CSCI5103
Project 1 Writeup
David Ma (maxxx818) and Andrew Walker (walk0655)

Overall Architecture

We use three queues to track the threads as they go about their various stages of execution. Any time a new thread is created, we make its context and add it to the ready queue. The only exception to this is the main thread, which doesn't need to have a context made. The scheduler periodically pops the next thread from the ready queue and loads its context, thus switching between threads. Each function is lightly documented, so the control flow of each function should be relatively easy to follow.

Suspending and resuming are quite simple, but join is slightly more complicated and bears a bit of explanation. Any time a thread joins on another one, it checks to see if the other thread is in the finished queue. If it is, the first thread simply modifies its parameter value and returns. If the other thread hasn't finished, the first thread is moved to the block queue. When the other thread finishes, its thread_exit function moves the first thread back to the ready queue. Because only threads in the ready queue are periodically run by the scheduler, threads in the block queue are effectively paused.

Tests

Details about each test can be found in the README.md. Every function is called at least once throughout the various tests.

Additional Questions

a. What (additional) assumptions did you make?
    i. We assume usec is an integer value -- the interrupt timer cannot run faster than 1 us.
    ii. The user code does not create more than MAX_THREAD_NUM=100 threads, or we throw an error.
    iii. The user code does not require a stack size of more than STACK_SIZE=4096 bytes.
    iv. We assume user code doesn't use or depend on SIGVTALRM.
    v. We assume the user resumes any threads they suspend, since the code doesn't automatically resume any threads. For example, if they suspend a thread that the main thread is waiting on, it might produce an error.
    vi. Users join on every thread created.
b. Did you add any customized APIs? What functionalities do they provide?

The vast majority of our additional functions on top of the uthread library are just helper functions used to manipulate the three queues that track the various threads. They're all relatively self-explanatory, although there's a bit of documentation about them in the code. In general, they just remove elements from the queues, add elements to the queues, and find various elements in the queues.

```
int uthread_init(int quantum_usecs)
  /* Initialize the thread library */
  // Return 0 on success, -1 on failure

int uthread_create(void* (*start_routine)(void*), void* arg)
  /* Create a new thread whose entry point is start_routine */
  // Return new thread ID on success, -1 on failure

int uthread_join(int tid, void **retval)
  /* Join a thread */
  // Return 0 on success, -1 on failure

int uthread_yield(void)
  /* yields currently running thread for another ready thread*/
  // Return 0 on success, -1 on failure

void uthread_exit(void *retval)
  /* Terminate this thread */
  // Does not return to caller. If this is the main thread,
exit the program

int uthread_suspend(int tid)
  /* Suspend a thread */
  // Return 0 on success, -1 on failure

int uthread_resume(int tid)
  /* Resume a thread */
  // Return 0 on success, -1 on failure

int uthread_self()
  /* Get the id of the calling thread */
  // Return the thread ID

THE FOLLOWING FUNCTIONS ARE NOT EXPOSED TO THE END USER

/* Get the total number of library quantums (times the quantum
has been set) */
// Return the total library quantum set count
int uthread_get_total_quantums();
/* Get the number of thread quantums (times the quantum has
been set for this thread) */
// Return the thread quantum set count
```

```
int uthread_get_quantums(int tid);
void startInterruptTimer(int quantum_usecs); // starts the
interrupt timer that signals SIGVTALRM every quantum_usecs
void disableInterrupts(); // disables the interrupt timer.
Intended to protect API code from being interrupted
void enableInterrupts(); // reenables the interrupt timer.
Intended to allow user code to be interrupted again
TCB* getThread(int tid); // get the TCB specified by tid
void addToReadyQueue(TCB *tcb); // add a TCB to the ready queue
TCB* popFromReadyQueue(); // pop the next TCB from the ready
queue
int removeFromReadyQueue(int tid); // remove the TCB specified
by tid from the ready queue
void addToBlockQueue(TCB *tcb, int waiting_for_tid); // add a
specified block_queue_entry to the block queue
join_queue_entry_t* popFromBlockQueue(); // pop the next
element from the block queue
int removeFromBlockQueue(int tid); // remove the specified
block queue entry by its TCB tid
void addToFinishedQueue(TCB *tcb, void *result); // add a
finished_queue_entry to the entry with its return value
finished_queue_entry_t* popFromFinishedQueue(); // pop the next
element from the finished_queue
int removeFromFinishedQueue(int tid); // remove an element from
the finished queue based on its TCB tid
bool isReady(int tid); // checks to see if a TCB with tid is in
the ready queue
bool isBlocked(int tid); // checks to see if a TCB with tid is
in the join queu
join_queue_entry_t* getBlocked(int tid); // gets the element
from the join queeu with TCB tid
bool isFinished(int tid); //function to check if tid is in the
finished queue
finished_queue_entry_t* getFinished(int tid);//" to get an
element of the finished queue via its tid
bool hasWaiter(int tid); //see if there is an entry in the
block queue with waiter tid
join_queue_entry_t* getWaiter(int tid); //return the queue
entry that is waiting on tid
int getsize(); // prints the size of the ready queue
void showQueues(); // prints the tid of all elements in each
queue
```

c. How did your library pass input/output parameters to a thread entry function? What must makecontext, do “under the hood” to invoke the new thread execution?
    i. It uses pointers to pass input/output parameters throughout the functions. For example, u_thread_join changes retval based on the return values of the threads.
    ii. Makecontext allocates an object that represents a parent “context”, where “context” is the parent’s allocated stack space, the stack pointer, the program counter. Makecontext paired with getcontext and setcontext allows for contexts to be swapped in and out as necessary, effectively pausing performance.

d. How do different lengths of time slice affect performance?

On apollo.cselabs.umn.edu, we did 4 trials each of ./uthread-demo 1000000000 16 1 and ./uthread-demo 1000000000 16 1000. The average times are provided below.

uthread-demo	usec = 1	usec = 1000
Real time	19.44	21.24175
User time	17.49075	17.46575
Sys time	0.224	0.4195

It would appear in terms of user time they are pretty close, but in terms of real time and system time it’s a bit longer. It’s hard to see any really big differences here, though, since the variance on all these numbers was pretty large.

We also tried the same procedure on ./uthread-solution-exe to see if there are any discrepancies between performance with varying usec time, and we see similar results.

uthread-solution-exe	usec = 1	usec = 1000
Real time	21.68725	20.1005
User time	17.627	17.45375
Sys time	0.6705	0.42125

In this case, though, real time and sys time are both lower with a longer quantum, but again not by much given the variance of these measurements. It appears different lengths of time slice do not impact performance.

e. What are the critical sections your code has protected against interruptions and why?
    i. Any time we modify or reference a shared data structure, we have interrupts disabled. This is to prevent any race conditions from occurring. Any time we return to user code, we have interrupts enabled.
f. If you have implemented a more advanced scheduling algorithm for extra credit, describe it.
    i. We decided to implement a priority queue that assigns a higher priority to threads that have had fewer quantas of CPU time. It is essentially a min heap that also maintains ordering for threads that have the same quanta count. This implementation is found in <main_folder>/priority_queue