An **Operating System** (**OS**) is an interface between a computer user and computer hardware. An **operating system** is a software that performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. An Operating System(O.S.) is a System software that manages the hardware resources and provides services to the Application software.

## Network Operating System

Network Operating System is a computer operating system that facilitates to connection and communication of various autonomous computers over a network. An Autonomous computer is an independent computer that has its own local memory, hardware, and O.S. It is self capable to perform operations and processing for a single user. They can either run the same or different O.S.

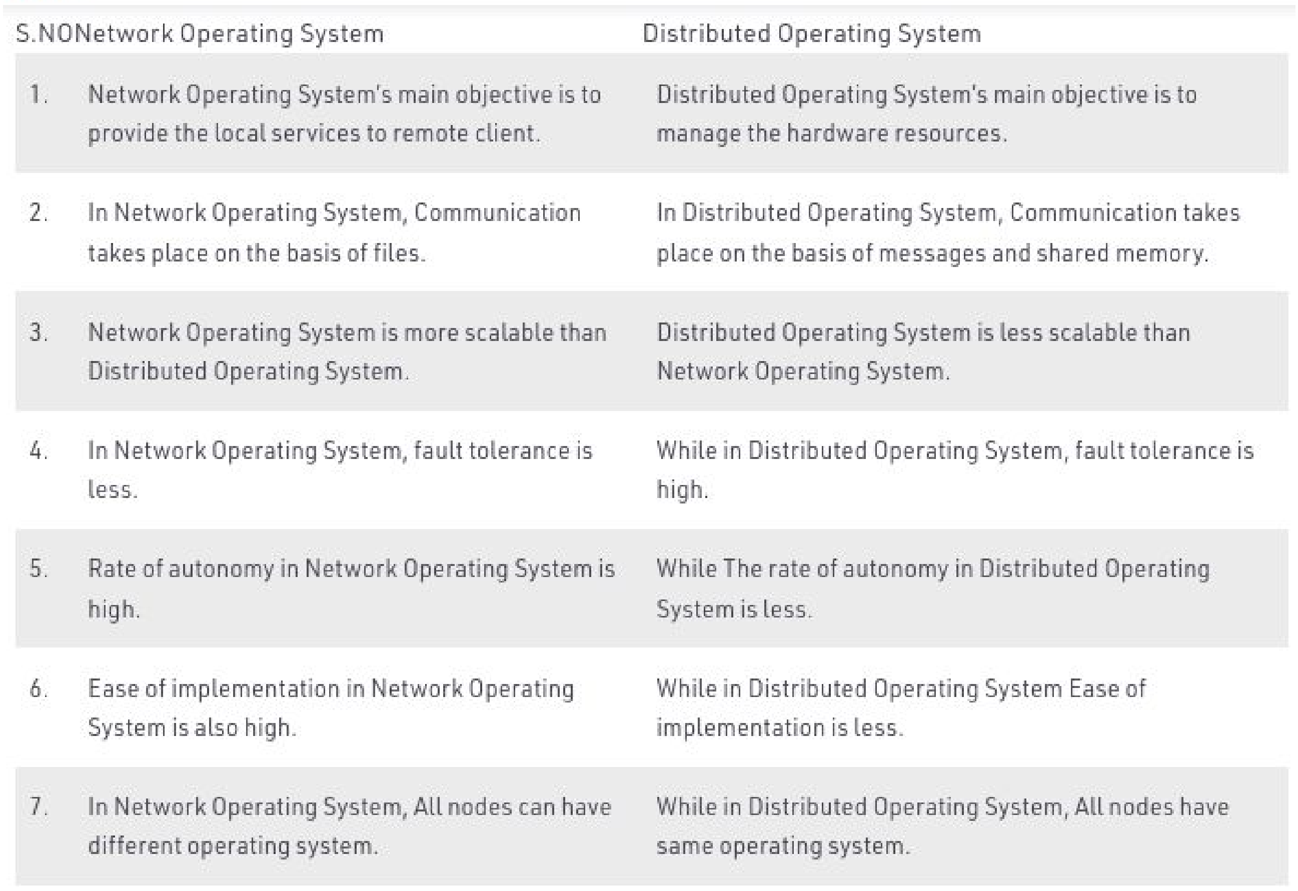
### *Following are the common functionalities of the Network Operating System:*

