8

Controlo de processos e sinais

Este capítulo aborda os processos. A criação e controlo de processos no Bash shell e pelo Linux API e sinais para processos e e seu tratamento:

Controlo simples

Cada processo tem um identificador único (pid - process id). O comando Unix **`ps`** permite ver os processos que estão a correr. Normalmente, um programa requer um só processo para correr. Mas, a execução dum programa pode lançar e envolver vários processos que comunicam entre si. Dái o conceito duma tarefa ou "Job". Existem processos especiais no sistema Unix. Por exemplo, o `pid 0` é normalmente o **scheduler process** (ou swapper) que faz parte do núcleo (kernel). Este processo não tem qualquer programa em disco. Em windows aparece como o "system idle process" ver em baixo,

Exemplo:

noe> **`ps -Af`**

PID	TTY	S	TIME	CMD
0	??	R<	10:36.27	[kernel idle]
1	??	I	00:01.57	/sbin/init –a
3	??	IW	00:00.03	/sbin/kloadsrv
21	??	S	01:32.49	/sbin/update
94	??	I	00:10.96	/usr/sbin/syslogd
350	??	I	00:43.24	sendmail:accepting connections
410	??	I	00:29.29	/usr/sbin/inetd
420	??	S	00:01.53	/usr/sbin/snmpd
445	??	I	00:00.34	/usr/sbin/cron
462	??	I	00:00.01	/usr/lbin/lpd

O `pid 1` é o processo **init** que é chamado pelo núcleo no fim do processo de arranque (boot). O processo **init** é responsável pela inicialização e configuração (os ficheiros `/etc/rc*`) do Unix. Este processo nunca morre. Não é um processo do sistema, mas sim um processo do utilizador. No entanto corre com os privilégios do superutilizador. O programa em disco deste processo é o ficheiro **/sbin/init**. Este processo nunca morre. Este processo torna-se progenitor de qualquer processo orfão.
Actualmente num sistema como fedora-21 ou ubuntu podem ver em vez do processo init outro como o systemd (system daemon) que serve o mesmo prepósito.

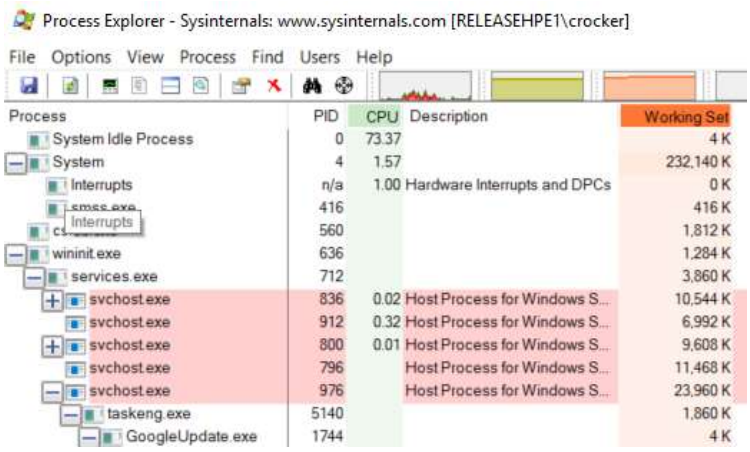

Figura: O explorador de Processos de Windows paar ver os processos e oe seus ID's

Alguns Artigos Interessantes:

- TheWindows idle process https://blog.codinghorror.com/why-is-the-system-idle-process-hogging-all-the-resources/
- Unix Swapper process history https://superuser.com/questions/377572/what-is-the-main-purpose-of-the-swapper-process-in-unix

JOBS

Em Linux existe o conceito da tarefa (*job*) que consiste em um ou mais processos (*process*). Cada processo contém uma ou mais linhas de execução (*Thread*)

- Existe um conjunto de comandos para o controlo de tarefas e processos com os comandos do sistema operativo disponíveis por linha de comando no Bash Shell. Os comandos mais relevantes são
  - ps listar processos. Um processo é identificado pelo PID (inteiro)
  - jobs listar tarefas. Uma tarefa é identificada pelo seu "job number" (%inteiro)
  - kill envie sinais para um processo ou tarefa (p.ex kill -9 1234 , kill %3)
  - bg, fg Executar tarefas de Background ou Foreground ou mover a sua execução dum plano para outro
  - & Execução em background

  - Outros Para Investigar: top, uptime, nice, nohup, at, atq etc

- O seu controlo e criação através das funções do API Linux
  - fork, wait, exit e kill.

