Operating Systems Protection Mechanisms

Ibéria Medeiros
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

Introduction

- Operating systems are a crucial component of computer security
  - ☞ so we'll see the basic protection mechanisms they provide
  - ☞ and discuss some issues

- Since modern OSs support multiprogramming, they must provide protection
  - ☞ ***among users***, namely between a legitimate user and an intruder / malware
  - ☞ ***of the OS itself*** from users, intruders and malware

PROTECTION OF RESOURCES

Protection

- A computer contains several resources called **objects**
  - ☞ Memory pages, memory segments
  - ☞ I/O devices (disks, networks, printers, monitors)
  - ☞ Dynamic libraries
- Objects are accessed by **subjects**
  - ☞ users, groups, processes

- Role of the OS: ensure that objects are only accessed by authorized subjects
  - ☞ processes only have ***direct*** access to their own memory segments
  - ☞ each file can be access only by a ***set*** of users

Protection

- There are two main mechanisms to ensure that objects are not accessed by unauthorized subjects

  ☞ **<u>separation</u>** – prevent arbitrary access to objects
  - needs to separate objects from subjects

  ☞ **<u>mediation</u>** – control the access to the object
  - decides what kind of access a subject has to a object

… lets start with *separation* first …

SEPARATION

Separation in the OS

Nowadays, there is a cooperation between the operating system and the hardware to enforce separation!

- Common operating systems (Unix, Windows) run software in two modes (aka levels, rings)
  - ☞ **Kernel mode** – software can play with any system resource (memory, I/O devices,...)
  - ☞ **User mode** – access to resources is *controlled* by the OS

- These modes are *enforced by the CPU*
  - ☞ Simply disables a set of its instructions in user mode (e.g., in/out, sti/cli, hlt)

    where "*disable*" means: generates exception or does nothing if a process tries to execute them, depending on the instruction

Separation in the OS (cont)

- In <u>user mode</u>, how can a software execute operations in objects outside their control?
  - ☞ software has to call the OS kernel to have it perform the privileged operations, using *system calls* – a sort of function but in the OS
  - ☞ the OS can then control accesses from user mode programs to all objects outside their memory, including system resources

- Two difficulties
  - ☞ OS kernel runs in kernel mode, not user mode
  - ☞ the kernel memory space is invisible to the process (jump?)
- Solution
  - ☞ software interruption (aka exception, trap), triggered by a special instruction (e.g., *int* in Intel CPUs) that forces the CPU to change to kernel mode

Example

- A program that opens a file

```
int main(int argc, char **argv) {
    open("/tmp/test.txt", O_WRONLY);
}
```

- The executed system call

```
.LC0:
    .string  "/tmp/test.txt"
    < ... >
    movl    $1, 4(%esp)      ; save the O_WRONLY on the stack
    movl    $.LC0, (%esp)    ; save the location of file name
    call    open             ; execute system call
    < ... >
```

Memory protection

- System calls solves the separation issues for most resources of the machine, but *what about memory*?
  - ☞ accesses to memory cannot be performed through system calls because of performance issues
  - ☞ so, what prevents a process in user mode from changing the memory of another process or the kernel?
  - ☞ by changing memory one can modify the loaded code, and therefore alter the behavior of the kernel or other process

- To enforce this protection there needs to be cooperation (again) between the hardware + OS

Strategies for separation

- We need a way to separate the memory of the various processes and kernel among them
- Possible strategies:
  - ☞ Physical separation: different processes use distinct devices (e.g., printers for different levels of security)
  - ☞ Temporal separation: processes with different security requirements are executed at different times
  - ***memory separation*** ☞ Logical separation: processes operate under the illusion that no other processes exist
  - ☞ Cryptographic separation: processes use crypto to conceal their data and/or computations in a way that they become unintelligible to other processes (e.g., Intel SGX)

Separation for memory protection

- Several solutions have been proposed over the years, but we are interested in those currently used
  - ☞ Segmentation
  - ☞ Paging
  - ☞ Segmentation + Paging

