INF1316 - Laboratório 03 - Envio de sinais entre processos

Nome:

- Guilherme Riechert Senko
- Pedro de Almeida Barizon

Matrícula:

- 2011478
- 2211350

Professor: Luiz Fernando Seibel

Turma: 3WA

Data: 18/09/2024

Objetivo: implementar e compreender a transmissão de sinais entre processos em sistemas Unix, com destaque para a função `signal`.

Observações preliminares

- Para mais detalhes, todos os códigos fonte comentados encontram-se em anexo. Por isso, para evitar redundâncias, não serão aqui exibidos.
- Todos os arquivos fonte foram compilados com auxílio do *GNU C Compiler* (GCC) fazendo `gcc -Wall -o programa fonte1.c fonte2.c … fonten.c` e executados com `./programa arg1 arg2 … argn`. Caso haja particularidades, estas serão abordadas ao longo do relatório.

Exercício 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.

1.1 Estrutura

Criou-se apenas um módulo, o `ctrl_c.c`, que continha as funções:

- `main`: responsável pela execução do programa.
- `intHandler`: responsável pelo tratamento de `SIGINT`, isto é, do sinal enviado pelo teclado via `Ctrl-C`. Apenas exibe uma mensagem que indica ter sido apertado `Ctrl-C`.
- `quitHanlder`: responsável pelo tratamento de `SIGQUIT`, isto é, do sinal enviado pelo teclado via `Ctrl-\`. Exibe uma mensagem de término e encerra.

1.2 Solução

Inicialmente, são exibidas duas mensagens na console: uma para indicar que `Ctrl-C` não encerrará o programa (diferentemente do usual), outra para informar que, em vez disso, deve-se terminá-lo com `Ctrl-\`.

Ademais, chama-se `signal` duas vezes: a primeira para acionar o *handler* de `Ctrl-C` (`SIGINT`), a segunda para o de `Ctrl-\` (`SIGQUIT`). Para cada chamada, o ponteiro para função de tipo `(void) (*f) (int)` retornado é subsequentemente exibido, cujo valor é sempre `NULL`. Esse corresponde ao endereço do *handler* anterior, que, na ausência de indicação pelo programador, era o *default*.

Finalmente, fica-se em *loop* à espera de sinais do usuário.

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab03_seibel/03_ex1$ ./ctrl_c
Ctrl-C nao interrompera o programa
Use Ctrl-\ para interromper o programa
Endereco do manipulador anterior (nil)
Endereco do manipulador anterior (nil)
^CVoce pressiounou ctrl-C (2)
^CVoce pressiounou ctrl-C (2)
^CVoce pressiounou ctrl-C (2)
^CVoce pressiounou ctrl-C (2)
^CVoce pressiounou ctrl-C (2)
^\Terminando o programa...
```

Figura 1.1 - Saída de `ctrl_c` com `signal`

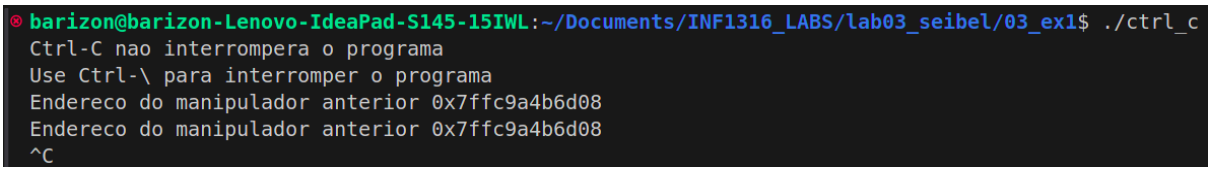

Figura 1.2 - Saída de `ctrl_c` sem `signal`

1.3 Observações e conclusões

Como era de se esperar, sempre que se apertava `Ctrl-C`, a mensagem correspondente definida em `intHandler` era exibida. Quando se apertou Ctrl-\, o programa foi devidamente encerrado segundo a `quitHandler`.

Quando se comentaram as chamadas de `signal`, os sinais retornaram ao tratamento default (`SIG_DFL`): `Ctrl-C` encerrava o programa, e `Ctrl-\`, além de o terminar, produzia um *core dump*. Vale ressaltar, por fim, que o ponteiro para função, por não ter sido atualizado por `signal`, armazenava *bytes* residuais de memória (“lixo”), o que explica o “`0x7ffc9a4b6d08`”.

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

2.1 Estrutura

Para que fosse enxuto, criou-se apenas um módulo, `kill_intercept.c`, que continha as funções:

- `main`: responsável pela execução do programa.
- `usrHandler`: responsável pelo tratamento de `SIGUSR1`, isto é, de um dos sinais programáveis pelo usuário. Apenas exibe uma mensagem que indica ter sido recebido o sinal `SIGUSR1`.
- `killHandler`: responsável pelo tratamento (ou, ao menos, pela tentativa de tratamento) de `SIGKILL`. Exibe uma mensagem que indica que o processo escapou da "morte".

2.2 Solução

Inicialmente, via `fork`, um processo pai cria um filho, que aciona tanto a `usrHandler` quanto a `killHandler` e depois espera por sinais do pai com um *loop*. Este, por sua vez, dorme por 1 segundo e, com auxílio de `kill`, envia um `SIGUSR1` ao filho. Em seguida, dá outro `sleep` de 1 segundo e, enfim, envia o `SIGKILL` para matar o filho. Para atestar o óbito, após mais 1 segundo de `sleep`, envia mais um `SIGUSR1` (que, supõe-se, não surtirá efeito no morto). Finalmente, para não haver dúvidas, obtém o *status* de término do filho com `waitpid` e, por intermédio da macro `WIFSIGNALED`, verifica se o processo foi encerrado por causa de um sinal. Caso o resultado seja verdadeiro, exibe a mensagem apropriada. A saída encontra-se abaixo:

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab03_seibel/03_ex2$ ./kill_intercept
Recebido sinal de usuario 1
Ninguem escapa da morte... ate mesmo um processo.
```

