Teoria dos Sistemas Operativos

Índice

Introdução

Sistema Operativo: Controla a execução dos programas do utilizador e atua como intermediário entre o utilizador de um computador e o hardware. Fornece ao utilizador uma máquina virtual.

- Núcleo: Contém código para os serviços fundamentais. Está em memória principal.
- Device Drivers: Código que fornece uma interface simples e consistente com os dispositivos de I/O.
- Programa: Um ficheiro contendo código (programa executável).
- Processo: Um programa em execução. Coleção de estruturas de dados e recursos do SO detidos por um programa enquanto está a ser executado.
- Ficheiro: Coleção de informação relacionada entre si. O SO mapeia os ficheiros em dispositivos físicos.
- 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.
- Shell: Interpretador de comandos dados ao sistema operativo.

Chamadas ao sistema:

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

Melhoramento: Sobreposição das operações de entrada e de saída (I/O).

Spooling

- Forma de buffering: usar discos para guardar temporariamente as I/O's.
- Permite sobrepôr a fase de cálculo de um processo com a fase de I/O de outro.

Multiprogramação: Execução intercalada de processos.

- Várias tarefas são mantidas em memória simultaneamente, e a CPU é partilhada entre elas.
- Quando o programa atual fica à espera que uma operação de I/O (para o mesmo dispositivo) se complete, o processador pode executar outro programa.
- Necessita de:
    - Escalonamento da CPU: o sistema deve escolher entre os vários 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.
    - Proteção: não deve haver possibilidade de os processos se afetarem mutuamente.

Sistemas de tempo partilhado (time-sharing) - Computação interactive:

- A CPU é partilhada entre diversas tarefas que são mantidas em memória e em disco.
- É possível a comunicação on-line entre o utilizador e o sistema.

Interrupções: Uma interrupção é um mecanismo que permite que o processamento normal de um processador seja interrompido. Permitem que enquanto decorre uma operação de E/S de um processo o processador continue a executar outros processos.

Acesso Direto à Memória (DMA)

- Necessário um controlador de DMA ligado ao barramento do sistema.
- 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 diretamente de/para a memória.
- Quando o DMA termina é gerada uma interrupção

O SO deve impedir que um programa incorreto impeça os outros programas de executar. Alguns erros de programação são detetados pelo hardware e tratados pelo SO.

Proteção do hardware

- duplo modo de operação
    - modo utilizador
    - modo supervisor / sistema / monitor / privilegiado (instruções privilegiadas)

- proteção de E/S's: os utilizadores não conseguem fazer E/S diretamente, só através do SO
- proteção da memória: proteção da área de memória do SO e dos utilizadores feita por registos especiais
- proteção do processador: temporizador impede que uma aplicação tome conta do processador indefinidamente

Pontos de vista de um sistema operativo:

1. serviços que fornece.
2. interface que disponibiliza para utilizadores e programadores.
3. 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 proteção/segurança

Estrutura monolítica: Ou não há estruturação, onde 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). É difícil de compreender, difícil de modificar e pouco fiável.

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.
- Cada camada só usa funções e serviços das camadas inferiores.
- Há dificuldades na definição adequada das camadas e tende a ser menos eficiente.

Estrutura baseada em microkernel:

- 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 procedimentos de sistema que correm em modo de utilizador; estes processos comunicam entre si através de mensagens.

Hardware de um sistema de computação

4 Elementos principais do hardware:

- Processador
- Memória principal
- Dispositivos de I/O
- Ligações entre os outros elementos (barramentos)

Registos do processador

- PC -> Program Counter - Contém o endereço da próxima instrução
- IR -> Instruction register - Contém a instrução atual
- PSW -> Processor Status Word – Contém informação acerca do estado do processador, interrupção, flags (carry, zero, overflow…)
- SP -> Stack Pointer – Aponta para o topo da stack
- Registos do utilizador – Usado para vários fins, na programação
- Outros – Gestão de memória

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 o processador continue a executar enquanto decorre uma operação de I/O -> base da multiprogramação.

Classes de interrupções

- Programa – geradas por uma condição que resulta da execução de uma instrução
- Timer – permite que o SO execute certas tarefas regularmente
- I/O – geradas por um controlador de I/O para assinalar o fim de uma operação ou certos erros
- Falha de hardware – falha na alimentação, erro de paridade de memória…

Processamento de interrupções

A rotina a executar em resposta a uma interrupção é determinada com base num vetor de interrupções.

Cada plataforma de hardware tem um procedimento particular para cada interrupção.

Combinação de responsabilidade do SO e hardware no interrupt handler.

Processamento de uma interrupção

Hardware

- Ocorrência da interrupção
- O processador termina instrução atual
- O processador assinala aceitação da interrupção
- O processador guarda a PSW e o PC na stack
- O processador carrega PC c/o endereço da rotina de tratamento da interrupção

Software

- Guardar restante informação do estado do processo (registos do processador,…)
- Processar a interrupção (executar rotina)
- Restaurar informação do estado do processo
- Restaurar PSW e PC

Interrupções múltiplas

Processamento sequencial – inibir as interrupções durante o processamento de uma interrupção

Processamento embutido – prioridades de interrupção

I/O

Gestão eficiente de I/O -> responsabilidade do SO

3 mecanismos principais para executar I/O:

- Polling
- Interrupção
- Acesso direto a memória (DMA)

Polling

É o módulo de I/O que controla a ação, não o processador.

O módulo de I/O indica o seu estado num Status Register.

Não há interrupções.

O processador está sempre ocupado a verificar o estado do módulo de I/O.

Problemas quando o dispositivo de I/O é lento.

Interrupção

Quando o I/O se completar é gerada uma interrupção pelo dispositivo de I/O.

Mais eficiente que Polling mas o processador continua responsável pela transferência de dados entre a memória e o dispositivo de I/O.

Acesso direto à memória (DMA)

Necessário um controlador de DMA ligado ao barramento do sistema.

Em caso de I/O:

- O processador informa o controlador do dispositivo de I/O do que pretende fazer e onde está ou vai ficar a informação a transferir.
- O processador continua a executar outras instruções
- O dispositivo de I/O transfere a informação diretamente de/para a memória
- Quando o DMA termina é gerada uma interrupção

Proteção do hardware

Duplo modo de operação

2 modos de operação: utilizador e supervisor

O computador arranca em modo supervisor, o SO é carregado e dá início aos processos do utilizador em modo utilizador.

Quando ocorre uma exceção (interrupção) o hardware comuta para modo de supervisor.

O sistema comuta sempre pra modo utilizador antes de ceder o controlo a um programa do utilizador.

Nota : chamadas ao sistema são o método usado para um processo pedir uma ação ao sistema operativo, implicação a comutação do modo utilizador para modo supervisor; usam em geral interrupções por software e os parâmetros são passados nos registos do processador.

Proteção de I/O

Impedir um utilizador de executar I/O “ilegal”

Definir todas as opções de I/O como privilegiadas, as quais os utilizadores não conseguem fazer diretamente, apenas através do SO.

Proteção da memória

Fundamental proteger: vetor de interrupções e rotinas de serviço de interrupção

Proteção da área de memória de cada utilizador:

- feita por hardware
- 2 registos que só podem ser manipulados pelo SO determinam a gama de endereços válidos a que um programa pode aceder.

Proteção do processador

Impedir que um programa do utilizador tome conta do processador indefinidamente e não retorne ao controlo ao SO.

Usar um timer que após um período especificado interrompe o programa em execução.

As instruções de manipulação do timer são privilegiadas.

O timer também é usado para:

- Implementar time-sharing
- Manter atualizada a hora do sistema

Requisitos de hardware para multiprogramação

Um SO com multiprogramação necessita de suporte de hardware:

- Temporizador
- Hardware de DMA
- Mecanismo de interrupções com prioridades
- Duplo momo de operação do processador
- Mecanismo de proteção da memória
- Mecanismo de atribuição dinâmica de endereços

Arranque de um PC (booting)

- Após o POST (Power On Self Test) que verifica o estado do hardware é feito o reset do processador.
- O processador procura uma instrução no endereço 0xFFFFFFF0
- A instrução neste endereço é um salto para o início do BIOS, em ROM
- O BIOS determina, na sua configuração, qual o boot device (disquete/disco/...)
- O BIOS lê o MBR-Master Boot Record (primeiro sector) do boot device; o MBR contém informação acerca das partições existentes no disco e o endereço do boot sector
- O BIOS carrega, em RAM, um pequeno programa, contido neste sector que poderá, por sua vez executar outros programas, cuja execução culminará com o carregamento do sistema operativo
- O sistema operativo executa vários procedimentos de inicialização
- Alguns processos começam a ser executados

Estrutura de um sistema operativo

- Componentes do sistema operativo
- Serviços do sistema operativo
- Chamadas ao sistema

- Programas de sistema
- Estrutura do sistema operativo

Componentes do S.O.

- **Pontos de vista:** serviços que fornece, interface que disponibiliza com componentes e interligações.
- **Componentes:** Gestão de processos, memória principal e secundária, entradas/saídas, ficheiros rede e protecção.

Serviços do S.O.

- **Tipos (facilitam o programador):** criação de programas (editores e debuggers), execução de programas, acesso a dispositivos de I/O, acesso a ficheiros, comunicações, detecção de erros/falhas, contabilidade de utilização,...
- **Serviços que garantem a eficiência do sistema:** Alocação de serviços, contabilidade do sistema, proteção (impedir que um processo interfira com outros) e segurança (acessos não autorizados).

Programas de Sistema

- Fornecem um ambiente conveniente .
- Alguns programas são interfaces simples usadas para chamadas ao sistema.
- O interpretador de comandos é o programa de sistema mais importante.
- **Programas:** edição de texto, manipulação de ficheiros e diretórios, informação de estado, suporte a linguagens de programação,
  carregamento e execução de programas, comunicações,...

Estrutura de um S.O.

- Estrutura **monolítica**;
- Estrutura **em camadas**;
- Estrutura **microkernel**.

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.
- **Dificuldades:** difícil de compreender, difícil de modificar, pouco fiável, difícil de manter,...

Em Camadas

- O S.O. é dividido num certo número de camadas (níveis), camada 0 - Hardware, camada de mais alto nível - interface com o utilizador.
- É um sistema modular.
- Cada camada só usa funções e serviços da camada inferior.
- Uma camada não necessita de saber como as operações da camada inferior são implementadas, mas apenas o que elas fazem.
  Dificuldades: Definição adequada das camadas, tende a ser menos eficiente do que outros tipos.

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 de memória
        - gestão básica da CPU
        - comunicação entre processos
        - suporte de I/O
    - A restante funcionalidade do S.O. é implementada em processos de sistema que correm em modo de utilizador. Estes processos comunicam entre si através de mensagens (modelo cliente-servidor) .

Estruturas de alguns S.O.'s LINUX

- Estrutura essencialmente monolítica.
- É relativamente modular, internamente, de forma a não ser muito difícil fazer modificações.
- O kernel tem uma camada portável (independente) e outra não portável (não dependente) da arquitectura do hardware.

Arquitectura do UNIX - Características

- Multitasking, multiutilizador . Kernel monolítico.

