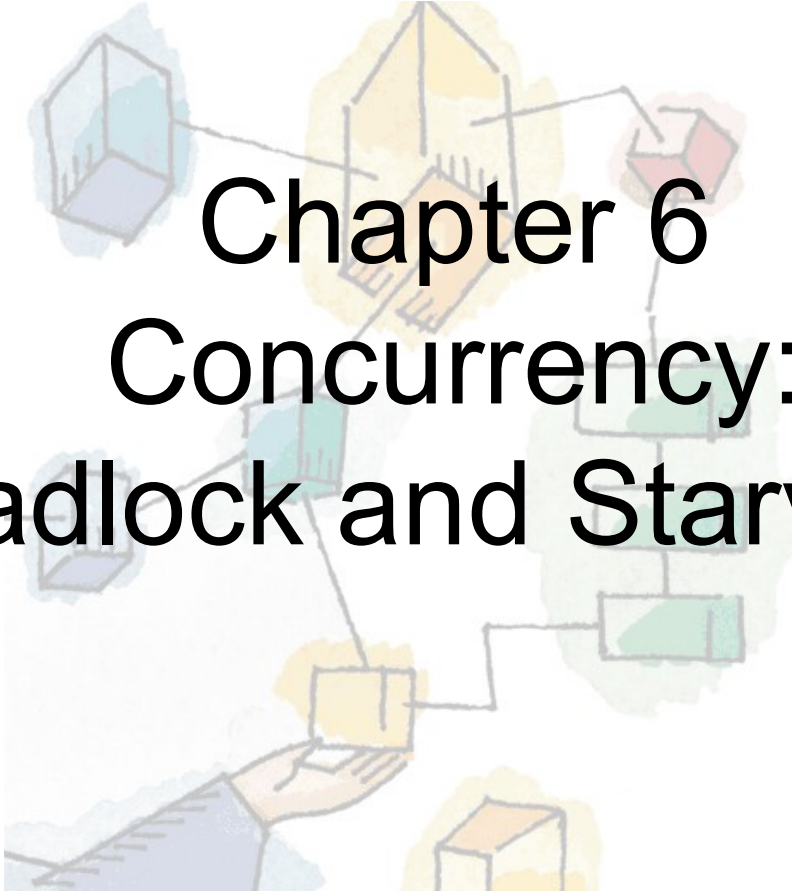
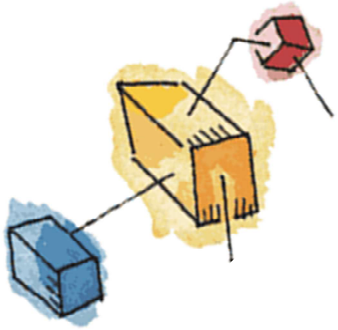


*Operating Systems:
Internals and Design Principles*
William Stallings



Chapter 6 Concurrency: Deadlock and Starvation

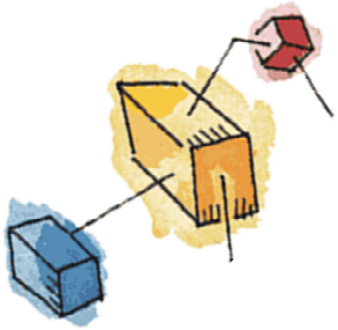


Roadmap

→ Principles of Deadlock

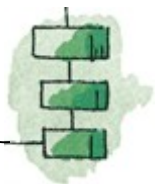
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Dining Philosophers Problem

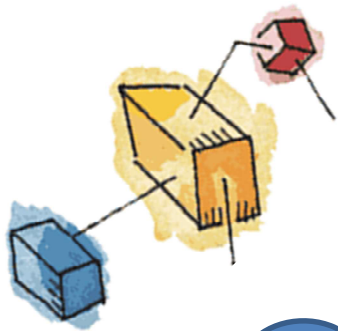




Deadlock

- **Deadlock** is the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other.
- A set of processes is **deadlocked** when each process in the set is blocked awaiting an *event* that can only be triggered by another blocked process in the set.
 - The event is typically the freeing up of some requested and obtained resources
- No efficient solution in the general case.



