CHAPTER 8 - DEADLOCKS

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 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 IN MULTITHREADED APPLICATIONS

Lets see how deadlocks can happen

DEADLOCK WITH PTHREADS

Recap

```
// Creates unlocked mutex
pthread_mutex_init()

// Locks a mutex
pthread_mutex_lock()

// Unlocks mutex
pthread_mutex_unlock()
```

DEADLOCK WITH PTHREADS

Initialization

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;


pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

DEADLOCK WITH PTHREADS

Thread 1

```
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);
}
```

DEADLOCK WITH PTHREADS

Thread 2

```
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 WITH PTHREADS

Deadlock does not always occur!

LIVELOCK

- Another form of liveness failure
- Livelock is similar to what sometimes happens when two people attempt to pass in a hallway
  - One moves to his right, the other to her left, still obstructing each other's progress.
  - Both move to the other side
  - They aren't blocked, but they aren't making any progress.

LIVELOCK

Thread 1

```
void *do_work_one(void *param) {

     int done = 0;

     while (!done) {
          pthread_mutex_lock(&first_mutex);
          if (pthread_mutex_trylock(&second_mutex)) {

               /* Do some work */
               pthread_mutex_unlock(&second_mutex);
               pthread_mutex_unlock(&first_mutex);
               done = 1;
          } else {
               pthread_mutex_unlock(&first_mutex);
      }
     }
     pthread_exit(0);
}
```

LIVELOCK

Thread 2

```
void *do_work_two(void *param) {

     int done = 0;

     while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {

                /* Do some work */

                pthread_mutex_unlock(&first_mutex);
                pthread_mutex_unlock(&second_mutex);
                done = 1;
        } else {
                pthread_mutex_unlock(&second_mutex);
     }
     }

     pthread_exit(0);
}
```

LIVELOCK NOTES

- Typically occurs when threads retry failing operations at the same time
- Generally be avoided by having each thread retry the failing operation at random times.
  - Exponential backoff in Ethernet networks
- Less common than deadlocks

DEADLOCK CHARACTERIZATION

DEADLOCK CHARACTERIZATION

Deadlock can arise if four conditions hold simultaneously.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

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 $\{P_0, P_1, \ldots, P_n\}$ 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.

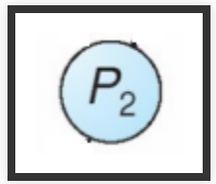
DEADLOCK WITH MUTEX LOCKS

