Operating Systems: Report 1

Joseph Jones (s2990652), Charles Randolph (s2897318),
Barnabas Busa (s2922673)
Group 04

March 9, 2018

Exercise 1: iwish

Introduction

The first exercise of this assignments asks that we implement a basic shell. The shell requires the following functionality be available:

- **Search-Path Support**: The shell should be able to search through the user's PATH variable and check the various directories for locations of a given program.
- **I/O Redirection**: The shell should be able to support explicit redirection (E.G: `ls -a > out.out`).
- **Background Processes**: The shell should be able to run commands in the background (E.G: `./sleep &`).
- **Pipes**: The shell should support inter-process piping (E.G: `echo Hello | cat | cat | cat > out.out`).

Design

Processing Input

In order to process input from the terminal, UNIX tools Bison and Flex were used. The IEEE's Shell Command Language was used as a starting point. We stripped their supplied grammar of inapplicable rules, and wrote a lex file to supply it with the necessary tokens. The grammar is available in the `iwish.y` Bison file. In order to properly stage programs for execution, we used a queue (see `queue.c` and `queue.h`) to collect all tokens that belong to a complete command (a sequence of piped programs finishing in an optional ampersand or semicolon). The queue is then processed by our evaluation function (`evalQueue`) each time a complete command is entered.

Command Evaluation and Background Processes

Evaluating a complete command necessitates parsing the token-queue. The starting routine `evalQueue` handles this task. It runs the `evalPipeSequence` routine on the queue, and receives in return the very last PID for the process launched in the pipe-sequence. If the queue contains another token, and it's an ampersand, then the shell forks itself and processes the parsed program before exiting. If there was no trailing symbol, or it was a semicolon, `evalQueue` simply waits for the last program to finish before returning control to the parser.

Pipe-Sequence Evaluation

Processing a pipe-sequence is the task of `evalPipeSequence`. The function begins by initializing a local variable `fd_in` to `stdin`, the standard input. It then uses the `parseProgram` routine to read from the queue and construct a program object. `parseProgram` sets the program name, arguments, and redirections in a convenient data structure. After parsing the first program, `evalPipeSequence` checks whether a pipe succeeds the program. If one does, it invokes `setPipe` to both create a pipe and redirect output from the program to the writing descriptor. `setPipe` also lauches a program given to it, since input is defined by `fd_in` and it will be outputting to the pipe's writing descriptor. `setPipe` then returns the reading-descriptor of the pipe, which is then set to `fd_in` as the future input for whatever program follows. Finally, the last program is directly executed by `evalPipeSequence` and given `stdout` as it's output descriptor.

Program Execution

When a program is to be executed, `evalProgram` is invoked. The function takes the input and output file-descriptors for the program, and the program object initialized by `parseProgram`. It forks to create the child process, and finally calls `execProgram` to apply any explicit redirections, replace stdin and stdout with the given file-descriptors, and attempt to `exec` on all paths until one is succeeds or a failure is printed.

Usage

The shell may be compiled by using `make`, and then executing `./iwish`. The shell prompts the user with `iwish$` when it is ready to accept commands.

Exercise 2: Simulating Shared Memory

Introduction

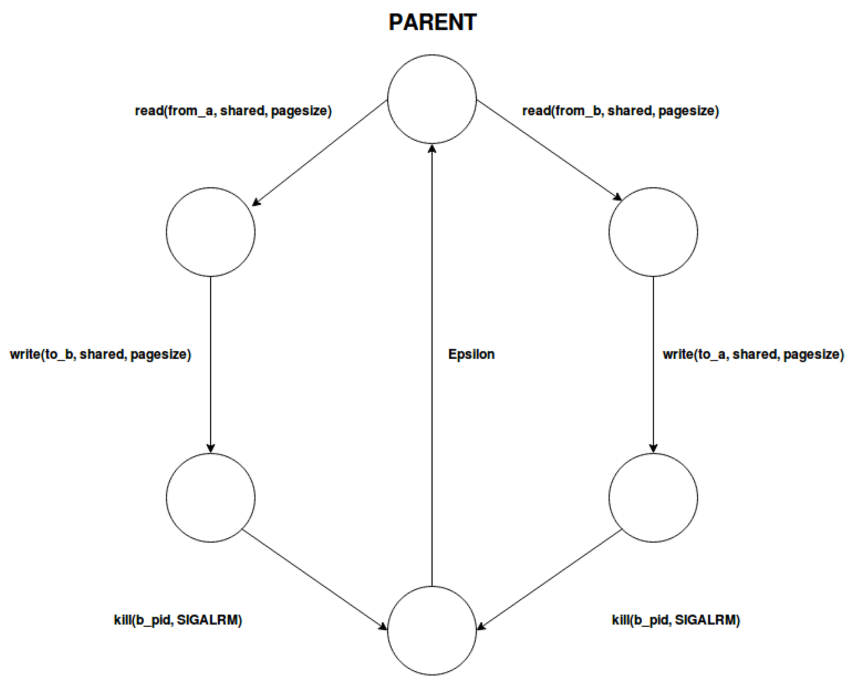
The simulated shared memory exercise requires that we construct a set of three programs; one parent, and two children. The parent program should play the role of a memory manager, synchronizing the state of a shared chunk of memory between two child processes. The child processes should run an interactive ping-pong game between each other, unaware that the variable they are accessing is being managed.

