Processos

Conceito de Processo

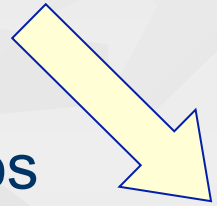
- Processo
  - Entidade activa, que corresponde a um **programa em execução**
  - Cada processo tem um **espaço de endereçamento próprio**
  - A gestão de processos é da responsabilidade do sistema operativo, que utiliza estruturas de dados (***process tables***) que descrevem o **contexto de execução** de cada processo
  - O próprio sistema operativo é também um conjunto de vários processos
- Programa
  - Sequência de instruções sem atividade própria – não confundir com processo

Multi-programação

- Num sistema multi-programado, mesmo que só exista um processador é possível vários processos estarem activos simultaneamente
- Contudo, em cada instante temporal, apenas um deles pode utilizar o processador
- A esta ilusão de vários processos correrem aparentemente em paralelo, dá-se o nome de **pseudo-paralelismo**

Multi-programação

- Não devem ser feitas assunções em relação à ordem de comutação do processador, devido a:
  - Existência de interrupções
  - Falta de recursos
  - Entrada de processos prioritários

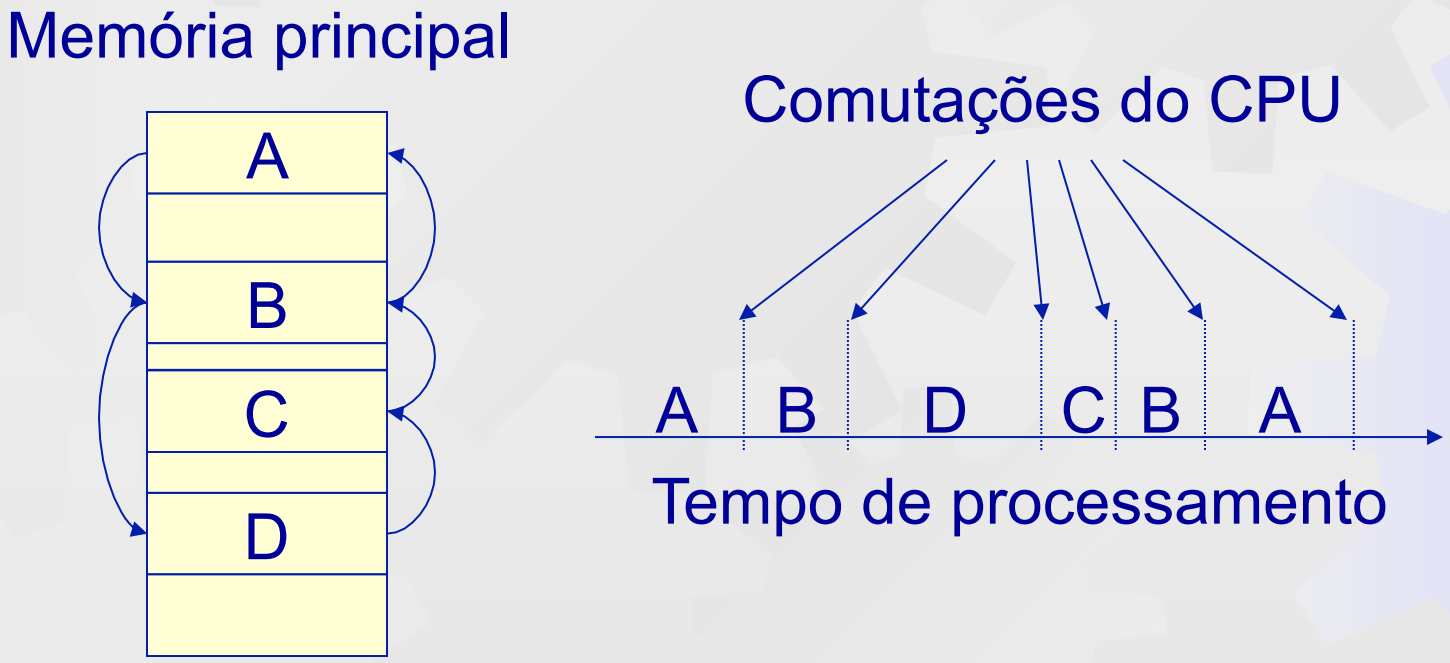


Não-determinista

- Depois de uma comutação do processador, o próximo processo a utilizá-lo é escolhido pelo **escalonador de processos** do SO

Multi-programação

- Exemplo: 4 processos a correr



Multi-programação

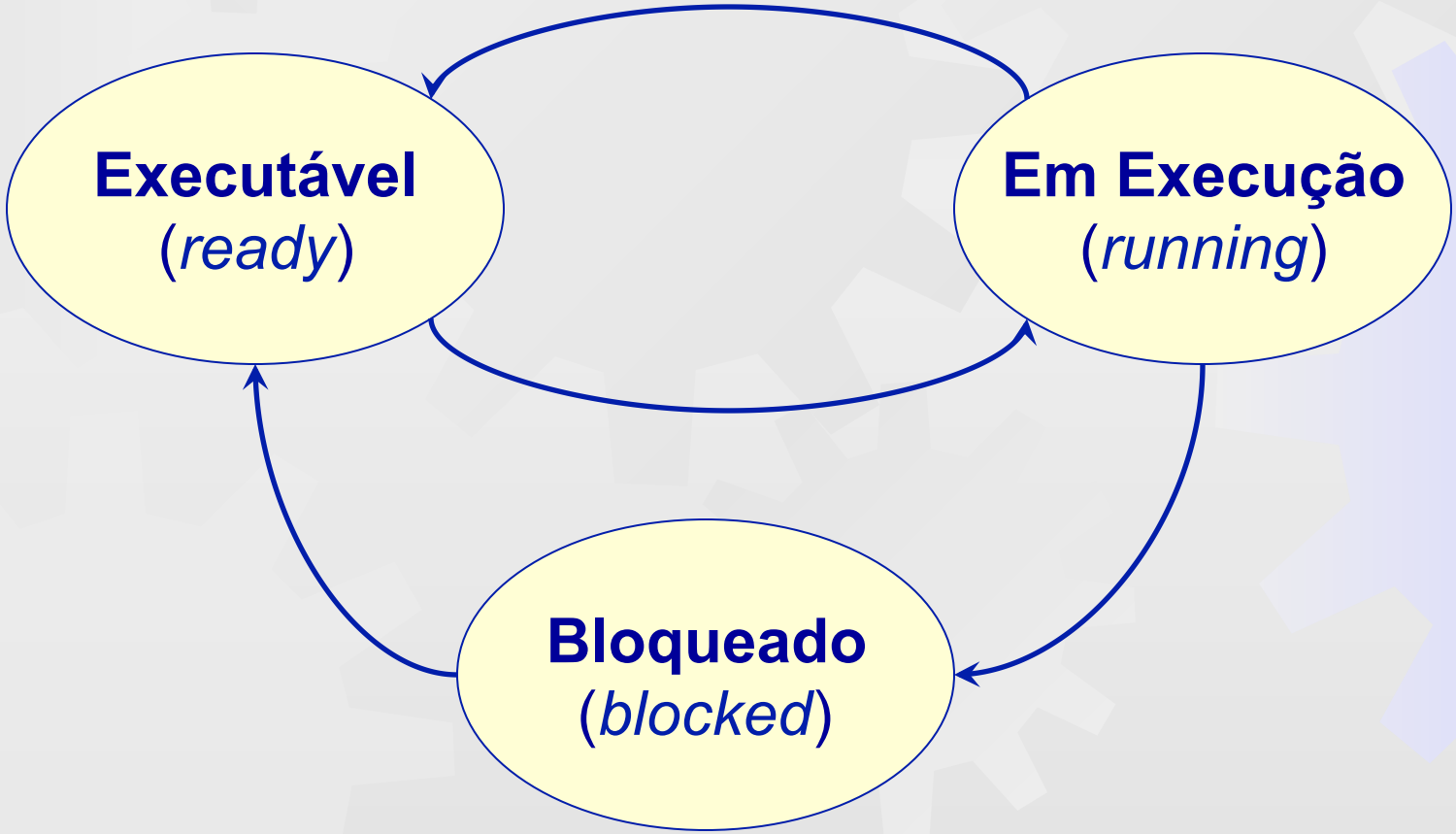
- Concorrência
  - Os vários processos “competem” entre si pela atenção do processador...
- Cooperação
  - ...mas também podem trabalhar em conjunto para a realização de tarefas mais complexas.
  - Esta cooperação exige ao SO a existência de mecanismos de sincronização e comunicação entre processos

