instituto superior de engenharia do porto

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

SISTEMAS OPERATIVOS I

Textos de Apoio às Aulas Práticas

Introdução à Programação Concorrente em Linux

Versão 1.02

Maio de 2004

Luís Lino Ferreira
Berta Batista
Jorge Pinto Leite
António Costa

Sugestões e participações de erros para: llf@dei.isep.ipp.pt

ÍNDICE

ÍNDICE 2
1 Introdução 2
2 Representação de um processo 3
3 Gestão de processos em Linux 3
3.1 Criação de um processo – fork 3
3.2 Funções exit 6
3.3 Funções wait e waitpid 6
3.4 Outras funções 8
4 Bibliografia 12

1 Introdução

Um programa é uma entidade inactiva e estática, constituída por um conjunto de instruções e respectivos dados. Normalmente, num Sistema Operativo (SO), um programa pode existir sob a forma de ficheiro em dois formatos básico, código fonte ou código executável. Quando chamado, é lido para memória e executado sob a forma de um processo.

Um processo é portanto uma instância de um programa em execução. Ao chamar um programa este cria inicialmente apenas um processo, posteriormente este pode criar outros processos cooperantes com o processo inicial. Os SOs modernos permitem a execução concorrente de múltiplos processos (normalmente referida como capacidade de multitarefa).

O evoluir da execução de um processo é controlado pelo SO de acordo com os estados representados na Figura 1.

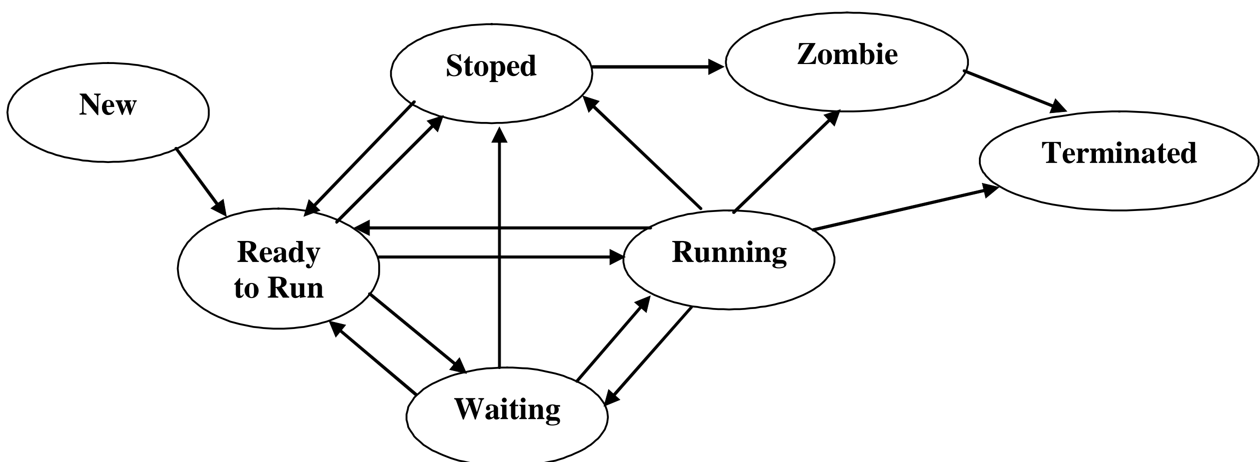

Figura 1 – Estados de um processo

No estado ***New*** o processo está a ser criado, passando de seguida para o estado ***Ready to Run***. Neste estado um processo espera até que o escalonador do SO liberte o(s) processador(es) e o coloque em execução (estado de ***Running***). Um processo é então executado por um determinado espaço de

tempo, de acordo com as políticas de escalonamento do SO. Pode deste estado passar para ***Waiting***, na existência de qualquer situação de bloqueio, por exemplo caso o processo esteja à espera de um semáforo. Pode também regressar ao estado ***Ready to run*** por decisão do escalonador. Finalmente quando termina a sua execução o processo passa para o estado de ***Terminated***, caso tenha terminado correctamente. Se deixar alguma informação pendente (p. e. se a função *exit* tiver retornado algum valor) então o processo passa para o estado de ***Zombie***. Um processo passa ao estado ***Stoped*** quando recebe uma indicação para suspender a sua execução, p.e. através de um sinal ou do comando *sleep*.

