Deadlocks

What is deadlock?

- Waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.
- E.g.- “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

Resources

- The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.
- If a system has two CPUs, then the resource type has two CPU instances. Similarly, the resource type printer may have five instances.
- Under the normal mode of operation, a process may utilize a resource in only the following sequence:

  1. Request
  2. Use
  3. Release
- Using system calls- request() release() device , open() and . close() file, allocate(), free() memory system calls.

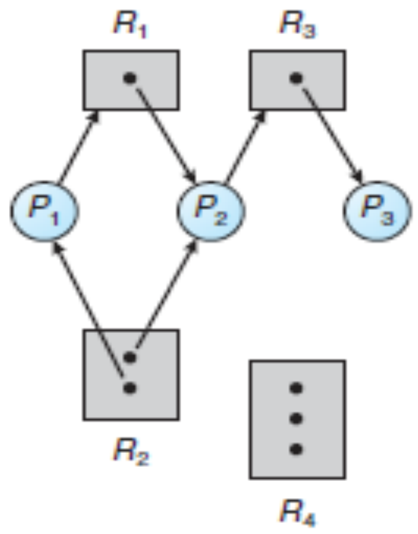
Deadlock Characterization

- **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, a resource 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 held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Example

```
void *do work one(void *param)
{ pthread mutex lock(&first mutex);
pthread mutex lock(&second mutex);
/** Do some work*/
pthread mutex unlock(&second mutex);
pthread mutex unlock(&first mutex);
pthread exit(0);
}
void *do work two(void *param)
{ pthread mutex lock(&second mutex);
pthread mutex lock(&first mutex);
/** Do some work*/
pthread mutex unlock(&first mutex);
pthread mutex unlock(&second mutex);
pthread exit(0);
}
```

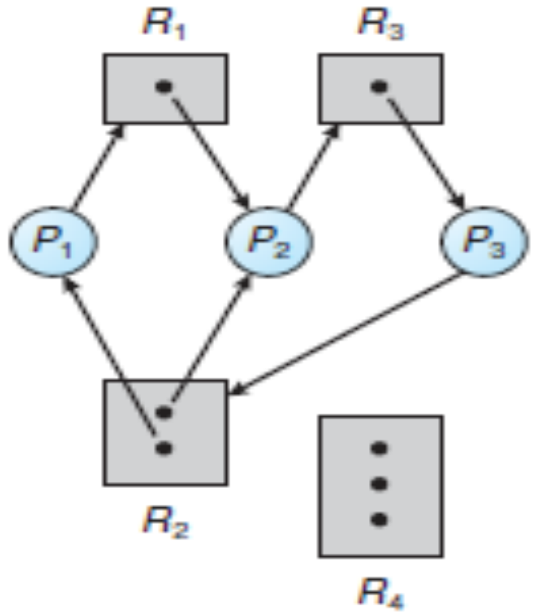
Resource-Allocation Graph



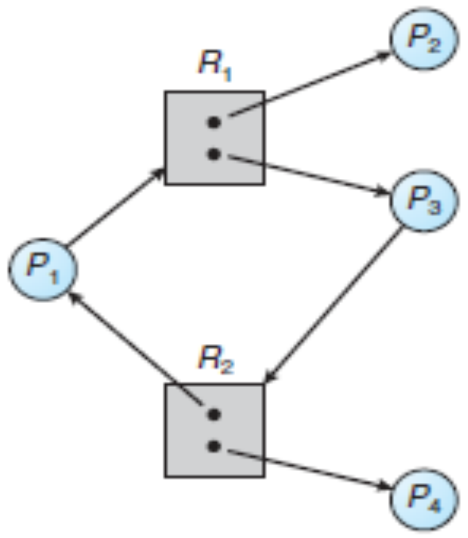
Rj → Pi - Rj has been allocated to process Pi

Pi → *Rj* - *Pi* has requested an instance of resource type *Rj*

- graph contains no cycles, then no process in the system is deadlocked
- graph does contain a cycle, then a deadlock may exist
- a cycle in the graph is both **a necessary and a sufficient condition** for the existence of deadlock if there is only on instance of each resource type.
- a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock- if there is multiple instances of each resources type



R_1
R_3
P_1
P_2
P_3
R_2
R_4



R_1
P_2
P_3
P_1
R_2
P_4

Methods for Handling Deadlocks

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.
- Ignoring the possibility of deadlocks is cheaper than the other approaches.