Características do Windows 2000/XP

- Sistema multitasking com um único utilizador
- Suporta multiprocessamento simétrico
- Suporta multithreading
- Suporta interfaces de outros sistemas operativos (subsistemas)
- Estrutura microkernel modificada - os serviços correm em modo privilegiado.
- Camadas: **HAL - Hardware Abstraction Layer, Microkernel, Serviços executivos, Subsistemas.**

Processos e threads

Noção de Processo

Processo != Programa

- Programa – entidade passiva
- Processo – entidade ativa

Programa - Loading - Processo

Loading – Memória do Processo inicializada com código do programa e dados

- S.O. cria Process Block Control (estrutura de dados com essa informação)

Multiprogramação

Problema: execução sequencial de processos pouco eficiente (instruções I/O mais lentas que instruções CPU)

Solução – execução concorrente de processos (multiprogramação)

i.e. processos que estejam à espera de instruções I/O são bloqueados permitindo a execução de instruções CPU de outro processo

Multiprogramação **com preempção** – O S.O. decide quando ceder a CPU

Multiprogramação **sem preempção** – Os processos decidem quando ceder a CPU

Multiprogramação é o contrário de Uniprogramação (apenas um processo em execução)

Dificuldades de implementação:

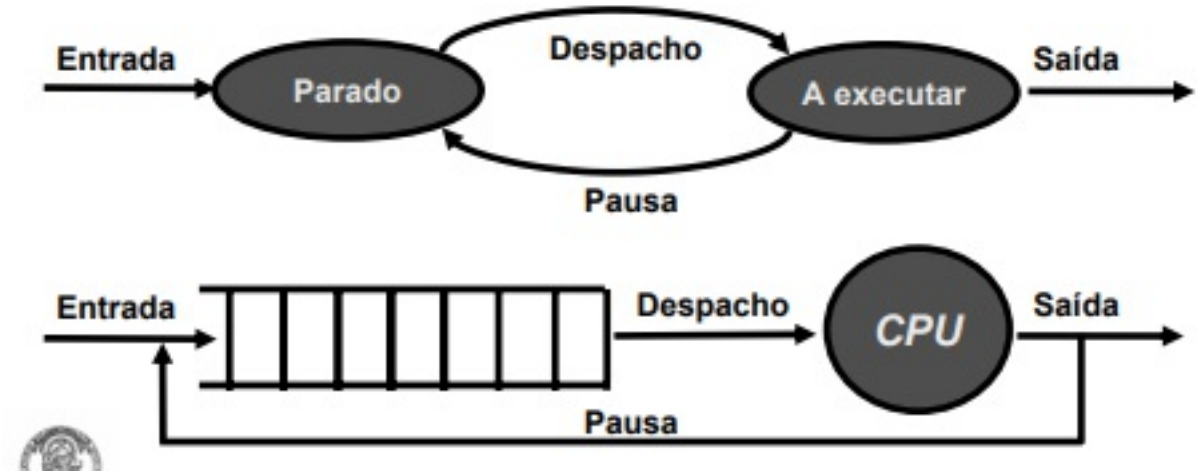
- Necessidade de proteção de recursos e controlo de acesso a memória, periféricos, …
- Hardware com características especiais (dois modos de funcionamento, registos especiais, …)
- Comunicação entre processos independentes

Processos e Threads

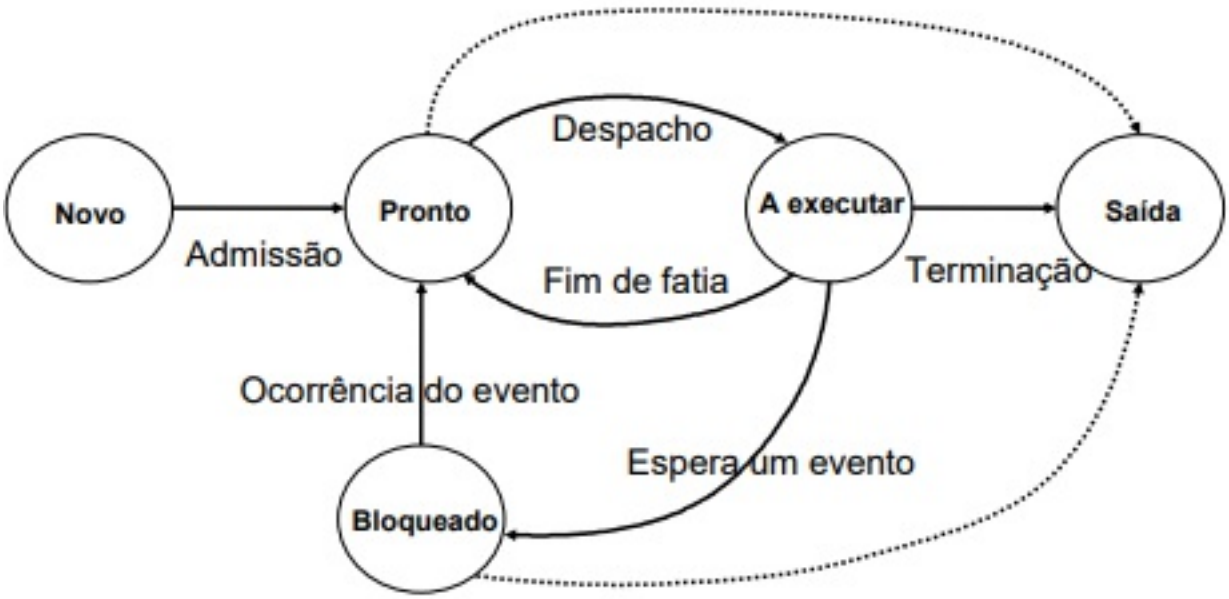
- Modelo *lightweight process*:
- Processo – posse de recursos (ficheiros, memória)
- Thread(s) – sequencia de execução(program counter, …)

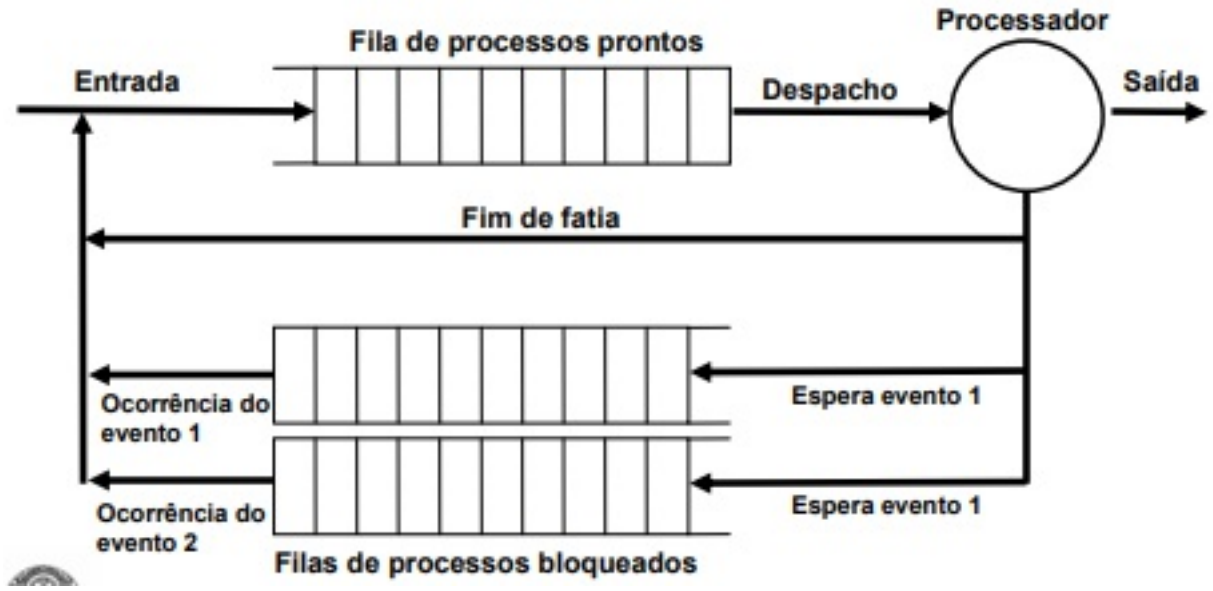
Estados de um Processo

Modelo 2 estados



Necessário guardar lista de processos bloqueados pela Multiprogramação -> Modelo 5 estados



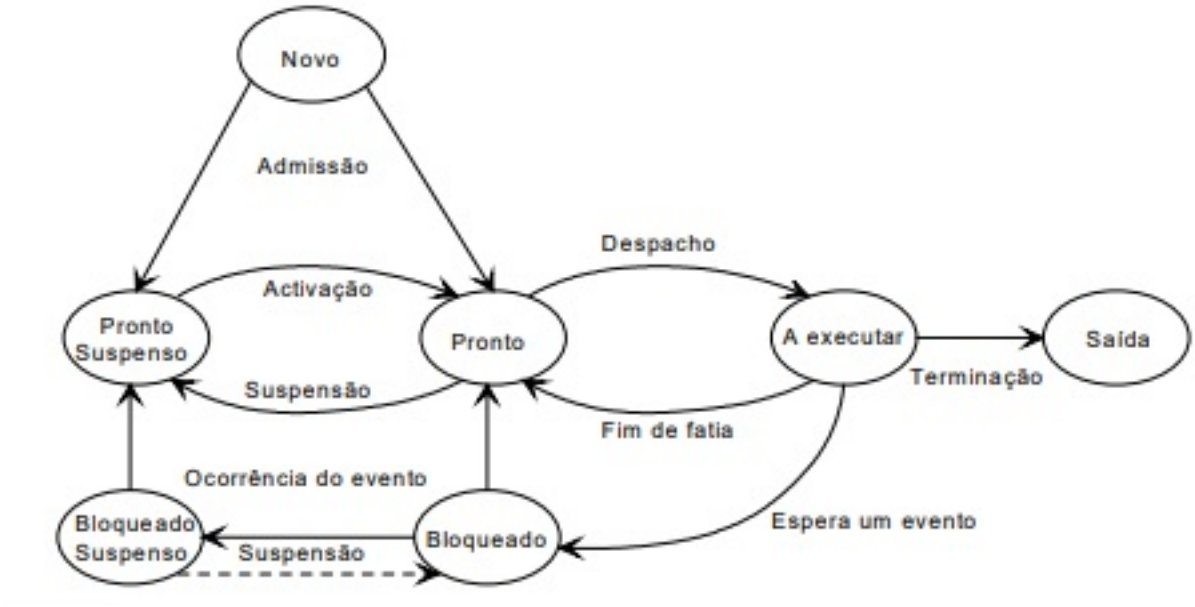


Swapping

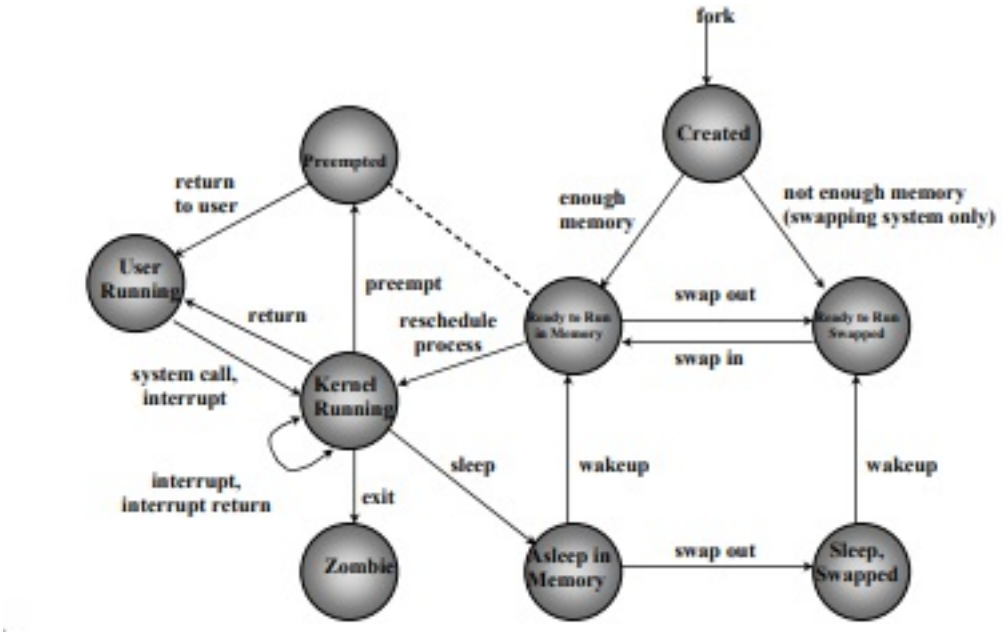
Problema: como o processador é muito mais rápido que a I/O será comum acontecer que todos os processos em memória estejam à espera de I/O.

