

# Sistemas Operativos

2020 – 2021

**Conceitos sobre inicialização da máquina e preparação do sistema operativo**

## Tópicos

- Bootstraping
- BIOS e firmware
- Partições, sector boot, MBR
- Dualboot
- UEFI e GPT

## O que distingue o SO dos restantes programas?

### Processo de arranque do sistema (“bootstrap”)

- Trata-se de uma questão de importância central

Exemplo de questão relacionada com o processo de arranque

O sistema operativo age como gestor da máquina e tem mais “poderes” do que qualquer aplicação regular (ditas aplicações utilizador).

- *Porquê? O que lhe dá esses “poderes” privilegiados?*

*(neste momento troca-se de uma questão de revisão de matéria)*

## O que distingue o SO dos restantes programas?

*Porquê? O que lhe dá esses “poderes” privilegiados?*

- O processador assume características e capacidades diferentes consoante está a executar código do SO ou de aplicação. A configuração destas características é feita pelo SO **durante o processo de arranque e inicialização**
- Excluindo o **firmware**, o SO é o primeiro programa completo a ser carregado para a máquina e configura o hardware de forma a atribuir a si mesmo as capacidades totais do **hardware**.
  - Todos os restantes programas (aplicações) são executados pelo sistema operativo que os coloca num ambiente de execução controlado com menos poderes sobre a máquina.
  - Basicamente, o sistema chega à máquina primeiro e “arranja” o cenário (configura a máquina=)de forma a quem os programas tenham apenas os privilégios que o sistema entende dar.
  - É então importante perceber como tudo começa: como é que o SO é carregado

-> Assunto destes slides: **como se processa a inicialização da máquina e o carregamento do sistema operativo?**

## Carga (“boot”) do sistema operativo

### Conceitos necessários

- Firmware, BIOS, POST
- Discos, Partições, MBR, Sector boot
- Bootstrapping, Boot loader, chainloading
- UEFI, GPT, Secure boot

## Firmware

### Firmware

- Software armazenado de forma permanente (não precisa de alimentação permanente), inicialmente ROM, agora memória flash
- Contém rotinas utilitárias para controlar aspectos do equipamento e rotinas para inicializar esse equipamento e colocá-lo num estado inicial coerente: *Basic Input Output System* (BIOS)
- Efectua um teste simplificado ao equipamento: *Power On Self Test* (POST)
- Em equipamentos simples, pode conter a totalidade do software necessário à operação do dispositivo.

### Nos computadores habituais

- Contém rotinas de arranque inicial da máquina, rotinas para interacção com dispositivos standard (ex.: discos, teclado, etc.)
- Corre normalmente em modo simplificado (não *privilegiado*) do processador e as suas capacidades de gestão são limitadas.
- Inclui normalmente o software habitualmente designado por BIOS

## BIOS

### BIOS (Basic Input Output System)

- Parte habitual do software habitualmente designado de *firmware*.
  - Chamar “BIOS” ao firmware é errado, mas é muito comum fazer isso
- Parte da norma estabelecida para os IBM-PC e compatíveis.
- Apresenta algumas limitações face ao equipamento moderno e está em processo de substituição (foi substituída) pela norma UEFI
- Contém software para
  - *Interacção com dispositivos standard normalmente presentes em qualquer máquina.*
  - Carregar ou iniciar a carga do sistema operativo

## Tarefas da BIOS

### Inicialização da máquina

- **Identificação e teste** dos componentes principais da máquina (ex., memória, teclado, etc.). Tarefa normalmente designada por POST (Power On Self Test)
- Enumeração dos dispositivos presentes e configuração de cada um de acordo com parâmetros standard
- Passagem do controlo (da execução) ao software que inicia a carga do sistema operativo (*índio do processo de bootstrap*) e que normalmente se encontra em disco

## Tarefas da BIOS

### Rotina de interacção com dispositivos standard

- Materializado sob a forma de um conjunto de rotinas acessíveis ao sistema operativo e a qualquer outro programa
- Podem agir como substituto de funções sistema em cenários em que o sistema operativo é muito simples
  - Exemplo: IBM PC/MSDOS.
  - Rotinas da BIOS (exemplos): acessíveis via int 10H, int 09H, etc.
- Limitadas na sua ação por:
  - Estarem preparadas para dispositivos standard, e portanto *não aproveitam capacidades específicas ou optimizadas de hardware melhor.*
  - Correrem habitualmente com o processador em modo não privilegiado (implica menor capacidade de ação)
- Estas rotinas são essenciais ao processo de arranque e configuração da máquina, sendo apenas dispensáveis se o sistema operativo tiver software que as substitua e apenas depois deste estar presente em memória

## Tarefas da BIOS - Bootstrapping

### Início do processo de bootstrap

- Leitura dos parâmetros de configuração da máquina em memória não volátil para determinação de qual o dispositivo por onde se inicia a carga do sistema (ex: disco, CD, USB, etc.)
- Análise do dispositivo em questão (o disco, CD, pen, o que for) e leitura do **boot loader** para memória
- Passagem do controlo da execução para o **boot loader** que foi carregado. Deste ponto em diante a BIOS age essencialmente como repositório de rotinas utilitárias. O processo de **bootstrap** prossegue com a execução do **bootloader**.

## Bootstraping

### **Bootloader**

- Pequeno programa armazenado em memória secundária que começa o processo de carga do sistema
- Razoavelmente *standard* quanto à localização e formato mas ainda assim algo dependente do sistema operativo e bastante dependente da arquitectura
  - Tamanho standard típico (norma MBR): menos que 512 bytes.  
Sendo um tamanho bastante pequeno, o *bootloader* tipicamente carrega outros programas (cada vez mais complexos e dependentes do sistema operativo) e transfere o controlo a esses programas. O efeito é o de uma sequência de peças de dominó a em que cada uma empurra a seguinte.
  - Locais típicos: **início do disco , início da partição activa do disco**
- A sua localização depende do tipo de dispositivo
  - Os exemplos e descrição seguintes assumem que o dispositivo é um disco rígido (cenário mais habitual)

## Discos rígidos

### **Organização dos discos rígidos**

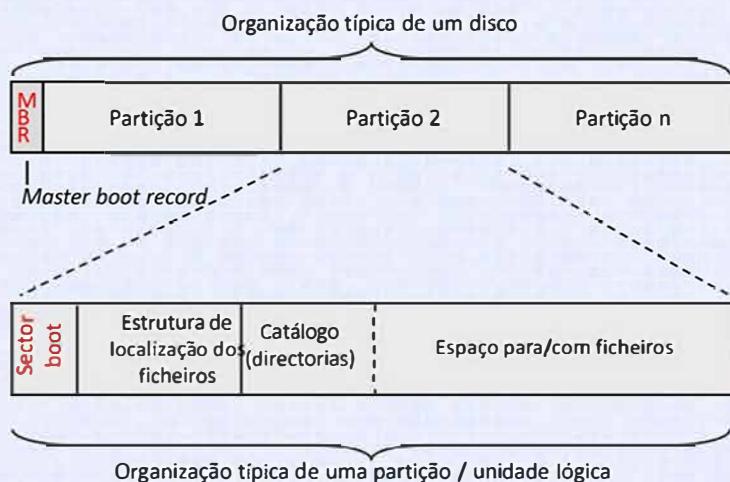
- Os discos rígidos estão normalmente organizados em **partições**.
  - Cada partição pode ter um sistema de ficheiros diferente
  - Cada partição pode hospedar sistema operativo independente
  - Alguns sistemas de ficheiros conseguem abranger várias partições dando a ideia da existência de um só sistema de ficheiros

## Discos rígidos

### Organização dos discos rígidos

- O particionamento dos discos (e do processo de bootstrap) é dependente de normas
- Existem duas normas principais
  - **BIOS/MBR** – original do IBM PC mas limitada e a cair em desuso
    - O nome é artificial (juxtaposição das partes) e foi dado retroativamente
    - Ainda bastante usada em sistemas pequenos ou de 32 bits
  - **UEFI** – Mais recente e poderosa, na prática a mais usada hoje em dia
    - Nome: *Unified Extensible Firmware Interface*

## Discos rígidos – Lógica Bios/MBR



## Discos rígidos - Partições

### **MBR – Master boot record**

- Normalmente um sector (pode ser mais), sendo o primeiro sector do disco
- Contém um pequeno programa que é um *boot loader*.
  - Este programa pode, em teoria, carregar o sistema directamente, mas normalmente passa o controlo para um outro *boot loader* que se encontra no sector boot da partição activa
- Mantém informação acerca da geometria do disco:
  - Quantas partições existem
  - Onde começa e acaba cada partição
  - Qual a partição que está activa (qual a que contém o S.O. de arranque)

## Sistemas de ficheiros – Partições: constituintes

### **Sector boot**

- Normalmente 1 sector (pode ser mais), sendo o primeiro sector de uma partição
- Contém um pequeno programa que é um boot loader e dá inicio ao arranque do sistema (se a partição for a activa)
  - Normalmente o sistema que está nessa partição, mas pode ser outro
  - Este boot loader é, normalmente, específico ao sistema operativo, enquanto que o boot loader no MBR é, normalmente, mais genérico.
  - A instalação de boot loader deve ser feita, sempre que possível, na partição do sistema operativo em questão, deixando o MBR intacto tanto quanto possível uma vez que esse diz respeito ao disco todo.
- Contém informação variada acerca da partição
  - Localização acerca das outras componentes
  - Tamanho dos blocos lógicos (*clusters*)

## Bootstrapping

### Processo de carga do sistema

1. A BIOS identifica o dispositivo onde deve procurar o sistema operativo (informação armazenada na configuração em memória não volátil)  
Assume-se nestes exemplos de que se trata de um disco rígido
2. A BIOS lê o MBR para memória e transfere a execução para o programa que se encontra dentro do MBR.
3. O programa no MBR determina qual a partição activa (tabela existente no MBR), lê o sector boot dessa partição para memória e transfere-lhe a execução.
4. O programa no sector boot prossegue a carga do sistema carregando os ficheiros do sistema presente na sua partição, ou apresentando um menu possibilitando passar para outro sector boot de outra partição (dual boot)  
O processo de um sector boot carregar o sector de outra partição e passar-lhe o controlo é designado de *chainloading*

Tanto o programa do MBR como o do sector boot da partição podem apresentar opções de escolha para dual boot. Normalmente esta escolha é feita a nível do sector boot pois as alterações no MBR afectam o disco todo (por oposição de afectar apenas uma partição) e portanto são de evitar tanto quanto possível

## Dual boot

Consiste em ter vários sistemas operativos na máquina (normalmente em partições diferentes) e ter a possibilidade de arrancar a máquina com qualquer um desses sistemas operativos (em alternativa)

### Método de dual boot em BIOS/MBR

Configuração do *boot loader* no sector boot para apresentar um menu que possibilite

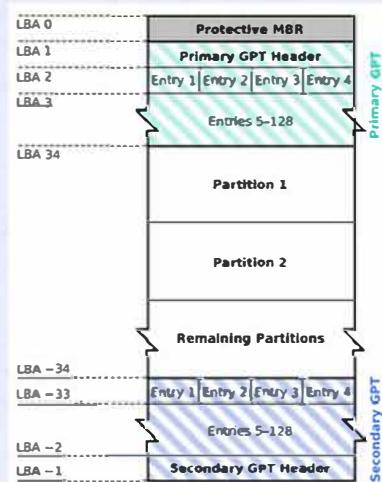
- Carregar os ficheiros do sistema presentes na partição em questão  
Ou então
  - Duplicar o procedimento efectuado pelo programa no MBR e
    1. Carregar um outro sector boot para memória
    2. Passar a execução para o código desse outro sector bootEste procedimento é conhecido com *chainloading*

## Norma UEFI

### UEFI – Unified Extensible Firmware Interface

- Norma que substitui a anterior (BIOS/MBR)
- Sendo mais moderna, permite um conjunto alargado de funcionalidades mais em linha de conta com o hardware recente, por exemplo:
  - Discos GPT (GPT = GUID Partition Table, GUID = Global Unique IDentifier)
    - São discos normais particionados segundo um esquema diferente daquele possível com o MBR.
  - Secure boot
    - Utilização de certificados digitais nos *loaders* para controlar o acesso de um sistema à máquina. Um sistema que não tenha um certificado válido quando comparado com os que estão presentes na memória da máquina, será impedido de se carregar e executar.
    - Os certificados podem ser adicionados e a opção de secure boot pode (eventualmente) ser desligada (acções do administrador)
  - Rotinas de interface com hardware mais evoluídas de carácter semelhante a *device drivers* em linguagem independente do processador
  - Suporte melhorado para placas gráficas recentes

## Organização de discos rígidos GPT



### Discos GPT

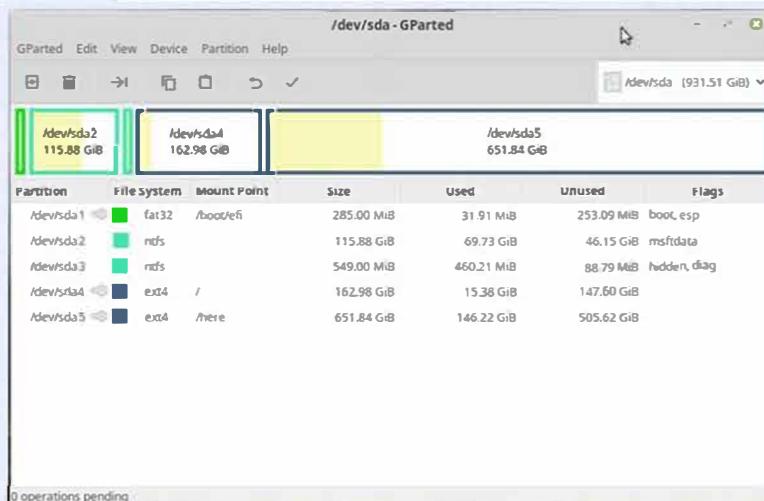
- Possíveis em *chipsets* e sistemas compatíveis com UEFI
- GPT: substitui a lógica antes atribuída ao MBR, mas essencialmente é a mesma coisa com mais capacidades e flexibilidade
- Permite 128 partições
- O arranque do sistema pode ser feito de forma mais flexível e através de pequenos programas (*loaders*) existentes na primeira partição (designada de "partição de sistema")
- Suportam características mais evoluídas tais como certificados de secure-boot

### Exemplo Dual Boot Linux + Windows em UEFI

Neste exemplo

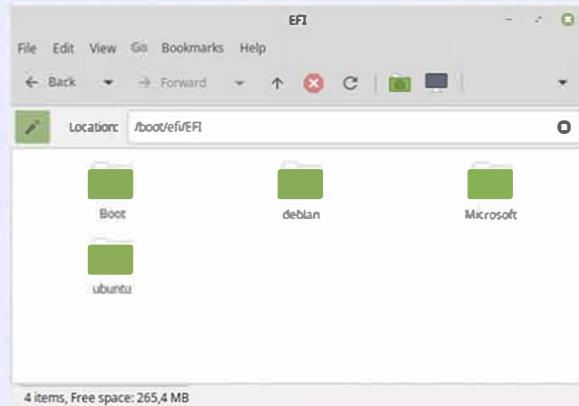
- A máquina está configurada com UEFI
- Existem dois sistemas instalados: Ubuntu Mint e Windows 10.
  - Existia um outro (Debian) que foi removido mas cujo *bootloader* não foi totalmente apagado e esse “resto” é visível
- Existem 5 partições GPT
  - Partição EFI – tem os bootloaders do disco, formatada com FAT32 para compatibilidade entre todos os sistemas
  - Partição Windows formatadas com NTFS
  - Partição de “Recuperação do Windows” formatada com NTFS
  - Partição Linux (para o sistema) formatada com EXT4
  - Partição Linux (para dados) formatada com EXT4 (para Linux)

### Exemplo Dual Boot Linux + Windows em UEFI



Partições existentes (vistas com o programa *gparted* em Linux)

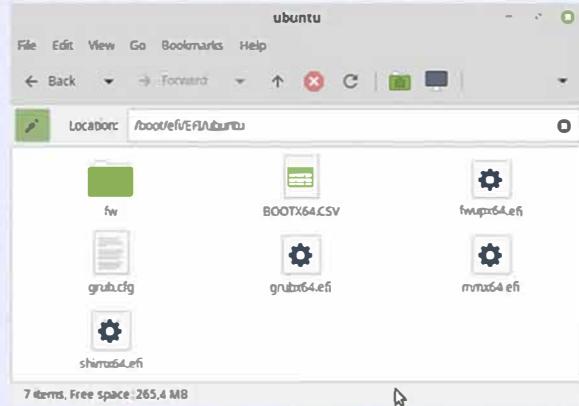
Exemplo Dual Boot Linux + Windows em UEFI



Conteúdo da partição EFI: contém os *bootloaders* dos vários sistemas presentes  
(O sistema debian já não está instalado mas permaneceu a directória referente ao seu  
bootloader)

DEIS/ISEC Sistemas Operativos – 2020/21 João Durães

Exemplo Dual Boot Linux + Windows em UEFI



## Ficheiros de arranque do Linux “Ubuntu Mint”

-> Notar onde a partição EFI “de sistema” foi montada: /boot/efi

Trata-se de ficheiros específicos a este sistema, mas com alguns ficheiros que a norma UEFI espera existir (ficheiro ".efi")

DEIS/ISEC Sistemas Operativos – 2020/21 João Durães

### Exemplo Dual Boot Linux + Windows em UEFI



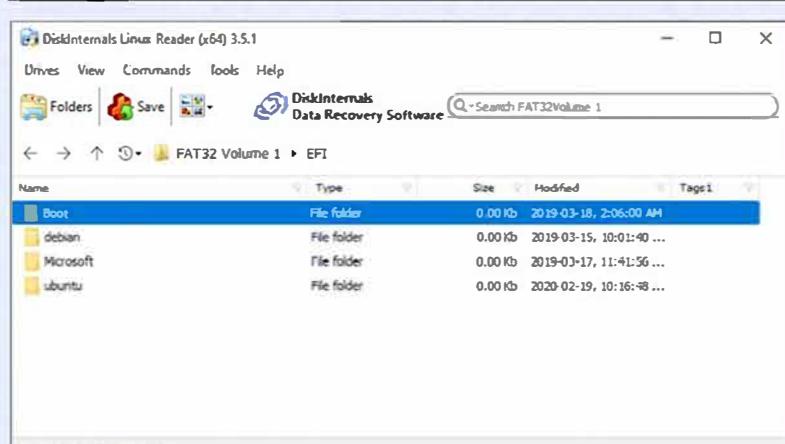
61 items. Free Space: 265.4 MB

Ficheiros de arranque do Windows (observados a partir de um Linux)

Ficheiros específicos a este sistema ("dll"), mas com alguns ficheiros que cumprem a norma geral UEFI (notar os ficheiros com extensão ".efi")

DEIS/ISEC      Sistemas Operativos – 2020/21      João Durães

### Exemplo Dual Boot Linux + Windows em UEFI



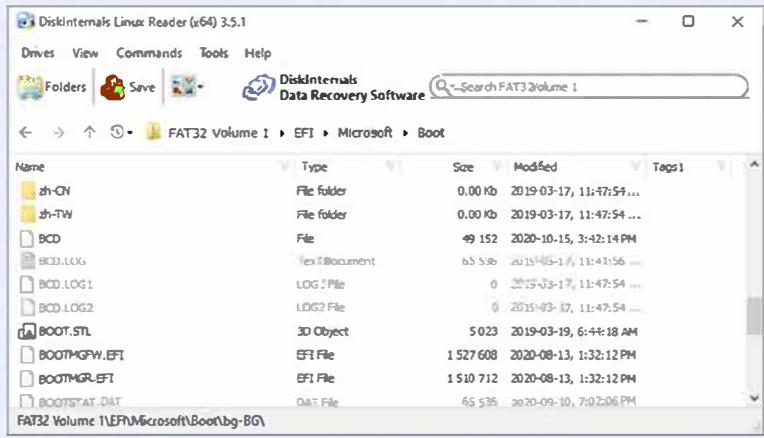
Name	Type	Size	Modified	Tags
Boot	File folder	0.00 kb	2019-03-18, 2:06:00 AM	
debian	File folder	0.00 kb	2019-03-15, 10:01:40 ...	
Microsoft	File folder	0.00 kb	2019-03-17, 11:41:56 ...	
ubuntu	File folder	0.00 kb	2020-02-19, 10:16:48 ...	

FAT32 Volume 1 \ EFN\Boot

A mesma informação observada a partir de um sistema Windows com recurso a ferramentas *third party* ("DiskInternals Linux Reader")

DEIS/ISEC      Sistemas Operativos – 2020/21      João Durães

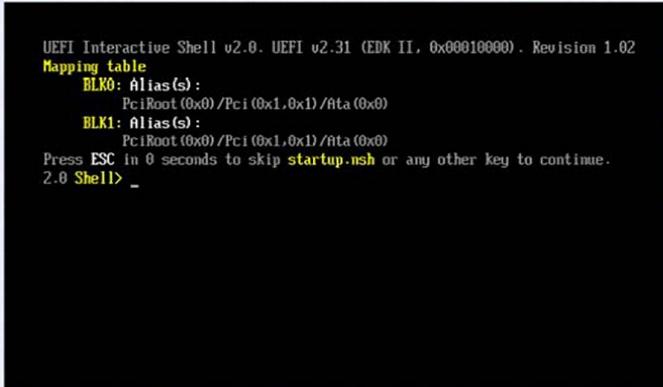
### Exemplo Dual Boot Linux + Windows em UEFI



Ficheiros específicos da Microsoft relativos ao arranque do Windows  
( Partição UEFI de sistema, directória \EFI\Microsoft\Boot )

DEIS/ISEC      Sistemas Operativos – 2020/21      João Durães

### Exemplo Interface UEFI no firmware (sem qualquer SO envolvido)



Aspecto do ecrã de uma *shell* de arranque num sistema com UEFI.  
Note-se a existência de um *prompt* para a execução de comandos  
Em alguns casos pode estar disponível um browser para a web

DEIS/ISEC      Sistemas Operativos – 2020/21      João Durães

### Exemplo Interface UEFI no firmware (sem qualquer SO envolvido)

```

dh          - Displays the device handles in the UEFI environment.
disconnect - Disconnects one or more drivers from the specified devices.
dmem        - Displays the contents of system or device memory.
dumpstore   - Manages all UEFI variables.
drivers     - Displays the UEFI driver list.
drvcfg      - Invokes the driver configuration.
drvdiag    - Invokes the Driver Diagnostics Protocol.
echo        - Controls script file command echoing or displays a message.
edit        - Full screen editor for ASCII or UCS-2 files.
eficompress - Compress a file using UEFI Compression Algorithm.
efidecompress - Decompress a file using UEFI Decompression Algorithm.
else        - Identifies the code executed when 'if' is FALSE.
endfor      - Ends a 'for' loop.
endif       - Ends the block of a script controlled by an 'if' statement.
exit        - Exits the UEFI Shell or the current script.
for         - Starts a loop based on 'for' syntax.
getetc      - Gets the MTC from BootServices and displays it.
goto        - Moves around the point of execution in a script.
help        - Displays the UEFI Shell command list or verbose command help.
hexedit     - Full screen hex editor for files, block devices, or memory.
if          - Executes commands in specified conditions.
ifconfig    - Modifies the default IP address of the UEFI IP4 Network Stack.
load        - Loads a UEFI driver into memory.
loadpciron - Loads a PCI Option ROM.
  
```

Exemplo 2 – lista parcial dos muitos comandos disponíveis (lista obtida com comando ‘help’) – Não há aqui nenhum SO envolvido

Notar funcionalidade complexa: comandos para *scripting* (*echo, for, if*)

### Exemplo Interface UEFI no firmware (sem qualquer SO envolvido)

```

parse      - Retrieves a value from a record output in a standard format.
pause      - Pauses a script and waits for an operator to press a key.
pci        - Displays PCI device list or PCI function configuration space.
ping       - Pings the target host with an IPv4 stack.
reconnect  - Reconnects drivers to the specific device.
reset      - Resets the system.
rm         - Deletes one or more files or directories.
sermode    - Sets serial port attributes.
set        - Displays or modifies UEFI Shell environment variables.
setszie   - Adjusts the size of a file.
setwvar   - Changes the value of a UEFI variable.
shift      - Shifts in-script parameter positions.
smbiosview - Displays SMBIOS information.
stall      - Stalls the operation for a specified number of microseconds.
time       - Displays or sets the current time for the system.
timezone   - Displays or sets time zone information.
touch      - Updates the filename timestamp with the current system date and
time.
type       - Sends the contents of a file to the standard output device.
unload    - Unloads a driver image that was already loaded.
ver        - Displays UEFI Firmware version information.
vol        - Displays or changes information about a disk volume.

Help usage:help [cmd|pattern|special] [-usage] [-verbose] [-section name] [-b]
Shell> -
  
```

Exemplo 3 – lista de comandos disponíveis no *firmware* (sem qualquer sistema operativo a correr ou sequer existente)

Notar funcionalidade: *ping* (redes), *setszie* (trabalha com ficheiros)

# Sistemas Operativos

2020/21

## Conceitos básicos de UNIX

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

## Tópicos

- Ambiente de execução UNIX
- Comandos básicos para utilização do sistema de ficheiros
- Particionamento do sistemas de ficheiros
- Gestão de contas e password.
- Mecanismos de segurança.
- Elevação temporária de privilégios. setuid e sudo
- Configuração de tarefas e de *boot loader*

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

## Ambiente de utilização Unix

- Após o *login* bem sucedido, o sistema lança uma *shell* em nome do utilizador
- A *shell* é um programa que interage com o utilizador e o sistema operativo, encaminhando as acções do utilizador para operações sobre o sistema
- Pode ser consola ou gráfica
  - Consola: o utilizador especifica o que pretende através de comandos escritos
  - Exemplo: bash
  - Gráfica: o utilizador especifica o que pretende através de ícones e menus
  - Exemplos: Gnome, KDE
- A *shell* é um programa como os outros (teoricamente não faz parte do sistema). Corre “em nome” do utilizador e todas as operações que desencadeia e programas que lança são implicitamente em nome desse utilizador

## Ambiente de execução Unix

- Cada **programa** que se executa corre no contexto de um **processo**.
  - Só se consegue correr um programa no contexto de um processo
- O **processo** define diversas características, afectando a forma como o programa no seu interior é executado
  - Isto inclui aspectos importantes **definidos pelo sistema**
    - Zona de memória
    - Prioridade
    - Identificação do utilizador (e consequentemente os seus privilégios)
    - Outros recursos controlados pelo sistema
- Inclui também **aspectos de ambiente, modificáveis pelo utilizador**
  - Directória actual / de trabalho
  - Variáveis de ambiente

-> *O conceito de processo será analisado com cuidado mais adiante*

## Ambiente de execução Unix

- A única forma de criar um processo é através de uma relação pai-filho: um processo duplica-se criando um novo (“filho”)
  - Anterior = “pai”, novo = “filho”
  - O processo filho adquire automaticamente alguma características do processo pai
    - Variáveis de ambiente
    - Ficheiros abertos (pode ser controlado)
    - Programa em execução
    - Identificação de utilizador
    - Directória actual

### Relevância

O sistema apenas necessita de lançar a shell em nome do utilizador. A partir daí, os programas que o utilizador lança (usando a shell) correm em nome desse utilizador, com as suas permissões e recursos (pode ser modificado – exemplo: sudo)

## Ambiente de execução Unix

- Ao criar um novo processo, o processo pai pode controlar várias das características que o processo filho vai ter
  - O processo filho não consegue controlar o ambiente do processo pai
  - Um exemplo onde isto se nota: propagação de variáveis de ambiente do processo pai para o filho e apenas neste sentido

### Relevância

A shell (consola, terminal) pode definir a forma como os programas que invoca se executam pois é a shell que cria os processos onde esses programas correm.

Exemplo de aplicação: redireccionamento (exemplo: ls / more)

*O assunto de criação de processos vai ser visto mais adiante com o devido detalhe*

## Variáveis de ambiente

- Definem aspectos operacionais do ambiente de trabalho do ambiente de execução dos programas e do ambiente do utilizador
- Forma: nome=valor
- Exemplos
  - PATH      Identifica as directórias onde são procurados os programas
  - USER      Contém o *username* do utilizador actual
  - PWD      Contém o nome da directória actual
- Podem ser criadas e modificadas por comandos. Exemplos:
  - declare    Declara e atribui atributos de variáveis
  - export     Marca variável para propagação para os programas seguintes
  - unset     Elimina variável
- As variáveis afectam tanto a execução de programas como o ambiente do utilizador
- São propagadas apenas para a frente: para os programas lançados

## Comandos Unix – alguns exemplos

Forma: comando opções params

- **cd** – muda de directória (comando interno)
- **ls** – mostra ficheiros e direct.                  -a -l -d -s
- **mkdir** – cria directórias                          -p -m
- **cp** – copia ficheiros                                  -u
- **mv** – mov e/rename ficheiros                          -f -i
- **rm** – apaga ficheiros                                  -p -r -d
- **rmdir** – apaga directórias                          -p
- **cat** – mostra ficheiros
- **chmod** – muda permissões de ficheiros
- **chown** – muda dono do ficheiro
- **sudo** – executa em nome de outro utilizador
- redireccionamento: < > |

## Comandos Unix:

- A maior parte dos comandos são simples programas executáveis que se encontram o disco e são executados quando invocados
  - Isto significa que não são funcionalidades internas à *shell*
  - Por isso são designados de “comandos externos”
  - O sistema é facilmente extensível porque a qualquer altura se podem acrescentar comandos novos
  - A variável PATH indica as directórias onde são procurados os programas a executar.
    - Exemplo: *whereis ls* indica onde está o programa correspondente ao comando “ls”
      - *whereis* é também um comando externo
    - Normalmente a própria directória não é pesquisada para executar programas (não está na PATH) e isso é por um questão de segurança

## Comandos Unix:

- Há “comandos” que não correspondem a ficheiros executáveis
- Exemplos: cd, set, export, declare
- O comando *cd* não é um comando externo
  - Está implementado directamente no interpretador de comandos (“ex. *bash*”) e *nem podia ser de outra forma*
    - Exercício/desafio: por que é que não podia ser de outra forma?

## Ficheiros em Unix – output do comando ls

### Comando ls

#### Exemplo de *output*

```
drwxr-xr-x 2 joao joao      4096 Set 19 20:41 Documents
```

- Uma letra indicativa do tipo de ficheiro

- d → directória
- - → ficheiro regular
- p → *named pipe* (mecanismo de comunicação)
- s → *socket* (mecanismo de comunicação em rede)
- l → link simbólico para outro ficheiro (indicado)
- b → device driver de bloco (exemplo: de disco)
- c → device driver de carácter (exemplo: porto série)

## Ficheiros em Unix – output do comando ls

### Comando ls

- Três grupos de permissões (comando chmod)

- r → read
- w → write
- x → execute

A letra aparece: tem permissão.

Aparece “-” no lugar da letra: não tem essa permissão.

- Organizados como dono (*user*), grupo (*group*), outros (*others*) (três letras para cada)
- Podem ser descritos em octal: 1 bit para cada letra, 3 bits por grupo
- Outras permissões possíveis:
  - t → eliminação restrita (só o dono pode apagar).
    - » Exercício: onde fará sentido usar esta opção?
  - S → (em conjunto com execução): bit setuid / setgid ligado

## Ficheiros em Unix – output do comando ls

### Comando ls

#### Exemplo de *output*

```
drwxr-xr-x 2 joao joao 4096 Set 19 20:41 Documents
```

- **Outros campos**

- Dono do ficheiro
- Grupo do ficheiro
- Tamanho
- Data e hora
- Nome

-> Os ficheiros são um aspecto central no sistema Unix. Muitos recursos são usados como se fossem ficheiros

# Sistemas de ficheiros

### Organizado em directórios de carácter bem definido

/

- Ponto inicial do sistemas de ficheiros

/bin

- Ficheiros executáveis (ex. comandos)

/dev

- Contém os gestores dos dispositivos ("device drivers")

/etc

- Contém bibliotecas e ficheiros de configuração (ex.: passwd,)

# Sistemas de ficheiros

## /lib

- Bibliotecas do sistema e de software de carácter geral

## /boot

- Contém os ficheiros de arranque (executáveis e configuração) e o kernel

## /home

- Contém as directórias pessoais dos utilizadores

## /mnt

- Usado para “montar” outras partições ou dispositivos (ex: cdrom, pen usb)

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

# Sistemas de ficheiros

## /opt

- Contém software de carácter opcional, específico a um sistema particular
- Exemplo: um servidor ou programa instalado apenas nesta máquina

## /proc

- Contém pseudo-ficheiros com informação dinâmica (*runtime*) do sistema, por exemplo, acerca dos processos em execução

## /tmp

- Ficheiros de carácter temporário. Normalmente acessível a todos os utiliz.

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

# Sistemas de ficheiros

## /usr

- Ficheiros de carácter geral (“não categorizados”), mas de âmbito do sistema e não de utilizadores em particular

## /var

- Usado para ficheiros de tamanho variável (ex.: ficheiros log)
- Em alguns casos usado como se fosse /opt

## /sbin

- Programas (ficheiros executáveis) de uso restrito (para o super utilizador), normalmente para administração

## /srv

- Para instalação de programas de carácter “servidor” (exemplo: servidores web)

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

# Sistema de ficheiros Unix

**Normalmente organizado em partições independentes com mountpoints em directorias específicas e de uso bem definido**

Existe uma única estrutura de directorias e sub-directorias e determinadas sub-directorias mapeam para outras partições (mesmo que outro disco)

A directoria onde a partição está mapeada é o mountpoint dessa partição

É possível que cada partição tenha o seu próprio sistema de ficheiros, diferente dos restantes, desde que suportados pelo sistema

Exemplo: CDROM -> Mapeado em /mnt/cdrom  
(o mountpoint /mnt/cdrom poderia ser outro)

Comandos envolvidos (uso restrito)

**mount      umount**

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

# Sistema de ficheiros Unix

Muitas vezes algumas directórias centrais estão mesmo mapeadas em partições independentes

**Casos típicos**

- | Casos típicos | Razão   |
|---------------|---|
| • /boot       | Para garantir que tem sempre algum espaço livre                                 |
| • /home       | Para poder levar a partição/disco para outro sistema                            |
| • /tmp        | Tem um sistema de ficheiros afinado para fich. temporários                      |
| • /var        | Tem um sistema de ficheiros otimizados para ficheiros de tamanho muito variável |
| • /mnt/xxx    | /mnt serve especificamente para agrupar mountpoints                             |

# Sistemas de ficheiros

Organização em partições independentes montadas ("penduradas") no sistema principal (o sistema com a directória principal "/" )

**Razões típicas para esta organização**

- Garantir que os ficheiros de uma directória (partição/dispositivo) não extravasam para o resto do sistema de ficheiros (ex., /tmp),
- Garantir que o excesso de ficheiros no resto do disco não passa para a directória (ex., /boot)
- Implementações específicas de partes do sistema de ficheiros (ex: /proc)
  - O conteúdo desta directória são pseudo-ficheiros
- Portabilidade dos dados para outra máquinas (ex., /home)
- Em geral: permitir a capacidade de gestão de cada parte do sistema de ficheiros de forma independente das outras partes
  - Exemplo: necessidade de formas diferenciadas de segurança ou fiabilidade consoante cada parte do sistema

**fstab (file systems table)**
*/etc/fstab*

Auxilia a forma como os vários dispositivos (partições, etc.) são montados no sistema de ficheiros

**Exemplo**

```

# device name  mount point    fs-type      options          dump-freq pass-n
LABEL=/        /              ext3         defaults         1 1
/dev/bda6     swap           swap         defaults         0 0
none          /dev/pts       devpts       gid=5,mode=620  0 0
none          /proc          proc         defaults         0 0
none          /dev/shm       tmpfs        defaults         0 0

# my removable media
/dev/cdrom    /mnt/cdrom     udf,iso9660  noauto,owner,ro  0 0
/dev/fd0      /mnt/floppy    auto         noauto,owner    0 0

# my NTFS Windows XP partition
/dev/hdal     /mnt/WinXP      ntfs         ro,defaults    0 0

# my files partition shared by windows and linux
/dev/hda7     /mnt/shared     vfat         umask=000      0 0

```

**Significado das colunas**

1. Nome do dispositivo ou a sua localização
2. mount point – em que ponto é que a partição vai ser inserida no sistema de ficheiros global
3. Tipo do sistema de ficheiros
4. Opções (slide seguinte)
5. Opções para arquivo da partição com utilitário *dump*.
6. Ordem pela qual o utilitário *fsck* (file system check) analisa as partições à procura de erros quando o sistema arranca

**Tipos de sistemas (coluna 3) – exemplos:**

- fat              MSDOS / Win85/98
- ext2             Linux
- iso9660        CDROM
- Ntfs            WindowsNT (2000,xp, vista, 7-10)

**Exemplo de opções para a coluna 4**

- nosuid – impede bit setuid nos binários nesse dispositivo
- auto / noauto – controla o mounting automático do dispositivo
- exec / noexec – controla a permissão de executar binários no dispositivo
- ro (read-only) / rw (read-write) – controla o acesso read/write
- uid=xxx, gid=xxx especifica user/group id para a posse dos ficheiros nesse dispositivo
- dmask=xxx,fmask=xxx – especifica bits de permissão (octal) para directorias e ficheiros tal como umask (ver umask)

**-> Algumas opções não são suportadas em todos os dispositivos**

**Exemplo: opções numa partição NTFS montada num sistema linux:**

nosuid,noexec,rw,auto,uid=0,gid=999, dmask=007,fmask=117

- Não permite bit uid
- Não permite executar programas nesse dispositivo
- Montado como read/write
- Automaticamente montado no arranque
- Ficheiros são vistos como pertencendo ao user com ID 0 (root) e ao grupo com ID 999
- Directorias adquirem permissão rwxrwx---
- Ficheiros adquirem permissões rw-rw----
- EM algumas situações o sistema consegue entender o tipo de sistema de ficheiros (NTFS, FAT, vboxsf, etc)

Exemplo: **fstab** para directória partilhada numa máquina VirtualBox

A informação é apresentada com as linhas quebradas por falta de espaço no slide  
# o nome da directória não é o mesmo que o dos documentos de instalação

```
share /mnt/dirpart vboxsf
nosuid,noexec,rw,auto,uid=0,gid=998,dmask=007,fmask=117
0 0
```

#### Significado dos campos e opções principais

- share -> identificador do dispositivo (nome definido no virtualbox)
- /mnt/partilhapt -> Directorian no FS do linux onde aparece o conteúdo partilhado
- vboxsf -> Tipo de sistema de ficheiros (Virtual Box Shared Folder)
- uid=0 -> Vê os ficheiros como pertencendo ao user 0 (root)
- gid=998 -> Vê ficheiros como pertencendo ao grupo 998 (vboxsf)  
Nesta configuração a directória é vista como pertencendo ao root. Os utilizadores "normais" que queiram aceder aos ficheiros deverão pertencer ao grupo vboxsf (tipicamente: GID 998)
- rw -> Os ficheiros em share (mnt/dirpart) vistos como tendo permissões read e write
- auto -> O sistema tenta fazer o mount automaticamente no arranque
- noexec -> Não permite execução de ficheiros existentes em share (/mnt/dirpart)
- nosuid -> Não permite bit setuid ligado (sem necessidade uma vez que "noexec")
- dmask=007 -> Definem permissões rwxrwx--- para as subdirectórias
- fmask=117 -> Define permissões rw-rw---- para os ficheiros

# Gestão de utilizadores

#### Organizados em grupos

- Um utilizador tem um grupo primário mas pode pertencer a mais do que um grupo

#### Ficheiros envolvidos

- /etc/passwd
- /etc/group
- /etc/shadow
- /etc/gshadow

Alguns comandos envolvidos

```
useradd userdel whoami id passwd
```

## /etc/passwd

- Contém a identificação das contas de utilizador
  - Tem, entre outros dados
    - Nome da conta (*login*)
    - Password (codificada)
    - Directória pessoal (*home dir*)
    - ID no sistema
- Este ficheiro está em *clear text*
  - Pode ser lido e escrito pelo administrador
  - Pode ser lido pelos utilizadores → para que fim?
    - Este aspecto é um ponto fraco de segurança → por que razão?

