Batch: C2 **Roll No.:** 16010122323

Experiment No. 07

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Implementation of Process synchronization algorithms using thread - producer consumer problem , reader-writers problem.

AIM: Implementation of Process synchronization algorithms using mutexes and semaphore – Dining Philosopher problem

---

Expected Outcome of Experiment:

CO 3. To understand the concepts of process synchronization and deadlock.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems", McGraw Hill Third Edition.**
3. **Sumitabha Das " UNIX Concepts & Applications", McGraw Hill Second Edition.**

Pre Lab/ Prior Concepts:

Knowledge of Concurrency, Synchronization, threads.

Description of the chosen process synchronization algorithm:

Readers:

- **Can operate concurrently, meaning multiple readers can access the resource at the same time as long as there are no active writers.**
- **When the first reader starts reading, it blocks writers until it finishes reading.**
- **The last reader to finish reading allows writers to proceed.**

Writers:

- **Require exclusive access to the resource. If a writer is active, no readers or other writers can access the resource.**
- **Writers must wait until all readers have finished before they can start writing.**

Implementation details:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h> // For sleep()
#include <cstdint>

sem_t mutex; // Mutex for reader count access
sem_t db;    // Semaphore for database access
int readercount = 0;

void *reader(void *p);
void *writer(void *p);

int main() {
    sem_init(&mutex, 0, 1);
    sem_init(&db, 0, 1);

    pthread_t readers[2], writers[2];

    for (int i = 0; i < 2; i++) {
        pthread_create(&readers[i], NULL, reader, (void *)(intptr_t)(i + 1));
        pthread_create(&writers[i], NULL, writer, (void *)(intptr_t)(i + 1));
    }

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

    sem_destroy(&mutex);
    sem_destroy(&db);

    return 0;
}

void *reader(void *p) {
    printf("Reader %ld is trying to read\n", (intptr_t)p);
    sem_wait(&mutex); // Acquire mutex for reader count
    readercount++;
    if (readercount == 1) {
        sem_wait(&db); // First reader locks the database
    }
    sem_post(&mutex); // Release mutex for reader count

    printf("Reader %ld is reading\n", (intptr_t)p);
    sleep(1); // Simulate reading

    sem_wait(&mutex); // Acquire mutex for reader count
    readercount--;
    if (readercount == 0) {
        sem_post(&db); // Last reader unlocks the database
    }
    sem_post(&mutex); // Release mutex for reader count

    printf("Reader %ld finished reading\n", (intptr_t)p);
    return NULL;
}

void *writer(void *p) {
    printf("Writer %ld is trying to write\n", (intptr_t)p);
    sem_wait(&db); // Wait for database access

    printf("Writer %ld is writing\n", (intptr_t)p);
    sleep(1); // Simulate writing

    sem_post(&db); // Release database access
    printf("Writer %ld finished writing\n", (intptr_t)p);
    return NULL;
}
```

Ln 10, Col 1 Spaces: 4 UTF-8 CRLF C Go Live Prettier

```
Reader 1 is trying to read
Reader 1 is reading
Writer 1 is trying to write
Reader 2 is trying to read
Reader 2 is reading
Writer 2 is trying to write
Reader 1 finished reading
Reader 2 finished reading
Writer 1 is writing
Writer 1 finished writing
Writer 2 is writing
Writer 2 finished writing


