


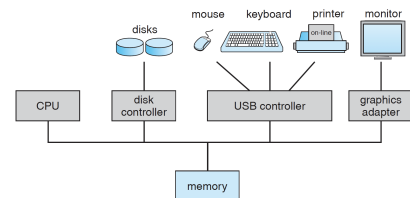


Sistemas Operativos

RESUMOS S0

INTRODUÇÃO

 <https://ops4you.notion.site/RESUMOS-S0-3209609fca63452985825b6045c67c47>



Conceitos Básicos

O que é um Sistema Operativo?

Motivação:

- O hardware por si só não é fácil de usar, sendo necessário algum tipo de software.
- Diferentes programas, geralmente requerem operações comuns para operar os recursos de hardware.
- Uma boa ideia seria juntar num único programa as funções comuns para o controlo e alocação dos recursos de hardware.
- Este programa comum é conhecido como Sistema Operativo.

Como podemos definir um Sistema Operativo ou parte dele?

- Não existe uma definição universalmente aceite.
- Uma definição mais restrita seria "Aquele programa que corre o tempo todo num computador", tudo o resto ou é um programa de sistema ou uma aplicação.

Um Sistema Operativo é o que gere o hardware do Computador:

- Atua como intermediário entre o utilizador e o hardware.
- Fornece o básico para aplicações.
- Alguns sistemas operativos são projetados para tornar o uso do sistema conveniente, outros para tornar o uso do hardware mais eficiente e outros combinam os dois casos.

O que devem fazer os Sistemas Operativos?

- Um sistema operativo foi concebido para maximizar a utilização de recursos e para garantir que todo o tempo de CPU, memória e I/O são usados de forma eficiente e justa entre todos os utilizadores.
- O Sistema Operativo é projetado, principalmente, para facilitar a utilização de um sistema, ou seja, contém um maior foco na performance do que na utilização de recursos.
- Configurado para correr com a mínima, ou até mesmo nenhuma, intervenção do usuário.

Um Sistema pode ser dividido em 4 componentes principais:

- **Hardware:** Os recursos básicos do sistema (CPU, memória, dispositivos I/O, ...);
- **Sistema Operativo:** Controla e coordena o uso do hardware entre os diferentes programas e utilizadores;
- **Programas:** Definem a maneira como os recursos do sistema são usados para resolver as necessidades do utilizador (processadores de palavras, browsers, bases de dados, jogos, compiladores);
- **Utilizadores** .

Princípios de um Sistema Operativo:

Sistema Operativo como um alocador de recursos/controlador:

- Atua como um gestor de todos os recursos: Tempo de CPU, espaço em memória, dispositivos I/O.
- Decide entre pedidos em conflito para um uso de recursos mais eficiente e justo.
- Previne erros e utilização "imprópria" do computador.
- Especialmente importante quando muitos utilizadores têm acesso aos mesmos recursos.

Sistema Operativo como um facilitador:

- Fornece serviços que todos precisam.
- Torna a programação um processo mais fácil, rápido e menos propenso a erros.
- Especialmente preocupado com a execução de vários programas.

Muitos SO aplicam os princípios anteriores:

- **Exemplo:** O gestor de ficheiros é necessário para todos os utilizadores (facilitador) e tem de ser eficiente e seguro (controlador).

Principais Componentes.

Os SO modernos, normalmente, têm como principais componentes os seguintes:

- Gestor de Processos;
- Gestor de Memória;
- Gestor de Armazenamento;
- Gestor de I/O (Input/Output);

Gestor de Processos:

Funções Principais:

↳ um programa em execução tem atividade própria

- Criar, suspender, resumir e terminar processos (Utilizador/Sistema);

- Fornecer mecanismos para a comunicação de processos;
- Fornecer mecanismos para a sincronização de processos;
- Fornecer mecanismos para a resolução de problemas ligados ao bloqueio de processos;

Gestor de Memória:

Funções Principais:

- Alocar e desalocar espaço na memória conforme for necessário;
- Rastreio das partes de memória que estão atualmente a ser utilizadas e por quem.
- Decidir que processos/dados mover para dentro/fora da memória e quando o fazer.

Gestor de Armazenamento:

Funções Principais:

- Fornece uma visão uniforme (abstrata) e lógica da informação no armazenamento (visão em ficheiros e diretorias lógicas).
- Apoia `primitivas` para criar, apagar e manipular ficheiros ou diretorias.
- Apoia o controlo de acesso para determinar quem tem, ou não, acesso a um determinado ficheiro ou diretoria.
- Mapeamento em suportes de armazenamento secundário.

Gestor de I/O :

Funções Principais:

Esconder peculiaridades de dispositivos de hardware do utilizador:

- Geralmente interface dispositivo/driver.

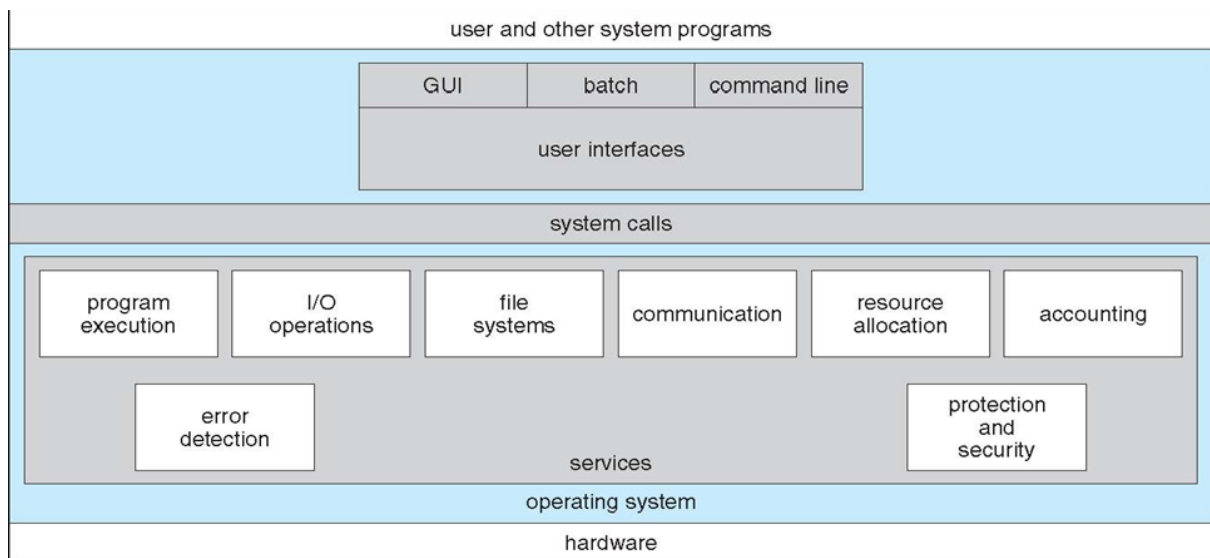
- Drivers para dispositivos de hardware específicos.

Responsável pela gestão de memória de I/O, incluindo:

- Buffering - Armazenamento temporário de dados enquanto estão a ser transferidos.
- Catching - Armazenamento de partes de dados em armazenamento rápido para melhorar a performance.
- Spooling - Sobreposição de um output de um processo com o input de outros.

Serviços Comuns de um SO:

SO fornecem um ambiente para a execução de programas oferecendo, para isso, serviços específicos para programas e para utilizadores



Os serviços fornecidos variam entre SO mas existem alguns comuns a todos:

- **Interfaces de Utilizador** : Permitem o funcionamento e controlo eficazes do sistema.
- **Execução de Programas** : Para carregar um programa em memória e corrê-lo.

- **Operações de I/O** : Fornece um meio para as operações ligadas a I/O.
- **Sistema de Ficheiros** : Permite uma manipulação eficaz de ficheiros e diretorias.
- **Comunicações** : Permite a troca de informação entre processos no mesmo dispositivo ou entre dispositivos.
- **Deteção de erros** : Para estar constantemente ciente de possíveis erros que podem ocorrer no CPU|Memória|I/O ou programas do utilizador de forma a tomar a ação adequada para assegurar uma computação correta e consistente.

Outros Serviços existentes não para proveito do utilizador mas do sistema em si:

- **Alocação de Recursos** : Quando múltiplos processos se encontram a correr de forma concorrente, os recursos disponíveis (CPU/Memória, Armazenamento de Ficheiros, ...) devem ser alocados de forma eficiente para cada um dos processos.
- **Accounting** : Para acompanhar a quantidade e os tipos de recursos utilizados por cada utilizador.
- **Proteção e Segurança** : Para evitar a interferência entre processos concorrentes ou com o próprio S0 e proteger o sistema contra "outsiders".

Multiprogramming

- Um dos aspetos mais importantes de um S0 é a capacidade de ter múltiplos programas a correr simultaneamente.
- Multiprogramming aumenta a utilização do CPU pela organização de processos para que o CPU possa sempre executar um processo.

A ideia é a seguinte:

- O S0 começa a executar processo a processo ordeiramente;
- Eventualmente, algum processo terá de esperar por alguma tarefa (por exemplo, uma operação I/O);

- Se não existisse multiprogramming, a CPU teria de ficar inativa enquanto esperava.
- Num sistema multiprogramming, o SO troca para outro processo. Quando um processo precisa de esperar por algo, a CPU volta a trocar para outro processo e assim sucessivamente. Eventualmente, o primeiro processo recebe as informação pelas quais se encontrava a aguardar e recupera a CPU.
- Assim sendo, desde que, pelo menos, um processo precise de ser executado a CPU nunca fica inativa.

Multitasking

- Multitasking é uma extensão logica de multiprogramming que aumenta o tempo de resposta na qual a CPU troca entre processos, permitindo ao utilizador interagir com o processo enquanto este se encontra a correr.

Arranque do Sistema

Bootstrap(Firmware):

É carregado quando se liga ou é dado reboot ao sistema.

- Inicializa todos os aspetos do sistema;
- Carrega o Kernel do Sistema Operativo e começa a executá-lo;
- Estando o Kernel carregado, o sistema pode começar a fornecer serviços ao utilizador.

Completada esta fase, o sistema está completamente iniciado e preparado para a ocorrência de qualquer evento.

Proteção da CPU

Não podemos permitir que um programa fique "preso" ou falhe e nunca devolva o controlo do sistema ao utilizador. Para evitar

isso podemos utilizar um `Timer` ;

Timer:

Pode ser configurado para interromper um processo depois de um certo período de tempo.

Se o timer interromper algum processo, o controlo é transferido de imediato para o S0 sendo a interrupção tratada como um "fatal error".

System Calls

As `System Calls` funcionam como uma interface para os serviços do S0.

São, na maior parte dos casos, acedidas através de uma API em vez de uma chamada direta ao sistema.

Por que razão é melhor usar uma API ao invés de uma chamada direta ao sistema?

- O mesmo programa deve compilar e correr num outro sistema que suporta essa API.
- As System Calls podem ser mais detalhas e difíceis de trabalhar quando comparadas às API disponíveis para o programador.

System Call -> Implementação

Tipicamente, um número é associado a cada System Call, em seguida, esse número é mantido numa tabela indexada de acordo com os números atribuídos a cada System Call.

Exemplos de Tipos de System Calls:

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Como podemos ver pela imagem, podem ser agrupadas em 6 categorias:

- Controlo de Processos;
- Manipulação de Ficheiros;
- Manipulação de Dispositivos;
- Manutenção de informação;
- Comunicação (Pipes);
- Proteção;

Ficheiros

Em Unix, tudo são ficheiros, logo a gestão deles é do mais importante do SO, sendo por isso, a primeira primitiva ensinada nas aulas práticas (Guião 1).

Descritor de Ficheiro

- Representação abstrata de um ficheiro utilizada para operar sobre o mesmo.
- Faz parte da interface POSIX.
- Representado por um inteiro não negativo.
- Pode também servir para representar outros recursos de I/O como pipes, sockets, dispositivos de E/S (p.e. teclado).

Descritores standard

Os descritores standard são os seguintes:

- 0 -> Standard input -> `STDIN_FILENO` em `<unistd.h>` -> `stdin` em `<stdio.h>`
- 1 -> Standard output -> `STDOUT_FILENO` em `<unistd.h>` -> `stdout` em `<stdio.h>`
- 2 -> Standard error -> `STDERR_FILENO` em `<unistd.h>` -> `stderr` em `<stdio.h>`

NB: Estes descritores podem ser redefinidos (Guião 4).

Estruturas Kernel

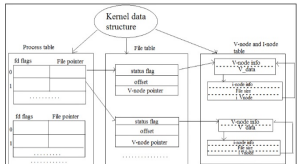
Tabela de Processo (TP)

- Uma tabela por processo.
- Guarda descritores de ficheiros abertos.
- Pode-se usar o comando `ulimit -n` em UNIX para saber quantos ficheiros podemos ter abertos.

Tabela de Ficheiros (TF)

- Tabela partilhada pelo Sistema Operativo.
- Guarda o modo de abertura e a posição de leitura/escrita de cada descritor.

V-node



- Abstração de um objeto Kernel que respeita a interface de ficheiro UNIX.
- Permite representar ficheiros, diretorias, FIFOs, domain sockets, etc...
- Guarda informação do tipo de objeto, apontadores para as funções sobre o mesmo e para o respetivo i-node.

I-node

- Guarda metadados/atributos dos ficheiros (p.e. nome do ficheiro, tamanho, ...).
- Guarda localização dos dados no recurso físico de armazenamento.

NB: Em Linux, os i-nodes servem também como v-nodes, não havendo uma implementação explícita para os v-nodes.

Notas:

- Entradas na tabela de ficheiros do sistema podem partilhar i-nodes.
- Descritores de processos distintos (p.e. via `fork()`) podem partilhar entradas na tabela de ficheiros de sistema.
- Descritores do mesmo processo (p.e. via `dup()`) podem partilhar entradas na tabela de ficheiros de sistema.

API

Open

```
int open(const char* path, int oflag [, mode]);
```

- Inicializa um descritor para um determinado ficheiro; *→ "abrir um ficheiro"*
- `path` -> caminho do ficheiro.
- `oflag` -> modo de abertura (`O_RDONLY`, `O_WRONLY`...).
- `mode` -> permissões de acesso para `O_CREAT`.

Valor de Retorno:

- `>= 0`: um descritor de ficheiro que pode mais tarde ser passado a várias funções para interagir com o ficheiro aberto
- `= -1`: indica que ocorreu algum erro

Flags:

- `O_RDONLY` - Abrir apenas para leitura.
- `O_WRONLY` - Abrir apenas para escrita.
- `O_RDWR` - Abrir para leitura e escrita.

Indicam o modo de abertura do ficheiro e não podem ser misturadas umas com as outras.

As outras flags que podem ser passadas indicam opções:

- `O_CREAT` - Cria o ficheiro se ele não existir.
- `O_APPEND` - Começa a escrever no fim do ficheiro em vez de no início.
- `O_TRUNC` - Apaga todo o conteúdo do ficheiro.

→ começando a poder escrever no início

Permissões de acesso

Permite especificar as permissões com que o ficheiro deve ser criado, este é o número octal que pode ser passado para o `chmod` para especificar as novas permissões do ficheiro.

NB:

- Em C um octal é começado por um 0. Modos de abertura
- Se não forem especificadas permissões aquando da criação do ficheiro, as permissões com que ele é criado também não são especificadas, podendo ser qualquer coisa.

Read

```
ssize_t read(int fd, void *buf, size_t nbyte)
```

- `fd` - Descritor de ficheiro de onde ler.

- `buf` - Buffer/array para onde o conteúdo é lido.
- `nbyte` - Quantos bytes devem ser lidos no máximo para dentro do buffer.

Valor de Retorno:

- `> 0` - Indica quantos bytes foram lidos.
- `= 0` - Indica que o ficheiro terminou.
- `= -1` - Indica que ocorreu algum erro.

Write:

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- `fildes` - Descritor de ficheiro para onde escrever.
- `buf` - Buffer/array onde ir buscar os dados que devem ser escritos.
- `count` - Quantos bytes do `buf` devem ser escritos.

Valor de Retorno:

- `>= 0` - Indica quantos bytes foram escritos.
- `= -1` - Indica que ocorreu algum erro.

Close:

```
int close(int fd);
```

- `fildes` - Descritor de ficheiro a fechar

Valor de Retorno:

- `0` - Se foi fechado com sucesso.
- `1` - Se ocorreu algum erro.

Razões para o fazer:

- razão para fazer close
- **Flushing.** Não é garantido que mal um `write()` seja executado que o texto seja escrito disco. Ao fechar o ficheiro temos essa garantia.
 - Cada processo tem um máximo de descritores de ficheiro que pode ter abertos ao mesmo tempo.

Posição (offset)

- A cada operação de leitura/escrita efetuada sobre o mesmo descritor, a posição a ler/escrever é atualizada consoante o número de bytes efetivamente lidos/escritos.

Performance

As *System Calls* `read` e `write` são das mais lentas e mais utilizadas, pelo que, é necessário ter o cuidado de tentar reduzir o número de vezes que são chamadas.

Processos

Conceito:

Um Processo é considerado uma unidade de trabalho na maioria dos sistemas e pode ser considerado como um programa em execução.

Um processo necessita de recursos para concretizar as suas tarefas, entre os quais:

- Tempo de CPU;
- Memória;
- Ficheiros;
- Dispositivos de I/O;
- ...

Recursos podem ser alocados para um processo no momento em que este é criado ou enquanto este está a executar.

Um processo tem múltiplas partes:

- **Secção de Texto** -> Contém o código do programa;
- **Secção de Dados** -> Contém as variáveis globais;
- **Heap** -> Contém memória alocada dinamicamente durante o runtime do processo;
- **Stack** -> Contém dados temporários (parâmetros de funções, endereços de retorno e variáveis locais).

Programas vs Processos:

Programa:

Uma entidade passiva (chamada maior parte das vezes de executável) que contém uma lista bem definida de instruções.

Processo:

Uma entidade ativa correspondente a uma sequência de execução de um programa.

