Lecture 10

Resource Allocation and Deadlock

Previously …

- We have seen how threads may be synchronized with each others by condition variable to execute the desired portion of code.
- We have seen how the powerful tools, called semaphores, are implemented as locks and condition variable.

Resource Allocation

- Resource allocation is a common requirement in operating systems
- Many resources need to be allocated dynamically, e.g. memory, printers, etc.
- Resource allocation introduces the issues of:
  - Mutual Exclusion – when a resource can only be used by one process (or thread) at a time
  - Starvation – a process (or thread) may be "locked out" by other processes (or threads) and prevented from acquiring a resource
  - Deadlock – two or more processes (or threads) may wait infinitely for a resource held by the other

In this lecture, we will use the term "process" rather than "thread"

Deadlock

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
  - Example
    - Semaphores A and B, each initialized to 0

P0	P1
sem_wait(A);	sem_wait(B)
sem_post(B);	sem_post(A)

Deadlock Conditions

- Deadlock can arise if four conditions happen simultaneously
  - **Mutual Exclusion:** only one process at a time can use a resource
  - **No Preemption:** a resource can only be released voluntarily by the process holding it, after that process has completed its task
  - **Hold and Wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **Circular Wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

Resources

- Resource types $R_1, R_2, \ldots, R_m$

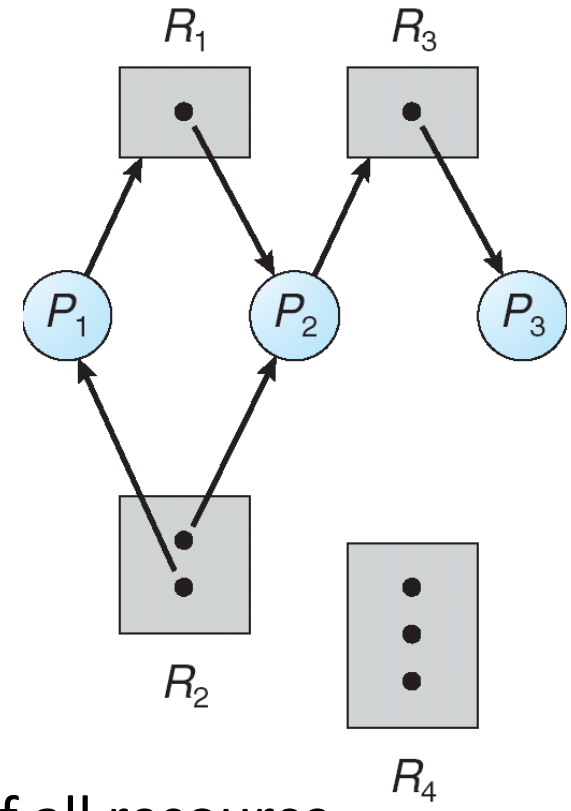
  CPU, memory space, I/O devices

- Each resource type R_i has W_i instances
- A resource can be utilized by each processes as follows:
  - requested (acquiring)
  - used (holding)
  - released

sem_wait(A);

sem_post(A);

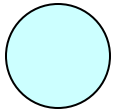
Resource Allocation Graph



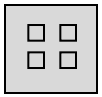
- A set of vertices *V* and a set of edges *E*
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
  - E is partitioned into two types:
    - Request edge – directed edge $P_i \rightarrow R_j$
    - Assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph

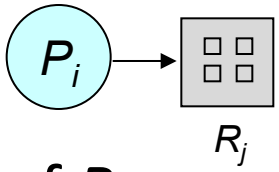
- Process



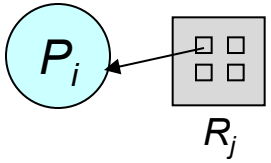
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is using an instance of R_j