1. Data and Resource sharing
2. Performance
3. Security
4. Robustness
5. Scalability
6. Memory management

## DISTRIBUTED OPERATING SYSTEM

A DOS is a system that contains multiple components located on different machines, which coordinate and communicate actions in order to appear as a single coherent working system to the user. Distributed Operating System is a model where distributed applications are running on multiple computers linked by communications. A distributed operating system is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network.

This system looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).



| S.NO | Network Operating System | Distributed Operating System |
|---|---|---|
| 1. | Network Operating System's main objective is to provide the local services to remote client. | Distributed Operating System's main objective is to manage the hardware resources. |
| 2. | In Network Operating System, Communication takes place on the basis of files. | In Distributed Operating System, Communication takes place on the basis of messages and shared memory. |
| 3. | Network Operating System is more scalable than Distributed Operating System. | Distributed Operating System is less scalable than Network Operating System. |
| 4. | In Network Operating System, fault tolerance is less. | While in Distributed Operating System, fault tolerance is high. |
| 5. | Rate of autonomy in Network Operating System is high. | While The rate of autonomy in Distributed Operating System is less. |
| 6. | Ease of implementation in Network Operating System is also high. | While in Distributed Operating System Ease of implementation is less. |
| 7. | In Network Operating System, All nodes can have different operating system. | While in Distributed Operating System, All nodes have same operating system. |

## CONTEXT SWITCHING

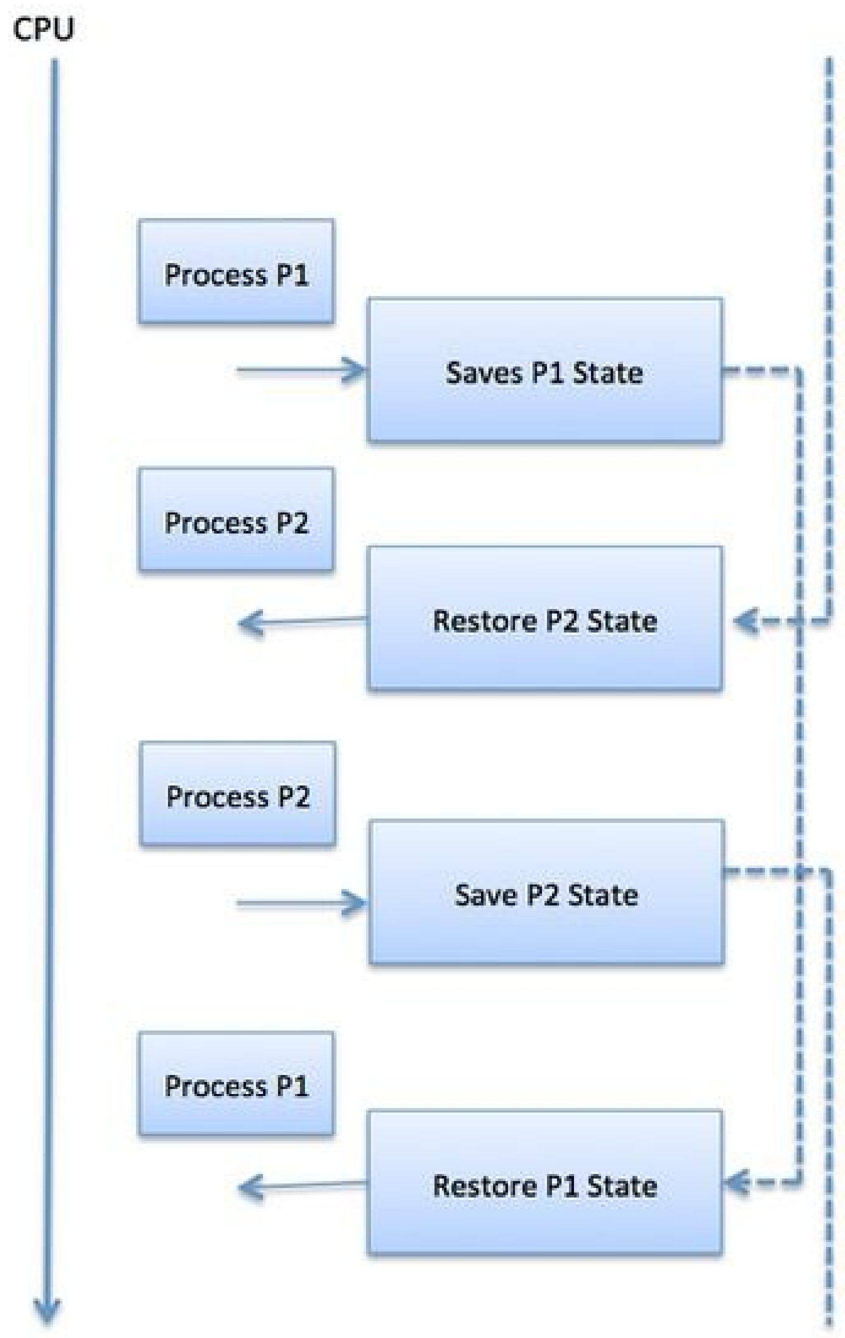
Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

