Politécnico Castelo Branco
Escola Superior de Tecnologia

Escola Superior de Tecnologia
Curso: Licenciatura em Engenharia Informática
2º Semestre do 2º Ano
Unidade Curricular: Sistemas Operativos

Ano Letivo 2024/2025

Ficha de Trabalho nº 3 – Processos em Linux

Índice

1 Objectivos 2

2 Processos em Linux 2

2.1 O que é um processo? 2

2.2 Obter o PID de um processo 3

2.3 Criar novos processos 3

2.4 Obter o PID do processo pai 4

2.5 Esperar que um processo termine a sua execução 4

2.6 Processos *Zombies* ou *Defuncts* 5

3 Execução de programas (*exec*) 7

4 Exercícios 8

1 Objectivos

- Conhecimento dos mecanismos para criação e gestão de processos em Linux
- Capacidade de construção de programas multitarefa com base na criação e gestão de processos

2 Processos em Linux

2.1 O que é um processo?

1. Execute o comando **ps**. O que faz? (***man ps***)
2. Execute agora o mesmo comando **ps** com as opções **aux**. O que mostra?
3. Copie o seguinte código (crie uma directoria **ficha3**, e aí dentro, crie o ficheiro **ex21.c**):

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

int main() {
  printf("SLEEP...\n");
  sleep(30);
  printf("THE END\n");
  return 0;
}
```

   a) Compile e execute o executável resultante em background.

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

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

4. Em baixo, são apresentados os comandos para controlar a execução dos processos:
   a) **jobs** – ver o estado dos *jobs* que estão em execução e mostra o seu estado.
   b) **fg** – passa um processo para *foreground*.
   c) **bg** – executa um processo que esteja em *background*.
5. Execute as seguintes instruções numa linha de comandos:
   a) Exercício 1:

```
./ex21
 [ctrl^Z]
ps
jobs        (verificar o índice do job ex21 e o estado)
bg <índice do job ex21>
jobs        (verificar o índice do job ex21 e o estado)
fg <índice do job ex21>
```

b) Exercício 2:

```
./ex21 &
jobs        (verificar o índice do job ex21 e o estado)
fg <índice do job ex21>
 [ctrl^C]
ps
```

2.2 Obter o PID de um processo

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

   a) Consulte o manual para a função *getpid* (***man 2 getpid***).

   b) Altere o código ex21.c de modo que, antes da chamada ao ***sleep*** seja apresentado o PID do processo em execução (ficheiro ex22.c):

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

   c) Compile e execute o programa, em *background* 3 vezes. De seguida, execute o comando **ps**.

2.3 Criar novos processos

1. A função ***fork*** (*system call*) cria um novo processo (designado por **processo filho**) que é uma cópia do processo que o criou (**processo pai**). Consulte o manual desta função (***man 2 fork***).

```
int fork()
```

   a) A chamada à função ***fork()*** **devolve um valor inteiro diferente a cada um dos processos pai e filho**:

      i) A função devolve ao **processo filho (processo que acabou de ser criado) o valor 0 (zero)**;

      ii) A função devolve ao processo **pai o valor do PID atribuído ao processo filho**.

   b) Em caso de **erro**, esta chamada à função ***fork()*** **devolve -1 ao processo pai**, o que significa que não foi possível criar um novo processo.

2. Após a criação de um novo processo, com a chamada à função ***fork()***, **tanto o processo pai como o processo filho avançam na sua execução para a linha de código (instrução) que estiver escrita imediactamente a seguir à chamada à função *fork()*.**

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

   a) Copie, compile e execute (ficheiro ex23.c).

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

int main() {
  pid_t p = fork();
  if(p == 0) {
       printf("Eu sou o processo filho.\n");
       sleep(20);
  } else if(p > 0) {
       printf("Eu sou o processo pai.\n");
       sleep(20);
  } else {
       printf("Fork error!\n");
  }
  return 0;
}
```

2.4 Obter o PID do processo pai

1. Altere o programa anterior para que cada processo apresente o seu PID e o PID do processo que o criou, ou seja, do seu processo pai (ficheiro ex24a.c). Consulte o manual (*man 2 getppid*).

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

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

2. Altere, novamente, o programa para que o processo pai mostre o PID do processo filho (ficheiro ex24b.c).

   a) Compile e execute.

