Programming Assignment 3: Synchronization and Mailbox

Due May 20, 2024

The Basics

In this assignment you will implement a producer – consumer paradigm, readers-writers, using shared memory. First a parent process should be launched. Then it should spawn 2 children processes. Each child process can communicate with the parent and visa-versa.

There are several related goals for this assignment:

- Implement the Spawn(), Waitpid(), and Exit() system calls. These calls will allow user processes to create new user processes. Spawn() will involve loading a user program from a file and then running it.
- Do some basic keyboard I/O, implementing Putchar() and Getchar() for user programs.
- Write a simple shell for Minix with support for Spawn() and Waitpid().
- Defined a shared memory, i.e. mailbox, from the parent to the children. The memory must be shared among all children as you need to implement readers-writers synchronization paradigm.
- Write a program and the necessary traps to display a list of all running processes (spawned from the shell, and otherwise). This list should include the program name, ID, mode (user/system) and state (running, ready, waiting, etc.) for each process in the system. In other words, this program would do the same thing as ps does on Unix. You'll need at least one new trap to implement this, and I'd suggest two traps (one to get the list of process IDs and one to find information about each process). Keep in mind that returning strings from a trap could be difficult.

To do this, you'll need to understand traps, shared memory, process creation and deletion, basic file I/O, and keyboard I/O. You'll also have to do some work on synchronizing user processes using the Locks and Condition variables.

The Details

Spawn() and Waitpid()

You'll need to implement two system calls, Spawn() and Waitpid(). Both calls deal with user processes, a necessity for a timeshared system. Spawn() takes two arguments — the name of the program to run in the new process and a single integer to be passed to the new process (as is done in Unix with argc and argv[]). Spawn() returns an positive integer (the process ID) that can be used to identify the process in future calls to the operating system. For now, assume that the memory allocated to a process by ProcessFork() is sufficient to run it.

- ProcessFork() is a wrapper function around fork()

Spawn() has to load user programs into memory. ProcessFork() already does this, and can be called from a trap. Spawn() must return a negative number if the call fails—the file isn't found, there aren't any process slots left, or any other errors such as running out of memory. Each error

should return a different number so a user program can differentiate the causes of the system call failure. You may need to modify ProcessFork() to return a negative number (rather than exiting) for some of these error conditions.

Waitpid() takes a single argument—a process ID returned from an earlier Spawn() call . The process calling Waitpid() is suspended until the process identified by the argument to Waitpid() completes. When it completes, the process that called Waitpid() is resumed. As with Spawn(), it's possible for Waitpid() to fail for reasons such as non-existence of the process whose ID is passed, in which case a negative number should be returned to indicate an error. Hint: you'll probably want to use locks and condition variables to implement Waitpid()....

Exit() takes a single argument—a process exit status that it returns to the parent process (in response to a Waitpid() call).

Input & output

Your operating system should support simple I/O routines—the ability to read & write a single character at a time. This functionality is provided to user programs via traps, but the OS has to do some work to make the traps work properly. Keep in mind that characters may come in before the user program has actually asked for them, so you may need to use a buffer to communicate between the keyboard trap handler and the trap handler that deals with requests to get a character. Also, your user programs may not use printf(); they may only use Putchar(). You may want to write the code for puts() to write out strings; this code should call Putchar().

Disk input and output must use the file system calls provided in Minix; see the sample code in **process.c** for examples of how to open, read, and close a file.

User shell

Your user shell should allow a user to type in commands and have them execute (you can modify the shell from assignment 1). The commands are actually Minix programs you've compiled separately, and are loaded in and run when the command is typed. There are two basic ways to run a program: foreground and background. To run a program in the foreground, type

program 1234

This will fork off a new process running "program", and pass the numeric argument 1234 to it. The shell will then wait for program to finish using the Waitpid() system call. Programs run in the background are called like this:

program 5678 &

This will run program in the background, allowing it to run at the same time as the shell (and perhaps other background processes). Immediately after forking off this process, the shell should return and allow the user to enter more commands.

Your shell should buffer up characters that are typed in, taking action only when a <RETURN> is pressed. You should correctly implement backspace (<CONTROL-H>), which should delete the last character from the buffer. To implement backspace on the display, output the character sequence <BACKSPACE><SPACE><BACKSPACE>. For information about which characters correspond to which, you can test keyboard input using the default version of the OS (it prints out the keys it receives).

The shell will use the Spawn() and Waitpid() traps, of course. It'll also need to use traps to get and put characters— Getchar() and Putchar(). You'll need to write these traps as well, though the OS code for both of them is partially done already.

For consistency's sake, please write the code for your user shell in shell.c.