Segmentation

- A program's memory is split in several parts called ***segments***
  - ☞ typical segments: *code, data, stack*, *heap*, …
  - ☞ each segment has a **name**
  - ☞ memory is (logically) addressed by ***(name, offset)***
  - ☞ this addressing allows for segments to be placed in any point of physical memory and to be relocated to other areas
  - ☞ they can also be stored in auxiliary memory (disk)

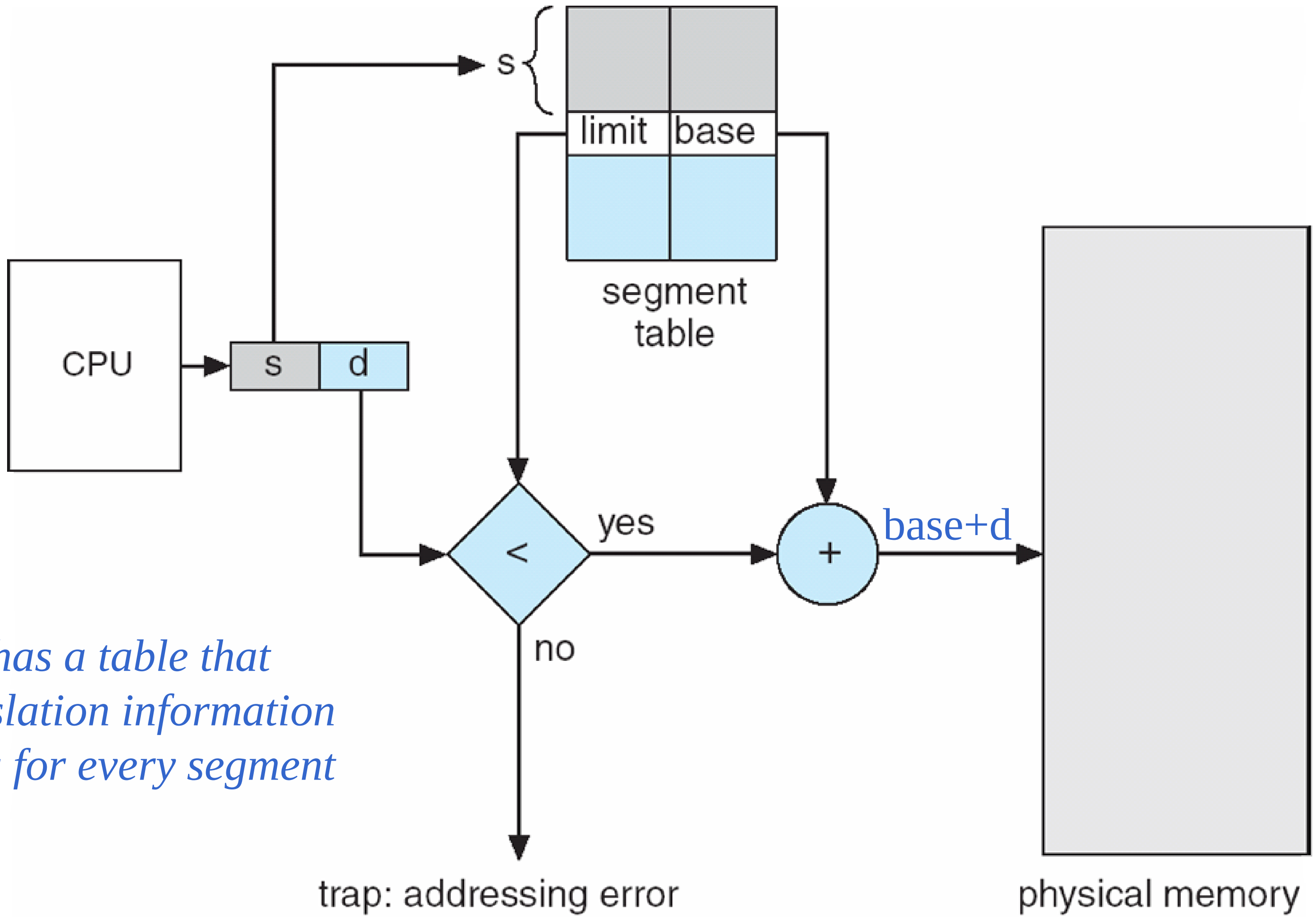
- How do we implement this?
  - ☞ there is a ***translation table*** with the beginning of each segment in memory per process to translate name to an address
  - ☞ due to performance reasons, the table is managed cooperatively between the OS and the hardware, through the CPU ***Memory Management Unit (MMU)***

