

Ficha 1 – Controlo de Processos

Tópicos abordados:

- Conceitos
- Controlo de processos: fork, wait, getpid, getppid, exit
- Substituição da imagem em memória por um processo (*exec*)
- A função system
- Exercícios

Duração prevista: 1 aula

©1999-2020:

{patricio.domingues, vitor.carreira, miguel.frade, rui.ferreira, nuno.costa, gustavo.reis, carlos.machado, leonel.santos, miguel.negrao}@ipleiria.pt

1. Configurações iniciais da máquina virtual

Se a máquina virtual que está a utilizar está a ter problema com o acesso aos símbolos ‘{’ e ‘}’, ou se o fuso horário não é o correto, então execute os próximos passos para resolver estes problemas. Caso contrário, então deve seguir para o capítulo 2.

1. Caso tenha problema com o acesso aos símbolos ‘{’ e ‘}’ na máquina virtual, modifique a última linha do ficheiro ~/.bashrc por forma a que tenha somente o seguinte conteúdo:

```
setxkbmap -option
```

Em alternativa pode executar a seguinte sequência de comandos:

```
sudo dpkg-reconfigure keyboard-configuration
```

e escolher as opções:

- Generic 105 (PC)
- Portuguese
- Portuguese
- Right Alt (ALTGr)
- OK
- No compose key
- Yes

2. O fuso horário do lubuntu pode ser alterado através do seguinte comando:

```
sudo dpkg-reconfigure tzdata
```

2. Controlo de processos

Programa – conjunto de instruções e dados mantidos num ficheiro. É sempre criado um novo processo para executar um programa.

Processo – ambiente de execução de um programa que consiste em três elementos: instruções, dados do utilizador e dados do sistema. Os dados do sistema (ou contexto do processo) podem ser ficheiros abertos, tempo de CPU, etc.

Processo *init* (ou processo *systemd* em sistemas mais recentes) – processo especial do sistema que é inicializado no arranque do sistema operativo com a particularidade de ser responsável pelo “arranque” de todos os outros processos, ou seja, ser o pai de todos os outros processos. O processo *init* tem PID (*process identifier*) igual a 1.

Nota: Por definição, um **descriptor** representa um recurso do sistema operativo, por exemplo, ficheiros, mecanismos IPC (*Inter-Process Communication*), processos, etc.

Para visualizar informação referente aos processos existentes num dado instante no sistema, pode utilizar-se o comando *ps*. Existe ainda no Linux o comando *pstree*, que permite visualizar a hierarquia de processos, mostrando o relacionamento de parentesco entre os diversos processos.

Para sinalizar um processo (o que, dependendo do sinal, pode levar ao término do mesmo) utiliza-se o comando *kill*, indicando-se o sinal a enviar, por exemplo:

```
$ ps
$ pstree
$ kill -SIGKILL <pid-processo>
$ kill -9 <pid-processo> # equivalente à linha anterior
```

Tenha em atenção que apenas o dono de um processo ou o administrador do sistema é que pode proceder ao envio de um *signal* a um processo.

Uma listagem de *signals* que descreve os respetivos efeitos, encontra-se disponível através da página “**signal**” da secção 7 do manual (acessível através do comando: **man 7 signal**). A matéria referente a sinais será lecionada em mais detalhe na próxima ficha.

2.1. getpid, getppid – Identificadores do processo

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

pid_t getpid(void);
```

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

2.1.1. Valores de retorno

A função *getpid()* devolve o *pid* do próprio processo, já a função *getppid()* devolve o *pid* do processo pai. No limite a função *getppid()* devolve o *pid* do processo *init* (ou *systemd*).

Nota: quando um processo pai termina, o processo pai dos seus filhos (órfãos) passa a ser o processo *init* (ou *systemd*).

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

int main(void) {
    printf("O meu PID e':%d \n\n", getpid());
    printf("O PID do meu pai e':%d\n\n", getppid());
    return 0;
}
```

Listagem 1 - As funções *getpid()* e *getppid()*

Lab 1

Gere o programa executável do código fonte da listagem 1. Compile utilizando diretamente o GCC na linha de comando;

Lab 2

Recorrendo ao template de projeto da UC, compile e execute o programa “lab2” utilizando o código fonte da listagem 1.

