Protection in Operating Systems

Segurança em Software

Pedro Adão, Ana Matos

(and Miguel Correia)

TÉCNICO LISBOA

Livro: Capítulo 2 (v1) / 3 (v2)

Introduction

- Operating systems are a crucial component of computer security
  - So we'll see the basic protection mechanisms they provide
  - And discuss some issues
- Modern OSs support multiprogramming so they must provide:
  - Protection between users and of legitimate users from intruders and malware
  - Protection of the OS itself from users, intruders and malware

Protection of resources

Protection

- A computer contains resources called <u>objects</u>:
  - Memory pages, memory segments
  - I/O devices (disks, networks, printers, monitors)
  - Dynamic libraries (DLLs, .so)
- Objects are accessed by <u>subjects</u>
  - Subjects = users, groups, processes
- Protection role of the OS: to ensure that objects are not accessed by unauthorized subjects
  - Each file can be access only by a set of users
  - Each memory segment can be access only by the process that it is part of

Protection

- To ensure that objects are not accessed by unauthorized subjects
- Two aspects:
  - Separation – prevent arbitrary access (next)
  - Mediation – control access (later)

Separation

Separation in OSs

- Common operating systems (Unix, Windows) run software basically in <u>two modes</u> (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
  - Note: today there's often a 3rd mode – see virtualization class/chapter
- These modes are *enforced by the CPU*
  - Simply disables a set of its instructions in user mode (e.g., in/out, sti/cli, hlt)
  - "Disable" means: generates exception or does nothing if the process tries to execute it, depending on the instruction

Separation in OSs (cont)

- In <u>user mode</u>, software has to call the OS kernel to make privileged operations (e.g., I/O)
  - System calls – sort of functions, but they are in the OS
  - Control the access from user mode programs to all objects outside their memory, including system resources

Separation in OSs (cont)

- In <u>user mode</u>, software has to call the OS kernel to make privileged operations (e.g., I/O)
  - System calls – sort of functions, but they are in the OS
  - Control the access 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 x86)

Memory protection

- *"probably the most fundamental hardware requirement for a secure system is memory protection"* – Gasser
  - Also for reliability
- The problem
  - What prevents a process in user mode from changing the memory of another process or the kernel?
- Implemented by hardware+OS

Forms of separation

- Is the basis for protection; e.g., in an OS:
- *Physical separation:* different processes use different devices (e.g., printers for different levels of security)
- *Temporal separation:* processes with different security requirements are executed at different times
- *Logical separation:* processes operate under the illusion than no other processes exist
- *Cryptographic separation:* processes use cryptography to conceal their data and/or computations in a way that they become unintelligible to other processes

Separation for memory protection

- *Logical separation:* processes operate under the illusion than no other processes exist
- There are several solutions but we are interested in those currently used:
  - Segmentation
  - Paging
  - Segmentation + Paging

Segmentation

Segmentation

- A program is split in pieces with logical unit, **segments**: code, data, stack,…
  - Each one has a name - memory is addressed by: *(name, offset)*
  - Can be relocated in physical memory
  - Can be stored in auxiliary memory (disk)

Segmentation

- A program is split in pieces with logical unit, **segments**: code, data, stack,…
  - Each one has a name - memory is addressed by: *(name, offset)*
  - Can be relocated in physical memory
  - Can be stored in auxiliary memory (disk)
- The OS has a table with the beginning of each segment in memory per process (translates *name* to an address)
  - A process can access a segment only if appears in its *segment translation table – otherwise does not even "see" that segment*
  - Each access passes through the OS so *access rights* can be checked (e.g., no execution of data segments)
  - Info about access rights is stored in the table