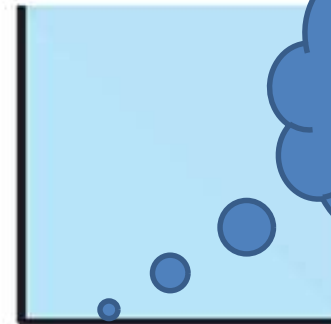


Potential Deadlock

I need
quad c
and d



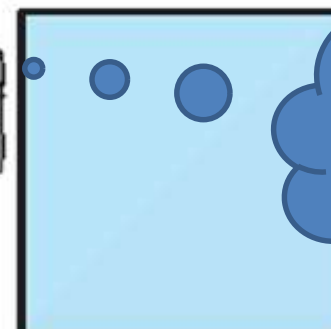
c



b



d

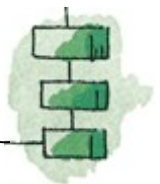


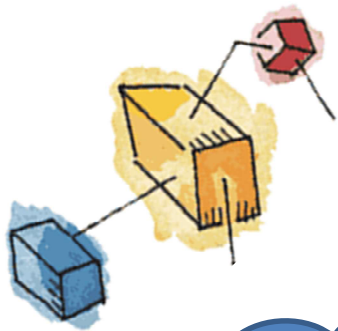
a

I need
quad d
and a

I need
quad a
and b

I need
quad b
and c





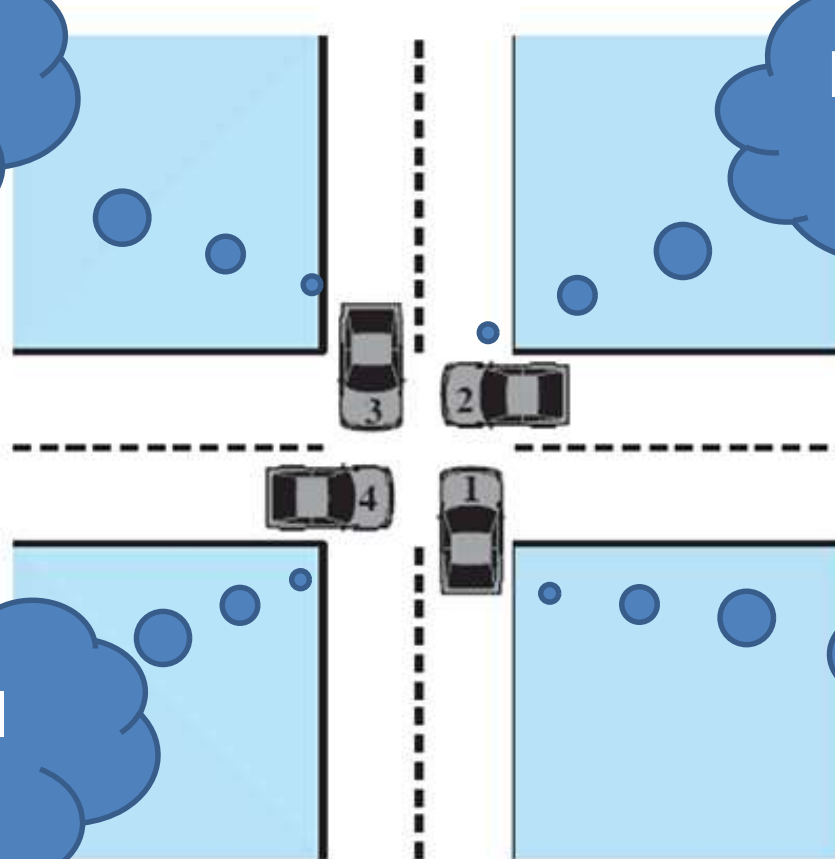
Actual Deadlock

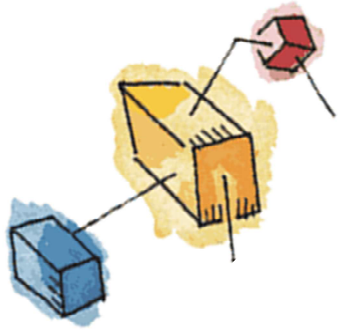
HALT until
d is free

HALT until
c is free

HALT until
a is free

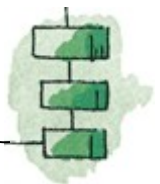
HALT until
b is free

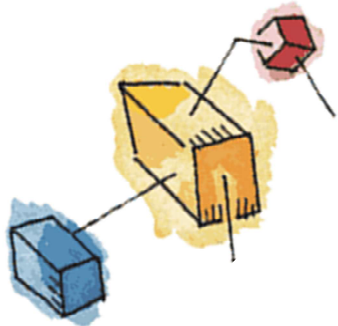




Resource Categories

- Two general categories of resources:
 - Reusable resources
 - can be safely used by only one process at a time and **is not depleted** by that use.
 - Examples: processors, main and secondary memory, devices, and data structures such as files, databases, and semaphores
 - Consumable resources
 - can be created (**produced**) and destroyed (**consumed**).
 - Examples: interrupts, signals, messages, and information in I/O buffers





Reusable Resources: Example of Deadlock

- Consider two processes that compete for exclusive access to a disk file D and a tape drive T.
- Deadlock occurs if **each process** holds **one resource** and **requests the other**, e.g., execution of $p_0 p_1 q_0 q_1 p_2 q_2$.

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

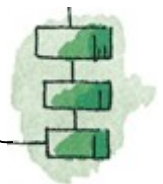
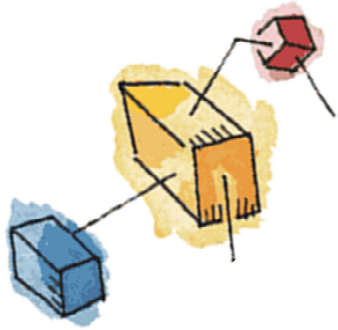
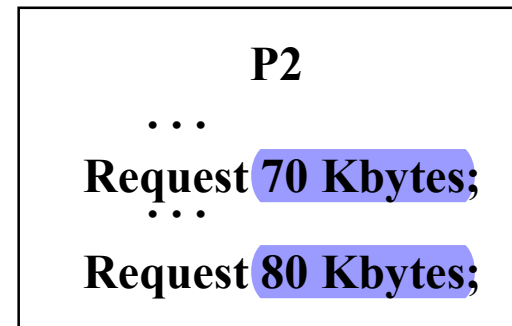
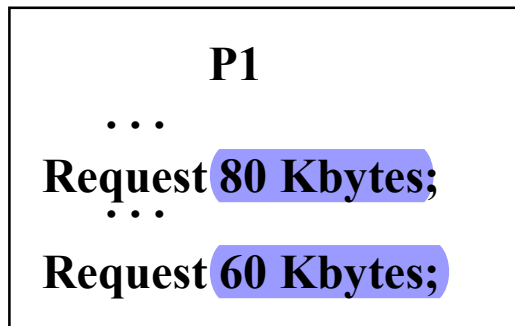


Figure 6.4 Example of Two Processes Competing for Reusable Resources

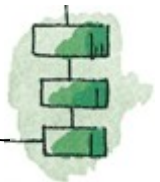


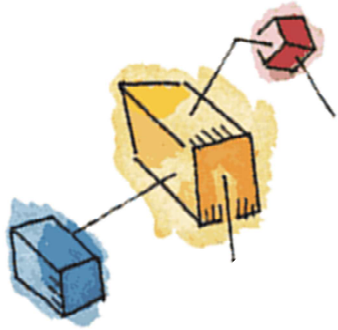
Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur.



- Deadlock occurs if both processes progress to their second request.



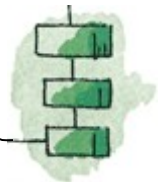


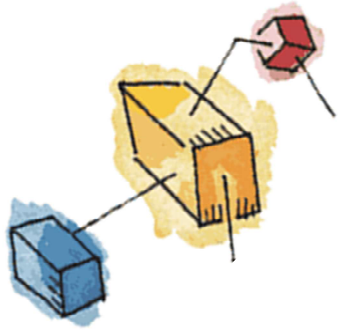
Consumable Resources: Example of Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process.
- Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received).

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

Such design errors are often embedded in complex program logic, making it difficult to detect.



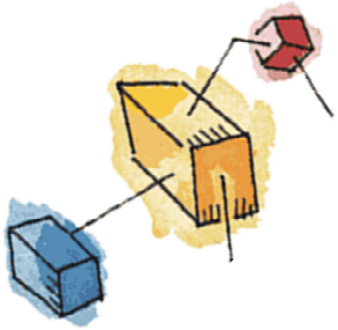


Conditions for *possible* Deadlock

- Mutual exclusion
 - Only one process may use a resource at a time.
 - No process may access a resource unit that has been allocated to another process
- Hold-and-wait
 - A process may hold allocated resources while awaiting assignment of others.
- No pre-emption
 - No resource can be forcibly removed from a process holding it.

These three conditions are necessary but not sufficient for a deadlock to exist.





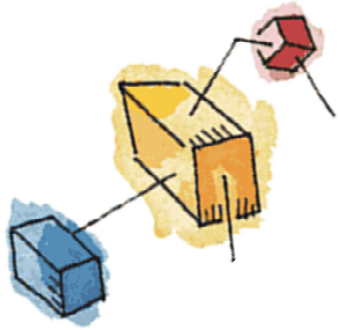
Actual Deadlock Requires ...

