Submitted By: Shanza Batool (23-NTU-CS-1209)

Submitted To: Sir Nasir Mehmood

Operating Systems – COC 3071

SE 5th A – Fall 2025

After-mid Homework -1

Part 1: Semaphore theory

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

   - **Initial value**:
     S = 7
   - **Each wait() decreases by 1:**
     10 wait → 7 - 10 = **-3**
   - **Each signal() increases by 1:**
     4 signal → -3 + 4 = **1**
   - **Final value:**
     1

2. **A semaphore starts with value 3. If 5 wait() and 6 signal() operations occur, calculate the resulting semaphore value.**
   - **Initial:**
     S = 3
   - **5 wait →**
     3 - 5 = -2
   - **6 signal → -**
     2 + 6 = 4
   - **Final value:**
     4

3. **A semaphore is initialized to 0. If 8 signal() followed by 3 wait() operations are executed, find the final value.**
   - **Initial:**
     S = 0
   - **8 signal →**
     0 + 8 = 8
   - **3 wait →**
     8 - 3 = 5
   - **Final value:**
     5

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?**

- **Initial:** S = 2
- Max 2 processes can enter the critical section (value > 0)
- Remaining 3 wait operations are blocked

a) Processes entered: 2

b) Processes blocked: 3

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?**

- **Initial:** S = 1
- **3 wait →** 1 - 3 = -2 (so 2 processes blocked)
- **1 signal →** -2 + 1 = -1

a) Processes blocked: 1 (still waiting)

b) Final value: -1

"Negative semaphore value = number of blocked processes"

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?**

Operation	S value	Entry in Critical Section
Initial (S)	3	
Wait (S)	2	Enters Critical Section
Wait (S)	1	Enters Critical Section
Signal (S)	2	Releases Critical Section
Wait (S)	1	Enters Critical Section
Wait (S)	0	Enters Critical Section

a) Processes entered: 4

b) Final value: 0

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?**

Operation	S value	Entry in Critical Section

Initial (S)	1	
Wait (S)	0	Enters Critical Section
Wait (S)	-1	Blocked one process
Signal (S)	0	Unblock one process
Signal (S)	1	Releases Critical Section

a) Processes blocked: 0 (all processes are unblocked after signal)

b) Final value: 1

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?**

- Initial: S = 1
- First wait → 0 → enters CS
- Next 4 wait → -1, -2, -3, -4 → blocked

Processes entered CS: 1

Processes blocked: 4

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

- Initial: S = 4
- First 4 processes → S = 0 → enter CS
- Last 2 processes → blocked

Processes proceed: 4

Processes blocked: 2

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?**

Operation	S value	Critical Section
Initial	2	
Wait(S)	1	Enters Critical Section
Wait(S)	0	Enters Critical Section
Wait(S)	-1	Block one process
Signal(S)	0	Unblock one process
Signal(S)	1	Release Critical Section
Wait(S)	0	Enters Critical Section

a) Semaphore after each operation: 2 → 1 → 0 → -1 → 0 → 1 → 0

b) Processes blocked at any time: 1

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?

Step 1: Execute 3 wait() operations

Initial value = 0

Step 2: Execute 5 signal() operations

Each signal() first wakes a blocked process if any exist.