```c
/* 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);
}
```

RESOURCE-ALLOCATION GRAPH

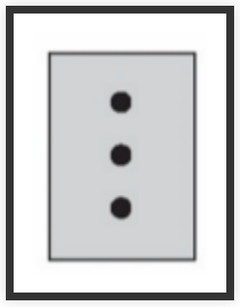
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, ..., R_n\}$, the set consisting of all resource types in the system
  - **request edge** – directed edge $P_i \rightarrow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

LEGEND



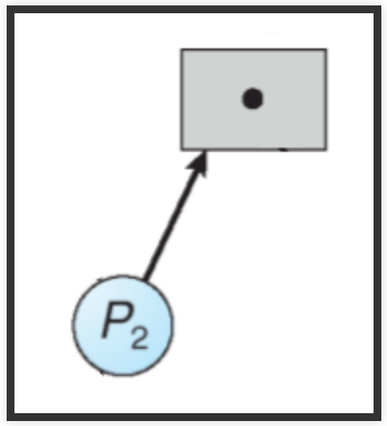
Process

LEGEND



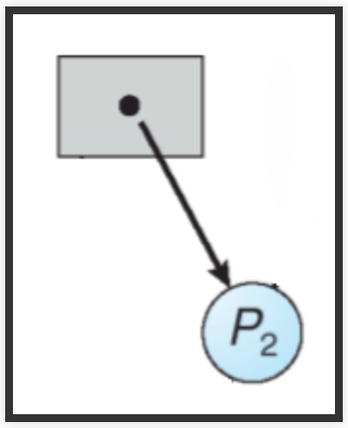
Resource Type with 3 instances

LEGEND



P_2 requests instance of R

LEGEND

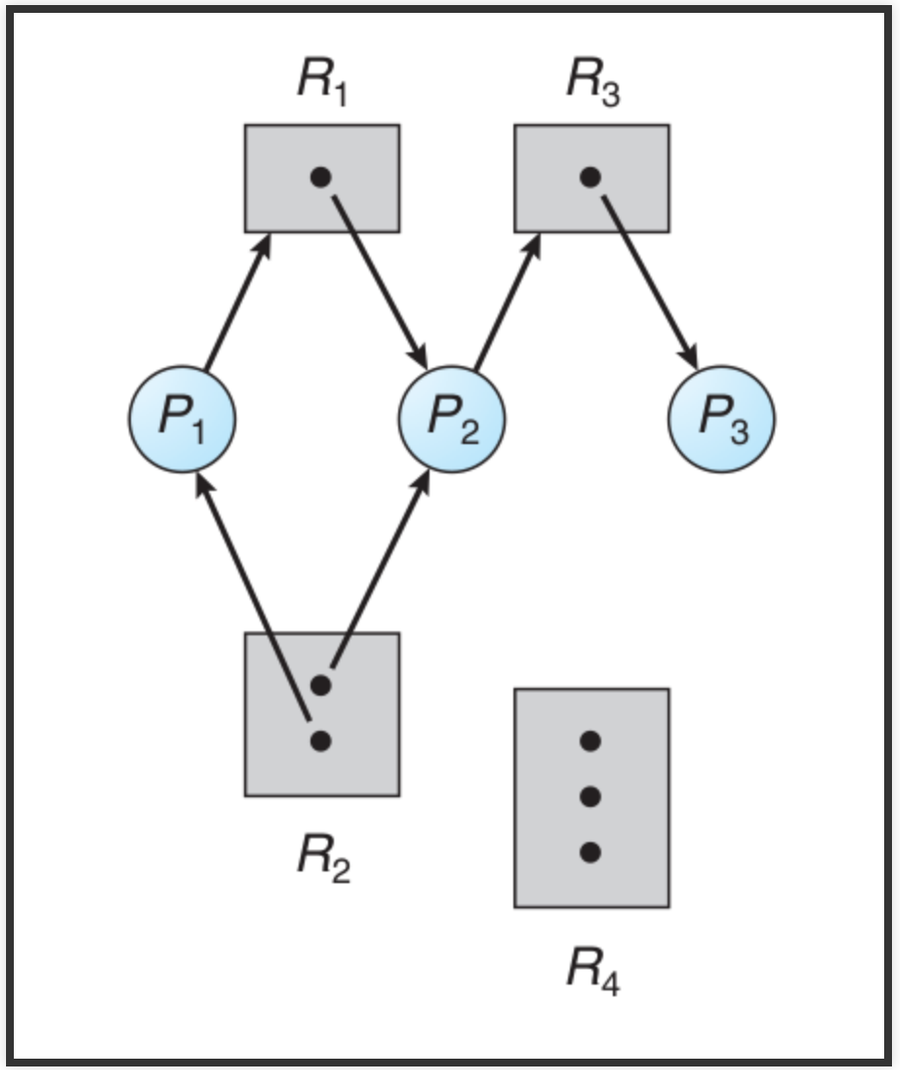


