CS236 Operating Systems Lab

Spring 2025-26

Lab 9: go parallel with pthreads

The goal of this lab is to gain insights into the pthreads library and its support for different types of synchronization primitives — mutexes, condition variables, and semaphores.

References —

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf
https://computing.llnl.gov/tutorials/pthreads/
http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

0. who are we — *threads*
what do we want — *a race*
when do we want it — *now!*

Refer to the file **`p0a.c`,** in the function **increment,** a global variable **total** is incremented by **x** (function argument) a million times. **`p0a.c`** repeats this 4 times, with **x** varying from 1 to 4. The final value of the global variable **total** is 10 million. Compile and run this file.

```
Shell
> ./p0a
Total: 10000000
```

The **race** is on.
Extend **p0a.c** to **p0b.c**. Instead of calling **increment** in a single thread four times, create 4 threads where each thread executes in parallel. Pass the value of **x** as an argument when invoking the thread. Output the extent by which ***each*** thread increments the variable **total** and also print the final total.

```
Shell
> ./p0b
Thread 1 incremented total by 1000000
Thread 2 incremented total by 2000000
Thread 3 incremented total by 3000000
Thread 4 incremented total by 4000000
Total: 3313774
```

Extend **`p0b.c`** to **`p0c.c`** and use mutex locks based synchronization to address the race condition situation.

```
Shell
> ./p0c
Thread 1 incremented total by 1000000
Thread 2 incremented total by 2000000
Thread 3 incremented total by 3000000
Thread 4 incremented total by 4000000
Total: 10000000
```

1. more pthreads!

The question covers some more usage of the **pthreads** API and the supported synchronization primitives — especially, condition variables, via the following task.

Refer to the file **task1.c,** which is part of this lab. The program reads two arrays from input files (input_q1a1.txt, input_q1b1.txt), and outputs the final sum of all the elements of the two arrays. To make this task faster, we can use multiple threads, each of which works on a certain section of the arrays. For example, if we have arrays of size 100 each, and 4 threads are used to parallelize the operation, then each thread adds 25 pairs of elements and stores 25 additions in the output array at each index of the input arrays. Once all the threads have finished execution, a main thread will sum the contents of the output array to generate the final total.

Implement the routine **addArrays()** to use a set of threads (specified as a command line argument) and distribute the array indices of both arrays A and B among those many threads.
Each thread performs `C[i] = A[i] + B[i]` for a subset of array indices **i**.
To measure the improvement achieved due to parallelism, run experiments varying the count of threads to measure the overall execution time of the program. The figure below shows performance with the number of threads varying from 1 to 6 threads.

Create a table / plot to show how the execution time varies on varying the number of threads.
Since the total sum of the output array is being calculated by the main thread, you are required to add some synchronization between the threads to obtain the correct total.

A program **task1.c** (part of the helper files) takes two arrays (file names for the array) and number of threads (among which the array indices are distributed) as command line arguments.
Note that **`task1sync`** is the binary of a solution. You can use it for sample outputs.

To compile the program: `gcc -o task1 task1.c -lpthread`

To run the program: `./task1 <inputfile1> <inputfile2> <num_of_threads>`

List of pthread APIs of interest —
`pthread_create, pthread_exit, pthread_join,`
`pthread_mutex_lock, pthread_mutex_unlock,`
`pthread_cont_wait, pthread_cond_signal, pthread_cond_broadcast`

Sample Output

```
Shell
labuser@synerg ~/c/l/task1> ./task1sync input_q1a4.txt input_q1b4.txt 1

sum of elements:109967103
Time spent: 68523.000000 us
labuser@synerg ~/c/l/task1> ./task1sync input_q1a4.txt input_q1b4.txt 4

sum of elements:109967103
Time spent: 38504.000000 us
```

* Demonstrate with additional output that your implementation is correct.

---

2. queue ready, threads are ready, what's the hold up?

This task (coded via **`task2.c`**) uses threads to emulate a scheduler per-CPU that consumes tasks from a ready queue.

A producer thread reads input from an input trace file that lists the workload (list of tasks and their description) and adds tasks to the ready queue. First line of the input file represents the number of items that producer has to read, after that each row has three numbers representing *<num_of_tasks, execution_time, next_arrival_time>* where **num_of_tasks** represents the number of tasks that have arrived, **execution_time** denotes the execution time of each task, and **next_arrival_time** denotes the duration after which the next tasks will arrive (whose description is in the next row). All times are in microseconds. Execution of a task on a CPU is emulated by a sleep corresponding to the execution time. Microsecond granularity sleep is via the **`usleep`** call.

Multiple CPUs are emulated using multiple threads. Each thread will dequeue one task at a time from the head of the queue and emulate its execution by sleeping for the desired duration specified for the task. Since multiple CPU threads will be accessing the same ready queue, synchronization is required.

Provided as part of this task are the boiler-plate code for the producer and consumer function without any synchronization and multi-threaded support. Complete the implementation of these functions to emulate working of the ready-queue and the scheduler.

