6

Controlo de processos

Este capítulo aborda os processos, o seu controlo com os comandos do sistema operativo disponíveis, o seu controlo e criaçao atraves das funçoes do API fork, wait, exit e kill.

Para finalizar a útima secção trate dos sinais (Signals) que nao é mais nada do que a comunicação de eventaos aos processos e a sua tratamento seguinte.

Também deverá ler os apontamentos adicionais "process Control" tirados do Linux Document Project. De acordo com o espirito de Bolonha deverá ler estes apontamentos antes da aula prática.

Controlo simples

O que é um processo?

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 envolver vários processos que comunicam entre si.

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.

Exemplo:

```
noe> ps -ac
```

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
96	??	I	00:00.01	/usr/sbin/binlogd
280	??	I	00:00.09	/usr/sbin/portmap
282	??	I	00:00.01	/usr/sbin/mount –i –n –d –n
284	??	I	00:00.00	/usr/sbin/nfsd –t8 –u8
286	??	I	00:00.00	/usr/sbin/nfsiod 7
288	??	IW	00:00.00	/usr/sbin/rpc.pcnfsd
291	??	I+	00:00.01	/usr/sbin/rpc.statd
293	??	IW	00:00.01	/usr/sbin/rpc.lockd
350	??	I	00:43.24	sendmail:accepting connections o
407	??	S	00:00.16	/usr/sbin/svrSystem_mib
408	??	S	00:00.25	/usr/sbin/svrMgt_mib
409	??	I	00:00.76	/usr/sbin/os_mibs
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.

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 6.1

Exercício 6.1:
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 introduzir uma estrutura de controlo, como se segue:

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

pid = fork();
if (pid < 0)
{
        printf("Cannot fork\n");
        exit();
}
if (pid == 0)
{/* Processo progenito …*/}
else
{/* pid do processo progenitor é pid do progénito …*/}
```

Ouput Esperado

```
Pai com processo ID 12345 e filho com ID 123456
Filho com ID 123456
Thu Apr 15 16:45:20 WEST 2004
Thu Apr 15 16:45:20 WEST 2004
```

Partilha de ficheiros

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. Consequentemente, pode haver problemas de sincronização na entrada e na saída.

Para evitar estes problemas, podemos usar as seguintes funções:

- `int wait(int status_location);` 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 a status_location.
- `void exit(int status);` Termina o processo que chama esta função e devolve o estado de exit.

```
A função wait é tipicamente usado nem conjunto com um ciclo:
      while ( wait(&status_filho) != pid_filho ) ;
```

Exercício 6.2:
Escreva um programa que contenha uma variável v=5. Antes da bifurcação, escreva para o ficheiro res.txt o seu valor. 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 identificadores do progenitor e do progénito para o mesmo ficheiro. Os valores do progénito (filho) devem ser escritos em primeiro lugar – precisará da função wait !

Ouput Esperado : cat res.txt

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

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()`.

Funções exec

Quando um processo chama uma das funções da família `exec`, o programa que começa a executar fá-lo em detrimento do seu progenitor. O pid não muda com um `exec` porque não há a criação dum processo novo. Basicamente, `exec` faz a substituição do processo (do seu contexto, 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 6.2:

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

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 6.3:

```
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

Exemplo 6.4 (controlo completo de processos):

Este exemplo pode servir de base à programação duma shell limitada.

rshell.c - example of a fork in a program used as a simple shell.
The program asks for Unix commands to be typed and inputted to a string.
The string is then "parsed" by locating blank, 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()`.

Nota: O Seu Professor insiriu alguns erros e avisos .. deverá corrigir o programa !

Este programa deverá ser feito de seguinte maneira com compilaçao seperada dentro dum ficheiro bash simples ou melhor um Makefile.

rshell.c – ficheiro com o programa principal
rshell.h – ficheiro com os protitpos e ficheiros de inclussoa do programa
parse.c – ficheiro com a funçao que particiona o comando Unix
execute.c – ficheiro com a funçao que cria um processo progénito e executa um programa

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

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

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

```
/*
  rshell.c – cria um processo progénito e executa um programa
*/
main()
{
        char buf[1024];     /* um comando */
        char args[64];      /* com um maximo de 64 argumentos */
        for (;;)
        {
             printf( "Introduza um comando Unix: " );
             if ( gets(buf) == NULL)
             {
                printf( "\n" );
                exit(0);
             }

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

        execute(args);    /* executa o comando */
        }
}
```

```
/*
   parse.c – particiona o comando Unix (armazenado em buf) em argumentos
*/

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

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

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

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`>).

```
/*
   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 */
                                                        /* string tem de ser seguido por um ponteiro 0. /*
        while (wait(&status) != pid)                                     /* O progenitor executa a espera */
        /* empty */ ;
}
```

Sinais (signals)

O que é um sinal?

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

Há 2 tipos de sinais: interrupções e excepções.

- As interrupções são geralmente geradas pelo programador, como um CTRL-C, por exemplo.
- 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.

Os tipos de sinais mais comuns em UNIX 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	SIGFTP	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)
17	SIGSTP	Stop Job	Stop signal (do processo)
18	SIGTSTP	Stop Job	Stop signal (do teclado)

Figura 9.4: 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 quando ocorrem erros. Um sinal faz com que uma "handler routine" seja imediatamente executado. Quando uma "handler routine" termina, a execução do programa é retomada onde o erro tinha ocorrido.

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>

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

Retorna: ponteiro para o "signal handler" anterior

O argumento *signo* é o nome dum dos sinais listados na Figura 9.4. 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 de tempo em segundos. A seguir restaure a acçao por defeito (SIG_DFL)

Exemplo 9.2:

```
#include <stdio.h> <signal.h>
main()
{
   int i;

   signal(SIGINT, SIG_IGN);  //vamos ignorar o sinal SIGINT

   for(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. Não podem ter parâmetros;
2. Têm que ser declaradas no código antes de serem referenciadas.

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 9.3:

```
#include <stdio.h> <stdlib.h> <signal.h>
main() {
   int i;
   signal(SIGINT, confirma);

   for (i=1; i<20; i++) {
       printf("Estamos no ciclo numero press ctrl-c%d.\n", i);
       sleep(1);
   }
}
int confirma() {
   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 ((sim_ou_nao == 'S') || (sim_ou_nao == 's')) exit();
}
```

Abate ou terminação dum processo

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

Normalmente, o sinal *sig* é `SIGTERM` (pedido para terminar) ou `SIGKILL` (força o processo a terminar).

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 9.4:

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>

int pid;

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

main()
{
   union wait status;

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

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