CSA-0460

ASSIGNMENT-1

(11)

U. Jayakrishna
192821060

The classical synchronization problems in operating systems refers to well-known problems used to demonstrate and solve issues related to process synchronization, mutual exclusion, deadlock and resource sharing.

producer - consumer problem (Bounded Buffer problem)

A producer thread generates data and puts in a buffer, while a consumer thread takes data from the buffer. The buffer has a limited capacity, so synchronization is required to avoid overfilling or underflowing.

10/10

Pseudo-code:

```
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
void producer() {
    while (true) {
        produce_item();
        wait(empty);
        wait(mutex);
        add_to_buffer();
        signal(mutex);
        signal(full);
    }
}
void consumer() {
    while (true) {
        wait(full);
        wait(mutex);
        remove_from_buffer();
        consume_item();
    }
}
```

CSA-0460

ASSIGNMENT-1

(11)

V. Jayakrishna
192821060

The classical synchronization problems in operating systems refers to well-known problems used to demonstrate and solve issues related to process synchronization, mutual exclusion, deadlock and resource sharing.

producer - consumer problem (Bounded Buffer problem)

A producer thread generates data and puts in a buffer, while a consumer thread takes data from the buffer. The buffer has a limited capacity, so synchronization is required to avoid overfilling or underflowing.

10/10

20m

Pseudo-code:

```
semaphore mutex = 0;
semaphore empty = n;
semaphore full = 0;
void producer() {
    while (true) {
        produce_item();
        wait(empty);
        wait(mutex);
        add_to_buffer();
        signal(mutex);
        signal(full);
    }
}
void consumer() {
    while (true) {
        wait(full);
        wait(mutex);
        remove_item();
        consume_item();
    }
}
```

Example:
- Buffering a printing system where documents are added to and printer processes them

2. Dining philosophers problem

Five philosophers sit at a circular table. Each philosopher alternates between thinking and eating. They need two forks to eat, and no two philosophers can use the same fork simultaneously this requires synchronization to prevent deadlock and starvation.

Pseudo-code:

```
semaphore fork[5] = {1, 1, 1, 1, 1};
void philosopher(int i){
    while(true){
        think();
        wait(fork[i]);
        wait(fork[(i+1)%5]);
        eat();
        signal(fork[i]);
        signal(fork[(i+1)%5]);
    }
}
```

example: Resource allocation in distributed systems where multiple processes shared limited resources.

- database transactions requiring multiple locks

Reader-writer problem:

Multiple processes accesses a shared resource. Readers can acesses the resource simultaneously but writers requires exclusive access.

Pseudo-code:

```
semaphore mutex = 1;
semaphore rw_mutex = 1;
int read_count = 0;

void reader() {
    while (true) {
        wait(mutex);
        read_count++;
        if (read_count == 1) wait(rw_mutex);
        signal(mutex);

        read_data();

        wait(mutex);
        read_count--;
        if (read_count == 0) signal(rw_mutex);
        signal(mutex);
    }
}

void writer() {
    while (true) {
        wait(rw_mutex);
        write_data();
        signal(rw_mutex);
    }
}
```

ex: concurrent access to shared database where multiple users can read, but only one can write at a time.

Barbershop problem:

A barber provides services to customers in a barbershop with a waiting area. if no customers are present, the barber sleeps. if customer arrives they wait if barber is busy or wake the barber if idle.

Pseudo - code:

```
semaphore customer = 0;
semaphore barber = 0;
semaphore mutex = 1;
int waiting = 0;

void customer() {
    wait(mutex);
    if (waiting < n) {
        waiting ++;
        signal(customers);
        signal(mutex);
        wait(barber);
        get_index();
    } else {
        signal(mutex);
    }
}

void barber() {
    while (true) {
        wait(customers);
        wait(mutex);
        waiting --;
        signal(barber);
        cut_hair();
    }
}
```

ex:- thread pools where workers serve incoming tasks
- customers service systems with limited staff