Given that the first 3 conditions exist, a sequence of events may occur that lead to the following fourth condition:

- Circular wait
 - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.
 - It is in fact the definition of deadlock.

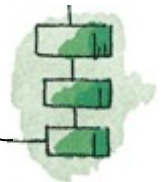
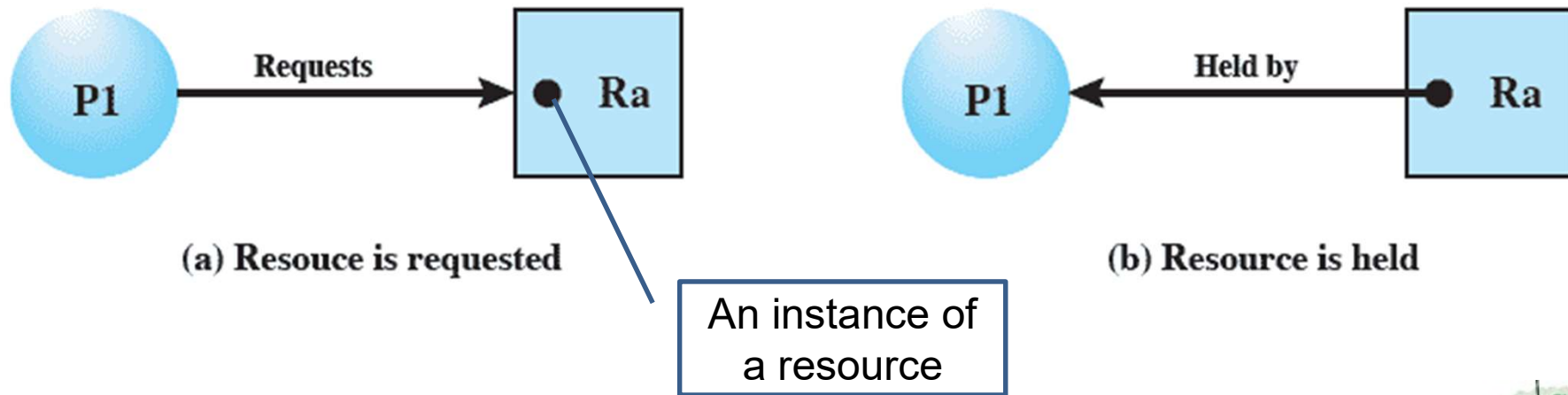
These four conditions, taken together, constitute necessary and sufficient conditions for deadlock.

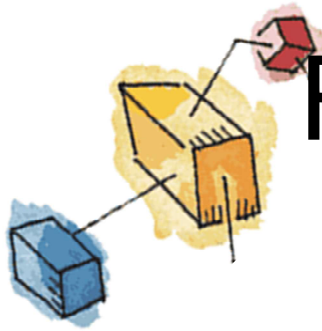




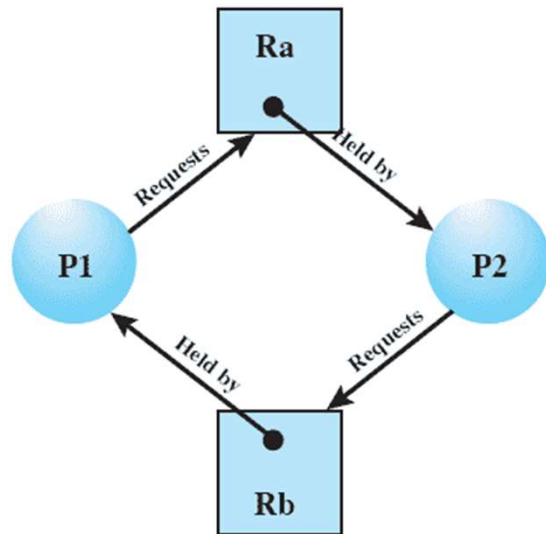
Resource Allocation Graphs

- A useful tool that characterizes the allocation of resources to processes.
- **Directed graph** that depicts a state of the system of resources and processes.





Resource Allocation Graphs (with deadlock)



(c) Circular wait

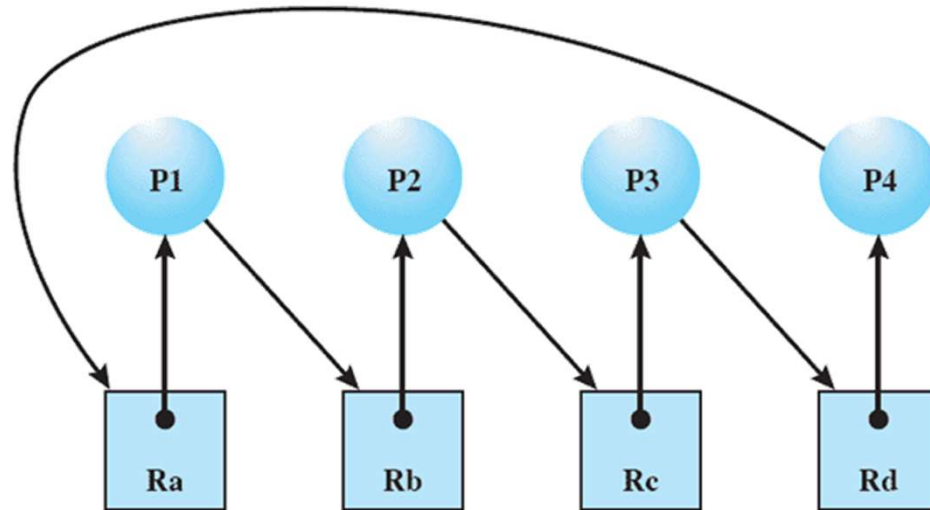
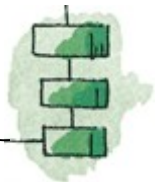
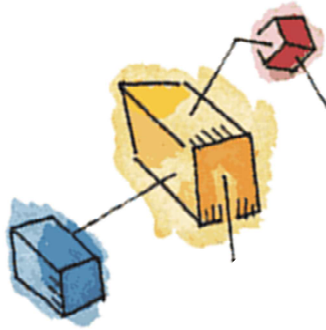


