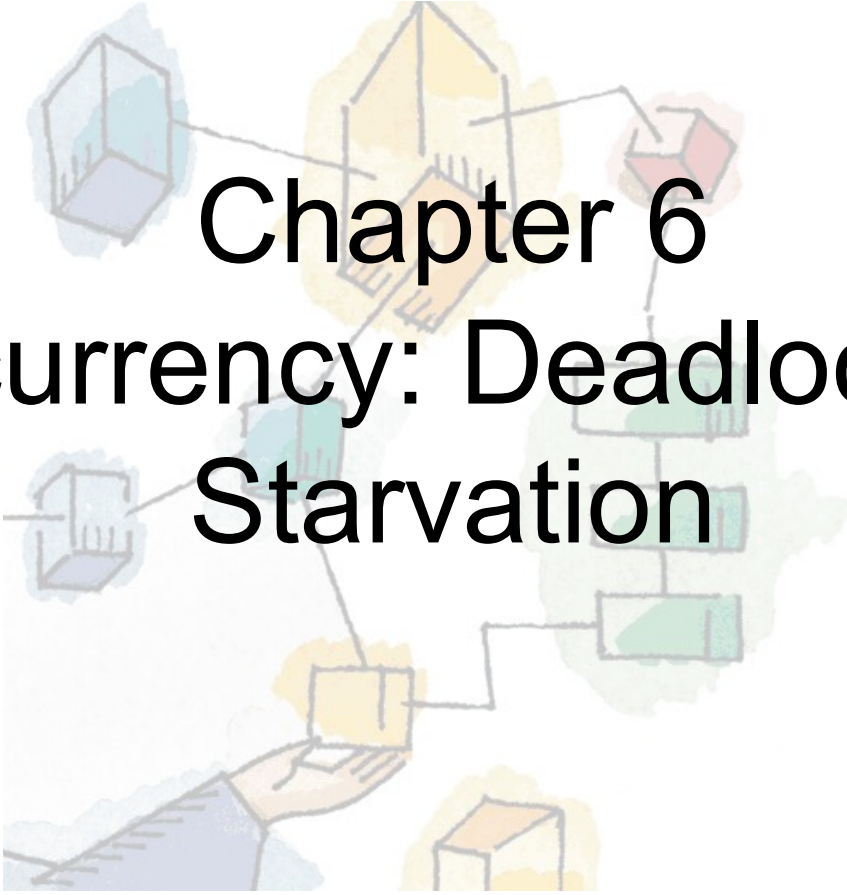


*Operating Systems:  
Internals and Design Principles, 6/E*  
William Stallings



# Chapter 6

## Concurrency: Deadlock and Starvation

Dave Bremer  
Otago Polytechnic, N.Z.  
©2008, Prentice Hall



# Roadmap

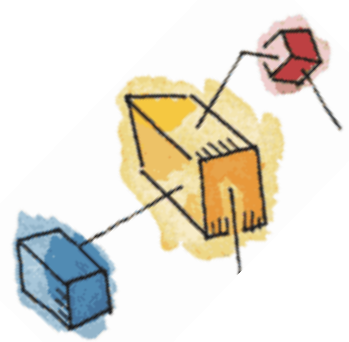
## → Principals of Deadlock

- Deadlock prevention
- Deadlock Avoidance
- Deadlock detection
- An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows




# Deadlock

- 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
  - Typically involves processes competing for the same set of resources
- No efficient solution



# Potential Deadlock

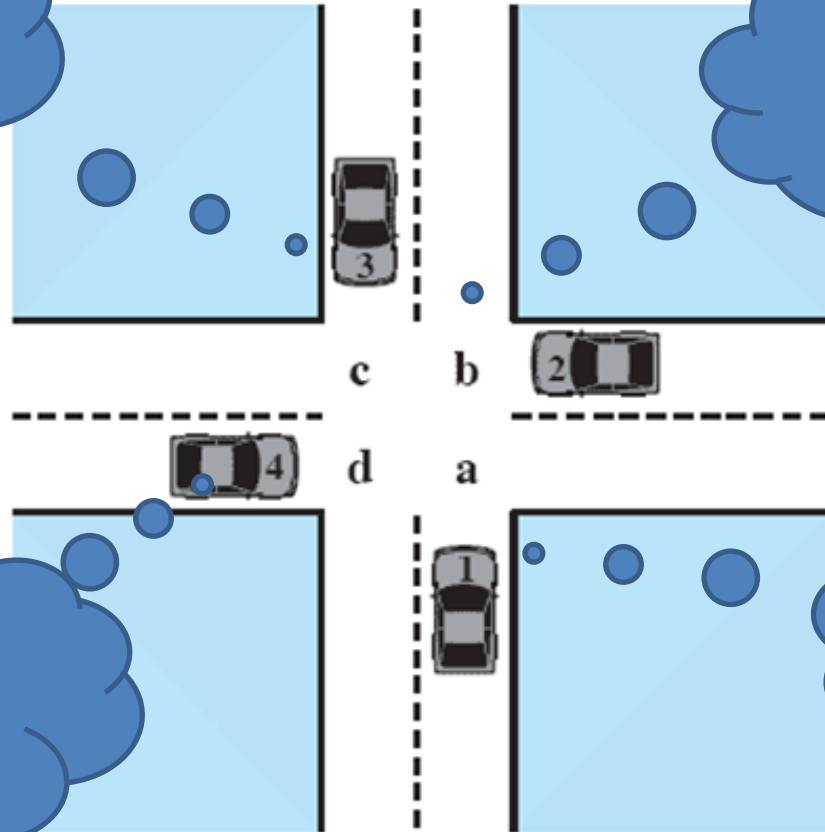




I need  
quad C  
and B

I need  
quad B  
and C

I need  
quad D  
and A

I need  
quad A and  
B



# Actual Deadlock



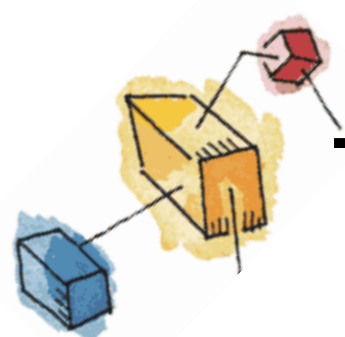
HALT until  
D is free

HALT until  
C is free

HALT until  
A is free

HALT until  
B is free





# Two Processes P and Q

- Lets look at this with two processes P and Q
- Each needing exclusive access to a resource A and B for a period of time

## Process P

...

Get A

...

Get B

...

Release A

...

Release B

...

## Process Q

...

Get B

...

Get A

...

