# Sistemas Operativos

Resumo

# Conteúdo

# Capítulo 1

# Sincronização e Threads

## 1.1 Sincronização

Acessos simultâneos a dados partilhados por vários fluxos de execução (entre processos ou entre threads) podem gerar inconsistências. São, portanto, necessários mecanismos para garantir a serialização de fluxos de execução concorrentes.

- Processos cooperando num mesma aplicação usando memória partilhada
- Threads dentro de um mesmo processo utilizando as mesmas varáveis globais

### 1.1.1 Secção Crítica

A secção crítica é a zona de código de um processo/thread que só pode ser executada por um processo/thread de cada vez, em exclusão mútua.

Existe uma competição para executar esta secção, que pode ter forma diferente entre cada processo/thread.

Sem exclusão mútua, o resultado da execução dos processos/threads é indeterminado.

### 1.1.2 Mutex Locks

Uma das soluções para o problema da secção crítica são os mutex locks. Consiste num número inteiro positivo, alternando entre 1 e 0, mantido pelo sistema operativo, que pode ser acedido por vários processos.

- Variável inteira
- Valor 1 por defeito
- Tem associada uma fila de espera para os outros processos que querem executar a sua secção crítica

Existem apenas duas operações associadas aos mutex locks:

- Para um processo/thread entrar na secção crítica deve primeiro obter o mutex através de `acquire()`
  - Se `mutex = 1`
    - `mutex = 0`
    - O processo/thread executa a secção crítica
  - Se `mutex = 0`
    - Coloca o processo/thread na fila de espera
    - Bloqueia o processo/thread
- Após executar/sair da secção crítica o mutex deve ser libertado através de `release()`
  - Se existirem processos/threads na fila de espera
    - Remove um processo/thread da fila de espera para executar
    - Desbloqueia o processo/thread
  - Se não existirem processos/threads na fila de espera
    - `mutex = 1`

### 1.1.3 Semáforos

Os semáforos representam uma outra solução para o mesmo problema. Tal como os mutex locks, estes consistem num número inteiro positivo mantido pelo sistema, no entanto, podem ser inicializados com qualquer valor $>= 0$. A principal diferença entre os mutex e os semáforos é que estes são um mecanismo de sinalização, podendo ser acedidos por qualquer processo/thread, enquanto que os mutex são mecanismos de bloqueio, pelo que têm de ser libertados pelo mesmo processo/thread que os bloqueou. Existem 2 tipos de semáforo:

#### Semáforo de Contador (Counting Semaphore)

- O controlo de acesso a um recurso é feito com sincronização de $n$ recursos distintos, i.e, permite que vários processos/threads acedam a um recurso simultaneamente
- O valor do semáforo está entre 0 e $N$, onde $N > 1$

#### Semáforo Binário(Binary Semaphore)

- O valor do semáforo pode ser 0 ou 1, tal como nos mutex locks
- O controlo de acesso a um recurso é feito por exclusão mútua

Existem duas operações atómicas realizáveis sobre semáforos:

- `sem_wait()` - Indica que o processo/thread quer aceder a um recurso
  - Se `sem > 0`
    - Decrementa `sem`

        * O processo/thread continua a executar o seu código, que pode ser uma secção crítica
    - Se `sem = 0`
        * Coloca o processo/thread na fila de espera
        * Bloqueia o processo/thread

- `sem_post()` - Indica que o processo/thread quer libertar um recurso
    - Se `sem > 0`
        * Incrementa o valor do semáforo
    - Se `sem = 0`
        * Remove um processo/thread da fila de espera
        * Desbloqueia o processo/thread

O sistema operativo garante que o semáforo tem sempre um valor válido (>= 0).

Os semáforos podem ser usados para garantir que o acesso a um recurso é efetuado em exclusividade, inicializando a 1 e fazendo `sem_wait()` antes de aceder e `sem_post()` após, ou para bloquear um processo/thread até que outro chegue a uma dada instrução, bloqueando um processo com `sem_wait()` até que outro execute `sem_post()`.

### 1.1.4 Problemas de Sincronização

#### Problema Produtor/Consumidor

No problema do produtor/consumidor, um conjunto de informações é partilhado por vários processos. Os produtores criam informações e inserem-nas no buffer. Os consumidores removem as informações do buffer. Surgem então dois problemas:

- Apenas um consumidor pode ler do buffer de cada vez
- Um consumidor só pode ler um item após um produtor produzir um item
- Um produtor escreve no buffer se houver espaço no buffer

Este problema pode ser resolvido através de 3 semáforos: `mutex`, `empty` e `full`. `empty` é inicializado com a mesma capacidade do buffer e indica se existe espaço disponível no buffer para escrever. `full` é inicializado a 0 e indica se existem items no buffer para serem lidos. `mutex` é inicializado a 1 e é utilizado para delimitar a secção crítica, atuando como um semáforo regular.

#### Problema Leitores/Escritores

Neste problema, um conjunto de dados é partilhado por vários processos. Os leitores apenas acedem para leitura, não alterando os dados. Os escritores podem aceder aos dados para leitura e escrita. Surgem os seguintes problemas:

- Vários leitores podem ler ao mesmo tempo
- Num dado momento, apenas um escritor pode aceder aos dados

Este problema é resolvido através do utilização de 2 semáforos: `mutex` e `rw_mutex`.

```
Escritor
do{
    sem_wait(&rw_mutex);
    # escreve dados
    sem_post(&rw_mutex);
} while (true);
```

```
Leitor
do{
    sem_wait(&mutex);
    readcount += 1;
    if (readcount == 1)
        sem_wait(&rw_mutex);
    sem_post(&mutex);
    # lê dados
    sem_wait(&mutex);
    readcount -= 1;
    if (readcount == 0)
        sem_post(&rw_mutex);
    sem_post(&mutex);
} while (true);
```

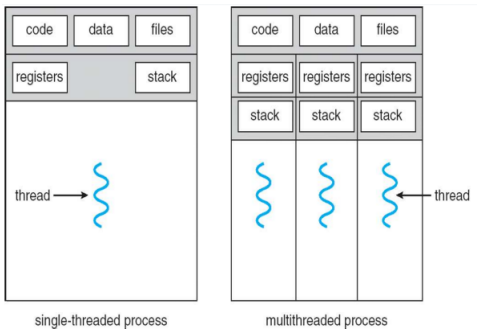
### 1.1.5 Monitores

Os monitores foram criados para evitar os problemas na utilização de semáforos. O monitor baseia-se no princípio que a sincronização é necessária para controlar o acesso a dados, ou estruturas de dados, e código, que podem ser utilizados por diversos processos de forma concorrente.
Os monitores definem essa estrutura de dados, lógica, etc. Esta definição é próxima da definição de classe das linguagens de programação OO.

