1. [4 points] On a computer running Linux, read the manual entry for the command kill.
   
   a. List the names of all signals available in the Linux computer you are using.  
      (Hint: You can do this easily with the kill command and the proper option.)
   
   b. Which UNIX signal in 1.a. is generated when you execute a program from a shell and hit ^C (control-C) on the keyboard before the program completes execution to kill the program?

2. [10 points] Normally if you start a program from a UNIX shell and hit ^C the program immediately terminates. In this exercise, write a non-terminating C program that, when executed in the foreground from a UNIX shell, ignores the interrupt key (^C). Turn in the source code, but do not turn in any output from the program.
   
   Hints:
   a. Use a computer running Linux for this problem.
   b. Read the man entry for function signal. You will need to call signal one time to indicate to the OS that your program should ignore the signal generated by ^C.
   c. You can make your program non-terminating by inserting an infinite loop in the program.
   d. You do not need to write more than five lines of C code to solve this problem.
   e. When you start the program from a shell, and hit ^C, nothing will happen – the program will continue executing (the infinite loop). Because you will be ignoring ^C in your program, you will have to be a bit clever when you are done testing your program and really want to terminate the program. You could terminate the program by either hitting ^Z to suspend the program (and then kill the suspended program with the kill command) or simply kill the shell to terminate your program.

3. [10 points] Modify the program you wrote for exercise 2 so that, when executed in the foreground from a UNIX shell, the program prints "Hello world\n" to stdout whenever the interrupt key (^C) is hit. Turn in the source code, but do not turn in any output from the program.
Hint: Write a function named `hello()` that prints "Hello world\n" to stdout. In your main program, call `signal` to get your program to call the "Hello world\n" function when ^C is hit.

4. [6 points] The purpose of this exercise is twofold. The first purpose is to observe that multiple executions of a program with concurrency can yield different execution orders. The second purpose is to give you a chance to compile and execute a Java program.

If you do not have a Java compiler on your computer, then download and install the Sun Java 2 SDK from java.sun.com. Type the program in Figure 4.9 into a file named `ThreadTester.java`. Compile the program by typing `javac ThreadTester.java`. Run the program three times by typing `java ThreadTester`. Turn in the output of the three executions as the solution to this assignment. Did you observe a different order of termination of the threads in your three outputs?


Solution

1a. Shown below are the signals available in Cygwin running on Windows 2000. Your list may vary depending on what shell you used when you executed the `kill -l` command.

```
~ [501]:kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
9) SIGHUP       10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGURG
17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGLOST     30) SIGUSR1     31) SIGUSR2
```

1b. The answer is SIGINT, which is listed in Figure 20.24 in the book as “interrupt from the keyboard”.

2.

```c
#include <signal.h>
main() {
    signal(SIGINT,SIG_IGN);
    while (1) { ; }
}
```

3.

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

void hello() {
    signal(SIGINT,hello);    // needed in some UNIX systems (e.g., linux)
    printf("Hello world\n");
}

main() {
    signal(SIGINT,hello);
    while (1) { ; }
}
```

4. Here is an example of output from three executions (your solution will look different). Note that the thread end in a different order the third time.

D:\VaTech\CS3204\CS3204 - 2004 Spring\Deitel Book\CodeExamples>java ThreadTester

Starting threads
Threads started, main ends

thread1 going to sleep for 40 milliseconds
thread2 going to sleep for 1246 milliseconds
thread3 going to sleep for 614 milliseconds
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping

>java ThreadTester

Starting threads
Threads started, main ends

thread1 going to sleep for 1098 milliseconds
thread2 going to sleep for 3743 milliseconds
thread3 going to sleep for 2255 milliseconds
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping

>java ThreadTester

Starting threads
Threads started, main ends

thread1 going to sleep for 4727 milliseconds
thread2 going to sleep for 2652 milliseconds
thread3 going to sleep for 2781 milliseconds
thread2 done sleeping
thread3 done sleeping
thread1 done sleeping

>

5. Fork and clone are alike in these ways:
   - From the scheduler’s viewpoint, both calls create a new scheduler activation.

Fork and clone differ in these ways:
   - Clone is quicker to execute because a cloned thread shares main memory with the parent.
   - With clone, the child can share resources (e.g., file descriptors) with the parent.

6. (To be added later.)

7. Here is one example. Consider Figure 5.15. Suppose there are two threads. Thread T1 starts doing \( P(\text{occupied}) \); and completes if \( \text{occupied} > 0 \) when its time quantum expires. Thus shared semaphore value \( \text{occupied} \) still has value 1. Thread T2 is scheduled next and
also executes $P(occupied)$ to completion (thereby setting $occupied$ to 0) and enters the critical section. While T2 is in the critical section, its time quantum expires and T1 is scheduled. T1 resumes execution of $P(occupied)$ by decrementing $occupied$ to -1 and entering the critical section. We now have both T1 and T2 in the critical section, and mutual exclusion is not enforced.