5. Comunicação entre processos - Sinais

5.1 Definição dos sinais

Os sinais são uma espécie de interrupção ao processo corrente. Podem ter diversas origens e são uma forma de tratar certos acontecimentos de carácter assíncrono.

Todos os sinais têm um nome simbólico que começa pelo prefixo `SIG`. Vêm listados no ficheiro de inclusão `<signal.h>`.

Possíveis origens: teclado - certas combinações de teclas dão origem a sinais; hardware - por exemplo, erros aritméticos como a divisão por 0; serviços do sistema operativo - `kill()`, `abort()`, `alarm()`, ...; comandos da shell - kill; software - certos eventos gerados no código dos processos dão origem a sinais, como erros de *pipes*.

Um processo pode programar a resposta a dar a cada tipo de sinal. As respostas permitidas são: ignorar o sinal, tratar o sinal com uma rotina do programador (*catch*), ou simplesmente deixar que o processo tenha a resposta por defeito (*default action*) para esse tipo de sinal.

Na tabela seguinte apresentam-se alguns sinais definidos nos Sistemas Operativos Unix:

Nome	Descrição	Origem	Acção por defeito
`SIGABRT`	Terminação anormal	`abort()`	Terminar
`SIGALRM`	Alarme	`alarm()`	Terminar
`SIGCHLD`	Filho terminou ou foi suspenso	S.O.	Ignorar
`SIGCONT`	Continuar processo suspenso	S.O. shell (fg, bg)	Continuar
`SIGFPE`	Excepção aritmética	hardware	Terminar
`SIGILL`	Instrução ilegal	hardware	Terminar
`SIGINT`	Interrupção	teclado (^C)	Terminar
`SIGKILL`	Terminação (*non catchable*)	S.O.	Terminar
`SIGPIPE`	Escrever num *pipe* sem leitor	S.O.	Terminar
`SIGQUIT`	Saída	teclado (^|)	Terminar
`SIGSEGV`	Referência a memória inválida	hardware	Terminar
`SIGSTOP`	Stop (*non catchable*)	S.O. (shell - stop)	Suspender
`SIGTERM`	Terminação	teclado (^U)	Terminar
`SIGTSTP`	Stop	teclado (^Y, ^Z)	Suspender
`SIGTTIN`	Leitura do teclado em *backgd*	S.O. (shell)	Suspender
`SIGTTOU`	Escrita no écran em *backgd*	S.O. (shell)	Suspender
`SIGUSR1`	Utilizador	de 1 proc. para outro	Terminar
`SIGUSR2`	Utilizador	idem	Terminar

5.2 Tratamento dos sinais

Quando se cria um processo novo a resposta a todos os sinais definidos no sistema é a acção por defeito. No entanto o processo, através da chamada de serviços do sistema, pode apanhar (*catch*) um determinado sinal e ter uma rotina (*handler*) para responder a esse sinal, ou simplesmente pode ignorá-lo (excepto para os sinais `SIGKILL` e `SIGSTOP`).

A instalação de um handler para o tratamento de um determinado sinal (ou a

determinação de o ignorar) pode ser feita através do serviço `signal()`.

Este serviço `signal()` faz também parte da biblioteca standard do C pelo que deve existir em todos os S.O.'s.

```
#include <signal.h>

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

onde **signo** é o identificador do sinal a "apanhar" e **handler** o nome da função que será chamada sempre que o processo receber um sinal do tipo **signo**.

A função retorna o endereço do handler anterior para este sinal ou `SIG_ERR` no caso de erro.

A declaração do serviço `signal()` é um pouco complicada. Compreende-se melhor se se declarar primeiro o seguinte tipo:

```
typedef void sigfunc(int);
```

Esta declaração define um tipo chamado **sigfunc** que é uma função com um parâmetro inteiro retornando nada (um procedimento). Com o tipo **sigfunc** definido podemos agora declarar o serviço `signal()` como:

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

ou seja `signal()` é uma função com dois parâmetros: um inteiro (identificador do sinal - **signo**) e um apontador para uma função - **handler**; `signal()` retorna um apontador para função do tipo **sigfunc**.

Assim os handlers de sinais deverão ser funções que não retornam nada com um parâmetro inteiro. Após a instalação de um handler para um determinado sinal, sempre que o processo recebe esse sinal o handler é chamado, com parâmetro de entrada igual ao identificador do sinal recebido.

Existem duas constantes que podem ser passadas no lugar do parâmetro **handler** de `signal()`:

- `SIG_IGN` - quando se pretende ignorar o sinal **signo**;
- `SIG_DFL` - quando se pretende reinstalar a acção por defeito do processo para o sinal **signo**.

5.3 Envio de sinais

Além das origens dos sinais já referidas atrás é possível um processo enviar um sinal qualquer a ele próprio ou a outro processo (desde que seja do mesmo utilizador). Os serviços utilizados para isso são:

