National Textile University
Department of Computer Science

Subject:
Operating System

Submitted to:

Sir Nasir

Submitted by:

Asbah Asif

Reg number:

23-NTU-CS-1141

Assignment no: Homework (After mids)

Semester: SE-A-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.

Solution:

$$S\ (final) = S\ (initial) - \#wait + \#signal$$

Initial = 7
Wait = 10
Signal = 4
Final = $7 - 10 + 4 = 1$
Final S = 1

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

$$S\ (final) = S\ (initial) - \#wait + \#signal$$

Initial = 3
Wait = 5
Signal = 6
Final = $3 - 5 + 6 = 4$
Final S = 4

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

$$S\ (final) = S\ (initial) - \#wait + \#signal$$

Initial = 0
Wait = 3
Signal = 8
Final = $0 - 3 + 8 = 5$
Final S = 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?

- first 2 waits succeed (S=0), 2 processes enter.
- Next 3 waits: block.
  Enter = 2
  Blocked = 3

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?

- First wait: S=0, 1 enters.
- Next 2 waits: block 2.
- 1 signal (): unblocks 1 of the 2 blocked
  Blocked = 1
  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?

Applying wait ():
S=3-1=2
S=2-1=1
Applying signal ():
S=1+1=2
Applying wait ():
S=2-1=1
S=1-1=0

a) Process enters critical section = 4
b) Final semaphore value = 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?

Applying wait ():
S=1-1=0
S=0-1=-1
Applying signal ():
S=-1+1=0
S=0+1=1

a) processes blocked= 1 (when 1-1=0), but at next step it was unblocked
b) final semaphore value= 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?

Applying wait ():
S=1-1=0
S=0
S=0
S=0
S=0

a) processes enter critical section = 1
b) blocked processes = 4

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

Applying wait ():
S=4-1=3
S=3-1=2
S=2-1=1
S=1-1=0
S=0-1=-1
S=-1-1=-2

a) processes proceed = 4
b) 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.
b) How many processes were blocked at any time?

S=2
Applying wait ():
2-1=1 (enter critical section)
1-1=0 (enter critical section)
0-1=-1 (Blocked)
Applying signal:
-1+1=0 (unblocked)
0+1=1
Applying wait:
1-1=0 (enter critical section)

a) semaphore value after each operation = 1, 0, -1, 0, 1, 0
b) Blocked processes = 1

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?

Applying wait ():
S=0-1=-1
S=-1-1=-2
S=-2-1=-3
Applying signal ():
S=-3+1=-2
S=-2+1=-1
S=-1+1=0
S=0+1=1
S=1+1=2

a) processes wakeup = 3 (3 blocked processes wakeup at signal)
b) final semaphore value = 2

Part 2: Semaphore Coding:

Part 1:

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

#define BUFFER_SIZE 5

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

sem_t empty_slots;
sem_t full_slots;
pthread_mutex_t buffer_lock;

void* producer(void* arg) {
   int producer_id = *(int*)arg;

   for (int i = 0; i < 3; i++) {
      int item = producer_id * 100 + i + 1;

      // empty slot ka wait
      sem_wait(&empty_slots);

      pthread_mutex_lock(&buffer_lock);

buffer[in] = item;
      printf("[PRODUCER %d] Produced item %d at position %d\n",
            producer_id, item, in);
      in = (in + 1) % BUFFER_SIZE;
```

```
        // Unlock buffer
        pthread_mutex_unlock(&buffer_lock);

        // slot full signal
        sem_post(&full_slots);

        sleep(1);  // Simulate production time
    }

    return NULL;
}

void* consumer(void* arg) {
    int consumer_id = *(int*)arg;

    for (int i = 0; i < 3; i++) {
        // Wait full slot ka
        sem_wait(&full_slots);

        // Lock buffer before reading
        pthread_mutex_lock(&buffer_lock);

        // Consume item from buffer
        int item = buffer[out];
        printf("[CONSUMER %d] Consumed item %d from position %d\n",
               consumer_id, item, out);
        out = (out + 1) % BUFFER_SIZE;

