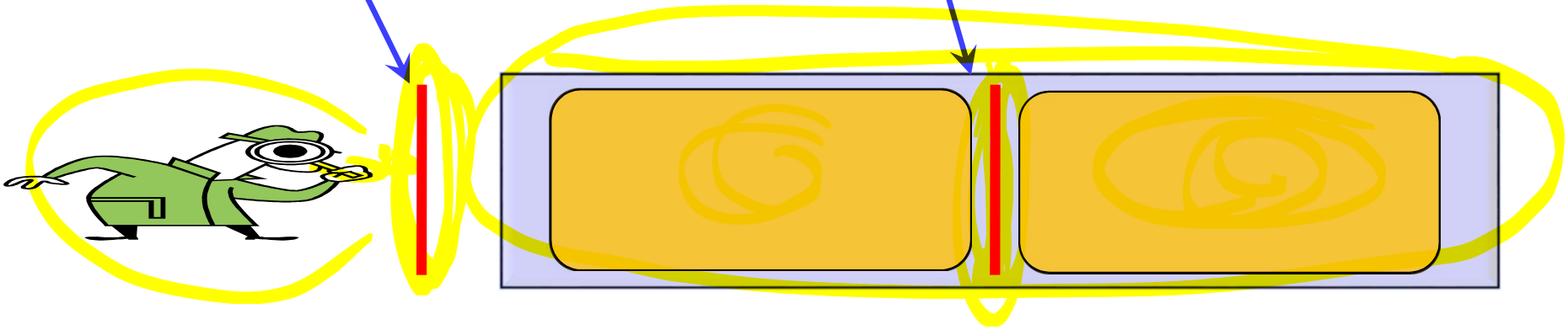
Protection and security

Protection and Security

- **Security:** Preventing the access from external agents.
  - To authenticate the system users to protect the integrity of the information stored in the system.
- **Protection:** Preventing the access of internal users from accessing resources that they are not allowed to.
  - To control the access of programs, processes or users to the resources provided within a computer system.



Protection and security are related but different concepts of OS's.

Goals of Protection

- **Protection problem stems from multiprogramming OS's**
  - untrustworthy users can safely share common resources, such as memory, files.
- **Protection**
  - Prevent violation of access restriction by a user
  - Distinguish between authorized and unauthorized usage
  - Provide a mechanism for the enforcement of the policies governing resource use
    - Some are fixed during the design of the system
    - Some are set by the management of the system
    - Others are defined by the users of the system

Principles of Protection

- **Guiding principle – principle of least privilege**
  - Programs, users and systems should be given just enough privileges to perform their tasks

- Ask:
  - What is the lowest set of privileges allowable for this user's tasks?
  - How long are the privileges required?

- If you hire a gardener,
  - grant them access to your yard – not your bedroom.
  - grant them access for the time they're working

Domain of Protection

- **Operating system consists of a collection of objects, hardware or software**
  - Files, directories, hardware, ..
    - A file can be readable but not writable..
- **Each object has a unique name and can be accessed through a well-defined set of operations.**
    - A CPU can only be executed on
    - Memory can be read or written
    - CD-ROM can only be read
- **Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.**

Domain Structure

- A protection domain specifies the resources that the process may access.
- Each domain defines
  - a set of objects and
  - the types of operations that may be invoked on each object.

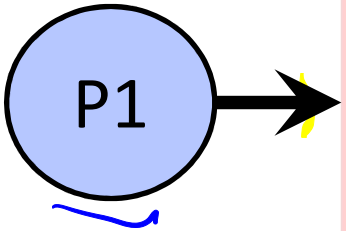
Domain = {access-right}

access-right = <object-name, {right}>

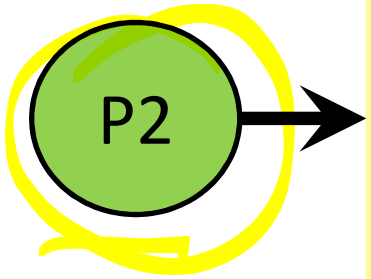
right = {read, write, execute ...}

D = {(O1, r, w) (O2, x) ... }

Domain structure



Domain1 = { <file-A, {read, write, execute}>,
<file-B, {read, execute}>,
<page-2, {read}>,
< printer-1, {print}>}

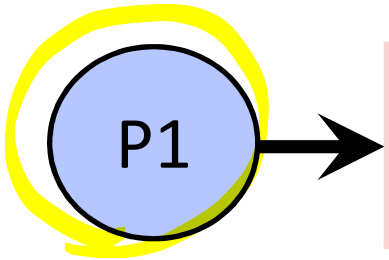


Domain2 = { <file-A, {read, write}>,
<file-B, {read, execute}>,
<page-1, {read, execute}>,
< printer-1, {print}>}

- Domains need not be disjoint.
- Both P1 and P2 processes can print on printer-1.
- Only P1 can execute file-A.
- Only P2 can read page-1.