...Program finished with exit code 0
```

Conclusion:

Learned reader - writer problem.

Post Lab Descriptive Questions

1. Differentiate between a monitor, semaphore and a binary semaphore?

Monitor:

- A high-level synchronization construct that combines mutex locks and condition variables. It provides a way to safely manage shared data.
- Enforces mutual exclusion by allowing only one thread to execute in the monitor at a time, with built-in mechanisms for thread waiting and signaling.
- Used in languages like Java, where synchronized methods or blocks act as monitors.

Semaphore:

- A low-level synchronization primitive that uses a counter to control access to shared resources. It can be binary (0 or 1) or counting (greater than 1).
- Allows multiple threads to access a resource up to a defined limit. Threads can wait (block) or signal (release) the semaphore.
- Commonly used in concurrent programming to manage resource pools.

Binary Semaphore:

- A special case of a semaphore that can only take values 0 or 1, effectively functioning like a mutex.
- Provides mutual exclusion, ensuring that only one thread can access a resource at a time.
- Often used for signaling between threads or protecting critical sections.

2. Producer-Consumer Problem:

a. What would happen if the mutex semaphore was not used in the producer-consumer implementation?
b. How can the buffer size affect the performance of the producer-consumer system?
c. What are the potential issues if the producer and consumer threads are not properly synchronized?

A] If the Mutex Semaphore Was Not Used:

- Without the mutex semaphore, multiple producers or consumers could access and modify the shared buffer concurrently. This could lead to:
- Race Conditions: Threads might read or write to the buffer simultaneously, causing inconsistent or corrupted data.
- Data Corruption: Multiple threads could overwrite each other's changes, leading to loss of information or invalid states.
- Undefined Behavior: The program may behave unpredictably, potentially crashing or producing erroneous results.

B] Effect of Buffer Size on Performance:

- Small Buffer:
  - Increases the likelihood of blocking: If the buffer fills up quickly, producers must wait for consumers to make space, leading to potential bottlenecks.
  - May result in higher context switching and overhead due to frequent waiting and signaling.
- Large Buffer:
  - Reduces blocking occurrences: Producers can add more items without waiting, leading to better throughput.
  - However, it increases memory usage and may introduce latency, as consumers might take longer to process items if they have a larger pool to consume from.
- The optimal buffer size balances memory usage and performance based on the workload characteristics.

C] Potential Issues Without Proper Synchronization:

- Data Inconsistency: Without synchronization, producers and consumers may operate on stale or incomplete data.
- Buffer Overflows/Underflows: Producers may add items to a full buffer, or consumers may attempt to remove items from an empty buffer, leading to crashes or incorrect behaviors.
- Deadlocks: If synchronization is mishandled, threads could enter a state where they wait indefinitely for each other to release resources, halting the system.
- Starvation: Some threads may never get access to the buffer if producers or consumers are not properly managed, leading to inefficiencies in processing.

3. Reader-Writers Problem:
a. Explain the importance of the rw_mutex semaphore in the reader-writers problem.
b. How does the implementation ensure that writers get exclusive access to the shared resource?
c. What modifications would you make to prioritize writers over readers?

a] Importance of the rw_mutex Semaphore:

- The rw_mutex semaphore (or equivalent synchronization mechanism) is crucial for managing access to the shared resource. It helps in:
  - Mutual Exclusion: It ensures that when a writer is writing, no other readers or writers can access the resource simultaneously, preventing data corruption.
  - Coordination: It regulates access so that multiple readers can read concurrently, enhancing performance while ensuring exclusive access for writers.

b] Ensuring Exclusive Access for Writers:

  - Reader Count: A counter to track the number of active readers. When the first reader starts, it blocks writers by acquiring the rw_mutex. The last reader releases it.
  - Semaphore Control: When a writer wants to write, it checks if there are any active readers. If there are, the writer waits until all readers have finished.
  - Sequential Access: Writers are allowed to proceed only after all current readers have released their access, ensuring they have exclusive access to the resource.

c] Modifications to Prioritize Writers Over Readers:

- To prioritize writers, consider implementing one of the following strategies:
  - Writer Preference: Adjust the implementation so that when a writer is waiting, new readers are blocked from starting. This can be done by modifying the condition that allows readers to acquire the rw_mutex if a writer is waiting.
  - Queue Management: Use a queue to manage waiting readers and writers. When a writer arrives, new readers should wait until the writer has finished.
  - Use of Flags: Introduce a flag that indicates whether a writer is waiting. If this flag is set, new readers should be prevented from accessing the resource until the writer is done.

Date: 10/10/24 **Signature of faculty in-charge**