Estados de um Processo

- Em Execução
  - O processo está a utilizar o processador
- Executável
  - O processo está activo, mas está à espera de ter a atenção do processador, que nesse instante está dedicado a outro processo
- Bloqueado
  - O processo está inactivo
    - à espera que termine uma operação de I/O
    - à espera que outro processo liberte recursos
    - devido à ocorrência de uma *page fault* – não possui recursos na memória principal

Estados de um Processo

- Diagrama de estados



Criação de Processos

- Inicialização do sistema
  - processo *init* no Linux
  - processo *smss.exe* no Windows 2000
- Um utilizador cria um novo processo ao mandar executar um programa
  - Clicar um ícone
  - Executar um programa a partir da *shell*
    - %> top
- Um processo é criado por outro, através de uma chamada ao sistema
  - Linux: *fork()*
  - Windows 2000: *CreateProcess(.)*

Terminação de Processos

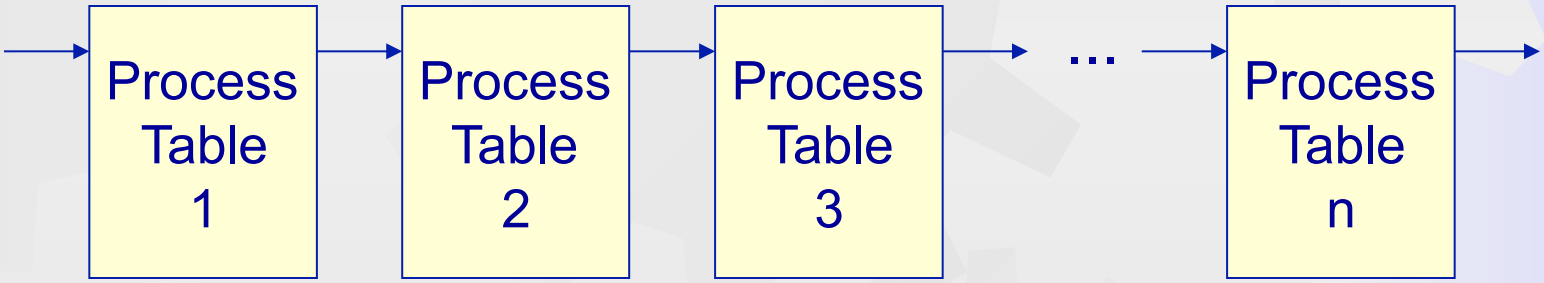
- Um processo pode terminar por diferentes causas:
  - saída voluntária
    - Normal
    - Em erro (previsto)
  - saída involuntária
    - Erro de execução
    - Terminado por outro processo

Hierarquia de Processos

- É comum um SO estabelecer uma hierarquia nos processos que se encontram a correr
  - Processo pai – processo que lança um novo processo
  - Processo filho – novo processo que é criado pelo pai
  - Um processo filho tem apenas um “pai” mas um processo pai pode ter vários “filhos”
    - No Linux é estabelecida uma hierarquia que podemos visualizar através do comando da shell *pstree*
    - No Windows 2000 não existe uma verdadeira hierarquia, pois quando é criado um novo processo, o controlo do mesmo pode ser passado para outro processo diferente do criador

Representação dos Processos

- Internamente, o SO mantém estruturas de dados referentes a cada processo – as *Process Tables*
- As várias *Process Tables* são agrupadas segundo listas ou *arrays* de estruturas



- Cada vez que ocorre uma comutação de processos, o SO salvaguarda e actualiza informação relevante na *Process Table* do processo que "perdeu" a CPU

Informação nas *Process Tables*

Gestão processos	**Gestão de memória**	**gestão de ficheiros**	**...**
PC, SP, ...	program segment	current dir	
estado, prioridade	heap segment (dados)	default dir	
PID, PPID	stack segment	open files	
user	...	...	
data/hora inicio			
tempo CPU			

Representação dos Processos

- Informação nas *Process Tables*
  - Gestão de processos
    - Conteúdo dos registos da CPU (incluindo PC e SP)
    - Estado do processo
    - Prioridade do processo
    - Identificação do processo – PID
    - Identificação do processo pai – (PPID)
    - Identificação do utilizador que lançou o processo
    - Data/hora em que o processo foi iniciado
    - Tempo de CPU utilizado pelo processo

Representação dos Processos

- Gestão de memória
  - Segmento de texto (programa)
  - Segmento de dados (heap)
  - Segmento do stack
- Gestão de ficheiros
  - Directório actual (de trabalho)
  - Directório por defeito (e.g. *root*, *home*)
  - Descritores dos ficheiros abertos
- Etc.

Processos – LINUX

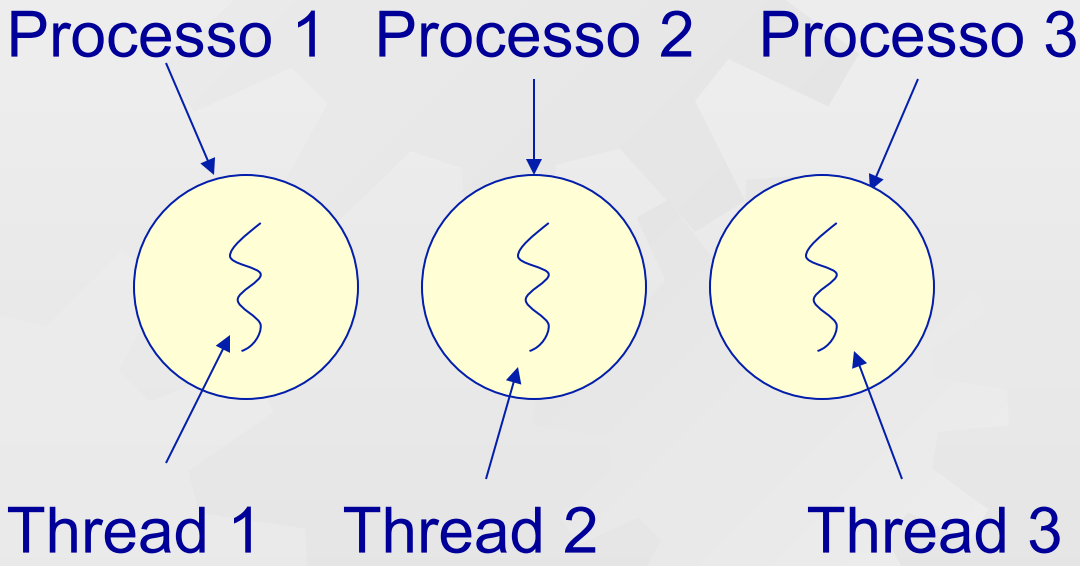
- Comandos da *shell*
  - `ps` – listar processos
  - `pstree` – ver hierarquia dos processos
  - `top` – ver informações adicionais sobre os processos
  - `kill` – enviar sinal a um processo (pode ser um sinal para terminar outro processo)
- Chamadas ao sistema
  - `fork()` – criar um novo processo filho
  - `exit(.)` – terminar processo
  - `kill(.)` – enviar sinal a um outro processo

