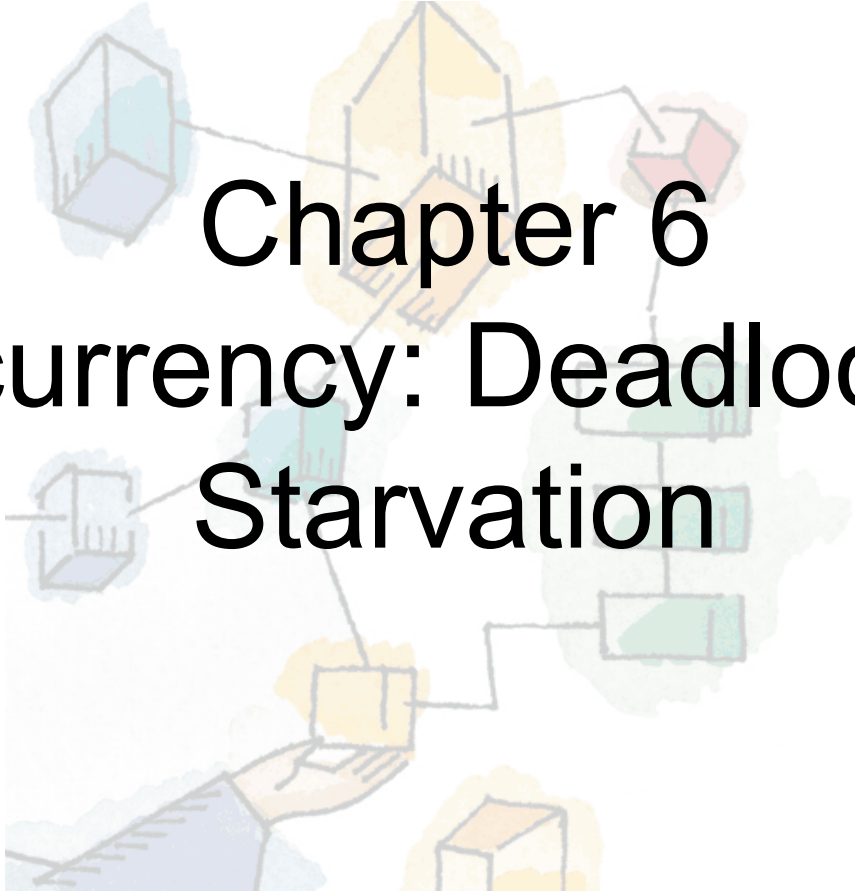


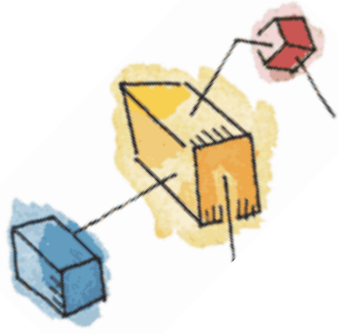
*Operating Systems:  
Internals and Design Principles, 6/E*  
William Stallings



# Chapter 6

## Concurrency: Deadlock and Starvation

Patricia Roy  
Manatee Community College, Venice, FL  
©2008, Prentice Hall



# Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes



# Deadlock

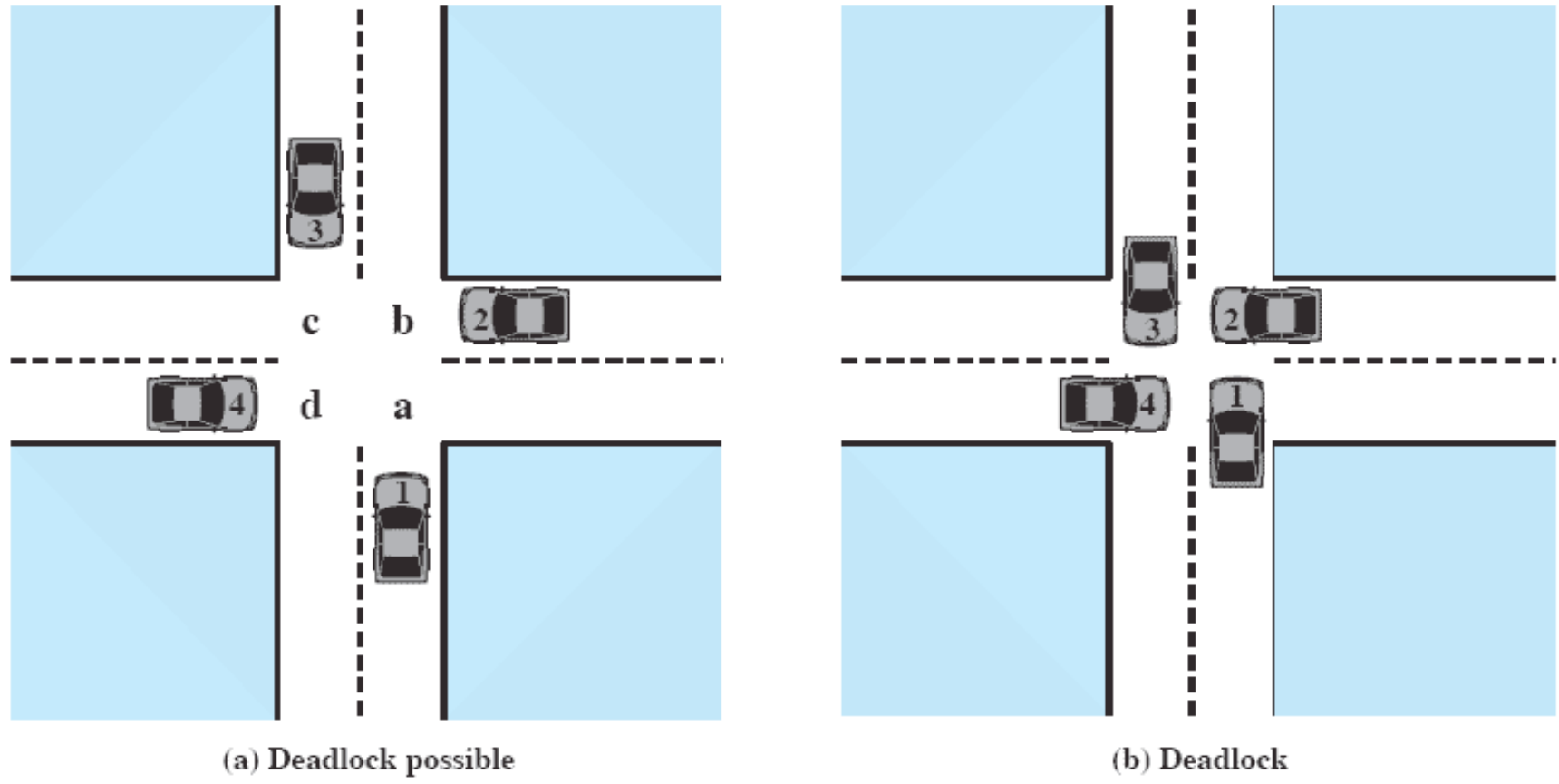
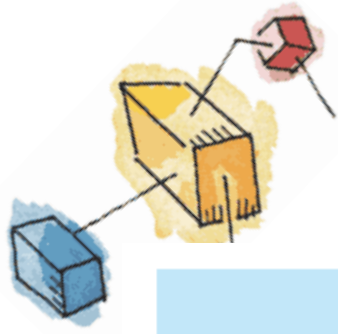


