# National Textile University

## Department of Computer Science

Subject:

Operating System

Submitted to:

Sir Nasir Mehmood

Submitted by:

Akasha Fatima

Reg. number:

23-NTU-CS-FL-1132

Semester:

5th - A

# After_Mid_HomeWork_01

## Part 01: Semaphores 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:** Semaphore Initialized Value = 7

Wait (S – 1) = 10, Signal (S + 1) = 4

Operations = 7 – 10 = -3(Blocked)

Then, Final Value of Semaphore = -3 + 4 = 1

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

**Ans:** Semaphore Starting Value = 3

Wait (S – 1) = 5, Signal (S + 1) = 6

Operations = 3 – 5 = -2(Blocked)

Then, Final Value of Semaphore = -2 + 6 = 4

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

**Ans:** Semaphore Initialized Value = 0

Wait (S – 1) = 3, Signal (S + 1) = 8

Operations = 0 – 3 = -3(Blocked)

Then, Final Value of Semaphore = -3 + 8 = 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?**

**Ans:** Semaphore Initialized Value = 2

Wait (S – 1) = 5,

- **a.** No. of processes in the critical section = initialized value of the semaphore

  So, **2** processes will enter in the critical sections

  (Moreover, we can also that wait operations () which has S >= 0 will only enter at the critical section at a single time.)
- **b.** Operations = 2 – 5 = -3(Blocked)

### 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?

**Ans:** Semaphore Starting Value: 1

Wait (S – 1) = 3, Signal (S + 1) = 1

- **a. Blocked Processes =** 1 – 3 = -2 (blocked)
- **b. Final Semaphore Value:** -2 + 1 = -1

After **signal** operation performed only **1** process will be in the blocked state but before the Signal Operation execution, there will be **2** processes in the block state.

### 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?**

**Ans:**

a. **Table for processes entering in critical section**

| **Operations** | **S Value (Before)** | **S Value (After)** | **Entry in Critical Section** |
|---|---|---|---|
| **Wait(S - 1)** | 3 | 2 | **Yes** |
| **Wait(S - 1)** | 2 | 1 | **Yes** |
| **Signal(S + 1)** | 1 | 2 | **-** |
| **Wait(S - 1)** | 2 | 1 | **Yes** |
| **Wait(S - 1)** | 1 | 0 | **Yes** |

b. **Final Value of S** = 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?**

**Ans: Blocked Processes:** Since Semaphore value is **1** So, as per unit time 1st process will be in execution state while the 2nd process will be in blocked state. Thus, No. of blocked Processes = 1

**Final Value of S:** As **Signal** causes **Increment** in semaphore value; So, final value of S is 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?**

**Ans:** Binary Semaphore Value = 1

Wait (S – 1) = 5, Signal (S + 1) = 0

**Critical Section:** Only 1 process will be entered in the critical section then S=0

**Blocked Processes:** Remaining 4 processes will be in blocked state due to negative value of semaphore.

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

**Ans:** Counting Semaphore Value = 4

Wait (S – 1) = 6, Signal (S + 1) = 0

**Proceed Process:** 4 process will be proceed to the critical section then S=0

**Blocked Processes:** Remaining 2 processes will be in blocked state due to negative value of semaphore.

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?

**Ans:** Semaphore Value = 2

a. **Semaphore value after each operation:**

| Operations | S Value (Before) | S Value (After) | Entry in Critical Section |
|---|---|---|---|
| **Wait(S - 1)** | 2 | 1 | **Yes** |
| **Wait(S - 1)** | 1 | 0 | **Yes** |
| **Wait(S – 1)** | 0 | -1 | **No** |
| **Signal(S + 1)** | -1 | 0 | **(Unblock 1 locked process)** |
| **Signal(S + 1)** | 0 | 1 | **-** |
| **Wait(S - 1)** | 1 | 0 | **Yes** |

