System Security: Introduction
TDDE62

Ulf Kargén

Department of Computer and Information Science
Linköping University

LiU LINKÖPING UNIVERSITY

System Security

- Security in computer systems
  - Hardware
  - (System) software

- Today's agenda:
  - Hardware basics (recap)
  - Operating System (OS) access control concepts and fundamentals
  - Memory protection
  - OS interface (system calls, etc.)
  - Shortcomings of traditional OS and hardware security
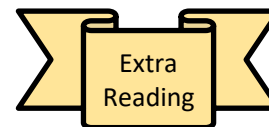
Extra Reading

Some slides are marked as "extra reading". These are not mandatory to read for the exam, but will provide more in-depth understanding and fill in some "blanks".

LINKÖPING UNIVERSITY

Hardware Basics: The Memory Hierarchy<superscript>3</superscript>
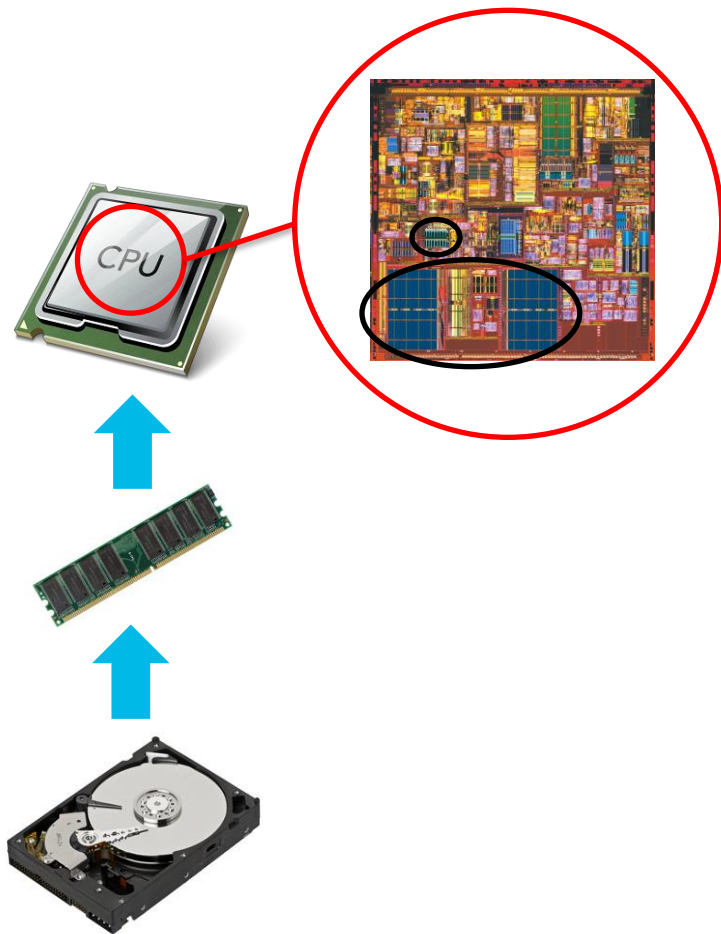
- Memory in computers are organized in a hierarchy
  - Large but slow at the bottom
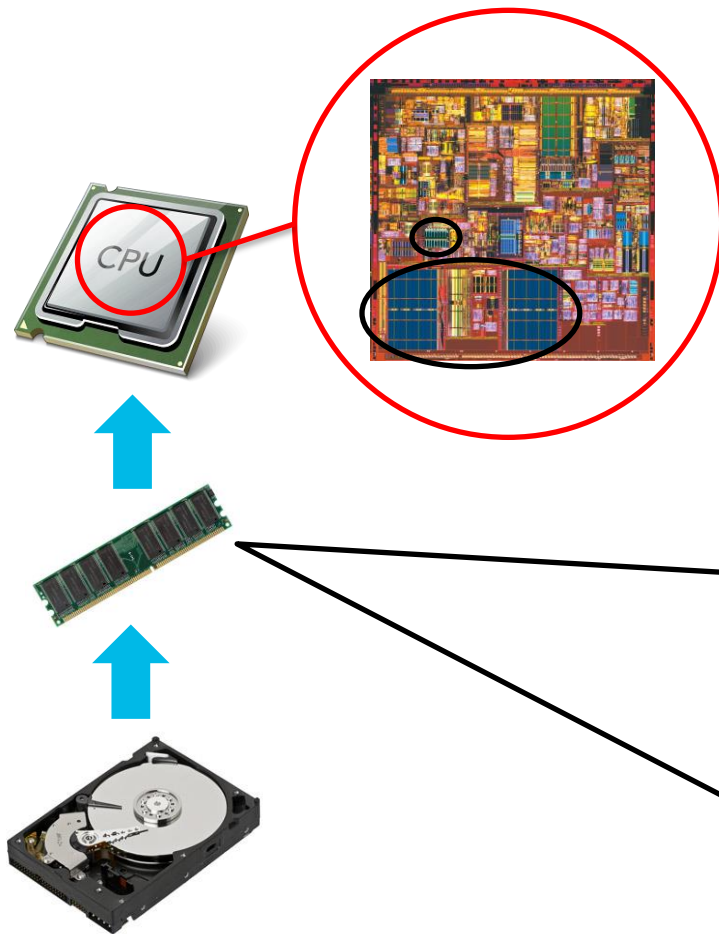 - Small but extremely fast at the top

Hardware Basics: The Memory Hierarchy
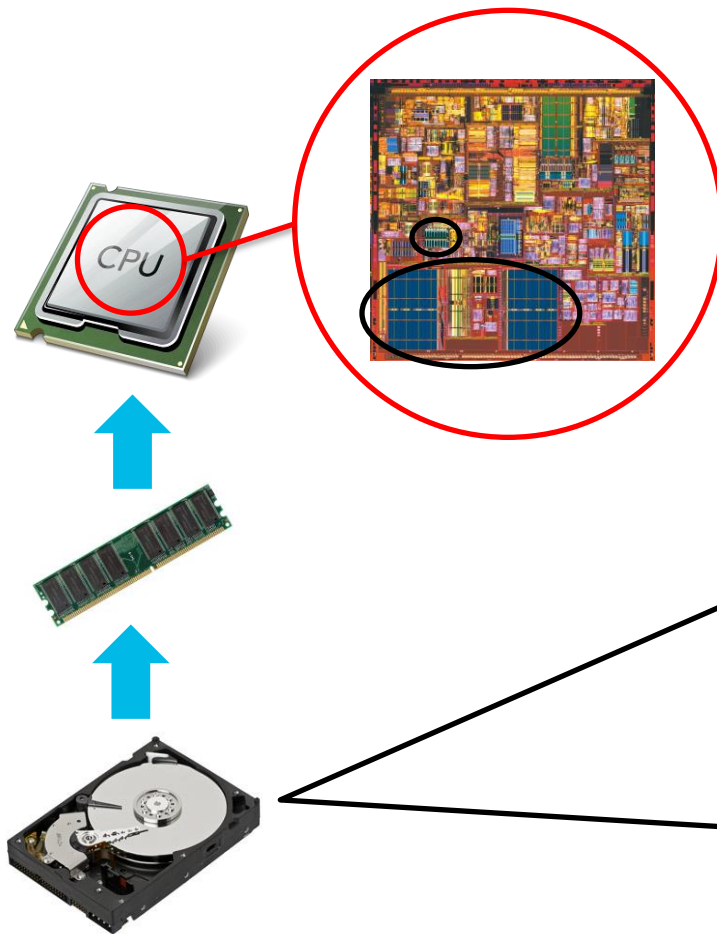
Hard disk drive (HDD) or solid state drive (SSD)

- Size: ~100 – 10 000 GB
- Access time: milliseconds

- Persistent storage after poweroff
- Stores program code and data

# Hardware Basics: The Memory Hierarchy

Random access memory (RAM)

- Size: ~10 GB
- Access time: ~100 nanoseconds

- Too slow to operate on data or code directly from HDD/SSD
- Data and code must be loaded into RAM before it can be used
- RAM is a limited resource – Must take care to only load data that is actually needed at the moment (more on this later).

