Sistem Programlama

60613METOZ-YZM0055

Purpose and Content

- Purpose: The aim of this course is to teach students assembly language programming, the programming interface of UNIX and Windows operating systems, the use of standard I/O library functions in C language and their applications at theoretical and practical levels.
- Course Content: Windows environment and tools; Unix environment and tools; Programming in C language; programming in assembly language; Unix system calls.
- Book: UNIX Systems Programming: Communication, Concurrency and Threads Kay A. Robbins and Steve Robbins (Jun 27, 2003)

Proje Odaklı Öğrenme

Item	# of items	Points per item	Total points for item (over 100)
Project	1	30	30
Homework	5	2	10
Midterm	1	10	10
Final	1	50	50

Syllabus - 1

Week	Topics	Evaluation
1	Introduction of Course Syllabus	
2	Systems Programming Basics	Project Assignment
3	Introduction to Linux Command Line	Homework Assignment
4	Introduction to Linux Shell Scripting	Project Review - I
5	Introduction to Linux I/O, File Access Techniques	Homework Assignment
6	Linux File System, File/Directory Management	Project Review - II
7	Linux File Monitoring (inotify API)	Homework Assignment

MIDTERM

Syllabus - 2

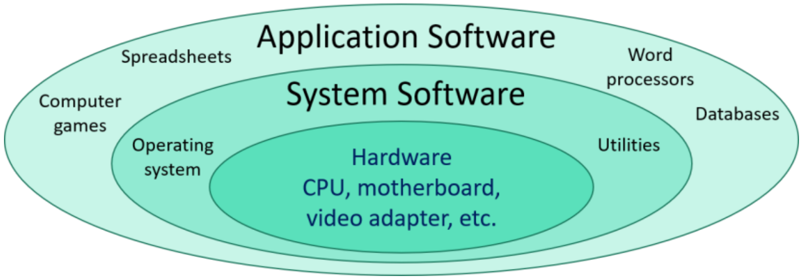
Week	Topics	Evaluation
1	Linux CLI Access using C	Project Review - III
2	Linux CLI Access using Python	Homework Assignment
3	Processes and Pipes on Linux	Project Review - IV
4	Linux Access, Identity and Permissions	Homework Assignment
5	Linux Signals	Project Presentations – Dry Runs

