

CS 332/532 Systems Programming

Lecture 21

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

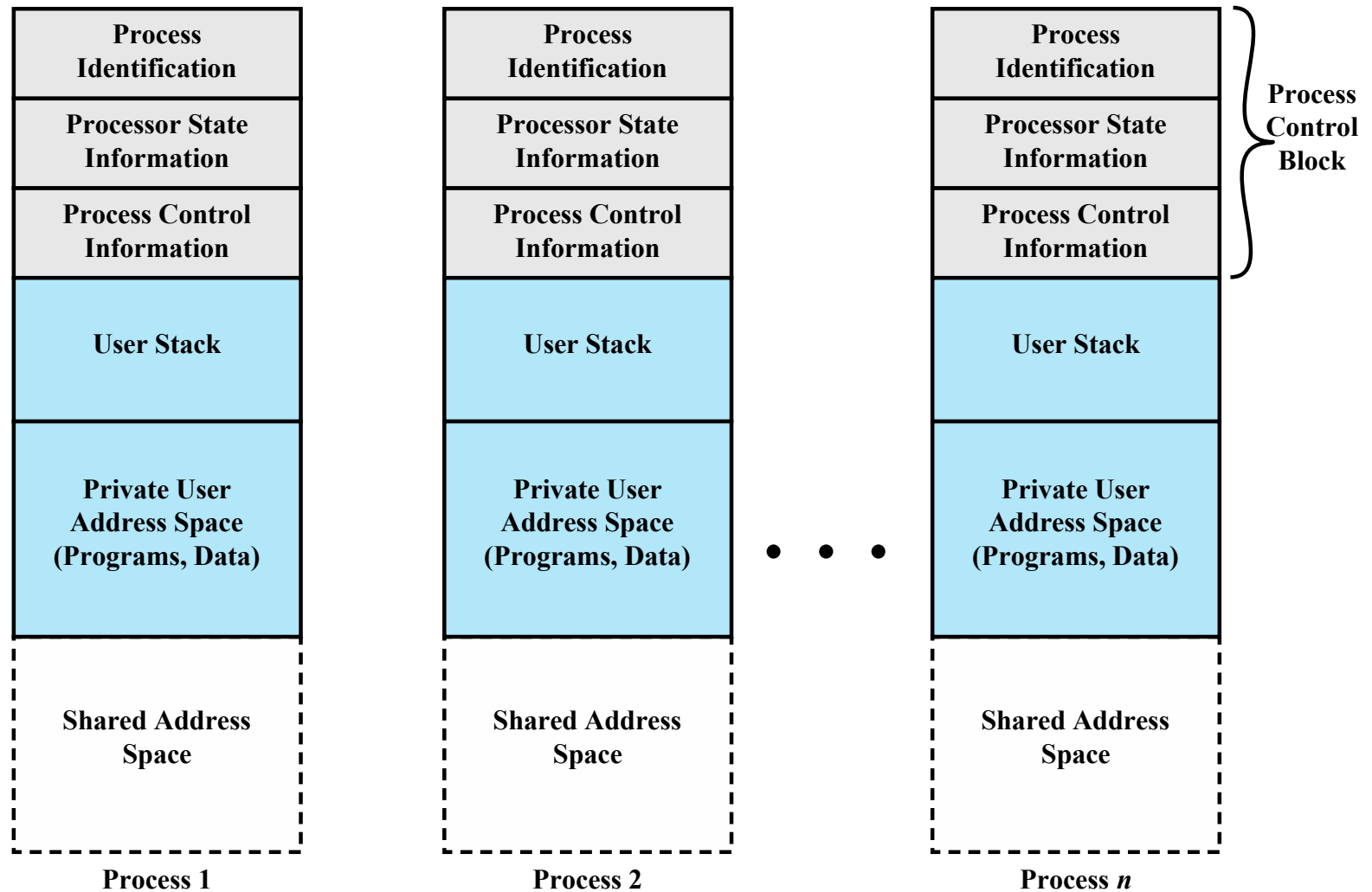


Figure 3.13 User Processes in Virtual Memory

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.

