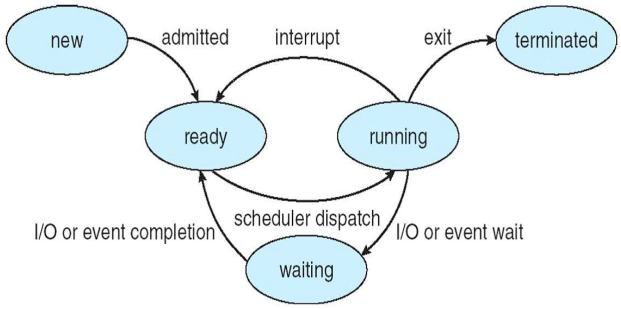
Module II
OPERATING SYSTEM

DEFINITION OF PROCESS

A program in Execution is called process. A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. A program is a passive entity while a process is an active entity.



PROCESS STATES

As a process executes, it changes state

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution

Only one process is running and all other processes are in ready or waiting.

PROCESS CONTROL BLOCK

PCB is a data structure in the operating system kernel containing the information needed to manage a particular process. The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

The PCB contains important information about the specific process including

- **Process State**: The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **Process Number**: Unique identification of the process in order to track "which is which" information.
- **Program Counter**: This register stores the next instruction to be executed.

- **CPU Registers**: MAR, MBR, PC, IR, GPR
- **Memory limits**: process stored location in memory
- **List of files**:
- **The priority of process**
- **A pointer to parent process.**
- **A pointer to child process** (if it exists).

OPERATIONS ON PROCESS

1. **Process creation**: A user requests and already running process can creates new processes. Parent process creates children processes using a system call, which, in turn create other processes, forming a tree of processes.
2. **Process preempting**: A process preempted if I/O event or timeout occurs. Then process moves from running state to ready state and CPU loads another process from ready state to running state, if available.
3. **Process blocking**: When a process needs I/O event during its execution, then process moves from running state to waiting state and dispatches another process to CPU.
4. **Process termination**: A process terminated if when a process completes its execution. Also, these events: OS, Hardware interrupt, and Software interrupt can cause termination of a process.

Process Creation

Process creation is a task of creating new processes. There are different situations in which a new process is created. There are different ways to create new process. A new process can be created at the time of initialization of operating system or when system calls such as fork () are initiated by other processes. The process, which creates a new process using system calls, is called parent process while the new process that is created is called child process. The child processes can create new processes using system calls. A new process can also create by an operating system based on the request received from the user.

The process creation is very common in running computer system because corresponding to every task that is performed there is a process associated with it. For instance, a new process is created every time a user logs on to a computer system, an application program such a MS Word is initiated, or when a document printed.

Process Preemption

Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemption.

Process Blocking

A blocking process is usually waiting for an event such as a semaphore being released or a message arriving in its message queue. In multitasking systems, such processes are expected to notify the scheduler with a system call that it is to wait, so that they can be removed from the active scheduling queue until the event occurs. A process that continues to run while waiting (i.e., continuously polling for the event in a tight loop) is said to be busy-waiting, which is undesirable as it wastes clock cycles which could be used for other processes.

Process Termination

Process termination is an operation in which a process is terminated after the execution of its last instruction. This operation is used to terminate or end any process. When a process is terminated, the resources that were being utilized by the process are released by the operating system. When a child process terminates, it sends the status information back to the parent process before terminating. The child process can also be terminated by the parent process if the task performed by the child process is no longer needed. In addition, when a parent process terminates, it has to terminate the child process as well became a child process cannot run when its parent process has been terminated.

The termination of a process when all its instruction has been executed successfully is called normal termination. However, there are instances when a process terminates due to some error. This termination is called as abnormal termination of a process.

PROCESS COMMUNICATION

Inter-process communication (IPC) is a set of programming interfaces that allows a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's

behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.

A mechanism through which data is shared among the process in the system is referred to as Inter-process communication. Multiple processes communicate with each other to share data and resources. A set of functions is required for the communication of process with each other. In multiprogramming systems, some common storage is used where process can share data. The shared storage may be the main memory or it may be a shared file. Files are the most commonly used mechanism for data sharing between processes. One process can write in the file while another process cam read the data for the same file.

Various techniques can be used to implement the Inter-Process Communication. There are two fundamental models of Inter-Process communication that are commonly used, these are:

1. **Shared Memory Model**
2. **Message Passing Model**

Shared Memory Model

In shared memory model. The co operating process shares a region of memory for sharing of information. Some operating systems use the supervisor call to create a share memory space. Similarly, Some operating system use file system to create RAM disk, which is a virtual disk created in the RAM. The shared files are stored in RAM disk to share the information between processes. The shared files in RAM disk are actually stored in the memory. The Process can share information by writing and reading data to the shared memory location or RAM disk.

