

# CS 332/532 Systems Programming

Lecture 23  
Processes in OS

Professor : Mahmut Unan – UAB CS

# Agenda

- Concurrency
- Deadlock
- Signals
- Pipes

# Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

# Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - The “loser” of the race is the process that updates last and will determine the final value of the variable



**KNOCK KNOCK  
RACE CONDITION  
WHO'S THERE?**

# Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - For example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- The need for mutual exclusion
- Deadlock
- Starvation

<b>Semaphore</b>	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a <b>counting semaphore</b> or a <b>general semaphore</b>
<b>Binary Semaphore</b>	A semaphore that takes on only the values 0 and 1.
<b>Mutex</b>	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
<b>Condition Variable</b>	A data type that is used to block a process or thread until a particular condition is true.
<b>Monitor</b>	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are <i>critical sections</i> . A monitor may have a queue of processes that are waiting to access it.
<b>Event Flags</b>	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
<b>Mailboxes/Messages</b>	A means for two processes to exchange information and that may be used for synchronization.
<b>Spinlocks</b>	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

## Common

## Concurrency

## Mechanisms

# Semaphore

- The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.
- Any complex coordination requirement can be satisfied by the appropriate structure of signals.
- For signaling, special variables called semaphores are used.
- To transmit a signal via semaphore  $s$  , a process executes the primitive `semSignal(s)` .
- To receive a signal via semaphore  $s$  , a process executes the primitive `semWait(s)` ; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

# Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

- 1) A semaphore may be initialized to a nonnegative integer value
- 2) The semWait operation decrements the semaphore value
- 3) The semSignal operation increments the semaphore value

# Mutual Exclusion Lock (mutex)

- A concept related to the binary semaphore is the mutual exclusion lock (mutex)
- A mutex is a programming flag used to grab and release an object. When data are acquired that cannot be shared or processing is started that cannot be performed simultaneously elsewhere in the system, the mutex is set to lock (typically zero), which blocks other attempts to use it.
- The mutex is set to unlock when the data are no longer needed or the routine is finished.
- A key difference between the a mutex and a binary semaphore is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

# Producer/Consumer Problem

General Statement:

One or more producers are generating data and placing these in a buffer

A single consumer is taking items out of the buffer one at a time

Only one producer or consumer may access the buffer at any one time

The Problem:

Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer

# Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

# Synchronization

- A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are a special data type in monitors which are operated on by two functions:
    - `cwait (c)` : suspend execution of the calling process on condition `c`
    - `csignal (c)` : resume execution of some process blocked after a `cwait` on the same condition

# Message Passing

- When processes interact with one another two fundamental requirements must be satisfied:

## Synchronization

- To enforce mutual exclusion

## Communication

- To exchange information

- Message passing is one approach to providing both of these functions
  - Works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

# Message Passing

- The actual function is normally provided in the form of a pair of primitives:

send (destination, message)  
receive (source, message)

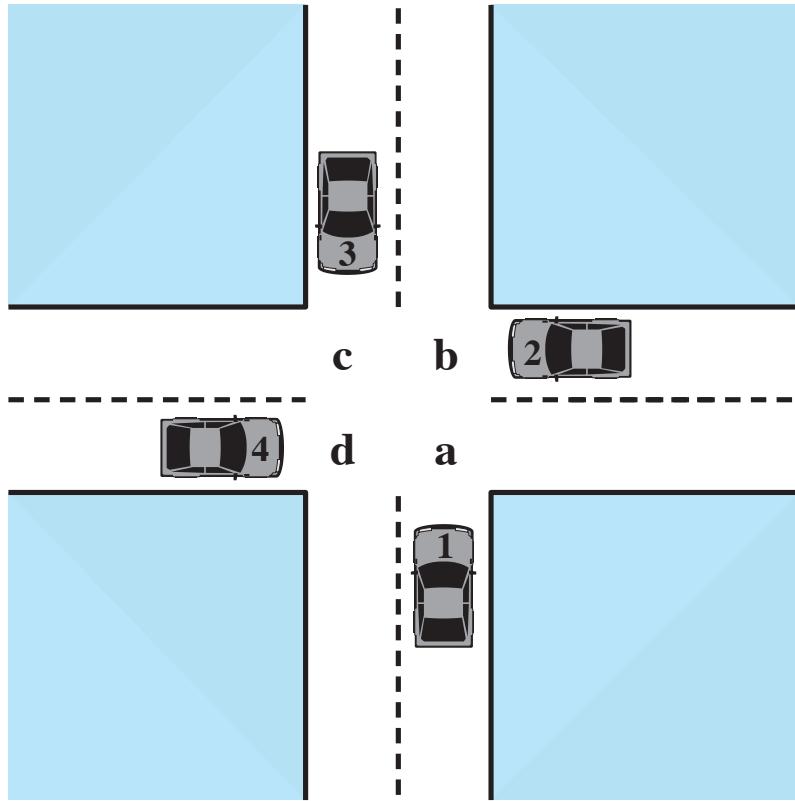
- » A process sends information in the form of a *message* to another process designated by a *destination*
- » A process receives information by executing the receive primitive, indicating the *source* and the *message*