Release B

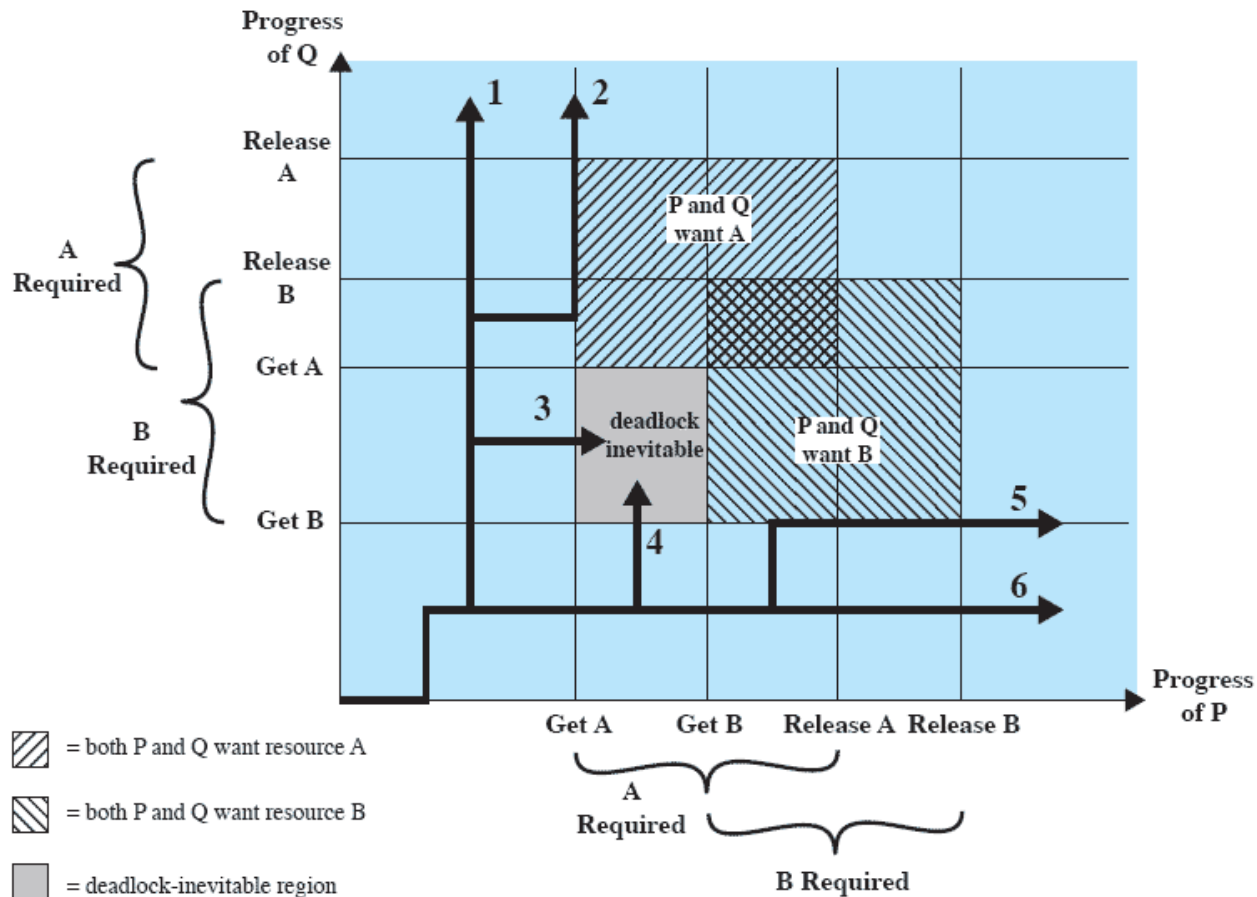
...

Release A

...



# Joint Progress Diagram of Deadlock



→ = possible progress path of P and Q.  
Horizontal portion of path indicates P is executing and Q is waiting.  
Vertical portion of path indicates Q is executing and P is waiting.

Figure 6.2 Example of Deadlock



# Alternative logic

- Suppose that P does not need both resources at the same time so that the two processes have this form

## Process P

• • •  
Get A  
• • •  
Release A  
• • •  
Get B  
• • •  
Release B  
• • •

## Process Q

• • •  
Get B  
• • •  
Get A  
• • •  
Release B  
• • •  
Release A  
• • •





# Diagram of alternative logic

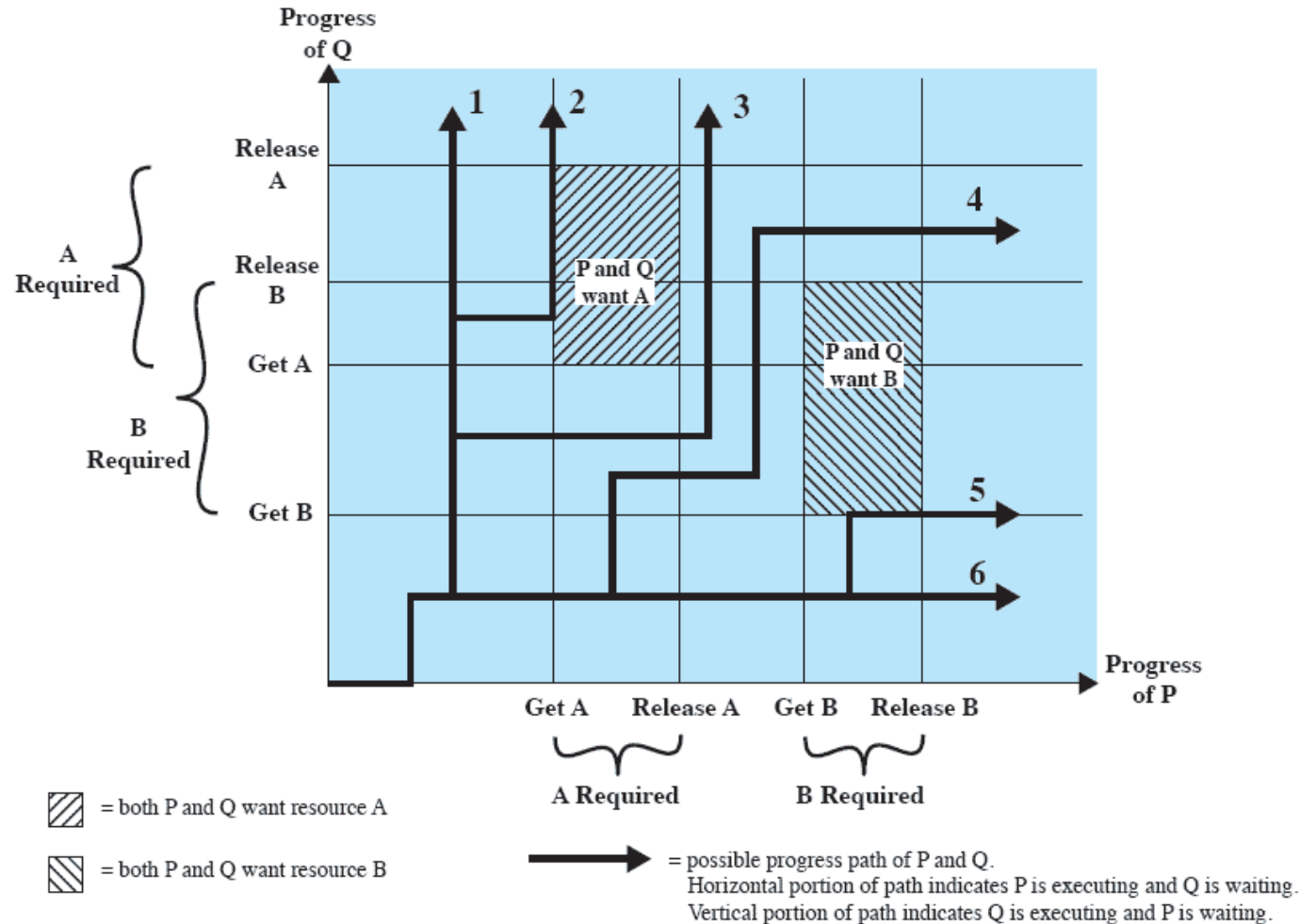
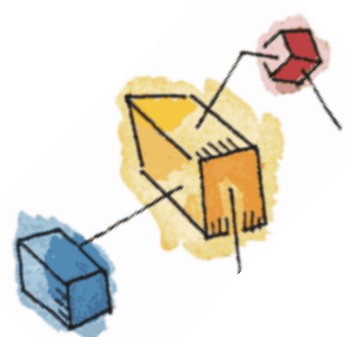


Figure 6.3 Example of No Deadlock [BACO03]

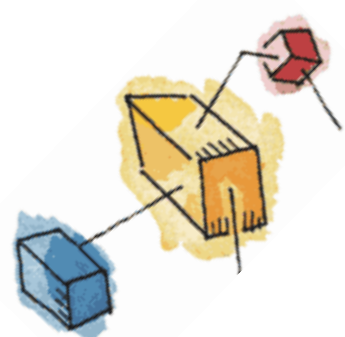


# Resource Categories

Two general categories of resources:

- Reusable
  - can be safely used by only one process at a time and ***is not depleted*** by that use.
- Consumable
  - one that can be created (***produced***) and destroyed (***consumed***).





