Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

SISTEMAS OPERATIVOS I

Textos de Apoio às Aulas Práticas

Programação Concorrente em Linux: Gestão de Processos

Versão 2.02

Maio de 2006

Luís Lino Ferreira
Berta Batista
Maria João Viamonte
Nuno Malheiro

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

Agradecimentos

Agradecemos aos colegas Jorge Pinto Leite e António Costa a sua participação na revisão deste documento.

ÍNDICE

1 Introdução

Estes apontamentos tem como objectivo dar a conhecer aos alunos de Sistemas operativos I alguns conceitos básicos sobre o modo de programação do Sistema Operativo (SO) Linux assim como sobre os mecanismos de programação multitarefa.

2 Tópicos sobre programação em Linux

Nesta secção são discutidos vários conceitos e algumas funções relativos à programação em Linux com particular interesse para o desenvolvimento de aplicações multitarefa.

2.1 Codificação de um programa

A codificação de uma rotina é normalmente feita em 4 passos: desenho da rotina, verificação do desenho da rotina, codificação da rotina, verificação do código da rotina. O insucesso de um deste passo pode levar a voltar ao passo anterior. Embora à primeira vista o estilo de programação possa parecer um tema de menor interesse, tem-se verificado que a forma como são escritos os programas tem uma elevada importância para a sua correcção, assim como, para uma rápida detecção de eventuais *bugs* no código pelo autor ou por outros.

O estilo de programação inclui vário temas: formatação do código, nomes das variáveis e funções, convenções relativas à interface das funções, escrita de rotinas, etc. De seguida são indicados alguns concelhos e exemplos (para um tratamento mais completo deste tema é aconselhada a consulta da bibliografia).

Formatação do código

Os objectivos de uma boa formatação do código são:

1. Representar com precisão a estrutura do código
2. Melhorar a compreensão do código
3. Facilitar modificações do código

Técnicas de formatação:

1. Utilização de espaços: a utilização de espaços, tabs e linhas em branco permite organizar um programa e melhorar a sua aparência. O exemplo seguinte mostra como se podem separar duas variáveis num **`if`**:

```
#include<stdio.h>

main()
{
 if (a == b)
 {
…
```

2. Agrupar secções de código: deve agrupar-se o código relacionado entre si em linhas consecutivas separadas por uma ou mais linhas em branco. Por exemplo agrupar as linhas que escrevem alguma informação no monitor (p.e. utilizando `printf`) das linhas que vão ler dados (p.e. utilizando `scanf`).
3. Alinhamento do código: pode alinhar-se secções de código que pertençam ao mesmo conjunto lógico de operações

```
#define DLL_MAC   = 123456
#define DLL_RETRY = 2
#define IP_ADDR   = 193.136.56.76
```

4. Identação: permite mostrar a estrutura lógica de um programa. Como regra devem ser identadas todas as linhas de código subordinadas à mesma estrutura de controlo. O exemplo seguinte mostra uma forma muito utilizada em C. Note-se que o fecho das estruturas de controlo `if` devem estar alinhadas com o respectivo `if` e devem ser inseridos comentários no ponto de fecho.

```
…
if (a == b)
{
     printf("abc\n");
     if ((c > d) &&
          (e > f) &&
          (f < g))
     {
          printf(">>>>>\n");
     } // if >
} // if a == b
```

Comentários

Um programa ou uma função devem ter **SEMPRE** comentários descrevendo o seu funcionamento.

No caso de um programa deve normalmente aparecer no início as funcionalidades principais do programa juntamente com os autores, (Figura 1). Como sugestão particular para a disciplina de SOP1 aconselha-se a inclusão no princípio do programa do texto do problema.

No caso de uma função, o texto deve descrever os objectivos da função, descrevendo em detalhe as variáveis de entrada e de saída da mesma, (Figura 2).