<b>Synchronization</b>	<b>Format</b>
Send	Content
blocking	Length
nonblocking	fixed
Receive	variable
blocking	
nonblocking	
test for arrival	
<b>Addressing</b>	<b>Queueing Discipline</b>
Direct	FIFO
send	
receive	
explicit	
implicit	
Indirect	Priority
static	
dynamic	
ownership	

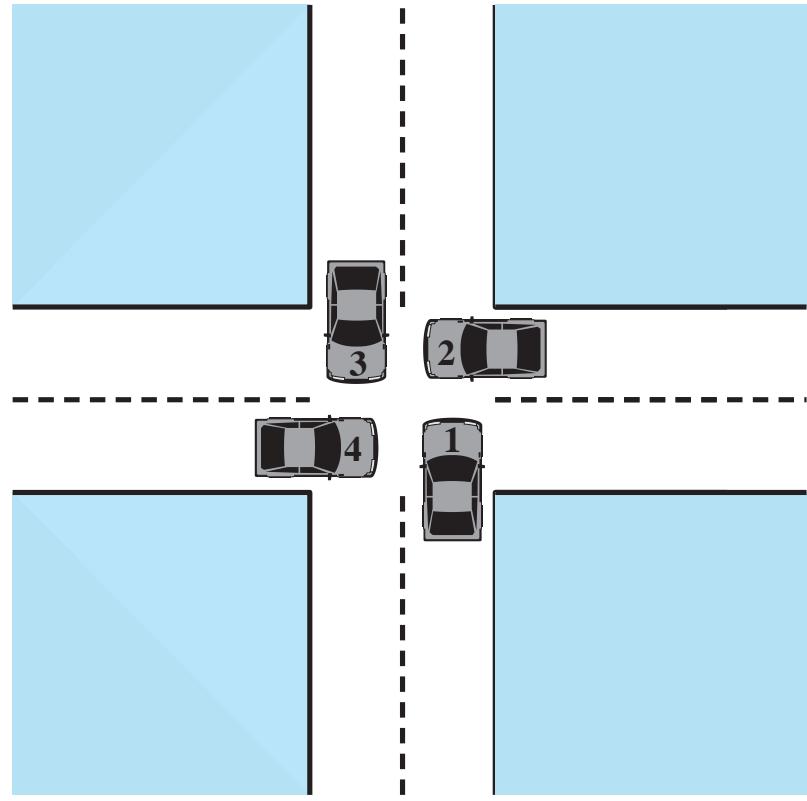
**Table 5.5**  
**Design Characteristics of Message Systems for**  
**Interprocess Communication and Synchronization**

# 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 because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible



(b) Deadlock

**Figure 6.1 Illustration of Deadlock**

**INTERVIEWER: EXPLAIN DEADLOCK AND I'LL HIRE YOU**



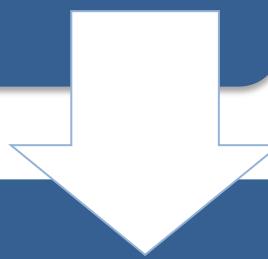
*©thecrazyprogrammer*

**PROGRAMMER: HIRE ME AND I'LL EXPLAIN IT TO YOU**

# 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

# 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;**

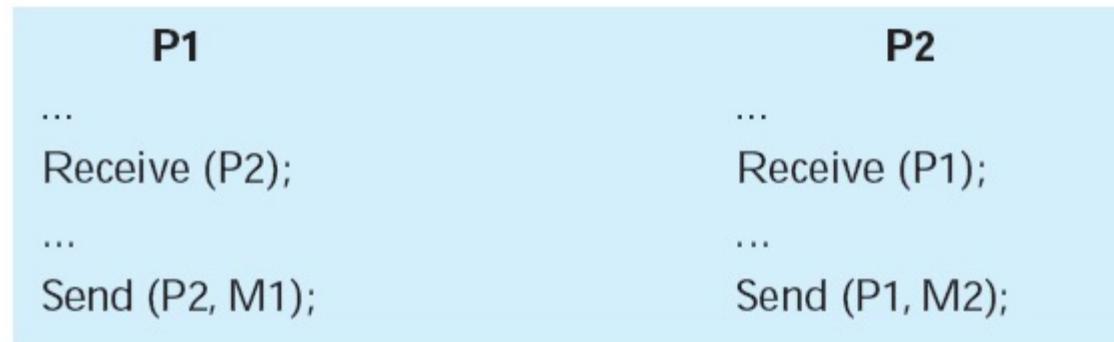
...