Design

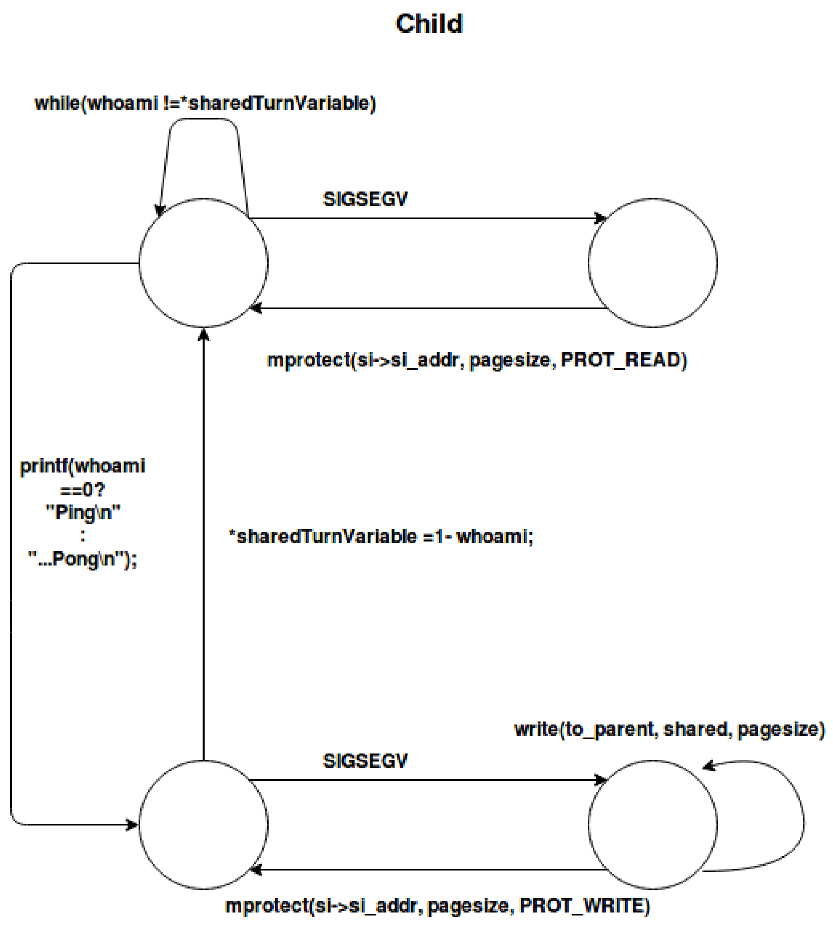
When a process forks, the child process is a copy in almost every aspect, including memory on the heap. This immediately posed a challenge because signal-handlers and protections configured within the parent process prior to forking would not be forwarded to that process once triggered in the child. In order to simulate shared memory, we decided to use bi-directional pipes between the parent and children to send data, and use `mprotect` and `kill` to raise signals in processes for the purposes of performing data synchronization tasks. A short summary of how protections and signals are used is as follows:

- `mprotect`: The system `mprotect` function is used to enable the child processes to send data to the parent process. When the children first launch, they have no read or write permissions on their memory. When they first attempt to read, the signal handler grants reading permissions, but only reading persmissions. Therefore, once the child tries to write to the shared memory chunk, the trap occurs once more. However this time, we use the opportunity to send data though a pipe to the parent. The writing (ONLY) permission is then granted to the child. Once they try to read later, the signal handler for the read call resets the writing permissions so that the same synchronization scheme can be fired next time it writes. Hence, `mprotect` allows us to send data to the parent periodically when any child writes to the memory.
- `kill`: Sending data to a parent process in a `mprotect` related handler is all well and good, but perfectly useless if the parent can't somehow interrupt the other child to get them to update their shared variable. This is where `kill` is used. Upon reading anything from the incoming pipes to it's children, the parent process copies the read page to the other child's writing pipe, and then dispatches a `kill(child_pid, SIGALRM)` call to the child that needs to be notified. Once it receives this signal, the child's handler for `SIGALRM` fires, and it uses the opportunity to update it's shared varible.

This entire scheme was devised with the sole intention of not having to modify the source code within the `procpingpong` routine, but we had to make once exception with regard to the `mprotect` call. Because setting writing permissions implicity set reading permissions, we couldn't force the signal to be triggered next time a child tried to read from a shared variable after writing to it. This was very important because we require that write-permissions be removed at this location in order for them to fire the handler later on. We solved this by simply adding the mprotect line explicitly to the `procpingpong` program.


PARENT
read(from_a, shared, pagesize)
read(from_b, shared, pagesize)
write(to_b, shared, pagesize)
Epsilon
write(to_a, shared, pagesize)
kill(b_pid, SIGALRM)
kill(b_pid, SIGALRM)

The following diagram illustrates the flow of the parent process. It simply waits for data to appear on it's pipes, and then forwards it to the opposite recipient.


Child
while(whoami !=*sharedTurnVariable)
SIGSEGV
mprotect(si->si_addr, pagesize, PROT_READ)
printf(whoami ==0? "Ping\n" : "...Pong\n");
*sharedTurnVariable =1- whoami;
write(to_parent, shared, pagesize)
SIGSEGV
mprotect(si->si_addr, pagesize, PROT_WRITE)

This diagram demonstrates the flow of the child process. What isn't shown is the signal handler for `SIGARLM` though. This is omitted because it is not a routine decision made by the child process.

Usage

Our solution program may be built with make, and executed as `./pingpong`. However, we must state that it isn't **consistently reliable**, and on occasion tends to just block after the first ”ping”. This has been addressed by introducing a small delay in the `procpingpong` process which somehow seems to avoid the blocking problem.