- Para finalizar a última secção trate dos sinais (Signals) que nao é mais nada do que a comunicação de eventos aos processos e a seu tratamento seguinte.

Também deverá ler apontamentos adicionais aqui e "Process Control" do Linux Document Project.

Experimentar este comando (kill, kilall, fg %x, jobs etc, ps .)

Escrever e compilar o programa seguinte e depois executar e experimentar os comandos !

./ola 1; ctrl c
./ola 1 &
./ola 2 &
ps
jobs
fg %1

```
//ola.c
#include <stdio.h> <unistd.h>
int main (int argc, char *argv[])
{
  if (1==argc) return 0;
  while (1) {
      printf ("ola %s\n", argv[1]);
      sleep(3);
  }
  return 0;
}
```

Devolução de identificadores de processos

Há funções que devolvem os identificadores associados com um dado processo, nomeadamente:

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

pid_t getpid(void);                    Retorna: ID do processo invocador
pid_t getppid(void);                   Retorna: ID do progenitor do processo invocador
uid_t getuid(void);                    Retorna: ID do utilizador real do processo invocador
uid_t geteuid(void);                   Retorna: ID do utilizador efectivo do processo invocador
gid_t getgid(void);                    Retorna: ID do grupo real do processo invocador
gid_t getegid(void);                   Retorna: ID do grupo efectivo do processo invocador
```

Criação dum novo processo

A única maneira de criar um novo processo é através da função `fork()` (com a excepção dos processos especiais como o swapper e o init). Esta função `fork()` é invocada a partir dum processo já existente que é, pois, o processo progenitor. O novo processo gerado através da função `fork()` é chamado o processo progénito.

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

pid_t fork(void);
Retorna: 0 a partir do progénito, ID do progénito a partir do progenitor, -1 em caso de erro
```

Note-se que a função fork é chamada uma vez, mas devolve duas vezes: uma a partir do processo progénito, outra a partir do processo progenitor.
O programa seguinte mostra a utilização da função `fork()` com duas cópias do programa a correr simultaneamente (multitasking).

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

main()
{
        int  pid;
        printf("Processo progenitor pid=%d\n", getpid());
        printf("Bifurcando o processo\n" );
        pid=fork();
        /* As instruções que seguem são executadas duas vezes:
           uma vez para o progenitor, outra vez para o progénito*/
        printf( "O processo progenitor id %d e o seu progenito %d\n", getpid(), pid);
        system("date");
}
```

Exemplo 8.1.

Exercício 8.1:

- Escreva e execute o programa anterior. Qual é o processo que executa primeiro, o progenitor ou o progénito?
  - Normalmente, nunca se sabe qual é o processo que começa primeiro, se é o progenitor ou o progénito, pois isso depende do algoritmo de escalonamento da CPU.
- A seguir modifique o programa e introduzir uma estrutura de controlo, como se segue, para gerar o output indicado em baixo.

```
int pid;                        /* identificador do processo*/

if ( (pid=fork()) < 0) {
        printf("Cannot fork\n");
        exit(0);
}
if (pid == 0)
   printf("Filho com ID %d\n",getpid());
else
  /* pid do processo progenitor é pid do progénito …*/

system("date");
```

Ouput Esperado

```
Pai com processo ID 12345 e filho com ID 123456
Filho com ID 123456
Thu Apr 15 16:45:20 WEST 2008
Thu Apr 15 16:45:20 WEST 2008
```

A função de Sincronização wait()

Existe a necessidade de sincronizar as operações dos processos progenitor e progénito. Uma das funções de sincronização disponível é a função wait(). Normalmente esta função é usado pelo processo progenitor esperar a terminação de um dos seus progénitos.

- `int wait(int *status);` Força o progenitor a esperar pela paragem ou terminação do progénito. Esta função devolve o pid do progénito ou –1 no caso de erro. O estado de exit do progénito é devolvido para a status.
- `void exit(int status);` Termina o processo que chama esta função e devolve o valor de status.
- A função `waitpid(pid_t pid, int *wstatus, int options)` espera a terminação dum progénito especifico.

  A função wait() é as vezes usado em conjunto com um ciclo caso que uma chamada a função wait não seja bloquenate