### Context Switching Steps

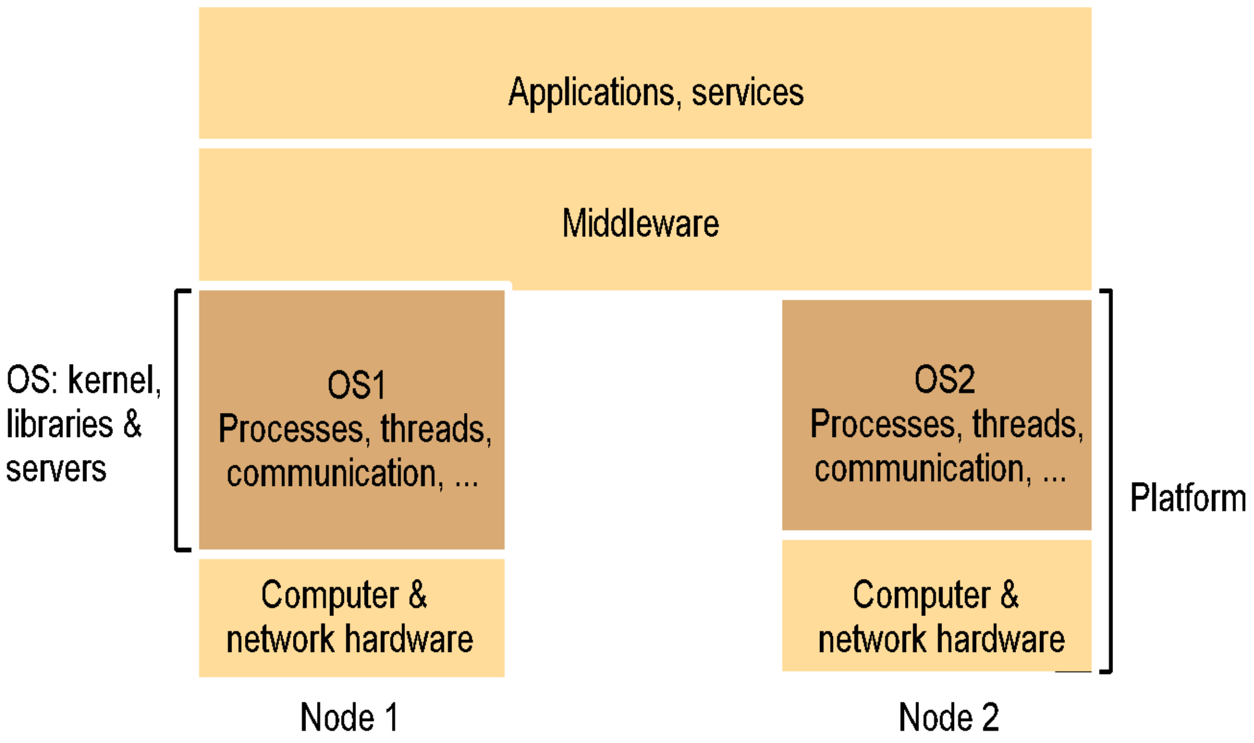
The steps involved in context switching are as follows −

- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue, etc.
- Select a new process for execution.

- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the process control block and registers.



## The Operating System Layer


Applications, services
Middleware
OS: kernel, libraries & servers
OS1 Processes, threads, communication, ...
OS2 Processes, threads, communication, ...
Platform
Computer & network hardware
Computer & network hardware
Node 1
Node 2

- Figure 1 shows how the operating system layer at each of the two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.
- Kernels and server processes are the components that manage resources and present clients with an interface to the resources.

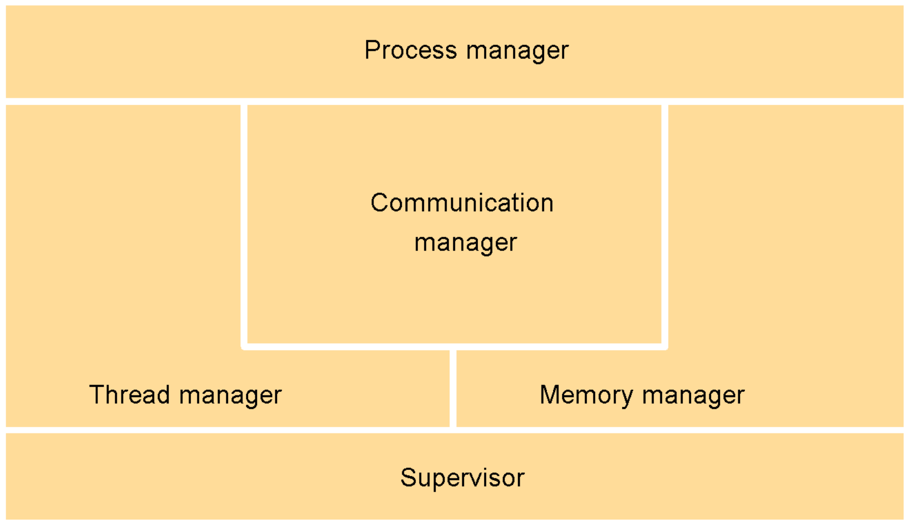


Fig: Core Functionality of OS

- **Process Manager**: handles the creation of and operations upon processes.
- **Thread manager**: Thread Creation, Synchronization, and Scheduling. Threads are schedulable activities attached to processes
- **Communication Manager** manages communication between threads attached to different processes on the same computer. Communication between threads in remote processes.
- **Memory Manager**: management of physical and virtual memory
- **Supervisor**: Dispatching of interrupts, system call traps, and other exceptions; control of memory management unit and hardware caches. Processor and floating-point unit register manipulations.

## PROTECTION

**Protection** refers to a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system.

**The need for Protection:**

- To prevent access to unauthorized users and
- To ensure that each active programs or processes in the system use resources only as of the stated policy,
- To improve reliability by detecting latent errors.

## PROCESS

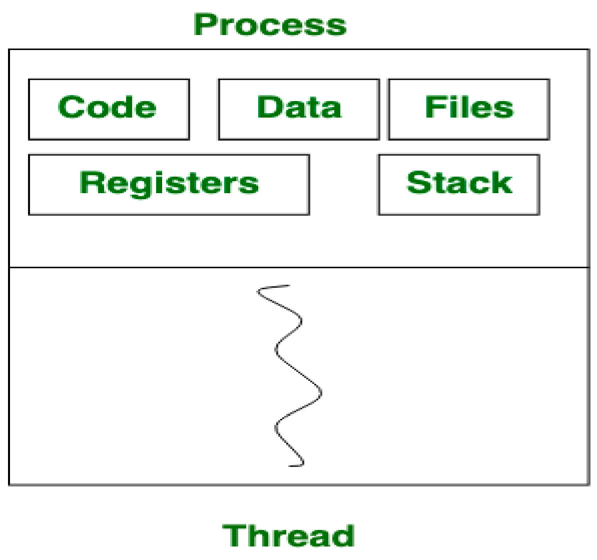
Process means any program is in execution. Process control block controls the operation of any process. Process control block contains information about processes for example Process priority, process id, process state, CPU, register, etc. A process can create other processes which are known as **Child Processes**. The process takes more time to terminate and it is isolated means it does not share a memory with any other process.

