CS416 Project 2: User-Level Thread Library Report

Elijah Ongoco (edo16), Michael Comatas (mac776)

23 October 2020

1 API Function Descriptions:

- int mypthread_create(mypthread_t * thread, pthread_attr_t * attr, void *(*function)(void*), void * arg)
  - Using a global idCounter variable, the create function will first and foremost check if it is being called for the first time by checking if idCounter is still 0. If it is being run for the first time, it will initialize all TCB structs in the runQueue to be in a NOTUSED (uninitialized) state, then create the first thread; the main thread. The idCounter will then increment, in order to make sure the first thread after the main thread will have the appropriate tid. After the main thread is initialized at index 0 of the runQueue, the timer is initialized. The timer will refer to the swapToScheduler() function every time the timer goes off, which happens during every QUANTUM block of time. QUANTUM is defined in the header file as 10000, which when put into the microseconds counter of the timer, becomes 10 milliseconds.
  - After the idCounter is no longer 0, the create function will initialize a new TCB at the runQueue index of the current value of the idTimer. The new thread's stack will be malloc'd, and the context will be set to this stack via makecontext(). The context will use the arguments given to give each thread the right function and arguments to run on. idCounter will increment to prepare the next thread to have an appropriate, distinct tid. At the very end of the function, there is an if statement that will use getcontext() on the main thread's TCB if idCounter is 2. This is because, after create is run for the very first time, idCounter will increment twice. idCounter being 2 at the end of the function means it is still the first run, so it is the right time to getcontext for the main thread. This makes the context of the main thread the context of the main() function that is calling all of the API's functions.

  - Once the main thread is able to exit we call the cleanup() function using atexit(). This allows us to free all the thread stacks that were created for each thread.

- int mypthread_yield()
  - Yield simply allows other threads to get a hold of the CPU, so it calls the schedule function. justExited is set to 0 to let the schedule know that the last thing that was done was not an exit, for use in schedule().

- void mypthread_exit(void *value_ptr)
  - First, the timer will pause as there is no point in an exiting thread to be put back into the runQueue. Then, the status of the thread will be set as DONE. Afterwards, exit() will check if any threads called join on the calling thread in the past. If there were, those threads will be set back to READY, and their waitingOn attribute will be set back to -1. This will allow them to be a valid candidate when it is time for the scheduler to pick a new thread to run. If the *value_ptr is not NULL, this means the exiting thread will need to save a return value in the TCB's returnValue attribute. Afterwards, the timer will be resumed, and the scheduler will run, after letting it be known that a thread has just exited.

- int mypthread_join(mypthread_t thread, void **value_ptr)
  - First, join will check if the thread it is referencing has already exited. If so, the scheduler will be called, and justExited will be set to 0, meaning a thread has not just exited.
  - If the value_ptr is not null, then the value_ptr will be set to the referenced thread's returnValue attribute, where the return value is/will be upon a thread's exit.
  - The referenced thread will have its beingWaitedOnBy attribute set to the id of the calling thread, so it knows who to contact once it exits. The calling thread will be set to a WAITBLOCK status, and its waitingOn attribute will be set to the tid of the referenced thread, so it will not be picked by the scheduler until the thread it is waiting on has exited. Afterwards, the scheduler will run, and justExited will be set to 0.

- int mypthread_mutex_init(mypthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
  - This function initializes a new mutex by using a global mutexIdCounter variable. The .mutexId of the mutex will be set to the mutexIdCounter, and the .lockState will start off as being UNLOCKED. Afterwards, the mutexIdCounter will increment, to allow the next mutex to have a distinct ID.

- int mypthread_mutex_lock(mypthread_mutex_t *mutex)
  - Using the atomic test and set function, the calling thread will check the lock state of the mutex. If it is still locked, it will be thrown into a while loop where the calling thread will be set as WAITBLOCK, and it will be marked that the thread is waiting on the mutex in question via the TCB's waitingOnMutex attribute. The thread will not be a viable candidate for the scheduler to pick in this state. The scheduler will then be called, and justExited will be set to 0.
  - If the calling thread manages to make it past the while loop, that means it has the rights to use the mutex above all other threads. The calling thread will then lock the mutex so it can do its work without other threads disrupting it.
- int mypthread_mutex_unlock(mypthread_mutex_t *mutex)
  - First and foremost, the mutex in the argument will be set to an UNLOCKED state. Then, a loop will run through the runQueue's currently used indices. If a thread at an index happens to be waiting on the mutex in the argument, its status will be set to READY and its waitingOnMutex attribute will be set back to -1. This will mark that the thread is ready to compete for access to the mutex again, so the scheduler will be able to choose it.
- int mypthread_mutex_destroy(mypthread_mutex_t *mutex)
  - Nothing was malloc'd during the API's initialization of a mutex, so destroy does the same thing as unlock. The mutex will be unlocked and any thread waiting on the mutex will be set back to a READY state.
- static void schedule()
  - This method will pause the timer and then pick the appropriate scheduling method to carry out. We are both undergraduate students, so it will only reach sched_stcf().
- static void sched_stcf()
  - This method will implement a shortest job first version of a scheduler. Since we do not know the expected run times for each thread, the scheduler will simply look for the thread that is in a READY state and has run for the shortest amount of quantums. This will be done by setting up an obscenely high lowestQuants variable as a starting point. Then, the scheduler will loop through all used threads, check if they are:
    * READY
    * Not waiting on a thread (from join threaD)

        * Not waiting on a mutex
        * Not done
        * Not in a WAITBLOCK state
        * In possession of a quantumsElapsed variable that is less than the current state of lowestQuants.
    - If a thread passes all of these requirements, variable toRun will be set to that thread's tid and lowestQuants will be set to that thread's quantumsElapsed. Then, the subsequent threads in the loop will have to have a lower quantumsElapsed than that in order to be marked as the next thread to run in toRun.
    - At the end of the loop, if lowestQuants was not changed from its high value, that means no new thread was selected, so the current thread will keep running.
    - If justExited is 1 and lowestQuants has changed from the initial value, that means we can safely setcontext() to the newly selected thread, marked in toRun. This is because the last thread exited, so we don't need to save the state of that thread's context.
    - If justExited is 0 and lowestQuants has changed from the initial value, that means we must save the state of that last thread's context. So, we must use swapcontext() from the last thread's context to the newly selected thread's context, so the last thread can pick up where it left off the next time it is chosen. In both of these if statements, the timer is resumed right before the context is switched, so the scheduler can be called after another 10 milliseconds has passed.

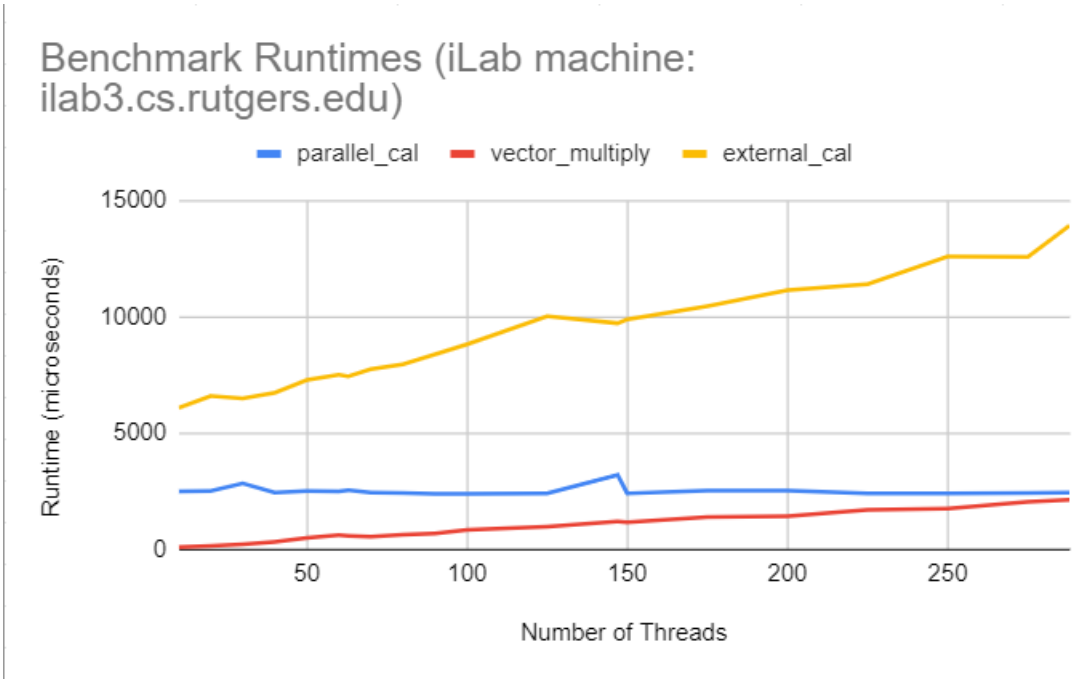
- static void sched_mlfq()
    - Not used, we are both 416 students.
- void swapToScheduler()
    - This is the function that is called every time the timer goes off, which is every QUANTUM (10 milliseconds ((10000 microseconds)). This thread pauses the timer, and it will increment the current thread's quantumsElapsed attribute, to mark that the thread has run for one more quantum. The actual scheduler is then called, and justExited will be set to 0.
- void pauseTimer()
    - Sets the timer in ITIMER_PROF to a zero'd out timer, so swapToScheduler will not be called no matter how much time passes. This is done so sensitive functions like the scheduler and exit are not interrupted.
- void resumeTimer()

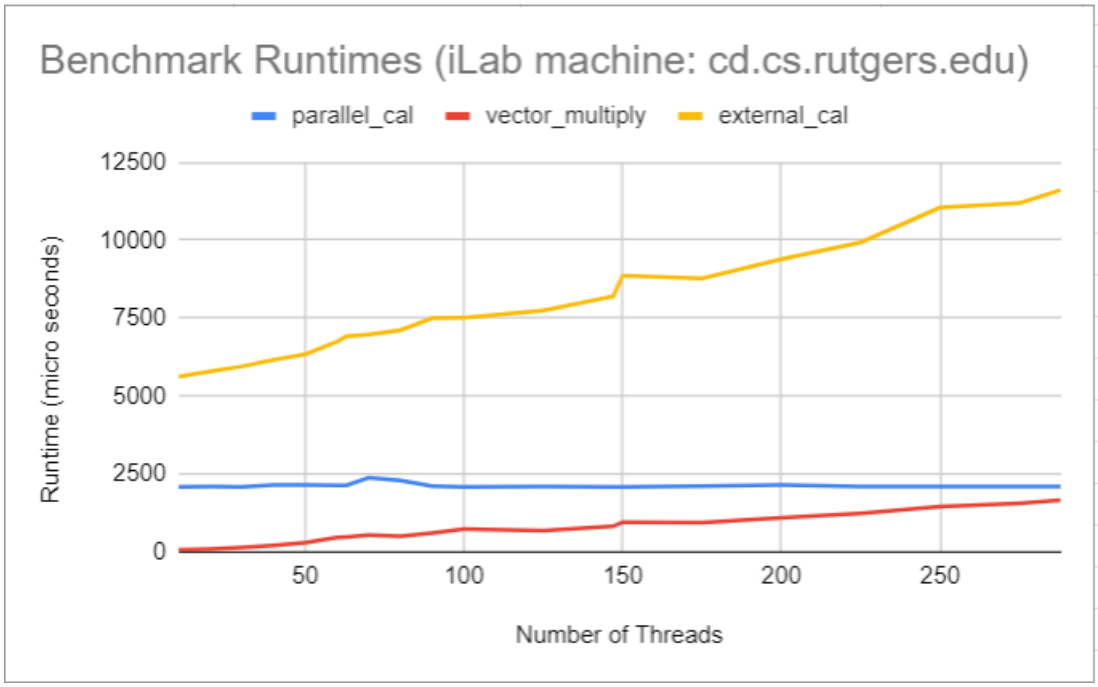
  - Sets the timer in ITIMER_PROF to the original timer made in pthread_create, so it will continue to ring swapToScheduler() every 10 milliseconds/10000 microseconds. Called when sensitive functions like exit and schedule are finished doing their work.

- void cleanup()
  - At exit we call the function cleanup in order to free all the thread stacks created for different threads. We use a while loop to iterate through each element of the runQueue and free the thread stack if the status is set to DONE.

2 Benchmark Results:

We ran the three different benchmark codes with multiple numbers of threads, ranging from the default amount of threads (no arguments) to 288. The results can be seen in the graphs.





We can see that external_cal takes the longest to run by quite a bit, but it also has the most variance since we must run ./genRecord.sh before hand which makes the results change every time that is run. It also increases in run time linearly as the number of threads increase.
parallel_cal take about the same amount of time to run, the amount of threads didn't seem affect the run time in any significant way.
vector_multiply was the fastest running code out of the three benchmarks, but followed the same pattern as external_cal, it went up in a linear motion as the number of threads increased.
Although we ran the benchmark codes on different machine, the run times only vaired a little, mostly in the external_cal code. Even though the run time varied a little, the pattern was still the same.

3 Challenges we encountered:

There were many challenges, so they will be listed per function.

- pauseTimer(): pausing the timer without making the timer permanently frozen, even after resumeTimer. Fixed by making a new zero'd out timer instead of fiddling with only one timer struct.
- swapToScheduler(). Properly updating the quantums elapsed. Fixed by making a new variable that kept track of which thread was currently running.

- schedule(): making it switch to the sched_stcf() function correctly. Fixed by declaring the functions
- sched_stcf(): Making sure the scheduler doesn't pick threads that are waiting on a mutex/join. Fixed by making an enum that has READY, RUNNING, WAITBLOCK, DONE, and NOTUSED states.
- join(): segfaults if called on a thread that already exited. Fixed by altering the runQueue's data structure to keep finished threads around, in order to check if they are done or not.
- create(): correctly initializing the main thread's context was a huge hurdle. Solved thanks to help from other people on piazza.

4 Further Improvements:

- One of the things we could improve on is the runtime. Compared to the actual pthread's library, our API takes up to 3x-4x longer on average.
- Another thing that can be improved is that, as can be seen with the initialization of the runQueue, our library makes the assumption that no more than 299 threads will be made at once. This can be fixed by using a different type of data structure for the runQueue, such as a linked list.