Criação e terminação de processos

A função fork é invocada 1 vez, mas tem 2 retornos e tem 2 fluxos de execução.
Após a chamada fork() pai e filho executam o mesmo código.

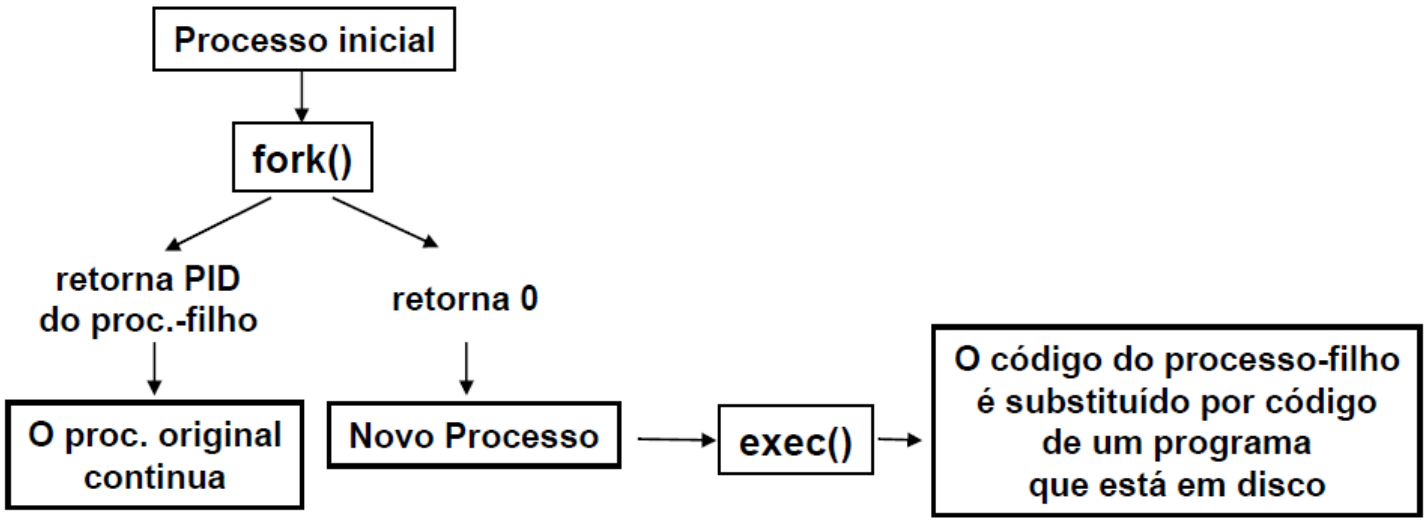
```
int main(){
    pid_t pid = fork();
    printf("Hello");
}
```

Na sequência de um fork, vão resultar 2 processos, que vão ser executados 2 vezes, uma pelo fork e outra pelo que resultou do fork, pelo que neste caso vão ser imprimidas 2 vezes "Hello". Se o fork retornar o pid do processo filho, o processo original continua (processo pai). Se retornar 0, então inicia-se um novo processo (processo filho). Se retornar -1, então ocorreu um erro no processo.

Para que pai e filho executem diferentes segmentos de código (objetivo de utilizar a função fork()), tem de se utilizar as intruções condicionais. Deste modo, o valor de retorno do fork(), já vai ser diferente para pai e filho.

```
int main(){
    pid_t pid = fork();
    printf("Hello");
    if(pid > 0)
        printf("Hello from parent\n");
    else
        printf("Hello from son\n");
}
```

Deste modo, só vai haver uma execução de um processo, caso pid > 0 será o processo pai, case pid = 0 será o processo filho.



```
pid_t pid = fork(); (program counter pai) (retorna 1235)
pid_t pid = fork(); (program counter filho) (retorna 0)
```

Estes 2 processos correm em paralelo, não há garantia que haja um que execute primeiro do que o outro.

