CSCI 7000-001 — Protections Against Malicious Code

November 15, 2001

1 Outline

- development-level controls
- deployment-level controls
- run-time controls

2 Development-level controls

Some software engineering methods can be used to detect malicious code. Such methods include:

- testing
- code reviews
- program analysis
- model checking and formal verification
- software process management
- configuration management

Development-level controls are effective against programmed-in malicious code. Typically:

- trapdoors
- trojan horses
- logic bombs

3 Deployment-level controls

Activities such as installation and update are critical for the integrity of fielded software systems. In particular, a secure installation/update process should:

- verify the authenticity and integrity of packages. Packages should be signed by trusted providers
- test new pieces of software in confined (safe) environment

Deployment-level controls are effective against malicious code inserted by tampering with software packages. These include:

- some forms of viruses
- trapdoors
- trojan horses
- logic bombs

4 Run-time controls

The protection mechanisms found in general-purpose operating systems offer good protection against some of the threats of malicious code. The level of protection depends on the security policy model implemented by the operating system.
The term "policy model" here refers to the usual relation between *subjects* and *objects*. *subjects* are users or more specifically user processes, while *objects* are entities managed by the operating system such as:

- memory
- I/O devices such as a disk
- sharable programs and subprocedures
- sharable data
- CPU usage

5 Principle of *separation*

One of the main services provided by an operating system consists in keeping users' processes *isolated* from one another. In general, an operating system may manage *separation* at different levels:

- *logical separation*, in which users' processes operate *as if* they where in separate boxes, where in fact the operating system is simply switching *context*, and the processor is guaranteeing access control within a context.
- *temporal separation*: processes with different security requirements or different security capabilities are executed at different times (e.g., batch systems)
- *cryptographic separation*, in which processes conceal their data using cryptographic techniques.

6 Controlled access to shared resources

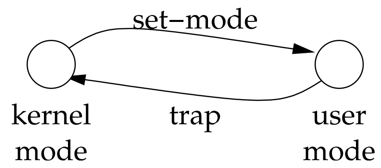
The other main functionality of an operating system is to control access to shared resources and objects. In doing so, the operating system:

- assures inter-leaving consistency. In essence it implements serializable access to shared resources, and
- implements some access control policy. Typical access control policy are based on:
  - user/profile and simple permission masks
  - access control list
  - capability list
  - quotas

7 Kernel and user modes

An essential requirement to implement system controls (through the operating system) is some form of hardware support for at least two modes of execution:

- *monitor-mode* or *kernel-mode*: executing code has access to reserved instructions that set special registers and tables
- *user-mode*: the CPU allows execution of non-privileged instructions only.



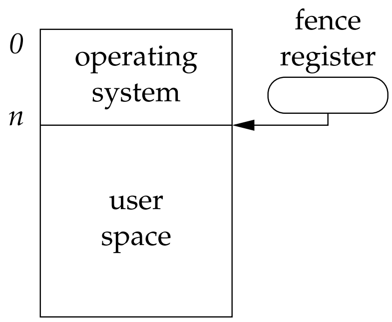
A special opcode switches to user-mode.
Interrupts are used to switch to *kernel-mode*. In response to such signals, the CPU jumps to predefined (and obviously protected) interrupt-specific procedures.

8 Issues in protecting memory

- potential performance bottleneck
- needs special hardware support

9 Primitive memory protections: fence

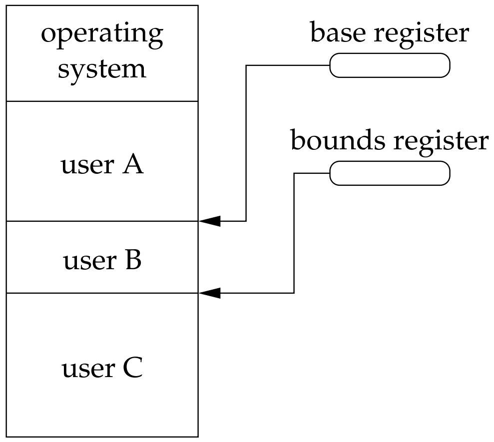
Remember the Commodore64. . . :



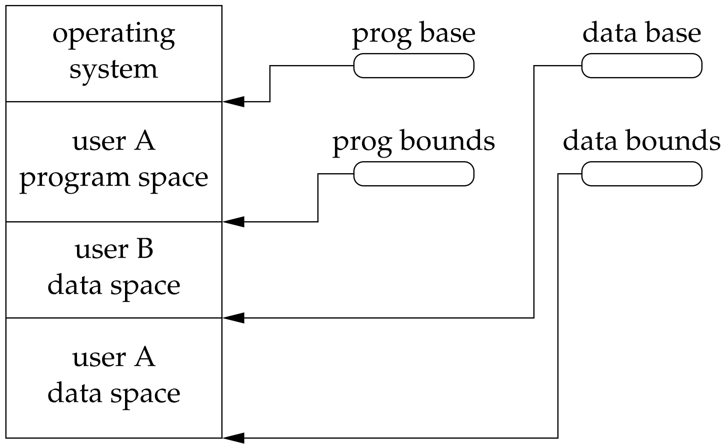
- the fence protects "low" (OS) from "high" (user)
- no multi-user protection
- no distinction between programs and data in user space
- coarse-grained access control: no distinction within one protected block

10 Base/bounds relocation and context-switch

- programs are relocatable
- operating system relocates programs
- operating system implements a *context switch* to set base and bounds registers



- still no distinction between programs and data in user space
- still coarse-grained access control



- still coarse-grained access control

11 Tagged architecture

In order to implement a finer-grained control over memory access, a few specialized machine implemented an alternative *tagged architecture*

memory word	tag
...	R
...	RW
...	X
...	...
...	...
...	...

