

Programação em UNIX

Introdução



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Objectivos

No final desta aula, os alunos devem ser capazes de:

- Explicar a diferença entre chamadas a funções da API e da Biblioteca standard de C
- Compilar um programa em C e usar o manual *online*
- Aceder aos argumentos da linha de comandos e às variáveis de ambiente
- Tratar situações de erro de execução de um programa
- Medir tempos de execução de um programa
- Descrever o percurso "normal" de execução de um programa em C



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

INTRODUÇÃO

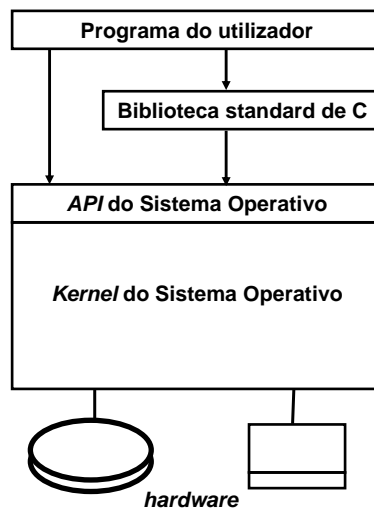
- Os programas pedem serviços ao Sistema Operativo através de chamadas ao sistema.
- Uma chamada ao sistema é um ponto de entrada directa no *kernel*.
- O *kernel* é um conjunto de módulos de *software* que executam em *modo privilegiado*, significando que têm controlo total sobre os recursos do sistema.
- As funções da biblioteca de C podem ou não invocar chamadas ao sistema.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Manual do UNIX

A maior parte dos sistemas UNIX têm documentação *online*, as designadas *man pages* (páginas do manual).

Tradicionalmente, estas páginas estão divididas em secções:

- 1- *user commands*
- 2- *system calls*
- 3- *C library functions*
- ...
- 8- *system maintenance*
- ...

As páginas do manual referem-se aos itens colocando o número da secção entre parêntesis.

- Exemplo: *open (2)*



Cada página também está organizada em secções:

- HEADER: o título da página em questão
- NAME: um sumário
- SYNOPSIS: descreve o uso
- AVAILABILITY: indica se está disponível no sistema
- DESCRIPTION: descreve o que faz o comando ou a função
- RETURN VALUES: indica os valores retornados (se aplicável)
- ERRORS: sumariza os valores de *errno* e as condições de erro
- FILES: lista os ficheiros que o comando ou a função usa
- SEE ALSO: lista comandos/funções relacionadas ou outras secções
- ENVIRONMENT: lista variáveis de ambiente relevantes
- NOTES: informação acerca de utilizações pouco usuais ou características de implementação
- BUGS: lista problemas conhecidos



As páginas do manual podem ser consultadas com o utilitário *man*.

- *man [section] word*
- *man -k keyword*

Exemplos: executar os seguintes comandos e interpretar o resultado

- *man ls*
- *man man*
- *man intro*
- *man write*
- *man 1 write*
- *man 2 write*
- *man -k mode*

A página de *write(1)* contém informação sobre um comando.

A página de *write(2)* descreve uma chamada ao sistema.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Chamadas ao sistema e Funções da biblioteca de C

Quando se usam chamadas ao sistema ou funções da biblioteca de C, convém consultar o manual para saber o protótipo da função, as *header files* necessárias, os parâmetros a incluir na chamada e o tipo de resultado obtido.

Exemplo: página do manual referente à função *write(2)*

- SYNOPSIS

```
#include <unistd.h>
```

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

A página indica que esta função escreve *nbyte* bytes de *buf* para um ficheiro especificado por *fildes* e que retorna o nº de bytes efectivamente escritos



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Compilação de programas

O compilador de C, *cc* ou *gcc*,
traduz programas fonte em C
em módulos objecto ou em módulos executáveis.

A compilação é feita em várias etapas:

- o preprocessador expande macros e inclui as *header files*
- o compilador faz vários passos pelo código traduzindo-o primeiro p/ linguagem *assembly* da máquina alvo e depois para linguagem máquina
- o resultado é um módulo objecto constituído por código máquina e tabelas de referências por resolver
- este módulo objecto é ligado a outros módulos para formar um executável em que todas as referências estão resolvidas



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Programa em C:

```
/* Programa hello.c */

#include <stdio.h>

int main(void)
{
    printf("Hello world !\n");
    exit(0);
}
```

Compilar:

ou

Executar:

> cc hello.c

> cc -o hello hello.c

> ./hello
Hello world !
>

Consultar no manual de *cc* ou *gcc*
outras opções do compilador.
Recomenda-se a utilização da opção *-Wall*

• por omissão,
o executável fica no
ficheiro *a.out*

• se o directório actual
não estiver no *PATH*
é preciso acrescentar *./*

• *PATH = \$PATH:.*
(na linha de comandos)
acrescenta o directório actual
ao *PATH*

• alternativa: editar o ficheiro
.profile, *.bash_profile*,
ou equivalente e fazer login



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Desenvolvimento de programas

Header files

- Para programar em C, precisamos de *header files* que contêm definições de constantes e declarações de chamadas ao sistema ou à biblioteca da linguagem.
- A maior parte destas *header files* estão localizadas em `/usr/include` e seus subdirectórios.
- É possível especificar outros directórios onde devem ser procurados *include files*, para além dos directórios *standard*, usando o *switch* de compilação `-I`

```
> cc -I/usr/myname/include -o prog1 prog1.c
```

- Para procurar *header files* contendo certas definições ou protótipos de funções pode usar-se o comando `grep`

```
> grep _SC_CLK_TCK /usr/include/*.h
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Library files

- *Library* (biblioteca) - colectânea de funções pré-compiladas que foram escritas de modo a serem reutilizáveis.
- As bibliotecas *standard* estão em `/lib` ou `/usr/lib` ou `usr/local/lib`
- Os nomes das bibliotecas começam sempre por `lib`. O resto do nome indica o tipo de biblioteca (ex: `libc`, indica a biblioteca de C, e `libm`, a biblioteca matemática). A última parte do nome indica o tipo de biblioteca:

» `.a` - biblioteca estática

- se houver vários programas que usem uma mesma função de uma biblioteca, quando os programas estiverem a correr "simultaneamente" existirão várias cópias da função em memória

» `.so` ou `.sa` - biblioteca partilhada

- o código das funções da biblioteca pode ser partilhado por vários programas

» Em Linux, por omissão, são usadas as bibliotecas partilhadas. Para forçar a utilização de bibliotecas estáticas deve-se incluir a opção `-static` ao invocar o compilador de C.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Quando se solicita ao S.O. a execução de um novo programa este começa por executar uma rotina, designada *C startup* (no caso da linguagem C)

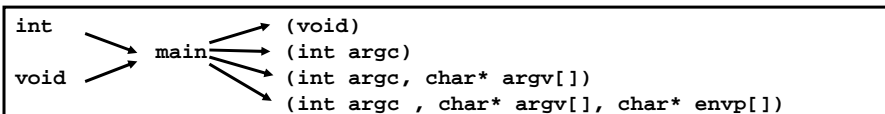
- vai buscar ao *kernel* os argumentos da linha de comandos e das variáveis de ambiente
- abre e disponibiliza 3 "ficheiros" ao programa (*standard input*, *standard output* e *standard error*)
- invoca a função `main()` do programa



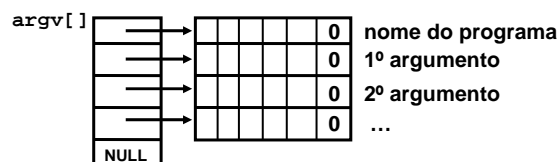
MIEIC

Faculdade de Engenharia da Universidade do Porto

Pode ser definida de muitas formas:



envp - *array* de apontadores p/*strings*, apontando variáveis de ambiente do programa



MIEIC

Faculdade de Engenharia da Universidade do Porto

Terminação de um processo

Terminação normal:

- executar `return` na função `main()`
- invocar `exit()`
- invocar `_exit()`

Terminação anormal:

- invocar `abort()`
- quando recebe certos sinais (não tratáveis) (v. adiante)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Terminação com `exit()`

```
#include <stdlib.h>
void exit(int status);
```

(ANSI C)

- Termina imediatamente o programa retornando o código de terminação `status` para o S.O.
- Liberta todos os recursos atribuídos ao programa, fecha os ficheiros abertos e transfere dados que ainda não tenham sido guardados p/ o disco

`status`

- a maior parte dos sistemas operativos permite testar o `exit status` do último processo executado
 - ex: `> echo $status` (na *C shell*; `$?` na *Bourne* e na *Korn shell*)
- valores habituais
 - 0 (zero) - se não aconteceu erro
 - `<>0` - se aconteceu erro



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Terminação com `_exit()`

```
#include <stdlib.h>
void _exit(int status);
```

(POSIX)

- Termina imediatamente o programa retornando o código de terminação `status` para o S.O.
- Liberta todos os recursos atribuídos ao programa, de forma rápida
- Podem ser perdidos dados que ainda não tenham sido guardados

O `status` fica indefinido se

- `exit()` ou `_exit()` forem invocadas sem especificar `status`
- `main()` fizer `return` sem especificar o valor de retorno
- `main()` atingir o fim sem fazer `return`, `exit` ou `_exit`



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

A função `atexit()`

A função `exit()` pode executar, antes de terminar, uma série de rotinas (*handlers*), que tenham sido previamente registadas para execução no final do programa.

Estas rotinas são executadas por ordem inversa do seu registo.

O registo destes *handlers* é feito através da função `atexit()`

```
#include <stdlib.h>
int atexit(void (*func) (void));
retorno: 0 se OK; <>0 se erro
```

O argumento é o end^o de uma função sem argumentos que retorna void

exemplo:

```
...
if (atexit(exithand2)!=0) {
...}
...
```

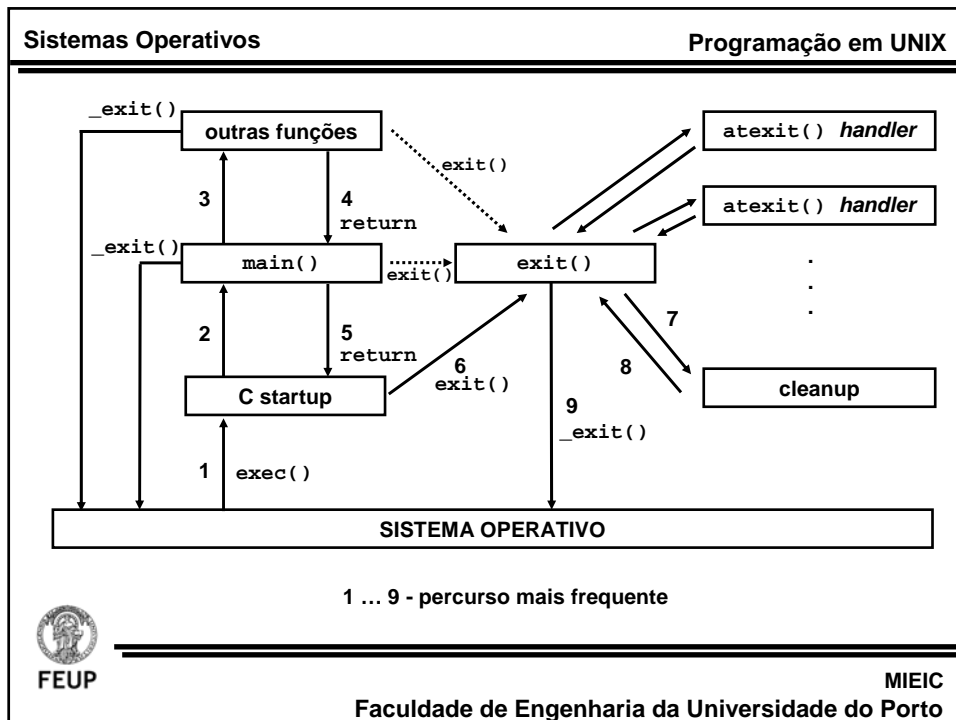
```
static void exithand2(void) {
...
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto



Sistemas Operativos
Programação em UNIX

Tratamento de erros

- Em geral, as chamadas ao sistema retornam um valor especial quando acontece um erro, por exemplo:
 - um valor negativo (frequentemente -1)
 - um apontador nulo.
- O tipo de erro que ocorreu é colocado numa variável global `errno`, do tipo `int`.
- O valor desta variável deve ser analisado imediatamente após a chamada que originou o erro.
- O ficheiro `errno.h` define
 - a variável `errno`
 - constantes para cada valor que `errno` pode assumir

ex.:

```

#define EPERM    1 /* Not owner */
#define ENOENT   2 /* No such file or directory */
#define ESRCH    3 /* No such process */
            
```

MIEIC
Faculdade de Engenharia da Universidade do Porto

Tratamento de erros (cont.)

Funções da biblioteca de C, úteis quando ocorrem erros:

```
#include <stdio.h>

void perror (const char *msg);
```

- Mostra a *string* msg, seguida de ": ", seguida de uma descrição do último erro que ocorreu numa chamada ao sistema.
- Se não houver erro a reportar, mostra a *string* "Error 0".

```
#include <string.h>

char *strerror (int errnum);
```

- Esta função retorna um apontador para uma *string* que contém uma descrição do erro cujo código foi passado no argumento errnum (que é tipicamente o valor de errno)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Medida de tempos de execução

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Preenche a estrutura cujo endereço se fornece em buf
Retorna o tempo actual do sistema, medido a partir de um instante arbitrário, passado (ex: o arranque do sistema; o instante depende da versão do S.O.)

```
struct tms {
clock_t tms_utime; /* tempo de CPU gasto em código do processo */
clock_t tms_stime; /* tempo de CPU gasto em código do sistema
                    chamado pelo processo */
clock_t tms_cutime; /* tempo de CPU dos filhos (código próprio) */
clock_t tms_cstime; /* tempo de CPU dos filhos (cód. do sistema) */
```

Todos os tempos são medidos em *clock ticks*.
O número de *ticks* por segundo pode ser determinado usando `sysconf()`:

```
ticks_seg = sysconf(_SC_CLK_TCK);
```



FEUP

VER EXEMPLO DE UTILIZAÇÃO NO "MATERIAL DE APOIO"

MIEIC
Faculdade de Engenharia da Universidade do Porto

Consola, Ficheiros e Directórios



Objectivos

No final desta aula, os alunos deverão ser capazes de:

- Explicar o que é um descritor de um ficheiro e quais são as principais estruturas de dados, usadas pelo sistema de ficheiros do UNIX/LINUX
- Identificar os 3 descritores *standard*
- Manipular algumas características da consola (ex: *eco*)
- Manipular ficheiros (criar, abrir, ler, escrever, destruir, ...), usando chamadas ao sistema
- Usar as chamadas *dup()* e *dup2()* e explicar a sua utilização no redireccionamento de entradas/saídas
- Usar as principais chamadas ao sistema relativas à manipulação de directórios (criar, listar os ficheiros/sub-dir.s, obter as propriedades de um ficheiro/sub-dir., ...)
- Explicar o conceito de *hard-link* e *symbolic link* entre ficheiros



Desafios

Escrever programas para:

1.

Ler uma *password* sem ecoar os caracteres escritos pelo utilizador

```
> read_password
```

2.

Copiar um ficheiro para outro ou mostrá-lo o écran, dependendo do nº de argumentos da linha de comandos

```
> copy source // mostra no écran
```

```
> copy source destination // copia p/outro ficheiro
```

3.

