CS 332/532 Systems Programming

Lecture 22

Processes in OS

Professor : Mahmut Unan – UAB CS

Agenda

- Process Image
- Thread vs Process
- Concurrency
- Deadlock
- Signals
- Pipes

Process Image

- Program or set of programs to be executed
- Data
- Stack

Process Identification

- Each process is assigned a unique numeric identifier
  - Otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier
- Many of the tables controlled by the OS may use process identifiers to cross-reference process tables
- Memory tables may be organized to provide a map of main memory with an indication of which process is assigned to each region
  - Similar references will appear in I/O and file tables
- When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication
- When processes are allowed to create other processes, identifiers indicate the parent and descendents of each process

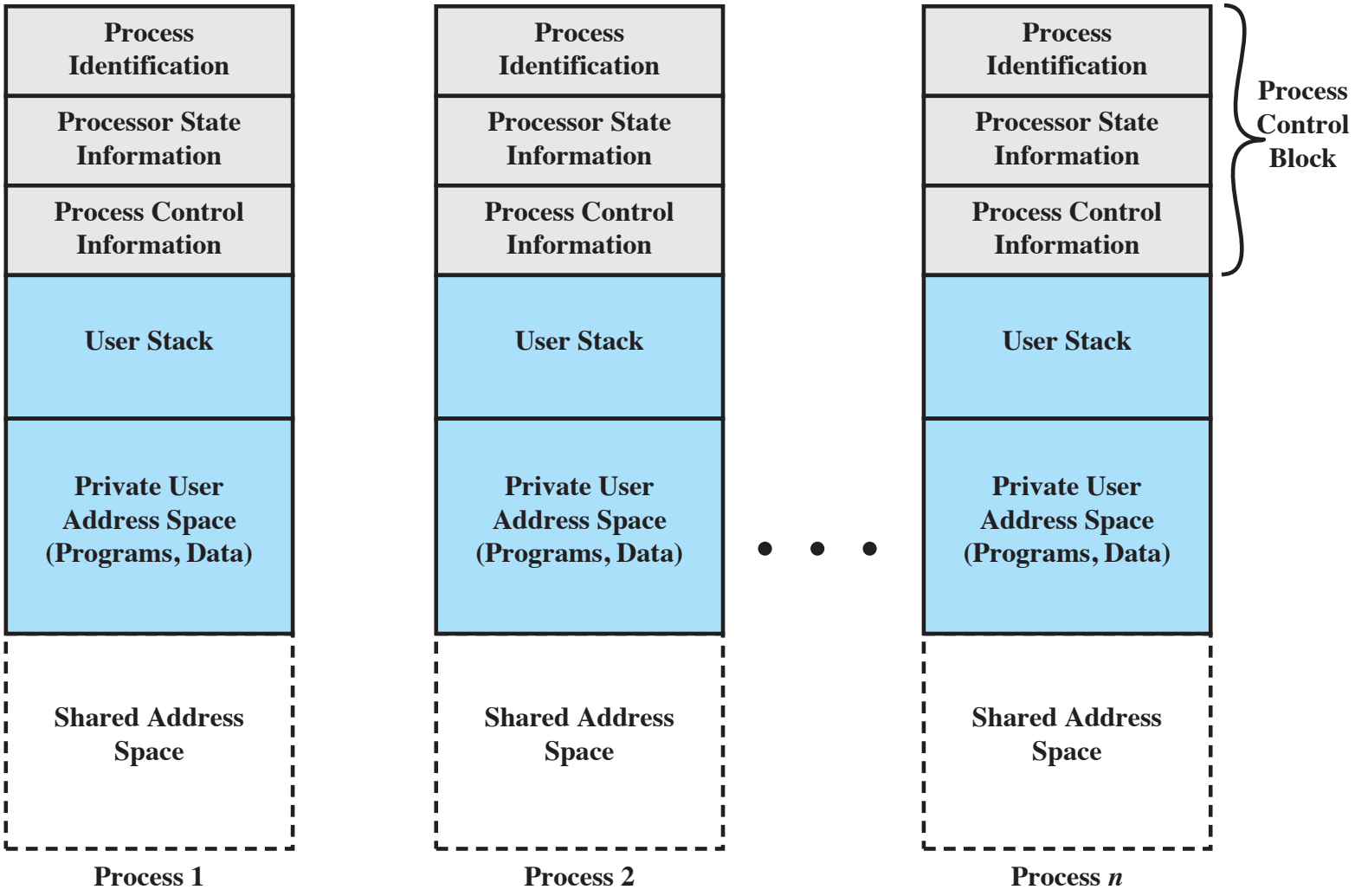


Figure 3.13 User Processes in Virtual Memory

Modes of Execution

User Mode

- Less-privileged mode
- User programs typically execute in this mode

System Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

Processes and Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***Multithreading -*** The ability of an OS to support multiple, concurrent paths of execution within a single process

Threads

- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control.
- Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS is an example

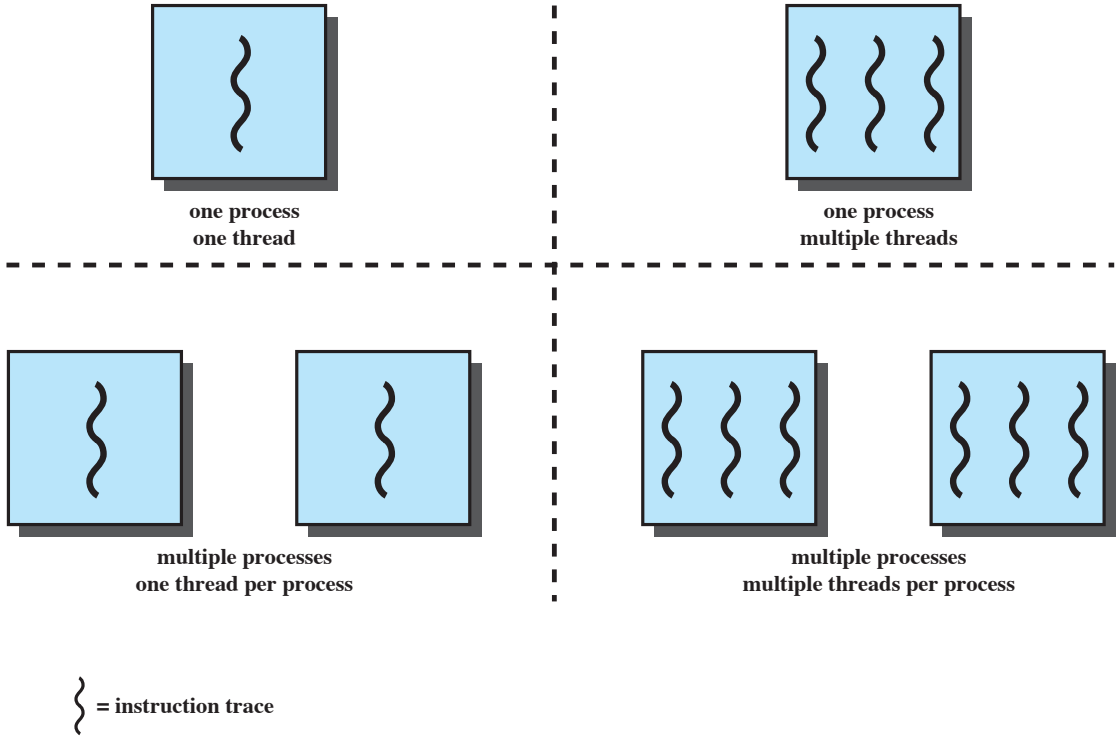


Figure 4.1 Threads and Processes

Multithreaded Approaches

- The right half of the figure depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