P_2 is holding an instance of R

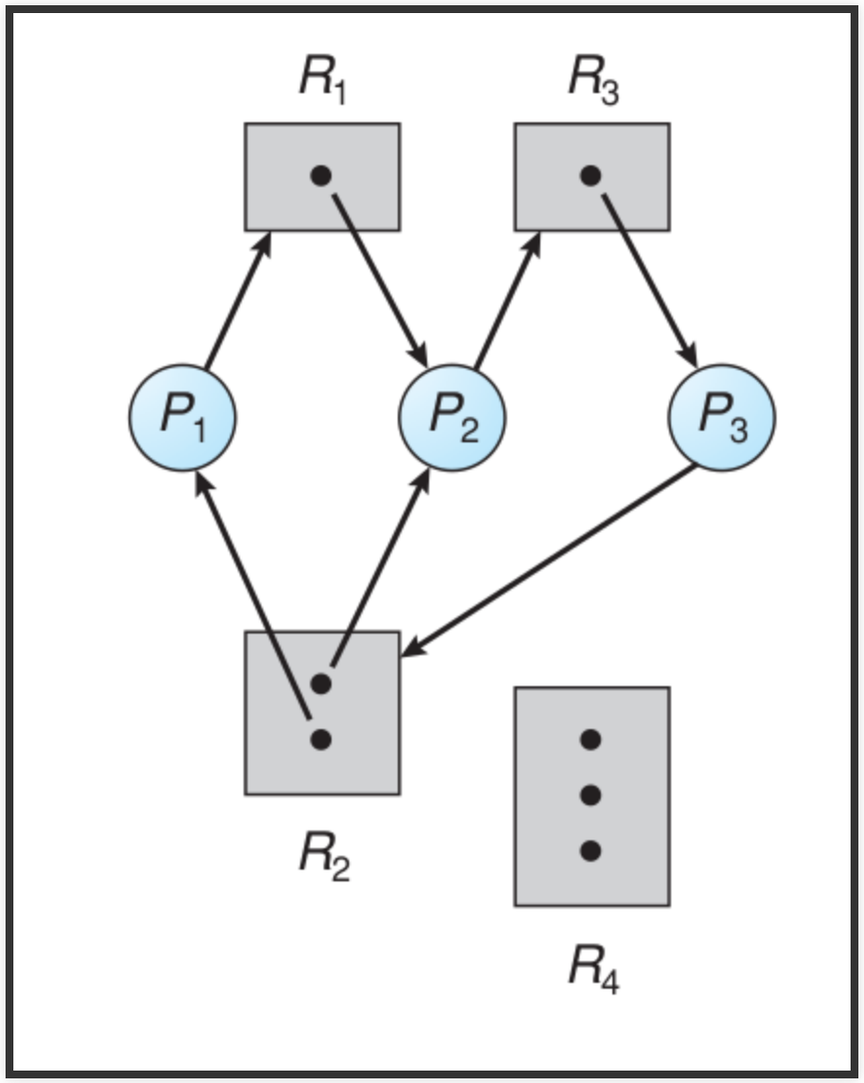
EXAMPLE RESOURCES

- $P = \{ P_1, P_2, P_3 \}$
- $R = \{ R_1, R_2, R_3, R_4 \}$
- $E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow p_1, R_3 \rightarrow P_3 \}$

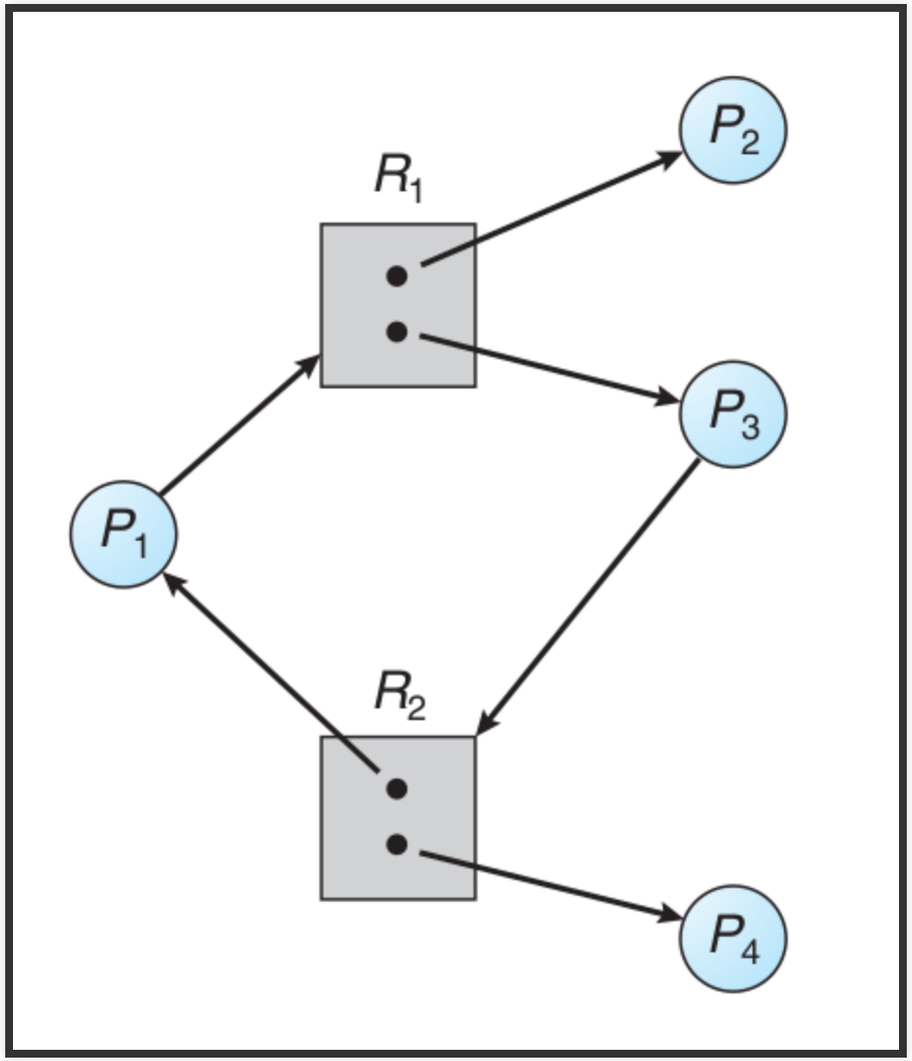
RESOURCE ALLOCATION GRAPH



GRAPH WITH A DEADLOCK



WITH A CYCLE BUT NO DEADLOCK



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

METHODS FOR HANDLING DEADLOCKS

METHODS FOR HANDLING DEADLOCKS

1. Ensure that the system will never enter a deadlock state
2. Allow the system to enter a deadlock state and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems

DEADLOCK PREVENTION

DEADLOCK PREVENTION

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

DEADLOCK PREVENTION

Hold and Wait – multiple ways

1. must guarantee that whenever a process requests a resource, it does not hold any other resources.
2. Request all resources up front before execution. (all or none)

DEADLOCK PREVENTION

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

DEADLOCK PREVENTION

Circular Wait

impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

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);
}
```

