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

Names: Michael Liu msl196

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

In this section, I will benchmark the results of my thread library with different configurations of worker thread numbers. Since the readme says the requirement is 50-100 threads, I will be testing my library with the following amount of threads: 3, 10, 50, 100

For all three benchmarks: external_cal, parallel_cal, and vector_multiply, I will screenshot the result logs for each thread amount and for each queue type.

Note that all my benchmark timed results are calculated in **micro-seconds**, and if you need more than **1000 threads** please change the **MAX_THREADS** macro in the header file.

SCHED=PSJF			
# threads	external_cal	parallel_cal	vector_multiply
3	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 3 *************************** Total run time: 350686 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 60 Average turnaround time 190246.000000 Average response time 122117.800000 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 3 *************************** Total run time: 1681415 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 166 Average turnaround time 573814.333333 Average response time 20052.666667 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 3 *************************** Total run time: 37226 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 3 Average turnaround time 22133.000000 Average response time 19114.000000 ***************************
10	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 10 *************************** Total run time: 324358 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 40 Average turnaround time 77494.700000 Average response time 55075.300000 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 10 *************************** Total run time: 3835735 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 382 Average turnaround time 437743.600000 Average response time 55182.700000 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 10 *************************** Total run time: 28544 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 10 Average turnaround time 18418.200000 Average response time 18224.600000 ***************************
50	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 50 *************************** Total run time: 330074 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 80 Average turnaround time 103917.440000 Average response time 99327.040000 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 50 *************************** Total run time: 2358171 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 253 Average turnaround time 293366.640000 Average response time 255833.920000 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 50 *************************** Total run time: 44189 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 50 Average turnaround time 26894.260000 Average response time 26884.080000 ***************************
100	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 100 *************************** Total run time: 332475 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 140 Average turnaround time 104019.519608 Average response time 105675.490196 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 100 *************************** Total run time: 3072651 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 349 Average turnaround time 531015.920000 Average response time 509691.430000 ***************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 100 *************************** Total run time: 40683 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 100 Average turnaround time 25004.850000 Average response time 25001.530000 ***************************

As we can see from the log above, the results varies,

- for external_cal, as the number of threads increased, the total runtime stayed about the same. But the total context switches, turnaround time and response time differs, the logs above shows that when there are 10 threads, all of those statistics are lower, meaning this algorithm running at 10 threads for my implementation is the most efficient.
- for parallel_calc, we can tell that 3 threads had the fastest runtime and least context switches, but not nessarily the fastest turnaround time, which 50 threads wins at that case,

- and for vector_multiply, 10 threads is the most efficient since overall it had the shortest times.

SCHED=MLFQ			
# of threads	external_cal	parallel_cal	vector_multiply
3	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 3 **************************** Total run time: 516958 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 91 Average turnaround time 201564.600000 Average response time 265617.400000 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 3 **************************** Total run time: 2216239 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 219 Average turnaround time 748771.000000 Average response time 26784.000000 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 3 **************************** Total run time: 25103 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 3 Average turnaround time 16705.000000 Average response time 15044.000000 ****************************
10	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 10 **************************** Total run time: 498828 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 59 Average turnaround time 130282.727273 Average response time 105929.727273 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 10 **************************** Total run time: 2360772 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 234 Average turnaround time 343425.600000 Average response time 131968.600000 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 10 **************************** Total run time: 33309 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 10 Average turnaround time 20664.800000 Average response time 20393.900000 ****************************
50	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 50 **************************** Total run time: 523681 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 122 Average turnaround time 290494.537037 Average response time 285673.537037 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 50 **************************** Total run time: 3111595 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 329 Average turnaround time 764754.980000 Average response time 732050.040000 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 50 **************************** Total run time: 40420 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 50 Average turnaround time 24823.780000 Average response time 24812.040000 ****************************
100	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 100 **************************** Total run time: 536796 micro-seconds verified sum is: -822709616 Total sum is: -822709616 Total context switches 162 Average turnaround time 258391.480392 Average response time 255675.794118 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 100 **************************** Total run time: 2838729 micro-seconds verified sum is: 83842816 Total sum is: 83842816 Total context switches 329 Average turnaround time 1246993.170000 Average response time 1245064.110000 ****************************	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 100 **************************** Total run time: 38485 micro-seconds verified sum is: 631560480 Total sum is: 631560480 Total context switches 100 Average turnaround time 24049.130000 Average response time 24046.780000 ****************************

