

# **SLEEPING TEACHER ASSISTANT PROBLEM**

M ArunKumar

Roll no.18PT05

K A Kaushik Ram

Roll no.18PT16

## **Operating Systems**



**APRIL 2020**

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATION SCIENCES

**PSG COLLEGE OF TECHNOLOGY**

(Autonomous Institution)

**Coimbatore - 641 004**

---

---

## CONTENTS

CHAPTERS	PAGES
<b>ACRONYMS</b>	<b>I</b>
<b>1.Introduction</b>	<b>4</b>
<b>2.Description</b>	<b>5</b>
<b>3.System calls used</b>	<b>6</b>
<b>4.Concepts and Tools</b>	<b>9</b>
<b>5.Work Flow</b>	<b>11</b>
<b>6.Result and Discussion</b>	<b>15</b>
<b>7.Conclusion</b>	<b>16</b>
<b>8.Bibliography</b>	<b>17</b>

---

## **ACRONYMS**

STA	- Sleeping Teacher Assistant
TA	- Teacher Assistant
CR	- Critical Region
GC	- Gantt Chart
FIFO	- First In First Out
FCFS	- First Come First Serve

---

## CHAPTER 1

### INTRODUCTION

In this report we will discuss different cases for the STA problem and possible outcomes of each solution and the optimised solution for the problem using threads, mutex locks, and semaphores. The STA problem is a real time application of process synchronization. There are many other problems that can be solved using the concept of Mutual exclusion and Semaphores such as Dining philosopher problem and Producer Consumer problem. If we try to implement the solution for these problems using threads alone, we might come across the problem of multiple processes accessing the same memory and this leads to inconsistency of the values stored in a variable. So we have to restrict access to the same variable by more than one process. The concurrent operations can be done without inconsistency of data by using the concepts like critical section and mutual exclusion.

---

## CHAPTER 2

### DESCRIPTION

A university computer science department has a TA who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the STA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

---

## CHAPTER 3

### SYSTEM CALLS USED

A description of the system calls that are used in the solution for the problem is discussed in this chapter. It gives an idea and overview of the system calls, and why are they used.

#### 3.1 Open

The open system call is used to open a new file and obtain its file descriptor. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process. This file descriptor can be used to denote the file in the subsequent system calls. A call to open creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags. A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file.

The new open file description is initially not shared with any other process, but sharing may arise through some other system call. The files can be open in different modes such as read, write or both combined. Each mode has its own flag denoting the operation. We use the open system call to open or create if not available, the file given as user input. This file descriptor that is obtained successfully is used for manipulating the file in the program.

---

## 3.2 Close

The close system call closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks held on the file it was associated with, and owned by the process, are removed regardless of the file descriptor that was used to obtain the lock.

If *fd* is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using unlink, the file is deleted. Close, returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

The close system call is used to close the file that we have used to display the output as per the user command.

## 3.3 Dup2

dup2 creates a copy of the file descriptor. After a successful return from dup2, the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of the descriptors, the offset is also changed for the other.

The two descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag for the duplicate descriptor is off. dup2 makes *new fd*

---

be the copy of *old fd*, closing *new fd* first if necessary. `dup2` returns the new descriptor, or -1 if an error occurred and the *errno* is set appropriately.

The `dup2` system call is used to duplicate the file descriptor of standard output into the file descriptor of the given output file. So when the output is printed on standard output, it is redirected to the given output file.



---

## **CHAPTER-4**

### **CONCEPTS AND TOOLS USED**

The tools and concepts that are used to solve this problem are discussed in this chapter.

#### **4.1 CONCEPTS**

The various concepts that have been used for solving the problem are discussed here.

##### **4.1.1 THREADS**

A thread is a light weight process and has its path of execution within a process. A process can contain multiple threads. The idea is to achieve parallelism by dividing a process into multiple threads.

In our problem we create N threads for students and one separate thread for TA.

