Sebenta de SO

Filesystems, Sofs17, Programação concorrente, Process switching e Processor Scheduling

PEDRO MARTINS

January 9, 2018

Contents

1 sofs2017

> *The sofs17 is a simple and limited file system, based on the ext2 file system, which was designed for purely educational purposes and is intended to be developed in the practical classes of the Operating Systems course in academic year of 2017/2018. The physical support is a regular file from any other file system.*

- Sistema simples e limitado
- Baseado no ext2
- Suporte físico: um ficheiro regular de outro sistema operativo
  - Este ficheiro será formatado para imitar uma unidade física formatada no formato sofs17

2 Organização das aulas durante o sofs17

2 horas:
- 1h30 : interagir relativamente ao trabalho pendente
- 0h30 : falar da próxima camada de software

3 Introduction

- Durante a execução de um programa, ele manipula informação (produz, acede e/ou modifica).
- Esta informação tem de ser guardada exteriormente (**mass storage**)
  - discos magnéticos
  - discos ópticos
  - SSD
  - …
- **mass storage** (armazenamento de massa): dispositivos organizados em arrays de blocos
  - 256 bytes até 8 Kbytes por bloco
  - os blocos são numerados sequencialmente (LBA model)
  - o acesso para R/W é efetuado através de um ID (identification number)

Block 0	Block 1	Block 2	Block 3	…	Block NTBK-1

Cada bloco tem BKZS bytes de informação - O acesso ao disco é feito bloco a bloco: - **Não é possível modificar um único byte**

> *Direct access to the contents of the device* ***should not be allowed to the application programmer.***

Porque:

- Um sistema de ficheiros é complexo
- A sua estrutura interna precisa de *enforce quality criteria* para garantir:
  - eficiência
  - integridade
  - partilha de acessos
- O utilizador não sabe o conteúdo de cada bloco de dados nem em que blocos a informação do ficheiro x está.

Daí a necessidade/exigência da existência de um *uniform interaction model* (Nível de abstração).

ficheiro:

- unidade lógica de armazenamento de massa
- *abstract data type*, sobre o ponto de vista do programador
  - composto por um conjunto de atributos e operações
- tipos:
  - NTFS
  - ext3
  - FAT*
  - UDF
  - APFS
  - …

> *Is the operating system's responsability to provide a set of from the file system point of view: system calls that implement such abstract data type. These system calls should be a simple and safe interface with the mass storage device. The component of the operating system dedicated size — the size in bytes of the file's data to this task is the file system*

Ou seja, operações de leitura e escrita **são sempre efetuadas no contexto de ficheiros**, através de syscall disponibilizadas pelo OS.

A interface de comunicação com o OS é a mesma, mas diferentes sistemas de ficheiros obrigam a diferentes técnicas e manipulação do filesystem, que são transparentes para o programador.

3.1 File as an abstract data type

Os atributos de um ficheiros dependem da implementação do sistema de ficheiros.

Os mais comuns:

- **name**:
- **internal identifier**: ID númerico (e interno - o user desconhece) que é usado pelo OS para aceder ao ficheiro
- **size**: tamanho do ficheiro em bytes
- **ownership**: Identificação de quêm o ficheiro pertence (usado para controlo de acessos)
- **permissions**: Atributos que em conjunto com a ownership (des)autorizam o acesso ao ficheiro
  - Possíveis permissões:
    * r: read
    * w: write
    * x: execute
    * d: directory
  - Nos diretórios, execução `x` significa que eu tenho permissões para atravessar o diretório (posso não ter permissões nem para ler nem para escrever, mas posso seguir no diretório para chegar a outro path)
- **acess monitoring**: data do último acesso e última modificação
- **localization of the data**: identificação dos clusters onde os dados do ficheiros estão guardados
- **type**: tipo dos ficheiros:
  1. ordinary or regular: qualquer ficheiro ”normal” para o utilizador [ID= -]
     - .txt
     - .doc
     - .png
     - .avi
     - .mp3
     - .pdf
     - .c
     - .exe
     - …
  2. directory: um tipo de **ficheiro** interno, com um formato pre-definido, usado para localizar outros ficheiros ou diretórios, permitindo visualizar o sistemas de ficheiros como uma árvore de diretórios e fichieros [ID= d]
  3. shortcut (symbolic link): ficheiro interno, com um formato predefinido, que contém uma referência para outro ficheiro/diretório [ID= s]
     - ref pode ser absoluta ou relativa
  4. character device(**special file**): *represents a device handles in bytes* [ID= c]
  5. block device(**special file**): *rep esents a device handles in block* [ID= b]
  6. socket(**special file**): *represents a file used for inter-process communication* [ID= s]
  7. named pipe: ***another special file*** *used for inter-process communication* [ID = p]

ID (`ls -ll`)	meaning
-	ordinary/regular file
d	directory
s	symbolic link

ID (`ls -ll`)	meaning
c	character device
b	block device
s	socket
p	named pipe

No sofs17 só serão considerados os três primeiros tipos de ficheiros.

3.1.1 Operações em ficheiros

- Dependem do OS
- Todas as operações estão disponíveis **apenas** através de syscalls (funções que funcionam como *entry-points* para o OS
- Syscalls em Linux para os tipos de ficheiros a usar no sofs17:

```
1  /*[TODO] Inserir descrição das operações para o teste*/
```

  - Comun aos três:
    * open
    * close
    * chmod
    * chown
    * utime
    * stat
    * rename
  - Comun para **ficheiros regulares** e **shortcuts**:
    * link
    * unlink
  - Só para **ficheiros regulares**:
    * mknod
    * read
    * write
    * truncate
    * lseek
  - Só para **diretórios**
    * mkdir
    * rmdir
    * getdents
  - Só para **shortcuts**
    * symlink
    * readlink

A descrição destas syscalls pode ser obtida executando num terminal o comando:

```
1  man 2 <syscall>
```

3.2 FUSE

Inserir um novo filesystem num OS requer: 1. Integração do software que implementa o novo filesystem no kernel 2. Instanciação de um ou mais dispositivos que usam o formato do novo filesystem

> *In monolitic kernels, the integration task involves the recompilation of the kernel, including the sofware that implements the new file system. In modular kernels, the new software should be compiled and linked separalely and attached to the kernel at run time.*

Tarefa morosa e difícil, que requer *deep knowledge of the hosting system* - **OUT OF THE SCOPE OF SO**

> FUSE (File system in User SpacE) is a canny solution that **allows for the implementation of file systems in user space** (memory where normal user programs run). Thus, **any effect of flaws of the suporting software are restricted to the user space, keeping the kernel imune to them.**

O novo filesystem é executado em cima do FUSE com permissões de user e não de root. Assim, certas operações que poderiam danificar fisicamente os dispositivos estão interditas e erros no código não geram kernel-panics.

Isola-se a execução deste novo filesystem do kernel.

3.2.1 Infrastrutura

- **Interface com o filesystem nativo**: funciona como mediador entre as syscalls do sistema nativo e as implementadas em user sapce
- **Implementation library**:
  - Estruturas de dados
  - Protótipos de funções (que devem ser desenvolvidas pelo user para criar o filesystem específico)
  - Métodos para instanciar e integrar o novo filesystem com o kernel

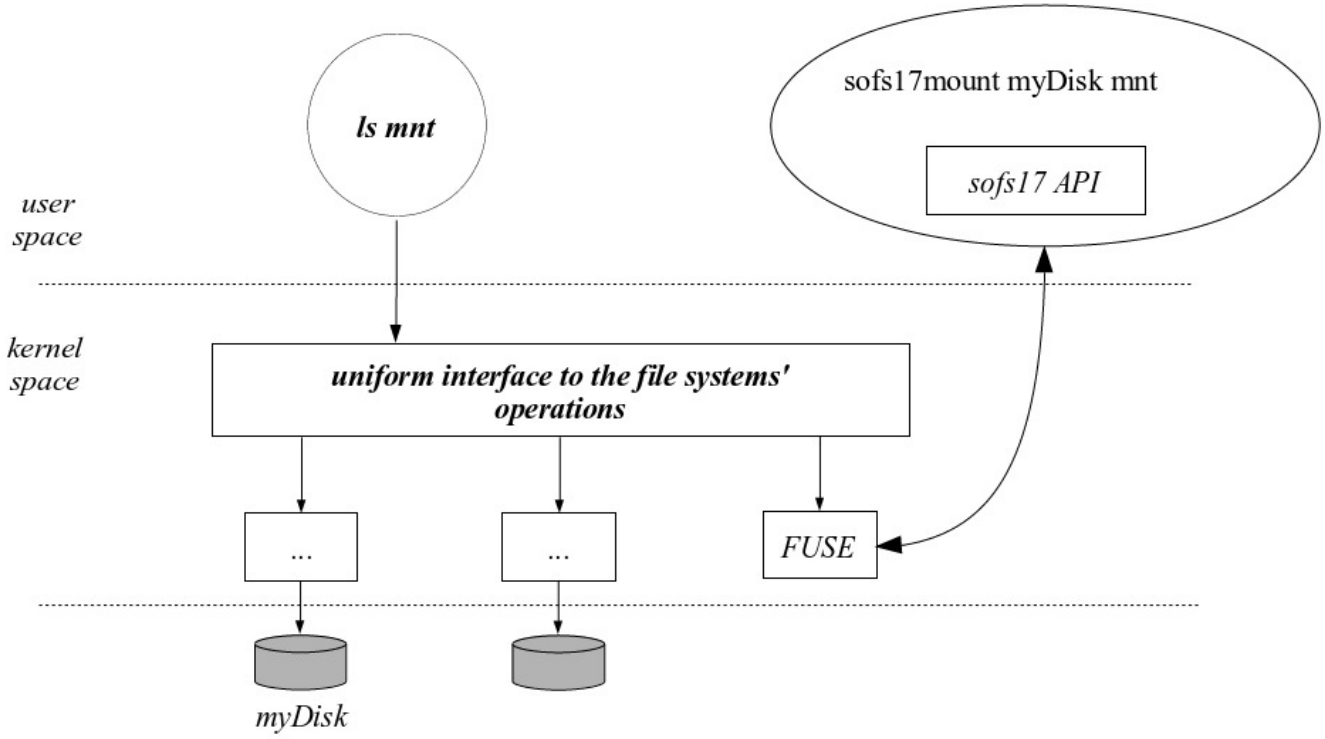

Figure 1: FUSE diagram with sofs17

4 SOFS17 Architecture

- Um disco é um conjunto de blocos numerados
  - No sofs17 cada bloco tem 512 bytes
- Os elementos principais na definição da arquitectura do sofs2017 são:
  - **superblock**: estrutura de dados guardada no bloco 0. Contém atributos globais para
    * o disco como um todo
    * outras estruturas de dados
  - **inode**: estrutura de dados que contém **todos os atributos de um ficheiro, excepto o nome**
    * Existe um região contínua no disco reservada para guardar todos os inodes (inode table)
    * A identificação de um inode é feita com um indíce que representa a sua posição relativa na inode table
  - **directory**: *special file* que permite a implementação de uma hierarquia (árvore) para acesso aos ficheiros
    * É composto por um conjunto de entradas (*directory entries*) em que cada uma associa um nome a um inode
    * Assume-se que o diretório de raiz (*root*) está associado ao inode 0
  - **disk blocks**: usados para guardar os dados
    * Estão organizados em grupos de 4 blocos contínuos -> **clusters**
    * A identificação de um cluster é dada através de um indíce que identifica a posição relativa do cluster na cluster zone
  - **cluster**: Para cada cluster existe um bit correspondente que representa o seu estado (vazio/preenchido)
    * Estes bits estão guardados no sistema de ficheiros numa área chamada *reference bitmpa table*

De forma geral, os N blocks de um disco formatado em sof17 organizam-se em 4 áreas:

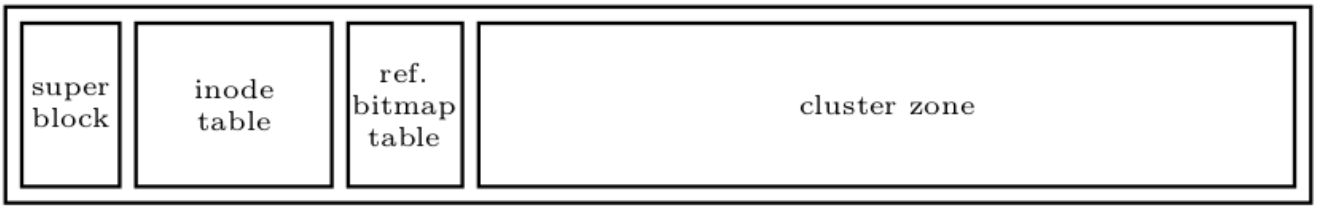

Figure 2: Organização de um disco formatado em sofs17

4.1 List of free inodes

- O número de inodes num disco sofs17 é **fixo após a formatação**.
- Quando um novo ficheiro é criado, deve-lhe ser atribuido um inode. Para isso é preciso:
  - Definir uma política (conjunto de regras) para decidir que free inode será usado
  - Definir e guardar no disco uma estrutura de dados adequada à implementação desta política

---

> *In sofs17 a FIFO policy is used, meaning that the first free inode to be used is the oldest one. The implementation is based in a double linked list of free inodes, built using the inodes themselves.*

- Na estrutura de inodes, existem dois campos que guardam os indíces do próximo inode e do inode anterior vazios (criam uma lista ligada)
- Estas listas ligadas de inodes são circulares, ou seja:
  - O *previous* inode livre do primeiro free inode é o último free inode
  - o *next* free inode do último inode é o primeiro free inode
  - Assim:
  - Cada numero da lista paonta sempre para o seguinte.
  - O previous aponta sempre para o elemento aterior.
  - Só preciso de saber a tail porque a previous do head é a tail
- No *superblock*, dois campos guardam o número total de free inodes e um indíce para o primeiro free inode
- O número de inodes por default é [NUM_BLOCKS]/8

Correspondência univoca entre o inode e o nome do ficheiro

- stat : mostra a estatísticas do ficheiro (filesize, blocks, ID Block, device, inode, links e datas de aceso, modificação e change)
  - Ficheiro `.` : diretório atual
  - Ficheiro `..` : diretório atual

4.2 List of free clusters

- Tal como os inodes, o número de clusters num disco é fixo após a formatação.
- Para manipular a estrutura de clusters é necessário:
  - Definir uma maneira de representar o estado (livre/usado) de todos os clusters no disco
  - Definir uma política para decidir que cluster (que estea livre) deve ser usado quanto é necessário um cluster
  - Definir e guardar no disco uma estrutura de dados adequada para representar o sistema de clusters e permitir a implementação dos pontos acima

Concretamente no sofs17:
- Existe uma estrutura de *bit map* unívoca que mapeia o estado de um cluster - Esta estrutura é formada por um blocos contínuos no disco (logo após a *inode table*) - Cada bit funciona como uma variável booleana que classifica o cluster que referencia como vazio ou ocupado - Existem duas caches guardadas no superblock que são usadas para guardar referências diretas para os clusters. São: - **retrieval cache**: - **insertion cache**: - Considera-se que um cluster está livre nas seguintes condições - A sua referência está em qualquer uma das caches - Ou o seu bit correspondente no bit map indica que está vazio

- As duas caches têm como função melhorar a eficiência de operações de **alocação**(atribiuir um cluster livre a um novo ficheiro a ser guardado em disco) e **libertação**(remover as referências para um dado cluster).
  - Na maioria das vezes as duas operações só precisam de aceder ao superblock e não fazem mais di que um acesso ao disco

4.2.1 Retrieval Chache

Serve para guardar as referências após eliminar um ficheiro. Se o disco tiver vazio, a referência deve ser max e não 0. O valor 0 significa que está cheio a retrieval cach está cheia.

4.2.2 Insertion cache

Serve para guardar as referências de ficheiros a inserir. Se o cache estiver vazia, a referência deve ser 0. O valor 0 significa que a insertion cache está cheia.

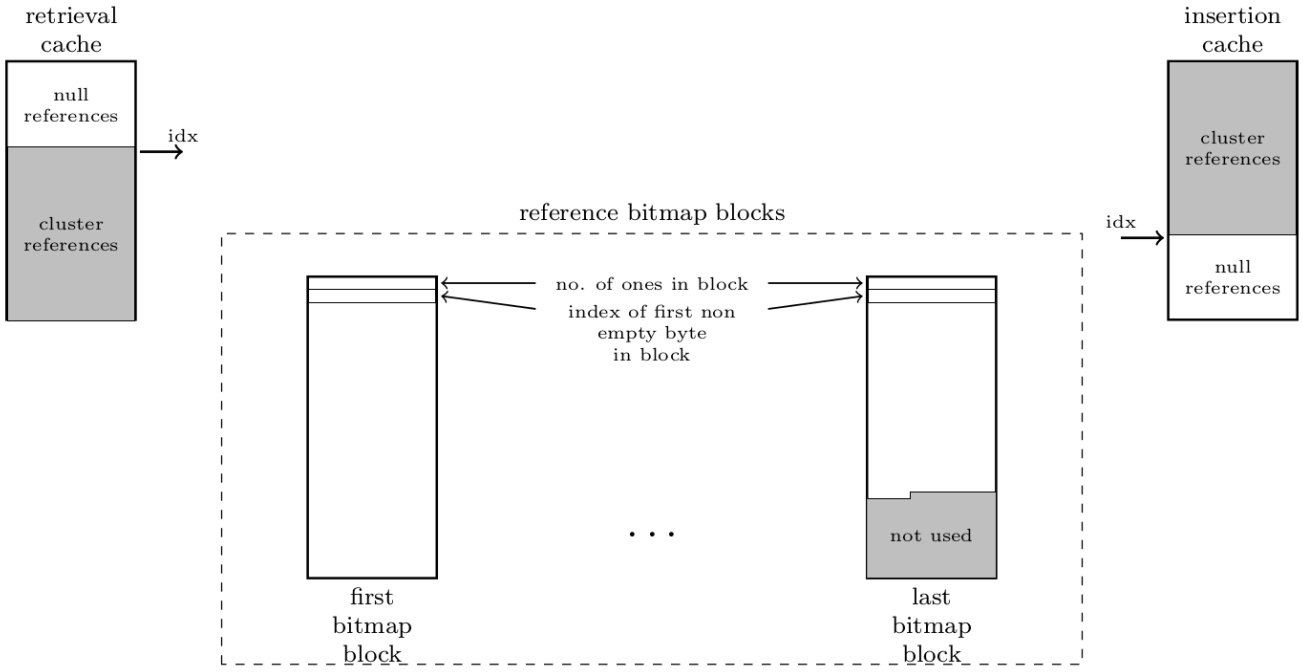

Figure 3: Caches and Reference bitmap blocks

4.2.3 Allocation

1. Uma referência para um cluster livre é obtida da retrieval cache
    1. Se a cache tiver vazia, são transferidas várias referências do bit map para a cache antes de se obter a referência para o cluster livre
    2. As referências transferidas para os clusters são transferidas de forma sequencial
2. Um byte iglobal (guardado no superblock) indica a localização no bit map de onde a transferência deve começar
    1. Assim cria-se rotatividade no uso dos clusters
3. Os clusters não funcionam estritamente como uma FIFO.
    1. Se a *retrieval cache* e o bit map estão vazios, as referências presentes na inserion cache são transferidas da insertion cache para o disco
    2. Se se efetua uma release operation a referência para o novo cluster livre é inserida na inserion cache
    3. Se esta cache está cheia, então as referências para a cache são transferidas para o bit map, antes de se proceder como anteriormente

4.3 List of clusters used by a file (inode)

> ***Clusters are not shared among files, thus, an in-use cluster belongs to a single file***

O número de clusters usado por um ficheiro é $N_c = roundup(\frac{size}{ClusterSize})$, onde:

- **size**: tamanho em bytes de um ficheiro
- **ClusterSize**: tamanho de um cluster em bytes

4.3.1 Considerações:

- N_c pode ser muito elevado.
  - Um disco com um block size de 512 bytes e com um *cluster size* de 4 blocos. Se o ficheiro a guardar tiver 2 GByte são necessários 1 milhão de clusters
- N_c pode ser nulo (0):
  - Se o ficheiro tiver 0 bytes, N_c = 0

> *Thus, it is impractical that all the clusters used by a file are contiguous in disk. The data structure used to represent the sequence of clusters used by a file must be flexible, growing as necessary.*

A **escrita** e a **leitura** no disco **não são sequenciais**, mas sim aleatórias.

Exemplo: pretendemos aceder ao indice *j* de um ficheiro. Para obter o cluster que contém esse ficheiro precisamos de saber o indice do cluster do ponto de vista de um ficheiro, $ClusterIndex = \frac{j}{ClusterSize}$

Para obter a localização do ficheiro no disco, temos de obter o número do cluster usando a estrutura do filesystem. e No sofs17:
- a *data strucuture* definida é dinâmica e permite uam identificação rápida de qualquer data cluster. - Cada inode permite o acesso a um array dinâmico, ***d***, que identifica a sequencia de clusters usados para guardar os dados associados com um ficheiro. - Sendo ***ClusterSize*** o tamanho em bytes de um cluster, temos:

Cluster	Descrição
d[0]	Número do cluster que contem os primeiros ClusterSize bytes
d[1]	Número do cluster que contem os segundos ClusterSize bytes
.	…
.	…
d[$N_c - 1$]	Número do cluster que contém os últimos ClusterSize bytes

O array ***d*** não é guardado num único local:
- Os **primeiros 6 elementos** são diretamente guardados no inode, no campo d (referência direta) - Os **próximos elementos, se existirem, são referenciados através dos campos: -** i_1**: referência indireta -** i_2**: referência indireta dupla

4.3.2 Campo i_1

- É usado para estender indiretamente o array ***d***
- O primeiro elemento, $i_1[0]$ é usado para referenciar um cluster onde cada bloco é um endereço para uma posição no disco (cluster) onde estão guardados os dados do ficheiro
- Permite extender o array d de d[6] para d[RPC+6-1]=d[RPC+5]
  - **RPC** é o número de referências para clusters que podem ser guardadas num cluster

4.3.3 Campo i_2

- Se mesmo assim não for possível guardar os dados do ficheiro usando referência indireta simples, pode ser usada referência indireta dupla.
- O campo i_2 do inode é usado para referenciar um cluster em que cada bloco do cluster referenciad um cluster de dados
- É usado para extender o array de referências indiretas i_1 usando as referências indiretas do cluster. Assim temos um array de referências indiretas de $i_1[1]$ até $i_1[RPC]$.
- O primeiro cluster do array de referências indiretas duplas é $i_1[1]$ ($i_1[0]$ corresponde às referências diretas).
  - Traduzindo para o array de ***d*** corresponde aos segmentos do array entre d[RPC + 6] e $d[2 * RPC + 5]$

4.3.4 NullReference

- É usada para representar uma referência que não existe
- Exemplos:
  - se d[1] for uma NullReference, o ficheiro não contém o index de cluster 1
  - se i_1, representando $i_1[0]$ é equal to NullReference, significa que entre d[6] até d[RPC+5] todos os indices são NullReference e o o ficheiro não contém estes indices
  - se i_2 for uma NullReference, significa que entre $i_1[1]$ to $i_1[RPC]$ são NullReferences e portanto d[RPC+6] até d[RPC^2 + RPC+ 5] são NullReferences e o ficheiro não contém esses indices

4.4 Directories

- Um diretório pode ser visto como um array de entrada para diretórios.
- No sofs17:
  - Um diretório é uma estrutura de dados composta por um array de bytes com tamanho fixo. Usados para guardar:
    * nome
    * referência que associa o diretório a um inode
  - A estrutura de dados foi definida para que um cluster suporte apenas um número inteiro de diretórios
    * As primeiras duas entradas `"."` e `".."` representam o diretório atual e o diretório pai
    * Um diretório pode tomar um de três estados:
      · **in-use**: contém o nome e o inode number de um ficheiro que existe (seja ele um regular file, diretório ou atalho)
      · **deleted**: o nome contém o 1 e último caracter trocado, passando a ser `\0 name[0:end-1]` (ou seja, uma null string, mas com o nome recuperável). Mantém o slot no diretório.
      · **clean**: Todos os caracteres do nome são `\0` e o reference field é uma NullReference

  - Quando um cluster é adicionado a um diretório, primeiro deve formatado como uma sequência de diretórios de entrada limpas.
    * O tamanho do diretório é sempre um múltiplo do tamanho de um cluster e nunca pode “encolher” (devido à forma como o delete está implementado

5 Formatting

A operação de formatação deve preencher todos os blocos do disco para criar um disco sofs17 vazio.

Um disco formatado contém:
- root directory
- duas entradas, `"."` e `".."`, que apontam para o inode 0 (root directory)

A operação de formatação deve:

- escolher o valor apropriado para o número de inodes, o número de clusters e o número de blocos usados pelo bit map
  - tem de ter em consideração o número de inodes especificado pelo utilizador e número total de blocos no disco.
  - todos os blocos no disco devem ser usados. Se não forem usados para outros propósitos devem ser adicionados à inode table
- Preencher a tabela de inodes:
  - inode number 0 é o root directory
  - todos os outros inodes estão livres
  - A lista de inodes livres começa no inode número 1 e termina no último inode
- Preencher o bit map, sabendo que:
  - O cluster 0 está a ser usado pelo diretório root
  - Todos os outros clusters estão livres
- Preencher o root directory, ocupando o cluster número 0
- Preencher com zeros todos os clusters livres, se especificado pela ferramenta de formatação

6 Code Structure

A estrutura do código é apresentada abaixo:

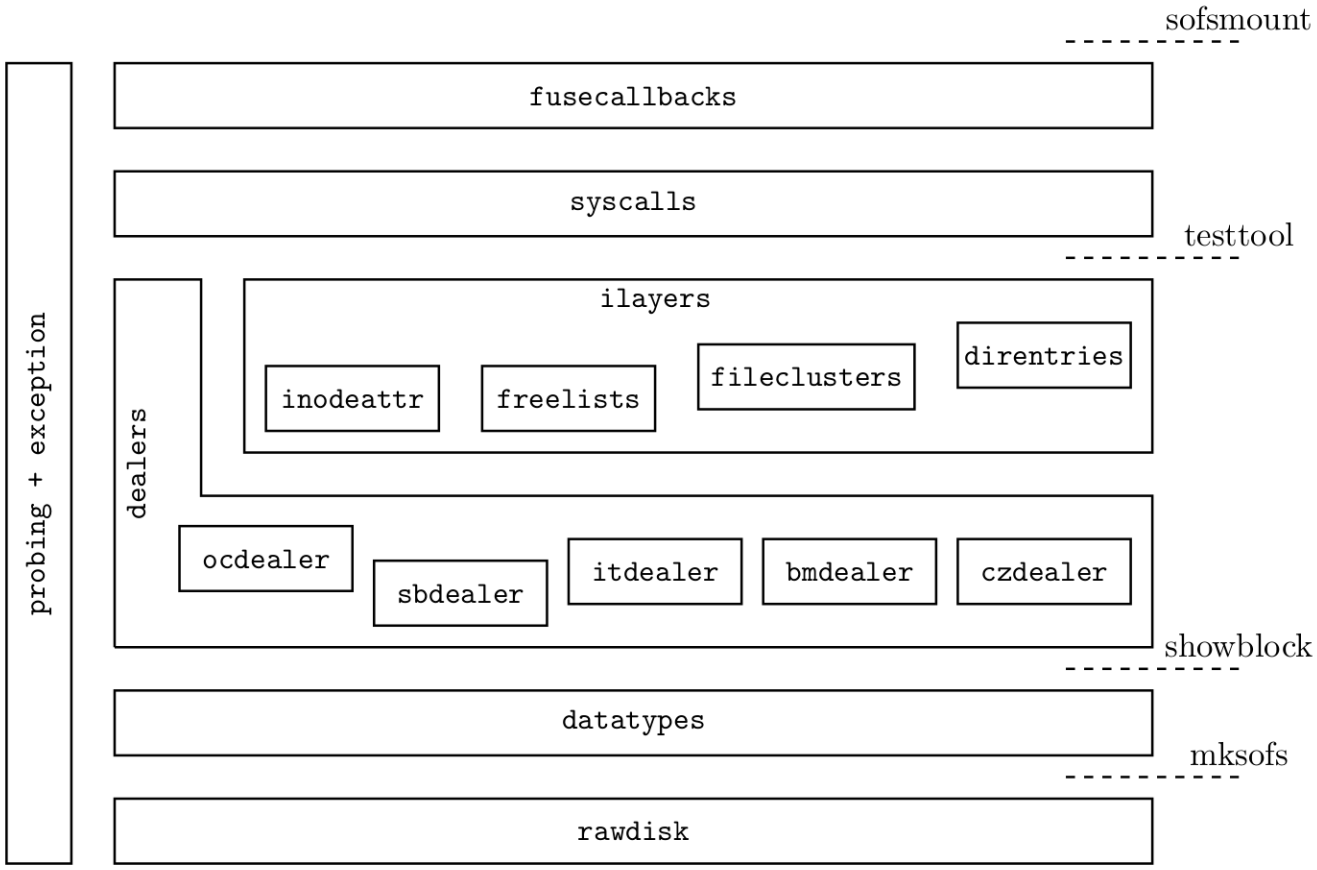

Figure 4: Code Strucuture to be developed

6.1 Rawdisk

Implementa o acesso físico ao disco

6.2 Dealers

- Implementam o acesso ao superblock, inodes, bit map e clusters
- São opcionais (só são feitas se os alunos desejarem ter notas mais altas)

6.2.1 sbdealer

Acesso ao superblock

6.2.2 itdealer

Acesso à inode table e aos inodes

6.2.3 bmdealer

Acesso às referências da bit map table

6.2.4 czdealer

Acesso à cluster zone, usando as cluster references

6.2.5 ocdelaer

Open/close the dealers

6.3 ilayers

Funções intermédias Obrigatórias

6.3.1 inodeattr

Lida com a manipulação dos campos especiais dos inodes

6.3.2 freelists

Manipular a lista dos inodes livres e a lista de clusters livres

6.3.3 filecluster

Lidar com os clusters de um inode (file clusters associados a um ficheiro)

6.3.4 direntries

Lidar com entradas de diretórios

6.4 syscalls

versão das syscalls de sistema adaptadas ao *sofs17* Cada grupo Só irá implementar 6 das 24 utilizadas.