  ```
  while ( wait(&status_filho) != pid_filho ) ;
  ```

Um exemplo da necessidade de sincronização é quando a entrada e saída estandardizadas do processo progenitor são redireccionadas, então o mesmo acontece com entrada e saída estandardizadas do processo progénito.
Além disso, todos os descritores dos ficheiros abertos pelo progenitor serão duplicados pelo progénito e também os conteúdos de quaisquer buffers das bibliotecas standard. Consequentemente, pode haver problemas de sincronização na entrada e na saída e output difícil de explicar.

Considere o seguinte exemplo

Exemplo 8.1b:

```
main()
{
   printf ("Bom Dia  ");
   if ( 0 == fork() )
       printf("disse o Filho ");
  else
       printf("ao Pai.\n");
}
```

O Output do programa Exemplo 8.1b é

Bom Dia ao Pai.
Bom Dia disse o Filho

Como é que podemos obter sempre o output

Bom Dia disse o filho ao Pai.

Exercício 8.1b : Escreva um programa para produzir sempre o output desejado.
Solução: utilizar a função fflush() e wait() para sincronização

Exercício 8.2:

Escreva um programa que inicialize uma variável `int v=5` e depois abre um ficheiro "res.txt" em modo escrita. Antes da bifurcação, escreva para o ficheiro res.txt o valor da variável v.
Depois da bifurcação, altere o seu valor para 10 no progenitor (pai) e para 15 no progénito (filho).
Depois, escreva estes valores conjuntamente com os valores dos PID's do progenitor e do progénito para este mesmo ficheiro. Os valores do progénito (filho) devem ser escritos em primeiro lugar – precisará novamente da função wait !

Ouput Esperado: cat res.txt

V = 5 Pai com processo ID 12345
V = 15 filho Processo com ID 54321
V = 10 Pai Processo com ID 12345

Notas: Situações de utilização da função fork():

- *Quando um processo pretende replicar-se*. Por exemplo, um servidor quando recebe um pedido dum cliente faz um fork para que o progénito trate do pedido enquanto o servidor continua a correr e à espera doutros pedidos.
- *Quando um processo pretende executar outro programa*. Por exemplo, dentro dum programa shell. Neste caso, o progénito faz um `exec()` logo a seguir ao `fork()`. Nalguns sistemas operativos, estas duas chamadas (fork e exec) são combinadas numa única operação chamada `spawn()`, por exemplo em Windows.

Funções exec

Um processo pode carregar um novo programa usando a chamada ao sistema exec(). Quando um processo chama uma das funções da família `exec`, o programa que começa a executar fá-lo substituindo o programa original pelo novo. O pid não muda com um `exec` porque não há a criação dum processo novo. Basicamente, `exec` faz a substituição do contexto do processo atual, ou seja, texto, dados, heap, stack, etc.) pelo novo programa, embora algumas características sejam herdadas (por exemplo, pid, ppid, user id, root directory, etc.).

Existem seis funções exec, mas aqui só vamos referir uma delas:

- `int execl(const char *pathname, const char *arg0, …);`
- Significa *execute* and *leave*, i.e. executa e termina o comando indicado pelo pathname com as opções indicadas pelos restantes argumentos da função.
- Para mais informações sobre a família exec ver `man 2 exec`

Exemplo 8.2:

```
main()
{
        printf("processo main %d\n", getpid());
        printf("Ficheiros na directoria:\n" );
        execl( "/bin/ls", "ls", "-l", 0 );
}
```

Consultar o man page para a função execvp e execl

Exercicios (Exame do 2017, Freq 2015)

Especificar qual é o output dos seguinte programas? O output é determinístico? Justifique! Deverá mostrar o *trace* do funcionamento do programa. A seguir deve implementar e executar o programa.

