# CS 332/532 Systems Programming

Lecture 22

Processes in OS

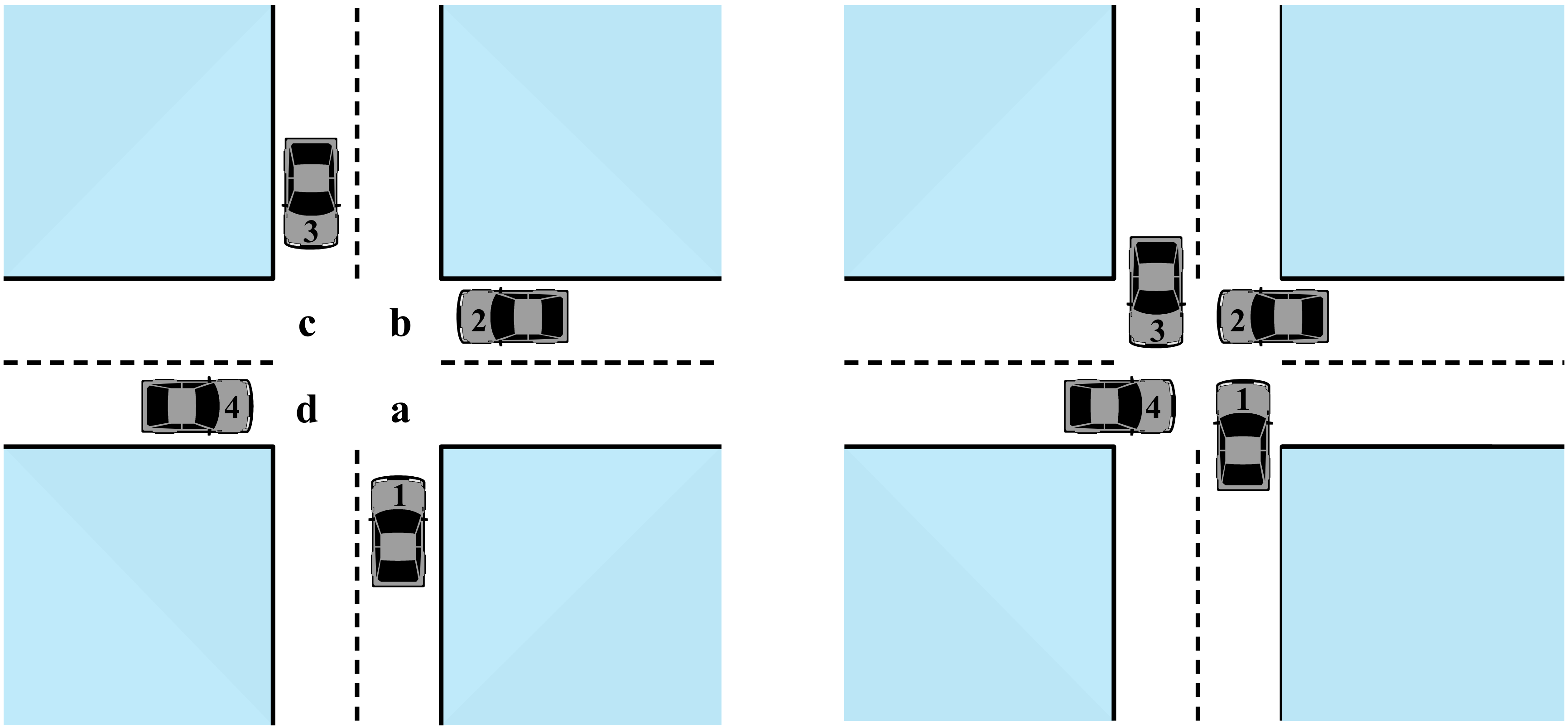
Professor : Mahmut Unan – UAB CS

# Agenda

- Deadlock
- Signals
- Pipes

# 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:

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

# 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

- Only one process may use a resource at a time
- No process may access a resource until that has been allocated to another process