Domain Structure - static/dynamic

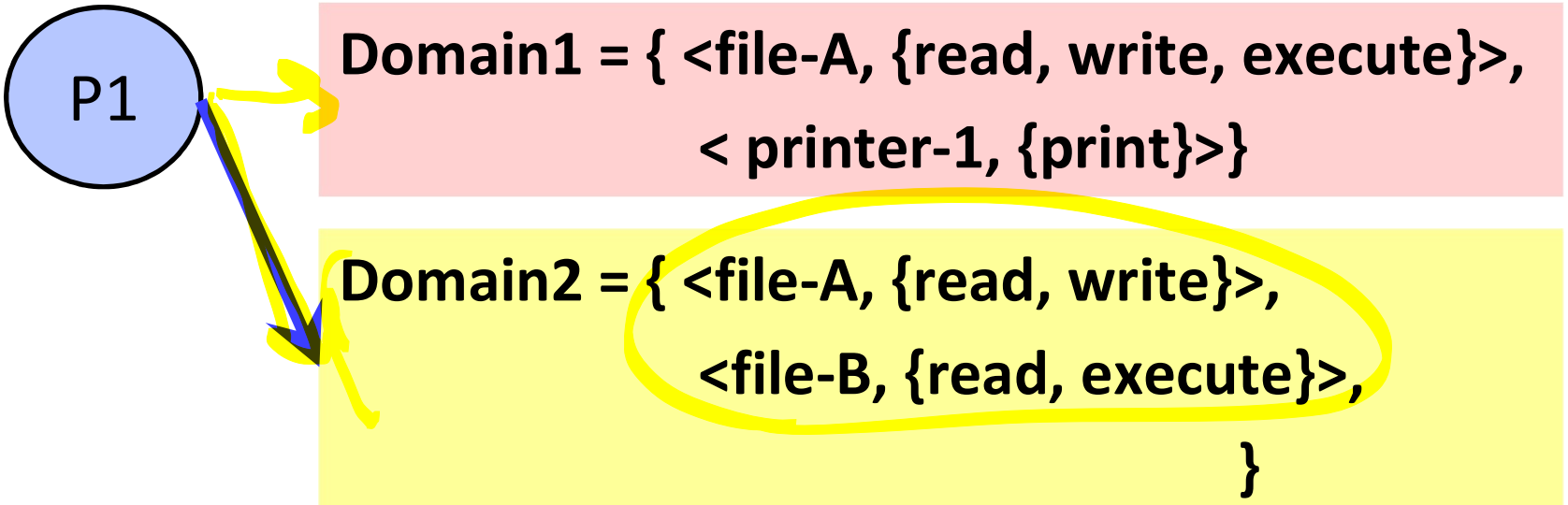
- **Associations between a process and a domain can be**
  - **Static**
    - Fixed at the time of creation of the process
    - May need to provide more rights than needed at the run time
  - **Dynamic**
    - A process can switch from one domain to another
    - The content of the domain can also be changed



Domain1 = { <file-A, {read, write, execute}>,
< printer-1, {print}>}

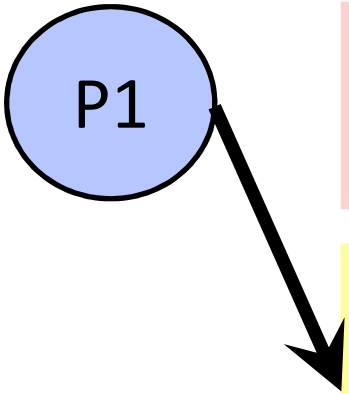
Domain Structure - static/dynamic

- **Associations between a process and a domain can be**
  - **Static**
    - Fixed at the time of creation of the process
    - May need to provide more rights than needed at the run time
  - **Dynamic**
    - **A process can switch from one domain to another**