- Not persistent – data is lost on power off

# Hardware Basics: The Memory Hierarchy [6]
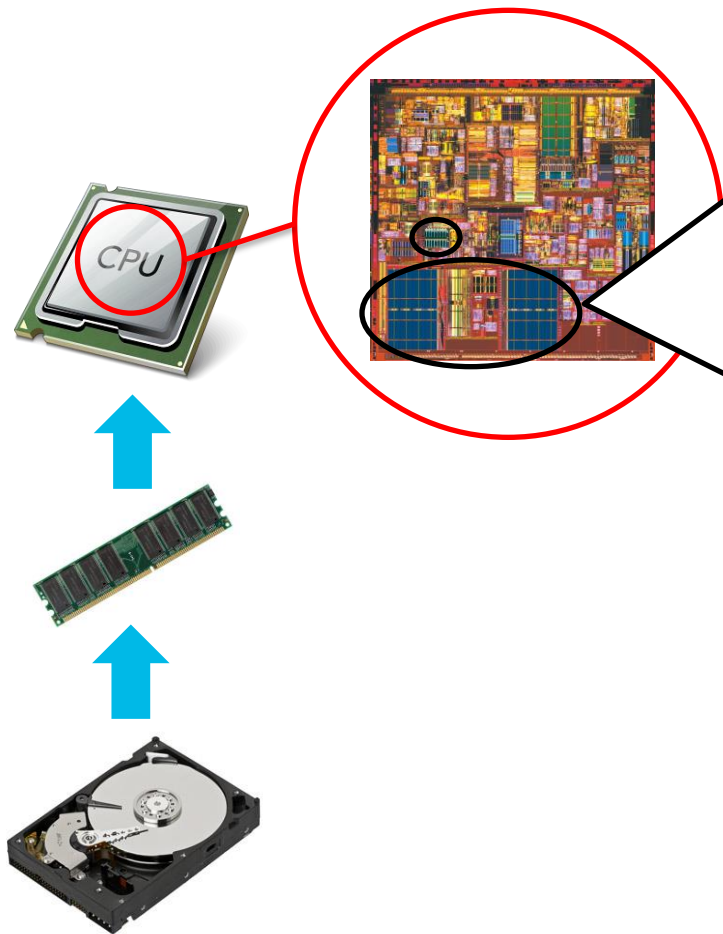
Cache memory
- Size: A few megabytes
- Access time: ~10 nanoseconds

- Modern CPUs can do billions of operations per second
  - RAM speed is still a bottleneck!
- Keep *frequently used* parts of RAM in a cache inside CPU
- CPU transparently handles fetching/updating data in cache, and write-back to RAM
  - Code always reference memory through address in RAM

Hardware Basics: The Memory Hierarchy
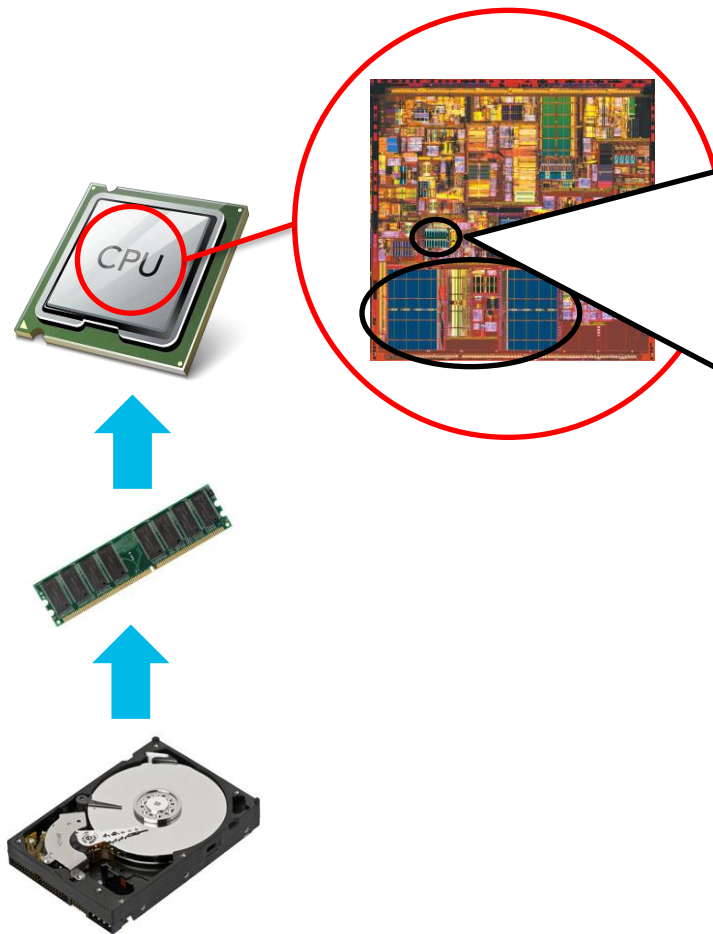
Registers
- Size: A few bytes
- Access time: < 1 nanosecond (same as CPU itself)

- Registers are used to hold data while computations are performed on it in CPU
- Typically:
  Read from memory address *A* into register *R*
  Do operation on data in *R*
  Write contents of *R* back to address *A*

LiU LINKÖPING UNIVERSITY

Hardware Basics: The Memory Hierarchy
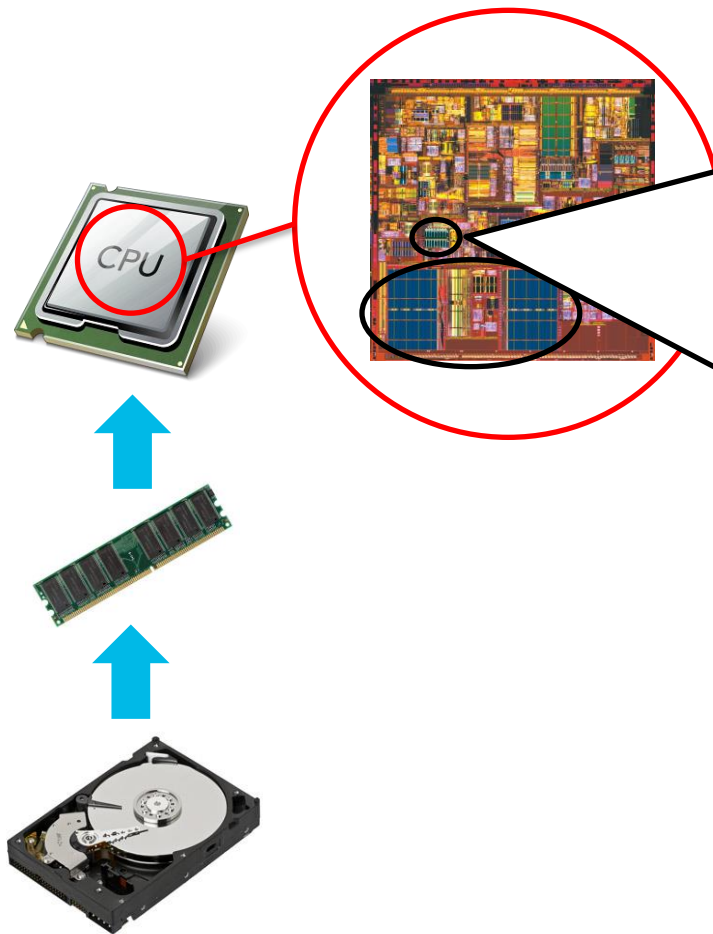
Registers
- Value of registers define the current *state* of the CPU
- All CPUs have some registers with special purposes
- Most important is the *instruction pointer* (IP) register
  - Always points at the address in memory where the currently executing machine code instruction is located

