Module 7 Assignment
Homework Project #3
System Development in the UNIX Environment (605.614)

Overview

The third homework assignment for this class is to create a library to allow a set of processes to create, manage, and stop threads in a process.

Getting Started

Set up your project hierarchy as you did in the first two Project Assignments. Call the 'root' of your project 'homework3' in the 614 directory under your home directory (~/614). Be sure to copy your homework #2's log manager library in ~614/homework3/src/lib/log_mgr, and be sure to make repairs to the library based on the feedback. Then you will be ready to add your new thread_mgr library and the applications that will use the library.

Objectives

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

- Create a general use library to create, manage, and exit threads
- Use threads to perform concurrent tasks in a process.

Submission

Completion of the assignment involves completing, on or before the due date, the following materials:

1. The directory hierarchy and Makefiles along with the required source code.

2. A comment block at the beginning of each Makefile, 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.

3. The Assignment should be compiled and run on the (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 Assignment are:

1. The programs must be written in the 'C' or the 'C++' programming language.

2. 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.

3. Assignments are due by midnight of the due date as specified in the Course Calendar.

4. 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 assignment, you will run the Perl/Python script developed as part of your first assignment. The output of the Perl/Python 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.

Grading of the Assignment

The assignment will be graded according to the following criteria:

1. The project hierarchy must be as specified.
2. The log manager and the thread manager libraries must work correctly as specified.
3. The application programs making use of the libraries must function correctly as specified.
4. The programs and libraries must abide by the class style guide.

The directory structure, Makefile and delivery script	10 points
The log manager library	10 points
The thread manager library	45 points
Application programs	25 points
Code style	10 points

Requirements

Library Description – Thread Management Library (thread_mgr)

This library allows programs to easily create threads using the pthreads library. Your library shall create threads, kill threads, maintain the state of the threads and use signals to allow the state of the threads generated by the library to be displayed to the user. The number of threads supported by this library can be limiited to 50 (dynamic allocation is not required). The following sections contain details on the calls that must be supported by the library.

int th_execute ( )

```
typedef int ThreadHandles;
typedef void *Funcptrs (void *);

ThreadHandles th_execute (Funcptrs);
```

This library call executes the argument function as an independent thread within the process. The library will create a name for the thread and maintain it within the library. The library shall also create a unique integer handle (ThreadHandles) and return it upon successful execution. If the function fails, it shall return THD_ERROR (-1).

int th_wait (ThreadHandles)

This call blocks the calling thread until the thread associated with the argument handle terminates. This call returns THD_ERROR if the argument is not a valid thread. Otherwise the thread returns THD_OK (0). After the thread terminates, the thread library should purge the stored thread information for the argument thread.

int th_wait_all (void)

This function blocks until all threads in the library terminate. This function returns THD_ERROR if the library is not managing any threads or upon any other error condition. Otherwise, the function returns THD_OK after all threads terminate. The thread library should purge the stored thread information for all threads upon successful execution of this call.

int th_kill (ThreadHandles)

This function cancels the executing thread associated with the argument thread handle, and updates the status of the thread appropriately. This function returns THD_ERROR if the argument is not a valid thread handle. Note that this call is not required to asynchronously kill the thread; the thread may be cancelled until the thread reaches its cancellation point, and cleaned up after the application waits for the thread.

int th_kill_all (void)

This function cancels all threads in the library. This function returns THD_ERROR if the library is not managing any threads. Otherwise, the function returns THD_OK after all threads are cancelled.

int th_exit (void)

This function should allow the thread that calls this function to clean up its information from the library and exit. The thread information in the library should not be purged at this time; however, proper status should be logged to the log file, and the internal status of the thread should be updated. The thread information in the library shouldn't be changed until another thread 'waits' for the thread (using one of the 'th_wait' calls).

This call does not return if executed successfully; thus the only possible return value for this function is THD_ERROR.

Signal Processing

In addition, your library should create a signal handler for the SIGINT signal. When a process using this library receives a SIGINT signal, the library should print out to standard output the thread handle, the thread name, and the status of the thread for all threads that are being managed by the library.

Your library also should catch the SIGQUIT signal. Upon receipt of this signal, the library should forcibly terminate or cancel all threads, and change their state to killed. You should not wait for the threads in the SIGQUIT signal handler, as this call blocks, and you should not block in a signal handler.

Make sure multiple signals of the same type do not terminate the process on either supported host. Also, you should make sure you use sigaction() to install the signal handlers to insure consistent behavior between the supported hosts.

The signal handler should be installed implicitly when the th_execute() function is called the first time.

The process should not exit based on either of these two signals.

Generally, busy loops should be avoided. (Although this is mentioned in the Signal Processing section, this is true in general.) If such a loop cannot be avoided, at least have some delay that would not be noticed by the user of a quarter second or so.

Documentation

As in the prior projects, make sure your thread_mgr.h file contains sufficient documentation on each of these calls for a user of the library.

Program Descriptions

The App1 Program

This program shall use the above library to create up to twelve threads, and then wait for all the threads to complete before exiting. The program shall require one argument, which shall be an integer indicating how many threads to create (which must be between 1 and 12, inclusive).

For example:

```
% ./app1 6
```

The program shall start all of these threads concurrently. Each thread shall execute a function that shall perform some extensive mathematical computations of your choice. One sample program incremented an integer variable five billion times; but this number depends on the machine running the program. Any thread should take about roughly 5-20 seconds to execute (make sure the thread is busy; don't allow the thread to block).

The threads in this program must not have to have any cancellation points, nor should they immediately terminate when cancelled. They should only terminate when they complete their extensive mathematical computations.

