Operating systems 2 project documentation

Project description:

The Sleeping TA project aims to simulate the interaction between Teaching Assistants (TAs) and students in an academic environment. The scenario involves a small office space where a TA provides assistance to students during office hours. The TA may be either helping a student, taking a nap when no students are present, or already engaged with another student. 3442

Project Components:

1. Office Space Configuration

The TA's office is a compact area with limited seating and resources:

Desk: A single desk equipped with a chair and a computer.

Hallway Chairs: Three chairs available outside the office for students waiting to see the TA. Each piece is then represented by a binary number consists of 5 bits, 3 bits for the piece and 2 bits for the move id which works with this piece on the board.

Team Members and Roles

GUI & Documentation: Yosef Ali and Hazim Abd Elhamed

Responsible for designing the graphical user interface and managing project documentation.

Students: Mahmoud Fatah and Muhamad Hany

Tasked with modelling student behaviour within the simulation, including arrival, interaction with TA, and waiting logic.

TA: Hossam Salah and Abdulla Hesham

Responsible for defining TA states, handling TA behaviour (sleeping, assisting, waiting), and managing synchronization with students.

Project Implementation:

Tools and Technologies Used:

Programming Language: Java
Concurrency Handling: Java Threads, Mutex Locks, Semaphores

Code Structure:

TASleeping.java:

Responsibility: Main orchestrator of student-TA interactions.
Description: Manages the main execution flow, coordinates student and TA interactions, and handles concurrency synchronization.

Student.java:

Responsibility: Representation of student threads interacting with the TA.
Description: Defines the behaviour and actions of student threads, including arrival, interaction with the TA, waiting logic, and synchronization mechanisms.

TeachingAssistant.java:

Responsibility: Models the TA as a thread.
Description: Implements the behaviour and states of the TA thread, including sleeping, assisting students, managing multiple student interactions, and synchronization mechanisms.

Work.java (GUI):

Responsibility: Handles the graphical user interface.
Description: Manages the GUI components, interfaces, and interactions for displaying the simulation, allowing user input, and representing the student-TA interactions visually.

MutexLock.java (Concurrency):

Responsibility: Implementation of mutex locks and synchronization mechanisms.
Description: Contains methods and structures for managing concurrency, mutex locks, semaphores, or other synchronization primitives necessary for coordinating interactions between students and the TA, ensuring thread safety and avoiding race conditions.

I. INTRODUCTION

Sleeping Teaching Assistant Problem: This is a story-based problem that helps in visualising the working of OS when multiple programs must access a shared resource. The problem states that: the teaching assistant must help the undergraduate students during office hours with their programming difficulties. The teaching assistant office is just a single room with a table and chair. When the teaching assistant is busy in assisting one student the other students must wait in the waiting area which is outside the office. If the chairs available in the waiting area are empty the student can occupy it and wait for their chance. Also, if all the chairs are occupied then the student decides to come later.

After assisting a student, teaching assistant checks if there is any student present in the waiting area. If the students are present, then the teaching assistant assists the next student. If no one is present, the teaching assistant takes a nap. If any student arrives when the teaching assistant is taking the nap then, student must wake up the teaching assistant for assistance.

From a technical point of view, this problem can be viewed as a critical section problem where each student is a process and the assistant's room is the critical section. In this paper, we have looked at the concepts required for the implementation of the above-mentioned problem.

II. PROCESS SYNCHRONISATION

The task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

It is mainly needed in a multi-process system when multiple processes are running together and more than one processes try to gain access to the same shared resources or any data at the same time.

A. Need for process synchronisation

Co-operating processes are processes in which one process can affect another process executing in the system. They can either directly share logical address space (both code and data) or share data through files or messages. When multiple processes are concurrently manipulating the data in the shared area, it can lead to data inconsistency.
Race condition is when several processes access and manipulate the same data concurrently and the outcome of execution depends on the particular order in which the access takes place. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. This is done through process synchronisation.

