Dhaval Patel dp918
Dan Garry dcg101

(a) What kind of data structures did you use to keep track of different thread threads/TCBs? A linked list? A linear array?

We planned the implementation of our scheduler with three things in mind: simplicity, risk-reduction and speed. The first two go hand in hand, a simple design is easy to verify, and even easier to extend. Thus, all of our routines have cyclomatic complexity values less than 10. The third is a little more tricky and is where data structure choice really comes into play.

With these values at the forefront of our mind, we decided to use a circular linked list as our fundamental way of storing TCBs. Circular linked lists checked all the boxes we were looking for in a data structure. Insertions and deletions are constant time, and since we are keeping track of thread status by whether or not it exists in the list, this is a necessity. Traversing is a breeze. By having a pointer from the rear to the head, we can traverse by following the “next” struct members without having to rely on if statements and recursion as you would with arrays and trees. With that being said, this data structure really only suffers on searches, which is equal to the amount of threads scheduled to run.

(b) What was the most challenging part of implementing the user-level thread library/scheduler?

Without a doubt, grasping the context family of routines was the greatest challenge for us. The routines themselves aren’t terribly complicated - a simple google search and 15 minutes of reading resolved any questions we had about their behavior. Our confusion mainly stemmed from their members, how to initialize them and once created, how to keep track of them in a manner that is both intuitive to us, the designers, and anyone who may use the code in the foreseeable future. This goes back to why we chose to use the circular linked list as the foundation of our scheduler.

Setting aside ambiguous error messages (Profile Timer Expired), everything else regarding the completion of this project was rather elementary. Creating the queue library, setting up the interval timer, signal handler and populating the provided routine skeletons were all hurdles that we tackled rather quickly. Our use of synchronization mechanisms, although imperfect, was something we had experience with in 198:214 and after a little refresher, we picked up pretty quickly.

Establishing the logic of the scheduler routine and its accompanying my_pthread routines was not particularly troubling. A few moments away from the keyboard, drawing diagrams on napkins and thinking about what we needed to do was enough for us to formulate a good plan of attack.

(c) Is there any part of your implementation where you thought you can do better?

Absolutely!!! As is the case with most projects, there are components of our implementation that can be overhauled to be more efficient, in regards to space and speed. For example, we could take a more realistic approach and use red black trees as is utilized in Linux's completely fair scheduling system. Or perhaps we could use an array (although reallocation on reaching maximum capacity is something that does not spark joy) to reduce the run time of checking the status of a thread.

As we observed, and alluded to before in the previous question, our implementation begins to break down once TIME_QUANTUM_MS approaches values sub 100 microseconds. This is due to a need for better synchronization, and atomicity with some of the switching that we are doing. Although we attempted to resolve these issues using mutexes, we were not entirely successful, mainly due to our own ignorance of many of the inner mechanisms of the OS. Given more time, and after some research, this is something we would definitely want to iron out of our scheduler.