**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:



# Deadlock Approaches

- There is no single effective strategy that can deal with all types of deadlock
- Three approaches are common:
  - **Deadlock avoidance**
    - Do not grant a resource request if this allocation might lead to deadlock
  - **Deadlock prevention**
    - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening
  - **Deadlock detection**
    - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

# Conditions for Deadlock

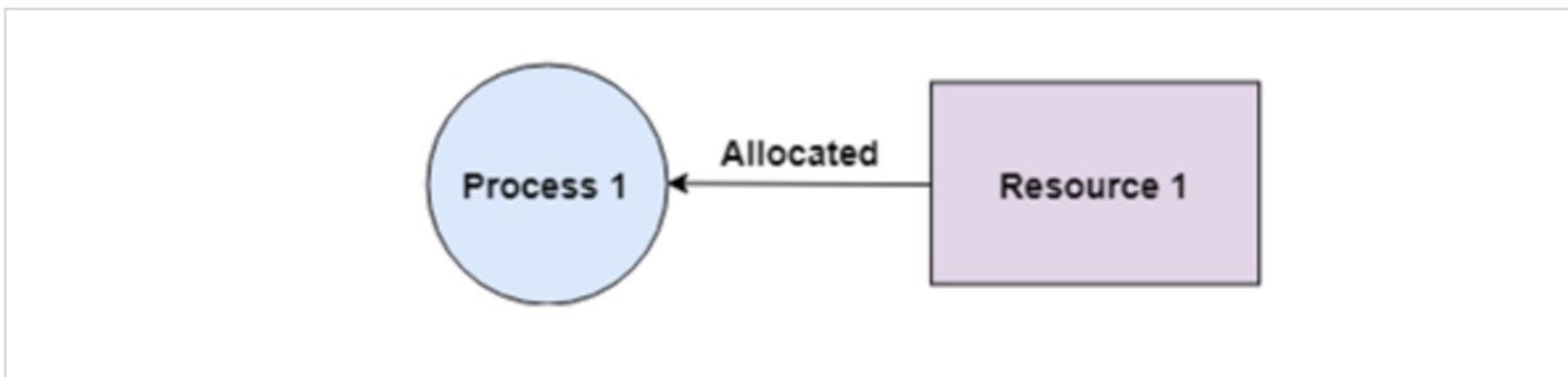
Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
<ul style="list-style-type: none"><li>• Only one process may use a resource at a time</li><li>• No process may access a resource until that has been allocated to another process</li></ul>	<ul style="list-style-type: none"><li>• A process may hold allocated resources while awaiting assignment of other resources</li></ul>	<ul style="list-style-type: none"><li>• No resource can be forcibly removed from a process holding it</li></ul>	<ul style="list-style-type: none"><li>• A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain</li></ul>