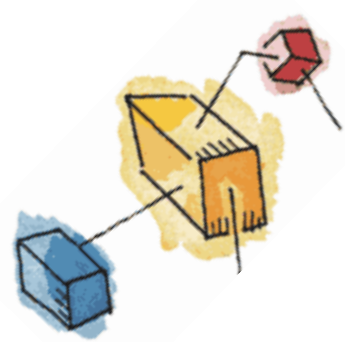
# Reusable Resources

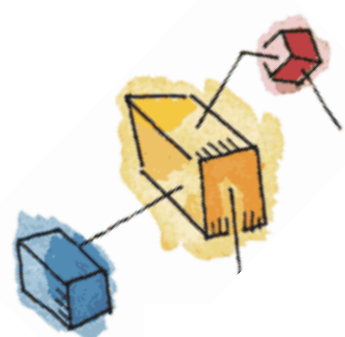
- Such as:
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other



# Example of Reuse 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.





# Reusable Resources Example

**Process P**

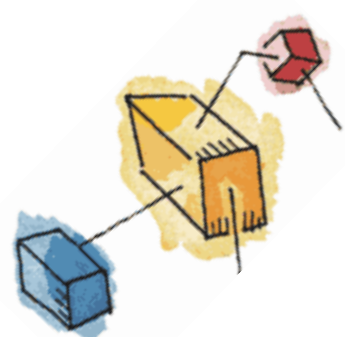
**Process Q**

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)

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**

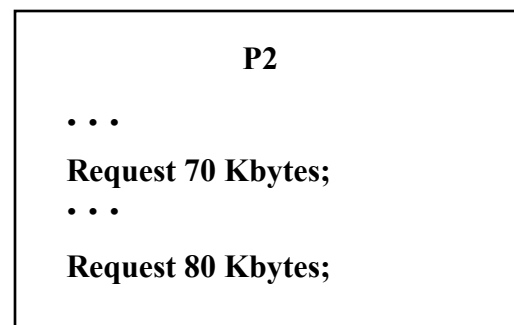
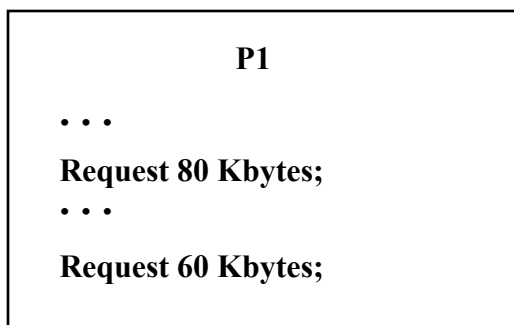




# Example 2:

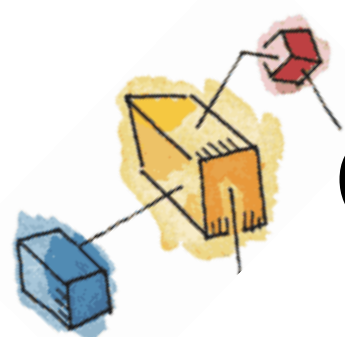
## Memory Request

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



- Deadlock occurs if both processes progress to their second request

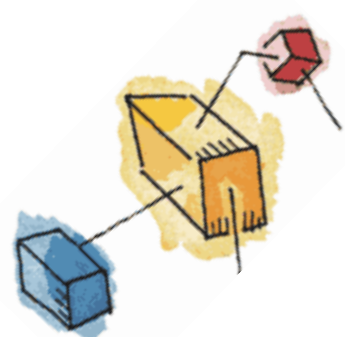




# Consumable Resources

- Such as Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock





# 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

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





An illustration in the top-left corner shows a yellow cube with a red cube on top of it, and a blue cube to the left. Lines connect these cubes, suggesting a hierarchy or allocation structure.

# Resource Allocation Graphs

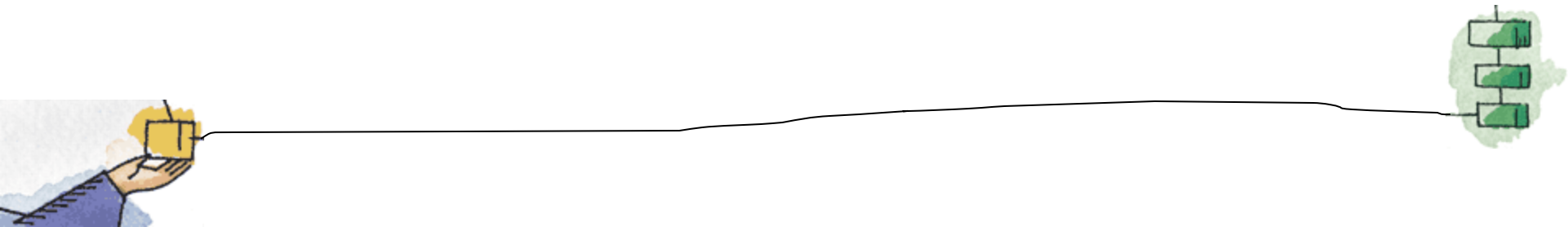
- Directed graph that depicts a state of the system of resources and processes



(a) Resource is requested



(b) Resource is held





# Conditions for ***possible*** Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- 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



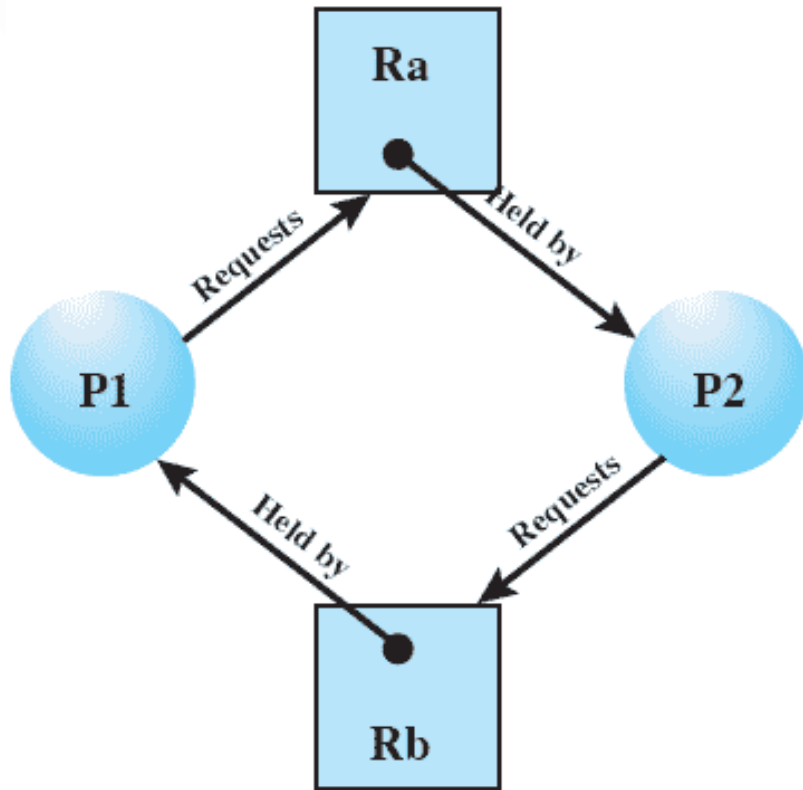
# Actual Deadlock Requires ...