Message Passing Model

In this model, data is shared between process by passing and receiving messages between co-operating process. Message passing mechanism is easier to implement than shared memory but it is useful for exchanging smaller amount of data.

In message passing mechanism data is exchange between processes through kernel of operating system using system calls. Message passing mechanism is particularly useful in a distributed

environment where the communicating processes may reside on different components connected by the network. For example, A data program used on the internet could be designed so that chat participants communicate with each other by exchanging messages. It must be noted that passing message technique is slower than shared memory technique.

A message contains the following information:

- Header of message that identifies the sending and receiving processes
- Block of data
- Pointer to block of data
- Some control information about the process

Typically Inter-Process Communication is based on the ports associated with process. A port represents a queue of processes. Ports are controlled and managed by the kernel. The processes communicate with each other through kernel.

In message passing mechanism, two operations are performed. Theses are sending message and receiving message. The function send() and receive() are used to implement these operations. Supposed P1 and P2 want to communicate with each other. A communication link must be created between them to send and receive messages. The communication link can be created using different ways. The most important methods are:

- Direct model
- Indirect model
- Buffering

COMMUNICATION IN CLIENT SERVER SYSTEM

Client-server concept underpins distributed systems over a couple of decades. There are two counterparts in the concept: a client and a server. In practice there are often multiple clients and single server. Clients start communication by sending requests to the server, the server handles them and usually returns responses back. Client processes often do not live long, while server process, which is sometimes called daemon, live till OS shutdown.

Sockets

IPC with sockets is very common in distributed systems. In nutshell, a socket is a pair of an IP address and a port number. For two processes to communicate, each of them needs a socket.

When server daemon is running on a host, it is listening to its port and handles all requests sent by clients to the port on the host (server socket). A client must know IP and port of the server (server socket) to send a request to it. Client's port is often provided by OS kernel when client starts communication with the server and is freed when communication is over.

Although communication using sockets is common and efficient, it is considered low level, because sockets only allow to transfer unstructured stream of bytes between processes. It is up to client and server applications to impose a structure on the data passed as byte stream.

Remote Procedure Calls

RPC is a higher level communication method. It was designed to mimic procedure call mechanism, but execute it over network. RPC is conceptually similar to message passing IPC and is usually built on top of socket communication. In contrast to IPC messages, RPC messages are well structured. Each message includes information about function to be executed and the parameters to be passed to that function. When the function is executed, a response with output is sent back to the requester in a separate message.

RPC hides the details of communication by providing a stub on the client side. When client needs to invoke a remote procedure, it invokes the stub and pass it the parameters. The stub marshals the parameters and sends a message to RPC daemon running on the server. RPC daemon (a similar stub on the server side) receives the message and invokes the procedure on the server. Return values are passed back to the client using the same technique.

Pipes

Pipe is one of the oldest and simplest IPC methods, that appeared in early UNIX systems. A pipe is an IPC abstraction with two endpoints, similar to a physical pipe. Usually one process puts data to one end of the pipe and another process consumes them from the other one.

Ordinary pipes

Ordinary pipes are unidirectional, i.e. allow only one-way communication. They implement standard producer-consumer mechanism, where one process writes to the pipe and another one reads from it. For two-way communication two pipes are needed. Ordinary pipes require a parent–child relationship between communicating processes, because a pipe can only be accessed from process that created or inherited it. Parent process creates a pipe and uses it to communicate with a child created via fork(). Once communication is over and processes terminated, the ordinary pipe ceases to exist.

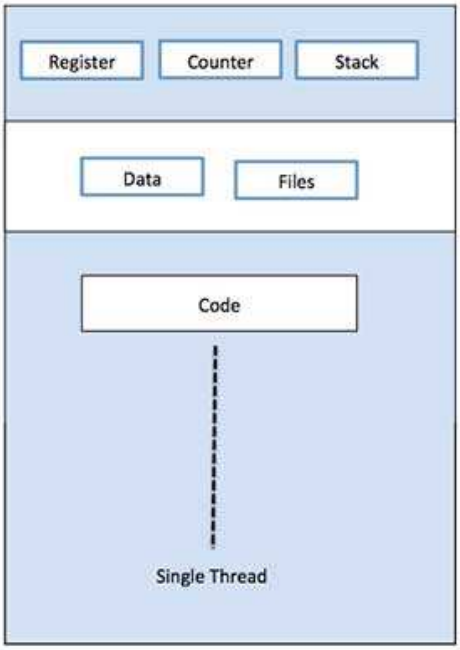
Named pipes

Named pipes are more powerful. They do not require parent–child relationship and can be bidirectional. Once a named pipe is created, multiple non related processes can communicate over it. Named pipe continues to exist after communicating processes have terminated. It must be explicitly deleted when not required anymore.