Figure 6.1 Illustration of Deadlock



# Deadlock

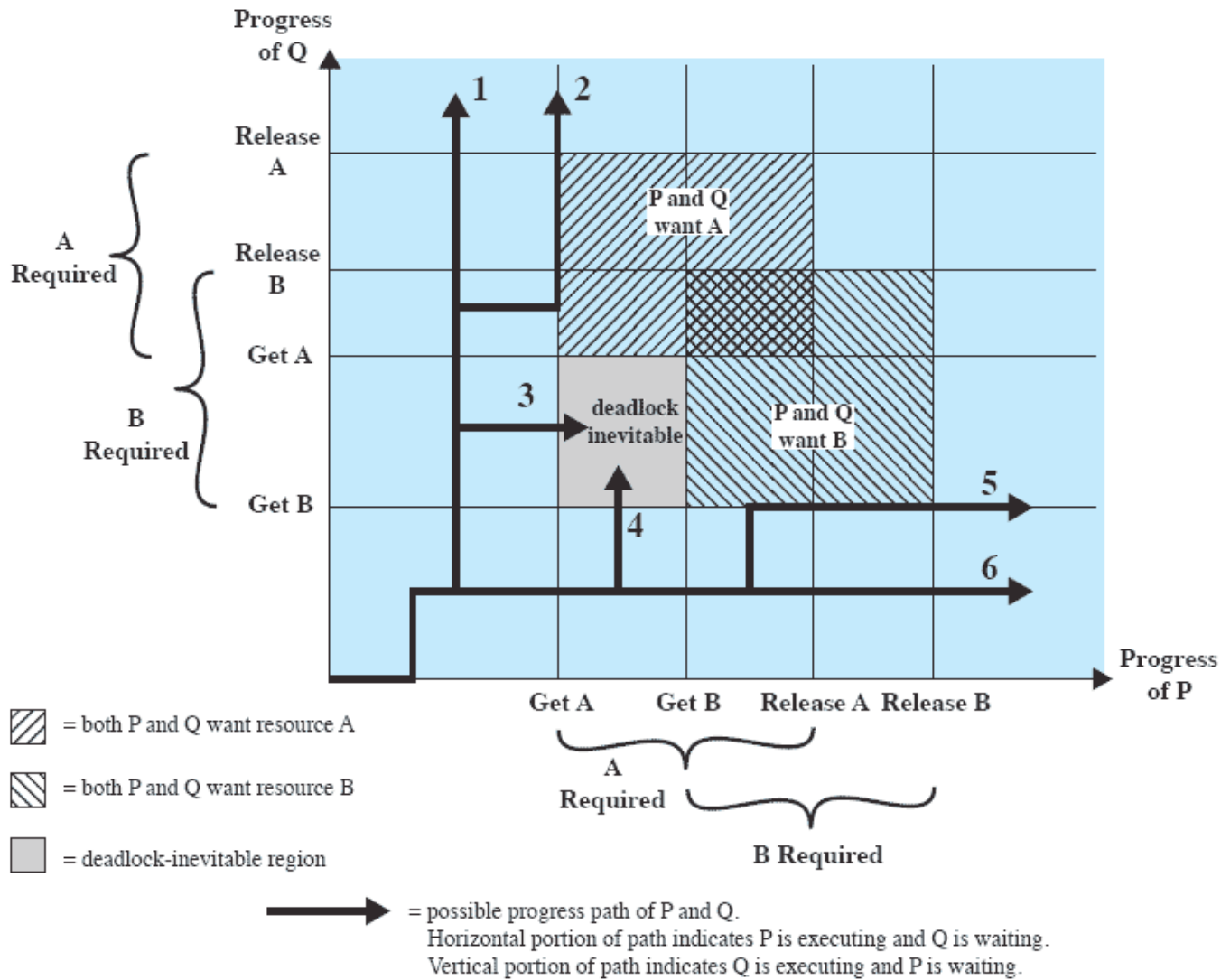
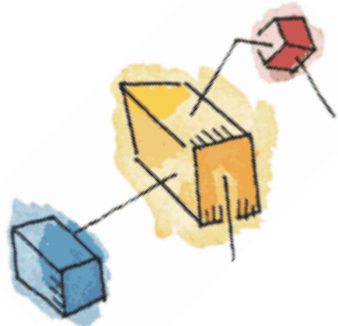