```
#include <signal.h>

int raise(int signo);
```

Retorna 0 no caso de sucesso; outro valor no caso de erro.

O serviço `raise()` envia o sinal especificado em **`signo`** ao próprio processo que executa o serviço. A resposta depende da disposição que estiver em vigor (acção por defeito, *catch*, ou ignorar) para o sinal enviado.

O serviço que permite o envio de sinais entre processos diferentes é:

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

int kill(pid_t pid, int signo);

Retorna 0 no caso de sucesso; outro valor no caso de erro.
```

O processo de destino do sinal **`signo`** é especificado em **`pid`**. No caso de **`pid`** especificar o próprio processo a função `kill()` só retorna após o tratamento do sinal enviado pelo processo.

5.4 Outros serviços relacionados com sinais

O serviço `alarm()` enviará ao próprio processo o sinal `SIGALRM` após a passagem de **`seconds`** segundos. Se no momento da chamada estiver activa outra chamada prévia a `alarm()`, essa deixa de ter efeito sendo substituída pela nova. Para cancelar uma chamada prévia a `alarm()` sem colocar outra activa pode-se executar `alarm(0)`.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

Retorna 0 normalmente, ou o número de segundos que faltavam a uma possível chamada
prévia a alarm().
```

O serviço `pause()` bloqueia o processo que o chamou (deixa de executar). O processo só voltará a executar quando receber um sinal qualquer não ignorado. O serviço `pause()` só retornará se o handler do sinal recebido também retornar.

```
#include <unistd.h>

int pause(void);

Quando retorna, retorna o valor -1.
```

O serviço `abort()` envia o sinal `SIGABRT` ao próprio processo. O processo não deve ignorar este sinal. A acção por defeito deste sinal é terminar imediatamente o processo sem processar eventuais handlers `atexit()` existentes (ver capítulo 1).

```
#include <stdlib.h>

void abort(void);
```

O serviço `sleep()` suspende (bloqueia) temporariamente o processo que o chama, pelo intervalo de tempo especificado em **`seconds`**. A suspensão termina quando o intervalo de tempo se esgota ou quando o processo recebe qualquer sinal não ignorado e o respectivo

handler retornar.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Retorna 0 normalmente, ou o número de segundos que faltam para completar o serviço.

Seguem-se alguns exemplos da utilização de sinais.

Exemplo 1 - processo que trata apenas os sinais `SIGUSR1` e `SIGUSR2`:

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

static void sig_usr(int);     /* o mesmo handler para os 2 sinais */

int main(void)
{
      if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
            fprintf(stderr, "can't catch SIGUSR1\n");
            exit(1);
      }
      if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
            fprintf(stderr, "can't catch SIGUSR2\n");
            exit(1);
      }
      for ( ; ; )
            pause();
}

static void sig_usr(int signo)    /* o argumento indica o sinal recebido */
{
      if (signo == SIGUSR1)
            printf("received SIGUSR1\n");
      else if (signo == SIGUSR2)
            printf("received SIGUSR2\n");
      else {
            fprintf(stderr, "received signal %d\n", signo);
            exit(2);
      }
      return;
}
```

Exemplo 2 - Estabelecimento de um alarme e respectivo handler

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

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

void main(void)
{
    signal(SIGALRM, alarmhandler);
    alarm(5);
    printf("Looping ...\n");
```

```
    while (!alarmflag)
        pause();
    printf("Ending ...\n");
}

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

Exemplo 3 - Protecção contra `CTRL-C` (que gera o sinal `SIGINT`)

