

*Operating  
Systems:  
Internals  
and Design  
Principles*

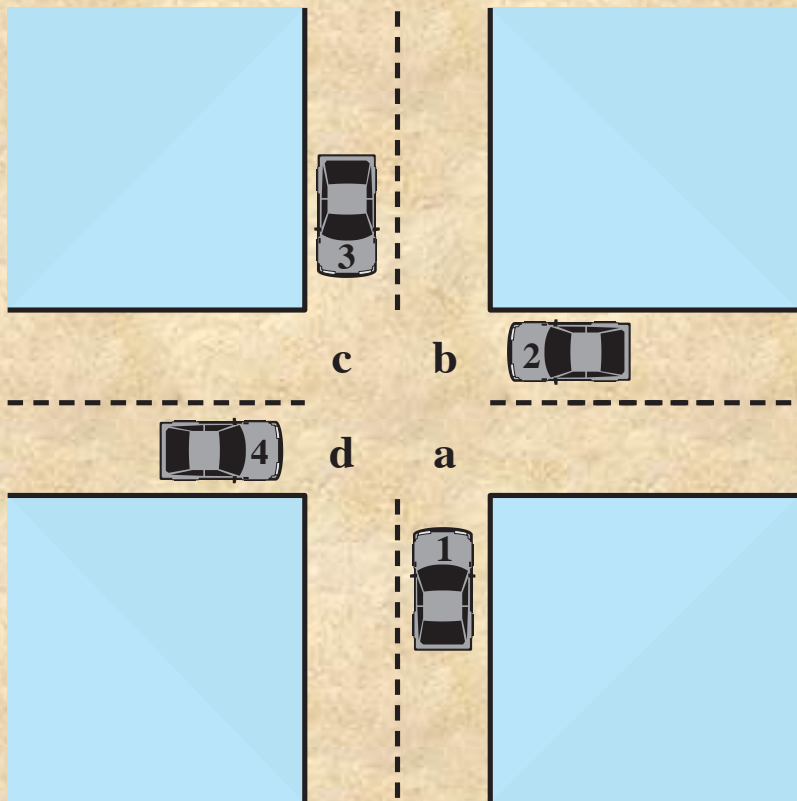
# Chapter 6 Concurrency: Deadlock and Starvation

Eighth Edition  
By William Stallings

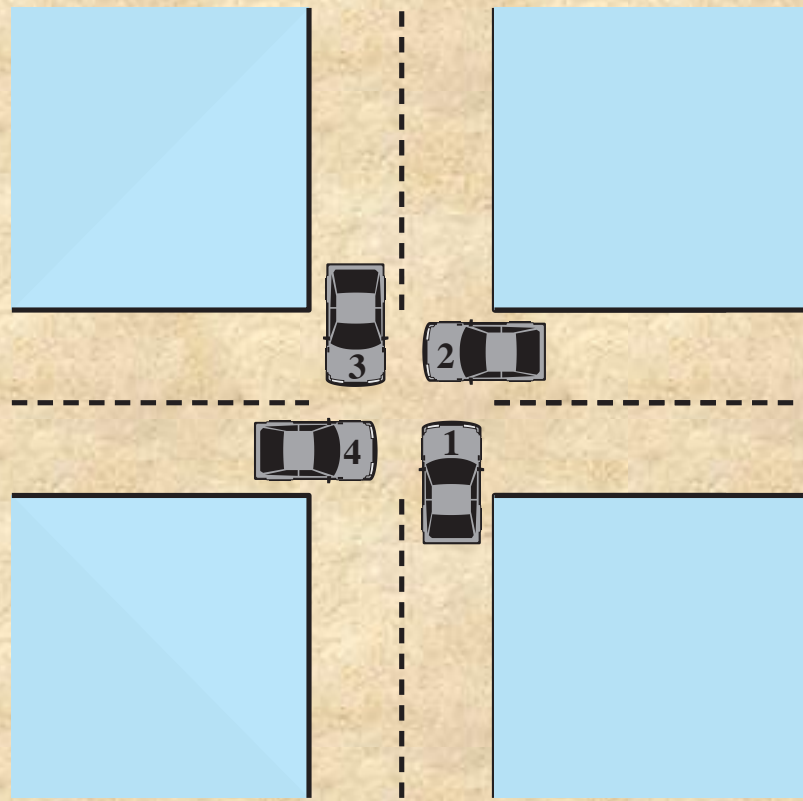
# Deadlock

- 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
- Permanent
- No efficient solution