### /etc/passwd

oracle:x:1021:1020:Oracle user:/data/network/oracle:/bin/bash

↓ 1    2    3    4    5    6    7

1. **Username:** “login” do utilizador. Entre 1 e 32 caracteres.
2. **Password:** Password encriptada, ou então um *X* que indica que a password (encriptada) está no ficheiro /etc/shadow
3. **User ID (UID):** Identificação numérica do utilizador (única a nível do sistema). 0 = root, 1-99 = reservados para contas predefinidas. 100-999 = reservadas para contas e grupos administrativos.
4. **Group ID (GID):** ID do grupo principal (do utiliz.) (no ficheiro /etc/group)
5. **User ID Info:** Campo de comentário acerca do utilizador, por exemplo, nome completo e telefone. O comando finger usa esta informação.
6. **Home directory:** Caminho absoluto da directória pessoal do utilizador. Se não for especificada será a raiz (“/”), mas isso não implica direitos de escrita
7. **Command/shell:** Pathname para o programa que serve de interpretador de comandos para este utilizador. Exemplo: /bin/bash). (Tipicamente uma shell; tipicamente a bash, mas pode ser qualquer programa.

## /etc/shadow

- Contém a password dos utilizadores
  - Pode ser lido e escrito pelo administrador
  - Não pode ser lido (nem escrito) pelos utilizadores
    - Isto dificulta ataques de força bruta para obter passwords
  - Contém informação acerca das regras de modificação de password (prazos)
- **Exercício/Desafio**
  - Como é que se podia montar um ataque se um utilizador normal pudesse ler o ficheiro /etc/shadow (ou seja, ler a password codificada)?

## /etc/shadow

### Exemplo

```
smithj:Ep6mckrOLChF.:10063:0:99999:7:::
```

1. **Username:** "login" do utilizador (igual à do ficheiro etc/passwd)
2. **Password:** Password encriptada ou um \* que indica que não é necessário password (má ideia)
3. Número de dias desde que a password foi modificada.
4. Número de dias até poder ser modificada.
5. Número de dias após qual a password deve mesmo ser modificada.
6. Número de dias até avisar que a password está a expirar.
7. Número de dias após password expirada que causa a conta ficar cancelada
8. Número de dias desde que a conta está cancelada.
9. Reservado.

/etc/shadow

**Método de encriptação**

Outro exemplo (focar apenas no campo relativo à password e nos \$)

```
smithj:$1$Etg2ExUZ$F9NTP7omafhKIlqaBMqng1:10063:0:99999:7:::
```

**O primeiro \$... indica o algoritmo**

- \$1 -> Hashing MD5
- \$2 -> Blowfish
- \$2a -> eksblowfish
- \$5 -> SHA-256
- \$6 -> SHA-512

Por omissão: Algoritmo DES com o programa crypt (não é o mais seguro)

Nota: não é para decorar esta tabela mas é para saber a lógica de como isto funciona

/etc/shadow

**Método de encriptação**

Outro exemplo (focar apenas no campo relativo à password e nos \$)

```
smithj:$1$Etg2ExUZ$F9NTP7omafhKIlqaBMqng1:10063:0:99999:7:::
```

**Restantes \$...**

- 2º ->Valor “salt” para configuração do algoritmo (valor aleatório para dificultar desencriptação)
- 3º -> resultado da “encriptação” da password com o salt usando o algoritmo especificado

- A análise dos algoritmos de encriptação é importante mas seria noutras disciplinas.  
Exemplos: “fundamentos teóricos de encriptação” (basicamente = matemática), ou “segurança”

**– Novamente: Exercício/Desafio**

- Desde o último desafio, que novas ideias lhe ocorrem para montar um ataque se pudesse ler o ficheiro /etc/shadow?

## Validação de permissões de operações

- O UNIX valida as operações com base em
  - Identificação dos utilizadores
  - Pertença dos recursos (ex.: ficheiros de configuração)
  - Atribuição de permissões
    - RWX | RWX | RWX
    - Dono | grupo | outros

R = Read, W = Write, X = eXecute

Comando para mudança das permissões: comando **chmod**

Em unix, os recursos manifestam-se como ficheiros ou pseudo-ficheiros. Assim, a operações dos utilizadores são verificadas analisado as permissões expressas nos (pseudo-)ficheiros correspondentes aos recursos manipulados

## Permissões e privilégios de programas

- Quem valida as permissões e privilégios?
- Será o programa?

```
main() {
    if (...user -pode-fazer-isso...)
        /* faz a operação pretendida */
    else {
        printf("lamento, não tem permissões");
    }
}
```

- Ou será o sistema?

```
main() {
    if (executaOperaçãoPretendida(...) == 0) /* função sistema */
        printf("lament, não parece ter permissões");
    else
        printf("ok, operação concluída");
}
```

## Permissões e privilégios de programas

- É obviamente o sistema que faz a validação
- Isto pode ser testado com um simples programa
  - Exemplo, um programa que tenta ler o ficheiro /etc/shadow
- Porquê “obviamente”?

## Permissões e privilégios de programas

- É obviamente o sistema que faz a validação
- Isto pode ser testado com um simples programa
  - Exemplo, um programa que tenta ler o ficheiro /etc/shadow
- Porquê “obviamente”?
  - Porque se fosse o programa a testar as permissões, qualquer um com conhecimentos de programação poderia fazer o seu próprio programa (sem testes nenhum) e fazer o que bem entendesse (mais ou menos)
  - O processo está associado a um utilizador e o sistema conhece essa associação. As operações desencadeadas por esse processo são validadas pelo sistema com base nessa identificação

## Permissões e privilégios de programas

- Porque se fosse o programa a testar as permissões, qualquer um com conhecimentos de programação poderia fazer o seu próprio programa (sem testes nenhum) e fazer o que bem entendesse (*mais ou menos*)

Acerca do “mais ou menos”

- Em última análise, o programador poderia tentar replicar o código do sistema dentro do seu programa e remover os testes e fazer o que bem entendesse na máquina.
- O que o impede?
  - > o **sistema operativo** e o **hardware** (detalhe a abordar novamente mais adiante)

## Permissões e privilégios de programas

O que impede o programa de replicar as operações feitas pelo sistema e fazer ele próprio as operações? -> o **sistema operativo** e o **hardware** (detalhe a abordar novamente mais adiante)

- O código do **núcleo** do sistema corre numa zona de memória reconhecida pelo **processador** como podendo conter determinadas instruções críticas
- Essas instruções são necessárias para (exemplos)
  - Ver/mudar a memória toda do computador
  - Por e tirar programas em execução
  - Interagir directamente com o hardware
- Se essas instruções aparecerem em zonas de memória que não as do núcleo, o processador recusa-se a executá-las e avisa o sistema operativo
- Assim, os processos “normais” não conseguem fazer nada que ponha em causa a máquina e outros utilizadores. A única forma é **passar pelo sistema (núcleo)**, o qual valida tudo em função da identificação do utilizador

## Permissões e privilégios de programas

### Exemplo / demonstração

-> **Fazer e depois executar um programa que tenta usar a instrução HLT**  
 Essa instrução pára o processador (núcleo do processador)

#### Description

HALT stops instruction execution and places the 80386 in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after HALT, the saved CS:IP (or CS:EIP) value points to the instruction following HALT.

#### Flags Affected

None

#### Protected Mode Exceptions

HLT is a privileged instruction; #GP(0) if the current privilege level is not 0

A execução dessa instrução afectaria **toda a máquina e todos os outros processos e utilizadores**. Não parece ser boa ideia permitir a sua execução em "meros" processos-utilizador. O que acontecerá se se executar tal programa?

## Permissões e privilégios de programas

### Exemplo / demonstração

→ Código do programa

```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("\n\nantes da instrução\n");
    __asm__ ( "hlt;" ); /* tenta parar o processador */
    printf("\ndepois da instrução\n\n");
    return 0;
}
```

→ Resultado da sua execução

```
antes da instrução
Segmentation fault
```

## Permissões e privilégios de programas

### Exemplo / demonstração

E se o programa fosse executado como root ou com sudo?

→ O resultado seria o mesmo

*Obviamente*

- O utilizador *root* continua a ser um utilizador e os seus programas são executados em “meros” processos-utilizador.
- Não deve ser confundido privilégios de utilizador (o que é que o utilizador pode ou não fazer) com privilégios de código (o que é que o código privilegiado no núcleo do sistema pode fazer quando comparado com código de processos-utilizador)

## Permissões e privilégios de programas

***“A única forma é passar pelo sistema (núcleo), o qual valida tudo em função da identificação do utilizador”***

O que significa isso?

- Qualquer operação desencadeada pelo código do programa que envolva recursos do Sistema passam sempre pelo Sistema, que valida tudo.
- Exemplo: ***um simples “printf” passa pelo sistema***  
*Afinal de contas, usa o recurso “ecrã”, que pertence ao Sistema e não ao programa*

**printf → fprintf → fwrite → write → device driver da gráfica → ecrã**

## Permissões e privilégios de programas

*um simples "printf" passa pelo sistema*

*Afinal de contas, usa o recurso "ecrã", que pertence ao Sistema e não ao programa*

**printf → fprintf → fwrite → write → device driver da gráfica → ecrã**

- As funções a azul são meras funções biblioteca, no espaço de memória do programa “utilizador”
- As funções a negro pertencem ao sistema operativo, no núcleo, numa zona de memória à qual o processador reconhece e aceita instruções necessárias para interagir com o periférico
  - O processo utilizador não consegue ver nem alterar essa memória

## Permissões e privilégios de programas

- Execução de programas

- É criado um novo processo para conter o código do programa pretendido
  - O novo processo tem as mesmas permissões e privilégios do processo que o lançou

*Assim, quando um utilizador lança a execução de um programa, o novo programa corre com as mesmas permissões desse utilizador*

- Esta é a situação default
  - **Pergunta de revisão:** Como é que isso ocorre?

## Permissões e privilégios de programas

- Execução de programas
    - É criado um novo processo para conter o código do programa pretendido
    - O novo processo tem as mesmas permissões e privilégios do processo que o lançou
- Assim, quando um utilizador lança a execução de um programa, o novo programa corre com as mesmas permissões desse utilizador*
- Esta é a situação default
  - **Pergunta de revisão:** Como é que isso ocorre?
- O utilizador está a usar a sua shell, com permissões associadas a si
  - A shell é quem lança o novo processo
  - Logo, o novo programa (processo) tem as mesmas permissões e privilégios que o utilizador

## Execução privilegiada temporária

### Problema:

- Como permitir a alguns utilizadores privilégio elevados sem partilhar a totalidade dos direitos de administração?
- Para, por exemplo: **permitir mudar a sua password**
  - Implica escrever em /etc/shadow
  - Os utilizadores normais não têm permissões para tal
  - No entanto podem mudar a password -> **como?**
- Pretende-se dizer caso a caso (comando a comando) quem pode executar
- Solução: setuid/setgid e sudo

## setuid – Set User ID upon execution

- Cada processo (“programa”) corre normalmente com os privilégios:
  - **setuid ligado** → do dono do executável: o processo pode fazer o que o utilizador dono também pode
  - **setuid desligado** → de quem lança o processo: o processo só pode fazer o que esse utilizador também pode
  
- Mudança do bit setuid/setgid
  - chmod u+s (adquire permissões do utilizador dono - setuid)
  - chmod g+s (adquire permissões do grupo dono - setgid)
- Os programas que têm o bit setuid/setgid ligado devem ser cuidadosamente construídos de forma a não terem vulnerabilidades → **Pergunta-desafio**: “por que razão?”

## setuid – Set User ID upon execution

- Exemplo de *ls -l* para um programa com setuid ligado

```
-rwsr-xr-x 1 root root 163988 Jun  5 2017 /usr/bin/exemplo
```

Os programas com o bit setuid são potencialmente perigosos (*porquê?*)  
 As shells habituais apresentam esses programas com cores bem visíveis  
 para assinalar esse perigo

Exemplo:

```
lrwxrwxrwx 1 root root      22 May 10  2017 strings -> i686-linux-gnu-strings
lrwxrwxrwx 1 root root      20 May 10  2017 strip  -> i686-linux-gnu-strip
-rwsr-xr-x 1 root root 163988 Jun  5  2017 sudo
lrwxrwxrwx 1 root root      4 Jun  5  2017 sudoedit -> sudo
-rwxr-xr-x 1 root root   42728 Jun  5  2017 sudoreplay
```

**Pergunta de raciocínio:** por que é que os programas com setuid ligado são mais perigosos (mesmo os que pertencem ao Sistema)?

## setuid – Set User ID upon execution

- Voltando à questão inicial

### Pergunta de avaliação de atenção na aula

*Como é que o comando **passwd**, que pode ser executado pelo utilizador, consegue mudar a password se implica modificar o ficheiro protegido /etc/shadow?*

## setuid – Set User ID upon execution

- Voltando à questão inicial – mudar a password

### Pergunta de avaliação de atenção na aula

*Como é que o comando **passwd**, que pode ser executado pelo utilizador, consegue mudar a password se implica modificar o ficheiro protegido /etc/shadow?*

-> (Obviamente), o programa passwd é “dos tais” que têm o bit setuid ligado e pertencem ao root

```
ls -la /usr/bin/passwd
-rwsr-xr-x 1 root root 63736 Jul 27 2018 /usr/bin/passwd
```

Trata-se de um programa que vem com o sistema, cujo código é conhecido (sabe-se o que faz / é de confiança) e faz apenas aquilo que está previsto que faça - o facto de correr como root não significa que faça qualquer coisa que apeteça ao utilizador

## setuid – Set User ID upon execution

### **Desafio:**

- Assuma que pode oferecer programas a outros utilizadores e que pode ligar o bit setuid.

- Mudar permissões incluindo bit setuid: **chmod**
- Mudar o dono de um ficheiro: **chown**

(afinal, se calhar, pode mesmo fazer isso)

**Questão:** Como poderia então montar um ataque para obter o controlo total da máquina?

(Nota: não pode oferecer programas a outros utilizadores a não ser que seja root)

## /etc/sudoers

### **Comando sudo**

- Permite a execução de comandos específicos a utilizadores específicos
  - O utilizador executa o comando sudo, especificando por parâmetro o comando que deseja correr
  - Exemplo: sudo fdisk
- O comando sudo corre com privilégios de administrador
  - **Pergunta de revisão:** → Como é que isso ocorre?

## /etc/sudoers

### Comando sudo

- Permite a execução de comandos específicos a utilizadores específicos
  - O utilizador executa o comando sudo, especificando por parâmetro o comando que deseja correr
  - Exemplo: sudo fdisk
  - O comando sudo corre com privilégios de administrador (root)
    - *Pergunta de revisão:* → Como é que isso ocorre?

Resposta

**O ficheiro sudo pertence ao root e tem o bit setuid ligado**

```
rwxrwxrwx 1 root root          22 May 10  2017 strings  -> i686-linux-gnu-strings
rwxrwxrwx 1 root root          20 May 10  2017 strip   -> i686-linux-gnu-strip
-rwsr-xr-x 1 root root 163988 Jun  5  2017 sudo
rwxrwxrwx 1 root root          4 Jun  5  2017 sudoedit -> sudo
rwxr-xr-x 1 root root 42728 Jun  5  2017 sudoreplay
```

## /etc/sudoers

### Comando sudo

- Posteriormente o comando valida a identificação do utilizador pedindo-lhe a password
  - Para garantir que é mesmo o utilizador autorizado quem está na consola
- De seguida valida o direito do utilizador de efectuar o comando desejado consultando o ficheiro sudoers
- Por fim executa o comando desejado com privilégios elevados lançando-o a partir do seu próprio contexto de execução (que já tem privilégios elevados)

**Pergunta-desafio:** avaliar o entendimento de conceitos

→ Qual a diferença entre setuid/setgid e sudo?

## Sudoers

/etc/sudoers (em versões mais antigas)

Controla o acesso a programas privilegiados por parte de utilizadores não privilegiados (evita a necessidade de partilhar a *password* do root)

Para aceder a um programa privilegiado o utilizador comum deve escrever *sudo comando*

### Exemplo

```
User_Alias    FULLTIMERS = millert, mikef, dowdy
User_Alias    PARTTIMERS = bostley, jwfox, crawl
Cmnd_Alias   KILL = /usr/bin/kill
```

```
# full time sysadmins can run anything on any machine without a password
```

```
FULLTIMERS ALL = NOPASSWD: ALL
```

```
# part time sysadmins may run anything but need a password
```

```
PARTTIMERS ALL = ALL
```

```
# matt needs to be able to kill things on his workstation when they get hung.
```

```
matt      valkyrie = KILL
```

```
# joe may su only to operator
```

```
joe  ALL = /usr/bin/su operator
```

**Versões mais recentes:** acrescentar o utilizador ao grupo sudo (GID 27)

## Outros ficheiros de configuração

- Existem inúmeros serviços e funcionalidades no Unix. Cada um é configurado por simples ficheiros de texto

### Ficheiros de texto:

- Podem ser lidos facilmente sem necessidade de programas específicos (exemplo: *regedit* como no Windows)
- Em caso de problemas de configuração na máquina, um simples editor de texto é o suficiente pra consertar o problema

- Nos slides seguintes, dois exemplos:

- Configuração de agendamento de tarefas
- Configuração do bootloader

**crontab (cron table)**
*/etc/crontab*
**Ficheiro que controla execução periódica de programas (tarefas)**

Existe um utilitário com o mesmo nome para editar este ficheiro.

crontab.allow e crontab.deny → Controla os utilizadores que podem alterar a tabela de cron

cron – É o processo responsável por lançar os programas especificados na tabela

Campo	Função
1º	Minuto
2º	Hora
3º	Dia do mês
4º	Mês
5º	Dia da semana ( 0 = Domingo)
6º	Utilizador
7º	Programa a executar

 Separador = *tab*

“\*” = Todos

Pode-se usar mais que um valor desde que separado por “,”

**Exemplo**

```

01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
  
```

**run-parts**

Executa todos os *scripts* (executáveis) dentro de uma determinada diretoria. Usado para executar programas de hora a hora, diariamente, semanalmente ou mensalmente

Directória	Período
/etc/cron.hourly	De hora a hora
/etc/cron.daily	Diariamente
/etc/cron.weekly	Semanalmente
/etc/cron.monthly	Mensalmente

**Configuração do GRUB (GRand Unified Boot loader)**

/boot/grub/menu.lst (pode variar por distribuição)

- Este *loader* é bastante poderoso e permite carregar sistemas em outros discos, ocultar partições, entre outras funcionalidades
- Identificação do dispositivo com o kernel: (disco, nº da partição)

**Exemplo simples**

```
title      Ubuntu, kernel 2.6.20-16-generic
root      (hd0,0)
kernel    /boot/vmlinuz-2.6.20-16-generic
root=UUID=4a62f231-3ad9-4cf8-bbdd-27586ab374b6 ro quiet
splash

title      Ubuntu, kernel 2.6.20-16-generic (recovery
mode)
root      (hd0,0)
kernel    /boot/vmlinuz-2.6.20-16-generic
root=UUID=4a62f231-3ad9-4cf8-bbdd-27586ab374b6 ro single
```

# Sistemas Operativos

2020 – 2021

**Modelo de programação Unix**  
**Processos, sinais, redireccionamento**

## Tópicos

Funções sistema principais

Criação e gestão de processos

Execução de programas

Sinais

Redireccionamento

Bibliografia específica:

- *Fundamentos de Sistemas Operativos*; 3<sup>a</sup> Ed.; Marques & Guedes  
Capítulos 6 e 11
- *Beginning Linux Programming*; Mathew & Stones  
Capítulos 10,11,12

## Modelo de programação UNIX

### Conceito de processo e programa

- Processos e programas são duas entidades bastante diferentes

#### Programa

- Trata-se de um conjunto de instruções. Descreve como o computador deve proceder para cumprir um determinado algoritmo.
- Essencialmente é apenas uma espécie de receita sem via própria

#### Processo

- Trata-se de um ambiente de trabalho contendo recursos necessários à execução de um programa. O programa é apenas um desses recursos.
- Cada processo tem sempre exactamente um programa, mas pode mudar o programa por outro
- Aquilo que se diz que está em execução são os processos e não os programas

## Modelo de programação UNIX

### Atributos de processos e programas

#### Programa

- Essencialmente **um** ficheiro executável em disco que é transferido para o interior de um processo para ser executado
- Contém instruções e valores iniciais de trabalho
- Para executar um programa é sempre necessário ter um processo

O mesmo programa pode ser executado por diversos processos em simultâneo.

- Corresponde à situação de executar um programa várias vezes em simultâneo
- Cada execução é independente uma da outra, tendo cada execução dados e evolução próprios

## Modelo de programação UNIX

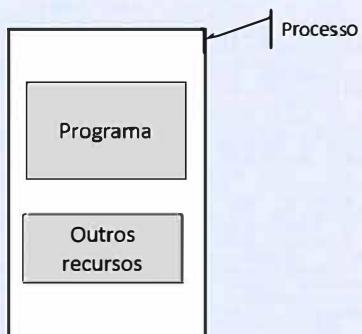
### Atributos de processos e programas

#### Processo

- Conjunto de atributos e recursos
- Espaço de memória organizada em zonas específicas (código, dados, pilha, *heap*)
- Identificação (PID), Prioridade, Directória de trabalho, etc.
- Programa em execução no processo
- Tabela de ficheiros abertos
- Muitos dos atributos do processo podem mudar ao longo da sua vida (inclusive o programa a executar)
- Um processo pode executar vários programas, um de cada vez.
  - A qualquer altura um processo pode mudar o programa por outro. O espaço de memória é adaptado ao novo programa.
- Processos diferentes podem executar o mesmo programa. Mas cada processo é independente dos restantes e as várias “cópias” do programa podem seguir caminhos e ter dados diferentes uns dos outros

## Modelo de programação UNIX

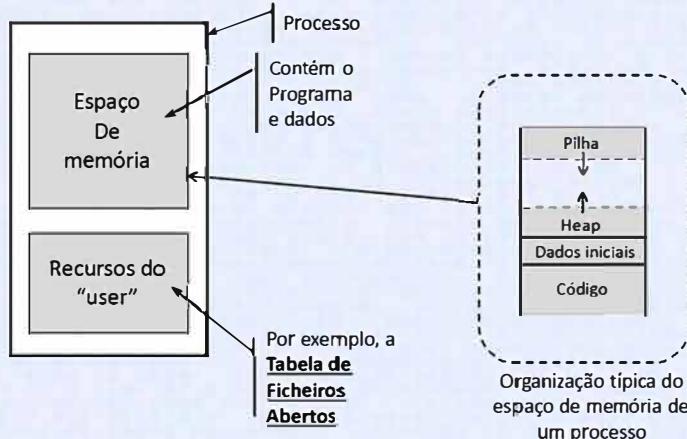
### Aspecto conceptual de um processo



- O conceito de processo e a diferença entre processo e programa aplica-se à generalidade de sistemas operativos e não apenas a Unix
- O tema de Processo vs programa será abordado novamente mais adiante

## Modelo de programação UNIX

Aspecto concreto de um processo (Unix e não só)



## Modelo de programação UNIX

Espaço de memória de um processo

**Código** (designado como **text** “segment”)

- Contém as instruções do programa
- Normalmente marcada como só leitura para execução (não é suposto as instruções do programa mudarem)

**Dados iniciais**

- Contém os valores iniciais do programa.
- Exemplos: valores iniciais de variáveis globais, strings fixas (ex., o “olá mundo” em printf(“olá mundo”), etc.
- Normalmente marcada como apenas de leitura e sem permitir execução. Nem sempre a característica de apenas-leitura é respeitada em todos os sistemas nesta zona de memória (o Unix tipicamente respeita)

## Modelo de programação UNIX

### Espaço de memória de um processo

#### Heap

- Contém variáveis e blocos de memória dinâmica (por exemplo, alocadas por malloc)
- O tamanho máximo desta zona não pode ser determinado à partida: depende da execução do programa. O tamanho é dinâmico e pode ser expandido pelo sistema em caso de necessidade
- Normalmente marcada como leitura/escrita e não permite execução porque se destina a ter dados

#### Pilha

- Contém as variáveis locais e os endereços de retorno das funções
- O tamanho máximo desta zona não pode ser determinado à partida: depende da execução do programa. O tamanho é dinâmico e pode ser expandido pelo sistema em caso de necessidade
- Normalmente marcada como leitura/escrita e não permite execução porque se destina a ter dados (isto impede, por exemplo, ataques *stack smashing*)

## Modelo de programação UNIX

### Espaço de memória de um processo

- As zonas de memória do processo adaptam-se ao programa que foi carregado para o processo.
- Normalmente cada zona é “marcada” com tipos de acesso específico (só leitura, execução: S/N, etc.) para aumentar a estabilidade do programa e sistema e impedir certos tipos de ataques e vírus.
- As zonas *heap* e pilha são dinâmicas e expandem-se consoante o necessário.
  - Expandem-se na direção uma da outra: *heap* expande para cima (para endereços superiores) e a pilha para endereços inferiores
  - Desta forma encontram-se (colidam) o mais tarde possível. Tendo em atenção que não se sabe à partida qual das duas irá necessitar de mais memória, evita-se assim impor limites prévios arbitrários a cada uma – esta “colisão” apenas acontece quando efectivamente não houver mais memória disponível!

O espaço de memória de processos será abordado novamente, por exemplo (mas não só) no capítulo e gestão de memória

## Modelo de programação UNIX

### Implementação em Linux (semelhante à dos outros sistemas)

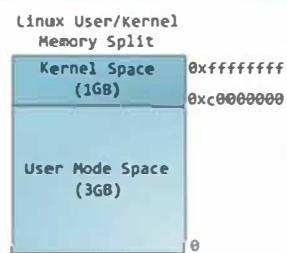
- O sistema atribui uma zona para uso exclusivo do processo.
- Cada processo tem uma zona de memória com aspecto semelhante e com gamas de endereços iguais
- Não há *colisão de memória* entre os vários processos porque os endereços vistos por cada um são mapeados para outros endereços pelo processador em run time de forma a não colidirem
  - Trata-se do **mecanismo de endereçamento virtual** a ver mais adiante
  - Endereços virtuais: Endereços vistos pelas instruções dentro do processo
  - Endereços reais: endereço que realmente estão a ser usados na memória. OS processos não se apercebem desta tradução. É feita por hardware

## Modelo de programação UNIX

### Implementação em Linux

Exemplo em arquitectura de 32 bits

- O conteúdo relativo ao processo fica nos endereços mais baixos.
  - Cada processo tem o seu próprio conteúdo (diferente)
- O sistema fica mapeado no último Gb
- Este último Gb é comum a todos os processos através de um esquema de memória partilhada simples, baseado no mecanismo de endereçamento virtual.
  - O sistema é o mesmo para todos os processos e por isso só existe uma cópia



## Modelo de programação UNIX

### Implementação em Linux (semelhante à dos outros sistemas)

Estrutura e nomenclatura das zonas de memória do processo

- Pilha -> **Stack**  
Variáveis locais não estáticas. Endereços de retorno
- Heap -> **Heap**  
Memória alocada dinamicamente
- Variáveis não inicializadas (podem ser modificadas) -> **BSS segment**  
Exemplo: variável global `int dados[10]; /* onde está o conteúdo de dados */`
- Variáveis inicializadas no código -> **Data segment**  
Exemplo: `char * p = "olá mundo"; /* onde está o "olá mundo"? */`
- Código (instruções) -> **Text Segment**

## Modelo de programação UNIX

### Implementação em Linux (semelhante à dos outros sistemas)

Estrutura e nomenclatura das zonas de memória do processo

- **Stack, Heap, BSS segment, Data segment, Text Segment**

Estes nomes podem ser observados usando a ferramenta ***objdump*** (inspecciona ficheiros objectos e ficheiros executáveis)

- Existem em Linux e em Windows

Esta organização é directamente transcrita do ficheiro executável

- O formato do ficheiro executável é, assim, muito importante e fortemente dependente do Sistema
- Em Linux: formato “**ELF**” (Executable and Linking Format)
- Windows: formato “**PE**” (Portable Executable) – parecido, mas diferente

## Modelo de programação UNIX

### Criação de processos e execução de programas em Unix

Em Unix, a criação de processos e a execução de programas são duas ações distintas e independentes uma da outra

#### Criação de um processo.

- É feito com base na duplicação de um processo que já existente  
-> Mecanismo **fork**
- O novo processo fica com uma cópia do conteúdo do original
- O novo processo fica a executar uma cópia do programa que estava a executar no original, com o mesmo contexto e valores de variáveis (copiadas)
- No novo processo a execução do programa prossegue no ponto em que já ia no contexto do processo original.
  
- O processo original (*pai*) e o seu clone (*filho*) são praticamente indistinguíveis. Terão identificação distintas e podem usar essa informação para decidirem fazer coisas diferentes um do outro.

## Modelo de programação UNIX

### Criação de processos e execução de programas em Unix

#### Execução de um novo programa

- Trata-se de uma operação desencadeada a partir de um processo já existente  
-> Mecanismo **exec**
- O processo passa simplesmente a executar o novo programa, perdendo-se o anterior.
- O novo programa começa a sua execução no inicio (exemplo, função *main*)
  
- Se se quiser executar um programa novo sem perder o que já estava em execução, terá que se criar um novo processo e nele executar o programa pretendido (ou seja, *fork + exec*)

## Modelo de programação UNIX

### Criação de processos

```
pid_t fork(void)
```

Cria um processo novo

Devolve:

- 1 : houve erro
- 0 : contexto do processo filho
- outro valor: contexto do processo pai

A criação do novo processo é feita através de uma forma optimizada baseada em partilha de memória, do mecanismo *copy-on-write* e apoiada nos mecanismos de endereçamento virtual do processador

Inicialmente os processos têm conteúdo igual e partilham a mesma zona de memória RAM. À medida que vão divergindo e tendo valores diferentes, o sistema vai duplicando a memória de forma a que cada processo tenha zonas de RAM diferentes e independentes. Isto é completamente transparente para os processos e será visto novamente no capítulo de gestão de memória

## Modelo de programação UNIX

### Identificação de processos

```
pid_t getpid(void)
```

Obtém identificação do processo

```
pid_t getppid(void)
```

Obtém identificação do processo pai (processo que criou o processo que invoca a função)

## Modelo de programação UNIX

### Execução de programas

```
int execl(char * path, char * arg, ...)
int execlp(char * file, char * arg, char * arg, ...)
int execle(char * path, char * arg, ..., char * envp[])
int execv(char * path, char * argv[])
int execvp(char * file, char * argv[])
int execve(char * file, char * argv[], char * envp[])
```

- Executa o programa indicado no contexto do processo que invoca a função
- Em caso de sucesso não devolve nada.
- O programa em execução é substituído pelo novo. O processo e os recursos mantém-se (exemplo, ficheiros abertos)

## Modelo de programação UNIX

### Sincronização simples de processos

```
pid_t wait(int * status)
      - Espera que um processo filho termine
      - Armazena o exit code em status
pid_t waitpid(pid_t pid, int * status, int options)
```

Espera que um processo filho específico termine

*pid* pode ser (exemplos):

- -1 ► espera pelo primeiro processo filho que terminar
- >0 ► espera pelo processo cujo PID é o indicado (em *pid*)

*options* pode ser :

- WNOHANG ► retorna imediatamente se nenhum processo filho tiver terminado
- WUNTRACED ► retorna também para processos filho que tenham sido parados

## Modelo de programação UNIX

Macros para obter informação a partir do *exit status*:

**WIFEXITED(status)**

- *true* se o processo filho terminou normalmente

**WEXISTATUS(status)**

- Obtém o valor indicado no *return* ou *exit* do processo filho (8 bits)

**WIFSIGNLADED(status)**

- *true* se o processo filho terminou como resposta a um sinal

**WTERMSIG**

- Obtém o sinal que causou o término do processo filho

ETC – consultar páginas man

## Modelo de programação UNIX - Sinais

### Sinais em Unix

Existem actualmente duas formas de funcionamento de sinais

- 1) signal/kill
- 2) sigaction/sigqueue

A primeira forma é mais simples mas menos poderosa

Os sinais correspondem a um mecanismo de notificação.