Normalmente o que se pretende fazer com o fork é que o processo pai e o processo filho executem diferentes tarefas.
Para que os processos executem diferentes códigos, utiliza-se uma chamada **exec("Prog_novo", ...)** (exs: servidores - o pai espera por um cliente; quando o cliente chega, passa o processo para o filho que vai resolve-lo; o pai volta a ficar a espera de um cliente). Outra hipótese é nos casos em que um processo quer executar um programa diferente (exs: shells - o filho faz exec (do comando) depois de retornar do fork.

funções **getpid** (retorna pid do filho) e **getppid** (retorna pid do pai).

A funçao fork pode falhar quando o nº total de processos no sistema ou do utilizador é muito elevado.

Os processos podem terminar de duas formas:

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

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*

Processos Orfãos

Casos em que o pai termina antes do filho. Neste caso, o filho é automaticamente aotado por init (processo com PID igual a 1).
Deste modo, qualquer processo tem um pai.
Quando o processo pai termina, o kernel percorre todos os processoa ativos a ver se algum deles é filho deste processo que terminou. Nesse caso, o PID desse processo passa a ser 1.

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.
A informação sobre o processo filho não pode desaparecer completamente. Assim, o kernel mantém essa informação de modo a que ela esteja disponível quando o pai executar um dos wait's. O resto da memória ocupada pelo processo filho é libertada e o processo é fechado.

Funções wait e waitpid

Quando um processo termina, 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 ficam zombies.
- Dispor de signal handler - O handler poderá executar um dos wait's para obter a PID do filho e o seu termination status.

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

O wait espera por que um qualquer filho termine, e depois recolhe o seu sinal.
O waitpid sabe qual é o pid do filho pelo qual tem de esperar.

O argumento statloc:

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

O argumento pid de waitpid:

- pid == -1 - espera por um filho qualquer ($\equiv$ 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).

O argumento options de waitpid é útil quando toma o valor WNOHANG que permite que o pai não fique pendurado à espera que o processo filho especificado por pid esteja disponível. Se o pid em waitpid for -1 e options for WNOHANG, isto permite que o processo pai não tenha de esperar por nenhum dos processos filhos caso estes demorem mais.

WNOHANG - indica que waitpid() não bloqueia se o filho especificado por pid não estiver imediatamente disponível (terminado). Neste caso o valor de retorno é igual a 0.
WUNTRACED - indica que, se a implementação suportar job control, o estado de terminação de qualquer filho especificado por pid que tenha terminado e cujo status ainda não tenha sido reportado desde que ele parou, é agora retornado.

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 .

Um processo que invoque wait() ou waitpid() pode:

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

Diferenças entre wait() e waitpid():

- serviço wait() pode bloquear o processo que o invoca até que um filho qualquer termine;
- serviço waitpid() tem uma opção que impede o bloqueio (útil quando se quer apenas obter o código de terminação do filho);
- waitpid() não espera que o 1º filho termine, tem um argumento para indicar o processo pelo qual se quer esperar.

Quando se usam vários forks consecutivos, convém que de vez enquanto se faça um ciclo com um wait para ver se algum dos processos filho ja terminou. Para isso, à medida que se vai criando filhos, guarda-se os pid's destes num array, por exemplo, e depois no ciclo faz-se por ex. waitpid(array[pid-1]).

