Sistemas Operacionais
Roteiro de Laboratório 2

1 Objetivos

O objetivo deste laboratório é ter um contato prático com os comandos para criação de processos do Linux.

2 A API para criação de processos no UNIX

Para se criar um novo processo a partir de outro já em execução, o padrão POSIX define a função `fork()`. Uma implementação típica que utiliza essa função pode ser vista abaixo.

```
#include <stdio.h>
#include <unistd.h>     // For the syscall functions.

int main() {
    pid_t pid = fork(); // Fork a child process.
    if (pid < 0) { // Error occurred.
        fprintf(stderr, "Fork failed!\n");
        return 1;
    } else if (pid == 0) { // Child process.
        printf("[CHILD]: PID: %d - PPID: %d\n", getpid(), getppid());
    } else { // Parent process.
        printf("[PARENT]: PID: %d - PPID: %d\n", getpid(), getppid());
    }
    return 0;
}
```

No momento de execução do `fork()`, um novo processo filho é criado, e a seguir, tanto o pai quanto o filho seguem a execução *concorrentemente* do mesmo código. No entanto, o valor de retorno do `fork()` muda conforme o processo. No processo filho o retorno é zero. No processo pai o retorno é o PID (*Process ID*) do filho. Compilando e executando esse programa, obtemos o seguinte resultado (PPID – *Parent Process ID*):

```
$ gcc -o fork0 fork0.c
$ ./fork0
[PARENT]: PID: 17607 - PPID: 17593
[CHILD]: PID: 17608 - PPID: 17607
$ ps
  PID TTY          TIME CMD
17593 pts/0    00:00:00 bash
17609 pts/0    00:00:00 ps
```

O processo filho recém criado é um *clone* do seu pai, o que significa que toda a área de memória do filho é copiada do pai, inclusive a área de texto (comandos) e de dados. Qualquer

variável declarada no processo pai vai ter o seu valor copiado para o filho no momento da invocação do `fork()`. (Mais informações em: `man 2 fork`.)

Diferentes estratégias podem ser aplicadas quando processos relacionados estão trabalhando juntos. Por exemplo: os processos podem executar de forma independente e cada um realiza uma tarefa, ou o pai ficar parado esperando pelos seus filhos e usar os resultados destes. Essa segunda estratégia é implementada com o uso da função `wait()` no processo pai. Um código de exemplo pode ser visto a seguir.

```
#include <stdio.h>
#include <unistd.h>     // For the syscall functions.
#include <sys/wait.h>   // For wait and related macros.

int main() {
    pid_t pid = fork(); // Fork a child process.
    if (pid < 0) { // Error occurred.
        fprintf(stderr, "Fork failed!\n");
        return 1;
    } else if (pid == 0) { // Child process.
        printf("[CHILD]: I'm finished.\n");
        return 42;
    } else { // Parent process.
        printf("[PARENT]: Waiting on child.\n");
        int wstatus;
        wait(&wstatus);
        if (WIFEXITED(wstatus)) {
            printf("[PARENT]: Child returned with code %d.\n",
                   WEXITSTATUS(wstatus));
        }
    }
    return 0;
}
```

A função `wait()` suspende a execução do processo pai até que o filho termine. Quando isso acontece, a variável `wstatus` é preenchida com uma série de informações. Essa variável inteira corresponde a uma série de *flags* binárias, portanto para determinar se alguma *flag* foi marcada, é necessário o uso de macros. Por exemplo, a macro `WIFEXITED` retorna verdadeiro se o processo filho terminou normalmente, através de uma chamada para `_exit()` ou retorno da função `main()`. (Mais informações em: `man 2 wait`.) Compilando e executando esse exemplos, temos o seguinte resultado:

```
$ gcc -o fork1 fork1.c
$ ./fork1
[PARENT]: Waiting on child.
[CHILD]: I'm finished.
[PARENT]: Child returned with code 42.
```

Para trocar o código binário do processo filho para algo diferente do código do programa principal (pai), o programador deve usar alguma função da família `exec()`, que carrega o arquivo binário passado como argumento como a imagem do processo filho. Exemplo:

```c
#include <stdio.h>
#include <unistd.h>     // For the syscall functions.
#include <sys/wait.h>   // For wait and related macros.

int main() {
    pid_t pid = fork(); // Fork a child process.
    if (pid < 0) { // Error occurred.
        fprintf(stderr, "Fork failed!\n");
        return 1;
    } else if (pid == 0) { // Child process.
        printf("[CHILD]: About to load command.\n");
        execlp("/usr/bin/ls", "ls", "-la", (char*) NULL);
    } else { // Parent process.
        printf("[PARENT]: Waiting on child.\n");
        wait(NULL);
        printf("[PARENT]: Child finished.\n");
    }
    return 0;
}
```