- Permitem a um processo assinalar a outro que algo ocorreu
- Não são mecanismos de comunicação: não transportam informação nenhuma para além do facto que algo ocorreu (forma 2 permite passar um inteiro)
- Existem diversos sinais. Cada sinal tem um código diferente, permitindo que se possa usar sinais diferentes para objectivos diferentes

Próximos slides: forma 1) = signal/kill

## Modelo de programação UNIX - Sinais

O mecanismo de sinais é completamente assíncrono

- O processo *A* que envia um sinal ao processo *B* fá-lo quando bem entender, não sendo necessário que o processo alvo *B* esteja previamente à espera, ou que seja acordado entre ambos uma altura para fazer o envio.
- O processo *B* que vai receber um sinal não tem que estar explicitamente à espera de um sinal. Estará simplesmente a executar o seu algoritmo e o sinal é entregue a qualquer altura (\*)

Usando uma analogia com a vida real, o mecanismo de sinais é semelhante às campainhas dos prédios com duas alterações:

- Existem várias campainhas para cada apartamento, cada uma com um som diferentes (há vários sinais), e
- Não há intercomunicador – apenas campainhas (os sinais não são mecanismos de comunicação)

(\*) Os sinais são detectados e entregues apenas quando a execução está a passar de código do núcleo para código do processo alvo. Cada sinal corresponde a um bit num vector: foi recebido/não foi recebido. É colocado a 1 quando é enviado e a 0 quando é recebido (não se acumulam enquanto não são tratados)

## Modelo de programação UNIX - Sinais

Sincronização simples de processos com sinais

```
typedef void (*sghandler_t)(int)      ► função para tratar um sinal
```

```
sighandler_t signal(int signum, sghandler_t handler)
```

Associa uma função *handler* ao sinal *signum*

(Significa que o sinal vai ser tratado pelo programa)

Valores especiais para *handler*:

SIG\_IGN – O sinal é ignorado

SIG\_DFL – O sinal causa o tratamento por omissão

Devolve a situação anterior (*handler*) relativa a esse sinal

```
int kill(pid_t pid, int sig)
```

Envia o sinal *sig* a um processo *pid*

- Os sinais apenas são atendidos na transição de código do núcleo para código do processo. O método baseia-se na manipulação da pilha utilizador de forma a criar um registo fictício de chamada a função (a função de tratamento do sinal)

## Modelo de programação UNIX - Sinais

Exemplo de tratamento de sinais (programa que apanha o ^C)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
int a = 0;
void trata_sinal(int s) {
    a++; printf("ouch %d ",a); /* fflush(stdout); */
    if (a==5) { printf("até logo "); exit(0); }
}

int main() {
    setbuf(stdout, NULL);
    signal(SIGINT,trata_sinal);
    while(1) pause();
    return 0;
}
```

## Modelo de programação UNIX - Sinais

Alguns dos sinais mais relevantes

Sinal	Valor	Acção por omissão
SIGHUP	(1)	Termina processo
SIGINT	(2)	termina processo
SIGQUIT	(3)	Termina processo + <i>dump core</i>
SIGKILL	(9)	Termina processo ► Não pode ser tratado
SIGALRM	(14)	termina processo
SIGUSR1	(30)	Termina processo ► Para uso do programador
SIGUSR2	(32)	Termina processo ► Para uso do programador
SIGCHLD	(20)	Ignorado

Há mais sinais – consultar páginas man

Nota: os números dos sinais podem variar de sistema para sistema

## Modelo de programação UNIX - Sinais

Os sinais **desbloqueiam** a execução do processo que os recebem

Processo bloqueado:

- Significa que a execução do processo está parada num determinado ponto à espera de algo.
- Por exemplo: está à espera de caracteres numa leitura desencadeada por um *scanf* (há muitos mais exemplos)
- Em SO, “bloqueado” não significa “encalhado” nem “crashado”. Trata-se se uma situação temporária normal e vulgar.

A receção de um sinal pode interromper, por exemplo

- Uma leitura de caracteres desencadeada por *scanf*
- Um operação *pause()* (aver mais adiante)
- Uma operação *sleep()* (a ver mais adeiante)
- Etc.

## Modelo de programação UNIX - Sinais

Interrupção de uma operação por um sinal

Como agir?

A receção de um sinal pode interromper, por exemplo, uma leitura de *scanf*.

- O que fazer? Repetir o *scanf*? O *scanf* recomeça automaticamente?
- O comportamento do *scanf*(e outros mecanismos) nesta situação pode variar de sistema para sistema. É necessário verificar o comportamento (o *scanf* retorna um inteiro que é o número de campos lidos. Se retornar EINTR significa que foi interrompido)
- No Linux habitual, o *scanf* continua a leitura no ponto onde tinha parado. O próximo exemplo ilustra este comportamento

## Modelo de programação UNIX - Sinais

Exemplo: sinais a meio de interacção com utilizador (1/3)

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int RecebeuSinal; // flag para uso do resto do programa

// função de atendimento de sinal
void atendeSinal(int snum) {
    // executa a acção assíncrona associada ao sinal
    // por exemplo, lê uma mensagem de um named pipe
    printf("\nSinal recebido. ");
    printf("\nProcessar acontecimento, ler named pipes, etc\n");
    // assinala resto do programa que foi recebido o sinal,
    // para o caso de ser relevante, usando a flag
    RecebeuSinal = 1;
}
```

## Modelo de programação UNIX - Sinais

Exemplo: sinais a meio de interacção com utilizador (2/3)

```
int main(int argc, char * argv[]) {
    char buffer[50];
    int res;
    printf("\n\nPID=%d\n",getpid()); // apresenta PID
    RecebeuSinal = 0; // Nenhum sinal recebido
    signal(SIGUSR1, atendeSinal); // associa função a SIGUSR1

    while (1) {
        printf("Favor escrever algo (\\"fim\\" para sair)\n--> ");
        res = scanf("%s", buffer);
        printf("scanf leu isto: %s\n", buffer);
        printf("resultado do scanf = %d\n", res);
```

## Modelo de programação UNIX - Sinais

Exemplo: sinais a meio de interacção com utilizador (3/3)

```

if (RecebeuSinal == 1) {
    RecebeuSinal = 0; // reset à flag
    printf("Parece que entretanto foi atendido ");
    printf("um sinal qualquer\n");
    printf("Lamentamos a eventual confusão na ");
    printf("interface com o utilizador\n");
}
if (strcmp(buffer,"fim")==0)
    break;
}
printf("ok\n\n");
}
  
```

## Modelo de programação UNIX - Sinais

Utilização do exemplo anterior

```

PID=2954
favor escrever algo ("fim" para sair)
--> ola
scanf leu isto: ola
resultado do scanf = 1
favor escrever algo ("fim" para sair)
--> asd
Foi escrito asdfgh
seguido, mas ocorreu
um sinal a meio.
Sinal recebido. Processar acontecimento, ler named pipes, etc
fgh
scanf leu isto: asdfgh
Após o sinal, o scanf
continuou e não
perdeu os caracteres
já lidos.
resultado do scanf = 1
Parece que entretanto foi atendido um sinal
Lamentamos a eventual confusão na user interface
favor escrever algo ("fim" para sair)
  
```

Envio do sinal (noutra consola foi escrito): `kill -s SIGUSR1 2954`

## Modelo de programação UNIX - Sinais

### Acerca do exemplo anterior

A capacidade dos sinais de interromper algo (tarefa 1), desencadeando a execução de uma função associada ao sinal (tarefa 2) pode ser muito útil em cenários em que é necessário a um processo (aparentemente) executar duas tarefas em simultâneo.

Situação exemplo: um processo precisa de fazer em simultâneo:

- Tarefa 1: interagir com o utilizador, (exemplo, `scanf`s e semelhantes)
- Tarefa 2: dar atenção a mensagens provenientes de um outro processo "servidor" (exemplo, ler dados de mecanismos de comunicação inter-processo *named pipes*)

Problema: se está a fazer uma coisa, não está a fazer a outra

Como resolver isto?

-> Resolução com **sinais**: Próximos slides

Obs:

Resolução com **select** -> Mais adiante na matéria

Resolução com **threads** -> Mais adiante na matéria

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

## Modelo de programação UNIX - Sinais

### Soluções

(para quando se quer fazer mais do que uma coisa ao mesmo tempo)

#### Genérica e melhor:

- Usar **threads** (**threads** são discutidas mais adiante e depois aprofundadas em SO2)

#### Específica ao caso de I/O:

- Usar o mecanismo **select** (a ver mais adiante)

#### "Desenrasque":

- Usar **sinais**: próximo slide
- "desenrasque" porque há formas melhores de resolver essa questão (as duas anteriores) e porque só funciona em sistemas onde houver sinais (Unix), e porque os sinais não são (nem deixam de ser) especificamente para esse objectivo

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

## Modelo de programação UNIX - Sinais

### Solução do cenário do slide anterior usando sinais

- Trata-se de um cenário muito semelhante ao exemplo do scanf/sinal apresentado atrás
- O processo **A** está a interagir com o utilizador. Quando o tal outro processo “servidor” **B** tem algo a dizer a este processo **A**, faz:
  - 1º escreve a mensagem no tal mecanismo *named pipe*, e
  - 2º envia um sinal ao processo **A**.O processo **A**, ao receber o sinal, executa uma função (previamente associada ao sinal) que lê a mensagem no *named pipe*, e volta ao que estava a fazer (interagir com o utilizador).

Este cenário/exemplo pode ter aplicação nos trabalhos práticos de SO. No entanto há formas melhores de fazer isto (exemplo: mecanismo **select**)

## Modelo de programação UNIX – Sinais 2: *sigaction*/*sigqueue*

### Segunda forma de lidar com sinais: *sigaction* / *sigqueue*

Podem-se usar duas outras funções para lidar com sinais:

- ***sigaction***: define o comportamento dos sinais (em vez de ***signal***)
- ***sigqueue***: envia sinais (em vez de ***kill***)

Esta forma de uso permite funcionalidade adicional:

- Configurar detalhes adicionais de comportamento dos sinais
- Enviar um valor juntamente com o sinal
- Saber o PID do processo que enviou o sinal

Forma mais recente e mais portável que ***signal/kill***

## Modelo de programação UNIX – Sinais → 2)

### Segunda forma de lidar com sinais: `sigaction/sigqueue`

Função para ver/modificar comportamento associado a um sinal

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);

signum
  Sinal a tratar

struct sigaction * act
  Ponteiro para estrutura que descreve o que fazer com o sinal (como passa a ser tratado o sinal)

struct sigaction * oldact
  Ponteiro para estrutura que é preenchida com a descrição de como era feito o tratamento do sinal
```

## Modelo de programação UNIX – Sinais -> (2)

### Estrutura `sigaction`

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Parâmetro	Significado
<b>handler</b>	Ponteiro para a função que atende o sinal, na forma <code>void func(int)</code>
<b>sigaction</b>	Ponteiro p/ função que atende o sinal, forma: <code>void func (int, siginfo_t *, void *)</code>
-->	Apenas um destes ponteiros deve estar definido (é uma union)
<b>sa_mask</b>	<b>Bit mask</b> com os sinais que ficam bloqueados durante o atendimento
<b>sa_flags</b>	Exemplo: SA_NODEFER (não bloquear o sinal durante o atendimento)
<b>sa_restorer</b>	Uso reservado. Não é para uso da aplicação

## Modelo de programação UNIX – Sinais → (2)

### Segunda forma de lidar com sinais: `sigaction/sigqueue`

Função para enviar um sinal

```
int sigqueue(pid_t pid, int sig, const union sigval value);  
  
pid  
PID do processo alvo  
  
sig  
Sinal a enviar  
  
value  
Valor a passar ao processo alvo juntamente com o sinal
```

## Modelo de programação UNIX - Sinais

### Segunda forma de lidar com sinais: `sigaction`

```
union sigval {  
    int sival_int;  
    void *sival_ptr;  
};
```

Trata-se de uma **union**: os campos estão sobrepostos  
-> Ou se usa um campo, ou se usa o outro

O programador envia ou um inteiro, ou um ponteiro

- O significado do valor é o programador que o decide.  
-> Nota (no caso de se enviar um ponteiro): **um ponteiro só tem significado no contexto do processo onde esse ponteiro foi obtido**  
-> O uso do ponteiro tem a ver com ser **um tipo de dados genérico** (como o void) e não é propriamente para “apontar”

### Modelo de programação UNIX – Sinais → 2)

#### Segunda forma de lidar com sinais: `sigaction/sigqueue`

Função para atender um sinal

```
void funcao(int sig, siginfo_t * siginfo, void * ucontext)

pid
  Sinal recebido

siginfo
  Ponteiro para estrutura siginfo_t com informação acerca do sinal recebido

ucontext
  Ponteiro para estrutura ucontext_t com informações de baixo nível acerca do
  contexto do processo (estado interno do processo, incluindo valores dos
  registo do processador salvaguardados, máscara dos sinais, etc.)
  -> ucontext.h para ver os campos (dependem da implementação do sistema)
```

### Modelo de programação UNIX - Sinais

#### Estrutura `siginfo_t` (os campos mais úteis estão assinalados)

```
typedef struct {
    int si_signo;   -> número do sinal
    int si_code;    -> Informação extra acerca do sinal
    union sigval si_value; -> valor enviado com o sinal
    int si_errno;   -> =errno (normalmente não usado em Linux)
    pid_t si_pid;   -> PID de quem enviou o sinal
    uid_t si_uid;   -> UID (User ID) do processo que enviou o sinal
    void *si_addr;  -> (SIGBUS/SGSEV/SIGILL/SIGFPE) endereço onde foi
                        originado o problema que causou o sinal
    int si_status;  ->(SIGCHLD) Código do sinal enviado ao filho
    int si_band;    -> Código de evento de controlo de I/O
} siginfo_t;
```

Esta estrutura tem mais campos consoante as diversas implementações do sistema

## Modelo de programação UNIX - Sinais

Informação em `si_code` (excerto)

**Geral**

<code>SI_ASYNCIO</code>	Completion of an asynchronous I/O (AIO) operation
<code>SI_KERNEL</code>	Sent by the kernel (e.g., a signal from terminal driver)
<code>SI_MESGQ</code>	Message arrival on POSIX message queue (since Linux 2.6.6)
<code>SI_QUEUE</code>	A realtime signal from a user process via <code>sigqueue()</code>
<code>SI_SIGIO</code>	SIGIO signal (Linux 2.2 only)
<code>SI_TIMER</code>	Expiration of a POSIX (realtime) timer
<code>SI_TKILL</code>	A user process via <code>tkill()</code> or <code>tgkill()</code> (since Linux 2.4.19)
<code>SI_USER</code>	A user process via <code>kill()</code> or <code>raise()</code>

**Para SIGCHLD**

<code>CLD_CONTINUED</code>	Child continued by <code>SIGCONT</code> (since Linux 2.6.9)
<code>CLD_DUMPED</code>	Child terminated abnormally, with core dump
<code>CLD_EXITED</code>	Child exited
<code>CLD_KILLED</code>	Child terminated abnormally, without core dump
<code>CLD_STOPPED</code>	Child stopped
<code>CLD_TRAPPED</code>	Traced child has stopped

**SIGFPE**

<code>FPE_FLTDIV</code>	Floating-point divide-by-zero
<code>FPE_FLTINV</code>	Invalid floating-point operation

ETC -> ver man pages

## Modelo de programação UNIX - Sinais

### `signal/kill` vs. `sigaction/sigqueue`

- **`sigaction/sigqueue` é mais recente e mais versátil**
  - Em princípio será mais vantajoso usar o API mais recente `sigqueue/sigaction`
- **O comportamento não é exactamente igual entre ambos**
  - Isto deve-se ao facto das *flags default* em `sigaction()` não corresponderem ao comportamento *default* em `signal()`

Exemplo:

  - O exemplo do comportamento de `scanf` apresentado atrás tem um resultado diferente se for usado `sigaction()` em vez de `signal()`. Com `sigaction()` o `scanf` é mesmo interrompido e os caracteres anteriormente lidos perdem-se
  - No entanto, este comportamento pode ser configurado de forma a ficar igual ao que se tem com `signal`. Bastara especificar a flag `SA_RESTART` no campo `sa_flags`
  - Em conclusão: `sigaction` é mais versátil e permite especificar melhor o comportamento desejado

## Modelo de programação UNIX - Sinais

### Funções relacionadas com o mecanismo de sinais

`int pause(void)`

- Aguarda que o processo receba um sinal

Esta função pode ser usada para sincronização simples entre dois processos. Um processo *A* que dependa de uma determinada acção *x* por parte de outro processo *B* poderá invocar `pause()`, ficando adormecido. O processo *B*, após concluir a acção *x*, enviará um sinal ao processo *A*, acordando-o.

Importante: a função `pause()` faz com que o processo que a invocou fique adormecido (o termo correcto é “bloqueado”). O processo não fará mais nada até receber um sinal, altura em que prosseguirá normalmente

## Modelo de programação UNIX - Sinais

### Funções relacionadas com o mecanismo de sinais

`unsigned int alarm(unsigned int seconds)`

- Faz com que seja enviado um sinal SIGALRM ao processo que invoca a função dentro de *seconds* segundos.
- Retorna o número de segundos que ainda faltavam referentes a um eventual pedido de alarme anterior (0 se nenhum)
- Após a invocação, o processo prossegue a sua execução normalmente

Esta função pode ser usada para construir um mecanismo de despertador. Um processo *A* pretende ser notificado quando se tiver passado *x* segundos, mas precisa de fazer várias acções e não quer ficar adormecido (“bloqueado”) durante esses *x* segundos.

- Ao invocar a função `alarm()`, o sistema vai enviar-lhe um sinal SIGALRM passados esses *x* segundos, durante os quais o processo prossegue a sua actividade normal.
- Em princípio, o processo terá associado uma função à recepção do sinal para efectuar algo no final dos *x* segundos

### Modelo de programação UNIX - Sinais

Funções relacionadas com o mecanismo de sinais

`unsigned int sleep(unsigned int seconds)`

- Faz com que o processo adormeça (não executa) até que um sinal chegue ou que o número de segundos especificado passe
- Retorna o número de segundos que ainda faltavam no caso do `sleep` ter sido interrompido (por exemplo, pela receção de um sinal)

Esta função pode ser usada para concretizar pausas de duração conhecida. Tal como a generalidade dos restantes mecanismos “bloqueantes”, a pausa pode ser interrompida pela receção de um sinal

### Modelo de programação UNIX – Sinais - Implementação

A informação acerca do estado e configuração dos sinais faz parte de cada processo

Isto significa que:

`fork` → O novo processo “herda” a configuração de sinais feita pelo processo pai (o processo filho obtém uma cópia da informação relativa aos sinais existente nos processo pai)

`exec` → O processo adquire o comportamento *default* para cada sinal uma vez que as funções de atendimento eventualmente programadas deixaram de existir (o código anterior foi substituído por outro)

## Modelo de programação UNIX – Sinais - Implementação

### Implementação do atendimento de sinais

- Cada processo tem uma tabela de *flags de sinais*. Para cada sinal (SIGINT, SIGALRM, SIGUSR1, etc.) existe a informação “sinal chegou / sinal não chegou”. É um bit: sim/não chegou. Não indica quantas vezes chegou “entretanto” um sinal.
- A flag é colocada a 1 quando o sinal é enviado ao processo, e colocada a 0 quando o sinal é atendido.
- Os sinais não acumulam: a informação respeitante a cada sinal é apenas chegou/não chegou (1 *flag* por cada sinal). Se um processo enviar muitos sinais iguais repetidos enquanto o processo alvo não está a executar, este apenas terá a percepção de um sinal X

## Modelo de programação UNIX – Sinais - Implementação

### Implementação do atendimento de sinais

- Os sinais são verificados (e atendidos, se for essa a configuração) quando a execução está a **regressar do sistema para dentro do código do processo**

#### Exemplos

- Retorno de uma função sistema
- Reposição do processo após preempção pelo sistema

Regresso ao código do processo:

- **Situação normal** (sem sinais a atender): o regresso ao código do processo limita-se a saltar para o endereço onde este estava quando perdeu a execução
- **Situação com um sinal associado a uma função**: O sistema “falsifica” uma chamada à função manipulando a pilha do processo. Detalhes -> próximo slide

## Modelo de programação UNIX – Sinais - Implementação

### Implementação do atendimento de sinais

Como é feita a invocação da função de atendimento do sinal no contexto do processo alvo do sinal, não havendo nenhuma chamada explícita feita pelo programador :

**Ao regressar do sistema para dentro do processo o sistema faz:**

- Coloca na pilha do processo um registo “artificial” referente ao endereço onde o processo estava actualmente.
- O sistema salta para o endereço da função de atendimento e não para o endereço onde o processo estava.
- Quando função de atendimento termina, a sua instrução final “RET” naturalmente retira o endereço da pilha que corresponde ao ponto onde o processo ia e a execução passa para esse ponto *tal e qual como se a função tivesse sido chamada explicitamente a partir desse ponto*

## Modelo de programação UNIX – Sinais

### Notas

O comportamento de sinais pode variar de versão de Unix para versão de Unix

(Nota: os vários Linuxes são todos a mesma versão de Unix)

O uso de *signal/kill* em vez de *sigaction/sigqueue* pode fazer variar alguns pormenores do seu funcionamento

-> É importante testar estes aspectos em pequenos programas de prova-de-conceito

### Desafio

(foi falado na aula, portanto é mais *revisão* que outra coisa)

-> *O que acontece ao atendimento de sinais quando*

- é feito *fork()*
- é feito *execxx()*

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento de input/output

- Entre um programa e um ficheiro
- Entre dois programas

O assunto de redireccionamento envolve os seguintes assuntos

- Funcionamento de ficheiros em Unix
- Funcionamento das funções de entrada e saída em Unix
- Tratamento de input/output standard
- Pipes anónimos

Este tópico está fortemente relacionado com os seguintes assuntos

- Processos em Unix e tabela de ficheiros abertos
- Percurso das operações de I/O desde as funções biblioteca até ao sistema
- Mecanismo *fork*
- Mecanismo *exec*

*OS slides seguintes abordam este assunto e no final apresentam três exemplos*

## Modelo de programação UNIX – Redireccionamento

### Funcionamento de ficheiros em Unix

#### Tabela de ficheiros abertos

- Cada processo em Unix tem uma **tabela de ficheiros abertos**
- Cada posição nessa tabela contém um **descritor**. Este descritor tem a informação de controlo acerca do uso do ficheiro (modo de abertura, posição actual no ficheiro etc.)
- Tudo o que o programa precisa de fazer para operar sobre um ficheiro é referir a posição nesta tabela que controla o ficheiro em questão. O sistema irá buscar a essa posição os dados que precisa.
- A função de abertura de ficheiro é responsável por colocar na tabela (primeira posição livre) a informação de controlo acerca do ficheiro
- O programador geralmente não interage directamente com a tabela de ficheiros abertos.

## Modelo de programação UNIX – Redireccionamento

### Funcionamento de ficheiros em Unix

#### Funções de manipulação de ficheiros

##### -> *Funções biblioteca standard C*

- *fopen, fclose, fread, fwrite, etc (com "f")*
- Usam FILE \* para identificar o ficheiro em causa
- Estas funções usam internamente as funções sistema.

##### -> *Funções sistema Unix para manipulação de ficheiros*

- *open, close, read, write, etc (sem "f")*
- São semelhantes às funções biblioteca C.
- Todo o acesso a ficheiros passa obrigatoriamente por estas funções
- Uma diferença muito visível é o facto de usarem um inteiro para identificar um ficheiro em vez de um FILE \*
- **Esse inteiro é simplesmente o índice (posição) dentro da tabela de ficheiros abertos do processo.**

## Modelo de programação UNIX – Redireccionamento

### Funcionamento de ficheiros em Unix

#### Funções de manipulação de ficheiros

##### -> *Funções biblioteca standard C*

- O uso de *fwrite* (exemplo) implica internamente o uso da função sistema *write*
- *printf* e *scanf* são operações de entrada/saída: passam necessariamente pelo uso das funções de manipulação de ficheiros
- *Exemplo: printf --> fprintf --> fwrite --> write*
  - Neste caso, a função *printf* desencadeia o uso de um ficheiro: *stdout* trata-se do pseudo-ficheiro que corresponde ao dispositivo “standard output” (consola). Este ficheiro está sempre na posição 1 da tabela de ficheiros abertos, identificado por **STDOUT\_FILENO**
  - De igual forma, *scanf* (usando *stdin*) implica uma operação *read* sobre o pseudo-ficheiro na posição 0 ou **STDIN\_FILENO** que deve estar sempre na posição 0 da tabela de ficheiros abertos

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento – Ideia base

- printf traduz-se em operações write sobre o ficheiro descrito pela **posição 1** da tabela de ficheiros abertos do processo
  - scanf traduz-se em operações read sobre o ficheiro descrito pela **posição 0**
- O que acontece se se modificar o conteúdo das posições 0 e 1 da tabela de ficheiros abertos de forma a “apontar” para outros ficheiros?
- As operações read e write (relativas aos scanf e printf) serão desviadas (“redireccionadas”) para os ficheiros em vez de teclado/écran
  - Esta ideia corresponde ao redireccionamento tal como em (exemplos)
    - ls > etc.txt
    - sort < abc.txt
  - Também se pode redireccionar a saída de erro standard (stderr), cujo descritor corresponde à posição 2 (STDERR\_FILENO) da tabela de ficheiros abertos

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento para ficheiros – Concretização

- > Substituir os descritores na tabela de ficheiros abertos correspondentes às posições *stdin* (posição 0 = STDIN\_FILENO) e *stdout* (posição 1 = STDOUT\_FILENO) por descritores para outro destino
- Os novos descritores podem referir ficheiros regulares ou mecanismo que seja compatível com a ideia de ficheiro (exemplo, pipes)
  - Tudo o que é necessário é que os descritores estejam nas posições standard (0 para *stdin*, 1 para *stdout*, 2 para *stderr*)
  - Exceptuando eventuais pormenores de sincronização, as operações iniciadas pelos printf/scanf continuam válidas apesar de terem sido redireccionadas para outros destino

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento para ficheiros – COncretização

Redireccionamento feito pela *Shell* na linha de comandos:

->A substituição dos descritores é feita pela *Shell* antes de lançar a execução do comando alvo do redireccionamento, e já dentro do contexto do processo criado para a sua execução (para não afectar as operações d I/O da própria *Shell*)

- Essa substituição ocorre *depois do fork e antes do exec*, na sequência habitual de acções da shell no lançamento de um comando
  - Depois do fork, já no processo filho -> de forma a não afectar a tabela de ficheiros abertos da própria shell (o redireccionamento é apenas para o comando em questão)
  - Depois do fork, já no processo filho mas antes do exec, no processo filho -> a invocação de exec substitui o código pelo do programa a executar mas o processo é o mesmo e mantém-se e a tabela de ficheiros: o novo código irá usar a tabela com a alteração feita e o redireccionamento é transparente para o programa executado

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento para ficheiros

Exemplo para redireccionamento de stdin (exemplo: *sort < abc.txt*)

- No contexto da shell, processo filho

```
close(STDIN_FILENO);           // liberta posição 0 (stdin)
open("abc.txt", O_RDONLY);     // abre o ficheiro (fica na pos. 0)
execvp("sort", "sort", NULL); // executa o programa sort
```

Exemplo para redireccionamento de stdout (exemplo: *ls > fich.txt*)

- No contexto da shell, processo filho

```
close(1);                     // liberta posição 0 (stdout)
open("fich.txt", O_WRONLY);    // abre o ficheiro (fica na pos. 1)
execvp("ls", "ls", NULL);     // executa o programa sort
```

Nota: podem-se usar ambos os redireccionamentos em simultâneo

É com que acontece com, por exemplo: *sort < abc.txt > ord.txt*

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento entre programas com *pipes* anónimos

O redireccionamento entre dois programas (exemplo: ls | sort) é mais complexo do que o redireccionamento para ficheiros. No entanto, o mecanismo básico é o mesmo

- Usa-se um mecanismo de comunicação entre processos: **pipe anónimo**
- Um pipe anónimo comporta-se como dois ficheiros e produz **dois descritores** de ficheiros abertos: num escreve-se (extremidade de escrita) e no outro lê-se aquilo que foi escrito (extremidade de leitura)
- Num processo redirecciona-se o output (stdout) para a extremidade de escrita do pipe
- No outro processo redirecciona-se o input (stdin) para a extremidade de leitura do mesmo pipe
- Assim, aquilo que um processo escreve (stdout) é conduzido para a entrada (stdin) do outro processo.

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento entre programas com *pipes* anónimos

Lógica geral (assumir exemplo: ls / sort)

- A shell irá criar dois processos: um para um processo (exemplo: ls), outro para o segundo processo (exemplo: sort)
- Antes de criar ambos é necessário criar um pipe.
- Após a criação de cada filho, e já no contexto de cada filho é feita a substituição dos descritores na tabela de ficheiros abertos

O slide seguinte acrescenta os pormenores

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento entre programas com *pipes* anónimos

- *Antes de criar ambos é necessário criar um pipe.*
  - Este passo cria na tabela de ficheiros abertos dois descritores: uma para a extremidade de leitura e outro para a extremidade de escrita
  - Este passo é feito antes da criação dos filhos para que ambos herdem a tabela de ficheiros já com os descritores para o pipe e assim tenham visibilidade sobre o pipe. *Esta é a única forma de partilhar o pipe anónimo em ambos os processos dado que este mecanismo não tem um identificado associado*
- *Após a criação de cada filho, e no contexto de cada filho:*
  - No processo que escreve (exemplo: ls), redirecciona-se o stdout para a extremidade de escrita do pipe. Isto corresponde a mover (com a função **dup**) o descritor do pipe para a posição do stdout na tabela de ficheiros abertos. Isto é feito antes do exec
  - No processo que escreve redirecciona-se o stdin para a posição do stdin de forma análoga

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento entre programas com *pipes* anónimos

Código exemplo (assumir exemplo: *ls / sort*)

Antes de criar ambos os filhos

```
int mat[2]; // matriz para os descritores das extremidades do pipe
pipe(mat); // cria o pipe.
            // mat[0] --> descritor do lado de leitura
            // mat[1] --> descritor do lado de escrita
// ... fork

// a matriz é preenchida com os índices na tabela de ficheiros abertos
// onde foram colocados os descritores das extremidades do pipe
// filhos herdam a tabela de descritores => herdam o acesso ao pipe
// assim podem comunicar um com o outro através desse pipe
```

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento entre programas com *pipes* anónimos

Código exemplo (assumir exemplo: *ls* / *sort*)

No contexto do processo filho “que escreve” (exemplo = para executar o *ls*)

```
close(1);      // liberta posição de stdout (pos. 1 = STDOUT_FILENO)
dup(mat[1]);  // duplica extremidade de escrita do pipe para pos. 1
close(mat[1]); // fecha extremidade de escrita porque já foi duplicada
close(mat[0]); // fecha extrem. de leitura do pipe porque não a vai usar
                //execvp("ls","ls",NULL);
```

- Deve ser usado o STDIN\_FILENO em vez de 0 e STDOUT\_FILENO em vez de 1
  - (este slide e seguinte não o fazem por razões de espaço)
- **dup()** é necessário para duplicar o descritor colocado a cópia na posição recém libertada pelo close() anterior, ficando na posição standard esperada
  - **Evitar o recurso a dup2()** pois oculta parte do funcionamento desta matéria

## Modelo de programação UNIX – Redireccionamento

### Redireccionamento entre programas com *pipes* anónimos

Código exemplo (assumir exemplo: *ls* / *sort*)

No contexto do processo filho “que lê” (exemplo = para executar o *sort*)

```
close(0);      // liberta posição de stdin (pos. 0 = STDIN_FILENO)
dup(mat[0]);  // duplica extremidade de leitura do pipe para pos. 0
close(mat[0]); // fecha extremidade de leitura porque já foi duplicada
close(mat[1]); // fecha extrem. de escrita do pipe porque não a vai usar
                //execvp("sort","sort",NULL);
```

Slides seguintes: código dos exemplos mostrados na aula com a máquina linux

## Modelo de programação UNIX – Redireccionamento

### Exemplo 1 (print2file.c)

```
int main(int argc, char * argv[]) {
    int f;
    if (argc<3) {
        printf("\n\nfaltam parâmetros: texto ficheiro\n\n");
        exit(1);
    }
    close(STDOUT_FILENO);
    f = open(argv[2], O_WRONLY | O_CREAT,
             S_IRWXU | S_IRGRP | S_IROTH);
    if (f==-1) {
        perror("\n\nnão foi possível criar o ficheiro");
        exit(2);
    }
    printf("%s\n", argv[1]);
    return 0;
}
```

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

## Modelo de programação UNIX – Redireccionamento

### Exemplo 2 (send2file.c)

```
int main(int argc, char * argv[]) {
    int f;
    if (argc<3) {
        printf("\n\nfaltam parâmetros\n\n"); exit(1);
    }
    close(STDOUT_FILENO);
    f = open(argv[2], O_WRONLY | O_CREAT,
             S_IRWXU | S_IRGRP | S_IROTH);
    if (f==-1) {
        printf("\n\nnão foi possível criar o ficheiro");
        exit(2);
    }
    execl(argv[1],argv[1],NULL);
    perror("\n\ncomando não encontrado ");
    close(f);
    return 3;
}
```

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

### Modelo de programação UNIX – Redireccionamento

#### Exemplo 3 (send2exe.c) – parte 1

```
int main(int argc, char * argv[]) {
    int fpid;
    int mat[2];
    if (argc<3) {
        printf("\n\nfaltam parâmetros\n\n"); exit(1);
    }
    if ( pipe(mat) == -1 ) {
        printf("\n\nnão foi possível criar o pipe\n\n"); exit(2);
    }

    fpid = fork();
    if (fpid== -1) {
        printf("\nfork falhou\n\n"); exit(3);
    }

    // filho criado. A seguir: fazer o redir pai -> filho
```

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

### Modelo de programação UNIX – Redireccionamento

#### Exemplo 3 (send2exe.c) – parte 2

```
if (fpid > 0) { // PAI - vai executar programa que escreve
    close (STDOUT_FILENO); // liberta stdout: STDOUT_FILENO=1
    dup(mat[1]); // duplica mat[1] (write) p/ lugar libertado
    close(mat[1]); // fecha este pq trabalha com o duplicado
    close(mat[0]); // fecha lado entrada pq n vai usar no pai
    execl(argv[1], argv[1], NULL);
    // ocorreu erro
    perror("\n\nprograma 1 não encontrado : ");
}
```

DEIS/ISEC

Sistemas Operativos – 2020/21

João Durães

### Modelo de programação UNIX – Redireccionamento

#### Exemplo 3 (send2exe.c) – parte 3

```
if (fpid == 0) { // FILHO - vai executar programa que le
    close (STDIN_FILENO); // liberta stdin: STDOUT_FILENO = 1
    dup(mat[0]); // duplica mat[0] (read) p/ lugar libertado
    close(mat[0]); // fecha este pq trabalha com o duplicado
    close(mat[1]); // fecha lado saida pq não vai usar no pai
    execlp(argv[2], argv[2], NULL);
    // ocorreu erro
    perror("\n\nprograma 2 não encontrado : ");
}

return 4;
}
```

->> Nota: este exemplo não é representativo da Shell no sentido que o processo pai se “sacrifica” para executar o código do programa 1

# Sistemas Operativos

2020– 2021

## Comunicação inter-processo em Unix com Named pipes (FIFOs)

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

## Tópicos

*Pipes (named pipes / FIFOs)*

Exemplo Cliente-Servidor

### Bibliografia específica:

- *Beginning Linux Programming*; Mathew & Stones  
Caps 10,11,12

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

## Mecanismos de comunicação Unix – FIFOs / Named Pipes

### Ficheiros em ambiente Unix Brevíssima introdução

- » Operações básicas
- » Flags e modo de abertura
- » Controlo de permissões por default

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

## open

```
int open(const char *pathname, int flags);  
Flags (as habituais)
```

### Tipo de operação

- O\_RDONLY → Só leitura
- O\_WRONLY → Só escrita
- O\_RDWR → Leitura e escrita

### Acesso ao ficheiro

- O\_CREAT → Cria se não existir
- O\_EXCL → Falha se já existir

### Comportamento no exec

- O\_CLOEXEC → Fecha em execxxx()

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

## open

```
int open(const char *pathname, int flags);
```

Flags (as habituais)

Acesso ao ficheiro

- O\_CREAT → Cria se não existir
- Cria o ficheiro se não existir

Quando há lugar a criação de ficheiro

Quais as permissões (*modo*) do ficheiro?

- O ficheiro é criado com as permissões especificadas no terceiro parâmetro da função (*-> próximo slide*)
- Se esse parâmetro não for especificado, o código da função vai buscar na mesma pilha o valor que corresponderia a esse parâmetro, resultando num modo de ficheiro imprevisível e provavelmente errado

## open

```
int open(const char *pathname, int flags, mode_t mode);
```

**Modo** → Permissões – flags conjugáveis. Valor em octal e significado

- S\_IRWXU (00700) → user pode: read, write, execute
- S\_IRUSR (00400) → user pode: read
- S\_IWUSR (00200) → user pode: write
- S\_IXUSR (00100) → user pode: execute
- S\_IRWXG (00070) → group pode: read, write, execute
- S\_IRGRP (00040) → group pode: read
- S\_IWGRP (00020) → group pode: write
- S\_IXGRP (00010) → group pode: execute
- S\_IROTH (00004) → others podem: read permission
- S\_IWOTH (00002) → others podem: write permission
- S\_IXOTH (00001) → others podem: execute permission

(Nota: User = dono)

## read / write

