INTRODUÇÃO

- **O que é um Sistema Operativo (SO) ?**
- **Objectivos de um SO.**
- **Alguns conceitos básicos sobre SO's.**
- **Evolução dos SO's. Tipos de SO's.**

O que é um Sistema Operativo ?

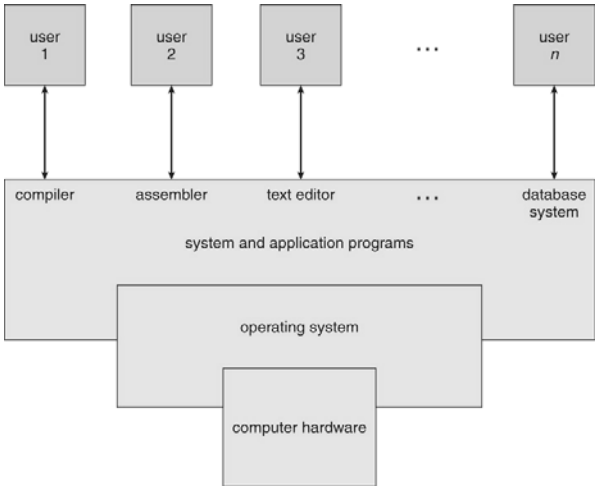
Um programa grande e complexo (/ conjunto de programas) que controla a execução dos programas do utilizador e actua como intermediário entre o utilizador de um computador e o *hardware*.

Objectivos principais de um SO:

- **fornecer uma gestão eficiente e segura dos recursos computacionais (gestão + controlo)**
- **fornecer ao utilizador uma máquina virtual mais fácil de programar do que o *hardware* subjacente (conveniência + eficiência)**

O que seria dos programadores sem um sistema operativo ?

Onde "encaixa" o SO ?



Alguns conceitos sobre SO's

Núcleo (*Kernel*)

- **A parte principal do SO.**
  Contém código para os serviços fundamentais.
  Está sempre em memória principal.

Device Drivers

- **Código que fornece uma interface simples e consistente com os dispositivos de *I/O***
- **Podem fazer parte do *kernel* ou não.**

Programa

- **Um ficheiro do disco contendo código numa linguagem de alto nível ou código-máquina (programa executável).**

Processo

- **Um programa em execução.**
- **A colecção de estruturas de dados e recursos do SO detidos por um programa enquanto está a ser executado.**

Ficheiro

- Colecção de informação relacionada entre si.
- Unidade lógica de armazenamento.
- O SO mapeia os ficheiros em dispositivos físicos onde a informação é gravada de forma permanente (memória secundária).
- Para muitos utilizadores, o sistema de ficheiros é o aspecto mais visível de um SO.

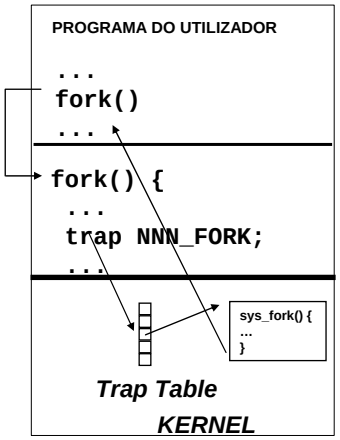
Chamadas ao sistema

- Os programas do utilizador comunicam com o SO e pedem-lhe serviços fazendo chamadas ao sistema.
- A cada chamada corresponde uma rotina da biblioteca de sistema.
- Esta rotina coloca os parâmetros da chamada ao sistema em locais especificados (ex.: registos do processador) e executa uma instrução de *trap* para passar o controlo ao sistema operativo.

Shell

- Interpretador de comandos dados ao sistema operativo.
- Frequentemente, não faz parte do sistema operativo.

Chamadas ao sistema

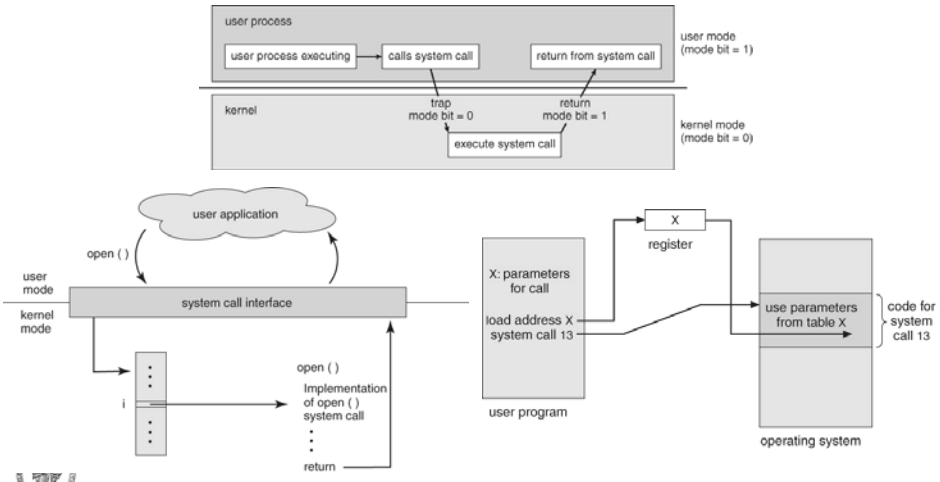


Modo utilizador

- Programa faz *trap* (interrupção gerada por *software*)
- O SO determina o nº do serviço
- O serviço é localizado e executado.
- O controlo retorna ao programa do utilizador

Modo supervisor
/ sistema
/ kernel

Chamadas ao sistema



Evolução dos sistemas operativos

- **Processamento em série**
- **Processamento em lote *(batch)***
- **Multiprogramação**
- **Tempo partilhado (*time-sharing*)**
- **Multiprocessamento**
- **Sistemas distribuídos**
- **Sistemas de tempo-real**

Evolução …

- **Processamento em série**
- **Processamento em lote (*batch*)**

… melhoramentos sucessivos

- **dispositivos de *I/O***
- **implementação dos SOs**

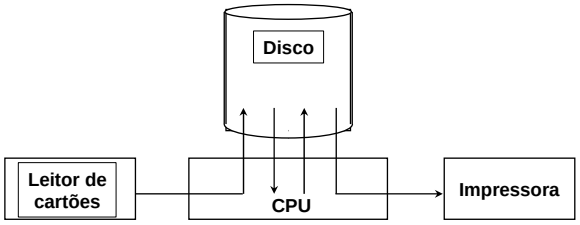
- **evolução do *SW* ↔ evolução do *HW***

Melhoramentos:

- **Sobreposição das operações de entrada e de saída (*I/O*)**
  (coincide c/ a introdução de canais de *DMA*, controladores de periféricos, ...)

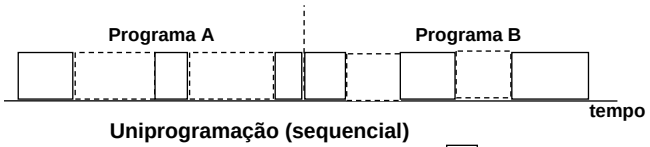
Spooling (Simultaneous Peripheral Operations On-Line)

- **Forma de *buffering*: usar discos p/guardar temporariamente as *I/O*'s.**
- **Permite sobrepor a fase de cálculo de um processo c/ a fase de *I/O* de outro.**

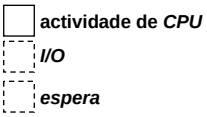


Multiprogramação

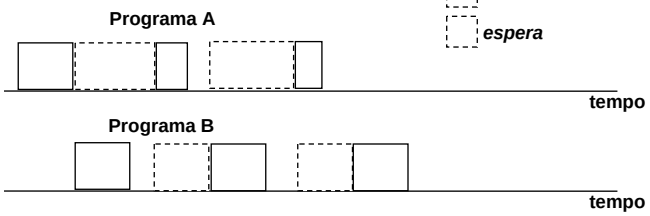
Várias tarefas são mantidas em memória simultâneamente, e a *CPU* é partilhada entre elas



Uniprogramação (sequencial)



Quando o programa actual fica à espera que uma operação de *I/O* ***(p/mesmo dispositivo)*** se complete, o processador pode executar outro programa



Multiprogramação c/ 2 programas & mesmo dispositivo de *I/O*

Multiprogramação

Multiprogramação: execução interlaçada de processos.

Algumas características do SO necessárias para multiprogramação

- **Escalonamento da *CPU***
  o sistema deve escolher entre os várias processos prontos a executar, aquele que vai ser executado
- **Gestão de memória**
  alocar a memória aos diferentes processos
- **Gestão de *I/O***
  controlar o acesso aos dispositivos de *I/O*
- **Protecção**
  não deve haver possibilidade de os processos se afectarem mutuamente em todos os níveis: escalonamento de *CPU*, memória acessível, *I/O*, ...

Sistemas de tempo partilhado (*time-sharing*) - Computação interactiva

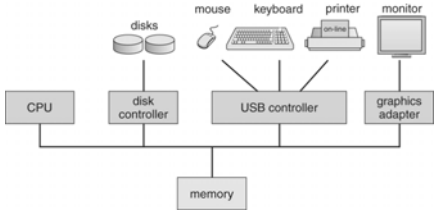
- Vários utilizadores simultâneos,
  cada um com a impressão de que tem o computador só para si.
- A *CPU* é partilhada entre diversas tarefas
  que são mantidas em memória e em disco
  (a *CPU* só é alocada a uma tarefa se ela estiver em memória).
- A comutação entre tarefas
  ocorre com uma elevada frequência.
- É possível a comunicação *on-line* entre o utilizador e o sistema;
- Deve existir um sistema de ficheiros *on-line*
  para que os utilizadores possam aceder aos programas e aos dados.

Outras evoluções

- Computadores pessoais
- Sistemas embebidos
- Sistemas de tempo-real
- Redes de computadores
- ...

HARDWARE DE UM SISTEMA DE COMPUTAÇÃO

- **Revisão de conceitos básicos sobre *hardware***
  - » **Processador *(recordar conceitos de Arq.Comp.)***
  - » **Interrupções**
  - » **Processamento de E/S**
  - » **Memória**
- **Protecção do *hardware***



Interrupções

- **Uma interrupção é um mecanismo que permite que o processamento normal de um processador seja interrompido.**
- **As interrupções são usadas para aumentar a eficiência, especialmente quando se usam componentes que operam a velocidades diferentes.**
- **Permitem que enquanto decorre uma operação de E/S de um processo o processador continue a executar outros processos**
- **→ base da multiprogramação**
- **Alternativa ao uso de interrupções: *polling***
  - ♦ ☹ **introduz *busy-waiting***

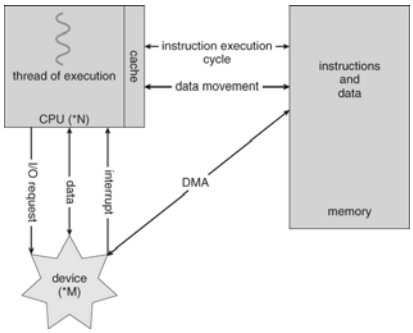
Acesso Directo à Memória *(DMA)*

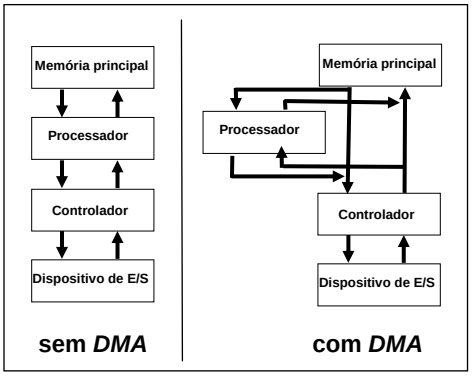
Necessário um controlador de *DMA* ligado ao barramento do sistema.

Quando é necessário fazer E/S

- **o processador informa o controlador do dispositivo de E/S do que pretende fazer e onde está ou vai ficar a informação a transferir**
- **o processador continua a executar outras instruções**
- **o dispositivo de E/S transfere a informação directamente de/para a memória**
- **quando o *DMA* termina é gerada uma interrupção**

Entrada / saída





Protecção do *hardware*

Aumentar a utilização do sistema
⇒ partilha do sistema
⇒ vários programas a executar em simultâneo
⇒ protecção

O SO deve impedir que
um programa incorrecto ou "mal intencionado"
impeça os outros programas de executar.

Alguns erros de programação são detectados pelo *hardware*.

Normalmente estes erros são tratados pelo SO.

Protecção do *hardware* (cont.)

Protecção

- **duplo modo de operação**
  - » **modo utilizador**
  - » **modo supervisor / sistema / monitor / privilegiado (instruções privilegiadas)**
- **protecção de E/S's**
  - » **os utilizadores não conseguem fazer E/S directamente, só através do SO**
- **protecção da memória**
  - » **protecção da área de mem. do SO e dos utilizadores feita por registos especiais**
- **protecção do processador**
  - » **temporizador impede que uma aplicação tome conta do processador indefinidamente**

Requisitos de *hardware* para multiprogramação

Um SO com multiprogramação necessita de suporte de *hardware*:

- **temporizador**
- ***hardware* de *DMA***
- **mecanismo de interrupções com prioridades**
- **duplo modo de operação do processador**
- **mecanismo de protecção da memória**
- **mecanismo de atribuição dinâmica de endereços**
- **...**

ESTRUTURA DO SISTEMA OPERATIVO

- **Componentes do sistema operativo**
- **Tipos de estrutura**
- **Estrutura de sistema operativos concretos**

Componentes do S.O.

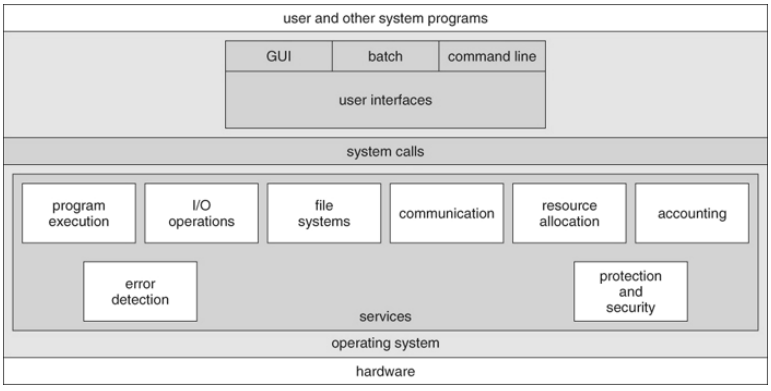
Pontos de vista de um sistema operativo:

- ♦ **serviços que fornece**
- ♦ **interface que disponibiliza p/ utilizadores e programadores**
- ♦ **seus componentes e interligações**

Componentes do sistema operativo:

- ♦ **Gestão de processos**
- ♦ **Gestão da memória principal**
- ♦ **Gestão da memória secundária**
- ♦ **Gestão de ficheiros**
- ♦ **Gestão de entradas/saídas**
- ♦ **Gestão de rede**
- ♦ **Sistema de protecção/segurança**

Serviços fornecidos por um SO



Tipos de estrutura de um S.O.

Estrutura:

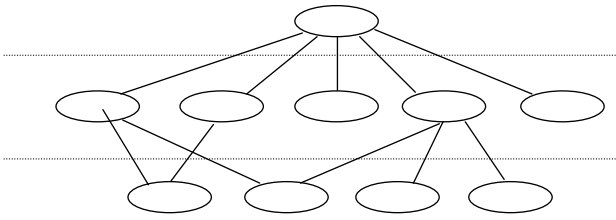
- ♦ **monolítica**
- ♦ **em camadas**
- ♦ ***microkernel***

Estrutura monolítica

Primeiros S.O.'s.

- **Não há estruturação ...**
  - ♦ **o S.O. é escrito como um conjunto de procedimentos cada um dos quais pode chamar qualquer outro.**

… ou há uma pequena estruturação (ex: MS-DOS):



proced. principal

proced.s de serviço
(executam as chamadas ao sist.)

proced.s utilitários
(ajudam os proced.s de serviço)

Dificuldades da estrutura monolítica:

- ♦ **difícil de compreender**
- ♦ **difícil de modificar**
- ♦ **pouco fiável (um erro "em qualquer lado" pode provocar um *crash*)**

Estrutura em camadas

- **O S.O. é dividido num certo número de camadas (níveis) cada qual construída por cima da anterior.**
  - ♦ **camada de mais alto nível - interface com o utilizador**
  - ♦ **camada 0 – *hardware***
- **Sistema operativo modular**
  - ♦ **Para cada camada especificar a funcionalidade e as características.**
  - ♦ **É possível alterar a estrutura interna de cada camada desde que a interface com as outras camadas se mantenha inalterada.**
  - ♦ **Cada camada só usa funções e serviços das camadas inferiores.**
  - ♦ **Uma camada não necessita de "saber" como as operações da camada inferior são implementadas, mas apenas o que elas fazem.**

Dificuldades da estruturação em camadas:

- ♦ **Definição adequada das camadas**
  - » **porque cada camada só deveria poder usar as funções do nível inferior, mas …**
  - » **… ex: o sistema de gestão de ficheiros deveria ser um processo numa camada superior à de gestão de memória virtual; por sua vez, esta deverá poder usar ficheiros (!)**
- ♦ **Tende a ser menos eficiente do que outros tipos**
  - » **ex: para um programa do utilizador executar uma operação de *I/O***
    - executa uma chamada ao sistema
    - que faz um *trap* à camada de *I/O*
    - que chama a camada de …
    - … até chegar ao *hardware*

Os sistemas são frequentemente modelados como estruturas em camadas mas nem sempre são construídos dessa forma.

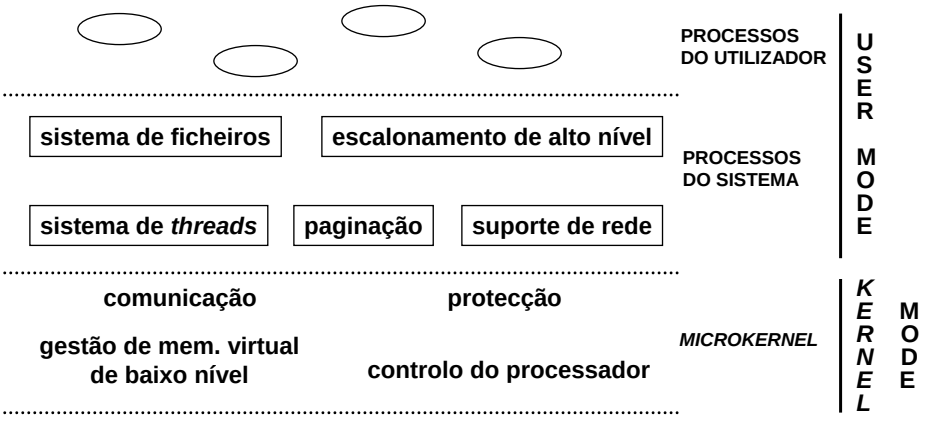
Estrutura baseada em *microkernel*

- **Tendência nos S.O.'s modernos:**
  - ♦ **Deslocar código para as camadas superiores deixando um *kernel* mínimo.**
  - ♦ **O *kernel* implementa a funcionalidade mínima referente a**
    - gestão básica da *CPU*
    - gestão de memória
    - suporte de *I/O*
    - comunicação entre processos
  - ♦ **A restante funcionalidade do S.O. é implementada em proc.os de sistema que correm em modo de utilizador; estes processos comunicam entre si através de mensagens (modelo cliente-servidor)**

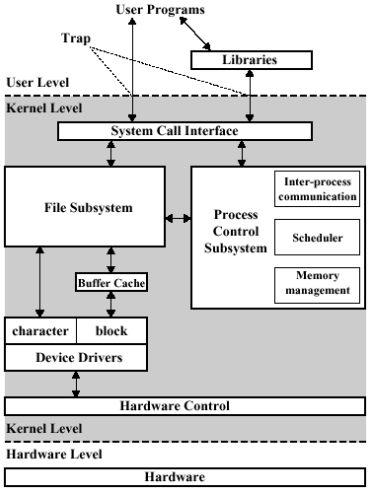
Primeiro sistema baseado em *microkernel*: Hydra (CMU, 1970)
Outros exemplos: Mach (CMU), Chorus (Unix-like, francês), Minix

Windows NT - estrutura *microkernel* modificada;
ao contrário de uma arquitectura *microkernel* "pura"
muitas das funções de sistema fora do microkernel
executam em modo *kernel*, por razões de performance

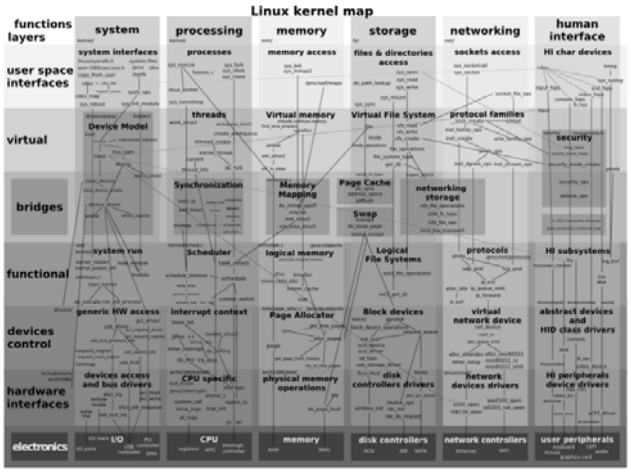
Estrutura baseada em *microkernel*:



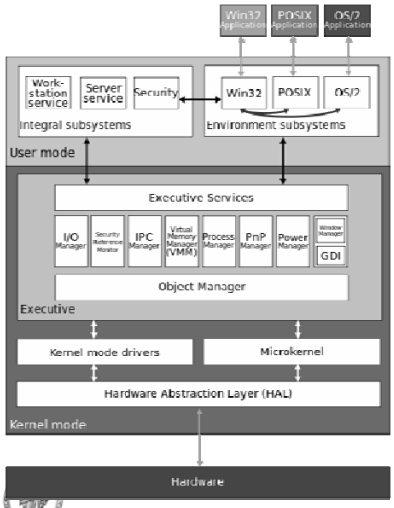
Arquitectura do Unix (Bach)

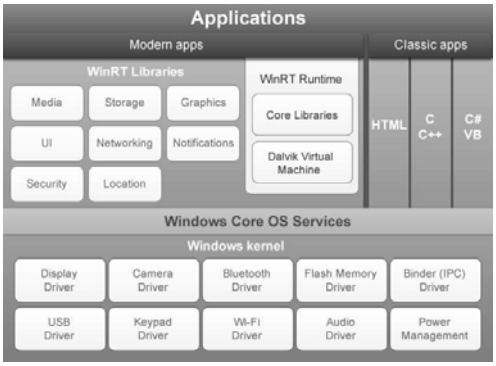


Arquitectura do Linux



Arquitectura do Windows NT & Windows 8

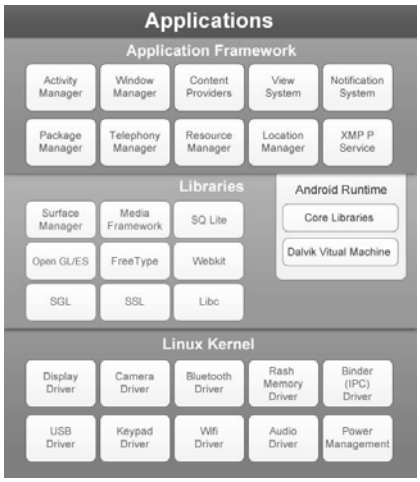




http://en.wikipedia.org/wiki/Architecture_of_Windows_NT

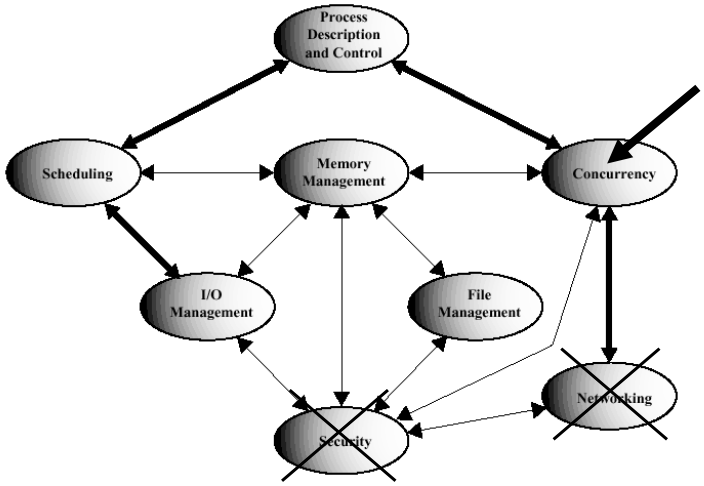
http://www.developer.com/ws/android/programming/guide-to-porting-android-applications-to-windows-8.html

Arquitectura do Android



http://www.developer.com/ws/android/programming/guide-to-porting-android-applications-to-windows-8.html

A seguir: tópicos sobre S.O.'s



PROCESSOS e *THREADS*

- Conceito de processo
- Estados de um processo
- Transições de estado
- Descrição de processos
- Estruturas de controlo de processos
- Operações sobre processos
- *Threads*

Noção de Processo

Processo
- programa em execução

Processo ≠ Programa
- programa - entidade passiva (conteúdo de um ficheiro)
  processo - entidade activa

Um processo engloba
- código + dados
- conteúdo do *program counter*, registos, *stack*, ...
- recursos

Programas e Processos

Um programa torna-se num processo através de um procedimento de carregamento (*loading*).

O ficheiro contendo o programa compilado é lido e a memória do novo processo é inicializada com o conteúdo do ficheiro
- **código do programa**
- **dados inicializados.**

O S.O. cria um novo Bloco de Controlo de Processo (*PCB - Process Control Block*)
- estrutura de dados contendo informação acerca do processo.

O processo inicia a execução no ponto de partida do programa quando o S.O. o despachar para execução.

Multiprogramação

Execução sequencial de programas ⇒
- **desperdício de recursos**
  - » **as operações de I/O são muito mais lentas do que a execução de instruções por parte da *CPU***

Solução:
- **interlaçar os cálculos com a *I/O* para aumentar a eficiência**
  - execução concorrente (multiprogramação)

Multiprogramação:
- **técnica que sobrepõe operações de I/O e de cálculo de diversos processos em execução.**

Multiprogramação

A ideia básica é permitir que múltiplos processos residam em memória ao mesmo tempo e

- **quando um processo bloqueia à espera de uma operação de *I/O* a *CPU* executa outro processo;**
- **quando este bloqueia, a *CPU* executa um 3º processo, …etc…**
- **quando a operação de *I/O*, de que o 1º processo estava à espera, termina, o S.O. marca o processo que estava bloqueado como pronto a executar.**

O processo pode ser obrigado a ceder a *CPU* antes de bloquear.

Multiprogramação sem preempção
- **Os processos decidem quando devem ceder a *CPU*.**

Multiprogramação com preempção
- **O S.O. decide quando um processo deve ceder a *CPU*.**

Uniprogramação

O S.O. permite
- **só um processo em execução**
- **só um processo algures entre o início e o fim de execução**

Multiprogramação

O S.O. permite
- **só um processo em execução**
- **múltiplos processos algures entre o início e o fim de execução**

Multiprocessamento

O S.O. (e o *hardware*) permite
- **múltiplos processos em execução**
- **múltiplos processos algures entre o início e o fim de execução.**

Exercício

Programa A	Programa B	Programa C
100 instrucões	ler 1 sector	1000 instrucões
escrever 1 sector	100 instrucões	escrever 1 sector
100 instrucões	escrever 1 sector	

Ler/Escrever 1 sector = 0.0020 segundos
executar 100 instrucões = 0.0001 segundos

Com uniprogramação e multiprogramação

- **Quanto tempo leva a executar cada programa ?**
  (considerar uma chegada quase simultânea pela ordem A,B,C)
- **Quanto tempo é que a *CPU* está inactiva ?**

Dificuldades da multiprogramação

- **Necessidade de proteger os recursos atribuídos a cada processo, nomeadamente, proteger e controlar o acesso a:**
  - **áreas de memória**
  - **certas instruções do processador**
  - **periféricos de *I/O***
- **Isto requer que o *hardware* possua certas características especiais, por exemplo**
  - **dois modos de funcionamento (utilizador e supervisor)**
  - **registos especiais usados na protecção de memória**
- **Necessidade de comunicação e sincronização entre processos interdependentes.**

Funções de administração de processos num S.O.

- **Criação e remoção de processos.**
- **Interlaçamento da execução dos processos e controlo do seu progresso garantindo o avanço da sua execução pelo sistema.**
- **Actuação por ocasião da ocorrência de situações excepcionais (erros aritméticos, ...).**
- **Alocação dos recursos de *hardware* aos processos.**
- **Fornecimento dos meios de comunicação de mensagens e sinais entre os processos.**

Processos e *Threads*

Um processo tem duas características
- **posse de recursos**
  - » **Ficheiros, memória, ... detidos pelo processo**
- **uma sequência / *thread* de execução**
  - » **Informação sobre o que é e onde está o processo (*PC*, *PSW* e outros registos)**

Os S.O.'s modernos usam o conceito de *thread* ou *lightweight process* (LWP).

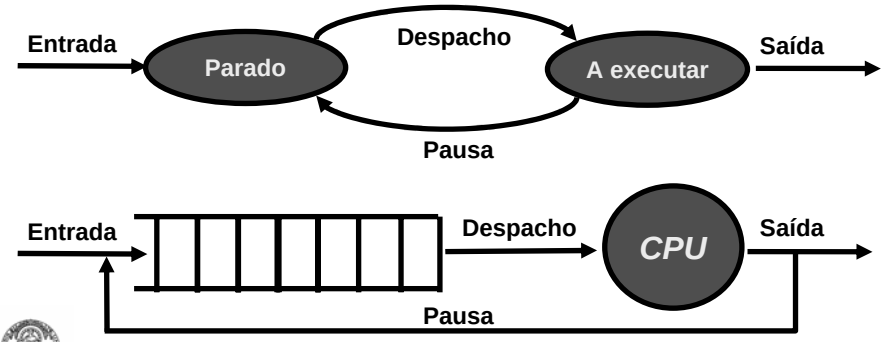
Processo / tarefa → posse de recursos
***Thread* / *LWP* → sequência de execução**

Múltiplas *threads* podem estar associadas a um processo.