A inserção de comentários no meio do código deve ser feita de forma criteriosa, utilizando uma linguagem de alto nível que descreva os objectivos de um pedaço de código, tal como é exemplificado no código descrito na Figura 2 (note-se que para um programador experiente o número de comentários existentes neste programa pode ser considerado excessivo!).

```
/*********************************************************************

     Nome do programa:  Exemplo
     Autor:             Joaquim Costa
     Organização:       ISEP
     Data:              23/04/2005
     Versão:            1.00

Aqui coloca-se o texto que descreva as funcionalidades do programa, p.e.
o número e o texto de uma pregunta.

*********************************************************************/
…

main ()
…
```

Figura 1 – Exemplo de um comentário a inserir no início de um programa

```
/*********************************************************************

     Nome da função:    sqr
     Autor:             Joaquim Costa
     Organização:       ISEP
     Data:              26/04/2005
     Versão:            1.00

     Parâmetros:
          double a:     valor a calcular, a deve ser maior ou igual a 0
     Saída:
          devolve um valor do tipo int com o resultado do factorial
          Em caso de erro é devolvido o valor -1

Esta função permite calcular o factorial de um número.
*********************************************************************/
int fact (int a)
{
 int f;       /* variável contadora */
 int res;     /* resultado */

 /* Factorial de 1 é igual a zero */
 if (res == 1)
 {
     res = 0;
```

```
 }
 else
 {
     /* Calculo para os restantes casos
      for(f = 1; f < a; f++)
     {
          res = res * f
      }
 }

 /* devolve o resultado */
 return(res);

} // fim sqr
```

Figura 2 – Exemplo de comentário a inserir no início de uma função

Nomes das Variáveis e funções

A escolha de "bons" nomes para as variáveis permite a escrita de código com propriedades auto-explicativas, melhorando a compreensão do programa ou rotina. A tabela seguinte apresenta alguns exemplos do que se deve e do que não se deve fazer.

Tabela 1 – Nomes para as variáveis

Objectivo da variável	Nomes correctos	Nomes incorrectos
Velocidade de um comboio	`Velocidade, velocidade, VelocidadeEmKm`	`vel, v, vc, x, x1, comboio`
Data de hoje	`DataHoje, Data_de_Hoje`	`DH, hoje, data`
Linhas por página	`LinhasPorPagina`	`LPP, linhas, x, x1`

Conselhos úteis:

- O nome das variáveis deve descrever o mais claramente possível o seu propósito (mesmo que para tal fique muito comprida!).
- Adoptar sempre o mesmo tipo de convenção para os nomes. Pode-se usar sempre maiúsculas para distinguir as palavras constituintes do nome da variável (NomeDaVariável) ou então o *underscore* (nome_da_variavel).
- Identificar variáveis globais. Exemplo: `g_var`, `gVariávelGloblal`
- Usar sempre maiúsculas nas constantes. Exemplo `PI`
- As variáveis `i`, `j`, `k`, `f` são normalmente utilizadas em ciclos. Se estas variáveis forem utilizadas fora dos ciclos ou em ciclos compostos, então o seu nome deve ser mais claro.
- Dar nomes às variáveis booleanas que impliquem verdadeiro ou falso, p.e.: `done`, `erro`.
- Usar sempre nomes positivos para as variáveis boolenas. Evitar utilizar nomes do tipo `NaoEncontrado`
- Identificar os tipos definidos pelo utilizador sempre da mesma forma, p.e.: `NomeDoTipo_t`

As funções devem seguir basicamente o mesmo tipo de convenções utilizadas nas variáveis de uso geral, i.e. devem reflectir o propósito da função.