Listar os ficheiros regulares e sub-directórios de um directório e ...

```
> listdir dirname
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Ficheiros

As chamadas ao sistema relacionadas com ficheiros permitem manipular ficheiros simples, directórios e ficheiros especiais, incluindo:

- ficheiros em disco
- terminais
- impressoras
- facilidades para intercomunicação entre processos, tais como *pipes* e *sockets*.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Descritores de ficheiros

Para o *kernel* todos os ficheiros abertos são referidos através de descritores.

Quando se cria ou se abre um ficheiro já existente, o *kernel* retorna um descritor ao processo que criou ou abriu o ficheiro. Este descritor é um dos argumentos das chamadas que permitem ler ou escrever no ficheiro.

Um descritor é um número inteiro, não-negativo, geralmente pequeno. Os descritores podem tomar valores entre 0 e `OPENMAX`.

Por convenção, as *shells* de Unix associam os 3 primeiros descritores a ficheiros especiais: 0 - *standard input*; 1 - *standard output*; 2 - *standard error*

Estes descritores estão definidos em `unistd.h` através de constantes : `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO`

Por exemplo, a função `printf()` escreve sempre usando o descritor 1 e a função `scanf()` lê sempre usando o descritor 0.

Quando se fecha um ficheiro, o descritor correspondente é libertado e pode ser reutilizado quando se abre um novo ficheiro.

Um ficheiro pode ser aberto várias vezes e por isso pode ter vários descritores a ele associados.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Descritores de ficheiros

Cada descritor de ficheiro tem um conjunto de propriedades associadas:

- um apontador (cursor) de ficheiro que indica a posição do ficheiro onde será executada a próxima operação de leitura/escrita
 - » colocado a 0 (zero) quando o descritor é criado
 - » avança automaticamente após cada operação de leitura/escrita
- uma *flag* que indica se o descritor deve ser automaticamente fechado se o processo invocar uma das funções `exec()`
- uma *flag* que indica se o que se escreve para o ficheiro deve ser acrescentado no fim do ficheiro

Existem outras propriedades que só se aplicam a ficheiros especiais, como *pipes* e *sockets*:

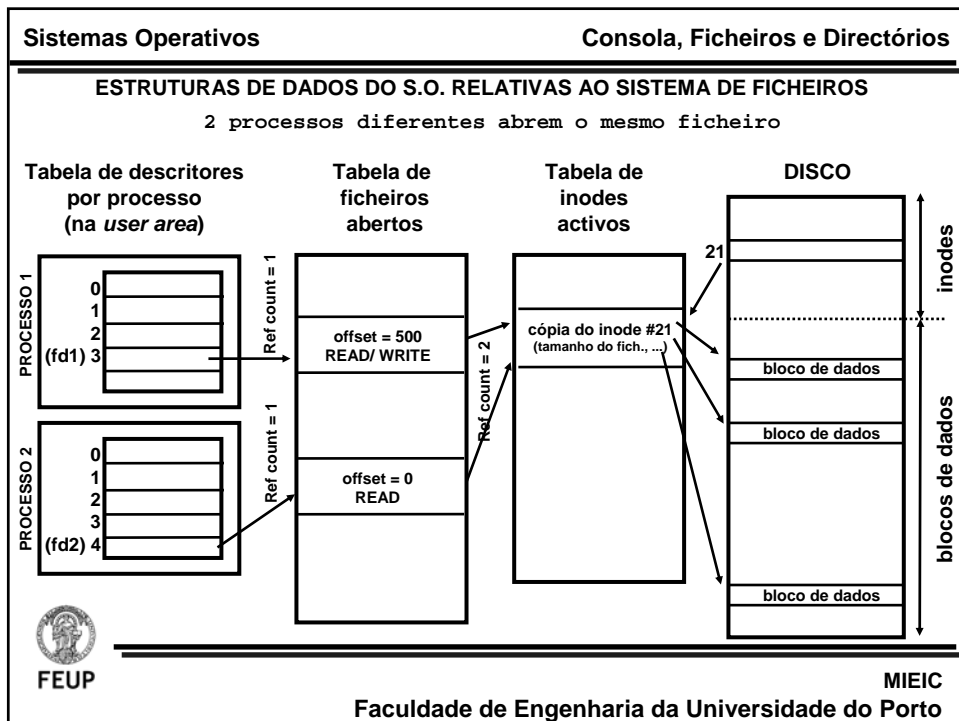
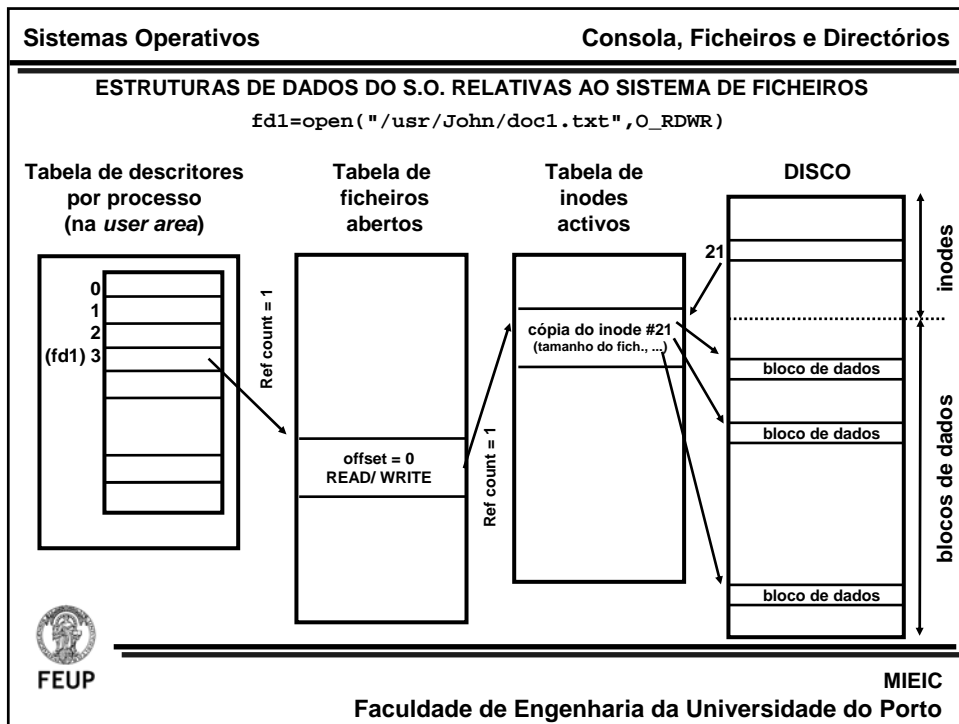
- uma *flag* que indica se um processo deve bloquear se tentar ler de um ficheiro quando ele está vazio.
- um número que indica o identificador de um processo ou de um grupo de processos a quem deve ser enviado o signal `SIGIO` se passarem a existir dados no ficheiro.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto



Consola

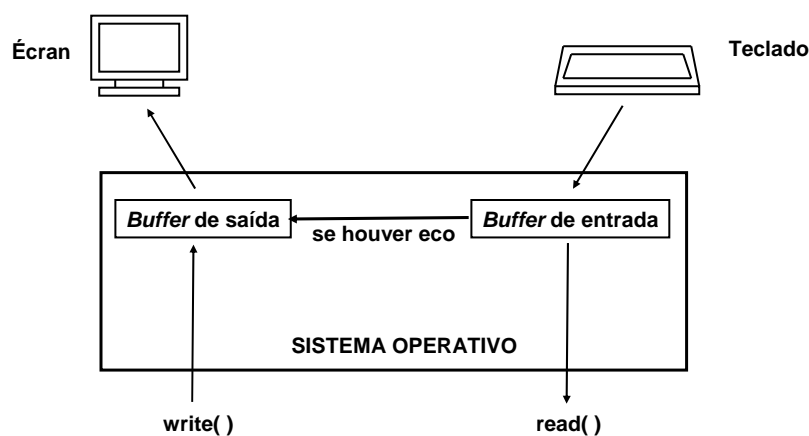
- A consola (teclado + écran) é vista pela generalidade dos S.O.'s como um ou mais ficheiros onde se pode ler ou escrever texto.
- Esses ficheiros são normalmente abertos pela rotina de *C startup*.
- A biblioteca standard de C inclui diversas funções de leitura e escrita directa nesses ficheiros:
 - » `printf()`, `scanf()`, `getchar()`, `putchar()`, ...
- Também é possível aceder àqueles periféricos através de serviços dos S.O.'s
 - » o Unix não define serviços especiais de leitura e escrita na consola
 - » deverão usar-se os serviços genéricos de leitura e escrita em ficheiros



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Consola



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Consola

Modos de funcionamento da consola em Unix:

- **modo canónico (*cooked*)**
 - existe uma série de caracteres especiais de entrada que são processados pela consola e não são transmitidos ao programa que está a ler
 - » ex: ctrl-U, ctrl-H, ctrl-S, ctrl-Q, ...
 - » muitos destes caracteres são alteráveis programaticamente
 - a entrada só é passada ao programa quando se tecla <Return>
- **modo primário (*raw*)**
 - não há qualquer processamento prévio dos caracteres teclados
 - eles são passados um a um ao programa



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Consola

Alteração das características da consola em Unix:

```
#include <termios.h>
```

```
int tcgetattr(int filedes, struct termios *term_ptr);  
int tcsetattr(int filedes, int opt, const struct termios *term_ptr);
```

tcgetattr() - preenche uma estrutura **termios** cujo **endº** é passado em **term_ptr** com as características da componente da consola cujo descritor é **filedes**

tcsetattr() - modifica as características da componente da consola cujo descritor é **filedes**, com os valores previamente colocados em **termios** cujo **endº** é passado em **term_ptr**

opt indica quando a modificação irá ocorrer:

- TCSANOW -> imediatamente
- TCSADRAIN -> após *buffer* de saída se esgotar
- TCSAFLUSH -> após *buffer* de saída se esgotar; além disso, esvazia *buffer* de entrada



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Consola

```
struct termios {
    tcflag_t c_iflag; /* input flags */
    tcflag_t c_oflag; /* output flags */
    tcflag_t c_cflag; /* control flags */
    tcflag_t c_lflag; /* local flags */
    cc_t c_cc[NCCS]; /* control characters */
}
```

`c_iflag`, `c_oflag`, `c_cflag`, `c_lflag`:
campos constituídos por *flags* de 1 ou mais *bits*
que permitem controlar as características da consola

`c_cc[]`:
array onde se definem os caracteres especiais que são processados pela consola
quando esta estiver a funcionar em modo canónico
ex:
`mytermios.c_cc[VERASE]=8; /* 8 = código ASCII de <ctrl-H> */`



O comando da *shell* `stty -a` permite ver os *settings* da estrutura `termios`

FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Consola

Frequentemente, pretende-se apenas activar ou desactivar determinadas *flag* dos campos de *termios* sem alterar as outras

Exemplo:

```
struct termios oldterm, newterm;

...

tcgetattr(STDIN_FILENO, &oldterm);
newterm=oldterm;
newterm.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL | ICANON);
tcsetattr(STDIN_FILENO, TCSAFLUSH, &newterm);
... /* executar operações usando a "nova consola" */
tcsetattr(STDIN_FILENO, TCSANOW, &oldterm);
...
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Consola

- Se acontecer algum erro que leve a que um programa que alterou as características da consola termine imprevistamente, pode acontecer que ela fique num estado que impossibilite a interacção com o utilizador.
- Para tentar repôr o estado "normal" existem várias alternativas:
 - 1) `stty sane` seguido de <return> ou <ctrl-J>
 - 2) `stty -g >save_stty` seguido de <return> ou <ctrl-J>
 ... (correr o programa)
`stty $(cat save_stty)` seguido de <return> ou <ctrl-J>
 - 3) ou ...
 fechar a consola actual e abrir outra



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Criação/abertura de ficheiros

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /*, mode_t mode */);

Retorna: descritor do ficheiro se OK, -1 se erro
```

pathname = nome do ficheiro

oflag = combinação de várias *flags* de abertura

O_RDONLY	- abertura só para leitura	<- só uma destas 3
O_WRONLY	- abertura só para escrita	
O_RDWR	- abertura para leitura e escrita	
O_APPEND	- p/ acrescentar no fim do ficheiro	
O_CREAT	- p/ criar o ficheiro se ele não existir; requer <i>mode</i>	
O_EXCL	- origina erro se o ficheiro existir e O_CREAT estiver activada	
O_TRUNC	- se o ficheiro existir fica com o comprimento 0	
O_SYNC	- só retorna depois de os dados terem sido fisicamente escritos	
...	no ficheiro	



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

mode = permissões associadas ao ficheiro

- só devem ser indicadas quando se cria um novo ficheiro
- pode ser o *OR bit a bit* (|) de várias das seguintes constantes:

S_IRUSR - user read
 S_IWUSR - user write
 S_IXUSR - user execute
 S_IRGRP - group read
 S_IWGRP - group write
 S_IXGRP - group execute
 S_IROTH - others read
 S_IWOTH - others write
 S_IXOTH - others execute

Alternativa:

owner	group	other
rwX	rwX	rwX
111	101	000
7	5	0

mode (em octal) #define MODE 0750

Nota:

- as permissões efectivas podem não ser exactamente as especificadas, consoante o valor da "file creation mask" (especificada c/ a chamada umask)
- o valor por omissão desta máscara é 022 (octal) o que significa anular as permissões de escrita excepto para o owner



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Duplicação de um descritor

Pode ser feita c/ as funções *dup* ou *dup2*.

```
# include <unistd.h>

int dup (int filedes);
int dup2 (int filedes, int filedes2);

Retornam: novo descritor se OK, -1 se houve erro
```

• **dup**

- procura o descritor livre c/ o número mais baixo e põe-no a apontar p/ o mesmo ficheiro que filedes.

• **dup2**

- fecha filedes2 se ele estiver actualmente aberto e põe filedes2 a apontar p/ o mesmo ficheiro que filedes;
- se filedes=filedes2, retorna filedes2 sem fechá-lo.

• **exemplo:**

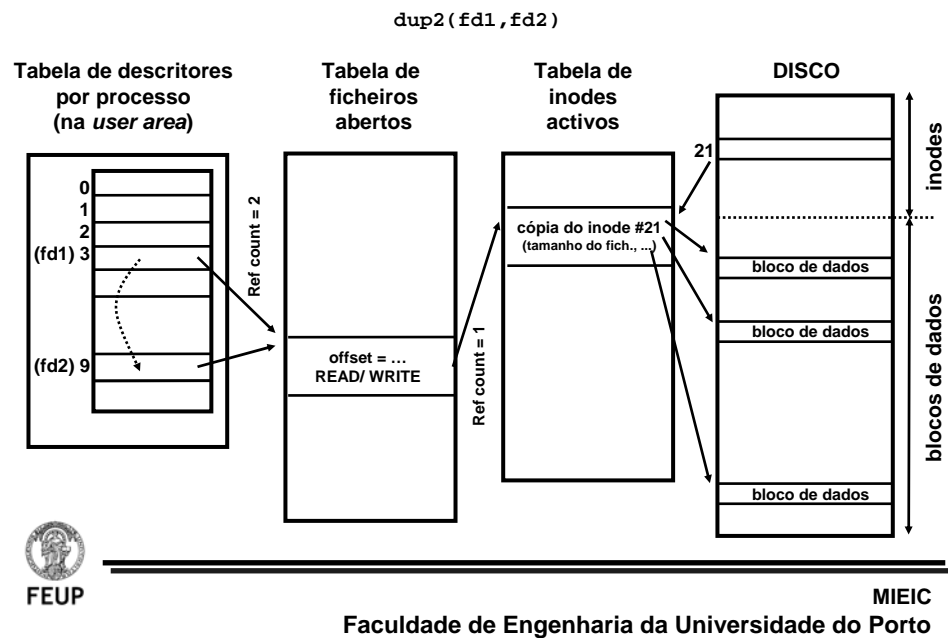
```
dup2 (fd, STDIN_FILENO)
redirecciona a entrada standard (teclado)
para o ficheiro cujo descritor é fd.
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto



Leitura de um ficheiro

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);

Retorna: o nº de bytes lidos, 0 se fim do ficheiro, -1 se erro
```

filedes = descritor do ficheiro
 buff = apontador p/o *buffer* onde serão colocados os valores lidos
 nbytes = nº de *bytes* a ler

Esta função não tem nenhuma das capacidades de formatação de `scanf()`.

Escrita num ficheiro

```
#include <unistd.h>

ssize_t write(int filedes, const void *buff, size_t nbytes);
```

Retorna: o nº de bytes escritos, -1 se erro

`filedes` = descritor do ficheiro
`buff` = apontador p/o *buffer* onde devem ser colocados os valores a escrever
`nbytes` = nº de *bytes* a escrever

Se a *flag* `O_APPEND` tiver sido especificada ao abrir o ficheiro o apontador do ficheiro é posto a apontar para o fim do ficheiro antes de ser efectuada a operação de escrita.

Esta função não tem nenhuma das capacidades de formatação de `printf()`.



Deslocamento do apontador do ficheiro

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

off_t lseek(int filedes, off_t offset, int whence);
```

Retorna: novo valor do apontador se OK, -1 se erro

`filedes` = descritor do ficheiro
`offset` = deslocamento (pode ser positivo ou negativo)
`whence` = a interpretação dada ao `offset` depende do valor deste argumento:

- `SEEK_SET` - o `offset` é contado a partir do início do ficheiro
- `SEEK_CUR` - o `offset` é contado a partir da posição actual do apontador
- `SEEK_END` - o `offset` é contado a partir do fim do ficheiro

Para determinar a posição actual do apontador do ficheiro fazer

```
curr_pos = lseek(fd,0,SEEK_CUR)
```



Fecho de um ficheiro

```
#include <unistd.h>

int close(int filedес);

Retorna: 0 se OK, -1 se erro
```

`filedes` = descritor do ficheiro

Fechar um descritor que já tinha sido fechado resulta num erro.

Quando um processo termina, todos os ficheiros abertos são automaticamente fechados pelo *kernel*.

Se `filedes` for o último descritor associado a um ficheiro aberto o *kernel* liberta os recursos associados a esse ficheiro quando se invoca `close()`.



Apagamento de um ficheiro

```
#include <unistd.h>

int unlink(const char *pathname);

Retorna: 0 se OK, -1 se erro
```

`pathname` = nome do ficheiro

Para se apagar um ficheiro é preciso ter permissão de escrita e execução no directório onde o ficheiro se encontra.

O ficheiro só será, de facto, apagado

- quando for fechado, caso esteja aberto por ocasião da chamada *unlink*
- quando a contagem do nº de *links* do ficheiro atingir o valor 0.



Outras chamadas

umask – modifica a máscara (parâmetro *mode* da chamada *open*)
de criação de ficheiros e directórios

stat, fstat, lstat

- retornam uma *struct* com diversas informações acerca de um ficheiro
 - » tipo, permissões, tamanho, nº de *links*, hora da última modificação, ...)
- existe um conjunto de macros (*S_ISREG()*, *S_ISDIR()*, ...) que permitem determinar qual o tipo de ficheiro, a partir de um campo dessa *struct*

mkdir - cria um novo directório

rmdir - apaga um directório

opendir / closedir - abre / fecha um directório

readdir - lê a entrada seguinte do directório e avança automaticam.

rewinddir - faz com que a próxima leitura seja a da 1ª entrada

getcwd - obtém o nome directório corrente

chdir - muda o directório corrente



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Criação e Terminação de Processos



Objectivos

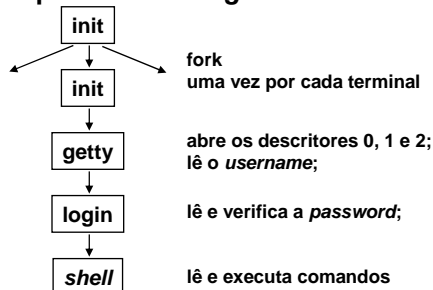
No final desta aula, os alunos deverão ser capazes de:

- Explicar a criação de um novo processo em Unix, usando a função *fork()*
- Usar as funções *fork()* e *exec()* para o lançamento em execução de novos programas
- Usar mecanismos de sincronização básicos entre processos
 - processos que esperam que outros terminem
- Descrever alguns dos estados por que passa um processo durante a sua execução e compreender o significado desses estados



Criação de novos processos

- A forma de um processo existente (processo-pai) criar um novo processo (processo-filho) é invocar a função *fork()*. A única exceção são processos especiais criados pelo *kernel*.
- O processo *init* (*PID=1*) é um processo especial, criado pelo *kernel*.
- Este processo é responsável por criar outros processos de sistema e por desencadear o processo de *login* dos utilizadores.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *fork*

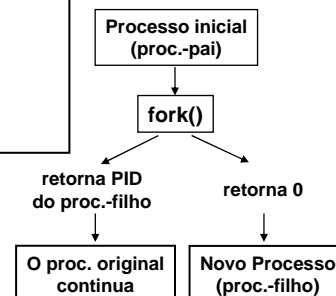
- A função *fork()* é invocada 1 vez mas retorna 2 vezes !
- Após a chamada *fork()* pai e filho executam o mesmo (!...) código.

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

```
pid_t fork(void);
```

Retorna:

```
0 - p/o processo-filho
PID do filho - p/o processo-pai
-1 - se houve erro
```

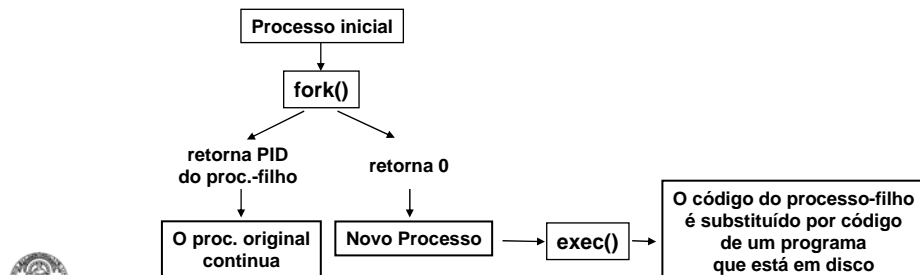


FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Criação de novos processos

- Em geral, pretender-se-á que pai e filho executem diferentes sequências de instruções.
- Isso consegue-se usando instruções condicionais, uma vez que o valor de retorno de *fork()* é diferente para pai e filho.
- Frequentemente o processo-filho substitui o seu código por um novo código invocando uma das funções *exec()* (v. adiante).

