CSCI 5103 - Operating Systems

User Level Thread Library

By

Loose Threads:

Bhaargav Sriraman - srira048@umn.edu
Edwin Nellickal - nelli053@umn.edu
Subhash Sethumurugan - sethu018@umn.edu

Table of Content

Overview

The project focuses on implementing basic thread library functionality, along with the relevant APIs for the control of the threads that are created/deleted by the self-declared APIs. With the help of the implemented APIs, basic Round-Robin scheduling is implemented in a preemptive fashion. Few test cases are written to showcase the multithreaded nature of the code that is written with the help of a few toy examples that showcase simple calculations.

Engineers in the industry are sometimes required to implement their own thread library due to:

- The absence of a pthread library on some embedded systems;
- The need for fine-grained control over thread behaviour; and
- Efficiency

The user-level thread library can replace the default pthread library for general use as it provides similar functionalities.

Assumptions:

1. Stubs aren't used to manage the thread of worker creation and exit. Stub was omitted from our implementation, as the stub reads from the stack when a worker thread is put into context.
2. Arguments are not passed to the worker function. The worker function would initialize any variable locally which it would want to use. It also updates a global variable that is used to achieve the final result.
3. The only interrupt of concern is the timer interrupt (SIGVTALRM). This alarm is used to keep track of how long a thread is running on the processor. Each time before a context switch is made, setitimer is invoked so as to restart the timer as per the time slice defined in uthread_create.
4. The time slice is an integer and is in milliseconds. The time slice value is an integer in nature as the timer data structure does not store fractional values.
5. The project works as a Single-core system, as per the scope of the project. Round-Robin scheduling is implemented, keeping in mind that one thread is executing on the processor at a given time.
6. All files are Stored and Compiled in the same folder, so as to avoid complicating the makefile.

API description:

1. **int uthread_create(void *(*start_routine)(void *), void *arg);**
   Creates a thread. The main thread is created when this API is called for the first time. This is to keep the context of the calling function. The state of the main thread will be RUNNING.
   After that, a separate context is created for each thread. This includes a stack whose size is 4096 bytes, a stack pointer which will be pointing to the head of the stack and a program counter pointing to the start_routine. The arguments are not passed to the routine. This functionality is yet to be implemented.
   It returns the thread ID.

2. **int uthread_init(int time_slice);**
   The input is the time slice the user would want to define for the thread that is being created. Returns 0 by default, since it is a simple wrapper to initialize a struct. Sets the time slice in milli-seconds. The time slice determines how often a thread is interrupted the timer interrupt and put back into the ready queue by the scheduling logic.

3. **int uthread_yield(void);**
   This API puts the running thread in the READY state. It will then swap context to the next thread chosen using the round-robin scheduling algorithm.

4. **int uthread_join(int tid, void **retval);**
   The API takes the ThreadID (tid) of the thread on which the current worker thread would want to wait. Also sets the running thread's state to BLOCKED, places the TCB of the current thread onto the blocked queue and switches context to the thread associated with the tid.
   Returns 1 by default.
   Returns 0 if the tid is invalid.

5. **int uthread_terminate(int tid);**
   The API moves the thread with thread ID tid to the finished queue. If it is the running thread terminating, it will switch to a new thread based on round-robin scheduling.
   Returns 1 after moving it to the finished queue.
   Returns 0 if tid is invalid or if the ready queue is empty

6. **int uthread_suspend(int tid);**
   The API changes the thread with thread ID tid to a suspended state. It will remain to stay in that state until a uthread_resume is called.
   Returns 1 after changing the state.
   Returns 0 if tid is invalid or if the ready queue is empty.

7. **int uthread_resume(int tid);**
   The API puts the thread with thread ID tid to a ready state. It will continue executing the current thread.
   Return 1 after pushing it to the ready queue.
   Return 0 if tid is invalid.

8. **int lock_init (lock_t*);**
   Initialize the lock variable to FREE.

9. **int acquire (lock_t*);**
   Implement a Test and Set atomic instruction to acquire the lock.
   Returns only after the lock is acquired.

10. **int release (lock_t*);**
    Resets the lock back to FREE which enables other threads to acquire it.
    Returns 1 by default.

Customized APIs

1. **void enableInterrupts(); & void disableInterrupts();**
   enableInterrupt sets the mask through sigemptyset. DisableInterrupt clears the mask that was set. Both APIs do not take any inputs and have the return type void.

