

# Sinais UNIX

Onofre Trindade Jr

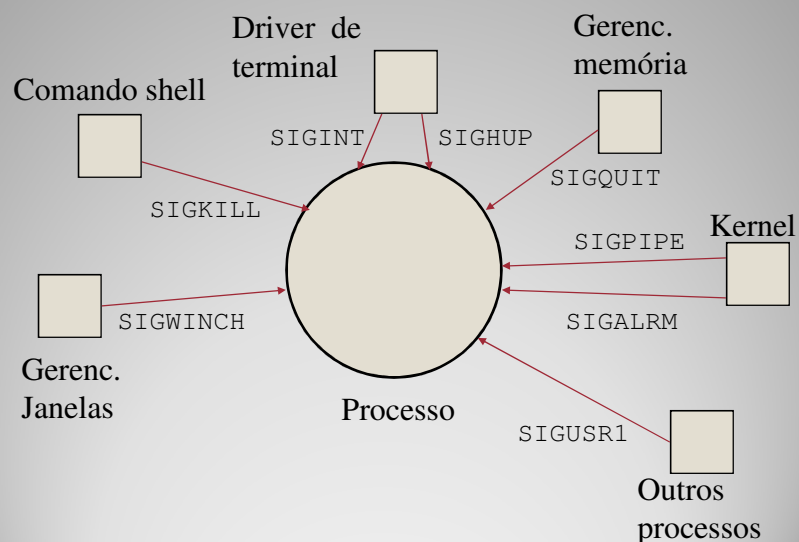
- Um sinal é um evento *assíncrono* que é enviado para um processo. Assíncrono significa que o evento pode ocorrer a qualquer momento
  - Pode não estar relacionado com a execução do processo, ex., o usuário digita `ctrl-C`

## Definição

- | <i>Nome</i> | <i>Descrição</i>            | <i>Ação default</i> |
|-------------|-----------------------------|---------------------|
| SIGINT      | Interrupt character typed   | terminate process   |
| SIGQUIT     | Quit character typed (^\\)  | create core image   |
| SIGKILL     | kill -9                     | terminate process   |
| SIGSEGV     | Invalid memory reference    | create core image   |
| SIGPIPE     | Write on pipe but no reader | terminate process   |
| SIGALRM     | alarm() clock 'rings'       | terminate process   |
| SIGUSR1     | user-defined signal type    | terminate process   |
| SIGUSR2     | user-defined signal type    | terminate process   |

- Veja `man 7 signal`

## Tipos de Sinais (31 no POSIX)



## Origem dos Sinais

- Comando UNIX:

```
$ kill -KILL 4481
```

- envia um sinal `SIGKILL` ao processo 4481
- verificação para ter certeza que o processo morreu
  - `ps -l`

## Gerando um Sinal

- Envia um sinal para um processo (ou grupo de processos).
- `#include <signal.h>`  
  
`int kill( pid_t pid, int signo );`
- Retorna 0 if ok, -1 se houver erro

## kill()

- | <i>pid</i> | <i>Significado</i>   |
|------------|--|
| > 0        | envia sinal ao processo <code>pid</code>   |
| == 0       | envia sinal para todos os processos cujo ID de grupo de processos seja igual o PGID, do processo que está enviando ex., pai mata todos os processos filhos |
- PID = identificador do processo
  - PPID = identificador do processo pai
  - PGID = identificador do grupo de processos

### Alguns Valores de PID

- Um processo pode:
  - ignorar/descartar o sinal (exceto para os sinais `SIGKILL` OU `SIGSTOP`)
  - executar uma função de atendimento do sinal, e então terminar ou continuar a execução
  - executar a ação default para aquele sinal
- Esta **escolha** é conhecida como a *disposição ao sinal* do processo

### Atendendo um Sinal

- Especifica como será tratado cada sinal
- `#include <signal.h>`  
`typedef void Sigfunc(int); /* my defn */`  
  
`Sigfunc *signal( int signo, Sigfunc  
*handler );`
  - `signal` retorna um ponteiro para uma função que retorna um *int* (i.e. retorna um ponteiro para `Sigfunc`)
- Retorna a disposição anterior do sinal se ok, `SIG_ERR` caso contrário

### Chamada de Sistema `signal()`

- O protótipo que aparece nas páginas de manual é uma extensão do tipo `Sigfunc`

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

- No Linux  
`typedef void (*sighandler_t)(int);`  
  
`sig_handler_t signal(int signo, sighandler_t  
handler);`

### Protótipo

A função `signal`, por sua vez, retorna um ponteiro para uma função. O tipo retornado é o mesmo da função que é passada como argumento, ou seja, uma função que recebe `int` e retorna `void`.

