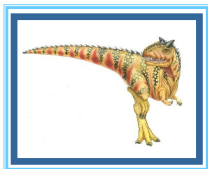


# Chapter 17: Protection



## Chapter 17: Protection

- Goals of Protection
- Principles of Protection
- Protection Rings
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix



## Objectives

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



## Goals of Protection

- In a protection model, computer system consists of a collection of **objects**, hardware or software
  - Hardware** objects: CPU, memory segments, printers, disks, and tape
  - Software** objects: files, programs, and semaphores
- Each object has a **unique name** and can be accessed through a **well-defined set of operations**
- Protection problem** is to ensure that each object is accessed correctly and only by those processes **allowed** to do so
- Mechanisms** are distinct from **policies**, in which **mechanisms determine how something will be done**, and **policies decide what will be done**.
  - The separation is important for **flexibility**, as policies are likely to change from **place** to **place** or from time to time.
  - The separation ensures that not every change in policy would require a change in the underlying mechanism.



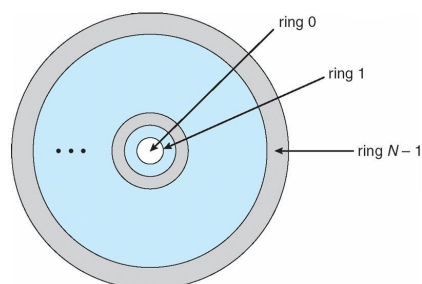
## Principles of Protection

- The guiding principle – **principle of least privilege** 最小权限原则
  - Programs, users and systems should be given just enough privileges to perform their tasks - mitigate the attack 缓解
  - In **file permissions**, this principle dictates that a user have **read access** but not write or execute access to a file. **The principle of least privilege** would require that the OS provides a mechanism to **only allow read access** but not write or execute access
- Properly set **permissions** (i.e., the access rights to an object) **can limit damage** if entity has a bug or gets abused 滥用
  - 好处



## Protection Rings

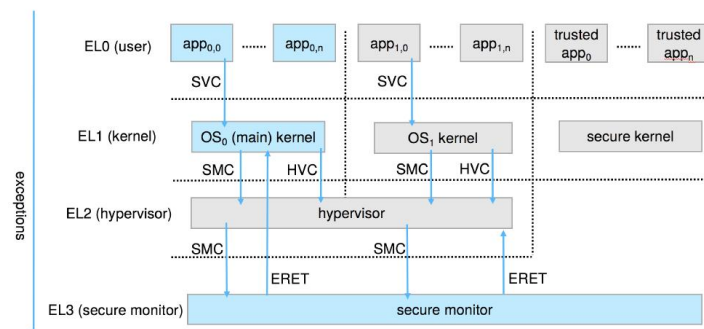
- User mode** and **kernel mode** – **privilege separation** 权限分离
- Hardware support required to support the notion of separate execution 概念
- Let  $D_i$  and  $D_j$  be any two domain rings
- If  $j < i \Rightarrow D_i \subseteq D_j$
- The innermost ring, ring 0, provides the **full set of privileges**



## Protection Rings (Cont.)

- Components ordered by amount of privilege and protected from each other
  - For example, the kernel is in one ring and user applications in another
  - This privilege separation requires hardware support
  - "Gates" used to transfer between rings, for example the `syscall` Intel instruction, also `traps` and `interrupts`
- Hypervisors** (Intel) is introduced (another ring) - virtual machine managers, which create and run virtual machines, and have more capabilities than the kernels of the guest operating systems
- ARM processors added **TrustZone** or **TZ** ring to protect crypto functions with access (more privileged than kernel)
  - This most privileged execution environment has exclusive access to hardware-backed cryptographic features, such as the NFC Secure Element and an on-chip cryptographic key, that make handling passwords and sensitive information more secure.