Um programa torna-se um processo quando está carregado na memória.

- Um programa pode ser um conjunto de vários processos.
- Dois processos associados com o mesmo programa são considerados duas sequências de execução separadas.

Exemplo:

Se o mesmo utilizador abrir várias cópias do programa Firefox:

- Cada cópia é considerada um processo diferente e embora o código seja equivalente, os dados, a heap e a stack podem variar.

Estados de um Processo:

À medida que um processo executa, vai mudando o seu estado de acordo com a atividade atual.

Um processo pode ser encontrado num dos seguintes estados:

- **New** -> O processo está a ser criado.

fork → cria um novo processo → pai / filho

exec → substitui o programa do processo por um outro

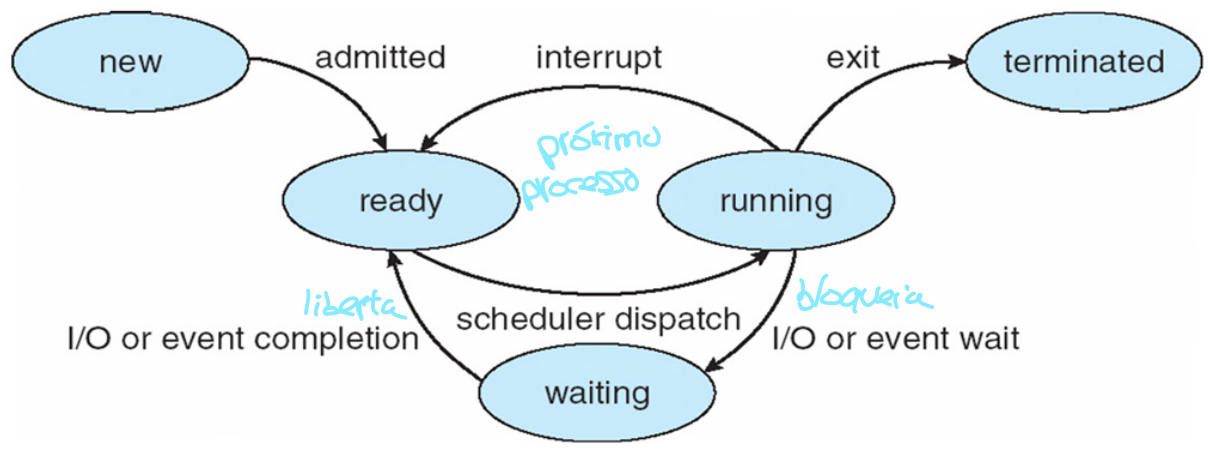
espera passiva → não consome CPU

→ e tem acesso ao CPU

→ se precisar de CPU por exemplo

- **Running** -> Instruções estão a ser executadas.
- **Waiting/Blocked** -> O processo está à espera que algo ocorra.
- **Ready** -> O processo está à espera de ser atribuído a um processador. → fila de espera com processos prontos para executar
- **Terminated** -> O processo terminou a sua execução.

É importante perceber que muitos processos podem estar no estado Ready ou Waiting, no entanto, apenas 1 processo pode estar a correr em cada core de cada vez.



Criação de Processos

Durante a sua execução, um processo pode criar vários novos processos.

- Chama-se ao processo criador "processo pai" e aos processos criados "processos filhos".
- Cada novo processo originado pelo pai pode criar outros novos processos sendo criada uma "árvore de processos".

A maioria dos SO identifica os processos de acordo com um único **process identifier** (pid), que tipicamente é representado por um inteiro.

- O pid fornece um único valor por cada processo no sistema e pode ser usado como um índice para aceder a vários atributos de um processo.

Alternativas da criação de Pai/Filho

Partilha de recursos:

Existem 3 casos possíveis no que toca à partilha de recursos

- Processos pais e filhos partilham todos os recursos;
- Processos filhos partilham um subconjunto de recursos dos pais;
- Processos pais e filhos não partilham qualquer recurso.

Execução:

Existem 2 casos:

- Pais e filhos executam de forma concorrente.
- Pais esperam até que um ou todos filhos acabem de executar.

Espaço de Endereço:

- Pais e filhos são duplicados (o filho começa com o programa/dados do processo pai (herança)).
- O processo filho contém um novo programa carregado dentro dele.

Quanto "custa" criar um processo?

Copiar o estado de I/O dos pais (Dispositivos alocados, lista de ficheiros abertos, etc...)

- Custo médio.

Configurar novas tabelas de memória para o espaço de endereço.

- Custo elevado.

Copiar dados do processo pai

- Originalmente, com um custo muito elevado
- O custo diminui com a utilização do copy-on-write

Criação de Processos - Unix/Linux

Em sistemas Unix/Linux, é possível criar novos processos através da System Call `fork()` ;

- O novo processo (filho) consiste numa cópia do endereço do processo original (pai);
- O valor de retorno do `fork()` é zero para o filho e o process identifier (pid) do filho é retornado para o pai;
- Ambos os processos (pai e filho) continuam a execução concorrente das instruções depois do `fork()` ;
- Este mecanismo permite ao processo pai comunicar de forma mais fácil com o processo filho.

```
// código do processo pai antes do fork()
if ((pid = fork()) < 0) {
    ... // fork falhou
} else if (pid == 0){
    ... // código que o processo filho deve executar
} else{
    ... // código que o processo pai deve executar
}
```

Normalmente depois de um `fork()`, um dos dois processos (pai ou filho) utiliza a System Call `exec()`

- `c execl("/bin/ls", "ls", NULL);` \\\\ execução do comando ls (fins explicativos)

Exec() :

Esta System Call substitui o espaço de memória do processo -> texto, dados, heap, stack -> com um novo programa vindo do disco começando a executar este novo programa na sua main, destruindo a imagem anterior do processo.

- Desta maneira, pais e filhos são capazes de seguir caminhos diferentes.

Fim de um processo

Um processo termina a sua execução quando explicitamente o mesmo utiliza a System Call `exit()` ou quando executa a última instrução (neste segundo caso, a chamada do `exit()` está implícita pelo uso de `return` na `main()`).

- Todos os recursos dos processos -> incluindo memória virtual/física, ficheiros abertos e I/O buffers -> são desalocados pelo SO.

Um valor de estado `exit()` (normalmente um inteiro) é disponibilizado ao processo pai através de outra System Call -> `wait()`;

Um processo pode terminar a sua execução através do seu processo pai.

Quando é que isso acontece?

- O processo filho excedeu o número de recursos alocados.
- A tarefa atribuída ao processo filho não é mais necessária.
- Se o processo pai está a dar `exit()` o sistema operativo não permite que o processo filho continue a sua execução sem o processo pai (cascading termination).

Se um processo terminou e nenhum processo pai se encontra à espera, esse processo entra num estado conhecido como estado zombie (Zombie Process).

- Apenas quando o pai chama `wait()`, o pid do processo zombie e a sua entrada na tabela são lançados.

Se um processo pai terminar antes do seu processo filho, então todos os processos filhos ficam conhecidos como processos órfãos (orphan processes).

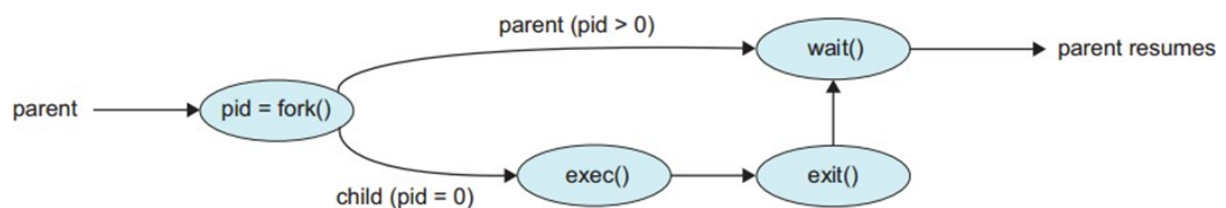
- Nos sistemas Unix os processos órfãos são abordados atribuindo o processo `init` como o novo progenitor desses processos.
- O processo `init` periodicamente usa `wait()`, permitindo a libertação do pid e da entrada na tabela do processo órfão.

Em sistemas Unix, podemos terminar um processo usando a invocação da System Call `exit()`, fornecendo um status como parâmetro.

- `exit(status);`

Para esperar pelo fim do processo filho e obter o exit status, podemos utilizar a System Call `wait()`, que também retorna o pid do processo que termina para que o pai seja capaz de dizer qual dos seus filhos é que já terminou a sua execução

- `pid = wait(&status)`



Forking do comando 'ls' - Unix

```
int main(){

    pid_t pid;

    /* criação do processo filho */
    pid = fork();

    if(pid < 0) { /* Ocorreu um erro */
        perror("Fork");
        return 1;
    }
    else if (pid == 0){
        /* Código a ser executado pelo processo filho */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* Processo pai */
        /* O pai irá esperar pelo processo filho antes de iniciar a sua execução */
        wait(NULL);
        fprintf(stderr, "Fork completed");
    }

    return 0;
}
```

Comunicação entre Processos

Os processos dentro de um sistema podem ser independentes ou cooperar:

- **Processos Independentes** : Não podem afetar nem ser afetados por outras execuções.
- **Processos Colaborativos** : Podem ser afetados por outras execuções.

Principais razões para os processos de cooperação:

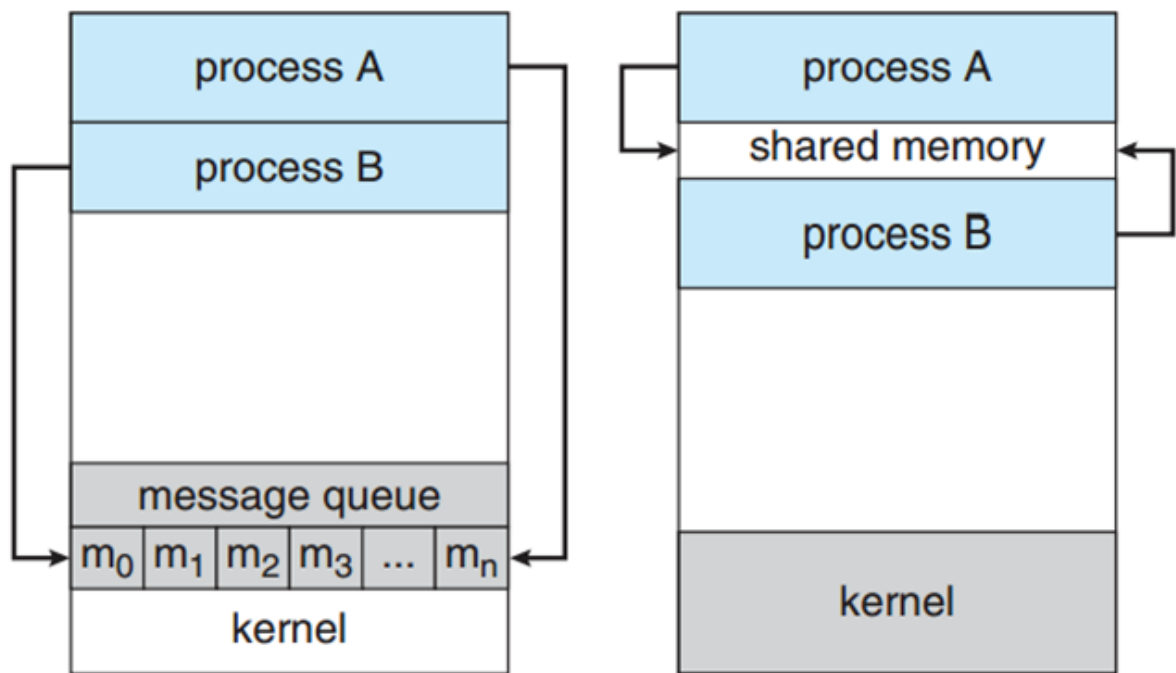
- **Partilha de Informação** -> Acesso concorrente ao mesmo pedaço de informação;
- **Modularidade** > Quebrar o cálculo em tarefas mais pequenas que fazem mais sentido;
- **Speedup** -> Executar tarefas mais pequenas concorrentes de forma paralela.

Para trocarem dados e informação, processos de cooperação precisam de suportar mecanismos de IPC (interprocess communication). Existem dois modelos fundamentais:

- **Troca de mensagens (Message passing)** -> Comunicação via enviar/receber mensagens;
- **Memória Partilhada (Shared Memory)** -> A comunicação ocorre simplesmente por ler/escrever para a memória partilhada.

NB:

Memória Partilha pode levar a problemas de sincronização complexos. No entanto é mais eficiente comparativamente à troca de mensagens.



Pipes

Contexto:

Pipes foram uns dos primeiros mecanismos de IPC e fornecem uma das formas mais simples para a comunicação entre processos.

Pipes permitem a comunicação num estilo standard (Producer - Consumer):

- O producer escreve para uma ponta do pipe (the write-end);
- O consumer lê pela outra ponta do pipe (the read-end);

Os pipes são unidirecionais, apenas permitem comunicação em um sentido.

NB:

Se for necessária comunicação nos dois sentidos, devem ser usados dois pipes, onde cada pipe envia dados em direções diferentes.

Pipes podem ser construídos no terminal usando o caractere `|`:

- `ls | sort`
- `cat fich.txt | grep xpto`

Pipes - UNIX

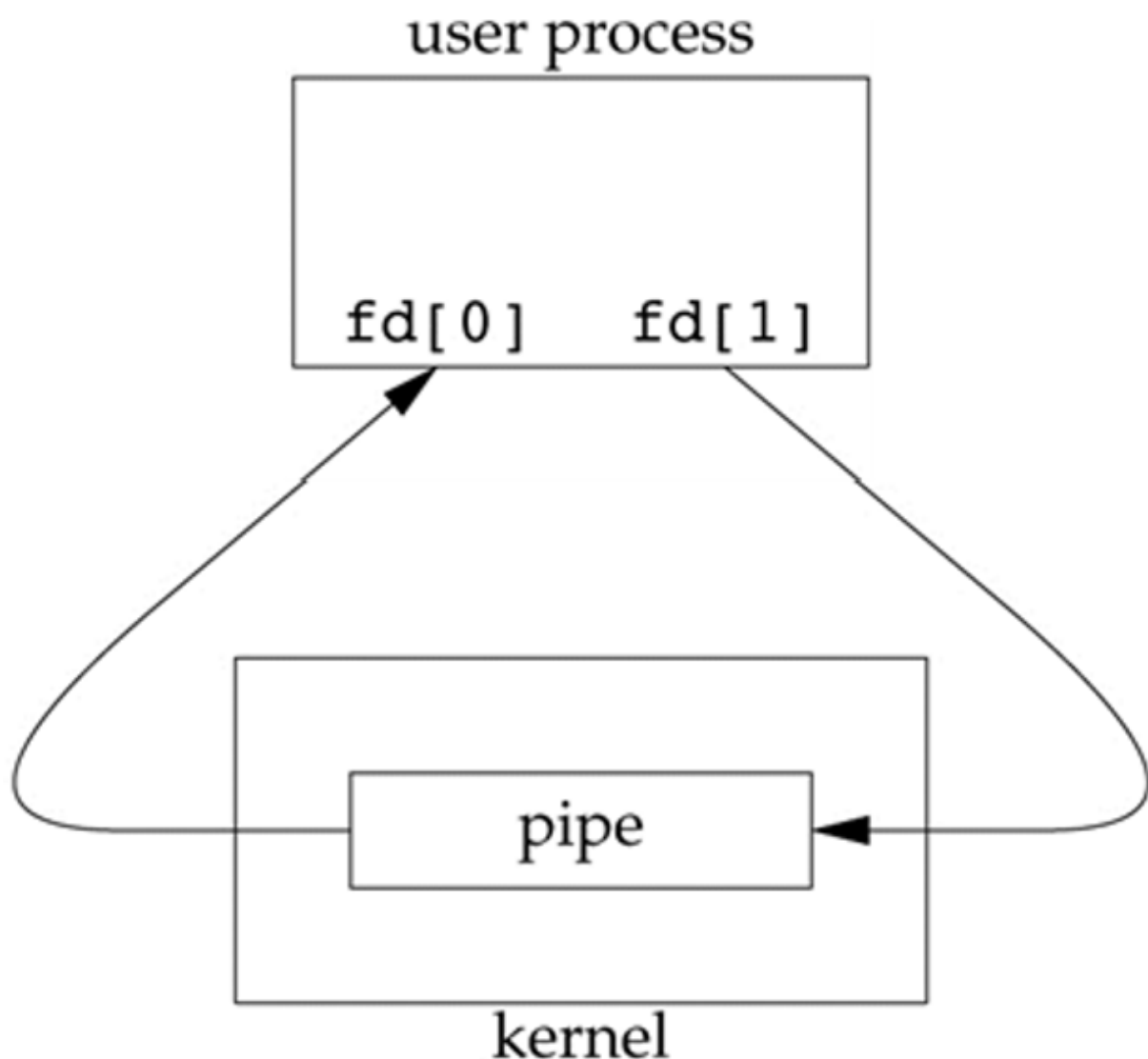
Em sistemas Unix, os pipes são criados pela System Call `pipe()`;

```
int pipe(int fd[2]);
```

A System Call `pipe()` cria um novo pipe e inicia `fd[2]` com os seguintes descritores:

- `fd[0]` é o read-end do pipe.
- `fd[1]` é o write-end do pipe.

