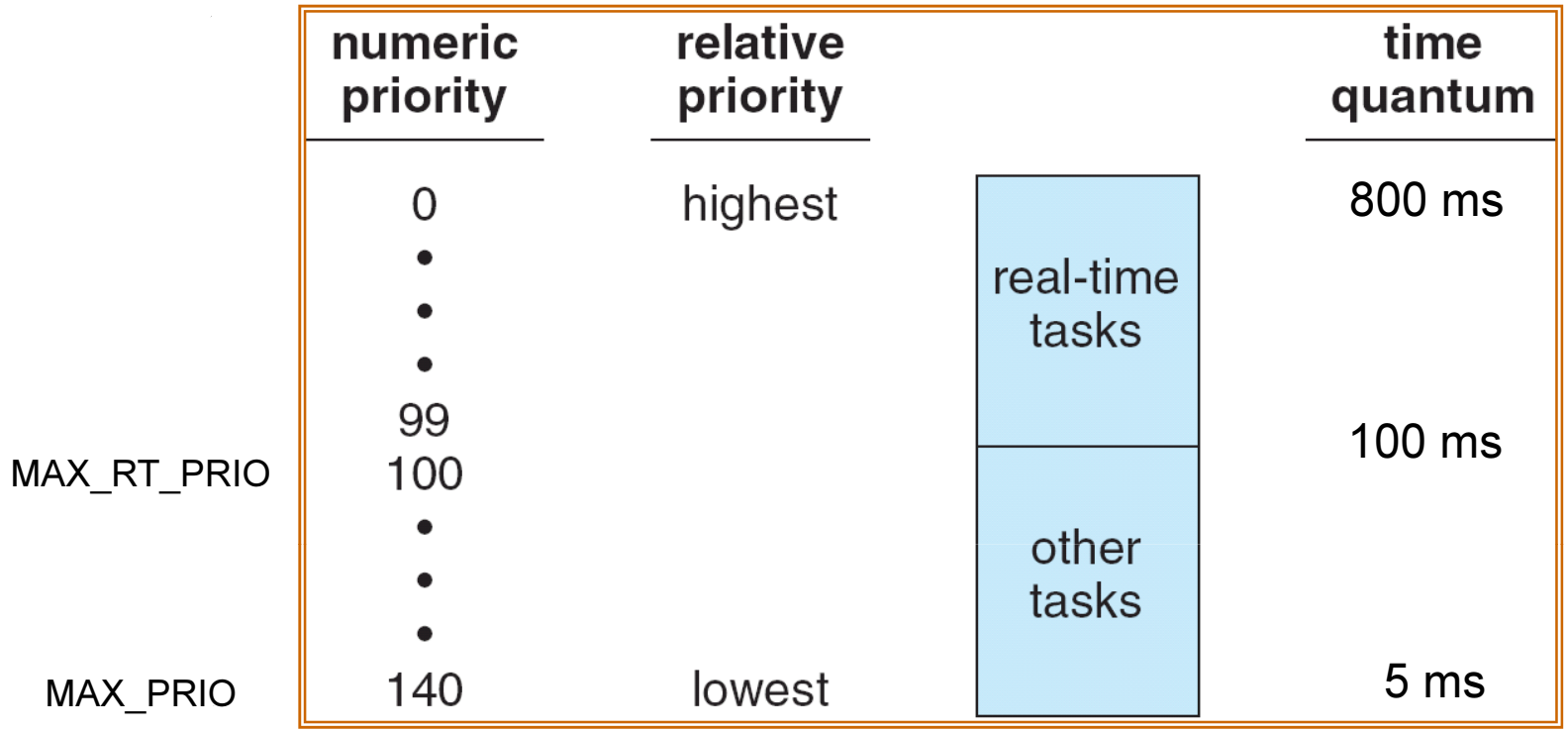
Requisitos do Scheduler Linux

- O scheduler foi completamente reformulado na versão 2.5 do kernel para garantir uma excelente escalabilidade do sistema
- Requisitos:
  - Manter excelente desempenho de aplicações interactivas mesmo em situações de carga elevada
    - Tempo de resposta de ordem constante: O(1)
  - Manter critérios equitativos e evitar *starvation*
  - Garantir eficiência em ambiente Multiprocessador (SMP)
    - Uma fila de espera por processador
  - Garantir afinidade dos processos por CPU
    - Evitar migrações desnecessárias
- Solução:
  - Utilização de diferentes algoritmos de scheduling em função do tipo de processo:
    - Round-robin preemptivo com prioridade e time quantum variáveis para processos normais
    - Soft real-time com prioridade estática para processos sistema
  - Optimização da gestão das prioridades e da selecção

Detalhes do Scheduler O(1) Linux

- Round-Robin preemptivo com *time quantum* variável para as tasks interactivas
  - A cada task é inicialmente atribuído um crédito (*time slice)* de execução, sendo inserida na *active list*
    - Valor típico de 100 ms (5 a 800 ms), indexado à prioridade
  - O *time slice* das tasks diminui à medida que vão utilizando o CPU
    - A sua prioridade é reavaliada a cada 20ms
    - Dentro da mesma prioridade á aplicado RR
  - Quando o *time slice* de uma task expira, é passada para uma outra fila de espera (*expired list*)
    - A sua prioridade é recalculada baseada no historial
    - É-lhe atribuído um novo *time slice* que depende da nova prioridade
    - É inserida na fila de espera de prioridade correspondente
    - Uma task muito interactiva pode voltar a ser inserida na *active list*
  - Quando já não há tasks na *active list*, as duas listas são trocadas
    - Expired <=> Active
- Real-time para as tasks de prioridade fixa
  - Soft real-time
  - Compatível com Posix.1b – duas classes
    - FCFS e RR
    - A task com maior prioridade passa sempre primeiro

