Relatório Lab4

Lucas Toscano Pimentel Appolinário Cerqueira - 2110695
Pedro Henrique Cariello de Freitas - 2011367

1- No código original, quando o Ctrl-C é digitado, é impressa o endereço de memória do handler e o processo não é encerrado porque o sinal enviado por esse comando (SIGINT) é ligado ao handler “intHandler”, o qual imprime uma mensagem. Já o Ctrl-\ foi ligado a um handler que imprime uma mensagem e encerra a execução.

Sem o tratamento de sinais, os comandos Ctrl-C e Ctrl-\ causam o término imediato do processo.

2- Não foi possível criar esse programa porque o sinal SIGKILL mata o processo antes. O SIGKILL não pode ser interceptado.

```
[lucas@fedora lab4]$ ./ex2 &
[3] 9315
Vai parar...
[lucas@fedora lab4]$ kill -s SIGKILL 9315
[3]+  Killed                  ./ex2
```

3-
a)

```
[lucas@fedora lab4]$ ./ex3 5
foiProgram (null) exceeded limit of 5 seconds!
```

O processo pai fica em sleep por 5 segundos, após esse tempo, ele envia um sinal SIGKILL para o kernel. Como o processo filho não termina nunca, o SIGKILL é que causa seu término.

b)

```
[lucas@fedora lab4]$ ./ex3 5
Child 10979 terminated within 5 seconds com estado 0.
[lucas@fedora lab4]$ ./ex3 2
Program (null) exceeded limit of 2 seconds!
```

Como o filho pode terminar e sua execução dura aproximadamente 3 segundos, quando se define o delay como 5 ele termina normalmente e quando o delay é 2 o pai força seu término.

c)

```
[lucas@fedora lab4]$ ./ex3 10 ./ex3_sleep5
indo dormir...
Acordei!
Child 11086 terminated within 10 seconds com estado 0.
```

Aqui o delay é definido como 10 e o tempo de execução do filho é de aproximadamente 5 segundos devido ao sleep(5). Por isso o filho consegue terminar normalmente.

d)

```
[lucas@fedora lab4]$ ./ex3 10 ./ex3_sleep15
indo dormir...
Program ./ex3_sleep15 exceeded limit of 10 seconds!
```

Já quando é usado o sleep(15) com o delay de 10, o filho excede o tempo de execução e o pai termina o processo filho.

4-

```
[lucas@fedora lab4]$ ./ex4 2.2 0
add: 2.200000
sub: 2.200000
mul: 0.000000
div: inf
```

Quando o segundo número é zero, a divisão tende ao infinito por isso ‘inf’. Após adicionar a captura do sinal SIGFPE, a execução com um dos números reais sendo zero ficou a mesma.

```
[lucas@fedora lab4]$ ./ex4 2 0
add: 2
sub: 2
mul: 0
Sinal 8 capturado!      Algum dos números de entrada é zero.
```

Quando os números foram convertidos para inteiros no código, o sinal pode ser capturado. O que ocorreu é que o C possui um tratamento padrão para os casos especiais da divisão como o “0/0” e “n/0” para reais, atribuindo a eles valores como “Inf” e “nan”. Isso não é verdade para números inteiros e por isso o sinal SIGFPE só é capturado quando números inteiros são usados.

5-
Arquivo ex5.c

```c
int main() {
    char *ext_prog = "./ex5_eternal_loop";
    char *ext_prog_args[] = {ext_prog, NULL};

    // Filho 1
    child1 = fork();
    if (child1 < 0) {
        perror("Falha na criacao do filho 1");
        exit(1);
    } else if (child1 == 0) {
        execv(ext_prog, ext_prog_args);
        exit(1);
    }

    // Filho 2
    child2 = fork();
    if (child2 < 0) {
        perror("Falha na criacao do filho 2");
        exit(1);
    } else if (child2 == 0) {
        execv(ext_prog, ext_prog_args);
        exit(1);
    }

    // COnfigura e inicializa o alarme
    signal(SIGALRM, alarmHandler);
    alarm(1);

    while (1)
    {
        if(timer >= 15)
        {
            kill(child1, SIGKILL);
            kill(child2, SIGKILL);
            printf("Tempo de excucao superior a 15s - filhos foram terminados.\n");
            exit(0);
        }
    }


    return 0;
}
```

1

2

3

4

5

Na main, em 2 e 3 ele cria os filhos, os quais executam o programa do loop eterno definido em 1. Em 4 ele cria a captura do sinal do alarme e inicializa ele. Em 5 ele espera 15 segundos (baseado na variável timer que é incrementada no handler do sinal do alarme) e após esse tempo ele termina os dois filhos e encerra a execução do processo pai.

```c
int child1_running = 1;
int child2_running = 0;
pid_t child1, child2;
int timer = 0;

void switch_children(int sign) {
    if (child1_running) {
        printf("Executando o filho 1\n");
        kill(child2, SIGSTOP);
        kill(child1, SIGCONT);
        child1_running = 0;
        child2_running = 1;
    } else {
        printf("Executando o filho 2\n");
        kill(child1, SIGSTOP);
        kill(child2, SIGCONT);
        child1_running = 1;
        child2_running = 0;
    }
}

void alarmHandler(int sign) {
    alarm(1);
    timer++;
    switch_children(0);
}
```

No handler do alarme, ele inicializa o alarme novamente e incrementa a variável timer, que marca o tempo de execução dos filhos. Após isso, ele chama a switch_children().
Na switch_child, ele envia para um filho o sinal SIGSTOP (faz o processo ficar em STOP) e para o outro o SIGCONT (faz o processo voltar a execução) dependendo que qual dos filhos estiver em execução.

Nessa questão tivemos dificuldade em fazer o controle do tempo, pois a função sleep(15) não funcionou. Após pesquisar na internet, vimos que isso não era um método recomendado pois em algumas situações pode causar assincronissidade.