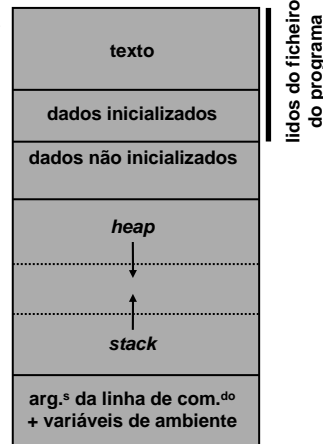


FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *fork*

- O filho é uma cópia do pai ficando com uma cópia do segmento de dados, *heap* e *stack*.
- Em alguns sistemas a cópia só é feita se um dos processos tentar modificar um destes segmentos.
- O segmento de texto é muitas vezes partilhado por ambos.
- Em geral, não se sabe quem começa a executar primeiro (o pai ou o filho).
- Se for necessária sincronização, é preciso usar mecanismos adequados (v. adiante).

end.os
baixosend.os
altos

PROGRAMA EM C
NA MEMÓRIA



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *getpid* e *getppid*

- Um processo pode obter a sua *PID* e a *PID* do seu pai usando, respectivamente, as funções seguintes:

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

pid_t getpid(void);    /* Obter a PID do próprio processo */

pid_t getppid(void);   /* Obter a PID do processo-pai */
```

Notas:

Estas funções são sempre bem sucedidas.
A *PID* do processo-pai do processo 1 é 1.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

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

int    glob = 6;        /* external variable in initialized data */

int main(void)
{
    int    var;          /* automatic variable on the stack */
    pid_t  pid;

    var = 88;

    printf("before fork\n");

    if ( (pid = fork()) < 0) fprintf(stderr, "fork error\n");
    else if (pid == 0) { /* child */
        glob++;          /* modify variables */
        var++; }
    else
        sleep(2);        /* parent ; try to guarantee that child ends first*/

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Resultado possível:

```
before fork
pid = 430, glob = 7, var = 89  as var.s do filho foram modificadas
pid = 429, glob = 6, var = 88  as do pai permanecem inalteradas
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *fork*

Algumas propriedades do pai que são herdadas pelo filho:

- ficheiros abertos
- *ID's* (*real user, real group, effective user, effective group*)
- *process group ID*
- terminal de controlo
- directório de trabalho corrente
- directório raiz
- limites dos recursos
- ...

Algumas diferenças entre pai e filho:

- o valor retornado por *fork* ()
- a *ID* do processo
- as *ID's* do processo-pai de cada um deles
- os alarmes pendentes são anulados para o filho
- o conjunto de sinais pendentes para o filho fica vazio
- ...



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *fork*

fork() pode falhar quando

- o nº total de processos no sistema for demasiado elevado (constante *MAXPID* em *sys/param.h*);
- o nº total de processos do utilizador for demasiado elevado (constante *CHILD_MAX* em *limits.h*).

Utilizações de *fork*()

- Quando um processo quer duplicar-se de tal modo que pai e filho executam diferentes secções de código ao mesmo tempo.
 - » Ex: servidor de rede
 - O pai espera por um pedido de serviço de um cliente.
 - Q.do o pedido chega ele faz *fork* e deixa o filho tratar do pedido.
 - O pai volta a ficar à espera do próximo pedido.
- Quando um processo quer executar um programa diferente
 - » Ex: *shells*
 - O filho faz *exec* depois de retornar do *fork*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Terminação de um processo

Modos de terminação de um processo:

- Normal

- » Executa return na função main.
- » Invoca a função exit() - biblioteca do C
 - Os exit handlers, definidos c/ chamadas atexit, são executados.
 - As I/O streams standard são fechadas.
- » Invoca a função _exit - chamada ao sistema

- Anormal

- » Invoca abort.
- » Recebe certos sinais gerados por
 - próprio processo
 - outro processo
 - kernel



As funções exit e _exit

```
# include <stdlib.h>

void exit (int status);

# include <unistd.h>

void _exit (int status);
```



As funções *exit* e *_exit*

Independentemente do modo como um processo termina (normal/anormal) será eventualmente executado certo código do *kernel*, código este que

- fecha os descritores abertos pelo processo
- liberta a memória que o processo usava
- ...

Terminação normal

- O argumento das funções *exit* (o *exit status*) indica ao proc.-pai como é que o proc.-filho terminou (*termination status*).

Terminação anormal

- O *termination status* do processo é gerado pelo *kernel*.

O processo-pai pode obter o valor do *termination status* através das funções *wait* ou *waitpid*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Processos órfãos

- Se o pai terminar antes do filho, o filho é automaticamente adoptado por *init* (proc. c/ *PID*=1)
- Fica assim garantido que qualquer processo tem um pai.
- Quando um processo termina (neste caso o pai) o *kernel* percorre todos os processos activos para ver se algum deles é filho do processo que terminou.

Se houver algum nestas condições, a *PID* do pai desse processo passa a ser 1.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Processos *zombie*

- Um processo que termina não pode deixar o sistema até que o seu pai aceite o seu código de retorno, através da execução de uma chamada *wait / waitpid*.
- ***Zombie*** - um processo que terminou mas cujo pai ainda não executou um dos *wait's*.
(Em geral, na saída do comando *ps* o estado destes processos aparece como Z)
- ***Excepção***:
quando um processo que foi adoptado por *init* terminar não se torna *zombie*, porque *init* executa um dos *wait's* para obter o seu *termination status*.



Processos *zombie*

- A informação acerca do filho não pode desaparecer completamente.
- O *kernel* mantém essa informação
(PID do processo, *termination status*, tempo de *CPU* usado, ...) de modo a que ela esteja disponível quando o pai executar um dos *wait's*.
- O resto da memória usada pelo filho é libertada.
- Os ficheiros são fechados.



As funções *wait* e *waitpid*

- Um pai pode esperar que um dos seus filhos termine e, então, aceitar o seu *termination status*, executando uma destas funções.
- Quando um processo termina (normalmente ou anormalmente) o *kernel* notifica o seu pai enviando-lhe um sinal (SIGCHLD).
- O pai pode
 - Ignorar o sinal
 - » Se o processo indicar que quer ignorar o sinal os filhos não ficarão *zombies*.
 - Dispor de um *signal handler*
 - » Em geral, o *handler* poderá executar um dos *wait's* para obter a *PID* do filho e o seu *termination status*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *wait* e *waitpid*

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);

Retornam:
    PID do processo - se OK
                   -1 - se houve erro
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *wait* e *waitpid*

Um processo que invoque *wait* ou *waitpid* pode

- **bloquear**
se nenhum dos seus filhos ainda não tiver terminado;
- **retornar imediatamente c/ o *termination status* de um filho**
se um filho tiver terminado e estiver à espera de retornar o seu *termination status* (filho *zombie*).
- **retornar imediatamente com um erro**
se não tiver filhos.

Diferenças entre *wait* e *waitpid*:

- *wait* pode bloquear o processo que o invoca até que um filho qualquer termine
- *waitpid* tem uma opção que impede o bloqueio (útil quando se quer apenas obter o *termination status* do filho)
- *waitpid* não espera que o 1º filho termine, tem opções para indicar o processo pelo qual se quer esperar



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *wait* e *waitpid*

O argumento *statloc*

- $\neq \text{NULL}$ - o *termination status* do processo que terminou é guardado na posição indicada por *statloc* ;
- $= \text{NULL}$ - o *termination status* é ignorado .

O *status* retornado por *wait* / *waitpid*
tem certos *bits* que indicam:

- se a terminação foi normal
- o número de um sinal, se a terminação foi anormal
- se foi gerada uma *core file*

O *status* pode ser examinado (os *bits* podem ser testados)
usando macros, definidas em `<sys/wait.h>` .

Os nomes destas macros começam por *WIF*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *waitpid*

O argumento *pid* de *waitpid*:

- *pid* == -1 - espera por um filho qualquer (\Leftrightarrow *wait*)
- *pid* > 0 - espera pelo filho com a PID indicada
- *pid* == 0 - espera por um qualquer filho do mesmo *process group*
- *pid* < -1 - espera por um qualquer filho cuja *process group ID* seja igual a *valor_absoluto(pid)*

waitpid retorna um erro (valor de retorno = -1) se

- o processo especificado não existir ;
- o processo especificado não for filho do processo que a invocou ;
- o grupo de processos não existir .



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *waitpid*

O argumento *options* de *waitpid*:

- 0 (zero) ou
- OR, *bit a bit* (operador |) das constantes
 - » **WNOHANG**
waitpid não bloqueia se o filho especificado por *pid* não estiver imediatamente disponível.
Neste caso o valor de retorno=0.
 - » **WUNTRACED**
se a implementação suportar *job control*
o *status* de qualquer filho especificado por PID que tenha terminado e cujo *status* ainda não tenha sido reportado desde que ele parou é retornado.

(*job control* - permite iniciar múltiplos *jobs*=grupos de processos a partir de um único terminal e controlar quais os *jobs* que podem aceder ao terminal e quais os *jobs* que são executados em *background*)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

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

int main (void)
{
    int pid, status, childpid;

    printf ("I'm the parent proc. w/PID %d\n", getpid());
    pid = fork();
    if (pid != 0)    //PARENT
    { printf ("I'm the parent proc. w/PID %d and PPID %d\n", getpid(), getppid());
      childpid = wait(&status);    /* wait for the child to terminate */
      printf("A child w/PID %d terminated w/EXIT CODE %d\n",childpid,
            WEXITSTATUS(status) );
    }
    else    //CHILD
    { printf("I'm the child proc. w/ PID %d and PPID %d\n",getpid(),getppid());
      exit(31);    /*exit with a silly number*/
    }
    printf("PID %d terminated\n",getpid()); exit(0);
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Macros p/ testar o *termination status*

WIFEXITED(status)

- **==True**, se o filho terminou normalmente.
- Neste caso, WEXITSTATUS(status) permite obter o exit status do filho (8 bits menos significativos de `_exit / exit`).

WIFSIGNALED(status)

- **==True**, se o filho terminou anormalmente, porque recebeu um sinal que não tratou.
- Neste caso, WTERMSIG(status) permite obter o nº do sinal (não há maneira portátil de obter o nome do sinal em vez do número)
WCOREDUMP(status) == True, se foi gerada uma *core file*.

WIFSTOPPED(status)

- **==True**, se o filho estiver actualmente parado (*stopped*).
O filho pode ser parado através de um sinal
 - » SIGSTOP, enviado por outro processo
 - » SIGTSTP, enviado a partir de um terminal (CTRL-Z)
- Neste caso, WSTOPSIG(status) permite obter nº do sinal que provocou o *stop*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

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

void pr_exit(int status);

int main(void)
{
    pid_t  pid;
    int    status;

    if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
    else if (pid == 0) exit(7); /* child */

    if (wait(&status) != pid) fprintf(stderr,"wait error\n");
    pr_exit(status); /* wait for child and print its status */
    //
    if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
    else if (pid == 0) abort(); /* child generates SIGABRT */

    if (wait(&status) != pid) fprintf(stderr,"wait error\n");
    pr_exit(status); /* wait for child and print its status */
    //
    if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
    else if (pid == 0) status /= 0; /* child - divide by 0 generates SIGFPE */

    if (wait(&status) != pid) fprintf(stderr,"wait error\n");
    pr_exit(status); /* wait for child and print its status */

    exit(0);
}
```

(continua)

Exemplo (cont.)

```
void pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n", WTERMSIG(status),
            #ifdef WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "";
            #else
                "";
            #endif
        );
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```



As funções exec

Existem 6 funções que designaremos genericamente por **exec**

fork - criar novos processos

exec - iniciar novos programas (quando um processo invoca **exec**, o processo é completamente substituído por um novo programa)

```
# include <unistd.h>

int execl (const char *pathname, const char *arg0, ... /* (char *)0 */);
int execl (const char *pathname, char *const argv[]);
int execl (const char *pathname, const char *arg0, ... /* (char *)0,
char *const envp[] */);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */);
int execvp(const char *filename, char *const argv[]);
```

Retorno:

não há - se houve sucesso
-1 - se houve erro



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções exec

Diferenças entre as 6 funções: estão relacionadas com as letras **l**, **v** e **p** acrescentadas a **exec**.

Lista de argumentos

- **l** - Lista, passados um a um separadamente, terminados por um apontador nulo.
- **v** - Vector, passados num vector.

Passagem das *strings* de ambiente (*environment*)

- **e** - Passa-se um apontador para um *array* de apontadores para as *strings*.
- **env** - Usar a variável *environ* se for necessário aceder às variáveis de ambiente no novo programa.

Path

- **p** - O argumento é o nome do ficheiro executável.
 - Se o *path* não for especificado, o ficheiro é procurado nos directórios especificados pela variável de ambiente *PATH*.
 - Se o ficheiro não for um executável (em código máquina) assume-se que pode ser um *shell script* e tenta-se invocar */bin/sh* com o nome do ficheiro como entrada *p/* a *shell*.
- **path** - O nome do ficheiro executável deve incluir o *path*.



FEUP

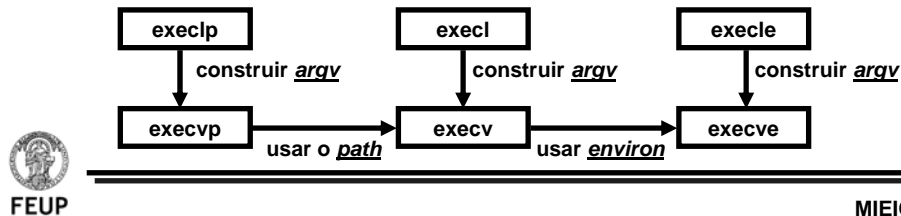
MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções exec

```
# include <unistd.h>

int execl (const char *pathname, const char *arg0, ... /* (char *)0 */);
int execv (const char *pathname, char *const argv[]);
int execlp(const char *pathname, const char *arg0, ... /* (char *)0,
char *const envp[] */);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */);
int execvp(const char *filename, char *const argv[]);

Retorno:
    não há - se houve sucesso
    -1 - se houve erro
```



MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplos

```
#include <unistd.h>
-----

execl("/bin/ls", "/bin/ls", "-l", NULL);

char *env_init[] = {"USER=unknown", "PATH=/tmp", NULL};
...
execl("./prog", "./prog", "arg1", "arg2", NULL, env_init);

execlp("prog", "prog", "arg1", NULL); /* o executável é procurado no PATH */

...

int main (int argc, char *argv[])
{
    if fork() == 0
    {
        execvp(argv[1], &argv[1]);
        fprintf(stderr, "Can't execute\n", argv[1]);
    }
}
```

Programa que executa outro, que lhe é passado como argumento, em *background*.
Se este programa se chamar *back*, o comando *back cc prog1.c* tenta executar *cc prog1.c* em *background*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *exec*

- Limite ao tamanho total da (lista de argumentos + lista de ambiente)
 - especificado por ARG_MAX (em <limits.h>)
- Propriedades que o novo programa herda do processo que o invocou:
 - *PID* e *PID* do pai
 - *real user ID*, *real group ID*
 - *process group ID*
 - *session ID*
 - terminal de controlo
 - directório corrente
 - sinais pendentes
 - limites dos recursos
 - ...
- O tratamento dados aos ficheiros abertos depende da *close-on-exec flag* (FD_CLOEXEC, actualizada pela função *fnctl*)
 - se estiver activada o descritor é fechado
 - se não estiver activada o descritor é mantido aberto (situação por omissão)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *system*

- Usada para executar um comando do interior de um programa.
 - Ex: `system ("date > file")`
- Não é uma interface para o sistema operativo mas para uma *shell*.
- É implementada recorrendo a *fork*, *exec* e *waitpid*.

```
# include <stdlib.h>

int system(const char *cmdstring);

Retorna: (v. a seguir)
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *system*

Retorno de *system*:

- Se *cmdstring* = null pointer
 - » retorna valor $\neq 0$ só se houver um processador de comandos disponível (útil p/ saber se esta função é suportada num dado S.O.; em UNIX é sempre suportada)
- Senão retorna
 - » -1
 - se *fork* falhou ou
 - *waitpid* retornou um erro \neq EINTR (indica que a chamada de sistema foi interrompida)
 - » 127
 - se *exec* falhou
 - » *termination status* da *shell*, no formato especificado por *waitpid*, quando as chamadas *fork*, *exec* e *waitpid* forem bem sucedidas.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Sinais



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Objectivos

No final desta aula, os estudantes deverão ser capazes de:

- Explicar o conceito de sinal, na *API* de Unix/Linux
- Nomear alguns sinais e conhecer as suas origens
- Descrever as diferentes formas que um processo tem de lidar com sinais e o tratamento que o sistema operativo dá aos sinais
- Aplicar as *APIs* (Unix System V e POSIX) relativas a sinais para:
 - enviar um sinal
 - instalar um *handler* de um sinal
 - manipular a máscara de sinais (POSIX)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Sinais

Um sinal é:

- uma notificação, por *software*, de um acontecimento
- uma forma, muito limitada, de comunicação entre processos

Possíveis origens de um sinal:

- **Teclado**
 - » certas teclas/combinções de teclas
ex.: DEL ou ctrl-C, ctrl-Z, ctrl-\ (v. adiante)
- **Hardware**
 - » divisão por 0
 - » referência inválida à memória
- **Função de sistema *kill***
 - » permite que um processo envie um sinal a outro processo ou grupo de processos
- **Comando *kill***
 - » permite enviar um sinal a um processo ou conjunto de processos a partir de *shell*
- **Software**
 - » certos acontecimentos gerados por *software* dão origem a sinais
ex.: quando um alarme, activado pelo processo, expirar



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Sinais

- Os sinais podem ser gerados:
 - sincronamente
 - » associados a uma certa acção executada pelo próprio processo
(ex: divisão por zero ou acesso a memória inválida)
 - assincronamente
 - » gerados por eventos exteriores ao processo que recebe o sinal
- Os processos podem informar o *kernel* do que deve fazer quando ocorrer um determinado sinal.
- O número de sinais depende da versão de Unix.
- Todos têm um nome simbólico que começa por SIG .
- Estão listados no ficheiro de inclusão /usr/include/signal.h .

Respostas possíveis a um sinal:

- Ignorar
- Tratar (*catch*)
- Executar a acção por omissão



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto


Sistemas Operativos		Sinais	
<h2>Exemplos de sinais</h2>			
Nome	Descrição	Origem	Acção por omissão
SIGINT	Interrupção de um processo	teclado (^C)	terminar
SIGQUIT	Saída de um processo	teclado (^)	terminar
SIGTSTP	Suspender processo (<i>job control</i>)	teclado (^Z)	suspender
SIGCONT	Continuar processo suspenso (depois de SIGTSTP)	<i>shell</i> (comandos: fg, bg)	continuar
SIGKILL	Terminação (<i>non catchable</i>)	sistema operativo	terminar
SIGSTOP	Parar a execução (<i>non catchable</i>)	sistema operativo	suspender
SIGTERM	Terminação	<i>default do comando kill</i>	terminar
SIGABRT	Terminação anormal	abort()	terminar
SIGALRM	Alarme	alarm()	terminar
SIGSEGV	Referência a memória inválida	<i>hardware</i>	terminar
SIGFPE	Excepção aritmética	<i>hardware</i>	terminar
SIGILL	Instrução ilegal	<i>hardware</i>	terminar
SIGUSR1	Definido pelo utilizador		terminar
SIGUSR2	Definido pelo utilizador		terminar



MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos		Sinais	
<h2>Respostas a um sinal</h2>			
<p><u>Ignorar o sinal</u></p> <ul style="list-style-type: none"> A maior parte dos sinais podem ser ignorados. SIGKILL e SIGSTOP nunca podem ser ignorados. 			
<p><u>Tratar o sinal</u></p> <ul style="list-style-type: none"> Indicar uma função a executar (<i>signal handler</i>) quando o sinal ocorrer. <ul style="list-style-type: none"> Exemplo: <p>Quando um processo termina ou pára, o sinal SIGCHLD é enviado ao pai. Por omissão este sinal é ignorado, mas o pai pode tratar este sinal, invocando, por exemplo, uma das funções <i>wait</i> para obter a PID e o <i>termination status</i> do filho.</p> 			
<p><u>Acção por omissão</u></p> <ul style="list-style-type: none"> Todos os sinais têm uma accção por omissão (v. adiante). Possíveis acções do <i>default handler</i> <ul style="list-style-type: none"> terminar o processo e gerar uma <i>core file</i> terminar o processo sem gerar uma <i>core file</i> ignorar o sinal suspender o processo continuar o processo 			



MIEIC

Faculdade de Engenharia da Universidade do Porto

Tratamento de sinais

A função *signal*

- A chamada de sistema *signal* permite associar uma rotina de tratamento (*signal handler*) a um determinado sinal.
 - ex.: `signal (SIGINT, inthandler);`
 - ex.: `signal (SIGINT, SIG_IGN);`
- Esta função retorna o endereço do *signal handler* anteriormente associado ao sinal
 - ex.: `oldhandler = signal (SIGINT, newhandler);`
- Limitação de *signal* :
 - não é possível determinar a acção associada actualmente a um sinal sem alterar essa acção
(é possível com a função *sigaction*)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *signal*

Protótipo:

```
#include <signal.h>

void (*signal(int signo, void (*func) (int))) (int);
```

Declaração complicada !
Para simplificar a interpretação fazemos:

```
typedef void sigfunc (int);

sigfunc *signal(int signo, sigfunc *func);
```

Signal é uma função que tem como

- argumentos
 - » um inteiro (o número de um sinal)
 - » um apontador p/ uma função do tipo *sigfunc* (o novo *signal handler*)
- valor de retorno
 - » um apontador p/uma função do tipo *sigfunc* (o *signal handler* anterior) ou SIG_ERR se aconteceu um erro (v. adiante)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *signal*

Outras constantes declaradas em `<signal.h>`:

- **SIG_ERR**
 - » usada para testar se *signal* retornou erro
 - `if (signal(SIGUSR1, usrhandler) == SIG_ERR) ...;`
- **SIG_DFL**
 - » usada como 2º argumento de *signal*
 - » indica que deve ser usado o *handler* por omissão para o sinal especificado como 1º argumento
- **SIG_IGN**
 - » usada como 2º argumento de *signal*
 - » indica que o sinal especificado como 1º argumento deve ser ignorado

```
#define SIG_ERR (void (*)(int)) -1
#define SIG_DFL (void (*)(int)) 0
#define SIG_IGN (void (*)(int)) 1
```

Cast de -1 / 0 / 1 para um apontador para uma função que retorna *void*.
Os valores -1, 0 e 1 poderiam ser outros,
mas não podem ser endereços de funções declaráveis.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

Signal handler que trata dois sinais definidos pelo utilizador e que escreve o nome do sinal recebido:

```
#include ...

void sig_usr(int); /* one handler for both signals */

int main(void)
{ if (signal(SIGUSR1, sig_usr) == SIG_ERR)
  { printf("Can't catch SIGUSR1"); exit(1); }
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
  { printf("Can't catch SIGUSR2"); exit(1); }
  for( ; ; ) pause();
}

void sig_usr(int signo) /* argument is signal number */
{ if (signo == SIGUSR1) printf("Received SIGUSR1\n");
  else if (signo == SIGUSR2) printf("Received SIGUSR2\n");
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo (cont.)

Resultado de execução do programa:

```
$ a.out & -----> iniciar o processo em background
[1] 4720 -----> a job-control shell escreve o nº do job e a PID do proc.
$ kill -USR1 4720 -----> enviar-lhe SIGUSR1
Received SIGUSR1 -----> escrito pelo signal handler
$ kill -USR2 4720 -----> enviar-lhe SIGUSR2
Received SIGUSR2 -----> escrito pelo signal handler
$ kill 4720 -----> enviar SIGTERM ao processo
[1] Terminated a.out -----> o processo foi terminado,
                                     dado que não trata o sinal e
                                     a acção por omissão é a terminação
$
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Tratamento de SIGCHLD

- Quando um processo termina, o *kernel* envia o sinal SIGCHLD ao processo-pai
- O processo-pai pode
 - instalar um *handler* para SIGCHLD e executar `wait()` / `waitpid()` no *handler*
 - ter anunciado que pretende ignorar SIGCHLD; neste caso
 - » os filhos não ficam no estado *zombie*
 - » se o processo-pai chamar `wait()`, esta chamada só retornará (-1) quando todos os filhos terminarem



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Tratamento dos sinais após *fork* / *exec*

Após *fork*

- O tratamento dos sinais é herdado pelo processo-filho.
- O filho pode alterar o tratamento.

Após *exec*

- O estado de todos os sinais será o tratamento por omissão ou ignorar.
- Em geral será o tratamento por omissão.
Só será ignorar se o processo que invocou *exec* estiver a ignorar o sinal.
- Ou seja
- Todos os sinais que estiverem a ser tratados passam a ter o tratamento por omissão.
O tratamento de todos os outros sinais mantém-se inalterado.

Por que será ? R: ao fazer *exec* as rotinas de tratamento “perdem-se” pois o código já não é o mesmo.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Permissão de envio de sinais

Um processo precisa de permissão para enviar sinais a outro.

Só o *superuser* pode enviar sinais a qualquer processo.

Um processo pode enviar um sinal a outro se
a *user ID* real ou efectiva do processo for igual
à *user ID* real ou efectiva do processo a quem o sinal é enviado.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

User ID e Group ID

Quando um processo executa, tem 4 valores associados a permissões:

- *real user ID*, *effective user ID*, *real group ID* e *effective group ID*
- apenas as *effective ID*'s afectam as permissões de acesso, as *real ID*'s só são usadas para contabilidade
- em geral, as permissões de acesso de um processo dependem de quem o executa, não de quem é o dono do executável ...
- ... mas há situações em que isto é indesejável
 - ex: num jogo em que os melhores resultados são guardados num ficheiro, o processo do jogo deve ter acesso ao ficheiro de resultados, mas o jogador não ...
- Para que isso seja possível existem 2 permissões especiais (*set-user-id* e *set-group-id*).

Quando um executável com *set-user-id* é executado

a *effective user ID* do processo passa a ser a do executável.

Idem para a *effective group ID*.

- ex: se o executável do jogo tiver a permissão *set-user-id* activada terá acesso ao ficheiro de pontuações; este terá permissões de escrita para o *s/dono*, impedindo o acesso de outros utilizadores

- API's: *setuid*, *seteuid*, *setgid*, *setegid*



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Alguns sinais do terminal

CTRL-C

- envia o sinal de terminação SIGINT a todos os processos do *foreground process group*

CTRL-Z

- envia o sinal de suspensão (SIGSTP) a todos os processos do *foreground process group*
 - » os *jobs* podem ser continuados com um dos comandos (da C/Korn shell) *fg* ou *bg*:
 - *fg [%job]* - continua o *job* especificado, em *foreground* ;
 - *bg [%job]* - continua o *job* especificado, em *background* ;
 - se não for especificado o *job*, assume o último *job* referenciado
- os processos em *background* não são afectados

CTRL-\

- envia o sinal de terminação SIGQUIT a todos os processos do *foreground process group*
- além de terminar os processos gera uma *core file*



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

As funções *kill* e *raise*

kill

- envia um sinal a um processo ou a um grupo de processos
- ao contrário do que o nome parece indicar, não tem necessariamente como consequência o fim do(s) processo(s)

raise

- envia um sinal ao processo que a invocar

Protótipos:

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

int kill (pid_t pid, int signo);
int raise (int signo);

Retorno: 0 se OK; -1 se ocorreu erro
```



FEUP

pid > 0 - o sinal é enviado ao processo cuja ID é *pid*
 outras valores de *pid* - v. Stevens

MIEIC

Faculdade de Engenharia da Universidade do Porto

O comando *kill*

`kill [-signalID] {pid}+`

- envia o sinal com código *signalID* à lista de processos enumerados
- *signalID* pode ser o número ou o nome de um sinal
- por omissão, é enviado o sinal SIGTERM que causa a terminação do(s) processo(s) que o receber(em)
- só o dono do processo ou o *superuser* podem enviar o sinal
- os processos podem proteger-se dos sinais excepto de SIGKILL (código=9) e SIGSTOP (código=17)
- exemplo:

```
$ kill -USR1 4720
```

`kill -l`

- permite obter a lista dos nomes dos sinais
- exemplo:

```
$ kill -l
HUP INT QUIT ILL TRAP IOT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM URG
STOP TSTP CONT CHLD TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH USR1 USR2
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

As funções *alarm* e *pause*

Protótipos:

```
#include <unistd.h>

unsigned int alarm(unsigned int count);
Retorno: 0 se OK; -1 se ocorreu erro

int pause(void);
Retorno: -1 com errno igual a EINTR
```

alarm (*ualarm*, argumento em microsegundos)

- indica ao *kernel* para enviar um sinal de alarme (SIGALRM) ao processo que a invocou, *count* segundos após a invocação
- se já tiver sido criado um alarme anteriormente, ele é substituído pelo novo
- se *count* = 0, algum alarme, eventualmente pendente, é cancelado
- retorna o número de segundos que faltam até que o sinal seja enviado

pause

- suspende o processo que a invocar, até que ele receba um sinal
- a única situação em que a função retorna é quando é executado um *signal handler* e este retorna



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *abort* e *sleep*

Protótipos:

```
#include <stdlib.h>

void abort(void);
Retorno: não tem

#include <unistd.h>

unsigned int sleep(unsigned int count);
Retorno: 0 ou o número de segundos que faltavam
```

abort (ANSI C; = raise(SIGABRT))

- causa sempre a terminação anormal do programa
- é enviado o sinal SIGABRT ao processo que a invocar
- pode, no entanto, ser executado um *signal handler* para tratar este sinal para executar algumas tarefas antes de o processo terminar

sleep (*usleep*, argumento em microsegundos)

- suspende o processo que a invocar, até que
 - se passem *count* segundos (retorna 0) ou
 - um sinal seja recebido pelo processo e o *signal handler* retorne (retorna o nº de segundos que faltavam)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

Estabelecimento de um alarme e respectivo *handler*

```
#include ...

int alarmflag = 0;
void alarmhandler(int signo);

int main(void)
{
    signal(SIGALRM, alarmhandler);
    alarm(5);
    printf("Pausing ...\n");
    while (!alarmflag) pause(); /* wait for alarm signal */
    printf("Ending ...\n");
    exit(0);
}

void alarmhandler(int signo)
{
    printf("Alarm received ...\n");
    alarmflag = 1;
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

Protecção de um programa contra Control-C (Control-C gera o sinal SIGINT)

```
#include ...

int main(void)
{
    void (*oldhandler)(int);

    printf("I can be Ctrl-C'ed\n");
    sleep(3);
    oldhandler = signal(SIGINT, SIG_IGN);
    printf("I'm protected from Ctrl-C now \n");
    sleep(3);
    signal(SIGINT, oldhandler);
    printf("I'm vulnerable again!\n");
    sleep(3);
    printf("Bye.\n");
    exit(0);
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

O que faz este programa ?

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

void childhandler(int signo);
int delay;

int main(int argc, char *argv[])
{
    pid_t pid;

    signal(SIGCHLD, childhandler);
    pid = fork();
    if (pid == 0)
        execvp(argv[2], &argv[2]);
    else
    {
        sscanf(argv[1], "%d", &delay);
        sleep(delay);
        printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
        kill(pid, SIGKILL);
    }
}
```

```
void childhandler(int signo)
{
    int status;
    pid_t pid;

    pid = wait(&status);
    printf("Child %d terminated within %d seconds. \n", pid, delay);
    exit(0);
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

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

void childhandler(int signo);
int delay;

int main(int argc, char *argv[])
{
    pid_t pid;

    signal(SIGCHLD, childhandler);
    pid = fork();
    if (pid == 0) /* child */
        execvp(argv[2], &argv[2]);
    else /* parent */
    {
        sscanf(argv[1], "%d", &delay); /* read delay from command line */
        sleep(delay);
        printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
        kill(pid, SIGKILL);
    }
}
```

Programa (limit) que lança outro programa (prog) e espera um certo tempo (τ) até que este termine. Caso isso não aconteça, termina-o de modo forçado.

Exemplo de linha de comando:
limit τ prog arg1 arg2 ... argn

```
void childhandler(int signo) /* Executed if child dies before parent */
{
    int status;
    pid_t pid;

    pid = wait(&status);
    printf("Child %d terminated within %d seconds.\n", pid, delay);
    exit(0);
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Funções Posix p/sinais

A norma Posix estabelece uma forma alternativa de instalação de *handlers*, a função `sigaction`, e funções de manipulação de uma máscara de sinais que pode ser utilizada para bloquear a entrega de sinais a um processo

`sigaction`

- especifica a acção a executar quando for recebido um sinal

`sigprocmask`

- usada para examinar ou alterar a máscara de sinais de um processo

`sigpending`

- útil para testar se um ou mais sinais estão pendentes e especificar o método de tratamento desses sinais, antes de se chamar `sigprocmask` para desbloqueá-los

`sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember`,

`sigset`, `sighold`, `sigrelse`, `sigignore`, `sigpause`

- criar e manipular a máscara de sinais



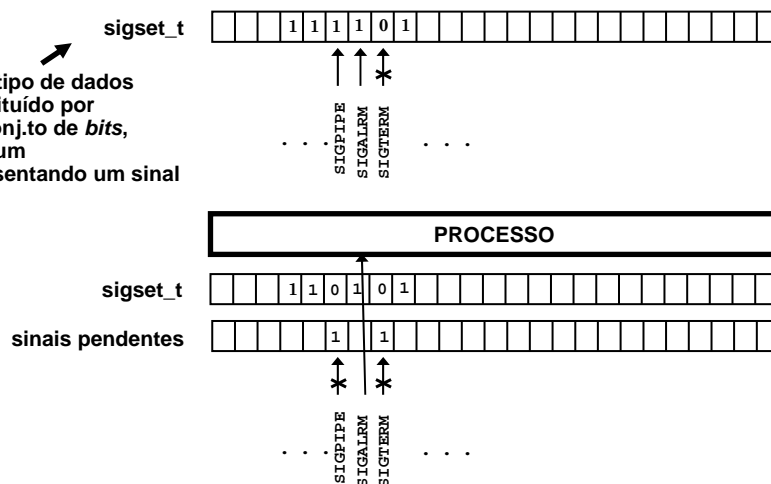
FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Máscara de sinais

é um tipo de dados constituído por um conj.to de *bits*, cada um representando um sinal



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Manipulação da máscara de sinais

```
#include <signal.h>

int sigprocmask(int cmd, const sigset_t *new_mask,
               sigset_t *old_mask)

Retorno: 0 se OK; -1 se ocorreu erro
```

- Alterar e/ou obter a máscara de sinais de um processo

cmd =

- SIG_SETMASK - substituir a máscara actual por new_mask
- SIG_BLOCK - acrescentar os sinais especificados em new_mask à máscara actual
- SIG_UNBLOCK - remover os sinais especificados em new_mask da máscara actual

new_mask =

- se NULL a máscara actual não é alterada; usado q.do se quer apenas obter a máscara actual

old_mask =

- se NULL a máscara actual não é retornada



FEUP

sigset_t é um tipo de dados constituído por um conj.to de *bits*,
cada um representando um sinal

MIEIC

Faculdade de Engenharia da Universidade do Porto

Manipulação da máscara de sinais

```
#include <signal.h>

int sigemptyset(sigset_t *sigmask);
int sigfillset(sigset_t *sigmask);
int sigaddset(sigset_t *sigmask, int sig_num);
int sigdelset(const sigset_t *sigmask, int sig_num);

Retorno: 0 se OK; -1 se ocorreu erro

int sigismember(sigset_t *sigmask, int sig_num);

Retorno: 1 se flag activada ou 0 se não; -1 se ocorreu erro
```

- Alterar / consultar a máscara de sinais de um processo

sigemptyset() - limpar todas as *flags* da máscara
 sigfillset() - activar todas as *flags* da máscara
 sigaddset() - activar a *flag* do sinal sig_num na máscara
 sigdelset() - limpar a *flag* do sinal sig_num na máscara
 sigismember() - testar se a *flag* indicada por sig_num está ou não activada

Q.do usado c/
 sigprocmask(
 SIG_SETMASK, sigmask,...)
 => todos os sinais passam



FEUP

NOTA: Não é possível bloquear os sinais que não podem ser ignorados

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo

Activar a *flag* de SIGINT, mantendo todas as outras *flags*

```
#include ...

int main(void)
{
    sigset_t sigmask;

    ...

    if (sigprocmask(SIG_SETMASK, NULL, &sigmask) == -1) /* obter másc. actual */
    {
        perror("sigprocmask"); exit(1);
    }
    else
    {
        sigaddset(&sigmask, SIGINT);
        if (sigprocmask(SIG_BLOCK, &sigmask, NULL))
            perror("sigprocmask"); exit(1);
        ...
    }
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sinais pendentes

```
#include <signal.h>

int sigpending(sigset_t *sigpset);

Retorno: 0 se OK; -1 se ocorreu erro
```

- Retorna o conjunto de sinais que estão pendentes, por estarem bloqueados; permite especificar o tratamento a dar-lhe(s), antes de invocar `sigprocmask()` para desbloqueá-lo(s)

EXEMPLO:

```
sigset_t set;
...
if (sigpending(&set) == -1)
{
    perror("sigpending"); exit(1);
}
else
if (sigismember(&set, SIGINT))
    printf("SIGINT is pending\n");
else
    printf("SIGINT is not pending\n");
...
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Instalação de um *handler*

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *action,
              struct sigaction *oldaction);

Retorno: 0 se OK; -1 se ocorreu erro
```

- Permite examinar e/ou modificar a acção associada a um sinal

signum = nº do sinal cuja acção se quer examinar ou modificar
 action =
 • se ≠ NULL estamos a modificar
 oldaction =
 • se ≠ NULL o sistema retorna a acção anteriormente associada ao sinal

Esta função usa a estrutura

```
struct sigaction {
    void (*sa_handler) (int); /* end.º do handler ou
                               SIG_IGN ou SIG_DFL */
    sigset_t sa_mask;         /* sinais a acrescentar à máscara */
    int sa_flags;             /* modificam a acção do sinal
                               v. Stevens */
};
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

A função *sigaction*

Esta função substitui a função `signal()` das primeiras versões de Unix.

Os sinais especificados em `sa_mask` são acrescentados à máscara antes do *handler* ser invocado.

Se e quando o *handler* retornar a máscara é reposta no estado anterior. Desta forma é possível bloquear certos sinais durante a execução do *handler*.

O sinal recebido é acrescentado automaticamente à máscara, garantindo, deste modo, que outras ocorrências do sinal serão bloqueadas até o processamento da actual ocorrência ter terminado.

Em geral, se um sinal ocorrer várias vezes enquanto está bloqueado, só uma dessas ocorrências será registada pelo sistema.

As `sa_flags` permitem especificar opções para o tratamento de alguns sinais:

ex: - se o sinal for SIGCHLD,
 especificar que este sinal não deve ser gerado
 quando o processo-filho for *stopped* (*job control*)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo

Instalação de um *handler* para o sinal SIGINT

```
#include ...

void sigint_handler(int sig) {
    printf("AUUU! - fui atingido pelo sinal %d\n",sig);
}

int main(void)
{
    struct sigaction action;

    action.sa_handler = sigint_handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGINT,&action,NULL);

    while(1) {
        printf("Ola' !\n"); sleep(5);
    }
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A função *sigsuspend*

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);

Retorno: -1, com errno=EINTR
```

- Substitui a máscara de sinais do processo pela máscara especificada em *sigmask* e suspende a execução do processo, retomando a execução após a execução de um *handler* de um sinal
- Se o sinal recebido terminar o programa, esta função nunca retorna.
- Se o sinal não terminar o programa, retorna -1, com *errno*=EINTR e a máscara de sinais do processo é reposta com o valor que tinha antes da invocação de *sigsuspend()*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

Esperando por um sinal específico (ex: SIGINT) ...

... usando `pause()`

```
#include ...

int flag=0;

void sig_handler(int sig)
{
    if (sig==SIGINT)
    {
        ... ; flag=1;
    }
}

int main(void)
{
    ...
    while (flag == 0) pause();
    ...
}
```

... usando `sigsuspend()`

```
#include ...

...

int main(void)
{
    sigset_t sigmask;
    ...
    sigaction(SIGINT,...);
    ...
    sigfillset(&sigmask);
    sigdelset(&sigmask,SIGINT);
    sigsuspend(&sigmask);
    ...
}
```



FEUP

O que acontece se o sinal chegar entre o teste de `flag` e `pause()` ?

MIEIC

Faculdade de Engenharia da Universidade do Porto

Notas finais

A utilização de sinais pode ser complexa.

É preciso algum cuidado ao escrever os *handlers* porque eles podem ser chamados assincronamente (um *handler* pode ser chamado em qualquer ponto de um programa, de forma imprevisível)

Sinais que chegam em instantes próximos

- Se 2 sinais chegarem durante um curto intervalo de tempo, pode acontecer que durante a execução de um *handler* de um sinal seja chamado um *handler* de outro sinal, diferente do primeiro (ver adiante).
- Se vários sinais do mesmo tipo forem entregues a um processo antes que o *handler* tenha oportunidade de correr, o *handler* pode ser invocado apenas uma vez, como se só um sinal tivesse sido recebido. Esta situação pode acontecer quando o sinal está bloqueado ou quando o sistema está a executar outros processos enquanto os sinais são entregues.

» Isto significa, por ex., que não se pode usar um *handler* para contar o número de sinais recebidos.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

O que acontece se chegar um sinal enquanto um *handler* está a correr ?

- Quando o *handler* de um dado sinal é invocado, esse sinal é, normalmente, bloqueado até que o *handler* retorne. Isto significa que se 2 sinais do mesmo tipo chegarem em instantes muito próximos, o segundo ficará retido até que o *handler* retorne (o *handler* pode desbloquear explicitamente o sinal usando `sigprocmask()`).
- Um *handler* pode ser interrompido pela chegada de outro tipo de sinal. Quando se usa a chamada `sigaction` para especificar o *handler*, é possível evitar que isto aconteça, indicando que sinais devem ser bloqueados enquanto o *handler* estiver a correr.
- **Nota:** em algumas versões antigas de Unix, quando o *handler* era estabelecido usando a função `signal()`, acontecia que a acção associada ao sinal era automaticamente estabelecida como `SIG_DFL`, quando o sinal era tratado, pelo que o *handler* devia reinstalar-se de cada vez que executasse (!). Nesta situação, se chegassem 2 sinais do mesmo tipo em instantes de tempo muito próximos, podia acontecer que o 2º sinal a chegar recebesse o tratamento por omissão (devido ao facto de o *handler* ainda não ter conseguido reinstalar-se), o que podia levar à terminação do processo.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do PortoChamadas ao sistema interrompidas por sinais (v. Stevens, p. 275)

- É preciso ter em conta que algumas chamadas ao sistema podem ser interrompidas em consequência de ter sido recebido um sinal, enquanto elas estavam a ser executadas.
- Estas chamadas são conhecidas por "slow calls"
ex:
 - » operações de leitura/escrita num *pipe*, dispositivo terminal ou de rede, mas não num disco
 - » abertura de um ficheiro (ex: terminal) que pode bloquear até que ocorra uma dada condição
 - » `pause()` e `wait()` e certas operações `ioctl()`
 - » algumas funções de intercomunicação entre processos (v. cap.s seguintes)
- Estas chamadas podem retornar um valor indicativo de erro (em geral, -1) e atribuir a `errno` o valor `EINTR` ou serem re-executadas, automaticamente, pelo sistema operativo.
- Ter em atenção o que dizem os manuais de cada S.O. acerca destas chamadas



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

Teste de erro em "chamadas lentas" (*slow calls*)

```
nread = read(fd_sock,buf,BUFSIZE);
if (nread<0)
    if (errno==EINTR)
        printf("The read() was interrupted. You can try to read again.\n");
    else
        printf("Error in read(), Don't know what to do about it.\n");
```

... ou, melhor...

```
(while (nread = read(fd_sock, buf, size), nread== -1 && errno==EINTR);
if (nread== -1)
    perror("read()");
```

comma operator

um par de expressões separadas por vírgula
é avaliada da esquerda para a direita e
o valor da expressão à esquerda é descartado;
o tipo e valor do resultado são os da expressão da direita
ex: $f(a, (t=3, t+2), b) \equiv f(a, 5, b)$



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pipes e FIFOs



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Objectivos

No final desta aula, os estudantes deverão ser capazes de:

- Explicar a utilidade de *Pipes* e *FIFOs*
- Explicar as diferenças entre uns e outros
- Utilizar *Pipes* e *FIFOs* para comunicação entre dois ou mais processos
- Identificar alguns dos problemas que podem surgir na utilização destes mecanismos de comunicação e tomar providências para evitá-los



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Comunicação entre Processos

Pipes e FIFOs

Pipes são:

- um mecanismo de comunicação que permite que dois ou mais processos a correr no mesmo computador enviem dados uns aos outros.

Tipos de pipes:

- *pipes* sem nome (*unnamed pipes* ou apenas *pipes*)
 - » São *half-duplex* ou unidireccionais. Os dados só podem fluir num sentido.
 - » Só podem ser usados entre processos que tenham um antecessor comum.
- *pipes* com nome (*named pipes* ou *FIFOs*)
 - » São *half-duplex* ou unidireccionais.
 - » Podem ser usados por processos não relacionados entre si.
 - » Têm um nome que os identifica, existente no sistema de ficheiros.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pipes

- Um *pipe* pode ser visto como um canal ligando 2 processos, permitindo um fluxo de informação unidireccional.
- Esse canal tem uma certa capacidade de *bufferização* especificada pela constante `PIPE_BUF` (ou outra com nome semelhante, em `<limits.h>`).
- Cada extremidade de um *pipe* tem associado um descritor de ficheiro.
- Um *pipe* é criado usando a chamada de sistema `pipe()` a qual devolve dois descritores, um representando a extremidade de escrita e outro a de leitura.
- Para o programador, os *pipes* têm uma interface idêntica à dos ficheiros. Um processo escreve numa extremidade do *pipe* como para um ficheiro e o outro processo lê na outra extremidade.
- Um *pipe* pode ser utilizado como um ficheiro ou em substituição do periférico de entrada ou de saída de um programa.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pipes

- Protótipo da função *pipe*:

```
# include <unistd.h>

int pipe (int filedes[2]);

Retorna: 0 se OK, -1 se houve erro
```

- A função retorna 2 descritores de ficheiros:
 - `filedes[0]` – está aberto para leitura
 - `filedes[1]` – está aberto para escrita
- As primitivas de leitura e escrita são *read* e *write*:

```
# include <unistd.h>
ssize_t read (int fd, char * buf, int count);
ssize_t write (int fd, char * buf, int count);

Retornam: n°de bytes lidos/escritos, 0 se EOF (só read),
-1 se houve erro
```



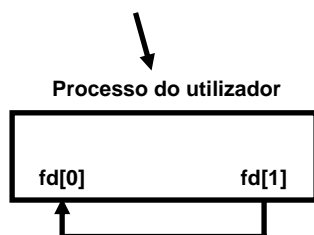
FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

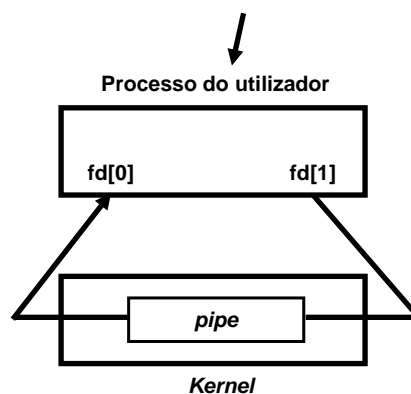
Pipes

Representação simplificada
de um *pipe* num único processo



- Um *pipe* envolvendo um único processo é praticamente inútil.

Convém não esquecer que, de facto,
os dados circulam através do *kernel*



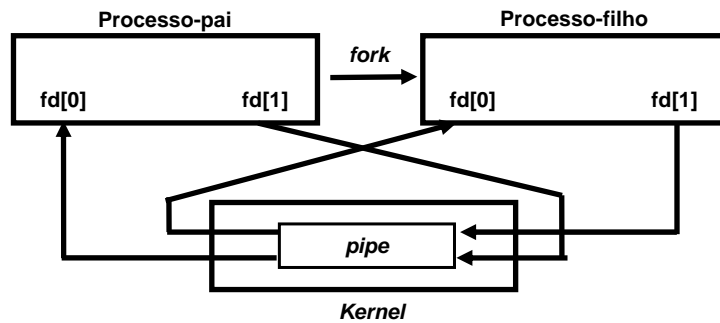
FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Pipes

- Normalmente, o processo que cria o *pipe*, invoca *fork* a seguir, criando assim um canal de comunicação entre pai e filho ou vice-versa.



- Só o processo que cria o *pipe* e os seus descendentes podem usar o *pipe*.

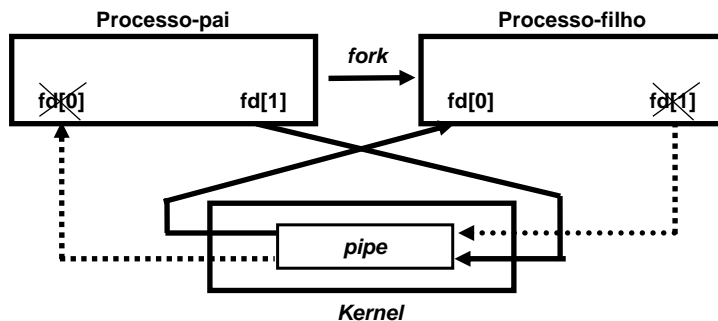


FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pipes

- O que se faz depois da chamada *fork* depende do sentido em que se pretende o fluxo de dados.
- Exemplo: fluxo no sentido do pai p/ o filho
 - o pai fecha a extremidade de leitura - *fd[0]*
 - o filho fecha a extremidade de escrita - *fd[1]*



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pipes

- Sequência típica de operações para a comunicação unidireccional entre o processo-pai e o processo-filho:
 - O processo-pai cria o *pipe*, usando a chamada *pipe()*.
 - O processo-pai invoca *fork()*.
 - O processo-escriptor fecha a sua extremidade de leitura do *pipe* e o processo-leitor fecha a sua extremidade de escrita do *pipe*.
 - Os processos comunicam usando chamadas *write()* e *read()*.
 - » *write* acrescenta dados numa extremidade do *pipe* (extremidade de escrita)
 - » *read* lê dados da outra extremidade do *pipe* (extremidade de leitura)
 - Cada processo fecha o seu descritor activo do *pipe* quando tiver terminado a sua utilização.
- A comunicação bidireccional é possível usando 2 pipes.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

Envio de dados do pai p/ o filho usando um *pipe*:

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

#define MAXLINE 4096
#define READ 0
#define WRITE 1

int main(void)
{
    int n, fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0) {fprintf(stderr, "Pipe error"); exit(1);}
    if ( (pid = fork()) < 0) {fprintf(stderr, "Fork error"); exit(2);}
    else if (pid > 0) { /* parent, writer */
        close(fd[READ]);
        write(fd[WRITE], "Hello world\n", 12);
    } else { /* child, reader */
        close(fd[WRITE]);
        n = read(fd[READ], line, MAXLINE);
        write(STDOUT_FILENO, line, n); // <--Why not printf ...?
    }
    exit(0);
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Pipes

Regras aplicáveis aos processos-leitores:

- Se um processo executar *read* de um *pipe* cuja extremidade de escrita foi fechada, depois de todos os dados terem sido lidos, *read* retorna 0, indicando fim de ficheiro.
 - » NOTA:
 - Frequentemente existe um único leitor e um único escritor de/para um *pipe*.
 - No entanto, é possível duplicar um descritor do *pipe*, usando as funções *dup()* ou *dup2()*, de modo a ter, por exemplo, vários escritores e um único leitor.
 - Neste último caso, o fim de ficheiro só é retornado quando todos os escritores tiverem fechado o terminal de escrita do *pipe*.
- Se um processo executar *read* de um *pipe* vazio cuja extremidade de escrita ainda estiver aberta fica bloqueado até haver dados disponíveis.
- Se um processo tentar ler mais *bytes* do que os disponíveis são lidos os *bytes* disponíveis e a chamada *read* retorna o número de *bytes* lidos.



FEUP

(ver notas finais acerca da activação da *flag* O_NONBLOCK)

MIEIC

Faculdade de Engenharia da Universidade do Porto

Pipes

Regras aplicáveis aos processos-escretores:

- Se um processo executar *write* para um *pipe* cuja extremidade de leitura foi fechada a escrita falha e ao escritor é enviado o sinal SIGPIPE. A acção por omissão deste sinal é terminar o receptor do sinal.
- Se um processo escrever PIPE_BUF *bytes* ou menos é garantido que a escrita é feita atomicamente, isto é, não é interlaçada com escritas de outros processos que escrevam para o mesmo *pipe*.
- Se um processo escrever mais do que PIPE_BUF *bytes* não são dadas garantias de atomicidade da escrita, isto é, os dados dos diversos escritores podem surgir interlaçados.

(ver notas finais acerca da activação da *flag* O_NONBLOCK)

FEUP

MIEIC

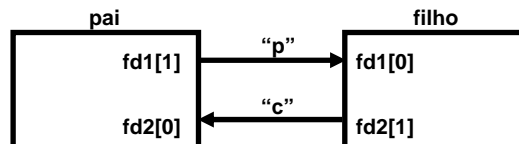
Faculdade de Engenharia da Universidade do Porto

Utilização dos *pipes*

1. Enviar dados de um processo p/ outro (exemplo anterior)

2. Sincronização entre processos

⇒ usar 2 *pipes*



3. Ligar a *standard output* de um processo à *standard input* de outro

⇒ duplicar os descritores de um *pipe* para a *standard input* de um dos processos e para a *standard output* do outro



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Duplicação de um descritor

Pode ser feita c/ as funções *dup* ou *dup2*.

