Overview

You are to write and test a C/C++ program that implements a simple threads library. The features of the library uth_lib are described below. This library supports preemptive scheduling of user-level threads and is very similar to the user-level thread interface already discussed in class. As extra credit, you may choose to support the synchronization primitives associated with mutex locks and condition variables.

User-Level Threads API

The library should support the following functions:

- **uth_init( int N, int TickSz, int Debug )**

  Initialize user thread system to support atmost N threads and a clock tick size of TickSz milliseconds. If TickSz is 0, threads should run without preemption (i.e., no interval timer). TickSz should never be smaller than the OSes hardclock period (usually 10 msec). If Debug is 1, output a synopsis of the run queue(s) and the state of SLEEPING or JOINING threads before each context switch. For example:

  Run Queues (curtid = 2, reason = P):

<table>
<thead>
<tr>
<th>Queue</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi</td>
<td>Empty</td>
</tr>
<tr>
<td>Lo</td>
<td>2 3</td>
</tr>
<tr>
<td>Idle</td>
<td>0</td>
</tr>
<tr>
<td>Sleeping</td>
<td>Empty</td>
</tr>
<tr>
<td>Joining</td>
<td>1</td>
</tr>
</tbody>
</table>

  shows a run queue with no high-priority threads, low-priority threads 2 and 3, and idle thread 0. The thread running before the context switch has tid 2 and the reason for the context switch is a (P)reemption. Other reasons might be (Y)ield, (S)leeping.

- **int uth_create( uth_tid_t *Thr, uth_pri_t Pri, func0_t *Thr, void *Arg )**

  Create a thread with priority Pri and parameter pointer Arg; put it at the end of the run queue and return 0 if OK, -1 otherwise. The three typedefs above are:

  ```
  typedef int uth_tid_t;
  typedef int uth_priority_t;
  typedef void func0_t(void);
  ```

  or equivalently:

  ```
  int uth_create( int *Tid, int Pri, void (*Thr)(void), void *Arg );
  ```
• uth_tid_t uth_self( void )

   Return my own task identifier. Typically, the idle thread has an id of 0, the main
   thread has an id of 1, and the other reads have an id of 2 or higher.

• void uth_exit( int Status )

   Return Status to the thread that calls uth_join on the current thread.

• int uth_join( uth_tid_t Tid, int *Result )

   Suspend the current thread until thread Tid calls uth_exit and return the result
   in Result. Return 0 if call is valid; else -1. It is an error if any other thread
   has already joined with thread Tid.

• void uth_yield( void )

   Yield control to the highest priority thread on the run queue. Place the current
   thread in the run queue so that it is behind all other READY threads in its priority.

• void uth_sleep( int Msec)

   Put the current thread to sleep for atleast Msec milliseconds.

• int uth_usage ( uth_tid_t Tid )

   Return the CPU usage (number of clock ticks) of the thread Tid.

Implementation Notes

As of the writing of this assignment, the u_context facility only works 100% on 32-bit Linux
OSes (e.g., hive.exe). It only works partially on 64-bit Linux OSes although I hope to post a
workaround, but that is not a guarantee. Watch the course web page for announcements of changes
to this state of affairs. But you should assume that the facility will NOT work on cygwin and
Solaris.

In this implementation, you can assume the following:

• The system will be small and therefore simple data structures are appropriate (i.e., there is
  no need at this time for sophisticated data structures). It is up to you to determine what you
  will need, but remember that simplicity will be a virtue in this assignment.

• The user will not make incorrect calls to the above interface functions. This means that the
  only error checking you need are those needed to protect your code and that if such errors
  occur, you can just print an error message and exit.

• An error should terminate your program with an error message if that will simplify your code.

The correctness requirements for this project are a little different than Project A in that there
is a sequence of milestones defined by test cases that I will supply. The documentation template
will ask that you submit the test output for all successful milestones and explain why the output
from your most impressive test case is correct. Note that these milestones do not represent equal
amounts of work. I suspect that Milestone A is the most work; B is a small step from A; and C is
a medium step from B. The milestones have the following main features:
• **Milestone A**

The main thread initializes and creates two low-priority threads that each compute the same small fibonacci number without preemption and communicate the result through shared memory (i.e., global variables). There are no high-priority threads. The calls used are: `uth_init`, `uth_create`, `uth_yield`, `uth_exit`.