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

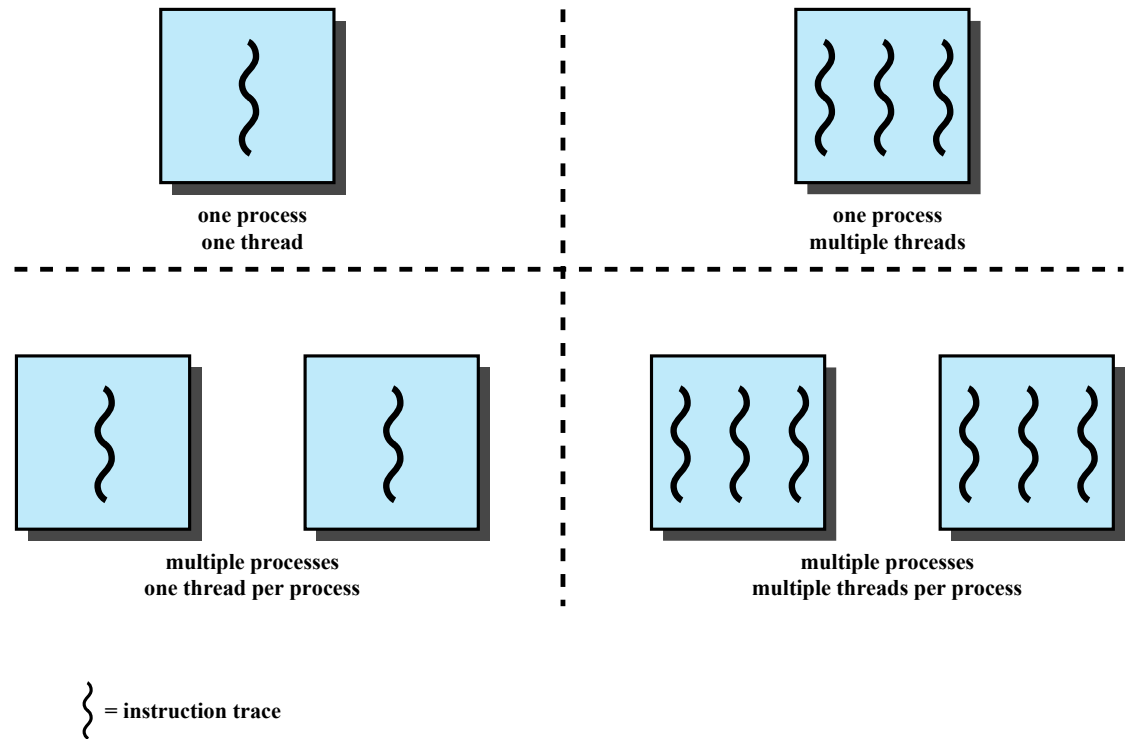


Figure 4.1 Threads and Processes