The process can have the following states like new, ready, running, waiting, terminated, suspended.

## THREAD

Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. A thread has 3 states: running, ready, and blocked.

The thread takes less time to terminate as compared to process and like process threads do not isolate.



| S.NO | Process | Thread |
|---|---|---|
| 1. | Process means any program is in execution. | Thread means segment of a process. |
| 2. | Process takes more time to terminate. | Thread takes less time to terminate. |
| 3. | It takes more time for creation. | It takes less time for creation. |
| 4. | It also takes more time for context switching. | It takes less time for context switching. |
| 5. | Process is less efficient in term of communication. | Thread is more efficient in term of communication. |
| 6. | Process consume more resources. | Thread consume less resources. |
| 7. | Process is isolated. | Threads share memory. |
| 8. | Process is called heavy weight process. | Thread is called light weight process. |

## Properties of Process

Here are the important properties of the process:

- The creation of each process requires separate system calls for each process.
- It is an isolated execution entity and does not share data and information.
- Processes use the IPC(Inter-Process Communication) mechanism for communication that significantly increases the number of system calls.
- Process management takes more system calls.
- A process has its stack, heap memory with memory, and data map.

## Properties of Thread

Here are important properties of Thread:

- Single system call can create more than one thread
- Threads share data and information.
- Threads shares instruction, global, and heap regions. However, it has its register and stack.
- Thread management consumes very few, or no system calls because of communication between threads that can be achieved using shared memory.

# Operating System Architecture

The kernel is the core of an operating system. It is the software responsible for running programs and providing secure access to the machine's hardware. Since there are many programs, and resources are limited, the kernel also decides when and how long a program should run.

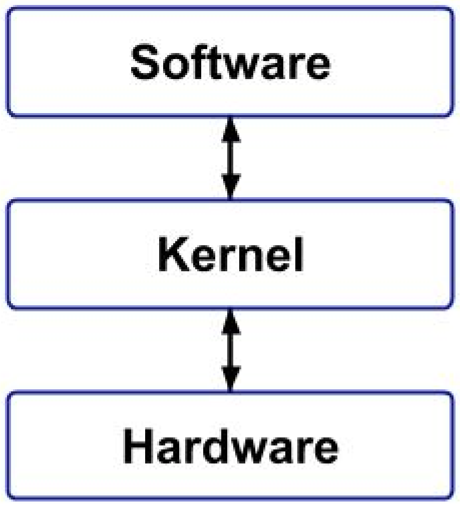
**Functions of a Kernel**

- **Access Computer resource:** A Kernel can access various computer resources like the CPU, I/O devices, and other resources. It acts as a bridge between the user and the resources of the system.
- **Resource Management:** It is the duty of a Kernel to share the resources between various processes in such a way that there is uniform access to the resources by every process.
- **Memory Management:** Every process needs some memory space. So, memory must be allocated and deallocated for its execution. All this memory management is done by a Kernel.
- **Device Management:** The peripheral devices connected to the system are used by the processes. So, the allocation of these devices is managed by the Kernel.

**Types of Kernel**

**1. Monolithic Kernels**

Monolithic Kernels are those Kernels where the user services and the kernel services are implemented in the same memory space i.e. different memory for user services and kernel services are not used in this case. By doing so, the size of the Kernel is increased and this, in turn, increases the size of the Operating System. As there is no separate User Space and Kernel Space, so the execution of the process will be faster in Monolithic Kernels.



