ESCALONAMENTO DO PROCESSADOR

- Conceito de escalonamento
- Níveis de escalonamento
- Algoritmos de escalonamento
- Avaliação dos algoritmos

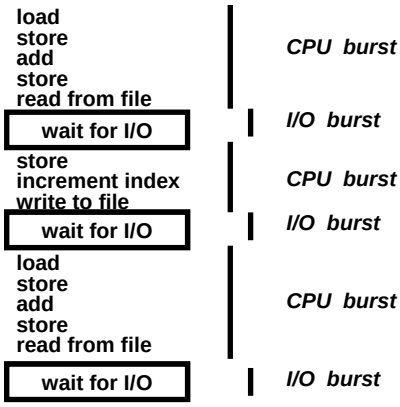
Conceitos básicos

Escalonamento do processador
- **estratégia de atribuição do *CPU* aos processos**

O escalonamento do processador é a base dos sistemas com multiprogramação.

A execução de um processo consiste em geral de
- **um "ciclo" de execução no *CPU* (*CPU burst*), seguido de**
- **uma espera por uma operação de *I/O* (*I/O burst*)**

Conceitos básicos



Conceitos básicos

A escolha do algoritmo de escalonamento depende do tipo de distribuição dos *bursts.*

Distribuição típica dos *CPU bursts*:
- **elevado nº de *bursts* de curta duração**
- **baixo nº de *bursts* de longa duração**

Programa
- ***CPU-bound***
  - » **passa a maior parte do tempo a usar o *CPU***
  - » **pode ter alguns *CPU bursts* muito longos**
- ***I/O bound***
  - » **passa mais tempo a fazer *I/O* do que computação**
  - » **tem, tipicamente, muitos *CPU bursts* curtos**

Níveis de escalonamento do *CPU*

Escalonamento de longo prazo
- **Determina que processos são admitidos para execução no sistema.**

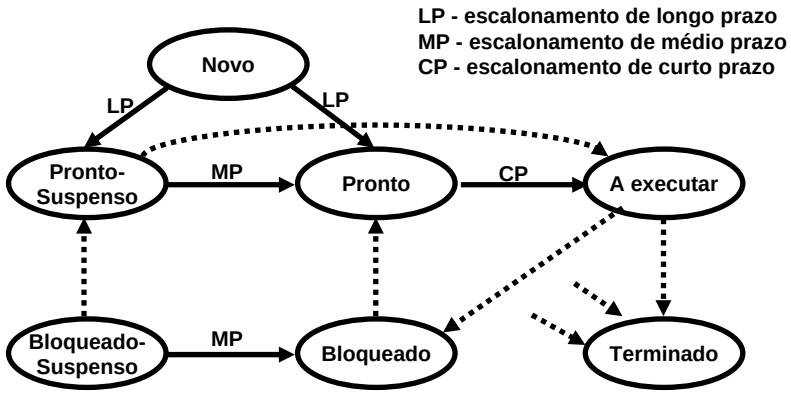
Escalonamento de curto prazo
- **Determina qual o processo a ser executado proveniente da fila de processos prontos.**

Escalonamento de médio prazo
- **Determina que processos são carregados, total ou parcialmente, em memória principal, depois de terem estado suspensos.**
- **Está ligado à função de *swapping.***

Sistemas Operativos Escalonamento do Processador

Níveis de escalonamento do *CPU*

Escalonamento e transições de estado dos processos



MIEIC
Faculdade de Engenharia da Universidade do Porto

Escalonamento de longo prazo e de médio prazo

Escalonamento de longo prazo

- Intervem na criação de novos processos.
- A decisão é, geralmente, apenas função de
  - » os recursos necessários e disponíveis
  - » o nº máximo de processos admissíveis
- Determina o grau de multiprogramação.
  - » grau de multiprogramação = nº de processos em memória

Escalonamento de médio prazo

- Intervem por ocasião da escassez de recursos
- Pode ser executado com intervalos de alguns segundos a minutos.

