Kunal Jangam

Ksj48

Thread control block contains the ucontext of the thread, a pointer to the thread control block of the parent, the status of the thread, the priority of the thread, the thread id, the return value of the thread, and its stack,. Status of 0 represents ready, 1 represents running, 2 represents blocked, and 3 represents done. Priority represents the number of time slices the thread has run for.

The mutex struct contains an integer to see if it is locked and a queue of all the threads that is has blocked.

A node struct and queue struct were made to store the tcbs of the threads.

For both PSJF and MLFQ I used an array of queues to hold the tcbs. The index of the array refers to the priority of the tcbs in the queue. For MLFQ the array was 4 large, and for PSJF the array was 150 large. I maintained a global schedule ucontext, a global tcb pointer to keep track of the current tcb, and a queue for threads that had exited called "done".

The first time create is run, the setup function is called, which allocates memory to the storage structure, sets up the main function tcb and puts it into the structure, and initializes the timer and schedule ucontext.

In general calls of create, the thread to be created gets a tcb and gets put into the storage structure, and the context is swapped to the scheduler.

Yield sets the current thread to ready, puts its tcb into the storage structure, and swaps to the scheduler.

Exit sets the current thread to done, adds it to the done queue, saves the return value, and swaps to the parent of the thread. Both yield and exit stop the timer.

Join searches the done queue for the specific thread. If found, it gets the return value and removes the thread from the done queue. If not found, it swaps to the scheduler.

Mutex_init sets the mutex to unlocked and initializes the blocked queue.

Mutex lock locks the mutex if it is unlocked. If it is locked, it adds it to the mutex's blocked queue and swaps to the scheduler.

Mutex unlock and mutex destroy unlock the mutex and move all the threads from the mutex's blocked queue to the tcb storage structure. Since no dynamic memory was allocated in mutex_init, none is freed here.

Schedule runs current if no nother tcb is in the storage structure. Otherwise, it calls either the mlfq scheduler or the stcf scheduler.

The stcf scheduler puts the current thread in at a lower priority if an involuntary context switch occured. It then activates the timer and swaps in the highest priority thread in the storage structure.

The mlfq structure puts the current thread in at a lower priority if an involuntary context switch occurs. It then activates the timer and swaps in the highest priority thread in the storage structure. Every 10 context switches, the priority of all the threads in the storage structure is increased.

The ring function swaps to the scheduler every time the timer goes off.

PSJF:

vector_multiply

1 thread: rpthread: 16 microseconds. Pthread: 46

10: rpthread: 19 microseconds. pthread: 374

20: rpthread: 20 microseconds. Pthread: 392

50: rpthread: 52 microseconds. Pthread: 408

100: rpthread: 35 microseconds. Pthread: 421

parallel_cal

1 thread: rpthread: 2229 microseconds. Pthread: 2228

10 rpthread: 2227 microseconds. Pthread: 477

20 rpthread: 2228 microseconds. Pthread: 343

50 rpthread: 2228 microseconds. Pthread: 326

100 rpthread: 2229 microseconds. Pthread: 323

external_cal

2 threads: rpthread: 8027 microseconds. Pthread: 4657

10 rpthread: 8035 microseconds. Pthread: 2877

20 rpthread: 8027 microseconds. Pthread: 2811

50 rpthread: 8027 microseconds. pthread: 2766

100 rpthread: 8022 microseconds. pthread:2810

MLFQ:

vector_multiply

1 thread: rpthread: 15 microseconds.

10 rpthread: 21 microseconds.

20 rpthread: 25 microseconds.

50 rpthread: 55 microseconds.

100 rpthread: 35 microseconds.

parallel_cal

1 thread: rpthread: 2227 microseconds.

10 rpthread: 2228 microseconds.

20 rpthread: 2217 microseconds.

50 rpthread: 2228 microseconds.

100 rpthread: 2229 microseconds.

external_cal

2 threads: rpthread: 8094 microseconds.

10 rpthread: 8084 microseconds

19 rpthread: 8166 microseconds

I could not get external_cal to run properly with 20+ threads.

The pthread library was a lot more efficient the more threads were used, especially for parallel_cal. My rpthread library did not really gain much efficiency the more threads were used, but was faster for vector_multiply.