Sistemas Operativos
5ª parte - Escalonamento do CPU

Prof. José Rogado
jrogado@ulusofona.pt
Prof. Pedro Gama
pedrogama@gmail.com
Universidade Lusófona

Escalonamento do CPU

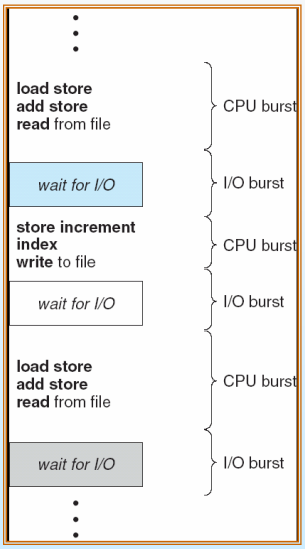
- Objectivos:
  - Perceber a importância da politica de escalonamento no comportamento dos processos
  - Perceber os problemas/vantagens inerentes
  - Discutir as políticas mais adequadas a cada utilização
  - Descrever as politicas em Windows e Linux
  - Descrever os algoritmos
- Conceitos Básicos
- 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

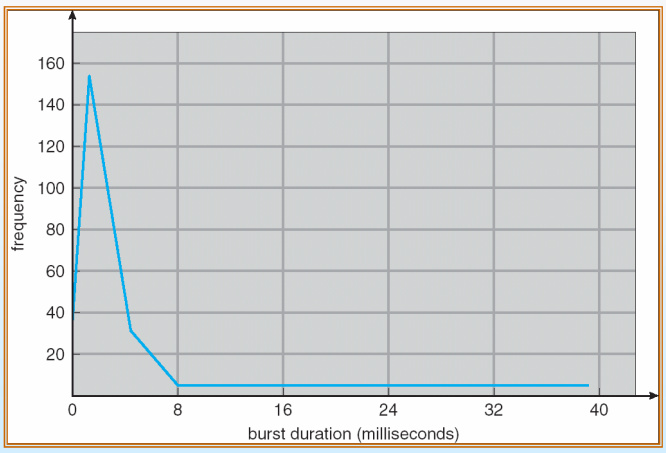
Baseia-se no seguintes factos:

- 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



Lembrar o que se discutiu sobre process CPU-bound e I/O-bound.

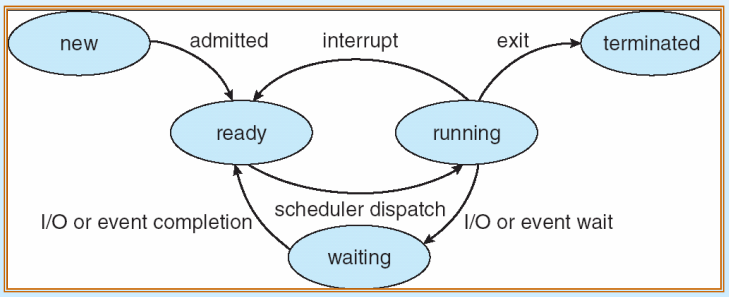
Histograma da utilização do CPU



- 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 waiting para ready
  4. Termina
- O scheduling no caso 2 é *preemptivo*
- Nos outros pontos é *não-preemptivo*



A *preemption* termo de origem latina usada em inglês, especialmente no direito, significa expropriação. Ocorre quando scheduler pára a execução de um processo sem que este tenha efectuado alguma chamada de sistema bloqueante (que obrigue o processos a esperar). As duas situações em que os schedulers preemtivos fazem isto são:

1) terminado um quantum de tempo.
2) aparecimento no estado “ready” de um processo de mais alta prioridade.

Exemplo de um SO não preemptivo: o Windows 3.x.

O aluno deve perceber qual a consequência prática de não haver preempção.

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
  - No Windows a funcionalidade de escalonamento está espalhada e misturada com o *dispatcher*
- *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

Os critérios de scheduling não são exclusivos dos SO. Também são usados em logística, administração, gestão, etc.

É importante perceber que a fluidez de interacção utlizador-máquina resulta da adequação dos critérios e capacidade de comutação de processos.

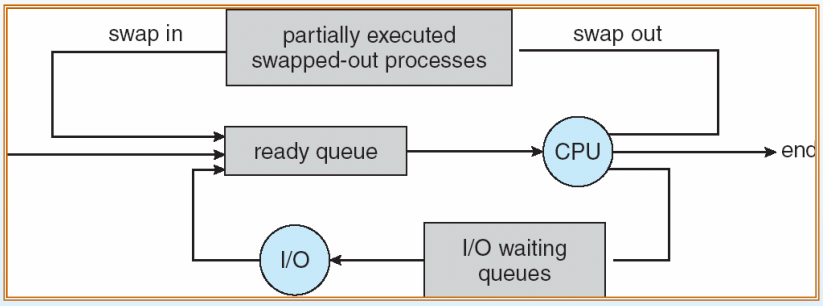
Tipos de Schedulers

- Geralmente nos Sistemas Multiprogramados existem dois tipos de Schedulers
- **Long-term scheduler** (ou job scheduler) – selecciona quais os processos que devem ser trazidos para a fila **ready**
  - Este scheduler tem um período de execução longo. Existe nos sistemas mais antigos com processamento em batch
  - Não existe nos Unix e nos Windows. Existe um scheduler de tarefas (tasks ou jobs) para uma dada hora
- **Short-term scheduler** (ou CPU scheduler) – selecciona qual o processo que está na lista **ready** que deve ser executado a seguir e atribui-lhe o CPU
  - Este scheduler tem um período de execução curto

Tipos de Schedulers (Cont.)