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

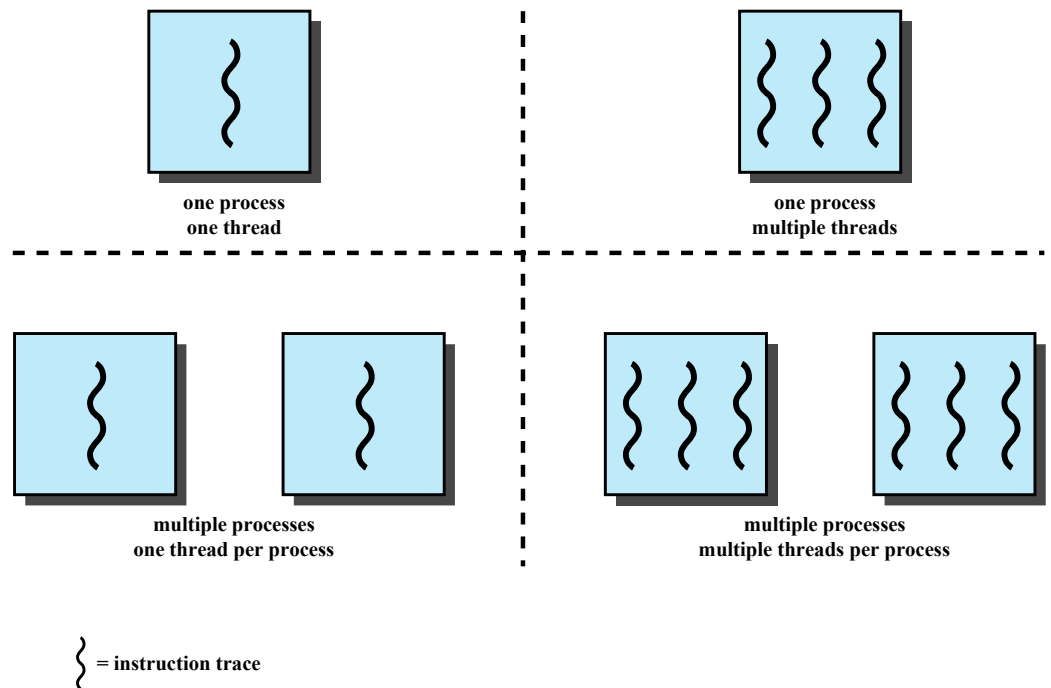
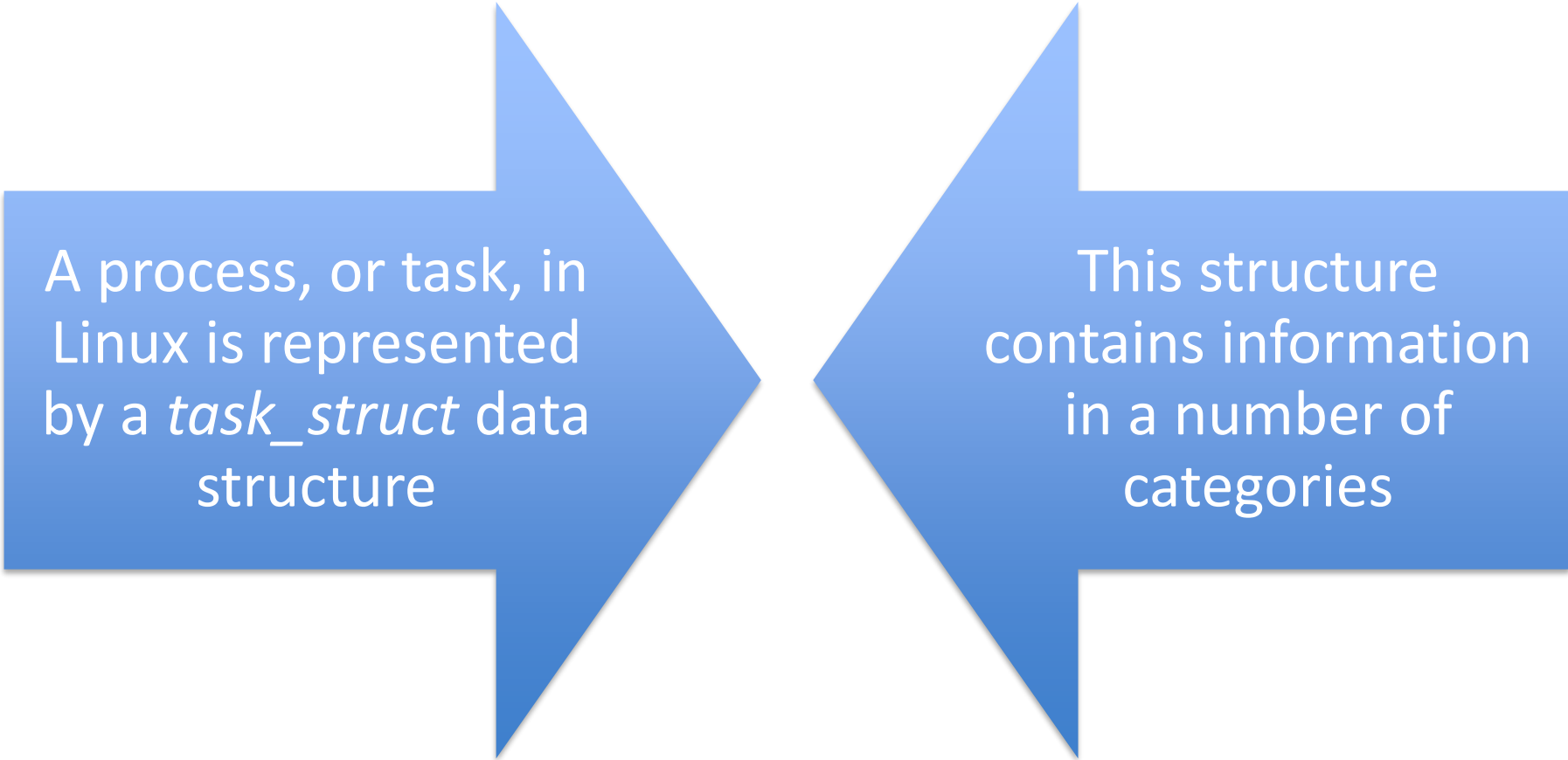


