

LAB3 - sinais

Érica Oliveira Regnier 2211893

Hanna Epelboim Assunção 2310289

1) Execute o programa “ctrl-c.c”. Digite Ctrl-C e Ctrl-\. Analise o resultado. Neste mesmo programa, remova os comandos signal() e repita o teste anterior observando os resultados. Explique o que ocorreu no relatório

Código fonte:

ctrl-c.c

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#define EVER ;;
void intHandler(int sinal);
void quitHandler(int sinal);
int main (void)
{
//void (*p)(int); // ponteiro para função que recebe int como
parâmetro
// p = signal(SIGINT, intHandler);
// printf("Endereco do manipulador anterior %p\n", p);
// p = signal(SIGQUIT, quitHandler);
// printf("Endereco do manipulador anterior %p\n", p);
printf ("Ctrl-C desabilitado. Use Ctrl-\\ para terminar\n");
for(EVER);
}
void intHandler(int sinal)
{
printf ("Você pressionou Ctrl-C (%d)\n", sinal);
}
void quitHandler(int sinal)
{
printf ("Terminando o processo...\n");
exit (0);
}
```

Comandos:

```
[c2310289@pantera-negra l3]$ gcc -Wall -o ctrlc ctrl-c.c
[c2310289@pantera-negra l3]$ ./ctrlc
```

Saída:

```
Ctrl-C desabilitado. Use Ctrl-\ para terminar
^C
[c2310289@pantera-negra l3]$
```

Explicação:

teste:

```
[c2310289@pantera-negra l3]$ ./ctrlc
Endereco do manipulador anterior (nil)
Endereco do manipulador anterior (nil)
Ctrl-C desabilitado. Use Ctrl-\ para terminar
^CVocê pressionou Ctrl-C (2)
^CVocê pressionou Ctrl-C (2)
^CVocê pressionou Ctrl-C (2)
^CVocê pressionou Ctrl-C (2)
^\\Terminando o processo...
[c2310289@pantera-negra l3]$
```

O programa `ctrl-c.c` é um exemplo de manipulação de sinais em C, utilizando a biblioteca `signal.h` para associar funções específicas a sinais do sistema. O código comentado (que pode ser descomentado para testes) configura os manipuladores de sinais usando a função `signal()`. Esta função associa funções específicas aos sinais `SIGINT` e `SIGQUIT`.

O programa exibe a mensagem: "Ctrl-C desabilitado. Use Ctrl-\ para terminar". Em seguida, o `loop for(EVER)` cria um loop infinito, fazendo o programa aguardar indefinidamente por sinais.

Com Manipuladores de Sinal Ativos (comentado):

- **Ctrl-C (SIGINT):** Quando pressionado, o sinal é interceptado pela função `intHandler()`, que exibe a mensagem "Você pressionou Ctrl-C (2)".
- **Ctrl-\ (SIGQUIT):** Quando pressionado, o sinal é interceptado pela função `quitHandler()`, que exibe a mensagem "Terminando o processo..." e encerra o programa.

Com Manipuladores de Sinal Removidos:

- **Ctrl-C (SIGINT):** Sem um manipulador para `SIGINT`, o comportamento padrão do sistema operacional é encerrar o programa imediatamente.

- **Ctrl-\ (SIGQUIT):** Sem um manipulador para SIGQUIT, o comportamento padrão também encerra o programa e pode gerar um core dump, dependendo da configuração do sistema.

Portanto, a principal diferença ao remover os manipuladores de sinal é que os sinais serão tratados de acordo com os comportamentos padrão do sistema operacional, em vez de serem tratados pelas funções definidas no programa.

2) Tente fazer um programa para interceptar o sinal SIGKILL. Você conseguiu? Explique.

Código fonte:

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

void handler(int signo) {
    printf("Sinal recebido e capturado: %d\n", signo);
}

int main(void) {
    signal(SIGKILL, handler);
    signal(SIGUSR1, handler);

    printf("%d\n", getpid());

    while (1) {

    }

    return 0;
}
```

Comandos:

```
[c2310289@paracatu l3]$ gcc -Wall -o ex2 ex2.c  
[c2310289@paracatu l3]$ ./ex2
```

```
c2310289@paracatu ~]$ kill -s USR1 5341  
c2310289@paracatu ~]$ kill -s KILL 5341
```

Saída:

```
5341  
Sinal recebido e capturado: 10  
Killed  
[c2310289@paracatu l3]$
```

Explicação:

O SIGKILL é um sinal utilizado para terminar processos de forma imediata e incondicional. Este sinal não pode ser interceptado, bloqueado ou ignorado pelo processo. Isso significa que, mesmo que o processo tenha manipuladores de sinais definidos, ele não conseguirá reagir ao SIGKILL.