Relação entre Prioridade e Time Quantum



- O Linux atribui um *quantum* de tempo maior aos processos com maior prioridade
  - Um processo recebe inicialmente metade do *quantum* do pai
- Real Time: de 0 a 99 (prioridades estáticas)
- Round Robin: de 100 a 139 (prioridades dinâmicas) recalculadas em função do comportamento do processo favorecendo as mais interactivas

Prioridades vistas pelo Utilizador

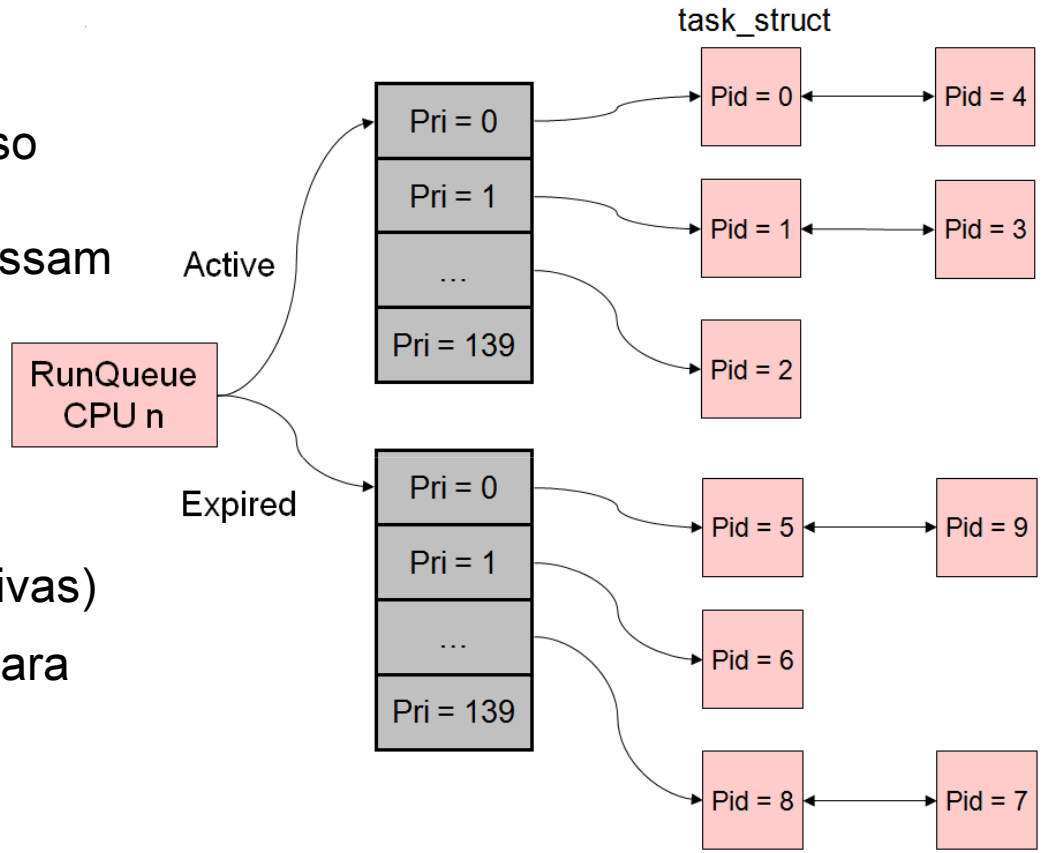
- O valor da prioridade (*nice*) visível pelo utilizador é diferente da prioridade interna:

```
USER   NI   PRI   %CPU   STAT   COMMAND
rml    0    15    0.0    S      vim
rml    0    18    0.4    S      bash
rml    0    25    91.7   R      infloop
```