Note-se que, autores diferentes, descrevem os estados de um processo de forma diferente, por isso é necessário estar atento à forma como o SO implementa estes estados.

2 Representação de um processo

De modo a que o SO possa controlar os processos em execução, é necessário guardar informação relativa ao estado e ambiente de cada processo. O SO guarda esta informação numa estrutura chamada *Process Control Block* (PCB), que contém os dados seguintes:

- Identificação do processo: Process IDentification number (PID), dono, grupo, etc.
- Estado do processo: semelhante aos estados definidos na Figura 1.
- Registos da CPU: os registos são gravados de modo a que o SO possa recolocar o processo em execução do ponto em que foi interrompido.
- Informação para escalonamento: prioridade, tipo de processo.
- Informação para gestão de memória: zonas de memória utilizadas pelo processo.
- Informação de I/O: ponteiros para os ficheiros e dispositivos de I/O abertos.
- Informação sobre sinais: Sinais recebidos pelo processo e ainda não processados.
- Apontadores para outros processos na mesma fila.
- Apontador para o processo Pai.

É com base nesta informação que o núcleo do SO vai gerir o evoluir de cada processo. O utilizador tem à sua disposição um conjunto de funções adequadas à programação multitarefa, incluindo a criação e gestão de processos, comunicação e sincronização entre processos, etc. Na secção seguinte iremos descrever algumas das funções do LINUX disponíveis para a criação e gestão (básica) de um processo.

3 Gestão de processos em Linux

3.1 Criação de um processo – fork

A criação de um processo em Linux é feita apenas através da função *fork*. Usando esta função, o processo criador (normalmente referido como processo pai) cria uma réplica quase idêntica de si próprio: o processo filho.

Esta função retorna -1 em caso de erro e atribui à variável *errno* (variável global de controlo de erros) o erro que ocorreu. Um forma de obter a descrição erro que ocorreu é recorrer à função *perror* (que irá ser descrita mais à frente). No caso de execução sem erros, ao processo filho retorna 0 e ao pai vai retornar o PID do filho.

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

pid_t fork(void);
```

A cópia criada é quase igual ao processo original diferindo apenas em alguns detalhes. As seguintes propriedades são herdadas pelo filho:

- Variáveis, *heap* e *stack*;
- *Group ID* do processo;
- *Session ID*;
- Terminal ao qual pertence o processo;
- Directoria de trabalho;
- Descriptores de ficheiros (ficheiros abertos pelo pai);
- Máscara de criação de ficheiros;
- Máscara de sinais (que sinais vão ser tratados pelo programa);
- Variáveis de ambiente;
- Segmentos de memória partilhada;
- Limites de recursos:
- etc.

As diferenças principais são as seguintes:

- O valor de retorno de *fork* (devolve 0 ao filho e o PID do filho ao pai);
- O PID do processo pai é diferente;
- É feito o reset a todos os locks a ficheiros;
- As acções a tomar quando é recebido um sinal são diferentes;
- etc.

Na Figura 2 é apresentado um exemplo simples.

Este exemplo apenas utiliza dois processos.

O processo pai imprime "Pai: Eu sou o processo Pai" ; o processo filho vai imprimir "Filho: Eu sou o processo Filho".

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

main()
{
 pid_t pid;
```

```
 pid = fork(); /* Cria um PROCESSO */
 if (pid >0) /* Código do PAI */
 {
     printf("Pai: Eu sou o processo Pai\n");
 }
 else /* Código do FILHO */
 {
      printf("Filho:Eu sou o processo Filho\n");
 }
} /* fim main */
```

Figura 2 – Exemplo 1 de *fork*

O exemplo da Figura 3, demonstra de que forma pode ser feito o controlo de erros do programa assim como a herança das variáveis actuais pelo filho. Verifica-se, durante a execução do programa, que as alterações ao valor das variáveis feita por cada um dos processos não se vai reflectir no outro.

