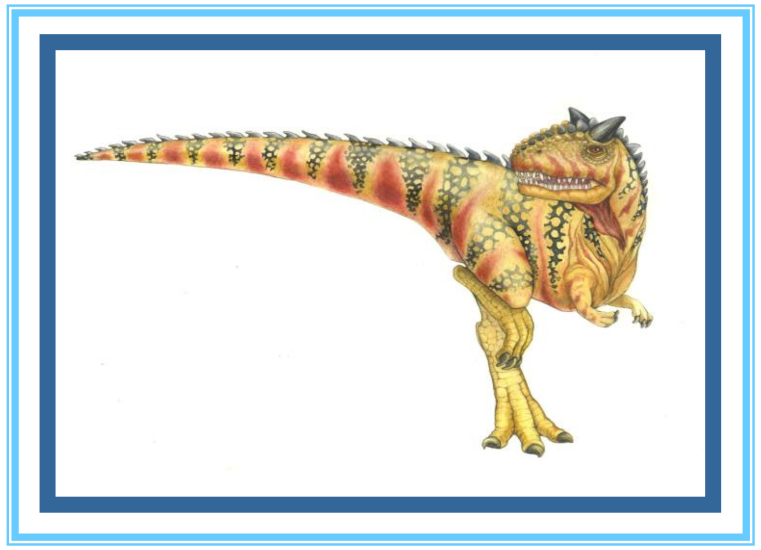
Chapter 14: Protection



Chapter 14: Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection

Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems

Goals of Protection

- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so
- The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use.
- In protection model, computer consists of a collection of objects, Hardware(such as the CPU, memory segments, printers, disks, and

  tape drives)) or software(files, programs, and semaphores)
- Each object has a unique name and can be accessed through a well-defined set of operations
- Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources.

.

Principles of Protection

- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given **just enough privileges** to perform their tasks
  - OS with this principle **limits damage** if entity has a bug, gets abused
  - provides system calls and services that allow applications to be written with **fine-grained access controls**.
  - Managing **users with the principle of least privilege** entails creating a **separate account** for each user, with just the privileges that the user needs
- “**Need to know”** a similar concept regarding access to data, where a process should be able to access only those resources that it currently requires to complete its task.

Domain of Protection

- A computer system is a collection of processes and objects.
- Object can be **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) **and software objects** (such as files, programs, and semaphores
- The operations that are possible may depend on the **object.**
- A process should be allowed to access only those resources for which it has authorization.
- A process should be able to access only those resources that it currently requires to complete its task.
- This second requirement, commonly referred to as the **need-to-know principle**, is useful in limiting the amount of damage a faulty process can cause in the system.

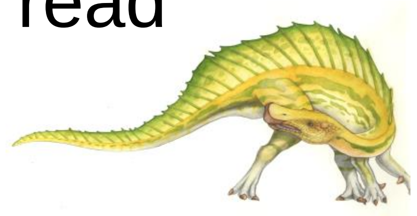
Domain Structure

- To implement domain of protection a process operates within a **protection domain,** which specifies the resources that the process may access
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- The ability to execute an operation on an object is an **access right**.
- A domain is a collection of **access rights**, each of which is an **ordered pair**

