National Textile University

Department of Computer Science

Subject:

Operating Systems

Submitted To:

Sir Nasir Mehmood

Submitted By:

Alishba Riasat

Registration No:

23-NTU-CS-1135

Assignment:

After Mid Homework-1

Semester:

5th

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.

Answer:

- Initial value of S=7
- After 10 wait () operations, S=7-10= -3
- 3 processes are blocked and 7 processes are in the critical section.
- After 4 signal () operations, S= -3 + 4= 1
- 3 processes wake up after the first 3 signal operations.

S=7-10+4

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.

Answer:

- Initial value of S=3
- After 5 wait () operations, S=3-5= -2
- 3 processes enter the critical section.
- 2 processes are blocked
- Signal () = 6
- The first 2 signals wake up the blocked processes and the remaining 4 signal increments S by 4.
- S= -2 + 6
- S=4

S= 3-5+6

Final value of semaphore=4

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

Answer:

- Initial value of S=0
- After 8 signal () operations, S = 8
- After 3 wait () operations, S=8-3=5
- S=5

S= 0+8-3

Final value of semaphore=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?

Answer:

- Initial value of S=2
- wait () = 5
- S=2-5 = -3

a) 2 processes enter the critical section
b) 3 processes are 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?

Answer:

- Initial value of S=1
- After 3 wait () operations, S=1-3= -2
- 2 processes are blocked
- After 1 signal () operation, 1 process wakeup, S= -2 + 1 = -1

a) 1 process remain blocked
b) Final value of semaphore= -1

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:

- Initial value of S=3
- After 1st wait () operation, S=3-1=2
- After 2nd wait () operation, S=2-1=1
- After 1st signal () operation, S=1+1=2
- After the next wait () operation, S=2-1=1
- After the final wait () operation, S=1-1=0

a) Total processes enter the critical section = 4, as all 4 wait () operations succeed immediately, so 4 processes enter the critical section.
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?

Answer:

- Initial value of S=1
- After 1st wait () operation, S=1-1=0
- After 2nd wait () operation, S=0-1= -1 (1 process is blocked)
- After 1st signal () operation, S= -1+1=0 (1 process wake up)
- After 2nd signal () operation, S=0+1=1

a) At the end, no processes are blocked, the only blocked one was woken up after the first signal operation.
b) Final value of 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?

Answer:

- Initial value of S=1
- After 1st wait () operation, S=1-1=0
- 1 process enter the critical section.
- After 2nd wait () operation, the value of S is checked, which is equal to zero and cannot be decremented further as S can't go below zero in binary semaphore.
- So, 2nd, 3rd, 4th and 5th wait operations processes are blocked.

  a) 1 process enter the critical section
  b) 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?

Answer:

- Initial value of S=4
- After 6 wait () operations, S=4-6= -2
- 2 processes are blocked
- 4 processes enter the critical section and after 4, S becomes 0.
  a) 4 processes proceed
  b) 2 processes are blocked.

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:

a) Track the semaphore value after each operation.

- Initial value of S=2
- After 1st wait () operation, S=2-1=1

- After 2nd wait () operation, S=1-1=0
- After 3rd wait () operation, S=0-1= -1 (1 process is blocked)
- After 1st signal () operation, S= -1+1=0 (1 process wakes up)
- After 2nd signal () operation, S=0+1=1
- After next wait () operation, S=1-1=0
- Final value of S=0

b) How many processes were blocked at any time?

Only 1 process was blocked after 3rd wait operation.

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:

- Initial value of S=0
- After 3 wait () operations, S=0-3= -3
- 3 processes are blocked and no process is in the critical section.
- After 5 signal () operations, S= -3+5= 2
- 3 processes wake up, and final value of S=2
    - **a)** 3 processes wake up.
    - **b)** Final value of S=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.