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

main(void)
{
 pid_t pid;
 int a;

 a = 1;
 pid = fork(); /* Cria um PROCESSO */
 if (pid < 0)
 {
     perror("Erro ao criar o processo:");
     exit(-1);
 }
 else
 {
     if (pid >0) /* Código do PAI */
     {
          printf("Pai: Eu sou o PAI\n");
          printf("Pai: a=%d\n", a);
          a = a + 1;
          printf("Pai:a+1=%d\n", a);

     }
     else /* Código do FILHO */
     {
          printf("Filho:Processo Filho\n");
          printf("Filho:a=%d\n", a);
          a = a + 1000;
          printf("Filho:a+1000=%d\n", a);
     }
 }
 exit(0);
} /* fim main */
```

Figura 3 – Exemplo 2 de *fork*

3.2 Funções exit

Um processo pode ser terminado através da função *exit* ou através de uma chamada à função *return* na função *main*. Seguidamente o kernel fecha todos os descriptores abertos, liberta a memória usada pelo processo e guarda informação mínima sobre o estado de saída do processo filho – PID, estado de finalização e tempo de CPU gasto. Esta informação poderá ser obtida pelo pai através das funções *wait* e *waitpid*.

Quando um processo termina, mas o seu pai ainda não foi buscar os seus dados de retorno, esse processo passa a ser designado pelo sistema operativo como zombie até que o pai chame uma das funções da Secção 3.3.

Um processo também pode terminar anormalmente, através de uma chamada à função de *abort* ou devido a ter recebido certos tipos de sinais. Para mais detalhes quanto a este assunto consultar a bibliografia aconselhada.

A sintaxe da função *exit* é a seguinte:

```
#include <unistd.h>

void exit(int status);
```

A variável *status* é um inteiro e permite retornar **apenas** os seus 8 bits menos significativos para o pai. Na secção seguinte será descrito como é que o processo pai pode obter o valor de *status*.

3.3 Funções wait e waitpid

Quando um processo termina o seu pai é informado desse facto através do sinal SIGCHLD. Este evento é assíncrono, por isso o pai pode receber este sinal em qualquer altura da sua execução. Juntamente com o sinal, o SO armazena o valor de retorno do processo filho juntamente com outra informação relativa ao estado de saída do filho. O pai pode optar por ignorar o sinal (situação por defeito) ou indicar uma função que irá ser chamada quando um filho terminar.

No âmbito de SOP1 vamos assumir que o pai ignora os sinais; posteriormente em SOP2 este assunto irá ser abordado em detalhe.

Ao chamar as funções *wait* ou *waitpid*, o processo evocador poderá ter um dos seguintes comportamentos:

- ficar bloqueado até que o filho termine;
- retornar imediatamente, caso o filho já tenha terminado;
- retornar imediatamente, com um erro, caso já não existam mais filhos em execução.

A função *wait* espera até que qualquer filho termine. *waitpit* espera até que um processo filho específico termine, assim como permite esperar por um filho sem bloquear. Os protótipos destas funções são apresentados a seguir.

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

pit_t wait(int *status);

pit_t waitpid(pit_t pid, int *status, int options)
```

status: é um apontador para um inteiro. Se este apontador não for passado como nulo, então poderá armazenar o estado de finalização do processo filho.

pid: processo pelo qual a função *waitpid* vai esperar. Se *pid* = -1, *waitpid* espera por qualquer processo filho, *pid* > 0 então *waitpid* espera por um processo cujo *pid* seja igual a *pid*.

options: 0 de modo a que o processo fique bloqueado à espera do processo filho; pode também ser igual ao OR (ou lógico) de duas constantes: WNOHANG, WUNTRACED. A primeira constante significa que a função retorna imediatamente se nenhum filho tiver terminado e a segunda retorna também o estado dos filhos que se encontrem no estado de STOP (opção pouco utilizada).