Funções exec()

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 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[]);

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

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

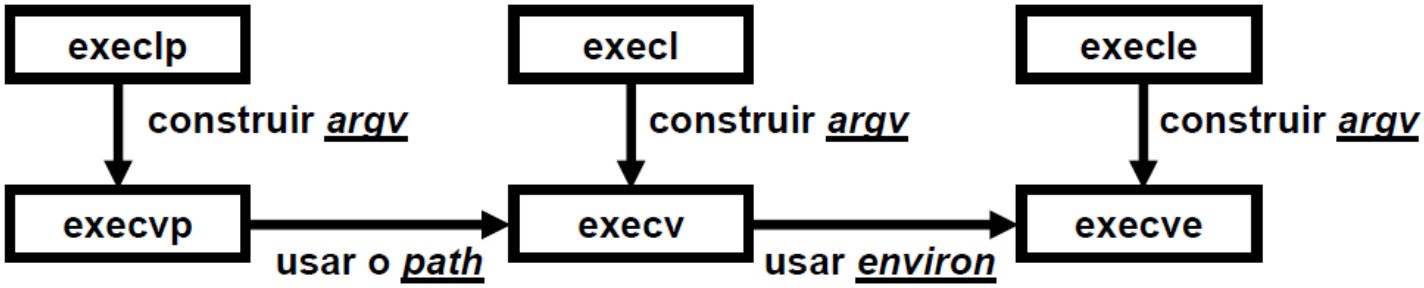
- **l** - Lista, passados um a um separadamente, terminados por um apontador nulo.
- **v** - Vector, passados num vector.

Passagem das strings de ambiente (environment):

- **e** - Passa-se um apontador para um array de apontadores para as strings .
- **sem e** - 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.
- **sem p** - O nome do ficheiro executável deve incluir o path.



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);
```

Retorno de system:

- Se cmdstring = null pointer
  - retorna valor □ 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 ≠ 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.

Sinais

Um sinal é:

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

Um sinal ode ter origem no teclado (certas combinações de teclas), no hardware (referência inválida à memória), na função de sistema kill (permite que um processo envie um sinal a outro processo ou conjunto de processos), no comando kill (permite enviar um sinal a um processo ou conjunto de processosa partir de shell) ou no software (certos acontecimentos gerados por software dão origem a sinais).

Os sinais podem ser gerados sincronamente ou assincronamente.
Todos os sinais têm um nome simbólico que começa por SIG.
Respostas possíveis a um sinal:

- Ignorar - a maior parte pode ser ignorada, exceto SIGKILL e SIGSTOP;
- Tratar (catch) - indicar uma função a executar quando o sinal correr;
- Executar a ação por omissão - todos os sinais têm uma ação por omissão.

Exemplos de sinais

Nome	Descrição	Origem	Acção por omissão
SIGINT	Terminar processo	teclado (^C)	terminar
SIGQUIT	Terminar processo e gerar *core dump*	teclado (^\)	terminar
SIGTSTP	Suspender processo (*job control*)	teclado (^Z)	suspender
SIGCONT	Continuar processo suspenso (depois de SIGSTP)	*shell* (comandos: fg, bg)	continuar
SIGKILL	Terminar processo (*non catchable*)	sistema operativo	terminar
SIGSTOP	Parar a execução (*non catchable*)	sistema operativo	suspender
SIGTERM	Terminar processo	*default* do comando kill	terminar
SIGABRT	Terminar processo (anormalmente)	abort()	terminar
SIGALRM	Alarme	alarm()	terminar
SIGSEGV	Referência a memória inválida	*hardware*	terminar
SIGFPE	Excepção aritmética	*hardware*	terminar
SIGILL	Instrução ilegal	*hardware*	terminar
SIGUSR1	Definido pelo utilizador		terminar
SIGUSR2	Definido pelo utilizador		terminar

Função signal

Chamada signal permite associar uma rotina de tratamento (signal handler) a um determinado sinal. (ex: signal(SIGINT, inthandler) ou signal(SIGINT, SIG_IGN) quando é possível ignorar o sinal).

```
#include <signal.h>
void (*signal(int signo, void (*func) (int))) (int);
```

Simplificando:

```
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

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.

ex: define SIG_IGN (void (*) (int)) 1 - apontador para uma função que tem como parametro um inteiro e que retorna void.

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 ou então pode 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).

Tratamento de 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.

