Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 8: Exceptional Control Flow
May 20, 2024

Practice Problems

Exercise 8.1. Consider three processes with the following starting and ending times:

Process	Start time	End time
A	0	2
B	1	4
C	3	5

For each pair of processes, indicate whether they run concurrently (Y) or not (N).

Solution: A pair of processes is concurrent if one starts after the other begins and before the other one ends. A and B are concurrent because B starts at $t = 1$, which is after the start time of A ($t = 0$) and the end time of A ($t = 2$). A and C are not concurrent because A starts at 0 and C has not yet begun (it starts at $t = 3$). Lastly, B and C are concurrent because C begins at $t = 3$ after B has started ($t = 1$) but before B ends ($t = 4$).

Process pair	Concurrent?
AB	Yes
AC	No
BC	Yes

Exercise 8.2. Consider the following program:

```c
int main()
{
    int x = 1;

    if (Fork() == 0)
        printf("p1: x=%d\n", ++x);
    printf("p2: x=%d\n", --x);
    exit(0);
}
```

(a) What is the output of the child process?

(b) What is the output of the parent process?

Solution:

(a) `fork()` system call in the child process returns 0. Therefore, both print statements execute. The output is:

```
p1: x=2
p2: x=1
```

(b) The `fork()` system call in the parent returns the process ID of the child, which is guaranteed to be a positive integer. Therefore, only the last print statement executes:

```
p2: x=0
```

Exercise 8.3. List all of the possible output sequences for the following program:

```
int main()
{
    if (Fork() == 0) {
        printf("a"); fflush(stdout);
    }
    else {
        printf("b"); fflush(stdout);
        waitpid(-1, NULL, 0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
```

Solution:

1. One possibility is that the child process completes all of its instructions before the parent gets a time slice after the work. Then the parent displays `b`, calls `waitpid` to reap the child but returns immediately because the child has already exited, and finally, the parent displays `c` before it too exits:

   ```
   acbc
   ```

2. Another possibility is that the child gets control after the work before the parent, displays `a`, but is then preempted. The parent begins executing and displays `b`. Then either it executes `waitpid` or is preempted before it does, allowing the child to display `c`. Finally, the parent displays `c`:

   ```
   abcc
   ```

3. Another possibility is that immediately after the work, the parent has the opportunity to displays `b`. Then the child gets control because the parent's time slice ends or because the parent calls `waitpid`. At this point, the child displays `a`. Since it has not terminated, the parent remains suspended due to `waitpid`, so the child gets to display `c`. Finally the parent reaps the child and displays `c`:

```
bacc
```

Exercise 8.4. Consider the following program:

```
int main()
{
    int status;
    pid_t pid;

    printf("Hello\n");
    pid = Fork();
    printf("%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1, &status, 0) > 0) {
            if (WIFEXITED(status) != 0)
                printf("%d\n", WEXITSTATUS(status));
        }
    }
    printf("Bye\n");
    exit(2);
}
```

(a) How many output lines does this program generate?

(b) What is one possible ordering of these output lines?

Solution:

(a) Assuming no errors, the child displays two lines: it executes `printf` statement immediately following the `Fork()`, skips the `if` block because its `pid` value is 0, and displays the `Bye` message. The parent, on the other hand, displays three lines: the one immediately after the `Fork()`, the exit status of the child inside the nested `if`, and then `Bye`. In total there are 5 output lines.

(b) Since the `stdout` stream from `stdio.h` is line-buffered by default, all lines will be displayed in the order they execute because they all have a
`n` as the last character in their strings. The initial `Hello` is always printed first by the parent. One possibility is that the parent gets control immediately after the work, displaying the result of the second `printf` call. Since the result of the `fork` call in the parent is the process ID of the child, the value of `pid` is a positive integer, so the expression `!pid` evaluates to 0, which in turn is displayed.