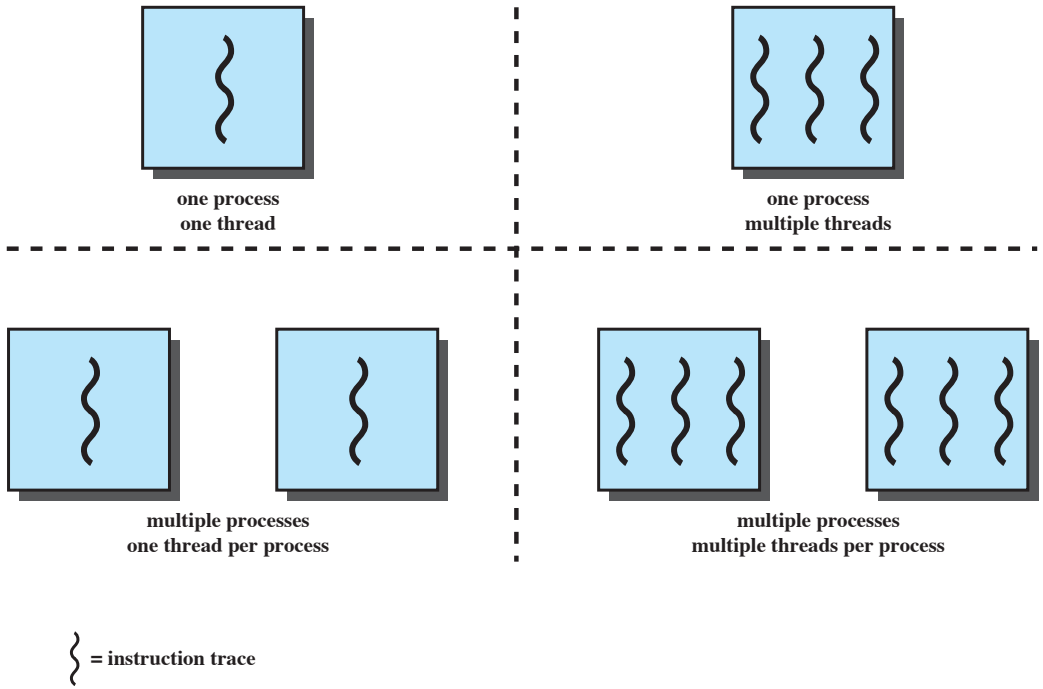
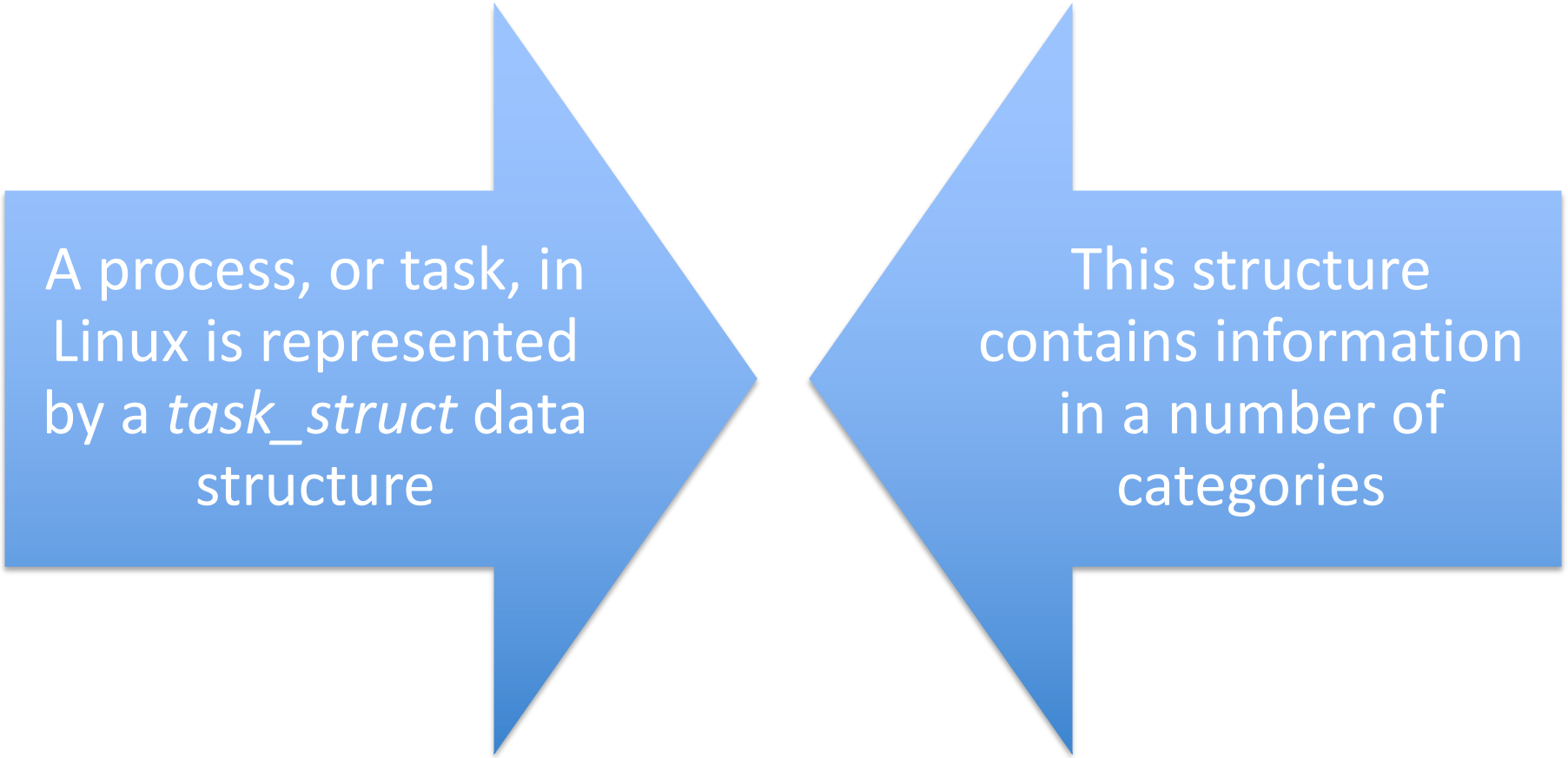