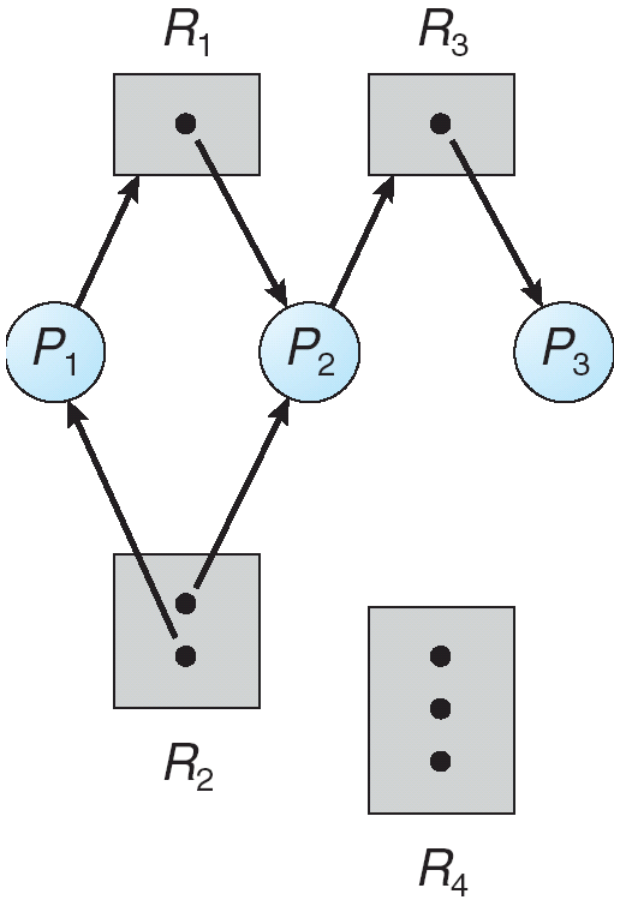
Deadlock?

Mutual Exclusion:

No Preemption:

Hold and Wait:

Circular Wait:



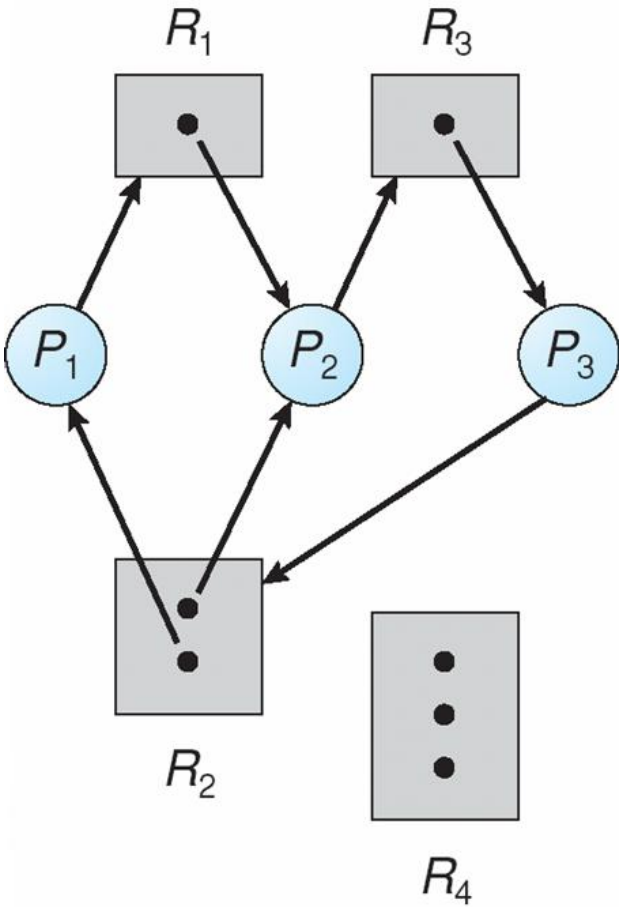
Graph With Deadlock

Mutual Exclusion:

No Preemption:

Hold and Wait:

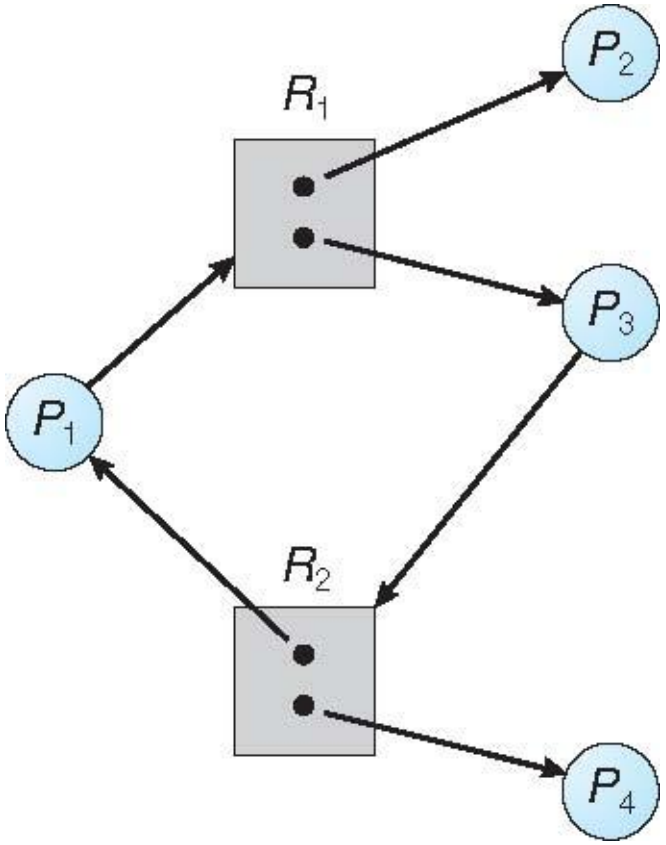
Circular Wait:



Graph With Cycle But No Deadlock

- If graph contains no cycles $\Rightarrow$ no deadlock

- If graph contains a cycle $\Rightarrow$
  - If only one instance per resource type, then deadlock
  - If several instances per resource type, possibility of deadlock



Deadlock Management

- Deadlock Prevention
  - Design the system in such a way that deadlocks can never occur
- Deadlock Avoidance
  - Allow the possibility of deadlock but avoid it before it occurs
- Deadlock Detection and Recovery
  - Allow the possibility of deadlock, determine if deadlock has occurred and which processes and resources are involved
  - After detection, clear the problem, allow processes to complete and resources to be reused (may require terminating processes)
- Ignore the Problem
  - Pretend that deadlocks never occur (some systems do this)

(1) Deadlock Prevention

Limit the ways, a request can be made to ensure that one of ***the four conditions*** for deadlock does not happen.

Recall: Deadlock Conditions

- **Mutual Exclusion:** only one process at a time can use a resource
- **No Preemption:** a resource can only be released voluntarily by the process holding it, after that process has completed its task
- **Hold and Wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **Circular Wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, … , P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

