**Chapter 14: System Protection**

**14.1 Goals of Protection**

1. To prevent malicious misuse of the system by users or programs.
2. To ensure that each shared resource is used only in accordance with system policies, which may be set either by system designers or by system administrators.
3. To ensure that errant programs cause the minimal amount of damage possible.

Protection systems only provide mechanisms for enforcing policies and ensuring reliable systems. It’s up to administrators and users to implement those mechanisms effectively.

* *Mechanism* distinct of *policies*. Mechanism determine how something will be done. Policies decide what will be done. *Separation* of both is important for flexibility. Policies change, but mechanisms are general.

**14.2 Principles of protection.**

The **principle of least privilege**: programs, users, and systems be given just enough privileges to perform their tasks.  
This ensures that failures do the least amount of harm and allow the least of harm to be done.

* If a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of ‘network’ or ‘backup’ or some other pseudo group, rather than SUID with root ownership.

This limits the amount of damage that can occur if something goes wrong.  
Typically each user is given their own account, and has only enough privilege to modify their own files.   
The root account should not be used for normal day to day activities - the sysadmin should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges.

**14.3 Domain of protection**

A computer can be viewed as a collection of processes and objects.

* **Hardware objects**: such as CPU, memory segments, printers, disks and tape drives.
* **Software objects**: such as files, programs, and semaphores.

Each object has a unique name and can be accessed only through defined operations. Objects are essentially abstract data types. Operations:

* CPU: only execute
* Memory segments: read/write
* CD-ROM: write-only
* Tape drives: read/write and rewound
* Data files: created, opened, read, written, closed, and deleted
* Program files: read, written, executed, and deleted.

The **need to know principle**: a process should only have access to those objects it needs to accomplish its tasks, and furthermore only in the modes (authorizations) for which it needs access and only during the time frame when it needs access.

The modes available for a particular object may depend upon its type.

**14.3.1 Domain structure**

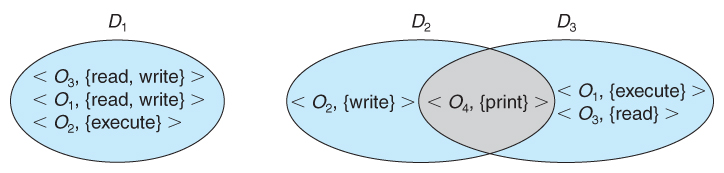
A **protection domain** specifies the resources that a process may access.

Each domain defines a set of objects and the types of operations that may be invoked on each object.

An **access right** is the ability to execute an operation on an object.

A domain is defined as a set of <object-name, rights-set> pairs (ex.: if domain D has access right <File F, {read, write, delete}, then process executing in D can read, write and delete, but not for example delete F), as shown below.

* Some domains may be disjoint while others overlap.

****

**Figure 14.1 - System with three protection domains.**

The association between a process and a domain may be static or dynamic.

* If the association is **static**, then the need-to-know principle requires a way of changing the contents of the domain dynamically. (domain must always reflect the minimum necessary access rights)
* If the association is **dynamic,** then there needs to be a mechanism for **domain switching**.
  + Also need to allow content of domain to be changed, if not possible, create new domain with changed content and switch to new domain.

Domains may be realized in different fashions - as users, or as processes, or as procedures:

* *User*: set of objects than be accessed depends on identity of user. Domain switching occurs when user is switched (user login/logout)
* *Process*: set of objects that can be accessed depends on identity of process. Domain switching occurs when one process sends a message to another process and then waits for a response.
* *Procedure*: set of objects that can be accessed correspond to the local variables defined within that procedure. Domain switching occurs when a procedure call is made.

**14.3.2 An example: UNIX (book edit needed)**

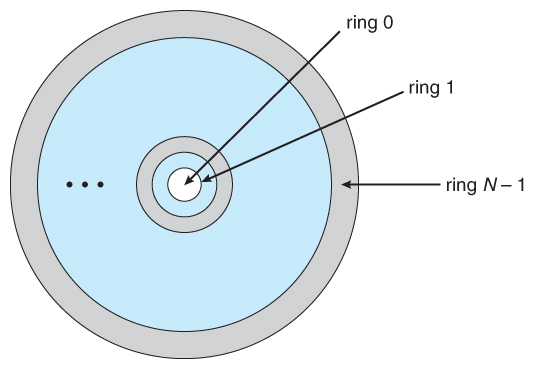
Unix associates domains with users.

Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.

An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.

Yet another alternative is to not allow the changing of ID at all. Instead, a special privileged **daemons process** is launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

**14.3.3 An example: MULTICS (book edit needed)**

The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:

**Figure 14.2 - MULICS ring structure**

* Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
* Each file is a memory segment, and each segment description includes an entry that identifies the ring number, as well as access bits to control read, write, and execution privileges.
* A *current-ring-number counter* is associated with each process, identifying the ring in which the process is executing currently.

A process operating in one ring can only access segments associated with higher (farther out) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.

Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:

* An **access bracket**. Pair of integers b1 and 2, where b1 <= b2.
* A **limit.** b3 where b3 > b2.
* A **list of gates**, identifying the entry points at which the segments may be called.

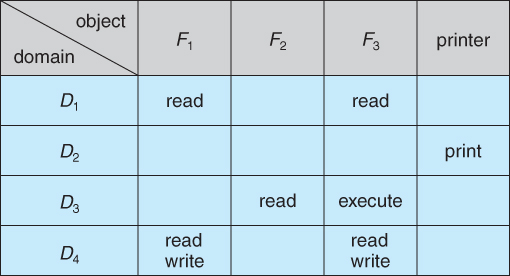
If a process operating in ring ‘p’ calls a segment whose bracket is such that b1 <= p <= b2, then the call succeeds and the process remains in ring p.

Otherwise a trap to the OS occurs, and is handled as follows:

* If p < b1, then the call is allowed, because we are transferring to a procedure with fewer privileges. However if any of the parameters being passed are of segments below b1, then they must be copied to an area accessible by the called procedure.
* If p > b2, then the call is allowed only if p <= b3 and the call is directed to one of the entries on the list of gates.

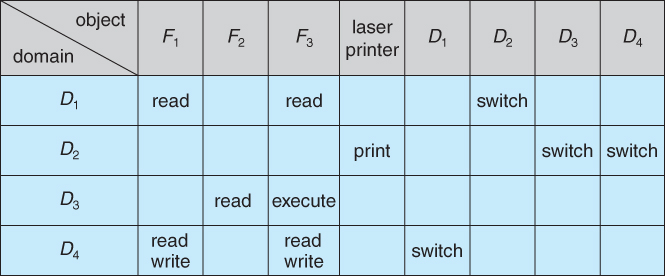
Overall this approach is more complex and less efficient than other protection schemes. (we prefer separating the mechanism from the protection policy, allowing system to have complex or simple protection depending on the needs of the user)

**14.4 Access Matrix**

The model of protection that we have been discussing can be viewed as an **access matrix**, in which columns represent different objects and rows represent different protection domains. Entries within the matrix indicate what access entries within the matrix indicate what access that domain has to that object.

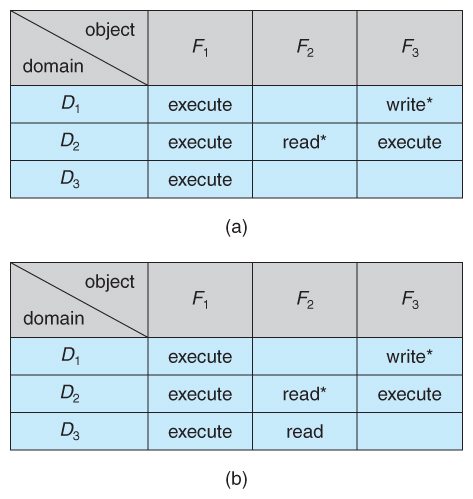
**Figure 14.3 - Access matrix**

Domain switching can be easily supported under this model, simply by providing “switch” access to other domains:

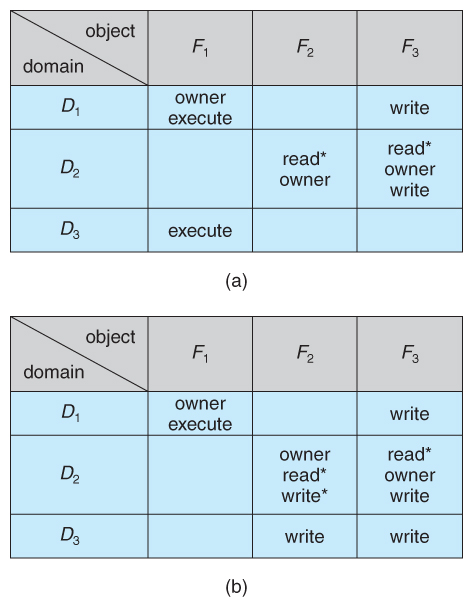
**Figure 14.4 - Access matrix of figure 14.3 with domains as objects.**

When we change the content of the access matrix, we are performing an operation on an object: the access matrix. We control these changes by including the access matrix itself as an object. Each entry can be modified individually, this means that each entry in the access matrix as an object must be protected.

The ability to **copy** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:

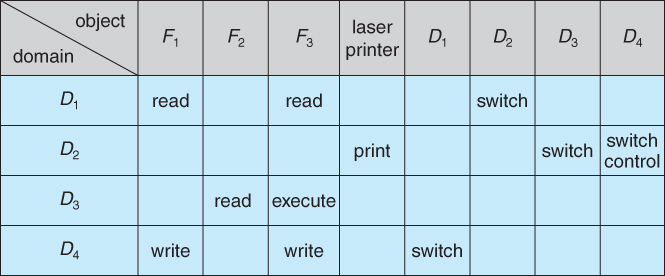
* If the asterisk is removed from the original access right, then the right is **transferred**, rather than being copied. This may be termed a **transfer** right as opposed to a **copy** right.
* ****If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a **limited copy** right, as shown in figure 14.5 below:

**Figure 14.5 - Access matrix copy rights**

****The **owner** right adds the privilege of adding new rights or removing existing ones:

**Figure 14.6 - Access matrix with owner rights**

Copy and owner rights only allow the modification of rights within a column. The addition of **control rights**, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains.

* In example below: process operating in domain D2 has the right to control any of the rights in domain D4.

**Figure 14.7 - Modified access matrix of figure 14.4**

The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem**. (generally unsolvable)

It’s up to the system designers and users to make the policy decisions concerning which domains are to have access to which objects in which ways.

**14.5 Implementation of Access Matrix**

**14.5.1 Global Table**

The simplest approach is one big global table with *<domain, object, rights >* entries.

Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques).

There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

**14.5.2 Access Lists for Objects**

Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.

For efficiency a separate list of **default set of access rights** can also be kept, and checked first, then search access list.

**14.5.3 Capability Lists for Domains**

In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.

**Capability lists** are associated with each domain, but not directly accessible by the domain or any user process. A cap. list for a domain is a list of objects together with the operations allowed on those objects.

A **capability** is an object often represented by its physical name or address. Capabilities are inherently protected objects, distinguished from other data in one of two ways:

* Each object has a ***tag,***possibly hardware implemented, to distinguishing this special type of data. (other types may be floats, pointers, booleans, etc.)
* The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by the operating system for maintaining the process's access right capability list.

**14.5.4 A Lock-Key Mechanism**

A lock-key mechanism:

* Each object has a list of unique bit patterns, termed locks.
* Each domain has its own list of unique bit patterns, termed keys.
* Access is granted if one of the domain's keys fits one of the locks of the object.
* Again, a process is not allowed to modify its own keys.

**14.5.5 Comparison**

Each of the methods here has certain advantages or disadvantages, depending on the particular situation and task at hand:

Global table:

* Simple.
* Table can be quite large and can often not take advantage of special groupings of objects or domains.

Access lists:

* Correspond directly to the user.
* Access-right information is not localized, this means that determining the set of access-rights for each domain is difficult.
* Every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time consuming.

Capability lists:

* Do not correspond directly to the needs of the user.
* Useful for localizing information for a given process.
* The process attempting access must present a capability for that access. Then, the protection system needs only to verify that the capability is valid.
* Revocation of capabilities, however, may be inefficient.