## ARM CPU Architecture



## Domain of Protection 保护域

- Protection rings separate functions into different domains and order them hierarchically 层次结构
- Domain** can be considered as a generalization of rings without a hierarchy 泛化
- A computer system can be treated as processes and objects
  - Hardware objects** (such as CPU, memory, disk) and **software objects** (such as files, programs, semaphores)
- Process for example should only have access to objects it currently requires to complete its task – the **need-to-know** principle (policy)
- Implementation can be via process operating in a **protection domain**
  - Protection domain specifies the set of resources a process may access
  - Each domain specifies set of objects and types of operations may be invoked on each object

## Domain of Protection (Cont.)

- 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>`
  - An 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. It cannot, however, perform any other operation on that object.
- Domains may share access rights
- Associations between processes and domains can be **static** if the set of resources available to the process is fixed throughout the process's lifetime, or can be **dynamic**
- If the association is dynamic, a mechanism is available to allow **domain switching**, enabling the process to switch from one domain to another during different stage of execution

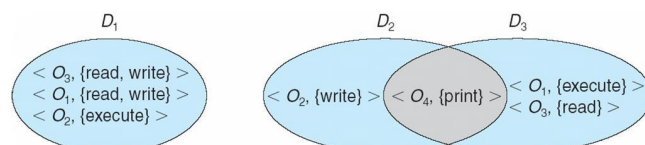
## Domain of Protection (Cont.)

Domain can be realized in a variety of ways: 实现 domain 的方法:

- Each **user** may be a domain - the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed
- Each **process** may be a domain - the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each **procedure** may be a domain - the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made

## Domain Structure 域结构

- Access-right** = `<object-name, rights-set>` where rights-set is a subset of all valid operations that can be performed on the object
- Domain** = set of access-rights
- The access right `<O4, {print}>` shared by domains D2 and D3, thus, a process executing in either of these two domains can print object O4.





## Access Matrix 访问矩阵.

- View protection as a matrix (**access matrix**)
- Rows** represent **domains**, and **columns** represent **objects**
- Access(i,j)** consists of a set of access rights - the set of operations that a process executing in Domain<sub>i</sub> can invoke on Object<sub>j</sub>

object domain	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	printer
D <sub>1</sub>	read		read	
D <sub>2</sub>				print
D <sub>3</sub>		read	execute	
D <sub>4</sub>	read write		read write	



## Use of Access Matrix

- The **access matrix** scheme provides the mechanism for specifying a variety of policies - **mechanism and policy separation**
- The **mechanism** consists of **implementing the access matrix** and **ensuring that the semantic properties hold**.
  - To ensure that a process executing in domain D<sub>i</sub> can access only those objects specified in row i.
- The **policy** decisions specify which rights should be included in the (i,j)th entry, and determine the domain in which each process executes
- If a process in Domain D<sub>i</sub> tries to do "op" on object O<sub>j</sub>, then "op" must be in the access matrix
- User who creates an object can define access column for that object**
  - When a user creates a new object O<sub>j</sub>, the column O<sub>j</sub> is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column j and other rights in other entries, as needed.



## Use of Access Matrix (Cont.)

This can be expanded to dynamic protection

- Operations to add, delete access rights
- Special access rights:
  - owner** of O<sub>i</sub> - can add and remove any right in any entry in column
  - copy** op from O<sub>i</sub> to O<sub>j</sub> (denoted by "\*\*") - **only** within the column (that is, for the object)
  - control** - D<sub>i</sub> can modify D<sub>j</sub> access rights - modify domain objects (a row)
  - transfer** - switch from domain D<sub>i</sub> to D<sub>j</sub>
- Copy** and **Owner** **applicable to an object** - change the entries in a **column**
- Control** **applicable to domain object** - change the entries in a **row**
- New objects and new domains can be created dynamically and included in the access-matrix model
- In a **dynamic** protection system, we may sometimes need to revoke access rights to objects shared by different users - **revocation** of access right **撤销**



## Access Matrix of Figure A with Domains as Objects

object domain	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	laser printer	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
D <sub>1</sub>	read		read			switch		
D <sub>2</sub>				print			switch	switch
D <sub>3</sub>		read	execute					
D <sub>4</sub>	read write		read write		switch			

"switch权限"通常指的是在计算机系统或网络中，用户或进程切换到不同权限级别或角色的能力。这种权限管理在多用户操作系统和网络环境中非常重要。以下是一些关键点：