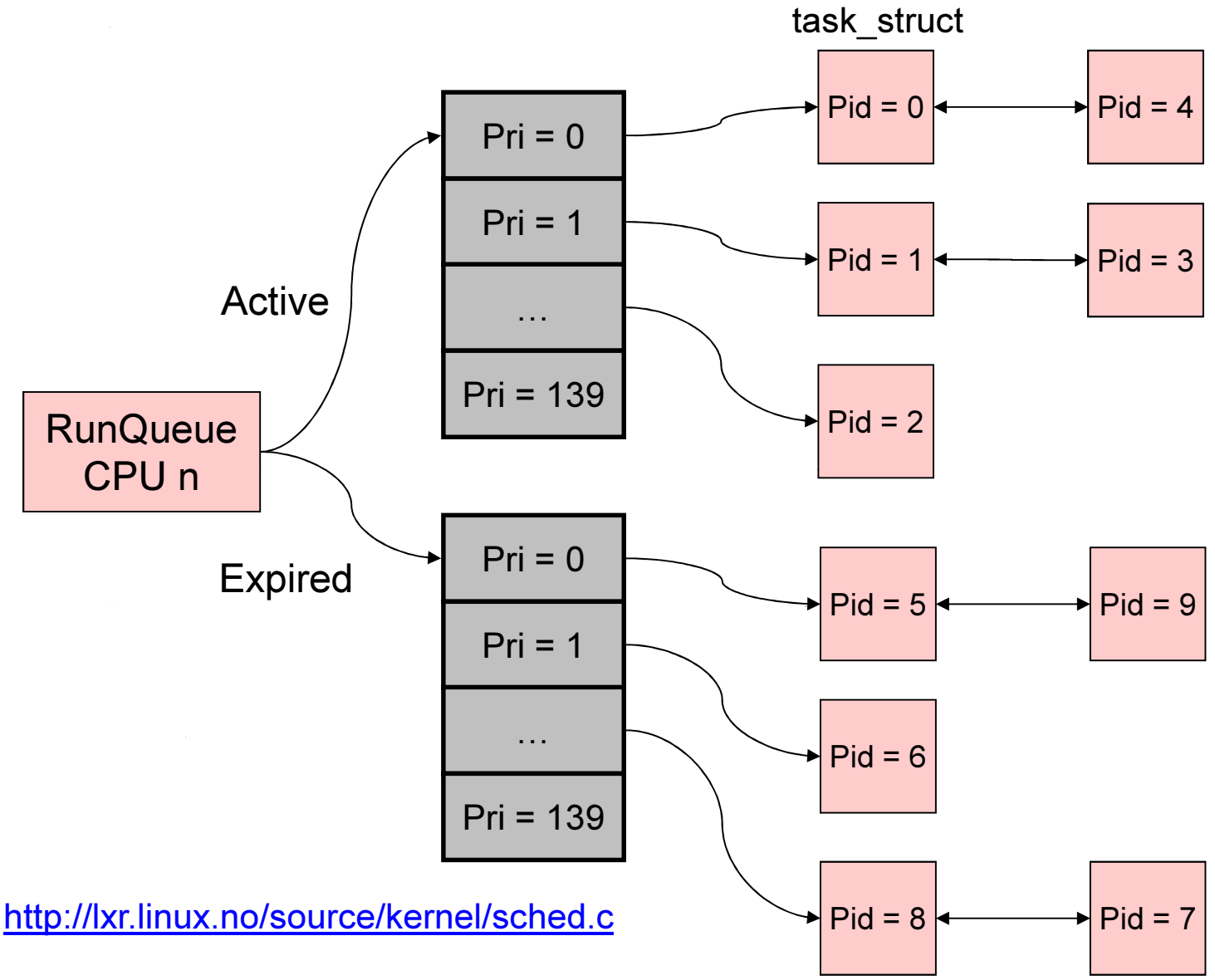
  - USER_PRIO (p) = p - MAX_RT_PRIO
  - A prioridade visível dos processos utilizador varia entre 0 e 39
  - Uma prioridade interna máxima (RT) corresponde ao valor -100
- A prioridade dos “processos” utilizador podem ser modificada de +19 (min) a -20 (max) valores em relação à prioridade normal
- Comando: `nice -n <valor> <comando>`
  - `nice -n 10 bash`
    - Executa o comando bash com a prioridade **diminuída** de 10 unidades
  - `nice -n -10 bash`
    - Executa o comando bash com a prioridade **aumentada** de 10 unidades
    - só o super user (root) pode aumentar a prioridade

Estruturas de Controlo

- A cada core são associadas duas tabelas de listas indexadas por prioridades
- As prioridades de cada processo são recalculadas a cada *time quantum* e de cada vez que passam para a tabela "*expired*"
  - Em função do tempo que esperaram por I/O
  - São favorecidas as que esperaram mais (+ interactivas)
- A mudança da tabela *expired* para *active* faz-se quando já não há processos na fila activa
  - Mudança de ponteiros na RunQueue