• **Milestone B**

The main thread initializes and creates two low-priority threads that each compute the same small fibonacci number without preemption and communicate the result through shared memory (i.e., global variables) but indicate that they are done through the `uth_join` call. There are no high-priority threads. The additional calls used are: `uth_join`.

• **Milestone C**

The main thread initializes, creates four low-priority threads that each compute the same large fibonacci number many times, and creates a single high-priority thread that monitors and periodically records the CPU usage of the fibonacci threads. A RR scheduler picks the highest priority thread to run in a RR fashion. The additional calls used are: `uth_usage` and `uth_sleep`.

**Additional Guidelines**

- Code readability is of the utmost importance. The Web page will contain a summary of coding guidelines that you should follow in spirit. I am not rigid about these guidelines, but unreadable code will be penalized.

- All ucontext-type calls that should never fail (e.g., `swapcontext`) should be wrapped so that any fatal errors will cause an error message to be displayed followed by an exit. By convention, the wrapped system call name will be the same as the actual system call name except the first character should be capitalized (e.g., `Swapcontext` is the wrapped version of `swapcontext`).

**What to Submit**

The CSE422S Web page contains a link to the documentation template. You should complete the template and submit it in both hardcopy AND electronic form. Submit the completed documentation template AND a listing of the source code. The electronic submission (described below) should include the completed documentation template, the source code, the Makefile, test scripts, and test output. **Both the electronic copy and the hardcopy are due by class time.** This submission is worth 100 points.

**Important:** Even if you do not completely implement a feature, you should still make a submission. In a few cases, a completely working project will suddenly refuse to do anything useful or start segfaulting in unexpected places. In these cases, adjust your tests and documentation template to give a clear, coherent picture of what aspects do/did work and an explanation of the evidence backing your claims.

**Electronic Submission**

The end result should be that you mail to kenw@arl.wustl.edu a single `shar` (shell archive) file containing your files. Do **NOT** submit object code or executables. The following commands will create a shar file named A.shar containing the files `xssh.c` and other files and then send mail to me:
shar README Makefile xssh.c ... other files ... > B.shar
mail -s B.shar kenw@arl.wustl.edu < B.shar       # mail is usually in /bin

The README file is the completed documentation template. If you prefer, send the shar file as
an attachment and use whatever mailer you are comfortable with. **NOTE:** The shar file should be
relatively small (try `ls -l`) and make sure it is not more than a few hundred thousand bytes.

**Late Policy**

You can submit this project late by one week for a 20 point penalty. Look at the grading
form to see the potential impact of bugs before electing this option. Note that you should submit
something even if the final version still has bugs.
Extra Credit (20 Points)

Extra credit will be given for fully supporting the synchronization primitives defined below.

- **int uth_lock( uth_lock_t L )**
  
  If another thread has lock L, put the current thread on a sleep queue for the lock; otherwise grant the ME (Mutual Exclusion) lock L to the current thread. Return 0 if the call is valid; -1 otherwise.

- **int uth_unlock( uth_lock_t L )**
  
  Give the ME lock L to the next waiting thread if there is one; otherwise release the lock. Return 0 if the current thread holds lock L; -1 otherwise.

- **void uth_cvwait( uth_cv_t *V, uth_lock_t L )**
  
  Put the current thread to sleep until a thread calls uth_cvsig. Return 0 if the call is valid; -1 otherwise.

- **void uth_cvsig( uth_cv_t *V )**
  
  Wakeup the thread that has waited the longest on V. Return 0 if the call is valid; -1 otherwise.

- **void uth_cvbcast( uth_cv_t *V )**
  
  Wake up all threads waiting on V. Return 0 if the call is valid; -1 otherwise.

- **int uth_lock_make( uth_lock_t L )**
  
  Initialize L to be a new ME lock object. Return 0 if the call is valid; -1 otherwise.

- **int uth_lock_free( uth_lock_t L )**
  
  End the use of L as a ME lock. Return 0 if the call is valid; -1 otherwise.

- **void uth_cv_make( uth_cv_t *V )**
  
  Initialize V to be a new CV (condition variable) object. Return 0 if the call is valid; -1 otherwise.

- **void uth_cv_free( uth_cv_t *V )**
  
  End the use of V as a CV object. Return 0 if the call is valid; -1 otherwise.

**Documentation**

Full extra credit can only be obtained if the features are well tested and documented. The documentation template contains a section at the end for documenting the extra credit features.