universidade de aveiro
theoria poiesis praxis

I get the biggest enjoyment from the random and unexpected places. Linux on cellphones or refrigerators, just because it's so not what I envisioned it. Or on supercomputers.

Linus Torvalds

3 SISTEMAS DE OPERAÇÃO

Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte**. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Sistemas de Operação, tal como foi lecionada, no ano letivo de 2016/2017, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

mais informações em ruieduardofalopes.wix.com/apontamentos

Tendo estudado as disciplinas de Arquitetura de Computadores I (a2s1) e Arquitetura de Computadores II (a2s2) onde se abordaram noções básicas sobre arquitetura e protocolos de comunicação com dispositivos de entrada/saída, Programação I (a1s1), Programação II (a1s2) e Programação III (a2s1) onde se praticou a programação sobre a linguagem Java e outras disciplinas de sistemas digitais, onde se estudaram conhecimentos de estruturas de dados estáticas e dinâmicas mais comuns e a sua aplicação à construção de diferentes tipos de memória (RAMs, pilhas, filas e memórias associativas), chegou o momento de juntar tudo e abordar uma aplicação genérica de todos estes conteúdos numa disciplina de Sistemas de Operação (a3s1).

Estudando um sistema de operação é natural que se tenham de dominar as técnicas necessárias para compreendermos o mecanismo de multiprogramação e da sua organização geral, realizar tarefas administrativas simples de configuração e gestão, desenvolver pequenas aplicações que tiram partido de APIs fornecidas pelo modelo de máquina virtual, tendo em vista promover a robustez e a portabilidade de código e ser capaz de projetar aplicações concorrentes simples [1, pp. 4-5].

1. Introdução aos Sistemas de Operação

Um computador, tal como os que estamos habituados a usar, como já tivemos oportunidade de ver em Arquitetura de Computadores II (a2s2), é uma máquina que possui uma ou mais unidades de memória, um ou mais processadores e alguns periféricos de entrada/saída, como impressoras, ratos, teclados, monitores, entre outros... Para que todos estes componentes interajam entre si deverá haver um intermediário que consiga compreender a forma com que as diversas partes comunicam. Por esta razão os computadores estão equipados com um *software* denominado de **sistema operativo** (ou um sistema de operação). Um sistema de operação é assim um componente virtual cujo trabalho é fornecer um modelo de computador melhor, mais abstrato e simples aos *softwares* e que sabe como gerir todos os recursos previamente mencionados [2, p. 1]. Os sistemas operativos devem assim ser capazes de assistir como uma *interface* entre utilizador e computador, como um gestor de recursos computacionais e como um facilitador da interação entre as aplicações criadas e a larga variedade de máquinas que se constroem para utilização.

sistema operativo

Introdução ao sistema UNIX

Comecemos assim por analisar a primeira necessidade de um sistema operativo, o qual se baseia num determinado grau de interação humano-computador. Como podemos ver na Figura 1.1, um computador pode ser descrito num conjunto de camadas funcionais sob um sistema de hierarquia. No topo desta figura permanece o utilizador, que não se tem de preocupar com nada que resida nas camadas mais profundas da hierarquia - o que o utilizador vê é apenas um conjunto de aplicações. Já do lado de quem cria as aplicações não seria bom que este tivesse de controlar todo o espaço das instruções da máquina, de forma a criar o seu programa, dado que nesse caso teria uma tarefa de elevada complexidade. Para facilitar esta tarefa conjuntos de programas de sistema são fornecidos ao programador, como utilitários ou **bibliotecas**. Tais conjuntos já implementam, muito frequentemente, funções já usadas, assistentes na criação de outros programas, na gestão de recursos e no controlo de dispositivos periféricos de entrada/saída. Sem estas bibliotecas os programas teriam todos de ser executados no nível mais baixo possível, este, denominado de **kernel** (modo também denominado de **modo de supervisão**). Pelo contrário, os programas que normalmente executamos e cuja interação com o humano é o mais direta possível estão em **modo de utilizador**, sobre o qual apenas um subconjunto de instruções estão disponíveis. Em termos mais gerais todas as instruções que afetam o controlo da máquina ou dos periféricos de entrada/saída (I/O) são proibidas de serem executadas por programas em execução em modo de utilizador. Mais à frente iremos voltar às diferenças entre o modo de kernel e o modo de utilização - esta questão é crucial para a compreensão de como um sistema de operação funciona, como o UNIX.

bibliotecas

kernel, modo de supervisão

modo de utilizador

Na Figura 1.1 o sistema de operação que está a ser retratado é o UNIX. O **UNIX** é um sistema de operação criado por Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy e Joe Ossanna, em 1971 (tendo começado a ser desenvolvido em 1969), que ainda hoje é base para muitos sistemas operativos como o macOS da Apple® ou o Linux (clone de UNIX). Neste sistema operativo existe já uma biblioteca de programas bastante extensa que permitem que outras aplicações, de um nível mais alto, lhes acedam para a partilha de funções. Um exemplo bastante concreto será o compilador da linguagem de programação C - o comando `cc` (*c compiler*). Esta aplicação tem base em quatro programas presentes nas bibliotecas do sistema UNIX, entre as quais o `cpp` (pré-processador da linguagem C), o `comp` (compilador de C de duas passagens), o `as` (*assembler* da linguagem C), e o `ld` (*linker* para a linguagem C) [3, secção 1.2]. Se alguns destes conceitos não estão relembrados, aconselhamos a consultar os apontamentos das disciplinas de Arquitetura de Computadores I (a2s1) e Linguagens Formais e Autómatos (a2s2).

UNIX
- Dennis Ritchie
- Brian Kernighan
- Ken Thompson
- Douglas McIlroy
- Joe Ossana

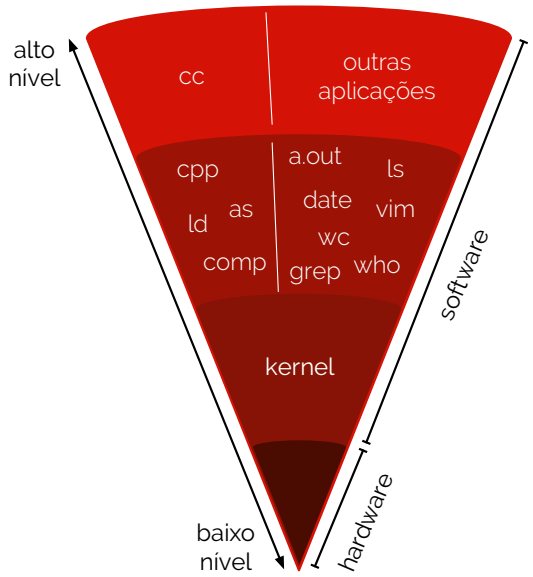


figura 1.1
arquitetura do sistema UNIX

Outros programas como é o caso do editor de texto `vim` usufruem da interação com funções e outros comandos específicos do kernel através de uma interface que este dispõe para os níveis superiores, a que se dá o nome de **system calls** (chamadas de sistema). Tais funções gerem todas as interações entre o nível de espaço de utilizador e espaço privilegiado (kernel). O modo de supervisão acaba assim por executar operações primitivas a pedido (por **interrupção**) de processos do modo de utilizador, de forma a poder suportar a interface anteriormente designada. Entre os vários serviços de que o modo privilegiado dispõe, este poderá controlar a execução de processos permitindo a sua criação, término ou suspensão (e comunicação entre processos), criação de tarefas designadas diretamente para o processador - o kernel é capaz de, enquanto o processador executa uma tarefa, suspender tal tarefa e agenda uma nova tarefa a ser executada pelo CPU, repondo a anterior quando esta termina. Também poderá alocar memória para um processo em particular, permitindo que o espaço de memória de um determinado processo seja partilhado com outro, sob circunstâncias muito específicas. Num caso mais prático, a *shell* lê o input de dados do utilizador por via de uma system call: o kernel, que se executa por meios do processo da consola, controla a operação do terminal e devolve os carateres inseridos pelo utilizador, especificando um conjunto de ações, que podem (ou não) requerer a invocação de novas system calls.

system calls

interrupção

Como já foi estudado na disciplina de Arquitetura de Computadores II (a2s2), o sistema UNIX permite que tanto os periféricos de entrada/saída como o relógio do sistema interrompam o processador central assincronamente. Numa interrupção desta génese, é papel do kernel preservar o contexto atual de execução (como um *snapshot*), determinando

a causa da interrupção e servindo-a. Após a interrupção, há então que restaurar a execução anterior, ação novamente garantida pelo kernel. Na outra disciplina também tivemos oportunidade de reparar que o hardware frequentemente dá prioridades diferentes aos diversos dispositivos a si conectados, de modo a realizarem interrupções: como vimos, quando o kernel recebe uma interrupção, este bloqueia aquelas cuja prioridade é menor, servindo primeiro as que possuem uma maior prioridade.

Quando uma interrupção é resultante de um pedido de informação com privilégios não atribuídos a quem pede ou de informação inexistente (entre outros casos possíveis) é lançada uma **exceção**. As exceções, por si, são diferentes das interrupções, dado que estas são provocadas por eventos externos aos processos: um processo P pode pedir para aceder ao endereço de memória 0x9F, mas se este não existir o erro será lançado após o pedido ter sido realizado, logo fora do contexto do processo que requeriu a informação [3, secção 1.5.1].

exceção

Interação com o utilizador (comandos na shell)

A consola (*shell*) do UNIX permite três tipos de comandos para execução: um ficheiro executável que contém código produzido numa linguagem de programação compilada, como a linguagem C; um ficheiro executável que contém um conjunto de instruções na linguagem bash (linguagem-base da consola); um comando interno bash, completo (sem necessidade de ficheiro).

A shell procura os comandos nela inseridos pelo utilizador numa dada sequência de diretórios, passível de ser alterada a pedido do utilizador. Usualmente, a consola executa os comandos de forma síncrona, esperando sempre que um comando termine a sua execução antes de avançar para o comando seguinte. Por exemplo, considerando sempre o símbolo `$` como o *prompt*, se executarmos o comando do Código 1.1, o sistema UNIX pesquisará no diretório `/bin` o comando `who` e irá imprimir uma lista de utilizadores que estão a partilhar a mesma máquina, estando ligados ao mesmo sistema.

código 1.1

```
$ who
sheldon console  Sep  9 19:49
sheldon ttys000  Sep 13 09:21
```

No entanto, se executássemos o mesmo comando com a pequena diferença de adicionar um *ampersand* no fim, passaríamos a executar o comando `who` de forma assíncrona, dizendo que o sistema executa o comando em **background**, ficando a shell à espera de um novo comando. Esta situação poder-se-á verificar no Código 1.2, onde embora possa não parecer, após a execução da linha de código `who &` a consola mostra-se pronta de imediato para um próximo comando, exibindo o *prompt* antes de imprimir o resultado da instrução `who`.

background

código 1.2

```
$ who &
$ sheldon console  Sep  9 19:49
sheldon ttys000  Sep 13 09:21
```

A filosofia do sistema UNIX é fornecer primitivas de sistemas de operação que permitam que os utilizadores escrevam pequenos programas modulares que possam ser usados como peças de construção de programas mais complexos, a uma escala maior. Uma destas primitivas está na sua capacidade de **redirecionamento de entrada e saída**. Em geral a entrada e a saída dos comandos está associada aos dispositivos *standard input* (usualmente referimos o teclado) e *standard output* (usualmente referimos o ecrã). Também associamos ao *standard error*, que geralmente imprimindo no ecrã, fornece todas as informações de partilha de erros. Assim sendo, o sistema UNIX tem em si implementados dois operadores básicos que permitem o redirecionamento dos dados entre entradas e saídas. Por exemplo, se quiséssemos guardar o conteúdo do diretório atual num ficheiro `dir.txt` poderíamos usar o operador `'>'` para redirecionar o que será imprimido no ecrã por `ls` para o ficheiro `dir.txt`, como podemos ver no Código 1.3.

redirecionamento de entrada e saída

```
$ ls > dir.txt
```
código 1.3

No entanto, se pretendermos adicionar mais texto ao fim do ficheiro `dir.txt` através de nova expressão de redirecionamento não podemos reutilizar o operador `'>'`, mas sim antes o operador `'>>'`, dado que o primeiro criaria um novo ficheiro, enquanto que este concatenaria a nova mensagem - veja-se o Código 1.4.

```
$ ls >> dir.txt
```
código 1.4

Para redirecionarmos a consola de erros para um ficheiro ou outra saída podemos usar o operador `'2>'`, como podemos ver em Código 1.5 onde tentamos ver o ficheiro `inexistent.txt`.

```
$ cat inexistent.txt 2> error.txt
```
código 1.5

Se verificarmos o ficheiro `error.txt` deparamo-nos com algo do género do Código 1.6.

```
$ cat error.txt
cat: inexistent.txt: No such file or directory
```
código 1.6

Outros operadores não menos importantes são o `'|'`, o `'2>&1'` e o `'>&2'` (ou `'1>&2'`). Por exemplo, o primeiro operador, vulgarmente designado de **pipe** (em português tubo) serve para desviar o *standard output* de um comando para o *standard input* do comando seguinte. Por exemplo, se executarmos o comando `cat /etc/passwd` podemos visualizar o conteúdo do ficheiro `passwd`. Se quisermos contar o número de linhas que tal ficheiro possui, podemos fazer um pipe com o comando `wc -l`, como podemos ver no Código 1.7.

pipe

```
$ cat /etc/passwd | wc -l
    17
```
código 1.7

Já os operadores `'2>&1'` e o `'>&2'` (ou `'1>&2'`) permitem desviar o *standard error* para o *standard output* e o *standard output* para o *standard error*, respetivamente [4, pp. 1-2].

Analogamente, em parte, a algumas expressões regulares que aprendemos em Linguagens Formais e Autómatos (a2s2), em bash, também podemos aplicar o símbolo `'*'` para designar todas as variações de carateres (ou nenhum carater) numa sequência destes. Por outro lado, aplicando o símbolo `'?'` podemos designar o mesmo com a diferença de haver, apenas e somente, um carater. Vejamos o seguinte caso, onde possuímos um diretório com os seguintes ficheiros: `a a1 a2 a3 a11 a22 a33 att b c c11 c12` (Código 1.8).

```
$ ls
a   a1  a11 a2  a22 a3  a33 att b   c   c11 c12
$ echo a*
a a1 a11 a2 a22 a3 a33 att
$ echo a?
a1 a2 a3
$ echo b*
b
$ echo b?
no matches found: b?
$ echo c*
c c11 c12
$ echo c?
no matches found: c?
```
código 1.8

Com os carateres `'['` e `']'` podemos delimitar um conjunto de possiblidades de escolha entre carateres. Por exemplo se fizermos o `echo` de `[ac]` teremos todos os ficheiros de nome `a` e `c` - note-se que se houver um diretório `ac` tal não é coberto por `[ac]`, dado que `[ac]` é diferente de simplesmente `ac`. Por outro lado, se tivermos `[a-c]` isto significa que aceita-se qualquer carater compreendido entre `a` e `c`, inclusive, no mesmo caso, resultando

em selecionar os ficheiros de seu nome `a`, `b` e `c`. Note-se novamente que ficheiros como `ab`, `ac`, `cb` e `abc` não são cobertos por `[a-c]`, dado que `[a-c]` não é equivalente a `abc`.

O comando `echo` também pode servir para mostrar o valor atual das várias variáveis instituídas e criadas na sessão atual da consola. Para tal podemos ver o valor de uma variável `variableA` através da indicação do símbolo `'$'` antes do nome da variável. Já para definir uma variável apenas se tem de atribuir com o operador `'='` um valor a uma variável. No Código 1.9 temos um exemplo de atribuição de valor e posterior verificação.

código 1.9

```
$ variableA=20.0
$ echo $variableA
20.0
```

nota!! note-se que para declarar uma variável não poderá existir espaços entre o nome da variável e o operador, tal como entre o operador e o valor. **nota**

Em bash uma forma de garantir a semântica pretendida para uma determinada situação, como o nome de uma variável (em comparação a uma mera sequência de carateres equivalente) podem-se usar chavetas, como podemos ver no Código 1.10.

código 1.10

```
$ echo ${variableA}variableA
20.0variableA
```

Os carateres `"`, `'` e `\` podem ser usados para retirar o significado especial de carateres, no entanto, se fizermos `echo "$variableA"` obtemos um resultado diferente que para `echo '$variableA'`, sendo que no primeiro caso teríamos imprimido o valor da variável e no segundo a sequência de carateres entre apóstrofes.

A criação de funções também é possível em bash, sendo que o corpo de uma é bastante análoga à de outras linguagens, embora não havendo argumentos explicitamente declarados na assinatura da função. No Código 1.11 temos uma função `listDirectory` que nos imprime detalhadamente o conteúdo do diretório atual, usando simplesmente o comando `ls -l`. Para executar esta função, depois apenas temos de escrever o nome da função, tomando-a como um novo comando para a sessão da consola atual.

código 1.11

```
$ listDirectory() {
  ls -l
}
$ listDirectory
total 0
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a1
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a11
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a2
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a22
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a3
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 a33
0 drwxr-xr-x  2 sheldon  staff  68 Sep 13 21:23 ac
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 att
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 b
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 c
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 c11
0 -rw-r--r--  1 sheldon  staff   0 Sep 13 21:15 c12
$ listDirectory | wc -l
    14
```

Para trabalhar com os argumentos apenas denominamos as expressões `$1`, `$2`, ..., `$*`, `$@` e `$#`, sendo `$1` o primeiro argumento (especificado logo a seguir ao comando que invoca a função), `$2` o segundo argumento (especificado logo a seguir ao primeiro argumento), ..., `$*` o conjunto de todos os argumentos, `$@` o array com os argumentos individualmente delimitados por aspas e `$#` o número de argumentos. Note-se também que indicar `$0` significa o nome do executável (antes do argumento primeiro), indicar `$-` significa que se pretende visualizar o estado atual das *flags* da sessão bash, `$$` indica o número atual do processo da consola, `$!` mostra o número do último processo a ir para background e, pelo contrário, `$?` indica o último valor de retorno do último comando que foi executado. Estes símbolos estão todos designados numa norma denominada de **POSIX** (acrónimo de *Portable Operative System Interface*). **POSIX**

O útlimo símbolo referido anteriormente é usualmente usado para verificar que valor é que um determinado programa que acaba de executar retornou ou para verificar o valor de um teste de uma condição através do comando `test`. Note-se que no teste de condições na bash interpreta-se como valor lógico verdadeiro o `'0'` e como valor lógico falso o `'1'`. Estas verificações podem ser usadas em blocos *if...then...else...fi*, como podemos ver no Código 1.12.

código 1.12

```
if test -f file
then
  echo "file exists"
else
  echo "file not exists"
fi
```

No Código 1.12 a validação com o comando `test` pode ser reduzida, sempre, por parênteses retos, ficando apenas `if [ -f file ]`. Se antes da validação usarmos o símbolo `'!'` isto fará com que a condição seja negada, dado que tal símbolo representa a negação. Os restantes operadores lógicos booleanos, tal como nas outras linguagens de programação (grande parte delas) são representados por `&&` e por `||` para o AND e OR, respetivamente.

Como também podemos reparar, em bash, grande parte dos blocos de código com funções de controlo de fluxo são delimitados pelo seu nome na forma normal e invertida, isto é, um bloco de controlo de fluxo *if* inicia com a palavra `if` e termina com o reverso da mesma palavra `fi`. Da mesma forma, se pretendermos criar uma estrutura de decisão múltipla por casos podemos criar um bloco tal que comece com `case` e termine com `esac`, como podemos ver no Código 1.13, designando os vários casos por sequências de decisão seguidas de um parênteses curvo e terminando com dois dois-pontos - o primeiro dois-pontos termina uma instrução e o segundo o caso concreto. Note-se também que por "sequências de decisão" pretendemos referir que uma linha de código definindo um caso com a seguinte forma `'8|10) ls -la;;'` significa que executa o comando `ls -la` tanto com o valor de caso 8 como com o valor de 10.

código 1.13

```
case variableA in
  0) echo "The variableA value is 0";;
  1) echo "The variableA value is 1";;
  2|3) echo "The variableA value is 2 or 3";;
  *) echo "The variableA value is not 0, 1, 2 or 3";;
esac
```

Para criar uma estrtura de repetição *for* há que iterar sobre uma lista. Para tal vejamos o Código 1.14 onde queremos que se acrescente um prefixo (neste caso `_a_`) ao nome de todos os ficheiros do diretório corrente, cujo nome começa com `a`.

código 1.14

```
prefix="_a_"
for f in a*
do
  echo "changing the name of \"$f\""
  mv $f $prefix$f
done
```

No caso do *for* o bloco de código já não é delimitado da mesma forma, mas antes pelo *for* e pela palavra *done*. A mesma coisa acontece para o *while* e para um *while* com condição negativa, o qual se designa por **until**. Se se pretender, numa lista de argumentos de uma função, passar para o argumento seguinte, dentro de um ciclo, podemos usar o comando **shift** para tal.

until

shift

Tal como qualquer outra linguagem de programação com a bash também é possível criar ficheiros executáveis, aos quais damos vulgarmente o nome de **script**. Estes scripts podem ser guardados sobre qualquer extensão, se bem que a correspondente à linguagem é a *.sh. Para executar um script bash devemos usar o comando `bash` seguido do nome do ficheiro. Se o ficheiro tiver privilégios para ser executado também poderá ser executado sob a forma `./ficheiroScript`. Caso este não possua privilégios para tal, estas poderão ser fornecidas através do comando `chmod +x ficheiroScript`.

script

Tendo já visto com algum detalhe algumas interações do UNIX para com o utilizador, através de comandos e símbolos especiais de execução, há que referir que as consolas, por serem meras aplicações, podem ser várias, todas elas *Bourne-derivative*, isto é, todas elas com derivação da bash [3]. Nos dias que correm a bash não é a consola mais usada, sendo tamém usada a Z-Shell (`zsh`) ou a C-Shell (`csh`) - existem muitas outras, cada uma com as suas derivações e diferenças da bash. Para executar um script sempre como bash poderá ser feita uma de duas alternativas, caso se esteja a utilizar uma *shell* diferente da bash: primeira alternativa é executar o comando `bash`, para abrir uma consola bash; outra alternativa é, no início dos scripts incluir a seguinte linha de código `#!/bin/bash` - linha conhecida como **shabang**.

⊙ Stephen Bourne

shabang

Tipos de sistemas de operação

Ao longo da história da computação vários **tipos** de sistemas de operação foram surgindo para tentar responder a várias necessidades dos diferentes utilizadores destes. O sistema de operação mais simples de todos, o sistema de **processamento em série** era simplesmente baseado na interação direta entre um conjunto de dispositivos de entrada, numa unidade de processamento lógico, o qual digitava informação num conjunto de dispositivos de saída. Este tipo de operação, representado na Figura 1.2, permitia que o utilizador tomasse o controlo de todo o sistema operacional, executado sucessivamente programas de sistema que lhe permitem o desenvolvimento e teste dos seus programas. Este processamento em série era também denominado de tipo em **batch simples**. Uma **batch** é um conjunto de trabalhos/tarefas a serem desenvolvidas. Assim sendo, este sistema recebia um conjunto de tarefas e executava-as, de acordo com o comando do utilizador. Tais comandos eram fornecidos através de dispositivos de entrada como cartões perfurados (ou fita), bandas magnéticas, recebendo as informações processadas pelos mesmos meios.

tipos

processamento em série

batch simples, batch

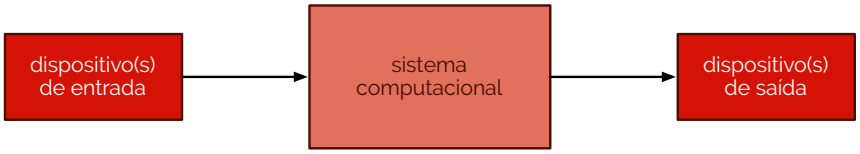


figura 1.2

Nesta primeira abordagem de sistema de operação pretendeu-se que a intervenção humana fosse reduzida, deixando assim um conjunto de tarefas para o processador interno da máquina com que o humano interage. Este conjunto de tarefas (batch, como anteriormente designámos) era fornecido através de uma linguagem de controlo de trabalhos (**job control language**), ao estilo do que podemos ver no Código 1.15.

job control language

```
$JOB #####, $FORTRAN #####, $RUN, $DATA #####, $END.
```

código 1.15

No entanto, por termos uma batch, não significa que tenhamos, obrigatoriamente, que executar um programa de cada vez, isto é, que só podemos executar uma tarefa B depois de uma tarefa A, executada primeiro, ter terminado. Esta foi a conclusão a que uns engenheiros chegaram tempos mais tarde que a solução de batch simples, criando os sistemas de **batch multiprogramada**. Com este tipo de solução conseguiu-se otimizar a ocupação do processador. Vejamos a Figura 1.3, onde mostramos a execução das tarefas A e B num processador onde se executa um sistema de operação do tipo batch simples, em comparação com o tipo batch multiprogramada.

batch multiprogramada

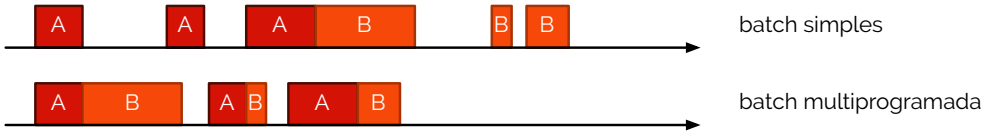


figura 1.3

Como podemos ver na Figura 1.3, o tipo de sistema de operação com batch multiprogramada permite que mal o processador esteja disponível para execução de uma parte de outra tarefa, executa-a.

Com o andar dos tempos outros paradigmas para a utilização dos sistemas computacionais foram surgindo, mais em particular, o paradigma de **time-sharing**. Nesta abordagem para a utilização dos computadores, ligados a uma única máquina central, era possível ligar vários terminais à mesma. Um pouco análogo ao que se pode fazer hoje-em-dia nos computadores com sistemas UNIX-*like*, onde fazendo a combinação de teclas Ctrl-Alt com uma tecla de função Fx, com $x = 1, \ldots, 8$, se pode mudar de `tty`.

time-sharing

No continuar da história da computação tambeém surgiram os sistemas de operação de **tempo real**, onde o computador era utilizado na monitorização e controlo *online* de processos físicos (assuntos que iremos abordar mais à frente) e os sistemas de operação **em rede**. Nestes últimos sistemas de operação o objetivo seria tirar partido das facilidades de interligação dos sistemas computacionais (ao nível de *hardware*) para estabelecer um conjunto de serviços comuns a toda uma comunidade.

tempo real

em rede

Por último, também obtivemos uma nova abordagem, que iremos estudar numa disciplina mais tarde - os sistemas de operação **distribuídos** (em Sistemas Distribuídos (a4s2)). Estes sistemas permitem tirar partido das facilidades de construção de sistemas computacionais com processadores múltiplos, ou de interligação de sistemas computacionais distintos, para estabelecer um ambiente integrado de interação com o(s) utilizador(es) que encara o sistema computacional paralelo como uma entidade única.

distribuídos

Multiprogramação e multiprocessamento

A discussão dos vários tipos de sistemas de operação leva-nos para um novo âmbito - o que é a multiprogramação e de que forma é que se distingue com o multiprocessamento? Ora, na transição dos sistemas de batch simples para os de batch multiprogramada vimos que era possível otimizar a ocupação do nosso processador e ir executando, embora às partes, duas tarefas em simultâneo. Isto apresenta-nos uma situação de **concorrência**, isto é, uma situação em que um sistema de computação poderá, eventualmente, criar a ilusão de aparentemente poder executar em simultâneo mais programas do que o número de processadores que realmente tem. Como já tivemos oportunidade de ver, isto exige que haja a capacidade de multiplexar, no tempo, a atribuição do(s) processador(es) para os diferentes programas presentes. Quando isto acontece dizemos que o sistema operativo em causa tem caraterísticas de **multiprogramação**.

concorrência

multiprogramação

Por outro lado, quando referimos **multiprocessamento** demos ser capazes de processar várias tarefas em simultâneo, e não concorrer para um mesmo recurso. Neste caso referimos o **paralelismo**, como a capacidade de poder executar dois ou mais programas em simultâneo, exigindo que o sistema computacional seja formado por mais do que um processador (um por cada programa que se pretende executar).

multiprocessamento

paralelismo

2. Gestão do Processador

Tendo já referido o contexto dos sistemas de operação e os assuntos a tratar nesta disciplina, então já temos todo o conjunto de conhecimentos necessários para avançarmos para o cerne do processamento interno de um computador - o **CPU**.

CPU

Execuções num processador

Sempre que falamos num processador e nas suas lógicas inerentes referimos o termo **execução**. Mas referimo-nos à execução de quê? Ora, em disciplinas anteriores estudámos sempre a execução de **programas**. Um programa é um conjunto de instruções que descreve a realização de uma determinada tarefa por um computador, quase como uma receita de culinária, que segue um ou mais algoritmos. Este programa, quando está a ser executado tem uma instância de execução a si associada. Consideremos um exemplo de um

execução

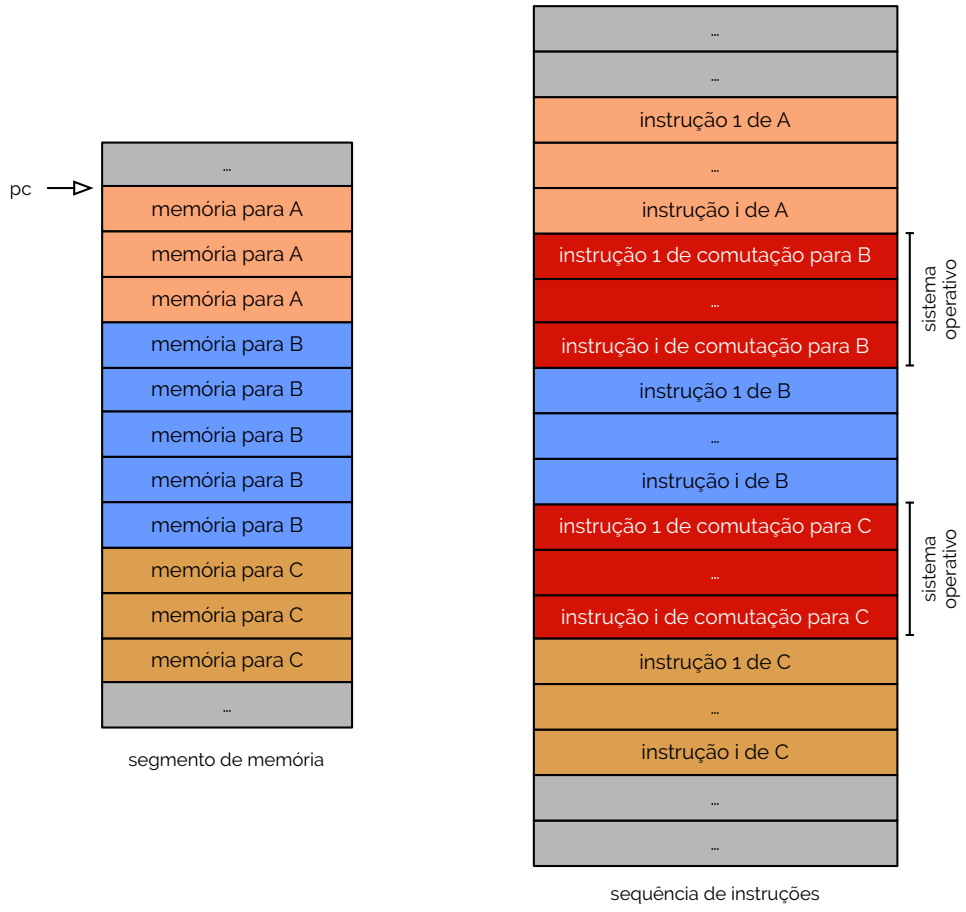
programas

pasteleiro que está a fazer um belíssimo pastel - o programa especifica o que o pasteleiro está a fazer, mas o pasteleiro... quem é que designa o que o pasteleiro faz? Ora, nesta analogia, o pasteleiro toma o valor de um **processo**, isto é, a execução de um programa. Esta execução é assim representada por um processo, que contém sempre um código do programa que está a executar e o valor atual de todas as variáveis que lhe estão associadas (um determinado espaço de endereçamento), o valor atual de todos os registos internos do processador, os dados que estão a ser transferidos dos dispositivos de entrada e para os dispositivos de saída e o seu estado de execução [5, p. 4].

processo

Sendo que cada programa é executado no contexto de um processo, de forma a que um mesmo processador possa ter vários processos em execução é importante que o sistema de operação saiba comutar entre eles, lendo diferentes regiões do segmento de memória para completar as instruções. Assim sendo, e tal como vimos em Arquitetura de Computadores I (a2s1) o registo `program counter` é responsável por indicar qual o endereço da próxima instrução a ser lida pelo processador. Lida tal instrução, o processador irá iniciar um determinado processo de execução, no fim, comutando, por ordem do sistema de operação, para uma nova região do segmento de memória de instrução onde se encontram as instruções para o processo de execução seguinte. Este processo poderá ser interpretado segundo a Figura 2.1.

figura 2.1



A ideia da multiprogramação (concorrência) torna muito complexa a perceção de diferentes atividades que estão em curso num processador. Uma forma de simplificar a sua perceção é ilustrando processadores virtuais, isto é, um processador por cada processo que concorrentemente coexiste, admitindo que os processos associados são executados em paralelo através da ativação ou desativação dos processadores respetivos. Consideremos, para tal, que a execução dos processos não é afetada pelo instante ou local no código onde ocorre a comutação e que não são impostas quaisquer restrições relativamente aos tempos de execução, totais ou parciais, dos processos. Sendo a comutação dos processos simulada pela ativação/desativação dos processadores virtuais e controlada pelo seu estado, é con-

veniente perceber que num sistema monoprocessador (real) o número de processadores ativos terá de ser, obrigatoriamente, no máximo um, enquanto que num sistema multiprocessador (real) o número de processadores terá de ser, no máximo, uma quantidade igual ao número de processadores existentes.

