CS 518 Assignment 1: Implementing User Threads and Scheduler

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

Date: 14 October, 2018

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

Instructions to run:

- We made no changes to the makefile in our project. Should run from the benchmark by including the my_pthread.c file.

Overview:

In this assignment, we have designed user threads simulating the POSIX library (pthread API). Our implementation makes use of several structs like tcb, tcb_list, my_pthread_t, my_scheduler_t and creates a scheduler in round-robin fashion. We have also implemented a multi-level feedback queue which promotes and demotes user threads based on their priorities which also varies the time-quantum assigned during the execution cycle. The threads also have the feature to request mutexes for thread synchronization which makes use of struct my_pthread_mutex_t on which several functions like my_pthread_mutex_lock and my_pthread_mutex_unlock etc. thus creating a fully-featured user thread scheduler.

Design:

1) Thread Control Block (tcb) - Thread control block struct stores the 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, current timeslice and return value of the thread if the thread has completed execution.

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

2) Thread Control Block Queue (tcb_list) – The struct maintains start and end pointers to our list of TCBs used in the creation of several queues like priority_queue in scheduler, wait_queue in tcb amd m_wait_queue in mutex.

```
typedef struct tcb_queue {
    tcb *start;
    tcb *end;
} tcb_list;
```

3) Handling critical sections – A global variable called SYS_MODE which is initialized with 0. Whenever operations enter critical section like thread creation, thread yield, thread exit, thread join, and in mutex functions, SYS_MODE is set to 1. When the operation exits critical section, SYS_MODE is set to 0.

4) Handling return values from thread – Our current implementation stores tcb values after the thread is terminated. If a thread calls join on another that has been terminated, we return this stored value which is currently housed in our tcb. An alternate design mechanism for this could be storing tids in my_pthread_return_values struct and return values from this struct when requested by a thread by comparing its corresponding tid.

Operation:

1) **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 the scheduler context. The newly created thread is then enqueued in the priority queue with the highest priority 0.
   b) **my_pthread_exit** calls make_scheduler() to initialize the scheduler if it is uninitialized. It then checks if the thread is already terminated and sets the value_ptr to the return value stored in inside our tcb struct. Otherwise, puts the running thread in scheduler's priority_queue and dequeue's all waiting threads in the current running thread. Ultimately calls the yield function to hand control to the scheduler.
   c) **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.
   d) **my_pthread_yield** performs the context switch from current running thread to the next selected thread to be exectued.

Sequence of events after thread creation:

- When a thread is running, and its time quantum ends, timer interrupt is generated, and signal handler is called.
- Signal handler then calls my_pthread_yield.
- my_pthread_yield performs a context switch by swapping the current context of the running thread to that of the next thread in the scheduler.
- The scheduler decides the running thread based on the feedback queue, traversing through each level and populating its running queue and swaps its context with the selected running thread's context.

2) **mutex implementation** – The following four functions are created to handle thread synchronization.
   a) **my_pthread_mutex_init** initializes the mutex struct and sets the initialized flag to 0.
   b) **my_pthread_mutex_lock** locks the user thread by setting it to 1 if its initialized else return -1.
   c) **my_pthread_mutex_unlock** checks if the mutex is not locked already, if it is already locked by the current thread it unlocks the mutex by setting lock to 0.
   d) **my_pthread_mutex_destroy** first checks if mutex is initialized. If not, it cannot be destroyed. Mutex can be destroyed in two cases: one, if owner is destroying the mutex. two, mutex is unlocked and no thread is waiting for it.

Scheduling:

1) **Design and maintenance cycle** - Scheduler struct stores multilevel feedback queue and information of the currently running thread and the running queue which is a queue of scheduled threads in the running threads. This struct also stores list of mutexes of type my_pthread_mutex_t. We have set LEVELS where we define the no. of levels in the feedback queue. The default TIME_QUANTUM for each running thread is set to 25 microseconds but is multiplied by a factor (i+1) where i is the priority of the queue.

```
typedef struct my_scheduler_t {
    tcb *running_queue;
    tcb *running_thread;
    tcb_list *priority_queue[LEVELS];
    my_pthread_mutex_t *mutex_list;
    thread_ret_val ret_vals;
} my_scheduler;
```

When scheduler initially runs, all threads from priority 0 are run in a round robin fashion. After execution of a cycle in a level we demote the priority of the threads by 1 making them go to a lower level. After all threads from running queue have been executed, maintenance cycle is called in the form of schd_maintenance().

In the first phase, maintenance cycle promotes all the threads in the feedback queue by 1 i.e. (i-1). Then we demote the process (i+1) in the running_queue as they have already have executed their time quantum. In the later process process we assign the new quanta to the threads in their respective levels and then rebuild the running_queue.

2) Priority inversion – Priority inversion is handled by defining certain important threads whose threshold cannot go below level = LEVELS/5. We defined important as threads that hold a mutex lock and if threads are waiting on it (someone is trying to join it). For such threads, we do not decrease its priority value beyond the threshold and the thread remains at the same priority level.

3) Initialization – Scheduler is initialized by creating context for main function using makecontext which is then enqueued at highest priority in priority queue. When make_scheduler is called, timer and signal handler are also initialized.

4) Multilevel feedback queue – This data structure consists of a linked list of queues with each level having a queue of type tcb_list and each queue consisting of several threads in the form of tcb. It is the queue that is used to improve our scheduler performance.