        // Unlock buffer
        pthread_mutex_unlock(&buffer_lock);

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

        sleep(2);  // Simulate consumption time (slower than producer)
    }

    return NULL;
}

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

    // Initialize semaphores
    sem_init(&empty_slots, 0, BUFFER_SIZE);  // All slots empty initially
    sem_init(&full_slots, 0, 0);             // No slots full initially
    pthread_mutex_init(&buffer_lock, NULL);

    // Create producer and consumer threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&producers[i], NULL, producer, &producer_ids[i]);
        pthread_create(&consumers[i], NULL, consumer, &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(&empty_slots);
    sem_destroy(&full_slots);
    pthread_mutex_destroy(&buffer_lock);

    printf("\nAll producers and consumers have finished successfully.\n");
    return 0;
}
```

Description:

This program solves the Producer–Consumer Problem using semaphores and a mutex to safely manage a shared fixed-size buffer.

- Producers add items to the buffer if space is available.
- Consumers remove items if the buffer is not empty.
- Empty/full semaphores prevent overflow/underflow.
- A mutex ensures only one thread accesses the buffer at a time.
  The code runs with 2 producers and 2 consumers, each handling 3 items, demonstrating correct synchronization without race conditions or deadlock.

Example 2:

```
#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 empty;
sem_t full;
pthread_mutex_t mutex;

void* producer(void* arg) {
    int id = *(int*)arg;

    for (int i = 0; i < 3; i++) {
        int item = id * 100 + i + 1;

        // Wait for empty slot
        sem_wait(&empty);

        // Lock buffer
        pthread_mutex_lock(&mutex);

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

        // Unlock buffer
        pthread_mutex_unlock(&mutex);

        // Signal full slot
        sem_post(&full);

        sleep(1);
    }

    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;

    for (int i = 0; i < 3; i++) {
        // Wait for full slot
        sem_wait(&full);

        // Lock buffer
        pthread_mutex_lock(&mutex);
```

```
        // Consume item
        int item = buffer[out];
        printf("Consumer %d consumed item %d from position %d\n", id, item, out);
        out = (out + 1) % BUFFER_SIZE;

        // Unlock buffer
        pthread_mutex_unlock(&mutex);

        // Signal empty slot
        sem_post(&empty);

        sleep(2);
    }

    return NULL;
}

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

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);  // All slots empty
    sem_init(&full, 0, 0);             // No slots full
    pthread_mutex_init(&mutex, NULL);