```
char *args[2] = {"date", NULL};
int status, pid, x = 9;
pid = fork();
if (0 == pid) {
    x++;
    pid = fork();
    if (0 == pid)
        execvp(*args, args);
    else
        wait(&status);
}
else
    x--;
printf("x=%d\n", x);
```

```
int pid, x = 2;
if  ( (pid = fork()) == 0){
   fork();
   fork();
   x--;
}
else{
   execl("/bin/date", "date", NULL);
   x = x + 2;
}
printf("x=%d\n", x);
```

Execução de comandos Unix a partir dum programa em linguagem C

Podemos executar comandos Unix a partir dum programa em C através do uso da função `system()` que se encontra declarada no ficheiro <`stdlib.h`>.

- `int system(char *string);` `string contém o nome do comando.`

Exemplo 8.3:

```
main()
{
        int res;
        printf("Ficheiros na directoria: " );
        res = system("ls -l" );
        printf( "%s\n", (0==res)? "sucesso": "insucesso");

        printf("Ficheiro VidaNova.txt na directoria: ");
        res = system( "ls –l VidaNova.txt" );
        printf("%s\n", (0==res)? "sucesso": "insucesso");
}
```

Nota: a função `system()` é uma chamada ao sistema composta por outras três chamadas ao sistema: `execl()`, `wait()` e `fork()` (veja-se o ficheiro <`unistd.h`>).

Um Shell Simples (soshell)

Exemplo 8.4 Um Interpretador de Comandos

Este exemplo serve de base à programação duma shell limitada.

```
soshell  - example of a fork in a program used as a simple shell.
  The program asks for Unix commands to be typed and are read into a string.
  The string is then "parsed" by locating blanks, etc.
  Each command and corresponding arguments are put in a args array.
 execvp is called to execute these commands in child process spawned by fork().
```

Os ficheiros deste projecto são :

- shell.h – ficheiro com os protótipos e ficheiros de inclusão do programa.
- main.c – ficheiro com o programa principal.
- parse.c – ficheiro com a função que particiona o comando Unix.
- execute.c – ficheiro com a funçao que cria um processo progénito e executa um programa.
- Makefile – ficheiro de gerencia de compilação do projecto.
- soshell – ficheiro executável criado pelo Makefile.

```
#A Simple Ilustrative Makefile for soshell
#
CC=cc
FLAGS=-c -Wall
LIBS=-lm
OBS=main.o execute.o parse.o

all :  soshell

main.o : shell.h main.c
	$(CC) $(FLAGS) main.c
execute.o : shell.h execute.c
	$(CC) $(FLAGS) execute.c
parse.o : shell.h parse.c
	$(CC) $(FLAGS) parse.c
soshell : $(OBS)
	$(CC)  -o soshell  $(OBS) $(LIBS)
clean limpar:
	rm -f soshell *.o *~
```

```
/*
   shell.h – ficheiros de inclusao e prototipos.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>

void parse(char *buf, char **args);
void execute(char **args);

int builtin (char **args);

/* constantes que podem tornar uteis*/
#define BG 0
#define FG 1
```

```
#include "shell.h"
/*
   main.c – cria um processo progénito e executa um programa
*/
char prompt[100];
int main()
{
        char buf[1024];     /* um comando */
        char *args[64];    /* com um maximo de 64 argumentos */
        strcpy (prompt, "SOSHELL: Introduza um comando: prompt>"); /*prompt inicial*/

        while (1)
        {
             printf( "%s", prompt );
             if ( gets(buf) == NULL)
             {
                printf( "\n" );
                exit(0);
             }

        parse(buf,args);  /* particiona a string em argumentos */

        if ( !builtin(args) )
           execute(args);    /* executa o commando se não for uma funcionalidade embutida */
        }
        return 0;
}

int builtin( char *args[] ) /* verifiicar se o comando é embutido no shell */
{
   if (strcmp (args[0], "sair") == 0)
    {
      exit (0);
      return 1;    /* funcionalidade embutida */
    }
   if (strncmp (args[0], "42", 2) == 0)
    {
      printf("42 is the answer to life the universe and everything\n");
      return 1;    /* funcionalidade embutida*/
    }

   return (0) ;   ;    /* indicar que vamos continuar para a função execute*/
}
```

```
#include "shell.h"
/*
   parse.c – particiona o comando Unix (armazenado em buf) em argumentos
*/

void parse (char *buf, char **args)
{
        while (*buf != '\0')
        {
        /* strip whitespace. Usa um NULL para indicar que o argumento anterior e' o ultimo */
            while ((*buf == ' ' ) || (*buf == '\t'))
                    *buf++ = '\0';