Segmentation

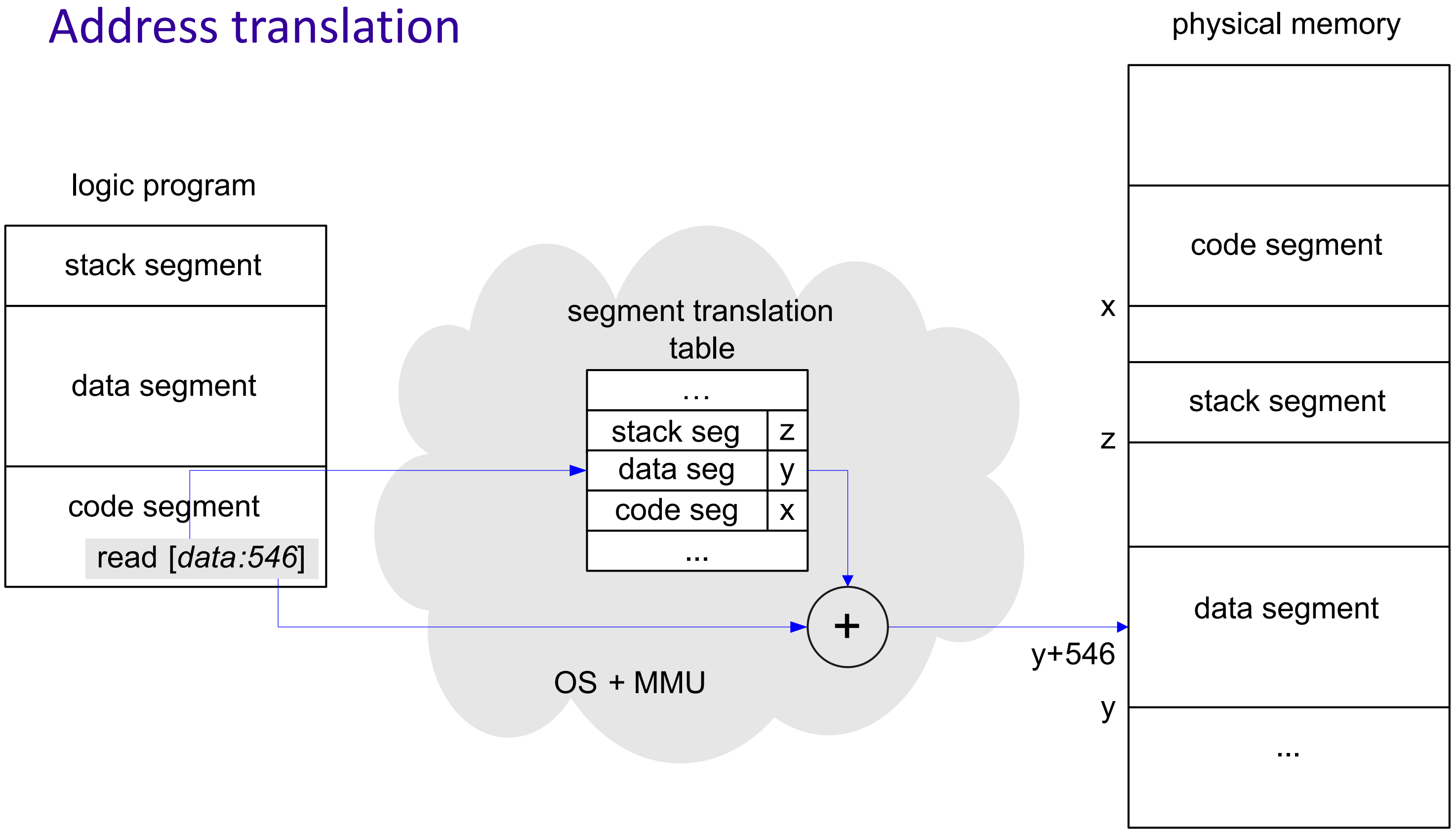
- A program is split in pieces with logical unit, **segments**: code, data, stack,…
  - Each one has a name - memory is addressed by: *(name, offset)*
  - Can be relocated in physical memory
  - Can be stored in auxiliary memory (disk)
- The OS has a table with the beginning of each segment in memory per process (translates *name* to an address)
  - A process can access a segment only if appears in its *segment translation table – otherwise does not even "see" that segment*
  - Each access passes through the OS so *access rights* can be checked (e.g., no execution of data segments)
  - Info about access rights is stored in the table
- **Problems**:
  - Checking the end of the segment efficiently
  - Causes fragmentation of the memory (sizes vary, can grow w/time)