a) Processes wake up: 3
b) Final semaphore value: 2

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

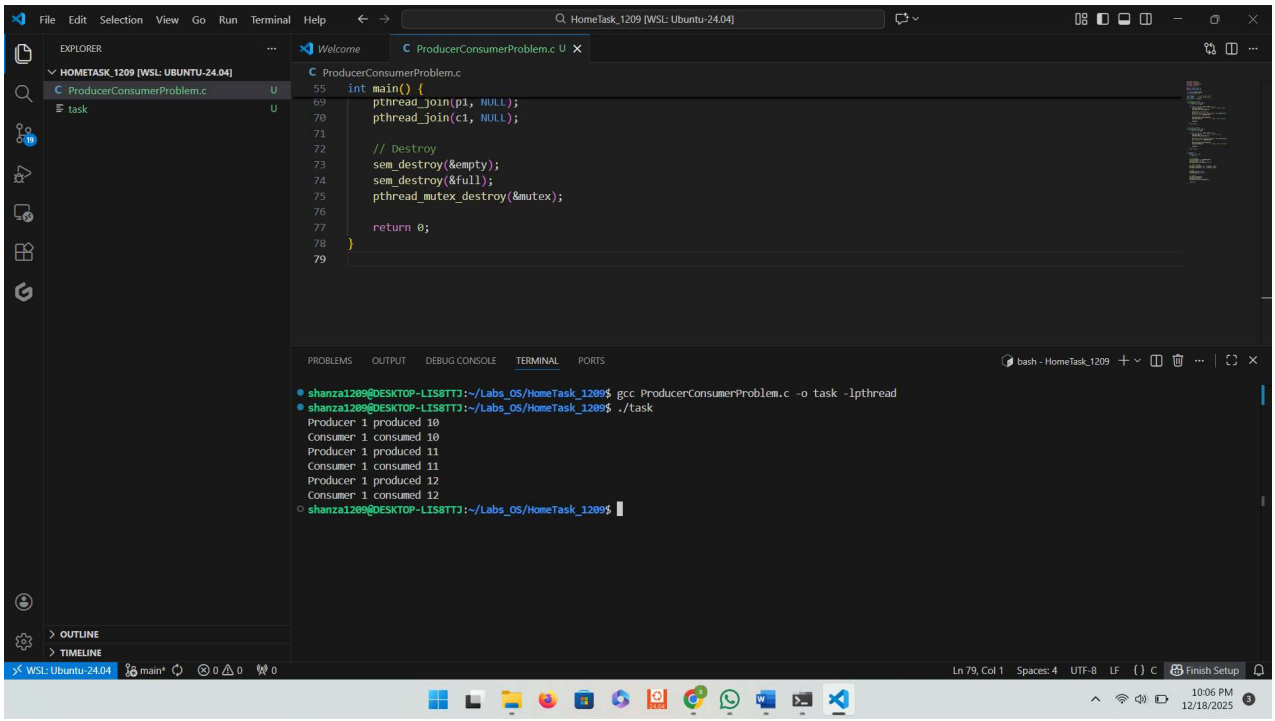
Code:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_CAPACITY 5
#define ITEMS_TO_PRODUCE 3

int sharedBuffer[BUFFER_CAPACITY];
int writeIndex = 0;
int readIndex = 0;

sem_t spacesAvailable;
sem_t itemsAvailable;
pthread_mutex_t bufferMutex;

// Producer function
```

```
void* produceItems(void* arg) {
    int producerId = (int)arg;

    for (int i = 0; i < ITEMS_TO_PRODUCE; i++) {
        sem_wait(&spacesAvailable);      // Wait for empty slot
        pthread_mutex_lock(&bufferMutex); // Lock buffer

        sharedBuffer[writeIndex] = producerId * 10 + i;
        printf("Producer-%d produced %d\n", producerId, sharedBuffer[writeIndex]);
        writeIndex = (writeIndex + 1) % BUFFER_CAPACITY;

        pthread_mutex_unlock(&bufferMutex); // Unlock buffer
        sem_post(&itemsAvailable);          // Signal item is available

        sleep(1); // Simulate production
    }

    return NULL;
}
// Consumer function
void* consumeItems(void* arg) {
    int consumerId = (int)arg;
    for (int i = 0; i < ITEMS_TO_PRODUCE; i++) {
        sem_wait(&itemsAvailable);       // Wait for item
        pthread_mutex_lock(&bufferMutex); // Lock buffer
        printf("Consumer-%d consumed %d\n", consumerId, sharedBuffer[readIndex]);
        readIndex = (readIndex + 1) % BUFFER_CAPACITY;