Figure 4.1 Threads and Processes

Linux Tasks



• State, • Scheduling information, • Identifiers, • Interprocess communication, • Links, • Times and timers, • File system, • Address space, • Processor-specific context:

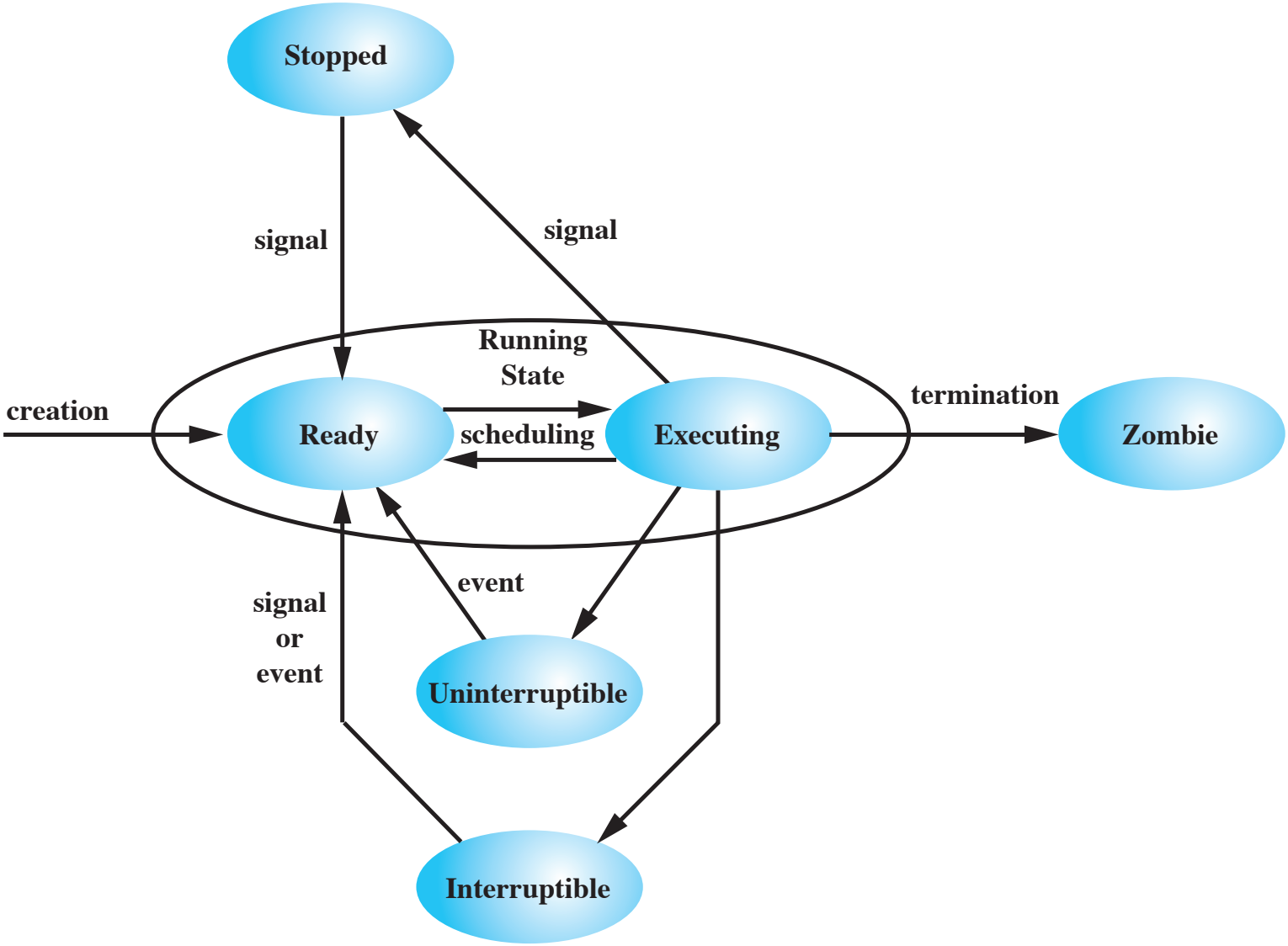
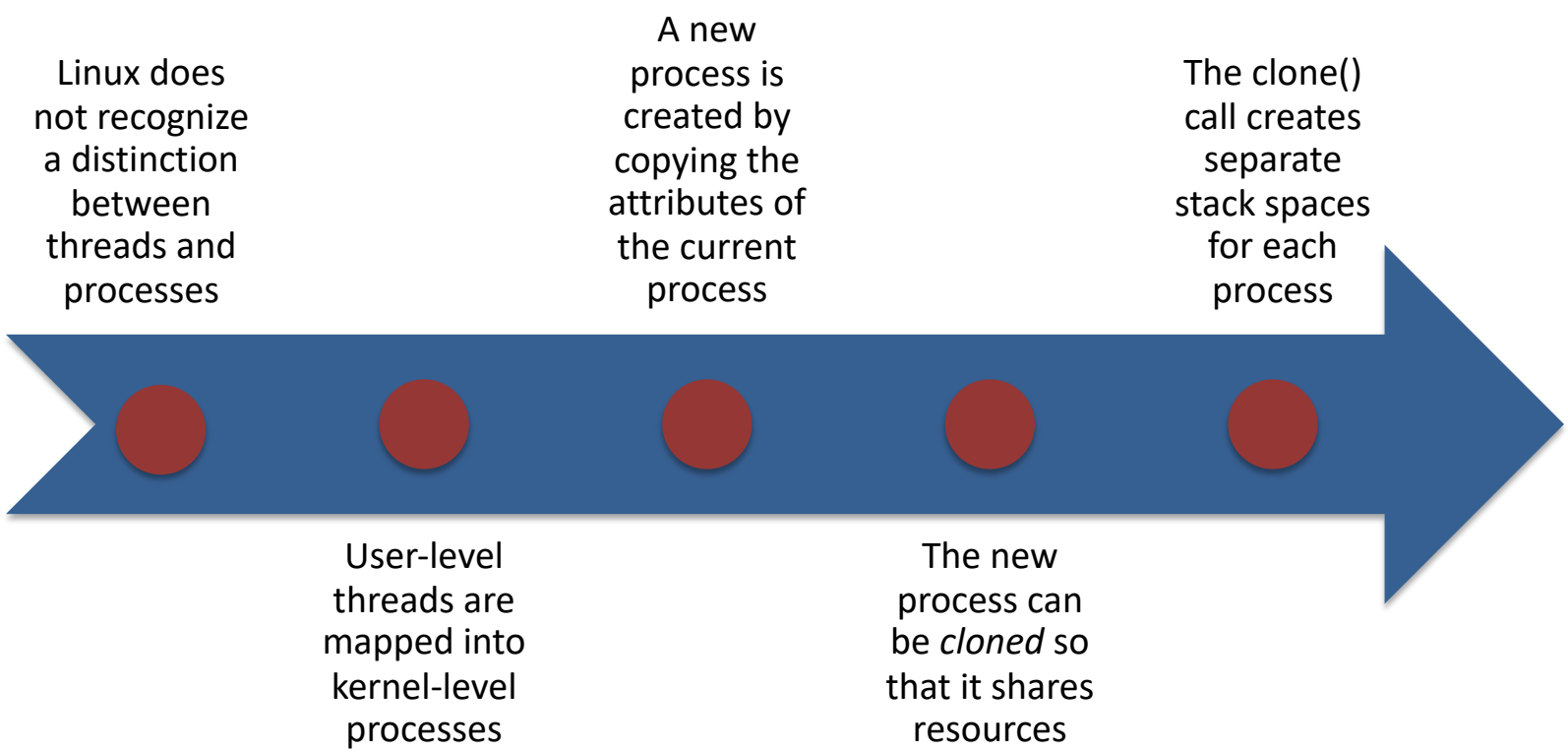