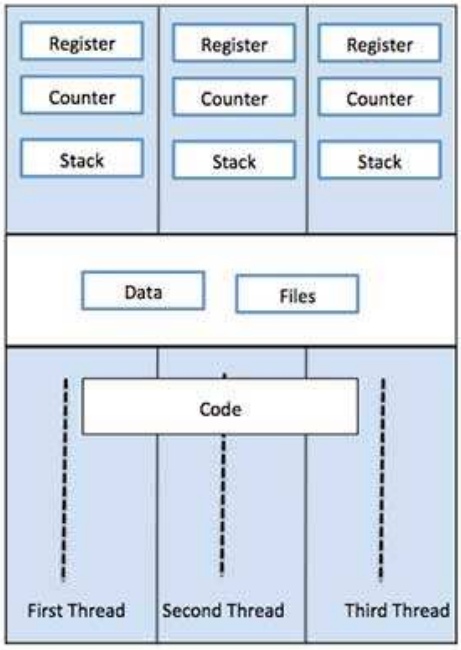
BASIC CONCEPTS OF THREADS

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history. A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a lightweight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

Sl.No.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways −

1. User Level Threads − User managed threads.
2. Kernel Level Threads − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

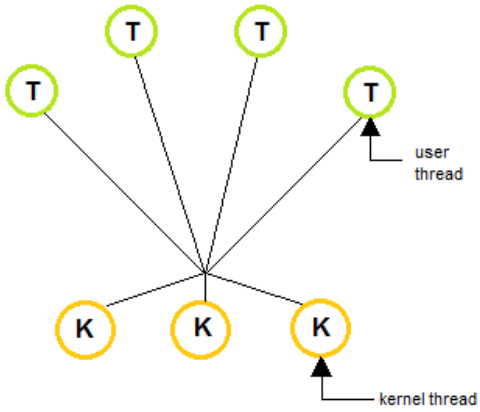
Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types:

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

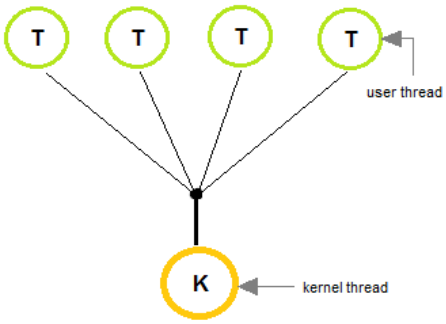
Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.



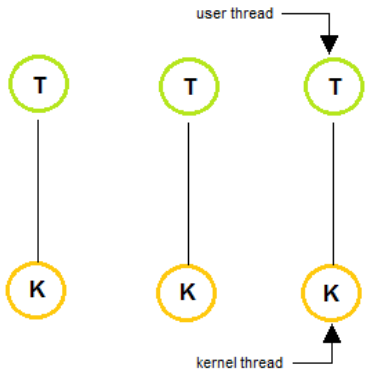
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Many to One Model



Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors. If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

One to One Model



There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

Difference between User-Level & Kernel-Level Thread

Sl.No.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

CONCURRENCY

Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Concurrent means something that happens at the same time as something else. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously. Concurrent processing is sometimes said to be synonymous with parallel processing.

Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon, of course. In the real world, at any given time, many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency. When dealing with concurrency issues in software systems, there are generally two aspects that are important: being able to detect and respond to external events occurring in a random order, and ensuring that these events are responded to in some minimum required interval. Concurrency cannot be avoided because:

- Users are concurrent - a person can handle several tasks at once and expects the same from a computer.
- Multiprocessors are becoming more prevalent.
- The Internet is perhaps a huge multiprocessor.
- A distributed system (client/server system) is naturally concurrent.
- A windowing system is naturally concurrent.

I/O is often slow because it involves slow devices such as disks, printers; many network operations are essentially (slow) I/O operations. When doing I/O it is helpful to handle the I/O concurrently with other work. Whenever concurrency is involved certain issues, discussed below, arise. An understanding of these issues is important when:

- writing an operating system.
- when interacting with the kernel, for example, when performing I/O.
- when generating multiple processes, for example, with forks and pipelines.
- when using multiple threads.