- As variáveis internas só são acedidas pelos procedimentos do monitor
- Os procedimentos são acedidos do exterior
- Só um processo de cada vez pode estar ativo no monitor

## 1.2 Threads

A utilização de vários processos distintos implica overhead na comunicação de dados e comutação de contexto e existem situações em que seria benéfico ter a possibilidade de partilha de recursos entre dois processos. As threads foram criadas como uma solução a estes problemas, funcionando como uma sequência de execução num processo.



O Scheduler atribui o CPU a cada processo, criando um fluxo de execução, ou single threaded process. No modelo clássico existe apenas um fluxo de execução, mas no modelo multithreaded, em cada processo existem vários fluxos de execução ou threads.

As várias threads de um processo partilham o espaço de endereçamento, mas têm contextos de execução distintos. Uma thread partilha com as outras do mesmo processo:

- Secção de código
- Secção de dados
- Recursos do processo (ficheiros, etc.)

### 1.2.1 Modelos Multithreading

Existe ainda o conceito de nível da thread. Threads de nível utilizador são geridas por bibliotecas ao nível utilizador, pelo que o kernel não reconhece a sua existência, tratando-as como se fossem processos single-threaded. As threads do núcleo são suportadas e geridas pelo kernel diretamente.

#### Modelos many-to-one

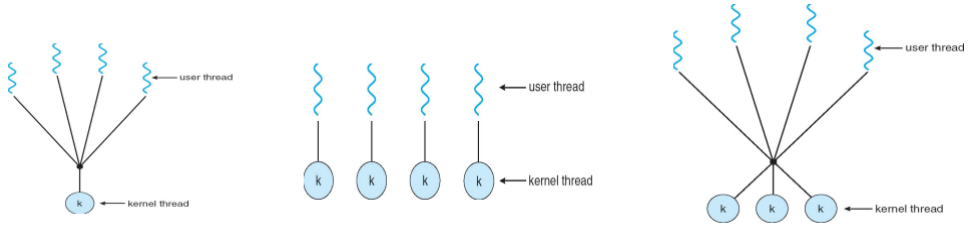
Associam várias threads do nível utilizador a uma thread do nível núcleo

#### Modelos one-to-one

Associam cada thread do nível utilizador a uma thread do nível núcleo. Gera mais concorrência que o many-to-one

#### Modelos many-to-many

Associam $N$ threads do nível utilizador a um nº igual ou inferior de threads do nível núcleo. Permite gerir o nº de threads do kernel.




Para além destes modelos existe ainda o conceito de thread pools. As thread pools são geralmente usadas em servidores web, onde se cria um conjunto de threads que permanecem á espera de trabalho. Esta prática melhora o desempenho, visto que as threads já estão criadas mas limita o número de threads que podem ser criadas.

### 1.2.2 Escalonamento de Threads

Existem dois tipos de escalonamento de threads, de acordo com a implementação do package de threads:

- Local: implementado e efetuado na biblioteca de threads, para decidir que thread irá correr em modo utilizador associada às threads kernel disponíveis
  - many-to-one
  - many-to-many
- Global: implementado e efetuado no kernel
  - one-to-one
  - scheduling threads $\equiv$ scheduling processos

Alguns CPUs fornecem a possibilidade de criar processadores lógicos (threads) dentro de cada core (HyperThreading). Um CPU com $x$ threads e $y$ cores consegue executar $y$ processos de cada vez e $x$ threads simultaneamente, $\frac{x}{y}$ de cada processo.

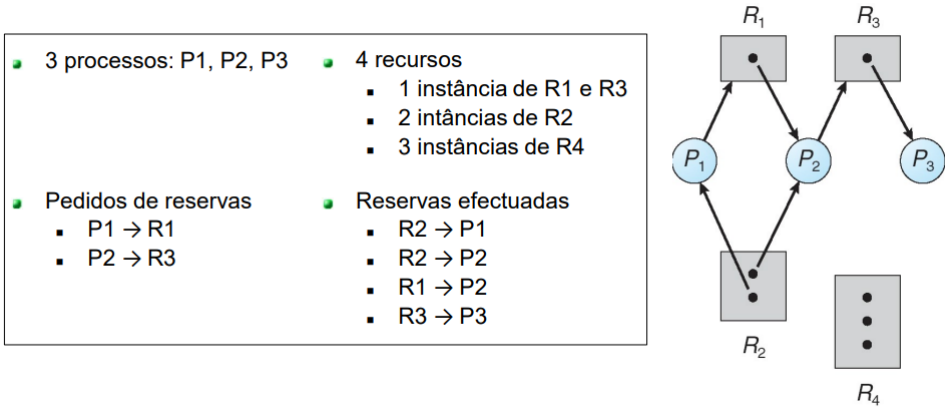
## 1.3 Deadlocks

Uma situação de deadlock pode ocorrer devido ás seguintes situações:

- Exclusão mútua - Apenas um processo de cada vez pode usar um determinado recurso
- Hold and wait - Um processo possui pelo menos um recurso (hold) e está à espera de obter recursos adicionais (wait) reservados para outros processos
- Sem preempção - Um recurso apenas pode ser libertado pelo processo que o possui, depois do processo ter terminado a sua tarefa
- Espera circular - Existe um conjunto de processos em espera $\{P_0, P_1, ...P_n\}$ tal que $P_0$ está à espera de um recurso possuído por $P_1$. $P_1$ está à espera de um recurso possuído por $P_2, ..., P_n$ está à espera de um recurso possuído por $P_0$.

