CS 333: Operating Systems Lab
Autumn 2018

Lab 2: The matter of processes

Goal

The goal of this lab is to use system calls related to process creation and usage. After completing this lab you will find yourself closer to building multi-process programs.

Before you begin

You must use 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 and other process ID related calls (get parend pid etc.).

The default first three file descriptors, which are opened and available for every process, are:
0 : Standard input (stdin)
1 : Standard output (stdout)
2 : Standard error (stderr)
While `stdin`, `stdout` and `stderr` are declared as `struct FILE*` in `stdio.h`, the system calls for read, write refer to the file descriptors by an integer index in the per process file descriptor table.

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 `p1.c` that forks a child and prints the following (in the parent and child process),
  `Parent : My process ID is: 12345`
  `Parent : The child process ID is: 15644`

  and the child process prints
  `Child : My process ID is: 15644`
  `Child : The parent process ID is: 12345.`

- **Task 2:** Write another program `p2.c` that does exactly same as in previous exercise, but the parent process prints it's messages only after the child process has printed its messages and exited. Parent process **waits** for the child process to exit, and then prints its messages and a child exit message,
  `Parent : The child with process ID 12345 has terminated.`

File Descriptors, Fork, and Exec

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 pipes, network sockets, disks, terminals etc. A file descriptor is is referenced as a non-negative integer index in a descriptor table. 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 descriptor entries are copied, system wide file information along with current offsets etc. are common and can be manipulated by each process.
The `exec` system call is used to load an executable in a process by overwriting the content of the existing process with contents 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` or it's variants to load and run the executable.

- **Task 3:** A program `mycat.c`, available as part of this lab, reads input from `stdin` and writes output to `stdout`. Write a program `p3.c` that executes the binary program `mycat` (compiled from `mycat.c`) as a child proces of `p3`.

  Hint: The program forks a child process, the child process executes `mycat` binary, and the parent process waits for the child process to exit.

  Also, check what happens when input is redirected to `p3`.

- **Task 4a: Writing to a file without opening it**
  Write a program `p4a.c` which takes a file name as command line argument. Parent opens file and forks a child process. Both processes write to the file, "`hello world! I am the parent`" and "`hello world! I am the child`". Verify that the child can write to the file without opening it. The parent process should wait for the child process to exit, and it should display and child exit message. Refer sample outputs.

- **Task 4b: Input file re-direction magic**
  Write a program `p4b.c` which takes a file name as a command line argument. The program should print content of the file to stdout from a child process. The child process should execute the `mycat` program.

  Cannot use any library functions like `printf`, `scanf`, `cin`, `read`, `write` in the parent of child process.

  Hint: `close` of a file results in it's descriptor to be reused on a subsequent `open`.
  Note: Do not make any modifications in `mycat.c`

Orphans and Zombies

A running process becomes an *orphan* when its parent has finished the execution or terminated. In a Unix-like operating system any orphaned process will be adopted by a special process, the `init` process. `init` is the first user-level process. A process that has completed its execution or terminated but still has

some state (pid, memory allocation, stack, etc.) in the memory is called a `zombie` process. Operating systems cleans up such zombie processes when the parent process executes the `wait` system call or when the parent process exits.

- **Task 5: Orphan**
  Write a program `p5.c` to demonstrate the state of process as an orphan. The program forks a process, and the parent process prints the the following messages.
  `Parent : My process ID is: 12345`
  `Parent : The child process ID is: 15644`
  The child process prints message
  `Child : My process ID is: 15644`
  `Child : The parent process ID is: 12345`
  sleeps for few seconds, and it prints the same messages one more time. By the time the child process wakes from sleep, the parent process should have exited.

- **Task 6: Zombie**
  Write a program `p6.c` to demonstrate the presence of zombie processes. The programs forks a process, and the parent process prints the message
  `Parent : My process ID is: 12345`
  `Parent : The child process ID is: 15644`
  and the child process prints the message
  `Child : My process ID is: 15644`
  `Child : The parent process ID is: 12345`
  After printing the messages, the parent process sleeps for 1 minute, and then waits for the child process to exit. The child process process waits for some keyboard input from user after displaying the messages, and then exits.

  Display the process state of child process while it was waiting for input and after the input using `ps` command.

  `ps -o pid,stat --pid <child's PID>`.
  Refer `man ps` for the details of different process states. Reason about the output.

- **Task 7.a: Recursive fork**
  Write a program `p7a.c` that takes a number `n` as a command line argument and creates `n` child processes recursively, i.e., parent process creates first child, the first child creates second child, and so on. The child processes should exit in the reverse order of the creation, i.e., the inner most child exits first, then second inner most, and so on. Print the order of creation of the child process, and their termination order as shown in sample output.

- **Task 7.b: Sequential fork**
  Write a program `p7b.c` that takes a number `n` as command line argument and creates `n` child processes sequentially, i.e., the first parent process (`p7b`) creates all children in a loop without any delays. Let the child processes sleep for a small random duration (use the `urand_r()` call, and print the creation and exit order of the child processes. Note that the random numbers used for sleep should be different across the child processes.

- **Task 7.c:**
  Write another program `p7c.c`, that creates n child processes similar to sequential creation of p7b.c. Further, each child also does the similar sleep action for a random duration.

  In the parent process, set it up so that child termination is in the reverse order of sequential creation.

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>/
  |__task1-2/
  |____p1.c
  |____p2.c
  |__task3-4/
  |____p3.c
  |____p4a.c
  |____p4b.c
  |__task5-6/
  |____p5.c
  |____p6.c
  |____p6-explanation.txt
  |__task7/
  |____p7a.c
  |____p7b.c
  |____p7c.c

- **Deadline: 23rd July 11.59 PM.**
  Expected time for completion of lab: 3 hours