All previous 3 conditions plus:

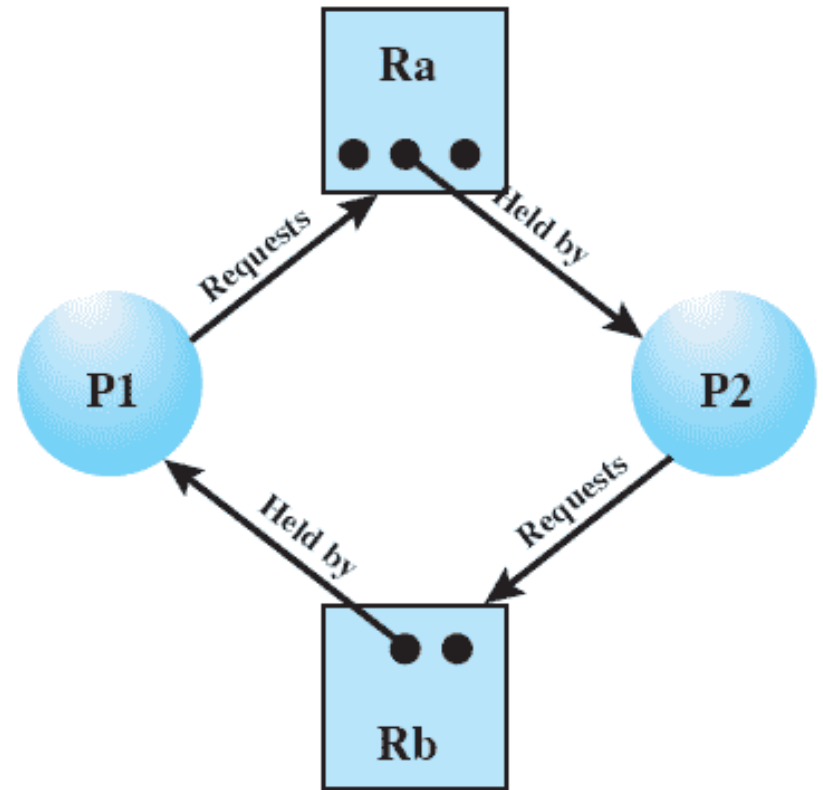
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



# Resource Allocation Graphs of deadlock

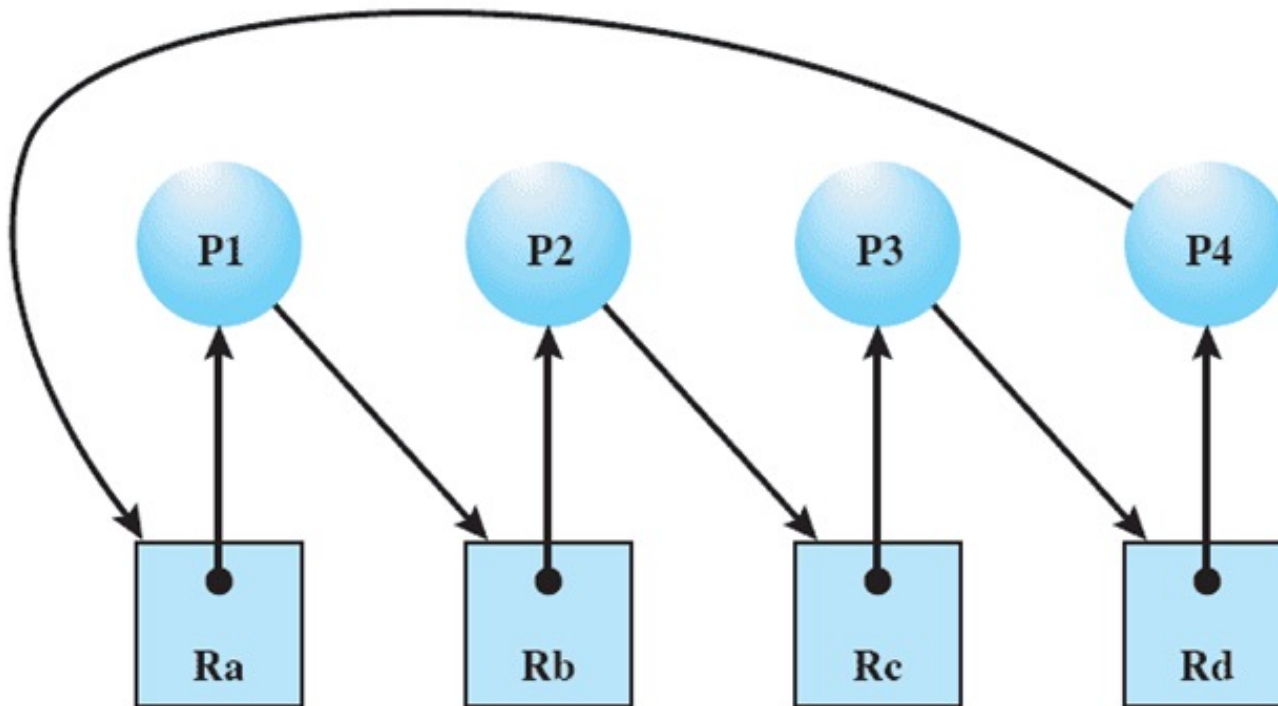


(c) Circular wait

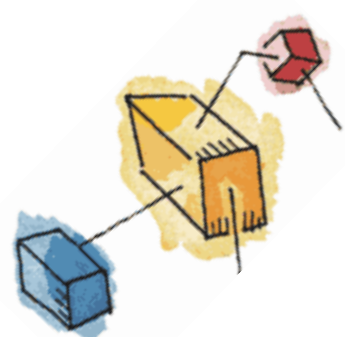


(d) No deadlock

# Resource Allocation Graphs



**Figure 6.6** Resource Allocation Graph for Figure 6.1b



# Dealing with Deadlock

- Three general approaches exist for dealing with deadlock.
  - Prevent deadlock
  - Avoid deadlock
  - Detect Deadlock





# Roadmap

- Principals of Deadlock

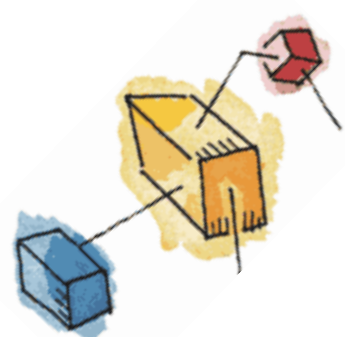
- Deadlock prevention

- Deadlock Avoidance
- Deadlock detection
- An Integrated deadlock strategy

- Dining Philosophers Problem

- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows



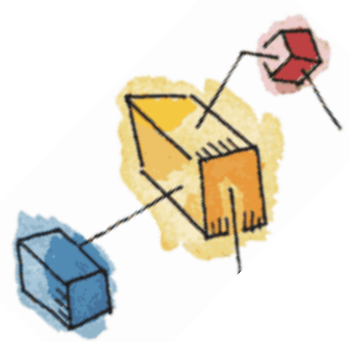


# Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.
- Two main methods
  - Indirect – prevent all three of the necessary conditions occurring at once
  - Direct – prevent circular waits



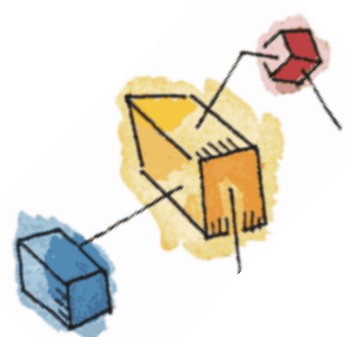




# Deadlock Prevention Conditions 1 & 2

- Mutual Exclusion
  - Must be supported by the OS
- Hold and Wait
  - Require a process request all of its required resources at one time





# Deadlock Prevention Conditions 3 & 4

- No Preemption
  - Process must release resource and request again
  - OS may preempt a process to require it releases its resources
- Circular Wait
  - Define a linear ordering of resource types

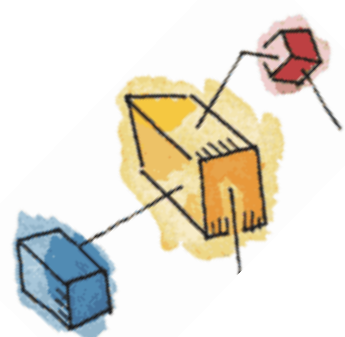




# Roadmap

- Principals of Deadlock
  - Deadlock prevention
  - Deadlock Avoidance
    - Deadlock detection
    - An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows

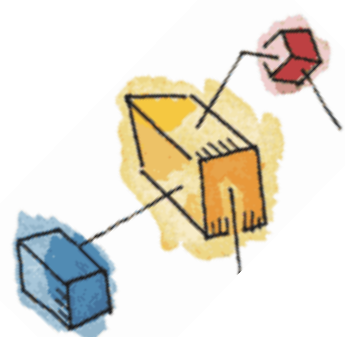




# Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests





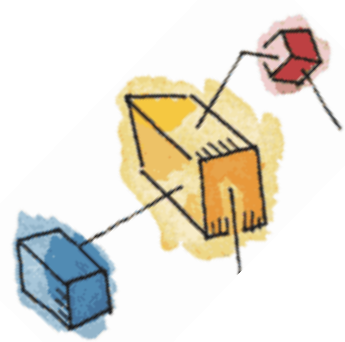
# Two Approaches to Deadlock Avoidance

- 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



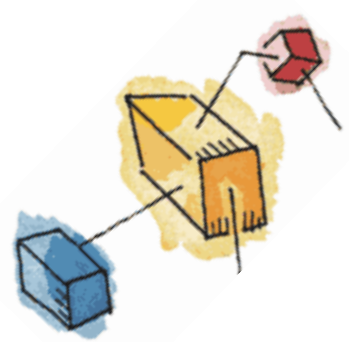
# Process Initiation Denial

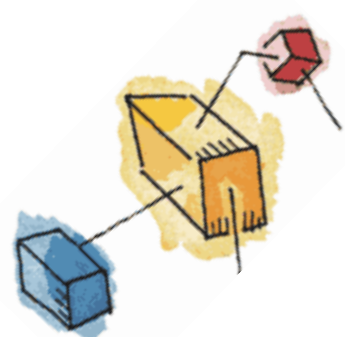
- A process is only started if the maximum claim of all current processes plus those of the new process can be met.
- Not optimal,
  - Assumes the worst: that all processes will make their maximum claims together.



# 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 process
  - **Safe state** is where there is at least one sequence that does not result in deadlock
  - **Unsafe state** is a state that is not safe





# Determination of Safe State

- A system consisting of four processes and three resources.
- Allocations are made to processors
- ***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

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

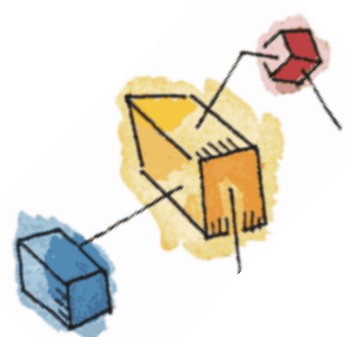




# Process $i$

- $C_{ij} - A_{ij} \leq V_j$ , 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 can be completed?

Note P2 is completed

	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

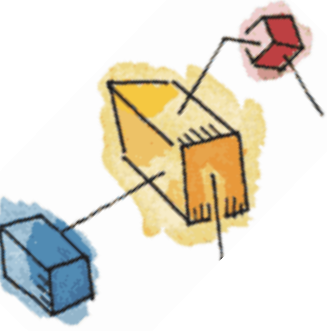
Resource vector R

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 state defined originally is a safe state.



# Determination of an Unsafe 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	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

	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

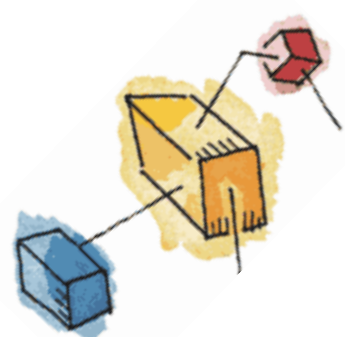
R1	R2	R3
0	1	1

Available vector V

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

This time  
Suppose that  
P1 makes the  
request for one  
additional unit  
each of R1 and  
R3.  
*Is this safe?*





# Deadlock Avoidance

- 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.






# Deadlock Avoidance Logic

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

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])  
    < error >;                                /* total request > claim*/  
else if (request [*] > available [*])  
    < suspend process >;  
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);  
}
```

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



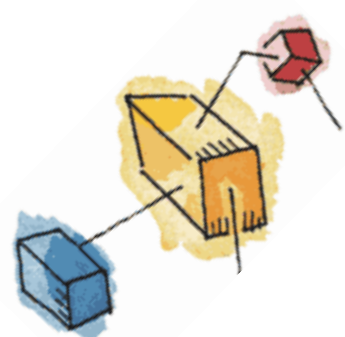


Figure 6.9 Deadlock Avoidance Logic



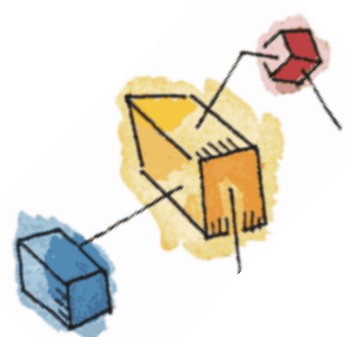




# Deadlock Avoidance Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection,
- It is less restrictive than deadlock prevention.





# Deadlock Avoidance Restrictions

- Maximum resource requirement 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

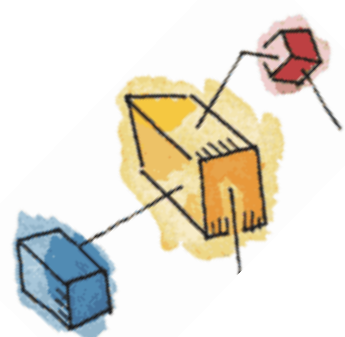




# Roadmap

- Principals of Deadlock
  - Deadlock prevention
  - Deadlock Avoidance
  - ➔ – Deadlock detection
  - ➔ – An Integrated deadlock strategy
- Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





# 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

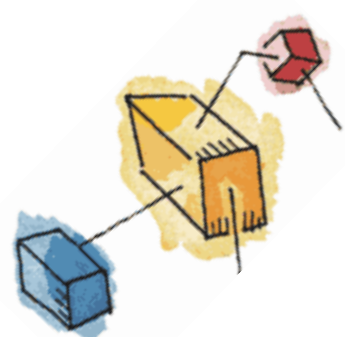




# A Common Detection Algorithm

- Use a Allocation matrix and Available vector as previous
- 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 that are not deadlocked
  - Initially that is all processes

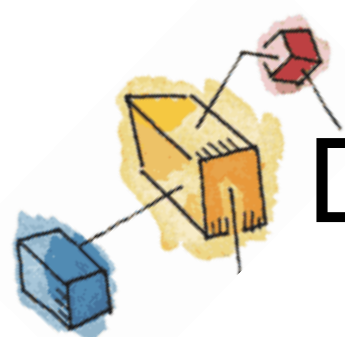




# Detection Algorithm

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  $1 \leq k \leq m$ .
  - If no such row is found, terminate





# Detection Algorithm cont.

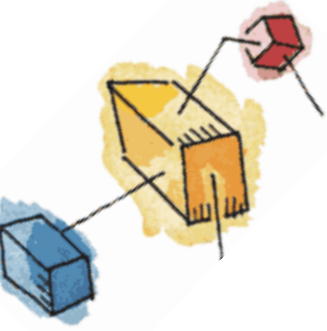
4. If such a row is found,
  - mark process  $i$  and add the corresponding row of the allocation matrix to  $W$ .
  - i.e. set  $W_k = W_k + A_{ik}$ , for  $1 \leq k \leq m$

Return to step 3.

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



# Deadlock Detection



	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

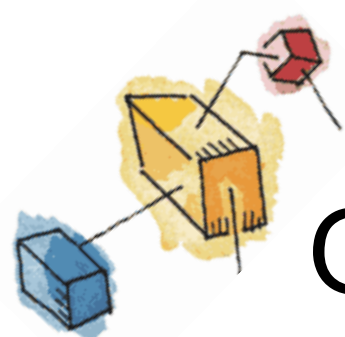
R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

**Figure 6.10 Example for Deadlock Detection**







# Recovery Strategies Once Deadlock Detected

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



# Advantages and Disadvantages

**Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]**

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



# Roadmap

- Principals of Deadlock
  - Deadlock prevention
  - Deadlock Avoidance
  - Deadlock detection
  - An Integrated deadlock strategy

## → Dining Philosophers Problem

- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows



# Dining Philosophers Problem: Scenario

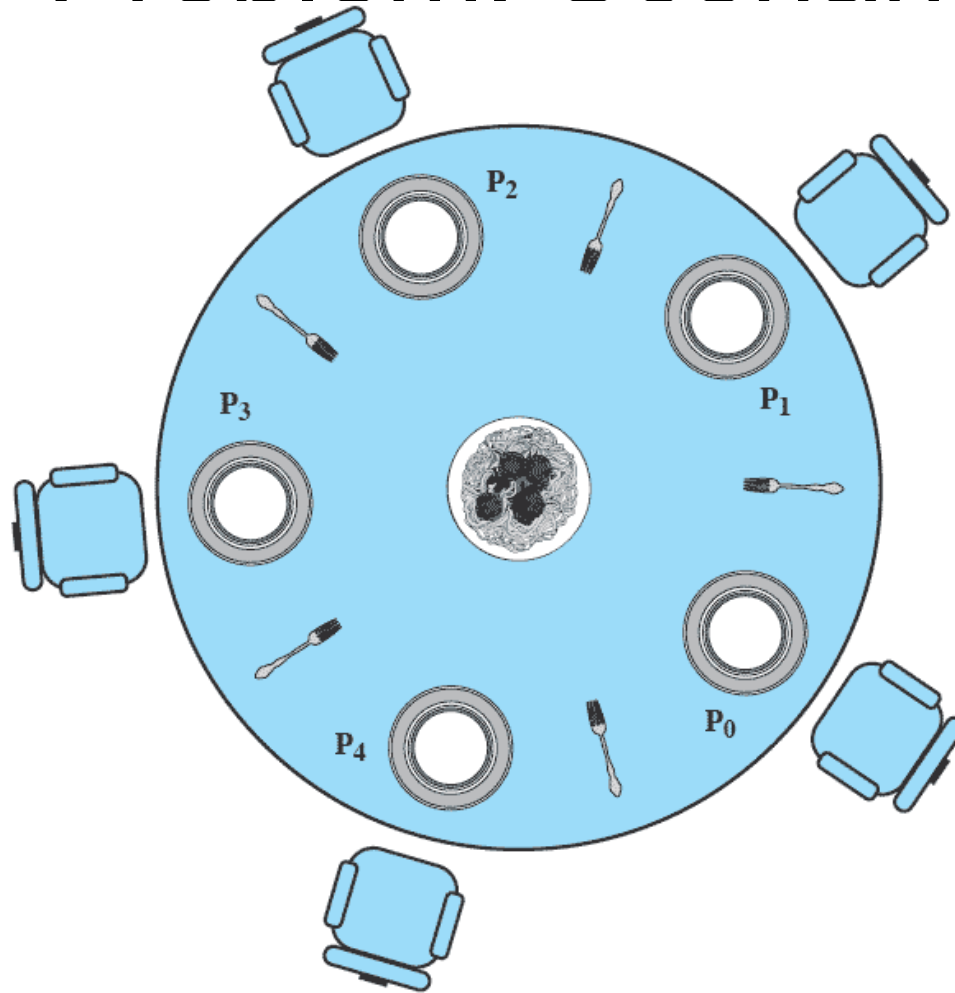
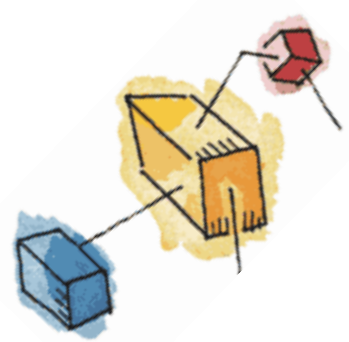
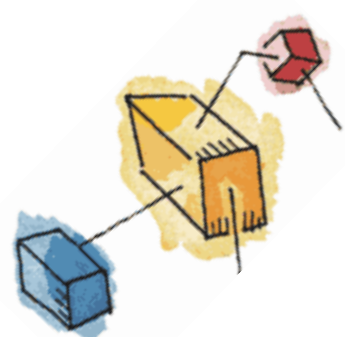


Figure 6.11 Dining Arrangement for Philosophers

# 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!)





# A first 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));
}
```