2.2. **exit** – Termina o processo atual

```
#include <stdlib.h>

void exit(int status);
```

A função *exit()* termina o processo que a executa. O parâmetro *status* é utilizado para devolver informação de estado acerca da terminação do processo ao sistema operativo¹.

Norma: convencionou-se que um programa deve devolver o valor 0 se a execução decorreu normalmente, devendo retornar um apropriado e documentado código de erro caso a sua execução seja interrompida por via de um erro (por exemplo: 1 na falha de alocação de memória; 2 no caso do ficheiro não existir; etc.).

¹ Outra forma que um programa tem para devolver um valor ao sistema operativo é através do *return* na função “main”.

Em Unix, o valor a devolver deve ser um inteiro de 8 bits sem sinal, significando que o valor a devolver deve estar compreendido entre 0 e 255.²

2.3. Fork – Criação de um novo processo

```
#include <unistd.h>

pid_t fork(void);
```

A função ***fork()*** cria um processo, denominado processo filho, absolutamente idêntico ao processo pai. Assim, o processo filho herda todo o contexto do processo pai e continua a executar o mesmo código na instrução seguinte ao ***fork()***. Isto significa que o processo filho partilha todos os recursos detidos pelo processo pai, tais como, os dispositivos de E/S, o nível de prioridade, i.e., o contexto do processo. No entanto, o processo filho passa a ter um novo *pid*.

2.3.1. Valores de retorno

Sucesso – No caso de sucesso, a função devolve um valor maior ou igual a zero. No processo filho, o valor devolvido é zero, enquanto no processo pai, esse valor é igual ao *pid* do processo filho, ou seja, maior que zero.

Insucesso – Neste caso a função devolve o valor -1 ao processo pai, não sendo criado o processo filho. Nesta situação, a variável *errno* contém o código de erro.

Ao usar a função ***fork()*** é necessário tratar os 3 casos possíveis relativos ao valor devolvido: menor que zero (erro), zero (processo filho), maior que zero (processo pai). Tal pode ser feito de várias formas: usando um *if* que contém outro *if*; usando um *switch*; usando condições de proteção.

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

int main(void){
    pid_t pid = fork();
    if (pid == 0)
        /* Processo filho */
        printf("PID do processo filho = %d\n", getpid());
    else if (pid > 0)
        /* Processo pai */
        printf("PID do processo pai = %d, PID do processo
               filho = %d\n", getpid(), pid);
    else
        /* <0 - erro*/
        ERROR(1, "Erro na execução do fork()");
}
```

² http://en.wikipedia.org/wiki/Exit_status

```

    /* Ambos os processos executam o resto do código */
    DEBUG ("O processo %d terminou", getpid ());
    return 0;
}

```

Listagem 2 - Criação de um processo e distinção entre processos com *if ... else*

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "debug.h"

int main(void){
    pid_t pid;

    switch (pid = fork()) {
        case -1:      /* erro */
            ERROR(1, "Erro na execução do fork()");
            break;
        case 0:       /* filho */
            printf("PID do processo filho = %d\n", getpid());
            break;
        default:      /* pai */
            DEBUG("PID do processo pai = %d, PID do processo
                filho = %d\n", getpid(), pid);
            break;
    }

    /* Ambos os processos executam o resto do código */
    DEBUG ("O processo %d terminou", getpid ());

    return 0;
}

