Announcements

- Assignment 3 due.
- Invite friends, co-workers to your presentations.
- Course evaluations on Friday.

Chapter 18: Protection

- Protection Goal.
- Protection Domain.
- Access Matrix.
- Java Protection.

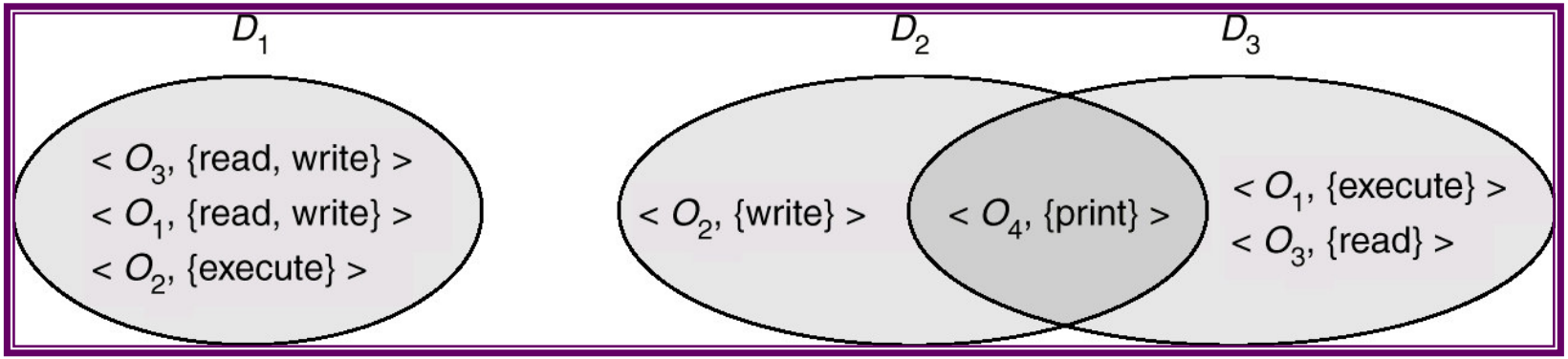
Protection Goal

- Operating system consists of a collection of objects, hardware and software.
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection goal: ensure that each object is accessed only by those processes that are allowed to do so, and only in a prescribed manner.
- Protection not concerned with *verifying identity* of
  - Objects (e.g. whether IP address is real or spoofed) or
  - Processes (e.g. whether a process executing on behalf of Bob was started by Bob, or Sue who broke into Bob's office).

  That is *security* (next chapter).

Protection Domain

- An access right is a pair consisting of
  - An object, e.g. a file.
  - A subset of all valid operations that can be performed on the object, e.g. read, write, execute.
- A domain is a set of access rights.
- A process executes “within” a domain, which limits what the process can do.



Access Matrix

- View protection as a matrix (*access matrix*):
  - Rows are domains D_i.
  - Columns are objects (files F_j in the example, and printer).
  - *Access(i, j)* is the set of operations that a process executing in D_i can invoke on F_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	

Process Domain in UNIX

- Each user associated with a domain:
  - When user logs in, his/her domain is identified.
- Each user group associated with a domain:
  - A user can belong to many groups but only one is current at a time.
- All users also belong to the world domain (*others*).
- Process domain: process usually started by user in the context of an interactive user *session*. Process domain is union of
  - World domain (operations anybody can do),
  - Session user's current group domain (users of the group can do),
  - Session user domain (only the user can do).
- Many refinements, such as:
  - For user to execute privileged operation (e.g. modify the printer's spool queue directory) in a safe manner (e.g. when printing a job), some processes must be executed under supervisor (root) domain.
  - Each program file has a user owner (possibly different than user executing program) and a permission bit called setuid.
  - If setuid==0, then process runs as above. If setuid==1, then process domain is that of user owner, not executing user.

Access Matrix: Add Domains

- Domains can be treated as objects:
  - We can now control whether a process can switch domains (e.g. setuid): `switch` right.
  - A process that runs in a domain can change the full matrix row for any other domain if it has the `control` right.

Process runs in D_2 and changes 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			

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: Add Refined Control

Copy right (*): right to copy a specific right from one domain onto another.

Owner right: right to modify all rights of a specific object.

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

(a)

Process runs in D_2 and copies read right for F_2 into D_3.

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

(b)

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

(a)

Processes run in D_1 , D_2 and change rights for F_1, F_2, F_3.

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

(b)

Access Matrix Usage

- Access matrix design separates *mechanism* from *policy*.
- Mechanism:
  - OS stores matrix.
  - OS defines operations to change matrix.
  - OS protects matrix storage and enforces changes only via prescribed operations.
  - OS forces processes to obey matrix.
- Policy:
  - User(s) decide what matrix contains, i.e.
    - what system objects are protected,
    - what domains exist, and
    - what rights are assigned to each combination.

Access Matrix Implementation

- Store each column with each object. This is the *Access-control list* (ACL) for that object. Example:
  Domain 1: Read, Write.
  Domain 2: Read.
- Store each row with each domain. This is the *Capability List* of the domain. Example:
  Object 1: Read.
  Object 4: Read, Write, Execute.
- Combination:
  - Explicit:
    - Process *P* starts with an empty capability list.
    - When *P* attempts to use object *O*, we consult ACL.
    - Add capability entry in *P*'s list: (*O*,<Read,Write>).
    - Future access to *O* consults list only. List is cache.
  - Implicit:
    - File `open()` returns file handle to process after ACL check.
    - All file operations use this handle.