Domain Structure - static/dynamic

- **Associations between a process and a domain can be**
  - **Static**
    - Fixed at the time of creation of the process
    - May need to provide more rights than needed at the run time
  - **Dynamic**
    - A process can switch from one domain to another
    - **The content of the domain can also be changed**



Domain1 = { <file-A, {read, write, execute}>,
< printer-1, {print}>}

Domain2 = { <file-A, {read, write}>,
<file-B, {read, execute}>,
< printer-1, {print}>}

Domain design

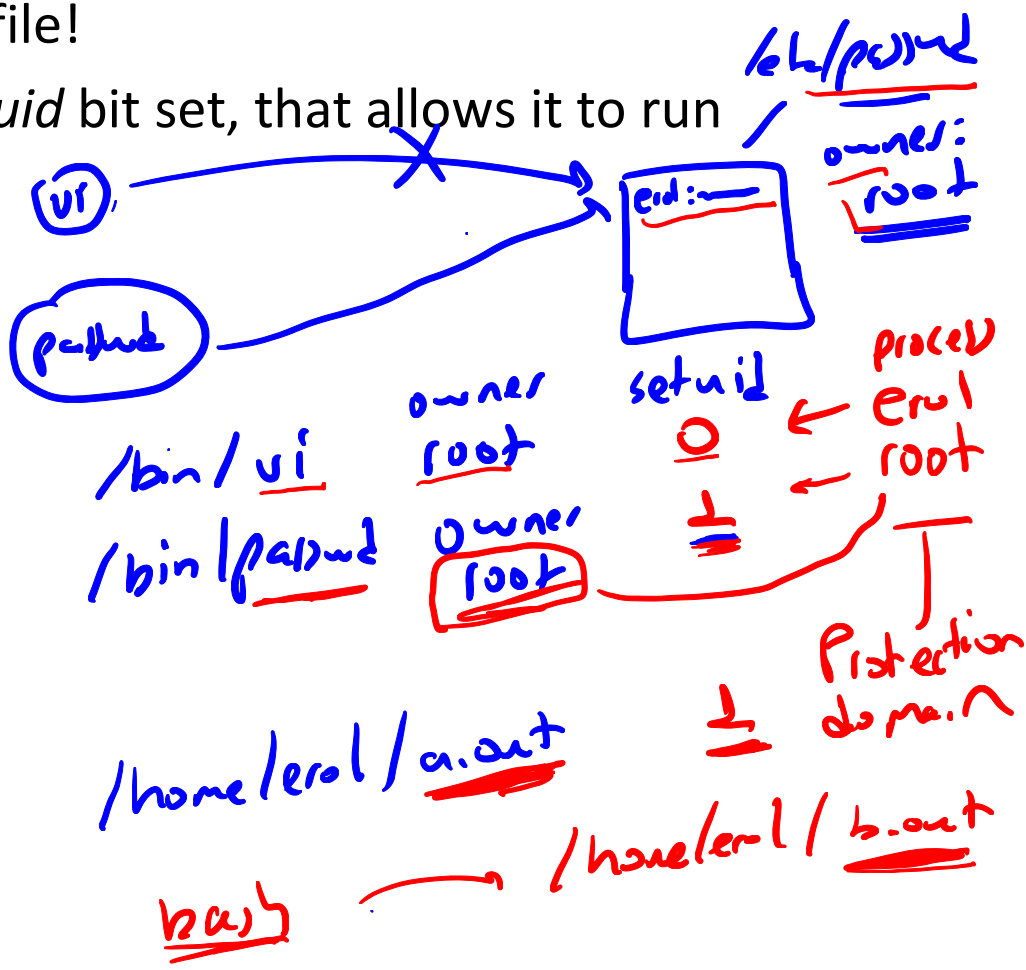
- **Each *user* may be a domain.**
  - Access rights depend on the **identity of the user**.
  - **Domain switching** occurs **when the user is changed**.
- **Each *process* may be a domain.**
  - Access rights depend on the **identity of the process**.
  - **Domain switching** corresponds to **one process sending a message to another process, and then waiting 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**
    - As an example, user and kernel modes define a dual domain system where the processes that run in kernel mode have the right to execute privileged instructions.
    - But we also need to protect users from each other!