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

Processos em swapping são considerados suspensos -> Modelo de 6/7 estados



Modelo do UNIX



Descrição de Processos

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 (composta por *Process Control Blocks*)

Informação em PCB:

- ID
- Estado
- Registos
- Escalonamento CPU (prioridade)
- Gestão Memória

- Contabilidade
- Estado I/O

Outras estruturas de controlo de processos:

- Tabelas de Memória
- Tabelas I/O
- Tabelas de Ficheiros

Gestão de Processos

Operações típicas do núcleo (kernel):

- criação e terminação de processos
- escalonamento e despacho

schedulller - selecciona o próximo processo a executar

dispatcher - dá o controlo da CPU ao processo seleccionado

- sincronização e suporte para intercomunicação entre processos
- gestão dos PCB's

Criação de um processo

- Criação de estruturas (PCB, …)
- Alocação de memória

Terminação de um processo

- É-lhe retirado o processador
- Informação relativa ao processo apagada (mantida temporariamente para que possa ser analisada por outros processos – Estado Zombie)

Threads

Thread – *Lightweight Process*

Sistemas baseados em threads:

- Um processo pode ter vários threads
- Threads passam a ser a unidade de escalonamento
- Threads são independentes em termos de prioridade de utilização de CPU
- Processo transforma-se em ambiente de execução para threads

Semelhanças com Processos Tradicionais (*Heavyweight Processes*)

- Estado (pronto, a executar, bloqueado, …)

- Partilha de CPU
- Execução sequencial dentro de si mesmo
- PC, Stack pointer, Thread Control Block (== PCB)
- Possibilidade de criar threads filho

Características Importantes

- Não existe proteção entre threads do mesmo processo
- Variáveis globais são inteiramente partilhadas por todos os threads
- Suspensão (Swapping) de um processo => suspensão de todos os seus threads
- Terminação de um processo => terminação de todos os seus threads

Vantagens de utilização

- Economia e velocidade
- Aumento da rapidez de resposta percebida pelo utilizador (Multiprogramação dentro de um só processo)
- Eficiência de comunicação (memória partilhada)
- Utilização da arquitetura multiprocessador (threads têm execução independente)

Dificuldade – Sincronização (Utilização comum do mesmo conjunto de variáveis)

User-Level Threads – Kernel desconhece os threads – gestão realizada pela aplicação utilizando bibliotecas apropriadas

Kernel-Level Threads – Gestão feita pelo kernel através de uma API de threads

Escalonamento do processador

Estratégia de atribuição de CPU a processos => base da multiprogramação

Execução de processos divide-se em duas partes:

- CPU Bursts – conjunto de instruções consecutivas executadas pelo CPU
- I/O Bursts – espera por uma operação I/O

Processo

- CPU-bound:
    - passa a maior parte do tempo a usar CPU
    - pode ter CPU bursts muito longos
- I/O-bound:
    - passa mais tempo a fazer I/O
    - tem muitos CPU bursts curtos

Algoritmo de escalonamento depende da distribuição de bursts

Tipos de Escalonamento

- Escalonamento a longo prazo (Big Picture) – determina o grau de multiprogramação, i.e., o número de processos simultâneos em memória
- Escalonamento a médio prazo – Resolução de escassez de recursos (Executado com intervalos de segundos/minutos)
- Escalonamento a curto prazo (*de facto*) – resolução de alteração de estado dos processos. Executado com intervalos de centenas de milissegundos. Pode ser:
    - **preemptivo**: o processo pode ser forçado a ceder CPU
    - **não preemptivo**: o processo executa até bloquear ou ceder a vez voluntariamente

Critérios de escolha de algoritmos de escalonamento

- Percentagem de utilização do processador
- *Throughput* (nº de processos completados / unidade tempo)
- Tempo de resposta
- Tempo de permanência de um processo no sistema
- Tempo total de espera (inatividade) de um processo

Estratégias

- First-Come-First-Served (FIFO)

Processos escalonados por ordem de chegada. Não-preemptivo.

Não existe tempo perdido em decisão. Não existe possibilidade de *starvation*, i.e., existe garantia de execução de todos os processos.

É pouco eficiente e causa elevados tempos totais de espera

- Shortest-Job-First

Selecionado o processo com o menor próximo CPU Burst

Preemptivo ou não preemptivo

Tempos médios de espera mínimos

Difícil implementação (dificuldade no cálculo de duração de CPU Bursts)

Possibilidade de *Starvation* (processos com elevados CPU Bursts podem nunca ser atendidos)

Alto overhead de cálculos

- Priority Scheduling

A cada processo é atribuído um inteiro representando a sua prioridade.

Processador atribuído ao processo de maior prioridade

Preemptivo ou não preemptivo

Alta probabilidade de *Starvation* (se as prioridades forem estáticas, processos de muito fraca prioridade podem nunca ser atendidos)

Solução: Envelhecimento de processos (a prioridade de cada processo aumenta com o passar do tempo)

- Round Robin

A cada processo é atribuído uma fatia de tempo (*quantum*), independente do tempo do seu CPU Burst

Após a fatia de tempo, o processo é colocado no fim da fila de prioridade (do tipo FIFO)

Não existe *Starvation*

Problemático com processos com elevados I/O Bursts (muito tempo de processador desperdiçado)

- Multilevel Queue

Baseado num sistema de prioridades, mas com várias filas escalonadas entre si

Cada fila tem o seu algoritmo de escalonamento

Escalonamento das filas:

- Prioridade Fixa (umas filas têm maior prioridade que outras – possibilidade de *Starvation*)
- Fatias de tempo (semelhante ao Round Robin)

Escalonamento em sistemas multiprocessador

- Escalonamento Mestre/Escravo

Um “processador-mestre” é responsável pelo S.O. e faz o despacho de tarefas
Os restantes “processadores-escravos” executam apenas programas do utilizador

- Auto-escalonamento

Todos os processadores acedem à lista de processos prontos a executar
Dificuldade de assegurar o manuseamento seguro da lista de processos entre os diferentes processadores

Escalonamento em sistemas de Tempo-Real

- Hard-Real-Time

Tarefas têm de ser completadas dentro de um intervalo de tempo garantido

Hardware muito específico

Necessidade de saber com exatidão o tempo de execução de todas as funções do S.O. (impossível em sistemas com memória virtual)

- Soft-Real-Time

Sistema de prioridades
Definição de processos críticos
Latência de despacho baixa
Possibilidade de preempção

Sincronização de processos

Execução concorrente – Execução logicamente ao mesmo tempo (multiprogramação)

- Pode ocorrer em sistemas uniprocessador ou sistemas multiprocessador (multiprogramação c/ multiprocessamento)

Execução em paralelo - Execução fisicamente ao mesmo tempo (multiprocessamento)

Processos a correr ao mesmo tempo precisam de partilhar dados/recursos, por isso é necessário protegê-los de erros / inconsistências.

Erros comuns envolvem dar update a variáveis partilhadas por vários processos, **race conditions** (vários processos manipulam dados ao mesmo tempo e o resultado da execução depende da ordem de acesso) e condições sobre variáveis que são alteradas por outros processos.

Secção crítica: Processo executa código que manipula dados/recursos partilhados. A execução destas deve ser mutualmente exclusiva, por isso cada processo deve pedir autorização para entrar na sua secção critica. A secção de código que implementa esse pedido é a **secção de entrada**, seguida pela **secção critica**, **secção de saída** e **secção restante** (resto do código). O problema das SC é conceber um protocolo para que o resultado não dependa da forma como os processos são interlaçados.

Requisitos para essa solução:

- **Exclusão mútua**: Só um processo de cada vez pode entrar na sua SC.
- **Progresso**: Um processo a executar uma secção não crítica não pode impedir outros de entrar na SC.
- **Espera limitada**: Um processo que pede para entrar na sua SC não pode ficar à espera indefinidamente.

Tipos de soluções: Baseadas em software (user), hardware ou Sistema Operativo.

Soluções baseadas em software

Algoritmo de Peterson (para 2 processos)

Variáveis partilhadas:

```
turn: 0/1;
flag: bool[2];
/* flag[i] = true significa que
Pi está pronto a entrar na sua SC*/
```

Inicialmente:

```
turn = qualquer valor, 0/1;
flag[0] = flag[1] = false;
```

Processo Pi:

```
flag[i] = True;

//Pi quer entrar, mas dá a vez a Pj
turn = j;

//Espera que Pj saia ou lhe de a vez
While (flag[j] && turn == j) {}

//Faz a sua secção crítica
{Secção Critica}
flag[i] = false;
```

Análise do algoritmo:

- Garante a exclusão mútua das secções críticas, o progresso e uma espera limitada
- A sua implementação para mais de 2 processos é complicada

Algoritmo da padaria (para n processos)

Antes de entrar na SC cada processo tira um ticket e entra quem tiver ticket mais pequeno (desempate por PID caso vários processos recebam o mesmo número)

Variavéis partilhadas:

```
Choosing: bool[n];
number: int[n];
```

Inicialmente:
Processo P1:

```
choosing[i] = true;
number[i] = max(numbers) + 1;
choosing[i] = false;

for(j = 0; j < n; j++)
{
    //Espera que todos tirem o ticket
    While(choosing[j]) {}

    While(number[j] != 0 && number[j] <
     number[i]) {}

    /*
    Espera por todos os que tenham tickets
    mais baixos (se igual desempata o PID)
    acabem a sua SC
    */
}

{Secção Critica}

number[i] = 0;
```

Limitações das soluções por software:

- São da competência do programador
- São complexas
- Requerem ***busy waiting*** (os processos que estão a pedir para entrar na sua SC estão a consumir CPU)

Soluções baseadas em hardware

Sistemas uniprocessador:

Para garantir exclusão mútua basta inibir interrupções

Processo Pi:

```
Disable_Interrupts();
{Secção Crítica}
Enable_Interrupts();
```

Sistemas multiprocessador:

Necessárias instruções especiais que permitam testar e modificar uma posição de memória num único passo.

Instrução **Test_and_Set**: (atómica)

```
While Test_and_Set(Lock) {}
{SecçãoCrítica}
Lock = false;
```

Não satisfaz a condição de espera limitada. A seleção do próximo processo a executar a secção crítica é arbitrária (um processo pode ficar indefinidamente à espera).

Instrução **Swap**: Troca o valor de 2 variáveis atomicamente

```
//Variáveis partilhadas: Lock: Boolean (valor inicial: false)

Key = true;
do {Swap(Lock, Key);} while(Key != false);
{Secção Crítica}
Lock = false;
```

Spinlock:

Mecanismo de sincronização em que um processo espera num ciclo testando se o lock está disponível (busy waiting, garante exclusão mútua).

Operações:

- `InitLock {lock = false}`
- `Lock {While (TestAndSet(lock)) {}}`
- `UnLock {Lock = false}`

Secção Critica:

```
Lock(Mutex);

{Secção Crítica}

UnLock(Mutex);
```

Características de soluções por hardware:

- Podem ser usadas em várias secções críticas (cada uma delas controlada por uma variável)
- São aplicáveis a qualquer número de processos e processadores
- Requerem instruções máquina especiais
- Usam *busy waiting*
- Se não forem tomadas precauções, levam à inação

Semáforos

Mecanismo de sincronização (do SO) que não requer espera ativa.