Porque será? Ao fazer exec, as rotinas de tratamento "perdem-se" pois o código já não é o mesmo.

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;

Funções Kill e Raise

Kill

- envia um sinal a um processo ou a um conjunto de processos;
- não tem necessariamente como consequência o fim dos processos.

Raise

- envia um sinal ao processo que a invocar.

```
#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;
```

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).

Funções alarm e pause

```
#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 (ularm, 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.

Funções abort e sleep

```
#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; ≡ waitraise(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).

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.

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 atual não é alterada; usado apenas quando se quer apenas obter a máscara atual;
- **old_mask**:
  - se NULL a máscara atual não é retornada.

sigset_t é um tipo de dados constituído por um conjunto de bits, cada um representando um sinal.

```
#include <signal.h>
int sigemptyset(sigset_t *sigmask);
int sigfillset(sigset_t *sigmask);
int sigaddset(sigset_t *sigmask, int sig_num);
int sigdelset(sigset_t *sigmask, int sig_num);

Retorno: 0 se OK; -1 se ocorreu erro

int sigismember(const 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 (Quando usado com sigpromask(SIG_SETMASK, sigmask, ...), todos os sinais passam);
- sigfillset() - ativar todas as flags da máscara;
- sigaddset() - ativar 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.

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).

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 ação associada a um sinal.

- **signum** - nº do sinal cuja ação se quer examinar ou modificar;
- **action** - se $\neq$ NULL estamos a modificar;
- **oldaction** - se $\neq$ NULL o sistema retorna a ação anteriormente associada ao sinal.

```
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 ou outro manual */
}
```

A função sigaction() substituí 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)).

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().

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.

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

- 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.

Chamadas ao sistema interrompidas por sinais

- 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.

Pipes e FIFOS

Pipes e FIFOS

Os pipes são um mecanismo de comunicação que permite que dois ou mais processos a correr no mesmo computador enviem dados uns aos outros.
Existem 2 tipos de pipes:

- pipes sem nome (unnamed pipes ou apenas pipes):
  - São half-duplex ou unidirecionais, ou seja, os dados só podem fluir num sentido;
  - Só podem ser usados entre processos que tenham um antecessor comum (podem ser usados, por ex., por um pai e um filho ou por um pai e um neto, etc. No entanto se for um processo entre um pai e um neto, tem de ser o avo a criar o pipe).
- pipes com nome (named pipes ou FIFOS):
  - São half-duplex ou unidirecionais;
  - Podem ser usados por processos não relacionados entre si;
  - Têm um nome que os identifica, existente no sistema de ficheiros.

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.

```
#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.

