Department of Systems and Computer Engineering, Carleton University
SYSC 4001 Operating Systems Winter 2018-19: Lab Exercise

Lab Exercise 3 - Process and Signals; Shared memory

This lab exercise is mainly based on the materials of Chapter 11 *Processes and Signals* and Chapter 14 (material on Shared Memory) in the Linux reference book. The main topics for the lab exercise include:

- Command line arguments (review and extension, useful for assignments)
- Process creation (review and extension)
- Signals
- Shared Memory (Chapter 14)

1. Command Line Arguments

- Review the description and the example on page 137 and page 138. For assignments, there is no need to use optional argument, e.g., -x.
- Write a simple program called sensor.c that gets information from the command line. The program prints the value of argc and each argument. Compile and name it sensor.
- Run the program multiple times. Ex:

    - $./sensor Temp1 23

    This command could be used to name the sensor Temp1 and set the number, e.g., 23, as the threshold.

    - $/sensor Temp2 23 19

    This command could be used to name the sensor Temp2 and pass two threshold values to the program.

  This program can be further extended to identify the type of sensor. Ex:
    - $/sensor temperature TempABC 23 19
    - $/sensor humidity 30

  In the program, the second argument, e.g., argv[1], can be used to identify the type of sensor.

2. Duplicating a Process Image

Review Figure 11-2 for the fork() function and make sure you understand how it works.

3. Waiting for a process using the *wait()* system call (wait.c on p. 475 – p. 477)

"The wait() system call causes a parent process to pause until one of its child processes is stopped. The call returns the PID of the child process." *wait()* is an important system call for process synchronization.

3.1. Identify the code that belongs to the parent and the code for the child. In the parent's code, add a print statement to print the pid from the fork(). Compare the value and the value for child_pid in the program.

3.2. For process state covered in lecture:

3.2.1. What is the state for the parent process after it makes the *wait()* system call?

3.2.2. Zombie Processes are described on page 477 and page 478 (first half page): Why is the Zombie state needed?

4. Signals and signal handling

"A signal is an event generated by Unix/Linux systems in response to some condition, upon receipt of which a process may in turn take some action."

4.1. Read the description for Signals (p. 481). You can skip the details for various signal names. What are the usages of signals (p. 481)? Go over the terms *raise* and *catch* for signals on p. 481.

4.2. Try it Out: Compile *alarm.c* (p. 485) and run it. Go over the explanation and walk though the code.

   4.2.1. Identify the code for signal handler.

   4.2.2. Identify the code for the parent and the code for the child.

   4.2.3. What signal is used and what is the format to link the signal and the signal handler?

   4.2.4. What does *kill()* do? What are the parameters?

   4.2.5. What is *pause()* used for?

   4.2.6. Change the *pause()* statement to an infinite loop. Compile it and run it. Compare it with the original *alarm.c*. Is there any difference from the user's perspective? What is the benefit of using *pause()*?

4.3. Using signals may not always be safe, i.e., some signals may be lost. A robust signals interface, sigaction, can be used. Modify *alarm.c* using sigaction (p.487–p.488), see the example on p. 488. Compile the code and run it.

NOTE: for systems calls, e.g., sigaction, use default values for system parameters. In other words, follow examples in the book, unless you are sure what other values mean.

Shared Memory

This section focuses on learning to use Shared Memory on Linux (discussed in Chapter 14 (starting on pg. 586) of the Linux reference book.

Accessing a shared data structure implies that the concurrent processes share access to the same main memory. That is the default for threads within the same process (sharing access to all data). But for processes, the OS usually ensures that each process has its own dedicated share of main memory and protects the memory assigned to process X from being accessed by process Y. However, in some cases, two or more **processes** may want to share access to joint data structures. This lab exercise makes you familiar with the facilities Linux provides for **processes** to set up and use shared memory in a controlled manner.

Shared memory provides an efficient way of sharing and passing data between multiple processes. However, shared memory doesn't provide any synchronization facilities. It's the responsibility of the programmer to synchronize access. Read the discussion on shared memory starting on page 586 of the Linux book

The primary functions for shared memory are listed in the book.

- **shmget()** and **shmctl()** are used for creating and for controlling shared memory. Read the description from the book.

- **shmat():** When you first create a shared memory segment, it's not accessible by any process. To enable access to the shared memory, you must attach it to the address space of a process. shmat() enables you to achieve this objective.

- Another point to remember is that you will need to format the shared memory according to your data structure or to assign the shared memory to your own data structure.

- The **shmdt()** function detaches the shared memory from the current process.

Try it out – Shared Memory

1. Run shm1.c and shm2.c from the Linux book. Go over the code and the description for the four shared memory functions and how they are used in the programs.

2. Write a small program:

   The program creates a child using fork(). Add a piece of shared memory for both the parent and child processes. The child process continuously generates a random number and puts it in the shared memory. If the random number is over a pre-configured threshold, the child process sends a signal to the parent which reads the current value from the shared memory. (You will find examples of using the shared memory functions in the example provided in the book.)

Submit an archive (using the command `tar`) containing your `Makefile` and all your source files (both `*.c` and `*.h`) on `cuLearn`