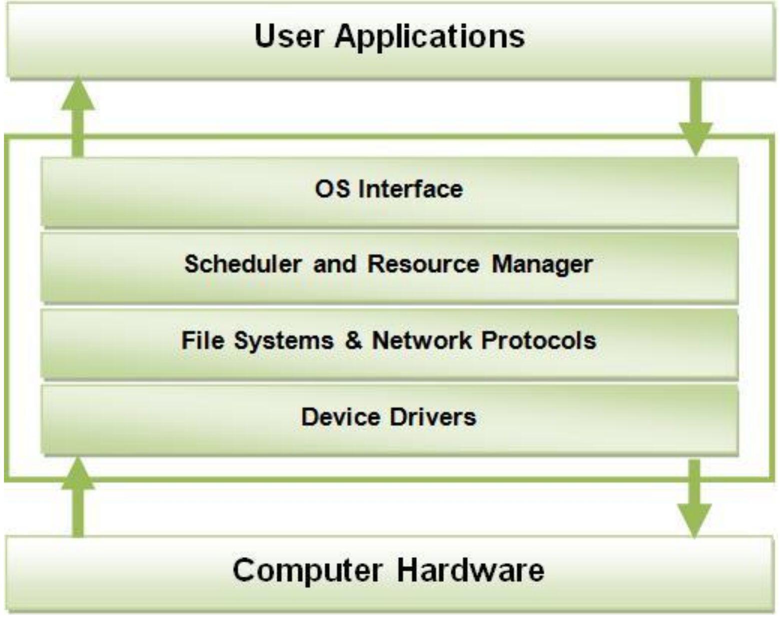
Project Presentations
FINAL Exam

Giriş

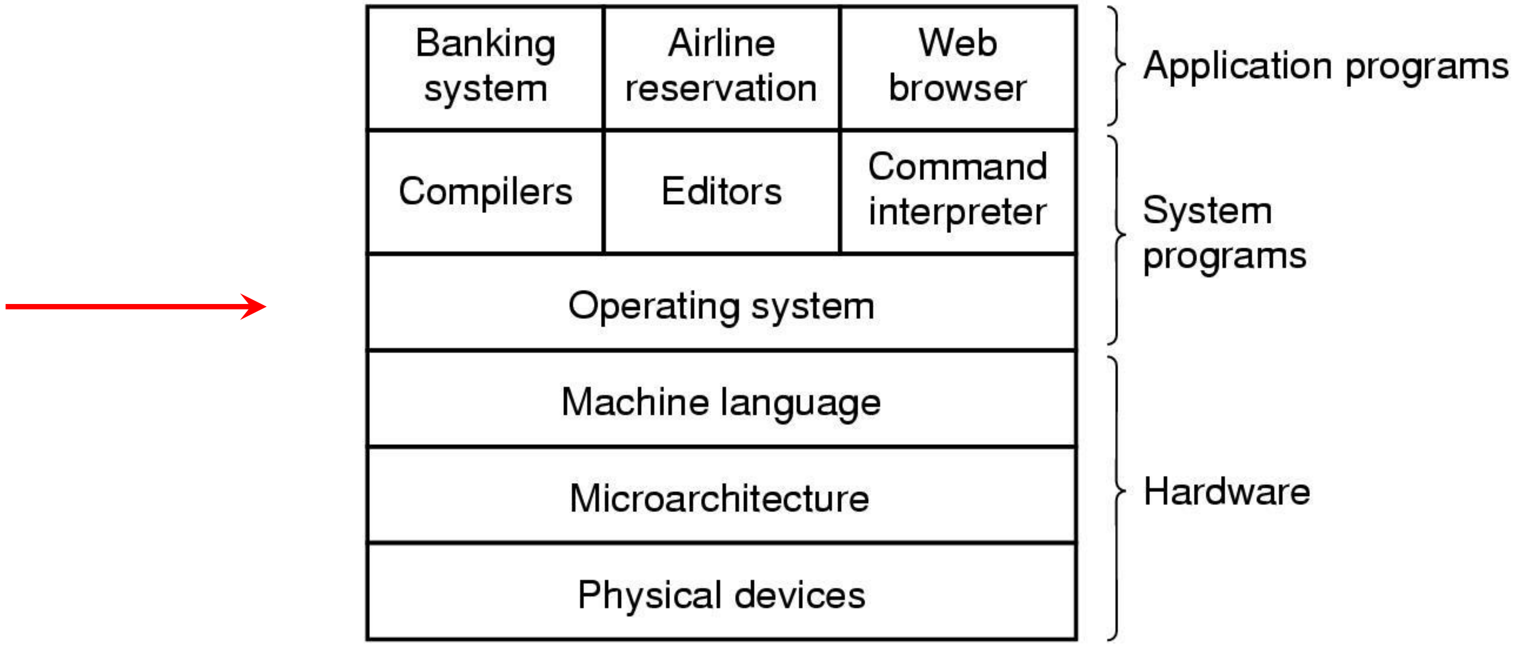
SİSTEM PROGRAMLAMA NEDİR

Sistem Yazılımları





Organization of a Computer System



- A computer system consists of
  - hardware
  - system programs
  - application programs

What is an Operating System

- It is an extended machine
  - Hides the messy details which must be performed
  - Presents user with a “virtual machine”
- It is a resource manager
  - Each program gets time with the resource
  - Each program gets space on the resource
- It is a protection mechanism
  - Hardware from damaging programs
  - Isolate programs from each other
  - Protect data

The OS as an Extended Machine

Real hardware is complicated.

The OS gives us a much nicer picture:

- Multitasking: A private computer (“process”) for each application, with nicer memory, I/O devices
- Communication among processes
- Simplified actions: I/O
- Complex actions: file system

- I/O == Input/Output

The OS as a Resource Manager

- 3 programs use a printer at one
  → garbage is printed.
- A program goes into an infinite loop
  → computer freezes.
- A process erases whole memory
  → other programs die.

The OS makes sure that resources are **shared fairly**.

Two types of multiplexing:

- Time: processes have to take turns (printer, network, CPU)
- Space: resources are split among processes (memory, disk)

The OS as a Protection Mechanism

- Isolates processes and protects information:
  - A process should not write or read another’s memory
  - A bug in one process shouldn’t affect others (including endless loops!)
  - A process should not issue raw I/O commands
- “Regular” applications cannot access the physical resources - only the OS should

Common Operating systems

Microsoft Windows

- Developed and sold by Microsoft
- Works on Intel x86 (8086, 80286, 80386, 486, Pentium)

- 2 separate lines of operating systems:

"Consumer":
DOS → Win95 → Win98 → Win ME → Win XP → Vista

"Professional":
Win NT → Win2000 → Win XP → Vista → Windows7

- The 2 lines have finally converged…
  (but: Home/Pro/Server variants).

- Consumer line had some very weak OS components because of DOS roots.

Commercial Unix

- Large class of operating systems, by many vendors, that share the same source code. Created in **Bell Labs** (then AT&T) in the 1970's, by Dennis Ritchie & Ken Thompson.
- Most developed from code improvements by **Berkeley** researchers: BSD (Berkeley Standard Distribution)
- Examples: Sun Solaris, IBM Aix, HP-UX, Apple Mac (from version X) is based on Unix with a special user interface. Each on dedicated hardware. Mostly gone by now.
- From personal computers to super-computers

Free Unixes

- Distributed for free as source code, for Intel x86 architectures.
- The “open source” movement
- **Linux**: written from scratch in 1991 by Finn student called Linus Torvalds. Recently adopted by major vendors: IBM, HP
- Other versions: FreeBSD, OpenBSD, NetBSD
- Since 2005 Sun Solaris 10 is free as well

Other operating systems

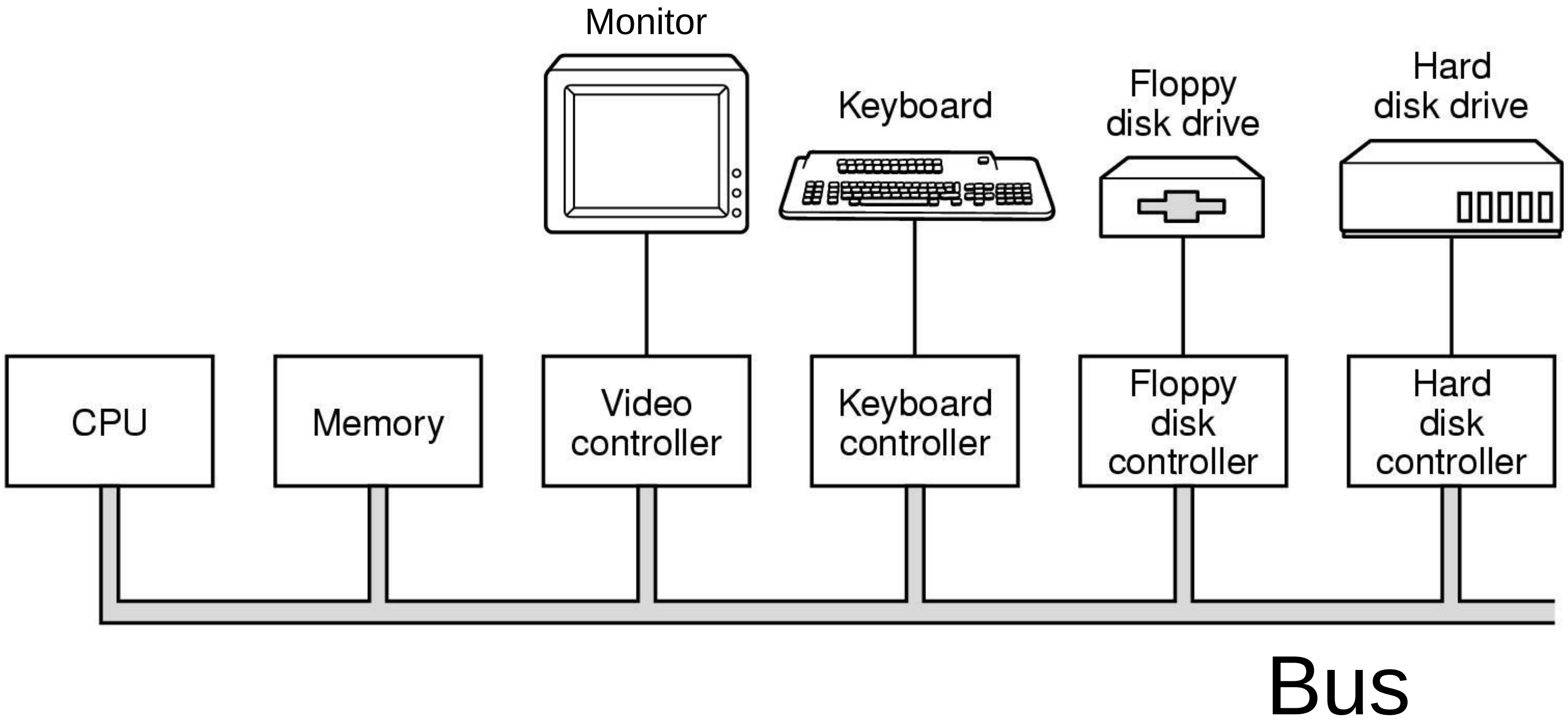
- Mainframes: IBM MVS, IBM VM/CMS. First developed in the 1960s, still in use on central servers.