Estados de um processo

À medida que um processo executa, muda de estado.

Modelo de 2 estados



Estados de um processo

Alguma informação que é necessário guardar:

- **estado actual do processo**
- **sua posição na memória**
- **lista de processos à espera de execução**

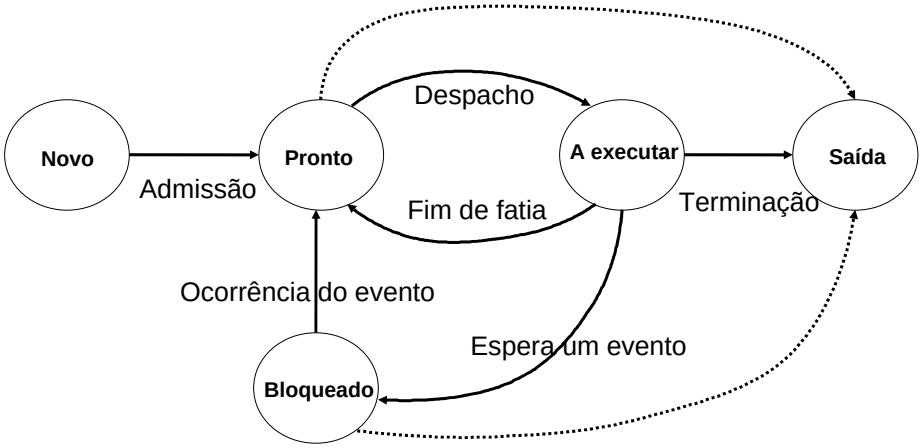
A lista de processos à espera de execução pode conter 2 tipos de processos:

- **processos prontos a correr**
- **processos bloqueados (à espera de *I/O*)**

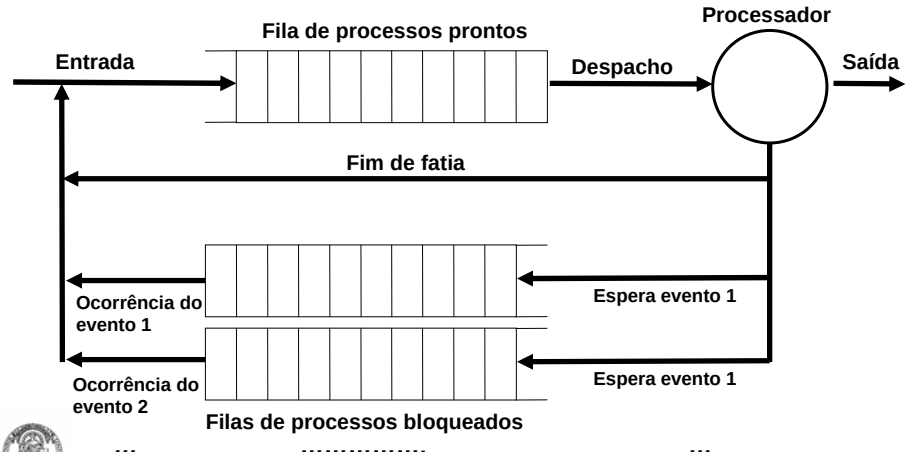
Surge assim o modelo de 5 estados.

Poderá existir
uma fila de processos prontos e uma fila de processos bloqueados
ou mesmo
uma fila de processos prontos por cada nível de prioridade
e uma fila de processos bloqueados por cada evento (dispositivo).

Modelo de 5 estados



Modelo de 5 estados



Modelo de 5 estados

Estados:

- **Novo**
  - » o processo acaba de ser definido,mas ainda não está em execução
- **Pronto**
  - » o processo está à espera que lhe seja atribuída a *CPU*
- **A executar**
  - » as instruçõers estão a ser executadas
- **Bloqueado**
  - » o processo está à espera da ocorrência de um acontecimento
- **Terminado**
  - » o processo terminou a execução (normalmente ou abortou)

Modelo de 5 estados

Transições de estado:

- **Novo → Pronto**
  - » q.do um processo é criado e inicializado
- **Pronto → A Executar**
  - » q.do a um processo é atribuída a *CPU*
- **A Executar → Pronto**
  - » q.do uma fatia de tempo expira
    (multiprogramação com preempção)
- **A Executar → Bloqueado**
  - » q.do um processo bloqueia à espera de um acontecimento
    (operação de *I/O*, acesso a ficheiro, serviço do S.O. ,
    comunicação c/outro processo, …)
- **A Executar → Terminado**
  - » q.do um processo termina a execução
- **Bloqueado → Pronto**
  - » q.do o acontecimento ocorre
- **Pronto, Bloqueado → Terminado**
  - » q.do o processo é forçado a terminar por outro processo

Estados de um processo

Num sistema sem memória virtual

- **cada processo a executar tem de estar totalmente carregado em memória principal**
- **todos os processos, de todas as filas de espera, têm de estar em memória principal**

Problema

- **como o processador é muito mais rápido que a *I/O* será comum acontecer que todos os processos em memória estejam à espera de *I/O*.**

Solução

- ***Swapping* - deslocar parte de (/ todo) um processo para o disco.**

Swapping

Quando nenhum dos processos em mem. principal está pronto o S.O. desloca um dos processos bloqueados para o disco e coloca-o numa fila de processos suspensos (modelo de 6 estados).

A activação do processo (Suspenso → Pronto) só deve ser feita quando acontecer o evento que deu origem a que o processo fosse suspenso
⇒ preferível dividir o estado Suspenso em 2 estados: Bloqueado Suspenso e Pronto Suspenso

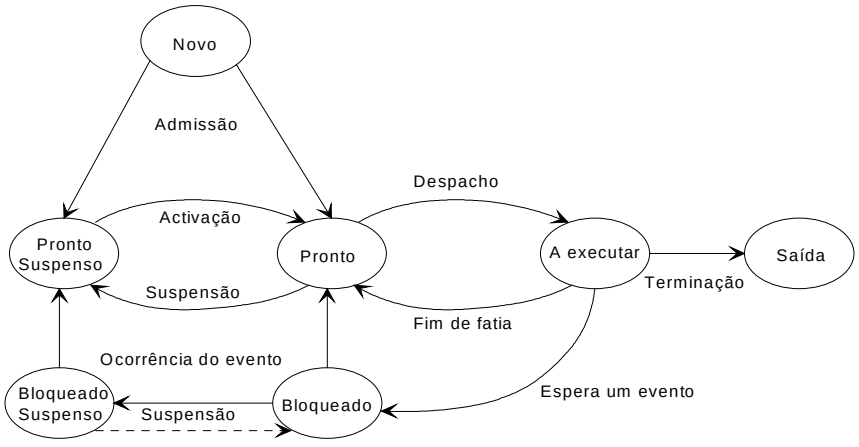
(modelo de 7 estados)

Processos suspensos

Razões para a suspensão de um processo:

- ***swapping***
- **pedido interactivo do utilizador**
- **pedido do processo-pai**
- **temporização**
  (ex.: processo executado periodicamente)
- **outra razão do S.O.**
  (ex.: processo que corre em *background*)

Modelo de 7 estados



Modelo de 7 estados

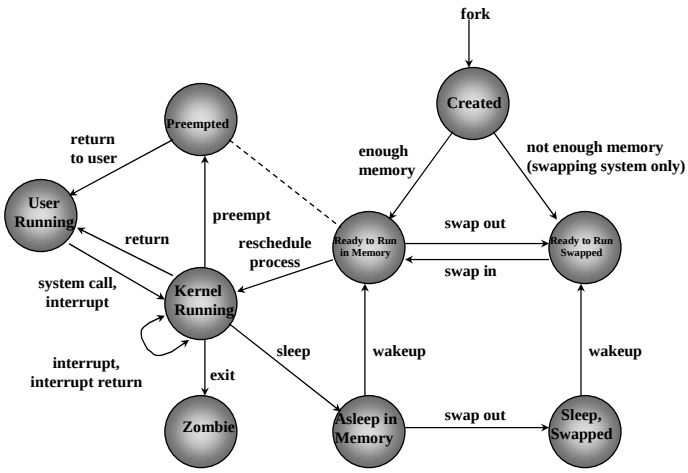
Transições de estado (algumas notas):

- **Pronto → Pronto Suspenso**
  - » em geral será pouco comum;
  - » será preferível suspender um processo bloqueado;
  - » mas pode acontecer p/libertar memória.
- **Bloqueado Suspenso → Bloqueado**
  - » q.do o processo BS tem maior prioridade do que qualquer um dos que está no estado Pronto Suspenso e o SO presume que o motivo do bloqueio desaparecerá em breve
- **A Executar → Pronto, Suspenso**
  - » o S.O. recorre à preempção (retirar a *CPU*) de um processo quando um processo de prioridade mais elevada fica Pronto.
- **Podem acontecer várias transições de diversos estados para Terminação.**

Preempção

- **acto de retirar o processador a um processo sem ser por ele estar bloqueado ou ter terminado.**

Estados dos Processos no UNIX System V



Estados dos Processos no UNIX System V

- 9 estados
- 2 estados "A Executar"
  - modo núcleo / supervisor
    - » como resultado de :
      - chamada ao sistema
      - interrupção do relógio
      - interrupção de *I/O*
  - modo utilizador
- Os estados *Ready to Run in Memory* e *Preempted* são essencialmente o mesmo. Existe uma única fila de espera para ambos.
- A transição *sleep* corresponde a bloqueamento
- A preempção só pode ocorrer na ocasião em que um processo que está a executar em modo supervisor (*kernel running*) vai passar a executar em modo utilizador (*user running).*

Estados dos Processos no UNIX System V

Estado *asleep*

- O processo está à espera de um determinado acontecimento (operação de *I/O*, à espera num semáforo, ...)

Estado *zombie*

- Um processo que termina não pode deixar o sistema até que o seu processo-pai aceite o seu código de retorno.
- Se o processo-pai estiver "vivo" mas nunca executar um *wait( )* o código de retorno do processo-filho nunca será aceite e este ficará *zombie*.
- Um processo *zombie* não tem código, nem dados, nem *stack*, mas continua a constar da tabela de processos.

Descrição de Processos

Para gerir e controlar os processos o S.O. deve saber:

- **onde cada processo está colocado**
  - » **bloco contíguo de memória ou**
  - » **blocos separados (paginação, segmentação) podendo alguns não estar em memória (mem. virtual)**
- **os atributos do processo**

O S.O. mantém uma tabela, a tabela de processos, com uma entrada por cada processo, contendo toda a informação relevante para a gestão dos processos.

A informação relativa a cada processo é mantida no respectivo Bloco de Controlo do Processo.

Os processos da tabela de processos poderão estar organizados em várias listas consoante o seu estado (pronto, a executar, ...).

Bloco de Controlo do Processo (*Process Control Block - PCB*)

Estrutura de dados contendo informação associada ao processo.

É no PCB que é guardado o estado de um processo por ocasião da comutação de processos.

Inclui

- **identificação do processo (=*Process ID* / *PID*, do processo, do pai)**
- **estado do processo (pronto, bloqueado, suspenso, ...)**
- **registos do processo (*program counter*, flags, registos da *CPU*, ...)**
- **informação de escalonamento da *CPU* (prioridade, ...)**
- **informação de gestão da memória (lim.s da zona de memória,...)**
- **informação de contabilidade (tempo de *CPU* gasto,...)**
- **informação de estado da *I/O* (fich.s abertos, operaç.s pendentes, ...)**

Estruturas de controlo do S.O.

Outras estruturas de dados necessárias para a gestão dos processos:

Tabelas de memória
- **alocação da memória principal e secundária**
- **protecções de acesso**
- **informação para a gestão de memória virtual**

Tabelas de *I/O*
- **estado das operações de *I/O***
- **localização dos dados de origem e de destino**

Tabelas de ficheiros
- **ficheiros existentes**
- **posição em memória secundária**
- **estado actual**
- **outros atributos**

Gestão de processos

Operações típicas do núcleo *(kernel):*
- **criação e terminação de processos**
- **escalonamento e despacho**
  - » ***scheduller* - implementa a política global de gestão da *CPU* (selecciona o próximo processo a executar)**
  - » ***dispatcher* - dá o controlo da *CPU* ao processo seleccionado ⇒**
    - **comutar de contexto**
    - **comutar para modo utilizador**
    - **saltar para o endereço adequado do programa**
- **sincronização e suporte para intercomunicação entre processos**
- **gestão dos *PCB*'s**

Criação e Terminação de processos

Criação de um processo (estado Novo)

- O sistema operativo
  - » cria as estruturas de dados necessárias p/gerir o processo
  - » aloca o espaço de endereçamento a ser usado pelo processo

Terminação de um processo

- O processo é retirado em 2 etapas
  - » 1 – É-lhe retirado o processador (normalmente ou abortou).
  - » 2 – A informação associada ao processo é apagada
    - Esta informação é mantida no sistema depois de o processador lhe ter sido retirado para que outros processos possam extrair informação relativa ao processo que acabou (ex.: tempo de *CPU*, recursos usados)

Criação e Terminação de processos

Razões para a criação de processos:

- novo "batch job"
- "log on" interactivo
- criado pelo SO p/fornecer um serviço (ex: impressão)
- criado por outro processo

Razões para a terminação de um processo (*):

- completação normal
- tempo limite excedido
- memória indisponível
- violação dos limites de memória
- erro de protecção
- erro aritmético
- tempo de espera excedido
- falha de *I/O*
- instrução inválida
- instrução priveligiada
- intervenção do operador ou do SO
- terminação do processo-pai
- pedido do processo-pai
- ...

(*) - algumas podem não ser aplicáveis em alguns S.O.'s

Criação de um processo

Etapas:

- **Atribuir um identificador ao processo**
- **Reservar espaço para o processo**
  - » **para todos os elementos da imagem do processo**
    - **programa + dados + *stack* + *PCB***
- **Inicializar o *PCB***
- **Colocar o processo na lista de processos Prontos**
- **Criar / actualizar outras estruturas de dados (ex.: dados de contabilidade do sistema)**

Comutação de Contexto

Sempre que um processo bloqueia e outro processo passa a ser executado ocorre uma comutação de contexto:

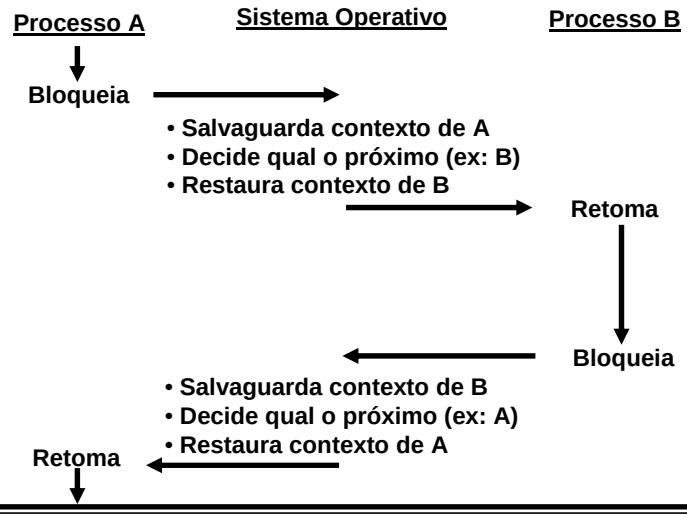
- **salvaguarda do estado actual do processo (registos, …)**
- **restauro do estado, previamente guardado, do próximo processo a executar**
- **passagem do controlo do processador para o novo processo**

Comutação de contexto ⇒ perda de tempo

Redução do tempo de comutação de contexto

- **máquinas onde existe mais do que um conjunto de registos;**
- **utilização de *threads.***

Comutação de contexto



Criação de processos em UNIX

fork( ) **cria um novo processo (processo-filho)**
que obtém uma cópia de toda a memória do processo-pai
e partilha os ficheiros que o processo-pai estiver a usar.

Os 2 processos (pai e filho) executam concorrentemente.

Não é carregado nenhum programa novo.
Os 2 processos correm o mesmo programa.
O processo divide-se em 2 cópias, ambas resultantes da chamada a *fork( )*, com todo o estado anterior em comum.

Existe um conjunto de chamadas *exec( )*,
(de facto *execXX( )*, em que *XX* depende da chamada)
para fazer o carregamento de um programa novo.
O código do programa que invocar *exec()* é substituído pelo código do programa que for indicado como argumento de *exec()*.

(ver apontamentos sobre API do UNIX)

Criação de processos em UNIX (cont.)

fork( ) **cria simultaneamente**
- **um novo processo**
- **um novo espaço de endereçamento**

Espaço de endereçamento
- **a memória em que um processo é executado**

É possível distinguir o processo-pai do processo-filho testando o valor retornado por *fork( )*:

= 0 ⇒ é o processo-filho
> 0 ⇒ é o processo-pai e
o valor retornado é o identificador do filho

= -1 ⇒ a chamada falhou

Criação de processos em UNIX (cont.)

EXEMPLO:

```
main()
{
  int pid;
  if ((pid = fork())==-1)
    return(-1);
  else
   if (pid==0)
    {
      printf("Eu sou o filho !\n");
      exit(1);
    }
  printf("Eu sou o pai\n");
}
```

Qual o resultado deste programa ?

E se a instrução `exit(1)` **fosse retirada ?**

Criação de processos em UNIX (cont.)

EXERCÍCIO: Qual o resultado do seguinte programa ?

```
 main()
{ printf("1\n");
  printf("2\n");
  fork();
  printf("3\n");
  printf("4\n");
}
```

Resposta:

Tanto pode ser →	como →
1	1
2	2
3	3
4	3
3	4
4	4

Tudo depende de como a sequência dos 2 programas for interlaçada, devido à multiprogramação.

Criação de processos em UNIX (cont.)

Para criar um novo processo, executando um programa diferente usa-se uma função *execXX( )*.

Uma função *execXX( )* é uma chamada à biblioteca do C que por sua vez chama uma rotina de sistema, *execve( )*.

EXEMPLO:

```
...
switch (pid=fork()) {
 case 0  : /* Este é o filho */
           execl("/bin/ls","ls","-l",NULL);
           /* se chegar aqui, o exec falhou */
           exit(1);
 case -1 : /* o fork() falhou */
           exit(2);
 default : /* Este é o pai */
           /* executa concorrentemente com o filho */
           ...
}
...
```

Threads

Um(a) *thread* é
um processo "leve" (*Lightweight Process),*
com um estado reduzido.

A redução de estado é conseguida
fazendo com que um grupo de *threads* (do mesmo processo)
partilhe recursos como memória, ficheiros, dispositivos de *I/O*, …

Nos sistemas baseados em *threads*,

- **um processo pode ter vários *threads;***
- **os *threads* tomam o lugar dos processos como a mais pequena unidade de escalonamento ;**
- **se a implementação for *kernel-level* enquanto um *thread* está bloqueado, outro pode estar a executar**
- **o processo serve como o ambiente p/ a execução dos *threads.***

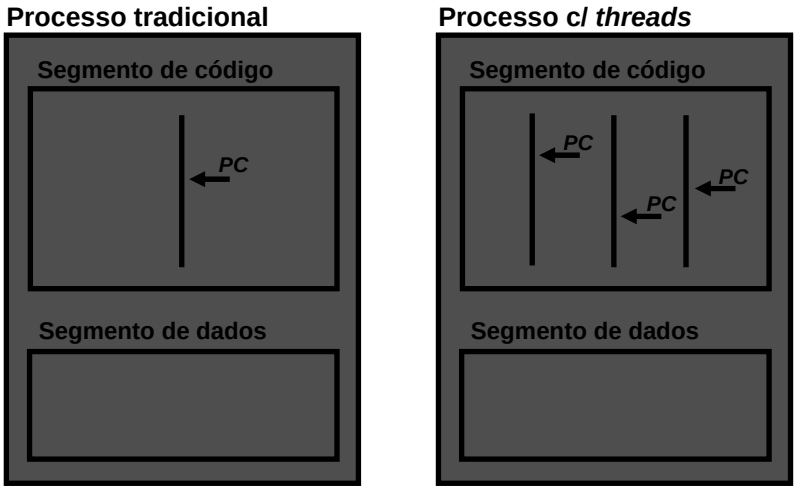
Threads

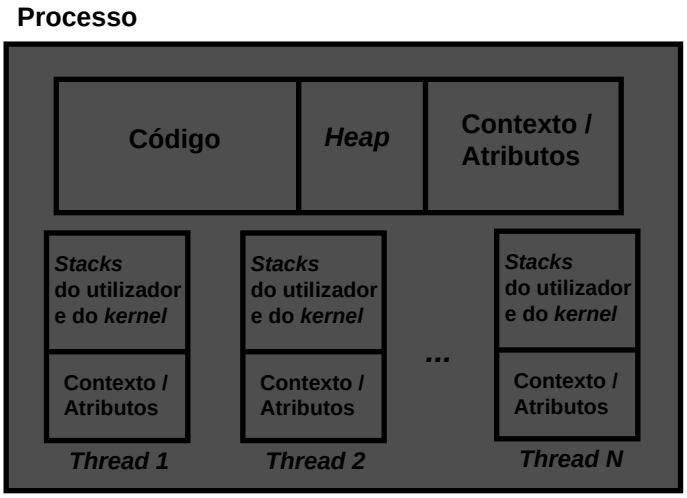
Processo tradicional (*Heavyweight Process*)
⇔ processo c/um único *thread.*

Um *thread* partilha c/ os outros *threads* do mesmo processo:

- **a secção de código**
- **a secção de dados**
- **os recursos do S.O.**

⇒ a comutação entre *threads* do mesmo processo
é muito menos pesada
do que entre processos tradicionais


Processo tradicional
Segmento de código
PC
Segmento de dados
Processo c/ threads
Segmento de código
PC
PC
PC
Segmento de dados


Processo
Código
Heap
Contexto / Atributos
Stacks do utilizador e do kernel
Contexto / Atributos
Thread 1
Stacks do utilizador e do kernel
Contexto / Atributos
Thread 2
...
Stacks do utilizador e do kernel
Contexto / Atributos
Thread N

Processos Tradicionais *vs. Threads*

Threads

Semelhanças c/ os processos

- **têm um estado (pronto, a executar, bloqueado, …)**
- **partilham a *CPU* entre si**
  (em cada instante apenas um *thread* está a executar, num sistema uniprocessador)
- **cada *thread* de um processo executa sequencialmente**
- **cada *thread* tem associado**
  - » **um *program counter***
  - » **um *stack pointer***
  - » **um *Thread Control Block***
    (c/ conteúdo dos registos da *CPU*, estado do *thread*, prioridade,…)
- **um *thread* pode criar *threads*-filho**

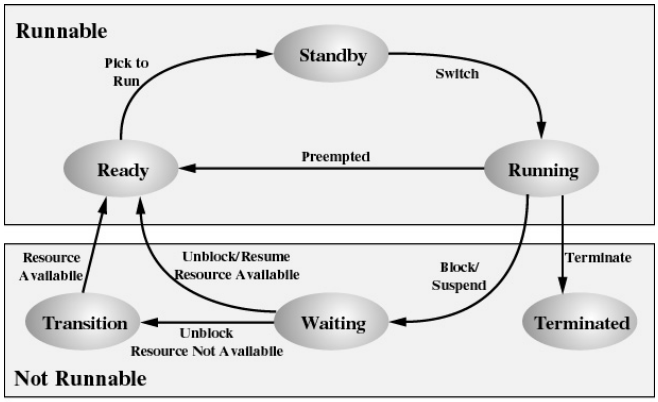
Processos Tradicionais *vs. Threads*

Threads

Algumas características importantes:

- **Não existe protecção entre *threads* do mesmo processo**
  - » **desnecessária (!) ;**
    os *threads* são concebidos para cooperarem numa tarefa comum
- **Qualquer alteração das variáveis globais de um processo é visível em todos os seus *threads***
  - » **Em alguns SOs é possível um *thread* criar variáveis globais cujo conteúdo depende do *thread* que refere essa variável*;***
    estas variáveis não são acedidas directamente,
    mas através de chamadas a funções específicas de acesso.
    Esta facilidade é conhecida por *TLS-Thread Local Storage*.
- **Suspensão (*swapping*) de um processo ⇒ suspensão dos seus *threads***
- **Terminação de um processo ⇒ terminação dos seus *threads***

Estados dos *threads* no Windows 2000



(Stallings)

Estados dos *threads* no Windows 2000

- **Ready**
  - **o *thread* pode ser escalonado para execução**
- **Standby**
  - **o *thread* foi seleccionado p/executar a seguir;**
    espera neste estado até o processador estar disponível
    (que o *thread* a executar bloqueie ou a sua fatia de tempo expire);
    se a prioridade deste *thread* for superior à do *thread* que está a correr
    este pode sofrer preempção
- **Running**
  - **a executar até sofrer preempção, expirar a sua fatia de tempo,**
    bloquear ou terminar; nos 2 primeiros casos volta p/ o estado Ready
- **Waiting**
  - **bloqueado num evento (ex: *I/O*) ou**
  - **à espera de um acontecimento de sincronização ou**
  - **recebeu ordem de suspensão**
- **Transition**
  - **está pronto a correr mas os recursos ainda não estão disponíveis**
    (ex: a *stack* do *thread* foi colocada em disco, em consequência da paginação)
- **Terminated**
  - **terminou normalmente ou foi terminado p/outro *thread* ou o proc.-pai terminou**

Threads

Algumas vantagens de utilização:

- **Economia e velocidade**
  - » **menos tempo p/ criar, comutar e terminar**
- **Aumento da rapidez de resposta percebida pelo utilizador**
  - » **ex: um *thread* lê comandos, outro executa-os; permite ler o próximo comando enq. o anterior é executado**
- **Eficiência de comunicação**
  - » **recorrendo à memória partilhada não é necessário invocar o *kernel***
- **Utilização de arquitecturas multiprocessador**
  - » **cada *thread* pode executar em paralelo num processador diferente**

Dificuldade

- **garantir a sincronização entre os *threads* quando manipulam as mesmas variáveis**

User-level e *Kernel-level Threads*

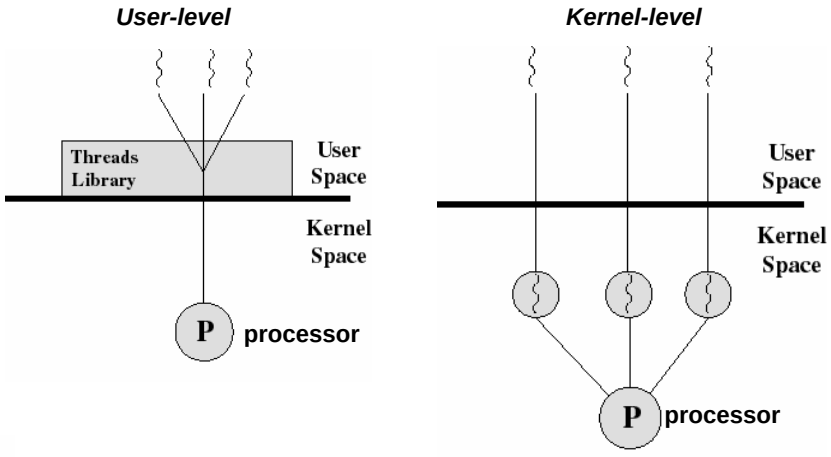
User-level threads

- **O *kernel* "não sabe" da existência de *threads*.**
- **Toda a gestão dos *threads* é feita pela aplicação usando uma biblioteca de funções apropriada.**
- **A comutação entre *threads* não requer privilégios de *kernel mode*.**
- **O escalonamento depende da aplicação.**

Kernel-level threads

- **Toda a gestão dos *threads* é feita pelo *kernel*.**
- **Não existe uma biblioteca de *threads* mas uma *API* de *threads*.**
- **A comutação entre *threads* requer a intervenção do *kernel*.**
- **O escalonamento é feito sobre os *threads*.**
- **O *kernel* mantém informação de escalonamento sobre os processos e sobre os *threads*.**

User-level e *Kernel-level Threads*



User-level Threads

Vantagens

- **A comutação entre *threads* não envolve o *Kernel*: não implica comutação para *Kernel mode*.**
- **O escalonamento pode ser específico de uma aplicação: possível escolher o algoritmo mais adequado.**
- **Podem ser usados em qualquer S.O. . Basta que se disponha de uma biblioteca adequada.**

Inconvenientes

- **Quando uma chamada ao sistema implica um bloqueio (ex:*I/O*) todos os *threads* do processo ficam bloqueados**
- **O *kernel* só pode atribuir processadores aos processos. Dois *threads* do mesmo processo não podem correr em simultâneo em dois processadores.**

Kernel-level Threads

Vantagens

- O *kernel* pode escalonar os diversos *threads* de um mesmo processo para executarem em diferentes processadores.
- O bloqueamento é feito ao nível dos *threads*.
  Quando um *thread* bloqueia, outros *threads* do mesmo processo podem continuar a executar.
- As rotinas do *kernel* podem ser *multithreaded*.

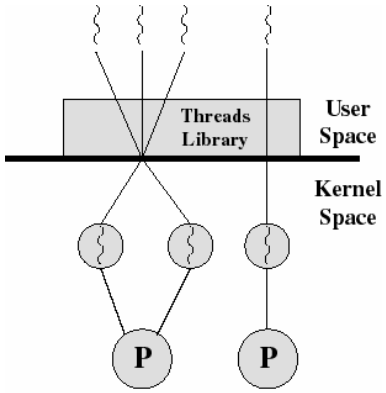
Inconvenientes

- A comutação entre *threads* do mesmo processo envolve o *kernel*.
  (2 comutações: *user mode* →*kernel mode* e *kernel mode* → *user mode*)
- Isto resulta numa comutação mais lenta.