- **O Short-term scheduler** é invocado muito frequentemente
  - Milisegundos ⇒ tem de ser rápido
- **Long-term scheduler** é invocado pouco frequentemente
  - Segundos, minutos ⇒ pode ser lento
- O **long-term scheduler** controla o *grau de multiprogramação* do sistema (nº de processos que podem estar a ser executados em simultâneo).
- Os Processos podem ter dois tipos extremos de características:
  - **I/O-bound process** – gasta mais tempo a fazer I/O do que computação
    - muitas e curtas utilizações do CPU
  - **CPU-bound process** – gasta mais tempo a fazer computação do que I/O;
    - Poucas mas longas utilizações CPU

Introdução de Medium Term Scheduling



- O Medium Term Scheduler actua quando demasiados processos estão a competir pela utilização do CPU
  - Tira processos da wait queue (que não estão a ser executados) e envia-os para uma zona de swap em disco
  - Mais tarde podem voltar para a ready queue quando o número de processos em memória diminui

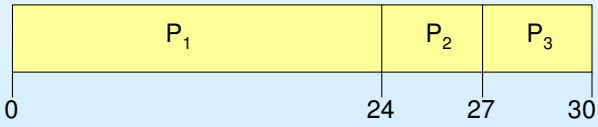
Bem perceptível no Windows c/ pouca memória

A colocação de um processo em disco (i.e. a maior parte do seu PCB e espaço de endereçamento) designa-se *swap*.

Algoritmos de Escalonamento
First-Come, First-Served (FCFS)

Processo	Tempo de CPU
P_1	24
P_2	3
P_3	3

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



- Tempo de espera para P_1 = 0; P_2 = 24; P_3 = 27
- Tempo de espera médio: (0 + 24 + 27)/3 = 17
- Implementação simples: lista FIFO

O aluno deve conhecer este algoritmos simples apesar de eles só em SO muito simples serem usados. Na prática usa-se um "mix" deste algoritmos!

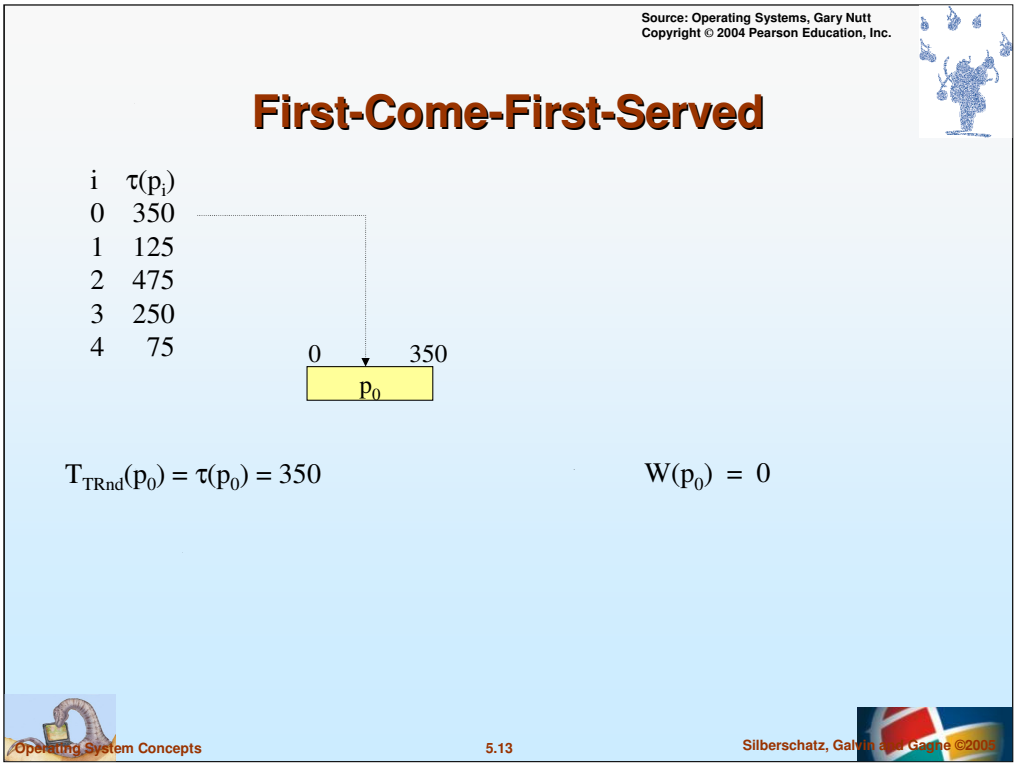
FCFS Scheduling (Cont.)

Supondo agora que os processos são criados pela ordem

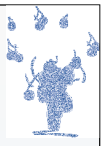
P_2 , P_3 , P_1

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

- Tempo de espera para $P_1 = 6; P_2 = 0, P_3 = 3$
- Tempo de espera médio: (6 + 0 + 3)/3 = 3
- Muito melhor do que no caso anterior
- Evitar efeito de comboio: processos de duração curta primeiro

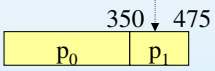


Cálculos do tempo de espera e tempo de despacho. Cálculos típicos em gestão e logística.



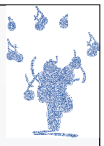
First-Come-First-Served

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



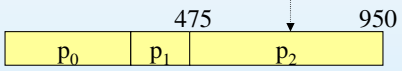
$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$



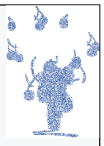
First-Come-First-Served

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



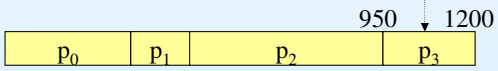
$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$



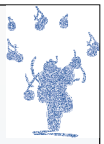
First-Come-First-Served

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



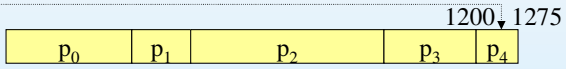
$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$



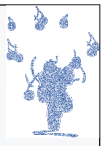
First-Come-First-Served

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



$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$
$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

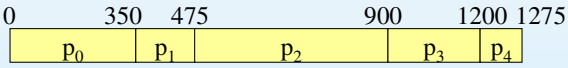
$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$
$W(p_4) = T_{TRnd}(p_3) = 1200$



FCFS Average Wait Time

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

- *Easy to implement*
- Ignores service time, etc
- Not a great performer



$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$
$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$
$W(p_4) = T_{TRnd}(p_3) = 1200$

$W_{avg} = (0+350+475+950+1200)/5 = 2974/5 = 595$

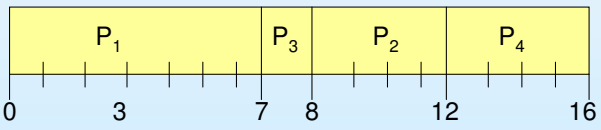
Algoritmos de Escalonamento
Shortest-Job-First (SJF)