## Hold-and-Wait

- A process may hold allocated resources while awaiting assignment of other resources

## No Pre-emption

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

## Circular Wait

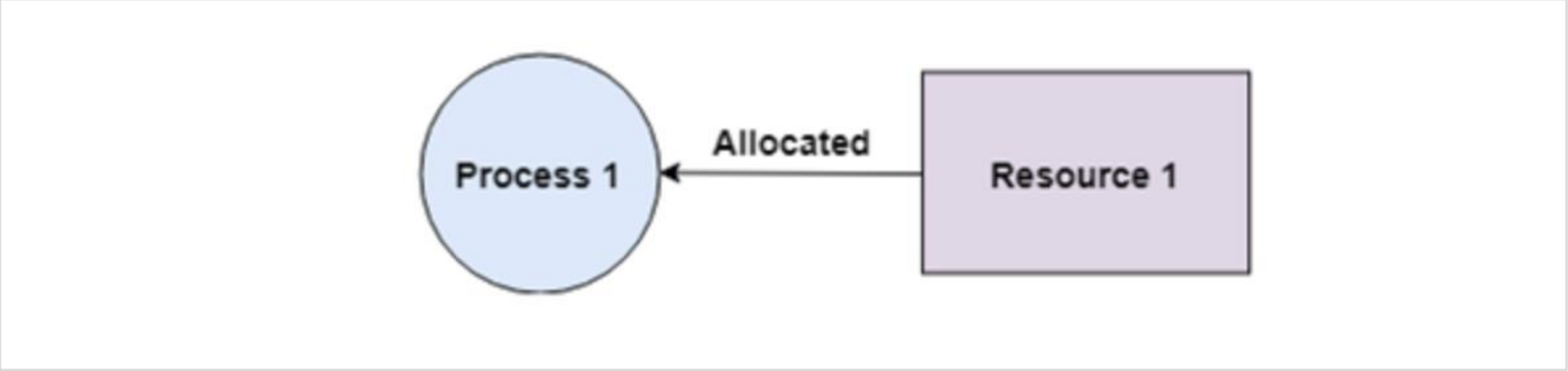
- A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

# 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

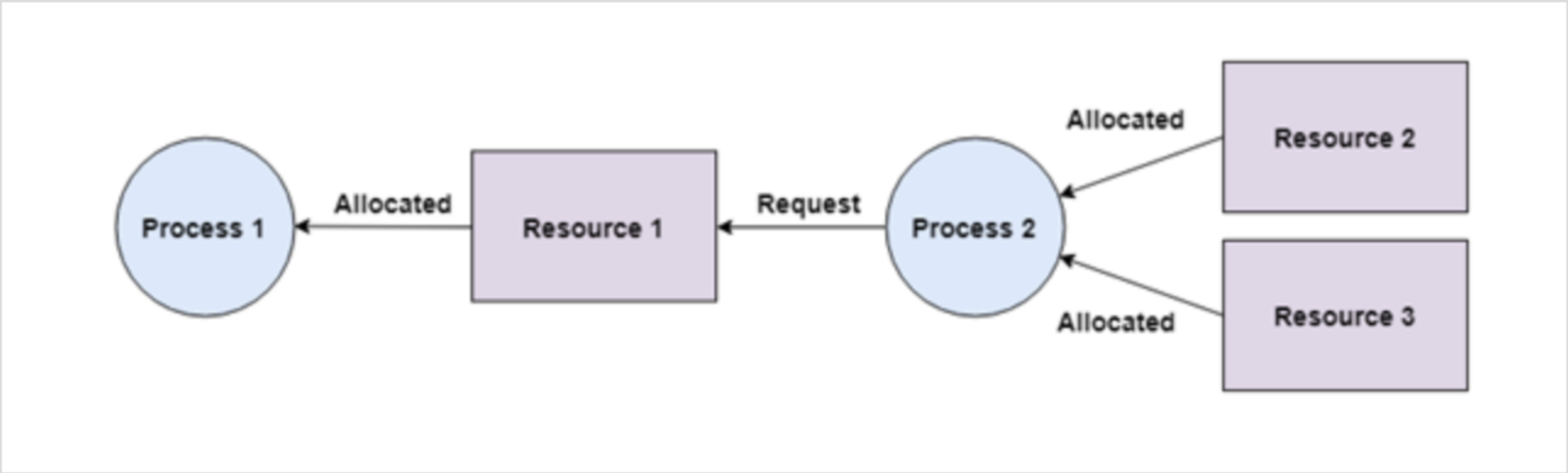




https://www.tutorialspoint.com/process-deadlocks-in-operating-system

# 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

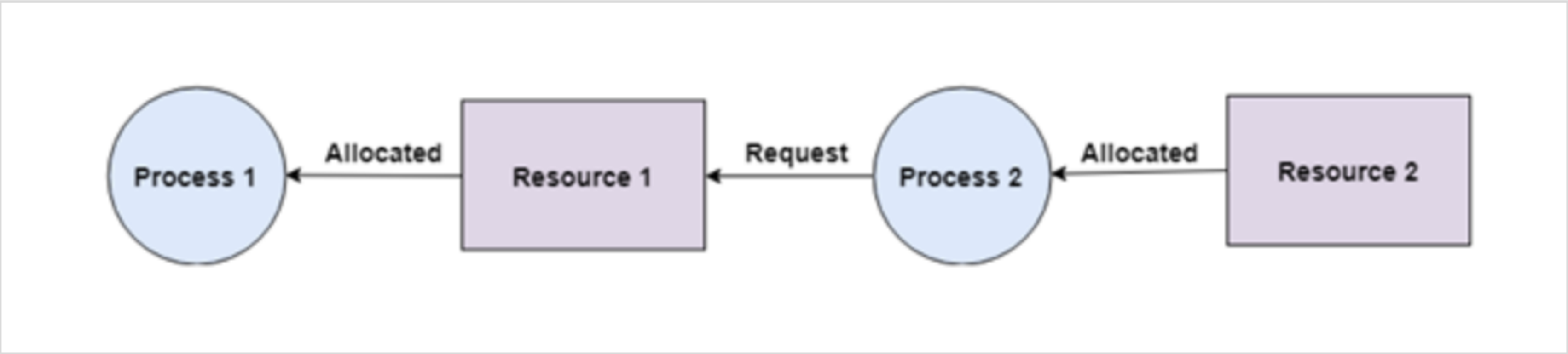




https://www.tutorialspoint.com/process-deadlocks-in-operating-system

# 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

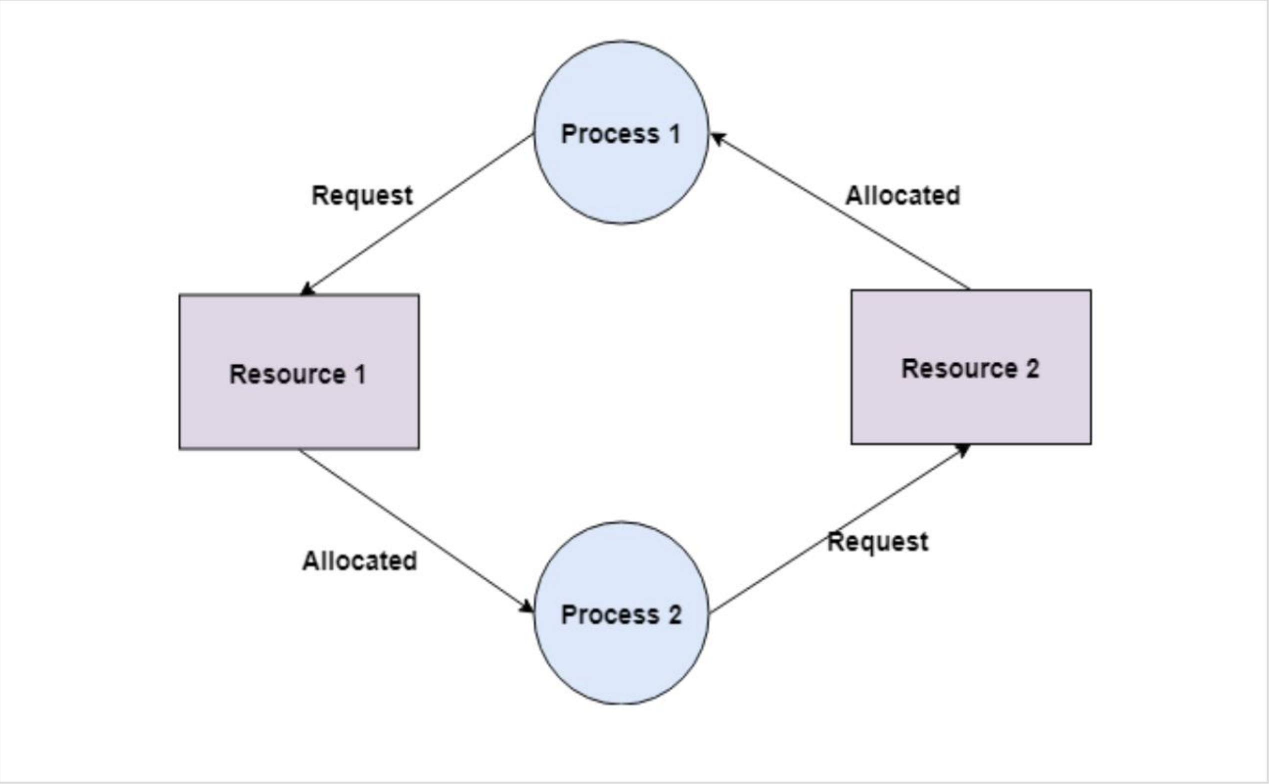




https://www.tutorialspoint.com/process-deadlocks-in-operating-system

# Deadlock Condition Prevention

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



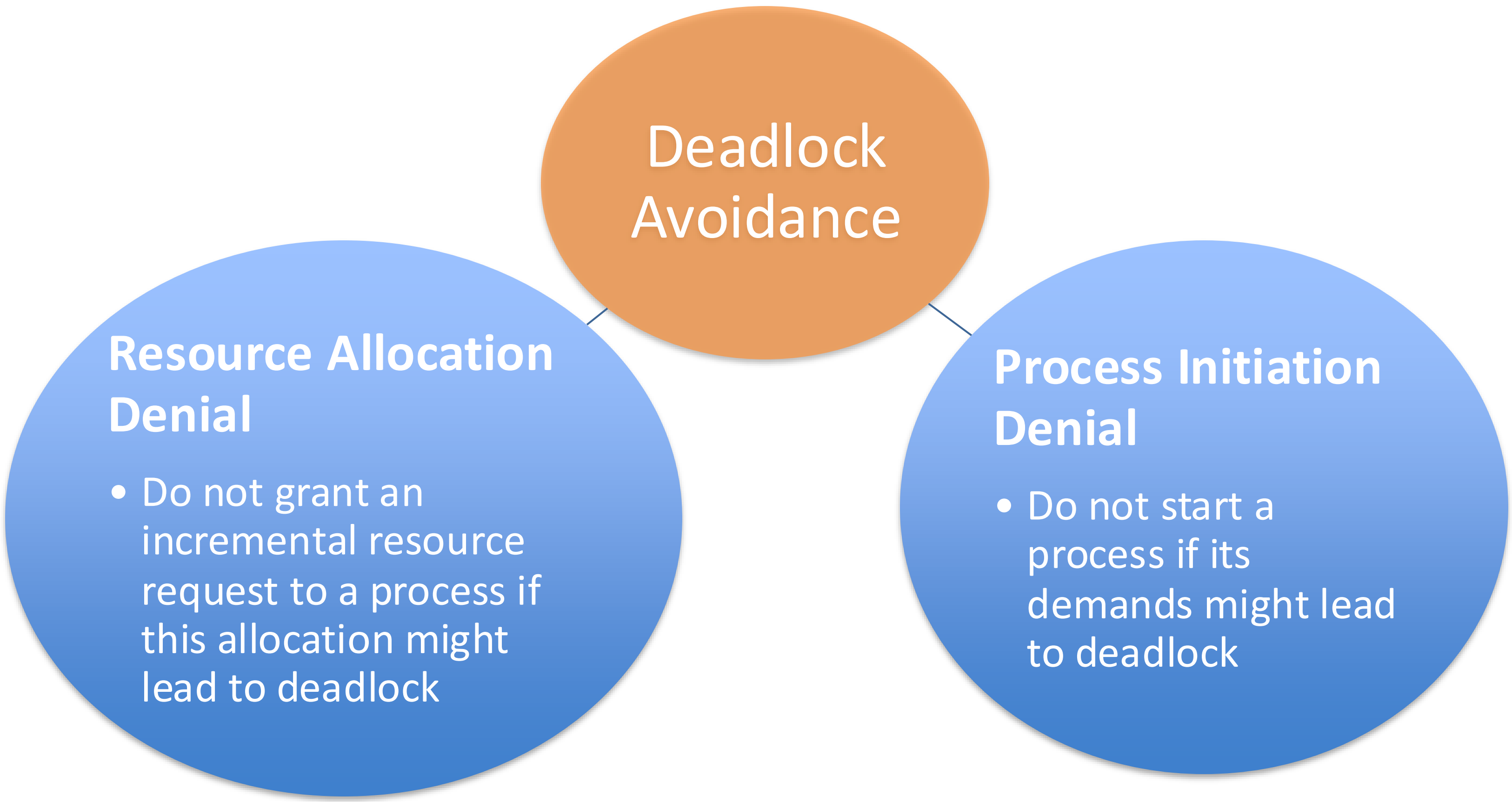


https://www.tutorialspoint.com/process-deadlocks-in-operating-system

# 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

| 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 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