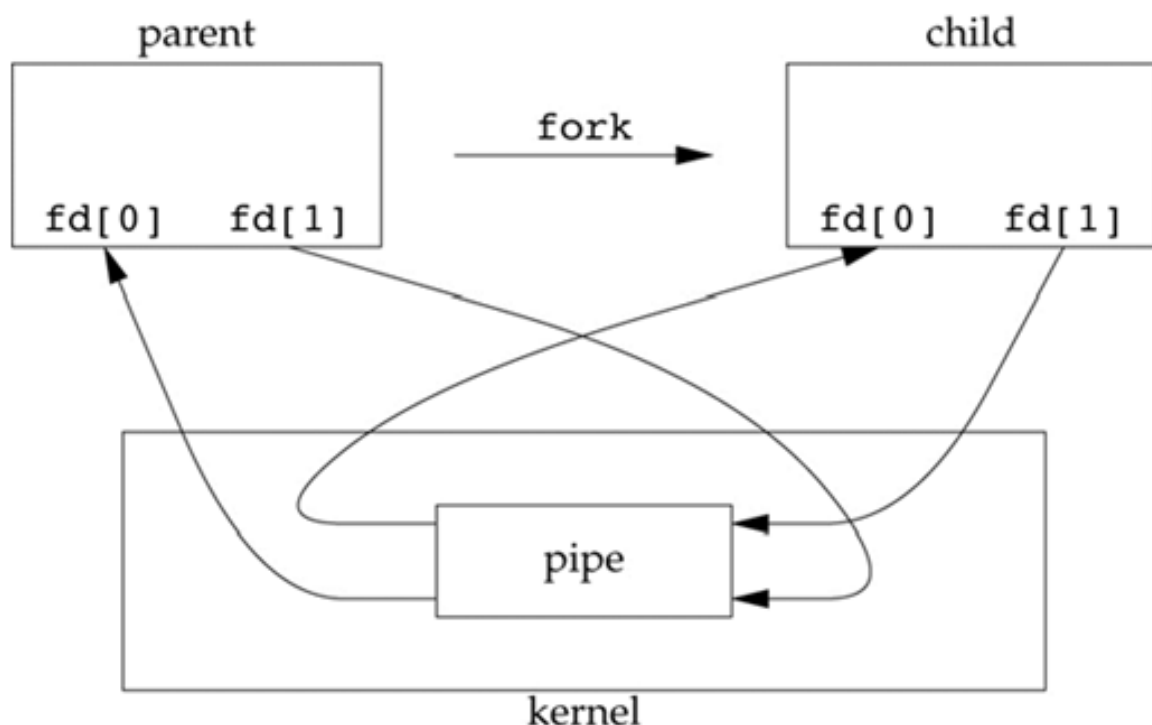
Em Unix os pipes são tratados como um tipo de ficheiro especial, o que permite o acesso aos mesmos usando as comuns System Calls `read()` e `write()`;



Forking de um Pipe

Um pipe não pode ser acessado fora do processo que o criou.

- Tipicamente, um processo pai cria um pipe e usa-o para comunicar com o processo filho que é criado via `fork()`;
- Como o pipe é considerado um tipo de ficheiro especial, o processo filho herda o pipe do processo pai.

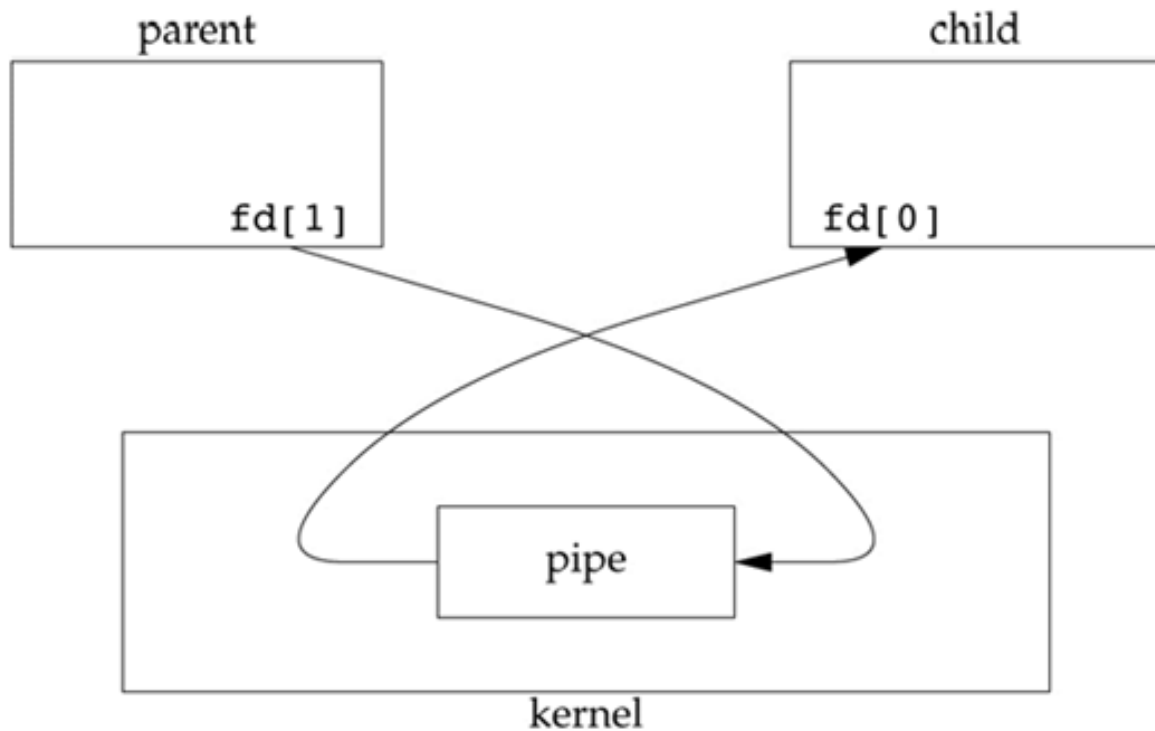


Pipe de Pai para Filho

Depois de um fork, podemos decidir a direção do fluxo dos dados dentro do pipe.

- Num pipe de Pai-Filho, o progenitor fecha a extremidade de leitura (read-end) do pipe `fd[0]` e o filho fecha a extremidade de escrita (write-end) `fd[1]`.
- A leitura a partir de um pipe aberto (i.e pelo menos um dos processos tem `fd[1]` aberto) bloqueia-o enquanto este se encontra vazio.





Comunicação por Pipes - UNIX

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void){
    char write_msg[BUFFER_SIZE] = "Folks";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    /* Criação do pipe */
    if(pipe(fd) == -1){
        perror("Creating pipe");
        return 1;
    }

    /* Fork do processo filho */
    pid = fork();

    if (pid < 0){ /* Ocorreu um erro */
        perror("Creating child");
        return 1;
    }

    if (pid > 0) {
        /* Processo pai */
        /* Fecho do fim inutilizado do pipe */
    }
}
```

```

close(fd[READ_END]);

/* Escrita no pipe */
write(fd[WRITE_END], write_msg , strlen(write_msg)+1);

/* Fecho do final escrito do pipe */
close(fd[WRITE_END]);
}
else{
/* Processo filho */
/* Fecho do fim inutilizado do pipe */
close(fd[WRITE_END]);

/* Leitura do pipe */
read(fd[READ_END], read_msg, BUFFER_SIZE);

/* Fecho do final escrito do pipe */
close(fd[READ_END]);
}

return 0;
}

```

Escalonamento de Processos

Motivação

O objetivo de multiprogramming é ter processos a correr o tempo todo, o que maximiza a utilização do CPU.

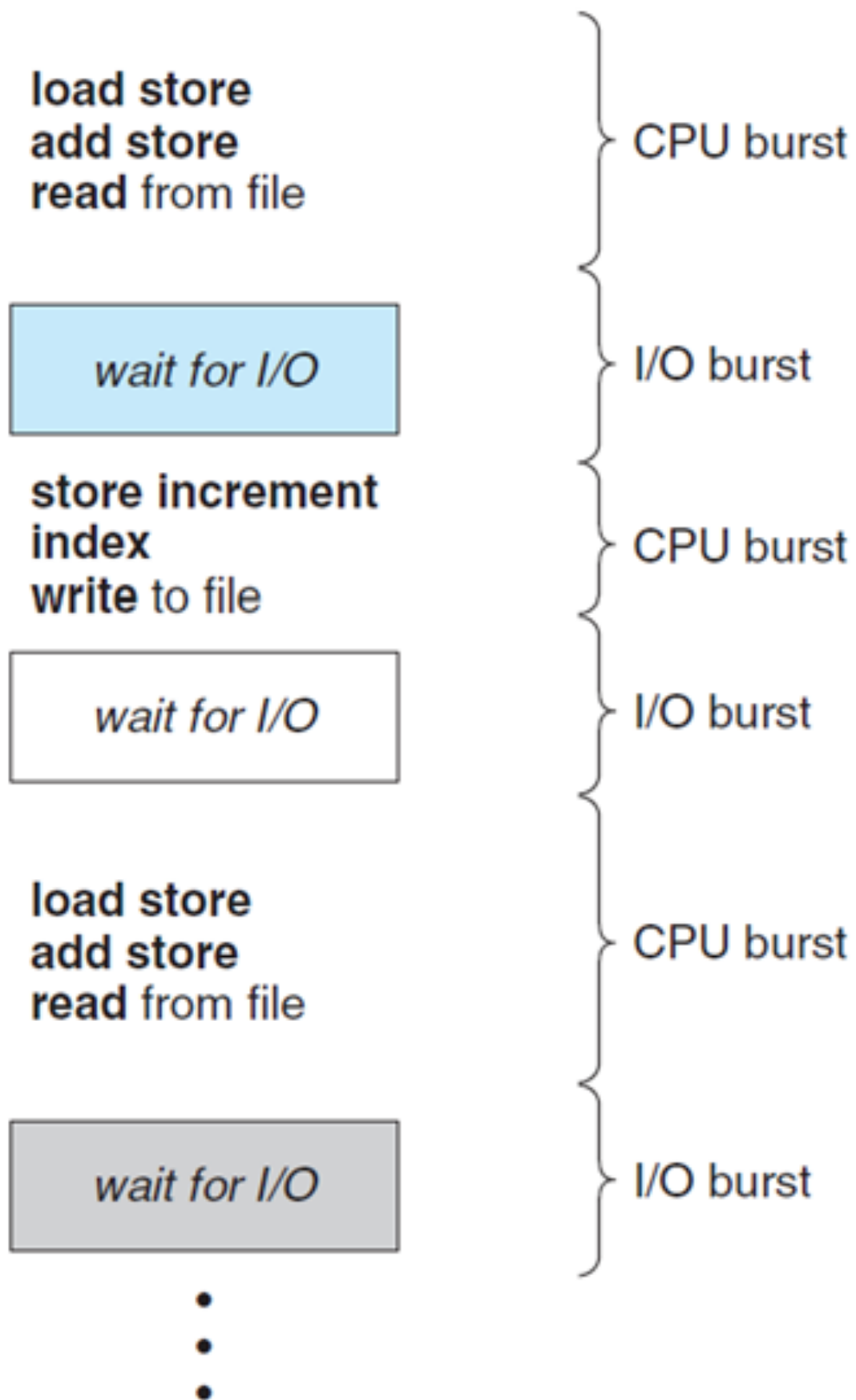
- Quando um processo tem de esperar, o sistema operativo retira esse processo do CPU e dá-lhe outro processo.

Uma função fundamental de um sistema operativo, sendo também a função básica do multiprogramming, é o Escalonamento de Processos.

- Ao escalonar eficientemente o CPU entre vários processos, o sistema operativo pode cobrir mais tarefas e tornar o sistema mais produtivo.

CPU-I/O Burst Cycle

A execução de um processo pode parecer um ciclo de execução do CPU e tempos de espera e I/O.



Decisões de Escalonamento:

Decisões de Escalonamento pode ocorrer quando um processo:

1. Troca do estado de running para o estado de waiting (como resultado de um I/O request);
2. Troca do estado de waiting para o estado de ready (como resultado de um I/O completion);
3. Troca do estado de running para o estado de ready (como o resultado de um interrupt);
4. Simplesmente termina.

O escalonamento escolhe entre todos os processos na Lista de Ready e aloca o CPU para um deles.

- Quando uma decisão de escalonamento ocorre tendo em conta o ponto 1 e 4 dizemos que o escalonamento é nonpreemptive (ou cooperativo);
- De outra forma, o escalonamento é preemptive.

Escalonamento do tipo preemptive requer hardware especial como, por exemplo, um timer.

Escalonamento Preemptive

O escalonamento preemptive pode resultar em condições de corrida (isto é, o output depende da sequência de execução de outros eventos incontrolláveis).

- Enquanto que um processo está a atualizar dados, o mesmo é temporariamente interrompido para que um segundo processo possa ocorrer. O segundo processo pode tentar ler os mesmos dados, que podem estar num estado inconsistente.
- O processo de uma system call pode envolver a mudança de alguns dados importantes ao nível do kernel. Se o processo for temporariamente interrompido (preempted) no meio destas mudanças e o kernel precisar de ler ou modificar a mesma estrutura, dá-se um CAOS.

Uma vez que interrupções podem ocorrer a qualquer momento, essas secções de código devem ser protegidas de acessos concorrentes por vários processos para que essas interrupções sejam desativadas ao entrar nessas secções e apenas reativadas à saída.

Critérios de Escalonamento

Foram sugeridos vários critérios para comparar algoritmos de escalonamento entre os quais se destacam:

- **Utilização do CPU** -> Manter o CPU o mais ocupado possível;
- **Throughput** -> Número de processos que completa a execução por unidade de tempo;
- **TurnAround/Completion Time** -> Quantidade de tempo necessário para executar um processo (intervalo de tempo desde a submissão e o tempo de conclusão);
- **Tempo de espera** -> Quantidade de tempo que o processo tem de esperar na queue de Ready;
- **Tempo de resposta** -> Quantidade de tempo que demora desde de que o pedido é submetido até à primeira resposta (no output) ser produzida.

Critérios de otimização:

- Maximizar a utilização do CPU e do Throughput;
- Minimizar o tempo de execução, o tempo de espera e o tempo de resposta.

Algoritmo First-Come First-Served (FCFS)

O processo que pede o CPU primeiro é o primeiro a ter o CPU alocado.

- Fácil de gerir com uma FIFO queue.
- Quando um processo entra na queue de Ready é ligado ao fim da queue.

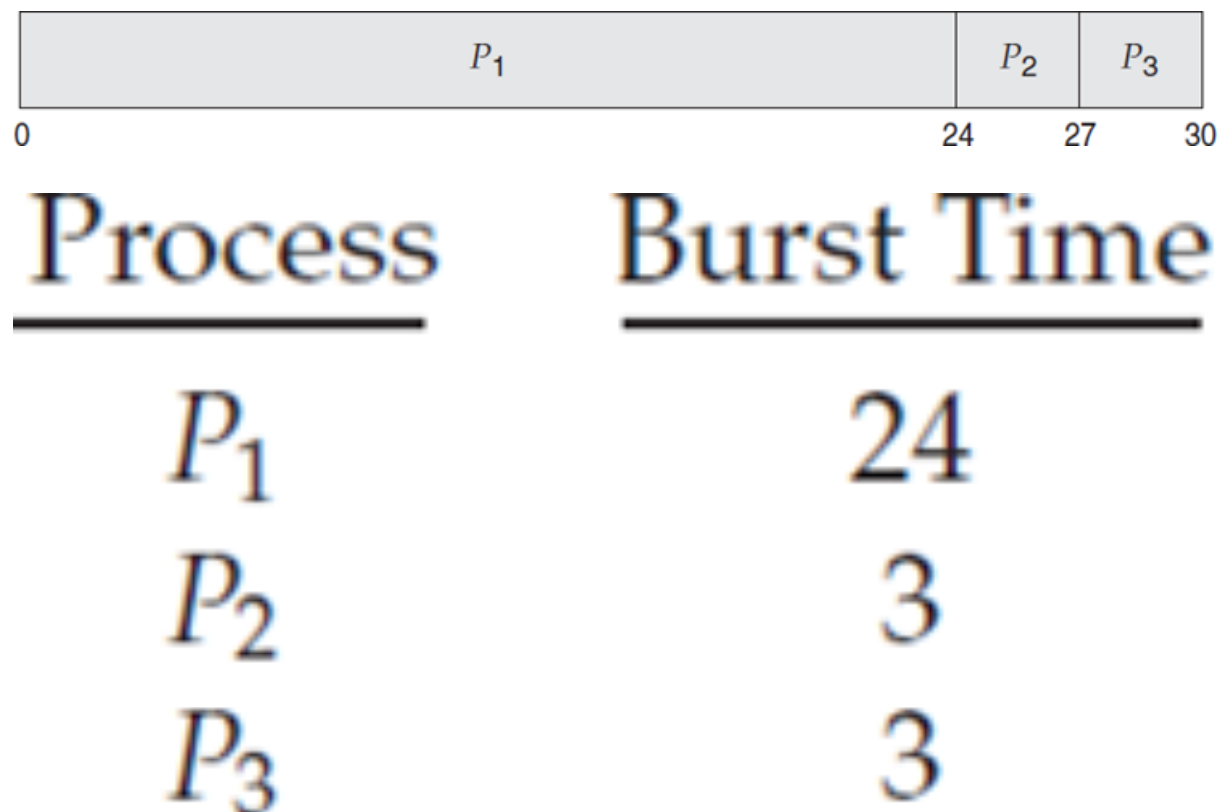
- Quando o CPU estiver livre, é alocado para o primeiro processo da queue de Ready (o processo é removido da queue).

→ cooperativo

FCFS é nonpreemptive, uma vez que o CPU foi alocado a um processo, esse processo controla o CPU até que o mesmo termine ou necessite de I/O.

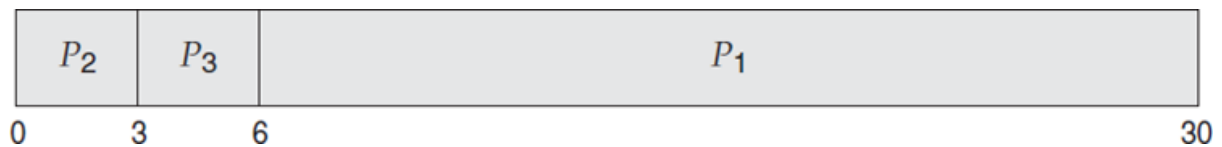
- A média de **turnaround** e tempo de espera é muitas vezes bastante longa.
- Problemático para os sistemas de partilha de tempo, onde é importante que cada utilizador obtenha uma parte do CPU em intervalos regulares (seria um inferno permitir que um processo mantivesse o CPU por um período prolongado).

Exemplo:



FCFS -> Média do tempo de espera.

- $(0 + 24 + 27) / 3 = 17$



FCFS -> Média do tempo de espera

- $(0 + 3 + 6) / 3 = 3$

Algoritmo Round Robin (RR)

Parecido com o FCFS mas com preemption, sendo desenhado para sistemas com partilha de tempo:

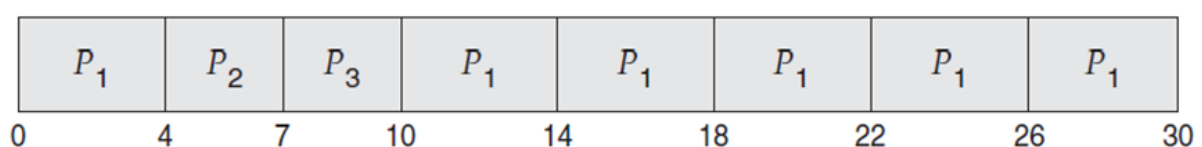
- Cada processo ganha um quantum ou uma "fatia" de tempo (pequena unidade de tempo do CPU);
- O temporizador (timer) interrompe cada quantum para agendar o próximo processo, o processo atual é antecipado e adicionado ao final da ready queue (sistema circular);

Se o tempo quantum for Q e houverem N processos na queue de Ready, então cada processo recebe $1/N$ do tempo do CPU em pedaços de no máximo Q (nenhum processo espera mais do que $(N-1) * Q$ unidades de tempo).

- Se o Q é grande -> FCFS;
- Se o Q for pequeno -> O overhead de mudanças de contexto pode ser muito alto degradando os níveis de utilização do CPU.

Round Robin (RR) com $Q = 4$;

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3



RR -> Tempo Médio de Espera

- $(\underset{P_1}{6} + \underset{P_2}{4} + \underset{P_3}{7}) / 3 = 17/3 = 5.66$

Shortest-Job-First (SJF)

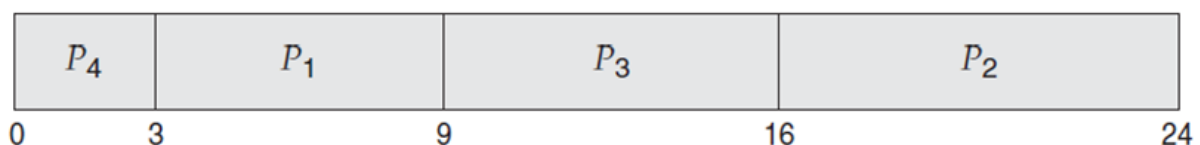
Associa cada processo com o tamanho do seu próximo CPU burst e usa estes tamanhos para agendar o processo com o burst mais curto de CPU.

- Se o próximo CPU Burst de dois processos for igual, usa-se o FCFS para quebrar o empate.

SJF é ideal porque dá sempre o tempo mínimo de espera médio para um determinado conjunto de processos

- Mover o processo mais curto para antes de um longo diminui o tempo de espera do processo mais curto no entanto aumenta o tempo de espera do processo mais longo.
- A dificuldade é saber o tamanho do próximo CPU burst.

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3



SJF -> Tempo médio de espera

- $(0 + 3 + 9 + 16) / 4 = 28 / 4 = 7$

FCFS -> Tempo médio de espera

- $(0 + 6 + 14 + 21) / 4 = 41/4 = 10.25$

Escalonamento Prioritário

É associado um número prioritário a cada processo e o CPU é alocado para o processo com a maior prioridade.

- Processos de igual prioridade são escalonados pela ordem de FCFS.
- SJF pode ser visto como um algoritmo de prioridade.

Escalonamento prioritário pode ser:

- Preemptive, antecipa o CPU se a prioridade de um processo recém chegado é maior que a prioridade do processo atualmente a correr.
- Non-Preemptive, permite ao processo atual acabar o seu CPU burst.

Principal Problema -> Blocking indefinido ou Starvation.

- Processos de baixa prioridade podem nunca ser executados e esperar infinitamente.
- A solução mais comum é o envelhecimento (aging), aumentando a prioridade dos processos pouco a pouco até que acabam por ser executados e terminar.

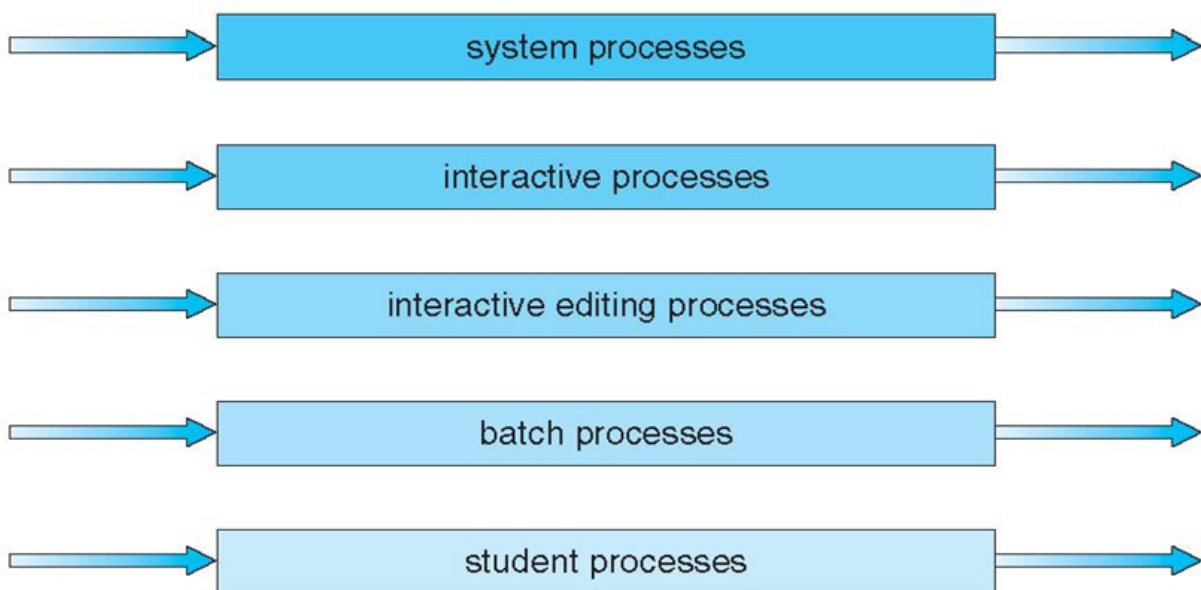
	PROS	CONS
FCFS	<ul style="list-style-type: none">• Simple	<ul style="list-style-type: none">• Short jobs get stuck behind long ones
RR	<ul style="list-style-type: none">• Better for short jobs	<ul style="list-style-type: none">• Context switching time adds up for long jobs
SJF	<ul style="list-style-type: none">• Optimal average waiting time• Big effect on short jobs	<ul style="list-style-type: none">• Hard to predict future• Starvation

Multilevel Queue (MLQ)

Divide a read queue em partições gerando várias queues.

- Os processos são permanentemente atribuídos a uma queue, geralmente baseado em alguma propriedade do processo, como tamanho em memória ou prioridade do processo.
- Cada queue tem o seu próprio algoritmo de escalonamento.
- Cada queue tem prioridade absoluta sobre a queue abaixo dela.
- Cada queue tem um certa quantidade de tempo de CPU que depois é dividido entre os processos

highest priority



lowest priority

Sincronização de Processos

Os processos cooperativos são uns dos quais podem afetar ou podem ser afetados por outros processos a correr ao mesmo tempo no sistema.

- Estes processos podem partilhar diretamente um endereço lógico ou serem autorizados a partilhar dados através de ficheiros ou mensagens.

O acesso concorrente a dados partilhados pode resultar em inconsistência de dados.

- Um processo pode ser interrompido em qualquer ponto, ficando parcialmente completo.
- A manutenção da consistência dos dados requer mecanismos para assegurar a execução ordenada dos processos de cooperação.

Regiões Críticas e Race Conditions

Uma região crítica é um pedaço do código que acede a um recurso partilhado.

- Alterar códigos de variáveis comuns, atualizar uma tabela, escrever um ficheiro.

Quando um processo está a executar numa região crítica, nenhum outro processo pode estar a executar nessa mesma região, isto é, dois processos não podem estar a executar na mesma região crítica ao mesmo tempo.

- Isto requer que os processos sejam sincronizados de alguma maneira.

Uma *race condition* ocorre quando vários processos têm permissão para aceder e manipular um recurso partilhado e o resultado depende da ordem em que o acesso ocorre, o que, na maior parte das vezes, leva a um resultado surpreendente e indesejável.

Problema da Secção Crítica

O objetivo do problema da secção crítica é desenhar um protocolo que os processos possam usar para cooperar:

- Os processos devem pedir permissão para entrar na secção crítica, a *entry section*.
- A região crítica deve ser seguida pela *exit section* ou por uma *remainder section* (secção não crítica).

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Uma solução para este problema tem de satisfazer 3 requisitos:

- **Exclusão mútua** - Se um processo estiver a ser executado numa secção crítica, então nenhum outro processo pode ser executado na mesma secção.
- **Progresso** - Se nenhum processo for executado numa secção crítica, então apenas os processos que não estão a executar numa secção restante podem participar na decisão de qual entrará na secção crítica em seguida, e esta decisão não pode ser adiada infinitamente.
- **Espera limitada** - Deve existir um limite no número de vezes que outros processos são autorizados a entrar nas suas secções críticas depois de um processo ter feito um pedido para introduzir a sua secção crítica e antes que esse pedido seja concedido.

No Capítulo Semáforos podemos ver uma técnica usada para resolver este problema.

Semáforos

Conceito:

Um semáforo é uma ferramenta de sincronização usada para controlar o acesso a um dado recurso constituído por um número finito de instâncias.

Semáforos são como inteiros, só que:

- Não é possível ler ou escrever valores, exceto para defini-lo inicialmente;
- Não utiliza valores negativos (quando um semáforo atinge o 0 todas as instâncias estão a ser usadas, significando que todos os processos que desejam usar o recurso ficam bloqueados até que o valor apresentado no semáforo seja maior que 0).

Existem 2 tipos de semáforos:

- **Counting semaphore** -> Pode variar sobre um domínio sem restrições.
- **Binary semaphore** -> Que só pode variar entre 0 e 1.

Operações com Semáforos:

Semáforos são acedidos através de duas operações atômicas padrão:

- **Wait()** -> Espera que o semáforo se torne positivo e depois decrementa-o por 1.
- **Signal()** -> Operação que incrementa o semáforo por 1.

