

Operating Systems

Prepared by Dr. Alaa Mohamed

Module 8



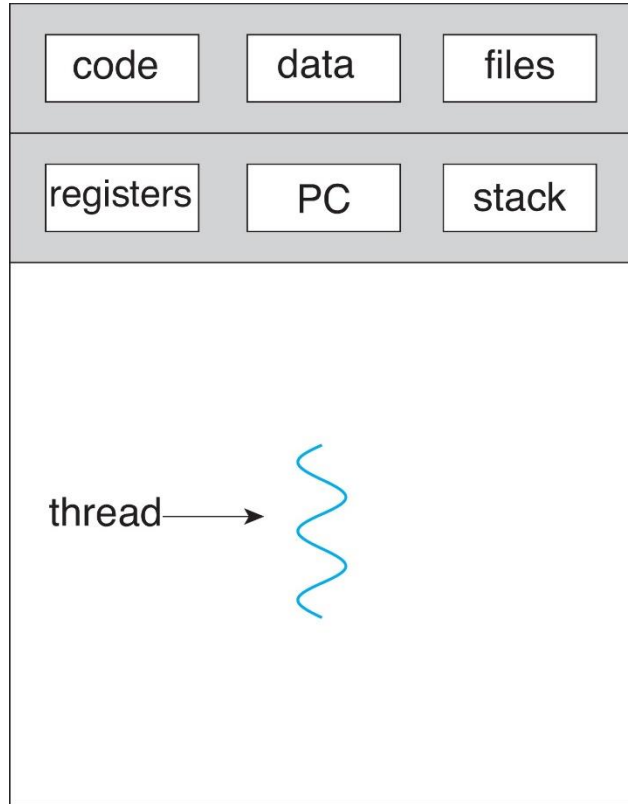
Threads & Protection in OS

- Motivation of using threads
- Multithreading Models
- Threading Issues

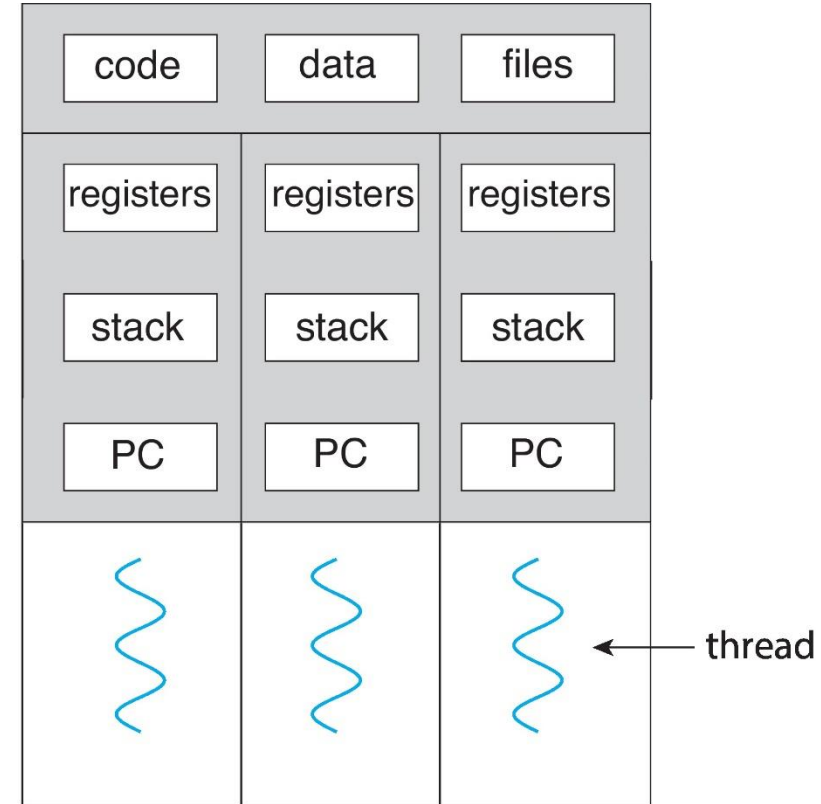
- The general structure of a process assumes that the process has an address space with a single sequential execution sequence. Which leads to low resource utilization.
- Thread is a mechanism to provide multiple execution control to the processes