```

Listagem 3 - Criação de um processo e distinção entre processos com *switch*

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "debug.h"

int main(void){
    pid_t pid;
    int var1 = 10;

    printf("PAI | Valor inicial da VAR1= %d\n", var1);
    pid = fork();
    if (pid == 0)
    { /* Processo filho */
        var1 = 20;
        printf("FILHO | Valor final da VAR1= %d\n", var1);
        exit(0); /* Termina o processo filho */
    }

    if (pid > 0)
    { /* Processo pai */

```

```

var1 = 5;
wait(NULL); /* Espera que o processo filho termine */
printf("PAI | Valor final da VAR1= %d\n", var1);
exit(0); /* Termina o processo pai */
}
/* < 0 -- erro */
ERROR(1, "Erro na execução do fork()");
}

```

Listagem 4 – Criação de processos com condições de proteção

2.3.2. Exemplo da criação de um processo

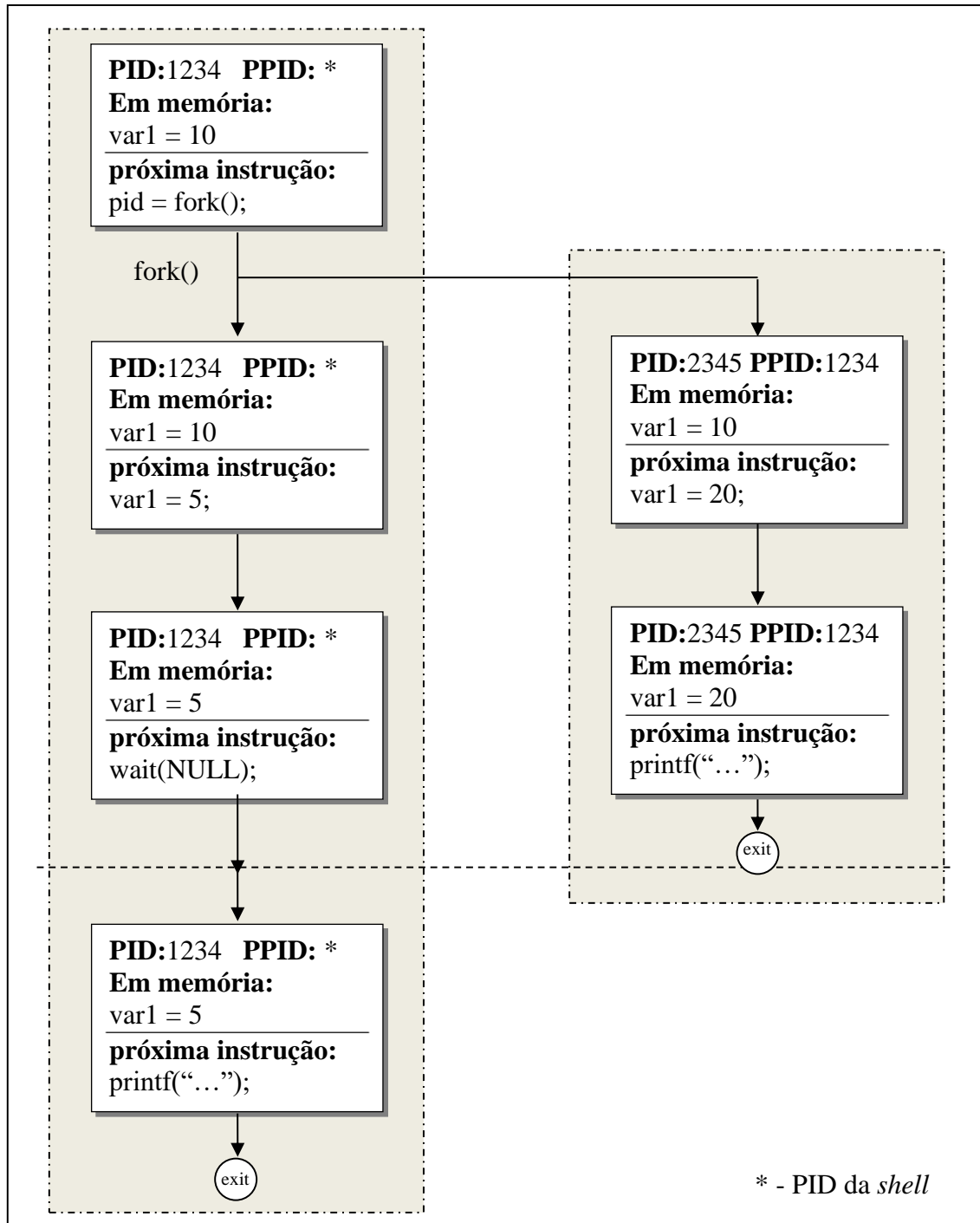


Figura 1 – Esquematização da criação de processos

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

#include "debug.h"

int main(void) {
    printf("PID=%d (%d)\n", getpid(), getppid());
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid == -1)
            ERROR(1, "Erro na execucao do fork");
        else
            printf("Processo %d (%d)\n", getpid(), getppid());
    }

    return 0;
}
```

