Atividade da Aula 04 (01/09/2021)

Aluno: Gabriel Hoffmann

Curso: Ciência da Computação

Refaça todos os exemplos demostrados em aula e descreva seus comportamentos baseado na teoria de Sistemas Operacionais.

Ubuntu > home > gabrielsh2 > workspace > C execlp.c

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

int main() {
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", NULL);
    printf("Done\n");
    exit(0);
}

```

Criamos um programa em C para testar a função do execlp. Essa função vai substituir o processo atual e chamar o processo “ps –ax”.

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc execlp.c -o execlp
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ls
a.out   execlp.c        gerencia_filas.c  gerencia_filas_2.c  gerencia_filas_3.c
execlp  gerencia_filas  gerencia_filas_2  gerencia_filas_3
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./execlp
Running ps with execlp
  PID TTY      STAT   TIME COMMAND
    1 ?        Ssl    0:02 /init
    7 tty1     Ss     0:00 /init
    8 tty1     S      0:00 -bash
   94 tty1     R      0:00 ps -ax
gabrielsh2@DESKTOP-827LBKF:~/workspace$
```

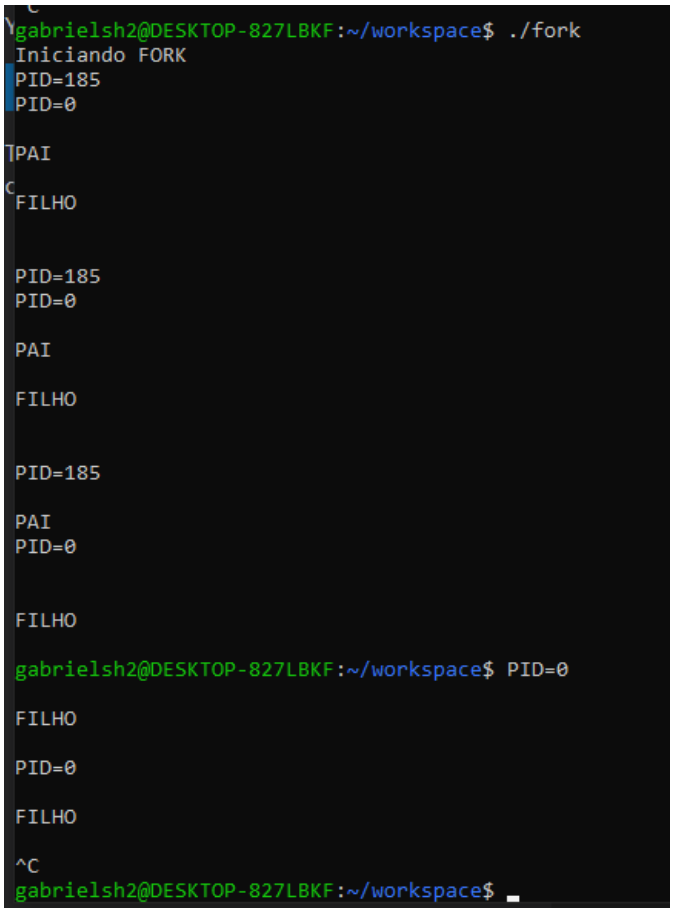
Podemos perceber que a linha de print “Done” não apareceu na execução do código pois o processo foi substituído pelo processo ps –ax.

Não achamos dados do processo do programa executado na listagem por conta dessa substituição feita pelo exec.

Ubuntu > home > gabrielsh2 > workspace > C fork.c > main()

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

int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("Iniciando FORK\n");
    pid = fork();

    switch (pid)
    {
    case -1:
        perror("errouu!\n");
        exit(1);
    case 0:
        message = "\nFILHO\n";
        n = 5;
        break;
    default:
        message = "\nPAI\n";
        n = 3;
        break;
    }

    for (; n > 0; n--)
    {
        printf("PID=%d\n", pid);
        puts(message);
        sleep(1);
    }

    exit(0);
}
```

Criamos um programa C para testar a função fork(), ela consiste em duplicar um processo e depois estamos printando o PID dos processos.



Na prática acontece que o processo pai acaba o loop e termina antes do filho que possui um n maior e isso impede a comunicação do filho com o shell e portanto o processo filho só é finalizado manualmente.

Ubuntu > home > gabrielsh2 > workspace > C pipe1.c > ...

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

int main() {
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value

    memset(buffer, '\0', sizeof(buffer));

    if(pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data, strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);

        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);

        exit(EXIT_SUCCESS);
    }

    exit(EXIT_FAILURE);
}

```

Criamos um código C com a finalidade de testar a função pipe que permite a conexão entre processos. Nesse programa estamos escrevendo um valor no nosso sistema operacional através do pipe e depois lemos esse valor e printamos.