Threads

- Conceito de *thread (tarefa)*
  - Tal como os processos, as *threads* são também entidades ativas
  - Um processo pode ser composto por várias *threads* que partilham o mesmo espaço de endereçamento
  - Processos diferentes possuem recursos diferentes...
  - ... mas um conjunto de *threads* dentro do mesmo processo partilha os mesmos recursos

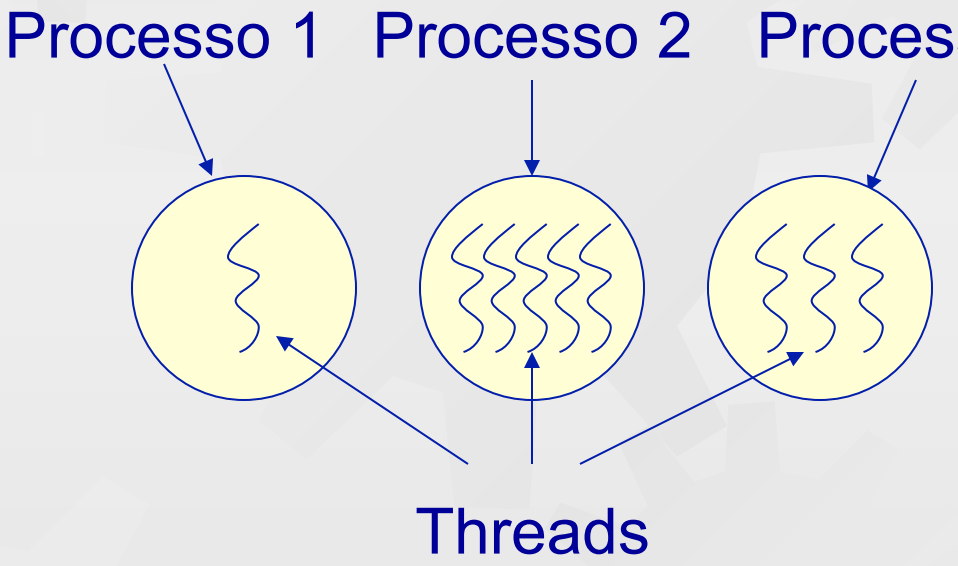
Threads

- Modelo clássico
  - por cada processo existe uma só *thread*
  - neste caso processo e *thread* correspondem ao mesmo conceito



Threads

- Modelo actual
  - por cada processo podem existir várias *threads*



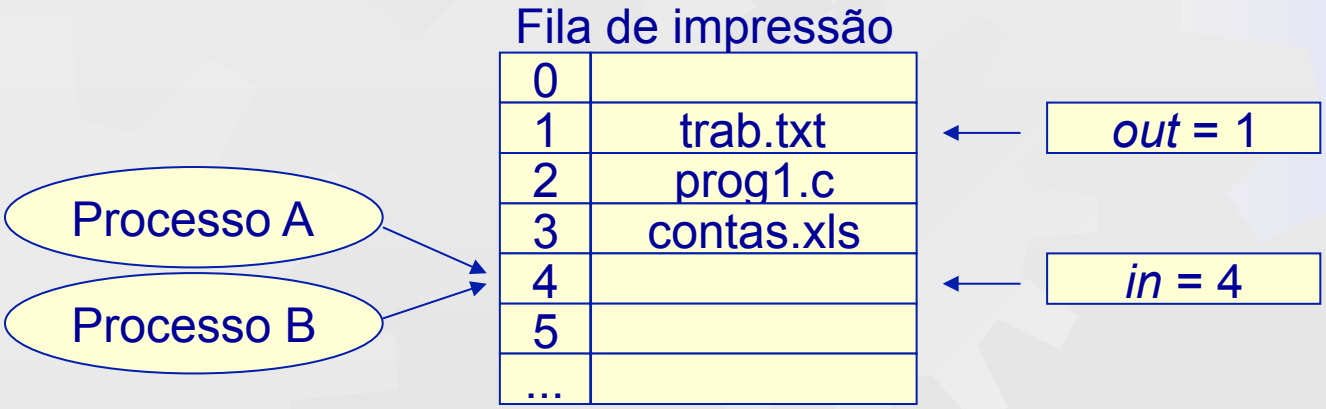
  - Cada *thread* tem registos, *program counter*, *stack* e estado próprios

Comunicação entre Processos

- IPC (*InterProcess Comunnication*)
- Programação concorrente
  - Elaboração de tarefas mais complexas
- É desejável que o SO inclua:
  - Mecanismos de comunicação
    - Troca de dados entre vários processos (através de mensagens ou de partilha de memória)
  - Mecanismos de sincronização
    - Ordem no acesso aos recursos
    - Ordem quando existe dependência entre processos (e.g., o processo A produz dados utilizados pelo processo B)

Regiões Críticas e Exclusão Mútua

- Exemplo – fila de impressão
  - Considere que dois processos A e B encaminham ficheiros para uma fila de impressão com vários *slots*.
  - Para gestão da fila utilizam-se duas variáveis:
    - ***in*** – variável partilhada pelos processos e que indica o próximo *slot* livre na fila
    - ***out*** – variável utilizada pelo processo "*printer daemon*" e que indica o *slot* do próximo trabalho a ser imprimido



Regiões Críticas e Exclusão Mútua

- Exemplo – fila de impressão
  - código utilizado pelos processos

```
void EnviarFicheiro(char NomeFicheiro[]) {
        ...
        ProxSlotLivre = LerPartilhada_in();
        CopiarString(NomeFicheiro, ProxSlotLivre);
        ++ProxSlotLivre;
        ActualizarPartilhada_in(ProxSlotLivre);
        ...
}
```

O que pode acontecer se ocorrer uma comutação de processos entre a leitura da variável *in* e a atualização do seu valor ?

