Chullipparambil, Chinnu Padman

Class Account: masc1445

CS 570, Summer 2014

Assignment #3, Design and implement a multiple process, time viewer.

Filename: README

----------------------------------------------------------------------

Files Included:

Source File: a3.cpp

Include File: a3.h

Makefile

README.docx

----------------------------------------------------------------------

Compile Instruction:

g++ -o a3 a3.cpp

No bugs found.

Covers all requirement specifications.

Pipe communication implemented for extra credit.

Error handling done for all system calls used.

----------------------------------------------------------------------

**Flow Chart:**

Main Process

Clock Process

fork()

Timer up. Send terminate signal.

Timer Process

fork()

Uptime Process

Timer up. Send terminate signal. Exit Timer.

Uptime Caller Process

fork() fork() & exec()every 5 sec uptime process ends. **Design Details:**

* Main process creates three child processes one by one.
* First child prints hour, minute and second once every second.
* Second child creates new child process every five seconds which in turn call uptime program. Uptime process gives a one line display of the following information: the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes. This is short lived process and which exits itself after printing this information.
* Third child implements a countdown timer. When time is up, it sends termination signal to the first and second child processes and terminates itself.
* Main process waits for each of its child processes to complete using waitpid() and then exits.

Note: Above flow chart does not show the pipe usage used for logging (for extra credit).

**Pipe Implementation:**

* Main process created three pipes, one for each child to communicate with its parent.
* In first and third child processes, instead of writing to stdout, used write() system call to write to the write end of the corresponding pipe.
* In second process, since the stdout message is coming directly from uptime program(not ours), used close(1) and dup() system calls to map the write end of pipe to stdout.
* In main process, used select() system call to monitor all the pipes until one or more become ready and correspondingly read from the pipe and print the message to stdout.
* In order to inform the main process that timer is up so that it can exit from the loop, made the third child to send a user signal to the main process using signal handler.

**Lessons Learned:**

1. Learned the usage of fork() to create multiple child processes and handle the parent and child code depending on the values returned from fork(). Implemented parent code for return value > 0(pid of the child is returned in the parent) and child code for return value = 0 (0 is returned in the child).
2. Since all processes are running concurrently and we have not designed them to run in any particular order, we cannot predict the order in which the output is displayed. We can only verify if first and third are printed once every sec and uptime is printed once every 5 sec.
3. To make the second child to run the uptime program, it was not possible to call exec() directly inside child 2, since it will replace that child process with the new process, which means the child 2 process will become uptime process. Further code written inside child 2 did not get executed. Hence implemented it by calling fork() inside child 2 and the new child would call exec. These steps are iterated inside a while loop every 5 sec.
4. To notify the sibling processes to terminate when the timer reached zero, used kill command. The command kill sends the specified signal to the specified process. Used the pids of the first and second child in the command and used SIGTERM as signal, so that it will terminate those processes which are identified by the pids mentioned.
5. In order to make the main process wait for all child processes to terminate, used waitpid command passing pids of each child process. This ensured that the main did wait till all child processes completed and printed the last statement in main which is a stdout message. Also used waitpid inside the child 2, while it was the parent for new child processes created for uptime call, to ensure the parent waited for the uptime process termination and then exited.
6. It was required that the third child process have the pids’ of first and second child processes to send signal for termination when the timer is up. For this, pid received after fork() call for child creation are stored in global variables of the main. Hence when the third child is created, it gets the exact copy of global variables of its parent and can thus use those pids to communicate with its siblings.
7. Initially I did not add exit(0) after exec call. Later realized it was required as I observed when the exec() was not successful (it happened because of wrong arguments) and there was no exit(), it was executing the parent code since the process image was not replaced. Hence added exit(0) after exec() to make sure it will exit successfully and will not enter the while loop in parent and create undesired outputs in case of exec() failure.
8. To find the proper argument list for exec, it took a bit more time. Had to go through variants for exec() system call and figure out the argument list properly. It was a good learning.

**Learnings from pipe implementation:**

1. Learned how to use pipe()and how to do unidirectional data transfer for inter process communication.
2. In second child process, as we have to redirect the output from an external program(uptime) to the main process, used below mechanism. Used close(1) and dup() system calls to map the write end of pipe to stdout. dup() uses the lowest-numbered unused descriptor for the new descriptor. When we call close(1), it releases file descriptor 1. Hence we make sure that the immediate dup() will be mapped to file descriptor 1 and thereby to stdout.
3. Used select() system call to handle multiple pipes in the main process concurrently. select() allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become ‘ready’. Then used fd\_isset to check for readiness for each pipe. Then correspondingly read from pipe and print message to stdout. With this implementation, read does not become a blocking call(no wait).
4. Understood that I have to reinitialize the descriptor sets before every select() system call to ensure proper functioning. Because select() modifies the descriptor sets.
5. Learned how to register signal handler with signal() system call and also using signal\_handler for a user signal. Used this for the third process to send a user signal to the main process to inform that time is up so that it can exit from the loop used for monitoring pipes.
6. Used memset() to reset the buffer used for reading from pipe. This ensured that even if the data comes in is without ‘\0’, still the data can be dealt as string.