L:EIC / SO2122:
Gestão de Processos
(usando a API do Kernel)

Q1. 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 a child process */
  fork();

  /* fork another child process */
  fork();

  /* and fork another */
  fork();

  return EXIT_SUCCESS;
}
```

Confirme o seu palpite, alterando o programa por forma a que o valor do `pid` de cada processo criado por uma chamada a `fork()` seja imprimido (veja a função `getpid()`).

Q2. 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();
  return EXIT_SUCCESS;
}
```

Confirme de novo o seu palpite, alterando o programa por forma a que o valor do `pid` de cada processo criado por uma chamada a `fork()` seja imprimido.

Q3. 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[]) {
  pid_t pid;
  int value = 0;

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

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

Q4. 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[]) {
  pid_t pid;

  /* fork a child process */
  if ((pid = fork()) == -1) {
    perror("fork");
    return EXIT_FAILURE;
  } else if (pid == 0) {
    /* child process */
    if (execlp(argv[1],argv[1],NULL) == -1) {
      perror("execlp");
      return EXIT_FAILURE;
    }
  } else {
    /* parent process */
    if (waitpid(pid, NULL, 0) == -1) {
      perror("waitpid");
      return EXIT_FAILURE;
    }
    printf("child exited\n");
  }
  return EXIT_SUCCESS;
}
```

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

Q5. 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[]) {
  char  buf[1024];
  char* command;
  pid_t pid;

  /* do this until you get a ^C or a ^D */
  for( ; ; ) {

    /* give prompt, read command and null terminate it */
    fprintf(stdout, "$ ");
    if((command = fgets(buf, sizeof(buf), stdin)) == NULL)
      break;
    command[strlen(buf) - 1] = '\0';

    /* call fork and check return value */
    if((pid = fork()) == -1) {
      fprintf(stderr, "%s: can't fork command: %s\n",
              argv[0], strerror(errno));
      continue;
    } else if(pid == 0) {
      /* child */
      execlp(command, command, (char *)0);
      /* if I get here "execlp" failed */
      fprintf(stderr, "%s: couldn't exec %s: %s\n",
              argv[0], buf, strerror(errno));
      /* terminate with error to be caught by parent */
      exit(EXIT_FAILURE);
    }

    /* shell waits for command to finish before giving prompt again */
    if ((pid = waitpid(pid, NULL, 0)) < 0)
      fprintf(stderr, "%s: waitpid error: %s\n",
              argv[0], strerror(errno));
  }
  exit(EXIT_SUCCESS);
}
```

Note o uso alternativo (à função `perror()`) da função `strerror()` para compreender o motivo da falha da chamada ao sistema. Esta função retorna a "string" com o erro correspondente ao valor da variável `errno` cujo valor é fixado pelo kernel antes de retornar da chamada ao sistema falhada (com `-1`). A "string" pode ser incluida em mensagens de erro mais ricas imprimidas com funções de I/O (no caso, `fprintf()`).

Porque é que não é possível executar comandos com argumentos, e.g., `ls -l` ou `uname -n`?

Q6. Altere o programa anterior por forma a que os comandos possam ser executados com argumentos. Sugestão: veja a página de manual das funções da família `exec`. Estas funções podem receber, para além de um comando, um número variável de argumentos. Poderá recolher esses argumentos da linha lida pela shell usando, por exemplo, a função `strtok` da Standard C Library.

Q7. 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: utilize a sequência `fork()-exec()` como implementada nesta shell para executar o comando; veja o que faz o comando da Bash shell `tail` (já o viu no primeiro conjunto de exercícios - "ficha 0").

Q8. Finalmente, altere novamente o programa anterior por forma a incluir um comando `exit` que termine com a shell.