Regiões Críticas e Exclusão Mútua

- **Região crítica**
  - secção do programa onde são efectuados acessos (para leitura e escrita) a recursos partilhados por dois ou mais processos
- **Exclusão mútua**
  - é necessário assegurar que dois ou mais processos não se encontrem simultaneamente na região crítica
  - assegura-se a exclusão mútua recorrendo aos mecanismos de sincronização fornecidos pelo SO
- Afirmações válidas também para as *threads* (é ainda mais crítico, pois todas as *threads* dentro do mesmo processo partilham os mesmos recursos)

Regiões Críticas e Exclusão Mútua

- Regras para programação concorrente
  - Dois ou mais processos não podem estar simultaneamente dentro de uma região crítica
  - Não se podem fazer assunções em relação à velocidade e ao número de CPUs
  - Um processo fora da região crítica não deve causar bloqueio a outro processo
  - Um processo não pode esperar infinitamente para entrar na região crítica

Mecanismos de Sincronização

- Desactivação das interrupções
  - Mecanismo mais básico, que impossibilita a comutação de processos, garantindo assim a exclusão mútua

```
...
DesactivarInts(); /* Desactivar as interrupções */
RegiaoCritica();
ActivarInts();    /* Activar as interrupções */
RegiaoNaoCritica();
...
```

  - Problema
    - É **muito perigoso** dar ao utilizador a possibilidade de desactivar as interrupções (imagina-se facilmente porquê)

Mecanismos de Sincronização

- Trincos lógicos (*locks*)
  - Outra ideia é ter uma variável binária, partilhada por vários processos, que controle o acesso à região crítica

```
...
while (lock==0); /* Em ciclo até poder entrar */
lock = 0;        /* Tranca o acesso */
RegiaoCritica();
lock = 1;        /* Destranca */
...
```

  - Problemas:
    - Pode falhar na garantia da exclusão mútua.
    - Conduz a uma espera ativa

Mecanismos de Sincronização

- Instrução TSL (*Test and Set Lock*)
  - Uma instrução do processador carrega num registo o valor lido de uma posição e de seguida escreve nessa posição um valor diferente de zero (e.g. 1)

```
...
while (TSL(lock)==0); // tenta fazer lock=1
RegiaoCritica();      // 0 se não consegue
lock = 0;
RegiaoNaoCritica();
...
```

  - Problema
    - Resolve a exclusão mútua, mas conduz também a uma espera ativa...

Mecanismos de Sincronização

- **Espera Ativa**

  Um processo ocupa o CPU sem realizar processamento útil, até poder entrar na região crítica.

- As esperas ativas devem ser evitadas porque
  - reduzem a eficiência do processador
  - podem originar um problema designado por problema da inversão da prioridade
    - Um processo prioritário pode dar entrada no sistema sem que outro processo liberte o acesso à região crítica, monopolizando o CPU e ficando infinitamente à espera.

Mecanismos de Sincronização

- Sleep e Wakeup
  - Duas chamadas ao sistema que funcionam do seguinte modo:
    - Sleep() – causa bloqueio ao processo que a invoca
    - Wakeup(PID) – desbloqueia o processo identificado por PID
  - A utilização destas duas chamadas evita esperas ativas, e em conjunto com outros mecanismos (e.g. TSL) consegue-se garantir a exclusão mútua
  - Problema
    - *lost Wakeup signal* – um processo manda “acordar” o outro sem este ter “adormecido” ainda

Mecanismos de Sincronização

- Semáforos
  - Propostos em 1965 por Dijkstra e muito utilizados hoje em dia (embora com variantes)
  - Um semáforo consiste basicamente num número inteiro não negativo
  - Foram originalmente sugeridas duas **operações atómicas** (indivisíveis) sob o ponto de vista do SO :
    - **UP(Sem)** – Incrementa em uma unidade o valor do semáforo *Sem*
    - **DOWN(Sem)** – Tenta decrementar em uma unidade o semáforo *Sem.* Caso o semáforo esteja a "0", o processo que invoca DOWN bloqueia até que o valor do semáforo permita o decremento e a operação seja finalizada

Mecanismos de Sincronização

- Semáforos – protecção da região crítica

```
/* S é um semáforo partilhado por vários processos
   e inicializado com o valor 1 */
...
DOWN(S);                /* Bloqueia se S estiver a 0 */
RegiaoCritica();
UP(S);                  /* Liberta o acesso */
RegiaoNaoCritica();
...
```

Problema do Consumidor e Produtor

- Dois processos partilham um buffer (ou array) de dimensão finita N:
  - processo produtor – coloca elementos no buffer
  - processo consumidor – extrai elementos do buffer

```
/* Produtor */

while (TRUE)
{
    Item = ProduzirItem();
    DepositarItem(Item);
}
```

```
/* Consumidor */

while (TRUE)
{
    Item = RetirarItem();
    ConsumirItem(Item);
}
```

Problema do Consumidor e Produtor

- O código proposto apresenta vários problemas...
  - Não se impede a ocorrência das seguintes situações:
    - o consumidor tenta extrair um elemento quando o *buffer* está vazio
    - o produtor tenta colocar um elemento no *buffer* quando este está cheio
  - O *buffer* é partilhado pelos dois processos, logo o seu acesso constitui uma região crítica – não está garantida a exclusão mútua
- Estes problemas podem ser todos resolvidos utilizando semáforos

Problema do Consumidor e Produtor

Inicialização dos semáforos

Livres - inicializado com N (N é a capacidade do buffer);
Ocups – incializado a 0;
Mutex – inicializado a 1;

```
/* Produtor */

while (TRUE) {
    Item = ProduzirItem();
    DOWN(Livres);
    DOWN(Mutex);
    DepositarItem(Item);
    UP(Mutex);
    UP(Ocups);
}
```

```
/* Consumidor */

while (TRUE) {
    DOWN(Ocups);
    DOWN(Mutex);
    Item = RetirarItem();
    UP(Mutex);
    UP(Livres)
    ConsumirItem(Item);
}
```

Deadlocks (Becos sem Saída)

- Quando se elabora um programa que envolva mecanismos de sincronização é necessário ter muito cuidado...

- No problema anterior, o que pode acontecer se trocarmos a ordem dos DOWNs e se considerarmos que o *buffer* está vazio ?

- Com a troca dos semáforos existe a possibilidade de ambos os processos bloquearem – esta situação designa-se **deadlock ou beco sem saída**

Deadlocks

- Definição:

  Um conjunto de processos está num *deadlock* se cada um dos processos está bloqueado à espera de um sinal dependente de outro processo nesse conjunto.

Exemplo

X e Y são inicializados a 1

Processo A	Processo B
DOWN(X)	DOWN(Y)
DOWN(Y)	DOWN(X)
...	...
UP(Y)	UP(X)
UP(X)	UP(Y)

Outros Mecanismos

- **Mutex**
  - Basicamente um semáforo mais simples que apenas assume os valores 0 e 1 (semáforo binário)
  - São amplamente utilizados para sincronização de *threads*
- **Barreiras**
  - Um mecanismo de sincronização utilizado em arquitecturas multiprocessador quando está envolvido processamento por fases.
  - A barreira não deixa passar nenhum processo para a fase seguinte antes de todos os processos terem terminado a fase corrente

Outros Mecanismos

- **Monitores**
  - Mecanismos de sincronização de alto nível com o objectivo de simplificar a programação concorrente
  - A ideia consiste em definir o código correspondente às regiões críticas dentro de uma rotina especial designada “monitor”
  - O “monitor” garante que apenas um processo pode estar no seu interior bloqueando todos os outros que tentem aceder antes do que lá está sair
  - Utilizados atualmente na linguagem Java (embora de uma forma diferente da definida originalmente)