Para usar a chamada `signal`, a função a ser tratada ou o sinal a ser tratado ou o tipo retornado é passado como argumento.

A função `handler` recebe um inteiro e retorna `void`.

Signal é uma função com dois argumentos: `sig` and `handler`.

A função a ser chamada quando o sinal é recebido é especificada como parâmetro.

A função retornada recebe um inteiro como parâmetro.

```
void (*signal( int sig, void (*handler)(int)) (int) ;
```

- `signal` retorna um ponteiro para o handler anterior de sinal

de si

```
int main()
{
    signal( SIGINT, foo );
    :
    /* do usual things until SIGINT */
    return 0;
}

void foo( int signo )
{
    :
    /* deal with SIGINT signal */
    return;
}
/* return to program */
```

**Exemplo**

```

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

void sig_usr( int signo );    /* handles two signals
    */

int main()
{
    int i = 0;
    if( signal( SIGUSR1,sig_usr ) == SIG_ERR )
        printf( "Cannot catch SIGUSR1\n" );
    if( signal( SIGUSR2,sig_usr ) == SIG_ERR )
        printf("Cannot catch SIGUSR2\n");
    :

```

**sinal.c**

```

:
while(1)
{
    printf( "%2d\n", I );
    pause();
    /* pause until signal handler
       * has processed signal */
    i++;
}
return 0;
}

```

```

void sig_usr( int signo )
/* argument is signal number */
{
    if( signo == SIGUSR1 )
        printf("Received SIGUSR1\n");
    else if( signo == SIGUSR2 )
        printf("Received SIGUSR2\n");
    else
        printf("Error: received signal
                %d\n", signo);

    return;
}

```

```

$ sinal &
[1]      4720
0
$ kill -USR1 4720
Received SIGUSR1
1
$ kill -USR2 4720
Received SIGUSR2
2
$ kill 4720          /* envia SIGTERM */
[1] + Terminated   sinal &
$

```

**Execução**



• Valor	Significado
SIG_IGN	Ignore / discard the signal.
SIG_DFL signal.	Use default action to handle
SIG_ERR	Returned by <code>signal()</code> as an error.

### Valores Especiais de Sigfunc \*

- Se muitos sinais do mesmo tipo estão esperando tratamento (ex. dois `SIGINTs`), a maioria dos sistemas UNIX envia para o processo somente um deles
  - os outros são descartados
- Se muitos sinais de *diferentes* tipos estão esperando tratamento (ex. `SIGINT`, `SIGSEGV`, `SIGUSR1`), não existe uma ordem fixa para eles serem enviados ao processo

### Múltiplos Sinais

- Suspende o processo até que um sinal seja recebido
- `#include <unistd.h>`  
`int pause(void);`
- Retorna -1 e `errno` com o valor `EINTR`. (Linux atribui `ERESTARTNOHAND`).
- `pause()` somente retorna após o retorno da função de tratamento do sinal (*handler*)

**pause()**

- No Linux (e muitos outros UNIXs), a disposição de um sinal em um é "resetada" para a ação default imediatamente após o sinal ter sido recebido
- Para tratar adequadamente o próximo sinal, a função `signal()` tem que ser chamada novamente

**O Problema do Reset**

```

int main()
{
    signal(SIGINT, foo);
    :
    /* do usual things until SIGINT */
}

void foo(int signo)
{
    signal(SIGINT, foo); /* reinstall */
    :
    return;
}

```

## O Problema do Reset

```

:
void ouch( int sig )
{
    printf( "OUCH! - I got signal\n" );
    (void) signal(SIGINT, ouch);
}

int main()
{
    (void) signal(SIGINT, ouch);
    while(1)
    {
        printf("Hello World\n");
        sleep(1);
    }
}

```

Para continuar o tratamento do sinal SIGINT, a função *signal* deve ser chamada novamente