### 1.3.1 Modelo

É possível modelar um sistema de forma a detetar deadlocks. Este modelo é feito na forma de um grafo em que os vértices $R_1, R_2, ...$ representam os recursos disponíveis, cada um com $R_i$ instâncias e $P_1, P_2, ...$ representam os processos em execução. Existem ainda dois tipos de arcos: Os arcos do tipo $P_i \rightarrow R_j$ representam pedidos de reserva e os arcos do tipo $R_i \rightarrow P_j$ representam as reservas efetuadas. Por exemplo:




Se este grafo não tem ciclos, então não existem deadlocks, caso contrário:

- Se o ciclo envolve apenas recursos com uma instância então existe deadlock
- Se cada tipo de recurso tem várias instâncias depende

### 1.3.2 Gerir Deadlocks

Existem duas formas de gerir deadlocks:

- Garantir que o sistema nunca entra em deadlock, prevenindo e evitando-os
- Permitir que o sistema entre em deadlock, detetar e recuperar

**Prevenir**

- Exclusão mútua
  - Podia ser evitada se todos os recursos fossem partilháveis sem problemas de concorrência (ex. recursos apenas de leitura)
  - Genericamente não podemos prevenir deadlocks evitando exclusão mútua
- Hold and wait
  - Tem de garantir que sempre que um processo faz um pedido de reserva de um recurso, não possui outros recursos
  - Requer que um processo reserve todos os recursos de que vai necessitar antes de ter início a sua execução
  - Alternativamente, o processo apenas pode reservar recursos quando não possui nenhum
- Sem preempção
  - Se um processo faz um pedido de reserva de um recurso que não pode ser imediatamente satisfeito, então todos os recursos que ele possui são libertados

  - Quando um processo faz um pedido de reserva, se o recurso não está disponível, verifica-se se ele está reservado por um processo que esteja em espera, em caso afirmativo o recurso é preemptivo
  - Aplica-se a recursos cujo estado pode ser guardado e recuperado facilmente

- Espera circular
  - Define uma ordem total para todos os recursos e requer que os pedidos de reserva sejam efetuados de acordo com esta ordem

**Evitar**

Para evitar um deadlock é necessário monitorizar o estado do sistema de modo a impedir uma reserva que dê origem a um estado de interbloqueio, pelo que requer informação sobre reserva de recursos à priori.

Um modelo mais simples requer que cada processo declare o número máximo de recursos de cada tipo que ele pode vir a necessitar.

Por outro lado, é possível usar um algoritmo que examina dinamicamente o estado da reserva de recursos de modo a evitar espera circular.

Existem então dois estados possíveis:

- Estado seguro: interbloqueio é impossível
- Estado inseguro: interbloqueio é possível

Critério de estado seguro:

Quando um processo faz um pedido de reserva, o sistema tem de determinar se essa reserva deixa o sistema num estado seguro. Isto é, existe uma sequência $< P_1, P_2, ..., P_n >$ de todos os processos tal que para cada processo $P_i$, os recursos que necessita possam ser satisfeitos pelos recursos disponíveis mais os recursos possuídos por todos os $P_j$, com $j < i$.

- Se os recursos que $P_i$ necessita não estão todos disponíveis, espera até que todos os $P_j$ terminem
- Quando $P_j$ termina $P_i$ executa, depois $P_{i+1}$...

Para garantir o estado seguro, havendo apenas uma instância de cada tipo de recurso, pode-se usar o grafo de reserva de recursos. Neste grafo existe um novo tipo de arco a tracejado, que indica que o processo pode vir a fazer o pedido de reserva. Se este grafo não tiver ciclos assegura-se o estado seguro, caso contrário verifica-se o estado inseguro.

Se existirem várias instâncias de recursos usa-se o banker's algorithm. Neste algoritmo cada processo tem de indicar a sua máxima utilização à priori e assume-se que pode ter de esperar ao fazer um pedido de reserva e que após ter os recursos tem de os libertar num período de tempo finito. Por exemplo:

Necessita = Máximo – Reservados

| | *Necessita* |
|---|---|
| | A B C |
| P1 | 7 4 3 |
| P2 | 1 2 2 |
| P3 | 6 0 0 |
| P4 | 0 1 1 |
| P5 | 4 3 1 |

| | *Reservados* | *Máximo* | *Disponível* |
|---|---|---|---|
| | A B C | A B C | A B C |
| P1 | 0 1 0 | 7 5 3 | 3 3 2 |
| P2 | 2 0 0 | 3 2 2 | |
| P3 | 3 0 2 | 9 0 2 | |
| P4 | 2 1 1 | 2 2 2 | |
| P5 | 0 0 2 | 4 3 3 | |

O sistema está em estado seguro porque a sequência <P2, P4, P5, P3, P1> satisfaz o critério de estado seguro

### Deteção de Deadlocks

Desta forma permite-se que o sistema entre em deadlock, recuperando posteriormente. Existe um algoritmo de deteção que determina quando procurar interbloqueios e aplica um esquema de recuperação:

- Terminação de um, ou mais, processos em interbloqueio até que este desapareça
- Preempção de recursos, i.e., selecionar um processo que perde o recurso, colocá-lo num estado seguro anterior e reiniciá-lo. Repetir se necessário

# Capítulo 2

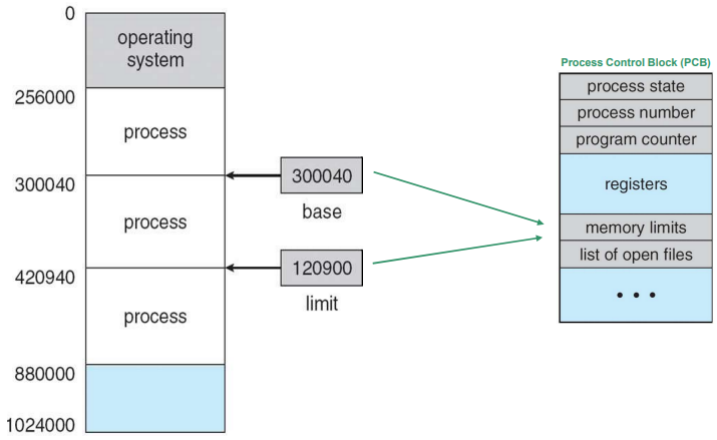
# Memória

## 2.1 Memória Principal

A memória principal, os registos e a cache únicos dispositivos de armazenamento que o CPU pode aceder diretamente.

Quando um processo é carregado em memória é-lhe atribuído um espaço de endereçamento físico, que pode ser qualquer zona de memória que esteja livre, pelo que deve haver proteção da memória, de forma a que os processos tenham sempre espaços físicos distintos.

Um processo sabe qual é o seu espaço de endereçamento físico através de uma ligação entre instruções e memória.