6.5 fusecallbacks

Interface com FUSE

6.6 probing

Biblioteca para debug

6.7 exception

o tipo de exceções lançadas em caso de erro

- **datatypes**: um conjunto de constantes que podem ser usadas para aceder aos ficheiros
  - InodesPerBlock
  - ReferencesPerBlock
  - ReferencesPerCluster
  - ReferencesPerBitmapBlock
  - BlocksPerCluster
  - CLusterSize
  - DirentriesPerCluster
  - NullReference

7 createDisk

- Cria um disco **não formatado** que serve de suporte a um sistema de ficheiros.
- Na prática, um disco é um ficheiro que possui uma estrutura de blocos fixa.
- Apenas é garantida que a estrutura do disco possui:
  - o número desejado de clusters
  - o número desejado de bytes por cluster
- Para o disco ser um sistema de ficheiros válido é necessário formatá-lo com ferramentas adequadas para o tipo de sistemas de ficheiros pretendido

7.1 Exemplo de utilização

```
./createDisk <diskfile> <numblocks>
```

O *output* após a execução do script para um disco com 1000 blocos é:

```
./createDisk <diskfle> 1000
1000+0 records in
1000+0 records out
512000 bytes (512 kB) copied, 0.05734 s, 8.9 MB/s
```

7.2 Implementação

O createDisk usa o comando *dd* para escrever para o disco/ficheiro e preenche-o com valores aleatórios obtidos do */dev/u-random*.

```
#!/bin/bash

if [ $# != 2 ]; then
        echo "$0 diskfile numblocks"
```

```
        exit 1
fi

dd if=/dev/urandom of=$1 bs=512 count=$2
```

8 showblock

- Permite visualizar a informação contida numa sequência de blocos do disco:
  - Os dados dos blocos podem ser formatados para serem facilmente interpretáveis por humanos
  - A formatação dos dados dos blocos pode ser feita de acordo com a função de cada um dos blocos

8.1 Utilização

```
# showblock -h imprime a ajuda
./showblock [ OPTION ] <disk filename>
```

8.1.1 Opções de Visualização

Option	Description
-x	show block(s) as hexadecimal data
-a	show block(s) as ascii/hexadecimal data
-s	show block(s) as superblock data
-i	show block(s) as inode entries
-d	show block(s) as directory entries
-r	show block(s) as cluster references
-b	show block(s) as bitmap references

8.2 Exemplos

```
# Showblock para o bloco 1 em hexadecimal
./showblock -x 1 disk.sofs17
0000: fd 41 02 00 e8 03 00 00 e8 03 00 00 00 08 00 00 01 00 00 00 dc ee f9 59 dc ee f9 59
    dc ee f9 59
0020: 00 00 00 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
0040: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00
    8f 00 00 00
0060: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
```

```
0080: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 00
    01 00 00 00
00a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
00c0: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00
    02 00 00 00
00e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
0100: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 00 00 00
    03 00 00 00
0120: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
0140: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00
    04 00 00 00
0160: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
0180: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 00 00 00
    05 00 00 00
01a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff
01c0: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00
    06 00 00 00
01e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff

# A informação não é diretamente percetível por humanos.
# Sabendo que este bloco corresponde ao primeiro bloco da inode table,
# se executarmos o mesmo comando mas o output vier formatado para inodes, temos:
./showblock -i 1 disk.sofs17
Inode #0
type =  directory, permissions = rwxrwxr-x, lnkcnt = 2, owner = 1000, group = 1000
size in bytes = 2048, size in clusters = 1
atime = Wed Nov  1 15:57:16 2017, mtime = Wed Nov  1 15:57:16 2017, ctime = Wed Nov  1
    15:57:16 2017
d[] = {0 (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #1
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 2, , prev = 143
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #2
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 3, , prev = 1
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #3
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 4, , prev = 2
```

```
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #4
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 5, , prev = 3
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #5
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 6, , prev = 4
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #6
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 7, , prev = 5
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
Inode #7
type = free clean, permissions = ---------, lnkcnt = 0, owner = 0, group = 0
size in bytes = 0, size in clusters = 0
next = 8, , prev = 6
d[] = {(nil) (nil) (nil) (nil) (nil) (nil)}, i1 = (nil), i2 = (nil)
----------------
```

9 rawlevel

- Camada que permite a manipulação do disco ao nível do bloco

9.1 Módulos

- **mksofs** : Formatador
- **rawdisk** : Acesso aos blocos do disco

10 rawdisk

- Permite o acesso aos blocos do disco
  - Os blocos são a menor unidade lógica no *filesystem*
- Medeia o acesso direto ao disco, impedindo que ocorram erros que podem danificar a estrutura do sistema de ficheiros

10.1 Macros

```
// block size (in bytes)
#define BlockSize (512U)
```

10.2 Funções

```
void soOpenRawDisk (const char *devname, uint32_t *np=NULL)
```

- Abre o dispositivo de armazenamento, criando um canal de comunicação com esse dispositivo
  - Supoem que o dispositivo está fechado e mais nenhum canal de comunicação para esse dispositivo está aberto
  - O dispositivo de armazenamento tem de existir
  - O dispositivo de armazenamento tem de ter um tamanho múltiplo do block size

10.2.1

```
void soCloseRawDisk (void)
```

- Fecha o dispositivo de armazenamento e o canal de comunicação.

10.2.2

```
void soReadRawBlock(uint32_t n, void *buf)
```

- Lê um bloco de dados do dispositivo
- **Parametros:**
  - *n*: número físico do bloco de dados no disco de onde a informação vai ser lida
  - *buf*: ponteiro para o buffer para onde os dados vão ser lidos

10.2.3

```
void soWriteRawBlock ( uint32_t  n, void * buf)
```

- Escreve um bloco de dados do dispositivo
- **Parametros:**
  - *n*: número físico do bloco de dados no disco onde a informação vai ser escrita
  - *buf*: ponteiro para o buffer que contém os dados a ser escritos # msksofs
- Script responsável por formatar o disco
- Cria um disco utilizável
  - Manipula os blocos do disco para implementar o *sofs17 filesystem*

10.3 Utilização

```
USAGE:
Sinopsis: mksofs [OPTIONS] supp-file
  OPTIONS:
   -n name --- set volume name (default: "sofs17_disk")
   -i num  --- set number of inodes (default: N/8, where N = number of blocks)
   -z      --- set zero mode (default: not zero)
   -q      --- set quiet mode (default: not quiet)
   -h      --- print this help
```

- Posso usar mais do que uma das opções na mesma execução ## Exemplos

10.3.1 No Options

- Basta indicar o número do ficheiro
- Por default, o número de inodes é o número de clusters/8

```
./mksofs ../disk.sofs17

Trying to install a 125-inodes SOFS17 file system in ../disk.sofs17.
Computing disk structure...
Filling in the superblock fields...
Filling in the table of inodes...
Filling in the bitmap of free clusters...
Filling in the root directory...
144-inodes SOFS17 file system was successfully installed in ../disk.sofs17.
```

10.3.2 Set name

```
$ ./mksofs.bin64 disk.sofs17 -n "my disk"

Trying to install a 125-inodes SOFS17 file system in disk.sofs17.
  Computing disk structure... done.
  Filling in the superblock fields... done.
  Filling in the table of inodes... done.
  Filling in the bitmap of free clusters... done.
  Filling in the root directory... done.
A 144-inodes SOFS17 file system was successfully installed in disk.sofs17.

$ ./showblock disk.sofs17 -s 0
Header:
   Magic number: 0x50F5
   Version number: 0x2017
   Volume name: my disk
   Properly unmounted: yes
   Number of mounts: 0
   Total number of blocks in the device: 1000
Inode table metadata:
```

```
20     First block of the inode table: 1
21     (...)
```

10.3.3 Set inodes

- O formatador tenta formatar o disco para o número desejado de inodes

```
1  ./mksofs.bin64 disk.sofs17 -i 2000
2
3  Trying to install a 2000-inodes SOFS17 file system in disk.sofs17.
4    Computing disk structure... done.
5    Filling in the superblock fields... done.
6    Filling in the table of inodes... done.
7    Filling in the bitmap of free clusters... done.
8    Filling in the root directory... done.
9  A 2000-inodes SOFS17 file system was successfully installed in disk.sofs17.
```

- Pode não ser possível formatar o disco para o número desejado de inodes
- Nesse caso, o formatador usa o número de inodes possível imediatamente superior ao pretendido

```
1  ./mksofs.bin64 disk.sofs17 -i 100
2
3  Trying to install a 100-inodes SOFS17 file system in disk.sofs17.
4    Computing disk structure... done.
5    Filling in the superblock fields... done.
6    Filling in the table of inodes... done.
7    Filling in the bitmap of free clusters... done.
8    Filling in the root directory... done.
9  A 112-inodes SOFS17 file system was successfully installed in disk.sofs17.
```

10.3.4 Zero Mode

- Ao usar a opção `-z` todos os clusters livres são preenchidos com zeros

```
1  ./mksofs ../disk.sofs17 -z
2
3  Trying to install a 125-inodes SOFS17 file system in ../disk.sofs17.
4  Computing disk structure...
5  Filling in the superblock fields...
6  Filling in the table of inodes... .
7  Filling in the bitmap of free clusters...
8  Filling in the root directory...
9  Filling in free clusters with zeros... cstart: 24, ctotal: 244
10 A 144-inodes SOFS17 file system was successfully installed in ../disk.sofs17.
```

11 computeStruture

- Calcula a divisão da estruturas no disco

  - número de clusters
  - número de blocos para inodes
  - número de blocos para reference map
-

11.1 Algoritmo

- No mínimo têm de existir 6 blocos no disco
  - 1 superblock
  - 0 inodes
  - 1 reference map
  - 1 cluster de dados
- Por default o número de inodes é $N_{inodes} = \frac{N_{clusters}}{8}$
- Caso o número de inodes não seja divisível por 8, é preciso alocar mais um bloco para os inodes
- O número temporário de blocos livres (falta o reference map) é:

$$N_{blocosdisco} - N_{blocosinodes} - 1$$

  - o "1" corresponde ao superblock
- O número de clusters é o resultado da divisão do número de blocos livres pelo número de blocos por cluster
- Através do número de clusters pode ser estimado o número de blocos necessários para a reference map
- Depois dessa estimativa é possível calcular o número de blocos restantes e atribuí-los à inode table

11.2 Utilização

```
1 void computeStructure(  uint32_t    ntotal,
2                         uint32_t    itotal,
3                         uint32_t *  itsizep,
4                         uint32_t *  rmsizep,
5                         uint32_t *  ctotalp
6                       )
```

11.2.1 Parameters

- **ntotal:** total number of blocks of the device
- **itotal:** requested number of inodes
- **itsizep:** pointer to mem where to store the size of inode table in blocks
- **rmsizep:** pointer to mem where to store the size of cluster reference table in blocks
- **ctotalp:** pointer to mem where to store the number of clusters

11.3 Testes

11.3.1 1000 blocos, 125 inodes (nblocos/8)

- Começamos por calcular o número de blocos necessários para os inodes

  - Existem 8 inodes por bloco

```
125 / 8
120   15
  5
```

- Obtemos 15 blocos para inodes
- E 5 blocos que sobram
- Se permitimos que 4 sejam usados para um cluster, temos 16 inodes
- O número de clusters é $1000 blocos - 16 inodes - 1 superblock = 983 blocos$
- O número de clusters para dados é:

```
983 / 4
980   245
  3
```

- Passamos a ter um sistema de ficheiros $15 + 3 = 18 blocos para inodes$
  - Isto equivale a ter $18 \times 8 = 144 inodes$ e não os 125 como inicialmente se desejava

12 fillInSuperBlock

- Preenche os campos dos superblock
- O *magic number* deve ser 0xFFFF
- As caches estão no *superblock*

12.1 Algoritmo

- Atribuições a serem feitas:
  - o magic number (identifica se o sistema é Big-Endian ou Little-Endian)
  - *version number*
  - nome do disco
    * Tem de ser truncado caso ultrapasse o *PARTITION_NAME_SIZE*
  - Número total de blocos
- Indicar que o disco ainda está unmounted
- Reset ao número de mounts
- Inode table metadata
  - **itstart:** Bloco onde começa a tabela de inodes
  - **itsize:** Número de blocos da inode table
  - **itotal:** Número total de inodes
  - **ifree:** Número de inodes livres
  - **ihead:** Índice para a head do primeiro inode
- Free Cluster table metadata

  - **rmstart:** bloco onde começa a reference map
  - **rmsize:** número de blocos usados pela reference table
  - **rmidx:** Primeira referência (*root dir*)

- Clusters metadata
  - **czstart:** bloco onde começa a cluster zone
  - **ctotal:** número total de clusters
  - **cfree:** número de clusters livres

- Retrieval cache
  - Inicializar com NullReferences
  - idx -> última posição da cache

- Insertion cache
  - Inicializar com NullReferences
  - idx -> primeira posição da cache

12.2 Utilização

```
void fillInSuperBlock(  const char * name,
                        uint32_t     ntotal,
                        uint32_t     itsize,
                        uint32_t     rmsize
                      )
```

12.2.1 Parameters

- **name:** volume name
- **ntotal:** the total number of blocks in the device
- **itsize:** the number of blocks used by the inode table
- **rmsize:** the number of blocks used by the cluster reference table

13 fillInInodeTable

- Preenche os blocos da inode table
- O inode **0** deve ser preenchido considerando que está a ser usado pelo diretório raiz
- Todos os outros inodes estão livres

13.1 Algoritmia

- Para cada bloco da inode table
  - Criar a lista biligada
  - Referências para os clusters:
    * Preencher com NullReference as *direct references (d)*

    * Preencher com NullReference as *indirect references (i1)*
    * Preencher com NullReference as *double direct references (i2)*

- Inicializar o inode da root directory (*inode 0*)
  - **mode:** permissões
  - **lnkcnt:** link count - número de caminhos que chegam a este (2: `.`, `..`)
  - **owner:**
  - **group:**
  - **size:** Tamanho do inode (1 cluster)
  - **clucnt:** file size in bytes
  - Modificar access times
  - Apontar para o root dir usando as referências diretas (_d[0])

13.2 Utilização

```
1 void fillInInodeTable(  uint32_t itstart,
2                         uint32_t itsize
3                       )
```

13.2.1 Parameters

- **itstart:** physical number of the first block used by the inode table
- **itsize:** number of blocks of the inode table

14 fillInFreeClusterTable

- Preeche a Free Cluster Table:
  - Estrutura que indica se os clusters estão livres ou ocupados
- Existe uma correspondência unívoca entre bits na *reference cluster table* e clusters no disco
- Os bits na tabela de referências crescem da:
  - Lower blocks to upper blocks
  - Lower bytes to upper bytes
  - Most Significant Bytes (MSB) to Least Significant Bytes (LSB)
- Assim, o bit "0" corresponde ao bit mais significativo do primeiro byte do primeiro bloco da tabela de bitmap
- Valor do bit:
  - **"1":** Cluster Livre
  - **"0":** Cluster Ocupado
- Em geral, o número de bits na tabela é maior que o número de clutsers
  - Os bits não usados (ou seja, que não correspondem ao estado de nenhum cluster) devem ser inicializados como se fosse usados
- Pode existir mais do que um bloco para a reference map table

14.1 Algoritmo

- Começa-se por definir algumas constantes:

```
#define CLUSTER_IN_USE 0
#define CLUSTER_FREE   1

#define BYTE_FREE   0xFF
#define BYTE_IN_USE 0x00

#define ROOT_DIR_MAP_MASK 0x7F
```

- Calcula-se:
  - Número de Clusters que não ficam referenciados num bloco completo

$$nExtraClusters = c_{total}\%ReferencesPerBitmapBlock$$

  - Número de Clusters da reference zone que estão totalmente ocupados

$$nFullRefBlocks = \frac{c_{total}}{ReferencesPerBitmapBlock}$$

  - Número de Blocos para a Reference Bitmap zone

$$nRefBlocks = nFullRefBlocks + (nExtraClusters! = 0)$$

  - Byte no último bloco de referências onde começam as referências não válidas

$$byteStartFreeBitmapPos = \frac{nExtraClusters}{8}$$

  - Bit no byte acima onde começam as referências não válidas

$$bitStartFreeBitmapPos = nExtraClusters\%8$$

- Blocos de referências completos
  - Para cada bloco de referências completo o número de referências é ReferencesPerBitmapBlock
  - O indice é 0 (o primeiro cluster livre está no início do bloco)
- Bloco parcialmente completo
  - Preencher o cnt com o número de clusters que esse bloco tem
  - Colocar os clusters não válidos como usados
- Referência do cluster 0
  - Indicar que está a ser usada pela root dir
  - Decrementar o count, porque existe menos um cluster vazio

14.1.1 Considerações

- O número de clusters pode ser inferior ou superior ao tamanho de um bloco.
- O primeiro bit do primeiro bloco deve estar em use
- Todos os bits que não referenciem um cluster devem ser colocados como em uso

14.2 Utilização

```
void fillInFreeClusterTable(uint32_t rmstart,
                            uint32_t ctotal
                           )
```

14.2.1 Parameters

- **rmstart:** the number of the fisrt block used by the bit table
- **ctotal:** the total number of clusters

14.2.2 Data Structure

SORefBlock : estrutura dos Reference bitmap Block data type

```
struct SORefBlock
{
    /** \brief number of references in block */
    uint16_t cnt;
    /** \brief index of first non-empty byte */
    uint16_t idx;
    /** \brief bit map */
    uint8_t map[ReferenceBytesPerBitmapBlock];
};
```

14.3 Testes

```
# 1000 blocks, 144-inodes, mksofs.bin64
block range: 19
cnt = 244, idx = 0
0000: 7f ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff f8 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
00a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
00c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
00e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
```

```
0120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
01a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
01c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
01e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00


# 200 blocks, 48-inodes, mksofs.bin64
block range: 7
cnt = 47, idx = 0
0000: 7f ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
00a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
00c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
00e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
01a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
01c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
01e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00


# 10000 blocks, 1264 inodes, mksofs17.bin64
```

```
block range: 159
cnt = 2459, idx = 0
0000: 7f ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0020: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0060: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0080: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00c0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0100: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0120: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff f0 00 00 00 00 00 00 00 00 00 00 00
0140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

15 fillInRootDir

- A root dir ocupa um cluster
- Os dois primeiros slots estão reservados para as entradas "." e ".."
- Todos os os outros slots devem estar limpos:
  - o campo *name* preenchido com zeros
  - o campo *inode* preenchido com NullReference

15.1 Algoritmia

- Colocar todos os blocos com zeros
- Na entrada 0
  - nome: "."

  - inode: 0 (raiz)

- Na entrada 1
  - nome: “..”
  - inode: 0 (raiz)

15.2 Utilização

```
1 void fillInRootDir(uint32_t rtstart)
```

15.2.1 Parameters

- **rtstart:** number of the block where the root cluster starts.

16 resetClusters

- Escrever com zeros um conjunto de clusters

16.1 Algoritmia

- Percorrer a sequência de clusters desejada e escrever zeros

16.2 Utilização

```
1 void resetClusters( uint32_t cstart,
2                     uint32_t ctotal
3                   )
```

16.2.1 Parameters

- **cstart:** number of the block of the first free cluster
- **ctotal:** number of clusters to be filled

17 freelists

- Funções para manipular a lista de inodes livres e a lista de clusters livres.
- A lista de inodes livres é mantida usando uma lista biligada de inodes
  - Política FIFO
- A lista de clusters é mantida com duas caches:
  - Retrieval Cache: Clusters livres para serem alocados

  - Insertion Cache: Clusters que foram libertados
- A lista de clusters segue uma estrutura parecida com FIFO
  - Não é bem FIFO porque existe rotatividade nos clusters
    * Impede que exista uma escrita desigual nos clusters
    * Antes de um cluster libertado puder voltar a ser escrito, todos os outros clusters no disco têm de ser escritos
    * Aumenta o tempo útil do disco

17.1 soAllocInode

uint32_t soAllocInode (uint32_t type)

Allocate a free inode.

An inode is retrieved from the list of free inodes, marked in use, associated to the legal file type passed as a parameter and is generally initialized.

Parameters

```
1       type the inode type (it must represent either a file, or a
2            directory, or a symbolic link)
```

Returns the number of the allocated inode

17.2 soFreeCluster

void soFreeCluster (uint32_t cn)

Free the referenced cluster.

Parameters

```
1        cn the number of the cluster to be freed
```

17.3 soFreeInode

void soFreeInode (uint32_t in)

Free the referenced inode.

The inode is inserted into the list of free inodes.

Parameters

```
1        in number of the inode to be freed
```

17.4 soReplenish

void soReplenish ( )

replenish the retrieval cache

References to free clusters should be transfered from the free cluster table (bit map) or insertion cache to the retrieval cache. Nothing should be done if the retrieval cache is not empty. The insertion cache should only be used if there are no bits at one in the map. Only a single block should be processed, even if it is not enough to fulfill the retrieval cache. The block to be processes is the one pointed to by the rmidx field of the superblock. This field should be updated if the processing of the current block reaches its end.

17.5 soDeplete

[MISING IN DOXYGEN]

Functions

uint32_t soAllocInode (uint32_t type) Allocate a free inode. More…

```
1    void  soFreeInode (uint32_t in)
2          Free the referenced inode. More...
```

uint32_t soAllocCluster () Allocate a free cluster. More…

```
1    void  soReplenish ()
2          replenish the retrieval cache More...
3
4    void  soFreeCluster (uint32_t cn)
5          Free the referenced cluster. More...
6
7    void  soDeplete ()
8          Deplete the insertion cache.
```

Detailed Description

Functions to manage the list of free inodes and the list of free clusters.

18 Cenário Inicial

campo ihead superblock: 101 campo ifree : 3

inode previous|next

101 7|5

5 10|7

7 5/101

18.1 freeinode

- Quero libertar o nó numero 200
- mante o tipo
- apenas altera flag para free
- o ficheiro passa a ser deleted file
- o nó que é libertado é obviamente o ultimo

```
# Estado inicial
----------------
Inode #3
type =  regular file, permissions = rw-rw-r--, lnkcnt = 1, owner = 1000, group = 1000
size in bytes = 42000, size in clusters = 7
atime = Thu Oct 26 23:02:47 2017, mtime = Thu Oct 26 23:02:47 2017, ctime = Thu Oct 26
    23:02:47 2017
d[] = {11 (nil) (nil) (nil) 12 13}, i1 = 14, i2 = (nil)
----------------
Após chamar o freeinode para o inode 3
---------------
Inode #3
type = free regular file, permissions = rw-rw-r--, lnkcnt = 1, owner = 1000, group = 1000
size in bytes = 42000, size in clusters = 7
next = 10, , prev = 1
d[] = {11 (nil) (nil) (nil) 12 13}, i1 = 14, i2 = (nil)
----------------
- Mantem o size in clusters
```

18.2 inserir inode 200

200 7|101

101 200|5

5 10/7

7 5|200

200 7|101

ihead:101 ifree: 4

18.3 soAllocateInode

- Retirar o nó da lista de inodes
- O primeiro no a retirar é o que apontado pelo ihead

ihead:5 ifree:2

5 7|7

7 5|5

Só pode existir uma cópia do suoerbloco

18.4 iOpen

- Abrir o inode
  - so o volta a ler do disco se já foi aberto
  - só posso pedir um pointer para esse inode
  - devolve me um inode handler
  - in: inode number
  - ih: inode handler
  - ip: inode pointer

18.5 iSave

- Guardar um inode

18.6 iClose

- Guardar o ficheiro

18.7 Interface com os inodes é suposto usar uma estrutura de inodes

função replentish tem como função transferir blocos para a cache - transforma bits em referências - os bits que forem transferidos vão passar de 1 a zero - rmidx: primeiro byte do map de bit que tem bits a 1 - comece por aqui. Antes não há bits a 1

19 mais difícis (5)

soReplenish (Patricia) soDeplete (Bernardo)

20 intermédias (3)

soAllocateInode (panda) soFreeInode (Gradim)

21 mais triviais (1)

soAllocClusters (Pedro) soFreeClusters (Mica) # AllocInode - Aloca um inode livre da estrutura de inodes -

21.1 Utilização

```
1 uint32_t soAllocInode ( uint32_t  type )
```

Allocate a free inode.

An inode is retrieved from the list of free inodes, marked in use, associated to the legal file type passed as a parameter and is generally initialized.

Parameters

```
1       type the inode type (it must represent either a file, or a directory, or a symbolic
            link)
```

Returns the number of the allocated inode

- Se for possível, aloca o inode que está na HEAD
- Incrementa a HEAD
  - tem de verificar se a inode table está no fim e se tem de voltar aos inodes que entretanto foram libertados
- Decrementa o número de inodes livres
- O inode tem de ser corretamente inicializado # Inodes
- Existem 6 posições para referência direta aos clusters do ficheiro
  - d[0 … 5]
- Uma posição para referência indireta
  - i1
  - Extende o array de d[6 … 517]
-

21.2 Uma posição para referência dupla indireta

- Cada ficheiro possui um inode
  - O número máximo de ficheiros num disco é o número máximo de inodes
- Um inode ocupa 64 bytes
  - Logo num disco com 512 bytes por bloco, existem 8 inodes em cada bloco # Fileclusters Functions to manage the clusters belonging by a file

21.3 Doxygen

uint32_t soGetFileCluster (int ih, uint32_t fcn) Get the cluster number of a given file cluster. More…

uint32_t soAllocFileCluster (int ih, uint32_t fcn) Associate a cluster to a given file cluster position. More…

```
1   void  soFreeFileClusters (int ih, uint32_t ffcn)
2         Free all file clusters from the given position on.
3         More...
4
5   void  soReadFileCluster (int ih, uint32_t fcn, void *buf)
6         Read a file cluster. More...
```

```
7
8   void  soWriteFileCluster (int ih, uint32_t fcn, void *buf)
9         Write a data cluster. More...
```

Detailed Description

Functions to manage the clusters belonging by a file.

Author Artur Pereira - 2008-2009, 2016-2017 Miguel Oliveira e Silva - 2009, 2017 António Rui Borges - 2010-2015

Remarks In case an error occurs, every function throws an SOException

Function Documentation

uint32_t soAllocFileCluster (int ih, uint32_t fcn)

Associate a cluster to a given file cluster position.

Parameters

```
1       ih  inode handler
2       fcn file cluster number
```

Returns the number of the allocated cluster

void soFreeFileClusters (int ih, uint32_t ffcn)

Free all file clusters from the given position on.

Parameters

```
1       ih   inode handler
2       ffcn first file cluster number
```

21.3.1 uint32_t soGetFileCluster (int ih,

```
1                           uint32_t  fcn
2                         )
```

Get the cluster number of a given file cluster.

Parameters

```
1       ih  inode handler
2       fcn file cluster number
```

Returns the number of the corresponding cluster

- Parece me que apenas tenho de retornar o endereço do cluster
- Não é preciso retornar tudo
- Mandar mail ao professor
- Perguntar panda
- Para que servem as fnções do prof??
- Aquilo que faz é usar o inode handler para saber onde está no disk e o file cluster number para obter a referência
- É preciso rever as duas estruturas

  - superblock
  - inode
  - cluster

O que é preciso fazer:

- Obter o inode
- Se o cluster index estiver nos 6 primeiros
  - Sai direto da estrutura de inodes
- Se o cluster index for referenciado diretamente (i_1)
  - está no cluster de referências
  - Ler esse cluster do disco
  - Calcular novo index (subtrair 6?)
  - Ler Retornar a referência em que este está
- Se o cluste index estiver no cluster de referências indiretas (i_2)
  - Calcular dois novos indexes:
    * index no cluster de referências indiretas
    * index no cluster de referências diretas
  - Ler a referência do disco
  - Retornar o valor
- A testtool já trata de fazer o iOpen
- A soGetFileCluster é chamada com o indice do inode
- É preciso usar a `iGetPointer` para obter o ponteiro para a estrutura

Testes

+==========+ | testing functions | +==========
| q - exit | sb - show block | | fd - format disk | spd - set probe depths | +————————–+————————-+ | ai - alloc inode | fi - free inode | | ac - alloc cluster | fc - free cluster | | r - replenish | d - deplete | +————————–+————————-+ | gfc - get file cluster | afc - alloc file cluster | | ffc - free file clusters | - NOT USED | | rfc - read file cluster | wfc - write file cluster | +————————–+————————-+ | gde - get dir entry | ade - add dir entry | | rde - rename dir entry | dde - delete dir entry | | tp - traverse path | - NOT USED | +————————–+————————-+ + cia - check inode access | sia - set inode access + + iil - increment inode lnkcnt | dil - decrement inode lnkcnt + +==========+

Your command: gfc Inode number: 1 File cluster index: 7 (711)–> iOpen(1) (711)–> –iOpenBin(1) (403)–> soGetFileCluster(0, 7) (403)–> –soGetFileClusterBin(0, 7) (712)–> iGetPointer(0) (712)–> –iGetPointerBin(0) (714)–> iClose(0) (714)–> –iCloseBin(0) Cluster number (nil) retrieved +==========+ | testing functions | +==========+ | q - exit | sb - show block | | fd - format disk | spd - set probe depths | +————————–+————————-+ | ai - alloc inode | fi - free inode | | ac - alloc cluster | fc - free cluster | | r - replenish | d - deplete | +————————–+————————-+ | gfc - get file cluster | afc - alloc file cluster | | ffc - free file clusters | - NOT USED | | rfc - read file cluster | wfc - write file cluster | +————————–+————————-+ | gde - get dir entry | ade - add dir entry | | rde - rename dir entry | dde - delete dir entry | | tp - traverse path | - NOT USED | +————————–+————————-+ + cia - check inode access | sia - set inode access + + iil - increment inode lnkcnt | dil - decrement inode lnkcnt + +==========+

Your command: gfc Inode number: 1 File cluster index: 8 (711)–> iOpen(1) (711)–> –iOpenBin(1) (403)–> soGetFileCluster(0, 8) (403)–> –soGetFileClusterBin(0, 8) (712)–> iGetPointer(0) (712)–> –iGetPointerBin(0) (714)–> iClose(0) (714)–> –iCloseBin(0) Cluster number (nil) retrieved +===+ | testing functions | +===+ | q - exit | sb - show block | | fd - format disk | spd - set probe depths | +———————————–+———————————-+ | ai - alloc inode | fi - free inode | | ac - alloc cluster | fc - free cluster | | r - replenish | d - deplete | +———————————–+———————————-+ | gfc - get file cluster | afc - alloc file cluster | | ffc - free file clusters | - NOT USED | | rfc - read file cluster | wfc - write file cluster | +———————————–+———————————-+ | gde - get dir entry | ade - add dir entry | | rde - rename dir entry | dde - delete dir entry | | tp - traverse path | - NOT USED | +———————————–+———————————-+ + cia - check inode access | sia - set inode access + + iil - increment inode lnkcnt | dil - decrement inode lnkcnt + +===+

Your command: gfc Inode number: 0 File cluster index: 1 (711)–> iOpen(0) (711)–> –iOpenBin(0) (851)–> sbGetPointer() (851)–> –sbGetPointerBin() (951)–> soReadRawBlock(1, 0x7fffd273b450) (403)–> soGetFileCluster(1, 1) (403)–> –soGetFileClusterBin(1, 1) (712)–> iGetPointer(1) (712)–> –iGetPointerBin(1) (714)–> iClose(1) (714)–> –iCloseBin(1) Cluster number (nil) retrieved +===+ | testing functions | +===+ | q - exit | sb - show block | | fd - format disk | spd - set probe depths | +———————————–+———————————-+ | ai - alloc inode | fi - free inode | | ac - alloc cluster | fc - free cluster | | r - replenish | d - deplete | +———————————–+———————————-+ | gfc - get file cluster | afc - alloc file cluster | | ffc - free file clusters | - NOT USED | | rfc - read file cluster | wfc - write file cluster | +———————————–+———————————-+ | gde - get dir entry | ade - add dir entry | | rde - rename dir entry | dde - delete dir entry | | tp - traverse path | - NOT USED | +———————————–+———————————-+ + cia - check inode access | sia - set inode access + + iil - increment inode lnkcnt | dil - decrement inode lnkcnt + +===+

21.4 Your command:

21.4.1 void soReadFileCluster (int ih,