Comparing the results from PSJF with MLFQ, we can see that

- for external_calc, any number of threads with MLFQ was slower than PSJF
- for parallel_calc, the results vary, MLFQ might have better runtime sometimes but PSJF also might have better turnaround and response time sometimes.
- For vector_multiply, I got similar results for both schedulers since it is a really short task.

Analysis and Comparison with Linux pthread

After commenting out #define USE_WORKERS 1, and I ran the same test with the linux pthread library and here are the results:

Linux pthread			
# of threads	external_cal	parallel_cal	vector_multiply
3	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 3 *************************** Total run time: 1413313 micro-seconds verified sum is: -822709616	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 3 *************************** Total run time: 594304 micro-seconds verified sum is: 83842816	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply *************************** Total run time: 88791 micro-seconds verified sum is: 631560480
10	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 10 *************************** Total run time: 2602387 micro-seconds verified sum is: -822709616	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal *************************** Total run time: 411679 micro-seconds verified sum is: 83842816	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 10 *************************** Total run time: 194482 micro-seconds verified sum is: 631560480
50	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_ca *************************** Total run time: 2570117 micro-seconds verified sum is: -822709616	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal 50 *************************** Total run time: 136644 micro-seconds verified sum is: 83842816	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply 5 *************************** Total run time: 375169 micro-seconds verified sum is: 631560480
100	msl196@ilab1:~/416/Project2/code/benchmarks$ ./external_cal 100 *************************** Total run time: 2264113 micro-seconds verified sum is: -822709616	msl196@ilab1:~/416/Project2/code/benchmarks$ ./parallel_cal *************************** Total run time: 183633 micro-seconds verified sum is: 83842816	msl196@ilab1:~/416/Project2/code/benchmarks$ ./vector_multiply *************************** Total run time: 393570 micro-seconds verified sum is: 631560480

Comparing my results with the linux pthread library, first of all, the verified sum from the pthread is the same as my library, which means that my final result is correct. Additionally, according to the runtime, overall my implementation for external_cal and vector_multiply was faster than pthread, but for parallel_cal, the pthread library was significantly faster than my library.

3. Extra Credit Scheduling Championship

Note: to compile matrix.c use matrix as the SCHED param (make SCHED=MATRIX)

Goal: multiply two randomly generated matrices (ensuring the result is correct) using a custom scheduling algorithm in the least amount of time.

Logic

The way we thought for doing this is splitting up the overall multiplying work, for instance if we use 40 threads, the matrix would be split up by columns of the second matrix 40 times and each threads would be responsible for multiplying one of those, this effectively splits up work and improve the overall runtime by a significant amount.

Matrix Scheduler

Our implementation for a scheduler just for multiplying matrix is a modified version of PSJF scheduler. Basically, all we doing that's difference is at each iteration, instead of dequeuing the thread that currently has the shortest runtime, we dequeue one that has the longest, aka, the run that has been running for the longest total time. We think that this is more efficiency for matrixes because we should prioritize finish multiplying a matrix instead of trying to start new ones, with retrieving the longest thread each time, we are finishing multiplying a region of a matrix faster, thus improving overall efficiency.

Results

```
msl196@ilab1:~/416/Project2/code/benchmarks$ ./matrix
***************************
Total run time: 94430492 micro-seconds
Total context switches 9452
Average turnaround time 2923216.700000
Average response time  576483.375000
***************************
```

Here are the results for running our scheduler and algorithm, as we can see, it took quite a while to finish multiplying but the average turnaround time isn’t that bad since we split it up into many different regions.