Figure 6.2 Example of Deadlock

# Deadlock

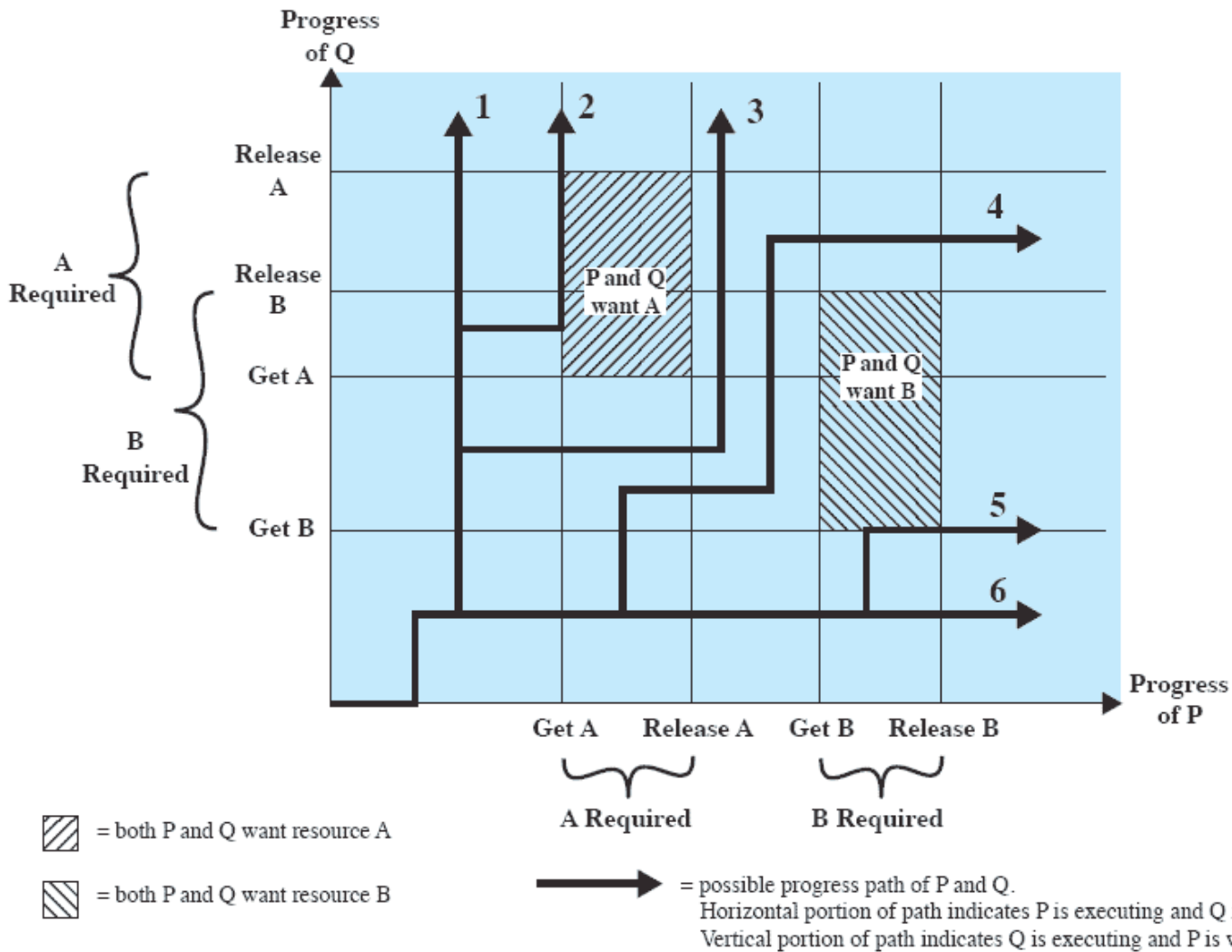
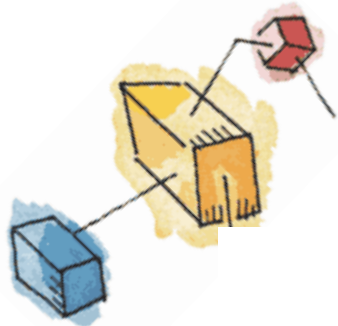
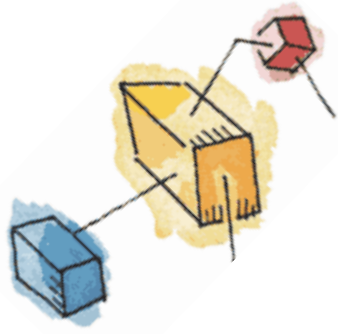


Figure 6.3 Example of No Deadlock [BACO03]

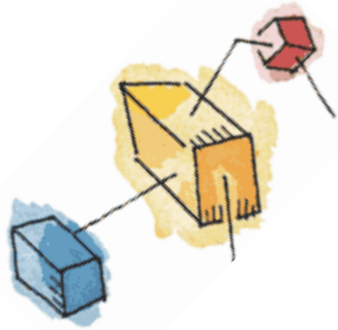




# Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes

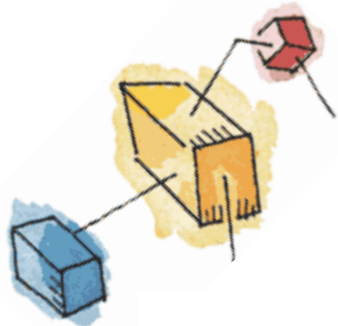




# Reusable Resources

- 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





# Reusable Resources

**Process P**

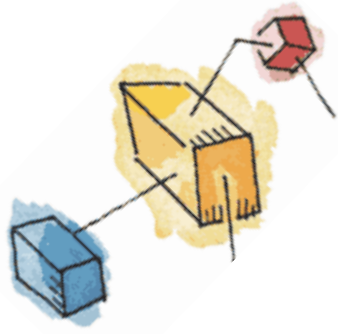
**Process Q**

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)