Segmentation

- A program is split in pieces with logical unit, **segments**: code, data, stack,…
  - Each one has a name - memory is addressed by: *(name, offset)*
  - Can be relocated in physical memory
  - Can be stored in auxiliary memory (disk)
- The OS has a table with the beginning of each segment in memory per process (translates *name* to an address)
  - A process can access a segment only if appears in its *segment translation table – otherwise does not even "see" that segment*
  - Each access passes through the OS so *access rights* can be checked (e.g., no execution of data segments)
  - Info about access rights is stored in the table
- **Problems**:
  - Checking the end of the segment efficiently
  - Causes fragmentation of the memory (sizes vary, can grow w/time)

Segmentation (cont)

- Address translation



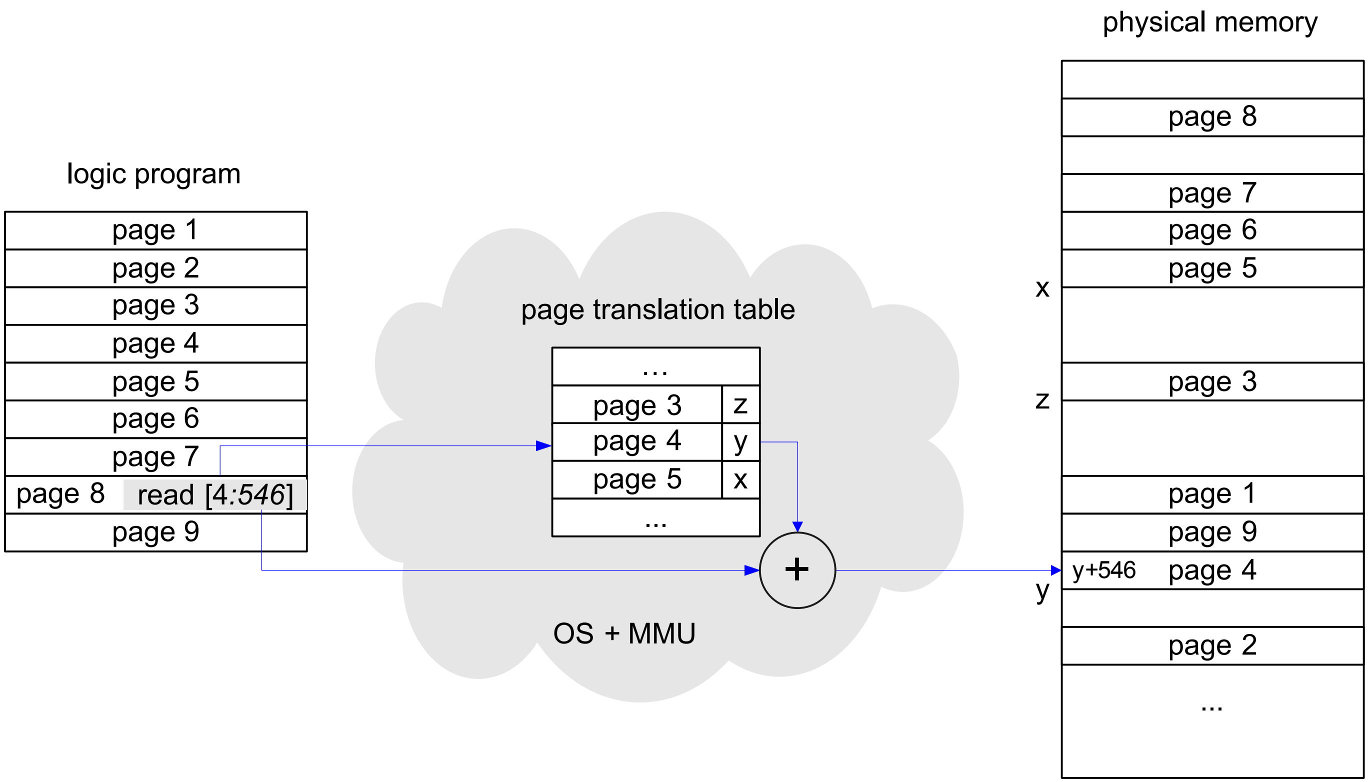
MMU = memory management unit

Paging

- Program is divided in **pages** of the same size (e.g., 4KB, typ. power of 2)
  - Memory is divided in **page frames** of the same size
  - … so there is no fragmentation and knowing the end is trivial
  - Memory 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 page only if it appears in its table
  - Access rights are enforced per access – info about access rights is stored in the table