III. CRITICAL SECTION PROBLEM

A. Critical Section:

For every process, there is a common segment of code in which the process might be updating a common variable, updating a table, writing a file and so on. This section is called critical section. No two processes can execute in the critical section at the same time.

B. Critical Section Problem:

The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. As shown in Fig. 1, a critical section environment contains:

- Entry section code implements the request of entry into the critical section.
- Critical section code in which only one process can execute at any one time.
- Exit section is the end of the critical section, releasing or allowing others in.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Fig. 1. General structure of a typical process.

C. Solution to Critical section problem

The solution to critical section problem must satisfy three requirements:

1) Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2) Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not in the remainder section can participate in the decision on which will enter its critical section next, and this selection next cannot be postponed indefinitely.
3) Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Assume that each process executes at a nonzero speed.

No assumption concerning relative speed of the N processes.

Considering these, we compare three solutions to the critical section problem - a hardware-based solution: test and set lock and two software-based solutions: Peterson's solution and Semaphores.

1) Test-and-set lock: Test and Set Lock is a synchronization mechanism which uses a test and set instruction to provide synchronization among the processes executing concurrently. The test and set instruction is an instruction that

returns the old value of a memory location and sets the memory location value to 1 as a single atomic, i.e., an uninterruptible operation. The caller can then test the result to see if the state was changed by the call. When a process is currently executing a test and set, no other process is allowed to begin another test and set until the first process test and set is finished. The test and set lock always return the value sent to

```
boolean lock;

boolean TestAndSet (boolean &target)
{
      boolean rv = target;
      target = true;
      return rv;
}

while(1)
{
      while (TestAndSet(lock));
      critical section
      lock = false;
      remainder section
}
```

Fig. 2. Test and Set Algorithm

it and sets lock to true. The first process will enter the critical section since TestAndSet(lock) will return false, and will break out of the while loop.The other process will not be able to enter as the lock is set to true, and thus the while loop continues to be true. Mutual Exclusion is ensured. Once the first process gets out of critical section, lock is changed to false and now the other process can enter one by one. Progress is thus ensured. This solution thus satisfies two of the three required conditions to solve the critical section problems. However, after the first process, any process can go in. No queue is

maintained, any new process that finds the lock to be false can enter. So bound waiting is not ensured.

2) Peterson's solution:

Peterson's solution classic software-based solution to the critical section problem. Although it does not work well in the modern computer architecture, it provides a good algorithmic description of solving the problem illustrating some of the complexities involved in designing software that addresses the requirement of mutual exclusion, progress, and bounded waiting requirements. The algorithm works on just two processes where one process waits till the other gets out of the critical section.

```
bool flag[2] = {false, false};
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] == true && turn == 1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] == true && turn == 0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

Fig. 3. Peterson's algorithm

As shown in Fig 3 the algorithm uses two variables: flag and turn. A flag[n] value of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process P_0 if P_1 does not want to enter its critical section or if P_1 has given priority to P_0 by setting turn to 0.

3) Semaphore: Semaphore is a process synchronization tool proposed by Dijkstra in 1965. It is an integer value that is shared between threads to manage the resources that are required to be accessed by various processes in an operating system. A semaphore operates through two functions, wait ()

and signal(). The value of semaphore indicates whether a resource is available or not.

There are two types of semaphores namely binary semaphore and counting semaphore.

- Binary semaphore can have only two values, 0 and 1.
- Counting semaphore can take any value ranging from $-\infty$ to $+\infty$. It is used to control access to a resource that has multiple instances.

IV. ANALYSING THE SOLUTIONS

- Test and set lock: This is a hardware-specific solution and makes it difficult for programmers to come up with a generalised solution to the critical section problem. This also does not satisfy the bounded-waiting condition.
- Peterson's solution: This solution works only for two processes at a time. Since our problem requires minimum of three processes, this solution is not viable.

- Semaphore: It is machine independent and follows mutual exclusion efficiently and strictly than other methods. Semaphore also helps in handling multiple number of processes simultaneously.

Looking at the three methods, semaphore is a worthy candidate for the Sleeping Teaching Assistant problem.

V. SEMAPHORE IMPLEMENTATION

A semaphore is an integer variable S that, apart from initialisation can be modified only through two standard atomic operations: wait () and signal(), where the wait() operation occurs in the entry section of resources and signal() operation occurs in the exit section of the resources. wait () is often

represented as P() and signal as V(). All the modifications of the integer value of semaphore should be done indivisibly in wait () and signal(). When one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value.