LINKÖPING UNIVERSITY

Hardware Basics: Architecture
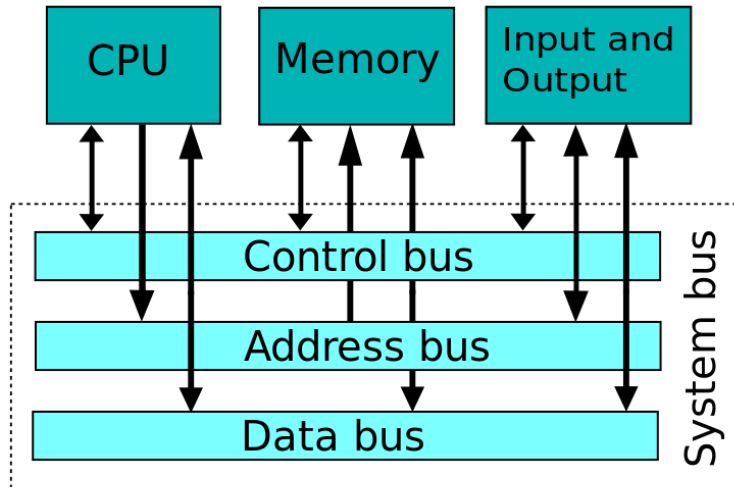
Source:
https://commons.wikimedia.org/wiki/File:Computer_system_bus.svg#/media/File:Computer_system_bus.svg

- The CPU communicates with the RAM and other hardware using several *buses* (essentially electrical wires)

- Modern computers typically have several different buses for memory, graphics cards, hard drives, input devices, etc.

LINKÖPING
UNIVERSITY

Hardware Basics: Booting

Typical startup sequence of a PC:

1. BIOS code executes.

   – BIOS (Basic Input Output System) is a program stored on a read-only memory chip (ROM) on the motherboard.

   – Does some hardware checks, and loads the *boot loader* code into RAM.

   – Boot loader is stored at a fixed location on hard drive.

2. BIOS transfers execution to boot loader.

   – Boot loader in turn loads the operating system *kernel* into memory, and transfers execution to the kernel.

LINKÖPING UNIVERSITY

Hardware Basics: Booting

3. OS kernel loads *device drivers* – code to handle communication with hardware – and initializes the hardware, filesystems, networking, etc.

   – The kernel is the central piece of system software in a computer

   – Responsible for managing hardware, enforcing access control, mediating programs' access to resources, etc.

4. Kernel loads programs to handle user interaction, login, graphical interfaces, etc.

5. The computer is ready for use

LINKÖPING UNIVERSITY

OS Access Control: Processes

The *process* is the basic entity to which access control is applied in operating systems

- Acts as a "proxy" on behalf of e.g. a "real life" user

- Whenever a process attempts to access a resource, e.g. a file, the OS checks if the user ID associated with the process is allowed to access the resource.

- A process can start other processes, which inherits the user ID of their parent process.

- OSes typically have a concept of a "superuser", which has access to "everything"

  - **root** user (Unix)

  - **SYSTEM** and **Administrator** users (Windows)

- The superuser account is necessary for maintenance and configuration of the system, and for running important system-level services

OS Access Control: Login

Process creation at login proceeds (conceptually) as follows:

1. When the kernel has finished initialization, it will launch a login process, running as superuser, e.g. root (Unix) or SYSTEM (Windows)

   – Can be a GUI login screen in a graphical environment, or simply a login prompt in a text-based interface

2. The user provides some *credentials* e.g. username and password

   – Since the login process runs as superuser, it has access to the list of users on the system, and their credentials.

3. Login process checks credentials, and if they match, it will launch a *shell process* running as the authenticated user

   – Uses a special OS mechanism (only available to superuser) to switch the user ID of the launched shell from superuser to the given "regular" user

4. The shell process has a user interface for launching *programs*. Each time a program is launched, a new process is started, running with the user ID of the logged in user

   – Shell can be a desktop environment in a GUI system, or simply a command interpreter in a text-based (non-GUI) system

LINKÖPING UNIVERSITY

OS Access Control: Process Tree Example[14]

root

User Name ▭
Password ▭
Login Cancel

Alice

Bob

Alice   Alice   Alice   Alice

Bob   Bob   Bob

Bob   Bob   Bob

Tab 1   Tab 2   Tab 3
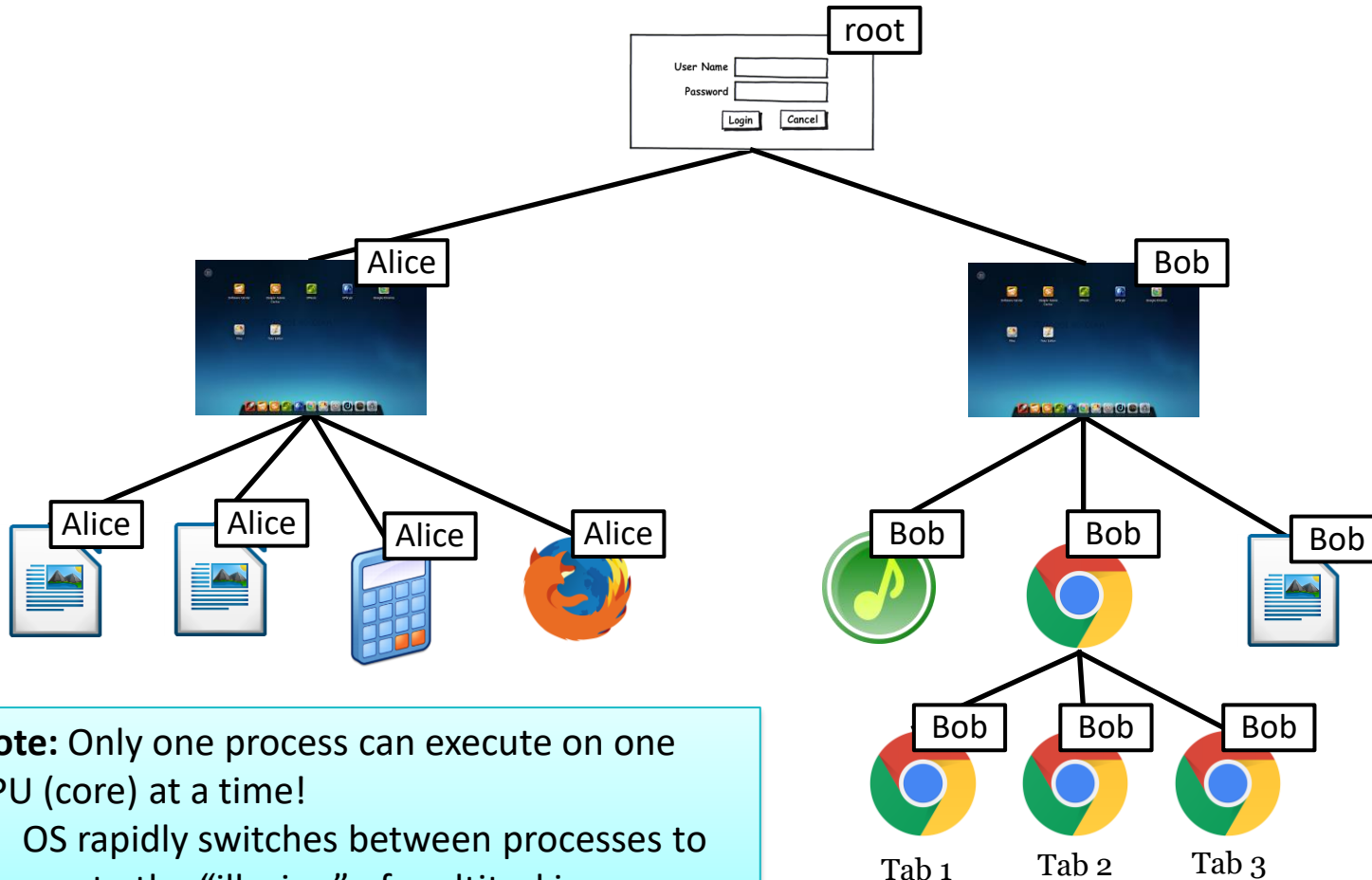
Note: Only one process can execute on one CPU (core) at a time!
- OS rapidly switches between processes to create the "illusion" of multitasking
- Details later…

