Connor Aleksandrowicz cja142
Ryan Berardi rtb100

Thread Library

Thread Creation:

Thread Control Block (TCB):

The TCB is defined in our header file containing the following attributes:

- A unique thread identifier which we called **tid**
- An integer named **state** that will switch between RUNNING (0), SCHEDULED (1), and BLOCKED (2)
- A ucontext_t named **context** that is used to store the thread's context
- A void pointer to the threads stack, named **stack**
- An integer named **priority** that will distinguish which queue this thread belongs to when running MLFQ.
- A TCB pointer named **joined** that allows us to track which thread has called join on this thread.
- An integer **j** used to distinguish if this thread has called rpthread_join on another thread and is waiting for it to finish.
- A TCB pointer **next** that is used to form a linked list of nodes that will represent the queue.

All of these values are set in our methods **tcb_create** and **rpthread_create**. The reason for creating two different functions for creating a thread is to allow our main thread to be created in a manner that allows for more control of its attributes.

Thread Context:

Our thread context uses a size of **SIGSTKSZ** for it's stack. We set up our thread context in rpthread_create for normal threads, and in **initMain** for the main thread. Normal threads have their **uc_link** pointed to our global **term** context, which points the thread to rpthread_exit to allow all threads that finish their jobs to exit. Our main thread does not have a uc_link, because once this thread finishes the program will end and it's uc_link will be ignored. A global scheduler context is created in **initSched** which points to our scheduler function.

Runqueue:

We have multiple runqueues in our program. They are each outfitted with three attributes. A pointer **front** to the first thread in the list, a pointer **rear** to the last thread in the list, and an integer **qsize** to tell us how many threads are currently in the queue.

Thread Yield:

Our rpthread_yield function begins by disabling the timer for the current thread, so that it is not interrupted while yielding. The **state** is then changed to SCHEDULED and the thread is **enqueued** back into the runqueue. Once this has finished the **context is swapped** from the current thread's context to the scheduler context.

Thread Exit:

Our rpthread_exit function begins by disabling the timer for the current thread, so that it is not interrupted while exiting. We have a global integer array called **finished**, and once a thread reaches rpthread_exit we change it's index in this array to 1 so that we know the thread is finished. We then free the thread's **stack** and if the thread has been joined by another we change the other thread's **j** value from 1 to 0 and change its state to **SCHEDULED**. We can now **free the thread itself**, set our current thread pointer to NULL and return to the **scheduler context**.

Thread Join:

Our rpthread_join function begins by checking if the passed thread has already finished (by checking our **finished array**), in which case we can just return and allow the current thread to continue running. If this is not the case, then we search through the queue(s) for the thread who's tid is passed to join. If the thread is not found, we exit the program with the error "Thread not found". If the thread is found, we set the found thread's **joined** attribute to point to our current thread, which will wait for it to finish, and set our current thread's **j** attribute to 1 as well as changing its **state to BLOCKED**. While the passed thread is still running our current thread's **j** value will not be changed, so we enter a while loop waiting for the value to become 0 again. While this value is not 0 we **enqueue** the current thread and **swapcontext** to the **scheduler**. The **j** value and **state** will change once the passed thread reaches exit.

Thread Synchronization:

Mutex Initialization:

Our mutex contains two attributes. The first attribute being a TCB pointer to the current **owner** of the mutex. The second attribute being an integer **flag** that will alternate between 0 and 1. When the **flag** attribute is set to 0 the mutex is unlocked and when it is set to 1 it is locked. Our rpthread_mutex_init function takes in a pointer to a mutex. It begins by disabling the timer so that nothing is interrupted throughout the function. The **owner** attribute is set to NULL and the flag **attribute** is set to 0.

Mutex Lock:

Our rpthread_mutex_lock function begins by saving the context of the current running thread so we can come back to it later. The timer is then disabled to allow this function to finish without being interrupted. We then enter a while loop, using our **TestAndSet** function to attempt to lock the mutex. If it fails, the context is switched to the scheduler and once this thread begins running again it will pick up at the beginning of the function where we saved our context. If the **TestAndSet** function is successful the function will continue past the while loop and the **owner** of the mutex will be set to the current thread. The timer will start again and the function will return.

Mutex Unlock:

Our rpthread_mutex_unlock function begins by disabling the timer so that the function cannot be interrupted. The function will check if the thread attempting to unlock the mutex is the **owner**, if it is not it will print an error message and exit. If the current thread is the **owner** then the **owner** attribute will be set to NULL and the **flag** attribute will be set to 0.