Segmentation (cont.)

- The ***translation table*** allows
  - ☞ a process can access a segment only if the segment appears in its translation table
  - ☞ info about access rights (ex: READ, WRITE, EXECUTE) is also stored in the table
  - ☞ every memory access has to go through the OS/MMU so access rights can be checked (e.g., no exec on data segments)
  - ☞ there can be several segments for data with different access rights

- Summary: memory protection ensures
  - ☞ a process cannot access (read/write/execute) memory areas that do not belong to it (other processes and the OS)
  - ☞ a segment of memory can only be accessed (read/write/exec) accordingly to the mode indicated in the translation table, which is setup by the OS

Segmentation: address translation



each process has a table that contains the translation information plus access rights for every segment

Segmentation (cont.)

- <u>Difficulties of segmentation</u>

  ☞ checking efficiently if memory accesses are beyond the end of the segment (might be hard as segments have different sizes)
    - what is the problem of going beyond the segment?

  ☞ can cause fragmentation of the memory (sizes vary, and they can grow with time)

Paging

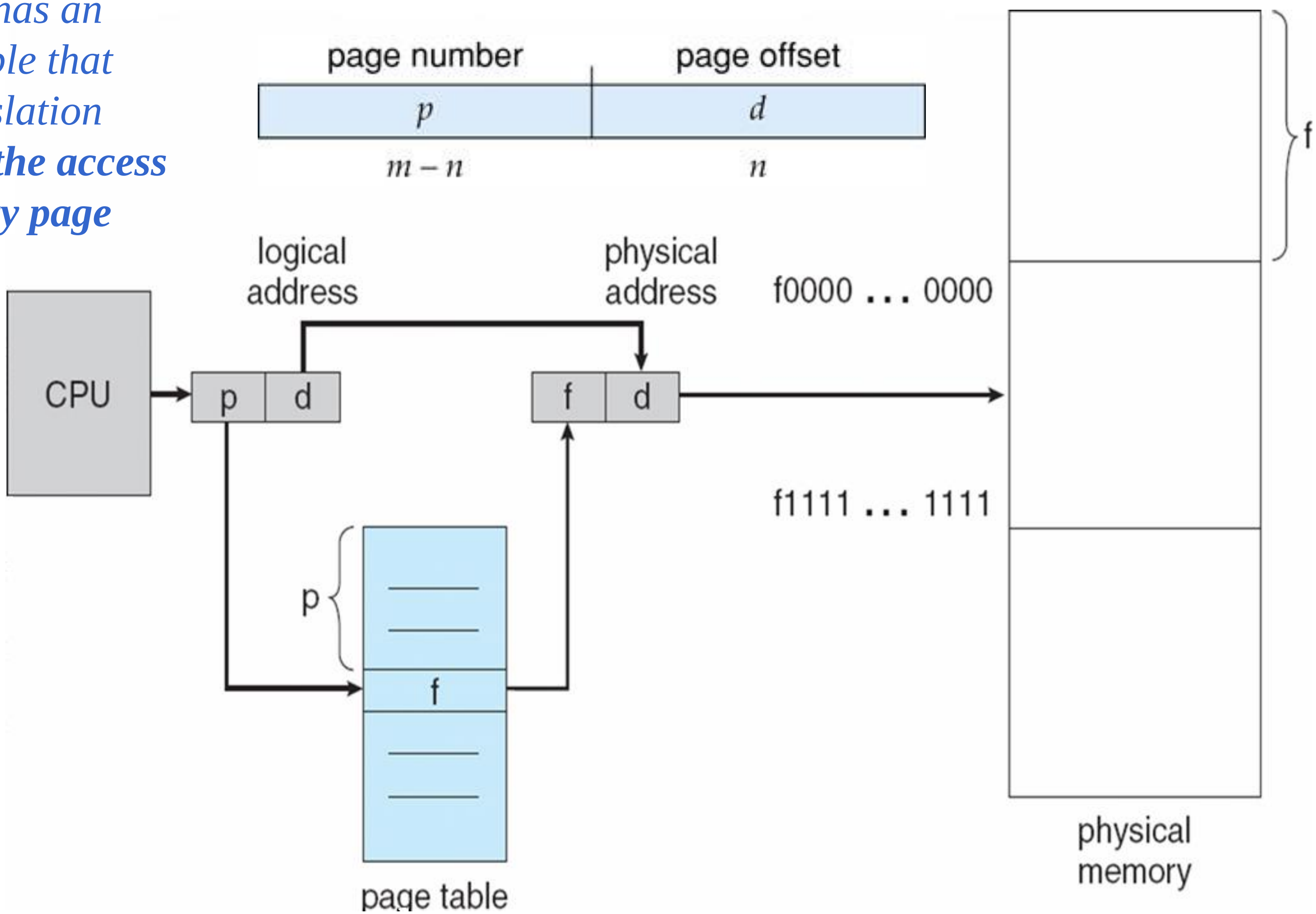
- The memory of a process is divided in **pages** of the same size (e.g., 4KB, typically power of 2)
  - ☞ physical memory is divided in **page frames** of the same size