Domain Implementation (UNIX)

- **A domain is associated with the user through *uid* (user id) and *guid* (group id)**
  - Switching domain = changing user identification temporarily
- **Domain switch accomplished via file system.**
  - Each file is associated with
    - An owner identification
    - a domain bit (known as the **setuid bit**).

setuid bit – how passwd works

- **How does the *passwd* program work**
  - When executed by the user, the process runs in the user's domain
    - Cannot modify the /etc/passwd file!
  - **Solution**: *passwd* program has its *setuid* bit set, that allows it to run with root access
    - Modify /etc/passwd file



- **User (with *user-id* = *A*) starts executing a file owned by *B*.**
  - When the setuid bit is *off*
    - the user-id of the process is set to A.
  - When the setuid bit is *on,*
    - the user-id of the process is set to B.

Model of Protection: Access Matrix

- **View protection as a 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 $Object_j$

	O1	O2	O3	O5
D1	read, write, execute, owner	read	access	read, write
D2	read, execute, switch(D1)	read, write, owner	access	read, write
D3			Read, write, access, owner	read
D4			access	read, write, owner
D5			access	read, write
D6			access	read

Access Matrix - dynamic protection

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
  - switch – switch from domain D_i to D_j

Example:

- A Unix ls -l output and content of /etc/group

```
Protection  Owner   Group   Object
-rwsr-x---  obi     jedi    useforce
-rw-r-----  luke    jedi    3po.man
drwx--x--x  darth   sith    ds.plan
-rw-rw-r--  han     free    mf.jpeg
```

```
GroupName:ShadowPass:UserList
jedi:x:yoda,obi,luke
sith:x:emperor,vader,doku
free:x:han,lea,obi,luke
robot:x:r2d3,3po
```

- Access matrix:

	useforce	3po.man	ds.plan	mf.jpeg
obi	read, write, execute, owner	read	access	read, write
luke	read, execute, switch(obi)	read, write, owner	access	read, write
darth			Read, write, access, owner	read
han			access	read, write, owner
lea			access	read, write
r2d2			access	read

Switching between domains

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			

- *A process executing in D_2 can switch to domain D_3 or D_4*
- *A process executing in D_4 can switch to D_1*

- **Linux sudo implements a domain switch controlled in /etc/sudoers**
- **Allowing controlled change to the contents of the access-matrix entries requires three additional operations: copy , owner , and control .**

Access Matrix with *Copy* Rights

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	write

(b)

- The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right.
- A process in D_2 can copy the read operation into any entry associated with file F_2

- Example:
  SQL GRANT with "GRANT OPTION":
  `GRANT INSERT,DELETE ON TABLE mytable WITH GRANT OPTION;`

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		write*

(a)

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

(b)

- **Two possible variants:**
  - A right is copied from access (I,j) to access (k,j); it is then removed from access (i,j); this action is a transfer of a right, rather than a copy.
  - Propagation of the copy right may be limited. That is, when the right R* is copied from access (i,j) to access (k,j), only the right R (not R*) is created. A process executing in domain cannot further copy the right R.

Access Matrix With *Owner* Rights

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)

- **Owner rights should allow the addition of new rights and removal of some rights.**
- **Unix implements chmod() system call to update access matrix by owner.**