Control is then gained by the child because either the parent is preempted or because it is willfully suspended as a result of the `waitpid` call. The child now calls its first `printf` call; since `pid` is 0 in the child, the result is that `!pid` is 1, so that its printed to standard out. Since the parent is waiting for the parent to terminate, the child is able to print its next statement, displaying `Bye`. Assuming no errors occur and the `exit(2)` call runs successfully, the `waitpid` calls returns the (positive) process ID of the child, entering the branch of the if-statement. The `WIFEXITED` macro detects that the child exited normally by calling `exit`, so the next if-statement also evaluates to true. Then the `WEXITSTATUS` macro returns the exit status, which was 2, and that value is displayed by `printf`. Finally,t he `Bye` message is displayed by the parent:

```
Hello
0
1
Bye
2
Bye
```

Exercise 8.5. Write a wrapper function for `sleep`, called `snooze`, with the following interface:

```
unsigned int snooze(unsigned int secs);
```

The `snooze` function behaves exactly as the `sleep` function, except that it prints a message describing how long the process actually slept:

```
Slept for 4 of 5 secs.
```

Solution: Below is my implementation, which is found at `./05-snooze/snooze.c`:

```
#include <unistd.h> /* sleep() */
#include <stdio.h> /* printf() */
#include "snooze.h"

unsigned int
snooze(unsigned int secs)
{
    unsigned int remaining = sleep(secs);
    printf("Slept for %d of %d secs\n", secs - remaining, secs);
    return remaining;
}
```

Exercise 8.6. Write a program called `myecho` that prints its command-line arguments and environment variables. For example:

```
linux> ./myecho arg1 arg2
Command-line arguments:
    argv[ 0]: myecho
    argv[ 1]: arg1
    argv[ 2]: arg2
```

```
Environment variables:
    envp[ 0]: PWD=/usr0/doh/ics/code/ecf
    envp[ 1]: TERM=emacs
    .
    .
    .
    envp[25]: USER=droh
    envp[26]: SHELL=/usr/local/bin/tcsh
    envp[27]: HOME/usr0/droh
```

Solution: Below is my implementation, which is found at `./06-myecho/myecho.c`:

```c
#include <stdio.h> /* printf() */

int
main(int argc, char *argv[], char *envp[])
{
    /* Compute width of argv indices */
    char buf[BUFSIZ];
    int argvIndexWidth = snprintf(buf, BUFSIZ, "%d", argc);

    /* Display command line arguments */
    printf("Command-line arguments\n");
    for (int i = 0; i < argc; i++)
        printf("\targv[%*d]: %s\n", argvIndexWidth, i, argv[i]);

    /* Count number of env variables and compute envp index width */
    int envpCount = 0;
    for (char **p = envp; *p != NULL; p++)
        envpCount++;
    int envpIndexWidth = snprintf(buf, BUFSIZ, "%d", envpCount);

    /* Display environment variables */
    printf("Environment variables:\n");
    for (int i = 0; i < envpCount; i++)
        printf("\tenvp[%*d]: %s\n", envpIndexWidth, i, envp[i]);
}
```

Exercise 8.7. Write a program called `snooze` that takes a single command-line argument, calls the `snooze` function from Problem 8.5 with this argument, and then terminates. Write your program so that the user can interrupt the `snooze` function by typing CTRL+C at the keyboard. For example:

```
linux> ./snooze 5
CTRL+C     # User hits CTRL+C after 3 seconds
Slept for 3 of 5 secs.
linux>
```

