National Textile University

Department of Computer Science

Subject

Operating system

Submitted to:

Sir Nasir Mehmood

Submitted by:

Abdul Rehman

Registration Number

23-NTU-CS-1122

Assignment No.

02

Semester

5^{th}

Part 1: Semaphore theory:

Question:

1. A counting semaphore is initialized to 7. If 10 wait() and 4 signal() operations are performed, find the final value of the semaphore.

Answer:

We start with 7 keys.

Each wait() takes 1 key. So 10 wait() calls take 10 keys.

Each signal() gives back 1 key. So 4 signal() calls return 4 keys.

Final keys = 7 − 10 + 4 = 1

The final semaphore value is 1.

Question:

2. A semaphore starts with value 3. If 5 wait() and 6 signal() operations occur, calculate the resulting semaphore value.

Answer:

Take the semaphore as keys:

Start with 3 keys.

5 wait() calls take 5 keys so 3 − 5 = −2.

6 signal() calls return 6 keys so −2 + 6 = 4.

Final semaphore value = 4.

Question:

3. A semaphore is initialized to 0. If 8 signal() followed by 3 wait() operations are executed, find the final value.

Answer:

Take the semaphore as keys:
Start with 0 keys.
8 signal() calls add 8 keys, so 0 plus 8 equals 8.
3 wait() calls take 3 keys, so 8 minus 3 equals 5.

Final semaphore value is 5.

Question:

4. A semaphore is initialized to 2. If 5 wait() operations are executed:
a) How many processes enter the critical section?
b) How many processes are blocked?

Answer:

Think of the semaphore as keys:
Start with 2 keys. The first 2 wait() calls take the keys and enter the critical section.
The next 3 wait() calls find no keys and have to wait.

a) Processes in the critical section: 2
b) Processes blocked: 3

Question:

5. A semaphore starts at 1. If 3 wait() and 1 signal() operations are performed:
a) How many processes remain blocked?
b) What is the final semaphore value?

Answer:

The semaphore starts with 1 key, so 1 process can enter.

- First wait() takes the key, Process 1 enters.
- Next 2 wait() calls find no keys, so Processes 2 and 3 are blocked.
- One signal() returns a key, Process 2 enters.
- Process 3 remains blocked.

a) Processes blocked: 1 (Process 3)
b) Final semaphore value: -1

Question:

6.

```
semaphore S = 3;
wait(S);
wait(S);
signal(S);
wait(S);
wait(S);
```

a) How many processes enter the critical section?
b) What is the final value of S?

Answer:

Semaphore starts with 3 keys, so 3 processes can enter.

1. First wait(): Process 1 takes 1 key, S = 2, enters.
2. Second wait(): Process 2 takes 1 key, S = 1, enters.
3. signal(): 1 key returned, S = 2.
4. Third wait(): Process 3 takes 1 key, S = 1, enters.
5. Fourth wait(): Process 4 takes 1 key, S = 0, enters.

a) Processes in critical section: 4
b) Final semaphore value: 0

Question:

7.
```
semaphore S = 1;
wait(S);
wait(S);
signal(S);
signal(S);
```

a) How many processes are blocked?
b) What is the final value of S?

Answer:

Semaphore starts with 1 key, so 1 process can enter.

1. First wait(): Process 1 takes 1 key, S = 0, enters.
2. Second wait(): Process 2 finds no key, is blocked, S = -1.
3. First signal(): Key is returned, Process 2 enters, S = 0.
4. Second signal(): Key is returned, S = 1.

a) Processes blocked: 1 (Process 2)
b) Final semaphore value: 1

Question:

8. A binary semaphore is initialized to 1. Five wait() operations are executed without any signal(). How many processes enter the critical section and how many are blocked?

Answer:

Binary semaphore has 1 key, so only 1 process can enter at a time.

1. First wait(): Process 1 takes the key, enters, semaphore = 0.
2. Next 4 wait(): No key available, Processes 2, 3, 4, and 5 are blocked.

Question:

9. A counting semaphore is initialized to 4. If 6 processes execute wait() simultaneously, how many proceed and how many are blocked?

Answer:

Semaphore has 4 keys, so 4 processes can enter at once.

1. First 4 processes take the keys and enter, semaphore = 0.
2. Next 2 processes find no keys and are blocked.

Processes that proceed: 4
Processes blocked: 2

Question:

10. A semaphore S is initialized to 2.

```
wait(S);
wait(S);
wait(S);
signal(S);
signal(S);
wait(S);
```

a) Track the semaphore value after each operation.
b) How many processes were blocked at any time?

Answer:

Semaphore starts with 2 keys.

1. Process 1 wait(): takes 1 key, semaphore = 1, enters.
2. Process 2 wait(): takes 1 key, semaphore = 0, enters.
3. Process 3 wait(): no keys, blocked, semaphore = -1.
4. signal(): key returned, Process 3 wakes up, semaphore = 0.
5. signal(): no one waiting, semaphore = 1.
6. Process 4 wait(): takes 1 key, semaphore = 0, enters.

a) Semaphore values after each step: 2 → 1 → 0 → -1 → 0 → 1 → 0
b) Processes blocked: 1 (Process 3)

Question:

11. A semaphore is initialized to 0. Three processes execute wait() before any signal(). Later, 5 signal() operations are executed.
a) How many processes wake up?
b) What is the final semaphore value?

