**CS 416 Project 2: User-level Thread Library and Scheduler**

**Names:**

**Project Implementation Description**

To begin, our implementation contains the following core structs:

(Note that more detailed descriptions of these can be found in the code files and the README file.)

* **scheduler\_t** – Stores all scheduler data needed — the runqueue, main/scheduler context, pointer to the currently running thread, etc..
* **tcb** – Stores all data for the current thread — thread id, status, context, priority, etc…
* **runqueue\_t** – A multi-level queue data structure representing the run queue. My implementation is basically just an array of **queue\_t**, where **queue\_t** is just a linkedlist-like queue with linked nodes.

**1. Detailed Logic of Each API Function**

Below I will summarize our implementation of the thread library api and both types of schedulers. More specific details about the functions or structs can be found in the comments in the code file as well as the README.

**1.1. Thread Creation** (**worker\_create**)

**Definition:**

This function creates a new context for the thread with the passed in function and its arguments. This process includes:

1. Creating and initializing a new **tcb** with default parameters such as thread ID, priority, and status.
2. Allocating and creating a new **ucontext** for the thread using **makecontext**.
3. After the initialization process, the **tcb** is added to the runqueue with **sch\_schedule**

Note: when this function is called for the first time, **scheduler\_t** will be initialized, this is explained in more detail in the scheduler section below.

**Thread Function**

* Notice that instead of directly using **makecontext** on the **function** parameter that the user has passed in, I decided to create a wrapper – **worker\_wrapper\_t** – which wraps the original function and its parameters into a struct object.
* And then instead of passing in the original function to **makecontext**, a wrapper function: **thread\_function\_wrapper**, with **worker\_wrapper\_t** as its parameter is passed in, which then the original function will be called inside the wrapper function.
* This essentially allow us to add any initialization or additional logic code before and after the original function is ran. For example, when the thread has finished running, there are some additional scheduler logic that might need to be done before the thread actually ends, like changing the status of the thread, or swapping back to the scheduler.

**1.2. Thread Yield** (**worker\_yield**)

**Definition:**

This function voluntarily gives up the current thread’s execution cycle, I did this by simply changing the current thread’s status from RUNNING to READY.

After changing the status, we immediately swap to the scheduler context for it to schedule the next thread.

**1.3. Thread Exit** (**worker\_exit**)

**Definition:**

This function will terminate the currently running thread by simply changing the thread’s status to FINISHED and immediately swapping to the scheduler context, similar to yield.

**1.4. Thread Join** (**worker\_join**)

**Definition:**

This function will yield the current thread until a target thread terminates. I decided to use a **spin lock** to implement this, while this does waste quite a lot of cpu time, it reduces the total number of context switches, since each time the thread just gets yielded. Thus we chose to implement spinning instead of managing for instance, a queue for the threads waiting.

* The function will also pass the return value of the target thread when it finishes executing, the value is retrieved from return\_value, which is set after the thread terminates. Notice that the value will only be set if value\_ptr is an valid pointer.

**1.5. Thread Synchronization**

The core struct that we used is **worker\_mutex\_t**, which stores all data of a mutex, for instance, an atomic lock variable used to control the mutex, the id of the thread that owns this mutex, and a **queue\_t** containing all the blocked threads that are waiting.

**1.5.1. (worker\_mutex\_init)**

**Definition:**

This function initializes all the variables in the **worker\_mutex\_t** pointer passed in.

**1.5.2. (worker\_mutex\_lock)**

**Definition:**

I implemented the mutex lock using an atomic variable, by utilizing the GCC built-in atomic function **\_\_sync\_lock\_test\_and\_set** and setting the **locked** field inside **worker\_mutex\_t**. Now, there are two cases that can happen if a thread attempted to enter the critical section control by this mutex:

* If **locked** is 0, meaning that the mutex has not been locked by anyone, the we store the thread id of the current thread and enter the critical section.
* If **locked** is 1, meaning that the mutex is locked and the critical section is currently being access by some other thread, the current thread will now be set to a BLOCKED status and put into the blocked\_thread queue inside **worker\_mutex\_t**, and then yielded until the mutex becomes available.

**1.5.2. (worker\_mutex\_unlock)**

**Definition:**

This is the paired function to **worker\_mutex\_lock** that releases the mutex lock. Using the built-in GCC function \_\_sync\_lock\_release, we set **locked** to 0 thus releasing the mutex. All the BLOCKED threads inside the blocked queue will now be rescheduled and the next one that enter will be able to access the critical section.

**1.5.2. (worker\_mutex\_destroy)**

**Definition:**

This simply frees all the dynamic memory that was used for the mutex.