2.2 Gestão de erros em Linux

Em Linux, normalmente e por convenção quando uma chamada a uma função falha é devolvido o valor -1 e é atribuido à variável externa *errno* um valor que identifica o tipo de 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.

```
#include <stdio.h>

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

A função `perror()` permite imprimir a *string* que lhe é passada através do apontador *s*, seguida de ":" e de uma descrição do último erro que ocorreu (p.e. *File not found.*). A *string* que é passada como argumento pode ser usada pelo programador, por exemplo, para indicar em que função é que ocorreu o erro.

O exemplo apresentado na Figura 3, mostra forma como a função `perror()` pode ser utilizada. Neste caso, dado que o ficheiro x.txt foi aberto apenas para leitura a função `perror()` na linha 13 indica que o descritor do ficheiro (variável `fp`) está incorrecto, mensagem *Bad file descriptor*, dado que o programa não tem permissões de leitura sobre o ficheiro aberto. Note-se que neste pequenos programas de exemplo não são colocados comentários uma vez que estes são descritos no texto e de modo a poupar espaço.

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

main ()
{
 FILE *fp;
 char buffer[20];
 int error;

 fp = fopen("x.txt", "w");
 if (fp == NULL)
 {
        perror("Erro na em fopen ");
        exit(-1);
 }

 error = fscanf(fp,"%s", buffer);
 if (error == -1)
 {
        perror("Erro na linha 19 ");
        exit(-1);
 }
}
```

Figura 3 – Exemplo de utilização da função `perror()`

2.3 Outras funções relacionadas com a gestão de processos

A função *sleep* permite que um processo adormeça durante *t* segundos ou até que o processo receba um sinal, i.e. o processo passa para o estado de **Waiting**. 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 <unistd.h>

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

A função `getpid()` permite devolver o PID do próprio processo e `getppid()` permite obter o PID do processo pai.

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

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

3 Processos em Linux

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 básicos, 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).

3.1 Estados de um processo

O evoluir da execução de um processo é controlado pelo SO Linux de acordo com os estados representados na Figura 4.

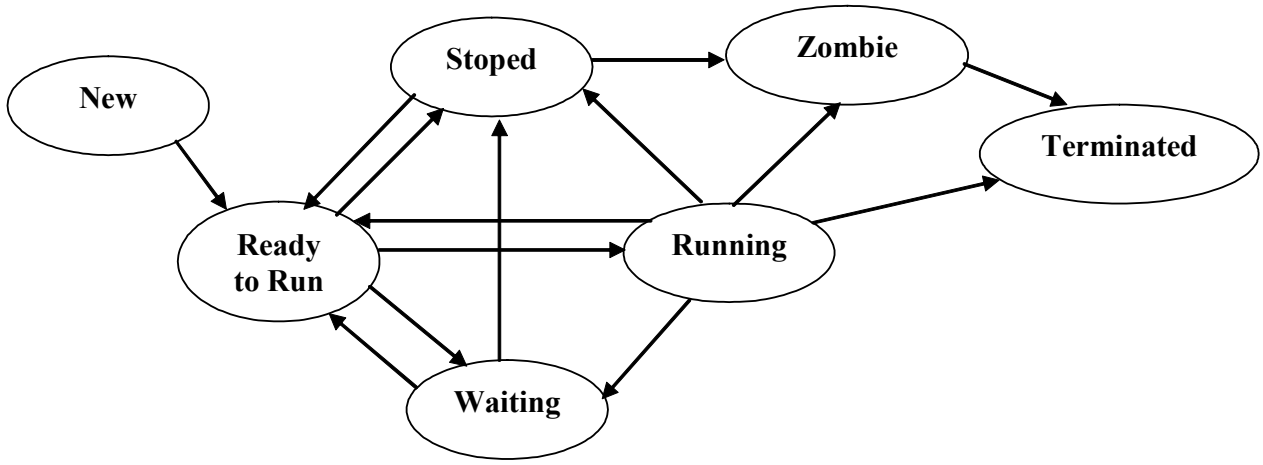

Figura 4 – Estados de um processo

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.

3.2 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 4.
- 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.