Sistemas operativos: Windows NT/2000/XP, Linux, Solaris

Aproximação mista

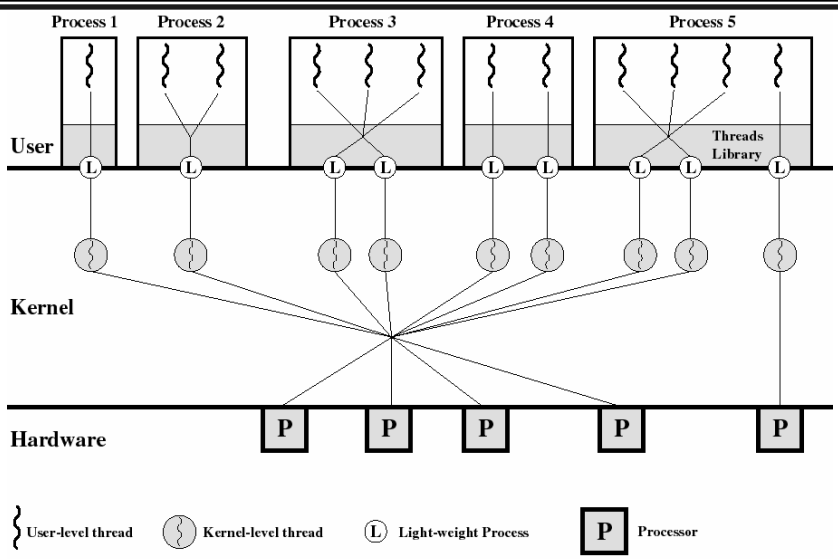
- A criação de *threads* é feita no espaço do utilizador.
- A maior parte do escalonamento e sincronização também são feitos no espaço do utilizador.
- As *User-level Threads* são mapeadas em *Kernel-level Threads* (nº de KLTs <= nº de ULTs).
- O utilizador pode ajustar o número de *Kernel-level Threads.*
- Permite combinar as vantagens de *ULT*s e *KLT*s.
- Exemplo: Solaris 2.x



Solaris 2.x

- **Um processo inclui o espaço de endereçamento do utilizador, a *stack* e o *PCB*.**
- ***User-level threads* (*threads library*)**
  - » **invisível para o S.O.**
- ***Kernel threads***
  - » **a unidade sujeita a despacho num processador**
- ***Lightweight processes* (*LWP*)**
  - » **cada *LWP* suporta um ou mais *ULT*s e mapeia-os exactamente num *KLT* (fig. seguinte).**

Solaris



Interface Pthreads (*Posix threads*)

Funções (~60) :

- **Criar e esperar por *threads***
  - **pthread_create**
  - **pthread_join**
- **Terminar *threads***
  - **pthread_cancel**
  - **pthread_exit**
  - **exit() - termina todos os *threads*;**
    return - termina o *thread* corrente
- **Determinar a ID de um *thread***
  - **pthread_self**
- **Sincronizar o acesso a variáveis partilhadas**
  - **pthread_mutex_init**
  - **pthread_mutex_lock, pthread_mutex_unlock**
  - **...**

(ver apontamentos sobre API do UNIX)

Posix *threads*

EXEMPLO (criação de um *thread*) :

```
#include <stdio.h>
#include <pthread.h>
...
char message[] = "Hello world !";

void *thread_function(void *arg) {
   printf(" Thread function is running. Argument is %s\n",(char *) arg);
   ...
   return NULL;
}

int main(void) {
   int res;
   pthread_t thread_id;


   res = pthread_create(&thread_id, NULL, thread_function, (void *) message);
   if (res != 0){ /* ERROR */ }
   ...
}
```

compilação:

```
$ cc thrprog.c -o thrprog -lpthread
```

(ver apontamentos sobre API do UNIX)

Win32 *API*

Primitivas da Win32 *API* :

- ***CreateProcess***
- ***CreateThread***
- ***SuspendThread***
- ***ResumeThread***
- ***ExitThread***
- ***TerminateThread***
- ***…***

CreateProcess( )

- **cria um novo espaço de endereçamento a partir de um ficheiro executável e cria um único *thread* executando no ponto de entrada do programa**

CreateThread( )

- **cria um novo *thread* dentro do espaço de endereçamento do *thread* original**

ESCALONAMENTO DO PROCESSADOR

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

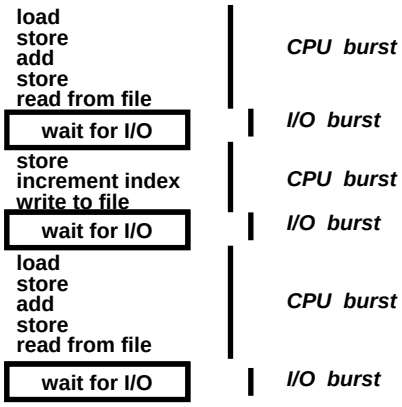
Conceitos básicos

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 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.**

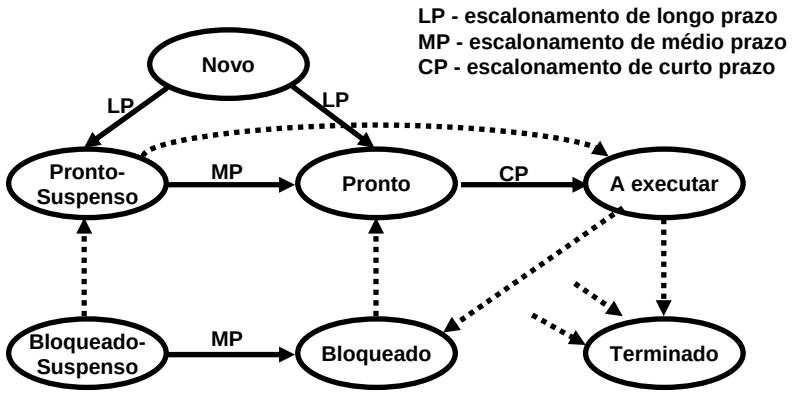
Escalonamento de curto prazo
- **Determina qual o processo a ser executado proveniente da fila de processos prontos.**

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:

$\alpha = 0.8 \Rightarrow$

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

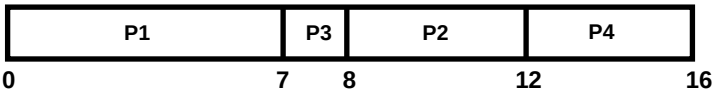
α **elevado** $\Rightarrow$ **dar muito peso às observações mais recentes**

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

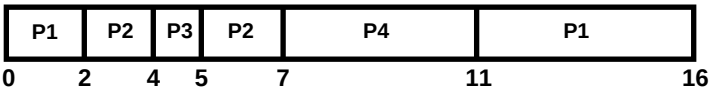
Exemplo (*SJF e SRTF*)

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

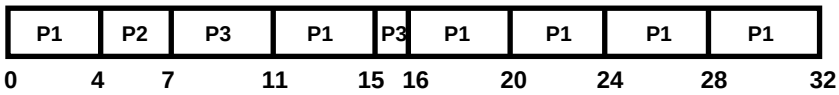
- Atribui-se a cada processo uma fatia de tempo (*quantum*) para executar.
- Um processo permanece no estado de execução até efectuar uma operação de I/O, ou ser interrompido por um temporizador que periodicamente (no fim da fatia de tempo atribuída ao processo) interrompe o processo que estiver em execução, 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 ou *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
(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

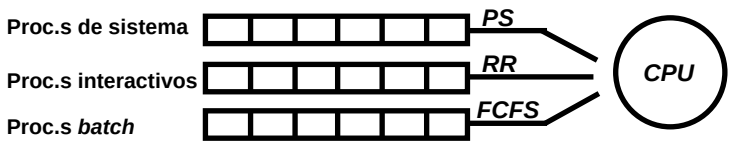
- **Sistema baseado em prioridades mas com várias filas de processos prontos.**
- **Cada fila tem o seu algoritmo de escalonamento.**
- **As filas são ordenadas por ordem decrescente de prioridade.**
- **Executa-se um processo de uma fila apenas quando não há processos prontos nas filas de maior prioridade.**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

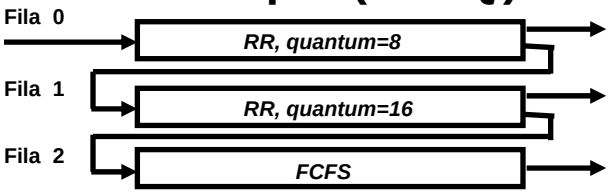
- **Sistema *Multilevel Queue* com regras para movimentar os processos entre as várias filas.**
- **Os processos mudam de fila de acordo com o seu comportamento anterior:**
  - **Os processos são admitidos na fila de maior prioridade**
  - **Um processo que usa demasiado tempo de *CPU* é despromovido 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 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)
    - » 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.

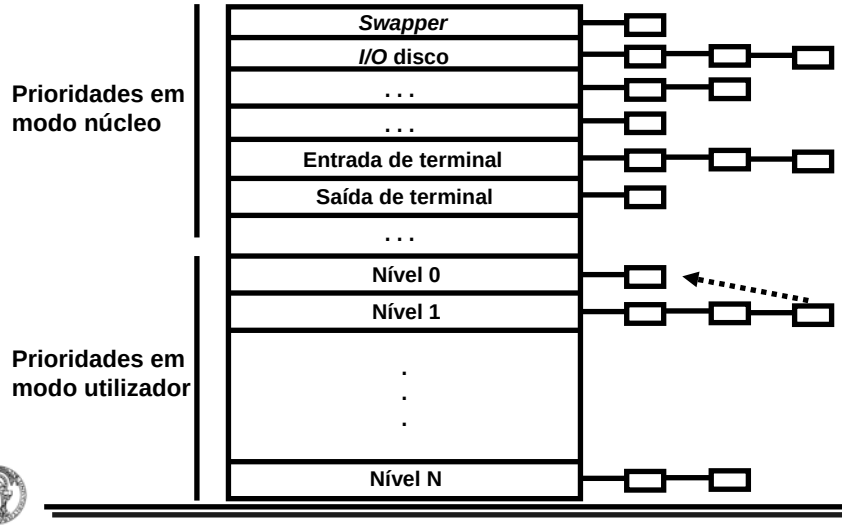
Valores da prioridade

- 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.

SINCRONIZAÇÃO DE PROCESSOS

- O problema das secções críticas
- Soluções baseadas em *software*
- Soluções basedas em *hardware*
- O mecanismo de sincronização básico: semáforos
- Mecanismos de sincronização mais sofisticados: monitores, passagem de mensagens, regiões críticas
- Problemas clássicos de sincronização

Execução concorrente

- Execução concorrente
  - execução cooperante ou não-cooperante logicamente ao mesmo tempo (ex: multiprogramação)
- Execução em paralelo
  - execução cooperante ou não-cooperante fisicamente ao mesmo tempo (ex.: multiprocessamento)
- A execução concorrente pode ocorrer em
  - sistemas uniprocessador
    - interlaçamento de execução (multiprogramação)
  - sistemas multiprocessador
    - interlaçamento e sobreposição de execução (multiprogramação c/ multiprocessamento)
- Os processos concorrentes precisam frequentemente de partilhar dados / recursos.
- O acesso concorrente a dados partilhados pode resultar em inconsistência desses dados, se o acesso não ocorrer de forma controlada.
- Execução concorrente ⇒
  - mecanismos de comunicação entre processos
  - sincronização entre as suas acções

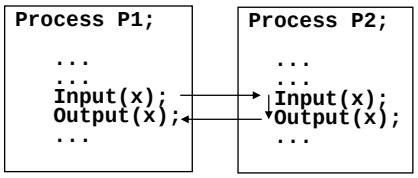
Necessidade de Sincronização - Exemplo 1

- Os processos P1 e P2 executam o seguinte código tendo acesso à mesma variável x.

```
Process P1;
 Begin
   ...
   Input(x);
   Output(x);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Input(x);
   Output(x);
   ...
 End;
```

- Os processos podem ser interrompidos em qualquer ponto.
- Se
  P1 for interrompido após a entrada de dados e
  P2 executar inteiramente
  então
  o carácter ecoado por P1 será o que foi lido por P2 !!!



Exemplo 2

- Dois processos
  - um lê caracteres do teclado
  - outro ecoa os caracteres lidos para o écran
  - os caracteres são inseridos num *buffer* circular
  - uma variável partilhada, `Count`, indica o nº de caracteres contidos no *buffer*

```
Process Keyboard;
 Begin
   ...
   Count:=Count+1;
   ...
 End;
```

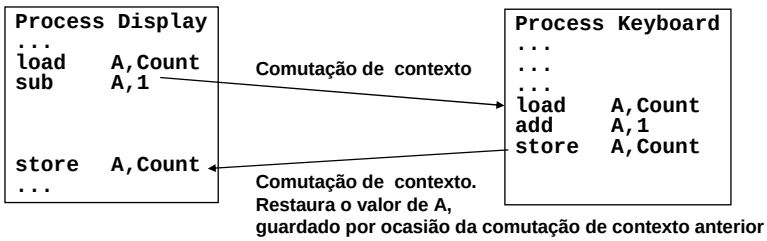
```
Process Display;
 Begin
   ...
   Count:=Count-1;
   ...
 End;
```

- O que parece uma operação única (actualização da variável `count`) pode ser traduzido numa série de instruções-máquina.
  Ex.:

```
...
Count:=Count+1;   ──►   load   A, Count
...                     add    A,1
                        store  A, Count
```

Exemplo 2 (cont.)

- A execução concorrente pode causar um problema (*race condition*):



- Admitindo que o valor inicial de `Count` era 5, qual será o valor final ?
  E qual deveria ser ?

- ***Race condition***
  - situação em que
    vários processos acedem e manipulam os mesmos dados, concorrentemente,
    e o resultado da execução depende da ordem em que se dá o acesso a esses dados.

Exemplo 3

```
struct {
   // nome, B.I., etc...;
   int saldo; } Conta;

int levantar (Conta *conta, int valor) {
   if (conta->saldo >= valor)
      conta->saldo = conta->saldo - valor;
   else
      valor = -1; // ERRO
   return valor;
}
```

Qual é o problema ?

Exemplo 3 (cont.)

```
struct {
   // nome, B.I., etc...;
   int saldo; } Conta;

int levantar (Conta *conta, int valor) {
   if (conta->saldo >= valor)
      conta->saldo = conta->saldo - valor;
   else
      valor = -1; // ERRO
   return valor;
}
```

```
Process P1;

   ...
   int levantar (...) {
      if (...) {
         conta->saldo =
            conta->saldo - valor;
         ...
```

```
Process P2;

   ...
   int levantar (...) {
      if (...) {
      ...

      conta->saldo =
       conta->saldo - valor;
      ...
```


Exemplo 3 (cont.)

```
struct {
   // nome, B.I., etc...;
   int saldo; } Conta;

int levantar (Conta *conta, int valor) {
   if (conta->saldo >= valor)
      conta->saldo = conta->saldo - valor;
   else
      valor = -1; // ERRO
   return valor;
}
```

```
...
mov AX,saldo
mv BX,valor
sub AX,BX
mov Saldo,AX
...
```

```
Process P1;

   ...
   int levantar (...) {
      if (...) {
         conta->saldo =
            conta->saldo - valor;
         ...
```

```
Process P2;

   ...
   int levantar (...) {
      if (...) {
      ...

      conta->saldo =
       conta->saldo - valor;
      ...
```

O problema das secções críticas

- Quando um processo executa código que manipula dados / recursos partilhados, diz-se que o processo está na sua secção crítica (para esses dados / recursos).
- A execução de secções críticas deve ser mutuamente exclusiva: em cada instante, apenas um processo poderá estar a executar na sua secção crítica (mesmo com múltiplos *CPUs*).
- Por isso, cada processo deve pedir autorização para entrar na sua secção crítica (SC).
- A secção de código que implementa este pedido é chamada a secção de entrada (SE).
- A SC é seguida por uma secção de saída (SS).
- O resto do código constitui a designada secção restante (SR).
- O problema das secções críticas é conceber um protocolo que os processos possam usar para que a sua acção não dependa da ordem pela qual a sua execução é interlaçada (mesmo com múltiplos *CPUs*).

Pressupostos p/ a análise de soluções

- Cada processo executa com velocidade não nula mas não é feita qualquer suposição acerca da velocidade relativa dos processos.
- A estrutura geral de uma secção crítica é:

```
Secção de entrada;
   Secção crítica;
Secção de saída;
   Secção restante
```

- Podem existir vários *CPU*'s mas o *hardware* de memória impede o acesso simultâneo à mesma posição de memória.
- Não são feitos pressupostos acerca da ordem de interlaçamento da execução.
- Nas soluções, é necessário especificar as secções de entrada e de saída.

Requisitos que uma solução do problema das SCs deve satisfazer

- **Só um processo de cada vez pode entrar na secção crítica (exclusão mútua).**
- **Um processo a executar numa secção não-crítica não pode impedir outros processos de entrar na secção crítica (progresso).**
- **Um processo que peça para entrar numa secção crítica não deve ficar à espera indefinidamente (espera limitada).**

- **Não são feitos pressupostos acerca da velocidade relativa dos processadores (ou do seu número)**
- **Supõe-se que um processo permanece numa secção crítica durante um tempo finito.**

Tipos de soluções

- **Soluções baseadas em *software* (do utilizador)**
  - **Código escrito pelo programador dos processos.**
  - **Baseadas em algoritmos cuja correcção não se baseia em nenhum pressuposto além dos anteriores.**
- **Soluções baseadas em *hardware***
  - **Baseadas em instruções-máquina especiais.**
- **Soluções baseadas em serviços do Sistema Operativo / Linguagens**
  - **Baseadas em funções e estruturas de dados, fornecidas pelo S.O..**
    - **semáforos**
    - **monitores**

Soluções baseadas em *software*

- Vamos analisar a evolução de algumas tentativas para resolver o problema (escrita do código da SE e SS).
- Admite-se que
  - Os processos podem partilhar algumas variáveis globais,para sincronizar as suas acções.
  - A operação de leitura (ou de escrita) da memória é atómica.
- Consideraremos primeiro o caso de 2 processos
  - O algoritmo 1 e o algoritmo 2 são incorrectos.
  - O algoritmo 3 (algoritmo de Peterson) é correcto.
- Apresentaremos uma solução mais geral, para *n* processos
  - O algoritmo da padaria (*bakery algorithm*)

Notação:

- Começamos com 2 processos, P0 e P1.
- Ao falar do processo Pi, Pj representa sempre o outro processo ($i \neq j$)

Algoritmo 1

Variáveis partilhadas:

```
Turn: 0..1; { Turn = i significa que o proc. Pi pode entrar na SC
              caso contrário não pode - Valor inicial: 0 ou 1 }
```

```
Process P0;                    Process P1;
 ...                            ...
 Repeat                         Repeat
  While Turn <> 0 do;            While Turn <> 1 do;
  { SecçãoCrítica }              { SecçãoCrítica }
  Turn:=1;                       Turn:=0;
  { SecçãoRestante };            { SecçãoRestante }
 Forever;                       Forever;
 ...                            ...
```

Análise do algoritmo:

- Garante a exclusão mútua das secções críticas:
  - Só um processo pode estar na sua secção crítica; Pi está em espera activa (*busy waiting*) se Pj estiver na SC.
- Não garante o progresso:
  - A execução das secções críticas é feita de forma estritamente alternada
    - Se Pi quiser entrar 2 vezes consecutivas na secção crítica não pode.

Algoritmo 2

```
Variáveis partilhadas:
 Flag: Array[0..1] of Boolean; { Flag[I]=True significa que
                                 o proc. Pi está pronto a entrar na SC }
Inicialmente
 Flag[0] = Flag[1] = false;

Process Pi;
 ...
  Flag[i] = True;     { Pi está pronto a entrar na SC}
  While Flag[j] do;   { Se Pj tb. estiver pronto, espera}
  { SecçãoCrítica }
  Flag[i] = False;    { Permite que P0 entre}
 ...
```

Análise do algoritmo:

- Garante a exclusão mútua das secções críticas.
- Não garante o progresso:
  - Se for executada a sequência
    - `t0 : Flag[0] = true;`
    - `t1 : Flag[1] = true;`

    ambos os processos ficarão eternamente à espera de entrar na SC (situação de *deadlock*)

Algoritmo 3

(algoritmo de Peterson)

```
Variáveis partilhadas:
 Turn: 0..1;
 Flag: Array[0..1] of Boolean;  // Flag[I]=True significa que
                                // o proc. Pi está pronto a entrar na SC

Inicialmente
 Turn = qualquer valor, 0 ou 1;
 Flag[0] = Flag[1] = False;


Process Pi;
 ...
  Flag[i] := True;                 // Pi está pronto a entrar na SC
  Turn := j;                       // mas dá a vez a Pj, se ele precisar
  While Flag[j] And (Turn=j) do;
  { SecçãoCrítica }
  Flag[i] = False;                 //Permite que Pj entre
 ...
```

Algoritmo 3
(algoritmo de Peterson)

Análise do algoritmo:

- **Garante a exclusão mútua das secções críticas**
  - **Um processo (ex: P1) só entra na SC se**
    - **o outro não quiser entrar** `( Flag[0]=False )`
    - **ou se for a sua vez** `( Turn=1 )`
  - **Mesmo que os processos executem a sequência**
    - `t0 : Flag[0] := True;`
    - `t1 : Flag[1] := True;`

    só um deles poderá entrar porque `turn` **só pode tomar um valor, 0 ou 1.**
- **Garante o progresso e uma espera limitada**
  - **P1 entrará na sua secção crítica (progresso) depois de, no máximo, uma entrada de P0 (espera limitada)**

A implementação deste algoritmo para mais de 2 processos é complicada.

Algoritmo da padaria (*bakery algorithm, Lamport*)

- **Solução para n processos.**
- **Antes de entrar na secção crítica, cada processo recebe um *ticket* com um número (como nas padarias, ...)**
- **Entra na SC o processo que tiver o número mais pequeno.**
- **Se vários processos receberem o mesmo número usa-se o identificador do processo para desempatar.**

Notação:

- **n = nº de processos**
- `(a,b) < (c,d) se (a<c) ou ((a=c) e (b<d))`
  - `a,c` **= nº do *ticket***
  - `b,d` **= identificador do processo**
- $\max(a_0,\ldots,a_{n-1})$ **é um número k (nº do *ticket*), tal que** $k \geq a_i$ **(**a_i**= nº do *ticket* de qq. um dos outros), para** `i=0..n-1`

Algoritmo da padaria

```
Variáveis partilhadas:
 choosing: Array[0..n-1] of Boolean;
 number  : Array[0..n-1] of Integer;

Inicialmente
 choosing[i] = false, para i=0..n-1;
 number[i] = 0, para i=0..n-1;

Bloco de entrada na SC
 choosing[i]:=true;
 number[i]:= max(number[0],..., number[n-1])+1;
 choosing[i]:=false;
 For j:=0 to n-1 do
  Begin
   While choosing[j] do;
   While (number[j]<>0) And
         ((number[j],j)<(number[i],i) do;
  End;

Bloco de saída da SC
  number[i]:=0;
```

- Anuncia intenção de tirar *ticket*
- Tira o *ticket*
- Anuncia que já tirou o *ticket*
- Espera que outros acabem de tirar o *ticket*
- Espera, se alguém está a executar a SC. Esse alguém deve
  - ter *ticket* → `number[j]<>0`
  - e ter direito a acesso antes de Pi → `(number[j],j)<(number[i],i)`
- Deita fora o *ticket*

Limitações das soluções por *software*

Características das soluções algoritmicas apresentadas:

- São da competência do programador.
- São complexas (principalmente, para mais do que 2 processos).
- Requerem espera activa *(busy waiting)*
  - os processos que estão a pedir para entrar na sua secção crítica estão a consumir tempo do processador, desnecessariamente.
  - Se as SCs forem demoradas seria mais eficiente bloquear os processos que estão à espera.

Soluções baseadas em *hardware*

- **Sistemas uniprocessador**
  - Os processos concorrentes não podem ter execução sobreposta no tempo, mas apenas interlaçada.
  - Para garantir a exclusão mútua bastaria inibir as interrupções, impedindo assim qualquer processo de ser interrompido.
    - perigoso permitir que os processos do utilizador o façam

```
Process Pi;
 ...
 Disable_Interrupts;
  SecçãoCrítica;
 Enable_Interrupts;
 ...
```

- **Sistemas multiprocessador**
  - A inibição de interrupções não garante a exclusão mútua.
  - São necessárias instruções especiais que permitam testar e modificar uma posição de memória num único passo (sem interrupção), mesmo com vários *CPU*s.

A instrução *Test-and-Set*

- **Equivalente numa linguagem de alto nível**

```
Function Test_and_Set(Var Target: Boolean): Boolean;
 Begin
  Test_and_Set := Target;
  Target := True;
 End;
```

- **Executada atomicamente (sem interrupção)**

- **Implementação da exclusão mútua usando *Test_and_Set*:**

```
Variáveis partilhadas:
 Lock: Boolean; // Valor inicial: false
}
```

```
Process Pi;
 ...
 While Test_and_Set(Lock) do;
  SecçãoCrítica;
 Lock:=false;
 ...
```

Não satisfaz a condição de espera limitada.
Quando um processo deixa a sua SC e há mais do que um processo à espera, a selecção do processo que entra a seguir é arbitrária.
Por isso, um processo pode ficar indefinidamente à espera (inanição).

A instrução *Swap*

- Permite trocar entre si o valor de 2 variáveis atomicamente.
- Implementação da exclusão mútua usando *Swap*:

```
Variáveis partilhadas:
 Lock: Boolean; { Valor inicial: False }
```

```
Process Pi;
 ...
 Key:=True;
 Repeat Swap(Lock,Key) Until Key=False;
  {SecçãoCrítica };
 Lock:=False;
 ...
```

Spinlocks

- É um mecanismo de sincronização em que um processo / *thread* espera num ciclo (*spins*), testando se o *lock* (ferrolho) está disponível.
- É uma solução do tipo *busy waiting* para o problema da exclusão mútua.
- As operações sobre *spinlocks* são: `InitLock`, `Lock` e `UnLock`.

```
Type Lock = Boolean;
```

```
InitLock(L: Lock)

Lock := False;
```

```
Lock(L: Lock)

While TestAndSet(L) do;
```

```
UnLock(L: Lock)

Lock := False;
```

Secção crítica:

```
...
Lock(Mutex);
   SecçãoCrítica;
Unlock(Mutex);
...
```

Características das soluções por *hardware*

+ Podem ser usadas com múltiplas secções críticas, cada secção crítica controlada por uma variável.
+ São aplicáveis a qualquer número de processos em sistemas uniprocessador ou multiprocessador

– Requerem suporte de *hardware* (instruções-máquina especiais).
– Usam espera activa (*busy waiting*)
  - Um processo à espera de entrar numa SC consome tempo de *CPU*.
– Se não forem tomadas precauções (v. pág. seguinte) é possível a inanição dos processos
  - Quando um processo deixa uma SC e há mais do que um processo à espera a selecção do processo que entra a seguir é arbitrária

Solução baseada em *hardware* com espera limitada

```
Type Lock    = Boolean;
     Waiting = Array[N] of Boolean;
```

```
InitLock(L: Lock)

Lock := False;
Waiting[1..N] := False;
```

```
Lock(L: Lock)

Waiting[i] := True;
Key := True;
while (Waiting[i] And Key)
   Key := TestAndSet(Lock);
Waiting[i] := False;
```

```
UnLock(L: Lock)

j := (i+1) Mod N;
while ((j<>i) And (Not Waiting[j]))
   j := (j+1) Mod N;
if (j=i)
   Lock := False;
else
   Waiting[j] := False;
```

Semáforos (Dijkstra,1965)

- **Semáforo**
  - mecanismo de sincronização (fornecido pelo S.O.) que não requer espera activa.
- **Semáforo S**
  - Variável inteira S inicializada com um valor não-negativo (>=0)
  - Depois de inicializada só pode ser actualizada através de duas operações atómicas:
    - `wait(S)` ou `P(S)` :
      ```
      S:=S-1;
      If S<0 then Block(S);
      ```
    - `signal(S)` ou `V(S)` :
      ```
      S:=S+1;
      If S<=0 Then WakeUp(S);
      ```
      - `Block(S)` - o processo que a invocou é bloqueado
      - `WakeUp(S)` - um processo que invocou anteriormente `Block(S)` fica pronto a correr
  - Para evitar espera activa, quando um processo tem de esperar é colocado numa fila de processos bloqueados no semáforo.