Advantages:

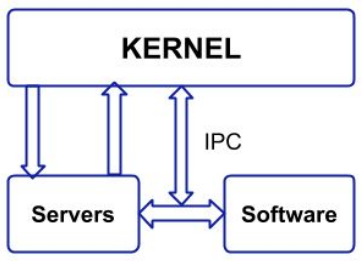
- It provides CPU scheduling, memory scheduling, file management through System calls only.
- Execution of the process is fast because there is no separate memory space for the user and kernel.

Disadvantages:

- If any service fails, then it leads to system failure.
- If new services are to be added then the entire Operating System needs to be modified.

**2. Microkernel**

A Microkernel is different from a Monolithic kernel because, in a Microkernel, the user services and kernel services are implemented into different spaces i.e. we use User Space and Kernel Space in the case of Microkernels. As we are using User Space and Kernel Space separately, so it reduces the size of the Kernel and this, in turn, reduces the size of the Operating System.



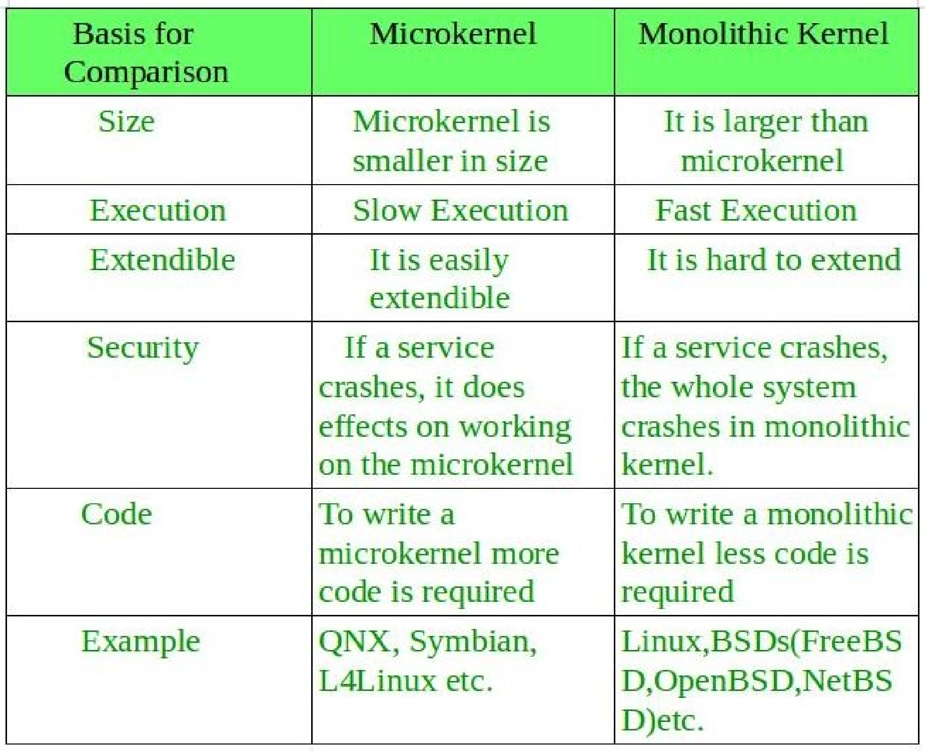
As we are using different spaces for user services and kernel service, so the communication between application and services is done with the help of message parsing and this, in turn, reduces the speed of execution.

Advantages:

- If new services are to be added then it can be easily added.

