Solution pseudocode:

Producer

```
while (true) {
 /* produce an item in next produced */
while (counter == BUFFER_SIZE) ;
/* do nothing */
 buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
 }
```

consumer

```
while (true) {
while (counter == 0) ;
/* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
/* consume the item in next consumed */
 }
```

Deadlock

A resource allocation graph tracks which resource is held by which process and which process is waiting for a resource of a particular type. If a process is using a resource, a directed link is formed from the resource node to the process node. If a process is requesting a resource, a directed link is formed from the process node to the resource node. If there is a cycle in the Resource Allocation Graph then the processes will deadlock. In the following Resource Allocation Graph R stands for a resource and P stands for a process. Here one of the deadlock cycles is

There are 3 processes $P1$, $P2$, $P3$

• There are 4 resources:

$R1$ (1 instance) , $R2$ (2 instances) , $R3$ (1 instance) , $R4$ (3 instances) . $R4$ (3 instances)

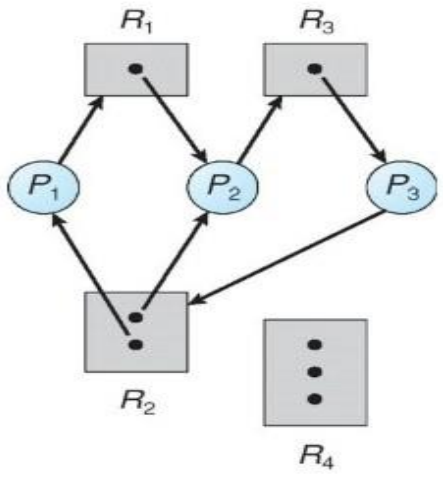
$P1$ holding 1 instance from $R2$

$P1$ request 1 instance from $R1$

$P2$ holding 1 instance from $R2$

$P2$ holding 1 instance from $R1$

$P2$ request 1 instance from $R3$



P3 holding 1 instance from R3

P3 request 1 instance from R2

And this is a deadlock:

Because P1 is waiting for P2 and,

P2 is waiting for P3 and,

P3 is waiting for P1 and P2

The solution by deadlock prevention:

>>hold and wait>> :

A process requests a resource only if it does not hold any other resources.

A process requests and is allocated all its resources before it begins execution

>>No Preemption>>

If a process holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.

Preempted resources are added to the list of resources for which the process is waiting.

Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

>>Circular Wait >>

Impose a total ordering on all resource types, and Require that each process requests resources in an increasing order of enumeration.

Other solution of deadlock:

A deadlock in a bound buffer multithreaded process can occur when two or more threads are waiting for a resource that is held by another thread, and those threads are unable to proceed because they are waiting for the resource to be released. This can lead to a situation where the threads are stuck in a loop, waiting indefinitely for the resource to become available.

One solution to this problem is to use a mutex (mutual exclusion) to control access to the shared resource. A mutex is a synchronization object that allows only one thread to access the resource at a time. When a thread acquires the mutex, it can access the resource, and when it is finished, it releases the mutex so that other threads can access the resource.

Another solution is to use a semaphore, which is a synchronization object that allows multiple threads to access a resource, but only up to a certain number at a time. When a thread tries to acquire the semaphore, it will block if the semaphore count is zero. When a thread releases the semaphore, the semaphore count is incremented, allowing another thread to acquire the semaphore.

Alternatively, you can use a monitor, which is a synchronization object that allows only one thread to execute a specific block of code at a time. When a thread tries to enter a monitor, it will block if another thread is already executing the code protected by the monitor. When the thread finishes executing the code, it releases the monitor, allowing another thread to enter.

It is also possible to prevent deadlocks by using a deadlock detection and recovery mechanism. This can involve setting a timeout on resource acquisition attempts, or using a global lock hierarchy to break the deadlock.

Starvation:

A system with multiple producers and consumers, the producer-consumer problem, also known as the bounded-buffer problem, can arise if the producers are producing items faster than the consumers are able to consume them. This can lead to a situation where the buffer or queue holding the items becomes full, causing the producers to block and wait until there is space available in the buffer. At the same time, if the consumers are not able to keep up with the rate at which items are being produced, the buffer may become empty, causing the consumers to block and wait until more items are available. This can lead to a cycle of blocking and waiting, resulting in a situation known as starvation.

One example of starvation of a bound buffer in Java might occur when a producer thread is trying to add items to the buffer, but the buffer is already full. In this case, the producer thread will be unable to add any more items to the buffer, and will be "starved" of the ability to do so.

One way to solve the producer-consumer problem is to use a semaphore to control access to the shared buffer. A semaphore is a synchronization object that controls access to a common resource by multiple processes in a parallel programming environment. The producer can acquire a semaphore before adding an item to the buffer, and the consumer can acquire a semaphore before removing an item from the buffer. This ensures that only one producer or consumer has access to the buffer at any given time, preventing race conditions and ensuring that the buffer is not simultaneously accessed by multiple producers or consumers.

Another solution is to use a mutex (mutual exclusion) to synchronize access to the shared buffer. A mutex is a synchronization object that allows a process to lock access to a resource, preventing other processes from accessing the resource until the process has finished using it. The producer can acquire a mutex before adding an item to the buffer, and the consumer can acquire a mutex before removing an item from the buffer. This ensures that only one producer or consumer has access to the buffer at any given time, preventing race conditions and ensuring that the buffer is not simultaneously accessed by multiple producers or consumers.

It is also possible to use a condition variable to synchronize access to the shared buffer. A condition variable is a synchronization object that allows a process to block and wait for a certain condition to be met before continuing execution. The producer can wait on a condition variable if the buffer is full, and the consumer can wait on a condition variable if the buffer is empty. This allows the producer and consumer to block and wait until the buffer has sufficient space or items, respectively, before continuing execution.

In summary, there are several ways to solve the producer-consumer problem in a system with multiple producers and consumers, including the use of semaphores, mutexes, and condition variables. These synchronization objects allow processes to coordinate their access to shared resources and prevent race conditions, ensuring that the shared resource is used efficiently and effectively.

Explanation for real world application:

The bound buffer idea for real world is the producer is the waiter will produce for example meals to customer and the consumer will consume this meals until buffer be empty.

The problem that the producer will produce foods with out stop,the consumer not to able to finish the food and the leftover food will go to waste as soon as if the producer will just produce limited amount of food so, the supply of food will eventually run out and if it happening the consumer will be in starvation case.

The solution of this problem:

that that the producer will produce food until the buffer full ,and the consumer will eat until the buffer is empty,,,,,,,,,, and the consumer will be wait for the buffer to be full again ,,

when the buffer empty again the producer will produce food again so, the process will be repeat .