<object-name, *rights-set*>

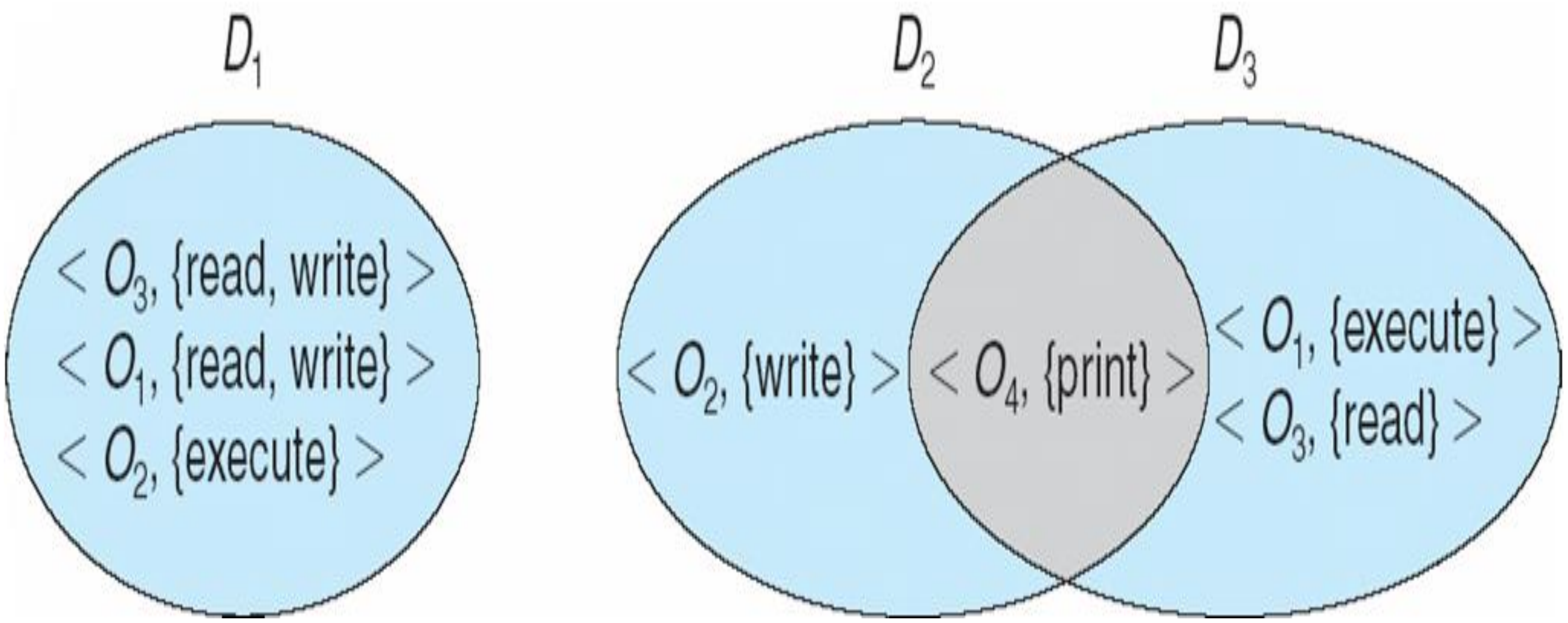
where *rights-set* is a subset of all valid operations that can be performed on the object

- For example, if domain D has the access right **<file F, {read,write}>,** then a process executing in domain D can both read and write file F.



Domain Structure

- Domains may share access rights.



Domain Structure

- The association between a process and a domain can be **static** (during life of system, during life of process) Or **dynamic** (changed by process as needed)
- In **dynamic,** we can allow **domain switching,** enabling the process to switch from one domain to another. Domain switching, **privilege escalation**

A domain can be realized in a variety of ways:

- Each **user** may be a domain. In this case, the set of objects that can be accessed depends on the **identity of the user**.
- Each **process** may be a domain. In this case, the set of objects that can be accessed depends on the **identity of the process**
- Each **procedure** may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure.

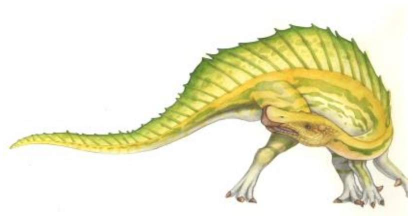
Domain Implementation (UNIX)

- In the UNIX operating system a domain is associated with **user,** and domain switching corresponds to **changing the user identification temporarily**.

- Domain switch accomplished via **file system** as follows
  - Each file has associated with it a domain bit (**setuid bit**)
  - This bit indicates whether a process executing this file should **temporarily inherit the owner's domain (user ID)**.
  - When **setuid = on(setuid=1)**, and a user executes the file then user-id is set to owner of the file. The process now runs with the **owner's privileges**, not the original user's.
  - When the **bit is off,** however, the userID does not change. The process runs **under the original user's domain** (UID) without any extra privileges.
  - When execution completes user-id is reset

Domain Implementation (UNIX)

- Domain switch accomplished via passwords
  - `su` command temporarily switches to another user’s domain when other **domain’s password provided**. Typically to gain administrative privileges or perform actions as a different user.
- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)



Domain Implementation (MULTICS)

MULTICS (**Multiplexed Information and Computing Service)** was a pioneering time-sharing operating system developed in the 1960s that laid the foundation for modern OS design, including influencing UNIX.