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

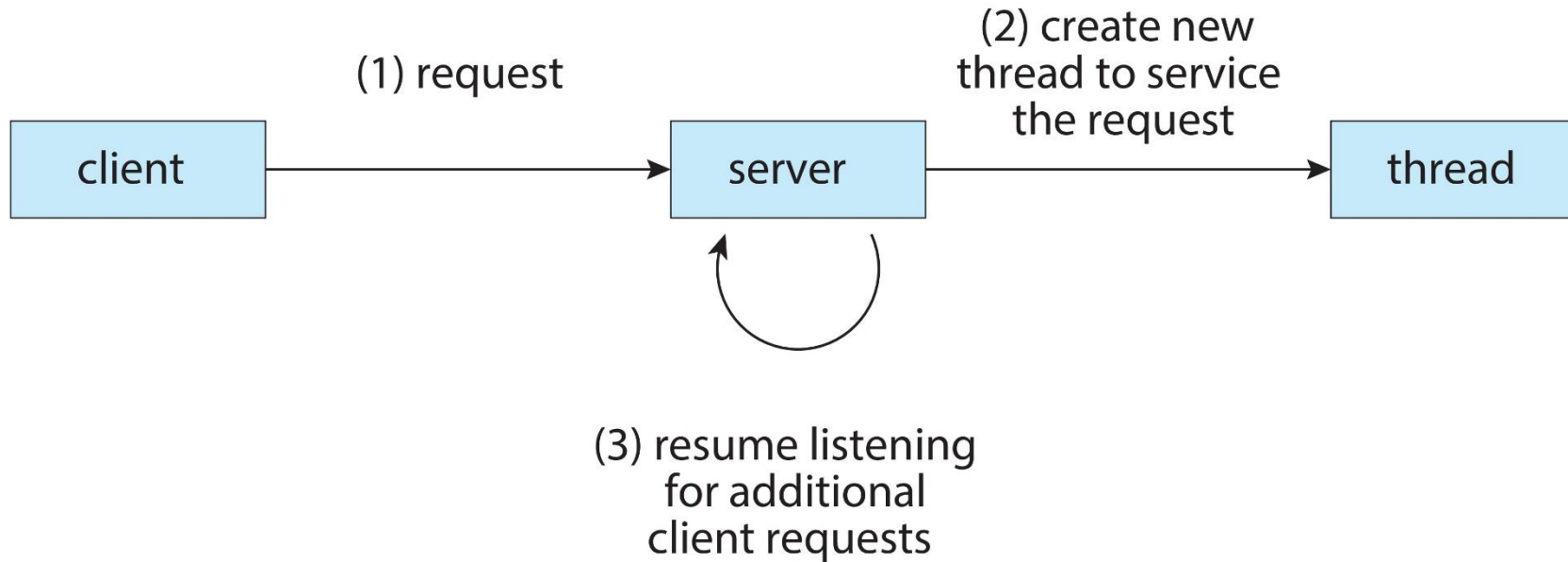
Single and Multithreaded Processes



single-threaded process



multithreaded process

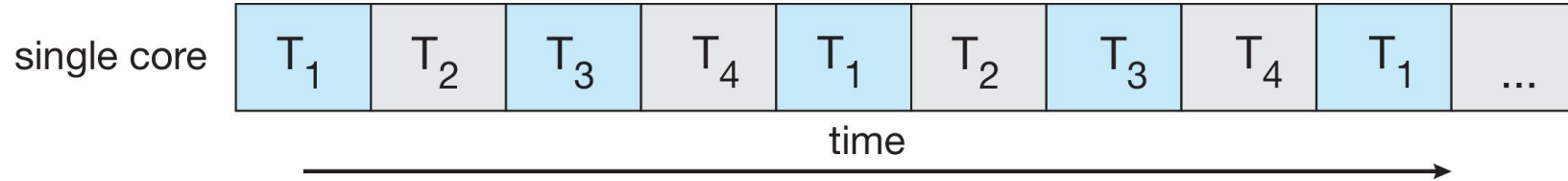


- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures

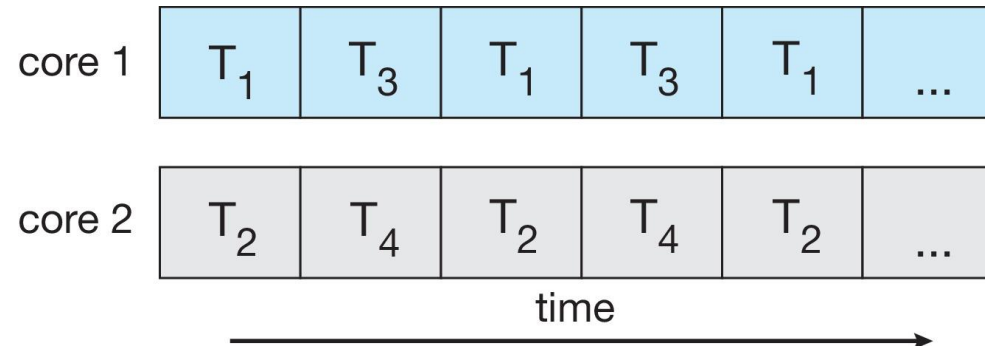
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