Semáforo S – Integer iniciado com um valor >= 0, que representa o número de threads/processos que podem executar a secção crítica.

Depois de inicializado só pode ser atualizado através de duas operações atómicas:

`wait(S): S -= 1; if (S < 0) Block(S);`

`signal(S) : S += 1; if (S <= 0) WakeUp(S);`

`Block(S)` – o processo que chamou é bloqueado

`WakeUp(S)` – um processo que está parado em Block(S) desbloqueia

`Wait()` e `Signal()` têm de ser atómicas.

Utilização:

Valor inicial: `Mutex = 1;`

```
Wait(Mutex)

{Secção Critica}

Signal(Mutex)
```

Neste caso, o semáforo é inicializado com 1. Portanto, o primeiro processo executa `wait()`, e como s=0, avança para a secção crítica. Se o outro processo também executar `wait()`, então fica bloqueado porque s<0, e apenas avança quando o outro processo fizer `signal()`.

Problemas comuns: Inicializar o semáforo a 0 em vez de 1 e *deadlocks* quando processos têm 2 semáforos e cada um deles está parado num diferente.

Inicializar o semáforo a 0 pode ser usado para sincronizar a execução de 2 processos.

Problemas clássicos de sincronização:

- Problema do Produtor/Consumidor
- Problema dos filósofos a jantar
- Problema dos Leitores/Escritores

Construções de alto nível para exclusão mútua e sincronização

(monitores, regiões críticas e passagem de mensagens)

Monitores

Módulo de software constituído por

- 1 ou mais procedimentos
- 1 secção de inicialização
- Dados locais

Só são visíveis os procedimentos, os dados locais são inacessíveis, a entrada no monitor apenas é possível por uma chamada e só um processo pode estar a executar no monitor de cada vez.

Assim, é fácil implementar a exclusão mútua. A sincronização é obtida através de ***condition variables***.

Os monitores podem ser implementados recorrendo a semáforos e vice-versa. Tal como nos semáforos é possível cometer erros de sincronização com os monitores.

Regiões críticas

Uma região crítica protege uma estrutura de dados partilhada. O compilador encarrega-se de garantir a exclusão mútua no acesso aos dados.

Requer um variável V Shared, que só pode ser acedida dentro de uma instrução do tipo:

Region V When B do S;

Onde B é um booleano e S uma instrução. Enquanto S estiver a ser executada, nenhum outro processo pode executar uma região associada a V. Quando um processo tentar executar uma região B é avaliado:

Caso seja true, S é executado

Caso seja false, o processo espera até que B == true e nenhum outro processo esteja a executar uma região associada a V.

Passagem de mensagens

Semáforos e monitores resolvem o problema de exclusão múltipla em sistemas com 1 ou mais CPUs com memória comum, mas não podem ser usados em sistemas distribuídos.

Por outro lado, passagem de mensagens pode ser usada em sistemas com memória partilhada bem como em sistemas distribuídos.

Os Sistemas Operativos implementam geralmente um sistema de mensagens que permite que os processos comuniquem e sincronizem as suas ações. 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. O sender normalmente não bloqueia após executar send() e o receiver normalmente bloqueia após executar receive().

Deadlocks

Um **deadlock** é um bloqueio permanente de um conjunto de processos que competem por recursos do sistema ou comunicam entre si.

Condições necessárias para a ocorrência de um deadlock:

- **Exclusão Mútua:** Só um processo pode usar um recurso de cada vez.
- **Retém e Espera:** Um processo pode deter recursos enquanto espera pela atribuição de outros recursos.
- **Não preempção dos recursos :** Quando um processo detém um recurso, só ele o pode libertar.
- **Espera Circular:** Deve existir um conjunto de processos {P1, P2, ..., Pn} tal que:
    - P1 está a espera de um recurso que P2 detém.
    - P2 está a espera de um recurso que P3 detém.
    - ...
    - Pn está a espera de um recurso que P1 detém.

Deadlock vs 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.

Métodos de Tratamento dos Deadlocks (Prevenir, Evitar, Detetar e Recuperar)

||Prevenir os Deadlocks||

Assegurar que pelo menos 1 das 4 condições necessárias não se verifica.

- **Exclusão Mútua:**
    - **Solução:** Usar apenas recursos partilháveis.
    - **Problema:** Certos recursos têm de ser usados com exclusão mútua.
- **Retém e Espera:**
    - **Soluções(Garantir que quando um processo requisita um recurso, não detém nenhum outro):**
        - Requisitar todos os recursos antes de começar a executar.

        - 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.
        - Possibilidade de Inanição.

- **Não preempção dos recursos:**

    - **Solução:** Quando 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.

- **Espera Circular:**

    - **Solução:** Os vários tipos de recursos são ordenados e os processos devem requisitá-los por essa ordem.
    - **Problemas:**

        - Ineficiência devido à ordenação imposta aos recursos.
        - Difícil encontrar uma ordenação que funcione.

||Evitar os Deadlocks||

Não conceder recursos a um processo se houver possibilidade de occorrer um deadlock. Os principais algoritmos para evitar deadlocks baseiam-se no conceito de estados seguro e inseguro:

Estado Seguro: Se o sistema conseguir alocar recursos a cada processo, por uma certa ordem, de modo a evitar deadlocks.
Estado Inseguro Estado que pode conduzir a deadlock.

Primeira 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.

Segunda Estratégia: Não conceder um recurso adicional se essa concessão for susceptível de conduzir a um deadlock.

- **Algoritmo do banqueiro**

```
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;
```

Vantagens em Evitar Deadlocks

- Menos restritivo do que a prevenção.
- Não requer a requisição simultânea de todos os recursos necessários.
- Não obriga à preempção dos recursos.

Dificuldades em Evitar Deadlocks

- Necessidade de conhecimento antecipado de todos os recursos necessários
- Utilidade prática limitada.
- Overhead necessário para detectar os estados seguros.

||Deteção 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.

Deteção

Quando fazer a detecção ?

- Sempre que é concedido um novo recurso.
- Overhead elevado.
- Com um período fixo.
- Quando a utilização do processador é baixa.

Método 1: Quando há uma única instância de cada tipo de recurso.

- Manter um grafo de processos e recursos . Periodicamente, invocar um algoritmo de detecção de ciclos em grafos.

Método 2: Quando há várias instâncias de cada tipo de recurso.

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

Como proceder à recuperação ?

- Avisar o operador e deixar que seja ele a tratar do assunto.
- O sistema recupera automaticamente:
    - **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.
    - **Preempção de Recursos:**
        - 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?
            - Tomar nota do nº de rollbacks no factor de custo.

Estratégia integrada

Combinar os 3 métodos, partindo os recursos em classes, e seleccionar o método mais adequado para cada classe.

Exemplos:

- **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

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:

- **Overlay** : (técnica de sobreposição) permite carregar partes diferentes de um programa na mesma memória (alturas diferentes) corre processos que ocupem mais memória do que a memória física disponível
- **Recolocação**: capacidade de carregar e executar um dado programa num lugar arbitrário de memória
- **MMU**: memory management unit
- **Linker** : junta um conjunto de módulos-objeto produzindo um módulo contendo programa global e os dados a serem passados ao loader
- **Loader :** coloca módulos carregáveis em memória
- **Swaping**: possibilidade de trocar blocos de memória entre memória principal e disco
- **PTBR**: page-table base register – endereço para a base de uma tabela de páginas

- **TLB**: Translation Look-aside Buffer – cache de acesso rápido onde é mantida a informação acerca das páginas acedidas recentemente

Técnicas de gestão de memória:

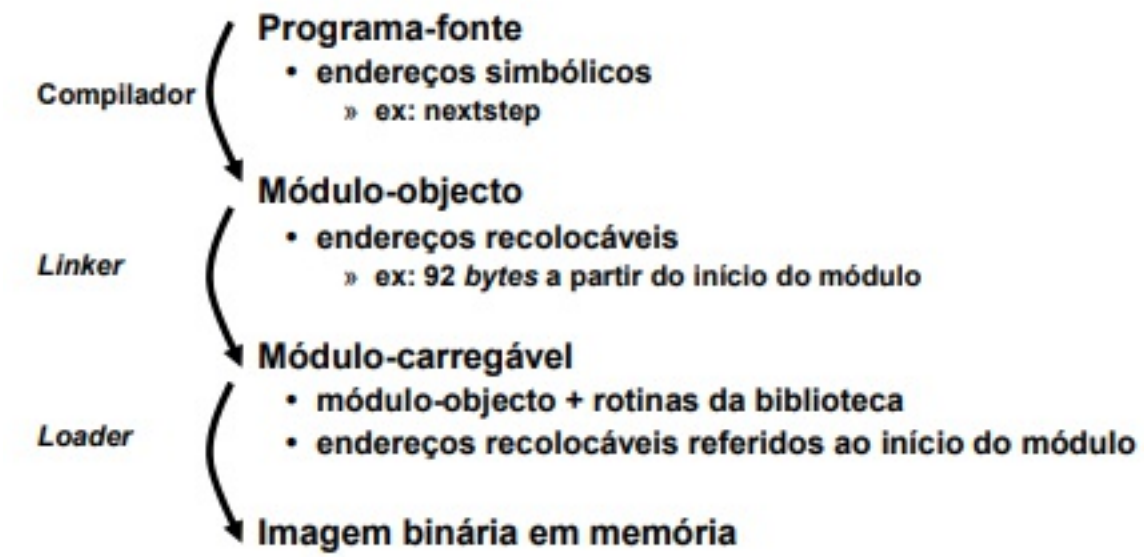
- Alocação contígua e não-contigua
- Partição fixa e dinâmica
- Paginação e segmentação
- Memória virtual

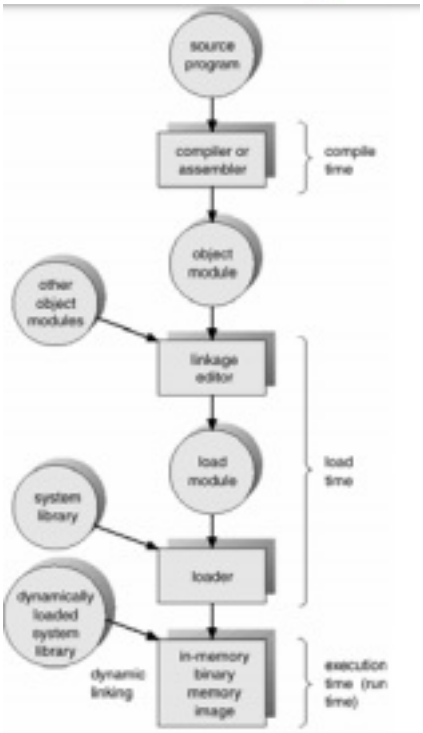
Sistemas

Monoprogramação: 1 processo em memória de cada vez; pode ocupar memória toda alocada ao user, necessário usar **overlay** para processos que ocupam mais memória do que a física

Multiprogramação: tem que haver **recolocação**, proteção (diferenciar espaços de endereço do SO e das aplicações) e partilha (cooperação processos)

Criação de programa executável





Recolocação

Estática (antes ou depois do carregamento) [Nota: não pode ser deslocado em memoria-swapping dificil]

-Antes: na compilação ou na ligação (linking) [há que conhecer onde o processo vai ser carregado]

-Durante: feito pelo loader – o compilador gera código recolocável

Dinâmica (necessita suporte de hardware para o base register – **MMU**)

Ocorre durante a execução do programa, pode ser deslocado em memória durante a execução.