Para designarmos os diferentes comportamentos dos processadores é então, mais conveniente, detalharmos um diagrama de estados para o mesmo, sendo que um determinado processo terá um determinado **estado** dependendo da situação. Nesta primeira fase iremos considerar que um processo terá, apenas, três estados, sendo eles o de **em execução** (*run*) - onde este detém a posse do processador e está em execução -, o **pronto-para-execução** (*ready-to-run*) - quando aguarda a atribuição do processador para começar ou continuar a sua execução - e o **bloqueado** (*blocked*) - quando está impedido de continuar até que um acontecimento externo ocorra, como um acesso a um recurso, conclusão de uma operação de entrada/saída, entre outros...

estado
em execução
pronto-para-execução
bloqueado

No detalhe de uma máquina de estados é importante também referir as transições, as quais, neste caso, resultam normalmente de uma intervenção externa, podendo ser também despoletadas pelo próprio processo. Assim, quando um processo se encontra pronto a ser executado, em fila de espera, e se for submetido a **envio** (*dispatch*), então será selecionado para execução. Já na fase de execução, uma de duas alternativas poderão acontecer: se se despoletar um sinal de **tempo excedido** (*timer-run-out*), então o processo volta para a fila de espera no estado pronto-a-executar; por outro lado, se houver um sinal de **dormir** (*sleep*), então este terá de transitar para o estado bloqueado. Neste último estado, o processo, para avançar, só terá de receber um sinal de **acordar** (*wake up*), de forma a poder transitar para o pronto-a-executar. Na Figura 2.2 podemos ver a máquina de estados devidamente representada.

envio
tempo excedido
dormir
acordar

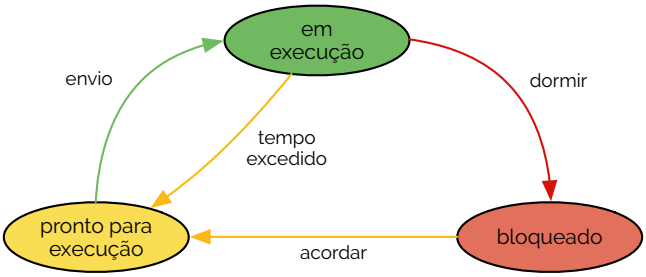


figura 2.2

Note-se que a parte do sistema de operação que lida com as várias transições da máquina de estados representada na Figura 2.2 é o **escalonador** (em inglês vulgarmente denominada de *scheduler*) e constitui parte integrante do seu núcleo central, o kernel.

escalonador

A memória principal, por muito grande que seja, é necessariamente finita, como já sabemos. Num ambiente multiprogramado, como o que pretendemos detalhar, terá de ser muito elevada, pelo que acaba por se tornar num fator limitativo ao seu crescimento. Para aliviar esta situação geralmente opta-se por criar uma extensão à memória de massa principal denominada de **área de swapping** (área de troca, em português). Esta região funciona como uma área secundária de armazenamento, onde se liberta o espaço em memória principal que se encontra reservado para processos que, de outro modo, não poderiam existir, potenciando-se, portanto, um aumento da taxa de utilização do processador. Por esta mesma razão, teremos de criar dois estados novos, associados à transferência do espaço de endereçamento do processo para a área de swapping: criamos assim o estado **suspenso e pronto** (*suspended-ready*) e o **suspenso e bloquado** (*suspended-blocked*). Assim sendo, também temos de incluir mais um conjunto de transições à nossa máquina de estados. Se um determinado processo estiver em fila de espera para ser atendida a sua execução, este poderá cair no estado suspenso e pronto se se ativar o sinal de **suspender** - suspende-se o processo, fazendo a transferência do seu espaço de endereçamento para a área de swapping, dizendo-se que o processo é **swapped out** -, voltando ao estado pronto para execução

área de swapping
suspenso e pronto, suspenso e bloqueado
suspender
swapped out

se e só se receber o sinal **continuar** (*resume*) - sinal com o qual o processo terá o seu espaço de endereçamento transferido da área de swapping para a memória principal, dizendo-se que o processo é **swapped in**. A mesma coisa acontece com os processos que possam transitar para o estado suspenso e bloqueado. Neste caso, no entanto, se o processo se encontrar no estado suspenso e bloqueado e receber um sinal de **evento concluído**, analogamente ao que aconteceria se estivesse no estado bloqueado e recebesse um sinal para acordar, este transita para o estado suspenso e pronto. Note-se que estas transições ocorrem num nível diferente das anteriores, sendo que as primeiras, da Figura 2.2 ocorrem ao nível de um escalonador de baixo nível (que gere o processador) e este atua ao nível de um escalonador de médio nível (que gere a memória principal), como podemos ver na Figura 2.3.

continuar

swapped in

evento concluído

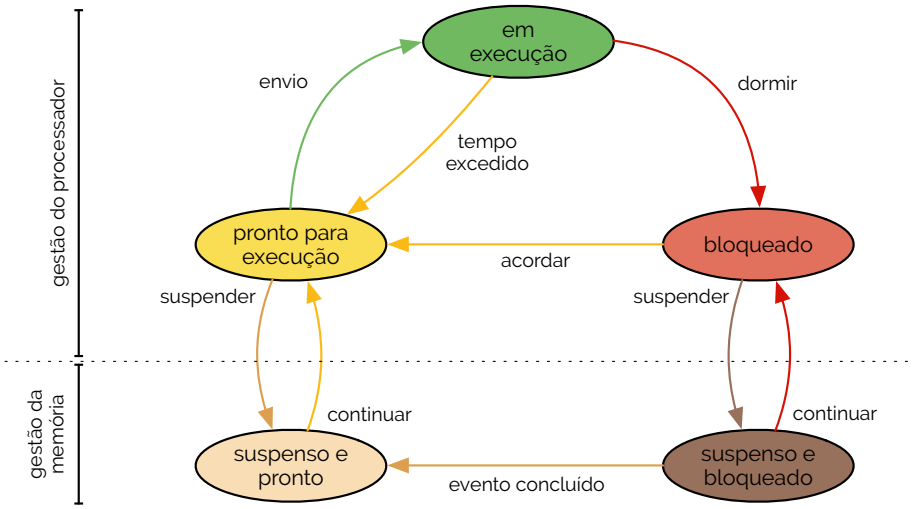


figura 2.3

Até agora temos considerado duas situações - os processos não são criados (já existem por si) e são eternos. Tal cenário não é real[1], sendo que um processo é uma instância de um programa que está a ser executado. Assim sendo, devemos ser capazes de criar e dar por terminados os processos. Para tal há que criar dois novos estados, entre os quais o estado **criado** (*created*) - onde se atribui um espaço de endereçamento e a inicialização das estruturas de dados destinadas a gerir o processo, operações que têm que ser realizadas antes que a execução possa ter lugar - e o estado **terminado** (*terminated*) - onde se faz a manutenção temporária, da informação relativa à história da sua execução, podendo vir a ser recolhida por programa específicos relacionados com a contabilização do tempo de uso do processador e de outros recursos do sistema computacional. Novamente, para adicionar tais estados há que adicionar e detalhar também novas transições. Assim sendo, quando um processo é criado é lançado o sinal para **ativar** o mesmo - isto faz com que o processo seja lançado para o estado de pronto para executar ou, caso não haja espaço em memória principal (isto é, na fila de espera), para o estado suspenso e pronto. Uma vez em execução, o processo, quando terminar a sua tarefa, ativa o sinal **sair** (*exit*), o qual o fará transitar para o estado terminado. No entanto não ficou concluída a máquina de estados. Isto acontece porque em quaisquer um dos estados é permitido que um determinado processo aborte a sua possível execução, lançando um sinal de **abortar**, levando-o, claro está, para o estado de terminado.

criado

terminado

ativar

sair

abortar

Sendo estas alterações feitas sobre a nossa máquina de estados da Figura 2.3 e estando elas num nível mais alto, onde se gere o ambiente de multiprogramação, é importante que se denote uma nova camada de escalonamento, como podemos ver na Figura 2.4, onde criamos uma nova camada abaixo, que representa o maior nível de abstração para o escalonamento dos processos.

1 Existem processos cujo tempo de vida é eterno enquanto o sistema de está ativo, mas tal depende do sistema operativo em causa e não é um caso frequente.

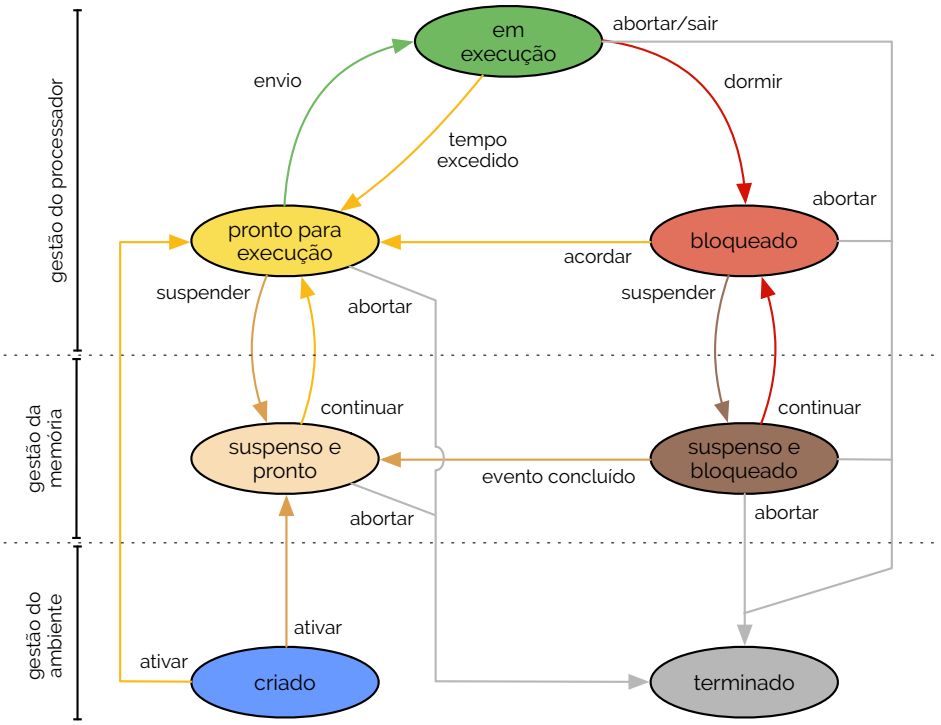


figura 2.4

No sistema UNIX, em particular, os processos são geridos de forma muito semelhante à que fomos demonstrando, na teoria. Na Figura 2.5 podemos ver a sua máquina de estados, para o sistema UNIX.

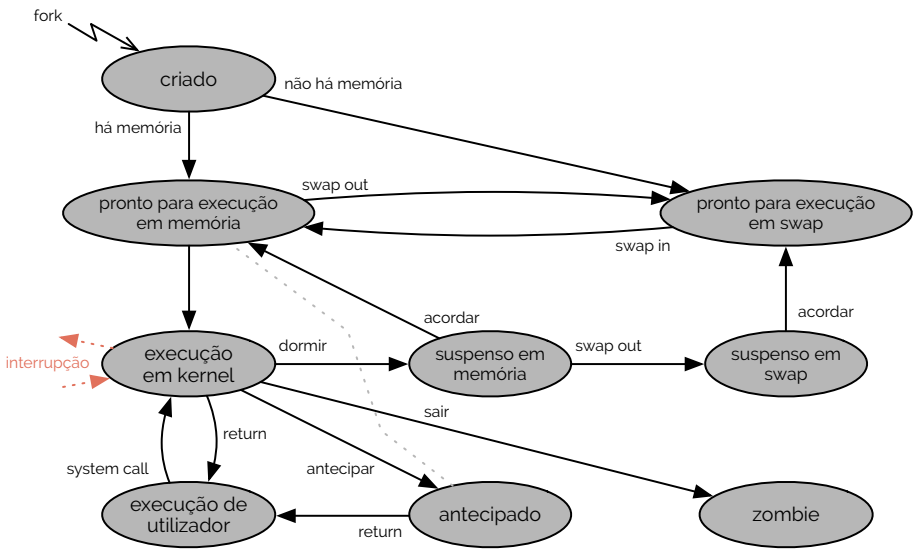


figura 2.5

Como podemos ver na Figura 2.5, o sistema UNIX considera dois estados como os de execução: o estado de **execução em kernel**, associado ao nível do supervisor, e o estado de **execução de utilizador**, associado ao nível do utilizador. Outra grande diferença está no estado pronto para execução estar dividido entre dois estados: o estado pronto para execução em memória e o estado antecipado, embora eles tenham muitos aspetos em comum, o que se pretende exibir pelo traço interrompido que os une.

execução em kernel
execução de utilizador

Em termos comportamentais, o que acontece é que sempre que um processo utilizador sai do nível supervisor, o escalonador tem a possibilidade de calendarizar para execução um processo de prioridade mais alta, fazendo então transitar o processo atual para o estado antecipado [5, p. 19]. Em termos mais práticos os processos nos estados pronto para

execução em memória e antecipado são colocados nas mesmas filas de espera, daí ter valor semelhante. De facto, é desta forma que é tratada a situação de esgotamento da janela de execução, estando a transição tempo excedido incluída no estado antecipado.

Mais do que a criação dos processos, os vários processos que são criados e executados numa só máquina constituem possibilidades de hierarquia, sendo que se um processo B é criado por um processo A, então dizemos que o processo A é **pai** do processo B e que B é **filho** de A. Vejamos assim a Figura 2.6 onde demonstramos a criação de um processo em UNIX.

pai
filho

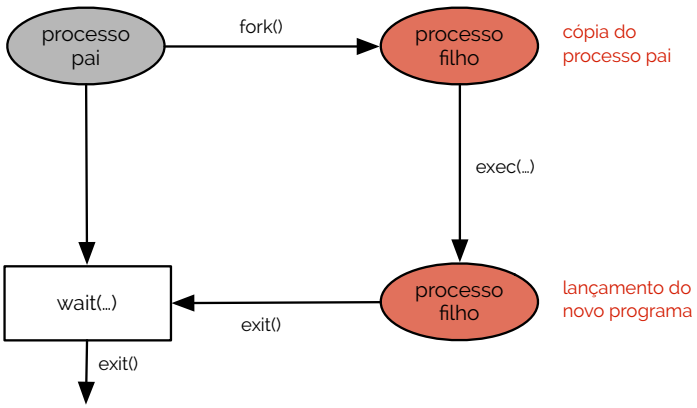


figura 2.6

Analisando a Figura 2.6 podemos ver que para criar um novo processo, filho de outro, usamos a chamada de sistema **fork** (designação que também se usa na Figura 2.5). Esta chamada de sistema permite que seja feita uma cópia do processo pai, num processo filho. De seguida, e para tornar este processo diferente do processo pai, há que executar as rotinas que se pretendiam e justificaram a criação de um novo processo, com a chamada de sistema `exec`, completada com os devidos argumentos para execução[2]. Em termos de código, para criar algo semelhante ao visto na Figura 2.6 podemos, em C, executar o Código 2.1.

fork

código 2.1

```
#include <sys/types.h>
#include <sys/wait.h>

void main(int argc, char* argv[]) {
  pid_t process_id = fork();
  if (process_id < 0) {
    perror("Erro na duplicação do processo");
    exit(EXIT_FAILURE);
  }
  if (process_id != 0) {
    if (wait(&status) != process_id) {
      perror("Erro na espera pelo processo filho");
      exit(EXIT_FAILURE);
    }
    printf("O meu filho, com id %d, já terminou.\n", process_id);
    if (WIFEXITED(status)) {
      printf("O seu estado de saída foi %d.\n, WIFEXITED(status));
    }
    printf("O meu filho é %d.\n", getpid());
  } else {
    perror("Erro no lançamento da aplicação");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

Espaço de endereçamento e tabela de controlo de um processo

Recordando alguns dos conhecimentos de Linguagens Formais e Autómatos (a2s2) temos que na criação de um processo é gerado um espaço específico para o mesmo em memória, esse, a que damos o nome de **espaço de endereçamento**. Este espaço, tendo a

espaço de endereçamento

2 Para ver os manuais das várias chamadas ao sistema (syscalls) podemos abrir um terminal e nele escrever "man 2 <nome-da-syscall>". Os manuais das syscalls estão no capítulo 2, daí "man 2".

forma representada na Figura 2.7, é preenchido por fases, dependendo das chamadas ao sistema.

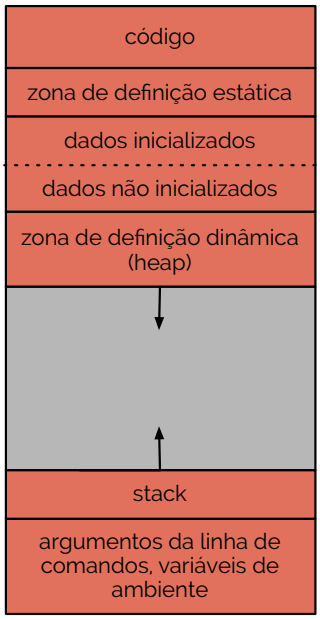


figura 2.7

As áreas do código do programa, zona de definição estática, dados inicializados e não inicializados são carregados diretamente do ficheiro executável aquando da execução da chamada ao sistema `exec(…)`, sendo que os dados não inicializados são colocados a zero por esta. A zona de definição dinâmica é feita através das várias funções derivadas das instruções de `malloc(…)`, `calloc(…)` e `realloc(…)`. Para libertar o vário espaço que fora ocupado com estas funções, podemos usar a instrução `free(…)`. Em termos da *stack* podemos alocar espaço nela através da instrução `alloca(…)`.

O facto de estarmos a trabalhar sobre uma lógica de multiprogramação faz com que tenhamos que gerir imensa informação acerca dos vários processos que se encontram em execução nas nossas máquinas. Para resolver esse problema usam-se as **tabelas de controlo de processos** (em inglês, *Process Control Table* - PCT). Esta tabela é assim intensivamente usada pelo escalonador para fazer a gestão do processador e de outros recursos do sistema computacional.

tabelas de controlo de processos

Cada índice desta tabela, devidamente identificando o número do processo em causa (`pid`) contém um conjunto de identificadores (quer do processo, do seu pai e do utilizador a que o processo pertence), a caraterização do espaço de endereçamento - definindo a sua localização em memória principal ou na memória de swapping, de acordo com o tipo de organização de memória estabelecido -, o contexto do processador - onde se alojam todos os valores dos registos internos do processador no momento em que se deu a comutação do processo (tal como referimos em Arquitetura de Computadores II (a2s2)) -, o contexto dos dispositivos de entrada/saída - onde se aloja informações acerca dos canais de comunicação e *buffers* associados - e informação acerca do estado e de escalonamento - onde se aloja o estado do processo, tal como referimos na secção anterior, e outros itens que possam estar associados. Na Figura 2.8 podemos ver uma representação da PCT, onde N é o número máximo de processos.

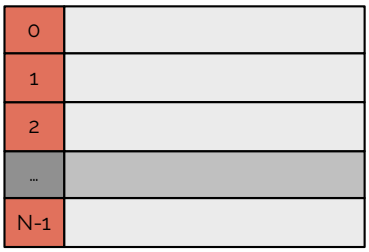


figura 2.8

Comutação de processos

Na Figura 2.1 representámos, de forma muito simples e meramente simbólica, a **comutação de processos**. Esta ação é importante para que possa ser suspenso ou terminado um processo e passar a execução para outro.

comutação de processos

Como já referimos também, os processadores atuais têm dois níveis de funcionamento: o nível de supervisor, onde se tem acesso ao conjunto de instruções completo, e o nível de utilizador, que constitui o modo normal de funcionamento. A passagem entre estes dois níveis é denotada num *bit* do `program status word`. Por razões de segurança, tal *bit* só poderá ser alterado, tal como a própria passagem do nível de utilizador para o de supervisor através de uma **exceção**, tal como já estudámos em Arquitetura de Computadores II (a2s2). Uma exceção é assim algo que interrompe o decurso normal da execução de um programa, como uma **interrupção** gerada por um dispositivo externo, a excução de uma instrução ilegal (inexistente, por exemplo) ou que conduz a um erro, como dividir por zero. Uma exceção também pode ser causada por uma instrução **trap** (em português ratoeira, tentando definir uma interrupção por software).

exceção

interrupção

trap

Para que o sistema permaneça em modo privilegiado, quando não é um dispositivo que provoca interrupções ao sistema, as chamadas ao sistema, todas elas, são implementadas através de instruções trap. Este tipo de abordagem terá assim de ser resolvido por rotinas de serviço às exceções. Esta rotina de serviço às exceções, geralmente, consiste na salvaguarda dos endereços presentes no `program counter` e do processo atual em execução `PSW` no *stack* do sistema. De seguida, após carregar o endereço da rotina de serviço no `program counter` deve haver a salvaguarda de todas as variáveis presentes nos registos internos do processador que vão ser usados, ocorrendo o processamento da exceção. Após o processamento todos os valores são novamente restaurados.

No nosso sistema em particular, as exceções serão tratadas de forma diferente, isto é, mais específica, havendo uma inicial salvagarda da caraterização do espaço de endereçamento atual e dos contextos do processador e de módulos I/O na entrada da PCT referente ao processo em execução. De seguida faz-se a atualização do estado do processo e de outra informação que esteja associada na entrada da tabela de controlo do processo, havendo a seleção do próximo processo para execução da fila de espera de processos prontos para execução, usando um algoritmo de calendarização (escalonamento). Há assim a colocação no estado de execução do processo agendado para execução e modificação de informação associada por atualização da entrada da PCT referente ao processo. Por fim, há o restauro de todas as informações de caraterização do espaço de endereçamento atual e dos contextos do processador e dos módulos I/O por transferência da entrada da tabela de controlo dos processos referente ao processo em causa.

Note-se que nestes casos a *stack* a que nos referimos é uma segunda *stack* que dá pelo nome de *stack* do sistema - não confundir com a *stack* existente no espaço de endereçamento de cada processo -, sobre a qual se guardam o `PC` e o `PSW`.

Políticas de escalonamento (scheduling)

Olhando novamente para a máquina de estados de um processo em UNIX da Figura 2.5 temos que podemos aplicar uma de duas **políticas de escalonamento** aos processos. Mais especificamente, podemos ter políticas de escalonamento com antecipação e sem antecipação. Numa política com antecipação o processador pode ser retirado ao processo que o detém por esgotamento do intervalo de tempo de execuão que lhe foi atribuído, por necessidade de execução de um processo de prioridade mais elevada. Por outro lado, numa política de escalonamento sem antecipação, após a atribuição do processador a um dado processo, este mantém-no na sua posse até bloquear ou terminar, não existindo a transição de tempo esgotado e sendo caraterístico dos sistemas de operação do tipo batch, para garantir que as suas tarefas são terminadas.

políticas de escalonamento

Não é fácil, no entanto, saber designar políticas para os vários tipos de sistemas de operação que existem e são aplicáveis a várias máquinas. Para tal criou-se um conjunto de

critérios que facilitam a sua escolha. Entre os vários critérios temos: a **justiça**, definida como o direito que todo o processo tem em relação a uma fração de tempo no processador; a **previsibilidade**, sendo que o tempo de execução de um processo deve ser razoavelmente constante e independente da sobrecarga pontual a que o sistema computacional possa estar sujeito; o **throughput**, onde se deve procurar maximizar o número de processos terminados por unidade de tempo; o **tempo de resposta**, onde se deve procurar, por outro lado, minimizar o tempo de resposta às solicitações feiras pelos processos interativos; o **tempo de turnaround**, onde se deve procurar minimizar o tempo de espera pela conclusão de um trabalho no caso de utilizadores num sistema batch; as **deadlines**, devendo-se procurar garantir o máximo de cumprimento possível das metas temporais impostas pelos processos em execução; e a **eficiência**, devendo-se procurar manter o processador o mais possível ocupado com a execução de processos dos utilizadores [5, p. 38]. Estes critérios, por si, também podemser enquadrados em duas perspetivas diferentes: numa perspetiva **sistémica**, onde os critérios são designados pelo utilizador e pelo sistema, isto é, são designados pelo comportamento do sistema de operação em termos de processos e utilizadores e com o uso dos recursos do sistema; numa perspetiva **comportamental**, olhando para o desempenho, entre outras referências...

critérios, justiça

previsibilidade

throughput
tempo de resposta

tempo de turnaround
deadlines

eficiência

sistémica

comportamental

Em termos de justiça, por exemplo, pretendemos atribuir o tempo mais justo de processador a cada processo. Para tal acontecer todos os processos são colocados em pé de igualdade e são servidos por ordem de chegada, sendo que em políticas de escalonamento sem antecipação é vulgarmente designada por **FCFS** (*First-Come, First-Serve*), em português, "quem chega primeiro, é servido primeiro". Por outro lado, em políticas de escalonamento com antecipação, usa-se **round-robin**. Estas duas designações, embora sejam ambas fáceis de implementar, existem para privilegiar os processos mais intensivos a nível de CPU em relação aos a nível de módulos de I/O.

FCFS

round-robin

Em muitas situações, considerar todos os processos como iguais pode não ser o procedimento mais adequado. A minimização do tempo de resposta exige, por exemplo, que seja dada preferência a processos intensivos a nível de módulos de entrada/saída sobre processos intensivos a nível de CPU, na calendarização para execução. Os processos podem assim, ser agrupados em níveis de prioridade distinta no que respeita ao acesso ao processador. Assim, aquando da selação do próximo processo a ser executado, o primeiro critério a ser seguido é o grau de prioridade relativa. Processos de prioridade maia baixa só serão selecionados quando na lista de espera do estado pronto para execução não existirem processos de prioridade mais elevada. Para definirmos as prioridades, podemos designá-las como **estáticas**, sobre as quais o método de definição é determinístico, isto é, sempre igual, ou como **dinâmicas**, sobre as quais o métodos de definição depende da história passada da execução do processo.

estáticas
dinâmicas

Em termos de prioridades estáticas [6] temos que, em múltiplos sistemas, a cada processo é designada uma determinada prioridade pelo escalonador, pelo que irá escolher sempre um elemento de prioridade mais alta sobre um de prioridade mais baixa. A Figura 2.9 mostra o uso de prioridades, sendo que o diagrama de fila está simplificado (ignoramos a existência de múltiplas filas bloqueadas e de estados de suspenção. Ao invés de termos, então, uma única fila de espera de processos prontos, fornecemos um conjunto de filas, numa ordem descendente em termos de prioridade: FP0, FP1, ..., FPn, onde a prioridade de FP0 é maior que a probabilidade de FP1[3]. Quando uma seleção de escalonamento está para ser concretizada, este irá começar no elemento da fila de maior prioridade (FP0). Se houver um ou mais processos na fila, um será selecionado através de alguma outra política de escalonamento. No caso de FP0 estar vazio, então FP1 será examinado, e por aí em diante.

Esta disciplina de escalonamento não é, de todo, a mais justa, dado que existe um risco claro da ocorrência de adiamento indefinido no agendamento para execução dos pro-

3 Nos sistemas operativos UNIX (e em muitos outros) prioridades com número mais alto são consideradas mais prioritárias que com números menores. Até prova em contrário, seguiremos sempre essa convenção, sendo que em sistemas como Microsoft Windows acontece precisamente o contrário: elementos com números de prioridade mais baixos têm mais prioridade que os com números mais altos.

cessos de prioridade mais baixa. No entanto, note-se que este é o método preferencial dos sistemas de tempo real.

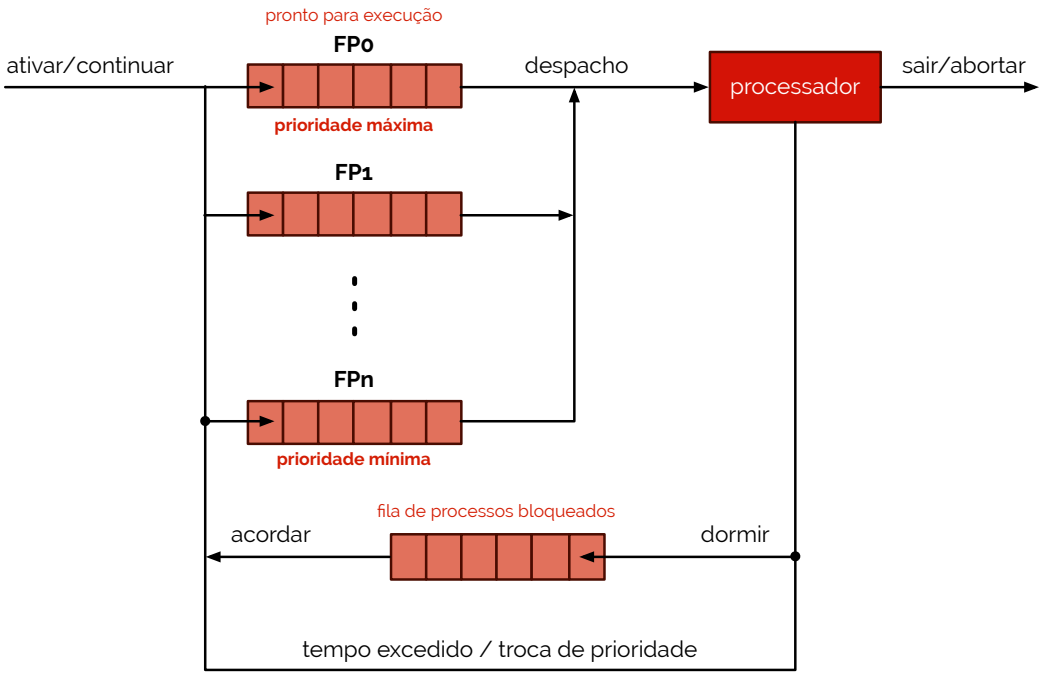


figura 2.9

Aquando da criação de uma política de escalonamento por prioridades estáticas, é atribuído a um processo um determinado nível de prioridade. Tal número é decrementado sempre que o processo sai do modo de execução por esgotamento da janela de execução (transição *tempo excedido / troca de prioridade* da Figura 2.9) e incrementado sempre que se bloqueia antes de esgotar a janela (transição *dormir*, da Figura 2.9). Este valor, no entanto, é sempre reposto quando atinge o seu valor mínimo.

Um método alternativo de privilegiar os processos interativos consiste na definição de **classes de prioridade** com um caráter funcional. Podemos assim criar quatro níveis de classes de prioridades, sobre as quais os processos transitam de acordo com a ocupação da(s) última(s) janela(s) de execução. Aqui consideremos que o nível 1 é o mais prioritário e o nível 4 é o menos prioritário. Num nível 1 teríamos processos considerados como terminais, isto é, uma classe para que transitam os processos após acordarem no seguimento da espera por informação do dispositivo de entrada *standard*. Num nível seguinte (nível 2) teríamos assim uma classe para que transitam os processos após acordarem no seguimento de espera por informação de um dispositivo distinto do dispositivo de entrada *standard* (I/O genérico). Mais abaixo temos uma classe orientada para as janelas de execução mais pequenas, isto é, para os processos que transitam quando completaram a sua última janela de execução. Por fim, num nível 4 preocupamo-nos com os processos cuja janela de execução é maior, numa classe para os que transitam, então, quando completaram em sucessão um número pré-definido de janelas de execução - tratamos de processos considerados CPU-intensivos, pelo que o objetivo será atribuir-lhes, no futuro, uma janela de execução mais longa, mas menos vezes.

classes de prioridade

Sendo que este último método já se carateriza por dinamizar as atribuições de prioridades, note-se que um problema que se coloca, não obstante, é a redução do tempo de *turnaround* dos trabalhos (*jobs*) que constituem a fila de processamento, em sistemas do tipo *batch*. Desde que se conheçam estimativas dos tempos de execução para todos os processos na fila é, então, possível estabelecer-se uma ordenação para a execução dos processos que minimiza o tempo médio de *turnaround* do grupo. A este método damos o nome de **shortest job first** (SJF) (ou *shortest process next*, SPN) e, com efeito, admitindo que uma fila contém j trabalhos, cujas estimativas dos tempos de execução são, respetivamente, te_n, com $n = 1, 2, \ldots, j$, então o tempo médio de *turnaround* vê-se em (2.1).

shortest job first

$$\text{tempo médio de } turnaround = te_1 + \frac{(j-1)}{j \cdot te_2} + \cdots + \frac{1}{j \cdot te_j} \quad (2.1)$$

Note-se que o tempo médio de *turnaround* de (2.1) é mínimo quando a ordenação dos processos é feita por ordem crescente de tempo de execução.

Uma abordagem semelhante [5] pode ser usada em sistemas interativos para se estabelecer a prioridade dos processos que competem pela posse do processador. Tal princípio consiste assim em procurar estimar-se a fração de ocupação da janela de execução seguinte em termos da ocupação das janelas passadas, atribuindo-se o processador ao processo para o qual esta estimativa é menor. Seja assim fe_1 a estimativa da fração de ocupação da primeira janela de execução de um dado processo e, f_1, a fração de ocupação efetivamente verificada. A estimativa para a fração de ocupação da próxima janela vem dada por $fre_2 = afe_1 + (1\text{-}a)f_1$ (com $a \in [0, 1]$) e, para a j-ésima janela, por $fe_j = afe_{j-1} + + (1\text{-}a)f_{j-1}$ (novamente com $a \in [0, 1]$) - o coeficiente a é utilizado para controlar o grau com que a história passada de execução vai influenciar a estimativa presente (controla a histerese do sistema).

Se houver uma carga muito grande de processos intensivos em termos de módulos de I/O, os processos CPU-intensivos correm o risco de sofrerem adiamento indefinido, se a disciplina de escalonamento for aplicada sem qualquer correção. Mas uma forma de corrigir tal problema está na incorporação no cálculo de prioridade do tempo que o processador aguarda no estado "pronto a executar". Sendo r esse tempo, que é normalizado em termos da duração do intervalo de execução, a prioridade p de cada processo pode ser definida por (2.2), em que o coeficiente b controla o peso com que o tempo de espera afeta a definição da prioridade.

$$p = \frac{1 + br}{fe_j} \quad (2.2)$$

Este tipo de disciplinas de escalonamento são conhecidas por promoverem o **envelhecimento dos processos** (processo de *aging*).

envelhecimento dos processos

Processos e threads

Mostrámos até agora que o conceito de processo é mais complexo e súbtil que o que apresentámos inicialmente e que, de facto, nele estão contidos dois conceitos distintos e potencialmente independentes: um conceito que se relaciona com a posse de um recurso e outro que se relaciona com a execução. Esta distinção leva-nos à discussão de uma nova construção, denominada de **thread**.

thread

Uma *thread* não é nada mais que, como o seu sentido literal indica, um fio de execução, sendo que existe assim um determinado *program counter* que sinaliza a localização de uma instrução que deve ser executada a seguinte, um conjunto de registos internos do processador que contêm os valores atuais das variáveis em processamento e uma *stack* que armazenará a história de execução. Estas propriedades, embora surjam reunidas num processo, podem ser tratadas em separado pelo sistema de operação. Quando tal acontece, os processos dedicam-se a agrupar um conjunto de recursos e os *threads*, também conhecidos processos de baixo peso (*light-weight processes*), constituem entidades executáveis independentes dentro do contexto de um mesmo processo.

Como é possível criar-se mais do que um fio de execução (mais do que uma *thread*), dizemos que um determinado sistema ou aplicação é **multithreaded**. Consideremos assim a Figura 2.10. Nesta figura podemos distinguir processos de *threads*, do ponto de vista da gestão de processos. Num modelo *single-threaded* (sem conceito de *thread*), a representação de um processo inclui o seu bloco de controlo do processo e espaço de en-

multithreaded

dereçamento, tal como *stacks* para o utilizador e para o *kernel*. Num ambiente *multithreaded* existe ainda um bloco único de controlo do processo e espaço de endereçamento associado, mas agora existem também espaços separados para cada *thread*, contendo as *stacks*, tal como um bloco de controlo para cada, contendo os valores dos registos, prioridades e outras informações relacionadas com o estado das *threads*.

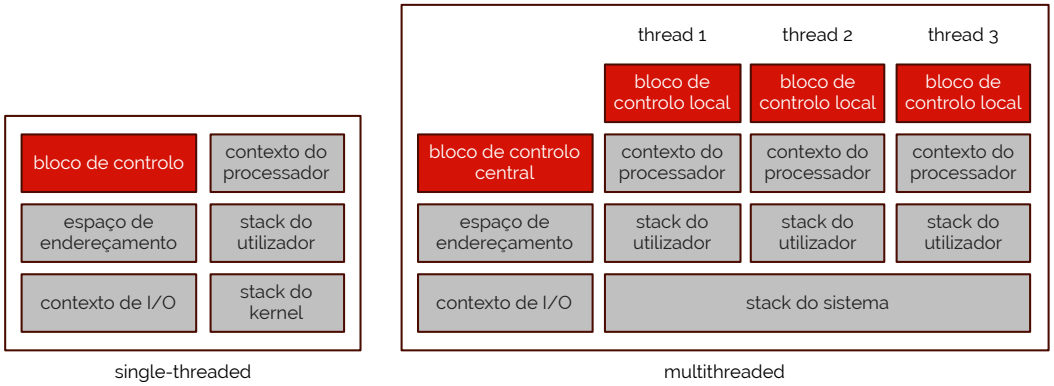


figura 2.10

O facto de reorganizarmos um programa para um ambiente *multithreaded* permite que haja uma maior simplicidade na decomposição de uma solução e uma maior modularidade na implementação em programas que envolvem múltiplas atividades e que atendem múltiplas solicitações. Por outro lado, também há uma melhor gestão de recursos do sistema computacional, havendo uma partilha do espaço de endereçamento e do contexto de I/O, entre as *threads* que compõem uma aplicação, e tornando-se mais simples gerir a ocupação da memória principal e o acesso eficiente aos dispositivos de entrada/saída. O facto de poder executar vários fios em simultâneo (para um só processo), permite que haja uma eficiência e maior velocidade de execução.

Este assunto será estudado com mais detalhe na disciplina de Arquitetura de Computadores Avançada (a4s1), mas note-se que trabalhar com paralelismo entre *threads* não é algo que seja muito fácil, para um programador habituado ao processamento e execução de instruções de forma sequencial. Embora cada *thread* geralmente esteje associada a uma função ou procedimento que implementa uma atividade específica, fazer o *debugging* de código com paralelismo é particularmente difícil.

O diagrama de estados de uma *thread* não é muito difícil. Na Figura 2.11 podemos ver o escalonamento de baixo nível de uma *thread* e com ele podemos perceber que é muito parecido ao do processo. Não valerá a pena representar nem o escalonamento de nível médio, nem alto, porque grande parte dos aspetos abordados e controlados a este nível são partilhados com o processo.

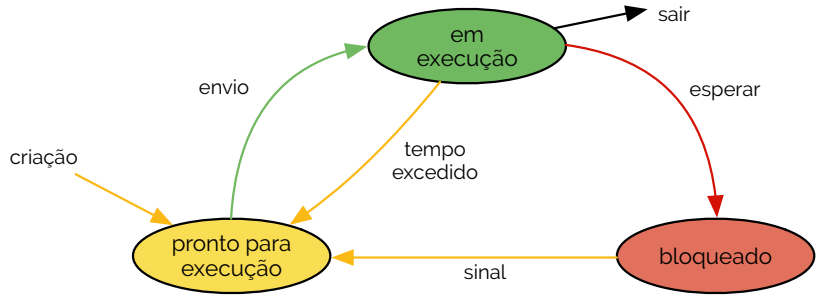


figura 2.11

Como podemos ver pela Figura 2.10 temos que cada *thread* possui uma *stack* para o nível de utilizador e uma partilhada a nível de *kernel*. Por esta mesma razão as *threads* que forem implementadas sobre o *kernel* são menos versáteis que as criadas sobre o espaço de utilizador. As *threads* de nível de utilizador são implementadas por bibliotecas específicas do espaço de utilizador, que fornecem apoio à criação, gestão e escalonamento de *threads* sem que o *kernel* tenha que intervir. Em particular, estas *threads* podem ser cri-

adas pela biblioteca `pthread` da linguagem C. No Código 2.2 podemos ver um programa onde se cria uma *thread* através da biblioteca `pthread`.

código 2.2

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static int status = 0;

void* threadLuke(void* arg) {
    printf("I am Luke Skywalker.\n");
    status = EXIT_SUCCESS;
    pthread_exit(&status);
}

int main(void) {
    pthread_t thread;
    if (pthread_create(&thread, NULL, threadLuke, NULL) != 0) {
        perror("Error launching Luke's thread.");
        return EXIT_FAILURE;
    }
    int* status_p;
    if (pthread_join(thread, (void**) &status_p) != 0) {
        perror("Error on waiting too much for Luke...");
        return EXIT_FAILURE;
    }
    printf("And I, Darth Vader, am your father.\n");
    return EXIT_SUCCESS;
}
```