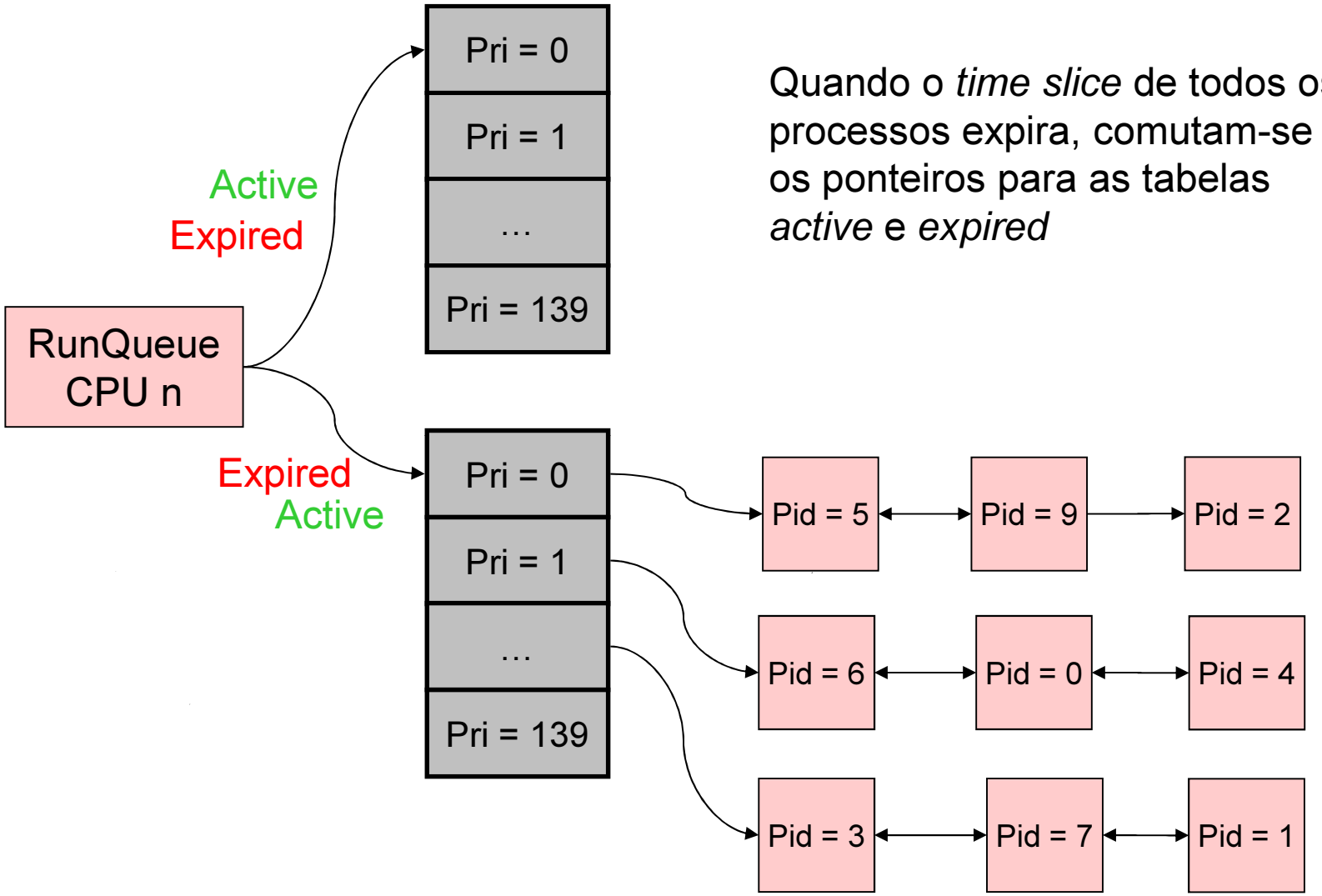


Funcionamento



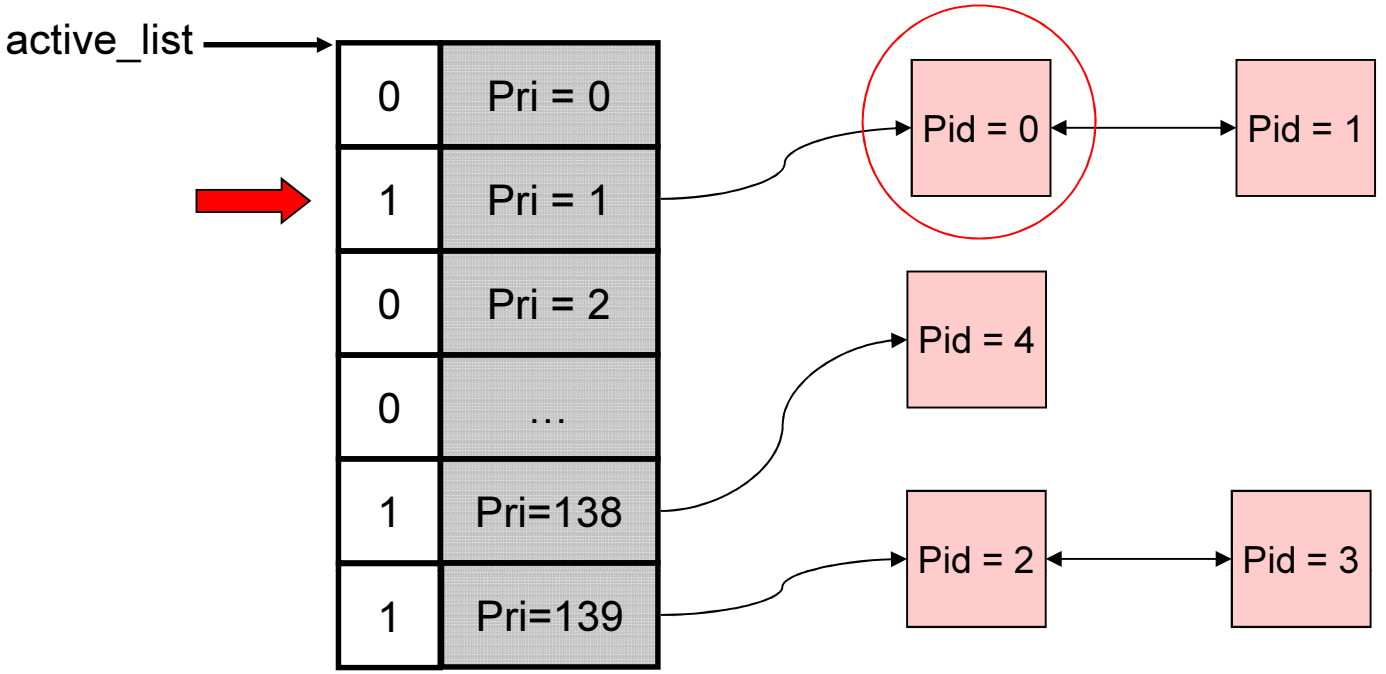
http://lxr.linux.no/source/kernel/sched.c