Each thread should announce its creation with a log message using your log_mgr library of homework 2.

It is allowed and recommended to have status of thread creation and exit displayed on standard output as well as in the log file. After all threads have completed their work, they can exit normally (using th_exit() function in your thread_mgr library). Be sure to log the event of thread exit as well.

The App2 Program

This program also shall take two arguments. The first argument indicates a number of threads to create between 1 and 25. The second argument shall be a float indicating an amount of time in seconds between 0.1 and 10.0. This program can create up to 25 threads; however, this program shall create one thread each specified time interval, until the requested number of threads is created. There should be no delay before creating the first thread. Once all threads have been created, the program shall wait five seconds starting once the last thread is created. After that delay, the app2 program should start killing each of the threads one per second until no more threads are running. (There should be no additional delay after the 5 second delay before the first thread is killed.) At that point, the app2 program can exit.

For example, this creates 6 threads; the first immediately, with a 2 second delay between subsequent thread creation:

```
% ./app2 6 2
```

In this program, the threads must block – there is no requirement for the thread to do any work. However, these threads should not terminate on their own.

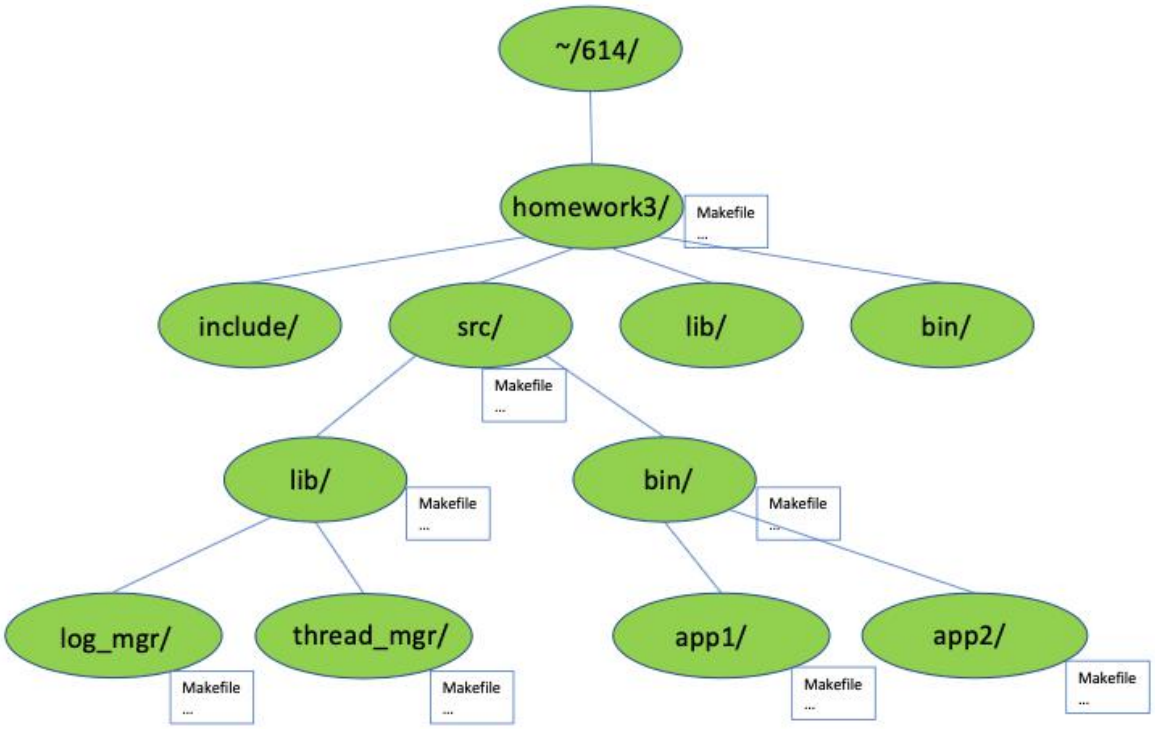
It is required to use the your homework2 log library to capture the status of thread creation, cancellation, thread exit and the result of waiting for the thread (if applicable). It is also required to display messages relating to thread creation and thread cancellation to standard output as well.

To implement the periodic behavior in this function, you must use the **setitimer( )** system call.

Implementation

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

- The homework shall consist of the two programs, the thread_mgr library, and your log library from homework #2 (use your log library to log appropriate errors to the log file) organized as described in this document.
- The following directory structure shall be enforced:
- In your ~/614 directory, there will be a directory called "homework3." The Perl or Python release script for the assignment will reside at this level.
- The directory hierarchy under this directory shall look as follows:



- The directory "homework3/lib" will contain the thread_mgr library as well as libraries that carried over from homework #2 (liblog_mgr.a).
- The directory "homework3/bin" will contain project-specific programs that you shall create (app1 and app2).
- The directory "homework3/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 homework3/bin or homework3/lib). The targets to be supported are the same as in the previous assignments; namely it, install, depend, and clean.
- The directory “homework3/include" will contain project specific include files (include files that need to be shared among distinct programs).

Perl/Python Script

A Perl or Python script will be written that will create tar files of your homework3. This script has no additional requirements beyond those in the previous assignments.

The Perl or 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 homework3 directory containing the root directory (homework3) and the binary directory (bin) and its contents (the executables). No libraries or source code should be included. The filename of this tar file will contain the Assignment number (for this time, “homework3”), the hostname, and the .tar extension. (e.g. homework3_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 homework3 directory structure to remove dependency and object files, and will then create a tar file of the entire homework3 structure. The resulting file will have a filename that includes the assignment number (homework3) and the .tar extension. (e.g. homework3.tar)