- 权限切换**：用户或进程可以在不同的权限级别之间切换，例如从普通用户权限切换到管理员权限。这种切换通常需要特定的认证或授权。



## Access Matrix with Copy Rights

object domain	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
D <sub>1</sub>	execute		write*
D <sub>2</sub>	execute	read*	execute
D <sub>3</sub>	execute		

(a)

object domain	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
D <sub>1</sub>	execute		write*
D <sub>2</sub>	execute	read*	execute
D <sub>3</sub>	execute	read	

(b)



## Access Matrix With Owner Rights

所有者权限

object domain	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
D <sub>1</sub>	owner execute		write
D <sub>2</sub>		read* owner	read* owner write
D <sub>3</sub>	execute		

(a)

object domain	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>
D <sub>1</sub>	owner execute		write
D <sub>2</sub>		owner read* write*	read* owner write
D <sub>3</sub>		write	write

(b)





## Modified Access Matrix of Figure B

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			



## Implementation of Access Matrix

- In general, the access matrix is **sparse**; that is, **most of the entries will be empty**.
- Option 1 – Global Table**
  - Store ordered triples  $\langle \text{domain}, \text{object}, \text{rights-set} \rangle$  in table
  - A requested operation  $M$  on object  $O_j$  within domain  $D_i \rightarrow$  search table for  $\langle D_i, O_j, R_k \rangle$  with  $M \in R_k$
  - But the table could be large  $\rightarrow$  might not fit in main memory, requires additional I/O – **virtual memory** techniques are often used
  - Difficult to group objects - For example, if everyone can read a particular object, this object must have a separate entry in every domain.



## Implementation of Access Matrix (Cont.)

- Each column = **Access-control list** for one object  
Defines who can perform what operation
  - Domain 1 = Read, Write
  - Domain 2 = Read
  - Domain 3 = Read
- Each row = **Capability List** (like a key)  
For each domain, what operations allowed on what objects
  - Object F1 – Read
  - Object F4 – Read, Write, Execute
  - Object F5 – Read, Write, Delete, Copy



## Implementation of Access Matrix (Cont.)

- Option 2 – Access lists for objects**
  - Each column implemented as an access list for one object
  - Resulting per-object list consists of ordered pairs  $\langle \text{domain}, \text{rights-set} \rangle$  defining all domains with non-empty set of access rights for the object
  - Obviously, the empty entries can be discarded.
  - This can be easily extended to define **default** set of access rights  $\rightarrow$  If  $M \in \text{default set}$ , also allow access (for all domains)



## Implementation of Access Matrix (Cont.)

- Option 3 – Capability list for domains**
  - Instead of object-based, list is domain-based
  - A **capability list** for domain is a list of objects together with operations allowed on them
  - An object represented by its name or address, called a **capability**
  - To execute operation  $M$  on object  $O_j$ , a process requests operation  $M$ , specifying the capability (or pointer) for object  $O_j$  as a parameter
  - Possession of capability means access is allowed
- Capability list associated with a domain, but never directly accessible by a process executing in that domain
  - Rather, the capability list itself is a **protected object**, maintained by OS and accessed by users only indirectly
  - This avoids the possibility of capability list modification by users
  - If all capabilities are secure, the object they protect is also secure against unauthorized access



## 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 of the object
  - As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain.
  - Users are not allowed to examine or modify the list of keys (or locks) directly.

user 不能直接检查或修改列表





## Comparison of Implementations

Choosing a technique for implementing an access matrix involves various trade-offs.

- **Global table** is simple, but large, lack of grouping of objects or domains
- **Access lists** **correspond directly to the needs of users**
  - An access list on an object is specified when a user creates the object
  - Determining set of access rights for each domain is difficult - every access to the object must be checked, requiring a search of the access list.
- **Capability lists** useful for **localizing information for a given process**
  - But revocation capabilities can be inefficient
- **Lock-key** can be effective and flexible **depending on the length of the keys**
  - Keys can be **passed freely** from domain to domain, easy revocation
- Most systems use combination of **access lists** and **capabilities**



## End of Chapter 17