    … so, there is no fragmentation and knowing the end is trivial
  - ☞ memory is addressed by ***(page, offset)***
  - ☞ pages have no logical unity (on the contrary to segments)

- From a protection point of view, pages are similar to segments
  - ☞ a process sees a physical page only if the page appears in its *page translation table*
  - ☞ information about access rights (ex: WRITE, EXECUTE) is stored in the table, and access rights are enforced per access

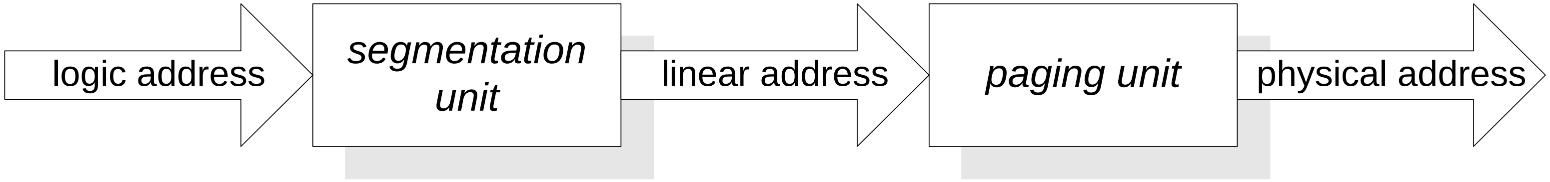
Paging: address translation

each process has an associated table that contains translation information ***plus the access rights for every page***