**Problema:** desde o momento em que a função de tratamento inicia até o momento em que a função *signal* é chamada novamente, o sinal SIGINT será tratado com seu comportamento padrão

## O Problema do Reset

- Existe um período dentro da rotina de tratamento do sinal durante o qual um novo sinal `SIGINT` será tratado da forma padrão (default), que é o encerramento do processo que o recebe
- Com a função `signal()` não existe solução para este problema
  - Funções de tratamento de sinais do **POSIX** solucionam este problema

### O Problema do Reset

- Ignorar um sinal
- Limpar e terminar
- Reconfiguração dinâmica
- Reportar status
- Ligar/desligar modo debug
- Restaurar *handler* anterior

### Uso Comum dos Sinais

```

:
int main()
{
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, SIG_IGN);
    :
    /* do work without interruptions */
}

```

- Não se pode ignorar SIGKILL ou SIGSTOP
- Deve-se verificar SIG\_ERR

## Ignorar um Sinal

```

:
/* global variables */
int my_children_pids;
:
void clean_up(int signo);

int main()
{
    signal(SIGINT, clean_up);
    :
}

```


## Limpar e Terminar

```

void clean_up(int signo)
{
    unlink("/tmp/work-file");
    kill(my_children_pids, SIGTERM);
    wait((int *)0);
    fprintf(stderr, "Program terminated\n");
    exit(1);
}

```

## Limpar e Terminar



```

:
if( signal(SIGINT, SIG_IGN ) != SIG_IGN )
    signal(SIGINT, clean_up);

if( signal(SIGQUIT, SIG_IGN ) != SIG_IGN )
    signal(SIGQUIT, clean_up);
:

```

- *Nota:* não é possível verificar uma disposição a sinal sem alterá-la (a função `sigaction()`, que será vista adiante, funciona de forma diferente)

## Verificando uma Disposição

```

:
void read_config(int signo);

int main()
{
    read_config(0);  /* dummy argument */

    while (1)
        /* work forever */
}

```

## Reconfiguração Dinâmica

```

void read_config(int signo)
{
    int fd;

    signal( SIGHUP, read_config );

    fd = open("config_file", O_RDONLY);
    /* read file and set global vars */
    close(fd);

    return;
}

```

## Reconfiguração Dinâmica

- Problema do Reset
- Interrupção da rotina de tratamento
  - qual é o efeito de um sinal `SIGHUP` no meio da execução da rotina `read_config()`?
- Pode somente afetar variáveis globais

### Reconfiguração Dinâmica - Problemas

```
    :
void print_status(int signo);
int count;    /* global */

int main()
{ signal(SIGUSR1, print_status);
  :
  for( count=0; count < BIG_NUM; count++ )
  {
    /* read block from tape */
    /* write block to disk */
  }
  ...
}
```

### Reportar Status



```
void print_status(int signo)
{
    signal(SIGUSR1, print_status);
    printf("%d blocks copied\n", count);
    return;
}
```

### Reportar Status

- Problema do Reset
- O valor de `count` nem sempre está definido
- Obriga o uso de variáveis globais para informação de status

### Reportar Status - Problemas

```

:
void toggle_debug(int signo);

int debug = 0; /* initialize here */

int main()
{
    signal(SIGUSR2, toggle_debug);

    /* do work */
    if (debug == 1)
        printf("...");
    ...
}

```

**Ligar/Desligar Modo Debug**

```

void toggle_debug(int signo)
{
    signal(SIGUSR2, toggle_debug);

    debug = ((debug == 1) ? 0 : 1);

    return;
}

```

**Ligar/Desligar Modo Debug**

```
    :
    Sigfunc *old_hand;

    /* set action for SIGTERM;
       save old handler */
    old_hand = signal(SIGTERM, foobar);

    /* do work */

    /* restore old handler */
    signal(SIGTERM, old_hand);
    :
```

### **Restaurar *handler* Anterior**

- Colocar um limite superior de tempo em uma operação que pode permanecer bloqueada para sempre
  - ex. `read()`
- `alarm()`, timeout ruim
- `setjmp()` e `longjmp()`, timeout melhor

