



National Textile University

**Department of Computer Science**

Subject: Operating System

---

Submitted to: Sir Nasir

---

Submitted by: Maha

---

Reg. number: 23-NTU-CS-1170

---

After Mid's first Homework

---

Semester: 5<sup>th</sup>

# 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 = 7**

- 10 wait()  $\rightarrow 7 - 10 = -3$
- 4 signal()  $\rightarrow -3 + 4 = 1$
- Semaphore value is 1

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

**Initial value = 3**

- 5 wait()  $\rightarrow 3 - 5 = -2$
- 6 signal()  $\rightarrow -2 + 6 = 4$
- Final semaphore value = 4

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

**Initial value = 0**

- 8 signal()  $\rightarrow 0 + 8 = 8$
- 3 wait()  $\rightarrow 8 - 3 = 5$
- Final semaphore value = 5

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

**Initial value = 2**

- 5 wait() operations

a) How many processes enter the critical section?

- Only **2** processes can enter (value becomes 0)

b) How many processes are blocked?

Remaining waits =  $5 - 2 = \mathbf{3}$  **blocked**

**Answer:**

- Enter CS = **2**
- Blocked = **3**

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

**Initial value = 1**

- 3 wait()  $\rightarrow 1 - 3 = -2$
- 1 signal()  $\rightarrow -2 + 1 = -1$

**a) Processes blocked**

- Semaphore  $-1 \rightarrow \mathbf{1}$  process blocked

**b) Final value**

- **Final semaphore value = -1**

6.

**semaphore S = 3;**

wait(S)=2

wait(S)=1

signal(S)=2

wait(S)=1

wait(S)=0

a) How many processes enter the critical section?

- Semaphore never goes negative → **5 processes enter**

b) What is the final value of S?

- $S=0$

**7.**

**semaphore S = 1;**

$S = 1$

$\text{wait}(S) \rightarrow 0$

$\text{wait}(S) \rightarrow -1$  (blocked)

$\text{signal}(S) \rightarrow 0$  (one wakes)

$\text{signal}(S) \rightarrow 1$

a) How many processes are blocked?

Only one

b) What is the final value of S?

One ( $s=1$ )

**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?

**Binary semaphore = 1**

- 5 wait() without signal
- 1 process enters CS
- Remaining 4 are blocked

**Answer:**

- Enter CS = **1**
- 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 value = 4**

- 6 wait() simultaneously
- 4 proceed
- $6 - 4 = 2$  **blocked**

**Answer:**

- Proceed = **4**
- Blocked = **2**

10. A semaphore S is initialized to 2.

a) Track the semaphore value after each operation.

- $\text{wait}(S)=1$
- $\text{wait}(S)=0$
- $\text{wait}(S)=-1$  (blocked)
- $\text{signal}(S)=0$
- $\text{signal}(S)=1$
- $\text{wait}(S)=0$

b) How many processes were blocked at any time?

- One process was blocked

11. A semaphore is initialized to 0. Three processes execute wait() before any signal(). Later, 5 signal() operations are executed.

**Initial value = 0**

- 3 processes wait() → **3 blocked**
- 5 signal() operations

a) How many processes wake up?

3 processes wake up

b) What is the final semaphore value?

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 BUF_SIZE 5
#define PRODUCE_COUNT 3
#define THREADS 2

// Shared buffer and indices
int sharedBuffer[BUF_SIZE];
int writeIndex = 0;
int readIndex = 0;

// Synchronization primitives
sem_t semEmptySlots;
sem_t semFilledSlots;
pthread_mutex_t bufferLock;

// Producer thread function
void* producerTask(void* arg) {
    int producerId = *(int*)arg;

    for (int i = 0; i < PRODUCE_COUNT; i++) {
        int data = producerId * 100 + i;

        // Wait until an empty slot is available
        sem_wait(&semEmptySlots);

        // Lock buffer before accessing it
        pthread_mutex_lock(&bufferLock);
        sharedBuffer[writeIndex] = data;
        writeIndex = (writeIndex + 1) % BUF_SIZE;
        pthread_mutex_unlock(&bufferLock);

        // Signal that a slot is filled
        sem_post(&semFilledSlots);
    }
}
```

```
pthread_mutex_lock(&bufferLock);

sharedBuffer[writeIndex] = data;
printf("Producer %d produced %d at index %d\n",
      producerId, data, writeIndex);

writeIndex = (writeIndex + 1) % BUF_SIZE;

// Unlock buffer
pthread_mutex_unlock(&bufferLock);