```
ssize_t read(int fd, void *buf, size_t count);  
  
ssize_t write(int fd, const void *buf, size_t count);
```

Transferem uma sequência de bytes de forma sequencia de ficheiro para memória (read) ou de memória para ficheiro (write)

É preciso indicar

- Qual o ficheiro (handle para o descriptor)
- Onde estão/vão estar os dados
- Quantos bytes a ler/escrever
- Retorna -1 (erro) ou o numero de bytes lidos/escritos e é importante confrmer se os bytes lidos/escritos são o valor que era esperado

## umask / fmask / dmask

Identificam os bits que podem ser definidos nas permissões de novos ficheiros e directorias criados por um processo

- Na shell – umask é um commando
- Em programação C – umask é uam função
- Em /fstab – Define default para o dispositivo. Também disponíveis: fmask (apenas ficheiros) dmask (apenas directorias)

**umask** → ficheiros e directorias

**fmask** → ficheiros (específico para /etc/fstab)

**dmask** → directorias (específico para /etc/fstab)

**fork()** → Filho herda umask do pai

**execxx()** → Não afecta esta definição (pertence ao processo)

## umask / fmask / dmask

Identificam os bits que podem ser definidos nas permissões de novos ficheiros e directórios criados por um processo

Octal digit in umask command	Permissions the mask will prohibit from being set during file creation
0	any permission may be set (read, write, execute)
1	setting of <u>execute permission</u> is prohibited (read and write)
2	<u>setting of write permission</u> is prohibited (read and <u>execute</u> )
3	setting of write and execute permission is prohibited (read only)
4	setting of read permission is prohibited (write and execute)
5	setting of read and execute permission is prohibited (write only)
6	setting of read and write permission is prohibited (execute only)
7	all permissions are prohibited from being set (no permissions)

## Mecanismos de comunicação Unix – FIFOs / Named Pipes

### FIFOs / Named pipes

Mecanismo semelhante aos pipes anónimos mas com capacidade de identificação própria (visível a processos independentes)

Utilizam-se de forma semelhante a ficheiros

Obs.

Estes slides contêm um exemplo. Há mais exemplos em documentos à parte dos slides  
Os exemplos destinam-se a ser analisados com calma fora de aula. Recomenda-se a modificação do código e experimentação. Uma metodologia só de “olhar” não tem grandes resultados. Os exemplos dos slides não são necessariamente para uso directo no trabalho, mas podem ajudar.

## Mecanismos de comunicação Unix - FIFO's

- Apesar de muito fáceis de utilizar, os pipes anónimos têm a desvantagem de não poderem ser utilizados por processos não relacionados entre si

**Razão:**

- Os identificadores de acesso às extremidades dos pipes são meros *handles* (índices na tabela de ficheiros abertos)
  - Só fazem sentido no contexto do processo que os criou (ou de processos filhos dele)
  - Outros processos têm tabelas de ficheiros abertos totalmente independentes e o descriptor de um pipe criado por um processo distinto (não relacionado) é totalmente inútil
- Para processos não relacionados pai/filho ou derivados do mesmo pai é necessário um mecanismo que tenha uma **identificação** que permita a qualquer processo de o referir
  - **named pipes** (slide seguinte)

## Mecanismos de comunicação Unix - FIFO's

### **Named pipes (Pipes com nome) ou FIFO's (First in First Out)**

- Mecanismo de comunicação de utilização análoga aos **pipes anónimos** (a interface tipo ficheiro é mantida) mas em que existe um **nome** que pode ser utilizado por processos não relacionados para obtenção de acesso ao FIFO
- A utilização de um FIFO é **análoga à utilização de um ficheiro**: qualquer processo o pode abrir/ler/escrever se souber o seu **pathname** (e tiver permissões)
- Há algumas diferenças na **semântica de bloqueio** entre ficheiros e FIFOs que são **importantes** (próximo slide)

Mecanismos de comunicação Unix - FIFO's

**Named pipes (Pipes com nome) ou FIFO's (First in First Out)**

Diferenças quando comparados com ficheiros regulares

- **leitura de ficheiro vazio / fim de ficheiro** vs. **Leitura de FIFO vazio**  
Ficheiro: devolve logo FIFO: bloqueia e aguarda (que “outro” processo escreva)
  - **Escrita em dispositivo cheio** vs. **Escrita em FIFO vazio**  
Ficheiro: devolve logo (erro) FIFO: bloqueia e aguarda (que “outro” processo leia)
  - **Abertura de ficheiro** vs. **Abertura de FIFO**  
Ficheiro: abre e devolve FIFO: Normalmente bloqueia e aguarda que outro processo abra para a operação “inversa” ( $R \leftrightarrow W$ )

DEIS/SEC

Sistemas Operativos – 2020/2021

João Durães

## Mecanismos de comunicação Unix - FIFO's

## Diferenças Ficheiros regulares vs. FIFO

A lógica das diferenças na semântica de bloqueio (“de aguardar”) é

Leitura

- Um ficheiro regular não é um mecanismo de comunicação. Se já não tem mais dados, não faz sentido aguardar que haja e a leitura devolve logo
  - Pelo contrário, num FIFO, como mecanismo de comunicação é natural que, se agora não tenham dados, é provável que venha a ter daquela a pouco (escritos pela outra parte) e o melhor é aguardar

## Escrita

- A mesma ideia: se um FIFO está cheio, provavelmente daqui a pouco já não estará porque a outra parte entretanto leu e retirou a informação

## Abertura

- A lógica aqui é “não vale a pena avançar já se “do outro lado” ainda não há ninguém para a operação inversa. Isto pode dar problema de deadlock

DEIS/SEC

Sistemas Operativos – 2020/2021

João Durães

## Mecanismos de comunicação Unix - FIFO's

### Diferenças ficheiros regulares vs. FIFO

- O uso descuidado das operações habituais de abertura/leitura/escrita pode causar problemas de espera mútua ("deadlock")
- Os processos ficam mutuamente à espera uns dos outros e nenhum avança
- É necessário ter cuidado com este aspecto
  - Pode facilmente tornar-se na parte mais complicada do uso de FIFOs mas é facilmente resolvido planeando com cuidado a sequência das operações
  - A semântica de bloqueio pode ser configurada e o uso dos FIFOs pode ser bloqueante = "síncrono" (descrito atrás) ou não-bloqueante (assíncrono)
    - A semântica síncrona é mais flexível mas mais complexa de usar

## Mecanismos de comunicação Unix: FIFO's

### FIFO's (*named pipes*):

- O facto de terem um nome que pode ser conhecido por outros processos (não relacionadas via `fork()`) permite a sua aplicação numa gama mais vasta de situações

### Utilização de FIFO's

- São utilizadas as funções sistema habituais para ficheiros.  
`open`  
`close`  
`read`  
`write`  
\*\*\*
- A única diferença que poderá existir relativamente a ficheiros verdadeiros está relacionada com questões de sincronização

## Mecanismos de comunicação Unix: FIFO's

Funções Unix/C necessárias (1)

- Os FIFO's são manipulados através dos mesmos mecanismos que os ficheiros normais

```
open(const char * filename, int flags)
```

Abre um FIFO/ficheiro já existente

```
write(int fd, const void * buffer, size_t size)
```

Escreve num FIFO/ficheiro previamente aberto

```
read(int, void * buf, size_t size)
```

Lê de um FIFO previamente aberto

## Mecanismos de comunicação Unix: FIFO's

Funções Unix / C necessárias (2)

```
mkfifo(const char * filename, int flags)
```

Cria um FIFO

```
unlink(const char * filename)
```

Remove um FIFO/ficheiro

```
fcntl(int fd, int command, long arg)
```

Manipula as propriedades do FIFO/ficheiro

## Mecanismos de comunicação Unix: FIFO's

Semântica de sincronização na utilização de FIFO's

- Existem duas categorias de uso: "blockante" e "não-blockante"

A semântica síncrona (com bloqueio) é a mais fácil de utilizar

Normalmente as funções de abertura, leitura e escrita bloqueiam caso a informação não possa ser lida/escrita imediatamente, ou, no caso da abertura, se não existir ainda nenhum processo com o mesmo FIFO aberto para a operação inversa (leitura - escrita)

- Depende da forma como FIFO é aberto

- Os FIFO's são abertos ou para escrita, ou para leitura
  - O mesmo processo pode ler e escrever no mesmo FIFO se o abrir duas ou mais vezes
- Na função `open()` pode-se especificar se se deseja a semântica blockante (situação por omissão) ou a não blockante (`O_NONBLOCK`)

→ Existem 4 combinações possíveis (e 4 comportamentos diferentes)

## FIFO's – Semântica de bloqueio de open, read e write

### 1 – Abertura do FIFO para leitura

#### 1-a) Comportamento síncrono (com bloqueio)

`open(<filename>, O_RDONLY)`

- A chamada bloqueia o processo até que algum processo abra o mesmo FIFO para escrita
- A chamada `read()` com o FIFO vazio bloqueia o processo

#### 1-b) Comportamento assíncrono (sem bloqueio)

`open(<filename>, O_RDONLY | O_NONBLOCK)`

- A chamada retorna logo mesmo que nenhum processo tenha o FIFO aberto para escrita
- A chamada a `read()` com o FIFO vazio retorna 0 (bytes lidos)

## FIFO's – Semântica de bloqueio de open, read e write

### 2 – Abertura do FIFO para escrita

#### 2-a) Comportamento síncrono (com bloqueio)

`open(<filename>, O_WRONLY)`

- A chamada bloqueia o processo até que algum processo abra o mesmo FIFO para leitura
- A chamada `write()` com o FIFO cheio bloqueia até poder escrever todos os dados

#### 2-b) Comportamento assíncrono (sem bloqueio)

`open(<filename>, O_WRONLY | O_NONBLOCK)`

- A chamada retorna imediatamente mesmo que nenhum processo tenha o FIFO aberto para leitura
- A chamada a `write()`, quando os dados não cabem:
  - Bytes a escrever ≤ PIPE\_BUF → falha
  - Bytes a escrever > PIPE\_BUF → escreve o que ainda couber e retorna

## FIFO's – Exemplo de aplicação a um caso concreto

### Exemplo de aplicação de FIFO's: Dicionário

- Construção de um sistema em arquitectura **cliente-servidor** que permita a um processo obter a tradução de algumas palavras

#### Servidor

- Mantém a informação que constitui o dicionário
- Não deve ser lançado mais do que uma vez em simultâneo
- A informação do dicionário não tem que ser repetida por vários processos que manipulam texto
- As suas únicas tarefas são: esperar perguntas, procurar a tradução e enviar respostas

#### Cliente

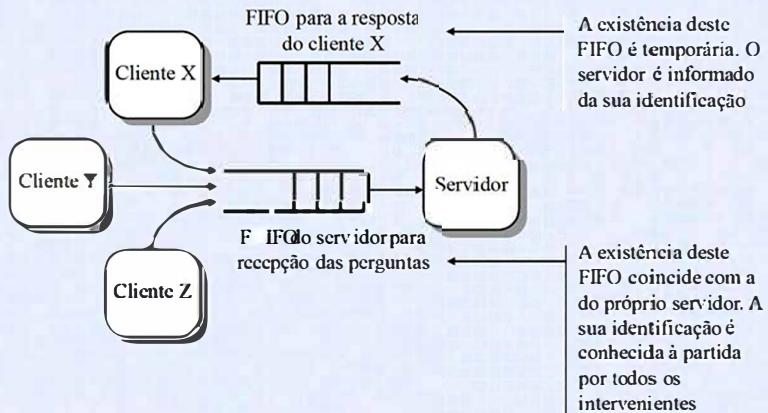
- Efetua perguntas ao servidor e obtém as respostas (tradução)
- Não tem que se preocupar como é que o dicionário funciona – apenas como se utiliza

**Nota:** Este exemplo também se encontra descrito num documento separado

Esse documento contém nova explicação acerca de *named pipes* e uma solução **melhor e mais simples** que a dada aqui. Recomenda-se a sua leitura

### FIFO's – Exemplo de um dicionário cliente-servidor

Esquema da interacção entre os vários processos (servidor e clientes)



### FIFO's – Exemplo de aplicação a um caso concreto

Situação a ter em atenção - I

- Este exemplo inclui uma situação que se não for devidamente acautelada pode levar a que os vários processos fiquem à espera uns dos outros eternamente (situação comum a muitos problemas)
- Devido à semântica de bloqueio dos FIFOs e à ordem pela qual os vários opens e read/writes são efectuados (tal como visto nas aulas)
  - O cliente tem que enviar nome do seu pipe na pedido de forma a que o servidor saiba para onde deve enviar a resposta
  - Deve criar o pipe antes de efectuar o pedido pois o nome previsto pode já estar ocupado
  - Se abrir também esse o pipe para leitura fica bloqueado pois o servidor ainda não o abriu para escrita (nem sequer o conhece)

### FIFO's – Exemplo de aplicação a um caso concreto

#### Resolução

- Só abrir o pipe da resposta para leitura depois de enviar o pedido
  - Ou seja, sendo possível, manipular a ordem das operações referidas acima, desde que possível no(s) programa(s) que se estão a construir.  
Nem sempre é possível mudar a ordem das operações.)
- Efectuar algumas operações sobre o pipe de forma não bloqueantes e posteriormente mudar a semântica do pipe para bloqueantes
  - Função `fcntl`

Neste caso

  - Abrir o pipe como não bloqueante – o processo prossegue
  - Mudar logo de seguida para bloqueante

Neste exemplo ambas as estratégias funcionariam. Vai ser usada a segunda solução para exemplificar a função `fcntl` e de seguida a primeira solução é apresentada de forma simplificada (apenas as partes relevantes do algoritmo)

### FIFO's – Exemplo de aplicação a um caso concreto

#### Situação a ter em atenção - 2

- Existido apenas um cliente (situação perfeitamente razoável) e esse cliente terminando, deixa de existir algum processo com o pipe do servidor aberto para escrita (situação que pode acontecer **frequentemente**)
  - Assim que isso acontece, as leituras do servidor no seu pipe (leituras de perguntas) deixam de bloquear (como se passasse a não bloqueante).
  - Os reads retomam logo e entra-se numa situação de espera activa
    - (Lê – não tem nada – volta a ler – repetir)
  - Espera activa é proibida em todo o lado, em particular em SO
- Resolução
  - O servidor abre o seu próprio pipe para leitura, garantindo assim que há sempre “alguém” com o pipe aberto para leitura (técnica “keep-alive”)
  - A ordem destas aberturas também deve ser ponderada para evitar situações como a descrita em “situação a ter em atenção – 1”

### FIFO's – Exemplo de um dicionário cliente-servidor

Algoritmo simplificado do processo cliente *típico*

```

    Abre o FIFO do servidor para escrita
    Cria o FIFO para receber a resposta
    Abre o FIFO das respostas para leitura
    Repete durante uma certa condição
      | Obtém palavra a traduzir
      | Constrói pergunta = palavra + nome do FIFO para a resposta
      | Envia a pergunta (escreve no FIFO do servidor)
      | Fica à espera da resposta (efectua uma leitura no FIFO para a resposta)
    Fecha o FIFO do servidor
    Fecha o FIFO para as respostas
    Remove o FIFO das respostas
  
```

→ “típico” porque os clientes podem fazer outras coisas além de obter traduções

### FIFO's – Exemplo de um dicionário cliente-servidor

Parte do algoritmo relativo à abertura do pipe não-bloqueante-e-depois-passar-a-bloqueante

Abre o FIFO do servidor para escrita

Cria o FIFO para receber a resposta

Abre o FIFO das respostas para leitura

Repete durante uma certa condição

| .....

Abre o FIFO das respostas para leitura como não-bloqueante

- a abertura não bloquia apesar do servidor ainda não ter aberto este pipe para a operação inversa)

- a utilização de pipes não bloqueante é mais complicada por isso vai-se mudar para bloqueante agora que já foi aberto)

Muda FIFO para bloqueante com a função `fcntl`

### FIFO's – Exemplo de um dicionário cliente-servidor

#### Algoritmo simplificado do processo servidor

Cria o FIFO do servidor para obter as perguntas

Repete durante uma certa condição

    | Obtém a próxima pergunta (efectua uma leitura no FIFO do servidor)

    | Obtém a tradução da palavra

    | Obtém a identificação do FIFO para a resposta: vinha junto com a pergunta

    | Abre o FIFO do cliente que enviou a pergunta = FIFO para a resposta

        | Envia a pergunta (escreve no FIFO para a resposta )

    | Fecha o FIFO para a resposta

    | Fecha o FIFO do servidor

    | Remove o FIFO do servidor

→ Ao invés do(s) cliente(s), o servidor apenas efectua traduções (não se aplica a palavra “típico” como no caso do cliente – só há um servidor a correr).

### FIFO's – Exemplo de um dicionário cliente-servidor

Padrão escolhido para a identificação dos vários FIFO's

- FIFO do servidor para enviar/receber as perguntas

`/tmp/dict_fifo`

– Este nome é previamente estabelecido de modo a que os clientes o possam abrir

- FIFO de cada cliente para receber as respostas

`/tmp/resp_pid_fifo`

– `pid` será o valor do PID do cliente

→ Cada cliente tem o seu próprio FIFO e não ocorrem conflitos

– Não é necessário definir este nome à partida: o servidor será informado do nome juntamente com cada pergunta que recebe

### FIFO's – Exemplo de um dicionário cliente-servidor

#### Ficheiro "dict.h"

```
/* ficheiro header necessário aos clientes e servidor */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>

/* ficheiro correspondente ao FIFO do servidor
#define SERVER_FIFO "/tmp/dict_fifo" ← Previamente estabelecido!

/* ficheiro correspondente ao FIFO do cliente n (n -> "%d")
#define CLIENT_FIFO "/tmp/resp_%d_fifo"
```

O código aqui apresentado foi testado. No entanto, pode ter havido erros na transcrição e formatação. Este aspecto deve ser tomado em consideração e qualquer dúvida comunicada ao docente.

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

### FIFO's – Exemplo de um dicionário cliente-servidor

#### Ficheiro "dict.h" (continuação)

```
/* tamanho máximo de cada palavra */
#define TAM_MAX 50

/* estrutura da mensagem correspondente ao um pedido */
typedef struct {
    pid_t pid_cliente;
    char palavra[TAM_MAX];
} pergunta_t;

/* estrutura da mensagem correspondente a uma resposta */
typedef struct {
    char palavra[TAM_MAX];
} resposta_t;
```

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do cliente ( 1 )

```
#include "dict.h"

int main() {
    int s_fifo_fd; /* identificador do FIFO do servidor */
    int c_fifo_fd; /* identificador do FIFO deste cliente */
    pergunta_t perg; /* mensagem do "tipo" pergunta */
    resposta_t resp; /* mensagem do "tipo" resposta */
    char buffer[80]; /* para a leitura da palavra a traduzir */
    char c_fifo_fname[25]; /* nome do FIFO deste cliente */
    long fflags;
    int read_res;
```

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do cliente ( 2 )

Abre o FIFO do servidor para as perguntas

- Se não conseguir, não é possível comunicar com o servidor (não está a correr ?) e termina

```
s_fifo_fd = open(SERVER_FIFO, O_WRONLY); /* bloqueante */
if (s_fifo_fd == -1) {
    fprintf(stderr, "\nO servidor não está a correr\n");
    exit(EXIT_FAILURE);
}
```

O nome do ficheiro de FIFO do servidor é conhecido

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

### FIFO's – Exemplo de um dicionário cliente-servidor

#### Código do cliente ( 3 )

Cria um FIFO para receber a resposta do servidor

- O nome deste FIFO será enviado ao servidor juntamente com a pergunta
    - Já não é tão crítico como no caso do FIFO do servidor
    - No entanto, interessa obter um nome que tenha uma alta probabilidade de não estar já tomado
- Ideia: Concatenação de "/tmp/resp\_" com o PID com "\_fifo"

```
perg.pid_cliente = getpid();
sprintf(c_fifo_fname, CLIENT_FIFO, perg.pid_cliente);
if (mkfifo(c_fifo_fname, 0777) == -1) {
    fprintf(stderr, "\nErro no FIFO para a resposta (1)");
    close(s_fifo_fd); exit(EXIT_FAILURE);
}
```

### FIFO's – Exemplo de um dicionário cliente-servidor

#### Código do cliente ( 4 )

Abre já o FIFO para receber a resposta do servidor

- Caso contrário o servidor deitaria a resposta fora
- No entanto, não pode ser com semântica bloqueante, senão ficar-se-ia já bloqueado aqui
  - Abre-se como não bloqueante agora, e depois modifica-se para bloqueante (interessa ficar bloqueado à espera da resposta)

```
c_fifo_fd = open(c_fifo_fname, O_RDONLY | O_NONBLOCK);
if (c_fifo_fd == -1) {
    fprintf(stderr, "\nErro no FIFO para a resposta (2)\n");
    close(s_fifo_fd);
    unlink(c_fifo_fname);
    exit(EXIT_FAILURE);
}
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do cliente ( 5 )

Muda-se a semântica do FIFO das respostas para bloqueante

- Simplifica as leituras
- A alternativa seria o algoritmo ter que efectuar a espera pela resposta de uma forma explícita analisando o resultado da leitura (+ complexo)

```
fflags = fcntl(c_fifo_fd, F_GETFL);
fflags ^= O_NONBLOCK; /* inverte a semântica de bloqueio */
fcntl(c_fifo_fd, F_SETFL, fflags); /* bloqueante = on */
```

flags ^= O\_NONBLOCK

- Deixa os outros bits como estavam e inverte o da flag que controla a semântica de bloqueio através de uma operação de ou exclusivo

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do cliente ( 6 )

Obtém a garantia de um '\0'

```
perg.palavra[TAM_MAX-1] = '\0';
```

Ciclo principal:

- a) Obtém a pergunta
- b) Envia a pergunta
- c) (Espera e) obtém a resposta

```
while (1) { /* "fim" termina */
/* ---- a) OBTEM PERGUNTA ---- */
printf("Palavra a traduzir ->");
scanf("%s",buffer);
if (!strcasecmp("fim",buffer)) break;
strncpy(perg.palavra,buffer,TAM_MAX-1);
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do cliente ( 7 )

```
/* ---- b) ENVIA A PERGUNTA ---- */
write(s_fifo_fd, & perg, sizeof(perg));
/* ---- c) OBTEM A RESPOSTA ---- */
read_res = read(c_fifo_fd, & resp, sizeof(resp));
if (read_res == sizeof(resp))
    printf("Traducao -> %s\n", resp.palavra);
else
    printf("Sem resposta ou resposta incompreensivel
            [%d]\n", read_res);
}
```

Verificações efectuadas

- Se o tamanho da resposta lida for diferente do tamanho da estrutura de uma resposta, então assume-se que não é uma resposta correcta
- Pode-se terminar o servidor escrevendo "fim" para o seu FIFO

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do cliente ( 8 )

Termina, libertando os recursos detidos

- Fecha o FIFO deste cliente
- Fecha o FIFO do servidor
  - A utilização do FIFO do servidor por parte de outros clientes não é afectada
- Remove o FIFO deste cliente

```
close(c_fifo_fd);
close(s_fifo_fd);
unlink(c_fifo_fname);
} /* fim da função main */
```

- A remoção do FIFO é feita com recurso a uma função do sistema de ficheiros  
`unlink(<nome do ficheiro>)`

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 1 )

```
#include "dict.h"
#include <signal.h>

#define NPALAVRAS 7      /* Número de palavras conhecidas */

char * dicionario[NPALAVRAS][2] = { /* O dicionário      */
{ "memory",     "memória" },      /* é constituído      */
{ "computer",   "computador" },   /* por uma matriz      */
{ "close",       "fechar" },       /* bidimensional de    */
{ "open",        "abrir" },        /* ponteiros para      */
{ "read",        "ler" },          /* caractere.          */
{ "write",       "escrever" },     /* [i][0] = palavra   */
{ "file",        "ficheiro" } };  /* [i][1] = tradução */

int s_fifo_fd, c_fifo_fd, keep_alive_fd; /* desc. de fich. */
```

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 2 )

```
int main() {
    pergunta_t perg; /* mensagem do "tipo" pergunta */
    resposta_t resp; /* mensagem do "tipo" resposta */

    int read_res,i;
    char c_fifo_fname[50];
    char * aux;

    signal(SIGINT, sig_int);
```

- ➔ A chamada `signal()` vai servir para associar uma função ao sinal SIGINT (interrupção via teclado) para se poder terminar o programa de forma adequada
- A função associada ao sinal (`sig_int(int)`) é descrita mais adiante

DEIS/ISEC

Sistemas Operativos – 2020/2021

João Durães

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 3 )

Cria o FIFO para recepção das perguntas

→ Se o FIFO já existir termina

- Esse recurso já está ocupado e não vale a pena continuar com um FIFO com outra identificação: os clientes não saberiam para onde enviar as perguntas

```
mkfifo(SERVER_FIFO, 0777);
s_fifo_fd = open(SERVER_FIFO, O_RDONLY); /* bloqueante */
if (s_fifo_fd == -1) {
    fprintf(stderr, "\nErro ao criar/abrir o FIFO");
    exit(EXIT_FAILURE);
}
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 4 )

**Problema:**

Quando o último cliente termina, o FIFO do servidor passa a estar na situação de não estar aberto para escrita por nenhum processo

- O servidor, que está bloqueado no `read()` à espera da próxima pergunta é desbloqueado com 0 bytes lidos
- `read()`'s subsequentes não bloqueiam

**Solução**

- Fecha-se o FIFO e volta-se a abrir novamente (para leitura), ou
- Abre-se já este FIFO para escrita, impedindo a situação descrita de ocorrer
  - Esta solução é a mais simples
  - A semântica de bloqueio aqui é indiferente porque o servidor nunca irá escrever neste FIFO

```
keep_alive_fd = open(SERVER_FIFO, O_WRONLY);
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 5 )

Ciclo principal

- a) obtém a próxima pergunta
- b) obtém a tradução (se existir)
- c) envia a resposta (o destinatário vinha identificado na pergunta)

```
while (1) {
    /* ---- a) OBTEM PERGUNTA ---- */
    read_res = read(s_fifo_fd, &perg, sizeof(perg));
    if (read_res < sizeof(perg)) {
        if (!strncasecmp("fim", (char *) &perg, 3)) break;
        else {
            fprintf(stderr, "\nPergunta incompleta!");
            continue;
        }
    }
}
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 6 )

```
/* ---- b) PROCURA TRADUCAO ---- */

aux = "DESCONHECIDO";
for (i=0; i<NPALAVRAS; i++)
    if (!strcasecmp(perg.palavra,dicionario[i][0])) {
        aux = dicionario[i][1];
        break;
    }
strcpy(resp.palavra,aux);

/* ---- OBTEM FILENAME DO FIFO PARA A RESPOSTA ---- */

sprintf(c_fifo_fname, CLIENT_FIFO, perg.pid_cliente);
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 7 )

```
/* ---- c) ENVIA RESPOSTA ---- */

c_fifo_fd = open(c_fifo_fname, O_WRONLY | O_NONBLOCK);
if (c_fifo_fd != -1) {
    write(c_fifo_fd, & resp, sizeof(resp));
    close(c_fifo_fd); /* FECHA LOGO O FIFO ! */
}
else
    fprintf(stderr, "\nNinguem quis a respsta [%s]",
            resp.palavra);

} /* ciclo principal */
```

- ➔ É concebível que o cliente tenha terminado e não queira a resposta. Nessa situação (*bug* ?) o FIFO para a resposta poderia já nem existir. O servidor verifica essa situação e apresenta uma mensagem nesse caso.

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 8 )

Termina, libertando os recursos detidos

- Fecha o FIFO do servidor
  - Lembrar que está aberto duas vezes: para leitura e para escrita
- Remove o FIFO deste cliente
  - Com recurso à função `unlink()`

```
close(keep_alive_fd);
close(s_fifo_fd);
unlink(SERVER_FIFO);
} /* fim da função main */
```

### FIFO's – Exemplo de um dicionário cliente-servidor

Código do servidor ( 9 )

Tratamento do sinal SIGINT

- ➔ Termina o servidor de uma forma adequada libertando os recursos detidos
- Feito da mesma forma como se se tivesse recebido ordem para terminar ("fim" escrito directamente para o FIFO do servidor)

