OPERATING SYSTEM(OS) Assignment

SUBMITTED FROM:

Name: EMRAN AHMED
Reg NO: 2020831040
Session: 2020-21

SUBMITTED TO:

PARTHAPROTIM PAUL
Lecturer,
IICT, SUST

Date of Submission: 16-11-2023

1. How does IPC impact system performance and resource utilization, especially in scenarios with high-frequency communication between processes? (5)

Solution: Inter-Process Communication (IPC) refers to the mechanisms and techniques used by operating systems to allow different processes to communicate with each other. The impact of IPC on system performance and resource utilization can vary depending on several factors, including the chosen IPC method, the frequency of communication, and the nature of the data being exchanged.

Here are five key considerations:

1.Latency and Overhead:

IPC introduces additional overhead due to the need for data transfer, synchronization, and coordination between processes. This overhead can impact latency, especially in scenarios with high-frequency communication, as the time taken for IPC operations becomes a significant factor.

2.Context Switching:

In a multi-process environment, frequent communication between processes can lead to increased context switching. Context switching involves saving and restoring the state of a process, which consumes CPU resources. High-frequency context switching can negatively impact overall system performance.

3.Synchronization Overhead:

Many IPC mechanisms involve synchronization to ensure data consistency and prevent race conditions. Excessive synchronization can introduce overhead, particularly in scenarios where processes need to coordinate and wait for each other, impacting system performance.

4.Memory Utilization:

Shared memory IPC mechanisms, such as mmap or shared memory segments, can impact memory utilization. In scenarios with high-frequency communication, large amounts of shared data may need to be managed, potentially leading to increased demand for physical memory.

5.Networked IPC:

In scenarios where IPC involves network communication (e.g., inter-process communication over a network), factors such as network latency, bandwidth, and reliability come into play. High-frequency communication can lead to increased network traffic and potential bottlenecks.

To mitigate these impacts and optimize performance:
i)Choose the Right IPC Mechanism: Different IPC mechanisms (e.g., shared memory, message passing, pipes, sockets) have different characteristics. Choose the one that best suits the communication needs of your application.

ii)Optimize Data Transfer: Minimize the amount of data transferred between processes and consider using efficient serialization/deserialization techniques.

iii)Use Asynchronous Communication: In scenarios with high-frequency communication, consider using asynchronous communication mechanisms to reduce wait times and improve overall system responsiveness.

iv)Tune Parameters: Depending on the IPC method, there may be parameters that can be tuned to optimize performance. For example, adjusting buffer sizes or timeout values.

IPC is essential for communication between processes, especially in a multi-process or distributed system and its impact on performance and resource utilization should be carefully considered. Choosing the right IPC mechanism and optimizing communication patterns can help mitigate potential drawbacks and ensure efficient system operation.

3. How threads are useful in a single CPU? (5)

Solution: Threads in a single CPU system can be beneficial in several ways:

1.Parallelism and Concurrency:

Threads enable parallelism, allowing multiple tasks to execute concurrently on a single CPU. This is particularly useful in situations where there are independent or loosely coupled tasks that can be performed simultaneously. While a single-core CPU can only execute

one thread at a time, the operating system can switch between threads quickly, giving the illusion of parallel execution.

2.Responsive User Interface:

In applications with a graphical user interface (GUI), using threads can help maintain a responsive user interface. For example, the main thread can handle user input and respond to events, while separate threads perform background tasks or lengthy computations. This prevents the UI from freezing or becoming unresponsive during resource-intensive operations.

3.Multitasking:

Threads allow for multitasking within a single process. Different threads within the same process can execute different tasks concurrently, improving overall system efficiency. This is particularly advantageous in scenarios where various activities need to be performed simultaneously, such as handling network requests, processing data, and managing user input.

4.Resource Sharing and Efficiency:

Threads within the same process share the same address space, which means they can easily share data and resources. This can improve efficiency by avoiding the need for inter-process communication (IPC) mechanisms. However, care must be taken to synchronize access to shared resources to prevent data corruption.

5.Simplified Programming Model:

Using threads can simplify the programming model, especially for applications with multiple tasks that need to be performed concurrently. Threads share the same memory space, making it easier to communicate and share data between different parts of the program. This can lead to cleaner and more modular code compared to managing multiple independent processes.

It's important to note that while threads can provide advantages in terms of concurrency and responsiveness in a single CPU system, they also introduce challenges related to synchronization, race conditions, and shared resource management. Developers need to carefully design and implement multithreaded applications to ensure proper synchronization and avoid issues such as data corruption or deadlock. Additionally, the effectiveness of multithreading in a single CPU system may be limited by the specific characteristics of the workload and the nature of the tasks being performed.