// Signal that a new item is available
sem_post(&semFilledSlots);

sleep(1);
}

return NULL;
}

// Consumer thread function
void* consumerTask(void* arg) {
    int consumerId = *(int*)arg;

    for (int i = 0; i < PRODUCE_COUNT; i++) {

        // Wait until a filled slot is available
        sem_wait(&semFilledSlots);

        // Lock buffer before accessing it
        pthread_mutex_lock(&bufferLock);

        int data = sharedBuffer[readIndex];
        printf("Consumer %d consumed %d from index %d\n",
              consumerId, data, readIndex);

        readIndex = (readIndex + 1) % BUF_SIZE;

        // Unlock buffer
        pthread_mutex_unlock(&bufferLock);

        // Signal that a slot is now empty
        sem_post(&semEmptySlots);

        sleep(2); // Consumer is slower
    }
}
```

```
    return NULL;
}

int main() {
    pthread_t producers[THREADS], consumers[THREADS];
    int threadIds[THREADS] = {1, 2};

    // Initialize semaphores and mutex
    sem_init(&semEmptySlots, 0, BUF_SIZE);
    sem_init(&semFilledSlots, 0, 0);
    pthread_mutex_init(&bufferLock, NULL);

    // Create producer and consumer threads
    for (int i = 0; i < THREADS; i++) {
        pthread_create(&producers[i], NULL, producerTask, &threadIds[i]);
        pthread_create(&consumers[i], NULL, consumerTask, &threadIds[i]);
    }

    // Wait for threads to finish
    for (int i = 0; i < THREADS; i++) {
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }

    // Cleanup resources
    sem_destroy(&semEmptySlots);
    sem_destroy(&semFilledSlots);
    pthread_mutex_destroy(&bufferLock);

    return 0;
}
```

Terminal:

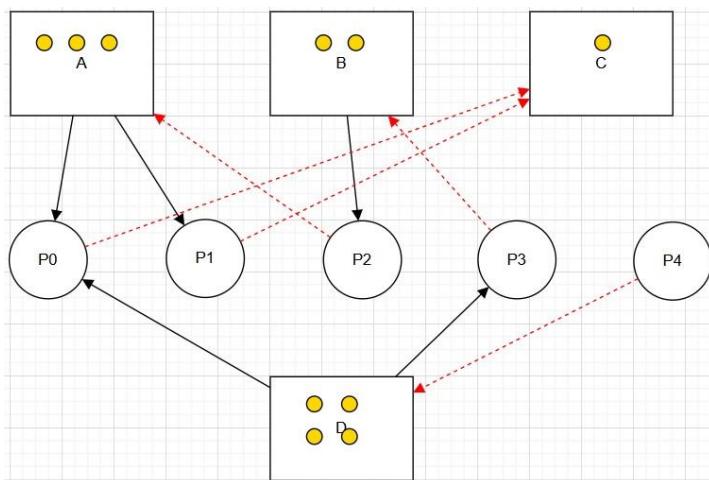
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1
bas@DESKTOP-PDJCJQV:~/1170-OS_LAB/aftermids(Homework1)$
● bas@DESKTOP-PDJCJQV:~/1170-OS_LAB/aftermids(Homework1)$ gcc part2.c -o part2 -lpthread
● bas@DESKTOP-PDJCJQV:~/1170-OS_LAB/aftermids(Homework1)$ ./part2
Producer 1 produced 100 at index 0
Consumer 1 consumed 100 from index 0
Producer 2 produced 200 at index 1
Consumer 2 consumed 200 from index 1
Producer 1 produced 101 at index 2
Producer 2 produced 201 at index 3
Consumer 1 consumed 101 from index 2
Producer 1 produced 102 at index 4
Producer 2 produced 202 at index 5
Consumer 2 consumed 201 from index 3
Consumer 1 consumed 102 from index 4
Consumer 2 consumed 202 from index 6
bas@DESKTOP-PDJCJQV:~/1170-OS_LAB/aftermids(Homework1)$

```

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

- Convert the following graph into matrix table ,



a) Allocation matrix

Process	A	B	C	D
P0	1	0	0	1

P1	1	0	0	0
P2	0	1	0	0
P3	0	0	1	1
P4	0	0	0	0

b) Request matrix

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

#### 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.

Total resources:

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

a) 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]

2. **Compute the Need Matrix:**

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

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

### 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.

To determine if the state is safe, we find a sequence where each process's **Need**  $\leq$  **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}