Linking

Ligação estática: cada módulo é criado com referências relativas ao inicio do módulo e todos os módulos são colocados num único módulo recolocável com referências relativas ao inicio do módulo global

Ligação dinâmica:

- durante carregamento: módulo é lido para a memória e avisa o loader de módulos externos

Vantagens: fácil incorporar novas versões de módulos; basta uma cópia de cada módulo

- durante execução: SO trata da ligação a módulos ausentes

Vantagens: permite alterar rotinas da biblioteca sem recompilar programas; permite que uma única copia da rotina seja partilhada por diferentes processos

Loading

Absoluto: programa sempre carregado no mesmo endereço inicial; referências a memória absolutas; atribuição de endereços é feita pelo programador, pelo assembler ou pelo compilador

Recolocável: Local onde o programa é carregado é variável; assembler produz endereços relativos ao início

Dinâmico em run time: permite **swapping**; Geração de endereços absolutos só é feita quando a intrução é efetivamente executada (com suporte de hardware)

Técnicas de overlay (já caiu em desuso)

Tática: dividir programa numa parte residente (sempre em memória) e em overlays que são módulos independentes; a parte residente gere os módulos overlay

Endereçamento real e virtual

Real: endereço é igual ao acedido na memória do computador (endereço físico)

Desvantangens: dimensão do programa limitada à dimensão da memória fisica

Programa só pode funcionar nos endereços físicos para que foi escrito

Lógico ou virtual: endereços gerados pelo programa são convertidos pela **MMU**

Swapping e partilha de memória

Seleciona processos que vão sofrer swap-out (processos bloqueados ou baixa prioridade) e substitui-os por processos que vão sofrer swap-in.

Swap-file: ficheiro que guarda as imagens dos processos que sofreram swap-out, apenas 1 swap-file mas cada processo tem uma imagem associada

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

Partição Fixa

Memória destinada a processos do utilizador está dividida em blocos de tamanho fixo (podem ser diferentes entre si).

Fragmentação interna: diferença entre o tamanho do processo e o tamanho do bloco a que foi alocado no caso em que o processo seja menor do que o bloco.

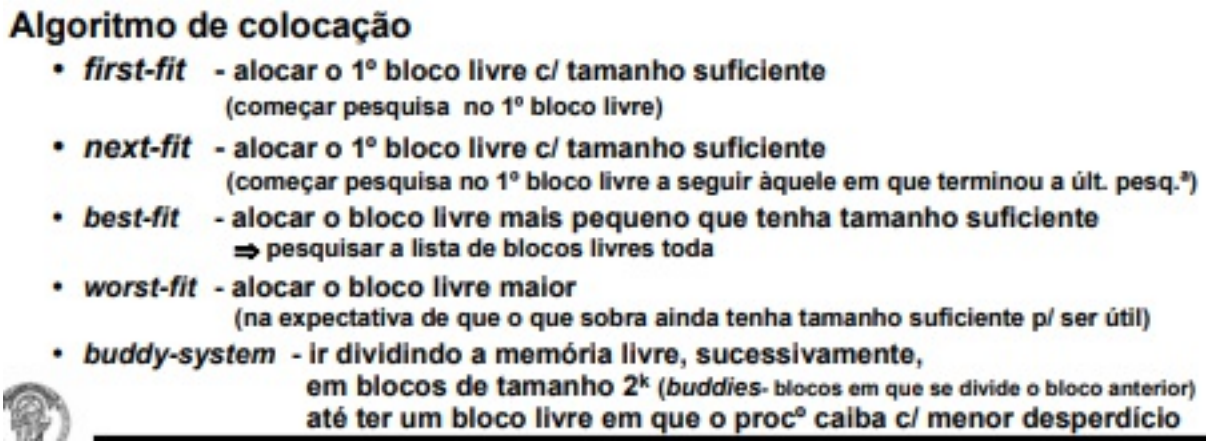
Mecanismo de protecção: par de registos onde são carregados os endereços máx. e min. da partição actual

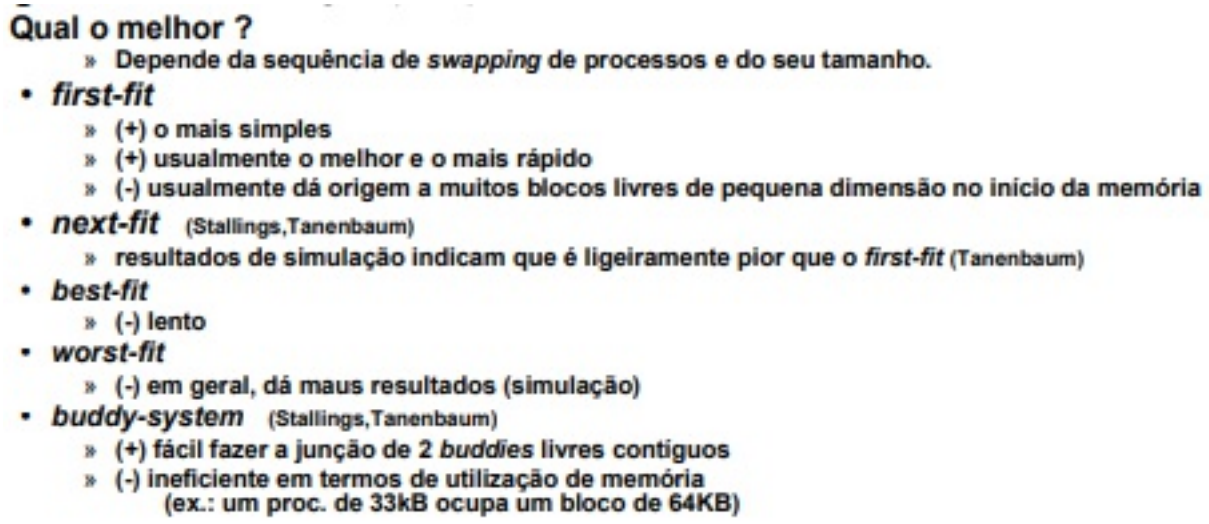
Desvantagens: o nº de partições limita o nº de processos activos; utilização ineficiente da memória, quando os processos são pequenos

Partição Dinâmica

Inicialmente 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.

Fragmentação externa : Ao fim de algum tempo existirão fragmentos de memória não utilizada espalhados pela memória do computador





Problemas:

- fragmentação externa
- perda de tempo na gestão de buracos livres muito pequenos (=sem utilidade)

- necessidade de compactação

Mapas de bits:

- dividir a memória em blocos, cada bloco é associado um bit que indica se está ocupado ou não
- overhead para verificação de blocos livres consecutivos para carregar um programa

Listas Ligadas:

- manter uma lista duplamente ligada com o mesmo propósito, mais fácil junção de blocos contíguos

Paginação (uma forma de recolocação dinâmica)

Objetivos: faciliar a alocação, facilitar o swapping, reduzir a fragmentação da memória

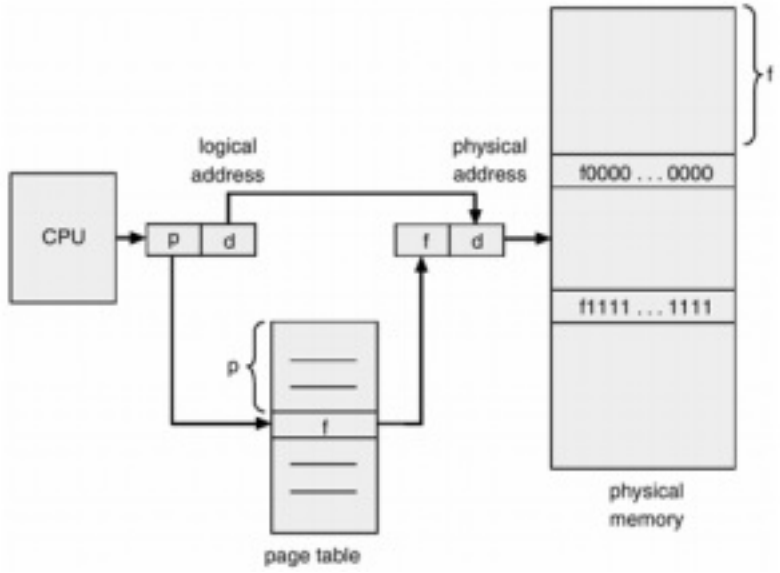
Procedimento:

- dividir a memória física em blocos chamados quadros (frames)
- dividir a memória lógica em blocos de tamanho igual aos quadros chamados páginas
- páginas de um processo são carregados em qualquer quadro que esteja livre
- S.O. mantém uma tabela de páginas, por cada processo, onde faz a correspondência página/frames

Hardware de suporte:
nPágina = EndereçoLógico DIV TamanhoPágina;
endBase = TabelaPáginas [nPágina];
offset = EndereçoLógico MOD TamanhoPágina;
EndereçoFísico = endBase + offset
ou
EndereçoFísico = frame*TamanhoPágina + offset
NOTA: a nível prático as páginas têm tamanho com potência de 2 para facilitar as contas

Extração direta
Um endereço lógico tem duas partes: O número da página e o offset.
Os bits do número da página são os mais significativos, e o contrário para o offset. Para determinar onde se faz a divisão, pegar no tamanho da página, e transformar em potência de 2: o expoente representa o número de bits do offset.

Os endereços físicos são constituídos por número de frame (parte mais significativa) e pelo offset.



Dificuldades: overhead por cada referência à memória, normalmente as tabelas de página são mantidas em memória principal e a MMU apenas tem o endereço inicial da tabela (porque as tabelas não cabem na MMU); necessário ir atualizando as páginas do processo que vai correr e alguns registos do hardware

Espaço ocupado por uma tabela:

Espaço de endereçamento de 32 bits => 4Gb = 2^32

Cada pagina com 4KB (2^12) e 4 bytes por elemento

Origina uma tabela com 4MB (2^32/2^12 *4)

Estruturação de uma tabela de páginas

- Nº do quadro (offset)
- Presente/ausente (1bit) indica se a entrada é válida
- Modificada (1bit) indica se foi modificada – importante para saber se tem de ser escrita em disco
- Referenciada (1bit) importante para gestão da memória virtual
- Proteção (1 ou 3 bits) se for 1 bit: 0-> read/write, 1-> read only

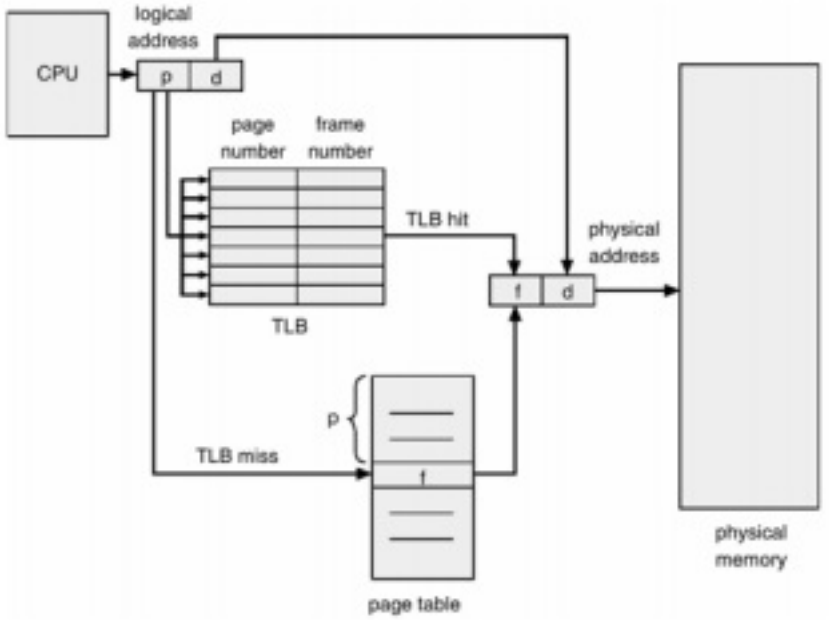
Se forem 3 bits: Read/write/execute (enable, disable)

-Inibição de caching (1bit) para mapeamentos em memórias de dispositivos I/O

TLB

- Cada registo tem 2 campos : uma chave e um valor
- Procura na TLB quase como se fosse um map
- Acesso à TLB paralela à tabela de páginas
- Caso haja hit na TLB o processo paralelo cancela

- Hit ratio de aproximadamente 100%



Paginação multinível – cada entrada está dividida em p1(10bits)p2(10bits)offset(12bits)

Analogia a uma pesquisa numa árvore binária (mas com base != 2)

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

Vantagem : evitar ter todas as tabelas de páginas em memória, simultaneamente.