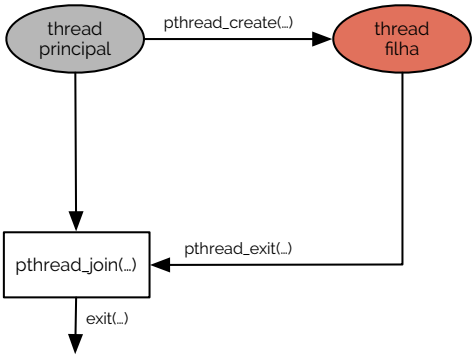
Se executarmos o código acima podemos obter algo como o que vemos no Output 2.1, depois de compilarmos com a opção `-pthread`. Este programa poderá ser encontrado com o nome `code2_2.c` e compilado (com a Makefile) fazendo `make code2_2`.

output 2.1

```
$ ./code2_2
I am Luke Skywalker
And I, Darth Vader, am your father.
```

Esta biblioteca do UNIX (`pthread`) possui então um ciclo de vida muito semelhante ao dos procesos, como podemos ver na Figura 2.12.

figura 2.12



No sistema operativo Linux, a questão da implementação de *threads* é feita de um modo muito artificioso [5], pelo que o *fork* cria um novo processo a partir de um já existente por cópia integral do seu contexto alargado e o *clone* cria um novo processo a partir de um já existente por cópia apenas do seu contexto restrito (contexto do processador), partilhando o espaço de endereçamento e o contexto de I/O, e iniciando a sua execução pela invocação de uma função que é passada como parâmetro. Se virmos bem, ambas as funções poderão ser usadas, e são, para a criação das *threads*, pelo que não há uma distinção efetiva entre processos e *threads*, que o Linux ainda designa de forma indiferente de tarefas (*tasks*), e eles são tratados pelo *kernel* da mesma maneira.

A única diferença que é causada neste processo é que as diversas *threads* lançadas no âmbito de um mesmo processo partilham a mesma identificação de grupo, o que permite que a mudança de contexto possa ser diferenciada.

3. Comunicação entre Processos

Quando falamos de ambientes multiprogramados, devemos ter em consciência que estamos a gerir vários processos que coexistem no tempo e em execução, sendo que estes poderão ser independentes ou cooperantes entre si. Dizer que dois processos são **independentes** significa que ambos vivem e morrem sem que nunca interajam de um modo propriamente explícito, havendo uma vertente de **competição** pelos vários recursos computacionais que possam usar ao longo do tempo. Por outro lado, dizer que dois processos são **cooperantes** significa que ambos partilham informação e comunicam entre si de um modo explícito, pelo que a partilha exige que haja um espaço de endereçamento comum e a comunicação entre os vários processos exige que haja uma via através da partilha de um espaço de endereçamento ou da existência de um canal de comunicação que interliga os processos intervenientes.

independentes

competição

cooperantes

No caso de vários processos independentes, por haver a partilha de recursos iguais numa situação de competição, precisamos de conseguir garantir uma situação de **exclusão mútua**, isto é, uma situação que só um processo interage à vez com um determinado recurso. Note-se que esta situação, também pode acontecer com processos cooperantes, sendo que ambos têm regiões partilhadas.

exclusão mútua

Princípios gerais

Nesta área de estudo, como já vimos anteriormente, um aspeto que está em comum é a **concorrência**. Num cenário de concorrência temos recursos partilhados por vários processos que, muitas vezes, terão de ser acedidos em exclusão mútua. A exclusão mútua, como também já vimos, é um requisito que, quando um processo está numa **região crítica** (secção de código de um processo que requer acesso a uma memória partilhada e que não pode ser executado enquanto outro processo está na mesma secção de código) que acede a memória partilhada, nenhum outro processo com acesso à mesma secção crítica pode entrar na mesma memória partilhada. Esta situação acaba por ser uma **condição de corrida**, dado que vários processos (ou *threads*) leem ou escrevem um item numa memória partilhada e o resultado final depende do tempo relativo da sua execução.

concorrência

região crítica

condição de corrida

Acontece que ao entrar num regime de exclusão mútua deverá existir um controlo muito cuidado para se evitarem situações de **deadlock**. Uma situação de *deadlock* é tal na qual dois ou mais processos deixam de ser capazes de prosseguir a sua execução porque cada um aguarda pelo outro que possa acabar, embora isso nunca vá acontecer. Uma forma de garantir que um determinado bloqueio de acesso não aconteça duas vezes ou mais vezes mais que os respetivos desbloqueios, é criando **operações atómicas**, isto é, conjuntos de instruções que são indivisíveis, isto é, onde nenhum outro processo poderá interferir num estado intermédio entre o estado inicial de operação e o estado final. Esta atomicidade garante uma isolação dos processos concorrentes.

deadlock

operações atómicas

Poderá também acontecer uma situação de **adiamento indefinido** (ou *starvation*) onde que um ou mais processos competem pelo acesso a uma região crítica e, devido a uma conjunção de circunstâncias em que surgem continuamente processos novos que o(s) ultrapassam nesse desígnio, o acesso é sucessivamente adiado, pelo que se está, por iss, perante um impedimento real à continuação dele(s).

adiamento indefinido

Consideremos um exemplo real, no Código 3.1.

código 3.1

```
void echo() {
  characterIn = getchar();
  characterOut = characterIn;
  putchar(characterOut);
}
```

Consideremos agora que temos um sistema multiprogramável de um único processador e que temos um único utilizador. O utilizador poderá executar vários programas, e

cada aplicação usa um mesmo teclado como *input* e o mesmo ecrã para *output*. Porque cada aplicação necessita de usar o procedimento `echo`, fará sentido que seja um procedimento partilhado carregado diretamente para uma zona de memória global, acessível por todas as partes - assim só uma cópia da função será criada, poupando espaço.

A partilha da memória principal entre os processos é útil para permitir uma interação mais eficiente e próxima entre os processos. No entanto, tal partilha poderá trazer problemas. Vejamos a seguinte sequência de eventos: primeiro, um processo P1 invocaria o procedimento `echo`, sendo logo interrompido depois da chamada de `getchar`, que devolve o seu valor e guarda-o em `characterIn` - neste ponto assumamos que o carater mais recentemente introduzido, x, foi guardado na variável `characterIn`; um processo P2 é ativado e invoca o procedimento `echo`, o qual é executado e guarda um carater mais recente, o y, em `characterIn` e depois mostra-o no ecrã; tendo o processo P2 terminado, P1 termina agora a sua execução pela impressão do seu conteúdo da variável `characterOut` que, tendo sido re-escrita pelo processo P2, irá imprimir y também.

Assim, o primeiro carater é perdido e o segundo será exibido duas vezes. A essência deste problema é a manipulação da variável global partilhada, `characterIn`, dado que múltiplos processos têm acesso a esta variável. Se um processo atualiza a variável global e depois é interrompido, outro processo poderá alterar a variável, antes que o primeiro use o seu valor. Suponhamos, no entanto, que nós permitimos que apenas um processo possa aceder a essa região de código a cada vez. Então teríamos uma ordem de eventos como: primeiro, um processo P1 invocaria o procedimento `echo`, sendo logo interrompido depois da chamada de `getchar`, que devolve o seu valor e guarda-o em `characterIn` - neste ponto assumamos que o carater mais recentemente introduzido, x, foi guardado na variável `characterIn`; um processo P2 é ativado e invoca o procedimento `echo`, embora permaneça suspenso - como P1 ainda está dentro do procedimento `echo`, P2 fica bloqueado de entrar no procedimento, sendo que é suspenso enquanto espera que P1 termine a sua execução de `echo`; tendo o processo P1 terminado, imprime o conteúdo da variável `characterOut` que será x; o P2 agora poderá prosseguir a sua execução, recebendo y e imprimindo y.

Este último exemplo mostra que é necessário proteger variáveis partilhadas globalmente (e recursos) e que a única forma de o fazer é controlando o código que acede à variável. Se impusermos assim uma disciplina em que só unicamente um processo à vez é que pode executar `echo` e que o procedimento `echo` só poderá ser executado de forma atómica antes de se tornar disponível para outro processo, então o tipo de erro que discutimos não ocorrerá.

Para aceder a uma região crítica com exclusão mútua temos que garantir alguns aspetos, de forma a que estas soluções se tornem invioláveis. Primeiro, deve haver uma independência da velocidade de execução relativa dos processos intervenientes, ou do seu número, dado que nada ser presumido acerca destes fatores. Mais, um processo que se encontre fora da região crítica não poderá impedir que outro lá entre, não podendo também ser adiado indefinidamente a possibilidade de acesso à região crítica a qualquer processo que o requeira. Por fim, o tempo de permanência de um processo na região crítica deverá ser necessariamente finito [7]. Esta exclusão mútua poderá ser garantida, em termos de código, por métodos que indiquem a entrada na região e a saída, como mostra o Código 3.2.

código 3.2

```
// processo 1
void process1 {
  while (true) {
    // código precedente
    entrarRegiaoCritica(id);
    // secção crítica
    sairRegiaoCritica(id);
    // código sucedente
  }
}

// processo 2
void process2 {
  while (true) {
    // código precedente
    entrarRegiaoCritica(id);
    // secção crítica
```

```
    sairRegiaoCritica(id);
    // código sucedente
  }
}

. . .

// processo N
void processN {
  while (true) {
    // código precedente
    entrarRegiaoCritica(id);
    // secção crítica
    sairRegiaoCritica(id);
    // código sucedente
  }
}
```

Tipo de soluções

Para este tipo de problemas de concorrência podemos ter, de forma global, dois tipos de soluções distintas: soluções por *software* e soluções por *hardware*. Começando pelas **soluções por software** temos que estas, quer sejam implementadas num monoprocessador, quer num multiprocessador com memória partilhada, supõem o recurso em última instância ao conjunto de instruções básico do processador, ou seja, as instruções de transferência de dados de e para a memória são de tipo *standard*: leitura e escrita de um valor.

soluções por software

Por outro lado, temos as **soluções por hardware**, as quais supõem o recurso a instruções especiais do processador para garantir, a algum nível, a atomicidade na leitura e subsequente escrita de uma mesma posição de memória. Estas soluções são muitas vezes suportadas pelo próprio sistema de operação e podem mesmo estar integradas na linguagem de programação utilizada.

soluções por hardware

Note-se, não obstante, que em relação às soluções *software*, a única suposição adicional diria respeito ao caso do multiprocessador, em que a tentativa de acesso simultâneio a uma mesma posição de memótia por parte de diferentes processadores é necessariamente serializada por intervenção de um árbitro.

Construção de uma solução por software

Tentemos então construir, passo-a-passo, uma solução para o nosso problema de acesso a uma região crítica. Basicamente, o que precisamos agora é de criar o corpo para as funções `entrarRegiaoCritica(id)` e `sairRegiaoCritica(id)`.

Consideremos para o efeito o Código 3.3.

código 3.3

```
// processo 0
while (true) {
  while (turn != 0);
  regiaoCritica();
  turn = 1;
  regiaoNaoCritica();
}

// processo 1
while (true) {
  while (turn != 1);
  regiaoCritica();
  turn = 0;
  regiaoNaoCritica();
}
```

Uma primeira forma de o fazer seria criar uma variável global e partilhada de nome `turn`, inicializada a 0, controla e acompanha a vez de quem é que pode aceder à região crítica e aceder ou atualizar a memória partilhada. Nesta solução a que damos o nome de **alternância estrita**, inicialmente o processo 0 inspeciona a variável `turn`, verificando que está a 0, e entra na região crítica. O processo 1, também inspeciona e vê que está a 0, pelo que fica num ciclo à espera que fique 1 (sendo que testa continuamente se o valor de `turn` é 1). Esta espera, **busy-waiting**, deve ser evitado, sendo que consome muito tempo de CPU.

alternância estrita

busy-waiting

Quando o processo 0 sai da região crítica coloca `turn` a 1, de forma a permitir que o processo 1 entre na região crítica. Agora consideremos que este processo termina a execução muito rapidamente, de forma a que ambos os processos (processos 0 e 1) estão em região não-crítica, com o valor de `turn` a 0, novamente. Neste momento o processo 0 termina o ciclo de forma rápida, saindo da sua região crítica e colocando `turn` a 1. Neste ponto `turn` está a 1 e ambos os processos estão na região não-crítica. Mas de repente, o processo 0 termina a sua região não-crítica e volta para o topo do seu ciclo. Infelizmente, e porque o valor de `turn` é 1, o processo 0 não poderá entrar na região crítica. E assim ficará até que o processo 1 entre na região crítica e volte a colocar o valor a 0.

Esta aplicação, com N processos, para a substituição das funções `entrarRegiaoCritica(id)` e `sairRegiaoCritica(id)` teria o aspeto do Código 3.4.

código 3.4

```
unsigned int turn = 0;

void entrarRegiaoCritica(unsigned int pid) {
  while (pid != turn);
}

void sairRegiaoCritica(unsigned int pid) {
  if (pid == turn) {
    turn = (turn + 1) % N;
  }
}
```

Esta proposta de solução, ao supor a entrada na região crítica dos processos intervenientes num regime de alternância estritamente sucessiva, não constitui uma solução geral para o problema de acesso a uma região crítica com exclusão mútua. Estão-se a violar aspetos que caraterizámos anteriormente como propriedades do controlo de acesso às regiões críticas, entre os quais a independência da velocidade de execução relativa dos processos intervenientes (sendo que o ritmo de execução desenvolve-se ao ritmo do processo que faz menos acessos por unidade de tempo à região crítica) e de que um processo fora da região crítica não pode impedir outro de lá entrar.

Considerando novamente dois processos, podemos agora extender a nossa solução substituíndo a nossa variável `turn` por um array de booleanos, com um número de índices igual ao número de processos candidatos a entrarem na região crítica. Criamos assim o Código 3.5.

código 3.5

```
unsigned bool isIn[2] = {false, false};

void entrarRegiaoCritica(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  while (isIn[otherPID]);
  isIn[pid] = true;
}

void sairRegiaoCritica(unsigned int pid) {
  isIn[pid] = false;
}
```

A solução do Código 3.5, numa análise cuidada, não garante a exclusão mútua em todas as circunstâncias. Consideremos que um processo 0 entra, testando a variável `isIn[1]` que tem o seu valor a falso e um processo 1 entra, testando `isIn[0]` que também se encontra a falso. O processo 0 e 1 alteram o valor de `isIn[0]` e `isIn[1]` para verdadeiro e ambos os processos entram na região crítica.

Então, como aparentemente o problema está no facto de se proceder ao teste da variável do outro e só depois se alterar o valor da variável própria, criemos uma nova solução onde fazemos o contrário, indicando a vontade de entrar na região crítica. Consideremos assim o Código 3.6.

código 3.6

```
unsigned bool wantEnter[2] = {false, false};

void entrarRegiaoCritica(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  wantEnter[pid] = true;
  while (wantEnter[otherPID]);
}
```

```
void sairRegiaoCritica(unsigned int pid) {
  wantEnter[pid] = false;
}
```

Neste caso, com a ordem invertida, temos que passou a ser garantida a exclusão mútua em todas as circunstâncias, mas surgiu, entretanto, uma consequência não desejada - agora há a possibilidade de ocorrência de *deadlock*. Consideremos então que o processo 0 entra e altera o valor de `wantEnter[0]` para verdadeiro e o processo 1 também entra e altera o valor de `wantEnter[1]` para verdadeiro. Quando o processo 1 testar a variável `wantEnter[0]`, por ela ter o valor verdadeiro, este vai ficar à espera da permissão de acesso à região crítica (e igual para o processo 0), pelo que os processos ficam em *deadlock*.

Para resolver o problema da solução do Código 3.6 é necessário que, pelo menos um dos processos, tenha que recuar temporariamente para que o impasse desapareça. Chegamos assim à solução do Código 3.7, a qual até é quase válida e usada por vezes na prática, como no protocolo *Ethernet* (vide apontamentos de Fundamentos de Redes (a3s1)), que utiliza uma variante deste para estabelecer a exclusão mútua no acesso ao canal de comunicação (CSMA/CD).

código 3.7

```
unsigned bool wantEnter[2] = {false, false};

void entrarRegiaoCritica(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  wantEnter[pid] = true;
  while (wantEnter[otherPID]) {
    wantEnter[pid] = false;
    delay(random);
    wantEnter[pid] = true;
  }
}

void sairRegiaoCritica(unsigned int pid) {
  wantEnter[pid] = false;
}
```

Sob o ponto de vista teórico, a solução do Código 3.7 está incorreta, isto porque poderá sempre considerar-se a ocorrência de uma combinação de circunstâncias tal que conduza inevitavelmente a situações de *deadlock* ou de adiamento indefinido. Isto acontece porque de forma teórica uma variável aleatória trata uma solução estocástica e não-determinística, pelo que, à partida, esta não resulta em 100% de probabilidade de sucesso.

Chegamos assim a um algoritmo já popular e que usa um mecanismo de alternância para resolver o conflito resultante da situação de contenção de dois processos. Este algoritmo, denominado de **algoritmo de Dekker**, está representado no Código 3.8.

⊙ **Theodorus Jozef Dekker**

algoritmo de Dekker

código 3.8

```
unsigned bool wantEnter[2] = {false, false};
unsigned int priority = 0;

void entrarRegiaoCritica(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  wantEnter[pid] = true;
  while (wantEnter[otherPID]) {
    if (pid != priority) {
      wantEnter[pid] = false;
      while (pid != priority);
      wantEnter[pid] = true;
    }
  }
}

void sairRegiaoCritica(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  priority = otherPID;
  wantEnter[pid] = false;
}
```

O algoritmo de Dekker, conforme Código 3.8, garante efetivamente a exclusão mútua no acesso à região crítica, evita situações de *deadlock* e de adiamento indefinido e não presume o que quer que seja quanto à velocidade de execução relativa dos processos intervenientes, no entanto a sua generalização, para N processos é difícil. De facto, não se con-

hece qualquer algoritmo que o faça, respeitando todas as propriedades desejáveis para a solução.

Em 1981, G. L. Peterson descobriu uma forma muito mais simples de obter exclusão mútua, deixando a solução de Dekker obsoleta. O **algoritmo de Peterson**, mostrado no Código 3.9, usa a seriação por ordem de chegada para resolver o conflito resultante da situação de contenção de dois processos. Isto é conseguido forçando cada processo a escrever a sua identificação numa mesma variável (`turn`). Assim, uma leitura subsequente permite, por comparação, determinar qual foi o último que aí escreveu.

⊙ Gary L. Peterson

algoritmo de Peterson

código 3.9

```
unsigned bool wantEnter[2] = {false, false};
unsigned int turn = 0;

void entrarRegiaoCritica(unsigned int pid) {
  unsigned int otherPID = 1 - pid;
  wantEnter[pid] = true;
  turn = pid;
  while (turn == pid && wantEnter[otherPID] == true);
}

void sairRegiaoCritica(unsigned int pid) {
  wantEnter[pid] = false;
}
```

Este algoritmo do Código 3.9 garante efetivamente a exclusão mútua no acesso à região crítica, evita situações de *deadlock* e de adiamento indefinido e não presume o que quer que seja em relação à velocidade de execução relativa dos processos intervenientes. A sua generalização, contrariamente ao algoritmo de Dekker, para N processos é bastante simples, se se tiver em conta a analogia de formação de uma fila de espera.

Construção de soluções por hardware (semáforos e monitores)

Tendo já abordado várias aproximações à solução apresentada por Peterson, com base no *software*, podemos agora ir-nos aproximando de soluções com base em *hardware*. Assim, comecemos por olhar para uma solução com uma pequena ajuda do *hardware* [2].

Alguns computadores, especialmente os que foram desenhados tendo em conta o multi-processamento, têm instruções do tipo `TSL Rx, lock` (denominadas de **test and set lock** - TSL) que funcionam da seguinte forma: é lido o conteúdo de memória da palavra em `lock` e colocado no registo `Rx`, sendo guardado um valor não-nulo no endereço de memória `lock`. As operações de leitura da palavra e de armazenamento desta devem ser garantidas de serem indivisíveis. Assim, o CPU bloqueia o bus de memória enquanto executa a instrução TSL, de forma a proibir que outros CPUs acedam à memória enquanto o processo se encontra em execução.

test and set lock

É importante notar que desativar um bus de memória é bem diferente de desativar interrupções. Desativar interrupções e executar uma leitura de uma palavra na memória, seguida de uma escrita, não previne que um segundo processador, ligado ao mesmo bus de dados, aceda à palavra, entre a leitura e a escrita. De facto, ao desativar as interrupções no processador número 1 nada acontece ao processador 2, como vimos em Arquitetura de Computadores II (a2s2). A única forma de manter o processador 2 fora da memória até que o processador 1 termine é bloquear o bus, o que requer um *hardware* especial (basicamente uma linha do bis que verifique se este está bloqueado e indisponível a outros processadores que não o que o bloqueou).