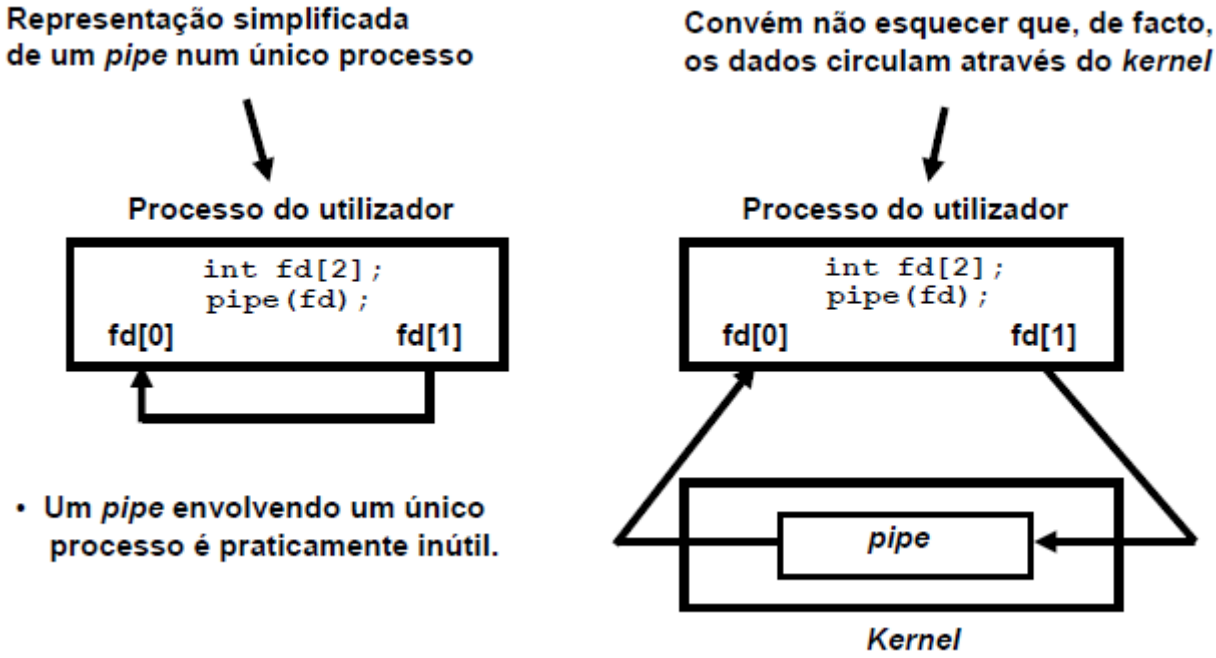
```
# include <unistd.h>
ssize_t read (int fd, char * buf, int count);
ssize_t write (int fd, char * buf, int count);

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

fdopen()

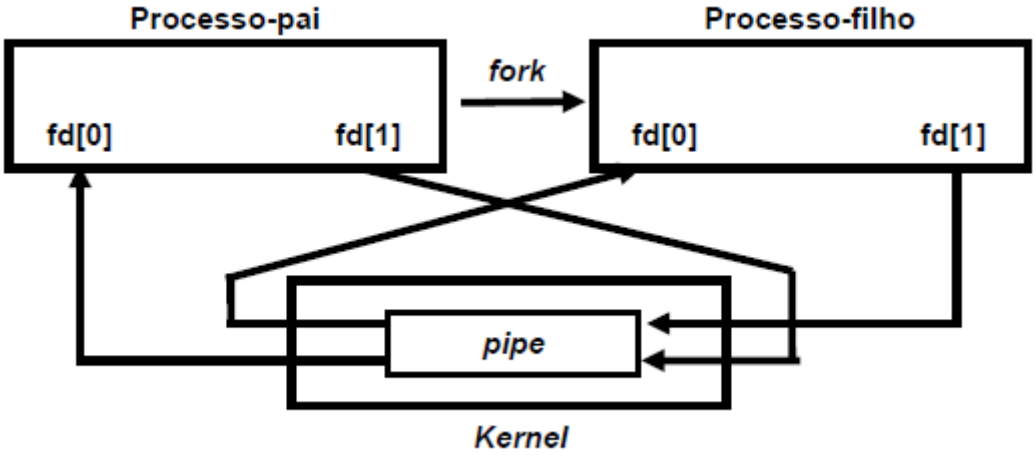
```
# include <stdio.h>
FILE * fdopen(int fildes, const char *mode)

Retorna: FILE * se bem sucedida ou NULL se houve erro (e errno é actualizada)
```

- fdopen() associa uma stream a um descritor, filedes, já existente;
- Desta forma, é possível aceder a um pipe usando as funções de leitura/ escrita da biblioteca standard de C: fscanf(), fprintf(), fread(), fwrite(), ...
- O modo da stream (um dos valores "r", "r+", "w", "w+", "a", "a+") deve ser compatível com o modo do descritor do ficheiro.

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. Apos o fork, o processo filho herda os descritores do pai, ficando-se com a capacidade do processo pai escrever e ler, ou processo filho ler e escrever ou entao um deles lê e o outro escreve.
- 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]