Solution: Funnily, I already wrote this program as part of Problem 8.5 to test my program. See `./05-snooze/main.c`:

```c
#include <stdio.h> /* stderr, fprintf(), printf() */
#include <stdlib.h> /* exit(), EXIT_FAILURE, EXIT_SUCCESS, strtol() */
#include "snooze.h" /* snooze() */

#define DEFAULT_SLEEP 5

/* Parse duration command-line argument */
unsigned int parseDuration(char *arg)
{
#define ANY_BASE 0
    char *endp;
    long duration = strtol(arg, &endp, ANY_BASE);
    if (*endp != '\0') {
        fprintf(stderr, "Invalid character while parsing duration %s: %s\n", arg,
            endp);
        exit(EXIT_FAILURE);
    }
    /* Unchecked: parsed sleep may be very large */
    if (duration <= 0) {
        fprintf(stderr, "Invalid sleep duration: %ld\n", duration);
        exit(EXIT_FAILURE);
    }
    /* Truncate if too large */
    printf("Parsed duration as %u\n", (unsigned int) duration);
    return (unsigned int) duration;
}

#define DURATION_ARG 1

/* Snooze for the specified number of time, or the default duration */
int
main(int argc, char *argv[])
{
    unsigned duration;

    if (argc > 2) {
        fprintf(stderr, "Usage: %s [duration]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    duration = (argc > 1) ? parseDuration(argv[DURATION_ARG]) : DEFAULT_SLEEP;
    printf("Sleeping for %u seconds...\n", duration);
    snooze(duration);

    exit(EXIT_SUCCESS);
}
```

Exercise 8.8. What is the output of the following program?

```
volatile long counter = 2;

void handler1(int sig)
{
    sigset_t mask, prev_mask;

    Sigfillset(&mask);
    Sigprocmask(SIG_BLOCK, &mask, &prev_mask); /* Block sigs */
    Sio_put1(--counter);
    Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */

    _exit(0);
}

int main()
{
    pid_t pid;
    sigset_t mask, prev_mask;

    printf("%ld", counter);
    fflush(stdout);

    signal(SIGUSR1, handler1);
    if ((pid = Fork()) == 0) {
        while(1) {};
    }
    Kill(pid, SIGUSR1);
    Waitpid(-1, NULL, 0);
    Sigfillset(&mask);
    Sigprocmask(SIG_BLOCK, &mask, &prev_mask); /* Block sigs */
    printf("%ld", ++counter);
    Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */

    exit(0);
}
```

Solution: Initially, the parent outputs 2, the initial value of the global `counter` variable. By default, the `SIGUSR1` signal is ignored. The program calls the `signal()` function to install a handler for `SIGUSR1` before it forks. When the child gains control, its `Fork()` call returns 0, causing it to enter an infinite loop. The parent, on the other hand, uses the `Kill` system call request the kernel deliver a `SIGUSR1` signal to the child on its behalf. Because the signal handler was installed before the fork and the `SIGUSR1` signal is not blocked, the process receives the signal and catches it with the `handler1` signal handler. The handler blocks all incoming signals with `Sigfillset()` and `Sigprocmask`, decrements `count`, and then uses the async-signal-safe function `Sio_put1` to displays its new value. It then restores the mask and terminates normally by calling `_exit()`.

The parent reaps the child with `Waitpid`, blocks signals as before so that it can increase `count` to 3 and display its new value with `printf` uninterrupted, and then terminates. Keep in mind that after the `Fork()`, the data segments of the two processes are private, so the modification to the value of `counter` in the child does not affect `counter` in the parent. The output is:

```
213
```

Exercise 8.9. Consider the four processes with the following starting and ending times:

Process	Start time	End time
A	5	7
B	2	4
C	3	6
D	1	8

For each pair of processes, indicate whether they run concurrently (Y) or not (N).

Solution: See approach in practice problem 8.1

Process pair	Concurrent?
AB	N
AC	Y
AD	Y
BC	Y
BD	Y
CD	Y

Exercise 8.10. In this chapter, we have introduced some functions with unusual call and return behaviors: `setjmp`, `longjmp`, `execve`, and `fork`. Match each function with one of the following behaviors:

(a) Called once, return twice.

(b) Called once, never returns.

(c) Called once, returns one or more times.

Solution:

(a) Called once, return twice: This is characteristic of `fork()`. It creates a child process. In the newly created child, it returns 0; in the parent, it returns the process ID of the child.