- In MULTICS, **domains refer to protection environments**—essentially, boundaries within which processes operate. This concept was central to MULTICS' **ring-based architecture**, which is one of its most influential contributions to operating system design.
- In the MULTICS system, the protection domains are organized hierarchically into a ring structure, each ring corresponds to a **single domain**
- **Ring 0** was the most privileged (kernel-level), while **Ring 7** was the least privileged (user-level).
- Each ring represented a domain of execution, and processes could only access resources permitted within their ring or lower (more privileged) rings.

Domain Implementation (MULTICS)

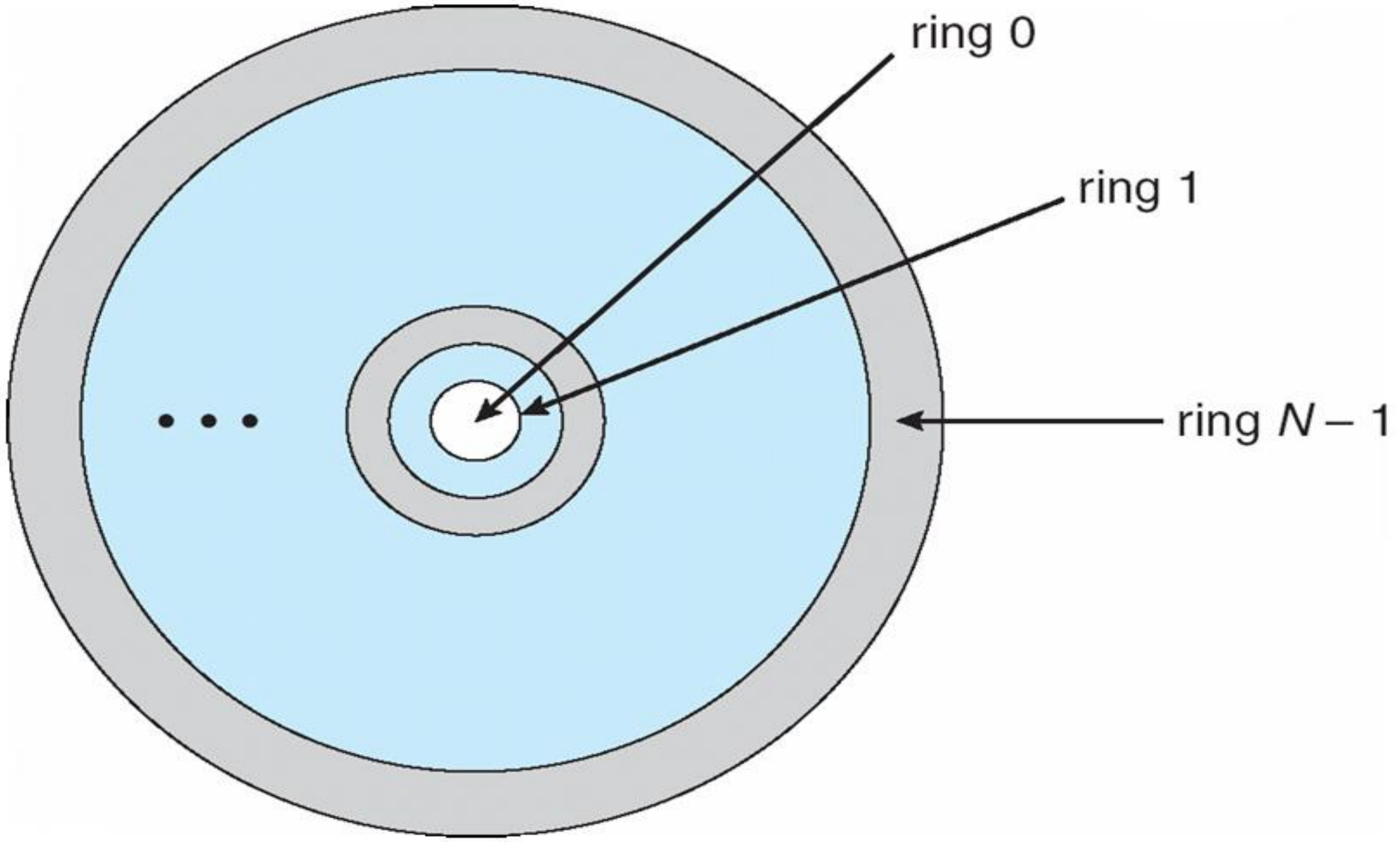
- Processes with **higher privilege levels (lower ring numbers),** while processes with **lower privilege levels (higher ring numbers)**
- Let Di and Dj be any two domain rings. **If j < i,** then **Di is a subset of Dj**.
- A process executing in domain Dj has more privileges than does a process executing in domain Di (j<i).
- A process executing in domain D0 has the most privileges.

