CS 333: Operating Systems Lab
Autumn 2022
Lab 7: lock-and-key with pthreads

Goal: Understand pthread synchronization mechanisms — mutexes, CVs and semaphores.

0. Threads are fun!

We have looked at processes; now, let us look at threads. There are two programs named `processes.c` and `threads.c`. Both programs have the global variable **x**. In `processes.c,` **x** is incremented by 5 by the child process and then printed by both parent and child processes. Furthermore, in `threads.c`, two threads are created, **x** is incremented by 5 by a thread in the routine *`foo()`*, and both threads print the value of **x**.
Compile and run both programs and study outputs. Specifically, observe the process ids and thread ids in the output and explain the values of **x** printed by both programs in a **report**.

Note: You must use `-lpthread` flag while compiling your code containing the pthread library
(e.g. gcc threads.c -o threads.o -lpthread)

1. Where did the money go?

Task 1A: Don't touch my Money

For this task, refer to the file `addmillion.c`; the file has a routine named *`increment()`*, which the bank uses to increment the amount of `account_balance` for each deposit (banks do not repeatedly add 1, demonstrating a flaw mentioned later in the question). A deposit can only be made of 1 million. If anyone goes to their bank and makes ten such deposits, they expect the account balance to be 10 million because the `account_balance` is incremented by a sequential program. However, now say, ten people at ten different banks make one deposit in one account, the `bank_balance` is still expected to be 10 million. The program `addmillion.c` simulates ten different transactions at ten different banks using ten threads. Compile and run the program.

Is the output 10 million?
No! Where did the money go?
This condition is called a **race condition**.
Now implement a locking mechanism using pthread mutexes and get the desired 10 million as the `account_balance`.

Task 1B: The Sweet Spot

Further, modify the `addmillion.c` to take the number of threads as a command line argument. Depending on the number of threads, your program should make a valid deposit of 2048 million (e.g., for two threads, each thread should deposit 1024 million each). Understand how arguments are passed to threads and modify the `increment()` routine to take the number of million as input. (Create an outer loop for iterating over the for loop that is already present). Run the program with threads ranging from 2 -

1024 (in powers of 2) and analyze the time taken for each run. So does increasing the number of threads result in better performance?

Submit the plot of the time taken by the different runs and conclude the key takeaway from the task in the **report.** You should also submit the final `addmillion.c` in your submission.

Note:

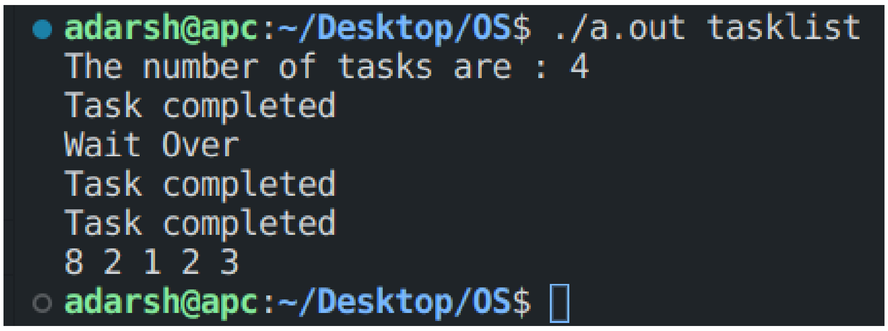
- You should measure time from the start of `main()` to program exit **WITHIN** the program and report it in the **EXACT** format: `Time spent: <time_taken> ms`
- Use the bash script `analysis.sh` to generate the plot.
- Install the prerequisites using: `sudo apt install plotutils`

2. Task server with a thread pool

For this question, you need to refer to the file `taskqueue.c`; the file contains self-explanatory global variables which are updated inside the routine *`processtask()`*, the program takes input from a file `tasklist`, and each line in the `tasklist` is a task or the time for which there is no task.
For each line:

I. A processing task (p) is written as **'p 2'** where p stands for processing task and 2 is the burst time (in seconds) for the task.
II. A waiting period (w) is written as **'w 1'**, where w stands for a waiting period, and 1 denotes the time (in seconds) for which there are no tasks.