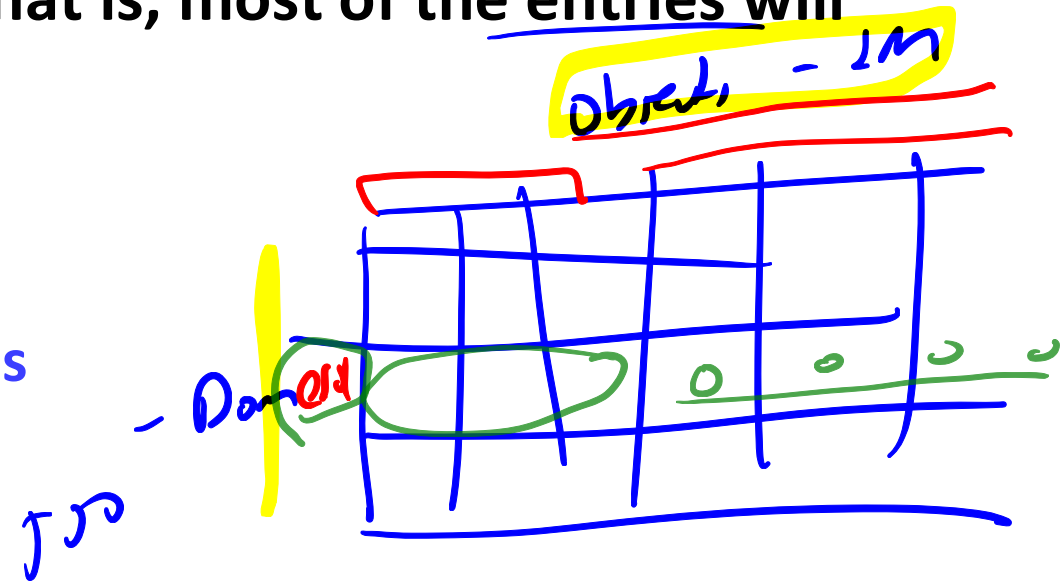
Access Matrix With *Control* Rights

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 control
D_3		read	execute					
D_4	write		write		switch			

- **D_2 can modify D_4 row.**
- **For example: Unix root has control right on all other domains**
- **The *copy* and *owner* rights provide us with a mechanism to limit the propagation of access rights.**
  - ***However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information***
- **The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called**
  - ***the confinement problem***

Implementation of Access Matrix

- How can the access matrix be implemented effectively?
- The matrix will be sparse; that is, most of the entries will be empty.
  - Global table
  - Access lists for Objects
  - Capability lists for Domains
  - A Lock-Key Mechanism



Global Table

- **A global table consisting of**
  - <domain, object, rights-set> triples

D1	O1	read, write, execute, owner
D1	O2	read, write
D1	…	…
D2	O1	read, execute, switch(obi)
D2	O2	read, write, owner
D2	…	….

Global Table – pros and cons

- **Simplest implementation**
- **The table is usually large and cannot be kept in memory**
- **Does not take into account special groupings of objects or domains**
  - If everyone can read a particular object, it must have a separate entry in every domain

D1	O1	read, write, execute, owner
D1	O2	read, write
D1	…	…
D2	O1	read, execute, switch(obi)
D2	O2	read, write, owner
D2	…	….

Access Lists for Objects

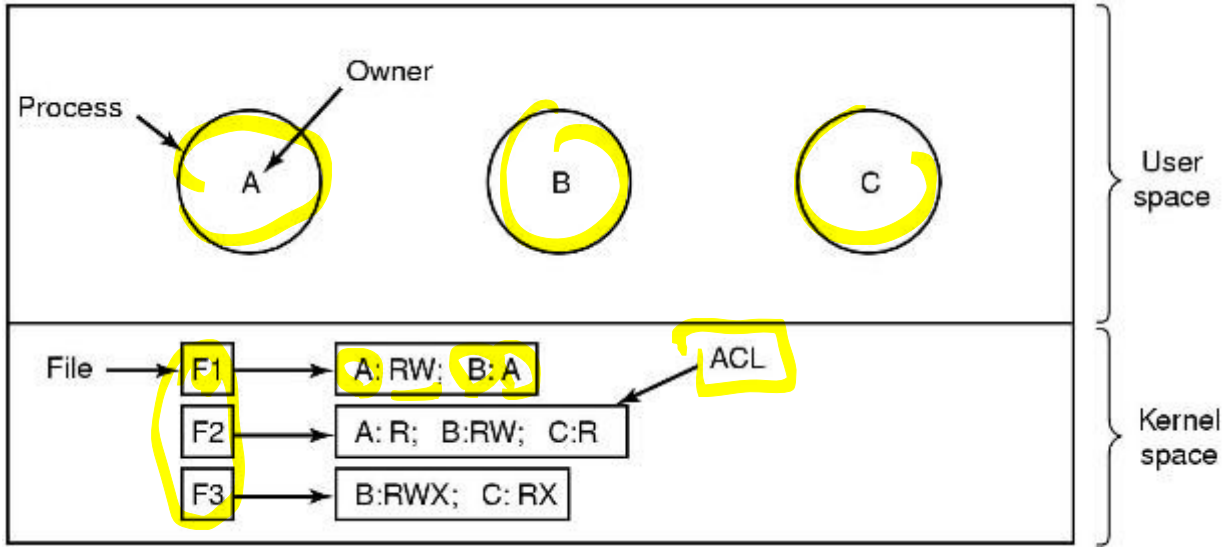
- **For each object store <domain, rights-set>, which define all domains with a nonempty set of access rights for that object.**
  - can be extended easily to define a list plus a default set of access rights.

