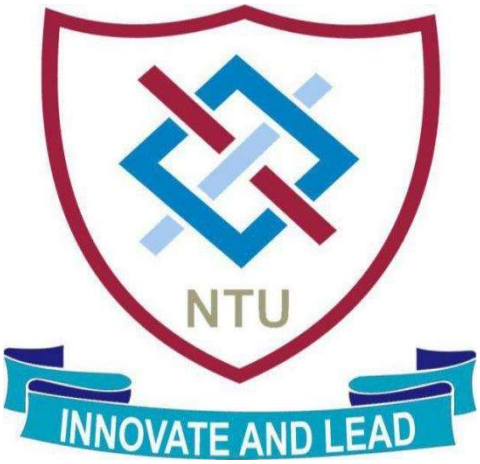



# NATIONAL TEXTILE

# UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE


## SUBMITTED BY:

Farah Naz 23-NTU-CS-1152

**SECTION SE: 5th(A)**


**Lab - 10**

## SUBMITTED TO:

Sir Nasir Mehmood

## SUBMISSION DATE:

12-18-2025

# 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.**
   **Ans:**
   Initial value of semaphore: 7
   After 10 wait() operations: 7-10= -3
   Now four signal() operations are performed: -3+4 = 1

   Final value of semaphore: 1

2. **A semaphore starts with value 3. If 5 wait() and 6 signal() operations occur, calculate the resulting semaphore value.**
   **Ans:**
   Initial value of semaphore: 3
   After 5 wait() operations: 3-5= -2
   Now 6 signal() operations are performed: -2+6 = 4

   Final value of semaphore: 1

3. **A semaphore is initialized to 0. If 8 signal() followed by 3 wait() operations are executed, find the final value.**
   **Ans:**
   Initial value of semaphore: 0
   Now 8 signal() operations are performed: 0+8=8
   After 3 wait() operations: 8-3= 5

   Final value of semaphore: 5

4. **A semaphore is initialized to 2. If 5 wait() operations are executed:**
   **Ans:**

a) How many processes enter the critical section = 2
b) How many processes are blocked =3

5. **A semaphore starts at 1. If 3 wait() and 1 signal() operations are performed:**

Ans:

a) **How many processes remain blocked** =
Wait 1 = Semaphore=0 => 1 process enters
Wait 2 = Semaphore=-1 =>1 process blocked
Wait 3 = Semaphore=-2 =>2 process blocked

b) **What is the final semaphore value=**
1signal() = -2+1 = -1

**6. semaphore S = 3;**
**wait(S); wait(S);**
**signal(S);**
**wait(S); wait(S);**

a) **How many processes enter the critical section?**
1. Wait 1 = Semaphore=3-1=2 => 1st process enters
2. Wait 2 = Semaphore=2-1=1 =>2nd process enters
3. Signal 1 = Semaphore=1+1=2 => 1st process leaves
4. Wait 3 = Semaphore=2-1=1 =>3rd process enters
5. Wait 4 = Semaphore=1-1=0 => 4th process enters

b) **What is the final value of S**? Final value=0

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

a) **How many processes are blocked:**
2 wait operations are performed:
1st wait() = 1-1 = 0 => 1 process enters.
2nd wait() = 0-1 = -1 => **1 process is blocked**
b) What is the final value of S?
**Final semaphore: -1+2 = 1**

8. **A binary semaphore is initialized to 1. Five wait() operations are executed without any signal().**
**How many processes enter the critical section:** 1 process enters the critical section
**how many are blocked:** 4 processes are blocked

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

**Ans:0**

Processes proceeding= 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.**

1. Wait 1 = Semaphore=2-1=1 => 1st process enters
2. Wait 2 = Semaphore=1-1=0 =>2nd process enters
3. Wait 3 = Semaphore=0-1=-1 =>3rd process BLOCKED
4. Signal 1 = Semaphore=-1+1=0 => 1st process leaves
5. Signal 2 = Semaphore=0+1=1 => 2nd process leaves
6. Wait 4 = Semaphore=1-1=0 => 3rd process enters

c) **How many processes were blocked at any time?**

1 process was blocked during the process.