Answer:

Semaphore starts with 0 keys, so no process can enter.

- 3 wait() calls:
  - Process 1 blocked, S = -1
  - Process 2 blocked, S = -2
  - Process 3 blocked, S = -3
- 5 signal() calls:

1. Signal 1 wakes Process 1, S = -2
2. Signal 2 wakes Process 2, S = -1
3. Signal 3 wakes Process 3, S = 0
4. Signal 4, S = 1
5. Signal 5, S = 2

a) Processes that wake up: 3 (all initially blocked)
b) Final semaphore value: 2 (2 keys available)

Part 2: Semaphore Coding

Consider the Producer–Consumer problem using semaphores as implemented in Lab-10 (Lab-plan attached). Rewrite the program in your own coding style, compile and execute it successfully, and explain the working of the code in your own words.

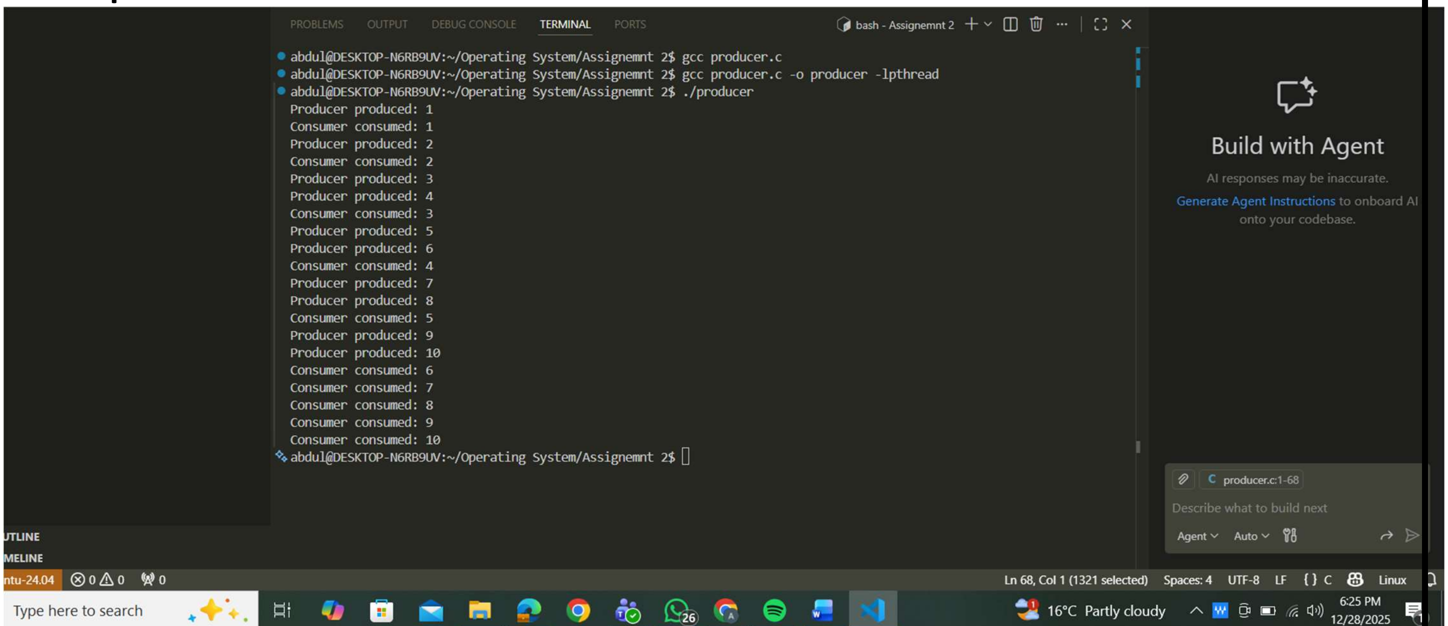
Submission Requirements:

- Your rewritten source code
- A brief description of how the code works
- Screenshots of the program output showing successful execution

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty, full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    for (int i = 1; i <= 10; i++) {
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        buffer[in] = i;
        printf("Producer produced: %d\n", i);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);

        sleep(1);
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 1; i <= 10; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);

        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Output:



Description:

1. A buffer of size 5 is created to store items.
2. Two indexes are used:

   in → position where producer puts an item

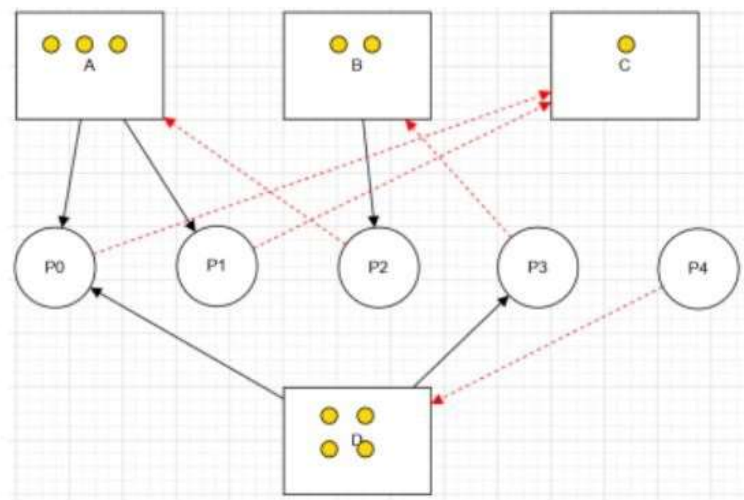
   out → position where consumer takes an item
3. Two semaphores are used:

   empty : keeps track of empty spaces in the buffer

   full : keeps track of filled spaces in the buffer
4. A mutex is used to ensure that only one thread accesses the buffer at a time.
5. Two producer threads are created.
6. Two consumer threads are created.
7. Each producer produces 3 items and puts them into the buffer.
8. Each consumer consumes 3 items from the buffer.
9. Producers wait if the buffer is full.
10. Consumers wait if the buffer is empty.
11. The buffer works like a circular queue using modulo operation.
12. Producers are faster (sleep(1)), and consumers are slower (sleep(2)).

Part 3: RAG (Recourse Allocation Graph)

- Convert the following graph into matrix table ,



Answer:

A. Allocation Matrix (Who HAS what)

Count the solid black arrows coming *out* of a resource box *into* a process circle.

Process	A	B	C	D
P0	1	0	0	1
P1	1	0	0	0
P2	0	1	0	0
P3	0	0	0	1
P4	0	0	0	0

Process	A	B	C	D
Total Allocated	2	1	0	2

B. Request Matrix (Who WANTS what)

Count the red arrows coming *out* of a process circle *into* a resource box.

Process	A	B	C	D
P0	0	0	1	0
P1	0	1	1	0
P2	1	0	1	0
P3	0	1	0	0
P4	0	0	0	1

C. Available Resources Vector:

A: 3 instances total - 2 allocated = 1 available

B: 2 instances total - 1 allocated = 1 available

C: 1 instance total - 0 allocated = 1 available

D: 4 instances total - 2 allocated = 2 available

Resource	A	B	C	D
Available	1	1	1	2

Part 4: Banker's Algorithm:

System Description:

- The system comprises five processes (P0–P3) and four resources (A,B,C,D).
- Total Existing Resources:

Total			
A	B	C	D
6	4	4	2

- Snapshot at the initial time stage:

	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	1	1	3	2	1	1				
P1	1	1	0	0	1	2	0	2				
P2	1	0	1	0	3	2	1	0				
P3	0	1	0	1	2	1	0	1				

-

Questions:

1. **Compute the Available Vector**:
   - Calculate the available resources for each type of resource.

Answer:

Total resources:

A=0, B=4 ,C=4, D=2

Available matrix Total allocated:

A: 2 + 1 + 1 + 0 = 4

B: 0 + 1 + 0 + 1 = 2

C: 1 + 0 + 1 + 0 = 2

D: 1 + 0 + 0 + 1 = 2

Total - total allocated

A: 6 - 4 = 2

B: 4 - 2 = 2

C: 4 - 2 = 2

D: 2 - 2 = 0

Available Vector: [2,2,2,0]

Question:

2. **Compute the Need Matrix**:
    - Determine the need matrix by subtracting the allocation matrix from the maximum matrix.

Answer:

The **Need Matrix** is calculated using the formula: Need = Max - Allocation.

Process	Max (A B C D)	Allocation (A B C D)	Need (A B C D)
P0	3 2 1 1	2 0 1 1	**1 2 0 0**
P1	1 2 0 2	1 1 0 0	**0 1 0 2**

P2	3 2 1 0	1 0 1 0	**2 2 0 0**
P3	2 1 0 1	0 1 0 1	**2 0 0 0**

Question:

3. **Safety Check**:
   - Determine if the current allocation state is safe. If so, provide a safe sequence of the processes.
   - Show how the Available (working array) changes as each process terminates.

Answer:

To determine if the state is safe, we find a sequence where each process's need available.
Once a process finishes, it releases its Allocation back to the Available pool.

Step	Process	Need	Available (Work)	Can it run	New Available (Work + Allocation)
1	**P0**	[1, 2, 0, 0]	[2, 2, 2, 0]	**Yes**	[2,2,2,0] + [2,0,1,1]={[4, 2, 3, 1]}
2	**P2**	[2, 2, 0, 0]	[4, 2, 3, 1]	**Yes**	[4,2,3,1] + [1,0,1,0]={[5, 2, 4, 1]}
3	**P3**	[2, 0, 0, 0]	[5, 2, 4, 1]	**Yes**	[5,2,4,1] + [0,1,0,1]={[5, 3, 4, 2]}
4	**P1**	[0, 1, 0, 2]	[5, 3, 4, 2]	**Yes**	[5,3,4,2] + [1,1,0,0] ={[6, 4, 4, 2]}

Yes. Safe Sequence: {P0, P2, P3, P1}