Para usar a instrução TSL, iremos usar uma variável partilhada denominada de *lock*, de forma a coordenar o acesso à memória partilhada. Quando o *lock* é 0, qualquer processo poderá trocá-lo para 1 usando a instrução TSL e ler ou escrever a memória partilhada. Quando terminado, o processo volta a pôr *lock* a 0 usando uma instrução comum de `move`.

Então como é que esta instrução pode ser usada a previnir dois processos de entrar em simultâneo nas regiões críticas? Ora, a solução está visível no Código 3.10, numa linguagem Assembly em pseudo-código.

```
entrarRegiao:    tsl     reg, lock        ; copiar lock para registo e por lock = 1
                 cmp     reg, #0          ; o lock é zero?
                 jne     entrarRegiao     ; se não for zero, lock = 1 e entrar ciclo
                 ret                      ; retornar para caller, entrou na região

sairRegiao:      move    lock, #0         ; colocar lock = 0
                 ret                      ; retornar para caller
```

código 3.10

Então, no Código 3.10, a primeira instrução copia o antigo valor de *lock* para o registo `reg`, colocando *lock* = 1. Depois o antigo valor é comparado com 0. Se não for zero, então o *lock* já foi feito, pelo que o programa irá repetir a verificação, executando novamente a função `entrarRegiao`. Mais cedo ou mais tarde o *lock* terá o valor de 0 (quando o processo que atualmente ocupa a região crítica sair desta) e a subrotina retorna, com um valor de *set* para o *lock*. Por outro lado, para sair da região crítica apenas é necessário comprovar que *lock* = 0.

Portanto, uma solução para o problema da região crítica é fácil e já está determinada. Antes de entrar na região crítica, então um processo deverá executar a função `entrarRegiao`, que fará um *busy-waiting* até que o *lock* fique livre. Depois de deixar a região crítica, o processo deverá então chamar a função `sairRegiao`, que coloca um 0 em *lock*.

Tanto esta solução como a solução vista e alcançada pelo algoritmo de Peterson contam com o defeito de terem *busy-wait*. Na sua essência, o que estes métodos fazem é: se quero entrar na região crítica, então vou tentando entrar, perguntando sempre se posso, e, quando tiver livre-passe, entro, caso contrário tenho de ficar à espera.

Não só este procedimento é exaustivo para o CPU, como também pode ter efeitos inesperados. Consideremos assim um computador com dois processos, A, com prioridade alta, e B, com prioridade baixa. O escalonamento é tal, como vimos, que A é executado desde que está no estado "pronto para execução". Num determinado momento, com B na região crítica, A torna-se pronto para execução (imaginemos que uma operação de I/O é terminada). Então A começará o período de *busy-waiting*, mas como B nunca é escalonado enquanto A está em execução, B nunca terá a sua oportunidade de sair da região crítica, pelo que A entra num ciclo sem cessar. Esta situação é muitas vezes referida como o **problema da prioridade invertida**.

problema da prioridade invertida

Agora vejamos algumas primitivas de comunicação entre processos que bloqueiam, ao invés de desperdiçar tempo de CPU quando estes não estão permitidos de entrar em regiões críticas. Uma das maneiras mais simples é manipular usando o par `sleep` e `wakeup`. O comando `sleep` é uma chamada de sistema que faz com que quem invoca (*caller*) bloqueie, isto é, seja suspendido até que outro processo o acorde. Assim, para o fazer, poderá usar a outra chamada de sistema, denominada de `wakeup`, que apenas possui um parâmetro de entrada (o processo a ser acordado). Alternativamente, tanto o `sleep` como o `wakeup`, cada um tem um parâmetro de entrada que é o endereço de memória usado para fazer uma correspondência entre `sleep`'s e `wakeup`'s. Contudo, o uso destas primitivas necessitam ainda de garantir a atomicidade das operações, uma vez que por si só não resolvem o problema todo.

Como resposta a este problema, em 1965, Edsger W. Dijkstra [8] sugeriu usar uma variável inteira para contar o número de *wakeups* para uso futuro. Na sua proposta, um novo tipo de variável, ao qual ele deu o nome de **semáforo**, foi introduzido. Um semáforo é assim uma variável tal que pode ter o valor 0, indicando que nenhum *wakeup* foi guardado, ou um número positivo se um ou mais *wakeups* estiverem pendentes.

⊙ **Edsger W. Dijkstra**

semáforo

Dijkstra propôs então que houvesse duas operações possíveis em semáforos, hoje em dia vulgarmente denominadas de *up* e *down* (generalizações de *sleep* e de *wakeup*, respetivamente). A operação de **down** serve para verificar se o valor de um semáforo é maior que 0. Se sim, então o seu valor será decrementado e continua. Se o valor for 0, então o processo é colocado em estado *sleep* (a dormir) sem que seja completado o *down* que foi pedido, por enquanto. O fazer estas verificações, mudar o valor do semáforo e possivelmente colocar o processo a dormir é tudo feito de forma atómica. É assim garantido que uma vez que uma operação sobre um semáforo é iniciada, nenhum outro processo poderá aceder ao semáforo até que este tenha terminado a operação (ou bloqueado). Esta atomi-

down

cidade é muitíssimo importante para resolver problemas onde a sincronização deverá ser atestada e onde se pretendem evitar situações de condições de corrida.

up

A operação de **up** incrementa o valor do semáforo endereçado. Se um ou mais processos estiverem a dormir num mesmo semáforo, à espera de poderem terminar um *down* que tenha ficado em espera, então um deles (escolhidos pelo sistema de operação) é permitido de fazer o *down*. Assim, depois de um *up* num semáforo com processos a dormir, o semáforo continuará a ser 0, sendo que haverá menos um processo a dormir. A operação de incremento do semáforo também é indivisível.

problema dos produtores/consumidores

Analisemos agora um problema que é clássico na área dos sistemas de operação e que nos permite ter uma visão mais concreta dos problemas que enfrentamos e como solucioná-los, que é o **problema dos produtores/consumidores**. Consideremos assim uma solução deste problema, bastante autoexplicativo pelo seu nome, no Código 3.11, com semáforos.

código 3.11

```
#define N 100                       // número de slots no buffer

typedef int semaphore;              // os semáforos são um tipo especial de inteiros

semaphore access = 1;               // controlo de acesso à região crítica
semaphore emptyness = N;            // controlo de slots vazios do buffer
semaphore fullness = 0;             // controlo de slots cheios do buffer

void producer(void) {
  int item;
  while (true) {
    item = produceItem();           // gerar qualquer coisa para pôr no buffer
    down(&emptyness);               // decrementar o número de slots vazios no buffer
    down(&access);                  // entrar na região crítica
    insertItem(item);               // inserir item novo no buffer (região crítica)
    up(&access);                    // sair da região crítica
    up(&fullness);                  // incrementar o número de slots cheios no buffer
  }
}

void consumer(void) {
  int item;
  while (true) {
    down(&fullness);                // decrementar o número de slots cheios no buffer
    down(&access);                  // entrar na região crítica
    item = removeItem();            // remover item do buffer (região crítica)
    up(&access);                    // sair da região crítica
    up(&emptyness);                 // incrementar o número de slots vazios no buffer
    consumeItem(item);              // fazer qualquer coisa com o item
  }
}
```

Com a solução do Código 3.11 não precisamos de usar métodos para verificar se o *buffer* está cheio ou vazio, dado que os próprios semáforos suspenderão o processo caso tal se verifique. Contudo, vejamos o Código 3.12 e tentemos designar o que é que está mal com as linhas a vermelho.

código 3.12

```
#define N 100                       // número de slots no buffer

typedef int semaphore;              // os semáforos são um tipo especial de inteiros

semaphore access = 1;               // controlo de acesso à região crítica
semaphore emptyness = N;            // controlo de slots vazios do buffer
semaphore fullness = 0;             // controlo de slots cheios do buffer

void producer(void) {
  int item;
  while (true) {
    item = produceItem();           // gerar qualquer coisa para pôr no buffer
    down(&access);                  // entrar na região crítica
    down(&emptyness);               // decrementar o número de slots vazios no buffer
    insertItem(item);               // inserir item novo no buffer (região crítica)
    up(&access);                    // sair da região crítica
    up(&fullness);                  // incrementar o número de slots cheios no buffer
  }
}

void consumer(void) {
  int item;
  while (true) {
    down(&fullness);                // decrementar o número de slots cheios no buffer
    down(&access);                  // entrar na região crítica
    item = removeItem();            // remover item do buffer (região crítica)
```

```
        up(&access);                // sair da região crítica
        up(&emptyness);             // incrementar o número de slots vazios no buffer
        consumeItem(item);          // fazer qualquer coisa com o item
    }
}
```

Ora, se executarmos o Código 3.12 iríamos entrar numa situação de *deadlock* porque estaríamos a entrar na região crítica antes do tempo. Isto resulta numa situação de *deadlock* porque da próxima vez que o consumidor tentar aceder ao *buffer* fará um *down* no `access`, agora 0, bloqueando também.

Esta solução do Código 3.11 usa três semáforos: um chamado `fullness` para a contagem do número de *slots* do *buffer* que estão cheios, um chamado `emptyness` para a contagem do número de *slots* do *buffer* que estão vazios e um chamado `access` para ter a certeza de que o produtor e o consumidor não acedem o *buffer* ao mesmo tempo. O `fullness`, inicialmente está a 0, `emptyness` inicializa-se ao número de espaços do *buffer* (tamanho) e o `access` a 1. Os semáforos que são incializados a 1 e são usados por dois ou mais processos para garantir que somente um deles é que podem entrar numa região crítica em simultâneo são denominados de **semáforos binários**. Se cada processo fizer uma operação de *down* mesmo antes de entrar na região crítica e um *up* após a deixar, a exclusão mútua é garantida.

semáforos binários

Quando a abilidade de contagem do semáforo não é necessária, uma versão simplificada deste poderá ser usada, chamada de **mutex**. Os *mutexes* (palavra proveniente de MUTual EXclusion) são ótimos apenas para manter exclusão mútua a alguns recursos partilhados ou a porções de código. Sendo fáceis e eficientes de implementar, geralmente estão disponíveis em pacotes de manipulação de *threads*, que são implementadas totalmente em espaço de utilizador.

mutex

Um *mutex* é assim uma variável partilhada que se encontra em um de dois estados possíveis: desbloqueada ou bloqueada. Por conseguinte, somente 1 bit é requirido para a representar, embora na prática seja sempre identificada por um inteiro, com 0 significando que está desbloqueada e qualquer outro valor o contrário.

A biblioteca `pthread` que vimos anteriormente fornece um número de funções que poderão ser usadas para sincronizar *threads*. O mecanismo mais básico usa uma variável *mutex*, que poderá estar bloqueada ou desbloqueada, de forma a proteger cada região crítica. Uma *thread* que queira, assim, entrar numa região crítica primeiro tem de tentar bloquear o *mutex* associado, pelo que se o *mutex* estiver desbloqueado, então a *thread* poderá entrar imediatamente e o bloqueio é efetuado. Caso contrário, então a chamada é bloqueada até que haja um desbloqueio. Se várias *threads* já estiverem à espera do mesmo *mutex*, então quando há um desbloqueio, só uma delas é que é permitida avançar e bloqueá-lo. Estes bloqueios não são obrigatórios, dado que cabe apenas ao programador usá-los, para que possa garantir que se estejam a usar bem as *threads*.

Contudo, o desbloqueio e o bloqueio não são as únicas funções próprias da biblioteca `pthread`. Para além das funções de criação das *threads* (e respetiva destruição), em adição aos *mutexes*, a biblioteca `pthread` oferece um segundo mecanismo de sincronização denominado de **variáveis de condição**. Os *mutexes* são muito bons a permitir ou negar o acesso a uma região crítica, mas as variáveis de condição permitem que *threads* bloqueiem devido a condições que não estão corretas ou verificadas positivamente. Quase todas as aplicações e usos da biblioteca `pthread` contam com os dois mecanismos de sincronização.

variáveis de condição

Como um pequeno exemplo, consideremos o problema dos produtores/consumidores outra vez: uma *thread* coloca coisas no nosso *buffer* e outro retira-as e faz coisas com elas. Se o produtor descobrir que não existem mais espaços onde colocar coisas novas, então deverá ter de bloquear até que haja espaço. Os *mutexes* permitem que isso aconteça de forma atómica sem que haja interferência de outras *threads*, mas tendo descoberto que o *buffer* está cheio, o produtor precisa de arranjar uma forma de bloquear e de ser acordado mais tarde, quando a condição do *buffer* deixar de estar cheio for válida. É aqui que as variáveis de condição entram.

As chamadas mais importantes para as variáveis de condição estão representadas na tabela da Figura 3.1, com uma respetiva descrição. Como já devemos estar à espera, existem chamadas igualmente para criar e destruir variáveis de condição. Estas poderão ter atributos e existem vários métodos para gerir vários cenários[4]. As operação primárias em variáveis de condição são a `pthread_cond_wait` e a `pthread_cond_signal`. Basicamente, a primeira operação bloqueia a *thread* que a invoca até que outra qualquer lhe envie um sinal (utilizando a segunda operação). As razões para o bloqueio e espera não fazem parte do protocolo de espera e sinalização. Note-se que, enquanto que o `signal` envia um sinal para uma qualquer *thread* que o aguarda, existe uma outra operação, mais forte que esta, que envia um sinal para qualquer uma (não apenas uma e arbitrária) que aguarde um sinal - a operação `pthread_cond_broadcast`.

figura 3.1

operação	descrição
`pthread_cond_init`	cria uma variável de condição
`pthread_cond_destroy`	destroi uma variável de condição
`pthread_cond_wait`	bloqueia uma thread à espera de um sinal
`pthread_cond_signal`	envia um sinal para uma thread arbitrária que o espera
`pthread_cond_broadcast`	envia um sinal para todas as threads que o esperam

As variáveis de condição e os *mutexes* são sempre usados em conjunto, de forma a que ambos formam um padrão em que uma *thread* bloqueia um *mutex* e só depois aguarda numa variável de condição quando não pode obter o que pretende. Eventualmente, uma segunda *thread* fará um sinal e a primeira poderá continuar.

Também é importante referir que as variáveis de condição, contrariamente aos semáforos, não têm memória. Se um sinal é enviado para uma variável de condição sobre a qual nenhuma *thread* está à espera, então o sinal é perdido. Uma forma de pensar neste problema é fazer uma analogia com os rádios FM/AM. Quando sintonizamos um destes equipamentos geralmente estamos à espera de ouvir um sinal. Vários rádios podem estar à espera de um só sinal, mas o sinal para ser propagado não precisa de ter alguém à sua espera, neste caso, sendo emitido e sendo perdido, como uma estação radioamadora que transmite numa banda em que ninguém está à escuta (ninguém os está a ouvir, mas eles estão a transmitir).

Com os semáforos e os *mutexes* as comunicações entre os vários processos parecem mais fáceis, mas olhemos bem para o problema que analisámos no Código 3.12. Este problema é apontado para mostrar o quão cuidadosos devemos ser ao usar os semáforos: um erro muito súbtil e tudo cai como que por efeito avalanche.

Para facilitar a forma como escrever programas com base em concorrência, Brinch Hansen e Hoare propuseram um nível mais alto de sincronização a que deram o nome de **monitor**. As suas propostas, de facto, não foram totalmente iguais, mas a ideia era de que um monitor seria uma coleção de procedimentos, variáveis e estruturas de dados, tudo junto num pequeno pacote ou módulo. No fundo, os processos poderão chamar os procedimentos num monitor quando bem lhes apetecesse, mas não poderiam aceder diretamente à estrutura de dados do monitor por procedimentos declarados fora deste. No Código 3.13 podemos ver um monitor escrito na sua linguagem original - a Pascal[5].

monitor

código 3.13

```
monitor example
  var
    val: DATA;                    (* dados partilhados *)
    c: condition;                 (* variável de condição de sincronização *)

  procedure procedure1 (. . .);
  end
```

4 Para mais informações verificar os manuais da biblioteca *pthread* acessíveis no terminal UNIX-like através do comando "man 3 pthread".

5 A linguagem C não pode ser usada aqui porque os monitores são um conceito de linguagem e a linguagem C não o tem.

```
  function function1 (. . .) : real;
  end

  begin
    . . .
  end
end monitor;
```

Para impedir a coexistência de duas *threads* dentro de um monitor, é necessária uma regraa que estipule como a contenção decorrente do sinal é resolvida. Assim sendo, várias soluções foram aparecendo, entre as quais os monitores de Hoare, os monitores de Brinch Hansen e os de Lampson/Redell.

- Sir Charles Hoare
- Per Brinch Hansen
- Butler W. Lampson
- David D. Redell

Comecemos pelos **monitores de Hoare**. Nestes monitores, a *thread* que invoca a operação de envio de sinal é colocado fora do monitor, para que a *thread* acordada possa prosseguir. Esta implementação está designada num diagrama na Figura 3.2.

monitores de Hoare

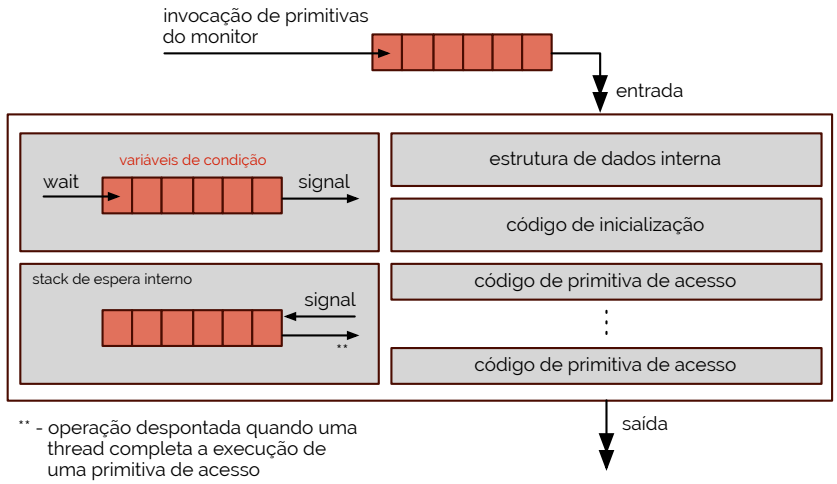


figura 3.2

A solução apresentada por Hoare é muito geral, mas a sua implementação exige também a existência de uma *stack* (não visível na Figura 3.2) onde são colocadas as *threads* postas fora do monitor por invocação de um sinal.

Com a solução implementada por Brinch Hansen, a *thread* que invoca a operação de envio de sinal liberta imediatamente o monitor, como podemos ver na Figura 3.3.

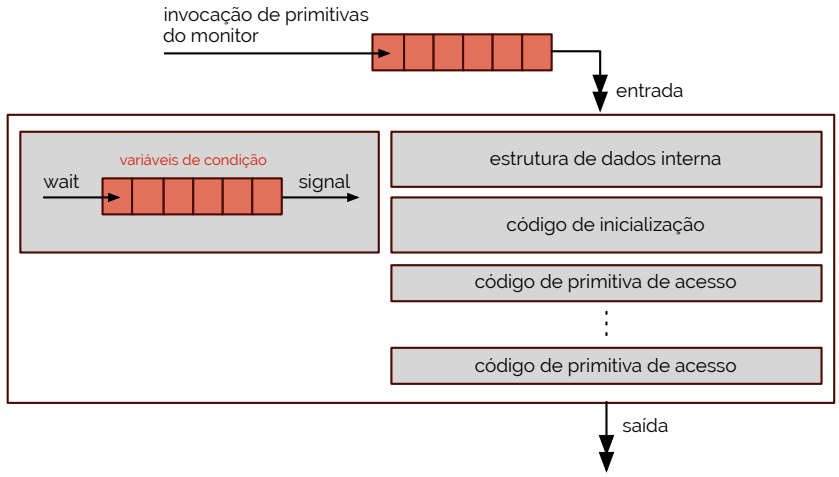


figura 3.3

A solução da Figura 3.3, sendo que lhe, desde a anterior, removemos-lhe o *stack* de espera interno, é simples de implementar, mas pode tornar-se bastante restritivo porque só há a possibilidade, agora, de execução de um sinal por cada invocação de uma primitiva de acesso. Para corrigir esta situação chega-nos o **monitor de Lampson/Redell**, onde a *thread*

monitores de Lampson/Redell

que invoca a operação de envio do sinal prossegue a sua execução e a *thread* acordada mantém-se fora do monitor, competindo pelo acesso a ele. Esta solução é simples de implementar, podendo ser vista na Figura 3.4, mas pode originar situações em que algumas *threads* são colocadas em adiamento indefinido.

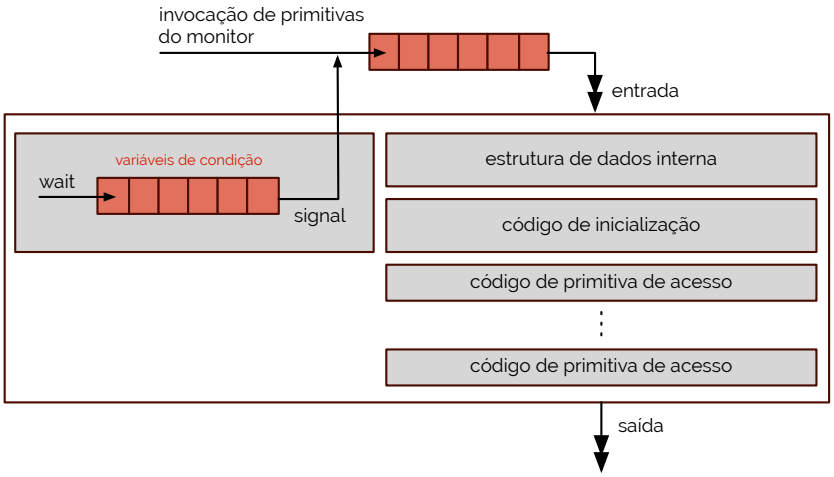


figura 3.4

Uma última solução possível de comunicação entre processos é a passagem de mensagens. Este mecanismo de comunicação usa duas primitivas, as chamadas de sistema `send` e `receive`. O princípio em que se baseia é muito simples: sempre que um processo P_R, digamos, remetente, pretende comunicar com um processo P_D (dito destinatário) envia-lhe uma mensagem através de um canal de comunicação estabelecido entre ambos (chamada `send`, com a assinatura `send(destinatário, &mensagem)`. Por outro lado, para P_D receber a mensagem, tem tão só que aceder ao canal de comunicação e aguardar a sua chegada (chamada `receive`, com a assinatura `receive(remetente, &mensagem)`.

Os sistemas de passagens de mensagens, contudo, têm alguns problemas que os monitores e semáforos não têm, especialmente quando os processos comunicantes se encontram em máquinas diferentes dentro de uma rede: pode acontecer, por exemplo, a perda de mensagens na rede. Para o sistema se proteger destes problemas, tanto o remetente como o destinatário devem assumir que, mal uma mensagem chegue a um destinatário, este terá que enviar uma mensagem especial de reconhecimento (**acknowledge**). Se o remetente ainda não tiver recebido o reconhecimento passado t tempo, então faz uma retransmissão.

acknowledge

Agora consideremos o que é que acontece se a mensagem é enviada com sucesso, mas o reconhecimento fica perdido pelo caminho. Num cenário destes, como o remetente não recebeu o reconhecimento, retransmitirá a mensagem, pelo que o recetor irá receber a mesma mensagem, no fim de contas, duas vezes. É assim essencial que o recetor consiga também distinguir uma mensagem nova, de uma retransmissão de uma antiga. Normalmente este problema é resolvido colocando números de sequência em cada mensagem original, sendo que se o destinatário de uma mensagem receber uma com um mesmo número de sequência que uma anterior, então saberá que se trata de um duplicado e ignorar-lo-á. Este estudo pode ser revisto na disciplina de Fundamentos de Redes (a3s1).

Este modo de comunicação define assim uma diferença entre níveis de **sincronização**, podendo haver então a troca de mensagens de forma síncrona, mas **não-bloqueante**, isto é, em que a sincronização é da responsabilidade dos processos intervenientes, sendo que quem envia fá-lo e regressa sem qualquer informação sobre se a mensagem foi efetivamente recebida ou não, ou de uma forma **bloqueante**, quando as operações de envio e de receção contêm em si mesmas elementos de sincronização, fazendo com que, por exemplo, a operação de envio envie a mensagem e bloqueie até que esta seja efetivamente recebida.

sincronização, não-bloqueante

bloqueante

Os sistemas de troca de mensagens também têm de saber lidar com a questão de como é que os processos são nomeados, de forma a que um processo especificado numa operação de envio ou de receção não seja, de todo, ambíguo. Assim, existem mecanismos de **autenticação** que permitem resolver este problema.

autenticação

Consideremos novamente, então, o problema dos produtores/consumidores, mas agora com a nuance de aplicação das trocas de mensagens. No Código 3.14 podemos então ver uma solução implementada.

código 3.14

```
#define N 100                                   // número de slots no buffer

void producer(void) {
  int item;
  message message;                              // buffer de mensagens

  while (true) {
    item = produceItem();                       // produzir algo para por no buffer
    receive(consumer, &message);                // esperar por uma resposta vazia
    buildMessage(&message, item);               // construir uma mensagem a enviar
    send(consumer, &message);                   // enviar item para consumidor
  }
}

void consumer(void) {
  int item;
  message message;

  for (int i = 0; i != N, i++) {
    send(producer, &message);                   // enviar N mensagens vazias
  }
  while (true) {
    receive(producer, &message);                // receber mensagem com item
    item = extractItem(&message);               // extrair item de mensagem
    send(producer, &message);                   // enviar mensagem vazia de resposta
    consumeItem(item);                          // fazer qualquer coisa com o item
  }
}
```

No Código 3.14 assumimos que todas as mensagens têm o mesmo tamanho e que as mensagens enviadas mas não recebidas são carregadas automaticamente pelo sistema de operação. Nesta solução, um total de N mensagens é usado, análogo aos N *slots* dos *buffers* de memória partilhada que usámos antes. Então o consumidor começa por enviar N mensagens vazias para o produtor. Sempre que o produtor tem um item para fornecer ao consumidor, pega numa mensagem vazia e envia de volta com o item. Desta forma, o número total de mensagens no sistema é sempre igual, pelo que poderá ser guardado num bloco de memória com um tamanho fixo e já pré-determinado.

Se o produtor trabalhar mais rápido que o consumidor, então todas as mensagens irão ficar cheias e à espera do consumidor. Neste ponto o produtor ficará bloqueado, à espera que a mensagem volte. Se o consumidor trabalhar mais depressa, então o contrário poderá acontecer: todas as mensagens ficarão vazias até que o produtor as encha. O consumidor ficará então bloqueado, à espera de uma mensagem completa.

Existem muitas variantes desta solução de passagem de mensagens. Olhemos então para a forma como as mensagens são endereçadas. Uma forma fácil de compreender e de aplicar é atribuindo um endereço único a cada processo e enviar mensagens para esses endereços. Uma forma diferente de aplicar a troca de mensagens é pensando numa estrutura de dados análoga a uma **caixa de correio** (*mailbox*). Uma caixa de correio é um local onde podemos colocar um *buffer* para um número fixo de mensagens, tipicamente especificado na criação desta. Quando as caixas são usadas, os endereços de envio e de receção deixam de ser o dos processos, mas o das caixas. Assim, quando um processo tenta enviar uma mensagem para uma caixa que já está cheia, este é suspenso até que uma mensagem seja removida dessa caixa, de forma a deixar um espaço vazio para essa.

caixa de correio

Para o problema do produtor/consumidor, ambos produtores e consumidores teriam de possuir caixas de correio grandes o suficiente para receberem N mensagens. O produtor enviaria as mensagens para a caixa de correio do consumidor e este responderia com mensagens vazias para a caixa de correio do produtor. Quando as caixas de correio são usadas, o mecanismo de *buffering* torna-se mais claro: a caixa de correio do destinatário aguenta mensagens que já foram enviadas para o processo, mas que ainda não foram aceites.

Uma estratégia mais radical, eliminando a necessidade de *buffering*, para a troca de mensagens, é a estratégia de **rendez-vous**. Aqui a operação de envio, se for feita antes

rendez-vous

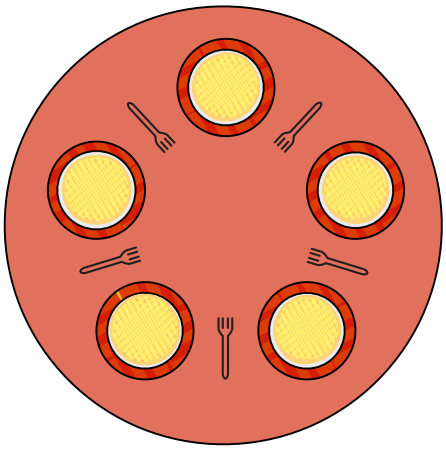
da operação de receção, então será bloqueada até que uma receção ocorra, num tempo t em que a mensagem pode ser copiada diretamente do remetente para o destinatário. De forma semelhante se for executada uma operação de receção primeiro, então o recetor é bloqueado à espera que um envio ocorra. Esta estratégia é mais simples de implementar, mas é menos flexível, dado que o remetente e o destinatário são forçados a trabalhar passo-a-passo, em concordância.

Problema do jantar dos filósofos

Em 1965, Dijkstra colocou e resolveu um problema de sincronização que rapidamente se tornou um clássico e algo obrigatório de teste, por cada vez que se cria uma nova solução de comunicação entre processos. A este problema ele deu o nome de **problema do jantar dos filósofos**.

problema do jantar dos filósofos

figura 3.5



Este problema é formulado da seguinte forma, tendo em conta a mesa desenhada na Figura 3.5. Cinco filósofos estão sentados ao longo de uma mesa circular e cada um tem um prato de esparguete à sua frente. O esparguete, para ser comido e porque é meio escorregadio, necessita de ser comido com a ajuda de dois garfos. Assim, entre cada prato está um garfo.

A vida de um filósofo consiste em períodos alternados de comer e pensar. Quando um filósofo fica com fome suficiente, ele tenta pegar no seu garfo esquerdo e direito, à vez, numa ordem qualquer. Se tiver sucesso a pegar nos talheres, então come durante algum tempo e depois coloca os talheres novamente na mesa, continuando a pensar. A questão que se coloca então é: conseguir-se-á criar um programa para cada filósofo, tal que faz o que é suposto e ninguém fica preso, à espera?

No Código 3.15 temos a solução mais óbvia. Um procedimento `takeFork` espera até que o talher específico fique disponível e depois pega nele. Infelizmente esta solução está errada. Suponhamos assim que todos os cinco filósofos decidem pegar nos seus talheres em simultâneo. Não tornando este jantar num encontro em que todos dão as mãos uns aos outros, nenhum dos filósofos conseguirá pegar nos talheres corretos, pelo que haverá uma situação de *deadlock*.

código 3.15

```
#define N 5

void philosopher(int i) {
  while (true) {
    think();
    takeFork(i);                          // pegar no garfo esquerdo
    takeFork((i+1) % N);                  // pegar no garfo direito
    eat();
    putFork(i);                           // largar garfo esquerdo
    putFork((i+1) & N);                   // largar garfo direito
  }
}
```

Podemos então fazer umas modificações no nosso programa do Código 3.15 de forma a que depois de pegar no garfo esquerdo, o programa verifique se o direito está disponível ou não. Se não estiver, o filósofo deverá baixar o garfo esquerdo, esperar algum tempo, e repetir o processo todo novamente. Acontece que esta aproximação a uma solução, por usar *busy-waiting*, também não é boa solução, e, teoricamente, falha, porque poderá inclusive, gerar uma situação de adiamento indefinido.

Uma melhoria que poderá ser feita ao Código 3.15, que não possui perigo de situação de *deadlock* nem de adiamento indefinido, é a proteção das chamadas a seguir à `think`, por um semáforo binário. Assim, antes de pegar nos talheres, cada filósofo faria um *down* num *mutex*. Depois de pegarem nos talheres, fariam um *up* ao *mutex*. De um ponto de vista mais teórico, esta solução é adequada ao caso. De um ponto de vista mais prático, apenas terá um pequeno senão: apenas um filósofo estará a comer à vez: com cinco talheres disponíveis, devemos permitir que dois filósofos possam comer ao mesmo tempo.