### **`read()` Timeout**

- Configura um alarme temporizado que dispara após o número especificado de segundos
  - Um sinal `SIGALRM` é gerado
- ```
#include <unistd.h>
long alarm(long secs);
```
- Retorna 0 ou o número de segundos restantes para o disparo

## alarm()

- Um processo pode ter somente um `alarm()` em execução de cada vez
- Se `alarm()` é chamada quando já existe um `alarm()` em andamento, ela retorna o número de segundos restantes da chamada antiga e configura o novo valor de tempo
- Uma chamada `alarm(0)` provoca o cancelamento do alarme em andamento

## Detalhes

```

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

#define MAXLINE  512

void sig_alm( int signo );

int main()
{
    int n;
    char line[MAXLINE];
    :

```

### read() Timeout – Implementação Ruim

```

if( signal(SIGALRM, sig_alm) == SIG_ERR )
{
    printf("signal(SIGALRM) error\n");
    exit(1);
}

alarm(10);
n = read( 0, line, MAXLINE );
alarm(0);

if( n < 0 ) /* read error */
    fprintf( stderr, "\nread error\n" );
else
    write( 1, line, n );
return 0;
}

```

### read() Timeout – Implementação Ruim

```
void sig_alm(int signo)
/* do nothing, just handle signal */
{
    return;
}
```

### read() Timeout – Implementação Ruim

- Assume-se que a função `read()` termina com um erro se for interrompida
- **Corrida crítica**: O Kernel pode levar mais de 10 segundos para iniciar a função `read()` depois da chamada da função `alarm()`
  - o alarme pode tocar antes do início da função `read()`
  - neste caso, a função `read()` não estará sendo monitorada e pode bloquear para sempre
  - Maneiras para resolver este problema:
    - `set jmp`
    - `sigprocmask` and `sigsuspend`

### Problemas com a Implementação

- Em C não se pode usar `goto` para saltar para um rótulo em uma função diferente da atual
  - Pode-se utilizar `setjmp()` and `longjmp()` para isso
- Usos razoáveis para `setjmp()` and `longjmp()` :
  - Gerenciamento de erros onde é necessário que uma função retorne para uma outra função muitos níveis de chamada superior (ex. de volta para a `main()`)
  - Codificar timeouts com sinais

## setjmp() e longjmp()

- ```
#include <setjmp.h>
int setjmp( jmp_buf env );
```
- Retorna 0 se chamada diretamente, não-zero no retorno de uma chamada `longjmp()`.
- ```
#include <setjmp.h>
void longjmp( jmp_buf env, int val );
```

## Protótipos

- Na chamada `setjmp()`, `env` é iniciada com a informação sobre o estado atual da pilha
- A chamada `longjmp()` faz com que a pilha seja restaurada para o estado definido por `env`
- A execução recomeça após a chamada `setjmp()`, mas neste caso, `setjmp()` retorna `val`.

## Comportamento

```
:
jmp_buf env;          /* global */

int main()
{
    char line[MAX];
    int errval;

    if(( errval = setjmp(env) ) != 0 )
        printf( "error %d: restart\n", errval );

    while( fgets( line, MAX, stdin ) != NULL )
        process_line(line);
    return 0;
}
```