Figure 6.6 Resource Allocation Graph for Figure 6.1b

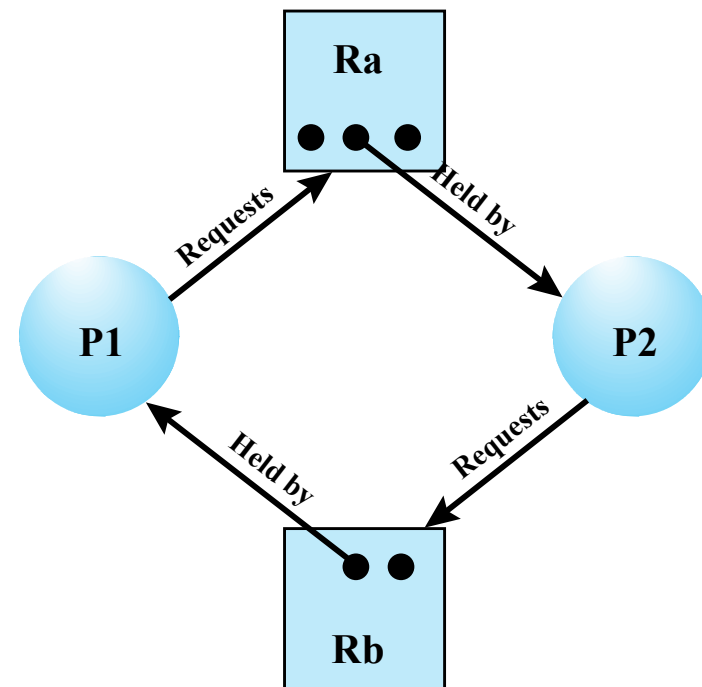
(the traffic deadlock shown in slide 5)



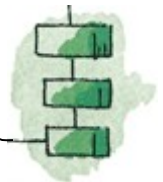


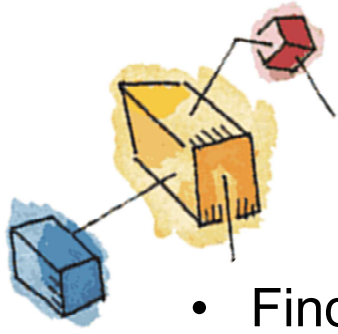
Resource Allocation Graph (no deadlock)

- There is no deadlock because multiple instances of each resource type are available.
- 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, then deadlock **MAY** exist.



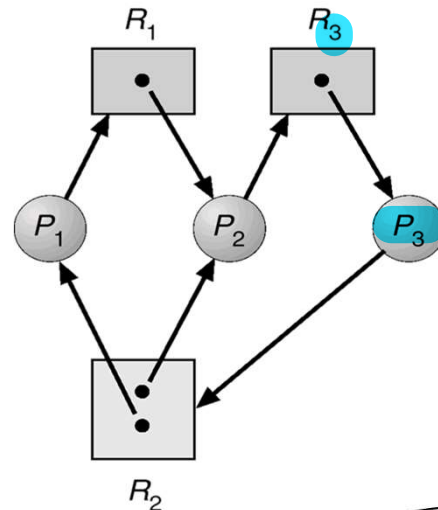
(d) No deadlock





How to find out whether there is a deadlock?

- Find a process that can have all its current requests satisfied
 - The “available amount” of any resource it wants is enough to satisfy the request.
- Erase that process
 - Grant the request, let it run, and eventually it will release the resource.
- Continue until we either erase the graph or have an irreducible component. In the latter case, we’ve identified a deadlock.
- Example:



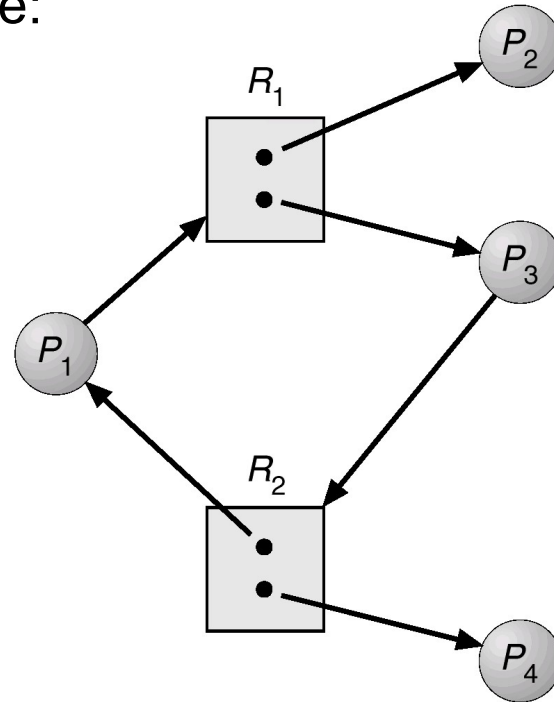
No such a process can be found to reduce the graph. **Deadlock exists.**





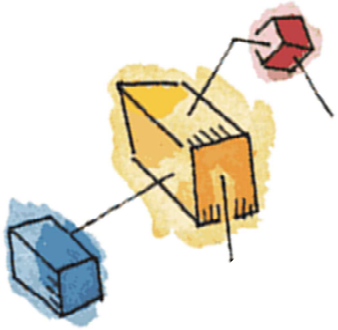
Reduce the Resource Allocation Graph

- Another example:



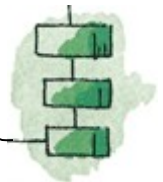
This graph can be fully reduced.
Thus, there is no deadlock.

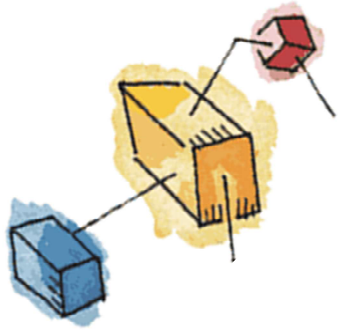




Dealing with Deadlock

- Three common approaches dealing with deadlock.
 - Deadlock prevention
 - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening
 - Deadlock avoidance
 - Do not grant a resource request if this allocation might lead to deadlock
 - Deadlock detection
 - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover



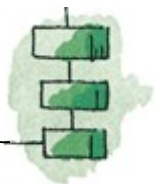


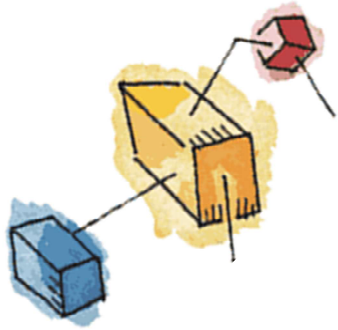
Roadmap

- Principles of Deadlock

→ Deadlock Prevention

- Deadlock Avoidance
- Deadlock Detection
- Dining Philosophers Problem

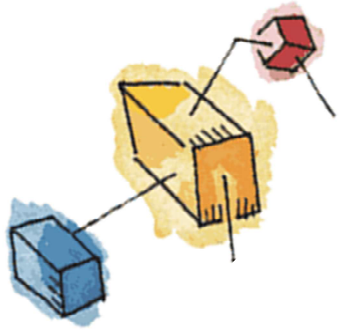




Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.
- Two main methods
 - Indirect
 - prevent the occurrence of one of the three necessary conditions
 - Direct
 - prevent the occurrence of a circular wait