Figure 4.15 Linux Process/Thread Model

Linux Threads



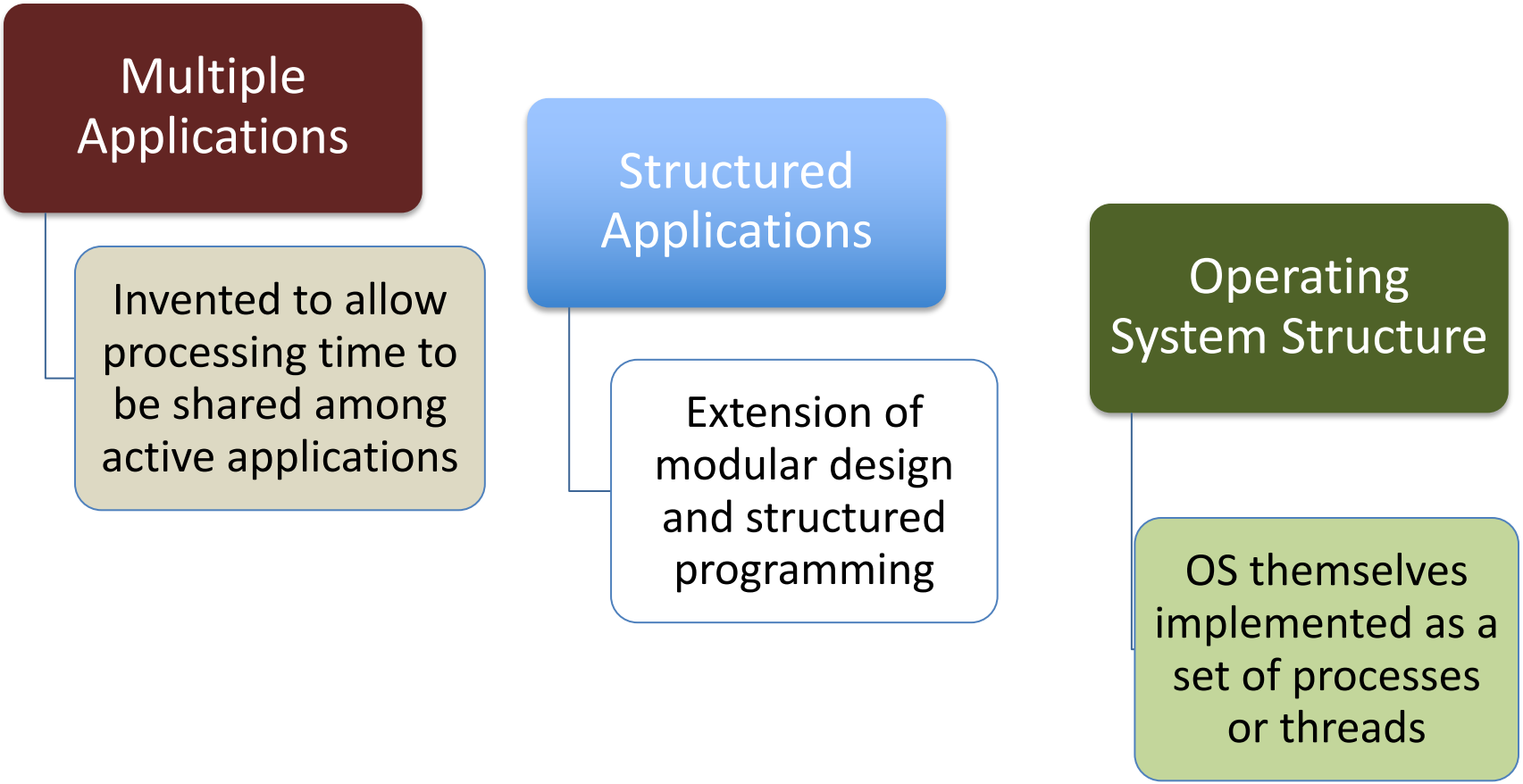
Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
    - » The management of multiple processes within a uniprocessor system
  - Multiprocessing
    - » The management of multiple processes within a multiprocessor
  - Distributed Processing
    - » The management of multiple processes executing on multiple, distributed computer systems
    - » The recent proliferation of clusters is a prime example of this type of system

Concurrency

- Fundamental to all of these areas, and fundamental to OS design, is **concurrency**.
- Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes.
- We shall see that these issues arise not just in multiprocessing and distributed processing environments but even in single-processor multiprogramming systems.

Concurrency Arises in Three Different Contexts:



Concurrency

- the basic requirement for support of concurrent process is the ability to enforce **mutual exclusion**
  - the ability to exclude all other processes from a course of action while one process is granted that ability
- Approaches to achieve mutual conclusion
  - Software solutions
  - Hardware solutions
  - OS
  - Compilers
  - Semaphores, monitors, message passing….