## Exemplo



```

:
void process_line( char * ptr )
{
:
cmd_add()
:
}

```

**Exemplo**

```

void cmd_add()
{
int token;

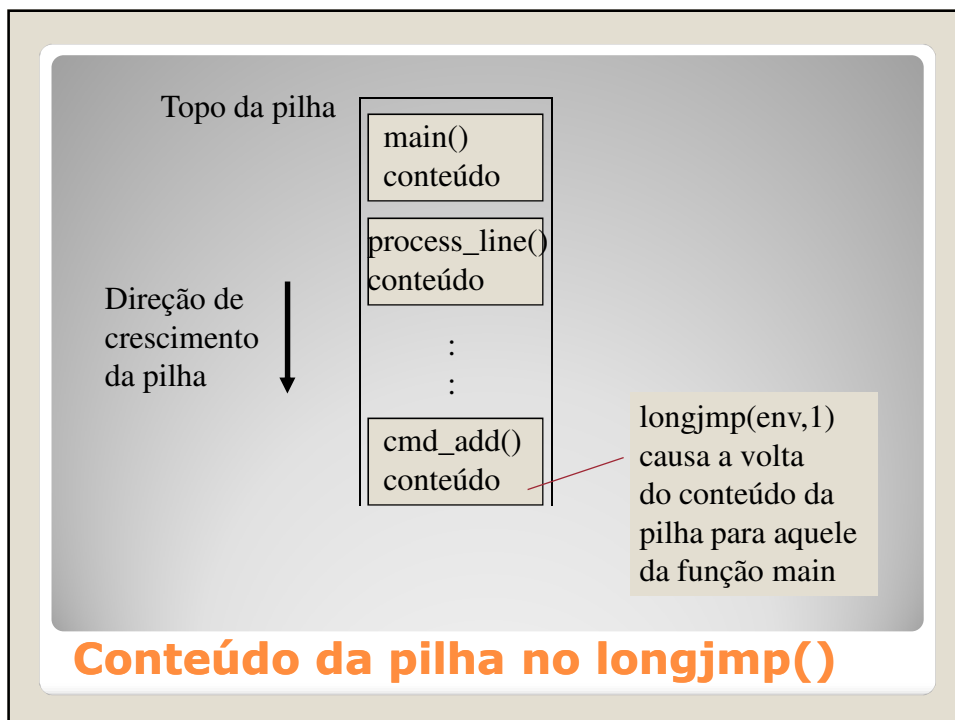
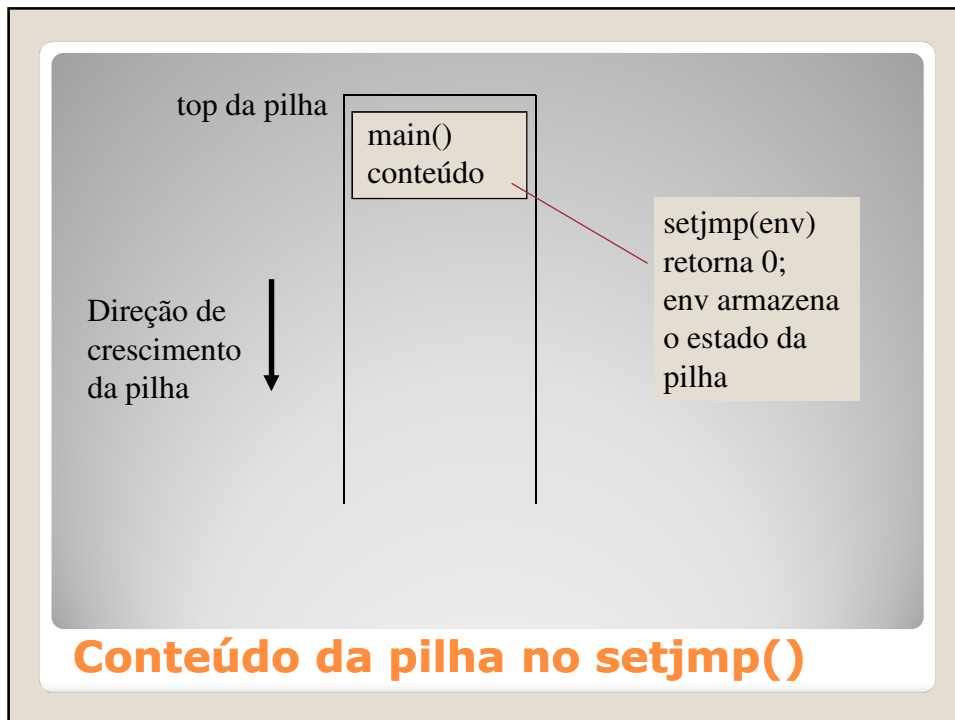
token = get_token();
if( token < 0 )      /* bad error */
    longjmp( env, 1 );

/* normal processing */
}

int get_token()
{
if( some error )
    longjmp( env, 2 );
}

```

**Exemplo**



```

#include <signal.h>
#include <unistd.h>

void sig_alm( int signo )
{
    return; /* return to wake up pause */
}

unsigned int sleep1( unsigned int nsecs )
{
    if( signal( SIGALRM, sig_alm ) == SIG_ERR )
        return (nsecs);
    alarm( nsecs );      /* starts timer */
    pause();             /* next caught signal wakes */

    /* turn off timer, return unslept time */
    return( alarm( 0 ) );
}