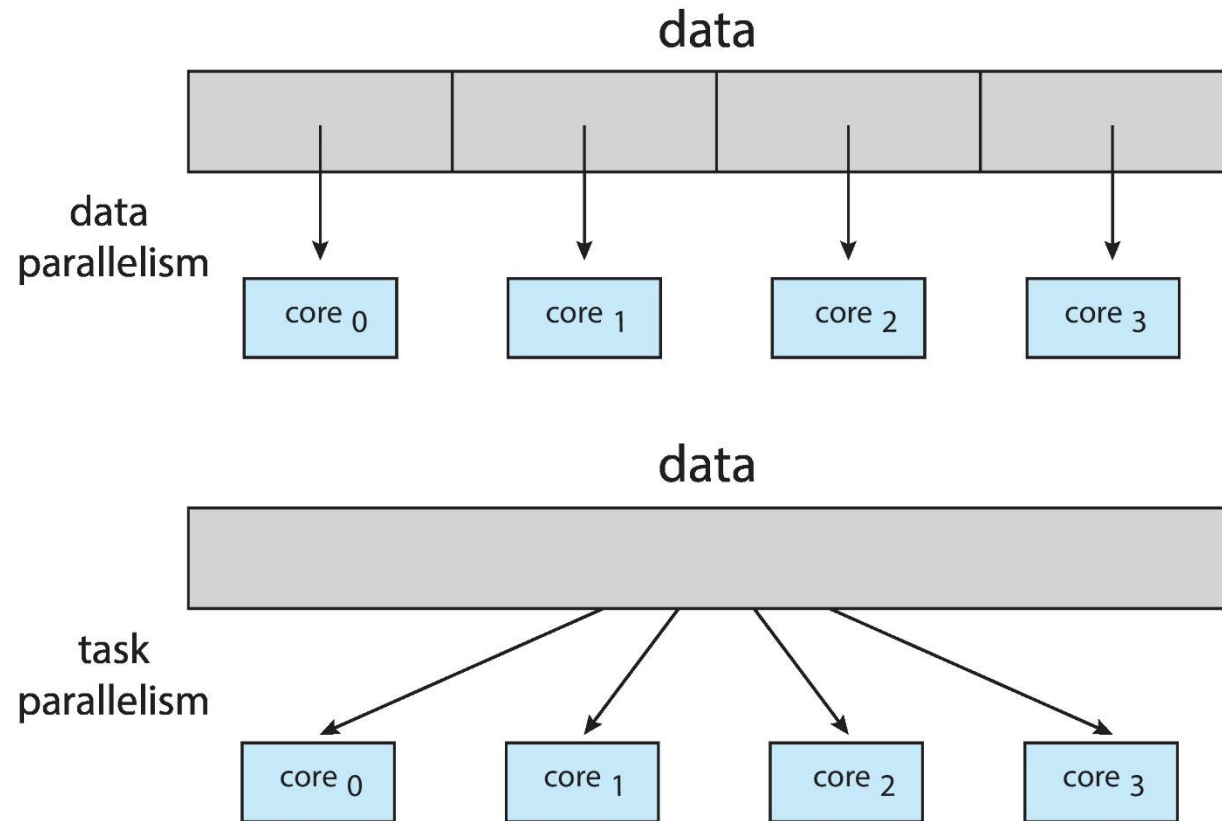


- **Parallelism on a multi-core system:**



- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation

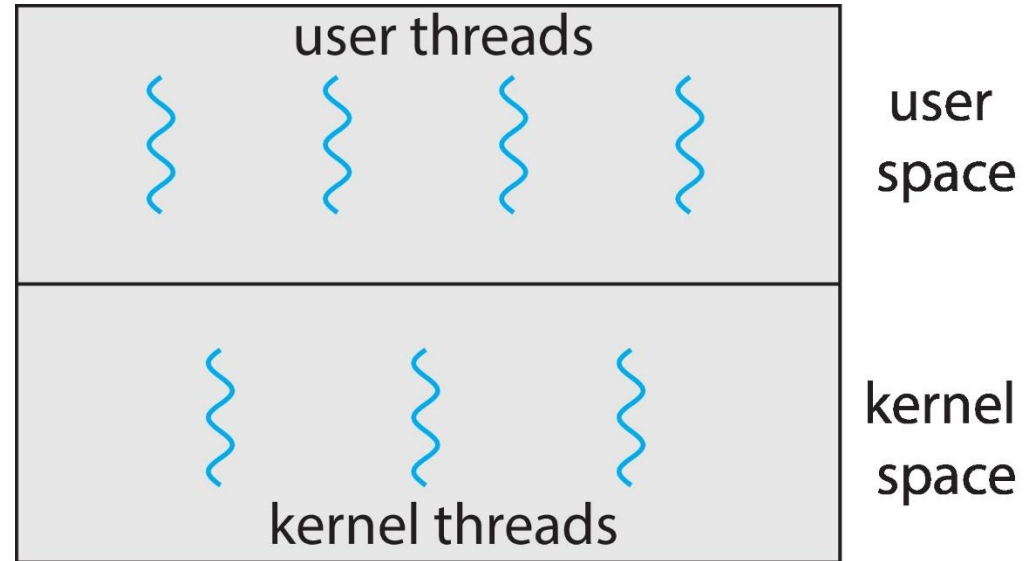
Data and Task Parallelism



User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Linux
 - Mac OS X
 - iOS
 - Android

