

# UNIX Concurrency Mechanisms

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

Pipes

Messages

Shared  
memory

Semaphores

Signals

# Pipes

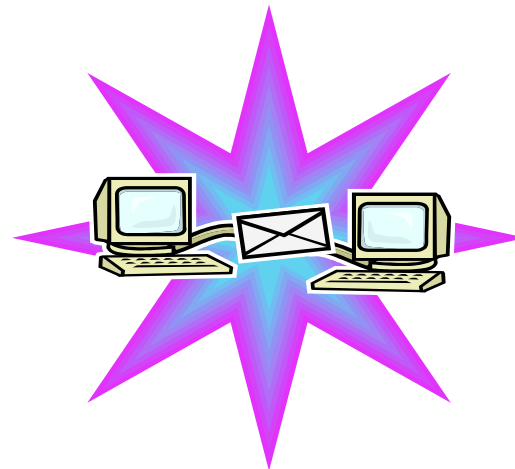
- Circular buffers allowing two processes to communicate on the producer-consumer model
  - first-in-first-out queue, written by one process and read by another

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

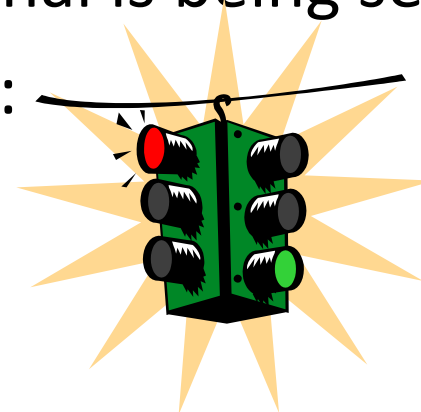
- Generalization of the `semWait` and `semSignal` primitives
  - no other process may access the semaphore until all operations have completed