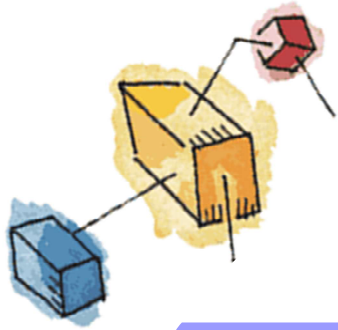




Deadlock Prevention Conditions 1 & 2

- Mutual Exclusion
 - If access to a resource requires mutual exclusion, then it must be supported by the OS
- Hold and Wait
 - Require a process request **all** of its required resources at **one** time and OS will block the process until all requests can be granted simultaneously
 - ☹ Inefficient and may be impractical

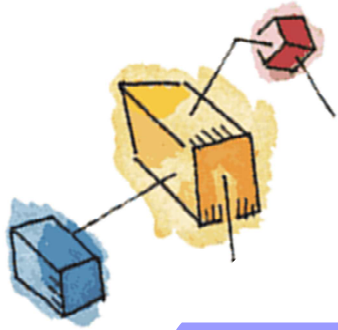




Deadlock Prevention Conditions 3

- No Preemption
 - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
 - If a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources.
 - Practical only for resources whose state can be easily saved and restored later, e.g., processor

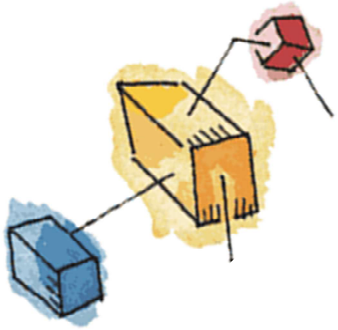




Deadlock Prevention Condition 4

- Circular Wait
 - Define a **linear ordering** of resource types
 - If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering
 - 🙅 **Inefficient**, slowing down processes and denying resource access unnecessarily





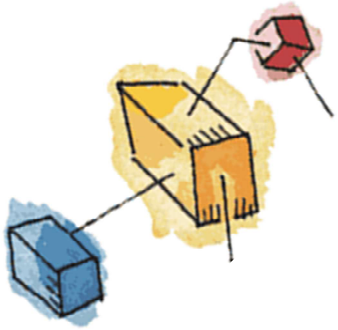
Roadmap

- Principles of Deadlock
- Deadlock Prevention

→ Deadlock Avoidance

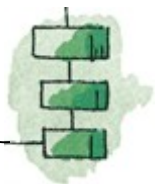
- Deadlock Detection
- Dining Philosophers Problem

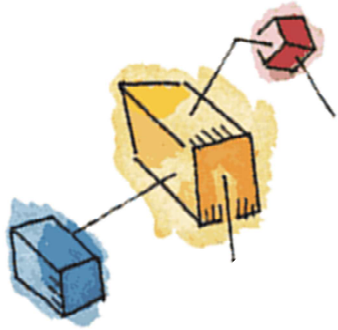




Deadlock Avoidance

- A decision is made **dynamically** whether the current resource allocation request will, ***if granted***, potentially lead to a deadlock
- 👍 Allows more concurrency than prevention
- 👎 Requires knowledge of future process requests

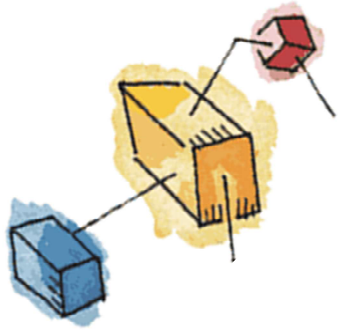





Two Approaches to Deadlock Avoidance

- Process Initiation Denial
 - Do not start a process if its demands might lead to deadlock
- Resource Allocation Denial
 - Do not grant an **incremental** resource request to a process if this allocation might lead to deadlock

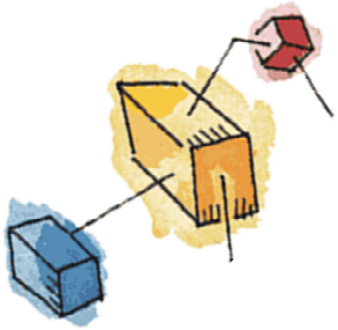




Process Initiation Denial

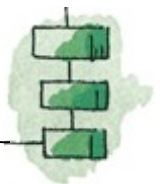
- A process is only started if the **maximum claim** (maximum requirement for each resource) of **all current processes plus** those of the **new process** can be **met by the total amount** of resources in the system.
-  **Not optimal**
 - Assumes the worst case that all processes will make their maximum claims together.

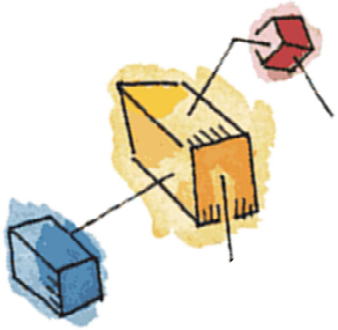




Resource Allocation Denial

- Referred to as the *banker's algorithm*
 - A strategy of resource allocation denial
- Consider a system with fixed number of resources
 - **State** of the system is the **current** allocation of resources to processes
 - **Safe state** is one in which there is **at least one sequence** of resource allocations to processes that **does not result in deadlock**, i.e., **all processes can be run to completion**.
 - **Unsafe state** is a state that is not safe





Determination of Safe State

- A system consisting of four processes and three resources.
- Is this a safe state?***

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C - A

requirement of process i for resource j

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

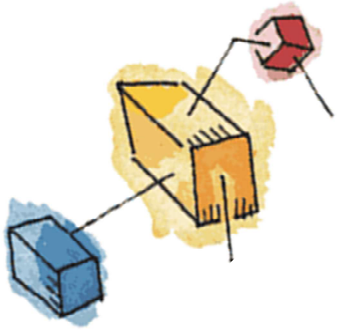
current allocation to process i of resource j

total amount of each resource

(a) Initial state

amount of each resource available for allocation



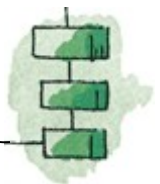


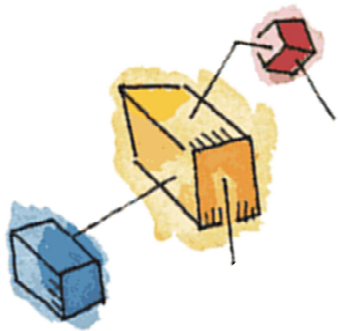
Process i

- A process i can run to completion if it meets the following condition:

$$C_{ij} - A_{ij} \leq V_j \text{ for all } j$$

- This is not possible for P1,
 - which has only 1 unit of R1 and requires 2 more units of R1, 2 units of R2, and 2 units of R3.
- If we assign one unit of R3 to process P2,
 - Then P2 has its maximum required resources allocated and can run to completion and return resources to 'available' pool





After P2

runs to completion

- Can any of the remaining processes be completed?