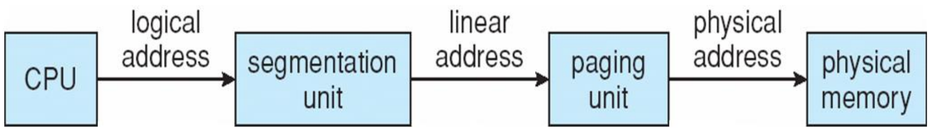


Segmentation + paging

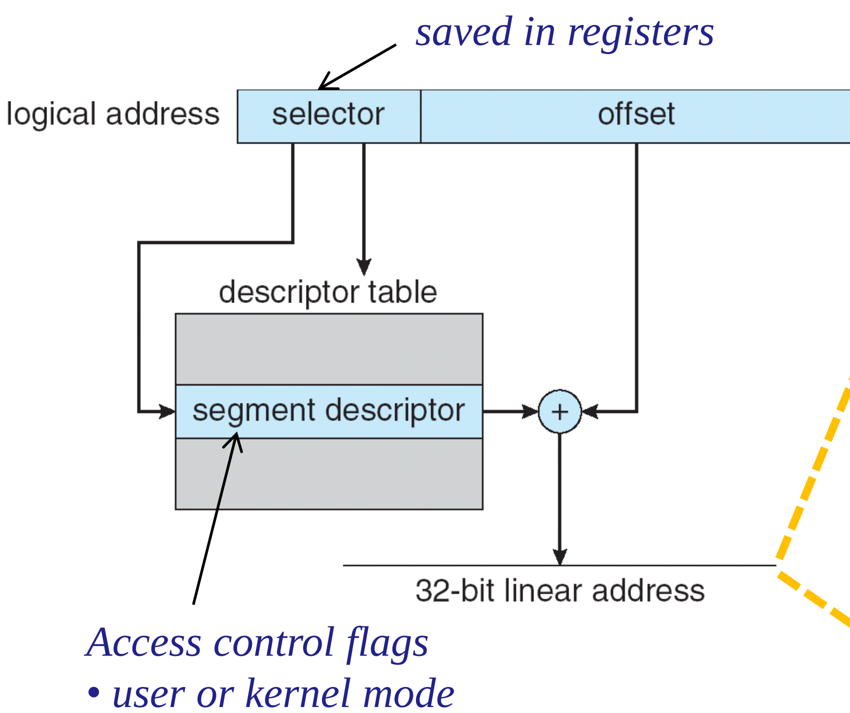
- Several architectures support both segmentation + paging, e.g., Intel x86 with 32 bits
- Linux on x86 32 bits uses both
  - ☞ programs use *logic addresses* composed of
    - – **segment selector** (16 bits) stored in a CPU register (CS, DS, SS)
    - – **offset** (32 bits)
  - ☞ converted by the MMU to *linear addresses*
    - – address of the virtual memory, split in 4KB **pages** (32 bits)
  - ☞ converted by the MMU to *physical addresses*
    - – if the page is not in RAM, then a *page fault* is generated