4 Programação multitarefa em Linux

4.1 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 do erro que ocorreu é recorrer à função `perror()`. Caso a execução da função decorra sem erros, ao processo filho é retornado o valor 0 e ao processo pai é retornado o PID do filho que acabou de ser criado.

```
#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;
- etc.

Na Figura 5 é apresentado um exemplo simples. que 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 <stdio.h>
#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 5 – Utilização da função *fork()*

O exemplo da Figura 6, 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 <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.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 6 – Actualização das variáveis

Note-se que nos programas das Figuras 5 e 6 as funções `printf()` tem imprimem sempre se "Pai:" ou "Filho:" no caso da impressão ser feita pelo processo pai ou pelo processo filho, respectivamente. Sugere-se

a utilização desta norma em todos os programas de modo a facilitar a análise dos resultados impressos no ecrã e permitir um *debugging* mais fácil dos programas.

4.2 Função `exit()`

Um processo pode ser terminado através da função `exit()` ou através de uma chamada à função *`return()`* (apenas se efectuado 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 posteriormente pelo pai através das funções `wait()` e `waitpid()`.[1]

Quando um processo termina, mas o seu pai ainda não foi buscar os seus dados de retorno, esse processo passa ao estado de ***Zombie*** até que o seu pai chame a função `wait()` ou `waitpid()`.

Um processo também pode terminar anormalmente, através de uma chamada à função de *abort* ou devido a ter recebido certo tipo de sinais (para mais detalhes quanto a este assunto consultar a bibliografia aconselhada).

A sintaxe da função `exit()` é a seguinte:

```
#include <stdlib.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`.

4.3 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, assim como, outra informação relativa ao estado de saída do processo 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. `waitpid()` espera até que um processo filho específico termine, além disso, esta função também permite esperar por um filho sem bloquear o processo pai. Os protótipos destas funções são apresentados a seguir.

[1] Adicionalmente o kernel do SO envia o sinal SIGCHLD ao respectivo processo pai.

```
#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. Note-se que este ponteiro já deve ter o seu espaço de memória reservado antes da evocação da função.

`pid`: processo pelo qual a função `waitpid()` vai esperar. Se `pid` igual a -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 ***Stopped*** (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 <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdlib.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 7 – Exemplo de `fork()`, `exit()` e `waitpid()`

Note-se que no código da Figura 7 foram omitidas a maior parte das verificações de erros de modo a simplificar o exemplo.

A relação entre processo pais e filhos pode ser representada através de um árvore de processos. Nesta tipo de árvores, os processos são representados por linhas verticais, caso necessário a execução de uma determinada linha de código também pode ser representada na mesma linha através de uma pequena linha horizontal. A execução de um fork, e a criação de um processo filho é representada por uma derivação à linha principal. A finalização de um processo é representada por uma circunferência a cheio. A comunicação entre os processos (troca de sinais, acessos à memória partilhada, etc) é representada através de setas. A Figura 8, mostra a árvore de processos para o programa da Figura 7.

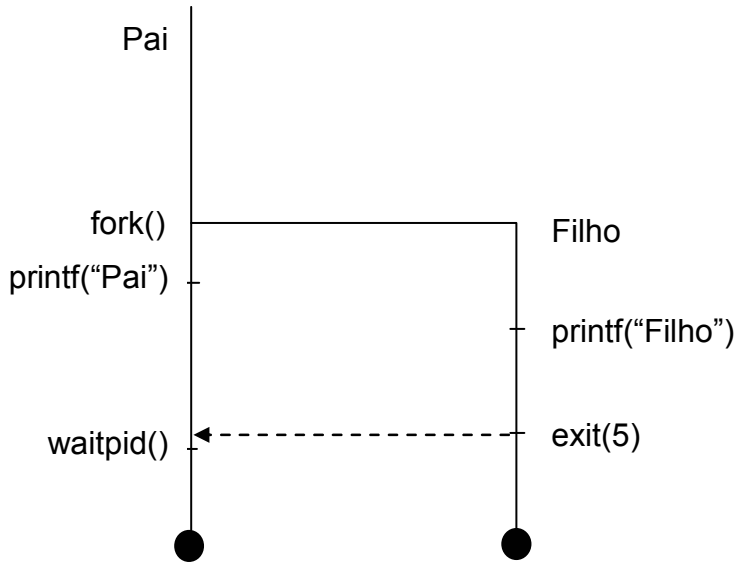

Figura 8 – Árvore de processos

5 Bibliografia

Orlando Sousa, "Processos", Apontamentos das 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

Steve McConnell, "CODE Complete: A Practical Handbook of Software Construction", Microsoft Press Books, 1993