Semáforos

- De facto, um semáforo é um registo (*record* ou *structure*):
  - ```
    Type Semaphore = Record
                       Count: Integer;
                       Queue: List_of_Process
                     End;
    ```

  ```
  Var S: Semaphore;
  ```

- E as operações sobre semáforos são:
  - ```
    Wait(S) :
    S.Count:=S.Count-1;
    If S.Count<0 then
     Begin
      Colocar este processo em S.Queue;
      Bloquear este processo
     End;
    ```
  - ```
    Signal(S):
    S.Count:=S.Count+1;
    If S.Count<=0 then
     Begin
      Remover um processo, P, de S.Queue;
      Colocar P na fila de proc.s prontos
     End;
    ```

- S.Count pode ser inicializada com um valor não negativo.
- Quando S.Count >=0, S.Count representa o nº de proc.s que podem executar Wait(S) sem bloquear.
- Quando S.Count <0, |S.Count| representa o nº de proc.s que estão à espera no semáforo.


    ```

Implementação dos semáforos

- A implementação dos semáforos introduz implicitamente uma secção crítica:
  - O incremento e decremento da variável introduz uma secção crítica, bem como,
  - a alteração do valor da variável seguida de um teste do seu valor na instrução seguinte.
- `Wait()` e `Signal()` têm de ser operações atómicas.
- Como implementar a secção crítica interna ?
  - Sistema uniprocessador
    - Inibir as interrupções durante a execução de Wait() e Signal() ...
  - Sistemas multiprocessador
    - Usar uma das "soluções por *software*", anteriormente analisadas.
      - Notar que não nos livramos da espera activa, mas ela fica limitada às operações Wait() e Signal() que são curtas.
    - Usar uma das "soluções por *hardware*", anteriormente analisadas, se disponíveis (ex: `Test_And_Set` ou `Swap`)

Utilização dos semáforos
para resolver problemas de secções críticas

- Variável partilhada pelos N processos:

```
Var Mutex: Semaphore;
```

- Valor inicial

```
Mutex.Count:= 1;
```

(só 1 processo consegue entrar na SC - exclusão mútua)

```
Process Pi;
...
Wait(Mutex);
Secção_Crítica;
Signal(Mutex);
...
```

- Um processo que não consiga entrar imediatamente na SC bloqueia e cede o processador.
- Os processos bloqueados retomam a sua execução, à medida que os outros processos forem saindo das SC's correspondentes.
- Um semáforo que é inicializado com o valor 1 e é usado por 2 ou mais processos para assegurar que só um deles consegue executar uma SC ao mesmo tempo é conhecido por semáforo binário ou *mutex*.

Problemas comuns c/ a utilização de semáforos *Mutex*

- Inicializar o semáforo com o valor 0, em vez de 1
  - ⇒ nenhum processo consegue executar a secção crítica
- Trocar as operações P e V (Wait e Signal não se trocam tão facilmente)
  - ⇒ não há exclusão mútua
- "Aninhamento" inadequado de semáforos *Mutex*
  - Ex:
    O que acontece se a ordem de execução for A, C, B, D ?

    R: *deadlock*

```
A  Process P1;
   ...
   Wait(Mutex1);
   ...
B  Wait(Mutex2);
   Secção_Crítica;
   Signal(Mutex1);
   ...
   Signal(Mutex2);
   ...
```

```
C  Process P2;
   ...
   Wait(Mutex2);
   ...
D  Wait(Mutex1);
   Secção_Crítica;
   Signal(Mutex2);
   ...
   Signal(Mutex1);
   ...
```

Utilização dos semáforos para resolver problemas de sincronização

- Os semáforos que são inicializados com `S.Count=0` fornecem um mecanismo p/ sincronizar a execução de 2 processos.

Ex:
A instrução S1 do proc. P1 tem de ser executada antes da instrução S2 do proc. P2

```
Sync.Count:=0; // Inicialmente

Process P1;
...
S1;
Signal(Sync);
...

Process P2;
...
Wait(Sync);
S2;
...
```

Ex:
Dois processos necessitam de esperar um pelo outro num determinado ponto

```
Sync1.Count:=0; //Inicialmente
Sync2.Count:=0;

Process P1;
...
Signal(Sync1);
Wait(Sync2);
...

Process P2;
...
Signal(Sync2);
Wait(Sync1);
...
```

Problemas com os semáforos

- Os semáforos constituem um mecanismo poderoso para garantir a exclusão mútua e a sincronização de processos.
- Contudo, é fácil cometer erros na sua utilização. Quando as operações `Wait()` e `Signal()` estão espalhadas por vários processos, pode ser díficil compreender os seus efeitos.
- A sua utilização tem de ser correcta em todos os processos.
- Um processo mal escrito (ou "mal intencionado") pode contribuir para a falha de um conjunto de processos.

Problemas clássicos de sincronização
Problema do Produtor/Consumidor

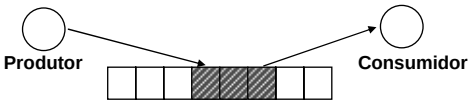
Enunciado

- Um processo produtor produz informação que é consumida por um processo consumidor.
  - ex: um programa de impressão produz caracteres que são consumidos por um *driver* de impressora
- Existe um *buffer* de itens que pode ser preenchido pelo produtor e esvaziado pelo consumidor.
- O processo produtor pode produzir um item enquanto o processo consumidor consome outro item.
- O produtor e o consumidor devem ser sincronizados de modo a que o consumidor não tente consumir um item ainda não produzido.
- Sendo o tamanho do *buffer* limitado, o produtor não pode acrescentar novos itens se o *buffer* estiver cheio.

Problema do Produtor/Consumidor

```
Var
 Buffer = ...
 Full, Empty, Mutex: Semaphore;

Inicialização:
   Full.Count:= 0;
   Empty.Count:= N;
   Mutex.Count:=1;
```



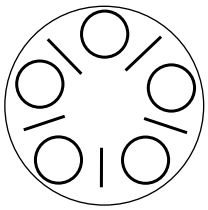

```
Process Producer;
...
Repeat
...
Produce(Item);
Wait(Empty);
Wait(Mutex);
Append(Item);
Signal(Mutex);
Signal(Full);
...
Until ...;
```

```
Process Consumer;
...
Repeat
...
Wait(Full);
Wait(Mutex);
Item=Take();
Signal(Mutex);
Signal(Empty);
Consume(Item);
...
Until ...;
```

- **Full**
  - p/ sincronizar os 2 processos;
  - não significa *buffer* cheio mas que tem pelo menos 1 item.
- **Empty**
  - p/ sincronizar os 2 processos;
  - não significa *buffer* vazio mas que se esvaziou um elem.to
- **Mutex**
  - p/implementar a exclusão mútua.

Problema dos filósofos a jantar

- **Enunciado:**
  - 5 filósofos estão sentados a uma mesa;
  - os filósofos passam a vida a pensar e a comer (arroz ...?!);
  - cada um precisa de 2 "pausinhos" para comer;
  - só há 5 "pausinhos" ...!!!
  - só pode pegar num pausinho de cada vez (e não pode roubar um do vizinho !)



- **Problema clássico de sincronização.**
- **Ilustra a dificuldade de alocar recursos entre processos sem provocar impasse / bloquemento fatal (*deadlock*) ou inanição (*starvation*).**

Problema dos filósofos a jantar

1ª tentativa de solução

- Cada filósofo é um processo.
- Um semáforo por "pausinho":
  - `Fork: Array[0..4] of Semaphore;`
- Conduz a *deadlock*
  se, por exemplo,
  cada filósofo começar por pegar
  no "pausinho" à sua esquerda (*/* direita).

```
Inicialização:

 For i:=0 to 4 do
  Fork[i].Count:=1;
```

```
Process Pi;
Repeat
 Think;
 Wait(Fork[i]);
 Wait(Fork[(i+1) Mod 5];
 Eat;
 Signal(Fork[(i+1) Mod 5];
 Signal(Fork[i];
Until ...;
```

Problema dos filósofos a jantar

2ª tentativa de solução

- Depois de pegar no "pausinho" à sua esquerda, por exemplo,
  vê se o "pausinho" da direita está livre.
  Se estiver pousa o da esquerda.

- Problema:
  - Todos pegam no "pausinho" da esquerda simultâneamente.
  - Ao verem que o "pausinho" da direita está ocupado
    pousam todos os da esquerda !!!
  - Conduz a inanição (*starvation*).

Problema dos filósofos a jantar

Uma solução

- Admitir que só 4 filósofos tentam comer simultaneamente.
- Usar um outro semáforo `M` que limita a 4 o número de filósofos que podem tentar comer.

  Inicialização:
  `M.Count:=4;`
- Então 1 filósofo pode sempre estar a comer enquanto os outros 3 seguram 1 "pausinho".

  Quando aquele terminar, um dos outros pode comer.

```
Inicialização:

M.Count:=4;
```

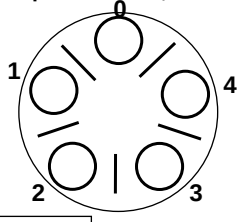
```
Process Pi;
Repeat
 Think;
 Wait(M);
 Wait(Fork[i]);
 Wait(Fork[(i+1) Mod 5];
 Eat;
 Signal(Fork[(i+1) Mod 5];
 Signal(Fork[i];
 Signal(M);
Until ...;
```

Problema dos filósofos a jantar

Outra solução ("melhor que a anterior, pois garante máximo paralelismo", *Tanenbaum*)

```
#define N 5 /* Número de filósofos */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==0) ? (N-1) : (i)-1)
#define THINKING 0
#define HUNGRY 1
#define EATING 2

semaphore mutex = 1;
semaphore s[N]; /* inicializados com zero */
```




```
void take_forks(int i) {
  wait(&mutex);
  state[i]= HUNGRY;
  test(i);
  signal(&mutex);
  wait(&s[i]);
}                          2
```

```
void test(int i) {
  if ( state[i] == HUNGRY &&          4
       state[LEFT(i)] != EATING &&
       state[RIGHT(i)] != EATING ) {
            state[i] = EATING;
            signal(&s[i]);
  }
}
```

```
void put_forks(int i) {
  wait(&mutex);
  state[i]= THINKING;
  test(LEFT(i));
  test(RIGHT(i));
  signal(&mutex);
}                          3
```

```
void philosopher(int i) {                    1
  while(.....) {
    think();
    take_forks(i); /*obtém 2 pausinhos ou bloqueia*/
    eat();
    put_forks(i);
  }
}
```

Problema dos Leitores/Escritores

- **O problema**
  - leitores e escritores acedem a informação comum
  - leitores - apenas leêm a informação
  - escritores - modificam a informação

- **Solução 1 - os leitores têm prioridade (mais simples)**
  - Enquanto um escritor estiver a aceder à informação nenhum outro escritor ou leitor pode aceder.
  - Quando um leitor estiver a aceder à informação outros leitores que entretanto cheguem podem aceder livremente.

- **Solução 2 - os escritores têm prioridade**
  - Impedir qualquer leitor de aceder à informação sempre que haja algum escritor à espera de a actualizar.
  - Quando o leitor/escritor actual terminar o acesso um escritor que esteja à espera tem prioridade sobre outros leitores.

Solução 1 - Prioridade aos leitores

```
Program ReadersWriters;
Var ReadCount: Integer;
    X, WSem: Semaphore (:=1);

Procedure Reader;
 Begin
  Repeat
   Wait(X);
    ReadCount:=ReadCount+1;
    If ReadCount=1 Then Wait(WSem);
   Signal(X);
   READUNIT;
   Wait(X);
    ReadCount:=ReadCount-1;
    If ReadCount=0 Then Signal(WSem);
   Signal(X)
  ForEver
 End;


Begin
 ReadCount:=0;
 ParBegin
  Reader;
  Writer;
 ParEnd
End.
```

```
Procedure Writer;
 Begin
  Repeat
   Wait(WSem);
    WRITEUNIT;
   Signal(WSem);
  ForEver
 End;
```

```
WSem - garante a exclusão mútua
       no acesso à informação partilhada;
       desde que um escritor esteja a aceder aos dados
       nenhum outro escritor ou leitor pode aceder;
       leitores ou escritores que cheguem entretanto
       têm de esperar em WSem.
X    - garante que a actualização de ReadCount
       é feita correctamente
ReadCount - para tomar nota do número de leitores;
            desde que haja pelo menos um leitor
            os leitores que cheguem entretanto
            não têm de esperar.
```

Solução 2 - Prioridade aos escritores

```
Program ReadersWriters;
Var ReadCount, WriteCount: Integer;
    X, Y, Z, WSem, RSem: Semaphore (:=1);

Procedure Reader;
 Begin
  Repeat
   Wait(Z);
    Wait(RSem);
     Wait(X);
      ReadCount:=ReadCount+1;
      If ReadCount = 1 Then Wait(WSem);
     Signal(X);
    Signal(RSem);
   Signal(Z);
   READUNIT;
   Wait(X);
    ReadCount:=ReadCount-1;
    If ReadCount=0 Then Signal(WSem);
   Signal(X)
  ForEver
 End;


Begin
 ReadCount:=0; WriteCount:=0;
 ParBegin
  Reader;
  Writer;
 ParEnd
End.
```

```
Procedure Writer;
 Begin
  Repeat
   Wait(Y);
    WriteCount:=WriteCount+1;
    If WriteCount=1 Then Wait(RSem);
   Signal(Y);
   Wait(WSem);
   WRITEUNIT;
   Signal(WSem);
   Wait(Y);
    WriteCount:=WriteCount-1;
    If WriteCount=0 Then Signal(RSem);
   Signal(Y);
  ForEver
 End;
```

```
Além dos semáforos e variáveis anteriores temos:
RSem - impede o acesso dos leitores
       enq.to houver pelo menos um escritor
       a querer aceder à informação partilhada
Y    - garante que a actualização de WriteCount
       é feita correctamente
WriteCount - controla o Signal a Rsem
Z    - só um leitor pode fazer fila em RSem
       os outros fazem fila em Z
```



```
Procedure Reader;
...
   Wait(Z);
    Wait(RSem);
     Wait(X);
      ReadCount:=ReadCount+1;
      If ReadCount = 1 Then Wait(WSem);
     Signal(X);
    Signal(RSem);
   Signal(Z);
   READUNIT;
   Wait(X);
    ReadCount:=ReadCount-1;
    If ReadCount=0 Then Signal(WSem);
   Signal(X);
...
```

```
Procedure Writer;
...
   Wait(Y);
    WriteCount:=WriteCount+1;
    If WriteCount=1 Then Wait(RSem);
   Signal(Y);
   Wait(WSem);
   WRITEUNIT;
   Signal(WSem);
   Wait(Y);
    WriteCount:=WriteCount-1;
    If WriteCount=0 Then Signal(RSem);
   Signal(Y);
...
```

Estado das filas dos semáforos:

Só leitores no sistema	WSem activado Não existem filas
Só escritores no sistema	WSem e RSem activados Os escritores fazem fila em WSem
Leitores e escritores, com leitor a aceder em 1º lugar	WSem activado pelo leitor RSem activado pelo escritor Todos os escritores fazem fila em WSem Um leitor faz fila em RSem Outros leitores fazem fila em Z
Leitores e escritores, com escritor a aceder em 1º lugar	WSem activado pelo escritor RSem activado pelo escritor Os escritores fazem fila em WSem Um leitor faz fila em RSem Outros leitores fazem fila em Z

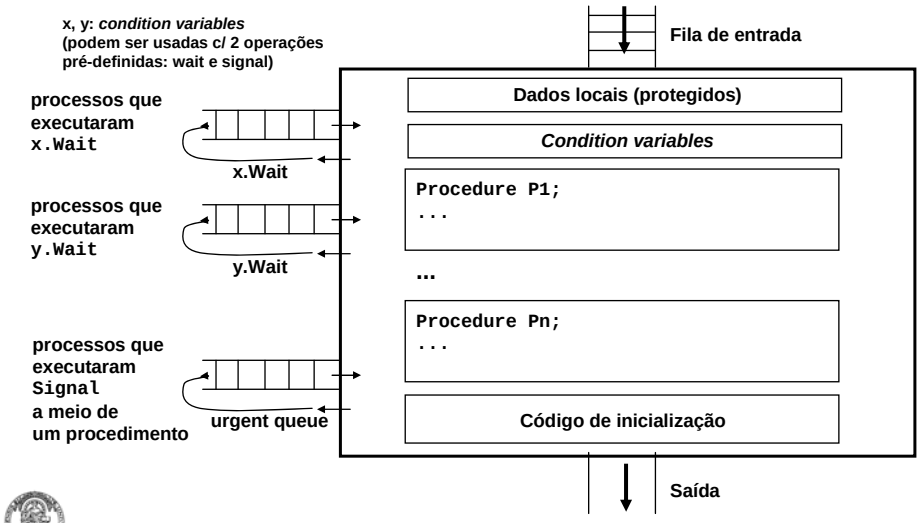
Construções de alto nível p/ exclusão mútua e sincronização

- Monitores
- Regiões críticas
- Passagem de mensagens

Monitores

- Monitor
  - módulo de *software* constituído por
    - 1 ou mais procedimentos
    - 1 secção de inicialização
    - dados locais (escondidos)
- O "mundo exterior" só "vê" os procedimentos.
- Os dados locais só podem ser manipulados no interior dos procedimentos.
- A entrada no monitor faz-se através de uma chamada a um procedimento.
- Só um processo pode estar a executar no monitor de cada vez.
- Deste modo
  os monitores permitem implementar facilmente a exclusão mútua.
- As variáveis de tipo condição (*condition variables*) permitem a sincronização.

Monitores



Exemplo (Problema do produtor/consumidor)

```
Monitor Bounded_Buffer;
Var
 Buffer: Array[0..N-1] of Char;
 NextIn, NextOut, Count: Integer;
 NotFull, NotEmpty: Condition;

Procedure Append (X: Char);
 Begin
  If Count=N then NotFull.Wait;
  Buffer[NextIn]:=X;
  NextIn:=(NextIn+1) Mod N;
  Count:=Count+1;
  NotEmpty.Signal;
 End;

Procedure Take (X: Char);
 Begin
  If Count=0 then NotEmpty.Wait;
  X:=Buffer[NextOut];
  NextOut:=(NextOut+1) Mod N;
  Count:=Count-1;
  NotFull.Signal;
 End;

Begin {Monitor initialization}
 NextIn:=0; NextOut:=0; Count:=0;
End;
```

```
(* Programa que usa o monitor *)
...
Procedure Producer;
 Var
  X: Char;
 Begin
  Repeat
   Produce(X);
   Append(X);
  Until ...
 End;

Procedure Consumer;
 Var
  X: Char;
 Begin
  Repeat
   Take(X);
   Consume(X);
  Until ...
 End;

Begin
 ParBegin
  Producer; Consumer;
 ParEnd;
End.
```

Monitores

- Tal como acontece com os semáforos é possível cometer erros de sincronização com os monitores.
  - Ex: omitir `NotFull.Signal`, no exemplo anterior
- A vantagem que os monitores têm sobre os semáforos é que todas as funções de sincronização ficam confinadas ao interior do monitor
  - mais fácil detectar e corrigir os erros de sincronização
- Os monitores podem ser implementados recorrendo a semáforos e vice-versa.
- Algumas linguagens de programação suportam monitores
  - ex: Java (http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/monitors.html)

Regiões críticas

- Uma região crítica protege uma estrutura de dados partilhada. O compilador encarrega-se de gerar código que garante a exclusão mútua no acesso aos dados.
- Requer uma variável V, de tipo T, declarada como segue:
  - `var V: Shared T;  {ex: var I: Shared Integer;}`
- A variável V só pode ser acedida dentro de uma instrução do tipo:
  - `Region V When B do S;`
    onde B é uma expressão booleana e
    S é uma instrução (simples ou composta);
- Enquanto S estiver a ser executada, nenhum outro processo pode executar esta ou outra região "guardada" pela variável V.
- Quando um processo executar a instrução Region, a expressão Booleana B é avaliada.
  - Se B for True, a instrução S é executada.
  - Se B for False, o processo é retardado até que (B seja True) e (nenhum outro processo esteja a executar numa região associada a V).

Exemplo (Problema do produtor/consumidor)

```
Var
 Buffer = Shared Record
                    Pool: Array[0..N-1] of Item;
                    Count, In, Out: Integer;
                  End;
```

```
Process Producer;
...
{Insere ItemP no buffer partilhado}
Region Buffer When Count<N do
 Begin
  Pool[In] := ItemP;
  In := (In+1) Mod N;
  Count := Count+1;
 End;
...
```

```
Process Consumer;
...
{Remove ItemC no buffer partilhado}
Region Buffer When Count>0 do
 Begin
  ItemC := Pool[Out];
  Out := (Out+1) Mod N;
  Count := Count-1;
 End;
...
```

Pascal-FC (linguagem p/o ensino de programação concorrente) suporta *conditional critical regions*

Passagem de mensagens

- **Semáforos e monitores**
  - **resolvem o problema da exclusão mútua em sistemas com 1 ou mais *CPU*s que tenham acesso a uma memória comum**
  - **não podem ser usados em sistemas distribuídos**
- **Semáforos**
  - **são construções de mais baixo nível**
- **Monitores**
  - **só estão disponíveis em algumas linguagens**
- **Passagem de mensagens**
  - **pode ser usada em sistemas c/ memória partilhada (uniprocessador ou multiprocessador), bem como em sistemas distribuídos**

Passagem de mensagens

- Os sistemas operativos implementam geralmente um sistema de mensagens que permite que os processos
  - comuniquem
  - sincronizem as suas acções
- Há pelo menos 2 operações que devem ser suportadas:
  - `send(destination,message)`
  - `receive(source,message)`
- Depois de executar `Send()/Receive()` os processos podem bloquear ou não.
- *Sender* (transmissor)
  o mais natural é não bloquear após executar `Send()`.
- *Receiver* (receptor)
  o mais natural é bloquear após executar `Receive()`.
- Por vezes, existem outras possibilidades
  - Ex: `Send()` c/bloqueio e `Receive()` c/bloqueio

Exemplo (resolução de problemas de exclusão mútua)

- Criar uma *mailbox* (ex. `Mutex`) partilhada por N processos.
- `Send()` não bloqueia.
- `Receive()` bloqueia quando `Mutex` estiver vazia.
- Inicialização:
  `Send(Mutex,Anything)`
- O 1º processo que executar `Receive()` entra na secção crítica.
  Os outros ficam bloqueados até que ele reenvie a mensagem.

```
Process Pi;

Var
 Msg: Message;
...
 Repeat
  Receive(Mutex,Msg);
  {Secção Crítica}
  Send(Mutex,Msg);
  ...
 Until ...
...
```

Exemplo (Problema do produtor/consumidor)

Usa-se 2 *mailboxes*
capacidade igual a `Capacity`

`MayConsume`
- **contém os itens**

`MayProduce`
- **contém mensagens nulas**

```
...
Begin
 Create_Mailbox(MayProduce);
 Create_Mailbox(MayConsume);
 For I:=1 to Capacity do
  Send(MayProduce,Null);
 ParBegin
  Producer; Consumer
 ParEnd
End.
```

```
...
Procedure Producer;
 Var
  PMsg: Message;
 Begin
  Repeat
   Receive(MayProduce,PMsg);
   PMsg:=produce();
   Send(MayConsume,PMsg);
  Until ...
 End;

Procedure Consumer;
 Var
  CMsg: Message;
 Begin
  Repeat
   Receive(MayConsume,CMsg);
   Consume(CMsg);
   Send(MayProduce,Null);
  Until ...
 End;
```

DEADLOCKS *

- O problema dos *deadlocks*
- Condições necessárias para a sua ocorrência
- Métodos de tratamento dos *deadlocks*

(*) *Deadlock* - impasse; bloqueio fatal; bloqueio permanente

O problema dos *deadlocks*

- Vários processos, executando concorrentemente, competem pelos mesmos recursos:
  - dispositivos físicos (ex: impressora, espaço de memória, ...)
  - dispositivos lógicos (ex: secção crítica, ficheiro, ...)
- Quando um processo detém um recurso, os outros têm de esperar.
- Em certas circunstâncias, o sistema pode encravar e nenhum processo pode avançar (*deadlock* ou bloqueio fatal)
- Os recursos alocados a processos encravados não são utilizáveis até que o *deadlock* seja resolvido.

Exemplos - possibilidade de *deadlock*

Exemplo 1

- Sistema com 1 disco + 1 *tape*
- 2 processos competindo pelo uso exclusivo destes recursos
- Acontece um *deadlock* se cada processo obtiver um recurso e requisitar o outro

```
Process P1;
 Begin
   ...
   Request(D);
   Request(T);
   ...
   ...
   Release(T);
   Release(D);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Request(T);
   Request(D);
   ...
   ...
   Release(D);
   Release(T);
   ...
 End;
```

Exemplo 2

- 2 processos utilizando 2 semáforos *Mutex*, S e Q
- Acontece um *deadlock* se a ordem de execução for, por ex.:

```
P1 - Wait(S)
P2 - Wait(Q)
P2 - Wait(S)
```

```
Process P1;
 Begin
   ...
   Wait(S);
   Wait(Q);
   ...
   ...
   Signal(Q);
   Signal(S);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Wait(Q);
   Wait(S);
   ...
   ...
   Signal(S);
   Signal(Q);
   ...
 End;
```

Exemplos

Exemplo 3

- 2 processos utilizando que comunicam entre si através de 2 filas de mensagens
- acontece *deadlock* se a operação Receive bloquear quando não há mensagens na fila

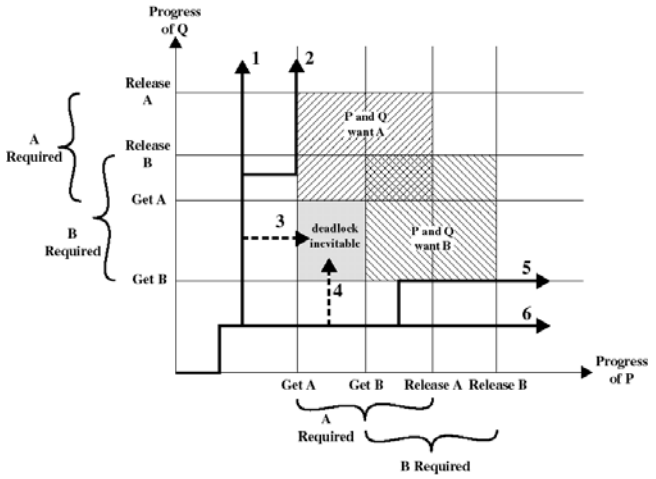
```
Process P1;
 Begin
   ...
   Receive(P2);
   ...
   ...
   Send(P1);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Receive(P1);
   ...
   ...
   Send(P2);
   ...
 End;
```

Exemplo: possibilidade de *deadlock*

```
Processo P;
…
Get(A);
…
Get(B);
…
Release(A);
…
Release(B);
…

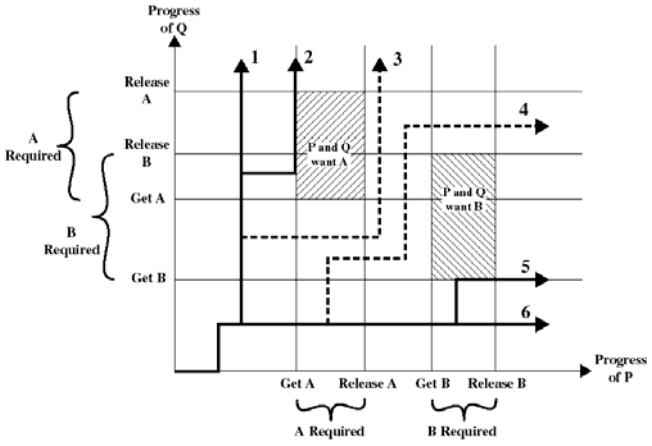
Processo Q;
…
Get(B);
…
Get(A);
…
Release(B);
…
Release(A);
…
```



Exemplo: impossibilidade de *deadlock*

```
Processo P;
…
Get(A);
…
Release(A);
…
Get(B);
…
Release(B);
…

Processo Q;
…
Get(B);
…
Get(A);
…
Release(B);
…
Release(A);
…
```



Definição

- ***Deadlock*** **:**
  - **bloqueio permanente de um conjunto de processos que competem por recursos do sistema ou comunicam entre si.**

***Deadlock** versus **starvation*:**

- ***Deadlock*** **(bloqueio fatal)** ***:***
  - **esperar indefinidamente por alguma coisa que não pode acontecer.**
- ***Starvation*** **(inanição) :**
  - **esperar muito tempo por alguma coisa que pode nunca acontecer.**

Atribuição e utilização de recursos

- **O sistema operativo força uma utilização adequada dos recursos reutilizáveis fornecendo serviços para a sua requisição / utilização / libertação.**
  - ***Request***
    - **Geralmente, formas de abrir (*open*) ou alocar (*alloc*) um recurso.**
    - **Bloqueia o processo até que recurso seja concedido.**
      (Nem sempre ! Pode negar o recurso e informar o processo.
      ex: o *open* de um ficheiro retorna erro se o ficheiro não puder ser aberto)
  - ***Use***
    - **Serviços especiais para o uso do recurso (ex: *read* e *write* de ficheiros).**
  - ***Release***
    - **Formas de fechar (*close*) ou libertar (*free*) um recurso.**
    - **O S.O. pode então conceder o recurso a outro processo.**

Atribuição e utilização de recursos

- Outro tipo de recursos, os chamados <u>recursos consumíveis</u> (recursos que podem ser criados e destruídos), são criados pelos processos e partilhados por eles, geralmente em exclusão mútua:
  - Mensagens
  - Sinais
  - Semáforos
- Certas combinações de acontecimentos <u>podem</u> produzir *deadlocks*.
  - Ex:

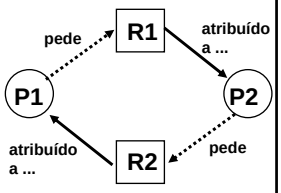
se o *Receive* de mensagens se fizer com bloqueio

```
Process P1;
 Begin
   ...
   Receive(P2);
   ...
   ...
   Send(P1);
   ...
 End;
```