- Real time operating systems: Nuclear reactors, flight systems, car computers, medical equipment. Have very strict timing requirements that regular OSs can't guarantee.

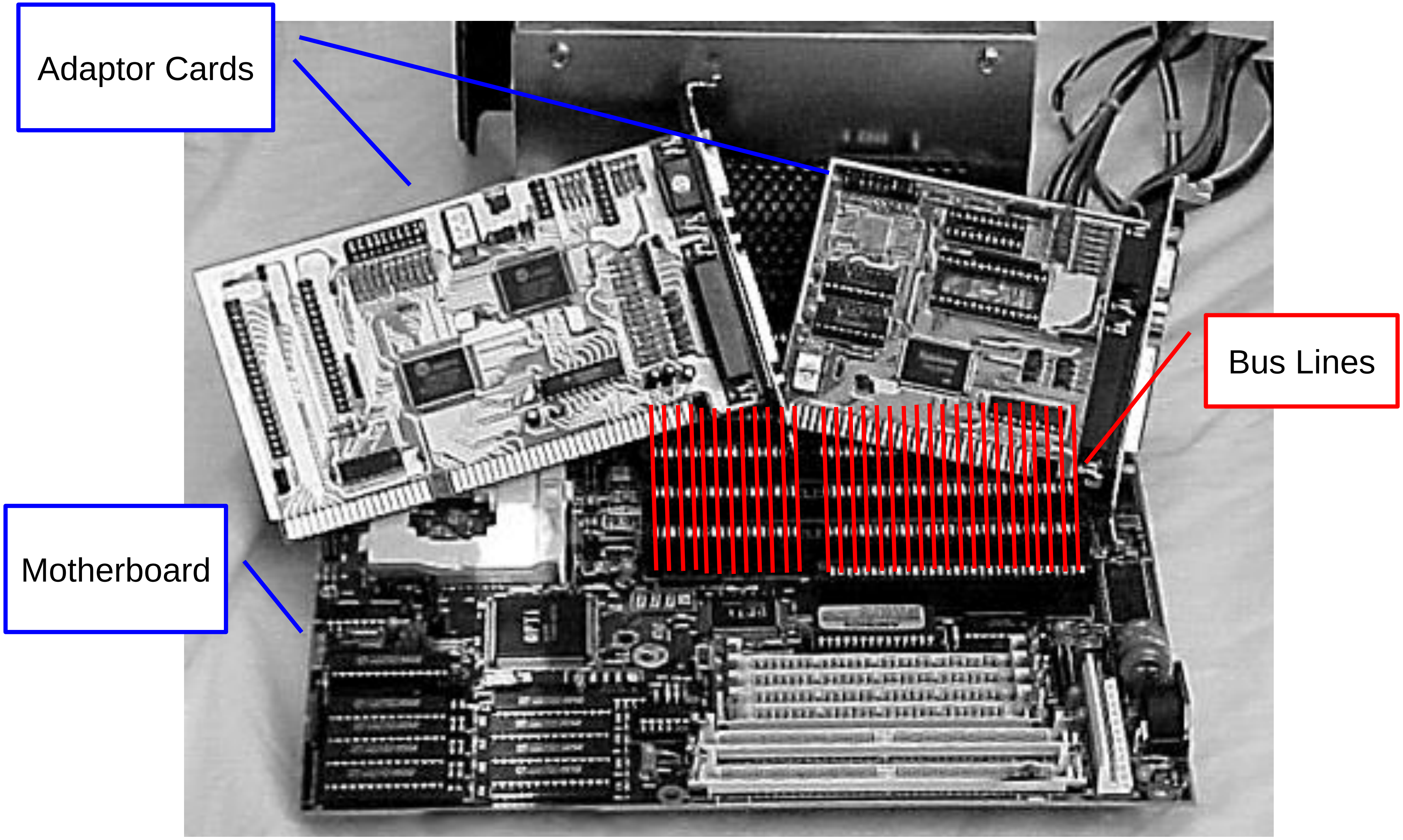
- OS for weak computers: smartcards, palm computers (Windows Mobile, PalmOS), phones (e.g., Android, Symbian) set-top terminals: hardware is too simple for full OS.