A solução apresentada no Código 3.16 [2] não tem *deadlocks* e permite o paralelismo máximo para um número arbitrário de filósofos. Usamos, nela, um array `state` para manter um controlo sobre quem é que está a comer, pensar ou com fome (a tentar pegar nos talheres). Um filósofo poderá transitar para o estado "comer" só se nenhum dos seus vizinhos está nesse estado. Os vizinhos do filósofo i estão então definidos pelas macros `LEFT` e `RIGHT`, para simplificar o código.

código 3.16

```
#define N         5                     // número de filósofos
#define LEFT      (i+N-1) % N           // número do filósofo à esquerda
#define RIGHT     (i+1) % N             // número do filósofo à direita
#define THINKING 0                      // estado "pensar"
#define HUNGRY    1                     // estado "com fome"
#define EATING    2                     // estado "a comer"

typedef int semaphore;
int state[N];
semaphore mutex = 1;                    // exclusão mútua para regiões críticas
semaphore semaphore[N];                 // um semáforo por filósofo

void philosopher(int i) {
  while (true) {
    think();                            // filósofo i pensa
    takeForks(i);                       // filósofo i está com fome
    eat();                              // filósofo i come
    putForks(i);                        // filósofo larga talheres para pensar
  }
}

void takeForks(int i) {
  down(&mutex);                         // entrar na região crítica
  state[i] = HUNGRY;                    // gravar que filósofo i está com fome
  test(i);                              // tentar pegar em 2 garfos
  up(&mutex);                           // sair da região crítica
  down(&semaphore[i]);                  // bloquear se garfos não conseguidos
}

void putForks(int i) {
  down(&mutex);                         // entrar na região crítica
  state[i] = THINKING;                  // gravar que filósofo i acabou de comer
  test(LEFT);                           // ver se vizinho esquerdo pode comer
  test(RIGHT);                          // ver se vizinho direito pode comer
  up(&mutex);                           // sair da região crítica
}

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

Sinais em UNIX

sinais

Uma forma de comunicação para com os processos é através de **sinais**. Um sinal constitui assim uma interrupção produzida no contexto de um processo onde lhe é comunicada a ocorrência de um acontecimento especial. Podendo ser despoletado pelo *kernel*, em resultado de situações de erro ao nível de *hardware* ou de condições específicas ao nível do *software*, pelo próprio processo, por outro processo ou pelo utilizador através do dispositi-

vo *standard* de entrada. Tal como o processador no tratamento de exceções, o processo assume uma de três atitudes possíveis relativamente a um sinal: ignorá-lo, bloqueá-lo (impedir que interrompa o processo durante intervalos de processamento bem definidos) ou executar uma ação associada - esta pode ser a ação estabelecida por defeito quando o processo é criado (conduz habitualmente à sua terminação ou suspensão de execução), ou uma ação específica que é introduzida (registada) pelo próprio processo em tempo de execução.

Existem, assim, em UNIX, um conjunto de chamadas ao sistema que possibilitam o processamento de sinais, como o `kill` (para enviar um sinal a um processo ou grupo de processos), `o raise` (para enviar um sinal ao próprio processo), entre outros... Na Figura 3.6 podemos ver uma lista de alguns sinais com o respetivo valor, ação e causa.

figura 3.6

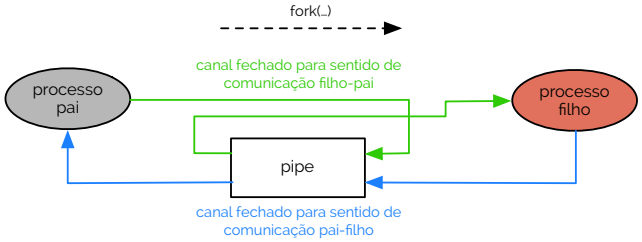
operação	valor	ação	descrição
SIGHUP	1	term	paragem detetada num controlador ou processo de controlador morreu
SIGINT	2	term	interrupção do teclado
SIGQUIT	3	core	saída do teclado
SIGILL	4	core	instrução ilegal
SIGABRT	6	core	sinal de abortar
SIGFPE	8	core	exceção de vírgula flutuante
SIGKILL	9	term	sinal de morte
SIGSEGV	11	core	referência de memória inválida
SIGPIPE	13	term	broken pipe: escrever no pipe sem leitores
SIGALRM	14	term	sinal de temporizador de um alarme
SIGTERM	15	term	sinal de terminação
SIGUSR1	10	term	sinal definido pelo utilizador (1)
SIGUSR2	12	term	sinal definido pelo utilizador (2)
SIGCHLD	17	ign	processo-filho terminado ou parado
SIGCONT	18		sinal de continuação, caso parado
SIGSTOP	19	stop	parar processo
SIGTSTP	20	stop	parar escrita no terminal tty
SIGTTIN	21	stop	transportar input do tty para o processo de fundo (em background)
SIGTTOU	22	stop	transportar output do tty para o processo de fundo (em background)

Pipes em UNIX

O UNIX disponibiliza um canal de comunicação elementar que é estabelecido entre dois ou mais processos, denominado de **pipe**. O mecanismo de *piping* constitui a forma tradicional de comunicação entre processos em UNIX, tendo sido inclusivamente introduzido por ele. Este processo apresenta, contudo, algumas limitações, como o facto de ser half-duplex (só permite comunicação num sentido), só poder ser usado entre processos que estão relacionados entre si (um *pipe* é criado por um processo, que a seguir se duplica uma ou mais vezes e a comunicação é então estabelecida entre o pai e o filho, ou entre dois filhos. A sua principal utilização está na composição de comandos complexos a partir de uma organização em cadeia de comandos mais simples, em que o *standard output* de um dado comando é redirecionado para a entrada de um *pipe* e o *standard input* do comando seguinte é redirecionado para a saída do mesmo *pipe*, como mostra a Figura 3.7.

pipe

figura 3.7



Prevenções de deadlock

Existem quatro condições que ocorrem necessariamente, sempre que estamos perante uma situação de *deadlock*: condição de exclusão mútua (cada recurso existente, ou está livre, ou foi atribuído a um e um só processo - a sua posse não pode ser partilhada); condição de espera com retenção (cada processo, ao requerer um novo recurso, mantém na sua posse todos os recursos anteriormente solicitados); condição de não-libertação (ninguém, a não ser o próprio processo, pode decidir da libertação de um recurso que lhe tenha sido previamente atribuído); e condição de espera circular (formou-se uma cadeia circular de processos e recursos, em que cada processo requer um recurso que está na posse do processo seguinte na cadeia). Tendo consciência desta implicação, podemos tentar negá-la e tentar verificar sobre que condições é que não há *deadlock*.

Negando a implicação temos que se não há exclusão mútua no acesso a um recurso ou não há espera com retenção ou há libertação de recursos ou não há espera circular, então não há *deadlock*. Assim, desde que uma das condições necessárias à ocorrência de *deadlock* seja negada pelo algoritmo de acesso aos recursos, o *deadlock* torna-se impossível. Políticas com esta caraterística designam-se de políticas de prevenção de deadlock no sentido estrito.

A negação do haver exclusão mútua no acesso a um recurso, da espera com retenção e da não-libertação de recursos já foi vista em secções anteriores, mas ainda não tivemos a oportunidade de abordar a negação de espera circular. Fazer isto, significa estabelecer uma ordenação linear dos recursos e impor que um processo, quando procura obter os recursos que necessita para a sua continuação, o faça sempre por ordem crescente (decrescente) do número associado a cada um. Desta maneira, a possibilidade de formação de uma cadeia circular de processos e recursos está posta de parte[6].

Por exemplo, no caso do jantar dos filósofos, um dos filósofos pega nos garfos pela ordem inversa aos outros. Face à forma utilizada para identificar os filósofos e os talheres, é fácil constatar que os filóosofos de 0 a n-2, ao pegarem primeiro no garfo da esquerda e, só depois, no da direita, solicitam os garfos (recursos) por ordem crescente do seu número de ordem, e, com o filósofo n-1 passa-se exatamente o contrário, sendo que o garfo da esquerda tem o número de ordem n-1 e o da direita o número de ordem 0. Assim, a imposição de uma ordenação crescente na atribuição de recursos exige apenas que o filósofo n-1 pegue nos garfos pela ordem inversa à dos outros: primeiro o da direita e, só depois, o da esquerda.

Estas políticas de prevenção de *deadlock* no sentido estrito, embora sendo seguras, são muito restritivas, pouco eficientes e difíceis de aplicar em situações muito gerais.

Uma alternativa menos restritiva do que a prevenção no sentido estrito é não negar *a priori* qualquer das condições necessárias à ocorrência de *deadlock*, mas monitorizar continuamente o estado interno do sistema, de modo a garantir que a sua evolução se faz apenas entre estados ditos seguros.

estado seguro

Neste contexto, define-se então um **estado seguro** como uma qualquer distribuição dos recursos do sistema, livres ou atribuídos aos processos que coexistem, que possibilita a terminação de todos eles. Por oposição, um estado diz-se **inseguro**, se não for possível fazer-se uma tal afirmação sobre ele. Políticas com esta caraterística dizem-se de políticas de prevenção de *deadlock* no sentido lato (em inglês, *deadlock avoidance*).

inseguro

Note-se que é necessário o conhecimento completo de todos os recursos do sistema e cada processo tem que indicar à cabeça a lista de todos os recursos que vai precisar - só assim se pode caraterizar um estado seguro. Mais, um estado inseguro não é sinónimo de *deadlock*, indo, contudo, considerar-se sempre como o pior caso possível para garantir a sua não-ocorrência.

---

6 A ocorrência de adiamento indefinido não está impedida, sendo que a solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. Para resolver este tipo de problemas geralmente usam-se mecanismos de *aging*.

Definimos assim um NTR_i como o número total de recursos do tipo i (com $i = 0, 1, \ldots, N - 1$), um $R_{i,j}$ como o número de recursos do tipo i requiridos pelo processo j (com $i = 0, 1, \ldots, N\text{-}1$ e $j = 0, 1, \ldots, M\text{-}1$) e um $A_{i,j}$ como o número de recursos do tipo i já atribuídos ao processo j (com $i = 0, 1, \ldots, N\text{-}1$ e $j = 0, 1, \ldots, M\text{-}1$). O nosso problema poderá então ser abordado de duas maneiras diferentes: haver um impedimento do lançamento de um novo processo, quando não podem ser garantidas as condições da sua terminação ou um impedimento de atribuição de um novo recurso a um dado processo. Vejamos primeiro o primeiro caso. Aqui, um processo P_M só é lançado se e só se (3.1) se verificar.

$$NTR_i \geq R_{i,M} + \sum_{j=0}^{M-1} R_{i,j} \qquad (3.1)$$

Relativamente ao segundo caso, um novo recurso de tipo i só é atribuído ao processo P_M se e só se for possível ordenar os processos numa sucessão $j' = f(i,j)$ em que se tem sempre que a desigualdade em (3.2) é válida, com $i = 0, 1, \ldots, N\text{-}1$ e $j' = 0, 1, \ldots, M\text{-}1$.

$$R_{i,j'} - A_{i,j'} < NTR_i - \sum_{k \geq j'}^{M-1} A_{i,k} \qquad (3.2)$$

algoritmo dos banqueiros de Dijkstra

Ao algoritmo de (3.2) damos o nome de **algoritmo dos banqueiros de Dijkstra** e aplica-se quando for possível encontrar pelo menos uma sucessão de atribuição de recursos que conduz à terminação de todos os processos que coexistem. Este algoritmo está assim modelado de forma a que um pequeno banco de uma cidade possa negociar com um grupo de clientes de forma a saber a quem é que abre novas linhas de crédito. O que o algoritmo faz é verificar se ao garantir um pedido, fica num estado inseguro. Se tal acontecer, então o pedido é negado. Se ao garantir um pedido, o banco permanecer num estado seguro, então este é tomado. Na Figura 3.8 (a) vemos quatro clientes (*A, B, C* e *D*), cada um a quem foi garantida a oportunidade de possuir unidades de crédito. O banqueiro sabe de antemão que nem todos os clientes vão necessitar da quantidade máxima de crédito disponível agora, pelo que só tem 10 unidades, ao invés de 22, por exemplo, para os servir. Os clientes vão então aplicando o crédito aos seus negócios e fazendo empréstimos de tempos em tempos (na nossa analogia será pedir recursos). Num certo instante, a situação dos clientes é a visível na Figura 3.8 (b) - este estado é seguro, porque com duas unidades em sobra, o banqueiro ainda consegue atrasar quaisquer pedidos à exceção dos de *C*, pelo que irá permitir que *C* termine e libertará todos os quatro dos seus recursos. Com quatro unidades na mão, o banqueiro poderá deixar tanto *D* como *B* terem os recursos que pretendem e assim por diante.

Consideremos agora o que é que aconteceria se um pedido de *B* por mais uma unidade fosse garantido em Figura 3.8 (b) - teríamos a situação da Figura 3.8 (c), que não é segura. Se todos os clientes pedissem, de repente, pelos seus créditos máximos para empréstimo, o banqueiro não poderia satisfazer nenhum dos quatro, e poderíamos ter um *deadlock*.

figura 3.8

	tem	max
A	0	6
B	0	5
C	0	4
D	0	7

livres = 10

(a)

	tem	max
A	1	6
B	1	5
C	2	4
D	4	7

livres = 2

(b)

	tem	max
A	1	6
B	2	5
C	2	4
D	4	7

livres = 1

(c)

O algoritmo do banqueiro de Dijkstra considera cada pedido, à medida que ocorre, verificando se pode garantir que fica num estado seguro. Se conseguir, então o pedido é dado, caso contrário, é adiado para depois. Para verificar se um estado é seguro, o banqueiro verifica se tem recursos suficientes para satisfazer o cliente. Se sim, então os empréstimos são assumidos como os que têm retorno, e o cliente mais próximo do limite de crédito, e assim por diante.

Uma última alternativa é pura e simplesmente atribuir os recursos sempre que eles estão disponíveis. A possibilidade de *deadlock* está então sempre presente e terá que ter sida em conta. Uma estratégia possível de seguir é então a conhecida **estratégia da avestruz**, que é basicamente ignorar pura e simplesmente o problema e, quando o *deadlock* acontecer, matar os procesos envolvidos ou, no caso limite, reiniciar o sistema (resposta mais comum). Por outro lado, de forma mais cautelosa, podemos aplicar estratégias de **deteção de deadlock**. Aqui estaríamos a monitorizar, de tempos a tempos, o estado dos diferentes processos, procurando determinar se existem cadeias circulares de processos e recursos.

estratégia da avestruz

deteção de deadlock

Após a deteção de *deadlock*, a cadeia circular de processos e recursos pode ser quebrada usando diversos métodos, entre os quais a libertação forçada de um recurso - em algumas situações, um recurso pode ser retirado a um processo, que é entretanto suspenso, permitindo o prosseguimento da execução dos restantes (mais tarde, o recurso volta a ser atribuído ao processo e este é recomeçado, sendo este um método mais eficaz, embora exija que o estado do recurso possa ser salvaguardado) -, o *rollback* - estabelecer um funcionamento do sistema que impõe o armazenamento periódico do estado dos diferentes processos (deste modo, quando ocorre *deadlock*, um recurso é libertado e o processo que o detinha vê a sua execução recuada até a um ponto anterior à atribuição desse recurso) - e a morte de processos - é o método mais radical e fácil de implementar.

4. Gestão da Memória

Consideremos que estamos a planear construir uma casa. Ao longo desse plano existem muitas peocupações que devem ser tidas em conta: lembramo-nos do número de pisos, se tem jardim ou não, se está virada para Norte ou se está virada para Oeste para ver o pôr do Sol todos os dias, ... mas há algo que é mais importante que tudo: temos **espaço** para o nosso plano? Quando falamos de espaço não podemos apenas falar de espaço para o produto final - no nosso caso a casa e um jardim, por exemplo -, mas também para o material de construção ao longo das obras. Algum destes espaços não têm de ser propriamente propriedade da casa, futuramente, mas este trabalho exige alguma **gestão** do espaço.

espaço

gestão

Em ciências da computação é análoga a preocupação com a **memória** e a respetiva **gestão de memória**. O espaço de endereçamento de um processo tem que estar, pelo menos parcialmente, residente em memória principal para que ele possa ser executado. De forma a maximizar a ocupação do processador e melhorar o tempo de resposta, um sistema computacional deve manter vários processos residentes em memória principal.

memória

gestão de memória

Embora a memória principal tenha vindo a crescer a um ritmo incessante ao longo dos anos, é um facto que "os programas tendem a expandir-se, preenchendo toda a memória disponível" (Lei de Parkinson).

⊙ **Cyril Northcote Parkinson**

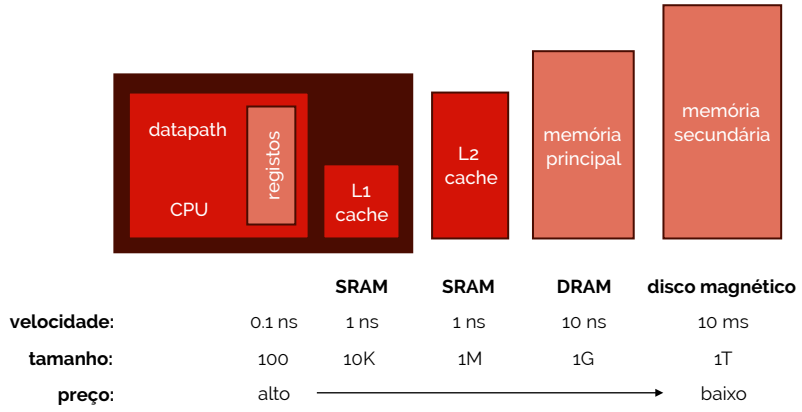
Hierarquia da Memória

Idealmente, o programador de aplicações gostaria de ter disponível uma memória que fosse infinitamente rápida, não volátil e barata. Como vimos em Arquitetura de Computadores II (a2s2) tal não é possível. Assim, a memória de um sistema computacional está organizada tipicamente em níveis diferentes, formando uma hierarquia como a que vemos na Figura 4.1.

Na hierarquia de memória temos então três grandes componentes: a memória **cache** (pequena, de dezenas de kB a unidades de MB, muito rápida, volátil e cara), a memória **principal** (de tamanho médio, de centenas de MB a unidades de GB, volátil e de velocidade de acesso e preço médios) e a memória **secundária** (grande, de centenas a milhares de GB, lenta, mas não-volátil e barata).

cache
principal
secundária

figura 4.1



A memória cache vai conter uma cópia das posições de memória (instruções e operandos) mais frequentemente referenciadas pelo processador no passado próximo. Para maximizar a velocidade de acesso, face à frequência de relógio dos processadores atuais, a memória cache está localizada no próprio circuito integrado do processador (nível 1) e num circuito integrado autónomo colado no mesmo substrato (níveis 2 e 3), e o controlo da transferência de dados de e para a memória principal é feito de um modo quase completamente transparente ao programador de sistemas.

A memória secundária, por outro lado, tem duas funções principais: cobrir o sistema de ficheiros e uma área de *swapping*.

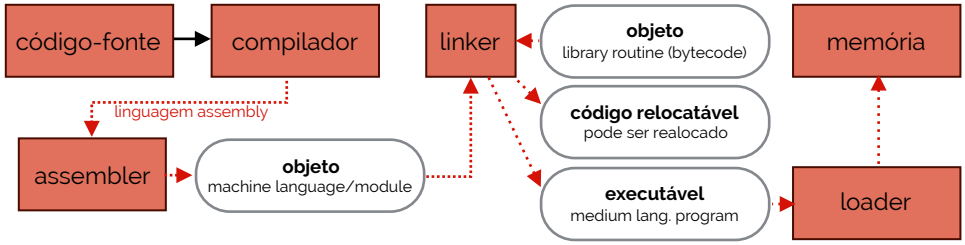
O tipo de organização que estamos a abordar baseia-se no pressuposto de que quanto mais afastado uma instrução, ou um operando, estiver do processador, menos vezes será referenciado. Nestas condições, o tempo médio de uma referência aproxima-se tendencialmente do valor mais baixo. A justificação deste pressuposto é o **princípio da localidade** (de referência) [9]. Aqui constata-se, heuristicamente, que o comportamento de um programa em execução, que estabelece que as referências à memória durante a execução de um programa, tendem a concentrar-se em frações bem definidas do seu espaço de endereçamento, durante intervalos mais ou menos longos.

princípio da localidade

Espaço de endereçamento

Em disciplinas como Arquitetura de Computadores I (a2s1) e Linguagens Formais e Autómatos (a2s2) estudámos o processo de assemblagem de um programa, como representado na Figura 4.2.

figura 4.2



Os **ficheiros objeto**, resultantes do processo de compilação, são **ficheiros relocatáveis**, isto é, ficheiros cujos endereços das diversas instruções e das constantes e variáveis são calculados a partir do início do módulo, por convenção o endereço 0. Estes

ficheiros objeto
ficheiros relocatáveis

ficheiros são depois parte da responsabilidade dos *linker*, dado que este terá de conseguir colecionar todos os ficheiros objeto num só ficheiro, denominado de **ficheiro executável**, resolvendo entre si as várias referências bibliotecas externas. As **bibliotecas** do sistema podem não ser incluídas no ficheiro executável para minimizar o seu trabalho, sendo que o *loader* irá construir a imagem binária do espaço de endereçamento do processo, combinando o ficheiro executável e, se for o caso, as bibliotecas de sistema, e resolvendo todas as referências externas que restam.

ficheiro executável
bibliotecas

Por oposição à situação anteriormente descrita, de *linkagem* estática, alguns sistemas de operação suportam a chamada **linkagem dinâmica**. Com a *linkagem* dinâmica cada referência no código do processo a uma rotina de sistema é substituída por um **stub**, que é um pequeno conjunto de instruções que determina a localização de uma rotina específica, se já estiver residente em memória principal, ou promove a sua carga em memória, caso contrário. Quando um *stub* é executado, a rotina associada é identificada e localizada em memória principal, o *stub* substitui então a referência ao seu endereço no código do processo, pelo endereço da rotina de sistema e executa-a. Quando essa zona de código for de novo atingida, a torna de sistema é agora executada diretamente. Todos os processos que usam uma mesma biblioteca de sistema, executam a mesma cópia do código, minimizando portanto a ocupação da memória principal.

linkagem dinâmica
stub

Organização de memória real

Como vimos no segundo capítulo deste documento, um processo tem que estar na memória para que possa ser executado. Um processo, no entanto, pode ser *swapped* temporariamente fora da memória para uma área especial e trazido de volta à memória para uma execução contínua [10].

A política de reserva de espaço em memória principal para armazenamento do espaço de endereçamento dos processos que correntemente coexistem, diz-se organizar uma **organização de memória real**, quando existe uma correspondência biunívoca (de um para um) entre o espaço de endereçamento de um processo e o seu espaço de endereçamento físico. O aspeto com que arrancámos esta secção é uma das consequências desta organização, juntamente com a limitação do espaço de endereçamento de um processo e a contiguidade do espaço de endereçamento físico que, embora não constitua uma condição necessariamente estrita, é naturalmente mais simples e eficiente supor que o espaço de endereçamento do processo é contíguo.

organização de memória real

Na comutação de processos, a operação de despacho (*dispatch*) carrega o **registo base** e o **registo limite** com os valores presentes nos campos correspondentes da entrada da tabela de controlo de processos, associada com o processo que vai ser agendado para execução. O registo base representa assim o endereço do início da região de memória principal onde está alojado o espaço de endereçamento físico do processo e o registo limite indica o tamanho, em bytes, do espaço de endereçamento.

registo base, registo limite

Sempre que há uma referência à memória, o endereço lógico é primeiro comparado com o conteúdo do registo limite, pelo que se for menor, então estamos perante uma referência válida, a qual ocorre dentro do espaço de endereçamento do processo, e o conteúdo do registo base é adicionado ao endereço lógico para produzir o endereço físico. Se, pelo contrário, for maior ou igual, trata-se de uma referência inválida, um acesso à memória nulo (*dummy cycle*) é posto em marcha e gera-se uma exceção por erro de endereço.

Na figura 4.3 podemos ver uma representação numa imagem da tradução de um endereço lógico num endereço físico.

A parte disponível da memória principal vai conter os espaços de endereçamento de diferentes processos. Quando um processo é criado, e posto no estado "criado", sendo inicializadas as estruturas de dados destinadas a geri-lo. A imagem binária do seu espaço de endereçamento é construída, o valor do campo registo limite da entrada da tabela de controlo de processos correspondente é determinado e, se houver espaço em memória principal, o seu espaço de endereçamento é carregado aí, o campo registo base é atualizado com o endereço inicial da região reservada e o processo é colocado na fila de espera dos

processos "prontos a executar". Caso contrário, o seu espaço de endereçamento é armazenado temporariamente na área de *swapping* e o processo é colocado na fila de espera dos processos "suspensos e prontos".

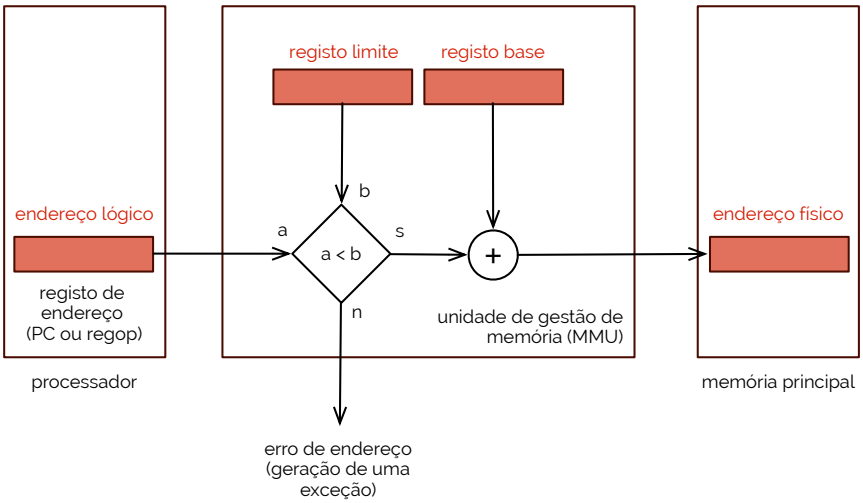


figura 4.3

Ao longo da sua execução, o espaço de endereçamento de um processo pode ser deslocado temporariamente para a área de *swapping*, passando o seu estado de "pronto para executar" para "suspenso e pronto", ou de "bloqueado" para "suspenso e bloqueado". Sempre que há espaço na memória, um dos processos presentes na fila de espera dos processos "suspenso e pronto" é selecionado, o seu espaço de endereçamento é carregado, o campo registo base da entrada da tabela de controlo de processos é atualizado com o endereço inicial da região reservada e o processo é colocado na fila de espera dos processos em "pronto a executar".

Episodicamente, se esta lista de espera estiver vazia e houver processos na fila de espera do processos "suspensos e bloqueados", um dles pode também ser selecionado, pelo que o mecanismo é semelhante ao descrito no ponto anterior, só que o processo é colocado agora na fila de espera dos processos "bloqueados".

Finalmente, quando um processo termina, passa para o estado "terminado" e o seu espaço de endereçamento é transferido para a área de *swapping*, se não estiver já lá, aguardando o fim das operações.

A parte disponível da memória é então dividida, concetualmente, num conjunto fixo de partições mutuamente exclusivas, não necessariamente iguais. Cada uma delas vai conter o espaço de endereçamento físico de um processo.

Diferentes disciplinas de escalonamento podem ser usadas na seleção do próximo processo cujo espaço de endereçamento vai ser carregado em memória principal. Se valorizarmos o critério de justiça, então devemos escolher o primeiro processo da fila de espera dos processos "suspensos e prontos" cujo espaço de endereçamento cabe na partição. Se, por outro lado, valorizarmos a ocupação da memória principal, devemos escolher o primeiro processo da fila de espera dos processos "suspensos e prontos" com o espaço de endereçamento de tamanho maior que cabe na partição.

Quando a última disciplina é aplicada, para se evitar o adiamento indefinido de processos com espaço de endereçamento pequeno, é comum associar um contador a cada processo na lista de espera e incrementá-lo sempre que for ultrapassado na seleção: ao atingir-se um valor pré-definido, o processo já não pode ser descartado e a primeira regra é aplicada.

Uma maneira de aumentar a taxa de ocupação da memória principal é supor que toda a parte disponível da memória constitui à partida um bloco único, e ir sucessivamente reservando regiões de tamanho suficiente para carregar o espaço de endereçamento dos processos que vão surgindo, e ir mais tarde libertando-as, quando deixaremde ser necessárias. A Figura 4.4 mostra exemplos desta aplicação.

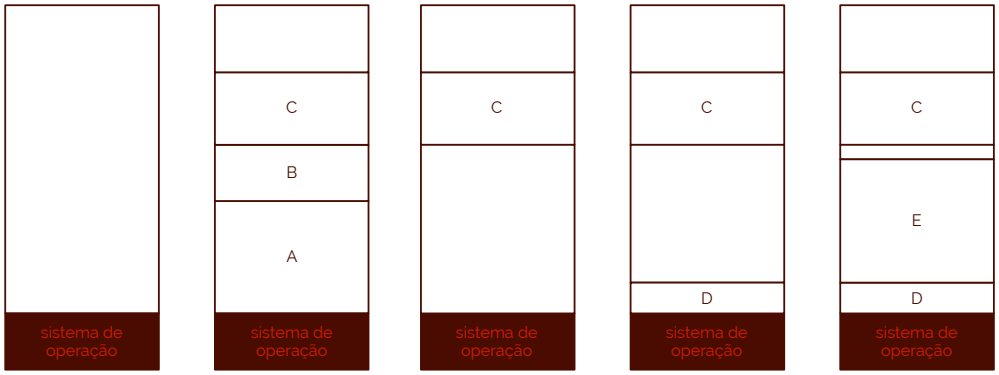


figura 4.4

Como a memória é reservada dinamicamente, o sistema de operação tem que manter um registo atualizado das regiões ocupadas e das regiões livres. Uma maneira de fazer isso é construindo duas listas biligadas: uma **lista das regiões ocupadas**, que localiza as regiões que foram reservadas para armazenamento do espaço de endereçamento dos processos residentes em memória principal; uma **lista das regiões livres**, que localiza as regiões ainda disponíveis para armazenamento do espaço de endereçamento dos novos processos que vão sendo criados, ou que venham a ser transferidos da área de *swapping*.

lista das regiões ocupadas

lista das regiões livres

Se a região de memória reservada fosse exatamente a suficiente para armazenamento do espaço de endereçamento do espaço de endereçamento do processo, corria-se o risco de formar regiões livres de tamanho tão pequeno que nunca poderiam ser utilizadas por si próprias, mas que seriam incluídas na lista das regiões livre, tornando pesquisas posteriores mais complexas. Assim, a memória principal é dividida tipicamente em blocos de tamanho fixo e a reserva de espaço é feita em unidades de tamanho desses blocos.

A valorização do critério de justiça é a disciplina de escalonamento geralmente adotada, sendo escolhido o primeiro processo da fila de espera dos processos "suspensos e prontos" cujo espaço de endereçamento pode ser colocado em memória principal.

O principal problema que se coloca numa arquitetura de partições variáveis, relaciona-se com o grau de fragmentação externa que é produzido na memória principal, pelas sucessivas reserva e libertações das regiões de armazenamento do espaço de endereçamento dos processos. Podem-se atingir situações em que, embora haja memória livre em quantidade suficiente, ela não é contínua e, por isso, o armazenamento do espaço de endereçamento de um novo processo deixa de ser possível.

Para resolver este problema, uma boa aproximação será aplicar uma **compactação do espaço livre** (em inglês *garbage collection*), agrupando todas as regiões livres num dos extremos da memória.

compactação do espaço livre

É, então, importante desenvolver-se métodos eficientes para a localização da região de memória principal que deve ser reservada para armazenamento do espaço de endereçamento de um processo. Entre tais métodos, destacam-se quatro:

- **first fit**, onde a lista das regiões livres é pesquisada desde o princípio até se encontrar a primeira região com tamanho suficiente;
- **next fit**, que sendo uma variante do *first fit*, consiste em iniciar a pesquisa a partir do ponto de paragem na pesquisa anterior;
- **best fit**, onde a lista das regiões livres é pesquisada na sua totalidade, escolhendo-se a região mais pequena de tamanho maior ou igual do que o espaço de endereçamento do processo;
- **worst fit**, onde a lista das regiões livres é pesquisada na sua totalidade, escolhendo-se a maior região existente.

first fit

next fit

best fit

worst fit

Em suma, a arquitetura de partições fixas é simples de implementar e eficiente (dado que não exige qualquer *hardware* ou estruturas de dados especiais e a seleção pode

ser feita muito rapidamente), no entanto conduz a uma grande fragmentação interna da memória principal e é criada para uma aplicação em específico. Do lado das arquiteturas de partições variáveis temos que a aplicação desta solução é perfeitamente geral, com uma implementação de baixa complexidade, mas que conduz a uma grande fragmentação externa da memória principal e é pouco eficiente.

Organização de memória virtual

Enquanto que os registos base e de limite podem ser usados para criar a abstração de espaços de endereçamento, existe ainda outro problema que terá de ser resolvido: gerir o **bloatware**. O *bloatware* é um *software* cuja utilidade é bastante reduzida, uma vez que requer memória e espaço em disco em excesso. Enquanto que o tamanho das memórias estão a aumentar rapidamente, o tamanho dos *softwares* estão a aumentar ainda mais. Nos anos '80 muitas universidades dos Estados Unidos tinham as suas máquinas a correr um sistema operativo em *timesharing* com dezenas de utilizadores a trabalharem em VAXs[7] de 4 MB. Agora a Microsoft recomenda ter, pelo menos, 2 GB para o seu sistema operativo Windows 10, de 64 bits [2].

bloatware

Como consequência de tais desenvolvimentos, existe a necessidade de correr programas que são muito grandes para caber em memória, e decerto existe a necessidade de haver sistemas possam suportar múltiplos programas a executarem em simultâneo, cada um cabendo em memória, mas todos juntos excedendo-a. Aqui o *swapping* não é uma solução ótima, dado que um disco típico SATA, como estudámos em Arquitetura de Computadores II (a2s2), possui uma transferência de pico de algumas centenas de MB/segundo, o que significa que demoraria segundos a fazer *swap* de um programa de 1 GB (quer *swap in*, quer *swap out*).

A política de reserva de espaço em memória principal para armazenamento do espaço de endereçamento dos processos que correntemente coexiste, diz-se originar uma organização de memória virtual quando o espaço de endereçamento lógico de um processo e o seu espaço de endereçamento físico estão totalmente disassociados. Como consequências disto poderá ocorrer a não-limitação do espaço de endereçamento de um processo, a não-contiguidade do espaço de endereçamento físico e uma área de *swapping*.

Na Figura 4.5 podemos então ver como é que se processa a tradução de um endereço lógico num endereço físico.

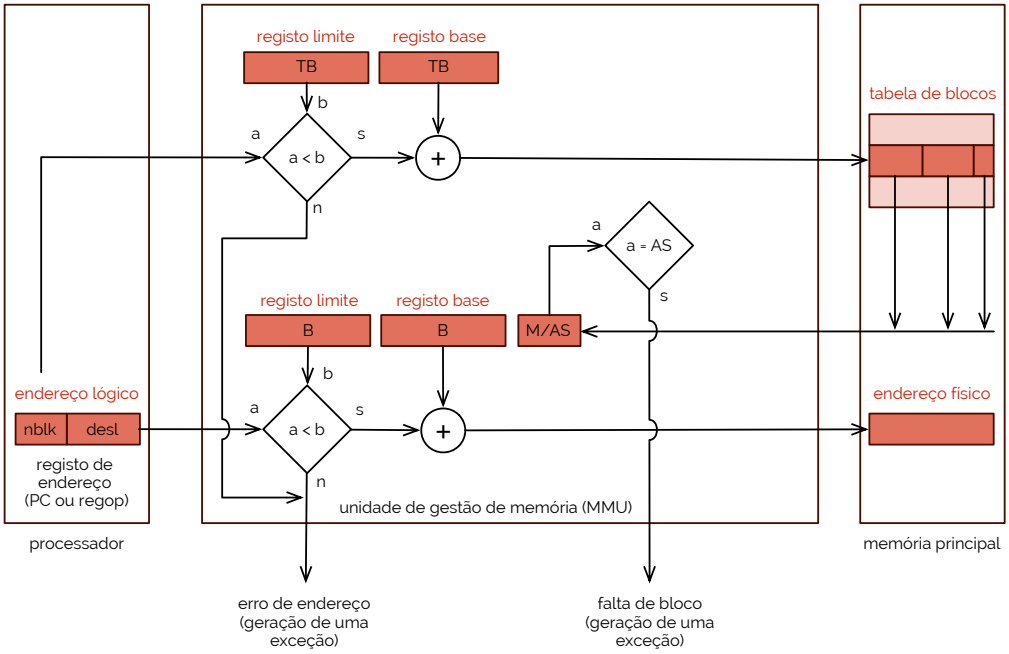


figura 4.5

7 VAX é um ISA (*Instruction Set Architecture*) descontinuado, do tipo CISC, acrónimo de *Virtual Address EXtension*.

O **endereço lógico** é então formado por dois campos, o *nblk*, que identifica um bloco especíico, e o *desloc*, que localiza uma posição de memória concreta dentro do bloco, através do cálculo da distância ao seu início. A unidade de gestão de memória (MMU) contém agora dois pares de registos base e limite - um que está associado com a **tabela de blocos** (estrutura de dados que fornece a descrição da localização dos vários blocos em que o espaço de endereçamento do processo está dividido) e outro que está associado com a descrição de um bloco particular, cuja referência está a decorrer num dado momento.

endereço lógico

tabela de blocos

Quando ocorre uma comutação de processos, a operação de despacho (*dispatch*) carrega o registo base e o registo limite da tabela de blocos com os valores presentes nos campos correspondentes da entrada da tabela de controlo de processos associada com o processo que vai ser agendado para execução. O valor do registo base da tabela de blocos representará assim o endereço do início da região de memória principal onde está alojada a tabela de blocos do processo e o valor do registo limite está relacionado com o número de entradas da tabela.

Esta referência à memória ocorrerá assim em três fases. Numa primeira fase, o campo *nblk* do endereço lógico é comparado com o valor do registo limite da tabela de blocos. Se o endereço for menor, então *nblk* será adicionado ao conteúdo do registo base da tabela de blocos para produzir o endereço da entrada da tabela de blocos descritiva do bloco. Caso contrário, se for maior ou igual, a referência é inválida, sendo que acontecerá um acesso à memória nulo (num *dummy cycle*) e gera-se uma exceção por erro de endereço.

Numa segunda fase avalia-se o valor do registo *M/AS*. Se o registo contiver o valor *M*, então os campos da entrada da tabela de blocos referenciada são transferidos para os registos respetivos da unidade de gestão de memória. Caso o registo contiver o valor *AS*, então o bloco não está atualmente presente na memória principal, pelo que a instrução é finalizada com um acesso à memória nulo e gera-se uma exceção por falta de bloco.

Numa terceira e última fase, o campo *desloc* do endereço lógico é comparado com o valor do registo limite do bloco, pelo que se for menor, então trata-se de uma referência válida, sendo que ocorre dentro do espaço de endereçamento do bloco, e *desloc* é assim adicionado ao conteúdo do registo base do bloco para produzir o endereço físico. Caso contrário, isto é, se o for maior ou igual, então trata-se de uma referência inválida, o que provocará um acesso à memória nulo, gerando uma exceção por erro de endereço.

O aumento da versatilidade na gestão do espaço em memória principal, introduzido por umaorganização de memória virtual, tem um custo que se traduz na transformação de cada acesso à memória em dois acessos. Num primeiro acesso é referenciada a entrada da tabela de blocos do processo, associada com o bloco descrito no campo *nblk* do endereço lógico, para se obter o endereço do início do bloco em memória. Já num segundo acesso, é referenciada a posição de memória específica, sendo o cálculo do seu endereço feito adicionando o campo *desloc* do endereço lógico ao endereço do início do bloco em memória.

Concetualmente, a organização de memória virtual resulta num fracionamento do espaço de endereçamento lógico do proceso em blocos que são tratados dinamicamente como subespaços de endereçamento autónomos numa organização de memória real (de partições fixas, se os blocos tiverem todos o mesmo tamanho, ou de partições variáveis, se puderem ter tamanho diferente). O que há de novo é a possibilidade de ocorrer um acesso a um bloco atualmente não-residente em memória principal, com a consequente necessidade de anulação do acesso e a sua repetição mais tarde, quando ele for carregado.

A necessidade deste duplo acesso à memória pode ser minimizada tirando partido do princípio da localidade de referência [9]. Como os acessos tenderão a estar concentrados num conjunto bem definido de blocos durante intervalos de tempo alargados de execução do processo, a unidade de gestão de memória mantém habitualmente armazenado numa memória associativa interna, designada de **translation lookaside buffer** (TLB) o conteúdo das entradas da tabela de blocos que foram ultimamente referenciadas (vide Arquitetura de Computadores II (a2s2)). Deste modo, o primeiro acesso poderá resultar num **hit**, quando a entrada está armazenada na TLB (caso em que o acesso é interno ao proces-

