**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** – Stores all scheduler data needed — the runqueue, main/scheduler context, pointer to the currently running thread, etc..

**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

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

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

**Benchmark Results**

**Analysis**

**Comparison with Linux pthread**