Figura 2.1 - Saída de `kill_intercept`

2.3 Observações e conclusões

Como era de se esperar, o filho recebeu o primeiro `SIGUSR1`, mas já tinha sido morto pelo pai quando este lhe enviou o segundo, o que explica apenas uma mensagem “Recebido sinal de usuario 1” ter aparecido. Ironia do destino ou não, fato é que nem mesmo processos podem escapar da morte: `SIGKILL` não pode ser interceptado. `killHandler` é apenas uma ilusão dos que aspiram à imortalidade.

Exercício 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.

3.1 Estrutura

Para não nos prolongarmos muito, não explicaremos o funcionamento nem citaremos a estrutura dos programas `sleep5.c` e `sleep15.c`, cujos códigos fontes se encontram nos slides da aula. Dito isso, foram criadas três subvariantes de `filhocidio.c`: `filhocidio_forever.c`, `filhocidio_sleep.c` e `filhocidio_exec.c` (cujos nomes são autoexplicativos). Os três programas são idênticos a menos do bloco de código a ser executado pelo processo filho. Todos possuem as funções:

- `main`: responsável pela execução do programa.
- `childHandler`: responsável pelo tratamento de `SIGCHLD`, isto é, sinal enviado a um processo pai sempre que um filho seu termine. Recupera PID do filho; exibe mensagem que indica que filho não foi morto pelo pai; e termina o programa.

3.2 Solução

Quando se executa o programa na linha de comando, deve-se passar também um inteiro que indica o tempo de tolerância até que o pai mate o filho. Feito isso, o programa inicia ao associar `SIGCHLD` a `childHandler` via chamada de `signal`. Em seguida, o processo pai dá origem a um filho com `fork`. O bloco a ser executado pelo filho dependerá da subvariante:

- *forever*: filho fica em um *loop* eterno.
- *sleep*: filho dorme por 3 segundos e termina.
- *exec*: filho executa o programa `sleep5` (ou `sleep15`).

Depois do `fork`, no processo pai, converte-se o valor de tolerância passado como argumento da `main` de *string* para `int` via a função `atoi`, cujo valor de

retorno é armazenado na variável global `delay`. O pai dorme por `delay` segundo(s) e, enfim, envia um `SIGKILL` ao filho. Ao final, uma mensagem de luto é exibida pelo pai, que encerra o programa. Caso, porém, o filho termine antes de o pai acordar, `SIGCHLD` acionará `childHandler`, que exibe uma mensagem que indica que o filho "sobreviveu" (isto é, não foi morto pelo pai) e, em seguida, encerra o programa. As saídas dos programas encontram-se abaixo:

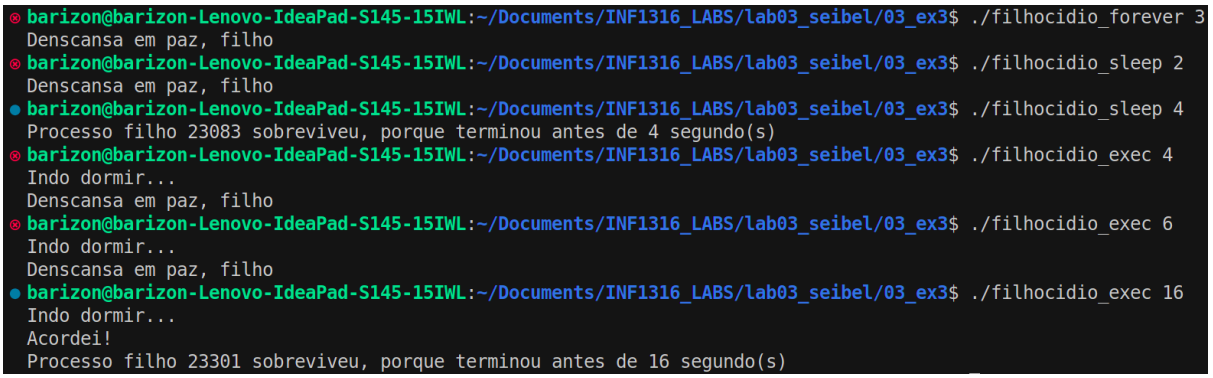

Figura 3.1 - Saída de subvariantes de `filhocidio.c` (foi usado `sleep15` na versão exec)

3.3 Observações e conclusões

Em síntese, o resultado é simples: se o tempo de retorno do processo filho for estritamente inferior ao tempo de tolerância, o processo filho não será morto pelo pai, porque `childHandler` será acionada antes de `kill`, o que encerrará o programa. Daí concluímos que o filho em *forever* sempre morrerá. Nos demais casos, portanto, isso dependerá diretamente do valor de `delay`. Cabe ressaltar, porém, que, nos casos em que houver igualdade entre tolerância e tempo de término do filho, observamos que o filho ainda morrerá. Por fim, para fins práticos, `sleep5` e `sleep15` comportaram-se da mesma maneira. Evidentemente, a diferença estava na quantidade de tempo que deveria ser tolerada para que o filho sobrevivesse: 6 s *versus* 16 s.

Exercício 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.

4.1 Estrutura

Criaram-se dois módulos: `le_floats.c` e `le_ints.c`, cujos nomes são autoexplicativos. Ambos possuíam funções análogas:

- `main`: responsável pela execução do programa.
- `fpeHandler`: responsável pelo tratamento de `SIGFPE`, isto é, da Exceção de Ponto Flutuante (***F**loating **P**oint **E**xception*). Exibe uma mensagem que indica ter sido recebido o sinal `SIGFPE` e termina o processo.

4.2 Solução

Os dois módulos são idênticos, com a exceção de que em `le_floats.c` declaram-se dois *floats* `f1` e `f2`; ao passo que em `le_ints.c`, os inteiros `i1` e `i2`. Por isso, para descrevermos ambos simultaneamente, usaremos o termo “número”.

Depois de declaradas as variáveis, chama-se a função `signal` para acionar `fpeHandler`, caso se receba `SIGFPE`. Em seguida, são atribuídos aos números valores via teclado com a chamada de `scanf`. Por fim, são exibidos, respectivamente, os resultados das operações de soma, subtração, multiplicação e divisão entre ambos com auxílio de `printf`. As saídas encontram-se abaixo:

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab03_seibel/03_ex4$ ./le_floats
Informe os dois numeros reais: 1.0 2.0

Soma:    +3.00
Subtr.:  -1.00
Prod.:   +2.00
Div.:    +0.50
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab03_seibel/03_ex4$ ./le_floats
Informe os dois numeros reais: 1.0 0.0

