**Task 1**: Semaphore is one of the concurrency mechanisms available. Find out about more concurrency mechanisms. How do these mechanisms protect critical sections? Compare their implementations with wait() and signal() operations of semaphores.

**Concurrency mechanisms:**

Concurrency mechanisms are essential for managing access to shared resources in multi-threaded or multi-process environments.

* **Reader-Writer Locks:**

**How They Work:**

* Reader-writer locks allow concurrent read access to shared resources but exclusive write access.
* Multiple readers can access the critical section simultaneously, but writers require exclusive access.

**Implementation:**

* Provide lock\_read(), unlock\_read(), lock\_write(), and unlock\_write() operations.
* Writers get priority or fairness policies to avoid starvation.
* Protection of Critical Sections:
* Ensure mutual exclusion for writers and allow multiple readers, optimizing for read-heavy scenarios.

**Comparison to Semaphores:**

Semaphores can be used to implement reader-writer locks, but reader-writer locks provide a more specialized and efficient solution for scenarios with frequent reads and infrequent writes.

* **Mutexes (Mutual Exclusion Objects):**

**How They Work:**

* A mutex is a lock that ensures only one thread can access a critical section at a time.
* A thread must acquire the mutex before entering the critical section and release it afterward.

**Implementation**:

* Typically, mutexes provide lock() and unlock() operations.
* If a thread attempts to lock a mutex that is already locked by another thread, it will block (wait) until the mutex becomes available.

**Protection of Critical Sections:**

Mutexes prevent multiple threads from executing the critical section simultaneously, ensuring mutual exclusion.

**Comparison to Semaphores:**

* Similar to semaphores’ wait() and signal(), mutexes have lock() and unlock().
* Semaphores can count and allow multiple threads up to a specified limit (counting semaphores), whereas mutexes are binary (either locked or unlocked).

**Task 2**: Implement the algorithm of Producer-Consumer problem given above, in C language.

**Code:**

#include <stdio.h>

void wait(int \*x) {

if (\*x > 0) {

(\*x)--;}}

void signal(int \*x) {

(\*x)++;}

int main() {

int a[3];

int semaphore\_empty = 3;

int semaphore\_full = 0;

int semaphore\_s = 1;

int choice;

while(1) {

printf("empty = %d full = %d semaphore = %d\n", semaphore\_empty, semaphore\_full, semaphore\_s);

printf("Enter 1 to produce a product and 2 to consume a product: ");

scanf("%d", &choice);

if(choice == 1) {

if (semaphore\_empty != 0 && semaphore\_s == 1) {

printf("Enter product value: ");

scanf("%d", &a[semaphore\_full]);

wait(&semaphore\_empty);

wait(&semaphore\_s);

signal(&semaphore\_s);

signal(&semaphore\_full);

} else {

printf("Buffer is full.\n");

}

} else if(choice == 2) {

if (semaphore\_full > 0 && semaphore\_s == 1) {

wait(&semaphore\_full);

wait(&semaphore\_s);

printf("Consumed product value is %d\n", a[semaphore\_full]);

signal(&semaphore\_s);

signal(&semaphore\_empty);

} else {

printf("Buffer is empty.\n");

}

} else {

break;

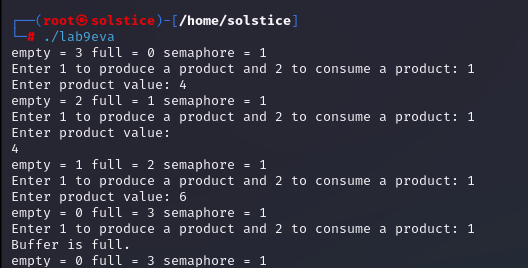
}

}

return 0;

}

**Producer Output:**



**Consumer Output**:

