Tutorial 2 : Process Management

August 20, 2019

Objective :

- This assignment is intended to learn how to create, work with and manipulate processes in Linux. You are expected to refer to the text book and references mentioned in the course website befor you start the lab. Some sample codes for process creation using *fork* system call have been provided for your reference.

Instructions

- You are expected to run all the sample codes provided in the *Helpful Resources* section. It will help you understand how OS virtualizes CPU and memory. Some of the codes are related to process creation and execution using Unix based system calls such as *fork,exit,wait and exec*

Tutorials

1. Tut1 : Memory Virtualization :

   Physical memory model presents Memory as an array of bytes. The data stored at an address of memory can be read by providing the address of the location wehere the data resides. Similarly, while writing data to any address of a memory, one needs to specify the address for writing as well as the data to be written to the given address. Memory is accessed all the time when a program is running. A program keeps all of its data structures in memory, and accesses them through various instruction. Instructions themeselves reside in the memory which have to be fetched.

   Instructions : Execute the code *mem.c* from the *Helpful Resources* section. Make sure to include the *common.h* file in the same folder as the *mem.c*. To execute use the following commands : *gcc mem.c*
   ./a.out ox200000

   The program does a couple of things. First, it allocates some memory(line a1). Then, it prints out the address of the memory (a2), and the puts the number *zero* into the first slot of the newly allocated memory(a3). Finally, it loops, delaying for a second and incrementing the value stored at the

address held in *p*. With every *print* statement, it also prints out what is called the *process identifier (the PID)* of the running program. This *PID* is unique per running process. The newly allocated memory is at address **0x200000**. As the program runs, it slowly updates the value and prints out the result.

Now, we again run multiple instances of this same program to see what happens.

Instructions : Open another tab in the terminal and execute the code *mem.c* again. To execute use the following commands : *gcc mem.c*
./a.out ox200000

To check whether two instances of *mem* have been created, use the *ps -l* command in a third tab of the terminal. You will find two processes named *a.out* . These are your two instances of *mem.c*

It is worth noting that each running program has allocated memory at the same address **0x200000**, and yet each seems to be updating the value at **0x200000** independently! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs.
Indeed, that is exactly what is happening here as the OS is virtualizing memory. Each process accesses its own private *virtual address space*(sometimes just called its address space), which the OS somehow maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes(or the OS itself); as far as the running program is concerned, it has physical memory all to itself. The reality, however, is that physical memory is a shared resource, managed by the operating system.

2. Tut2 : Process Creation :
   Theoretical Background
   Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
   Processes are organized hierarchically. Each process has a parent process, which explicitly arranged to create it. The processes created by a given parent are called its child processes. A child inherits many of its attributes from the parent process. A process ID number names each process. A unique process ID is allocated to each process when it is created. The lifetime of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.
   Processes are created with the *fork()* system call (so the operation of creating a new process is sometimes called forking a process). The child

process created by *fork()* is a copy of the original parent process, except that it has its own process ID.
After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling *wait()* or *waitpid()*. These functions give you limited information about why the child terminated–for example, its exit status code.
A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from fork to tell whether the program is running in the parent process or the child process. When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action. A process that is waiting for its parent to accept its return code is called a *zombie process*. If a parent dies before its child, the child (*orphan process*) is automatically adopted by the original "init" process whose PID is 1

Instructions : Execute the code *fork1.c* and *fork1.c* from the *Helpful Resources* section to see how processes are created using *fork* system call.
About fork : The fork function is the primitive for creating a process. It is declared in the header file *"unistd.h"*. The fork function creates a new process.If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process. If process creation failed, fork returns a value of -1 in the parent process and no child is created.

The specific attributes of the child process that differ from the parent process are:

(a) The child process has its own unique process ID.

(b) The parent process ID of the child process is the process ID of its parent process. The child process gets its own copies of the parent process's open file descriptors.Subsequently changing attributes of the file descriptors in the parent process won't affect thefile descriptors in the child, and vice versa. However, both processes share the file position associated with each descriptor.