Soma:    +1.00
Subtr.:  +1.00
Prod.:   +0.00
Div.:    +inf
```

Figura 4.1 - Saída de `le_floats`

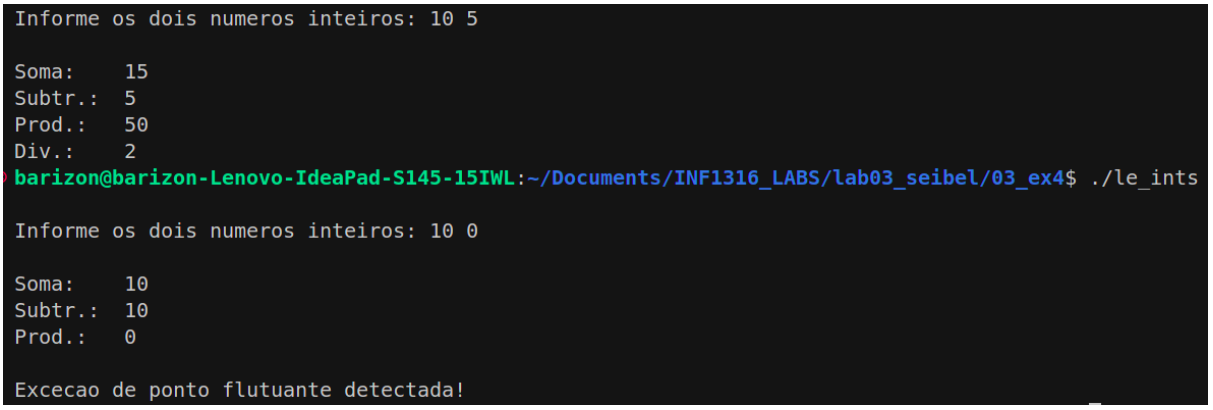

Figura 4.2 - Saída de `le_ints`

4.3 Observações e conclusões

No programa `le_floats`, com ou sem `signal` atrelado a `SIGFPE`, não foi detectado sinal de FPE. Isso se explica pelo fato de que o padrão IEEE 754 de representação de pontos flutuantes, adotado pelas máquinas atuais, prevê a representação de $\pm\ \infty$. Dessa forma, a divisão de algum *float* por 0 é associada ao valor infinito, representado na console como `inf` (a menos do sinal `+` ou `-`, que dependerá do dividendo).

Assim, apesar de contradizer um dos pilares da Matemática, a divisão por zero é aceitável em termos de pontos flutuantes. Isso se explica pelo fato de que os pontos flutuantes não representam perfeitamente os números reais: números suficientemente próximos de zero terão a mesma representação que ele. Logo, se toda vez que uma divisão por “zero” ocasionasse um erro, simulações numéricas e outras aplicações científicas lançariam constantes exceções.

Em `le_ints`, por outro lado, a divisão por zero, com ou sem `signal`, fez emitir `SIGFPE`. É claro: como a representação de inteiros não enfrenta as mesmas limitações, se for detectada uma divisão por zero, sabemos com certeza se tratar de um zero. Por isso, faz sentido lançarmos uma exceção. O único ponto de estranhamento foi uma operação entre inteiros causar uma exceção de ponto flutuante. Isso permanece um mistério… Ou quem sabe uma economia de sinais por parte dos criadores do Unix.

Exercício 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.

5.1 Estrutura

Criaram-se dois módulos: `escalonador.c` e `filho.c`. O primeiro foi responsável por escalonar dois processos, que executavam o programa `filho.c`.

5.1.1 `escalonador.c`

Continha as funções:

- `main`: responsável pela execução do programa.
- `alarmHandler`: responsável pelo tratamento de `SIGALRM`, isto é, do sinal de alarme emitido pelo *clock* da máquina. Mata todos os processos filhos; verifica que foram mortos; exibe uma mensagem que indica o fim do escalonamento; e termina o programa.

5.1.2 `filho.c`

Continha a função:

- `main`: responsável pela execução do programa. Apenas exibe o PID do processo atual a cada 0,500 segundo dentro de um *loop* eterno.

5.2 Solução

Comecemos por `filho.c`. O processo conhece seu PID via `getpid`. Em seguida, entra em um *loop* eterno composto por uma chamada de `printf` que exibe o PID e por um `usleep` de 500.000 microsegundos, ou seja, 0,500 segundo.

