Module 10 Assignment
Homework Project 4
System Development in the UNIX Environment (605.614)

Overview

The fourth homework assignment for this class is to create some applications to exercise basic UNIX interprocess communication using pipes (specifically the **popen()** call) and shared memory.

Getting Started

Set up your project hierarchy as you did in the previous Project Assignments. Call the ‘root’ of your project ‘homework4’ in the 614 directory under your home directory (~/614). Be sure to copy your log manager library and the thread_mgr library from the previous projects in ~614/homework3/src/lib and be sure to make repairs to the libraries based on the feedback. Then you will be ready to add your new shmlib library and the applications that will use the library.

Objectives

After completion of this project, you will be able to:

- Create a general-purpose library to create, manage, detach, and destroy shared memory
- Demonstrate ow to use shared memory in a process
- Use pipes (via the popen() library call) to start a new process, and to receive and process its standard output.

Submission

Completion of the assignment involves submitting to the instructor, on or before the due date, the following materials:

- Printed listings of the Makefiles, programs' and libraries' source code are not necessary.
- A comment block at the the beginning of each program and library source file shall include the following information: the student, the course, the name of the file, and a short description of the file's purpose.
- The homework should be compiled and run on the Oracle Solaris11 (UNIX) system (dev4.jhuep.com) and Linux system (absaroka.jhuep.com). The executable programs must be installed in the structure mandated in the last section of this document.

The fundamental rules governing the homework are:

- The programs must be written in the 'C' or the 'C++' programming language.
- No copying or plagiarism. If copying or plagiarism occurs, the person or persons involved will receive no credit. They will be treated as not having submitted the assignment.
- Assignments are due by midnight of the specified due date.
- Extensions are only granted if arrangements are made with the instructor well in advance of the due date of the assignment. No last minute extensions will be given; if a student has not completed the assignment by the due date, the student has the choice of receiving a grade based on the work completed to that point, or receiving a 5%

penalty for each day (or part of a day) it takes to complete the assignment. In these cases, the assignment is not considered complete until the instructor is notified by the student of the completion of the assignment.

To submit your homework, you will run the Perl or Python script developed as part of your first assignment. The output of the script will be used by the grader. You should ensure that the class group has at least read permission on the files. The grader will copy those files to a separate directory for grading. Reminder: 2 binary releases (1 for dev4 and 1 for absaroka) and 1 source release are required.

Grading of the Assignment

The assignment will be graded according to the following criteria:

- The shared memory library must perform the assigned functions.
- All executable programs must work correctly.
- The project hierarchy must be as specified and Makefiles must exist in the appropriate directories and must function properly, supporting all targets as specified below.
- The programs and libraries must abide by the class style guide.

The homework will be graded based on 100 points.

The directory structure, Makefile and delivery script, the log manager library	10 points
The random fortune generator	20 points
The shared memory library	25 points
The shared memory application programs	20 points
The install_and_monitor application along with the thread_mgr library	15 points
Code style	10 points

Requirements

The Random Fortune Generator

This program will exercise your knowledge of the **popen()** function call, as well as gaining experience with non-blocking or multiplexed I/O. (Note that this program is standalone, and does not require the shared memory library required to be built as part of this homework.) We didn't explicitly cover this call in our course content, but it is described in Section 15.3 of your text.

This program will execute a loop that runs (at random intervals ranging between 1 and 8 seconds) the ~jcn/unix_class/fortune/fortune (dev4) and ~jcn/unix_class/fortune_absaroka/fortune (absaroka) command (via the **popen()** function call). The implemented delay must not be a busy wait, and there should be no delay before the first fortune is printed. The output of the fortune command will be placed (in the pipe created by **popen()**) to be read from your main program. This output will be translated to upper case, then output to the screen.

