Project 2 report

Harsha Somisetty, HS884

October 24, 2021

Logic

To begin, we initialize various variables, included a running thread queue, a queue of finished threads, the current executing thread, and pointers for timers, scheduler, and the main thread

Thread Functions

Create

On the first thread creation, we initialize contexts for the main thread and scheduler (the latter contains the rest of the instructions to execute, the former manages all the created threads)

In the rest of the calls to $\text{pthread}_{\text{create}}$, we first increment the thread counter to serve as an id, keep track of the thread execution status and how many times it executed, and initialize it's own stack and context. Get context specifically initializes the struct containing the context variables, and then make context changes the struct to point to the desired function to execute

Yield

Here, the context of the calling thread is simply paused, and control is given back to the scheduler to choose a different thread to execute

Exit

Here, we mark the status of the thread as finished so the thread is not reentered into the running queue, and additinally, we add the finsihed thread ot the finished queue

Join

Here, we first check if the thread to be joined is already executed (in the finished queue). if it is, then we store the result if any and continue. if not, we yield to another thread, and wait until the thread to be finsihed is exited

Mutexes

We first initialize the mutext, which keeps track of the current holding thread, and a list of all threads which requested that mutex.

Then, when a mutex is activated, we first make sure the mutex isn't already locked. We do this by checking if a thread holds the mutex, and if yes, then we me mark the requesting thread as blocking, then switch back to the scheduler (without reinserting the blocked thread). If not, then the thread gets control of the mutex

Finally, when the mutex is released, we restore all the blocked threads to the scheduler, and open up the mutex

Scheduler

For the scheduler, we have a priority queue that is a linked list of threads, where the nodes contain threads, and are organized by increasing elapsed time. Then, when the scheduler wants to decide on the next thread, it pops the head and sets that as the Current executing thread. to enqueue a thread back, it iterates through the linked list, and inserts the thread in the proper spot

Benchmark

(Times in microseconds)

Vector Multiply

threads	My Pthread	Default
1	2804	53
10	2855	259
100	3145	577
1000	2936	814

Parallel Calc

threads	My Pthread	Default
1	error	267
10	285	40
100	286	18
1000	295	35

External Calc

threads	My Pthread	Default
1	10251	692
10	10354	2512
100	10308	2718
1000	10204	2411

Challenges

The biggest initial challenge was figuring out how the context actually worked, and what each of the get, make, set, and swap context functions did, and when each were supposed to be used. After experimenting with the functions for a while, I finally understood when to use which, but the deadline was quickly approaching, so I still wasn't able to explore things like the uc link pointers and more

Also, debugging the context switches and setting up the timers at the right points was extremely difficult to fine tune.

Improvements

My implementation could first be better debugged so the tests all properly run.

Since this thread implementation still runs on one processor, my implementation can inherently not be as fast as the default implementation.

To increase speed, I would have to explore the average idle time spent. Specifically, since many of the thread calls are not very time intensive, an extremely large timer interrupt would waste too much time. However, if my timer is too small, then the constant switches again wastes computation.