- 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 scheduling SJF é o óptimo do ponto de vista do tempo de espera médio
  - Permite o mínimo de tempo de espera média para um dado conjunto de processos

Exemplo de SJF não preemptivo

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

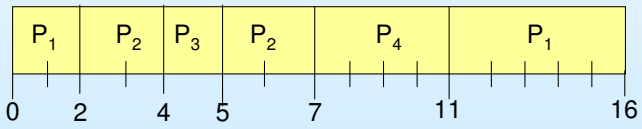


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

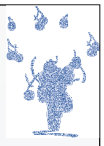
Exemplo de SJF preemptivo

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

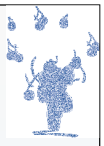


Shortest Job Next

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

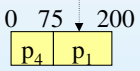
$T_{TRnd}(p_4) = \tau(p_4) = 75$

$W(p_4) = 0$



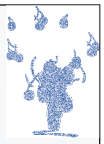
Shortest Job Next

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



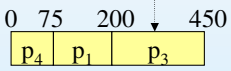
$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$



Shortest Job Next

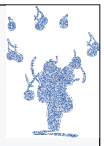
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

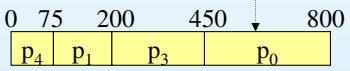
$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$



Shortest Job Next

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

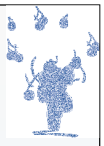


$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

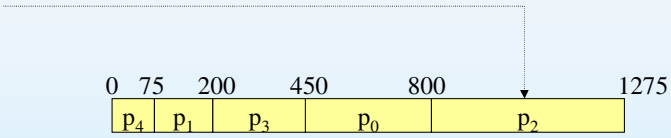
$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$



Shortest Job Next

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



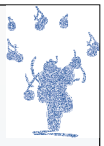
$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 475+350+250+125+75 = 1275$ $W(p_2) = 800$

$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

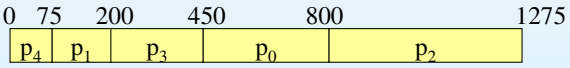
$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$



Shortest Job Next

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

- *Minimizes wait time*
- May starve large jobs
- Must know service times



$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$
$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$
$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 475+350+250+125+75 = 1275$
$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$
$T_{TRnd}(p_4) = \tau(p_4) = 75$

$W(p_0) = 450$
$W(p_1) = 75$
$W(p_2) = 800$
$W(p_3) = 200$
$W(p_4) = 0$

$W_{avg} = (450+75+800+200+0)/5 = 1525/5 = 305$

Determinação da Duração do Ciclo de CPU

- A determinação da duração do próximo ciclo de utilização de CPU só pode ser feita por estimativa
  - Baseada nos valores dos ciclos anteriores
- Pode ser estimada utilizando a duração dos ciclos prévios, utilizando uma média exponencial
- Define-se:
  1. t_n = duração do ciclo de CPU *n*
  2. τ_{n+1} = valor estimado para o próximo ciclo
  3. Sendo 0 <= α <= 1
  4. $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$
- 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)

Estimação dinâmica da duração do proximo ciclo é usada nos modernos SO.

Propriedades da Média Exponencial

Interpretação dos valores limites:

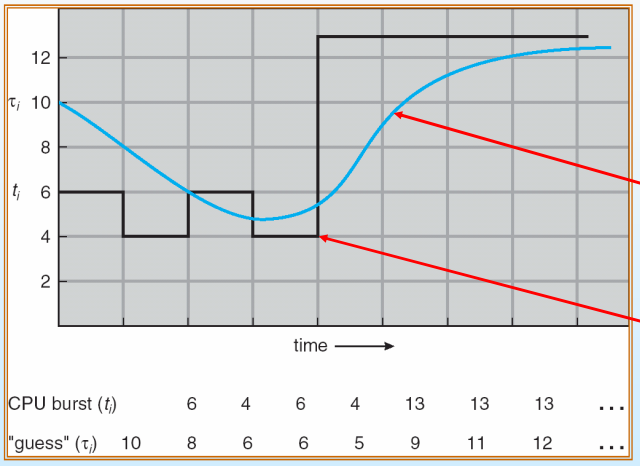
- Se $\alpha = 0$ então $\tau_{n+1} = \tau_n$
  - Valor previsto para o ciclo anterior: o valor efectivo anterior não entra em linha de conta, só é considerado o valor estimado
- Se $\alpha = 1$ então $\tau_{n+1} = t_n$
  - Valor real efectivo do ciclo anterior: só é considerada a duração do ciclo anterior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

Esta forma de estimação é usada correntemente em logística, por exemplo.

Exemplo de Estimativas



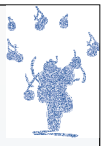
Valores obtidos com $\alpha = 0.5$ e $\tau_0 = 10$

Escalonamento Baseado na Prioridade

- A prioridade é representada por um valor inteiro associado ao processo (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.

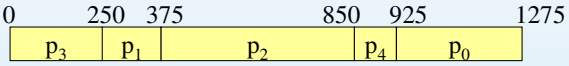
Importante perceber o mecanismo de **Aging**

Priority Scheduling



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

- *Reflects importance of external use*
- May cause starvation
- Can address starvation with aging



$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_4)+\tau(p_2)+\tau(p_1))+\tau(p_3) = 350+75+475+125+250 = 1275$ $W(p_0) = 925$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_3) = 125+250 = 375$ $W(p_1) = 250$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_1)+\tau(p_3) = 475+125+250 = 850$ $W(p_2) = 375$

$T_{TRnd}(p_3) = \tau(p_3) = 250$ $W(p_3) = 0$

$T_{TRnd}(p_4) = \tau(p_4)+ \tau(p_2)+ \tau(p_1)+\tau(p_3) = 75+475+125+250 = 925$ $W(p_4) = 850$

$$W_{avg} = (925+250+375+0+850)/5 = 2400/5 = 480$$

Scheduling em Round Robin (RR)

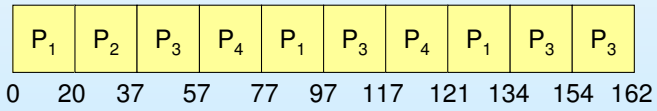
- Cada processo recebe uma pequena fracção do tempo de CPU (*time quantum*), da ordem dos 10-100 milissegundos
- Depois de esgotado deste tempo, o processo é interrompido e passado para o fim da fila de espera (*Ready Queue*)
- Se houver ***n*** processos na *Ready Queue* e o quantum de tempo for ***q***, 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.

