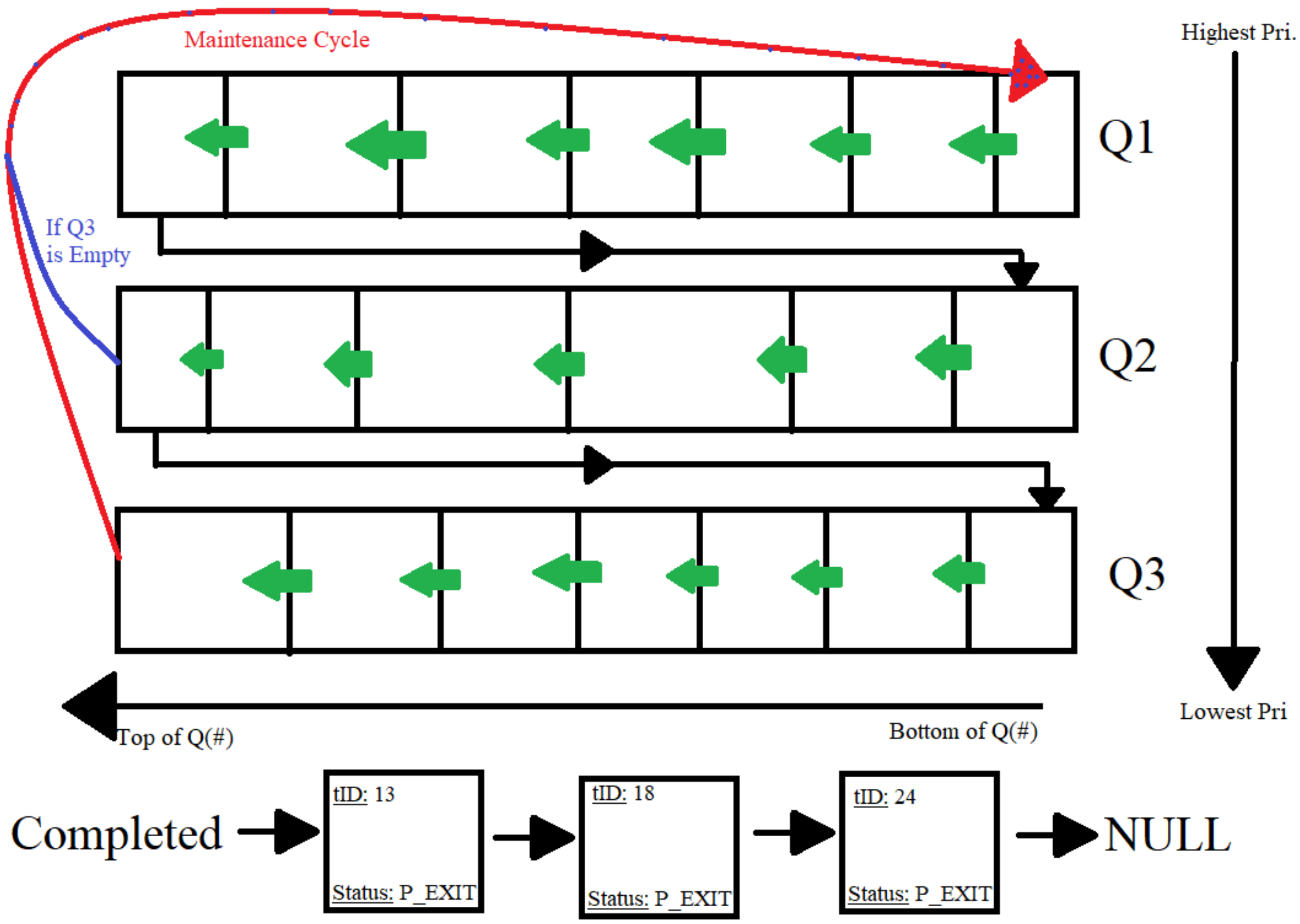
USR-Thread-Lib

Names: Liam Davies, Kevin Lee, Brian Ellsworth
NetIDs: lmd312, kjl156, bje40
ilab machine: factory.cs.rutgers.edu

Multi-Level Feedback Queue Scheduling Plan:

Our scheduler implements a Multi-Level Feedback Queue that uses a maintenance cycle to prevent starvation. There are three queues that we've used. Queue1, Queue2, and Queue3. These queues each have different priorities as well as different time quanta. Queue 1 will have the highest priority, and Queue 3 will have the lowest priority. This can be seen in the picture below.