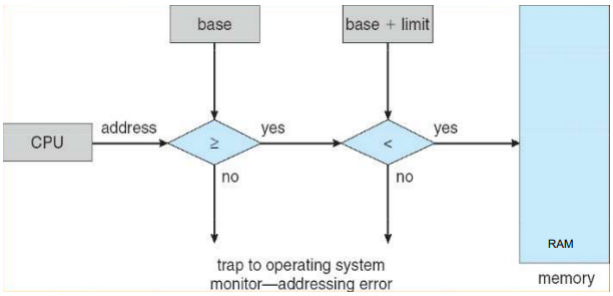

### 2.1.1 Proteção

É importante considerar a distinção entre os diferentes tipos de endereço:

- Endereço lógico: gerado pelo processador, endereço virtual
- Endereço físico: endereço usado pela memória principal, endereço real

Os programas não manipulam endereços físicos, a unidade de gestão de memória (MMU), inserida no CPU faz a tradução de endereços lógicos para físicos.

Através dos registos base e limite (ambos com valores diferentes para cada processo) implementado um esquema de proteção da seguinte forma: Quando o endereço gerado pelo CPU é inferior à Base ou superior ao (Base + Limite) é gerada uma exceção para o sistema operativo.




Assim, o registo base contém sucessivamente o endereço de inicio de cada processo ativo e o registo de limite contem o valor máximo do intervalo de endereços lógicos autorizados para cada processo.

### 2.1.2 Tradução de Endereços Lógicos em Físicos

**Forma 1**

- Os endereços lógicos e físicos são idênticos quando calculados em tempo de compilação e carregamento
- A MMU verifica se o endereço lógico é maior ou igual do que o registo base e menor que a (base + registo limite)

**Forma 2**

- Os endereços lógicos e físicos são diferentes quando calculados em tempo de compilação e carregamento
- A MMU verifica se o endereço lógico é menor que o registo limite, se sim adiciona o valor do registo de realocação (base) ao endereço lógico

A todos os endereços lógicos gerados pelo CPU é adicionado o valor contido num registo de realocação.

### 2.1.3 Swapping

O swapping ocorre quando existe falta de memória principal e consiste em retirar processos da memória temporariamente, armazenando-os temporariamente em disco e mais tarde devolvendo à memória.

### 2.1.4 Alocação de Memória aos Processos

Existem 3 forma de alocar memória aos processos: alocação contígua, segmentação e paginação.

### Alocação Contígua

A alocação de memória contígua é feita através do particionamento do espaço físico disponível em múltiplos blocos.

- Um bloco livre é um espaço de memória livre localizado em qualquer sítio da memória física
- Quando um processo é carregado é-lhe atribuído um bloco livre
- O sistema operativo mantém informações sobre os blocos livres e blocos ocupados

É então necessário saber responder a um pedido de alocação de $N$ bytes a partir de uma lista de blocos livres:

- First-fit - alocar o primeiro bloco livre suficientemente grande para conter $N$ bytes
- Best-fit: alocar o mais pequeno bloco que seja suficientemente grande para conter $N$ bytes
  - É preciso percorrer a lista toda, se esta não estiver ordenada por tamanhos
  - Produz melhores resultados, com menos desperdícios
- Worst-fit: alocar o maior bloco que seja suficientemente grande
  - Necessita percorrer a lista toda
  - Produz fragmentos maiores

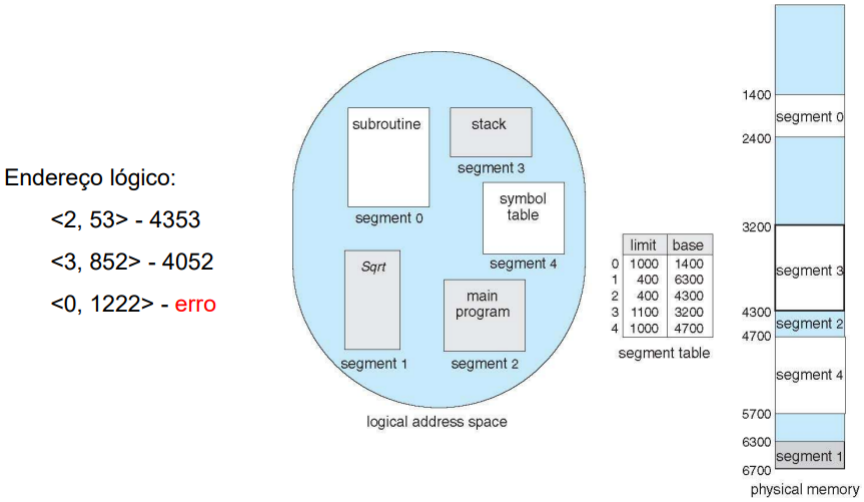
### Segmentação

O espaço de endereçamento lógico é constituído por um conjunto de segmentos que podem ter tamanhos diferentes e cada um tem um identificador. Um endereço lógico é constituído pelo identificador do e o deslocamento no segmento (offset). Um processo tem diversos segmentos: Código, bibliotecas, variáveis, pilha, etc.

A tradução de endereços é feita com base numa tabela de segmentos:

- Endereço lógico: <segmento, deslocamento>
- Tabela de segmentos:
  - Base - contém o endereço de início da memória física associada ao segmento
  - Limit – define a dimensão do segmento

Segue-se um exemplo:



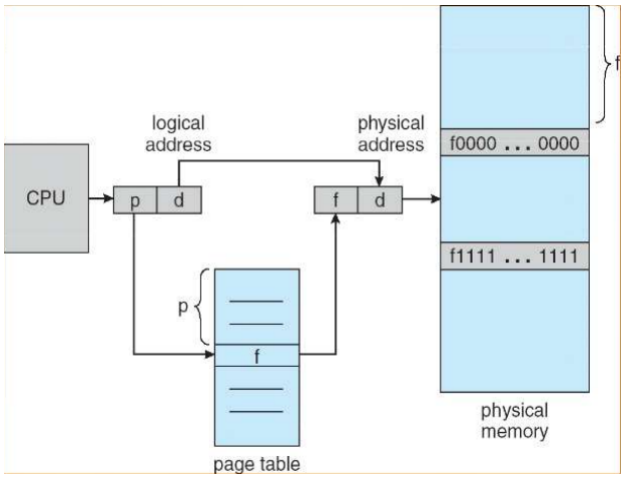

### Paginação

Neste modelo a memória física está dividida em blocos de dimensão fixa (frames), sendo que a sua dimensão é sempre uma potência de 2. A memória lógica está dividida em blocos da mesma dimensão (páginas).