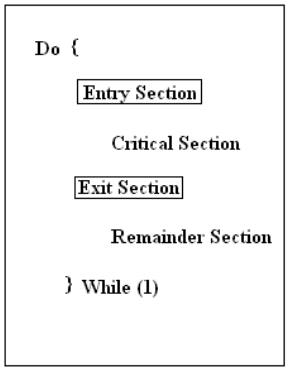
Concurrency issues:

- **Atomic**. An operation is atomic if the steps are done as a unit. Operations that are not atomic, but interruptible and done by multiple processes can cause problems. For example, an lseek followed by a write is not atomic. A process is likely to lose its time quantum between the lseek (a slow operation if the distance seeked is large!) and the write. If another process has the file open and does a write then the result is not what is intended.
- **Race conditions**. A race condition occurs if the outcome depends on which of several processes gets to a point first. For example, fork( ) can generate a race condition if the result depends on whether the parent or the child process runs first. Other race conditions can occur if two processes are updating a global variable.
- **Blocking and starvation**. While neither of these problems is unique to concurrent processes, their effects must be carefully considered. Processes can block waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable. Starvation occurs when a process does not obtain sufficient CPU time to make meaniful progress.
- **Deadlock**. Deadlock occurs when two processes are blocked in such a way that neither can proceed. The typical occurrence is where two processes need two non-shareable resources to proceed but one process has acquired one resource and the other has acquired the other resource. Acquiring resources in a specific order can resolve some deadlocks.

PRINCIPLES OF CONCURRENCY

Concurrency is the tendency for things to happen at the same time in a system. It also refers to techniques that make program more usable. Concurrency can be implemented and is used a lot on single processing units, nonetheless it may benefit from multiple processing units with respect to speed. If an operating system is called a multi-tasking operating system, this is a synonym for supporting concurrency. If we can load multiple documents simultaneously in the tabs of our browser and we can still open menus and perform more actions, this is concurrency. If we run distributed-net computations in the background, that is concurrency.

MUTUAL EXCLUSION AND CRITICAL SECTION



Consider a system consisting of n processes {P0, P1, …, Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. A picture (right side) showing general structure of a process Pi.

A solution to the critical-section problem must satisfy the following three requirements:

- **Mutual Exclusion**: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

- **Progress**: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

- **Bounded Waiting**: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

SEMAPHORE

To generalize to more complex problems the solutions to the critical-section problem are not easy. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait, to test) and V (for signal, to increment). The classical definition of wait in pseudo code is:

```
wait(S) {
while (S 5 0)
; // no-op
s--;
}
```