Access Matrix: Revoking Rights

- *Access List*: Delete access rights from access list. Simple and immediate.
- *Capability List* (alone, or in part): scheme required to locate capability throughout system before capability can be revoked.
  - Reacquisition: process must reacquire capability at regular intervals (e.g. re-check ACL in combination system).
  - Back-pointers from objects to their capabilities (e.g. from files to processes that are using them; invalidate file handle if file ACL changed to prevent further access).
  - And others.

Java Protection

- Protection is handled by the Java Virtual Machine (JVM).
- When JVM loads a class, it assigns it a *protection domain*; all instances operate within this domain. Each domain is a set of permissions. Example:

```
grant codeBase "file:/${java.home}/lib/" {
  permission java.io.FilePermission "/home/-", "read"; }
```

  All JDK code can read files from directory `/home` and its subdirectories.
- The above snippet is from the Java *security policy* file.
- Run-time checks: a thread can execute an operation iff
  - It is executing a method of a class which has the necessary permissions, *and*
  - The same applies for the method's caller, and its caller, all the way to the bottom of the stack (*stack inspection*).
- So if unsafe applet calls system code to open file, `open()` will fail because applet doesn't have such permission.

Java Protection: `doPrivileged`

- What if applet wants to show dialog box? Internally, JDK must load font file but applet cannot open files. Solution is

```
void showDialogBox() {
  ...
  AccessController.doPrivileged
   (new PrivilegedAction() {
     public Object run() {
      /* load font */ } } )
  ... }
```

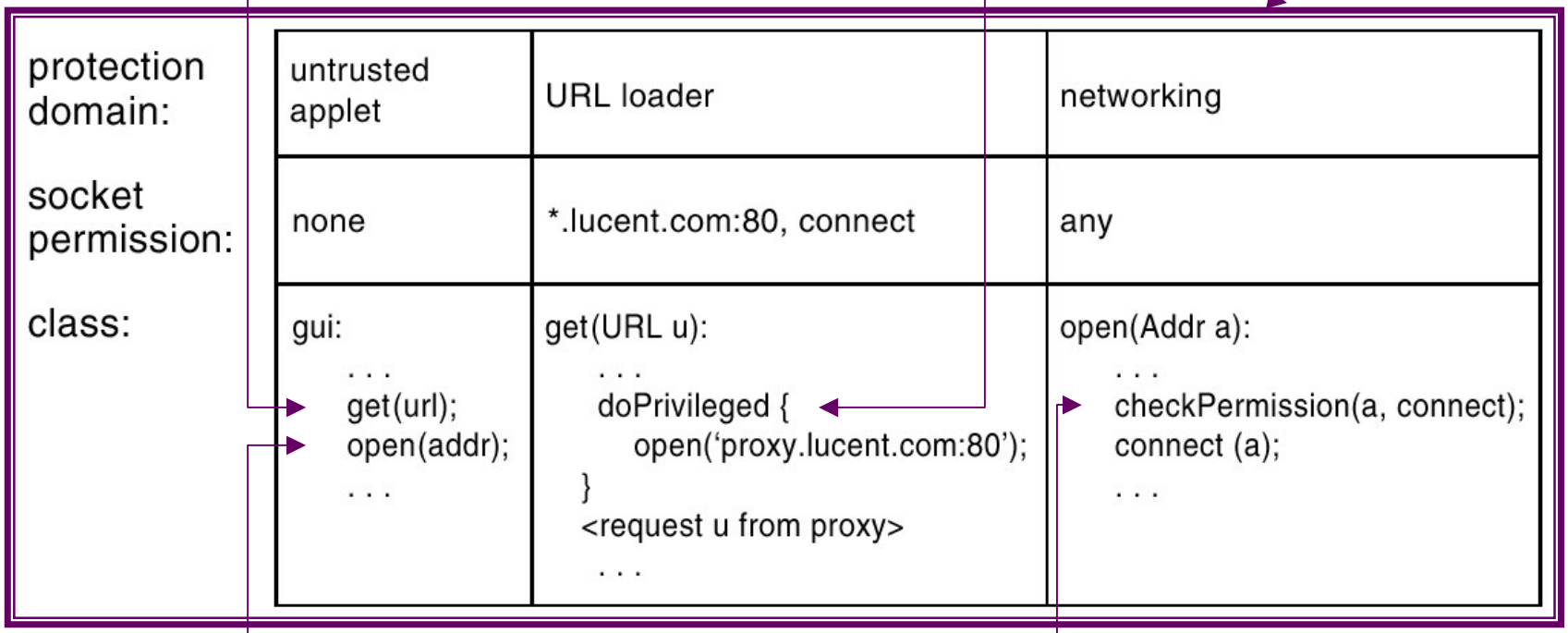
- Change to run-time checks: we stop stack inspection when we encounter a privileged block.
- Invoking stack inspection (inside `open()`):

```
AccessController.checkPermission
   (new FilePermission(name,access));
```

  Either throws (no permission) or returns silently.
- Fully extensible: new permissions, explicit checks, etc.

Java Protection Example

Will succeed because `doPrivileged` ends stack inspection, and until `doPrivileged` all callers have permission.



Will fail:because `checkPermission` will realize applet doesn't have required permission.