Listagem 5 - Criação de vários processos

Lab 3

Sem recorrer à execução do programa, indique quantos processos são criados com o código da listagem anterior?

Lab 4

Analisando apenas o código fonte, elabore a árvore de processos resultante da execução do seguinte código, indicando para cada processo o valor de cada uma das variáveis:

```
int a, b, c, d;
a = b = c = d = -1;
a = fork();
if (a == 0)
    b = fork();
c = fork();
if (b == 0 && a > 0)
    d = fork();
```

Listagem 6 - Código fonte do lab4

2.4. Sleep – Adormece o processo

```
#include <unistd.h>

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

A função *sleep()* suspende o processo corrente até que tenham decorrido *seconds* segundos ou até que chegue um sinal. Como argumento, a função recebe um número inteiro positivo que define o número de segundos que o processo deve dormir.

Nota: Caso necessite de especificar o tempo a “dormir” com maior precisão, use a função *nanosleep()*.

2.4.1. Valores de retorno

A função *sleep()* devolve zero caso todo o tempo especificado no argumento tenha passado efetivamente. Caso a função devolva um valor superior a zero isso quer dizer que a função terminou e ainda faltava “dormir” esse tempo.

2.5. Wait – Suspende o processo

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

pid_t wait(int *stat_loc);

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

A função *wait()* suspende a execução do processo que a executa até que termine um processo filho ou que receba um outro sinal.

A função *waitpid()* tem um funcionamento idêntico à função *wait()* com a vantagem de se poder especificar por que processo filho se pretende esperar, usando o argumento *pid*. Se for passado o valor -1, no argumento *pid*, então a função *waitpid()* comporta-se da mesma forma que a *wait()*, ou seja, espera por um qualquer processo filho. No caso de ser passado um valor positivo, então a função espera especificamente pelo processo filho identificado por esse *pid*. Para mais informação consultar o manual.

Os ponteiros *stat_loc* e *status* servem para receber, do sistema operativo, o código numérico devolvido pelo processo cujo término levou a que a respetiva função *wait()/waitpid()* terminasse. O valor devolvido deverá ser interpretado pelas macros *WIFSIGNALED(status)* e *WTERMSIG(status)* e não diretamente (consultar “man 2 wait”).

O parâmetro *options* permite personalizar o comportamento da função *waitpid()*. De salientar que caso a opção *WNOHANG* seja especificada neste parâmetro, o processo chamado não espera (e, portanto, a chamada não se comporta de forma bloqueante) caso não existam processos que possam terminar.

2.5.1. Valores de retorno

Sucesso – Neste cenário, a função devolve o *pid* do processo que terminou.

Insucesso – Em caso de insucesso, a função devolve -1 e é especificado o código de erro na variável *errno* de sistema.

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

#include "debug.h"

int main(void) {
    pid_t pid = fork();
    if (pid == 0) { /* Processo filho */
        printf("Filho: %d\n", getpid());
    } else if (pid > 0) { /* Processo pai */
        pid_t pid_retorno = wait(NULL);
        printf("Pai: Terminou o processo %d\n", pid_retorno);
    } else /* < 0 -- erro */
        ERROR(1, "Erro na execução do fork()");

    return 0;
}
```

Listagem 7 - Uso da função *wait()*

Lab 5

Recorrendo ao template de projeto da UC, compile e execute o programa “lab5” utilizando o código fonte da listagem 7.

