Andrew Park (ap1614), Elysia Heah (eyh24)
CS416: Operating Systems
Fall 2020

Project 02 Report: User-level Thread Library and Scheduler

In order to implement our own version of a user-level thread library, several key structures needed to be established to manage the threads and scheduler: the thread control blocks, a TCB list containing all of the blocks, and a priority queue that would abide by the STCF protocol by weighing jobs with shorter completion times above those with longer ones. These are organized into two separate but related files: mypthread.c and queue.c (with their respective header files). The mypthread.c file deals with thread status and control blocks, pre-emption and scheduling, and mutexes to ensure thread safety. Meanwhile queue.c is dedicated entirely to establishing, ordering, and cleaning the priority queue. This ties back to the scheduling function in mypthread.c that applies the pre-emptive STCF algorithm with a timer and signal handler.

1. mypthread

The mypthread header file establishes a TCB as a structure holding several important values specific to that thread being held: the thread id, status, context, quantum, waiting thread, and value pointers. The status of a thread is defined by the value of an enumerator with the following options: runnable (ready), sleep, finished, cleanup. Finally we have the library tcblist that is simply a linked list of all the blocks; storing it in this way makes adding and deleting any node (block) at any position of the list much easier.

In the mypthread.c file, several things are declared globally to make it easier to manipulate them anywhere in the program: ThreadQueue, a priority queue organizing all the threads, newTID which incrementally assigns each new thread an ID, and the ID and TCB for the current thread being run.

In *mypthread_create*: the general procedure is to create and allocate space for the corresponding control block. The thread id is assigned as the current thread being examined and tied to the tid attribute of the tcb as well. Since the thread has just been created, the waiting, quantum, and value pointer attributes are all set to the default null/zero values. The newTID value is then incremented in preparation for the next thread to be created. The difference lies in the check for the newTID value at the beginning of the function. If newTID is

not zero, then this is not the first thread in the library and we have to add this to the priority queue where it will wait for its turn. We call the enqueue function to add it to ThreadQueue and add the thread context with getcontext() and makecontext(). If newTID==0, we turn this into our main block/thread being run and assign it to runningBlock in *mainThreadAdd()*. This can be started immediately because there is nothing ahead of it to wait on. We run the first thread and set up the pre-emption, making sure that we signal schedule() to take care of swapping out threads according to STCF (running with pre-emption will be expanded more on in the *schedule()* function explanation). It is also in the *mainThreadAdd()* function where we make sure to go through the entire queue of threads and free all the data at exit once the program terminates in full.

In *mypthread_yield*: all this function does is give up the currently running thread and put it back in the queue to run the next one. Therefore, we can save a lot of effort and just simply call schedule() which will automatically do that for us.

In *mypthread_exit*: a finished thread is terminated. We update its status to FINISHED and check whether or not it has to hold a value pointer. If it is not null, we assign it to save the return value and instead mark the status of the thread to CLEANUP. This indicates that it is necessary to go back and free all of the block's data later. Afterwards we set up the next thread by going back into the queue and looking for the following waiting thread; we update the queue to set this next thread as RUNNABLE, or ready, and grab its TCB. Finally, we have to enact schedule() again to start the scheduling procedure up again now that the running thread has terminated successfully.

In *mypthread_join*: this function ensures that the thread has FINISHED first. In this case, it saves and passes back the return value if the value pointer is valid and marks CLEANUP. Otherwise, the thread in runningBlock is put to the status SLEEP and waits on the thread first. We note that since this has terminated, it is necessary to call the schedule() function once again at the end to restart the entire scheduling procedure.

In *mutex threads*: these are fairly straightforward. Mutex_init initializes the mutex lock structure. Mutex_lock in turn uses a built-in atomic_flag_test_and_set function in order to test the mutex passed in. if it is successfully acquired, the critical section can be entered; otherwise, push the current thread onto the block list and context switch to the scheduler thread. The mutex_unlock function makes the mutex available again and puts the appropriate blocked threads back to RUNNABLE and puts into queue such that they can compete for the mutex again. The mutex_destroy function does exactly as it says and destroys the mutex structure by deallocating it.

In *schedule() / sched_stcf():* upon entering this function, we make sure that we call a cleanup function to free any finished threads with the status CLEANUP. We also ensure that we don't get interrupted by a timer by signal(SIGPROF, SIG_IGN). Now we can start manipulating the queue without fear. The current thread that runningBlock is keeping track of is now going to be stowed back in ThreadQueue, so we save it in a temporary variable prevThread and instead grab the immediate thread from the priority queue with dequeue. If the thread we dequeue is valid and in fact prevThread, it means that there is only one left; we reset it to runningBlock and simply allow schedule() to return. Otherwise, we increase the quantum of the prevThread to signify that this thread has already elapsed once. We can proceed to enqueue the prevThread with its updated quantum value back into threadQueue and then enforce pre-emption for the new running thread. Sigaction and itimerval structures are created; the signal handler is set to schedule() such that this function will keep continually managing the running threads and updating the status of the queue. The signal being caught is SIGPROF, indicating that it is waiting for the profiling timer to expire. The timer sets off the SIGPROF signal according to the interval TIME_QUANTUM, whose value was experimented on for some time before settling on 20 milliseconds. Finally, we remember to switch the contexts of the two threads.