        pthread_mutex_unlock(&bufferMutex); // Unlock buffer
        sem_post(&spacesAvailable);         // Signal space is available

        sleep(2); // Simulate consumption
```

```
    }
    return NULL;
}
int main() {
    pthread_t producers[2], consumers[2];
    int ids[2] = {101, 102};
    // Initialize semaphores and mutex
    sem_init(&spacesAvailable, 0, BUFFER_CAPACITY);
    sem_init(&itemsAvailable, 0, 0);
    pthread_mutex_init(&bufferMutex, NULL);
    // Start producer and consumer threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&producers[i], NULL, produceItems, &ids[i]);
        pthread_create(&consumers[i], NULL, consumeItems, &ids[i]);
    }
    // Wait for all threads to finish
    for (int i = 0; i < 2; i++) {
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }
    // Clean up
    sem_destroy(&spacesAvailable);
    sem_destroy(&itemsAvailable);
    pthread_mutex_destroy(&bufferMutex);
    return 0;
}
```

Output:



Description Summary:

- Uses a shared buffer of fixed size for storing items.
- Producer thread: produces items and adds them to the buffer.
- Consumer thread: consumes items from the buffer.
- Semaphores:
- emptySlots → counts empty spaces; producer waits if full.
- filledSlots → counts filled slots; consumer waits if empty.
- Mutex ensures mutual exclusion when accessing the buffer.
- **Producer workflow**:

1. Waits for empty slot (sem_wait(emptySlots))
2. Locks buffer (pthread_mutex_lock)
3. Adds item
4. Unlocks buffer (pthread_mutex_unlock)
5. Signal item available (sem_post(filledSlots))

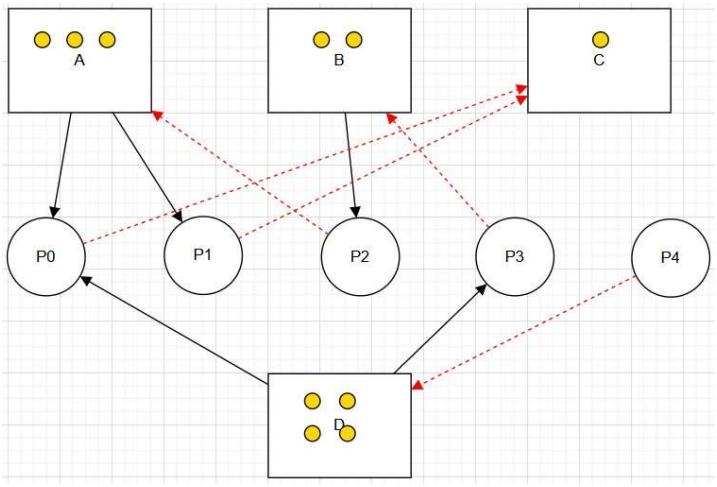
- **Consumer workflow**:

1. Waits for filled slot (sem_wait(filledSlots))
2. Locks buffer

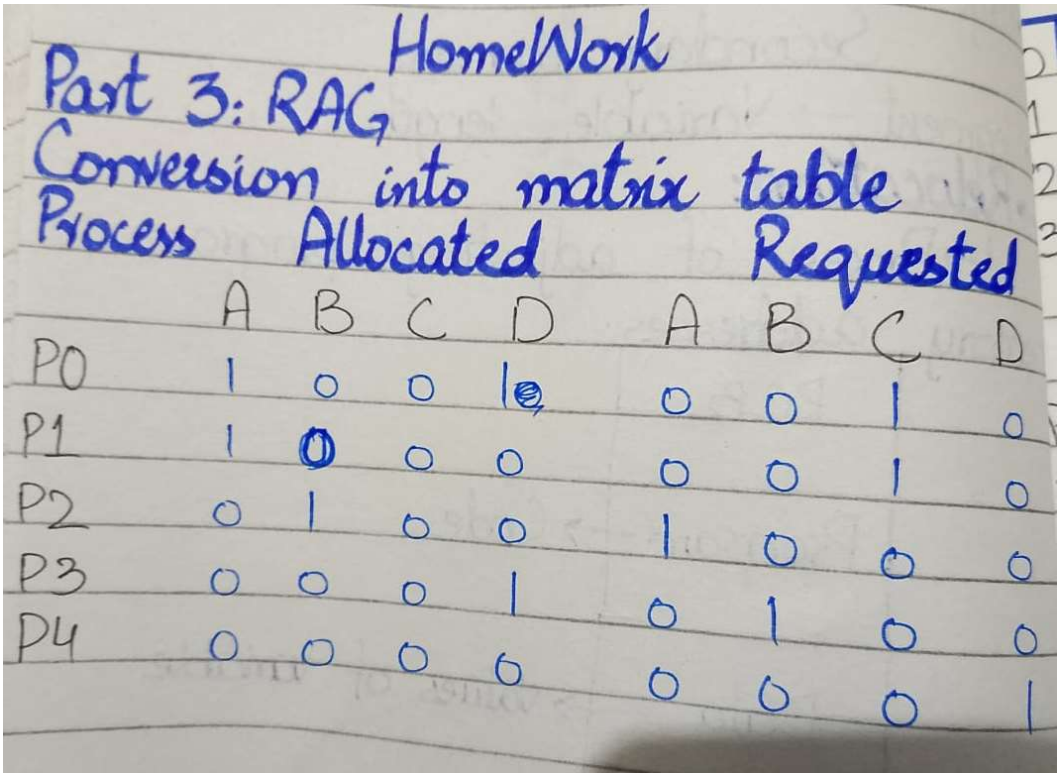
3. Removes item
4. Unlocks buffer
5. Signal empty slot (sem_post(emptySlots))