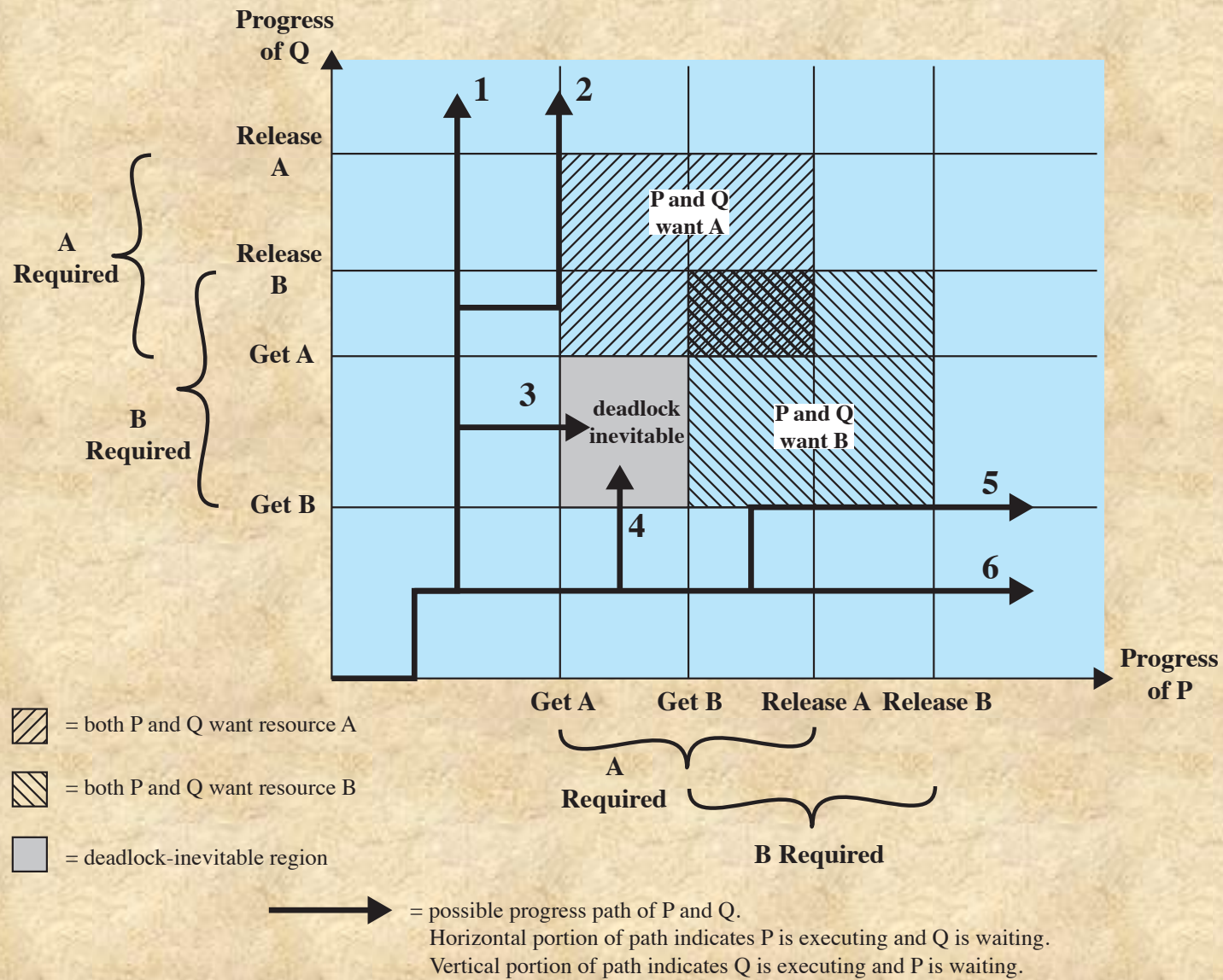


(a) Deadlock possible

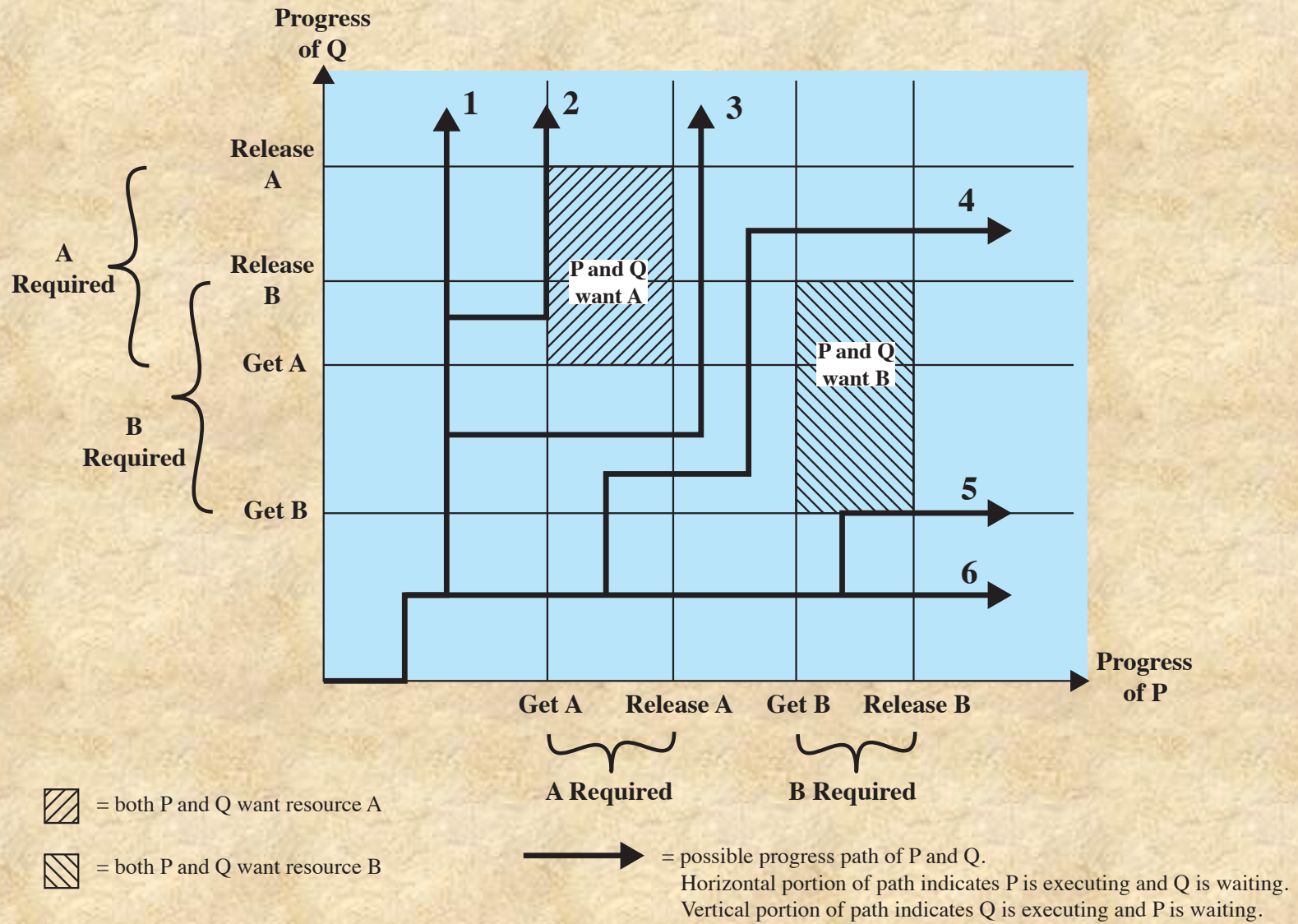


(b) Deadlock

Figure 6.1 Illustration of Deadlock



**Figure 6.2 Example of Deadlock**

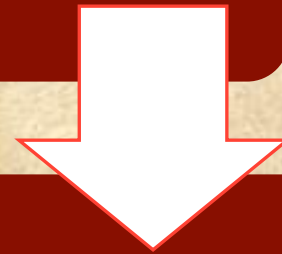


**Figure 6.3 Example of No Deadlock**

# Resource Categories

## Reusable

- can be safely used by only one process at a time and is not depleted by that use
- processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores



## Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information
  - in I/O buffers

<b>Process P</b>	
<b>Step</b>	<b>Action</b>
p <sub>0</sub>	Request (D)
p <sub>1</sub>	Lock (D)
p <sub>2</sub>	Request (T)
p <sub>3</sub>	Lock (T)
p <sub>4</sub>	Perform function
p <sub>5</sub>	Unlock (D)
p <sub>6</sub>	Unlock (T)

<b>Process Q</b>	
<b>Step</b>	<b>Action</b>
q <sub>0</sub>	Request (T)
q <sub>1</sub>	Lock (T)