Computer Architecture Review

Basic computer components



Inside the box



Example of CPU Instruction Set

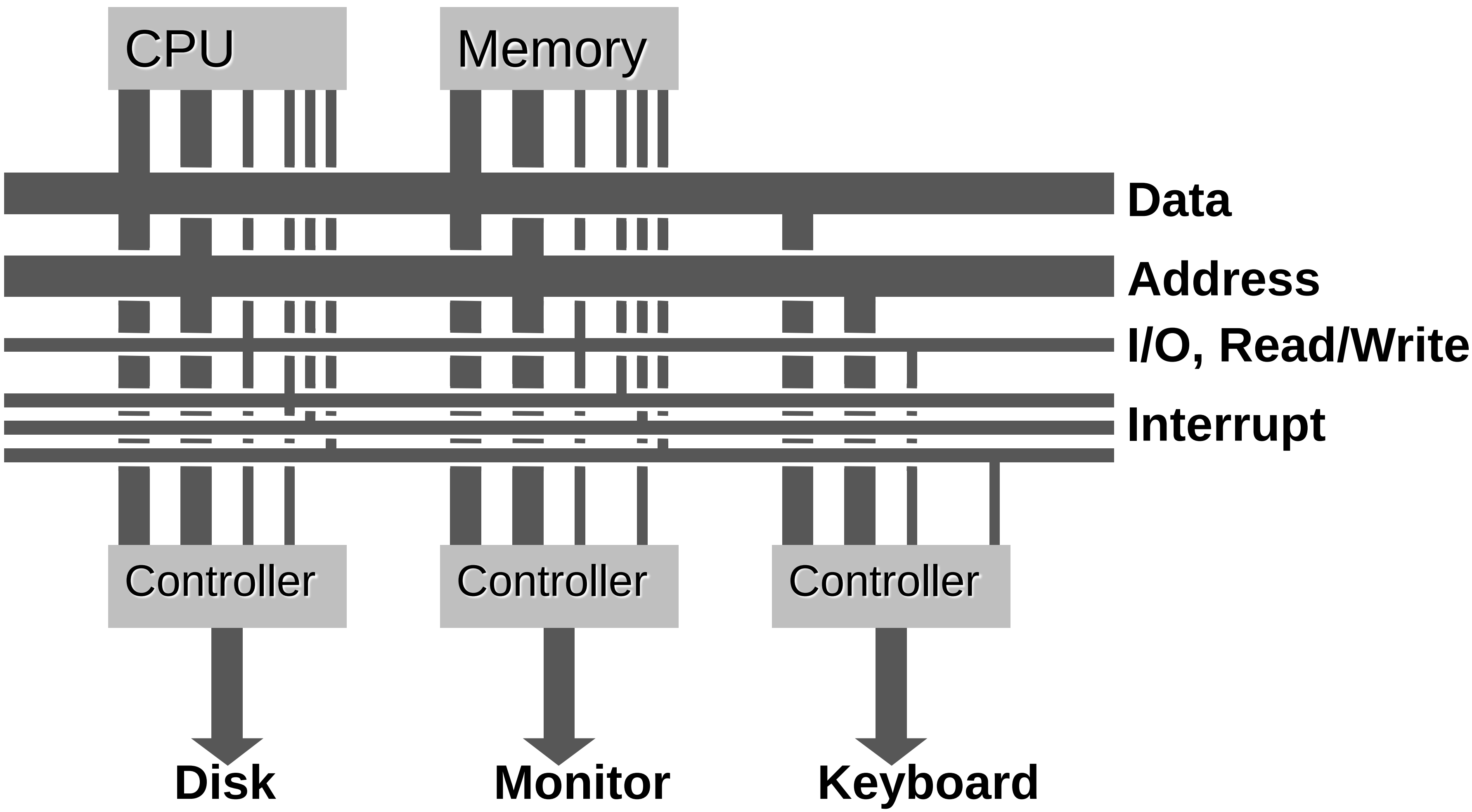
- **Copy:**
  - `load`
  - `store`
  - `move`
  - `push/pop`
  - `...`
- **Algebraic:**
  - `add`
  - `subtract`
  - `xor`
  - ...
- **Flow control:**
  - `branch`
  - `compare`
  - `call (jump to subroutine)`
  - `ret`
- **I/O:**
  - `in`
  - `out`
- **Resource interface:**
  - `syscall`
  - `rti` (return from interrupt)

Example CPU Structure

- general purpose registers of 32 bits each: **`r0`**`,`...`,`**`r31`**

- **`fp`** - is the **frame pointer**
- **`sp`** - is the **stack pointer**
- **`ia`** - is the **instruction address** (program counter)

