CS 333: Operating Systems Lab
Autumn 2017

Lab 2: The matter of processes

Goal

The goal of this lab is to get comfortable with processes, from their creation to reaping dead processes, and to get used to system calls related to process creation and usage. After completing this lab you will find yourself capable of constructing multi-process systems.

Before you begin

You must use C/C++ for this lab.

A set of system calls that we will use in this lab are listed below. You should look up the appropriate man pages for details of each. You may have to use non-default man page sections for some of these. For example, you need to use `man 2 open` instead of `man open`.

- **fork**: This system call is used to create a new process. After the process is created, both parent and child processes continue execution from the point after `fork()`. The return value is different in child and parent processes: zero in the child and the process-id(`pid`) number of the child in the parent. In case of error, -1 is returned and new process is not created.
- **wait**: This call makes the calling process wait till a child process terminates and reclaims the resources associated to it. The return value is the `pid` of the child process. A variant of this, `waitpid` waits for a process with a given pid.
- **exec**: This call is used to run a new executable by replacing calling process's code with the executable program's code. The code after `exec()` in the original process is executed only if exec fails. If it succeeds, execution continues from the first line of the executable. Check out the several variants of this in the man page.
- **open**: This call is used to open a file or create one. The return value is the file descriptor of the opened file. This call assigns the smallest unused non-negative integer as the file descriptor.
- **close**: This call loses a file descriptor. The resources associated with the open file are freed.
- **read**: This call is used to read a specified number of bytes from a file descriptor into a buffer. It does not necessarily read the requested number of bytes as the file may have lesser number of bytes. The number of bytes read is returned.
- **write**: This call is used to write a specified number of bytes from a buffer to a file descriptor. The number of bytes written is returned
- **getpid**: This returns the process ID of the calling process.

Standard file descriptors which are opened for every process:
0 : Standard input (stdin)
1 : Standard output (stdout)
2 : Standard error (stderr)

Introduction to Processes

A process is a basic unit of execution in an OS. The main job of any OS is to run processes, while managing their life cycle from creation to termination. Processes are typically created in Unix-like systems by **forking** from an existing process.

- **Task 1: Baby steps to forking**
  Write a program `simple_fork.c` that forks a child and prints
  `Parent: The child's process ID is : {child_PID}`
  and then waits for the child process to exit. The child on execution prints

`Child: The child's process ID is : {child_PID}`.
After the child exits, the parent prints
`Parent: The child has terminated.`

File Descriptors and Fork

A file descriptor (a.k.a `fd`) is an abstract indicator (handle) used to access a file or other input/output resources, such as a pipe, network socket, disk, terminal etc. A file descriptor is a non-negative integer, generally represented in the C programming language as the type integer . When a fork operation occurs, the file descriptor table is copied for the child process, which allows the child process access to the files opened by the parent process. Note: Only file handles get copied, the offset within a file for read and write are independently manipulated.

- **Task 2: Writing to a file without opening it**
  Write a program `smart_file_write.c` that opens a file (output.txt) and gets a file descriptor, writes "`Hi I am the Parent`" to the file, and then forks a child process.
  The child should then be able to write to the file without re-opening the file using the copied file descriptor. The child should write "`Hi I am the Child!`".
  What do you expect the final contents in the file?

Fork and Exec

The `exec` system call is used to run an executable in a process by overwriting the content of the existing process with the content of the executable. Typically, in Unix systems, as `fork` is the primary way of creating processes, a `fork+exec` combination is used to spawn new programs. In this technique, to run an executable, the parent process forks a new process and the child process uses `exec` to load and run the executable.
Before we use `exec`, let's learn how to use some I/O system calls.

- **Task 3: Writing your own "head" program**
  Write a program `my_head.c` which takes two arguments, an integer n and a file name. It should read the given file and print the first n characters in the file to standard output. For this, you should not use any library functions like `printf`, `scanf cin` for I/O. Use standard system calls `read()` and `write()`.
  The integer given as argument to this is available as a character array in the program. You can use the `atoi()` method to convert it to integer.
  Assume that n is a positive integer and is less than the number of characters in the file.

  Sample run: `./my_head 10 sample.txt`
  Note: Compile `my_head.c` and name the executable as `my_head`. You need to use it in the following tasks.

- **Task 4: fork+exec**
  Write a program `multi_head.c` which takes a list of file names as arguments. It should print the first 10 characters of each of the files in the given order using the `my_head` program written above. For each given file, you can fork a new process and `exec` the `my_head` program with relevant arguments. The parent waits till child exits and then continues to the next file.

  Sample run: `./multi_head abc.txt def.txt ghi.txt jkl.txt`

- **Task 5: Output redirection**
  Write a program `write_head.c` which takes the arguments similar to `my_head.c` in Task 3 and one additional output file name. It should use the `my_head` executable from Task 3 and print the first n characters to the given output file in the argument. This can be done by forking a new process, closing child's standard output, opening the output file and executing `my_head` with required arguments. Make sure you give atleast read permissions when you try to create a non-existent file using `open()`.
  Whenever a new file is opened, it is allocated the smallest `fd` which is free. In this case, as the file descriptor of stdout is closed, the output file is given stdout's `fd` when it is opened.

  Sample run: `./write_head 10 input.txt output.txt`

Signal Handling

A signal is an asynchronous notification sent to a process in order to notify it of an event that occurred. When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal. A signal is mechanism to inject explicit non-deterministic software interrupts targeted for a process. For example, pressing `Ctrl+C` during the execution of any process sends a signal called the SIGNINT signal to the process.
Also, for each process, a custom 'handler' of a signal can be used. The custom handler can be then set as the signal handler by using the **signal** system call.

- **Task 6: Implementing a Safe Version of Ctrl+C**
  Write a program *safe_ control_ c.c.* This program should override the current SIGINT signal handler to ask if the user really wanted to send the SIGINT signal to the running process every time the user presses Ctrl+C. The handler should then ask for either yes or no and then exit or not accordingly.
  The program should go to an infinite loop and in between you should press Ctrl+C to check the correctness of the new signal handler.

Submission Guidelines

- All submissions via moodle. Name your submissions as: **<rollno_lab2>.tar.gz**
- The tar should contain the following files in the following directory structure:
```
<roll_number_lab2>/
|__task1/
|____simple_fork.c
|__task2/
|____smart_file_write.c
|__task3/
|____my_head.c
|__task4/
|____multi_head.c
|__task5/
|____write_head.c
|__task6/
|____safe_control_c.c
```
- **Deadline: 29th July 11.59 A.M.**
  Expected time for completion of lab: 3 hours