Object1	D1	read, write, execute, owner
	D2	read, execute, switch(obi)

Object2	D1	read, write
	D2	read, write, owner

Access Lists for Objects - example

- **For each object store <domain, rights-set>, which define all domains with a nonempty set of access rights for that object.**
  - can be extended easily to define a list plus a default set of access rights.



Domain: User = {A,B,C}
Object: File = {F1, F2, F3}
Rights: {R, W, X}

File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Capability Lists for Domains

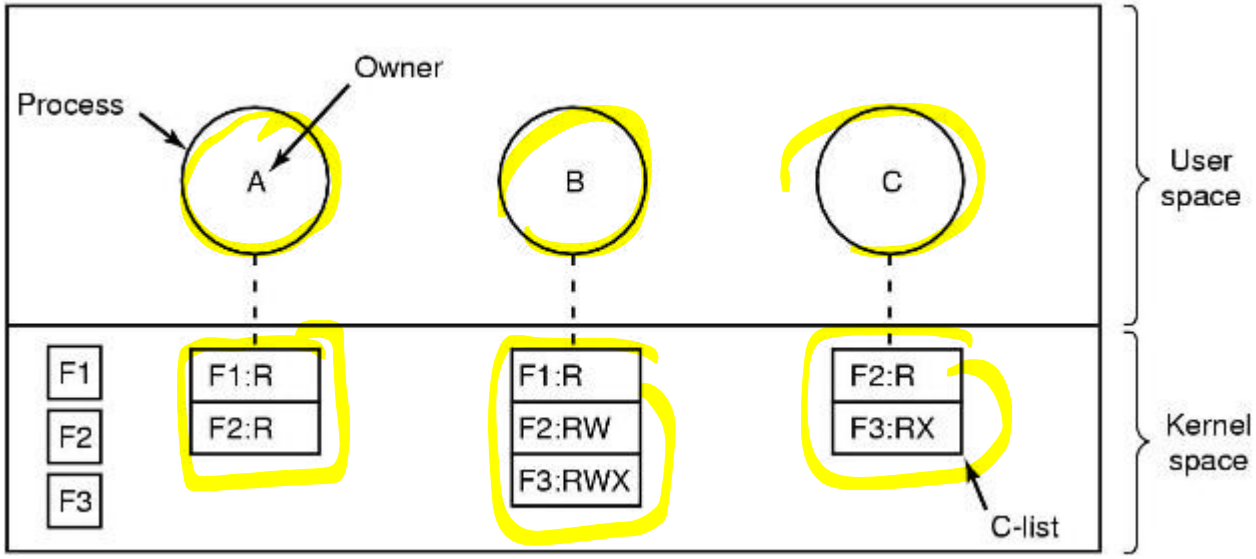
- **A capability list for a domain is a list of objects together with the operations allowed on those objects.**
  - The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain.
  - Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly.

Domain1	O1	read, write, execute, owner
	O2	read, execute, switch(obi)

Domain2	D1	read, write
	D2	read, write, owner

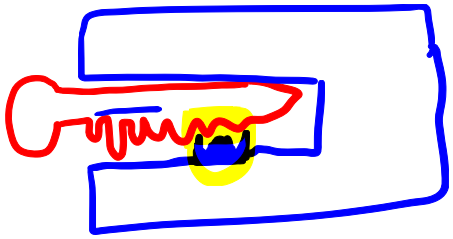
Capability Lists for Domains - example

- **A capability list for a domain is a list of objects together with the operations allowed on those objects.**
  - The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain.
  - Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly.