(b) Called once, never returns: The `longjmp` function jumps to a section previously marked by a call to `setjmp`. The `execve` loads and runs the executable at the pathname provided in its first argument in the context of the calling process. If successful, it never returns. It overwrites the address space of the current process and it does not create a new process, with the same PID, and it inherits all file descriptors that were open at the time of the call.

(c) Called once, returns one or more times: The `setjmp` call marks a point that `longjmp` can jump to by first saving the current environment, including the program counter, stack pointer, and general-purpose registers. Its initial call returns 0; later calls return a value passed to the second argument of `longjmp`.

Exercise 8.11. How many "`hello`" output lines does this program have?

```
#include "csapp.h"

int main()
{
    int i;

    for (i = 0; i < 2; i++)
        Fork();
    printf("hello\n");
    exit(0);
}
```

Solution: The first `Fork()` creates a child. At this point, both parent and child are in the loop with `i = 0`, about to start the iteration with`i=1`. Then, both call `Fork()`, each creating a new process. At this point, there are 4 processes in existence, and `i` increases to 2 in all of them, causing them all to exit the loop and proceed to `printf`. The `hello` message is then printed 4 times, one per line.

Exercise 8.12. How many "`hello`" output lines doe this program print?

```
#include "csapp.h"

void doit()
{
    Fork();
    Fork();
    printf("hello\n");
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

Solution: The call to `doit()` starts with a `Fork()`. Then parent and the child then both call `Fork()`. Now there are 4 processes in existence, all currently in the `doit` procedure about to print `hello`. After all 4 output hello, they all move back to `main` and also print `hello`. Therefore there are 8 "`hello`" output lines.

Exercise 8.13. What is one possible output of the following program?

```
#include "csapp.h"

int main()
{
    int x = 3;

    if (Fork() != 0)
        printf("x=%d\n", ++x);
    printf("x=%d\n", --x);
    exit(0);
}
```

Solution: In general, the output is undetermined because we cannot guarantee the order of execution of the parent and child process after the call to `Fork()`. Suppose that the child gets control first. Its `Fork()` call returns 0, so it skips if-statement block. Assuming it has the chance to execute `printf` before it is preempted, we will see the text `x=2` displayed on the screen. When control returns to the parent process, its `Fork()` call returns the nonzero process ID of the child, which at this point may have already terminated and is a zombie. The parent prints `x=4` because its data segment and stack are separate from the child's, which means that its value of `x` is unchanged by the previous `--x` in the child. Finally, it decrements `x` back to 3 and prints 4. One possible output is thus:

```
243
```

Exercise 8.14. How many "hello" output line does this program print?

```
#include "csapp.h"
void doit()
{
    if (Fork() == 0) {
        Fork();
        printf("hello\n");
        exit(0);
    }
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

Solution: The parent calls `doit()`, which in turn calls `Fork()`. In the parent, this returns the nonzero process ID of the newly created child, so the parent returns from `doit`, and prints `hello` before exiting. In the child, `Fork()` returns 0, and in its `if`-statement body it proceeds to also call `Fork()`. Both it and its child print the `hello` message in the `if`-statement, but they call `exit(0)`, so they do not print the `hello` from `main`. Overall, 3 `hello` lines are output.

Exercise 8.15. How many "hello" lines does this program print?

```
#include "csapp.h"

void doit()
{
    if (Fork() == 0) {
        Fork();
        printf("hello\n");
        return;
    }
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

Solution: The parent calls `doit()`, which in turn calls `Fork()`. In the parent, this returns the nonzero process ID of the newly created child, so the parent returns from `doit`, and prints `hello` before exiting. In the child, `Fork()` returns 0, and in its `if`-statement body it proceeds to also call `Fork()`. Both it and its child print the `hello` message in the `if`-statement, and both print the `hello` back in main. Overall, 5 "`hello`" lines are output by this program.

Exercise 8.16. What is the output of the following program?

