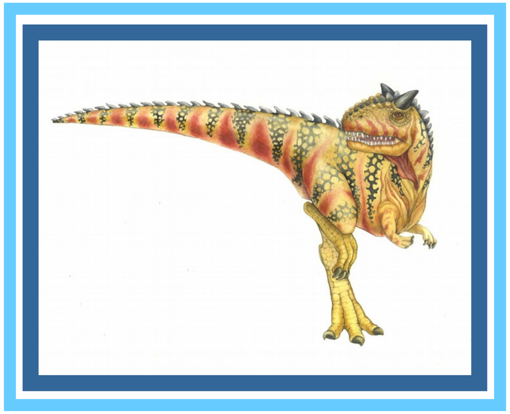
Chapter 16: Protection



Chapter 16: 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

- A 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

Principles of Protection

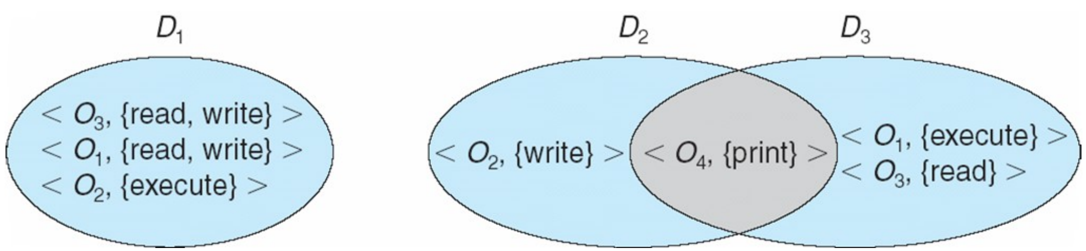
- A **privilege** is the right to execute a particular operation on a given object
- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough **privileges** to perform their tasks
    - Limits damage if entity has a bug, gets abused
  - Privileges can be one of:
    - Static (during life of system, during life of process)
    - Dynamic (changed by process as needed) – **domain switching**, **privilege escalation**
  - “Need to know” a similar concept regarding access to data

Principles of Protection (Cont.)

- Must consider "grain" aspect
  - Rough-grained privilege management easier, simpler, but least privilege now done in large chunks
    - For example, traditional Unix processes either have abilities of the
      - associated user, or
      - of the root
  - Fine-grained management more complex, more overhead, but more protective
    - ACL (Access Control List)
    - RBAC (Role Based Access control)
- Domain can be user, process, procedure

Domain Structure

- Access-right = <*object-name*, *rights-set*>
  - *rights-set* is a subset of all valid operations that can be performed on the object
- Domain = set of access-rights
- A process, at any point in time, is associated with one domain.
  - Can switch domain (in controlled way)
- Domains may overlap.

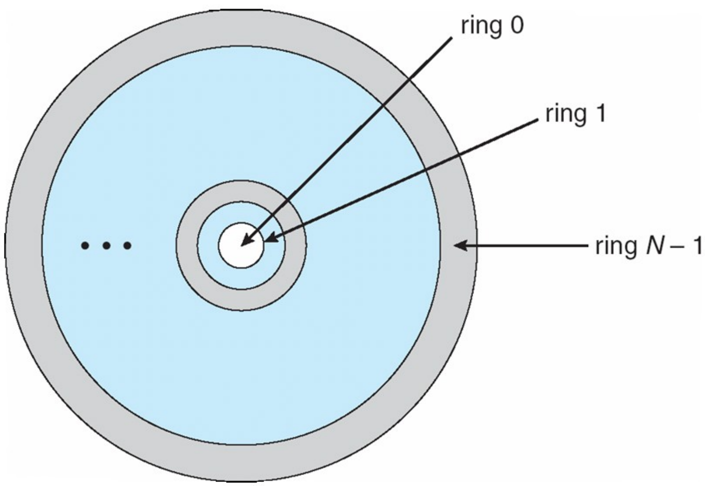


Domain Implementation -- UNIX

- Domain = user-id
- Domain switch accomplished via file system
  - Each file has associated with it a domain bit (setuid bit)
  - When file is executed and setuid = “on”, then user-id is set to the owner of the file being executed
  - When execution completes user-id is reset
- Domain switch accomplished via passwords
  - **`su`** command temporarily switches to another user’s domain when other domain’s password is provided
- Domain switching via commands
  - **`sudo`** command prefix executes specified command in another domain (if original domain has privilege or password given)

Domain Implementation -- MULTICS

- Domains are organized hierarchically into a ring structure.
- Each ring corresponds to a single domain
- Let D_i and D_j be any two domain rings
- Let D_i and D_j be any two domain rings
- If $j < I \Rightarrow D_i \subseteq D_j$
- Thus, D_0 has the most privileges.