Figure 4.1 Threads and Processes

Linux Tasks



A process, or task, in Linux is represented by a *task_struct* data structure

This structure contains information in a number of categories

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

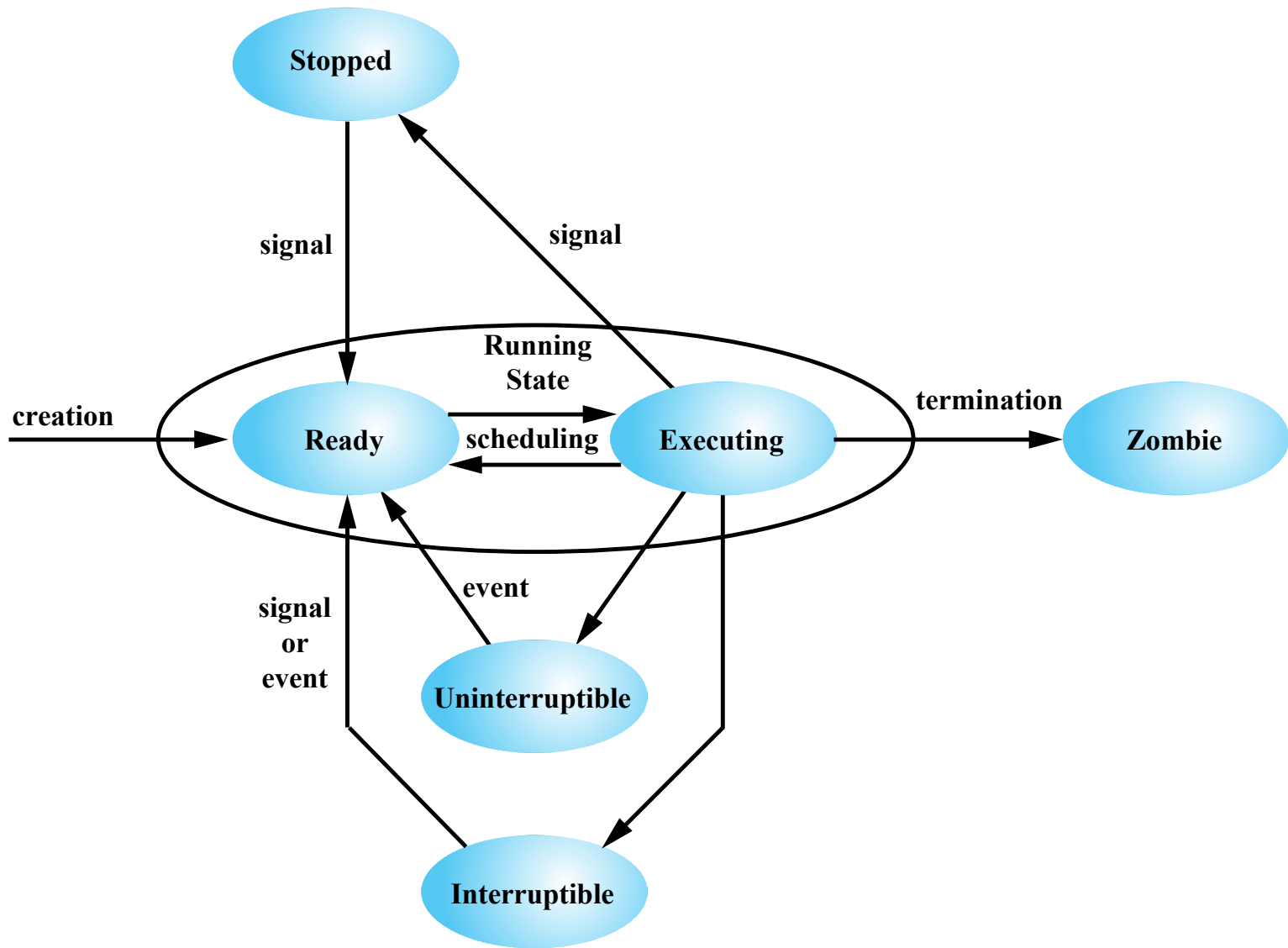
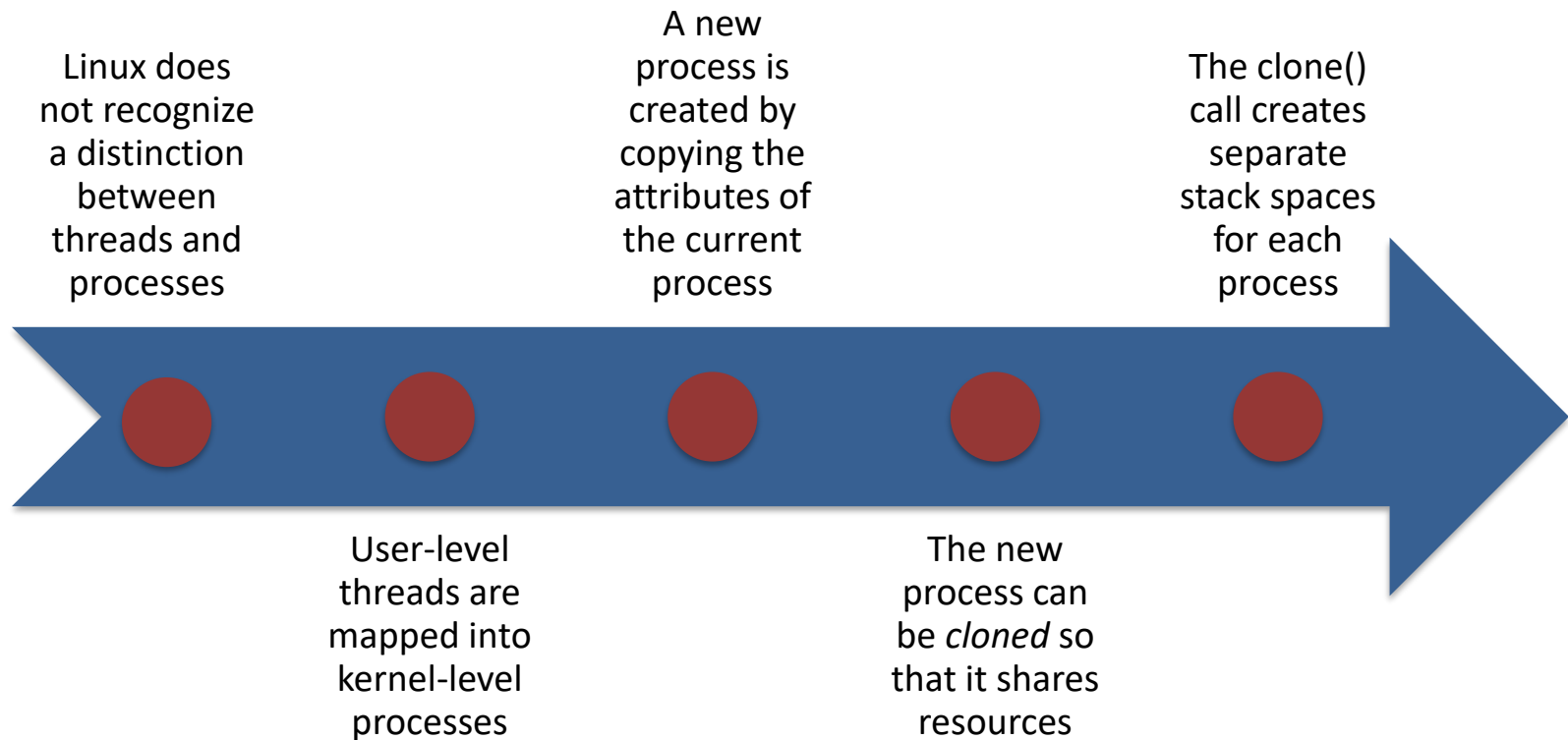