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));
}
```





Figure 6.13 A Second Solution to the Dining Philosophers Problem







# Solution using Monitors

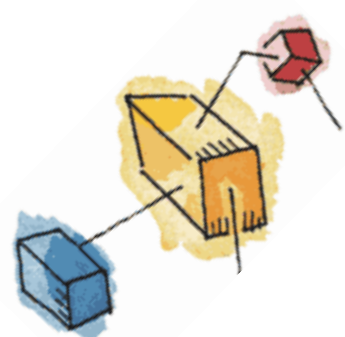
```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};    /* availability status of each fork */

void get_forks(int pid)      /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork(left))
        cwait(ForkReady[left]);          /* queue on condition variable */
    fork(left) = false;
    /*grant the right fork*/
    if (!fork(right))
        cwait(ForkReady[right]);         /* queue on condition variable */
    fork(right) = false;
}

void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left])           /*no one is waiting for this fork */
        fork(left) = true;
    else                                  /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right])          /*no one is waiting for this fork */
        fork(right) = true;
    else                                  /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```







# Monitor solution cont.

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
    while (true) {
        <think>;
        get forks(k);                 /* client requests two forks via monitor */
        <eat spaghetti>;
        release forks(k);             /* client releases forks via the monitor */
    }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor





# Roadmap

- Principals of Deadlock
    - Deadlock prevention
    - Deadlock Avoidance
    - Deadlock detection
    - An Integrated deadlock strategy
  - Dining Philosophers Problem
- Concurrency Mechanisms in UNIX, Linux, Solaris and Windows





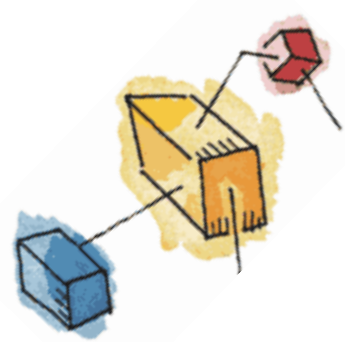
# UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals



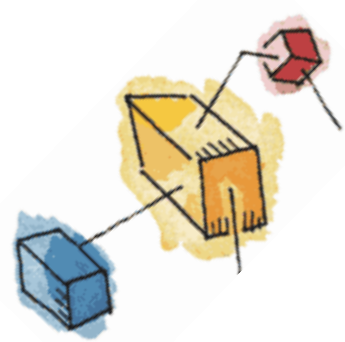
# Pipes

- A circular buffer allowing two processes to communicate on the producer-consumer model
  - first-in-first-out queue, written by one process and read by another.
- Two types:
  - Named:
  - Unnamed



# Messages

- A block of bytes with an accompanying type.
- UNIX provides *msgsnd* and *msgrcv* system calls for processes to engage in message passing.
- Associated with each process is a message queue, which functions like a mailbox.





# Shared Memory

- A common block of virtual memory shared by multiple processes.
- Permission is read-only or read-write for a process,
  - determined on a per-process basis.
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory.





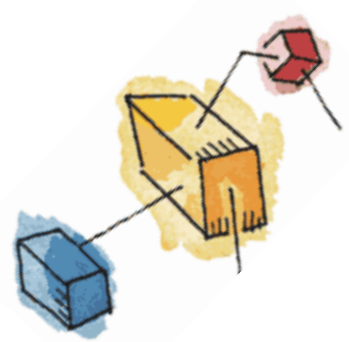
# Semaphores

- SVR4 uses a generalization of the ***semWait*** and ***semSignal*** primitives defined in Chapter 5;
- Associated with the semaphore are queues of processes blocked on that semaphore.




# Signals

- A software mechanism that informs a process of the occurrence of asynchronous events.
  - Similar to a hardware interrupt, without priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent.