# 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

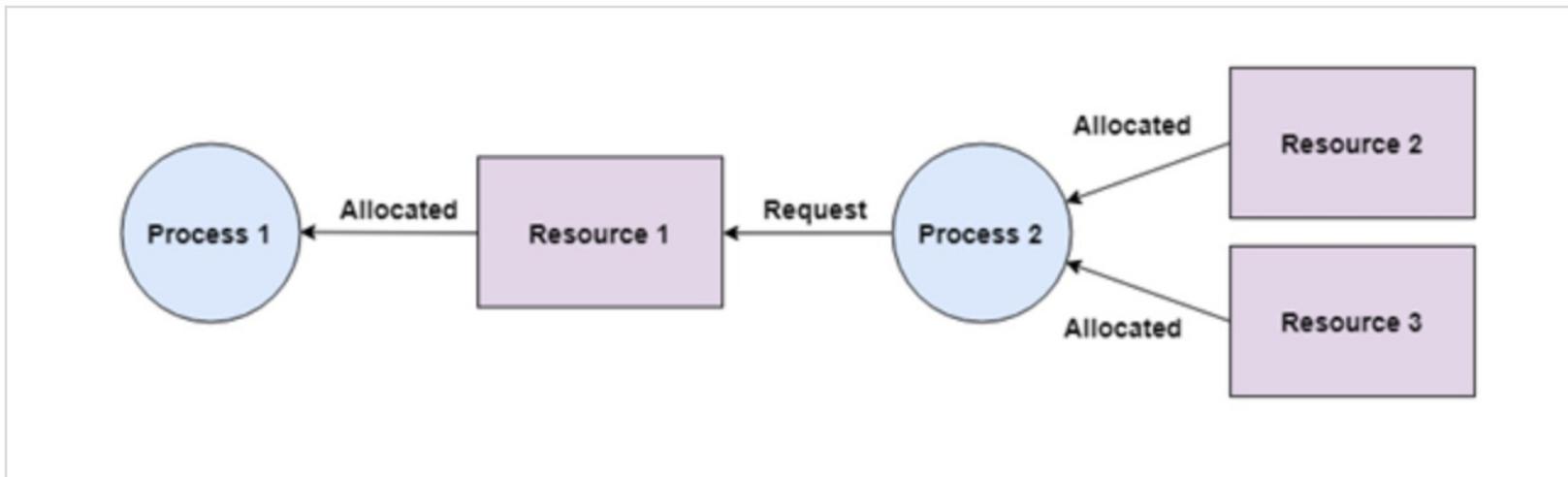
# Deadlock Condition Prevention

- Mutual exclusion
  - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
  - Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
  - Even in this case, deadlock can occur if more than one process requires write permission