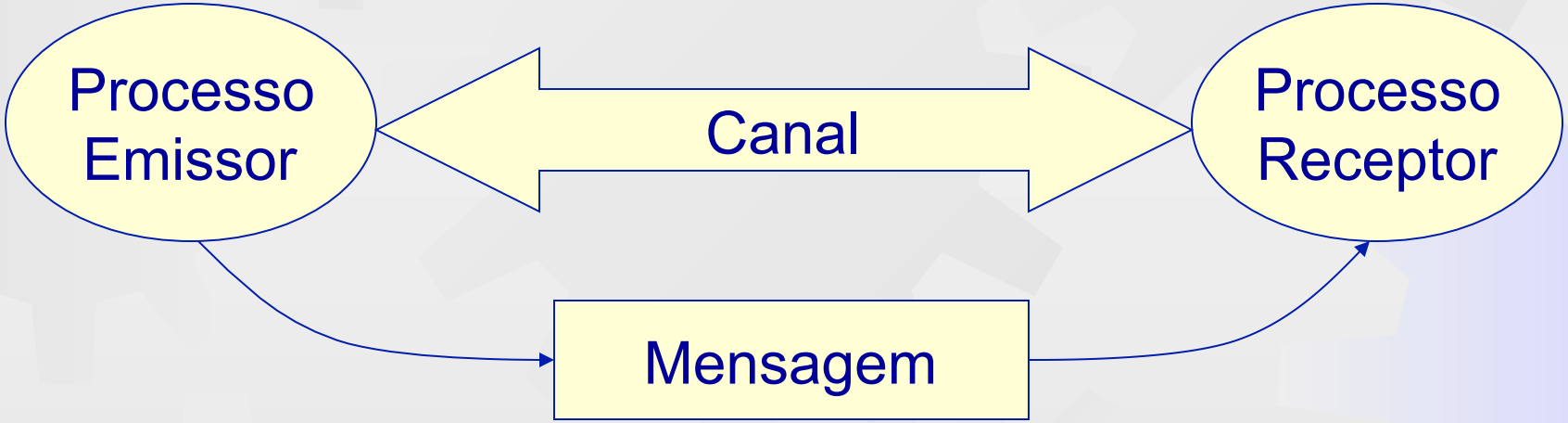
Comunicação - Mensagens

- Dois ou mais processos distintos podem ter necessidade de trocar dados entre si
- Os dados que são trocados constituem uma **mensagem**
- Chamadas ao sistema do tipo
  - *Enviar(Destino, Mensagem)*
  - *Receber(Origem, &Mensagem)*

  As chamadas ao sistema poderão ser bloqueantes

Comunicação - Mensagens

- Modelo de comunicação



Comunicação - Mensagens

- Existem várias formas diferentes de conseguir a comunicação entre processos:
  - Ficheiro
    - Forma trivial
    - Comunicação lenta e com muitas limitações
  - Memória partilhada
    - Dois ou mais processos partilham um segmento de memória
    - Comunicação rápida, mas desprovida de sincronização

Comunicação - Mensagens

- Caixa de correio (ou fila de mensagens)
  - Fila com capacidade para armazenar um número limitado de mensagens
  - Permite a troca de mensagens entre diversos processos
  - Cada mensagem poderá ter um tipo associado, o que facilita a ordem no acesso às mensagens
- Comunicação síncrona *(Rendez-vous)*
  - De cada vez que um processo envia uma mensagem a outro, bloqueia até que o segundo a leia, trocando-se nessa altura os dados de forma directa
  - Poupa-se memória, mas perde-se alguma eficiência em processamento

IPC – Unix/Linux

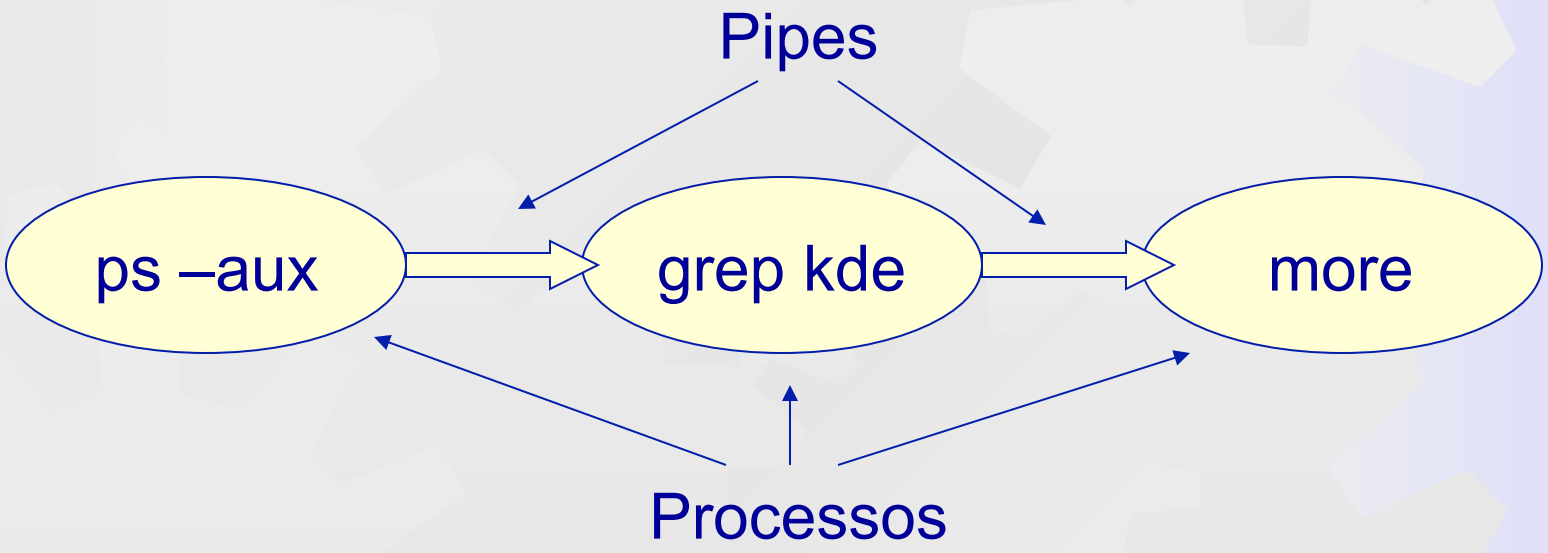
- No Linux existem diversos mecanismos para comunicação e sincronização de processos
  - Pipes
  - Memória Partilhada
  - Filas de Mensagens (ou *Mailboxes*)
  - Sockets (para comunicação entre processos em máquinas diferentes)
  - Semáforos

IPC – Linux

- Pipes
  - Mecanismo original de comunicação entre processos nos sistemas Unix
  - Pipes half-duplex
    - Utilizados para estabelecer um canal de comunicação unidireccional entre processo pai e processo filho
    - O canal de comunicação reside no núcleo do SO
    - Limitação – só podem ser utilizados entre processos relacionados hierarquicamente
    - Os pipes na *shell* são também deste tipo

IPC – Linux

- Pipes half-duplex
  - %> ps -aux | grep kde | more



IPC – Linux

- Pipes half-duplex
  - Chamadas ao sistema (funções C)
    - pipe(.) – criar um pipe
    - read(.) – ler (bloqueia se o pipe está vazio)
    - write(.) – escrever no pipe (bloqueia se o pipe está cheio)
    - close(.) – fechar um dos canais do pipe

    - popen(.) – lançar processo filho e abrir pipe
    - pclose(.) – fechar pipe após terminação do processo filho