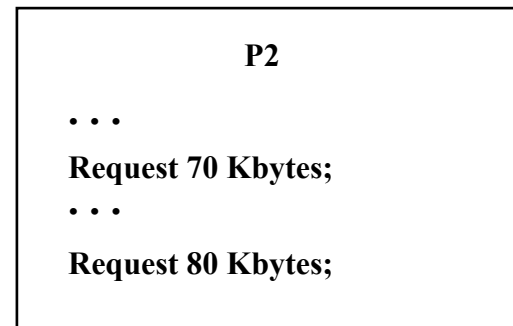
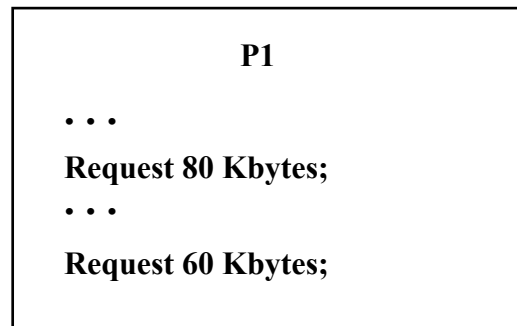
**Figure 6.4 Example of Two Processes Competing for Reusable Resources**





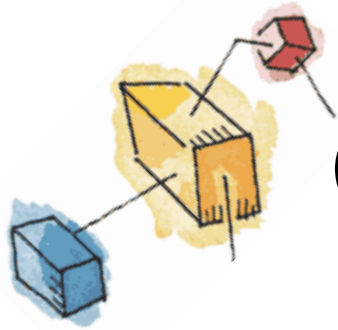
# Reusable Resources

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



- Deadlock occurs if both processes progress to their second request

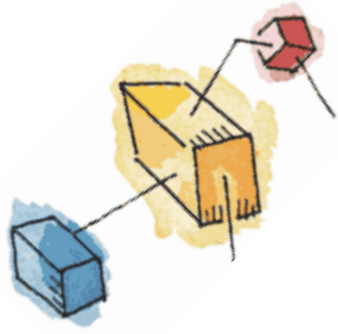




# Consumable Resources

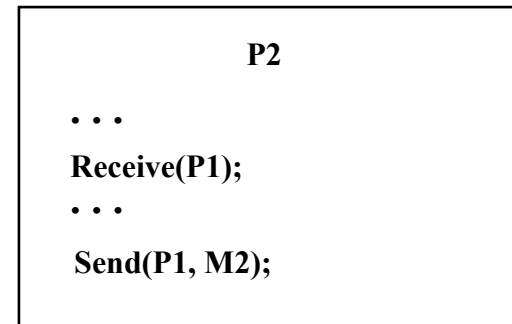
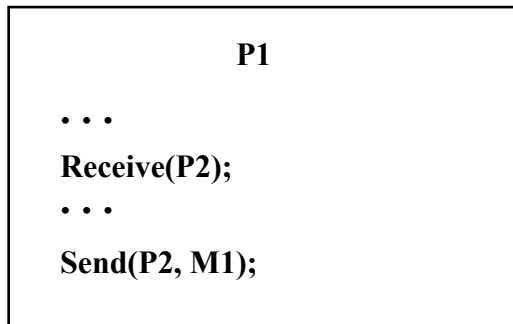
- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