```
Process P2;
 Begin
   ...
   Receive(P1);
   ...
   ...
   Send(P2);
   ...
 End;
```

Condições necessárias para a ocorrência de um *deadlock*

- <u>Exclusão mútua</u>
  - Só um processo pode usar um recurso de cada vez.
- <u>Retém e espera</u>
  - Um processo pode deter recursos enquanto está à espera que lhe sejam atribuídos outros recursos.
- <u>Não preempção dos recursos</u>
  - Quando um processo detém um recurso só ele o pode libertar.
- <u>Espera circular</u>
  - Deve existir um conjunto de processos $\{P_1, P_2, ..., P_n\}$ tal que P_1 está a espera de um recurso que P_2 detém, P_2 está a espera de um recurso que P_3 detém, ... , P_n está a espera de um recurso que P_1 detém.



Condições para *deadlock*

- **O *deadlock* ocorre se e só se a condição de espera circular não tiver solução.**
- **A condição de espera circular não tem solução quando as 3 primeiras condições se verificam.**
- **As 3 primeiras condições são necessárias mas não suficientes para que ocorra uma situação de *deadlock*.**
- **Por isso, as 4 condições tomadas em conjunto constituem condições necessárias e suficientes para um *deadlock*.**

Métodos de tratamento dos *deadlocks*

- **Prevenir (*prevent*)**
  - **Assegurar que pelo menos 1 das 4 condições necessárias não se verifica.**
- **Evitar (*avoid*)**
  - **Não conceder recursos a um processo, se essa concessão for susceptível de conduzir a *deadlock*.**
- **Detectar e recuperar**
  - **Conceder sempre os recursos enquanto existirem disponíveis; periodicamente, verificar a existência de processos encravados e, se existirem, resolver a situação.**

Alternativa (por parte do S.O.): ignorar os *deadlocks*

Prevenir os *deadlocks*

Assegurar que pelo menos uma das 4 condições não se verifica.

- **Exclusão mútua**
  - **Solução: usar só recursos partilháveis …!**
  - **Problema:**
    - **certos recursos têm de ser usados com exclusão mútua.**
  - **A utilização de *spooling* (ex: da impressora) ajuda a prevenir esta condição**

- **Retém e espera**
  - **Solução: Garantir que quando um processo requisita um recurso não detém nenhum outro recurso ⇒**
    - **Requisitar todos os recursos antes de começar a executar ou**
    - **Requisitar os recursos incrementalmente, mas libertar os recursos que detém quando não conseguir requisitar os recursos de que precisa.**
  - **Problemas:**
    - **Sub-utilização dos recursos.**
    - **Necessidade de conhecimento prévio de todos os recursos necessários. (não faz sentido em sistemas interactivos)**
    - **Possibilidade de inanição.**

Prevenir os *deadlocks*

- **Não preempção de recursos**
  - **Solução: Permitir a preempção de recursos. Q.do a um processo é negado um recurso deverá libertar todos os outros, *ou* o processo que detém esse recurso deverá libertá-lo.**
  - **Problema: só é aplicável a recursos cujo estado actual pode ser guardado e restaurado facilmente (ex.: memória e registos da *CPU*)**

- **Espera circular**
  - **Solução: Protocolo para impedir espera circular; os vários tipos de recursos são ordenados e e os processos devem requisitá-los por essa ordem**
    - **Ex. 1-*tapes* ; 2-ficheiros ; 3-impressoras**
    - **O processo deve requisitar os recursos sempre pela mesma ordem.**
    - **Se já requisitou ficheiros, então, só pode requisitar a impressora.**
    - **Se o processo necessitar de várias instâncias do mesmo recurso deve requisitá-las de uma só vez.**

Prevenir os *deadlocks*

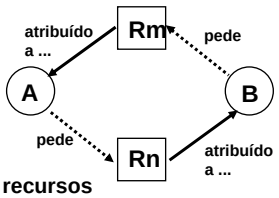
◆ **Espera circular** (cont.)

■ **Demonstração que o protocolo funciona (p/2 processos)**

- **Pressuposto: o recurso Ri precede Rj, se i<j**
- **Admitamos que 2 processos A e B estão encravados porque**

o processo A possui Rm e requisitou Rn ⇒ m<n
o processo B possui Rn e requisitou Rm ⇒ n<m

É impossivel que (m<n) e (n<m) !!!
(demonstração por contradição)



A já tem Rm **Se m>n, A não pode requisitar Rn**
B já tem Rn **Se n<m, B não pode requisitar Rm**
Logo, nunca se pode fechar o ciclo.

■ **Problemas**

- **Ineficiência devido à ordenação imposta aos recursos**
  - **os recursos têm de ser requisitados por uma certa ordem em vez de serem requisitados à medida que são precisos.**
  - **certos recursos são negados desnecessariamente**
- **Difícil encontrar uma ordenação que funcione.**

Evitar os *deadlocks*

◆ **Prevenir**

■ **Evitar os *deadlocks* indirectamente, impedindo que uma das 4 condições se verifique.**

◆ **Evitar**

■ **Permitir que aquelas condições se verifiquem, e decidir, perante cada pedido de recursos, se ele pode conduzir a um *deadlock*, caso os recursos sejam atribuídos. Se sim, negar a atribuição dos recursos pedidos.**

■ **⇒ Examinar dinamicamente o estado de alocação de recursos para assegurar que não vai ocorrer uma situação de espera circular.**

◆ **Duas estratégias p/ evitar *deadlocks*:**

■ **Não começar a executar um processo se as suas necessidades, juntamente c/ as necessidades dos que já estão a correr, forem susceptíveis de conduzir a um *deadlock*.**

■ **Não conceder um recurso adicional a um processo se essa concessão for susceptível de conduzir a um *deadlock*.**

Evitar os *deadlocks*

Notação:

- `n` - nº de processos
- `m` - nº de classes de recursos
- `Available[1..m]` - quantidade de recursos de cada classe disponíveis num determinado instante
- `Max[1..n,1..m]` - necessidades máximas de cada processo relativamente aos diferentes recursos
- `Allocation[1..n,1..m]` - número de recursos de cada classe atribuídos a cada processo
- `Need[1..n,1..m]` - necessidades que falta satisfazer
  `Need[i,j]= Max[i,j]-Allocation[i,j]`

- Se `X` e `Y` são vectores de comprimento n
  - Diz-se que `X<=Y` se `X[i]<=Y[i]`, para `i=1..n`
  - Diz-se que `X<Y` se `X<=Y` e `X<>Y`
- Se `M` é uma matriz
  - M_i representa a linha `i` da matriz

Evitar os *deadlocks*

- Os principais algoritmos para evitar *deadlocks* são baseados no conceito de estado seguro.
- Um estado diz-se seguro se o sistema conseguir alocar recursos a cada processo, por uma certa ordem, de modo a evitar *deadlocks*.
- Evitar ⇒ assegurar que o sistema nunca entra num estado inseguro (estado que pode conduzir a *deadlock*).

- **1ª estratégia**
  - O início de execução de um novo processo é negado se as máximas necessidades de todos os processos em execução mais as necessidades deste novo processo excederem a quantidade de qualquer classe de recurso
  - Esta estratégia é demasiado restritiva.

Evitar os *deadlocks*

- **2ª estratégia**
  - **Não conceder um recurso adicional se essa concessão for susceptível de conduzir a um *deadlock*.**
- **Algoritmo do banqueiro / teste de estado seguro (*Dijkstra*)**
  - **1. Vectores**
    - **Work[1..m]** - quant. de recursos disponíveis em cada instante da simulação
    - **Finish[1..n]** - indica se o proc. i tem possibilidade de obter os recursos que precisa

    Inicializar:
    ```
    Work := Available;    {recursos disponíveis no instante da chamada }
    Finish[i] := False, para i=1..n;
    ```
  - **2. Encontrar um processo i, tal que**
    (Finish[i]=False) and ($Need_i$<=Work);
    Se não existir tal i, saltar para 4. {não existe se condição ant. = False}
  - **3. Work := Work + $Allocation_i$** {se entrar aqui significa que encontrou um processo}
    Finish[i] := True; { que tem possibilidade de terminar.}
    Saltar para 2; { Após terminar, os seus recursos são libertados; daí o sinal + }
  - **4. Se Finish[i]=True, para i=1..n**
    então o sistema está num estado seguro.

ou, de outra forma:

```
C = {conjunto de todos os processos};
While (C != CONJUNTO_VAZIO) {
  Procurar um P, elemento de C, que possa terminar;
  Se não existir nenhum P {
      o estado é INSEGURO;
      terminar; }
  senão {
      remover P de C;
      adicionar os recursos de P aos rec.s disponíveis; }
}
O estado é SEGURO;
```

Evitar os *deadlocks*

- **Algoritmo de requisição de recursos**
  - $Request_i$ - vector que representa as necessidades do processo P_i
  - 1. Se $Request_i <= Need_i$, saltar para 2,
    senão assinalar erro(P_i excedeu os limites que tinha declarado)
  - 2. Se $Request_i <= Available$, saltar para 3
    senão P_i tem de esperar, dado que os recursos não estão disponíveis.
  - 3. Simular a alocação de recursos ao processo P_i

    $Available := Available - Request_i$;
    $Allocation_i := Allocation_i + Request_i$;
    $Need_i := Need_i - Request_i$;

    Se o estado resultante for seguro {--> ALGORITMO DO BANQUEIRO}
    a transacção é completada e o processo P_i recebe os recursos.

    Se o estado resultante for inseguro,
    P_i tem de esperar por $Request_i$ e
    o estado de alocação anterior é restaurado.
- **Algoritmo de libertação de recursos**
  - Quando um recurso é libertado
    actualizar o vector Available e
    reconsiderar os pedidos pendentes para esse recurso, se os houver.

Evitar os *deadlocks*

- **Vantagens (algoritmo do banqueiro):**
  - **Menos restritivo do que a prevenção.**
  - **Não requer a requisição simultânea de todos os recursos necessários.**
  - **Não obriga à preempção dos recursos.**
- **Dificuldades**
  - **Necessidade de conhecimento antecipado de todos os recursos necessários**
    ⇒ utilidade prática limitada.
  - ***Overhead* necessário para detectar os estados seguros.**

Exemplo

- 5 processos (P0..P4)
- 3 tipos de recursos
  - A (10 instâncias)
  - B (5 instâncias)
  - C (7 instâncias)

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

- O sistema está actualmente num estado seguro porque a sequência P1, P3, P4, P2, P0 satisfaz o critério de segurança

Exemplo

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

3 0 2 → simulação da alocação de recursos

0 2 0 → simulação da alocação de recursos

2 3 0

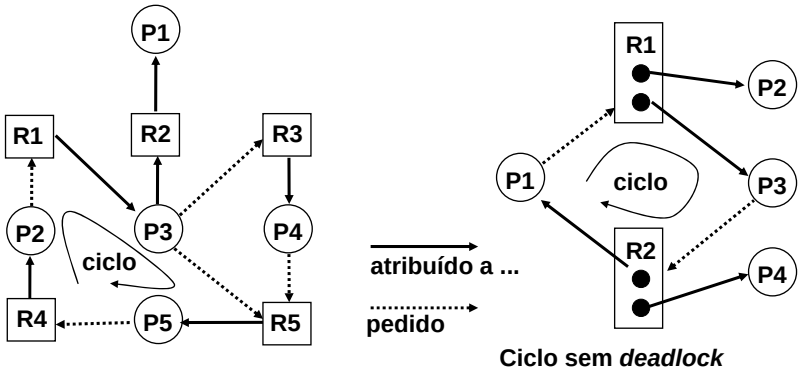
- Quando P1 faz o pedido → $Request_1=(1,0,2)$
  - Verificar que $Request_1$ <= Available → (1,0,2) <= (3,3,2) ? → True
  - A execução do teste de segurança mostra que a sequência P1, P3, P4, P0, P2 também satisfaz o critério de segurança
  - Por isso, o pedido pode ser atendido.

Detecção e Recuperação

- **Os recursos são concedidos se estiverem disponíveis. Periodicamente detecta-se a ocorrência de *deadlocks*. Se existir *deadlock*, aplica-se uma estratégia de recuperação.**
- **Quando fazer a detecção ?**
  - **Sempre que é concedido um novo recurso ⇒ *overhead* elevado.**
  - **Com um período fixo.**
  - **Quando a utilização do processador é baixa.**
- **Como proceder à recuperação ?**
  - **Avisar o operador e deixar que seja ele a tratar do assunto.**
  - **O sistema recupera automaticamente**
    - **Abortando alguns processos envolvidos numa espera circular ou**
    - **Fazendo a preempção de alguns recursos.**

Detecção

- **<u>Método 1</u>: quando há uma única instância de cada tipo de recurso**
  - **Manter um grafo de processos e recursos *(wait for graph).***
  - **Periodicamente, invocar um algoritmo de detecção de ciclos em grafos.**



Ciclo com *deadlock*

Ciclo sem *deadlock*
(há mais do que uma instância de cada recurso)

Detecção

- **Método 2: quando há várias instâncias de cada tipo de recurso.**
- **Estruturas de dados**
  - `Available [1..m]` `Finish[1..n]`
    `Allocation [1..n,1..m]` `Work[1..m]`
    `Request[1..n,1..m]`
- **Algoritmo**
  - `1. Inicializar Work := Available;`
    `Para i:=1 até n`
    `Se Allocation`$_i$ `<> 0 então Finish[i] := False {Pi pode estar encravado}`
    `senão Finish[i] := True;`
  - `2. Encontrar um i tal que`
    `(Finish[i] = False) e (Request`$_i$`<=Work);` `{Pi pode terminar}`
    `Se não existir tal i, saltar para 4.`
  - `3. Work := Work + Allocation`$_i$`;` `{Quando terminar,`
    `Finish[i] := True;` `libertará os recursos}`
    `Saltar para 2;`
  - `4. Se Finish[i]=False para qualquer i, 1 <= i<= n,`
    `o sistema está num estado de` ***deadlock***`.`
    `Além disso, se Finish[i]=False, o processo Pi está encravado.`

Recuperação

Alternativas

- **Terminação de processos**
- **Preempção de recursos**

- **Terminação de processos**
  - **Abortar todos os processos encravados.**
  - **Abortar sucessivamente um processo até eliminar o *deadlock*.**
    - **Por que ordem ?**
      Factores a ter em conta
      - **prioridade dos processos**
      - **tempo de computação passado (e futuro ...? → estimado)**
      - **recursos usados**
      - **recursos necessários para acabar**
      - **tipo de processo (interactivo ou *batch*)**
      - **...**
  - **Correr o algoritmo de detecção após cada terminação de um processo.**

Recuperação

- **Preempção de recursos**
  - **Retirar sucessivam. recursos aos processos até desfazer o *deadlock* .**
  - **Questões:**
    - **Que recursos e que processo seleccionar ?**
      - **Factores de custo:**
        - nº de recursos detidos pelos processos encravados;
        - tempo de computação que os processos já usaram.
    - **Que fazer com o processo a quem foram retirados os recursos ?**
      - **Fazer o *rollback***
        - Retornar o processo a um estado seguro e continuar a partir daí (difícil!)
        - Abortar o processo e recomeçar de início.
    - **Como evitar a inanição de um processo, isto é, que seja sempre o mesmo processo a ser seleccionado como "vítima" ?**
      - **Tomar nota do nº de *rollbacks* no factor de custo.**

Estratégia integrada

- **Nenhum dos métodos analisados anteriormente é adequado para todos os tipos de problemas de alocação de recursos.**
- **Solução: combinar os 3 métodos, partindo os recursos em classes, e seleccionar o método mais adequado para cada classe.**
- **Exemplo:**
  - **Espaço de *swap*, em disco (*swappable space*)**
    - ***Prevenir* a condição de retém e espera:**
      todo o espaço de *swap* em disco deve ser requisitado de uma única vez.
  - **Memória de dados ou código**
    - ***Prevenir* a condição de não preempção:**
      quando não há memória suficiente no sistema para a próxima alocação, um ou mais processos são *swapped* para disco libertando assim a memória.
  - **Recursos internos do SO**
    - ***Prevenir* a condição de espera circular**
      através da ordenação dos recursos e da requisição e alocação por essa ordem.
  - **Recursos dos processos (dispositivos de *I/O*, ficheiros, ...)**
    - **Evitar o *deadlock*:**
      o processo indica à partida os recursos que necessita ...

Ignorar os *deadlocks*

- **Aproximação usada em muitos sistema operativos, incluindo o UNIX.**
- **Considera-se que, é preferível que ocorra um *deadlock*, de vez em quando, do que estar sujeito ao *overhead* necessário para os evitar/detectar.**
- **O UNIX limita-se a negar os pedidos se não tiver os recursos disponíveis.**
- **Alguns sistemas (ex: VMS) iniciam um temporizador sempre que um processo bloqueia à espera de um recurso. Se o pedido continuar bloqueado ao fim de um certo tempo, é então executado um algoritmo de detecção de *deadlocks*.**
- **Os *deadlocks* ocorrem essencialmente nos processos do utilizador, não nos processos do sistema.**

GESTÃO DE MEMÓRIA

- **Conceitos introdutórios**
  - Criação de um programa executável
  - Recolocação
  - Ligação (*linking*)
  - Carregamento (*loading*)
  - Endereços reais e endereços virtuais
  - *Swapping*
  - Protecção de memória
- **Técnicas de gestão de memória**
  - Alocação contígua e alocação não-contígua
  - Partição fixa e partição dinâmica
  - Paginação e segmentação
  - Memória virtual

Sistemas c/ Monoprogramação e Multiprogramação

Sistemas c/ monoprogramação

- um processo em memória de cada vez
- o processo pode acupar toda a memória disponível p/ o utilizador
- técnica de sobreposição (***overlay)*** permitia correr processos que ocupavam mais memória do que a memória física

Sistemas c/ multiprogramação

- necessidade de **recolocação**
  - » carregar o programa numa zona arbitrária da memória
- necessidade de **protecção**
  - » isolar os espaços de endereçamento do S.O. e das aplicações
- necessidade de **partilha**
  - » cooperação entre processos
- técnica de memória virtual permite correr processos que ocupam mais memória do que a memória física

Criação de um Programa Executável

Programa-fonte
- **endereços simbólicos**
  - » **ex: nextstep**

Compilador

Módulo-objecto
- **endereços recolocáveis**
  - » **ex: 92 *bytes* a partir do início do módulo**

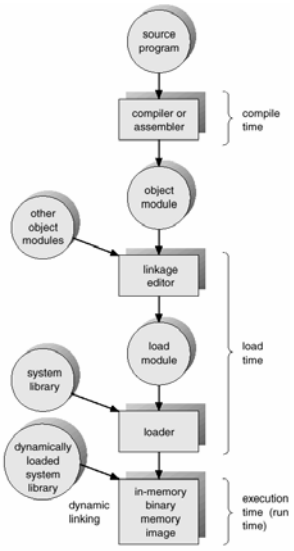
Linker

Módulo-carregável
- **módulo-objecto + rotinas da biblioteca**
- **endereços recolocáveis referidos ao início do módulo**

Loader

Imagem binária em memória

Criação de um Programa Executável



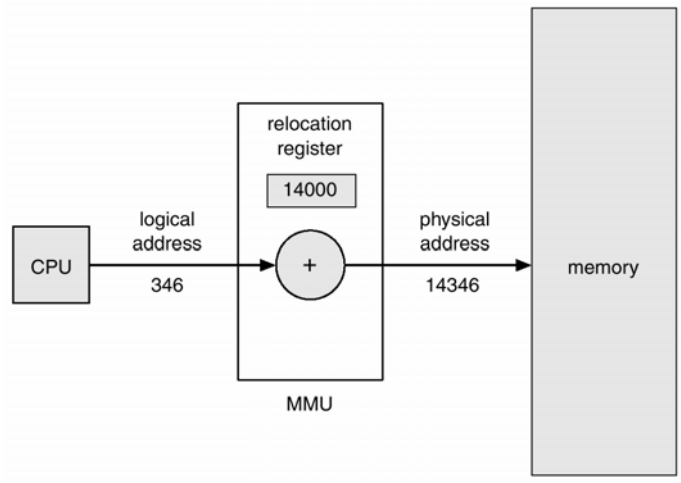
Recolocação

Capacidade de carregar e executar um dado programa num lugar arbitrário da memória

Formas de recolocação:

- **estática**
  - » **antes ou durante o carregamento do programa**
    - **antes → na compilação ou na ligação (*linking*)**
      ⇒ conhecer *a priori* onde o processo vai ser carregado
      (ex: programas com a extensão .COM em MS-DOS)
    - **durante → feita pelo *loader***
      o compilador gera código recolocável
  - » **o programa não pode deslocado na memória (difícil o *swapping*)**
- **dinâmica**
  - » **durante a execução do programa**
  - » **o processo pode ser deslocado na memória durante a execução**
  - » **⇒ suporte de *hardware* → *base register* (*s)***
    conteúdo = endereço físico inicial do programa ou de um segmento do programa

Recolocação



Ligação *(Linking)*

Juntar um conjunto de módulos-objecto produzindo um módulo contendo o programa global e os dados a serem passados ao *loader*.

A ligação *(linking)* pode ser feita:

- **estaticamente**
- **dinâmicamente**

Ligação estática:

- **cada módulo-objecto, compilado ou assemblado é criado com referências relativas ao início do módulo;**
- **todos os módulos são colocados num único módulo recolocável com referências relativas ao início do módulo global.**

Ligação *(Linking)*

Ligação dinâmica

- **durante o carregamento *(load-time)***
  - » **O módulo carrregável é lido para memória e qualquer referência a um módulo externo faz com que o *loader* carregue este módulo e altere as referências à memória necessárias.**
  - » **Vantagens:**
    - **fácil incorporar versões novas ou alteradas do módulo externo sem recompilar**
    - **fácil partilhar código entre várias aplicações (basta um cópia de cada módulo)**
- **durante a execução *(run-time)***
  - » **Parte da ligação é adiada até à altura da execução.**
  - » **Quando é feita referência a um módulo ausente, o S.O. localiza-o, carrega-o e liga-o ao módulo que o invocou.**
  - » **Vantagens:**
    - **permite alterar rotinas da biblioteca sem recompilar os programas ;**
    - **permite que uma única cópia de uma rotina seja partilhada por diferentes processos .**

Carregamento *(Loading)*

Operação de colocação de um módulo carregável em memória.

Vários tipos de carregamento

- **Absoluto**
  - » **O programa é carregado sempre no mesmo endereço inicial .**
  - » **Todas as referências à memória devem ser absolutas .**
  - » **A atribuição de endereços às referências de memória e feita pelo programador, pelo *assembler* ou pelo compilador.**
- **Recolocável**
  - » **A decisão quanto ao sítio onde se carrega o programa é tomada na altura do carregamento.**
  - » **O *assembler* / compilador não produz endereços absolutos mas relativos ao início do programa.**
- **Dinâmico em *run-time***
  - » ***Swapping* ⇒ possibilidade de carregar o mesmo processo em zonas diferentes da memória .**
  - » **A geração de endereços absolutos não pode ser feita por altura do carregamento inicial.**
  - » **O endereço absoluto só é calculado quando a instrução é efectivamente executada ⇒ suporte de *hardware*.**

Técnica de *overlays*

Técnica que permitia (actualmente caiu em desuso) correr processos que ocupavam mais memória do que a memória física disponível.

Ideia:

- **Dividir o programa numa parte residente (sempre em memória) e em *overlays* que são módulos independentes que são carregados em memória a pedido do programa.**
- **A comunicação entre os *overlays* é feita através da parte residente.**

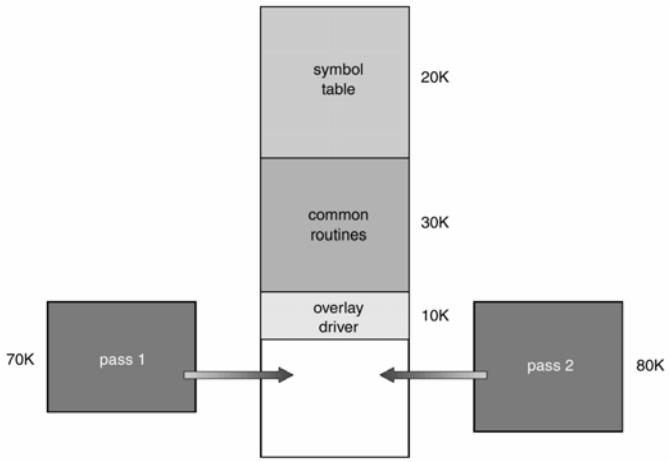
⇒ *overlay driver*

Dificuldades

- **(dimensão da parte residente + dimensão dos *overlays*) < dimensão da memória**
- **Dividir certos programas (compete ao programador fazê-lo)**

Alguns compiladores facilitavam a tarefa do programador.

Técnica de *overlays*



***Assembler* de 2 passos**

Endereços reais e Endereços virtuais

Endereçamento real

- **O endereço indicado pelo programa é exactamente o que é acedido na memória do computador (endereço físico), sem qualquer transformação operada pelo *hardware*.**
- **Desvantagens:**
  - » **dimensão dos programas limitada à dimensão da memória física (técnica de *overlay* permitia ultrapassar esta limitação)**
  - » **o programa só pode funcionar nos endereços físicos para que foi escrito**
  - » **multiprogramação difícil / impossível**

Endereçamento lógico ou virtual

- **Os endereços gerados pelo programa são convertidos pelo processador (pela *MMU-Memory Management Unit*), durante a execução, em endereços físicos.**
- **A palavra referenciada pelo endereço virtual pode estar em memória principal ou secundária (⇒carregá-la previamente)**

Endereços reais e Endereços virtuais

Endereço lógico / virtual
- endereço gerado pela *CPU*

Endereço físico / real
- endereço visto pela unidade de memória (carregado no *memory address register*)

Espaço de endereçamento lógico
- conjunto de todos os end.os lógicos gerados por um programa

Espaço de endereçamento físico
- conjunto de todos os end.os físicos correspondentes àqueles end.os lógicos

O mapeamento entre os 2 espaços é feito em *run-time* pela *Memory Management Unit - MMU* .

Swapping

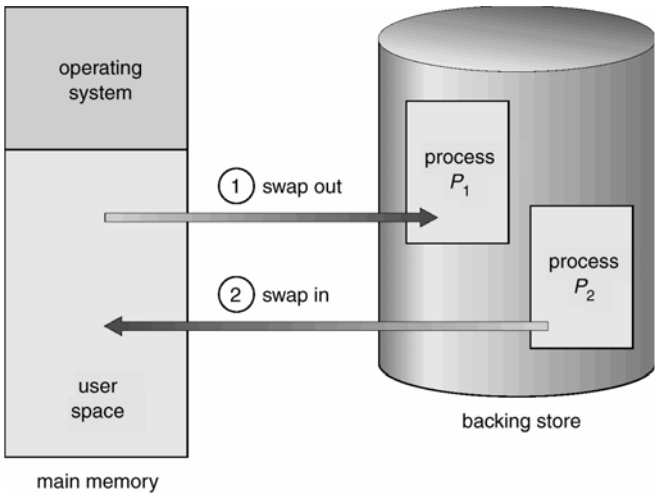
Swapping
- transferência de programas entre a memória principal e o disco

***Swapping* ⇒**
- possibilidade de recolocação
- comutação de contexto mais demorada

Swapper
- processo do S.O.
  - » selecciona processo(s) que vai sofrer *swap-out* (processos bloqueados, processos c/baixa prioridade, …)
  - » selecciona processo que vai sofrer *swap-in* (baseado no tempo que passou em disco, prioridade,…)
  - » aloca e gere o espaço de *swapping*

Swap-file
- onde é guardada a imagem do processo *swapped-out*
  - » uma única p/ todos os processos, com tamanho fixo e acesso sem ser através do *file system* (mais rápido)
  - » uma p/ cada processo

Swapping



Protecção de memória

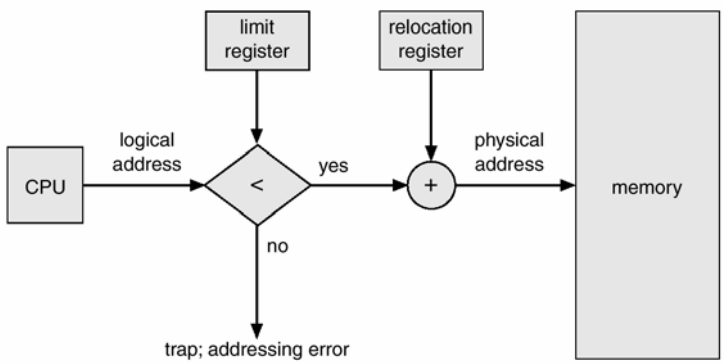
Cada processo (do S.O / do utilizador) deve ser protegido contra interferências indesejáveis de outros processos, acidentais ou intencionais.

Todas as referências de memória têm de ser verificadas em tempo de execução *(run-time)* ⇒

Suporte de *hardware*

- ***limit register***
  - » **conteúdo = endereço virtual mais elevado referenciado no programa**

Protecção de memória



Partilha de memória

Permitir que vários processos acedam à mesma zona de memória.

- **Processos que executam o mesmo programa devem ter a possibilidade de partilhar o código do programa.**
- **As bibliotecas partilhadas e as bibliotecas com ligação dinâmica são partilhadas por vários processos.**
- **Os processos também podem ter necessidade de partilhar dados.**

O *swapping* complica a partilha.

Para facilitar a partilha
as regiões de memória partilhada podem ser reservadas
no espaço de endereçamento do S.O.
e este passa a cada aplicação o endereço dessas regiões.

Técnicas de gestão de memória

Alocação contígua

- **Partição fixa**
  - » **partições de tamanho igual**
  - » **partições de tamanho diferente**
- **Partição dinâmica**

Alocação não-contígua

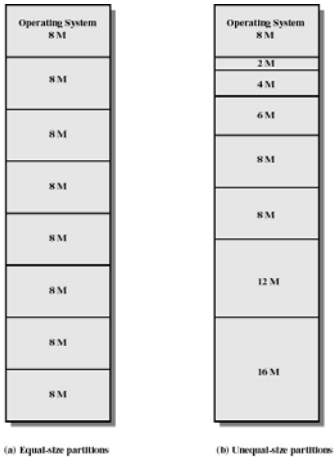
- **Paginação**
- **Segmentação**
- **Segmentação c/paginação**

Partição fixa

- **A memória destinada aos processos do utilizador é dividida em partições de tamanho fixo (eventualmente diferentes entre si).**
- **O S.O. mantém uma tabela com indicação das partições ocupadas.**
- **Inicialmente ... os programas eram compilados p/uma determinada partição ⇒**
  - **uma partição podia ter uma fila de programas à espera de poder executar enquanto outras filas estavam vazias**
- **Posteriormente ... possibilidade de recolocação**
  - **um programa pode ser carregado em qualquer partição**
- **Se necessário recorrer a *swapping*.**

Partição fixa

- **Partições de tamanho igual**
  - **Dificuldades**
    - » **um programa pode não caber nas partições ⇒ usar *overlays***
    - » **<u>fragmentação interna</u> - utilização ineficiente da memória quando o programa não ocupa a partição toda**
- **Partições de tamanho diferente**
  - **Ex.:**
    - » **1 partição de 1 MB**
    - » **1 partição de 768KB**
    - » **2 partições de 512KB**
    - » **1 partição de 320 KB**



Exemplo de partição fixa numa memória c/ 64 MB

Partição fixa

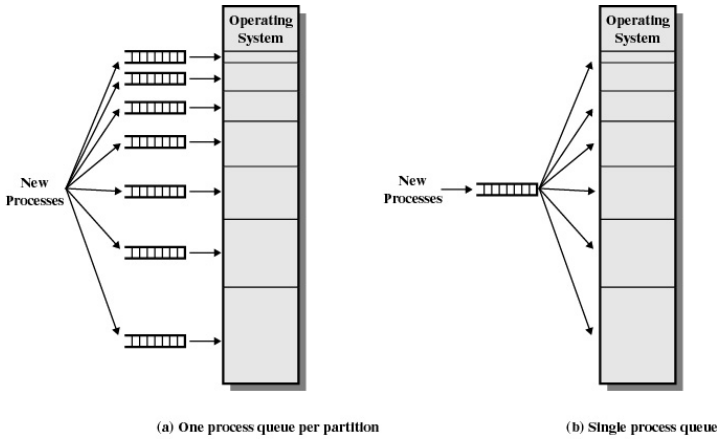
- **Mecanismo de protecção**
  - **par de registos onde são carregados os endereços máx. e min. da partição actual**
- **Algoritmo de colocação**
  - **tamanho igual - carregar o processo em qualquer partição disponível**
  - **tamanho diferente**
    - » **atribuir o proc. à menor partição em que ele cabe (mínimo desperdício) → uma fila por partição**

      <u>ou</u>
    - » **escolher a menor partição disponível capaz de conter o processo quando ele for carregado → uma fila única para todas as partições**

Desvantagens:

- **o nº de partições limita o nº de processos activos**
- **o tamanho das partições é fixado por ocasião da geração do sistema ⇒ utilização ineficiente da memória, q.do os processos são pequenos.**

Partição fixa



(a) One process queue per partition (b) Single process queue

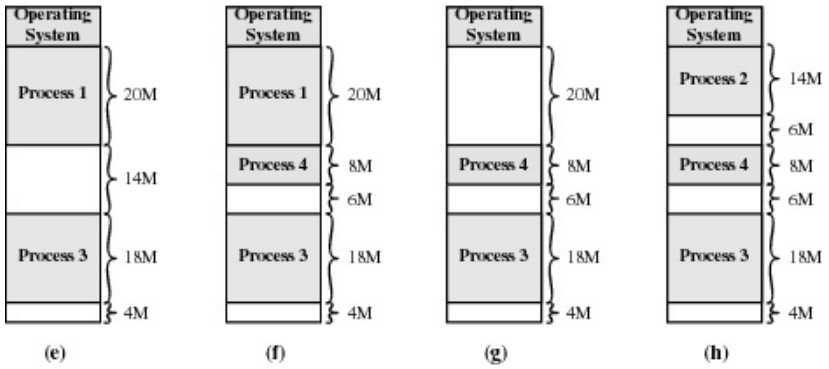
Algoritmos de colocação para partições fixas

Partição dinâmica

- **Inicialmente (q.do não há nenhum processo carregado) ...**
  → existe uma única partição, ocupando toda a memória.
- **Quando é executado um programa ...**
  → alocar zona de memória para o colocar.
- **Idem, para os programas seguintes.**

- **O nº da partições e o seu tamanho é variável**
- **Quando um processo termina, a memória é libertada e pode ser usada para carregar outro programa.**
- **Ao fim de algum tempo existirão fragmentos de memória não utilizada espalhados pela memória do computador (fragmentação externa).**
- **De tempos a tempos a memória terá de ser compactada.**

Partição dinâmica



Fragmentação externa

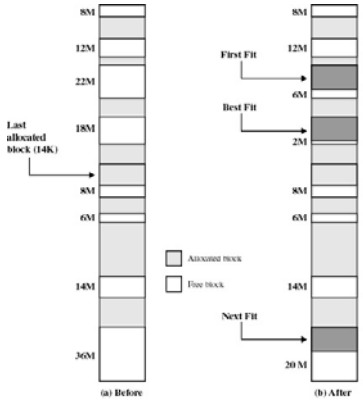
Partição dinâmica

- **Dimensão dos programas**
  - **(+) limitada pela memória física**
  - **(+) não é necessário parar o sistema p/ reconfigurar as partições**
- **Mecanismo de protecção**
  - **semelhante ao da partição fixa**
- **Algoritmo de colocação**
  - ***first-fit*** **- alocar o 1º bloco livre c/ tamanho suficiente**
    (começar pesquisa no 1º bloco livre)
  - ***next-fit*** **- alocar o 1º bloco livre c/ tamanho suficiente**
    (começar pesquisa no 1º bloco livre a seguir àquele em que terminou a últ. pesq.ª)
  - ***best-fit*** **- alocar o bloco livre mais pequeno que tenha tamanho suficiente**
    ⇒ pesquisar a lista de blocos livres toda
  - ***worst-fit*** **- alocar o bloco livre maior**
    (na expectativa de que o que sobra ainda tenha tamanho suficiente p/ ser útil)
  - ***buddy-system*** **- ir dividindo a memória livre, sucessivamente,**
    em blocos de tamanho 2^k (*buddies*- blocos em que se divide o bloco anterior)
    até ter um bloco livre em que o procº caiba c/ menor desperdício

Partição dinâmica

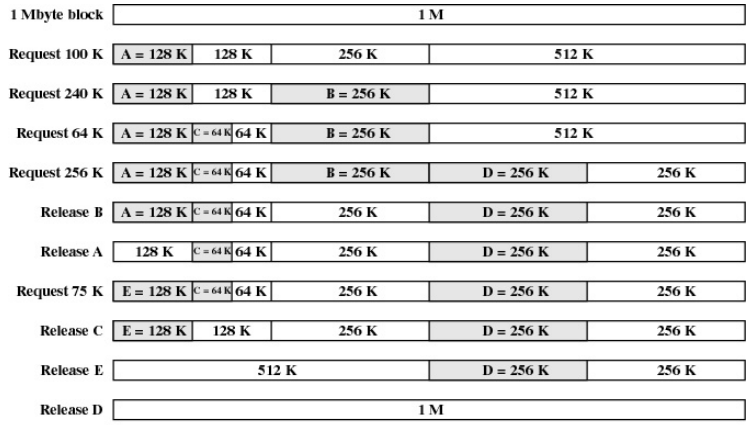
- **Algoritmo de colocação (cont.)**
  - **Qual o melhor ?**
    - » **Depende da sequência de *swapping* de processos e do seu tamanho.**
  - ***first-fit***
    - » **(+) o mais simples**
    - » **(+) usualmente o melhor e o mais rápido**
    - » **(-) usualmente dá origem a muitos blocos livres de pequena dimensão no início da memória**
  - ***next-fit*** **(Stallings,Tanenbaum)**
    - » **resultados de simulação indicam que é ligeiramente pior que o *first-fit* (Tanenbaum)**
  - ***best-fit***
    - » **(-) lento**
  - ***worst-fit***
    - » **(-) em geral, dá maus resultados (simulação)**
  - ***buddy-system*** **(Stallings,Tanenbaum)**
    - » **(+) fácil fazer a junção de 2 *buddies* livres contíguos**
    - » **(-) ineficiente em termos de utilização de memória (ex.: um proc. de 33kB ocupa um bloco de 64KB)**

Partição dinâmica



Configuração da memória antes e depois da alocação de um bloco de 16 MB usando diversos algoritmos de colocação

Partição dinâmica



Exemplo do *buddy-system*

Partição dinâmica

- **Algoritmo de substituição**
  - **quando não há memória livre para carregar um processo, (mesmo após compactação) que processo retirar da memória para ganhar espaço livre ?**
    (v. adiante, a propósito da memória virtual)
- **Problemas:**
  - **fragmentação externa, qualquer que seja o algoritmo de alocação usado**
  - **perda de tempo na gestão de buracos livres muito pequenos (=sem utilidade)**
    - » **ex.: bloco livre de 20000 bytes**
      um processo precisa de 19998 bytes
      solução: alocar pequenos buracos, juntamente c/ o pedido
  - **necessidade de compactação**
    - » **consome tempo**
    - » **⇒ capacidade de recolocação dinâmica**
    - » **difícil arranjar estratégia óptima**
      - **Compactar num único bloco ou em vários blocos grandes ?**
      - **Concentrar os blocos livres num dos extremos da memória ou minimizar os deslocamentos ?**
      - **Quando compactar, sempre que um processo termina ou só quando for necessário ?**

Estruturas de dados usadas na gestão de memória

Mapas de *bits*

- **Dividir a memória em blocos.**
- **A cada bloco é associado um *bit* que indica se ele está ocupado ou não.**
- **Tamanho da memória e dos blocos determinam o nº de *bits* necessários.**
- **Dificuldade : *overhead* necessário p/encontrar o nº de blocos livres consecutivos necessários p/carregar um programa.**

Listas ligadas

- **Manter uma lista duplamente ligada de blocos livres e ocupados (em geral por ordem crescente de endereços)**
- **Vantagem: fácil fazer a junção de 2 blocos livres contíguos, quando um processo liberta memória.**

Paginação

Mecanismos de gestão de memória anteriores:

- **A memória alocada a um processo é contígua.**
- **Problema : utilização ineficiente da memória**
  - » **partição fixa → fragmentação interna**
  - » **partição dinâmica → fragmentação externa**

<u>Paginação</u> :

- **o espaço de endereçamento físico de um processo pode ser não-contíguo**

<u>Objectivos da paginação</u>:

- **facilitar a alocação**
- **facilitar o *swapping***
- **reduzir a fragmentação da memória**

Paginação

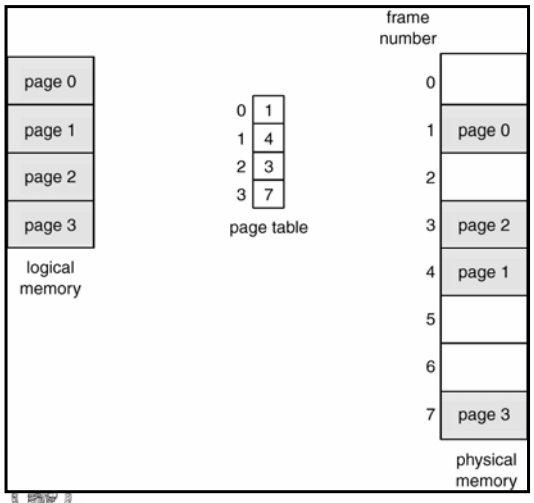
Método básico:

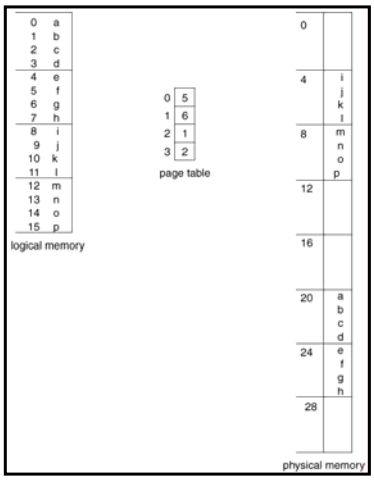
- **Dividir a memória física em blocos de tamanho fixo chamados quadros *(frames)*.**
- **Dividir a memória lógica em blocos de tamanho fixo chamados páginas.**
  - » **a dimensão das páginas é igual à dos quadros**
  - » **a dimensão das páginas depende da arquitectura da máquina**
  - » **algumas máquinas suportam vários tamanhos de página**
- **As páginas constituintes de um processo são carregadas em quaisquer quadros livres.**
- **O S.O. mantém uma tabela de páginas (*page table*), por cada processo, que estabelece a correspondência entre páginas e *frames*.**

***Nota*:**

- **O utilizador continua a ver a memória como um único espaço contíguo.**
- **O mapeamento entre os espaços de endereçamento lógico e físico está escondido do utilizador, sendo feito sob controlo do S.O. c/ o auxílio de *hardware* especial (*MMU*).**

Paginação





Exemplos

Paginação

***Hardware* de suporte**

- **Método geral de tradução de endereços**
  - » **Dividir o endº lógico pelo tamanho da página p/ determinar o nº da página**
  - » **Aceder à tabela de páginas p/ determinar o endº-base do quadro**
  - » **Adicionar o deslocamento (*offset*) dentro da página (resto da divisão anterior) ao endº-base do quadro, para obter o endº físico**

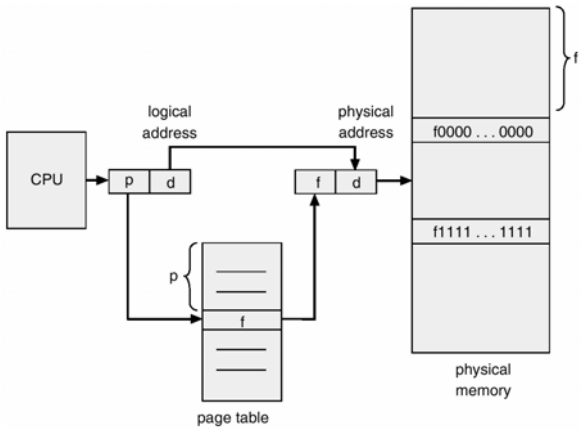
```
EndereçoFísico = TabelaPáginas [ EndereçoLógico DIV TamanhoPágina ] +
               + EndereçoLógico MOD TamanhoPágina
```

- **Na prática**
  - » **Usar tamanhos de página que sejam potências de 2**

    → possibilidade de usar *shifts* para fazer DIV e MOD ou (melhor !)
    → extrair directamente do endereço lógico os *bits* que formam o nº da página e o deslocamento

***Nota:* a paginação é uma forma de recolocação dinâmica**

Paginação



Tradução de endereços lógicos em endereços físicos

Paginação

Multiprogramação com paginação

- O S.O. mantém uma tabela de quadros (*frame table*) c/ a indicação dos quadros ocupados e livres.
- A partir do tamanho do ficheiro executável é determinado o nº de páginas necessário.
- O *long term scheduler* verifica se esse nº de páginas está disponível. Se estiver, constrói uma tabela de páginas p/ o novo processo à medida que carrega o programa.

Vantagens da paginação:

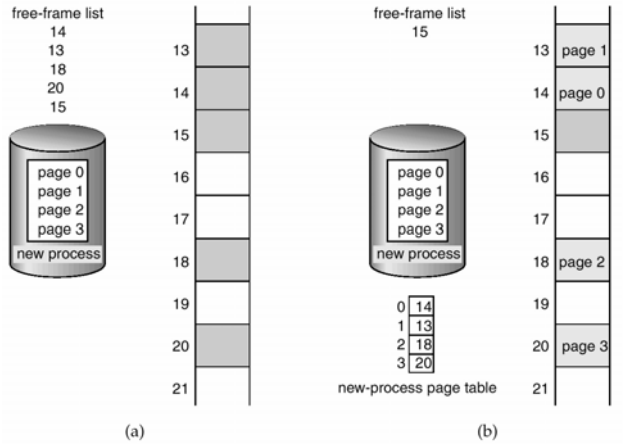
- A alocação é fácil
  - » manter uma lista de quadros livres e alocá-los por qualquer ordem;
  - » facilidade de *swapping* dado que tudo tem o mesmo tamanho
- Elimina a fragmentação externa

Paginação

Dificuldades da paginação

- Eficiência de acesso (*overhead* por cada referência à memória)
  - » Tabelas de página (mesmo q.do pequenas) são, em geral, demasiado grandes p/ carregar na memória rápida da *MMU*.
  - » Pode acontecer que as tab.s de página sejam mantidas em mem. principal e a *MMU* só tenha o endº-base da tabela.
- Espaço ocupado pela tabela
  - » Se as páginas forem pequenas o tamanho da tabela pode ser enorme
    - ex.: espaço de endereçamento de 32 *bits* (4GB= 2^{32}) c/ páginas de 4KB (=2^{12}) e 4 *bytes* por elemento da tabela tabelas de páginas c / 4MB (= $2^{32}/2^{12}*4$)
- Fragmentação interna
  - » Quando o tamanho do processo não é múltiplo do tamanho da página.
  - » Quanto maior for a página maior a fragmentação.
  - » Fragmentação média = 1/2 página.
- Aumento do tempo de comutação de contexto
  - » Necessário carregar a tabela de páginas do processo que vai correr e actualizar certos registos do *hardware*.

Paginação

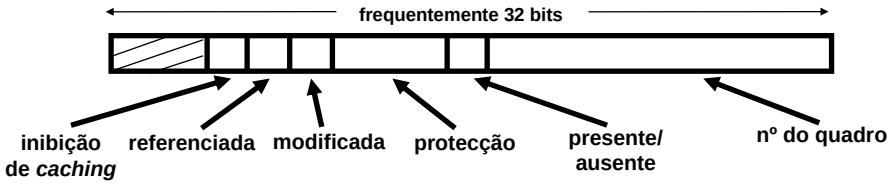


***Frames* livres**

Paginação

Implementação da tabela de páginas

- **Estrutura dos elementos da tabela de páginas**
  - » **Os campos de cada elemento variam, consoante o S.O. mas o tipo de informação presente é sensivelmente o mesmo.**



- **Nº do quadro**
  - **É o campo mais importante (imprescindível)**
- **Presente / ausente (1 *bit*)**
  - **Indica se esta entrada da tabela é válida ou não, isto é, se a página está ou não em memória (↔ memória virtual)**
  - **Uma tentativa de referenciar uma página inválida dá origem a um *trap* p/o S.O.**

Paginação

- **Modificada** (1 *bit*)
  - Indica se a página foi modificada ou não.
  - Importante p/ saber se a pág. tem de ser escrita em disco (↔ memória virtual)
- **Referenciada** (1 *bit*)
  - Indica se a página foi referenciada p/ leitura ou escrita
  - Importante p/ a gestão de memória virtual
- **Protecção** (1 ou mais *bits*)
  - Indica que tipo de acesso é permitido
  - Só 1 *bit* → 0 = Read/Write ; 1 = Read only
  - 3 *bits* → 1 *bit* = Read (enable/disable) ;
    1 *bit* = Write (enable/disable) ;
    1 *bit* = Execute (enable/disable) ;
- **Inibição de *caching*** (1 *bit*)
  - Importante p/ páginas que são mapeadas em registos de dispositivos e não em memória.
    Se o S.O. necessitar de aceder a um dispositivo de I/O é necessário inibir o *caching* de modo a que ele vá buscar a informação ao dispositivo e não à *cache* de memória.

Paginação

Implementação da tabela de páginas

- Varia de sistema operativo para sistema operativo.
- Muitos sistemas operativos usam uma tab. de páginas por processo. No *PCB* é guardado um apontador para a tabela de páginas.
- Se a dimensão da tab. de páginas for pequena
  - » Usar um conjunto de registos para manter a tabela de páginas.
  - » Ex.: DEC PDP-11 (anos 70)
    16 bits de endereço (64KB de memória)
    tamanho da página = 8KB
    ⇒ 8 entradas / tabela de páginas mantidas em registos de acesso rápido
- Se a dimensão da tab. de páginas for grande (a maior parte dos computadores contemporâneos)
  - » A tab. de páginas é mantida em memória principal.
  - » O *Page-Table Base Register* (*PTBR*) aponta p/ a tab. de páginas.
  - » Problema: cada acesso a uma posição de memória implica 2 ref.as físicas à memória.
  - » Solução : usar uma *cache* de acesso rápido onde é mantida informação acerca das páginas acedidas mais recentemente chamada *Translation Look-aside Buffer (TLB).*

Paginação

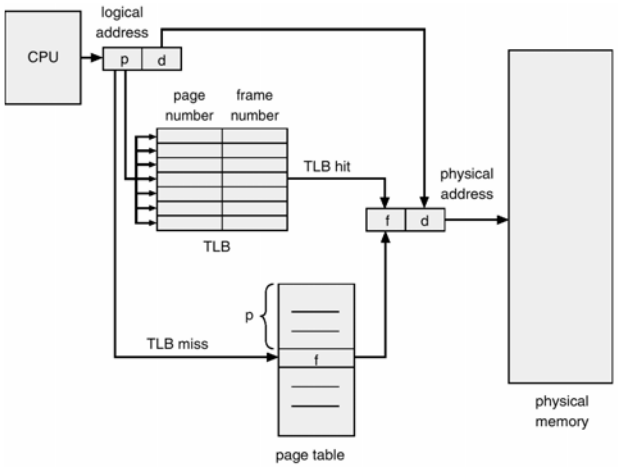
Translation Look-aside Buffer (*TLB*)

- Memória de tipo associativo.
- Cada registo da memória associativa tem 2 campos: uma chave e um valor.
- Quando é apresentado um item aos registos associativos ele é comparado com todas as chaves simultâneamente.
- Se o item for encontrado no campo chave, o valor correspondente é apresentado na saída.
- Se não for encontrado, isso é assinalado ao *hardware* de gestão de memória.

- De facto, a pesquisa na *cache* é lançada em paralelo c/ o acesso à tab. de páginas.
- Se a chave for encontrada na memória associativa, é interrompido o acesso à tabela de páginas.

- *Hit ratio* - percentagem de vezes que o nº da página é encontrada nos registos associativos. Esta percentagem, indicada pelo fabricante do processador é, em geral, muito próxima de 100% (ex: 98%)

Paginação



Paginação* com *TLB

Paginação

Quando as tabelas de páginas são muito grandes ...

» **Ex: endereços lógicos de 32 *bits* (4GBytes de memória) e páginas de 4Kbytes**
⇒ 2^{20} entradas na tabela de páginas

... existem várias soluções:

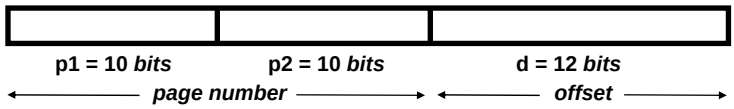
- **Armazenar as tabelas de páginas em memória virtual (cap. seguinte).**
- **Usar paginação multinível.**
- **Usar uma tabela de páginas *hashed.***
- **Usar uma tabela de páginas invertida.**

- **Armazenar as tabelas de páginas em memória virtual (cap. seguinte)**
  - **As próprias tab.s de páginas estão sujeitas a paginação (!)**
  - **Quando um processo está a executar apenas parte da tab. de páginas estará, em geral, em mem. principal.**

Paginação

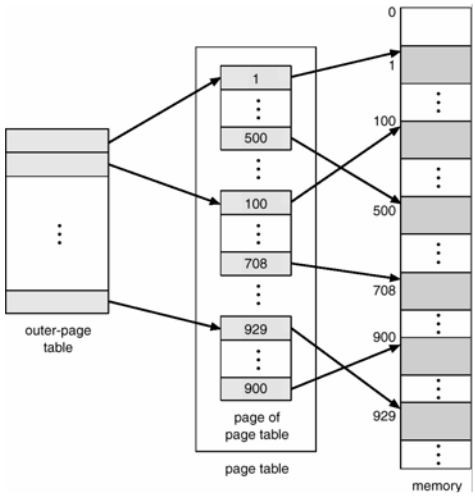
- **Usar paginação multinível**

» **Ex.:**



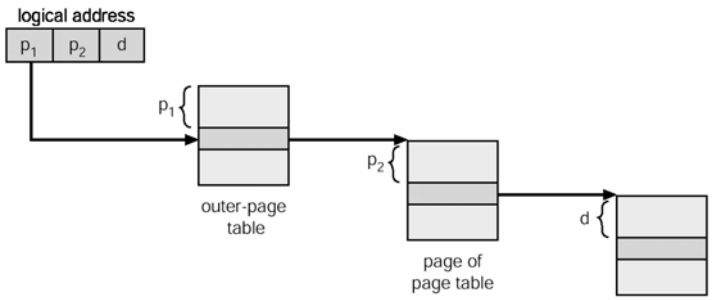
- **Existe um directório de tabelas de páginas com 2^{p1} elementos.**
- **Cada elemento aponta para uma tabela de páginas com 2^{p2} elementos.**
- **Em geral, o comprimento máximo de cada tabela de páginas não pode ser superior à dimensão de uma página.**
- **Vantagem : evitar ter todas as tab.s de páginas em memória, simultâneamente.**

Paginação



Paginação multinível (tabela de páginas c/ 2 níveis)

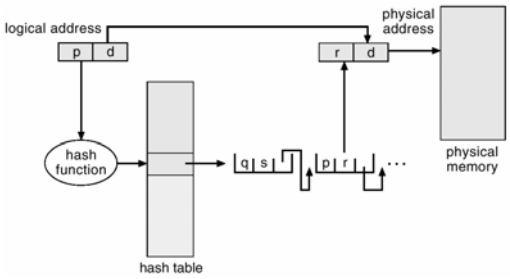
Paginação



Tradução de endereços em paginação multinível

Paginação

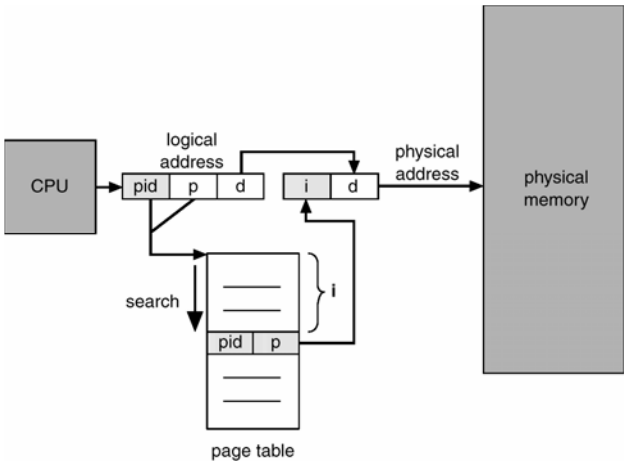
- **Usar uma tabela de páginas *hashed***
  - **Comuns quando o espaço de endereçamento é > 32 bits.**
  - **O número da página é convertido num índice da tabela de páginas.**
    Cada elemento da tabela de páginas aponta para uma lista de páginas que deram origem ao mesmo índice.
    Cada elemento da lista contém, para cada página o nº do quadro respectivo.
  - **A lista é percorrida até encontrar a página, obtendo-se então o nº do quadro**



Paginação

- **Usar uma tabela de páginas invertida**
  - **A tabela tem uma entrada por cada quadro da memória física.**
  - **A informação contida em cada elemento da tabela é:**
    - » **a *PID* do processo a que pertence o quadro**
    - » **o endº virtual da página que está actualmente no quadro.**
  - **Usa-se uma tabela de *hash* p/aceder aos elementos da tab. de pág.s invertida de forma rápida (alternativa: pesquisa sequencial)**
  - **Vantagem :**
    - » **só existe uma tabela de páginas no sistema e a sua dimensão é fixa e mais pequena do que a das tabelas convencionais.**
  - **Desvantagens :**
    - » **A tabela de páginas deixa de conter informação acerca do espaço de endereçamento lógico de um processo (necessária q.do a página referenciada não está em memória) ⇒ manter uma tab. de pág.s convencional, por cada processo, em mem. secundária**
    - » **Aumento do tempo de acesso à memória (devido ao acesso intermédio à tabela de *hash* ou a pesquisa sequencial) Solução : usar memória associativa p/ manter informação acerca dos acessos mais recentes.**

Paginação



Tradução de endereços usando uma tabela de páginas invertida

Paginação

Tamanho da página

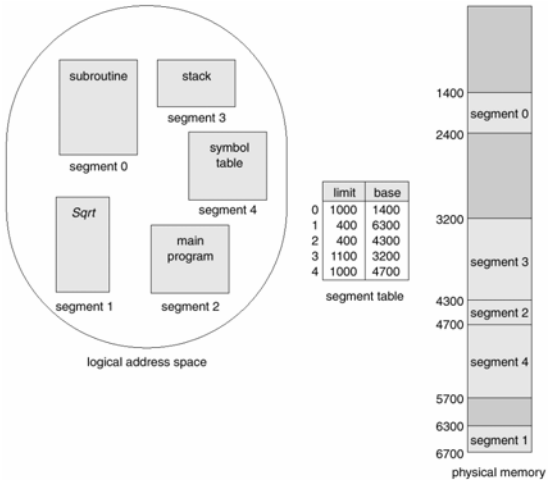
- **Decisão do *designer* do *hardware*.**
- **Factores a considerar:**
  - » **Fragmentação interna**
    - **diminui quando o tamanho da página diminui**
  - » **Nº de páginas / processo ↔ dimensão da tabela de páginas**
    - **o nº de pág.s / proc.º aumenta quando o tamanho da página diminui**
    - **(em sistemas c/ mem. virtual)**
      tab.s de pág.s muito grandes podem implicar uma dupla falta de página
      - uma por falta da parte da tab. de páginas necessária
      - outra por falta da página necessária
  - » **Taxa de falta de páginas (↔ mem. virtual, princípio da localidade)**
    - **pág.s pequenas - taxa baixa**
