\*\*producer consumer problem\*\*

Here's a Solution :

One solution of this problem is to use semaphores.

The semaphores which will be used here are:

• empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.

• full, a counting semaphore whose initial value is 0. At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

1. create the public class "producer\_consumer".

2. Initialize capacity to four. //shared data

The pseudocode of the producer function>>>

do

{

wait(); //wait if the buffer if full

/\* perform the insert operation in a slot \*/

// release lock

notify()

}

while(TRUE)

• Looking at the above code for a producer, we can see that a

producer first waits until there is at least one empty slot.

• Then it decrements the empty semaphore because, there will

now be one less empty slot, since the producer is going to insert

data in one of those slots.

• Then, it acquires lock on the buffer, so that the consumer cannot

access the buffer until producer completes its operation.

• After performing the insert operation, the lock is released and

the value of full is incremented because the producer has just

filled a slot in the buffer.

The pseudocode for the consumer function>>>>>>>>>>>

do

{

wait(); wait if the buffer is empty

/\* perform the remove operation in a slot \*/

notify();

}

while(TRUE);

• The consumer waits until there is atleast one full slot in the

buffer.

• Then it decrements the full semaphore because the number of

occupied slots will be decreased by one, after the consumer

completes its operation.

• After that, the consumer acquires lock on the buffer.

• Following that, the consumer completes the removal operation

so that the data from one of the full slots is removed.

• Then, the consumer releases the lock.

• Finally, the empty semaphore is incremented by 1, because the

consumer has just removed data from an occupied slot, thus

making it empty

Examples of deadlock and solutions

Deadlock: it’s group of processes or threads to make cycle inside the system,p1 wait p2 and p2 wait p1 ,p1 and p2 waiting infinite.

Solution: reorder the processes or threads until don’t make cycle.

1- class Account {

double balance;

void withdraw(double amount){

balance -= amount;

}

void deposit(double amount){

balance += amount;

}

void transfer(Account from, Account to, double amount){ sync(from);

sync(to);

from.withdraw(amount);

to.deposit(amount);

release(to);

release(from);

}

}

Solution: release (to); replace to release (from);

2- public class Deadlock {

public static void main(String[] args){

final Object resource1 = "resource1";

final Object resource2 = "resource2";

Thread t1 = new Thread() {

public void run() {

synchronized(resource1){

System.out.println("Thread 1: locked resource 1");

try{ Thread.sleep(50);

}

catch (InterruptedException e) {}

synchronized(resource2){

System.out.println("Thread 1: locked resource 2");

}

}

}

};

Thread t2 = new Thread(){

public void run(){

synchronized(resource2){

System.out.println("Thread 2: locked resource 2");

try{ Thread.sleep(50);

}

catch (InterruptedException e){}

synchronized(resource1){

System.out.println("Thread 2: locked resource 1");

}

}

}

};

t1.start();

t2.start();

}

}

Solution: synchronized (resource2) replace to synchronized (resources1)

3- public void method1() {

synchronized (String.class) {

System.out.println("Acquired lock on String.class object"); synchronized (Integer.class) {

System.out.println("Acquired lock on Integer.class object");

}

}

}

public void method2() {

synchronized (Integer.class) {

System.out.println("Acquired lock on Integer.class object"); synchronized (String.class) {

System.out.println("Acquired lock on String.class object");

}

}

}

Solution: Synchronized(Integer.class) replace to synchronized(String.class)

4- public class SimpleDeadLock {

public static Object l1 = new Object();

public static Object l2 = new Object();

private int index;

public static void main(String[] a) {

Thread t1 = new Thread1();

Thread t2 = new Thread2();

t1.start();

t2.start();

}

private static class Thread1 extends Thread { public void run() { synchronized (l1) {

System.out.println("Thread 1: Holding lock 1...");

try {

Thread.sleep(10);

}

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2..."); synchronized (l2) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

private static class Thread2 extends Thread { public void run() { synchronized (l2) {

System.out.println("Thread 2: Holding lock 2...");

try {

Thread.sleep(10);

}

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 1..."); synchronized (l1) {

System.out.println("Thread 2: Holding lock 2 & 1...");

}

}

}

}

}

Solution: synchronized (I2) replace to synchronized (l1)

5-Public class DeadLock {

Public static void main(String[] args) {

Object resource1 = new Object();

Object resource2 = new Object();

SharedObject s = new SharedObject(resource1, resource2); TestThread11 t1 = new TestThread11(s);

TestThread22 t2 = new TestThread22(s);

T1.start();

T2.start();

}

}

Class SharedObject {

Object o1, o2; SharedObject(Object o1, Object o2) {

This.o1 = o1; This.o2 = o2; }

Void m1() {

Synchronized(o1) {

System.out.println(“locked on o1 from m1()”); Synchronized(o2) {

System.out.println(“locked on o2 from m1()”);

}

}

}

Void m2() {

Synchronized(o2) {

System.out.println(“locked on o2 from m2()”); Synchronized(o1) {

System.out.println(“locked on o1 from m2()”);

