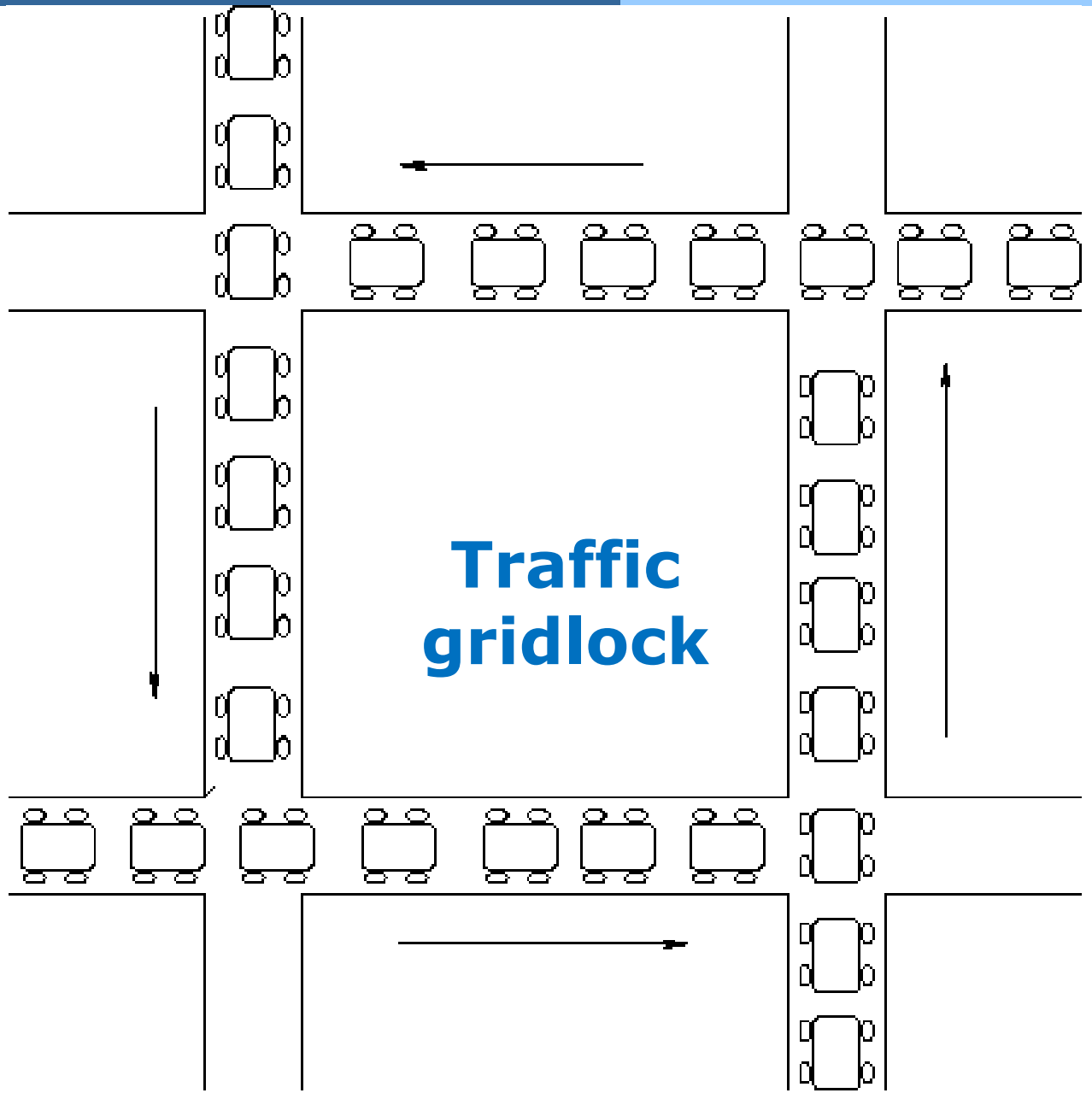
Chapter 7: Deadlocks



Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

- To develop a **description of deadlocks**, which prevent sets of concurrent processes from completing their tasks
- To present a number of different **methods** for ***preventing or avoiding deadlocks*** in a computer system

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each **process utilizes a resource** as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P0.

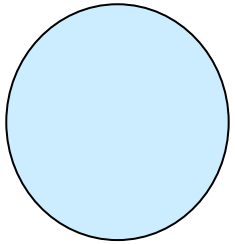
Resource-Allocation Graph

A set of vertices **V** and a set of edges **E**.

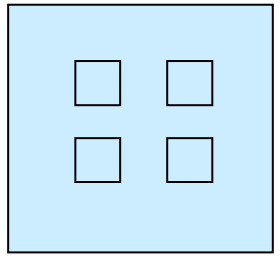
- **V** is partitioned into two types:
  - **P** = $\{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    R = $\{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