b. **Blocked Processes:** 1

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

## How many processes wake up?

## What is the final semaphore value?

**Ans:** Semaphore Value = 0

Wait (S – 1) = 3,  Signal (S + 1) = 5

**Wake Up Processes:** 0 – 3 = -3 (3 wake up processes)

**Final Semaphore Value:** 0 – 3 + 5 = 2

## Part 02: Semaphores Coding:

**Source Code:**

```
// Producer Consumer Problem in C


#include <stdio.h>
```

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

#define BUFFER_SIZE 4       // buffer size indicating the maximum number
of items it can hold
#define NUM_ITEMS 4       // total number of items to produce/consume
int buffer[BUFFER_SIZE];  // shared buffer
int in = 0;              // index for the next produced item
int out = 0;             // index for the next consumed item

sem_t empty;            // semaphore to count empty buffer slots
sem_t full;             // semaphore to count full buffer slots
pthread_mutex_t mutex;   // mutex for critical section

void* Producer(void* arg) {
    int id = *((int*)arg);
    for (int i = 0; i < NUM_ITEMS; i++) {
        int item = id * 100 + i;           // produce an item
        sem_wait(&empty);                  // wait for an empty slot
        pthread_mutex_lock(&mutex);        // entering in critical section

        buffer[in] = item;                 // add item to buffer
        in = (in + 1) % BUFFER_SIZE;       // update index for next item

        printf("Producer %d produced %d\n", id, item);
        pthread_mutex_unlock(&mutex);      // leaving from critical section
        sem_post(&full);                   // signal that a new item is available
```

```
    }
    return NULL;
}

void* Consumer(void* arg) {
    int id = *((int*)arg);
    for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full);                    // wait for a full slot
        pthread_mutex_lock(&mutex);         // entering in critical section

        int item = buffer[out];             // remove item from buffer
        out = (out + 1) % BUFFER_SIZE;      // update index for next item

        printf("Consumer %d consumed %d\n", id, item);
        pthread_mutex_unlock(&mutex);       // leaving from critical section
        sem_post(&empty);                   // signal that a slot is free
    }
    return NULL;
}

int main() {
    pthread_t producers[3], consumers[3];
    int producer_ids[3] = {1, 2, 3};
    int consumer_ids[3] = {1, 2, 3};

    sem_init(&empty, 0, BUFFER_SIZE);       // initialize empty slots to
BUFFER_SIZE
    sem_init(&full, 0, 0);                  // initialize full slots to 0
    pthread_mutex_init(&mutex, NULL);       // initialize mutex
```

```
    for (int i = 0; i < 3; i++) {          // create producer and consumer threads
        pthread_create(&producers[i], NULL, Producer, &producer_ids[i]);
        pthread_create(&consumers[i], NULL, Consumer, &consumer_ids[i]);
    }

    for (int i = 0; i < 3; i++) {          // wait for producer and consumers
threads to finish
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }

    sem_destroy(&empty);          // destroy semaphores and mutex
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

**Working of Code:**

This program solves the **Consumer-Product** problem by using semaphore and mutex to provide synchronization among the multiple consumer and products threads as well. The process starts with:

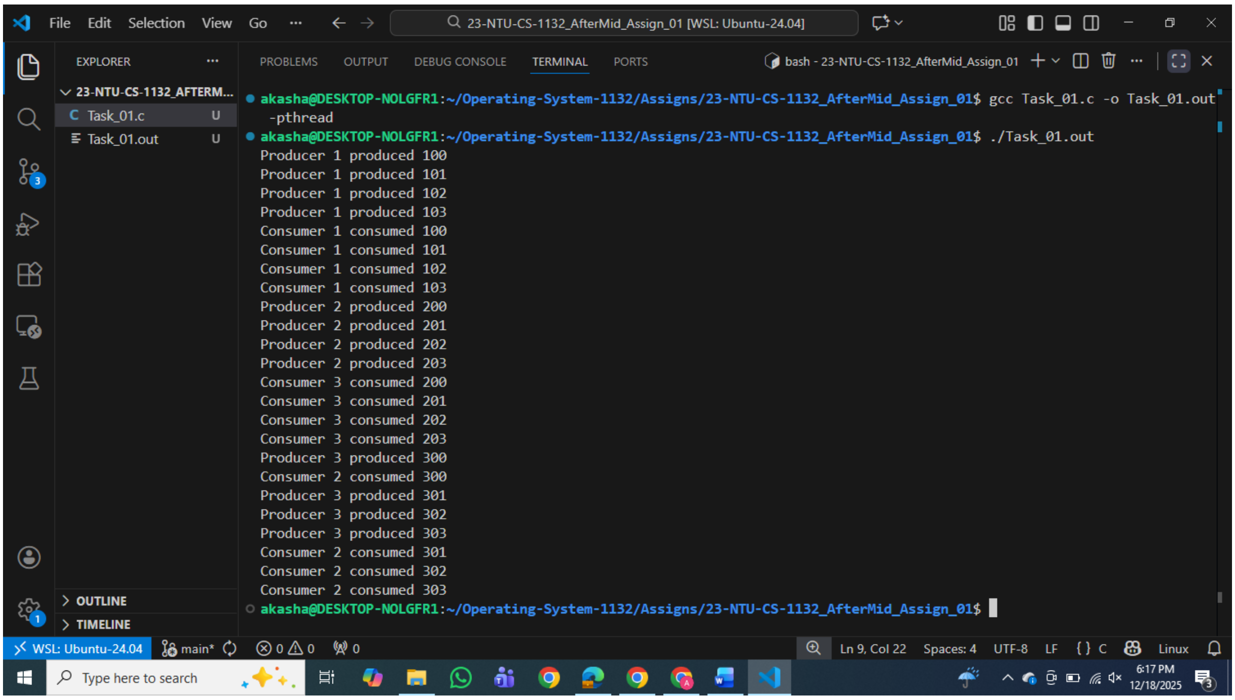
- ❖ The initialization of buffer size and items.
- ❖ Then **in** and **out** variables are initialized to store the indexes of next produced and consumed items.
- ❖ **Producer's** process is created to produce the next item:

  Produce item → wait for empty slot → enter in critical section → item added to buffer → update the **in** index → exit critical section → signal for new item

❖ **Consumer's** process is created to consume the items:

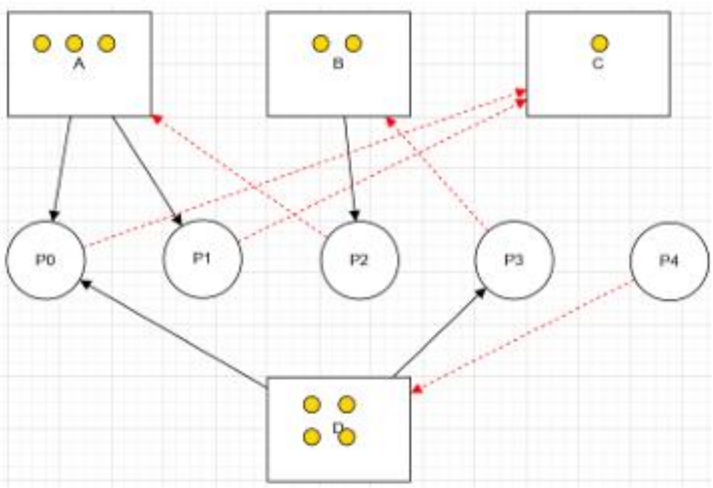
Waiting for full slot → enter in critical section → item removed from buffer → update the **out** index → exit critical section → signal for free slot