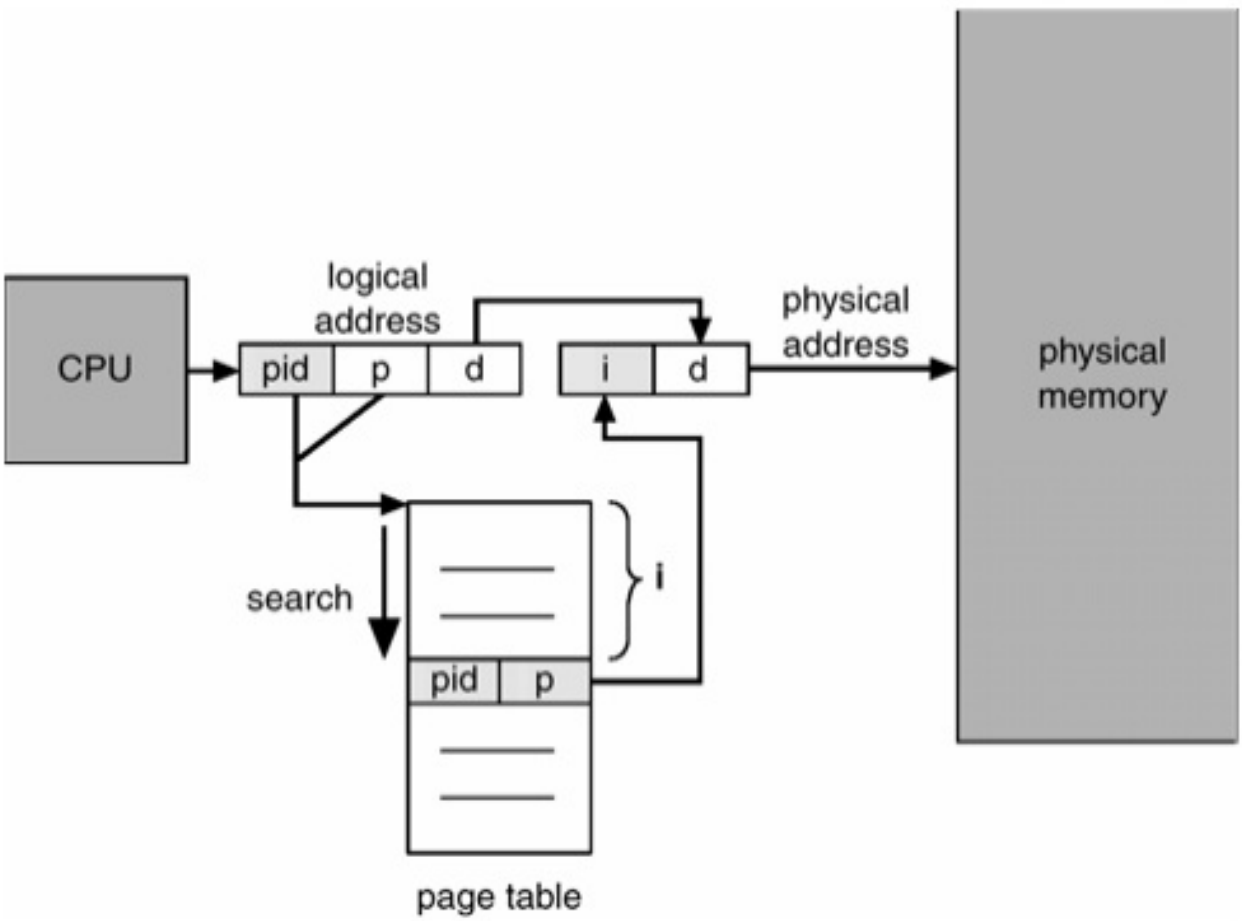
Paginação de páginas hashed

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 usando uma tabela de páginas invertida



Endereço lógico contém pid, p e d.

Usando pid e p como chave na tabela de páginas, descobrir o offset dessa “chave”. Será o índice para o endereço físico (concatenado ao offset dado pelo endereço lógico)

- Só existe 1 tabela com tamanho fixo

Desvantagens:

- A tabela de páginas deixa de conter informação acerca do espaço de endereçamento lógico de um processo (necessária quando a página referenciada não está em memória) ⇒ manter uma tabela de páginas convencional, por cada processo, em memória secundária
- Aumento do tempo de acesso à memória (devido ao acesso intermédio à tabela de hash ou a pesquisa sequencial). Solução : usar memória associativa para manter informação acerca dos acessos mais recentes

Tamanho da página

Fragmentação interna: diminui quando o tamanho da pagina diminui

Nº paginas/processos aumenta quando o tamanho da pagina diminui

Taxa de falta de paginas

- paginas pequenas -> taxa baixa
- paginas grandes -> taxa elevada

Nota: a taxa também depende do nº de quadros /processo – quando este aumenta a taxa de falta diminui

Segmentação

Dividir o programa e os dados em partes de tamanho diferente (segmentos).

Um segmento é uma unidade lógica. (uma função, um procedimento, as variáveis globais, a stack…)

Um endereço lógico é constituído por um par . Os segmentos são carregados em blocos de memória livres, não necessariamente contíguos.

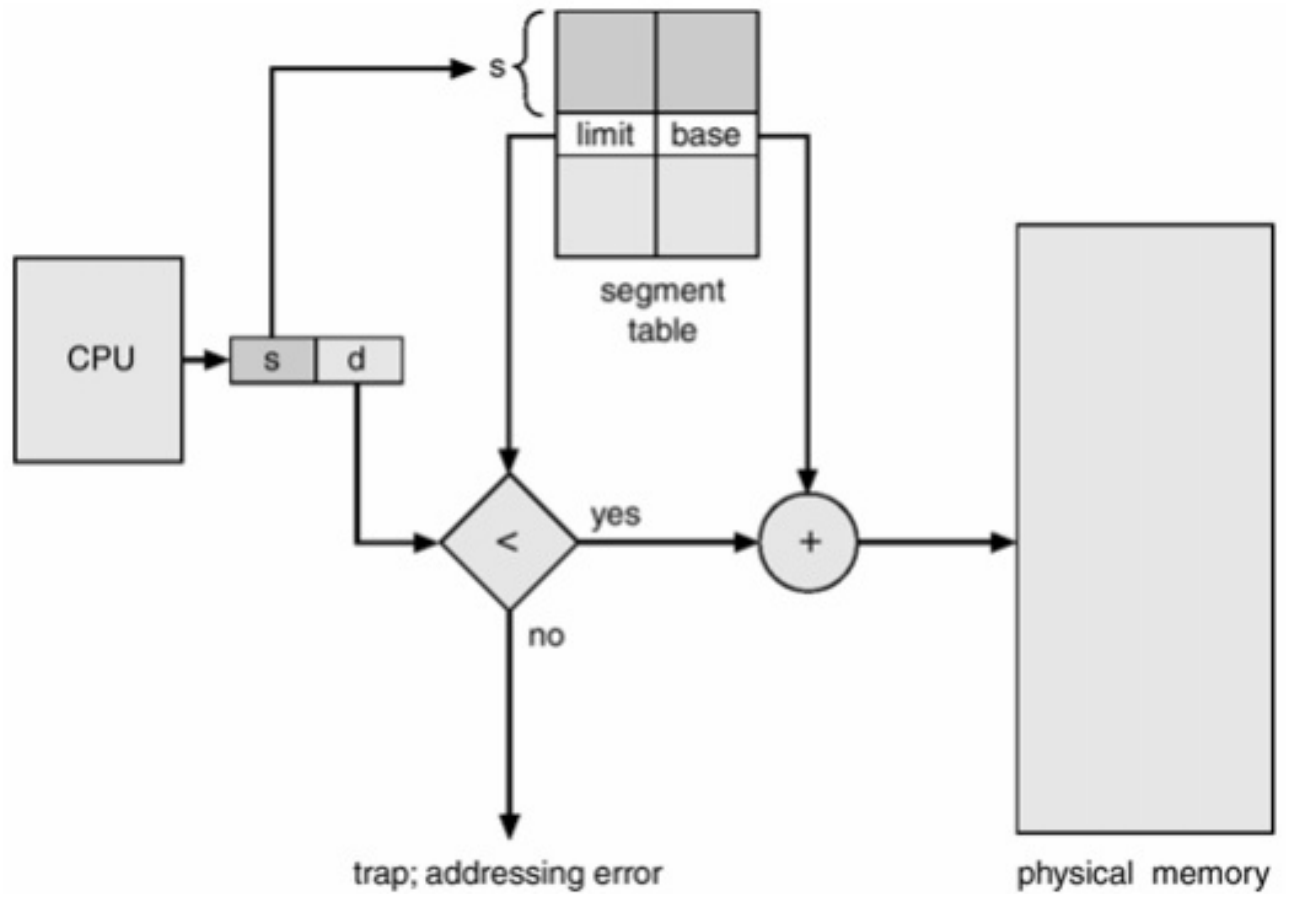
Cada entrada na tabela tem o endereço inicial do segmento e o seu comprimento

Tradução de endereço lógico para físico:

» Extrair, do endereço lógico, o número do segmento (bits mais significativos).

» Aceder à tabela de segmentos, usando este número, para obter o endereço físico do início do segmento

» Comparar o deslocamento (bits menos significativos do endereço lógico) com o comprimento do segmento; se aquele for maior do que este o endereço é inválido.



Notas:

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.