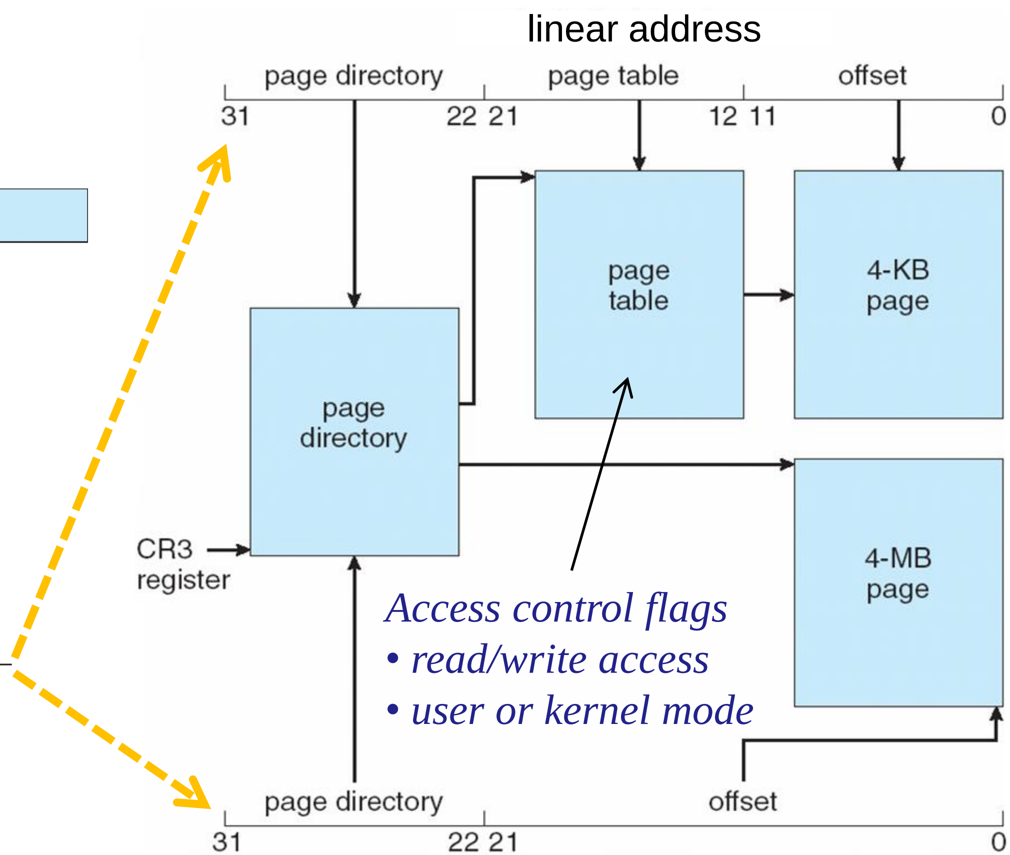
Intel Pentium



Segmentation



Paging



Linux/x86 : Segments

- Each process (and kernel) has different *segment selectors*, which means the OS has to update the CPU registers in every context switch
- The segment selector contains 2 bits to indicate the Current Privilege Level (CPL) of the CPU (only 2 levels are used in Linux)
  - ☞ 0 – kernel mode (all privileges)
  - ☞ 3 – user mode, some instructions not permitted, such as in/out, sti/cli, hlt,…
- The information about **segments** is stored in two tables
  - Global Descriptor Table (GDT) for the whole system
  - Local Descriptor Table (LDT) for each process (usually not used in Linux)
- The *descriptors* in those tables have 64 bits, and contain
  - ☞ type of segment (4 bits): code, data, …
  - ☞ linear address where the segment starts (32 bits)
  - ☞ size of the segment (20 bits)
  - ☞ *Descriptor Privilege Level* (DPL, 2 bits), where access is granted iff CPL <= DPL (if DPL=0 the segment can only be accessed in kernel mode)

MEDIATION THROUGH ACCESS CONTROL

Access control

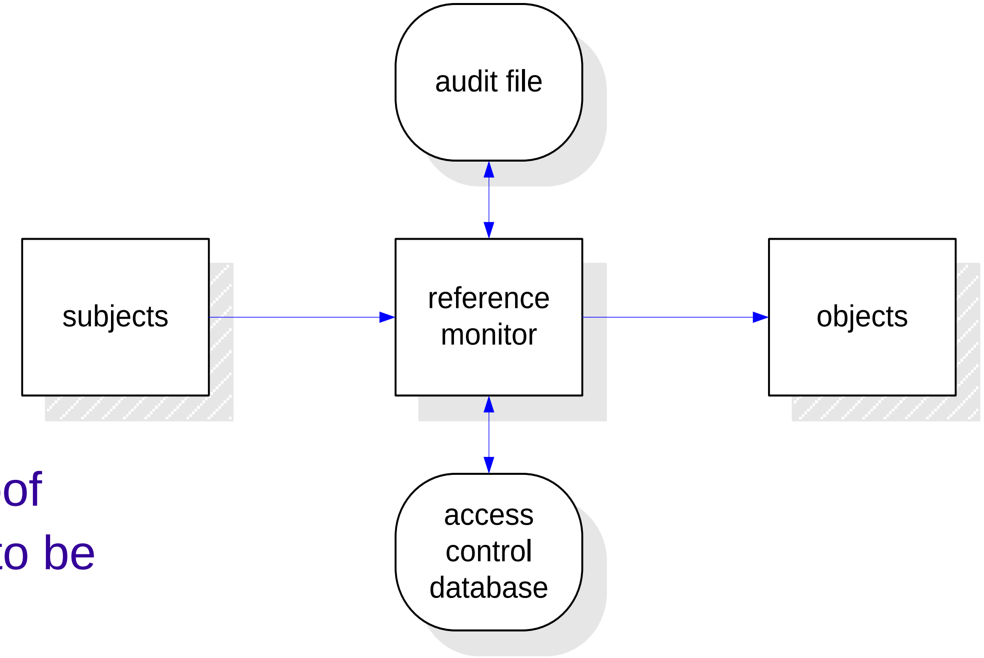
- Objects are accessed by subjects (users, groups, processes)