Multics Benefits and Limits

- Ring / hierarchical structure provided more than the basic
  - kernel/user or
  - root/normal-user

  design
- Fairly complex ➔ more overhead
- But does not allow strict need-to-know
  - An object accessible in domain D_j but not in domain D_i, implies that ***j*** must be < ***i***
  - But then every object accessible in domain D_i also accessible in domain D_j

Access Matrix

- View protection as a matrix (**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	

Use of Access Matrix

- If a process in Domain D_i tries to do “op” on object O_j, then “op” must be in the access matrix
- A user who creates an object can define access column for that object
- The access matrix can implement policy decisions concerning protection.
  - Which rights should be in included in a specify entry in the matrix,
  - Which domain is a process first executing in (usually controlled by the OS.

Use of Access Matrix (Cont.)

- **Access matrix** design separates mechanism from policy
  - Mechanism
    - Operating system provides access-matrix + rules
    - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
  - Policy
    - User dictates policy
    - Who can access what object and in what mode
- But doesn't solve the general **confinement problem**
  - guaranteeing that no information initially held in an object can migrate outside of its execution environment
  - More later

Domain Switching

- Access matrix also provides a mechanism to control the switching from one domain to another.
- Columns can be either “objects” or “domains”.
- Have a special access right:
  - *switch* – to designate the privilege to transfer from one domain to another.
  - If an entry in the matrix contains “switch”, then a switch is allowed.

Access Matrix with Domains as Objects

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			

Access Matrix -- Dynamic Protection

- Operations to add and delete access rights
- Special access rights:
  - *owner of O_i*
  - *copy op from O_i to O_j (denoted by "*")*
  - *control – D_i can modify D_j access rights*
  - *transfer – switch from domain D_i to D_j*
- *Copy* and *Owner* applicable to an object
- *Control* applicable to domain object

Access Matrix With Copy (*) Rights

- A process in domain D_2 can copy the "read" access right to any entry in column F_2
  - Two variants. "transfer" and "limited copy"
- Figure (a) before copy transfer. Figure (b) after copy transfer

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

A process executing in domain D_1 can add and remove any right in any entry 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)

Access Matrix with *control* Rights

A process executing in domain D_2 can add and remove any right in any entry in row D_4.

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

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?
- **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?
- **Time dependent:** Active for a specific time period? play game only at night

Implementation of Access Matrix

- The access matrix is generally sparse
- Several options for implementing:
  - Global table
  - ACL – access control list
  - Capability list
  - Lock-key

Implementation – Global Table

- Store ordered triples

  <domain, object, rights-set>

  in a table
- A requested operation M on object O_j within domain D_i
  - Search the table for $\langle D_i, O_j, R_k \rangle$
  - with $M \in R_k$
- But table could be large ➔ won't fit in main memory and will require extra I/O
- Difficult to take advantage of special groupings of objects or domains.
  - For example, if everyone can read a particular object, this object must have a separate entry in every domain.

Implementation – ACL

- Each column in the access matrix can be implemented as an access list for one object
  - Obviously, the empty entries can be discarded
- The resulting list for each object consists of ordered pairs

  `<domain, rights-set>`

  which define all domains with a nonempty set of access rights for that object.
- Defines who can perform what operation (domain = user ID)

  Domain 1 = {Read, Write}
  Domain 2 = {Read}
  Domain 3 = {Read}
- Analogous to controlling access to a concert hall where there is a list of all people who are allowed to enter. Revocation is simple.

Implementation – Capability list

- Each row in the access matrix can be implemented as a **capability list**
  - List of objects together with operations allowed on them
- An object is represented by its name or address, is called a **capability**
- To execute operation M on object O_j:
  - The process executes the operation M, specifying the capability (or pointer) for object O_j, as a parameter.
  - Possession of capability means access is allowed
- Analogous to controlling access to a concert hall where each person posses a ticket. Revocation is difficult.

Implementation – Capability list (cont.)