translation lookaside buffer

hit

sador), ou num **miss**, quando a entrada não está armazenada na TLB (caso em que há um acesso externo à memória principal. **miss**

O tempo médio de acesso a uma instrução, ou um operando, aproxima-se assim tendencialmente do valor mais baixo (um acesso ao TLB e um acesso à memória principal).

A parte disponível da memória principal vai conter porções dos espaços de endereçamento de diferentes processos. Quando um processo é criado, é posto no estado "criado", sndo inicializadas as estruturas de dados destinadas a geri-lo: a imagem binária do seu espaço de endereçamento é construída, sendo transferida para a área de *swapping*, pelo menos, a sua parte variável; a tabela de blocos associada é organizada a seguir; se houver espaço, a tabela de blocos, o primeiro bloco de código do processo e o bloco da *stack* são carregados em memória principal, as entradas correspondentes da tabela de blocos são atualizadas com o endereço inicial das regiões reservadas e o processo é colocado na fila de espera dos processos "pronto a executar". Caso contrário, o processo é colocado na fila de espera dos processos "suspensos e prontos".

Ao longo da execução do processo, sempre que ocorra um acesso a um bloco não-residente em memória principal, o processo é posto no estado "bloqueado", enquanto ocorre a transferência do bloco da área de *swapping* para a memória principal. Quando esta transferência terminar, o processo é novamente colocado na fila de espera dos processos "prontos a executar".

Todos os blocos residentes em memória principal, e pertencentes a um mesmo processo, podem ser deslocados temporariamente para a área de *swapping*, passando o seu estado de "pronto para execução" para "suspenso e pronto", ou de "bloqueado" para "suspenso e bloqueado".

Sempre que há espaço na memória, um dos processos presentes na fila de espera dos "suspensos e prontos" é selecionado, a tabela de blocos e um grupo de blocos do seu espaço de endereçamento são carregados, as entradas correspondentes da tabela de blocos são atualizadas com os endereços iniciais das regiões reservadas e o processo é colocado na fila de espera dos processos "prontos a executar".

Se a lista de "suspensos e prontos" estiver vazia e houver processos na fila de espera dos processos "suspensos e bloqueados", um deles pode também ser selecionado. O mecanismo é semelhante ao descrito no parágrafo anterior, só que o processo é agora colocado na fila de espera dos processos "bloqueado".

Finalmente, quando um processo termina, passa para o estado "terminado" e a imagem do seu espaço de enderaçamento residente na área de *swapping*, ou pelo menos da sua parte variável, é atualizada com a consequente libertação de todos os blocos existentes em memória principal, aguardando o fim das operaçoões.

O sistema de operação deve então providenciar meios para resolver o problema de referência a um endereço localizado num bloco que não está atualmente presente em memória. Como foi referido atrás, a MMU gera nestas circunstâncias uma exceção por falta de bloco, a qual aciona uma rotina de serviço que faz a seguinte sequência de operações:

1. salvaguardar o contexto do processo na entrada correspondente da tabela de controlo de processos, colocando o seu estado em "bloqueado" e atualizando o *program counter* para o endereço que produziu a falta de bloco;
2. determinar se existe espaço em memória principal para carregar o bloco em falta:
   - 2.1. caso exista, selecionar uma região livre;
   - 2.2. caso não exista, selecionar uma região cujo bloco vai ser substituído:
     - 2.2.1. se o bloco tiver sido modificado, proceder à sua transferência para a área de *swapping*;
     - 2.2.2. atualizar a entrada da tabela de blocos do processos a que o bloco pertence, com a indicação de que o bloco já não está residente em memória;

3. transferir o bloco em falta da área de *swapping* para a região selecionada;
4. invocar o escalonador para agendar para execução um dos processos da fila de espera dos processos "prontos a executar";
5. quando a transferência estiver concluída, atualizar a entrada da tabela de blocos do processo com a indicação de que o bloco está residente em memória e de qua é a sua localização, e mudar o seu estado para "pronto para executar", colocando-o na fila de espera correspondente.

O problema de programas maiores que a memória instalada num computador tem sido discutido desde a criação dos computadores, embora de formas limitadas, como na ciência e engenharia. Uma solução adotada nos anos 1960 foi de partir os programas em várias peças, chamadas de **sobreposições** (*overlays*). Quando um programa desses arrancava, seria carregado para a memória um gestor das sobreposições, que imediatamente carregava e executava a sobreposição número 0. Quando esta estivesse pronta, então haveria uma troca de mensagens e o gestor trocaria a sobreposição 0 pela 1, sobrepondo os conteúdos da memória.

sobreposições

Embora pareça uma solução viável, partir programas aos pedaços é uma solução morosa, cansativa e passível de cometer erros. Além disso, poucos eram os programadores que conseguiam fazê-lo com sucesso.

O método que foi entretanto, em 1961, escolhido como alternativo, foi um que passou a ser conhecido como **memória virtual**. Desenhado inicialmente por Fritz-Rudolf Güntsch [11], a ideia básica por detrás deste conceito é que cada programa tem o seu próprio espaço de endereçamento, que é particionado naquilo a que chamamos de **páginas**. Tal como nas Páginas Brancas, cada página terá um intervalo contíguo de endereços. Estas páginas são depois mapeadas com a memória física, mas nem todas têm de estar na memória física enquanto o programa estiver em execução. Quando o programa referencia uma parte do espaço de endereçamento físico, o *hardware* faz o mapeamento necessário de forma muito veloz. Mas quando o programa referencia uma parte do espaço de endereçamento que não está na memória física, então o sistema de operação é alertado para ir buscar a parte que falta e tentar a instrução de novo. Por um lado, a memória virtual é uma generalização da ideia de registo base e registo limite.

memória virtual
Fritz-Rudolf Güntsch
páginas

O conceito de memória virtual também funciona muito bem com sistemas multiprogramáveis, com bits e partes de vários programas na memória em simultâneo. Enquanto um programa aguarda pelas suas partes para ser lido, o CPU poderá ser dado a um outro processo, para outras execuções

Grande parte dos sistemas operativos com memória virtual usam uma técnica denominada de **paginação**. Em qualquer computador, os programas referenciam um conjunto de endereços de memória. Quando um programa executa um código como `move reg, 1000` ele fá-lo copiando os conteúdos do endereço de memória 1000 para `reg` (ou vice-versa, dependendo do computador). Os endereços podem ser gerados usando indexação, registos base, segmentos de registos, entre outros...

paginação

Estes endereços gerados pelos programas são chamados **endereços virtuais** e formam o espaço de endereçamento virtual. Em computadores sem memória virtual, o endereço virtual é colocado diretamente no bus de memória e isto faz com que seja lida ou escrita a palavra com o mesmo endereço na memória física. Quando a memória virtual é usada, os endereços virtuais não se ligam diretamente ao bus de memória, mas antes à unidade de gestão de memória, MMU, que tem o trabalho de mapear os seus endereços em endereços físicos.

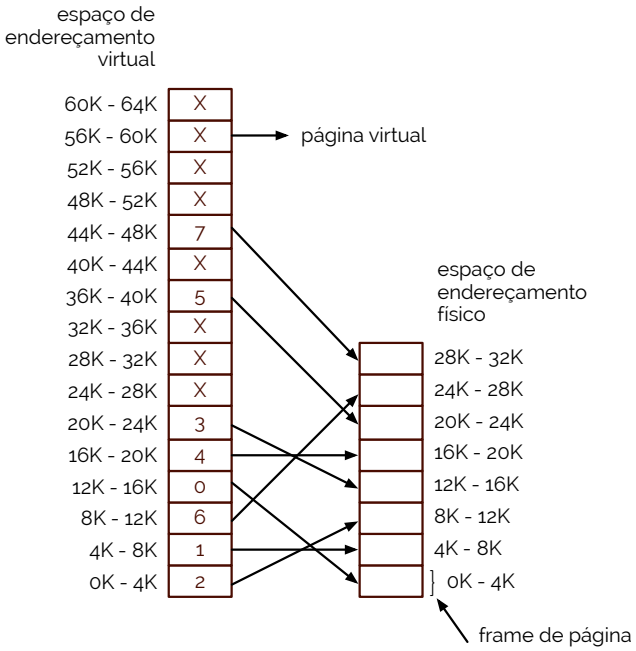
endereços virtuais

Um exemplo muito simples de como é que este mapeamento funciona [2] é dado na Figura 4.6. Neste exemplo, temos um computador que gera endereços de 16 bits, de 9 até 64K - 1 - estes são os endereços virtuais. Este computador, no entanto, apenas tem 32 kB de memória física. Assim, embora os programas de 64 kB possam ser escritos, eles não podem ser carregados diretamente e na totalidade na memória e correr. Como o programa está completamente acessível no disco, podemos ir buscar as várias peças necessárias à sua execução, à medida do necessário.

O espaço de endereçamento virtual consiste assim em unidades de tamanho fixo denominadas de páginas. Às unidades correspondentes às páginas, mas na memória física, damos o nome de **frames de página** e geralmente têm o mesmo tamanho que as páginas. Neste exemplo eles são 4 kB, mas o tamanho das páginas variam entre 512 bytes e 1 GB em sistemas reais. Com 64 kB de espaço de endereçamento e 32 kB de memória física, obtemos 16 páginas virtuais e 8 frames.

frames de página

figura 4.6



As transferências entre a memória RAM e o disco são sempre feitas na totalidade. Muitos processadores suportam tamanhos múltiplos de página que inclusive podem ser misturados e correspondidos consoante for conveniente para o sistema de operação.

A notação usada na Figura 4.6 é a seguinte: os intervalos marcados 0K - 4K significa que o endereçamento físico ou virtual nessa página está entre 0 e 4095. O intervalo 4K - 8K refere-se aos endereços de 4096 a 8191, e por assim em diante... Cada página contém precisamente 4096 endereços a começar por um múltiplo de 4096.

Assim, quando um programa tenta aceder ao endereço 0, usando uma instrução, por exemplo, `move reg, 0`, o endereço virtual 0 é enviado para a MMU. A MMU identifica que se trata de um endereço virtual e que este cai na página 0 (de 0 a 4095), o que de acodo com o seu mapeamento, é o frame de página 2 (8192 a 12287). Sabendo isto, transforma o endereço para 8192 e coloca o endereço no bus de memória. Como a memória desconhece da MMU, apenas vê um pedido de acesso a um determinado endereço de memória.

Com o mesmo exemplo de aplicação, a instrução `move reg, 8192` é traduzida para `move reg, 24576`, porque o endereço virtual 8192 (na página virtual 2) é mapeada em 24576 (no frame físico 6). Como um terceiro exemplo, o endereço virtual 20500 está 20 bytes depois do início da página virtual 5 (endereços virtuais 20480 até 24575) e mapeia-se no endereço físico 12288 + 20 = 12308.

Por si só, esta habilidade de mapear as 16 páginas virtuais num dos 8 frames de página questionando a MMU não resolve o problema do espaço de endereçamento virtual ser bem maior que o espaço de endereçamento físico. Sendo que nós só temos oito frames de página, somente oito páginas virtuais é que estarão mapeadas na nossa memória física. As outras, apresentadas na Figura 4.6 com um 'X' não estão mapeadas. Num *hardware* real, o que acontece que é que existe um bit, denominado de **present/absent bit**, que controla quais são as páginas que estão fisicamente presentes na memória.

present/absent bit

Então e o que é que acontece quando tentamos aceder ao endereço virtual 32780? Portanto, 32780 há de ser o byte 12 dentro da página virtual 8 (que começa em 32768?). A

MMU irá então reparar que esta página não possui qualquer mapeamento (indicado por um 'X' na figura) e provoca que o CPU lance uma `trap` para o sistema operativo. Esta "rasteira" é denominada de **page fault**. O sistema operativo escolhe um frame pouco usado e escreve o seu conteúdo de volta no disco (se já lá não estiver). De seguida, carrega a página que acaba de ser referenciada para a página que acaba de libertar, altera o mapeamento, e reinicia a instrução que provocou o `trap`.

page fault

Olhemos agora para a MMU e tentemos perceber porque é que escolhemos um tamanho de página que é potência de base 2. Na Figura 4.7, podemos ver um exemplo de endereço, o 8196 (0010 0000 0000 0100 em binário), a ser mapeado usando o mapeamento MMU da Figura 4.6. Os 16 bits de endereço virtual são assim separados num **número de página** de 4 bits e num deslocamento de 12 bits. Com 4 bits para o número de página, podemos ter até 16 páginas e, com 12 bits de deslocamento, podemos endereçar até 4096 bytes dentro de uma só página.

número de página

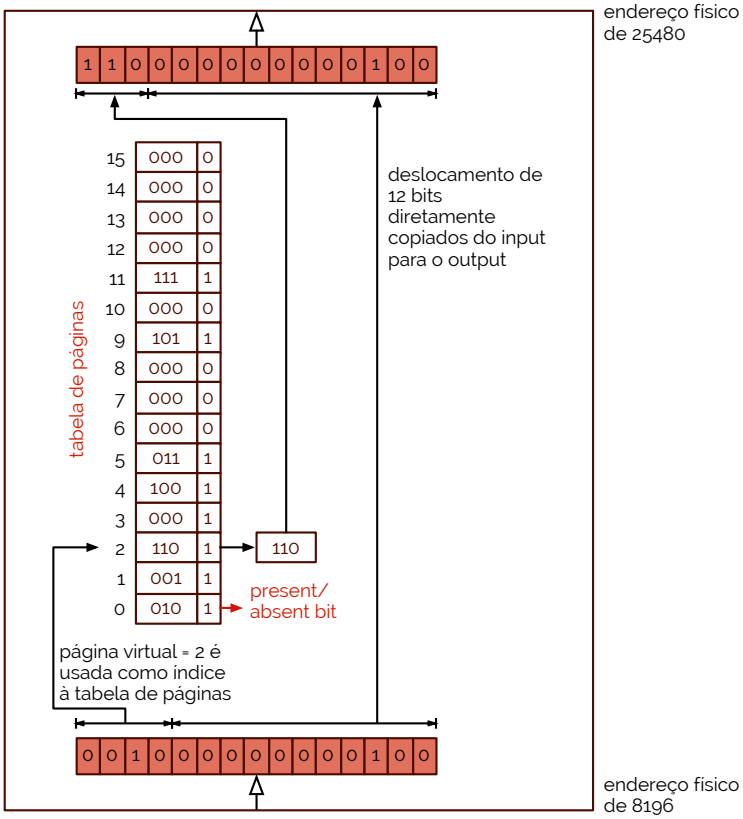


figura 4.7

O número de página é usado como um índice na **tabela de páginas**, fazendo corresponder o número a um frame de página, novamente, correspondente a uma determinada página virtual. Se o bit *present/absent* esiver a '0', então será provocada uma `trap` ao sistema operativo. Se este bit for 1, então o número de frame encontrado na tabela de páginas será copiado para os 3 bits mais significativos do registo de saída, juntamente com o deslocamento de 12 bits, que são copiados sem serem modificados, do endereço de entrada, virtual. Juntos, então, formam um endereço de 15 bits, físico. O registo de saída é então colocado no bus de memória, como endereço físico de memória.

tabela de páginas

Numa implementação simples, o mapeamento de um endereço virtual num endereço físico pode ser sumariado da seguinte forma [2]. O endereço virtual é dividido num número de página virtual (bits mais significativos) e num deslocamento (bits menos significativos). Por exemplo, com um endereço de 16 bits e um tamanho de página de 4 kB, os 4 bits mais significativos poderiam especificar uma das 16 páginas virtuais e os 12 bits menos significativos poderiam especificar o deslocamento (entre 0 e 4095), na página selecionada. No entanto, uma divisão com 3 ou 5 (ou outras quantidades) de bits para a pági-

na também são possíveis - divisões diferentes implicam, claro está, diferentes tamanhos de página.

O número de uma página virtual é usado como índice para a tabela de páginas, de forma a encontrar a entrada para uma determinada página virtual. Da entrada da tabela, o número do frame é encontrado, sendo anexado ao extremo de maior ordem do deslocamento, substituindo o número de página virtual, de forma a criar um endereço físico que possa ser enviado para a memória. Assim, o propósito da tabela de páginas é mapear as páginas virtuais nos seus frames. Matematicamente falando, é como se de uma função estivessemos a falar. Usando o resultado desta função, o campo da página virtual (num endereço virtual) poderá ser substituído pelo campo de frame, por conseguinte, formando um endereço físico.

Em termos mais concretos, uma entrada da tabela de páginas tem a forma que está exibida na Figura 4.8 [10].

figura 4.8

A estrutura de um entrada de página é altamente dependente da máquina, mas o tipo de informação que está presente costuma ser a mesma, de máquina para máquina. O campo mais importante de todos é o número de bloco em *swapping* (que fornece a localização da página na área de *swapping*, se já lhe foi atribuído espaço) e o número do frame na memória (localização da página, se residente em memória principal). No fundo, estes dois campos globalizam-se num só respetivo ao número de frame de página, com um bit próprio (o mais significativo) para *present/absent*.

Os bits `Perm` indicam que tipos de acesso é que são permitidos. Na sua forma mais simples, este campo contém 1 bit, sendo 0 para leitura e escrita e 1 para só leitura. Também poderá ter uma descrição mais detalhada, com a forma `rwx`, com sinalização em separado de acesso para leitura e/ou escrita (operandos) ou de execução (instruções).

O bit de modificação (`Mod`) e de referência (`Ref`) mantêm um controlo da utilização da página. Quando uma página está a ser escrita, o *hardware* automaticamente o deteta e coloca o bit a *set*. Este bit tem o ser valor quando o sistema de operação decide reclamar um frame de página. Se a página tiver sido modificada (isto é, se está "suja"), então deverá ser re-escrita no disco. Caso contrário (isto é, se está "limpa"), então poderá ser simplesmente abandonada, sendo que a cópia em disco está ainda válida. A este bit algumas vezes dá-se o nome de **dirty bit**, dado que reflete o estado da página. O bit de referência é colocado a *set* sempre que uma página é referenciada, quer seja para leitura ou para escrita. O seu valor é usado para ajudar o sistema de operação a escolher uma página, de forma a evitar *page faults*.

dirty bit

Finalmente, os campos M/AS e O/L são dois bits que sinalizam se a página está ou não residente em memória e a ocupação ou não desta entrada (a não ocupação significa que ainda não foi reservado espaço na área de *swapping* para esta página), respetivamente.

Em suma, e dando continuidade às figuras anteriores, na Figura 4.8 podemos ver uma possível representação de uma arquitetura paginada, da forma que explicitámos ao longo desta secção. Como vimos, e em comparação com as arquiteturas anteriores, temos que esta é mais vantajosa por ter um âmbito de aplicação mais global, fazer um maior aproveitamento da memória principal - não conduzindo à fragmentação externa ou interna que é praticamente desprezável - e não exige requisitos especiais de *hardware*. Por outro lado, o acesso à memória acaba por ser mais longo, dado o duplo acesso por consulta prévia da tabela de paginação (este aspeto poder ser minimizado, contudo, como já abordámos, se a MMU contiver TLB, para armazenamento das entradas da tabela de paginação recentemente mais referenciadas. Mais, a operacionalidade, dada a interação de tantos componentes, é muito exigente, o que torna mais difícil a tarefa de aumentar a eficiência.

Como alternativa a esta arquitetura paginada, surge ainda uma **arquitetura segmentada**. A arquitetura paginada divide o espaço de endereçamento lógico do processo de

arquitetura segmentada

uma forma cega, praticamente sem usar qualquer informação sobre a estrutura subjacente. A exceção é, como foi referido atrás, a prática habitual de colocar sempre no início de uma nova página (no fim, no caso da *stack*) cada uma das regiões funcionalmente distintas. Isto provoca algumas consequências: a estrutura modular que está na base do desenvolvimento de uma aplicação com alguma complexidade, não é tida em conta e, por consequência, não é possível usar o princípio da localidade de referência de modo a minimizar o número de páginas que tenham que estar residentes em memória principal em cada etapa de execução do processo; a gestão do espaço disponível entre a zona de definição dinâmica e a *stack* torna-se complicada e pouco eficiente, sobretudo, quando há a possibilidade de surgirem em tempo de execução múltiplas regiões de dados partilhados de tamanho variável, ou estruturas de dados de crescimento contínuo.

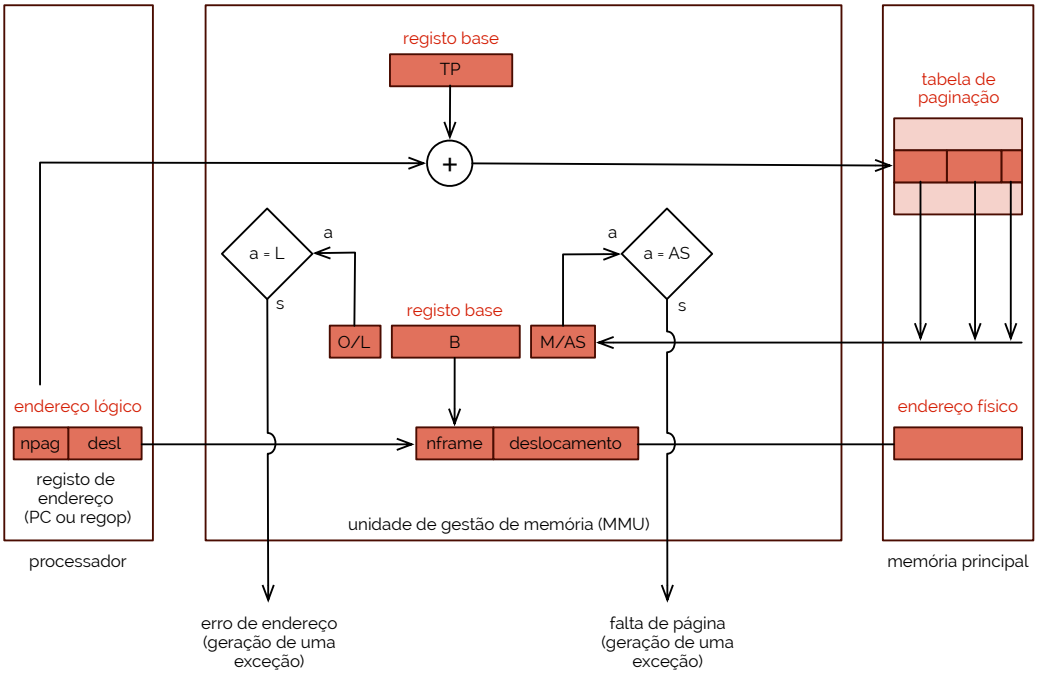


figura 4.9

Uma solução para o problema é desdobrar-se o espaço de endereçamento lógico do processo numa multiplicidade de espaços de endereçamento lineares autónomos definidos na fase de *linkagem*. Assim, casda módulo da aplicação vai originar dois espaços de endereçamento autónomos: um para o código e outro, na zona de definição estática, para as variáveis globais à aplicação (definidas localmente) e para as variáveis localmente globais (internas ao módulo).

Cada um destes espaços de endereçamento autónomos designa-se de **segmento** e uma organização de memória virtual, baseada neste tipo de decomposição, constitui a chamada arquitetura segmentada.

segmento

A arquitetura segmentada na sua versão mais pura tem pouco interesse prático, porque, tratando a memória principal como um espaço contínuo, exige a aplicação de técnicas de reserva de espaço para carregamento de um segmento em memória que são em tudo semelhantes às usadas em organizações de memórial real de partições virtuais. Daqui resulta uma enorme fragmentação externa da memória principal com o consequente desperdício de espaço.

Além disso, os segmentos de dados de crescimento contínuo conduzem a problemas adicionais: são facilmente concebíveis situações em que um acréscimo de tamanho do segmento não poderá ser realizado na sua localização presente, originando a sua transferência na totalidade para outra região da memória, ou, num caso limite em que não há espaço suficiente, ao bloqueio, ou suspensão, do processo, com a remoção do segmento, ou de todo o espaço de endereçamento residente, para a área de *swapping*.

Poderá ainda ser feita, para resolver estes problemas, uma versão híbrida de arquitetura paginada com segmentada, apresentada na Figura 4.9.

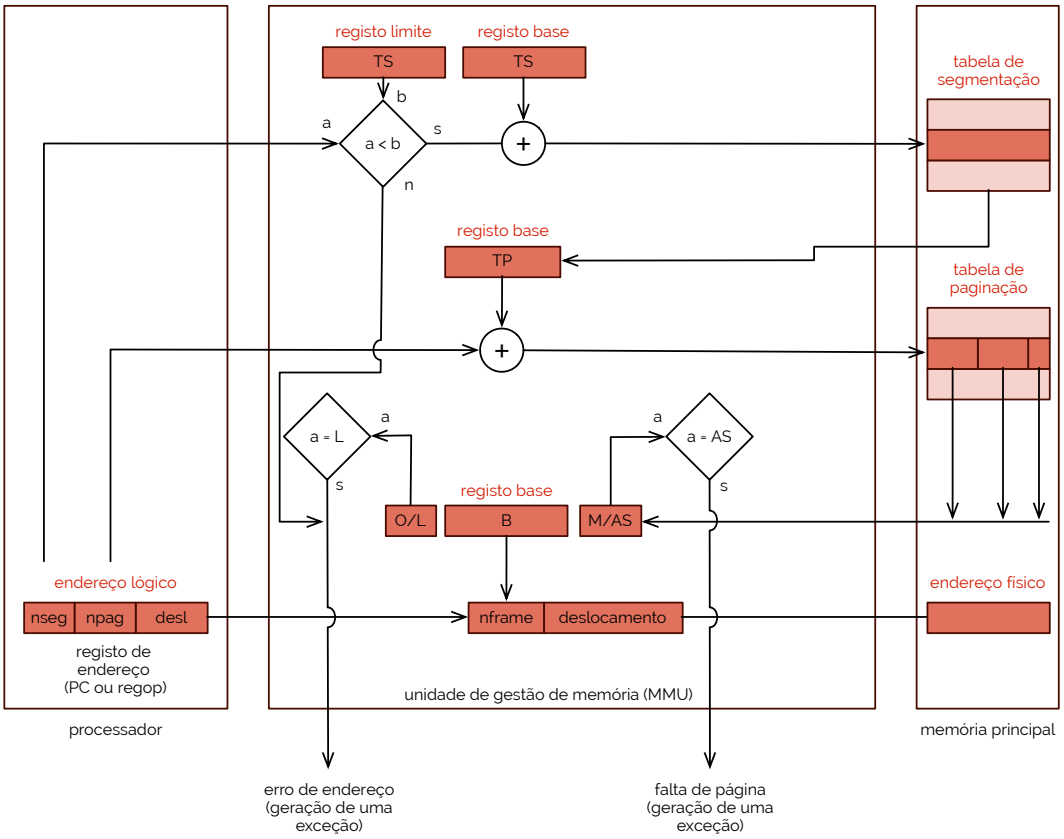


figura 4.10

Nesta arquitetura tem-se que a divisão do espaço de endereçamento lógico é primeiramente segmentada, com a atribuição, na fase de *linkagem*, de múltiplos espaços de endereçamento lineares autónomos e, cada um destes espaços, é dividido depois em páginas, originando um mecanismode carregamento de blocos em memória principal com todas as caraterísticas da arquitetura paginada.

Esta abordagem tem as mesmas vantagens que a versão paginada, mas com a melhoria da minimização do número de páginas que têm de estar residentes em memória principal, por cada etapa de execução do processo. Não obstante, também traz outras desvantagens, entre as quais uma maior exigência especial de *hardware*, um acesso à memória mais longo (agora com base num triplo acesso) e uma operacionalidade muito exigente.

Políticas de substituição de páginas em memória

Numa arquitetura paginada ou segmentada/paginada a memória principal é vista como dividida operacionalmente em frames do tamanho de cada página. Cada frame vai, em princípio, permitir o armazenamento do conteúdo de uma página do espaço de endereçamento lógico de um processo. As páginas podem assim estar em dois estados distintos: num estado **bloquado** (*locked*), quando não podem ser removidas da memória, como o caso das páginas associadas com o *kernel* do sistema de operação do *buffer cache* do sistema de ficheiros ou de um ficheiro mapeado em memória; ou num estado **desbloqueado** (*unlocked*), quando podem ser removidas da memória, como o caso das páginas associadas com os processos convencionais [10].

bloqueado

desbloqueado

Os frames ocupados, associados com as páginas desbloqueadas, estão organizados numa lista biligada que descreve os frames passíveis de substituição, e, de forma semelhante, os frames livres, também estão organizados numa lista biligada. O tipo de memória implementado pela lista dos frames ocupados depende do algoritmo de substituição utilizado.

Quando ocorre um *page fault*, a situação mais comum é a lista dos frames livres estar vazia e, por isso, torna-se necessário selecionar um frame para substituir da lista dos frames ocupados. Alternativamente, poder-se-á manter sempre na lista dos frames livres alguns frames, sendo um deles usado para carregar a própria página em falta, e proceder-se em seguida à substituição de um frame ocupado. Como as operações decorrerm em paralelo, este segundo método é mais eficiente.

Mas que frame é que escolhemos para a substituição? Teoricamente, deveria ser um frame que não irá mais ser referenciado ou, sendo-o, sê-lo-á o mais tarde possível. A este princípio damos o nome de **princípio da otimalidade**. Minimiza-se assim a ocorrência de outros *page faults*. O princípio da otimalidade é, porém, um princípio não casual e não pode ser diretamente implementado. Procura-se, assim, encontrar estratégias de substituição que sejam realizáveis e que, ao mesmo tempo, se aproximem tanto quanto possível do princípio da otimalidade.

princípio da otimalidade

Uma boa aproximação é feita seguindo a estratégia **LRU** (sigla de *Least Recently Used*), que visa encontrar o frame que não é referenciado há mais tempo.

LRU

Assumindo o princípio da localidade de referência, se um frame não é referenciado há muito tempo, é fortemente provável que também não o venha a ser no futuro próximo. Assim sendo, cada referência à memória tem que ser sinalizada com o instante da sua ocorrência (conteúdo de um temporizador ou de um contador), pelo que a MMU tem de ter capacidade para o fazer ou deverá existir *hardware* específico que terá de ser acoplado. Quando ocorre um *page fault*, a lista ligada dos frames ocupados tem de ser percorrida para determinar aquele cujo último acesso foi realizado há mais tempo. Esta solução transporta um custo de implementação, portanto, elevado e pouco eficiente.

Se virmos as coisas pelo lado oposto, podemos aproximarmo-nos de uma solução usando uma estratégia **NRU** (do inglês *Not Recently Used*), onde são apenas usados os bits `Ref` e `Mod` que são processados tipicamente por uma MMU convencional. Periodicamente, o sistema de operação percorre a lista dos frames ocupados e coloca a zero o bit `Ref`. Assim, quando ocorre uma *page fault*, os frames da lista de frames ocupados enquadram-se numa das classes da tabela da Figura 4.11.

NRU

classe	Ref	Mod
0	0	0
1	0	1
2	1	0
3	1	1

figura 4.11

A seleção da página a substituir será então feita entre aquelas pertencentes à classe de ordem mais baixa existente atualmente na lista dos frames ocupados.

Um outro critério, desta vez, baseado no tempo de estadia das páginas em memória principal, isto é, no pressuposto de que quanto mais tempo as páginas residirem em memória, menos provável será que elas sejam referenciadas a seguir. Consideremos assim que a lista dos frames ocupados está organizada numa fila que reflete a ordem de carregamento das páginas correspondentes em memória principal. Quando acontece um *page fault* retira-se da fila o elemento correspondente à página que está há mais tempo em memória.

Neste caso o próprio pressuposto é extremamente falível, dado que poderá ficar robusto se alterarmos o que acontece em caso de *page fault*. Assim, consideremos que quando tal cenário acontece, retira-se da fila o elemento correspondente à página há mais tempo em memória e, se o seu bit `Ref` estiver a zero, a página associada é escolhida para substituição. Caso contrário, coloca-se a zero o bit `Ref`, o nó é reintroduzido no fim da fila e o processo repete-se. A este algoritmo damos o nome de **algoritmo da segunda oportunidade**, nome o qual advém da segunda oportunidade dada a uma página antes de ser substituída.

algoritmo da segunda oportunidade

Se ao algoritmo anterior, que é baseado numa fila, tiver a sua estrutura de dados base convertida numa lista circular, e não linear, chegamos ao **algoritmo do relógio**, sobre o qual as operações de *in* e de *out* passam a ser meros incrementadores de um ponteiro. Aqui, quando ocorre um *page fault*, enquanto o bit `Ref` do frame apontado pelo ponteiro não for zero, este é colocado a zero e o ponteiro avança uma posição. Neste estado, a página apontada é escolhida para substituição e o ponteiro avança uma posição.

algoritmo do relógio

Quando um processo é colocado pela primeira vez na fila de espera dos processos "prontos a executar", apenas a primeira e a última páginas do seu espaço de endereçamento são colocadas em memória principal. No momento em que o processador é atribuído ao processo, suceder-se-ão inicialmente *page faults* a um ritmo relativamente rápido e, depois, o processo entrará numa fase mais ou menos longa, onde a execução decorrerá sem mais sobressaltos. Está-se então perante uma situação em que, de acordo com o princípio da localidade de referência, as páginas associadas com a fração do espaço de endereçamento que o processo está atualmente a referenciar estão todas presentes em memória principal. A este conjunto de páginas damos o nome de **working set** (de um processo).

working set

Ao longo do tempo o *working set* do processo vai variar, não só no que respeita ao número, como às páginas concretas que o definem. Se o número de frames em memória principal atribuído ao processo é fixo e inferior à dimensão do seu *working set* atual, então o processo está continuamente a gerar *page faults* e o seu ritmo de execução é muito lento, dizendo-se que está em **thrashing**. Caso contrário, o processo alternará período curtos em que sofrerá um conjunto de *page faults* a um ritmo muito elevado, com períodos mais ou menos longos em que elas quase que não ocorrem [12].

thrashing

Tentar manter o *working set* do processo sempre presente em memória principal constitui, assim, o objetivo prioritário de qualquer política de substituição. Um meio de consegui-lo é atribuindo novos frames ao processo sempre que ele se encontre num período de ritmo elevado de *page faults* e ir-lhe retirando os frames recentemente não referenciados em períodos de acalmia [10].

Quando um processo é introduzido na fila de espera dos processos "prontos a executar", pela primeira vez ou, mais tarde, em resultado de uma suspensão, é preciso decidir que páginas deslocar para a memória principal. Para isto podemos escolher uma de duas abordagens: uma estratégia minimalista e menos eficiente, em que nenhuma página é, em princípio, colocada e onde se joga com o mecanismo de geração de *page faults* para formar o *working set* do próprio processo (**demand paging**); ou uma estratégia mais eficiente em que se procura adivinhar o *working set* do processo para minimizar a geração de *page faults*, pelo que na primeira vez são colocadas as duas páginas atrás referidas e, nas vezes seguintes, o conjunto de páginas residente no momento em que ocorreu a suspensão - estratégia **prepaging**.

demand paging

prepaging

Um último aspeto a considerar nas políticas de substituição de páginas é o âmbito da aplicação dos algoritmos. Podemos ter um âmbito quer **local** (se a escolha for efetuada entre o conjunto de frames atribuído ao processo), quer **global** (se a escolha for efetuada entre todos os frames que constituem a lista dos frames ocupados.

local

global

O âmbito de aplicação global é geralmente preferível, dado que permite enquadrar grandes variações no *working set* dos processos, com parte do seu espaço de endereçamento residente em memória principal, sem que daqui resulte desperdício de memória ou *thrashing*[8].

5. Sistemas de Ficheiros

Como vimos em Arquitetura de Computadores II (a2s2), para que um sistema computacional realize trabalho útil, é necessário que exista um local para armazenamento

8 O *thrashing* não é completamente eliminado. Pode sempre ocorrer uma situação em que o somatório dos *working sets* dos processos é superior ao número de frames desbloqueados e disponíveis em memória principal. Neste caso a solução é ir sucessivamente suspendendo processos até que o problema desapareça.

mais ou menos permanente, das diferentes aplicações que são executadas. A este local demos o nome de **memória secundária** (ou de massa).

memória secundária

Assim, quando um sistema computacional é ligado, só existe, em memória principal, um programa, numa pequena região de tipo ROM, o **bootloader**, cuja função principal é ler de uma região específica da memória de massa um programa de maior dimensão que carrega em memória principal, e põe em execução o programa que implementa o ambiente de interação com o utilizador. Além disso, quase todos os programas, durante a sua execução, produzem, consultam e/ou alteram quantidades variáveis de informação que está armazenada de um modo mais ou menos permanente na memória de massa.

bootloader

Note-se que um conjunto de requisitos devem ser impostos à memória de massa. Primeiro, esta não pode ser volátil, isto é, a sua informação deve poder existir antes da criação do(s) processo(s) que a vai(ão) usar, e sobreviver à sua terminação, mesmo quando o sistema computacional é desligado. Depois, esta memória deverá ter grande capacidade de armazenamento, providenciar um acesso eficiente, manter a integridade da informação e a sua partilha.

A memória de massa, seja ela qual for, pode ser percebida em termos operacionais através de uma abstração muito simples, onde cada dispositivo é entendido como representando uma sequência de N **blocos** de armazenamento, cada um deles formado por b bytes[9]. Também se torna simples dado que o acesso a cada bloco, para leitura ou escrita, é feito de um modo aleatório e o tempo associado não depende do tipo de operação. Temos assim que um disco segue a represntação da Figura 5.1.

blocos

figura 5.1

A manipulação direta da informação contida no dispositivo físico não pode ser deixada inteiramente à responsabilidade do programador de aplicações. A complexidade inerente à sua estruturação e a necessidade de imposição de critérios de qualidade, relacionados com a eficiência no acesso, a integridade e a partilha, exigem a criação de um modelo uniforma de interação [13].

Conceito de ficheiro

Da mesma forma que vimos como é que o sistema operativo abstrai o conceito do processador criar abstrações de processos e como é que é que este abstraía (perdão pelo pleonasmo) o conceito de endereço físico de forma a oferecer espaços de endereçamento (virtuais) aos processos, também podemos resolver outro problema com uma nova abstração: o ficheiro. Juntas, as abstrações de processos (e *threads*), espaços de endereçamento, e ficheiros são os conceitos mais importantes relacionados com sistemas de operação.

Os computadores podem guardar informação em vários tipos de dispositivos, como discos magnéticos, cassetes ou discos óticos. Para que o sistema de um computador seja útil, o sistema de operação deverá fornecer uma visão mais lógica de toda a informação que retém. Assim, este abstrai tudo num conceito que define a unidade de armazenamento lógica: o **ficheiro**. Os ficheiros são mapeados pelo sistema operativo em dispositivos físicos. Estes equipamentos de armazenamento são usualmente, e como já sabemos, não-voláteis, pelo que os seus conteúdos são persistentes mesmo depois de falhas de energia ou reinícios da máquina.

ficheiro

Um ficheiro é assim uma coleção de informação relacionada, com um nome, que é guardado em memória secundária. Da perspetiva de quem usa, um ficheiro é o aglomerado mais pequeno de lógica armazenado na memória secundária, isto é, por outras palavras, quaisquer dados não podem ser escritos em memória de massa sem que estejam contidos num ficheiro.

9 O número de blocos, b, normalmente está entre 256 e 32K.

A informação presente num ficheiro é definida pelo seu criador. Muitos tipos diferentes de informação poderão ser assim guardados num ficheiro - códigos-fonte de programas, programas-objeto, programas executáveis, dados numéricos, ... Um ficheiro tem uma **estrutura**, a qual terá de depender do seu tipo. Este é, então, constituído por um **nome** (forma genérica de referenciar a informação) e um determinado **conteúdo**, isto é, a informação propriamente dita, organizada numa sequência de bits, bytes, linhas ou registos, cujo formato tem de ser conhecido por quem o usa.

estrutura, nome

conteúdo

Um ficheiro recebe então um nome, para facilitar a legibilidade por parte de humanos, que é uma sequência de carateres como `example.c`. Quando recebe um nome, o ficheiro torna-se independente do processo, do utilizador e até mesmo do sistema que o criou. Por exemplo, um utilizador poderia editar o ficheiro `example.c` especificando apenas o seu nome.

Os ficheiros têm assim um conjunto de **atributos** que variam de sistema de operação em sistema de operação, mas que tipicamente consistem nos seguintes:

atributos

- nome - um nome simbólico é a única informação acerca do ficheiro que é mantido numa forma legível aos humanos;
- identificador - esta etiqueta única, usualmente um número, identifica o ficheiro dentro de um sistema de ficheiros (esta é a única forma completamente ilegível num ficheiro);
- tipo - esta informação permite discriminar um tipo entre vários de cada sistema de ficheiros;
- local - ponteiro para um dispositivo e para o local do ficheiro nesse ficheiro;
- tamanho - o tamanho atual do ficheiro (em bytes, palavras ou blocos) e possivelmente o tamanho máximo também fica designado neste campo;
- proteção/permissões - controlo sobre quem é que pode ler, escrever ou executar um ficheiro;
- tempo, data e identificação do utilizador - esta informação poderá ser mantida para a verificação da data e outros critérios de criação, última modificação e último uso. Estes dados poderão ser úteis para questões de proteção, segurança e monitorização de uso dos componentes do computador.

Dependendo dos sistemas de ficheiros (sistemas que gerem os ficheiros no disco) os nomes poderão ser escritos com letras minúsculas e maiúsculas (o sistema FAT-16, do MS-DOS não é *case-sensitive*, pelo que para ele `hello.c` e `Hello.c` são o mesmo ficheiro) e, em alguns casos, terão a forma `nome.extensão`, onde a **extensão** procura indiciar, de algum modo, o tipo de organização interna apresentado pelos dados.

extensão

A informação sobre todos os ficheiros também se encontra na memória de massa. Tipicamente uma **entrada de diretório** consiste no nome do ficheiro (o diretório geralmente também é um ficheiro - tipo diferente) e no seu identificador único. O identificador, por outro lado, identifica os atributos do ficheiro.

entrada de diretório

Num sistema operativo da família UNIX (derivado de), como o macOS ou Linux, podemos ter uma ideia dos atributos de um ficheiro através da listagem de ficheiros num diretório ou, mais especificamente, com o comando `stat`, como mostramos no Código 5.1.

código 5.1