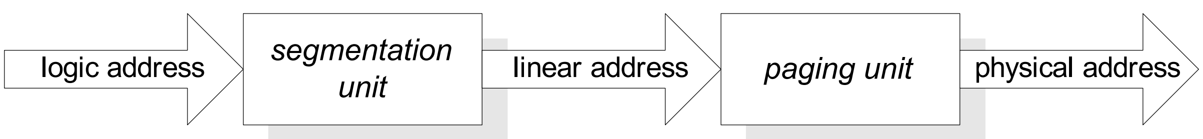
Paging (cont)

- Address translation



Segmentation + paging

- Some architectures support both, e.g., x86
- Linux on x86 uses both
  - Programs use *logic addresses:*
    - **Segment** selector (16 bits), stored in a CPU register (e.g., CS, DS, SS)
    - Offset (32 bits)
  - converted to *linear addresses:*
    - Address of the virtual memory, split in 4KB **pages** (32 bits)
  - converted to *physical addresses*
    - if page is not in RAM, a *page fault* is generated



Linux/x86

- Register CS contains
  - the *<u>current privilege level</u> (CPL)* of the CPU; only 2 in Linux:
  - 0 – kernel mode (all privileges)
  - 3 – user mode – some instructions blocked: in/out and variants, sti/cli, hlt,…
- Info about **<u>segments</u>** is stored in two tables:
  - Global Descriptor Table (GDT); Local Descriptor Table (LDT, not used in Linux)
  - The *descriptors* in those tables contain:
  - *<u>Descriptor Privilege Level</u>* (DPL, 2 bits) - counter-intuitive but max. prim. = 0
  - Access granted iff CPL <= DPL
  - if DPL=0, segment can only be accessed in kernel mode (typ. it's part of the OS)
  - Type (4 bits): access for read, write, execute
- Info about **<u>pages</u>** is stored in page tables; each page has:
  - Read/Write flag: says if page can be read/written
  - User/Supervisor flag: says if can be accessed in user/kernel mode

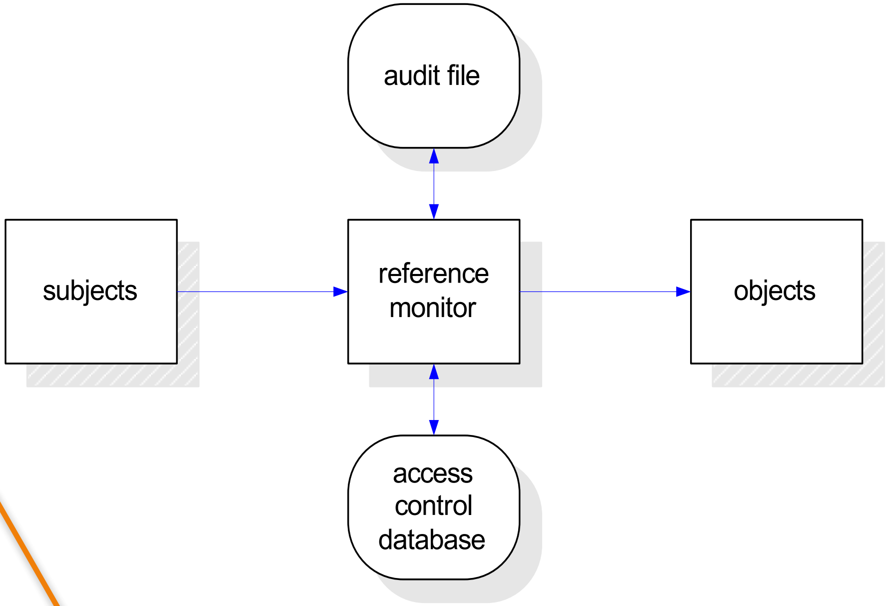
Access control

Access control

- Objects are accessed by subjects (users, groups, processes)
- After the separation, how to mediate the access?
- Access control is concerned with validating the access rights of subjects to resources of the system

Reference monitor

- Access control should be implemented by a reference monitor
  – It's an abstract component
- 3 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 controls what?

