Operating Systems & Networks, 2008
0440949 Andreas van Cranenburgh
Mon Nov 17 17:02:02 CET 2008

Abstract

This paper describes an exercise demonstrating the use of `fork()` and `pipe()` calls. The main process listens for keystrokes[1], some of which will spawn children which also receive the same keystrokes. The children die when they receive 'poison,' for which they can collect anntidote which makes them temporarily immune. The cleaning up of finished child processes is handled through a signal handler trapping the SIGCHLD signal.

1 Forks and pipes

1.1 Forking

In order for a program to do more than one thing at the same time, or asynchronously, it might spawn multiple versions of itself. The traditional way to do this is using the `fork()` system call. When this call is issued an exact copy of the current process is made, with even its program counter at the same spot as the original process (ie., just after the `fork()` call). Using the return value of `fork()` it can be determined if the current process is the parent or the child process, so that the appropriate code path can be followed.

1.2 Pipes

If a process wants to communicate with its children the easiest way is to set up a pipe. This pipe is created just before forking. Using the return value of `fork()` the parent can close the child's end of the pipe, and vice versa for the parent's end in the child. This pipe can be read and written to like any other file.

1.3 Zombies

When a child process finishes it does not disappear completely. The process will remain in the process table, changing its state to 'zombie.' This means that the only thing left is the exit status of the process, which the parent process might want to know about. This is done using the `wait()` call in the parent process.

A simple minded approach would be to issue a `wait()` whenever 'poison' is being sent. However, this has the unfortunate side-effect that the master process will be waiting indefinitely if 'poison' is being sent while there are no child processes. This can be avoided by checking the number of child processes and only waiting if there is at least one.

Even then these child processes might have received a sufficient amount of 'antidote,' so it might still be a mistake to wait for a process to die. Issuing a `wait()` means the process blocks itself until it receives a signal, which means the program will fail to respond to other events.

[1] keystroke is meant here as the input of one character followed by a return, which is necessary because STDIN is normally buffered, unless redirected or put into raw mode

1.4 Don't fear the reaper

The real solution is to install a *signal handler*. At the start of the main process this signal handler is installed to trap specific signals with our own function. The signal of interest here is SIGCHLD: the signal that will be sent to the parent process if one of its children has finished. Our call to `wait()` will be issued in this signal handler and all of the lost children will be taken care of.

1.5 `fgetc()`, interrupted

Just as with most system calls, `fgetc()` does not like to be interrupted. It will return an error and that means the child processes will receive an EOF through its pipe when in fact an error has occured. The child process will then finish because it thinks its pipe will not get any more data.

Instead of ignoring such cases in the child process I have chosen to temporarily block child signals during the execution of `fgetc()`. This works well, although on Linux the result is that after sending 'poison' which kills a child process it will only be waited for after the next call to `fgetc()` finishes, since `fgetc()` patiently waits for a keystroke while signals have been disabled. On OpenBSD the signal arrives immediately.

1.6 Log files

The program uses three log files. The first two are shared among all child processes, one of which is buffered, the other unbuffered. These are opened by the master process and handed over to the child processes. The third log file is opened by the child processes. When there is more than one child process, the last one to open the third log file gets to write it (the output of the other child processes simply disappears).

The difference between buffered and unbuffered is that the contents of the buffered logfile are written in bursts, so that chunks of lines from one child are together. The unbuffered logfile is written immediately, so that the output of the various children is interleaved directly, in exact chronological order. Eventually all information is written to both log files so no information is missing.

1.7 A few experiments

The following keystrokes are used:

f: fork a new child

P: send poison

A: send antidote which protects once against poison

q: quit (both for child and master process)

Some experiments and associated behavior:

`f a b f c d P e g P x x q`**:** Two children are spawned and then poisoned.

`P P P P f f f f f x P q`**:** Five children are spawned and then poisoned, apparantly in random order (as shown by repeated experiments)

`A A A A P P P P f f f f f x P q`**:** Five children are spawned, two of which are poisoned. The others receive enough antidote to survive (again in unpredictable order).

`f f f a b c d e g h i j k l q:` Three children are spawned and killed on exit.

`a b c d e f g h i j k P q:` One child is spawned and killed on exit.

`a b c d e f g h i j k P:` idem.

1.8 References

Code for signal handling was adapted from:
`http://www.aquaphoenix.com/ref/gnu_c_library/`