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc pipe1.c -o pipe1
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./pipe1
Wrote 3 bytes
Read 3 bytes: 123
gabrielsh2@DESKTOP-827LBKF:~/workspace$
```

Ubuntu > home > gabrielsh2 > workspace > C pipe2.c > ...

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

int main() {
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();

        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        } else {
            data_processed = write(file_pipes[1], some_data, strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }

    exit(EXIT_SUCCESS);
}

```

Neste programa em C testamos misturar os dois conceitos vistos anteriormente, o pipe e o fork. Como o fork é uma duplicação de processos podemos ver o funcionamento do pipe conseguindo ligar dois processos lendo através de um e escrevendo através de outro.

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc pipe2.c -o pipe2
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

Na prática os dados de saída são os mesmos.

Ubuntu > home > gabrielsh2 > workspace > C pipe3.c > ...

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

int main() {
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();

        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void) execl("pipe4", "pipe4", buffer, (char *) 0);
            exit(EXIT_FAILURE);
        } else {
            data_processed = write(file_pipes[1], some_data, strlen(some_data))
            printf("Wrote %d bytes\n", data_processed);
        }
    }

    exit(EXIT_SUCCESS);
}

```

Nesse código foi acrescentado o uso do execl ao programa anterior. A lógica continua a mesma para a escrita, porém na leitura utilizamos o execl para substituir o processo atual pelo pipe4, programa que terá a responsabilidade de ler o valor do buffer do pipe.

Ubuntu > home > gabrielsh2 > workspace > C pipe4.c > ...

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

int main(int argc, char *argv[]) {
    int data_processed;
    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);


    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}

```

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc pipe3.c -o pipe3
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./pipe3
Wrote 3 bytes
```

Caso o pipe3 rode sem o pipe4 estar compilado no diretório informado, percebe-se que apenas a escrita no pipe funciona, pois o processo responsável pela leitura não está disponível.

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc pipe4.c -o pipe4
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./pipe3
Wrote 3 bytes
gabrielsh2@DESKTOP-827LBKF:~/workspace$ 263 - read 3 bytes: 123
^C
```

Após compilar pipe4, a leitura é feita ao executar novamente o pipe3.

Ubuntu > home > gabrielsh2 > workspace > C wait1.c > ...

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

int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("Iniciando FORK: PID=%d\n", getpid());
    pid = fork();

    switch (pid)
    {
    case -1:
        perror("errouu!\n");
        exit(1);
    case 0:
        message = "child: %d\n";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "parent: %d\n";
        n = 3;
        exit_code = 0;
        break;
    }

    for (; n > 0; n--)
    {
        printf(message, getpid());
        sleep(1);
    }

    if (pid != 0) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Parent terminated\n");
        if(WIFEXITED(stat_val))
            printf(
                "Message from parent: child proc %d exited with code %d\n",
                child_pid,
                WEXITSTATUS(stat_val)
            );
        else
            printf(
                "Message from parent: child proc %d terminated abnormally\n",
                child_pid
            );
    } else
        printf("Child has terminated\n");

    exit(exit_code);
}
```

O código fork.c tem o problema do processo pai ser encerrado antes do filho, para solucionar esse problema utilizamos do wait, criamos um arquivo wait.c que é uma melhoria do fork.c, após o for é verificado para quando for o processo pai não encerrar enquanto o processo filho ainda estiver rodando, utilizando as funções wait (retorna o ID do processo e mantém um ponteiro com o status desse processo) e WIFEXITED que retorna um booleano de acordo com o status do processo.

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc wait1.c -o wait1
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./wait1
Iniciando FORK: PID=285
parent: 285
child: 286
child: 286
parent: 285
parent: 285
child: 286
child: 286
child: 286
Child has terminated
Parent terminated
Message from parent: child proc 286 exited with code 37
gabrielsh2@DESKTOP-827LBKF:~/workspace$
```

Executando o código percebe-se que o problema do programa ter que ser encerrado manualmente por conta do processo do parente finalizar antes não existe mais.

Ubuntu > home > gabrielsh2 > workspace > C signal1.c > ...

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

void ouch(int sig) {
    printf("Ouch! - I got signal %d\n", sig);
    // Replace current signal handler with default handler
    (void) signal(SIGINT, SIG_DFL);
}

int main() {
    (void) signal(SIGINT, ouch); // Gatilho gerado pelo Ctrl+C
    while (1) {
        printf("Hello World!\n");
        sleep(1);
    }

}

```

Nesse programa estamos utilizando do conceito do signal para, a partir de gatilhos, realizarmos interrupções no código. Nesse exemplo o gatilho escolhido foi o SIGINT que dispara ao selecionar o CTRL + C, quando ocorre essa interrupção printamos na tela e substituímos o gatilho de volta para o gatilho default.

```
gabrielsh2@DESKTOP-827LBKF:~/workspace$ gcc signal1.c -o signal1
gabrielsh2@DESKTOP-827LBKF:~/workspace$ ./signal1
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
^COuch! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
^C
```