2.Explain deadlock prevention and avoidance.

Deadlock prevention and avoidance are two methods of dealing with deadlocks, which are situations where a set of processes or threads are blocked because each one is holding a resource that another one needs. [Deadlock prevention and avoidance differ in how they prevent deadlocks from occurring1](https://www.geeksforgeeks.org/deadlock-prevention/)[2](https://www.geeksforgeeks.org/difference-between-deadlock-prevention-and-deadlock-avoidance/)[3](https://pediaa.com/what-is-the-difference-between-deadlock-prevention-and-deadlock-avoidance/).

## Deadlock Prevention

Deadlock prevention is a method of ensuring that at least one of the four necessary conditions for deadlock to occur will never hold. The four necessary conditions for deadlock are:

* Mutual exclusion: A resource can be held by only one process or thread at a time.
* Hold and wait: A process or thread that is holding some resources can request additional resources and wait for them.
* No preemption: A resource cannot be taken away from a process or thread that is holding it until it releases it voluntarily.
* Circular wait: There exists a circular chain of processes or threads, each one waiting for a resource held by the next one.

Deadlock prevention eliminates any of these conditions by imposing some constraints on how processes or threads can request and hold resources. For example:

* Eliminate mutual exclusion: Make the resources sharable among multiple processes or threads. However, this is not possible for some resources, such as printers or files, that are inherently non-sharable.
* Eliminate hold and wait: Require a process or thread to request all the resources it needs at once before starting its execution, or to release all the resources it holds before requesting any new ones. However, this may lead to low resource utilization and starvation.
* Eliminate no preemption: Allow a process or thread to be preempted and lose some or all of the resources it holds if another process or thread with higher priority needs them. However, this may incur high overhead and complexity due to context switching and rollback.
* Eliminate circular wait: Impose a total ordering on all the resources and require a process or thread to request them in increasing order of the ordering. However, this may not be feasible for some resources that do not have a natural ordering.

## Deadlock Avoidance

Deadlock avoidance is a method of ensuring that the system will never enter an unsafe state, which is a state where there is no possibility of granting all the current requests of all the processes or threads without causing deadlock. Deadlock avoidance requires some knowledge or prediction of the future requests and releases of each process or thread.

Deadlock avoidance uses an algorithm that dynamically examines the resource allocation state before granting any request. The algorithm determines whether granting the request will leave the system in a safe state or an unsafe state. If the request will leave the system in a safe state, it is granted; otherwise, it is denied or delayed until a safe state can be reached.

One of the common algorithms for deadlock avoidance is the banker’s algorithm, which simulates a bank that grants loans to customers and checks whether they can repay them. The banker’s algorithm requires each process or thread to declare in advance the maximum number of each type of resource it may need. The algorithm then keeps track of the available, allocated, and maximum resources for each process or thread, and grants a request only if it satisfies the safety criterion.

## Comparison

The main difference between deadlock prevention and deadlock avoidance is that deadlock prevention ensures that at least one of the necessary conditions for deadlock can never occur, while deadlock avoidance ensures that the system does not enter an unsafe state. Some other differences are:

* Deadlock prevention does not require any information about the future requests and releases of each process or thread, while deadlock avoidance does.
* Deadlock prevention imposes some constraints on how processes or threads can request and hold resources, while deadlock avoidance does not.
* Deadlock prevention may result in low resource utilization and reduced system throughput, while deadlock avoidance may result in higher resource utilization and increased system throughput.
* Deadlock prevention may be simpler and faster to implement than deadlock avoidance, which may require complex algorithms and calculations.

3.With the help of diagram, explain the bounded buffer problem.

The bounded buffer problem, also known as the producer-consumer problem, is a classic problem of synchronization. It involves two processes, a producer and a consumer, that share a common buffer of a fixed size. The producer generates data and puts it into the buffer, while the consumer consumes data from the buffer. [The problem is to coordinate the activities of the producer and the consumer so that they do not interfere with each other1](https://www.studytonight.com/operating-system/bounded-buffer)[2](https://www.baeldung.com/cs/bounded-buffer-problem)[3](https://www.i2tutorials.com/os-introduction/os-bounded-buffer-problem/).