P2 is completed already

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

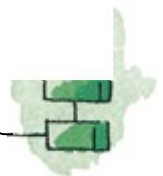
R1	R2	R3
9	3	6

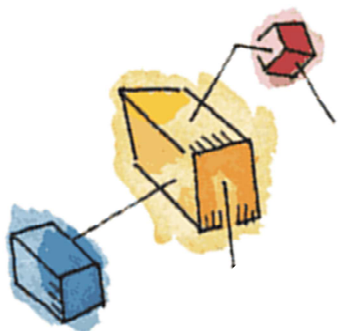
Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion





After P1 completes

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

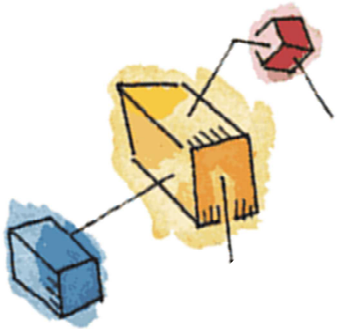
Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion





P3 Completes

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

R1	R2	R3
9	3	6

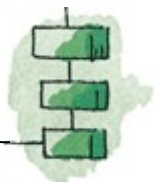
Resource vector R

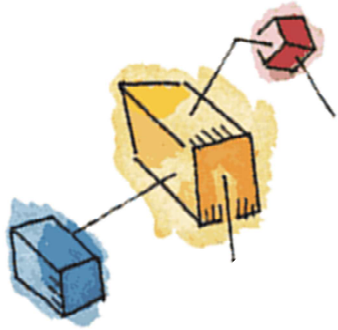
R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

Thus, the initial state is a safe state.

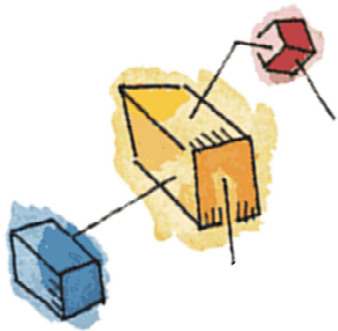




Determination of Granting a Request

- When a process makes a request for a set of resources,
 - assume that the request is granted,
 - update the system state accordingly
- Then determine if the result is a safe state
 - If so, grant the request and,
 - if not, block the process until it is safe to grant the request.





Determination of Granting a Request

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

Suppose that **P2** requests one additional unit each of R1 and R3. Is it safe to grant this request? Yes

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

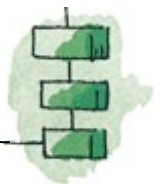
C - A

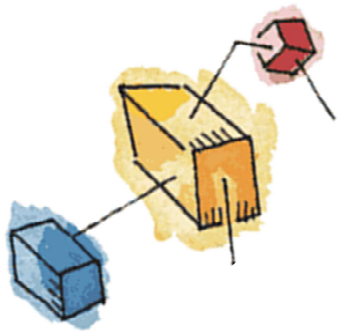
R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V





Determination of Granting a Request

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

This time, suppose that **P1** requests one additional unit each of R1 and R3. Is it safe to grant this request? No

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

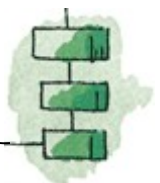
R1	R2	R3
9	3	6

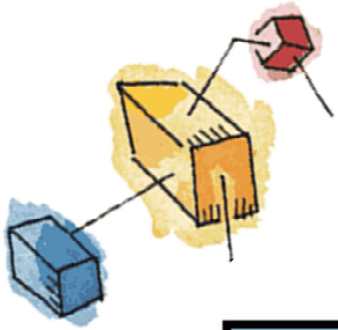
Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3





Deadlock Avoidance Logic

```
struct state {  
    int resource[m];  
    int available[m];  
    int claim[n][m];  
    int alloc[n][m];  
}
```

n processes and m different types of resources

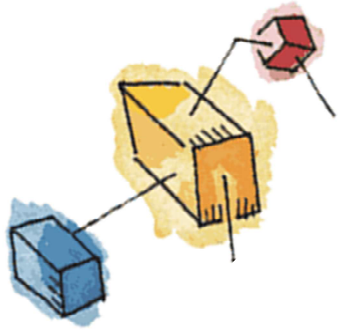
request[*] is a vector defining the resources requested by process i

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])  
    < error >; /* total request > claim*/  
else if (request [*] > available [*])  
    < suspend process >; no enough resource  
else { /* simulate alloc */  
    < define newstate by:  
    alloc [i,*] = alloc [i,*] + request [*];  
    available [*] = available [*] - request [*] >;  
    }  
    if (safe (newstate))  
        < carry out allocation >;  
    else {  
        < restore original state >;  
        < suspend process >;  
    }  
}
```

(b) resource alloc algorithm





Deadlock Avoidance Logic