## 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

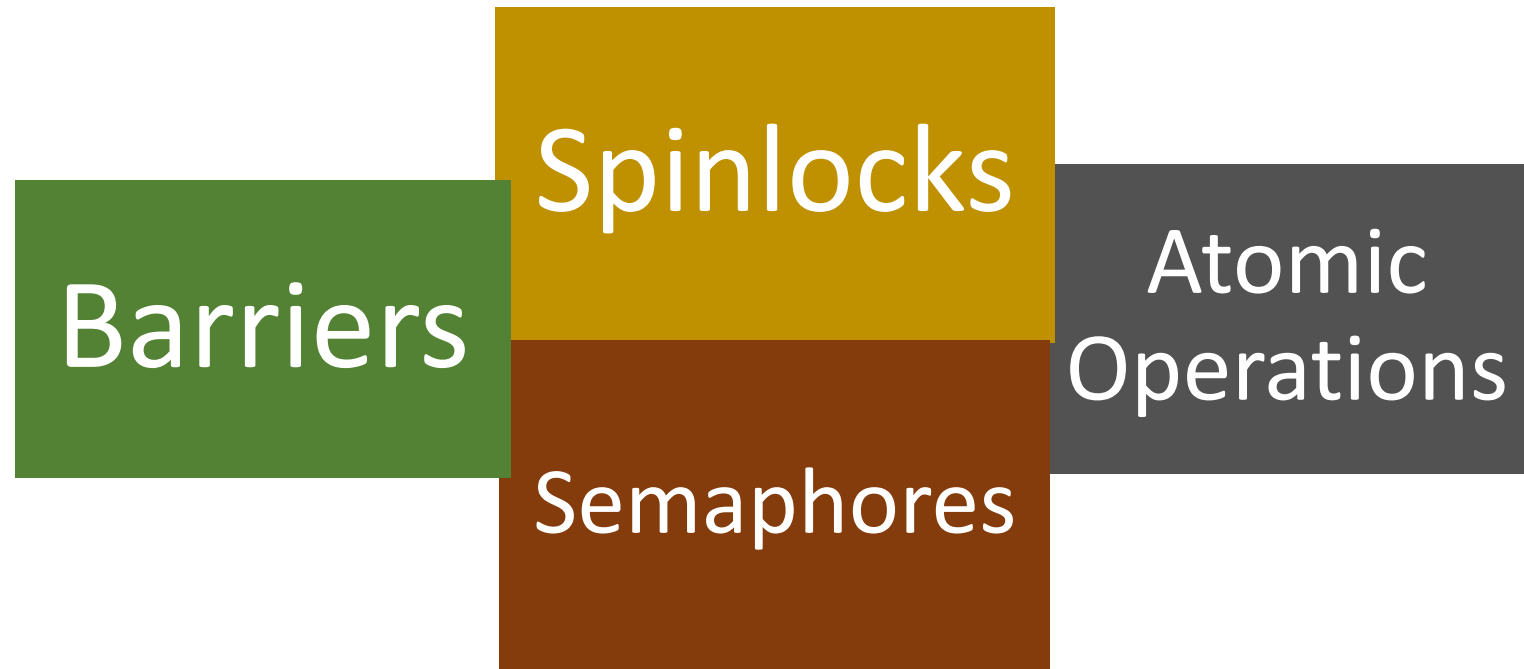
**Table 6.2**

**UNIX Signals**

(Table can be found on page 286 in textbook)

# Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus:





# 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

Atomic Integer Operations	
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
Atomic Bitmap Operations	
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 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