            *args++ = buf;          /* salvaguarda argumento */

             while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))   /* salta sobre o argumento*/
                     buf++;
        }
       *args = NULL;             /* o ultimo argumento e' NULL */
}
```

Nota: Pode-se usar a função isspace() para detectar espaço em branco – o chamado *whitespace*

```
#include "shell.h"
/*
   execute.c – cria um processo progénito e executa um programa
*/

void execute(char **args)
{
        int pid, status;

        if ((pid = fork()) < 0)                                          /* cria um processo progenito */
        {
           perror("fork");    /* NOTE: perror() produz uma pequema mensagem de erro para o stream */
           exit(1);                        /* estandardizado de erros que descreve o ultimo erro encontrado */
                                           /* durante uma chamada ao sistema ou funcao duma biblioteca */
        }
        if (pid == 0)
        {
           execvp(*args, args);    /* NOTE: as versoes execv() e execvp() de execl() sao uteis quando */
           perror(*args);              /* o numero de argumentos nao e' conhecido. Os argumentos de  */
           exit(1);                   /* execv() e execvp() sao o nome do ficheiro a ser executado e um */
        }                                                    /* vector de strings que contem os argumentos. */
                                  /* O ultimo argumento do string tem de ser seguido por um ponteiro 0. /*

        /* Execução em foreground */
        while (wait(&status) != pid)                                  /* O progenitor executa a espera */
           /* ciclio vazio repare que o ; é importante */ ;
}
```

Exercícios :

1 **Implemente e Estude o Exemplo**

O código está num servidor seguro de subversion que poderá ser acedido usando um browser com o endereço seguinte : https://spocs.it.ubi.pt:8443/svn/crocker/sisops/

Utilizando o programa "svn" (linha do comando) poderá usar o seguinte comando para obter os ficheiros:
svn --username aluno checkout https://spocs.it.ubi.pt:8443/svn/crocker/sisops/trunk/soshell

Verifique o código do main() para ver qualquer alteração desta ficha e o código do svn
(a função 42 não está implementada e o main utilize agora fgets() em vez do gets())

2 Aumente a funcionalidade do seu Shell

(i) implemente a possibilidade de mudar dinamicamente (durante runtime) o prompt usando o comando prompt> PS1=String : onde String será o novo prompt

(ii) implemente um comando embutido chamado "quemsoueu" que mostre detalhes sobre a identificação do usuário. Baste chamar a função system("id") na função "builtin".

(iii) implemente o comando embutido "socp" que permite o shell copiar um ficheiro através do comando prompt> socpy destino fonte

Detalhes – escreva uma função void socp(char *fonte, char *destino) num ficheiro socp.c e chame-a dentro da função builtin do programa principal. Esta função deverá utilizar a função solowlevelcopy() das fichas anteriores. Também terá que modificar o Makefile.

(iv) implemente controlo de processos com a possibilidade de execução em foreground ou em background (adicionado o símbolo & no fim do comando)

(v) implemente redireccionamento do stdin e stdout (<) e (>)

Sinais (signals)

O que é um sinal?

Sinal é um evento que pode ocorrer enquanto um programa está a correr.

Podemos distinguir dois tipos de sinais: interrupções e excepções.

- As interrupções são geralmente geradas pelo programador, como um CTRL-C (terminar), por exemplo ou com um CTRL-Z (suspender)
- As excepções são erros que ocorrem durante a execução de programas, como um "*overflow*" ou um ponteiro "*out-of-range*", por exemplo.

A ocorrência dum sinal despoleta uma acção que lhe foi atribuída previamente, por defeito.
Por exemplo, a acção associada a CTRL-C é terminar o programa.

Alguns sinais mais comuns em Linux são:

Número	Sinal	Acção por defeito	Significado
1	SIGHUP	Exit	Perdeu a ligação com o terminal (*hangup*)
2	SIGINT	Exit	Interrupção (CTRL-C) na Shell
3	SIGQUIT	Core Dumped	*Quit*
4	SIGKILL	Core Dumped	Sinal ilegal
5	SIGTRAP	Core Dumped	Interrupção de *trace* (usado por *debbugers* como o dbx)
8	SIGFPE	Core Dumped	*Floating Pointing Exception*
9	SIGKILL	Exit	*Terminate execution* (não pode ser ignorado)
10	SIGBUS	Core Dumped	Erro no *bus* (violação da protecção de memória)
11	SIGSEGV	Core Dumped	Violação da segmentação de memória
14	SIGALARM	Exit	Alarme do relógio – *time out*
15	SIGTERM	Exit	Termina a execução (não pode ser ignorado)
18	SIGCONT	Continue	Iniciar um processo parado
19	SIGSTP	Stop Job	Stop signal (do processo)
20	SIGTSTP	Stop Job	Stop signal (do teclado)

Figura 8.5: Sinais em Unix.

Na ocorrência dum "core dump", o conteúdo de um programa (código, variáveis e estado) são colocados num ficheiro chamado **core** antes do programa terminar.

Todos estes sinais estão definidos no ficheiro `signal.h`

Rotinas de Gestão (*Handler Routines*)

As "handler routines" são chamadas pelo programa ou ambiente de execução em reaçao a um sinal como um error ou interrupt. Um sinal faz com que o program seja suspense e uma "handler routine" seja imediatamente executado. Quando uma "handler routine" termina e retorna, a execução do programa é retomada onde o sinal tinha ocorrido menso que seja chamado longjump() para retomar execução do programa num outro ponto.

Utilização de Sinais

A função **`signal`** especifica a "handler routine" a ser executada quando um certo sinal ocorre. Esta função permite também re-definir a acção ou a "handler routine" quando um sinal ocorre. Além do mais, esta função pode ser usada para ignorar sinais. A sua sintaxe é a seguinte:

```
#include <signal.h>
typedef void (*sighandler_t)(int);    //Uma extensão de GNU
sighandler_t signal(int signum, sighandler_t handler);

void (*signal(int signum, void (*func)(int)))(int);  //sem extensão de GNU
```

Retorna: ponteiro para o "signal handler" anterior

O argumento *signum* é o numero dum dos sinais listados na Figura 8.5. O valor de *func* é um dos seguintes:

- a constante SIG_IGN;
- a constante SIG_DFL;
- o endereço duma função (ou "handler routine") que é chamada quando o sinal ocorre.

Se a constante SIG_IGN é especificada na função signal, isso quer dizer que estamos a dizer ao sistema para ignorar o sinal. (Há, no entanto, dois sinais, SIGKILL e SIGSTOP, que não podem ser ignorados.). A constante SIG_DFL é usada quando se pretende que a acção associada com o sinal seja a acção por defeito.

Hibernação dum processo durante um período de tempo (segundos)

A função sleep pára a execução do programa durante um determinado número de segundos. A sua sintaxe é a seguinte:

```
#include <unistd.h>

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

Retorna: 0 se o período se esgotou; tempo que falta para esgotar o período *seconds*

Esta chamada usa o alarme de interrupção SIGALARM para atrasar o programa. Note-se que ao fim do período *seconds* não há a garantia que o programa retome imediatamente a sua execução, pois isso depende da política de escalonamento da fila dos processos prontos-a-correr ("ready queue").

Existe uma função semelhante à função sleep, designada por usleep. Funciona da mesma forma que o sleep, excepto que o período de hibernação ou de espera é em microsegundos e não em segundos. A sua sintaxe é a seguinte:

```
#include <unistd.h>

void usleep(unsigned long usec);
```

Retorna: nada.

Ignorar e restaurar acções por defeito O Exemplo 9.2 mostra como ignorar um sinal e restaurar a acção por defeito que lhe está associada. Usa ainda a função `sleep`, fazendo com que o processo espere um determinado período em segundos. A seguir restaure a acçao por defeito (SIG_DFL)

Exemplo 8.6:

```
#include <stdio.h> <signal.h>
main()
{
   signal(SIGINT, SIG_IGN);  //vamos ignorar o sinal SIGINT

   for(int i=0; i<15; i++) {
      printf("Nao pode usar CTRL-C para terminar. Experimente !\n");
      sleep(1);
   }
   signal(SIGINT, SIG_DFL);
   printf("\nAgora pode Primar CTRL-C para terminar...\n");
   sleep(10);
}
```

Ignorando o sinal `SIGINT` (CTRL-C), o programa não pode ser interrompido durante 15 segundos. Depois dos 15 segundos, é restaurado a "handler routine" por defeito e o CTRL-C já pode interromper o programa.