Figure 4.15 Linux Process/Thread Model

Linux Threads



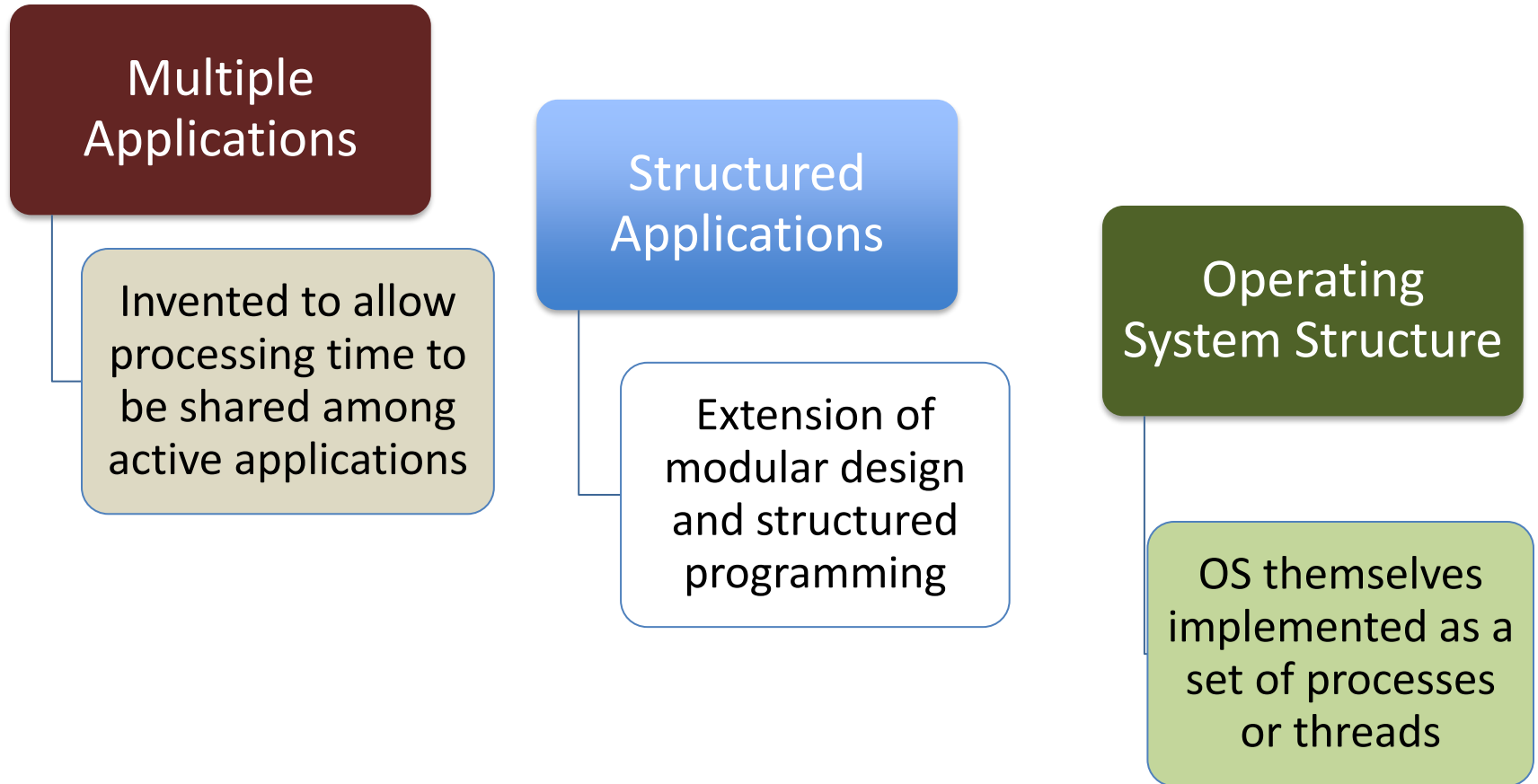
Multiple Processes

- 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 on multiple, distributed computer systems executing
 - » 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