É essencial perceber este algoritmo simples, pois ele é sempre utilizado nos SO de uso genérico.

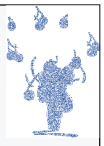
Exemplo de RR com Time Quantum = 20

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)



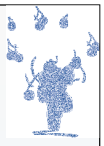
Round Robin (TQ=50)

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

0 50

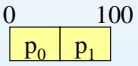
p_0

$W(p_0) = 0$

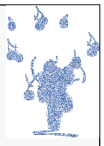


Round Robin (TQ=50)

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

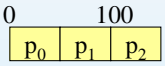


$W(p_0) = 0$
$W(p_1) = 50$

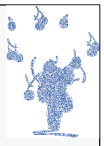


Round Robin (TQ=50)

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

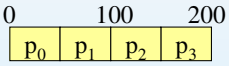


$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$

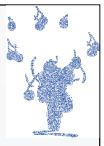


Round Robin (TQ=50)

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

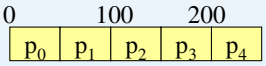


$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$

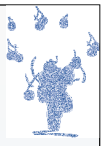


Round Robin (TQ=50)

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

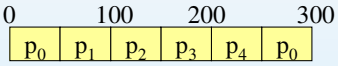


$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

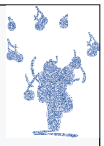


Round Robin (TQ=50)

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

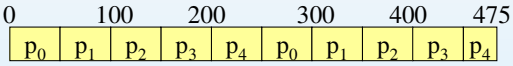


$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$



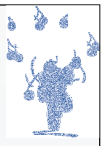
Round Robin (TQ=50)

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



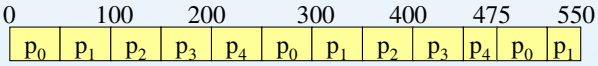
$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

$T_{TRnd}(p_4) = 475$



Round Robin (TQ=50)

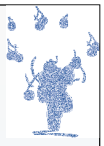
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_1) = 550$

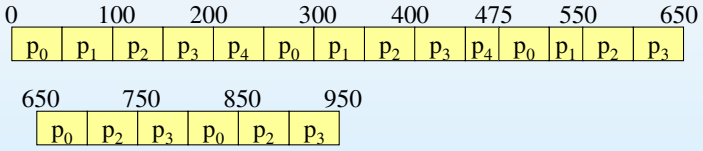
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$



Round Robin (TQ=50)

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

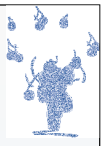


$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_3) = 950$

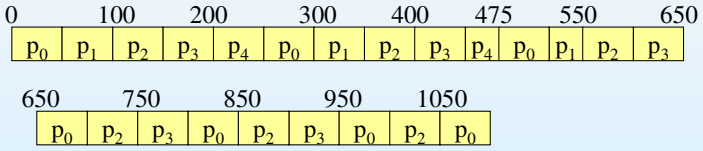
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$



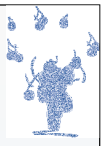
Round Robin (TQ=50)

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



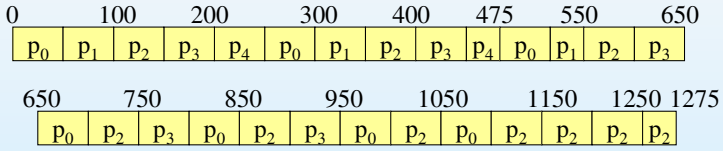
$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$



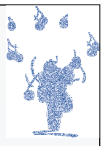
Round Robin (TQ=50)

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



$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_2) = 1275$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

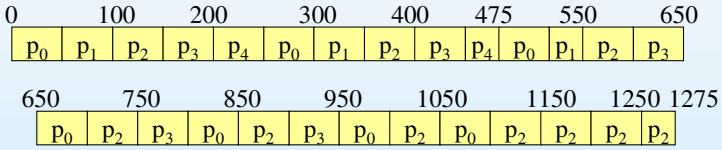
$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$



Round Robin (TQ=50)

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

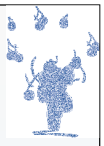
- Equitable
- Most widely-used
- Fits naturally with interval timer



$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_2) = 1275$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

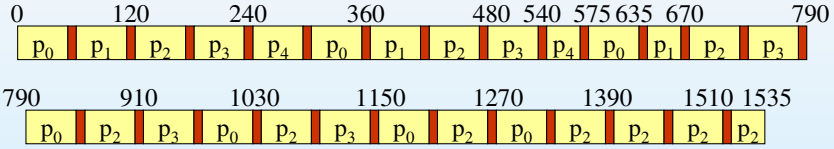
$T_{TRnd-avg} = (1100+550+1275+950+475)/5 = 4350/5 = 870$
$W_{avg} = (0+50+100+150+200)/5 = 500/5 = 100$



RR with Overhead=10 (TQ=50)

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

- Overhead must be considered

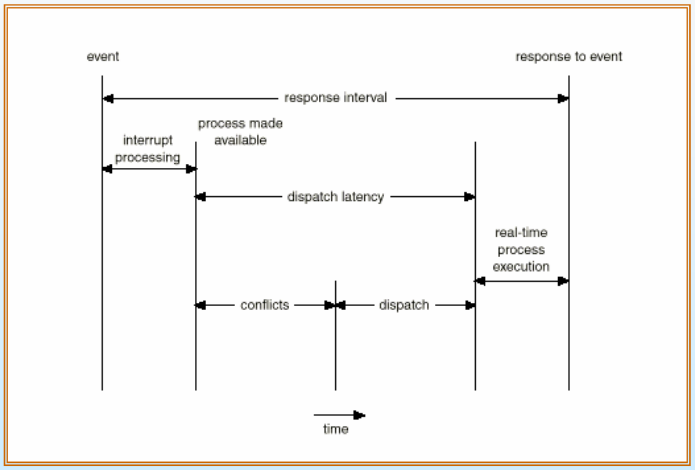


$T_{TRnd}(p_0) = 1320$ $W(p_0) = 0$
$T_{TRnd}(p_1) = 660$ $W(p_1) = 60$
$T_{TRnd}(p_2) = 1535$ $W(p_2) = 120$
$T_{TRnd}(p_3) = 1140$ $W(p_3) = 180$
$T_{TRnd}(p_4) = 565$ $W(p_4) = 240$

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

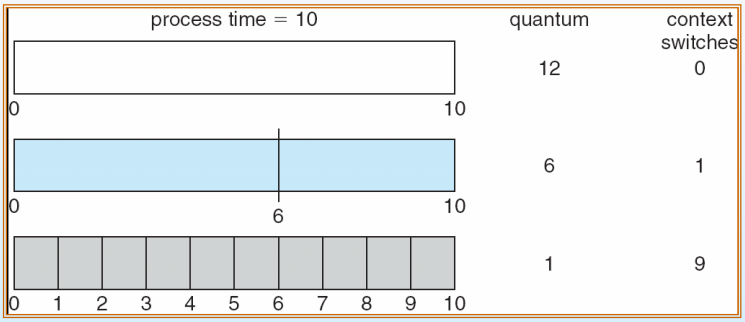
$W_{avg} = (0+60+120+180+240)/5 = 600/5 = 120$

Dispatch Latency



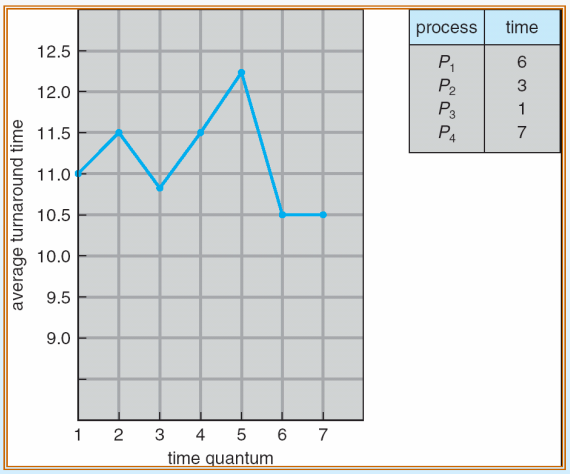
Enfatizar que a comutação implica um *overhead*, sempre presente. Em função deste *overhead* pode não fazer sentido efectuar uma comutação ou definir quanta inferiores.

Quantum e Mudança de Contexto



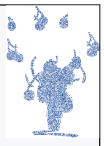
- À medida que o time quantum diminui, aumenta o número de mudanças de contexto durante a execução do processo
- Típicamente, o tempo de comutação de contexto é da ordem dos 10 μs, ou seja um milésimo do valor do quantum habitual (10-100 ms)

Tempo de Resposta vs. Time Quantum



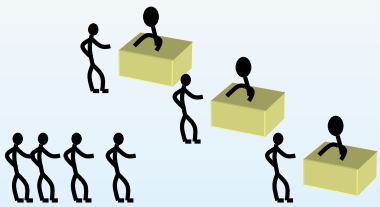
process	time
P_1	6
P_2	3
P_3	1
P_4	7

- 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

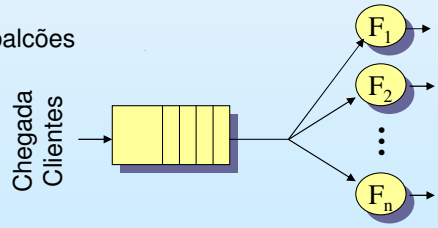


Scheduling Multilevel

Analogia com o processo de atendimento com vários balcões

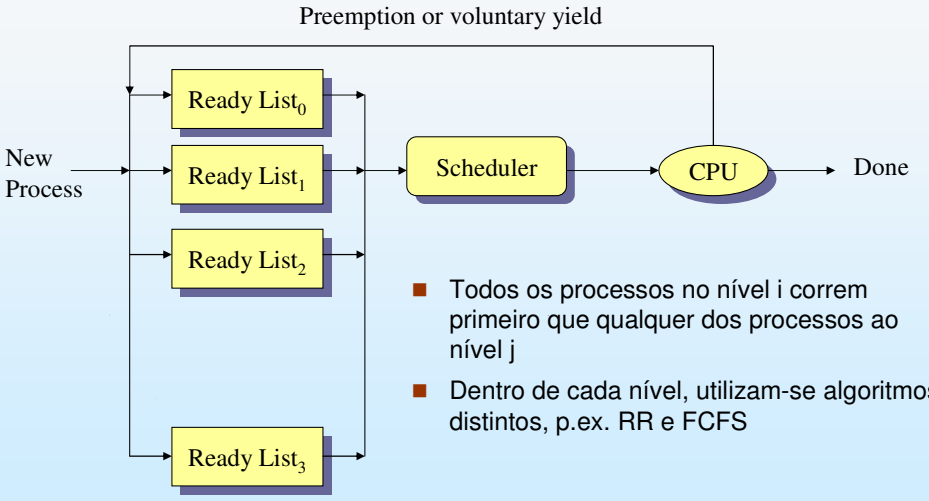


Fila de espera para múltiplos balcões



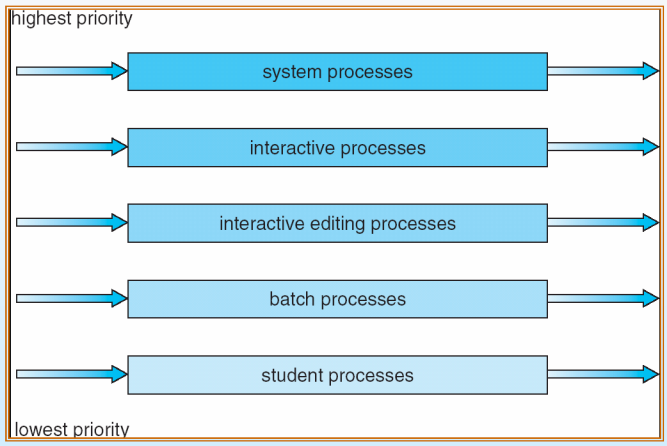
Modelo de fila de espera

Multi-Level Queues



- Todos os processos no nível i correm primeiro que qualquer dos processos ao nível j
- Dentro de cada nível, utilizam-se algoritmos distintos, p.ex. RR e FCFS

