ISEP – LEEC - Sistemas Computacionais 2021/2022

Ficha 2 - Programação de aplicações com múltiplos processos com a API POSIX

Objetivos

O aluno deverá ser capaz de aplicar as funções básicas relacionadas com a criação e gestão de processos (`fork`, `wait`, `waitpid`, `system`, família `exec`)
O aluno deverá ser capaz de analisar as interações existentes entre processos executados numa mesma sessão, no que respeita à gestão de memória e à execução concorrente.

Introdução

Sistemas operativos tais como o Microsoft Windows e as diversas distribuições GNU/Linux são designados por sistemas operativos **multiprocesso**. Essa designação advém do facto destes sistemas operativos permitirem executar simultaneamente vários programas, assim como várias instâncias do mesmo programa, através do mecanismo de **processo**. Neste contexto, um **processo** corresponde à execução de um **programa.**

Cada vez que o utilizador dá a ordem de execução de um programa (através do ambiente gráfico, linha de comando, etc.), é criado um novo processo para executar o programa. O programa é apenas um conjunto de instruções enquanto o processo é o mecanismo que permite gerir a execução dessas instruções.

O mecanismo de processos permite, por sua vez, que o sistema operativo faça a gestão da utilização do processador, memória e dispositivos de E/S de modo que os diversos programas possam ser executados sem entrar em conflito entre si e sem colocar em causa a estabilidade do sistema. Naturalmente, os programas irão "concorrer" entre si pelos recursos da máquina (tempo de CPU, memória disponível), contudo os dados de um processo não podem ser alterados pelos restantes programas em execução.

Nos sistemas tipo UNIX ou Microsoft Windows, cada processo é identificado por um número inteiro, o identificador de processo (**PID**, do inglês *process identifier*).

A API POSIX disponibiliza a função **`getpid`** que permitem obter o identificador do processo responsável pela execução do programa:

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

O tipo de dados `pid_t` é do tipo inteiro, mas o respetivo número de bits pode variar de sistema para sistema.

A linha de comando ("*shell*") é um programa que é executado com o mesmo nível de privilégios que os restantes programas que podemos desenvolver com a linguagem C ou outras. Quando é dada a ordem de execução de um programa, a *shell* solicita a criação de um novo processo e o programa é carregado e executado nesse novo processo.

No caso da API POSIX, essa operação é feita com base em duas funções: a função **`fork`**, que cria um *clone* do processo atual, e a função **`execve`**, que carrega e inicia a execução do programa especificado:

```
pid_t fork(void);
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

A função `fork` tem um comportamento bastante específico. Quando um processo chama esta função, o sistema operativo cria um novo processo (processo "filho"/*child process*)

que é praticamente um duplicado do processo que chama a função `fork`. O novo processo, que tem o seu próprio PID, contém uma cópia do código e dos dados do processo que o criou, sendo que ambos os processos (o processo inicial e o novo processo) continuam a sua execução na instrução após a chamada ao `fork`. No entanto, o valor de retorno da função `fork` é diferente para cada processo (é sempre 0 no novo processo) pelo que, através dessa propriedade, é possível definir num mesmo programa qual o código que deverá ser executado em cada processo. Esse procedimento é ilustrado no exercício que se segue.

Exercício 1

Considere o seguinte programa:

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4
5
6   int main(int argc, char *argv[]) {
7     pid_t r;
8
9     printf("\nProcesso %d a executar %s\n\n", getpid(), argv[0]);
10
11    r = fork(); //retorna 0 no novo processo
12
13    printf("Valor de retorno da função fork para o processo %d: %d\n\n",
14     getpid(), r);
15
16    if(r == 0) {
17      for(int i = 0; i < 10; ++i) {
18        sleep(1);
19        printf("Novo processo, com PID = %d (\"filho\" de %d)\n",
20          getpid(), getppid());
21      }
22      exit(0); //termina o novo processo
23    }
24
25    for(int i = 0; i < 10; ++i) {
26      sleep(2);
27      printf("Processo inicial (PID = %d, \"filho\" de %d).\n",
28        getpid(), getppid());
29    }
30
31    return(0);
32  }
```

Além das funções `fork` e `getpid` previamente apresentadas, este programa utiliza a função `getppid`. A função `getppid` retorna o identificador do processo "pai" do processo que a executa. A questão da "parentalidade" de processos em sistemas POSIX tem alguns detalhes adicionais que serão analisados mais à frente. No contexto deste exercício, o "pai" de um processo X é o processo que criou o processo X, i.e., o processo que fez a chamada à função `fork` responsável pela criação do processo X.

a) Compile e teste o programa. Observe que ambos os ciclos `for`, nas linhas 17 e 25, são executados em simultâneo.
b) Qual o valor de retorno da função `fork` no processo pai? Qual o significado deste valor?
c) Por que motivo o `printf` da linha 13 é executado duas vezes?
d) Qual é o processo pai do processo criado para iniciar a execução deste programa? Que tipo de programa é executado por esse processo? Confirme ou obtenha essa informação através da execução (noutro terminal) do comando `ps j` enquanto o programa faz as impressões.
e) Qual é o comportamento do programa se omitir a instrução `exit(0);` na linha 22?

A API POSIX disponibiliza um conjunto de funções que permitem a um processo aguardar que um dos seus processos filho termine. As funções, por ordem crescente de grau de controlo e de informação que permitem obter, são as seguintes:

```
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

A função `wait` bloqueia o processo, caso este tenha processos filho, até que um processo filho termine. Logo que um processo filho termine (ou caso já tenha terminado antes da chamada à função `wait`) a função `wait` retorna, sendo o seu valor de retorno o PID do filho cuja terminação foi detetada. O parâmetro `wstatus` pode ser usado para guardar o valor de retorno do processo filho numa variável. No caso de uma terminação normal, o valor de retorno do processo contém o valor de retorno do programa (valor de retorno da função `main` ou argumento da função `exit`, conforme o modo de terminação). Caso não se pretenda usar o valor de retorno do processo filho, basta passar NULL como argumento.
As funções `waitpid` e `waitid`, entre outras funcionalidades, permitem aguardar a terminação de um processo filho específico.

Exercício 2

Considere o seguinte programa:

```
1:  int i = 0;
2:
3:  int main() {
4:
5:    pid_t r = fork();
6:    if(r == 0)  {
7:      sleep(10);
8:      printf("%d: %d %d\n", getpid(), i, r);
9:      return 0;
10:   }
11:
12:   i = i + 1;
13:   wait(NULL);
14:   printf("%d: %d %d\n", getpid(), i, r);
15:
16:   return 0;
17: }
```

a) Indique a sucessão de mensagens impressas no ecrã durante a execução do programa.

b) Por que motivo a impressão da linha 8 é feita antes da impressão da linha 14 apesar da pausa de 10 segundos da linha 7?

c) Indique a sucessão de mensagens no caso da linha 9 ser omitida.

Exercício 3

O espaço de memória de cada processo é isolado dos restantes processos, independentemente do tipo de utilização da memória (variáveis globais, *stack* ou memória alocada dinamicamente na *heap*). Cada processo apresenta ao programa que está a executar um espaço de memória virtual e independente de todos os outros processos.

Considere o seguinte programa:

```
1     #define NELEM 20
2
3     int main(int argc, char *argv[]) {
4       double *dados;
5       pid_t r1, r2;
6
7       dados = malloc(NELEM*sizeof(double));
8       for(int i = 0; i < NELEM; ++i)
9         dados[i] = i;
10
11      r1 = fork();
12      if(r1 == 0) {
13        for(int i = 0; i < NELEM/2; ++i)
14          dados[i] = dados[i] * 2.0;
15        exit(0);
16      }
17
18      r2 = fork();
19      if(r2 == 0) {
20        for(int i = NELEM/2; i < NELEM; ++i)
21          dados[i] = dados[i] * 2.0;
22        exit(0);
23      }
24
25      waitpid(r1, NULL, 0);
26      waitpid(r2, NULL, 0);
27
28      for(int i = 0; i < NELEM; ++i)
29       printf("%f ", dados[i]);
30      printf("\n");
31
32      return(0);
33    }
```

Na linha 9, é armazenada em **`dados`** uma sequência de valores correspondente à progressão aritmética 0, 1, … , 19. De seguida todos os valores são multiplicados por 2, sendo que, cada metade dos elementos é processada num diferente processo (linhas 13, 14, 20 e 21).

Qual a sequência de valores impressa por este programa (linhas 28 e 29)? Por que motivo o vetor **dados** mantém os valores iniciais?

Exercício 4

A programação multiprocesso pode ser usada para tirar partido da existência dos vários núcleos de processamento da CPU (*dual-core, quad-core*, etc.). Ao dividir as tarefas de um programa por diferentes processos é possível reduzir o tempo de computação por um fator que, no limite, poderá ser diretamente proporcional ao número de núcleos de processamento da CPU. Isso acontece porque o sistema operativo executa, em paralelo, um processo em cada núcleo da CPU.

No entanto, para que os processos possam partilhar dados entre si de forma eficiente é necessário recorrer a um mecanismo designado "memória partilhada" que, tal como o nome indica, permite criar um bloco de memória partilhado entre vários processos. É possível criar blocos de memória partilhada entre um processo e todos os seus descendentes usando a função POSIX `mmap`. A função abaixo implementa essa funcionalidade:

```
#include <sys/mman.h>

void *shmalloc(size_t size) {
  return mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
}
```

Substitua, no programa do exercício anterior, a chamada à função `malloc` por uma chamada à função `shmalloc` e verifique qual o resultado da impressão.

Exercício 5

Escreva um programa que execute a seguinte sequência de ações:

- Alocar um bloco de 8 bytes da memória de dados (`char *ptr = malloc(8)`).
- Guardar a *string* "pai" no bloco alocado (use a função `strcpy`).
- Criar um processo filho que deverá escrever a *string* "filho" no bloco apontado por `ptr`, fazer a impressão do conteúdo do mesmo no ecrã e terminar.
- Aguardar a conclusão do processo filho (`waitpid`) e imprimir a *string* apontada por `ptr`.

Qual o resultado esperado?

Exercício 6 (Processos "orfão")

Quando um processo termina, os seus processos filho continuam em execução. No entanto, o sistema atribui um novo processo pai a esses processos.

De forma a verificar o funcionamento deste mecanismo, faça as seguintes alterações ao código do Exercício 1:

- Na linha 18, altere o argumento da função `sleep` para 2 (pausa de 2 segundos).
- Na linha 26, altere o argumento da função `sleep` para 1 (pausa de 1 segundo).

Após realizar as alterações, compile e teste o programa. Vai observar que o processo inicial termina ao fim de aproximadamente 10 segundos, ficando a linha de comando novamente disponível para execução de comandos, enquanto o seu processo filho continua as suas impressões. Deverá também observar que, após a terminação do processo inicial, o valor retornado pela função `getppid` no seu processo filho muda.

Use o comando `ps -e` para identificar qual o programa correspondente a esse PID.

Exercício 7 (Processos "zombie")

Por omissão, um processo não termina imediatamente quando termina a execução do respetivo programa. O processo só termina completamente quando o seu processo pai "lê"[1] o respetivo valor de retorno usando uma das funções `wait`, `waitpid` ou `waitid`. Durante esse período – desde o fim do programa até à leitura do valor de retorno do processo – diz-se que o processo está no estado *zombie* (também assinalado como *defunct* nas listagens do comando `ps`)

Na prática, quando um processo termina a execução do respetivo programa, todos os seus processos filho no estado *zombie* são eliminados do sistema. Esta terminação "automática" de processos *zombie* deve-se ao mecanismo de adoção de processos órfãos. Nesses casos, a leitura do valor de retorno do processo *zombie* é feita pelo processo adotivo (`init`, `systemd`, etc.) imediatamente após a adoção do processo. Em rigor, logo que um processo passa ao estado *zombie*, todos os seus processos filho são imediatamente adotados pelo processo adotivo do sistema.

a) Volte a executar o programa do exercício 1 e, noutro terminal, execute o comando "`ps j`" na altura em que o processo filho já terminou e o processo inicial continua as suas impressões. Identifique o processo no estado *zombie*.
b) Confirme, novamente com o comando `ps j`, que após a terminação do processo inicial, o processo no estado zombie é também eliminado.

Exercício 8

Quando um processo passa ao estado *zombie* deixa de usar tempo de CPU e a maior parte da memória que utilizava é libertada. No entanto, o seu identificador de processo continua atribuído. Tendo em conta esse aspeto, a acumulação de processos *zombie* pode, a certa atura, impedir a criação de novos processos, devido à ocupação de todos os identificadores de processo.

Uma das formas de evitar acumulação de processos *zombie* sem ser necessário que o respetivo pai fique a aguardar ou tenha de detetar a sua terminação é criar um processo "neto" para realizar a tarefa que queremos realizar no novo processo, em vez de a realizar imediatamente no processo filho:

```
r =fork();
if(r == 0) {
  r = fork();
  if(r != 0)
    exit(0);

  //executar tarefa desejada para o novo processo aqui

  exit(0)
}
waitpid(r, NULL, 0); //aguarda terminação do processo filho, que é imediata
```

[1] As funções `wait`, `waitpid` e `waitid` "acusam" (ao sistema) a leitura do valor de retorno do processo filho mesmo que não lhes seja passada nenhuma variável para receber esse valor.

De forma a melhor compreender o método descrito, analise o funcionamento do seguinte programa:

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

int main(int argc, char *argv[]) {
  pid_t r;

  printf("processo inicial: %d\n", getpid());

  r = fork();
  if(r==0) {
      printf("Novo processo, que irá terminar imediatamente: %d\n", getpid());
      r = fork();
      if(r!=0)
        exit(0);

      sleep(1);
      printf("\nPID = %d, PPID = %d\n\n", getpid(), getppid());
      sleep(3);
      printf("\nPID = %d, PPID = %d vai terminar\n\n", getpid(), getppid());
      exit(0);
  }

  sleep(2);
  system("ps -f");

  printf("\nChamada à função wait\n\n");

  wait(NULL);
  system("ps -f");
  sleep(2);
  system("ps -f");

  return(0);
}
```

Observe que, tal como desejado, o processo criado pela segunda chamada à função `fork` não fica *zombie.* O processo criado pelo primeiro `fork` só fica *zombie* até ser chamada a função waitpid, o que normalmente é feito imediatamente (as chamadas à função `sleep` só foram acrescentadas para se perceber melhor o funcionamento do programa).