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

#include "debug.h"

int main(void) {
    pid_t pid1 = fork();
```

```

if (pid1 == 0) { /* Processo filho */
    printf("Filho 1\n");
    sleep(6);
} else if (pid1 > 0) { /* Processo pai */
    printf("Pai criou o filho 1\n");
    pid_t pid2 = fork();
    if (pid2 == 0) { /* Processo filho */
        printf("Filho 2\n");
        sleep(2);
    } else if (pid2 > 0) { /* Processo pai */
        printf("Pai criou o filho 2\n");
        printf("Pai 'a espera do filho 2\n");
        waitpid(pid2, NULL, 0);
        printf("Filho2 terminou\nPai 'a espera do filho
1\n");
        waitpid(pid1, NULL, 0);
        printf("Filho 1 terminou\nPai acabou!!!\n");
    } else /* < 0 - erro */
        ERROR(2, "Erro na execucao do fork()");
} else /* < 0 - erro */
    ERROR(1, "Erro na execucao do fork()");

return 0;
}

```

Listagem 8 - Uso da função *waitpid()*

Lab 6

Utilizando o projeto existente, compile e execute o código da listagem 8.

3. Substituição da imagem em memória de um processo

Certas situações requerem que se aceda ao resultado de um determinado comando executado via *shell*, para que esse resultado seja processado pelo nosso programa. Para que tal seja possível, torna-se necessário criar um processo novo com a função *fork()*, e de seguida, **substituir a imagem** em memória do processo por outro processo. A título de exemplo, quando se digita *date* na linha de comandos do Linux, a *shell* chama a função *fork()* e momentaneamente existem duas *shells* a executar, mas a seguir, o código da *shell* filha é substituído pelo código do programa *date* usando uma função da família *exec*.

Deste modo, a substituição da imagem de um processo faz-se com recurso às funções da família *exec*, cujos protótipos se encontram listados a seguir:

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...
          /* (char *) NULL */);
int execlp(const char *file, const char *arg, ...
          /* (char *) NULL */);
int execl_e(const char *path, const char *arg, ...
            /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

As chamadas usam a variável de ambiente *environ* que não é mais do que um *array* de *strings* que guarda informação de ambiente tal como utilizador, diretoria *home*, *path*, etc. Esta variável está referenciada no ficheiro *unistd.h* da seguinte forma:

```
extern char **environ;
```

Nos argumentos *path* e *file* define-se o caminho para o ficheiro binário que constitui a nova imagem a sobrepor.

O argumento “const char *arg, ...” corresponde a uma lista de argumentos *arg0*, *arg1*, ..., *argn*. Estes argumentos são ponteiros para *strings* e guardam os argumentos a serem usados pela nova imagem sobreposta. A lista deverá ser terminada com NULL.

O argumento *argv[]* é um *array* de ponteiros para *strings* que guardam os argumentos a serem usados pela nova imagem de programa.

O argumento *envp[]* é um *array* de ponteiros para *strings*, as quais constituem o ambiente para a nova imagem do programa. Também este *array* deverá ser terminado com NULL. Esse ambiente poderá ser baseado a partir da variável externa *environ*.

De lembrar que tanto o *arg0* como o *argv[0]* devem ter sempre o nome/caminho do executável que se pretende executar.

Na tabela que segue estão esquematizadas as diferenças entre as diversas funções da família *exec*.

| Chamada | Formato dos Argumentos | Ambiente de Trabalho | Procura na PATH |
|-----------------------|------------------------------|----------------------|-----------------|
| <code>execl()</code> | Lista | Auto | Não |
| <code>execv()</code> | Vetor de ponteiros para char | Auto | Não |
| <code>execle()</code> | Lista | Manual | Não |
| <code>execve()</code> | Vetor de ponteiros para char | Manual | Não |
| <code>execlp()</code> | Lista | Auto | Sim |
| <code>execvp()</code> | Vetor de ponteiros para char | Auto | Sim |

Tabela 1 - Funções da família *exec*

3.1. Valores de retorno

Se alguma das funções retornar, então é porque um erro ocorreu. O valor de retorno é **-1** e a variável global *errno* é preenchida.

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

#include "debug.h"

int main(void) {
    pid_t pid;
    switch (pid = fork()) {
        case -1:
            /* erro */
            ERROR(1, "Erro na execução do fork()");
            /*break; aqui não teria qualquer efeito, ERROR irá
            terminar programa */

        case 0:
            /* filho */
            /*Como o execlp() tem um no variável de parâmetros o
            * último parâmetro tem de ser sempre NULL*/
            execlp("ls", "ls", "-lF", "-a", NULL);
            ERROR(1, "erro no execlp");
            /*break; aqui não teria qualquer efeito, ERROR irá
            terminar programa */

        default:
            /* pai */
            wait(NULL);
            printf("Fim da execução do comando ls -lF -a.\n");
    }
}
```

```
        break;
    }
    return 0;
}
```