    - **pág.s grandes - taxa elevada (mas ...pág.s muito grandes albergam o proc.º todo → taxa nula !)**
    - ***Nota*:**
      a taxa depende não só da dimensão das páginas
      mas também do nº de quadros / processo
      (q.do este aumenta a taxa de falta de páginas diminui)

Segmentação

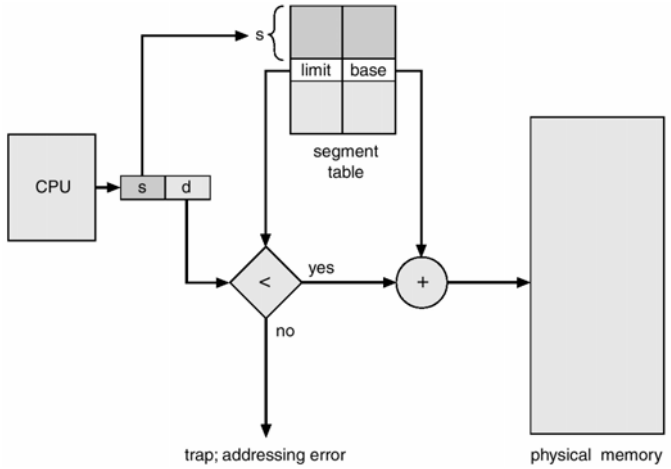
Método básico:

- Dividir o programa e os dados em partes de tamanho diferente (segmentos).
- Um segmento é uma unidade lógica.
  - » Ex.: uma função, um procedimento, as variáveis globais, a *stack*, ...
- Um endereço lógico é constituído por um par <nº do segmento, deslocamento> .
- Os segmentos são carregados em blocos de memória livres, não necessariamente contíguos.
- A tabela de segmentos (uma por cada processo) faz o mapeamento entre os endereços lógicos e os endereços físicos.
- Cada entrada da tabela de segmentos contem:
  - » endereço inicial do segmento
  - » comprimento do segmento
- A tradução de um endereço lógico num endereço físico é feita do seguinte modo:
  - » Extrair, do endereço lógico, o número do segmento (*bit*s mais significativos).
  - » Aceder à tabela de segmentos, usando este número, p/ obter o endereço físico do início do segmento
  - » Comparar o deslocamento (*bit*s menos significativos do endereço lógico) com o comprimento do segmento; se aquele for maior do que este o end.º é inválido.
  - » Endereço físico = endereço físico inicial do segmento+ deslocamento.

Segmentação



Segmentação



Tradução de endereços lógicos em endereços físicos

Segmentação

- A alocação de memória pode ser feita usando um dos métodos estudados na alocação contígua, dinâmica (*first-fit*, *best-fit*, ...).
- A recolocação é feita dinamicamente, recorrendo à tabela de segmentos.
- A paginação é invisível para o programador. A segmentação é usualmente visível.
  O programador ou o compilador coloca o programa e os dados em segmentos diferentes.

Fragmentação da memória:

- Evita a fragmentação interna
- Conduz a fragmentação externa.

Segmentação

Vantagens:

- **Elimina a necessidade de alocação contígua de todo o espaço de endereçamento de um processo (também a paginação).**
- **Facilita a protecção, através de *bits* de protecção associados a cada segmento.**
- **Facilita a partilha.**
  - » **Segmentos partilhados (ex. código) podem ser mapeados no espaço de endereçamento de todos os processos que estão autorizados a referenciá-los.**

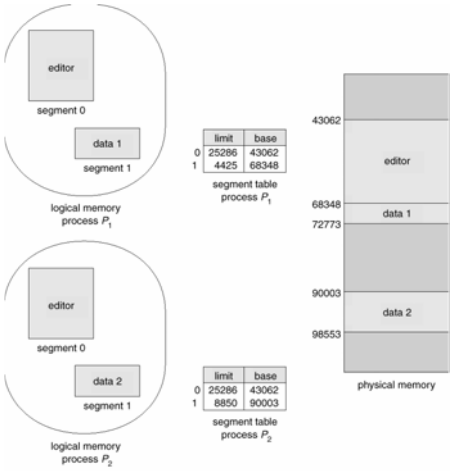
    ***Nota*: é preciso cuidado c/o *swaping* de um segmento partilhado p/vários processos.**
  - » **Partilha de código → poupança de memória**

Desvantagens:

- **Necessidade de compactação.**
- **Necessidade de acessos adicionais à memória p/ obter os endereços físicos (também na paginação).**

A utilização de segmentação simples é cada vez mais rara.

Segmentação



Partilha de segmentos

Segmentação com Paginação

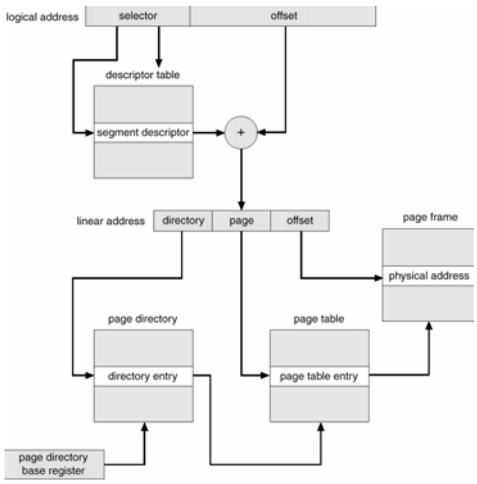
A paginação foi usada para resolver os problemas da partição dinâmica. Porque não aplicar a paginação aos segmentos ?

Método básico:

- **O programador / compilador divide o espaço de endereçamento em segmentos.**
- **Cada segmento é dividido em páginas de tamanho fixo (=tamanho dos quadros da memória física) .**
- **O deslocamento dentro do segmento traduz-se em nº de página + deslocamento dentro da página**
- **Cada segmento tem uma tabela de páginas associada.**

Combina vantagens da paginação e da segmentação.

Segmentação com Paginação



Tradução de endereços no Intel 80386

MEMÓRIA VIRTUAL

- **Introdução**
- ***Demand paging* / Paginação a pedido**
- **Performance da paginação a pedido**
- **Substituição de páginas**
- **Algoritmos de substituição de páginas**
- **Alocação de *frames***
- ***Thrashing***
- ***Demand segmentation* / Segmentação a pedido**

Memória virtual

- **Limitação importante dos mecanismos de gestão de memória descritos anteriormente:**
  - **todo o espaço de endereçamento lógico de um processo deve estar em memória física, simultaneamente.**
- **Isto pode ter um efeito adverso no grau de multiprogramação dado que pode limitar o nº de processos que podem correr simultaneamente.**
- **No entanto, os programas não necessitam de aceder a todo o s/espaço de endereçamento simultaneamente:**
  - **há partes do programa que raramente são executadas**
  - **há dados que raramente/nunca são acedidos**
- **Além disso verifica-se normalmente que as referências ao programa (instruções) e dados tendem a ser localizadas em períodos curtos de tempo (princípio da localidade de referência).**

Memória virtual

Memória virtual:

- **Técnica que permite a execução de processos que podem não estar completamente em memória principal.**
  - » **Um processo pode executar com apenas parte do s/espaço de endereçamento lógico carregado na memória física**
- **O espaço de endereçamento lógico pode pois ser muito maior do que o espaço de endereçamento físico.**
  - » **O utilizador / programador "vê" uma memória potencialmente muito maior - memória virtual - do que a memória real.**

O que é necessário:

- **Divisão de um processo em páginas ou segmentos.**
- **Tradução dos endereços virtuais em endereços reais executada pelo (S.O.+*HARDWARE*) em *run-time*.**
- **Mecanismo de transferência do conteúdo da memória lógica (em disco) para a memória física, à medida que for necessário (*swapping* incremental).**

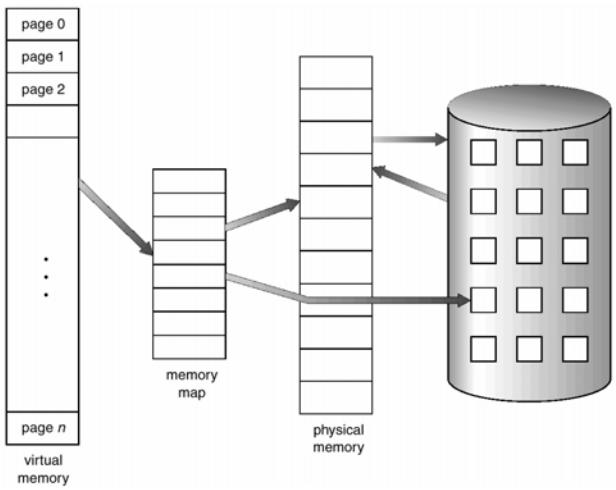


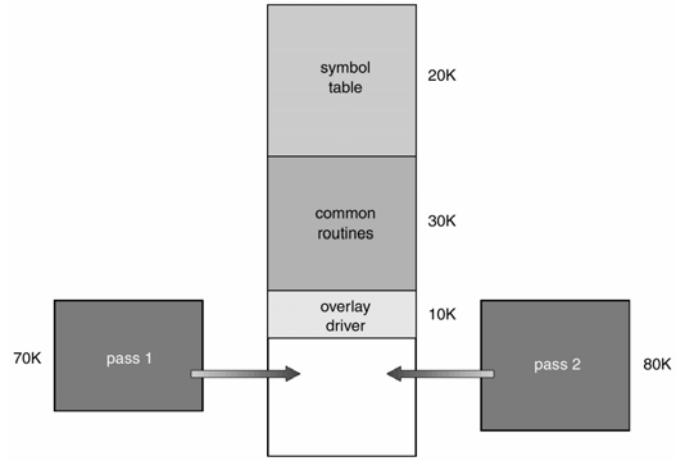
Diagrama mostrando memória virtual maior do que a memória física

Memória virtual

Overlays - técnica de memória virtual usada antigamente

- **Parte da memória era reservada p/uma secção de *overlay*.**
- **Partes do programa, identificadas pelo programador, são compiladas e *linkadas* de modo a poderem correr nos endereços da secção de *overlay*.**
- **Um *overlay driver* (sob controlo do programa) carrega diferentes *overlays* da memória secundária p/ a secção de *overlay*.**
- **O carregamento é feito dinâmicamente: os procedimentos e dados são trazidos p/ memória quando necessário, através de código gerado pelo compilador**
  (ex: a chamada a uma função testa primeiro se ela está em memória)
- **Problema:**
  - » **Os *overlays* não podiam referenciar-se mutuamente.**

Técnica de *overlays*



***Assembler* de 2 passos**

Demand paging / Paginação a pedido

Paginação a pedido

- A maior parte dos S.O's modernos são baseados em paginação a pedido (por necessidade ou por exigência).
- Semelhante à paginação convencional excepto que as páginas só são transferidas p/ a memória principal quando são necessárias.
  Pode conduzir a
  - » redução de *I/O*
  - » resposta mais rápida (só se carregam as páginas necessárias)
  - » redução da memória necessária por processo
  - » maior grau de multiprogramação
  - » mais utilizadores
- Quando é referenciada uma página (um endereço de memória)
  - » se a referência é inválida ⇒ abortar
  - » se a referência é válida e a página não está em memória ⇒ trazê-la p/ memória

Paginação a pedido

Bit de página válida/inválida (ou presente/ausente)

- Cada entrada da tabela de páginas tem um *bit* que indica se a página em questão está ou não em memória (ex.: 1 = presente / 0 = ausente).
- Durante a tradução de endereço (lógico→físico) se o *bit* estiver a 0 ⇒ falta de página .

Falta de página (→ *trap* p/ o S.O.) ⇒

- Verificar na tabela de páginas se a referência é válida ou inválida.
- Referência inválida ⇒ abortar
  Referência válida ⇒ continuar (trazer a página p/ mem. principal) .
- Obter um *frame* livre .
- Ler a página necessária .
- Actualizar a tabela de páginas c/ indicação de que a pág. está em memória, e em que *frame* está.
- Recomeçar a instrução interrompida devido à falta de página.

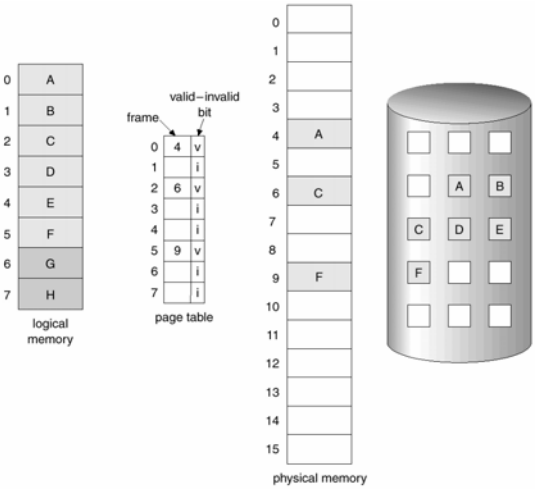
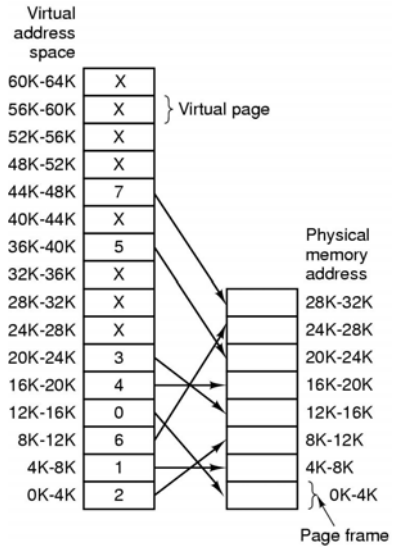
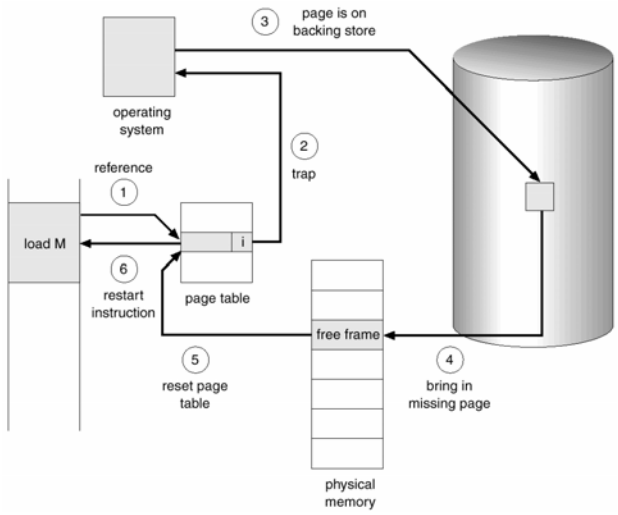


Tabela de páginas quando algumas páginas não estão na memória principal



Relação entre endereços virtuais e endereços físicos dada pela tabela de páginas



Passos no tratamento de uma falta de página

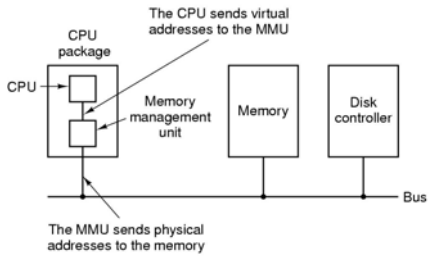
Paginação a pedido

O que acontece se não houver *frames*/quadros livres ?

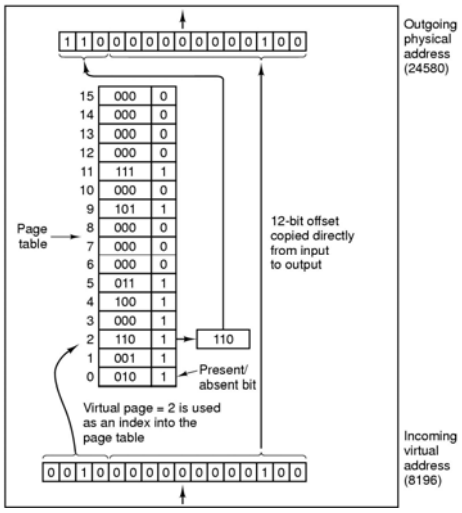
- **Substituição de página -**
  encontrar uma página em memória que não esteja a ser utilizada e fazer o *swap out* dessa página ⇒
  - » **algoritmo de substituição de página que resulte num número mínimo de faltas de página (ver adiante)**

Suporte de *hardware* necessário

- **Tabela de páginas c/ *bit* de página válida / inválida.**
- **Possibilidade de recomeçar uma instrução que falhou devido a uma falta de página ⇒**
  - » **Guardar o estado inicial de uma instrução e repô-lo após o *trap*.**
  - » **Por vezes os processadores guardam um estado de execução parcial e continuam a instrução onde ela foi interrompida.**
  - » **Dificuldade principal: instruções que movimentam blocos de dados.**
- **Disco rápido p/ guardar as páginas que não estão em memória.**



Posicionamento e função da *MMU - Memory Management Unit*



Funcionamento interno de uma *MMU* num sistema com 16 páginas de 4 KB

Performance da paginação a pedido

- **A paginação a pedido pode implicar uma degradação significativa da performance do computador.**

Tempo efectivo de acesso à memória, T_{eam}

$$T_{eam} = (1-p) \times T_{am} + p \times T_{fp}$$

p = taxa de falta de páginas $0.0 \leq p \leq 1.0$
p=0, não ocorrem;
p=1, todas as referências conduzem a falta de página

T_{am} = tempo de acesso à memória

T_{fp} = tempo que o S.O. demora a processar uma falta de página

- **O princípio da localidade de referência indica que p deve ser próximo de 0.**
- **Contudo T_{fp} pode ter um efeito significativo, pois é muito superior a T_{am} .**

Performance da paginação a pedido

Estimação de T_{fp} (tempo de processamento de uma falta de página):

- **tempo de processamento de uma interrupção**
- **tempo de carregamento da página**
- **custo do recomeço da instrução**

Exemplo:

p = 1/1000 (1 falta de pág. em cada 1000 ref.as à memória); T_{am} = 100 ηs; T_{fp} = 25 ms

$\Rightarrow T_{eam} = (1-1/1000)\times 100 + 1/1000\times 25000000\ \eta s \approx 100 + p \times 25000000\ \eta s$

Degradação < 10% ⇒

$100 + 10 > 100 + 25000000\ p \Rightarrow p < 4\times 10^{-7}$ (uma falha em 2.5×10^6 ref.as)

T_{am} $10\%\times T_{am}$ $(1-p)\times 100 \approx 100$

Performance da paginação a pedido

Tempo de acesso a disco

- **O acesso ao espaço de *swap* é em geral mais rápido do que o acesso a um ficheiro normal**
  - » **O acesso é feito directamente, e não através do sistema de ficheiros.**

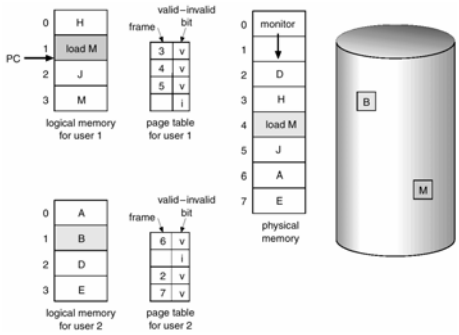
Pode-se melhorar a velocidade de acesso às páginas copiando a imagem do ficheiro p/ o espaço de *swap*, inicialmente, e executar o carregamento das páginas a partir daí.

Outras opções:

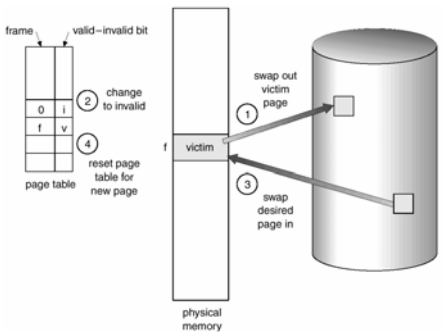
- » **Se o espaço de *swap* for limitado o código do programa pode ser sempre carregado a partir do ficheiro. Quando houver necessidade de substituir uma página de código não há necessidade de a escrever.**
- » **Inicialmente, ir buscar as páginas usando o sistema de ficheiros. As páginas que forem substituídas vão p/ o espaço de *swap*. Se estas forem necessárias posteriormente são carregadas do espaço de *swap*.**

Substituição de páginas

- **Como proceder quando uma página necessitar de ser carregada em memória e não houver nenhum *frame* livre ?**
  - **proceder ao *swap out* de todo um processo**
  - **libertar um *frame* (a melhor opção)**
- **Se a página que está no *frame* libertado tiver sido modificada desde o s/ último carregamento ⇒ escrever a página no disco**
  - **a escrita no disco pode duplicar o T_{fp} (⇒ 1 escrita + 1 leitura)**
- **Se não tiver sido modificada (página de código ou *read only*) o *frame* pode ser usado à vontade.**
- **Um *dirty bit / modified bit*, na tabela de páginas, pode ser usado para saber se a página a retirar foi modificada**
  - **activado pelo *hardware* por cada escrita no *frame***



Necessidade de substituição de página



Substituição de página

Substituição de páginas

A substituição de páginas completa a separação entre a memória lógica e a memória física:

- **uma memória virtual grande pode ser fornecida com base numa memória física mais pequena**

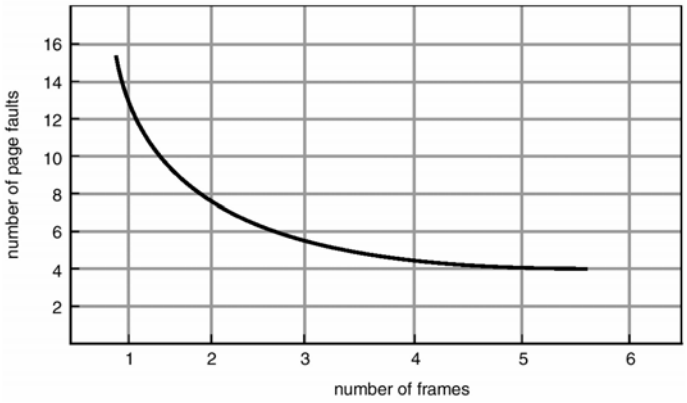
Decisões fundamentais a tomar

- **Quantos *frames* atribuir a um processo ?**
  (ALGORITMO DE ALOCAÇÃO DE *FRAMES*)
- **Que páginas substituir e quando ?**
  (ALGORITMO DE SUBSTITUIÇÃO DE PÁGINAS)

Algoritmos de substituição de páginas

- **Que algoritmo usar p/ seleccionar a página a substituir ?**
  - ***FIFO***
  - **Óptimo**
  - ***LRU* e *LRU* aproximado**
  - **Baseados em contagens (*LFU* e *MFU*)**
- **Como avaliar um algoritmo ?**
  - **Avaliam-se os algoritmos aplicando-os a uma determinada sequência de referências à memória e determinando o nº de faltas de página nessa sequência.**
  - **Sequência (basta o nº das páginas referenciadas)**
    - » **aleatória**
    - » **seguimento das referências de um sistema real**
  - **Determinação do número de faltas de página ⇒ conhecer o nº de *frames* disponíveis**

    Em geral, se
    nº de *frames* ↑ ⇒ nº de faltas de página ↓



Evolução do nº de faltas de página
em função do nº de *frames* atribuídos a um processo

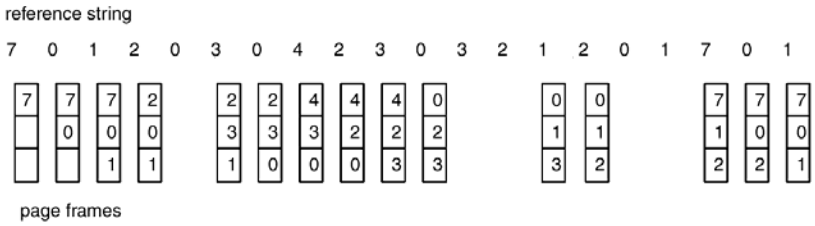
Algoritmos de substituição de páginas

ALGORITMO *FIRST-IN FIRST-OUT* (*FIFO*)

- **A página substituída é a que estiver há mais tempo em memória.**
- **Implementação eficiente ⇒**
  - » **manter uma lista *FIFO* com as páginas que estão em memória**
  - » **substituir a página à cabeça da lista**
  - » **inserir a página carregada no fim da lista**
- **Problema:**
  - » **uma página frequentemente referida pode ser retirada, se tiver sido carregada há muito tempo**

ALGORITMO *FIRST-IN FIRST-OUT* (*FIFO*)

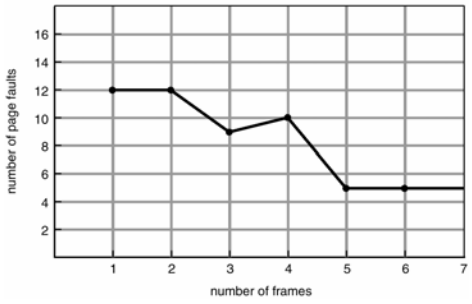
- **Exemplo:**
  - » **sequência: 70120304230321201701**
  - » ***frames*: 3**
  - » **faltas de página: 15**



Algoritmos de substituição de páginas

ALGORITMO *FIRST-IN FIRST-OUT* (*FIFO*) (cont.)

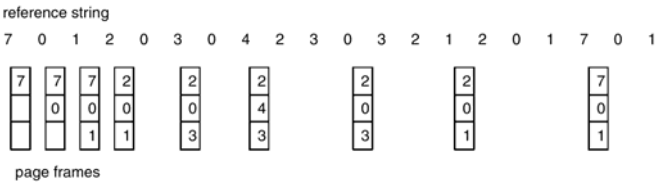
- **Performance**
  - » **Altamente influenciada pelo nº de frames disponíveis.**
  - » **Exemplo:**
    - **sequência anterior**
    - ***frames* = 4**
    - **faltas de página = 10**
- **Anomalia de Belady** →
  - » **Mais *frames* ⇏ Menos faltas de página**
  - » **Exemplo:**
    - **sequência: 123412512345**
    - **3 *frames* ⇒ 9 faltas de página**
    - **4 *frames* ⇒ 10 faltas de página**



Algoritmos de substituição de páginas

ALGORITMO ÓPTIMO

- **Substituir a página que não será usada por um período de tempo mais longo, no futuro (!!!).**
- **Problema:**
  - » **Como saber isso ?!**
- **Exemplo:**

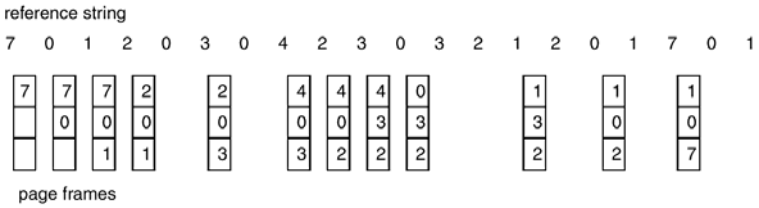


- **Algoritmo frequentemente usado para avaliar a performance de outros algoritmos.**

Algoritmos de substituição de páginas

ALGORITMO *LEAST-RECENTLY-USED* (*LRU*)

- **A página substituída é a que não foi referenciada há mais tempo (a página usada menos recentemente)**
- **É uma aproximação ao algoritmo óptimo**
  - » **usar o comportamento passado p/ prever o futuro**
- **Exemplo:**



Algoritmos de substituição de páginas

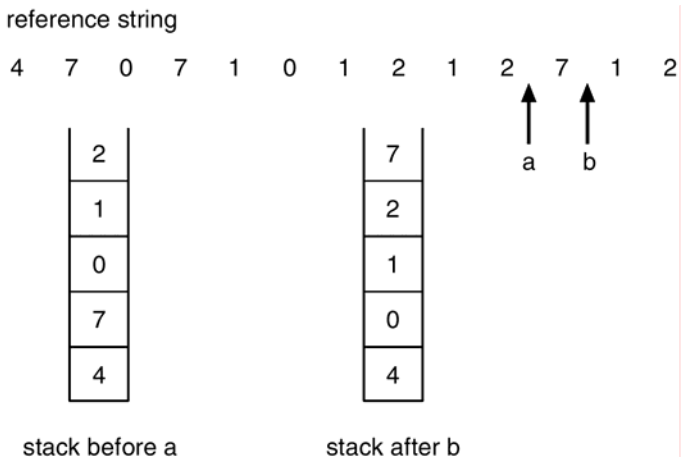
ALGORITMO *LEAST-RECENTLY-USED* (*LRU*)

- **Algoritmo usado frequentemente (/ algoritmos que o aproximam). Comportamento bastante bom.**
- **Problema:**
  - » **manter e actualizar o tempo em que uma página foi referenciada.**
    ⇒ gastos de tempo e de espaço

Algoritmos de substituição de páginas

ALGORITMO *LEAST-RECENTLY-USED* (*LRU*) **(cont.)**

- **Implementação**
  - » **Baseada em contadores**
    - **Associar um campo "tempo" a cada página da tabela de páginas.**
    - **De cada vez que a página é referenciada, copiar um *clock* p/ aquele campo.**
    - **Substituir a página com o "tempo" mais baixo (referênciada há mais tempo).**
    - **Problemas:**
      - – **Pesquisa p/ encontrar a página a substituir.**
      - – **Actualização do campo "tempo" por cada referência à memória.**
  - » **Baseada numa *stack***
    - **Manter uma lista duplamente ligada c/ os números das páginas referenciadas.**
    - **De cada vez que uma página é referenciada, colocá-la no topo da lista**
    - **Problema: actualização de diversos apontadores por cada ref.a à memória.**
    - **Vantagem: a decisão da página a substituir é rápida (pág. do fim da lista).**



Utilização de uma *stack*
para guardar as páginas referênciadas mais recentemente

Algoritmos de substituição de páginas

ALGORITMOS QUE APROXIMAM O ALGORITMO *LRU*

- **A implementação do algoritmo *LRU* é "pesada".**
  Poucos sistemas implementam o *LRU* real.
- **Aproximações:**
  - » **Utilizando o *bit* de referência.**
  - » **Utilizando o algoritmo da 2ª oportunidade ou do relógio.**
  - » **Utilizando o algoritmo da 2ª oportunidade melhorado.**

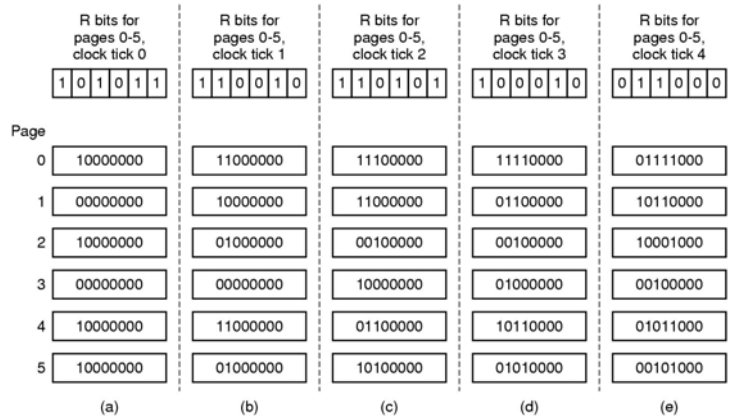
APROXIMAÇÃO AO *LRU* USANDO O *BIT* DE REFERÊNCIA

- **Algoritmo:**
  - » **Associar a cada página um *bit* de referência (inicialmente=0).**
  - » **Quando uma página é referenciada, colocar o *bit* =1 .**
  - » **Substituir uma página com o *bit* = 0 (se existir)**
    mas impedir a substituição de uma pág. carregada recentemente
- **Dificuldade:**
  - » **Não se sabe a ordem pela qual as páginas foram referenciadas.**

Algoritmos de substituição de páginas

APROXIMAÇÃO AO *LRU* USANDO O *BIT* DE REFERÊNCIA (cont.)

- **Variante (melhoria do algoritmo anterior):**
  - » **Usar um registo de *bits* de referência (ex: 8 *bits*) por cada página da tabela de páginas.**
  - » **Com intervalos regulares introduzir o *bit* de referência de cada página no *bit* mais significativo do registo respectivo, deslocando o s/ conteúdo p/ a direita, e limpar todos os *bits* de referência da tabela de páginas.**
  - » **Considerando o conteúdo do registo como um número em binário a página com o número mais baixo é a que foi acedida há mais tempo.**
- **Dificuldades desta variante:**
  - » **Perde-se o que se passa entre 2 *ticks* de *clock*.**
  - » **Perde-se as referências feitas há mais tempo do que *N×Intervalo*, em que N=nº de *bits* do registo**

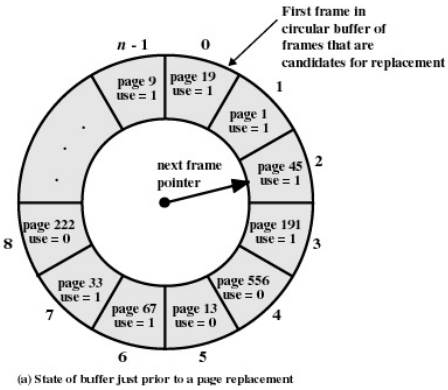


Aproximação ao *LRU*
utilizando um registo de *bits* de referência

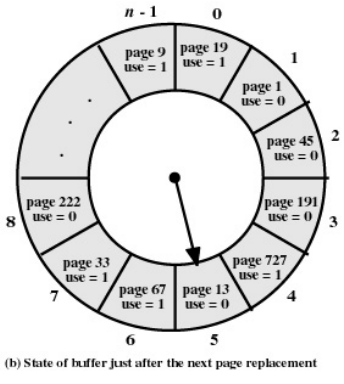
Algoritmos de substituição de páginas

APROXIMAÇÃO AO *LRU* USANDO O ALGORITMO DA 2ª OPORTUNIDADE OU DO RELÓGIO

- **Algoritmo:**
  - » **Manter os *frames* numa lista circular.**
  - » **Quando uma página é carregada pela 1ª vez o *bit* de referência é colocado em 1.**
  - » **Quando é necessário substituir uma página, percorrer os *frames* circularmente e a qualquer *frame* que tenha o *bit* de referência igual a 1 é dada uma 2ª oportunidade, colocando o *bit* de referência igual a 0, e avançando p/ o *frame* seguinte.**
  - » **Se o *bit* de referência ainda estiver em 0 na 2ª passagem, a página é substituída.**



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Algoritmo da 2ª oportunidade ou do relógio

Algoritmos de substituição de páginas

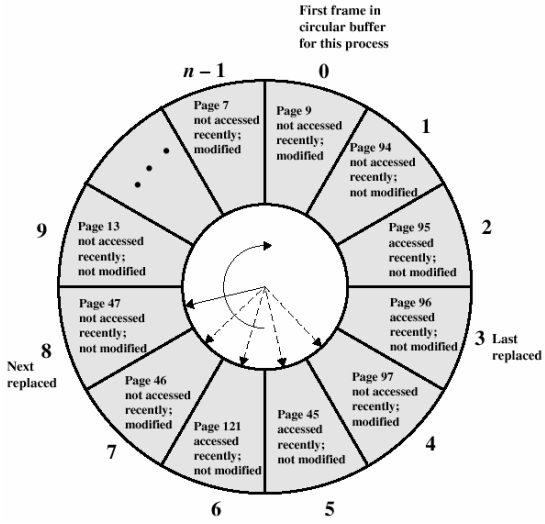
APROXIMAÇÃO AO *LRU* USANDO
O ALGORITMO DA 2ª OPORTUNIDADE MELHORADO

- **Usar 2 *bits* por cada *frame***
  - » **R - *frame* referenciado**
  - » **M - *frame* modificado**
- **4 classes possíveis de *frames***
  - » **R=0, M=0**
    - **página nem referenciada recentemente, nem modificada**
    - **corresponde a uma das melhores páginas para substituir**
  - » **R=0, M=1**
    - **esta página não é muito boa p/ substituir pois tem de ser escrita**
  - » **R=1, M=0**
    - **esta página, provavelmente (?), vai ser referenciada outra vez, a seguir**
  - » **R=1, M=1**
    - **esta página, provavelmente, vai ser referenciada outra vez, a seguir**
    - **e tem de ser escrita no disco, se for substituída**

Algoritmos de substituição de páginas

APROXIMAÇÃO AO *LRU* USANDO
O ALGORITMO DA 2ª OPORTUNIDADE MELHORADO (cont.)

- **Algoritmo**
  - » **1-**
    Percorrer a lista circular, começando na posição actual do apontador.
    Durante este percurso, não alterar o *bit* R.
    O 1º *frame* encontrado c/ (R=0,M=0) é seleccionado p/ ser substituído.
  - » **2-**
    Se o passo 1 falhar,
    percorrer a lista à procura de um *frame* c/ (R=0,M=1).
    O 1º *frame* encontrado é seleccionado p/ ser substituído.
    Durante esta passagem,
    fazer R=0 em todos os *frames* por onde passar.
  - » **3-**
    Se o passo 2 falhar,
    o apontador deve ter retornado à sua posição original.
    Voltar ao passo 1. Desta vez encontrar-se-á um *frame* p/ substituir.



Algoritmo da 2ª oportunidade melhorado

Algoritmos de substituição de páginas

ALGORITMOS BASEADOS EM CONTAGENS

- Manter um contador do nº de referências feitas a cada página.
- ALGORITMO *LFU - Least Frequently Used*
  - » Substituir a página c/ a contagem mais pequena
  - » Problema:
    - as páginas que foram muito usadas há muito tempo são mantidas em memória
  - » Solução:
    - técnica de envelhecimento: dividir a contagem por 2 c/ intervalos regulares
- ALGORITMO *MFU - Most Frequently Used*
  - » Substituir a página com contagem mais elevada (!!!)
  - » Pressuposto:
    - talvez as páginas c/ contagens mais baixas sejam mais recentes e venham a ser necessárias (?!)
- Estes algoritmos são pouco utilizados.

Buffering de páginas

- Usado c/ um qualquer dos algoritmos de substituição de páginas.
- Uma página substituída
  não é imediatamente retirada da memória interna
  é colocada numa de 2 listas (a página não é deslocada na memória):
  - lista de páginas livres (*FIFO*), se a página não foi modificada
  - lista de páginas modificadas (*FIFO*), se a página foi modificada
- Quando é preciso carregar uma página em memória
  usa-se o *frame* do início da lista de páginas livres.
- As páginas da lista de páginas modificadas
  são escritas em disco, quando o dispositivo de paginação estiver livre.
  - » Vantagem: podem ser escritas em grupos (mais rápido)
- Vantagem do *buffering*:
  - Possibilidade de evitar ir buscar uma página ao disco
    se ela ainda estiver no *buffer* (*cache*) de páginas substituídas.
  - Permite ler uma pág. do disco sem que a pág. substituída seja escrita no disco.

Alocação de *frames*

- Como distribuir o número fixo de *frames* da memória
  pelos vários processos ?
- Questões a resolver:
  - Nº mínimo de *frames* por processo ?
  - Algoritmo de alocação ? Alocação fixa ou variável ?
  - Alcance da substituição ? Local ou global ?
- Alguns factores a ter em conta:
  - Quando a quantidade de memória alocada a cada processo diminui
    - nº de processos em memória pode aumentar
    - probabilidade de encontrar um processo pronto a correr pode aumentar
    - *swapping* pode diminuir
  - Quando o nº de páginas de um processo em memória diminui
    - a taxa de falta de páginas pode aumentar
  - A partir de um certo tamanho,
    o aumento da memória alocada a um processo
    não tem nenhum efeito notável na taxa de falta de páginas.

Alocação de *frames*

NÚMERO MÍNIMO DE *FRAMES* POR PROCESSO

- **Cada processo necessita de um nº mínimo de *frames* em memória.**
- **O nº mínimo depende da arquitectura do processador:**
  - » **É o nº máximo de páginas que podem ser acedidas numa única instrução**
  - » **Exemplo:**
    - **Uma instrução MOVE,**
      que permite copiar um bloco de dados de uma zona de memória p/ outra,
      podendo os endereços ser indicados de forma indirecta
    - **a instrução ocupa 6 bytes …….. ⇒ pode abranger 2 páginas**
      end.º de origem indicado indirectamente ⇒ pode abranger 2 páginas
      endº de destino indicado indirectamente ⇒ pode abranger 2 páginas
      ----
      6 páginas
- **O nº máximo é limitado pela memória física.**

Alocação de *frames*

ALGORITMOS DE ALOCAÇÃO

- **Alocação fixa**
  - **O nº de *frames* alocados a um processo é fixo.**
  - **Quando é necessário substituir uma página,**
    é escolhido um dos *frames* pertencentes ao processo.
  - **O nº de *frames* é decidido quando o processo é carregado,**
    podendo ser determinado com base em:
    - » **tipo de processo (interactivo, *batch*, tipo de aplicação, …)**
    - » **informação do utilizador ou do gestor do sistema**
  - **Variantes:**
    - » **partição igual: cada processo recebe *int(NFrames/NProcs) frames***
    - » **partição proporcional: baseada no tamanho ou na prioridade dos processos**
- **Alocação variável**
  - **O nº de *frames* alocados a um processo pode variar**
    durante a sua existência, de acordo com
    - » **grau de multiprogramação (grau de multiprog. ↑ ⇒ nº pág.s / processo ↓)**
    - » **tamanho e/ou prioridade dos processos**
    - » **taxa de falta de páginas (taxa de falta de pág.s ↑ ⇒ nº pág.s / processo ↑)**

Alocação de *frames*

ALCANCE DA SUBSTITUIÇÃO

- **Ao seleccionar uma página p/ substituir, o algoritmo de substituição pode fazê-lo c/ alcance local ou global.**

- **Alcance local**
  - **Escolher a página a substituir entre as páginas residentes do processo que originou a falta de página.**
  - **Desvantagem:**
    - » **um processo pode "embargar" outros processos ao não ceder *frames* de que pode não estar a precisar.**

- **Alcance global (mais comum)**
  - **Qualquer uma das páginas residentes pode ser substituída, mesmo que pertença a outro processo.**
  - **Vantagem:**
    - » **um processo prioritário pode retirar páginas a outros.**
  - **Inconveniente:**
    - » **o conjunto de páginas de um processo em memória depende do comportamento dos outros processos (o tempo de execução pode variar muito de uma vez p/ outra).**

Thrashing

Thrashing

- **acontece quando um processo passa mais tempo em actividades de paginação do que a executar.**

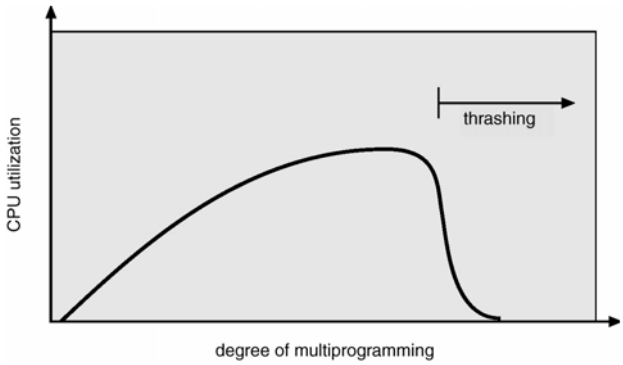
Causa

- **Uma má política de paginação.**
- **Se um processo não tiver páginas suficientes em memória a taxa de falta de páginas é muito elevada ⇒**
  - » **elevada actividade de transferência de páginas**
  - » **baixa utilização do processador**
  - » **baixa utilização de outros dispositivos**

  ← **sintomas de *thrashing***

  - » **o S.O. "pensa que" pode aumentar o grau de multiprogramação**
  - » **outro processo é acrescentado ao sistema**
  - » **a utilização do processador baixa ainda mais, …etc**
  - » **… colapso !!!**

- **O *thrashing* ocorre porque o número de páginas que são activamente usadas é superior ao tamanho total da memória.**



Thrashing

Thrashing

Solução

- **Fazer o *swap out* de um ou mais processos.**
  (decisão do *medium term scheduller*)
- **Retomar a sua execução quando houver mais memória livre.**
- **Por vezes, impõe-se limites mínimos ao tempo que um processo tem de estar em memória ou em disco.**
  - » **Um processo em estado executável tem de permanecer pelo menos T1 segundos em memória antes de ser *swapped out*.**
  - » **Um processo em disco tem de permanecer aí pelo menos T2 segundos antes de ser *swapped in*.**

Limitar os efeitos do *thrashing*

- **Algoritmo de substituição local de páginas**
  - » **Evita que se um processo entrar em *thrashing*, outros também entrem.**
  - » **No entanto, basta que um processo entre em *thrashing* para que o tempo efectivo de acesso à memória dos outros processos aumente.**

Como evitar o *thrashing* ?

- **Estratégia do conjunto de trabalho (*working-set strategy*).**
- **Estratégia da frequência de falta de página.**

Estratégia dos conjuntos de trabalho
(*Working-set strategy*)

Trata de determinar simultâneamente
- **Quantos *frames* alocar a um processo.**
- **Que páginas manter nesses *frames.***

Objectivo:
- **Evitar o *thrashing*,**
  mantendo um grau de multiprogramação tão alto quanto possível.

A ideia:
- **Usar as necessidades recentes de um processo**
  para adivinhar as necessidades futuras (reduzir a taxa de falta de páginas)
  baseado no princípio da localidade de referência.

Estratégia dos conjuntos de trabalho

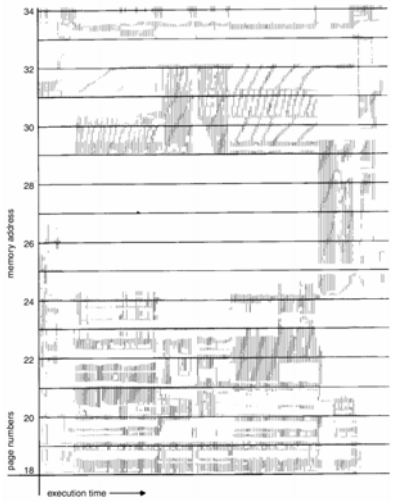
Conjunto de trabalho *(working set)* de um processo
num instante t → W(t,Δ)
- **conjunto de páginas referenciadas nas últimas Δ referências à memória**

O nº ideal de *frames* a atribuir a um processo é
o necessário para guardar o seu conjunto de trabalho.

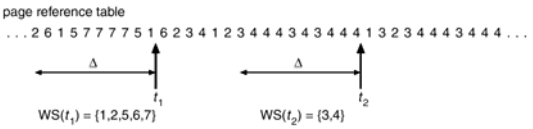
Procedimento:
- **Monitorizar o conjunto de trabalho de cada processo.**
- **Um processo nunca será executado a não ser que**
  o seu conjunto de trabalho esteja em memória principal.
- **Uma página não pode ser removida da memória**
  se fizer parte do conjunto de trabalho de um processo.

  WSSi = dimensão do conjunto de trabalho do processo Pi
  D=Σ WSSi = necessidade total de *frames;*
  N = nº total de *frames* existentes

- **Se D>N suspender um processo para evitar o *thrashing.***
- **Se D<N pode-se iniciar um novo processo.**



Localidade num conjunto de referências à memória



Estratégia dos conjuntos de trabalho

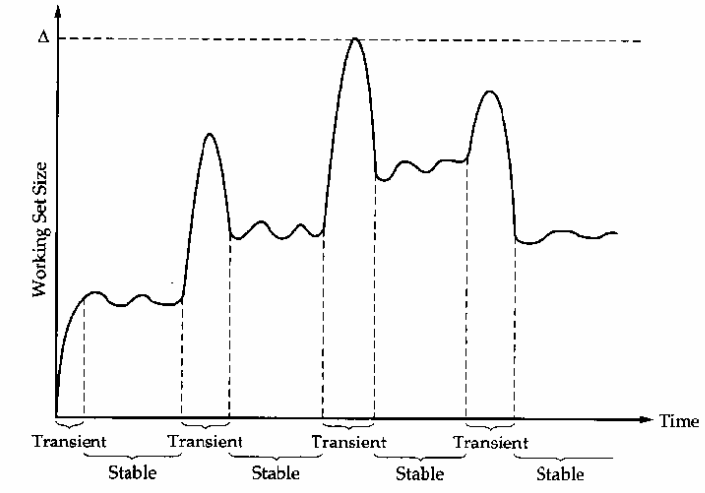
Estratégia dos conjuntos de trabalho

Problemas:

- **O passado nem sempre ajuda a prever o futuro**
  - » **O conjunto de trabalho pode variar com o tempo alternando períodos de alguma estabilidade com períodos de mudança brusca.**
- **Dificuldade em manter actualizado o conjunto de trabalho**
  - » **Necessária uma "janela móvel" sobre a lista de referências à memória.**
  - » **Aproximação:**
    - ***timer*** **que gera interrupções +** ***bit*** **de referência + registo de n** ***bits*****, por página.**
    - **Copiar o** ***bit*** **de referência para o registo, a intervalos regulares.**
    - **Se um dos** ***bits*** **do registo estiver activado a página pertence ao conjunto de trabalho do processo**
    - **<u>Dificuldade</u>: não se consegue saber o que se passou entre interrupções.**
    - **<u>Solução (…)</u> : aumentar o tamanho do registo e reduzir o intervalo entre interrupções.**
- **Determinar o valor óptimo de Δ**
  - » **Δ demasiado pequeno**
    - **pode não englobar toda uma** ***localidade*** **(pág.s activamente usadas, em conjunto)**
  - » **Δ demasiado grande**
    - **pode englobar várias** ***localidades*** **e abranger mais pág.s do que o necessário**

Sequence of Page References	Window Size, Δ			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Evolução do conjunto de trabalho de um processo em função do tamanho da janela



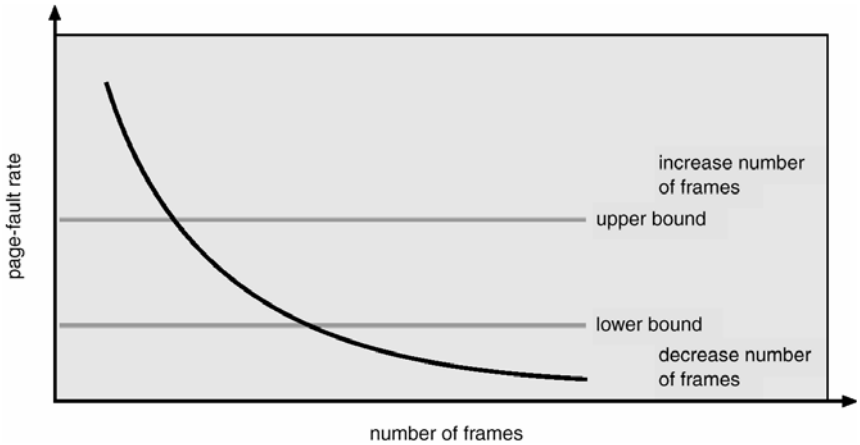
Evolução típica do conjunto de trabalho de um processo

Estratégia da frequência de falta de página

FREQUÊNCIA DE FALTA DE PÁGINA:
Estratégia para evitar o *thrashing*
mais simples do que a dos conjuntos de trabalho.

Procedimento:

- **Monitorizar a frequência de falta de páginas de um processo.**
- **Estabelecer um gama de frequências aceitáveis.**
- **Acima de uma certa frequência atribuir mais um *frame* ao processo; se não houver *frames* disponíveis, suspender o processo.**
- **Abaixo de uma certa frequência, retirar um *frame* ao processo.**



Estratégia da frequência de falta de página

Outras considerações

Além dos
algoritmos de substituição de páginas e
da estratégia de alocação de *frames*
há outros factores a ter em conta:

- **Deve usar-se pré-paginação ?**
- **Qual o tamanho de página mais adequado ?**
- **Como é que a estrutura de um programa pode influenciar a sua performance, tendo em atenção a existência de paginação ?**
- **Será conveniente proceder à fixação de algumas páginas em memória ?**

Pré-paginação

A paginação a pedido "pura"
conduz a elevado nº de faltas de página

- **quando um processo começa a correr;**
- **quando o processo retoma a execução, após um *swap out*.**

Pré-paginação

- **Procura evitar este elevado nº de faltas de página carregando mais páginas do que as exigidas pela falta de página, procurando aproveitar o facto de, o carregamento consecutivo poder ser mais rápido do que o individual (se as pág.s estiverem em posições consecutivas do disco)**

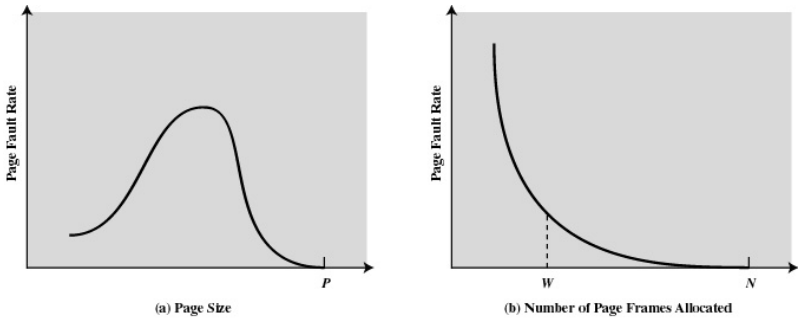
Interesse duvidoso

- **Pode ser vantajoso em algumas situações.**
- **Será o seu custo menor do que servir as faltas de página ?**
- **Pode acontecer que muitas das páginas carregadas não venham a ser usadas !**

Tamanho da página

- **Usualmente determinado pelo *hardware*.**
  (Alguns processadores admitem tamanhos de página variáveis.)
- **Não existe um tamanho ideal.**
- **Argumentos a favor de <u>páginas pequenas</u>:**
  - **Reduz a fragmentação interna.**
  - **Permite isolar mais facilmente a memória que é efectivamente necessária**
    (↔princípio da localidade de referência)
  - **Reduz a *I/O* necessária.**
  - **Permite que a memória ocupada por um processo possa ser reduzida (relativamente a quando as páginas são grandes)**
- **Argumentos a favor de <u>páginas grandes</u>:**
  - **Reduz o tamanho da tabela de páginas.**
  - **A *I/O* é mais eficiente.**
    (o *overhead* devido ao posicionamento da cabeça do disco pode pesar significativamente no tempo total de *I/O* de uma pág. pequena)
  - **Reduz o nº de faltas de página, a partir de certa dimensão das páginas.**

Tendência histórica: aumentar o tamanho das páginas



P = size of entire process
W = working set size
N = total number of pages in process

Evolução típica da taxa de falta de páginas de um programa em função: a) do tamanho das páginas; b) do nº de quadros alocados

Estrutura de um programa

- A paginação a pedido é transparente para o programador.
  No entanto, a performance de um programa pode ser melhorada se o programador estiver consciente do modo como é feita a paginação.
- Exemplo:
  - Programa em C
  - int A[1024][1024]; ← armazenado linha a linha
  - 1 inteiro = 4 bytes
  - *Frames* de 4KB; 1 *frame* pode conter 1 linha da matriz (1024×4 bytes)
  - Programa 1