OS Access Control: Access Control Models

OS needs some model to assign access permissions to resources, e.g. files

- All general-purpose OSes support Discretionary Access Control (DAC)
  - Users can assign permission at *their own discretion*
  - **Example:** Alice marks some files as readable and writable by herself, and read-only for everyone else
  - The Unix read/write/execute permissions for owner/group/others is a well-known example of a DAC model

LINKÖPING UNIVERSITY

OS Access Control: Access Control Models

More advanced/powerful models:

- Mandatory Access Control (MAC)

  - Apart from DAC rules, access is also restricted according to system-wide *security policy*

  - **Example policy:** *Files in users' home directories should **never** be writable by anyone other than the owner and the root user* (overrides DAC rules set by users)

- Role-Based Access Control (RBAC)

  - Access determined by the *role* of a subject (one subject can have several roles) **Example:**

    1. The manager role has access to personnel files, but not to web server configuration.

    2. Webmaster role has access to web configuration, but not to personnel files

    But: If manager and webmaster is the same person, he/she has access to both

LINKÖPING UNIVERSITY

OS Access Control: Enforcement

Central question: How are access controls actually enforced in an OS?

- What if processes have direct (physical) access to the hard drive?

  ⇨ Processes can request data at arbitrary positions on the hard drive

  ⇨ Nothing prevents one of Alice's processes from accessing files created by Bob!

  ⇨ No way for OS to apply e.g. DAC!

LINKÖPING
UNIVERSITY

OS Access Control: Enforcement

- What if all processes share the physical memory (RAM)?

  ⇨ Processes must "play nice", e.g. not overwriting other processes data or code, or using up all memory for themselves

  ⇨ Nothing prevents one of Alice's processes from accessing data loaded by one of Bob's processes

  ⇨ Even if file-system access can somehow be restricted, Alice can still read Bob's confidential data once it is loaded into memory!

- There are also serious performance issues with this approach

  - What if Bob starts a bunch of memory hungry processes, and then goes home for the weekend.

  - Now there is no memory left for Alice to launch a process, even if Bob's processes aren't actually doing anything!

OS Access Control: Enforcement

Conclusions:

- Each process should only have access to its own code/data

- Functionality of system should not depend on processes "playing nice"

  ⇨ Memory sharing should be handled transparently by kernel

  ⇨ Processes should not "see" the memory layout of other processes

  ⇨ Requires strong isolation between processes' memory space!

- Processes must not be allowed to access hardware (e.g. hard drive) directly

  ⇨ Process must ask OS kernel to access hardware on its behalf. Kernel checks if access is allowed before proceeding.

  ⇨ Need a robust interface for this!

Memory Protection: Virtual Memory

Idea: Every process has its own *virtual memory* (VM)

- From the point of view of one process, *all* memory is available to it

    ⇨ Only sees its own memory

- Kernel keeps an internal map of memory regions that are actually used, and which parts of the physical RAM they map to

Memory Protection: Virtual Memory

Details:

- On e.g. a 32-bit CPU, each process "sees" a contiguous virtual memory space of 2^{32} bytes, *only containing its own data and code*

- An address in VM is simply an integer between 0 and $2^{32}-1$, specifying an offset in this memory space

- Virtual memory is divided into *pages* (commonly 4 kB in size).

  - The kernel has a mapping (one per process) between pages of the process, and so-called *frames* of corresponding size in physical RAM

  - This mapping is called a *page table*

  - Each page table entry also specifies if the memory in the page should be readable/writable/executable

LINKÖPING UNIVERSITY

Memory Protection: Virtual Memory

More details:
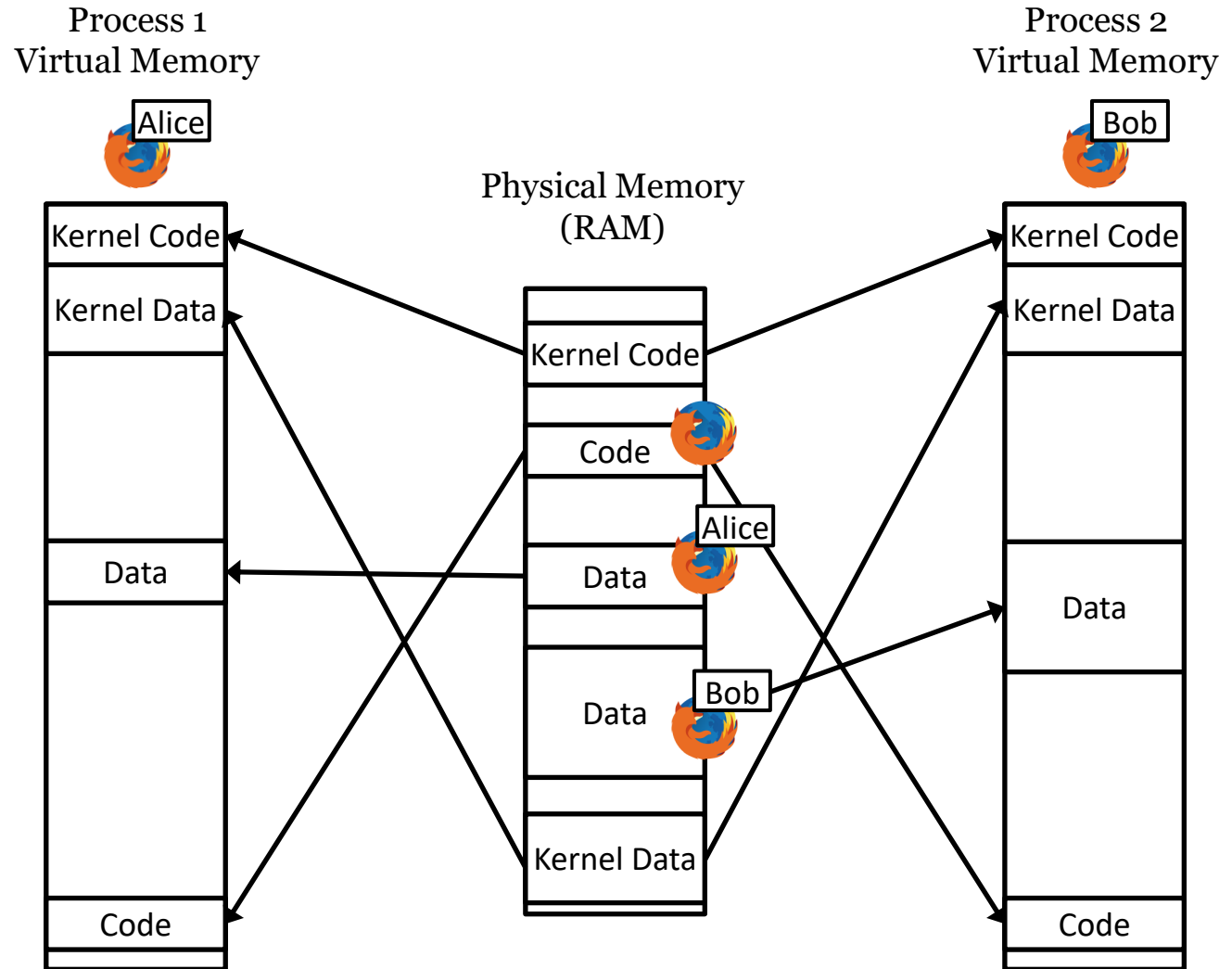
- Every time the active process accesses memory, the virtual address is translated to the corresponding physical address using the active page table

    – Permissions are also checked during translation.