Escalonamento de curto prazo

Escalonamento de curto prazo

- As decisões relativas ao escalonamento podem ter lugar quando um processo
  - » 1 - comuta de "a executar" → "bloqueado"
  - » 2 - comuta de "a executar" → "pronto"
  - » 3 - comuta de "bloqueado" → "pronto"
  - » 4 - termina
- Em geral, é invocado com intervalos muito curtos. (algumas centenas de milisegundos).
- Deve ser o mais rápido e eficiente possível.
- Pode ser
  - » preemptivo - o processo pode ser forçado a ceder o *CPU*
  - » não preemptivo - o processo executa até bloquear ou ceder a vez voluntariamente

Despacho

O módulo de despacho (*dispatcher*)
dá o controlo do *CPU* ao processo seleccionado
pelo módulo de escalonamento de curto prazo.

Isto envolve:

- **comutação de contexto**
- **comutação p/ modo utilizador**
- **"saltar" p/ o endereço adequado do programa do utilizador**

Algoritmos de escalonamento

- ***First-Come First-Served* (*FCFS*)**
- ***Shortest Job First* (*SJF*) *e***
  ***Shortest Remaining Time First* (*SRTF*)**
- ***Priority Schedulling* (*PS*)**
- ***Round-Robin* (*RR*)**
- ***Multilevel Queue* (*MLQ*)**
- ***Multilevel Feedback Queue* (*MLFQ*)**

Critérios de avaliação dos algoritmos de escalonamento

Utilização do processador
- percentagem de tempo em que o processador está ocupado

Taxa de saída / *eficiência* (*throughput*)
- nº de processos completados por unidade de tempo
  - » importante em sistemas *batch*

Tempo de resposta
- tempo que o sistema demora a começar a responder
  - » importante em sistemas interactivos

Tempo de permanência (*turnaround time*)
- intervalo de tempo desde que o processo é admitido até que é completado pelo sistema

Tempo de espera
- tempo total que o processo fica à espera na fila de proc.s prontos de ser seleccionado p/ execução

Optimização dos algoritmos

Maximizar
- utilização do processador
- *throughput*

Minimizar
- tempo de permanência
- tempo de espera
- tempo de resposta

First-Come First-Served

- **Os processos são escalonados por ordem de chegada (fila *FIFO*).**
- **Não-preemptivo.**
- **Vantagens:**
  - **Fácil de implementar.**
  - **Simples e rápido na decisão.**
  - **Não há possibilidade de inanição *(starvation)* - todos os processos têm oportunidade de executar.**
- **Desvantagens**
  - **O tempo médio de espera é frequentemente longo.**
  - **Pode conduzir a baixa utilização do *CPU* e dos dispositivos de *I/O.***
- **Inadequado p/ sistemas *time-sharing*.**

Exemplo (*FCFS*)

Processo	*CPU burst time* (ms)
P1	24
P2	3
P3	3

tempos do 1º *burst cycle* de cada processo

Qual o tempo de espera médio quando a ordem de chegada é
a) P1 - P2 -P3 ?
b) P2 - P3 - P1 ?

Todos os processos chegam em t=0

Ordem de chegada: P1 - P2 -P3

Processo	Tempo de espera
P1	0
P2	24
P3	27

média = 17

Ordem de chegada: P2 - P3 -P1

Processo	Tempo de espera
P1	6
P2	0
P3	3

média = 3

First-Come First-Served

Exemplo:
(situação dinâmica, 1 processo *CPU-bound* e muitos processos *I/O-bound*)

- **Em certa altura, um processo *CPU-bound* toma conta do processador.**
- **Durante este tempo, os processos *I/O-bound* terminam a *I/O* e vão p/ a lista dos processos prontos.**
- **Enquanto isto, os dispositivos de *I/O* ficam inactivos.**
- **Quando o processo *CPU-bound* liberta o processador e fica à espera de *I/O* os processos *I/O-bound* usam o processador durante um curto intervalo e voltam p/ as filas de espera de *I/O*.**
- **Neste momento, estão todos os processos à espera de *I/O* e o processador inactivo.**
- **A baixa utilização da *CPU* poderia ser evitada se não se usasse *FCFS* (deixando que os processos mais curtos corressem primeiro)**

Efeito do *FCFS* sobre os processos:

- » **penaliza os processos curtos**
- » **penaliza os processos *I/O* bound**

Shortest-Job-First

- **Cada processo deverá ter associada a duração do próximo *CPU-burst.* (possível ? …)**
- **É seleccionado o processo com o menor próximo *CPU-burst*.**
- **Dois esquemas:**
  - **Não preemptivo**
    - » **uma vez atribuído o *CPU* a um processo não lhe pode ser retirado até que ele complete o *CPU-burst***
  - **Preemptivo (*Shortest Remaining Time First - SRTF*)**
    - » **se chegar um novo processo com uma duração do *CPU-burst* menor do que o tempo que resta ao processo em execução faz-se a preempção**