- After separation, how can we mediate the access to ensure that only the authorized actions are performed in the objects?

  Access control is concerned with the validation of the access rights of subjects to resources of the system

Reference monitor

- Access control should be implemented by a reference monitor
  - ☞ it is an abstract component
- Ensure three principles
  - ☞ *Completeness:* it must be impossible to bypass
  - ☞ *Isolation:* it must be tamperproof
  - ☞ *Verifiability:* it must be shown to be properly implemented
- General purpose OS
  - ☞ Access control is scattered through the kernel…..



Basic access control mechanisms

- Access control lists (ACLs)
  - ☞ each object is associated with a list
  - ☞ the list contains pairs *(subject, rights)*

- Capabilities
  - ☞ each subject has a list of objects that it may access
  - ☞ the list contains capabilities, i.e., pairs *(object, rights)*
  - ☞ capabilities are cryptographically protected against modification and forging

- Access control matrix
  - ☞ a matrix with lines per subject, columns per object, rights in the cells

Basic access control mechanisms

(a) Access control lists (ACLs)
(b) Capabilities
(c) Access control matrix

object 1
(subject 1, read)
...
(subject n, read and write)

(a)

subject 1		
(object 1, read)	(object 2, all)	...

(b)

	object 1	object 2	…	object m
subject 1	read	all	...	--
subject 2	--	--	...	read and write
…	...	...	...	...
subject n	read and write	read	...	read

(c)

Who defines the rights?

- Who defines the access control policy for each object?
  - ☞ Usually each subject sets policy for its objects
  - ☞ E.g., a user for its files, a process for its shared memory objects

- What about administrative operations?
  - ☞ Add/remove users? Execute network services?
  - ☞ The usual solution is to have a special user
    - – Superuser or root in Unix
    - – Administrator in Windows

- Notice: a superuser is not the kernel! The kernel simply allows the processes of this user to have a higher number of rights to access the objects of the machine

Unix access control model (I)

- The user has a username associated to an account
- Each user has a **user id** (UID) and belongs to one or more groups, each with a **group id** (GID)
  - ☞ UID 0 – administrator (root account) with (almost) all rights
  - ☞ early Unix: initial GID=100 (group users);
  - ☞ today: typ. initial GID=UID

- Each <u>object</u> (file, directory, device) has
  - ☞ **owner** UID and GID
  - ☞ **access permissions** rwx (read, write, exec) for owner, group, others (9 bits)

- How can a normal user access privileged resources (e.g., a protected file)?

