User-level Thread Library and Scheduler

Partners:

Steven Nguyen - shn27

Ethan Febinger - eef45

pThread Functions & Structures:

rpthread_create:

- allocated a new thread control block (tcb), sets it up, and enqueues it
- if the scheduler is not initialized, initialize the scheduler by making the main context a listItem, making a context for the schedule function, and sets up the timer

rpthread_exit:

marks a thread as finished and continues to scheduler

rpthread_join:

- checks if the thread to be joined is ENDED, if not, continue to the scheduler
- when scheduled again, repeat above

rpthread_mutex_t:

- mutexes store the thread that locked it and a linked list of threads
- on an attempt to lock, if already locked, continue through the scheduler to be added to the mutex's list of threads

schedule:

- if not on mlfq mode, sched_stcf(), else, sched_stcf()

sched_stcf:

- this is triggered from a context swap, either from the timer or when continuing from a function like rpthread_exit
- if triggered from interrupt or yield or join, adjusts the time the current context has run, then enqueue back into the scheduler
- if triggered from mutex, adjust the time the current context has run, then add to the mutex's list of threads
- if triggered from exiting a thread, the context stack is deallocated and dereferenced
- afterwards, the timer is restarted and the scheduler context swaps to the next context

sched_mlfq:

- Like sched_mlfq, this is triggered from a context swap from either a timer or from continuing from a function like rethread_exit
- Searches rpthread_mlfq, which is an array of linked lists.

Each element of the array represents a different level in the MLFQ, and each link list in the level stores the threads at that level.

The currently executing thread and its associated level were stored as global variables

The currently executing thread was always stored at the front of its linked list in the MLFQ

When this function is called, MLFQ removes the currently executing thread from the front of its linked list and inserts it at the rear of the same linked list

If the thread used up the entirety of its time slice, it is moved down a level instead

The scheduler then finds the first element of the first nonempty linked list and begins executing that thread

This ensures that elements with higher priority are always executed first and elements with lower priority are executed in round-robin

If a thread has just finished executing, it is not added back to the MLFQ

If a thread was just blocked by a mutex, it is added to a linked list of blocked threads for that mutex, and is only added back to the MLFQ when the mutex is unlocked

After 100 schedules, the priorityBoost() function is called and all functions are returned to the top level of the MLFQ

Benchmark Results:

	STCF (timer = 2.5ms)	MLFQ (timer = 2.5ms)	PTHREAD
./vector_multiply 2	24 micro-seconds	18 micro-seconds	140 micro-seconds
./vector_multiply 55	78 micro-seconds	46 micro-seconds	459 micro-seconds
./vector_multiply 1000	55 micro-seconds	60 micro-seconds	520 micro-seconds
./parallel_cal 2	2497 micro-seconds	2265 micro-seconds	1394 micro-seconds
./parallel_cal 55	2501 micro-seconds	2232 micro-seconds	556 micro-seconds
./parallel_cal 1000	2505 micro-seconds	2242 micro-seconds	558 micro-seconds
./external_cal 2	9061 micro-seconds	8216 micro-seconds	5194 micro-seconds
./external_cal 55	9103 micro-seconds	8130 micro-seconds	3490 micro-seconds
./external_cal 200	9055 micro-seconds	8123 micro-seconds	3542 micro-seconds

Compared to the STCF scheduler, the PTHREAD scheduler is slower in vector_multiply and faster in the other benchmarks. Compared to the MLFQ scheduler, the PTHREAD scheduler is also slower in vector_multiply and faster in the other benchmarks. When comparing the two different schedulers we created, MLFQ always performed better than STCF with the exception of vector_multiply with an input value of 1000.