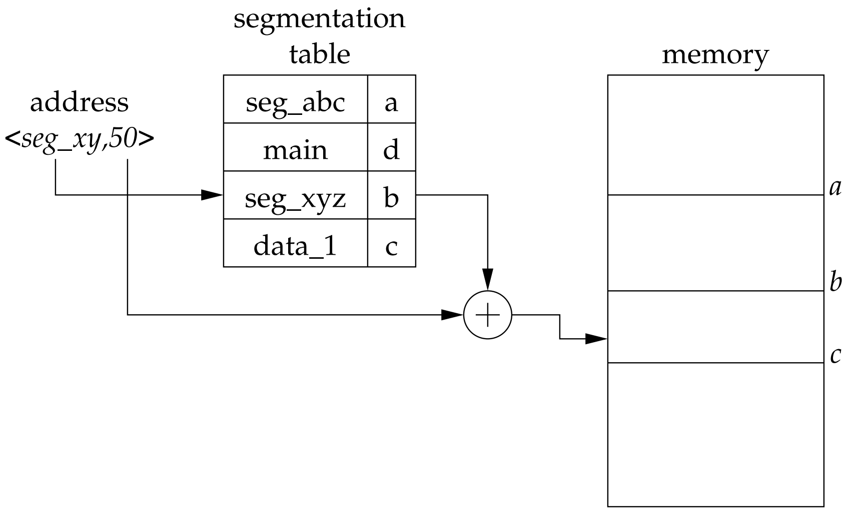
Tags can also distinguish *data types*, and therefore perform checks on the specific operation applied to a given memory word. For example, you can distinguish data (such as integers and characters) from pointers, and therefore not allow a dereference operation on a data value.
Obviously, tagged architectures suffer from a series of problems:

- overhead
- not enough support from available OS

12 Segmentation

Another protection technique that does not require much specialized hardware is *segmentation*.

- memory is divided into segments
- addresses are pairs (segment_id, offset)
- the OS maintains per-process tables of actual addresses

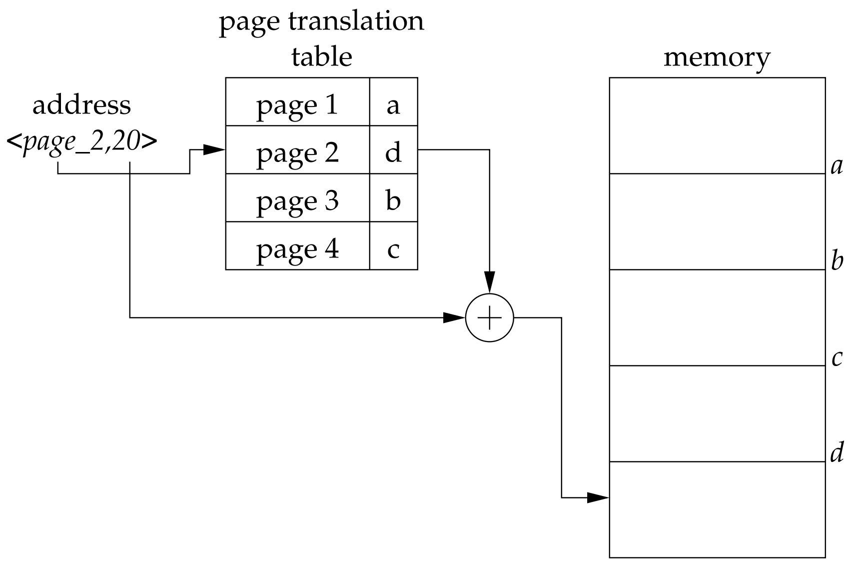


Comments:

+ segments can easily grow and shrink, allowing good use of memory for dynamic data structures.
+ two processes may refer to the same physical segment, therefore sharing memory. This is particularly useful to share executables and libraries.
− segmentation leads to fragmentation, and therefore poor memory utilization.

13 Paging

Paging is an alternative method of partitioning memory. A *page* is a fixed-size memory block. More specifically, a *page* in *virtual memory* is a logical block of data or program, which corresponds to a block of equal size in physical memory called *page frame*.



+ Paging is a better (simpler and faster) technique to implement *virtual memory* than segmentation, however paging looses some of the benefits of segmentation

– pages do not define any type of logical perimeter, therefore they can not be used to define access-control boundaries.