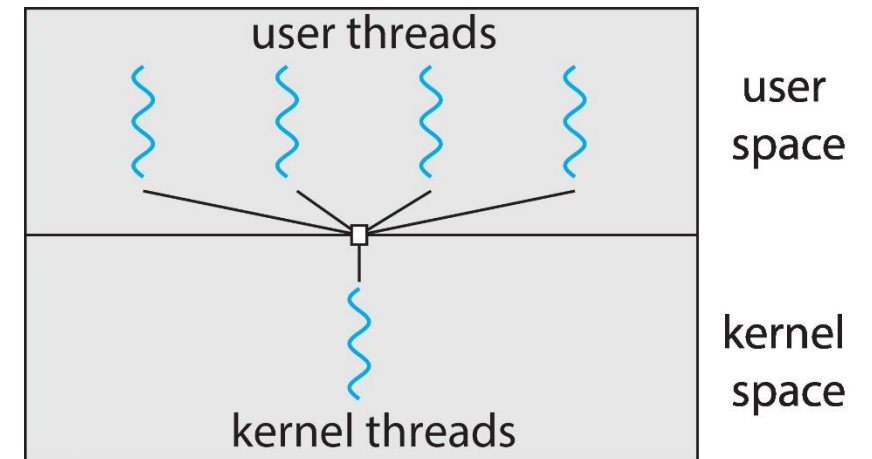
User and Kernel Threads



- Many-to-One
- One-to-One
- Many-to-Many

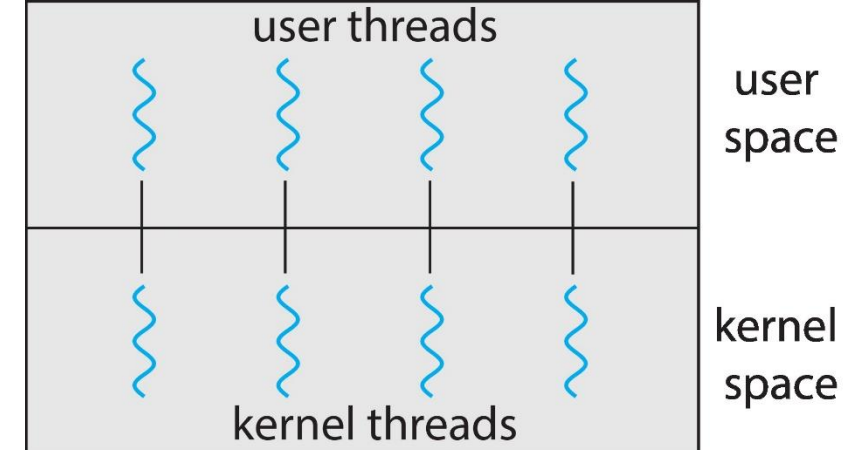


- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads



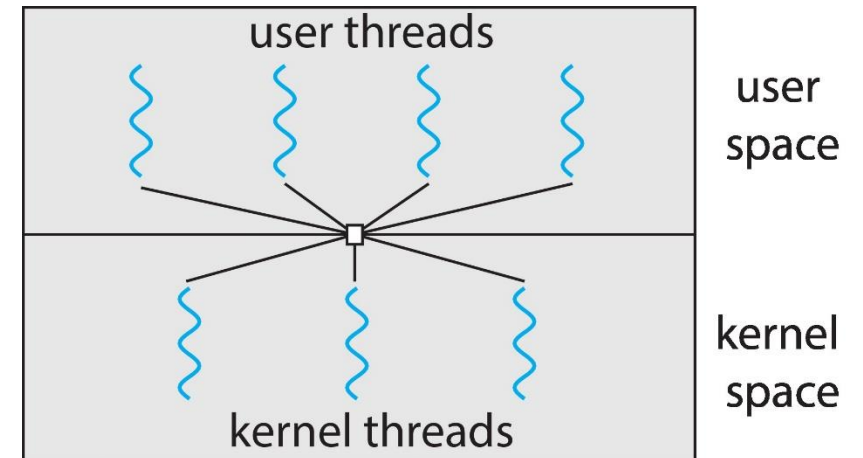
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux



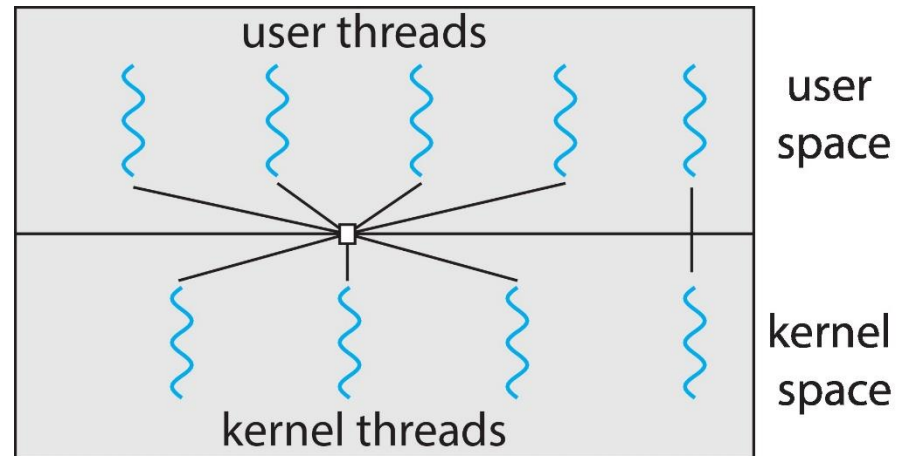
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

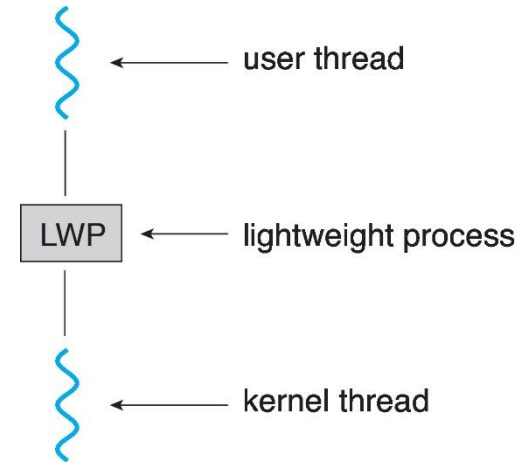
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
 - Cancellation only occurs when thread reaches **cancellation point**

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to **static** data
 - TLS is unique to each thread

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP attached to kernel thread
 - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



- Goals of Protection
- Principles of Protection
- Protection Rings
- Domain of Protection
- Access Matrix
- Revocation of Access Rights
- Access Control methods

- **Protection:** a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.
- This mechanism should impose controls together with a means of enforcement.