```
void sig_int(int i) {
    fprintf(stderr, "\nServidor de dicionario a terminar
                (interrompido via teclado)\n\n");
    close(keep_alive_fd);
    close(s_fifo_fd);
    unlink(SERVER_FIFO);
    exit(EXIT_SUCCESS); /* termina o processo */
}
```

### FIFO's – Exemplo de um dicionário cliente-servidor - 2

Modificação para evitar o recurso à função fcntl - Cliente

Abre o FIFO do servidor para escrita  
 Cria o FIFO para receber a resposta  
 Abre o FIFO das respostas para leitura

Repete durante uma certa condição

- Obtém palavra a traduzir
- Constrói pergunta = palavra + nome do FIFO para a resposta
- Envia a pergunta (escreve no FIFO do servidor)
- Fica à espera da resposta (efectua uma leitura no FIFO para a resposta)
- Fecha o FIFO do servidor
- Fecha o FIFO para as respostas
- Remove o FIFO das respostas

### FIFO's – Exemplo de um dicionário cliente-servidor

Parte do algoritmo de abrir o FIFO das respostas (bloqueante logo de origem)

Após o envio da pergunta ao FIFO das respostas

- Eventualmente bloqueia temporariamente
- Não levanta problemas porque o servidor já tem a identificação desse FIFO e irá abri-lo para escrita e escrever nele

A abertura é efectuada dentro do ciclo principal

- Apenas é necessário testar se o FIFO já estava a aberto pois a sua abertura é agora efectuada dentro do ciclo principal  
(Ou então abrir e fechar mas essa opção é desnecessariamente pesada)

A única alteração necessária é no cliente. A modificação algorítmica é apresentada a seguir. A sua implementação (trivial) é um exercício (que se recomenda)

### FIFO's – Exemplo de um dicionário cliente-servidor

Parte do algoritmo que abre o pipe

Não bloqueante, após o envio da pergunta, dentro do ciclo principal

Repete durante uma certa condição

Obtém palavra a traduzir

Constrói pergunta = palavra + nome do FIFO para a resposta

Envia a pergunta (escreve no FIFO do servidor)

**Abre o FIFO das respostas para leitura**

Fica à espera da resposta (efectua uma leitura no FIFO para a resposta)

Verifica se o FIFO já estava aberto (flag, etc.)

Se não estava

Abre o FIFO como bloqueante

(Eventualmente bloqueia temporariamente)

Assinala (flag? etc.) que o FIFO está aberto

## FIFO's – Considerações

Recomendações para a resolução de problemas com *named pipes*

- Planear cuidadosamente a interacção entre os processos
- Identificar o modelo de comunicação mais apropriado (cliente-servidor, caixa de correio, diálogo, etc.)
- Identificar quais os pipes que vão existir e atribuir papéis a cada processo
  - Quem cria/apaga cada pipe
  - Quem lê/escreve em cada pipe
- Responder à pergunta – como sabem os processos para onde enviar a informação?
- Planear cuidadosamente a ordem pela qual as operações de abertura e escrita/leitura são feitas para evitar esperas mútuas
- Definir um protocolo de comunicação entre os intervenientes (significado e estrutura da informação)
  - Qualquer coisa pode ser enviada pelo pipe, mas tanto quem envia como quem recebe têm que saber o que é que está a ser enviado.
  - Se forem enviadas coisas diferentes, a ordem pela qual são enviadas e recebidas é fundamental
- Prever mecanismos de recuperação de mensagens erradas (fora de ordem? incompletas?), clientes ou servidor que não respondem (timeouts?) e terminação ordenada de clientes.
- Não deixar pipes por apagar

**Importante: Soluções confusas e pouco intuitivas normalmente estão erradas**

# Sistemas Operativos

2020 – 2021

## Conceitos fundamentais sobre sistemas operativos

## Tópicos

Constituição e objectivos dos sistemas operativos

Evolução, tipo, estrutura e arquitectura

Tópicos de implementação de funções sistema

Estado de processos

### Bibliografia específica:

- *Fundamentos de Sistemas Operativos*; 3<sup>a</sup> Ed.; Marques & Guedes  
Capítulos 1,2 e parte do 3
- *Operating Systems Concepts* ; Silberschatz & Galvin  
Capítulo 1

## Fundamentos de sistemas operativos

Este conjunto de slides está organizado em 4 partes

- 1 – Objectivos, constituição e tipos de sistema operativos
- 2 – Arquitecturas de sistemas operativos
- 3 – Implementação de funções sistema e mecanismo de interrupções
- 4 – Processos e *threads* (Nota: também já abordado antes)

## Importante

Esta matéria é maioritariamente descritiva e em parte apresentada de forma interactiva com os alunos (os que estão presentes).

Assim, estes slides são uma versão pálida do conteúdo da matéria deste capítulo. É muito importante consultar a bibliografia.

Existem alguns slides neste ficheiro com perguntas de revisão que podem ser usadas para auto-aférição de conhecimentos

- Essas perguntas são maioritariamente de compreensão. Tome nota que nos exames sairão também perguntas de outros tipos (ex., aplicação a novas situações)
- Nos seus treinos/revisão, não se limite a repetir o conteúdo fornecido pelo professor. Nos exames pretende-se avaliar o aluno e não o professor.
- Estas perguntas são para treinar os seus conhecimentos e a sua eficácia de resposta em exames. Se não for coerente, a resposta de pouco serve. Tente dar a resposta a um colega ou amigo que não frequente a disciplina. Se ele perceber (e se o conteúdo da resposta for correcto) está a sair-se bem.
- A matéria de algumas perguntas pode parecer que “não estão nos slides”. Se lhe acontecer isso é sinal que não foi às aulas nem leu a bibliografia.
- De forma alguma deve ser assumido aqui que as perguntas do exame são um subconjunto destas, ou que seguirão o mesmo formato (estas são para revisão). A parte teórica do exame poderá ser de escolha múltipla (mesmo conteúdo, formato diferente).

## Fundamentos de sistemas operativos

### Parte 1 – Objectivos, constituição e tipos de sistema operativos

- A retirar desta parte
  - Constituição típica de sistemas operativos. Componentes dos SO
  - Interacção entre SO – hardware – utilizador
  - Tipos de sistemas operativos

## Sistema Operativo: definição geral

### O que é um sistema operativo:

- Programa ou conjunto de programas
- Serve de intermediário entre o equipamento físico e o “utilizador”
- Efectua a gestão de tudo o que há na máquina

“Utilizadores” tanto podem ser pessoas como programas: ambos utilizam os serviços do sistema operativo

- Um *utilizador-pessoa* utiliza a máquina (gerida pelo SO) para obter um determinado serviço
- Um *utilizador-programa* utiliza os recursos proporcionados pelo sistema para concluir um determinado algoritmo

Para simplicidade de expressão, nestes slides iniciais:

- Utilizador – significa utilizador / operador humano
- Aplicação, programa, processo utilizador – significa um programa a correr em cima do (exterior ao) sistema operativo  
(Obs.: mais adiante ver-se-á que processo e programa são conceitos distintos)

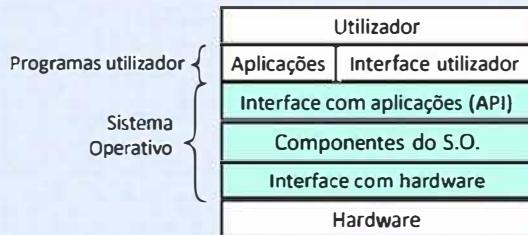
### Objectivos gerais dos sistemas operativos

- **Efectuar a gestão da máquina**
  - Gestão dos recursos de *hardware*
  - Implementação de políticas de optimização
- **Virtualizar o equipamento electrónico (hardware)**
  - Obtém-se uma máquina virtual mais fácil de operar com operações uniformizadas
- **Prestação de serviços e ambiente de trabalho**
  - Disponibilização dos recursos da máquina em serviços úteis aos “utilizadores”
    - Ao utilizador-pessoa: é-lhe disponibilizado uma ambiente de trabalho e um conjunto de ferramentas para poder usar a máquina
    - Ao utilizador-programa: é-lhe fornecido um ambiente de execução e um conjunto de recursos acessíveis através de uma interface de programação – API (um conjunto de funções que podem ser invocadas)

### Objectivos gerais dos sistemas operativos

- **Garantir a estabilidade da máquina**
  - Determinar a utilização adequada dos recursos
    - Verifica a validade das operações pedidas
    - Verifica o cumprimento das regras e permissões
  - Garantir que os utilizadores não interferem uns com os outros
  - Garantir que os programas não interferem uns com os outros  
(Obs.: “interferir” ≠ “interagir”)
  - Proporcionar mecanismos de detecção e recuperação de falhas

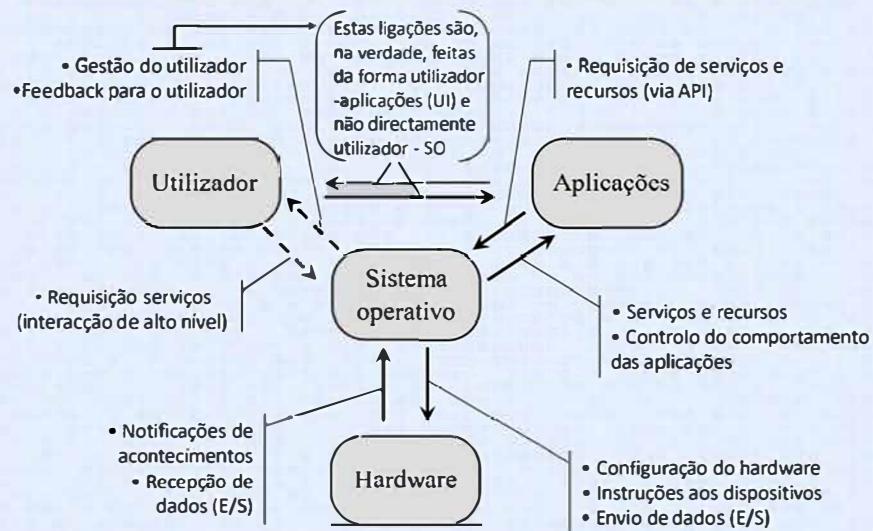
### Posicionamento do sistema operativo



A interface com o utilizador é uma aplicação a correr por cima do SO.

Alguns sistemas estão tão associados a uma interface específica que esta se confunde com mais um componente do sistema

### Entidades que interagem com o sistema operativo



### Interacção Utilizador – Sistema Operativo : a *shell*

- O utilizador apercebe-se do sistema de uma forma indirecta: trata-se de uma entidade que de alguma forma faz com que a máquina funcione
- O utilizador interage com o sistema tipicamente através de uma interface (ex: *explorer*, *bash*, *command.com* etc.)
- Normalmente a interface é considerada como sendo externa ao sistema
- O objectivo da interface é a de facilitar a utilização da máquina de uma forma consistente e fácil mesmo para utilizadores com poucos conhecimentos acerca de computadores

### Interacção Aplicações – SO : o API / Chamadas sistema

- O sistema operativo é visto como um conjunto de **recursos e serviços**
- Esses recursos e serviços constituem uma máquina virtual (ambiente de execução) de que o processo dispõe para se executar
- O programa obtém os recursos ou serviços através de **chamadas ao sistema**. O conjunto de funções sistema acessíveis às aplicações constitui o API do sistema
- O facto de sistemas diferentes terem API diferentes faz com que as aplicações para um sistema operativo normalmente tenham que ser reescritas para correr noutro sistema
  - Se apenas a implementação do API variar pode ser apenas necessário recompilar o programa (sem modificar o código fonte)
- Por vezes a invocação do API é feito indirectamente através de funções biblioteca que podem ser fornecidas com o ambiente de desenvolvimento ou com o próprio sistema

### Interacção Aplicações – SO : o API / Chamadas sistema

- As chamadas ao sistema são uma parte do sistema que tem que ser criteriosamente implementada. Nomeadamente tem que se garantir que:
  - As aplicações devem ter acesso a essas funções sem serem obrigadas a conhecer o endereço exacto onde residem (para permitir actualizações ao código do SO mantendo a compatibilidade)
  - As aplicações não devem poder saltar para pontos arbitrários dentro do código das funções sistema
  - As aplicações não devem ter possibilidade de alterar o código das funções

Normalmente as funções sistema são implementadas através de *traps* (detalhes mais adiante)

### Interacção Hardware – SO : os Gestores de periféricos

- O sistema é visto como a única entidade responsável pela máquina. O hardware não interage directamente com o utilizador nem com as aplicações
- Tarefas do sistema operativo quanto ao hardware
  - Configurar adequadamente todo o hardware no arranque do sistema
  - Controlar o estado do hardware, validando as operações efectuadas sobre ele e efectuando a gestão de erros
  - Receber notificações enviadas pelo hardware (periféricos)  
As notificações são processadas e reencaminhadas para as aplicações destinatárias

Existe normalmente um conjunto de sub-programas do sistema operativo para estas tarefas: **gestores de periféricos (device drivers)**

## Tipos de sistemas operativos

Os sistemas operativos podem ser classificados segundo:

- A relação entre o tempo da máquina e o tempo real
  - **Sistemas de tempo virtual**
    - O tempo real não é importante para a actividade do computador
  - **Sistemas de tempo real**
    - O tempo real é relevante para o sistema
    - Objectivo: garantir que as tarefas sejam cumpridas em intervalos de tempo real predeterminados
- Segundo o número de processadores e natureza da sua utilização
  - **Sistemas paralelos**
    - Memória e relógio partilhados por todos os μP
  - **Sistemas distribuídos**
    - Memória e relógio independentes para cada μP

## Algumas questões de revisão

Algumas perguntas para revisão e aferição de conhecimentos

- Afinal, o que é um sistema operativo?
- Que pontos de vista diferentes existem quando ao que é ou deixa de ser um sistema operativo?
- Que entidades interagem com um sistema operativo
- Em que condições um programa (uma sequência de instruções qualquer) pode aceder ao hardware?
- O utilizador (ser humano) interage directamente com o sistema operativo? De que forma?
- Como é que o hardware interage com o sistema operativo?
- Quais os objectivos principais de um sistema operativo moderno?
- O que é um API?
- Como é que os programas-utilizador utilizam os recursos do sistema operativo?
- O que é o isolamento de processos?
- O que é isso de abstrair o hardware? Dê exemplos.
- O que é que impede um programa-utilizador de tomar o controlo da máquina?
- Que tipos de sistema existem. Dê exemplos de aplicação para cada um.

## Fundamentos de sistemas operativos

### Parte 2 – Arquitecturas de sistemas operativos

- A retirar desta parte
  - Arquitecturas de sistemas. Para cada uma:
    - Características principais
    - Vantagens e desvantagens

## Arquitectura dos sistemas operativos

Filosofia de implementação de sistemas operativos:

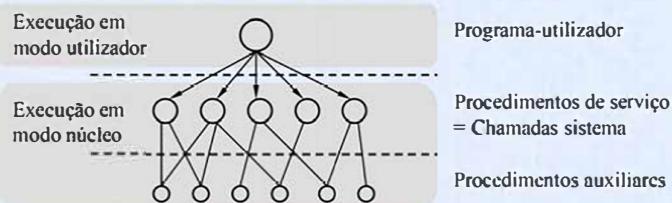
(*arquitectura* do S.O.)

- Sistemas **monológicos**
- Sistemas em **camadas**
- **Máquinas virtuais**
- Cliente-servidor (**microkernel**)

## Arquitectura dos sistemas operativos

### Sistemas monolíticos

- Conhecido como “*Big Mess*”
- Caracteriza-se pela ausência de estrutura modular
- O sistema operativo é constituído por um conjunto de procedimentos ou funções que se podem chamar uns aos outros indiscriminadamente
- Não existe encapsulamento funcional



## Arquitectura dos sistemas operativos

### Sistemas monolíticos

#### Vantagens

- Ligeiramente mais rápidos (dependente da implementação)

#### Desvantagens

- Falta de estrutura leva a dificuldade de implementação
- Dificuldade em isolar responsabilidades no sistema

#### Conceitos importantes:

- **Núcleo**
- **Modo de execução:** utilizador e núcleo

Protecção a nível de hardware quanto a

- Execução de um conjunto de instruções privilegiadas
- Visibilidade sobre regiões de memória

### Núcleo do sistema operativo

#### **Núcleo do S.O.**

- Conjunto de funcionalidades (camadas) do sistema onde se incluem as funções mais importantes e mais privilegiadas
- Deve ser sempre executado no modo mais privilegiado disponível na máquina

(Algumas das) justificações para esse agrupamento de funções

- Necessidade de executar instruções privilegiadas (ex.: gestão do hardware)
- Necessidade de protecção quanto a interferências do resto do sistema ou programas utilizador
- Execução prioritária (em resposta a acontecimentos internos ao sistema ou gerados pelo hardware)

### Núcleo do sistema operativo

#### Funções do núcleo (implementações típicas)

- Gestão de processos
  - Controlo das interrupções
  - Escalonamento
  - Despacho
  - Mecanismos de sincronização
- Gestão de memória

## Arquitectura dos sistemas operativos

### Sistemas em camadas

- Manutenção facilitada
  - Tarefas e responsabilidades do S.O. atribuídas a uma camada específica
- Nem sempre corresponde inteiramente à realidade de alguns sistemas operativos, no entanto, facilita a compreensão e utilização dos recursos do sistema
- Bastante didáctico

### Camadas

- Gestão de processos
- Gestão de memória
- Comunicação e E/S
- Sistemas de ficheiros
- Interface do sistema (*funções sistema e interpretador de comandos*)
- Aplicações

### Exemplos

- MULTICS
- THE

## Arquitectura dos sistemas operativos

### Sistemas em camadas – Tarefas de cada camada:

#### Gestão de processos

- Criação terminação de processos
- Escalonamento e despacho de processos
- Comutação de processador
- Tratamento de interrupções
- Suporte para sincronização de processos

#### Gestão de Memória

- Controlo da utilização de memória física e memória virtual

#### Comunicação e E/S

- Mecanismos de comunicação entre processos
- E/S - caso particular de comunicação (comunicação com o exterior)

## Arquitectura dos sistemas operativos

### Sistemas em camadas

#### Sistema de ficheiros

- Virtualização dos dispositivos de massa

#### Interface sistema – duas partes distintas:

- Funções sistema - Serviços proporcionados pelo sistema
- Interpretador de comandos
  - Aplicação que corre em modo utilizador
  - Interpreta os comandos do utilizador segundo uma linguagem simples

#### Aspectos gerais (comuns a todas as camadas)

- Optimização do sistema
  - Envolve : memória, E/S, comutação dos processos
- Protecção
  - Impedir a interferência com outros processos ou com o próprio sistema

## Arquitectura dos sistemas operativos

### Sistemas em camadas

- Principal dificuldade
  - Nem sempre é fácil determinar a hierarquia das camadas
- Exemplo
  - Deverá a gestão de memória deve “vir depois” do sistema de ficheiros para a implementação de memória virtual ? Ou deverá a memória virtual ser apoiada na gestão de memória ?
- Idealmente
  - Se o hardware o suportar, cada camada terá o seu próprio modo de execução
    - A camada mais privilegiada é a mais baixa (gestão de processos)

#### Actualmente

Alguns sistemas operativos começaram por ser monolíticos, tendo evoluído para uma arquitectura mista (poucas camadas, mas não monolítico)

## Arquitectura dos sistemas operativos

### Máquinas virtuais

- Evolução sobre os sistemas de tempo partilhado
  - Ideia base – Separação dos dois conceitos
    - Multiprogramação
    - Virtualização do hardware
- Os níveis mais baixos do sistema simulam a existência de várias máquinas (normalmente iguais), podendo cada uma ter o seu próprio sistema operativo
  - Grande parte do(s) sistema(s) operativo(s) é movido para um nível superior
    - Ganha-se simplicidade e estabilidade
  - Torna-se possível “ter” várias máquinas numa só

Exemplo: IBM VM/370

## Arquitectura dos sistemas operativos

### Máquinas virtuais

#### Aplicações vantajosas

- Desenvolvimento de novos sistemas operativos
  - A instabilidade do sistema em desenvolvimento não afecta o resto da máquina

#### Principais desvantagens

- Difícil de implementar em algumas plataformas (hardware)
  - Ex.: Os modos de execução podem não ser suficientes

#### Conceitos importantes

- Modo núcleo
- Modo núcleo virtual
- Modo utilizador virtual

## Arquitectura dos sistemas operativos

### Máquinas virtuais

#### Hypervisor

- Sistemas (operativo)
  - Trata-se de um sistema operativo não-genérico: não se destina ao mesmo uso dos sistemas operativos “habituais” (estes sim, são genéricos: não têm uma finalidade específica)
  - O hypervisor está focado na funcionalidade de virtualização e não inclui outro tipo de funcionalidade (por exemplo, gestão de utilizadores).

## Arquitectura dos sistemas operativos

### Máquinas virtuais

#### Hypervisor tipo I (“nativos”)

- Corresponde a um sistema que fica totalmente abaixo das máquinas virtuais. O hardware é totalmente gerido pelo hypervisor. Trata-se da concretização do verdadeiro conceito de máquinas virtuais. Exemplo: Xen

#### Hypervisor tipo II (“hóspedes”)

- O virtualizador corre em cima, ou pelo menos, ao mesmo nível que o sistema. Já não gera directamente o hardware. Exemplo: VirtualBox, VMWare Player

## Arquitectura dos sistemas operativos

### Máquinas virtuais

#### Hypervisor tipo I e tipo II

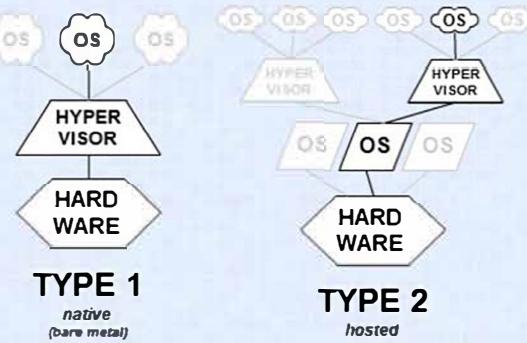


Imagen: Wikimedia

## Arquitectura dos sistemas operativos

### Máquinas virtuais

#### Hypervisor tipo I ("nativos")

- Vantagens: melhor gestão do hardware (eventualmente melhor performance disponibilizadas às máquinas virtuais). Independência de um sistema operativo genérico
- Desvantagens: menor facilidade de interacção com as máquinas virtuais por parte dos utilizadores. Cenário típico = o acesso é remoto através de outra máquina (remote desktop, VNC, etc.)

#### Hypervisor tipo II ("hóspedes")

- Vantagens: Menor investimento e maior facilidade de integração com um ambiente previamente existente (o virtualizador é apenas mais um programa a correr)
- Desvantagens: dependência de um sistema genérico sobre o qual corre. Eventualmente menos performance disponível para as máquinas virtuais. Menor poder de gestão do hardware

## Arquitectura dos sistemas operativos

### **Cliente-servidor (*Micro-Kernel*)**

Divisão lógica do sistema operativo em unidades funcionais

- Tem-se um núcleo muito reduzido e simplificado (*micro-kernel*)
  - Os serviços do sistema são proporcionados por programas que correm de forma semelhante aos programas utilizador
  - A requisição de serviços é efectuada através de mensagens entre processos
- As tarefas do núcleo resumem-se a
  - Escalonamento e despacho
  - Gestão da comunicação entre programas (processos)
- Ficam fora do núcleo (entre outros, variando com as implementações)
  - Gestão de memória
  - Gestão de ficheiros

## Arquitectura dos sistemas operativos

### **Cliente-servidor (*Micro-Kernel*)**

#### Vantagens

- Núcleo simplificado e rápido
- Sistema mais estável
- Facilmente adaptável a sistemas distribuídos
- Portabilidade melhorada

#### Exemplo

WindowsNT

#### Conceitos importantes

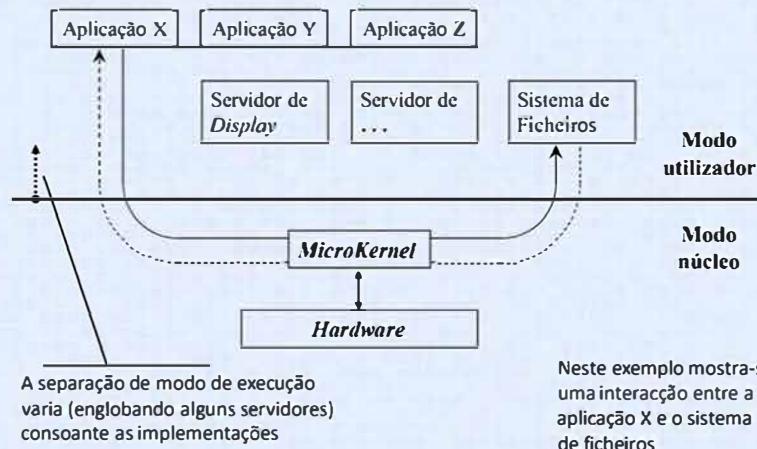
##### Filosofia de construção

- Política
  - De que maneira são utilizados os recursos → modo utilizador
- Mecanismo
  - Método de manipulação dos recursos → modo núcleo

Actualmente: tendência para mover o máximo possível para as camadas superiores do S.O.

## Arquitectura dos sistemas operativos

Esquema de um sistema genérico com arquitectura cliente-servidor



## Algumas questões de revisão

- O que é o núcleo? Como pode o identificar facilmente?
- O núcleo corresponde ao sistema operativo? Justifique e dê exemplos.
- Quais as arquitecturas de sistemas operativo que conhece? Descreva-as detalhadamente e de forma coerente. Para cada uma indique uma vantagem e uma desvantagem.
- Qual é melhor? Um sistema monolítico ou um sistema em micro-kernel? (junte-se ao debate dos Srs. Tanenbaum e Torvalds e tente chegar a uma conclusão sua justificada coerentemente).
- Identifique exemplos de aplicação em sistemas de máquinas virtuais.
- Qual a diferença principal entre um sistema de tempo virtual e um sistema de tempo real?
- Qual a diferença principal entre um sistema paralelo e um distribuído?
- Aplicaria um sistema de tempo virtual para controlar o sistema de direcção de um missil? Porquê?

## Fundamentos de sistemas operativos

### Parte 3 – Mecanismo de interrupções e implementação de funções sistema

- A retirar desta parte
  - Mecanismo genérico de interrupções
  - Interrupções – Funcionamento e utilização no SO
  - Excepções – Funcionamento e utilização no SO
  - *Traps* – Funcionamento e utilização no SO
  - Implementação das funções sistema: chamada e funcionamento.

## Interrupções – Importância para os sistemas operativos

O sistema operativo precisa de um mecanismo que o possa notificar de forma assíncrona (sem estar especificamente à espera) da ocorrência de muitas situações

### Exemplos

- Fim de operação de E/S
- Houve uma tentativa de utilização de um endereço inválido
- O processador encontrou uma instrução que não conhece
- É preciso ir buscar um bloco de memória ao disco e colocá-lo em memória
- É preciso transferir a execução do código de um processo para o código do SO (ex., foi chamada uma função sistema)
- Terminou o tempo máximo concedido a um processo, sendo a vez de outro se executar agora.

► **O mecanismo de interrupções vai permitir responder a estas questões e constitui um pilar fundamental na implementação dos S.O.**

## **Interrupções – Importância para os sistemas operativos**

Existem três tipos distintos de “interrupções”

- **Interrupções**
    - Geradas por periféricos
    - Essencial para muitas operações de E/S
  - **Traps**
    - “Interrupções” causadas (pedidas) pelo próprio programa
    - Úteis para a implementação de chamadas ao sistema
    - Exemplo (x86): INT 21H Invoca a rotina de serviço 21H
  - **Excepções**
    - “Interrupções” causadas pelo próprio processador
    - Essenciais para o tratamento de erros pelo sistema operativo
    - Exemplos: Endereços inválidos, Operações inválidas, ...

**Obs.: Em alguns processadores esta distinção é apenas conceptual**

**Interrupções – Importância para os sistemas operativos**

Modo de funcionamento das interrupções (simplificado)

- 1 – Um dispositivo (controlador) activa a linha IRQ
  - 2 – O processador interrompe o curso normal de execução e executa para uma rotina de serviço de interrupção (ISR)

O endereço da rotina de serviço encontra-se numa tabela numa zona fixa (conhecida) na memória = *Tabela de interrupções*

- 3 – A rotina ISR averigua a causa da interrupção

  - A rotina ISR irá desencadear a passagem ao estado executável dos processos bloqueados à espera da conclusão de operações E/S  
(Tópico a desenvolver mais adiante)
  - A rotina ISR deve residir no núcleo do S.O.
  - ➔ A comutação de processos é conseguida à custa de uma ISR associada a uma interrupção de relógio

### Interrupções – Importância para os sistemas operativos

Exemplo – Tabela (parcial) de interrupções – Arquitectura 80x86

0000-0003	00	Divisão por zero
		• • •
0008-000B	02	Interrupção não mascarável - NMI_INT
		• • •
0020-0023	08	Relógio - TIMER_INT
0024-0027	09	Teclado - KB_INT
		• • •
0040-0043	10	Serviços da placa gráfica - VIDEO_IO
		• • •
0084-0087	21	Chamadas sistema do DOS
		• • •
0094-0097	25	Leritura de disco via BIOS
0098-009B	26	Escrita em disco via BIOS

### Interrupções – Importância para os sistemas operativos

Funcionamento das interrupções (mais detalhado)

- Existem várias interrupções possíveis
  - Geralmente cada interrupção está associado a um dispositivo  
(Mas o mesmo dispositivo pode ter várias, e também podem ser partilhadas)
  - Para cada interrupção existe uma entrada na tabela de interrupções
  - Cada entrada possui o endereço da rotina de serviço da interrupção correspondente
- A ISR deve salvaguardar o contexto de hardware no início e restaurá-lo no final
  - Durante a execução da ISR podem ocorrer outros pedidos de interrupções
  - Se a ISR for lenta não deve inibir as interrupções
  - Algumas arquitecturas possuem um esquema de interrupções hierarquizado que permite inibir apenas as interrupções de prioridade menor

## Interrupções – Importância para os sistemas operativos

### Funcionamento geral das interrupções

- O atendimento a uma interrupção funciona de modo similar a uma instrução CALL, em que o endereço de retorno é colocado na pilha
  - No entanto, um CALL é uma instrução explicitamente colocada no código
  - As interrupções e as exceções não são explicitas. Podem ocorrer a qualquer instante (a não ser que o SO as tenha desabilitado)
  - As traps são explicitas. Correspondem (no Intel) a uma instrução INT xx que se encontra no código
  - O atendimento de uma interrupção pode implicar a passagem a modo núcleo

### Funcionamento das Traps

- Análogo ao das interrupções
  - São um tipo de exceção
- Análogo a um CALL
  - Com a diferença que não é preciso conhecer o endereço para onde se está a saltar. Apenas o número da interrupção. O endereço destino está na tabela de interrupções

## Interrupções e chamadas a funções sistema

### Implementação das chamadas ao sistema

- As chamadas ao sistema são a única forma de um processo requisitar os serviços do sistema.
- Implica passar a execução de dentro do código no processo para o código no sistema.

### Problemas

- Como saber para onde se deve saltar (onde está o código da rotina a invocar)? Um novo kernel implicará a mudança de sítio das rotinas. No entanto, observa-se que os programas continuam a correr.
- Como garantir que a transição ocorre para o início da rotina e não para outro lado qualquer (por exemplo, para tentar evitar passar por cima de uma verificação no início da rotina)?
- Como saltar de uma zona de código menos privilegiada (código em modo utilizador) para uma zona de código mais privilegiado (código do SO em modo núcleo)? Os processadores normalmente não permitem isso.

## **Interrupções e chamadas a funções sistema**

Implementação das chamadas ao sistema com interrupções:

A utilização do mecanismo de interrupções (mais especificamente *traps*) soluciona estes problemas

- Saber qual o endereço da rotina

Aquilo que é conhecido e nunca varia é o número da interrupção (*trap*). O endereço da rotina está na tabela, não é conhecido pelo processo e pode mudar à vontade de kernel para kernel

- Protecção contra saltos para endereços indevidos

A *trap* salta sempre para o endereço que está na tabela de interrupções, que será o início da rotina a invocar. Essa tabela estará protegida contra escrita em modo utilizador

- Passagem de modo utilizador para modo núcleo

Por saltar sempre para um endereço que é controlado pelo SO e não pode ser mudado por código em modo utilizador, o processador permite que se passe de modo menos privilegiado para modo mais privilegiado através das *traps*

## **Interrupções e chamadas a funções sistema**

Implementação das chamadas ao sistema através das *traps*

- Convencionou-se que uma determinada “interrupção” (*trap*) corresponde a uma determinada chamada ao sistema
- Os parâmetros da chamada são colocados principalmente em registos do processador, são bem especificados e não variam.
- Normalmente todas as funções sistema passam pelo mesmo *trap* (INT xx). Para seleccionar qual a função a invocar é usado um registo (cada valor corresponde a uma função)
- O endereço da chamada ao sistema será colocado na entrada (da tabela) correspondente à “interrupção” convencionada
- Em versões posteriores do mesmo S.O. a implementação da função sistema pode variar, bastando colocar o novo endereço na entrada convencionada na tabela de interrupções

## Interrupções e chamadas a funções sistema

Implementação das chamadas ao sistema através das *traps*

- O conhecimento acerca de que interrupções são usadas para que funções não faz parte do compilador nem é o programador que tem que lidar com esse assunto
  - Já existe código que faz essa “ponte” entre a aplicação e o sistema
  - O código responsável por traduzir uma chamada a função na linguagem do programador a uma INT (ou equivalente na arquitectura alvo) está codificado em funções biblioteca já feitas
  - Essa biblioteca será disponibilizada para uma determinada arquitectura e sistema operativo
  - Essa biblioteca junta-se ao programa como qualquer outra (biblioteca C standard, ncurses, etc.). Normalmente, o *linker* já usa essa biblioteca automaticamente.

Ou seja

- Uma biblioteca que acompanha o SO faz a ponte entre uma chamada na linguagem de alto nível (ex.: open(...)) e os INTs para esse sistema) (MSDOS: 21H, Windows: 2EH, Linux: 80H). A adaptação parâmetros e registo já está tratada e os parâmetros da chamada são colocados nos registos adequados. Essa biblioteca é de uso transparente ao programador.

## Interrupções e chamadas a funções sistema

**Exemplo: MSDOS**

- Existe um único ponto de acesso ao sistema operativo: **INT 21H**
  - O valor 21H é uma convenção
    - Todos os programas assumem que é este valor.
    - Se numa versão nova do sistema passasse a ser outro valor, nenhum programa correria
- Existe mais que uma função sistema
  - A selecção da função específica é feita através do registo **AH**
    - Cada função tem um valor para AH convencionado
- Os parâmetros para a função são colocados em registos (AX, BX ....)
  - Os parâmetros também são convencionados

### Funções sistema do MSDOS (INT 21h)

Tabela parcial de funções sistema acessíveis via INT 21H

09	Apresenta uma cadeia de caracteres (asciiz)
...	
2A	Obtém a data do sistema
2B	Modifica a data do sistema
...	
30	Obtém a versão do sistema
...	
3C	Cria um ficheiro (via file handles)
3D	Abre um ficheiro (via file handles)
...	
48	Atribui memória (em parágrafos de 16 bytes)
49	Liberta memória previamente atribuída
...	
4B	Carrega e executa um programa

### Interrupções e chamadas a funções sistema

#### Exemplo

A função (ambiente MSDOS) ( reparar que não é a função biblioteca fopen)