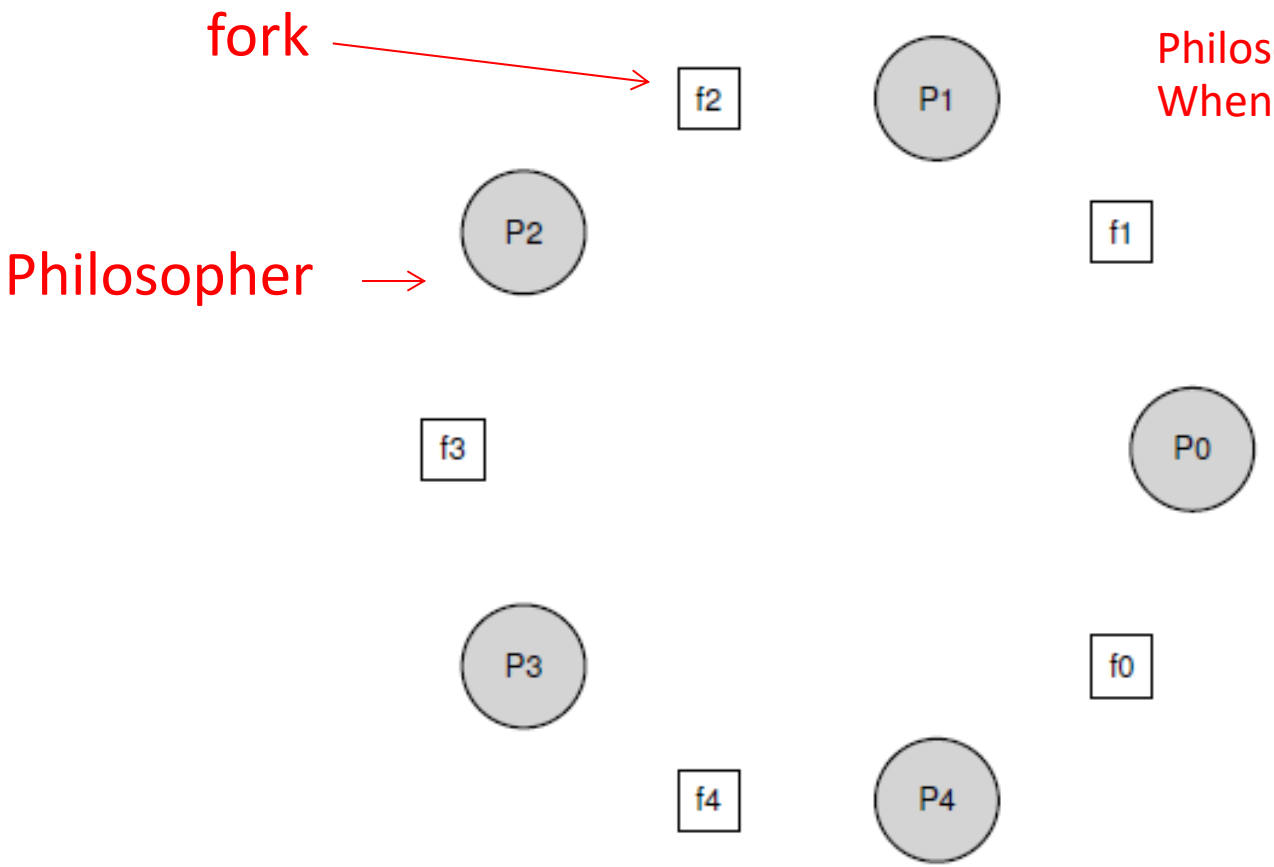
Deadlock Prevention: Circular Wait

- Impose a total ordering of all resources types and require that each process requests resources in increasing order of resource type
- If a resource of type j is held, process can only request resources of types > j
  - Example: Semaphores A and B, each initialized to 1

P0	P1
sem_wait(A);	sem_wait(B)
sem_post(B);	sem_post(A)

  - Example: The Dining Philosophers
    - Require that each philosopher requests forks in increasing order of fork number

The Dining Philosophers



Philosophers either think or eat
When eat, they need two forks

```
while (1)
{
    think();
    getforks(p);
    eat();
    putforks(p);
}
```

Write the routines getforks(p) and putforks(p) such that there is no deadlock, no philosopher starves, and as many philosophers can eat at the same time as possible

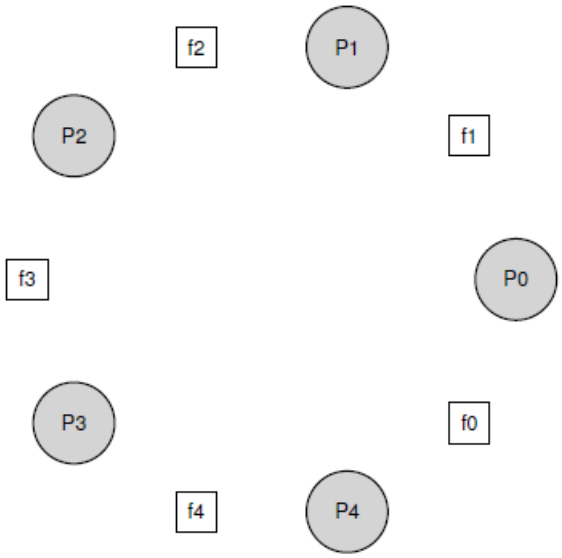
Dining Philosophers, Deadlock

```
sem_t left(int p) { return p; }

sem_t right(int p) { return (p + 1) % 5; }

void getforks(p) {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks(p) {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```



A circular wait occurs if each philosopher picks up the fork to the left and then waits to pick up the fork to the right

Dining Philosophers, No Deadlock

```
sem_t left(int p) { return p; }

sem_t right(int p) { return (p + 1) % 5; }

void getforks(p) {
    if (p == 4) {
        sem_wait(forks[right(p)]);
        sem_wait(forks[left(p)]);
    } else {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
    }
}

void putforks(p) {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

If philosopher 4 picks up the fork to the right first , no circular wait occurs as each philosopher requests forks in the same order

Deadlock Prevention: Hold and Wait

- Prevent this condition by acquiring all resources at once
- Use a semaphore "Prevention" to ensure that this is done atomically

```
sem_wait(Prevention);
sem_wait(R1); // acquire resource R1
sem_wait(R2); // acquire resource R2
...
sem_post(Prevention);
```

