CPE453 Laboratory Assignment #2
The CPE453 Monitor

Michael Haungs, Spring 2011

1 Objective

As multi-core CPUs become commonplace, there is an increasing need to parallelize legacy applications. In this assignment, you will parallelize the performance monitor, *mond*, that you created in Lab#1. You will use pthreads, Linux's POSIX compliant thread library, to create separate threads to collect data on processes, collect system statistics, and handle real-time user commands. Your threads will have to use IPC synchronization mechanisms to protect shared resources.

2 Resources

You should read the following Linux manual pages:

- pthread_create: Performs thread creation.
- pthread_join: Blocks calling thread until targeted thread terminates.
- pthread_cancel: cancel execution of a thread.
- pthread_testcancel: exit the thread if it has been given a cancel request.
- pthread_cleanup_push, pthread_cleanup_pop: add/remove a cleanup function for thread exit.
- pthread_setcancelstate, pthread_setcanceltype: set cancelability state.
- sem_post, sem_wait: increment and decrement operations on semaphores.
- pthread_mutex_lock, pthread_mutex_unlock: Locks/Unlocks a mutual exclusion device.

3 Assignment

I'm assuming you, or your partner, have a working version of Lab#1. If you do not, your first priority should be revising your Lab#1 submission.

Your threaded *mond* program needs to meet all the requirements of Lab#1, except:

1. You no longer have commandline arguments to mond. All commands will be typed dynamically by the user using a simple text-based interface.
2. You'll have the ability to monitor multiple processes with a unique thread monitoring each process. Each thread will individually report its data every "interval" seconds. Before, all data was reported at the same time.

3.1 Threads

You need to create three different types of threads in your *mond* program: a *command* thread, *systems* thread, and a *monitor* thread. The responsibility of each type of thread, and when to create it, is described next.

3.1.1 Command Thread

We are making *mond* a more flexible and powerful tool. The *command* thread is the thread that will read commands typed in by the user, parse them, and execute them. Unlike Lab#1, this thread can block waiting for user input.

The behavior of *mond* is now controlled by a small set of user commands. In executing these commands, the *command* thread will be required to create and destroy *monitor* threads and/or the *systems* thread, list active threads, set default values, and exit.

Here is a brief summary of the commands:

- add <-s || -p processID || -e executable > [-i interval] [-f logfile]
  - This command creates a new *monitor* thread to either collect system or process statistics. The process can either be an existing process indicated by *processID* or a new process indicated by *executable*[1]. Instead, the user can just use "-s" to indicate collecting system statistics. The user is only allowed to create one system monitoring thread and should be given an informative message if they try and create more than one. The *add* command optionally takes an interval and logfile parameter. If the interval or logfile default values have not been set and they are not provided in this command, then it will return an error message stating that the user needs to set default values for these. The program should continue normally after printing the error message.
  - Examples of correct usage:
    - add -s -i 100 -f systemData
    - add -p 3905 -i 25
    - add -p 4727 -f dataLog
    - add -s
  - Examples of incorrect usage:
    - add 3905 -i 10 -f myLogFile
    - add -s myLogFile
- set interval <numberInMicroseconds>
  - This command sets the default interval value.
- set logfile <logFileName>
  - This sets the default name for the log file.
- listactive
  - Displays a table in which each row contains the following information about a specific item being monitored:
    - Thread id of the thread doing the monitoring, the process id of the process being monitored or the text "system" if the thread is monitoring system statistics, timestamp indicating when monitoring started, monitoring interval, and, finally, the name of the log file the thread is writing data to.

[1] In this case, you will fork() and exec() the executable in the same way as you did for Lab1.