**Bounded Buffer Problem Definition**

The bounded buffer problem can be defined as follows:

* There is a buffer of n slots, each capable of storing one unit of data.
* The buffer is initially empty and can be accessed by both the producer and the consumer.
* The producer generates data and tries to insert it into an empty slot of the buffer. If the buffer is full, the producer has to wait until there is an empty slot available.
* The consumer consumes data and tries to remove it from a filled slot of the buffer. If the buffer is empty, the consumer has to wait until there is a filled slot available.

**Bounded Buffer Problem Diagram**

The bounded buffer problem can be illustrated by a diagram that shows the state of the buffer and the actions of the producer and the consumer. For example:

The diagram shows that the buffer has 8 slots, numbered from 0 to 7. The producer and the consumer use two pointers, in and out, to indicate the next slot to insert or remove data. The producer generates data items A, B, C, D, E, F, G, H and puts them into the buffer one by one. The consumer consumes data items A, B, C from the buffer one by one. The diagram also shows how the in and out pointers are updated after each insertion or removal operation.

**Bounded Buffer Problem Solutions**

The bounded buffer problem requires synchronization mechanisms to ensure that:

* The producer and the consumer do not access or modify the same slot of the buffer at the same time. This requires mutual exclusion between the producer and the consumer.
* The producer does not overwrite existing data in the buffer before it is consumed by the consumer. This requires synchronization between the producer and the buffer.
* The consumer does not consume data that has already been consumed or that has not been produced yet. This requires synchronization between the consumer and the buffer.

Some of the common solutions for the bounded buffer problem are:

* Using semaphores: Semaphores are integer variables that can be manipulated atomically by two operations: wait and signal. They can be used to represent available resources or signal events. To solve the bounded buffer problem using semaphores, we can use three semaphores: empty, full, and mutex. The empty semaphore represents the number of empty slots in the buffer; initially, it is set to n. The full semaphore represents the number of filled slots in the buffer; initially, it is set to 0. The mutex semaphore is a binary semaphore that is used as a lock to ensure mutual exclusion between the producer and the consumer; initially, it is set to 1. The pseudocode for using semaphores is as follows:

// Code for producer

do {

// produce data

...

// wait until there is an empty slot

wait(empty);

// acquire lock on buffer

wait(mutex);

// insert data into buffer

...

// release lock on buffer

signal(mutex);

// signal that there is a filled slot

signal(full);

} while (true);

// Code for consumer

do {

// wait until there is a filled slot

wait(full);

// acquire lock on buffer

wait(mutex);

// remove data from buffer

...

// release lock on buffer

signal(mutex);

// signal that there is an empty slot

signal(empty);

// consume data

...

} while (true);

Copy

* Using monitors: Monitors are high-level synchronization constructs that encapsulate shared data and operations on them within an abstract data type. They ensure mutual exclusion by allowing only one process or thread to access a monitor at a time. They also provide condition variables that can be used to suspend and resume processes or threads based on some conditions. To solve the bounded buffer problem using monitors, we can define a monitor that contains an array of n slots as its shared data and two condition variables: notFull and notEmpty. The monitor also provides two procedures: insert and remove, that are used by the producer and the consumer to access the buffer. The pseudocode for using monitors is as follows:

// Monitor definition

monitor Buffer {

// shared data

int buffer[n];

int in, out;

int count;

// condition variables

condition notFull, notEmpty;

// monitor procedures

procedure insert(data) {

// if buffer is full, wait until not full

if (count == n) {

notFull.wait();

}

// insert data into buffer

buffer[in] = data;

in = (in + 1) % n;

count++;

// signal that buffer is not empty

notEmpty.signal();

}

procedure remove(data) {

// if buffer is empty, wait until not empty

if (count == 0) {

notEmpty.wait();

}

// remove data from buffer

data = buffer[out];

out = (out + 1) % n;

count--;

// signal that buffer is not full

notFull.signal();

}

}

// Code for producer

do {

// produce data

...

// insert data into buffer

Buffer.insert(data);

} while (true);

// Code for consumer

do {

// remove data from buffer

Buffer.remove(data);

// consume data

...

