CS 518 Assignment 1: Implementing User Threads and Scheduler

Contributors: Sukumar Gaonkar (sg1425), Amogh Kulkarni (ark159), Vatsal Parikh (vp406)

ILab machine used:

Date: 14 October, 2018

Source code: https://github.com/Sukumar-Gaonkar/pthread_lib (currently private)

Instructions to run:

Design:

1) Thread Control Block (tcb) - Thread control block struct stores relevant information about each thread like thread id, the current state of the thread, pointer to the next thread, current thread context, run count of thread (number of times thread has been run by scheduler), thread priority, and return value of the thread denoting whether the thread has completed execution.

```
typedef struct threadControlBlock {
        my_pthread_t tid;
        struct threadControlBlock *next;
        ucontext_t ucontext;
        enum state state;
        uint run_count;
        void *return_val;
        struct tcb_queue *tcb_wait_queue;
        uint priority;
} tcb;
```

2) Thread Control Block Queue (tcb_list) - We have three types of queues in our design.
   a) wait_queue holds a list of all the threads that are ready to be executed and entered into the scheduler.
   b) priority_queue represents multilevel priority queue of the scheduler which schedules threads for execution.
   c) m_wait_queue is the mutex queue which holds locks for various critical sections. Any thread trying to access the same critical section is put into the mutex waiting queue for that mutex lock. When the mutex lock is released for a thread, the thread waiting for that particular lock is given the lock in FIFO manner.

3) Handling critical sections – A global variable called SYS_MODE is initialized with 0. Whenever an operation enters critical section like thread creation, thread yield, thread exit, thread join, and mutex functions, SYS_MODE is set to 1. When the operation exits critical section, SYS_MODE is set to 0. Signal_handler calls pthread_yield if SYS_MODE is 1 else returns.

Operation:

1) Expected execution cycle -
2) pthread implementation - The following four functions are created to handle thread operations.
   a) my_pthread_create function generates tid, and then calls make_scheduler to initialize the scheduler if it is uninitialized. It then gets current context (using getcontext), makes new thread context (using makecontext), and points the uc_link of newly created thread to the previous current context. The newly created thread is then enqueued
   b) my_pthread_yield calls make_scheduler to initialize the scheduler if it is uninitialized. It then voluntarily hands over CPU to another user thread and resets the timer. Yielding thread also swaps context with the next thread to be run.
   c) my_pthread_exit calls make_scheduler to initialize the scheduler if it is uninitialized. It then checks if the thread is already terminated, and simply prints if that's the case. Otherwise, puts the running thread in wait_queue, deallocates memory of this thread from wait_queue by pointing to the next thread and using dequeue, and finally calls scheduler to execute next thread by using pthread_yield.
   d) my_pthread_join calls make_scheduler to initialize the scheduler if it is uninitialized. It then checks the thread's status (terminated or running). Sets the thread's state to WAITING if thread is running, puts the thread in wait_queue, and calls pthread_yield to execute next thread.

3) mutex implementation – The following four functions are created to handle concurrency.
   a) my_pthread_mutex_init initializes the mutex struct and sets the lock to 0.
   b) my_pthread_mutex_lock first asserts that the mutex is initialized. If thread asking for mutex lock is already set to 1, the thread is put in m_wait_queue and the thread yields. If mutex lock is set to 0, it first checks if there are any threads in m_wait_queue requesting that lock. If no, requesting thread is given that lock and the thread starts running and enters critical section. If there is a m_wait_queue,
   c) my_pthread_mutex_unlock
   d) my_pthread_mutex_destroy

Scheduling:

1) Design - Scheduler struct stores multilevel priority queue and information of the currently running thread. MUTEX??? We have 20 LEVELS for our priority queue with 0 being the highest and 19 being the lowest. The TIME_QUANTUM for each running thread is set to 25 microseconds, and the maintenance cycle is called after every 50 time quanta.
2) Initialization -
3) Timer and signal handler -
4) Multilevel priority queue -
5) Maintenance cycle –

Testing and Analysis: