HA-3

1. Demonstrate how communication and data sharing among threads is faster than processes and illustrate the reasons for this.

Thread Communication vs. Process Communication

1. **Shared Memory:** Threads share the same address space, avoiding IPC overhead.
2. **Lower Context Switching:** No need to switch memory mappings.
3. **Efficient Synchronization:** Mutexes & condition variables are faster than IPC mechanisms.

Illustration: Thread vs. Process Communication

Thread Communication (Faster, Shared Memory)

```
int shared_data = 0;
void *thread_function() {
   shared_data++;
}
```

Process Communication (Slower, Needs IPC)

```
shm_id = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
char *shm_ptr = shmat(shm_id, NULL, 0);
strcpy(shm_ptr, "Data Transfer");
```

Key Takeaways

- Threads access shared memory directly.
- Processes need extra IPC mechanisms.
- Threads are significantly faster than processes for communication.

2. Apply your knowledge to explain System V Shared Memory with an illustration.

System V Shared Memory (SHM)
An IPC mechanism where multiple processes share a memory segment for fast communication.

Steps:
1. Create/Get SHM → shmget()
2. Attach → shmat()
3. Read/Write Data
4. Detach → shmdt()
5. Remove → shmctl()

Illustration:
- Process A creates SHM, writes data, detaches.
- Process B attaches, reads data, detaches.
- SHM removed when no longer needed.

3. Explain what happens to running threads when a process exits and describe the function prototypes for thread creation and exit.

Threads and Process Exit

- If a process exits, all threads terminate immediately.
- If a thread calls exit(), the entire process exits.
- If a thread calls pthread_exit(), only that thread exits.

Function Prototypes

- **Create Thread:**

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

- **Exit Thread:**

```
void pthread_exit(void *retval);
```

4. Demonstrate how a parent thread can check whether a child thread has completed before continuing (commonly called join ()) using condition variables and explain the pseudocode for this implementation.

Thread Join Using Condition Variables

Steps:

1. Parent creates child thread.
2. Child sets done = 1, signals cond, unlocks mutex.
3. Parent waits on cond, resumes when signaled.

C Code:

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int done = 0;

void *child_thread(void *arg) {
   pthread_mutex_lock(&mutex);
   done = 1;
   pthread_cond_signal(&cond);

   pthread_mutex_unlock(&mutex);
   return NULL;
}

int main() {

pthread_t thread;
   pthread_create(&thread, NULL, child_thread, NULL);

   pthread_mutex_lock(&mutex);
   while (!done) pthread_cond_wait(&cond, &mutex);
   pthread_mutex_unlock(&mutex);

   printf("Child completed.\n");
   return 0;
}
```

5. Analyze how two unsynchronized threads incrementing the same variable can lead to a race condition. Provide pseudocode that includes a critical section and protects it using mutex locks to prevent the race condition.

Race Condition in Unsynchronized Threads

- **Issue:** Two threads modify counter simultaneously, causing incorrect updates.

Without Mutex (Race Condition)

```
counter = 0;
function increment() {
   temp = counter;
   temp = temp + 1;
   counter = temp;
}
```

Problem: Lost updates due to concurrent access.

With Mutex (Safe)

```
mutex lock;
counter = 0;
function increment() {
   lock(mutex);
   counter = counter + 1;
   unlock(mutex);
}
```

Solution: Mutex ensures only one thread updates counter at a time.

6. Illustrate the implementation of a concurrent linked list that allows only one thread to access the entire list at any given instant. Demonstrate how this can be achieved through synchronization mechanisms.

Thread-Safe Linked List (C - Pthreads)

Uses mutex to ensure only one thread accesses the list.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int data;
   struct Node *next;
} Node;

Node *head = NULL;
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;

void insert(int value) {
   pthread_mutex_lock(&list_mutex);
   Node *newNode = (Node *)malloc(sizeof(Node));
   newNode->data = value;
   newNode->next = head;
   head = newNode;
   pthread_mutex_unlock(&list_mutex);
}

void display() {
   pthread_mutex_lock(&list_mutex);
   Node *temp = head;
   while (temp) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
   pthread_mutex_unlock(&list_mutex);
}

int main() {
   insert(10);
   insert(20);
```

```
    insert(30);
    display();
    return 0;
}
```

7. Differentiate the inter-process communication (IPC) methods provided by pipes and shared memory. Discuss the advantages and disadvantages of each method and determine scenarios where one might be more appropriate than the other.

Pipes vs. Shared Memory (IPC)

Feature	Pipes	Shared Memory
Mechanism	Data stream	Shared memory region
Speed	Slower (system calls)	Faster (direct access)
Sync	Automatic (blocking I/O)	Needs mutex/semaphores
Persistence	Data lost on exit	Data persists until removed
Best For	Simple IPC (parent-child)	High-speed, large data sharing

Usage:

- Pipes → Sequential communication (e.g., shell commands).
- Shared Memory → Fast, frequent data transfer (e.g., real-time apps).

8. Demonstrate how two unsynchronized threads incrementing the same variable can lead to a race condition. Develop a pseudocode that includes a critical section and protects it using mutex locks to prevent the race condition.

Race Condition (Without Mutex)

```
counter = 0;
function increment() {
   temp = counter;
   temp = temp + 1;
   counter = temp;
}
```

Issue: Concurrent access leads to incorrect updates.

Fixed (With Mutex)

```
mutex lock;
function increment() {
   lock(mutex);
   counter = counter + 1;
   unlock(mutex);
}
```

Solution: Mutex ensures only one thread modifies counter at a time.

9. Discuss the key components of a Thread API in an operating system. Provide an overview of functions and methods commonly found in a Thread API and explain their significance in multithreaded programming.

Key Components of a Thread API

1. Thread Creation – pthread_create(), std::thread (Creates a new thread).
2. Thread Termination – pthread_exit(), std::this_thread::exit() (Ends a thread).
3. Synchronization – pthread_mutex_lock(), pthread_cond_wait() (Prevents race conditions).
4. Thread Joining – pthread_join() (Waits for a thread to finish).
5. Thread Detachment – pthread_detach() (Allows independent execution).
6. Thread Attributes – pthread_attr_init(), pthread_attr_setstacksize() (Configures properties).

Importance

- Enables concurrency and better CPU utilization.
- Requires synchronization to prevent race conditions.

10. Discuss the reader-writer problem, where multiple readers and writers contend for access to a shared resource. Explain the challenges of balancing reader and writer access and provide a code example that demonstrates a solution using semaphores.

Reader-Writer Problem

- Readers can read simultaneously.
- Writers need exclusive access.
- Challenge: Prevent starvation while ensuring efficiency.

Semaphore Solution (C - Pthreads)

```
sem_t rw_mutex, mutex;
int read_count = 0;

void *reader(void *arg) {
   sem_wait(&mutex);
   if (++read_count == 1) sem_wait(&rw_mutex);
   sem_post(&mutex);

   printf("Reader reading...\n");

   sem_wait(&mutex);
   if (--read_count == 0) sem_post(&rw_mutex);
   sem_post(&mutex);
}

void *writer(void *arg) {
   sem_wait(&rw_mutex);
   printf("Writer writing...\n");
   sem_post(&rw_mutex);
}
```

11. Explain in detail the locks, locks data structures and condition variables, illustrate with the help of an example.

Locks, Lock Data Structures, and Condition Variables

1. Locks

Ensure mutual exclusion, preventing multiple threads from accessing shared resources.

2. Lock Data Structures

- Mutex: Allows one thread at a time.
- Spinlock: Busy-wait lock for short critical sections.
- Read-Write Lock: Multiple readers, one writer.

3. Condition Variables

Used for synchronization, allowing threads to wait for conditions before proceeding.

Example: Mutex & Condition Variable (C - Pthreads)

```
pthread_mutex_t lock;
pthread_cond_t cond;
int ready = 0;

void *worker(void *arg) {
   pthread_mutex_lock(&lock);
   while (!ready) pthread_cond_wait(&cond, &lock);
   printf("Worker running!\n");
   pthread_mutex_unlock(&lock);
}

void *controller(void *arg) {
   pthread_mutex_lock(&lock);
   ready = 1;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&lock);
}
```

12. Describe the concept of thread synchronization and mutual exclusion. Write a code example that uses synchronization primitives (e.g., semaphores or mutexes) to protect a shared resource accessed by multiple threads, preventing data corruption.

Thread Synchronization & Mutual Exclusion

- Synchronization ensures correct execution order.
- Mutex prevents simultaneous access to shared resources.

Example: Mutex (C - Pthreads)

```
pthread_mutex_t lock;
int counter = 0;

void *increment(void *arg) {
   pthread_mutex_lock(&lock);
   counter++;
   printf("Counter: %d\n", counter);
   pthread_mutex_unlock(&lock);
}
```

13. Illustrate how semaphores or mutex locks can be used to solve the Readers-Writers problem.

Readers-Writers Problem Using Semaphores

- Readers read simultaneously.
- Writers need exclusive access.
- Semaphores ensure mutual exclusion.

Implementation (C - Pthreads)

```
sem_t rw_mutex, mutex;
int read_count = 0;

void *reader(void *arg) {
   sem_wait(&mutex);
   if (++read_count == 1) sem_wait(&rw_mutex);
   sem_post(&mutex);

   printf("Reader reading...\n");

   sem_wait(&mutex);
   if (--read_count == 0) sem_post(&rw_mutex);
   sem_post(&mutex);
}

void *writer(void *arg) {
   sem_wait(&rw_mutex);
   printf("Writer writing...\n");
   sem_post(&rw_mutex);
}
```

14. Identify the role of the Resource Allocation Graph algorithm to deal with deadlock problem and give the limitations of this approach.

Resource Allocation Graph (RAG) for Deadlock Detection

- **Role:**
  - Models processes and resources as a graph.
  - Detects cycles indicating deadlock.

Limitations

- Only for single-instance resources.
- Cycle detection is computationally expensive.
- Does not prevent deadlock, only detects it.

15. Provide a comprehensive explanation regarding concurrency, including its fundamental principles, common problems, and challenges, notable advantages, as well as drawbacks and issues associated with concurrency in operating systems.

Concurrency in Operating Systems

1. Fundamental Principles

- Parallel execution of multiple tasks.
- Context switching enables multitasking.
- Synchronization ensures correct execution order.

2. Common Problems & Challenges

- Race conditions: Multiple threads modifying shared data.
- Deadlocks: Circular waiting for resources.
- Starvation: Some threads never get CPU time.

3. Advantages

- Improved CPU utilization and responsiveness.
- Efficient resource sharing.
- Faster execution of tasks.

4. Drawbacks & Issues

- Increased complexity in programming.
- Synchronization overhead.
- Difficult debugging due to non-deterministic execution.

16. How two unsynchronized threads incrementing the same variable lead to a race condition. Give pseudocode that has a critical section and protect it with mutex locks.

Race Condition in Unsynchronized Threads

- Problem: Multiple threads modifying a shared variable simultaneously.
- Effect: Inconsistent or incorrect results.

Pseudocode Without Mutex (Race Condition)

```
int counter = 0;
void *increment() {
   counter++;
}
```

- Threads may read/update counter at the same time, causing incorrect values.

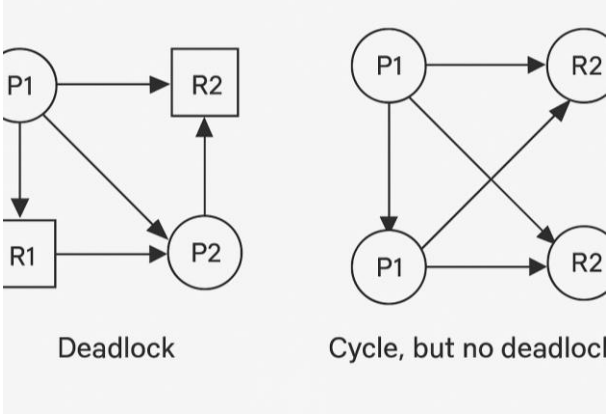
Pseudocode With Mutex (Safe Execution)

```
pthread_mutex_t lock;
int counter = 0;

void *increment() {
   pthread_mutex_lock(&lock);
   counter++;
   pthread_mutex_unlock(&lock);
}
```