Domain Implementation (MULTICS)

- Each process is associated with a **current-ring-number counter**, identifying the ring in which the process is executing currently.
- When a process is executing in ring i, it cannot access a segment associated with **ring j ($j < i$).** It can access a segment associated with **ring k ($k \geq i$).**
- In MULTICS segment is associated with one of the rings. A segment description includes an entry that identifies the ring number.
- Each procedure (code segment) is assigned a ring number, defining what level of access it has.

Domain Implementation (MULTICS)

- Let Di and Dj be any two domain rings
- If j < I $\Rightarrow$ Di $\subseteq$ Dj



Domain Switching (MULTICS)

- **Domain switching** in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring, in a controlled manner.
- In controlled domain switching, the ring field of the **segment descriptor must** include the following:

1) **Access bracket.** A pair of integers, b1 and b2, such that b1 ≤ b2.

which defines the range of rings that can **call** this segment *without switching rings*.

Example: Segment has bracket (1, 3).

Process in ring 2 calls it → allowed, no ring switch.

2) **Limit**. An **integer b3** such that b3 > b2.

If the call is from a **less privileged ring** (higher number than b_2), MULTICS still allows a domain switch — but only under strict control.

So: If $i > b_2$, the call is allowed only if $i \leq b_3$.

If $i > b_3$, the call is not allowed at all → trap to the OS (protection violation).

Ex: Segment has $(b_1,b_2,b_3) = (1,3,5)$

Process in ring 4 calls it → allowed (4 ≤ 5), but must go through a gate.

Process in ring 6 calls it → trap, denied.

Domain Switching (MULTICS)

3.

The **list of gates** defines specific **entry points** inside the segment that can be used for controlled calls from less privileged rings.

- When a call crosses a ring boundary (e.g., ring 4 → ring 1), the hardware and OS verify:
  - The entry point is in the segment's gate list.
  - The caller's ring number satisfies the rules above.
- If not, the system raises a trap (protection fault).

Access Matrix

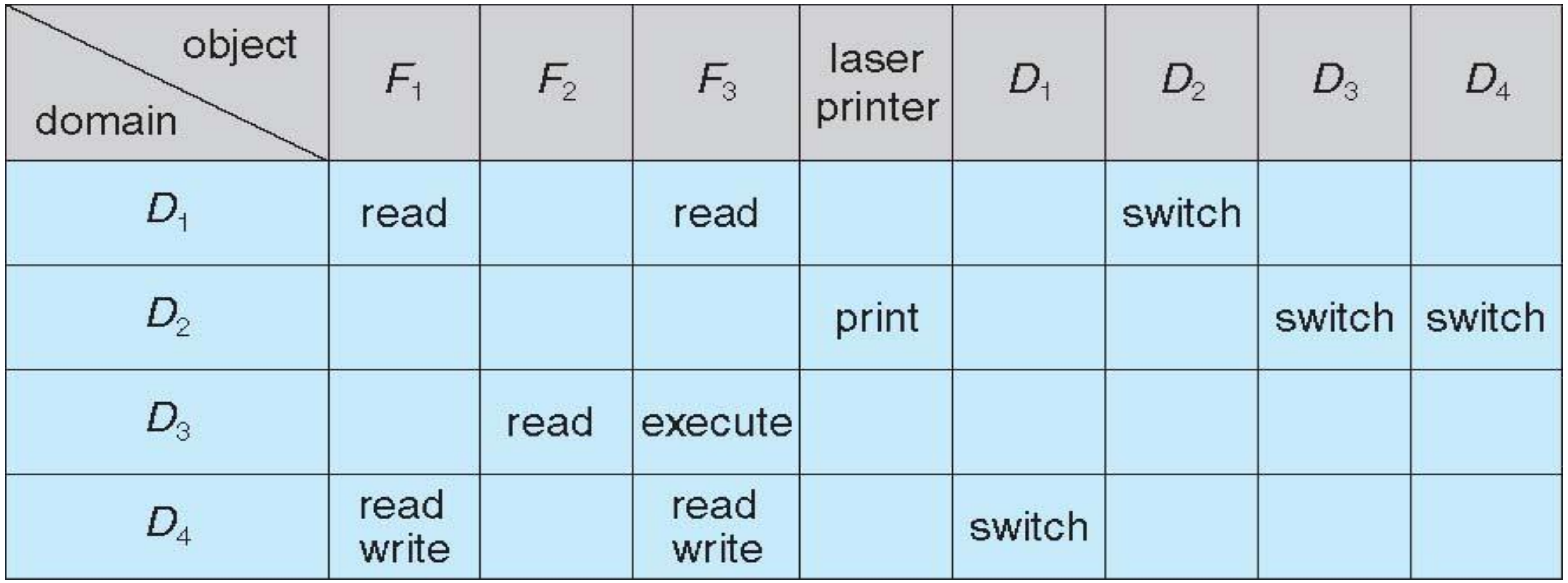
- The **Access Matrix Protection System** is a **model used in computer security** to describe and manage how different subjects (users, processes, etc.) can access various objects (files, devices, data, etc.) in a computer system.
- It provides a **formal and structured way** to represent **access rights** — who can do what to which resource.
- It provides a 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$_j$

Access Matrix

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	

Domains as Objects

- The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association between **processes and domains**.
- Processes should be able to switch from one domain to another.
- We can control **domain switching** by including **domains** among the **objects** of the access matrix.



domain \ object	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			

Use of Access Matrix

- If a process in Domain Di tries to do "op" on object Oj, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - owner of Oi
    - copy op from Oi to Oj (denoted by "*")
    - control – Di can modify Dj access rights
    - transfer – switch from domain Di to Dj

Access Matrix with *Copy* Rights

- Allowing controlled change in the access matrix content requires three additional operations:

1) Copy
2) Owner
3) control

- The access right can be copied from one **domain(row)** to another denoted by **an asterisk(*)** appended to the a**ccess right.**

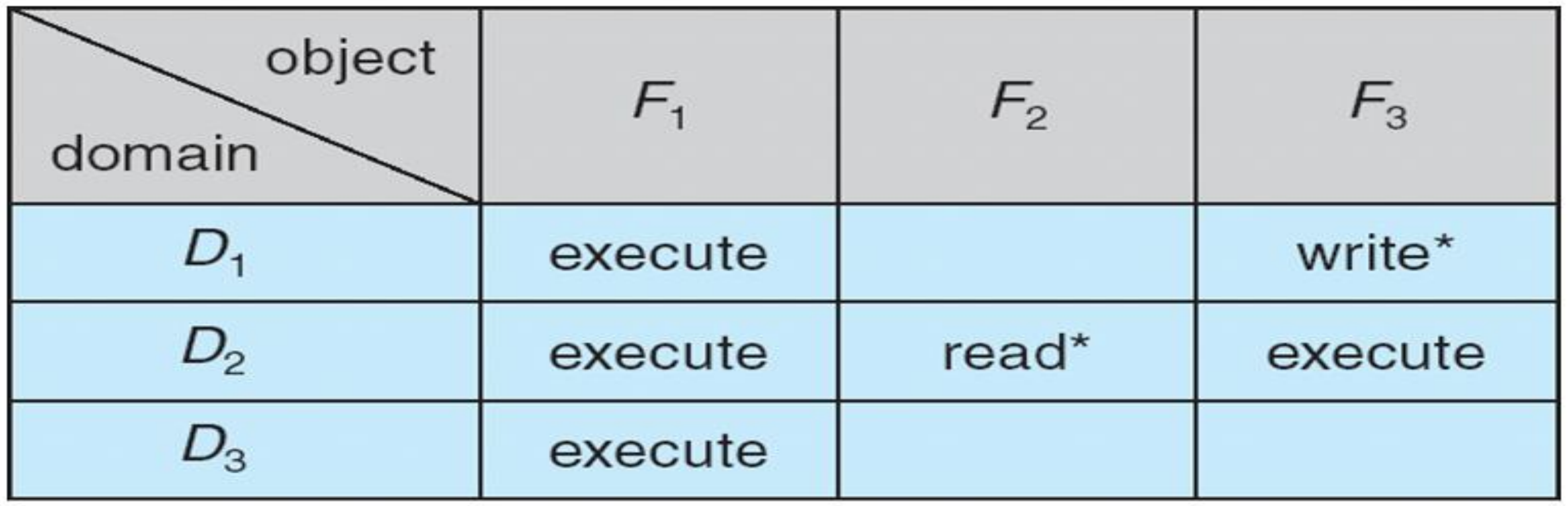
- The **copy right** allows the access right to be copied only within the **column (that is, for the object)** for which the right is defined.