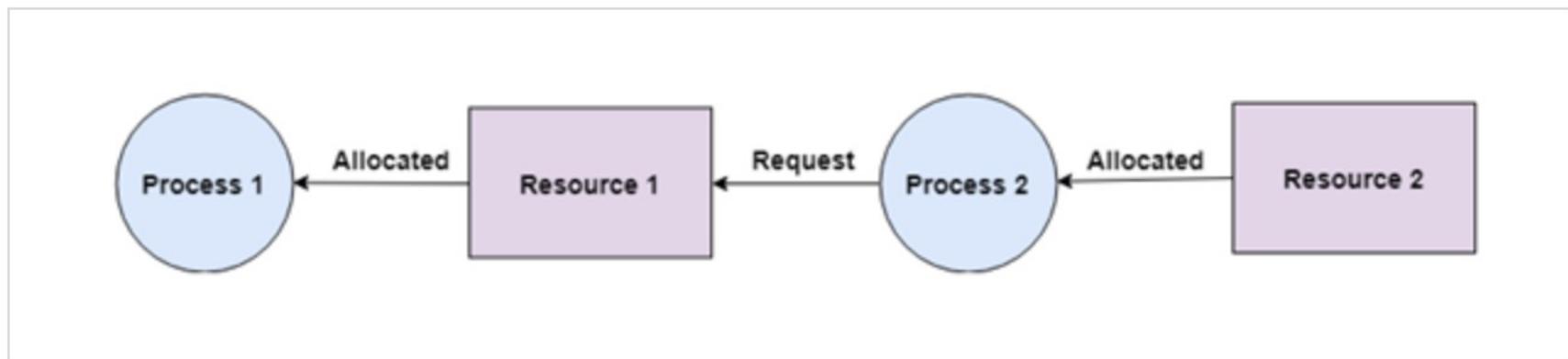
# Deadlock Condition Prevention

- Hold and wait
  - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously



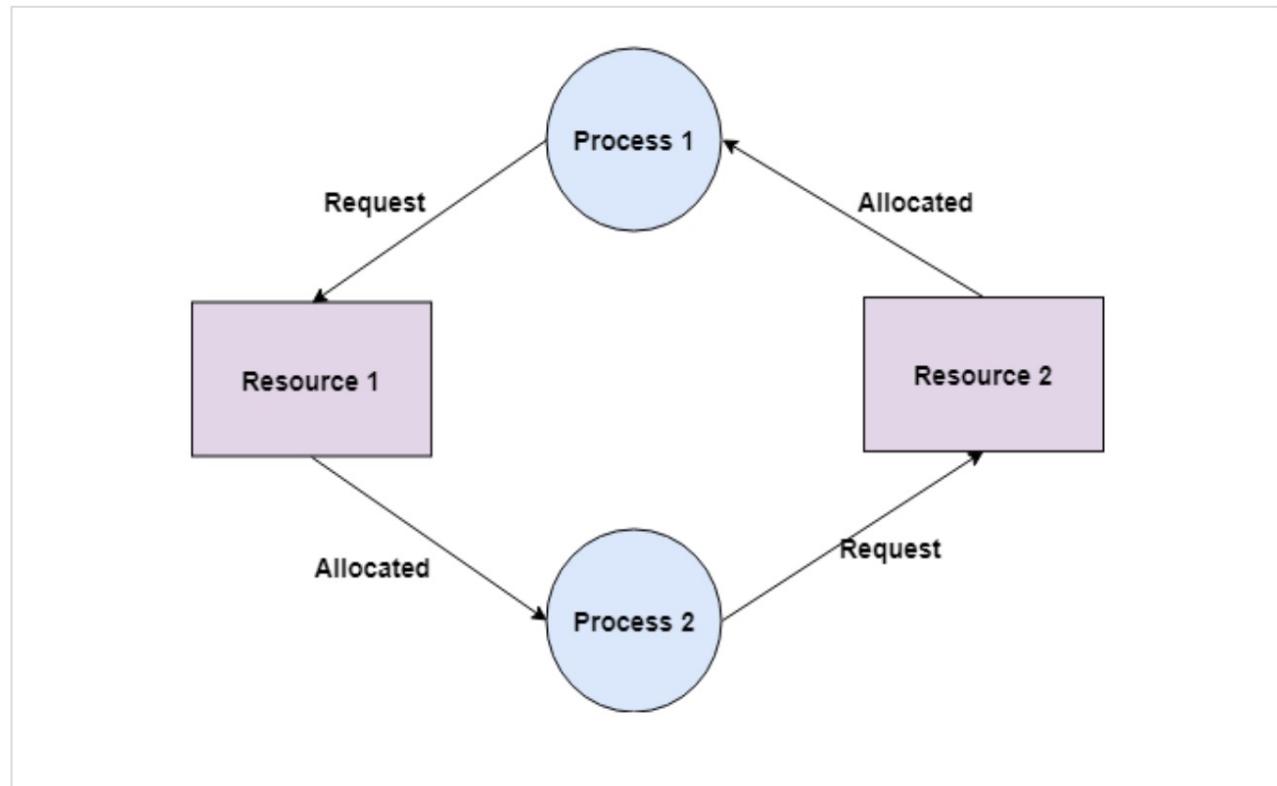
# Deadlock Condition Prevention

- 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



# Deadlock Condition Prevention

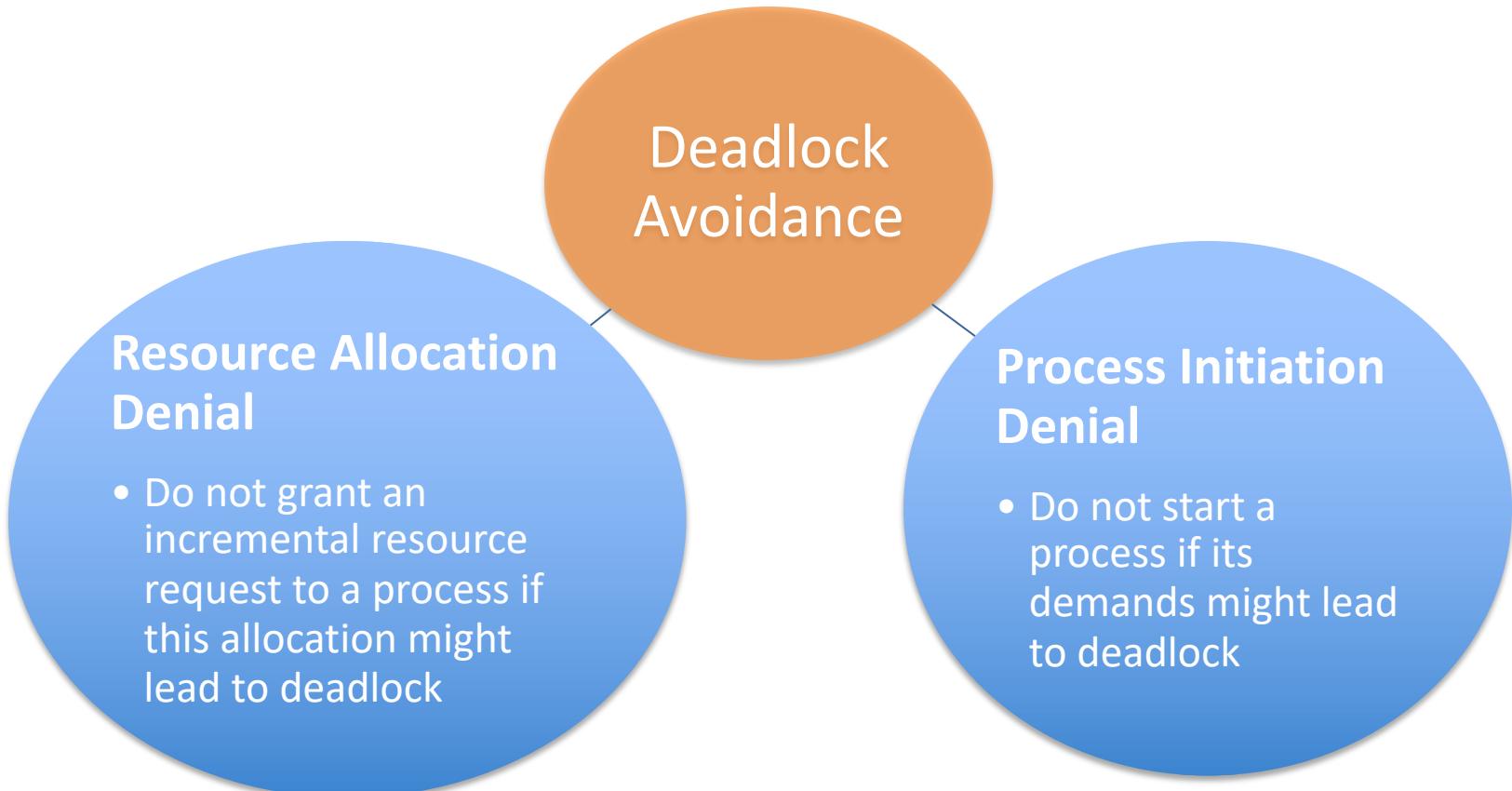
- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types