Shortest-Job-First

O algoritmo *SJF* é:

- **Óptimo**
  - » **Resulta num tempo médio de espera mínimo para um dado conjunto de processos.**
- **Difícil de implementar**
  - » **Como determinar a duração do próximo *CPU-burst* ?**

Estimação da duração do próximo *CPU-burst*

- **Fazer a média exponencial de tempos medidos anteriormente**

$$T_{n+1} = \alpha\, t_n + (1-\alpha)\, T_n$$

$0 \le \alpha \le 1$

t_n = duração do *burst* n

T_n = tempo estimado do *burst* n

Shortest-Job-First

Notar que nesta estimativa
todos os valores são considerados
mas, os mais distantes no tempo têm menor peso.

$$T_{n+1} = \alpha\, t_n + (1-\alpha)\, \alpha\, t_{n-1} + (1-\alpha)^2\, \alpha\, t_{n-2} + \ldots$$
$$\ldots + (1-\alpha)^i\, \alpha\, t_{n-i} + \ldots + (1-\alpha)^n\, T_0$$

Exemplo:

$\alpha = 0.8 \Rightarrow$

$$T_{n+1} = 0.8\, t_n + 0.16\, t_{n-1} + 0.032\, t_{n-2} + 0.0064\, t_{n-3} + \ldots$$

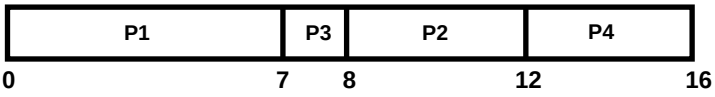
α **elevado** $\Rightarrow$ **dar muito peso às observações mais recentes**

α **baixo** $\Rightarrow$ **a média tem em conta um maior nº de observações**

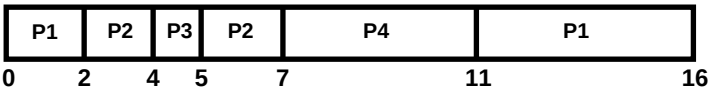
Exemplo (*SJF e SRTF*)

Processo	Hora de chegada	*CPU burst time* (ms)
P1	0	7
P2	2	4
P3	4	1
P4	5	4

***SJF* (não preemptivo)** Tempo de espera médio = (0+6+3+7) / 4 = 4 ms



***SRTF* (preemptivo à cheg.)** Tempo de espera médio = (9+1+0+2) / 4 = 3 ms



SJF e *SRTF*

Características gerais de *SJF* e *SRTF*:

- **Penaliza os processos que fazem uso intensivo do *CPU***
- **Possibilidade de inanição (*starvation*) de alguns processos**
- ***Overhead* elevado**

O algoritmo *SJF* poderia ser usado no escalonamento de longo prazo.

A estimativa do tempo de execução de um programa deveria ser fornecida pelo utilizador.

Esta estimativa deve ser o mais correcta possível.

- **estimativa de valor baixo ⇒**
  resposta mais rápida
- **mas… um valor demasiado baixo ⇒**
  exceder o limite de tempo ⇒
  submeter o programa de novo p/execução !

Priority schedulling

- **A cada processo é associada uma prioridade.**
  - » **prioridade = nº inteiro**
  - » **a gama de valores e o seu significado depende do S.O.**
- **O processador é atribuído ao processo com maior prioridade.**
- **Dois esquemas:**
  - **Não preemptivo**
    - » **um novo processo é colocado na fila de processos prontos e aguarda que o processo actual liberte o *CPU***
  - **Preemptivo**
    - » **sempre que um processo chega à fila de processos prontos a s/prioridade é comparada c/ a do processo em execução e, se for maior, o *CPU* é atribuído ao novo processo.**

Priority schedulling

Problema principal

- **inanição** (se as prioridades forem estáticas)

Solução

- **aumentar gradualmente a prioridade dos processos que estão à espera de execução à medida que o tempo passa** (envelhecimento)