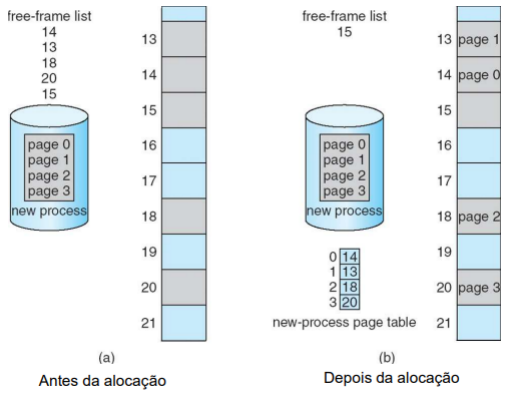
- O sistema operativo mantém uma lista de frames livres
- Para executar um programa com tamanho de $n$ páginas, é necessário procurar $n$ frames livres e carregar o programa em memória
- Tabela de páginas é usada para traduzir endereços lógicos em físicos

A tradução de endereços é feita da seguinte forma:

- Endereço lógico: <página, deslocamento>
  - Número de página (p) – usado como índice numa tabela de páginas que contém o endereço de base da frame correspondente em memória física
  - Deslocamento (offset) (d) – deslocamento dentro da página. É combinado com o endereço de base da frame correspondente à página para formar o endereço físico requerido na memória




Segue-se um exemplo:




De forma a proteger a memória associa-se ainda um bit de proteção associado a cada entrada da tabela de páginas que assume os seguintes estados:

- Válido (v): a página pertence ao espaço de endereçamento do processo
- Inválido (i): a página não pertence

Alternativamente é possível usar um modelo mais sofisticado, com um bit de leitura/escrita.

## 2.2 Memória Virtual

A MMU introduz uma diferenciação entre a memória lógica do utilizador e a memória física do sistema. Dado que apenas algumas partes de cada processo precisam de estar em memória para execução o espaço de endereçamento lógico pode ser maior que o físico. Algumas partes do espaço físico podem ser partilhadas por diferentes espaços lógicos de vários processos. Pode ser implementada através de:

- Paginação - Demand paging (paginação sob demanda)
- Segmentação - Demand segmentation (segmentação sob demanda)

### 2.2.1 Paginação sob Demanda

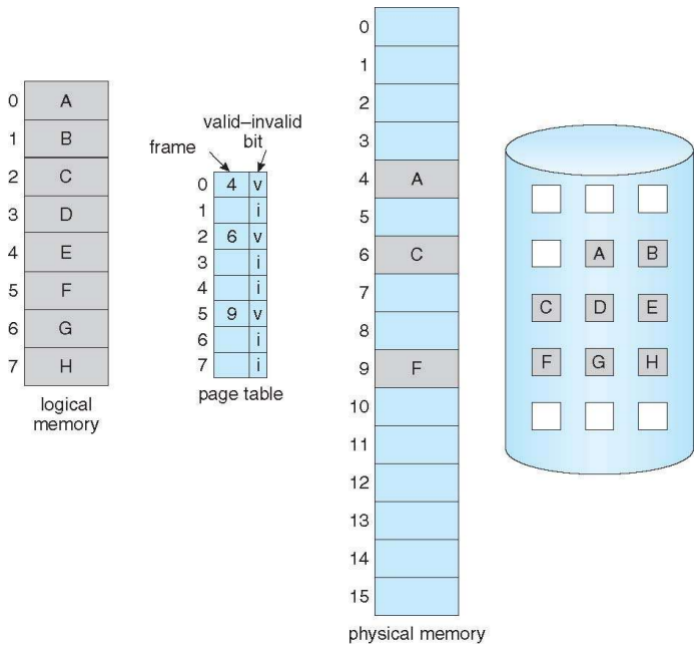
Este princípio consiste em trazer páginas para a memória principal só quando são necessárias (lazy swapping). Um swapper que gere páginas é designado por pager (paginador). Ao aceder a uma página as seguintes situações podem ocorrer:

- Página válida e presente: acesso autorizado
- Referência inválida: gerar exceção e terminar o processo
- Referência válida mas não presente: gerar exceção (page fault) e trazer o conteúdo da página para memória

A cada página está associado um bit de proteção que indica o estado da página:

- $v$: em memória
- $i$: não presente

Inicialmente, o bit de validade é posto a $i$ em todas as páginas sendo passado para $v$ à medida que as páginas são carregadas. Durante a tradução de endereços, se o bit estiver a $i$ é dada uma page fault. Segue-se um exemplo de um processo parcialmente carregado:



#### Tratamento de uma Falta de Página

Ao encontrar uma referência a uma página não presente é gerada uma page fault e o controlo é transferido para o sistema operativo:

- É consultada a tabela de páginas:
  - Referência inválida: aborta
  - Página não presente: continuar tratamento
- Escolhe uma frame vazia na memória física e verifica se a página existe na memória virtual
- Trás o conteúdo da página para a frame escolhida
- Atualiza a tabela de páginas com o endereço da frame e mudar o valor do bit $v$
- Reinicia o processo na instrução exata que causou a falha de página

### 2.2.2 Substituição de Páginas

Coloca-se então a questão de quantas frames se devem reservar para cada processo. O número máximo é a memória disponível, mas o número mínimo é dependente da arquitetura, já que, por exemplo, uma instrução pode estar em 2 páginas. No caso de não haver memória livre disponível, a escolha de uma nova frame para carregar a página em falta tem de passar por um esquema de substituição de páginas: escolher uma página em memória que não esteja a ser utilizada, enviá-la para disco (swap-out) e utilizar a frame assim libertada.

Trata-se de um processo complexo que implica:

- A utilização de um algoritmo de seleção e substituição de páginas não utilizadas
- O algoritmo tem de minimizar o número de falta de páginas no sistema
- Tem consequências críticas no desempenho do sistema em situações de sobre utilização de memória

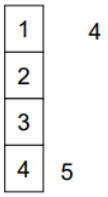
Pode-se praticar uma substituição global, em que a frame de substituição (vítima) é escolhida do conjunto de todas as frames (pode pertencer a outro processo) ou substituição local, em que a frame de substituição é escolhida do conjunto das frames do processo. Alguns algoritmos de substituição de páginas são:

#### Algoritmo Ótimo

Substituir a página que não vai ser usada pelo período mais longo. Usado para medir qualidade relativa dos algoritmos.

