Ficha 1 – Controlo de Processos

2025

{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 utiliza tem 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` para ter 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

Termo	Definição
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 descritor 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
```

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`).

Listing 1: As funções getpid() e getppid()

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

2.1.2 Lab 1

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

2.1.3 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

```c
#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 operativo [1].

> ***Nota***
>
> *Convenciona-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.).*

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.

2.3 `fork` - Cria um novo processo

```c
#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

Estado	Valor devolvido
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.

Listing 2: Criação de um processo e distinção entre processos com if ... else

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include "debug.h"
6
7  int main(void) {
8      pid_t pid = fork();
9      if (pid == 0)
10         /* Processo filho */
11         printf("PID do processo filho = %d\n", getpid());
12     else if (pid > 0)
13         /* Processo pai */
14         printf("PID do processo pai = %d, PID do processo filho = %d\n",
15                getpid(), pid);
16     else
17         /* <0 – erro*/
18         ERROR(1, "Erro na execução do fork()");
19     /* Ambos os processos executam o resto do código */
20     DEBUG("O processo %d terminou", getpid());
21     return 0;
22 }
```

Exemplo de saída do programa na Listagem 2:

```
PID do processo pai = 298819, PID do processo filho = 298820
[fork1.c@20] DEBUG - O processo 298819 terminou
PID do processo filho = 298820
[fork1.c@20] DEBUG - O processo 298820 terminou
```

Listing 3: Criação de um processo e distinção entre processos com switch

```c
#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;
}
```

Listing 4: Criação de processos com condições de proteção

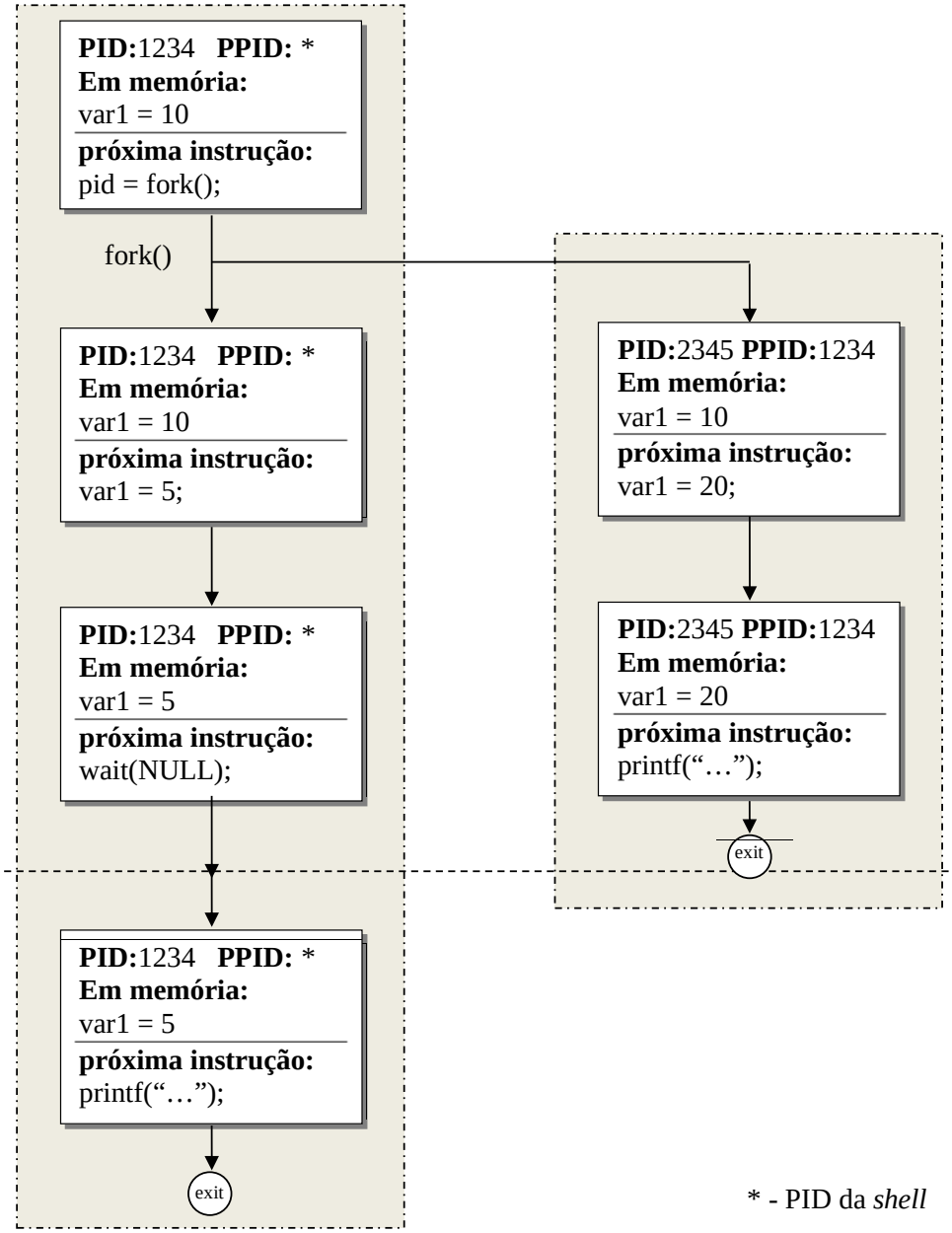
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include "debug.h"

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()");
}
```

2.3.2 Exemplo da criação de um processo



2.3.3 Lab 3

Listing 5: Criação de vários processos

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

#include "debug.h"

int main(void) {
    printf("PID=%d (%d)\n", getpid(), getppid());
    fflush(stdout);
    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());
            fflush(stdout);
        }
    }

    return 0;
}
```

Sem recorrer à execução do programa, indique quantos processos são criados com o código da Listagem 5.