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

**Table 6.5**

**Linux**

**Semaphores**

**Table 6.6**

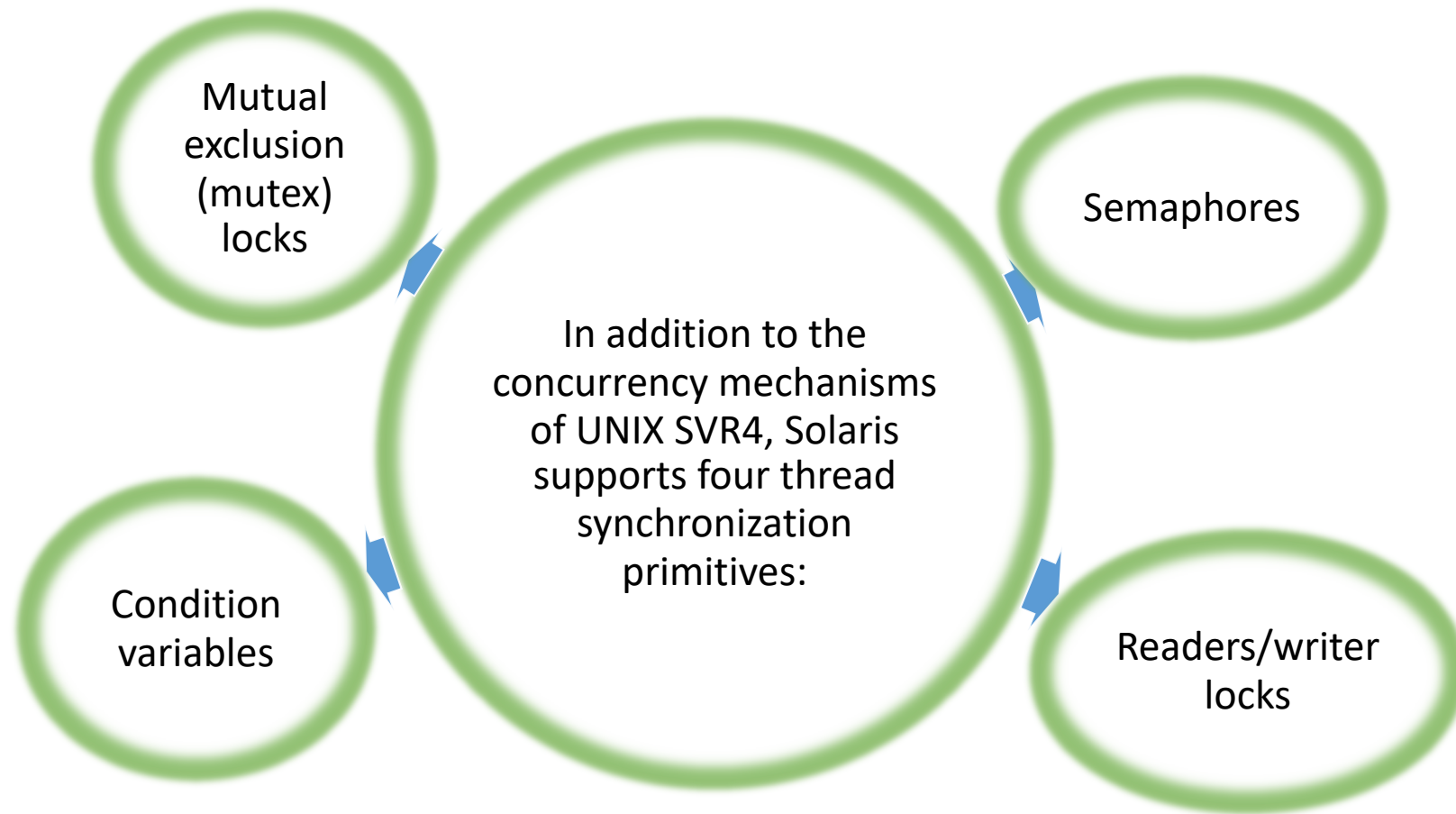
**Linux Memory Barrier Operations**

<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>

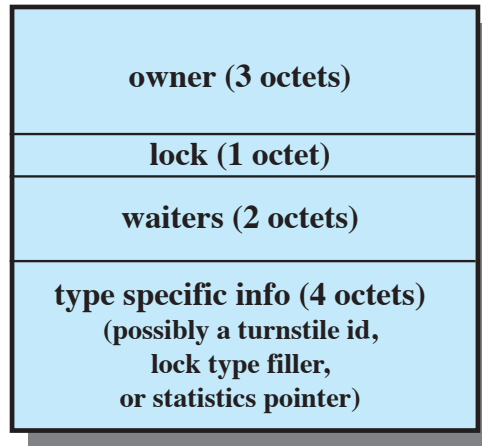
SMP = symmetric multiprocessor

