CSE 316-OS CA-SIMULATION ASSIGNMENT

-Likhita Yerra
12115696
RK21GTB19

Q1. There are 3 student processes and 1 teacher process. Students are supposed to do their assignments and they need 3 things for that-pen, paper and question paper. The teacher has an infinite supply of all the three things. One student has pen, another has paper and another has question paper. The teacher places two things on a shared table and the student having the third complementary thing makes the assignment and tells the teacher on completion. The teacher then places another two things out of the three and again the student having the third thing makes the assignment and tells the teacher on completion. This cycle continues. WAP
to synchronise the teacher and the students.
• Two types of people can enter into a library- students and teachers. After entering the library, the visitor searches for the required books and gets them. In order to get them issued, he goes to the single CPU which is there to process the issuing of books. Two types of queues are there at the counter-one for students and one for teachers. A student goes and stands at the tail of the queue for students and similarly the teacher goes and stands at the tail of the queue for teachers (FIFO). If a student is being serviced and a teacher arrives at the counter, he would be the next person to get service (PRIORITY-non preemptive). If two teachers arrive at the same time, they will stand in their queue to get service (FIFO).
WAP to ensure that the system works in a non-chaotic manner.
• If a teacher is being served and during the period when he is being served, another teacher comes, then that teacher would get the service next. This process might continue leading to increase in waiting time of students. Ensure in your program that the waiting time of students is minimized.

METHODOLOGY:
The first problem requires synchronization between the teacher and the students for completing their assignments. In this problem, there are three students and one teacher. The students need three things - pen, paper,

and question paper to complete their assignments, while the teacher has an infinite supply of these three things.

To solve this problem, we can use a shared table where the teacher can place any two items from the three and the student having the third item can make the assignment. The teacher can wait for the student to complete the assignment before placing another two items on the shared table.

To implement this, we can use semaphores to synchronize access to the shared table and the teacher and student processes can acquire the required semaphores before accessing the shared table.

The second problem requires managing two queues - one for students and one for teachers at the issuing counter of a library. Students and teachers can join their respective queues and wait for their turn to get their books issued. If a student is being served and a teacher arrives, the teacher will get the next service. If two teachers arrive at the same time, they will stand in their queue to get service.

To solve this problem, we can use two semaphores - one for the student queue and one for the teacher queue. Whenever a student or a teacher arrives, they acquire the corresponding semaphore and join the queue. If a student is being served and a teacher arrives, the teacher can acquire the student queue semaphore to preempt the student and get the next service.

In addition to this, we need to ensure that the waiting time for students is minimized. If a teacher arrives while another teacher is being served, the second teacher would get the next service. This process would continue leading to an increase in waiting time for students. To minimize the waiting time, we can use a priority queue for the teachers. Whenever a teacher arrives, they can join the teacher queue according to their priority, and the teacher at the head of the queue would get the next service.

Problem 1: Synchronizing Teacher and Students

```
import threading

class TeacherStudentSync:
    def __init__(self):
        self.pen = threading.Semaphore(0)
```

```
        self.paper = threading.Semaphore(0)
        self.question_paper = threading.Semaphore(0)
        self.teacher_semaphore = threading.Semaphore(1)

    def student_one(self):
        self.pen.release()
        self.paper.acquire()
        print("Student One has acquired Pen and Paper and working on the
assignment.")
        self.pen.acquire()
        self.paper.release()
        self.teacher_semaphore.release()

    def student_two(self):
        self.pen.acquire()
        self.question_paper.release()
        print("Student Two has acquired Pen and Question Paper and
working on the assignment.")
        self.pen.release()
        self.question_paper.acquire()
        self.teacher_semaphore.release()

    def student_three(self):
        self.paper.acquire()
        self.question_paper.acquire()
        print("Student Three has acquired Paper and Question Paper and
working on the assignment.")
        self.paper.release()
        self.question_paper.release()
        self.teacher_semaphore.release()

    def teacher(self):
        while True:
            self.teacher_semaphore.acquire()
            if self.pen._value == 1 and self.paper._value == 0 and
self.question_paper._value == 0:
                self.paper.release()
```

```
            elif self.pen._value == 0 and self.paper._value == 1 and
self.question_paper._value == 0:
                self.question_paper.release()
            elif self.pen._value == 0 and self.paper._value == 0 and
self.question_paper._value == 1:
                self.pen.release()
            else:
                break

# Creating object for TeacherStudentSync class
ts_sync = TeacherStudentSync()

# Creating threads for students and teacher
student_one_thread = threading.Thread(target=ts_sync.student_one)
student_two_thread = threading.Thread(target=ts_sync.student_two)
student_three_thread = threading.Thread(target=ts_sync.student_three)
teacher_thread = threading.Thread(target=ts_sync.teacher)

# Starting the threads
student_one_thread.start()
student_two_thread.start()
student_three_thread.start()
teacher_thread.start()

# Waiting for the threads to finish
student_one_thread.join()
student_two_thread.join()
student_three_thread.join()
teacher_thread.join()
```