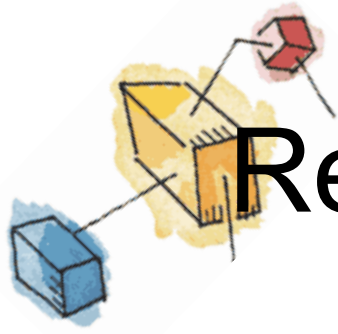




# Example of Deadlock

- Deadlock occurs if receives blocking





# Resource Allocation Graphs

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

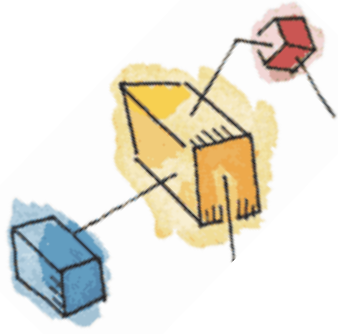


(a) Resource is requested



(b) Resource is held

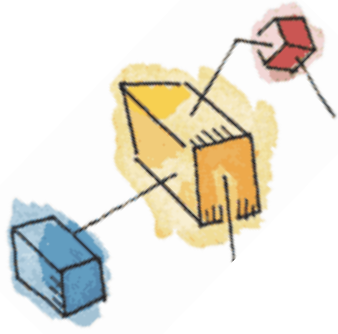




# Conditions for Deadlock

- 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

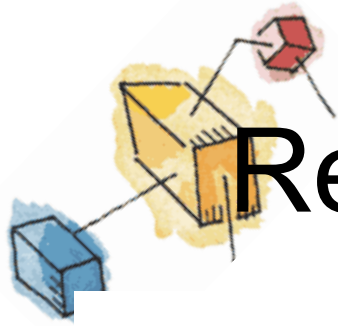




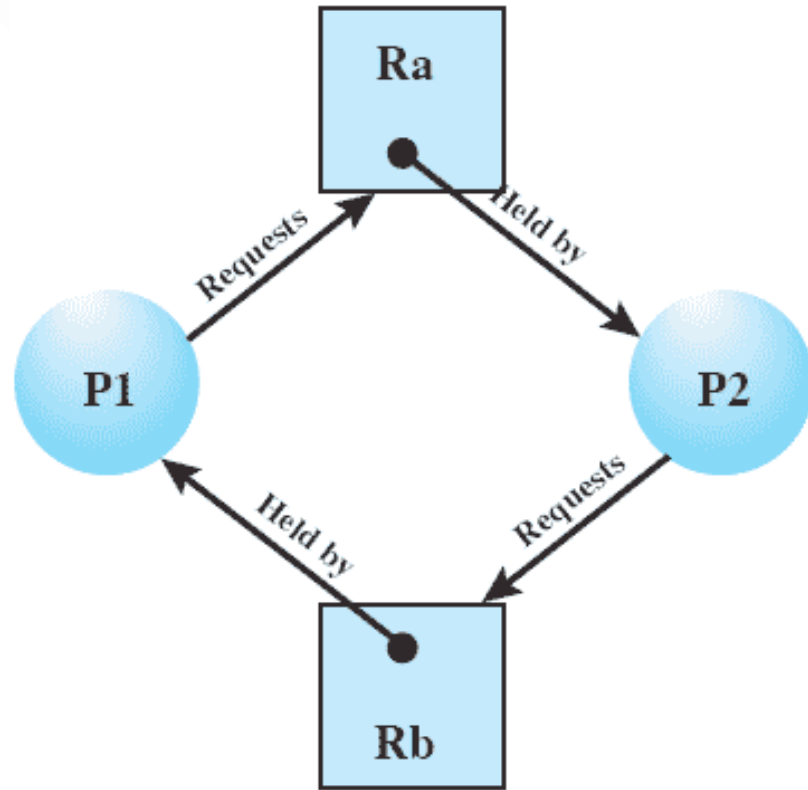
# Conditions for Deadlock

- No preemption
  - No resource can be forcibly removed from a process holding it
- 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

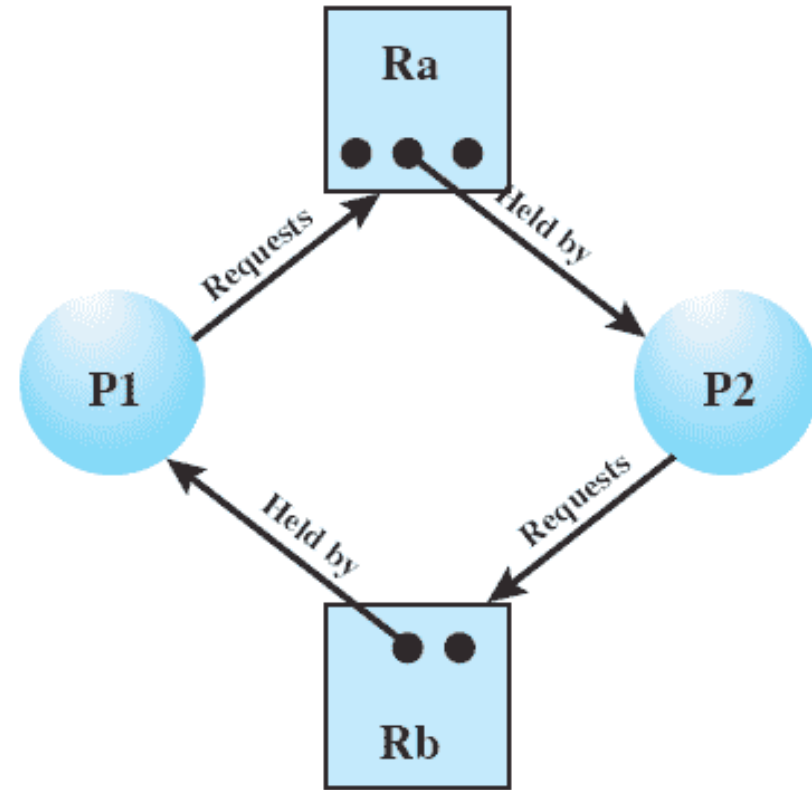




# Resource Allocation Graphs

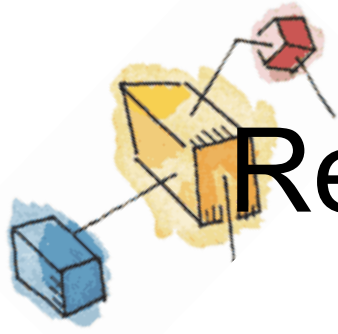


(c) Circular wait



(d) No deadlock





# Resource Allocation Graphs

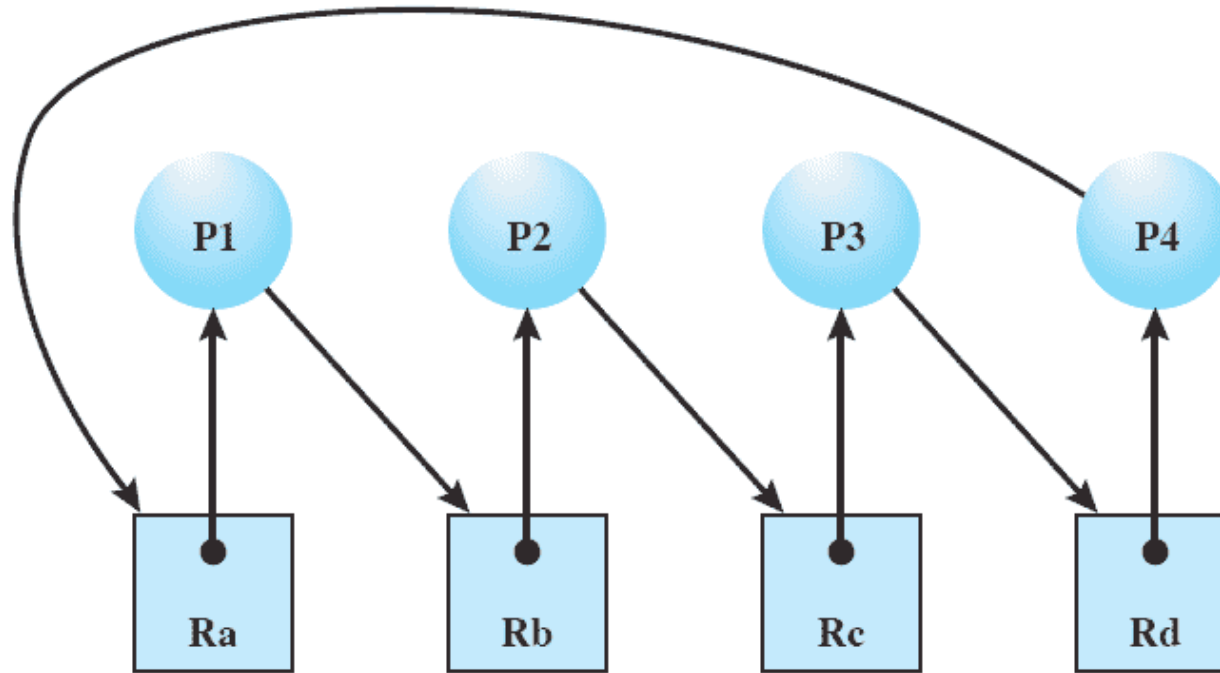
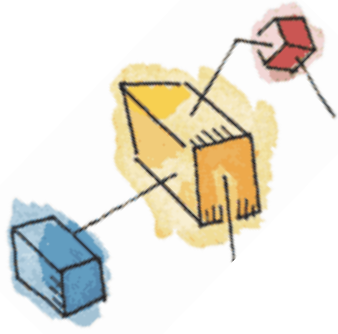