A função `signal()` tenta definir um manipulador para o sinal SIGKILL, mas isso não é possível porque o SIGKILL não pode ser interceptado ou manipulado. O sistema operacional garante que o SIGKILL seja tratado de forma incondicional para encerrar o processo. Assim, o código deve exibir a mensagem "Não foi possível definir o manipulador para SIGKILL" (como indicado na saída). Caso o código fosse executado sem falhas, o programa ficaria em um loop infinito esperando por sinais.

Portanto, o código tenta demonstrar que não é possível manipular o sinal SIGKILL. Embora o código esteja escrito para definir um manipulador para SIGKILL, o comportamento esperado é que o sistema operacional não permita isso e retorne um erro.

3) Execute e explique o funcionamento de `filhocidio.c`, com as 4 opções:

a- `for(EVER) /* filho em loop eterno */`

b- sleep(3) /* filho dorme 3 segundos */
c- execvp(sleep5) /* filho executa o programa sleep5 */
d- execvp(sleep15) /* filho executa o programa sleep15 */
Explique o que ocorreu em cada programa.

a)

Código fonte:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#define EVER ;;

void childhandler(int signo);
int delay;
int main (int argc, char *argv[])
{
    pid_t pid;
    signal(SIGCHLD, childhandler);
    if ((pid = fork()) < 0)
    {
        fprintf(stderr, "Erro ao criar filho\n");
        exit(-1);
    }
    if (pid == 0) /* child */
        for(EVER); /* ou sleep(3); ou, no próximo exercício, execvp(argv[2],
        argv);*/
    else /* parent */
    {
        sscanf(argv[1], "%d", &delay); /* read delay from command line */
        sleep(delay);
        printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
        kill(pid, SIGKILL);
    }
}
```

```

return 0;
}
void childhandler(int signo) /* Executed if child dies before parent
*/
{
int status;
pid_t pid = wait(&status);
printf("Child %d terminated within %d seconds com estado %d.\n", pid,
delay, status);
exit(0);
}

```

Comandos:

```

[c2310289@yukon l3]$ gcc -Wall -o aa filhocidio.c
[c2310289@yukon l3]$ ./aa 10

```

Saída:

```

Program (null) exceeded limit of 10 seconds!
[c2310289@yukon l3]$

```

Explicação:

No programa filhocidio.c, o processo pai lê um tempo de espera (delay) da linha de comando(nesse caso 10), espera esse tempo, e então mata o processo filho, caso ele ainda esteja ativo. Isso ocorre por meio da manipulação de sinais, com o SIGCHLD. O processo filho, nesse caso, é um loop infinito e, por isso, o pai mata o filho (o tempo passado para o pai foi excedido) e a função handler não é ativada, dado que o SIGCHLD não foi passado, uma vez que o processo filho não terminou.

b)

Código fonte:

```

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

```

```

#include <sys/wait.h>
#define EVER ;;

void childhandler(int signo);
int delay;
int main (int argc, char *argv[])
{
    pid_t pid;
    signal(SIGCHLD, childhandler);
    if ((pid = fork()) < 0)
    {
        fprintf(stderr, "Erro ao criar filho\n");
        exit(-1);
    }
    if (pid == 0) /* child */
        sleep(3); /* ou sleep(3); ou, no próximo exercício, execvp(argv[2],
        argv);*/
    else /* parent */
    {
        sscanf(argv[1], "%d", &delay); /* read delay from command line */
        sleep(delay);
        printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
        kill(pid, SIGKILL);
    }
    return 0;
}

void childhandler(int signo) /* Executed if child dies before parent
*/
{
    int status;
    pid_t pid = wait(&status);
    printf("Child %d terminated within %d seconds com estado %d.\n", pid,
    delay, status);
    exit(0);
}

```

Comandos:

```
c2310289@paracatu:l3

[c2310289@paracatu l3]$ gcc -Wall -o filhocidioB filhocidiosleep3.c
[c2310289@paracatu l3]$ gcc -Wall -o p1 prog2Filhocidio.c
[c2310289@paracatu l3]$ ./filhocidioB 10
Child 4446 terminated within 10 seconds com estado 0.
[c2310289@paracatu l3]$
```

Saída:

```
c2310289@paracatu:l3

[c2310289@paracatu l3]$ gcc -Wall -o filhocidioB filhocidiosleep3.c
[c2310289@paracatu l3]$ gcc -Wall -o p1 prog2Filhocidio.c
[c2310289@paracatu l3]$ ./filhocidioB 10
Child 4446 terminated within 10 seconds com estado 0.
[c2310289@paracatu l3]$
```

Explicação:

No programa filhocidio.c, o processo pai lê um tempo de espera (delay) da linha de comando(nesse caso 10), espera esse tempo, e então mata o processo filho, caso ele ainda esteja ativo. Isso ocorre por meio da manipulação de sinais, com o SIGCHLD. O processo filho, nesse caso, é um sleep(3) e, por isso, o pai não mata o filho, já que a função handler é ativada, dado que o SIGCHLD foi passado, quando o processo filho terminou. Assim, como o processo filho não excedeu o tempo que foi passado ao pai, ele conseguiu terminar e não foi morto.

c)

Código prog2Filhocidio.c para letra c


```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    fprintf(stdout, "indo dormir...\n");
    sleep(5);
    fprintf(stdout, "Acordei!\n");
    return 0;
}
```

Código fonte:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#define EVER ;;

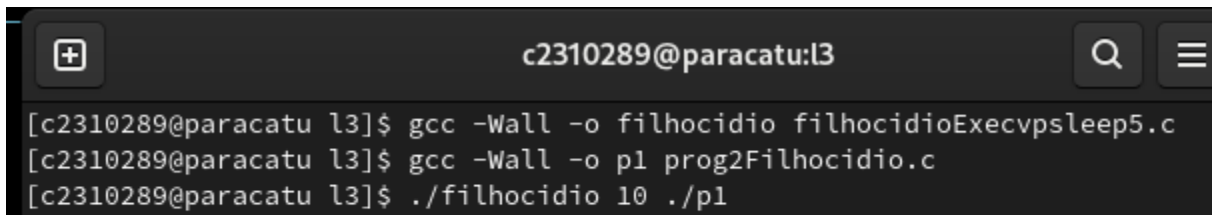
void childhandler(int signo);
int delay;
int main (int argc, char *argv[])
{
    pid_t pid;
    signal(SIGCHLD, childhandler);
    if ((pid = fork()) < 0)
    {
        fprintf(stderr, "Erro ao criar filho\n");
        exit(-1);
    }
    if (pid == 0) /* child */
        execvp(argv[2], argv); /* ou sleep(3); ou, no próximo exercício,
        execvp(argv[2], argv);*/
    else /* parent */
```

```

{
sscanf(argv[1], "%d", &delay); /* read delay from command line */
sleep(delay);
printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
kill(pid, SIGKILL);
}
return 0;
}
void childhandler(int signo) /* Executed if child dies before parent
*/
{
int status;
pid_t pid = wait(&status);
printf("Child %d terminated within %d seconds com estado %d.\n", pid,
delay, status);
exit(0);
}

```

Comandos:

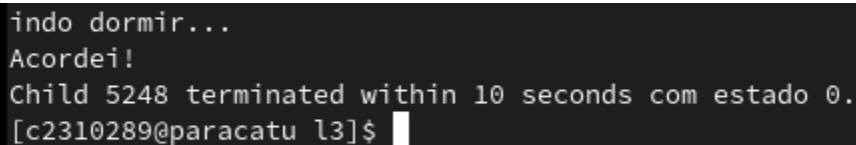


```

c2310289@paracatu:l3
[c2310289@paracatu l3]$ gcc -Wall -o filhocidio filhocidioExecvpsleep5.c
[c2310289@paracatu l3]$ gcc -Wall -o p1 prog2Filhocidio.c
[c2310289@paracatu l3]$ ./filhocidio 10 ./p1

