

## Índice

1. Aspectos gerais dos programas em C no UNIX
2. Consola, Ficheiros e Diretórios
3. Criação e Terminação de Processos
4. Sinais
5. Pipes e FIFOs
6. Threads
7. Filas de Mensagens, Memória Partilhada, Semáforos e Variáveis de Condição
8. Funções de C

## Aspectos gerais dos programas em C no UNIX

### A função `main()`

```
int main()
int main(int argc)
int main(int argc, char* argv[])
int main(int argc, char* argv[], char* envp[])
```

- `argc`  
nº de argumentos da linha de comandos, incluindo nome do programa
- `argv`  
*array* de apontadores para strings, com os parâmetros passados ao programa, incluindo nome do programa
- `envp`  
*array* de apontadores para strings, com as variáveis de ambiente, do género "VAR=VALOR"

*Arrays* terminam em `NULL`.

### Terminação de um programa

```
void exit(int status);
```

### Tratamento de erros

```
/*
Mostra msg seguido de ': ', e descrição do último erro em chamada ao sistema.
*/
void perror (const char *msg);
```

## Consola, Ficheiros e Diretórios

### Descritores

'Identificador' de um ficheiro.

Um ficheiro pode ser aberto várias vezes e ter vários descritores (diferentes) a ele associados.

Constantes:

STDIN\_FILENO, STDOUT\_FILENO, STDERR\_FILENO

## Criação/abertura de ficheiros

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

Retorno: descritor do ficheiro se OK, -1 se erro

oflag: combinação das seguintes flags:

- O\_RDONLY
- O\_WRONLY // faz overwrite ao ficheiro
- O\_RDWR
- O\_APPEND // permite adicionar conteúdo no fim do ficheiro
- O\_CREAT // criar ficheiro se não existir, requer mode
- O\_EXCL // origina erro se ficheiro existir e O\_CREAT ativado
- O\_TRUNC // ficheiro fica com comprimento 0

mode: permissões do ficheiro, em octal (0 do lado esquerdo)

Exemplo:

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

**mode**  
**(em octal)**      **#define MODE 0750**

## Duplicação de um descritor

```
/*  
Procura descritor livre mais baixo e põe-no a apontar para o mesmo ficheiro que  
filedes.  
*/  
int dup (int filedes);  
  
/*  
Fecha filedes2 se estiver aberto, e põe-no a apontar para o mesmo ficheiro que  
filedes.  
*/  
int dup2 (int filedes, int filedes2);
```

Retorno: novo descritor se OK, -1 se erro

Exemplo:

```
/*  
Cada vez que se receber algo de stdin,  
na verdade vai parar a fd  
*/  
dup2(fd, STDIN_FILENO);
```

## Leitura, escrita, fecho e apagamento

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

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

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

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

```
int close(int fildes);
```

Retorna: 0 se OK, -1 se erro.

```
int unlink(const char *pathname);
```

Retorna: 0 se OK, -1 se erro.

## Chamadas/funções úteis

```
umask //modifica máscara de criação de ficheiros  
stat, fstat, lstat  
opendir / closedir  
readdir  
getcwd  
chdir
```

Exemplo: Percorrer diretório

```

DIR *dir;
int line;
struct dirent *dentry;
struct stat stat_entry;
...
if ((dir = opendir(argv[1])) == NULL)
{
    perror(argv[1]);
    return 2;
}
chdir(argv[1]);
printf("Ficheiros regulares do directorio '%s'\n", argv[1]);
line = 1;
while ((dentry = readdir(dir)) != NULL)
{
    stat(dentry->d_name, &stat_entry);
    if (S_ISREG(stat_entry.st_mode))
    {
        printf("%-25s%12d%3d\n", dentry->d_name,
            (int)stat_entry.st_size, (int)stat_entry.st_nlink);
    }
}
...

```

## Criação e Terminação de Processos

### A função fork

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

Retorna: 0 para filho, PID do filho para o pai, -1 se erro.

Após chamada, ambos os processos executam o mesmo código (usar *ifs* com o valor de retorno).

Após fork, não se sabe quem começa a executar primeiro.