Definição das prioridades

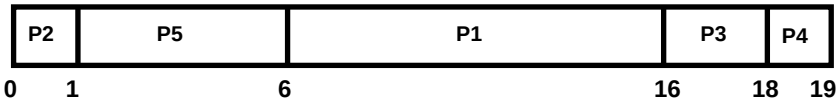
- **definidas internamente, atentendo a**
  - » **limites de tempo**
  - » **necessidades de memória**
  - » **nº de ficheiros abertos**
  - » **quociente entre *I/O burst* médio e *CPU burst* médio**
- **definidas externamente, atentendo a**
  - » **importância do processo**
  - » **importância do utilizador**
  - » **...**

Exemplo (*PS*)

Os processos P1..P5 chegaram em t=0 por esta ordem →

valor baixo = prioridade elevada

Processo	Prioridade	*CPU burst time* (ms)
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5



Tempo de espera médio = 8.2 ms

Round Robin

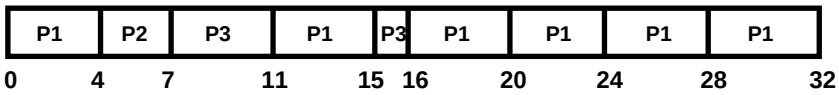
- Atribui-se a cada processo uma fatia de tempo (*quantum*) para executar.
- Um processo permanece no estado de execução até efectuar uma operação de I/O, ou ser interrompido por um temporizador que periodicamente (no fim da fatia de tempo atribuída ao processo) interrompe o processo que estiver em execução, ou terminar
- A lista de processos prontos é uma fila do tipo *FIFO*. Sempre que há uma mudança de contexto o processo que deixa o processador vai p/ o fim da lista.

Exemplo (*RR*)

Processo	*CPU burst time* (ms)
P1	24
P2	3
P3	5

Os processos P1..P5 chegaram em t=0 por esta ordem →

quantum = 4 ms



0 4 7 11 15 16 20 24 28 32

Tempo de espera médio = __(?) ms

Round Robin

- Efeito da fatia de tempo / *quantum* (q)
  - fatia grande
    ≡ *FCFS*
  - fatia pequena
    ⇒ muitas mudanças de contexto;
    perda de eficiência
- As fatias devem ser pequenas mas bastante maiores do que o tempo gasto na mudança de contexto.
- Nenhum processo espera mais do que $(n-1)\cdot q$ unidades de tempo pela sua fatia de tempo seguinte (n = nº de processos).

Round Robin

Características gerais do *RR*:

- Não há perigo de inanição
- Favorece os processos *CPU-bound*
  - Um processo *I/O-bound* usará frequentemente menos do que 1 *quantum*, ficando bloqueado à espera das operações de *I/O*
  - Um processo *CPU-bound* esgota o seu *quantum* passando para a fila de processos prontos e passando à frente dos processos bloqueados

Round Robin virtual

Solução para o problema anterior
(favorecimento dos processos *CPU-bound*)

- Quando um processo bloqueado fica com a sua operação de *I/O* completa, em vez de ir para a fila de processos prontos, vai para uma fila auxiliar que tem preferência sobre a fila de processos prontos
- Quando um processo da fila auxiliar é despachado irá correr apenas um tempo que é igual ao *quantum* menos o tempo de processador que utilizou imediatamente antes de ficar bloqueado

Multilevel Queue

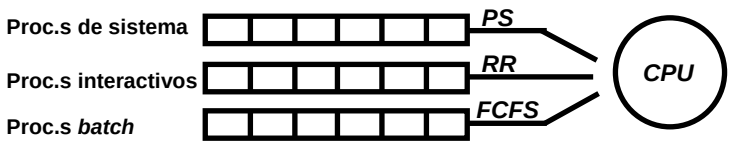
- **Sistema baseado em prioridades mas com várias filas de processos prontos.**
- **Cada fila tem o seu algoritmo de escalonamento.**
- **As filas são ordenadas por ordem decrescente de prioridade.**
- **Executa-se um processo de uma fila apenas quando não há processos prontos nas filas de maior prioridade.**
- **É necessário fazer um escalonamento entre filas.**

Multilevel Queue

Escalonamento entre filas:

- **Com prioridade fixa**
  - » **Servir das filas de mais alta prioridade para as de mais baixa prioridade.**
  - » **Possibilidade de inanição.**
- **Fatia de tempo**
  - » **Cada fila recebe uma fatia de tempo que distribui pelos seus processos.**

ex: no caso de 2 filas
atribuir 80% à de maior prioridade e 20% à outra



Multilevel Feedback Queue

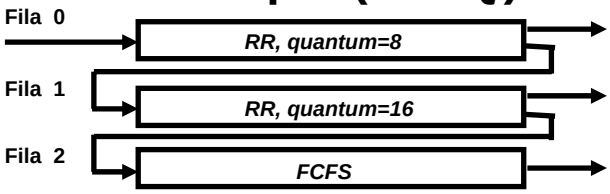
- **Sistema *Multilevel Queue* com regras para movimentar os processos entre as várias filas.**
- **Os processos mudam de fila de acordo com o seu comportamento anterior:**
  - **Os processos são admitidos na fila de maior prioridade**
  - **Um processo que usa demasiado tempo de *CPU* é despromovido p/ uma fila de prioridade mais baixa.**
  - **Um processo que esteja há muito tempo à espera numa fila de baixa prioridade vai sendo deslocado p/ filas de maior prioridade.**
    - » **Esta operação, dita de envelhecimento do processo, pode impedir a inanição.**

Multilevel Feedback Queue

Parâmetros de um *MLFQ scheduller:*

- **Nº de filas**
- **Algoritmo de escalonamento para cada fila**
  - » **Quase sempre *Round Robin***
- **Método usado p/ determinar quando se deve promover um processo**
  - » **Quando ele bloqueia antes de terminar a sua fatia de tempo**
- **Método usado p/ determinar quando se deve despromover um processo**
  - » **Quando usa completamente a sua fatia de tempo**
- **Método usado p/ determinar em que fila entra o processo pela 1ª vez**
  - » **Benefício da dúvida: começar por uma de alta prioridade**
- **Política de "envelhecimento"**
  - » **Mover p/ filas de maior prioridade quando o tempo de espera começa a ser elevado**

Exemplo (*MLFQ*)



- Primeiro, executar todos os processos da fila 0.
  Quando ela estiver vazia, passar aos da fila 1, …
- Processo da fila 1 a executar, mas chega processo p/ a fila 0 ⇒
  preempção do processo da fila 1
- Um processo da fila 0 recebe um *quantum* de 8 ms.
  Se não acabar nesse tempo, vai para a fila 1.
- Se a fila 0 estiver vazia
  é dado um *quantum* de 16 ms ao 1º processo da fila 1.
  Se ele não acabar nesse período, passa para a fila 2.
- Processos com *CPU burst* < 8ms são servidos rapidamente.
- Os processos longos acabam por cair na fila 2 *(FCFS).*

Escalonamento em sistemas multiprocessador

- Escalonamento mais complexo
  do que em sistemas uniprocessador.
- Sistemas
  - Homogéneos (processadores idênticos)
    - » Facilitam a partilha de carga .
    - » Em geral, usa-se uma fila única p/ todos os processadores.
  - Heterogéneos (processadores diferentes)
- Escalonamento
  - Mestre/Escravo (*Master/Slave*)
  - Auto-escalonamento

Escalonamento em sistemas multiprocessador

- **Escalonamento Mestre/Escravo**
  - **O processador-mestre "corre o S.O." e faz o despacho das tarefas p/ os processadores-escravo.**
  - **Os "escravos" só correm programas do utilizador.**
- **Auto-escalonamento**
  - **Cada processador manipula a lista de proc.s prontos.**
  - **A manipulação da lista torna-se complicada.**
    - » **Assegurar que não há 2 processadores**
      - **a seleccionar o mesmo processo**
      - **a actualizar a lista simultâneamente**

Escalonamento em Sistemas de Tempo-real

- **Sistemas *Hard-Real-Time***
  - **Têm de completar as tarefas dentro de um intervalo de tempo garantido**
    - » **O escalonador tem de saber o tempo máximo que demora a executar cada função do S.O. .**
      - **Garantia impossível em sistemas com mem. virtual.**
    - » ***Hardware* dedicado, muito específico.**
- **Sistemas *Soft-Real-Time***
  - **Apenas requerem que certos processos críticos tenham prioridade sobre os outros**
    - » **Escalonamento c/ prioridades, sendo atribuída prioridade elevada aos proc.s críticos.**
    - » **A latência de despacho deve ser curta.**
    - » **Deve existir possibilidade de preempção.**

Performance dos Algoritmos de Escalonamento

- Como avaliar os algoritmos de escalonamento ?
  - ⇒ Definir importância relativa dos critérios de avaliação
    (utilização do *CPU*, *throughput*, tempo de resposta, ...)
  - Exemplo: Maximizar a utilização do *CPU* desde que o tempo de resposta máximo seja *x*.
- Como avaliar os diversos algoritmos sobre as restrições definidas ?
  - Avaliação analítica
  - Modelação de filas
  - Simulação
  - Implementação real

Escalonamento no UNIX (SVR3 e 4.3BSD)

- Escalonamento do tipo *Multilevel Feedback Queue* com *RoundRobin* em cada fila .
- As prioridades em modo núcleo são fixas e são sempre superiores às prioridades em modo utilizador.

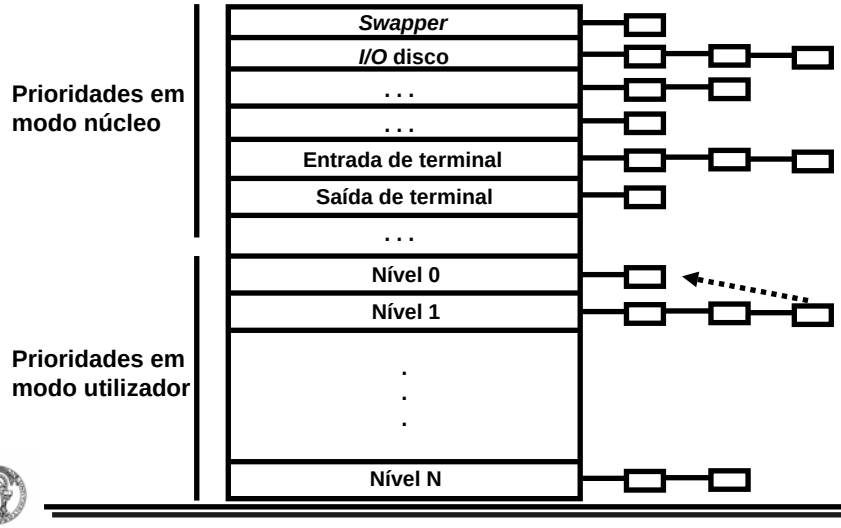
Valores da prioridade

- em modo núcleo - valores negativos
- em modo utilizador - valores não negativos
  valor baixo ⇒ prioridade elevada

Prioridades em modo utilizador

- São actualizadas periodicamente *(de 1 em 1 seg. ?)*
- A fórmula de actualização visa diminuir a prioridade dos processos que utilizaram recentemente o processador durante mais tempo.

Escalonamento no UNIX (SVR3 e 4.3BSD)



Escalonamento no UNIX (SVR3 e 4.3BSD)

Actualização de prioridade em modo utilizador, feita pelo algoritmo de escalonamento:

$$P_j(i) = Pbase_j + CPU_j(i-1)/2 + nice_j$$

$$CPU_j(i) = (U_j(i) + CPU_j(i-1)) / 2$$

$P_j(i)$ = Prioridade do processo j no início do intervalo i
$Pbase_j$ = Prioridade base do processo j
$U_j(i)$ = Utilização do *CPU* pelo processo j, no intervalo i
$CPU_j(i)$ = Média exponencial da utilização do *CPU* pelo processo j no intervalo i
$nice_j$ = Factor de ajuste controlável pelo utilizador (comandos *nice* e *renice* – permitem alterar a prioridade dos processos)

(NOTA: versões mais recentes de UNIX usam outros algoritmos)

Escalonamento no Linux

- **Algoritmo de escalonamento preemptivo, relativamente simples, baseado em prioridades**
  - **A preempção só pode ocorrer quando o processo estiver a executar em modo utilizador**
    - » **se o seu *quantum* terminar**
    - » **se chegar à fila de processos prontos um processo com prioridade superior à do processo que está a executar**
  - **A prioridade de um processo é dinâmica (o escalonador vai tomando nota da actividade dos processos e ajusta as prioridades periodicamente)**
    - » **os processos *I/O-bound* são favorecidos relativamente aos *CPU-bound***
- **Tipos de processos:**
  - **Normal**
  - **Tempo-Real (*Real-Time*)**
    - » **executam sempre antes dos Normais**
    - » **o Linux é *soft real-time***