DEADLOCK AVOIDANCE

DEADLOCK AVOIDANCE

Requires that the system has some additional a priori information available

Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need

DEADLOCK AVOIDANCE

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

SAFE STATE

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

System is in safe state if there exists a **safe sequence**

$$<P_1, P_2, \ldots, P_n>$$

of ALL the processes in the systems 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 $j < i$

SAFE STATE

That is:

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on

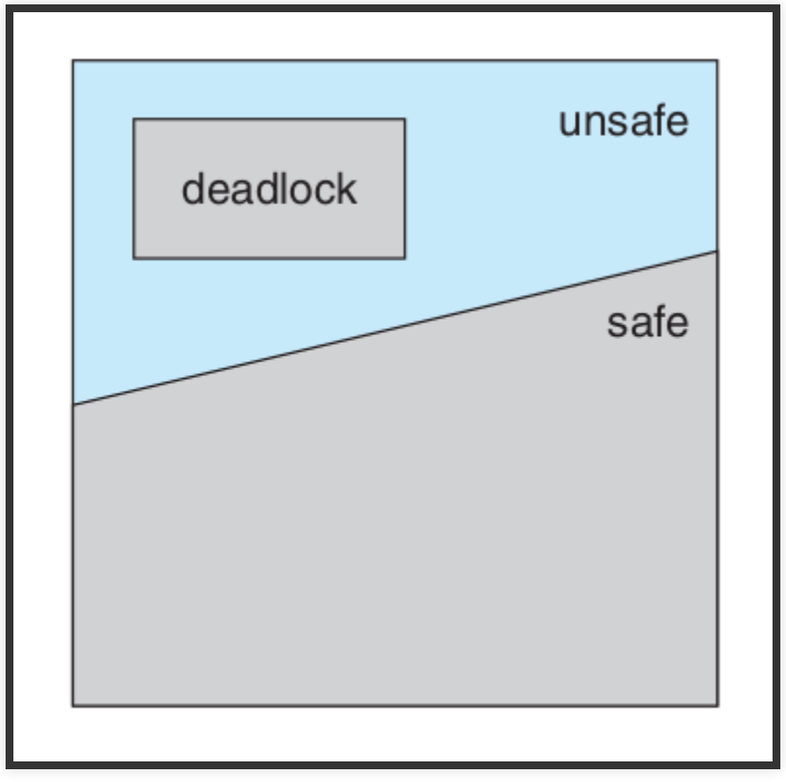
BASIC FACTS

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.

SAFE, UNSAFE, DEADLOCK STATE



AVOIDANCE ALGORITHMS

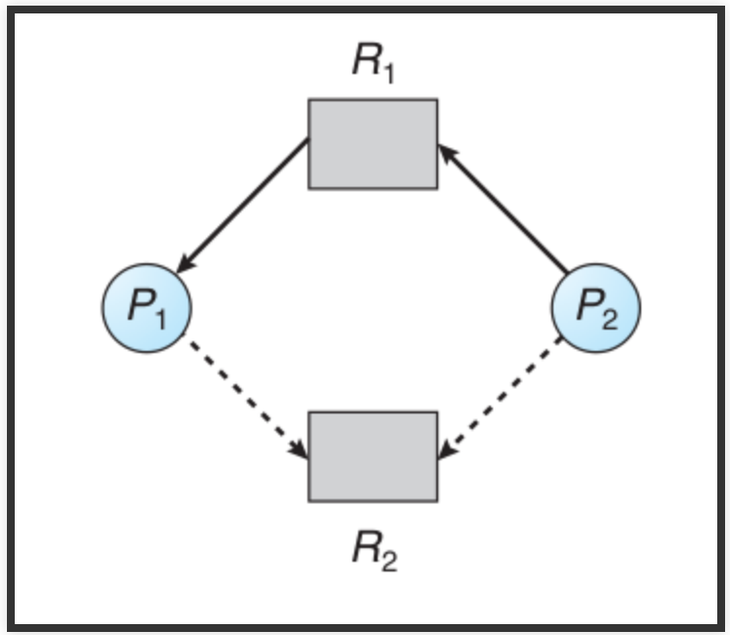
Single instance of a resource type → Use a resource-allocation graph