Ambas as funções *wait* e *waitpid* retornam o PID do processo que terminou em caso de sucesso e -1 em caso de erro.

Waitpid retorna 0 caso nenhum filho tenha terminado e se a opção WNOHANG tivesse sido usada na chamada da função.

Mais uma vez, em caso de erro, a variável global *errno* é actualizada podendo o seu significado ser apresentado ao utilizador através da função *perror*.

No valor de *status* vêm codificada bastante informação, nomeadamente: os 8 bits menos significativos que foram passados como parâmetro à função *exit* (chamada pelo filho), o sinal que provocou o fim (anormal) do processo filho, se o fim do filho deu origem a um ficheiro de core, etc.

Contudo, a forma mais fácil de extrair essa informação é através de macros específicas.

A macro WIFEXITED(*status*) devolve verdadeiro se o filho retornou normalmente e nessa altura pode-se chamar outra macro, WEXITSTATUS(*status*), que vai retornar os 8 bits menos significativos que foram passados como parâmetro à função *exit*.

O exemplo seguinte cria um filho que após ficar suspenso de execução durante 5 segundos, termina retornando o número 5. O pai espera sem bloquear que o filho termine. A função *sleep* usada neste programa permite suspender a execução de um processo durante x segundos.

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

int main(void)
{
 pid_t pid;
 int aux;
 int status;

 pid=fork();
 if (pid<0)
 {
     perror("Erro ao cria o processo\n");
```

```
    exit(-1);
 }
 else
 {
    if (pid > 0) /* Código do Pai */
    {
         printf("Pai\n");
         do
         {
              aux = waitpid(pid, &status, WNOHANG);
              if (aux==-1)
              {
                   perror("Erro em waitpid");
                   exit(-1);
              }
              if (aux == 0)
              {
                   printf(".\n");
                   sleep(1);
              }
         } while (aux == 0);
         if (WIFEXITED(status))
         {
              printf("Pai: o filho retornou o valor:%d\n", WEXITSTATUS(status));
         }
    }
    else /* Código do filho */
    {
         printf("Filho\n");
         sleep(5);
         printf("Filho a sair\n");
         exit(5);
    }
    exit(0);
 }
}
```

Figura 4 – Exemplo de *fork* e *exit*

3.4 Outras funções

Nesta secção são descritas algumas funções que foram utilizadas ao longo do texto e que tem particular interesse dentro deste contexto.

```
#include <stdio.h>

void perror(const char *s);
```

Na maior parte dos casos as funções do SO devolvem o valor -1 em caso de erro, e actualizam a variável externa *errno* com uma indicação do erro ocorrido. É considerado um bom hábito de programação que o valor de retorno das funções evocadas seja analisado de modo a detectar se ocorreu um erro. Caso tenha ocorrido um erro este deverá ser tratado e no limite o programa deve imprimir uma mensagem de erro e sair. A função *perror* permite imprimir a string que lhe é passada como apontador, seguida de ":" e da descrição do último erro que ocorreu. A *string* que é passada

como argumento pode ser usada pelo programador, por exemplo, para indicar em que função é que ocorreu o erro.

```
#include <unistd.h>

Unsigned int sleep(unsigned int t);
```

A função *sleep* permite que um processo adormeça durante *t* segundos ou até que o processo receba um sinal. Retorna zero caso tenha adormecido durante a totalidade dos *t* segundos ou os segundos que faltam para o seu término normal, caso a sua execução tenha sido interrompida por um sinal.

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

pid_t getpid(void);
pid_t getppid(void);
```

A função *getpid* permite devolver o PID do próprio processo e *getppid* permite obter o PID do processo pai.