Escalonamento no Linux

Operação básica do escalonador:

- **percorrer a fila de processos prontos procurando o melhor processo para ser executado**
- **para cada processo da fila calcular o valor da sua *goodness* (bondade)**
- **se atingir o fim da fila sem encontrar um processo com *goodness* <> 0, "envelhecer" os processos e percorrer de novo a lista**

Algoritmo de escalonamento:

- **O tempo de *CPU* é dividido em "épocas".**
- **Numa época, cada processo tem direito a um *quantum* cuja duração é calculada no início da época.**
- **Em geral, diferentes processos têm diferentes *quanta.***
- **Quando um processo esgota o seu *quantum* sofre preempção e é substituído por outro processo executável.**
- **Um processo pode ser seleccionado várias vezes, na mesma época, desde que não tenha esgotado o *quantum* inicialmente atribuído.**
- **A época termina quando todos os processos executáveis tiverem esgotado o seu *quantum*.**
- **Nessa altura, o escalonador volta a calcular os *quanta* e começa uma nova época.**

Escalonamento no Linux

Algoritmo de escalonamento (cont.)

- Cada processo tem um *base time quantum* (= prioridade-base do processo)
  - » *quantum* que lhe é atribuído sempre que ele esgotar o *quantum* anterior
  - » os utilizadores podem alterá-lo através das chamadas `nice()` e `setpriority()`
  - » valor típico = 210 ms (20 ticks do clock)
- Ao seleccionar um processo para executar, o escalonador tem em conta a prioridade de cada processo
- Os processos *Real-Time* têm sempre prioridade sobre os processos Normais
- Há 2 tipos de prioridade:
  - » prioridade estática
    - a prioridade dos processos *Real-Time*
    - varia entre 1 e 99
    - nunca é alterada pelo escalonador
  - » prioridade dinâmica
    - a prioridade dos processos Normais
    - = *base time quantum* +
      + nº de *ticks* de *CPU* que faltam para um processo terminar o seu *quantum* na época corrente

NOTA: quando se cria um novo processo, o nº de *ticks* que faltam para o proc.-pai são divididos em 2 metades, uma para o proc.-pai, outra para o proc.-filho.

Escalonamento no Linux

- Cada processo tem associada uma classe de escalonamento
  - » SCHED_FIFO (*First-In-First-Out*)
    - aplica-se a processos *Real-Time*
    - se não houver processos *Real-Time* com prioridade superior o processo continua a executar enquanto quiser mesmo que haja processos c/ prioridade igual à sua na fila de proc.s prontos
  - » SCHED_RR (*Round-Robin*)
    - aplica-se a processos *Real-Time*
  - » SCHED_OTHER
    - aplica-se a processos *time-shared* convencionais
  - » SCHED_YIELD
    - aplica-se a processos que cederam voluntariamente o processador
- Um processo pode usar a chamada ao sistema `sched_yield()` para libertar voluntariamente o processador

Escalonamento no Linux

Cálculo da *goodness* de um processo:

- **etapa fundamental do algoritmo de escalonamento**
- **indica quão desejável é que o processo seja seleccionado p/executar**

- ***goodness* = –1000**
  - » **o processo nunca deve ser seleccionado**
  - » **este valor só é retornado quando a fila de proc.s prontos só contém init_task()**
- ***goodness* = 0**
  - » **o processo gastou o seu *quantum***
- **0 < *goodness* < 1000**
  - » **processo convencional que não esgotou o seu *quantum***
- ***goodness* >= 1000**
  - » **processo *Real-Time***

Escalonamento no Linux

Função goodness:

```
int goodness(struct task_struct *p, struct task_struct *prev, ....)
{  ...

   if (p->policy != SCHED_OTHER)
        return 1000 + p->rt_priority; /* REAL-TIME PROCESS */
   /* NORMAL PROCESSES */
   if (p->num_ticks_left == 0)
        return 0;
   if (p->mm == prev->mm) /*slight advantage to the current thread */
        return  p->priority + p->num_ticks_left + 1;
   return p->priority + p->num_ticks_left;
   ...
}
```

Escalonamento no Linux

Execução do escalonador:

- **Invocação directa**
  o escalonador é invocado directamente quando
  - » **o processo em execução tem de ser bloqueado porque necessita de recursos não disponíveis**
- **Invocação *lenta* (*lazy*)**
  ocorre quando
  - » **o processo em execução esgotou o seu *quantum***
  - » **um processo foi acrescentado à fila de processos prontos e a sua prioridade é superior à do processo em execução**
  - » **um processo invoca `shed_yield()`**

  Nestes casos é activada uma *flag*, e a execução do escalonador tem lugar posteriormente.