```
boolean safe (state S) {  
    int currentavail[m];  
    process rest[<number of processes>];  
    currentavail = available;  
    rest = {all processes};  
    possible = true;  
    while (possible) {  
        <find a process  $P_k$  in rest such that  
            claim  $[k, *] - \text{alloc } [k, *] \leq \text{currentavail};$ >  
        if (found) {                                /* simulate execution of  $P_k$  */  
            currentavail = currentavail + alloc  $[k, *]$ ;  
            rest = rest -  $\{P_k\}$ ;  
        }  
        else possible = false;  
    }  
    return (rest == null);  
}
```

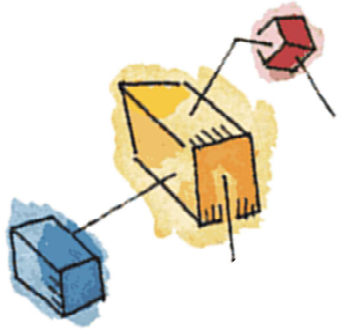
The process can
run to completion

The process
returns resources
upon completion

(c) test for safety algorithm (banker's algorithm)

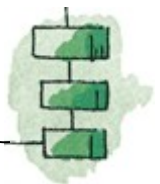
Figure 6.9 Deadlock Avoidance Logic

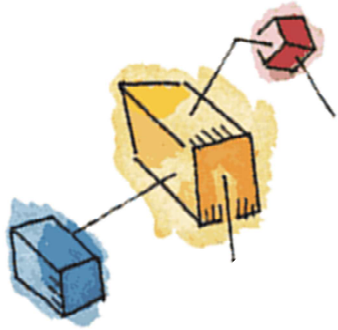




Deadlock Avoidance Advantages

- 👍 It is **less restrictive** than deadlock prevention.
- 👍 It is **not necessary to preempt** and **rollback processes**, as in deadlock detection (to be discussed)

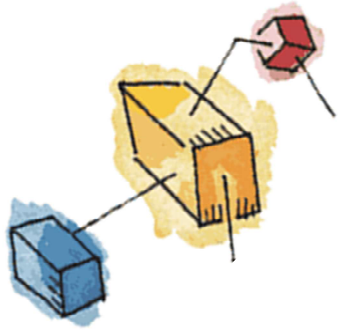




Deadlock Avoidance Restrictions

- ☞ Maximum resource requirement for each process must be stated in advance
- ☞ Processes under consideration must be independent and with no synchronization requirements
- ☞ There must be a fixed number of resources to allocate
- ☞ No process may exit while holding resources



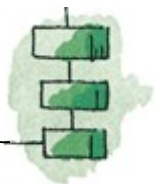


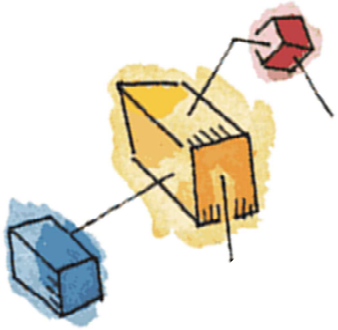
Roadmap

- Principles of Deadlock
- Deadlock Prevention
- Deadlock Avoidance

→ Deadlock Detection

- Dining Philosophers Problem

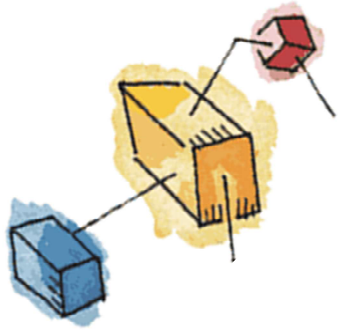




Deadlock Detection

- Deadlock prevention strategies are very conservative
 - limit access to resources and impose restrictions on processes.
- Deadlock detection strategies do the opposite
 - Resource requests are granted whenever possible.
 - Regularly check for deadlock.

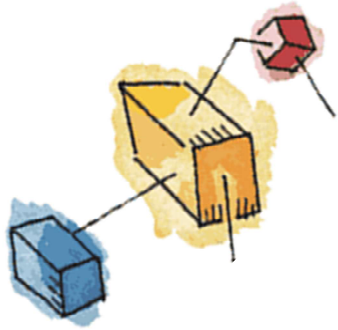




A Common Detection Algorithm

- Main idea:
 - Find and **mark** a process whose resource requests can be **satisfied** with the available resources
 - Assume that those resources are granted and that the process runs to completion and releases all its resources
 - Look for another process to satisfy
 - A **deadlock exists** if and only if there are **unmarked** processes at the end

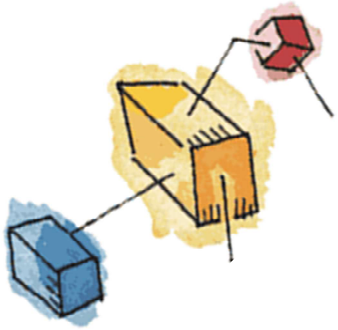




A Common Detection Algorithm

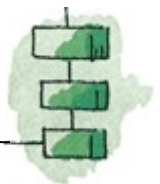
- Use a Allocation matrix and Available vector as in the Banker's algorithm
- Also use a Request matrix **Q**
 - Where Q_{ij} indicates that an amount of resource j is requested by process i
- First, 'un-mark' all processes.

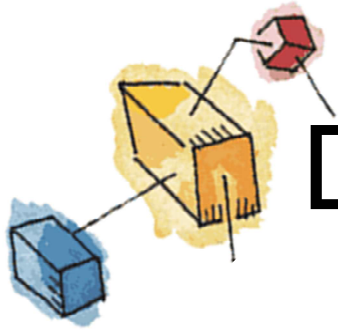




Detection Algorithm

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector W to equal the Available vector.
3. Find an index i such that process i is currently unmarked and the i th row of Q is less than or equal to W .
 - i.e. $Q_{ik} \leq W_k$ for $1 \leq k \leq m$ for m different types of resources
 - If no such row is found, terminate





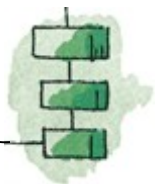
Detection Algorithm cont.

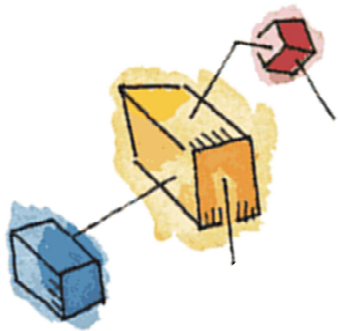
4. If such a row is found,

- mark process i and add the corresponding row of the allocation matrix to \mathbf{W} .
- i.e. set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$;

Return to step 3.

- A deadlock exists if and only if there are unmarked processes at the end
- Each unmarked process is deadlocked.





Detection Algorithm Example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
→ P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3 ✓	0	0	0	1	0
P4 ✓	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