q <sub>2</sub>	Request (D)
q <sub>3</sub>	Lock (D)
q <sub>4</sub>	Perform function
q <sub>5</sub>	Unlock (T)
q <sub>6</sub>	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:

**P1**  
...  
**Request 80 Kbytes;**  
...  
**Request 60 Kbytes;**

**P2**  
...  
**Request 70 Kbytes;**  
...  
**Request 80 Kbytes;**

- Deadlock occurs if both processes progress to their second request



# Consumable Resources 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);

- Deadlock occurs if the Receive is blocking

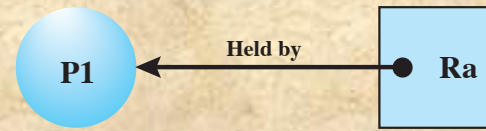
Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> <li>•Works well for processes that perform a single burst of activity</li> <li>•No preemption 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>

**Table 6.1**

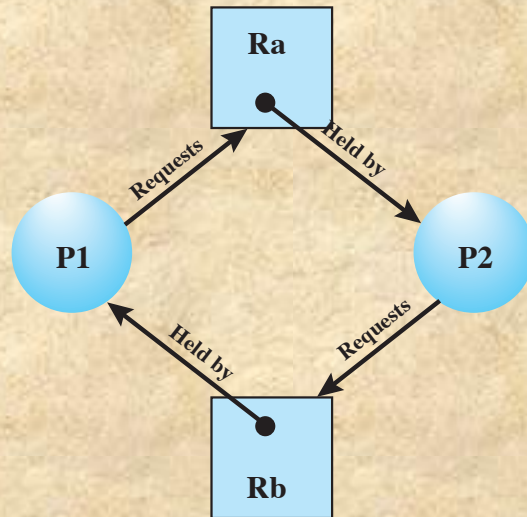
**Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]**