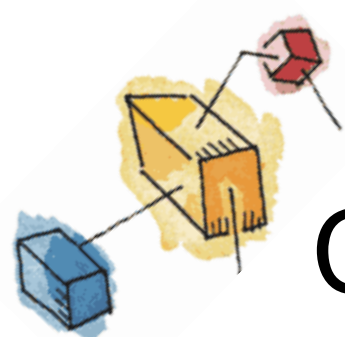


# Signals defined for UNIX SVR4.



Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure



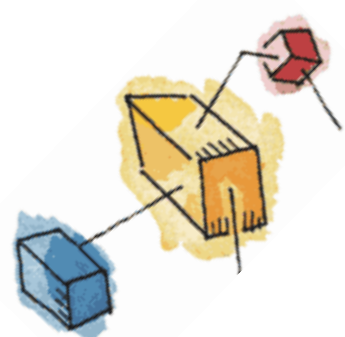


# Linux Kernel

## Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus
  - Atomic operations
  - Spinlocks
  - Semaphores (slightly different to SVR4)
  - Barriers

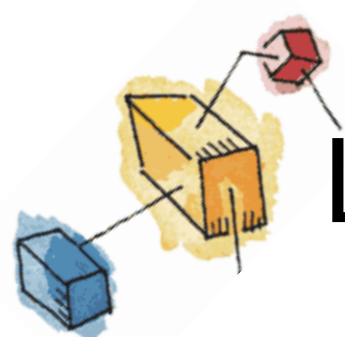




# Atomic Operations

- Atomic operations execute without interruption and without interference
- Two types:
  - Integer operations – operating on an integer variable
  - Bitmap operations – operating on one bit in a bitmap

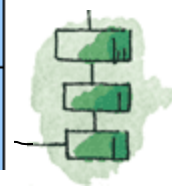


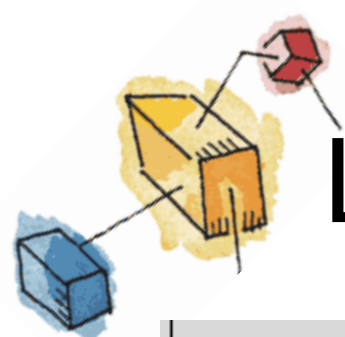


# Linux Atomic Operations

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
<code>ATOMIC_INIT (int i)</code>	At declaration: initialize an <code>atomic_t</code> to <code>i</code>
<code>int atomic_read(atomic_t *v)</code>	Read integer value of <code>v</code>
<code>void atomic_set(atomic_t *v, int i)</code>	Set the value of <code>v</code> to integer <code>i</code>
<code>void atomic_add(int i, atomic_t *v)</code>	Add <code>i</code> to <code>v</code>
<code>void atomic_sub(int i, atomic_t *v)</code>	Subtract <code>i</code> from <code>v</code>
<code>void atomic_inc(atomic_t *v)</code>	Add 1 to <code>v</code>
<code>void atomic_dec(atomic_t *v)</code>	Subtract 1 from <code>v</code>
<code>int atomic_sub_and_test(int i, atomic_t *v)</code>	Subtract <code>i</code> from <code>v</code> ; return 1 if the result is zero; return 0 otherwise
<code>int atomic_add_negative(int i, atomic_t *v)</code>	Add <code>i</code> to <code>v</code> ; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
<code>int atomic_dec_and_test(atomic_t *v)</code>	Subtract 1 from <code>v</code> ; return 1 if the result is zero; return 0 otherwise
<code>int atomic_inc_and_test(atomic_t *v)</code>	Add 1 to <code>v</code> ; return 1 if the result is zero; return 0 otherwise





# Linux Atomic Operations

Atomic Bitmap Operations	
<code>void set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr
<code>void clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr
<code>void change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by addr
<code>int test_and_set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_and_clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_and_change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_bit(int nr, void *addr)</code>	Return the value of bit nr in the bitmap pointed to by addr





# Spinlock



- Only one thread at a time can acquire a spinlock.
  - Any other thread will keep trying (spinning) until it can acquire the lock.
- A spinlock is an integer
  - If 0, the thread sets the value to 1 and enters its critical section.
  - If the value is nonzero, the thread continually checks the value until it is zero.





# Linux Spinlocks

<code>void spin_lock(spinlock_t *lock)</code>	Acquires the specified lock, spinning if needed until it is available
<code>void spin_lock_irq(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables interrupts on the local processor
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Like <code>spin_lock_irq</code> , but also saves the current interrupt state in flags
<code>void spin_lock_bh(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables the execution of all bottom halves
<code>void spin_unlock(spinlock_t *lock)</code>	Releases given lock
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Releases given lock and enables local interrupts
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Releases given lock and restores local interrupts to given previous state
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Releases given lock and enables bottom halves
<code>void spin_lock_init(spinlock_t *lock)</code>	Initializes given spinlock
<code>int spin_trylock(spinlock_t *lock)</code>	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
<code>int spin_is_locked(spinlock_t *lock)</code>	Returns nonzero if lock is currently held and zero otherwise





# Semaphores

- Similar to UNIX SVR4 but also provides an implementation of semaphores for its own use.
- Three types of kernel semaphores:
  - Binary semaphores
  - counting semaphores,
  - reader-writer semaphores.









# Linux Semaphores

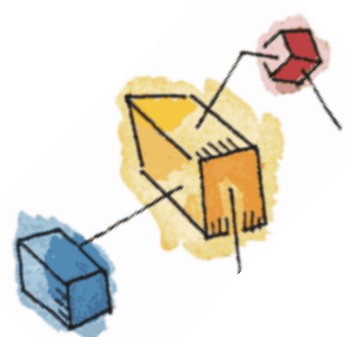
## Traditional Semaphores

<code>void sema_init(struct semaphore *sem, int count)</code>	Initializes the dynamically created semaphore to the given count
<code>void init_MUTEX(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 0 (initially locked)
<code>void down(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
<code>int down_interruptible(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
<code>int down_trylock(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
<code>void up(struct semaphore *sem)</code>	Releases the given semaphore

## Reader-Writer Semaphores

<code>void init_rwsem(struct rw_semaphore, *rwsem)</code>	Initializes the dynamically created semaphore with a count of 1
<code>void down_read(struct rw_semaphore, *rwsem)</code>	Down operation for readers
<code>void up_read(struct rw_semaphore, *rwsem)</code>	Up operation for readers
<code>void down_write(struct rw_semaphore, *rwsem)</code>	Down operation for writers
<code>void up_write(struct rw_semaphore, *rwsem)</code>	Up operation for writers





# Barriers

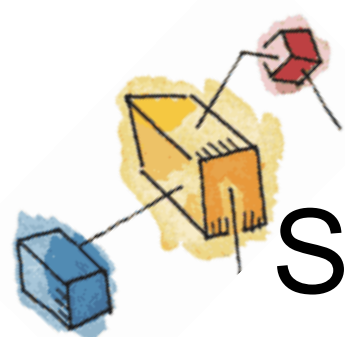
- To enforce the order in which instructions are executed, Linux provides the memory barrier facility.

**Table 6.6 Linux Memory Barrier Operations**

<code>rmb()</code>	Prevents loads from being reordered across the barrier
<code>wmb()</code>	Prevents stores from being reordered across the barrier
<code>mb()</code>	Prevents loads and stores from being reordered across the barrier
<code>Barrier()</code>	Prevents the compiler from reordering loads or stores across the barrier
<code>smp_rmb()</code>	On SMP, provides a <code>rmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_wmb()</code>	On SMP, provides a <code>wmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_mb()</code>	On SMP, provides a <code>mb()</code> and on UP provides a <code>barrier()</code>

SMP = symmetric multiprocessor  
UP = uniprocessor





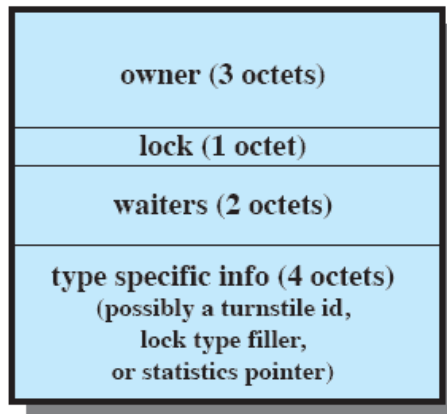
# Solaris Thread Synchronization Primitives