```
wait(semaphore S){
    while ( S == 0); //busy waiting
    S--;
}

signal(semaphore S){
    S++;
}
```

Implementação de Semáforos

A implementação deve garantir que não são executados ao mesmo tempo 2 `wait()` e/ou `signal()`.

- Operações de `wait()` simultâneas não podem descer para valores abaixo de zero.
- É possível perder um incremento do `signal()` se a operação de espera acontecer simultaneamente.

Novamente, existem duas maneiras diferentes para minimizar o tempo de espera.

- Em uniprocessadores através de disabling interrupts.
- Em multiprocessadores através de disabling interrupts e instruções atômicas.

Implementação de um Semáforo em um uniprocessador

```
typedef struct {
    int value; // valor do semáforo
    PCB *queue; // queue associada de processos em espera
} semaphore;

init_semaphore(semaphore S){
    S.value = 0;
    S.queue = EMPTY;
}

wait(semaphore S){
    // desabilita as interrupções
    if(S.value == 0){
        // evita esperas ativas
        add_to_queue(current PCB, S.queue);
        suspend();
        // o kernel reabilita as interrupções antes de recomeçar aqui
    }
    else {
        S.value--;
        // habilita as interrupções
    }
}

signal(semaphore S){
    // desabilita as interrupções
    if(S.queue != Empty){
        // mantém o valor do semáforo e "acorda" um processo em espera
        PCB = remove_from_queue(S.queue);
        add_to_queue(PCB, ready queue);
    }
}
```

```

else {
    S.value++;
}
// habilita as interrupções
}

```

Implementação de um Semáforo em um multiprocessador

```

typedef struct{
    boolean guard; // para garantir a atomicidade
    int value;      // valor do semáforo
    PCB *queue;     // queue associada de processos em espera
} semaphore;

init_semaphore(semaphore S){
    S.guard = false;
    S.value = 0;
    S.queue = Empty;
}

wait(semaphore S){
    // desabilita as interrupções
    while (test_and_set(&S.guard)); // tempo de espera ativa curto
    if (S.value == 0){
        add_to_queue(current PCB, S.queue);
        S.guard = false;
        suspend();
        // o kernel reabilita as interrupções antes de recomeçar aqui
    }
    else {
        S.value--;
        S.guard = false;
        // habilita as interrupções
    }
}