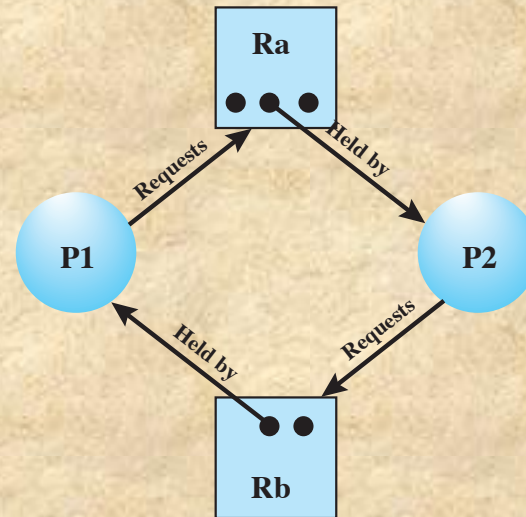
(a) Resource is requested



(b) Resource is held

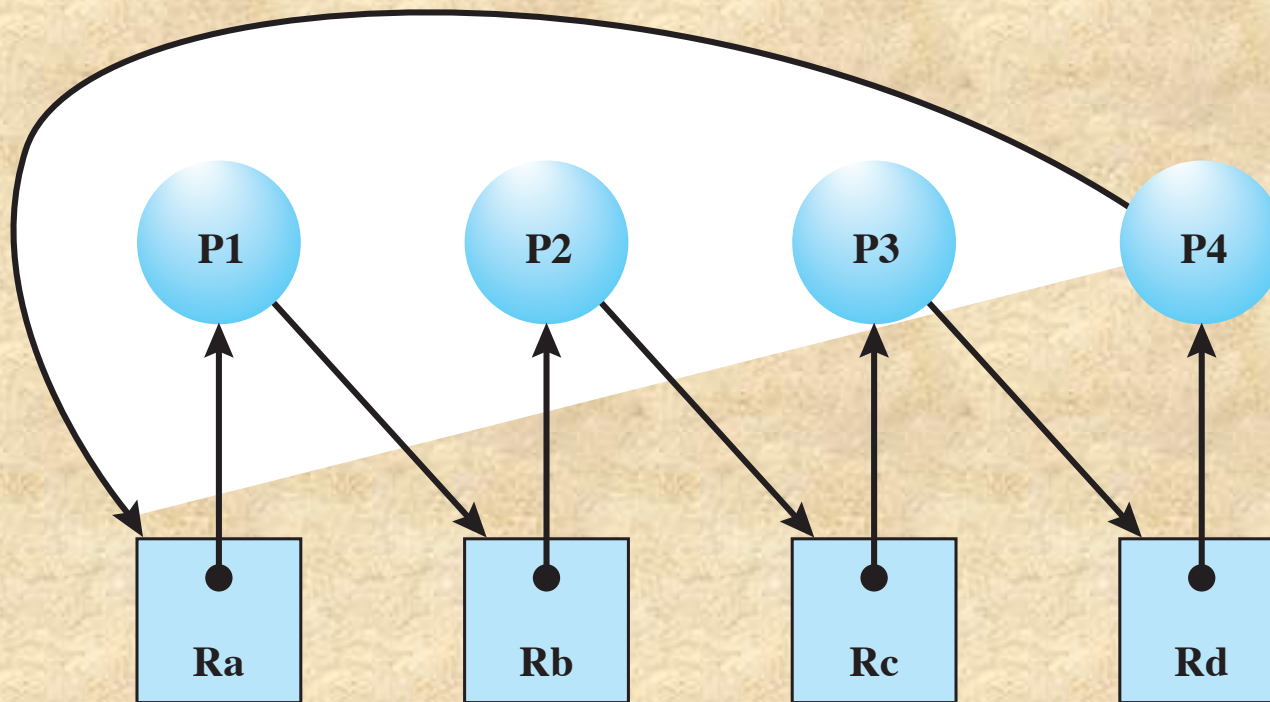


(c) Circular wait



(d) No deadlock

Figure 6.5 Examples of Resource Allocation Graphs



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

# 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

## No Pre-emption

- 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

# Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

## Prevent Deadlock

- adopt a policy that eliminates one of the conditions

## Avoid Deadlock

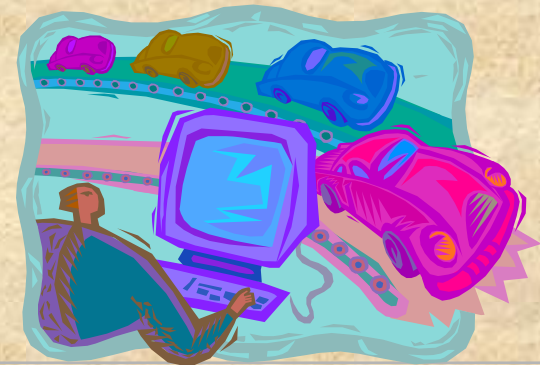
- make the appropriate dynamic choices based on the current state of resource allocation

## Detect Deadlock

- attempt to detect the presence of deadlock and take action to recover

# 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 Condition Prevention

## Mutual Exclusion

if access to a resource requires mutual exclusion then it must be supported by the OS

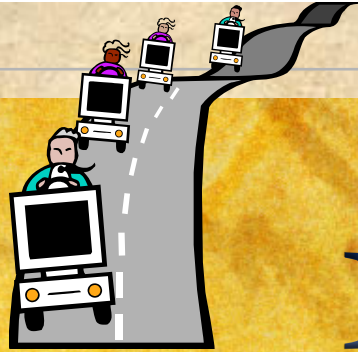
## Hold and Wait

require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously



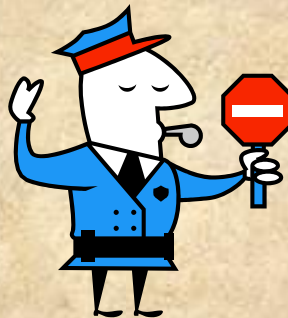
# Deadlock Condition Prevention

- No Preemption
  - if a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release 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

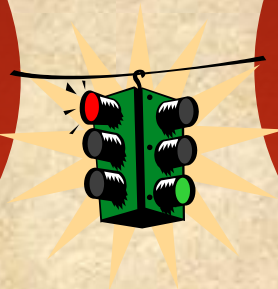
## Deadlock Avoidance

### Resource Allocation Denial

- do not grant an incremental resource request to a process if this allocation might lead to deadlock

### Process Initiation Denial

- do not start a process if its demands might lead to deadlock



# Resource Allocation Denial

- Referred to as the *banker's algorithm*
- *State* of the system reflects 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 a deadlock
- *Unsafe state* is a state that is not safe



	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

**Figure 6.7 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**

**Figure 6.7 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**

(c) **P1 runs to completion**

**Figure 6.7 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

**Figure 6.7 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	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

Figure 6.8 Determination of an Unsafe State

```

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