Principles of Concurrency

- Interleaving and overlapping
  - Can be viewed as examples of concurrent processing
  - Both present the same problems
- Uniprocessor – the relative speed of execution of processes cannot be predicted
  - Depends on activities of other processes
  - The way the OS handles interrupts
  - Scheduling policies of the OS

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**

`Semaphore`	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 **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	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).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	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 *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	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).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	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	Communication
• To enforce mutual exclusion	• 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*

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

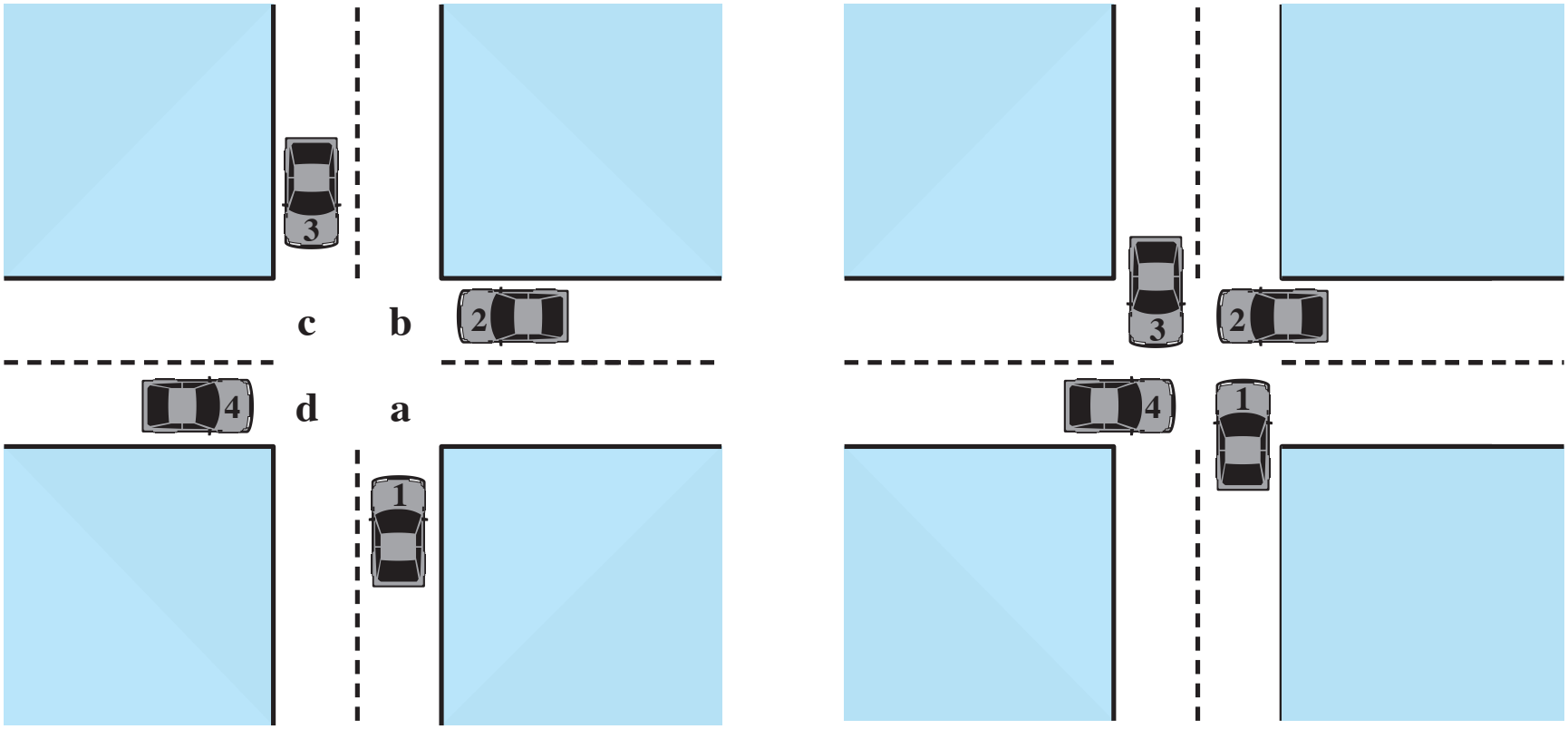
Format
- Content
- Length
  - fixed
  - variable

Queueing Discipline
- FIFO
- Priority

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

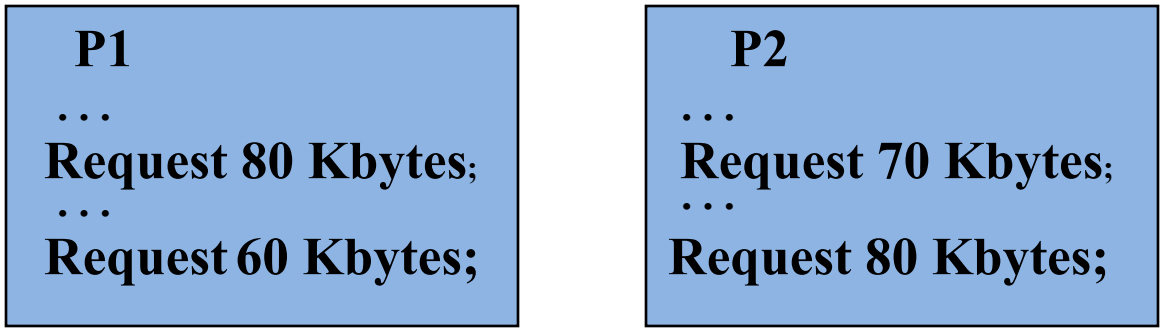
- Can be safely used by only one process at a time and is not depleted by that use
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

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:



- 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 prevention**
  - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening

- **Deadlock avoidance**
  - Do not grant a resource request if this allocation might lead to deadlock

- **Deadlock detection**
  - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