- Capability list associated with domain but never directly accessible to a process executing in that domain
  - The capability list is itself a protected object, maintained by the OS and accessed indirectly
  - Like a “secure pointer”
  - Idea can be extended up to applications
- For each domain, what operations allowed on what objects

  Object F1 = {Read}

  Object F4 = {Read, Write, Execute}

  Object F5 = {Read, Write, Delete, Copy}

Implementation – Lock-key

- Compromise between access list and capability list
- 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

Comparison of Implementations

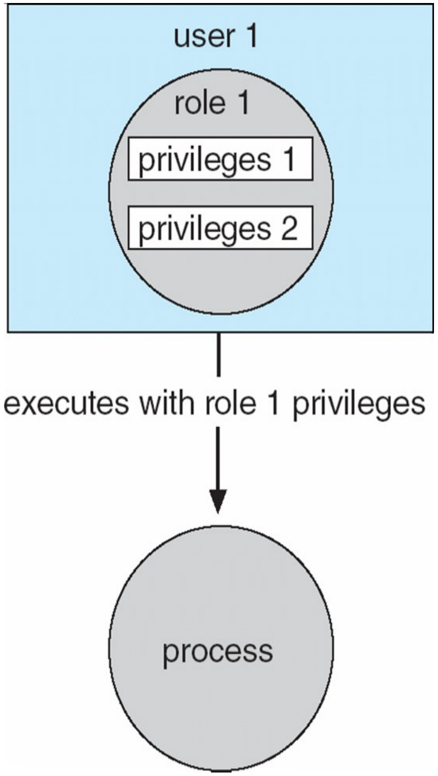
- Global table is simple, but can be large
- Access list correspond to needs of users
  - Every access to an object must be checked
    - Many objects and access rights ➔ slow
- Capability list is useful for localizing information for a given process
  - But revocation of capabilities can be inefficient
- Lock-key effective and flexible, keys can be passed freely from domain to domain, easy revocation

Comparison of Implementations (Cont.)

- Most systems use combination of access lists and capabilities
  - First access to an object → access list searched
    - If allowed, capability created and attached to process
      - Additional accesses need not be checked
    - After last access, capability destroyed
    - Consider a file system with ACLs per file

Access Control

- Protection can be applied to non-file resources
- Oracle Solaris 10 provides **role-based access control** (**RBAC**) to implement least privilege
  - ***Privilege*** is right to execute system call or use an option within a system call
  - Can be assigned to processes
  - Users assigned ***roles*** granting access to privileges and programs
    - Enable role via password to gain its privileges
  - Similar to access matrix



Revocation of Access Rights

- **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**
- **Capability List** – Scheme required to locate capability in the system before capability can be revoked
  - **Reacquisition** – periodic delete, with require and denial if revoked
  - **Back-pointers** – set of pointers from each object to all capabilities of that object (Multics)

Capability List Revocation (Cont.)

- **Indirection** – capability points to global table entry which points to object – delete entry from global table, not selective (CAL)
- **Keys** – unique bits associated with capability, generated when capability created
  - Master key associated with object, key matches master key for access
  - Revocation – create new master key
  - Policy decision of who can create and modify keys – object owner or others?

Capability-based Systems

- **Hydra** -- A capability-based microkernel designed to support a wide range of possible operating systems to run on top of it. Hydra was created at the Carnegie Mellon university in the 1970s
- **CAP** – A simpler and superficially less powerful than that of Hydra. However, closer examination shows that it, too, can be used to provide secure protection of user-defined objects.CAP was developed at the University of Cambridge University of Cambridge Computer Laboratory in the 1970s.

Hydra

- Fixed set of access rights known to and interpreted by the system
  - For example, read, write, or execute each memory segment
  - User can declare other **auxiliary rights** and register those with protection system
  - Accessing process must hold capability and know the name of operation
  - **Rights amplification** allowed by trustworthy procedures for a specific type (discussed later)

Hydra (Cont.)

- Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights
- Operations on objects defined procedurally – procedures are objects accessed indirectly by capabilities
- Solves the *problem of mutually suspicious subsystems*
- Includes library of prewritten security routines

Cambridge CAP System

- Simpler then Hydra but powerful
- **Data capability** - provides standard read, write, execute of individual storage segments associated with object – implemented in microcode
- **Software capability** -interpretation left to the subsystem, through its protected procedures
  - Only has access to its own subsystem
  - Programmers must learn principles and techniques of protection

Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system

Protection in Java 2

- Protection is handled by the Java Virtual Machine (JVM)
- A **class** is assigned a protection domain when it is loaded by the JVM
- The protection domain indicates what operations the class can (and cannot) perform
- If a library **method** is invoked that performs a privileged operation, the stack is **inspected** to ensure the operation can be performed by the library
- Generally, Java's load-time and run-time checks enforce **type safety**
- Classes effectively **encapsulate** and protect data and methods from other classes

Stack Inspection

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	. . .
	get(url);
	open(addr);
	. . .
		get(URL u):
		. . .
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		. . .
			open(Addr a):
			. . .
			checkPermission (a, connect);
			connect (a);
			. . .

End of Chapter 16