Operating Systems Principles A1

Student #: S3783375

Email: [s3783375@student.rmit.edu.au](mailto:s3783375@student.rmit.edu.au)

Name: Manav Gadhoke

2.

## Issues and limitations of implementation

The following limitations/issues exist in my implementation

### Problem A – Producer-Consumer problem

I use “truly” random values, and so there is no inherent priority assigned to a producer or consumer. As a result, some may produce or consume less than others.

### Problem C – The dining Philosopher’s problem

At the beginning of the program, there is at least one instance of thinking before eating which leads to some time being wasted.

3.

The program solution implements 2 algorithms

1. Algorithm A: Producer – Consumer problem
2. Algorithm C: The dining Philosopher’s problem

4.

## Algorithm A: Producer – Consumer problem

This problem, first proposed by Dijkstra deals with multi-process synchronization. A common buffer is shared between, in this example, a producer and a consumer. A random number (in my implementation, a random number between 1 and 200) is used as the ‘item’ being produced and consumed. Production involves writing the item to the buffer, and consuming it refers to the item being removed from the buffer.

Problem that might arise and need to be addressed

1. Consumers attempting to empty buffer when it is empty
2. Producers trying to add to the buffer when it is full
3. Producers and consumers manipulating the buffer at the same time

Here’s how they’re addressed in my implementation

Text

Description automatically generated

A global is\_running flag dictates when the program is running, which in this case is 10 seconds.

The while loop is used to generate random numbers alongside the sem\_wait function, which helps prevent problem 2. The mutex\_lock and mutex\_unlock methods are used to lock the buffer which in turn help prevent problem 3.

Text

Description automatically generated

In the consumer function, we again use the is\_running flag to only run our program for 10 seconds.

In the loop, instead of producing new items, we’re now consuming items by removing them from the buffer. Use of semaphores let us do that while avoiding problem 1.

Text

Description automatically generated

Here’s the main method where the program starts by initializing the threads, semaphores and mutex.

Two loops create the threads for consumers and producers. The sleep function helps the program exit without errors after 10 seconds.

Two loops are then used for to join the producer and consumer threads, in order to terminate them.

Before returning 0 and exiting, the semaphores and mutex are destroyed to prevent memory leaks, which are quite common in multi-threaded programs such as this one.

## Algorithm C: Dining Philosopher’s problem

This problem requires us to imagine that there are 5 philosophers at a round-table waiting to eat. With 5 forks available, which in this implementation are represented by an array of mutex, with the additional constraint that each philosopher needs 2 forks to eat. At any given time, a philosopher is either thinking or eating.

Text

Description automatically generated

A custom struct has been used to keep a record of the index which makes managing the philosophers much easier.

A global `is\_running` flag is used to manage a global run-condition so the program can “gracefully” exit after 10 seconds of running.

The main method simply initializes the creation of threads and the mutex. The threads are then destroyed by joining the mutex inside two loops to prevent memory leaks, which are often found in programs involving multithreading.

Text

Description automatically generated

Here’s the philosopher function where the magic happens. In a nutshell, a philosopher is asleep until there is a fork available, which is kept track of by the `fork\_available` flag. If a fork becomes available, the loop Is exited, and the eating takes place for a random amount of time. This goes on ad-infinitum for each philosopher until the 10-second mark is reached.

The act of forks being put down is simulated by using mutex\_unlock for the other philosopher after eating. Overall, no race conditions or deadlocks are expected to occur.

The sched\_yield() method is used, which ensures each philosopher gets his share of food by scheduling the threads.

## Real life examples

**Producer-Consumer problem**

1. A printer with a queue accepting requests from multiple people illustrates this problem. Each person issuing a print command would be a producer, and each printer would be a consumer. Using the algorithm, the queue relay can be used between the people issuing requests and printers becoming available to ‘consume’ said requests.
2. A server with resources being accessed by different clients illustrates this problem really well. Imagine a scenario where multiple clients send requests to a server. The browser client in this case becomes the producer, and the server essentially consumes this request, as both of these happen at variable rates.

**The dining philosopher’s problem**

1. In computer operating systems where 2 programs try to access the same file at the same time, it can cause a deadlock. The solution can be used here, and each program will essentially be the philosopher and the files themselves will be the forks from the example.
2. Hypothetically, if there are 10 bank accounts, and 2 are trying to participate in a transaction, to ensure that the correct amount is credited and debited, all accounts will essentially have to ‘wait’ for the transaction to complete. In this analogy, the transactions become the philosophers, and the accounts become the forks.