```
# include <unistd.h>

int dup (int filedes);
int dup2 (int filedes, int filedes2);

Retornam: novo descritor se OK, -1 se houve erro
```

- **dup**

- procura o descritor livre c/ o número mais baixo e põe-no a apontar p/ o mesmo ficheiro que *filedes*.

- **dup2**

- fecha *filedes2* se ele estiver actualmente aberto e põe *filedes2* a apontar p/ o mesmo ficheiro que *filedes*;
- se *filedes=filedes2*, retorna *filedes2* sem fechá-lo.

- **exemplo:**

```
dup2(fd, STDIN_FILENO)
redirecciona a entrada standard (teclado)
para o ficheiro cujo descritor é fd.
```



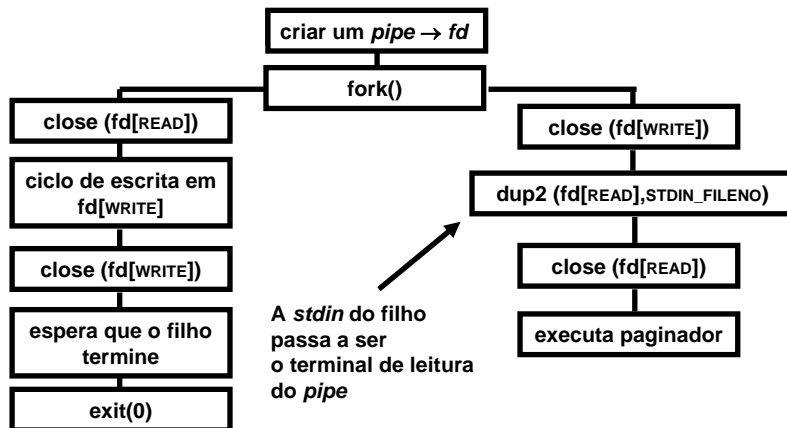
FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo (utilização 3)

Programa que mostra a sua saída, uma página de cada vez, usando o paginador do UNIX (programa 14.2-Stevens)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *popen* e *pclose*

- ***popen***
 - Cria um *pipe* entre o processo que a invocou e um programa a executar; este programa tanto pode receber como fornecer dados ao processo.
 - Faz parte do trabalho do exemplo anterior:
 - criar um *pipe*;
 - executar *fork*;
 - executar um programa invocando uma *subshell* (*sh*) à qual o programa é passado como comando a executar.
 - Vantagem: a *subshell* faz a expansão dos argumentos (por exemplo *.c) o que permite executar com *popen* comandos que seria mais complicado executar com *exec*.
 - Desvantagem: é criado um processo adicional (a *subshell*) para executar o programa.
 - Retorna um apontador para um ficheiro que será o ficheiro de entrada ou de saída do programa, consoante um parâmetro de *popen*.
- ***pclose***
 - Fecha o ficheiro.
 - Espera que o programa termine (mais concretamente, a *shell*).
 - Retorna o *termination status* da *subshell* usada para executar o programa.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

As funções *popen* e *pclose*

```
#include <stdio.h> /* FUNÇÕES DA BIBLIOTECA DE C */

FILE *popen(const char *cmdstring, const char *type);

Retorna: file pointer se OK; NULL se houve erro

int pclose(FILE *fp);

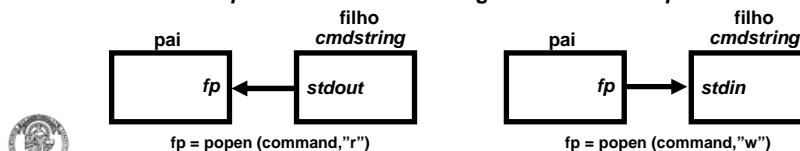
Retorna: termination status de cmdstring se OK; -1 se houve erro
```

cmdstring

- programa a executar

type

- “r” – o file pointer retornado está ligado à *standard output* de *cmdstring*
- “w” – o file pointer retornado está ligado à *standard input* de *cmdstring*



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo

Programa que mostra um ficheiro, página a página, usando o paginador do UNIX.

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

#define MAXLINE 1000
#define PAGER "/usr/ucb/more"

int main(int argc, char *argv[])
{
    char line[MAXLINE];
    FILE *fpin, *fpout;

    if (argc != 2) { printf("usage: a.out filename"); exit(1); }
    if ((fpin = fopen(argv[1], "r")) == NULL) { fprintf(stderr, "can't open %s", argv[1]); exit(1); }
    if ((fpout = popen(PAGER, "w")) == NULL) { fprintf(stderr, "popen error"); exit(1); }
    /* copy filename contents to pager - file=argv[1] */
    while (fgets(line, MAXLINE, fpin) != NULL)
    {
        if (fputs(line, fpout) == EOF) { printf("fputs error to pipe"); exit(1); }
    }
    if (ferror(fpin)) { fprintf(stderr, "fgets error"); exit(1); }
    if (pclose(fpout) == -1) { fprintf(stderr, "pclose error"); exit(1); }
    exit(0);
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Filtros

Filtro

- um programa que lê da *standard input* e escreve p/ a *standard output* normalmente ligado a outros processos, constituindo um *pipeline*

```
#include <ctype.h> /*char handling */
#include <stdio.h>

int main(void)
{
    int c;

    while ((c=getchar()) != EOF)
    {
        if (isupper(c)) c=tolower(c);
        if (putchar(c)==EOF)
        {
            printf("output error"); exit(1);
        }
        if (c=='\n') fflush(stdout);
    }
    exit(0);
}
```

Filtro que converte
maiúsculas em minúsculas
(executável → up_to_low)



FEUP

Programa que usa o filtro (terminar com CTRL-D = *end of input*)

```
#include <stdio.h>

#define MAXLINE 1000

int main(void)
{
    char line[MAXLINE];
    FILE *fpin;

    if ((fpin=popen("up_to_low","r")) == NULL)
    {
        printf("popen error"); exit(1);
    }
    for ( ; ; )
    {
        fputs("prompt > ",stdout); fflush(stdout);
        if (fgets(line,MAXLINE,fpin) == NULL) break;
        if (fputs(line,stdout) == EOF)
        {
            fprintf(stderr,"fputs error"); exit(1);
        }
    }
    if (pclose(fpin)==-1)
    {
        fprintf(stderr,"pclose error"); exit(1);
    }
    putchar('\n');
    exit(0);
}
```

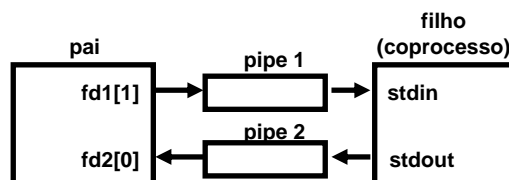
MIEIC

Faculdade de Engenharia da Universidade do Porto

Coprocessos

Coprocesso

- é um filtro especial
cuja *standard input* e *standard output* estão ligadas a um outro processo,
através de pipes



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo (coprocesso)

```
#include <stdio.h>
#include <unistd.h>

#define MAXLINE 100

int main(void)
{
    int    n, int1, int2;
    char   line[MAXLINE];

    while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0)
    { line[n] = 0; /* null terminate */
      if (sscanf(line, "%d%d", &int1, &int2) == 2)
      { sprintf(line, "%d\n", int1 + int2);
        n = strlen(line);
        if (write(STDOUT_FILENO, line, n) != n)
        { fprintf(stderr, "write error"); exit(1); }
      }
      else
      { if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
        { fprintf(stderr, "write error"); exit(1); }
      }
    }
    exit(0);
}
```

Coprocesso

- Lê 2 números da *standard input*
- calcula a sua soma
- e escreve o resultado na *standard output*

O executável deverá chamar-se **somador**



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo (coprocesso - cont.)

Programa que invoca o coprocesso

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

#define MAXLINE 1000
#define READ 0
#define WRITE 1

void sig_pipe(int signo);
void err_sys(char *msg);
void err_msg(char *msg);

int main(void)
{
    int    n, fd1[2], fd2[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");
    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) err_sys("fork error");
    else
    {
        if (pid > 0) /* PARENT */
        {
            close(fd1[READ]); close(fd2[WRITE]);
            while (fgets(line, MAXLINE, stdin) != NULL)
            {
                n = strlen(line);
                if (write(fd1[WRITE], line, n) != n)
                    err_sys("write error to pipe");
                if ((n = read(fd2[READ], line, MAXLINE)) < 0)
                    err_sys("read error from pipe");
                if (n == 0) { err_msg("child closed pipe"); break; }
                line[n] = 0;
                if (fputs(line, stdout) == EOF) err_sys("fputs error");
            }
            if (ferror(stdin)) err_sys("fgets error on stdin");
            exit(0);
        }
    }
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo (coprocesso - cont.)

Programa que invoca o coprocesso (cont.)

```
else      /* CHILD */
{
    close(fd1[WRITE]); close(fd2[READ]);
    if (fd1[READ] != STDIN_FILENO)
    {
        if (dup2(fd1[READ],STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        close(fd1[READ]);
    }
    if (fd2[WRITE] != STDOUT_FILENO)
    {
        if (dup2(fd2[WRITE],STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        close(fd2[WRITE]);
    }
    if (execlp("somador","somador",(char *) 0) < 0)
        err_sys("execlp error");
}
}
```

Ver justificação
para este teste
na pág. 433
do livro de W. Stevens
(Advanced Programming
in the UNIX Environment)

```
void sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}

void err_sys(char *msg)
{
    fprintf(stderr,"%s\n",msg);
    exit(1);
}

void err_msg(char *msg)
{
    printf("%s\n",msg); return;
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

FIFOs / Named Pipes

- *pipes (unnamed)*
 - troca de dados entre processos c/um antecessor comum.
- *FIFOs (named)*
 - troca de dados entre processos não relacionados entre si a correr no mesmo *host* (ver adiante) .
- Um *FIFO* é um tipo de ficheiro.
Tem um nome que existe no sistema de ficheiros.
- Podemos testar se um ficheiro é um *FIFO* c/ a macro *S_ISFIFO*.
- Um *FIFO* pode ser criado usando *mkfifo* ou *mknod*
 - *mknod* - SVR3 (função / utilitário)
 - *mkfifo* - POSIX.1, SVR4 (invoca *mknod*) (função / utilitário)
- Um *FIFO* tem existência até ser explicitamente destruído.
 - *unlink* (função)
 - *rm* (utilitário)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

FIFOS

Função mkfifo

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

int mkfifo(const char *pathname, mode_t mode);

Retorna: 0 se OK, -1 se houve erro
```

pathname

- nome do FIFO a criar

mode

- permissões de acesso (read, write, execute) p/ owner, group e other

	owner	group	other
	rwX	rwX	rwX
	111	101	000
mode →	7	5	0

Nota:

a permissão de acesso final é afectada pelo valor da file creation mask (default = 022)
permissão de escrita só para o owner)
(v. função umask)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

FIFOS

Utilização de um FIFO:

- criar usando mkfifo ou mknod
- abrir usando open ou fopen
 - » include files - sys/types.h, sys/stat.h, fcntl.h
 - » int open (const char *filename, int mode [, int permissions]);
 - mode - OR bit a bit de (O_RDONLY ou O_WRONLY) e O_NONBLOCK
(um FIFO é half-duplex, não deve ser aberto em modo read-write (O_RDWR))
- escrever / ler usando write / read
 - » include files - unistd.h
 - » ssize_t read (int fd, char * buf, int count);
 - » ssize_t write (int fd, char * buf, int count);
- fechar usando close
 - » include files - unistd.h
 - » int close (int fd);
- destruir usando unlink
 - » include files - unistd.h
 - » int unlink (const char *pathname);



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

FIFOS

Regras aplicáveis aos processos que usam *FIFOS*:

Abertura

- Se um processo tentar abrir um *FIFO* em modo *read only* e nenhum processo tiver o *FIFO* actualmente aberto p/ escrita o leitor esperará que um processo abra o *FIFO* p/ escrita a menos que a *flag* `O_NONBLOCK` esteja activada (a activação pode ser feita ao fazer *open* ou com a função *fcntl*), caso em que *open* retornará imediatamente.
- Se um processo tentar abrir um *FIFO* em modo *write only* e nenhum processo tiver o *FIFO* actualmente aberto p/ leitura o escritor esperará que um processo abra o *FIFO* p/ leitura a menos que a *flag* `O_NONBLOCK` esteja activada, caso em que *open* falha imediatamente (retorna -1).



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

FIFOS

Regras aplicáveis aos processos que usam *FIFOS*:

Leitura / Escrita

- Escrita p/ um *FIFO* que nenhum processo tem aberto p/ leitura ⇒ o sinal `SIGPIPE` é enviado ao processo-escritor.
 - » Se este sinal não for tratado conduz à terminação do processo.
 - » Se o sinal for ignorado ou se for tratado e o *handler* retornar, então *write* retorna o erro `EPIPE`
- Após o último escritor ter fechado um *FIFO*, um EOF é gerado em resposta às leituras seguintes, após o *FIFO* ficar vazio.
- Se houver vários processos-escritores, só há garantia de escritas atômicas quando se escreve no máximo `PIPE_BUF` bytes.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

```
/* PROGRAMA reader */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>

int readline(int fd, char *str);

int main(void)
{
    int fd;
    char str[100];

    mkfifo("myfifo", 0660);
    fd=open("myfifo", O_RDONLY);
    while(readline(fd, str)) printf("%s", str);
    close(fd);
}

int readline(int fd, char *str)
{
    int n;

    do
    {
        n = read(fd, str, 1);
    }
    while (n>0 && *str++ != '\0');
    return (n>0);
}
```

```
/* PROGRAMA writer */
#include <stdio.h>
#include <string.h>
#include <sys/file.h>

int main(void)
{
    int fd, messagelen, i;
    char message[100];

    do
    {
        fd=open("myfifo", O_WRONLY);
        if (fd==-1) sleep(1);
    }
    while (fd==-1);

    for (i=1; i<=3; i++)
    {
        sprintf(message, "Hello no. %d from process\n", i, getpid());
        messagelen=strlen(message)+1;
        write(fd, message, messagelen);
        sleep(3);
    }
    close(fd);
}
```

Engenharia da Universidade do Porto

Exemplo (cont.)

Exemplo de execução:

```
/usr/users1/silva> reader & writer & writer & ← Lança 1 reader e 2 writers
[1] 29161 ← processo reader
[2] 29162 ← 1º writer
[3] 29163 ← 2º writer

/usr/users1/silva> Hello no. 1 from process 29162
Hello no. 1 from process 29163
Hello no. 2 from process 29162
Hello no. 2 from process 29163
Hello no. 3 from process 29163
Hello no. 3 from process 29162

[2] Done writer
[3] Done writer
[1] Done reader

/usr/users1/silva> ls -la myfifo ← o fifo não foi destruído
prw-r----- 1 silva 0 Nov 19 14:44 myfifo ← pelos programas
/usr/users1/silva> rm myfifo ← destrói o fifo
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

FIFOS

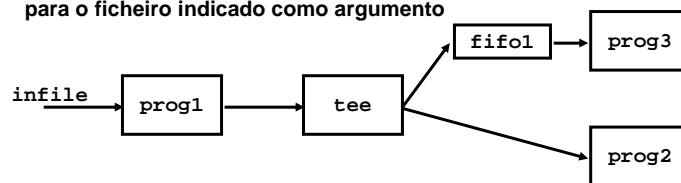
Utilitário *mkfifo*

Exemplo: utilização de *FIFOs* p/ duplicar *output streams* numa sequência de comandos da *shell*

```
$ mkfifo fifo1
$ prog3 < fifo1 &
$ prog1 < infile | tee fifo1 | prog2
```

tee

- copia a sua *standard input* para a sua *standard output* e para o ficheiro indicado como argumento

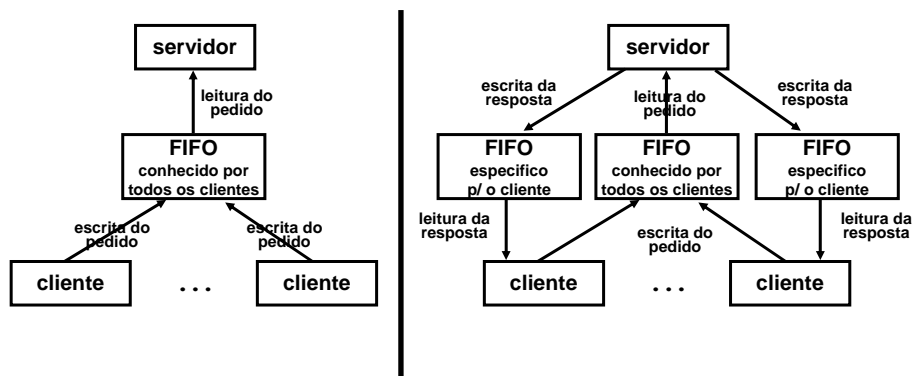


FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

FIFOS

Exemplo: utilização de *FIFOs* p/ comunicação cliente-servidor



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Propriedades adicionais de Pipes e FIFOs

- `mkfifo()` tem implícito o modo `O_CREAT | O_EXCL`, isto é, cria um novo *FIFO* ou retorna o erro `EEXIST` se já existir um *FIFO* com o nome especificado
- Ao abrir um *FIFO*, pode-se activar a *flag* `O_NONBLOCK` :

```
fd=open(FIFO1,O_WRONLY|O_NONBLOCK);
```

- Esta *flag* influencia o comportamento de `open()`, `read()` e `write()`.
- Se um descritor já estiver aberto pode usar-se `fcntl()` para activar a *flag* `O_NONBLOCK`.

» Com *pipes* esta é a única possibilidade de activar esta *flag*, dado que não se usa `open()`.

```
...
int flags;
...
flags = fcntl(fd,F_GETFL,0);
flags = flags | O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
...
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

```
open(FIFO1,O_RDONLY);
```

- `open()` bloqueia até que um processo abra o *FIFO* para escrita.

```
open(FIFO1,O_RDONLY|O_NONBLOCK);
```

- `open()` é bem sucedida e retorna imediatamente mesmo que o *FIFO* ainda não tenha sido aberto para escrita por nenhum processo.

```
open(FIFO1,O_WRONLY);
```

- `open()` bloqueia até que um processo abra o *FIFO* para leitura.