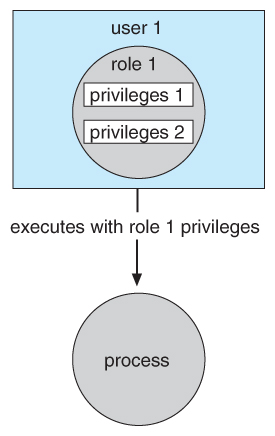
Lock-key mechanism:

* Compromise access lists and capability lists.
* Mechanism can be both effective and flexible, depending on length of the keys.
* The keys can be passed freely from domain to domain.
* Access privileges can be effectively revoked by the simple technique of changing some of the locks associated with the object.

Many systems employ some combination of the listed methods.  
Example on p. 624-625. (MULTICS and CAL system example + generic example)

**14.6 Access Control**

***Role-Based Access Control, RBAC,***assigns privileges to users, programs, or **roles** as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls. (used in Solaris)

RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to setuid or superusers programs.

**Figure 14.8 - Role-based access control in Solaris 10.**

**14.7 Revocation of Access Rights**

The need to revoke access rights dynamically raises several questions:

* Immediate versus delayed - If delayed, can we determine when the revocation will take place?
* Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
* Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
* Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?

With an access list scheme revocation is easy, immediate, and can be selective, general, partial, total, temporary, or permanent, as desired.

With capabilities lists the problem is more complicated, because access rights are distributed throughout the system. A few schemes that have been developed include:

* Reacquisition - Capabilities are periodically revoked from each domain, which must then re-acquire them.
* Back-pointers - A list of pointers is maintained from each object to each capability which is held for that object. (used in MULTICS - implementation quite costly)
* Indirection - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry, which may affect multiple processes, which must then re-acquire access rights to continue. (used in CAL - does not allow selective revocation)
* Keys - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process.  
  + A **master key** is associated with each object.
  + When a capability is created, its key is set to the object's master key.
  + As long as the capability's key matches the object's key, then the capabilities remain valid.
  + The object master key can be changed with the set-key command, thereby invalidating all current capabilities.
  + More flexibility can be added to this scheme by implementing a ***list*** of keys for each object, possibly in a global table.

**14.8 Capability-Based Systems**

**14.8.1 An Example: Hydra**

Hydra is a capability-based system that includes both system-defined ***rights*** and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be.

Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities.

The names of user-defined procedures must be identified to the protection system if it is to deal with user-defined rights.

When an object is created, the names of operations defined on that object become **auxiliary rights,**described in a capability for an **instance** of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.

Hydra also allows **rights amplification,**in which a process is deemed to be **trustworthy,**and thereby allowed to act on any object corresponding to its parameters.

Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

**14.8.2 An Example: Cambridge CAP System**

The CAP system has two kinds of capabilities:

* ***Data capability,***used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.
* ***Software capability,***is protected but not interpreted by the CAP microcode.
  + Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.
  + When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability.
  + This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.
  + Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.
  + Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

**14.9 Language-Based Protection**

Language-based protection:

* As systems have developed, protection systems have become more powerful, and also more specific and specialized.
* To refine protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

**14.9.1 Compiler-Based Enforcement**

In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.

This approach has several advantages:

1. Protection needs are simply declared, as opposed to a complex series of procedure calls.
2. Protection requirements can be stated independently of the support provided by a particular OS.
3. The means of enforcement need not be provided directly by the developer.
4. Declarative notation is natural, because access privileges are closely related to the concept of data types.

Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.

Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments can’t be modified, data segments can't be executed.)

There are several areas in which compiler-based protection can be compared to kernel-enforced protection:

* **Security.** Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.
* **Flexibility.** A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.
* **Efficiency.**The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that many checks can be made off-line, at compile time, rather than during execution.

The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:

1. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.
2. Specifying the ***type*** of operations a process may execute on a resource, such as reading or writing.
3. Specifying the ***order*** in which operations are performed on the resource, such as opening before reading.

