

| **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:**

Reader-writer problem

#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];

// Create reader and writer threads

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));

}

// Wait for all threads to complete

for (*int* i = 0; i < 2; i++) {

pthread\_join(readers[i], NULL);

pthread\_join(writers[i], NULL);

}

// Cleanup semaphores

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);

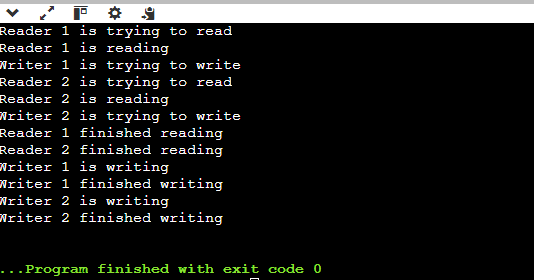
sleep(1); // Simulate writing

sem\_post(&db); // Release database access

printf("Writer %ld finished writing\n", (*intptr\_t*)p);

return NULL;

}



Producer-consumer problem

**Code**

import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;

class BoundedBuffer {

private final int capacity;

private final Queue<Integer> buffer;

private int mutex;

private int full;

private int empty;

private int x;

public BoundedBuffer(int capacity) {

this.capacity = capacity;

this.buffer = new LinkedList<>();

this.mutex = 1;

this.full = 0;

this.empty = capacity;

this.x = 0;

}

private void wait(int[] semaphore) {

while (semaphore[0] <= 0) {}

semaphore[0]--;

}

private void signal(int[] semaphore) {

semaphore[0]++;

}

public void produce() {

wait(new int[]{empty});

wait(new int[]{mutex});

x++;

buffer.add(x);

System.out.println("Produced: " + x);

signal(new int[]{mutex});

signal(new int[]{full});

}

public void consume() {

wait(new int[]{full});

wait(new int[]{mutex});

int item = buffer.poll();

System.out.println("Consumed: " + item);

signal(new int[]{mutex});

signal(new int[]{empty});

}

}

public class ProducerConsumerProblem {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the capacity of the bounded buffer: ");

int capacity = scanner.nextInt();

BoundedBuffer boundedBuffer = new BoundedBuffer(capacity);

while (true) {

System.out.println("\nMenu:");

System.out.println("1. Produce Item");

System.out.println("2. Consume Item");

System.out.println("3. Exit");

System.out.print("Choose an option: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

boundedBuffer.produce();

break;

case 2:

boundedBuffer.consume();

break;

case 3:

System.out.println("Exiting...");

scanner.close();

return;

default:

System.out.println("Invalid choice. Please try again.");

}

}

}

}

**Output:**

Enter the capacity of the bounded buffer: 45

Menu:

1. Produce Item

2. Consume Item

3. Exit

Choose an option: 1

Produced: 1

Menu:

1. Produce Item

2. Consume Item

3. Exit

Choose an option: 1

Produced: 2

Menu:

1. Produce Item

2. Consume Item

3. Exit

Choose an option: 1

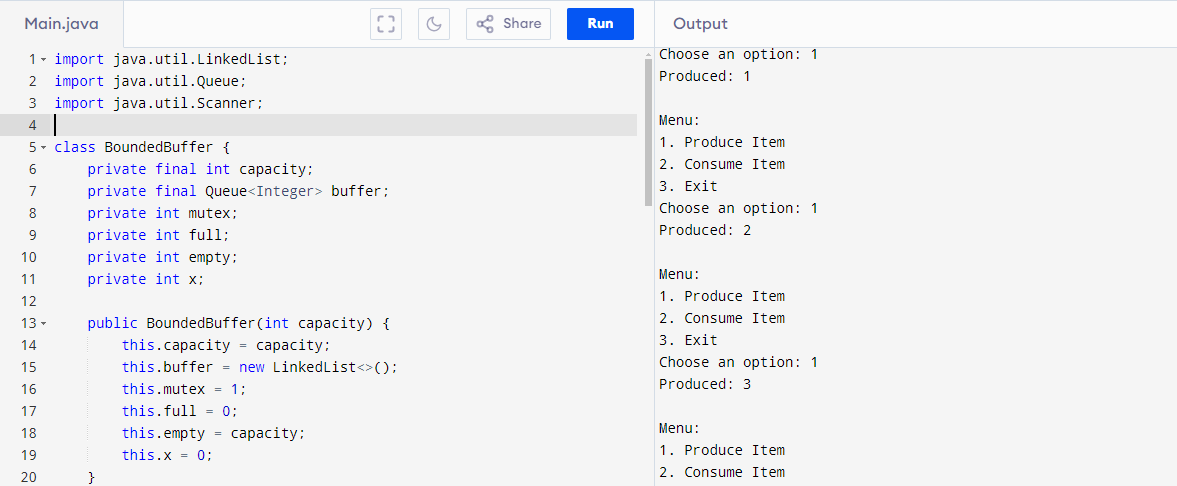
Produced: 3

Menu:

1. Produce Item

2. Consume Item

3. Exit



**Conclusion:**

Learned reader - writer and producer - consumer 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.

1. Producer-Consumer Problem:
   1. What would happen if the mutex semaphore was not used in the producer-consumer implementation?
   2. How can the buffer size affect the performance of the producer-consumer system?
   3. 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.

1. Reader-Writers Problem:
   1. Explain the importance of the rw\_mutex semaphore in the reader-writers problem.
   2. How does the implementation ensure that writers get exclusive access to the shared resource?
   3. 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**