Deadlock Prevention

- By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.
- by examining each of the four necessary conditions separately.

Mutual Exclusion

- Read-only files are a good example of a sharable resource.
- But, some resources are intrinsically non sharable.- cannot prevent deadlocks by denying the mutual-exclusion condition

Hold and Wait

- 1. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.
- 2. An alternative protocol allows a process to request resources only when it has none.

E.g. consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.

- **two main disadvantages**

1. utilization may be low
2. Starvation is possible.

No Pre-emption

Two Protocols:

- 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 the process is currently holding are preempted. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- we check whether resources 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. **Or** If the resources are neither available nor held by a waiting process, the requesting process must wait.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as mutex locks and semaphores.

Circular Wait

- to require the resources, each process requests resources in an increasing order of enumeration.

E.g., F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12, define a one-to-one function $F: R \rightarrow N$,

Obeys $F(R_i) \geq F(R_j)$.

To use the tape drive and printer at the same time must first request the tape drive and then request the printer.

- $F(R0) < F(R1) < \dots < F(Rn) < F(R0)$ fails in case of circular wait.
- It is up to application developers to write programs that follow the ordering.

witness

- lock-order verifier
- Witness records the relationship that first mutex must be acquired before second mutex If later thread two acquires the locks out of order, witness generates a warning message on the.
- It is also important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. system console.
- void transaction(Account from, Account to, double amount)

```
{ mutex lock1, lock2;
lock1 = get lock(from);
lock2 = get lock(to);
acquire(lock1);
acquire(lock2);
withdraw(from, amount);
deposit(to, amount);
release(lock2);
release(lock1);
}
```

```
transaction(checking account, savings account, 25);

transaction(savings account, checking account, 50);
```

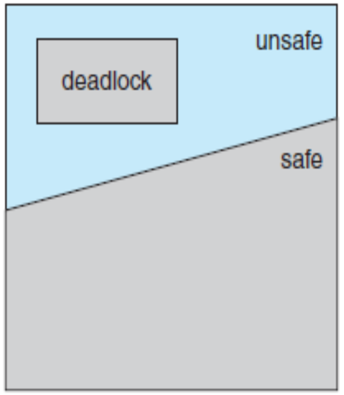
Deadlock Avoidance

- Prevents deadlocks by limiting how requests can be made.
- Disadvantage- are low device utilization and reduced system throughput.
- Extra knowledge – order of request for P and Q i.e. printer or drive AND maximum number of resources of each type that it may need.
- deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- 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.
- E..g- total resources- 12

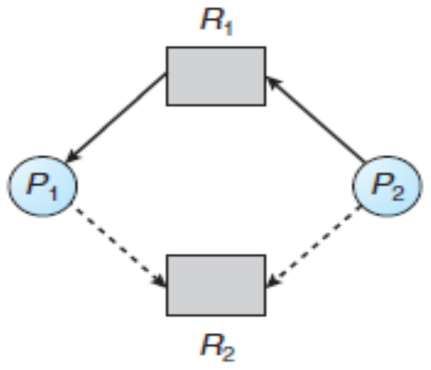
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

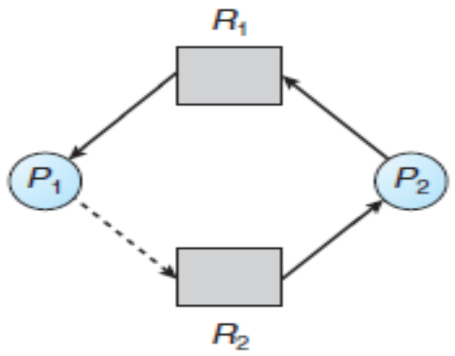


- <*P*1, *P*0, *P*2> satisfies the safety condition
- A system can go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated one more tape drive.
- The request is granted only if the allocation leaves the system in a safe state.