Domain: User = {A,B,C}
Object: File = {F1, F2, F3}
Rights: {R, W, X}

Lock-Key Mechanism



- **The lock–key scheme is a compromise between access lists and capability lists.**
  - Each object has a list of unique bit patterns, called locks.
  - Similarly, each domain has a list of unique bit patterns, called keys.
  - A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

- **The list of keys for a domain must be managed by the OS on behalf of the domain.**
- **Users are not allowed to examine or modify the list of keys (or locks) directly.**

Access lists vs. Capability lists vs. Lock-key

- **Access lists correspond directly to the needs of the users.**
  - When a user creates an object, she can specify which domains can access the object, as well as the operations allowed.
- **Capability lists do not correspond directly to the needs of the users; they are useful, however, for localizing information for a given process.**
  - The process attempting access must present a capability for that access.
- **The lock–key mechanism is a compromise between these two schemes.**
  - The mechanism can be both effective and flexible, depending on the length of the keys.
  - The keys can be passed freely from domain to domain.

Authentication

- **Crucial part of OS security.**
- **If a request is really done by a user/host that it claims.**
- **Host authentication:**
  Mostly relates to network requests.
  Out of scope for this course.

- **User authentication: done when user starts a session or asks a privileged operation.**
- **Authentication factors:**
  - Something (only) you know (password, pin code, TCKN?)
  - Something you have (id card, credit card, cell phone, smart card)
  - Something you are (finger, retina, blood, DNA sample…)

Password Authentication

- **Relies on only user knows a common passphrase.**
- **User password is compared against the information stored on system.**
  - A match results in success.
- **Password is the critical part of security.**
  - Protecting password database is crucial.

- **Bad idea: storing passwords in plain.**
  - If protection of password database is compromised security of system collapses.
  - Privileged users can see content. Use it for other systems
- **Solution use cryptography.**
- **Hash/digest functions: map a string of bytes into a fixed string where:**
  - Given the result, original string cannot be computed
  - Small change in input string ends up extensive changes in result, no correlation can be found.
  - Having two input strings result in same has value is extremely unlikely.

Password Authentication

- **User passwords are stored in database as crypto hashed values.**
- **With cryptohash() function, authentication becomes:**
  - Input “uname” and plain password “ppass” from user
  - Calculate cpass = cryptohash(ppass)
  - Check password database for an entry username==uname and password == cpass
- **No cryptohash(cpass) function giving ppass is defined.**
- **POSIX define crypt(key, salt) functions for password test:**
  `strcmp(crypt(ppass, salt), cpass) == 0`
- **`/etc/shadow` is used as password store in a standalone Unix/Linux system**
- **Not a perfect solution, vulnerable to:**
  - Dictionary attacks: Test all possible passwords from a dictionary
  - Social engineering attacks: Learn information from user, birthday, team he is supporting etc.
  - Key-loggers intercepting user input and reporting to third parties.

One Time Passwords (OTP)

- **Major problem in password authentication is its lifetime.**
  - A user can use same password for year.
  - Frequent changes of password/pin code is required.
- **OTP uses cryptography to generate dynamic passwords as user is authenticated or by time.**
- **Sequence based:**
  $OTP_t = otpgen(secret, OPT_{t-1})$
- **Time based:**
  $OTP_t = otpgen(secret, time\ of\ day)$
- **User cannot compute `otpgen` so either it is precomputed or s/he is given a device to generate OTP's as needed:**
  - OTP token devices / cell phone applications
- **OTPs turn into "something you have" factor authentication**

Third Party Authentication

- As implementing password or OTP based authentication per target system gets complicated, authentication may need to be centralized.
- User asking for authentication is sent to authentication services on network.
- User authenticates him/herself in server, gets a ticket.
- Ticket is given back to the original system to finish authentication.
- Cryptography makes sure ticket is coming from the trusted service.
- Protocols and services exists like kerberos, Openid, Oauth

Multi-factor Authentication

- **High security systems and software requires at least two factor in authentication:**
  - password + mobile SMS
  - Credit card + pin code
  - Retina scan + id card
- **Something you have and something you are requires hardware devices to implement**
  - Fingerprint scanners
  - Retina scanners
  - Smart cards + readers
  - Mobile phones
  - SIM cards

Other Uses of Cryptography in OS

- **Cryptographic hash functions/digests:**
  - Integrity of data. If a file (i.e. a system binary) has changed in system. For example a virus.
  - Software package authentication.
- **Symmetric cryptography:**
  - Data privacy.
  - Encrypted content (disk, files, messages)
- **Public key cryptography:**
  - Integrity of data (message signing and verification)
  - Authentication (electronic certificates)
  - Encryption (encryption without shared key)
  - Key exchange