# Deadlock Avoidance

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- Requires knowledge of future process requests

# Two Approaches to Deadlock Avoidance



# 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

# Deadlock Avoidance Advantages

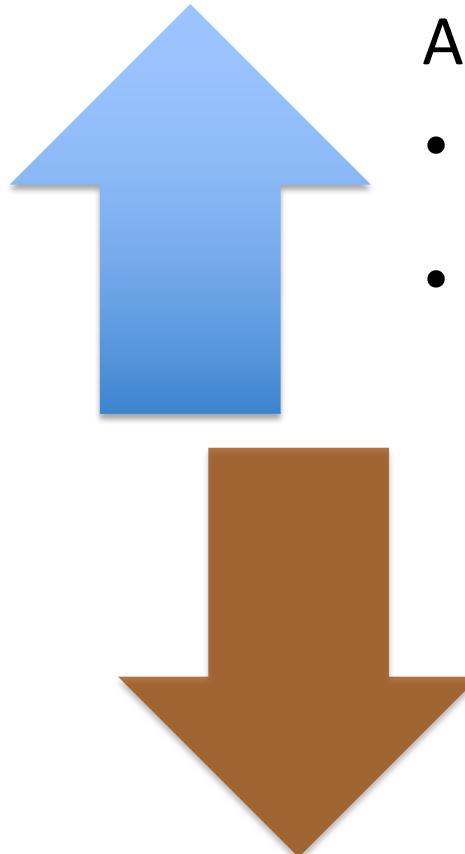
- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

# 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 Detection Algorithm

- 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

# 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

<b>Value</b>	<b>Name</b>	<b>Description</b>
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 288 in textbook)

# Real-time (RT) Signals

- Linux includes all of the concurrency mechanisms found in other UNIX systems
- Linux also supports real-time (RT) signals
- RT signals differ from standard UNIX signals in three primary ways:
  - Signal delivery in priority order is supported
  - Multiple signals can be queued
  - With standard signals, no value or message can be sent to the target process – it is only a notification
  - With RT signals it is possible to send a value along with the signal

# 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

## Table 6.2

# Linux Atomic Operations

<b>Atomic Integer Operations</b>	
ATOMIC_INIT (int i)	At declaration: initialize an atomic_t to i
int atomic_read atomic_t *v)	Read integer value of v
void atomic_set(atomic_t *v, int i)	Set the value of v to integer i
void atomic_add(int i, atomic_t *v)	Add i to v
void atomic_sub(int i, atomic_t *v)	Subtract i from v
void atomic_inc(atomic_t *v)	Add 1 to v
void atomic_dec(atomic_t *v)	Subtract 1 from v
int atomic_sub_and_test(int i, atomic_t *v)	Subtract i from v; return 1 if the result is zero; return 0 otherwise
int atomic_add_negative(int i, atomic_t *v)	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
int atomic_dec_and_test(atomic_t *v)	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
int atomic_inc_and_test(atomic_t *v)	Add 1 to v; return 1 if the result is zero; return 0 otherwise
<b>Atomic Bitmap Operations</b>	
void set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr
void clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr
void change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr
int test_and_set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr; return the old bit value
int test_bit(int nr, void *addr)	Return the value of bit nr in the bitmap pointed to by addr