##### **4.1.2 MUTEX LOCKS**

Mutex locks are used for Thread Synchronisation. It ensures that two or more consequent processes(Threads) do not simultaneously execute some particular program segment known as CR. In our problem we use Mutex\_Locks for locking and unlocking chair access for the students.

---

### 4.1.3 SEMAPHORES

One of the solutions to manage concurrent processes entering into the CR by using Simple integer value, which is called Semaphore.

There are 2 Semaphores:

- Binary Semaphore.
- Counting Semaphore

In our problem we use Binary Semaphore which has only binary values(0 and 1).

### 4.2 TOOLS

The C programming language is used to program and simulate the problem with possible use cases. The gcc compiler is used to compile the program. The C language has versatile libraries that support multiple threads , thread synchronization and solutions for other CR problems.

---

## CHAPTER-5

### WORKFLOW

The workflow of the program is discussed in this chapter. We will discuss where the concepts that we have seen earlier are applied in the program and the detailed solution.

#### 5.1 Initialization

The initial conditions are given below. The team uses three semaphore variables, and a mutex lock for chair access. Initially, the semaphores and the variable are given the following initial values.

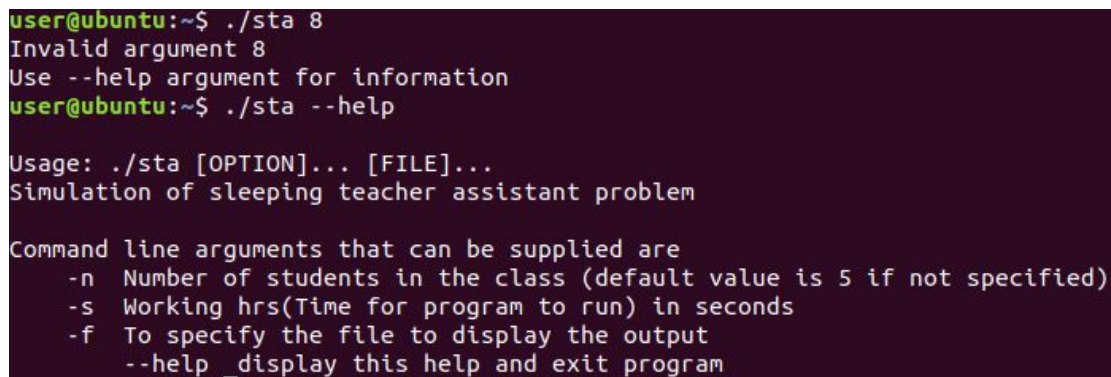
- Student = 0
- TA = 0
- Chair Access = 1
- Waiting students = 0

From the initial values, it is clear that no student is there at the start and the teaching assistant is sleeping. The chairs are free, so if any student arrives, he can occupy the chair. After the initialization of these variables, the user input is taken as a command line argument and the threads for 'n' students and one TA are allocated the required resources and they start to run in parallel. If the stop time is specified, the program will run only for the specified time, and after that the threads are terminated and the memory allocated is freed.

---

## 5.2 Command line arguments

The command line arguments are used to supply the required input for the program. The next part of the program is to parse the command line arguments and get the required input from them. Figure 5.1 one shows a demo run of the program and it displays all available command line argument options.

A terminal window with a dark purple background. The prompt is 'user@ubuntu:~\$'. The first command is './sta 8', which results in the output 'Invalid argument 8' and 'Use --help argument for information'. The second command is './sta --help', which displays the usage and a list of command line arguments.

```
user@ubuntu:~$ ./sta 8
Invalid argument 8
Use --help argument for information
user@ubuntu:~$ ./sta --help

Usage: ./sta [OPTION]... [FILE]...
Simulation of sleeping teacher assistant problem

Command line arguments that can be supplied are
  -n Number of students in the class (default value is 5 if not specified)
  -s Working hrs(Time for program to run) in seconds
  -f To specify the file to display the output
  --help _display this help and exit program
```

**Figure 5.1** Command Line arguments

The -n option is used to give the number of the students in the class. The -s option is used to give the stop time of the program in seconds. It is clear that even if the stop time occurs and a student is currently clearing doubt, the teacher cannot leave the student immediately, so he waits till the doubt is cleared. The -f option is used to display the output of the program in the specified file.