Bus architecture in more detail

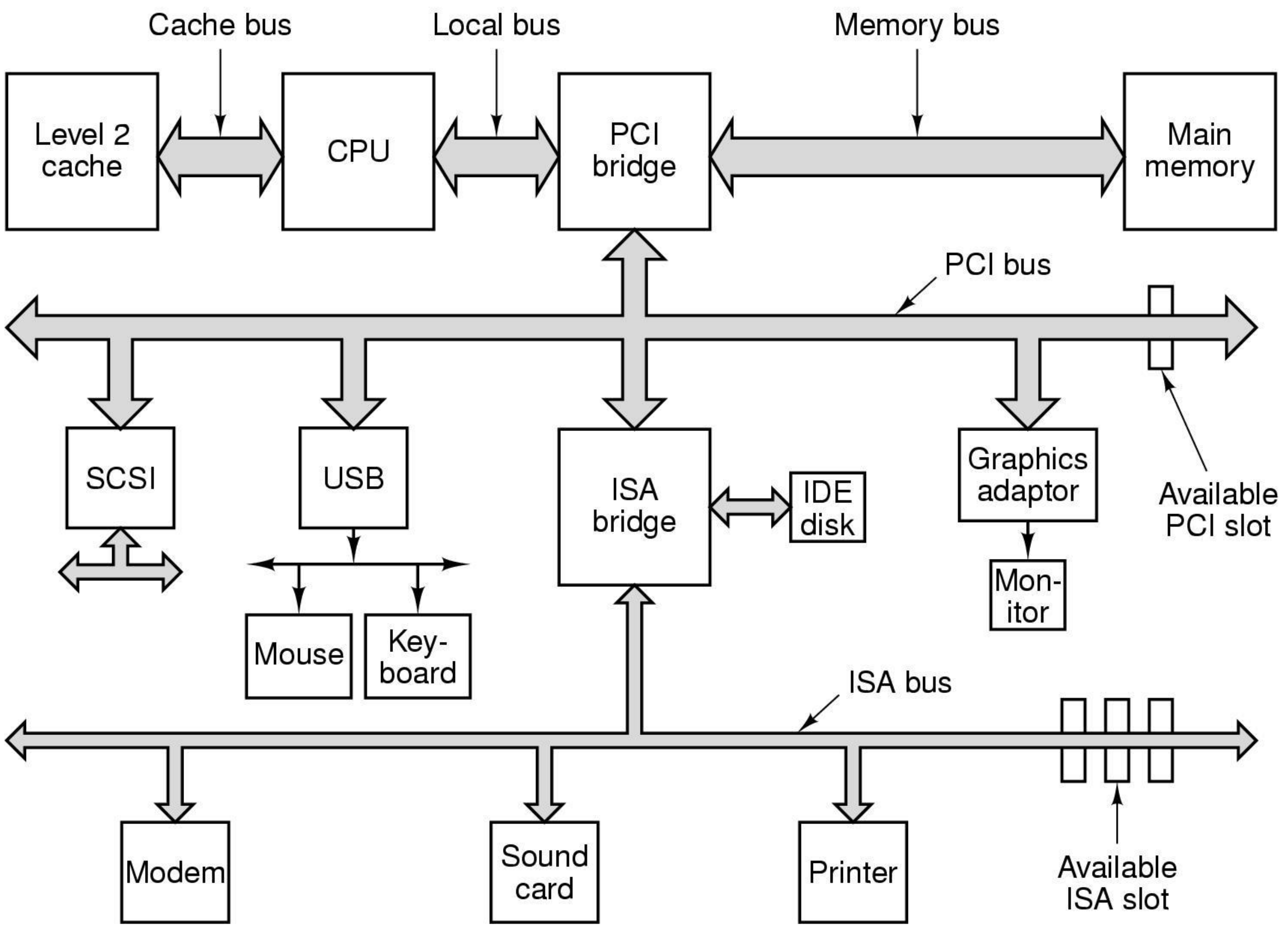


Moving data Memory ⇔ CPU

```
load r0 = 0x001000F8
```

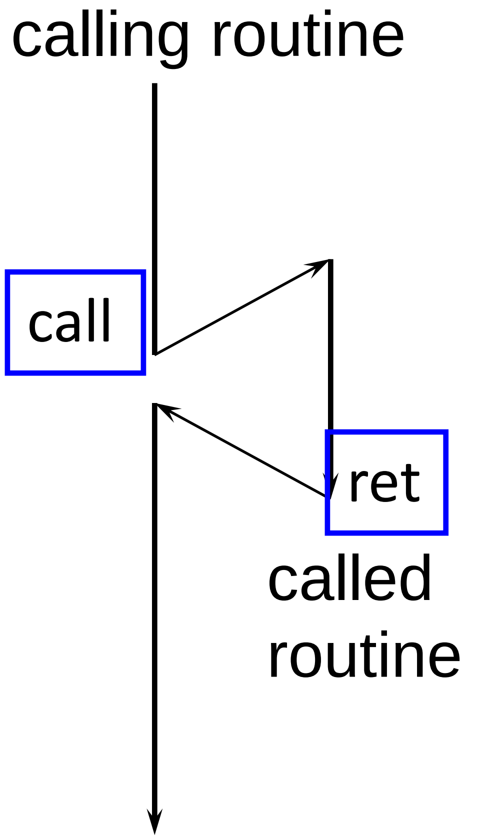
- CPU puts address 001000F8 (32 bit) on bus "Address" lines – 1 bit per line
  - "32 bit address bus"
- CPU raises the "Read" line
- RAM (Memory) copies 32 bits from address 001000F8, puts on bus "Data" lines
  - "32 bit data bus"
- CPU copies bits from Data lines into CPU register r0

Structure of a large Pentium system



Regular procedure call

- load registers/stack with argument values, including return address (calling convention)
- push registers on stack (sometimes)
- pass control to procedure by changing **`ia`**
- when returning, restore registers using **`fp`**:
  - general purpose from saved value on stack
  - stack pointer is restored (effectively popping all local info)
  - **`ia`** is restored—returning control to the point after the procedure call



Procedure call example

- C statement: y = sin(x);
- machine code:
  - `load r8 = *(sp-20)` //assuming this is x
  - `push r8;`
  - `call sin` // this stores current **`ia`** as return address

  (now sin runs, uses registers, allocates memory on stack etc.; presumably, **`r1`** is filled. Finally, we reach...)
  - `ret` // this restores **`sp`** and **`ia`** from **`fp`**

  (now we can work on, with **`r1`** holding sin(x)).

The Program Status Word

- Special register in CPU called **`psw`** (program status word).
- Contains various bit fields.
- Two important bits:
  - **`ku`** (or mode) bit: kernel/user mode.
  - **`ie`** bit: interrupt enable.

Modes: Kernel and User

- **`Ku`** bit: also called “privileged”, or “supervisor”.
- Things that are only allowed in Kernel mode (**`ku`** == 1):
  - Access to memory that belongs to another process or to OS.
  - Raw Input/Output commands.
  - Changing the **`ku`** bit.
  - Disabling/enabling interrupts.

How does the OS get control?

- The OS is not “running in the background”!
- CPU usually running a process, in user mode (**`ku`**==0)

- OS gets control in 2 cases:
  1. Hardware interrupt.
  2. User process issued the **`syscall`** instruction.

- In both cases the **`ku`** bit is set to 1 (kernel mode).
- An important interrupt is the timer: e.g., every 10ms the OS gets control.