So when dealing with previously running processes, we'll check what queue and status the thread has and decide where it will go based on those parameters. As well as having different priorities, these queues have different time slices. Queue 1 has 25 MSECS, Queue 2 has 50 MSECS, and Queue 3 has 100 MSECS. We chose the numbers of queues and time slices after testing out time slices and seeing which one worked the fastest. We also have a linked list of *completed* processes (with P_EXIT status) for the threads that have been run to completion. This saves the scheduler from having to look at processes that are done.

Thread Statuses:

The thread status was a viable part of our scheduler. Listed below is a general summary of what the scheduler will do and how a certain thread with this status will behave. * **P_RUN** - The thread ran for the full allocated time * *Upon Choosing:* * N/A * *Upon Putting Away:* * Place into one *Q#* lower than from where it was picked. (If from *Q3*, place back into *Q3*)

* **P_YIELD** - The thread stopped early due to a yield() * *Upon Choosing:* * **SHOULD NEVER HAPPEN.** * *Upon Putting Away:* * Change status to **P_RUN** * Place into the *same Q#* from where it was picked.

* **P_WAIT_M** - The thread is waiting on a mutex * *Upon Choosing:* * Check to see if the **mutex** it is waiting on is unlocked * If it is unlocked, run this thread. Otherwise, place into the *same Q#* from where it was picked. * Proceed to check the next thing in the current *Q#*. * Make sure you don't cycle the queue by storing the first node and checking against the number of runs. * *Upon Putting Away:* * Place into the **same** *Q#* from where it was picked.

* **P_WAIT_T** - The thread is waiting for another thread * *Upon Choosing:* * Check to see if the **thread** it is waiting on is the *completed* linked list * If it is, run this thread. Otherwise, place into the *same Q#* from where it was picked. * Proceed to check the next thing in the current *Q#*. * Make sure you don't cycle the queue by storing the first node and checking against the number of runs. * *Upon Putting Away:* * Place into the **same** *Q#* from where it was picked.

* **P_EXIT** - The thread has finished and run exit() * *Upon Choosing:* * **SHOULD NEVER HAPPEN.** * *Upon Putting Away:* * Instead of placing back into the MLFQ, place into a separate linked list "*completed*" where the thread wait()ing on it can find it, take its return value, and delete it.

Scheduler:

1. Set up sigmask.
2. Decide which thread will run next using a function checkQueue(). We first check the first queue. If that queue is empty, we get the second queue. If the second queue is empty, we get the third queue.
    - In checkQueue(), we check the status of the thread. What happens next depends on the status of the thread, such as setting the thread up to go next, as seen above.
3. Determine where the currently running thread will go.
    - Check above to see more of *Thread Status* and what they do.
4. Perform the *Maintenance Cycle*. (as seen below)
5. Prepare to swap contexts
6. Start the timer for the next thread.
7. Drop sigmask.
8. Swap contexts.

Maintenance Cycle:

Our maintenance cycle occurs in the scheduler method. After the scheduler runs 10 times then the scheduler will dequeue whatever is at the lowest queue and enqueue it back to the top. This prevents threads from being starved of time if they are in lower queues. 1. Pop the top process of *Q3*. If *Q3* has none, then pop the top process of *Q2*. If *Q2* has none then skip the remaining steps. 2. Call this popped process **mProc**. 2. Take **mProc** and place it into *Q1*. This maintain cycle, as well as the certain quanta for each priority queues, successfully prevented any starvation.

Test and Test Results:

We had several tests in order to check if our p_thread library did what it was supposed to. Below are some of the tests conducted.

Test1:

Assesses: * my_pthread_t * my_pthread_exit * my_pthread_create (and its passing arguments) and p_thread_join

```
void *f1(void * i){
    int * iptr = (int *) i;
    *iptr += 1;
    my_pthread_exit(NULL);
}

void * f2(void * i){
    int * iptr = (int *) i;
    *iptr += 2;
    my_pthread_exit(NULL);
}

void * f3(void * i){
    int * iptr = (int *) i;
    *iptr += 3;
    my_pthread_exit(NULL);
}

void main(int argc, char ** argv){
    my_pthread_t t1, t2, t3, t4, t5;
    void * (*f1ptr)(void *) = f1;
    void * (*f2ptr)(void *) = f2;
    void * (*f3ptr)(void *) = f3;

    int i1 = 0, i2 = 0, i3 = 0, i4 = 0, i5 = 0;
    my_pthread_create(&t1, NULL, f1ptr, &i1);
    my_pthread_create(&t2, NULL, f2ptr, &i2);
    my_pthread_create(&t3, NULL, f3ptr, &i3);
    my_pthread_create(&t4, NULL, f1ptr, &i4);
    my_pthread_create(&t5, NULL, f2ptr, &i5);

    void * ret;
    my_pthread_join(t1, &ret);
    my_pthread_join(t2, &ret);
    my_pthread_join(t3, &ret);
    my_pthread_join(t4, &ret);
    my_pthread_join(t5, &ret);
```

```
        printf("Complete.\n");
        printf("i1: %d, i2: %d, i3: %d, i4: %d, i5: %d\n", i1, i2, i3, i4, i5);
    }
```