- This has the problem of low resource utilization and starvation is possible

Deadlock Prevention: No Preemption

- If a process that holds some resources requests another resource that cannot be immediately allocated, then all resources currently held are released
- This can be done by using trylock() (or trywait() on semaphores) to acquire the second or subsequent resource

```
do {
    sem_wait(R1); // acquire R1
    flag = sem_trywait(R2); // try to acquire R2
    if (flag != 0) { // failed to acquire R2
        sem_post(R1); // release R1
        yield(); // or random delay
    }
} while (flag != 0)
...
```

Deadlock Prevention: Mutual Exclusion

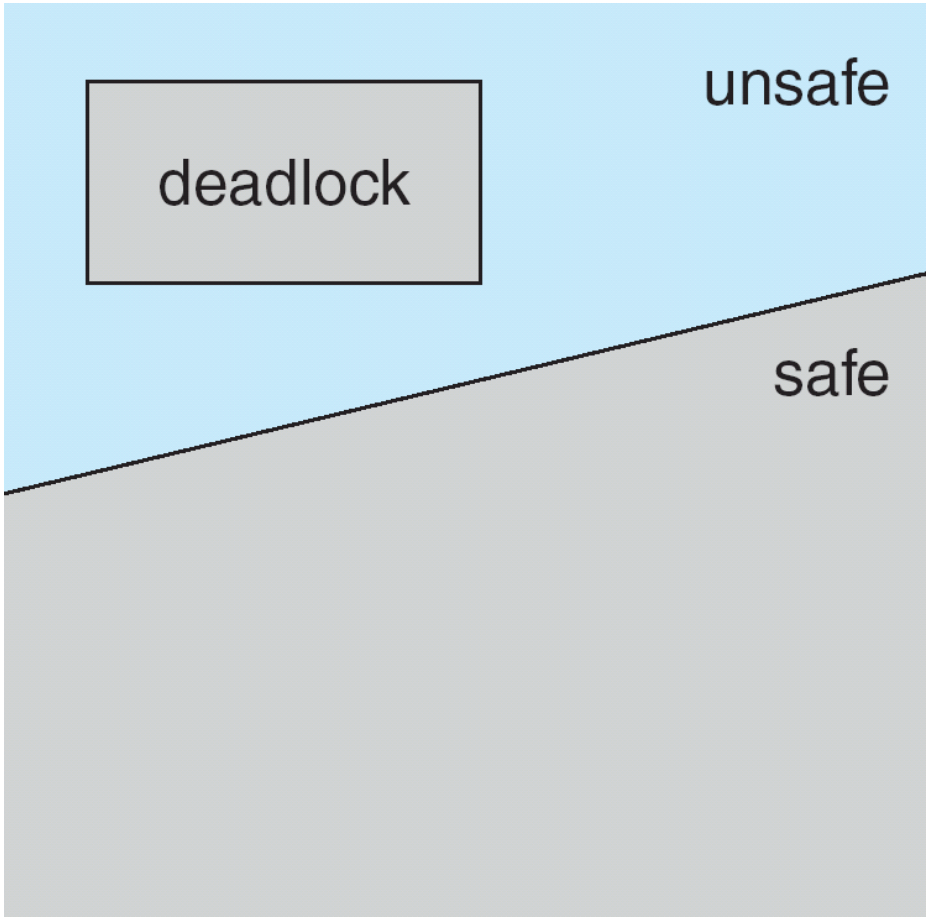
- We know this is difficult, because the code we wish to run does indeed have critical sections.
  - Some resources may be sharable and do not need mutual exclusion
  - Using wait-free locking
    - Not use explicit locking, there is called wait-free technique. (require powerful h/w instruction)

(2) Deadlock Avoidance

- Requires that the system has some additional information available
  - The deadlock avoidance algorithm dynamically examines *the resource allocation state* to ensure that there can never be a circular wait condition
  - Resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
  - Simplest and most useful model requires that each process declare the *maximum number* of resources' instance of each type that it may need

Safe, Unsafe, Deadlock State

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state



Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*
  - *A safe state* if one where there exists a sequence $<P_1, P_2, \ldots, P_n>$ of *all* processes such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When all P_j have finished, P_i can obtain needed resources, execute, return allocated resources, and finish
  - When P_i is finished, P_{i+1} can obtain its needed resources, and so on