```

Saída:



```

indo dormir...
Acordei!
Child 5248 terminated within 10 seconds com estado 0.
[c2310289@paracatu l3]$

```

Explicação:

No programa filhocidio.c, o processo pai lê um tempo de espera (delay) da linha de comando(nesse caso 10), espera esse tempo, e então mata o processo filho, caso ele ainda esteja ativo. Isso ocorre por meio da manipulação de sinais, com o SIGCHLD. O processo filho, nesse caso, é um execvp, de forma que ele executa um outro programa que, neste item, contém um sleep(5) e duas mensagens, uma antes e outra depois deste sleep. Como o tempo de execução não excede o passado para o pai(10), ambas as mensagens são exibidas e o pai não mata o filho, já que a função handler é ativada, dado que o SIGCHLD foi passado, quando o processo filho terminou. Assim, como o processo filho não excedeu o tempo que foi passado ao pai, ele conseguiu terminar e não foi morto.

d)

Código prog2Filhocidio.c para a letra d

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    fprintf(stdout, "indo dormir...\n");
    sleep(15);
    fprintf(stdout, "Acordei!\n");
    return 0;
}
```

Código fonte:

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

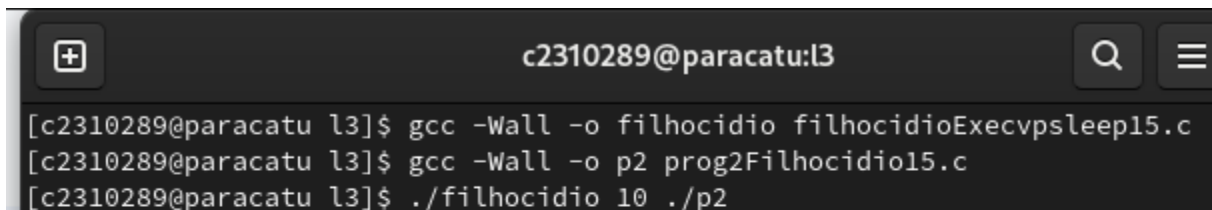
void childhandler(int signo);
int delay;
int main(int argc, char *argv[]) {
    pid_t pid;
    signal(SIGCHLD, childhandler);
    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erro ao criar filho\n");
        exit(-1);
    }
    if (pid == 0) /* child */
        execvp(argv[2], argv); /* ou sleep(3); ou, no próximo exercício,
                                execvp(argv[2], argv);*/
    else /* parent */
    {
        sscanf(argv[1], "%d", &delay); /* read delay from command line */
        sleep(delay);
        printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
    }
}
```

```

        kill(pid, SIGKILL);
    }
    return 0;
}
void childhandler(int signo) /* Executed if child dies before parent */
{
    int status;
    pid_t pid = wait(&status);
    printf("Child %d terminated within %d seconds com estado %d.\n", pid, delay,
        status);
    exit(0);
}

```

Comandos:

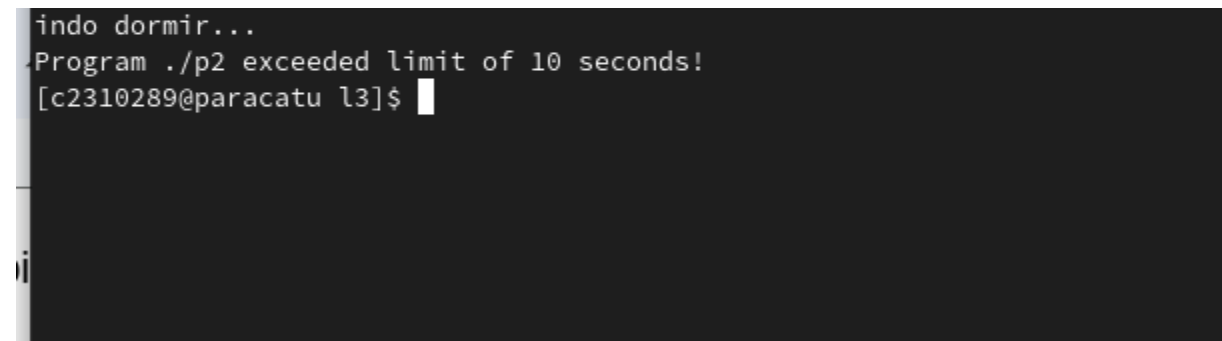


```

c2310289@paracatu:l3
[c2310289@paracatu l3]$ gcc -Wall -o filhocidio filhocidioExecvpsleep15.c
[c2310289@paracatu l3]$ gcc -Wall -o p2 prog2Filhocidio15.c
[c2310289@paracatu l3]$ ./filhocidio 10 ./p2