Filho fica com cópia do segmento de dados, *heap*, e *stack*.

O processo filho pode aceder e alterar as variáveis declaradas antes de `fork()`, sem que essas alterações se sintam no processo pai.

```

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

```

### Processos *zombie*

Processo que terminou mas cujo pai ainda não executou um `wait`.

### As funções `wait` e `waitpid`

Usando estas funções, o pai espera pela terminação do filho, aceitando o seu código.

Quando processo termina, é enviado ao pai o sinal `SIGCHLD`.

```

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

```

Retorno: PID do processo se OK, -1 se erro.

## Macros para testar exit status do filho

- `WIFEXITED(status)`  
valor positivo, se terminou normalmente  
`WEXITSTATUS(status)` permite obter *exit status* do filho
- `WIFSIGNALED(status)`  
valor positivo, se recebeu sinal que não tratou
- `WIFSTOPPED(status)`  
valor positivo se filho está parado

## As funções exec

```
int execl (const char *pathname, const char *arg0, ... /* (char *)0 */);  
int execv (const char *pathname, char *const argv[]);  
int execle(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[]);
```

Apenas retornam se ocorrer erro (-1).

- l  
argumentos passados um a um, terminados por NULL
- v  
argumentos passados em *array*
- e  
passa-se variáveis de ambiente
- p  
filename é nome do ficheiro executável

## Sinais

Tratamento por omissão da maioria dos sinais é terminar o processo.  
De certa forma, podem servir para comunicação entre processos.

## A função signal

Versão simplificada do protótipo:

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

signo: identificador do sinal para o qual se vai instalar o handler

func: handler para o sinal

Retorna: apontador para o handler anterior

## Ignorar sinal

```
// Ignora sinal SIGINT (CTRL-C)
signal(SIGINT, SIG_IGN);
```

## Tratamento dos sinais após fork /exec

Após fork:

Tratamento dos sinais é herdado, mas pode ser alterado.

Após exec:

Tratamento é por omissão, ou ignorar se o processo que invocou exec também estiver a ignorar.

## As funções kill e raise

```
// Envia um sinal ao processo pid.
int kill (pid_t pid, int signo);
```

Retorno: 0 se OK, -1 se erro

## As funções alarm e pause

```
unsigned int alarm(unsigned int count);
```

Retorno: 0 se OK; -1 se ocorreu erro

Processo que invocou recebe SIGALRM após count segundos.

Se count = 0, algum alarme pendente é cancelado.

## Funções Posix p/sinais

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

Retorno: 0 se OK; -1 se ocorreu erro

Exemplo de instalação:

```
struct sigaction action;
action.sa_handler = sigint_handler;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
sigaction(SIGINT,&action,NULL);
```

Função handler é do género:

```
void handler(int signo)
{}
```

## *Pipes e FIFOs*

Permitem que dois processos a correr no mesmo computador enviem dados um ao outro.

## *Pipes vs FIFOs*

Em ambos, os dados apenas podem fluir num sentido.

Os pipes têm que ser usados por processos com antecessor comum, enquanto que no caso dos FIFOs estes não têm que ser relacionados.

## Pipes

Funcionalidade:

- Criar pipe;
- Invocar fork;
- Escritor fecha extremidade de leitura, leitor fecha de escrita.
- Usar write e read;
- No final, fechar outras extremidades

fd[0] -> aberto para leitura

fd[1] -> aberto para escrita

```
int pipe (int fd[2]);
```

Retorno: 0 se ok, -1 se erro

### Utilização

1. Enviar dados de processo para outro
2. Sincronizar dois processos (usar dois pipes)
3. Ligar stdout de um processo a stdin de outro
  - duplicar descritores de um pipe para a stdin de um processo e stdout de outro.

### Funções popen e pclose

```
/*  
Cria pipe entre processo que a invocou e programa a executar  
*/  
FILE *popen(const char *cmdstring, const char *type);
```

cmdstring -> programa a executar

type -> 'r' para leitura, 'w' para escrita

Retorna: file pointer se OK; NULL se houve erro

```
int pclose(FILE *fp);
```

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

Estes comandos são úteis para **filtros**: um programa lê da stdin e escreve para stdout. Outro programa utiliza-o, criando um pipeline com as funções acima.