Part 3: RAG (Recourse Allocation Graph)

- Convert the following graph into matrix table ,



Solution:



Part 4: Banker's Algorithm

System Description:

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

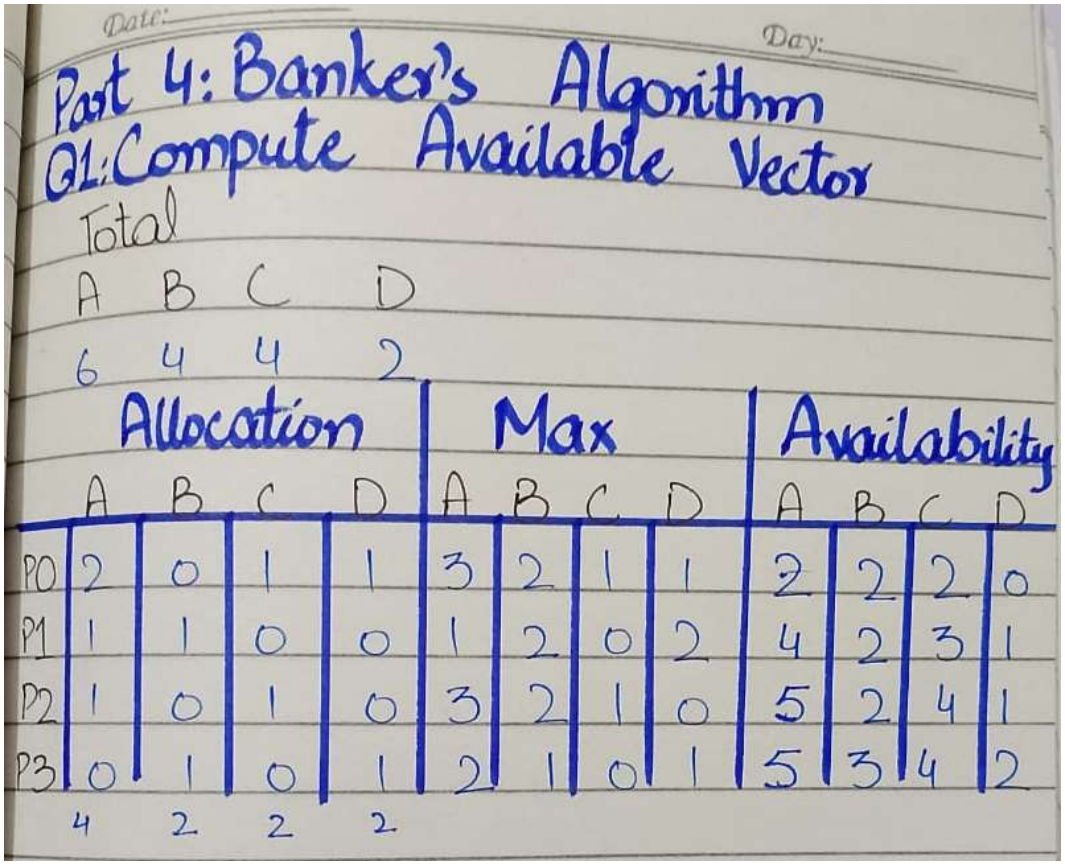
Total			
A	B	C	D
6	4	4	2

- 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				

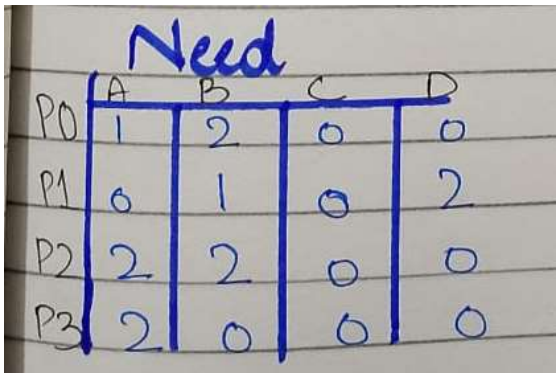
Questions:

1. **Compute the Available Vector**:



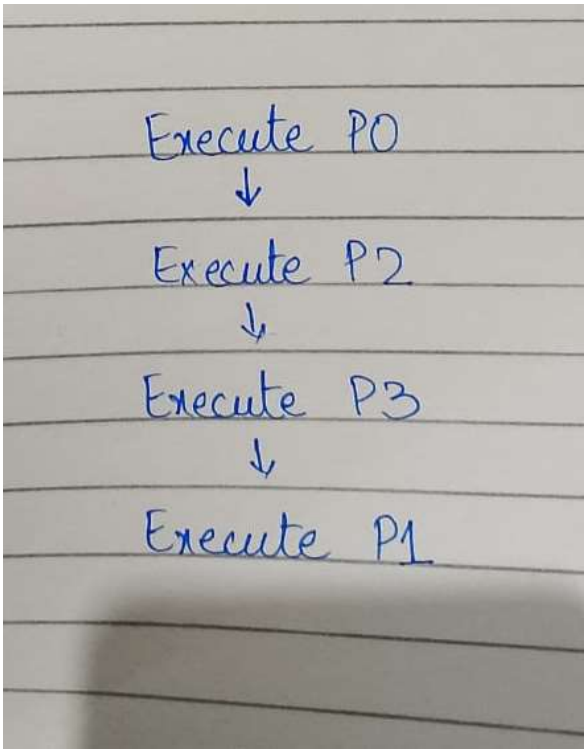
2. **Compute the Need Matrix**:

   • Determine the need matrix by subtracting the allocation matrix from the maximum matrix.



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.



Submission Guidelines:

- Ensure all answers are well-explained and calculations are shown step-by-step.
- Submit your assignment on MS Team and GitHub in a PDF format.
- VIVA based Evaluation so Develop your own solution after getting help.