```

**sleep1()**

```

static void jmp_buf env_alm;

void sig_alm( int signo )
{
    longjmp( env_alm, 1 );
}

unsigned int sleep2( unsigned int nsecs )
{
    if( signal( SIGALRM, sig_alm ) == SIG_ERR )
        return (nsecs);
    if( setjmp( env_alm ) == 0 )
    {
        alarm( nsecs );      /* starts timer */
        pause();             /* next caught signal wakes */
    }
    return( alarm( 0 ) );
}

```

**sleep2()**

- Na função `sleep1`, o alarme pode disparar antes da execução da função `pause`, e neste caso, o sinal `SIGALARM` não poderá mais interrompe-la
- A função `sleep2` resolve a corrida crítica mesmo que a função `pause` nunca for executada
- Ainda existe um problema que será visto adiante

## Sleep1 and Sleep2

- O padrão POSIX diz:
  - Os valores de variáveis **globais** e **estáticas** não são alterados com a chamada `longjmp()`
- Nada é especificado a respeito de variáveis locais. Elas têm os valores restaurados ao valores originais (no momento da chamada `setjmp()`) ?
  - Elas podem ou não ter seus valores restaurados para o valor do primeiro `setjmp()`
    - A maioria das implementações não restaura os valores dessas variáveis

## Valores das variáveis

```

#include <stdio.h>
#include <unistd.h>
#include <setjmp.h>
#include <signal.h>

#define MAXLINE 512
void sig_alm( int signo );
jmp_buf env_alm;

int main()
{ int n;
  char line[MAXLINE];
  :

```

## read() Timeout melhorado

```

if( signal(SIGALRM, sig_alm) == SIG_ERR)
{
    printf("signal(SIGALRM) error\n");
    exit(1);
}

if( setjmp(env_alm) != 0 )
{
    fprintf(stderr, "\nread() too slow\n");
    exit(2);
}

alarm(10);
n = read(0, line, MAXLINE);
alarm(0);

```

## read() Timeout melhorado

```
if( n < 0 ) /* read error */
    fprintf( stderr, "\nread error\n" );
else
    write( 1, line, n );
return 0;
}
```

**read() Timeout melhorado**

```
void sig_alrm(int signo)
/* interrupt the read() and jump to
   setjmp() call with value 1
*/
{
    longjmp(env_alrm, 1);
}
```

**read() Timeout melhorado**

#### WARNINGS

If `longjmp()` or `siglongjmp()` are called even though `env` was never primed by a call to `setjmp()` or `sigsetjmp()`, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

#### Jumps não locais, advertência

- Se um programa tem várias funções de tratamento de sinais:
  - Uma delas pode estar em execução quando um sinal de alarme acontece
  - A chamada `longjmp()` salta para a localização da chamada `setjmp()`, e aborta a outra rotina de tratamento de sinal, podendo ocorrer perda ou corrupção de dados

#### Problemas

- No POSIX, as funções de tratamento podem controlar os sinais de maneiras diferentes:
  - Podem bloquear temporariamente os sinais (bom para seções críticas de código)
  - Podem desligar o reset dos sinais ao seu tratamento default depois do primeiro tratamento (o problema do reset)

### **Tratamento de Sinais no POSIX**

- O tratamento de sinais no POSIX usa conjuntos de sinais para lidar com sinais pendentes de tratamento que poderiam ser perdidos enquanto outro sinal está sendo processado

### **Conjunto de Sinais**



- Um conjunto de sinais armazena coleções de tipos de sinais
- Conjuntos são usados pelas funções de tratamento de sinais para definir que tipos de sinais devem ser processados
- O POSIX contém várias funções para criar, alterar e examinar conjuntos de sinais

## Conjuntos de Sinais

```
• #include <signal.h>

int sigemptyset( sigset_t *set );
int sigfillset( sigset_t *set );

int sigaddset( sigset_t *set, int signo );
int sigdelset( sigset_t *set, int signo );

int sigismember( const sigset_t *set,
                  int signo );
```

## Protótipos das Funções

- Um processo usa um conjunto de sinais para criar uma máscara que define quais sinais estão sendo bloqueados
- Bom para seções críticas de código
- `#include <signal.h>`  
`int sigprocmask( int how,`  
`const sigset_t *set,`  
`sigset_t *oldset);`