**14.9.2 Protection in Java**

Java was designed from the very beginning to operate in a distributed environment, where code would be executed from a variety of trusted and untrusted sources. As a result the Java Virtual Machine, JVM incorporates many protection mechanisms

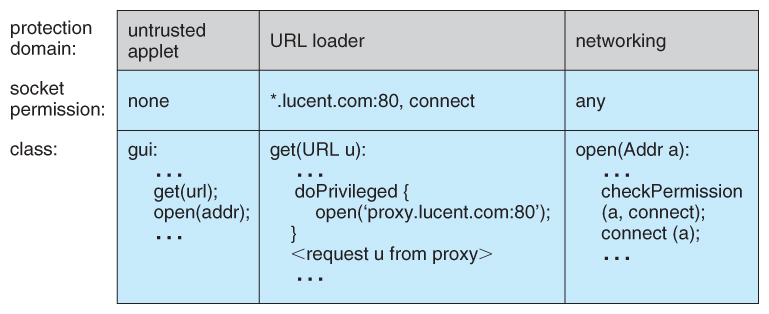
When a Java program runs, it load up classes dynamically, in response to requests to instantiates objects of particular types. These classes may come from a variety of different sources, some trusted and some not, which requires that the protection mechanism be implemented at the resolution of individual classes, something not supported by the basic operating system.

As each class is loaded, it is placed into a separate protection domain. The capabilities of each domain depend upon whether the source URL is trusted or not, the presence or absence of any digital signatures on the class (Chapter 15), and a configurable policy file indicating which servers a particular user trusts, etc.

When a request is made to access a restricted resource in Java, (e.g. open a local file), some process on the current **call stack** must specifically assert a privilege to perform the operation. In essence this method **assumes responsibility** for the restricted access. Naturally the method must be part of a class which resides in a protection domain that includes the capability for the requested operation. This approach is termed **stack inspection,** and works like this:

* When a caller may not be trusted, a method executes an access request within a doPrivileged( ) block, which is noted on the calling stack.
* When access to a protected resource is requested, checkPermissions() inspects the call stack to see if a method has asserted the privilege to access the protected resource.
* If a suitable doPrivileged block is encountered on the stack before a domain in which the privilege is disallowed, then the request is granted.
* If a domain in which the request is disallowed is encountered first, then the access is denied and a AccessControlException is thrown.
* If neither is encountered, then the response is implementation dependent.

In the example below the untrusted applet's call to get( ) succeeds, because the trusted URL loader asserts the privilege of opening the specific URL lucent.com. However when the applet tries to make a direct call to open( ) it fails, because it does not have privilege to access any sockets.

  
**Figure 14.9 - Stack inspection.**

Generally, Java’s load-time and run-time checks enforce **type safety** (public, private, …) of java classes. Type safety ensures that classes cannot treat integers as pointers, write past the end of an array, or otherwise access memory in arbitrary ways.  
Rather, a program can access an object only via the methods defined on that objects via its class. This is the foundation of Java protection, since it enables a class to effectively **encapsulate** and protect its data and methods from other classes loaded in the same JVM.

**14.10 Summary**

Computer systems contain many **objects**, and they **need to be protected** from misuse. Objects may be **hardware** (such as memory, CPU time, and I/O devices) or **software** (such as files, programs, and semaphores).   
An **access right** is permission to perform an operation on an object. A **domain** is a set of access rights in the domain to access and manipulate objects. During its lifetime, a **process** may be either bound to a protection domain or allowed to switch from one domain to another.

The **access matrix** is a general model of protection that provides a mechanism for protection without imposing a particular protection policy on the system or its users. The **separation of policy and mechanism** is an important design property.  
  
The access matrix is sparse. It is normally implemented either as **access lists** associated with each object or as **capability lists** associated with each domain.   
We can include **dynamic protection** in the access matrix model by considering domains and the access matrix itself as objects. Revocation of access rights in a dynamic protection model is typically easier to implement with an access list than with a capability list.  
  
**Real systems** are much more **limited than the general model** and tend to provide protection only for files.  
**UNIX** is representative, providing read, write, and execution **protection** separately for the owner, group, and general public **for each file**.   
**MULTICS** uses **a ring structure** in addition to file access.   
**Hydra**, the Cambridge **CAP** system, and **Mach** are **capability systems** that extend protection to user-defined software objects.  
**Solaris 10** implements the principle of least privilege via **role-based access control**, a form of the access matrix.  
  
**Language based protection** provides finer-grained arbitration of request and privileges than the operating systems is able to provide.   
For example, a single **Java** JVM can run several threads, each in a different protection class. It enforces the resource requests through sophisticated **stack inspection** and via the **type safety** of the language.