    – If process e.g. tries to write to read-only page, or page that is not mapped $\rightarrow$ Segmentation fault (i.e. OS is notified and kills program – program "crashes")

    – CPU has built-in hardware for translation/checking. Incurs almost no overhead. Hardware for translation is called **Memory Management Unit** (MMU)

- Note that one physical frame can map to several pages in potentially several different processes

    – Allows sharing of e.g. read-only code (programs) between several processes

        ⇨   More efficient use of memory

    –   The kernel code and data is e.g. mapped into the same location in VM of every process. (More on why this is so later.)

LINKÖPING UNIVERSITY

Memory Protection: VM Example

Memory Protection: VM Example

Process 1
Virtual Memory

Process 2
Virtual Memory

Alice

Bob

Physical Memory
(RAM)

Data pages
are private to
each process

| Kernel Code |
| Kernel Data |

| Kernel Code |
| Kernel Data |

| Kernel Code |

| Code |

| Data |

Alice

| Data |

| Data |

| Data |

Bob

| Data |

| Kernel Data |

| Code |

| Code |

Memory Protection: VM Example

Process 1
Virtual Memory

Process 2
Virtual Memory

Alice

Bob

Physical Memory
(RAM)
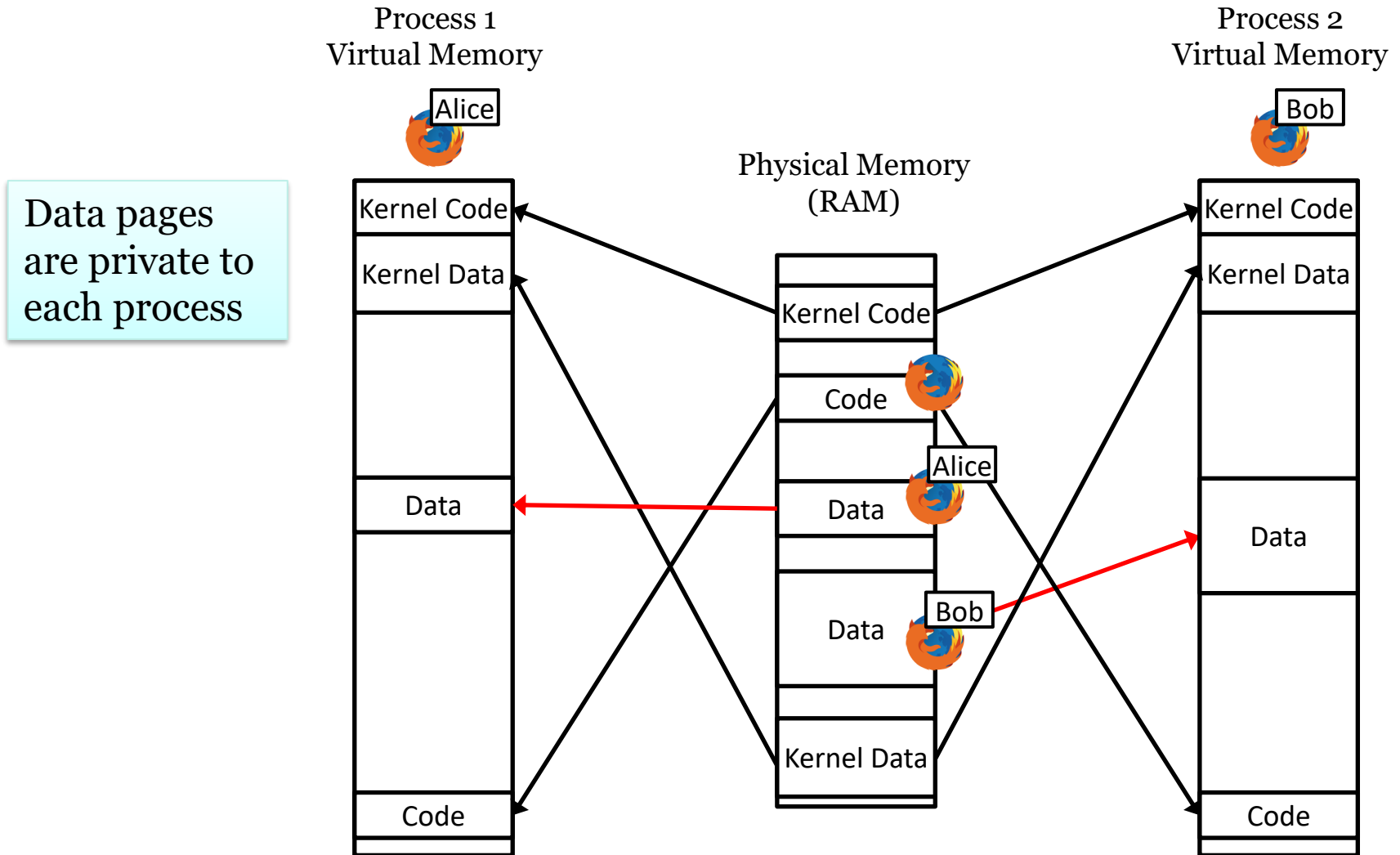
Code pages
are read-only,
and can be
shared
between
processes

| Kernel Code |
| Kernel Data |
| |
| |
| Data |
| |
| Code |

| Kernel Code |
| |
| Code |
| |
| Data |
| |
| Data |
| |
| Kernel Data |
| |

| Kernel Code |
| Kernel Data |
| |
| |
| Data |
| |
| Code |

LINKÖPING
UNIVERSITY

Memory Protection: VM Example

Process 1
Virtual Memory

Alice

Physical Memory
(RAM)

