EST

ESCOLA SUPERIOR DE TECNOLOGIA

Instituto Politécnico de Castelo Branco

Ficha de Trabalho nº 4

Sistemas de Exploração

Processos em Linux

2º Ano / 1º Semestre – 2005/2006

1. Objectivos

- Familiarização com a criação de processos, conceitos de processo Pai e processo Filho.
- Desenvolvimento de programas de criação e execução de processos em ambiente UNIX (Linux).

2. Processos em Linux

2.1. O que é um processo?

1. Execute o comando **ps**. O que faz? (***man ps***)

2. Execute agora o mesmo comando com as opções **aux**. O que mostra?

3. Codifique o seguinte código (crie um directório **ficha4** na directoria de trabalho, e aí dentro o ficheiro **ex31.c**):

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

int main(int argc,char *argv[]) {
   printf("SLEEP...\n");
   sleep(30);
   printf("THE END\n");
   return 0;
}
```

   a) Compile e execute o binário resultante em background.

```
prompt $ <ficheiro_exe>&
```

   b) Execute o mesmo executável 3 vezes, do mesmo modo. Agora execute o comando **ps**.

4. De seguida são apresentados os comandos para controlar a execução dos processos:

   a) **jobs** – ver o estado os *jobs* que estão em execução e mostra o seu estado.

   b) **fg** – passa um processo suspenso para *foreground*.

   c) **bg** – passa um processo suspenso para *background*.

5. Execute as seguintes instruções numa linha de comandos:

   a) Exercício 1:

```
./ex31
[ctrl^D]
jobs (verificar o número do job n e o estado do job)
bg n
jobs (verificar o estado do job)
fg n
ctrl^D
jobs
bg n
```

   b) Exercício 2:

```
./ex31 &
jobs (verificar o número do job n e o estado do job)
fg n
[ctrl^D]
jobs
bg n
```

2.2. Obter o PID de um processo

1. *Process IDentifier* (PID). O que é? Como é que um processo obtém o seu PID?

   a) Consulte o manual para obter o PID (***man getpid***).

b) Acrescente, antes da chamada ao ***sleep***, uma linha em que imprime o PID do processo em execução (ficheiro **ex32.c**):

```
...
printf("Eu sou o processo %d\n", ????);
...
```

c) Compile e execute o binário resultante em background 3 vezes. Agora execute o comando **ps**.

2.3. Criar novos processos

1. A função ***fork*** (*system call*) cria um novo processo (designado processo filho) que é uma cópia do processo que o criou (processo pai). Consulte o manual desta função (***man fork***).

```
int fork()
```

a) Esta chamada ao sistema devolve um valor inteiro:

i) **0** ao **processo filho**,

ii) O **PID do processo filho** ao **processo pai**,

b) Em caso de erro esta chamada ao sistema devolve:

i) **-1** ao processo pai em caso de **erro** (não conseguiu criar o processo).

2. Qual é o *output* do seguinte programa?

a) Copie, compile e execute (ficheiro **ex33.c**).

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

int main(int argc, char* argv[])
{
   pid t pid = fork();

   if (pid == 0)
   {
        printf("Eu sou o processo filho.\n");
        sleep(20);
   }
   else if (pid > 0)
   {
        printf("Eu sou o processo pai.\n");
        sleep(20);
   }
   else
   {
        printf("Fork error %d.\n", errno);
   }
   return 0;
}
```

2.4. Obter o PID do processo pai

1. Altere o programa anterior para que cada processo imprima o seu PID e o PID do processo que o criou, o processo pai (ficheiro **ex34a.c**). Consulte o manual (***man getppid***).

```
...
printf("[%d] Eu sou o processo xpto. O meu pai é o processo %d\n", getpid(), ????);
...
```

   a) Compile e execute. Execute o comando **ps**. Quem é o pai do "processo pai"?

2. Altere, novamente, o programa para que o processo pai mostre o PID do processo filho (ficheiro **ex34b.c**).

   a) Compile e execute.

2.5. Esperar que um processo termine a sua execução

1. Um processo termina a sua execução normal quando existe uma chamada ao sistema ***exit*** (***man 3 exit***) ou o retorno da função *main* (***return***) que indique que o processo terminou. Normalmente o valor inteiro 0 (zero) indica ao processo pai que o processo terminou sem erros. Os restantes valores inteiros permitem indicar erros específicos.

```
void exit(int status);
```

2. As funções ***wait*** e ***waitpid*** (*system call*) bloqueiam um processo até que um processo filho termine a sua execução. Consulte o manual do ***wait*** e ***waitpid*** (***man 2 wait*** e ***man 2 waitpid***).

