***Producer consumer problem***

What is producer consumer problem ?

The producer-consumer problem is an example of a [multi-process synchronization](https://www.geeksforgeeks.org/introduction-of-process-synchronization/) problem. The problem describes two processes, the producer and the consumer that shares a common buffer use it as a [queue](https://www.geeksforgeeks.org/queue-data-structure/).

* The producer’s job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Problem: Given the common buffer, the task is to make sure that the producer can’t add data into the buffer when it is full and the consumer can’t remove data from an empty buffer.

Solution: The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same manner, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer ,the solution must be free from deadlock starvation,we use semaphore to solve the problem (mutex,full,empty) .

***Pseudocode:***

*Producer:*

while (true) {

...

/\* produce an item in next\_produced \*/

...

wait(empty);

wait(mutex);

...

/\* add next produced to the buffer \*/

...

signal(mutex);

signal(full);

}

*Consumer :*

while (true) {

wait(full);

wait(mutex);

...

/\* remove an item from buffer to next\_consumed \*/

...

signal(mutex);

signal(empty);

...

/\* consume the item in next consumed \*/

...

}

***Deadlock***

*What is deadlock?*

• A deadlock occurs when two or more threads wait forever for a lock

or resource held by another of the threads.

• An application may stall or fail as the deadlocked threads cannot

progress.

Solution :

1. Deadlock prevention: The possibility of deadlock is excluded before making requests, by eliminating one of the necessary conditions for deadlock.

Example: Only allowing traffic from one direction, will exclude the possibility of blocking the road.

1. Deadlock avoidance: Operating system runs an algorithm on requests to check for a safe state. Any request that may result in a deadlock is not granted.

Example: Checking each car and not allowing any car that can block the road. If there is already traffic on road, then a car coming from the opposite direction can cause blockage.

1. Deadlock detection & recovery: OS detects deadlock by regularly checking the system state, and recovers to a safe state using recovery techniques.

Example: Unblocking the road by backing cars from one side. Deadlock prevention and deadlock avoidance are carried out before deadlock occurs.

Examples :

EX1:

Let Sand Q be two semaphores initialized to 1

P0 P1

wait (S ); wait (Q);

wait (Q); wait(S);

……………. ……………

signal (S); signal (Q);

signal (Q); signal(S);

Consider if P0. executes wait(S) and P1, wait(Q). When P0. executes

wait(Q), it must wait until P1, executes signal(Q)

However, P1, is waiting until P0. execute signal(S).

Since these signal() operations will never be executed,P0 and P1 are deadlocked .

EX2:

public void operation1() {

first\_mutex.lock();

System.out.println("first\_mutex acquired by Thread:"+Thread.currentThread().getName()+“

, waiting to acquire second\_mutex.");

second\_mutex.lock();

System.out.println("second\_mutex acquired");

try {

System.out.println("executing first operation.");

Thread.sleep(50);

} catch (InterruptedException e) {

throw new RuntimeException(e);

}

second\_mutex.unlock();

first\_mutex.unlock();

}

EX3:

Problem :

The possibility of deadlock in the producer consumer problem depends on what kind of implementation you are looking at and how you have used it.

Let us assume that we have one producer and one consumer with a single shared buffer between them.

Using sleep() and wakeup() implementation:

The deadlock might happend if there is no consistency between the producer and consumer leading them to access the critical section simultaneously. This will lead to a scenario where both the consumer and the producer goes to sleep and hence leading to deadlock.

# *Solution* :

Using semaphores:

While using semaphores the concept of ownership of mutex and the order of increment and decrement of the semaphores should be kept in mind . Any change in order may lead to deadlock.

*Starvation :*

What is starvation ?

Is a resource management problem where a process does not get the resources it needs for a long time because the resources are being allocated to other processes. Or, a condition where a process does not get the resources it needs for a long time because the resources are being allocated to other processes. It occurs generally in a Priority-based scheduling System where High Priority requests get processed first thus a request with the least priority may never be processed .

*solution :*

1. Aging It is a technique to avoid starvation in a scheduling system. It works by adding an aging factor to the priority of each request. The aging factor must increase the request priority as time passes and must ensure that a request will eventually be the highest priority request (after it has waited long enough)
2. A condition which is used to reduce starvation of low priority tasks. It is a process which gradually increases the priority of task depending on waiting time. It ensure that jobs in the lower level queues will eventually complete their execution.

*Examples:*

*EX 1:*

If a process P is having a priority number as 85 at 0ms . Then after every 10ms ( you can use any time here) , we can decrease the priority number of the process P by 1(here also instead of 1 you can take any other number). So, after 10ms, the priority of the process P will be 84. Again after 10ms, we will decrease the priority number of process P by 1. So, after 10ms, the priority of the process P will become 83 and this process will continue. After a certain period of time, the process P will become a high priority process when the priority number comes closer to 0 and the process P will get the CPU for its execution. In this way, the lower priority process also gets the CPU.

*EX 2:*

*Problem:*

* Consumer on certain condition is not returning the previously consumed buffer to empty buffer queue and continuing to wait for next ready buffer ready to be consumed.
* Or Producer on certain condition is not returning produced buffer to ready buffer queue and continuing to wait for empty buffer to produce.
* Then eventually this kind of situation will lead to starvation.

This kind of "waiting while holding buffer" scenario can lead to starvation.

* Consumer on certain condition is not returning the previously consumed buffer to empty buffer queue and continuing to wait for next ready buffer ready to be consumed.
* Or Producer on certain condition is not returning produced buffer to ready buffer queue and continuing to wait for empty buffer to produce.
* Then eventually this kind of situation will lead to starvation.

This kind of "waiting while holding buffer" scenario can lead to starvation.

# *Solution* :

we used Queue and linked list

# *Real world application* :

*Fast Food :*

Have a producer and a consumer (tills and worker) So the tills take the order(Place them in a buffer, so a circular array..) and the worker processes that order(Takes orders out of the circular array).

We have :

* number of workers = 2
* number of tills = 3
* number of choices= 4

We have 5 classes:

Class worker

Class Buffer

Class order

Class tills

Class FastFood