No código acima, a função `execlp` carrega o programa binário `/usr/bin/ls` e o executa com os argumentos `ls` e `-la`. (Lembre que por convenção do C, o primeiro argumento é sempre o nome do executável.) É importante destacar que a lista de argumentos *deve* ser terminada por um ponteiro `NULL`, e que esse ponteiro deve sofrer *cast* para `char*`. (Mais informações em: `man 3 exec`.) Note também que neste exemplo a função `wait` recebeu um ponteiro nulo, indicando que o processo pai não está interessado no status de retorno do filho.

```
$ gcc -o fork2 fork2.c
$ ./fork2
[PARENT]: Waiting on child.
[CHILD]: About to load command.
total 36
drwxr-xr-x 2 zambon zambon 4096 set 12 14:56 .
drwxr-xr-x 4 zambon zambon 4096 set 12 13:14 ..
-rw-rw-rw- 1 zambon zambon  475 set 12 13:42 fork0.c
-rw-rw-rw- 1 zambon zambon  696 set 12 14:35 fork1.c
-rwxr-xr-x 1 zambon zambon 8696 set 12 14:56 fork2
-rw-rw-rw- 1 zambon zambon  617 set 12 13:55 fork2.c
-rw-rw-rw- 1 zambon zambon  590 set 12 14:10 fork3.c
[PARENT]: Child finished.
```

Antigamente no UNIX, uma chamada de `fork()` era bastante demorada pois exigia a cópia de toda a área de memória do processo pai para o processo filho. Por conta disso, foi criada uma função `vfork()` que não realiza essa cópia. Após a execução de `vfork()` o filho deve imediatamente chamar a função `exec()`. No entanto, nas implementações atuais do UNIX (Linux, BSD, etc), a função `fork()` é muito mais eficiente, pois evita qualquer cópia desnecessária da memória. Por conta disso, o uso de `vfork()` não é mais recomendado. (Mais informações em: `man 2 vfork`.)

O último exemplo dessa seção trata de processos *zumbis*. Um processo filho é um zumbi quando ele termina a sua execução mas o processo pai ainda não realizou uma chamada de `wait()`. Isso pode ser visto no código abaixo.

```
#include <stdio.h>
#include <unistd.h>     // For the syscall functions.
#include <sys/wait.h>   // For wait and related macros.

int main() {
    pid_t pid = fork(); // Fork a child process.
    if (pid < 0) { // Error occurred.
        fprintf(stderr, "Fork failed!\n");
        return 1;
    } else if (pid == 0) { // Child process.
        printf("[CHILD]: I'm done already.\n");
    } else { // Parent process.
        printf("[PARENT]: Going to sleep.\n");
        sleep(20); // Sleep for 20 seconds.
        printf("[PARENT]: Child finished, hopefully?!?.\n");
    }
    return 0;
}
```

Compilando e executando, o terminal fica bloqueado por 20 segundos.

```
$ gcc -o fork3 fork3.c
$ ./fork3
[PARENT]: Going to sleep.
[CHILD]: I'm done already.
```

Se abrirmos um outro terminal e executarmos o comando `ps -la`, vemos o seguinte:

```
$ ps -la
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 18263 17593  0  80   0 -  1047 hrtime pts/0    00:00:00 fork3
1 Z  1000 18264 18263  0  80   0 -     0 -      pts/0    00:00:00 fork <defunct>
0 R  1000 18265 18256  0  80   0 -  6835 -      pts/1    00:00:00 ps
```

Os estados dos processos são mostrados na coluna `S`; o processo com estado `Z` é um zumbi.