```
                        uint32_t  fcn,
                        void *    buf
                      )
```

Read a file cluster.

Data is read from a specific data cluster which is supposed to belong to an inode associated to a file (a regular file, a directory or a symbolic link).

If the referred file cluster has not been allocated yet, the returned data will consist of a byte stream filled with the character null (ascii code 0).

Parameters

```
      ih  inode handler
      fcn file cluster number
      buf pointer to the buffer where data must be read into
```

void soWriteFileCluster (int ih, uint32_t fcn, void * buf)

Write a data cluster.

Data is written into a specific data cluster which is supposed to belong to an inode associated to a file (a regular file, a directory or a symbolic link).

If the referred cluster has not been allocated yet, it will be allocated now so that the data can be stored as its contents.

Parameters

```
ih  inode handler
fcn file cluster number
buf pointer to the buffer containing data to be written
```

21.5 soFreeFileCLusters

- Liberta todos os clusters do inode começando na posição atual
- Se o inode ficar sem clusters, é apagado # direntries

22 soGetDirEntry

- Obtem o inode associado ao nome da função
- É preciso fazer o parse do nome do diretório para chegar ao diretório pretendido
- Chama a traverse Path
- Tem de verificar se a entrada já existe

uint32_t soGetDirEntry (int pih, const char * name)

Get the inode associated to the given name.

The directory contents, seen as an array of directory entries, is parsed to find an entry whose name is name.

The name must also be a base name and not a path, that is, it can not contain the character "/".

Parameters

```
pih  inode handler of the parent directory
name the name entry to be searched for
```

Returns the corresponding inode number

23 soRenameDirEntry

- Renomeia a entrada de um diretório

void soRenameDirEntry (int pih, const char * name, const char * newName)

Rename an entry of a directory.

A direntry associated from the given directory is renamed.

Parameters

```
pih     inode handler of the parent inode
name    current name of the entry
newName new name for the entry
```

24 soTraversePath

- Obtem o inode associado com um dado caminho
- Atravessa a estrutura do sistema do sistema de ficheiros para obter o inode cujo nome do ficheiro é a componente mais à direita do caminho
- O caminho deve ser absoluto
- Todos elementos do caminho (com exceção do último) devem ser diretório ou symbolic links com permissão de travers (x)

uint32_t soTraversePath (char * path)

Get the inode associated to the given path.

The directory hierarchy of the file system is traversed to find an entry whose name is the rightmost component of path. The path is supposed to be absolute and each component of path, with the exception of the rightmost one, should be a directory name or symbolic link name to a path.

The process that calls the operation must have execution (x) permission on all the components of the path with exception of the rightmost one.

Parameters

```
1       path the path to be traversed
```

Returns the corresponding inode number

25 soAddDirEntry

- Adiciona uma nova entrada no diretório pai
- Uma direntry é adicionada ligando o parent inode ao child inode
  - O lnkcnt do inode filho **não** é incrementado nesta função

void soAddDirEntry (int pih, const char * name, uint32_t cin)

Add a new entry to the parent directory.

A direntry is added connecting the parent inode to the child inode. The refcount of the child inode is not incremented by this function.

Parameters

```
1       pih  inode handler of the parent inode
2       name name of the entry
3       cin  number of the child inode
```

26 soDeleteDirEntry

- Remove uma entrada do parent directory
- O lnkcnt do inode filho **não** é decrementado

uint32_t soDeleteDirEntry (int pih, const char * name, bool clean = false)

Remove an entry from a parent directory.

A direntry associated from the given directory is deleted. The refcount of the child inode is not decremented by this function.

Parameters

```
pih   inode handler of the parent inode
name  name of the entry
clean if true (different than zero) clean the corresponding dir entry, otherwise
    keep it dirty
```

Returns the inode number in the deleted entry

27 Extra

filesystem check - atribui ficheiros para a o disco sempre que uma função falahe gera uma exceção

28 soRenameDirEntry

- dá um novo nome ao diretório

29 soDeleteDirEntry

- remove o diretório

30 soGetDirEntry

- quando alguém a nível superior quer abrir/escrever, ver permissões precisa de sabe o inode # itdealer
- Conjunto de funções que manipulam diretamente a estrutura de inodes

30.1 iOpen

- Abre um inode
  - Transfere o seu conteudo para a memória
  - Set ao *usecount*
- Caso o inode já esteja aberto, incrementa a *usecount*
- Em qualquer dos casos devolve o handler (referência para a posição de memória) para o inode # Syscalls

30.2 Main syscalls

- **soLInk:** Cria um link para um ficheiro
- **soMkdir:** Cria um diretório
- **soMknod:** Cria um ficheiro regular com tamanho nulo
- **soRead:** Lê os dados de um ficheiro regular previamente aberto
- **soReaddir:** Lê uma entrada para um diretório de um dado diretório
- **soReadLink:** Lê um *symbolic link*
- **soRename:** Muda um nome de um ficheiro ou a sua localização na estrutura de diretórios
- **soRmdir:** Remover um diretório
  - O diretório de ve estar vazio
- **soSymlink:** Cria um symbolic link com o caminho desejado
- **soTruncate:** Trunca o tamanho de um regular file para o desejado
- **soUnlink:** Remove um link para um ficheiro através de um diretório
  - Remove também o ficheiro se o lnkcnt = 0
- **soWrite:** Escreve dados num regular file previamente aberto

30.3 Other syscalls

- **exceptions** : sofs17 exception definition module "'cpp struct SOException:public std::exception int en; ///< (system) error number const char *msg; ///< name of function that has thrown the exception

digraph x{ a -> b [label="b"] a -> c [label="a"] a -> d [label="d"] } # Existem duas camadas de syscalls - main syscalls - 12 funções (temos de saber para o mini teste) - other syscalss

30.4 soLink

- Usar a transverse path para saber qual o nó que está na ponta
- link("/b", "a/c")
  - saber qual é o nó que está na pomnta do /b
    * uso o traverse para saber
    * abro e pergunto para saber o tipo
    * base name
    * verifico se é o diretório e tenho permissoes de excrita
    * verifico se ja tem o ficheirp que quero criar
    * chamar idirentry para criar o diretorio
    * chamar increment link count

30.5 unLink

- Não apaga o ficheiro
- Quebra a ligaão
- dde - delete dir entry
- decrementa o link count

- se o tiver 0 links
  - chama a free inode para libertar o inode
  - chama a free cluster para libertar o cluster
- APgar ficheiros é derivado do unlink

Enquanto o ficheiro estive aberto não pode ser destruido É o close do sistema operativo que apaga um fucheiro

30.6 soRename

- função complicada
- soRename(“/b”, “/a/c”)
- OU é um rename se o novo pathe e o path antigo orem iguais

```
1      soRename(”/b”, ”/c”)
```

- Equivale no caso do move a fazer delete direentry e add direntry
- Não tem o link nem o dec
- O nó de destino passa a ser o mesmo

30.7 soMKnod

- Cria um nod do tipo ficheiro
- COrrepsonde a fazer:
  - Começar por criar um inode: alloc indode,
  - add dir entry
  - increment link count
- Tem de validar primeiro:
  - Verificar se o “/a” existe, é um diretório e tem permisssoes de escrita
  - Verificar se o “/c” não existe

30.8 soRead

- Posso quer ler dois bytes e ter de ler dois clusters
  - Ultimo byte do 1º cluster
  - Primeiro byte do 2º cluster
- A função read não pode ler para a lém do fim de ficheiro
- O size é que determina o fim do ficheiroo
- Indiretamento o write també, pode alocar clusters
- Tipicamente o wrtie tem de ler priemiro caos vá alterar parcialemtne um cluster
- O que interessa em termos de miniteste é o papel e efeito da função, não como o ocódigo é feito
- O que interessa é a consequemncia da execuação de u comando

30.9 soTrucnate

- Alçtera o tamanho de um gicheiro ou para cima ou para baixo
- È assim que se cria buracos num ficheiro
  - Trunco e vou acrescentando
- Gunção joga com o size e
- Trunco o tamanho para 10
- Depois volto a troncar para 20
  - Ou no truncar para cima ou no truncar para baixo tenho de garantir que nºao existe lixo entre as zonas dos meus dados
    * Ou escrevo zeros., ou escrevo NullReferences
      · Null References dentro do size são lidas como zeros
  - O ficheiro é o mesmo, estou só a alterar o tamanho que esse ficheiro tem no disco
  - O truncate não quer saber o que lá está, simplesmente trunca os dados inteirores
  - Ou eu ponho zeros quando encolhi, ou ponho zeros quando abro
- Se fize ro fopne de um ficheiro já abero, o SO chama a truncate e mete os dados desse ficheiro a zero

30.10 soMkdir

- Sempre ue há u novo direentry é preciso adicionar o linkcount
- Pressuposto: só se pode aoagar diretórios vazios
- Ler man2 para saber os erros que tem de emitir

30.11 soReadDIr

- É usado para fazer o ls
- Leê entradas de um diretorio
- Sempre que esta funlção é lida, ele vai ler a próxima direntry
- Em cadainvocaçáo eu tenho que lhe dzwr qenaotos bytes já passei
- Posso ter de processar duas entradas para lhe dar uma . QUando a fubnção rreaddir devolve zero, já não existem mais entradas naquele deiretorio

30.12 soSymlink

- Creates a symbolic link

30.13 so ReadLink

- Valor de retorno do symbolic link # HOW to use sofs17 (so1718 - Aula prática 29 Sep) ## Documentação

```
# Gerar documentação
cd ./doc
doxygen
```

A documentação fica na pasta `./doc/html/`

31 Make

```
# O make compila sempre tudo e não somente o conteudo da pasta
make
make -C <path_to_start>     % indica o caminho onde o make começar
```

Na linkagem necessita da biblioteca fuse.h. Está contida na biblioteca libfuse-dev que pode ser instalada com:

```
sudo apt-get install libfuse-dev
```

[TODO] mksofs - msksofs : formatador para o sistema de ficheiros sofs17

Para já as funções

- compute structure:
  - nao altera os dados no disco.
  - Apenas calcula os blocos de inodes, clusters, etc.
- cada função vai preencher a àrea do disco respetiva
  - **fillInSuperBlock** : computes the structural division of the disk
  - **fillInInodeTable** :

32 soFreeFileCLusters

- Apagamento da esquerda para a direita
- As posições são alteradas e escritas com Null Reference
- o size só é alterado por syscalls e não ao libertar um inode/cluster

33

- Um diretório tem um size múltiplo do cluster
- Os diretórios crescem cluster a cluster

34

uint32_t soGetDirEntry (int pih, const char * name)

Get the inode associated to the given name.

The directory contents, seen as an array of directory entries, is parsed to find an entry whose name is name.

The name must also be a base name and not a path, that is, it can not contain the character “/”.

Parameters

```
      pih  inode handler of the parent directory
      name the name entry to be searched for
```

Returns the corresponding inode number

35

uint32_t soDeleteDirEntry (int pih, const char * name, bool clean = false)

Remove an entry from a parent directory.

A direntry associated from the given directory is deleted. The refcount of the child inode is not decremented by this function.

Parameters

```
pih   inode handler of the parent inode
name  name of the entry
clean if true (different than zero) clean the corresponding dir entry, otherwise
    keep it dirty
```

Returns the inode number in the deleted entry

Comentários - soWriteRawBlock - char blk[blockSize] - SOSuperblock sb; - SOTnode it[inodesPerBlock] - soWriteRawBlock(uint32_t n, void *buf) - blk - fsb - it

35.1 Notes

função alloc cluster tem de verificar se ficou tudo bem no disco função replentish transfer da reference bitmao block para a retrieval cache

Capacidade: $(6 + 2^9 + (2^9)^2) \cdot 2^1 1$

36 3 Nov 2017

- As direntries não mexem no lnkcnt
- Add mexe no size

37 Unlink

1. dde
2. dec
3. if(dec ==0) 3.1 ffc 3.2 fi

O dec devolve o devolve

38 Remove

39 mtime vs ctime

- **mtime:** conteudo do ficheiro
- **ctime:** metadados do ficheiro

- Não podemos ter nenhum diretório apontado por dois diretórios

```
ln: 'ddd/': hard link not allowwd for directory
```

40 Conceitos Introdutórios

Num ambiente multiprogramado, os processos podem ser:

- Independentes:
  - Nunca interagem desde a sua criação à sua destruição
  - Só possuem uma interação implícita: **competir por recursos do sistema**
    - e.g.: jobs num sistema bacth, processos de diferentes utilizadores
  - É da responsabilidade do sistema operativo garantir que a atribuição de recursos é feita de forma controlada
    - É preciso garantir que não ocorre perda de informação
    - Só **um processo pode usar um recurso num intervalo de tempo** - ***Mutual Exclusive Acess***
- Cooperativos:
  - **Partilham Informação** e/ou **Comunicam** entre si
  - Para **partilharem** informação precisam de ter acesso a um **espaço de endereçamento comum**
  - A comunicação entre processos pode ser feita através de:
    - Endereço de memória comum
    - Canal de comunicação que liga os processos
  - É da **responsabilidade do processo** garantir que o acesso à zona de memória partilhada ou ao canal de comunicação é feito de forma controlada para não ocorrerem perdas de informação
    - Só **um processo pode usar um recurso num intervalo de tempo** - ***Mutual Exclusive Acess***
    - Tipicamente, o canal de comunicação é um recurso do sistema, pelo quais os **processos competem**

O acesso a um recurso/área partilhada é efetuada através de código. Para evitar a perda de informação, o código de acesso (também denominado zona crítica) deve evitar incorrer em **race conditions**.

40.1 Exclusão Mútua

Ao forçar a ocorrência de exclusão mútua no acesso a um recurso/àrea partilhada, podemos originar:

- **deadlock:**
  - Vários processos estão em espera **eternamente** pelas condições/eventos que lhe permitem aceder à sua respetiva **zona crítica**
    - Pode ser provado que estas condições/eventos **nunca se irão verificar**
  - Causa o bloqueio da execução das operações
- **starvation:**
  - Na competição por acesso a uma zona crítica por vários processos, verificam-se um conjunto de circunstâncias na qual novos processos, com maior prioridade no acesso às suas zonas críticas, continuam a aparecer e **tomar posse dos recursos partilhados**
  - O acesso dos processos mais antigos à sua zona crítica é sucessivamente adiado

41 Acesso a um Recurso

No acesso a um recurso é preciso garantir que não ocorrem **race conditions**. Para isso, **antes** do acesso ao recurso propriamente dito é preciso **desativar o acesso** a esse recurso pelos **outros processos** (reclamar *ownership*) e após o acesso é preciso restaurar as condições iniciais, ou seja, **libertar o acesso** ao recurso.

```c
/* processes competing for a resource - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    forever
    {
        do_something();
        access_resource(p);
        do_something_else();
    }
}

void acess_resource(unsigned int p)
{
    enter_critical_section(p);
    use_resource();     //  critical section
    leave_critical_section(p);
}
```

42 Acesso a Memória Partilhada

O acesso à memória partilhada é muito semelhante ao aceso a um recurso (podemos ver a memória partilhada como um recurso partilhado entre vários processos).

Assim, à semelhança do acesso a um recurso, é preciso **bloquear o acesso de outros processos à memória partilhada** antes de aceder ao recurso e após aceder, **re-ativar o acesso a memória partilhada** pelos outros processos.

```c
/* shared data structure */
shared DATA d;

/* processes sharing data - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    forever
    {
        do_something();
        access_shared_area(p);
        do_something_else();
    }
}

void access_shared_area(unsigned int p)
{
    enter_critical_section(p);
    manipulate_shared_area(); // critical section
    leave_critical_section(p);
}
```

42.1 Relação Produtor-Consumidor

O acesso a um recurso/memória partilhada pode ser visto como um problema Produtor-Consumidor:

- Um processo acede para **armazenar dados**, **escrevendo** na memória partilhada *(Produtor)*
- Outro processo acede para **obter dados**, **lendo** da memória partilhada *(Consumidor)*

42.1.1 Produtor

O produtor "produz informação" que quer guardar na FIFO e enquanto não puder efetuar a sua escrita, aguarda até puder **bloquear** e **tomar posse** do zona de memória partilhada

```
/* communicating data structure: FIFO of fixed size */
shared FIFO fifo;

/* producer processes - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    DATA val;
    bool done;


    forever
    {
        produce_data(&val);
        done = false;
        do
        {
            // Beginning of Critical Section
            enter_critical_section(p);
            if (fifo.notFull())
            {
                fifo.insert(val);
                done = true;
            }
            leave_critical_section(p);
            // End of Critical Section
        } while (!done);
        do_something_else();
    }
}
```

42.1.2 Consumidor

O consumidor quer ler informação que precisa de obter da FIFO e enquanto não puder efetuar a sua leitura, aguarda até puder **bloquear** e **tomar posse** do zona de memória partilhada

```
/* communicating data structure: FIFO of fixed size */
shared FIFO fifo;

```

```
/* consumer processes - p = 0, 1, ..., M-1 */
void main (unsigned int p)
{
DATA val;
bool done;
    forever
    {
        done = false;
        do
        {
            // Beginning of Critical Section
            enter_critical_section(p);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&val);
                done = true;
            }
            leave_critical_section(p);
            // End of Critical Section
        } while (!done);
        consume_data(val);
        do_something_else();
    }
}
```

43 Acesso a uma Zona Crítica

Ao aceder a uma zona crítica devem ser verificados as seguintes condições:

- **Efective Mutual Exclusion:** O **acesso** a uma **zona crítica** associada com o mesmo recurso/memória partilhada só pode ser **permitida a um processo de cada vez** entre **todos os processos** a competir pelo acesso a esse mesmo recurso/memória partilhada
- **Independência** do número de processos intervenientes e na sua velocidade relativa de execução
- Um processo fora da sua zona crítica não pode impedir outro processo de entrar na sua zona crítica
- Um processo **não deve ter de esperar indefinidamente** após pedir acesso ao recurso/memória partilhada para que possa aceder à sua zona crítica
- O período de tempo que um processo está na sua **zona crítica** deve ser **finito**

43.1 Tipos de Soluções

Para controlar o acesso às zonas críticas normalmente é usado um endereço de memória. A gestão pode ser efetuada por:

- **Software:**
  - A solução é baseada nas instruções típicas de acesso à memória
  - Leitura e Escrita são indepentes e correspondem a instruções diferentes
- **Hardware:**
  - A solução é baseada num conjunto de instruções especiais de acesso à memória
  - Estas instruções permitem ler e de seguida escrever na memória, de forma **atómica**

43.2 Alternância Estrita *(Strict Alternation)*

Não é uma solução válida

- Depende da velocidade relativa de execução dos processos intervenientes
- O processo com menos acessos impõe o ritmo de acessos aos restantes processos
- Um processo fora da zona crítica não pode prevenir outro processo de entrar na sua zona crítica
- Se não for o seu turno, um processo é obrigado a esperar, mesmo que não exista mais nenhum processo a pedir acesso ao recurso/memória partilhada

```
/* control data structure */
#define R       /* process id = 0, 1, ..., R-1 */

shared unsigned int access_turn = 0;
void enter_critical_section(unsigned int own_pid)
{
    while (own_pid != access_turn);
}

void leave_critical_section(unsigned int own_pid)
{
    if (own_pid == access_turn)
        access_turn = (access_turn + 1) % R;
}
```

43.3 Eliminar a Alternância Estrita

```
/* control data structure */
#define R 2     /* process id = 0, 1 */

shared bool is_in[R] = {false, false};

void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid_ = 1 - own_pid;

    while (is_in[other_pid]);
    is_in[own_pid] = true;
}

void leave_critical_section(unsigned int own_pid)
{
    is_in[own_pid] = false;
}
```

Esta solução não é válida porque não garante **exclusão mútua**.

Assume que:

- P_0 entra na função `enters_critical_section` e testa `is_in[1]`, que retorna Falso

- P_1 entra na função `enter_critical_section` e testa `is_in[0]`, que retorna Falso
- P_1 altera `is_in[0]` para *true* e entra na zona crítica
- P_0 altera `is_in[1]` para *true* e entra na zona crítica

Assim, ambos os processos entra na sua zona crítica **no mesmo intervalo de tempo**.

O principal problema desta implementação advém de **testar primeiro** a variável de controlo do **outro processo** e só **depois** alterar a **sua variável** de controlo.

43.4 Garantir a exclusão mútua

```
1  /* control data structure */
2  #define R 2     /* process id = 0, 1 */
3
4  shared bool want_enter[R] = {false, false};
5
6  void enter_critical_section(unsigned int own_pid)
7  {
8      unsigned int other_pid_ = 1 - own_pid;
9
10     want_enter[own_pid] = true;
11     while (want_enter[other_pid]);
12 }
13
14 void leave_critical_section(unsigned int own_pid)
15 {
16     want_enter[own_pid] = false;
17 }
```

Esta solução, apesar de **resolver a exclusão mútua**, **não é válida** porque podem ocorrer situações de **deadlock**.

Assume que:

- P_0 entra na função `enter_critical_section` e efetua o *set* de `want_enter[0]`
- P_1 entra na função `enter_critical_section` e efetua o *set* de `want_enter[1]`
- P_1 testa `want_enter[0]` e, como é *true*, **fica em espera** para entrar na zona crítica
- P_0 testa `want_enter[1]` e, como é *true*, **fica em espera** para entrar na zona crítica

Com **ambos os processos em espera** para entrar na zona crítica e **nenhum processo na zona crítica** entramos numa situação de **deadlock**.

Para resolver a situação de deadlock, **pelo menos um dos processos** tem recuar na intenção de aceder à zona crítica.

43.5 Garantir que não ocorre deadlock

```
1  /* control data structure */
2  #define R 2     /* process id = 0, 1 */
3
4  shared bool want_enter[R] = {false, false};
5
6  void enter_critical_section(unsigned int own_pid)
```

```
{
    unsigned int other_pid_ = 1 - own_pid;

    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        want_enter[own_pid] = false;    // go back
        random_dealy();
        want_enter[own_pid] = true;     // attempt a to go to the critical section
    }
}

void leave_critical_section(unsigned int own_pid)
{
    want_enter[own_pid] = false;
}
```

A solução é quase válida. Mesmo um dos processos a recuar ainda é possível ocorrerem situações de **deadlock** e **starvation**:

- Se ambos os processos **recuarem ao "mesmo tempo"** (devido ao `random_delay()` ser igual), entramos numa situação de **starvation**
- Se ambos os processos **avançarem ao "mesmo tempo"** (devido ao `random_delay()` ser igual), entramos numa situação de **deadlock**

A solução para **mediar os acessos** tem de ser **determinística** e não aleatória.

43.6 Mediar os acessos de forma determinística: *Dekker agorithm*

```
/* control data structure */
#define R 2     /* process id = 0, 1 */

shared bool want_enter[R] = {false, false};
shared uint p_w_priority = 0;

void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid_ = 1 - own_pid;

    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        if (own_pid != p_w_priority)            // If the process is not the prioritary
            process
        {
            want_enter[own_pid] = false;        // go back
            while (own_pid != p_w_priority);    // waits to acess to his critical section
                while
                                                //  its is not the prioritary process
            want_enter[own_pid] = true;         // attempt to go to his critical section
        }
    }
```

```
}

void leave_critical_section(unsigned int own_pid)
{
    unsigned int other_pid_ = 1 - own_pid;
    p_w_priority = other_pid;                   // when leaving the its critical section, assign the
                                                // priority to the other process
    want_enter[own_pid] = false;
}
```

É uma **solução válida**:

- Garante exclusão mútua no acesso à zona crítica através de um mecanismo de alternância para resolver o conflito de acessos
- **deadlock** e **starvation não estão presentes**
- Não são feitas suposições relativas ao tempo de execução dos processos, i.e., o algoritmo é **independente** do tempo de execução dos processos

No entanto, **não pode ser generalizado** para mais do que 2 processos e garantir que continuam a ser satisfeitas as condições de **exclusão mútua** e a ausência de **deadlock** e **starvation**

43.7 Dijkstra algorithm (1966)

```
/* control data structure */
#define R 2     /* process id = 0, 1 */

shared uint want_enter[R] = {NO, NO, ..., NO};
shared uint p_w_priority = 0;

void enter_critical_section(uint own_pid)
{
    uint n;
    do
    {
        want_enter[own_pid] = WANT;                 // attempt to access to the critical section
        while (own_pid != p_w_priority)             // While the process is not the prioritary process
        {
            if (want_enter[p_w_priority] == NO)     // Wait for the priority process to leave its critical section
                p_w_priority = own_pid;
        }

        want_enter[own_pid] = DECIDED;              // Mark as the next process to access to its critical section

        for (n = 0; n < R; n++)             // Search if another process is already entering its critical section
        {
```

```
23              if (n != own_pid && want_enter[n] == DECIDED)   // If so, abort attempt to
                    ensure mutual exclusion
24                  break;
25          }
26      } while(n < R);
27  }
28
29  void leave_critical_section(unsigned int own_pid)
30  {
31      p_w_priority = (own_pid + 1) % R;           // when leaving the its critical section,
            assign the
32                                                  // priority to the next process
33      want_enter[own_pid] = false;
34  }
```

Pode sofrer de **starvation** se quando um processo iniciar a saída da zona crítica e alterar `p_w_priority`, atribuindo a prioridade a outro processo, outro processo tentar aceder à zona crítica, sendo a sua execução interompida no for. Em situações "especiais", este fenómeno pode ocorrer sempre para o mesmo processo, o que faz com que ele nunca entre na sua zona crítica

43.8 Peterson Algorithm (1981)

```
1   /* control data structure */
2   #define R 2     /* process id = 0, 1 */
3
4   shared bool want_enter[R] = {false, false};
5   shared uint last;
6
7   void enter_critical_section(uint own_pid)
8   {
9       unsigned int other_pid_ = 1 - own_pid;
10
11      want_enter[own_pid] = true;
12      last = own_pid;
13      while ( (want_enter[other_pid]) && (last == own_pid) );     // Only enters the
            critical section when no other
14                                                                  // process wants to enter
                                                                        and the last request
15                                                                  // to enter is made by the
                                                                        current process
16  }
17
18  void leave_critical_section(unsigned int own_pid)
19  {
20      want_enter[own_pid] = false;
21  }
```

O algoritmo de *Peterson* usa a **ordem de chegada** de pedidos para resolver conflitos:

- Cada processo tem de **escrever o seu ID numa variável partilhada** (*last*), que indica qual foi o último processo a pedir para entrar na zona crítica

- A **leitura seguinte** é que vai determinar qual é o processo que foi o último a escrever e portanto qual o processo que deve entrar na zona crítica

	P_0 quer entrar		P_1 quer entrar	
	P_1 não quer entrar	P_1 quer entrar	P_0 não quer entrar	P_0 quer entrar
last = P_0	P_0 entra	P_1 entra	-	P_1 entra
last = P_1	-	P_0 entra	P_1 entra	P_0 entra

É uma solução válida que:

- Garante exclusão mútua
- Previne deadlock e starvation
- É independente da velocidade relativa dos processos
- Pode ser generalizada para mais do que dois processos (variável partilhada -> fila de espera)

43.9 Generalized Peterson Algorithm (1981)

```
/* control data structure */
#define R ...   /* process id = 0, 1, ..., R-1 */

shared bool want_enter[R] = {-1, -1, ..., -1};
shared uint last[R-1];

void enter_critical_section(uint own_pid)
{
    for (uint i = 0; i < R -1; i++)
    {
        want_enter[own_pid] = i;

        last[i] = own_pid;

        do
        {
            test = false;
            for (uint j = 0; j < R; j++)
            {
                if (j != own_pid)
                    test = test || (want_enter[j] >= i)
            }
        } while ( test && (last[i] == own_pid) );       // Only enters the critical
            section when no other
                                                                // process wants to enter
                                                                    and the last request
                                                                // to enter is made by the
                                                                    current process
    }
}

void leave_critical_section(unsigned int own_pid)
```

```
30 {
31     want_enter[own_pid] = -1;
32 }
```

> *needs clarification*

44 Soluções de Hardware

44.1 Desativar as interrupções

Num ambiente computacional com **um único processador:**

- A alternância entre processos, num ambiente **multiprogramado**, é sempre causada por um evento/dispositivo externo
  - **real time clock (RTC):** origina a transição de time-out em sistemas *preemptive*
  - **device controller:** pode causar transições *preemptive* no caso de um fenómeno de *wake up* de um **processo mais prioritário**
  - Em qualquer dos casos, o **processador é interrompido** e a execução do processo atual parada
- A garantia de acesso em **exclusão mútua** pode ser feita desativando as interrupções
- No entanto, só pode ser efetuada em **modo kernel**
  - Senão código malicioso ou com *bugs* poderia bloquear completamente o sistema

Num ambiente computacional **multiprocessador**, desativar as interrupções num único processador não tem qualquer efeito.

Todos os outro processadores (ou *cores*) continuam a responder às interrupções.

44.2 Instruções Especiais em Hardware

44.2.1 Test and Set (TAS primitive)

A função de hardware, `test_and_set` se for implementada atomicamente (i.e., sem interrupções) pode ser utilizada para construir a primitiva **lock**, que permite a entrada na zona crítica

Usando esta primitiva, é possível criar a função *lock*, que permite entrar na zona crítica

```
1  shared bool flag = false;
2
3  bool test_and_set(bool * flag)
4  {
5      bool prev = *flag;
6      *flag = true;
7      return prev;
8  }
9
10 void lock(bool * flag)
11 {
12     while (test_and_set(flag);  // Stays locked until and unlock operation is used
13 }
```

```
14
15 void unlock(bool * flag)
16 {
17     *flag = false;
18 }
```

44.2.2 Compare and Swap

Se implementada de forma atómica, a função `compare_and_set` pode ser usada para implementar a primitiva lock, que permite a entrada na zona crítica

O comportamento esperado é que coloque a variável a 1 sabendo que estava a 0 quando a função foi chamada e vice-versa.