Comutação de Tabelas



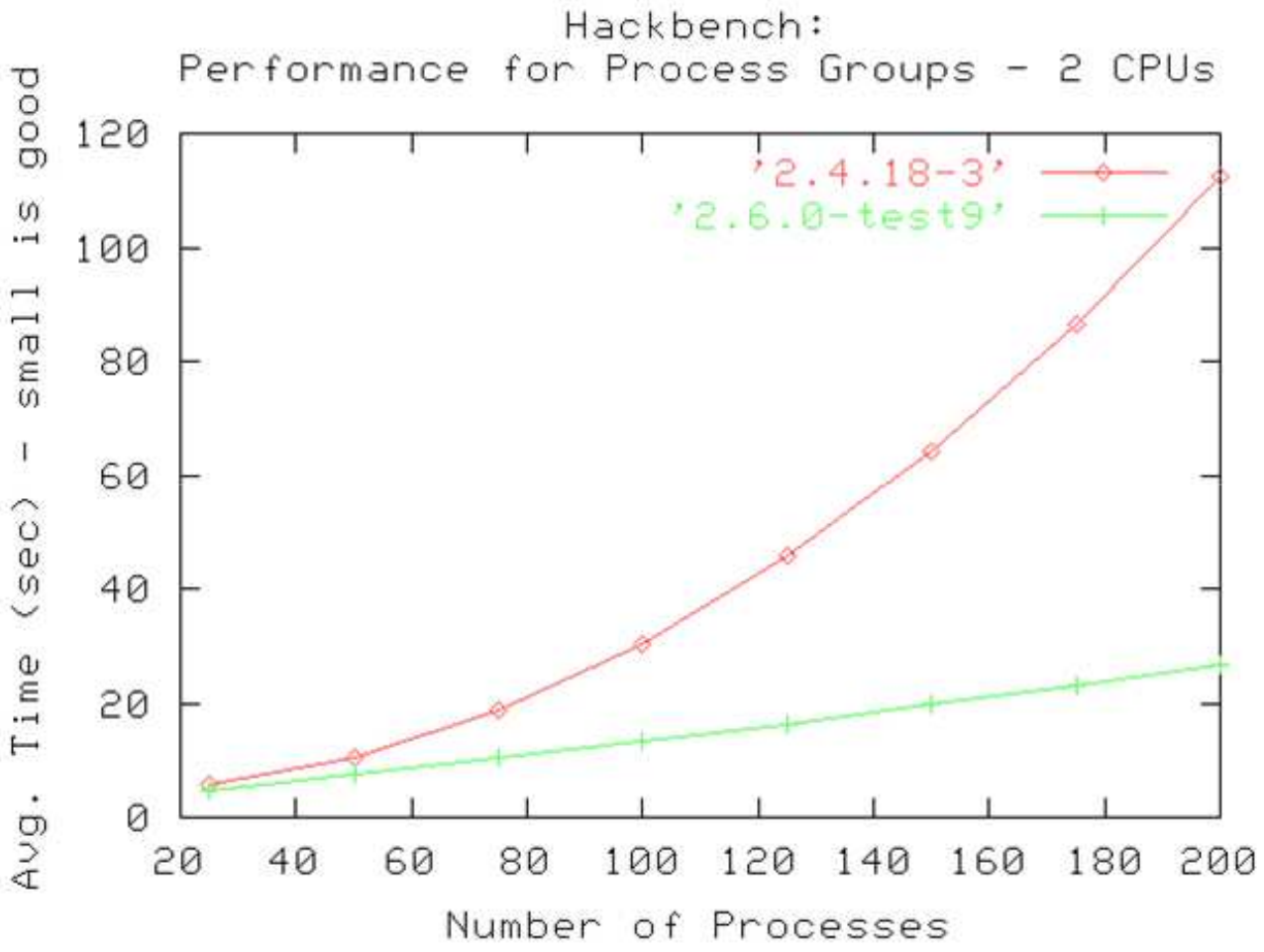
Quando o *time slice* de todos os processos expira, comutam-se os ponteiros para as tabelas *active* e *expired*

Selecção da task a activar



- A tabela de filas de espera contém um bit por cada nível de prioridade que tem valor 1 se houver *tasks* nesse nível
- Para seleccionar a task a activar basta percorrer todos os bits começando pelas mais alta prioridade e seleccionar o primeiro elemento da fila que tem o bit a 1
- Operação realizada por uma instrução assembler que devolve o índice do primeiro bit com valor 1 num registo de 32 ou 64 bits
  - 32 bits: ~5 instruções
  - 64 bits: ~3 instruções