Conditions for Deadlock

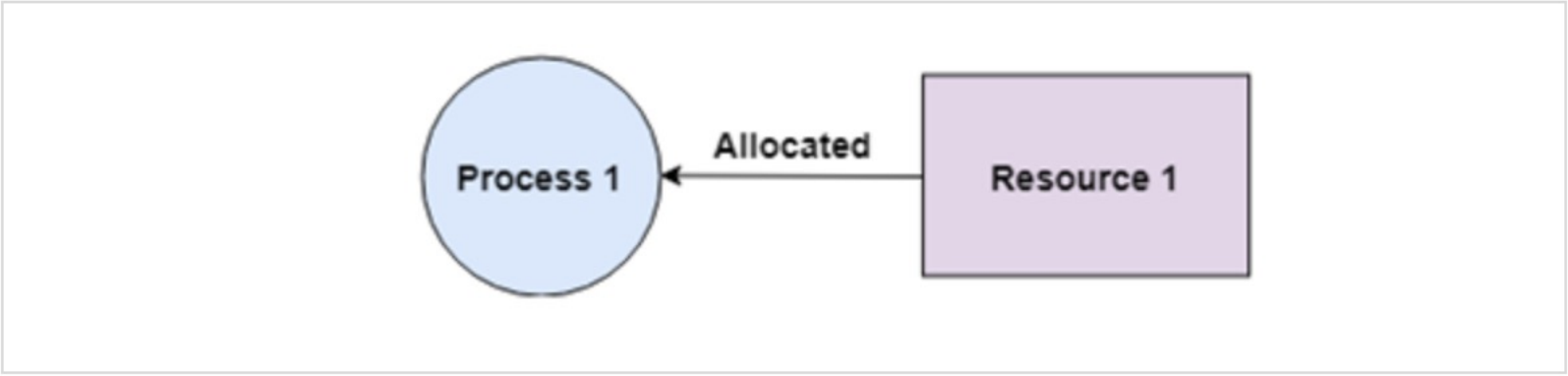
Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• Only one process may use a resource at a time • No process may access a resource until that has been allocated to another process	• A process may hold allocated resources while awaiting assignment of other resources	• No resource can be forcibly removed from a process holding it	• 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

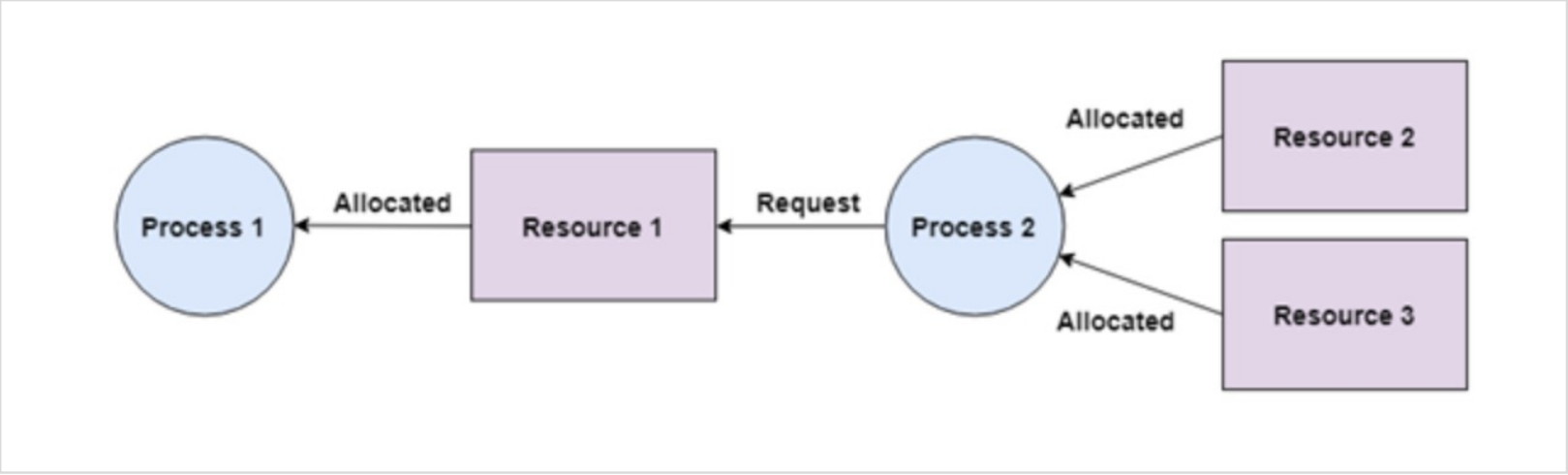
- Mutual exclusion
  - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
  - Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
  - Even in this case, deadlock can occur if more than one process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



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

Deadlock Condition Prevention

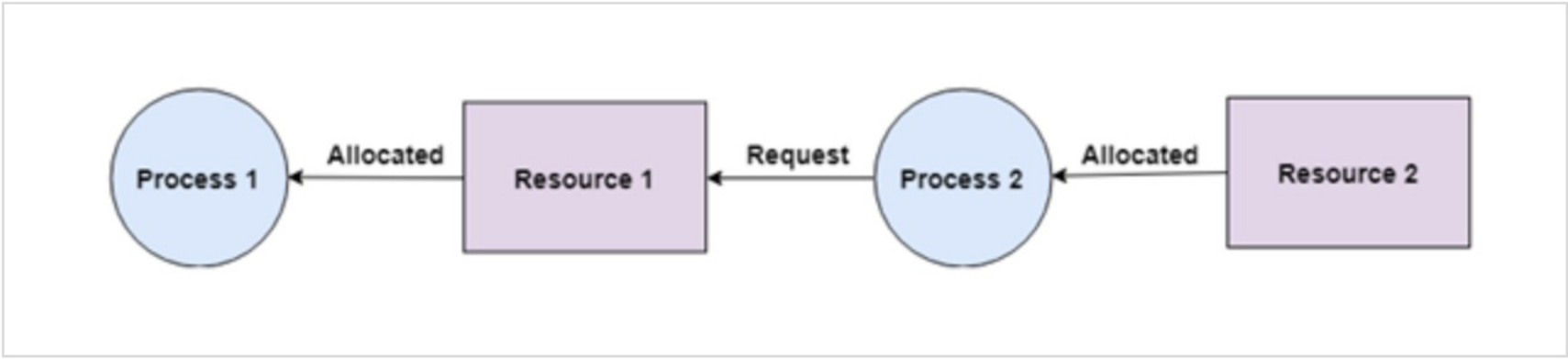
- Hold and wait
  - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously



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

Deadlock Condition Prevention

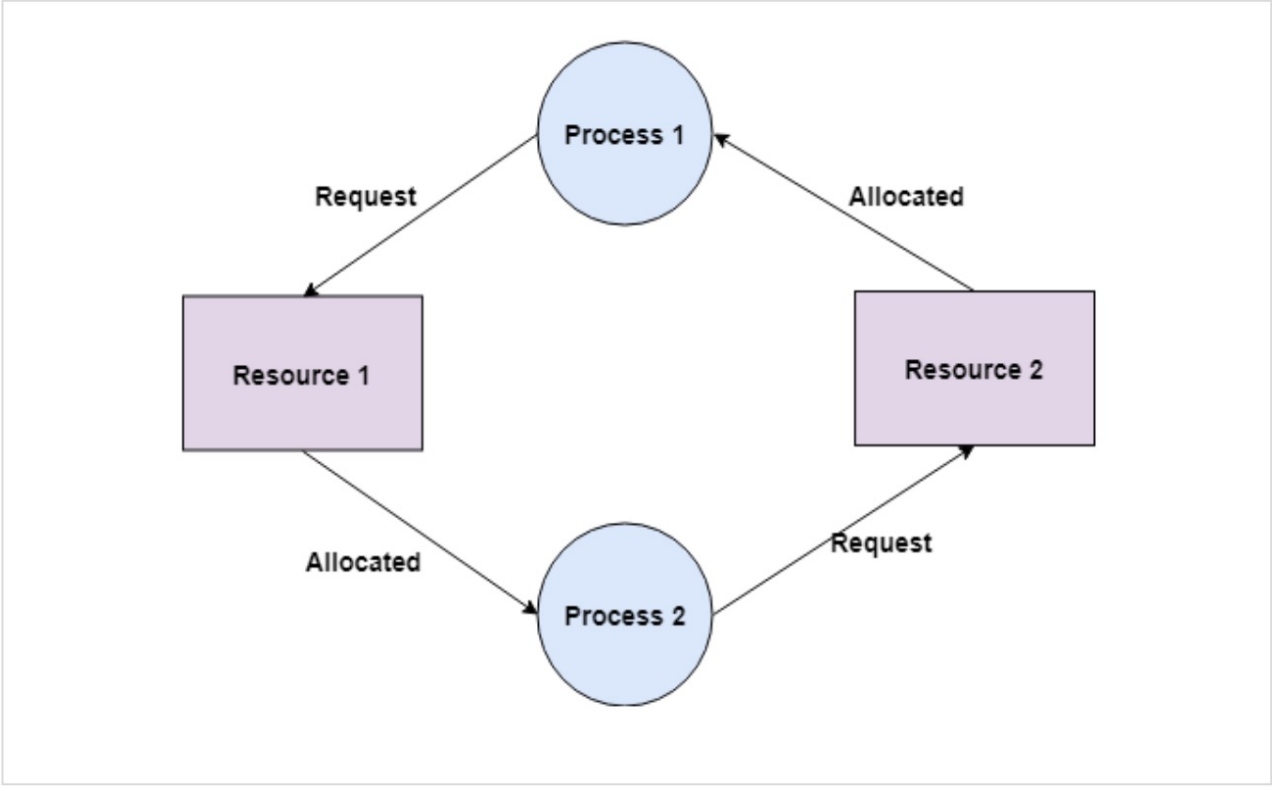
- No Preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release its resources



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

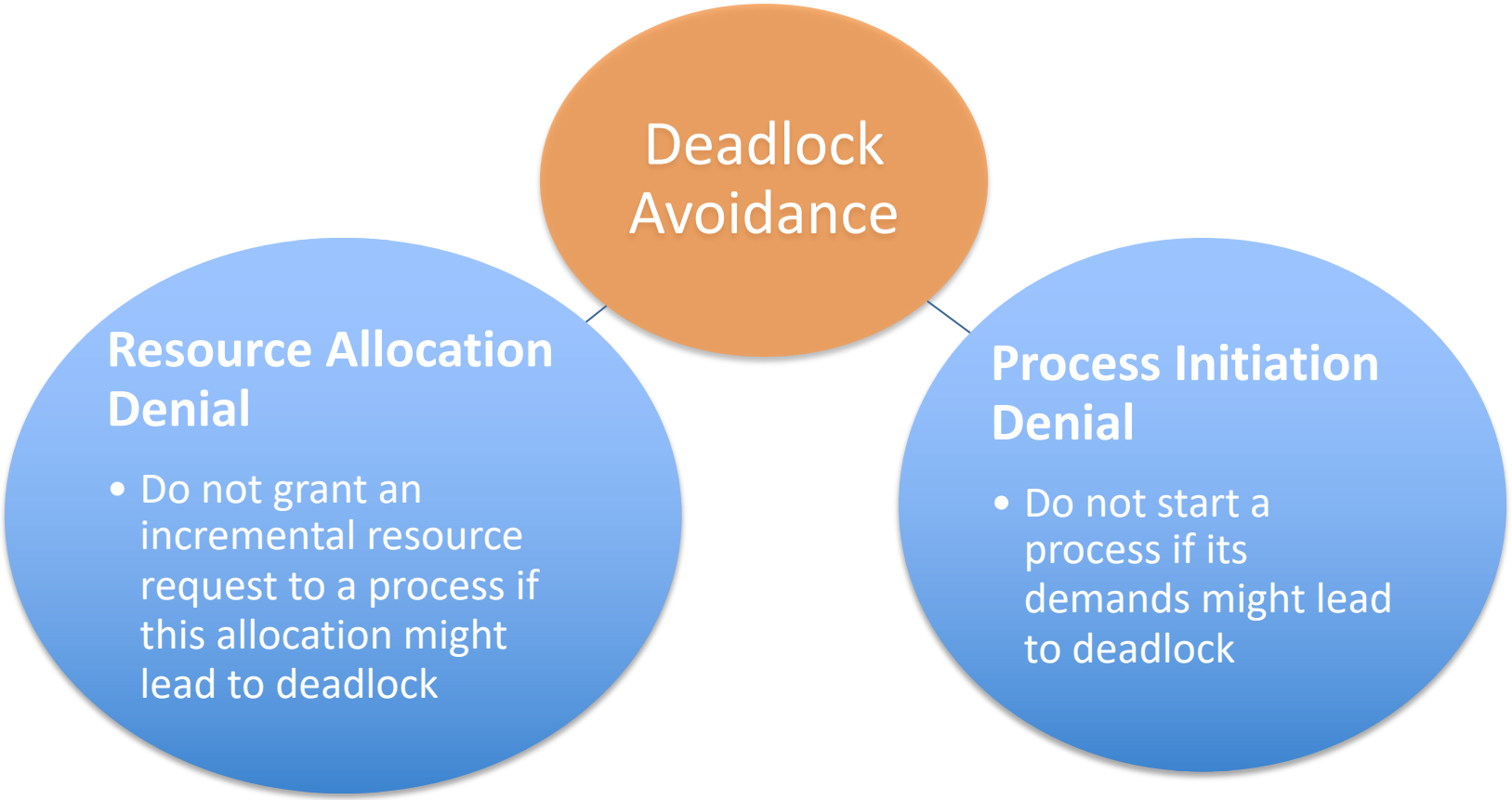


https://www.tutorialspoint.com/process-deadlocks-in-operating-system

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

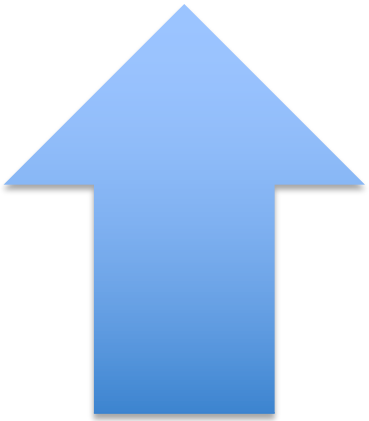
- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