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 <i>critical sections</i> . 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

- 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*

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

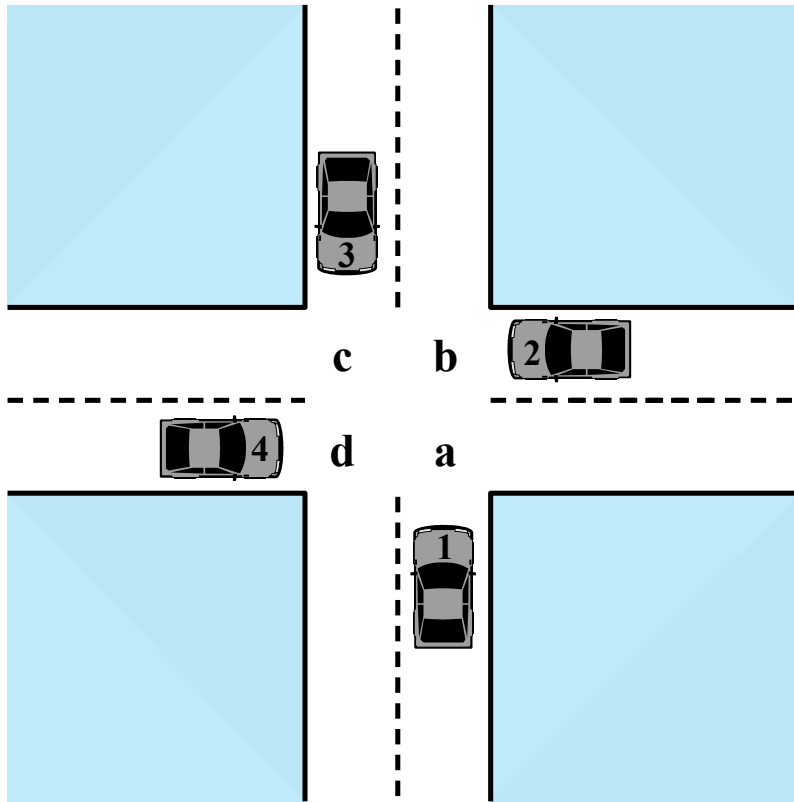
FIFO

Priority

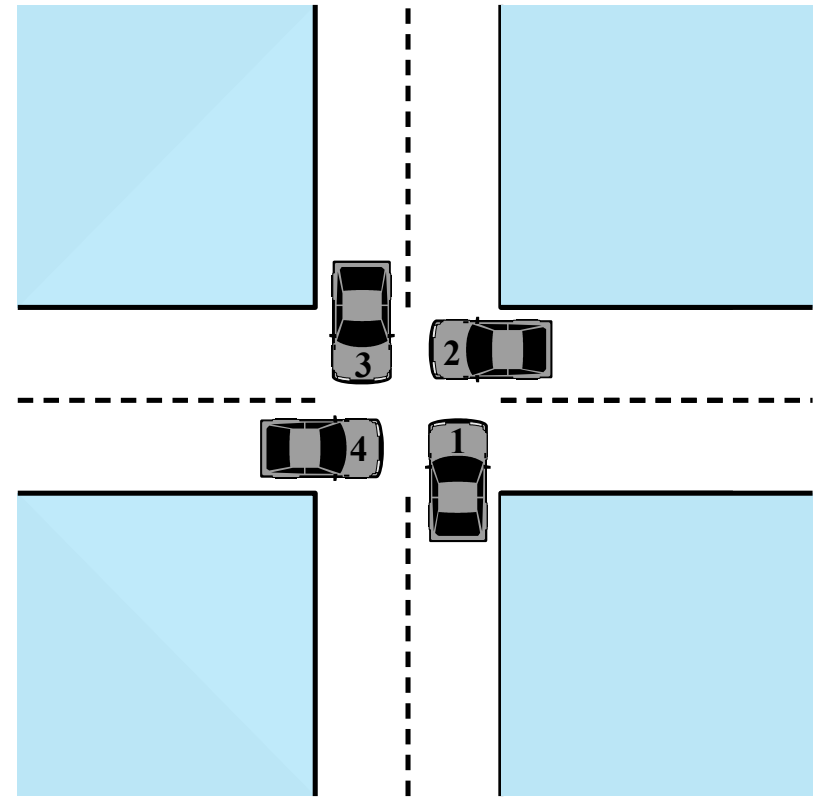
Table 5.5**Design Characteristics of Message Systems for
Interprocess Communication and Synchronization**