Acessos a memória: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Processo com 4 frames: 6 faltas de página

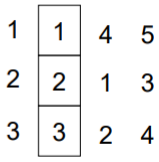



### Algoritmo FIFO

A página a substituir é a mais antiga. Para a contagem de faltas de páginas tem-se em conta que as frames estão vazias de início

Acessos a memória: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Processo com 3 frames: 9 faltas de página



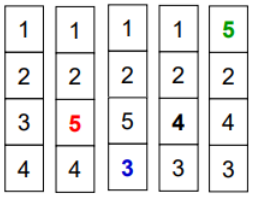

### Algoritmo LRU

Substituir a página que não é usada durante mais tempo.

- Concretizado com um contador incrementado a cada acesso
- Guardar para cada página o valor do contador do seu último acesso
- Quando for necessário substituir uma página escolhe aquela com o contador menor

Acessos a memória: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Processo com 4 frames: 8 faltas de página




Este algoritmo pode também ser implementado através de uma pilha de números de páginas:

- A cada referência, move página para o topo da pilha
- Se a pilha for uma lista duplamente encadeada não é necessário pesquisar, a página que não é usada á mais tempo está no fundo.

Uma última implementação é a second chance, feita através de uma lista circular de páginas com bit de referência. Itera-se pela lista de páginas:

- Se a próxima página tem reference bit a 0 é a vítima
- Caso contrário, o bit é colocado a 0
- Passa á página seguinte

# Capítulo 3

# Tópicos Gerais

## 3.1 Ficheiros

Um ficheiro é um tipo de dados abstrato que possui atributos, tais como nome, identificador, tipo, localização, tamanho, etc. É possível realizar operações sobre ficheiros, nomeadamente criar, remover, truncar, escrever, ler e reposicionar dentro do ficheiro. Ao abrir, o so procura o ficheiro na diretoria e move os seus atributos para a memória. Ao fechar os seus atributos em memória são gravados no disco e removidos da memória.

### 3.1.1 Métodos de acesso a ficheiros

- Acesso sequencial
  - Operações de read e write acedem e avançam o file pointer
- Acesso direto ou aleatório
  - Ficheiro constituído por registos lógicos de tamanho fixo
  - Estes registos são identificados de forma relativa em relação ao inicio do ficheiro
  - Lê-se o bloco $n$

Existe ainda o ficheiro índice, que é guardado em memória e contém apontadores para blocos que por sua vez contêm records. O nome do ficheiro índice é o nome do primeiro record de um bloco.

### 3.1.2 Diretórios

Diretórios são uma coleção de nós que contêm informação sobre ficheiros e, tal como estes, residem em disco. Possuem algumas operações, tais como a procura, criação, remoção e renomeação de ficheiros. Existem várias formas distintas de estruturar diretórios:

- Estrutura de um nível
  - Existe uma única diretoria para todos os ficheiros de todos os utilizadores

  - É inconveniente pois tem um espaço de nomes único e não permite agrupamento

- Estrutura de dois níveis
  - Uma diretoria por utilizador
  - Elimina alguns dos problemas anteriores e torna a pesquisa mais eficiente

- Estrutura em árvore
  - Introduz a noção de diretório corrente
  - Procura eficiente
  - Possibilidade de agrupar ficheiros
  - Espaço de nomes estruturado

- Grafos Acíclicos
  - Permite partilha de ficheiros e sub-diretórios
  - Aliasing: vários nomes para a mesma entidade
  - Obtido com o conceito de link
  - Cópia de ficheiro vs. vários nomes para o mesmo ficheiro

### 3.1.3 Mounting

Um sistema de de ficheiros precisa de ser montado para ser acedido. Para tal, é escolhido um mount point: um local da árvore de diretórios onde a raiz fica. Na operação de mount:

- Dá-se a fusão entre duas árvores
- Uma árvore torna-se sub-árvore da outra

### 3.1.4 Partilha de Ficheiros

A partilha é desejável em sistemas multi-utilizador. Para uma partilha segura é necessário implementar um esquema de proteção, em que a identificação de utilizadores usa:

- user IDs - permitem proteção por utilizador
- group IDs - permitem proteção por grupo de utilizadores

A partilha de ficheiros remotos pode ser feita de vários modos:

- Manualmente, através de ftp (anónimo ou autenticado)
- Automático: Sistema de ficheiros distribuído
- Semi: WWW (semelhante a ftp)
- Modelo cliente-servidor: Cliente podem fazer mount de sistemas de ficheiros remotos e o servidor pode servir vários clientes.

### 3.1.5 Proteção

O criador de um ficheiro deve controlar o que pode ser feito e por quem. Para isto, existem diversos tipos de acesso (leitura, execução...). Para tal usam-se mecanismos de controlo de acesso:

- ACL (access-control list)
  - Lista de utilizadores especifica quem pode aceder
  - Esta lista pode ser extensa e não conhecida à partida
  - Entrada que descreve diretório tem que ser de tamanho variável
- Versão condensada (usada em UNIX)
  - É associado um utilizador e um grupo de utilizadores ao ficheiro
  - Conjuntos de utilizadores são classificados como user, group, other.
  - São definidos três tipos de acesso: read, write, execute
  - Acesso é controlado em $3 \times 3 = 9$ bits

## 3.2 Sistema de Ficheiros

O sistema de ficheiros pode ser visto de duas perspetivas: na perspetiva do utilizador existem ficheiros e diretorias e na perspetiva do sistema existe uma associação entre ficheiros e diretorias ao espaço em disco.

É, portanto, necessário organizar o sistema de ficheiros em disco, geralmente em camadas. Alguns sistemas de ficheiros são, por exemplo: FAT, FAT32, ext2, ...

### 3.2.1 Implementação do Sistema de Ficheiros

Á semelhança dos processos, cada ficheiro tem a sua file control block (FCB), uma estrutura de dados com informação sobre o ficheiro correspondente (permissões, datas,...).