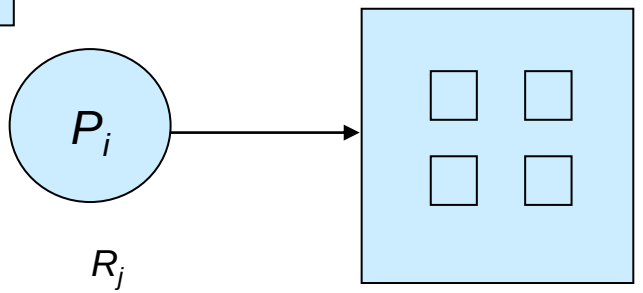
- Process



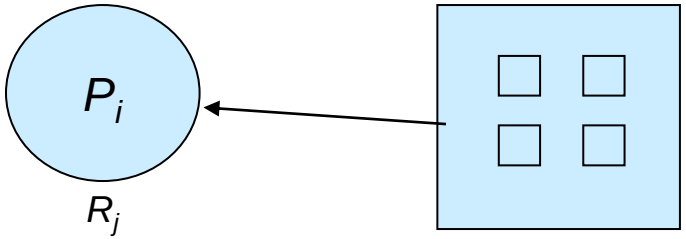
- Resource Type with 4 instances



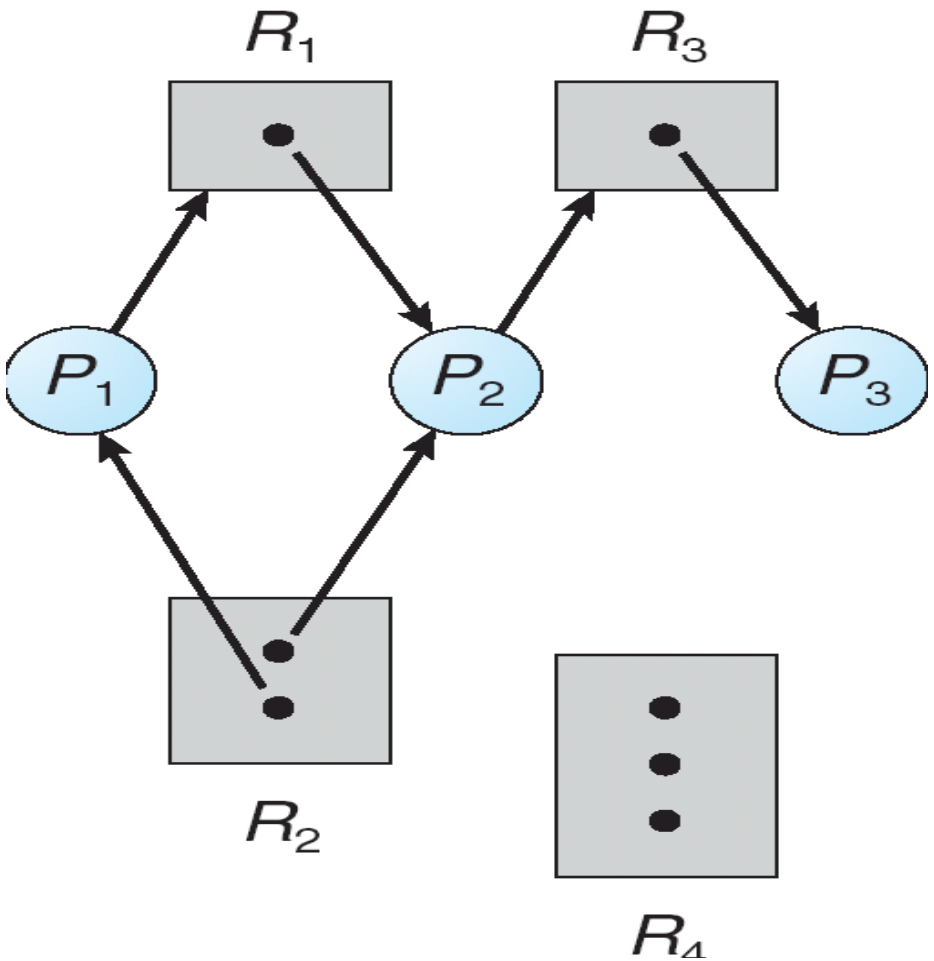
- P_i requests instance of R_j



- P_i is holding an instance of R_j

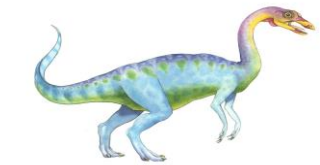


Example of a Resource Allocation Graph

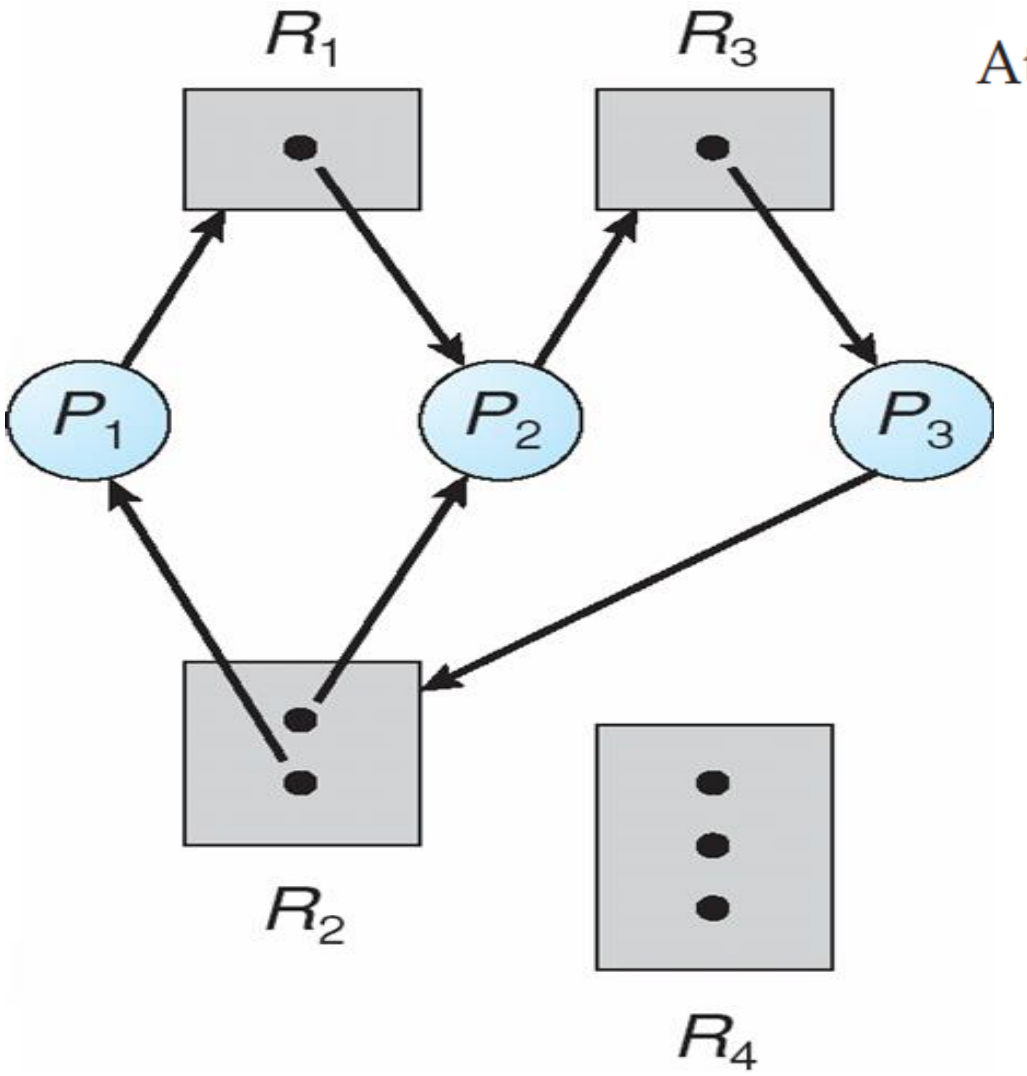


The sets P, R, and E:

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$



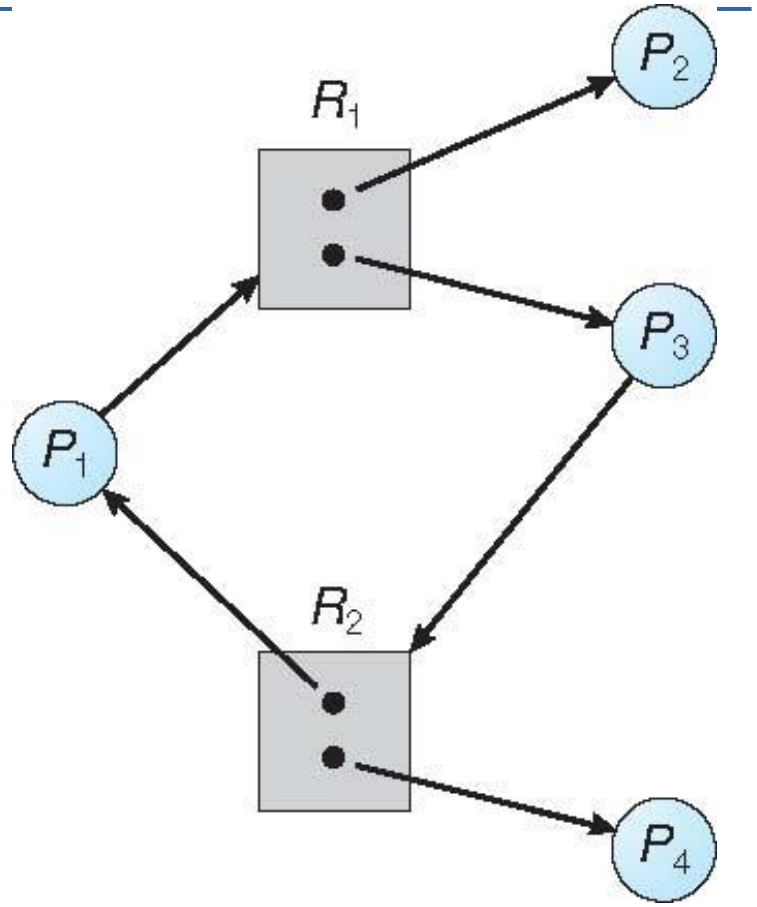
Resource Allocation Graph With A Deadlock



At this point, two minimal cycles exist in the system:

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

Graph With A Cycle But No Deadlock



$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

we also have a cycle

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

Basic Facts

- If graph contains **no cycles** ⇒ **no deadlock**
- If graph **contains a cycle** ⇒
  - if only **one instance per resource** type, then **deadlock**
  - **if several** instances per resource type, **possibility** of deadlock

BREAK!

RECAP QUESTIONS

1. Name necessary conditions for deadlock.
2. Multiple instances per resource possible (Yes/No)?
3. If a graph contains a cycle, and if only one instance per resource type, then does a deadlock occur (Yes/No)?
4. Name methods for handling deadlock.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - **Deadlock prevention**
  - **Deadlock avoidance**
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Used by most operating systems, including Linux and Windows; Up to the application developer to write programs that handle deadlock

Deadlock Prevention

Set of methods to ensure that at least one of the necessary conditions **cannot hold**

Restrain the ways request can be made

Remove

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **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 will be *restarted only when it can regain its old* resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a **total ordering of all resource types**, and require that each process requests resources in an increasing order of enumeration

Deadlock Prevention (Circular wait) Example.

- Let R = {R1, R2, ..., Rm} be the set of resource types.
- Assign to **each resource type** a **unique integer num**ber, which allows us to *compare two resources* and to determine whether one precedes another in ordering.
- Formally, we define a one-to-one function F: R → N, where N is the set of natural numbers.
  - For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F is defined as
    - F(tape drive) = 1
    - F(disk drive) = 5
    - F(printer) = 12

Deadlock Prevention (Circular wait) Example.

- Consider the following protocol to prevent deadlocks:
  - Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say, Ri . After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri).
  - For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

Deadlock Prevention (Circular wait) Example.

- Consider the following protocol to prevent deadlocks:
  - Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj).
  - Note also that if several instances of the same resource type are needed, a ***single request*** for all of them must be issued.
  - If these two protocols are used, then the circular-wait condition cannot hold.

Deadlock Example

```
/* thread one runs in this function */
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);
}
/* thread two runs in this function */
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);
}
```

Deadlock Example with Lock Ordering

```
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);
}
```

Transactions 1 and 2 execute concurrently.