(Table can be found on page 289 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**

(Table can be found on page 291 in textbook)

# 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

**Table 6.5**

**Linux**

**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 -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
<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 can be found on page 293 in textbook)

## Table 6.6

### Linux Memory Barrier Operations

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

SMP = symmetric multiprocessor

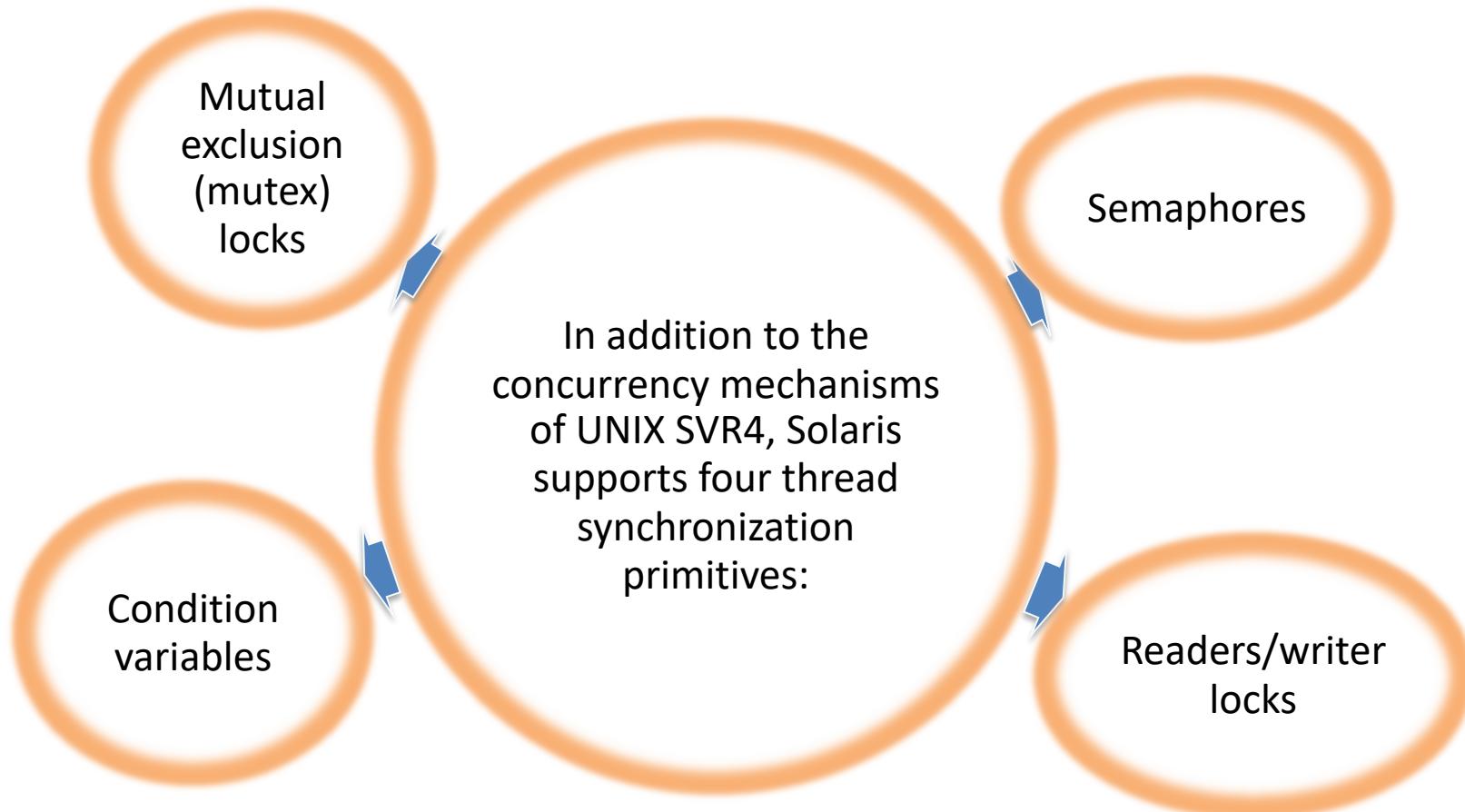
UP = uniprocessor

(Table can be found on page 294 in textbook)