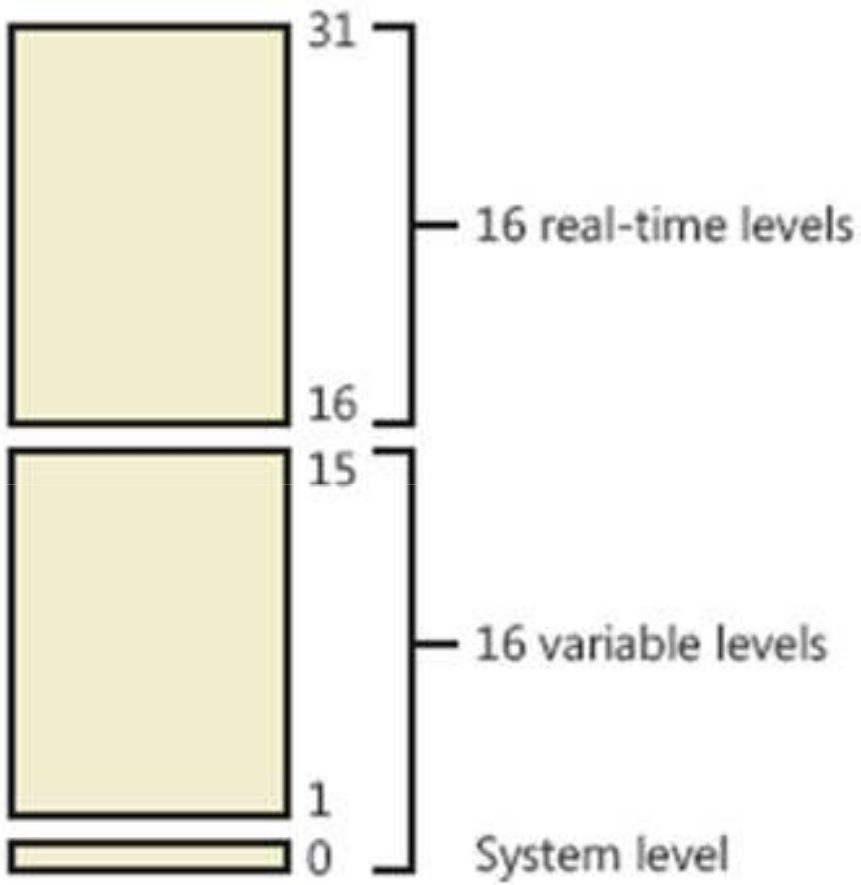
Comparação de Desempenho



Source: http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main

Scheduling Windows

- O Windows utiliza um algoritmo *multilevel* preemptivo baseado na prioridade, com prioridades e *time quantum* variáveis
  - A unidade de escalonamento é a thread
  - Utiliza condições de modificação da prioridade semelhantes às do Linux
- Cada thread pode ter 32 níveis de prioridade relativas
  - Prioridade variável: 1 a 16
  - Prioridade Fixa "Tempo Real": 16 a 31
- Inicialmente, cada thread é inicializada com a prioridade da classe do processo a que pertence



Classes de Prioridade

Classes de Prioridade

Prioridade Relativa	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

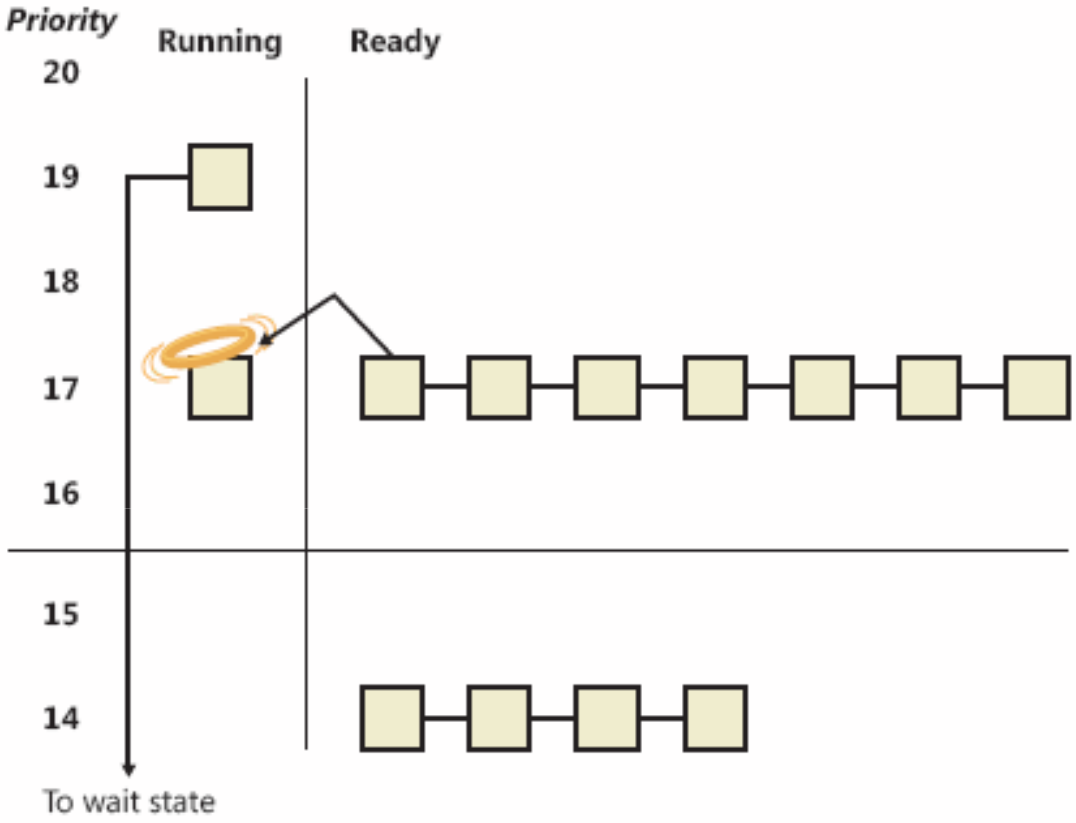
- As prioridades estão divididas em 6 classes com valores fixos que são atribuídas por processo através de funções da API Win32.
  - *SetpriorityClass*
- Dentro de cada classe os 7 níveis de prioridade relativa são ajustáveis por *thread,* por deltas negativos ou positivos
  - *SetThreadPriority*
  - *Time critical (15), Highest (2), Above Normal (1), Normal (0)*
  - *Below Normal (-1) Lowest (-2), Idle (-15)*

Real Time Priorities

"You can raise or lower thread priorities within the dynamic range in any application; however, you must have the increase scheduling priority privilege to enter the real-time range.
Be aware that many important Windows kernel-mode system threads run in the real-time priority range, so if threads spend excessive time running in this range, they might block critical system functions (such as in the memory manager, cache manager, or other device drivers)".