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

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

    // Cleanup
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    printf("\nAll operations completed successfully.\n");
    return 0;
}
```

Description:

Main Components:

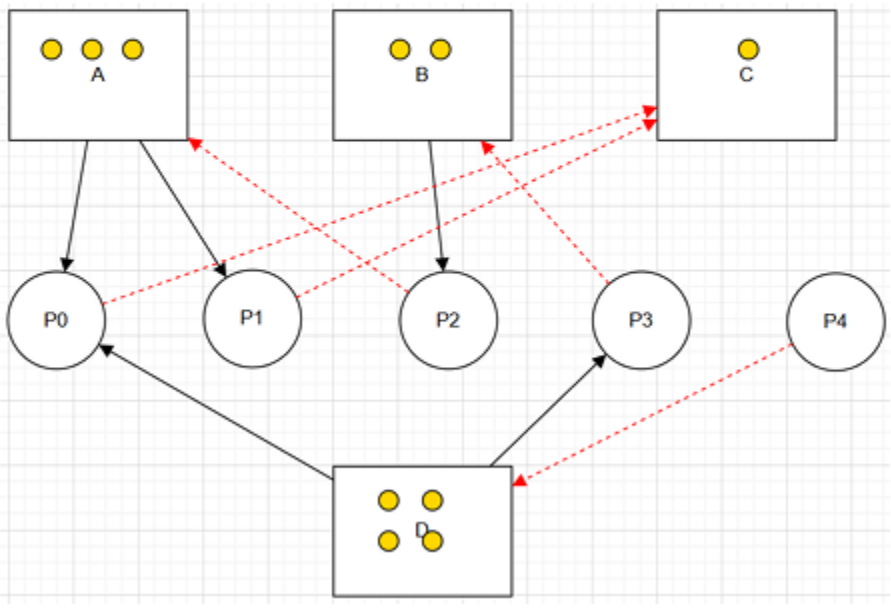
1. **Shared Buffer** – Fixed-size array (size 5) that stores items.
2. **Producers** – Threads that create items and add them to the buffer.
3. **Consumers** – Threads that remove items from the buffer.
4. **Semaphores**:
    - empty – Tracks empty slots in buffer (starts at 5).
    - full – Tracks filled slots (starts at 0).
5. **Mutex** – Ensures only one thread accesses the buffer at a time.

How It Works:

- **Producers** wait for an empty slot (sem_wait(&empty)), lock the buffer, add an item, unlock, and signal a full slot (sem_post(&full)).
- **Consumers** wait for a full slot (sem_wait(&full)), lock the buffer, remove an item, unlock, and signal an empty slot (sem_post(&empty)).

Part 3: RAG (Recourse Allocation Graph)

- Convert the following graph into matrix table ,



Processes = P0 - P4
Resources = A, B, C, D

Allocation Matrix:

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

Request Matrix:

	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: 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.

2.Compute the Need Matrix:
• Determine the need matrix by subtracting the allocation matrix from the maximum matrix.

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.

Solution:
Total Resources:
A = 6, B = 4, C = 4, D = 2

Allocation Matrix:

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

Max Matrix:

	A	B	C	D
P0	3	2	1	1
P1	1	2	0	2
P2	3	2	1	0
P3	2	1	0	1

1. Available Vector

Available = Total − Sum of Allocated

Sum of Allocation:

- **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 = 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. Need Matrix:

Need = Max − Allocation

- **P0:**
  A: 3 − 2 = 1
  B: 2 − 0 = 2
  C: 1 − 1 = 0
  D: 1 − 1 = 0
- **P1:**
  A: 1 − 1 = 0
  B: 2 − 1 = 1
  C: 0 − 0 = 0
  D: 2 − 0 = 2
- **P2:**
  A: 3 − 1 = 2
  B: 2 − 0 = 2
  C: 1 − 1 = 0
  D: 0 − 0 = 0
- **P3:**
  A: 2 − 0 = 2
  B: 1 − 1 = 0
  C: 0 − 0 = 0
  D: 1 − 1 = 0

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

3. Safety Check

Available = [2, 2, 2, 0]
Step 1:
Check P0: Need [1, 2, 0, 0] ≤ Available [2, 2, 2, 0] → Yes.
Available += Allocation[P0] = [2, 2, 2, 0] + [2, 0, 1, 1] = [4, 2, 3, 1]
Finished[P0] = True
Sequence = [P0]

Step 2:
Check P1: Need [0, 1, 0, 2] ≤ Available [4, 2, 3, 1] → No (D: 2 > 1) → Skip.
Check P2: Need [2, 2, 0, 0] ≤ Available [4, 2, 3, 1] → Yes.
Available += [1, 0, 1, 0] = [5, 2, 4, 1]
Finished[P2] = True
Sequence = [P0, P2]

Step 3:
Check P1 again: Need [0, 1, 0, 2] ≤ Available [5, 2, 4, 1] → No (D: 2 > 1) → Skip.
Check P3: Need [2, 0, 0, 0] ≤ Available [5, 2, 4, 1] → Yes.
Available += [0, 1, 0, 1] = [5, 3, 4, 2]
Finished[P3] = True
Sequence = [P0, P2, P3]

Step 4:
Now only P1 left: Need [0, 1, 0, 2] ≤ Available [5, 3, 4, 2] → Yes.
Available += [1, 1, 0, 0] = [6, 4, 4, 2]
Finished[P1] = True
Sequence = [P0, P2, P3, P1]

- System is SAFE
- Safe sequence: P0 → P2 → P3 → P1