3 Exercícios de fixação

1. Qual é o valor exibido pelo processo pai no programa abaixo? Justifique sua resposta.

   ```
   #include <stdio.h>
   #include <unistd.h>     // For the syscall functions.
   #include <sys/wait.h>   // For wait and related macros.
   ```

```
int value = 5;

int main() {
    pid_t pid = fork(); // Fork a child process.
    if (pid == 0) { // Child process.
        value += 15;
        printf("[CHILD]: value = %d\n", value);
    } else if (pid > 0) { // Parent process.
        wait(NULL);
        printf("[PARENT]: value = %d\n", value);
    }
}
```

2. Contando com o processo pai inicial, quantos processos são criados pelo programa abaixo? Justifique sua resposta.

```
#include <stdio.h>
#include <unistd.h>     // For the syscall functions.

int main() {
    printf("%d\n",getpid());
    fork();
    printf("%d\n",getpid());
    fork();
    printf("%d\n",getpid());
    fork();
    printf("LAST: %d\n",getpid());
}
```

4 Comunicação simples entre processos através de sinais

A forma mais simples de comunicação entre processos é feita através de sinais. O SO utiliza sinais o tempo todo para "conversar" com os processos de usuário. Um programador pode utilizar a função `signal()` para instalar funções (*handlers*) para um ou mais sinais. A exceção é o sinal 9 (SIGKILL) que não pode ser reprogramado e sempre mata o processo imediatamente. Um exemplo simples do uso de sinais é dado abaixo:

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

unsigned long i = 0;

// Custom SIGINT (Ctrl+C) signal handler.
void my_handler(int sig_no) {
    printf("Process %d killed after %ld iterations.\n", getpid(), i);
    exit(0);
}
```

```
int main(void) {
    signal(SIGINT, my_handler); // Setting a new handler for SIGINT.
    printf("Process %d started.\n", getpid());
    // Infinite loop - SIGINT aborts.
    while(1) {
        i++; // No output by printf(), etc: CPU-oriented process.
    }
}
```

Exemplo de compilação e execução.

```
$ gcc -o sig sig.c
$ ./sig
Process 19062 started.
^CProcess 19062 killed after 412824647 iterations.
```

5 Exercícios de programação

IMPORTANTE: os exercícios desta seção NÃO devem ser enviados para o AVA. Veja a seção Tarefa, a seguir.

1. Faça o *download* no AVA dos arquivos de exemplos. Compile-os e execute-os como indicado acima e discutido pelo professor no início da aula.

2. Altere o arquivo `fork0.c` para fazer com que tanto o processo pai como o filho fiquem em *loop* infinito, exibindo os seus PIDs e PPIDs. Dessa forma você terá um modelo típico de processos orientados a I/O. Realize algumas observações (por exemplo, usando o comando `top`) sobre como o tempo de CPU é dividido entre os dois processos.

3. Utilizando as funções `kill()` e `signal()`, escreva um programa que cria um processo filho e a seguir tanto o processo pai como o filho ficam em *loop* infinito incrementando a sua própria cópia de um contador global `i`. Os processos não exibem esse contador na tela durante o *loop*. Dessa forma você terá um modelo típico de processos orientados a CPU. Instale um *handler* para o sinal SIGINT de forma que um comando de `Ctrl+C` mata ambos os processos. O *handler* deve exibir na tela o valor do contador de cada processo no momento que ele for terminado. Um possível exemplo de saída fica como abaixo:

   ```
   $ ./ex3
   [PARENT]: PID 20019, starts counting
   [CHILD]: PID 20020, starts counting
   ^CProcess 20020 killed, i = 532634516.
   Process 20019 killed, i = 530995662.
   ```

   Realize experimentos com diferentes tempos de execução do programa e tente inferir algo sobre a qualidade do compartilhamento de tempo do seu SO: os processos são tratados de forma mais ou menos igualitária pelo escalonador de CPU?

6 Tarefa

Nesta tarefa você deve implementar um *shell* simples que recebe comandos do usuário e a seguir executa cada comando em um processo separado. O *shell* deve exibir o *prompt* % e ler uma linha de comando de no máximo 80 caracteres. O *shell* fica em execução até ser interrompido por um SIGINT. Nesse momento o *shell* deve exibir uma mensagem e terminar. Segue um exemplo de execução abaixo:

```
$ ./myshell
% date
ter set 12 17:39:29 -03 2017
% who
zambon   tty7         2017-09-12 07:59 (:0)
% pwd
/home/zambon/Teaching/2017-2/SO/laboratorio/lab02/src/SO_Lab02_Tarefa
% ls
a.out  myshell  myshell.c
% abc
Could not run command: abc!
% ^CExiting my shell.
```

Envie a sua implementação pelo AVA na tarefa correspondente. O prazo (inadiável) para submissão é dia 19/09/2017 (terça-feira) às 23:55. Essa tarefa vale um *bit* (1 – fez corretamente dentro do prazo, 0 – caso contrário) na nota dos exercícios de laboratório.