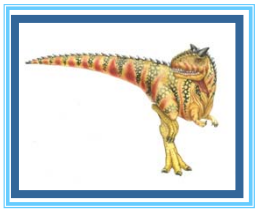
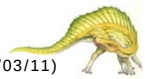
Chapter 14: Protection



(slides improved by R. Doemer, 03/03/11)

Chapter 14: Protection

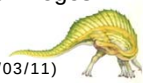
- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection



(slide modified by R. Doemer, 03/03/11)

Goals and Principles of Protection

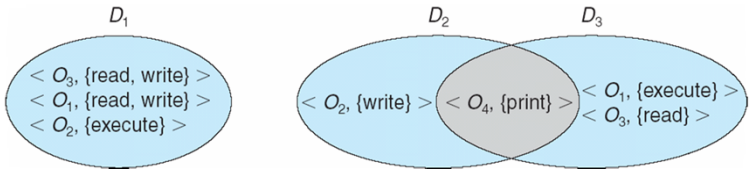
- **Operating system** consists of a **collection of objects**, hardware or software
- Each object has a unique name and can be accessed through a **well-defined set of operations**.
- **Protection problem**
  - ensure that each object is accessed *correctly*, and
  - only by those processes that are *allowed* to do so.
    - **Protection** is addresses only an *internal* problem! (in contrast to **Security**, see next chapter!)
- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough privileges to perform their tasks

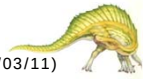


(slide modified by R. Doemer, 03/03/11)

Domain Structure

- **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





(slide modified by R. Doemer, 03/03/11)

Domain Implementation in UNIX

- System consists of 2 types of domains:
  - Each **User**
  - **Supervisor** (aka. super-user, root)
- Specifically in UNIX
  - Object = file
  - 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 owner of the file being executed.
    - When execution completes user-id is reset.
  - Example:
    - Homework submission script on Unix file system

(slide modified by R. Doemer, 03/03/11)

Access Matrix

- View protection as a matrix: **Access Matrix**
  - Rows represent **domains**
  - Columns represent **objects**
- **Access(i, j)** is the set of operations

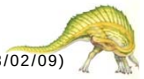
  that a process executing in **Domain$_i$** can invoke on **Object$_j$**

(slide modified by R. Doemer, 03/03/11)

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	

Simple access matrix example

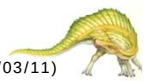


(slide fixed by R. Doemer, 03/02/09)

Use of Access Matrix

- **Protection**:
  - If a process in domain D_i tries to perform "op" on object O_j,

    then "op" must be in the access matrix.
- Can be expanded to **dynamic protection**.
  - Operations to add, delete access rights.
  - Special access rights:
    - ***owner*** *of* O_i
    - ***copy*** *op from* O_i *to* O_j
    - *control –* D_i *can modify* D_j *access rights*
    - *transfer –* ***switch*** *from domain* D_i *to* D_j



(slide modified by R. Doemer, 03/03/11)

Access Matrix with *Switch* Rights

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			

Example of extended access matrix
(***switch*** between domains)

(slide modified by R. Doemer, 03/03/11)

Access Matrix with *Copy* Rights

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

(a) Before ***copy***

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

(b) After ***copy***

(slide fixed by R. Doemer, 03/02/09)

Access Matrix With *Owner* Rights

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

(a) Before ***owner*** change

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

(b) After ***owner*** change

(slide modified by R. Doemer, 03/03/11)

Use of Access Matrix

- Access matrix design separates **mechanism** from **policy**.
  - **Mechanism**
    - Operating system provides access-matrix + rules.
    - It 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.

(slide modified by R. Doemer, 03/03/11)

Implementation of Access Matrix

- By **column** = **Access-control list** for one object
  Defines who can perform what operation.

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

  ⋮

- By **row** = **Capability List** (like a key)
  Fore each domain, what operations are allowed on what objects.

  Object 1 – Read
  Object 4 – Read, Write, Execute
  Object 5 – Read, Write, Delete, Copy

Revocation of Access Rights

- **Access List** –
  Delete access rights from access list.
  - Simple
  - Immediate

- **Capability List** –
  Scheme required to locate capability in the system before it can be revoked.
  - Reacquisition
  - Back-pointers
  - Indirection
  - Keys

(slide fixed by R. Doemer, 03/02/09)

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.

(slide modified by R. Doemer, 03/03/11)

Protection in Java

- Protection is handled by the **Java Virtual Machine (JVM)**
- Each **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.
  - **doPrivileged block** annotates **stack frame** for tree of privileged calls
  - If a class method is invoked that performs a **privileged operation**, the **stack is inspected** to ensure the operation can be performed by the class.

(slide modified by R. Doemer, 03/03/11)

Stack Inspection in Java

- **Example** of protected method call in Java:
  - **`gui`** method of untrusted applet calls **`get`** and **`open`**
    - **`get`** succeeds because **`checkPermission`** finds **`doPrivileged`** stack frame
    - **`open`** fails because no **`doPrivileged`** stack frame is found

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);
			...

- **Note**: Stack must be protected from any manipulation!
  - Java uses *safe* pointers!

(slide modified by R. Doemer, 03/03/11)

End of Chapter 14