```
1  shared int value = 0;
2
3  int compare_and_swap(int * value, int expected, int new_value)
4  {
5      int v = *value;
6      if (*value == expected)
7          *value = new_value;
8      return v;
9  }
10
11 void lock(int * flag)
12 {
13     while (compare_and_swap(&flag, 0, 1) != 0);
14 }
15
16 void unlock(bool * flag)
17 {
18     *flag = 0;
19 }
```

44.3 Busy Waiting

Ambas as funções anteriores são suportadas nos *Instruction Sets* de alguns processadores, implementadas de forma atómica

No entanto, ambas as soluções anteriores sofrem de **busy waiting**. A primitva lock está no seu **estado ON** (usando o CPU) **enquanto espera** que se verifique a condição de acesso à zona crítica. Este tipo de soluções são conhecidas como **spinlocks**, porque o processo oscila em torno da variável enquanto espera pelo acesso

Em sistemas **uniprocessador**, o **busy_waiting** é **indesejado** porque causa:

- **Perda de eficiência:** O **time quantum** de um processo está a ser desperdiçado porque não está a ser usado para nada
- ** Risco de deadlock**: Se um** processo mais prioritário** tenciona efetuar um **lock** enquanto um processo menos prioritário está na sua zona crítica, **nenhum deles pode prosseguir**.
  - O processo menos prioritário tenta executar um unlock, mas não consegue ganhar acesso a um *time qantum* do CPU devido ao processo mais prioritário
  - O processo mais prioritário não consegue entrar na sua zona crítica porque o processo menos prioritário ainda não saiu da sua zona crítica

Em sistemas **multiprocessador** com **memória partilhada**, situações de busy waiting podem ser menos críticas, uma vez que a troca de processos *(preempt)* tem custos temporais associados. É preciso:

- guardar o estado do processo atual
  - variáveis
  - stack
  - $PC
- copiar para memória o código do novo processo

44.4 Block and wake-up

Em **sistemas uniprocessor** (e em geral nos restantes sistemas), existe a o requerimento de **bloquear um processo** enquanto este está à espera para entrar na sua zona crítica

A implementação das funções `enter_critical_section` e `leave_critical_section` continua a precisar de operações atómicas.

```
#define  R ... /* process id = 0, 1, ..., R-1 */

shared unsigned int access = 1;     // Note that access is an integer, not a boolena

void enter_critical_section(unsigned int own_pid)
{
    // Beginning of atomic operation
    if (access == 0)
        block(own_pid);

    else access -= 1;
    // Ending of atomic operation
}

void leave_critical_section(unsigned int own_pid)
{
    // Beginning of atomic operation
    if (there_are_blocked_processes)
        wake_up_one();
     else access += 1;
    // Ending of atomic operation
}
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
```

```
            lock(p);
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        unlock(p);
    } while (!done);
    do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            lock(c);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
            unlock(c);
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

45 Semáforos

No ficheiro `IPC.md` são indicadas as condições e informação base para:

- Sincronizar a entrada na zona crítica
- Para serem usadas em programação concurrente
- Criar zonas que garantam a exclusão mútua

Semáforos são **mecanismos** que permitem por implementar estas condições e **sincronizar a atividade** de **entidades concurrentes em ambiente multiprogramado**

Não são nada mais do que **mecanismos de sincronização**.

45.1 Implementação

Um semáforo é implementado através de:

- Um tipo/estrutura de dados
- Duas operações **atómicas**:
  - down (ou wait)
  - up (ou signal/post)

```
typedef struct
{
    unsigned int val;   /* can not be negative */
    PROCESS *queue;     /* queue of waiting blocked processes */
} SEMAPHORE;
```

45.1.1 Operações

As únicas operações permitidas são o **incremento**, up, ou **decremento**, down, da variável de controlo. A variável de controlo, `val`, **só pode ser manipulada através destas operações!**

Não existe uma função de leitura nem de escrita para `val`.

- `down`
  - **bloqueia** o processo se `val == 0`
  - **decrementa** `val` se `val != 0`
- `up`
  - Se a `queue` não estiver vazia, **acorda** um dos processos
  - O processo a ser acordado depende da **política implementada**
  - **Incrementa** `val` se a `queue` estiver vazia

45.1.2 Solução típica de sistemas *uniprocessor*

```
/* array of semaphores defined in kernel */
#define R /* semid = 0, 1, ..., R-1 */

static SEMAPHORE sem[R];

void sem_down(unsigned int semid)
{
    disable_interruptions;
    if (sem[semid].val == 0)
        block_on_sem(getpid(), semid);
    else
        sem[semid].val -= 1;
    enable_interruptions;
}

void sem_up(unsigned int semid)
{
    disable_interruptions;
    if (sem[sem_id].queue != NULL)
        wake_up_one_on_sem(semid);
```

```
21      else
22          sem[semid].val += 1;
23      enable_interruptions;
24  }
```

A solução apresentada é típica de um sistema *uniprocessor* porque recorre à diretivas **disable_interrutions** e **enable_interruptions** para garantir a exclusão mútua no acesso à zona crítica.

Só é possível garantir a exclusão mútua nestas condições se o sistema só possuir um único processador, poruqe as diretivas irão impedir a interrupção do processo que está na posse do processador devido a eventos externos. Esta solução não funciona para um sistema multi-processador porque ao executar a diretiva **disable_interrutions**, só estamos a **desativar as interrupções para um único processador**. Nada impede que noutro processador esteja a correr um processo que vá aceder à mesma zona de memória partilhada, não sendo garantida a exclusão mútua para sistemas multi-processador.

Uma solução alternativa seria a extensão do **disable_interruptions** a todos os processadores. No entanto, iriamos estar a impedir a troca de processos noutros processadores do sistema que poderiam nem sequer tentar aceder às variáveis de memória partilhada.

45.2 Bounded Buffer Problem

```
1   shared FIFO fifo; /* fixed-size FIFO memory */
2
3   /* producers - p = 0, 1, ..., N-1 */
4   void producer(unsigned int p)
5   {
6       DATA data;
7       forever
8       {
9           produce_data(&data);
10          bool done = false;
11          do
12          {
13              lock(p);
14              if (fifo.notFull())
15              {
16                  fifo.insert(data);
17                  done = true;
18              }
19              unlock(p);
20          } while (!done);
21      do_something_else();
22      }
23  }
24
25  /* consumers - c = 0, 1, ..., M-1 */
26  void consumer(unsigned int c)
27  {
28      DATA data;
29      forever
30      {
31          bool done = false;
```

```
32          do
33          {
34              lock(c);
35              if (fifo.notEmpty())
36              {
37                  fifo.retrieve(&data);
38                  done = true;
39              }
40              unlock(c);
41          } while (!done);
42          consume_data(data);
43          do_something_else();
44      }
45  }
```

45.2.1 Como Implementar usando semáforos?

A solução para o *Bounded-buffer Problem* usando semáforos tem de:

- Garantir **exclusão mútua**
- Ausência de busy waiting

```
1   shared FIFO fifo;   /*fixed-size FIFO memory */
2   shared sem access;  /*semaphore to control mutual exclusion */
3   shared sem nslots;  /*semaphore to control number of available slots*/
4   shared sem nitems;  /*semaphore to control number of available items */
5
6
7   /* producers - p = 0, 1, ..., N-1 */
8   void producer(unsigned int p)
9   {
10      DATA val;
11
12      forever
13      {
14          produce_data(&val);
15          sem_down(nslots);
16          sem_down(access);
17          fifo.insert(val);
18          sem_up(access);
19          sem_up(nitems);
20          do_something_else();
21      }
22  }
23
24  /* consumers - c = 0, 1, ..., M-1 */
25  void consumer(unsigned int c)
26  {
27      DATA val;
28
29      forever
```

```
30      {
31          sem_down(nitems);
32          sem_down(access);
33          fifo.retrieve(&val);
34          sem_up(access);
35          sem_up(nslots);
36          consume_data(val);
37          do_something_else();
38      }
39  }
```

Não são necessárias as funções `fifo.empty()` e `fifo.full()` porque são implementadas indiretamente pelas variáveis:

- **nitens:** Número de "produtos" prontos a serem "consumidos"
  - Acaba por implementar, indiretamente, a funcionalidade de verificar se a FIFO está empty
- **nslots:** Número de slots livres no semáforo. Indica quantos mais "produtos" podem ser produzidos pelo "consumidor"
  - Acaba por implementar, indiretamente, a funcionalidade de verificar se a FIFO está full

Uma alternativa **ERRADA** a uma implementação com semáforos é apresentada abaixo:

```
1   shared FIFO fifo;   /*fixed-size FIFO memory */
2   shared sem access;  /*semaphore to control mutual exclusion */
3   shared sem nslots;  /*semaphore to control number of available slots*/
4   shared sem nitems;  /*semaphore to control number of available items */
5
6
7   /* producers - p = 0, 1, ..., N-1 */
8   void producer(unsigned int p)
9   {
10      DATA val;
11
12      forever
13      {
14          produce_data(&val);
15          sem_down(access);       // WRONG SOLUTION! The order of this
16          sem_down(nslots);       // two lines are changed
17          fifo.insert(val);
18          sem_up(access);
19          sem_up(nitems);
20          do_something_else();
21      }
22  }
23
24  /* consumers - c = 0, 1, ..., M-1 */
25  void consumer(unsigned int c)
26  {
27      DATA val;
28
29      forever
30      {
31          sem_down(nitems);
```

```
32          sem_down(access);
33          fifo.retrieve(&val);
34          sem_up(access);
35          sem_up(nslots);
36          consume_data(val);
37          do_something_else();
38      }
39  }
```

A diferença entre esta solução e a anterior está na troca de ordem de instruções `sem_down(access)` e `sem_down(nslots)`. A função `sem_down`, ao contrário das funções anteriores, **decrementa** a variável, não tenta decrementar.

Assim, o produtor tenta aceder à sua zona crítica sem primeiro decrementar o número de slots livres para ele guardar os resultados da sua produção *(needs_clarification)*

45.3 Análise de Semáforos

45.3.1 Vantagens

- **Operam ao nível do sistema operativo:**
  - As operações dos semáforos são implementadas no *kernel*
  - São disponibilizadas aos utilizadores através de *system_calls*
- São **genéricos** e **modulares**
  - por serem implementações de baixo nível, ganham **versatilidade**
  - Podem ser usados em qualquer tipo de situação de programão concurrente

45.3.2 Desvantagens

- Usam **primitivas de baixo nível**, o que implica que o programador necessita de conhecer os **princípios da programação concurrente**, uma vez que são aplicadas numa filosofia *bottom-up* - Facilmente ocorrem **race conditions** - Facilmente se geram situações de **deadlock**, uma vez que **a ordem das operações atómicas são relevantes**
- São tanto usados para implementar **exclusão mútua** como para **sincronizar processos**

45.3.3 Problemas do uso de semáforos

Como tanto usados para implementar **exclusão mútua** como para **sincronizar processos**, se as condições de acesso não forem satisfeitas, os processos são bloqueados **antes** de entrarem nas suas regisões críticas.

- Solução sujeita a erros, especialmente em situações complexas
  - pode existir **mais do que um ponto de sincronismos** ao longo do programa

45.4 Semáforos em Unix/Linux

POSIX:

- Suportam as operações de `down` e `up`

  - `sem_wait`
  - `sem_trywait`
  - `sem_timedwait`
  - `sem_post`

- Dois tipos de semáforos:
  - **named semaphores:**
    * São criados num sistema de ficheiros virtual (e.g. /dev/sem)
    * Suportam as operações:
      · `sem_open`
      · `sem_close`
      · `sem_unlink`
  - **unnamed semaphores:**
    * São *memory based*
    * Suportam as operações
      · `sem_init`
      · `sem_destroy`

System V:

- Suporta as operações:
  - `semget` : criação
  - `semop` : as diretivas `up` e `down`
  - `semctl` : outras operações

46 Monitores

Mecanismo de sincronização de alto nível para resolver os problemas de sincronização entre processos, numa perspetiva **top-down**. Propostos independentemente por Hoare e Brinch Hansen

Seguindo esta filosofia, a **exclusão mútua** e **sincronização** são tratadas **separadamente**, devendo os processos:

1. Entrar na sua zona crítica
2. Bloquear caso nao possuam condições para continuar

Os monitores são uma solução que suporta nativamente a exclusão mútua, onde uma aplicação é vista como um conjunto de *threads* que competem para terem acesso a uma estrutura de dados partilhada, sendo que esta estrutura só pode ser acedida pelos métodos do monitor.

Um monitor assume que todos os seus métodos **têm de ser executados em exclusão mútua**:

- Se uma *thread* chama um **método de acesso** enquanto outra *thread* está a exceutar outro método de acesso, a sua **execução é bloqueada** até a outra terminar a execução do método

A sincronização entre threads é obtida usando **variáveis condicionais**:

- `wait`: A *thread* é bloqueada e colocada fora do monitor
- `signal`: Se existirem outras *threads* bloqueadas, uma é escolhida para ser "acordada"

46.1 Implementação

```
1   monitor example
2   {
3   /* internal shared data structure */
4   DATA data;
5
6   condition c; /* condition variable */
7
8   /* access methods */
9   method_1 (...)
10  {
11      ...
12  }
13  method_2 (...)
14  {
15      ...
16  }
17
18  ...
19
20  /* initialization code */
21  ...
```

46.2 Tipos de Monitores

46.2.1 Hoare Monitor

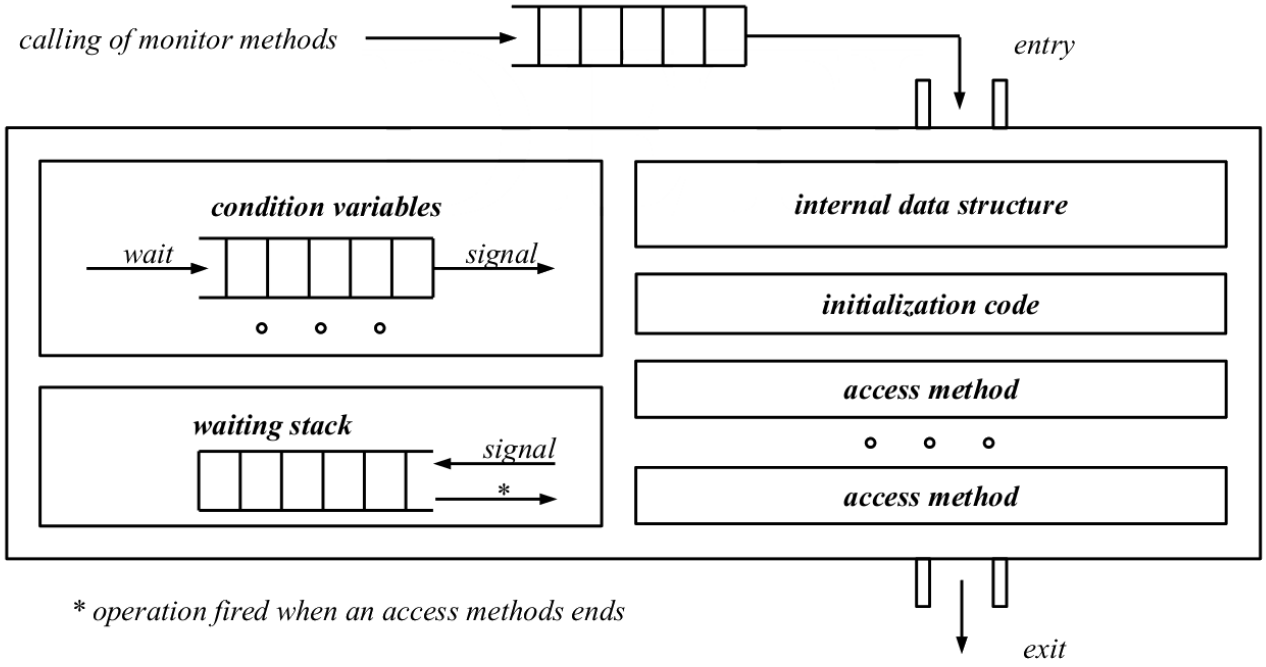

Figure 5: Diagrama da estrutura interna de um Monitor de Hoare

- Monitor de aplicação geral
- Precisa de uma stack para os processos que efetuaram um `wait` e são colocados em espera
- Dentro do monitor só se encontra a *thread* a ser executada por ele
- Quando existe um `signal`, uma *thread* é **acordada** e posta em execução

46.2.2 Brinch Hansen Monitor

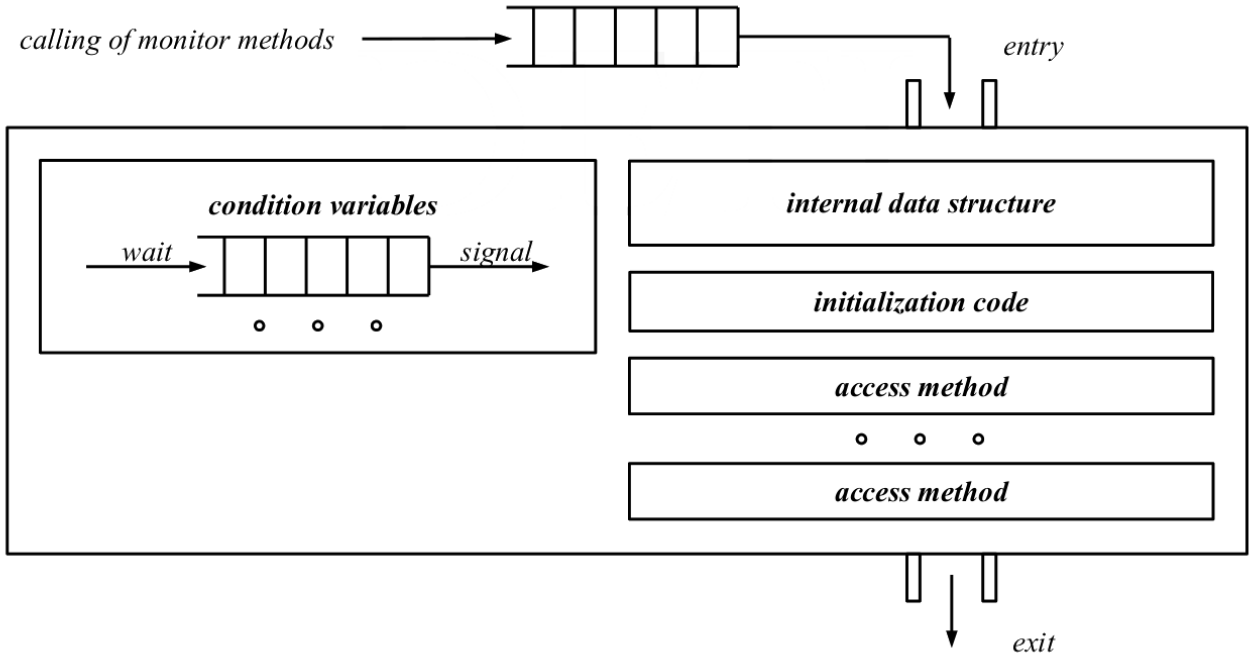

Figure 6: Diagrama da estrutura interna de um Monitor de Brinch Hansen

- A última instrução dos métodos do monitor é `signal`
  - Após o `signal` a *thread* sai do monitor
- **Fácil de implementar:** não requer nenhuma estrutura externa ao monitor
- **Restritiva: Obriga** a que cada método só possa possuir uma instrução de `signal`

46.2.3 Lampson/Redell Monitors

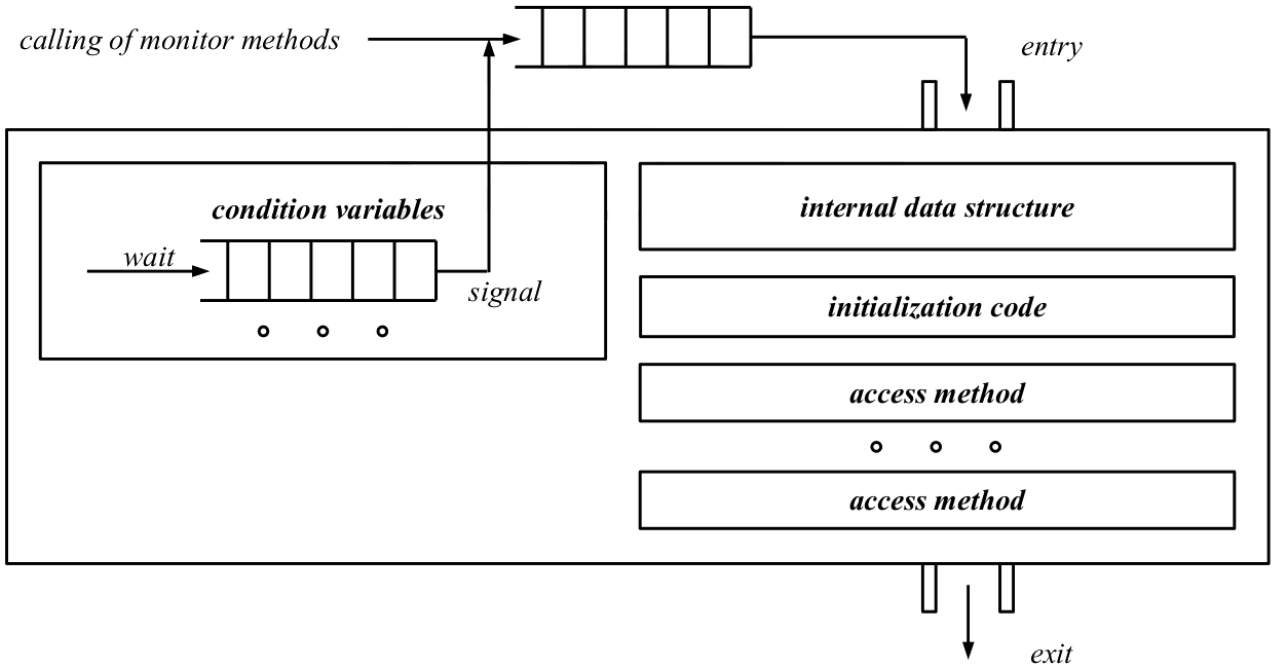

Figure 7: Diagrama da estrutura interna de um Monitor de Lampson/Redell

- A *thread* que faz o `signal` é a que continua a sua execução (entrando no monitor)
- A *thread* que é acordada devido ao `signal` fica fora do monitor, **competindo pelo acesso** ao monitor
- Pode causar **starvation**.
  - Não existem garantias que a **thread** que foi acordada e fica em competição por acesso vá ter acesso
  - Pode ser **acordada** e voltar a **bloquear**
  - Enquanto está em `ready` nada garante que outra *thread* não dê um `signal` e passe para o estado `ready`
  - A *thread* que ti nha sido acordada volta a ser **bloqueada**

46.3 Bounded-Buffer Problem usando Monitores

```
1  shared FIFO fifo;            /* fixed-size FIFO memory */
2  shared mutex access;         /* mutex to control mutual exclusion */
3  shared cond nslots;      /* condition variable to control availability of slots*/
4  shared cond nitems;      /* condition variable to control availability of items */
5
6  /* producers - p = 0, 1, ..., N-1 */
7  void producer(unsigned int p)
8  {
9      DATA data;
10     forever
11     {
12         produce_data(&data);
13         lock(access);
```

```
14          if/while (fifo.isFull())
15          {
16              wait(nslots, access);
17          }
18          fifo.insert(data);
19          unlock(access);
20          signal(nitems);
21          do_something_else();
22      }
23  }
24
25  /* consumers - c = 0, 1, ..., M-1 */
26  void consumer(unsigned int c)
27  {
28      DATA data;
29      forever
30      {
31          lock(access);
32          if/while (fifo.isEmpty())
33          {
34              wait(nitems, access);
35          }
36          fifo.retrieve(&data);
37          unlock(access);
38          signal(nslots);
39          consume_data(data);
40          do_something_else();
41      }
42  }
```

O uso de `if`/`while` deve-se às diferentes implementações de monitores:

- `if`: **Brinch Hansen**
  - quando a *thread* efetua o `signal` sai imediatamente do monitor, podendo entrar logo outra *thread*
- `while`: **Lamson Redell**
  - A *thread* acordada fica à espera que a *thread* que deu o `signal` termine para que possa **disputar** o acesso
- O `wait` internamente vai **largar a exlcusão mútua**
  - Se não larga a exclusão mútua, mais nenhum processo consegue entrar
  - Um wait na verdade é um `lock(..)` seguid de `unlock(...)`
- Depois de efetuar uma **inserção**, é preciso efetuar um `signal` do nitems
- Depois de efetuar um **retrieval** é preciso fazer um `signal` do nslots
  - Em comparação, num semáforo quando faço o up é sempre incrementado o seu valor
- Quando uma *thread* emite um `signal` relativo a uma variável de transmissão, ela só **emite** quando alguém está à escuta
  - O `wait` só pode ser feito se a FIFO estiver cheia
  - O `signal` pode ser sempre feito

É necessário existir a `fifo.empty()` e a `fifo.full()` porque as variáveis de controlo não são semáforos binários.

O valor inicial do **mutex** é 0.

46.4 POSIX support for monitors

A criação e sincronização de *threads* usa o *Standard POSIX, IEEE 1003.1c*.

O *standard* define uma API para a **criação** e **sincronização** de *threads*, implementada em unix pela biblioteca *pthread*

O conceito de monitor **não existe**, mas a biblioteca permite ser usada para criar monitores *Lampsom/Redell* em C/C++, usando:

- `mutexes`
- `variáveis de condição`

As funções disponíveis são:

- `ptread_create`: **cria** uma nova *thread* (similar ao *fork*)
- `ptread_exit`: equivalente à `exit`
- `ptread_join`: equivalente à `waitpid`
- `ptread_self`: equivalente à `getpid`
- `pthread_mutex_*`: manipulação de **mutexes**
- `ptread_cond_*`: manipulação de **variáveis condicionais**
- `ptread_once`: inicialização

47 Message-passing

Os processos podem comunicar entre si usando **mensagens**.

- Não existe a necessidade de possuirem memória partilhada
- Mecanismos válidos quer para sistemas **uniprocessador** quer para sistemas **multiprocessador**

A **comunicação** é efetuada através de **duas operações**:

- `send`
- `receive`

Requer a existência de um **canal de comunicação**. Existem 3 implementações possíveis:

1. **Endereçamento direto/indireto**
2. Comunicação **síncrona/assíncrona**
    - Só o `sender` é que indica o **destinatário**
    - O destinatário **não indica** o `sender`
    - Quando existem **caixas partilhadas**, normalmente usam-se mecanismos com políticas de **round-robin**
        1. Lê o processo N
        2. Lê o processo $N + 1$
        3. etc…
    - No entanto, outros métodos podem ser usados
3. **Automatic or expliciting buffering**

47.1 Direct vs Indirect

47.1.1 Symmetric direct communication

O processo que pretende comunicar deve **explicitar o nome do destinatário/remetente:**

- Quando o `sender` envia uma mensagem tem de indicar o **destinatário**
  - `send(P, message`
- O destinatário tem de indicar de quem **quer receber** (`sender`)
  - `receive(P, message)`

A comunicação entre os **dois processos** envolvidos é **peer-to-peer**, e é estabelecida automaticamente entre entre um conjunto de processos comunicantes, só existindo **um canal de comunicação**

47.2 Assymetric direct communications

Só o `sender` tem de explicitar o destinatário:

- `send(P, message`:
- `receive(id, message)`: receve mensagens de qualquer processo

47.3 Comunicação Indireta

As mensagens são enviadas para uma **mailbox** (caixa de mensagens) ou **ports**, e o `receiver` vai buscar as mensagens a uma `poll`

- `send(M, message`
- `receive(M, message)`

O canal de comunicação possui as seguintes propriedades:

- Só é estabelecido se o **par de processos** comunicantes possui uma **`mailbox` partilhada**
- Pode estar associado a **mais do que dois processos**
- Entre um par de processos pode existir **mais do que um link** (uma mailbox por cada processo)

Questões que se levantam. Se **mais do que um processo** tentar **receber uma mensagem da mesma `mailbox`**…

- … é permitido?
  - Se sim. qual dos processos deve ser bem sucedido em ler a mensagem?

47.4 Implementação

Existem várias opções para implementar o **send** e **receive**, que podem ser combinadas entre si:

- **blocking send:** o `sender` **envia** a mensagem e fica **bloquedo** até a mensagem ser entregue ao processo ou mailbox destinatária
- **nonblocking send:** o `sender` após **enviar** a mensagem, **continua** a sua execução
- **blocking receive:** o `receiver` bloqueia-se até estar disponível uma mensagem para si
- **nonblocking receiver:** o `receiver` devolve a uma mensagem válida quando tiver ou uma indicação de que não existe uma mensagem válida quando não tiver

47.5 Buffering

O link pode usar várias políticas de implementação:

- **Zero Capacity:**
  - Não existe uma `queue`
  - O `sender` só pode enviar uma mensagem de cada vez. e o envio é **bloqueante**
  - O `receiver` lê uma mensagem de cada vez, podendo ser bloqueante ou não
- **Bounded Capacity:**
  - A `queue` possui uma capacidade finita
  - Quando está cheia, o `sender` bloqueia o envio até possuir espaço disponível
- **Unbounded Capacity:**
  - A `queue` possui uma capacidade (potencialmente) infinita
  - Tanto o `sender` como o `receiver` podem ser **não bloqueantes**

47.6 Bound-Buffer Problem usando mensanges

```
1  shared FIFO fifo;            /* fixed-size FIFO memory */
2  shared mutex access;         /* mutex to control mutual exclusion */
3  shared cond nslots;      /* condition variable to control availability of slots*/
4  shared cond nitems;      /* condition variable to control availability of items */
5
6  /* producers - p = 0, 1, ..., N-1 */
7  void producer(unsigned int p)
8  {
9      DATA data;
10     MESSAGE msg;
11
12     forever
13     {
14         produce_data(&val);
15         make_message(msg, data);
16         send(msg);
17         do_something_else();
18     }
19 }
20
21 /* consumers - c = 0, 1, ..., M-1 */
22 void consumer(unsigned int c)
23 {
24     DATA data;
25     MESSAGE msg;
26
27     forever
28     {
29         receive(msg);
30         extract_data(data, msg);
31         consume_data(data);
```

```
32          do_something_else();
33      }
34  }
```

47.7 Message Passing in Unix/Linux

System V:

- Existe uma fila de mensagens de **diferentes tipos**, representados por um inteiro
- `send` **bloqueante** se **não existir espaço disponível**
- A receção possui um argumento para espcificar o **tipo de mensagem a receber**:
  - Um tipo específico
  - Qualquer tipo
  - Um conjunto de tipos
- Qualquer que seja a política de receção de mensagens:
  - É sempre **obtida** a mensagem **mais antiga** de uma dado tipo(s)
  - A implementação do `receive` pode ser **blocking** ou **nonblocking**
- System calls:
  - `msgget`
  - `msgsnd`
  - `msgrcv`
  - `msgctl`

POSIX

- Existe uma **priority** **`queue`**
- `send` **bloqueante** se **não existir espaço disponível**
- `receive` obtêm a mensagem **mais antiga** com a **maior prioridade**
  - Pode ser blocking ou nonblocking
- Funções:
  - `mq_open`
  - `mq_send`
  - `mq_receive`

48 Shared Memory in Unix/Linux

- É um recurso gerido pelo sistema operativo

Os espaços de endereçamento são **independentes** de processo para processo, mas o **espaço de endereçamento** é virtual, podendo a mesma **região de memória física**(memória real) estar mapeada em mais do que uma **memórias virtuais**

48.1 POSIX Shared Memory

- Criação:
  - `shm_open`
  - `ftruncate`
- Mapeamento:
  - `mmap`
  - `munmap`
- Outras operações:
  - `close`
  - `shm_unlink`
  - `fchmod`
  - …

48.2 System V Shared Memory

- Criação:
  - `shmget`
- Mapeamento:
  - `shmat`
  - `shmdt`
- Outras operações:
  - `shmctl`

49 Deadlock

- **recurso:** algo que um processo precisa para proseeguir com a sua execução. Podem ser:
  - **componentes físicos** do sistema computacional, como:
    * processador
    * memória
    * dispositivos de I/O
    * …
  - **estruturas de dados partilhadas**. Podem estar definidas
    * Ao nível do sistema operativo
      · PCT
      · Canais de Comunicação
    * Entre vários processos de uma aplicação

Os recursos podem ser:

- **preemptable:** podem ser retirados aos processos que estão na sua posse por entidades externas

  - processador
  - regiões de memória usadas no espaço de endereçamento de um processo

- **non-preemptable:** os recursos só podem ser libertados pelos processos que estão na sua posse
  - impressoras
  - regiões de memória partilhada que requerem acesso por exclusão mútua

O **deadlock** só é importante nos recursos **non-preemptable**.