Source: Windows Internals 6th Ed., Russinovitch, Salomon & Ionescu, 2012, Microsoft Press

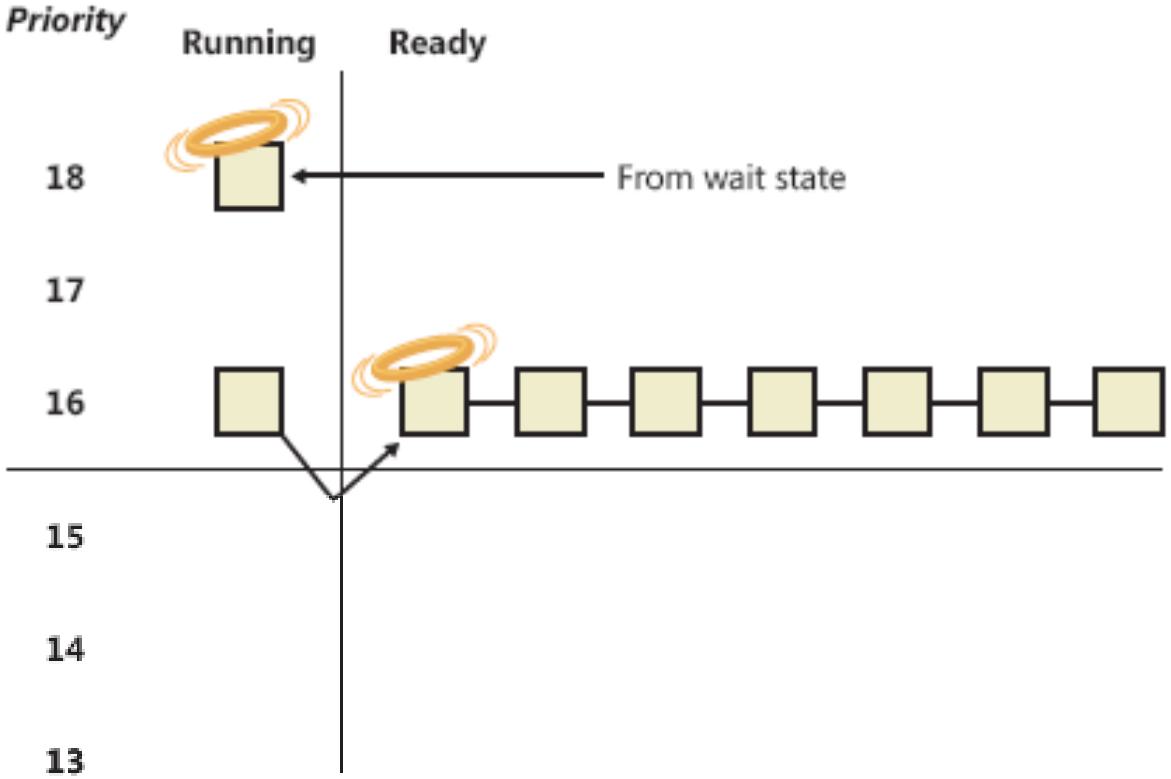
Cenários de Escalonamento: Espera



- Cedência Voluntária do CPU: uma thread entra em espera
  - A thread *running* passa para o estado *waiting*
  - É seleccionada a thread de maior prioridade que toma o seu lugar

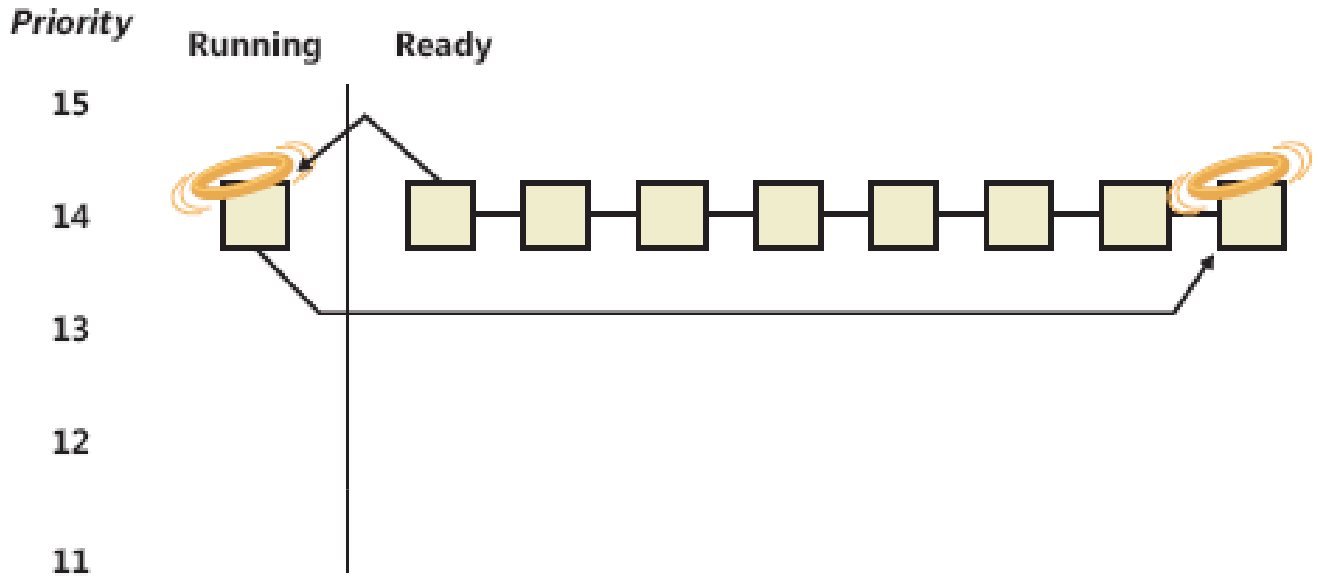
Source: Windows Internals 6th Ed., Russinovitch, Salomon & Ionescu, 2012.,Microdoft Press

Cenários de Escalonamento: Preempção



- Preempção: uma thread obtém maior prioridade o que a corrente
  - A thread *running* passa para o estado *waiting* para o topo da *wait queue*
  - A de maior prioridade toma o seu lugar