Figure 6.6 Resource Allocation Graph for Figure 6.1b

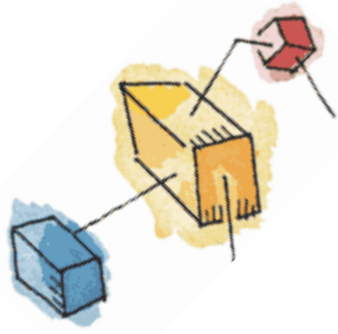




# Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

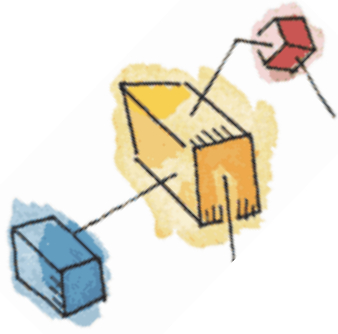




# Existence of Deadlock

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

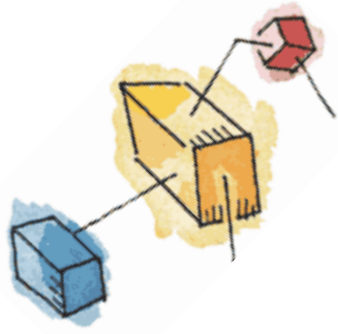




# Deadlock Prevention

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

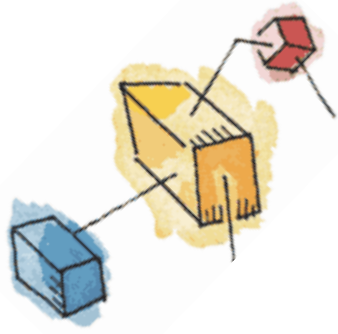




# Deadlock Prevention

- 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

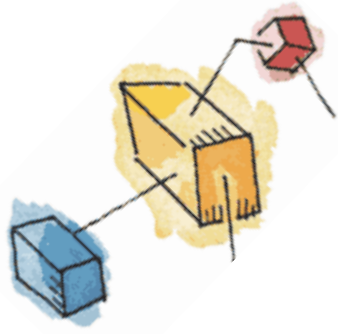




# Deadlock Avoidance

- 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

- Do not start a process if its demands might lead to deadlock
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock

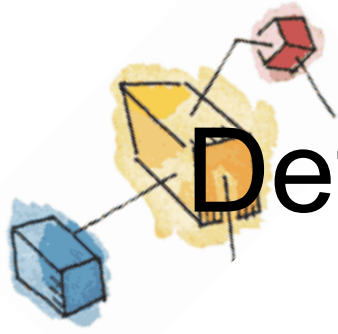




# Resource Allocation Denial

- Referred to as the banker's algorithm
- 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 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

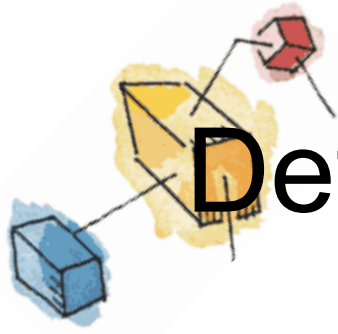
Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state





# Determination of a Safe State

	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

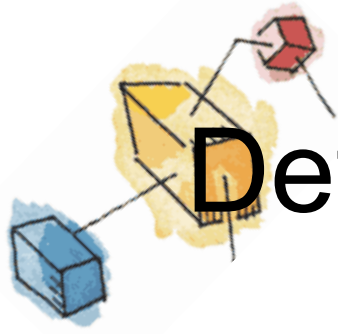
Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion





# Determination of a Safe State

	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

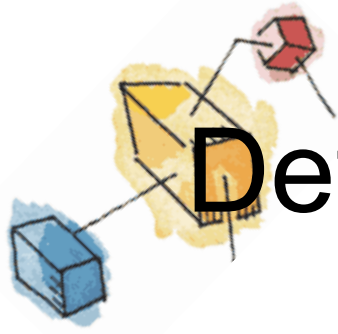
Resource vector R

R1	R2	R3
7	2	3

Available vector V

(e) P1 runs to completion





# Determination of a Safe State

	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

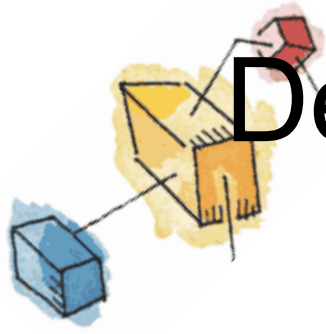
Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion





# 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

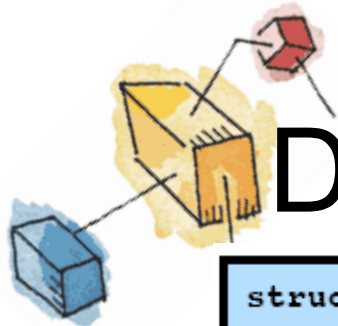
Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

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





# Deadlock Avoidance Logic

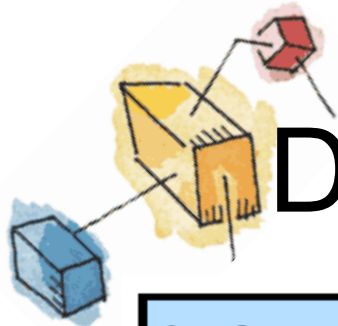
```
struct state {  
    int resource[m];  
    int available[m];  
    int claim[n][m];  
    int alloc[n][m];  
}
```

