David Gasperini (dlg195)

Devin Macalalad (dtm97)

CS 416

1. Logic
    a. <u>Mutex lock</u>: The user level mutex locking mechanism uses a user implemented test and set. Because we control the scheduler, we don't need to use assembly or the OS Test-and-Set function, so we implemented our own version of it, preventing a context switch during the atomic critical section of the condition variable. The mutex related methods are based on this. The functions will allow a blocked queue, so if a thread requests a resource it will enter a blocked queue, so that it will unblock when the resource is available and before any threads that enter the queue later.
    b. <u>Create</u>: The function will initialize a new thread, initializing the TCB structure and context, then add that thread to the scheduler, or create the scheduler and initialize the timer first if it is the first thread to run.
    c. <u>Yield</u>: Simply forfeits the CPU
    d. <u>Exit</u>: De-schedule, return some return value, and deallocate resources
    e. <u>Join</u>: Blocks until the completion of a specified thread and reaps the return value from it. The structures maintain a field that contains the threads joined on the current thread, to signal them when they can switch from blocked to ready and provide them with a return value.
    f. <u>Scheduler</u>: Our schedulers both use a list of queues (SJF only uses the highest priority queue), each containing a circular linked list of jobs to be scheduled that's serves as a run-queue
    g. <u>SJF</u>: For SJF, a time quantum is held inside of a TCB structure. Each time our timer goes off, the current running job's time quantum is incremented. When the scheduler is called, the job in the queue with the lowest quantum that is runnable (not yielding, blocked or destroyed), is added to the front of the queue to be context switched into. If the lowest quantum is the current running job, it is rescheduled. If the current running thread has yielded, joined, or been destroyed and has the lowest quantum, the next runnable job is scheduled
    h. <u>MLFQ</u>: For MLFQ, all 4 run queues are utilized (global array of run-queue pointers). Additionally, the each TCB structure has a value for amount of quantums it has gone without being scheduled. Each time a new job is scheduled, every job at the currently running queue and lower (higher queues are all blocked or destroyed since a lower one is running) that is ready to be scheduled has its quantums since it has been scheduled incremented. Next, the current running job is placed into a queue based on its quantum (4 runqueues,

they have quantums of 0-1, 1-2,2-4, then 4-8+). Then, the first runnable job from the queues is searched for, starting from the first in the highest priority to the last in the lowest priority. Once it is found, its quantums since last scheduled is set back to 0 and it is scheduled.

2. Benchmarks rpthread
    a. Running Parallel_cal gives the following times

Threads	Time (micro-s)
1	2506
2	2505
4	2506
8	2506
16	2507
32	2508

    b. Running External_cal gives the following times

Threads	Time (micro-s)
1	9075
2	9066
4	9067
8	9063
16	9071
32	9085

    c. Running Vector_multiply gives the following times

Threads	Time (micro-s)
1	26
2	27
4	28
8	27
16	31
32	59

3. Benchmarks pthread
    a. Running Parallel_cal gives the following times

Threads	Time (micro-s)
1	2524
2	1643
4	1054
8	660
16	555
32	526

User threads here are much slower than kernel threads probably because they split up the time within one process while the kernel threads use up multiple processer time slots, functioning as light weight processes. Only light-weight processes can run in true parallel.

    b. Running External_cal gives the following times

Threads	Time (micro-s)
1	9552
2	5226
4	4293
8	3887
16	3518
32	3547

The file IO time will be the same for both user and kernel threads, but once again the ability to parallel compute any information outside the critical section will provide a faster computation time, which can only be achieved by kernel threads.

    c. Running Vector_multiply gives the following times

Threads	Time (micro-s)
1	49
2	106
4	355
8	376
16	370

32	402

User rpthread library was much faster than the kernel pthread which got worse and worse with more threads. One reason for this could be because vectors are represented as arrays stored consecutively and having them all in the same process’s CPU cache could speed up the mathematics, where kernel threads would context switch.