Listagem 9 - Novo processo com a imagem do comando ls -lF -a

Lab 7

Utilizando o projeto existente, compile e execute o código da listagem 9.

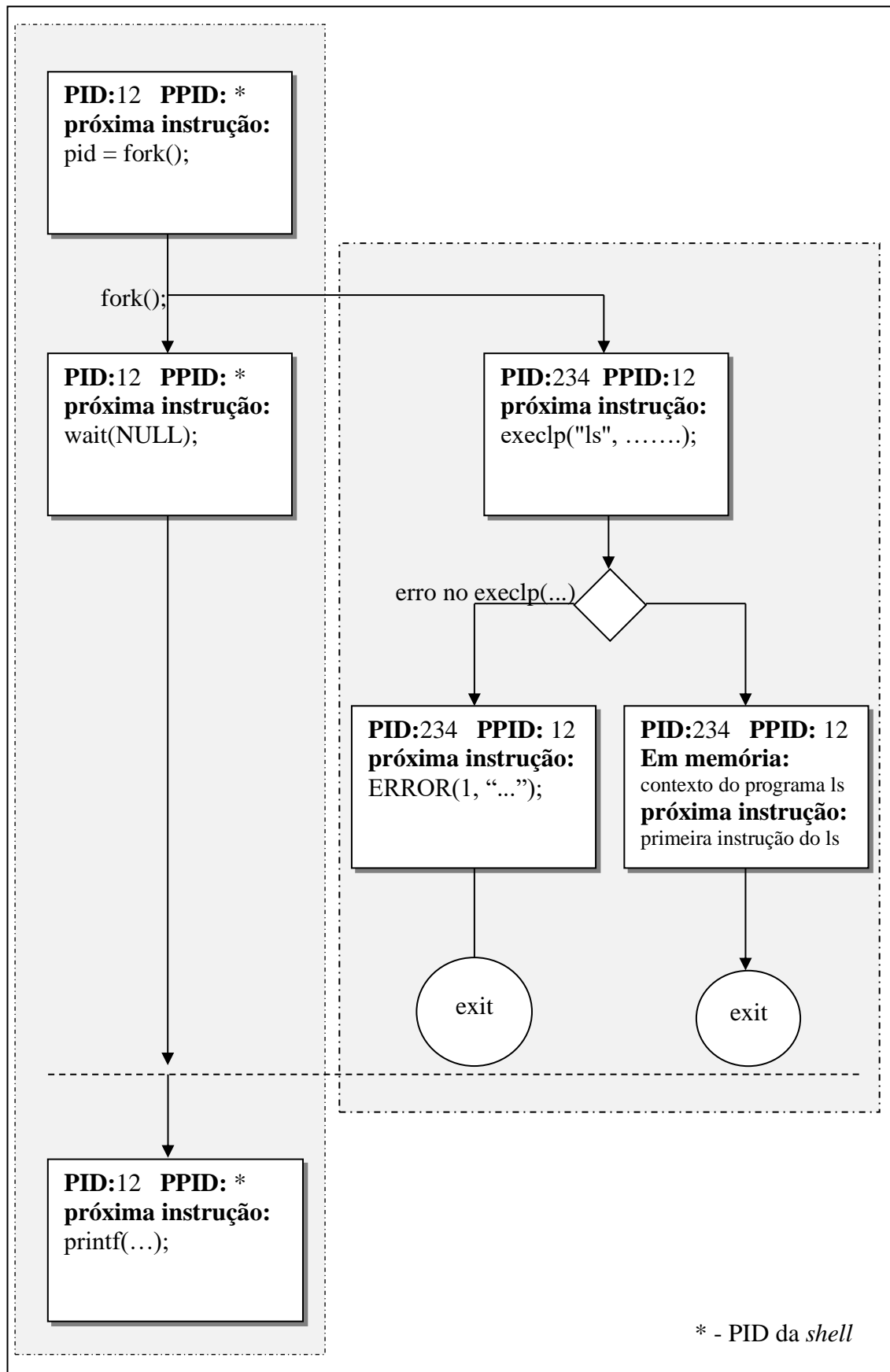


Figura 2 - Esquematização da utilização da função *execvp()* com a utilização de um *fork()*

4. A função `system`

```
#include <stdlib.h>

int system(const char *command);
```

A chamada `system` (man 3 `system`) proporciona uma forma simples de um processo executar um programa externo, sem ter necessidade de recorrer à metodologia `fork` + `exec`. De facto, a chamada ao sistema `system` tenta executar a linha de comando que lhe é passada como único argumento. Por exemplo, a execução da linha de comando “`ps -l`” pode ser feita da seguinte forma:

```
system("ps -l");
```

Dado que a *string* passada como parâmetro é interpretada como linha de comando, a função `system` permite a execução de linhas de comando que contenham múltiplos comandos ligadas por *pipes* e redireccionamento de ficheiros. Por exemplo:

```
system("ps aux | grep root");
```

4.1. Valores de retorno

Sucesso – No caso de sucesso, a função devolve o valor de retorno do último comando que foi executado.

Insucesso – `-1` se não for possível criar o processo filho (no *fork*). Para os restantes casos aconselha-se a leitura do manual (`SYSTEM(3)`).

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

#include "debug.h"

int main(void) {
    int result = system("ps -l");
    if (result < 0) {
        ERROR(1, "Chamada 'a funcao system() falhou.");
    } else {
        printf("Chamada 'a funcao system() retornou: %d.\n",
result);
    }
}
```

```
    return 0;  
}
```

Listagem 10 - Novo processo com a imagem do comando ls -lF -a

Lab 8