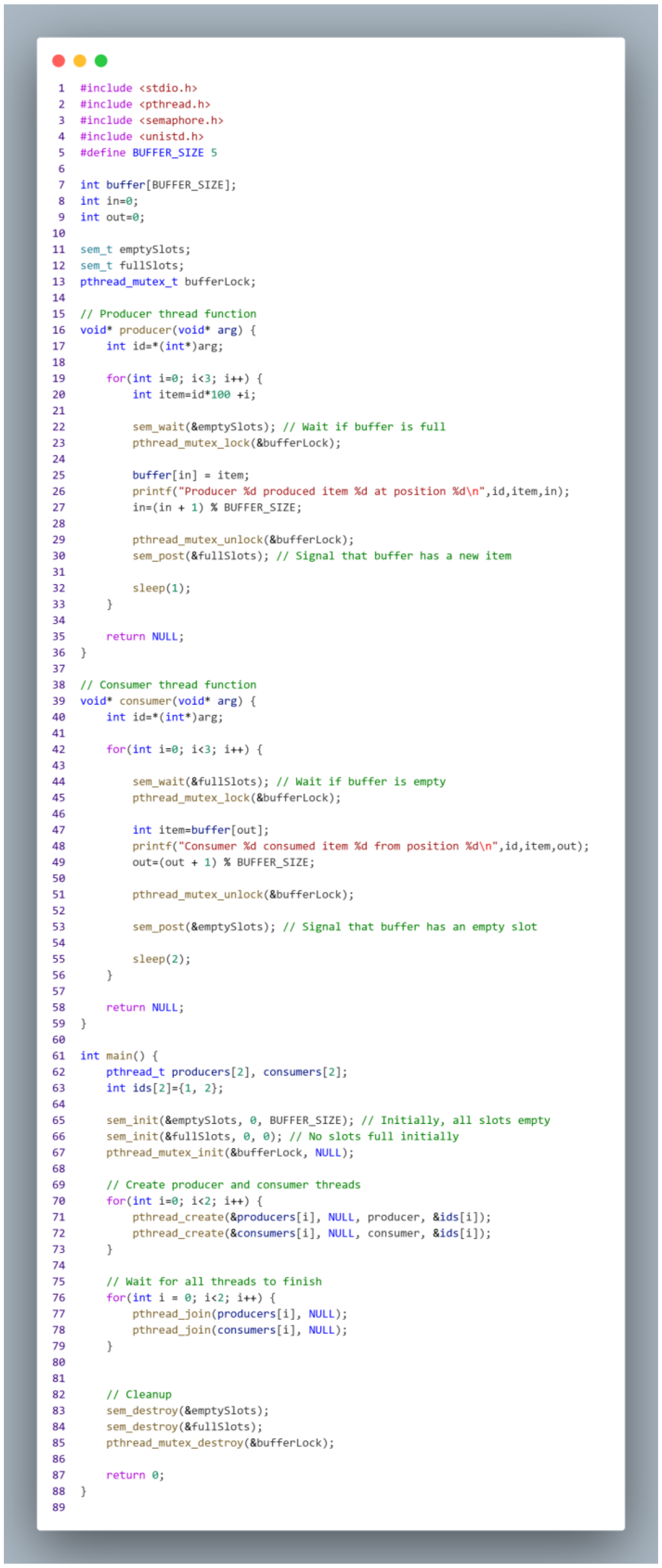
Submission Requirements:

- Your rewritten source codes

- A brief description of how the code works

- Screenshots of the program output showing successful execution

- **Rewritten source code**

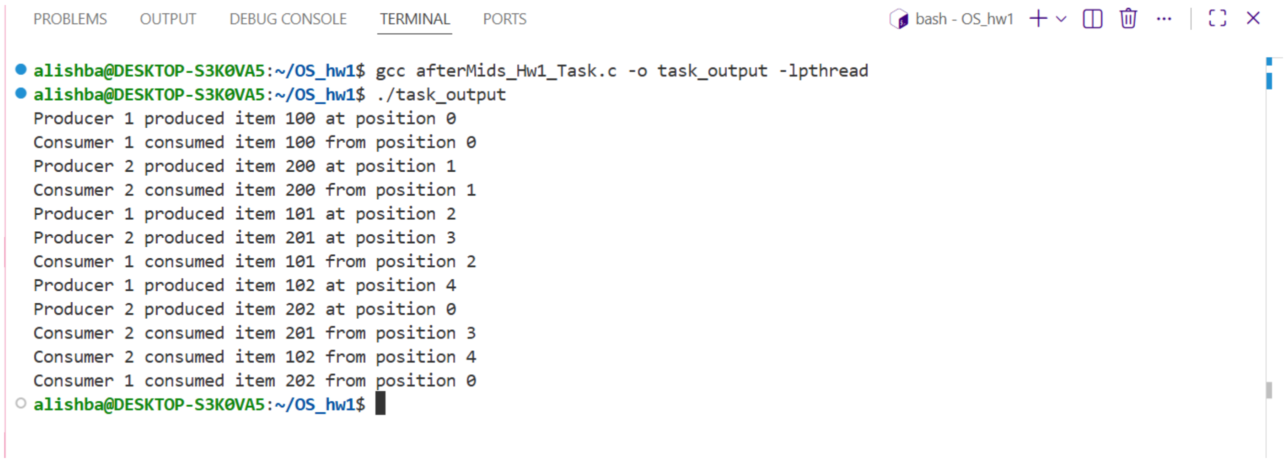


```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_SIZE 5

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

sem_t emptySlots;
sem_t fullSlots;
pthread_mutex_t bufferLock;

// Producer thread function
void* producer(void* arg) {
    int id=*(int*)arg;

    for(int i=0; i<3; i++) {
        int item=id*100 +i;

        sem_wait(&emptySlots); // Wait if buffer is full
        pthread_mutex_lock(&bufferLock);

        buffer[in] = item;
        printf("Producer %d produced item %d at position %d\n",id,item,in);
        in=(in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&bufferLock);
        sem_post(&fullSlots); // Signal that buffer has a new item

        sleep(1);
    }

    return NULL;
}

// Consumer thread function
void* consumer(void* arg) {
    int id=*(int*)arg;

    for(int i=0; i<3; i++) {

        sem_wait(&fullSlots); // Wait if buffer is empty
        pthread_mutex_lock(&bufferLock);

        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(&bufferLock);

        sem_post(&emptySlots); // Signal that buffer has an empty slot

        sleep(2);
    }

    return NULL;
}

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

    sem_init(&emptySlots, 0, BUFFER_SIZE); // Initially, all slots empty
    sem_init(&fullSlots, 0, 0); // No slots full initially
    pthread_mutex_init(&bufferLock, NULL);

    // Create producer and consumer threads
    for(int i=0; i<2; i++) {
        pthread_create(&producers[i], NULL, producer, &ids[i]);
        pthread_create(&consumers[i], NULL, consumer, &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);
    }


    // Cleanup
    sem_destroy(&emptySlots);
    sem_destroy(&fullSlots);
    pthread_mutex_destroy(&bufferLock);

    return 0;
}
```

- **A brief description of how the code works**

    - ➢ In this producer consumer problem, there is a shared buffer whose size is specified to 5. There are two semaphores initialized as empty and full. Empty semaphore counts empty slots in the buffer. Initially all slots are empty and size of empty is same as buffer size.
    - ➢ We have 2 producer threads and 2 consumer threads. One producer thread produces three items as 100, 101 and 102 and other as 200, 201 and 202. Both of the consumer threads run three times to consume total of 6 items.
    - ➢ In producer function, sem_wait(&empty) decrements the empty slots, lock the mutex and go into critical section to produce the item. After that mutex is unlocked and sem_post(&full) increments the full slots which indicates that some items are produced by producer threads.
    - ➢ In consumer function, sem_wait(&full) decrements the full slots, lock the mutex, go into critical section and consumes the items. After that mutex is unlocked and sem_post(&empty) increments the empty slots which indicates that some items are consumed by consumer threads.