- 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

Who controls what?

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

Unix access control model (I)

- 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), (almost) all rights

Unix access control model (I)

- 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), (almost) all rights
- Each object (file, directory, device) has:
  - **Owner** UID and GID
  - **Access permissions** rwx (read, write, exec) for owner, group, world (9 bits)

Unix access control model (I)

- 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), (almost) all rights
- Each <u>object</u> (file, directory, device) has:
  - **Owner** UID and GID
  - **Access permissions** rwx (read, write, exec) for owner, group, world (9 bits)
- 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
  - i.e., the question asked is: does process with EUID=N1 and EGID=N2 has permission to do action X?
  - Typically EUID = real UID and EGID = real GID but…

Unix access control model (II)

- Two more access bits: **setuid, setgid**
  - Important security-wise
  - Aim to allow access to resources the user cannot access
- Ex.: /etc/passwd must not be modified arbitrarily so:
  - It is owned by root
  - User modifies its entry using a program called *setpasswd* that must run as root. How? *setpasswd* has <u>setuid root</u>
  - This means that when a user runs *setpasswd* the <u>effective UID</u> (EUID) of the process is 0 ≠ user UID
- Privilege escalation attacks in Unix often aim programs with *setuid* and owner UID 0…

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 using *seteuid* or *setegid*
  - Divide the software in components and run only minimal components with high privileges
  - 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; e.g., some programs must use /dev/null, /dev/random,...
  - Use capabilities instead (next slide)

Linux capabilities

- POSIX standard includes more <u>fine-grained privileges</u> that it calls **capabilities** *(careful: not the usual meaning of capabilities)*
- Linux now implements these capabilities
  - Applications do not need to run with EUID=0 but only with the required capabilities
  - Examples:

Capability Name	**Meaning**
CAP_KILL	Allow sending signals to processes belonging to others
CAP_SETUID	Allow changing of the UID
CAP_NET_BIND_SERVICE	Allow binding to ports below 1024
CAP_NET_RAW	Allow use of raw sockets
CAP_SYS_MODULE	Allow inserting modules in the kernel
...	...

Mandatory/Discretionary Access Cont.

- Question is who defines the access control policy for objects
- **Discretionary A.C. (DAC)** – access policy defined by the user
  - the one we saw
- **Mandatory A.C. (MAC)** – access policy defined by an administrator
  - Capabilities allow doing MAC in Linux
    - Some capabilities can be discarded until the next reboot, so not *even the superuser* can use them (ex, CAP_SYS_MODULE…)
  - SELinux also implements MAC in Linux (including Android)

Windows access control model (I)

- (Windows NT, 2000, XP)
- **Security IDs** (SIDs): account SIDs (≈UIDs), group SIDs (≈GIDs), computer SIDs
- Access to <u>resources</u> is controlled by **Access Control Lists** (ACLs)
  - Resources: files, file shares, registry keys, shared memory,...
  - Each ACL contains one or more *Access Control Entries (ACEs)*
    - ACE = account SID (≈UID) + permissions -- not only for owner, group, world
  - 4 standard permissions: No access; Read access, Change access, Full control
  - Higher granularity than Unix's scheme...
  - ...but very often users run as administrator! (worse than setuid!)

Windows access control model (II)

- Windows has a kind of Mandatory Access Control
  - User accounts have **privileges** that allow/disallow operations that apply to all the 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
  - Data structures associated to a (running) process
  - They are **capabilities** in the classical sense
  - Contains SIDs (at least the user’s SID) and privileges

Summary

- Mechanism: Access control, e.g. ACLs
- Problem: What about administration operations?
  - Solution: Privileged account (e.g., root)
- Problem: Privileged account has too many privileges
  - Solution: MAC and fine-grained privileges (e.g. SELinux)

Summary

- Resource protection
  - CPU operation modes
  - Memory protection
- Access control
  - Access control in Unix, Windows
  - MAC vs DAC