(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





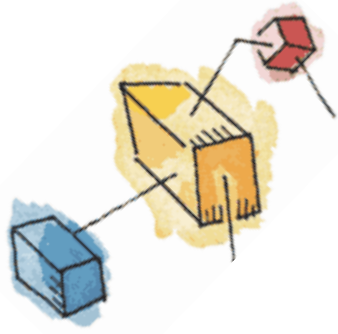
# Deadlock Avoidance Logic

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

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

Figure 6.9 Deadlock Avoidance Logic

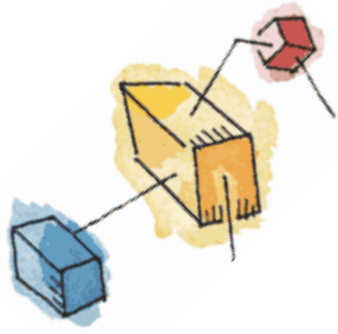




# Deadlock Avoidance

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





# 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

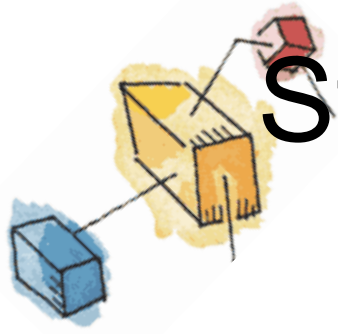
  

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

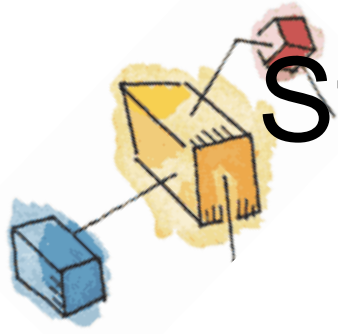




# Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur

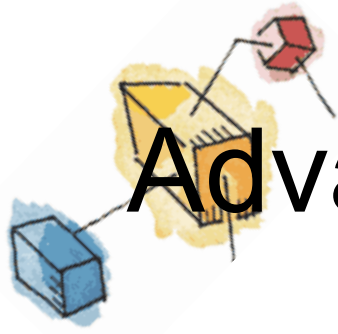




# Strategies Once Deadlock Detected

- 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>



# Dining Philosophers Problem

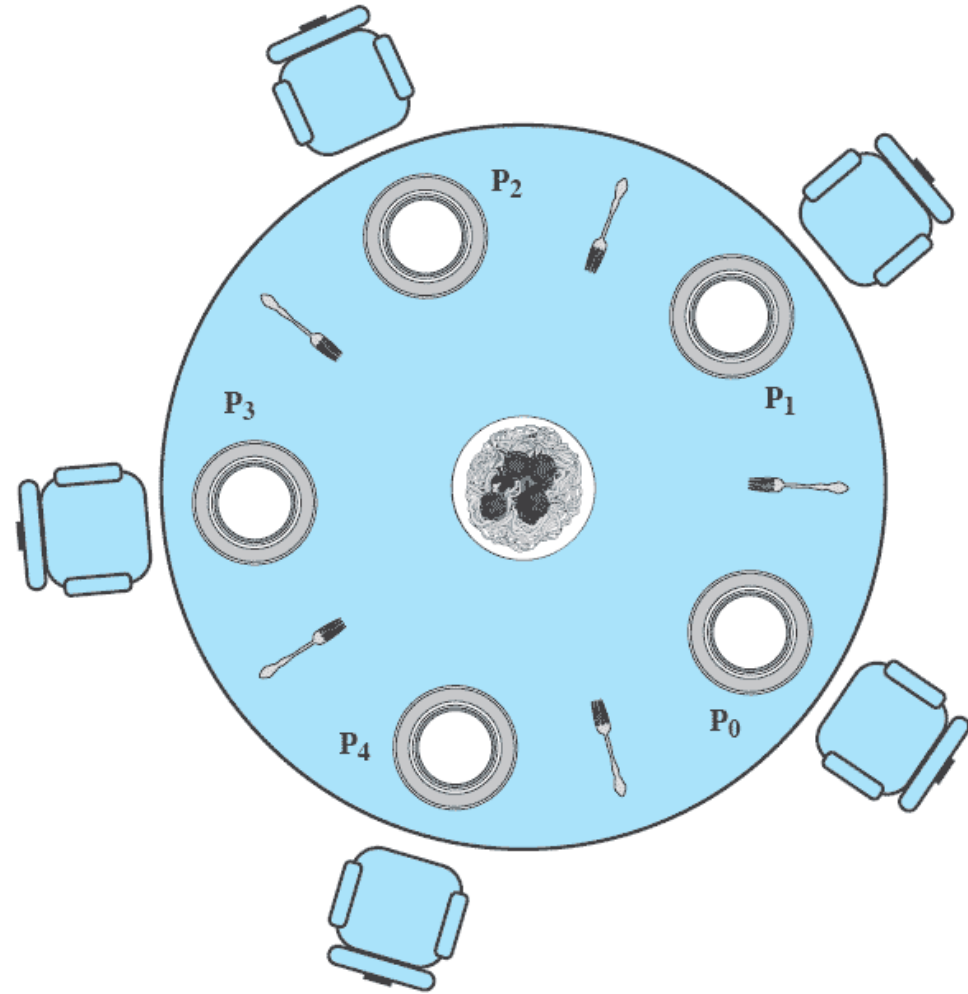



Figure 6.11 Dining Arrangement for Philosophers




# Dining Philosophers Problem

```
/* 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





# Dining Philosophers Problem

```
/* 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





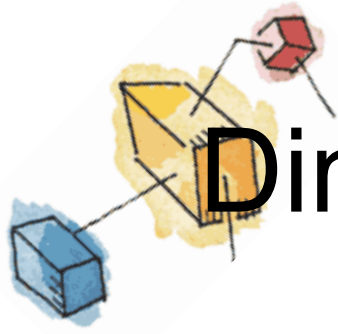
# Dining Philosophers Problem

```
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]);
}
}
```