O caso mais simples de deadlock ocorre quando:

1. O processo P_0 pede a posse do recurso A
   - É lhe dada a posse do recurso A, e o processo P_0 passa a possuir o recurso A em sua posse
2. O processo P_1 pede a posse do recurso B
   - É lhe dada a posse do recurso B, e o processo P_1 passa a possuir o recurso B em sua posse
3. O processo P_0 pede agora a posse do recurso B
   - Como o recurso B está na posse do processo P_1, é lhe negado
   - O processo P_0 fica em espera que o recurso B seja libertado para puder continuar a sua execução
   - No entanto, o processo P_0 não liberta o recurso A
4. O processo P_1 necessita do recurso A
   - Como o recurso A está na posse do processo P_0, é lhe negado
   - O processo P_1 fica em espera que o recurso A seja libertado para puder continuar a sua execução
   - No entanto, o processo P_1 não liberta o recurso B
5. Estamos numa situação de **deadlock**. Nenhum dos processos vai libertar o recurso que está na sua posse mas cada um deles precisa do recurso que está na posse do outro

49.1 Condições necessárias para a ocorrência de deadlock

Existem 4 condições necessárias para a ocorrência de **deadlock:**

1. **exclusão mútua:**
   - Pelo menos um dos recursos fica em posse de um processo de forma não partilhável
   - Obriga a que outro processo que precise do recurso espere que este seja libertado
2. **hold and wait:**
   - Um processo mantêm em posse pelo menos um recurso enquanto espera por outro recurso que está na posse de outro processo
3. **no preemption:**
   - Os recursos em causa são non-preemptive, o que implica que só o processo na posse do recurso o pode libertar
4. **espera circular:**
   - é necessário um conjunto de processos em espera tais que cada um deles precise de um recurso que está na posse de outro processo nesse conjunto

Se **existir deadlock**, todas estas condições se verificam. *(A => B)*

Se **uma delas não se verifica**, não há deadlock. *(~B => ~A)*

49.1.1 O Problema da Exclusão Mútua

Dijkstra em 1965 enunciou um conjunto de regras para garantir o acesso **em exclusão mútua** por processo em competição por recursos de memória partilhados entre eles.[1]

1. **Exclusão Mútua:** Dois processos não podem entrar nas suas zonas críticas ao mesmo tempo
2. **Livre de Deadlock:** Se um process está a tentar entrar na sua zona crítica, eventualemnte algum processo (não necessariamento o que está a tentar entrar), mas entra na sua zona crítica
3. **Livre de Starvation:** Se um processo está atentar entrar na sua zona crítica, eentão eventualemnte esse processo entr na sua zona crítica
4. **First-In-First-Out:** Nenhum processo qa iniciar pode entrar na sua zona crítica antes de um processo que já está à espera do seu trunos para entrar na sua zona crítica

49.2 Jantar dos Filósofos

- 5 filósofos sentados à volta de uma mesa, com comida à sua frente
  - Para comer, cada filósofo precisa de 2 garfos, um à sua esquerda e outro à sua direita
  - Cada filósofo alterna entre períodos de tempo em que medita ou come
- Cada **filósofo** é um **processo/thread** diferente
- Os **garfos** são os **recursos**

Uma possível solução para o problema é:

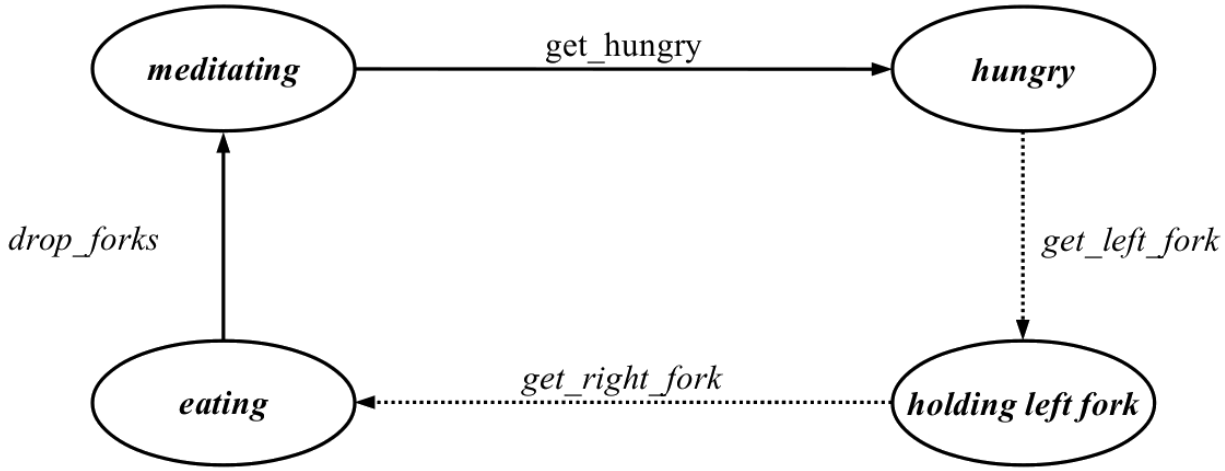

Figure 8: Ciclo de Vida de um filósofo

```
1  enum {MEDITATING, HUNGRY, HOLDING, EATING};
2
3  typedef struct TablePlace
4  {
5      int state;
```

[1] *"Concurrent Programming, Mutual Exclusion (1965; Dijkstra)".* Gadi Taubenfeld, The Interdisciplinary Center, Herzliya, Israel

```
} TablePlace;

typedef struct Table
{
    Int semid;
    int nplaces;
    TablePlace place[0];
} Table;

int set_table(unsigned int n, FILE *logp);
int get_hungry(unsigned int f);
int get_left_fork(unsigned int f);
int get_right_fork(unsigned int f);
int drop_forks(unsigned int f);
```

Quando um filósofo fica *hungry*:

1. Obtém o garfo à sua esquerda
2. Obtém o garfo à sua direita

A solução **pode sofrer de deadlock:**

1. **exclusão mútua:**
   - Os garfos são partilháveis
2. **hold and wait:**
   - Se conseguir adquirir o `left_fork`, o filósofo fica no estado `holding_left_fork` até conseguir obter o `right_fork` e não liberta o `left_fork`
3. **no preemption:**
   - Os garfos são recursos non-preemptive. Só o filósofo é que pode libertar os seus garfos após obter a sua posse e no fim de comer
4. **espera circular:**
   - Os garfos são partilhados por todos os filósofos de forma circular
     - O garfo à esquerda de um filósofo, `left_fork` é o garfo à direita do outro, `right_fork`

Se todos os filósofos estiverem a pensar e decidirem comer, pegando todos no garfo à sua esquerda ao mesmo tempo, entramos numa situação de **deadlock**.

49.3 Prevenção de Deadlock

Se uma das condições necessárias para a ocorrência de deadlock não se verificar, não ocorre deadlock.

As **políticas de prevenção de deadlock** são bastantes **restritas**, **pouco efetivas** e **difíceis de aplicar** em várias situações.

- **Negar a exclusão mútua** só pode ser aplicada a **recursos partilhados**
- **Negar *hold-and-wait*** requer **conhecimento *a priori* dos recursos necessários** e considera sempre o pior caso, no qual os recursos são todos necessários em simultâneo (o que pode não ser verdade)
- **Negar *no preemption***, impondo a libertação (e posterior re-aquisição) de recursos adquiridos por processos que não têm condições (aka, todos os recursos que precisam) para continuar a execução pode originar grandes atrasos na execução da tarefa
- **Negar a *circular wait*** pode resultar numa má gestão de recursos

49.3.1 Negar a exclusão mútua

- Só é possível se os recursos puderem ser partilhados, senão podemos incorrer em **race conditions**
- Não é possível no jantar dos filśofos, porque os garfos não podem ser partilhados entre os filósofos
- Não é a condição mais vulgar a negar para prevenir *deadlock*

49.3.2 Negar *hold-and-wait*

- É possível fazê-lo se um processo é obrigado a pedir todos os recursos que vai precisar antes de iniciar, em vez de ir obtendo os recursos à medida que precisa deles
- Pode ocurrer **starvation**, porque um processo pode nunca ter condições para obter nenhum recurso
  - É comum usar *aging mechanisms* to para resolver este problema
- No jantar dos filósofos, quando um filósofo quer comer, passa a adquirir os dois garfos ao mesmo tempo
  - Se estes não tiverem disponíveis, o filósofo espera no `hungry state`, podendo ocorrer **starvation**

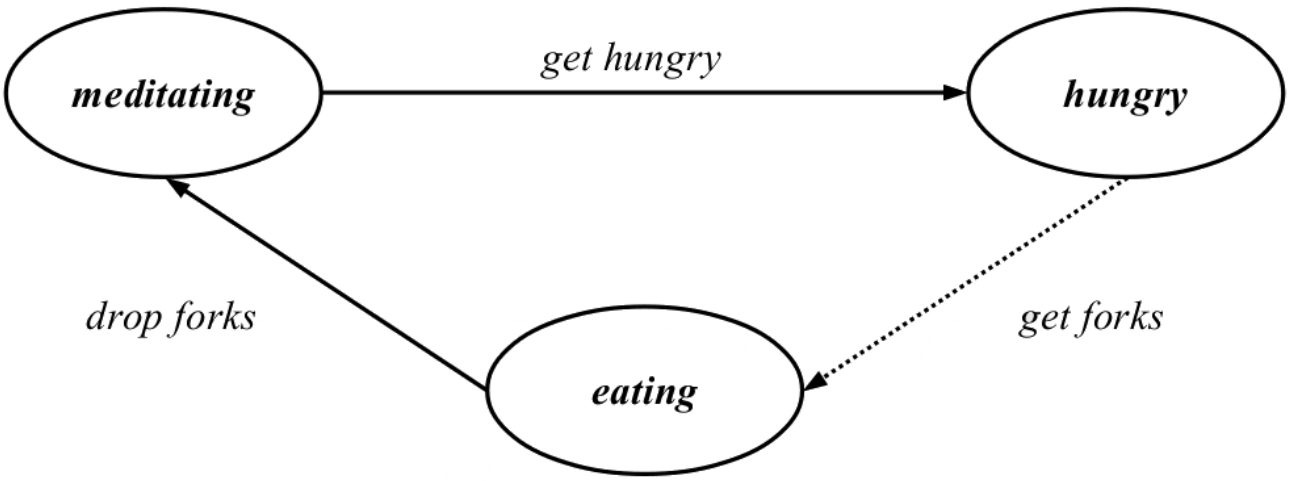

Figure 9: Negar *hold-and-wait*

Solução equivalente à proposta por Dijkstra.

49.3.3 Negar *no preemption*

- A condição de os recursos serem *non-preemptive* pode ser implementada fazendo um processo libertar o(s) recurso(s) que possui se não conseguir adquirir o próximo recurso que precisa para continuar em execução
- Posteriormente o processo tenta novamente adquirir esses recursos
- Pode ocorrer **starvation** and **busy waiting**
  - podem ser usados *aging mechanisms* para resolver a starvation

  - para evitar busy waiting, o processo pode ser bloqueado e acordado quando o recurso for libertado

- No janta dos filósofos, o filósofo tenta adquirir o `left_fork`
  - Se conseguir, tenta adquirir o `right_fork`
    * Se conseguir, come
    * Se não conseguir, liberta o `left_fork` e volta ao estado `hungry`

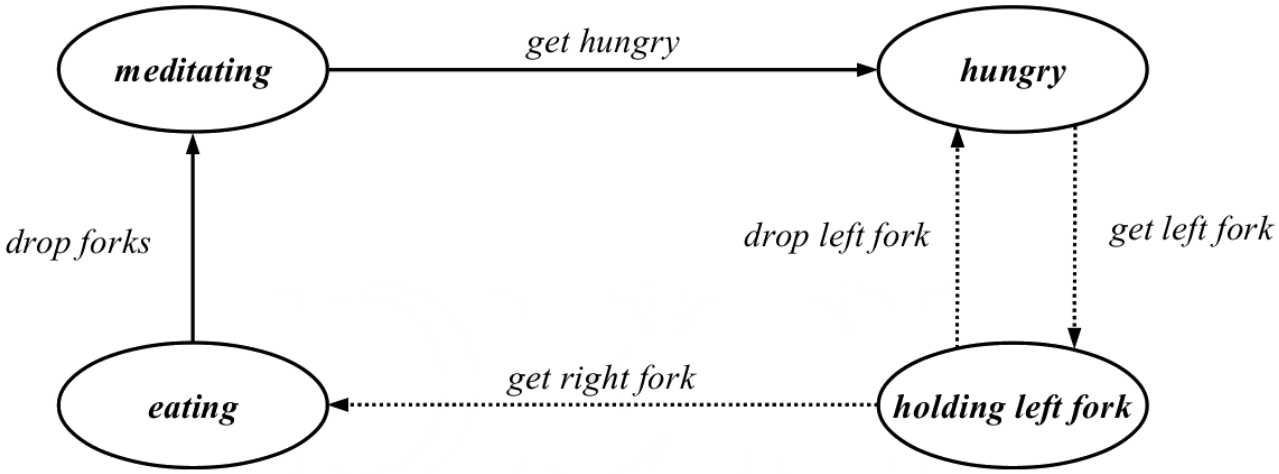

Figure 10: Negar a condição de *no preemption* dos recursos

49.3.4 Negar a espera circular

- Através do uso de IDs atribuídos a cada recurso e impondo uma ordem de acesso (ascendente ou descendente) é possível evitar sempre a espera em círculo
- Pode ocorrer **starvation**
- No jantar dos filósofos, isto implica que nalgumas situações, um dos filsófos vai precisar de adquirir primeiro o `right_fork` e de seguida o `left_fork`
  - A cada filósofo é atribuído um número entre 0 e N
  - A cada garfo é atribuído um ID (e.g., igual ao ID do filósofo à sua direita ou esquerda)
  - Cada filsófo adquire primeiro o garfo com o menro ID
  - obriga a que os filósofos 0 a N-2 adquiram primeiro o `left_fork` enquanto o filósofo N-1 adquir primeiro o `right_fork`

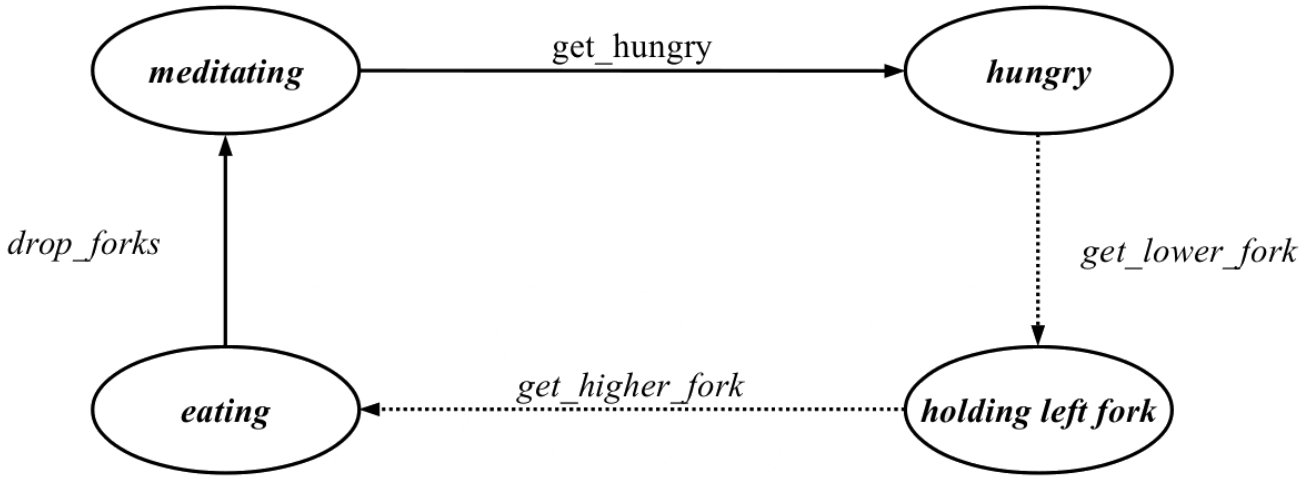

Figure 11: Negar a condição de espera circular no acesso aos recursos

49.4 Deadlock Avoidance

Forma menos restritiva para resolver situações de deadlock, em que **nenhuma das condições necessárias à ocorrência de deadlock é negada**. Em contrapartida, o sistema é **monitorizado continuamente** e um recurso **não é atribuído** se como consequência o sistema entrar num **estado inseguro/instável**

Um estado é considerado seguro se existe uma sequência de atribuição de recursos na qual todos os processos possa terminar a sua execução (não ocorrendo *deadlock*).

Caso contrário, poderá ocorrer deadlock (pode não ocorrer, mas estamos a considerar o pior caso) e o estado é considerado inseguro.

Implica que:

- exista uma lista de todos os recursos do sistema
- os processos intervenientes têm de declarar *a priori* todas as suas necessidades em termos de recursos

49.4.1 Condições para lançar um novo processo

Considerando:

- NTR_i - o número total de recursos do tipo `i` *(i = 0, 1, …, N-1)*
- $R_{i,j}$: o número de recursos do tipo `i` requeridos pelo processo `j`, *(i=0, 1, …, N-1 e j=0, 1, …, M-1)*

O sistema pode impedir um novo processo, `M`, de ser exectuado se a sua terminação não pode ser garantida. Para que existam certezas que um novo processo pode ser terminado após ser lançado, tem de se verificar:

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

49.4.2 Algoritmo dos Banqueiros

Considerando:

- NTR_i: o número total de recursos do tipo `i` *(i = 0, 1, …, N-1)*
- $R_{i,j}$: o número de recursos do tipo `i` requeridos pelo processo `j`, *(i=0, 1, …, N-1 e j=0, 1, …, M-1)*
- $A_{i,j}$: o número de recursos do tipo `i` atribuídos/em posse do processo `j`, *(i=0, 1, …, N-1 e j=0, 1, …, M-1)*

Um novo recurso do tipo `i` só pode ser atribuído a um processo **se e só se** existe uma sequência $j' = f(i, j)$ tal que:

$$R_{i,j'} - A_{i,j'} < \sum_{k \geq ji'}^{M-1} A_{i,k}$$

Table 5: Banker's Algorithm Example

		A	B	C	D
	total	6	5	7	6
	free	3	1	1	2
	p1	3	3	2	2
maximum	p2	1	2	3	4
	p3	1	3	5	0
	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
	p1	2	1	0	1
needed	p2	0	2	0	1
	p3	0	1	4	0
	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	0	0	0

Para verificar se posso atribuir recursos a um processo, aos recursos `free` subtraio os recursos `needed`, ficando com os recursos que sobram. Em seguida simulo o que aconteceria se atribuisse o recurso ao processo, tendo em consideração que o processo pode usar o novo recurso que lhe foi atribuído sem libertar os que já possui em sua posse (estou a avaliar o pior caso, para garantir que não há deadlock)

Se o processo **p3** pedir 2 recursos do tipo C, o **pedido é negado**, porque **só existe 1 disponível**

Se o processo **p3** pedir 1 recurso do tipo B, o **pedido é negado**, porque apesar de existir 1 recurso desse tipo disponível, ao **longo da sua execução processo vai necessitar de 4** e só **existe 1 disponível**, podendo originar uma situação de **deadlock**, logo o **acesso ao recurso é negado**

Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

- Cada filósofo primeiro obtém o `left_fork` e depois o `right_fork`

- No entanto, se um dos filósofos tentar obter um `left_fork` e o filósofo à sua esquerda já tem na sua posse um `left_fork`, o acesso do filósofo sem garfos ao `left_fork` é negado para não ocorrer **deadlock**

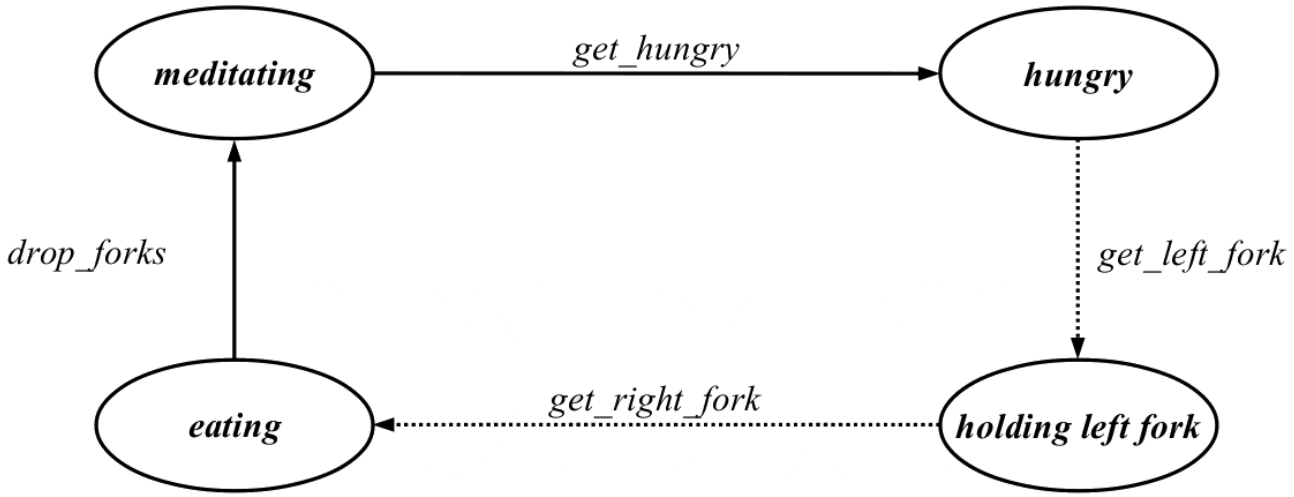

Figure 12: Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

49.5 Deadlock Detection

Não são usados mecanismos nem para prevenir nem par evitar o deadlock, podendo ocorrer situações de deadlock:

- O estado dos sistema deve ser examinado para determinar se ocorreu uma situação de deadlock
  - É preciso verificar se existe uma **dependência circular de recursos** entre os processos
  - Periodicamente é executado um algoritmo que verifica o estado do registo de recursos:
    * recursos `free` vs recursos `granted` vs recursos `needed`
  - Se tiver ocorrido uma situação de deadlock, o SO deve possuir uma **rotina de recuperação** de situações de deadlock e executá-la
- Alternativamente, de um ponto de vista "arrogante", o problema pode ser ignorado

Se **ocorrer uma situação de deadlock**, a rotina de recuperação deve ser posta em prática com o objetivo de interromper a dependência circular de processos e recursos.

Existem três métodos para recuperar de deadlock:

- **Libertar recursos de um processo**, se possível
  - É atividade de um processo é suspensa até se puder devolver o recurso que lhe foi retirado
  - Requer que o estado do processo seja guardado e em seguida recarregado
  - Método eficiente
- **Rollback**
  - O estado de execução dos diferentes processos é guardado periodicamente
  - Um dos processos envolvidos na situação de deadlock é *rolled back* para o instante temporal em que o recurso lhe foi atribuído
  - A recurso é assim libertado do processo

- **Matar o processo**
  - Quando um processo entra em deadlock, é terminado
  - Método radical mas fácil de implementar

Alternativamente, existe sempre a opção de não fazer nada, entrando o processo em deadlock. Nestas situações, o utilizador é que é responsável por corrigir as situações de deadlock, por exemplo, terminando o programa com `CTRL + C`

50 Processes and Threads

50.1 Arquitectura típica de um computador

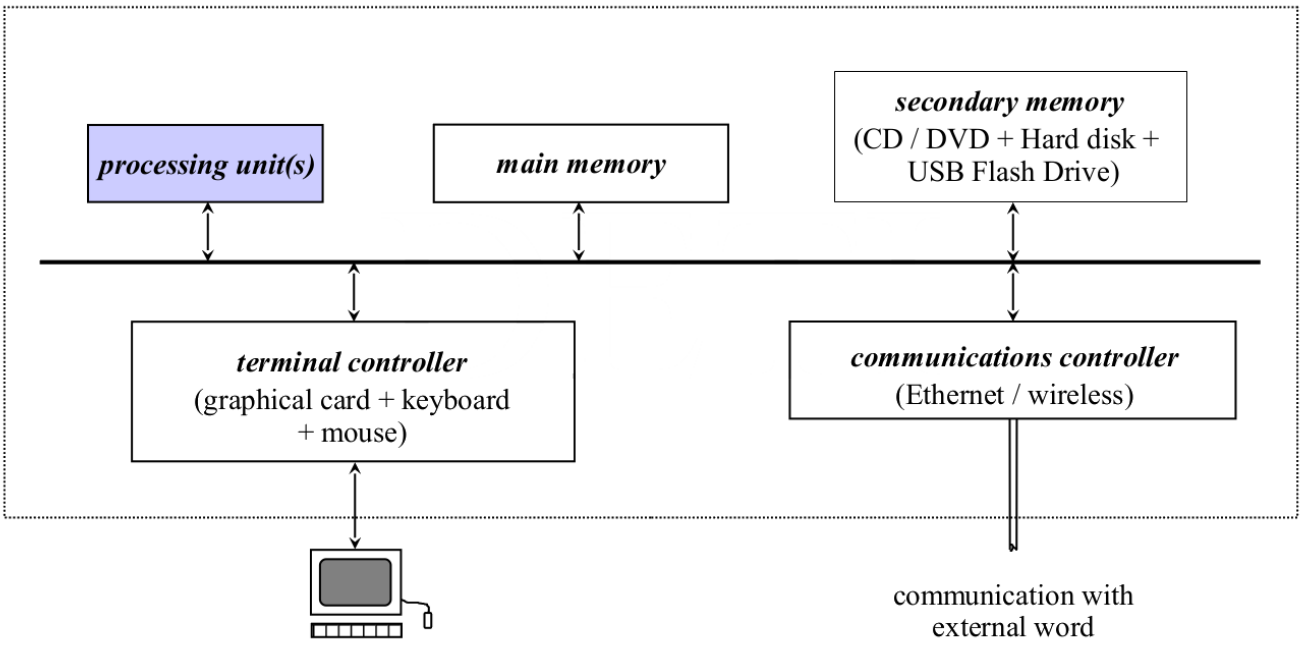

Figure 13: Arquitectura típica de um computador

50.2 Programa vs Processo

- **programa**: conjunto de instruções que definem como uma tarefa é executada num computador
  - É apenas um **conjunto de instruções** (código máquina), nada mais
  - Para realizar essas **funções/instruções/tarefas** o código (ou a versão compilada dele) tem de ser executado(a)
- **processo:** Entidade que representa a **execução de um programa**
  - Representa a sua atividade
  - Tem associado a si:
    * código que ao contrário do programa está armazenado num endereço de memória (addressing space)
    * **dados** (valores das diferentes variáveis) da execução corrente
    * valores atuais dos registos internos do processador
    * dados dos I/Os, ou seja, dados que estão a ser transferidos entre dispositivos de input e output
    * Estado da execução do programa, ou seja, qual a próxima execução a ser executada (registo PC)
  - Podem existir diferentes processos do mesmo programa
    * Ambiente **multiprogramado** - mais processos que processadores

50.3 Execução num ambiente multiprogramado

O sistema assume que o processo que está na posse do processador irá **ser interrompido**, pudendo assim executar outro processo e dar a "sensação" em **macro tempo** de **simultaneidade**. Nestas situações, o OS é responsável por:

- tratar da **mudança do contexto de execução**, guardando
  - o valor dos registos internos
  - o valor das variáveis
  - o endereço da próxima instrução a ser executada
- chamar o novo processo que vai ocupar agora o CPU e:
  - Esperar que o novo processo termine a realização das suas operações **ou**
  - Interromper o processo, **parando a sua execução no** processador quando este esgotar o seu `time quantum`

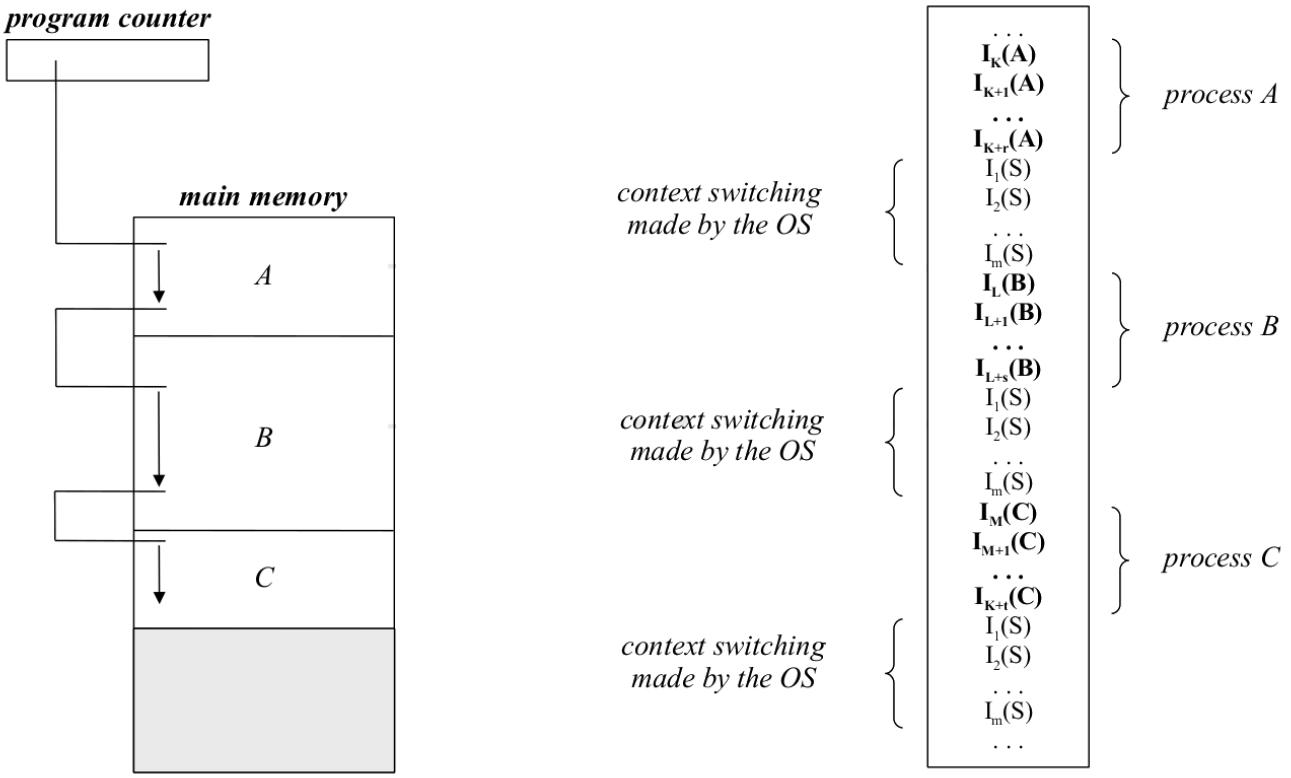

Figure 14: Exemplo de execução num ambiente multiprogramado

50.4 Modelo de Processos

Num ambiente **multiprogramado**, devido à constante **troca de processos**, é difícil expressar uma modelo para o processador. Devido ao elevado numero de processo e ao multiprogramming, torna-se difícil de saber qual o processo que está a ser executado e qual a fila de processos as ser executada.