| 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 6.2

# Linux Atomic Operations

(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

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

- Three types of kernel semaphores:
  - Binary semaphores
  - Counting semaphores
  - Reader-writer semaphores

# Table 6.5

# Linux Semaphores

| Traditional Semaphores | |
|---|---|
| void sema init(struct semaphore *sem, int count) | Initializes the dynamically created semaphore to the given count |
| void init MUTEX(struct semaphore *sem) | Initializes the dynamically created semaphore with a count of 1 (initially unlocked) |
| void init MUTEX LOCKED(struct semaphore *sem) | Initializes the dynamically created semaphore with a count of 0 (initially locked) |
| void down(struct semaphore *sem) | Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable |
| int down interruptible(struct semaphore *sem) | 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 |
| int down trylock(struct semaphore *sem) | Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable |
| void up(struct semaphore *sem) | Releases the given semaphore |
| **Reader-Writer Semaphores** | |
| void init rwsem(struct rw_semaphore, *rwsem) | Initializes the dynamically created semaphore with a count of 1 |
| void down read(struct rw semaphore, *rwsem) | Down operation for readers |
| void up read(struct rw semaphore, *rwsem) | Up operation for readers |
| void down write(struct rw_semaphore, *rwsem) | Down operation for writers |
| void up write(struct rw semaphore, *rwsem) | 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() |