- listcompleted
  - Displays a table in which each row contains the following information about a specific item that was being monitored, but either was stopped with the *remove* command (see below) or the process has exited normally:
    * Same fields as described in the "listactive" command, except that there is one additional timestamp. This timestamp indicates when the monitoring of the process finished and is displayed right after the timestamp indicating the starting time.
- remove <-s || -t threadID>
  - This command cleanly terminates the thread indicated by *threadID* or the systems thread if *-s* was specified. The thread stops monitoring, releases any shared resources it had, closes the logfile if it is the last thread using said file, and terminates. Note: Monitored processes are unaffected by this command.
- kill <processID>
  - This command is very similar to the *remove <-t threadID>* command except that it terminates all threads associated with the process *processID* and then terminates the process.
- exit
  - This command cleanly terminates (closing open files, terminating threads, releasing shared resources) *mond*. If there is still active monitoring in process, prompt the user with the following, "You still have threads actively monitoring. Do you really want to exit? (y/n) ". If the user types "n", then ignore the *exit* command, otherwise proceed in performing the *exit* command.

3.1.2 Systems Thread

Simply, the systems thread is the thread you will create to monitor system statistics. If you are currently not monitoring system statistics, you will not have one of these threads running. This thread will write directly to the appropriate log file.

3.1.3 Monitor Threads

These are the threads that will gather processes statistics. There will be one of these threads for each process being monitored. Each one will independently update their log file every interval seconds. Each thread is also responsible for shutting down all acquired resources before terminating.

3.2 Developement

Threaded programs can have some very complex interactions. The usual trial-and-error method of programming won't be very successful in this type of environment. Before you start programming, you should think about the threads and shared resources in your program. To aide you in this, do the following exercise:

1. List all the threads in your program and their responsibilities.
2. List all the resources that any two of your threads may share.
3. Draw a diagram of your threads and resources:
   (a) Draw a circle for each type of thread you have in your program.

(b) Draw a square for each resource you have in your program.

(c) Draw a line from a thread (circle) to a resource (square), if that thread will access that resource anytime during the execution of *mond*. Label those lines in the following way:

i. If the thread will only read the resource, label it with an "r".

ii. If the thread will only change the resouce, label it with a "w".

iii. If the thread will both read and update, label it with an "r+w".

In lab, I may ask you to see this diagram before answering your question.

3.3 Synchronization

Race conditions are:

> A situation where more than one process [or thread] access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

–Silberschatz et al

Places in your code that can result in a race condition are referred to as critical sections. You will have multiple critical sections in your program that you will need to synchronize access to using a semaphore or mutex. In this lab, you will have a critical section everytime you have one or more threads trying to update a shared resouce. Luckily, you identfied all your shared resources above in the "Development" section.

3.4 Analysis

For every critical section in your code, you are going to have to provide the following:

- A description of the crtical section that includes what threads use this critical section and what shared resource is being protected.
- A short justification of why each line of code is included in the critical section. You should also discuss performance concerns in this section.
- Discuss why you decided to use a semaphore or mutex to protect this critical section.

4 Examples

Coming soon.

Deliverables

You will submit your work to me on April 21 before midnight. You need to submit the following:

1. All ".c" and ".h" files associated with your solution.

   (a) Each file should start with a comment formatted like this:
   http://users.csc.calpoly.edu/~mhaungs/courses/CSC453/comment_hdr.htm

2. A Makefile (must be spelled like that) that will compile your entire project when "make" is ran and produce an executable named *mond*.

3. A README (again, must be spelled exactly like that) file that describes (1) what you got working, (2) test cases that will work with your code, (3) Any bugs in your code, and (4) the discussion of your critical sections discussed in Section 3.4. Your README file should also list your name and your partner's name at the top of the document.

Use the *Digital DropBox* tool in Blackboard to submit your files:

1. Create a zip archive of your work.

   (a) Rename the zipped folder in the following way:
   Student1Lastname_Student2LastName_Lab2.zip.

2. When using the Digital Dropbox tool, be sure to use the *Send File* button and not the *Add File* button to send me your compressed folder.

   (a) Only select me, Michael Haungs, as the recipient of your send.

GRADING NOTE 1: You must name all your files exactly as I specified above. Your output must exactly match the examples I have given above.

GRADING NOTE 2: Remember, you must have a different partner than the one you had in Lab#1.