```

Saída:



```

indo dormir...
Program ./p2 exceeded limit of 10 seconds!
[c2310289@paracatu l3]$

```

Explicação:

No programa filhocidio.c, o processo pai lê um tempo de espera (delay) da linha de comando(nesse caso 10), espera esse tempo, e então mata o processo filho, caso ele ainda esteja ativo. Isso ocorre por meio da manipulação de sinais, com o SIGCHLD. O processo filho, nesse caso, é um execvp, de forma que ele executa um outro programa que, neste item, contém um sleep(15) e duas mensagens, uma antes e outra depois deste sleep. Como o tempo de execução excede o passado para o pai(10), apenas a primeira mensagem é exibida e o pai mata o filho, já que a função handler não é ativada, dado que o SIGCHLD não foi passado, já

que o processo filho não terminou. Assim, como o processo filho excedeu o tempo que foi passado ao pai, ele foi morto.

4) Faça um programa que leia 2 números reais e imprima o resultado das 4 operações básicas sobre estes 2 números. Verifique o que acontece se o 2º. número da entrada for 0 (zero). Capture o sinal de erro de floating point (SIGFPE) e repita a experiência anterior. Faça o mesmo agora lendo e realizando as operações com inteiros. Explique o que ocorreu nas duas situações.

Código fonte:

- **Com float:**

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

void handler(int sig){
    printf("Erro! Divisão por zero!");
    exit(1);
}

int main(void){
    float numero1,numero2;

    signal(SIGFPE,handler);

    printf("Digite o primeiro número: ");
    scanf("%f", &numero1);

    printf("Digite o segundo número: ");
    scanf("%f", &numero2);

    printf("Soma: %.2f\nSubtração: %.2f\nMultiplicação: %.2f\nDivisão: %.2f\n",
    numero1+numero2, numero1-numero2, numero1*numero2, numero1/numero2);

    return 0;
}
```

- **Com int:**

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

```

void handler(int sig){
    printf("Erro! Divisão por zero!\n");
    exit(1);
}

int main(void){
    int numero1,numero2;

    signal(SIGFPE,handler);

    printf("Digite o primeiro número inteiro: ");
    scanf("%d", &numero1);

    printf("Digite o segundo número inteiro: ");
    scanf("%d", &numero2);

    printf("Soma: %d\nSubtração: %d\nMultiplicação: %d\nDivisão: %d\n",
    numero1+numero2, numero1-numero2, numero1*numero2, numero1/numero2);

    return 0;
}

```

Comandos:

```

[c2310289@paracatu l3]$ gcc -Wall -o calcfloat ex3Calculadora.c
[c2310289@paracatu l3]$ ./calcfloat
[c2310289@paracatu l3]$ gcc -Wall -o calcfloat ex3Calculadora.c
[c2310289@paracatu l3]$ ./calcI

```

Saída:

```

Digite o primeiro número inteiro: 1
Digite o segundo número inteiro: 0
Erro! Divisão por zero!