Scheduling Multilevel



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

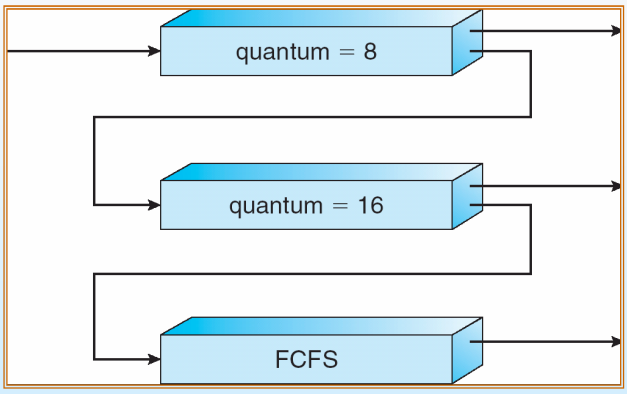
Implementação Scheduling Multilevel

- 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

Multilevel Feedback Queues



Os processos podem passar de umas filas para as outras

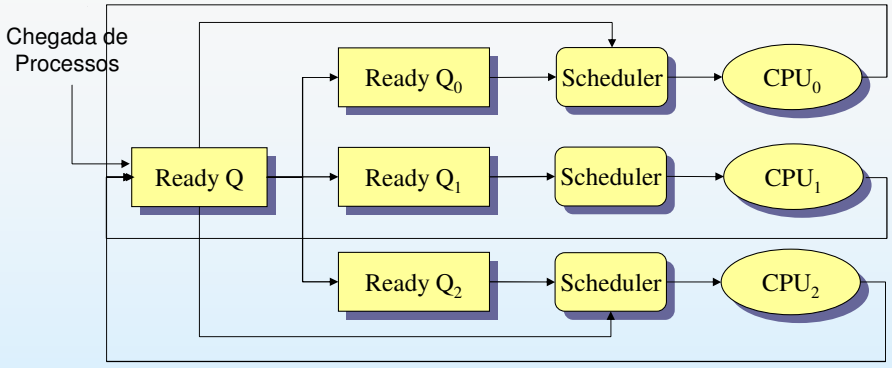
Exemplo de Gestão de Filas Multilevel

- 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

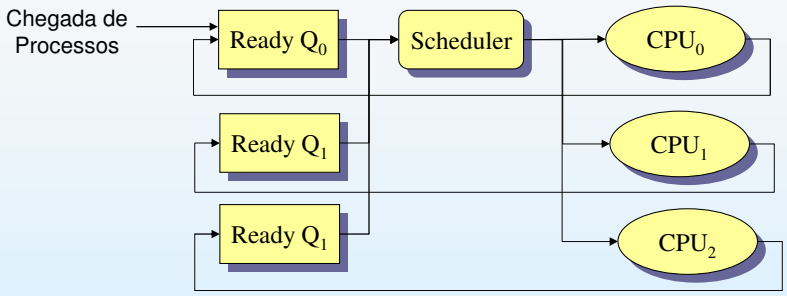
- O scheduling torna-se mais complexo quando se dispõe de vários CPUs. Vários casos são possíveis:
- Processadores indiferenciados ou simétricos (SMP)
  - Uma *ready queue* única*:* cada processador retira processos da fila sempre que fica livre
    - Problemas de afinidade (cache de memória, MMU, etc)
  - 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*: um processo examina a carga de cada processador periodicamente e distribui os processos
  - Tecnica de *pull*: 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: cada processador tem uma especificidade diferente
  - I/O, Gráfico, Cálculo virgula flutuante (FPP)
- Só um processador executa o código do scheduler e acede as Ready Queues, evitando a necessidade de partilhar dados
- A atribuição do CPU é feita de acordo com a necessidade de cada processo em termos do recurso específico

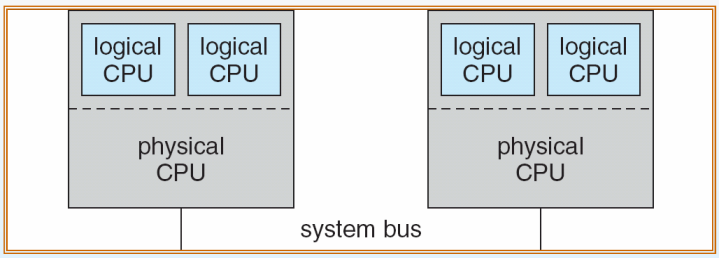
Scheduling Real-Time

- *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

API de Scheduling das pthreads

- As pthreads permitem seleccionar se o scheduling é realizado a nível utilizador ou a nível do kernel
  - Contention Scope
    - PCS (Process Contention Scope): o scheduling é feito primeiro a nível do processo que contém várias threads
    - SCS (System Contention Scope): o scehduling é feito a nível das threads do kernel
  - Alguns sistemas só suportam um destes esquemas
    - Linux, MacOSx só suportam SCS
    - Solaris suporta ambos

Perceber que a API POSIX para threads permite a definição para cada threads de diferentes politicas de scheduling, para o **programador**, poder ajustar finamente às necessidades da aplicação.

API de Scheduling: Exemplo

```c
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

#define NUM_THREADS 5
int main(int argc, char *argv[])
{
  int i;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;
  void *runner(void *);
  int scope;

  /* get the default attributes */
  pthread_attr_init(&attr);
  printf("Scope values are: PROCESS = %d SYSTEM %d\n", PTHREAD_SCOPE_PROCESS,
                                                       PTHREAD_SCOPE_SYSTEM);
 /* set the scheduling algorithm to PROCESS or SYSTEM */
  if ((errno = pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS)) != 0)
    perror("Scope");
  pthread_attr_getscope(&attr, &scope);
  printf("Scope: %d\n", scope);
  /* set the scheduling policy - FIFO, RT, or OTHER */
  pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
  /* create the threads */
  for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i],&attr,runner,(void *)i);
  /* now join on each thread */
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *index)
{
  int scope;
  pthread_attr_t attr;

  printf("I am thread %d\n", index);
  pthread_attr_getscope(&attr, &scope);
  printf("Scope: %d\n", scope);

  pthread_exit(0);
}
```

A título de Exemplo.