```
open(<nome do ficheiro>,<modo de abertura>)
```

Corresponde à função

3DH da INT 21H

Entradas	AH = 3DH	- Número da função
	DS:DX	- Ponteiro para o nome em ASCIIZ
	AL	- 0=leitura; 01=escrita; 2 =leitura e escrita
Saídas	CF	- Se for 1 então houve erro
	AX	- Handle (id) do ficheiro

## Interrupções e chamadas a funções sistema

Exemplo: Abertura de um ficheiro

<pre> num_han DW      ? arquivo DB      'programa.xyz',0                 ... MOV     AH,3DH      : função open LEA     DX,arquivo   : aponta para o nome MOV     AL,0         : modo read-only INT     21H         : invoca o DOS JC      erro        : se houve erro, c=1 MOV     num_han,AX   : guarda o handle                 ... erro:    ... </pre>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Parte de um programa em C</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">Código correspondente para chamar a função sistema</div>
---	--

## Interrupções e chamadas a funções sistema

Então tem-se que:

- A passagem ao SO é feita via mecanismo de *traps*
- Os valores de registo indicam qual a função específica que se pretende e parâmetros para essa função

*Significa isto que o compilador terá que saber estes pormenores para capilar uma chamada a uma função para o INT correspondente?*

-> A resposta é Não.

Como se faz, então? -> prox. slide

## Interrupções e chamadas a funções sistema

*O compilador terá que saber os pormenores acerca de como se invocam as funções sistema para cada plataforma/SO ?*

**Obviamente que não. Seria impensável**(desafio: porquê “impensável”?)

O que se passa é:

- O código contendo a instrução INT e a manipulação dos registo na entrada e na saída da chamada ao sistema já vem implementada numa **função adaptadora** implementada como as funções bibliotecas.
- Quando o programador “invoca” a função sistema está, na verdade, a invocar uma dessas funções adaptadoras, a qual tem os pormenores específicos a essa plataforma (SO) acerca de como se faz a passagem para a função sistema
- Essas funções adaptadoras já vem implementadas em ficheiros de código objecto (tal como as bibliotecas *standard*) disponibilizadas como bibliotecas para a plataforma em questão
- Em momento algum o compilador tem conhecimento acerca de como funcionam as chamadas ao sistema.

## Interrupções e chamadas a funções sistema

Atendimento de uma *Trap* – O que o processador faz:

INT 21H        (21H é apenas a título de exemplo)

- Guarda o registo de *flags* na pilha
- Guarda o endereço actual (CS:IP) na pilha
- Obtém o endereço guardado na entrada 21H na tabela de interrupções (256 entradas, cada uma ocupa 4 bytes)
- Salta para o endereço obtido (como se fosse um CALL)
  - A rotina de atendimento entra em execução, terminando com a instrução IRET, que repõe o processador a executar a instrução no endereço CS:IP guardado na pilha, restaurando também as *flags* (quase todas)

Alguns pormenores foram omitidos por simplicidade

## Interrupções e chamadas a funções sistema

### Mecanismo de chamadas ao sistema do MSDOS

- Tem pouca ou nenhuma protecção
  - Qualquer programa pode alterar os vectores de interrupção (ponteiros armazenados na tabela de interrupções)  
Exemplo de aplicação: um vírus informático
    - Alterar a função 3DH da INT 21H (abrir ficheiro)
      - Sempre que um ficheiro executável for aberto, vai ser modificado (nova “funcionalidade”)
      - Depois da modificação feita, o código original da função 3DH é executado
      - Esta nova “funcionalidade” permanece desconhecida para os outros programas
    - Qualquer programa pode saltar para o meio do código de uma função sistema
      - Basta obter o ponteiro armazenado na tabela e saltar mais para diante
- Exemplo de aplicação:
  - Evitar um conjunto de testes de validação ou segurança

## Interrupções e chamadas a funções sistema

### Resolução do problema anterior de falta de protecção

- Utilização do mecanismo de interrupções juntamente com o mecanismo de modos (privilégios) de execução
  - O atendimento da interrupção (ou *trap*) desencadeia (directa ou indirectamente) a mudança de modo de execução (utilizador → protegido)
  - A gestão de memória está directamente ligada ao modo de execução
  - A rotina de atendimento a uma interrupção (neste caso uma chamada ao sistema) residirá numa zona de memória não directamente acessível aos processos em modo utilizador
  - A própria tabela de interrupções residirá numa zona de memória protegida

Os problemas apontados atrás ficam resolvidos

**Os mecanismos de protecção variam de processador para processador**

## Interrupções e chamadas a funções sistema

### Implementação de Chamadas Sistema

Algumas observações

- O modelo de implementação de chamadas ao sistema através de traps + mudança de modo de execução corresponde ao método geral
- No entanto, os pormenores reais podem variar consoante:
  - Mecanismos de protecção disponibilizados pelo processador
  - Mecanismo de gestão de memória disponibilizado
  - Implementação do sistema operativo

Exemplo

- Na arquitectura x86, a mudança de modo de execução é feita indirectamente: a protecção está associada à zona de memória que está em execução

## Interrupções e S.O. – Excepções

### Funcionamento das excepções

- Funcionamento geral análogo ao das interrupções e das *traps*
- Não é desencadeada a pedido do programa
- A interrupção é desencadeada pelo próprio processador
- Causas: situação de “erro” (situações de excepção)
  - Exemplos
    - Operações inválidas (*op-code* desconhecidos)
    - Tentativa de acesso a memória de outros processos
    - Falta de página (*Page-fault*)  
Mecanismo essencial para a implementação de memória virtual  
(Tema a desenvolver mais adiante)
    - Tentativa de execução de uma instrução privilegiada em modo utilizador
  - A causa da excepção tanto pode estar relacionada com o S.O. propriamente dito como com o processo que estava em execução

## Interrupções – Exemplo de hardware

Suporte pelo processador: Exemplo do processador Pentium e seguintes

Registo EFLAGS – Contém as *flags* do processador

Entre outras,

- IOPL (*I/O privilege level*)
  - Se estiver a "1" faz com que o processador gere uma excepção em todos os acessos a dispositivos I/O (instruções IN e OUT. Exemplo: OUT DX,AL)
  - Essa excepção é apanhada pelo S.O. e a operação é traduzida + validada e executada (ou recusada) pelo S.O.
  - Consegue-se assim impedir que os programas em modo utilizador consigam comunicar com os periféricos
- IF (*Interrupt enable flag*)
  - Se estiver a "1" o processador aceita interrupções externas (dos dispositivos)
- TF (*Trap flag*)
  - Se estiver a "1", gera uma interrupção após a execução de cada instrução.
  - Utilizado para *debugging*

## Algumas questões de revisão

- Como é que os programas-utilizador encontram as rotinas do sistema operativo?
- O que é o modo núcleo? Em que difere do modo utilizador?
- É possível implementar um Linux ou Windows moderno num 8086? Porquê?
- que é uma *trap* e para que serve?
- Como é que o sistema detecta que um programa está a tentar usar um endereço inválido?
- Como é que o controlador do disco rígido (por exemplo) avisa o sistema que ocorreu um erro?
- O que é uma função sistema?
- Qual a diferença entre função sistema e função biblioteca?
- Dê um exemplo de função biblioteca
- Dê um exemplo de função sistema
- O que é a fase de linkagem e para que ser (o que faz)?
- que era o "int 21h" do MSDOS? Era importante? Porquê?

## Fundamentos de sistemas operativos

### Parte 4 – Processos e *threads*

- A retirar desta parte
  - Revisão: Conceito de processo e de *thread*
  - Aspectos de implementação e estrutura interna de processos (em parte revisão)

## Entidades activas em execução

Como vê o sistema as entidades activas em execução

Os seguintes conceitos descrevem essas entidades

- Processo
- *Thread*
- Espaço de endereçamento
- Estado
- Identificador de processo
- Contexto hardware
- Contexto software
- Listas de processos (executáveis e bloqueados)

## Entidades activas em execução

### Programas x Processo (x *Thread*)

(Sistemas multi-programados, interactivos ou não)

- Um programa é uma sequência de instruções armazenadas de uma forma passiva sem qualquer concretização
  - Corresponde a uma ideia (um algoritmo)
- As entidades activas que partilham o processador são os **processos**
  - Apenas através de um processo é que um programa se pode executar
  - Correspondem ao concretizar das instruções do programador
- O escalonamento é efectuado em termos de processos
  - O mesmo programa pode estar a ser executado em processos distintos
- Alguns sistemas subdividem um processo em **threads**
  - Correspondem à ideia de ter “processos dentro de processos” – as *threads* irão ser vistas em detalhe em SO2

## Processos

### Programa x Processo

Programa  $\Leftrightarrow$  Entidade passiva = ficheiro em memória secundária

- Sequência de instruções sem actividade própria

Processo  $\Leftrightarrow$  Entidade activa em execução

- Entidade através da qual o conjunto de acções determinado por um programa é executado

Analogia: Roteiro turístico x Autocarro

- Processador  $\Leftrightarrow$  Condutor do autocarro  
Permite que o autocarro cumpra um roteiro
- Programa  $\Leftrightarrow$  Roteiro turístico  
Instruções (algoritmo) para uma viagem; a ser cumprido por um condutor através de um autocarro
- Processo  $\Leftrightarrow$  Autocarro em viagem com os turistas  
Conduzido por um condutor a executar as instruções de um roteiro

## Processos

Um processo constitui um ambiente de execução para um programa

Inclui:

- Um conjunto de operações
  - = Operações elementares proporcionadas (aparentemente) pelo hardware
    - Instruções máquina (reais ou virtuais - simuladas pelo S.O.)
    - Interacção com dispositivos físicos (reais ou virtuais - simulados pelo S.O.)
- Um espaço de endereçamento
  - = Memória endereçável (visível) pelo programa
    - Dá a aparência de não existir mais memória na máquina para além da que é visível (endereçável) pelo processo (protege-se assim os outros processos)
- Um contexto de execução
  - = Informação relativa ao processo
    - Contexto de hardware      ⇒ Informação necessária ao processador
    - Contexto de software      ⇒ Informação necessária ao S.O.

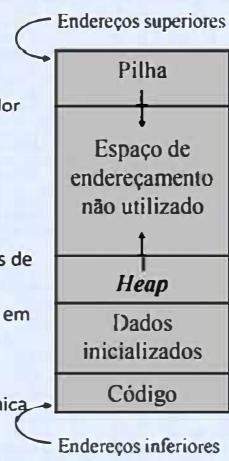
## Processos

### Espaço de endereçamento típico de um processo

(Assunto também já abordado no contexto do API Unix fork/exec)

#### Regiões

- **Código**
  - Contém as instruções a serem executadas pelo processador
- **Dados inicializados**
  - Dados cujo valor já está definido no próprio ficheiro executável (ex. variáveis estáticas ou globais)
- **Pilha**
  - Espaço onde são colocadas as variáveis locais e endereços de retorno das chamadas
  - Aumentar (até um certo limite) e diminui continuamente em run-time. A pilha cresce para baixo.
- **Heap**
  - Contém as zonas de memórias atribuídas de forma dinâmica (*malloc*, *new*, etc)
  - Pode ser aumentado (até um certo limite) em run-time.



## Processos

### Espaço de endereçamento típico de um processo

O tamanho das regiões é definido inicialmente em função do ficheiro correspondente ao programa

#### Código

#### Dados inicializados

Estas regiões têm o tamanho necessário para conter as instruções e os dados inicializados existentes no ficheiro executável.

#### Pilha

#### Heap

- O tamanho inicial destas duas regiões pode ser especificado na fase de ligação (a informação fica no ficheiro executável)
- O tamanho é dinâmico. As regiões crescer à medida do necessário.
- Crescem a partir de extremidades opostas e em sentido contrário de forma a colidirem o mais tarde possível (evitando ter de decidir arbitrariamente tamanhos máximos para cada uma)

## Processos

### Espaço de endereçamento típico de um processo

- O espaço de endereçamento dos processos é constituído por endereços virtuais
  - Do ponto de vista do código do programa desse processo, parecem endereços normais de memória
  - Os endereços virtuais são traduzidos (mapeados) de forma transparente pelo processador em endereços de memória real (ou memória física) ditos endereços reais
- Diferentes processos terão as suas regiões mapeadas em zonas distintas na memória
  - Assim, um processo não consegue afectar a memória de outro processo
  - Excepção: memória partilhada, que tem que ser pedida explicitamente por todos os intervenientes

#### Tipicamente:

- As regiões pertencentes ao processo encontram-se normalmente na metade superior do espaço de endereçamento
- Na metade inferior encontram-se as regiões correspondentes ao sistema operativo. De cada processo, as regiões do SO são mapeadas nas mesmas regiões em memória física (só existe uma cópia em memória do sistema)

## Processos

### **Contexto de software**

- Informação necessária ao sistema operativo para controlar a execução do processo
  - Identificação do processo e do utilizador
  - Estado do processo
  - Periféricos utilizados
  - Prioridade do processo
  - Ficheiros abertos
  - Programa em execução
  - Directória actual e por omissão
  - Contabilização da utilização dos recursos

Dependente da implementação do sistema

## Processos

### **Contexto de hardware**

- Estado do processador tal como ele se encontrava quando o processo foi retirado de execução
  - Acumulador
  - Registos de uso geral
  - Ponteiro para a próxima instrução (ex. IP)
  - Ponteiro de pilha (ex. SP)
  - Registo de estado do processador (*flags*)
  - Eventuais registos relativos à gestão de memória

Dependente da arquitectura da máquina

### **Processos – modo de execução**

Um processo pode executar-se em dois modos: utilizador e núcleo

- Modo utilizador
  - Corresponde às instruções definidas pelo programador
  - O sistema encontra-se protegido pois as ações do processo são limitadas
- Modo núcleo
  - A execução decorre dentro de funções sistema invocadas pelo processo ou de outro código invocado por estas
  - O sistema encontra-se protegido porque as instruções em execução não são determinadas pelo programador (o código é fiável)

Para cada processo existem duas pilhas

- Pilha utilizador
  - Contém variáveis locais e endereços de retorno de funções chamadas pelo processo
- Pilha núcleo
  - Contém endereços de retorno das funções internas ao núcleo em execução ao serviço do processo

### **Algumas questões de revisão**

- Qual a diferença entre processo e programa?
- Em que circunstâncias pode um processo estar associado a nenhum programa?
- Um programa pode estar a ser executado em vários processos?
- Um processo pode estar a executar vários programas?
- Descreva as zonas de memória típicas de um processo. Para que serve cada uma?
- Qual a diferença entre processo e *thread*?
- É possível um processo ter zero *threads*?
- Dê um exemplo concreto onde seja útil ter mais que uma *thread*.

# Sistemas Operativos

2020 – 2021

**Gestão de memória**

**Endereçamento virtual**

**Memória virtual**

## Tópicos

Conceitos fundamentais de gestão de memória;

Endereçamento real vs. endereçamento virtual;

Memória segmentada; memória paginada;

Memória virtual;

Algoritmos de atribuição, de transferência e de substituição;

Modelo de Espaço de Trabalho.

### Bibliografia específica:

- *Fundamentos de Sistemas Operativos*; 3<sup>a</sup> Ed.; Marques & Guedes
- Capítulos 4 e 5
- *Operating Systems Concepts* ; Silberschatz & Galvin  
Capítulos 8 e 9
- *Operating Systems - Internals and Design Principles*; William Stallings  
Capítulos 7 e 8

## Gestão de memória

### Mecanismos de gestão de memória

- Proporcionado processador/Hardware
- Determinam a organização da memória do computador
  - Qual o modo de endereçamento: real ou virtual
  - Qual o tamanho dos blocos (páginas ou segmentos)
  - Quais os mecanismos de protecção e validação de endereços

Cabe ao S.O. configurar e aproveitar adequadamente os mecanismos de gestão proporcionados pelo hardware

## Gestão de memória

### Algoritmos de gestão de memória

- Técnicas para a gestão de memória principal (física) e secundária (ex. implementação de políticas de atribuição de blocos)
  - Os detalhes da gestão de memória efectuada pelo S.O. devem ser ocultos aos diversos processos em execução
  - Os algoritmos de gestão de memória têm forte impacto no desempenho do sistema

É da responsabilidade do S.O. gerir de forma eficiente a memória, utilizando para isso os mecanismos proporcionados pelo hardware

## Gestão de memória

### **Endereçamento real (memória real)**

- Os endereços gerados e utilizados pelos programas em execução correspondem directamente a posições na memória física existente no computador
  - A dimensão dos programas fica limitada pela dimensão da memória física o computador
  - Normalmente um programa é preparado (pelo compilador) para correr numa determinada gama de endereços
  - O facto anterior dificulta a multi-programação pois quase de certeza que irão existir diversos programas a querer utilizar a mesma gama de endereços

## Gestão de memória – Sistemas de endereçamento real

### Ponto de vista de hardware:

→ Mais simples de implementar (circuitos mais simples)

### Ponto de vista do S.O.:

→ Apresenta dificuldades e limitações importantes



- Fragmentação decorrente da atribuição/libertação de memória
- Código não recolocável
- Limites quanto ao tamanho máximo dos programas
- Dificuldades na concretização de isolamento e segurança

Estes problemas têm solução parcial mas implicam uma perda de eficiência no desempenho da máquina, e complexidade acrescida para o SO e para os programadores das aplicações utilizador.

## Gestão de memória

### **Endereçamento virtual**

- Cada processo apercebe-se um conjunto de endereços (espaço de endereçamento) **sem relação directa** com qualquer porção de memória real
  - O processador traduz automaticamente esses endereços para endereços físicos.
  - A tradução é feita de forma transparente

#### **Endereços virtuais**

- São os endereços utilizados pelos processos em execução e que não correspondem directamente à memória física

#### **Endereços reais (endereços em memória física)**

- Referem posições da memória física (memória que existe de facto)

## Gestão de memória – Sistemas de endereçamento virtual

### **Endereçamento virtual**

O espaço de endereçamento de cada processo é subdividido num **conjunto de blocos**.

- A tradução end. virtual → end. real é feito bloco a bloco
- Todos os endereços virtuais do mesmo bloco têm as mesmas regras de tradução para endereços reais
- Dois blocos distintos podem ter regras distintas e tradução para endereçamento real
- Cada bloco pode estar mapeado em qualquer zona da memória física: não é preciso manter a mesma ordem que os blocos aparecem ter no espaço de endereçamento virtual
- Usando memória virtual, pode-se armazenar temporariamente em disco alguns dos blocos do espaço de endereçamento que não façam falta de momento (→ “**Memória virtual**”)

## Gestão de memória

### ➤ *memória virtual:*

- O espaço de endereçamento virtual é bastante maior que o total da memória física realmente existente.
- Algumas áreas do espaço de endereçamento encontram-se efectivamente mapeadas (algures) em memória física.
- As áreas dos espaços de endereçamento que não estão mapeadas em memória física encontram-se temporariamente armazenadas em memória secundária (disco).
- O S.O. gera a transferência das diversas áreas entre memória física e memória secundária (de uma forma transparente para os processos)

## Gestão de memória

### Endereçamento virtual e memória virtual

- É da responsabilidade do S.O. gerir os vários espaços de endereçamento virtuais:
  - Trazer para memória física os blocos que estão a ser necessários
  - Libertar espaço em memória física, transferindo para memória secundária blocos que não estejam a ser usados
  - Indicar ao processador de que forma é efectuada a tradução de endereços virtuais para endereços reais
- Esta gestão é efectuada de uma forma transparente para os processos em execução. Cada processo vê a memória como estando toda disponível para si.

## Gestão de memória – Sistemas de endereçamento virtual

### Memória segmentada

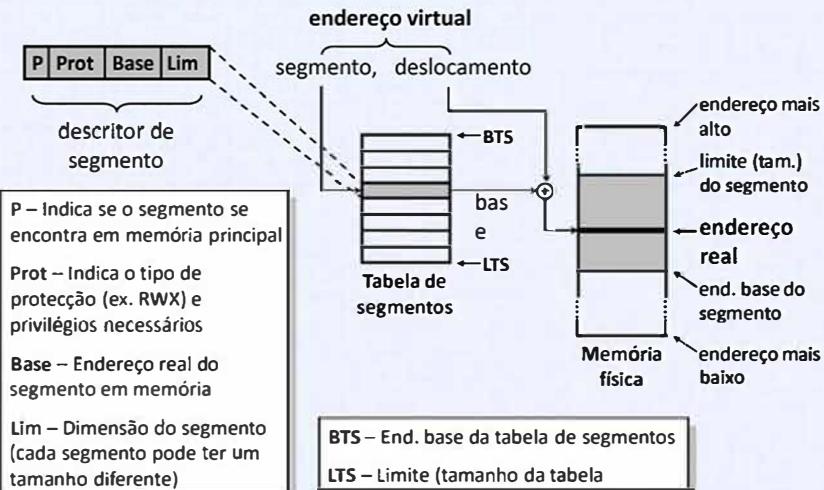
- Bloco = “segmento”
- Segmentos diferentes podem ter tamanhos diferentes
- A tradução de endereços virtuais em endereços reais é feita pelo próprio processador com recurso a uma ou mais tabelas, chamadas **tabelas de segmentos**, as quais são criadas e preenchidas pelo S.O.
- Cada entrada numa tabela de segmentos chama-se **descritor de segmento**, e descreve o segmento quanto a:
  - Tamanho e localização em memória física
  - Tipo de acesso permitido (proteção: leitura, escrita, execução)
  - Estado de presença em memória física (bit “P”)
- Registos específicos no processador indicam a base da tabela de segmentos e a dimensão da tabela de segmentos em uso.

## Gestão de memória – Sistemas de endereçamento virtual

### Memória segmentada

- Um endereço virtual é composto por um conjunto de bits que indicam qual o descritor de segmento dentro da tabela de segmentos que vai ser utilizado (abrev. “qual o segmento”) e um conjunto de bits que indica qual o deslocamento dentro desse segmento
- Para optimizar o custo do acesso à tabela de segmentos durante a tradução de endereços, utiliza-se um conjunto de registos de segmento que contêm os valores dos descritores de segmento em uso. Se o programa não estiver sempre a mudar de segmento, evita-se grande parte dos acessos à tabela de segmentos.

## **Memória segmentada – Tradução dos endereços**



DE|S/SEC

Sistemas Operativos – 2020/21

1

Joséo Durães

Gestão de memória – Sistemas de endereçamento virtual

## Memória paginada

Análoga à memória segmentada com a diferença que todos os blocos, chamados de páginas, têm o mesmo tamanho (geralmente pequenas)

- A tradução de endereços é feita de maneira análoga à da memória segmentada: em vez de tabelas de segmentos utilizam-se tabelas de páginas; cada entrada nessas tabelas denomina-se de PTE (*Page Table Entry*). Cada PTE descreve uma página quanto a:
    - Localização em memória física (por outras palavras, qual a *page-frame* em memória física onde está mapeada)
    - Tipo de acesso permitido (protecção)
    - Estado de presença em memória física (bit "P")
    - Utilização da página quanto a referência (bit "R") e modificação (bit "M")
  - Os registos BTP e LTP são os correspondentes aos BTS e LTS da memória segmentada

DEIS/SEC

Sistemas Operativos – 2020/21

1

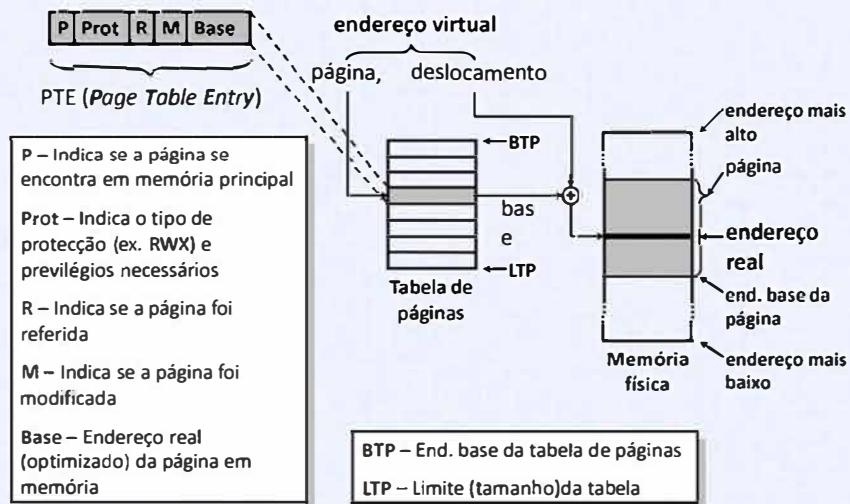
João Durães

### Gestão de memória – Sistemas de endereçamento virtual

#### Memória paginada (cont.)

- O facto das páginas terem todas o mesmo tamanho tem como consequências:
    - É possível a criação de um espaço de endereçamento perfeitamente linear
    - É possível optimizar ou simplificar os algoritmos de gestão (a ver mais adiante)
  - Para optimizar o custo do acesso à tabela de páginas durante a tradução de endereços, utiliza-se uma *cache* que consiste numa tabela de memória associativa muito rápida para servir de *cache* às PTE recentemente utilizadas (que serão provavelmente, pelo princípio da localidade de referência, as próximas a serem necessárias).
- Esta tabela denomina-se de TLB (*Translation Lookaside Buffer*)

### Memória paginada – Tradução dos endereços



### Gestão de memória – Memória paginada multi-nível

- Se o espaço de endereçamento for muito grande a memória paginada torna-se ineficiente

#### Exemplo

- Endereços de 64 bits
- Tamanho de página: 4k (dimensão típica): 12 bits
- Restante:  $64 - 12 = 52$  bits

→ Tabela de páginas teria (até)  $2^{52}$  entradas !

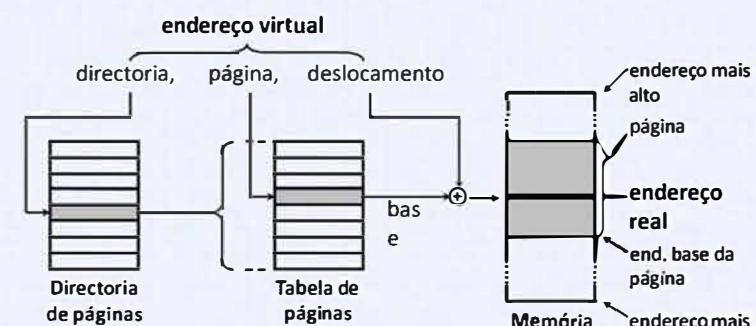


Não é viável manter a tabela em memória

#### Solução:

- Utilizar vários níveis de tabelas de páginas (em cascata)

### Memória paginada multi-nível



Cada entrada na directória de páginas indica uma tabela de páginas. A partir deste ponto a tradução é como na situação anterior.  
Pode existir mais que um nível intermediário  
Pode-se associar uma entrada na directória de páginas a cada processo

Exemplo - Processadores Intel:  
Memória paginada de nível duplo  
Directória – 10 bits  
Página – 10 bits  
Deslocamento – 12 bits

## Gestão de memória – Sistemas de endereçamento virtual

### Memória segmentada

#### Vantagens

- Adequa-se bastante à divisão lógica dos programas: um segmento para código, outro para dados, etc. Não é obrigatório que só haja um segmento de cada tipo
- Permite trabalhar eficientemente sobre uma zona de memória inteira (um segmento)

#### Desvantagens

- Pode não permitir criar um espaço de endereçamento perfeitamente linear e contínuo de uma forma transparente para o processo
- Os algoritmos de gestão são mais complicados; o mapeamento de segmentos em memória pode gerar fragmentação (porque cada segmento pode ter um tamanho diferente dos outros)
- As transferências entre memória principal e secundária é sempre feita em blocos (segmentos) inteiros. No caso dos segmentos serem grandes, têm-se grandes custos de eficiência

## Gestão de memória – Sistemas de endereçamento virtual

### Memória paginada

#### Vantagens

- Totalmente transparente para o programador e processo
- Os algoritmos de gestão são mais simples e mais eficientes pelo facto de todas as páginas terem o mesmo tamanho; as transferências entre memória principal e memória secundária são feitas por páginas individuais (tamanho típico: 512 bytes a 8K), sendo muito rápidas
- Só há fragmentação (interna) na última página do processo

#### Desvantagens

- Não se adequa à divisão lógica dos programas tão bem como a memória segmentada pois cada parte lógica dos programas (código, dados, etc.) irão precisar de mais do que uma página
- As faltas de páginas representam uma perda de eficiência maior do que as faltas de segmento (por serem potencialmente em maior número)

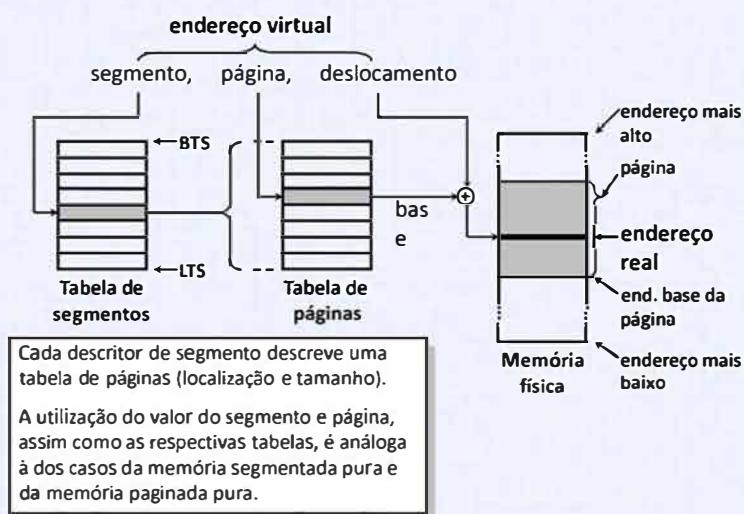
## Gestão de memória – Sistemas de endereçamento virtual

### Memória segmentada-paginada

Mecanismo com semelhanças à memória paginada multi-nível e junta as melhores características da memória segmentada e da memória paginada

- Um endereço virtual é composto por um segmento, uma página e o deslocamento dentro da página; o segmento indica qual o descritor dentro da tabela de segmentos que deve ser utilizado na tradução do endereço; o descritor indicado contém informação acerca de qual a tabela de páginas a ser utilizada para o restante da tradução do endereço (esse restante é feito como no caso da memória paginada)
- Consegue-se ter, desta forma, um segmento subdividido em N páginas, mantendo-se as vantagens da adequação dos segmentos à estrutura lógica dos programas e a eficiência associada às páginas
- Continuam a ser utilizados ambos os mecanismos de optimização de eficiência: registos de segmentos e TLB

## Memória segmentada-paginada – Tradução dos endereços



### Gestão de memória – Sistemas de endereçamento virtual

Vantagens da utilização de memória virtual (através de memória segmentada, paginada ou suas variantes)

- Facilidade na implementação de sistemas multi-programados: cada processo tem o seu próprio espaço de endereçamento, sendo evitada a necessidade de código recolocável; os algoritmos de gestão de memória física podem recolocar os blocos de memória em localizações diferentes sem que isso afecte os processos em execução
- Implementação automática de segurança e isolamento dos espaços de endereçamento de processos diferentes: cada processo terá a(s) sua(s) própria(s) tabela(s) de segmentos ou páginas
- A partilha de zonas de memória entre processos distintos é simples: dois (ou mais) descritores de segmento (ou PTE's) descreverão o mesmo bloco em memória física (eventualmente com protecções diferentes, se isso for desejável)

### Gestão de memória – Políticas de transferência

#### **Matéria seguinte:**

- Algoritmos de atribuição
- Algoritmos de transferência
- Algoritmos de substituição

## Gestão de memória

### Algoritmos de gestão de memória

- **Algoritmos de atribuição (alocação)**
  - Determinam onde (em memória física) fica mapeado um bloco de memória virtual
- **Algoritmos de transferência**
  - Determinam a política quanto às transferências entre memória principal e memória secundária
- **Algoritmos de substituição**
  - Determinam qual o bloco que deve ser retirado de memória física para dar lugar a outro

## Gestão de memória - Algoritmos

### Algoritmos de atribuição

#### Em memória paginada

- Uma vez que todas as páginas têm o mesmo tamanho, a decisão de qual a *page-frame* (página de memória física) que fica atribuída a uma página de memória virtual resume-se a encontrar uma que esteja livre.
- Para tal apenas é preciso manter uma estrutura de dados que indique quais as páginas ocupadas e quais as páginas livres.
- A primeira página livre encontrada é a escolhida.

## Gestão de memória – Algoritmos de alocação

### Algoritmos de atribuição

#### Em memória segmentada

- Cada segmento pode ter um tamanho diferente.
- Pode surgir o problema de fragmentação se não se escolher bem qual o bloco a atribuir.

⇒ A decisão deixa de ser trivial como no caso da memória paginada

Existem vários algoritmos

- **Best-Fit**
- **Worst-Fit**
- **First-Fit**
- **Next-Fit**
- **Buddy (binário)**

## Gestão de memória – Algoritmos de alocação

### Algoritmo **Best-Fit**

- Procura o bloco que melhor se adapte à dimensão pedida (que será o menor bloco que ainda consiga conter aquilo que foi pedido)
- Este algoritmo visa optimizar a ocupação da memória. No entanto, geralmente o bloco atribuído excede em pouco a dimensão pedida. O excesso vai formar um novo bloco, geralmente demasiado pequeno para servir para alguma coisa.
- A estrutura de dados associada consiste numa lista de blocos, ordenada por ordem crescente. Em média será preciso percorrer metade da lista até encontrar o bloco desejado. O bloco resultante do excedente tem que ser inserido na posição adequada na lista.

## Gestão de memória – Algoritmos de alocação

### Algoritmo *Worst-Fit*

- Procura o maior bloco, sendo esse o bloco atribuído.
- Este algoritmo tenta prevenir o aparecimento de blocos pequenos como no caso do *best-fit*. A ideia é a seguinte: o excedente do bloco (a diferença entre o que foi atribuído e o que foi pedido) ainda será suficientemente grande para ser útil a pedidos posteriores. No entanto os blocos grandes desaparecem rapidamente tendo como consequência a possibilidade de pedidos grandes posteriores não poderem ser satisfeitos.
- A estrutura de dados associada será uma lista de blocos ordenada por ordem decrescente. A determinação de qual é o maior bloco fica bastante facilitada. O bloco resultante do excedente tem que ser inserido na posição adequada na lista.

## Gestão de memória – Algoritmos de alocação

### Algoritmo *First-Fit*

- Escolhe para atribuição o primeiro bloco que é suficientemente grande para satisfazer o pedido.
- Este algoritmo tem como objectivo diminuir o tempo de pesquisa do bloco a atribuir sem ter a desvantagem do *worst-fit*.
- É gerada fragmentação num dos extremos de memória enquanto se salvaguarda a existência de blocos grandes livres no outro extremo.
- A estrutura de dados associada consiste numa lista de blocos ordenada pelo endereço do bloco. Os blocos resultantes do excedente entre o bloco atribuído e o bloco pedido são mais facilmente reintroduzidos que nos algoritmos anteriores.

## Gestão de memória – Algoritmos de alocação

### Algoritmo **Next-Fit**

- Versão modificada do *first-fit*, em que cada pesquisa começa no ponto onde a anterior terminou.
- Esta modificação visa evitar o aparecimento de fragmentação apenas num dos extremos da lista, melhorando a velocidade de pesquisa, tendo no entanto como consequência a dispersão de blocos pequenos por toda a memória.

## Gestão de memória – Algoritmos de alocação

### Algoritmo **Buddy-Binário**

- Este algoritmo atribui blocos de tamanho  $2^n$  de tal forma que a dimensão pedida  $TAM$  verifica  $2^{n-1} < TAM \leq 2^n$ . Se não existir nenhum bloco livre nessa condição, tenta-se encontrar um bloco maior (sempre com uma dimensão potência de 2), o qual será dividido em dois de uma forma recursiva até se obter um bloco que verifique a condição. Os outros blocos resultantes deste processo de divisão ficam livres para outros pedidos.
- Este algoritmo tem a vantagem de conseguir um bom equilíbrio entre a velocidade de pesquisa e a fragmentação: A quantidade de blocos manipulados nunca é demasiado grande; por outro lado os blocos libertos vão sendo recombinados, reconstruindo-se facilmente os blocos maiores originais.

## Gestão de memória – Políticas de transferência

### Políticas de transferência

- Três situações

- **Por pedido**

O S.O. (ou um processo) determina quando se deve carregar um bloco em memória (ex.: na criação de um processo)

- **Por necessidade**

O bloco de memória é acedido numa altura em que não se encontrava em memória principal; dá-se uma falta de página ou segmento e torna-se necessário carregar o bloco em memória física

- **Por antecipação**

Para tentar aumentar o desempenho, o bloco é carregado em memória quando há fortes probabilidades de ele vir a ser necessário nos próximos instantes (mas antes de ser efectivamente necessário)

## Gestão de memória – Políticas de transferência

### Transferência de páginas

- As transferências de páginas são geralmente feitas por necessidade:
  - Quando um processo é criado, as suas tabelas de páginas são inicializadas mas as páginas não são colocadas em memória;
  - À medida que o processo vai progredindo, irá gerar endereços que irão causar faltas de páginas - nessas ocasiões, o S.O. colocará as páginas em memória de uma forma transparente para o processo (porque estão a ser necessárias);
  - Também é habitual usar a política de antecipação para optimização dos acessos a memória secundária
  - As páginas que nunca chegam a ser necessárias (código que não chegou a ser executado ou dados que não chegaram a ser necessários, por exemplo) nunca serão carregadas, tendo-se pouparido tempo.

## Gestão de memória – Políticas de transferência

### Transferência de segmentos

Normalmente, um processo precisa de vários segmentos em simultâneo para se poder executar: código, dados, pilha, *heap*, etc. Em processos pequenos pode coincidir com a totalidade da sua memória. Quando o sistema coloca o processo em execução, coloca em memória os seus segmentos.

- As transferências de segmentos são, geralmente, feitas por pedido
- Se o processador suportar excepções do tipo falta de segmento, então também se podem efectuar transferências de segmento por necessidade. No entanto, mesmo que o processador suporte este tipo de excepções, raramente os SOs as utilizam pois uma falta de segmento tem um *overhead* muito maior que uma falta de página

## Gestão de memória – Algoritmos de Substituição

### Substituição de páginas

- O factor decisivo quanto à escolha das páginas que são retiradas de memória para dar lugar a outras está relacionado com a sua utilização.
- Idealmente, as páginas a retirar serão aquelas que apenas serão necessárias em último lugar. Este algoritmo não é directamente concretizável pois depende de conhecimento acerca do futuro.
- Através do princípio de localidade de referência, pode-se utilizar o conhecimento da utilização passada das páginas para inferir qual irá ser a sua utilização futura.
- Os seguintes algoritmos consistem na tentativa de aproximação ao algoritmo ideal:
  - FIFO                  *First In First Out*
  - LRU                  *Least Recently Used*
  - NRU                  *Not Recently Used*

### Gestão de memória – Substituição de páginas – FIFO

#### Algoritmo FIFO – *First In First Out*

- Este algoritmo mantém as páginas numa lista ordenada pelo tempo em que foram carregadas em memória.
- Quando se carrega uma página em memória, coloca-se no fim da lista.
- Quando é necessário remover uma página, escolhe-se a que está no início da lista.

Este algoritmo tem a desvantagem de não conseguir distinguir uma página que já não é necessária (foi carregada à muito tempo) de uma que está a ser sempre necessária (de tal forma que foi das primeiras a serem carregadas).

⇒ Pode acontecer retirar-se uma página que vai voltar a ser necessária daí a pouco tempo.

### Gestão de memória – Substituição de páginas – LRU

#### Algoritmo LRU – *Least Recently Used* (menos usada recentemente)

- Este algoritmo baseia-se no tempo em que a página foi acedida em vez de o tempo em que a página foi carregada. Desta forma consegue-se distinguir páginas que estão sempre a ser acedidas e que foram carregadas há muito tempo das que foram carregadas há muito tempo mas que já não são utilizadas.
- Idealmente, o próprio processador manteria a idade da página desde o último acesso nas PTE. No entanto, tal solução seria demasiado custosa em termos de complexidade de *hardware*, pelo que não é vulgar.  
⇒ A contabilização do tempo que passou desde o ultimo acesso a cada página será feita pelo S.O.

### Gestão de memória – Substituição de páginas – LRU

#### Algoritmo LRU (cont.)

- A cada página é associado um contador, inicializado a zero quando a página é carregada.
- Sempre que uma página é acedida, o bit “R” da PTE através do qual foi acedida é colocado a “1”. Normalmente, esta actualização é feita pelo próprio processador (caso o processador não suporte esta característica, ainda é possível ao S.O. conseguir efectuar esta actualização mas com grande penalização em termos de eficiência).
- Periodicamente todas as tabelas de páginas são inspecionadas; se o bit “R” estiver a “0”, o contador associado será incrementado (a página está a ficar “velha” por não ser utilizada); se o bit “R” estiver a “1”, o contador e esse bit são recolocados a zero (a página voltou a ser nova porque está a ser utilizada).
- Quando o contador atinge o valor máximo, a página associada é marcada como inválida (o bit “P” é posto a zero) para todos os efeitos, o espaço ocupado por ela em memória física está livre.

### Gestão de memória – Substituição de páginas – NRU

#### Algoritmo NRU – *Not Recently Used* (não usada recentemente)

- Simplificação do algoritmo LRU e que consiste analisar os bits “R” e “M” das PTE para classificar as páginas quanto à sua utilização.
- Interessa averiguar apenas se uma página foi ou não acedida ou modificada recentemente (não interessando distinguir o tempo decorrido); as páginas serão classificadas consoante as 4 combinações possíveis de valores para estes dois bits.
  - ➔ Periodicamente as tabelas de páginas são percorridas:
    - O bit “R” é posto a zero pelo processo paginador
    - O bit “M” é posto a zero quando a página é gravada em disco

#### Convém lembrar que:

- O bit “R” é colocado a “1” sempre que a página é acedida;
- O bit “M” é colocado a “1” sempre que a página é modificada;
- Normalmente, estas duas actualizações são feitas automaticamente pelo próprio processador quando traduz os endereços virtuais.

### Gestão de memória – Substituição de páginas – NRU

#### Algoritmo NRU (cont.)

- As páginas são escolhidas como vítimas para sairem da memória pela seguinte ordem

ordem	R	M	
1º	0	0	Nem referida nem modificada
2º	0	1	Não referida mas modificada
3º	1	0	Referida mas não modificada
4º	1	1	Referida e modificada (pior caso)

**Referida** ⇒ está a ser necessária (utilizada recentemente) – provavelmente será necessária brevemente (pelo princípio da localidade de referência)

**Modificada** ⇒ Tem que se gravar em memória secundária para não se perderem as modificações – tem o custo adicional de I/O

### Gestão de memória – Algoritmos de Substituição

#### Substituição de segmentos

Como cada segmento tem o seu próprio tamanho, a escolha do(s) segmento(s) a rer(em) retirado(s) de memória já não pode ser feito da mesma forma que em memória paginada

- Normalmente retiram-se segmentos em função do processo a que pertencem
- Interessa ver qual o processo que não precisa (não se prevê que vá) executar nos próximos instantes; tal processo é uma boa “vítima” para ter os seus segmentos retirados de memória principal

Factores a levar em conta em relação ao processo

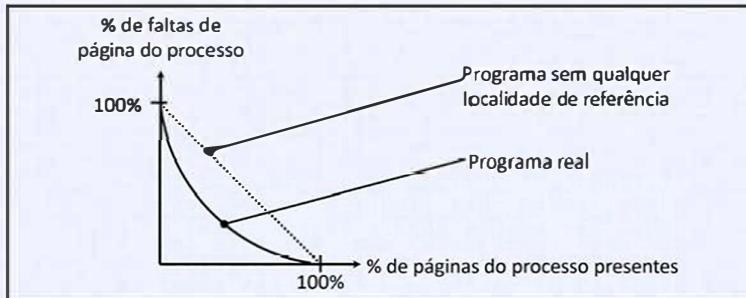
(cujos segmentos vão ser retirados)

- Qual a sua importância / prioridade
- Qual a sua utilização do processador
- Qual a sua dimensão (total dos seus segmentos)

### Gestão de memória – Substituição de páginas

#### Modelo de espaço de trabalho

- Observa-se que compensa nunca colocar em memória mais do que uma determinada percentagem das suas páginas:
- Não aumenta muito o número de faltas de página
  - Ficam mais páginas livres para outros processos



Variação das faltas de páginas em função da percentagem de páginas presentes

### Gestão de memória – Substituição de páginas

#### Modelo de espaço de trabalho (cont.)

“Espaço de trabalho” : quantidade de páginas que um processo deve ter presentes em memória

- A dimensão do espaço de trabalho não é necessariamente a mesma para todos os processos.
- Quando um processo é criado, são carregadas algumas páginas (por antecipação) até se atingir um espaço de trabalho mínimo.
- À medida que o processo vai pedindo mais páginas, sem exceder o seu espaço de trabalho máximo, as novas páginas são obtidas do espaço atribuído a outros processos.
- Quando o processo atinge o valor máximo do seu espaço de trabalho, as novas páginas que lhe são atribuídas são obtidas do seu próprio espaço.
- Quando a quantidade de páginas de um processo presentes em memória desce abaixo do seu valor mínimo de espaço de trabalho (por lhe terem sido tiradas por outros processos), todo o processo é enviado para memória secundária.

### Aferição de conhecimentos neste capítulo

Perguntas que se podem fazer sobre esta matéria (exemplos não exaustivos\*)

Compreensão/descrição directa

- Quais as desvantagens do endereçamento real face ao endereçamento virtual
- Descreva o mecanismo de tradução de endereço de memória *<um dos que foi dado>*
- Elabore uma análise comparativa de memória segmentada e memória paginada salientando as vantagens e desvantagens de um face ao outro.
- Descreva o conceito de espaço de trabalho e diga como pode ser usado para aumentar o desempenho do computador

Compreensão com um pouco de raciocínio

- Porque razão o mecanismo de exceções é fundamental para a construção de memória virtual?
- Como é que se constrói o isolamento entre processos recorrendo à memória segmentada? E em memória paginada, há alguma diferença significativa?
- Quais as vantagens da memória segmentada-paginada face à memória segmentada pura e memória paginada pura?

(\*) Convém sobrar uma ou duas para os exames (mas isto não quer dizer que saiam estas)

### Aferição de conhecimentos neste capítulo

Perguntas que se podem fazer sobre esta matéria (cont.)

Aplicação de conhecimentos – exemplo de perguntas (dados concretos em aberto)

- Dado este conjunto de páginas e os tempos em que foram acedidos e modificados, diga qual será a primeira a sair de acordo com o algoritmo FIFO/LRU/NRU e porquê
- Dada esta tabela de páginas e estas instruções a serem executadas no espaço de endereçamento que lhe corresponde, diga quais irão ser os endereços reais manipulados e se existirá algum comportamento específico por parte do SO (se sim, qual e porquê)
- Dado este processo (código) diga qual acha que será a dimensão (%) do seu espaço de trabalho.
- Dado este cenário de ocupação de memória real em memória segmentada, diga onde irá ser encaixado um novo segmento de tamanho segundo o algoritmo *<inserir algoritmo aqui>*

## Mecanismo *select* – Resumo, API e exemplo

Este documento resume o mecanismo *select* abordado nas aulas e apresenta um exemplo de utilização.

### Conteúdo

1. Mecanismo *select* (explicação e cenário de utilização)
2. Forma de uso
3. API e Estruturas de dados envolvidas
4. Exemplo com código

### 1. Mecanismo *select*

O mecanismo *select* permite aguardar por mais do que uma operação de entrada/saída sem ser necessário saber qual delas se completa em primeiro lugar. Este mecanismo possibilita, por exemplo, aguardar informação em mais do que um *pipe*, ou num *pipe* e num teclado, sem usar sinais para sinalização nem *threads*.

#### Cenários de utilização.

O mecanismo é útil quando um programa está a usar operações bloqueantes e deseja efectuar mais do que uma. Como as operações são bloqueantes, corre-se o risco de aguardar por aquela que ocorre em último lugar, perdendo-se tempo e eventualmente deixando de atender um acontecimento (ex.: chegada de informação) por se estar bloqueado à espera da “outra” operação.

### **Exemplo de aplicação:**

Considere-se um programa que necessita de dar atenção simultaneamente a um *pipe* (leitura) e ao teclado. Usam-se funções bloqueantes e é um requisito responder imediatamente à chegada de informação. Só existem as seguintes soluções em Unix:

- **Recurso a sinais:** o programa associa o atendimento de um sinal à rotina de leitura do *pipe* e fica a aguardar na leitura do teclado. O outro programa responsável por enviar informação a este pelo *pipe* terá também que lhe enviar um sinal para o “avisar” e causar a execução da sua rotina de leitura do *pipe*.
  - Pouco vantajosa: envolve mais um mecanismo (sinais). É pouco versátil e só funciona para situações onde há um programa do outro lado capaz de enviar sinais (no caso do teclado, não é útil).
- **Utilização de *threads*:** o programa lança uma *thread* cujo código só lê o *pipe*, e outra *thread* só lê o teclado.
  - É simples, aplicável a muitas situações, e costuma ser a melhor solução. Neste momento ainda não se falou de *threads* e irá ser dado um exemplo mais tarde com este cenário. Por esta razão não se vai considerar esta hipótese neste documento.
- **Utilização de operações não bloqueantes:** os *read/write* deixam de bloquear e o programa nunca fica bloqueado “à espera na outra fonte de dados”.
  - Resolve o problema, mas levanta outros piores. As operações assíncronas exigem ao programador algoritmos mais complicados, e exigem ao programa o teste contínuo acerca da disponibilidade de dados/conclusão da operação (*polling*), ocupando o processador e prejudicando a performance geral da máquina.
- **Mecanismo *select*:** o programador indica quais as fontes de dados/mecanismo de comunicação que deseja usar e o sistema aguarda em todas, detectando qual a primeira em que a operação pretendida está pronta. O programa apenas espera numa coisa.
  - É relativamente simples de usar e apenas a solução com *threads* será melhor (por ser mais genérica).
  - Esta é a solução tratada e exemplificada neste documento.

## 2. **select** - Forma de uso:

O programa encontra-se eventualmente (mas não obrigatoriamente) num ciclo onde desencadeia operações E/S. Podem ser de leitura, de escrita, ou de atendimento de acontecimentos de excepção. Dentro do ciclo (se houver) vai fazer:

- Prepara o conjunto **fd\_set** que vai conter os descritores de ficheiros correspondentes aos dispositivos/mecanismos de comunicação onde deseja efectuar operações.
  - Coloca o conteúdo a zero – **FD\_ZERO**.
  - Acrescenta os descritores – **FD\_SET**.
    - Indica qual o descritor numericamente mais elevado +1.
  - Irá existir um conjunto para descritores para operações de leitura, outro para os descritores em operações de escrita, e outro para as notificações de excepção.
- Coloca o *timeout* (prazo máximo a aguardar no *select*) numa estrutura **timeval**.
- O programa invoca a função **select** indicando o ponteiro para cada um dos descritores e a estrutura **timeval** com o *timeout*.
- A função *select* bloqueia e devolve apenas quando uma das operações pretendidas está em condições de ser concluída de imediato.
- Após a função *select* retornar, o programa deve averiguar com **FD\_ISSET** cada um dos descritores para saber qual deles é que está pronto a usar.
- A função *select* modifica a estrutura **fd\_set** e **timeval**. Caso se esteja em ciclo, é necessário colocar a informação de início nestas estruturas.

## 3. API e Estruturas de dados envolvidas

- Set de flags (prontidão) de descritores de ficheiros

**fd\_set**

- Aguardar dados disponíveis/possibilidade de escrita através dos descritores indicados

```
int select(int nfds,  
          fd_set *readfds, fd_set *writefds, fd_set *exceptfds,  
          struct timeval *timeout);
```

- Remover descriptor ao conjunto de flags a observar

```
void FD_CLR(int fd, fd_set *set);
```

- Verificar disponibilidade de dados/ possibilidade de escrita num descriptor

```
int FD_ISSET(int fd, fd_set *set);
```

- Adicionar descriptor ao conjunto de flags a observar

```
void FD_SET(int fd, fd_set *set);
```

- Limpar conjunto de flags de descritores

```
void FD_ZERO(fd_set *set);
```

#### 4. Código exemplo:

O exemplo consiste num programa que consegue ler de dois pipes e do teclado em simultâneo.

#### Função principal

```
int main(int argc, char* argv[]) {
    int fd_a, fd_b, fd_c;      //file descriptor dos pipes
    int nfd;                  //valor retorno select()
    fd_set read_fds;          //conjunto das flags para desc. ficheiros
    struct timeval tv;        //timeout para select

    signal(SIGINT, trataCC); //para interromper via ^C

    //cria os pipes
    mkfifo("pipe_a", 00777);
    mkfifo("pipe_b", 00777);

    // abre os pipes. RDRW vs RD – notar isto
    fd_a = open("pipe_a", O_RDWR | O_NONBLOCK);
```

```

if (fd_a == -1)
    sayThisAndExit("Erro no open pipe_a");
fd_b = open("pipe_b", O_RDWR | O_NONBLOCK);
if (fd_b == -1)
    sayThisAndExit("Erro no open pipe_b");

while (1) {

    tv.tv_sec = 10;      // segundos      (10 = apenas um exemplo)
    tv.tv_usec = 0;      // micro-segundos (se ambos a 0 então faz polling)

    FD_ZERO(& read_fds);          // inicializa conjunto de fd (watch list)
    FD_SET(0, & read_fds);        // adiciona stdin ao conj de fd a observar
    FD_SET(fd_a, & read_fds);    // adiciona pipe_a ao conj de fd a observ.
    FD_SET(fd_b, & read_fds);    // adiciona pipe_b ao conj de fd a observ.