Possible Interrupts

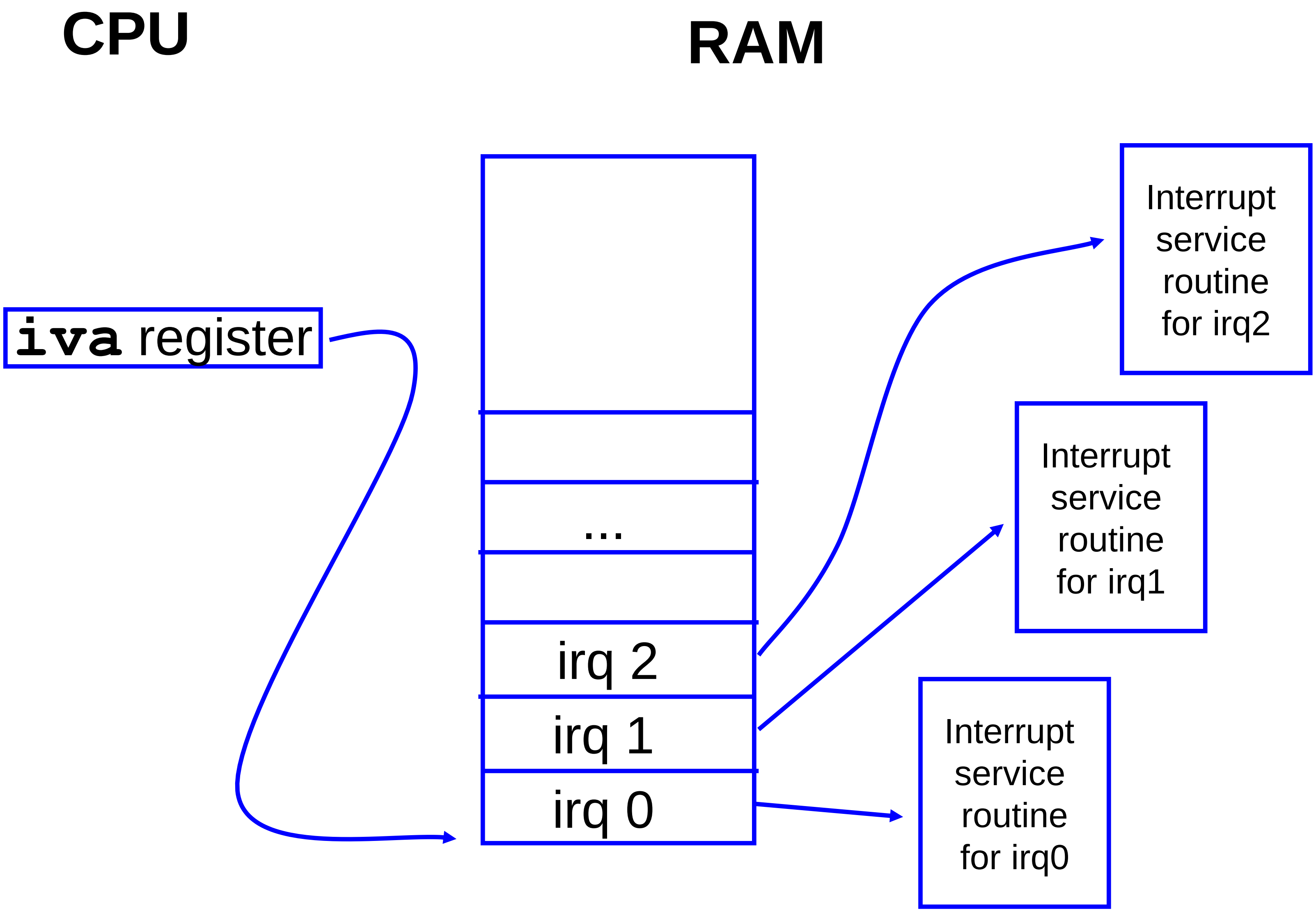
- Inputs: keyboard controller, network card, disk controller
- Output: printer controller, disk controller
- Timer down to 0
- Illegal instruction
- Illegal address
- Division by 0

Some more control registers

- **`irq`**: interrupt request
  (number of interrupt that happened)
- **`iva`**: interrupt vector address

- **`iia`**: saved **`ia`** value
- **`ipsw`**: saved **`psw`** value

The Interrupt Vector



Interrupt Processing Example

- key is pressed
- electrical signals sent to CPU on the bus control lines.
- **`irq`** register is filled (indicating interrupting device).
- **`iia = ia`**; **`ipsw = psw`**; // save process state
- **`ku = 1`**; // set mode to kernel
- **`ie = 0;`** // disable interrupts
- **`ia = iva[irq];`**

(this causes the interrupt handler routine to run, until...)

- **`rti;`** // restores **`ia,psw`**

(restoring **`psw`** sets **`ku`** back to 0 and enables interrupts)

System Call Invocation

- load registers with system call **identity** and parameters (OS convention)
- issue **`syscall`** instruction. Also called: **`trap`**
- hardware:
  - **`ku = 1;`** // set mode to kernel
  - **`ie = 0;`** // disable interrupts
  - saves register values
  - passes control to syscall handler routine (entry in **`iva`**)
- handler routine invokes actual system routine
- system routine does service, ends it with **`rti`**

System Call vs. Procedure Call

- System calls are a fixed repertoire; procedures are written by the user.
- System calls run in kernel mode; procedures run in user mode.