Disadvantages:

- Since we are using User Space and Kernel Space separately, so the communication between these can reduce the overall execution time.



| Basis for Comparison | Microkernel | Monolithic Kernel |
|---|---|---|
| Size | Microkernel is smaller in size | It is larger than microkernel |
| Execution | Slow Execution | Fast Execution |
| Extendible | It is easily extendible | It is hard to extend |
| Security | If a service crashes, it does effects on working on the microkernel | If a service crashes, the whole system crashes in monolithic kernel. |
| Code | To write a microkernel more code is required | To write a monolithic kernel less code is required |
| Example | QNX, Symbian, L4Linux etc. | Linux,BSDs(FreeBSD,OpenBSD,NetBSD)etc. |

## Deadlock In Distributed System

A deadlock is a condition in a system where a set of processes (or threads) have requests for resources that can never be satisfied. Essentially, a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, Coffman defined four conditions that have to be met for a deadlock to occur in a system:

- Mutual exclusion A resource can be held by at most one process.
- Hold and wait Processes that already hold resources can wait for another resource.
- Non-preemption A resource, once granted, cannot be taken away.
- Circular wait Two or more processes are waiting for resources held by one of the other processes.

The same conditions for the deadlock in uniprocessors apply to distributed systems. Unfortunately, as in many other aspects of distributed systems, they are harder to detect, avoid, and prevent. Four strategies can be used to handle deadlock:

- ignorance: ignore the problem; assume that a deadlock will never occur. This is a surprisingly common approach.
- detection: let a deadlock occur, detect it, and then deal with it by aborting and later restarting a process that causes deadlock.
- prevention: make a deadlock impossible by granting requests so that one of the necessary conditions for deadlock does not hold.
- avoidance: choose resource allocation carefully so that deadlock will not occur. Resource requests can be honored as long as the system remains in a safe (non-deadlock) state after resources are allocated.

An alternative to detecting deadlocks is to design a system so that deadlock is impossible. We examined the four conditions for deadlock. If we can deny at least one of these conditions then we will not have a deadlock.

**Mutual exclusion**

To deny this means that we will allow a resource to be held (used) by more than one process at a time. If a resource can be shared then there is no need for mutual exclusion and deadlock cannot occur. Too often, however, a process requires mutual exclusion for a resource because the resource is some object that will be modified by the process.

**Hold and wait**

Denying this means that processes that hold resources cannot wait for another resource. This typically implies that a process should grab all of its resources at once. This is not practical either since we cannot always predict what resources a process will need throughout its execution.

**Non-preemption**
A resource, once granted, cannot be taken away. In transactional systems, allowing preemption means that a transaction can come in and modify data (the resource) that is being used by another transaction. This differs from mutual exclusion since the access is not concurrent but the same problem arises of having multiple transactions modify the same resource. We can support this with optimistic concurrency control algorithms that will check for out-of-order modifications at commit time and rollback (abort) if there are potential inconsistencies.

**Circular wait**
Avoiding circular wait means that we ensure that a cycle of waiting on resources does not occur. We can do this by enforcing an ordering on granting resources and aborting transactions or denying requests if an ordering cannot be granted.

There are three approaches to detect deadlocks in distributed systems. They are as follows:

**Centralized approach –**
In the centralized approach, there is only one responsible resource to detect deadlock. The advantage of this approach is that it is simple and easy to implement, while the drawbacks include excessive workload at one node, single-point failure (that is the whole system is dependent on one node if that node fails the whole system crashes) which in turns makes the system less reliable.

**Distributed approach –**
In the distributed approach different nodes work together to detect deadlocks. No single point failure ( that is the whole system is dependent on one node if that node fails the whole system crashes) as the workload is equally divided among all nodes. The speed of deadlock detection also increases.

**Hierarchical approach –**
This approach is the most advantageous. It is the combination of both centralized and distributed approaches of deadlock detection in a distributed system. In this approach, some selected nodes or clusters of nodes are responsible for deadlock detection and these selected nodes are controlled by a single node.

# Checkpoint

The basic idea behind checkpoint-recover is the saving and restoration of the system state. By saving the current state of the system periodically or before critical code sections, it provides the baseline information needed for the restoration of lost state in the event of a system failure.

The basic mechanism of checkpoint-recovery consists of three key ideas - the saving and restoration of the executive state, and the detection of the need to restore the system state.