Exercícios:

1 – Faça um programa que crie um processo e:

- Se for o processo *pai* deve escrever "Eu sou o *pai*"
- Se for o processo *filho* deve escrever "Eu sou o processo *filho*"

2 – Faça um programa que crie um processo e:

- O processo *pai* escreve os números de 1 a 10.
- O processo *filho* escreve os números de 11 a 20.

3 – Considere o programa seguinte:

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

void main(void)
{
 pid_t pid;
 int f;

 for (f = 0; f < 3; f++)
 {
     pid = fork(); /* Cria um PROCESSO */
     if (pid > 0) /* Código do PAI */
     {
          printf("Pai: Eu sou o PAI\n");
     }
     else /* Código do FILHO */
     {
          sleep(1);
     }
 }
} /* fim main */
```

a) Analise o código deste programa e verifique quantos processo vão ser criados.

b) Teste o programa de modo a verificar se o resultado da alínea anterior está correcto.

c) Desenhe uma arvore que descreva os processo criados.

d) Altere o programa de modo a que apenas sejam gerados 3 filhos.

e) Altere o programa de modo a que todas as possíveis situações de erro sejam tratadas.

f) Altere o programa de modo que o pai espere pelo fim de cada um dos filhos.

g) Altere o programa de modo a que cada filho devolva o seu número de ordem ao pai. O pai deverá imprimir qual o número de ordem de cada um dos filhos que vai terminando, assim como o seu PID.

h) Altere o programa de modo a que o pai apenas espere pelo segundo filho, mas sem bloquear.

4 – Considere o programa seguinte:

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

void main(void)
{
 pid_t pid;
 int f;

 fork();
 printf("1\n");
 fork();
 printf("2\n");
 fork();
 printf("3\n");

}
```

a) Desenhe uma árvore que descreva o conjunto dos processos criados.

b) Será possível que o número 1 apareça depois do 3? Porquê?

5 - Faça um programa que crie 2 processos e:

- Escreve "Eu sou o *pai*" no processo *pai*.
- Escreve "Eu sou o 1º *filho*" no primeiro *filho*.
- Escreve "Eu sou o 2º *filho*" no segundo *filho*.

6 - Faça um programa que crie um processo e:

- O processo *filho* escreve os números 1 .. 500.
- O *filho* quando terminar, "retorna" o valor 5.
- O processo *pai* escreve os números 501 .. 1000.
- O *pai* só deve terminar quando o *filho* terminar e tem de escrever o "valor de saída" do *filho*.

a) O que observa de estranho?

b) Modifique o seu programa de modo a garantir que os números apareçam de forma correcta.

7 - Faça um programa que crie 5 processos e:

- Cada processo escreve 200 números:

- 1º processo: 1 .. 200
- 2º processo: 201 .. 400
- 3º processo: 401 .. 600
- 4º processo: 601 .. 800
- 5º processo: 801 .. 1000
- O processo *pai* tem de esperar que todos os processos *filho* terminem.

8 - Tendo um array de 1000 posições, faça um programa que crie 5 processos e:

- Dado um número, procurar esse número no array.
- Cada processo *filho*, procura 200 posições.
- O processo que encontrar o número, deve imprimir a posição do array onde se encontra. Também deve "retornar" como valor de saída o número do processo (1, 2, 3, 4, 5).
- Os processos que não encontrarem o número devem "retornar" como *valor de saída* o valor 0.
- O processo *pai* tem de esperar que todos os *filhos* terminem e imprimir o número do processo onde esse número foi encontrado (1, 2, 3, 4, 5).

Nota: O array não tem números repetidos.

4 Bibliografia

Orlando Sousa, “Processos”, Apontamentos da aulas práticas de SOP2, Instituto Superior de Engenharia do Porto, 2004.

W. Richard Stevens, “Advanced Programming in the UNIX Environment”, Addison Wesley, 1994

John Shapley Gray, “Interprocess Communications in UNIX – The Nooks & Crannies, 2nd Edition”, Prentice Hall, 1998