Homework 3
(Shared Memory Portion)
CSC 456

Due on Sunday, May 1, 2015

Dr. Karlsson

Lisa Woody

Overview

Documentation for shared memory portion of Homework 3, as well as the supporting documentation from Homeworks 1 and 2

Purpose and Description

The purpose of the this portion of Homework 3 is to implement inter-process communication with shared memory in the shell developed over homeworks 1 and 2. The shell will setup the blocks of shared memory and the required semaphores to control access to the shared memory.

The shell is a command line program that presents a prompt with a prompt string: *dash* > and accepts commands to be executed at the prompt.

The following commands were implemented in program 3:

- **mboxinit** <***number of mailboxes***> <***size of mailbox in kbytes***> - Allocates shared memory mailboxes
- **mboxdel** - Cleans up semaphores and shared memory
- **mboxwrite** <***mailbox number***> - Write data to a shared memory mailbox
- **mboxread** <***mailbox number***> <***pid***> - Read data from a shared memory mailbox
- **mboxcopy** <***first mailbox number***> <***second mailbox number***> - Copy the contents of the first mailbox to the second

The following commands were implemented in program 2:

- <**command + arguments**> - Run non-intrinsic command in a new process
- **cd** <***directory***> - Changes the working directory of the process
- **signal** <***sig_num***> <***pid***> - Sends a signal to a process
- <***cmd1***> | <***cmd2***> - Pipes output from cmd1 as input into cmd2
- <***command + arguments***> > <***file***> - Redirects output from cmd into file
- <***command + arguments***> < <***file***> - Uses file as input for command

The following commands were implemented in program 1:

- **cmdnm** <***process id***> - Displays the command string (name) that started the process for a given process ID
- **pid** <***command string***> - Displays all process IDs whose command strings contain the given string.
- **systat** - Displays process information: Linux system version, system uptime, memory usage information, and CPU information.
- **exit** - exits the program

Implementation

The program is implemented using a loop that prints out *dash* > to the screen and reads in a line of user input from the iostream. The line of user input is tokenized and the individual words placed in a vector. The first word is then matched to a command function or, if no match exists, returns a command not found error to the user. Use of the < *enter* > key simply reruns the loop and places the curser on a new *dash* > input line. A string that matches an existing command sends the command vector to a function that carries out that command on the rest of the user input.

Each command is implemented as follows:

- **mboxinit <*number of mailboxes*> <*size of mailbox in kbytes*>**
  One contiguous block of memory was allocated to hold all of the mailboxes. A header block was created at the beginning of the memory to store the number of mailboxes, the size of each box, and reader/writer lock structures for each mailbox.

- **mboxdel**
  After acquiring the shared memory id using the shared memory key, the program checks if the shared memory block has been initialized. After this verification is complete, the mailboxes are removed and the memory released with the following call: shmctl(*shared memory id*, IPC_RMID, 0)

- **mboxwrite <*mailbox number*>** and **mboxread <*mailbox number*> <*pid*>**
  A memory block is copied from the user input into the shared memory or vice versa. To implement data synchronization, a reader/writer lock was constructed, allowing multiple processes to read data concurrently but only one process to write at a given time. Semaphores were used to implement the locks.

- **mboxcopy <*first mailbox number*> <*second mailbox number*>**
  Using the semaphored first box mailbox is locked for reading, while the second mailbox is locked for writing. The block of memory is then read from the first box and written to the second using the strcopy command.

- **<command + arguments>**
  The dash shell is able to execute non-intrinsic Linux commands. This is done by calling fork() to start a new process, which calls exec() to execute the commands. A test is done to determine if the command contains redirect or pipe symbols, in which case the execution is handled as described below.

- **cd <*directory*>**
  The working directory of the process is changed by implementing the following call: chdir(*directory*)

- **signal <*sig_num*> <*pid*>**
  The shell calls the function kill(pid, sig_num), which is a system call that sends the given signal (identified by sig_num) to process pid.

- **signal catching**
  Uses the signal() function from signal.h to register a callback function that catches and reports all signals received.

- **Piping**
  Child process started in the fork_exec() function, which handles non-shell-intrinsic commands, forks again to create a grandchild process. The grandchild process used stdout as the input to the pipe,

and implements the commands on the left-hand (output) side of the pipe. The child process uses that output as input for its implementation of the right hand side (input side) commands.