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Macros para determinar a forma como o processo filho terminou:

```
WIFEXITED(status)
WEXITSTATUS(status)
WIFSIGNALED(status)
WTERMSIG(status)
WIFSTOPPED(status)
WSTOPSIG(status)
```

3. Compile e execute o seguinte código (ficheiro **ex35a.c**):

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

int main(int argc,char *argv[]) {
        int status=0;
```

```
        pid t pid = fork();
        if (pid==0) {
            exit(10);
        } else if (pid > 0) {
            waitpid(pid,&status,0);
            printf("O processo filho terminou com estado %d.\n",WEXITSTATUS(status));
        } else {
            printf("Fork error %d.\n",errno);
        }
        return 0;
}
```

4. Implemente um programa (ficheiro **ex35b.c**) em que o processo pai cria um processo, imprime a *mensagem de apresentação*, espera que o processo filho termine a sua execução e imprime uma mensagem em que indica que o processo filho terminou (com o PID do filho). O processo filho deverá imprimir uma linha de *apresentação* ("[PID] Eu sou o filho. O meu pai é o processo PPID") e *dormir* 15 segundos.

   a) Compile e execute.

2.6. Processos Zombies e Defuncts

Os processos zombie são processos mortos. Não é possível matar o que já está morto. Todos os processos morrem, tornando-se zombie. Os processos zombie, praticamente, não consomem recursos.

A razão da existência de processos zombie deve-se ao facto do processo pai destes processos poder ir buscar o resultado de saída e estatísticas de utilização de recursos dos processos zombie. O processo pai indica ao sistema operativo que não necessita mais de um processo zombie utilizando uma chamada ao sistema wait.

Quando um processo morre, os seus processos filho tornam-se filhos do processo número um (1), que corresponde ao processo inicial. O processo inicial está sempre à espera que os filhos morram. Assim, estes processos não permanecem como zombies.

A existência de processos zombie significa que o processo pai não ficou à espera que estes processos terminassem. Para resolver a situação pode-se:

- Recodificar o processo pai, de modo a que este aguarde a conclusão dos processos filho;
- Matar o processo pai.

Um processo defunct também é conhecido como zombie, indicando que um processo terminou, mas continua dependente do processo pai que continua "vivo".

1. Leia o seguinte código.

   a) Repare nos valores dos parâmetros das chamadas ***sleep***. Qual é a ordem temporal dos acontecimentos (criação e terminação de processos)?

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

int main(int argc, char* argv[])
{
   pid_t pid = fork();
   if (pid == 0)
   {
        printf("[%d] Eu sou o processo filho. O meu pai é o %d.\n", getpid(), getppid());
        sleep(15);
   }
   else if (pid > 0)
   {
        int status;

        printf("[%d] Eu sou o processo pai.\n", getpid());
        sleep(30);
        printf("O meu filho %d terminou.", wait(&status));fflush(stdout);
        sleep(15);
   }
   else
   {
        printf("Fork error %d.\n", errno);
   }
}
```

   b) Copie o código e compile-o (ficheiro **ex36a.c**). Abra duas consolas antes de executar o binário. Execute-o numa consola e execute o comando **ps –a** na outra. Vá repetindo o comando **ps –a**.

      i) O que acontece ao fim de 15 segundos? E ao fim de 30? E ao fim de 45?

2. Leia o seguinte código.

   a) Repare nos valores dos parâmetros das chamadas ***sleep***. Qual é a ordem temporal dos acontecimentos (criação e terminação de processos)?

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

int main(int argc, char* argv[])
{
   pid t pid = fork();
   if (pid == 0)
   {
     while(1){
       printf("[%d] Eu sou o processo filho. O meu pai é o %d.\n", getpid(), getppid());
       sleep(1);
     }
   }
   else if (pid > 0)
   {
        printf("[%d] Eu sou o processo pai. \n", getpid());
        sleep(5);
   }
   else
```

```
    {
        printf("Fork error %d.\n", errno);
    }
}
```

b) Copie o código, compile-o (ficheiro **ex36b.c**) e execute-o.

2.7. Enviar sinais a processos

1. Copie, compile e execute o seguinte código (ficheiro **ex37a.c**):

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

int main(int argc,char *argv[]) {
   printf("INICIO...\n");
   while (1) {
        printf("PID %d - Aguarda sinal...\n",getpid());
        sleep(1);
   }
   return 0;
}
```