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

void 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");
}
```

Exemplo 4 - Aqui pretende-se um programa que lance um outro e espere um certo tempo para que o 2º termine. Caso isso não aconteça deverá terminá-lo de modo forçado.

Exemplo de linha de comando:
`limit n prog arg1 arg2 arg3`

`n` - nº de segundos a esperar
`prog` - programa a executar
`arg1, arg2, ..., argn` - argumentos de **`prog`**

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

int delay;
void childhandler(int signo);

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

    signal(SIGCHLD, childhandler);   /* quando um processo filho termina */
    pid = fork();                    /* envia ao pai o sinal SIGCHLD */
    if (pid = 0)                     /* filho */
        execvp(argv[2], &argv[2]);
    else {                              /* pai */
        sscanf(argv[1], "%d", &delay);  /* transforma string em valor */
        sleep(delay);
```

```
        printf("Program %s exceeded limit of %d seconds!\n",
               argv[2], delay);
        kill(pid, SIGKILL);
    }
}

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

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

5.5 Bloqueamento de sinais

A norma POSIX especifica outras formas mais complicadas de instalação de handlers para sinais e permite um outro mecanismo para o tratamento de sinais. Com os serviços POSIX é possível bloquear selectivamente a entrega de sinais a um processo. Se um determinado sinal estiver bloqueado (diferente de ignorado) quando for gerado, o S.O. não o envia para o processo, mas toma nota de que esse sinal está pendente. Logo que o processo desbloqueie esse sinal ele é imediatamente enviado ao processo (pelo S.O.) entrando em acção a disposição que estiver em vigor (acção por defeito, *catch* com handler, ou ignorar). Se vários sinais idênticos forem gerados para um processo, enquanto este os estiver a bloquear, geralmente o S.O. só toma nota de que um está pendente, perdendo-se os outros.

Para permitir especificar qual ou quais os sinais que devem ser bloqueados por um processo, este deve construir uma máscara. Esta máscara funciona como uma estrutura de dados do tipo "conjunto" que é possível esvaziar, preencher completamente com todos os sinais suportados, acrescentar ou retirar um sinal, ou ainda verificar se um dado sinal já se encontra na máscara.

Os serviços definidos em POSIX para estas operações são os seguintes e operam sobre uma máscara de tipo `sigset_t` definido no ficheiro de inclusão `signal.h`:

```
#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);
```

Todos estes serviços têm como primeiro argumento o endereço de uma máscara (**`set`**) do tipo `sigset_t`.
Os quatro primeiros retornam 0 se OK e -1 no caso de erro. O quinto retorna 1 se *true* e 0 se *false*.

O serviço `sigemptyset()` preenche a máscara como vazia (sem nenhum sinal), enquanto que `sigfillset()` preenche a máscara com todos os sinais suportados no sistema. O serviço `sigaddset()` acrescenta o sinal **`signo`** à máscara, emquanto que `sigdelset()`

retira o sinal **signo** à máscara. Finalmente `sigismember()` testa se o sinal **signo** pertence ou não à máscara.

Tendo construído uma máscara contendo os sinais que nos interessam podemos bloquear (ou desbloquear) esses sinais usando o serviço `sigprocmask()`.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Se **oset** não for `NULL` então é preenchido com a máscara corrente do processo (contendo os sinais que estão bloqueados).
Se **set** não for `NULL` então a máscara corrente é modificada de acordo com o valor do parâmetro **how**:

- `SIG_BLOCK` - a nova máscara é a reunião da actual com a especificada em **set**, i.e. **set** contém sinais adicionais a bloquear (desde que ainda não estejam bloqueados);
- `SIG_UNBLOCK` - a nova máscara é a intersecção da actual com o complemento de **set**, i.e. **set** contém os sinais a desbloquear;
- `SIG_SETMASK` - a nova máscara passa a ser a especificada em **set**.

Retorna 0 se OK e -1 no caso de erro.

Para, por exemplo, bloquear os sinais `SIGINT` e `SIGQUIT`, podíamos escrever o seguinte código:

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

sigset_t sigset;

sigemptyset(&sigset);          /* comecemos com a máscara vazia */
sigaddset(&sigset, SIGINT);    /* acrescentar SIGINT */
sigaddset(&sigset, SIGQUIT);   /* acrescentar SIGQUIT */
if (sigprocmask(SIG_BLOCK, &sigset, NULL))
      perror("sigprocmask");
```