> ***Nota***
>
> *Ao correr o lab3 é possível que algumas linhas de saída apareçam por baixo ou à frente do prompt da shell. Tal deve-se à ordem pela qual os processos irão correr ser indeterminada, e ao primeiro processo pai poder terminar, desbloquando a shell, antes dos restantes processos terminarem.*

2.3.4 Lab 4

Listing 6: Código-fonte do lab4

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

int main(void) {
    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();

    printf("Processo %d (%d) ", getpid(), getppid());
    printf("a = %d b = %d c = %d d = %d\n", a, b, c, d);

    return 0;
}
```

Analisando apenas o código-fonte, elabore a árvore de processos resultante da execução do código na Listagem 6, indicando para cada processo o valor de cada uma das variáveis.

2.4 `sleep` - Adormece o processo

```c
#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

```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, 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. O ponteiro `wstatus` serve 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(wstatus)` e `WTERMSIG(wstatus)` 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

Estado	**Valor devolvido**
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.

2.5.2 Lab 5

Listing 7: Código-fonte do lab5

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/wait.h>
4  #include <unistd.h>
5
6  #include "debug.h"
7
8  int main(void) {
9      pid_t pid = fork();
10     if (pid == 0) { /* Processo filho */
11         printf("Filho: %d\n", getpid());
12     } else if (pid > 0) { /* Processo pai */
13         pid_t pid_retorno = wait(NULL);
14         printf("Pai: Terminou o processo %d\n", pid_retorno);
15     } else /* < 0 -- erro */
16         ERROR(1, "Erro na execução do fork()");
17
18     return 0;
19 }
```

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

2.5.3 Lab 6

Listing 8: Código-fonte do lab6

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.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("Filho 2 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;
}
```

Recorrendo ao template de projeto da UC, compile e execute o programa “lab6” utilizando o código-fonte da Listagem 8.

3 `exec` - 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 execle(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 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`. As strings são do tipo `"nome=valor"`, por exemplo, `"HOME=/home/ana"`.

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
`execl()`	Lista	Auto	Não
`execv()`	Vetor de ponteiros para char	Auto	Não
`execle()`	Lista	Manual	Não
`execve()`	Vetor de ponteiros para char	Manual	Não
`execlp()`	Lista	Auto	Sim
`execvp()`	Vetor de ponteiros para char	Auto	Sim

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.

3.2 Lab 7

Listing 9: Código-fonte do lab7

```
 1  #include <errno.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5  #include <sys/types.h>
 6  #include <sys/wait.h>
 7  #include <unistd.h>
 8
 9  #include "debug.h"
10
11  int main(void) {
12      pid_t pid;
13      switch (pid = fork()) {
14      case -1:
15          /* erro */
16          ERROR(1, "Erro na execução do fork()");
17          /*break; aqui não teria qualquer efeito, ERROR irá terminar programa */
18
19      case 0:
20          /* filho */
21          /*Como o execlp() tem um no variável de parâmetros o
22           * último parâmetro tem de ser sempre NULL*/
23          execlp("ls", "ls", "-lF", "-a", NULL);
24          ERROR(1, "erro no execlp");
25          /*break; aqui não teria qualquer efeito, ERROR irá terminar programa */
26
27      default:
28          /* pai */
29          wait(NULL);
30          printf("Fim da execução do comando ls –lF -a.\n");
31          break;
32      }
33      return 0;
34  }
```

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

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 usando a shell `/bin/sh`,

a qual em muitos sistemas Linux redireciona para outra *shell*, tal como a bash. 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

Estado	**Valor devolvido**
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)`).

4.2 Lab 8

Listing 10: Código-fonte do lab8

```
 1  #include <errno.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5  #include <sys/types.h>
 6  #include <sys/wait.h>
 7  #include <unistd.h>
 8
 9  #include "debug.h"
10
11  int main(void) {
12      int result = system("ps -l");
13      if (result < 0) {
14          ERROR(1, "Chamada 'a funcao system() falhou.");
15      } else {
16          printf("Chamada 'a funcao system() retornou: %d.\n", result);
17      }
18
19      return 0;
20  }
```

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.*
>
> *Na UC de PA não é autorizado o uso de Variable Length Arrays, cujo uso ocorre quando o tamanho de um array é especificado usando uma variável em vez de uma constante (p.ex.* `int n = 3;int valores[n];` *). O template da UC está preparado para causar um erro de compilação quando são usados VLAs. Em vez de VLAs deve ser usada a função* `malloc`*.*

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 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 executa 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 diretoria e respetivas subdiretorias. 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 2×2. 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.

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

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

---

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