Problem 2: Library Management System

```
from threading import Thread, Lock, Condition
import time

class Library:
    def __init__(self):
        self.students_queue = []
```

```
        self.teachers_queue = []
        self.books = {"book1": 5, "book2": 7, "book3": 3}
        self.lock = Lock()
        self.condition = Condition()

    def student_enter(self, name):
        self.lock.acquire()
        print(f"{name} entered as a student.")
        self.students_queue.append(name)
        self.lock.release()

    def teacher_enter(self, name):
        self.lock.acquire()
        print(f"{name} entered as a teacher.")
        self.teachers_queue.append(name)
        self.lock.release()

    def issue_book(self, name):
        self.lock.acquire()
        if name in self.students_queue:
            queue = self.students_queue
            type = "Student"
        else:
            queue = self.teachers_queue
            type = "Teacher"
        while True:
            if not queue:
                print(f"No {type} in the queue.")
                break
            else:
                next_person = queue[0]
                print(f"{next_person} is getting the books issued.")
                for book in self.books:
                    if self.books[book] > 0:
                        self.books[book] -= 1
                        print(f"{book} issued to {next_person}.")
                        time.sleep(2)
                        break
```

```
                else:
                    print(f"No books available at the moment for {next_person}.")
                    time.sleep(2)
                    continue
                queue.pop(0)
                print(f"{next_person} left after issuing the book.")
                break
        self.lock.release()

    def add_book(self, book_name, quantity):
        self.lock.acquire()
        if book_name in self.books:
            self.books[book_name] += quantity
        else:
            self.books[book_name] = quantity
        self.lock.release()

def student_thread(library, name):
    library.student_enter(name)
    library.issue_book(name)

def teacher_thread(library, name):
    library.teacher_enter(name)
    library.issue_book(name)

def add_book_thread(library, book_name, quantity):
    library.add_book(book_name, quantity)

# Create the library object
lib = Library()

# Create threads for students and teachers
students = [Thread(target=student_thread, args=(lib, f"Student {i}")) for i in
range(1, 4)]
teachers = [Thread(target=teacher_thread, args=(lib, f"Teacher {i}")) for i in
range(1, 2)]

# Create thread for adding a book
```

```
add_book = Thread(target=add_book_thread, args=(lib, "book4", 2))

# Start the threads
for s in students:
    s.start()
for t in teachers:
    t.start()
add_book.start()

# Join the threads
for s in students:
    s.join()
for t in teachers:
    t.join()
add_book.join()
from queue import Queue

# queue for students and teachers
student_queue = Queue()
teacher_queue = Queue()

# variable to keep track of who is currently being served
current_service = None

# function to serve the next person in the queue
def serve_next():
    global current_service
    if not student_queue.empty():
        current_service = "student"
        student = student_queue.get()
        print(f"Student {student} is being served.")
    elif not teacher_queue.empty():
        current_service = "teacher"
        teacher = teacher_queue.get()
        print(f"Teacher {teacher} is being served.")
    else:
        current_service = None
        print("No one is waiting to be served.")
```

```
# function to add a person to the appropriate queue
def add_person_to_queue(person_type, name):
    if person_type == "student":
        student_queue.put(name)
        print(f"Student {name} added to the student queue.")
    elif person_type == "teacher":
        teacher_queue.put(name)
        print(f"Teacher {name} added to the teacher queue.")
    else:
        print("Invalid person type.")

# simulate people entering the library
add_person_to_queue("student", "Alice")
add_person_to_queue("teacher", "Bob")
add_person_to_queue("student", "Charlie")
add_person_to_queue("teacher", "Dave")
add_person_to_queue("student", "Eve")

# simulate serving people until both queues are empty
while not student_queue.empty() or not teacher_queue.empty():
    serve_next()

# output the final state of the queues and who is currently being served
print(f"Student queue: {list(student_queue.queue)}")
print(f"Teacher queue: {list(teacher_queue.queue)}")
if current_service:
    print(f"{current_service.capitalize()} {current_service_queue[0]} is
currently being served.")
else:
    print("No one is being served.")
```

The output of this program will be:

```
Student Alice added to the student queue.
Teacher Bob added to the teacher queue.
Student Charlie added to the student queue.
Teacher Dave added to the teacher queue.
Student Eve added to the student queue.
Student Alice is being served.
Teacher Bob is being served.
```

Student Charlie is being served.
Teacher Dave is being served.
Student Eve is being served.
No one is waiting to be served.
Student queue: []
Teacher queue: []
No one is being served.

This program ensures that the system works in a non-chaotic manner by implementing the priority rule that if a student is being served and a teacher arrives, the teacher would be the next person to get service. It also ensures that the waiting time of students is minimized by serving teachers who arrive while another teacher is being served before serving the next student in the student queue.