Process 2
Virtual Memory

Bob
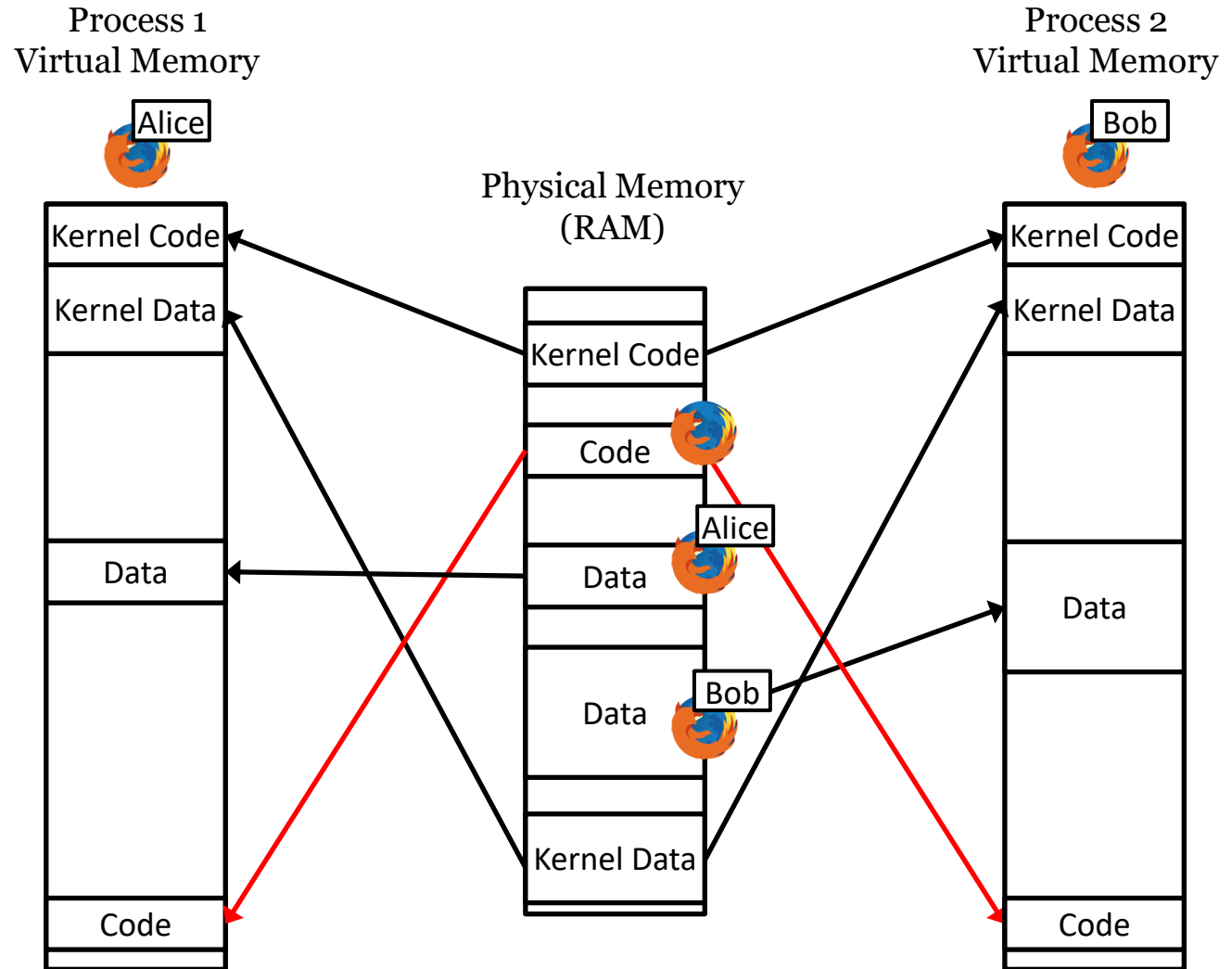
Kernel code and data pages are mapped in to same location in every process VM

(We'll see why soon)

Process 1 VM	Physical Memory	Process 2 VM
Kernel Code		Kernel Code
Kernel Data	Kernel Code	Kernel Data
	Code	
	Alice	
Data	Data	Data
	Bob	
	Data	
	Kernel Data	
Code		Code

LINKÖPING
UNIVERSITY
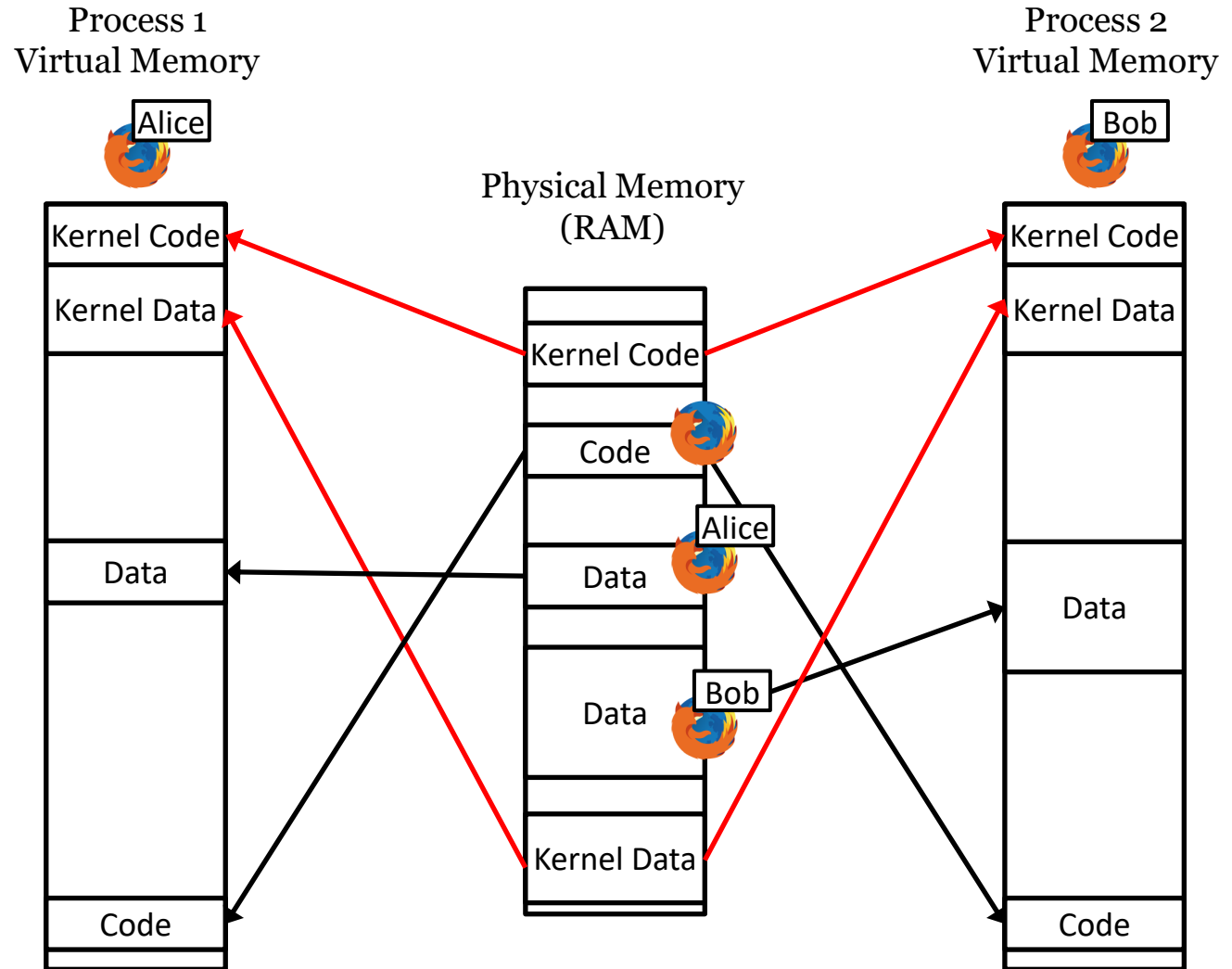
Memory Protection: Swapping

- The virtual memory space may be (much) larger than the actual size of RAM

    ⇨ Only keep pages that are *actively used at the moment* in memory

    ⇨ If low on RAM – move unused pages to hard drive until needed again. This is called *swapping*

- Page table entries also have a field for indicating if a page is in RAM or on hard drive

    – Checked during translation

    – If requested memory is in swapped-out page, load page into RAM before proceeding – may cause other page in same or different process to be swapped out

- OS uses some algorithm to decide which pages to swap out

    – Details not so relevant for this course

LINKÖPING UNIVERSITY

Kernel Interface

We covered how the memory of processes are separated using virtual memory

- But how does a process request (allocate) virtual memory?

- How does a process open a file for reading into memory?

Kernel Interface: CPU Modes

Basic design idea:

- CPU can run in two *modes*: *kernel mode* and *user mode*

  - In **kernel mode**, CPU can execute all possible instructions, e.g. those directly interfacing with hardware, physical memory, etc.

  - In **user mode**, only a restricted subset of instructions is available.

    ⇨ Trying to execute a privileged instruction while CPU is in user mode results in an access violation – process crashes

- As one might guess, kernel code runs in kernel mode and regular processes runs in user mode

  ⇨ How to switch modes in a secure fashion?

> Side note: Many CPUs have more than two modes
> - x86 CPUs e.g. have four (Ring 0-3)
> - Uncommon for general-purpose OSes to use more than two due to portability issues

LiU LINKÖPING UNIVERSITY

Kernel Interface: System Calls

⇨   How to switch modes in a secure fashion?