É mais fácil assumir que o ambiente multiprogramado pode ser representado por um **conjunto de processadores virtuais**, estando um processo atribuído a cada um.

O processador virtual está: - **ON:** se o processo que lhe está atribuído está a ser executado - **OFF:** se o processo que lhe está atribuído não está a ser executado

Para este modelo temos ainda de assumir que: - Só um dos processadores é que pode estar ativo num dado período de tempo - O número de **processadores virtuais ativos é menor** (ou igual, se for um ambiente `single processor`) ao número de **processadores reais** - A execução de um processo **não é afetada** pelo instante temporal nem a localização no código em

que o processo é interrompido e é efetuado o switching - Não existem restrições do número de vezes que qualquer processo pode ser interrompido, quer seja executado total ou parcialmente

A operação de **switching entre processos** e consequentemente entre processadores virtuais ocorre de forma não **controlada** pelo programa a correr no CPU

Uma **operação de switching** é equivalente a efetuar o `Turning Off` de um processo virtual e o `Turning On` de outro processo virtual.

- `Turning Off` implica **guardar** todo o **contexto de execução**
- `Turning On` implica carregar todo o contexto de execução, **restaurando o estado do programa** quando foi interrompido

50.5 Diagrama de Estados de um Processo

Durante a sua existência, um processo pode assumir diferentes estados, dependendo das condições em que se encontra:

- **run:** O processo está em execução, tendo a posse do processador
- **blocked:** O processo está bloqueado à **espera de um evento externo** para estar em condições retomar a sua execução. Esse evento externo pode ser:
  - Acesso a um recurso da máquina
  - Fim de uma operação de I/O
  - …
- **ready:** O processo está pronto a ser executado, mas está à espera que o processador lhe dê a ordem de `start`/`resume` para puder retomar a sua execução.

As **transições entre estados** normalmente resultam de **intervenções externas ao processo**, mas podem depender de situações em que o processo força uma transição: - termina a sua execução antes de terminar o seu `time quantum` - Leitura/Escrita em I/O (scanf/printf)

Mesmo que um processo **não abandone o processador por sua iniciativa**, o `scheduler` é responsável por **planear o uso do processador pelos diferentes processos**.

O `(Process)Scheduler` é um módulo do kernel que **monitoriza e gere as transições entre processos**. Assim, um `while (1)` não é executado *ad eternum*. Um processador `multiprocess` só permite que o ciclo infinito seja executado quando é atribuído `CPU time` ao processo.

Existem diferentes políticas que permitem controlar a execução destas transições

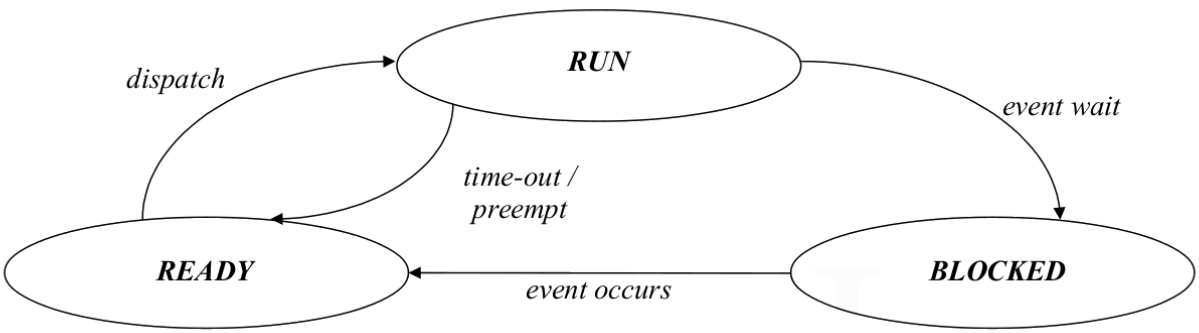

Figure 15: Diagrama de Estados do Processador - Básico

`Triggers` das transições entre estados:

- **dispatch:**
  - O processo que estava em modo `run` perdeu o acesso ao processador.
  - Do conjunto de processos prontos a serem executados, tem de ser escolhido **um** para ser executado, sendo lhe atribuído o processador.
  - A escolha feita pelo `dispatcher` pode basear-se em:
    * um sistema de prioridades
    * requesitos temporais
    * aletoriedade
    * divisao igual do CPU
- **event wait:**
  - O processo que estava a ser executado sai do estado `run`, não estando em execução no processador.
    * Ou porque é impedido de continuar pelo scheduler
    * Ou por iniciativa do proprio processo.
      · *scanf*
      · *printf*
  - O CPU guarda o estado de execução do processo
  - O processo fica em estado `blocked` à **espera da ocorrência de um evento externo**, `event occurs`
- **event occurs:**
  - Ocorreu o evento que o processo estava à espera
  - O processo transita do estado `blocked` para o estado `ready`, ficando em fila de espera para que lhe seja atribuído o processador
- **time_out:**
  - O processo esgotou a sua janela temporal, `time quantum`
  - Atraves de uma interrupção em *hardware*, o sistema operativo vai forçar a saída do processo do processador
  - Transita para o estado *ready* até lhe ser atribuído um novo `time-quantum` do CPU
  - A transição por time-out ocorre em qualquer momento do código.
  - Os sistemas podem ter `time quantum` diferentes e os `time slots` alocados não têm de ser necessariamente iguais entre dois sistemas.
- **preempt**:
  - O processo que possui a posse do processador tem uma prioridade mais baixa do que um processo que acordou e está pronto a correr (estado `ready`)
  - O processo que está a correr no processador é **removido** e transita para o estado `ready`
  - Passa a ser **executado** o processo de **maior prioridade**

50.5.1 Swap Area

O diagram de estados apresentado não leva em consideração que a **memória principal** (RAM) é **finita**. Isto implica que o número de **processos coexistents em memória é limitado**.

É necessário usar a **memória secundária** (Disco Rígido) para **extender a memória principal** e aumentar a capacidade de armazenamento dos estados dos processos.

A **memória** **swap** pode ser:

- uma partição de um disco
- um ficheiro

Qualquer processo que **não esteja a correr** por ser `swapped out`, libertando memória principal para outros processos

Qualquer processo `swapped out` pode ser `swapped in`, **quando existir memória principal disponível**

Ao diagrama de estados tem de ser adicionados: - dois novos estados: - **suspended-ready:** Um processo no estado `ready` foi `swapped-out` - **suspended-blocked:** O processo no estado `blocked` foi `swapped-out` - dois novos tipos de transições: - **suspend:** O processo é `swapped out` - **activate:** O processo é `swapped in`

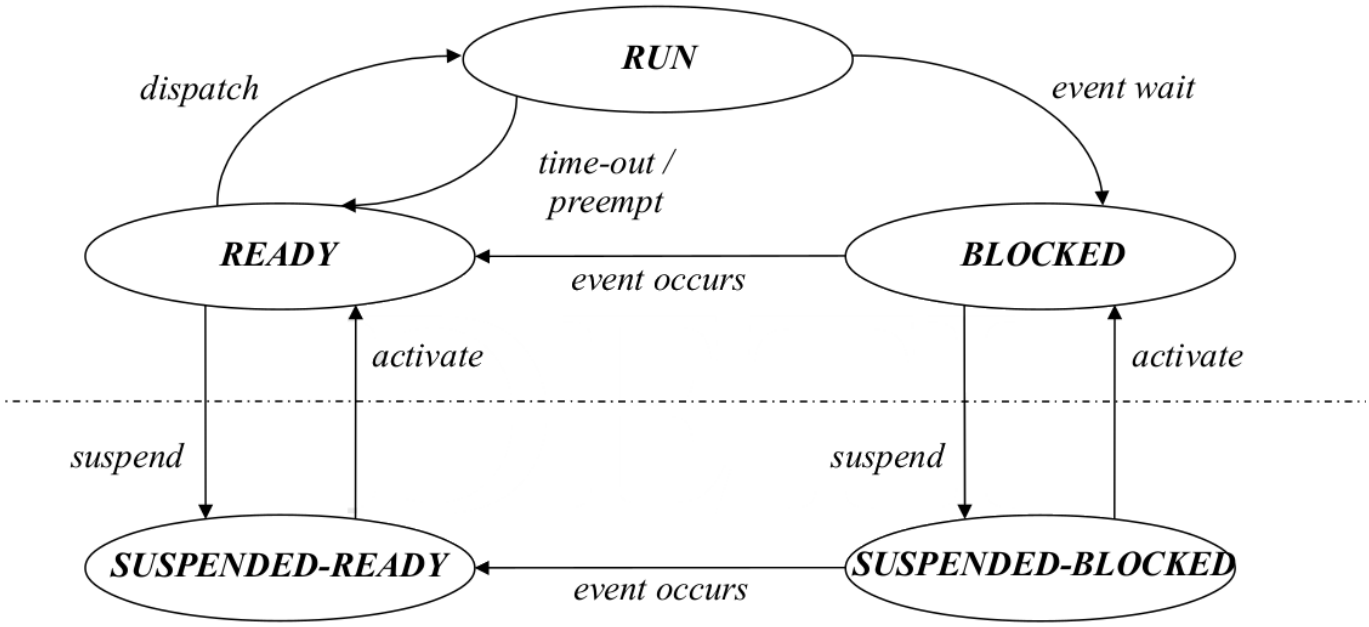

Figure 16: Diagrama de Estados do Processador - Com Memória de Swap

50.5.2 Temporalidade na vida dos processos

O diagrama assume que os processos são **intemporais**. Excluindo alguns processos de sistema, todos os processos são **temporais**, i.e.:

1. Nascem/São criados
2. Existem (por algum tempo)
3. Morrem/Terminam

Para introduzi a temporalidade no diagrama de estados, são necessários dois novos estados: - **new:** - O processo foi criado - Ainda não foi atribuido à `pool` de processos a serem executados - A estrutura de dados associado ao processo é inicializada - **terminated:** - O processo foi descartado da fila de processos executáveis - Antes de ser descartado, existem ações que tem de tomar *(needs clarification)*

Em consequência dos novos estados, passam a existir três novas transições: - **admit:** O processo é admitido pelo OS para a `pool` de processos executáveis - **exit:** O processo informa o SO que terminou a sua execução - **abort:** Um processo é forçado a terminar. - Ocorreu um `fatal error` - Um processo autorizado abortou a sua execução

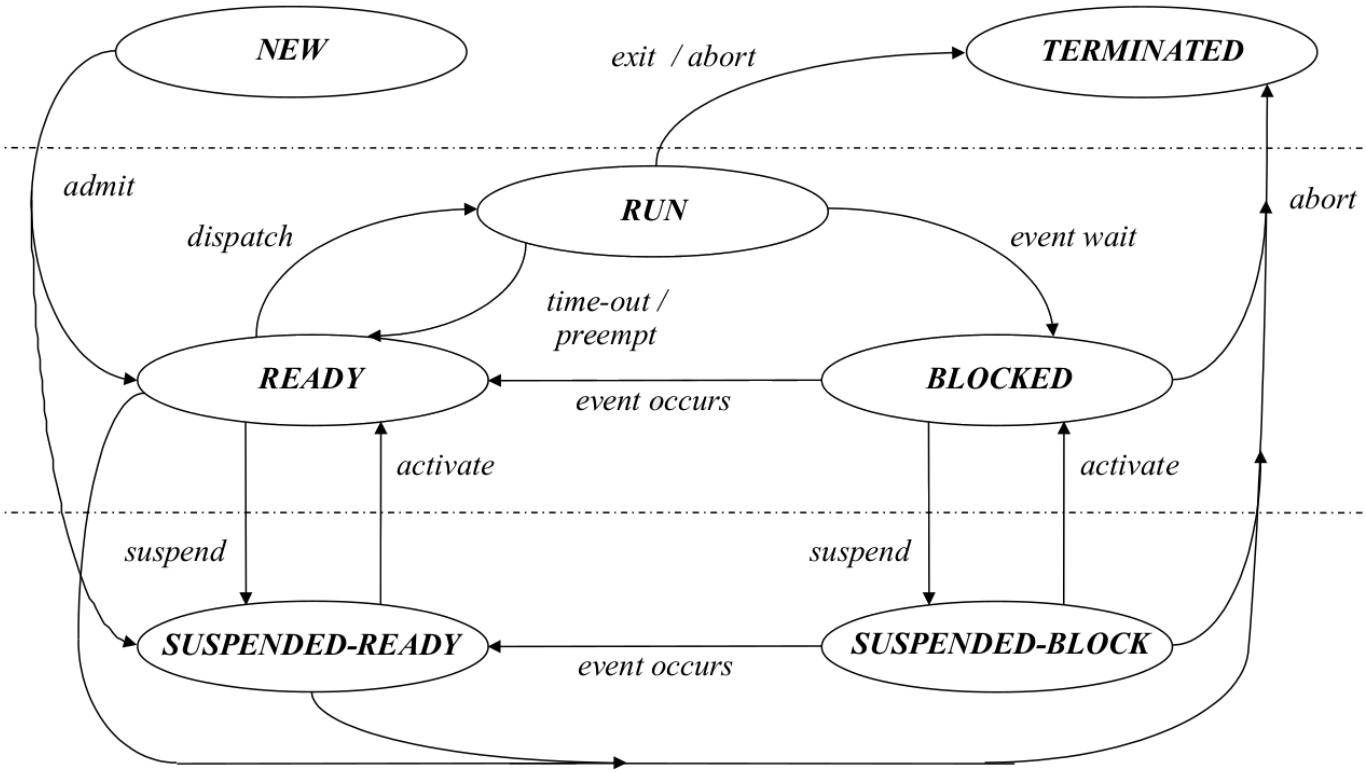

Figure 17: Diagrama de Estados do Processador - Com Processos Temporalmente Finitos

50.6 State Diagram of a Unix Process

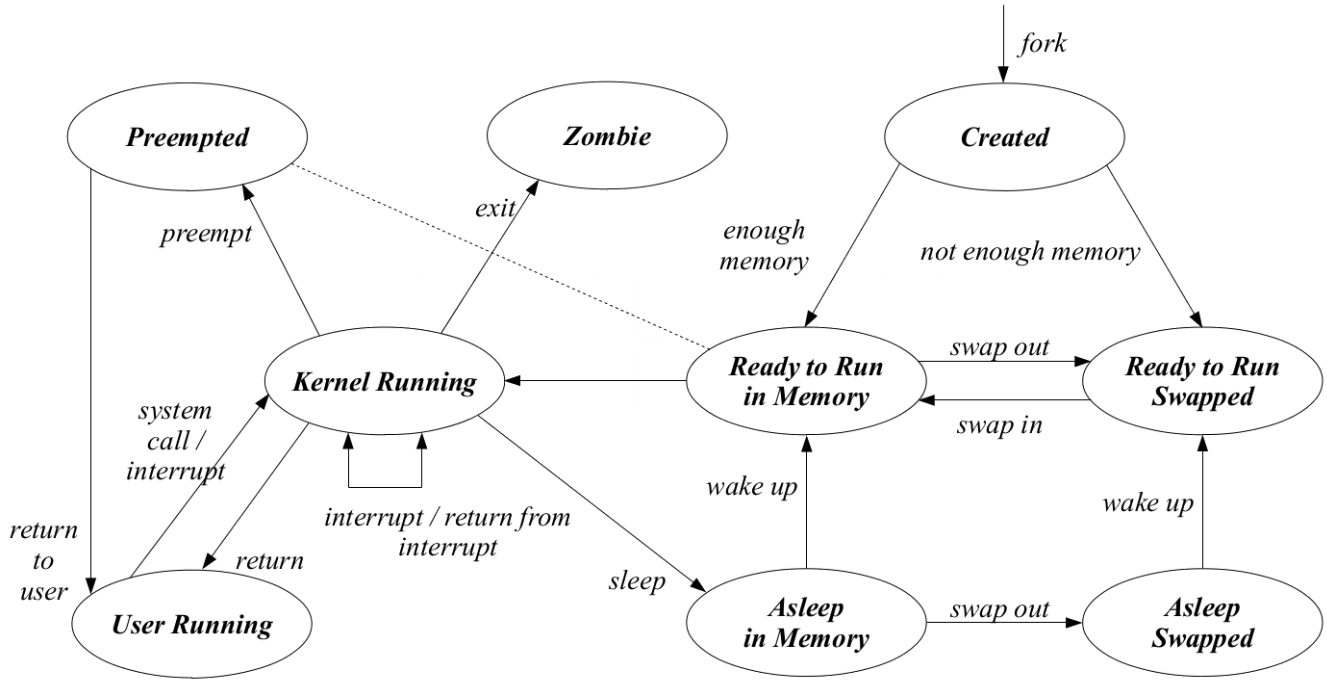

Figure 18: Diagrama de Estados do Processador - Com Memória de Swap

As três diferenças entre o diagrama de estados de um processo e o diagrama de estados do sistema Unix são

1. Existem **dois estados run**
    1. **kernel running**
    2. **user running**
    - Diferem **no modo** como o processador **executa o código máquina**, existindo **mais instruções e diretivas disponíveis** no modo supervisor (`root`)
2. O estado **ready** é dividido em dois estados:
    1. **ready to run in memory:** O processo está pronto para ser executado/continuar a execução, estando guardado o seu estado em memória
    2. **preempted:** O processo que estava a ser executado foi **forçado a sair do processador** porque **existe um processo mais prioritário para ser executado**
    - Estes **estados são equivalentes** porque:
        - estão ambos **armazenado na memória principal**
        - quando um processo é `preempted` continua pronto a ser executado (não precisando de nenhuma informação de I/O)
        - Partilham a mesma fila (`queue`) de processos, logo são tratados de forma idêntica pelo OS
    - Quando um **processo do utilizador abandona o modo de supervisor** (corre com permissões `root`), **pode ser preempted**
3. A transição de `time-out` que existe no diagrama dos estados de um processo em UNIX é coberta pela transição `preempted`

50.7 Supervisor preempting

Tradicionalmente, a **execução** de um processo **em modo supervisor** (`root`) implicava que a execução do processo **não pudesse ser** interrompida, ou seja, o processo não pudesser ser **`preempted`**. Ou seja, o UNIX não permitia **real-time processing**

Nas novas versões o código está dividido em **regiões atómicas**, onde a **execução não pode ser interrompida** para garantir a **preservação de informação das estruturas de dados a manipular**. Fora das regiões atómicas é seguro interromper a execução do código

Cria-se assim uma nova transição, **`return`** `to kernel` entre os estados `preempted` e `kernel running`.

50.8 Unix – traditional login

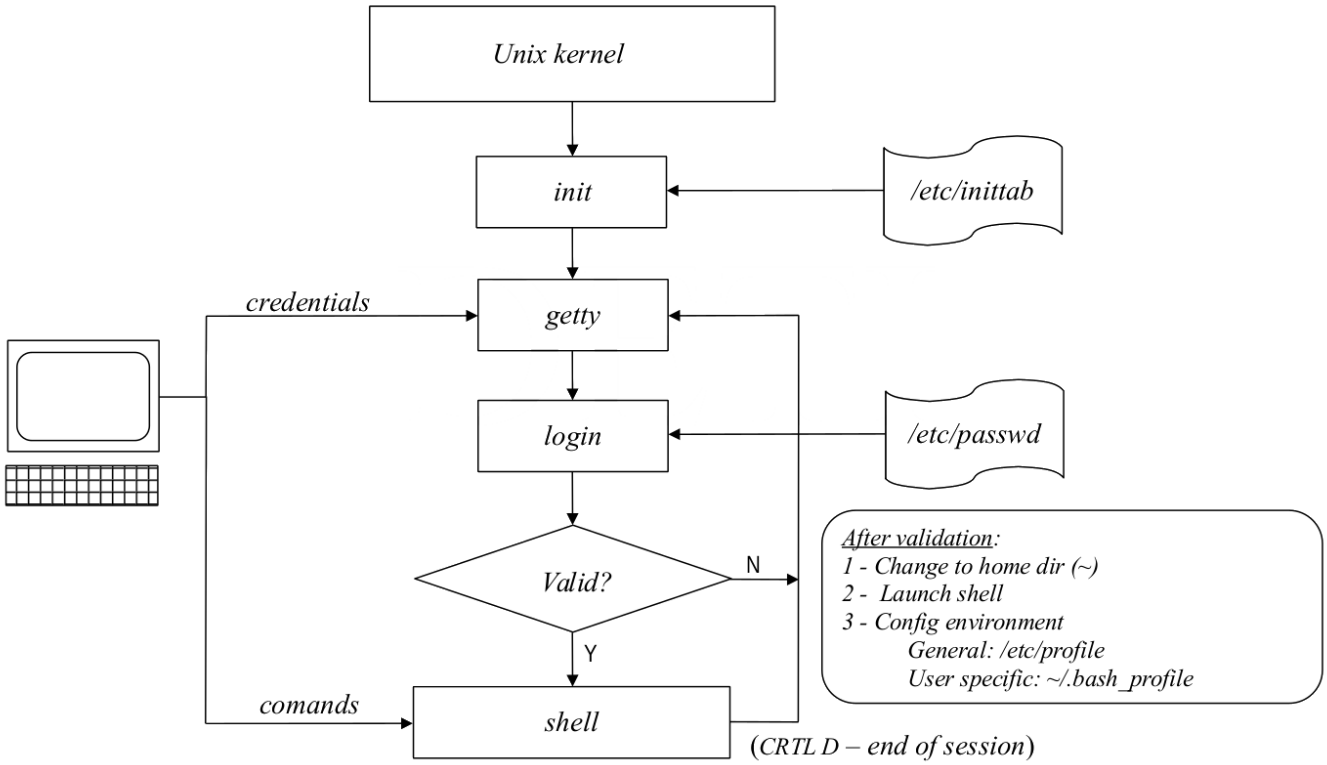

Figure 19: Diagrama do Login em Linux

50.9 Criação de Processos

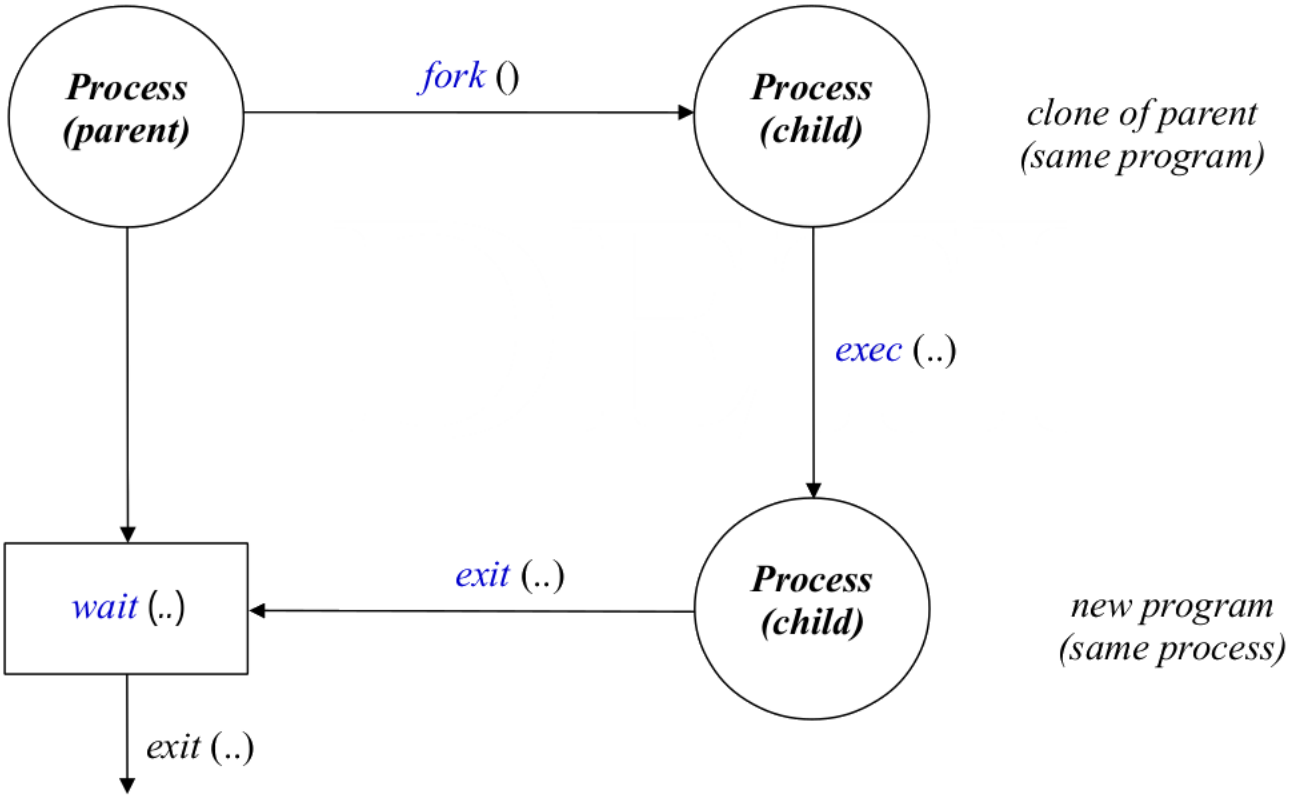

Figure 20: Criação de Processos

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("Before the fork:\n");
    printf(" PID = %d, PPID = %d.\n",
    getpid(), getppid());

    fork();

    printf("After the fork:\n");
    printf(" PID = %d, PPID = %d. Who am I?\n", getpid(), getppid());

    return EXIT_SUCCESS;
}
```

- **fork**: **clona** o processo existente, criando uma **réplica**

  - O estado de execução é igual, incluindo o PC *(Program Counter)*
  - O **mesmo programa** é executado em **processos diferentes**
  - Não existem garantias que o pai execute primeiro que o filho
    * Tudo depende do `time quantum` que o processo pai ocupa antes do `fork`
    * É um ambiente multiprogramado: os dois programas correm em **simultâneo no micro tempo**

- O **espaço de endereçamento** dos dois processos é **igual**
  - É seguida uma filosofia **copy-on-write**. Só é efetuada a cópia da página de memória se o **processo filho modificar** os conteúdos das variáveis

Existem variáveis diferentes:

- **PPID:** Parent Process ID
- **PID:** Process ID

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("Before the fork:\n");
    printf(" PID = %d, PPID = %d.\n",
    getpid(), getppid());

    int ret = fork();

    printf("After the fork:\n");
    printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);

    return EXIT_SUCCESS;
}
```

O retorno da instrução `fork` é diferente entre o processo pai e filho:

- pai: `PID` do filho
- filho: `0`

O retorno do `fork` pode ser usado como variável booleana, de modo a **distinguir o código a correr no filho e no pai**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("Before the fork:\n");
    printf(" PID = %d, PPID = %d.\n", getpid(), getppid());

    int ret = fork();
```

```
12
13      if (ret == 0)
14      {
15          printf("I'm the child:\n");
16          printf(" PID = %d, PPID = %d\n", getpid(), getppid());
17      }
18      else
19      {
20          printf("I'm the parent:\n");
21          printf(" PID = %d, PPID = %d\n", getpid(), getppid());
22      }
23
24      printf("After the fork:\n");
25      printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);
26
27      return EXIT_SUCCESS;
28  }
```

O `fork` por si só não possui grande interesse. O interesse é puder executar um programa diferente no filho.

- **`exec` system call:** Executar um programa diferente no processo filho
- **`wait` system call:** O pai esperar pela conclusão do programa a correr nos filhos

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <unistd.h>
5
6   int main(void)
7   {
8       /* check arguments */
9       if (argc != 2)
10      {
11          fprintf(stderr, "spawn <path to file>\n");
12          exit(EXIT_FAILURE);
13      }
14      char *aplic = argv[1];
15
16      printf("================================\n");
17
18      /* clone phase */
19      int pid;
20      if ((pid = fork()) < 0)
21      {
22          perror("Fail cloning process");
23          exit(EXIT_FAILURE);
24      }
25
26      /* exec and wait phases */
27      if (pid != 0) // only runs in parent process
28      {
29          int status;
```

```
30          while (wait(&status) == -1);
31          printf("================================\n");
32          printf("Process %d (child of %d)""
33           ends with status %d\n",
34          pid, getpid(), WEXITSTATUS(status));
35      }
36      else // this only runs in the child process
37      {
38          execl(aplic, aplic, NULL);
39          perror("Fail launching program");
40          exit(EXIT_FAILURE);
41      }
42  }
```

O `fork` pode **não ser bem sucedido**, ocorrendo um `fork failure`.

