

---

---

# Sinais

---

---

Jorge Silva

MIEIC / FEUP

---

---

## 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)

---

---

Jorge Silva

MIEIC / FEUP

---

---

# 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.: ctrl-C, ctrl-Z, ctrl-\ (v. adiante)
- *Hardware*
  - » referência inválida à memória
- Função de sistema *kill*
  - » permite que um processo envie um sinal a outro processo ou conjunto 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

---

---

# Sinais

- Os sinais podem ser gerados:
  - sincronamente
    - » associados a uma certa acção executada pelo próprio processo  
(ex: acesso inválido a memória)
  - 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

## Exemplos de sinais

Nome	Descrição	Origem	Acção por omissão
SIGINT	Terminar processo	teclado (^C)	terminar
SIGQUIT	Terminar processo e gerar <i>core dump</i>	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	Terminar processo ( <i>non catchable</i> )	sistema operativo	terminar
SIGSTOP	Parar a execução ( <i>non catchable</i> )	sistema operativo	suspender
SIGTERM	Terminar processo	<i>default</i> do comando kill	terminar
SIGABRT	Terminar processo (anormalmente)	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

Jorge Silva

MIEIC / FEUP

## Respostas a um sinal

### Ignorar o sinal

- A maior parte dos sinais podem ser ignorados.
- SIGKILL e SIGSTOP nunca podem ser ignorados.

### Tratar o sinal

- Indicar uma função a executar (*signal handler*) quando o sinal ocorrer.
  - » Exemplo:  
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 *wait* para obter a PID e o *termination status* do filho (cuidado, se muitos filhos terminarem "simultaneamente" ...)

### Acção por omissão

- Todos os sinais têm uma accção por omissão (v. tabela) .
- Possíveis acções do *default handler*
  - » terminar o processo e gerar uma *core file*
  - » terminar o processo sem gerar uma *core file*
  - » ignorar o sinal
  - » suspender o processo
  - » continuar o processo

Jorge Silva

MIEIC / FEUP

# 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*)
- NOTA : `signal ( )` é uma chamada obsoleta e não portátil;  
é aqui introduzida para facilitar a introdução dos conceitos;  
recomenda-se o uso de `sigaction ( )`, a introduzir posteriormente

## A função *signal*

### Protótipo:

```
#include <signal.h>

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

### Declaração complicada !

Para simplificar a interpretação podemos fazer:

```
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)

# 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.

## 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");
}
```

O uso de "signo" como parâmetro do *handler* permite usar o mesmo *handler* para tratar vários sinais ...

## 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
$
```

## 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

---

# 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.

---

# 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.

# 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*

# 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 *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*



## 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
```

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

## 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
```

## 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

## 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; $\equiv$ 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)

# Exemplo

## Estabelecimento de um alarme e respectivo *handler*

```
#include ...

int alarmflag = 0;

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

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

# 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);
}
```

# 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("...?????\n");
        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);
}
```

# 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 (t) até que este termine. Caso isso não aconteça, termina-o de modo forçado.**

**Exemplo de linha de comando:**

**limit t 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);
}
```

# 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

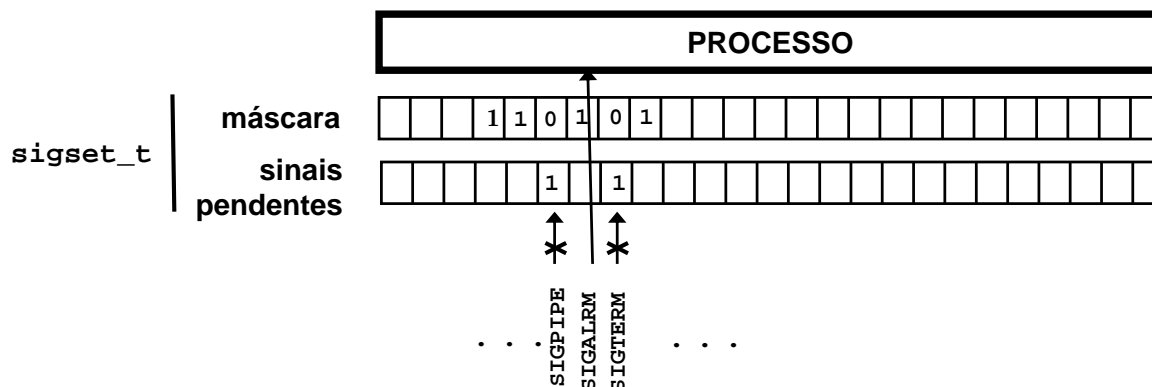
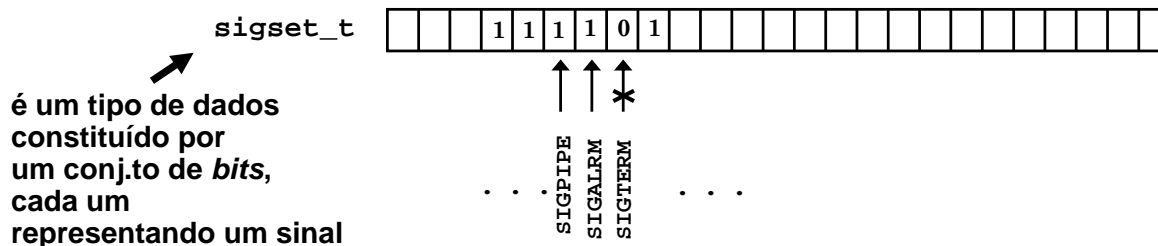
`sigpending`

- 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`, `sigignore`, `sighold`, `sigrelse`, `sigpause`, ...

- criar e manipular a máscara de sinais

## Máscara de sinais



# 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

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

# 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(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

**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

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

# Exemplo


## Bloquear SIGINT, mantendo o tratamento dos outros sinais

Nota: bloquear != ignorar

```
#include ...

int main(void)
{
    sigset_t sigmask;

    ...

    if (sigprocmask(0,NULL,&sigmask)==-1) /* obter másc. actual */
    {
        perror("sigprocmask"); exit(1);
    }
    else
    {
        sigaddset(&sigmask,SIGINT);
        if (sigprocmask(SIG_BLOCK, &sigmask, NULL)) 
            perror("sigprocmask"); exit(2);
    }
    ...
}
```

# 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");
...
```

## 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 (nota: pode ter mais campos)

```
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*

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)*



# 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);
    }
}
```

# 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 repostada com o valor que tinha antes da invocação de `sigsuspend()`.

# 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();
    ...
}
```

O que acontece se for recebido  
outro sinal diferente de SIGINT ?



... usando sigsuspend ( )

```
#include ...

...

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



O que acontece se o sinal chegar  
e o *handler* for executado  
entre o teste de flag e pause ( ) ?



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

---

**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.

---

**Chamadas 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

---


# 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 ...

```
(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:  $f(a, (t=3, t+2), b) \equiv f(a, 5, b)$