The idea: applications must have access to physical resources. The OS allows them only to submit “forms” with their requests. The set of forms determines the OS/Application interface (API)

Operating System Concepts

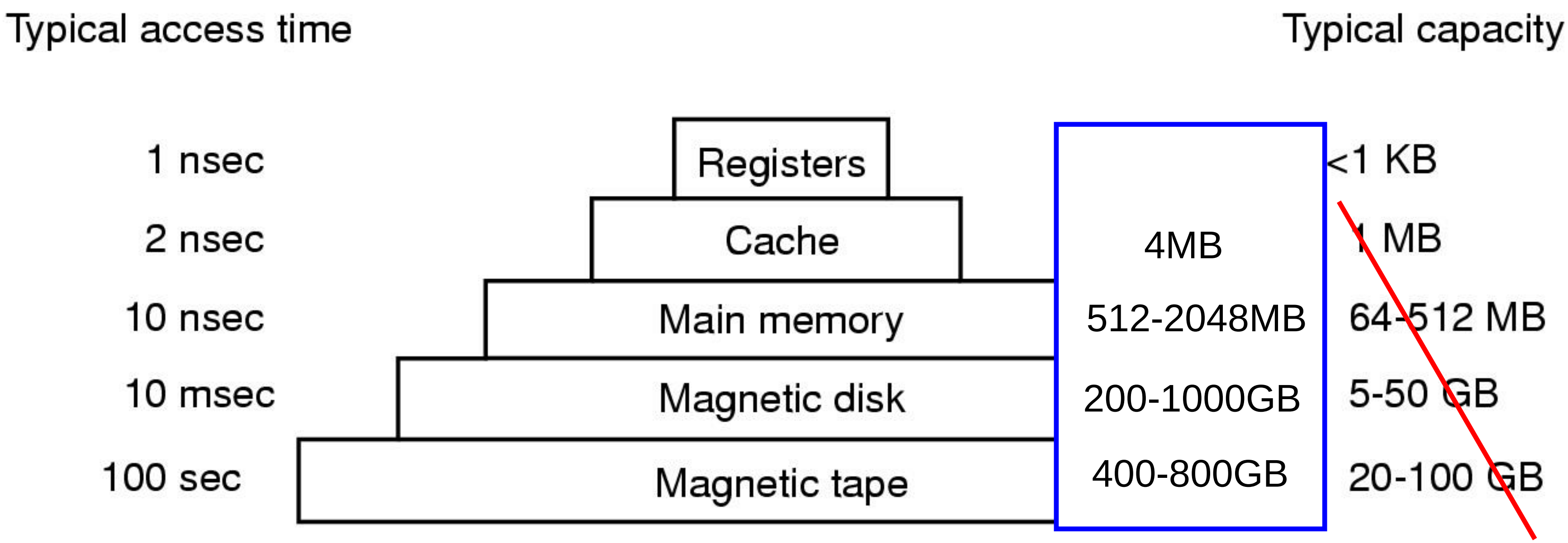
Processes and Threads

- Process: A program in execution, with a virtual computer.
- Has:
  - An address space (area in memory)
  - The program's code loaded into memory
  - The program's data and stack
  - Register contents, including **`ia,sp,fp`**, and **`psw`**
- Thread: A virtual CPU (within a process).
  - Shares address space with other threads in process.
  - Has its own register contents

Scheduling

- Which process should get the CPU now?
- How can a process be stopped so it can be resumed?
- How does the OS switch between one process and another (**context switch**)?
- What to do when a process is waiting for I/O?
- Fairness?
- Priorities?