IPC – Linux

- *Named Pipes* (FIFOS)
  - A grande diferença em relação aos pipes half-duplex é a comunicação ser efectuada através de um ficheiro especial – FIFO – o canal de comunicação passa a residir no sistema de ficheiros.
    - mknod e mkfifo
  - Cria-se este ficheiro especial e após isso são utilizadas funções normais para escrita e leitura em ficheiros
    - fopen e fclose (abrir e fechar)
    - fgets e fputs (ler e escrever string)
    - etc.
  - Os pipes com nome podem ser utilizados para estabelecer a comunicação entre quaisquer processos

IPC – Linux

- Filas de mensagens
  - Seguem o modelo de comunicação por caixa de correio
  - São utilizadas para comunicação entre vários processos
  - Chamadas ao sistema (funções C)
    - msgget – criação ou associação
    - msgsnd – envio de mensagens (causa bloqueio se a fila estiver cheia)
    - msgrcv – recepção de mensagem (causa bloqueio se a fila não tiver nenhuma mensagem pretendida)
    - msgctl – operações de controlo e remoção

IPC – Linux

- Memória partilhada
  - Define-se um conjunto de posições de memória que é partilhada por dois processos
  - Chamadas ao sistema (funções C)
    - shmget – criação ou associação
    - shmat – mapeamento do segmento de memória partilhada para o espaço de endereçamento do processo
    - shmdt – liberta o segmento do espaço de endereçamento do processo
    - shmctl – controlo e remoção
  - Atenção, pois estas chamadas ao sistema não são bloqueantes, pelo que é necessária a existência de mecanismos de sincronização

IPC - Linux

- Semáforos
  - No Unix/Linux, existem algumas extensões às operações sobre semáforos atrás descritas:
    - Podem-se efectuar UPs e DOWNs com mais do que uma unidade
    - Pode-se operar com semáforos como se estes fossem binários
  - Chamadas ao sistema (funções C)
    - semget – criação ou associação a um grupo de semáforos
    - semop – operações sobre um grupo de semáforos
    - semctl – controlo, inicialização e remoção

Gestor de processos

- Despacho (*dispatcher*)
  - rotina executada sempre que o processo em execução deva ser comutado ou exista essa possibilidade
- Escalonador (*scheduler*)
  - Executa-se quando ocorre uma comutação de processos
  - determina quais os processos que devem prioritariamente executar-se, tentando otimizar a utilização dos recursos (em particular do CPU)
  - A escolha é feita de acordo com um dado algoritmo de escalonamento
  - Após a escolha, o despacho encarrega-se de colocar o processo em execução.

Escalonamento

- Objectivos gerais do escalonamento
  - Justiça – garantir que todos os processos terão direito a uma fatia justa de tempo de CPU
  - Equilíbrio – manter os recursos do sistema com uma elevada taxa de ocupação, sempre que possível
  - Prioridades – dar maior tempo de CPU aos processos com maior importância
- mas ... o projeto do escalonador deve ter em conta as características do sistema em causa
  - Sistema em lotes (batch)
  - Sistema interativo
  - Sistema em tempo real

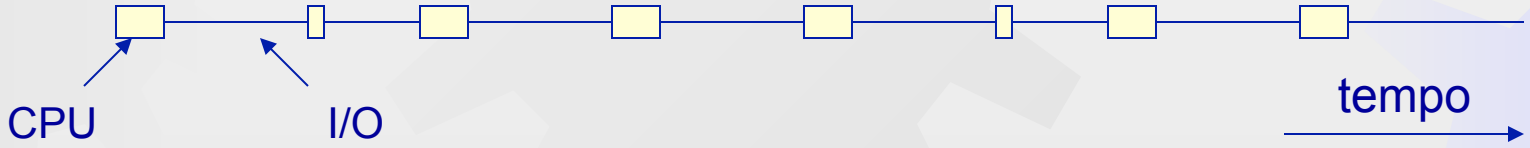
Escalonamento

- Objectivos do escalonamento
  - Sistemas Batch
    - Maximizar nº de processos concluídos por unidade tempo (*throughput*)
    - Maximizar a taxa de utilização do CPU em processamento útil
    - Minimizar o tempo entre a submissão e terminação
  - Sistemas interativos
    - Minimizar o tempo de resposta
    - Proporcionalidade – satisfazer espectativas dos utilizadores
    - Prioridades – dar maior tempo de CPU aos processos com maior importância
  - Sistemas real-time
    - Cumprir os *deadlines*
    - Previsibilidade – um mesmo programa deve ser corretamente executado, independentemente da carga do sistema

Mas ... alguns dos objectivos são contraditórios...

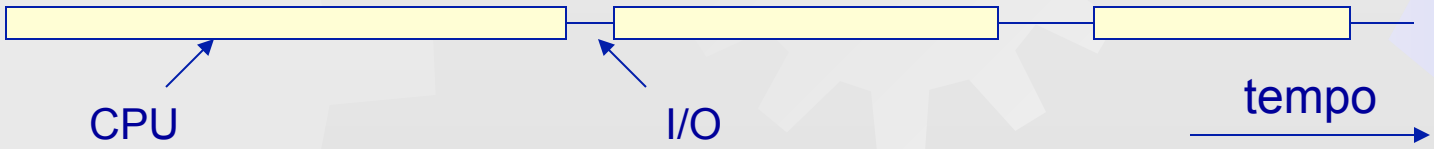
Escalonamento

- Comportamento dos processos
  - ***I/O-bound***
  - Processo caracterizado por uma taxa elevada de operações I/O face à utilização do CPU



  - ***Compute-bound***

    Processo caracterizado por uma taxa elevada de utilização do CPU face a operações de I/O

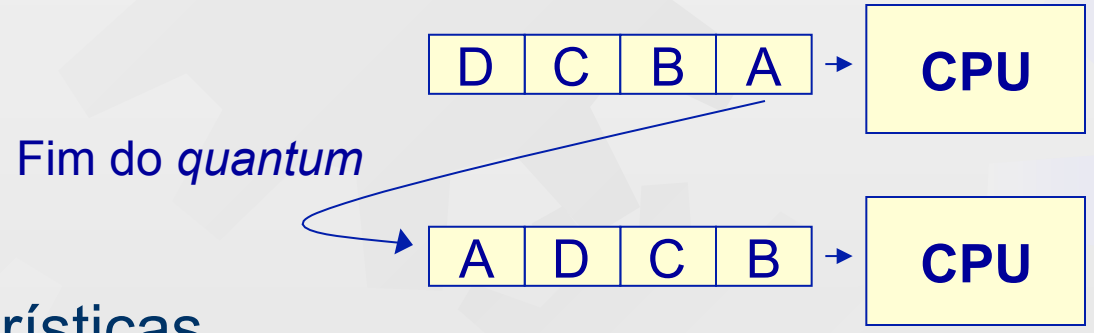


Algoritmos de escalonamento

- *First-come, first-served* (ou FIFO)
  - O CPU é atribuído aos processos pela sua ordem de chegada
  - Cada processo monopoliza o CPU até terminar, ou até bloquear numa operação de I/O
  - Características:
    - Algoritmo muito simples
    - Não é aplicável para processamento interativo, mas pode ser utilizado em conjunto com outros algoritmos
    - Utilizado em sistemas *batch*