}

}

}

}

Class TestThread11 extends Thread {

SharedObject s;

TestThread11(SharedObject s) {

This.s = s;

}

Public void run() {

s.m1();

}

}

Class TestThread22 extends Thread {

SharedObject s;

TestThread22(SharedObject s) {

This.s = s;

}

Public void run() {

s.m2();

**Solution: synchronized (o2) replace to synchronized (o1)**

\*\*Starvation\*\*

Starvation:

Starvation is a situation when a thread is in waiting state from long period because it not getting access of shared resources or because higher priority threads are coming

Example:-

JAVA

// Java program to illustrate Starvation concept

class StarvationDemo extends Thread {

static int threadcount = 1;

public void run()

{

System.out.println(threadcount + "st Child" +

" Thread execution starts");

System.out.println("Child thread execution completes");

threadcount++;

}

public static void main(String[] args)

throws InterruptedException

{

System.out.println("Main thread execution starts");

// Thread priorities are set in a way that thread5

// gets least priority.

StarvationDemo thread1 = new StarvationDemo();

thread1.setPriority(10);

StarvationDemo thread2 = new StarvationDemo();

thread2.setPriority(9);

StarvationDemo thread3 = new StarvationDemo();

thread3.setPriority(8);

StarvationDemo thread4 = new StarvationDemo();

thread4.setPriority(7);

StarvationDemo thread5 = new StarvationDemo();

thread5.setPriority(6);

thread1.start();

thread2.start();

thread3.start();

thread4.start();

// Here thread5 have to wait because of the

// other thread. But after waiting for some

// interval, thread5 will get the chance of

// execution. It is known as Starvation

thread5.start();

System.out.println("Main thread execution completes");

}

}

The output:-

Main thread execution starts

1st Child Thread execution starts

Child thread execution completes

2st Child Thread execution starts

Child thread execution completes

3st Child Thread execution starts

Child thread execution completes

4st Child Thread execution starts

Child thread execution completes

Main thread execution completes

5st Child Thread execution starts

Child thread execution completes

A white paper with writing on it

Description automatically generated with medium confidence

How to solve starvation:-

Solution to Starvation: Aging

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priority range from 127(low) to 0(high), we could increase the priority of a waiting process by 1 Every 15 minutes. Eventually, even a process with an initial priority of 127 would take no more than 32 hours for the priority 127 process to age to a priority-0 process.

Diagram

Description automatically generated with medium confidence

using semaphore:

Blocking queue is least susceptible, followed by blocking set and then busy wait semaphore.

Now, let’s examine the different semaphore types:

Busy Wait semaphore is the simplest semaphore, where blocked processes repeatedly check if semaphore condition allows them to grab lock.

int i = 0;

void m1() {

/\* wait for lock \*/

while (i != 0) {}

i = 1;

/\* execute m1 logic \*/

i = 0;

}

void m2() {

/\* wait for lock \*/

while (i != 0) {}

i = 2;

/\* execute m2 logic \*/

i = 0;

}

However, it is considered a bad design choice, as you can’t predict how long will the “do nothing” loop be executed, before checking again for the condition. In the code above it is not known how many times while (i != 0) loop executes, and that can eat a lot of CPU time, leading to unnecessary CPU cycles being burned. It should be considered if and only if locks are taken for a short amount of time.

Blocking Queue semaphore is a more advanced version of the semaphore concept, where available resources are stored in a queue. When a process wants to acquire a lock, it takes one resource from the queue. When a process returns the lock, the queue either takes the same resource back, or recycles it if it has to be returned to a blank state. If the queue does not have a resource, the process suspends, until a resource becomes available, instead of burning CPU cycles. It is important to note that the queue wakes processes in FIFO order of their suspension.

Blocking Set semaphore operates similarly to the blocking queue, but an important difference is that it wakes a random process, when a resource becomes available, without preserving the order of requests for the resource.

Conclusion:

blocking queue semaphore is least susceptible to starvation, as it not only avoids senseless use of CPU cycles, but also serves requests in FIFO order. Then comes blocking set, as it does not burn CPU time, but serves processes in random order, making it more prone to starvation. Finally, busy wait semaphores should not be used, as they not only are unable to guarantee the order of requests, but also burn CPU cycles as all of the requesters check if the resource is available.

\*\*Explanation for real world application\*\*

 Suppose you are owner of a Bakery shop named **“Bakes and Bytes”**; and you can make 4 cakes at max because you don’t have a big oven yet. Now, you have several customers to buy it, but let us consider only one customer. Your bakery is advanced and has great management system.

\*\*How did apply the problem\*\*

The producer consumer problem is a synchronization problem. There is a **fixed size buffer** (here, bakery store) and the **producer** (the baker) produces **items**(cakes) and enters them into the buffer. The **consumer**(customer) removes the items from the buffer and **consumes**(buys) them.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer. So the buffer should only be accessed by the producer or consumer at a time.

To solve this, there comes the concept of Semaphores.

A **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable. This variable is used to solve **“Critical Section”** problems and to achieve process synchronization in the multi processing environment.