Idea:

- CPU has special mechanism which *automatically*:
    - Switches CPU to kernel mode
    - Moves execution to *predetermined* kernel code
    - Saves registers (i.e. CPU state)
- This mechanism is used to implement *system calls*
    - The main interface between user-mode processes and kernel code

> This is set up when OS boots

LINKÖPING UNIVERSITY

Kernel Interface: System Calls

When process needs access to hardware, e.g. file system on hard drive, it performs a system call in the following way:

1. Process puts parameters in registers (these are like arguments to a function call)

2. One parameter (a number) specifies the particular system call to execute – determines how rest of parameters are interpreted

3. Process executes a kernel-switch instruction

4. CPU mode is switched and predetermined system-call handling code in kernel is executed

5. Kernel code saves registers (i.e., caller state)

6. Kernel inspects parameters, checks if process is allowed to perform the specific action, and performs the action if so

7. Kernel code restores state (registers) of calling process and executes a special return instruction which:

    - Switches CPU back to user mode

    - Resumes process execution where it left off

Kernel Interface: System Calls

Alice

| Kernel Code |
| Kernel Data |
| |
| Data |
| |
| Code |
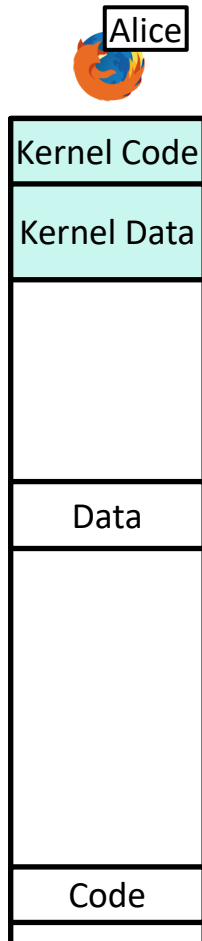
- Remember that kernel code/data is mapped in at fixed location in every process?

  - This is to make system call handling (among other things) easier

    ⇨ Kernel code always at same location when invoking system call handler

- But what if a process attempts to move execution to kernel code, or read kernel data?

  - Page table entries also have a flag specifying if page should be accessible only in kernel mode

  - If a user-mode process tries to use kernel memory it will crash

    ⇨ Assures proper isolation between kernel and processes

LINKÖPING UNIVERSITY

Kernel Interface: System Calls

- System calls is the main mechanism used by processes to interface with the kernel

- Modern OSes have several hundred different system calls

  - Opening/reading/writing files

  - Setting file permissions

  - Opening/reading from/writing to network sockets

  - Allocating/requesting more virtual memory

  - Starting/stopping (child) processes

  - Switching process ownership/permissions (subject to restrictions)

 - etc. ...

Side Note: Interrupts
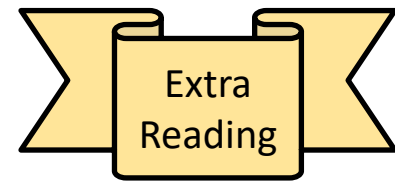
Up until this point we've been a bit handwavy with a few things, specifically:

- How can processes and the kernel share the CPU?

- What actually happens when a program crashes due to some access violation (e.g. accessing unmapped virtual memory)?

To answer these questions, we need some additional background knowledge about *interrupts*:

- Interrupts is an *asynchronous* communication mechanism used by CPU

  – Used to communicate with hardware, among other things

- Several interrupt numbers (for example 256 in Intel CPUs) assigned to e.g. different hardware for asynchronous communication with CPU

LiU LINKÖPING UNIVERSITY

Side Note: Interrupts
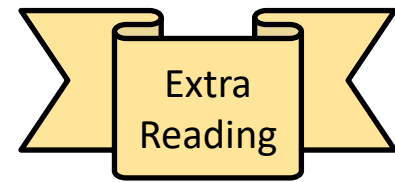
For example, instead of having to wait for keyboard input in a loop, OS registers an interrupt handler to take care of keyboard events

- OS goes about its business, running processes etc.

- When you press a key on the keyboard, an interrupt signal is sent to CPU

- This will *automatically* (i.e. as part of the CPU hardware):

  – Store CPU registers in temporary scratch space

  – Switch CPU to kernel mode

  – Look up the address to correct interrupt handling code in the *Interrupt Descriptor Table* (IDT), corresponding to the keyboard's interrupt number

    • IDT is a list of kernel code addresses, one per interrupt number

    • IDT is set up by OS or boot loader while booting

  – Jump to code to read input from keyboard on an input bus

- When done handling interrupt, OS executes a CPU instruction that restores registers from scratch space, switches CPU to user mode, and continues execution where it was before interrupt

LiU LINKÖPING UNIVERSITY
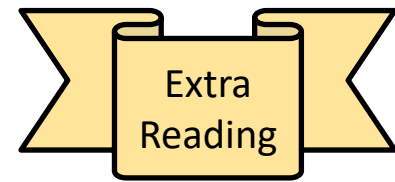
Side Note: Interrupts

- Note that interrupt handling sounds very similar to the way system calls are implemented?

  ⇨ Many OSes use interrupts to implement the user-mode/kernel-mode switch

  ⇨ Process can generate interrupt directly from CPU using special instruction

  ⇨ Some modern CPUs have specialized instructions for issuing system calls

- Interrupts are also used to handle access violations

  – When code attempts to access illegal memory or similar, a special interrupt is signaled

  – Causes CPU to jump to exception-handling code in kernel

  – Kernel knows which process is currently executing, and can e.g. decide to kill it after inspecting the cause of the problem

LINKÖPING UNIVERSITY

Side Note: Process Switching
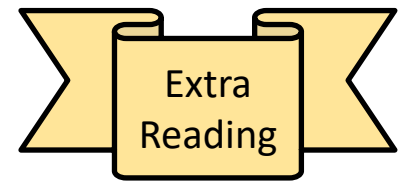
Typically there are many more running processes than there are CPU cores

⇨   Remember: Only one process can execute at a time on one CPU core

⇨   CPU is a shared resource!

- OS rapidly switches execution between running processes, to create the illusion of all process running "simultaneously"

- Switch from process A to process B proceeds as follows:

  1. Execution of A is paused, kernel code starts executing

  2. Kernel saves values of all registers

  3. Old saved register values for B are copied into CPU registers

  4. Kernel sets B's page table to be the "active" one.

  5. Kernel transfers execution to B's code (using saved IP value of where B's execution "left off")

  6. Execution of B is now resumed

> Note: Kernel also need to share the CPU with all processes!
>
> ⇨   Need a way for the kernel to regularly "step in", switch processes, and relinquish CPU to the new process

LiU LINKÖPING UNIVERSITY

Side Note: Process Switching

How can the kernel "step in" and switch processes?

⇨   Interrupts!

- OS sets a built in hardware timer to send an interrupt at fixed intervals
  - Allows running process to be suspended and kernel to briefly run scheduling code
  - Kernel decides if a process switch should happen before resuming process execution

LINKÖPING UNIVERSITY

Summary So Far

- Booting/Login:
  BIOS → Boot Loader → Kernel → Login Process → Shell Process→ Application Processes

- The process is the main entity on which access control is performed

- DAC/MAC/RBAC

- No direct access to RAM/hardware for processes

  – Kernel must mediate access to resources

LINKÖPING UNIVERSITY

Summary So Far

- Virtual Memory

  - Each process isolated in its own memory space

  - Hardware-implemented translation between virtual and physical memory

- System Calls

  - Main interface between kernel and processes

  - Two CPU modes: kernel and user mode

  - Uses hardware mechanism to switch between untrusted code in user mode and trusted code in kernel mode

    - Ensures that control is transferred to trusted code upon switch to kernel mode

LINKÖPING UNIVERSITY

Shortcomings of Traditional System Design

- Most hardware and OS protection mechanisms mostly aimed at stability and robustness

  - Prevent processes from *mistakenly* corrupting data belonging to other processes or kernel

- Security has also been an early concern is OS/hardware design

  - but perhaps not the main concern...