2. **void swapcontext(TCB* new_thread, TCB* old_thread);**
   The API takes the pointer to the TCB for a new thread, the thread to which we would want to switch to an old thread, the pointer to the TCB from which we are going to switch from. It first sets the timer as per the slice defined in uthread_init(). Sets the jmpbuf by calling sigsetjump and sets the running thread pointer to the new thread TCB. After which it sets the state as RUNNING and enables interrupts. When the PC returns after siglongjmp to the sigsetjump call, the return value would be 1, implying that siglongjump is successful and enables interrupts. The return type is void.

3. **TCB* getTCB(int tid);**
   The API Takes thread id as int. This looks through the running thread, blocked queue, ready queue and finished queue to find the TCB associated with the tid passed to the wrapper. Returns pointer to the TCB associated with the tid that is queried or returns NULL in the case no corresponding TCB is found.

4. **void timer_handler(int sig);**
   The API takes the signal value to which the timer handler is listening on. At first, the API disables interrupts. Pushes the running thread to the ready queue after setting its state to READY.
   If the ready queue is empty, it sets the timer so as to wait for any other thread to be created or resumed. If the ready queue has a TCB to be scheduled, the TCB pointer is picked up and a context switch is done to that thread. Returns void.

Questions:

How does the different length of timeslice affect performance?

1. If the timeslice is small, the processor spends more time switching context than running the worker function. Excessive context switches keep the processor busy and reduce the throughput.
2. If the timeslice is too large, few threads may horde the processor for longer, which may not be desired in an interactive environment. This may lead to starvation of a few threads which may require little time to complete their task.

What are the critical sections your code has protected against interruptions and why?

1. The interrupts are disabled while updating the queues and during swap context. This process needs to be atomic. If there is a timer interrupt in between, it will preempt the running process which will lead to inconsistencies.
2. They are also disabled when the timer interrupt handler is running. It should not be interrupted with another timer interrupt while handling the previous one.
3. It is important to note that the library assumes that it is running on a single-core system. This is why the critical data structures are not explicitly implemented to be thread-safe.

If you have implemented any advanced scheduling algorithm(s), describe them.

1. We implemented Round-robin scheduling with preemption.

Does your library allow each thread to have its own signal mask?

1. All of the threads have the same signal mask. The API for enabling/disabling interrupt does not take any input. The masks are saved in the jmpbuf.

How did your library pass input/output parameters to a thread entry function?

1. As per the scope of the project, we assume that the worker threads have no arguments. The threads receive all required information from global variables.
2. The threads update the global variables by acquiring locks. The final result is stored in global variable after all threads complete.

README

I. Tests:

1. **test_all_APIs.cpp:**
   Description - tests all APIs. The test does the following:
   a. Spawns 10 threads
   b. Thread 5 is suspended
   c. Each thread increments a global variable after acquiring the lock
   d. All threads (except 5) join the main thread after terminating themselves.
   e. Thread 5 is resumed.
   f. Thread 5 joins the main thread after completion
   g. Print output

2. **test_array_sum.cpp:**
   Description - Computes the sum of an array by spawning given number of threads. The test does the following:
   a. Accepts number of threads as an argument
   b. Generates an integer array with random numbers
   c. Spawns n threads
   d. Each thread reads one array element and updates the sum after acquiring the lock
   e. All threads join the main thread after terminating themselves
   f. Print output

3. **test_lock.cpp**
   Description - Tests lock. The test does the following:
   a. Accepts number of threads as an argument
   b. Each thread increments a global variable after acquiring a lock. (Thread releases lock after update)
   c. All threads join the main thread after terminating themselves
   d. Print output

4. **test_suspend_resume.cpp**
   Description - Tests suspend and resume. The test does the following:
   a. Spawns 10 threads
   b. Each thread increments a global variable after acquiring lock. (Thread releases lock after update)
   c. All threads join the main thread after terminating themselves
   d. Print output

II. Steps to build the tests:

`make all`	: builds all tests
`make test_name`	: Builds the particular test
`make clean`	: Deletes the object files for all tests

III. Steps to run the tests:

`./all_api`	:Runs test_all_APIs.cpp (10 threads)
`./array_sum <num_threads>`	:Runs test_array_sum.cpp (Accepts number of threads as argument (default 10))
`./lock_test <num_threads>`	**:**Runs test_lock.cpp (Accepts number of threads as argument (default 10))
`./suspend_test`	:Runs test_suspend_resume.cpp (10 threads)

References:

[1] http://vmresu.me/blog/2016/02/09/lets-understand-setjmp-slash-longjmp
[2] operating systems: principles & practice, 2nd edition, anderson and dahlin, 2014
[3] http://man7.org/linux/man-pages/man2/setitimer.2.html
[4] https://linux.die.net/man/3/sigemptyset