```

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);
}

```

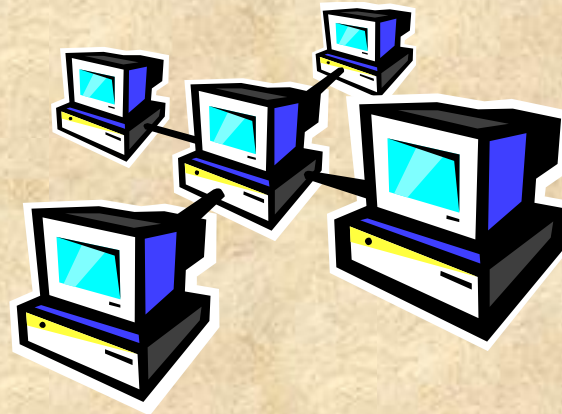
(c) test for safety algorithm (banker's algorithm)

## Figure 6.9

# Deadlock Avoidance Logic

# Deadlock Avoidance 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 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

# Deadlock Strategies

Deadlock prevention strategies are very conservative

- limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- resource requests are granted whenever possible

# Deadline Detection Algorithms

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur



## Advantages:

- it leads to early detection
- the algorithm is relatively simple



## Disadvantage

- frequent checks consume considerable processor time

	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

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



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>

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

# Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)
- No philosopher must starve to death (avoid deadlock and starvation)

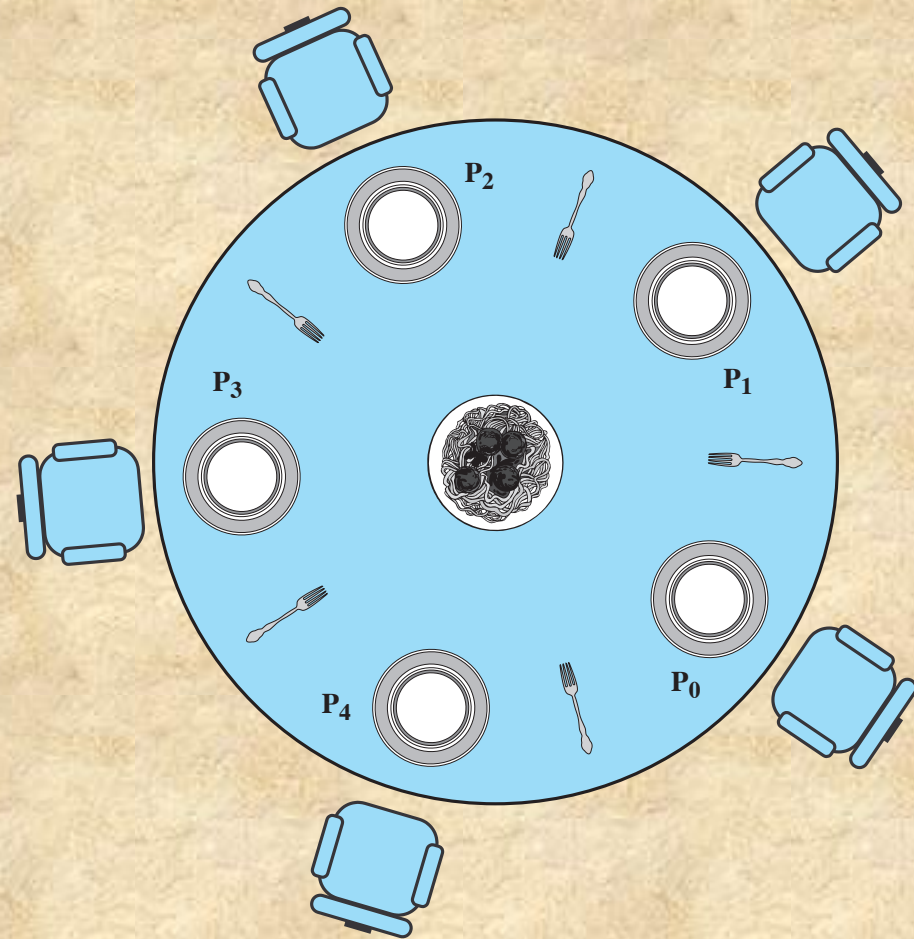


Figure 6.11 Dining Arrangement for Philosophers

```
/* 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

```
/* 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**

```

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 /* awoken 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 /* awoken a process waiting on this fork */
        csignal(ForkReady[right]);
}

```

```

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

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

Messages

Shared  
memory

Semaphores

Signals

# Pipes

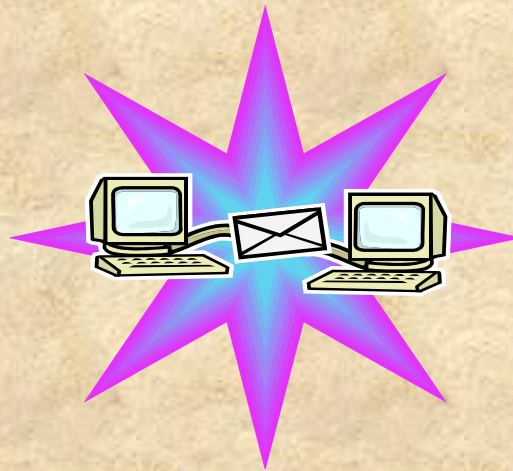
- Circular buffers allowing two processes to communicate on the producer-consumer model
  - first-in-first-out queue, written by one process and read by another

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

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

# Semaphores

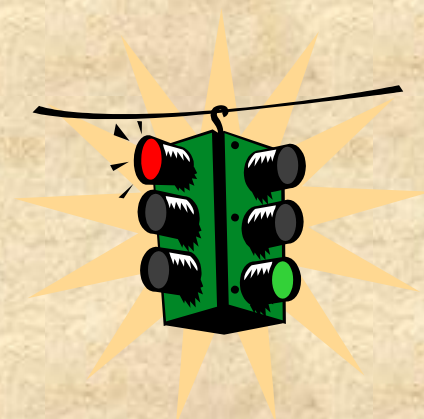
- Generalization of the `semWait` and `semSignal` primitives
  - no other process may access the semaphore until all operations have completed

## Consists of:

- current value of the semaphore
- process ID of the last process to operate on the semaphore
- number of processes waiting for the semaphore value to be greater than its current value
- number of processes waiting for the semaphore value to be zero

# Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
  - similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
  - performing some default action
  - executing a signal-handler function
  - ignoring the signal



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

**Table 6.2**  
**UNIX Signals**

(Table can be found on page 286 in textbook)

# Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus:

Barriers

Spinlocks

Atomic  
Operations

Semaphores

# Atomic Operations

- Atomic operations execute without interruption and without interference
- Simplest of the approaches to kernel synchronization
- Two types:



## Integer Operations

operate on an integer variable

typically used to implement counters

## Bitmap Operations

operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

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

**Table 6.3**

**Linux**

**Atomic**

**Operations**

(Table can be found on page 287 in textbook)

# Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time
  - any other thread will keep trying (spinning) until it can acquire the lock
- Built on an integer location in memory that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- Disadvantage:
  - locked-out threads continue to execute in a busy-waiting mode



<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

**Table 6.4 Linux Spinlocks**

# Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4
- Internally:
  - implemented as functions within the kernel and are more efficient than user-visible semaphores
- Three types of kernel semaphores:
  - binary semaphores
  - counting semaphores
  - reader-writer semaphores



<b>Traditional Semaphores</b>	
<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
<b>Reader-Writer Semaphores</b>	
<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