## sigprocmask()

- **how**
  - indica como a máscara é modificada
- | <i>Valor</i>             | <i>Significado</i>                                      |
|--------------------------|---------------------------------------------------------|
| <code>SIG_BLOCK</code>   | Os sinais de <code>set</code> são adicionados à máscara |
| <code>SIG_UNBLOCK</code> | Os sinais de <code>set</code> são removidos da máscara  |
| <code>SIG_SETMASK</code> | A nova máscara passa a ser <code>set</code>             |

## Parâmetro how

```

sigset_t newmask, oldmask;

sigemptyset( &newmask );
sigaddset( &newmask, SIGINT );

/* block SIGINT; save old mask */
sigprocmask( SIG_BLOCK, &newmask, &oldmask );

/* critical region of code */

/* reset mask which unblocks SIGINT */
sigprocmask( SIG_SETMASK, &oldmask, NULL );

```

### Região de Código Crítica

- Mais poderosa que a função `signal()`
  - `sigaction()` pode ser usada como uma função `signal()` sem o problema do reset
- `#include <signal.h>`

```

int sigaction(int signo,
               const struct sigaction *act,
               struct sigaction *oldact );

```

### sigaction()

```

struct sigaction
{
    /* action to be taken or SIG_IGN, SIG_DFL */
    void (*sa_handler)( int );

    /* signals to be blocked */
    sigset_t sa_mask;

    /* modifies action of the signal */
    int sa_flags;

    void (*sa_sigaction)( int, siginfo_t *, void *
);
}

```

### Estrutura sigaction

- Um sinal `signo` faz com que `sa_handler` seja chamada
- Enquanto a `sa_handler` está em execução, os sinais em `sa_mask` estão bloqueados. Quaisquer sinais `signo` adicionais também estão bloqueados
- `sa_handler` permanece instalada até que seja trocada por outra chamada para a função `sigaction()`, Eliminando o problema do reset

### sigaction() Behavior

```
int main()
{
    struct sigaction act;

    act.sa_handler = ouch;

    sigemptyset( &act.sa_mask );
    act.sa_flags = 0;

    // ... SIGINT, &act, 0 ...
    // ... Hello World ...
}
```

```
struct sigaction
{
    void (*) (int) sa_handler
    sigset_t sa_mask
    int sa_flags
}
```

Define a função ouch para tratar o sinal

Pode-se manipular conjuntos de sinais...

Não são necessários flags neste caso. Possíveis flags: SA\_NOCLDSTOP SA\_NOCLDSTOP SA\_NOCLDSTOP SA\_NOCLDSTOP

Esta chamada define a função de tratamento do sinal SIGINT (ctrl-C)

## Tratamento dos sinais

- No programa anterior, o sinal ctrl-C (SIGINT) é capturado continuamente
- O comportamento default não é restaurado após o sinal ter sido capturado pela primeira vez
- Para terminar o programa, deve-se digitar ctrl-\\, o sinal SIGQUIT

## Tratamento dos sinais

```

/* sigexPOS.c - demonstrate sigaction() */
/* include files as before */

int main(void)
{
    /* struct to deal with action on signal set */
    static struct sigaction act;

    void catchint(int); /* user signal handler */

    /* set up action to take on receipt of SIGINT */
    act.sa_handler = catchint;

```

## Exemplo

```

/* create full set of signals */
sigfillset(&(act.sa_mask));

/* before sigaction call, SIGINT will terminate
the process */

/* now, SIGINT will cause catchint to be executed */
sigaction( SIGINT, &act, NULL );
sigaction( SIGQUIT, &act, NULL );

printf("sleep call #1\n");
sleep(1);

etc...