- **Propagation of the copy right may be limited**. That is, when the right $R*$ is copied from access(i, j) to access(k, j), only the right R (not $R*$) is created. A process executing in domain *Dk* cannot further copy the right *R*.

Example:

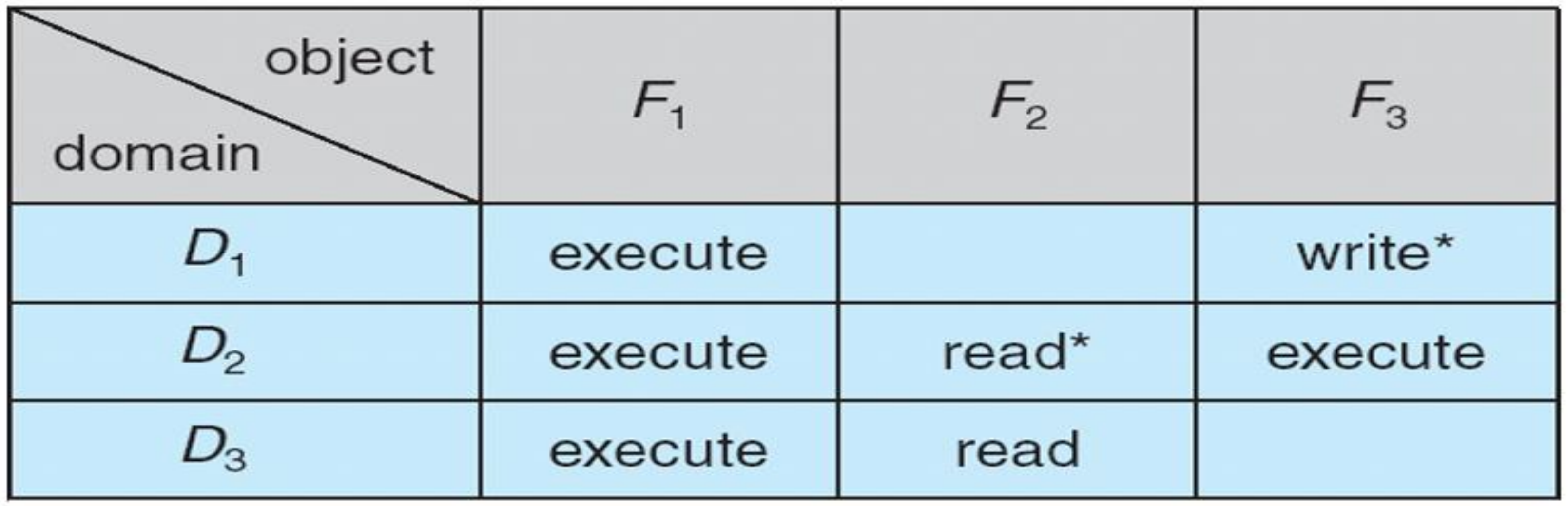
a process executing in domain *D*2 can copy the read operation into any entry associated with file *F*2.

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	

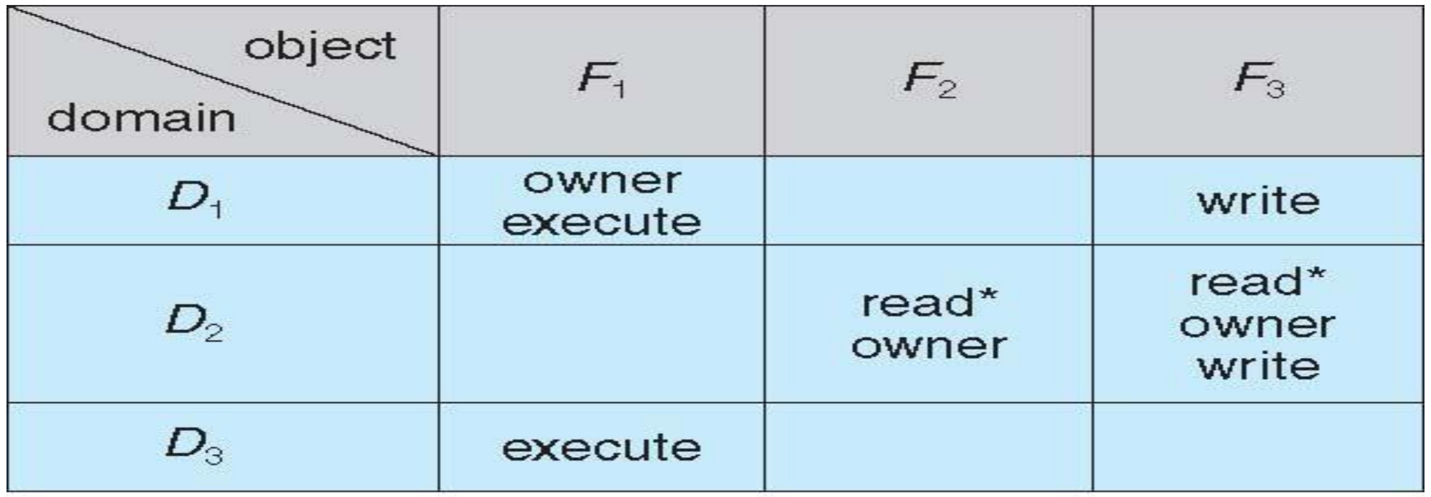
(b)