```
P(Semaphore S){
        while(S<=0);
        //no operation
        S --;
}
```

```
V(Semaphore S){
        S++;
}
```

Fig. 4. Semaphore Working

If the resource required by the process is available, then the wait() operation decrements the semaphore value and let the process access the resource. If the resource required by the process is not available then the wait() operation decrements the value of semaphore and blocks the process.

If a process releases the resource then the signal() operation would increment the value of semaphore and unblocks a process in the waiting queue. [Fig 4]

There are two data members and two functions associated with each process in a waiting queue:

- A value of type integer.
- A pointer to the next process in the queue.
- Block – Function to add a process to an appropriate waiting queue in case the resource that the process is trying to invoke is not currently available.
- Wakeup – Function to remove a process from the waiting queue and to add it to the ready queue.

VI. IMPLEMENTATION OF SLEEPING TEACHING ASSISTANT

Considering all the conditions, we implemented the solution to the Sleeping Teaching Assistant problem in Java using semaphores and threads. Two semaphores are used to capture two events. One counting semaphore to keep track of the chairs and let the students in and another binary semaphore that says if the assistant is available. While the teaching assistant assists a student, other students are blocked, and the teaching assistant give preference to the student who is waiting for maximum time amongst all to assist next. If the chairs in the waiting area are occupied, the remaining students will be asked to start programming on their own till the teaching assistant becomes free. When there is no student to assist, the teaching assistant takes nap. If the assistant is sleeping, the first student to reach next will wake him/her up.

VII. ADVANTAGES AND DISADVANTAGES OF SEMAPHORE

A. Advantages

- Semaphore resolves the process synchronization issues.

- Waiting list associated with each semaphore avoids busy waiting and lets CPU perform other process productively.
- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the micro-kernel. So, they are machine independent.

B. Disadvantages

