# ESCALONAMENTO DO PROCESSADOR

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





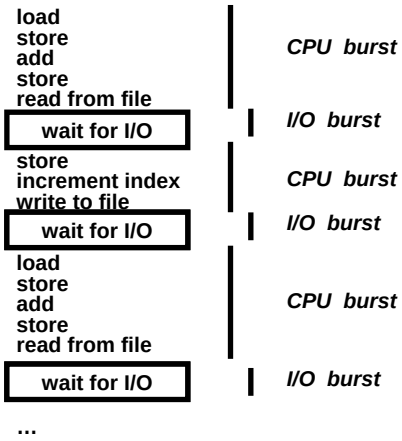
# Conceitos básicos

**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* em processos interactivos**

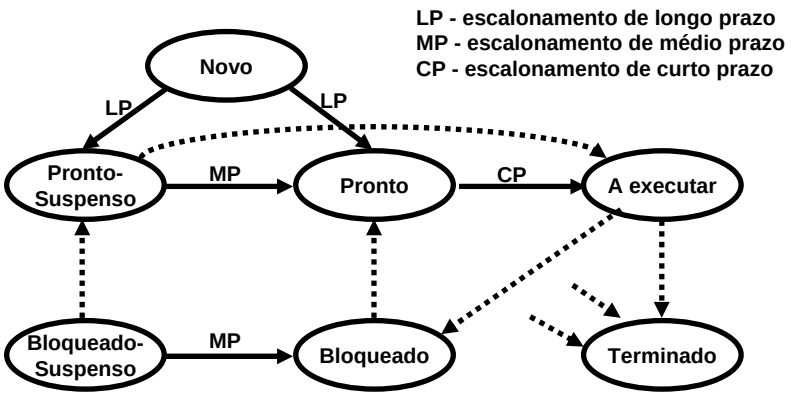
- **elevado nº de *bursts* de curta duração**
- **baixo nº de *bursts* de longa duração**

**Processo**

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

# Níveis de escalonamento do *CPU*

**Escalonamento e transições de estado dos processos**








# 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 "novo" → "pronto"
  - » 2 - comuta de “a executar” → “bloqueado”
  - » 3 - comuta de “a executar” → “pronto”
  - » 4 - comuta de “bloqueado” → “pronto”
  - » 5 - 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 (chamada `sched_yield()`, em Linux)





# Despacho

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

Isto envolve:

- comutação de contexto
- comutação p/ modo utilizador
- “saltar” p/ o endereço adequado do programa do utilizador

# Algoritmos de escalonamento em sistemas uniprocessador

- ***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**

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

$$0 \leq \alpha \leq 1$$

$t_n$ = 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:**

$\alpha = 0.8 \Rightarrow$

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

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

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





# *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 $\Rightarrow$**
  **resposta mais rápida**
- **mas… um valor demasiado baixo $\Rightarrow$**
  **exceder o limite de tempo $\Rightarrow$**
  **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 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**

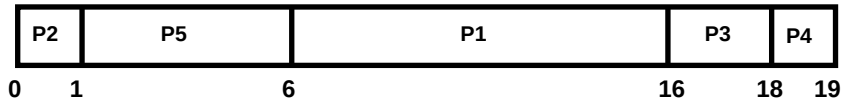
- **definidas internamente, atentendo a**
  - » **duração dos *I/O bursts* e/ou *CPU bursts***
  - » **recursos detidos pelo processo**
  - » **...**
- **definidas externamente, atentendo a**
  - » **importância do processo**
  - » **importância do utilizador**
  - » **...**

# 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*

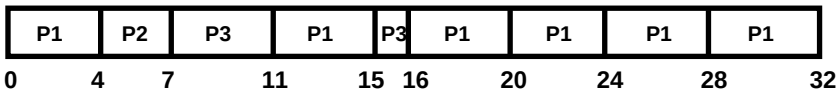
- Atribui-se a cada processo uma fatia de tempo (*quantum*) para executar.
- Um processo permanece no estado de execução até
  bloquear, ou
  ser interrompido por um temporizador que periodicamente (no fim da fatia de tempo atribuída ao processo) interrompe o processo que estiver em execução,
  ou terminar
- 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)*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* e vai logo para a fila de processos prontos passando à frente dos processos bloqueados





# *Round Robin* virtual

## Solução para o problema anterior
**(problema de 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.**
- **É necessário fazer um escalonamento entre filas.**



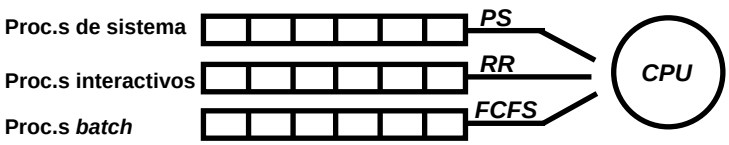


# *Multilevel Queue*

**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**



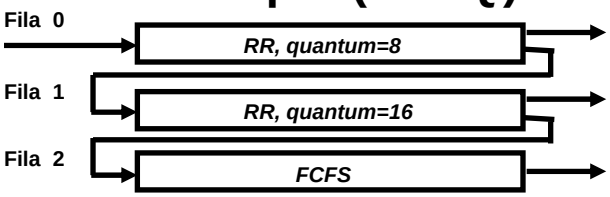
# *Multilevel Feedback Queue*

- **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.**





# 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 bursts* < 8ms são servidos rapidamente.**
- **Processos com *CPU bursts* longos acabam por cair na fila 2 *(FCFS).***

# Escalonamento em sistemas multiprocessador

- Escalonamento mais complexo do que em sistemas uniprocessador.
- Sistemas
  - Homogéneos (processadores idênticos)
  - 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 simultaneamente