```
#include csapp.h"
int counter = 1;

int main()
{
    if (fork() == 0) {
        count--;
        exit(0);
    }
    else {
        Wait(null);
        printf("counter = %d\n", ++count);
    }
}
```

Solution: The parent calls `fork()`, which returns 0 in the child, causing it to enter the `if`-statement. The child decrements `count` and immediately exits. In the parent, the `Wait` call suspends the parent until the child terminates. Its `counter` value is unaffected by the child. The parent uses `++count` to increment its value to 2, and then displays its value:

```
counter = 2
```

Exercise 8.17. Enumerate all of the possible outputs of the program in Practice Problem 8.4.

Solution: The parent prints `Hello` and then forks. After, the following disjoint outcomes may occur:

(a) Child gets control. Its `Fork()` call returns 0, assigning that to the `pid` variable. The `printf` outputs the value of `!pid`, which is 1. Then the child process calls `Bye` and exits. The parent gains control, outputs `0` because its `Fork()` call returns the nonzero value of its child's PID, which meaning `!pid` is 0. The parent suspends while it waits to reap its child by calling `waitpid`, but the call returns immediately because the child has already terminated. The following `if`-statements are entered under "normal" conditions, so the exit status of 2 of its child is printed. Finally, It prints`Bye`:

```
Hello
1
Bye
0
2
Bye
```

(b) Again, child gets control, and it has a chance to output `1`. But it is preempted, giving control the parent, which now prints `0`. The parent is preempted, or it reaches the `waitpid` call; in either case, the child regains control and prints `Bye`. The parent regains control and the same as in (a) happens:

```
Hello
1
0
Bye
2
Bye
```

(c) The child gets control first again. However it is preempted before outputting 1. The parent prints 0, and it is either preempted or is suspended due to `waitpid`. Now the child can print all of its lines, and then the parent can print the rest of its lines:

```
Hello
0
1
Bye
2
Bye
```

(d) The parent gets control, printing 0. Then it is preempted or the is suspended by `waitpid`, giving control to the child. Control won't return to parent until the child has finished, so the child prints all of its lines and then the parent does the same:

```
Hello
0
1
Bye
2
Bye
```

Exercise 8.18. Consider the following program:

```c
#include "csapp.h"

void end(void)
{
    printf("2"); fflush(stdout);
}

int main()
{
    if (Fork() == 0)
        atexit(end);
    if (Fork() == 0) {
        printf("0"); fflush(stdout);
    }
    else {
        printf("1"); fflush(stdout);
    }
    exit(0);
}
```

Determine which of the following outputs are possible. *Note*: The `atexit` function takes a pointer to a function and adds it to a list of functions (initially empty) that will be called when the `exit` function is called.

(a) 112002

(b) 211020

(c) 102120

(d) 122001

(e) 100212

Solution:

(a) This outcome is possible as follows. After *parent* has called `Fork()`, it has created *child1*. When *child1* gets control, it registers `end` with `atexit`. Both *parent* and *child1* call `Fork()` again in the expression within the second `if`-statement; *parent* creates *child2* and *child1* creates *grandchild*. If `parent` gets control first, it prints 1 and exits. If `child1` gets control after, it too prints 1. If it exits and its `atend`is called before it is preempted, it displays 2. If *child2* then gains control, it displays 0 and exits. In this case. `end` is not called because *child2* is a child of *parent*, which did not call `atexit`, so this registration is not inherited. However, when *grandchild* gets control, it prints 0, and it also prints 2 because it inherits the registration of `end` from *child1*.

(b) This is impossible. Either *child1* or *grandchild* has to execute first to display `2`. But if *child1* executes first, it displays 1 before exiting; if *grandchild* executes first, it displays 0 before it exiting.

(c) This outcome is possible. *parent* may receive control first. Then *grandchild* receives control afterwards, displaying 0 and exiting; since it inherited the `atexit` registration from *child1*, it prints 2.. Next, *child1* gets control, displaying 1 and exiting, causing it to output 2. Lastly, *child2* displays 0.

(d) Impossible. This implies *child1* gets control and runs to completion, causing it to print `12`. But the other 2 has to come from *grandparent*, which must print 1 before its registered `end` function, which is called upon exit, prints 2.

(e) Its possible. The parent gets control and runs through completion, printing 1. Then *child2* runs to completion, printing 0. Then, *child1* runs to completion, printing 02. Finally, *child1* runs to completion, printing 12.

Exercise 8.19. How many lines of output does the following function print? Give your answer as a function of n. Assume $n \geq 1$.

