System model

- Resources: CPU, memory, files, devices, semaphores, etc.
  - divided into several types
  - each type has one or more *identical* instances
- Processes:

1. Request resource.
   (If resource cannot be granted immediately, process waits until it can acquire resource.)
2. Use resource.
3. Release resource.

Deadlock conditions

- **Mutual exclusion:** resources cannot be shared
- **Hold and wait:** processes must not release resources just because they are waiting
- **No preemption:** a resource can only be released voluntarily by the process holding the resource
- **Circular wait:** there must exist a set $\{P_0, \ldots, P_{n-1}\}$ of waiting processes such that P_i is waiting for a resource held by $P_{(i+1)\ \%\ n}, i = 0,\ \ldots,\ n-1$.

Resource allocation graph

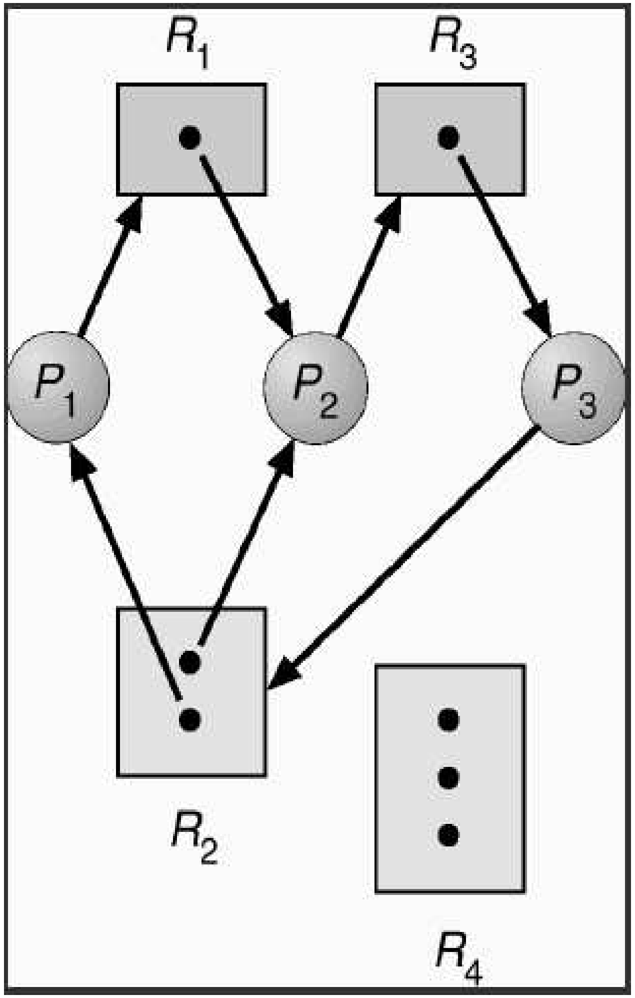
Let $P = \{P_1, \ldots, P_n\}$ be set of active processes in the system
$R = \{R_1, \ldots, R_m\}$ be set of all resource types in the system

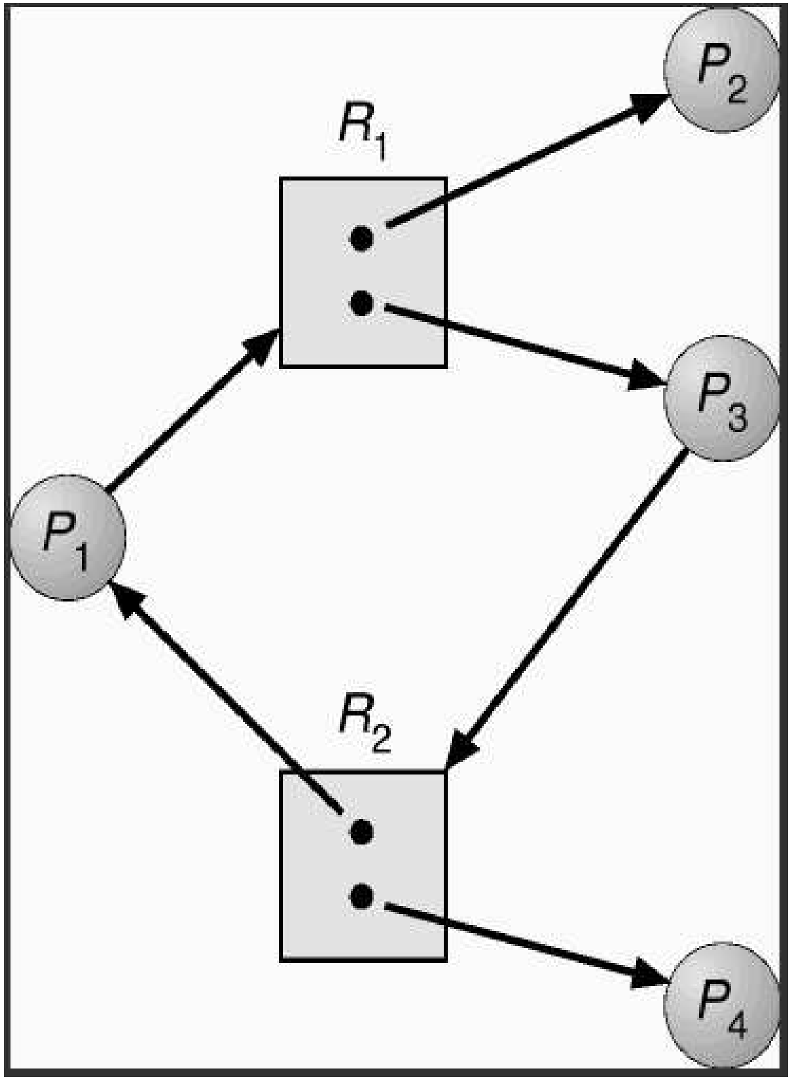
$V = P \cup R$
$E = E_{req} \cup E_{assign}$
$E_{req} = \{P_i \rightarrow R_j \mid P_i$ has requested an instance of R_j and is waiting for it$\}$
$E_{assign} = \{R_j \rightarrow P_i \mid P_i$ holds an instance of$R_j\}$