# 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 em S.O.'s concretos

**UNIX**

**•Baseia-se num sistema c/prioridades dinâmicas que procura priveligiar os processos *I/O-bound***

**•A ideia básica é:**

- **tornar menos prioritários os processos que executaram mais recentemente**
- **mas, gradualmente, restabelecer a prioridade destes processos, para não serem penalizados para sempre**

**•Problema:**

- **má escalabilidade com o número de processos a correr no Sistema, devido ao cálculo das prioridades de todos os processos ser feita com intervalos fixos**





# Escalonamento em S.O.'s concretos

**LINUX**

**•Diferentes algoritmos de escalonamento têm sido propostos ao longo dos tempos.**

- **Linux 2.2 & 2.4 scheduler (2002)**
  - **o tempo é dividido em *epochs***
  - **durante cada época todos os processos têm a possibilidade de executar durante um *quantum* variável**
  - **caso não usem todo o seu *quantum*, o *quantum* na próxima *epoch* aumenta e a prioridade também, pois esta é proporcional à parte do quantum que ficou por utilizer**
  - **Isto beneficia os processos *I/O-bound***
- **Linux 2.6.23 - Completely Fair Scheduler (Molnar, 2007)**
- **Brain Fuck Scheduler (Kolivas, 2009)**

# 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

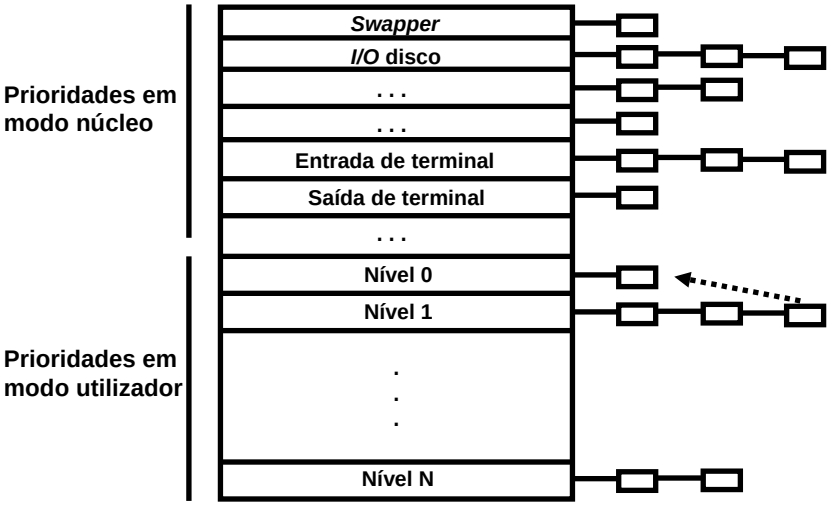
Prioridades em modo utilizador

- 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

- A partir da versão 2.6.23 do núcleo o Linux passou a utilizar o algoritmo de escalonamento designado *Completely Fair Scheduler (CFS)* para o escalonamento dos processos interactivos dos utilizadores.
  - No *CFS* cada processo está ordenado pelo tempo de *CPU* que lhe foi atribuído (*vruntime*)
  - Os processos com menor *vruntime* são os primeiros a executar
  - De cada vez que o escalonador é executado,
    - » ao *vruntime* do processo em execução é somado o tempo que ele esteve em execução e
    - » se o *vruntime* obtido for superior ao menor *vruntime* dos outros processos, é-lhe retirado o *CPU*
- Quando um processo é criado
  - é-lhe atribuído o menor dos *vruntime's* dos processos em execução, dando-lhe oportunidade de executar de imediato.
    - » nota: no entanto, nunca lhe é atribuído o tempo igual a zero
- Os processos interactivos terão, em geral, *vruntime's* baixos, pelo que terão, naturalmente, prioridade sobre os outros.
- Em 2009, foi proposto o *BFS - Brain Fuck Scheduler !!!*
- http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

# Escalonamento no Linux

- O *CFS* não possui prioridades de "tempo real", isto é, processos que são sempre escalonados prioritariamente, independentemente do seu tempo de execução.
- Os processos que se enquadrariam neste tipo de prioridades são geridos por outro escalonador.
- De facto, a partir da versão 2.6.23 do núcleo passou a existir um metaescalonador que gere uma lista de escalonadores e que escolhe qual deles irá escolher o processo a executar.
- A política deste metaescalonador é a seguinte.
  - os vários escalonadores são mantidos numa lista ordenada por prioridade e
  - os processos dos escalonadores de menor prioridade só são escolhidos se não existirem processos activos nos escalonadores de maior prioridade.
- Como o escalonador dos processos de "tempo real" possui uma prioridade superior ao *CFS*, o primeiro tem sempre prioridade sobre o segundo.
- O escalonador dos processos de "tempo real" escolhe sempre os processos mais prioritários, independentemente do seu tempo de execução.





# Escalonamento no Linux

**ESCALONAMENTO EM MULTIPROCESSADORES**

- O Linux suporta a utilização de *multiprocessadores* (desde a v. 2.0) e, posteriormente, de *multicores*
- Para garantir a afinidade dos processos a determinados processadores, o escalonador mantém uma fila de processos para cada processador.
- Para evitar assimetrias de carga entre processadores é calculada uma média do nº de processos na fila de cada processador e, periodicamente, é executado um algoritmo de balanceamento de carga que, se detectar elevadas assimetrias, procede à migração de alguns processos da fila de processos dos processadores com maior carga para outros com menor carga.

# Escalonamento no Windows

- 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 níveis da classe *Real-Time* ).
  - Classe *Real-Time* (prioridade 16..31)
    - » As prioridades dos *threads* não são ajustadas automaticamente.





# Escalonamento no Windows

- Para o escalonador do Windows 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

- **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

- **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

- **Multiprocessamento**
  - **Por omissão, 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.**