- In any protection model, computer consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so

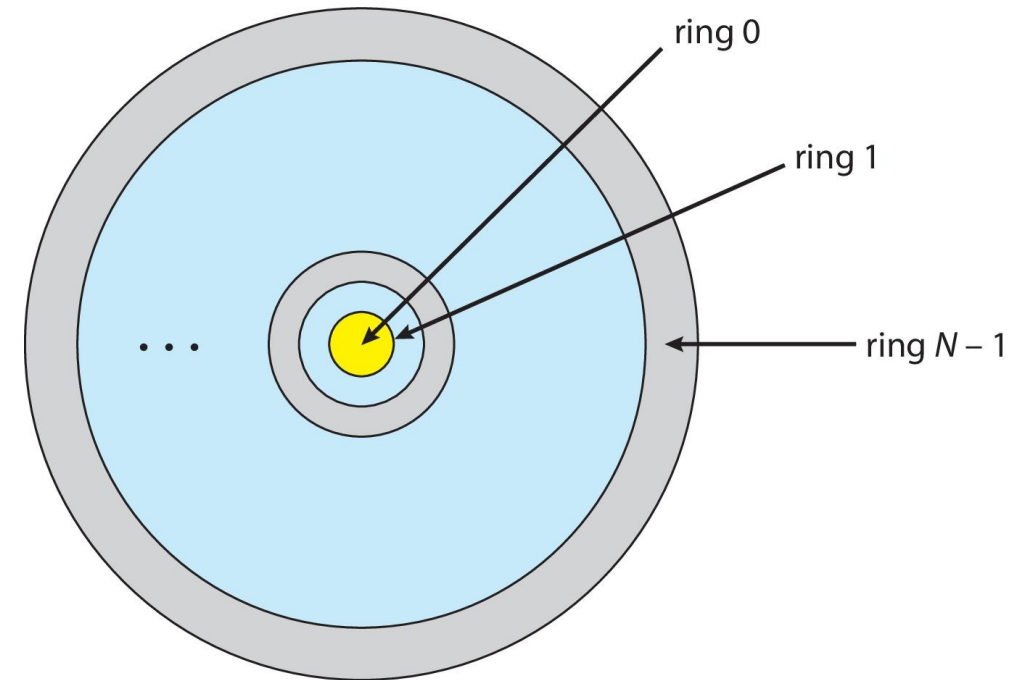
- Guiding principle – An operating system following the **principle of least privilege**
 - Programs, users and systems should be given just enough **privileges** to perform their tasks
 - So, failure or compromise of a component does the minimum damage and allows the minimum damage to be done.

- For Example:
 - **In UNIX OS**— Normal user should not run as root.
 - root user can execute privileged commands.
 - Most users respect that to avoid accidental delete operation for which there is no corresponding undelete.
- Now consider that rather than human error, damage may result from malicious attack (Virus ,....etc).
 - **The principle of least privilege** would give the system a chance to mitigate the attack
 - if malicious code cannot obtain root privileges, then there is a chance that adequately defined permissions may block all, or at least some, of the damaging operations.
 - Permissions can act like an immune system at the operating-system level.

- Another important principle,
- often seen as a derivative of the principle of least privilege, is **compartmentalization**.
 - **Compartmentalization** is the process of protecting each individual system component through the use of specific permissions and access restrictions.
 - if a component is subverted, another line of defense will “kick in” and keep the attacker from compromising the system any further.
 - Compartmentalization is implemented in many forms such as virtualization.

- The main component of modern operating systems is the kernel, which manages access to system resources and hardware.
- The kernel, is a trusted and privileged component and therefore must run with a higher level of privileges than user processes.
- This privilege separation requires hardware support
- A popular model of privilege separation is called **protection rings**.

- In this model, execution is defined as a set of concentric rings, with ring i providing a subset of the functionality of ring j for any $j < i$.
- The innermost ring, ring 0, thus provides the full set of privileges.

