

# CS 333 Operating Systems Lab

## Autumn 2023

### 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>

#### Task 1: hello pthreads!

We begin with an introduction to the usage of the **pthreads** API and the supported synchronization primitives 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 pair 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.

```
• nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-1$ ./task1sync.o input_q1a4.txt input_q1b4.txt 1
sum of elements:109967103
Time spent: 46929.000000 us
• nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-1$ ./task1sync.o input_q1a4.txt input_q1b4.txt 2
sum of elements:109967103
Time spent: 32253.000000 us
• nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-1$ ./task1sync.o input_q1a4.txt input_q1b4.txt 3
sum of elements:109967103
Time spent: 31977.000000 us
• nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-1$ ./task1sync.o input_q1a4.txt input_q1b4.txt 4
sum of elements:109967103
Time spent: 28018.000000 us
• nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-1$ ./task1sync.o input_q1a4.txt input_q1b4.txt 5
sum of elements:109967103
Time spent: 31928.000000 us
• nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-1$ ./task1sync.o input_q1a4.txt input_q1b4.txt 6
sum of elements:109967103
Time spent: 29441.000000 us
```

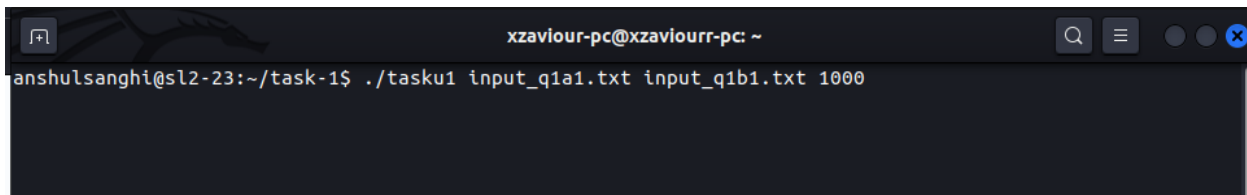
Implement the routine **addArrays()** to use the given number of threads (specified as a command line argument) and distribute the array indices of both arrays A and B among those many threads. To measure the improvement achieved due to parallelism, run several such experiments varying

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.

Try running the program with and without adding synchronization to see how the obtained output varies. This condition when you obtain different results because of a lack of synchronization mechanisms is called a **race condition**. Figure below shows outputs of the program for adding arrays without synchronization (program is stuck since the condition is never satisfied). The binary for this program is given, and is named **tasku1**, you can execute with different inputs to check output.

A terminal window with a dark background. The title bar shows 'xzaviour-pc@xzaviourrr-pc: ~'. The prompt is 'anshulsanghi@sl2-23:~/task-1\$'. The command entered is './tasku1 input\_q1a1.txt input\_q1b1.txt 1000'. The output is not visible, suggesting the program is stuck or has not yet produced output.

```
xzaviour-pc@xzaviourrr-pc: ~
anshulsanghi@sl2-23:~/task-1$ ./tasku1 input_q1a1.txt input_q1b1.txt 1000
```

To fix the race condition, implement a locking mechanism using pthread **mutexes** to get a correct count of the thread compute statues. Further, make the main thread use a pthread condition variable to sleep and wake up on the condition that all threads are done processing.

The program **task1.c** will take both arrays (file names for the array) and number of threads (among which the array indices are distributed) as command line arguments.

**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\_cond\_wait, pthread\_cond\_signal, pthread\_cond\_broadcast

#### [Optional]

Redo the above with semaphores. The semaphore API is part of `#include <semaphore.h>` and the calls are ... `sem_wait`, `sem_post`, `sem_init`, `sem_destroy` etc.

## **Task 2: just queue it!**

This task use 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 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 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/CPU's and the average waiting time for acquiring locks per thread for different numbers of CPU's.

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;
```

**Check the accompanying README file for a description of usage of commands to run different test cases and expected outputs.**

### Sample usage

```
nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-2$ ./task2.o input_q2a.txt 4 1000
CPU ID : 0
Number of lock requests : 22
Number of Tasks done: 22
Average waiting time for lock : 0 us
Max waiting time of a task : 426 us
Min waiting time of a task : 27 us
Average waiting time of a task : 181 us

CPU ID : 1
Number of lock requests : 24
Number of Tasks done: 24
Average waiting time for lock : 7 us
Max waiting time of a task : 568 us
Min waiting time of a task : 16 us
Average waiting time of a task : 222 us

CPU ID : 2
Number of lock requests : 18
Number of Tasks done: 18
Average waiting time for lock : 1 us
Max waiting time of a task : 415 us
Min waiting time of a task : 14 us
Average waiting time of a task : 189 us

CPU ID : 3
Number of lock requests : 16
Number of Tasks done: 16
Average waiting time for lock : 0 us
Max waiting time of a task : 420 us
Min waiting time of a task : 41 us
Average waiting time of a task : 217 us

Number of drops done: 0
Average size of ready queue : 3
Total execution time for all the requests : 3438 us
Average waiting time for all threads 2.000000
nikitarenu@nikitarenu-IdeaPad-3-14ITL6:~/IITBnew/TA-OS/lab 9/task-2$
```

---

## **Optional:**

### **Task 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/CPU's and the average waiting time of locks per thread for different number of CPU's.

---

## Submission instructions

- All submissions are to be via Moodle only.  
Name your submission as <rollnumber\_lab3>.tar.gz (e.g., 190050096\_lab3.tar.gz)
  - The tar should contain the following files in the following directory structure:

```
<rollnumber_lab9>/
|__task1
|    |__task1unsync.c
|    |__task1sync.c
|
|__ task2
|    |__task2.c
|__task3 (optional)
|    |__task3.c
```
  - Your code should be well commented and readable.
  - `tar -czvf <rollnumber_lab9>.tar.gz <rollnumber_lab9>`
-