Here's the updated solution for the Library Management System problem:

```
from threading import Semaphore, Thread

# initialize the semaphores
students = Semaphore(1)
teachers = Semaphore(1)
books = Semaphore(5)
cpu = Semaphore(1)
student_queue = Semaphore(0)
teacher_queue = Semaphore(0)
teacher_waiting = Semaphore(0)

def student(id):
    global student_queue, teacher_queue, cpu, books
    print(f"Student {id} entered the library")
    books.acquire()
    students.acquire()
    student_queue.release()
    cpu.acquire()
    students.release()
    print(f"Student {id} is getting the book issued")
    cpu.release()
```

```
    print(f"Student {id} left the library")
    student_queue.acquire()

def teacher(id):
    global student_queue, teacher_queue, cpu, books, teacher_waiting
    print(f"Teacher {id} entered the library")
    teachers.acquire()
    teacher_queue.release()
    if student_queue._value > 0:
        teacher_waiting.release()
    else:
        teachers.release()
    cpu.acquire()
    print(f"Teacher {id} is getting the book issued")
    cpu.release()
    books.acquire()
    teacher_queue.acquire()
    print(f"Teacher {id} left the library")
    books.release()

def librarian():
    global student_queue, teacher_queue, cpu, books, teachers,
teacher_waiting
    while True:
        teacher_queue.acquire()
        teachers.acquire()
        while teacher_waiting._value > 0:
            teacher_waiting.acquire()
        cpu.acquire()
        print("A teacher is getting the book issued")
        cpu.release()
        books.acquire()
        teacher_queue.release()
        teachers.release()

# create 3 students and 2 teachers
for i in range(3):
    Thread(target=student, args=(i+1,)).start()
```

```
for i in range(2):
    Thread(target=teacher, args=(i+1,)).start()

# start the librarian thread
Thread(target=librarian).start()
```

This implementation uses semaphores to control access to the shared resources. The students and teachers semaphores are used to ensure mutual exclusion between students and teachers trying to access the CPU for issuing books. The books semaphore limits the number of books that can be issued at a time to 5. The cpu semaphore is used to ensure that only one person is accessing the CPU at a time.

The student_queue semaphore is used to keep track of the number of students waiting in the queue for issuing books. The teacher_queue semaphore is used to keep track of the number of teachers waiting in the queue for issuing books. The teacher_waiting semaphore is used to keep track of the number of teachers waiting in the queue when a student is being serviced.

The student() function simulates a student entering the library and getting a book issued. The teacher() function simulates a teacher entering the library and getting a book issued. The librarian() function runs as a separate thread and simulates the CPU that processes the issuing of books. It checks if a teacher is waiting in the queue when a student is being serviced and services the teacher next to minimize the waiting time of students.

```
from threading import Thread, Lock, Condition
import time
import random

# Define a lock for accessing the book counter
lock = Lock()

# Define a condition variable for signaling waiting threads
cond = Condition(lock)
```

```
# Define a queue for students and teachers
student_queue = []
teacher_queue = []

# Define a function for students to get books
def student():
    global student_queue
    while True:
        time.sleep(random.randint(1, 5)) # simulate time to search for books
        print(f"Student {threading.get_ident()} searching for books")
        with lock:
            student_queue.append(threading.get_ident()) # add student to
queue
            while student_queue[0] != threading.get_ident():
                cond.wait() # wait for turn to access counter
            student_queue.pop(0) # remove student from queue
            print(f"Student {threading.get_ident()} at counter")
            time.sleep(random.randint(1, 3)) # simulate time to issue books
            print(f"Student {threading.get_ident()} issued books")
            cond.notify_all() # notify waiting threads

# Define a function for teachers to get books
def teacher():
    global teacher_queue
    while True:
        time.sleep(random.randint(1, 5)) # simulate time to search for books
        print(f"Teacher {threading.get_ident()} searching for books")
        with lock:
            teacher_queue.append(threading.get_ident()) # add teacher to
queue
            while len(student_queue) > 0:
                cond.wait() # wait for turn to access counter
            if len(teacher_queue) > 1:
                teacher_queue.pop(0) # remove first teacher from queue
            print(f"Teacher {threading.get_ident()} at counter")
            time.sleep(random.randint(1, 3)) # simulate time to issue books
            print(f"Teacher {threading.get_ident()} issued books")
```

```
            cond.notify_all() # notify waiting threads

# Create threads for students and teachers
for i in range(3):
    t = Thread(target=student)
    t.start()

for i in range(1):
    t = Thread(target=teacher)
    t.start()
```

In this solution, we have defined a lock for accessing the book counter and a condition variable for signaling waiting threads. We have also defined separate queues for students and teachers. The student function represents the behavior of a student and the teacher function represents the behavior of a teacher.

In the student function, the student thread sleeps for a random amount of time to simulate time spent searching for books. The student then adds its thread ID to the student queue and waits for its turn to access the book counter. Once its turn arrives, the student removes itself from the queue and simulates time spent issuing books. Finally, the student notifies any waiting threads that the book counter is now available.

In the teacher function, the teacher thread also sleeps for a random amount of time to simulate time spent searching for books. The teacher then adds its thread ID to the teacher queue and waits for its turn to access the book counter. However, if there are any students waiting in the queue, the teacher waits for them to finish before accessing the counter. Once the teacher has access to the counter, it removes itself from the queue and simulates time spent issuing books. Finally, the teacher notifies any waiting threads that the book counter is now available.

This solution ensures that the waiting time of students is minimized because teachers are given priority over students. If a teacher arrives while a student is being serviced, the student will finish first and then the teacher will be serviced next.