(c) The elapsed processor times for the child process are set to zero.

(d) The child doesn't inherit file locks set by the parent process.

(e) The child doesn't inherit alarms set by the parent process

(f) The set of pending signals for the child process is cleared.

Monitoring Process : To monitor the state of your processes under Unix use the ps command. Usually *ps -l]* which list of all currently running processes.

3. Tut3 : Creating Multiple Processes :

   Execute *multiple_fork.c* to see how mutiple processes can be created

4. Tut4 : Processes Completion :

   Execute *fork_wait.c* to see how a process (parent) can be made to wait until another process (child) gets completed.

   The *wait()* function will force a parent process to wait for a child process to stop or terminate. *wait()* returns the pid of the child or -1 for an error. The exit status of the child is returned to *status_ptr*. The function is declared in the header file *"sys/wait.h"* and the syntax is
   pid_t wait(int status_ptr)

   exit() terminates the process which calls this function and returns the exit status value. The syntax is : **void exit (int status)**
   Both UNIX and C (forked) programs can read the status value. By convention, a status of *0* means normal termination. Any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the *sys/stat.h* header file. We can easily derive our own conventions

   sleep() command suspends a process for a period of time
   Syntax : **unsigned int sleep (seconds))**

5. Tut5 : Orphan and Zombie Processes :

   Orphan process : When a parent dies before its child, the child is automatically adopted by the original *"init"* process whose PID is 1. To, illustrate this insert a sleep statement into the child's code. This ensured that the parent process terminated before its child

   Execute *orphan_process.c* to see an orphan process is created. Once the orphan has been created, use the *ps -l* command to find who becomes the parent of the orphan.

   Zombie process : A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it'll already have been adopted by the "init" process, which always accepts its children's return codes. However, if a process's parent is alive but never executes a *wait ( )*, the process's return code will never be accepted and the process will remain a zombie.

   Execute *zombie_process.c* to see how a zombie process is created. Use the *ps -l* command to find to indicate the zombie being created.

6. Tut6 : Execute a new program :

   Execute the code *fork_execute.c* from the *Helpful Resources* section to see how a process is created and then overwritten by a new program. The code here uses the *fork()* call to create a new process and then loads the child with the image of a new program (in this case the *"ls"*. The new program to overwrite the child is provided as argument to the *exec()* system call.

7. Tut7 : Modifying Data in Child Process :

   Execute the code *modify_data_in_child.c* from the *Helpful Resources* section to see how a data gets modified in the child process while the parent maintains the oroginal copy of the data. The interesting point to note would be how the same variable intialized to some value before the fork would retain the value in the parent but would be update din the child. However, the address of the variable in both parent and child would be the same. Identify the reason and report.

8. Assignments :

   (a) Write a program in C that creates a child process, waits for the termination of the child and lists its PID, together with the state in which the process was terminated (in decimal and hexadecimal)

   (b) In a C program, print the address of the variable and enter into a long loop (say using while(1)).

   - Start three to four processes of the same program and observe the printed address values.
   - Show how two processes which are members of the relationship parent-child are concurrent from execution point of view, initially the child is copy of the parent, but every process has its own data.

   (c) Test the source code below:

   $for(i = 1; i \leq 10; i++)\{$
   fork();
   printf("The process with the PID=%d",getpid());
   }

   In the next phase, modify the code, such as after all created processes have finished execution, in a file *process_management.txt* the total number of created processes should be stored.

   (d) Write two programs : one called *client.c*, the other called *server.c*. The client program lists a prompter and reads from the keyboard two integers and one of the characters '+' or '-'. The read information is transmitted with the help of the system call *excel* to a child process, which executes the server code. After the child (server) process finishes the operation, it transmits the result to parent process (client) with the help of the system call *exit*. The client process prints the result on the screen and also reprints the prompter, ready for a new reading.