# CS3523: OS-II - ASSIGNMENT 2

**Aim: The aim of this assignment is to understand how classical IPC synchronization problems surface when developing concurrent programs.**

## Problem:

A kindergarten school wants to conduct musical chairs competition among "n" players who show interest in the competition. The game proceeds like this:

1. The school arranges n-1 chairs at random places in a ground.
2. n players will be standing in the ground at random places.
3. The game iterates n-1 times. In each of the $i^{th}$ lap,
    a. The umpire starts lap by ensuring that all players are ready.
    b. Once all players are ready the umpire starts the game by starting music.
    c. Each of the players keeps running around in the ground till music plays.
    d. When music is stopped by the umpire, each of the players quickly chooses a chair to occupy it. But if the chair is already occupied, the player steps back and chooses another free chair, and so on.
    e. One player will not get any chair as there is one less chair, so this player is out of game.
    f. The umpire stops the lap by ensuring that one player is knocked out, all other players are up from the chairs and a chair is removed from the game to start the next lap.

Write a program to mimic this game by creating one thread for the umpire and one thread for each of n players. The umpire thread works with all the players threads in lockstep synchronization. Each iteration, one player thread exits. Main thread waits for umpire thread and the n players to join back, and declares who has won the game.

## Input specification

Examples shown below has commands to control umpire and player's execution.

- Only the umpire thread reads these input commands.
- lap_start, lap_stop, music_start, music_stop are commands for letting player threads to synchronize with umpire and act on.
- "umpire_sleep <sleep>" which is an optional command specified between music_start and music_stop to make umpire run slower (just by the umpire sleep) between

music_start and music_stop for a specific lap. If umpire_sleep is not specified in any lap, by default umpire threads does not sleep between music_start and music_stop.

- “player_sleep <playerid> <sleep>” is another optional command specified between lap_start and music_start to make these player threads run slower (just by making them sleep) between music_start and music_stop for a specific lap. If player_sleep is not specified in any lap, by default player threads do not sleep between music_start and music_stop.

  If player_sleep is specified in a lap, but the player thread already exited, then the sleep time is ignored.

## Syntax:

```
musicalchairs --nplayers <n>
```

## Eg 1 (No sleep for players and umpire)-

```
$ cat input4randfast.txt
lap_start
music_start
music_stop
lap_stop
lap_start
music_start
music_stop
lap_stop
lap_start
music_start
music_stop
Lap_stop

$ ./a.out --np 4 < input4randfast.txt
Musical Chairs: 4 player game with 3 laps.
======= lap# 1 =======
2 could not get chair
**********************
======= lap# 2 =======
0 could not get chair
**********************
======= lap# 3 =======
3 could not get chair
**********************
Winner is 1
Time taken for the game: 1728 us
```

## Eg 2 (Sleeping umpire)-

```
$ cat input4rand.txt
lap_start
music_start
umpire_sleep 200
music_stop
lap_stop
lap_start
music_start
umpire_sleep 200000
music_stop
lap_stop
lap_start
music_start
umpire_sleep 800000
music_stop
lap_stop
```

```
$ ./musicalchairs --np 4 < input4rand.txt
Musical Chairs: 4 player game with 3 laps.
======= lap# 1 =======
2 could not get chair
**********************
======= lap# 2 =======
1 could not get chair
**********************
======= lap# 3 =======
3 could not get chair
**********************
Winner is 0
Time taken for the game: 1014484 us
```

## Eg 3 - (Sleeping players and sleepy umpire)

```
$ cat input4deterministic.txt
lap_start
player_sleep 0 1000
player_sleep 1 2000
player_sleep 2 3000
player_sleep 3 4000
music_start
umpire_sleep 200
music_stop
lap_stop
lap_start
player_sleep 0 1000
player_sleep 1 2000
player_sleep 2 3000
music_start
umpire_sleep 200000
music_stop
lap_stop
```

```
lap_start
player_sleep 0 1000
player_sleep 1 2000
music_start
umpire_sleep 800000
music_stop
lap_stop
```

```
$ ./a.out --np 4 < input4deterministic.txt
Musical Chairs: 4 player game with 3 laps.
======= lap# 1 =======
3 could not get chair
**********************
======= lap# 2 =======
2 could not get chair
**********************
======= lap# 3 =======
1 could not get chair
**********************
Winner is 0
Time taken for the game: 1012762 us
```

## Notes:

1. Group assignment: 2 students per group.
2. The timing shown in above examples are fictitious.
3. Do not print any output other than given in the input/output specification as autograder can only check strict outputs as given above. Any debug prints can confuse autograder and award 0 marks for test cases.
4. Do not write any computation logic in main().
5. Your program must use only C++ threads and C++ synchronization primitives. Eg. std::thread, std::mutex, std::unique_lock, std::condition_variable, std::atomic_flag, etc.
6. Your program should NOT use any C style process, thread, thread synchronization primitives in Linux eg. sem_overview(7), shm_overview(7), svipc(7), pthreads(7).
7. Measure cpu time elapsed for the entire game and return the time in microsecs to main.
7. You need to Strictly follow C coding standard of GNOME project (available at this link) and DON lab C standard written by Prof. Gonsalves for completing this assignment.
8. Create a README which should contain instructions on how to compile, run the program and sample outputs from your system. Do not submit object files, assembler files, or executables.
9. Create a report with your observations on games with different sleep quantum of umpire and players for various sizes of inputs. You need to run the program multiple times to observe no deadlocks, livelocks and starvation.
10. Marks of this assignment are divided for CODE files, test results, coding style and documentation (report+README).

11. main() must return either EXIT_SUCCESS or EXIT_FAILURE to shell. if exit value is not success (eg. thread or synchronization primitive creation error.), autograder will not match stdout, stderr.
12. Please do not overuse the autograder. Try to limit to three attempts per day. Also do not use autograder for unit-testing of your code.
13. Marks may be deducted for cooking output just to appease the autograder without real problem solution.

**Deliverables:**

1. A report describing how you completed above task(s). Measurements of cpu time elapsed and your observations. Make sure that your report is technically sound and readable.
2. README file which helps to know list of files submitted and how to compile and run your program
3. Upload the program, README, report to autograder.

**PLAGIARISM STATEMENT <Include it in your report>**

*We certify that this assignment/report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, We understand our responsibility to report honour violations by other students if we become aware of it.*

Name:

Date:

Signature: <keep your initials here>