To terminate the loop, your program will query standard input (without delaying or blocking the fortune loop). If input is successfully read from standard input, the first character of the input line will be checked. If it is a 'q', the program will then exit. There are many ways to query standard input; you may make standard input non-blocking, you might use signal-driven I/O or you might use multiplexed I/O using **select**(). (Note that it is ok for the user of your program to have to enter a line starting with a 'q', and then terminate the line with a carriage return to exit your program. There is no requirement to perform non-canonical (raw) mode terminal I/O – since that is covered in a later module – although it is not prohibited.)

Although there are no requirements in this exercise to handle signals, you should make sure that your program terminates cleanly upon receipt of a signal as well as normal exit via the line of standard input starting with ‘q’.

Shared Memory Library

This homework shall require the creation of one new library, called *shmlib*. This library will be built from a source file called shared_mem.c. This files will contain three (visible) functions, **connect_shm()**, **destroy_shm()**, and **detach_shm()**.

*void ****connect_shm***(int key, int size)*

This function has two arguments. The first argument serves as the key for the shared memory segment. The second argument contains the size (in bytes) of the shared memory segment to be allocated. The return value for this function is a pointer to the shared memory area which has been attached (and possibly created) by this function. If, for some reason, this function cannot connect to the shared memory area as requested, it shall return a NULL pointer. A program using this library function must be able to use it to attach the maximum number of shared memory segments to the calling process. (Note that Solaris 10 does not have a limit to the number of attachments, so you can use the limit that Linux supports – use the limit you find on our Linux server absaroka.apl.jhu.edu).

int ***detach_shm****(void *addr)*

This function detaches the shared memory segment attached to the process via the argument *addr*. The associated shared memory segment is not deleted from the system. This function will return OK (0) on success, and ERROR (-1) otherwise.

int ***destroy_shm****(int key)*

This function detaches **all** shared memory segments (attached to the calling process by **connect_shm**( )) associated with the argument *key* from the calling process. The shared memory segment is then subsequently deleted from the system. This function will return OK (0) on success, and ERROR (-1) otherwise.

Programs to Exercise the Shared Memory Library

Two programs will be written to exercise the shared memory library. For these programs, the shared memory shall have a required structure as described below. One function shall write information into the shared memory according to the required structure at certain intervals; the second will monitor the state of the data contained in the shared memory and report certain statistics as described below. Use the *log_mgr* library completed in homework2 to log any errors to a logfile. A third program which duplicates the functionality of these two programs in a single program using your thread library from homework3 shall also be required.

The format of the shared memory

The two programs described immediately following this section (4.3.2 install_data and 4.3.3 monitor_shm) communicate using shared memory as an array of structures. Each element of the array must be the following structure:

```
        struct {
                        int is_valid;
                        float x;
                        float y;
                };
```

The array will contain 20 elements. The first member of the structure, *is_valid*, shall contain 0 if the array element is not valid, and 1 if the array element is valid. Any valid floating point number can be entered into structure members *x* and *y*. Upon creation of shared memory, assure that all members of each array element are set to 0. (Make sure your shared memory library does not make any presumptions about this particular format; the shared memory format should only affect the following programs.)

install_data

The *install_data* program takes a file name as an argument. The file will contain lines of text which describes the data to be written into shared memory and the time (relative to the start time of the program) at which the data shall be placed in the shared memory area. The program will perform the following:

- Verify that the argument file is provided on the command line, and that the file can be opened for reading.
- Call **connect_shm( )** which should return a pointer to the shared memory area.
- Process the data from the file, a line at a time. Write the data to the shared memory at the designated time. (Be sure to verify that the index given in the input file is valid.) After you have installed all the data according to the input file, don't put any other data into shared memory.
- Call **destroy_shm( )** to delete the shared memory segment from the system. (Does this need to be coordinated with the use of the shared memory by *monitor_shm*? Why or why not?)
- Exit.