```

## Exemplo

- Sinais podem ser ignorados, com exceção dos sinais SIGKILL e SIGSTOP
- Se no programa anterior em vez de usar:  
`act.sa_handler = catchint /* or whatever */`
- For utilizado  
`act.sa_handler = SIG_IGN;`

A tecla ^C será ignorada

## **Ignorando Sinais**

- Utilizando o terceiro parâmetro da função `sigaction()`

```
/* save old action */
sigaction( SIGTERM, NULL, &oact );

/* set new action */
act.sa_handler = SIG_IGN;

sigaction( SIGTERM, &act, NULL );

/* restore old action */
sigaction( SIGTERM, &oact, NULL );
```

## **Restaurando Disposição Anterior**

```

#include <signal.h>

Sigfunc *signal( int signo, Sigfunc *func )
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset( &act.sa_mask );
    act.sa_flags = 0;

    act.sa_flags |= SA_INTERRUPT;
    if( signo != SIGALRM )
        act.sa_flags |= SA_RESTART;
    /* any system call interrupted by a signal
     * other than alarm is restarted */
    if( sigaction( signo, &act, &oact) < 0 )
        return(SIG_ERR);
    return( oact.sa_handler );
}

```

## Codificando a função signal()

- sigpending()      examinar sinais bloqueados
- sigsetjmp()  
siglongjmp()      funções de salto para uso em rotinas de tratamento de sinais
- sigsuspend()      automaticamente reset a máscara e suspende o processo

## Outras Funções POSIX



- `longjmp`, `siglongjmp`  
O POSIX não especifica se o `longjmp` restaura ou não o contexto de execução. Se você quiser salvar e restaurar máscaras de sinais, utilize `siglongjmp`
- `setjmp`, `sigsetjmp`  
O POSIX não especifica se `setjmp` salva ou não o contexto do sinal. (No SystemV não salva. No BSD4.3 salva, e existe uma função `_setjmp` que não salva). Se você deseja salvar máscaras de sinais, use `sigsetjmp`

## [sig]longjmp & [sig]setjmp

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

void handler(int sig)
{
    siglongjmp(buf, 1);
}

main()
{
    signal(SIGINT, handler);

    if( sigsetjmp(buf, 1) == 0 )
        printf("starting\n");
    else
        printf("restarting\n");
    ...

    ...
    while(1)
    {
        sleep(5);
        printf(" waiting...\n");
    }
}
```

> a.out  
starting  
waiting...  
waiting... ← Control-c  
restarting  
waiting...  
waiting...  
waiting...  
restarting ← Control-c  
waiting...  
restarting ← Control-c  
waiting...  
waiting...

## Exemplo

- Quando uma chamada de sistema, (ex. `read()`) é interrompida por uma rotina de tratamento de sinal, quando esta rotina retorna o que acontece?
- Na maioria dos sistemas UNIX, chamadas de sistema *lentas* não prosseguem a execução. Ao contrário, elas terminam com um código de erro `errno = EINTR`

### Chamadas de Sistema Interrompidas

- Chamadas de sistema lentas são aquelas que definem funções de E/S que podem bloquear indefinidamente:
  - Pipes, drivers de terminais, redes
  - Algumas funções de IPC
  - `pause()`, e alguns usos de `ioctl()`
- Pode-se utilizar sinais para codificar timeouts nessas funções

### Chamadas de Sistema Lentas

- A maioria das chamadas de sistema são não lentas, incluindo aquelas de E/S em disco
  - ex. `read()` para um arquivo em disco
  - `read()` algumas vezes é uma função lenta, outras vezes não
- Alguns sistemas UNIXs terminam as funções não lentas após o retorno da rotina de tratamento de sinal
- Alguns sistemas UNIXs somente chamam as rotinas de tratamento de sinal após o retorno das chamadas não lentas

### Chamadas de Sistema não Lentas

- Se uma chamada de sistema é feita dentro de uma rotina de tratamento de sinal, ela pode interagir com uma chamada interrompida para a mesma função dentro do código principal
  - ex. `malloc()`
- Isto não constitui um problema se a função é *reentrante*
  - ex. `read()`, `write()`, `fork()`, etc...

### Chamadas de Sistema Dentro das Rotinas de Tratamento de Sinais

- Funções podem ser não reentrantes por uma série de razões:
  - Usam uma estrutura de dados estática
  - Manipulam a heap: `malloc()`, `free()`, etc.
  - Usam a biblioteca de E/S padrão
    - `ex`, `scanf()`, `printf()`
    - Esta biblioteca utiliza estruturas de dados globais

### Funções não Reentrantes

- `errno` é usualmente uma variável global
- Seu valor pode ser substituído por uma rotina de tratamento de sinal

### O Problema do `errno`