Example in Ubuntu

- Definition of users: `/etc/passwd`

username

UID

GID

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
ses:x:1000:1000:ses,,,:/home/ses:/bin/bash
< ... >
```

- Owner UID & GID and access permissions of a file

```
drwxrwxr-x 6 ses ses 4096 Dec 18  2013 apps
drwxr-xr-x 2 ses ses 4096 Dec 13  2013 Desktop
drwxr-xr-x 2 ses ses 4096 Dec 13  2013 Documents
drwxr-xr-x 2 ses ses 4096 Dec 18  2013 Downloads
```

Access rights for owner / group / others

owner

group

Unix access control model (II)

- Objects are accessed by processes (i.e. running programs)
  - ☞ the **effective UID** (EUID) and the **effective GID** (EGID) are compared with the object permissions to grant/deny access
    - – the question is asked: Does process with EUID=N1 and EGID=N2 has permission to do action X in this object?
  - ☞ typically **EUID = real UID** and **EGID = real GID** but…
- Two more access bits: **setuid, setgid**
  - ☞ serve to allow access to resources the user cannot access
  - ☞ For example: /etc/passwd must not be modified arbitrarily
    - – it is owned by root
    - – user modifies its entry using a program called *passwd* that must run as root. How? *passwd* has setuid root
    - – this means that when a user runs *passwd* the effective UID (EUID) of the process is 0 ≠ user real UID

Privilege escalation attacks often aim programs with *setuid* and owner UID 0!

Example of Sticky Bits

- See the file:

```
ses@ses-VirtualBox:~/$ ls -l /etc/passwd
-rw-r--r-- 1 root root 1567 2016-09-30 18:38
/etc/passwd
```

- See the program:

```
ses@ses-VirtualBox:~/$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 37132 2016-02-21 00:16
/usr/bin/passwd
```

Unix access control model (III)

- Ideas about applying the *least privilege principle*
  - ☞ Execute privileged operations in the beginning (e.g. *bind* a reserved port) then reduce the privileges (e.g., set EUID = user's real UID)
  - ☞ Divide the software in components and run only minimal components with high privileges
  - ☞ Change the execution environment with *chroot()*
    - – changes the root directory allowing the program to use only files below the new root
    - – hard to put to work since all files (e.g., libs) must be below new root
    - – some programs must use /dev/null, /dev/random,...

Access Control Models: MAC/DAC

- Discretionary A.C. – configured by the user (*traditional solution*)
- Mandatory A.C. - configured by the administrator for all system
- POSIX standard was being extended with more fine-grained privileges: **capabilities** *(careful: not the usual meaning of capabilities) (was, no longer!)*

  ☞ Linux has an implementations based on bitmaps
    - **effective capabilities (E)** : checked when there is an attempt to perform a privileged operation (instead of EUID = 0)
    - **permitted capabilities (P)** : capabilities that a process may use; usually, P=E but the program may remove some capabilities from E
    - Inheritable capabilities (I) : capabilities given to processes created by the current one (fork)

  ☞ Ex: CAP_KILL (send signals), CAP_NET_RAW (use raw sockets)

  ☞ MAC: some capabilities can be discarded until the next reboot, so ***not** even the superuser* can use them (CAP_SYS_MODULE…)

Windows access control model (I)

Main ideas of access model

- **Security IDs** (SID): account SIDs (≈UID of unix), group SIDs (≈GID of unix), computer SIDs
- Access to resources is controlled with DAC and ACL
  - ☞ resources are files, file shares, registry keys, shared memory,...
  - ☞ each ACL contains one or more *Access Control Entries (ACE)*
    - – ACE = SID + permissions
  - ☞ permissions
    - – standard: No access, Read access, Change access, Full control
    - – but can also be access restrictions (e.g., Deny full control)
- Higher granularity than Unix's scheme ...

  ... but very often users run as administrator! (worse than setuid!)

Windows access control model (II)

- Access control is also based in MAC and Capabilities

  ☞ **user accounts** have **Privileges** that allow/disallow operations that apply to all computer, not only to some resources
  - Examples:
  - Backup files and directories - SeBackupPrivilege
  - Restore files and directories – SeRestorePrivilege
  - Act as part of the operating system - SeTcbPrivilege

  ☞ **Token** are like capabilities in the classical sense
  - data structures associated to a user when he/she makes login, and then passed to the (running) process
  - contains *account SID* + *group SIDs* + *privileges*

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 3)