Banker's Algorithm

- Used for deadlock avoidance when there are multiple instances of each resource type
- Safety Algorithm
  - Used to check whether the system is in a safe state
- Resource Request Algorithm
  - Used to determine if a request can be granted
    - If requested resources are not available, process must wait
    - If requested resources are available, "pretend" to allocate them and check whether the system is in safe state
    - If unsafe, process must wait, and the old state is restored

Example

- Suppose there is a single resource type, with 10 instances
  - At a particular time, the resource allocation state is as follows:

	Allocation	Max	Need
P1	1	6	5
P2	1	5	4
P3	2	4	2
P4	4	7	3

Need = Max – Allocation

Available = 2 instances

  - If P4 asks for one more instance, does this lead to a safe or unsafe state?
  - If P3 asks for one more instance, does this lead to a safe or unsafe state?

Example

- If P4 asks for one more instance, does this lead to a safe or unsafe state?
  - "Pretend" to allocate request – the state changes to:

	Allocation	Max	Need
P1	1	6	5
P2	1	5	4
P3	2	4	2
P4	5	7	2

Available = 1 instance

  - The request from P4 is unsafe, as for all P_i, $Need_i$ > Available
  - If all P_i request the number of instances they still need, deadlock will occur – therefore request is not granted

Example

- If P3 asks for one more instance, does this lead to a safe or unsafe state?
  - "Pretend" to allocate request – the state changes to:

	Allocation	Max	Need
P1	1	6	5
P2	1	5	4
P3	3	4	1
P4	4	7	3

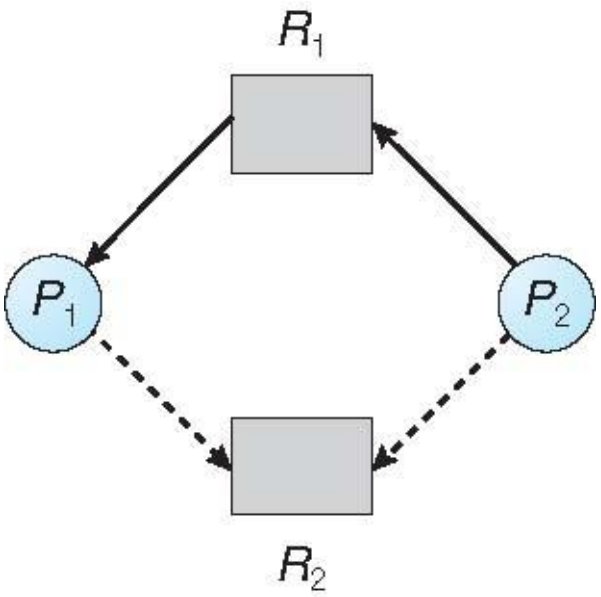
Available = 1 instance

  - If all P_i request the number of instances they still need, the sequence <P3, P2, P1, P4> satisfies safety requirement
  - The request from P3 is safe – therefore request is granted

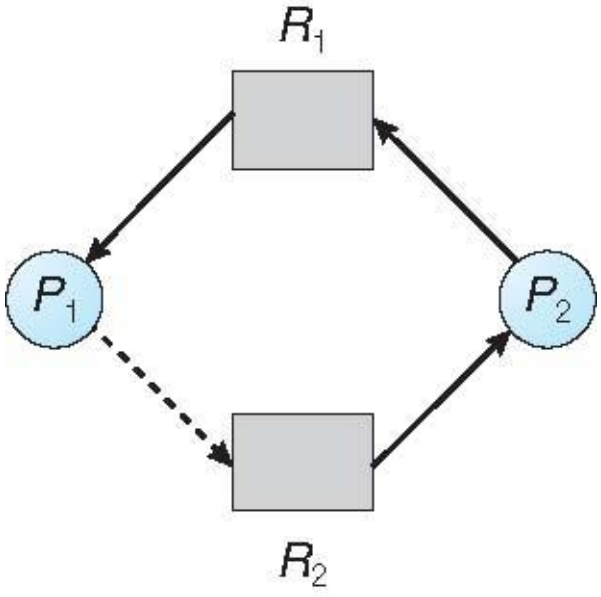
Avoidance Graph

- Used for deadlock avoidance when there is only one instance of each resource type
  - *Claim edge* $P_i \rightarrow R_j$ represented by a dashed line indicates that process P_i may need a resource R_j
  - Claim edge converts to *request edge* when the process requests a resource
  - Request edge converts to an *assignment(used) edge* when the resource is allocated to the process
  - When a resource is released by a process, assignment edge reconverts to a *claim edge*
  - Resources must be claimed in advance