} while (true);

Copy

* Using circular buffers: Circular buffers are data structures that use a fixed-size array as a queue. They use two pointers, head and tail, to indicate the first and last elements of the queue. They also use a counter to keep track of the number of elements in the queue. To solve the bounded buffer problem using circular buffers, we can use a circular buffer as the shared buffer between the producer and the consumer. The producer and the consumer can use atomic operations to update the head, tail, and counter variables without locking. The pseudocode for using circular buffers is as follows:

// Circular buffer definition

struct CircularBuffer {

// shared data

int buffer[n];

int head, tail;

int count;

}

// Code for producer

do {

// produce data

...

// wait until there is an empty slot

while (count == n) {

// busy wait

}

// insert data into buffer atomically

atomic {

buffer[tail] = data;

tail = (tail + 1) % n;

count++;

}

} while (true);

// Code for consumer

do {

// wait until there is a filled slot

while (count == 0) {

// busy wait

}

// remove data from buffer atomically

atomic {

data = buffer[head];

head = (head + 1) % n;

count--;

}

// consume data

...

} while (true);

4.What is monitor? Justify with an example.

A monitor is a high-level synchronization construct that provides mutual exclusion, condition variables, and data encapsulation in a single module. [A monitor is implemented as a programming language construct, typically in object-oriented languages, and simplifies the process synchronization by providing a high-level abstraction for data access and synchronization1](https://www.geeksforgeeks.org/monitors-in-process-synchronization/)[2](https://en.wikipedia.org/wiki/Monitor_%28synchronization%29)[3](https://www.tutorialandexample.com/monitors-in-operating-system).

**Monitor Definition**

A monitor can be defined as follows:

* A monitor is a module that encapsulates a shared resource and provides access to that resource through a set of procedures.
* The procedures provided by a monitor ensure that only one process or thread can access the shared resource at any given time, and that processes or threads waiting for the resource are suspended until it becomes available.
* A monitor also provides condition variables that can be used to suspend and resume processes or threads based on some conditions.

**Monitor Example**

One example of using a monitor to solve a synchronization problem is the bounded buffer problem, which involves a producer that generates data and a consumer that processes the data. The data is stored in a shared buffer with a limited capacity. The buffer is responsible for handling the synchronization and communication between the producer and the consumer processes.

To solve the bounded buffer problem using a monitor, we can define a monitor that contains an array of n slots as its shared data and two condition variables: notFull and notEmpty. The monitor also provides two procedures: insert and remove, that are used by the producer and the consumer to access the buffer. The pseudocode for using the monitor is as follows:

// Monitor definition

monitor Buffer {

// shared data

int buffer[n];

int in, out;

int count;

// condition variables

condition notFull, notEmpty;

// monitor procedures

procedure insert(data) {

// if buffer is full, wait until not full

if (count == n) {

notFull.wait();

}

// insert data into buffer

buffer[in] = data;

in = (in + 1) % n;

count++;

// signal that buffer is not empty

notEmpty.signal();

}

procedure remove(data) {

// if buffer is empty, wait until not empty

if (count == 0) {

notEmpty.wait();

}

// remove data from buffer

data = buffer[out];

out = (out + 1) % n;

count--;

// signal that buffer is not full

notFull.signal();

}

}

// Code for producer

do {

// produce data

...

// insert data into buffer

Buffer.insert(data);

} while (true);

// Code for consumer

do {

// remove data from buffer

Buffer.remove(data);

// consume data

...

} while (true);

The monitor works as follows:

* The producer generates data and calls the insert procedure of the monitor to put it into the buffer. If the buffer is full, the producer waits on the notFull condition variable until there is an empty slot available. After inserting the data, the producer signals on the notEmpty condition variable to notify the consumer that there is data available.
* The consumer calls the remove procedure of the monitor to get data from the buffer. If the buffer is empty, the consumer waits on the notEmpty condition variable until there is a filled slot available. After removing the data, the consumer signals on the notFull condition variable to notify the producer that there is space available.
* The monitor ensures that only one process can access the buffer at any given time, and that processes waiting for the buffer are suspended until it becomes available.