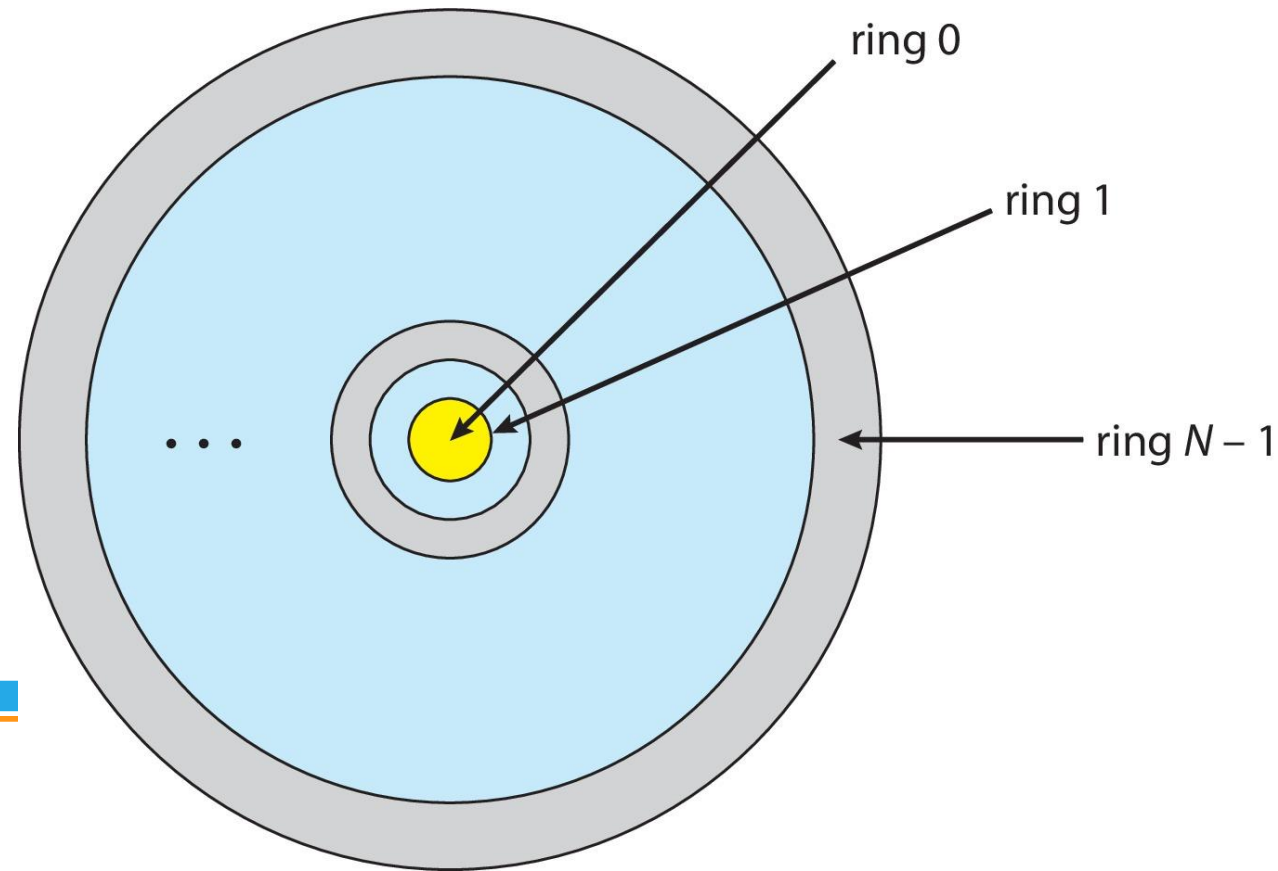


- When the system boots, it boots to the highest privilege level.
- Code at that level performs necessary initialization before dropping to a less privileged level.
- To return to a higher privilege level, code usually calls a special instruction, called gate.
- The syscall instruction (in Intel) is one example (Switch from user to kernel).