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?**

3 wait() = 0-3= -3 (3 processes blocked)

5 signal() = -3+5= 2 (**3 processes wakes**)

b) **What is the final semaphore value?**

Final 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>
```

```c
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0; // Producer index
int out = 0; // Consumer index
sem_t empty; // Counts empty slots
sem_t full; // Counts full slots
pthread_mutex_t mutex;
    void* producer(void* arg) {
        int id = *(int*)arg;
        for(int i = 0; i < 3; i++) { // Each producer makes 3 items
        int item = id * 100 + i;
        // This decrements the empty slot as a slot is taken
        sem_wait(&empty);
        // TODO: Lock the buffer
        pthread_mutex_lock(&mutex);
        // Add item to buffer
        buffer[in] = item;
        printf("Producer %d produced item %d at position %d\n",
        id, item, in);
        in = (in + 1) % BUFFER_SIZE;
        // TODO: Unlock the buffer
        pthread_mutex_unlock(&mutex);
        // This increments the buffer so that consumer can consume it
        sem_post(&full);
        sleep(1);
        }
        return NULL;
    }
    void* consumer(void* arg) {
    int id = *(int*)arg;
    for(int i = 0; i < 3; i++) {
        //This decrements the full
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %d consumed item %d from position %d\n",
        id, item, out);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        //This increases the empty slot and signals that the consumer has
successfully consumed.
        sem_post(&empty);
        sleep(2); // Consumers are slower
    }
    return NULL;
    }
    int main() {
```

```
    pthread_t prod[2], cons[2];
    int ids[2] = {1, 2};
    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE); // All slots empty initially
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    // No slots full initially
    // Create producers and consumers
    for(int i = 0; i < 2; i++) {
        pthread_create(&prod[i], NULL, producer, &ids[i]);
        pthread_create(&cons[i], NULL, consumer, &ids[i]);
    }
    // Wait for completion
    for(int i = 0; i < 2; i++) {
        pthread_join(prod[i], NULL);
        pthread_join(cons[i], NULL);
    }
    // Cleanup
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    return 0;

    }
```

**Description of how the code works:**

This program solves the **Producer–Consumer problem** using **semaphores and mutex**.

A buffer of size **5** is shared between producers and consumers.

- **empty semaphore** keeps count of empty slots in the buffer.
- **full semaphore** keeps count of filled slots.
- **Mutex** is used to avoid race condition while accessing the buffer.

**The producer:**

- Waits if the buffer is full (sem_wait(empty)).
- Locks the mutex and puts item in the buffer.
- Unlocks mutex and signals the consumer (sem_post(full)).

**The consumer**:

- Waits if the buffer is empty (sem_wait(full)).
- Locks the mutex and removes item from buffer.
- Unlocks mutex and signals the producer (sem_post(empty)).’

**Speed of consumer and Producer:**

In this code the **consumer** has a sleep time of 2 seconds making it **slower** than the producer which has a sleep time of 1 second.

**Impact:**

- The producer is getting more time to fill up the buffer.
- The producer is waiting longer for the empty slot.

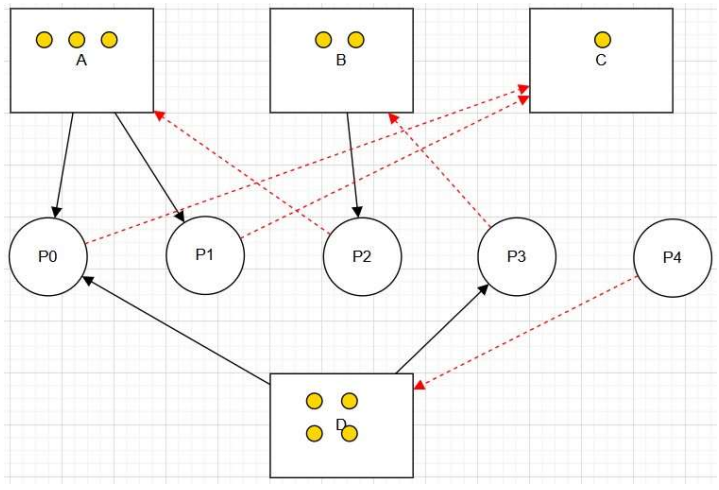
Two producers and two consumers are created.
This ensures **proper synchronization** and **safe access** to the shared buffer.

**Output**:

```
farah@DESKTOP-QLMRLIR:~/After_Mid_1152$ gcc run test.c -o producer
/usr/bin/ld: cannot find run: No such file or directory
collect2: error: ld returned 1 exit status
farah@DESKTOP-QLMRLIR:~/After_Mid_1152$ gcc test.c -o producer
farah@DESKTOP-QLMRLIR:~/After_Mid_1152$ ./p
bash: ./p: No such file or directory
farah@DESKTOP-QLMRLIR:~/After_Mid_1152$ ./producer
Producer 1 produced item 100 at position 0
Consumer 1 consumed item 100 from position 0
Producer 2 produced item 200 at position 1
Consumer 2 consumed item 200 from position 1
Producer 1 produced item 101 at position 2
Producer 2 produced item 201 at position 3
Consumer 2 consumed item 101 from position 2
Consumer 1 consumed item 201 from position 3
Producer 2 produced item 202 at position 4
Producer 1 produced item 102 at position 0
Consumer 1 consumed item 202 from position 4
Consumer 2 consumed item 102 from position 0
farah@DESKTOP-QLMRLIR:~/After_Mid_1152$
```

**Part 3: RAG (Recourse Allocation Graph)**

- Convert the following graph into matrix table ,



**Ans:**

**RAG:**

| 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 |

**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 | | | | |

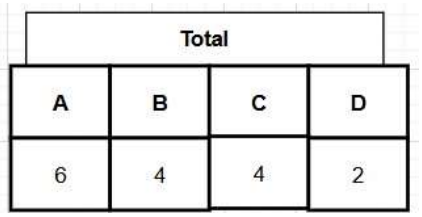
- 

**Questions:**

1. **Compute the Available Vector**:

- Calculate the available resources for each type of resource.

Available:



| Total | | | |
|---|---|---|---|
| A | B | C | D |
| 6 | 4 | 4 | 2 |

## 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

## Available resource= 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]

2. **Compute the Need Matrix**:

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

| — | Allocation: A | Allocation: B | Allocation: C | Allocation: D | Max: A | Max: B | Max: C | Max: 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 |

| Process | NEED: A | NEED : B | NEED : C | NEED : D |
|---|---|---|---|---|
| **P0** | 1 | 2 | 0 | 1 |
| **P1** | 0 | 1 | 0 | 2 |
| **P2** | 2 | 2 | 0 | 0 |
| **P3** | 2 | 2 | 0 | 0 |

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.

- **Step 1: P0**
  - Can P0 be satisfied? Need (1, 2, 0, 0) < Available (2, 2, 2, 0). **Yes.**
  - P0 terminates and releases its allocation.
  - **New Available** = (2, 2, 2, 0) + (2, 0, 1, 1) = (4, 2, 3, 1).
- **Step 2: P2**
  - Can P2 be satisfied? Need (2, 2, 0, 0) < Available (4, 2, 3, 1). **Yes.**
  - P2 terminates and releases its allocation.
  - **New Available** = (4, 2, 3, 1) + (1, 0, 1, 0) (5, 2, 4, 1)
- **Step 3: P3**
  - Can P3 be satisfied? Need (2, 0, 0, 0) < Available (5, 2, 4, 1). **Yes.**
  - P3 terminates and releases its allocation.

- **New Available** = (5, 2, 4, 1) + (0, 1, 0, 1) = (5, 3, 4, 2).

- **Step 4: P1**
  - Can P1 be satisfied? Need (0, 1, 0, 2) < Available (5, 3, 4, 2). **Yes.**
  - P1 terminates and releases its allocation.
  - **New Available** = (5, 3, 4, 2) + (1, 1, 0, 0) = (6, 4, 4, 2).

**Conclusion: The system is in a Safe State**.
**Safe Sequence**: P0 -> P2 -> P3 -> P1