Typical memory hierarchy



- numbers shown are rough approximations

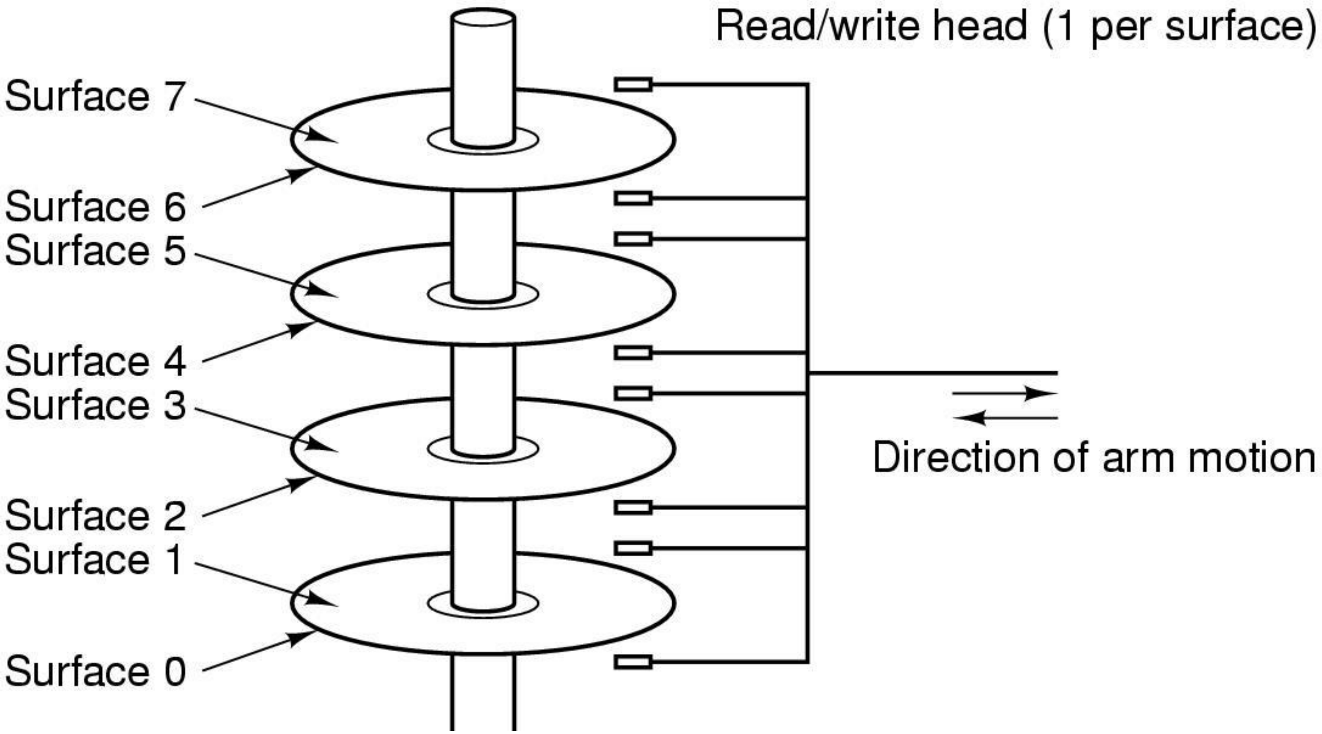
Memory Management

- Hardware usually takes care of the CPU Cache

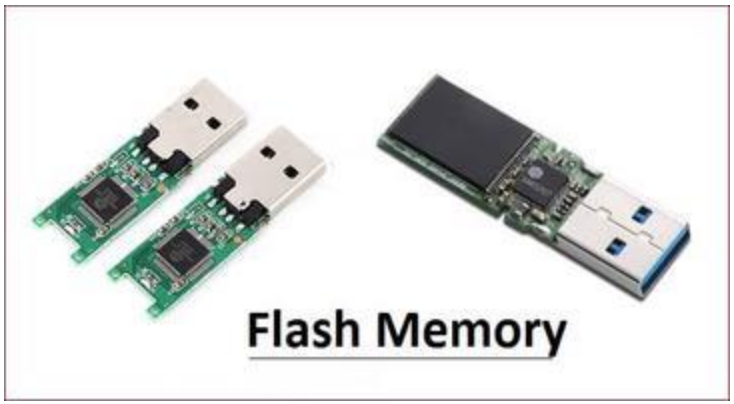
OS decides:

- How to allocate Main Memory between processes?
- What to do when Main Memory is smaller than processes want?
- How to use the hard disk to extend **Virtual Memory** (without a performance hit)?

Structure of a disk drive



Flash Memory

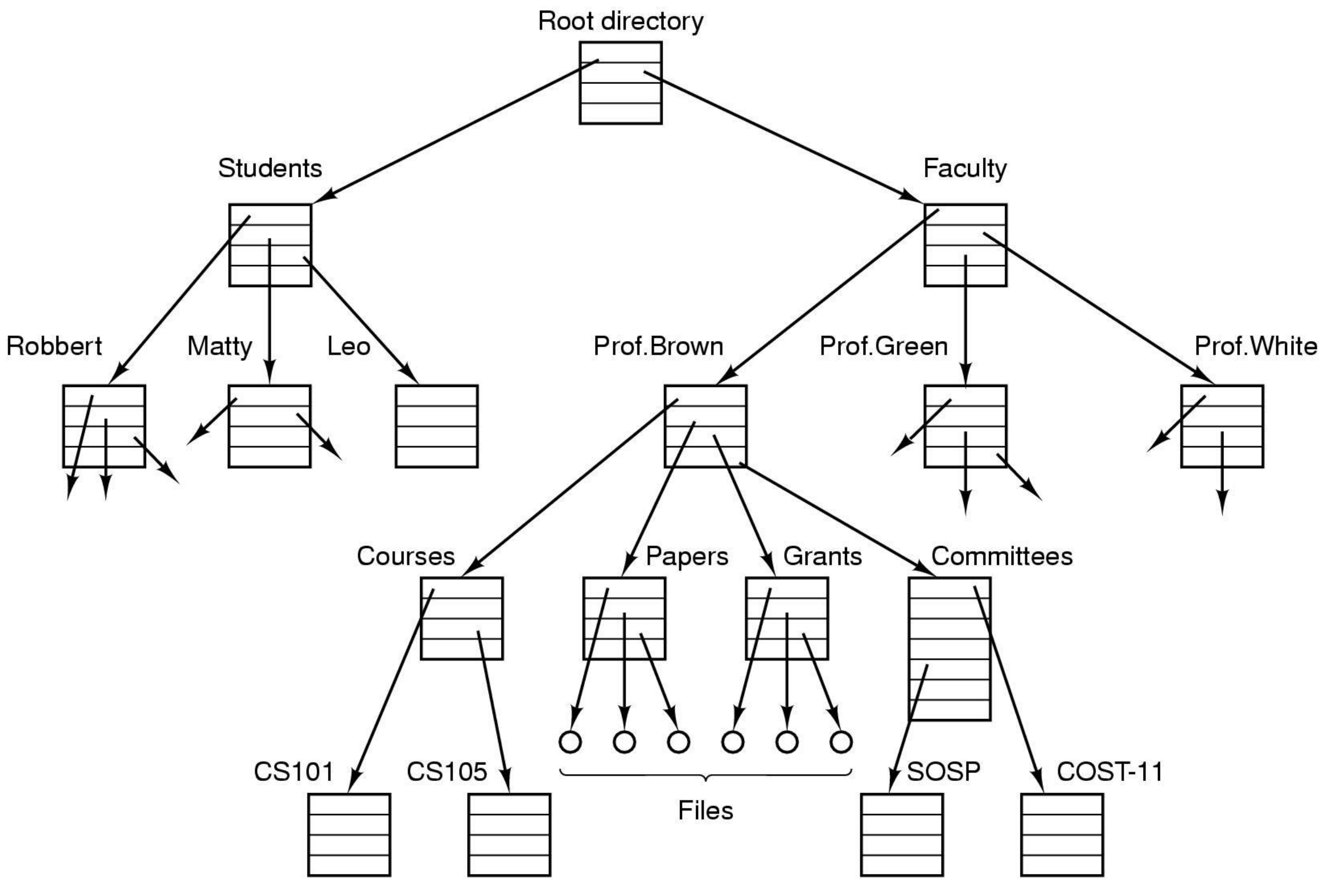


Files

- A file is a "virtual disk"
- Files have a name
- Directory structure:
  - C:\My Documents\yash\lecture1.ppt (Windows)
  - /home/users/yash/isp-course/grades.txt (Unix)

- Operating system decides how to organize files on physical disk

File system



Root directory
Students
Faculty
Robbert
Matty
Leo
Prof.Brown
Prof.Green
Prof.White
Courses
Papers
Grants
Committees
CS101
CS105
Files
SOSP
COST-11

Other objects

- Print queue
- Network connections
- Users, User groups
- Permissions and Security

Concepts for review

- Bus
- Interrupt
- PSW (Program Status Word)
- Kernel mode / User mode
- System call