- If the graph does not contain a cycle, no deadlock exists
- If the graph contains a cycle **and** each resource node in the cycle has exactly one instance, then deadlock exists
- Otherwise, deadlock may or may not be present

Resource allocation graph

Examples:





Deadlock handling

- Prevention: system/processes ensure(s) that at least one of the necessary conditions does not hold
- Avoidance: processes follow a protocol to ensure that deadlock never happens
- Recovery
- Ignore deadlock: system may have to be restarted if deadlock occurs

  (used in most operating systems)

Deadlock prevention

- Hold and wait
  - process must request and be allocated all resources before it begins execution, **or**
    a process can request resources only when it has none
  - low resource utilization; starvation possible
- No Preemption
  - if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - preempted resources are added to the list of resources for which the process is waiting
  - process is restarted only when it can get old resources + newly requested resources
- Circular Wait
  - all resource types totally ordered
  - processes must requests resources in increasing order

Deadlock avoidance

Resource allocation state:

- # of available / allocated instances of each type
- maximum demand of each process

Safe sequence: For an allocation state, a sequence $\langle P_1, \ldots, P_n \rangle$ is safe if for each P_i, the maximum resources that P_i can request can be satisfied by currently available resources + resources held by all P_j $(j < i)$

Safe state: System istb in safe state if there exists a safe sequence consisting of all processes

Deadlock avoidance: RAG algorithm

Case I: only one instance of each resource type

- Claim edge $P_i \rightarrow R_j \;\Leftrightarrow\; P_i$ *may* request resource R_j (represented by a dashed line)
- Claim edge is converted to a request edge when a process requests a resource
- Assignment edge is converted to a claim edge when a process releases a resource

Method:
Request $P_i \rightarrow R_j$ is granted only if converting the request edge to an assignment edge does not result in a cycle in the RAG

NOTE: Resources must be claimed a priori

Deadlock avoidance: Banker's algorithm

Case II: multiple instances of each resource type

Data structures:

- `Available[i]`: number of available instances of resource R_i
- `Max[i,j]`: maximum number of instances that P_i may request of resource R_j
- `Alloc[i,j]`: # of instances of R_j currently allocated to P_i
- `Request[i,j]`: # of instances of R_j currently requested by P_i
- `Need[i,j]`: `Max - Alloc`

Deadlock avoidance: Banker's algorithm

Safety algorithm:

```
    Work = Available   Finish = {0 .. 0}
L1: find i such that (Finish[i] = 0 && Need[i] <= Work)
    if (no such i exists) goto L2
    else {
       Work = Work + Allocation[i]
       Finish[i] = 1
       goto L1
    }
L2:  if (Finish [i] == 1 for all i) return safe
    return unsafe
```

Deadlock avoidance: Banker’s algorithm

Resource allocation algorithm:

```
if (Request[i] > Need[i]) error /* maximum exceeded */
if (Request[i] > Available) wait /* resources not available */

/* pretend to allocate requested resources */
Allocation[i] += Request[i]
Available -= Request[i]
Need[i] -= Request[i]

if (safety_algorithm() == safe) allocate resources
else {
  restore old resource-allocation state
  put Pi to sleep
}
```

Deadlock detection

Single instance of each resource:

- **Wait-for graph** = (V, E) where
  $V = \{P_1, \ldots, P_n\}$ (set of active processes in the system)
  $E = \{P_i \rightarrow P_j \mid P_i$ is waiting for a resource held by $P_j\}$
- Deadlock exists iff wait-for graph contains a cycle

Deadlock detection

Multiple instances of each resource:

Data structures:

- `Available[i]`: number of available instances of resource R_i
- `Alloc[i,j]`: # of instances of R_j currently allocated to P_i
- `Request[i,j]`: # of instances of R_j currently requested by P_i

Deadlock detection

```
    Work = Available
    for each i: if (Alloc[i] != 0) Finish[i] = 0; else Finish[i]
L1: find i such that (Finish[i] = 0 && Request[i] <= Work)
    if (no such i exists) goto L2
    else {
       Work = Work + Allocation[i]
       Finish[i] = 1
       goto L1
    }
L2: for each i: if (Finish [i] == 0) Pi is deadlocked
```

Deadlock recovery

Section 7.7

Process termination

- Abort all deadlocked processes
- Select one process at a time and abort until deadlock cycle is eliminated
  - priority
  - computation time
  - amount and type of resources held / required

Resource preemption

- Victim selection
- Starvation
- Rollback