Quanto a `escalonador.c`, inicialmente o processo coordenador cria dois filhos com `fork`, armazenando seus respectivos PIDs no vetor `pid`. Os filhos executam `filho.c` graças ao uso de `execvp`. Cabe mencionar que, para que os

filhos compartilhem do mesmo `stdout` que o pai, é necessário incluir "./" antes do nome do executável na chamada de `execvp`.

Em seguida, o pai prepara um alarme de 15 segundos com `alarm` e aciona `signal` para atrelar `SIGALRM` a `alarmHandler`. Feito isso, envia `SIGSTOP` a todos os filhos, para que entrem em pausa. Define-se um contador `i`, que inicia em 0. Enfim, começará a coordenação:

1. O pai envia `SIGCONT` ao filho com `pid[i]` para que continue e dorme por 1 segundo (tempo de escalonamento).
2. Ao acordar, envia novamente `SIGSTOP` ao filho atual (*preempção*) e incrementa o contador, passando ao próximo filho.
3. Para garantir ciclicidade, faz-se `i %= NUM_FIL`, em que `NUM_FIL` é o número de processos filhos. Isso garante que sempre `0 <= i < NUM_FIL`.
4. Em seguida, repete a partir da etapa 1.

Ao término dos 15 segundos, o *clock* emite ao pai um `SIGALRM`. `alarmHandler` será, pois, acionada no processo pai, que, então, mata os filhos e encerra. A saída do programa encontra-se abaixo:

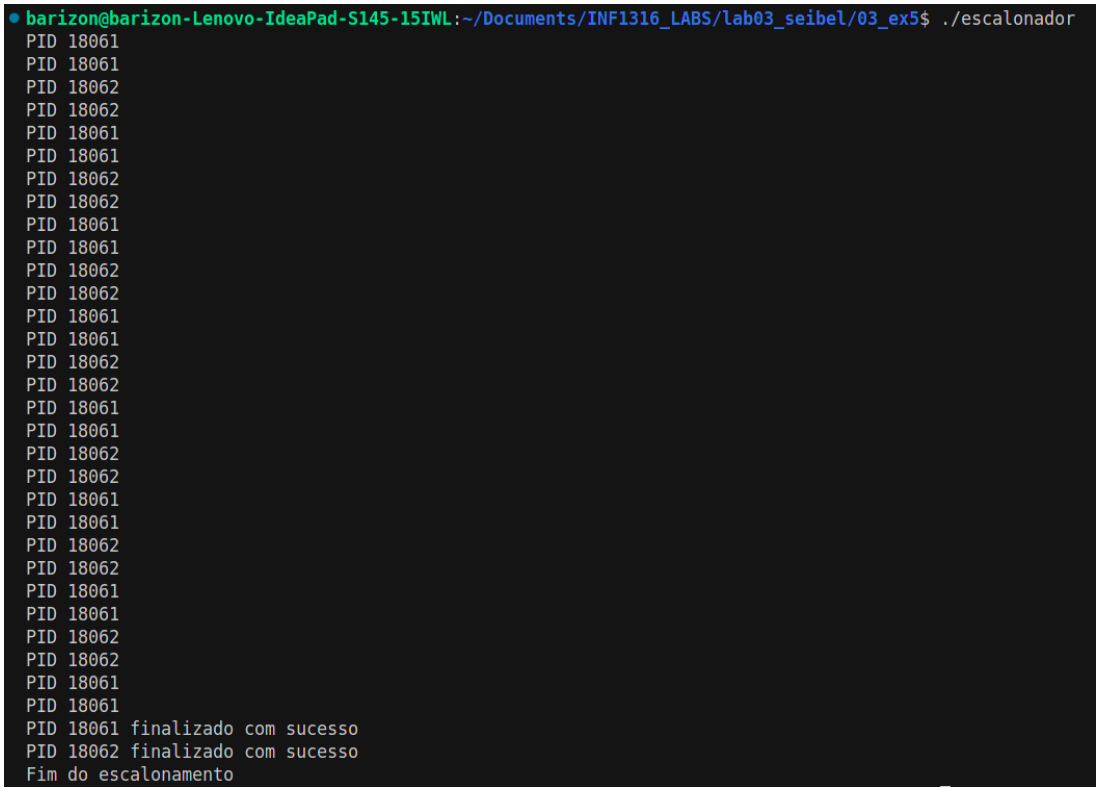

Figura 5.1 - Saída de `escalonador`

5.3 Observações e conclusões

Não houve grandes dificuldades na implementação do escalonador, tendo o resultado saído como esperado: foram exibidas exatas 30 mensagens de PID, que teoricamente, distavam 0,500 segundo entre si, o que totaliza exatos 15 segundos, o tempo total de escalonamento previsto. Além disso, cada processo exibia duas mensagens antes de ser interrompido, o que corresponde ao tempo de preempção previsto, de 1 segundo. Cabe mencionar, porém, que não sabíamos *a priori* se o uso de `sleep` seria um bom meio de coordenar a preempção; a julgarmos pelo resultado, parece que o é. Por fim, a analogia entre um escalonador e um guarda de trânsito que coordena diferentes filas de carros foi uma abstração útil, que contribuiu com a construção do código.

6 Conclusões finais

Os programas foram compilados sem erro usando-se o GCC, o ambiente de desenvolvimento do Visual Studio Code e o sistema operacional Ubuntu. Além disso, conforme se observa nas saídas dos programas, todos os testes obtiveram êxito. Por fim, devemos reconhecer a importância deste laboratório, uma vez que nos permitiu vislumbrar o grande poder do envio de sinais entre processos, que nos possibilitou construir, dentre outros, um pequeno escalonador.