17. Illustrate a Resource Allocation Graph showing a deadlock and a graph with a cycle but no deadlock using diagrams.

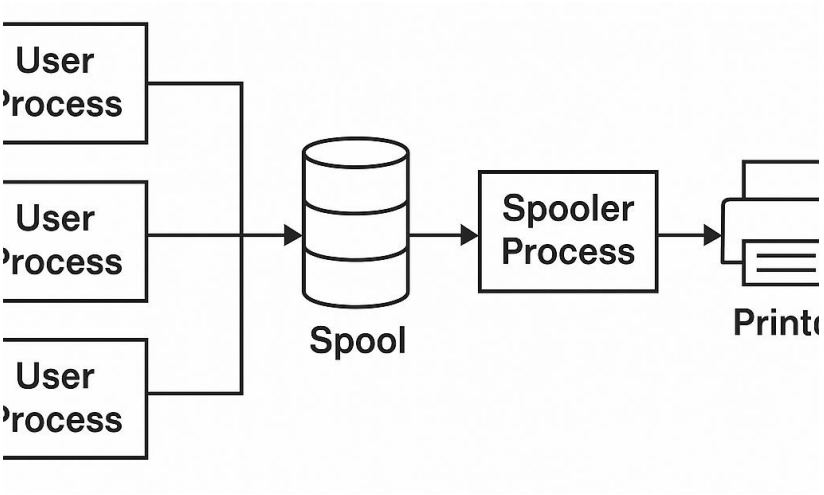
1. Deadlock Example (Cycle Present & No Resource Release)
2. Cycle Without Deadlock (Resource Eventually Released)



18. Given a scenario where multiple users need access to a shared printer, describe how an operating system could implement spooling to handle print requests efficiently. Illustrate your explanation with a diagram.

Spooling for Shared Printer Access

- **Concept:**
  - Jobs are placed in a queue (spool).
  - Printer processes jobs sequentially.
  - Allows multiple users to submit print requests concurrently.
- **Implementation:**
  - User Process: Sends print job to spool.
  - Spooler Process: Manages queue and sends jobs to printer.
  - Printer: Fetches and prints jobs one by one.



19. Apply the concept of IPC to demonstrate how shared memory enables multiple processes to access and communicate through a common memory area. Use an example to show how data is exchanged and coordinated between these processes.

Inter-Process Communication (IPC) via Shared Memory

- **Concept:**
  - Multiple processes read/write to a common memory area.
  - Faster than pipes or message queues.

Example (C - Shared Memory)

```
int shm_id = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
char *shm_ptr = shmat(shm_id, NULL, 0);
strcpy(shm_ptr, "Hello from Process A");
```

```
char *shm_ptr = shmat(shm_id, NULL, 0);
printf("%s\n", shm_ptr);
```

20. Give the monitor general structure and working principle along with an algorithmic approach case 1: Monitor-based wait and signal operations case 2: Monitor's implementation using semaphores.

Monitor: Structure & Working Principle

- Encapsulates shared data & synchronization.
- Only one process executes inside at a time.

Case 1: Monitor with Wait & Signal

```
monitor Example {
   condition x;
   void function() {
      wait(x);
      signal(x);
   }
}
```

Case 2: Monitor Using Semaphores

```
semaphore mutex = 1, next = 0;
int next_count = 0;

void wait(semaphore s) {
   s--;
   if (s < 0) {
      next_count++;
      signal(mutex);
      wait(next);
      next_count--;
   }
}

void signal(semaphore s) {
   s++;
   if (s <= 0) signal(next);
}
```

21. Demonstrate how semaphores can be used to synchronize multiple processes. Examine with pseudocode how a binary semaphore can be used for mutual exclusion, similar to a mutex.

Synchronization Using Semaphores

- Binary semaphore (0 or 1) acts like a mutex.
- Ensures mutual exclusion for shared resources.

Pseudocode Using Binary Semaphore

```
semaphore mutex = 1;

void process() {
   wait(mutex);
   signal(mutex);
}
```

22. Examine in detail the use of locks, lock data structures, and condition variables, and illustrate with an example how they are utilized to ensure proper synchronization in multi-threaded environments.

Locks, Lock Data Structures, and Condition Variables

1. Locks

- Used for mutual exclusion.
- Ensures only one thread accesses critical section.

2. Lock Data Structures

- Spinlocks (busy-waiting).
- Mutex locks (block until available).

3. Condition Variables

- Allows threads to wait until a condition is met.
- Requires a lock to modify shared data.

Example: Producer-Consumer Problem

```
pthread_mutex_t lock;
pthread_cond_t cond;
int buffer = 0;

void *producer() {
   pthread_mutex_lock(&lock);
   while (buffer != 0)
      pthread_cond_wait(&cond, &lock);
   buffer = 1;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&lock);
}

void *consumer() {
   pthread_mutex_lock(&lock);
   while (buffer == 0)
      pthread_cond_wait(&cond, &lock);
   buffer = 0;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&lock);
}
```