signal(semaphore S){
    //disable interrupts;
    while(test_and_set(&S.guard)); // tempo de espera ativa curto
    if(S.queue != Empty){
        // mantém o valor do semáforo e "acorda" um processo em espera
        PCB = remove_from_queue(S.queue);
        add_to_queue(PCB, read queue);
    }
    else{
        S.value ++;
    }
    S.guard = false;
    // habilita as interrupções
}

```

Starvation e Deadlock

Conceitos:

Starvation (indefinite blocking) corresponde à situação em que um processo aguarda indefinidamente por um evento que pode nunca vir a ocorrer.

Deadlock é uma situação em que um conjunto de processos está à espera indefinidamente de um evento que nunca irá ocorrer porque esse evento só pode ser causado por um dos processos em espera nesse conjunto.

Quando ocorrer um deadlock, uma starvation não acontece e vice-versa.

- Starvation pode acabar (apesar de não ter de o fazer);
- Deadlock não termina sem intervenção externa.

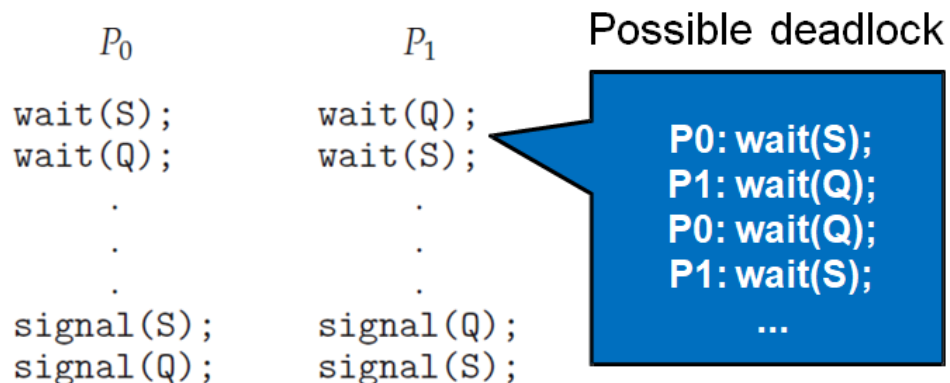
Deadlock

Deadlocks ocorrem quando estamos a aceder a múltiplos recursos.

- Não se pode resolver um deadlock por cada recurso de forma independente.

Deadlocks não são determinísticos e nem sempre acontecem no mesmo pedaço de código

- Tem de ser exatamente no momento certo (ou momento errado?)



O que se segue não foi falado nas aulas teóricas, mas pode ser complementar e necessário.

Problemas Clássicos de Sincronização

- Bounded Buffer Problem
- Readers and writers Problem
- Dining philosophers Problem

Caracterização do Deadlock

Um deadlock só pode surgir se as próximas 4 condições se mantiverem simultaneamente:

1. Mutual exclusion -> Apenas um processo de cada vez pode utilizar um recurso (um processo de pedido deve ser adiado até que o recurso seja libertado);
2. Hold and wait -> Um processo que detém (pelo menos) um recurso está à espera para adquirir recursos adicionais detidos por outros processos;
3. No preemption -> Um recurso não pode ser antecipado, ou seja, um recurso só pode ser libertado voluntariamente pelo processo que o mantém, depois desse processo ter concluído a sua tarefa;
4. Circular wait -> Existe um conjunto de processos de espera $\{P1, P2, \dots, PN\}$ de tal forma que P1 está à espera de um recurso detido pelo P2, que por sua vez está à espera de um recurso detido por P3 e assim sucessivamente até chegar a PN que está retido porque precisa de um recurso de P1.

Lidar com deadlocks

Para garantir que deadlocks nunca ocorem, o sistema pode usar um protocolo para prevenir ou para evitar deadlocks.

- Prevenção de deadlocks -> Métodos que garantem que pelo menos uma das quatro condições necessárias não ocorre, limitando a

forma como os pedidos de recursos podem ser feitos.

- **Evitar deadlocks** -> Métodos que exigem que o sistema operativo seja previamente informado, sobre quais os recursos que um processo irá utilizar durante a sua vida útil, de modo a decidir se o pedido de recursos consegue ser satisfeito ou se deve ser adiado.

Se o sistema não implementar nenhum dos protocolos anteriores então é provável que ocorra o aparecimento de um deadlock:

- **Deteção de deadlocks** -> Método que examina o estado do sistema para determinar se ocorreu algum deadlock e fornece algoritmos para recuperar dos mesmos.

No caso de ausência de métodos para prevenir, evitar ou recuperar de deadlocks chegamos a uma situação em que o sistema se encontra num estado bloqueado e não tem forma nenhuma de reconhecer o que aconteceu.

- **Deadlocks não detetáveis** -> podem causar a deteriorização da performance do sistema, uma vez que os recursos estão a ser retidos por processos que não podem correr e devido à acumulação de cada vez mais e mais processos que necessitam dos recursos que se encontram bloqueados acabando esses por entrar também em deadlock (eventualmente o sistema irá quebrar e será necessário forçá-lo a reiniciar manualmente).

Prevenção de Deadlocks

Prevenir mutual exclusion -> tentar evitar recursos não partilhados (cada recurso é tornado partilhado e assim um processo nunca precisa de esperar por qualquer recurso).

- **Problema**: Não é muito realista, uma vez que alguns recursos são obrigatoriamente não partilhados.

Prevenir hold and wait -> Garantir que sempre que um processo solicita recursos, não detém quaisquer outros.

- Exige que o processo solicite todos os recursos antes da execução começar (prever o futuro é difícil, tende-se a estimar recursos em demasia)

- Alternativamente, requer que processo liberte todos os recursos que possui antes de solicitar recursos adicionais.
- **Problema**: Pouca utilização de recursos e torna a starvation possível.

Prevenir no-preemption -> libertar recursos voluntariamente.

- Se o processo P falha ao alocar alguns recursos, então libertamos todos os recursos que P está a utilizar e apenas quando P ganhar novamente os antigos e os novos recursos é que podemos reiniciar o processo.
- Alternativamente, se o processo P falha a alocar alguns recursos, verificamos se esses recursos estão alocados a um outro processo Q que esteja à espera de recursos adicionais, antecipamos os recursos desejados de Q e alocámo-los para o processo P.

Problema: Geralmente não se pode aplicar a recursos como semáforos.

Prevenir circular wait -> Impor uma "encomenda" total de todos os tipos de recursos e exigir que cada processo solicite recursos numa ordem crescente de enumeração.

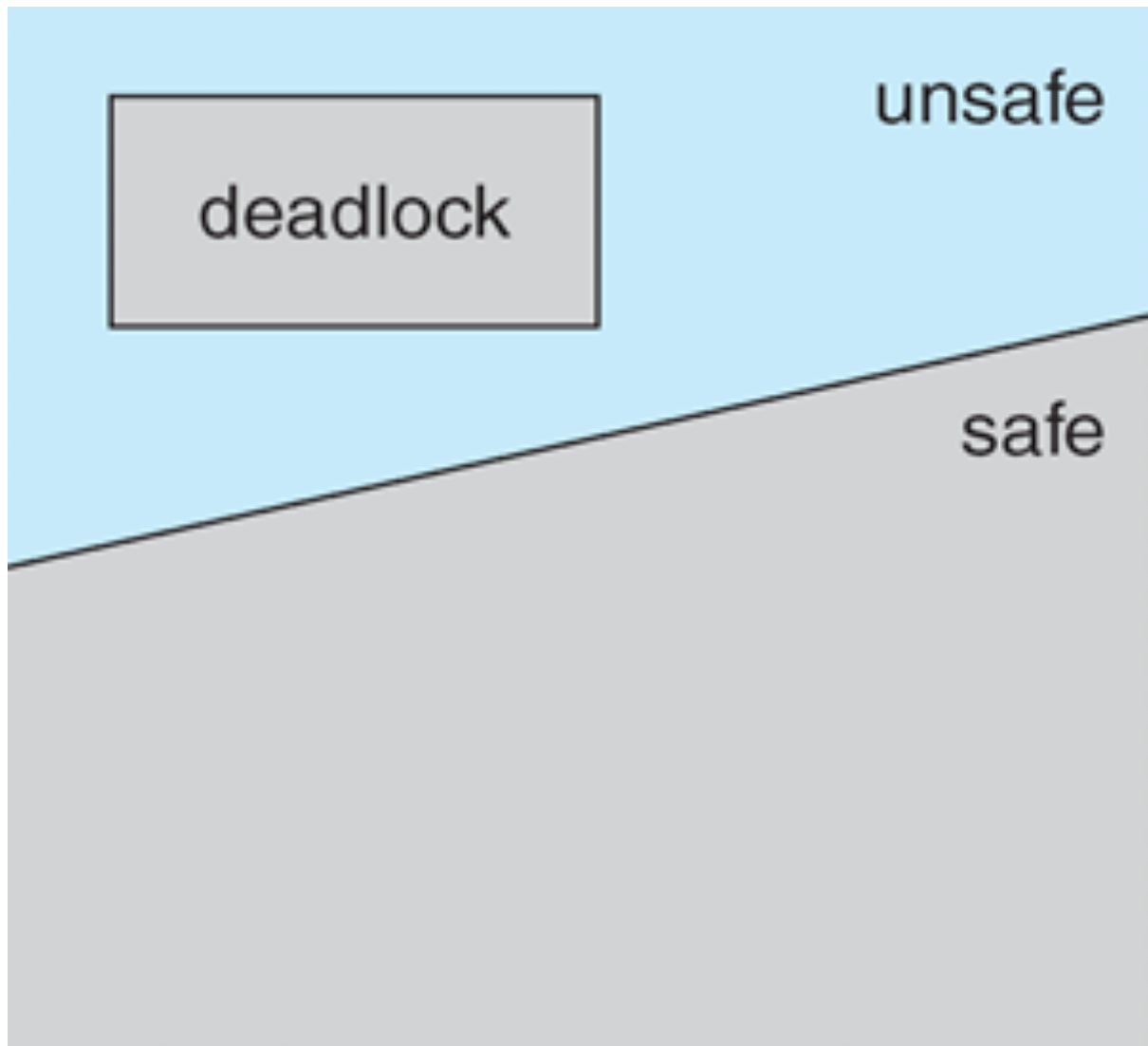
Problema: Programadores podem escrever programas sem seguir esse tipo de ordem.

Evitar Deadlocks

Os métodos para evitar deadlocks requerem informação adicional à priori, sobre quais os recursos que um processo utilizará durante a sua vida útil.

- Métodos mais simples e úteis requerem que cada processo declare o número máximo de recursos de cada tipo que possa vir a precisar.

Quando um processo solicita um recurso disponível, temos de decidir se a sua alocação imediata deixa o sistema num estado seguro.



Recuperar de um Deadlock

Para recuperar de um deadlock, podemos abortar processos utilizando dois métodos:

- Abortar todos os processos em deadlock -> O resultado de cálculos parciais feitos por tais processos são perdidos e provavelmente terão de ser refeitos.
- Abortar um processo de cada vez até que o ciclo de deadlock seja eliminado > Provoca sobrecargas consideráveis, uma vez que o algoritmo de deteção de deadlocks deve ser invocado após cada processo ser abortado.

Qual a ordem pela qual devemos decidir abortar?

1. Prioridade do processo;

2. Quanto tempo o processo demorou a correr e a concluir;
3. Recursos que o processo usou;
4. Recursos que o processo precisa para completar;
5. Quantos processos terão de ser terminados.

Em alternativa, para recuperar de um deadlock, podemos sucessivamente antecipar alguns recursos dos processos e dá-los a outros até que o ciclo do deadlock seja quebrado.

Três questões têm de ser abordadas

- **Seleção da "vítima"** -> tentar minimizar os custos tais como o número de recursos que um processo em deadlock possui e a quantidade de tempo que o processo tem consumido até agora.
- **Rollback** -> Devolver o processo a um estado de segurança e reiniciá-lo a partir desse estado.
- **Starvation** -> O mesmo processo pode ser sempre escolhido como vítima levando ao estado de starvation e, portanto, a solução mais comum é incluir o número de rollbacks no fator de custo.

Gestão de Memória

Background:

A memória (ou memória principal) consiste num grande array de bytes cada um com o seu próprio endereço.

- A unidade de memória vê apenas um fluxo de endereços de memória e não sabe como são gerados ou para que servem.

Memória e Registos são o único tipo de armazenamento a que o CPU pode aceder diretamente.

- Os programas devem ser trazidos do disco para a memória para ser corridos.
- O CPU, em seguida, recolhe as instruções da memória de acordo com o counter do programa.

- Estas instruções podem causar carregamento/armazenamento adicional de/para endereços de memória específicos.