This test was successful, as it printed out
Complete.
il: 1, i2: 2, i3: 3, i4: 1, i5: 2

Test2:

Test 2 tests the following: * argument passing * basic locks * my_pthread_yield()

```
typedef struct largerData {
   int d1;
   int d2;
   int d3;
   my_pthread_mutex_t m;
} ldata;

void * f1(void * i){
   printf("f1 start.\n");
   ldata * d = i;
   my_pthread_yield();
   my_pthread_mutex_lock(&((*d).m));
   my_pthread_yield();
   my_pthread_yield();
   my_pthread_yield();
   my_pthread_mutex_unlock(&((*d).m));
   printf("f1 end.\n");
   my_pthread_exit(NULL);
   return NULL;
}

void * f2(void * i){
   printf("f2 start.\n");
   ldata * d = i;
   my_pthread_mutex_lock(&((*d).m));
   my_pthread_yield();
   my_pthread_yield();
   my_pthread_mutex_unlock(&((*d).m));
   printf("f2 end.\n");
   my_pthread_exit(NULL);
   return NULL;
}

void * f3(void * i){
   printf("f3 start.\n");
   ldata * d = i;
   my_pthread_mutex_lock(&((*d).m));
   my_pthread_mutex_unlock(&((*d).m));
   printf("f3 end.\n");
   my_pthread_exit(NULL);
   return NULL;
}
```

```
void main(int argc, char ** argv){
   printf("main start.\n");
   my_pthread_t t1, t2, t3;
   void * (*f1ptr)(void *) = f1;
   void * (*f2ptr)(void *) = f2;
   void * (*f3ptr)(void *) = f3;
   ldata * data = malloc(sizeof(ldata));
   my_pthread_mutex_init(&((*data).m), NULL);
   (*data).d1 = 27;

   // Make sure no crash on double lock/unlock - w/ 1 thread
   my_pthread_mutex_lock(&((*data).m));
   if(my_pthread_mutex_lock(&((*data).m)) == 0){
      printf("ERROR: Mutex double lock - 1 thread - no error.\n");
   }

   my_pthread_mutex_unlock(&((*data).m));
   my_pthread_create(&t1, NULL, f1ptr, data);
   my_pthread_create(&t2, NULL, f2ptr, data);
   my_pthread_yield();
   my_pthread_yield();
   my_pthread_yield();
   my_pthread_create(&t3, NULL, f3ptr, data);

   void * ret;
   my_pthread_join(t2, &ret);
   my_pthread_join(t1, &ret);
   my_pthread_join(t3, &ret);
   printf("main end.\n");
}
```

The second test printed out:
main start.
f1 start.
f2 start.
f1 end.
f3 start.
f2 end.
f3 end.
main end.

The second test was also successful, as there were no errors.

Test5:

The fifth test was used to see if we could create a thread within a thread.

```
typedef struct data_test {
   int d1;
   int d2;
} data;
```

```
void * createe(void * i) {
   printf("createe begins.\n");

   printf("createe creates ret struct.\n");
   data * ret = malloc(sizeof(data));
   (*ret).d1 = 5;
   (*ret).d2 = 20;

   printf("createe ends.\n");
   my_pthread_exit(ret);
   return NULL;
}

void * t_creater(void * i) {
   printf("t_creater begins.\n");
   my_pthread_t t2;

   printf("t_creater creates createe.\n");
   my_pthread_create(&t2, NULL, createe, NULL);

   void *ret;
   printf("t_creater joining on createe\n");

   my_pthread_join(t2, &ret);
   printf("t_creater ends.\n");
   my_pthread_exit(ret);
   return NULL;
}
void main(int argc, char ** argv) {
   printf("Main begins.\n");
   my_pthread_t t1;

   printf("Main creates t_creater.\n");
   my_pthread_create(&t1, NULL, t_creater, NULL);

   void * r;
   printf("Main joining on t_creater\n");
   my_pthread_join(t1, &r);

   data * ret = r;
   printf("Main ends, ret values -> d1 = %d, d2 = %d.\n", (*ret).d1, (*ret).d2);
}
```

The 5th test was successful. It printed out:

Main begins.

Main creates t_creater.

t_creater begins.

t_creater creates createe.

Main joining on t_creater

createe begins.

createe creates ret struct.

createe ends.

t_creater joining on createe

t_creater ends.
Main ends, ret values -> d1 = 5, d2 = 20.