Algoritmos de escalonamento

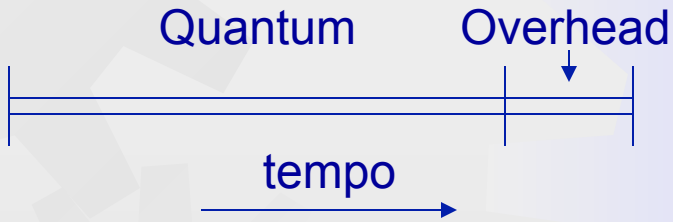
- *Round-robin*
  - Cada processo tem direito a um certo tempo de CPU – o ***quantum*** – ou *time-slice*
  - Após o fim do *quantum* é colocado no fim da fila dos processos executáveis



  - Características
    - Trata todos os processos de modo igual
    - Permite interatividade
    - Utilizado em conjunto com outros algoritmos

Algoritmos de escalonamento

- *Round-robin*
  - O dimensionamento do *quantum* pode ter um impacto muito forte no desempenho do sistema
    - Quantum pequeno – o CPU perde rendimento...
    - Quantum grande – aumenta o rendimento, mas perde-se interactividade...



$$\text{Rendimento} = \frac{\text{Tempo de processamento útil}}{\text{Tempo de processamento total}} = \frac{\text{Quantum}}{\text{Quantum} + \text{Overhead}}$$

- *Overhead* – tempo necessário para o SO actualizar estruturas quando ocorre uma comutação de processos

Algoritmos de escalonamento

- Prioridades
  - Existem processos mais importantes do que outros
    - A cada processo é atribuído um valor de prioridade
    - O escalonador ordena os processos por ordem de prioridade
    - O CPU é atribuído ao processo com maior prioridade
  - Para evitar que processos prioritários monopolizem o CPU, a prioridade poderá ter duas componentes
    - **Prioridade = prioridade base + prioridade dinâmica**
    - **Base** – valor fixo, correspondendo à prioridade com que o processo é iniciado
    - **Dinâmica** – valor variável ao longo do tempo, calculada pelo escalonador em certos instantes temporais

Algoritmos de escalonamento

- Prioridades
  - Prioridades dinâmicas (exemplo de algoritmo simples)
    - Definir a prioridade dinâmica com base na fracção de tempo do *quantum* – *f* – que um processo utilizou antes de iniciar uma operação de I/O
    - Prioridade dinâmica = 1/f
    - Com este algoritmo beneficiam-se os processos I/O-bound

Quantum = 10 ms

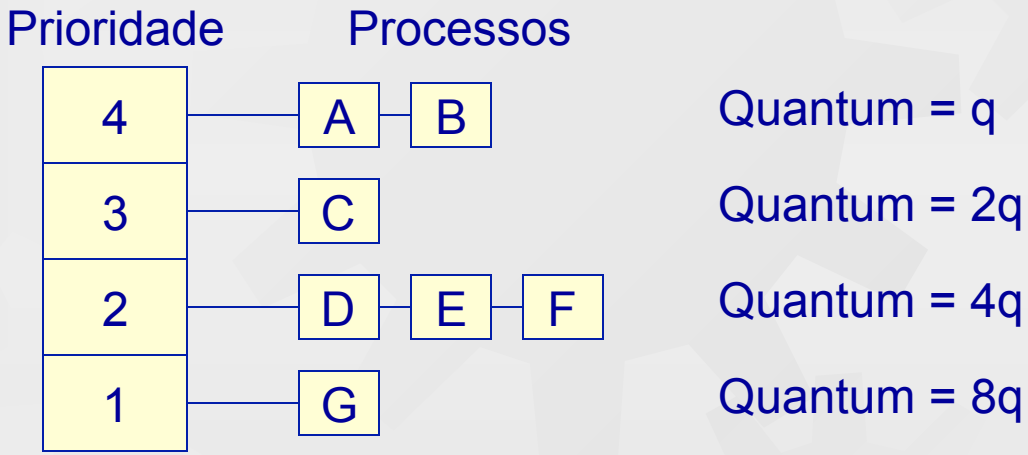
Bloqueio ao fim de 1 ms => prior. din. = 10
Bloqueio ao fim de 2 ms => prior. din. = 5
Bloqueio ao fim de 5 ms => prior. din. = 2

Algoritmos de escalonamento

- Multifila
  - Algoritmo que utiliza várias filas *round-robin* (ou FIFO) com prioridades e *quantuns* diferentes em cada uma
    - *Quantuns* pequenos para prioridades altas
    - *Quantuns* grandes para prioridades baixas
  - Em cada comutação um processo baixa a prioridade mas aumenta o seu *quantum*
    - Processos I/O-bound ficarão com prioridades altas e *quanta* pequenos
    - Processos compute-bound ficarão com prioridades baixas, mas correm de um modo mais eficiente com *quantuns* altos

Algoritmos de escalonamento

- Multifila
  - Exemplo



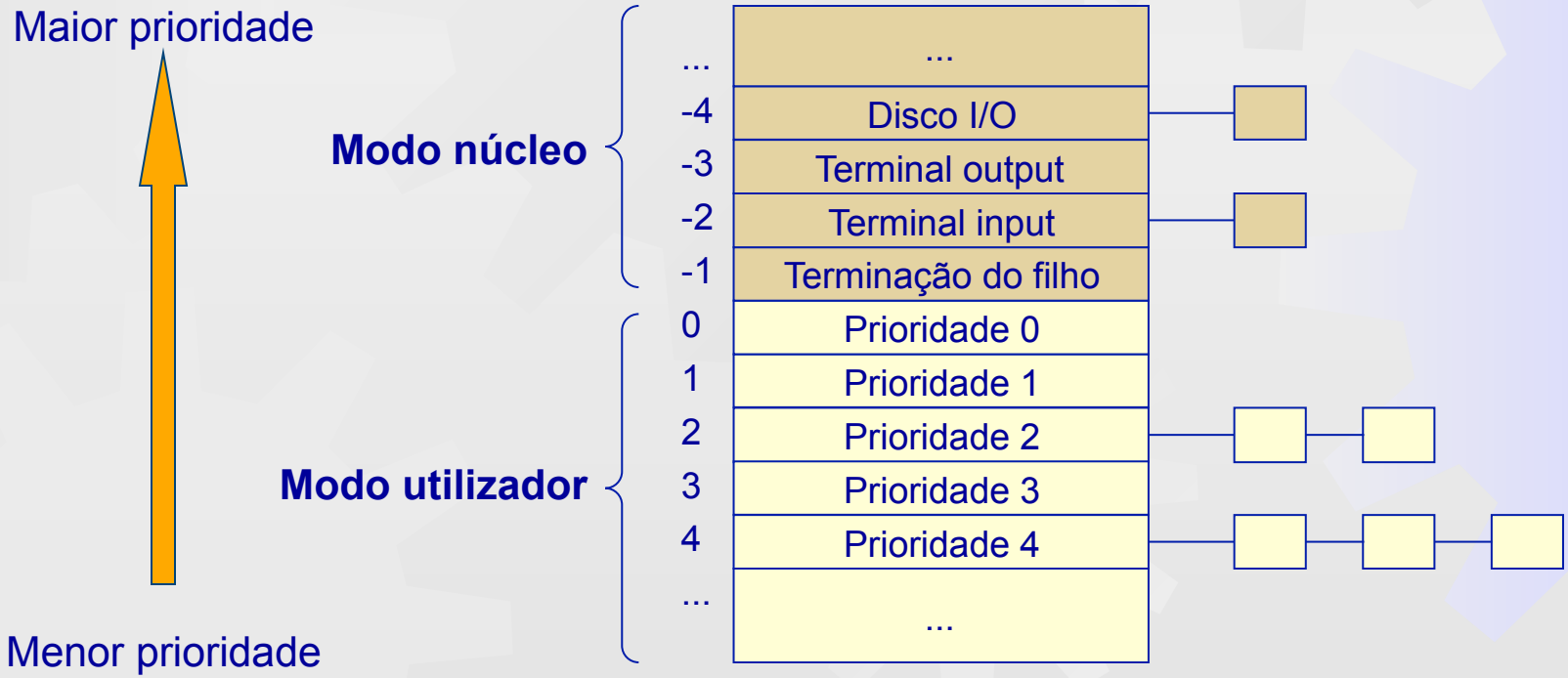
- Calcular quantas comutações seriam necessárias para um processo cujo tempo de execução seja de 20q. Supondo que o overhead é de q/5, calcule o rendimento e compare com o obtido se o *quantum* fosse sempre q.