#### Operação Create

- Reserva um novo FCB
- Atualiza a diretoria com o novo FCB e nome de ficheiro

#### Operação Open

Esta operação recebe o nome do ficheiro como argumento. Verifica se o ficheiro já está aberto noutro processo:

- Acede à tabela global de ficheiros abertos (system-wide open-file table)
- Se sim cria entrada em tabela de ficheiros abertos do processo (per-process open-file table)

- Se não é feita a pesquisa do ficheiro na estrutura de diretórios + entrada na tabela global + entrada na tabela do processo

Para gerir um ficheiro aberto é necessário o apontador do ficheiro, o seu contador de aberturas, a sua localização em disco e direitos de acesso. Estas informações são mantidas em dois tipos de tabelas:

- Tabela por processo
  - File pointer
  - Permissões
  - Apontador para a tabela global
- Tabela global
  - Localização do ficheiro em disco
  - Dimensão do ficheiro
  - Contador do nº de vezes que o ficheiro está aberto

**Operações Read e Write**

- Open retorna apontador (descritor do ficheiro)
- Operações de acesso são efetuadas através deste apontador

**Operação Close**

- Removido da tabela de ficheiros abertos do processo
- Removido da tabela de ficheiros abertos global

### 3.2.2 Implementação de Diretórios

- Lista com nomes de ficheiros e apontadores para blocos simples
  - Procura de nome potencialmente demorada
  - Lista ordenada facilita procura, mas dificulta alterações
- Hash Table
  - Reduz o tempo de procura
  - Possibilidade de colisões
  - Problema: tamanho fixo da hash table e dependência da função de hash desse tamanho

### 3.2.3 Métodos de Reserva

Existem vários métodos para atribuir blocos do disco a ficheiros, sendo que normalmente um sistema de ficheiros usa apenas um método.

**Reserva Contígua**

- Cada ficheiro ocupa um conjunto de blocos contíguos
- Basta guardar para cada ficheiro o bloco inicial e o tamanho
- Permite fácil acesso direto
- Gestão de espaço livre
- Ficheiros não podem crescer

**Reserva de Lista Ligada**

- Cada ficheiro é uma lista encadeada de blocos
- Blocos podem estar espalhados pelo disco
- Basta guardar o nó do primeiro bloco pois cada bloco contém o número do próximo
- Sem acesso aleatório

Existe uma outra variante: FAT (file allocation table)

- É mantida uma tabela separada com uma entrada por bloco do disco
- Cada ficheiro é representado por uma lista ligada na FAT
- Pode ser feita cache da FAT para aumentar eficiência no acesso

**Reserva Indexada**

- Usa-se uma tabela de índices de blocos (todos os apontadores num só local)
- Permite acesso aleatório
- Sem fragmentação
- Acesso sempre indireto (tabela de índice)
- Tabela ocupa espaço

É também possível concretizar uma implementação multi-nível para suporte a ficheiros de maior dimensão. Desta forma usam-se índices multi-nível (bloco de índices de 1º nível aponta para blocos de índices de 2º nível que apontam para blocos do ficheiro).

### 3.2.4 Gestão de Espaço Livre

Pode ser feita a identificação dos blocos livres através de um vetor de bits:

- Bit a 0: livre
- Bit a 1: ocupado

É uma abordagem simples e eficiente desde que o vetor não seja guardado em disco.

Uma outra abordagem consiste na utilização de uma lista encadeada, que não ocupa espaço de forma inútil mas não é fácil encontrar blocos contíguos. Pode-se implementar uma variante com contador de blocos contíguos.

### 3.2.5 Recuperação

Dado que a informação está em memória e em disco, falhas no sistema podem levar a perdas de dados e incoerências.

Para manter a coerência o sistema operativo possuí formas de comparar a info das diretorias com info dos blocos de disco, que dependem dos algoritmos de reserva e gestão do espaço livre.

Podem ser usadas técnicas de sistemas transacionais, em que todas as alterações são registadas em log ou backups (completos ou incrementais).

## 3.3 Armazenamento

No nível inferior ao sistema de ficheiros temos o armazenamento. Alguns dispositivos de armazenamento são, por exemplo, a fita magnética, os discos rígidos ou os discos SSD.

### 3.3.1 Estrutura de Disco

As unidades de disco são endereçadas como um vetor de blocos (unidade de transferência). O vetor de blocos é projetado sequencialmente nos setores do disco. O bloco 0 inclui o primeiro sector da primeira pista do cilindro mais externo e a projeção continua em ordem na pista, no cilindro, e depois em direção ao interior do disco.

### 3.3.2 Ligação de Discos a Computadores

Localmente, o computador usa os barramentos de E/S para aceder às unidades de disco.

É também possível aceder ao armazenamento pela rede (network attached storage, NAS). No UNIX existe o network file system (NFS) que fornece uma implementação de remote procedure calls (RPC) entre o host e o NAS.

Existe ainda o conceito de storage area network (SAN), usado em ambientes

de grande armazenamento. Usa protocolos de armazenamento e de rede para permitir que vários hosts se conectem a storage array-flexible.

### 3.3.3 Gestão de Disco

A formatação é um processo de baixo nível. Pode ser física, na divisão das pistas em setores( nos discos magnéticos) ou lógica, aquando da gravação da estrutura do sistema de ficheiros (essencial para usar o disco). É possível criar partições de disco (partes virtuais dentro de uma unidade).

Também é possível organizar o disco em clusters (grupos de blocos) ou em formato raw disk, em que não existem estruturas de dados do sistema de ficheiros.

### 3.3.4 Estrutura de RAID

As estruturas de RAID usam a redundância para conseguir maior fiabilidade(contra falhas de discos). Verificam-se ainda melhorias de desempenho devido ao paralelismo.

- RAID 0 - Distribuição em faixas (stripping)
  - Grupo de discos forma uma unidade de armazenamento
- RAIDs que implementam redundância e fiabilidade
  - RAID 1 - Espalhamento (mirroring, shadowing), replicação dos dados em mais de um disco
  - RAID 1+0 - Striped mirrors ou RAID 0+1 - Mirrored stripes fornecem melhor performance e fiabilidade
  - RAID 4,5,6 - Paridade de discos tem menor redundância

RAID dentro de um storage array pode também falhar se o array falhar. Automaticamente a replicação de arrays entra em funcionamento.