2.5 Esperar que um processo termine a sua execução

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

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

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

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

Macros para determinar a forma como o processo filho terminou:

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

3. Compile e execute o seguinte código (ficheiro ex25a.c):

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

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

4. Implemente um programa (ficheiro ex25b.c) em que o processo pai cria um processo, imprime a *mensagem de apresentação,* espera que o processo filho termine a sua execução e imprime uma mensagem em que indica que o processo filho terminou (com o PID do filho). O processo filho deverá imprimir uma linha de *apresentação* ("[PID] Eu sou o filho. O meu pai é o processo PPID") e *dormir* 15 segundos.

   a) Compile e execute.

2.6 Processos *Zombies* ou *Defuncts*

Os processos pai têm a responsabilidade de esperar a terminação da execução dos seus processos filho. Este mecanismo permite aos processos pai poder obter o resultado de saída e estatísticas de utilização de recursos dos seus processos filho. Esta espera é realizada pelo processo pai usando uma chamada *wait* sobre os seus processos filho. Apenas nesse momento o processo em causa é removido do sistema. Caso o processo pai não realize a espera pelos seus processos filho, estes permanecem no estado *zombie* (ou *defunct*). Os processos *zombie* (ou *defunct*) são processos "mortos" em termos de execução, ou seja, estes processos já terminaram toda a sua execução A razão da existência de processos *zombie* deve-se ao facto dos seus processos pai ainda não terem feito a espera pela terminação destes processos, ou seja, a chamada *wait* sobre estes processos. Os processos *zombie,* praticamente, não consomem recursos, no entanto, só o facto de eles existirem significa uma sobrecarga adicional para o sistema que deve ser evitada.

A existência de processos *zombie* significa que o processo pai não ficou à espera que estes processos terminassem. Para resolver a situação o processo pai deve ser recodificado de modo que este aguarde a conclusão dos seus processos filho. Em resumo, um processo *zombie* indica que um processo terminou, mas ainda continua na tabela de processos, aguardando que o seu processo pai consulte o estado com que terminou (*exit status*). O processo *zombie* continua dependente do seu processo pai que continua "vivo".

Caso um processo pai morra, os seus processos filho são "adoptados" por um processo do sistema e continuam a sua execução normal. Ao terminarem a execução, estes processos são esperados pelo processo do sistema que os "adoptou" evitando assim que estes processos permaneçam em estado *zombies*.

1. Leia o seguinte código.

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

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

int main() {
  int status;
  pid_t p = fork();
  if(p == 0) {
       printf("[%d] Eu sou o processo filho. O meu pai é o %d.\n", getpid(), getppid());
       sleep(15);
  } else if(p > 0) {
       printf("[%d] Eu sou o processo pai.\n", getpid());
       sleep(30);
       waitpid(p, &status, 0);
       printf("O meu filho terminou.\n");
       sleep(15);
  } else {
       printf("Fork error!\n");
  }
  return 0;
}
```

   b) Copie o código e compile-o (ficheiro **ex26a.c**). Abra duas consolas antes de executar o binário. Execute-o numa consola e execute o comando **ps –a** na outra. Vá repetindo o comando **ps –a**.

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

2. Leia o seguinte código.

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

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

int main() {
  pid_t p = fork();
  if(p == 0) {
    while(1) {
       printf("[%d] Eu sou o processo filho. O meu pai é o %d.\n", getpid(), getppid());
       sleep(1);
    }
  } else if(p > 0) {
       printf("[%d] Eu sou o processo pai. \n", getpid());
       sleep(5);
  } else {
       printf("Fork error!\n");
  }
  return 0;
}
```

b) Copie o código, compile-o (ficheiro ex26b.c) e execute-o.

c) Para comunicar com processos podemos usar o envio de sinais. Para tal, devemos a partir de uma consola usar o comando ***kill***. Por exemplo, para terminar um processo que esteja em execução podemos usar a opção -9 do comando ***kill***. Esta opção permite o envio de um sinal que obriga o processo a terminar. Para mais informações consulte as páginas do manual referentes aos comandos ***kill*** e ***killall*** (*man kill* e *man killall*):