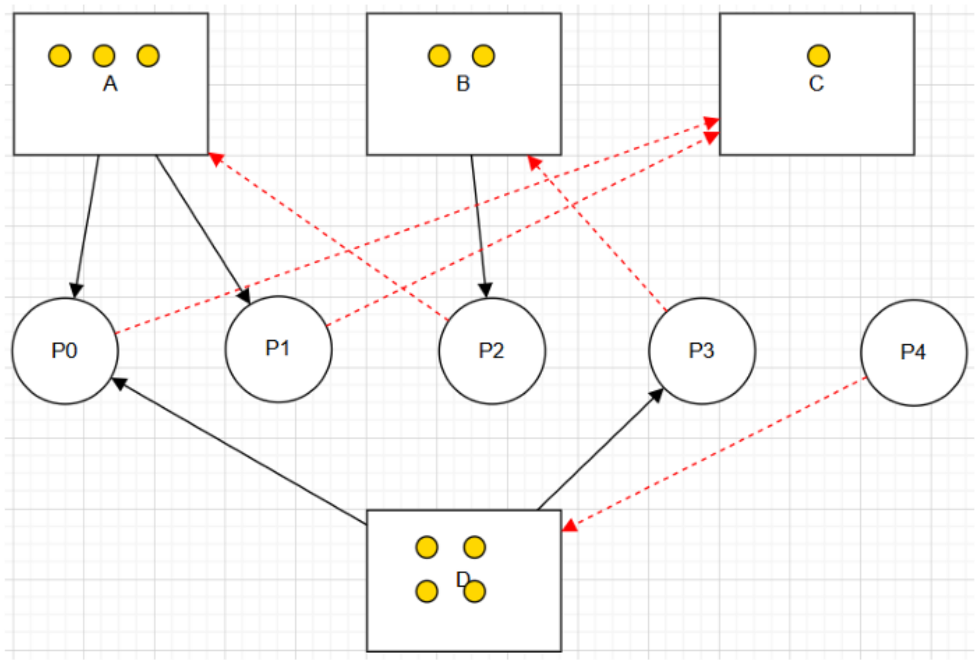
- **Screenshots of the program output showing successful execution**



```
alishba@DESKTOP-S3K0VA5:~/OS_hw1$ gcc afterMids_Hw1_Task.c -o task_output -lpthread
alishba@DESKTOP-S3K0VA5:~/OS_hw1$ ./task_output
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 1 consumed item 101 from position 2
Producer 1 produced item 102 at position 4
Producer 2 produced item 202 at position 0
Consumer 2 consumed item 201 from position 3
Consumer 2 consumed item 102 from position 4
Consumer 1 consumed item 202 from position 0
alishba@DESKTOP-S3K0VA5:~/OS_hw1$
```

Part 3: RAG (Recourse Allocation Graph)

• Convert the following graph into matrix table,



Solution:

Allocation Matrix:

Processes	**Resources**			
	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

Request Matrix:

Processes	**Resources**			
	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 Matrix:

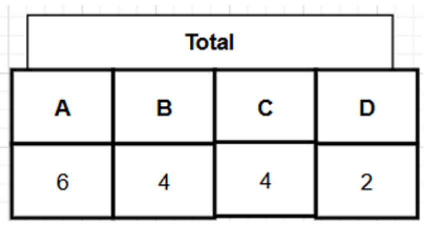
Resource	Total Instances	Total Allocated	Available
A	3	2	**1**
B	2	1	**1**
C	1	0	**1**
D	4	2	**2**

Part 4: Banker's Algorithm

System Description:

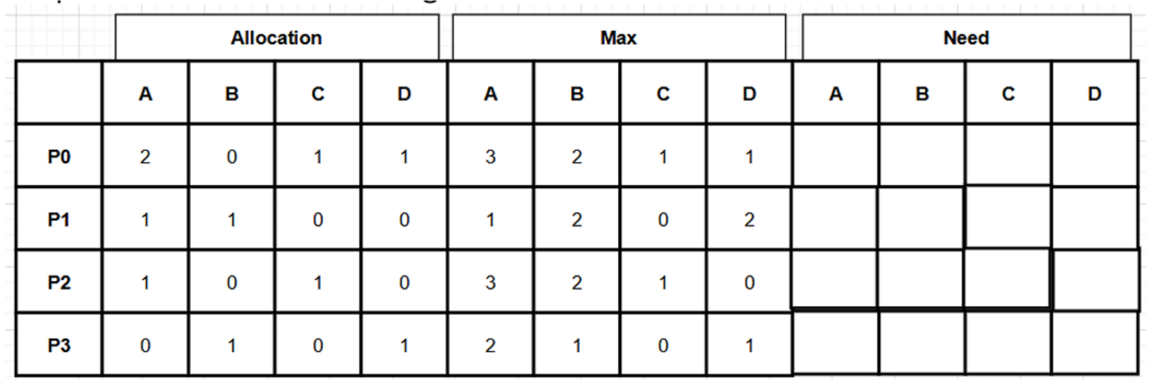
• The system comprises four processes (P0–P3) and four resources (A, B, C, D).

- 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.

Solution:

Total Resources:

A: 6
B: 4
C: 4
D: 2

Sum of allocated resources:

A: 2 + 1 + 1 + 0 = **4**
B: 0 + 1 + 0 + 1 = **2**
C: 1 + 0 + 1 + 0 = **2**
D: 1 + 0 + 0 + 1 = **2**

Resource	***Total - Allocated***	***Available***
A	6 − 4	**2**
B	4 − 2	**2**
C	4 − 2	**2**
D	2 − 2	**0**

Available vector: (2, 2, 2, 0)

Compute the Need Matrix:

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

Solution:

Processes	***Resources***			
	A	**B**	**C**	**D**
P_0	1	2	0	0
P_1	0	1	0	2
P_2	2	2	0	0
P_3	2	0	0	0

2. Safety Check:
   - Determine if the current allocation state is safe. If so, provide a safe sequence of the processes.

Answer:

Yes, the **current allocation state is Safe.** Using the Banker's Algorithm safety check, all processes can complete execution without causing a deadlock.

One possible **safe sequence** is:

$P_0 \rightarrow P_3 \rightarrow P_1 \rightarrow P_2$

All processes can finish execution in the given order and a safe sequence exists, the system is in a safe state.

- Show how the Available (working array) changes as each process terminates.

Solution:

Initial Available Resources:
Available = (2, 2, 2, 0)

Step 1: Execute P_0:

Need of P_0: (1, 2, 0, 0)

Available: (2, 2, 2, 0)

Since Need $\leq$ Available, P_0 can execute

After P_0 finishes, it releases its allocation (2, 0, 1, 1):

Available = (2, 2, 2, 0) + (2, 0, 1, 1)

Available = (4, 2, 3, 1)

Step 2: Execute P_3:

Need of P_3: (2, 0, 0, 0)

Available: (4, 2, 3, 1)

Since Need ≤ Available, P_3 can execute

After P_3 finishes, it releases its allocation (0, 1, 0, 1):

Available = (4,2,3,1) + (0,1,0,1)

Available = (4,3,3,2)

Step 3: Execute P_1:

Need of P_1: (0, 1, 0, 2)

Available: (4,3,3,2)

Since Need ≤ Available, P_1 can execute

After P_1 finishes, it releases its allocation (1, 1, 0, 0):

Available = (4,3,3,2) + (1,1,0,0)

Available = (5,4,3,2)

Step 4: Execute P_2:

Need of P_2: (2, 2, 0, 0)

Available: (5, 4, 3, 2)

Since Need ≤ Available, P_2 can execute

After P_2 finishes, it releases its allocation (1, 0, 1, 0):

Available = (6,4,4,2)

Final Vector: Available = (6,4,4,2)