For processing each task, the *`processtask()`* routine is called. The routine simulates a task by sleeping for the given task's burst time. With the given tasks in the `tasklist` file, the current sequential program will take 10 seconds. Moreover, it gives the following output:



```
adarsh@apc:~/Desktop/OS$ ./a.out tasklist
The number of tasks are : 4
Task completed
Wait Over
Task completed
Task completed
8 2 1 2 3
adarsh@apc:~/Desktop/OS$
```

In the output the last line denotes the final values of the global variables, after the processing is done by all processing tasks.

Implement a **<u>multi-threaded version</u>** of this program to reduce the time to process all the tasks by implementing the following:

- The main thread reads the number of worker threads as a command line argument and creates a pool of worker threads.
- The main thread reads the tasks one by one from the file and enters the task into a queue.
- A free thread from the thread pool picks up the task and processes it.
- Once all tasks are complete, the main thread joins the other threads and the global variables are printed.

Note: The first line of the `tasklist` file contains the number of tasks, followed by a task or waiting period.
Testing: To test your multithreaded version use the file `tasklistmultithreaded` as input.

3. Cricket goes threads!

There is a cricket ground in the city where at max 22 players can play a game of cricket, i.e., 11 players per side. Since the T20 World Cup is near, the ground sees much activity, and many players want to play a match. However, the ground can accommodate only 22 players at a given time.

Design a mechanism such that at a given time, only 22 players are allowed to enter the ground. Upon entering the ground, the first 11 players bat for team **Capitals** and the next 11 bat for **Titans** (Read bullet 4 of Structure). In the match, each player scores some runs simulated with a random number generator that can generate a number between 0-100. The team's score is the sum of all scores made by its players (i.e., the 11 players). At the end of each match, print the highest individual score in the match and the total runs scored by both teams and display the name of the winning team (the team with the higher score wins the match) along with the margin of victory.

When all matches have concluded display the summary for the day. Display the total matches played, match records for both the teams, Capitals and Titans, the highest team score for the day and the highest individual score (refer to sample output for format).

Implement the mechanism and provide two separate solutions using:

- Condition Variables & Mutexes: Provide the solution in `cricket-cv-mutex.c`
- Semaphores: Provide the solution in `cricket-semaphores.c`

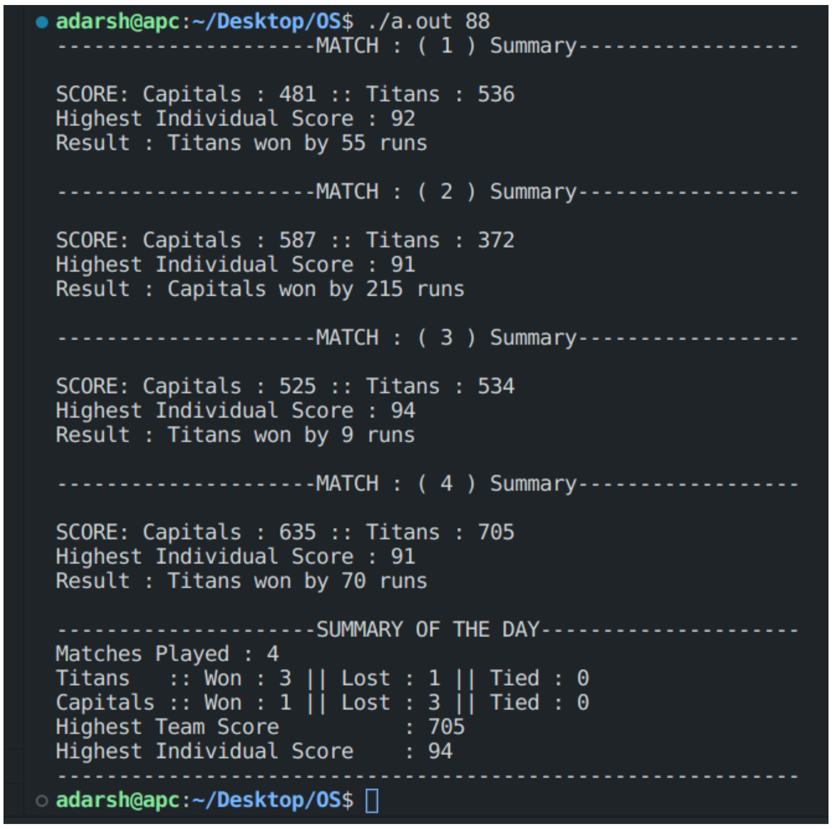
Assumptions:

- One player can play only one match.
- The number of matches that can be played will depend on the total number of players and is the floor of the number of players divided by 22. The program ends when the max number of matches that can be played is done. When the number of players is not a multiple of 22 then the match cannot be scheduled. So for input ./a.out 50, 2 matches can be scheduled and 6 players can not play a match so output the fact that match 3 cannot take place.

Structure of Solution:

- Each player is a thread
- The match starts when 22 players have entered the ground.
- You can create a game thread that handles the start and end of a game.
- When a thread enters, it checks the value of the Capitals counter variable to see if it is less than 11, if so, the thread bats (makes random runs), adds score to Capital score, and then exits. This happens for the first 11 threads, indicating that the capitals side has batted (all 11 players have made runs). When the capital counter reaches 11, the Titan's side is in, so the next 11 threads will bat (make runs), and they will add score to Titans.
- The match ends when both teams have batted after which they leave the ground.
- Only when all 22 players have left the ground the next set of 22 players can enter.
- At the completion of a match, all the players (threads) have left (joined in the main thread). New threads representing new players then play the next match.

Sample Output:
`$./a.out <number_of_players>`



```
adarsh@apc:~/Desktop/OS$ ./a.out 88
 --------------------MATCH : ( 1 ) Summary-----------------

 SCORE: Capitals : 481 :: Titans : 536
 Highest Individual Score : 92
 Result : Titans won by 55 runs

 --------------------MATCH : ( 2 ) Summary-----------------

 SCORE: Capitals : 587 :: Titans : 372
 Highest Individual Score : 91
 Result : Capitals won by 215 runs

 --------------------MATCH : ( 3 ) Summary-----------------

 SCORE: Capitals : 525 :: Titans : 534
 Highest Individual Score : 94
 Result : Titans won by 9 runs

 --------------------MATCH : ( 4 ) Summary-----------------

 SCORE: Capitals : 635 :: Titans : 705
 Highest Individual Score : 91
 Result : Titans won by 70 runs

 --------------------SUMMARY OF THE DAY--------------------
 Matches Played : 4
 Titans   :: Won : 3 || Lost : 1 || Tied : 0
 Capitals :: Won : 1 || Lost : 3 || Tied : 0
 Highest Team Score         : 705
 Highest Individual Score   : 94
 ----------------------------------------------------------
adarsh@apc:~/Desktop/OS$ 
```

Testing: You can use `testcricket.awk` to test your program's correctness (if it is free from deadlocks) like so

```
./a.out 660 | awk -f testcricket.awk
```

Submission Instructions

- All submissions are to be done on moodle only.
- Name your submissions as: `<rollnumber>_lab7.tar.gz` (e.g `123456_lab7.tar.gz`)
- The tar should contain the following files in the specified directory structure:

```
<rollnumber>_lab7/
          ├──── addmillion.c
          ├──── analysis.sh
          ├──── cricket-cv-mutex.c
          ├──── cricket-semaphores.c
          ├──── report.pdf
          ├──── tasklist
          ├──── tasklistmultithreaded
          ├──── taskqueue.c
          └──── testcricket.awk
```

Please adhere to this format strictly

- Your modified code/added code should be well commented on and readable.
- Command to compress your submission directory:
  `tar -zcvf <rollnumber>_lab7.tar.gz <rollnumber>_lab7`

Submission Due: 3rd Oct, 5 P.M. via moodle.