## FIFOs

Tipo de ficheiro. Tem um nome e pode ser visto no sistema de ficheiros.

Pode ter vários processos a escrever nele, mas apenas 1 a ler a sua informação.

No terminal existem os seguintes utilitários:

```
mkfifo nome //criar fifo com nome
```

rm nome // elimina fifo com nome

A API fornece as seguintes funções:

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

Retorno: 0 se OK, -1 se erro

Se FIFO existir, acontece o erro EEXIST.

Onde mode representa as permissões de acesso, em octal. Começa por 0, e os três restantes algarismos representam as permissões de leitura, escrita e execução, para o dono, grupo, outros, respetivamente.

Comummente, o seu valor é 0660.

Para manipular o fifo, usar as funções já referidas open, close, read, e write.

### Regras:

#### Abertura

- Em modo O\_RDONLY, espera que outra extremidade seja aberta para escrita, se não estiver e O\_NONBLOCK estiver desativada.
- Em modo O\_WRONLY, espera que outra extremidade seja aberta para leitura, se não estiver e O\_NONBLOCK estiver desativada. Se estiver ativada, retorna -1.

#### Leitura/Escrita

Leitura de um pipe não aberto para escrita - retorna 0 (eof)

Escrita para FIFO que não está aberto para leitura - SIGPIPE

```
// Destrói o fifo de pathname.  
int unlink (const char *pathname);
```

Exemplo de arquitetura comunicacional:





## Exemplos

### Criar um FIFO

```
char * myfifo = "/tmp/myfifo"; //Path do FIFO
mkfifo(myfifo, 0660); //Criar FIFO chamado myfifo na pasta tmp
```

## Threads

### Criação de *threads*

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

- tid  
apontador para identificador do *thread* (preenchido pela função)
- attr  
atributos do *thread*, normalmente NULL (valor por defeito)
- func  
função executada pelo *thread*,  
do género:

```
void* func(void* arg)
```

- arg  
apontador para argumento do *thread*

### Terminação de *threads*

Quando main termina, *threads* criados por si também são terminados.  
Mas, se terminar com chamada pthread\_exit(), os outros *threads* continuam em execução.

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

Não retorna.

```
// Thread bloqueia até que thread tid terminar.
int pthread_join (pthread_t tid, void **status);
```

### Dicas para uso de *threads*

Função da *thread* pode retornar qualquer apontador. Não esquecer de fazer free no final do programa.

Passagem de argumentos:

Ao criar *threads* em ciclo for, os argumentos devem ser passados através de um *array* externo:  
Errado:

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

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

Certo:

```
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]);
    ...
}
```

Em determinadas alturas, pode ser conveniente alocar dinamicamente memória para o argumento.

Para passar vários argumentos, usar *structs*.

Para vários threads acederem ao mesmo conteúdo, este pode ser declarado com variável global, ou em região de memória partilhada.

```
// Saber o tid da própria thread : útil quando se pretende que uma thread se
torne *detached*
pthread_t pthread_self (void);
```

Para tornar uma thread *detached*, utiliza-se o serviço:

```
int pthread_detach(pthread_t thread);
// thread é o indentificador da thread que se pretende tornar *detached*
```

Threads *detached* não são *joinable*, ou seja, é impossível esperar por elas com `pthread_join()`. Este tipo de threads, quando terminam, libertam todos os seus recursos, incluindo o seu valor de retorn.

## Filas de Mensagens, Memória Partilhada, Semáforos, *Mutexes*, *Condition variables*

### Comunicação entre Processos

Correndo na mesma máquina:

- Pipes e FIFOs
- Mensagens
- Semáforos
- Memória partilhada

### Semáforos

Semáforos com nome podem ser partilhados por vários processos, sem nome por processos com acesso a memória comum

```
sem_t* sem_open(char* name, int flags, mode_t mode, unsigned value);
int sem_unlink(char* name);
//Deve ser inicializado antes das threads serem criadas
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);
```

name deve ter '/' no início.