## 5.3 Possible Cases

The following four possible cases of the problem are handled in the solution.

---

### **5.3.1 CASE 1:**

There will be zero students coming to visit the TA and the TA will check the hallway outside his office to see if there are any students seated and waiting for him. If there are none, the TA will sleep in his office.

### **5.3.2 CASE 2:**

When a student arrives at the TA's office and finds the TA sleeping. Then the student will awaken the TA and ask for help. When the TA assists the student, the student's semaphore changes from 0 to 1 and waits for the TA's semaphore. When the TA finishes helping one student, he will check if there is any other student waiting in the hallway. If yes, he will help the next student and if not, TA goes back to sleeping and TA's semaphore becomes 1 and awaits the student's semaphore.

### **5.3.3 CASE 3:**

When a student arrives while the TA is busy with another student. Then the student who arrived will have to check if the TA is busy. If the TA is busy, the student will have to wait seated outside in the hallway until the TA is done with his session. When the TA completes his session, the student seated outside will be called in by the TA for a review session. Once all students have finished their sessions and left the TA's office, the TA will go back to sleep after making sure no students are waiting.

---

#### **5.3.4 CASE 4:**

When a student arrives while the TA is busy in a review session, and all the seats in the hallway are occupied. Then, students will have to leave the hallway and come back later. When the student comes back, eventually, and there is a seat available, he will take a seat and wait for his turn with the TA.

---

## **CHAPTER-6**

### **RESULT AND DISCUSSIONS**

There are many possible solutions for the STA problem. Each solution for different scenarios serves its own purposes. In our result the motive is to achieve Deadlock avoidance and Starvation avoidance for an optimised result. This chapter deals with Deadlock and Starvation avoidance.

#### **6.1 DEADLOCK AVOIDANCE**

Both the TA and students will have their own semaphores and separate threads. By using a mutex-lock the situation of a deadlock can be avoided as either the TA or the student will have exclusive rights to change their state and this can be accomplished by only one person at a given instance that is only one process will be in CR at a time.

#### **6.2 STARVATION AVOIDANCE**

By using a First-In First-Out queue, starvation can be avoided as we know every student will be served according to the time they arrive at the TA's office. Therefore, no student will be able to skip the line. The scheduling done is similar to the FCFS scheduling. The team approaches the Teaching Assistant Problem by solving 4 different scenarios as discussed earlier.

---

## **CHAPTER-7**

### **CONCLUSION**

The STA problem has been successfully completed along with all possible cases and optimised solution. There have been no errors regarding the solution and implementation part and also its user friendly. The project helped us gain more knowledge on process synchronisation and various implementation of GC. Also, the project made us clearly understand what are semaphores, mutex locks and how they work. It gave us opportunities to learn how to present a Report. Overall, the project gave a complete view of process synchronisation and scheduling students using FCFS algorithm and FIFO queue.



---

## BIBLIOGRAPHY

### WEBSITES:

1. <https://www.freelancer.in/projects/c-programming/sleeping-teaching-assistant-9029750/>
2. <https://www.geeksforgeeks.org/python-basic-gantt-chart-using-matplotlib/>
3. <https://www.tutorialspoint.com/semaphores-in-operating-system>
4. [http://www.uobabylon.edu.iq/download/M.S%202013-2014/Operating\\_System\\_Concepts,\\_8th\\_Edition%5BA4%5D.pdf](http://www.uobabylon.edu.iq/download/M.S%202013-2014/Operating_System_Concepts,_8th_Edition%5BA4%5D.pdf)
5. <https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>
6. <https://www.softprayog.in/programming/posix-threads-synchronization-in-c>