50.10 Execução de um programa em C/C++

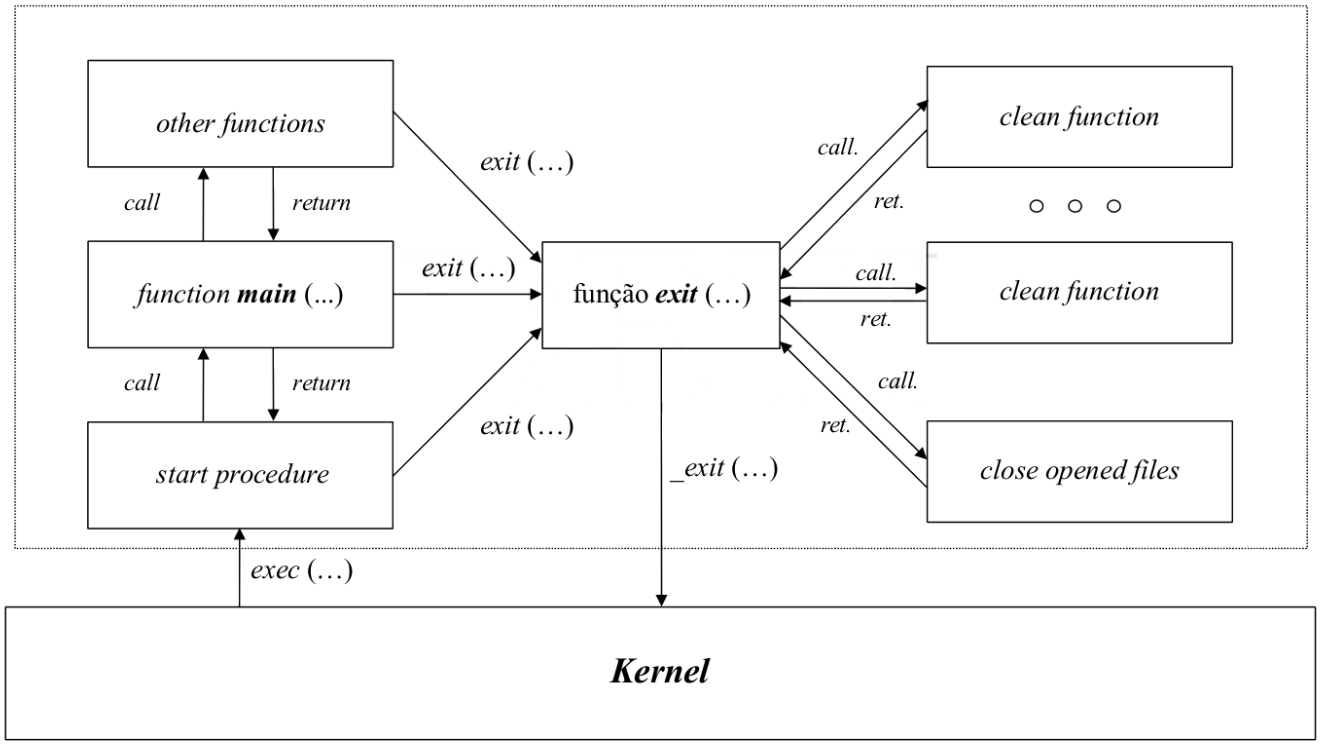

Figure 21: Execução de um programa em C/C++

- Em C/C++ o nome de uma função é um ponteiro para a sua função.
- Em C/C++ um include não inclui a biblioteca
  - Indica ao programa que vou usar uma função que tem esta assinatura
- **atexit:** Regista uma função para ser chamada no fim da execução normal do programa
  - São chamadas em ordem inversa ao seu registo

50.11 Argumentos passados pela linha de comandos e variáveis de ambiente

- **argv:** array de strings que representa um conjunto de parâmetros passados ao programa
  - **argv[0]** é a referência do programa
- **env** é um array de strings onde cada string representa uma variável do tipo: `name-value`
- **getenv** devolve o valor de uma variável definida no array **env**

50.12 Espaço de Endereçamento de um Processo em Linux

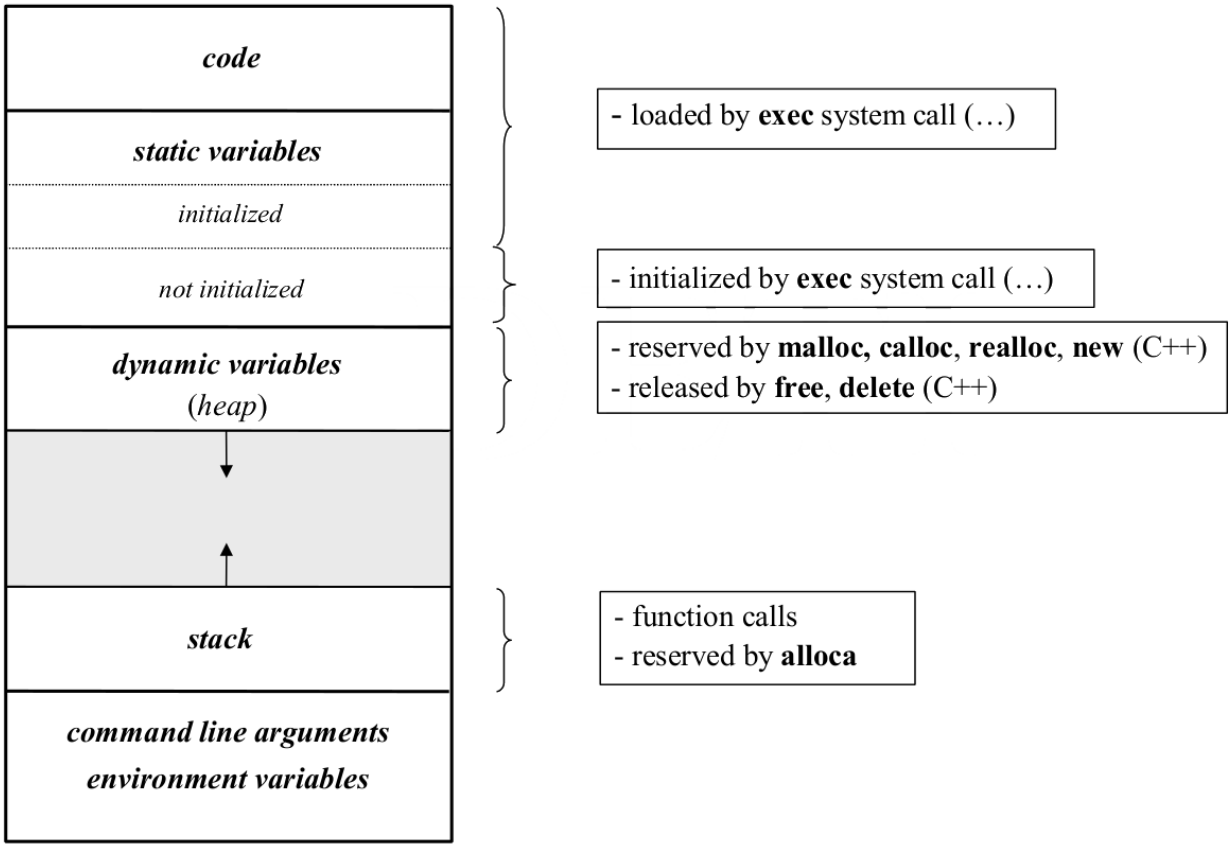

Figure 22: Espaço de endereçamento de um processo em Linux

- As variáveis que existem no processo pai também existem no processo filho (clonadas)
- As variáveis globais são comuns aos dois processos
- Os endereços das variáveis são todos iguais porque o espaço de endereçamento é igual (memória virtual)
- Cada processo tem as suas variáveis, residindo numa página de memória física diferente
- Quando o processo é clonado, o espaço de dados só é clonado quando um processo escreve numa variável, ou seja, após a modificação é que são efetuadas as cópias dos dados
- O programa acede a um endereço de memória virtual e depois existe hardware que trata de alocar esse endereço de memória de virtual num endereço de memória física

- Posso ter dois processos com memmorias virtuais distintas mas fisicamente estarem ligados *ao mesmo endereço de memória*
- Quando faço um `fork` não posso assumir que existem variáveis partilhadas entre os processos

50.12.1 Process Control Table

É um array de `process control block`, uma estrutura de dados mantida pelo sistema operativo para guardar a informação relativa todos os processos.

O `process controlo block` é usado para guardar a informação relativa a apenas um processo, possuindo os campos:

- `identification`: id do processo, processo-pai, utilizador e grupo de utilizadores a que pertence
- `address space`: endereço de memória/swap onde se encontra:
  - código
  - dados
  - stack
- `processo state`: valor dos registos internos (incluindo o PC e o `stack pointer`) quando ocorre o switching entre processos
- `I/O context`: canais de comunicação e respetivos buffers que o processo tem associados a si
- `state`: estado de execução, prioridade e eventos
- `statistic`: *start time*, *CPU time*

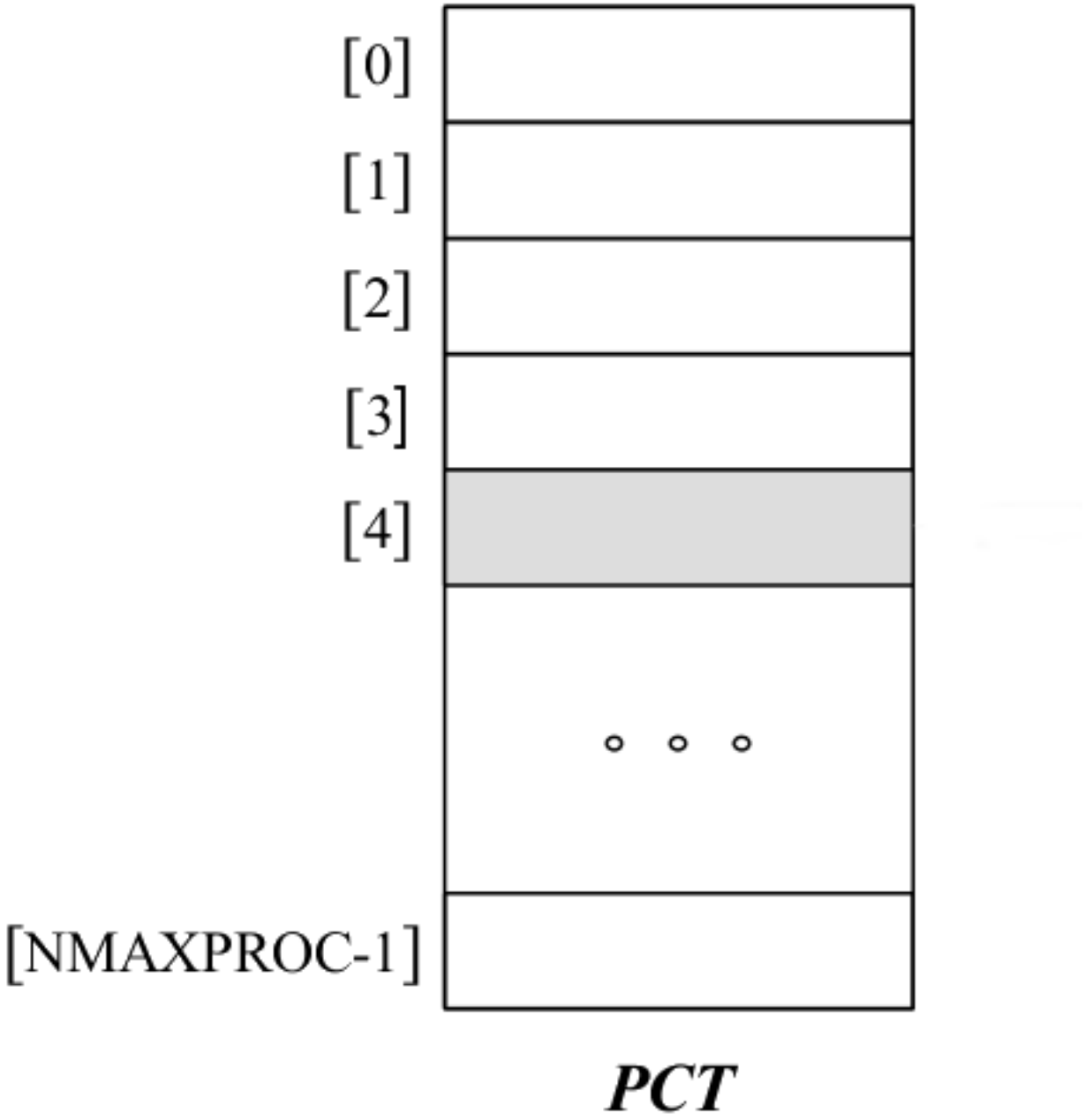

Figure 23: Process Control Table

51 Threads

Num sistema operativo tradicional, um **processo** inclui:

- um **espaço de endereçamento**
- um **conjunto de canais de comunicação** com dispositivos de I/O

- uma única **`thread` de controlo** que:
  - incorpora os **registos do processador** (incluindo o PC)
  - **stack**

Existem duas stacks no sistema operativo:

- `user stack`: cada `processo`/`thread` possui a sua (em memória virtual e corre com privilégios do `user`)
- `system stack`: global a todo o sistema operativo (no `kernel`)

Podendo estes dois componentes serem **geridos de forma independente**.

Visto que uma `thread` é apenas um **componente de execução** dentro de um processo, várias `threads` **independentes** podem coexisitir no mesmo processo, **partilhando** o mesmo **espaço de endereçamento** e o mesmo contexto de **acesso aos dispositivos de I/O**. Isto é **`multithreading`**.

Na prática, as `threads` podem ser vistas como *light weight processes*

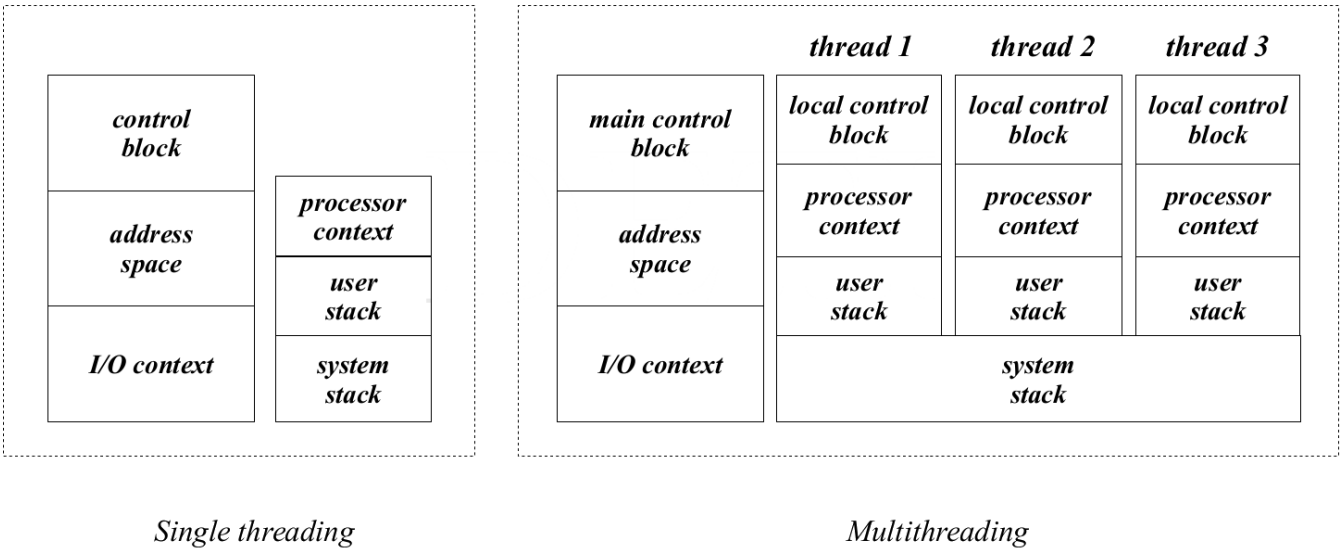

Figure 24: Single threading vs Multithreading

O controlo passa a ser centralizado na `thread` principal que gere o processo. A `user stack`, o **contexto de execução do processador** passa a ser dividido por todas as `threads`.

51.1 Diagrama de Estados de uma thread

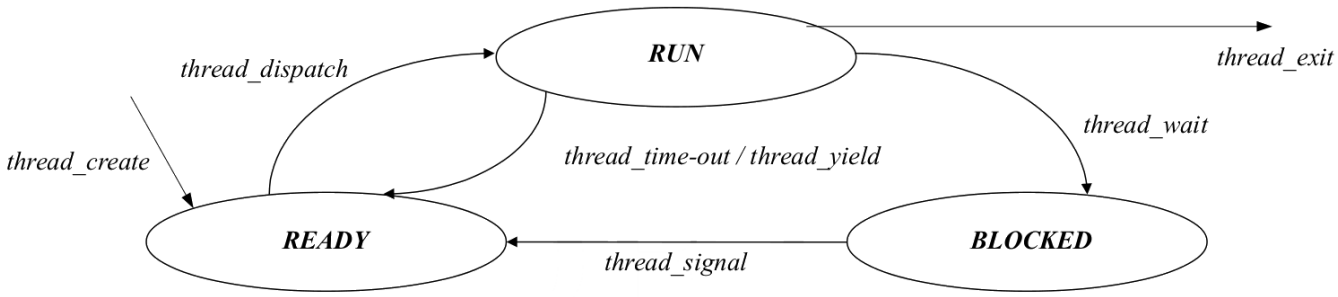

Figure 25: Diagrama de estados de uma thread

O diagrama de estados de um `thread` é mais simplificado do que o de um processo, porque só são "necessários" os estados que interagem **diretamente com o processador**:

```
- 'run'
- 'ready'
- 'blocked'
```

Os estados `suspend-ready` e `suspended-blocked` estão relacionados com o **espaço de endereçamento** do **processo** e com a zona onde estes dados estão guardados, dizendo respeito ao **processo e não à thread**

Os estado **`new`** e `terminated`não estão presentes, porque a a gestão do ambiente multiprogramado prende-se com a restrição do número de `threads` que um processo pode ter, logo dizem respeito ao processo

51.2 Vantagens de Multithreading

- **facilita a implementação** (em certas aplicações):
  - Existem aplicações em que **decompor a solução** num conjunto de `threads` que correm paralelamente facilita a implementação
  - Como o `address space` e o `I/O context` são partilhados por todas as `threads`, `multithreading` favorece esta decomposição
- **melhor utilização dos recursos**
  - A criação, destruição e `switching` é mais eficiente usando `threads` em vez de processos
- **melhor performance**
  - Em aplicações `I/O driven`, `multithreading` permite que **várias atividades se sobreponham**, **aumentando a rapidez** da sua execução
- **`multiprocessing`**
  - É possível **paralelismo em tempo real** se o processador possuir **múltiplos CPUs**

51.3 Estrutura de um programa multithreaded

- Cada `thread` está tipicamente associada com a execução de uma função que implementa alguma atividade em específico
- A **comunicação entre `threads`** é efetuada através da estrutura de dados do **processo**, que é vista pelas `threads` como uma estrutura de dados global
- o **programa principal** também é uma `thread`
  - A 1ª a ser criada
  - Por norma a última a ser destruída

51.4 Implementação de Multithreading

`user level threads`:

- Implementadas por uma biblioteca
  - Suporta a criação e gestão das `threads` sem intervenção do `kernel`
- Correm com permissões do utilizador
- Solução versátil e portável
- Quando uma `thread` executa uma `system call` bloqueante, **todo o processo bloqueia** (o *kernel* só "vê" o processo)
- Quando passo variáveis a `threads`, elas têm de ser estáticas ou dinâmicas

`kernel level threads`

- As `threads` são implementadas diretamente ao nível do kernel
- Menos versáteis e portáteis
- Quando uma `thread` executa uma `system call` bloqueante, **outra *thread* pode entrar em execução**

51.4.1 Libraria pthread

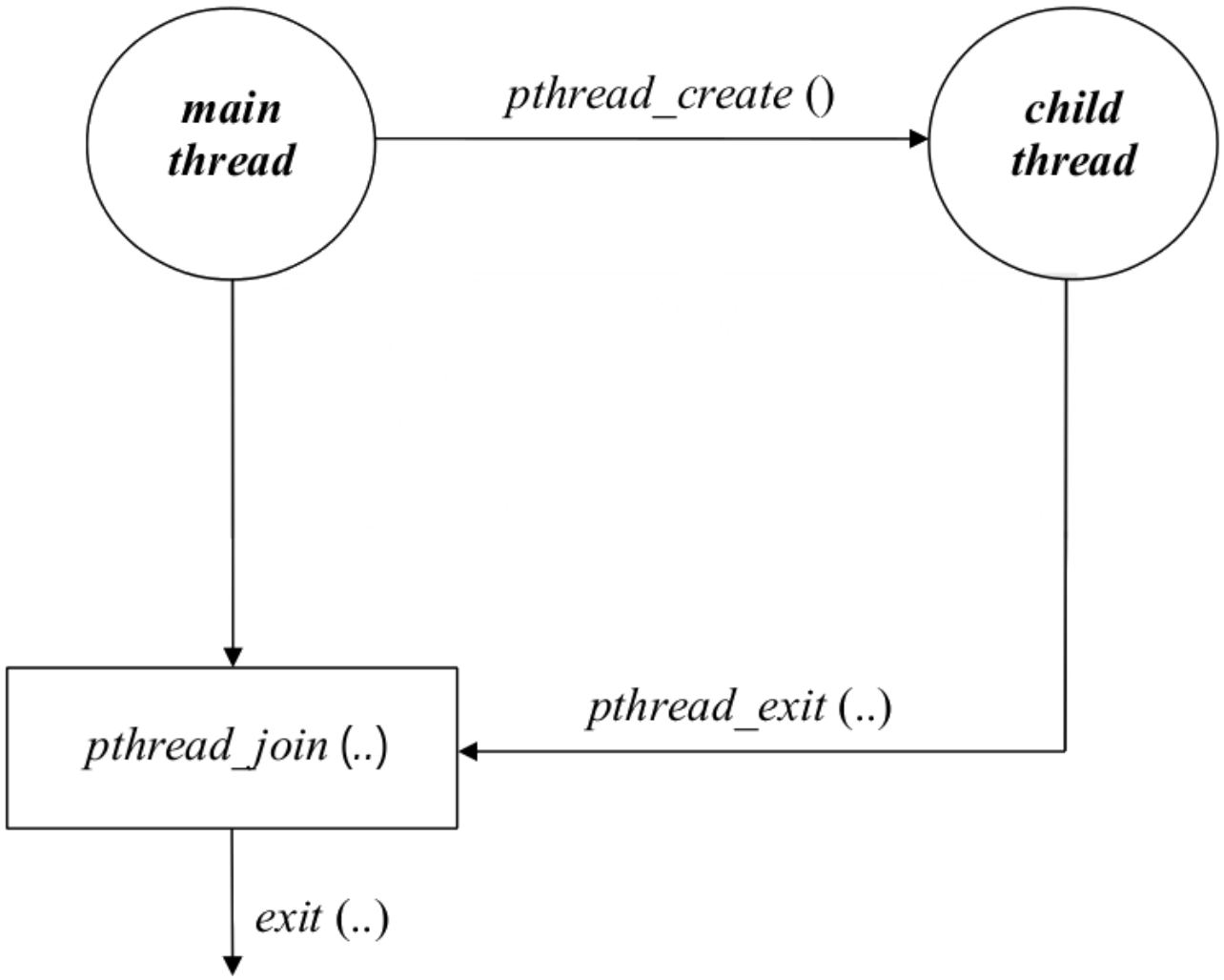

Figure 26: Exemplo do uso da libraria pthread

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

/* return status */
int status;

/* child thread */
void *threadChild (void *par)
{
    printf (”I’m the child thread!\n”);
    status = EXIT_SUCCESS;
    pthread_exit (&status);
}

```

```c
/* main thread */
int main (int argc, char *argv[])
{
    /* launching the child thread */
    pthread_t thr;
    if (pthread_create (&thr, NULL, threadChild, NULL) != 0)
    {
        perror ("Fail launching thread");
        return EXIT_FAILURE;
    }

    /* waits for child termination */
    if (pthread_join (thr, NULL) != 0)
    {
        perror ("Fail joining child");
        return EXIT_FAILURE;
    }

    printf ("Child ends; status %d.\n", status);
    return EXIT_SUCCESS;
}
```

51.5 Threads em Linux

2 system calls para criar processos filhos:

- fork:
  - cria uma novo processo que é uma **cópia integral** do processo atual
  - o address space é I/O context é duplicado
- clone:
  - cria um novo processo que pode partilhar elementos com o pai
  - Podem ser partilhados
    * espaço de endereçamento
    * tabela de file descriptors
    * tabela de signal handlers
  - O processo filho executa uma dada função

Do ponto de vista do kernel, processos e threads são **tratados de forma semelhante**

Threads do **mesmo processo** foma um thread group e possuem o **mesmo** thread group identifier (TGID). Este é o valor retornado pela função getpid() quando aplicada a um grupo de threads

As várias threads podem ser distinguidas dentro de um **grupo de threads** pelo seu unique thread identifier (TID). É o valor retornado pela função gettid()

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

```c
#include <unistd.h>
#include <sys/types.h>

pid_t gettid()
{
    return syscall(SYS_gettid);
}

/* child thread */
int status;
void *threadChild (void *par)
{
    /* There is no glibc wrapper, so it was to be called
     * indirectly through a system call
     */

    printf ("Child: PPID: %d, PID: %d, TID: %d\n", getppid(), getpid(), gettid());
    status = EXIT_SUCCESS;
    pthread_exit (&status);
}
```

O `TID` da *main thread* é a mesma que o `PID` do processo, **porque são a mesma entidade**.

Para efetuar a compilação, tenho de indicar que a biblioteca pthread tem de ser usada na linkagem:

```
g++ -o x thread.cpp -pthread
```

52 Process Switching

Revisitando a o diagrama de estados de um processador `multithreading`

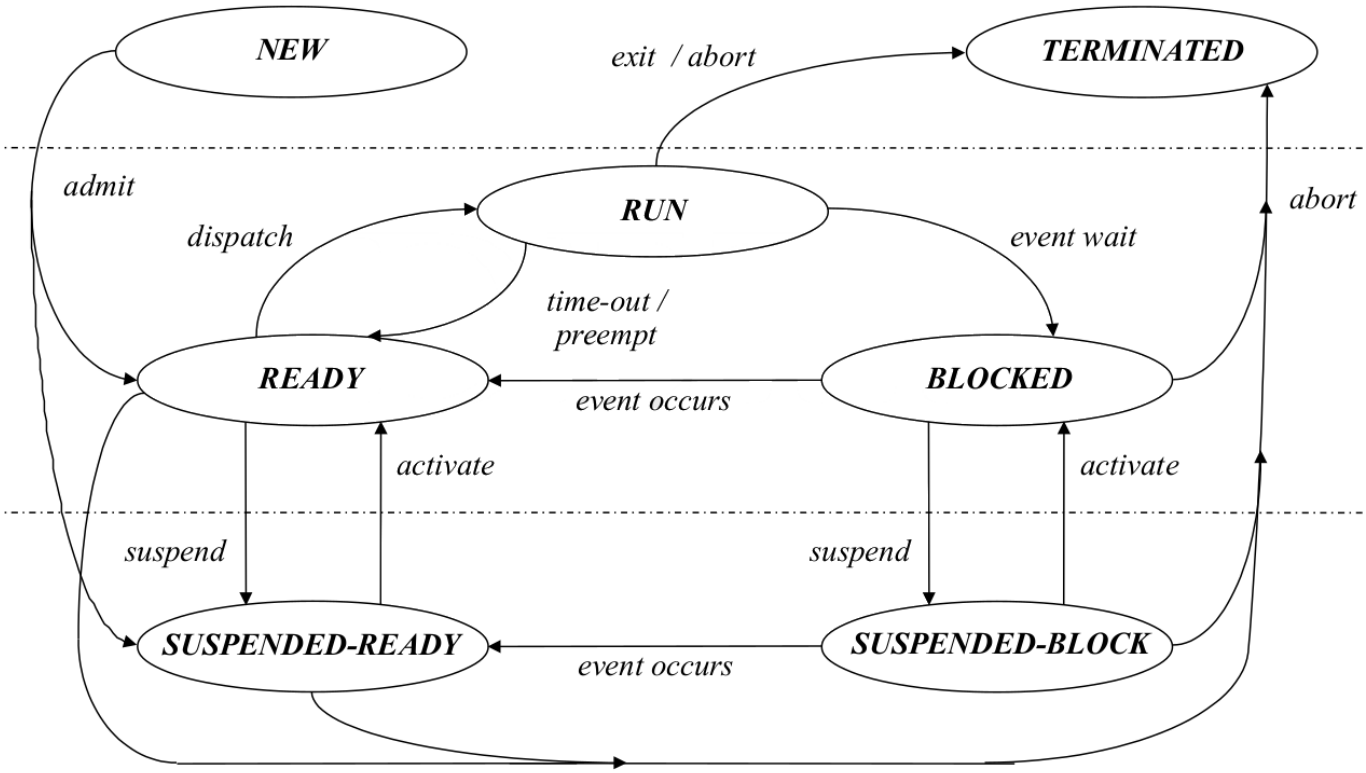

Figure 27: Diagrama de estados completo para um processador `multithreading`

Os processadores atuais possuem **dois modos de funcionamento:**

1. `supervisor mode`
   - Todas as intruções podem ser executadas
   - É um modo **priveligiado**, **reservado para o sistema operativo**
   - O modo em que o **sistema operativo devia funcionar**, para garantir que pode aceder a todas as funcionalidades do processador
2. `user mode`
   - Só uma **secção reduzida do instruction set** é que pode ser executada
   - Instruções de I/O e outras que modifiquem os registos de controlo não são executadas em `user mode`
   - É o **modo normal de operação**

A **troca entre os dois modos de operação**, `switching`, só é possível através de `exceções`. Uma `exceção` pode ser causada por:

- Interrupção de um dispositivo de I/O
- Instrução ilegal
  - divisão por zero
  - bus error
  - …
- trap instruction (aka interrupção por *software*)

As **funções do `kernel`**, incluindo as `system calls` só podem ser lançadas por:

- **hardware** $\Longrightarrow$ `interrupção`
- **traps** $\Longrightarrow$ `intreeupção por software`

O ambiente de operação nestas condições é denominado de `exception handling`

52.1 Exception Handling

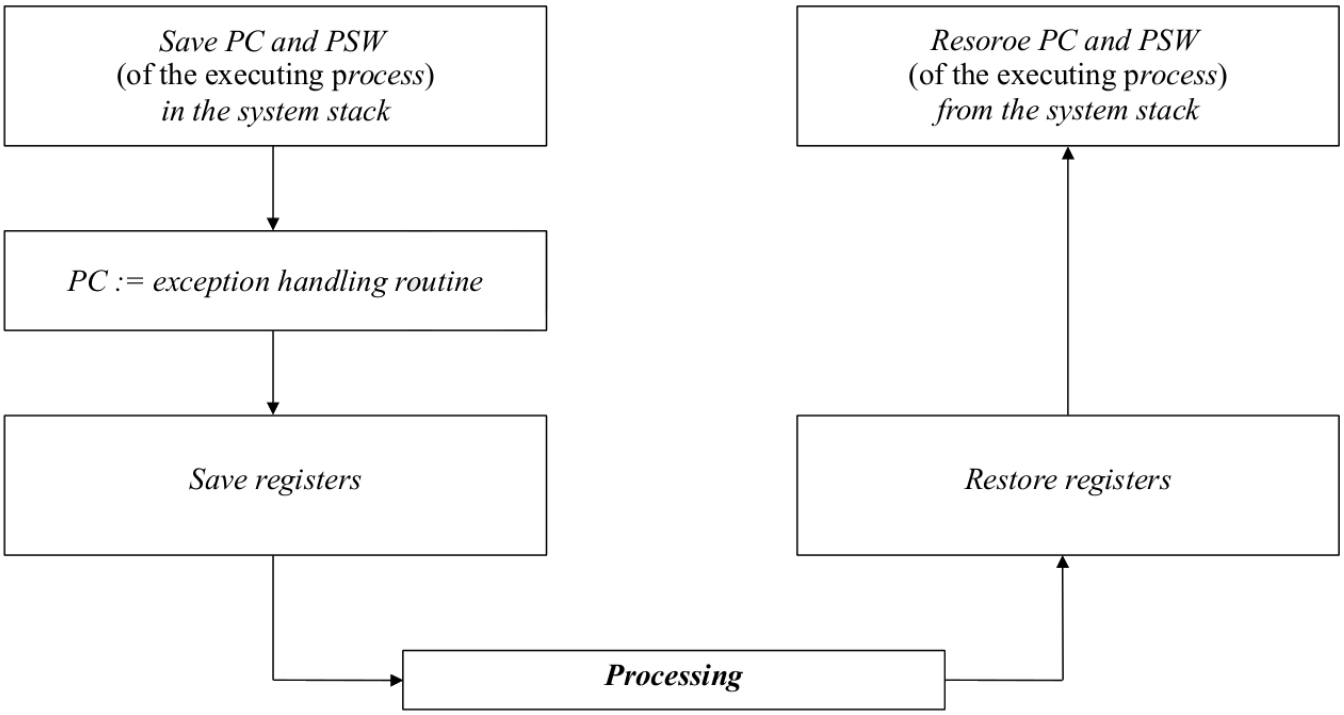

Figure 28: Algoritmo a seguir para tratar de exeções normais

A **troca do contexto de execução** é feita guardando o estado dos registos PC e PSW na stack do sistema, saltando para a rotina de interrupção e em seguida salvaguardando os registos que a rotina de tratamento da exceção vai precisar de modificar. No fim, os valores dos registos são restaurados e o programa resume a sua execução

52.2 Processing a process switching

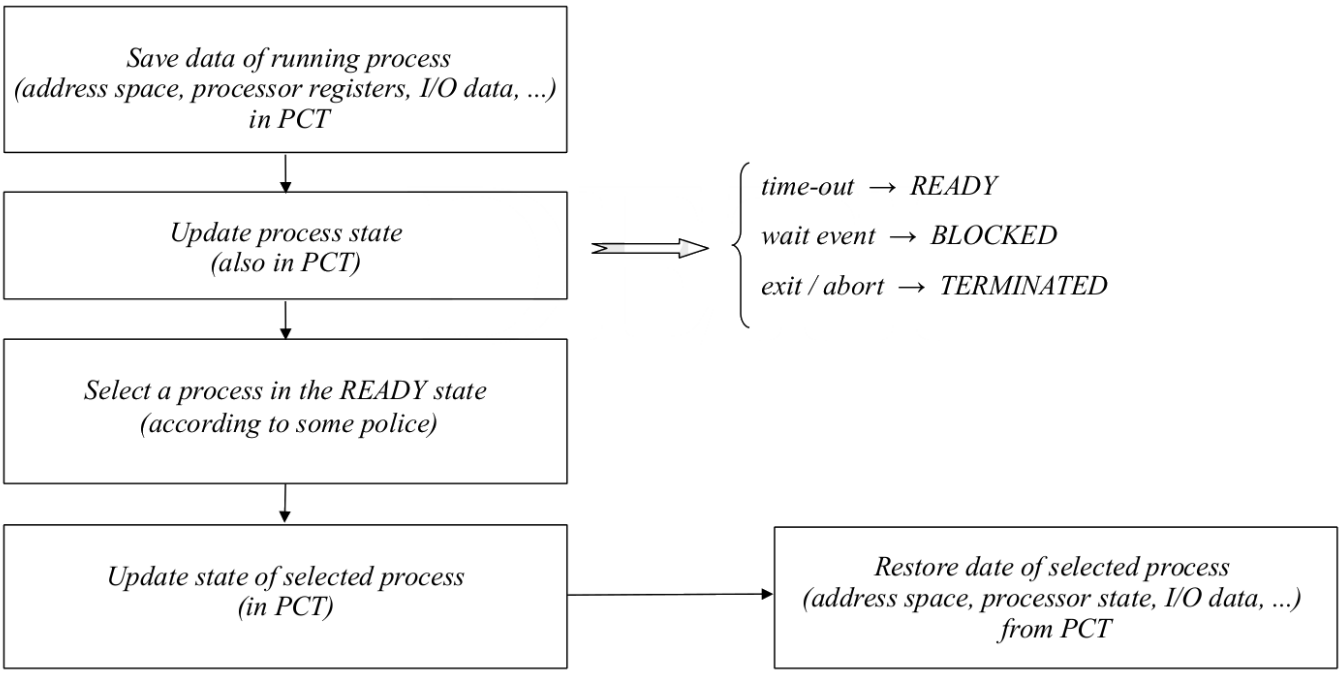

Figure 29: Algoritmo a seguir para efetuar uma process switching

O algoritmo é bastante parecido com o tratamento de exceções:

1. Salvaguardar todos os dados relacioandos com o processo atual
2. Efetuar a troca para um novo processo
3. Correr esse novo processo
4. Restaurar os dados e a execução do processo anterior

53 Processor Scheduling

A execução de um processo é uma sequência alternada de períodos de:

- `CPU burst`, causado pela execuçao de intruções do CPU
- `I/O burst`, causados pela espera do resultado de pedidos a dispositivos de I/O

O processo pode então ser classificado como:

- `I/O bound` se possuir muitos e curtos `CPU bursts`
- `CPU bound` se possuir poucos e longos `CPU bursts`

O objetivo da **multiprogramação** é obter vantagem dos períodos de `I/O burst` para permitir outros processos terem acesso ao processador. A componente do sistema responsável por esta gestão é o `scheduler`.

A funcionalidade principal do `scheduler` é decidir da `poll` de processos prontos para serem executados que coexistem no sistema:

- quais é que devem ser executados?

- quando?
- por quanto tempo?
- porque ordem?

53.1 Scheduler

Revisitando o diagrama de estados do processador, identificamos três `schedulers`

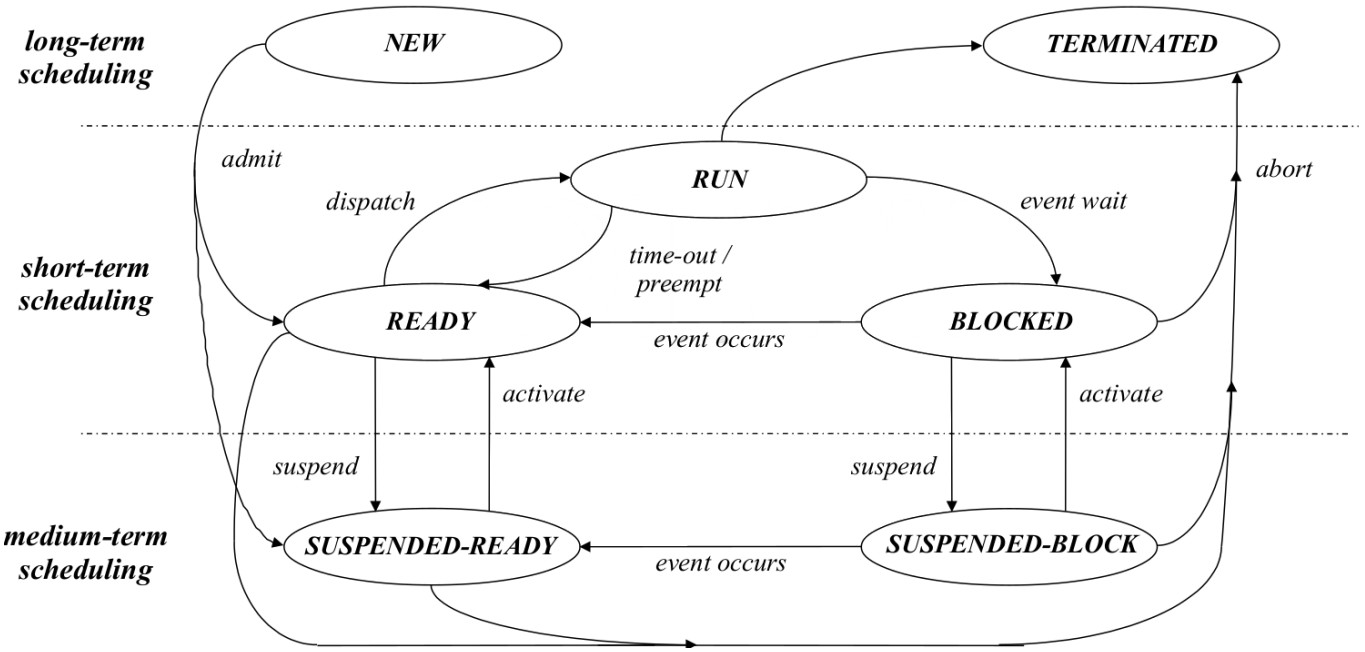

Figure 30: Identificação dos diferentes tipos de schedulers no diagrama de estados dos processos

53.1.1 Long-Term Scheduling

Determina que **programas são admitidos para serem processados**:

- Controla o **grau de multiprogramação** do sistema
- Se um programa do utilizador ou `job` for aceite, torna-se um **processo** e é adicionado à ***queue* de processos `ready` em fila de espera**
  - Em princípio é adicionado à `queue` do **`short`**`-term scheduler`
  - mas também é possível que seja adicionada à `queue` do `medium-term scheduler`
- Pode colocar processos em `suspended ready`, libertando quer a memória quer a fila de processos

53.1.2 Medium Term Scheduling

Gere a `swapping area`

- As decisões de `swap-in` são **controladas pelo grau de multiprogramação**
- As decisões de `swap-in` são **condicionadas pela gestão de memória**

53.1.3 Short-Term Scheduling

Decide qual o **próximo processo a executar**

- É invocado quando existe um evento que:
  - **bloqueia o processo atual**
  - **permite que este seja `preempted`**
- Eventos possíveis são:
  - interrupção de relógio
  - interrupção de I/O
  - `system calls`
  - signal (e.g. através de semáforos)

53.2 Critérios de Scheduling

53.2.1 User oriented

Turnaround Time:

- Intervalo de de tempo entre a submissão de um processo até à sua conclusão
- Inclui:
  - Tempo de execução enquanto o processo tem a posse do CPU
  - Tempo dispendido à espera pelos recursos que precisa (inclui o processador)
- Deve ser minimizado em sistemas `batch`
- É a medida apropriada para um `batch job`

Waiting Time:

- Soma de todos os períodos de tempo em que o processo esteve à espera de ser colocado no estado `ready`
- Deve ser minimizado

Response Time:

- Intervalo de tempo que decorre desde a submissão de um pedido até a resposta começa a ser produzida
- Medida apropriada para sistemas/processos interativo
- Deve ser minimizada para este tipo de sistemas/processos
- O número de processos interativos deve ser máximizado desde que seja garantido um tempo de resposta aceitável

Deadlines:

- Tempo necessário para um processo terminar a sua execução
- Usado em sistemas de tempo real
- A percentagem de `deadlines` atingidas deve ser maximizada, mesmo que isso implique subordinar/reduzir a importância de outras objetivos/parâmetros do sistema

Predictability:

- Quantiza o impacto da carga (de processos) no tempo de resposta dos sistema
- Idealmente, um `job` deve correr no **mesmo intervalo de tempo** e gastar os **mesmos recursos de sistema** independentemente da carga que o sistema possui

53.2.2 System oriented

Fairness:

- Igualdade de tratamento entre todos os processos
- Na ausência de diretivas que condicionem os processos a atender, deve ser efetuada um gestão e partilha justa dos recursos, onde todos os processos são tratados de forma equitativa
- Nenhum processo pode sofrer de `starvation`

Throughput:

- Medida do número de processos completados por unidade de tempo ("taxa de transferência" de processos)
- Mede a quantidade de trabalho a ser executada pelos processos
- Deve ser máximizado
- Depende do tamanho dos processos e da **política de escalonamento**

Processor Utilization:

- Mede a percentagem que o processador está ocupado
- Deve ser maximizada, especialmente em sistemas onde predomina a partilha do processador

Enforcing Priorities:

- Os processos de **maior prioridade** devem ser sempre favorecidos em detrimento de processos menos prioritários

É **impossível favorecer todos os critérios em simultâneo**

Os **critérios a favorecer** dependem da **aplicação específica**

53.3 Preemption & Non-Preemption

Non-preemptive scheduling:

- O processo mantêm o processador até este ser bloqueado ou terminar
- As **transições são sempre por `time-out`**
- Não existe `preempt`
- Típico de sistemas `batch`
  - Não existem deadlines nem limitações temporais restritas a cumprir

Preemptive Scheduling:

- O processo pode **perder o processador devido a eventos externos**
  - esgotou o seu `time-quantum`
  - um processo **mais prioritário** está `ready`
  - Tipico de **sistemas interativos**
    * É preciso garantir que a resposta ocorre em intervalos de tempo limitados
    * É preciso "simular" a ideia de paralelismo no `macro-tempo`
  - Sistemas em tempo real são `preemptive` porque existem `deadlines` **restritas** que precisam de ser cumpridas
  - Nestas situações é importante que um **evento externo** tenha capacidade de libertar o processador

53.4 Scheduling

53.4.1 Favouring Fearness

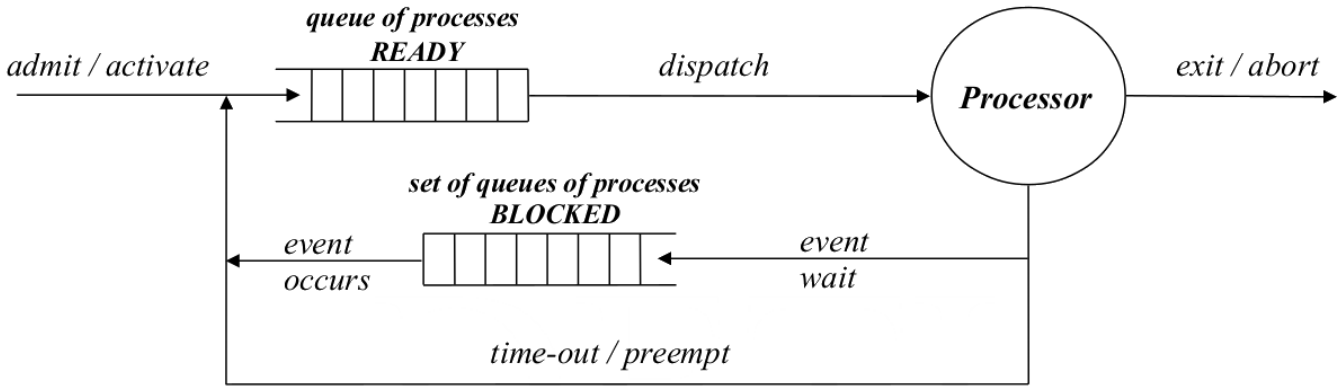

Figure 31: Espaço de endereçamento de um processo em Linux

Todos os processos **são iguais** e são atendidos por **ordem de chegada**

- É implementado usando **FIFOs**
- Pode existir mais do que um processo à espera de eventos externos
- Existe uma fila de espera para cada evento
- Fácil de implementar
- Favorece processos `CPU-bound` em detrimento de processos `I/O-bound`
  - Só necessitam de acesso ao processador, não de recursos externos
  - Se for a vez de um processo `I/O-bound` ser atendido e não possuir os recursos de I/O que precisa tem de voltar para a fila
- Em **sistemas interativos**, o `time-quantum` deve ser escolhido cuidadosamente para obter um bom compromisso entre `fairness` e `response time`

Em função do scheduling pode ser definido como:

- `non-preemptive scheduling` $\implies$ `first come, first-served` (FCFS)
- `preemptive scheduling` $\implies$ `round robin`

53.4.2 Priorities

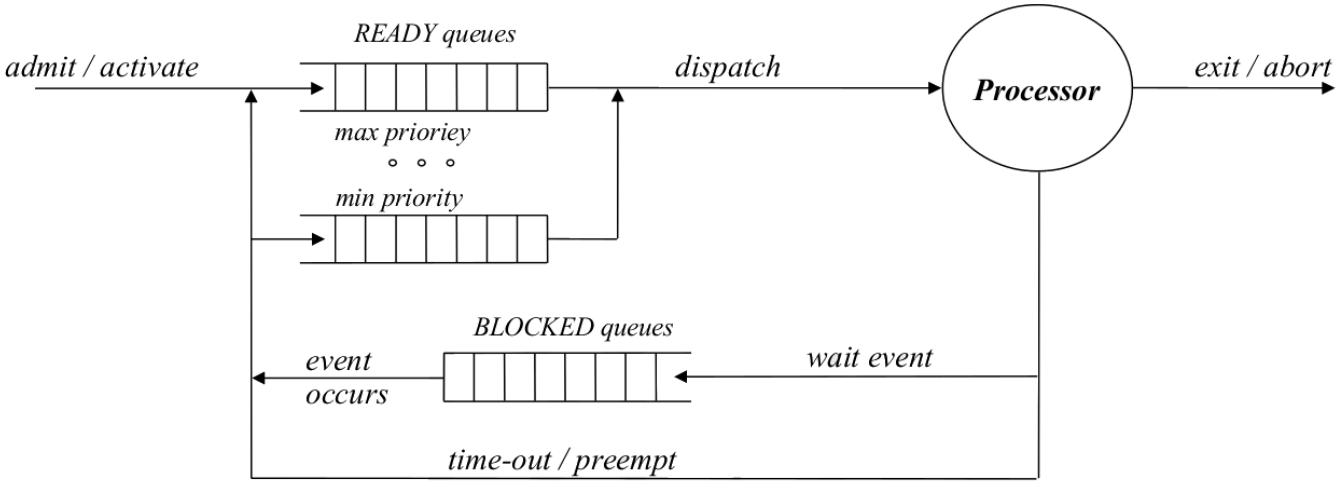

Figure 32: Espaço de endereçamento de um processo em Linux

Segue o princípio de que atribuir a mesma importância a todos os processos pode ser uma solução errada. Um sistema injusto *per se* não é necessariamente mau.

- A **minimização do tempo de resposta** (`response time`) exigue que os processos `I/O-bound` sejam **priveligiados**
- Em **sistemas de tempo real**, os processos associados a **eventos/alarmes** e **ações do sistema operativo** sofrem de várias **limitações e exigências temporais**

Para resolver este problema os processos são **agrupados** em grupos de **diferentes prioridades**

- Processos de maior prioridade são executados primeiros
- Processos de menor prioridade podem sofrer `starvation`

Prioridades Estáticas

As prio ridades a atribuir a cada processo são determinadas *a priori* de forma **determinística**

- Os processos são **agrupados em classes de prioridade fixa**, de acordo com a sua importância relativa
- Existe risco de os processos menos prioritários sofrerem `starvation`
  - Mas se um **processo de baixa prioridade não é executado** é porque o sistema foi **mal dimensionado**
- É o sistema de `scheduling` mais injusto
- É usado em sistemas de tempo real, para garantir que os processos que são críticos são sempre executados

Alternativamente, pode se fazer:

1. Quando um processo é criado, é lhe **atribuído um dado nível de prioridade**
2. Em `time-out` a prioridade do processo é **decrementada**
3. Na ocorrência de um `wait event` a prioridade é **incrementada**
4. Quando o valor de **prioridade atinge um mínimo**, o valor da prioridade sofre um `reset`
   - É colocada no valor inicial, garantindo que o processo é executado

Previnem-se as situações de `starvation` impedindo que o processo não acaba por ficar com uma prioridade tão baixa que nunca mais consegue ganhar acesso

Prioridades Dinãmicas

- As classes de prioridades estão definidas de forma funcional *a priori*
- A mudança de um processo de classe é efetuada com base na utilização última janela de execução temporal que foi atribuída ao processo

Por exemplo:

- **Prioridade 1:** `terminais`
  - Um processo entra nesta categoria quando se efetua a transição `event occurs` (evento de escrita/leitura de um periférico) quando estava à espera de dados do `standard input device`
- **Prioridade 2:** `generic I/O`
  - Um processo entra nesta categoria quando efetua a transição `event occurs` se estava à espera de dados de **outro tipo de input device** que não o `stdin`
- **Prioridade 3:** `small time quantum`
  - Um processo entra nesta classe quando ocorre um `time-out`
- **Prioridade 4:** `large time quantum`
  - Um processo entra nesta classe após um sucessivo número de `time-outs`
  - São claramente processos `CPU-bound` e o objetivo é atribuir-lhes janelas de execução com grande `time quantum` , mas menos vezes

Shortest job first (SJF) / Shortest process next (SPN)

Em sistemas `batch`, o `turnaround time` deve ser minimizado.

Se forem conhecidas **estimativas do tempo de execução** *a priori*, é possível estabelecer uma **ordem de execução** dos processos que **minimizam o tempo de `turnaround` médio** para um dado grupo de processos

Assumindo que temos N `jobs` e que o tempo de execução de cada um deles é te_n, com $n = 1, 2, ..., N$. O `average turnaround time` é:

$$t_m = te_1 + \frac{N-1}{N} \cdot te_2 + ... + \frac{1}{N} \cdot te_N$$

onde t_m é o `turnaround time` mínimo se os `jobs` forem sorteados por ordem ascendente de tempo de execução (estimado)

Para **sistemas interativos**, podemos usar um sistema semelhante:

- Estimamos a taxa de ocupação da próxima janela de execução baseada na taxa de ocupação das janelas temporais passadas
- Atribuímos o processador ao processo cuja estimativa for a **mais baixa**

Considerando fe_1 como sendo a **estimativa da taxa de ocupação** da primeira janela temporal atribuída a um processo e f_1 a fração de tempo efetivamente ocupada:

- A estimativa da segunda fração de tempo necessária é

$$fe_2 = a \cdot fe_1 + (1-a) \cdot f_1$$

- A estimativa da e-nésima fração de tempo necessária é:

$$fe_N = a \cdot fe_{N-1} + (1-a) \cdot f_{N-1}$$

  Ou alternativamente:

$$a^{N-1} \cdot fe_1 + a^{N-2} \cdot (1-a) \cdot fe_2 + a \cdot (1-a) \cdot fe_{N-2} + (1-a) \cdot fe_{N-1}$$

Com $a \in [0, 1]$, onde a é um coeficiente que representa o peso que a história passada de execução do processo influencia a estimativa do presente

Esta alternativa levanta o problema que que processos `CPU-bound` podem sofrer de `starvation`. Este problema pode ser resolvido contabilizando o tempo que um processo está em espera (`aging`) enquanto está na fila de processos `ready`

Normalizando esse tempo em função do período de execução e denominando-o R, a **prioridade** de um processo pode ser dada por:

$$p = \frac{1 + b \cdot R}{fe_N}$$

onde b é o coeficiente que **controla o peso do aging** na fila de espera dos processos `ready`

53.5 Scheduling Policies

53.5.1 First Come, First Serve (FCFS)

Também conhecido como `First In First Out` (FIFO). O processo mais antigo na fila de espera dos processos `ready` é o primeiro a ser selecionado.

- `Non-preemptive` (em sentido estrito), podendo ser combinado com um esquema de prioridades baixo
- Favorece processos `CPU-bound` em detrimento de processos `I/O-bound`
- Pode resultar num **mau uso** do processador e dos dispositivos de I/O
- Pode ser utilizado com **low priority schemas**

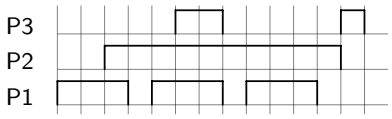

Figure 33: Problema de Scheduling

Usando uma política de `first come first serve`, o resultado do scheduling do processador é:

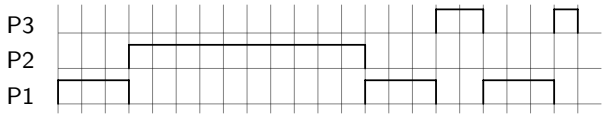

Figure 34: Política FCFS

- O P1 começa a usar o CPU.
- Como é um sistema FCFS, o processo 1 só larga o CPU passado 3 ciclos.
- O processo P2 é o processo seguinte na fila `ready`, e ocupa o CPU durante 10 ciclos.
- Quando P2 termina, P1 é o processo que está à mais tempo à espera, sendo ele que é executado

- Quando P2 abandona voluntariamente o CPU, o processo P1 corre os seus primeiros dois ciclos
- Quando P3 liberta o CPU, o processo P1 termina os últimos 3 ciclos que precisa
- Quando P3 liberta o CPU, o processo P1 como é `I/O-bound` e precisa de 5 ciclos para o dispositivo estar pronto fica mais dois ciclos à espera para puder terminar executando o seu último ciclo

53.5.2 Round-Robin

- `Preemptive`
  - O `scheduler` efetua a gestão baseado num `clock`
  - A cada processo é atribuído um `time-quantum` máximo antes de ser `preempted`
- O processo **mais antigo** em `ready` é o **primeiro a ser selecionado**
  - não são consideradas prioridades
- Efetivo em sistemas `time sharing` com objetivos globais e sistemas que processem transações
- **Favorece** `CPU-bound` em detrimento de processos `I/O-bound`
- Pode resultar num **mau uso de dispositivos I/O**

Na escolha/otimização do `time quantum` existe um **tradeoff:**

- **tempos muito curtos** favorecem a execução de **processos pequenos**
  - estes processos vão ser executados **rapidamente**
- **tempos muito curtos** obrigam a `processing overheads` devido ao `process switching` **intensivo**

Para os processos apresentados acima, o diagrama temporal de utilização do processador, para um `time-quantum` de 3 ciclos é:

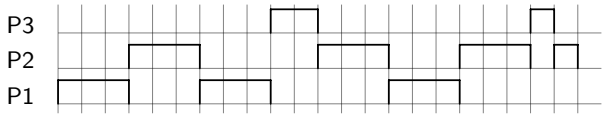

Figure 35: Política Round-Robin

A história de processos em `ready` em fila de espera é: 2, 1, 3, 2, 1, 2, 3, 1

53.5.3 Shortest Process Next (SPN) ou Shortest Job First (SJF)

- `Non-preemptive`
- O process com o `shortest CPU burst time` (menor tempo espectável de utilização do CPU) é o **próximo a ser selecionado**
  - Se vários processos tem o **mesmo tempo de execução** é usado FCFS para desempatar
- Existe um **risco de** **`starvation`** para grandes processos
  - o seu acesso ao CPU pode ser **sucessivamente adiado** se existir "forem existindo" processos com **tempo de execução menor**
- Normalmente é usado em escalonamento de logo prazo, **`long`**`-term scheduling` em sistemas `batch`, porque os utilizadores esperam estimar com precisão o tempo máximo que o processo necessita para ser executado

53.5.4 Linux

No Linux existem 3 classes de prioridades:

1. **FIFO**, `SCHED_FIFO`
   - `real-time threads`, com politíca de prioridades
   - uma `thread` em execução é `preempted` apenas se um processo de **mais alta prioridade da mesma classe** transita para o estado `ready`
   - uma `thread` em execução pode **voluntariamente abandonar o processador**, executando a primitiva `sched_yeld`
   - dentro da mesma classe de prioridade a política escolhida é `First Come, First Serve` (FCFS)
   - Só o `root` é que pode lançar processos em modo FIFO
2. **Round-Robin real time threads**, `SCHED_RR`
   - `threads` com prioridades com necessidades de execução em tempo real
   - Processos nesta classe de prioridades são `preempted` se o seu `time-quantum` termina
3. **Non real time threads**, `SCHED_OTHER`
   - Só são executadas se não existir nenhuma `thread` com necessidades de execução em tempo real
   - Está associada à processos do utilizador
   - A política de escalonamento tem mudado à medida que a são lançadas novas versões do *kernel*

A **escala de prioridades** varia

- 0 a 99 para `real-time threads`
- 100 a 139 para as restantes

Para lançar uma `thread` (sem necessidades de execução em tempo real) com diferentes prioridades, pode ser usado comando `nice`.

Por *default*, o comando lança uma `thread` com prioridade 120. O comando aceita um `offset` de [-20, +19] para obter a prioridade mínima ou máxima.

Algoritmo Tradicional

- Na classe `SCHED_OTHER` as prioridades são baseadas em **créditos**
- Os créditos do processo em execução são **decrementados** à medida que ocorre uma interrupção do `real time clock`
- O processo é `preempted` quando são atingidos zero créditos
- Quando todos os processos `ready` têm zero créditos, os créditos de **todos os processos** (incluindo os que estão bloqueados) são **recalculados** segundo a fórmula:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} + PBase_j + nice_j$$

  onde são tido em conta a **história passada de execução do processo** e as **prioridades**
- O `response time` de processos `I/O-bound` é minimizado
- A `starvation` de processos `CPU-bound` é evitada
- Solução **não adequada para múltiplos processadores** e é má se o número de processos é elevado

53.6 Novo Algoritmo

- Os processos na classe `SCHED_OTHER` passam a usar um `completely fair scheduler (CFS)`
- O scheduling é baseado no `vruntime`, *virtual run time*, que mede durante quanto tempo uma `thread` esteve em execução
  - o `virtual run time` está relacionado quer com o **tempo de execução real** (`physical run time`) e a **prioridade** da `thread`
  - Quanto maior a prioridade de um processo, menor o `physical run time`
- O `scheduler` seleciona as `threads` com menor `virtual run time`
  - Uma `thread` com prioridade mais elevada que fique pronta a ser executada pode "forçar" um `preempt` uma `thread` com menor prioridade
    * Assim é possível que uma `thread I/O bound` "forçar" o processador a `preempt` um processo `CPU-bound`
- O algoritmo é implementado com base numa `red-black tree` do processador