The file which *install_data* reads will be a text file. Each line of the file will follow the following format:

```
<index> <x_value> <y_value> <time increment>
```

where index ranges from 0 to 19 and indicates which element of the shared memory structure is to be written to; x_value and y_value are floating point numbers which are to be installed in the x and y members of that structure. If the time increment variable is non-negative, then this value represents the integral number of seconds to delay until the data on that line are installed in the shared memory. If the time increment value is negative, the absolute value of this increment represents the integral number of seconds to delay before making the corresponding index invalid. (The x and y values are ignored in this case.) There can be any number of white space (tabs or spaces) between each field on a line.

Additionally, *install_data* should handle errors in the input data file; the output of *install_data* (that is, what gets installed into shared memory) is undefined in this case, but the program should never "core dump" or get caught in an infinite loop due to errors in the format of the input file. (For more details on the operation of *install_data*, see the Example below.)

Additionally, upon receipt of a SIGHUP signal, the install_data program should clear the shared memory area, and re-install the requested data as described above from the beginning of the input file. Also, upon the receipt of a SIGTERM signal, the install_data program shall detach and destroy the shared memory segment and then exit.

monitor_shm

The *monitor_shm* program shall take one optional argument. This argument, if present, would be an integer which represents the amount of time in seconds to monitor the shared memory segment. If the argument is not present, 30 seconds will be the default value.

Approximately each second, this program will print a line of information to the screen about the contents of the shared memory. Information to be reported includes:

- Count of the active array elements (i.e. the number of elements which are valid)
- The average x value over the active array elements, and
- The average y value over the active array elements.

Before *monitor_shm* exits, it shall detach (but not destroy) the shared memory segment.

An example

Assume that the file input_data contains the following:

```
0 40.0 20.0 1
1 30.0 26.0 0
3 15.4 0.0 3
2 0.0 4.0 2
1 0.0 0.0 -1
3 0.0 0.0 -1
2 0.0 0.0 -1
0 0.0 0.0 -1
19 99.0 -10.0 2
18 -15.0 -2.5 1
```

If the following command was executed:

```
bin> install_data input data & monitor_shm 10
```

monitor_shm might report the following (your results may differ due to slight timing differences):

```
At time 0:no elements are active
At time 1:2 elements are active:x = 35.00 and y = 23.00
At time 2:2 elements are active:x = 35.00 and y = 23.00
At time 3:2 elements are active:x = 35.00 and y = 23.00
At time 4:3 elements are active:x = 28.47 and y = 15.33
At time 5:3 elements are active:x = 28.47 and y = 15.33
At time 6:4 elements are active:x = 21.35 and y = 12.50
At time 7:3 elements are active:x = 18.47 and y = 8.00
At time 8:2 elements are active:x = 20.00 and y = 12.00
At time 9:1 elements are active:x = 40.00 and y = 20.00
At time 10:no elements are active
```

Note that the times provided in the input file are time increments; if the time increment is 0, then that entry is installed with no delay. Note that a index cannot be deleted with no delay; a delay of at least a second is required.

install_and_monitor

After you have successfully completed the above, you then should combine these two programs into a single multi-threaded program using your thread_mgr library from homework 3. This *install_and_monitor* program shall work as described above, but only in the context of a single process - the *install_data* program should be one thread within the program, and the *monitor_shm* shall be in another thread. This program should **not** use the shared memory library - in this program the "shared memory" should only be global memory available to both threads.