Digite o primeiro número real: 6
Digite o segundo número real: 0
Soma: 6.00
Subtração: 6.00
Multiplicação: 0.00
Divisão: inf

```

Explicação:

Foto antes de capturar o sinal:

```
[c2310289@jari l3]$ ./calc
Digite o primeiro número: 1
Digite o segundo número: 0
Soma: 1.00
Subtração: 1.00
Multiplicação: 0.00
Divisão: inf
```

```
Digite o primeiro número inteiro: 5
Digite o segundo número inteiro: 0
Floating exception (core dumped)
```

Após capturar sinal:

```
Digite o primeiro número real: 6
Digite o segundo número real: 0
Soma: 6.00
Subtração: 6.00
Multiplicação: 0.00
Divisão: inf
```

```
Digite o primeiro número real: 3
Digite o segundo número real: 0
Erro! Divisão por zero!
```

A divisão por zero em float tem como resultado infinito, pois o 0 em float, não é exatamente 0, logo existe a divisão, porém por ser uma divisão por um número muito pequeno, a operação dá infinito. Já com inteiros, a divisão por zero não existe e essa dá um erro (Floating exception), caso ocorra. Assim, capturamos o sinal SIGFPE, de modo que, quando ativado, ou seja, quando há uma divisão por zero - com números inteiros- , nesse caso, será chamada a função handler que exibe uma mensagem e termina o programa. Com números reais, o sinal não é acionado, de forma que não há mudanças com a alteração do código para captura de sinais.

5) Faça um programa que tenha um coordenador e dois filhos. Os filhos executam (execvp) um programa que tenha um loop eterno. O pai coordena a execução dos filhos realizando a preempção dos processos, executando um deles por 1 segundo, interrompendo a sua execução e executando o outro por 1 segundo, interrompendo a sua execução e assim sucessivamente. O processo pai fica então coordenando a execução dos filhos, é, na verdade, um escalonador. Faça o processo pai executar por 15 segundos e, ao final, ele mata os processos filhos e termina. Explique como realizou a preempção, se o programa funcionou a contento e as dificuldades encontradas.

Código fonte:

escalonadorzinho.c

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

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int pid1,pid2;
int tempo = 0, troca = 0;

void handler(int signo){
    if(tempo == 15){
        printf("Acabou o tempo! %d segundos decorridos.\n",tempo);
        kill(pid1,SIGKILL);
        kill(pid2,SIGKILL);
        exit(0);
    }

    if(troca == 0){
        kill(pid1,SIGSTOP);
        kill(pid2,SIGCONT);
    }
    else{
        kill(pid2,SIGSTOP);
        kill(pid1,SIGCONT);
    }

    troca = 1-troca;
    tempo++;
    alarm(1);
}

int main (int argc, char *argv[]){

    signal(SIGALRM, handler);

    if ((pid1 = fork()) == 0)
    {
        execvp(argv[1], argv);
    }
}
```



```
}
kill(pid1, SIGSTOP);

if ((pid2 = fork()) == 0)
{
    execvp(argv[2], argv);
}
kill(pid2, SIGSTOP);

alarm(1);

while(1){
    pause();
}

return 0;
}
```

ex5F1.c

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

int main(void){
while(1){
    printf("%d\n", getpid());
    sleep(1);
}
}
```

ex5F2.c

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

int main(void){
while(1){
    printf("%d\n", getpid());
    sleep(1);
}
}
```

Comandos:

```
[c2310289@paracatu l3]$ gcc -Wall -o F1 ex5F1.c
[c2310289@paracatu l3]$ gcc -Wall -o F2 ex5F2.c
[c2310289@paracatu l3]$ gcc -Wall -o escalonador escalonadorzinho.c
[c2310289@paracatu l3]$ ./escalonador ./F1 ./F2
```

Saída:

```
26810
26809
26810
26809
26810
26809
26810
26809
26810
26809
26810
26809
26810
26809
26810
Acabou o tempo! 15 segundos decorridos.
```

Explicação:

Há dois programas filhos, que rodam um loop infinito, de forma que a cada 1 segundo eles exibem seus respectivos pid. O processo pai gerencia seus dois filhos, de forma que sempre que o alarm(1) chama a função handler a cada um segundo. Essa faz a troca dos processos filhos - com SIGSTOP e SIGCONT - a cada um segundo. A função handler se chama ao final, com outro alarm, porém não entra em loop infinito, dado que a variável tempo controla há quanto tempo o pai está controlando seus filhos e quando esse tempo chega a 15 segundos, o pai mata seus filhos e encerra. Não houve dificuldades, tudo funcionou conforme o esperado.