Transaction 1 transfers $25 from account A to account B

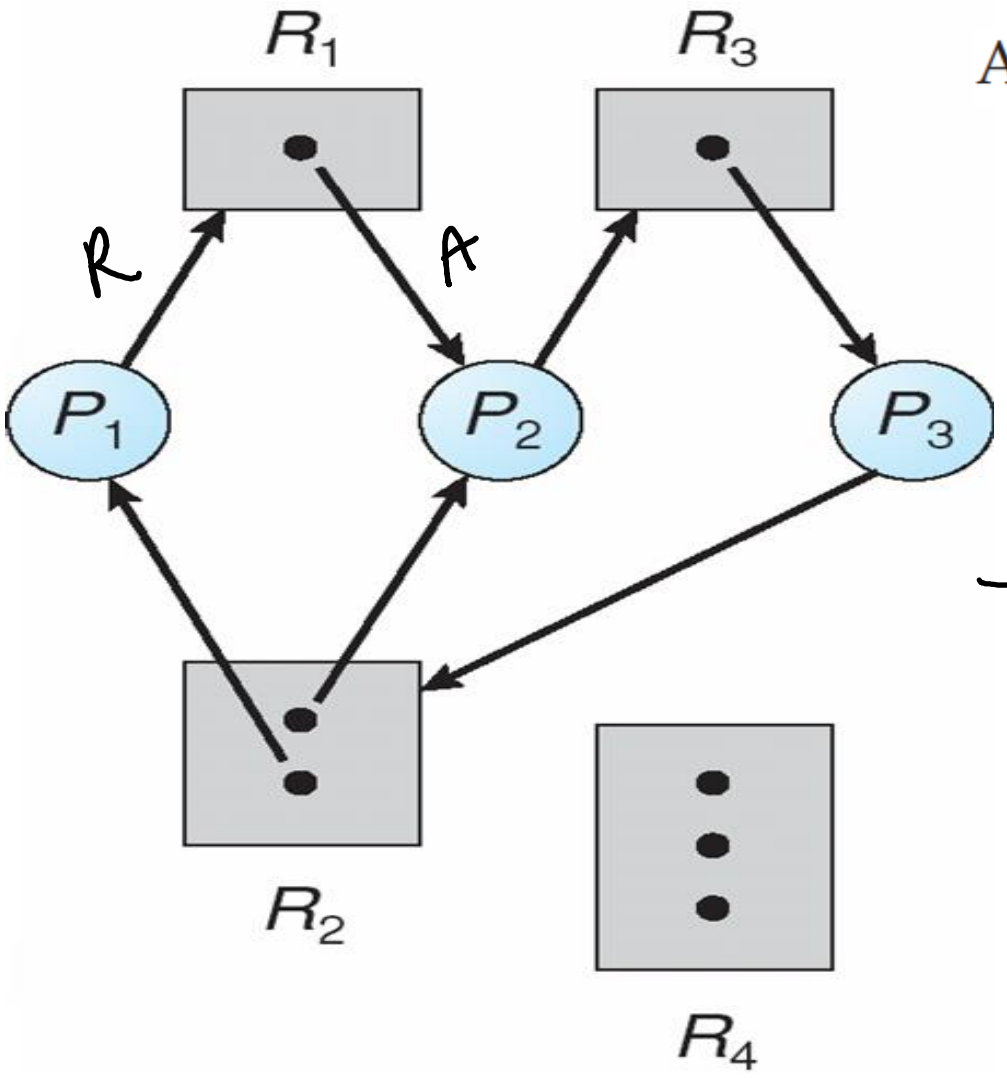
Transaction 2 transfers $50 from account B to account A.

Deadlock Avoidance

Requires that system has some additional ***a priori*** information available. The OS decides for each request whether or not the process should wait

- Simplest and most useful model requires that each process declare the ***maximum number of resources*** of each type that it may need
- 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:
  1. Number of available and allocated resources, and
  2. Maximum demands of the processes

Resource Allocation Graph With A Deadlock



At this point, two minimal cycles exist in the system:

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

Availability

Allocation:

	R_1	R_2	R_3
P_1	0	1	0
P_2	1	1	0
P_3	0	0	1

Need

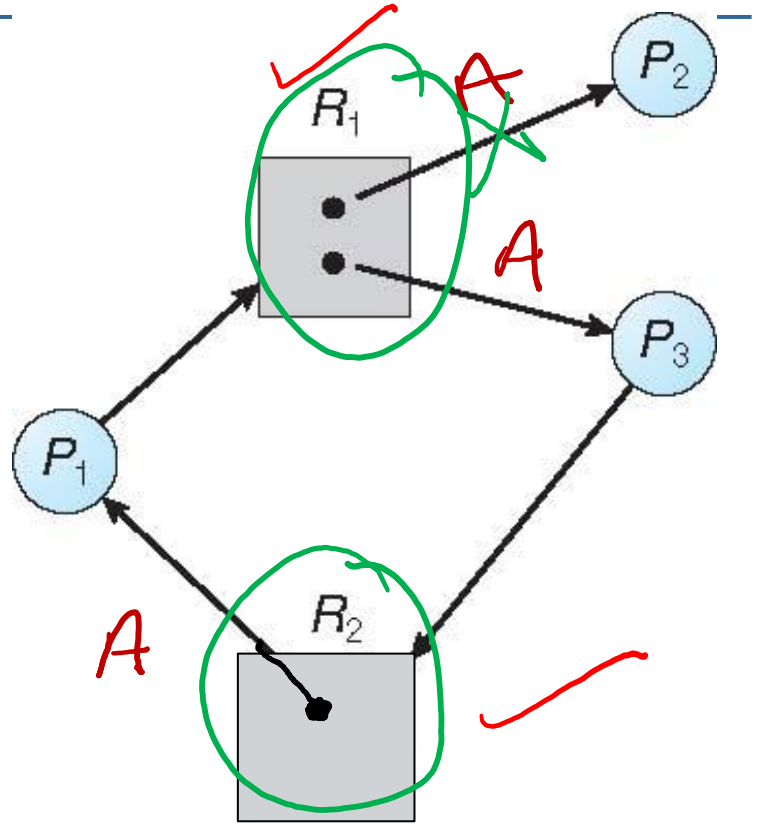
	R_1	R_2	R_3
P_1	1	0	0
P_2	0	0	1
P_3	0	1	0

R_1	R_2	R_3
0	0	0

Graph With A Cycle But No Deadlock

	Allocation		Need		Available	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	0	1	1	0	0	0
P_2	1	0	0	0	1	0
P_3	1	0	0	1	1	1
	1	1			2	1

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$



we also have a cycle

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

BREAK.