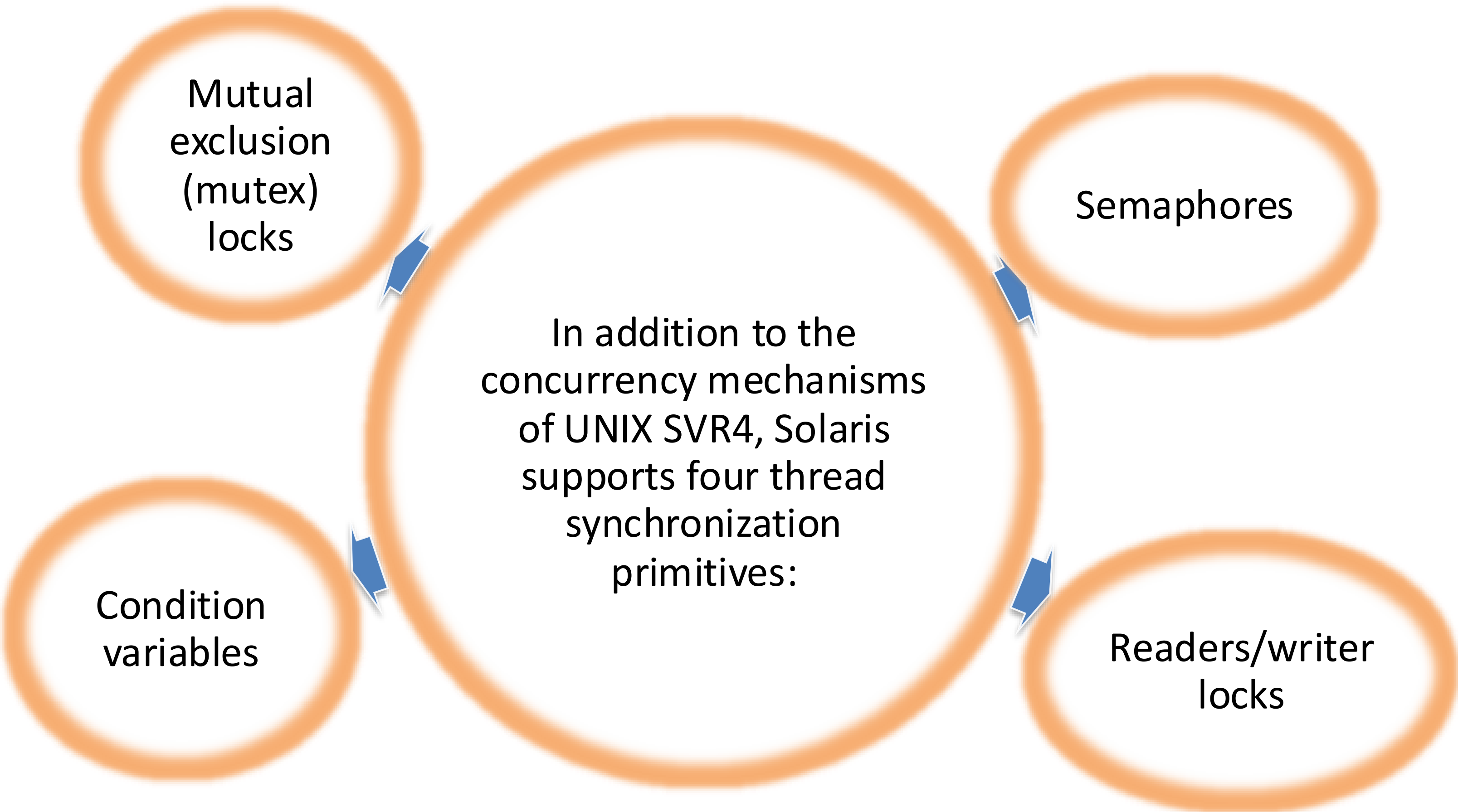
SMP = symmetric multiprocessor
UP = uniprocessor

(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 liburcu
- 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() Decrements the semaphore, potentially blocking the thread
- sema_v() Increments the semaphore, potentially unblocking a waiting thread
- sema_tryp() 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