National Textile University

Department of Computer Science

Subject:

Operating system

Submitted To:

Sir Nasir

Submitted By:

Noor Ul Ain

Registration No:

23-NTU-CS-1221

Assignment No:

2

Semester:

5

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 = 7
   10 wait() // 7 − 10 = −3
   4 signal() // −3 + 4 = 1
   Final value of the 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 = 3
   5 wait() :

   3 − 5 = −2

   6 signal():

   -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.**

   Answer:

   Initial value = 0
   8 signal() :
   0 + 8 = 8
   3 wait() :
   8 − 3 = 5
   Final semaphore 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?**

   Answer:

   Initial value = 2,
   wait() operations=5
   a) How many processes enter the critical section?
   Only 2. The semaphore allows two entries before hitting zero.
   b) How many processes are blocked?
   5 − 2 = 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 = 1
   3 wait(), 1 signal()
   Step-by-step:
   After 3 wait():

   1 − 3 = −2 two processes blocked
   After 1 signal():

   −2 + 1 = −1 ; one process is still blocked

   a) How many processes remain blocked?
   1 process
   b) Final semaphore value = −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:

   semaphore S = 3
   wait(S):
   =2
   wait(S):

=1
signal(S):
= 2
wait(S):
1
wait(S): →0
processes enter the critical section:
→ 4 processes successfully pass wait() while S ≥ 0.
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:

Start: **S = 1**

1. `wait(S)` → 1 → 0
   One process enters
2. `wait(S)` → 0 → −1
   One process is blocked
3. `signal(S)` −1 → 0
   One blocked process wakes up and enters
4. `signal(S)` → 0 → 1
   Extra free spot added

- ✓ **only one process is blocked** (only the second `wait()` caused blocking)
- ✓ **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:

Five `wait()` operations, **no signal()**

Binary semaphore means **only one process allowed at a time**.

Start: **S = 1**

• First `wait()` cause one process to enter.
• Remaining 4 `wait()` gets 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?

Answer:

As the semaphore is initialized to 4 only 4 processes will be allowed to enter the critical section while the remaining two gets 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:

Semaphore initial value is 2

- Wait() = 2-1=>1 (one process enters the critical section)
- Wait()=1-1 =>0 (second process enters the critical section)
- Wait()=0-1 => -1 (next process is blocked)
- Signal()=-1+1 =>0 (one process leaves the critical section)
- Signal()=0+1 =>1 (another process leaves the critical section)
- Wait()= 1-1 => 0 (Third process enters the critical section)
    - One process was blocked at the third 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:

Wait() 0-1=>-1

Wait() -1-1=>-2

Wait() -2-1=>-3
Signal() -3+1=>-2
Signal() -2+1=>-1
Signal() -1+1=>0
Signal() 0+1=>1
Signal() 1+1=>2
Thress process gets wake up during first 3 signals and final semaphore value is 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

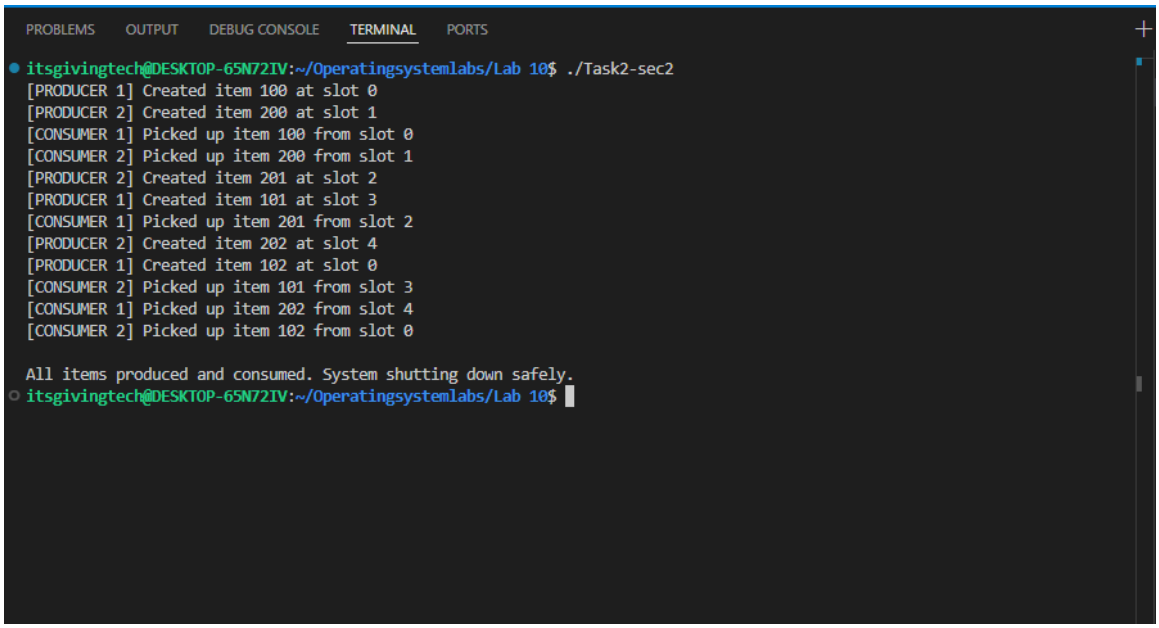
Code:

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


#define BUFFER_SIZE 5
#define ITEMS_PER_THREAD 3

int shared_buffer[BUFFER_SIZE];
int write_index = 0;
int read_index  = 0;
```

```c
sem_t slots_available;
sem_t items_ready;
pthread_mutex_t buffer_lock;


void* producer_workflow(void* thread_id_ptr) {
    int producer_id = *(int*)thread_id_ptr;

    for (int i = 0; i < ITEMS_PER_THREAD; i++) {
        int new_item = (producer_id * 100) + i;


        sem_wait(&slots_available);


        pthread_mutex_lock(&buffer_lock);


        shared_buffer[write_index] = new_item;
        printf("[PRODUCER %d] Created item %d at slot %d\n", producer_id,
new_item, write_index);
        write_index = (write_index + 1) % BUFFER_SIZE;


        pthread_mutex_unlock(&buffer_lock);
        sem_post(&items_ready);

        sleep(1);
    }
    return NULL;
}

void* consumer_workflow(void* thread_id_ptr) {
    int consumer_id = *(int*)thread_id_ptr;

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


        sem_wait(&items_ready);


        pthread_mutex_lock(&buffer_lock);


        int item_received = shared_buffer[read_index];
```

```
        printf("[CONSUMER %d] Picked up item %d from slot %d\n",
consumer_id, item_received, read_index);
        read_index = (read_index + 1) % BUFFER_SIZE;


        pthread_mutex_unlock(&buffer_lock);
        sem_post(&slots_available);

        sleep(2);
    }
    return NULL;
}

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


    sem_init(&slots_available, 0, BUFFER_SIZE);
    sem_init(&items_ready, 0, 0);
    pthread_mutex_init(&buffer_lock, NULL);


    for (int i = 0; i < 2; i++) {
        pthread_create(&producers[i], NULL, producer_workflow,
&thread_ids[i]);
        pthread_create(&consumers[i], NULL, consumer_workflow,
&thread_ids[i]);
    }


    for (int i = 0; i < 2; i++) {
        pthread_join(producers[i], NULL);
        pthread_join(consumers[i], NULL);
    }


    sem_destroy(&slots_available);
    sem_destroy(&items_ready);
    pthread_mutex_destroy(&buffer_lock);