Resource Allocation Graph



Safe State – No Cycle



Unsafe State – Cycle

A request can be granted only if converting the request edge to an assignment edge does not result in a cycle in the graph

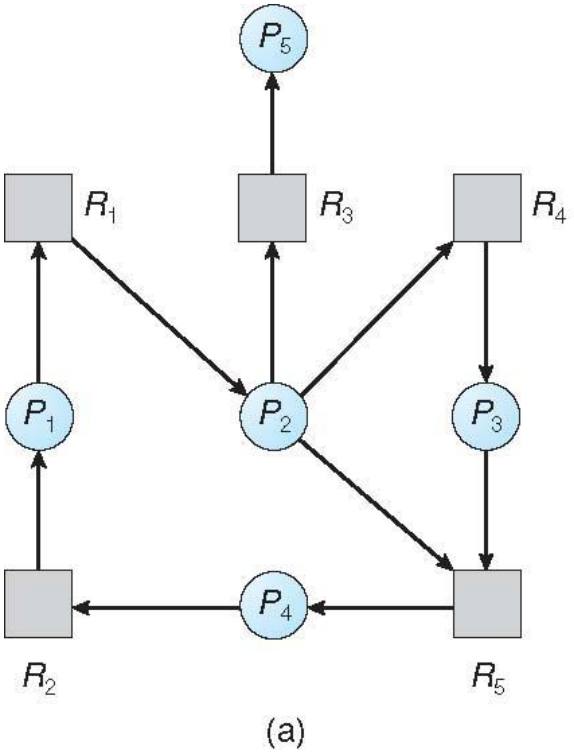
- Request by P_1 for R_2 can be granted
- Request by P_2 for R_2 cannot be granted

(3) Deadlock Detection and Recovery

- Allow system to enter deadlock state
  - Instead of preventing or avoiding deadlock, we allow deadlock to occur
- Detection algorithm
  - Execute a deadlock detection algorithm periodically to check for deadlocks
- Recovery scheme
  - If there is a deadlock, take some action to recover

Wait-For Graph

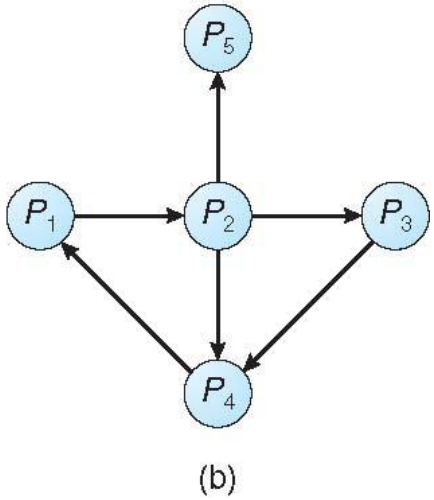
- If only one instance of each resource type, we maintain a Wait-For graph



Resource Allocation Graph

Nodes are processes

$P_i \rightarrow P_j$ if P_i is waiting for P_j



Corresponding Wait-For Graph

Deadlock Detection

- If only one instance of each resource type
  - Periodically invoke an algorithm that searches for a cycle in the Wait-For graph – if there is a cycle, there is a deadlock
  - An algorithm to detect a cycle in a graph requires $O(n^2)$ operations, where n is the number of vertices in the graph
- If multiple instances of each resource type, a more general algorithm is required
  - A general algorithm to detect deadlock requires $O(m \times n^2)$ operations, where m is the number of resources types and n is the number of processes

Deadlock Recovery

- Terminate all deadlocked processes, or
- Terminate one process at a time until the deadlock cycle is eliminated
- In which order should we choose to terminate?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated?
  - Is process interactive or batch?

Summary

- Concurrency exists between different processes and different threads of the same process
- Critical Sections (e.g. accessing shared resources or shared data) need to be executed with mutual exclusion
- Locks and Condition Variables may be used by threads for mutual exclusion and synchronization
- Semaphores may be used by threads or processes for mutual exclusion and synchronization
- Resource Allocation may lead to deadlock where two or more processes (or threads) may wait indefinitely for a resource held by the other