Use your implementation to measure the following performance and runtime parameters —

- Number of lock requests per thread
- Total number of tasks per thread
- Number of requests dropped by the producer
- Average wait time of lock per thread
- Maximum waiting time of task per thread
- Minimum waiting time of task per thread
- Average waiting time of task per thread
- Average ready queue size

To compile the program: `gcc -o task2 task2.c -lpthread`

To run the program:
`./task2 <trace_file> <thread_count> <max_ready_queue_size>`

trace_file denotes the path of the trace file, **thread_count** denotes the number of CPUs to emulate and **max_ready_queue_size** denotes the maximum number of requests that can be stored in the ready queue.
/* descriptions are given in *task2.c* */

Plot a graph to show the relation between number of threads/CPUs and the average waiting time for acquiring locks per thread for different numbers of CPUs.

For timestamps and time estimation use the **clock_gettime API …**

```
struct timespec start, stop;
clock_gettime(CLOCK_MONOTONIC, &start);
somework();
clock_gettime(CLOCK_MONOTONIC, &stop);
accum = (stop.tv_sec - start.tv_sec ) * 1000000
          + ( stop.tv_nsec - start.tv_nsec ) / 1000;
```

Sample output

Shell

```
 ./task2 input_q2a.txt 4 1000
CPU ID : 0
Number of lock requests : 23
```

```
Number of Tasks done: 22
Average waiting time for lock : 0 us
Max waiting time of a task : 71 us
Min waiting time of a task : 1 us
Average waiting time of a task : 642 us

CPU ID : 1
Number of lock requests : 22
Number of Tasks done: 21
Average waiting time for lock : 0 us
Max waiting time of a task : 68 us
Min waiting time of a task : 0 us
Average waiting time of a task : 595 us

CPU ID : 2
Number of lock requests : 19
Number of Tasks done: 18
Average waiting time for lock : 1 us
Max waiting time of a task : 72 us
Min waiting time of a task : 3 us
Average waiting time of a task : 629 us

CPU ID : 3
Number of lock requests : 20
Number of Tasks done: 19
Average waiting time for lock : 0 us
Max waiting time of a task : 74 us
Min waiting time of a task : 16 us
Average waiting time of a task : 808 us

Number of drops done: 0
Average size of ready queue : 1
Total execution time for all the requests : 2185 us
Average waiting time for all threads 0.250000
```

Optional

3: just multi-queue it!

In this task, the single ready-queue(in task2) is replaced with a ready queue per CPU, i.e., multiple ready queues in the system (one per CPU).

Producer's work will be to distribute incoming tasks in a round-robin manner to all the ready-queues. The producer always starts distributing tasks from the first CPU onwards every time and takes input in a similar manner as described in task2.

Multiple CPU threads emulate their own scheduler by dequeuing one work at a time from the head of their own ready-queues and emulating its execution by sleeping for the desired duration specified for the work. Access on the shared ready-queues by each CPU and producer needs synchronization.
E.g., the following line is a line in the trace file on a 4 CPU emulation
6, 2, 1
Demand of 6 tasks, CPUs 1 and 2 get 2 tasks, CPUs 3 and 4 get 1 each.
The duration of work for each task is 2 units and the next work is coming up in 1 second for the producer to handle.

A load balancer thread (*migrationThread)* waits on the condition that if the ready queue of any CPU is zero, the migration thread moves requests from other ready queues to fill up the empty ready queues of other CPUs. The migration thread aims to balance the load across all ready queues.

Note that while the producer and migration thread access all the ready queues, each of the CPUs only accesses its own local ready queue.

Update the *producer()* ,*cpu(), migration_thread() etc.* routines (based on *task2.c)* to enqueue and dequeue the work from the shared ready queue to schedule and balance the work among all the CPUs in a synchronized way.

Use your implementation to measure the following performance and runtime params:
- number of lock requests per thread
- average wait time for lock per thread
- max, min and average wait time of tasks
- duration to dequeue all requests in the queue
- ready queue size per unit time per CPU
- number of tasks scheduled on each CPU, total number of tasks scheduled,
  number of tasks requesting CPU
- number of migration thread invocations
- number of tasks migrated out and in from each ready queue

Plot a graph to show the relation between number of threads/CPUs and the average waiting time of locks per thread for different numbers of CPUs.

Submission instructions

- All submissions are to be via Moodle only.
  Name your submission as `<rollnumber_lab9>.tar.gz` (e.g., `190050096_lab9.tar.gz`)

- The tar should contain the following files in the following directory structure:

```
<rollnumber_lab9>/
   |__task0
         |__p0a.c
         |__p0b.c
         |__p0c.c
   |__task1
         |__task1.c
   |
   __ task2
        |_task2.c
   |__task3 (optional)
        |__task3.c
```

- Your code should be well commented and readable.
- `tar -czvf <rollnumber_lab9>.tar.gz <rollnumber_lab9>`