2. Numa consola à parte envie um sinal para o processo que está em execução (a opção -9 envia um sinal para terminar o processo). Consulte o manual do ***kill*** e ***killall*** (***man kill*** e ***man killall***):

```
kill -9 <pid do processo em execução>
killall <nome do processo>
```

3. Lance novamente o programa do ponto 1.

4. Copie, compile e execute o seguinte programa (ficheiro **ex37b.c**):

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Ver mais sinais: man 7 signal
int main(int argc,char *argv[]) {
   if (argc != 3) {
        printf("Usage: %s pid kill|stop|cont\n",argv[0]);
        exit(1);
   }
   if (strcmp(argv[2],"kill")==0) {
        kill(atoi(argv[1]),SIGKILL);
   } else if (strcmp(argv[2],"stop")==0) {
        kill(atoi(argv[1]),SIGSTOP);
   } else if (strcmp(argv[2],"cont")==0) {
        kill(atoi(argv[1]),SIGCONT);
   } else {
        printf("Usage: %s pid kill|stop|cont\n",argv[0]);
        exit(1);
   }
   return 0;
}
```

a) Primeiro execute o programa com a opção ***stop***, depois ***cont*** e por fim ***kill***.

3. Exercícios

1. Implemente um programa que crie uma cadeia de NUM_PROCS processos. NUM_PROCS deve ser uma constante global > 2. O processo original (processo 1) cria um processo (processo 2), o processo 2 cria o 3, o 3 cria o 4, até existirem NUM_PROCS processos. Cada processo deve esperar que o seu filho termine (se tiver filho). Cada processo deve imprimir a mensagem de apresentação e uma mensagem em que indica que o filho terminou (ficheiro **ex41.c**).

2. Desenvolvimento de programas de contagem.
   a) Implemente um programa de contagem de 1 a 10, por um processo apenas (processo único). Cada processo identifica-se com o seu nome e o seu pid (ficheiro **ex42a.c**).
   b) Alteração do programa de modo a que a contagem seja inicializada pelo processo anterior (pai), que depois de contar, cria novo processo (filho), ficando o processo pai em espera. O processo filho continua a contagem de 10 a 20. O processo pai continua a contagem de 20 a 30 depois de terminar a espera (dois Processos). (ficheiro **ex42b.c**).
   c) Alteração do programa para a criação de um terceiro processo, por parte do segundo (filho). (ficheiro **ex42c.c**)
      i) O primeiro (pai) conta de 1 a 10.
      ii) O segundo (filho) de 10 a 20.
      iii) O terceiro (filho do filho) de 20 a 30.
      iv) O segundo depois da espera do primeiro, conta de 30 a 40.
      v) O primeiro depois da espera do segundo, conta de 40 a 50.

3. Implemente um programa que cria dois processos filho. O primeiro executa explicitamente o comando "*ls -la*", o segundo executa um comando, que o programa principal recebe como parâmetro (ficheiro **ex43.c**). Consulte o manual: ***man 3 exec***.

4. Implemente um programa que mede o tempo de execução de um comando, que o programa principal recebe como parâmetro (ficheiro **ex44.c**).

5. Os seguintes exercícios visam a realização de uma auto-avaliação da aprendizagem sobre a criação de processos.

   a) Indique o *output* – **sem codificar** – dos seguinte programas.

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

int main(int argc, char* argv[])
{
   printf("INICIO\n");
   fork();
   printf("FIM\n");
   exit(0);
}
```

(ficheiro **ex45a.c**)

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

int main(int argc, char* argv[])
{
   printf("INICIO\n");
   fork();
   fork();
   printf("FIM\n");
   exit(0);
}
```

(ficheiro **ex45b.c**)

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

int main(int argc, char* argv[])
{
   printf("INICIO\n");
   if (fork()==0) {
        printf("FIM FILHO\n");
   }
   printf("FIM\n");
   exit(0);
}
```

(ficheiro **ex45c.c**)

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

int main(int argc, char* argv[])
{
   printf("INICIO\n");
   if (fork()==0) {
        printf("FIM FILHO\n");
        exit(0);
   }
   printf("FIM\n");
   exit(0);
}
```

(ficheiro **ex45d.c**)

b) Copie-os, compile-os e execute-os. Compare os resultados com os da alínea a).