Concurrency and Synchronization

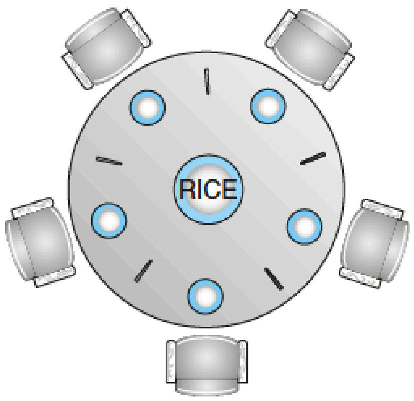
1 Objectives

Process synchronization and coordination is a central topic in Operating Systems. In running concurrent process, a major problem could occur: *race* condition. The use of sempahores, locks, conditions, and other constructs can help to synchronize concurrent processes and avoid such problems. This assignment is two-fold. The first part covers a solution to a variant of a classical problem in synchronization — the dining philosophers problem. In the second part, the objective is to solve a particular problem using different number of threads and to gain some insights about the performance of multithreading with increased number of threads.

2 Part 1 - Variant of Dining Philosophers with Waiter

The dining-philosophers problem is a classical problem in synchronization. It represents a large class of concurrenceny problems where a limited number of resources need to be allocated to a certain number of processes without deadlock nor starvation.

For simplicity, we consider five philosophers each sitting on a chair around a circular table. These philosophers spent their life only thinking and eating. The scenario is that a bowl of spaghetti/rice is placed in the middle of the table, a plate is given to each philosophers but only five single chopsticks are placed between these plates. Typically the philosophers spend their time thinking without interaction with others around the table. When a philosopher becomes hungry, s/he tries to pick up the two chopsticks that are closest (left and right) one at a time. When the hungry philosopher manages to pick up two chopsticks, s/he can eat without releasing the chopsticks. Only when s/he finished eating to begin thinking, s/he will put the chopsticks back on the table. A typical solution to this problem was discussed in class but it does not avoid deadlock. One solution in the literature is the Waiter Solution.



This provides a simple way to solve the Dining Philosophers problem, assuming an external entity called the waiter taking into account the following:

1. No chopsticks are placed on the table. Instead, a philosopher should request each of the two chopsticks from a waiter.
2. For convenience, we assume that all philosophers request their left chopstick first, then their right chopstick.
3. The waiter always provides chopsticks upon request unless only one chopstick remains unused. In that case, the waiter honors the request only if a right chopstick is requested; requests for a left chopstick are deferred until another philosopher finishes eating.

2.1 Variant of the Waiter solution

In this part you need to implement a monitor solution to the waiter approach of the dinning philosophers problem using semaphores. You need to consider the following in your implementation:

1. Suppose that we have 5 philosophers.
2. Assume that the waiter has n chopsticks where n takes an integer value from 5 to 10. Your program should take this value n as input.

3. Use a random number between 1 and 5 to select which philosopher changes its state from *Thinking* to *Hungry*. You may assume that the eating time for each philosopher is fixed to 5 seconds.

4. You can avoid starvation by not allowing a philosopher to eat again before all the other philosophers had their turn to eat.

5. Once every philosopher has eaten 3 times, your program should report the average amount of time that the philosophers had to wait to begin eating after becoming hungry.

3 Part 2 - Composite Numbers & Mutli-threading

In multithreading, multiple components of a program could be executed at the same time. These components are known as threads and are lightweight processes available within the process. So multithreading leads to maximum utilization of the CPU by multitasking. In this part you will apply multithreading on the problem of finding composite numbers and compare the performance of your solution using various number of threads.

A composite number[1] c is a number that has more than two factors, including 1 and c. For example, 12 is a composite number because it can be divided by $1, 2, 3, 4, 6$ and 12. So, the number 12 has 6 factors. One way to find if a number is a composite is to check if it can be divided by $2, 3, 5, 7, 11$, and 13. If it is even, we can start with division by 2, and if the number ends with 0 or 5, we can divide by 5.

In this part you need to do the following:

1. Write a program that reads a set of integers and determine how many of the input numbers are composite numbers.

2. Rewrite your program to using multi-threading. To do so you need to do the following:

   (a) Assume that your program uses n threads.

   (b) Distribute the work involved as equally as possible among these n threads.

   (c) Structure your program to read the number of threads as an optional parameter on a command line. For instance, if you program is Mycomposite, then you can run it with n threads using: Mycomposite -n

[1] The number 1 is not considered as composite number.

   (d) Run your program with different values of $n = 1, 2, 4, 8, 16$ and measure the turnaround time for each these values. For the timing involved, you can use the **time** command on linux shell.

```
time  ./MyComposite
```

3. Make sure that the range of numbers you use in the above is big enough for multi-threading effects to be observable.
4. Provide a brief performance report showing the time against the number of threads used, and comment on the expected computation time vs the usage of multiple threads.

4 To Submit

You need to submit source files, executables, along with a *Makefile* that will be used to compile and link your shell. You also need to submit a PDF detailing your report for both Part 1 and Part 2. Please include the reports in a single PDF.