CSUS, College of Engineering and Computer Science
Department of Computer Science
CSC/CPE 139 – Operating System

Programming Assignment 2 – User-level Threading
Release Date: March 30, 2017
Due Date: April 24, 2017 (11:59pm)

Goal: In this project, you will implement a user-level threads package in C.

Acknowledgements: Significant portions of this project have been adapted from a project given by Prof. Urgaonkar of Penn State University, which was further adapted from Prof. Saran of IIT Delhi.

Hardware/Software: You need access to a machine with Intel-based CPU, running UNIX/Linux, and loaded with C-related software (see below). It is tested that virtual machines with Linux environment is OK to run.

Instructions: Here is a list of things to do.

1. Refresh your understanding of linking, compiling, and debugging (e.g., gcc, gdb, make) C programs.

2. Learn about the following:

- *sigsetjmp* and *siglongjmp*. You should carefully read the man pages for these functions. You may choose to use variants/enhanced versions of these functions, if available. Check out the sample file (demo.c), also available in the same folder as this document, for a simple example of context switching in user space.
- How signals are sent/handled by C programs (system calls *signal* () and *kill* ()). Check out a second sample program (timer.c), also available in the same folder as this document, which implements a very simple signal handler. Play with it to reinforce your understanding of the creation and handling of signals.
- Function pointers in C. See below for why they are needed in this assignment.

3. Implement context switching using *setjmp* and *longjmp* as discussed in class. Implement two preemptive scheduling algorithms:

- Round-robin
- Lottery scheduling.

4. Design data structures for thread entities.

5. Implement the following functions:

- int CreateThread (void (*f) (void)) - this function creates a new thread with f() being its entry point. The function returns the created thread id (>= 0) or -1 to indicate failure. It is assumed that f() never returns. A thread can create other threads. The created thread is appended to the end of the ready list (state = READY). Thread ids are consecutive and unique, starting with 0.
- void Go() - This function is called by the main process to start the scheduling of threads. It is assumed that at least one thread is created before Go() is called. This function is called exactly once and it never returns.
- int GetMyId() - This function is called within a running thread. The function returns the thread id of the calling thread.
- int DeleteThread(int thread_id) - deletes a thread. The function returns 0 if successful and -1 otherwise. A thread can delete itself, in which case the function doesn't return.
- void Dispatch(int sig) - the thread scheduler. This function is called by the interval clock interrupt and it schedules threads using FIFO order. It is assumed that at least one thread exists all the time - therefore there is a stack at any time. It is not assumed that the thread is in ready state.
- void YieldCPU() - this function is called by the running thread. The function transfers control to the next thread. The next thread will have a complete time quantum to run.
- int SuspendThread(int thread_id) - this function suspends a thread until the thread is resumed. The calling thread can suspend itself, in which case the thread will YieldCPU as well. The function returns id on success and -1 otherwise. Suspending a suspended thread has no effect and is not considered an error. Suspend time is not considered waiting time.
- int ResumeThread(int thread_id) resumes a suspended process. The process is resumed by appending it to the end of the ready list. Returns id on success and -1 on failure. Resuming a ready process has no effect and is not considered an error.
- int GetStatus(int thread_id, status *stat) - this call fills the status structure with thread data. Returns id on success and -1 on failure.
- void SleepThread(int sec) - the calling process will sleep until (current-time + sec). The sleeping time is not considered wait time.
- void CleanUp() - shuts down scheduling, prints relevant statistics, deletes all threads and exits the program.

6. Finally, implement the following two functions:

- int CreateThreadWithArgs(void *(*f)(void *), void *arg) f is a ptr to a function which takes (void *) and returns (void *). Unlike the threads so far, this

thread takes arguments, and instead of uselessly looping forever, returns a value in a (void *). This function returns the id of the thread created.

- void *GetThreadResult(int tid) waits till a thread created with the above function returns, and returns the return value of that thread (pardon the alliteration!) tid is the id of thread. This function, obviously, waits until that thread is done with. Use semaphores (we would have revised these in class by the time you get to this part of the implementation) to ensure GetThreadsResult waits for the thread to return.

7. Some important constants to use:

- MAX_NO_OF THREADS 100 /* in any state */
- STACK_SIZE 4096
- TIME_QUANTUM 1*SECOND

Hints: Here are some important hints:

1. The C calling convention says that parameters are pushed on the stack. So, to pass an argument to the function, put the argument (the void ptr) at the top 4 bytes of the stack (make sure it's in little endian order!), and then decrement the stack ptr.

2. Where does a function (thread) return? To the return address. Where is the return address? On the stack (surprise!). It comes after the params. So after pushing the param, push the return address, and again decrement the stack ptr by 4.

3. But where to return? To a procedure which does some wrapping up, like setting the state of the thread to FINISHED, so that the dispatcher doesn't try to schedule it again, and storing the return value somewhere.

4. Finally, where is the return value? On Intel machines, at least, it's returned in the eax register. That's part of the C calling convention: return values are in eax. So all you have to do is get the contents of eax. Here is one way to do it. (Also, remember that setjmp reads the register file into a buffer, so that may be another option.) Write a three line assembly routine, and call it like any normal function from your program.

Outputs: Some outputs I would like to see:

You will have to think of writing functions that various threads in your demonstration will execute to best bring out the following aspects.

1. Demonstration of the proper working of round robin and lottery scheduling policies.
2. For each of the threads that are created: (a) thread id, (b) state transition sequence (running, ready, sleeping, suspended). (c) number of bursts (none if the thread never ran), (d) total execution time in msec(N/A if thread never

ran), (e) total requested sleeping time in msec (N/A if thread never slept), (f) average execution time quantum in msec (N/A if thread never ran), (g) average waiting time (status = READY) (N/A if thread never ran).

Deliverable: Submit the following through SacCT, in a **tar-ball**:

1) Source files i.e. .c files, .o file, and a report;
2) The report should contain source code and necessary screenshots. Please submit both an **electronic version** to **SacCT** and a **printed submission** to me/my office.

Requirement: The report will all be evaluated based on the following grading criteria.

Functionality	60%
Robustness	40%