Access Matrix With *Owner* Rights

To allow addition of **new rights and removal of some rig**hts the owner right controls these operations

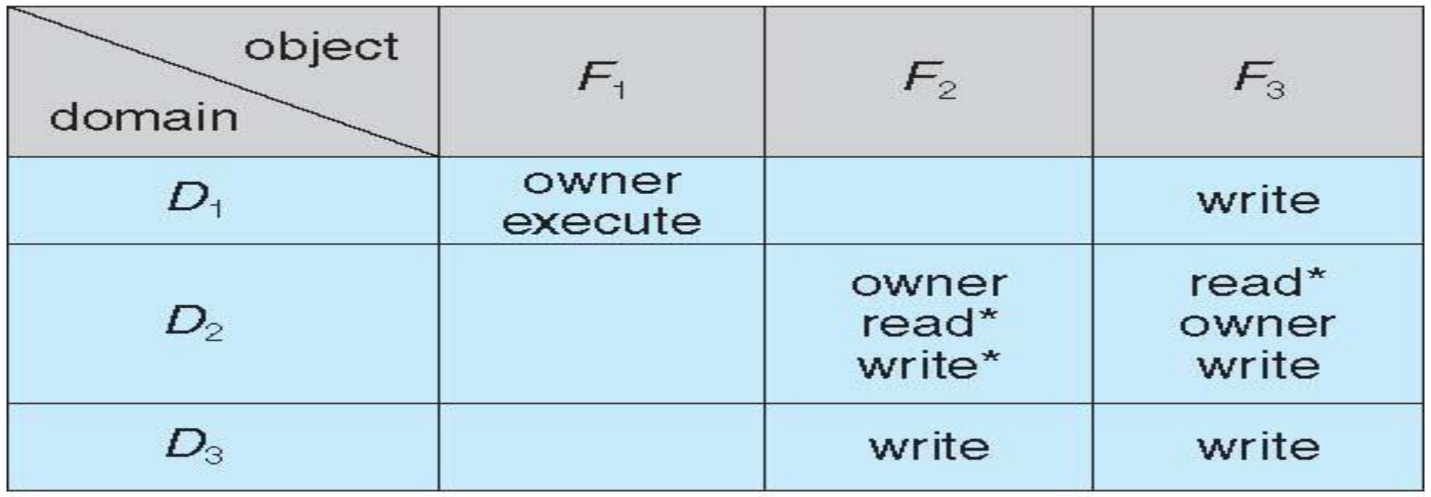
Example:

domain *D*1 is the owner of *F*1 and thus can add and delete any valid right in column *F*1.



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

(a)