Escalonamento no Windows 2000

- **Escalonamento preemptivo, com múltiplos níveis de prioridade, com *Round-Robin* em cada nível.**
- **Classes de prioridade:**
  - **Classe Variável / Dinâmica (prioridade 1..15)**
    - » **Um *thread* começa com um valor inicial de prioridade, a qual pode aumentar ou diminuir durante a execução.**
      - **aumentar - se bloqueou à espera de *I/O***
      - **diminuir - se usou toda a fatia de tempo**
    - » **Prioridade inicial - determinada a partir das prioridades-base do processo e do *thread* (a somar à prioridade do processo).**
    - » **prioridade inicial ≤ prioridade dinâmica da *thread* ≤ 15**
    - » ***RR* em cada nível de prioridade, mas um processo pode migrar p/ outros níveis (excepto os da classe *Real-Time*).**
  - **Classe *Real-Time* (prioridade 16..31)**
    - » **As prioridades dos *threads* não são ajustadas automaticamente.**

Escalonamento no Windows 2000

- Para o escalonador do Windows 2000 o que é escalonado são *threads*, não processos
- Os processos recebem uma certa classe de prioridade ao serem criados :
  - Idle, Below Normal, Normal, Above Normal, High, Realtime
- Os *threads* têm uma prioridade relativa dentro da classe
  - Idle, Lowest, Below Normal, Normal, Above Normal, Highest, Time Critical
- Existem 32 filas (*FIFO*) de *threads* prontos a executar
- Quando um *thread* fica pronto
  - corre imediatamente, se o processador estiver disponível, ou
  - é inserido no final da fila correspondente à s/prioridade actual
- Os *threads* de cada fila são executados em *round-robin*

Escalonamento no Windows 2000

- Preempção
  - Se um *thread* com uma prioridade superior à do *thread* que está a executar fica pronto
    - » o *thread* de menor prioridade sofre preempção
    - » este *thread* vai para a "cabeça" da sua fila de processos prontos
  - Estritamente guiada por eventos
    - » não espera pelo próximo *clock tick*
    - » não garante um período de execução, antes da preempção
- Comutação voluntária
  - Quando o *thread* em execução cede o *CPU* porque
    - » bloqueou
    - » terminou
    - » houve um abaixamento explícito de prioridade

Escalonamento no Windows 2000

- Se o *thread* vê o seu *quantum* expirar
  - a prioridade é decrementada, a não ser já seja igual à prioridade-base do *thread*
  - o *thread* vai para o fim da fila correspondente à sua nova prioridade
  - pode continuar a executar se não houver *threads* com prioridade igual ou superior (volta a ter um novo *quantum*)
- *Quantum*-padrão
  - 2 *clock ticks*
  - se um processo c/ prioridade Normal possuir a janela de *foreground* os seus *threads* podem receber um *quantum* maior
- Inanição
  - O *Balance Set Manager* é um *thread* com nível de prioridade 16, que executa de 1 em 1 segundo e
  - procura *threads* que estejam prontos há 4 segundos ou mais
  - aumenta a prioridade de até 10 *threads* em cada passagem
  - Não se aplica aos *threads* da classe *real-time*
    - » Isto significa que o escalonamento destes *threads* é "previsível"
    - » No entanto, não significa que haja garantia de uma certa latência

Escalonamento no Windows 2000

- Multiprocessamento
  - Por defeito, os *threads* podem correr em qualquer processador disponível
- *Soft affinity*
  - Cada *thread* tem um "processador ideal"
  - Quando um *thread* fica pronto a correr
    - » se o "processador ideal" estiver disponível, executa nesse processador
    - » senão, escolhe outro processador de acordo com regras estabelecidas
- *Hard affinity*
  - Restringe um *thread* a um subconjunto dos *CPUs* disponíveis
  - Raramente adequada.