Multiple instances of a resource type → Use the banker's algorithm

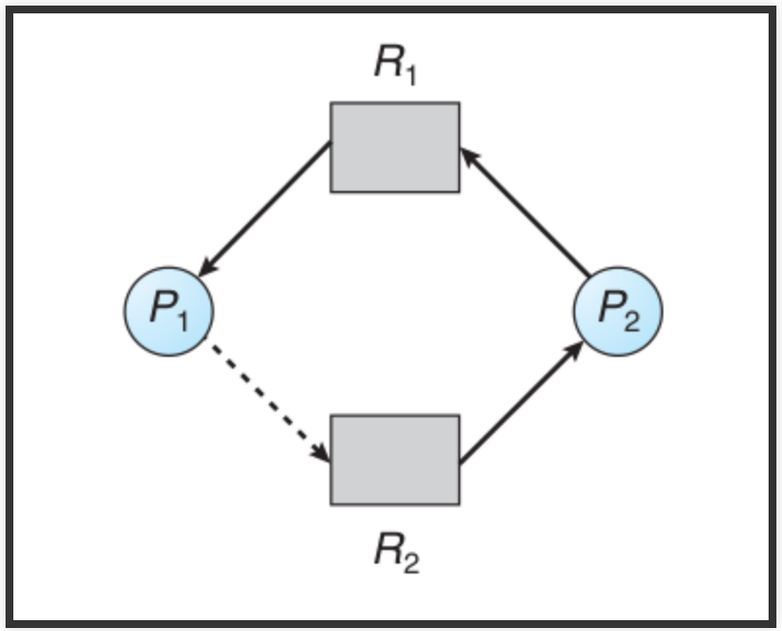
SCHEME

- Claim edge $P_i \rightarrow R_j$ indicated that process P_i 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 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 a priori in the system

RESOURCE-ALLOCATION GRAPH



UNSAFE STATE IN RESOURCE-ALLOCATION GRAPH



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

BANKER'S ALGORITHM SETUP

Let n = number of processes,

Let m = number of resources types.

DATA STRUCTURES NEEDED

- **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

SAFETY ALGORITHM 2

Find an `i` such that both:

1. `Finish[i] = false`
2. $Need_i \leq Work$

If no such i exists, go to step 4

SAFETY ALGORITHM 3

- `Work = Work + Allocation`$_i$
- `Finish[i] = true`
- go to step 2

SAFETY ALGORITHM 4

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

RESOURCE-REQUEST ALGORITHM

Let `Request`$_i$ be request vector for for Process P_i

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

When resource request made by P_i:

RESOURCE-REQUEST ALGORITHM

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

RESOURCE-REQUEST ALGORITHM

If safe → the resources are allocated to P_i

If unsafe → P_i must wait, and the old resource-allocation state is restored

EXAMPLE OF BANKER'S ALGORITHM

5 processes P_0 through P_4

3 resource types:

- A (10 instances)
- B (5 instances)
- C (7 instances)

EXAMPLE OF BANKER'S ALGORITHM

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

EXAMPLE OF BANKER'S ALGORITHM

	Need
	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

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

EXAMPLE OF BANKER'S ALGORITHM

Now a request from P_1 comes for (1,0,2)