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 e a partilha (poupa memória)

Desvantagens:

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

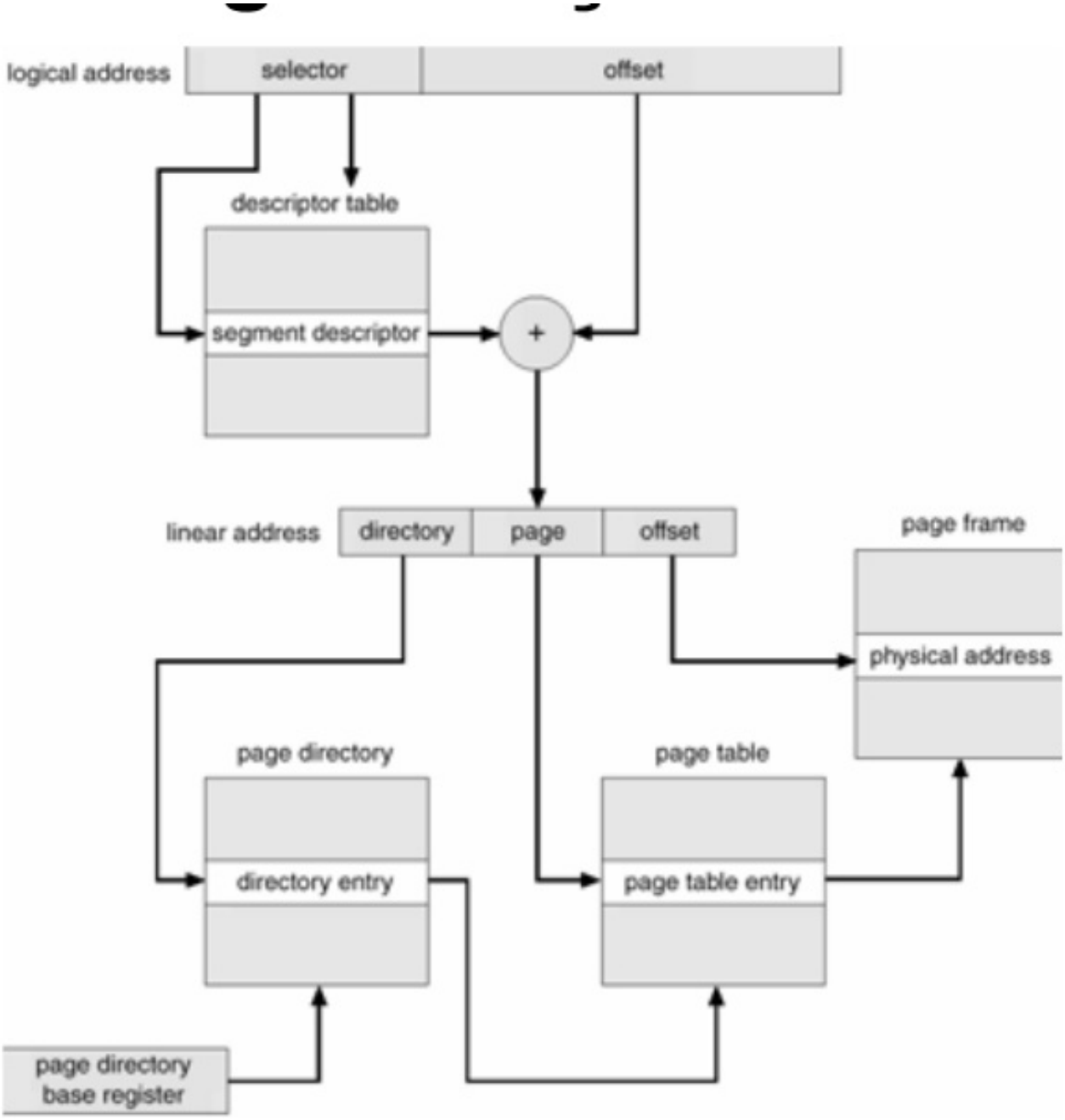
Segmentação com Paginação

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.



Gestão de Memória Virtual

||Memória Virtual||

- Técnica que permite a execução de processos que podem não estar completamente em memória principal.
- 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.

Faz com que o computador inspecione áreas da RAM que não foram utilizadas recentemente e copia estas para o disco, libertando espaço em RAM que pode assim voltar a ser utilizado.

O que é necessário:

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

||Overlays||

Substituir um bloco de instruções ou dados armazenados com outros.
Permite que programas sejam maiores que a memória principal do computador.

- 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 para a secção de overlay.
- O carregamento é feito dinamicamente: os procedimentos e dados são trazidos para 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.

||Paginação a Pedido||

Semelhante à paginação convencional excepto que as páginas só são transferidas para 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):

- Referência Inválida ⇒ abortar .
- Referência Válida e página não em memória ⇒ colocar em memória.

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

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

Falta de página (→ trap para o S.O.): quando um programa acede a uma página mapeada no espaço de memória virtual, mas que não foi carregada na memória física do computador.

- Verificar na tabela de páginas se a referência é válida ou inválida.
- Referência inválida ⇒ abortar
- Referência válida ⇒ continuar (trazer a página para memória principal) .
- Obter um frame livre .
- Ler a página necessária .

- Actualizar a tabela de páginas com indicação de que a página está em memória, e em que frame está.
- Recomeçar a instrução interrompida devido à falta de página.

Performance da paginação a pedido

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

- Tempo efectivo de acesso à memória(Team):
  $Team = (1-p) \times Tam + p \times Tfp$
    - p = taxa de falta de páginas
        - $0.0 \leq p \leq 1.0$
        - q p=0, não ocorrem.
        - p=1, todas as referências conduzem a falta de página.
    - Tam = tempo de acesso à memória.
    - Tfp = 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 Tfp pode ter um efeito significativo, pois é muito superior a Tam .

Estimação de Tfp (tempo de processamento de uma falta de página):

- Tempo de processamento de uma interrupção
- Tempo de carregamento da página
- Custo do recomeço da instrução.

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 para o espaço de swap e executar o carregamento das páginas a partir daí.

O que acontece se não houver frames/quadros livres?

||Substituição de página:||

- 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 último carregamento ⇒ escrever a página no disco.
- 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.

Algoritmos de Substituição de Página

- **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.

  A performance deste algoritmo é altamente influenciada pelo nº de frames disponíveis (maior número de frames tende a diminuir o número de faltas de página). No entanto mais frames não implica menos faltas de página (**Anomalia de Belady**).

- **ALGORITMO ÓPTIMO**

  - Substituir a página que não será usada por um período de tempo mais longo, no futuro.
  - Algoritmo frequentemente usado para avaliar a performance de outros algoritmos.

- **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 (usa o comportamento passado para prever o futuro).
  - Algoritmo usado frequentemente . Comportamento bastante bom.
  - **Problema:** manter e actualizar o tempo em que uma página foi referenciada. ⇒ gastos de tempo e de espaço.

- **ALGORITMOS QUE APROXIMAM O ALGORITMO LRU**

  - **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.

    - **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 conteúdo para 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 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, avançando para o frame seguinte.
        - Se o bit de referência ainda estiver em 0 na 2ª passagem, a página é substituída.

- **ALGORITMOS BASEADOS EM CONTAGENS**
  Manter um contador do nº de referências feitas a cada página.

    - **ALGORITMO LFU - Least Frequently Used**

        - Substituir a página com 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 com intervalos regulares.

    - **ALGORITMO MFU - Most Frequently Used**

        - Substituir a página com contagem mais elevada.

  Estes algoritmos são pouco utilizados.

||Buffering de páginas||

Uma página substituída não é imediatamente retirada da memória interna é colocada numa de 2 listas:

- 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(podem ser escritas em grupos).

Vantagem do buffering:

- Possibilidade de evitar buscar uma página ao disco se ela ainda estiver no buffer (cache) de páginas substituídas.
- Permite ler uma página do disco sem que a página substituída seja escrita no disco.

||Alocação de frames||

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:

- 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.

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.
O nº máximo é limitado pela memória física.

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:
    - 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 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 ↑)

ALCANCE DA SUBSTITUIÇÃO

- 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.

||Trashing||

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

Sintomas de trashing:

- Elevada actividade de transferência de páginas.
- Baixa utilização do processador.
- Baixa utilização de outros dispositivos.

O Sistema Operativo “pensa que” pode aumentar o grau de multiprogramação, outro processo é acrescentado ao sistema e a utilização do processador baixa ainda mais,resultando em colapso.

Causa:

- Uma má política de paginação.
- Se um processo não tiver páginas suficientes em memória a taxa de falta de páginas é muito elevada.

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

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.

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.

Evitar o thrashing:

- **Estratégia dos conjuntos de trabalho (Working-set strategy):**
  Trata de determinar simultâneamente:
    - Quantos frames alocar a um processo.
    - Que páginas manter nesses frames.

  Conjunto de trabalho (working set) de um processo num dado instante: conjunto de páginas referenciadas nas últimas Δ referências à memória.

  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.

  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.

  Problemas:
    - O passado nem sempre ajuda a prever o futuro.
    - Dificuldade em manter actualizado o conjunto de trabalho
    - Determinar o valor óptimo de Δ:
        - **Δ demasiado pequeno:** Pode não englobar toda uma localidade (páginas activamente usadas, em conjunto)
        - **Δ demasiado grande:** Pode englobar várias localidades e abranger mais páginas do que o necessário.

- **Estratégia da 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.

||Outras considerações||

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

- **Pré-paginação:**
  Procura evitar o elevado nº de faltas de página que surgem através da paginação a pedido carregando mais páginas do que as exigidas pela falta de página, procurando aproveitar o facto do carregamento consecutivo poder ser mais rápido do que o individual.

- **Interesse duvidoso:**

  - Pode ser vantajoso em algumas situações.
  - Pode acontecer que muitas das páginas carregadas não venham a ser usadas.

- **Tamanho da página:**
  Não existe um tamanho ideal.

- **Argumentos a favor de páginas pequenas:**

  - 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 páginas grandes:**

  - 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 página pequena)
  - Reduz o nº de faltas de página, a partir de certa dimensão das páginas.

- **Estrutura de um programa**
  A performance de um programa pode ser melhorada se o programador estiver consciente do modo como é feita a paginação.
  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. Certas linguagens fazem uso intensivo de apontadores.

- **Fixação de Páginas**

  - Alguns frames podem ser "fechados" (locked) isto é, as páginas neles contidas não podem ser substituídas ⇒ usar um lock bit.
  - Impedir que uma página recentemente carregada seja substituída antes de ser usada pelo menos uma vez.

Segmentação a pedido

Alguns processadores podem não suportar paginação mas suportar segmentação. 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 para o S.O) é necessário carregá-lo.
- Se houver necessidade de substituir um segmento para carregar outro usa-se um dos algoritmos de substituição descritos anteriormente.
- Usa-se um bit de referência para saber os segmentos que foram acedidos.

Maior diferença relativamente à paginação a pedido:

- Necessidade de compactação para 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).

Sistema de Ficheiros

Ficheiro

- programa ou conjunto de dados gravados em memória secundária e identificado por um nome.
- conjunto de dados persistentes, geralmente relacionados entre si e identificados por um nome.

Sistema de gestão de ficheiros / Sistema de ficheiros

- Parte do sistema operativo responsável pelos serviços de
    - **manipulação de ficheiros** (criação, destruição, escrita, leitura,cópia, …)
    - **controlo de acesso** aos ficheiros
    - **gestão de recursos**, garantindo a sua fiabilidade

Interface do sistema de ficheiros

Ficheiro

- **Tipos** : dados(numericos,caracteres,…) ; programas (fonte, objecto)
- **Atributos** : nome, tipo, localização, tamanho, data, dono … ; Mantidos na estrutura do directório.
- **Operações** : criação, abertura, fecho, leitura, escrita, posicionamento do apontador do ficheiro, ….
- **Tabela de ficheiros abertos** : é mantida em memória (acesso rápido); Open – criar entrada na tabela; Close – retirar elemento da tabela.

- **Estrutura de um ficheiro** : Vista pelo utilizador (coleçao de bytes e registos, estrutura complexa); Vista pelo sistema Operativo (coleçao de blocos)
- **Métodos de acesso** : Sequencial, Direto / aleatório , outros (Indexado, sequencial indexado, ...)
- **Facilidades fornecidas por alguns sistemas operativos** : Partilhar secções de um ficheiro entre vários processos; Fazer o lock de seções de um ficheiro aberto; Mapear secções.