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

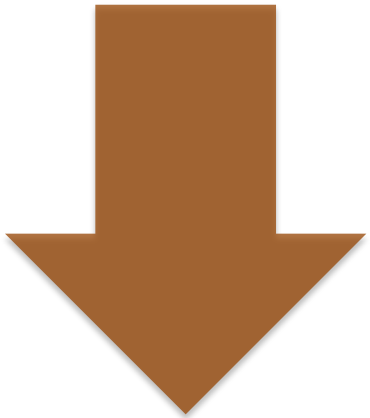
Deadlock Detection Algorithm

- 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

- 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

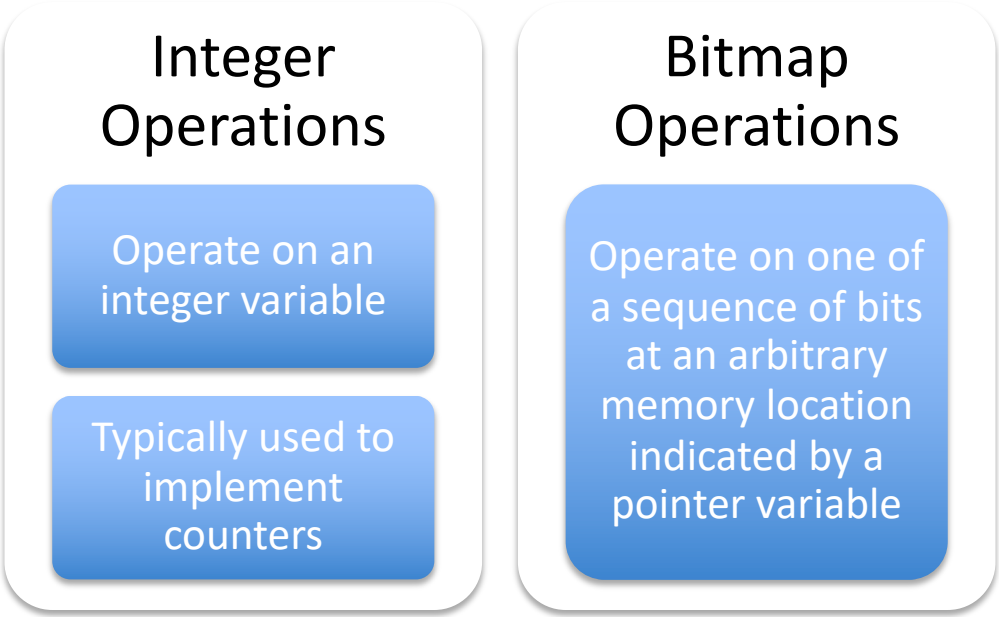
(Table can be found on page 288 in textbook)

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:



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

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 6.5

Linux Semaphores

(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()

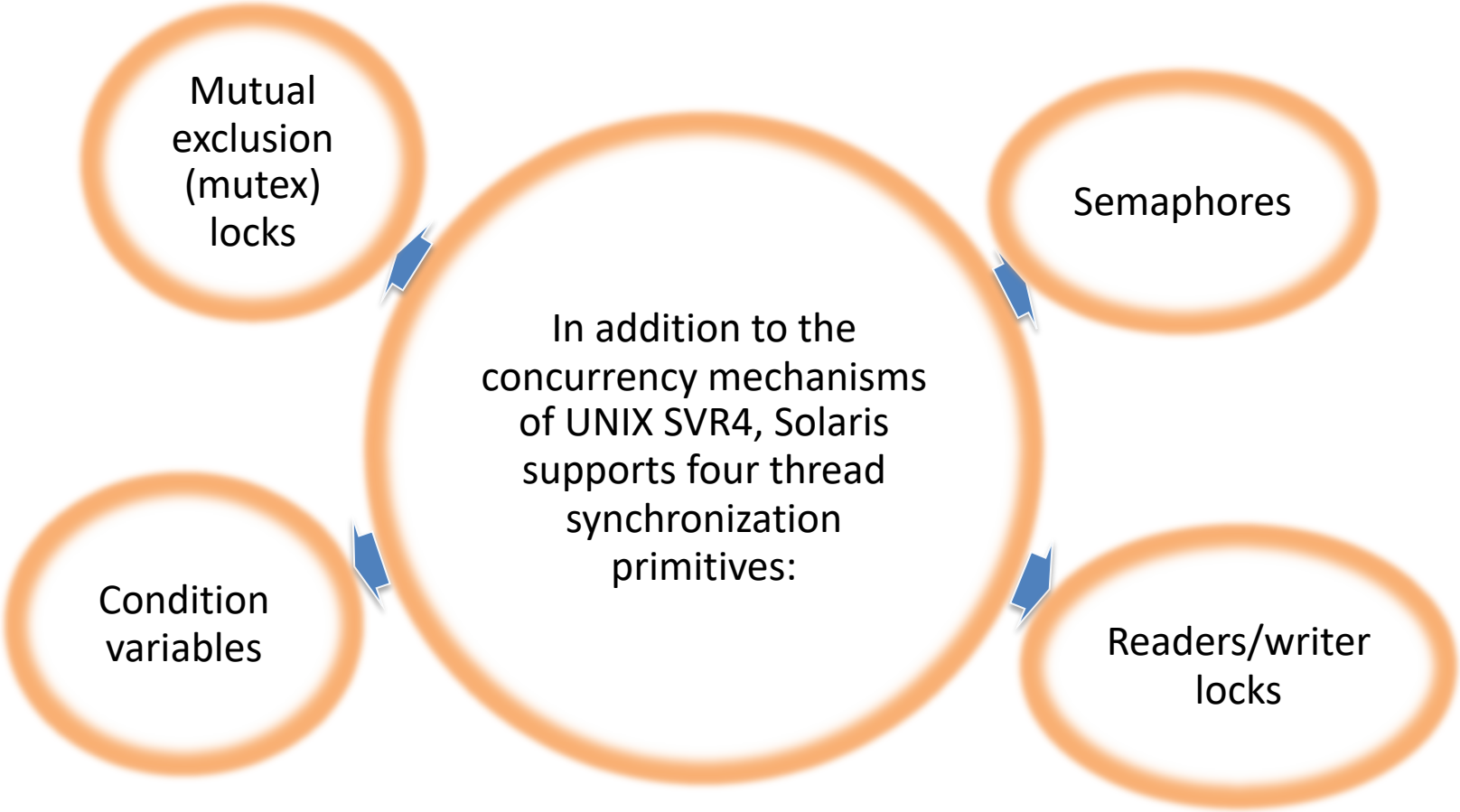
SMP = symmetric multiprocessor
UP = uniprocessor

(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