# FOLHA DE PROBLEMAS Nº 3

## Criação e Terminação de Processos

---

### 1. – Criação de processos. *PID* de um processo. Comunicação básica entre processos.

Considere o seguinte programa (*):

```
// PROGRAMA p1.c
#include ...

int global=1;

int main(void) {
  int local = 2;
  if(fork() > 0) {
    printf("PID = %d; PPID = %d\n", getpid(), getppid());
    global++;
    local--;
  } else {
    printf("PID = %d; PPID = %d\n", getpid(), getppid());
    global--;
    local++;
  }
  printf("PID = %d - global = %d ; local = %d\n", getpid(), global, local);
  return 0;
}
```

**a)** Compile o programa, execute-o e interprete os resultados. Identifique, pelo nome, o processo-pai de cada um dos processos envolvidos. Identifique a forma de comunicação entre processos utilizada neste exemplo e em que sentido(s) (pai->filho, filho->pai) ela funciona.

**b)** Execute o programa diversas vezes e interprete as alterações que acontecem na saída.

### 2. – Criação de processos. Quem corre primeiro, o processo-pai ou o processo-filho?

Considere o seguinte programa (*):

```
// PROGRAMA p2.c
#include ...

int main(void) {
 write(STDOUT_FILENO,"1",1);
 if(fork() > 0) {
   write(STDOUT_FILENO,"2",1);
   write(STDOUT_FILENO,"3",1);
 } else {
   write(STDOUT_FILENO,"4",1);
   write(STDOUT_FILENO,"5",1);
 }
 write(STDOUT_FILENO,"\n",1);
 return 0;
}
```

(*)- para mais fácil interpretação do código não foram feitos testes de erro de execução nas chamadas `fork` e/ou `write`; na prática corrente, estes testes devem ser sempre efectuados.

**a)** Analise o programa e, antes de o executar, identifique todos números que podem ser escritos no écrã como resultado da sua execução.

**b)** Compile e teste o programa.

**c)** Substitua as chamadas `write` por chamadas `printf` (sem o carácter *newline* no fim de cada *string*) e interprete os resultados.

**d)** Idem, substituindo `print("1")` por `print("1\n")`.

## 3. – Escalonamento de processos (execução interlaçada).

Considere o seguinte programa:

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

#define MAX 50000

int main(void) {
 int i;
 pid_t pid;
 char str[10];

 pid=fork();
 switch (pid) {
        case -1:
              perror("fork");
              break;
        case  0: //filho
              for (i=1; i<=MAX; i++) {
                    sprintf(str,"-%d",i);
                    write(STDOUT_FILENO,str,strlen(str));
              }
              break;
        default: //pai
              for (i=1; i<=MAX; i++) {
                    sprintf(str,"+%d",i);
                    write(STDOUT_FILENO,str,strlen(str));
              }
 }
 return 0;
}
```

Compile o programa e execute-o (dê o comando `./p3 | more`). Verifique que os processos executam as suas instruções em sequências alternadas.

## 4. – Sincronização básica entre processos (1)

Pretende-se que dois processos, pai e filho, escrevam conjuntamente, no écran, a frase "Hello world!". Resolva o problema de duas formas diferentes:

**a)** o processo-filho escreve "Hello" e o processo-pai escreve "world !";

**b)** o processo-pai escreve "Hello" e o processo-filho escreve "world !". Execute o programa diversas vezes e interprete os resultados.

## 5. – Sincronização básica entre processos (2)

Escreva um programa que crie três processos que escrevam conjuntamente a frase "Hello my friends!". Cada processo só deve escrever uma das palavras da frase.

## 6. – Criação de uma "família" de processos. Estado dos processos: processos *zombie*.

Considere o programa seguinte em que um processo lança três processos-filhos em execução:

```
// PROGRAMA p6.c

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

int main(void)
{
 pid_t pid;
 int i, j;

 printf("I'm process %d. My parent is %d.\n", getpid(),getppid());
 for (i=1; i<=3; i++) {
       pid = fork();
       if ( pid < 0) {
             printf("fork error");
             exit(1);
       }
       else if (pid == 0) {
             printf("I'm process %d. My parent is %d.
                     I'm going to work for 1 second ...\n",
                     getpid(),getppid());
             sleep(1); // simulando o trabalho do filho
             printf("I'm process %d. My parent is %d.
                     I finished my work\n", getpid(),getppid());