Check first that

$$Request_i \leq Available$$

EXAMPLE OF BANKER'S ALGORITHM

	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	

<P_1,P_3,P_4,P_0,P_2> satisfies safety criteria

EXERCISE

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

DEADLOCK DETECTION

DEADLOCK DETECTION

Allow system to enter deadlock state

Detection algorithm

Recovery scheme

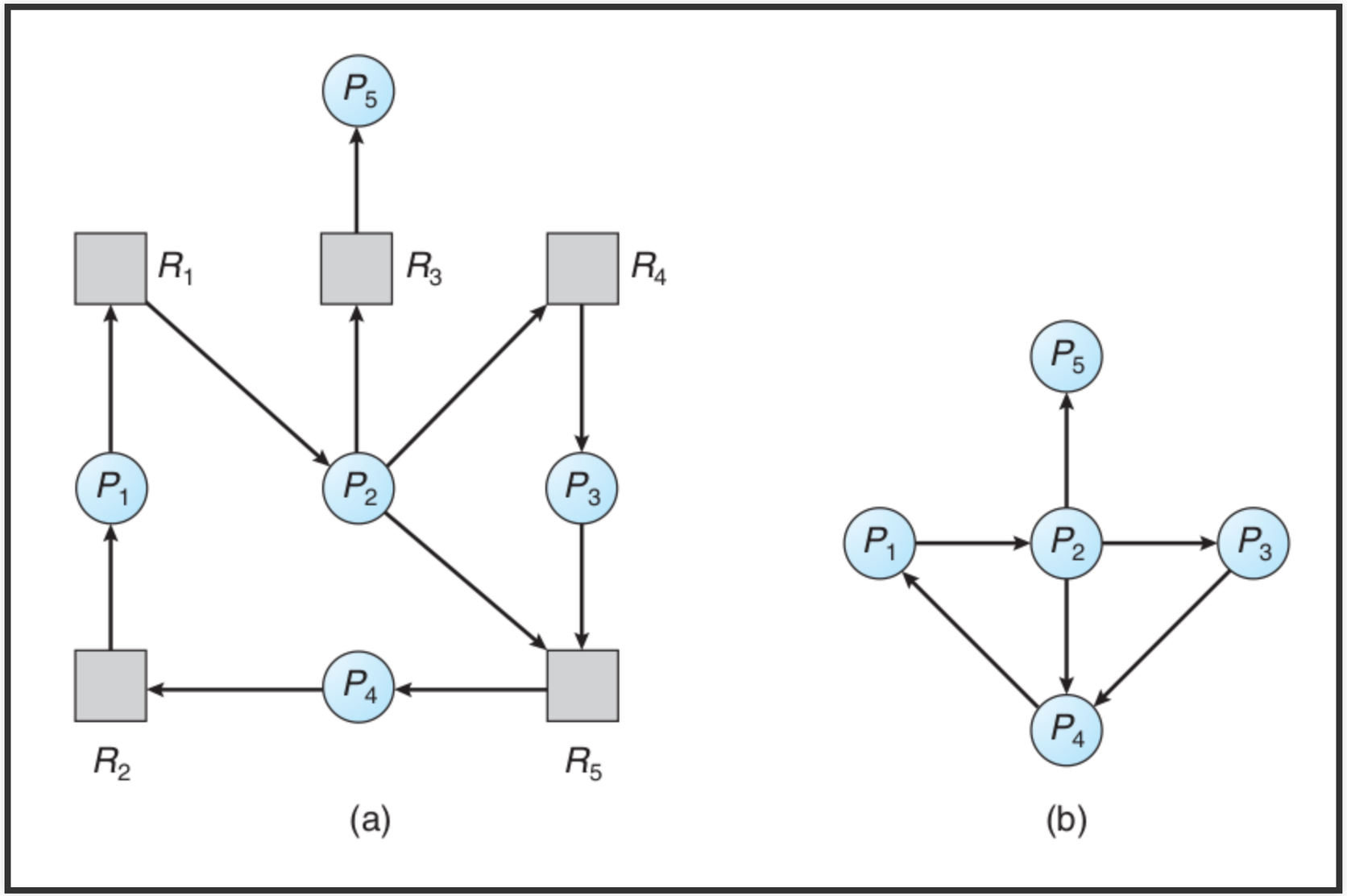
SINGLE INSTANCE OF EACH RESOURCE TYPE

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 O(n+m) operations, where n is the number of vertices in the graph and m is the number of edges

GRAPHS



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

DETECTION ALGORITHM 1

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

Initialize:

- `Work = Available`
- For i = 1,2, …, n, if $\texttt{Allocation}_i \neq 0$, then `Finish[i] = false`; otherwise, `Finish[i] = true`

DETECTION ALGORITHM 2

Find an index i such that both:

- `Finish[i] == false`
- `Request i ≤ Work`

If no such i exists, go to step 4

DETECTION ALGORITHM 3

- `Work = Work +` $\texttt{Allocation}_i$
- `Finish[i] = true`
- go to step 2

DETECTION ALGORITHM 4

- If `Finish[i] == false`, for some i, $1 \leq i \leq n$, then the system is in deadlock state.
- Moreover, if `Finish[i] == false`, then P_i is deadlocked
- Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

EXAMPLE

- Five processes P_0 through P_4
- Three resource types
  - A (7 instances)
  - B (2 instances)
  - C (6 instances)

EXAMPLE

Snapshot at time T_0

	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	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence <P_0, P_2, P_3, P_1, P_4> will result in `Finish[i] = true` for all i

EXAMPLE

P_2 requests an additional instance of type C

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

- State of system?

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

PROCESS TERMINATION

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated

PROCESS TERMINATION

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?

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

QUESTIONS

BONUS

Exam question number 5: **Deadlocks**