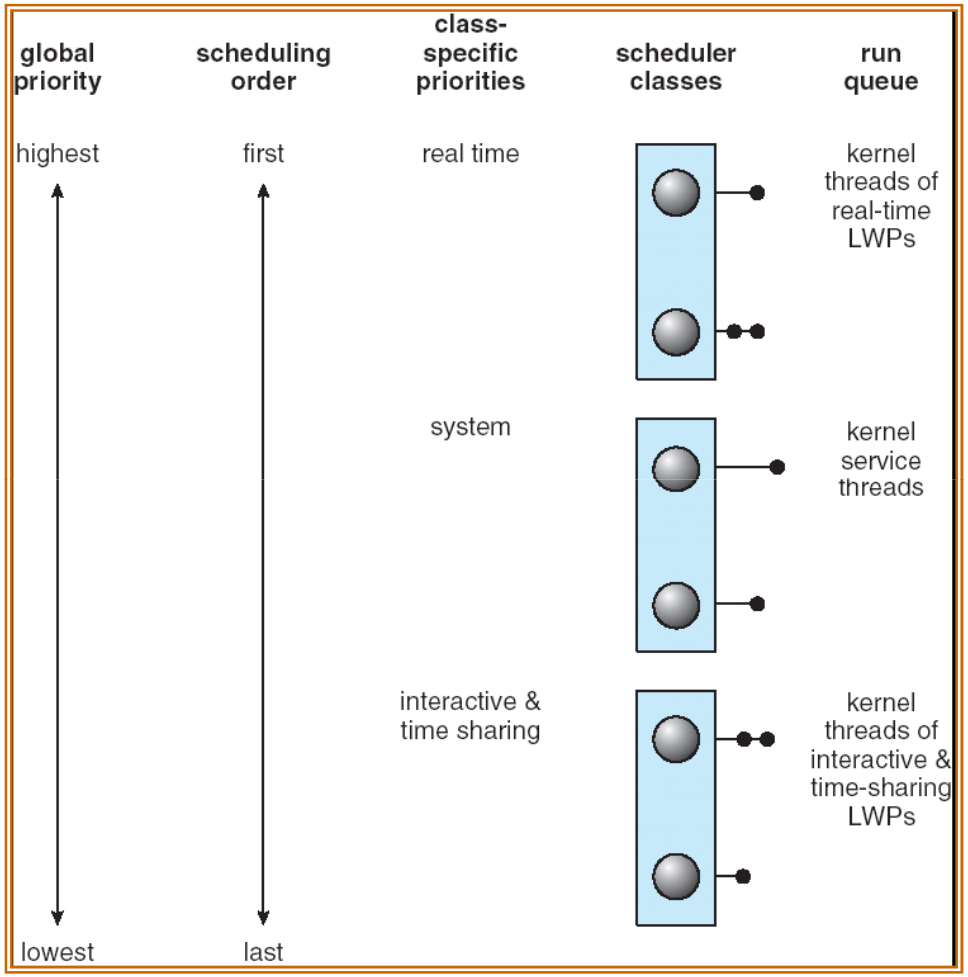
Cenários de Escalonamento: Expiração



- Expiração: uma thread expira o seu time quantum
  - A thread *running* passa para o estado *waiting* para o fim da *wait queue*
  - Se a prioridade se mantém, fica na mesma, senão passa para outra *queue*
  - É seleccionada a thread de maior prioridade que toma o seu lugar

Scheduling Solaris

- O Solaris efectua o scheduling de threads baseado na prioridade
- Tem 4 classes de prioridades
  - Real time
  - System
  - Time Sharing
  - Interactive
- A classe por defeito é *interactive* que utiliza multi-level feedback scheduling
  - Níveis de prioridade com time quantums inversamente proporcionais
  - Maior prioridade = menor time quantum
  - Aproximação inversa do Linux



Dispatch Table do Solaris

- Tabela de Dispatch para classes Interactive e TimeSharing
- Priority: as várias possíveis prioridades desta classe
- Time Quantum: o tempo de execução atribuído a cada prioridade
- Time Quantum Expired: a nova prioridade duma thread que excedeu o time quantum sem bloquear ►CPU bound
- Return from Sleep: a nova prioridade atribuída a uma thread que esteve bloqueada à espera de um evento ►I/O bound
  - A nova prioridade entre 50 e 59 favorece as threads interactivas
- As Dispatch tables podem ser carregadas dinamicamente

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Avaliação de Algoritmos

- Como se escolhem algoritmos de *scheduling* para um Sistema Operativo
  - É um dos factores que mais condiciona o desempenho
- Modelação determinística
  - Com base num perfil de processos simulado define-se a performance de execução de vários algoritmos
    - Já foi feito na aula passada
    - Mais exercícios na prática
- Modelos de Filas de Espera
  - Na maioria dos sistemas, os tipos de processos que correm variam constantemente
  - Não pode ser utilizado um modelo determinístico
  - Utilizam-se modelos matemáticos baseados em
    - Distribuição probabilística dos tempos de ciclo de CPU
    - Análise de redes de filas de espera
    - Resultados nem sempre se aproximam da realidade

Evolução dos Schedulers

- A optimização dos schedulers continua a ser uma área aberta
- É um tema muito complexo, com múltiplas soluções possíveis
- Ex: O scheduler do Linux já foi reformulado várias vezes desde a versão 4 do kernel e continua a ser objecto de optimizações e correcção de bugs
  - Scheduler O(1) -> descrito nestes slides
  - Completely Fair Scheduler (CFS) -> a versão actual
- Recentemente um artigo apresentado no EuroSys 2016 (Londres, Abril 2016) aponta expõe vários problemas no CFS, detectáveis sobretudo em computação de alto desempenho
  - The Linux Scheduler: a Decade of Wasted Cores (Lozi et al. 2016)

    http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf
- Continua…

Fim do Scheduling