Avoidance Algorithms

- **Single instance** of a resource type
  - Use a **resource-allocation graph**
    - System resources
- **Multiple instances** of a resource type
  - Use the **banker's algorithm**

Safe State

- When a process requests an available resource, system must decide if **immediate allocation** leaves the system in a **safe state.**
- Which means
  - System is in **safe state** if there exists a sequence

    <P1, P2, …, Pn>

  - **of ALL the processes** in the systems such that for each Pi
  - Resources that **Pi** can still **request** *can be satisfied* by currently available resources **+** resources held by all the Pj, with j < i

Safe State

- System is in **safe state** if there exists a sequence <P1, P2, …, Pn>
- That is:
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
  - When Pi terminates, Pi +1 can obtain needed resources, and so on.

Basic Facts

- 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.

Example.

$\langle P_1, P_0, P_2 \rangle$

- Consider a system:
  - **Magnetic tape drives: 12**
  - **Processes**: **3 (P0, P1, P2)**
    - P0 requires 10
    - P1 may need 4,
    - P2 may need 9

MAX

How many free tape drives? 3

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

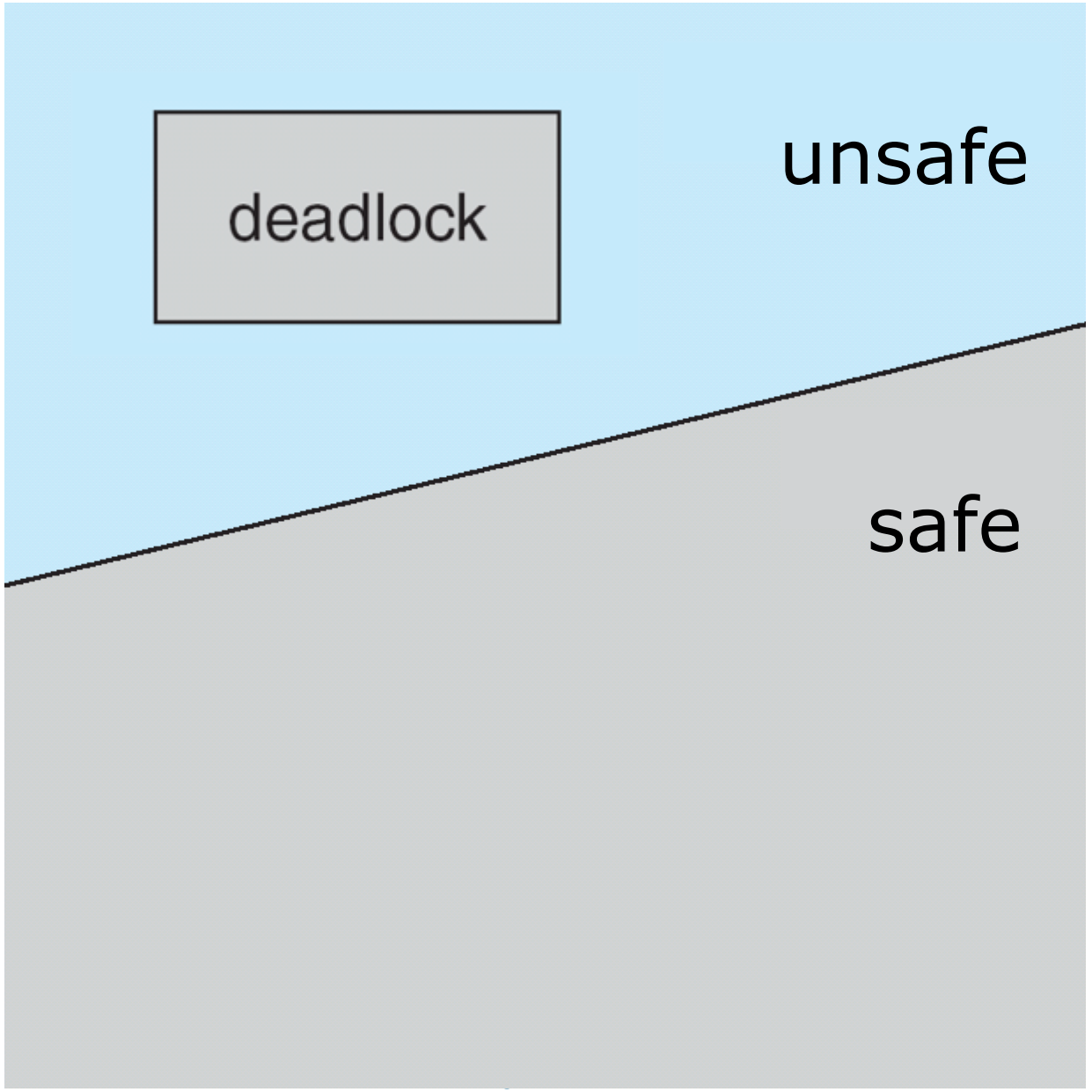
- Suppose at time t0
  - P0 is holding 5
  - P1 is holding 2
  - P2 is holding 2