Exemplo:

Uma *thread*, antes de aceder a variável partilhada, executa

`sem_wait()` com o semáforo associado;

no final, executa `sem_post()` com o outro semáforo.

## Memória partilhada

```
//Retorna o descritor associado
int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);

/* Especifica tamanho da região de
memória partilhada identificada por fd */
int ftruncate(int fd, off_t length);

/*
junta região de memória partilhada ao espaço de enderçamento do processo
*/
void* mmap(void *start, size_t length, int prot , int flags,
int fd, off_t offset);

/*
Inverso da chamada anterior - retira
*/
int munmap(void *start, size_t length);
```

## Sincronização de threads

### ***Mutexes***

Podem ser vistos como semáforos inicializados a 1, usados para garantir exclusão mútua de secções críticas.

Sequência de utilização:

- Criar e inicializar *mutex*
- Vários threads tentam dar lock ao *mutex*
- Só um consegue
- O dono do *mutex* executa secção crítica

- Este dá unlock ao *mutex*
- Outro thread adquire *mutex* e processo é repetido
- *Mutex* é destruído

## Inicialização

Preferivelmente, no *scope* global

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
/* ou */
int pthread_mutex_init ( pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);
```

## Lock e unlock

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
/* tenta adquirir o mutex, se já estiver locked,
bloqueia até que fique unlocked */
int pthread_mutex_trylock (pthread_mutex_t *mutex);
/* se estiver unlocked, faz lock, senão
retorna EBUSY */
int pthread_mutex_unlock (pthread_mutex_t *mutex);
/* faz unlock do mutex, retorna erro se já estiver unlocked
ou estiver na posse de outro thread */
int pthread_mutex_destroy (pthread_mutex_t *mutex);
/* destrói o mutex */
```

## Condition variables

*Mutexes* permitem sincronização no acesso aos dados, trancando-o;

*Condition variables* permitem sincronização com base no valor dos dados, esperando.

Sem *condition variables*, se um programa quisesse esperar por certa condições teria que estar continuamente a testar esse valor, consumindo tempo de processador.

Com elas, não há *busy-waiting*.

Devem se **sempre** usadas a par com *mutexes* (ver exemplo em baixo).

## Inicialização

Deve ser feita antes da invocação dos *threads* (preferivelmente, no *scope* global)

```
pthread_cond_t mycondvar = PTHREAD_COND_INITIALIZER;
/* ou */
int pthread_cond_init ( pthread_cond_t *cvar,
const pthread_condattr_t *attr);
```

```

/*
bloqueia thread até condição se assinalar
deve ser chamada após pthread_mutex_lock()
*/
int pthread_cond_wait (pthread_cond_t *cvar, pthread_mutex_t *mtx);

/*
assinala/acorda outro thread
*/
int pthread_cond_signal (pthread_cond_t *cvar);
/*
desbloqueia todos threads que estiverem bloqueados em cvar
*/
int pthread_cond_broadcast (pthread_cond_t *cvar);

/*
destrói condition variable
*/
int pthread_cond_destroy (pthread_cond_t *cvar);

```

Exemplo:

```

//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);

```

## Uso em processos

*Mutexes e condition variables* podem ser partilhados entre processos se forem criados em memória partilhada e inicializados com um atributo que inclua a propriedade `PTHREAD_PROCESS_SHARED`.

## Funções de C

```
// Envia output formatado para str.
int sprintf(char *str, const char *format, ...)

// Converte str em int.
int atoi(const char *str)

// Aloca memória e retorna-a.
void *malloc(size_t size)

// Liberta memória alocada previamente.
void free(void *ptr)

// Escreve para um ficheiro.
int fprintf(FILE *stream, const char *format, ...)

// Abre um ficheiro em modo 'r' ou 'w'
FILE *fopen(const char *filename, const char *mode)

// Envia str para stream
int fputs(const char *str, FILE *stream)

// Lê uma linha (ou max n chars) de stream para str.
char *fgets(char *str, int n, FILE *stream)

// Retorna tamanho de uma string
size_t strlen(const char *str)

// Adiciona src ao final de dest (deve ter espaço suficiente)
char *strcat(char *dest, const char *src)
```