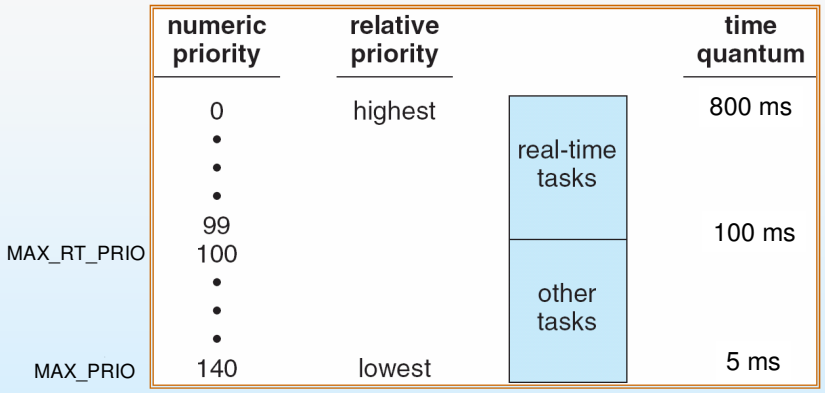
Exemplos de Scheduling

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

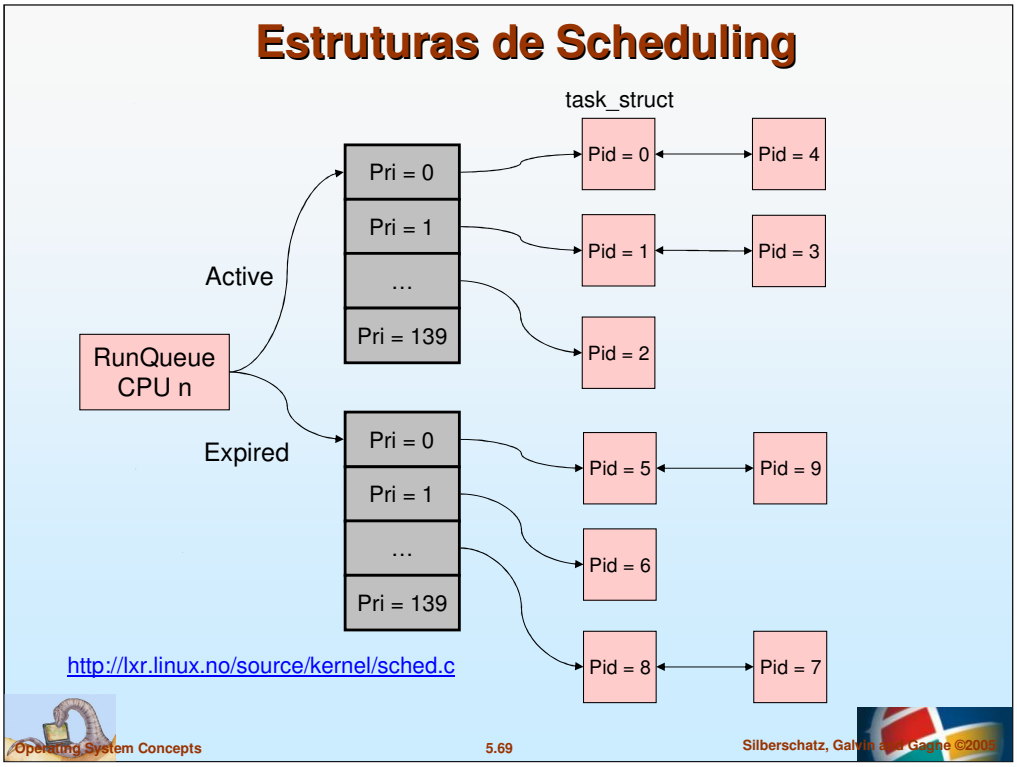
Scheduling Linux

- O scheduler foi completamente reformulado na versão 2.5 do kernel
- Objectivo
  - Manter excelente desempenho de aplicações interactivas mesmo em situações de carga elevada
    - Tempo de resposta linear
  - 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
- Utiliza vários algoritmos de scheduling diferentes segundo o tipo de processo:
  - Round-robin preemptivo com prioridade e time quantum variáveis
  - Soft real-time com prioridade estática
    - FCFS
    - RR

Relação entre Prioridade e Time Quantum



- Contrariamente a outros sistemas, o Linux atribui *quanta* de tempo maiores aos processos com maior prioridade
- 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



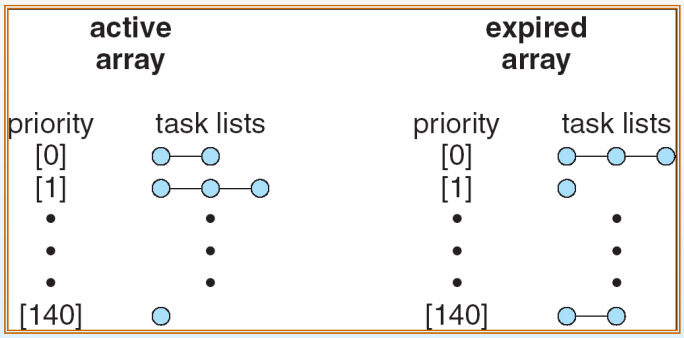
Ilustrativo, mas as estruturas de dados usadas devem ser já familiares ao aluno!

Algoritmos Scheduling Linux

- Round-Robin com *time quantum* 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 interactica 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

O aluno tem que perceber que os SO genéricos usam um mix dos diferentes algoritmos simples descritos e saber interpretar o comportamento observável dos processos como consequência destes algoritmos.