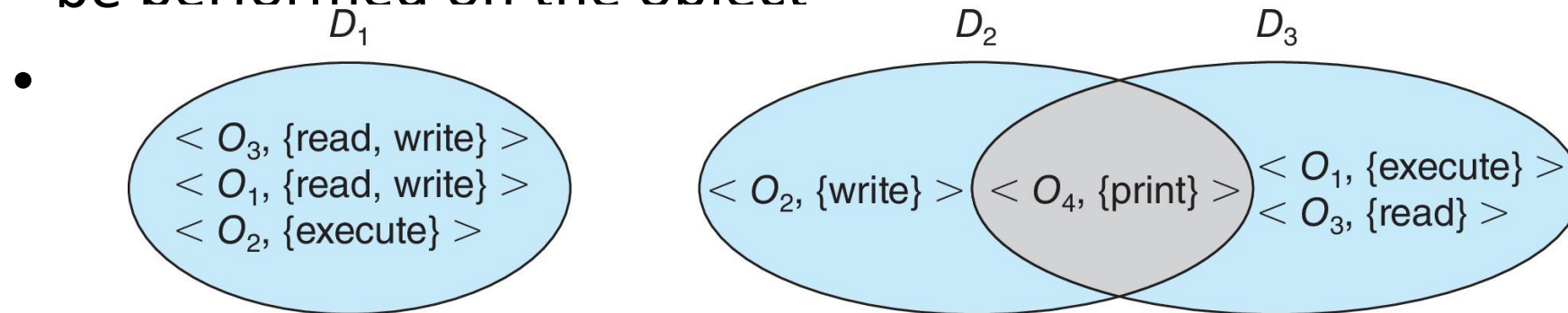
- Rings of protection separate functions into domains and order them hierarchically
- Computer can be treated as processes and objects
 - **Hardware objects** (such as devices) and **software objects** (such as files, programs, semaphores)
- Process should only have access to objects it currently requires to complete its task.

Protection Rings

- Let D_i and D_j be any two domain rings
- If $j < i \Rightarrow D_i \subseteq D_j$



- Domain = set of access-rights
- Access-right = $\langle \text{object-name}, \text{rights-set} \rangle$
- where *rights-set* is a subset of all valid operations that can be performed on the object



- The general model of protection can be viewed abstractly as a matrix, called an **access matrix**.
- Rows represent domains
- Columns represent objects
- **Access** (i, j) is the set of operations that a process executing in Domain _{i} can invoke on Object _{j}

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	



- The access matrix can implement policy decisions concerning protection.
- if a process in Domain D_i tries to do “op” on object O_j , then “op” must be in the access matrix
- User who creates object can define access column for that object
- To switch a process from one domain to another, (switch operation) is executing an on an object (the domain).

Use of Access Matrix Cont.....

Access Matrix of
Domains as Objects

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

- Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.
- The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right.
- The copy right allows the access right to be copied only within the column (that is, for the object) for which the right is defined.
- In the following example, process executing in domain D2 can copy the read operation into any entry associated with file F2.

Access Matrix with *Copy* Rights

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Owner Rights

Allowing controlled change in the contents of the access-matrix

Copy and Owner applicable to an object

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Modified Access Matrix of Figure 1

Control rights : A mechanism is also needed to change the entries in a row.

Control applicable to domain object

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Revocation of Access Rights

- In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users.
- Various options to remove the access right of a domain to an object
 - **Immediate vs. delayed**
 - Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
 - **Selective vs. general**
 - When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?

Revocation of Access Rights

- **Partial vs. total**
 - Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- **Temporary vs. permanent**
 - Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?
- **Access List** – Delete access rights from access list
 - **Simple** – search access list and remove entry
 - **Immediate, general or selective, total or partial, permanent or temporary**

Discretionary and Mandatory Access Access Control (DAC Vs MAC)

- **Access control** regulates which users, applications, and devices can add, edit, view and delete resources in an organization's environment to protect sensitive data from misuse, theft, abuse, and other threats.
- Mandatory Access Control (MAC) and Discretionary Access Control (DAC) are two of the popular access control models in use.

Discretionary and Mandatory Access Access Control (DAC Vs MAC)

- The key difference among these two approaches is in how they provide access to users.
 - With **MAC**, admins creates a set of levels where each user is linked to a specific access level where that user can access all the resources that are not greater than his access level.
 - In contrast, each resource in **DAC** has a list of users who can access it. DAC provides access by identity of the user and not by permission level.

Discretionary and Mandatory Access Access Control (DAC Vs MAC)

- As a brief example of MAC, consider a simple set of Levels, ordered according to level of privilege:
 - “unclassified,” “secret,” and “top secret.”
- A user with “secret” level, will have access to “unclassified” and “secret” files, but not to “top secret” files.
- No user or processes will be aware of the existence of “top secret” files, the operating system filters them out of all file operations
- Same with User processes, "unclassified" process would not be able to see or perform IPC requests to a “secret” (or “top secret”) process.

End of Module 8