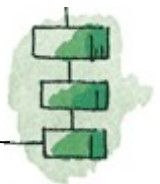
Available vector

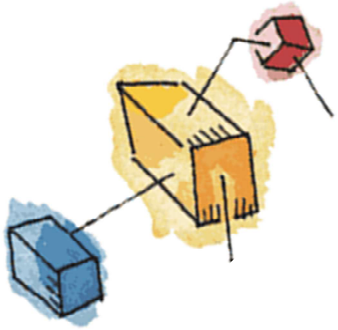
Figure 6.10 Example for Deadlock Detection

$$W = (0 \ 0 \ 0 \ 0 \ 1) \text{ Ava. Vector}$$

$$W = W + (0 \ 0 \ 0 \ 1 \ 0) = (0 \ 0 \ 0 \ 1 \ 1)$$

P1 and P2 are **deadlocked**

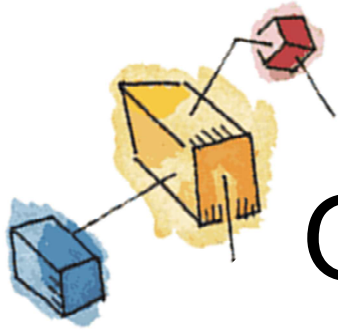




Deadlock Detection

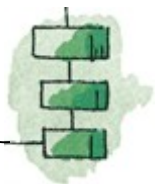
- A check for deadlock can be made
 - as frequently as each resource request,
 - 👍 It leads to early detection.
 - 👍 The algorithm is relatively simple.
 - 👎 Frequent checks consume considerable processor time.
 - or, less frequently, depending on how likely it is for a deadlock to occur.

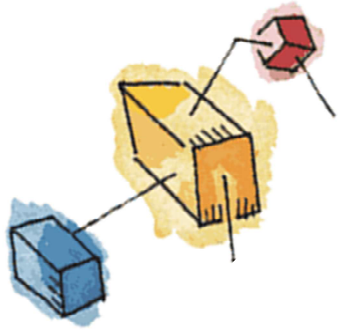




Recovery Strategies Once Deadlock Detected

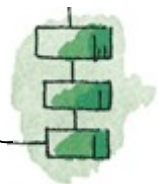
- Abort all deadlocked processes
- Back up (rollback) each deadlocked process to some previously defined checkpoint, and restart all processes
 - Risk of deadlock recurring
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources and rollback the preempted process until deadlock no longer exists

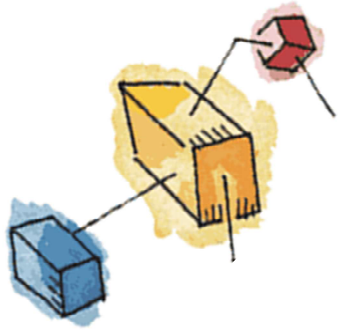




Advantages and Disadvantages

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> • Works well for processes that perform a single burst of activity • No preemption necessary 	<ul style="list-style-type: none"> • Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	<ul style="list-style-type: none"> • Convenient when applied to resources whose state can be saved and restored easily 	<ul style="list-style-type: none"> • Preempts more often than necessary
		Resource ordering	<ul style="list-style-type: none"> • Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design 	<ul style="list-style-type: none"> • Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> • No preemption necessary 	<ul style="list-style-type: none"> • Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"> • Never delays process initiation • Facilitates online handling 	<ul style="list-style-type: none"> • Inherent preemption losses



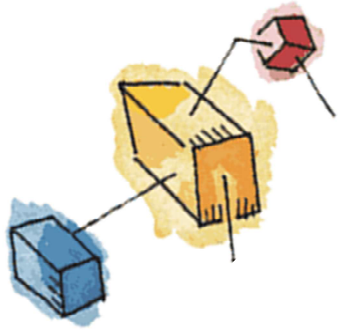


Roadmap

- Principles of Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

➔ Dining Philosophers Problem





Dining Philosophers Problem: Scenario

- The life of a philosopher consists of thinking and eating spaghetti.
- A philosopher requires two forks to eat spaghetti.
- A philosopher wishing to eat goes to his assigned place and uses the two forks on either side of the plate to eat some spaghetti.

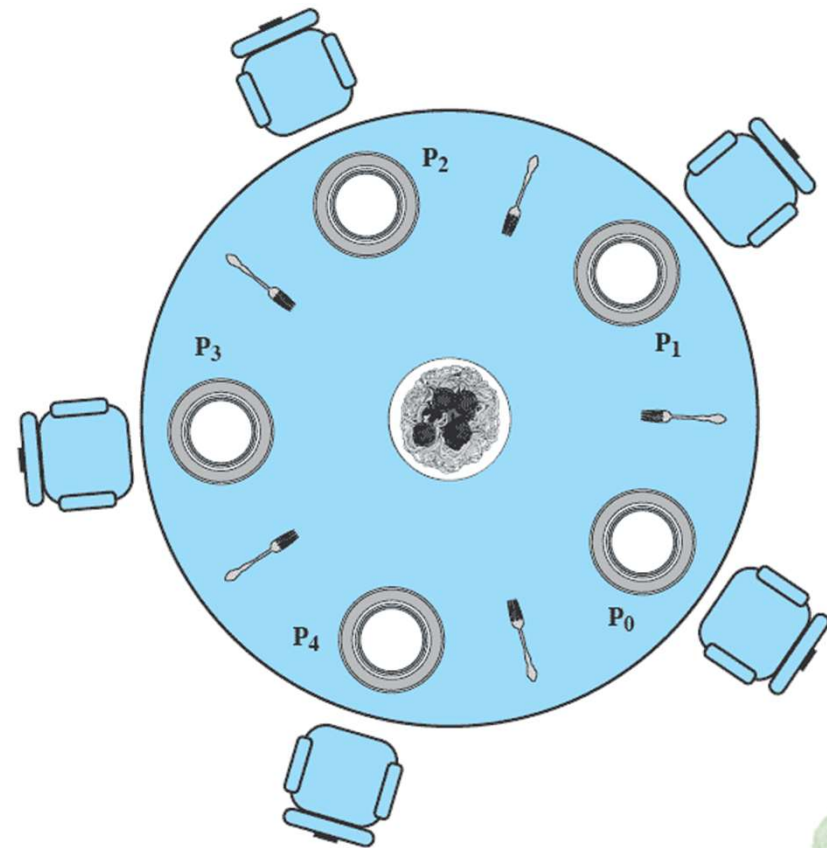
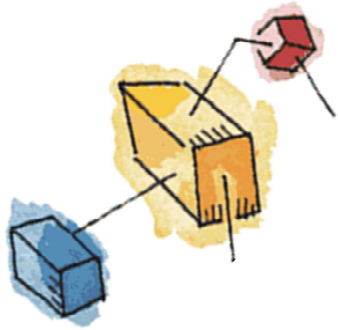


Figure 6.11 Dining Arrangement for Philosophers



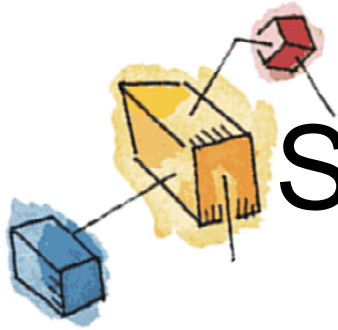


The Problem

- Devise a ritual (algorithm) that will allow the philosophers to eat.
 - No two philosophers can use the same fork at the same time (mutual exclusion)
 - No philosopher must starve to death (avoid deadlock and starvation ... literally!)

This is a representative problem to illustrate basic problems in deadlock and starvation.





Solution Using Semaphores

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
}
```

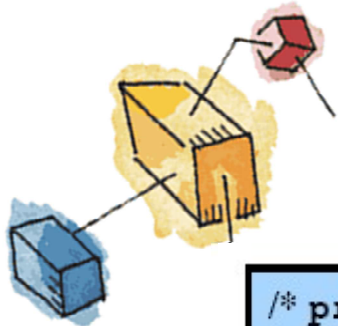
Each philosopher picks up first the fork on the left and then the fork on the right.

What will happen if all of the philosophers are hungry at the same time?

After eating, the two forks are replaced on the table.



Figure 6.12 A First Solution to the Dining Philosophers Problem



Avoiding Deadlock

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}

void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

An attendant only allows four philosophers at a time into the dining room.

This solution is free of deadlock and starvation.



Figure 6.13 A Second Solution to the Dining Philosophers Problem