    printf("\nAll items produced and consumed. System shutting down
safely.\n");
    return 0;
}
```

Output:

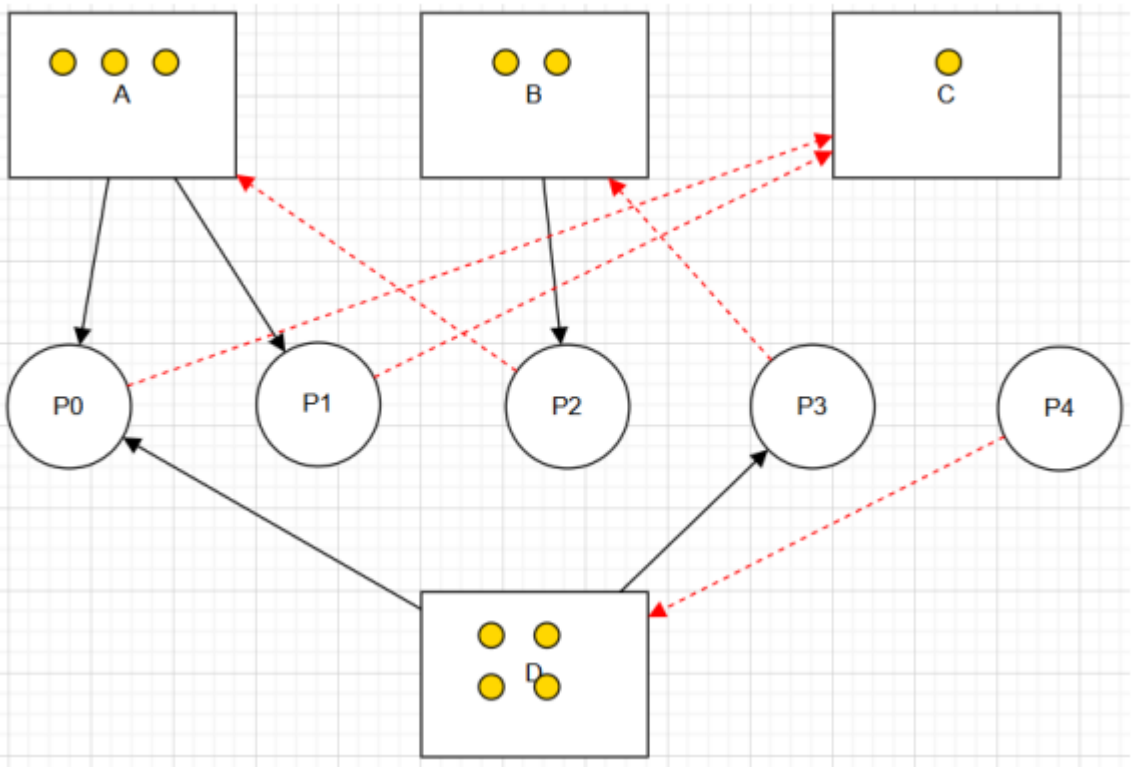


Remarks:

This program simulates a factory where producers create items and consumers use them through a shared buffer. Semaphores are used to make sure producers do not add items when the buffer is full and consumers do not remove items when it is empty. A mutex lock ensures that only one thread accesses the buffer at a time, preventing errors. By coordinating producers and consumers safely, the program ensures that all items are produced and consumed correctly without conflicts or data loss.

Part 3:

RAG (Recourse Allocation Graph) • Convert the following graph into matrix table.



	Allocation			
	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

Waiting:

	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

Part 4:

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				

Need matrix:

	A	**B**	**C**	**D**
P0	**1**	**2**	**0**	**0**
P1	**0**	**1**	**0**	**2**
P2	**2**	**2**	**0**	**0**
P3	**2**	**0**	**0**	**0**

Available resources:

A :2

B:2

C:2

D:0

Available vector is (2,2,2,0)

Safety check:

. Check P0

- **Need:** (1, 2, 0, 0)
- **Available:** (2, 2, 2, 0)
- **Can it run? Yes**, (1, 2, 0, 0) <= (2, 2, 2, 0)
- **New Available:** (2, 2, 2, 0) + Allocation (2, 0, 1, 1) = **(4, 2, 3, 1)**

2. Check P1

- **Need:** (0, 1, 0, 2)
- **Available:** (4, 2, 3, 1)
- **Can it run? No**, because Need for Resource D (2) is greater than Available D (1).

3. Check P2

- **Need:** (2, 2, 0, 0)
- **Available:** (4, 2, 3, 1)
- **Can it run? Yes**, (2, 2, 0, 0) <= (4, 2, 3, 1).
- **New Available:** (4, 2, 3, 1) + Allocation (1, 0, 1, 0) = **(5, 2, 4, 1)**

Check P3

- **Need:** (2, 0, 0, 0)
- **Available:** (5, 2, 4, 1)
- **Can it run? Yes**, (2, 0, 0, 0) <=(5, 2, 4, 1).
- **New Available:** (5, 2, 4, 1) + Allocation (0, 1, 0, 1) = **(5, 3, 4, 2)**

. Return to P1 (The only remaining process)

- **Need:** (0, 1, 0, 2)
- **Available:** (5, 3, 4, 2)
- **Can it run? Yes**, (0, 1, 0, 2) <= (5, 3, 4, 2).
- **New Available:** (5, 3, 4, 2) + Allocation (1, 1, 0, 0) = **(6, 4, 4, 2)**

Sequence of processes : P0→P2→P3→P1

Step	Process	Work (Available)	Release (Allocation)	New Available
1	**P0**	(2, 2, 2, 0)	(2, 0, 1, 1)	**(4, 2, 3, 1)**
2	**P2**	(4, 2, 3, 1)	(1, 0, 1, 0)	**(5, 2, 4, 1)**
3	**P3**	(5, 2, 4, 1)	(0, 1, 0, 1)	**(5, 3, 4, 2)**
4	**P1**	(5, 3, 4, 2)	(1, 1, 0, 0)	**(6, 4, 4, 2)**