**Table 6.5**

**Linux Semaphores**

## Table 6.6

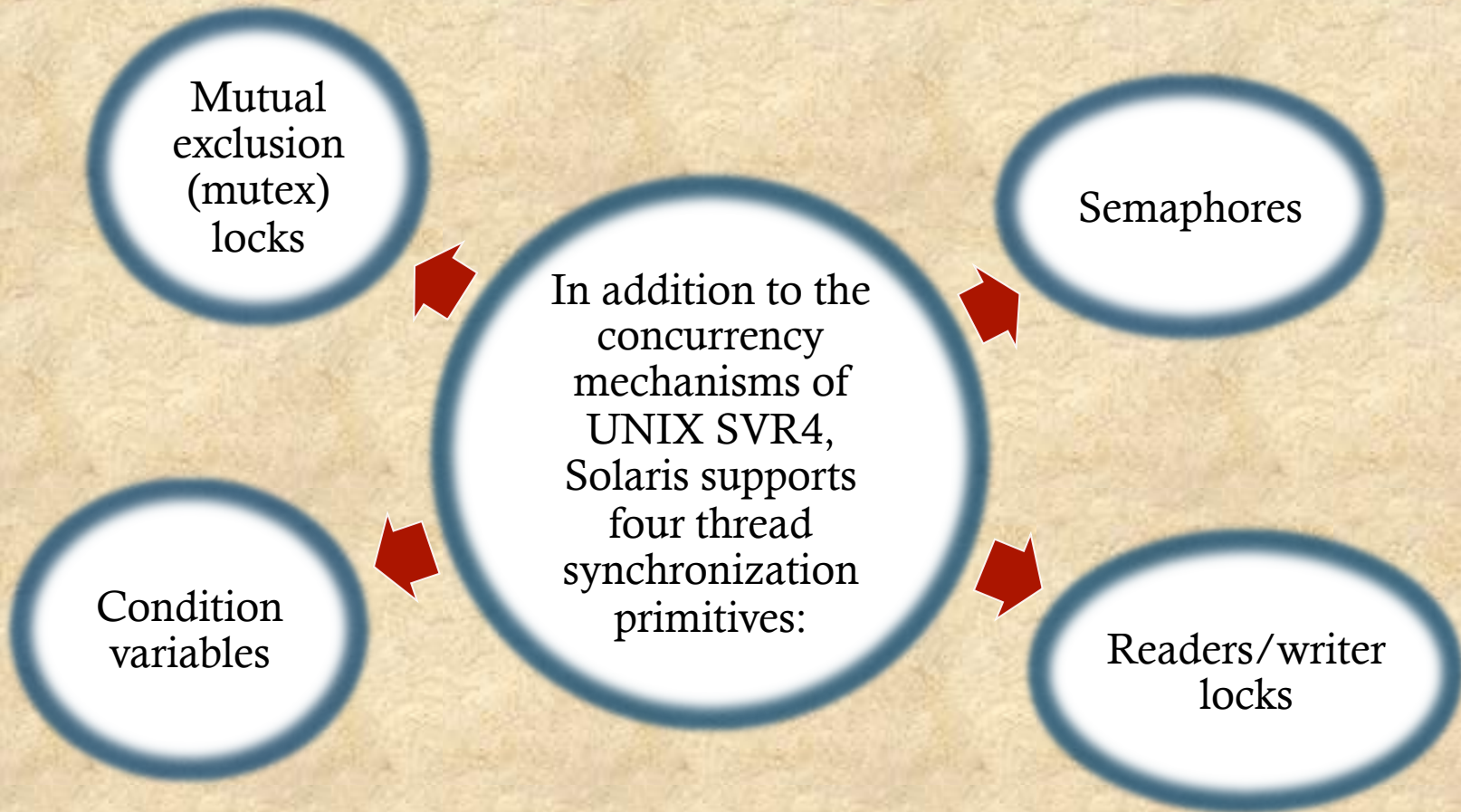
### Linux Memory Barrier Operations

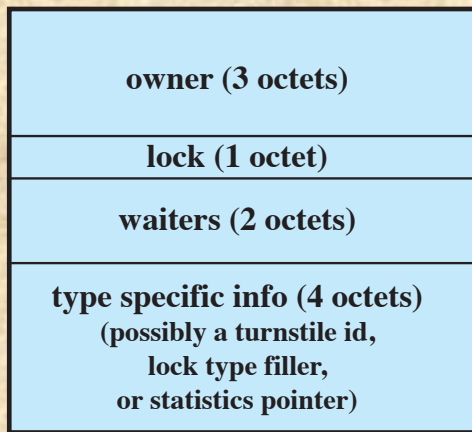
<code>rmb()</code>	Prevents loads from being reordered across the barrier
<code>wmb()</code>	Prevents stores from being reordered across the barrier
<code>mb()</code>	Prevents loads and stores from being reordered across the barrier
<code>Barrier()</code>	Prevents the compiler from reordering loads or 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