Traps

There are already some basic traps provided for you, such as Open. You should look at these traps as examples for the traps you'll need to implement. If you create code for new traps, you should put them into a separate file called usertraps.s, and compile this file along with every user program you write.

The full list of system calls is located at: minix/include/minix/callnr.h

Note: There is no method to directly send data from the user to kernel, except via servers.

- The TRAP goes into the kernel.
- The caller's stack and register have parameters indicating which function to call in which server.
- Each server has its own set of system calls
    - File system server provides system calls for accessing files
    - Process manager manages processes
- Some system calls map the call to another kernel call
    - SYS_FORK (System call in servers, kernel call in kernel/system)

Open Trap (servers): /minix/servers/vfs/open.c

Open Trap (kernel):

Putchar: /minix/src/lib/stdio.c

Remember that every trap (user or system) must be caught explicitly by the operating system. When the OS catches a trap, it should make sure the arguments are valid and not assume anything. Parameters to traps are passed in the user's address space, but need to be accessed by the operating system in kernel space. See the code for the Open and Putchar for examples of how to copy arguments from user space to kernel space.

You'll need at least the following traps to implement your shell

- char Getchar(): return a character from standard input. The process should wait until a character is ready.
- void Putchar(char c): write a character to standard output. This trap has already been implemented for you.
- int Spawn(char *prog, int arg): create and run a new process with argument arg and return its process ID (i.e., some way of identifying it later).
- int Waitpid(int pid): wait for the process whose process ID is passed.

Hint: Since we are sending characters here, you can consider this design as a pipe. 1 Write, N readers, this implies the writer broadcasts to all readers.

Testing

- Implement the N readers, 1 writer producer-consumer scheme.
    - Writer can only change the mailbox after each reader is no longer accessing it.
    - You can prevent the writer from accessing the mailbox by using an integer count lock
- To test the mailbox, you parent and child processes should track their process IDs.
- Each process on launch should identify itself via the tty “Process <?> launching”.
- Launch 2 children processes.
- Parent process should prompt for a number. Parent should copy number to mailbox (should be empty), then set flag(s) to notify the children processes.
- Each child process should print the received number and their process ID “Process <N> received <x>”
- Parent should prompt for a new number when all child has finished reading the mailbox.

Design hints

The shell is a user program, and isn't compiled into the operating system. As such, you shouldn't compile it with the rest of the OS files. Instead, compile it separately and load it in. Since the basic operating system comes with the ability to execute a user-level process (via the -u option), you might want to experiment with them first and use that information to design your shell.

The operating system learns of incoming keystrokes (characters) by being interrupted. When it gets a keyboard interrupt, the interrupt handler should place the incoming character into an array. When it's called, Getchar() removes a character from this array if one is available. If none is ready, it should wait (using semaphores or locks & condition variables) until one becomes available. Note that this is a variation on the producer-consumer problem we studied in class. There's only one difference—it would be impossible for the producer (the interrupt handler for the keyboard) to wait until the array became non-full, so throw away characters if there's no space in the array for them.

There are several miscellaneous functions in misc.c and misc.h that implement functions from the standard C library. Feel free to use these functions in your user programs (or OS) and (if you

like) add more functions. In particular, you may find the dstrtol() function (which converts strings to integers) and string functions useful for your shell.

Big Hint: Look at the function mmap() and think about the order of the system calls necessary to achieve the memory mailbox.

Design document items:

1) Be sure to address the following questions:
    a. Which files did you change?
    b. Which functions did you use?
    c. Why?
2) Explain the flow of a message from one process to another, i.e. explain step-by-step via a flow diagram how the mailbox memory you used enables a message lets one process communicate with another.
3) Explaim which traps were used in your design. Why?
4) Was the mailbox unidirectional or bi-directional?
5) What challenges did you face when implementing the memory mailbox?
6) How did you address memory mailbox schronization? **Describe in detail.**
7) How did you prevent the same message being read by a single process multiple times? **Describe in detail.**

What to turn in

A compressed tar file of your project directory, including your design document. You must do "make clean" before creating the tar file. In addition, include a README file to explain anything unusual to the TA — testing procedures, etc. Your code and other associated files must be in a single directory so they'll build properly in the submit directory.

REMEMBER: *Do not* submit object files, assembler files, or executables. Only submit source files and the design document(s).

Useful references:

1) System and Kernel Calls

https://lass.cs.umass.edu/~shenoy/courses/spring20/lectures/Lec06.pdf

2) How to Add a New System Call for Minix 3

https://web.ecs.syr.edu/~wedu/seed/Labs/Documentation/Minix3/How_to_add_system_call.pdf