```
$ ls -l
-rw-r--r--  1 apontamentos  staff    93B Jan 18 00:30 hello.cpp
-rwxr-xr-x  1 apontamentos  staff    15K Jan 18 00:30 hello.out
-rw-r--r--  1 apontamentos  staff    23B Jan 18 00:27 hello.pl
$ stat hello.pl
16777220 17610706 -rw-r--r-- 1 apontamentos staff 0 23 "Jan 18 00:27:06 2017"
     "Jan 18 00:27:02 2017" "Jan 18 00:27:02 2017" "Jan 18 00:27:02 2017"
     4096 8 0 hello.pl
$ stat hello.out
16777220 17611018 -rwxr-xr-x 1 ruilopes staff 0 15204 "Jan 18 00:30:55 2017"
     "Jan 18 00:30:55 2017" "Jan 18 00:30:55 2017" "Jan 18 00:30:55 2017"
     4096 32 0 hello.out
```

Com estes comandos também podemos ter uma noção do **tipo de ficheiro**.

tipo de ficheiro

Na sequência de carateres `-rwxr-xr-x` temos, no fundo, dois atributos explícitos: o primeiro carater diz respeito ao tipo de ficheiro (sendo '-' significa que é ficheiro regular) e os outros nove carateres mostram os níveis de permissão de leitura '`r`', escrita '`w`' e de execução '`x`' para o utilizador atual, para o grupo atual e para outros. Mas atentemos no tipo de ficheiros e, para tal, vejamos o Código 5.2.

código 5.2