Current

9

P_1	2	3	5
P_0	5	5	10
P_2	2	10	12

Safe, Unsafe, Deadlock State

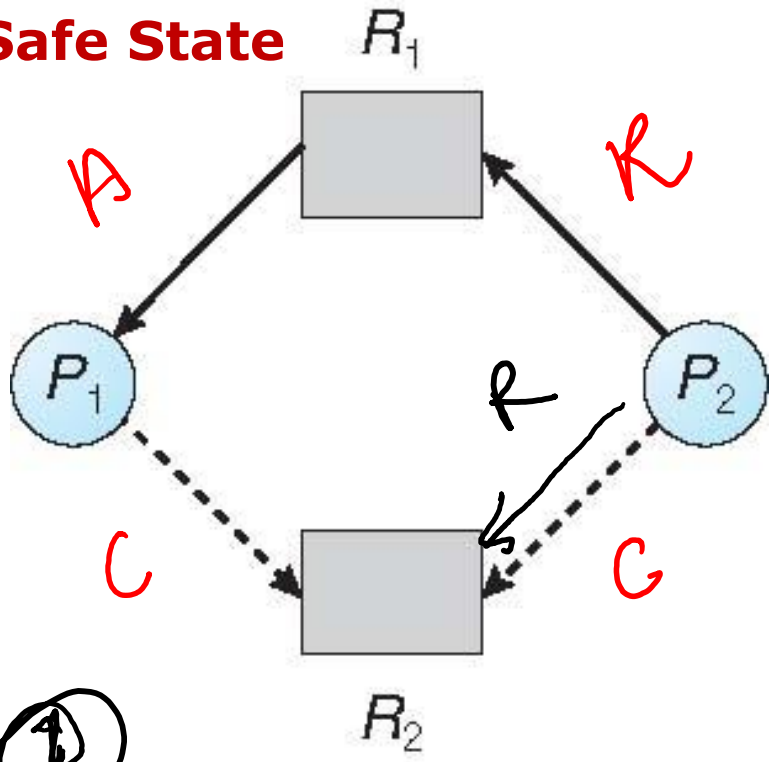


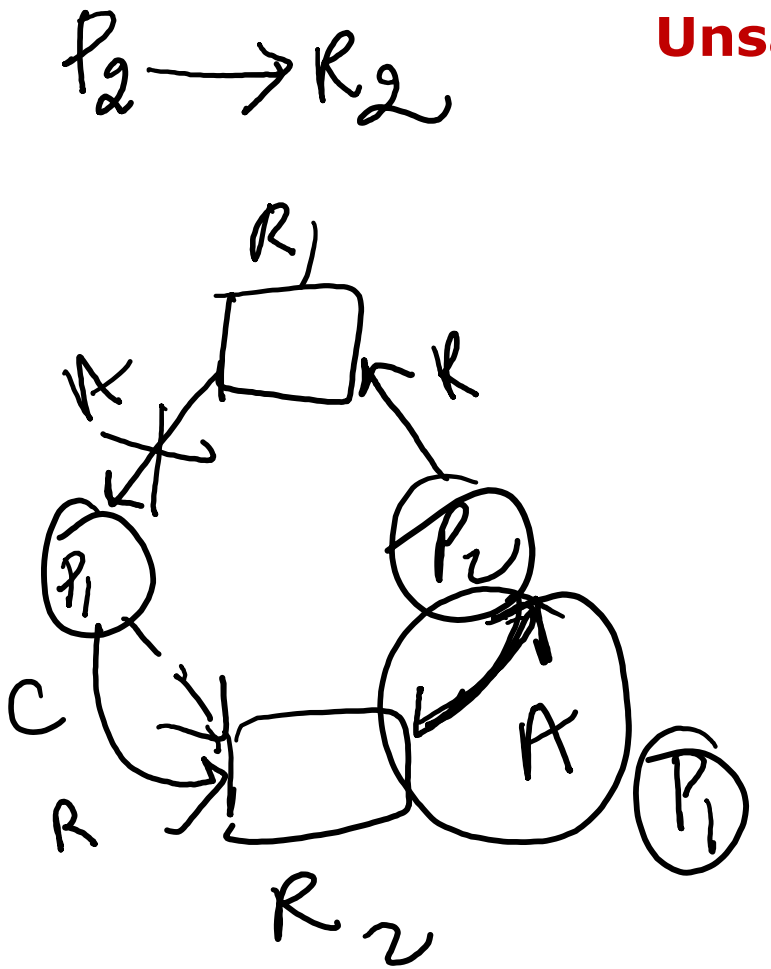
Resource-Allocation Graph Scheme

- **Claim edge $P_i \rightarrow R_j$** indicated that process P_j may request resource R_j; represented by a dashed line
- **Claim edge converts to request edge** when *a process **requests** a resource*
- **Request edge converted to an assignment edge** when *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 a priori in the system**

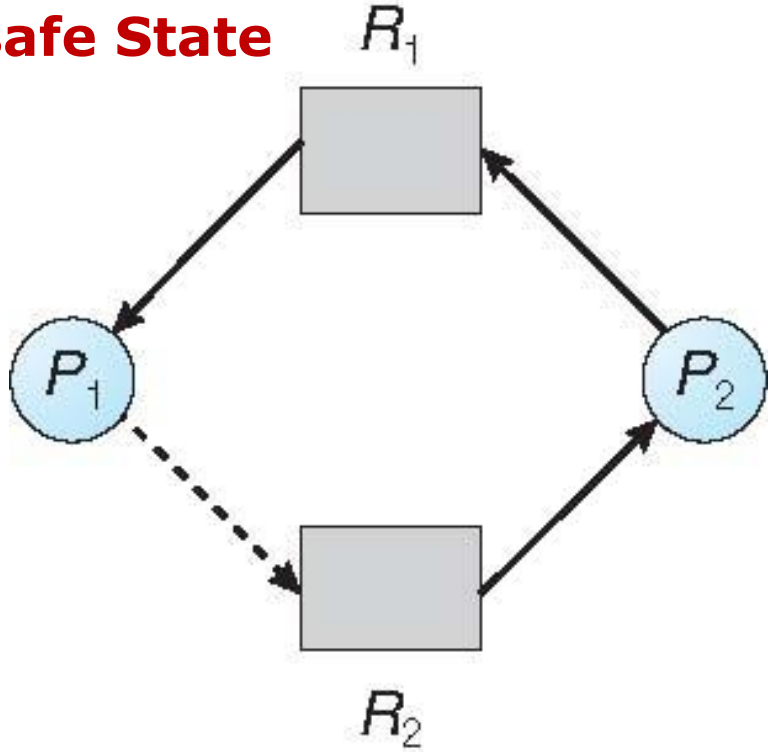
Resource-Allocation Graph

Safe State





Unsafe State



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The **request can be granted** only if ***converting the request edge to an assignment edge does not result in the formation of a cycle*** in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a **priori claim maximum** use
- When a **process requests a resource** it may have to **wait**
- When a process gets all its resources it must **return** them in a **finite** amount of **time**

Data Structures for the Banker's Algorithm

Let ***n*** = number of processes, and *m* = number of resources types.