**Output:**




# Part 03: RAG (Recourse Allocation Graph):

⇨ **Convert the following graph into matrix table**




## Solution:

**Processes (Rows):** P0, P1, P2, P3, P4

**Resources (Columns):** A, B, C, D

**Resources with Instances:**

| Resources | A | B | C | D |
|---|---|---|---|---|
| **Instances** | 3 | 2 | 1 | 4 |

**Resource Allocation Table:**

| | 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 |

**Resource Request Table:**

| | A | B | C | D |
|---|---|---|---|---|
| **P0** | 0 | 0 | **1** | 0 |
| **P1** | 0 | 0 | **1** | 0 |
| **P2** | **1** | 0 | 0 | 0 |
| **P3** | 0 | **1** | 0 | 0 |
| **P4** | 0 | 0 | 0 | **1** |

**Available Resources with Instances:**

| Resources | A | B | C | D |
|---|---|---|---|---|
| **Instances** | 1 | 1 | 1 | 2 |

## Part 04: Banker's Algorithm:

**Processes (Rows):** P0, P1, P2, P3, P4

**Resources (Columns):** A, B, C, D

**Total Exiting Resources:**

| Resource | A | B | C | D |
|---|---|---|---|---|
| Total | 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 | | | | |

## Question_01: Compute the available Vector:

⇨ **Calculate the available resources for each type of resource.**

| Resources: | A | B | C | D |
|---|---|---|---|---|
| **Total Resources:** | 6 | 4 | 4 | 2 |
| **Allocated:** | 4 | 2 | 2 | 2 |
| **Available:** | 2 | 2 | 2 | 0 |

**Available Vector:** {2, 2, 2, 0}

## Question_02: Compute the Need Matrix:

⇨ **Determine the need matrix by subtracting the allocation matrix from the maximum matrix.**

| | A | B | C | D | Need Vector |
|---|---|---|---|---|---|
| **P0** | 3 – 2 = 1 | 2 – 0 = 2 | 1 – 1 = 0 | 1 – 1 = 0 | **[1, 2, 0, 0]** |

| | | | | | |
|---|---|---|---|---|---|
| **P1** | 1 – 1 = 0 | 2 – 1= 1 | 0 – 0 = 0 | 2- 0 = 2 | **[0, 1, 0, 2]** |
| **P2** | 3 – 1 = 2 | 2 – 0 = 2 | 1 – 1 = 0 | 0 – 0 = 0 | **[2, 2, 0, 0]** |
| **P3** | 2 – 0 = 2 | 1 – 1 = 0 | 0 – 0 = 0 | 1 – 1 = 0 | **[2, 0, 0, 0]** |

**Question_03: 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.**

**Working Array:**{ 2, 2, 2, 0}

| | Need Vector | Working Vector | Allocation Vector | Is Need Vector <= Working Vector | New Working Vector (Work + Alloc) |
|---|---|---|---|---|---|
| **P0** | [1, 2, 0, 0] | [2, 2, 2, 0] | [2, 0, 1, 1] | Yes | [4, 2, 3, 1] |
| **P1** | [0, 1, 0, 2] | [4, 2, 3, 1] | [1, 1, 0, 0] | No | (D need 2 resources while only 1 is available) |
| **P2** | [2, 2, 0, 0] | [4, 2, 3, 1] | [1, 0, 1, 0] | Yes | [5, 2, 4, 1] |

| | | | | | |
|---|---|---|---|---|---|
| **P3** | [2, 0, 0, 0] | [5, 2, 4, 1] | [0, 1, 0, 1] | Yes | [5, 3, 4, 2] |
| **P1** | [0, 1, 0, 2] | [5, 3, 4, 2] | [1, 1, 0, 0] | Yes | [6, 4, 4, 2] (Original Vector) |

## Results:

**Safe Sequence:** P0 → P2 → P3 → P1 (The System is safe)

**Working Array as each Process Terminates:**

[2, 2, 2, 0] → [4, 2, 3, 1] → [5, 2, 4, 1] → [5, 3, 4, 2] → [6, 4, 4, 2]