             exit(0);  // a eliminar na alinea c)
       }
       else            // simulando o trabalho do pai
             for (j=1; j<=10; j++) {
                   sleep(1);
                   printf("father working ...\n");
             }
 }
 exit(0);
}
```

**a)** Compile o programa e execute-o. Numa outra janela de terminal execute, repetidas vezes, o comando `ps u`. Verifique a existência de processos *zombie*.

**b)** Modifique o programa por forma a evitar a existência de processos *zombie*, mas de modo a que o processo-pai não fique bloqueado à espera que os seus filhos terminem.

**c)** Procure antever o que acontecerá se a chamada `exit(0)`, assinalada no código, for eliminada. Faça a alteração e verifique se o resultado é o esperado. (**NOTA MUITO IMPORTANTE**: nesta situação, não altere o número de iterações do ciclo `for` para um número superior a 4 ou 5 ; devido ao crescimento exponencial do número de processos com o número de iterações, a realização de um número não muito elevado de iterações pode facilmente esgotar o número máximo de processos que tem permissão para executar; esse valor é especificado por `RLIMIT_NPROC` que poderá obter usando a chamada `getrlimit`).

## 7. – Substituição do código de um processo: as chamadas `exec`.

Considere o programa seguinte:

```
// PROGRAMA p7.c
#include ...
int main(int argc, char *argv[])
```

```
{
   char prog[20];

   sprintf(prog,"%s.c",argv[1]);
   execlp("gcc","gcc",prog,"-Wall","-o",argv[1],NULL);
   printf(...);
   exit(...);
}
```

Explique o que faz o programa. Complete as chamadas `printf` e `exit` com valores adequados.

### 8. – O processo-filho executa um código diferente do executado pelo processo-pai.

Considere o programa seguinte:

```
// PROGRAMA p8.c

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

int main(int argc, char *argv[], char *envp[])
{
   pid_t pid;
   if (argc != 2) {
     printf("usage: %s dirname\n",argv[0]);
     exit(1);
   }
   pid=fork();
   if (pid > 0)
     printf("My child is going to execute command
            \"ls -laR %s\"\n", argv[1]);
   else if (pid == 0){
     execlp(...);
     printf("Command not executed !\n");
     exit(1);
   }
   exit(0);
}
```

**a)** Complete a chamada `execlp` por forma a que o processo-filho faça o que o processo-pai anuncia que ele vai fazer. Compile o programa e execute-o.

**b)** Idem, por forma a usar `execl`. Para determinar o directório onde está o executável `ls`, pode usar o comando da *shell* `which ls`.

**c)** Idem, por forma a usar `execvp`.

**d)** Idem, por forma a usar `execv`.

**e)** Idem, por forma a usar `execve`. Considere que as variáveis de ambiente do processo-filho devem ter o mesmo valor que as do pai.

### 9. – Análise do estado de terminação de um processo-filho.

**a)** Modifique o programa por forma a que o processo-pai espere pela terminação do processo-filho e mostre o seu código de terminação (*exit code*). Execute o programa em três situações diferentes

- o argumento é um directório existente
- o argumento é um directório inexistente
- o argumento é um directório existente, com muitos ficheiros, e o processo-filho foi terminado recorrendo ao comando `kill` (para ter tempo de dar este comando, deverá indicar como argumento um directório que contenha muitos ficheiros)

e verifique se o código de terminação é o esperado.

**b)** Modifique novamente o programa por forma a que o processo-pai, além de mostrar o código de terminação do processo-filho, distinga se este terminou normalmente (uma das duas primeiras situações) ou anormalmente (última situação). Sugestão: use as macros `WIFEXITED` e `WIFSIGNALED`.

### 10. – Redireccionamento de entrada/saída do processo-filho.

Pretende-se que o programa do problema 8 possa a aceitar, na linha de comando, um segundo argumento, opcional, indicando o nome de um ficheiro onde o utilizador pretende guardar a listagem gerada pelo processo-filho. Se este argumento for fornecido, o ficheiro deve ser criado pelo processo-filho e a *standard output* do processo-filho deve ser redireccionada para esse ficheiro, antes de executar a chamada `exec`. Proceda às alterações necessárias ao programa 8, por forma a cumprir este objectivo.

### 11. – Mini-interpretador de comandos.

Escreva um mini-interpretador de comandos, cujo prompt seja "`minish >`", que execute um ciclo em que aceite, do teclado, um comando com possíveis parâmetros e o execute. Qualquer um dos comandos poderá ter no fim da sua lista de parâmetros `-o <file>`. Quando isto acontecer a saída standard do comando deverá ser redireccionada para `file` sendo estes dois parâmetros retirados da lista de parâmetros que é passada ao comando. Um novo *prompt* só deverá aparecer após conclusão da execução do comando anterior. O mini-interpretador deverá terminar quando o utilizador escrever o comando `quit`.

### 12. – Aplicação multi-processo.

Escreva um programa que copie, "em paralelo", todos os ficheiros regulares de um directório para outro já existente. Os directórios devem ser passados como argumentos do programa. Para cada ficheiro comum (não para subdirectórios) deverá ser criado um novo processo que o copia do directório original para o directório de destino. Sugestão: recorra ao código já desenvolvido anteriormente para determinar os ficheiros comuns de um directório e para copiar um ficheiro para outro.