Utilizando o projeto existente, compile e execute o código da listagem 10 várias vezes, interpretando os resultados ao nível dos PID mostrados pela saída do comando “ps”.

5. Exercícios

Nota: na resolução de cada exercício deverá utilizar o *template* de exercícios que inclui uma *makefile* bem como todas as dependências necessárias à compilação.

5.1. Para a aula

1. Escreva o programa **CriaNProcs** que deve criar n processos, sendo que cada processo deverá escrever o seu PID e a respetiva ordem de criação. Por exemplo, o n -ésimo processo deverá escrever: Processo # n (PID= ____). O argumento n deve ser indicado através da opção **-n <numero>** ou **-num_procs <numero>**.

NOTA: deve ser empregue a ferramenta `gengetopt` para tratamento de parâmetros.

2. Escreva um programa que cria 4 processos. O processo original cria 2 filhos e depois imprime “eu sou o pai”; Os filhos imprimem “eu sou o filho 1” e “eu sou o filho 2” respetivamente. O primeiro filho cria um processo que imprime “eu sou o neto”. Cada processo deve também escrever o seu PID e PID do respetivo processo pai.
3. Repetir o exercício anterior, mas de modo a que a escrita das mensagens se processe na seguinte ordem: eu sou o neto; eu sou o filho 1; eu sou o filho 2; eu sou o pai.

5.2. Exercícios extra-aula

4. No Linux, o número máximo de descritores por processo é obtido através da chamada à função `sysconf` para obter o valor da variável de configuração `_SC_OPEN_MAX`. Elabore, recorrendo à linguagem C, o programa `get_open_max` que deve mostrar na saída padrão o número máximo de descritores por processo.
5. Usando uma das funções da família **exec** escreva o programa `ExecComandos` capaz de executar todos os comandos passados na linha de comando, pela ordem que os comandos forem passados (assuma que os comandos não possuem argumentos). Por exemplo:

```
$ ./ExecComandos ls who finger
```

6. Recorrendo à função do sistema “execvp” escreva o programa que implemente uma mini-shell com capacidades de executar comandos e avisar caso o comando a executar não exista. O programa deve indicar, em milissegundos, o tempo gasto na execução do comando. O funcionamento deste programa deverá ser o seguinte: quando for chamado o executável, deverá ficar à espera de comandos para executar até que se insira o comando terminar.

```
$ ./mini-shell
# MINI-SHELL
Comando? Who
estudante_105 pts/4      Mar 22 16:49 (192.168.232.88)
estudante_109 pts/5      Mar 22 19:26 (192.168.246.20)
Mini-shell report: comando "who" executado em 10ms
Comando?
```

7. Altere o exercício anterior de forma a ser apresentado também o tempo realmente gasto pelo CPU. Sugestão: ver a função *times*.
8. Altere de novo o exercício anterior para que o processo não fique bloqueado à medida que vai executando o comando. Neste caso, deverá mostrar o tempo decorrido ciclicamente. Sugestão: ver função *wait*, opção *wnohang* e função *nanosleep*.
9. Escreva um programa que conte o número de ficheiros contidos nas pastas indicadas na linha de comando. Por cada pasta, o programa deverá criar um processo filho que efetue a contagem dos ficheiros pertencentes à pasta e respetivas subpastas. O processo filho deverá apresentar no final o número total de ficheiros. O processo pai deve esperar por todos os filhos antes de terminar. Por exemplo:

```
$ ./ContaFicheiros /home/user /usr/include
/home/user: 20 ficheiros
/usr/include: 260 ficheiros
```

10. Recorrendo à linguagem C, elabore o programa “**run_extern**” cujo propósito é o de executar um programa externo e de mostrar na saída padrão (*stdout*) o código de término do programa externo. O “**run_extern**” deve ser capaz de distinguir situações em que o programa externo tenha terminado com código de retorno e situações em que tenha sido terminado através de um sinal.

O nome do programa externo deve ser indicado como primeiro parâmetro do “**run_extern**”, sendo que eventuais parâmetros da linha de comandos a serem passados para a execução do programa externo devem também eles serem indicados como parâmetros do run_extern. Por exemplo, para se executar o programa externo “ls -la xpto”, especificar-se-á:

run_extern ls -la xpto

- 11.** Recorrendo à programação concorrente, escreve a versão paralela de um programa que multiplique 2 matrizes. A aplicação deverá criar e preencher (com valores aleatórios entre 0 e 10) duas matrizes de tamanho 2x2. Cada processo deverá calcular cada um dos elementos da matriz resultante. Como cada elemento da matriz resultante é independente dos restantes, a multiplicação de matrizes pode ser paralelizada. No final, a aplicação deverá escrever para o ecrã as duas matrizes e a matriz resultante.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

```
$ ./multiplica
A=| 1  2 |
   | 3  4 |
B=| 5  6 |
   | 7  8 |
C=| 19 22 |
   | 43 50 |
```