# Dining Philosophers Problem

```
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



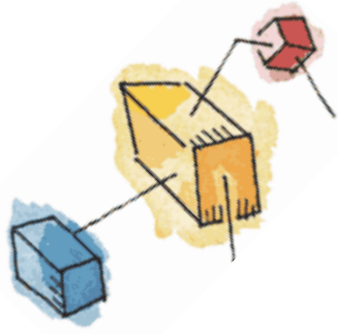


# UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

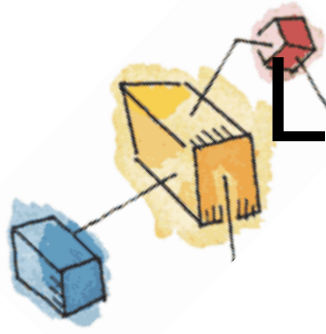


# UNIX Signals



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
- Atomic operations execute without interruption and without interference

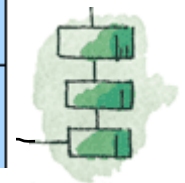


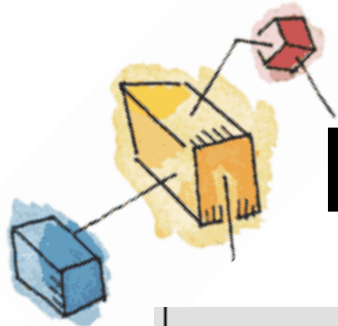


# Linux Atomic Operations

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

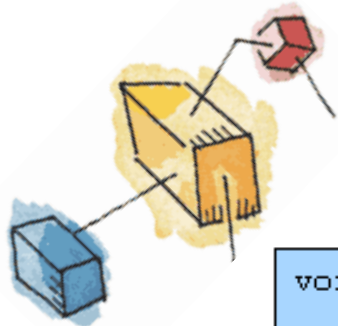




# Linux Atomic Operations

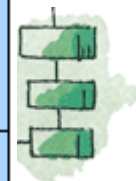
Atomic Bitmap Operations	
<code>void set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr
<code>void clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr
<code>void change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by 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

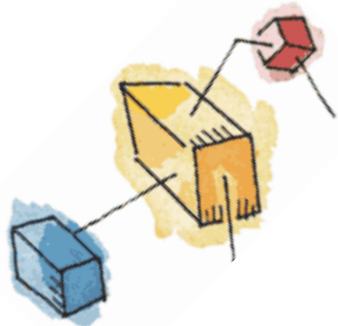




# Linux Spinlocks

<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

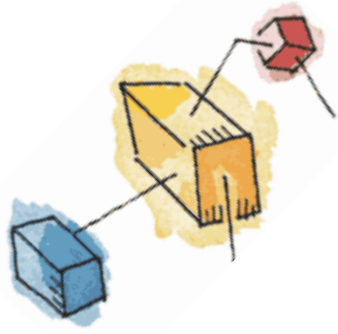




# Linux Semaphores

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 <code>-EINTR</code> 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





# Linux Memory Barrier Operations

**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



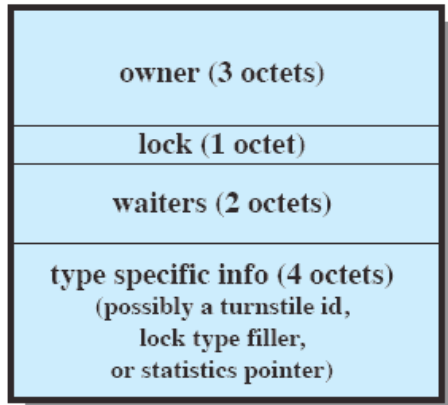
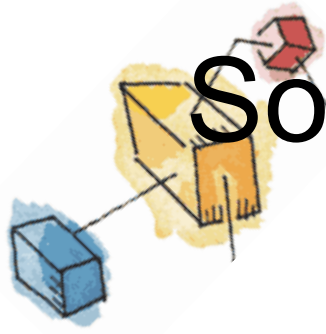


# Solaris Thread Synchronization Primitives

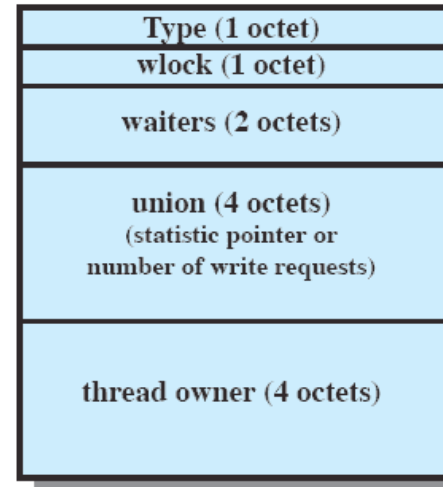
- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables



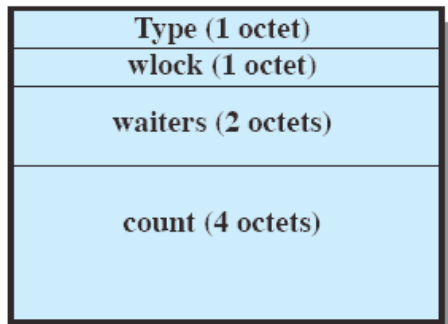
# Solaris Synchronization Data Structures



(a) MUTEX lock



(c) Reader/writer lock



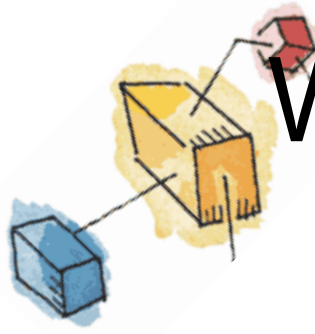
(b) Semaphore



(d) Condition variable



Figure 6.15 Solaris Synchronization Data Structures



# Windows Synchronization 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.