## 3.4 Sistema de Entradas e Saídas

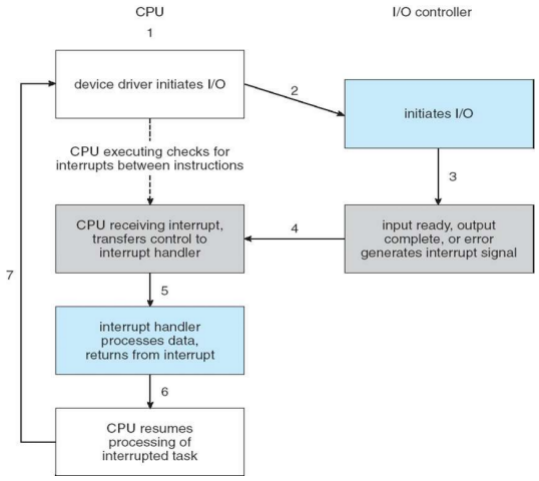
Existe uma grande variedade de sistemas de entrada e saída que usam conceitos em comum, tais como o porto, barramento (bus), cadeia, controlador, etc. Existem três tipos de bus na estrutura de barramento típica de um pc:

- Data bus - usada para transferir os dados entre os componentes do sistema. Composto por data lines, onde cada linha transfere um bit
- Address bus - especifica o endereço origem ou destino dos dados que estão a ser transmitidos no data bus
- Control bus - permite controlar sinais o que está a ser transferido e para quem, coordenando por emissão das operações a realizar

Existe uma fila de espera ativa em que o estado de um dispositivo E/S é verificado ciclicamente.

### 3.4.1 Interrupções

- Interrupção do CPU iniciada por um dispositivo
  - Algumas interrupções podem ser mascaradas (ignoradas)
  - Pode haver prioridade entre elas
- Interrupção do processador
  - Encontra endereço do tratador no vetor de interrupções
  - Desvia para tratador




### 3.4.2 Acesso Direto à Memória

O acesso direto direto à memória é usado para mover blocos de dados, deixando de lado o CPU na transferência entre dispositivo e memória. Necessita de um controlador de DMA.

### 3.4.3 Device Drivers

Dado que os dispositivos podem ser muito diferentes uns dos outros, o objetivo dos drivers é fornecer uma interface de E/S normalizada, ocultando diferenças entre os vários controladores.

### 3.4.4 Operações E/S

- Bloqueantes
  - Processo é suspenso até a operação de E/S ser realizada
- Não bloqueantes
  - Operação retorna a informação disponível
- Assíncronas
  - Processo continua e é avisado quando a operação de E/S terminar

### 3.4.5 Gestão de Erros e Proteção

Podem ocorrer erros de leitura em disco, dispositivos indisponíveis, de escrita, etc. Geralmente devolve-se um código de erro à aplicação, podendo haver um registo de erros. Um processo pode tentar causar erro no sistema (propositadamente ou não)

- Todas as instruções de E/S são privilegiadas
- Processos não têm acesso à memória dos dispositivos
- Todas as operações de E/S são feitas pelo SO

Existem mecanismos de proteção para evitar a realização de operações de E/S ilegais.

### 3.4.6 Estrutuas de Dados para E/S

O kernel mantém várias estruturas de informação de estado, tais como tabelas de ficheiros, filas de acesso a disco, buffers, etc. Alguns sistemas são orientados por objetos e outros usam troca de mensagens.

### 3.4.7 Processamento de um Pedido de E/S

Segue-se um exemplo de um pedido de leitura de um ficheiro:

- Determinar dispositivo que contém o ficheiro
- Traduzir de nome em índice de blocos
- Fisicamente ler dados do disco para um buffer
- Tornar bloco de dados lido disponível ao processo
- Retornar ao processo

## 3.5 Proteção

A proteção tem como objetivo garantir que cada objeto é acedido corretamente. Segue-se o principio do mínimo privilégio.

### 3.5.1 Domínios de Proteção

Um domínio pode ser um utilizador, processo, procedimento que possui um conjunto de permissões. Processos são executados num domínio de proteção que depende de várias identidades (ex. execução em modo utilizador ou modo administrador).

**Estrutura em Domínios**

- Direito de acesso:
  - <nome do objeto, conjunto de direitos>
- Conjunto de direitos - subconjunto de todas as operações que são permitidas sobre um objeto

Em POSIX existem dois tipos de domínios: utilizador e administrador. O domínio é definido pela identificação do utilizador (userid) e a troca de domínio é feita através do sistema de ficheiros. Existe um bit de escolha de identificação de utilizador (set-userid). Ao executar um ficheiro com set-userid o utilizador se torna no dono do ficheiro.

### 3.5.2 Matriz de Acessos

A proteção pode ser vista como uma matriz, em que as linhas representam os domínios e as colunas representam os objetos. $Acesso[i][j]$ é o conjunto de operações que o domínio $i$ pode fazer no objeto $j$.

| domain \ object | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

### 3.5.3 Controle de Acesso

O controle de acesso é baseado em papéis. Os utilizadores têm papéis e cada papel tem um conjunto de direitos, pelo que os utilizadores não têm permissões específicas. A gestão de utilizadores consiste em atribuir o papel correto a cada um.

Existem listas de controle de acesso (ACL), matrizes em forma de lista, para cada objeto. O sistema percorre a lista até decidir. Exemplo:
Ficheiro XPTO: João pode ler, Maria pode escrever, Paulo pode ler e escrever, Alunos (grupo) podem ler, Outros, não podem fazer nada.

## 3.6 Segurança

### 3.6.1 Segurança de Recursos

A segurança de recursos consiste em controlar o acesso a uma localização bem delimitada, o perímetro. Exige controlo em:

- Acesso a serviços
- Direção de utilização

- Utilizadores
- Comportamento

Permite negar a utilização de um objeto por uma pessoa.

### 3.6.2 Segurança de Comunicação

Para uma comunicação segura é necessário:

- Estabelecer um canal de comunicação seguro entre os intervenientes da comunicação
- Trocar informação de forma segura

### 3.6.3 Segurança de Informação

Para proteger a informação/serviços contra o acesso de leitura de intrusos deve-se:

- Limitar o acesso à informação/serviço apenas às pessoas autorizadas
- Proteger informações/serviços privados de intrusos

Para garantir que uma informação é genuína e está protegida contra a personificação de intrusos devem-se usar mecanismos de autenticação.

#### Encriptação

Podem-se usar chaves de encriptação de forma a proteger a informação. Chaves podem ser simétricas ou assimétricas.

É possível usar chaves de encriptação para assinar documentos.