- **Redirection**
  Child process started in the fork_exec() function, which handles non-shell-intrinsic commands, opens the file declared in the user command, creating one if no such file exists. If the command calls for redirection from a commmand to a file, the child process closes stdout, which is then replaced with the file. Any output is written to that file. Otherwise, if the redirection is from a file to the command, the stdin is replaced by the file, and serves as input for the command.

- **cmdnm <*process id*>**
  Attempts to open and reads the first line of the file $/proc/ < pid > /cmdline$. If the file exists, this line, which represents the command that initialized the process, is printed to the screen. Otherwise an error lets the user know that no matching process number was found.

- **pid <*command string*>**
  The dirent library is used to browse each subdirectory in /proc directory whose name is determined to be an number. For each such subdirectory, its name is used to open its matching $/proc/ < pid > /cmdline$ file. The user provided command string is then matched to all substrings of the first line of this file, which represents the command that initialized the process. Since this is done for every active process, all process numbers whose initialization commands contain a substring that matches the user provided command string are returned. These process numbers are printed to the screen.

- **systat**
  The following directories are accessed to display the system information:
  $/proc/version$ - Linux version information.
  $/proc/uptime$ - System uptime in seconds.
  $/proc/meminfo$ - System memory information
  $/proc/cpuinfo$ - CPU information
  Each item is output to the screen.

- **exit**
  Exits the input loop, which subsequently ends the program.

External Resources

- This program relies on read access to the information exported by the kernel to the /proc directory.

- GNU C and POSIX libraries utilized in this program:
  - Process control commands such as fork() and exec()
  - Pipe command
  - Signal processing/handling
  - Directory access

- Some functions were based on the following code:
  - http://www.mcs.sdsmt.edu/ckarlsso/csc456/spring15/code/jchen.c
  - http://www.mcs.sdsmt.edu/ckarlsso/csc456/spring15/code/pipe1.c

Submitted Files

- prog1.pdf - assignment documentation
- Makefile - make file for compiling the included programs
- dash.cpp - main program file
- prog1.cpp - functions specific to assignment 1
- prog1.h - header file specific to assignment 1
- prog2.cpp - functions specific to assignment 2
- prog2.h - header file specific to assignment 2
- prog3.cpp - functions specific to assignment 3
- prog3.h - header file specific to assignment 3

Compilation

1. Unzip the prog3.tgz file
2. Use the terminal to change into the unzipped *prog*3 directory
3. Enter the command $make into the terminal
4. The source code will compile and executables will be located in the *prog*3 directory.

Usage

Usage instructions for the following executable(s):

- $ *./dash* - initialize the program
- *dash >* [*user command*] - type a command into the shell

Testing and Verification

- The the generated results were matched against the output from Linux system tools such as $ *ps aux* and the */proc* directory to confirm their validity. Invalid or missing command names and pid numbers returned errors successfully. Commands were successfully matched to substrings of null-terminated command strings.
- Non-shell-intrinsic command results were tested successfully against the results from running the same command in bash.

Here are the tests I did to make sure the semaphores work correctly:

Test 1:

- Create 10 mailboxes
- Add blocking function (cin) to critical section of write function.
- Write to mailbox 4, allow cin function to block execution of critical section code

- Attempt to read mailbox 4 from another dash process.

Result = Reading process is blocked until writing is complete and the updated message is read.

Test 2:

- Create 10 mailboxes
- Add blocking function (cin) to critical section of read function.
- Dash process 1: Read from mailbox 1 (cin blocks in critical section)
- Dash process 2: Read from mailbox 1 (cin blocks in critical section)
- Dash process 3: Write to mailbox 1

Result = Writing process is blocked until all readers have finished critical section.

Test 3:

- Create 10 mailboxes
- Add blocking function to critical section of reading method
- Write to mailbox 1
- Dash process 1: Read from mailbox 1 (blocking in critical section)
- Dash process 2: Copy from mailbox 1 to 2

Result = Copy function can proceed without waiting for process 1 to finish reading

Test 4:

- Create 10 mailboxes
- Add blocking function to critical section of reading method
- Write to mailbox 1
- Dash process 1: Read from mailbox 2 (blocking in critical section)
- Dash process 2: Copy from mailbox 1 to 2

Result = Copy function must wait for process 1 to finish reading

Test 5:

- Create 10 mailboxes
- Add blocking function to critical section of copy method
- Write to mailbox 1
- Process 1: Copy mailbox 1 to 2 (blocking in critical section)
- Process 2: Read from mailbox 1
- Process 2: Read from mailbox 2

Result = Process 2 can read from mailbox 1 but must wait for process 1 to finish copying before reading from mailbox 2.