- Next, we'll look at some common attacks against computer systems, bypassing traditional OS/hardware security measures

  - In the coming lectures, you will see some newer techniques to combat these threats

LiU LINKÖPING UNIVERSITY

When Attacker has Physical Access

Reading filesystem "offline"

- OS can only apply access controls when it is running

  - Attacker can physically remove hard drive and plug it in to another computer

  - Attacker can boot own OS from a USB stick/CD/DVD

- If not encrypted, all files are physically available on hard drive

- Swap space may also contain old RAM contents after power off

  - Passwords, etc.

LINKÖPING
UNIVERSITY

When Attacker has Physical Access

Cold Boot attacks

- Even if files are encrypted, attackers may be able to extract encryption keys from RAM, if machine is powered on

- In practice:

  - While computer is running:

  - Spray RAM chips with "cooling spray", cools chips to downwards of -50° C

  - Power off machine, pull out RAM modules

  - Cooling retains RAM contents for several minutes after power loss

  - If chips are put in a can of liquid nitrogen, contents are retained for hours

  - Attacker now have plenty of time to plug in the RAM modules in his/her own machine

    - Alternatively, reboot machine into special-purpose OS on USB stick

  - Can now read out RAM contents, e.g. encryption keys for hard-drive encryption

When Attacker has Physical Access

DMA attacks

- For devices that require very high throughput, interacting with the system solely through the OS kernel (system calls, interrupts) becomes a bottleneck

- Modern computers therefore often implement standards that allow peripherals to bypass the kernel and send data **directly to physical memory**, bypassing the virtual memory translation. This is called Direct Memory Access (DMA).

    - Firewire, Thunderbolt, PCI Express, etc.

- Attack scenario:

    - You leave your laptop, locked but powered on, in your hotel room

    - Attacker gains access to room, briefly (less than a second), plugs in a device in the Thunderbolt or Firewire port of the computer

    - Attacker has now written malicious code directly into the kernel code segment, which when executed installs spyware onto the machine

    - Alternatively, attacker may read out sensitive data directly from RAM (e.g. encryption keys)

LINKÖPING
UNIVERSITY

When Attacker has Physical Access

Reading data off buses:

- An attacker that have physical access to the computer can also connect probes onto the buses and read out the data flowing in them

  – Encryption keys, other sensitive data

- Could be a big problem for devices in "enemy territory"

  – "Kiosk" computers, mobile base stations, etc.

- Also a serious challenge for DRM schemes

When Attacker has Physical Access

Conclusions:

- Encryption helps, but is not undefeatable

- Physical security very important!

- Very hard to protect a device from its owner!

LINKÖPING UNIVERSITY

Attacks From Below

- The security of higher-level components depends critically on the security of lower-level components

  - If attackers can gain access to low-level components, they completely own everything on higher levels!

Attacks From Below

Rootkits

- Special kind of malware that embeds itself "deep" in system software to hide its presence

- For example, loads as a device driver very early during boot

- Runs in kernel mode ⇨ Has complete control over system (same as OS itself)

- A rootkit can, for example:

  - hide its presence on the file system by manipulating results of system calls that list directory contents, etc.

  - disable antivirus software

- Hard to detect and remove!

LINKÖPING
UNIVERSITY

Attacks From Below

BIOS attacks:

- Computers typically have mechanisms to update BIOS code

  - If an attacker gets access to this *once*, he/she can establish a permanent backdoor

- Even if hard drive is wiped (or even changed) BIOS malware can reinstall e.g. rootkit on hard drive

LINKÖPING UNIVERSITY

Attacks From Below

Firmware attacks:

- Apart from the BIOS, computer add-on hardware also has firmware

  - Hard drives, network cards, etc.

- These can also be infected!

- Hard drive firmware controls what is returned when system asks for data from drive

  - Can silently manipulate code while it is being fetched from hard drive, to insert malicious behavior

    - When scanning for malicious files, nothing is found, because there are no malicious files! (Infected device can also manipulate results of scanning.)

  - Firmware is inaccessible to e.g. antivirus software – almost impossible to detect infection

  - Known examples of spyware that utilize e.g., hard drive firmware infection

LINKÖPING UNIVERSITY

Attacks From Below

Conclusions:

- Impossible (in the general case) to defend from attacks from lower levels!

- Must make sure that the *whole chain of hardware/software* used for initializing a system can be trusted

- 1st level must verify integrity of 2nd level,
  2nd level must verify integrity of 3rd level...

- First level must be trusted *a priori* – a so-called **Root of Trust**

  – Ideally, it should be impossible to alter

  – ...but what to do if it has bugs?

LINKÖPING UNIVERSITY

Attacks Against System Software

User-mode process exploits:

- If a program has certain types of bugs, an attacker can supply it with specially crafted inputs that will corrupt internal data structures

    - Allows attackers to inject machine code into the running process and execute it

    - Attacker's code now runs with the privilege of the process – has access to all your files!

- For example, attacker sends malicious PDF file as email attachment

    - When opened in vulnerable PDF viewer, attacker code will run and install malware

- Not the main focus of this course, in-depth treatment in
  TDDC90 – Software Security

LINKÖPING UNIVERSITY

Attacks Against System Software

Kernel-mode exploits:

Consider the Unix **read** system call for reading data from a file into memory:

`read(int file_descriptor, char* mem_buffer, size_t size)`

Unique ID of an open file

Address in memory where file contents should be written

Number of bytes to read

Extremely important that kernel verifies that the address in `mem_buffer` is valid!

Why?

The address should point somewhere here, but what if it instead points here?

Remember: Process cannot directly write to kernel data, but what if it can trick a system call to do it? Can manipulate own permissions, etc.!

Kernel Code
Kernel Data
Data
Code

LINKÖPING UNIVERSITY
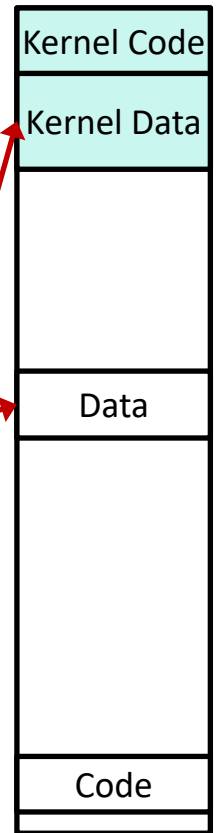
Attacks Against System Software

Kernel-mode exploits:

The `read` system call is (hopefully) safe in most modern OSes, but conceptually similar problems have been discovered in common OSses in the past

Also, OS kernel code is not the only thing running in kernel mode

- Device drivers need to communicate directly with hardware, and so must typically execute in kernel mode. (Loaded into the kernel code segment in VM.)

- These can (and frequently do) have bugs!

Buggy code running in kernel mode can also lead to arbitrary code execution, similar to user-mode process exploits described earlier

LINKÖPING UNIVERSITY

Attacks Against System Software

Conclusions:

- Kernel-mode/User-mode privilege model is perhaps too coarse-grained?
  - Primarily designed to ensure system stability, not (data) security
  - Is *all* code running in kernel mode (e.g. drivers) really more trusted than *all* code running in user mode? Trusted with *what*?
  - User mode processes handling sensitive data can always be tampered with by kernel-mode code
  - Perhaps more fine-grained permission models are needed?
- How to avoid "attacks from below"?
  - Need some way to check that system software has not been tampered with, *which doesn't rely on trusting that system software has not been tampered with...*
  - Cannot rely solely on software for this!

That's All!