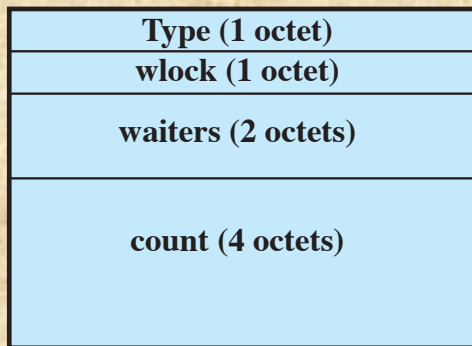
UP = uniprocessor

# Synchronization Primitives

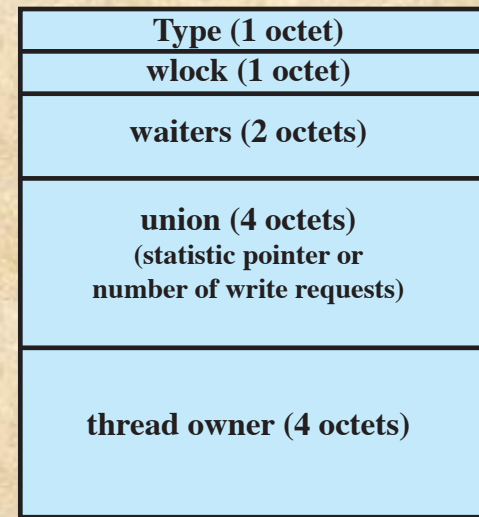




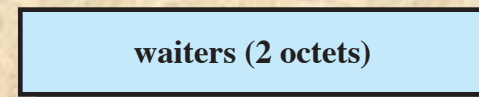
(a) MUTEX lock



(b) Semaphore



(c) Reader/writer lock



(d) Condition variable

**Figure 6.15 Solaris Synchronization Data Structures**

# Mutual Exclusion (MUTEX) Lock

- 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
- A thread attempts to acquire a mutex lock by executing the `mutex_enter` primitive
- Default blocking policy is a spinlock
- An interrupt-based blocking mechanism is optional



# Semaphores

Solaris provides classic counting semaphores with the following primitives:

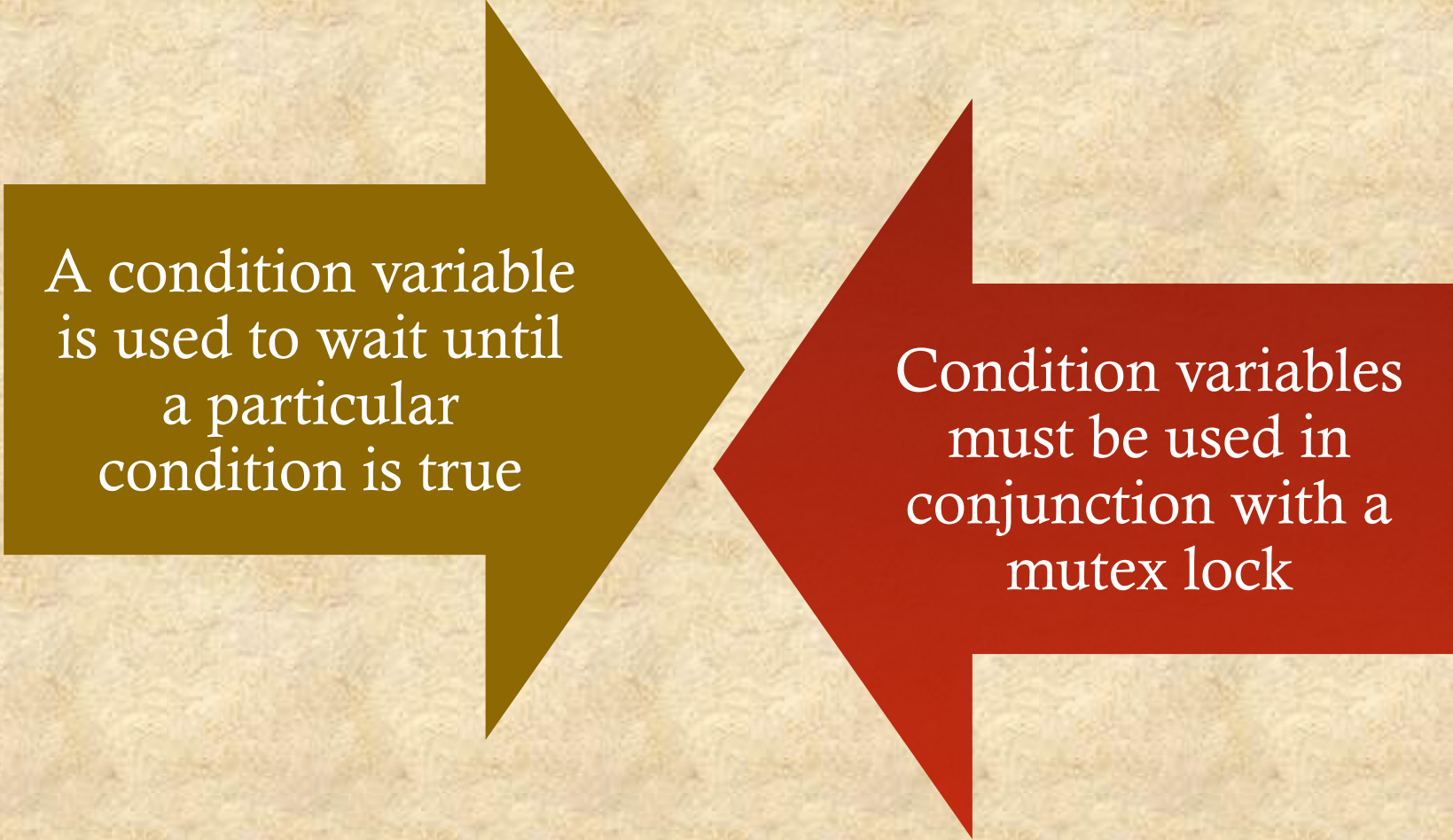
- `sema_p()` Decrements the semaphore, potentially blocking the thread
- `sema_v()` Increments the semaphore, potentially unblocking a waiting thread
- `sema_tryv()` Decrements the semaphore if blocking is not required



# Readers/Writer Locks

- Allows multiple threads to have simultaneous read-only access to an object protected by the lock
- Allows a single thread to access the object for writing at one time, while excluding all readers
  - when lock is acquired for writing it takes on the status of write lock
  - if one or more readers have acquired the lock its status is read lock

# 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 7 Concurrency Mechanisms

- Windows provides synchronization among threads as part of the object architecture

Most important methods are:

- executive dispatcher objects
- user mode critical sections
- slim reader-writer locks
- condition variables
- lock-free operations

