Group member's names: Michael Zhang, Andrew Cater
NetIDs: mbz27, ajc398

1. Logic

Running Threads:

When the user creates a pthread for the first time, the library creates an MLFQ structure no matter if the scheduler is set to round robin or MLFQ. The only difference is that when the scheduler is set to round-robin it only allocates the first level of the MLFQ and obviously if the scheduler is set to MFLQ it will allocate all four levels. The library then creates and adds the thread context to the end of the first level queue. When the user joins a thread the library will continually check if the joining thread has terminated or not. If the thread has not been terminated then the scheduler will be called to keep on running thread contexts until the joining thread has been terminated. When a thread is terminated all of the mutexes that have been locked by the thread will be unlocked and if there is a return value it will store it in the structure. After a thread terminates and has been joined the structure that stores the thread will be deallocated and the return value will be stored in the parameter if need be.

Mutexes:

When a user initializes a mutex the mutex will be added to a link list and will have a blocked thread list and a pointer that keeps track of the thread that locks the mutex. When a thread locks a mutex the mutex will check if the mutex is locked. If the mutex is locked the thread will be set to a blocked state and will be added to the blocked thread list. But if the mutex is unlocked the mutex will be set to a locked state and the tracking thread pointer will be set. When a thread is in a blocked state the scheduler will simply ignore that thread and pick a different thread to run. When a thread unlocks a mutex the mutex will check if there are any threads in the blocked thread list. If there are threads in the list the mutex will stay locked and the first thread in the list will be unblocked and set as the thread that locked the mutex. If there are no threads in the list the mutex will simply get unlocked and the thread tracker will be cleared. (the reason we track the thread that locks the mutex is that when a thread exits it should release all its threads so we can find the mutex by using the thread's id and comparing it to the tracked thread in the mutex)

pthread_create()

On the first call of pthread_create(), we will initialize and allocate the MLFQ structure that will either contain 1 level for round-robin or 4 levels for MLFQ. We will then create the context and link it to the pthread_exit() function and add it to the end of the first level of the MLFQ structure regardless if the scheduler is set to round robin or MLFQ. (since round-robin is essentially MLFQ with one level)

pthread_yield()

When calling the pthread_yield() function the timer's time will be stored and the timer will be disabled. It will then set the yieldThread flag to 1 to indicate to the scheduler that the current thread context is yielding and calls the scheduler afterward. The scheduler will then determine if the thread context needs to be either pushed to the end of its current queue (if it's round-robin or the timeslice was not used up) or demoted a level (if it's MLFQ and the thread has used up its entire timeslice)

pthread_exit()

When calling the pthread_exit() function the timer is disabled since the thread is finished running. All the mutexes that the thread context locked will be unlocked, the retValue will be saved if there is any, and the status of the thread will be set to TERMINATED flagging the thread as finished. Then it will swap out the context with the scheduler context to finish the scheduler call.

pthread_join()

The pthread_join() function will check if the thread has finished running (ie. threads with status TERMINATED) and if the thread has not finished it will call the scheduler to run the next thread and will continue to check if the joining thread has finished. When the thread is finished the return value will be set if the pthread_join requires it and the joined thread will be deallocated from the queue.

pthread_mutex_init()

When calling the pthread_mutex_init() function we allocated a link list to store all of the mutexes and add the mutex to the list. In the mutex structure, we keep track of the mutexID, the status of the mutex (whether it is locked/unlocked), the thread context that locked the mutex, and a list of thread contexts that are blocked because of the locked mutex.

pthread_mutex_lock()

The pthread_mutex_lock() function will store the current timer's time (needed later if the timer needs to resume) and disables the timer. If the mutex in the link list is unlocked then it will lock the mutex and will resume the timer and the current thread context resumes. If the mutex in the link list is already locked by a different thread then the thread is added to the block list in the mutex structure and the status of the thread context is set to BLOCKED then it will yield the thread since the thread is blocked by a mutex lock.

pthread_mutex_unlock()

The pthread_mutex_unlock() function will store the current timer's time (needed later if the timer needs to resume) and disables the timer. If the mutex has other threads in the blocked list it will unblock the first thread in the list and set that thread to a READY state (the mutex will still be locked). If there are no other threads in the mutex blocked list then the mutex status is set to UNLOCKED and is free to be locked by other threads. After unlocking the mutex or unlocking a thread in the blocked list the timer is resumed and the current thread context resumes.