```
open(FIFO1,O_WRONLY|O_NONBLOCK);
```

- `open()` retorna imediatamente; se algum processo tiver o *FIFO* aberto para leitura, retorna um descritor do *FIFO* se não retorna `-1` (erro `ENXIO`) e o *FIFO* não será aberto.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

- `read()` de mais dados do que os disponíveis no *Pipe/FIFO*
 - » retorna os dados disponíveis
- `read()` de um *Pipe/FIFO* vazio, não aberto para escrita
 - » retorna 0 (*end of file*), independentemente de `O_NONBLOCK`
- `read()` de um *Pipe/FIFO* vazio, já aberto para escrita
 - » se `O_NONBLOCK` não estiver activado
 - bloqueia até que sejam escritos dados no *Pipe/FIFO* ou até que o *Pipe/FIFO* deixe de estar aberto para escrita
 - » se `O_NONBLOCK` estiver activado
 - retorna um erro, `EAGAIN`
- `write()` num *Pipe/FIFO*, não aberto para leitura
 - » `SIGPIPE` é enviado ao escritor, independentemente de `O_NONBLOCK`
- `write()` num *Pipe/FIFO*, já aberto para leitura (→ a seguir)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

- `write()` num *Pipe/FIFO*, já aberto para leitura
 - » Se `O_NONBLOCK` não estiver activado
 - Se nº de *bytes* a escrever \leq `PIPE_BUF`
 - Se há espaço no *Pipe/FIFO* para o nº de *bytes* pretendido, todos os *bytes* são escritos.
 - Se não bloqueia até haver espaço no *Pipe/FIFO* para escrever os dados.
 - Se nº de *bytes* a escrever $>$ `PIPE_BUF`
 - escreve parte dos dados, retornando o nº de *bytes* efectivamente escritos (pode ser zero).
 - » Se `O_NONBLOCK` estiver activado
 - o valor de retorno de `write()` depende do nº de *bytes* a escrever e do espaço disponível nesse momento, no *Pipe/FIFO*:
 - Se nº de *bytes* a escrever \leq `PIPE_BUF`
 - Se há espaço no *Pipe/FIFO* para o nº de *bytes* pretendido, todos os *bytes* são escritos.
 - Se não há espaço, `write()` retorna imediatamente com o erro `EAGAIN`.
 - Se nº de *bytes* a escrever $>$ `PIPE_BUF`
 - Se houver espaço no *Pipe/FIFO* para pelo menos 1 *byte* o *kernel* transfere para lá o nº de *bytes* que lá couberem e `write()` retorna o nº de *bytes* escritos.
 - Se o *Pipe/FIFO* estiver cheio, `write()` retorna imediatamente com o erro `EAGAIN`.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

FIFOS e NFS

- Os *FIFOS* são um mecanismo de *IPC* que pode ser usado num único *host*.
- Apesar de terem nomes no sistema de ficheiros, só podem ser usados em sistemas de ficheiros locais, e não em sistemas de ficheiros montados através de *NFS*.
- Alguns sistemas, permitem criar *FIFOS* num sistema de ficheiros montado em *NFS*, no entanto, não permitem a passagem de dados entre 2 sistemas, através desses *FIFOS*.

Neste caso o *FIFO* só terá utilidade como mecanismo de *rendez-vous* entre 2 processos. Um processo num *host* não pode enviar dados a outro processo, noutra *host*, através do *FIFO*, apesar de ambos poderem abrir o *FIFO*, que está acessível a ambos através de *NFS*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Threads



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Objectivos

No final desta aula, os estudantes deverão ser capazes de:

- escrever programas que usem um ou múltiplos *threads* (*multithreaded*)
- passar dados (entrada/saída) entre *threads*
- identificar os problemas que se colocam na manipulação de dados comuns e na passagem de dados entre *threads* e resolver esses problemas
- usar um mecanismo básico de sincronização entre *threads* (`pthread_join()`)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Threads

- A *Pthreads API* está definida na norma ANSI / IEEE POSIX 1003.1 (1995)
- As funções desta *API* (mais de 60) podem ser divididas em 3 grupos:
- **GESTÃO DE THREADS**
 - Permitem criar e terminar *threads*, esperar pela sua terminação, etc.
Incluem funções para ler/alterar os atributos dos *threads*
(de escalonamento e outros, ex: se é possível esperar que um *thread* termine).
- **MUTEXES**
 - Permitem proteger uma secção crítica .
Incluem funções para criar, destruir, trancar (*lock*) e destrancar (*unlock*) *mutexes* e alterar os seus atributos.
- **CONDITION VARIABLES (VARIÁVEIS DE CONDIÇÃO)** (ver cap. sobre sincronização)
 - Permitem bloquear um *thread* até que se verifique uma certa condição e entrar protegido numa secção crítica.
São usadas em conjunto com um *mutex* associado.



As principais funções destes 2 últimos grupos serão analisadas no capítulo sobre sincronização.

MIEIC
Faculdade de Engenharia da Universidade do Porto

Compilação e execução

- Todos os programas que usem chamadas Posix relacionadas com *threads*, devem incluir a seguinte linha de controlo:

```
# include <pthread.h>
```
- Para compilar, por exemplo o programa `prog1.c`, dar o comando:

```
gcc prog1.c -o prog1 -D_REENTRANT -lpthread -Wall
```

 - `-D_REENTRANT` - para incluir a versão reentrante das bibliotecas de sistema
(<http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html>; em alguns compiladores pode não ser necess.)
 - `-lpthread` - para "*linkar*" com a biblioteca Posix de *threads* (`libpthread`)
(`-pthread` ou `-pthreads` em alguns compiladores)
- Valor de retorno das chamadas relacionadas com *threads*:

```
Retorno:  
0 se OK  
ou um valor positivo (Exxx, definido em errno.h) se erro
```



NOTA: há algumas chamadas que não retornam a quem as invoca (ex: `pthread_exit()`)

MIEIC
Faculdade de Engenharia da Universidade do Porto

Criação de *threads*

```
int pthread_create ( pthread_t *tid, const pthread_attr_t *attr,
                    void * (*func)(void *), void *arg );
```

função de início do *thread*

```
void *func (void *arg) {
    /* CÓDIGO DO THREAD */ ... }
```

tid

- apontador para a identificação do *thread*, retornada pela chamada
- a *tid* é usada noutras chamadas da *API* de *threads*

attr

- usado para especificar os atributos do *thread* a criar, ex: política de escalonamento, tamanho da *stack*, ...; ver chamadas *pthread_attr_xxx*
- NULL = usar atributos por defeito; é a situação mais frequente

func

- função que o *thread* executará quando for criado
- esta função só admite 1 argumento, que lhe é passado através do parâmetro *arg*

arg

- apontador para o(s) argumento(s) do *thread*; pode ser NULL
- **NOTA:** para passar vários argumentos é necessário compactá-los numa estrutura de dados



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Terminação de *threads*

Formas de um *thread* terminar:

- O *thread* retorna normalmente (na função inicial é executada a instrução *return* ou atinge-se a "}" final)
- O *thread* invoca *pthread_exit()*
- O *thread* é cancelado por outro *thread*, através de *pthread_cancel()*
- O processo a que o *thread* pertence termina
- O processo a que o *thread* pertence substitui o seu código devido a uma chamada *execxx()*

Notas:

- Se *main()* terminar porque executou *exit()*, *_exit()*, *return* ou atingiu a última instrução os *threads* por si criados também terminarão automaticamente.
- No entanto, se *main()* terminar com a chamada *pthread_exit()* os outros *threads* continuarão em execução; as variáveis globais não serão destruídas e os ficheiros abertos não serão fechados.
- Um *thread* pode esperar que outros *threads* terminem usando a chamada *pthread_join()*.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Terminação de *threads*

```
void pthread_exit (void *status);
```

Não retorna a quem fez a chamada.

status

- valor de retorno, especificando o estado de terminação do *thread*
- NULL, quando não se pretende retornar nada

NOTAS:

- se a função inicial do *thread* terminar com `return ptr`, o valor de `status` será o apontado por `ptr` (ver exemplo adiante)
- o apontador `status` não deve apontar para um objecto que seja local ao *thread* pois esse objecto deixará de existir quando o *thread* terminar
- `pthread_exit()` não fecha os ficheiros que estiverem abertos



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Esperando pela terminação *threads*

```
int pthread_join (pthread_t tid, void **status);
```

- O *thread* que invocar esta função bloqueia até que o *thread* especificado por `tid` termine

tid

- *thread* pelo qual se quer esperar
(= valor obtido ao invocar `pthread_create()`)

status

- apontador para apontador para o valor de retorno do *thread*

- Os *threads* podem ser *joinable* (por defeito) ou *detached*.

É impossível esperar por um *detached thread*.

Quando um *joinable thread* termina a sua *ID* e *status* são mantidos pelo S.O. até que outro *thread* invoque `pthread_join()`.

- **NOTA:** Não há forma de esperar por qualquer um dos *threads* como acontecia no caso dos processos com as chamadas `wait()` e `waitpid(-1, ...)`



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Outras chamadas

Um *detached thread* (*thread* separado) é um *thread* pelo qual não é possível esperar.

Quando termina, todos os recursos que lhe estão associados são libertados.

Usando `pthread_detach()` é possível transformar um *joinable thread* em *detached*.

```
int pthread_detach (pthread_t tid);
```

Esta função é frequentemente invocada pelo *thread* que quer passar de *joinable* a *detached*,

o que pode ser conseguido executando `pthread_detach(pthread_self())`

```
pthread_t pthread_self (void);
```

Retorna: *thread ID* do *thread* que fez a chamada

Para criar um *thread* no estado *detached* ao invocar `pthread_create()` é necessário preencher devidamente o atributo `attr` desta chamada.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - criação e terminação

NOTA:
nos exemplos que se seguem não são feitos testes de erro nas chamadas
para melhorar a legibilidade dos programas

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

int global;

void *thr_func(void *arg);

int main(void)
{
    pthread_t tid;

    printf("Hello from main thread\n");
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_exit(NULL);
}

void *thr_func(void *arg)
{
    sleep(3);
    printf("Hello from auxiliar thread\n");
    return NULL;
}
```

NÃO FAZER ISTO NOS
TRABALHOS PRÁTICOS

NOTA:
desta forma, o *thread* auxiliar
pode continuar a executar
mesmo depois de `main()` terminar



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - criação de múltiplos *threads*

```

...
void * thrfunc(void * arg)
{
    int i;

    fprintf(stderr, "Starting thread %s\n", (char *) arg);
    for (i = 0; i < 10000; i++) write(1, (char *) arg, 1);
    return NULL;
}

int main()
{
    ...
    int retcode;
    pthread_t ta, tb;
    void * retval;

    retcode = pthread_create(&ta, NULL, thrfunc, "A");
    retcode = pthread_create(&tb, NULL, thrfunc, "B");
    ...
    retcode = pthread_join(ta, &retval);
    retcode = pthread_join(tb, &retval);
    ...
    return 0;
}

```

SAÍDA:

```

Starting thread A
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
AAStarting thread B
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
...
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
...
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
...
...
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - passagem de valores usando variáveis globais

```

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

int global;

void *thr_func(void *arg);

int main(void)
{
    pthread_t tid;

    global = 20;
    printf("Main thread: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_exit(NULL);
}

void *thr_func(void *arg)
{
    printf("Aux thread: %d\n", global);
    return NULL;
}

```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - passagem de valores usando variáveis globais

```

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

int global;

void *thr_func(void *arg);

int main(void)
{
    pthread_t tid;

    global = 10;
    printf("Main thread: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_join(tid, NULL);
    printf("Main thread: %d\n", global);
    return 0;
}

void *thr_func(void *arg)
{
    global++;
    printf("Aux thread: %d\n", global);
    return NULL;
}

```

o *thread* principal esperou que o *thread* auxiliar terminasse

o programa pode terminar sem problema



FEUP

Exemplo - passagem / retorno de valores em argumentos

```

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

void *thr_func(void *arg);

int main(void)
{
    pthread_t tid;
    int k = 10;
    void *r;

    pthread_create(&tid, NULL, thr_func, &k);
    pthread_join(tid, &r);
    printf("Main thread: %d\n", *(int *)r);
    free(r);
    return 0;
}

void *thr_func(void *arg)
{
    void *ret;
    int value;

    value = *(int *) arg;
    printf("Aux thread: %d\n", value);
    value++;
    ret = malloc(sizeof(int));
    *(int *)ret = value;
    return ret;
}

```

NOTAR



FEUP

Exemplo - passagem de argumentos

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadnum)
{
    printf("Thread %d: Hello World!\n",
           *(int *)threadnum);
    pthread_exit(NULL);
}

int main()
{
    pthread_t tid[NUM_THREADS];
    int t;
    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        pthread_create(&tid[t], NULL, PrintHello, &t);
    }
    ...
}
```

PASSAGEM INCORRECTA DE ARGUMENTOS:

Porquê ?



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - passagem de argumentos

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadnum)
{
    printf("Thread %d: Hello World!\n",
           *(int *)threadnum);
    pthread_exit(NULL);
}

int main()
{
    pthread_t tid[NUM_THREADS];
    int t;
    for(t=0; t<NUM_THREADS; t++){
        printf("Creating thread %d\n", t);
        pthread_create(&tid[t], NULL, PrintHello, &t);
    }
    ...
}
```

```
...
int thrarg[NUM_THREADS];
for(t=0; t < NUM_THREADS; t++)
{
    thrarg[t] = t;
    printf("Creating thread %d\n", t);
    pthread_create(&threads[t], NULL,
                  PrintHello,
                  &thrarg[t]);
}
...

```

PASSAGEM CORRECTA DE ARGUMENTOS:

PASSAGEM INCORRECTA DE ARGUMENTOS:

o ciclo que cria os *threads* modifica
o conteúdo do endereço passado como argumento
possivelmente antes de o *thread* criado conseguir aceder-lhe



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - passagem de argumentos

```
...
int main(void) {
    int thrarg[NUM_THREADS];

    for(t=0; t < NUM_THREADS; t++)
    {
        thrarg[t] = t;
        printf("Creating thread %d\n", t);
        pthread_create(&threads[t], NULL,
                      PrintHello,
                      &thrarg[t]);
    }
    ...
}
```

**PASSAGEM CORRECTA
DE ARGUMENTOS**

```
...
int main(void) {
    int *thrarg[NUM_THREADS];
    ...

    for(t=0; t < NUM_THREADS; t++)
    {
        thrarg[t] = (int *) malloc(sizeof(int));
        *thrarg[t] = t;
        printf("Creating thread %d\n", t);
        pthread_create(&threads[t], NULL,
                      PrintHello,
                      thrarg[t]);
    }
    ...
}
```

SOLUÇÃO ALTERNATIVA

Quando se justifica que o espaço
p/os argum.s seja reservado no "heap" ?
Haveria alguma alternativa ?



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - criação e terminação

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

int global=0;

void *thr_func(void *arg)
{
    while (global++ < 20) {
        printf("t%d - %d\n", *(int *)arg, global); sleep(1);
    }
    return NULL;
}

int main(void)
{
    pthread_t tid;
    int t1=1, t2=2;

    printf("Hello from main thread\n");
    pthread_create(&tid, NULL, thr_func, (void *)&t1);
    pthread_create(&tid, NULL, thr_func, (void *)&t2);
    pthread_exit(NULL);
}
```

SAÍDA:

```
Hello from main thread
t1 - 1
t2 - 2
t2 - 3
t1 - 4
t1 - 5
t2 - 6
t2 - 7
t1 - 8
t1 - 9
t2 - 10
t2 - 11
t1 - 12
t1 - 13
t2 - 14
t2 - 15
t1 - 16
t1 - 17
t2 - 18
t2 - 19
t1 - 20
```

Haverá aqui algum "perigo" na utilização da variável global?



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo - passagem de argumentos múltiplos

```
...
struct thread_data {
    int  thread_num;
    int  value;
    char *message; };

struct thread_data thr_data_array[NUM_THREADS];

void *PrintHello(void *thread_arg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) thread_arg;
    tasknum = my_data->thread_num;
    value = my_data->value;
    hello_msg = my_data->message;
    ...
}

int main()
{
    ...
    thread_data_array[t].thread_num = t;
    thread_data_array[t].value = x;
    thread_data_array[t].message = messages[t];
    pthread_create(&threads[t], NULL, PrintHello, (void *) &thr_data_array[t]);
    ...
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto**Notas finais**

- • As funções invocadas num *thread* têm de ser thread-safe.
- As funções thread-unsafe podem ser classificadas em 4 classes:
- » Classe 1 - não protegem variáveis partilhadas
 - » Classe 2 - baseiam-se na persistência de estado entre invocações
 - » Classe 3 - retornam um apontador para uma variável estática
 - » Classe 4 - invocam funções *thread-unsafe*
- Uma função diz-se reentrante se e só se não aceder a variáveis partilhadas quando invocada por várias *threads*;
as funções reentrantes são um sub-conjunto das funções *thread-safe*
- A maior parte das chamadas de sistema em Unix são *thread-safe* com poucas exceções
- » ex: `asctime`, `ctime`, `gethostbyaddr`, `gethostbyname`,
`inet_ntoa`, `localtime`, `rand`
- Destas, todas pertencem à Classe 3 (acima) com excepção de `rand` que pertence a Classe 2.
Para estas funções existe normalmente uma função reentrante com o mesmo nome acrescido de `_r` (ex: `ctime_r`).



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Comunicação entre Processos

IPC- Interprocess communication

- designação de um conjunto de mecanismos através do qual dois ou mais processos comunicam entre si

A comunicação entre processos é suportada por todos os sistemas UNIX. Contudo, diferentes sistemas UNIX implementam diferentes métodos de *IPC*.

O UNIX suporta:

- Para processos correndo na mesma máquina
 - » Pipes e FIFOs
 - » Mensagens
 - » Semáforos
 - » Memória partilhada
- Para processos correndo em máquinas diferentes
 - » Sockets
 - » TLI - Transport Layer Interface
 - » XTI - X/Open Transport Interface (baseado em TLI)

Os métodos de *IPC* definidos no standard Posix.1b são mensagens, semáforos e memória partilhada, mas a sua sintaxe é diferente da do System V.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Métodos de *IPC* do UNIX System V

Pipes & FIFOs

- Permitem que processos a correr na mesma máquina troquem dados entre si, através de um *pipeline*.

Mensagens

- Permitem que processos a correr na mesma máquina troquem dados entre si, através de uma fila de mensagens.

Memória partilhada

- Permite que vários processos a correr na mesma máquina partilhem uma região de memória comum.

Semáforos & *Mutexes*


- Fornecem mecanismos para que os processos/*threads* sincronizem as suas acções.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

A seguir ...

- Filas de mensagens
 - Semáforos
 - Memória partilhada
- 
- ~~System V - IPC~~
 - Posix



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Métodos POSIX para IPC

Os métodos *IPC* estão definidos na norma POSIX.1003.1b .

Os métodos POSIX para *IPC* são os mesmos do *System V*:

- filas de mensagens
- semáforos
- memória partilhada

A sintaxe das primitivas POSIX que suportam estes métodos é totalmente diferente das do *System V* .

Para compilar programas que usem estas primitivas é necessário *linkar* com a biblioteca *realtime* (*librt.a*)

```
gcc prog1.c -o prog1 -lrt -Wall
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Filas de mensagens

Primitivas:

```
#include <mqueue.h>
```

```
mqd_t mq_open(char* name, int flags, mode_t mode, struct mq_attr* attrp);
int mq_send(mqd_t mqid, const char* msg, size_t len, unsigned priority);
int mq_receive(mqd_t mqid, char* buf, size_t len, unsigned* prio);
int mq_close(mqd_t mqid);
int mq_notify(mqd_t mqid, const struct mq_sigevent* sigvp);
int mq_getattr(mqd_t mqid, struct mq_attr* attrp);
int mq_setattr(mqd_t mqid, struct mq_attr* attrp, struct mq_attr* oattrp);
```

LINUX:

- Estas primitivas só estão disponíveis a partir da versão 2.6.6-rc1 do *kernel* do Linux, precisando ainda, para funcionamento correcto, de uma *user-space library*
- O comando `uname -r` permite saber qual a versão do *kernel*



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Semáforos

A norma POSIX suporta 2 tipos de semáforos:

- semáforos com nome (*named semaphores*)
 - podem ser partilhados por vários processos
- semáforos sem nome (*unnamed semaphores*)
 - podem ser partilhados por vários processos que têm acesso a memória comum; para isso, o objecto semáforo tem de ser criado em memória partilhada

named semaphores

```
sem_open()
```

unnamed semaphores

```
sem_init()
```

```
sem_wait()
sem_trywait()
sem_post()
sem_getvalue()
```

```
sem_close()
sem_unlink
```

```
sem_destroy()
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Semáforos

Primitivas:

```
#include <semaphore.h>

sem_t* sem_open(char* name, int flags, mode_t mode, unsigned value);

int sem_unlink(char* name);

int sem_init(sem_t* sem, int pshared, unsigned value);

int sem_close(sem_t* sem);

int sem_destroy(sem_t* sem);

int sem_getvalue(sem_t* sem, int* sval);

int sem_wait(sem_t* sem);

int sem_trywait(sem_t* sem);

int sem_post(sem_t* sem);
```

NOTA:

name deve ser da forma /some_name
com "/" no início



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Semáforos

**PRODUTOR-
-CONSUMIDOR**
com *buffer* de
tamanho 1



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define SHARED 0 /* sem. is shared between threads */

void *Producer(void *); /* the two threads */
void *Consumer(void *);

sem_t empty, full; /* the global semaphores */
int data; /* shared buffer */
int numIters;

int main(int argc, char *argv[]) {
    pthread_t pid, cid;
    numIters = atoi(argv[1]);
    sem_init(&empty, SHARED, 1); /* sem empty = 1 */
    sem_init(&full, SHARED, 0); /* sem full = 0 */

    printf("Main started.\n");
    pthread_create(&pid, NULL, producer, NULL);
    pthread_create(&cid, NULL, consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    printf("Main done.\n");
    return 0;
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

```
/* Put items (1, ..., numItems) into the data buffer and sum them */
void *producer(void *arg) {
    int total=0, produced;
    printf("Producer running\n");
    for (produced = 1; produced <= numItems; produced++) {
        sem_wait(&empty);
        data = produced;
        total = total+data;
        sem_post(&full);
    }
    printf("Producer: total produced is %d\n",total);
    return NULL;
}

/* Get values from the data buffer and sum them */
void *consumer(void *arg) {
    int total = 0, consumed;
    printf("Consumer running\n");
    for (consumed = 1; consumed <= numItems; consumed++) {
        sem_wait(&full);
        total = total+data;
        sem_post(&empty);
    }
    printf("Consumer: total consumed is %d\n",total);
    return NULL;
}
```

```
> gcc pc.c -o pc -lpthread -lrt -Wall
> ./pc 20
Main started.
Producer running
Consumer running
Producer: total produced is 210
Consumer: total consumed is 210
Main done.
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Memória partilhada

Primitivas:

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

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

NOTA:

- `shm_open()` não permite especificar o tamanho da região de memória partilhada
- o tamanho tem de ser especificado numa chamada `ftruncate()` subsequente
- para juntar e retirar a região de mem. partilhada do espaço de endereçamento de um processo é necessário usar as chamadas `mmap()` e `munmap()`



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Memória partilhada

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

int      ftruncate(int fd, off_t length);

#include <sys/mman.h>

void*    mmap(void *start, size_t length, int prot , int flags,
             int fd, off_t offset);

int      munmap(void *start, size_t length);
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Exemplo

```
// POSIX shared memory & semaphore - usage example
// writer.c
// Program that writes a digit sequence in shared memory
// and waits for a reader (reader.c) to read it
// The reader must write an '*'
// at the beginning of the shared memory region
// for signaling the writer that the shared memory can be removed
// (another semaphore could have been used instead)
// JAS
// gcc writer.c -lrt -Wall -o writer

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/types.h>

#define SHM_SIZE 10

//names should begin with '/'
char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";

int main()
{
    int shmfd;
    char *shm, *s;
    sem_t *sem;
    int i, n;
    long int sum = 0;
```

Utilizando mem. partilhada
e
semáforo(s)
para comunicação entre
dois processos

ESCRITOR

```
//create the shared memory region
shmfd = shm_open(SHM_NAME,O_CREAT|O_RDWR,0600);
if(shmfd<0)
{ ... }
if (ftruncate(shmfd,SHM_SIZE) < 0)
{ ... }
//attach this region to virtual memory
shm = (char *) mmap(0,SHM_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
if(shm == MAP_FAILED)
{ ... }

//create & initialize semaphore
sem = sem_open(SEM_NAME,O_CREAT,0600,0);
if(sem == SEM_FAILED)
{ ... }
```

CONT →

Faculdade de Engenharia da Universidade do Porto


```
//write into shared memory region
s = shm;
for(i=0; i<SHM_SIZE-1; i++)
{
    n = i % 10; sum = sum + n;
    *s++ = (char) ('0' + n);
}
*s = (char) 0;

printf("sum = %ld\n", sum);

sem_post(sem);
```

```
//this loop could be replaced by semaphore use
//TO DO by students

printf("Busy waiting for 'reader' to read shared
memory ...\n");
while(*shm != '**')
{
    sleep(1);
}

//close and remove shared memory region and semaphore
sem_close(sem);
sem_unlink(SEM_NAME);

if (munmap(shm, SHM_SIZE) < 0)
{ ... }
if (shm_unlink(SHM_NAME) < 0)
{ ... }

exit(0);
}
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Exemplo

```
// POSIX shared memory & semaphore - usage example
// reader.c
// Program that reads a digit sequence
// written in shared memory by a writer (writer.c)
// After reading, this program writes an '**'
// at the beginning of the shared memory region
// signaling to the writer that the region can be removed
// (another semaphore could have been used instead)
// JAS
// gcc reader.c -lrt -Wall -o reader
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

#define SHM_SIZE 10

//names should begin with '/'
char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";
```

```
int main()
{
    int shmfd;
    char *shm, *s, ch;
    sem_t *sem;
    long int sum = 0;
```

```
//open the shared memory region
shmfd = shm_open(SHM_NAME, O_RDWR, 0600);
if (shmfd < 0)
{ ... }

//attach this region to virtual memory
shm = (char *) mmap(0, SHM_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, shmfd, 0);
if (shm == MAP_FAILED)
{ ... }

//open existing semaphore
sem = sem_open(SEM_NAME, 0, 0600, 0);
if (sem == SEM_FAILED)
{ ... }

//wait for writer to stop writing
sem_wait(sem);
```

Utilizando mem. partilhada
e
semáforo(s)
para comunicação entre
dois processos

LEITOR

FEUP

CONT →

Faculdade de Engenharia da Universidade do Porto

```
//read the message
s = shm;
for (s=shm; *s!=0; s++)
{
    ch = *s;
    putchar(ch);
    sum = sum + (ch - '0');
}
printf("sum = %ld\n", sum);

//once done, signal exiting of reader
//could be replaced by semaphore use (TO DO by students)

*shm = '*';

//close semaphore and unmap shared memory region
sem_close(sem);

if (munmap(shm, SHM_SIZE) < 0)
{ ... }

exit(0);
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Sincronização de *threads*

A sincronização de *threads* pode ser feita recorrendo a:

- **semáforos** (ver cap. anterior)
- **mutexes**
 - podem ser vistos como semáforos inicializados em 1 servindo fundamentalmente p/ garantir a exclusão mútua de secções críticas
- **condition variables** (variáveis de condição)
 - permitem que um *thread* aceda a uma secção crítica apenas quando se verificar uma determinada condição sem necessidade de ficar a ocupar o processador para testar essa condição; enquanto ela não se verificar o *thread* fica bloqueado

Estes 2 últimos mecanismos de sincronização foram introduzidos pela norma POSIX que definiu a *API* de utilização de *threads*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Mutexes

Sequência típica de utilização de um *mutex*:

- Criar e inicializar a variável do *mutex*.
- Vários *threads* tentam trancar (*lock*) o *mutex*.
- Só um deles consegue. Esse passa a ser o dono do *mutex*.
- O dono do *mutex* executa as instruções da secção crítica.
- O dono do *mutex* destranca (*unlock*) o *mutex*.
- Outro *thread* adquire o *mutex* e repete o processo.
- ...
- Finalmente, o *mutex* é destruído



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Mutexes - Inicialização

Um *mutex* é uma variável de tipo `pthread_mutex_t`.

Antes de poder ser usado, um *mutex* tem de ser inicializado.

Há 2 formas alternativas de fazer a inicialização.

Inicialização estática, quando a variável é declarada:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Inicialização dinâmica, invocando `pthread_mutex_init()`:

```
int pthread_mutex_init ( pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr);
```

`mutex`

- apontador p/ a variável que representa o *mutex*

`attr`

- permite especificar os atributos do *mutex* a criar; ver `pthread_mutexattr_init()`
- se igual a `NULL` é equivalente a inicialização estática (por omissão)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Mutexes - Lock e Unlock

```
int pthread_mutex_lock (pthread_mutex_t *mutex);  
int pthread_mutex_trylock (pthread_mutex_t *mutex);  
int pthread_mutex_unlock (pthread_mutex_t *mutex);  
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

`pthread_mutex_lock`

- tenta adquirir o *mutex*;
se ele já estiver *locked*, bloqueia o *thread* que executou a chamada
até que o *mutex* esteja *unlocked*

`pthread_mutex_trylock`

- se o *mutex* ainda não estiver *locked*, faz o *lock*
- se o *mutex* estiver *locked*, não bloqueia o *thread* e retorna `EBUSY`

`pthread_mutex_unlock`

- faz o *unlock* do *mutex*
- retorna erro se o *mutex* já estiver *unlocked*
ou estiver na posse de outro *thread*
(NOTA: *lock* e *unlock* de um dado *mutex* têm de ser feitos pelo mesmo *thread*)

`pthread_mutex_destroy`

- destrói o *mutex*



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

M
u
t
e
x
e
s

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

#define MAXPOS 10000000 /* nr. max de posições */
#define MAXTHRS 100 /* nr. max de threads */
#define min(a, b) (a)<(b)?(a):(b)
int npos;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER; /* mutex para a sec.crit. */
int buf[MAXPOS], pos=0, val=0; /* variáveis partilhadas */

void *fill(void *);
void *verify(void *);

int main(int argc, char *argv[]) {
    int k, nthr, count[MAXTHRS]; /* array para contagens */
    pthread_t tidf[MAXTHRS], tidv; /* tid's dos threads */
    if (argc != 3) {
        printf("Usage: fillver <nr_pos> <nr_thrs>\n");
        return 1;
    }
    npos = min(atoi(argv[1]), MAXPOS); /* nr. efectivo de posições */
    nthr = min(atoi(argv[2]), MAXTHRS); /* nr. efectivo de threads */
    for (k=0; k<nthr; k++) {
        count[k] = 0; /* criação dos threads fill() */
        pthread_create(&tidf[k], NULL, fill, &count[k]);
    }
    for (k=0; k<nthr; k++) {
        pthread_join(tidf[k], NULL); /* espera pelos threads fill() */
        printf("count[%d] = %d\n", k, count[k]);
    }
    pthread_create(&tidv, NULL, verify, NULL);
    pthread_join(tidv, NULL); /* thread-verificador */
    return 0;
}
```

continua

```
void *fill(void *nr)
{
    while (1) {
        pthread_mutex_lock(&mut);
        if (pos >= npos) {
            pthread_mutex_unlock(&mut);
            return NULL;
        }
        buf[pos] = val;
        pos++; val++;
        pthread_mutex_unlock(&mut);
        *(int *)nr += 1;
    }
}

void *verify(void *arg)
{
    int k;
    for (k=0; k<npos; k++)
        if (buf[k] != k) /* detecta valores errados */
            printf("buf[%d] = %d\n", k, buf[k]);
    return NULL;
}
```

```
> fillver 10000000 5
count[0] = 2802585
count[1] = 2469019
count[2] = 1361699
count[3] = 1765168
count[4] = 1601529
```



FEUP

Condition variables

- **Mutexes**
 - permitem a sincronização no acesso aos dados
 - são usados para trancar (*lock*) o acesso
- **Condition variables**
 - permitem a sincronização com base no valor dos dados
 - são usadas para esperar

Sem *condition variables*,
um programa que quisesse esperar que uma certa condição se verificasse teria de estar continuamente a testar (possivelmente numa secção crítica)
o valor da condição (*polling*), consumindo, assim, tempo de processador.

As *condition variables*
permitem fazer este teste sem *busy-waiting*.

Uma *condition variable* é sempre usada conjuntamente com um *mutex*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Espera pela condição ($x == y$) em *busy-waiting*:

```
...  
while (1) {  
    pthread_mutex_lock(&mut);  
    if (x == y)  
        break;  
    pthread_mutex_unlock(&mut);  
}  
  
/* ...  
   SECÇÃO CRÍTICA  
   ...  
*/  
  
pthread_mutex_unlock(&mut);  
...
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Espera pela condição ($x == y$) usando variáveis de condição:*Thread A*

```
...
pthread_mutex_lock(&mut);
while (x != y)
    pthread_cond_wait(&var,&mut);

/* SECÇÃO CRÍTICA */
pthread_mutex_unlock(&mut);
...
```

Thread B

```
...
pthread_mutex_lock(&mut);

/* MODIFICA O VALOR DE x E/OU y */
pthread_cond_signal(&var);
pthread_mutex_unlock(&mut);
...
```

Se ($x != y$) `pthread_cond_wait` bloqueia o *thread A* e simultaneamente (de forma indivisível) liberta o *mutex* `mut`.

Quando a *thread B* sinalizar a variável de condição `var`, o *thread A* é desbloqueado; `pthread_cond_wait()` só retorna depois de *A* ter readquirido o *mutex* `mut`, tendo, para isso, de "competir" com outras *threads* que necessitem do *mutex*.

Quando a *thread A* readquirir o *mutex* isso não significa que a condição ($x==y$, neste caso) ainda seja verdadeira. Daí a necessidade do ciclo `while`, na *thread A*.



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

***Condition variables* - Inicialização**

Uma *condition variable* é uma variável de tipo `pthread_cond_t`.

Antes de poder ser usada, uma *condition variable* tem de ser inicializada e tem de ser criado um *mutex* associado.

Inicialização estática, quando a variável é declarada:

```
pthread_cond_t mycondvar = PTHREAD_COND_INITIALIZER;
```

Inicialização dinâmica, invocando `pthread_cond_init()`:

```
int pthread_cond_init ( pthread_cond_t *cvar,
                      const pthread_condattr_t *attr);
```

`cvar`

- apontador p/ a *condition variable*

`attr`

- permite especificar os atributos do *condition variable* a criar;
ver `pthread_condattr_xxx()`
- se igual a `NULL` é equivalente a inicialização estática (por defeito)



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Condition variables - wait e signal

```
int pthread_cond_wait (pthread_cond_t *cvar, pthread_mutex_t *mutex);
int pthread_cond_signal (pthread_cond_t *cvar);
int pthread_cond_broadcast (pthread_cond_t *cvar);
int pthread_cond_destroy (pthread_cond_t *cvar);
```

pthread_cond_wait

- bloqueia o *thread* que fez a chamada até que a condição especificada seja "assinalada"
- deve ser chamada após `pthread_mutex_lock()`
- durante o bloqueio o *mutex* é libertado

pthread_cond_signal

- usada para "assinalar" ou "acordar" outro *thread* (MANUAL: "desbloqueia pelo menos 1 *thread*") que está à espera da condição

pthread_cond_broadcast

- desbloqueia todos os *threads* que nesse momento estiverem bloqueados na variável cvar (os *threads* desbloqueados "lutarão" pela aquisição do *mutex* de acordo com a política de escalonamento, como se cada um tivesse executado `pthread_mutex_lock()`; prosseguirá o que obtiver o *mutex*)

pthread_cond_destroy

- destrói a *condition variable*

NOTA: `pthread_cond_signal` e `pthread_cond_broadcast` não têm qualquer efeito se não houver processos bloqueados em *cvar*



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Condition variables

```
int x = 0, y = 10;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cvar = PTHREAD_COND_INITIALIZER;
```

```
void *test(void *a)
{
    while (1) {
        pthread_mutex_lock(&mut);
        while (x != y)
            pthread_cond_wait(&cvar, &mut);
        printf("x = y = %d\n", x);
        x = 0;
        y = y + 10;
        pthread_mutex_unlock(&mut);
    }
}
```

```
void *incr(void *a)
{
    while (1) {
        pthread_mutex_lock(&mut);
        x = x + 1;
        if (x == y)
            pthread_cond_signal(&cvar);
        pthread_mutex_unlock(&mut);
    }
}
```



FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

Condition variables

Problema do PRODUTOR-CONSUMIDOR

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

```
#define BUFSIZE 8
#define NUMITEMS 100
```

```
int buffer[BUFSIZE];
int bufin = 0;
int bufout = 0;
int items = 0;
int slots = 0;
```

```
pthread_mutex_t
buffer_lock = PTHREAD_MUTEX_INITIALIZER;
int sum = 0;
pthread_cond_t slots_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t items_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slots_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t items_lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(void){
    pthread_t prodtid, constid;
    int i, total;
    slots = BUFSIZE;
    total = 0;
    for (i = 1; i <= NUMITEMS; i++)
        total += i;
    printf("The checksum is %d\n", total);
    if (pthread_create(&constid, NULL, consumer, NULL)){
        perror("Could not create consumer");
        exit(EXIT_FAILURE);
    }
    if (pthread_create(&prodtid, NULL, producer, NULL)){
        perror("Could not create producer");
        exit(EXIT_FAILURE);
    }
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum %d\n", sum);
    exit(EXIT_SUCCESS); //EXIT_SUCCESS e EXIT_FAILURE <- stdlib.h
}
```

continua

FEUP

MIEIC
Faculdade de Engenharia da Universidade do Porto

```
void put_item(int item){
    pthread_mutex_lock(&buffer_lock);
    buffer[bufin] = item;
    bufin = (bufin + 1) % BUFSIZE;
    pthread_mutex_unlock(&buffer_lock);
    return;
}
```

```
void get_item(int *item){
    pthread_mutex_lock(&buffer_lock);
    *item = buffer[bufout];
    bufout = (bufout + 1) % BUFSIZE;
    pthread_mutex_unlock(&buffer_lock);
    return;
}
```

```
void *producer(void * arg1){
    int i;
    for (i = 1; i <= NUMITEMS; i++) {
        /* acquire right to a slot */
        pthread_mutex_lock(&slots_lock);
        while (!(slots > 0))
            pthread_cond_wait(&slots_cond, &slots_lock);
        slots--;
        pthread_mutex_unlock(&slots_lock);
        put_item(i);
        /* release right to an item */
        pthread_mutex_lock(&items_lock);
        items++;
        pthread_cond_signal(&items_cond);
        pthread_mutex_unlock(&items_lock);
    }
    pthread_exit(NULL);
}
```

```
void *consumer(void *arg2){
    int myitem;
    int i;
    for (i = 1; i <= NUMITEMS; i++) {
        pthread_mutex_lock(&items_lock);
        while(!(items > 0))
            pthread_cond_wait(&items_cond, &items_lock);
        items--;
        pthread_mutex_unlock(&items_lock);
        get_item(&myitem);
        sum += myitem;
        pthread_mutex_lock(&slots_lock);
        slots++;
        pthread_cond_signal(&slots_cond);
        pthread_mutex_unlock(&slots_lock);
    }
    pthread_exit(NULL);
}
```

FEUP

The checksum is 5050
The threads produced the sum 5050

Faculdade de Engenharia da Universidade do Porto

Notas finais

- Os *mutexes* e as *condition variables* poderão ser partilhados entre processos se forem criados em memória partilhada e inicializados com um atributo em que se inclua a propriedade `PTHREAD_PROCESS_SHARED`
- Em Linux os *threads* são implementados através da chamada `clone()` a qual cria um processo-filho do processo que a invocou partilhando parte do contexto do processo-pai.
- As funções invocadas num *thread* têm de ser *thread-safe*
 - » as funções *thread-unsafe* são, tipicamente, funções não reentrantes que guardam resultados em variáveis partilhadas
 - » em Unix, algumas chamadas de sistema que são *thread-unsafe* têm versões *thread-safe* (têm o mesmo nome acrescido de `_r`)