4. Suppose I have the following codes for two threads and I am calling them from the main function. Is there any possibility of 'deadlock'? Explain your answer. If your answer is yes, then rewrite the code so that 'deadlock' can be avoided. (5)

```
Thread thread1 = new Thread(() => {
lockA.lock();
System.out.println("Acquired lockA");
try {
Thread.sleep(100);
} catch (InterruptedException e) {
e.printStackTrace();
```

```
}
lockB.lock();
System.out.println("Acquired lockB");
// Do something with lockA and lockB
lockB.unlock();
lockA.unlock();
});

Thread thread2 = new Thread(() -> {
lockB.lock();
System.out.println("Acquired lockB");
lockA.lock();
System.out.println("Acquired lockA");

// Do something with lockB and lockA
lockA.unlock();
lockB.unlock();
});
```

Solution: Yes, there is a possibility of deadlock in the provided code. A deadlock occurs when two or more threads wait indefinitely for each other to release locks, leading to a situation where no progress can be made. In this case, thread1 locks lockA and then lockB, while thread2 locks lockB and then lockA. If both threads reach a point where they have acquired one lock and are waiting for the other, a deadlock can occur.

To avoid deadlock, you need to ensure that the locks are acquired in a consistent order across all threads. One way to achieve this is to define a global order for acquiring locks and ensure that all threads follow this order. Here's a corrected version of your code:

```
Object lockA = new Object();
Object lockB = new Object();

Thread thread1 = new Thread(() -> {
    synchronized (lockA) {
        System.out.println("Acquired lockA");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (lockB) {
            System.out.println("Acquired lockB");
            // Do something with lockA and lockB
        }
    }
});

Thread thread2 = new Thread(() -> {
    synchronized (lockA) {
        System.out.println("Acquired lockA");
        synchronized (lockB) {
            System.out.println("Acquired lockB");
            // Do something with lockA and lockB
        }
    }
});

// Start the threads
thread1.start();
thread2.start();
```

```
// Wait for threads to finish
try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

In this corrected code, the order of acquiring locks (lockA and then lockB or vice versa) is consistent across both threads, preventing the possibility of a deadlock. The synchronized blocks ensure exclusive access to the critical sections of code, and the consistent lock acquisition order avoids circular waiting, which is a condition for deadlock.

2. A shop has the following policy. At any point in time (a) max 3 orders can be processed (b) if only 1 new order is being processed, then any number of old orders can be processed. Complete the code of the following for this. (Keep in mind - the variable names should be self-explanatory) (5)

```
    Lock (orders_mutex) {
        [INSERT CHECK HERE]
            wait(orders_mutex, new_cond)
        new_order++;
}
```

Solution: Certainly! To implement the described policy where a maximum of 3 orders can be processed at any point in time and if only 1 new order is being processed, then any number of old orders can be processed, you can use the following code snippet:

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Shop {
    private Lock ordersMutex = new ReentrantLock();
    private Condition newOrderCondition =
ordersMutex.newCondition();

    private int newOrder = 0;
    private int oldOrders = 0;

    public void processOrder() throws InterruptedException {
        ordersMutex.lock();
        try {
            // Check if max 3 orders are already being processed
            while (newOrder + oldOrders >= 3) {
                newOrderCondition.await();
            }

            // Check if only 1 new order is being processed
            if (newOrder == 1) {
                // Process any number of old orders
                oldOrders++;
            } else {
                // Process a new order
                newOrder++;
            }
        } finally {
            ordersMutex.unlock();
        }
```

```
    }

    public void orderCompleted() {
        ordersMutex.lock();
        try {
            if (newOrder > 0) {
                newOrder--;
            } else if (oldOrders > 0) {
                oldOrders--;
            }

            // Signal waiting threads that new order condition may have
changed
            newOrderCondition.signalAll();
        } finally {
            ordersMutex.unlock();
        }
    }

    // Other methods for the shop class
}
```

Explanation:
i)The ‘processOrder’ method uses the ordersMutex lock and checks two conditions before processing a new order:

a)If the total number of orders (new + old) is already 3 or more, it waits on the ‘newOrderCondition’ until the condition is satisfied.
b)If only 1 new order is being processed, it processes any number of old orders; otherwise, it processes a new order.

ii)The ‘orderCompleted’ method is called when an order is completed. It decrements the count of new or old orders accordingly and signals waiting threads that the ‘newOrderCondition’ condition may have changed.