pthread_mutex_destroy()

The pthread_mutex_destroy() function will unblock all blocked threads in the blocked list in the mutex structure and removes and deallocate the structure that stores the mutex from the mutex list.

sched_rr() - Round Robin

The scheduler for Round Robin will handle yields and the scheduling of threads to run. If the yieldThread flag is set the scheduler will push the current running thread to the back of the queue. If the yieldThread flag is not set the scheduler will pick the first thread in the queue that is in a READY state to run. After

picking a thread the scheduler will set the timer and swap out the context to the thread context and stores the current context of the scheduler to return to after running.

sched_mlfq() - MLFQ

The scheduler for MLFQ again will handle yields and the scheduling of threads to run. If the yieldThread flag is set the scheduler will check if the thread has taken up its entire TIMESLICE. If the thread has taken up its entire TIMESLICE then the thread will be demoted a level and sent to the back of the queue in that level. If the thread has not taken up its entire TIMESLICE then the thread is simply pushed to the back of the queue. If the yieldThread flag is not set then the scheduler will pick the topmost thread in the multi-level queue structure that is in a READY state to run. After picking a thread the scheduler will set the timer and swap out the context to the thread context and stores the current context of the scheduler to return to after running.

2. Benchmarks

external_cal.c:

- pthread lib implementation:
  - Number of threads - 4
    - Time: 4524 milli-seconds
  - Number of threads - 100
    - Time: 4947 milli-seconds

- Round Robin:
  - rpthread user-level implementation:
    - Number of threads - 4
      - Time: 44255 milli-seconds
    - Number of threads - 100
      - Time: 45450 milli-seconds

- MLFQ:
  - rpthread user-level implementation:
    - Number of threads - 4
      - Time: 44619 milli-seconds
    - Number of threads - 100
      - Time: 45684 milli-seconds

parallel_cal.c:

- pthread lib implementation:
  - Number of threads - 4
    - Time: 979 milli-seconds
  - Number of threads - 100
    - Time: 202 milli-seconds

- Round Robin:

- rpthread user-level implementation:
  - Number of threads - 4
    - Time: 3425 milli-seconds
  - Number of threads - 100
    - Time: 3401 milli-seconds

- MLFQ:
  - rpthread user-level implementation:
    - Number of threads - 4
      - Time: 4812 milli-seconds
    - Number of threads - 100
      - Time: 3410 milli-seconds

vector_multiply.c:

- pthread lib implementation:
  - Number of threads - 4
    - Time: 328 milli-seconds
  - Number of threads - 100
    - Time: 638 milli-seconds

- Round Robin:
  - rpthread user-level implementation:
    - Number of threads - 4
      - Time: 10574 milli-seconds
    - Number of threads - 100
      - Time: 10615 milli-seconds

- MLFQ:
  - rpthread user-level implementation:
    - Number of threads - 4
      - Time: 10238 milli-seconds
    - Number of threads - 100
      - Time: 11013 milli-seconds

3. Analysis

Comparing my user-level implementation of pthread to the pthread lib implementation it is obvious to say that my user-level implementation is quite a bit worse in terms of time taken to finish any of the benchmark programs. This is especially apparent when running the external_cal.c program where my user-level implementation of pthread took 40 seconds longer than the pthread lib implementation. This is probably because the pthread lib implementation is implemented in the kernel and is running threads in parallel, unlike my user-level implementation. The longer times might also be due to my implementation of the pthread library since I was using link lists to search through all the threads instead of hashtables which I think would slightly improve my times. I also think that if I stored the TERMINATED threads,

the BLOCKED threads, and the READY threads in different data structures it would speed up the search time of finding a thread in a specific state.

Though comparing my user-level implementation of the round-robin scheduler to my implementation of the MLFQ scheduler it isn't quite obvious which scheduler is better. The two schedulers seem to get practically the same times but in most cases, the MLFQ scheduler gets a slightly higher time but I think it's mostly due to the data structures I used to construct the MLFQ scheduler.