- **Available***:* Vector of length ***m*****.** If available [*j*] = *k*, there are ***k*** instances of resource type R_j available
- **Max***: **n x m matrix.*** If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j
- **Allocation***:* ***n x m matrix***. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j
- **Need***:* ***n x m matrix***. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

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

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $Need_i \leq Work$

   If no such ***i*** exists, go to step 4

3. $Work = Work + Allocation_i$

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Order of $m \times n^2$ operations to determine whether a state is safe

Example.

- To illustrate, consider a system with **12 magnetic tape drives** and
- **3 processes**: P0, P1, and P2.
- Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives.
- Suppose that, at time t0, process P0 is holding five tape drives, process
- P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

Example of Banker's Algorithm

- 5 processes P_0 through P_4 and 3 resource types:

  A (**10** instances), B (**5** instances), and C (**7** instances)

- Snapshot at time T_0:

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

Need is defined to be ***Max – Allocation***

Finish[i] = false $Need_i \leq Work$

Need Matrix

P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

F 7 4 3 $\not\leq$ 3 3 2 X

F 1 2 2 $\leq$ 3 3 2 ✓ → 3 3 2 + 2 0 0 = 5 3 2 ||T

F 6 0 0 $\not\leq$ 5 3 2 X ||F

F 0 1 1 $\leq$ 5 3 2 ✓ → 2 1 1 + 5 3 2 = 7 4 3 ||T

F 4 3 1 $\leq$ 7 4 3 ✓ → 0 0 2 + 7 4 3 = 7 4 5 ||T

$<P_1, P_3, P_4>$ ✓

Iteration 1 completes

Need Matrix

	A	B	C	
P_0	7	4	3	✗
P_1	1	2	2	✓
P_2	6	0	0	✗
P_3	0	1	1	✓
P_4	4	3	1	✓

At the end of the previous iteration Sequence was $< P_1, P_3, P_4$ and Available

A	B	C
7	4	5

Take,

$P_0 \Rightarrow 743 \leq 745$ ✓ ⇒ 010 (Allocation)

7	5	5

P_0

Now,

$P_2 \Rightarrow 600 \leq 755$ ✓ ⇒ 302

10	5	7

No. of instance available

A B C

$< P_1, P_3, P_4, P_0, P_2 >$ ✓

The system is in a **safe state** since the sequence ***< P_1, P_3, P_4, P_2, P_0> satisfies safety criteria***

BREAK

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$.

If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \le Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \le Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$
$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$
$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Determining whether requests can be safelygranted

Request Resource Algorithm

[Q: Can this request be granted?]

Use

To

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	~~3 3 2~~	7 4 3
				0 2 0
P_1 ✓	2 0 0 → 3 0 2	3 2 2	(WORK)	~~1 2 2~~
			2 3 0	
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Request P_1 (1, 0, 2) (A, C)

↓

Check conditions

↓

1) $R \leq N$? $102 \leq 122$ ✓

2) $R \leq W$ $102 \leq 332$ ✓

Allocation	Available	Need
200	332	122
+ 102	− 102	− 102
302	230	020

(Request)

Now, check this snapshot is safe → use safety Algorithm

[Process P_1 requests +1 → A, +2 → C]

A	B	C
10	5	7

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available, that is, (1,0,2) ≤ (3,3,2) ⇒ true

(P_1)

	Allocation	*Need*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Yes, request (1,0,2) for P_1 can be granted

- Executing safety algorithm shows that sequence **< P_1, P_3, P_4, P_0, P_2>** satisfies safety requirement

Example: P_4 Request (1,0,2)

- Can request for (3,3,0) by P_4 be granted?

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Check

Request ≤ Need

3,3,0 ≤ 4,3,1 ?

Request ≤ Available?

3,3,0 ≤ 2,3,0

NO

So, request (3,3,0) cannot be granted.

Example: P_1 ~~Request (1,0,2)~~

HW

- Can request for (0,2,0) by P0 be granted?

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Check

↓

$Request_0 \leq Need_0$

~~$0,2,0 \leq 7,4,3$~~ ✓

$Request_0 \leq Available$

$0,2,0 \leq 2,3,0$ ✓

If safe → grant request

If not safe → do not grant

Check for safe state.