The classical definitions of signal in pseudo code are:

```
Signal (S) {
    S++;
}
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait (S), the testing of the integer value of S (S 5 O), and its possible modification (S--), must also be executed without interruption.

DEAD LOCK

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

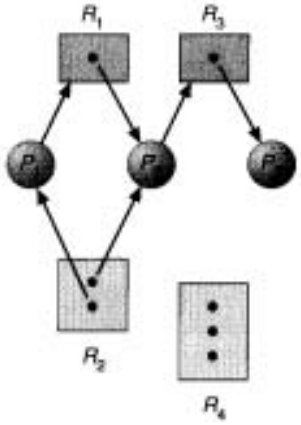
- **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

- **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

- **No preemption**: Resources cannot be preempted; that is, resources can be released only voluntarily by the process holding it, after that process has completed its task.

- **Circular wait**: A set {P0, P1, ..., Pn) of waiting processes must exist such that P0 is waiting for a resource that is held by PI, PI is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by P,, and P, is waiting for a resource that is held by P0.

Resource-Allocation Graph (RAG)



Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm), the set consisting of all resource types in the system.

The resource-allocation graph show depicts the following situation.

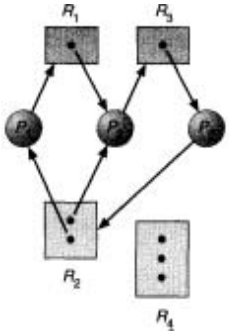
The sets P, R, and E:

P = {P1, P2, P3}
R = {R1, R2, R3, R4}
E = {P1 R1, P2 R3, R1 P2, R2 P2, R2 P1, R3 P3}

Resource instances:

One instance of resource type R1
Two instances of resource type R2
One instance of resource type R3
Three instances of resource type R4

Process states:



Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1
Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
Process P3 is holding an instance of R3.

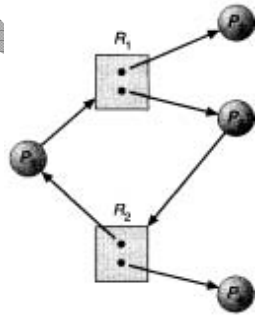
A request edge P3 R2 is added to the graph. At this

point, two minimal cycles exist in the system. They are:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

The processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1. The Resource allocation graph with a deadlock in the figure.

Now consider the resource-allocation graph in following Figure. In this example, we also have a cycle. However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state.



HANDLING DEADLOCK

We can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

DEAD LOCK PREVENTION

Prevent deadlocks by restraining how requests can be made. For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only

files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically non-sharable.

Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

No Preemption

The third necessary condition is that there is no preemption of resources that have already been allocated. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. If a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

Circular Wait

The fourth for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. We can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) → F(Rj). Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

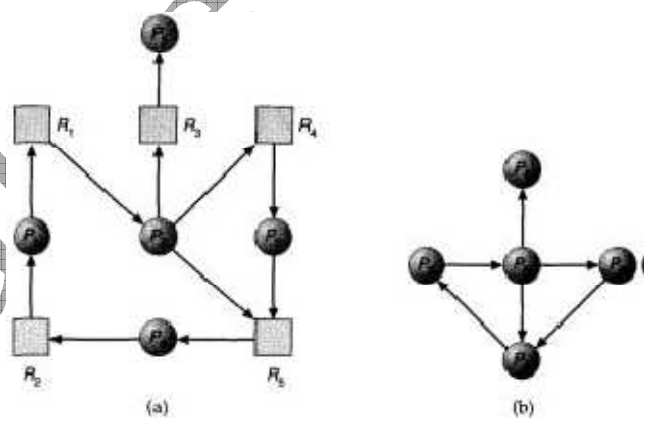
DEAD LOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.



More clearly, an edge from Pi to Pi in a wait-for graph implies that process Pi is waiting for process Pi to release a resource that Pi needs. An edge Pi → Pi exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq. For example, in Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.

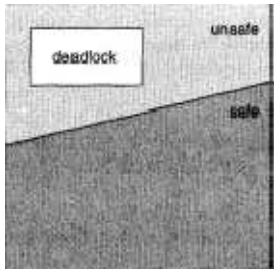
If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow. Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.

DEAD LOCK AVOIDANCE

The method for avoiding deadlocks is to require additional information about how resources are to be requested. Each process declares the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the deadlock-avoidance approach.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

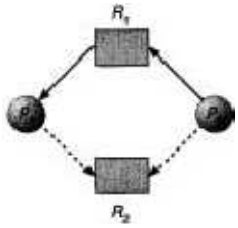
Safe State



A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. The Safe, unsafe, and deadlock state spaces are shows in the figure. A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock.

Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in previous section can be used for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge Pi→Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi→Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi the assignment edge Rj →Pi is reconverted to a claim edge Pi → Rj. We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-



allocation graph. We can relax this condition by allowing a claim edge Pi→Rj to be added to the graph only if all the edges associated with process Pi are claim edges.

Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process Pi will have to wait for its requests to be satisfied.

Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

We need the following data structures:

- **Available**: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.

- **Max**: An n, x m matrix defines the maximum demand of each process. If Max[i,j] = k, then process Pi may request at most k instances of resource type Ri.

- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.

- **Need**: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then process Pi may need k more instances of resource type Ri to complete its task. Note that Need[i,j] = Max[i,j] - Allocafion[i,j].

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.

```
Initialize Work := Available   and
Finish[i] :=false for i = 1,2, ..., n.
```

2. Find an i such that both

```
a.  Finish[i] =false
b.  Needi 5 Work.
```

If no such i exists, go to step 4.

3. Work := Work + Allocation

```
Finish[i] := true
go to step 2.
```

4. If Finish[i] = true for all i, then the system is in a safe state. This algorithm may require an order of m x n2 operations to decide whether a state is safe.

Resource-Request Algorithm

Let Requesti be the request vector for process Pi. If Request;[j] = k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Requesti 5 Needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Requesti 5 Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

```
Available := Available - Request,;
Allocationi := Allocation; + Request;;
Needi := Needi - Requesti;
```

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti and the old resource-allocation state is restored.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, one possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.

- **Abort one process at a time until the deadlock cycle is eliminated**: This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

- **Selecting a victim**: Which resources and which processes are to be preempted? As in process termination, we must determine the order of pre-emption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

- **Rollback**: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

- **Starvation**: How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.