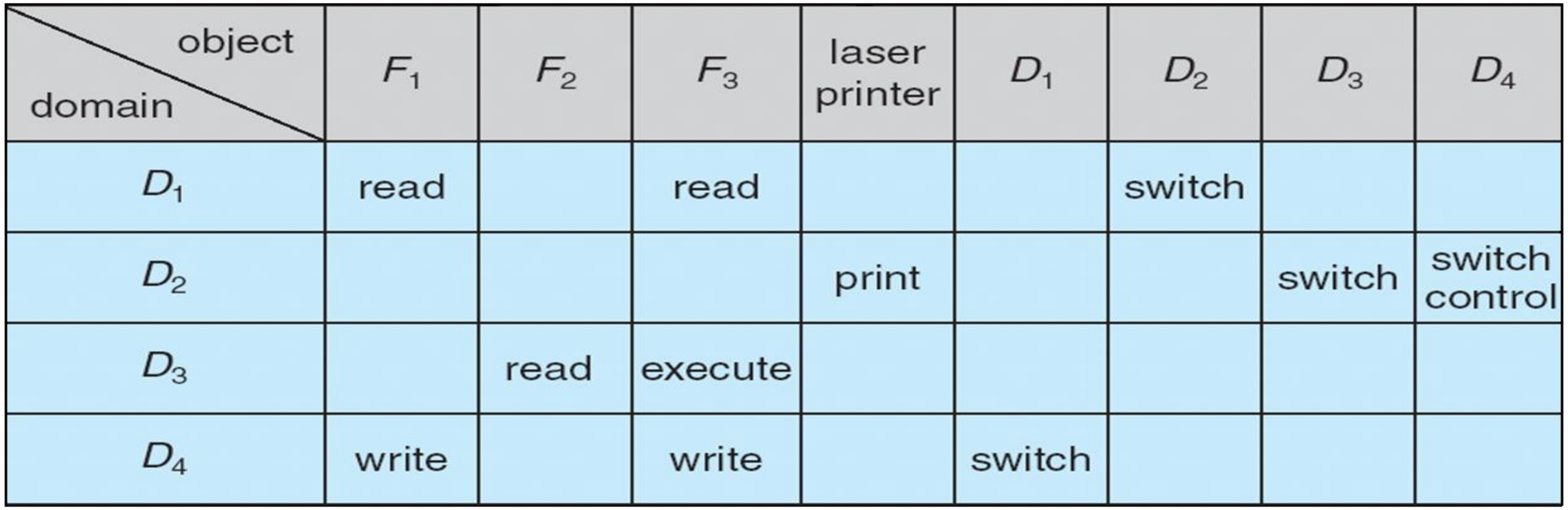


object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Modified Access Matrix: control right

- The **control right:** A mechanism is also needed to change the entries **in a row**.
- If access(*i*, *j*) includes the control right, then a process executing in domain *Di* can remove any access right from row *j*.
- **Example,** suppose that, we include the control right in **access(*D*2, *D*4)** Then, a process executing in domain *D*2 could modify domain *D*4



domain \ object	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			

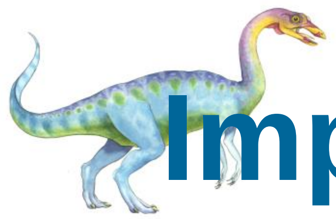
Implementation of Access Matrix

- Generally implemented as a sparse matrix
- Option 1 – **Global table**
  - Store ordered triples **<** ***domain, object, rights-set*** > in table
  - A requested operation M on object O_j within domain D_i -> search table for < D_i, O_j, R_k >
    - with $M \in R_k$
  - But table could be **large** -> won't fit in main memory
  - Difficult to **group objects** (consider an object that all domains can read), then that object will be in entries in every domain.

- Option 2 – **Access lists for objects**
  - Each column implemented as an **access list for one object**
  - Resulting per-object list consists of ordered pairs ***<domain, rights-set >*** defining all domains with set of access rights for the object
  - Easily extended to contain **default set(set of access rights)** -> If $M \in$ default set, also allow access

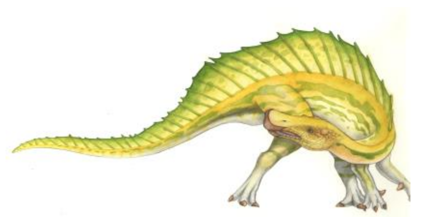
Implementation of Access Matrix (Cont.)

- Option 3 – **Capability list for domains**
  - Instead of object-based, list is **domain based**
  - Capability list for domain is **list of objects** together with **operations** allows on them
  - Object represented by its name or address, called a **capability**
  - Execute operation M on object Oj, process requests operation and specifies capability as parameter
    - Possession of capability means access is allowed
  - Capability list associated with domain but never directly accessible by domain
    - Rather, protected object, maintained by OS and accessed indirectly
    - This ensures that capabilities are not allowed to migrate into any **address space directly accessible by user process**



Implementation of Access Matrix (Cont.)

- Option 4 – **Lock-key**
  - Compromise between access lists and capability lists
  - Each object has list of unique **bit patterns, called locks**
  - Each domain as list of unique **bit patterns called keys**
  - Process in a domain can only access object if domain has key that matches one of the locks
  - The process is not allowed to modify its keys.



End of Chapter 14