Exercício 9

Escreva um programa que crie um novo processo. O processo filho deverá imprimir os seus PID e PPID e terminar. O processo pai deverá aguardar alguns instantes (e.g., 3 segundos, usando função `sleep`) e então executar o comando "`ps -f`" através da função `system`:

```
int system(const char *command); //da norma ANSI C
```

Antes de terminar, o processo pai deverá imprimir a mensagem "Vou terminar". Após a terminação do programa volte a executar o comando "`ps -f`", na *shell*. Analise os resultados.

Exercício 10

Crie uma nova versão do programa elaborado no exercício anterior em que a execução do comando "`ps -f`" é feita através duma chamada a uma das funções da família `exec` (e.g., `execlp`), em substituição da chamada à função `system`.

Ao executar o programa vai observar que a mensagem "Vou terminar" deixa de aparecer no ecrã. A que é que se deve este comportamento?

Nota: pode comparar as várias funções da família `exec` escrevendo o seguinte comando na *shell*: `man 3 exec`

Exercício 11

Considere o seguinte programa:

```
void fun1(int *d) {
  ++d[0];
  printf("Novo valor gerado em fun1: %d\n", d[0]);
  sleep(2);
}

void fun2(int *d) {
  sleep(1);
  printf("Valor processado por fun2: %d\n", d[0]);
}

int main(){
  int dados = 0;

  while(1) {
    fun1(&dados);
    fun2(&dados);
  }
}
```

Altere o programa de modo que a chamada à função `fun2` seja feita num novo processo logo que a chamada a `fun1` termina. Adicionalmente, o processo pai não deverá aguardar pela conclusão da execução de `fun2` para executar novamente `fun1`.

Exercício 12

Considere o seguinte programa:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include <sys/time.h>
5   #include <float.h>
6
7   //NELEM must be a multiple of 2
8   #define NELEM 20000000
9   #define NITER 30
10
11  void rectangular2polar(double *, double *, int);
12  double mytime();
13
14  int main() {
15    double t0;
16
17    double *dados_in = (double *) malloc(sizeof(double)*NELEM);
18    double *dados_out = (double *) malloc(sizeof(double)*NELEM);
19    if(dados_in == NULL || dados_out == NULL) {
20      perror("malloc");
21      exit(1);
22    } else {
23      //assign random initial values
24      double maxv = DBL_MAX;
25      srand48(time(NULL));
26      for(int i = 0; i < NELEM; ++i)
27        dados_in[i] = 2*(drand48()-0.5)*maxv;
28    }
29
30    //get current time, for benchmarking
31    t0 = mytime();
32
33    //This cycle is used only for benchmarking purposes
34    for(int j=0; j < NITER; ++j)
35      rectangular2polar(dados_out, dados_in, NELEM);
36
37    printf("Computation took %.1f s\n", mytime() - t0);
38
39    return 0;
40  }
41
42  double mytime() {
43    struct timeval tp;
44    struct timezone tzp;
45
46    gettimeofday(&tp,&tzp);
47    return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
48  }
```

O processamento principal deste programa é feito na função **`rectangular2polar`** (chamada na linha 35), que processa os valores em **`dados_in`** e guarda os respetivos resultados em **`dados_out`**. Esta função está definida no ficheiro **`ex12-math.c`**.
A função `mytime` é usada para obter um valor aproximado do tempo de execução da função **`rectangular2polar`**, cuja chamada é repetida várias vezes (ciclo `for` da linha 34) de forma a diminuir o impacto do erro de medida desse tempo.

a) Compile o programa através do seguinte comando:

```
cc -o ex12 ex12-main.c ex12-math.c -lm -O3
```

b) Teste o programa e altere o valor de NITER de modo a obter um tempo de execução de aproximadamente 10 segundos.

c) Altere o programa anterior de modo que o processamento dos elementos em **`dados_in`** pela função **`rectangular2polar`** seja dividido por 30 processos. Não repita o código 30 vezes, use um ciclo `for` para criar os processos e outro ciclo `for` para aguardar que cada um deles termine. A função **`rectangular2polar`** não deve ser alterada.
   Compile e teste o programa. O valor de NITER deverá ser o mesmo que usou na alínea b). Relacione o tempo obtido com o número de núcleos de processamento identificados pelo comando `lscpu`.