Resource-Allocation-Graph Algorithm

- resource-allocation graph with a new new type of edge, called a claim edge.
- Represented bv a dashed line

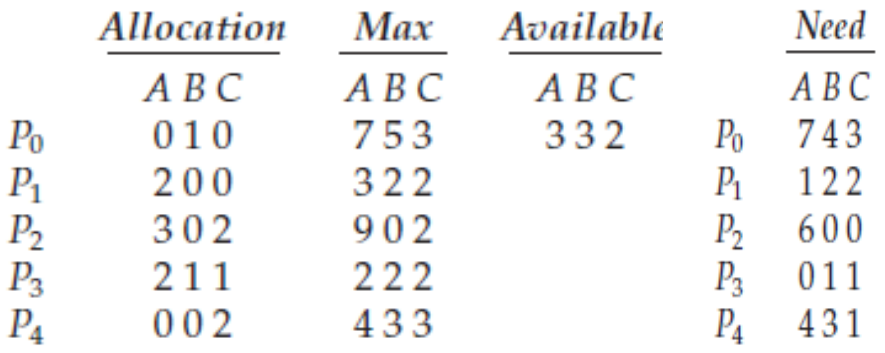




- The request can be granted only if converting the request edge Pi-> Rj to an assignment edge Rj-> Pi does not result in the formation of a cycle in the resource-allocation.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource Allocation system with multiple instances of each resource type.
- Process must declare the maximum number of instances of each resource type that it may need.



	Allocation	Max	Availabl		Need
	A B C	A B C	A B C		A B C
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2		P_1	1 2 2
P_2	3 0 2	9 0 2		P_2	6 0 0
P_3	2 1 1	2 2 2		P_3	0 1 1
P_4	0 0 2	4 3 3		P_4	4 3 1

CASES :

(1,0,2) by P1
(3,3,0) by P4
(0,2,0) by P_0

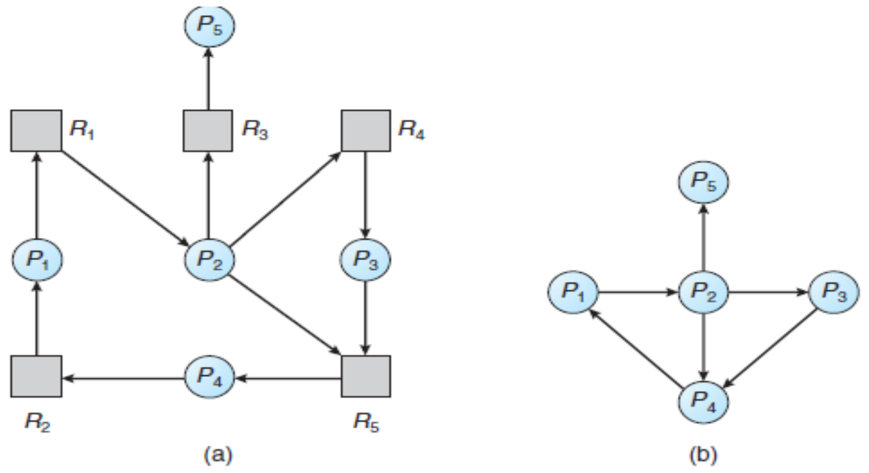
<P1, P3, P4, P2, P0>

Deadlock Detection

- 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

- Wait for graph



Several Instances of a Resource Type