A norma POSIX estabelece um novo serviço para substituir `signal()`. Esse serviço chama-se `sigaction()` e além de permitir estabelecer uma disposição para o tratamento de um sinal, permite numerosas outras opções, entre as quais bloquear ao mesmo tempo outros sinais.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
                         struct sigaction *oact);
```

O serviço `sigaction()` permite modificar e/ou examinar a acção associada a um sinal especificado em **signo**. Faz-se uma modificação se **act** for diferente de `NULL` e um exame se **oact** for diferente de `NULL`. O parâmetro **oact** é preenchido pelo serviço com as disposições actuais, enquanto que **act** contém as novas disposições.
O serviço retorna 0 se OK e -1 em caso de erro.

A definição da estrutura `sigaction` em `signal.h` é a que se apresenta a seguir (poderá conter mais campos):

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

O campo **sa_handler** contém o endereço de um handler (ou as constantes SIG_DFL ou SIG_IGN); o campo **sa_mask** contém uma máscara de sinais que são automaticamente bloqueados durante a execução do handler (juntamente com o sinal que desencadeou a execução do handler) unicamente se **sa_handler** contiver o endereço de uma função e não as constantes SIG_DFL ou SIG_IGN; o campo **sa_flags** poderá conter a especificação de comportamentos adicionais (que são dependentes da implementação - ver man).

No seguinte exemplo estabelece-se um handler para SIGINT (Control-C) usando o serviço sigaction():

```
    char msg[] = "Control - C pressed!\n";

    void catch_ctrl_c(int signo)
    {
        write(STDERR_FILENO, msg, strlen(msg));
    }
    ...
    struct sigaction act;
    ...
    act.sa_handler = catch_ctrl_c;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);
    ...
```

O serviço pause() permite bloquear um processo até que este receba um sinal qualquer. No entanto se se pretender bloquear o processo até que seja recebido um sinal específico, uma forma de o fazer seria usar uma *flag* que seria colocada em 1 no handler desse sinal (sem ser modificada noutros handlers) e escrever o seguinte código:

```
    int flag = 0;
    ...
    while (flag == 0)
        pause();
    ...
```

No entanto este código tem um problema se o sinal que se pretende esperar surgir depois do teste da *flag* e antes da chamada a pause(). Para obviar este problema a norma POSIX especifica o seguinte serviço:

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Quando retorna, retorna sempre o valor -1.

Este serviço põe em vigor a máscara especificada em **sigmask** e bloqueia o processo até este receber um sinal. Após a execução do handler e o retorno de sigsuspend() a máscara original é restaurada.

Exemplo:

Suponhamos que queríamos proteger uma região de código da ocorrência da combinação Control-C e, logo a seguir, esperar por uma dessas ocorrências. Poderíamos ser tentados a escrever o seguinte código:

```
sigset_t newmask, oldmask;
...
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

...     /* região protegida */

sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();
...
```

Este processo ficaria bloqueado se a ocorrência de Control-C aparecesse antes da chamada a `pause()`.

Uma versão correcta deste programa seria:

```
sigset_t newmask, oldmask;
...
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

...     /* região protegida */

sigsuspend(&oldmask);
sigprocmask(SIG_SETMASK, &oldmask, NULL);
...
```

Porque é necessário o último `sigprocmask()` ?

5.6 Interacção com outros serviços do Sistema Operativo

Dentro de um handler só é seguro chamar certas funções da API do S.O. Como os handlers podem ser chamados assincronamente em qualquer ponto do código de um processo os serviços aí chamados têm de ser reentrantes. A norma POSIX e cada fabricante de UNIX especificam quais são as funções que podem ser chamadas de forma segura do interior de um handler (essas funções dizem-se *async-signal safe*).

Certas funções da API do UNIX são classificadas como "*slow*" porque podem bloquear o processo que as chama durante intervalos de tempo longos (p. ex. operações de entrada/saída através de uma rede (*sockets*), através de *pipes*, abertura de um terminal ligado a um modem, certas operações de comunicação entre processos, etc). Esses serviços "*slow*" podem ser interrompidos quando da chegada de um sinal.
Quando isso sucede, e após a execução do handler do sinal que as interrompeu, essas funções retornam com a indicação de um erro específico (**`errno`** = `EINTR`). Para a correcta execução do programa que sofreu este tipo de interrupção é necessário tornar a chamar estes serviços. O código geralmente usado para isso, quando se chama um serviço "*slow*", é então:

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

O ciclo `while` torna a chamar `read()` (num *socket* (rede)) se este serviço tiver sido interrompido pela chegada de um sinal.

Cada versão do UNIX define quais são os seus serviços "*slow*". Algumas versões do UNIX permitem especificar no campo **`sa_flags`** da estrutura `sigaction` uma opção que torna a chamar automaticamente os serviços "*slow*" que tenham sido interrompidos por esse sinal. Quando existe, essa opção tem o nome de `SA_RESTART`.