# Distributed Commit

The distributed commit problem involves having an operation being performed by each member of a process group, or none at all. Assuming that no failures occur, the protocol consists of the following two phases, each consisting of two steps

1. The coordinator sends a vote-request message to all participants.
2. When a participant receives a vote-request message, it returns either a vote-commit message to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction or otherwise a vote-abort message.
3. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a global-commit message to all participants. However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a global-abort message.
4. Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a global-commit message, it locally commits the transaction. Otherwise, when receiving a global-abort message, the transaction is locally aborted as well.

The different distributed commit protocols are −

- One-phase commit
- Two-phase commit
- Three-phase commit

## Distributed One-phase Commit

Distributed one-phase commit is the simplest commit protocol. Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are −

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- The slaves wait for “Commit” or “Abort” message from the controlling site. This waiting time is called **window of vulnerability**.
- When the controlling site receives “DONE” message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

## Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

**Phase 1: Prepare Phase**

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
- A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

**Phase 2: Commit/Abort Phase**

- After the controlling site has received “Ready” message from all the slaves −
    - The controlling site sends a “Global Commit” message to the slaves.
    - The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
    - When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first “Not Ready” message from any slave −
    - The controlling site sends a “Global Abort” message to the slaves.
    - The slaves abort the transaction and send a “Abort ACK” message to the controlling site.
    - When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

## Distributed Three-phase Commit

The steps in distributed three-phase commit are as follows −

**Phase 1: Prepare Phase**

The steps are same as in distributed two-phase commit.

**Phase 2: Prepare to Commit Phase**

- The controlling site issues an “Enter Prepared State” broadcast message.
- The slave sites vote “OK” in response.

**Phase 3: Commit / Abort Phase**

The steps are same as two-phase commit except that “Commit ACK”/”Abort ACK” message is not required.

## DOS as Middleware

A distributed software support layer that abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, operating systems, and implementation languages. The primary role of middleware is to ease the task of developing, deploying, and managing distributed applications by providing a simple, consistent, and integrated distributed programming environment.
Middleware in the context of distributed applications is software that provides services beyond those provided by the operating system to enable the various components of a distributed system to communicate and manage data. Middleware supports and simplifies complex distributed applications. It includes web servers, application servers, messaging, and similar tools that support application development and delivery. Middleware is especially integral to modern information technology based on XML, SOAP, Web services, and service-oriented architecture.

Middleware is the infrastructure that facilitates the creation of business applications, and provides core services like concurrency, transactions, threading, messaging, and the SCA framework for service-oriented architecture applications. It also provides security and enables high availability functionality.

In distributed systems, it hides the distributed nature of the application. It keeps a collection of interconnected parts that are operational and running in distributed locations, out of view making things easier and simpler to manage.

There are a specific set of roles that middleware carries out such as it confines the distributed complexion of a system from the users. In other words, the middleware is able to keep the system active in multiple (distributed) locations but doesn’t often reveal details that it is a part of a complex system. These details would be programming codes. The middleware has the role to handle heterogeneous environments of a distributed system. The term heterogeneity refers to how users can access and use distributed applications through a range of devices and networks. But middleware does not manage each and every individual node. This heterogeneity applies to network, hardware, software, and programming languages. Through the use of middleware, we can facilitate heterogeneity.

Middleware is a software layer situated between applications and operating systems. Middleware is typically used in distributed systems where it simplifies software development by doing the following:

- Hides the intricacies of distributed applications

- Hides the heterogeneity of hardware, operating systems, and protocols
- Provides uniform and high-level interfaces used to make interoperable, reusable, and portable applications
- Provides a set of common services that minimize duplication of efforts and enhances collaboration between applications

## WHICH KERNEL IS BEST FOR DOS ?

1. Microkernel interfaces being rigidly defined, proving the correctness of both services and the kernel are easier. Since the complexity of RCAs go up massively with distributed architectures, this is certainly a benefit. At least some part of the system is provably correct.
2. The services for a distributed system are almost inevitably more complex. Having the segregation of microkernel helps.
3. The chief counter argument to micro kernels is the possible latency in the messaging. With distributed, a messaging is involved every so often to communicate with the non local systems anyway. Thus, on top of the benefit of faster comm that microkernel provides, there is the fact that the overhead is lower.