Test4:

Test 4 conducts a stress tests to the mutex lock We did this by locking and unlocking the lock that contains a for loop that is adding 1 to 10000000 to a shared variable.

```
my_pthread_mutex_t lock;
int sum;

void *add2(void *param) {
   printf("In the add method #2\n");
   my_pthread_mutex_lock(&lock);
   int i;
   for (i = 0; i < 10000000; i++) {
      sum = sum + 1;
   }
   my_pthread_mutex_unlock(&lock);
   printf("sum #2:%d\n",sum);
   printf("exited #2\n");
   my_pthread_exit(NULL);
   return NULL;
}

void *add1(void *param) {
   printf("In the add method #1\n");
   my_pthread_mutex_lock(&lock);
   int i;
   for (i = 0; i < 10000000; i++) {
      sum = sum + 1;
   }
   my_pthread_mutex_unlock(&lock);
   printf("sum #1:%d\n",sum);
   printf("exited #1\n");
   my_pthread_exit(NULL);
   return NULL;
}

void main(int argc, char ** argv){
   my_pthread_t tid1, tid2;
   my_pthread_mutex_init(&lock, NULL);

   printf("Starting\n");
   my_pthread_create(&tid1, NULL, add1, NULL);
   my_pthread_create(&tid2, NULL, add2, NULL);

   void *ret;
   my_pthread_join(tid2, &ret);
   my_pthread_join(tid1, &ret);
   my_pthread_mutex_destroy(&lock);
   printf("Ended, %d\n",sum);
}
```

Test four was succesful, as it printed out:
Starting
In the add method #1
sum #1: 10000000
exited #1
In the add method #2
sum #2: 20000000
exited #2
Ended, 20000000

Test7:

Test 7 was also a stress test. We stressed out the scheduler by creating several threads (14) and having a mutex lock that locks 1000 times.

```
int data = 0;
int limit = 1000;
my_pthread_mutex_t lock;
void *print(void *param) {
   printf("Thread Num - %d\n",*(int *)param);
   int x = 0;
   my_pthread_mutex_lock(&lock);
   for(x = 0; x < limit; x++){
      data ++;
   }
   my_pthread_mutex_unlock(&lock);
   my_pthread_exit(NULL);
   return NULL;
}

void main(int argc, char ** argv){
   my_pthread_t tid0, tid1, tid2, tid3, tid4, tid5, tid6, tid7, tid8, tid9, tid10,
tid11, tid12, tid13;
   my_pthread_mutex_init(&lock, NULL);
   my_pthread_create(&tid0, NULL, print, &data);
   my_pthread_create(&tid1, NULL, print, &data);
   my_pthread_create(&tid2, NULL, print, &data);
   my_pthread_create(&tid3, NULL, print, &data);
   my_pthread_create(&tid4, NULL, print, &data);
   my_pthread_create(&tid5, NULL, print, &data);
   my_pthread_create(&tid6, NULL, print, &data);
   my_pthread_create(&tid7, NULL, print, &data);
   my_pthread_create(&tid8, NULL, print, &data);
   my_pthread_create(&tid9, NULL, print, &data);
   my_pthread_create(&tid10, NULL, print, &data);
   my_pthread_create(&tid11, NULL, print, &data);
   my_pthread_create(&tid12, NULL, print, &data);
   my_pthread_create(&tid13, NULL, print, &data);

   void * ret;
   my_pthread_join(tid0, &ret);
   my_pthread_join(tid1, &ret);
   my_pthread_join(tid2, &ret);
```

```
    my_pthread_join(tid3, &ret);
    my_pthread_join(tid4, &ret);
    my_pthread_join(tid5, &ret);
    my_pthread_join(tid6, &ret);
    my_pthread_join(tid7, &ret);
    my_pthread_join(tid9, &ret);
    my_pthread_join(tid8, &ret);
    my_pthread_join(tid10, &ret);
    my_pthread_join(tid11, &ret);
    my_pthread_join(tid12, &ret);
    my_pthread_join(tid13, &ret);
    my_pthread_mutex_destroy(&lock);
    printf("Can you hear me?\n");
}
```

The 7th test was successful as well. It printed out:
Thread Num - 0
Thread Num - 1000
Thread Num - 2000
Thread Num - 3000
Thread Num - 4000
Thread Num - 5000
Thread Num - 6000
Thread Num - 7000
Thread Num - 8000
Thread Num - 9000
Thread Num - 10000
Thread Num - 11000
Thread Num - 12000
Thread Num - 13000
Can you hear me?

There were many other tests that we conducted to this thread library, and all of them successfully worked.

Testing Suite Compilation:

A suite of test programs are provided in the Tests directory. They are labeled "Test#" where # is an integer test number. These tests can be compiled by running the "make Test#" command. For example, "make Test5" will compile Test5. Make all will compile all tests.