    //vê se há dados em alguns dos fd (stdin, pipes) - modifica os sets
    //bloqueia ate: sinal, timeout, há dados para ler EOF, exception
    nfd = select(           // bloqueia até haver dados ou EOF no read-set
        max(fd_a,fd_b)+1,   // max valor dos vários fd + 1
        & read_fds,          // read fd set
        NULL,                // write fd set - (nenhum aqui)
        NULL,                // exception fd set - (nenhum aqui)
        & tv);   // timeout - se ambos = 0-> retorna logo (p/ polling)
    //actualiza tv -> quanto tempo faltava para o timeout

    if (nfd == 0) {
        printf("\n(Estou a espera....)\n"); fflush(stdout);
        continue;
    }

    if (nfd == -1) {
        perror("\nerro no select");
        close(fd_a); close(fd_b);
        unlink("pipe_a"); unlink("pipe_b");
        return EXIT_FAILURE;
    }

    if (FD_ISSET(0, & read_fds)) {      // stdin tem algo para ler?
        trataTeclado();
        // sem "continue" -> não vai logo para a próxima iteração
    } // porque pode ser que pipe_a ou pipe_b também tenha algo
}

```

```

if (FD_ISSET(fd_a, & read_fds)) {    //fd_a tem algo para ler?
    leEMostraPipes("A", fd_a);        //função auxiliar para ler pipes
    // sem "continue" -> não vai logo para a próxima iteração
} // porque pode ser que pipe_b também tenha algo

if (FD_ISSET(fd_b, & read_fds)) {    //fd_b tem algo para ler?
    leEMostraPipes("B", fd_b);
    // o código dentro do ciclo acaba já a seguir e volta a esperar por dados
}
}

// em princípio, neste exemplo, não deve chegar aqui
return EXIT_SUCCESS;
}

```

#### Funções seguintes:

São de natureza auxiliar. O código de tratamento de leitura de caracteres é mais complexo que o necessário e deixa-se a sua simplificação para trabalho em casa

#### Atendimento da chegada de informação via pipe(s)

```

void leEMostraPipes(char * quem, int fd) {
    char buffer[200];
    int bytes;
    bytes = read(fd, buffer, sizeof(buffer));
    buffer[bytes] = '\0';
    if ( (bytes > 0) && (buffer[strlen(buffer)-1] == '\n') )
        buffer[strlen(buffer)-1] = '\0';
    printf("\n%s: (%d bytes) [%s]\n", quem, bytes, buffer);
    if (strcmp(buffer, "sair") == 0) {
        unlink("pipe_a");
        unlink("pipe_b");
        exit(EXIT_SUCCESS);
    }
}

```

## Atendimento da chegada de informação via stdin

```
void trataTeclado() {
    char buffer[200];
    int bytes;
    fgets(buffer, sizeof(buffer), stdin); // scanf("%s",buffer);
    if ( (strlen(buffer)>0) && (buffer[strlen(buffer)-1]=='\n') )
        buffer[strlen(buffer)-1] = '\0';
    printf("\nKBD: [%s]\n", buffer);
    fflush(stdout);
    if (strcmp(buffer, "sair")==0) {
        unlink("pipe_a"); unlink("pipe_b");
        exit(EXIT_SUCCESS);
    }
}
```

## Funções secundárias: max, trataCC, sayThisAndExit

### Terminação em caso de erro

```
void sayThisAndExit(char * p) {
    perror(p);
    exit(EXIT_FAILURE);
}
```

### Obtenção do máximo de dois inteiros

```
int max(int a, int b) {
    return (a>b) ? a: b;
}
```

## Atendimento de SIGINT (recepção de ^C)

```
void trataCC(int s) {
    unlink("pipe_a"); unlink("pipe_b");
    printf("\n ->CC<- \n\n");
    exit(EXIT_SUCCESS);
}
```

## Ficheiros *header* usados

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#include <signal.h>
#include <string.h>
```

## **Named pipes - Resumo e Exemplo Cliente/Servidor**

### **1. Enquadramento**

Este exemplo mostra como se pode construir um sistema de informação cliente-servidor utilizando *named pipes* (também referidos como FIFO) para transportar informação entre os clientes e o servidor. Neste exemplo, tal como acontece genericamente quando se fala em “cliente-servidor”, tanto o servidor como o cliente enviam e recebem informação (ou seja, a informação flui nos dois sentidos).

#### **1.1 Named pipes em Unix**

Os *named pipes* em unix têm as seguintes características genéricas:

- Correspondem por alto a um tubo: escreve-se informação numa extremidade e retira-se essa informação pela outra extremidade. A informação é retirada pela mesma ordem com que foi enviada, daí o nome alternativo para este mecanismo de comunicação: FIFO – “First In First Out”.
- O uso de um *named pipe* é semelhante ao de um ficheiro. Usam-se as mesmas funções para abrir, fechar, ler e escrever. A informação a escrever/ler é semelhante à noção de “ficheiro binário” no sentido em que o que é escrito é simplesmente *uma determinada quantidade de bytes* cujo significado é da responsabilidade dos processos que usam o *named pipe* e não do sistema operativo (nota: recordar que em unix não existe distinção entre ficheiros binários/texto).
- A estrutura interna da mensagem, e o tamanho é totalmente da responsabilidade dos processos, que devem conhecer à partida a estrutura e quantos bytes é suposto ler/escrever.
- A comunicação através de *named pipes* é, normalmente, unidireccional. Para se conseguir ter uma comunicação bidireccional entre dois processos serão usados, normalmente, dois *named pipes*, uma para cada direcção.
- Normalmente, num determinado *named pipe* só existirá um processo a ler. Podem haver um ou vários processos a escrever, dependendo do modelo de comunicação a ser implementado.

#### **1.2 Named pipes para o modelo cliente-servidor**

Os *named pipes* são independentes do modelo de comunicação (caixa de correio, cliente-servidor, difusão, peer-to-peer, etc.). No modelo cliente-servidor, tem-se normalmente a seguinte forma de uso de *named pipes*:

- Um *named pipe* “do servidor”. Este *named pipe* serve para os clientes comunicarem algo ao servidor. Os vários clientes escrevem nesse *pipe* mas apenas o servidor lê.
- Um *named pipe* por cada cliente (*named pipe* “do cliente”). Estes *named pipe* servem para o servidor enviar algo a cada cliente. Só o servidor escreve nestes *named pipes*, em, em cada um deles, apenas um cliente lê.
- O *named pipe* “do servidor” terá uma identificação pré-estabelecida, sendo conhecida em todos os processos envolvidos. A identificação dos *named pipes* “dos clientes” não é conhecida à partida pelo servidor, devendo cada cliente informar o servidor acerca da identificação do “seu” *named pipe*.
- O servidor cria o “seu” *named pipe* (função *mkfifo*). Cada cliente cria o “seu” *named pipe*. Os processos que criaram o *named pipe* são responsáveis por o remover (função *unlink*) quando terminam.

### 1.3 Persistência dos *handles* abertos para os *pipes* de outros processos

Normalmente, cada processo mantém o “seu” *named pipe* (onde lê) aberto para leitura durante toda a sua execução. Quanto aos *named pipes* dos outros processos, nos quais vai escrever, existem duas alternativas:

- Abrir e fechar o *named pipe* de cada vez que o vai usar (para escrita):
  - Mais lento em termos de processamento (mais operações *open* e *close*).
  - Mais simples em termos de estruturas de dados mantidas internamente pelo programa e pelo sistema operativo. Em sistemas como muitos clientes em simultâneo, é a solução mais viável (por exemplo, sistemas *web*).
  - Eventualmente mais simples de implementar para o programador.
  - Esta escolha será mais comum no servidor do que no cliente. **No entanto**, se o servidor precisar de *manter um conhecimento permanente ao longo do tempo acerca dos clientes* (por exemplo, para lhe enviar informação por iniciativa própria sem ser apenas para responder a um pedido<sup>1</sup>), então esta alternativa não é viável.
- Abrir o *named pipe* no primeiro uso, mante-lo aberto, e fechá-lo apenas quando determinar que não vai voltar a interagir com esse processo/*named pipe*.
  - Mais rápido em termos de processamento uma vez que se evita a repetição de *open/close*.
  - Obriga à manutenção em memória de *handles* (descritores de ficheiros) para ficheiros (*named pipes*) abertos. Normalmente existe um limite para o número de *handles* abertos e em sistemas reais com muitos clientes em simultâneo esta alternativa pode ser inviável.

---

<sup>1</sup> Tal como pode acontecer no trabalho prático.

- Eventualmente exige um pouco mais de esforço da parte do programador para manter a informação acerca de cada processo/*named pipe* conhecido. Se se estiver a interagir com apenas um outro processo, a diferença em termos de esforço para o programador é nula e o algoritmo acaba por ficar mais simples. Esta escolha faz todo o sentido no lado do cliente.
- Esta alternativa pressupõe que os processos (clientes) enviam informação uns aos outros (ao servidor) acerca de quando vão deixar de interagir com eles para que esse outro processo possa fechar os *handles* para os *named pipes* envolvidos.

#### **1.4 Modos *blockante* e *não-blockante* dos *named pipes* em Unix**

Quanto à sincronização, os *named pipes* têm dois modos de funcionamento. É importante recordar o material apresentado nas aulas e já disponibilizado no moodle quanto a este aspecto, do qual apenas se vai aqui fazer um breve resumo. Os dois modos de funcionamento são:

- **Modo *blockante* / modo síncrono.** As operações *open*, *read*, *write* podem bloquear o processo que as invoca. Este comportamento simplifica bastante os algoritmos, pois têm-se a garantia que o processo só avança quanto existem condições para tal (por exemplo, um *read* só avança quando a informação pedida está realmente disponível / foi escrita por um outro processo). No entanto existe o perigo de se dois ou mais processos entrarem em espera mútua uns pelos outros e ficarem permanentemente bloqueados.
- **Modo *não-blockante* / assíncrono.** As operações *open*, *read*, *write* não bloqueiam o processo que as invoca. Avançando logo. Este comportamento complica os algoritmos, pois é necessário verificar sempre se efectivamente se obteve o que se pretendia (por exemplo, dados num *read*), e, caso não se tenha obtido, o que fazer, como e quando (voltar a ler? Quando? O que se faz entretanto?). No entanto, este comportamento não coloca o perigo bloqueio mútuo e, em certos casos, pode ser necessário recorrer a uma operação assíncrona precisamente para evitar uma situação de espera mútua. Em princípio, em situações moderadamente simples, não será necessário recorrer a comportamento assíncrono.

Normalmente o modo *blockante* é o que se pretende usar. No modo *blockante* existem dois aspectos mais salientes a ter em atenção:

- A abertura do *named pipes* bloqueia até que outro processo abra também o mesmo *pipe* para a operação inversa (*read* → *write*, *write* → *read*). Este é um dos pontos mais críticos quanto às situações de espera mútua e bloqueio permanente: o avanço de um processo fica logo dependente da existência de outro processo.

Há duas formas para lidar com esta situação:

- Planear cuidadosamente as operações de abertura em ambos os processos cliente e servidor para evitar situações de espera mútua.

- Num dos processos, abrir um dos *pipes* em modo não-bloqueante e depois mudá-lo para modo bloqueante com a função *fcntl*. No entanto, essa operação pode controlar pode afectar operações entretanto já desencadeadas nesse *pipe* por outros processos, quebrando a lógica de comunicação entre eles.
- Tendo um *named pipe* aberto para uma determinada operação (*read* ou *write*), quando deixa de existir outro processo com esse *named pipe* aberto para a operação inversa, o *pipe* reverte para o modo não bloqueante, complicando bastante o algoritmo. Esta situação ocorre, por exemplo quando o servidor se mantém em existência (como é normal) e, momentaneamente, deixa de haver clientes ligados a ele, ou seja, deixa de haver processos como *named pipe* do servidor aberto para escrita, fazendo com que as operações de leitura no servidor passem a ser não-bloqueante.
  - Uma maneira de lidar com esta situação corresponde ao próprio processo abrir o seu *named pipe* também para a operação inversa àquela que precisa para ter a garantia que existira sempre um processo (ele próprio) com o *named pipe* aberto “para a operação inversa”, nunca revertendo este para o comportamento não-bloqueante. Bastará manter o *named pipe* aberto para a “operação inversa” não sendo necessário concretizar essas operações.

As duas situações descritas acima podem ser resolvidas simultaneamente de uma forma muito simples: cada processo (cliente e servidor) vai abrir o seu próprio *named pipe* para leitura e escrita em simultâneo (e em modo bloqueante). Desta forma:

- A função *open* não bloqueia – há logo um processo (ele próprio) com o *pipe* aberto para a operação inversa, e portanto a função retorna logo.
- O processo tem sempre a garantia da existência de um processo (ele próprio) com o *pipe* aberto para a “operação inversa”

## 2. Contexto do exemplo e estratégia de implementação

O exemplo implementa uma situação cliente-servidor. O servidor tem a funcionalidade de traduzir palavras: recebe um pedido com uma palavra e responde com a tradução dessa palavra. Trata-se de uma situação simples, para servir de exemplo com as seguintes características:

- O protocolo de comunicação e interacção entre cliente e servidor não exige a memorização dos clientes por parte do servidor – trata-se de um exemplo simples. Assim, o servidor abre e fecha imediatamente o *pipe* do cliente e não mantém informação acerca deles
- Tanto o cliente como o servidor abrem o seu próprio *named pipe* para leitura e para escrita. No entanto, apenas vão ler. Trata-se de seguir a solução mais simples para (descrita acima) para garantir que há sempre um processo com o *named pipe* aberto para a operação inversa, impedindo-o de entrar em modo não-bloqueante, e garantir que a operação *open* não bloqueia.

### 3. Linhas de trabalho sobre o exemplo

O exemplo pode ser usado como ponto de partida para diversos melhoramentos

- Manter no servidor a informação acerca de cada cliente.
- Prever um protocolo de comunicação (tipos de mensagens) mais complexo, que permita a um cliente informar que está a chegar, terminar, etc.
- Manter os descritores dos *named pipes* sempre abertos.
- Prever uma funcionalidade que envolva o servidor enviar informação aos clientes sem ser na situação de resposta a uma pergunta explícita.
- Permitir notificações assíncronas do servidor para o cliente com o auxílio de sinais
- Permitir que um cliente veja informação originária de outro cliente.
- Permitir que o servidor avise os clientes de que vai encerrar.
- Permitir encerrar o servidor através de um pedido explícito enviado por um cliente.
- Acrescentar a noção de utilizadores e de autenticação.
- Usar uma funcionalidade mais complexa em vez de um simples dicionário, nomeadamente algo que envolva dados que persistam de uns pedidos para outros.

### 4. Implementação e código dos programas

O código dos programas é apresentado a partir da página seguinte.

#### *dicionario-cliente-servidor-codigo.txt*

É suposto colocar o código numa máquina unix para compilar e experimentar, e nomeadamente, seguir as indicações indicadas no ponto 3 deste documento.

O código deve ser organizado em três ficheiros

- dict.h
- servidor
- cliente

Notas:

- O código do exemplo menciona FIFO – trata-se da designação alternativa dos *named pipes*.
- Existem algumas mensagens no ecrã para efeitos de *debug* (podem ser removidas). Também se podem executar os programas com redireccionamento do stderr para /dev/null para não ver essas mensagens.

### dict.h - Definições comuns a incluir em ambos servidor e cliente

---

```
/* ficheiro header necessário aos clientes e servidor */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <cctype.h>

/* nome do FIFO do servidor */
#define SERVER_FIFO "/tmp/dict_fifo"

/* nome do FIFO cada cliente. P %d será substituído pelo PID com sprintf */
#define CLIENT_FIFO "/tmp/resp_%d_fifo"

/* tamanho máximo de cada palavra */
#define TAM_MAX 50

/* estrutura da mensagem correspondente a um pedido cliente -> servidor*/
typedef struct {
    pid_t pid_cliente;
    char palavra[TAM_MAX];
} pergunta_t;

/* estrutura da mensagem correspondente a uma resposta servidor -> cliente */
typedef struct {
    char palavra[TAM_MAX];
} resposta_t;
```

## Código do servidor

---

```
#include "dict.h"
#include <signal.h>

#define NPALAVRAS 7          /* Número de palavras conhecidas */

char * dicionario[NPALAVRAS][2] = { /* O dicionário      */
{ "memory",     "memória" },      /* é constituído      */
{ "computer",   "computador" },   /* por uma matriz      */
{ "close",       "fechar" },       /* bidimensional de   */
{ "open",        "abrir" },        /* ponteiros para      */
{ "read",        "ler" },          /* carácter.           */
{ "write",       "escrever" },     /* [i][0] = palavra   */
{ "file",        "ficheiro" } };    /* [i][1] = tradução */

int s_fifo_fd, c_fifo_fd; /* descritores de ficheiros (pipes) */

/* esta função vai atender o sinal SIGINT para terminar o servidor */
void trataSig(int i) {
    fprintf(stderr, "\nServidor de dicionario a terminar "
                  "(interrompido via teclado)\n\n");
    close(s_fifo_fd);
    unlink(SERVER_FIFO);
    exit(EXIT_SUCCESS); /* para terminar o processo */
}

int main() {
    pergunta_t perg; /* mensagem do "tipo" pergunta */
    resposta_t resp; /* mensagem do "tipo" resposta */

    int i, res;
    char c_fifo_fname[50];
    char * aux;
    printf("\nServidor de dicionario");
    if (signal(SIGINT, trataSig) == SIG_ERR) {
        perror("\nNão foi possível configurar o sinal SIGINT\n");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "\nSinal SIGINT configurado");
```

```

res = mkfifo(SERVER_FIFO, 0777);
if (res == -1) {
    perror("\nmkfifo do FIFO do servidor deu erro");
    exit(EXIT_FAILURE);
}
fprintf(stderr, "\nFIFO servidor criado");

/* prepara FIFO do servidor */
/* abertura read+write -> evita o comportamento de ficar */
/* bloqueado no open. a execução prossegue e as */
/* operações read/write (neste caso APENAS READ) */
/* continuam bloqueantes (mais fácil de gerir) */
s_fifo_fd = open(SERVER_FIFO, O_RDWR); /* *bloqueante */
if (s_fifo_fd == -1) {
    perror("\nErro ao abrir o FIFO do servidor (RDWR/blocking)");
    exit(EXIT_FAILURE);
}
fprintf(stderr, "\nFIFO aberto para READ (+WRITE) bloqueante");

/* ciclo principal: read pedido -> write resposta -> repete */
while (1) { /* sai via SIGINT */
    /* ---- OBTEM PERGUNTA ---- */
    res = read(s_fifo_fd, &perg, sizeof(perg));
    if (res < sizeof(perg)) {
        fprintf(stderr, "\nRecebida pergunta incompleta "
                  "[bytes lidos: %d]", res);
        continue; /* não responde a cliente (qual cliente?) */
    }
    fprintf(stderr, "\nRecebido [%s]", perg.palavra);

    /* ---- PROCURA TRADUCAO ---- */
    strcpy(resp.palavra, "DESCONHECIDO"); /* caso n encontre */
    for (i=0; i<NPALAVRAS; i++)
        if (!strcasecmp(perg.palavra,dicionario[i][0])) {
            strcpy(resp.palavra,dicionario[i][1]);
            break;
        }
    fprintf(stderr, "\nResposta = [%s]", resp.palavra);

    /* ---- OBTEM FILENAME DO FIFO PARA A RESPOSTA ---- */
    sprintf(c_fifo_fname, CLIENT_FIFO, perg.pid_cliente);

    /* ---- Abre FIFO do cliente p/ write ---- */
}

```

```

c_fifo_fd = open(c_fifo_fname, O_WRONLY);
if (c_fifo_fd == -1)
    perror("\nErro no open - Ninguem quis a resposta");
else {
    fprintf(stderr,"\nFIFO cliente aberto para WRITE");

    /* ---- ENVIA RESPOSTA ---- */
    res = write(c_fifo_fd, & resp, sizeof(resp));
    if (res == sizeof(resp))
        fprintf(stderr,"\\nescreveu a resposta");
    else
        perror("\\nerro a escrever a resposta");

    close(c_fifo_fd); /* FECHA LOGO O FIFO DO CLIENTE! */
    fprintf(stderr,"\nFIFO cliente fechado");
}

/* fim do ciclo principal do servidor */

/* em principio não chega a este ponto - sai via SIGINT */
close(s_fifo_fd);
unlink(SERVER_FIFO);
return 0;
} /* fim da função main do servidor */

```

*(código do cliente na página seguinte)*

## Código do cliente

---

```
#include "dict.h"

int main() {
    int s_fifo_fd;           /* identificador do FIFO do servidor */
    int c_fifo_fd;           /* identificador do FIFO deste cliente */
    pergunta_t perg;          /* mensagem do "tipo" pergunta */
    resposta_t resp;          /* mensagem do "tipo" resposta */
    char c_fifo_fname[25];   /* nome do FIFO deste cliente */
    int read_res;

    /* cria o FIFO do cliente */
    perg.pid_cliente = getpid();
    sprintf(c_fifo_fname, CLIENT_FIFO, perg.pid_cliente);
    if (mkfifo(c_fifo_fname, 0777) == -1) {
        perror("\nmkfifo FIFO cliente deu erro");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "\nFIFO do cliente criado");

    /* abre o FIFO do servidor p/ escrita */
    s_fifo_fd = open(SERVER_FIFO, O_WRONLY); /* bloakeante*/
    if (s_fifo_fd == -1) {
        fprintf(stderr, "\nO servidor não está a correr\n");
        unlink(c_fifo_fname);
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "\nFIFO do servidor aberto WRITE / BLOCKING");

    /* abertura read+write -> evita o comportamento de ficar */
    /* bloqueado no open. a execução prossegue logo mas as */
    /* operações read/write (neste caso APENAS READ)           */
    /* continuam bloqueantes (mais fácil)                      */
    c_fifo_fd = open(c_fifo_fname, O_RDWR); /* *bloqueante */
    if (c_fifo_fd == -1) {
        perror("\nErro ao abrir o FIFO do cliente");
        close(s_fifo_fd);
        unlink(c_fifo_fname);
        exit(EXIT_FAILURE);
    }
```

```

fprintf(stderr, "\nFIFO do cliente aberto para READ (+Write) BLOCK");

memset(perc.palavra, '\0', TAM_MAX);

while (1) { /* "fim" para terminar cliente */
    /* ---- a) OBTEM PERGUNTA ---- */
    printf("\nPalavra a traduzir -> ");
    scanf("%s",perc.palavra);
    if (!strcasecmp(perc.palavra,"fim"))
        break;

    /* ---- b) ENVIA A PERGUNTA ---- */
    write(s_fifo_fd, & perc, sizeof(perc));
    /* ---- c) OBTEM A RESPOSTA ---- */
    read_res = read(c_fifo_fd, & resp, sizeof(resp));
    if (read_res == sizeof(resp))
        printf("\nTraducao -> %s", resp.palavra);
    else
        printf("\nSem resposta ou resposta incompreensivel"
               "[bytes lidos: %d]", read_res);
}

close(c_fifo_fd);
close(s_fifo_fd);
unlink(c_fifo_fname);
return 0;
} /* fim da função main do cliente*/

```

# Sistemas Operativos

2020 – 2021

**Sincronização Simples – exclusão mútua**

## Tópicos

Fundamentos de sincronização – Problema da exclusão mútua

Semáforos binários

### Bibliografia específica:

- *Fundamentos de Sistemas Operativos*; 3<sup>a</sup> Ed.; Marques & Guedes
- *Operating Systems Concepts* ; Silberschatz & Galvin

## Sincronização

### Nesta parte da matéria:

- Conceitos sobre sincronização. Problemas de sincronização: como os identificar. Secção crítica
- Conceito de exclusão mútua. Implementação da exclusão mútua a baixo nível
- Conceito de semáforo (usado como mecanismo genérico para resolver problemas de sincronização)
- Conceito de Mutex
- API Mutex (documento à parte e exemplo)

**Nota:** o API de semáforos em Unix não é abordado.

### A conseguir nesta parte a matéria:

- Saber identificar problemas de sincronização (exclusão mútua)
- Saber resolver a nível abstracto com recurso a semáforos/mutexes
- Saber usar o API de mutexes em Unix

Importante: Este documento é um resumo. É importante ver a bibliografia

## Sincronização

### Motivos para sincronização

1. Num sistema no qual existam em execução simultânea várias *entidades* (processos e *threads*) que utilizem recursos e dados partilhados entre si exige a coordenação do acesso a esses mesmos recursos ou dados
2. Qualquer sistema que seja composto por mais do que uma entidade activa em que uma das entidades seja dependente de acontecimentos originados por outra(s) dessas entidades exige a coordenação da sua execução em função da execução dessas outras entidades

### Exemplos típicos da bibliografia

- Gestores de recursos (memória partilhada), gestão de produtores/consumidores

### Exemplos concretos da vida real

- Sistemas gestores de dados, sistemas de utilização simultânea por vários utilizadores (ex.: sistemas de vendas de bilhetes; máquinas ATM)

## Sincronização

Situações típicas (exemplos) que envolvem sincronização

### Cooperação (a ver apenas em SO2)

- Diversas actividades concorrem para a conclusão de uma aplicação comum

### Competição (a ver apenas em SO2)

- Diversos processos competem pela obtenção de um recurso limitado
- A competição deve ser resolvida de forma a que o recurso seja utilizado de forma coerente

### Exclusão mútua (faz parte de SO)

- A utilização concorrente de uma zona de dados partilhada pode levar a que os dados fiquem inconsistentes
- A utilização deve ser feita de uma forma exclusiva: apenas uma entidade activa utiliza o recurso.
- A execução de uma secção de código que manipula dados partilhados constitui uma situação típica de acesso em exclusão mútua

## Sincronização

### Plano do estudo da matéria sobre sincronização

#### 1º Exclusão mútua – SO & SO2

- Pode ser vista como um caso particular de competição:  
Competição pelo acesso exclusivo secção de código (secção crítica)  
O problema que se pretende resolver é normalmente o acesso concorrente a dados

#### 2º Competição (SO2)

- Pode ser vista como uma generalização da exclusão mútua:  
A competição é feita sobre recursos com mais do que uma unidade.  
Pretende simplificar a espera/sinalização de recursos. Não envolve necessariamente secções críticas
- As soluções encontradas para a exclusão mútua irão servir de base para as soluções para a competição

#### 3º Cooperação (SO2)

- Os processos querem sincronizar as suas ações de forma explícita uns com os outros. Em vez de competição para avançar, podem mesmo pedir explicitamente para se suspenderem até receberem uma notificação
- Os mecanismos utilizados para os casos anteriores podem ser utilizados com grande simplicidade para conseguir os objectivos da cooperação

## Sincronização – Exclusão mútua

### O problema na forma mais simples – A exclusão mútua

- Os problemas de sincronização podem ser decompostos a situações simples de exclusão mútua – Se um processo executa uma acção (normalmente sobre dados partilhados) os restantes estão temporariamente excluídos de o fazer
- Nestas situações o problema reside num conjunto restrito de instruções em que existe uma sequência teste-(decisão)-acção. Do resultado do teste irá depender a execução da acção (a qual é em exclusão mutua)
- Idealmente um processo assinalaria os restantes que já se encontraria a executar a acção (fazendo-os abdicar de executar essa acção). O problema ocorre quando o SO retira a execução ao processo que já decidiu executar a acção antes que ele tenha tido tempo de fazer essa sinalização

## Sincronização – Exclusão mútua – Secção crítica

### Exemplo

#### Situação

- Uma função que manipula uma estrutura de dados partilhada (`var_partilhada`).
- O acesso deve ser em exclusão mútua (se um processo escreve, os outros aguardam a conclusão da escrita e eventualmente a própria leitura) para evitar leituras de dados incompletos ou destruição de informação ainda não lida

#### Solução proposta

- O controlo do acesso aos dados é feito através de uma variável indicadora também partilhada (`livre`) mas não vai funcionar pois o teste da variável indicadora `livre` e a subsequente atribuição do valor lógico não-livre a essa variável pode ser interrompido a meio (e então dois ou mais processos poderiam usar a estrutura de dados partilhada `var_partilhada` em simultâneo sem se aperceberem)
- À falta de garantia de funcionamento acresce a utilização de espera activa que ocupa o processador desnecessariamente em testes repetitivos

## Sincronização – Exclusão mútua – Secção crítica

Algoritmo da solução proposta para o exemplo anterior

```

Função Escreve (valor)
Começa
  Enquanto livre é falso |
    Mantém-se no ciclo
    // saiu => livre => avança
    Coloca livre a falso
    var_partilhada <- valor
    coloca livre a verdadeiro
  Termina
  
```

Espera activa. Só por si já invalida esta "solução"

Se for interrompido entre o teste e o assinalar na variável livre, a exclusão mútua pode falhar (até mesmo o teste da condição no ciclo é fálibel)

O problema ocorre quando

Entre descobrir que livre está a 1 (significa pode avançar) e colocar livre a zero (decide avançar e assinala que já não está livre) acontece:

- Ocorre uma comutação de processo
- Outro processo executa
- Esse outro processo vê também a variável livre a 1 e portanto decide avançar
- Ambos os processos já decidiram avançar e não voltam a testar nada
- Ambos os processos vão mexer nos dados e gerar incoerência nestes

O problema está nos dados e não no código. Mas é no código que vai ter que ser encontrada a solução deste problema

## Sincronização – Exclusão mútua – Secção crítica

### Exemplo (código)

```

int escreve(tipo dados valor) {
  while (1)
    if (livre==1) break;
    ←
    livre=0;
    var_partilhada=valor;
    livre=1
}
  
```

Enquanto livre é falso mantém-se no ciclo.

Feito desta forma é espera activa

Se ocorrer uma comutação de processos aqui a exclusão mútua pode falhar

→ Um outro processo veria a variável de controlo livre ainda a 1 e também decidir avançar para a alteração da variável partilhada

→ O problema aconteceu porque a sequência teste-decisão-acção foi divisível

O teste da variável indicadora, a actualização do seu valor e a acção (acesso à estrutura de dados partilhada) é uma secção crítica

## Sincronização – Exclusão mútua – Secção crítica

### **Secção crítica**

- Conjunto de operações que deve ser efectuado de forma atómica (indivisível), normalmente relacionadas com dados partilhados (directa ou indirectamente)

Exemplo típico: acesso a dados em memória partilhada

- A origem da questão que leva a que uma secção de código seja uma secção crítica tem a ver com os dados que são manipulados e não com as instruções propriamente ditas
- Duas zonas distintas de código com aparência completamente dispareira podem estar relacionadas através dos dados que manipulam

## Sincronização – Exclusão mútua

Soluções algorítmicas para a exclusão mútua:

- Existem diversas propostas algorítmicas (no código do próprio processo e sem passar pelo SO). Exemplos: Algoritmos de Lamport, algoritmo de Dekker, etc. (mais pormenores na bibliografia)
- Estas soluções não resolvem totalmente o problema e implicam o aumento de complexidade do código a cargo do programador

Exemplos de problemas que ficam por resolver (dependendo do algoritmo usado):

- Alternância estrita (dois processos acedem alternadamente, limitando-se um ao outro desnecessariamente em muitos casos)
- Espera activa (desperdício intensivo do processador)
- Dependência do número de processos envolvidos (soluções não genéricas)

## Sincronização – Exclusão mútua – Semáforos

**Uma solução para garantir o acesso em exclusão mutua a uma secção crítica**

- Impedir a hipótese de ocorrer comutação de processos a meio do teste-decisão-acção (que poderia levar a que outros processo decidisse a mesma coisa por a variável de teste não estar ainda actualizada)

Como:

- Reduz-se o teste-(decisão)-acção a uma única instrução do tipo test-and-set  
Assim já não poderia ser interrompida e portanto não haver comutação de processos (eventualmente para outro que fosse mexer nesses dados)

Instruções Assembly para esse fim

- Motorola – Instrução TASM – Teste coloca a 1
- Intel – Instrução XCHG – Troca dois valores entre si

## Sincronização – Exclusão mútua – Semáforos

**Utilização da instrução XCHG para efectuar um teste-decisão-acção de forma indivisível para controlar o acesso a uma secção crítica**

Linguagem de alto nível

```
while (ocupado == 1);      /* 1 significa fechado. Outro processo põe a 0 */
teste = 1;                /* põe logo a 1 = ocupado */
/* acede a dados partilhados */
```

Compilado (não optimizado)

```
MOV AX, 1
ciclo: LOCK XCHG DS:[ocupado], AX /* LOCK garante bom funcionamento em SMP */
JNZ endereco           /* ainda tem espera activa, mas ao menos funciona */
/* acede a dados partilhados */
```

**Problema: Ainda mantém a espera activa**

## Sincronização – Exclusão mútua – Semáforos

**Solução ideal para garantir o acesso em exclusão mútua a uma secção crítica e dados**

- Manter a indivisibilidade do teste-decisão-acção e acrescentar:

**Eliminação da espera activa:**

- *O processo deve ficar bloqueado em vez de estar em ciclo de teste*
- Quando puder avançar é acordado e ter a garantia que pode avançar sem testar mais nada

**Problema**

- O SO é a única entidade com poderes garantidos para bloquear ou acordar qualquer processo
- O SO não sabe que um processo está à espera de poder avançar. Um clico de teste são apenas más instruções como tantas outras

**Solução:**

- Transformar a tentativa de acesso a uma zona de código (dados) numa espera de um recurso controlado pelo SO (para o SO saber que o processo está à espera e quando pode avançar)
- O SO bloqueará o processo se esse recurso não estiver disponível e desbloqueá-lo-á quanto voltar a estar disponível
- Este mecanismo deve ser muito simples de usar e genérico
- Existem vários recursos para esse efeito. O mais genérico é o **Semáforo**

## Sincronização – Exclusão mútua – Semáforos

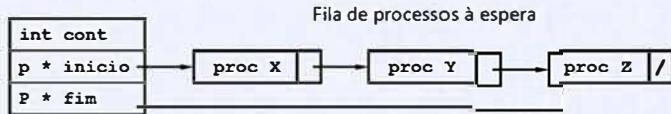
**Semáforo**

- Recurso controlado pelo SO. Semelhante a um semáforo real, mas com a capacidade de ter um "número de autorizações" (número de processos/threads que o podem ter em simultâneo)
- Constituído por uma estrutura de dados que inclui
  - Variável de controlo (inteira) = número de autorizações restantes
  - Fila de espera (de processos bloqueados)
- As operações sobre semáforos são
  - Esperar – Requisita o recurso / Requisita uma autorização / (pede acesso a uma secção crítica)
  - Assinalar – Liberta o recurso / Devolve uma autorização / (liberta a secção crítica)
- Se o semáforo já não tiver mais nenhuma autorização (contador a zero) o processo que efectua o esperar fica bloqueado. Posteriormente é automaticamente desbloqueado quando algum outro processo efectua esperar / devolve uma autorização. Os "esperar" ficam memorizados e pode haver mais que um processo em espera
- O bloqueio / desbloqueio é ser transparente para o processo

## Sincronização – Exclusão mútua – Semáforos

### Semáforos

Aspecto de um semáforo



A implementação dos semáforos no núcleo garante que o teste-decisão-acção é de carácter atómico (indivisível), através de instruções *test and set* ou outro mecanismo

## Sincronização – Exclusão mútua – Semáforos

### Semáforos (em resumo)

- Objectivo: informar o sistema de que o processo está à espera de acesso a um recurso, podendo o SO bloquear a execução do processo até que o recurso esteja livre
- Método: Cada semáforo tem um contador interno. As operações sobre o semáforo são a de **esperar** (diminuir o contador) e **assinalar** (aumentar o contador). Esse contador actua como número de processos que ainda podem passar sem ficarem bloqueados.
- O sistema garante que o teste-decisão-acção interno à implementação do núcleo não é susceptível à ocorrência de comutação de processo, seja por instruções do tipo *test and set*, seja por outros mecanismos disponíveis no núcleo

### Sincronização – Exclusão mútua – Semáforos – Esperar

Utilização de semáforos por parte dos processos

#### Esperar(Semáforo)

- Pede acesso a uma secção crítica (ou do recurso em questão)
- Decrementa a variável de controlo
  - Se ficou a zero, então já outro processo está a utilizar a secção crítica guardada pelo semáforo
    - Neste caso o processo é bloqueado, ficando na lista de espera desse semáforo
  - Caso contrário
    - A secção crítica guardada pelo semáforo estava livre
- Os semáforos para controlo de secções de exclusão mútua são inicializados com a variável de controlo a 1 (apenas um de cada vez pode usar)

### Sincronização – Exclusão mútua – Semáforos – Assinalar

Utilização de semáforos por parte dos processos

#### Assinalar(Semáforo)

- Indica libertação de uma secção crítica (ou do recurso em questão)
- Incrementa a variável de controlo
  - Se houver processos à espera, o primeiro pode avançar
    - O primeiro processo da fila é desbloqueado e o seu estado passa a executável (não é necessariamente posto logo em execução) (\*)
    - A secção crítica (ou recurso) guardada pelo semáforo fica novamente ocupada
  - Caso contrário
    - A secção crítica (ou recurso) guardada pelo semáforo fica livre

(\*) Depende do algoritmo de escalonamento ser ou não preemptivo e da existência de prioridades

## Sincronização – Exclusão mútua – Secção crítica

O exemplo anterior agora com semáforos

```

Semáforo sem = CriarSemaforo(1);
/* ... */
int escreve(tipo_dados valor) {
    esperar(sem);
    var_partilhada=valor;
    assinalar(sem);
}
  
```

Já não há espera activa  
 O processo bloqueia temporariamente se não poder avançar

O SO garante que apenas um (o número inicial no semáforo) processo estará aqui num dado instante

As funções

- CriarSemaforo (cria um semáforo – o parâmetro é o contador inicial)
- Assinalar (operação *assinalar*)
- Esperar (operação *esperar*)

são simplificações das funções realmente existentes nos SO

## Sincronização – Exclusão mútua – Semáforos

- A utilização de semáforos leva ao bloqueio e desbloqueio (transparente para o algoritmo do programa) dos processos que os utilizam

Bloquear um processo envolve

- Retirá-lo de execução
- Salvaguardar o seu contexto (de hardware e de software)
- Marcar o estado do processo como “Bloqueado”
- Colocá-lo no fim da lista de espera

Desbloquear um processo envolve

- Retirá-lo da fila de espera de processos bloqueados onde se encontrava
  - Marcar o estado do processo como “Executável”
  - Colocá-lo no fim da lista de processos executáveis
- O processo será posto em execução quando o despacho o seleccionar

### Mecanismos de apoio à sincronização – Algumas notas

A utilização de semáforos:

- **Bloqueia o processo e não ocupa processador**
- O processo pode ficar bloqueado indefinidamente porque o processador pode continuar a executar outros processos
- É um mecanismo justo
  - Existindo uma fila de espera, obtém-se a garantia de atendimento de todos mais cedo ou mais tarde → permite evitar a situação de *starvation*
- É compatível com implementação de políticas de prioridades nos processos bloqueados
- **Elimina a espera activa** – Consiste na principal vantagem sobre os trincos lógicos (trincos lógicos – ver mais pormenores na bibliografia)
- As funções de manipulação dos semáforos devem pertencer ao S.O.
  - As variáveis internas do semáforo não devem ser directamente visíveis
  - Os nomes das funções não são, obviamente, esperar e assinalar, tendo em alguns sistemas um aspecto bastante mais complicado

### Sincronização – Semáforos

Semáforos e a transição de estados dos processos



Legenda: Transições que podem ser causadas pela utilização de semáforos

As transições associadas ao escalonamento e os estados “Novo” e “Terminado” não se encontram representados.

Os estados dos processos e threads vão ser vistos com mais detalhe mais adiante

## Sincronização

### Exclusão mútua

- Trata-se na verdade de um caso particular de competição por um recurso
  - O recurso é o **acesso a uma secção de código** (na verdade, **dados**)
  - O **número de unidades disponíveis desse recurso é 1**

Pode-se generalizar o conceito de exclusão mútua

- Qualquer recurso pode ser alvo de competição e não apenas secções de código
- O número de unidades disponíveis pode ser qualquer número inteiro maior que zero
- Segundo esta generalização, os semáforos continuam a poder ser utilizados como mecanismo preferencial de gestão do recurso alvo da competição
  - O valor inicial do semáforo é o número inicial de unidades do recurso disponíveis

## Sincronização – Mutex

### Mutex

- Resolve directamente o problema da exclusão mútua (**MUTual EXclusion**)
- Geralmente mais simples de usar que um semáforo
- Normalmente disponível nos sistemas *multithreaded*, tal como o Unix
- É abordado no trabalho prático
- O API e um exemplo encontram-se num documento à parte deste.

## Sincronização – Semáforos Vs. Mutex

### Variáveis condicionais

- Abordam o problema de esperar por um acontecimento sem entrar em espera activa
  - > Não é a mesma coisa que a questão de acesso em exclusão mútua
- Normalmente disponível em todos os sistemas operativos.
- Unix: O API de variáveis condicionais está relacionado com o API das threads.
- **O API e exemplo encontram-se num documento à parte deste.**
- Poderá ser útil no trabalho prático

## Sincronização – Semáforos Vs. Mutex

### Semáforos

- Consiste na generalização do conceito de Mutex, mas em que se permite N processos/threads em simultâneo
  - É útil para resolver situações de coordenação entre processos/threads na gestão de recursos usados por estes
    - Exemplo: produtores/consumidores
    - Se se considerar N = 1, então é como um Mutex
- Normalmente disponível em todos os sistemas operativos.
- Unix: Tem API de semáforos bastante completo, mas não é abordado no trabalho prático e dado apenas a nível conceptual (em SO2 vê-se com detalhe)

## ***Threads POSIX***

### API e exemplo

## ***Mutexes***

### API

Este ficheiro destina-se a ser projectado e, portanto, tem um formato e tamanho de letra diferente do habitual.

#### Tópicos

1. Compilação de programas com threads POSIX
2. Ficheiros header relacionados
3. Estruturas de dados usadas com threads e mutexes
4. Resumo da funcionalidade de threads e mutexes
5. API detalhado – threads e mutexes
6. Questões relativas ao controlo das threads
7. Exemplo - Threads

## 1. Compilação de programas com *threads* posix:

`gcc etcetc.c -pthread`

## 2. Ficheiros header directamente envolvidos

`pthread.h`

## 3. Estruturas de dados mais usadas

- **ID de thread**

`pthread_t`

- **Atributos de thread**

`pthread_attr_t`

- **Mutex para sincronização entre threads**

`pthread_mutex_t`

## 4. API (Resumo)

>> [Threads](#)

- **Criação de uma thread**

`pthread_create`

- **Obter o ID da (própria) thread**

`pthread_self`

- **Terminar a (própria) thread**

`pthread_exit`

- **Terminação de (outra) thread**

`pthread_cancel`

- **Modificar o estado cancel-state (da própria thread)**

`int pthread_setcancelstate`

- **Modificar o tipo de cancel-state (da própria thread)**

`int pthread_setcanceltype`

- **Esperar que uma thread termine**

`pthread_join`

- **Enviar um sinal a uma thread do mesmo processo**

`pthread_kill`

>> Mutexes

## Operações fundamentais

- **Criação / Inicialização**

`int pthread_mutex_init (pthread_mutex_t *,`

```
const pthread_mutexattr_t *restrict attr)
```

- **Eliminação**

```
int pthread_mutex_destroy (pthread_mutex_t *)
```

- **Esperar, bloqueando**

```
int pthread_mutex_lock(pthread_mutex_t *)
```

- **Tentar esperar, sem bloquear (usar só em caso bem justificado)**

```
int pthread_mutex_trylock(pthread_mutex_t *)
```

- **Libertar**

```
int pthread_mutex_unlock(pthread_mutex_t *)
```

## 5. API - Detalhado

### >> Threads

---

#### Criação de uma thread

```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

- **thread**  
Ponteiro para o ID da thread
- **attr**

Atributos iniciais da thread

- **start\_routine**

Ponteiro para a função da thread

- **void \*arg**

Argumento a passar para a função da thread

A *thread* criada ficará imediatamente a executar.

Pode-se passar qualquer valor para o seu argumento desde que ocupe o mesmo número de bits que o ponteiro para *void*. Se for necessário passar mais informação, então passar-se-á um ponteiro para a estrutura com a informação

---

## Obter o ID da (própria) *thread*

```
pthread_t pthread_self(void)
```

O ID da *thread* tem a mesma utilidade que o PID de um processo.

---

## Terminar a *thread* (a própria *thread*)

```
void pthread_exit(void *retval);
```

- **retval**

Valor a usar como código de terminação da *thread*

Este valor pode ser obtido com *pthread\_join*

---

## Terminar (outra) *thread*

```
int pthread_cancel(pthread_t thread)
```

- **thread**

ID da *thread* a terminar

A *thread* alvo terminará ou não consoante o seu estado de cancel-state

-> *Usar apenas em último caso e desde que bem justificado*

Deve evitar-se a terminação abrupta de *threads* com esta função.

É melhor usar a estratégia geral da variável de condição: uma variável que é consultada periodicamente pela *thread* em questão que terminará naturalmente quando detecta que o seu valor mudou (modificada por uma outra *thread* qualquer do programa)

---

## Modificar o estado *cancel-state* (da própria *thread*)

```
int pthread_setcancelstate(int state, int *oldstate)
```

- **state**

Estado desejado

- **oldstate**

Ponteiro para armazenar o estado anterior

PTHREAD\_CANCEL\_ENABLE

PTHREAD\_CANCEL\_DISABLE

---

## Modificar o tipo de *cancel-state* (da própria thread)

```
int pthread_setcanceltype(int type, int *oldtype)
```

- **state**  
Estado desejado
- **oldstate**  
Ponteiro para armazenar o estado anterior

PTHREAD\_CANCEL\_DEFERRED      (*default*)

PTHREAD\_CANCEL\_ASYNCHRONOUS

---

## Esperar que uma *thread* termine

```
int pthread_join(pthread_t thread, void **retval)
```

- **thread**  
ID da thread a esperar
- **retval**  
Ponteiro para o valor de retorno usado pela thread

---

## Enviar um sinal a uma *thread* do mesmo processo

```
int pthread_kill(pthread_t thread, int sig)
```

- **thread**  
Thread alvo (dentro do mesmo processo)
- **sig**

Sinal a enviar

### Efeito

- Envia o sinal *sig* à thread com o ID indicado no primeiro parâmetro.
- O tratamento de sinais é geral ao processo. No entanto, se o sinal estiver a ser tratado por uma função, essa função será executada no contexto da thread indicada.

Usos típicos

- Lidar com situações em que se deseja que uma thread processe imediatamente uma determinada situação mesmo que esteja bloqueada em algo (ex., um *read*), sem, no entanto, forçar a thread a interromper imediatamente

>> Mutex

---

### Inicializar uma variável *mutex*

```
int pthread_mutex_init (pthread_mutex_t * pmutex,  
                      const pthread_mutexattr_t * attr)
```

- **pmutex**  
Ponteiro para a variável *mutex*
- **attr**  
Atributos de inicialização do *mutex*.  
NULL faz com que sejam usados os atributos *default*

É necessário inicializar o *mutex* antes de o utilizar

Alternativamente o *mutex* pode ser inicializado com a atribuição:

```
var_mutex = PTHREAD_MUTEX_INITIALIZER
```

O *mutex* corresponde a um semáforo binário de utilização simplificada no contexto de *threads*

---

### “Des-inicializar” uma variável mutex

```
int pthread_mutex_destroy (pthread_mutex_t * pmutex)
```

- **pmutex**  
Ponteiro para a variável *mutex*

Não corresponde a uma “destruição”: a variável *mutex* continuará a existir, mas já não poderá ser usada (a não ser que seja novamente inicializada).

Esta função é chamada quando já não se deseja voltar a usar o *mutex*. Só deve ser chamada quando o *mutex* não está ocupado.

---

### Auardar e obter a posse do mutex

```
int pthread_mutex_lock(pthread_mutex_t * pmutex)
```

- **pmutex**  
Ponteiro para a variável *mutex*

Esta função bloqueia até que o *mutex* esteja livre.  
Ao avançar, o *mutex* fica ocupado (pela *thread* que invocou a função)

---

## Tentar obter a posse do mutex sem bloquear

```
int pthread_mutex_trylock(pthread_mutex_t * pmutex)
```

- **pmutex**

Ponteiro para a variável *mutex*

Esta função tenta obter a posse do *mutex*. Se o *mutex* estivesse livre, fica ocupado por esta *thread*. Se o *mutex* já estivesse ocupado, a função retorna logo com o código EBUSY

Esta função remete para uma lógica não bloqueante e torna-se menos interessante porque complica as tarefas do programador.

Poderá, no entanto, ser útil quando uma *thread*, por alguma razão, necessite de garantir que não fica bloqueada.

---

## Libertar a posse do mutex

```
int pthread_mutex_unlock(pthread_mutex_t * pmutex)
```

- **pmutex**

Ponteiro para a variável *mutex*

Esta função liberta o *mutex*. Uma das *threads* que estiver à espera do *mutex* poderá avançar (obtendo essa *thread* a posse dele).

## 6. Algumas questões relativas ao controlo das threads

- Muitas vezes é necessário indicar a uma thread que deve terminar. A forma correcta de o fazer é de indicar através de uma variável de controlo que a thread deverá terminar assim que possível (evitar `pthread_cancel`).
- No entanto, a thread pode estar ocupada em algo e não consultar rapidamente (ou de todo) essa variável. Exemplos:
  - Pode estar bloqueada numa leitura de teclado
  - Pode estar bloqueada numa leitura de named pipe
  - Pode estar bloqueada em algo

Como resolver a questão sem usar `pthread_cancel`?

-> A solução é dependente do cenário presente em cada caso e remete o programador (terá que “ser engenheiro/a”)

### Exemplos:

- Se a *thread* estiver bloqueada numa leitura de pipe, então a parte do programa que está a indicar à *thread* que deve terminar pode, por exemplo enviar uma informação *mock-up* para esse pipe de forma a que a *thread* saia do *read* e consulte a variável de controlo. A informação enviada para o *pipe* pode precisamente ter o significado “*thread* deve terminar”.
- **Este é apenas um exemplo. Haverá muitas outras estratégias**

**Outra ideia, mais genérica e mais facilmente reutilizável em diversos cenários:**

- Usar a função `pthread_kill`

Esta função permite a um programa enviar um sinal a uma thread **do mesmo programa**. O sinal é atendido (em código, se estiver a ser tratado) no

contexto da *thread* alvo. O tratamento do sinal é feito nos moldes habituais, com as propriedades “desbloqueantes” já vistas anteriormente.

## 7. Exemplo:

Este exemplo ilustra:

- Criação de *threads*
- Ordenar a terminação de *threads* sem usar *pthread\_cancel*
- “junção” de *threads* = “uma *thread* aguardar que outra termine”

```
#include <stdio.h>
#include <string.h>
#include <pthread.h> // Notar este include
#include <stdlib.h>
#include <unistd.h>

/* dados de controlo para cada thread */
/* definidos pelo programador - conforme o que for preciso */

typedef struct {
    int continua;
    char caracter;
    int vezes;
    pthread_t tid; /* ID da thread */
    void * retval; /* Código de terminação */
} TDados;

/* função de suporte à(s) thread(s) */

void * imprime(void * arg) {
    int i;
    TDados * dados = (TDados *) arg;

    for (i=0; i<dados->vezes; i++) {
        printf("%c ", dados->caracter);
        fflush(stdout); /* para o carácter aparecer logo */
        sleep(1);
        if (dados->continua == 0)
            break;
    }
    printf(" thread %c terminou ", dados->caracter);
```

```

    return NULL;
}

#define NUMTHR 3

int main() {
    int res, i;
    char temp[30];
    TDados workers[NUMTHR];

    for (i=0; i<NUMTHR; i++) {
        workers[i].continua = 1;
        workers[i].vezes = (i+1)*10;
        workers[i].caracter = 'A'+i;
        res = pthread_create(
            & workers[i].tid, // & variavel p/ ID da thread
            NULL,             // atributos default
            imprime,          // funcao da thread
            (void *) &workers[i]); // argumento -> ptr para
                               // dados da thread ( melhor: workers + i )
        if (res != 0) {
            perror("\nErro na criação da thread");
            exit(1);
        }
    }

    printf("\nmain: estou a trabalhar. "
           "Escreve coisas, sair para terminar\n");
    while (1) {
        fgets(temp, 30, stdin);
        if (temp[strlen(temp)-1] == '\n')
            temp[strlen(temp)-1] = '\0';
        printf("disseste: [%s]\n", temp);
        if (strcmp(temp,"sair")==0)
            break;
    }

    /* indicar às threads para terminarem */
    for (i=0; i<NUMTHR; i++)
        workers[i].continua = 0; // nunca terminar thread à força
}

```

```

/* esperar que as threads terminem mesmo */
/* sleep(1) -> Nunca usar sleeps para esperar fim de threads*/

/* forma correcta = pthread_join */
for (i=0; i<NUMTHR; i++)
    pthread_join(workers[i].tid, & workers[i].retval);

printf("\ntodas as threads terminaram\n");
printf("\nmain a encerrar\n");
return 0;
}

```

## Acerca do exemplo

- Não inclui sincronização (mutexes, semáforos, etc.)
- É bastante simples – apenas o suficiente para mostrar criação e junção de *threads*
- As tarefas das duas *threads* são tão parecidas que poderiam ser mais logicamente feitas pela mesma função em vez de duas funções separadas como está agora  
TPC: melhorar este exemplo para usar apenas uma única função para ambas as *threads*

## Outros exemplo com threads

(aulas ou outros documentos)

- Leitura simultânea de dados de várias fontes com **threads**
- Exemplo de variáveis condicionais, que usa **threads** e **mutexes**

Nota. Este documento destina-se a ser projectado como se fossem slides. Por essa razão o formato é diferente do habitual, incluindo o tamanho da letra e organização do texto.

## Variáveis Condicionais

### Conceito, API e exemplo

Contexto: *Threads, Sincronização*

Este ficheiro destina-se a ser projetado e, portanto, tem um formato e tamanho de letra diferente do habitual.

#### Tópicos

1. Compilação de programas com uso de variáveis condicionais
2. Ficheiros header usados
3. Conceito de variáveis condicionais
4. API
5. Exemplo

#### 1. Compilação de programas com variáveis condicionais:

As variáveis condicionais são usadas no contexto de *threads* e o API está incluído no das *threads*.

Para compilar um programa com *pthreads* é necessário indicar a bibliotecas *pthread*

**gcc etcetc.c -pthread**

#### 2. Ficheiros header diretamente envolvidos

**pthread.h**

### 3. Conceito de Variáveis Condicionais

- As variáveis condicionais são um mecanismo que permite a uma **thread aguardar um acontecimento sem usar espera ativa**. O acontecimento será assinalado por uma outra *thread* do mesmo processo.
- As variáveis condicionais **não são uma alternativa a mutexes**.
  - Os mutexes resolvem o problema do acesso em exclusão mútua.
  - As variáveis condicionais resolvem um problema diferente: **aguardar um acontecimento sem ocupar o processador**.
- As variáveis condicionais são usadas normalmente em conjunto com um mutex pela razão de que existe uma **variável de controlo avança/não avança** que fará, necessariamente, parte de uma **secção crítica**.

Cenário de aplicação

- O ciclo onde a espera (ativa) ocorre (uma *thread*) testa repetidamente uma variável de controlo. **Trata-se de espera ativa que o uso da variável condicional resolverá**
- A sinalização (outra *thread*) corresponde a atualizar a variável de controlo. **Esta thread continuará a executar, inabilitizando o uso de phread join para a deteção do acontecimento pronto pela outra thread**
- O acesso concorrencial a esta variável de controlo implicaria, normalmente, um mutex (sem exclusão de outros dados que justifiquem a existência desse mutex)

#### Em resumo – variáveis de controlo

- Permitem eliminar um teste de espera ativa em situações onde uma *thread* aguarda um dado acontecimento.

## Exemplo em duas partes: sem e com variáveis condicionais

“Thread A aguarda por um acontecimento produzido pela *thread B*”

### Parte 1 – Implementação sem usar variáveis condicionais

A lógica da primeira implementação recorre ao teste em espera ativa

- A *thread A* testa repetidamente a variável de controlo “flag\_partilhada”
- Usa um *mutex*, dado que a variável de controlo é partilhada
- Tem que obter e libertar repetidamente o *mutex*, caso contrário a outra *thread* não poderia nunca atualizar a variável de controlo
  - Esta solução é péssima  
Tem **espera ativa** e ainda um padrão deselegante de *liberta-readquiere* contínuo e repetitivo do *mutex*
- A *thread B* assinalará a variável de controlo, acedendo primeiro ao *mutex* que a guarda

#### Thread que aguarda

```
while (1) {
    pthread_mutex_lock(Mutex_A);
    // ... dentro de secção crítica
    if (flag_partilhada == 1)
        break;
    pthread_mutex_unlock(Mutex_A);
} // break->precisa sair da
sec.c
pthread_mutex_unlock(Mutex_A);
```

#### Thread que assinala

```
//...
pthread_mutex_lock(Mutex_A);
// ... dentro de secção crítica
flag_partilhada = 1;
// ... dentro de secção crítica
pthread_mutex_unlock(Mutex_A);
// ...
```

- A única hipótese que a *thread B* tem para executar e mudar *flag\_partilhada* é entre o breve período entre *unlock* e *lock* seguinte da *thread A*.
- O código da *thread A*, nesta solução, é bastante deselegante.

A solução correta passa pelo uso de uma **variável condicional** que faz a gestão do acesso à *flag* partilhada e do acesso à secção crítica de uma forma que não causa espera ativa

## Parte 2 – Implementação com uma variável condicional

- É necessário um *mutex* na mesma – para gerir o acesso à variável condicional

### Thread A – a que aguarda

- A *thread* que aguarda começa por esperar no *mutex*
  - O uso de variável condicional implica sempre o uso de um *mutex*
- De seguida espera na variável condicional.
  - Se a variável condicional não estiver assinalada (“pronta”), a *thread* entra no estado bloqueado (“em espera”) e o *mutex* é automaticamente libertado
  - Quando a variável condicional for assinalada pela outra *thread*, é feita uma espera automática no *mutex* e a *thread* avança apenas quando este estiver livre
- No final, a *thread* está na posse do *mutex*, que deverá libertar

### Thread B – a que assinala

- Para assinalar a variável condicional precisa primeiro de aceder ao *mutex*
  - Este deverá estar livre, libertado quando a *thread* que aguarda aguardou na variável condicional
  - No final da assinalar a variável condicional a *thread* liberta o *mutex*, permitindo à outra *thread* avançar na espera pelo *mutex* implícita que ocorre após a espera na variável condicional

#### Thread que aguarda

```
// ...
pthread_mutex_lock(Mutex_A);
// dentro de secção crítica
pthread_cond_wait(Cond_Var);
// secção crítica é recuperada
pthread_mutex_unlock(Mutex_A);
// ...
```

#### Thread que assinala

```
//...
pthread_mutex_lock(Mutex_A);
// ... dentro de secção crítica
pthread_cond_signal(Cond_Var);
// ... secção crítica é recuperada
pthread_mutex_unlock(Mutex_A);
// ...
```

-> Nesta solução a espera ativa removida. As duas *threads* competem pela secção crítica cuja semântica é preservada sem ocupar o processador.

No exemplo anterior

- A função `pthread_cond_wait` e `pthread_signal_var` foram apresentadas de forma esquemática -> os parâmetros reais são ligeiramente diferentes
- O uso de variável condicional é feito em associação com um mutex, que, neste caso, já existia no código inicial

De uma forma esquematizada

**Na thread que aguarda**

```
// Thread que aguarda  
pthread_mutex_lock (&Mutex_A);  
  
// liberta implicitamente o mutex e aguarda  
pthread_cond_wait (&[redacted], &Mutex_A);  
// readquire o mutex automaticamente  
  
// Resto da secção crítica e posterior libertação  
// do mutex  
pthread_mutex_unlock (&Mutex_A);
```

**Na thread que assinala o acontecimento**

```
// o mutex terá ficado livre pelo uso de pthread_cond_wait  
// na thread que aguarda permitindo a esta avançar  
pthread_mutex_lock (& Mutex_A);  
  
// assinala a variável condicional permitindo à thread  
// que aguarda avançar ("acorda-a / acorda-as")  
pthread_cond_signal (&[redacted]);  
// assinala uma das threads que aguardam a variável.  
// Essa thread terá ainda que aguardar pelo mutex  
// (alternativa: pthread_cond_broadcast acorda todas as  
// threads que estivessem a aguardar. Essas threads  
// competirão depois pelo mutex)  
pthread_mutex_unlock (& Mutex_A);
```

- Nestes dois últimos excertos de código as funções `pthread_cond_wait` e `pthread_cond_signal` já são apresentadas usando os parâmetros reais

O ciclo de espera ativa na *thread* que aguarda pode ter outra aparência

Exemplo:

```
// thread que aguarda
pthread_mutex_lock (& Mutex_A);
while ( 1 ) {
    // liberta mutex para que outra thread possa
    // mudar a variável sinalizadora
    pthread_mutex_unlock(& Mutex_A);
    // faz algo / espera explicitamente
    // (tenta) volta a adquirir o mutex
    pthread_mutex_lock(& Mutex_A);
    // testa variável e sai se for caso disso
    if (flag_partilhada == 1) // "1", ou outro valor
        break;
}
// fim de secção crítica - liberta mutex
pthread_mutex_unlock(& Mutex_A);
```

Independentemente da aparência, nota-se a existência de um ciclo fechado em teste exaustivo, ocupando o processador (espera ativa)

## Em resumo

- As variáveis condicionais permitem transformar o padrão de código de espera por um acontecimento produzido por outra *thread* noutro em que foi removida a espera ativa
- Não estão relacionadas com a solução de problemas de acesso em exclusão mútua

## 4. API – Variáveis condicionais

### Tipo de dados para variável condicional: `pthread_cond_t`

Declaração de uma variável condicional “MyCondVar”

```
pthread_cond_t MyCondVar;
```

### Inicialização de variável condicional

Usar a função seguinte

```
pthread_cond_init (pthread_cond_t *)
```

ou então usar a simples atribuição

```
pthread_cond_t MyCondVar = PTHREAD_COND_INITIALIZER;
```

(a forma anterior é preferível)

### Destruição de variável condicional (“des-inicialização”)

```
pthread_cond_destroy (pthread_cond_t *)
```

### Esperar numa variável condicional

```
pthread_cond_wait (pthread_cond_t *,pthread_mutex_t *)
```

- Liberta *mutex* associado e espera na variável (até ficar “sinalizada”)
- Quando a variável é assinalada, o *mutex* é readquirido automaticamente em competição com a *thread* que assinalou a variável -> a semântica de secção crítica é preservada

```
pthread_cond_timedwait (pthread_cond_t *,
                         pthread_mutex_t *, const struct timespec *)
```

- Como a anterior, mas com um *timeout* de tempo máximo de espera

## Sinalização da variável

(=dar permissão a outra(s) *thread(s)* que aguardava(m) para avançar)

```
pthread_cond_signal (pthread_cond_t * )
```

- Acorda uma das *threads* que aguardava nesta variável condicional. A *thread* que é assinalada compete depois pelo acesso ao *mutex*, preservando-se assim a semântica de secção crítica

```
pthread_cond_broadcast (pthread_cond_t * )
```

- Acorda todas as *threads* que aguardam nesta variável condicional. Essas *threads* terão ainda que adquirir o *mutex*, preservando-se a lógica da secção crítica

## 5. Exemplo

Este exemplo ilustra o uso de variáveis condicionais

### Cenário do exemplo

*"Existe uma thread que aguarda que sejam produzidos valores com determinadas características. Os valores estão a ser procurados por outra(s) thread(s). Trata-se de um exemplo e a produção de valores é baseada em números aleatórios"*

- São criadas uma ou mais *threads* que produzem valores aleatórios entre 0 e 99. Estas são as *threads* que produzem algo que deve ser aguardado (são as *threads* que “assinalam”)
  - Quando o valor sorteado se encontra numa determinada gama (indicada a cada *thread*), é considerado que ocorreu um acontecimento que deve ser assinalado à *thread* que aguarda. É necessário:
    - Disponibilizar o valor encontrado e uma identificação acerca de qual a *thread* que o encontrou
    - Assinalar o acontecimento à *thread* que aguardaA *thread* continua a trabalhar à procura do próximo valor
- A *thread* que executa a função *main* cumpre o papel de *thread* que aguarda. Quando é encontrada uma determinada quantidade de valores, esta *thread* indica às outras que podem parar de procurar
- O uso de *threads* segue a lógica canónica:
  - Evita o uso de variáveis globais através do uso do ponteiro parâmetro da função da *thread*
  - Evita o uso de *pthread\_kill*, socorrendo-se de uma flag “*deve\_continuar*” que é analisada periodicamente pelas *threads*

## Situações de espera/sinalização

- A função que executa a função *main* é a função que aguarda um acontecimento.
- Esse acontecimento é o surgimento de um valor sorteado dentro da gama pretendida (configurada e indicada à *thread* que o procura)
- A *thread* da função *main* **aguarda** por esse acontecimento através de uma **variável condicional**.
- A *thread* que aguarda decide a determinada altura que já não precisa de mais valores, indicando às restantes que devem terminar. No entanto, após essa indicação aguarda elas várias *threads* com *pthread\_join* dado que as *threads* envolvidas não reagem imediatamente.

## Estruturas de dados

### - Recursos comuns a todas as *threads*

- Dados do valor encontrado
- Mutex
- Variável condicional
- Flag *deve\_continuar*

### - Recursos específicos a cada *thread*

- Configuração específica de cada *thread*
  - Identificação, duração de pausa, etc.
- Ponteiro que conduz aos dados comuns

O uso destas estruturas segue as linhas gerais já explicadas no contexto da matéria sobre *threads*

Neste exemplo não são verificadas situações de erro nas chamadas sistema invocadas para manter o código mais claro e focado no que se pretende transmitir.

-> É evidente que em código real é obrigatório analisar todos os valores de retorno e possíveis situações de erro (exemplo “código de real”: trab. prático)

Para compilar o programa do exemplo: “linkar” com pthread

Assumindo que o ficheiro de código fonte se chama exemplo\_vc2.c

```
gcc -pthread exemplo_vc2.c -o exemplo_vc2
```

## Código fonte

```
// Ficheiros header usados
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
```

Dados comuns a todas as threads

```
// informação comuns a TODAS as threads
// - o mutex e a variável condicional são obviamente
//   comuns a todas as threads
// - a flag deve_continuar pode ser diferente para
//   cada thread. A decisão tem a ver com o facto de
//   se desejar ou não mandar parar todas vs. individualmente
// - Quem e Val são a informação a partilhar com a thread
//   que aguarda e o seu acesso é feito em exclusão mútua
```

```
typedef struct {
    char Quem;
    int Val;
    pthread_mutex_t mutex;
    pthread_cond_t vcond;
    int deve_continuar; // afecta todas as threads
} SharedData;
```

Dados específicos a cada thread

```
// Informação específica a cada thread
// - configuração de cada thread: identificação (myChar)
// - gama de interesse pretendido para os valores (myProb)
// - Pausa entre dois sorteiros (mySleep)
// Inclui o ponteiro que conduz à estrutura com os dados comuns
typedef struct {
    char myChar;
    int myProb;
    int mySleep;
    SharedData * comum;
    // também podia ser uma deve_continuar por cada thread
    pthread_t myTID;      // thread ID
    int myID;
} ThreadData;
```

Configuração do número de *threads* (mudar 1 para outra coisa)

```
#define NUM_THREADS 1
```

Função da thread que assinala

```
void * funcaoThread(void* info) {
    ThreadData * myInfo = (ThreadData *) info;
    int num;
    while (myInfo->comum->deve_continuar) {
        sleep(myInfo->mySleep); // para se ter tempo para ver
        num = random() % 100;
        if (num <= myInfo->myProb) {
            pthread_mutex_lock(& myInfo->comum->mutex);
            myInfo->comum->Quem = myInfo->myChar;
            myInfo->comum->Val = num;
            pthread_cond_signal(& myInfo->comum->vcond);
            // neste exemplo não há nada de especial a fazer aqui
            pthread_mutex_unlock(& myInfo->comum->mutex);
        }
    }
    printf("\nThread %c a terminar\n", myInfo->myChar);
    return NULL;
}
```

A função *main* lança as *threads* que assinalam e depois entra em espera por elas, cumprindo o papel de *thread* que aguarda

```
int main() {
    SharedData sd;
    ThreadData td[NUM_THREADS];
    int total = 3; // número de valores a encontrar
    int localVal;
    char localQuem;
    int i;

    srand(time(NULL));
```

Preparação do mutex e variável condicional

```
// sd.mutex = PTHREAD_MUTEX_INITIALIZER;
// sd.vcond = PTHREAD_COND_INITIALIZER;
// usar pthread_..._init() é melhor
pthread_mutex_init (& sd.mutex, NULL);
pthread_cond_init( & sd.vcond, NULL);
```

Preparação e lançamento das *threads* que produzem os valores e assinalam

```
sd.deve_continuar = 1;

printf("Thread Inicial a correr a função main\n");

for (i=0; i<NUM_THREADS; i++) {
    td[i].myChar = 'A' + i;
    td[i].mySleep = 1;
    td[i].myProb = 25;
    td[i].comum = & sd;
    pthread_create( & td[i].myTID, NULL, funcaoThread, td+i);
}
printf("Outras thread lancadas\n\n");
```

A função *main* entra agora na lógica de *thread* que aguarda, esperando pela produção de valores. Para cada valor, a *thread* aguarda sem espera ativa, ficando bloqueada. Desbloqueia quando lhe é assinalado o acontecimento “valor encontrado”, e repete até que a quantidade de valores atinja o número desejado

```
while (total > 0) {
    pthread_mutex_lock( & sd.mutex );
    pthread_cond_wait( & sd.vcond, & sd.mutex );
    localVal = sd.Val;
    localQuem = sd.Quem;
    --total;
    pthread_mutex_unlock(& sd.mutex);
    printf("Thread \"main\" acordou:\n");
    printf("Encontrado %d por %c\n\n", localVal, localQuem);
}
```

Tendo obtido a quantidade de valores pretendido, esta *thread* vai indicar às restantes que devem parar, aguardando que realmente terminem antes de prosseguir

```
sd.deve_continuar = 0;
for (i=0; i<NUM_THREADS; i++)
    pthread_join(td[i].myTID, NULL);
printf("\nTodas as threads terminaram\n\n");
```

O processo termina de seguida. Caso continuasse seria importante “desconstruir” o *mutex* e a variável condicional, dado que já não iriam ser usados novamente

```
return 0;
}
```

## Notas

- Para experimentar este exemplo é importante escrever o código em vez de apenas fazer *copy & paste* para melhor interiorizar o que está a fazer
- Desafio: existirá alguma uma possibilidade do valor produzido por uma *thread* nunca chegar a ser visto pela *thread* que aguarda? Se sim, como é que isso pode acontecer e o que poderia ser feito para melhorar a situação?