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



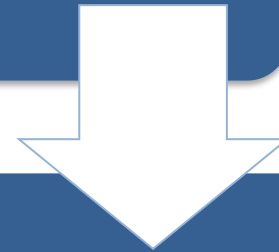
@thecrazyprogrammer

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

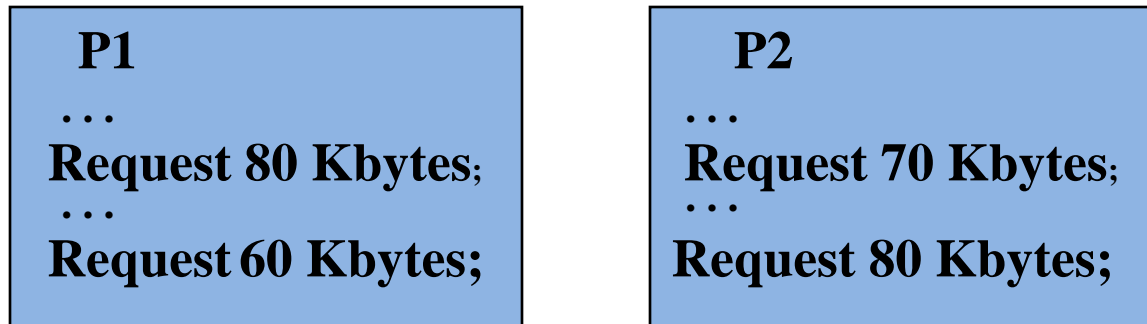


Consumable

- One that can be created (produced) and destroyed (consumed)
 - Interrupts, signals, messages, and information
 - In I/O buffers

Example 2: Memory Request

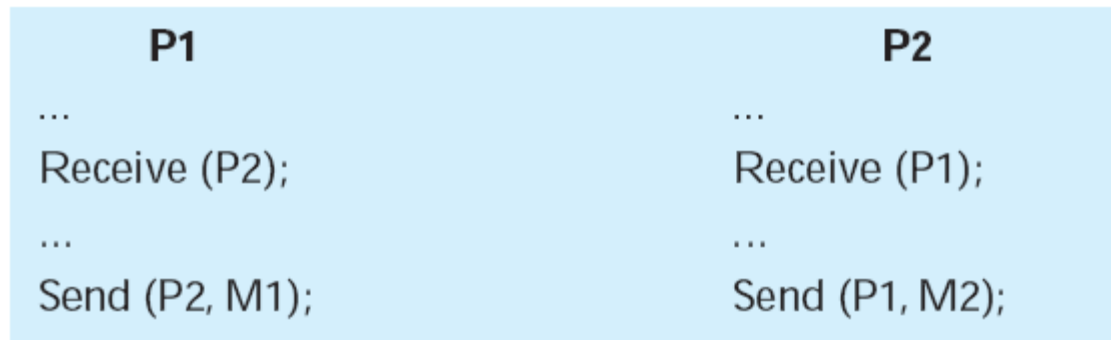
- Space is available for allocation of 200Kbytes, and the following sequence of events occur:



- 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:



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

- 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