# Wait Functions

Allow a thread to block its own execution

Do not return until the specified criteria have been met

The type of wait function determines the set of criteria used

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

**Table 6.7**  
**Windows Synchronization Objects**

*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 reader-writer lock enters the kernel to block only after attempting to use a spin-lock
- It is *slim* in the sense that it normally only requires allocation of a single pointer-sized piece of memory



# Condition Variables

- Windows also has condition variables
- The process must declare and initialize a `CONDITION_VARIABLE`
- Used with either critical sections or SRW locks
- Used as follows:
  1. acquire exclusive lock
  2. `while (predicate()==FALSE)SleepConditionVariable()`
  3. perform the protected operation
  4. release the lock



# Lock-free Synchronization

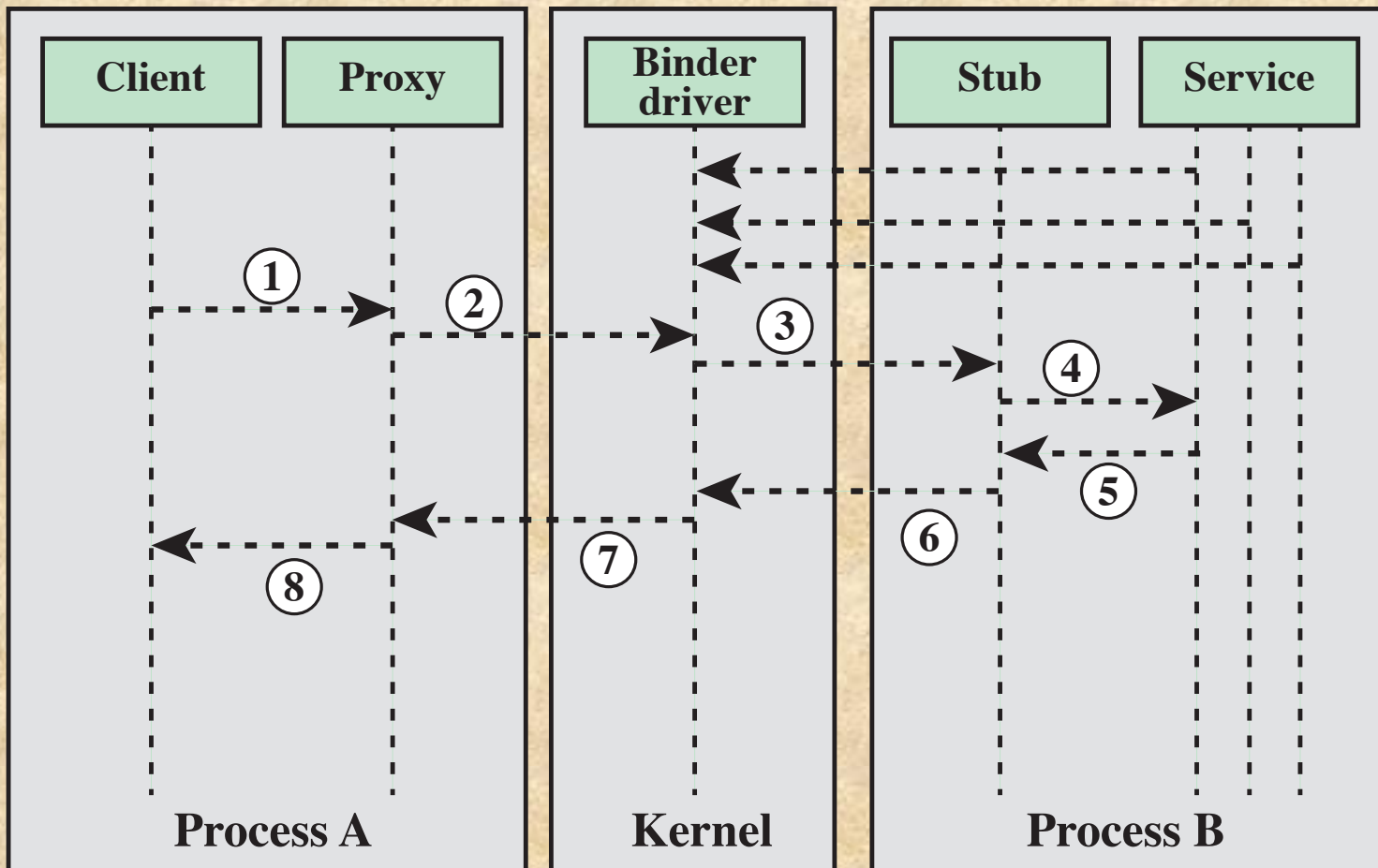
- Windows also relies heavily on interlocked operations for synchronization
  - interlocked operations use hardware facilities to guarantee that memory locations can be read, modified, and written in a single atomic operation

## “Lock-free”

- synchronizing without taking a software lock
- a thread can never be switched away from a processor while still holding a lock

# Android Interprocess Communication

- Android adds to the kernel a new capability known as Binder
  - Binder provides a lightweight remote procedure call (RPC) capability that is efficient in terms of both memory and processing requirements
  - also used to mediate all interaction between two processes
- The RPC mechanism works between two processes on the same system but running on different virtual machines
- The method used for communicating with the Binder is the `ioctl` system call
  - the `ioctl` call is a general-purpose system call for device-specific I/O operations



**Figure 6.16 Binder Operation**

# Summary

- Principles of deadlock
  - Reusable/consumable resources
  - Resource allocation graphs
  - Conditions for deadlock
- Deadlock prevention
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Deadlock avoidance
  - Process initiation denial
  - Resource allocation denial
- Deadlock detection
  - Deadlock detection algorithm
  - Recovery
- Android interprocess communication
- UNIX concurrency mechanisms
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals
- Linux kernel concurrency mechanisms
  - Atomic operations
  - Spinlocks
  - Semaphores
  - Barriers
- Solaris thread synchronization primitives
  - Mutual exclusion lock
  - Semaphores
  - Readers/writer lock
  - Condition variables
- Windows 7 concurrency mechanisms