```
for (j=0; j<1024; j++)
    for (i=0; i<1024; i++)
            A[i][j] = 0;
```

⇒ O 1º elemento A[i][j] acedido está numa página, o 2º está noutra página, etc.

  - Programa 2

```
for (i=0; i<1024; i++)
    for (j=0; j<1024; j++)
            A[i][j] = 0;
```

⇒ Os primeiros 1024 elementos A[i][j] podem estar todos na mesma página, os segundos 1024 elementos podem estar todos noutra página, etc.

Estrutura de um programa

- Uma selecção cuidadosa das estruturas de dados e das estruturas de programação pode reduzir o nº de faltas de página e o nº de páginas no conjunto de trabalho.
  - *stack* - boa localidade de referência
  - tabela de *hash* - má localidade de referência
  - utilização de apontadores - tende a introduzir má localidade de referência
- A linguagem de programação utilizada também pode influenciar.
  - ex.: certas linguagens fazem uso intensivo de apontadores

Estrutura de um programa

- A paginação a pedido é transparente para o programador.
  No entanto, a performance de um programa pode ser melhorada se o programador estiver consciente do modo como é feita a paginação.
- Exemplo:
  - Programa em Pascal
  - Var A: Array [1024,1024] of integer; ← armazenado linha a linha
  - 1 inteiro = 4 bytes
  - *Frames* de 4KB; 1 *frame* pode conter 1 linha da matriz (1024×4 bytes)
  - Programa 1

```
For j:=1 to 1024 do
   For i:=1 to 1024 do
         A[i,j] := 0;
```

⇒ O 1º elemento A[i,j] acedido está numa página, o 2º está noutra página, etc.

  - Programa 2

```
For i:=1 to 1024 do
   For j:=1 to 1024 do
         A[i,j] := 0;
```

⇒ Os primeiros 1024 elementos A[i,j] podem estar todos na mesma página, os segundos 1024 elementos podem estar todos noutra página, etc.

Fixação de páginas

- Alguns *frames* podem ser "fechados" (*locked*)
  isto é, as páginas neles contidas não podem ser substituídas
  ⇒ usar um ***lock bit***.
- Exemplo:
  - Grande parte do núcleo do S.O. e das estruturas de dados do S.O. .
  - Processos com tempos de execução críticos.
  - *Buffers* de *I/O* (em memória do utilizador)
    - » Objectivo: evitar que aconteça o seguinte
      - um processo pede *I/O*
      - a seguir, bloqueia
      - o processo que entra em execução gera uma falta de página
      - a página do *buffer* de *I/O* é substituída
      - o pedido de *I/O* é executado p/ uma pág. que não pertence ao processo que fez o pedido
    - » Soluções:
      - Nunca fazer *I/O* para a memória do utilizador mas p/ a memória do S.O. ou
      - Permitir que as páginas com *I/O* pendente sejam fixadas.

Fixação de páginas

- **Outra utilização da fixação de páginas:**
  - **Impedir que uma página recentemente carregada seja substituída antes de ser usada pelo menos uma vez.**

- **Exemplo:**
  - **Um processo de baixa prioridade tem uma falta de página.**
  - **A página é carregada.**
  - **Enquanto a página é carregada o procesador é atribuído a um processo de prioridade mais elevada.**
  - **Entretanto, a página pedida pelo processo de baixa prioridade é carregada.**
  - **O processo de alta prioridade também sofre uma falta de página.**
  - **A página substituída pode ser a que foi carregada a pedido do processo de baixa prioridade.**

Segmentação a pedido

- **Alguns processadores podem não suportar paginação mas suportar segmentação (ex.: Intel 80286).**

- **A segmentação a pedido é semelhante à paginação a pedido:**
  - **Um processo não precisa de ter todos os segmentos em memória para executar.**
  - **O descritor de cada segmento tem um *bit* de segmento válido / inválido.**
    (Os descritores contêm informação acerca do tamanho, protecção e localização dos segmentos)
  - **Quando um segmento referenciado não está em memória (→ *trap* p/ o S.O) é necessário carregá-lo.**
  - **Se houver necessidade de substituir um segmento p/ carregar outro usa-se um dos algoritmos de substituição descritos anteriormente.**
  - **Usa-se um *bit* de referência p/ saber os segmentos que foram acedidos.**

- **Maior diferença relativamente à paginação a pedido:**
  - **Necessidade de compactação p/ reduzir a fragmentação externa**
    - » **Se o espaço livre total for suficiente mas não houver nenhum bloco livre de tamanho suficiente ⇒ compactação**
    - » **Se o espaço livre total não for suficiente ⇒ substituição de segmentos e compactação (eventualmente)**