Gestão de Processos
(usando a API do kernel)

---

1. Considere o seguinte programa que faz múltiplas chamadas à função `fork()`. Compile o programa e execute-o. Quantos processos, incluindo o processo pai, são criados? Porquê?

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

int main(int argc, char* argv[]) {
  fork();
  fork();
  fork();
  exit(EXIT_SUCCESS);
}
```

Confirme o seu palpite, alterando o programa por forma a que cada processo imprima o seu identificador (`PID`). Sugestão: veja a função `getpid()`.

2. Considere ainda este outro programa que também usa a função `fork()`. Compile-o e execute-o. Quantos processos, incluindo o processo pai, são criados? Porquê?

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

int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++)
    fork();
  exit(EXIT_SUCCESS);
}
```

Confirme de novo o seu palpite, alterando o programa por forma a que o valor do `PID` de cada processo seja imprimido.

3. Considere agora o seguinte programa que cria um processo filho. Como explica o valor da variável `value` obtido por pai e filho? Sugestão: faça um desenho que represente os espaços de endereçamento antes e após o `fork()`. O que acontece à dita variável?

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

int main(int argc, char* argv[]) {
  int value = 0;

  pid_t pid = fork();
  if (pid == -1) { perror("fork"); exit(EXIT_FAILURE); }

  if (pid == 0) {
    /* child process */
    value = 1;
    printf("CHILD: value = %d, addr = %p\n", value, &value);
    exit(EXIT_SUCCESS);
  }
  else {
    /* parent process */
    int retv = waitpid(pid, NULL, 0);
    if (retv == -1) { perror("waitpid"); exit(EXIT_FAILURE); }
    printf("PARENT: value = %d, addr = %p\n", value, &value);
    exit(EXIT_SUCCESS);
  }
}
```

Observe os valores e endereços da variável `value` imprimidos pelos processos pai e filho. Como explica os resultados?

Nota: a função `perror()` imprime a "string" indicada como argumento, seguida da "string" com o erro correspondente ao valor da variável `errno` cujo valor é fixado pelo "kernel" antes de retornar da chamada ao sistema com resultado `-1`.

4. Considere o seguinte programa que cria um processo filho que depois executa um comando fornecido na linha de comando. Compile-o e execute-o. Leia com atenção o código e compreenda como funciona.

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

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

int main(int argc, char* argv[]) {
  /* fork a child process */
  pid_t pid = fork();
  if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  if (pid == 0) {
    /* child process */
    int retv = execlp(argv[1],argv[1],NULL);
    if (retv == -1) { perror("execlp"); exit(EXIT_FAILURE); }
  }
  else {
    /* parent process */
    int retv = waitpid(pid, NULL, 0);
    if (retv == -1) { perror("waitpid"); exit(EXIT_FAILURE); }
  }
  exit(EXIT_SUCCESS);
}
```

Se a função `execlp` executa com sucesso, como é que o processo filho sinaliza o seu término ao processo pai?

5. Considere o seguinte programa que implementa uma "shell" muito simples. Compile-o e execute-o. Leia com atenção o código e compreenda como funciona.

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

int main(int argc, char* argv[]) {
  for( ; ; ) {
    /* give prompt, read command and null terminate it */
    fprintf(stdout, "$ ");
    char  buf[1024];
    char* command = fgets(buf, sizeof(buf), stdin);
    if(command == NULL)
```

```
      break;
    command[strlen(buf) - 1] = '\0';
    /* call fork and check return value */
    pid_t pid = fork();
    if(pid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
    }
    if(pid == 0) {
      /* child */
      int retv = execlp(command, command, (char *)0);
      if(retv == -1) {
        perror("execlp");
        exit(EXIT_FAILURE);
      }
    }
    /* shell waits for command to finish before giving prompt again */
    int retv = waitpid(pid, NULL, 0);
    if (retv == -1) {
      perror("waitpid");
      exit(EXIT_FAILURE);
    }
  }
  exit(EXIT_SUCCESS);
}
```

Porque é que não é possível executar comandos com argumentos, e.g., `ls -l` ou `uname -n`? Altere o programa anterior por forma a incluir um comando `exit` que termine com a shell.

6. Altere o programa anterior por forma a que os comandos possam ser executados com argumentos. Sugestão: use a função `execvp` (em vez de `execlp`) e use a a função `strtok` para separar o nomes do comando e os respectivos argumentos dados na "string" `command`.

7. Altere o programa anterior por forma a manter uma história dos comandos por ela executados. Implemente um comando `myhistory` que recebe um inteiro `n` como argumento e imprime os últimos `n` comandos executados pela shell. Sugestão: aproveite o comando `tail` da Bash shell.