UP = uniprocessor

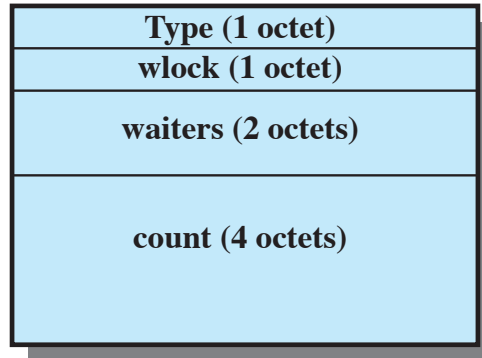
# Synchronization Primitives



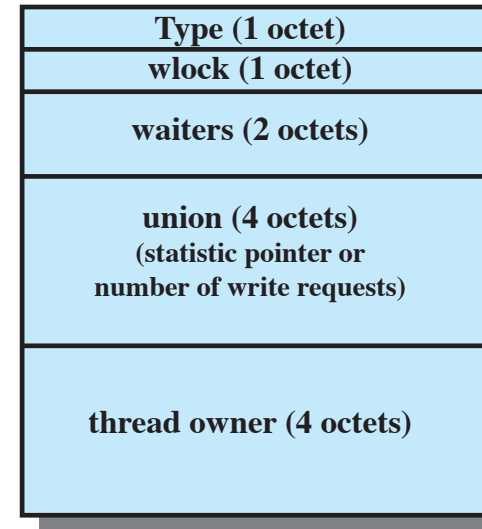




(a) MUTEX lock



(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

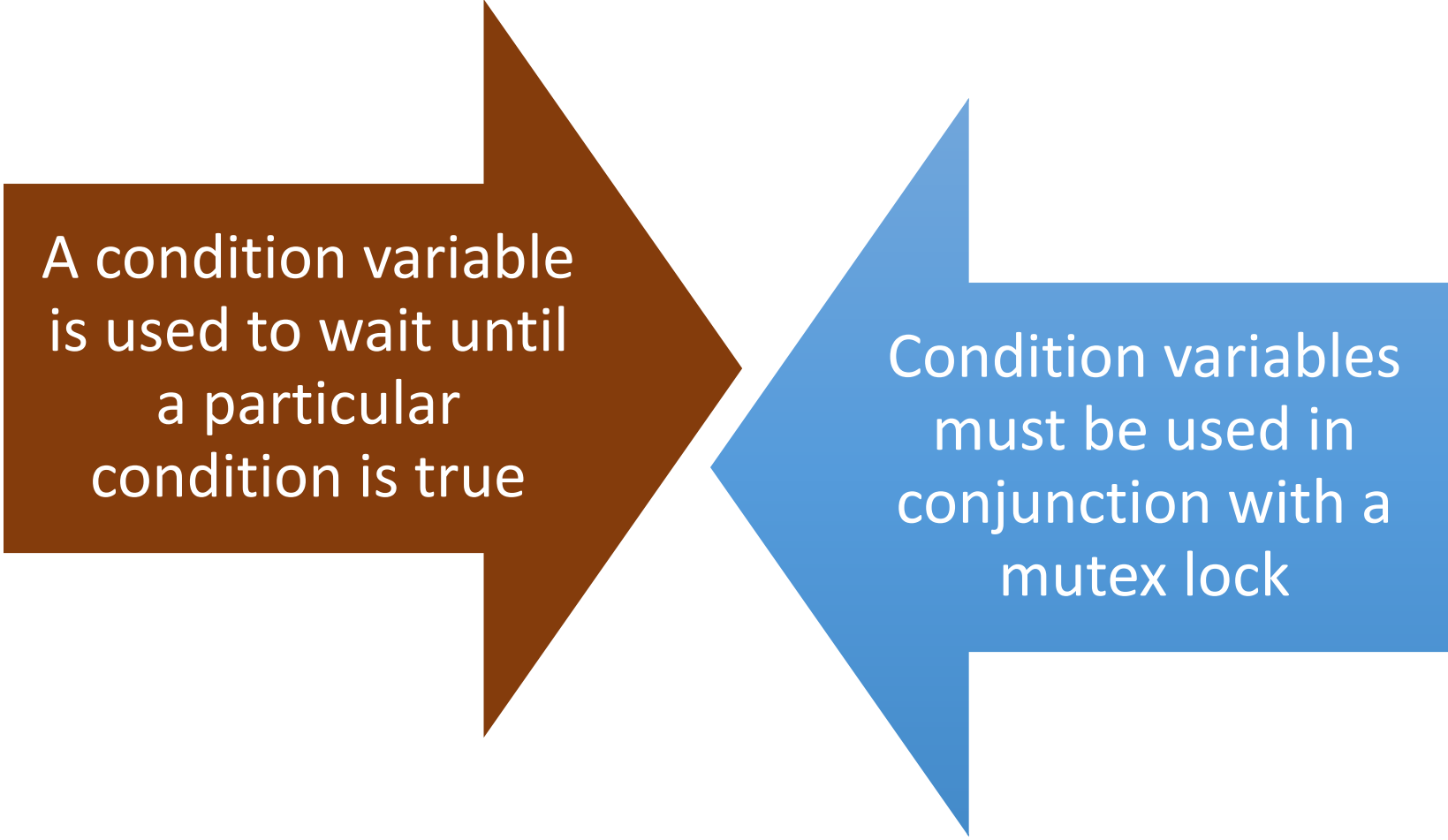
Solaris provides classic counting semaphores with the following primitives:

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

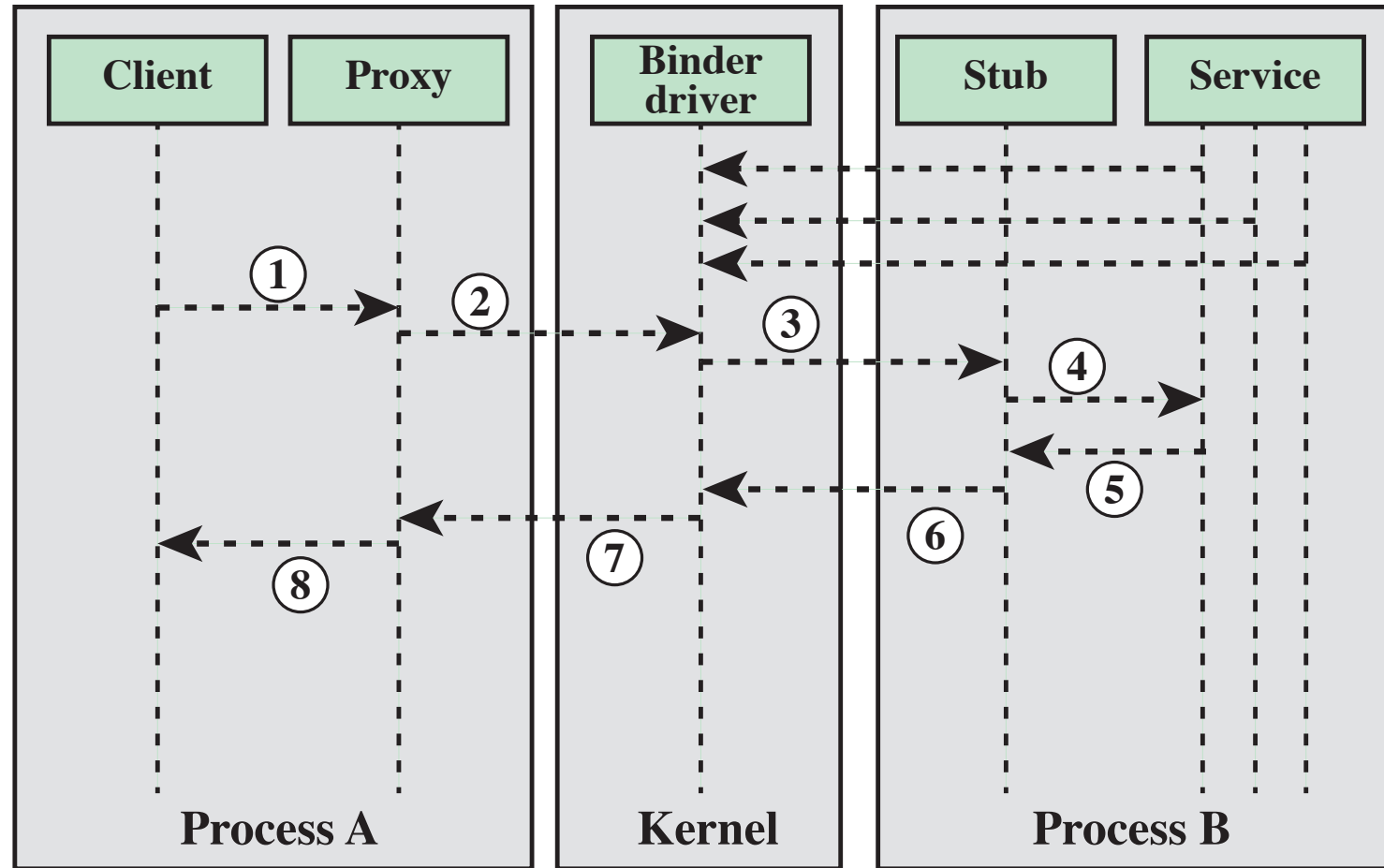
- Windows also relies heavily on interlocked operations for synchronization
  - interlocked operations use hardware facilities to guarantee that memory locations can be read, modified, and written in a single atomic operation

## “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**