**2. Scheduler Implementation**

**Core Data Structure**

**scheduler\_t** – This is our core data structure that stores all the scheduler data needed globally, here is the list of fields:

* **run\_queue** – A **runqueue\_t** representing my scheduler's runqueue.
* **main\_context** – The main context
* **scheduler\_context** – The scheduler context
* **scheduler\_stack** – Pointer to the scheduler context stack
* **main\_thread** – This is a pointer to the main thread, stored here for easy access
* **current\_thread** – This is the current thread that is being executed/running
* **thread\_table** – this is a mapping that is used to find threads by their id, kinda waste space probably need better management

**Scheduler Initialization**

In the first ever call to **worker\_create**, **sch\_init** is used to initialize the scheduler global variable which has the type **scheduler\_t.** The scheduler init process includes:

1. Initializing and allocating space for the runqueue
2. Initializing, allocating space and getting the scheduler context
3. Creating a **tcb** for the main thread and enqueuing it into the runqueue
4. Creating and starting the interrupt timer

**Running the** **Scheduler**

Our scheduler runs an infinite while loop inside the **schedule** function, which is the function passed to the scheduler context during **makecontext**.

In each iteration of the loop, the following steps happen in order:

1. If the runqueue is not empty, dequeue a thread from the runqueue (choice determined by the type of scheduler, explained more below)
2. Set the thread’s status to RUNNING
3. Swap to the thread’s context for it to run
4. After the timer expires, the context will be swapped back to the scheduler’s, now we need to see if the thread is finished or not to determine whether to enqueue it back to the runqueue

* IF the thread’s status is RUNNING or READY:
  + Set it to READY and enqueue it back to the runqueue
* IF the thread’s status is BLOCKED or FINIsHED
  + Set current\_thread to NULL and do NOT enqueue it back to the runqueue

**Timer Interrupts**

I implemented timer interrupts using a looping itimer, using our utility functions **create\_timer** and **timer\_disable**, we initial the timer as the last step of the scheduler initialization process mentioned above and disable and enable it whenever there is an operation on our data structure so it doesn’t interrupt the thread during the middle of an operation.

Note that since our timer is looped, itimer doesn’t need to be set every single time an interrupt is needed.

**timer\_schedule\_handler**: this is the function that is passed to itimer and called every time the timer expires, and simply just swaps back to the scheduler.

Now below I will explain the differences that I implemented the scheduler depending on the type. Note that our core data structure runqueue is always the same – an array of linkedlist, MLFQ needs multiple queues so that works but since PSJF threads does not have priority, it only needs a single queue, we simply just use one of the linkedlists (the DEFAULT\_PRIO one with index 1) in the runqueue array and leave all other unused.

The main difference between the implementation of the schedulers is the **dequeue** and **enqueue** process. I will describe them in detail below

**2.1. Pre-emptive SJF (PSJF)**

**Dequeue**:

For **PSJF**, we need to dequeue the thread that has the shortest elapsed time, so I have a utils function - q\_dequeue\_shortest\_runtime that gets the shortest elapsed thread and dequeues it.

* + **Assumption**: “the more time quantum a thread has run, the longer this job will run to finish”. Based on this assumption, we kept track of how MANY time quantums a thread has ran and just picked the shortest one everytime.
  + Note that we are keeping track of how MANY time quantums a thread has ran, NOT how long, we are basically using time\_quantums as a unit. This this because I think that actually timing how long a thread has ran for is not really necessary because we know that it will run for TIME\_QUANTUM time.

**Enqueue**:

For **PSJF**, after the thread has ran for a time quantum, we simply just inserts the thread back to the runqueue without changing anything.

**2.2** **Multi-level Feedback Queue (MLFQ)**

**Dequeue**:

For **MLFQ**, we need to dequeue the thread that has the highest priority, so I also made a utils function - rq\_get\_index\_highest\_nonempty – that will get the highest priority queue that is nonempty in my multi-level runqueue. And it will dequeue the first element in that.

**Enqueue:**

For **MLFQ**, after the thread has ran for a time quantum, we need to move it to the next lower runqueue, I simply just decrease the priority index by one and re-enqueue it to that queue.

Additionally, to prevent starvation, we also needed to periodically move the low priority threads to the highest. For that, I just made it so that timer\_schedule\_handler would dequeue everything from the lowest priority queue and add it to the highest every REFRESH\_QUANTUM cycles (this is defined in multiples of TIME\_QUANTUM). For example, by default, every 3 time quantums all lowest priority threads will be moved to the highest.

**Benchmark Results**

**Analysis**

**Comparison with Linux pthread**