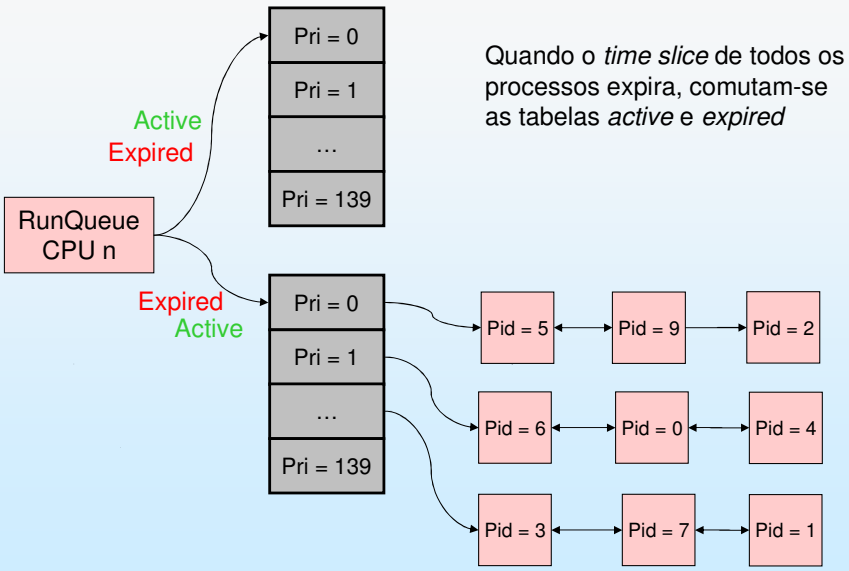
Duas tabelas de listas indexadas por prioridades



- 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

Ilustrativo da forma como o Kernel Linux moderno gere o escalonamento.

Comutação de Tabelas



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

Prioridades vistas pelo Utilizador

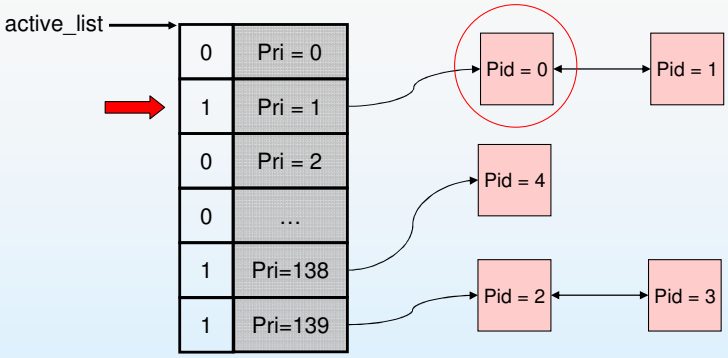
- O valor da prioridade (*nice*) visível pelo utilizador é diferente da prioridade interna:
  - USER_PRIO (p) = p - MAX_RT_PRIO
  - A prioridade visível varia entre 0 e 39
  - Uma prioridade interna máxima (RT) aparece com o valor -100

```
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
```

- A prioridade dos "processos" utilizador podem ser alterada 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 autorizada ao super user

Informação prática que é necessário ter em mente na utlização em UNIX.

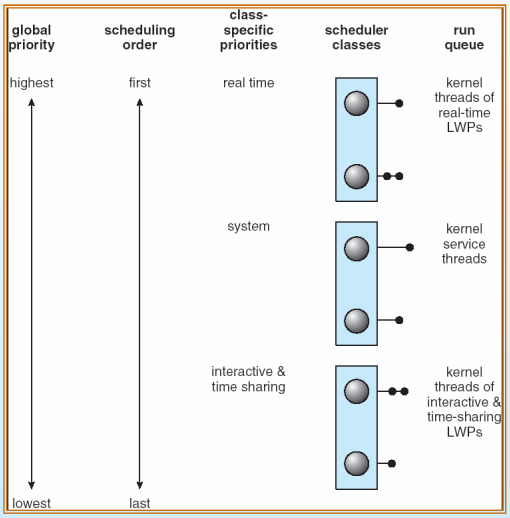
Selecção da task a activar



- 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

Scheduling Solaris

- Solaris usa thread scheduling 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



Ilustrativo. Enfatiza que os SO modernos usam mix de algoritmos.

Dispatch Table do Solaris

- Tabela de Dispatch para classes Interactive e TimeSharing
- Priority: as várias possíveis prioridades desta classe
- Time Quantum: os quanta de tempo 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

\- (high priority)

\+ (low priority)

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

Ilustrativo.

Scheduling Windows XP

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

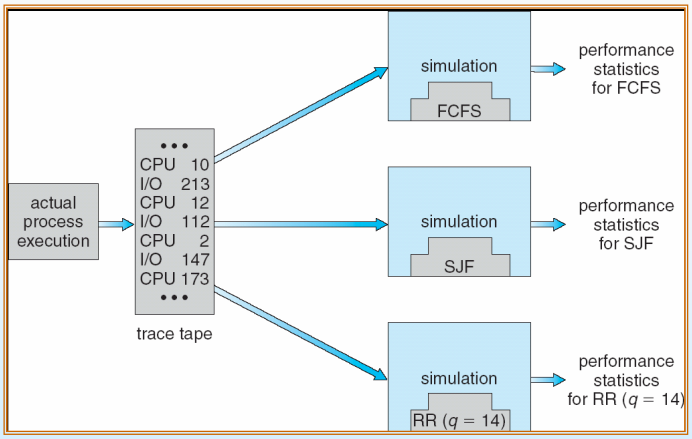
- O Windows utiliza um algoritmo preemptivo baseado na prioridade
- Existem 32 diferentes níveis de prioridade
  - Prioridade variável: 1 a 16
  - Tempo Real: 16 a 31
- As prioridades estão também em 6 classes que podem ser posicionadas através da API Win32.
- Dentro de cada classe existem 7 níveis de prioridade relativa

Informação prática para o caso do Windows. Perceber a prioridade relativa ou dinâmica que é ajustada em função da natureza do processo e da transição de estado!

Avaliação de Algoritmos

- Como se escolhe um algoritmo de CPU para um Sistema Operativo
- 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
  - **O modelo determinístico não pode ser utilizado**
  - 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**

Simulação Baseada em Logs



- Uma outra possibilidade consiste e correr os algoritmos de scheduling em simuladores
- Alimentados por *logs* de situações reais de carga de sistemas ou geradores de carga aleatória
- Testar diferentes algoritmos de scheduling em simulação
  - Aulas práticas

Métodos de simulação: são usados quando é necessário um “fine tunning” do SO para um **determinado padrão de utilização.**

Fim do Scheduling