```
$ ls -l /dev
brw-r-----  1 root      operator       1,   0 Jan 17 17:03 disk0
brw-r-----  1 root      operator       1,   1 Jan 17 17:03 disk0s1
brw-r-----  1 root      operator       1,   2 Jan 17 17:03 disk0s2
brw-r-----  1 root      operator       1,   3 Jan 17 17:03 disk0s3
brw-r-----  1 root      operator       1,   4 Jan 17 17:03 disk1
brw-r-----  1 root      operator       1,   5 Jan 17 20:41 disk2
brw-r-----  1 root      operator       1,   6 Jan 17 20:41 disk2s1
brw-r-----  1 root      operator       1,   7 Jan 17 20:41 disk2s2
. . .
lr-xr-xr-x  1 root      wheel              0 Jan 17 17:03 stderr -> fd/2
lr-xr-xr-x  1 root      wheel              0 Jan 17 17:03 stdin -> fd/0
lr-xr-xr-x  1 root      wheel              0 Jan 17 17:03 stdout -> fd/1
. . .
crw-rw-rw-  1 root      wheel          4,  48 Jan 17 17:03 ttys0
crw--w----  1 apontamentos  tty           16,   0 Jan 18 00:40 ttys000
crw-rw-rw-  1 root      wheel          4,  49 Jan 17 17:03 ttys1
crw-rw-rw-  1 root      wheel          4,  50 Jan 17 17:03 ttys2
crw-rw-rw-  1 root      wheel          4,  51 Jan 17 17:03 ttys3
crw-rw-rw-  1 root      wheel          4,  52 Jan 17 17:03 ttys4
crw-rw-rw-  1 root      wheel          4,  53 Jan 17 17:03 ttys5
. . .
$ ls -l /dev/fd
crw--w----  1 apontamentos  tty      16,   0 Jan 18 00:47 0
crw--w----  1 apontamentos  tty      16,   0 Jan 18 00:47 1
crw--w----  1 apontamentos  tty      16,   0 Jan 18 00:47 2
$ ls -l
drwxr-xr-x  2 apontamentos  staff     68 Jan 18 00:44 code-samples
-rw-r--r--  1 apontamentos  staff     93 Jan 18 00:30 hello.cpp
-rwxr-xr-x  1 apontamentos  staff  15204 Jan 18 00:30 hello.out
-rw-r--r--  1 apontamentos  staff     23 Jan 18 00:27 hello.pl
```

No Código 5.2 podemos ver vários ficheiros com os diferentes tipos para a família UNIX. Um ficheiro normal, isto é, um **ficheiro regular** é um ficheiro convencional que contém todo o tipo de informação que é habitualmente armazenada em memória de massa, e estão representados alguns exemplos a castanho, no código acima. A rosa, estão designados um segundo tipo de ficheiro - os **diretórios**. Estes ficheiros são internos, com um formato pré-definido, que descrevem a estrutura subjacente. Se olharmos para as linhas a azul podemos ver que a designação do tipo é '`l`', sendo estes *links*, isto é, **atalhos**, que são ficheiros internos, com formato pré-definido, que descrevem a localização de um outro ficheiro através da especificação do seu encaminhamento na estrutura hierárquica pré-existente, o que podemos ver em particular com as três linhas do Código, que apontam para ficheiros do diretório `fd`.

ficheiro regular

diretórios

atalhos

Os ficheiros para que os atalhos apontam são muito especiais. Mais em particular, o sistema que produziu o Código 5.2 (sistema operativo macOS) tem uma organização tal que assume que os dispositivos de *standard in*, *out* e *error* são atalhos para ficheiros em `/dev/fd`. Possivelmente agora ficamos um pouco com dúvidas, mas de facto os dispositivos aqui também são ficheiros, em particular para o *standard in*, *out* e *error*, **ficheiros de tipo carater**, daí a letra '`c`'. São do tipo carater porque relacionam-se com portas série, como os terminais (`tty`), impressoras, teclados, redes, ... Estes ficheiros estão identificados pela cor vermelha. Por fim temos os ficheiros identificados a verde, que são os **ficheiros de tipo bloco**. Tais ficheiros fazem respeito a discos de memória em massa, cuja abstração é que o seu conteúdo está em blocos, como já verificámos.

ficheiros de tipo carater

ficheiros de tipo bloco

Analisemos então um diretório. Os dispositivos físicos atuais, face à grande capacidade de armazenamento interno, podem conter em condições normais, milhões de ficheiros regulares. Se todos eles estivessem colocados ao mesmo nível, a referência a um deles em particular, ou a atribuição de um nome a um novo ficheiro, tornar-se-iam operações quase impossíveis de realizar na prática pela complexidade que lhes está inerente. Assim, qualquer sistema de ficheiros contemporâneo pressupõe uma organização hierárquica da informação, onde um **diretório raiz** denota a origem do sistema de ficheiros.

diretório raiz

Na Figura 5.2 podemos então ver dois tipos de sistemas de diretórios.

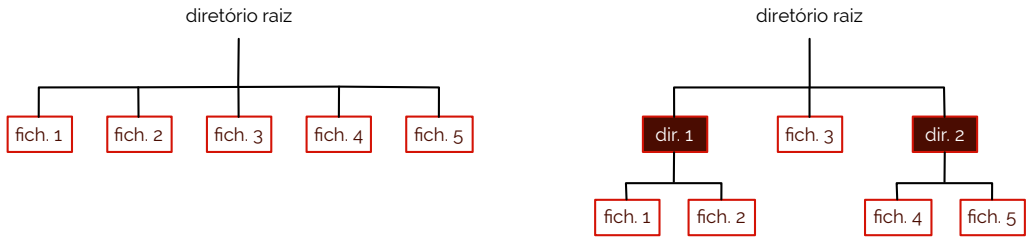


figura 5.2

sistema de diretório único **sistema hierárquico de diretórios**

Quando o sistema de ficheiros está organizado numa árvore de diretórios, terá que se arranjar uma forma qualquer de especificar os nomes dos ficheiros. Assim, dois métodos muito comuns são usados. Um primeiro método consiste em usar um nome que é atribuído a cada ficheiro e que consiste no caminho completo da raiz até ao ficheiro. Este método denomina-se de **encaminhamento absoluto**. Como exemplo, o caminho `/apontamentos/so/cap5` indica que o diretório raiz possui um diretório de nome `apontamentos`, que contém um diretório `so` e que, por sua vez, contém um ficheiro (não sabemos o tipo dele) chamado `fs`. Estes nomes são sempre únicos. Há apenas de ter cuidado, de sistema operativo em sistema operativo, porque no UNIX os componentes de um caminho são separados pelo carater '`/`', mas noutros como no Windows™ são separados por '`\`'. No entanto, o caminho qualquer seria escrito, no caso do Windows, `\apontamentos\so\fs`.

encaminhamento absoluto

O outro método possível é o **encaminhamento relativo**. Este método é usado em junção com o conceito de **working directory** (variável de sistema que aloja o diretório atual num terminal[10]). O que acontece neste caso é que o utilizador não precisa de escrever o caminho todo até um ficheiro, podendo escrever o caminho que toma para lhe chegar desde a posição atual. Por exemplo, se um utilizador estiver em `/apontamentos/so/fs` e nele existir um executável `hello.out`, o utilizador poderá executá-lo escrevendo o caminho completo (`./apontamentos/so/fs/hello.out`) ou simplesmesnte escrever o executável à frente do comando (`./hello.out`).

encaminhamento relativo
working directory

Em qualquer diretório, se executarmos `ls -la` podemos ver dois diretórios com nomes especiais: '`.`' e '`..`'. Estes diretórios são representações do diretório atual e do diretório-pai (o anterior), respetivamente. No caso do diretório raiz, como não existe nenhum diretório acima deste, ambos '`.`' e '`..`' apontam para si próprio.

Em UNIX, existe uma única hierarquia de diretórios que integra todos os sistemas de ficheiros residentes nos vários dispositivos de memória de massa presentes no, ou acessíveis pelo, sistema computacional. Estes vários sistemas de ficheiros carecem de estarem **montados** para que possam ser usados e para que possam fornecer diretórios particulares. Numa pequena variante, em Windows, cada dispositivo de memória de massa presente no, ou acessível pelo, sistema computacional tem a sua própria hierarquia e é identificado por uma letra do alfabeto latino seguido de '`:`' (`A:`, `B:`, ...).

montados

Em relação aos **atalhos**, estes também poderão usar encaminhamento absoluto ou relativo. A sua razão de existência prende-se com o facto de poder haver uma maior transparência de acesso a sucessivas versões de uma mesma aplicação, redirecionar o acesso a zonas de dados protegidas e permitir o acesso a diretórios e a ficheiros regulares residentes em sistemas de ficheiros distintos. Note-se que nada nos pode impedir de criarmos um atalho que aponte para outro atalho. Apenas nos temos de preocupar caso haja um ciclo formado ao longo dos atalhos.

atalhos

Existe um conjunto de **operações sobre ficheiros regulares** que podem ser feitas (dependem do sistema de ficheiros). Consideremos as seguintes operações, que formam o núcleo-base que de algum modo está sempre presente em todos os sistemas de operação:

operações sobre ficheiros regulares

10 O seu valor poderá ser consultado com o comando pwd.

- criar um ficheiro - são necessários dois passos para criar um ficheiro. Primeiro há que encontrar um espaço vazio para este no disco. Depois deverá ser criada uma entrada de diretório, e preenchida com nome e identificação;
- abrir um ficheiro - alternativamente à criação, sendo efetuada sempre que o ficheiro referenciado já existe, aqui as entradas de diretório são pesquisadas para localizar a entrada respetiva e para aceder aos atributos.
- escrever um ficheiro - para escrever um ficheiro devemos efetuar uma chamada de sistema a especificar tanto o nome do ficheiro, como a informação que se pretende escrever no ficheiro. Dado o nome do ficheiro, o sistema deverá procurar no diretório para encontrar o seu paradeiro. O sistema deverá então colocar e manter um ponteiro para a escrita, no local onde a próxima escrita deve ser tomada. O ponteiro para a escrita deve ser atualizado sempre que ocorre uma escrita.
- ler um ficheiro - para ler de um ficheiro, usamos uma chamada de sistema que especifica o nome do ficheiro e onde (em memória) é que o próximo bloco do ficheiro deverá ser colocado. Novamente, o diretório é revistado para se encontrar uma entrada associada, e o sistema precisa de manter um ponteiro de leitura no ficheiro onde a próxima leitura será tomada. Uma vez que a leitura fica feita, o ponteiro para leitura é atualizado. Na Figura 5.3 podemos ver uma representação da abstração que se faz do conteúdo de um ficheiro visto como um *continuum* de dados;
- posicionar um ficheiro - o conteúdo do ficheiro regular é percebido pelo programador de aplicações como um *continuum* de dados, sendo que o local nesse *continuum* onde irá decorrer a próxima transferência de informação pode ser definido através da operação de posicionamento que fixa a chamada **posição atual** (note-se que quando um ficheiro é criado ou aberto a posição atual aponta diretamente para a posição inicial desse ficheiro);
- truncar um ficheiro - o utilizador poderá querer apagar os conteúdos de um ficheiro mas manter os seus atributos. Ao invés de forçar o utilizador a apagar um ficheiro e depois recriá-lo, esta função permite-lhe que todos os atributos sejam mantidos (à exceção do tamanho do ficheiro);
- fechar um ficheiro - esta operação traduz o encerramento da comunicação com o ficheiro regular, sendo que o bloco de dados relativo à parte do conteúdo que estava a ser referenciado na altura é escrito no dispositivo, se tiver ocorrido alteração;
- eliminação de um ficheiro - para eliminar um ficheiro, o diretório é pesquisado pelo nome do ficheiro. Tendo-o encontrado (sob a forma de uma entrada de diretório), libertamos todo o espaço por ele ocupado e apagamos a entrada de diretório.

posição atual

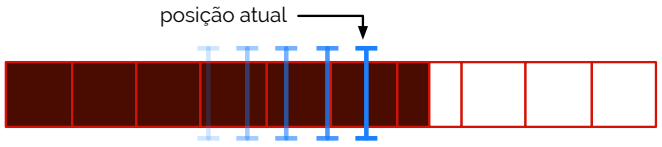


figura 5.3

Também existem **operações sobre diretórios** que, dependendo de sistema em sistema de ficheiros, geralmente estas operações estão presentes:

operações sobre diretórios

- criação - é reservada no diretório uma entrada para o novo diretório. A entrada vai conter o nome e os atributos, ou uma referência para os atributos, que são inicializados. Ao contrário do que se passa com um ficheiro regular, o tamanho do diretório não é colocado a zero, pelo que um diretório vazio contém sempre referências para si próprio '.' e para o diretório imediatamente acima '..', para que a navegação na árvore seja mais eficiente;

- eliminação - o diretório é removido do sistema de ficheiros. Tal como um ficheiro regular, os blocos de dados de armazenamento do seu conteúdo são libertados e a entrada do diretório correspondente (bem como a região onde estão armazenados os restantes atributos, se não estiverem localizados na entrada de diretório) são sinalizadas vazias. O diretório tem de estar vazio para que a operação possa ter lugar;
- listagem do conteúdo - as entradas de diretório ocupadas são lidas para se obter uma descrição dos atributos dos ficheiros que são visíveis a esse nível da organização hierárquica interna.

Finalmente, e do mesmo modo, existem **operações sobre atalhos**:

operações sobre atalhos

- criação - é reservada no diretório uma entrada para o atalho. A entrada vai conter o nome e os atributos, ou uma referência para os atributos, que são inicializados. Ao contrário do que se passa com um ficheiro regular, o tamanho de um atalho é colocado num valor que corresponde ao tamanho da *string* que define o encaminhamento e o conteúdo informativo contém esse encaminhamento;
- eliminação - o atalho é removido do sistema de ficheiros. Tal como para um ficheiro regular, os blocos de dados de armazenamento do seu conteúdo são libertados e a entrada de diretório correspondente (bem como a região onde estão armazenados os restantes atributos, se não estiverem localizados na entrada de diretório), são sinalizadas vazias;
- listagem do conteúdo - o conteúdo do atalho é lido para ser posteriormente usado como um caminho na localização de um outro ficheiro.

De forma geral, também podemos consultar os atributos, alterá-los e modificar o nome (de qualquer tipo de ficheiro).

Implementação de um sistema de ficheiros

O principal problema que se coloca ao programador de sistemas na implementação de um sistema de ficheiros é como é que se pode organizar o espaço de armazenamento interno do dispositivo, entendido como um *array* de blocos, de modo a fornecer a visão abstrata esperada pelo programador de aplicações, aquilo que se designa de **arquitetura interna** [13].

arquitetura interna

Os ficheiros de sistema serão guardados em discos. Grande parte dos discos podem ser divididos em uma ou mais partições, com sistemas de ficheiros independentes entre si. O setor 0 do disco é denominado de **MBR** (*Master Boot Record*) e é usado para ligar o computador (operação de *boot*), sendo que o seu fim contém a tabela de partições. Esta tabela indica qual é que é o início e fim de cada partição em disco, através de endereços. Quando um computador é então ligado, a BIOS irá ler o setor e executar o MBR, sendo que a primeira coisa que esta faz é ativar a partição indicada e nele ler um primeiro bloco, chamado de **boot block**, e executá-lo. [2]

MBR

boot block

Mais do que iniciar uma execução através do *boot block*, o esquema da partição de um disco varia muito de sistema em sistema de ficheiros. Muitas vezes este também contém um bloco (geralmente bloco número 0) cujo nome é **superbloco** e que contém toda a informação acerca da partição do disco.

superbloco

Consideremos que o nosso disco teria um **armazenamento contínuo**, isto é, que a alocação de espaço era feita de forma contígua. Assim, num disco com blocos de 1kB, um ficheiro de 50kB ocuparia 50 blocos consecutivos e com blocos de 2kB, alocar-se-iam 25 blocos consecutivos.

armazenamento contínuo

Na Figura 5.4 podemos ver um exemplo de alocação contígua. Aqui os primeiros 40 blocos são mostrados, a começar no bloco 0 à esquerda. Inicialmente, o disco estava vazio. Entretanto adicionámos um ficheiro A (com tamanho de 4 blocos) e um ficheiro B (3 blocos), que foi escrito mesmo à frente do ficheiro A. Note-se que cada ficheiro é escrito

no início de um bloco, sendo que se algum ficheiro ocupe 6.5 blocos, então algum espaço será desperdiçado no fim do último bloco.

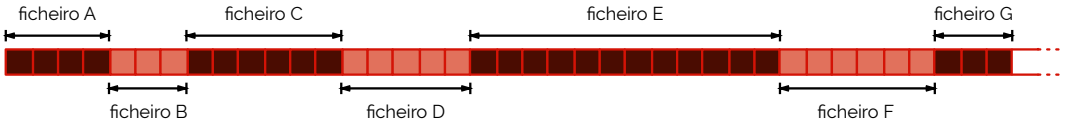


figura 5.4

A alocação de espaço de disco de forma contígua tem duas vantagens: é simples de implementar porque, no fundo, só precisamos de estar conscientes de dois números (endereço do início do espaço livre e endereço final da partição), e a leitura tem um desempenho perfeito porque um ficheiro inteiro pode ser lido ininterruptamente, dada a sua continuidade no disco.

Infelizmente esta estratégia tem um problema sério. Embora a fragmentação interna, por si, já fosse visível (dado que só parte do bloco do conteúdo informativo de cada ficheiro é que é eventualmente desperdiçado), também temos uma grande fragmentação externa, pelo que se começarmos a eliminar ficheiros, como podemos ver na Figura 5.5, onde os ficheiros D e F foram eliminados, deixando espaço para ficheiros apenas de menos de 7 blocos. Assim, esta implementação é restrita a sistemas de *backup*, residentes em bandas magnéticas ou discos de leitura ótica, ou a informação só-de-leitura, residente em discos de leitura-ótica.

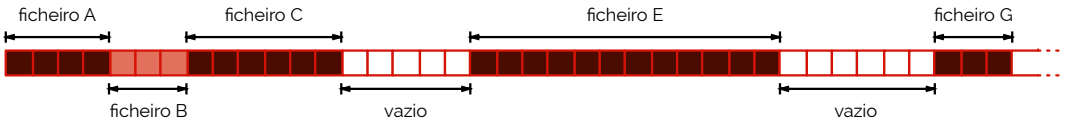


figura 5.5

Entramos assim em abordagens de **armazenamento disperso**, isto é, onde quer o conteúdo informativo, quer o espaço livre, são descritos pelos agregados de blocos que lhes estão associados. Esta estratégia pode ser implementada usando vários métodos. Analisemos alguns deles.

armazenamento disperso

Um primeiro método para armazenar ficheiros de forma dispersa é criando uma **lista ligada** de blocos de disco, isto é, em que os nós são os próprios blocos. Assim, a primeira palavra de cada bloco será usada para indicar onde é que se encontra o bloco seguinte (ponteiro), sendo que o resto dos blocos são dados. Uma representação desta implementação está feita na Figura 5.6.

lista ligada

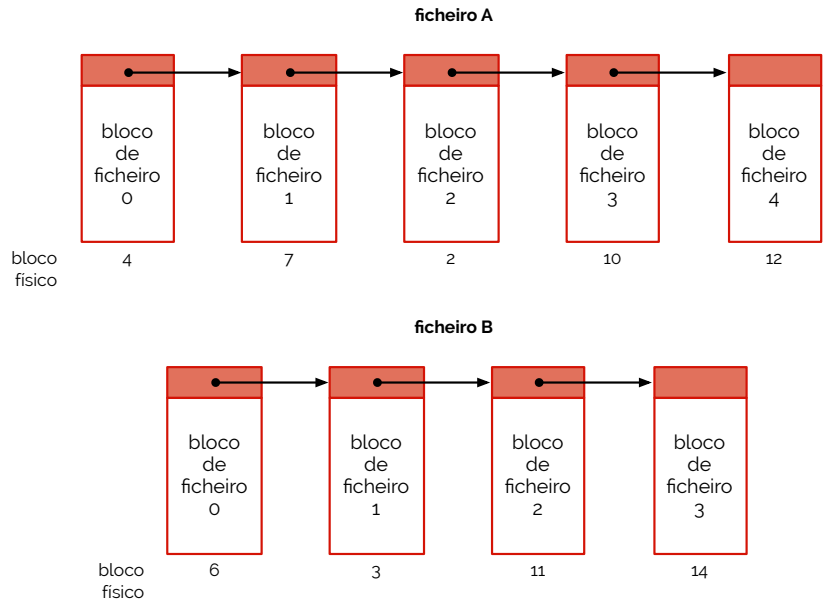


figura 5.6

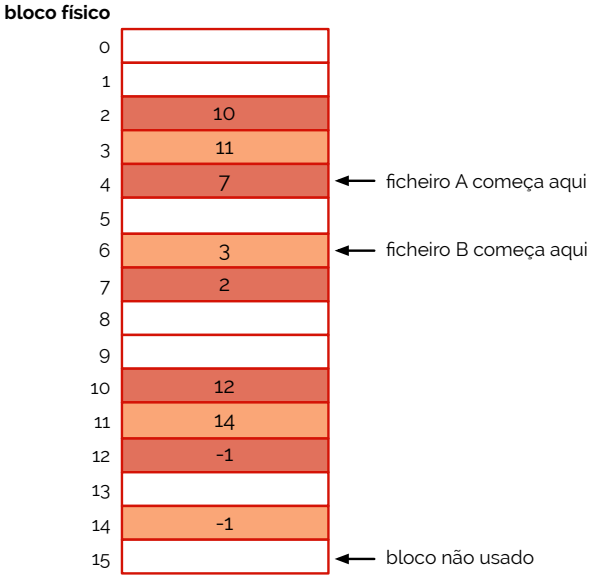
Contrariamente à alocação contígua, aqui cada bloco pode ser usado, sendo que não é perdido qualquer espaço em fragmentações (exceto a fragmentação interna, dentro dos blocos). Mais, também será suficiente que as entradas de diretório apenas guardem referência para o primeiro bloco, dado que o resto poderá ser concluído através deste.

Por outro lado, embora a ler um ficheiro nos pareça bastante razoável, o acesso aleatório é extremamente lento, sendo que para obtermos um bloco b, o sistema operativo terá que iniciar no início e ler b-1 blocos anteriores a este, um de cada vez. Mais, a quantidade de dados que um bloco preserva deixa de ser uma potência de base 2, dado que o ponteiro acrescenta alguns bytes. Mesmo não sendo uma causa fatal, o facto de termos de contar com um tamanho peculiar exige que haja tratamentos especiais em termos de manipulações de blocos, o que muitos programas podem não estar preparados.

Com as conclusões da aplicação de listas ligadas, uma forma de melhorar o acesso aleatório é mapeá-lo numa tabela que nos indique, de forma mais rápida, onde é que está o ficheiro e quais são os blocos que lhe estão associados. Na Figura 5.7 podemos ver um exemplo de tal tabela, onde resumimos as listas da Figura 5.6. Em ambas as figuras o ficheiro *A* usa os blocos 4, 7, 2, 10 e 12 (nesta ordem) e o *B* usa os blocos 6, 3, 11 e 14 do disco. Usando a tabela, podemos começar pelos blocos 4 e seguir a ligação em cadeia até ao fim. A mesma coisa poderá ser feita com o bloco 6. As ligações em cadeia terminam com um carater especial, que não é um índice válido, como o -1 (em alguns sistemas equivalente ao ponteiro nulo - `nil`). A tal tabela damos o nome de **FAT** (do inglês *File Allocation Table*).

FAT

figura 5.7



Através desta organização, o bloco inteiro fica disponível para dados. Mais, o acesso aleatório torna-se muito mais simples, como já tinhamos concluído. Embora a cadeia deva ser seguida para procurar um determinado desvio num ficheiro, esta está enteiramente na memória, pelo que poderá ser seguida sem que haja novas referências ao disco.

O único grande problema deste esquema, é que a tabela deverá estar toda em memória de forma a poder funcionar. Ou seja, com um disco de 1TB com blocos de 2kB, a tabela necessitaria de 1 milhar de milhão de entradas, uma para cada um do 1 milhar de milhão de blocos em disco. Cada entrada tem de ter, no mínimo, 3 bytes, para poder endereçar todos os blocos, ou 4 bytes, para otimizar a procura. Assim a nossa tabela, exclusivamente, terá um tamanho de 3GB ou 2.4GB na memória principal, sempre, o que não é muito prático. Claramente o sistema de ficheiros FAT não se adequa para discos grandes. Este foi o sistema de ficheiros inicial do Windows™ e ainda é suportado por todas as suas versões.

O nosso último método para acompanhar que blocos é que pertencem a que ficheiro é associando com cada ficheiro uma estrutura de dados chamada de **nó-i** (nó-

nó-i

índice), que lista os atributos e endereços dos blocos do disco. Um exemplo muito simples está visível na Figura 5.8. Dado um nó-i é então possível encontrar todos os blocos de um ficheiro. A grande vantagem deste esquema perante o anterior é que o nó-i só precisa de estar em memória quando o ficheiro correspondente está aberto. Se cada nó-i ocupar n bytes e um máximo de k ficheiros podem ser abertos em simultâneo, o total de memória ocupada pelo array com os nós-i dos ficheiros abertos é somente kn bytes.

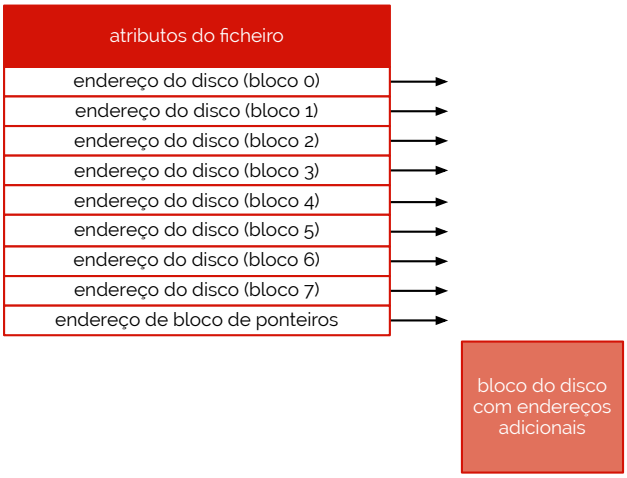
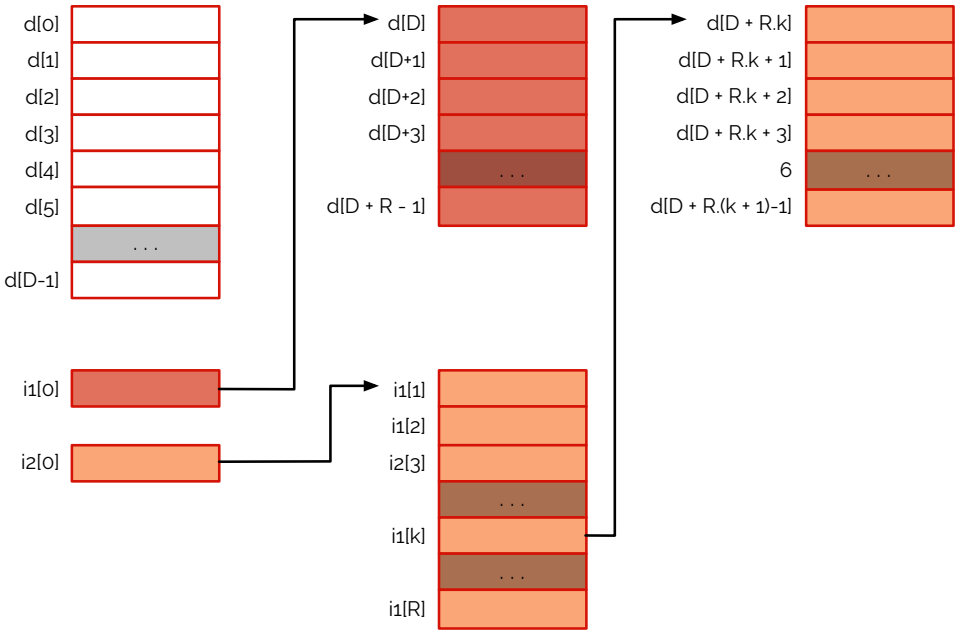


figura 5.8

Este array é usualmente mais pequeno que o espaço ocupado pela tabela descrita anteriormente. Contudo, um problema com as tabelas de nós-i é que cada uma tem apenas espaço para um número fixo de endereços de disco, sendo que se houver necessidade de mais endereços, então usa-se um bloco do disco com endereços adicionais, numa **referência simplesmente indireta**. Se ainda não for suficiente, ainda se pode extender a tabela com mais uma lista de blocos de disco com endereços adicionais, numa **referência duplamente indireta**. Na Figura 5.9 podemos ver uma representação melhorada da tabela de nós-i com este ponto em consideração, onde o array `d[]` é array de referências diretas, `i1[]` é o array de referências simplesmente indiretas e `i2[]` o array de referências duplamente indiretas.

referência simplesmente indireta

referência duplamente indireta



** R é o número de referências por bloco

figura 5.9

No exemplo ilustrado na Figura 5.9 os ficheiros com tamanho até $D.b$ bytes (sendo b o tamanho de um bloco) não necessitam de qualquer leitura intercalar de blocos para

efetuarem um acesso aleatório a qualquer ponto do seu conteúdo informativo. Ficheiros maiores, mas com tamanho até $(D + R).b$ bytes necessitam no máximo de uma leitura intercalar (considere-se que R é o número de referências por bloco), e ficheiros maiores, mas com tamanho até $(D + R + R^2).b$ bytes necessitam no máximo de duas leituras intercalares.

Antes de um ficheiro ser lido, este deve ser aberto primeiro. Quando um ficheiro é aberto, o sistema de operação usa o nome (do caminho) dado pelo utilizador para encontrar a entrada de diretório no disco. Uma **entrada de diretório** fornece, assim, a informação necessária para encontrar os blocos respetivos de um ficheiro no disco. Dependentemente do sistema, esta informação poderá ser o endereço do ficheiro inteiro (com alocação contígua), o número do primeiro bloco (com alocação em listas ligadas) ou simplesmente o número do nó-i. Em todo o caso, a maior função das entradas de diretório é mapear essa identificação com um nome codificado em ASCII (por exemplo), na informação necessária para se localizar um ficheiro.

entrada de diretório

Um assunto muito relacionado com este é onde é que os atributos devem ser guardados, tal como a pertença do ficheiro ou o tempo de criação. Uma forte possibilidadade é guardar esta informação nas entradas de diretório, mas em sistemas que usam nós-i, uma outra, ainda mais forte, possibilidade é guardá-los nos próprios nós-i, pelo que as nossas entradas de diretório passariam a ser apenas um conjunto de linhas com um nome ASCII e uma identificação do nó-i (um número).

Quando um diretório é criado, o seu conteúdo é formado pelo conjunto de entradas que podem ser armazenadas numbloco da zona de dados. Destas, só as duas primeiras são ocupadas: a primeira com uma referência ao próprio diretório ('.') e a segunda com uma referência ao diretório imediatamente acima na hierarquia ('..'). À medida que outros ficheiros vão sendo criados e incluídos no diretório, as restantes entradas vão sendo ocupadas e, quando não houver já entradas livre, um novo bloco será associado ao conteúdo informativo para aumentar o tamanho do array.

O campo do nome, da entrada de diretório, coloca um problema mais complicado. Impor um número pequeno de carateres para o armazenamento do nome não é, hoje em dia, uma opção muito viável. A tendência atual é permitir a utilização de identificadores de comprimento mais ou menos longo, formados eventualmente por uma sucessão de palavras. Uma solução possível é, ainda assim, limitar o tamanho do campo a um número de carateres assumindo, como razoável, 64 ou 256. Esta alternativa permite que os algoritmos de pesquisa não sejam muito complexos.

Por outro lado, se não se quiser impor um limite efetivo ao tamanho dos identificadores que nomeiam os ficheiros, pode sempre seguir-se uma das alternativas seguintes: fixar o tamanho do campo nome a um valor pré-definido e usar, sempre que se justifique, entradas de diretório sucessivas para armazenar o identificador do ficheiro; organizar o conteúdo informativo do diretório num array de entradas de diretório e uma região para armazenamento dos identificadores dos ficheiros de uma forma completa, de forma aque a contribuição do campo nome para o tamanho da entrada possa ser considerada mínima.

Com o uso de atalhos, a hierarquia de diretórios deixa de constituir uma árvore e passa a ser um grafo acíclico dirigido (DAG). Com isto, a nomeação de um mesmo ficheiro através de atalhos pode ser diferente e, portanto, o nome do ficheiro passa a ser uma propriedade meramente local e não uma propriedade central dos objetos. Assim, os atributos do ficheiro têm de conter um elemento adicional que conta o número de referências existentes do sistema de ficheiros, associadas com esse ficheiro. A elminação de um ficheiro num diretório passa a representar a eliminaçao da entrada de diretório correspondente e não necessariamente a remoção do ficheiro do sistema de ficheiros. Isto só ocorrerá quando o contador de referências estiver a zero.

Por esta mesma razão, cada entrada de diretório constitui aquilo a que se designa de **hard link** ao ficheiro, por oposição a um atalho (**symbolic link**) que constitui uma ligação simbólica ao ficheiro.

hard linn, symbolic link

Disco magnético

Como memórias externas podemos ter discos magnéticos ou *solid-state disks*. Começando pelos **discos magnéticos** (HDD) estes têm princípios puramente mecânicos. Durante muitos anos, a tecnologia do **magnetismo** dominou a área do armazenamento em massa. O exemplo mais comum, que ainda hoje se usa, são os discos magnéticos, nos quais existe um disco de superfície magnética que gira e guarda informação. Cabeças de escrita e leitura estão posicionadas por cima e/ou por baixo do disco, para quando este girar, cada cabeça faça um círculo, este, chamado de **faixa** (*track*). Ao reposicionar as cabeças de escrita/leitura diferentes faixas concêntricas são passíveis de serem acedidas. Em muitos casos, sistemas de armazenamento em disco consistem em vários discos montados num mesmo eixo, uns por cima dos outros, com determinados espaços entre eles, de modo a que caibam as cabeças de escrita/leitura, entre as superfícies. Em certos casos as cabeças movem-se em uníssono. Cada vez que estas são reposicionadas, um novo conjunto de faixas - chamado de **cilindro** - torna-se acessível. Todos estes detalhes podem ser vistos na Figura 5.10.

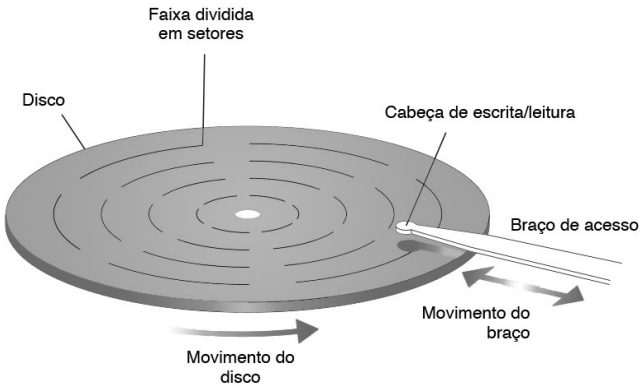
discos magnéticos

magnetismo

faixa

cilindro

figura 5.10



Sendo que uma faixa pode conter muito mais informação do que a que nós pretendemos manipular em simultâneo, cada faixa é dividida em pequenos arcos chamados **setores** onde a informação é gravada numa contínua sequência de bits. Todos os setores num disco contêm o mesmo número de bits (as capacidades típicas são entre 512 bytes a uns poucos kilobytes), e no mais simples disco, todas as faixas apresentam o mesmo número de setores. Por conseguinte, nesse disco, as informações gravadas na faixa mais exterior estão menos compactadas do que na mais interior, sendo que as primeiras são maiores que as últimas. Em discos de alta capacidade de armazenamento, as faixas mais afastadas do centro são as com maior probabilidade de ter mais setores que as interiores, capacidade adquirida pelo uso da técnica de **gravação de bit localizada** (*zoned-bit recording*). Usando esta técnica, várias faixas adjacentes são coletivamente conhecidas como zonas, sendo que cada disco contem cerca de dez zonas. Aqui, todas as faixas de uma dada zona têm o mesmo número de setores e cada zona tem mais setores por zona que as suas inferiores. Desta forma consegue-se maior rendimento no que toca ao aproveitamento de todo o espaço do disco. Independentemente dos detalhes, um sistema de armazenamento em disco consiste em vários setores individuais, cujos podem ser acedidos como sequências independentes de bits.

setores

gravação de bit localizada

Diversas medidas são usadas para avaliar o desempenho de um disco:

- **tempo de procura** (*seek time*): o tempo necessário para mover uma cabeça de escrita/leitura de uma faixa para outra;
- **atraso de rotação** ou **tempo latente** (*rotation delay*): metade do tempo necessário para que o disco faça uma rotação completa, a qual é a média do tempo total para que uns dados desejados girem até às cabeças de escrita/leitura, desde o momento em que estas já se localizam por cima da faixa pretendida;

tempo de procura

atraso de rotação, tempo latente

- **tempo de acesso** (*access time*): a soma do tempo de procura e do atraso de rotação; **tempo de acesso**
- **taxa de transferência** (*transfer rate*): a taxa de velocidade a qual a informação pode ser transportada para ou do disco. **taxa de transferência**

Note-se que no caso da gravação de bit localizada, o total de dados que passam as cabeças de escrita/leitura numa única rotação do disco é maior nas faixas exteriores que nas faixas interiores, por conseguinte a taxa de transferência também diferirá consoante a faixa em questão.

Um fator que limita o tempo de acesso e a taxa de transferência é a velocidade a que o disco roda. De modo a facilitar essa velocidade as cabeças de escrita/leitura não tocam nunca no disco, mas antes "flutuam" sobre a sua superfície. O espaço entre a cabeça e o disco é tão pequeno que um simples grão de pó pode travar o sistema, destruindo ambos - um fenómeno chamado de **head crash**. Tipicamente, estes discos vêm, de fábrica, selados, dadas essas possibilidades. Só assim é que os discos podem girar a velocidades de várias centenas por segundo, alcançando taxas de transferência que são medidas em megabytes por segundo. Há porém, sempre um *overhead* de cerca de uma dezena de milissegundos no acesso ao disco magnético no início de cada nova transferência. **head crash**

Sendo que os sistemas de armazenamento requerem movimento para as suas operações, estes ficam a perder quando comparados a circuitos eletrónicos. Tempos de atraso, num circuito eletrónico, são medidos em nanossegundos ou menos, enquanto que tempos de procura, de latência e de acesso são medidos em milissegundos. Assim, o tempo necessário para receber algum dado de um sistema de armazenamento em disco pode parecer um "eternidade", em comparação a um circuito eletrónico que aguarda uma resposta. [14]

E assim terminam os apontamentos para a disciplina de Sistemas de Operação (a3s1), continuando, estes conteúdos, na disciplina de Arquitetura de Computadores Avançada (a4s1) e Sistemas Distribuídos (a4s2).

1. Introdução aos Sistemas de Operação

Introdução ao sistema UNIX 2
Interação com o utilizador (comandos na shell) 4
Tipos de sistemas de operação 8
Multiprogramação e multiprocessamento 9

2. Gestão do Processador

Execuções num processador 9
Espaço de endereçamento e tabela de controlo de um processo 14
Comutação de processos 16
Políticas de escalonamento (scheduling) 16
Processos e threads 19

3. Comunicação entre Processos

Princípios gerais 22
Tipo de soluções 24
Construção de uma solução por software 24
Construção de soluções por hardware (semáforos e monitores) 27
Problema do jantar dos filósofos 35
Sinais em UNIX 36
Pipes em UNIX 37
Prevenções de deadlock 38

4. Gestão da Memória

Hierarquia da Memória 40
Espaço de endereçamento 41
Organização de memória real 42
Organização de memória virtual 45
Políticas de substituição de páginas em memória 53

5. Sistemas de Ficheiros

Conceito de ficheiro 56
Implementação de um sistema de ficheiros 61
Disco magnético 66

As referências abaixo correspondem às várias citações (quer diretas, indiretas ou de citação) presentes ao longo destes apontamentos. Tais referências encontram-se dispostas segundo a norma IEEE (as páginas Web estão dispostas de forma análoga à de referências para livros segundo a mesma norma).

[1] A. R. Borges, "Slides da disciplina de Sistemas Operativos - Apresentação", Universidade de Aveiro, 2015.

[2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4 ed. New Jersey: Pearson, 2015.

[3] M. J. Bach, *The Design of the UNIX Operating System*, 1 ed. New Jersey: Prentice-Hall, 1986.

[4] A. Pereira. (2016, Utilização da bash em ambiente GNU/Linux. *Guiões das aulas práticas*, volume 1).

[5] A. R. Borges and A. Pereira, "Slides da disciplina de Sistemas de Operação - Gestão do Processador", Universidade de Aveiro, 2016.

[6] W. Stallings, Operating Systems: Internals and Design Principles, 8 ed. New Jersey: Pearson, 2015.

[7] A. R. Borges and A. Pereira, "Slides da disciplina de Sistemas de Operação - Comunicação entre Processos", Universidade de Aveiro, 2016.

[8] E. W. Dijkstra, "Co-operating Sequential Processes," unpublished, 1965.

[9] P. J. Denning, "The Locality Principle," *Communications of the ACM,* vol. 48, pp. 19-24, July 2005.

[10] A. R. Borges and A. Pereira, "Slides da disciplina de Sistemas de Operação - Gestão da Memória", Universidade de Aveiro, 2016.

[11] Jessen, Elke, "Origin of the Virtual Memory Concept," IEEE Annals of the History of Computing, vol. 26, pp. 71-72, 2004

[12] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts Essentials*: John Wiley & Sons, 2011.

[13] A. R. Borges and A. Pereira, "Slides da disciplina de Sistemas de Operação - Sistemas de Ficheiros (Princípios)", Universidade de Aveiro, 2016.

[14] R. Lopes, *Apontamentos de Arquitetura de Computadores II*: Apontamentos, 2015.

Apontamentos de Sistemas de Operação
1ª edição - janeiro de 2017

SO

Autor: Rui Lopes
Agradecimentos: professor Artur Pereira e professor João Manuel Rodrigues

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.

apontamentos