Mutex Destroy:

Our rpthread_mutex_destroy function sets the **owner** attribute to NULL and **unlocks** the mutex by setting the flag attribute to 0. No memory deallocation is necessary because our mutex does not allocate any memory.

Scheduler

Round Robin:

Our Round Robin scheduler begins by checking if the current **qsize** is greater than 0. If it is, we begin by popping off the first thread in the queue. If the state of the thread is **BLOCKED** then we **enqueue** the thread and restart the scheduler function by using **setcontext**. If the thread is not blocked we change its state from **SCHEDULED** to **RUNNING**, begin our timer by calling **startTimer**, and **setcontext** to the **current popped thread's context**. Once a thread has finished its job it will switch to its uc_link context. This is defined as a global **term** context which links to **rpthread_exit**. Inside of rpthread_exit the thread is deallocated and the **context returns back to the scheduler.** Once the entire queue is empty, including the main thread, the scheduler will continue past the if condition and free the scheduler stack, the term stack, the main stack, and the main TCB pointer.

MLFQ:

Our MLFQ scheduler works by checking if the highest priority **qsize** is greater than 0, if it is it runs all of the threads in that priority level round robin style as described above. If the thread uses all of the allocated time slice our handler function is invoked and sets the priority of the thread down a level and re-enqueues the thread in it's proper priority level. If the thread is at the bottom level it's priority is then set back to the highest level then re-enqueued. If the highest priority level has a qsize of 0 we then check the next highest priority level's qsize and repeat the round robin style of scheduling for that priority level.

Benchmark Results

Round Robin:

- External_cal
  - ./external_cal 100
    - Runtime: 41463 microseconds
  - ./external_cal 50
    - Runtime: 42064 microseconds
  - ./external_cal 10
    - Runtime: 42400 microseconds
  - ./external_cal 1
    - Runtime: 42277 microseconds
- Parallel_cal
  - ./parallel_cal 100
    - Runtime: 2796 microseconds
  - ./parallel_cal 50
    - Runtime: 3012 microseconds
  - ./parallel_cal 10
    - Runtime: 2784 microseconds
  - ./parallel_cal 1
    - Runtime: 2744 microseconds
- Vector_multiply
  - ./vector_multiply 100
    - Runtime: 10141 microseconds
  - ./vector_multiply 50
    - Runtime: 10401 microseconds
  - ./vector_multiply 10
    - Runtime: 9969 microseconds
  - ./vector_multiply 1
    - Runtime: 10151 microseconds

MLFQ:

- External_cal
  - ./external_cal 100
    - Runtime: 42485 microseconds
  - ./external_cal 50
    - Runtime: 41961 microseconds
  - ./external_cal 10
    - Runtime: 41893 microseconds
  - ./external_cal 1
    - Runtime: 42189 microseconds
- Parallel_cal
  - ./parallel_cal 100
    - Runtime: 2767 microseconds
  - ./parallel_cal 50
    - Runtime: 2786 microseconds
  - ./parallel_cal 10
    - Runtime: 2771 microseconds
  - ./parallel_cal 1
    - Runtime: 2763 microseconds
- Vector_multiply
  - ./vector_multiply 100
    - Runtime: 10130 microseconds
  - ./vector_multiply 50
    - Runtime: 10411 microseconds
  - ./vector_multiply 10
    - Runtime: 9898 microseconds
  - ./vector_multiply 1
    - Runtime: 10023 microseconds

Pthread Library:

- External_cal
  - ./external_cal 100
    - Runtime: 6142 microseconds
  - ./external_cal 50
    - Runtime: 6401 microseconds
  - ./external_cal 10
    - Runtime: 4693 microseconds
  - ./external_cal 1
    - Runtime: 7612 microseconds
- Parallel_cal
  - ./parallel_cal 100
    - Runtime: 171 microseconds
  - ./parallel_cal 50
    - Runtime: 183 microseconds
  - ./parallel_cal 10
    - Runtime: 327 microseconds
  - ./parallel_cal 1
    - Runtime: 3219 microseconds
- Vector_multiply
  - ./vector_multiply 100
    - Runtime: 636 microseconds
  - ./vector_multiply 50
    - Runtime: 435 microseconds
  - ./vector_multiply 10
    - Runtime: 260 microseconds
  - ./vector_multiply 1
    - Runtime: 62 microseconds