Criar "handler routines" As "handler routines" são funções que têm as seguintes restrições:

1. Têm que ser declaradas no código antes de serem referenciadas.
2. Têm prototipo. void confirma(int signo) ; onde signo é o numero de sinal enviado.

O Exemplo 9.3 mostra como se pode resolver o problema acidental do utilizador premir CTRL-C, o que provoca a terminação do programa. A "handler routine" `confirma` associada ao sinal `SIGINT` (CTRL-C) pede ao utilizador para confirmar ou não o fim do programa.

Exemplo 8.7:

```
#include <stdio.h> <stdlib.h> <signal.h> <unistd.h> <ctype.h>
void confirma(int signo);
int main()
{
   signal(SIGINT, confirma);
   for (int i=1; i<20; i++) {
       printf("Estamos no ciclo numero press ctrl-c%d.\n", i);
       sleep(1);
   }
   return 0;
}
void confirma(int signo)
{
   char sim_ou_nao, enter;

   printf("\nQuer mesmo terminar? (S/N)");
   scanf("%c%c", &sim_ou_nao, &enter);

   printf("%c\n", sim_ou_nao);
   if ( 's' == (tolower(sim_ou_nao))) exit(0);
}
```

Abate, Terminação ou Controlo dum processo

A função `raise` permite função que um processo possa enviar um sinal a si proprio.
A função `kill` permite que um processo possa enviar um sinal de abate ou terminação a outro processo. Enviando um sinal `SIGTERM` ou `SIGKILL` para outro processo faz com que este termine, desde que:

1. Os processos pertençam ao mesmo utilizador, ou o processo emissor de signal pertença ao super-utilizador root.
2. O processo saiba o identificador (pid) do processo a matar.

A sintaxe da função `kill` é a seguinte:

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

int kill(pid_t pid, int sig);
```

Retorna: 0 se OK; -1 em caso de erro

Exemplos do sinal *sig*

`SIGTERM` (pedido para terminar, permite um processo chamar os seus signal handlers)
`SIGKILL` (força o processo a terminar).
`SIGCONT` (força o processo a continuar).
`SIGTSTP` (Signal Terminal Stop) suspends the process.
`SIGSTOP` (Signal Stop). suspends the process programatically (via kill)

Permite a implementação do "Job/Process Control" num interpretador de comandos

Programação de alarmes para emissão de sinais

Todas as versões UNIX têm um relógio de alarmes. O relógio pode ser programado de tal modo que uma determinada "handler routine" pode ser executada quando determinada hora/instante chega.
O sinal `SIGALARM` é enviado quando um alarme de relógio é gerado, fazendo com que uma rotina "handler" seja executada.
A programação temporal dum alarme é feita com a função `alarm`, a qual tem a seguinte sintaxe:

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Um processo não pára a sua execução quando faz uma chamada de alarme ao sistema; ele retorna imediatamente da chamada sem esperar e continua a sua execução.

Nota: a função `sleep` faz o *reset* do relógio de alarmes. Assim, é muito importante não usar a função `sleep` se também se pretende usar um alarme.

Exemplo 8.8:

O programa seguinte usa a função `sleep` e a função `alarm` sem quaisquer problemas. Tudo funciona correctamente porque o processo filho usa a função `sleep`, enquanto o processo pai usa a função `alarm`, separadamente.
O processo filho corre um ciclo infinito.
O processo pai envia um alarme para parar cinco segundos e espera que o processo filho termine. Depois dos cinco segundos, e se o processo filho ainda não terminou, o pai mata o filho com a função `kill`.

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

int pid;

void myalarm(int signo)
{
   kill(pid, SIGKILL);
   printf("Adeus filho!\n");
}

int main()
{
   int status;
   signal(SIGALRM, myalarm);

   pid = fork();
   if (-1 == pid)
   {
       perror("Erro no fork.");
       exit(1);
   }

   if (0 == pid)
      for(;;)
         {
         printf("Sou um processo filho - louco e livre!!\n");
         sleep(1);
         }
   else
   {
      alarm(5);
      wait(&status);
   }
}
```

Refs
https://www.geeksforgeeks.org/signals-c-language/