2. <u>queue</u>

Queue is dedicated entirely to managing the priority queue of threads/thread control blocks that is the backbone of the SCTF algorithm. We establish a PQueue node structure in the header file that makes up a linked list of TCBs and keeps track of each of their quantum values. The priority queue orders its elements based on ascending quantum. Using a linked list to represent this queue is highly suitable for this program because we rely heavy on enqueueing and dequeueing thread nodes. Inserting, deleting, and ordering elements is made easy with a linked list which is extremely flexible with resizing and reordering.

In *enqueue*: this function takes in an appropriate queue, TCB block, and its quantum. It runs through a check to see if its parameters are valid, and then sorts the block into the queue based on the quantum value.

In *dequeue*: this function takes in the queue and returns the first thread that has its status set to RUNNABLE. It runs through the entire list to find such a thread; once found, it updates the queue to remove it, connect its predecessor and successor, free it from the queue, and returns the block.

In *updateQueueRunnable*: this updates the status of the queue to find the next waiting thread according to its input parameter and sets it from SLEEPING (waiting) to RUNNABLE, meaning that it is now ready to go.

In *checkIfFinished*: pretty self-explanatory, this checks if the given thread has been set to FINISHED yet and allows the function that called it to proceed with an appropriate decision.

In *getBlock*: also self-explanatory, just a getter function that retrieves a certain TCB from the queue with the thread id requested.

In *cleanup*: this traverses the entire queue and searches for any blocks that have the status CLEANUP. If so, that means that this thread is no longer necessary and we proceed to free all of its data: the context within the TCB, the TCB block, and the queue's node itself.

3. Benchmarks

First, the tests were run over the benchmarks using our program. We started with a small number of six threads to get a decent gauge, and found:

Ext Calc: 5945 us

Parallel Cal: 2307 us

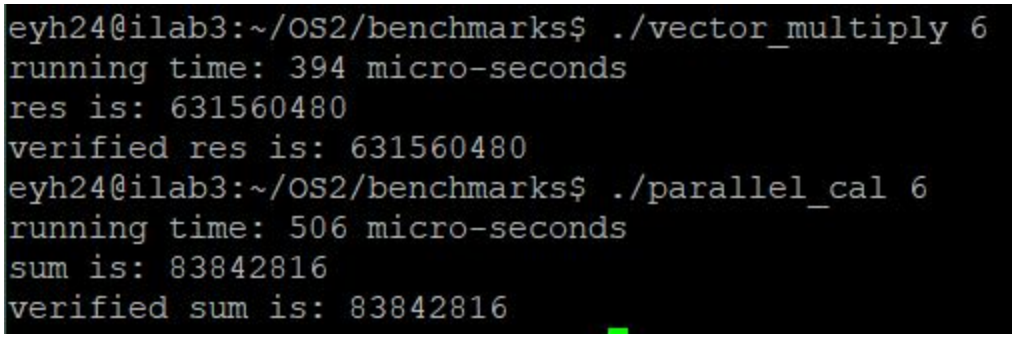
Vec Mult: 15 us

Some slight stress-testing was also done to see it would perform with a much larger quantity of threads, and we found that even more substantial tests performed relatively well:

```
eyh24@ilab3:~/OS2/benchmarks$ ./parallel_cal 750
running time: 2790 micro-seconds
sum is: 83842816
verified sum is: 83842816
eyh24@ilab3:~/OS2/benchmarks$ ./parallel_cal 300
running time: 2475 micro-seconds
sum is: 83842816
verified sum is: 83842816
eyh24@ilab3:~/OS2/benchmarks$ ./parallel_cal 150
running time: 2471 micro-seconds
```

Even raising the number of threads by several times increases the running time but a relatively small amount, so this is satisfactory.

Next, we commented out the macro USE_MYTHREAD 1 in mypthread.h to try and see what the runtimes were like using the linux library, and how it compared to ours. Curiously, these were some of the results:



We used a low thread count of six to match against the initial tests we did with our version. Parallel_cal ran much faster than ours, which is to be expected given our schedule implementation. Curiously enough, vector_multiply was worse with the linux library. The reason for this is unclear thus far, but does not pose a problem.

4. Challenges

   Probably the most challenging part of the project was the scheduler. Making the library itself was manageable but implementing the scheduling meant that it was necessary to keep track of a lot more variables and ensure that everything was updating appropriately. Another linked list was necessary to organize and keep track of the TCB priority queue and the schedule function had to carefully handle signals and timers that only signal at the correct time such that it does not get interrupted prematurely. It was also necessary to think critically about how to fit in the schedule function to work in conjunction with everything else-- most significantly, where it was appropriate to call the schedule function (once a thread has finished).

5. What to do better

   The outcome of this project was satisfactory and all aspects of the project were completed, but there were some parts that could probably have been handled better. For example, memory management was enforced by using the function atexit() to do a cleanup of the entire queue and free everything in it once the entire program was finished. However, it might have been more appropriate to procedurally free TCBs/threads and all of their data values as they are joined or exited.