- Semaphore requires busy waiting, i.e., when a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively.
- Implementing semaphore can lead to priority inversion where the two processes get into spin-lock condition.
- If not implemented properly then semaphore can cause deadlock condition.

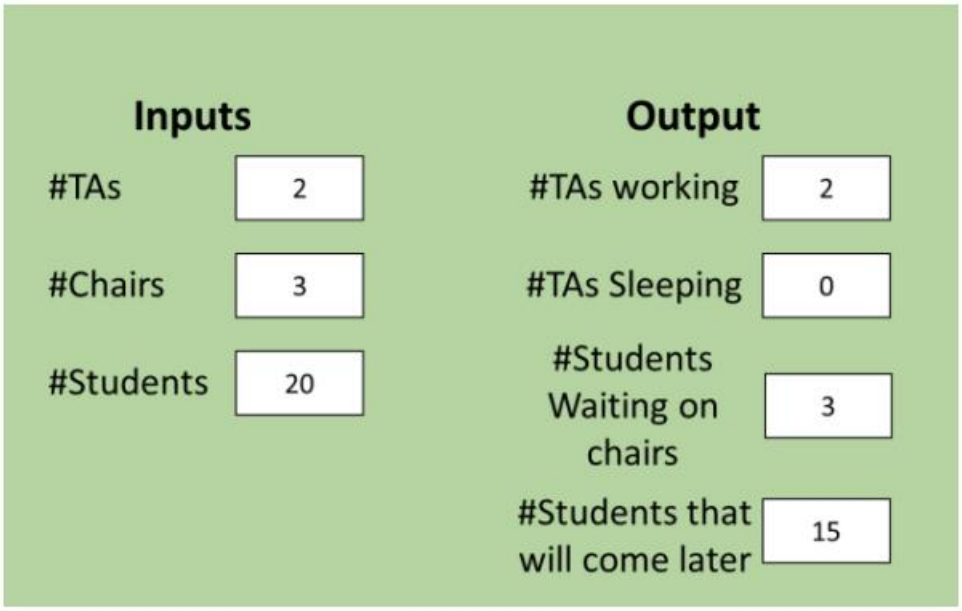
VIII. CONCLUSION

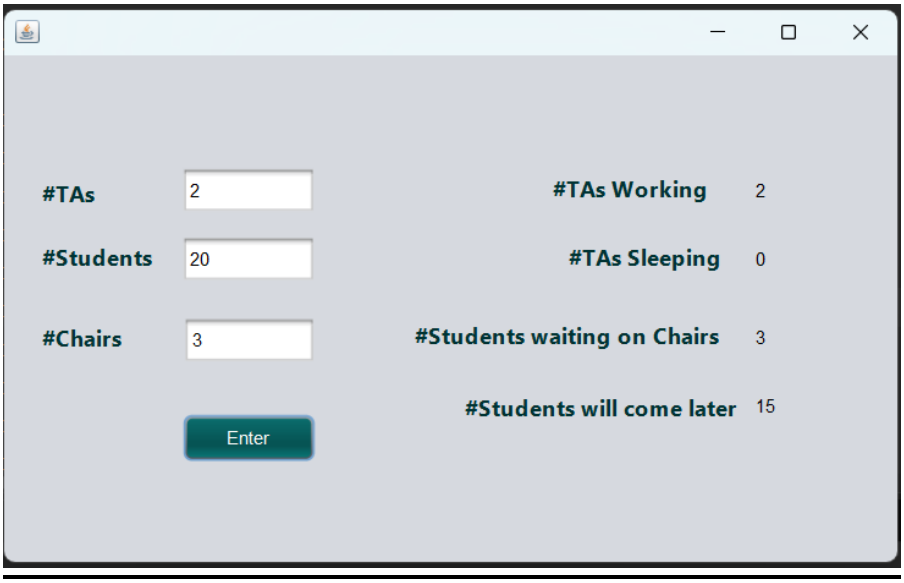
The Sleeping Teaching Problem was viewed as a Critical Section Problem, where each student represents a process and the teaching assistant's room is the critical section. The solution to the Sleeping Teaching Assistant problem was implemented using semaphores and threads. Three solutions for the Critical Section problem were explored, Test and Set Lock, Peterson's solution and Semaphore. Test and Set Lock only satisfies two of the three requirements in the critical section problem and Peterson's solution works only with two processes at time, hence Semaphore was chosen as the best solution. It is machine-independent and satisfies all the conditions of a solution of critical section problem. It can also handle multiple number of processes simultaneously.

IX. LIMITATIONS OF THE IMPLEMENTATION AND SCOPE OF IMPROVEMENT

- Our code currently does not check if the process really needs to access the critical section. The teaching assistant deals with one subject and he/she might not be able to help a student that seeks assistance for another subject. The student will be waiting in the queue for no gain if the teaching assistant cannot help the student. Hence, it is a waste of time and resources (the chair) because the chance of another student was wasted here. This could be handled by having a check before letting the student wait in the queue to meet the assistant.
- The code assumes that every student takes exactly 100 milliseconds to work out on a problem before asking help to the teacher, and that the teaching assistant takes exactly 100 milliseconds to assist the student. But in real-world scenario, this is not the case. So, we could improve the code by adding a random time assignment to the students before they can ask for help instead of a fixed 100ms.

Graphical user interface:




#TAs
2
#Students
20
#Chairs
3
Enter
#TAs Working
2
#TAs Sleeping
0
#Students waiting on Chairs
3
#Students will come later
15