Dois tipos de escalonamento

- Escalonamento sem preempção (*Non-preemptive*)
  - O algoritmo de escalonamento só corre após o bloqueio do processo que ocupa o CPU ou quando o seu *quantum* expira
    - Operação de I/O
    - Bloqueio num semáforo
    - Ocorrência de *Page fault,* etc.
    - Terminou o *quantum*
- Escalonamento com preempção (*Preemptive*)
  - O algoritmo de escalonamento corre em todas as situações que podem mudar o estado dos processos.
    - Quando o processo que ocupa o CPU bloqueia
    - Em instantes temporais pré-determinados (interrupção do relógio)
    - Quando entra um processo prioritário (em sistemas de tempo-real)

Escalonamento – UNIX (caso geral)

- Baseado em filas de prioridades divididas em modo núcleo (negativas) e modo utilizador (positivas)
- Algoritmo *round-robin* em cada fila de prioridade



Escalonamento – UNIX (caso geral)

- Prioridades em modo núcleo
  - Várias filas com prioridades negativas, cada uma delas correspondendo à saída de um bloqueio causado por uma dada situação em particular
    - Espera pela terminação do filho
    - Espera por *input* do terminal
    - Etc.
  - O objectivo deste esquema consistem em
    - acelerar pedidos de I/O dos processos *I/O-bound*;
    - servir rapidamente processos interativos;
    - libertar rapidamente os processos do modo núcleo.

Escalonamento – UNIX (caso geral)

- Prioridades em modo utilizador
  - Prioridades dinâmicas que visam lidar com processos compute-bound. Usam-se 2 campos:
    - p_cpu: acumulador do tempo de execução pelo processo
    - p_pri: valor atual da prioridade do processo
  - Periodicamente (10ms)
    - incrementa-se p_cpu do processo em execução
  - 1 vez por segundo, recalculam-se os valores das prioridades e tempos de execução, de acordo com
    - Prioridades: p_pri = prioridade base + p_cpu/2+ nice
    - Tempos de execução: p_cpu = p_cpu / 2
  - Deste modo, não se penaliza tanto um processo CPU-bound

Escalonamento - LINUX

- Linux tigre.iul.lab 3.2.0-4-amd64 #1 SMP Debian 3.2.46-1+deb7u1 x86_64 GNU/Linux

```
top - 19:25:21 up 17 days,  1:05, 18 users,  load average: 0.04, 0.03, 0.05
Tasks: 187 total,   1 running, 183 sleeping,   3 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.2 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1023480 total,   524720 used,   498760 free,    80200 buffers
KiB Swap:  2031612 total,        0 used,  2031612 free,   126800 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
10105 fmmb      20   0 23288 1668 1180 R   0.7  0.2   0:00.73 top
22280 a54430    20   0 73256 1816  992 S   0.3  0.2   0:01.07 sshd
    1 root      20   0 10648  836  696 S   0.0  0.1   0:14.11 init
    2 root      20   0     0    0    0 S   0.0  0.0   0:00.01 kthreadd
    3 root      20   0     0    0    0 S   0.0  0.0   0:21.43 ksoftirqd/0
    4 root      20   0     0    0    0 S   0.0  0.0   0:58.00 kworker/0:0
    5 root      20   0     0    0    0 S   0.0  0.0   0:00.00 kworker/u:0
    6 root      rt   0     0    0    0 S   0.0  0.0   0:03.56 migration/0
    7 root      rt   0     0    0    0 S   0.0  0.0   0:05.59 watchdog/0
    8 root      rt   0     0    0    0 S   0.0  0.0   0:02.89 migration/1
    9 root      20   0     0    0    0 S   0.0  0.0   3:12.42 kworker/1:0
   10 root      20   0     0    0    0 S   0.0  0.0   0:38.72 ksoftirqd/1
   11 root      20   0     0    0    0 S   0.0  0.0   0:38.28 kworker/0:1
   12 root      rt   0     0    0    0 S   0.0  0.0   0:05.28 watchdog/1
   13 root       0 -20     0    0    0 S   0.0  0.0   0:00.00 cpuset
   14 root       0 -20     0    0    0 S   0.0  0.0   0:00.00 khelper
   15 root      20   0     0    0    0 S   0.0  0.0   0:00.00 kdevtmpfs
   24 root      25   5     0    0    0 S   0.0  0.0   0:00.00 ksmd
   25 root      39  19     0    0    0 S   0.0  0.0   0:00.00 khugepaged
   26 root      20   0     0    0    0 S   0.0  0.0   0:00.00 fsnotify_mark
```

Escalonamento - LINUX

- Funcionamento do algoritmo
  - A escolha do escalonador é feita segundo a *goodness* de cada thread – escolhe o que obter *goodness* mais alta

```
if (classe == realtime)
        goodness = 1000 + prioridade;
if (classe == timesharing && quantum>0)
        goodness = quantum + prioridade;
if (classe == timesharing && quantum==0)
        goodness = 0;
```

  - Valores de prioridade entre 1 e 40 (40 é a mais elevada) – 20 é o valor por defeito
  - O *quantum* é medido em *clock ticks* (chamados ***jiffys***). Cada ***jiffy*** corresponde a 10ms (por defeito)
  - Em cada *clock tick*, o valor do *quantum* é decrementado em uma unidade

Escalonamento - LINUX

- A partir do kernel versão 2.6.23
  - Completely Fair Schedule (CFS) [Molnar, 2007]
    - Usa uma red-black tree para manter os processos ordenados por tempo de execução.
      - Evita o problema da escalabilidade

Escalonamento – LINUX

- Uma *thread* perde o processador se:
  - O seu *quantum* chegar a 0;
  - Bloquear (semáforo, I/O, etc.)
  - Uma *thread* com maior *goodness* desbloquear
- Nessa altura é calculado um novo valor para os *quantuns* (em jiffys) de todas as *threads* (activas e bloqueadas)
  - Novo quantum = quantum que sobrou / 2 + prioridade
- O objectivo é beneficiar *threads* I/O-bound
  - *I/O-bound* – o *quantum* tende para o dobro do valor da prioridade, consequentemente aumentando a *goodness*
  - *compute-bound* – o *quantum* tende para um valor igual ao da prioridade