# Read-Copy-Update (RCU)

- The RCU mechanism is an advanced lightweight synchronization mechanism which was integrated into the Linux kernel in 2002
- The RCU is used widely in the Linux kernel
- RCU is also used by other operating systems
- There is a userspace RCU library called librcu
- The shared resources that the RCU mechanism protects must be accessed via a pointer
- The RCU mechanism provides access for multiple readers and writers to a shared resource

# Synchronization Primitives



# 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

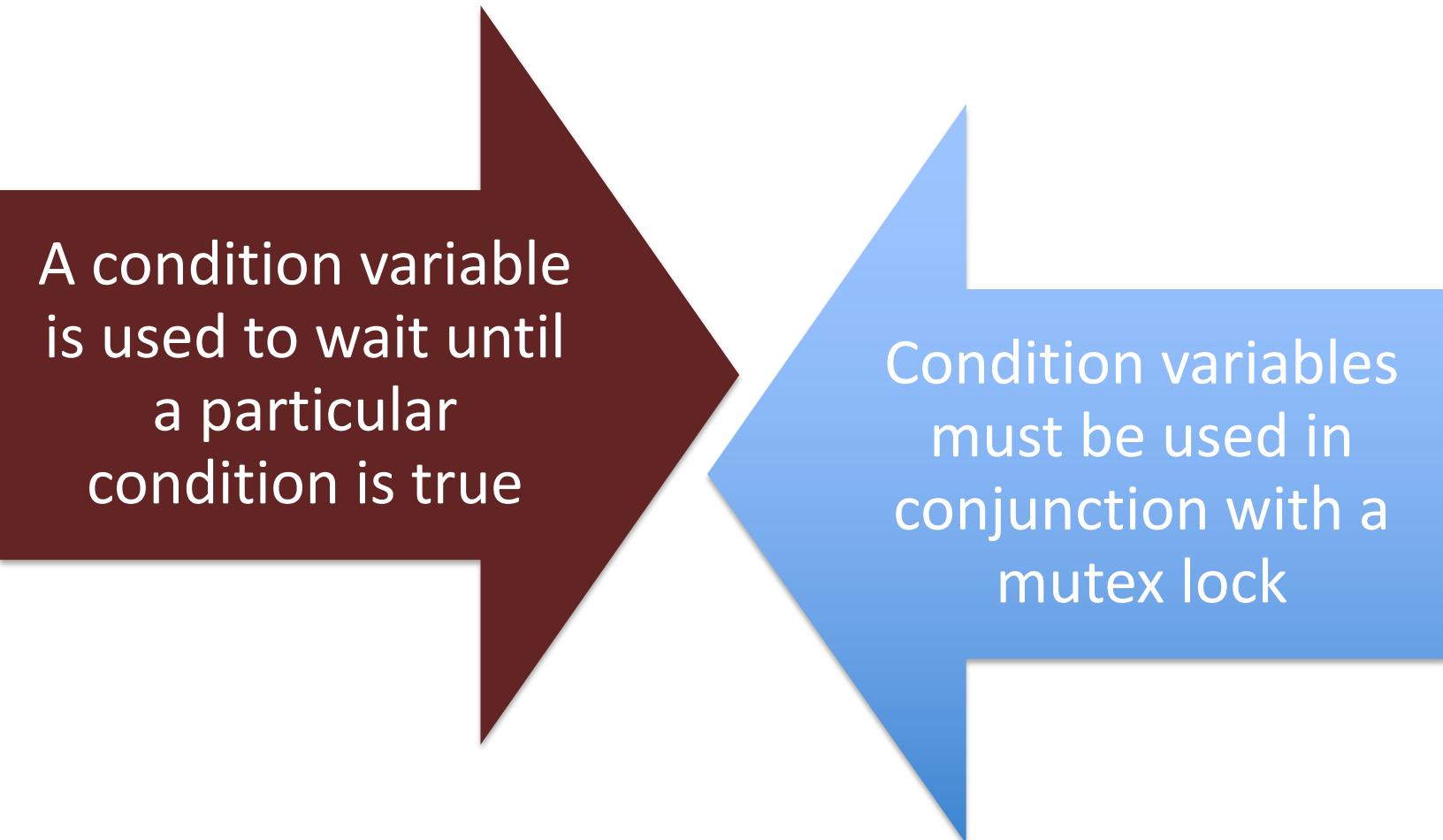
Solaris provides classic counting semaphores with the following primitives:

- `sema_p()` Decrement the semaphore, potentially blocking the thread
- `sema_v()` Increments the semaphore, potentially unblocking a waiting thread
- `sema_try()` Decrement 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