- In addition to the concurrency mechanisms of UNIX SVR4
  - Mutual exclusion (mutex) locks
  - Semaphores
  - Multiple readers, single writer (readers/writer) locks
  - Condition variables

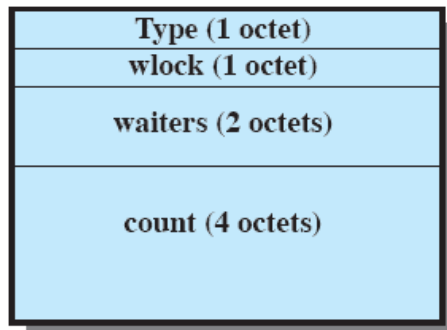




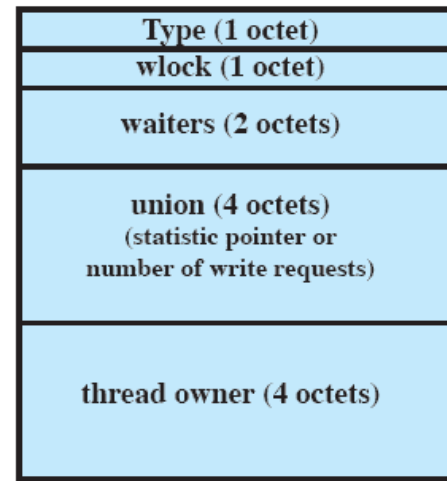
# Solaris Synchronization Data Structures



(a) MUTEX lock



(b) Semaphore



(c) Reader/writer lock



(d) Condition variable



Figure 6.15 Solaris Synchronization Data Structures



An illustration showing a central yellow rectangular resource. A blue cube is on the left, a red cube is on the top right, and a red cube is on the bottom right. Lines connect these cubes to the central resource, representing threads attempting to access it.

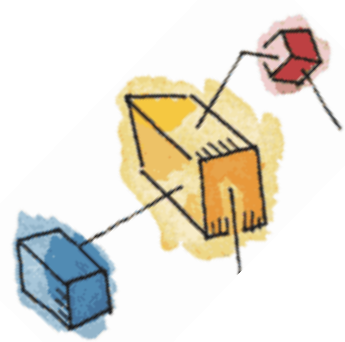
# MUTEX Lock

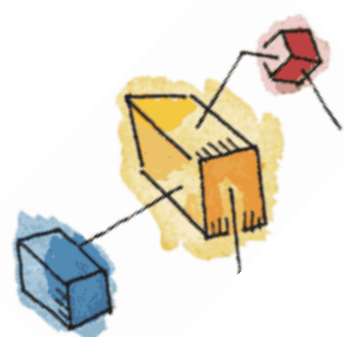
- A mutex is used to ensure only one thread at a time can access the resource protected by the mutex.
- The thread that locks the mutex must be the one that unlocks it.



# Semaphores and Read/Write locks

- Solaris provides classic counting semaphores.
- The readers/writer lock allows multiple threads to have simultaneous read-only access to an object protected by the lock.
  - It also allows a single thread to access the object for writing at one time, while excluding all readers.

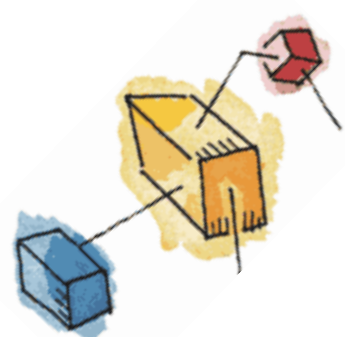




# Condition Variables

- A condition variable is used to wait until a particular condition is true.
- Condition variables must be used in conjunction with a mutex lock.





# Windows concurrency mechanisms

- Windows provides synchronization among threads as part of the object architecture.
- Important methods of synchronization are
  - Executive dispatcher objects (using Wait functions),
  - user mode critical sections,
  - slim reader-writer locks, and
  - condition variables.







# Wait Functions



- The wait functions allow a thread to block its own execution.
  - The wait functions do not return until the specified criteria have been met.
  - The type of wait function determines the set of criteria used.





# Dispatcher Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released



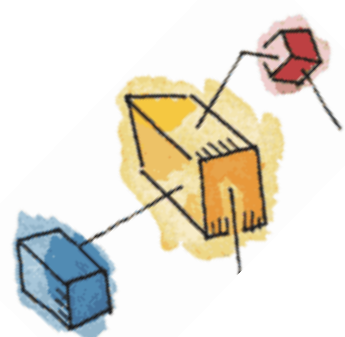
*Note:* Shaded rows correspond to objects that exist for the sole purpose of synchronization.



# Critical Sections

- Similar mechanism to mutex
  - except that critical sections can be used only by the threads of a single process.
- If the system is a multiprocessor, the code will attempt to acquire a spin-lock.
  - As a last resort, if the spinlock cannot be acquired, a dispatcher object is used to block the thread so that the Kernel can dispatch another thread onto the processor.

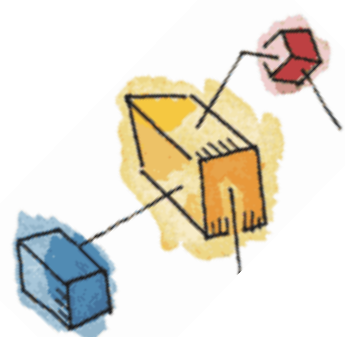




# Slim Read-Writer Locks

- Windows Vista added a user mode reader-writer.
- The readerwriter lock enters the kernel to block only after attempting to use a spin-lock.
- ‘Slim’ as it normally only requires allocation of a single pointer-sized piece of memory.

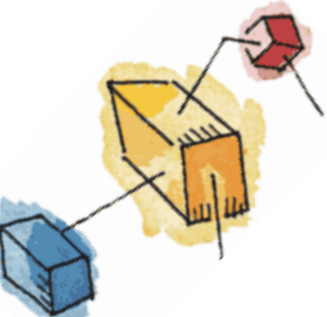




# Condition Variables

- Windows Vista also added condition variables.
- The process must declare and initialise a `CONDITION_VARIABLE`
- Used with either critical sections or SRW locks





# Windows/Linux Comparison

Windows	Linux
Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying wait/signal mechanism	Common synchronization primitives, such as semaphores, mutexes, spinlocks, timers, based on an underlying sleep/wakeup mechanism
Many kernel objects are also dispatcher objects, meaning that threads can synchronize with them using a common event mechanism, available at user-mode. Process and thread termination are events, I/O completion is an event	
Threads can wait on multiple dispatcher objects at the same time	Processes can use the select() system call to wait on I/O from up to 64 file descriptors
User-mode reader/writer locks and condition variables are supported	User-mode reader/writer locks and condition variables are supported
Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported	Many hardware atomic operations, such as atomic increment/decrement, and compare-and-swap, are supported
A non-locking atomic LIFO queue, called an SLIST, is supported using compare-and-swap; widely used in the OS and also available to user programs	







# Windows/Linux Comparison cont.

Windows	Linux
A large variety of synchronization mechanisms exist within the kernel to improve scalability. Many are based on simple compare-and-swap mechanisms, such as push-locks and fast references of objects	
Named pipes, and sockets support remote procedure calls (RPCs), as does an efficient Local Procedure Call mechanism (ALPC), used within a local system. ALPC is used heavily for communicating between clients and local services	Named pipes, and sockets support remote procedure calls (RPCs)
Asynchronous Procedure Calls (APCs) are used heavily within the kernel to get threads to act upon themselves (e.g. termination and I/O completion use APCs since these operations are easier to implement in the context of a thread rather than cross-thread). APCs are also available for user-mode, but user-mode APCs are only delivered when a user-mode thread blocks in the kernel	Unix supports a general signal mechanism for communication between processes. Signals are modeled on hardware interrupts and can be delivered at any time that they are not blocked by the receiving process; like with hardware interrupts, signal semantics are complicated by multi-threading
Hardware support for deferring interrupt processing until the interrupt level has dropped is provided by the Deferred Procedure Call (DPC) control object	Uses tasklets to defer interrupt processing until the interrupt level has dropped