Partições e Directórios

- **Necessidade de aceder aos ficheiros por nome** :
- **Necessidade de organização dos ficheiros** : partições de disco, directórios (eficiência, conveniência e organização)
- **Operações sobre directórios** : procurar ficheiro, criar / destruir ficheiro, listar, ...
- **Estrutura dos directórios** : unico nivel, em árvore, grafo acíclico, grafo genérico.
- **Diretórios com único nivel** : todos os ficheiros no mesmo diretório. **Problemas:** nomes únicos, tempos de pesquisa longos;
- **Diretório com estrutura em +arvore** : Estrutura mais comum. Um diretório contém subdiretórios. Usar um caminho para especificar o ficheiro (absoluto/relativo) o conhecido PATH.
- **Diretório em forma de grafo acíclico** : generalização da estrutura em árvore. Permite partilha de ficheiros/diretórios. Um ficheiro pode ter vários pathnames.
- ***Hard links*** : várias entradas de diretórios referenciam a mesma entrada do mapa de ficheiros (inode, UNIX) . Ex: **ln *exisiting_file new_file***
- ***Soft links*** : as entradas do diretório contêm pathnames (útil para referenciar noutros file systems) Ex: **ls -s existing_file new_file**
- **Evitam duplicação de dados.**
- **Facilitam a partilha de dados.**

Proteção

- **Proteger conta** : dano fisico (fiabilidade), acesso impróprio.
- **Proteger → controlar** : o que pode ser feito sobre o ficheiro/directório (Read, Write, Delete, List,...) e por quem.
- **Listas de acesso** : lista associada a cada ficheiro/directorio indicando o *username* e o tipo de acesso. **Problemas:** comprimento da lista e tamanho da variável.
- **Grupos de utilizadores (ex: UNIX)** : dividir os utilizadores em classes: dono, grupo, outros; o gestor do sistema cria os grupos e associa os utilizadores a um grupo; para cada ficheiro definir o tipo de acesso permitido (Read, Write, Execute)
- **Passwords** : associar uma password a cada ficheiro/directório.

Estrutura do sistema de ficheiros

- **I/O entre memória e disco –** feita em blocos para melhorar a eficiência. 1 bloco = N sectores.
- **Estruturação do sistema de ficheiros em níveis**
    - **Aplicações;**
    - **Sistema de ficheiros lógico;**
    - **Módulo de organização de ficheiros;**
    - **Sistema de ficheiros básico;**
    - **Controlo de I/O;**
    - **Dispositivos.**
- **Tabela de ficheiros abertos :** mantida em memoria pelo S.O.
- **Montagem (mount) de um sistema de ficheiros :** sistema de ficheiros tem de ser montado antes de poder ser usado.

Métodos de alocação

- **Como alocar espaço para os ficheiros, de modo a** :
    - usar o espaço do disco eficientemente;
    - aceder aos ficheiros rapidamente.
- **Vários métodos** : alocação contígua; alocação ligada; alocação indexada.

- Alocação contígua

- **Cada ficheiro é armazenado num conjunto de blocos contíguos do disco.**
- **A entrada do diretório para cada ficheiro deve indicar**
    - o endereço do bloco inicial
    - o numero de blocos
- **Vantagens**
    - simples
    - fácil acesso (sequencial / directo)
    - poucos posicionamentos (seeks) para aceder ao ficheiro
    - efeciente em aplicações que processam o ficheiro completo.
- **Dificuldades**
    - indicar o tamanho final do ficheiro quando ele é criado.
    - Tentação para alocar demasiado espaço → desperdício.
    - Encontrar espaço livre com a dimensão necessária.

    - Dificuldade de crescimento.
    - Apagamento de dados pouco eficiente.
    - Fragmentação externa → necessidade de compactação do espaço livre.

- Alocação ligada

Resolve todas as dificuldades da alocação contígua, mas tem outros problemas.

- Cada ficheiro é uma lista ligada de blocos
- Os blocos podem estar dispersos pelo disco.
- O diretório tem apontadores para o primeiro e o último bloco do ficheiro.
- **Vantagens**
    - Os ficheiros podem crescer dinamicamente.
    - É fácil inserir / eliminar blocos intermédios.
    - Não há fragmentação externa.
    - É fácil gerir o espaço livre.
- **Dificuldades**
    - O acesso direto é “pesado”.
    - Os apontadores consomem espaço de disco
    - Fiabilidade reduzida

Variante deste método

- Usar uma FAT (File Allocation Table) para manter informação de quais os blocos (clusters) que fazem parte de um ficheiro.
- Também mantém informação de quais são os blocos que estão livres ou em mau estado.
- Esta tabela é constituída por uma entrada por cada bloco do disco.
    - Cada entrada é indexada por um número de bloco e contém o endereço do bloco seguinte.
      O último bloco contém um valor especial (por ex: 1111111111…)
    - O directório contém o nº do 1º bloco do ficheiro.
- **Vantagens:**
    - Melhora o acesso aleatório.
    - Reduz o número de posicionamentos (a FAT pode estar em memória principal).

- Alocação indexada

- Cada ficheiro tem uma tabela de índices com tantas entradas quantos os blocos do ficheiro.
- O elemento *i* da tabela contém o endereço do bloco *i* do ficheiro.
- O directório contém o endereço da tabela de índices de cada ficheiro.
- **Dificuldades**
    - Conhecer previamente o tamanho do ficheiro para determinar o tamanho da tabela de índices.
        - Pode-se reservar, à partida, um bloco para a tabela de índices (bloco de índices) → o tamanho do ficheiro fica limitado.
        - Soluções para permitir o crescimento do ficheiro
    - Espaço ocupado pela tabela de índices.
    - Nº de posicionamentos elevado.
- **Vantagens**
    - Facilidade de crescimento do ficheiro (até um máximo especificado).
    - Acesso directo rápido.
    - Evita a fragmentação externa.
- **Soluções para permitir o crescimento do ficheiro**
    - **Lista ligada de índices:** Vários blocos de índices, constituindo uma lista ligada.
    - **Índice multinível** : Usar um índice de 1º nível que aponta para um índice de 2º nível que aponta para os blocos de dados. Extensível para N níveis.
    - **Índice direto, combinado com índice multinível:** As primeiras entradas do bloco de índices apontam directamente para blocos de dados. As últimas entradas do bloco de índices apontam para outros blocos de índices.
    - **Índice directo, combinado com índice multinível :**
        - **Vantagens**
            - Possibilidade de crescimento dos ficheiros. Existe um tamanho máximo que é suficiente para a maioria das aplicações.
            - Acesso rápido para ficheiros pequenos (acedidos usando apontadores diretos)
        - **Dificuldades**
            - Acesso a ficheiros grandes não é muito eficiente.
            - Muitos posicionamentos.
        - **Exemplo: Inodes do UNIX**

        - Cada ficheiro, directório ou dispositivo de I/O tem um Inode associado.
        - Cada entrada do diretório aponta para o Inode respetivo.
        - Cada Inode contém:
            - tipo de ficheiro: regular, diretório, block special, character special
            - permissões de acesso
            - identificador do dono e do grupo
            - contagem de hard links
            - data e hora da última modificação
            - localização dos blocos (só para ficheiros regulares e diretórios)
            - major e minor device numbers (só para ficheiros especiais)
            - valor do symbolic link (se for um symbolic link).

Performance dos Métodos de Alocação

- Depende do tipo de acesso mais usado: sequencial ou directo.
- **Alocação contígua**
    - boa para qualquer tipo de acesso
- **Alocação ligada**
    - boa para acesso sequencial
    - má para acesso aleatório
- **Alocação indexada**
    - a performance depende de
        - nº de níveis dos índices
        - tamanho do ficheiro
        - posição do bloco desejado

Gestão do espaço livre

- **Técnicas:**
    - tabela de bits
    - lista ligada de blocos livres
- **Tabela de bits**

    - 1 bloco ↔ 1 bit (ex: 1 bloco ocupado; 0 bloco livre)
    - Vetor de bits para representar todos os blocos do disco.
    - **Vantagens:**
        - simples de implementar
        - ocupa pouco espaço; pode ser mantido em memória
        - fácil encontrar o 1º bloco livre ou conjunto de blocos livres consecutivos
    - **Dificuldades**
        - o mapa de bits também tem de ser mantido no disco;
          as cópias em memória e no disco têm de ser consistentes.
- **Lista ligada de blocos livres**
    - Ligar, através de apontadores, todos os blocos livres, mantendo numa posição especial do disco, um apontador para o 1º bloco livre.
    - **Vantagem**
        - não ocupa espaço
    - **Dificuldades**
        - travessia da lista pouco eficiente (operação rara, a menos que se queira encontrar n blocos livres consecutivos)
    - **Variante deste método**
        - O 1º bloco livre contém os endereços de n blocos livres.
        - Os primeiros n-1 endereços indicam blocos livres, de facto.
        - O n-ésimo endereço indica um bloco livre com uma constituição semelhante a este
        - Vantagem (relativamente à lista ligada simples):
            - os endereços de um grande nº de blocos livres podem ser encontrados rapidamente.

Implementação dos Directórios

- **Lista linear**
    - Cada elemento da lista deve conter pelo menos o nome do ficheiro e um apontador para os blocos de dados.
    - Em alguns sistemas, alguma da informação associada ao ficheiro é guardada num cabeçalho do ficheiro.
    - (-) O tempo de pesquisa necessário para encontrar um ficheiro pode ser grande.

- **Tabela de hash**
    - A tabela de hash tem como entrada um valor calculado a partir do nome do ficheiro e retorna um apontador para a entrada correspondente ao ficheiro, numa lista linear.
    - Diminui o tempo de pesquisa de um ficheiro.
    - Necessário tratar situações de colisão, quando 2 ficheiros dão origem ao mesmo índice.

Eficiência e Performance

- **Eficiência na utilização do disco depende de:**
    - Algoritmos de alocação de espaço para os ficheiros e de manipulação de directórios
        - ex: para reduzir fragmentação interna, usar 2 tamanhos de clusters:
            - (o último cluster de um ficheiro pode ser mais pequeno que os outros)
            - 4.2 BSD → blocos de 4KB e fragmentos de 1KB
                - 1 ficheiro com 18KB ocupa 2 blocos + 2 fragmentos
    - Tipos de dados mantidos em cada entrada do directório
        - ex: data da última modificação → importante para os backup's mas... data da última leitura (ocupa tempo e espaço, necessário ?)
    - Tipo de apontadores usados para aceder aos dados
        - Maior nº de bits →
            - maior espaço consumido na manutenção de índices, listas ligadas, …
              (ex: métodos de alocação, gestão do espaço livre)
            - mas ... possibilidade de usar discos maiores sem aumentar a fragmentação interna
              Ex: FAT12, FAT16, FAT32
- **Algumas formas de a melhorar:**
    - **cache do controlador**
        - para manter uma pista do disco a partir do sítio onde a cabeça ficou, após o posicionamento
    - **cache de disco**
        - para manter blocos acedidos recentemente
    - **no acesso sequencial**
        - técnica de free-behind

            - remover um bloco do buffer logo que se lê outro bloco do disco
        - técnica de read-ahead
            - ler o bloco pedido e alguns que se lhe seguem
    - **RAM disk ou disco virtual**
        - aceita as operações comuns sobre discos mas efectua-as sobre RAM
        - → device driver adequado
        - útil para ficheiros temporários (ex: usados por um compilador)

Recuperação

- **Verificação de consistência**
    - correr um programa que compara os dados na estrutura de directórios com os dados nos blocos de dados e tenta determinar inconsistências.
        - ex: fsck(file system check) do UNIX; scandiskdo MS Windows
- **Backup e Restore**
    - ciclo de cópias integrais (todos os ficheiros) e incrementais (apenas os ficheiros alterados desde a última cópia) usando diferentes discos/fitas que permite recuperar qualquer ficheiro acidentalmente destruído durante o ciclo.
- **Journaling**
    - registar automaticamente as operações executadas sobre o sistema de ficheiros, mantendo um registo contínuo dessas operações num arquivo ("journal")
    - após uma falha, este registo pode ser usado para repor o sistema de ficheiros num estado conhecido, consistente.
    - sistemas de ficheiros que journaling : ext3, ext4, NTFS, e outros