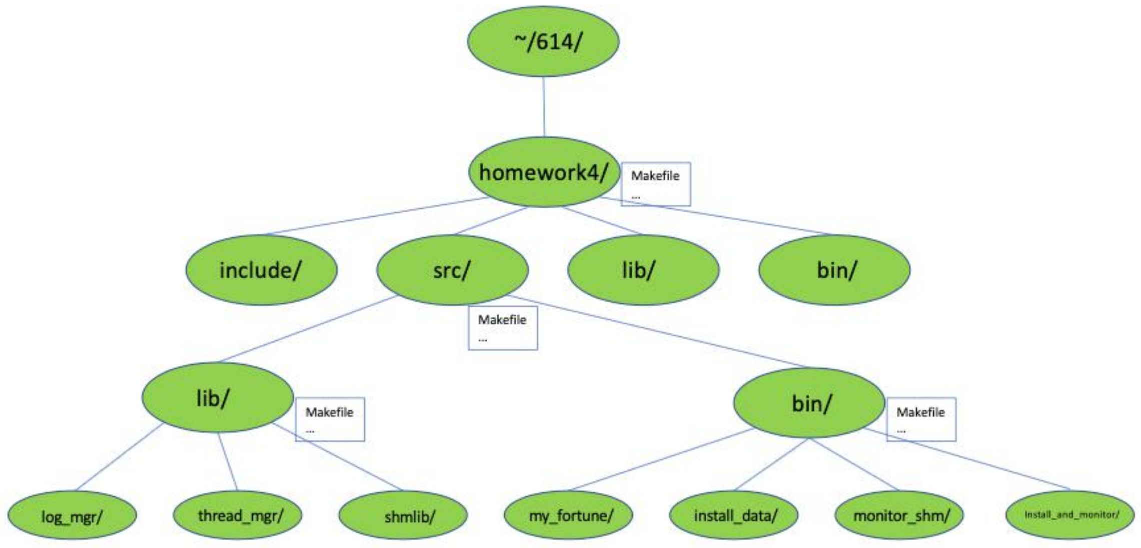
Homework Implementation

The design and implementation of the homework shall follow these guidelines:

- The homework shall consist of the required programs and libraries (including your log library from homework #2 (use your log library to log appropriate errors to the log file as well as printing the errors to the screen) and the thread_mgr library from homework #3 as organized as described in this document.

The following directory structure shall be enforced:

- In your ~/614 directory, there will be a directory called "homework4." The Perl script for the assignment will reside at this level.
- The directory hierarchy under this directory shall look as follows:



- The directory "homework4/lib" will contain project-specific libraries that you carried over from homeworks #2 and #3 (liblog_mgr.a, libthreads_mgr.a), as well as your new libshmlib.a).
- -The directory "homework4/bin" will contain project-specific programs that you shall create (my_fortune, install_data, monitor_shm, and install_and_monitor).
- -The directory "homework4/src" will contain the source code for the project, organized as shown.
- -The source for each program will be in the deepest directories of the above hierarchy. Each of these directories will contain a Makefile; the function of the Makefile will allow the UNIX command “make” to make the executable (or the library) and install it in the appropriate installation directory (either homework4/bin or homework4/lib).
- In addition, each upper-level directory should contain a Makefile; these Makefiles should allow 'make' to be run in these directories, the effect being to run make with the specified argument in each of the directories underneath. The targets to be supported are the same as in the previous assignments; namely it, install, depend, and clean.
- The directory “homework4/include” will contain project specific include files (include files that need to be shared among distinct programs).

Perl/Python Script

A Perl/Python script will be written that will create tar files of your homework4 for grading. This script shall be called “release.pl” or “release.py”, and it should reside in your ‘homework4’ directory. The Perl/Python script will support two options.

1.) Binary release (-b)

The script will confirm to the user that they have requested a binary release be generated. If the user confirms this is correct via (Y/N) on standard input, then the script will prompt the user for a hostname. (ex. absaroka) The script will then create a *tar* file of the homework4 directory containing the root directory (homework4) and the binary directory (bin) and its contents (the executables). The filename of this tar file will contain the homework assignment number (homework4), the hostname, and the .tar extension. (ex. homework4_absaroka.tar)

2.) Source release (-s)

The script will confirm that the user has requested a source release be created. If the user confirms this is correct via (Y/N) on standard input, then the script will perform a ‘make clean’ on the homework4 directory structure to remove dependency and object files, and will then create a *tar* file of the entire homework4 structure. The resulting file will have a filename that includes the assignment number (homework4) and the .tar extension. (ex. homework4.tar)

Final Words

The instructors reserve the right to clarify any point that is unclear in this document, and to correct any errors.

. . . .

605.614 System Development in the UNIX Environment