Hardware Básico:

A maioria dos CPUs consegue decodificar as instruções e realizar operações simples em registos à taxa de uma ou mais operações por clock cycle.

A memória é acedida pelo memory bus o que pode levar alguns clock cycles do CPU.

A cache encontra-se entre a memória e os registos do CPU. Permite aumentar a velocidade de acesso à memória.

Espaço de Memória

Uniprogramming :

- As aplicações correm sempre no mesmo local na memória física;
- As aplicações podem aceder a qualquer endereço físico;
- Não é necessária a proteção da memória uma vez que apenas é executada um aplicação de cada vez.

Multiprogramming :

- As aplicações podem correr em diferentes localizações na memória física;
- As aplicações não podem aceder a todos os endereços físicos (memória normalmente dividida em duas zonas:
 - (i) sistema operativo, geralmente em memória baixa juntamente com o vetor de interrupt;
 - (ii) processos de utilizador, geralmente em alta memória);
- A proteção de memória é necessária para prevenir o overlap de endereços entre processos.

Proteção de Memória

A proteção de memória é necessária para garantir um funcionamento correto.

- Proteger os processos do sistema operativo do acesso por processos do utilizador;
- Proteger processos do utilizador uns dos outros.

A proteção de memória deve ser fornecida pelo hardware.

- Normalmente, o Sistema Operativo não intervém entre o CPU e os seus acessos à memória devido à consequente penalização de desempenho.

Precisamos de garantir que cada processo tem um espaço separado em memória.

- Separar o espaço de memória por processo protege os processos uns dos outros e é fundamental para ter múltiplos processos carregados em memória para execução simultânea.
- Para separar o espaço em memória, precisamos da capacidade de determinar o leque de endereços legais a que o processo pode aceder.

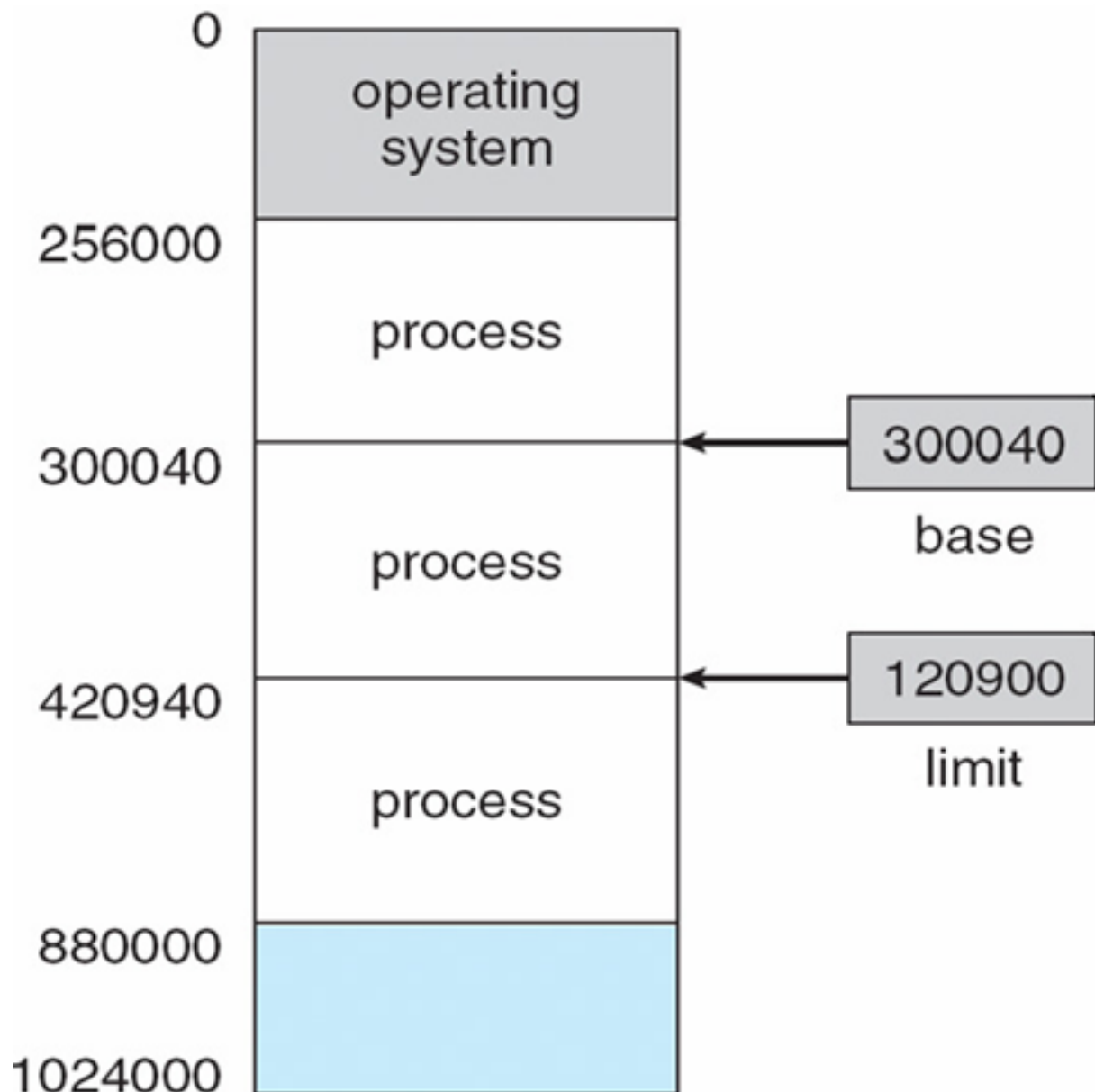
Registos de base e limites

Um par de registos define o espaço de endereço de um processo:

- O registo base contém o menor endereço legal da memória física.
- O registo limit especifica o tamanho.

Os utilizadores não têm permissão para alterar os registos base/limite.

Sem esta proteção, qualquer bug poderia levar ao crash de outros programas ou até mesmo do sistema operativo.



Espaço de endereçamento lógico vs Espaço de endereçamento físico

O conceito de um espaço de endereço lógico que está ligado a um espaço de endereço físico separado é fulcral para uma gestão adequada de memória.

- **Logical Address**: Endereço gerado pelo CPU;
- **Physical Address**: Endereço visto pela unidade de memória;

O conjunto de todos os endereços lógicos gerados por um programa é o espaço de endereço lógico. O conjunto de todos os

endereços físicos correspondentes a cada um dos endereços lógicos é o espaço de endereços físicos.

No tempo de compilação e de load, o "esquema" de ligação de endereços gera os mesmos endereços lógicos e físicos.

No tempo de execução, o "esquema" de ligação de endereços gera um endereço lógico e físico diferentes.

Unidade de Gestão de Memória (MMU - Memory Management Unit)

O dispositivo de hardware que no tempo de execução mapeia endereços lógicos para físicos é chamado de MMU.

- Existem diferentes métodos para completar tal mapeamento.

Swapping

O que acontece se nem todos os processos tiverem espaço na memória?

- Usa-se uma forma extrema de context switch onde alguns processos em memória são temporariamente trocados da memória para uma backing store.
- Torna possível que o espaço total de endereços físicos de todos os processos possa exceder a verdadeira memória física do sistema.

Backing store -> Geralmente um disco rápido e grande o suficiente para acomodar cópias de todas as imagens de memória para todos os utilizadores.

- A read queue mantém os processos ready-to-run dos quais têm imagens de memória na backing store.

Dispatcher -> Verifica se um processo está agendado em memória para executar.

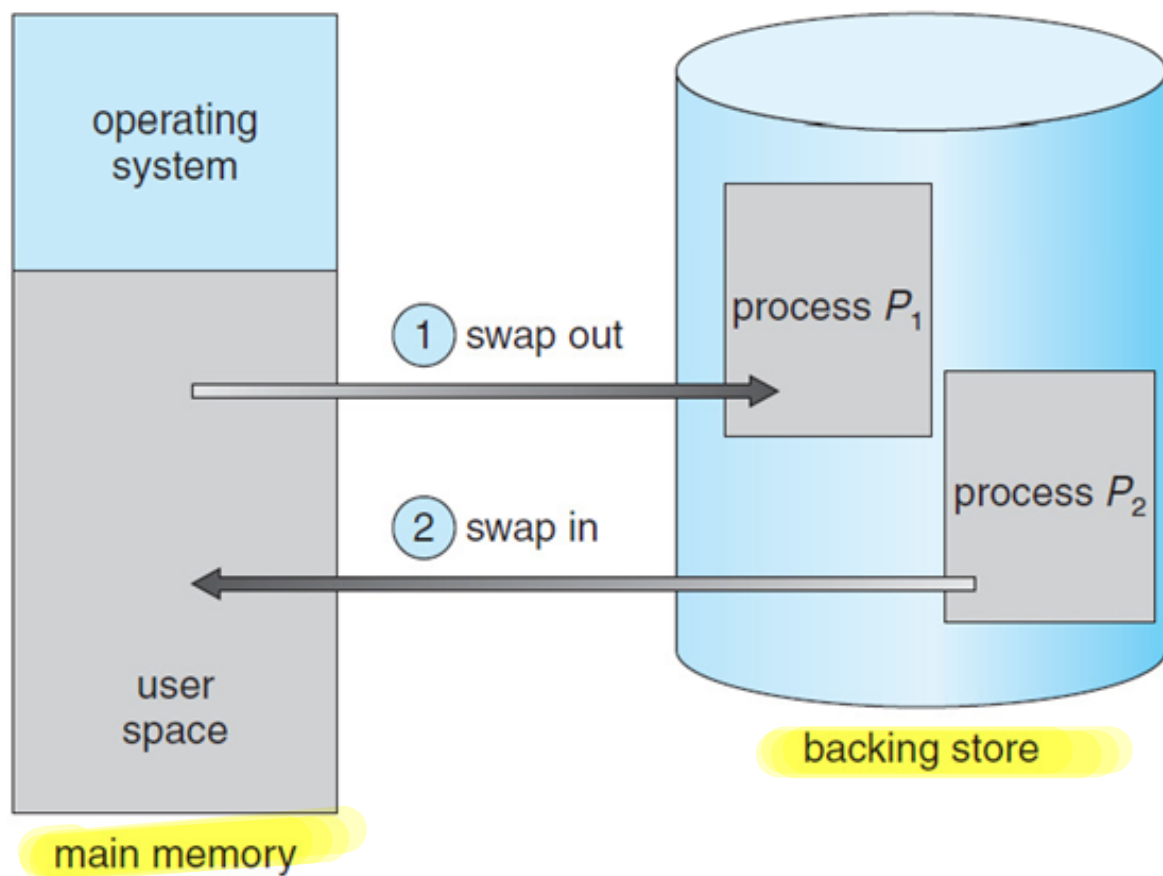
- Se não estiver e se não houver memória livre nessa região, o dispatcher troca o processo atual em memória pelo

processo desejado.

- O tempo do Context-switch num sistema de swapping é bastante elevado.

Maior parte do tempo de swap é na verdade tempo de transferência (temos de trocar em ambos out e in)

- O tempo total de transferência é diretamente proporcional à quantidade de memória trocada.



Um processo trocado para a backing store precisa de ser colocado no mesmo endereço físico de onde foi retirado?

- Não, caso tenha sido usada alocação dinâmica para mudar o relocation register.

Pode um processo que está à espera de uma operação de I/O ser swapped out?

- Pode, se usarmos double buffering e a operação de I/O for executada dentro de buffers do kernel (transferências entre buffers do kernel e buffers de memória de processo, só ocorrem quando o processo é trocado back in.)

Alocação contígua de memória

Alocação contígua de memória é um método inicial em que cada processo está contido numa única secção contígua de memória.

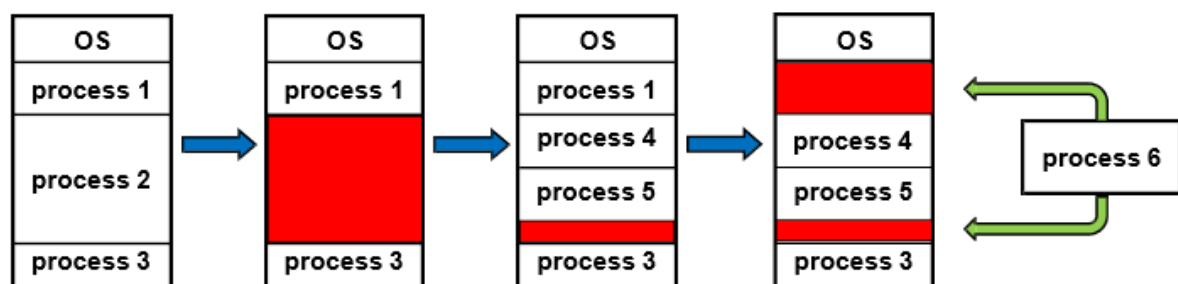
Um esquema simples é dividir a memória em várias partições de tamanho fixo.

- Cada partição contém exatamente 1 processo.
- O grau de multiprogramação está ligado pelo número de partições

A generalização é o esquema de partição variável

- Cada partição contém na mesma um único processo mas o tamanho da partição é exatamente aquele que o processo precisa.
- O Kernel mantém uma tabela que indica que partes da memória estão ocupadas e que partes estão disponíveis (memory holes).

Esquema de Partições de Variáveis



Fragmentação interna

Fragmentação interna existe quando a memória alocada a um processo é ligeiramente maior do que memória solicitada.

- A diferença de tamanhos está relacionada com memória que não é usada que é interna à partição.

É o resultado de tentar evitar o overhead quando se mantém o rasto de pequenos buracos na memória.

Exemplo:

- Consideremos que temos um buraco na memória para 1000 bytes e o processo que está a pedir essa parte da memória necessita apenas de 998 bytes, cria-se um buraco pequeno que apenas tem espaço para 2 bytes -> é melhor ignorar esse pequeno buraco do que lidar com ele.
- Um aproximação geral para evitar pequenos espaços na memória é separar a memória física em blocos de tamanho fixo e alocar unidades de memória baseado no tamanho de cada bloco.

Fragmentação Externa

Fragmentação externa acontece quando existe espaço suficiente na memória para satisfazer o pedido de alocação, no entanto, o espaço disponível não é contíguo.

- O armazenamento está fragmentado num número grande de buracos pequenos.
- No pior caso, podemos ter um espaço de dois em dois processos em toda a memória.

Uma possível solução seria "misturar" (shuffle) a memória para compactar toda a que está livre num único espaço.

- Só é possível com alocação dinâmica, feita em tempo de execução.
- Pode ser muito dispendioso.

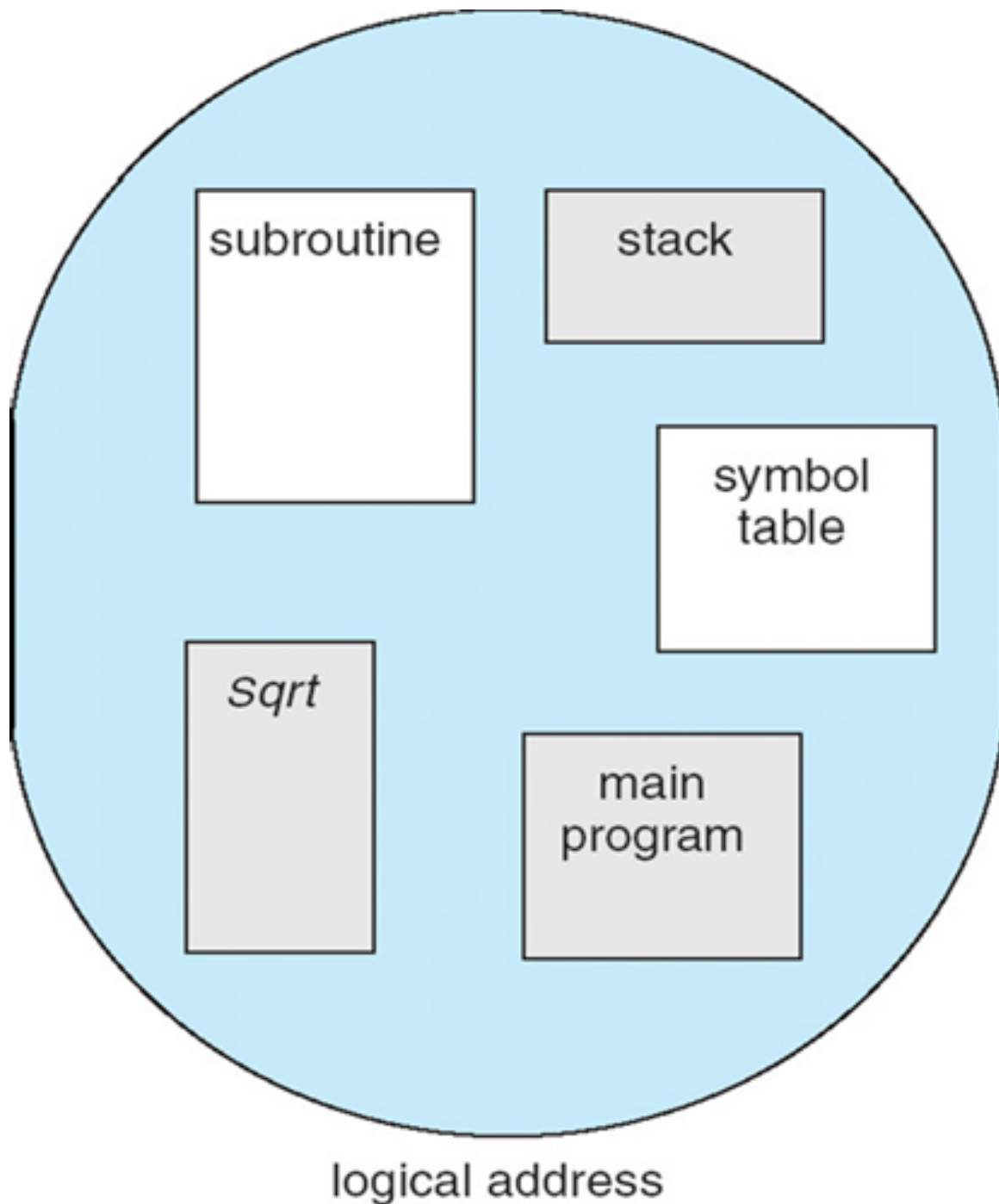
Outra solução possível seria usar esquemas de alocação de memória não contíguos.

- **Segmentation** e **Paging** são dois esquemas possíveis.

O que é que o programador vê da memória?

A memória é vista como uma coleção de unidades lógicas separadas e de tamanho variável sem nenhuma ordem necessária entre elas.

1. Programa main
2. Procedimentos/Funções
3. Objetos/Métodos
4. Variáveis locais e globais
5. Stack
6. Tabela de Símbolos
7. Arrays



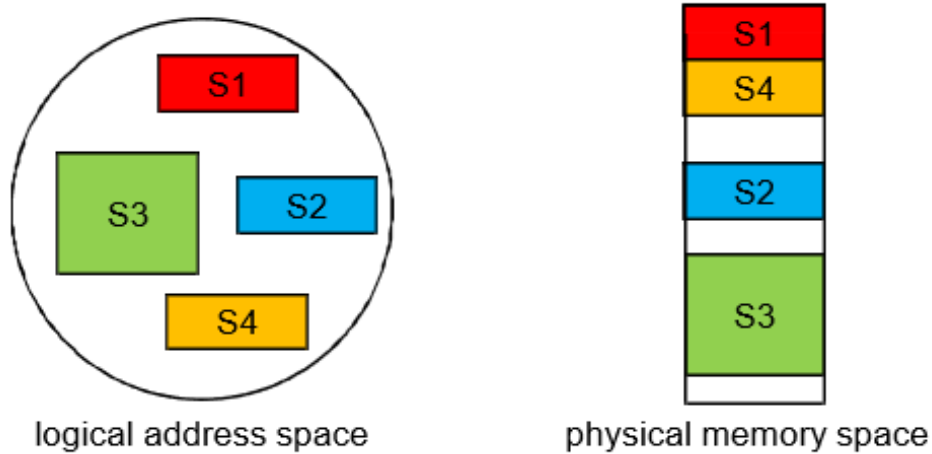
Segmentation e Paging

Segmentation:

Segmentação é um esquema de gestão de memória que satisfaz a visão da memória do programador.

- Um espaço de endereços lógicos é uma coleção de segmentos.

- A cada segmento é dada uma região contígua na memória (com o registo base e limit) que pode residir em qualquer local da memória física.



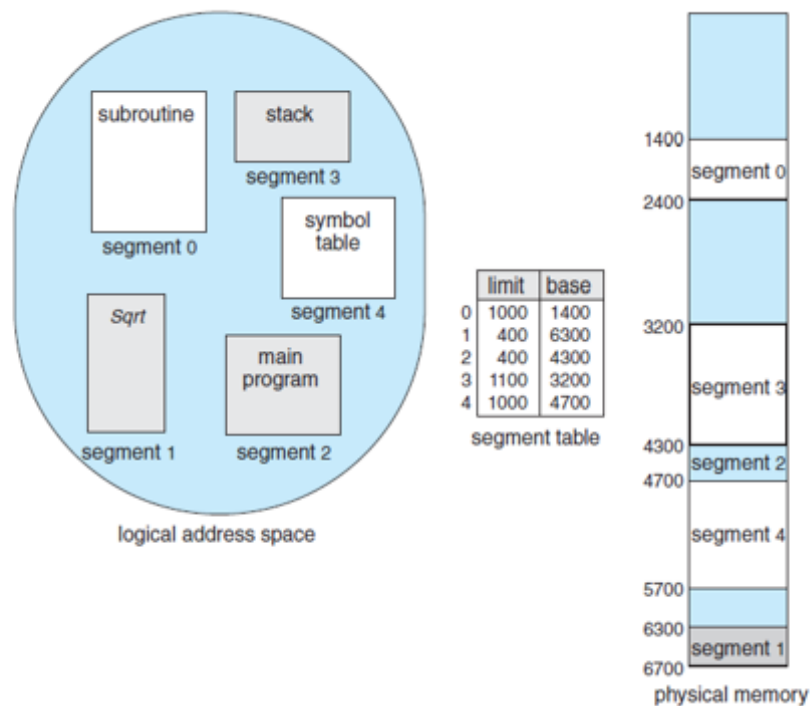
Cada segmento tem um identificador e um tamanho.

Endereços lógicos (dentro do segmento) existem na forma de tuplos:

```
<segment-number, offset-within-segment>
```

A tabela dos segmentos mapeia endereços lógicos de duas dimensões para endereços físicos de uma dimensão. Cada entrada inclui:

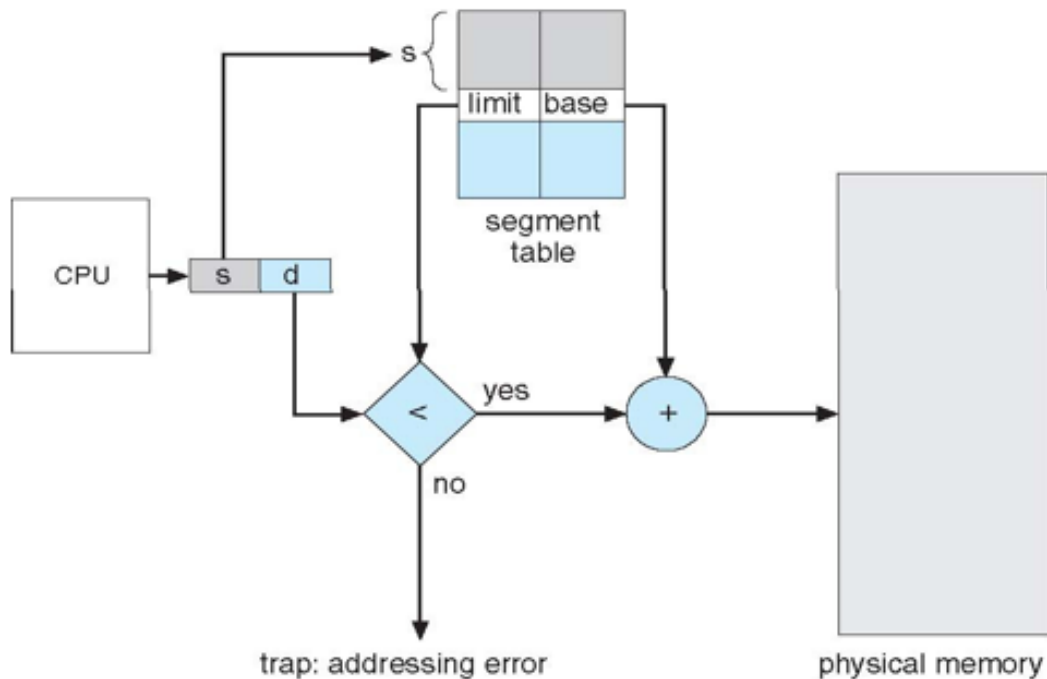
- **Segment base** : Endereço físico inicial onde o segmento reside na memória.
- **Segment limit** : Tamanho do segmento.



Arquitetura de Segmentation

A tabela de Segmentos encontra-se no PCB (Process Control Block). Esta tabela inclui também proteção de dados tais como:

- Validação de Bit (segmento ilegal/legal).
- Privilégios de Ler/escrever/executar/partilhar.



Gestão de Segmentos

Quando um novo processo é carregado em memória:

- Cria-se uma nova tabela de segmento e armazena-se no PCB de processos;
- Aloca-se espaço na memória física para todos os segmentos do processo.

Se não houver espaço em memória física:

- Compacta-se a memória (move-se segmentos, atualiza-se as bases) para tornar o espaço contíguo.
- Swap de um ou mais segmentos para fora do disco.

Para aumentar um segmento S:

- Se existe espaço livre em baixo do segmento, só é necessário atualizar o limit do segmento e usar esse espaço livre.
- Caso contrário, mover o segmento S para um espaço livre maior;
- Swap do segmento acima do S para o disco.

Advantages:	Disadvantages:
Os segmentos não têm de ser contíguos.	Sofre do problema de alocação dinâmica do armazenamento
Os segmentos podem ser swapped de forma independente	Sofre de fragmentação externa
Os segmentos podem ser partilhados entre diferentes processos	

Paging

Tal como a Segmentation, **Paging** é outro esquema de gestão de memória que permite que o espaço do endereço físico de um processo não seja contíguo.

Comparado ao Segmentation, o **Paging**:

- Torna a alocação e o swapping mais fácil (não existem segmentos de tamanho variável);
- Evita fragmentação externa;
- Não é necessária compactação.

Devido a estas vantagens, Paging é usado de várias formas na maioria dos Sistemas Operativos, desde mainframes até aos telemóveis.

A ideia deste esquema é organizar a memória em blocos de tamanho fixo.

- Dividir a memória lógica em blocos de tamanho fixo chamados **pages**.
- Dividir a memória física (e a backing store) em blocos do mesmo tamanho chamados **frames**.

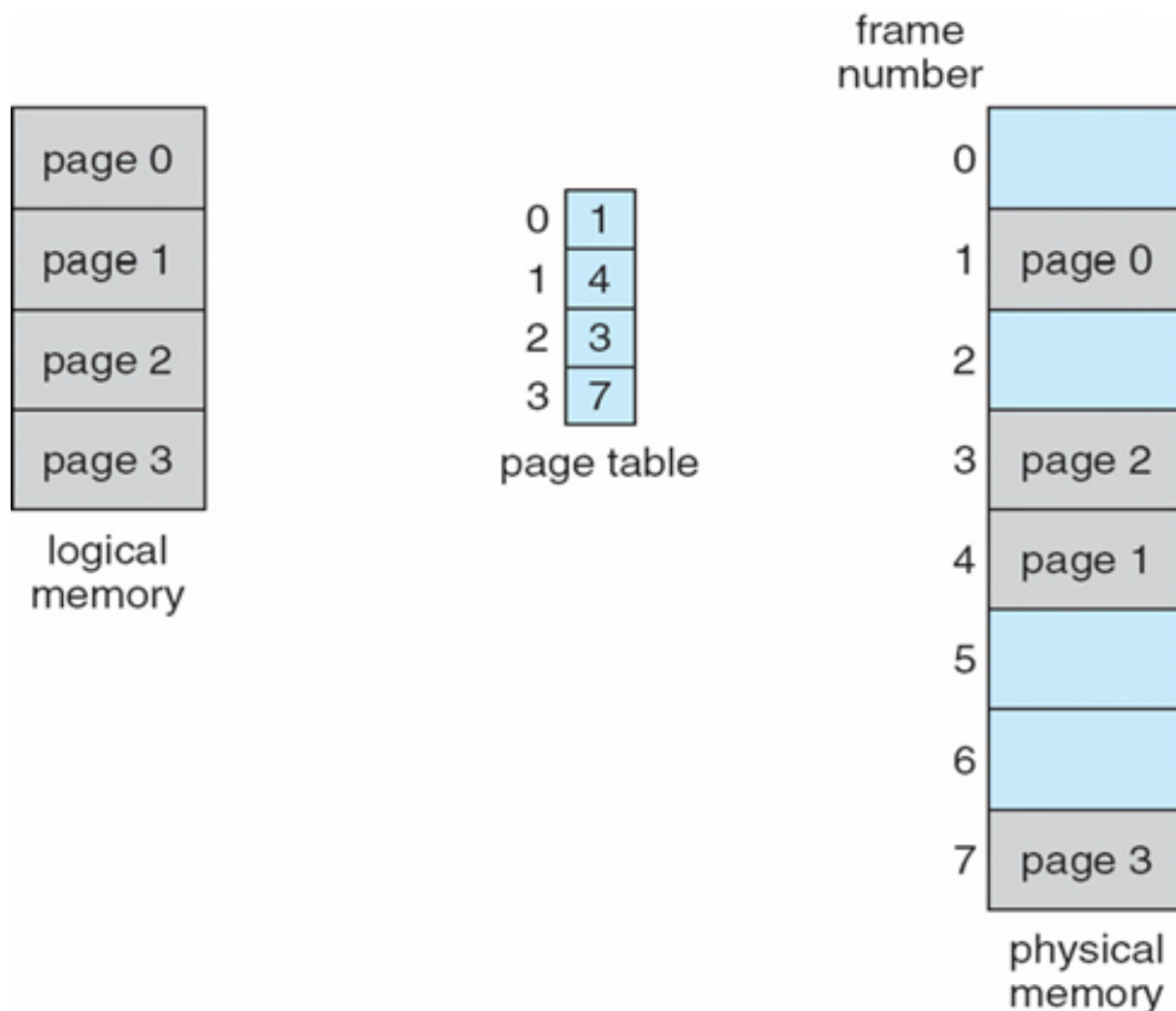
Para carregar um processo de N pages, é necessário encontrar N frames livres.

- As pages não têm de ser carregadas num conjunto contíguo de frames.
- É necessário acompanhar as frames livres.

Modelo de Memória de Paging

Uma tabela de page acompanha todas as pages de um processo em particular.

- Cada entrada contém o frame correspondente na memória física;
- Traduz endereços lógicos em endereços físicos.



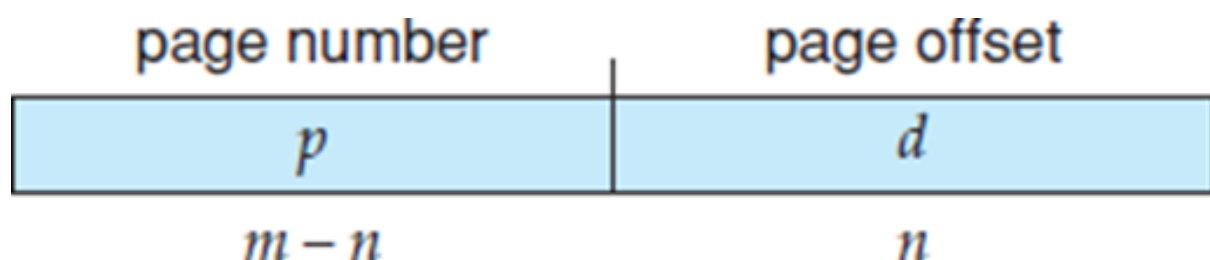
Esquema de tradução de endereços

Todos os endereços lógicos são divididos em 2 partes:

- **Page Number (p)** : índice na tabela de pages.
- **Page offset (d)** : deslocamento dentro da page.

O endereço base na tabela de pages combinado com o page offset definem o endereço físico de memória que é enviado para a unidade de memória.

Se o tamanho do espaço do endereço lógico é 2^m e o tamanho da page é 2^n bytes ($m > n$), então a high-order $m-n$ bits do endereço lógico designa o **page number (p)**, e a lower-order de n bits indica a **page offset (d)**.



Nota: O hardware necessário para suportar paging é relativamente menor que o necessário para a segmentação.

Implementação de Paging

O sistema operativo intervém 4 vezes na paginação:

1. Criação do Processo

- Determinar o tamanho do programa;
- Criação da tabela de páginas.

1. Execução do Processo

- Re-inicializar a MMU para o novo processo;
- Limpar a TLB (Translation lookaside buffer).

3. Na Page Fault

- Determinar o endereço virtual causador da page fault;
- Colocar a página em memória, se o endereço for legal.

1. Fim da execução do processo

- Libertar a tabela de páginas e as páginas associadas.

Paginação + Segmentação

É possível combinar estes dois métodos em dois níveis de mapeamento:

- Processos são vistos como um conjunto de segmentos lógicos de tamanho variável
- Cada segmento é depois dividido em pages de tamanho fixo.

Os endereços lógicos consistem mais uma vez em tuplos:

```
<segment-number, page_within_segment, offset_within_page>
```

Implementação

- Uma tabela de segmentos por processo e uma tabela de paginação por segmento.
- Evita fragmentação externa.

A partilha pode ser dada em ambos os níveis:

- Partilhar um segmento completo por ter a mesma base em duas tabelas de segmento.
- Partilhar uma frame por ter a mesma `frame reference` nas duas tabelas de paginação.

Memória Virtual:

Background

O código precisa de estar em memória para executar, mas raramente se usa um programa inteiro.

- Código para lidar com situações fora do comum é raramente executado.
- Estruturas de dados grandes costumam alocar mais memória do que a que realmente precisam.
- Mesmo que todo o programa seja usado, não é todo necessário ao mesmo tempo.

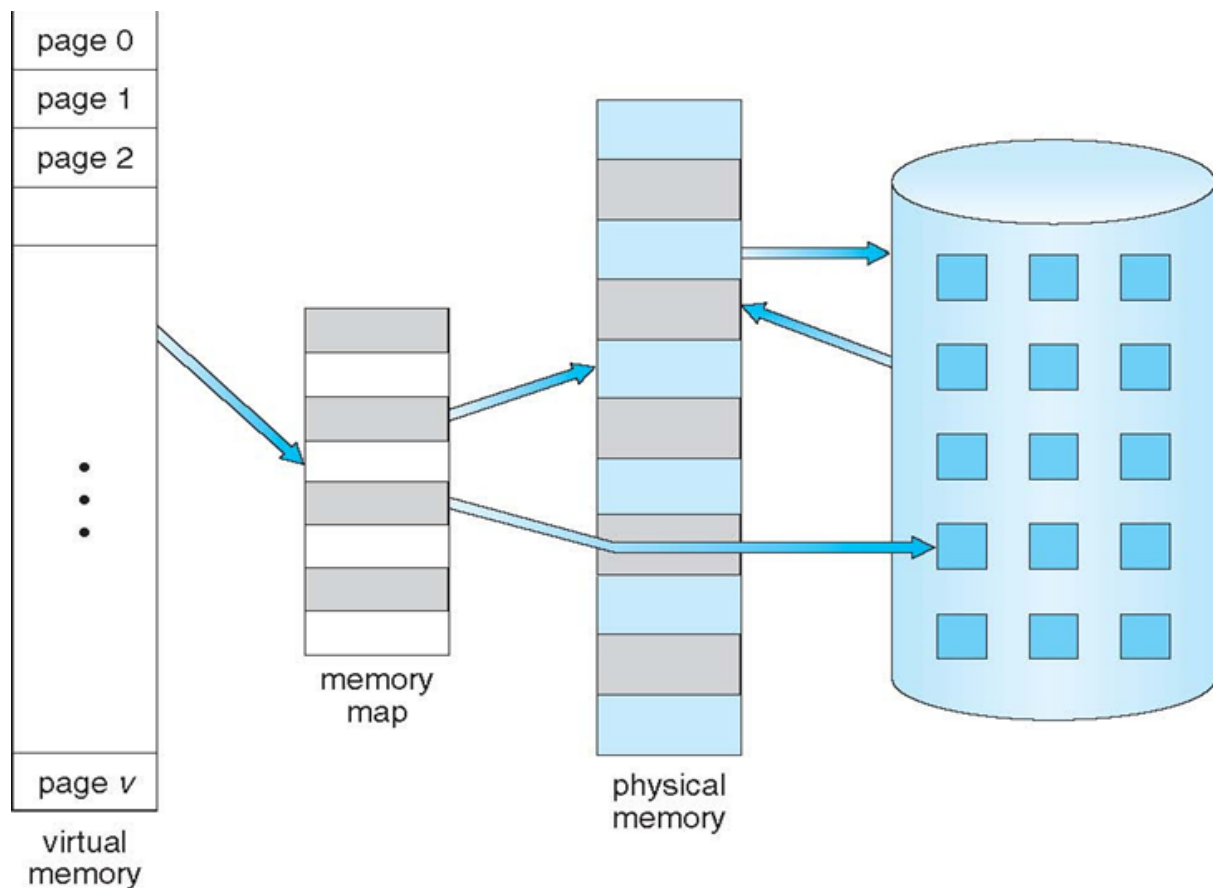
Memória Virtual é uma técnica que permite a execução de processos que não estão totalmente em memória.

- Resume a memória principal num extremamente grande e uniforme array de armazenamento.
- Liberta os programadores de preocupações como as limitações da armazenamento em memória.

Executar um processo que não se encontra totalmente em memória tem benefícios para o utilizador e para o Sistema Operativo:

- Permite um menor uso da memória;
- Permite a criação mais eficiente de projetos;
- Menor uso de operações de I/O necessárias para o load ou swaps de processos para a memória;
- Mais programas podem correr de forma concorrente;
- O espaço dos endereços lógicos pode ser mais largo que o espaço dos endereços físicos.

De memória virtual para memória física



Demand Pagine

Esta parte não foi referida nas aulas teóricas, mas pode ser complementar e necessária.

90-10 rule: Os programas gastam 90% do seu tempo em 10% do seu código

Demand Pagine é uma técnica que carrega uma page em memória apenas quando esta é necessária.

- Evita o uso desnecessário de I/O;
- Menos uso de memória e menos uso de I/O;
- Resposta mais rápida;
- Páginas que nunca são acedidas nunca são carregadas na memória física (normalmente essas páginas residem em memória secundária).

Espaço de endereços virtuais de um processo

Tem a ver com a visão lógica (virtual) de como um processo é guardado em memória.

Tipicamente, um processo começa num determinado endereço lógico (p.e. endereço 0) e existe em memória contígua até ao endereço lógico mais alto permitido.

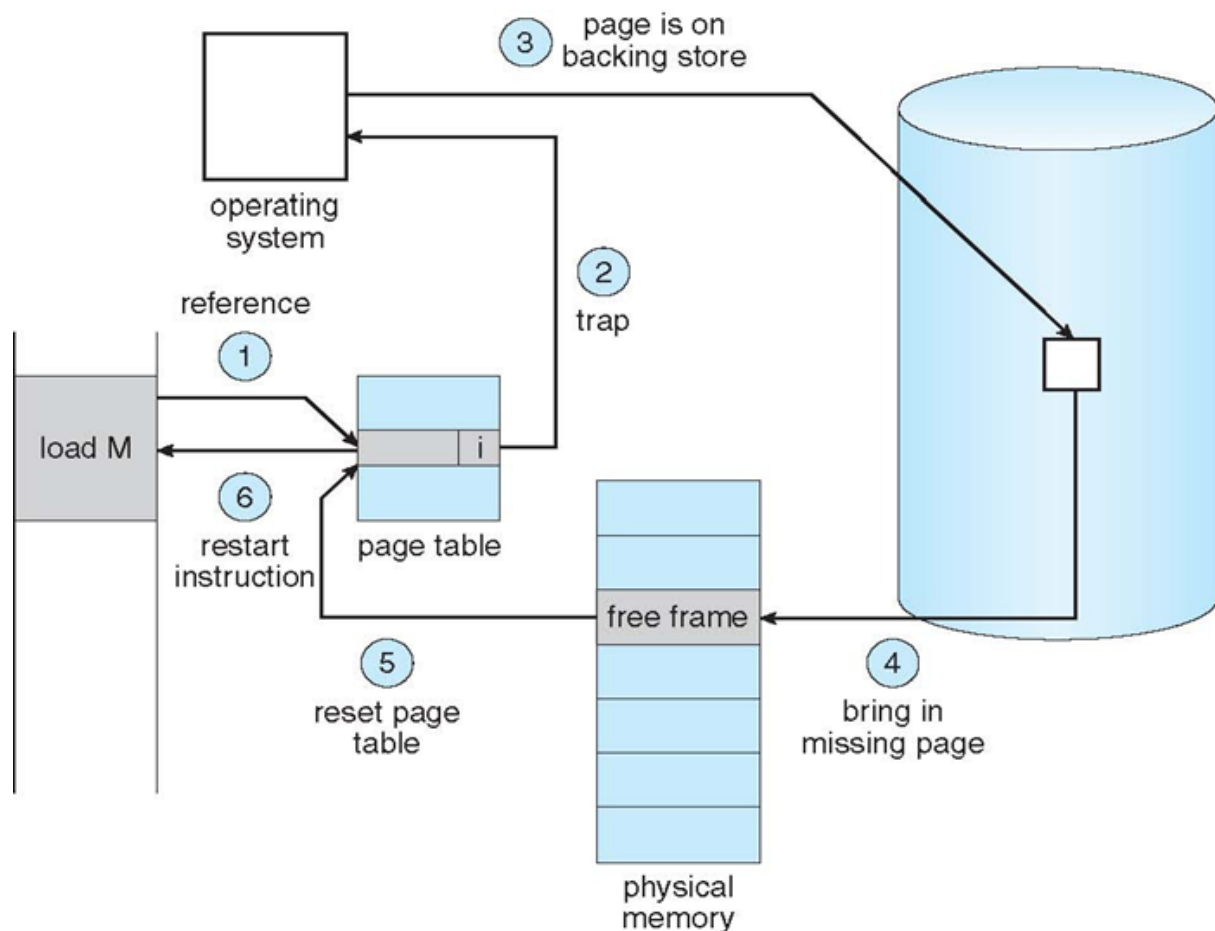
Page Fault

Page fault ocorre quando o hardware de paging acede a uma página marcada como inválida.

- Na tradução do endereço pela tabela de pages, o paging hardware irá notar que um bit inválido foi colocado, causando uma trap (exception/fault) no sistema operativo.

O procedimento para resolver uma page fault é simples:

1. Verificar o PCB para determinar se a referência era um acesso legal ou ilegal à memória (se ilegal terminar o processo).
2. Encontrar uma frame livre e mover a página em falta para essa frame (o carregamento é feito da backing store).
3. Dar reset à tabela de pages para indicar que a página está agora em memória.
4. Repetir a instrução que causou o page fault.



O que acontece se não existir nenhuma frame livre?

Poderá dar-se o caso de não existir nenhuma frame livre para carregar a página pretendida, nesse caso:

- O sistema operativo poderia dar swap out a um processo, libertando todas as frames que esse processo ocupava.
- A solução mais comum é **page replacement** (encontra alguma página que não está a ser usada e liberta-a).

Page replacement completa a separação entre memória lógica e memória física

- Uma grande memória virtual pode ser fornecida numa memória física mais pequena.
- A mesma página pode ser levada para a memória várias vezes.

Queremos um algoritmo de page replacement que resulte no menor número possível de page faults.

Como funciona o algoritmo de Page Replacement:

1. Seleciona a página "vítima";
2. Escreve essa página em disco;
3. Traz a página desejada para um novo espaço livre;
4. Atualiza a tabela de páginas.

Nota: A transfêrencia de duas páginas é necessária (uma fora || uma dentro)

FIFO Page Replacement (First-In First-Out)

É um exemplo de um algoritmo de Page Replacement.

A página que vai ser substituída é a mais antiga.

A FIFO queue é usada para seguir a idade das páginas:

- Quando uma página é trazida da memória é inserida na cauda da lista.
- E a página que se encontra na cabeça da lista é a próxima a ser substituída.

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2	2	4	4	4	0		0	0		7	7	7
	0	0	0		3	3	3	2	2	2		1	1		1	0	0
		1	1		1	0	0	0	3	3		3	2		2	2	1

page frames

LRU Page Replacement (Least Recently Used)

A página que vai ser substituída é aquela que não é utilizada à um maior período de tempo.

- LRU associa a cada página o tempo do seu último uso.
- Geralmente é um bom algoritmo (melhor que o FIFO).

HOW TO IMPLEMENT LRU PAGE REPLACEMENT ALGORITHM

Second Chance Page Replacement

Substitui uma página antiga, no entanto, não necessariamente a mais antiga.

- É semelhante ao FIFO mas usa um bit de referência.

Quando verifica a página na cabeça da lista, também verifica o bit de referência.

- Se o bit for 0 -> substitui a página
- Se o bit for 1 -> limpa o bit e move a página para a cauda (é dada uma segunda chance à página)

A página que teve a sorte de ganhar uma segunda chance agora só será substituída quando todas as páginas à sua frente na queue forem substituídas ou lhes for dada também uma segunda chance.

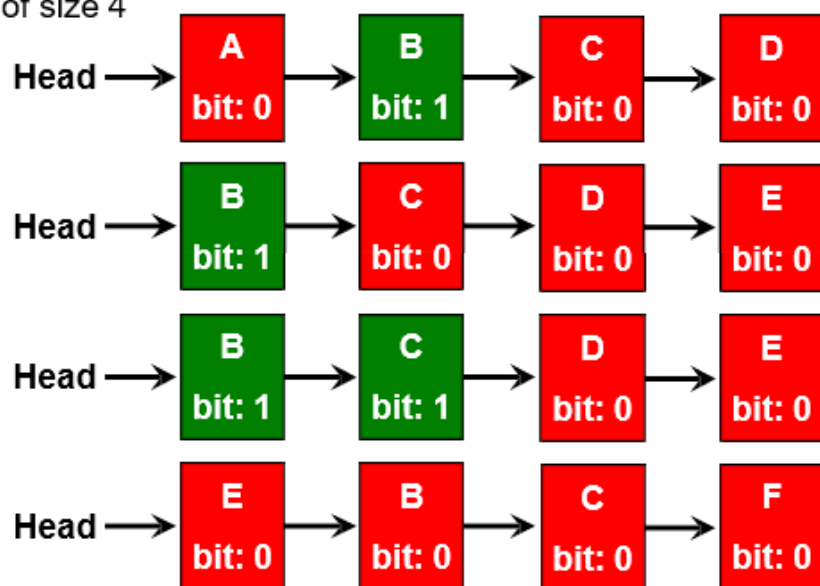
- Se uma página é usada frequentemente, o seu bit de referência é sempre colocado a 1 e essa página nunca será substituída.

Problema: Mover páginas para a cauda da queue é uma operação complexa.

Exemplo:

Consider a page table of size 4

- Page A arrives
- Page B arrives
- Access page B
- Page C arrives
- Page D arrives
- Page E arrives
- Access page C
- Page F arrives



Clock Page Replacement

É uma implementação mais eficiente do algoritmo anterior, as páginas são organizadas numa queue circular e existe um apontador que indica qual a próxima página a ser verificada. Quando é necessário escolher a página que vai ser trocada, o apontador avança até encontrar a página que tem o bit de referência a 0.

- À medida que avança vai apagando os bits das outras páginas
- Assim que a página for encontrada, ela é substituída e a nova página é inserida na queue circular na posição que a outra página deixou livre.

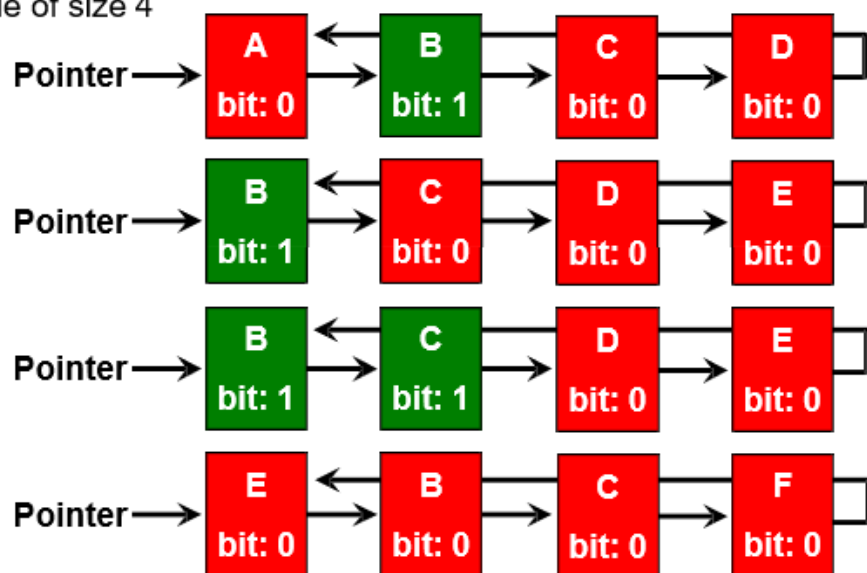
Encontrará sempre uma página ou vamos ter um loop infinit?

- No pior caso, quando todos os bits estão definidos, o ponteiro passa por toda a fila, é dada a cada página uma segunda chance e, em seguida, substitui a página inicial.

Exemplo:

Consider a page table of size 4

- Page A arrives
- Page B arrives
- Access page B
- Page C arrives
- Page D arrives
- Page E arrives
- Access page C
- Page F arrives



NRU Page Replacement (Not Recently Used)

Deixa em memória as páginas que foram usadas recentemente.

Considera-se um bit de acesso (A) e um bit de escrita (W) (definidos quando a página é modificada ou escrita) tornam-se possíveis as seguintes classes (A , W):

1. Class 0 = (0, 0) não foi recentemente usada nem modificada
- Melhor escolha para substituir
2. Class 1 = (0, 1) não foi recentemente usada mas foi modificada - A página teria de ser escrita para o disco antes de ser substituída.
3. Class 2 = (1, 0) recentemente usada mas não modificada - Provavelmente será usada novamente dentro de pouco tempo
4. Class 3 = (1, 1) recentemente usada e modifica - Provavelmente vai ser usada novamente dentro de pouco tempo e ainda seria necessário escrever a página novamente no disco antes de a substituir.

A configuração dos bits normalmente é feita pelo hardware.

How to implement a NRU Algorithm

Algoritmos Diferentes

Algoritmo	Características
Ótimo	Inexequível. Padrão para comparação.
NRU (não usado recentemente)	Aproximação grosseira.
FIFO	Leva à rejeição de páginas importantes.
Segunda Oportunidade	Melhoramento do FIFO.
Relógio	Solução realista.
LRU (menos recentemente usado)	Muito bom. Implementação exacta difícil.
NFU (menos frequentemente usado)	Aproximação grosseira do LRU.
Aging (envelhecimento)	Aproximação boa e eficiente do LRU.
Working set	Implementação ineficiente.
WSClock	Aproximação boa e eficiente.

Alocação de Frames

Como alocar uma quantidade fixa de memória física entre processos?

- Não podemos alocar mais do que o número total de frames avaliáveis.
- Devemos alocar um número mínimo de frames para manter justo o rácio de page-faults.

Equal Allocation -> Alocar uma parte igual para todos os processos:

- Se existem 100 frames e 2 processos -> 50/50;

Proportional Allocation -> Alocar proporcionalmente ao tamanho dos processos:

- Se existem 100 frames e 2 processos, um dos processos tem 20 pages e outro 180 pages -> são dados 10 frames ao primeiro ($20/200 \times 100$) e 90 frames ao segundo ($180/200 \times 100$).

Priority allocation -> Usa-se alocação proporcional, mas, em vez de termos em conta o tamanho do processo, temos em conta a sua prioridade.

- No caso de page-fault, escolhemos um frame desse processo ou a frame de um processo de lower priority.

Global Vs Local Replacement

Global Replacement -> Permite a um processo selecionar uma frame de todo o conjunto de frames, mesmo aquelas que estão atualmente alocadas a outro processo. (Um processo pode "roubar" uma frame de outro).

- O tempo de execução do processo pode variar muito, uma vez que o seu comportamento de paginação depende dos outros processos a executar em simultâneo.
- Resulta numa maior produção do sistema, logo é mais comum.

Local Replacement -> Cada processo seleciona apenas do seu próprio conjunto alocado de frames

- Performance mais consistente por processo.

- Pode sobrecarregar a memória, mas não permite o uso das pages menos usadas por outros processos

Thrashing

Thrashing ocorre quando um processo está ocupado a trocar pages in e out e demora mais tempo nessa troca do que a executar.

Se um processo não tem frames suficientes a sua taxa de page-fault é bastante alta.

- Page fault para obter a page que precisa.
- Substitui frames existentes.
- Uma vez que todas as suas páginas estão em uso ativo, rapidamente deve precisar de trazer de volta a página que foi substituída.
- Como consequência, entra-se novamente num estado de page fault (repetidamente) substituindo páginas que vai precisar de ir buscar imediatamente.

Isto leva a uma utilização baixa do CPU.

- O sistema operativo passa maior parte do tempo a fazer swapping para o disco.

Muitas operações de swap acabam por danificar o disco, sendo por isso, recomendados os backups.

Thread

Conceito:

Até agora vimos que um processo é um programa em execução com um único fio (thread) de execução, o que significa que só pode executar uma tarefa de cada vez.

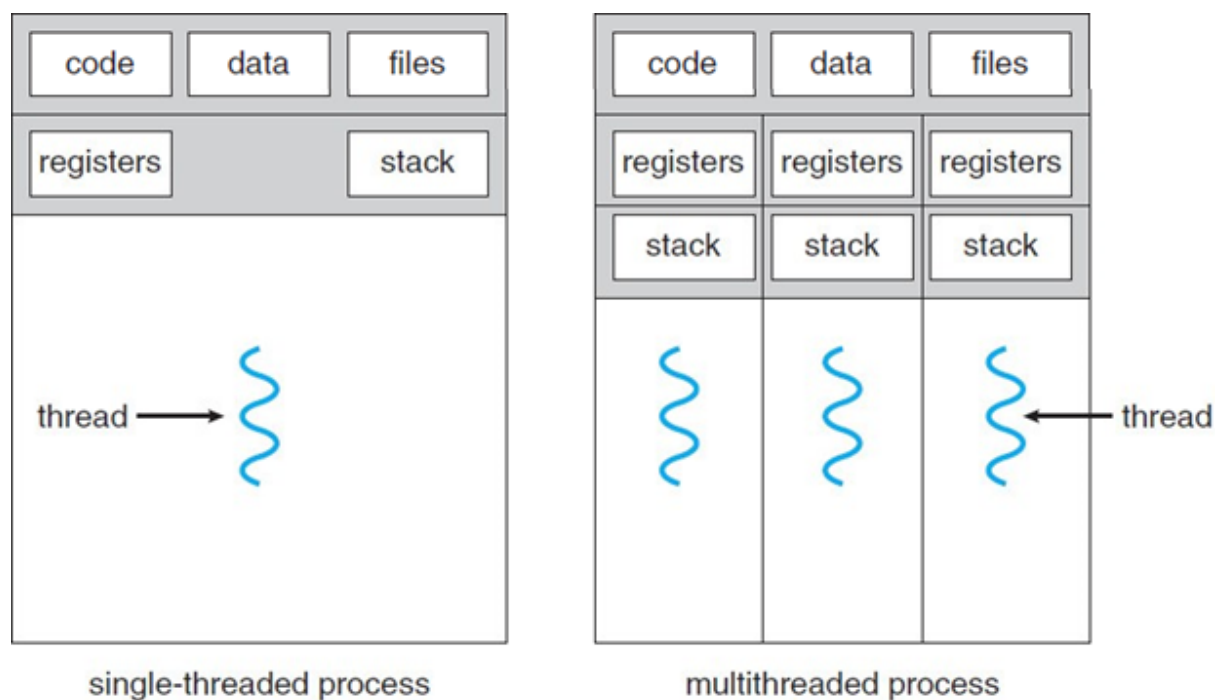
Sistemas operativos mais modernos estenderam o conceito de processo de modo a permitir que o mesmo tenha acesso a múltiplas threads de execução e assim executar mais do que uma tarefa de cada vez.

- A maioria das aplicações modernas pode ser vista como um conjunto de múltiplas tarefas.
- O processamento de tarefas independentes ou eventos assíncronos podem ser intercalados atribuindo uma thread separada para lidar com cada tarefa ou evento.

Processos singulares e em multithreading

Single-threaded é a abordagem tradicional de uma única thread de execução por processo, em que o conceito de thread não é reconhecido.

Multithreading refere-se à habilidade de um sistema operativo suportar múltiplos caminhos simultâneos de execução dentro de um único processo.



Principais Benefícios:

Existem 4 benefícios na programação multithreaded:

1. **Performance** -> Em geral, é significativamente mais demorado criar e gerir processos do que threads, a criação de threads é mais barata do que a criação de processos e trocar entre threads tem um overhead mais baixo do que trocar entre contextos.

2. **Partilha de Recursos** -> Threads partilham memória por default e os recursos do processo a que pertencem, enquanto vários processos têm de usar mecanismos complexos do sistema operativo para partilhar memória ou outros recursos.
3. **Resposta** -> O multithread de uma aplicação interativa pode permitir que um programa continue a funcionar mesmo que parte dele esteja bloqueada ou esteja a realizar uma operação longa, isto torna-se especialmente importante para as interfaces do utilizador.
4. **Paralelismo** -> Um processo em single-threaded pode correr em apenas um core, independentemente de quantos estejam disponíveis. Em multithreading, as threads podem estar a funcionar em paralelo em diferentes processadores/núcleos de processamento.

Estados de uma thread

Assim como os processos, os estados-chave para as threads são:

- **Running** -> Instruções estão a ser executadas.
- **Waiting/Blocked** -> A thread está à espera que algum evento ocorra.
- **Ready** -> A thread está à espera de ser atribuída a um processador.

Existem 4 operações básicas que podem alterar o estado de uma thread:

- **Spawn** > Quando uma nova thread é gerada (A thread é fornecida com o seu próprio contexto e espaço na stack e é colocada na ready queue).
- **Block** -> Quando uma thread precisa de esperar por algum evento que vai ocorrer (a thread guarda o seu contexto de registo e os stack pointers).
- **Unblock** > Quando o evento pela qual a thread estava a aguardar ocorre (a thread é movida para a ready queue).
- **Finish** > Quando a thread completa a execução (os registos de contexto e o espaço na stack são libertados).

Dispositivos de Armazenamento

Transferência de Dados

Um unidade de disco é anexada a um computador através de um conjunto de fios chamados de I/O bus

- EIDE, ATA, SATA, USB, SCSI, etc...

As transferências de dados são levadas a cabo por hardware especial conhecido por controladores.

- O `host controller` é o controlador na extremidade do computador do bus.
- O `disk controller` é o controlador incorporado em cada disco.

Para realizar um operação de I/O no disco:

- O sistema operativo começa por colocar um comando no host controller;
- O host controller em seguida envia esse comando para o disk controller;
- O disk controller opera o hardware do HDD/SSD para realizar o comando;
- A transferência de dados no HDD/SSD acontece entre a superfície do disco e uma cache incorporada no disk controller;
- A transferência de dados para o host ocorre entre a cache e o host controller.

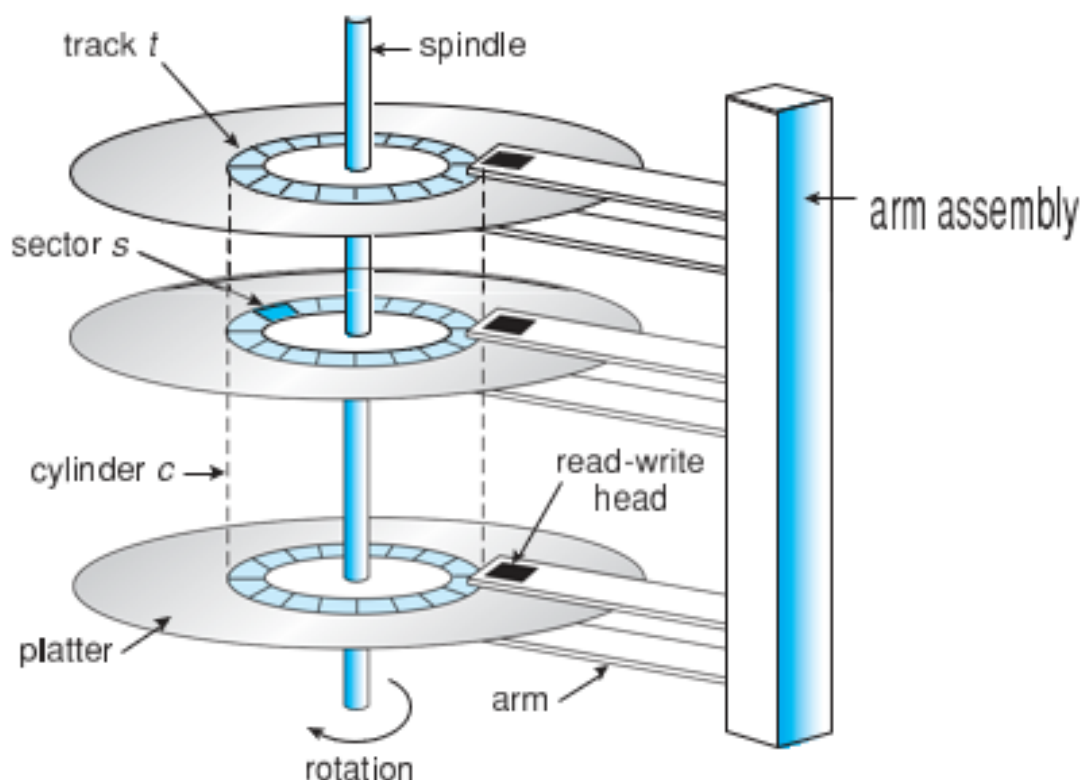
Tempos de Latência:

Qualquer operação de I/O é um processo de 5 fases:

- `Queuing time` -> Tempo para o driver do dispositivo processar o pedido.
- `Controller time` > Tempo para o disk controller levar a cabo o pedido.
- `Seek time` > Tempo para mover o braço do disco para o cilindro desejado (HDD).

- **Rotational latency** > Tempo para o setor desejado rodar debaixo da cabeça do disco.
- **Transfer time** > Tempo para transferir os dados desejados entre a drive e o computador.

Latência do Disco -> dada pela soma de todos os tempos acima. Corresponde ao tempo total entre um pedido e a disponibilização do seu resultado



Agendamento do Disco (HDD):

Um pedido I/O especifica várias informações:

- O tipo de operação (input ou output);
- Os endereços de disco e de memória para transferência;
- O número de setores a ser transferidos.

O sistema operativo mantém uma fila de pedidos pendentes por disco:

- Quando um pedido é completado, o sistema operativo escolhe qual o próximo pedido a atender.

Como é que o sistema operativo programa a manutenção dos pedidos do disco?

- Existem imensos algoritmos de otimização.
- Os algoritmos de otimização apenas fazem sentido quando as queues existem, de outra forma um pedido pendente pode ser executado imediatamente.

Agendamento do Disco (SSD)

Algoritmos de agendamento de discos concentram-se principalmente em minimizar a quantidade de movimentos da cabeça (ver constituição do HDD na imagem em cima), no entanto os discos SSD não contêm esse mecanismo.

- Muitos dos schedulers de SSDs ordenam usando o método de FCFS.