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

3 Execução de programas (*exec*)

A família de funções *exec...* é composta pelas seguintes funções:

```c
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Estas funções, quando chamadas, substituem o executável actual, associado ao processo, por um novo executável. Este novo executável, é indicado pelo seu nome nos parâmetros de entrada das funções *exec...*. As funções começadas por *execl...* podem ter vários argumentos, sendo o primeiro sempre o nome do novo executável a carregar no processo e os seguintes a lista de argumentos (incluindo novamente o nome do executável) necessários para essa execução. A lista de argumentos, numa chamada *execl...*, tem sempre de ser terminada com o marcador NULL.

A função *execlp* vai usar como fonte da procura do programa a executar, todos os caminhos definidos na variável de ambiente PATH.

Para mais informações sobre as funções da família *exec...* consulte a página do manual.

1. Teste o seguinte programa com as várias alternativas em comentário (ficheiro ex31.c).
   a) A mensagem "THE END" aparece no ecrã em situações que não se verificam erros nas chamadas às funções *exec...*? Justifique.

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

int main() {
  int res_exec;

  printf("STARTED...\n");
  res_exec = execlp("ls", "ls", "-la", NULL);
  // Alternativa:
  // res_exec = execl("/bin/ls", "ls", "-la", NULL);
  if(res_exec == -1) {
       printf("exec() error");
  }

  printf("THE END\n");
  return 0;
}
```

4 Exercícios

1. Construa um programa que aceite e execute comandos, do sistema Linux, indicados pelo utilizador. O programa deve estar sempre disponível para aceitar e executar novos comandos indicados pelo utilizador, e apenas deve terminar quando o utilizador digitar "quit".

2. Construa um programa que crie uma cadeia de NUM_PROCS processos. NUM_PROCS deve ser uma constante global > 2. O processo original (processo 1) cria um processo (processo 2), o processo 2 cria o 3, o 3 cria o 4, até existirem NUM_PROCS processos. Cada processo deve inicialmente apresentar uma mensagem com a sua identificação na hierarquia e o seu identificador. Em seguida, cada processo deve esperar que o seu filho termine, após o que deve apresentar uma mensagem indicando a terminação do seu filho.

3.
   a) Construa um programa que apresente os valores inteiros de 1 a 10. Cada processo antes de apresentar os valores inteiros deve identificar-se com o seu PID.
   b) Altere o programa anterior de modo que, o processo original (pai) após apresentar os valores inteiros entre 1 e 10 crie um processo filho, ficando o processo pai à espera da terminação deste filho. O processo filho, após ser criado, deve apresentar os valores inteiros entre 10 e 20 e em seguida terminar. O processo pai, após terminar a espera pelo processo filho, deve apresentar os valores inteiros entre 20 e 30 e em seguida terminar também.

c) Altere o programa anterior para que inclua a criação de um terceiro processo. Este novo processo deve ser criado como filho do filho.

i) O primeiro processo (pai) apresenta os valores de 1 a 10.

ii) O segundo processo (filho) apresenta os valores de 10 a 20.

iii) O terceiro processo (filho do filho) apresenta os valores de 20 a 30.

iv) O segundo processo, depois de terminar a espera pelo terceiro, apresenta de 30 a 40.

v) O primeiro processo, depois de terminar a espera pelo segundo, apresenta de 40 a 50.

4. Considere o exemplo de código seguinte:

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

int main() {
  int c = 2;
  pid_t p = 0;
  int status;

  p = fork();
  if(p == 0) {
     c = c + 5;
  } else if(p > 0) {
     c = c + 2;
     waitpid(p, &status, 0);
  } else {
     printf("Fork error!\n");
  }
  printf("c=%d\n", c);
  exit(0);
}
```

Antes de codificar, determine qual será o valor apresentado por "c"? Confirme a sua resposta executando o código. (ficheiro ex44.c)

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

a) **Antes de codificar e executar**, pense qual será o *output* dos seguintes programas:

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

int main() {
  printf("INICIO\n");
  fork();
  printf("FIM\n");
  exit(0);
}
```

(ficheiro ex45a.c)

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

int main() {
  printf("INICIO\n");
  fork();
  fork();
  printf("FIM\n");
  exit(0);
}
```

(ficheiro ex45b.c)

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

int main() {
  printf("INICIO\n");
  if(fork() == 0) {
       printf("FIM FILHO\n");
  }
  printf("FIM\n");
  exit(0);
}
```

(ficheiro ex45c.c)

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

int main() {
  printf("INICIO\n");
  if(fork() == 0) {
       printf("FIM FILHO\n");
       exit(0);
  }
  printf("FIM\n");
  exit(0);
}
```

(ficheiro ex45d.c)

b) Execute agora cada um dos programas em cima e confirme se os resultados gerados estão de acordo com os que obteve na alínea a). Caso contrário, volte a analisar o código o tente entender o resultado obtido.