```c
void foo(int n)
{
    int i;

    for (i = 0; i < n; i++)
        Fork();
    printf("hello\n");
    exit(0);
}
```

Solution: It prints 2^n lines.

Exercise 8.20. Use `execve` to write a program called `myls` whose behavior is identical to the `/bin/ls` program. Your program should accept the same command-line arguments, interpret the identical environment variables, and produce the identical output.

The `ls` program gets the width of the screen from the `COLUMNS` environment variable. If `COLUMNS` is unset, then `ls` assumes that the screen is 80 columns wide. Thus, you can check your handling of the environment variable by setting the `COLUMNS` environment to something less than 80:

```
linux> setenv COLUMNS 40
linux> ./myls
 .
 . // Output is 40 columns wide
 .
linux> unsetenv COLUMNS
linux> ./myls
 .
 . // Output is now 80 columns wide
 .
```

Solution: See `./20-myls/myls.c`:

```c
#include <unistd.h> /* execve */
#include <stdlib.h> /* exit(), EXIT_SUCCESS, EXIT_FAILURE */
#include <stdio.h> /* fprintf(), stderr */

#define LS_PATH "/bin/ls"

/* Runs ls */
int
main(int argc, char *argv[], char *envp[])
{
    if (execve(LS_PATH, argv, envp) == -1) {
        fprintf(stderr, "%s: Failed to exec %s\n", argv[0], LS_PATH);
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Exercise 8.21. What are the possible output sequences from the following program?

```
int main()
{
    if (fork() == 0) {
        printf("a"); fflush(stdout);
    }
    else {
        printf("b"); fflush(stdout);
        waitpid(-1, NULL, 0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
```

Solution:

1. `bacc`: The parent gets control immediately after the `fork`. It is preempted because it time slice has run out, or suspended by `waitpid`. The child prints `ac`. The parent finally prints `c`.

2. `acbc`: The child gets control first and runs through completion, printing `ac`. Then the parent runs through completion, printing `bc`.

3. `abcc`: The child gets control, prints `a`, then is preempted, allowing the parent to print `b`. Finally, the child prints `c`, and then the parent prints `c`.

Exercise 8.22. Write your own version of the Unix `system` function

```
int mysystem(char *command);
```

The `mysystem` function executes `command` by invoking `/bin/sh -c command` and then returns after `command` has completed. If `command` exits normally (by calling the `exit` function), then `mysystem` returns the `command` exit status. For example, if `command` terminates by calling `exit(8)`, then `mysystem` returns the value 8. Otherwise, if `command` terminates abnormally, then `mysystem` returns the status returned by the shell.

Solution: See `./22-mysystem/mysystem.c`:

```c
#include <sys/types.h> /* pid_t */
#include <sys/wait.h> /* waitpid() */
#include <unistd.h> /* execve(), fork() */
#include "mysystem.h"

#define SHELL_PATH "/bin/sh"

extern char **environ;

int mysystem(char *command)
{
    pid_t pid;
    int status;
    char *args[4] = { SHELL_PATH, "-c", command, NULL};
    switch(pid = fork()) {
        case -1: /* Failure */
            return -1;
        case  0: /* Child */
            if (execve(SHELL_PATH, args, environ) == -1)
                return -1;
        default: /* Parent */
            if (waitpid(pid, &status, 0) == -1)
                return -1;
            if (WIFEXITED(status))
                return WEXITSTATUS(status);
            else if (WIFSIGNALED(status))
                return WSTOPSIG(status);
            return -1;
    }
}
```