↳ Use safety algorithm

↓

Safe sequence

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Deadlock Detection

Single instance resource

Wait-for graph

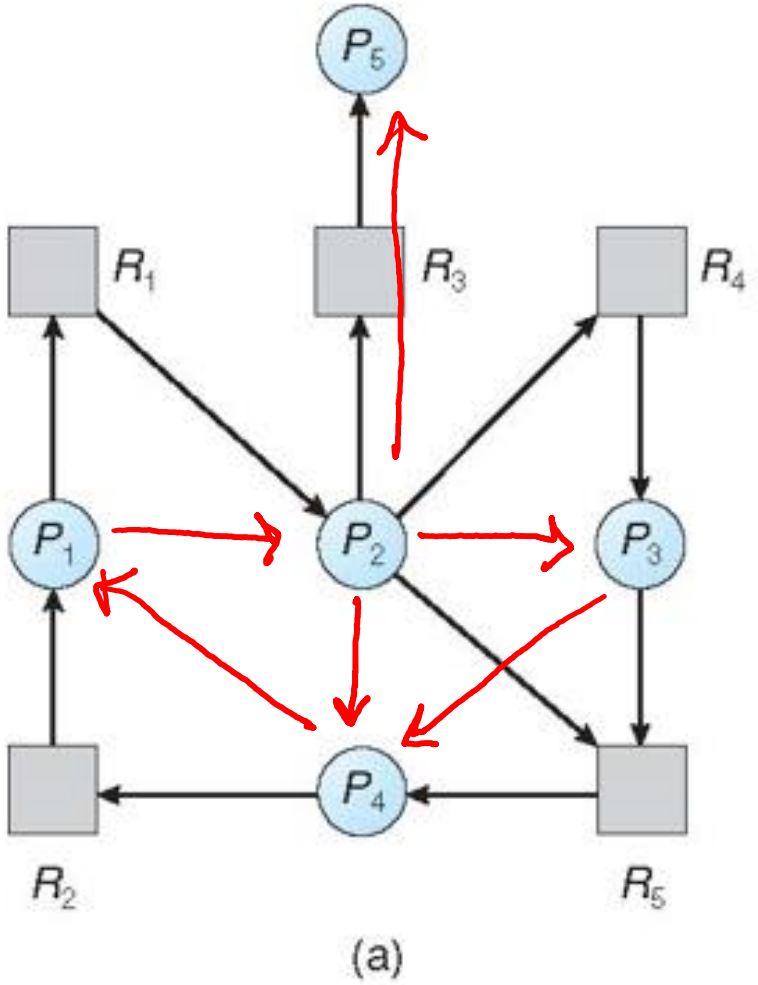
Multiple instances resource

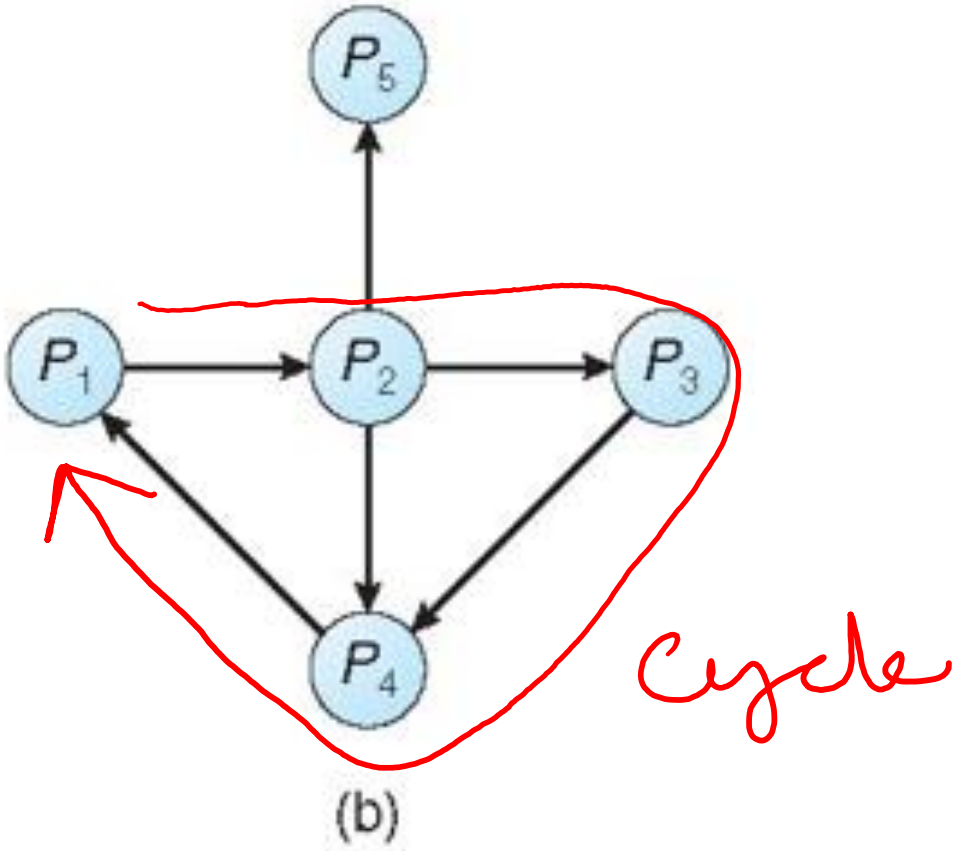
Banker's algorithm

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process
- **Request**: An ***n*** **x** ***m*** matrix indicates the current request of each process. If ***Request*** **[*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Need

A B C

P_1 → 1 0 1

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available***. For $i = 0, 1, ..., n-1$, if ***Allocation***$_i \neq 0$, then ***Finish***$[i]$ = ***false***. Otherwise, ***Finish***$[i]$ = ***true***.

2. Find an index i such that both

   a. ***Finish***$[i]$ == ***false***

   b. ***Request***$_i \leq$ ***Work***

   If no such i exists, go to step 4.

3. ***Work*** = ***Work*** + ***Allocation***$_i$
   Finish$[i]$ = ***true***
   Go to step 2.

4. If ***Finish***$[i]$ == ***false*** for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***$[i]$ == ***false***, then process P_i is deadlocked.

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

No deadlock

- Five processes P0 through P4; three resource types: A (7 instances), B (2 instances), and C (6 instances)

A (7) B (2) C (6)

- Snapshot at time T0:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	0 1 0
P_2	3 0 3	0 0 0	3 1 3
P_3	2 1 1	1 0 0	5 2 4
P_4	0 0 2	0 0 2	5 2 6
	7 2 6		7 2 6

= Work

1) Request ≤ Work W = W + Allocation

P_0 000 ≤ 000 (T) 000 + 010

P_1 202 ≤ 010 (F)

P_2 000 ≤ 010 (T) 010 + 303

P_3 100 ≤ 313 (T) 313 + 211

P_4 002 ≤ 524 (T) 524 + 002

→ P_1 202 ≤ 526 T 526 + 200

<P_0, P_2, P_3, P_4, P_1>

results in Finish[i] = true for all i

Example (Cont.)

A B C
7 2 6

- P_2 requests an additional instance of type ***C***

T_1

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1 +1
P_3	1 0 0
P_4	0 0 2

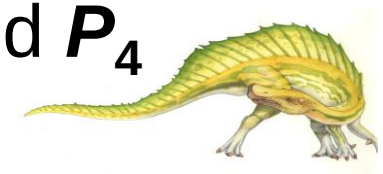
Apply Banker's Algorithm

A B C
0 0 0
0 1 0

- State of system?

$\langle P_1, P_2, P_3, P_4 \rangle$ in deadlock

  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4



Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

OPTIMISTIC approach

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

cost factors

Pessimistic approach

End of Chapter 7