Exercise 8.23. One of your colleagues is thinking of using signals to allow a parent process to count events that occur in a child process. The idea is to notify the parent each time an event occurs by sending it a signal and letting the parent's signal handler increment a global `counter` variable, which the parent can inspect after the child has terminated. However, when he runs the test program in Figure 8.45 on his system, he discovers that when the parent calls `printf`, `counter` always has a value of 2, even though the child has sent five signals to the parent.

```c
#include "csapp.h"

int counter = 0;

void handler(int sig)
{
    counter++;
   sleep(1); /* Do some work in the handler */
   return;
}

int main()
{
   int i;

   Signal(SIGUSR2, handler);

   if (Fork() == 0) { /* Child */
       for (i = 0; i < 5; i++) {
           Kill(getppid(), SIGUSR1);
           printf("sent SIGUSR2 to parent\n");
       }
       exit(0);
   }

   Wait(NULL);
   printf("counter=%d\n", counter);
}
```

Perplexed, he comes to you for help. Can you explain this bug?

Solution: The child requests that 5 signals be sent to its parent. When the first signal is delivered by the kernel and received by the parent, the handler catches the signal and increments `counter` to 1, then suspends for 1 second via the `sleep(1)` statement. During the execution of the handler, delivery of signals of type `SIGUSR2` are blocked, so they are added to the set of pending signals. A duration of 1 second is humongous compared to the time it takes the child to send the 5 signals. Therefore the remaining 4 signals are sent while the handler in the parent is executed. The first of the remaining 4 signals is added to the set of pending signals. However, because signals of the same type are not queued, the rest of the signals are discarded. When the parent process continues and the handler returns, the kernel delivers the pending `SIGUSR2` signal, incrementing `counter` to 2. Again it suspends and then it ends. Control returns to the `main` procedure, where it was waiting for its child to complete. It has long since completed, so the parent reaps the child and prints the value of `counter`, which is 2.

Exercise 8.24. Modify the program in Figure 8.18 so that the following conditions are met:

(a) Each child terminates abnormally after attempting to write to a location in the read-only text segment.

(b) The parent prints output that is identical (except for the PIDs) to the following:

```
child 12255 terminated by signal 11: Segmentation fault
child 12254 terminated by signal 11: Segmentation fault
```

Hint: Read the `man` page for `psignal(3)`.

Exercise 8.25. Write a version of the `fgets` function, called `tfgets`, that times out after 5 seconds. The `tfgets` function accepts the same inputs as `fgets`. If the user doesn't type an input line within 5 seconds, the `tfgets` returns `NULL`. Otherwise, it returns a pointer to the input line.

Solution: See my implementation at `./25-tfgets/tfgets.c`:

```c
#include <stdio.h> /* fgets(), FILE, NULL */
#include <signal.h> /* struct sigaction, sigaction(), sig_atomic_t, SIGALRM */
#include <errno.h> /* errno */
#include <unistd.h> /* alarm() */

#define TIMEOUT 5

static volatile sig_atomic_t interrupted = 0;

static void
sigAlarmHandler(int sig)
{
    interrupted = 1;
}

char *tfgets(char *s, int size, FILE *stream)
{
    /* Install SIGALRM handler */
    struct sigaction sa, oldsa;
    sa.sa_flags = 0; /* Do not restart system calls */
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = sigAlarmHandler;
    if (sigaction(SIGALRM, &sa, &oldsa) == -1)
        return NULL;

    /* Schedule SIGALRM and get input */
    alarm(TIMEOUT);
    char *r = fgets(s, size, stream);

    /* Restore handler */
    if (sigaction(SIGALRM, &oldsa, NULL) == -1)
        return NULL;

    /* Remove alarm if still pending */
    int savedErrno = errno;
    alarm(0);
    errno = savedErrno;

    /* Assign depending on whether call was interrupted */
    r = (interrupted) ? NULL : r;

    /* Restore flag */
    interrupted = 0;

    return r;
}
```