INF1060:
Introduction to Operating Systems and Data Communication

Operating Systems: Introduction

Pål Halvorsen

Wednesday, September 19, 2012

Overview

- Basic execution environment – an Intel example
- What is an operating system (OS)?
- OS components and services
  (extended in later lectures)
- Booting
- Kernel organization

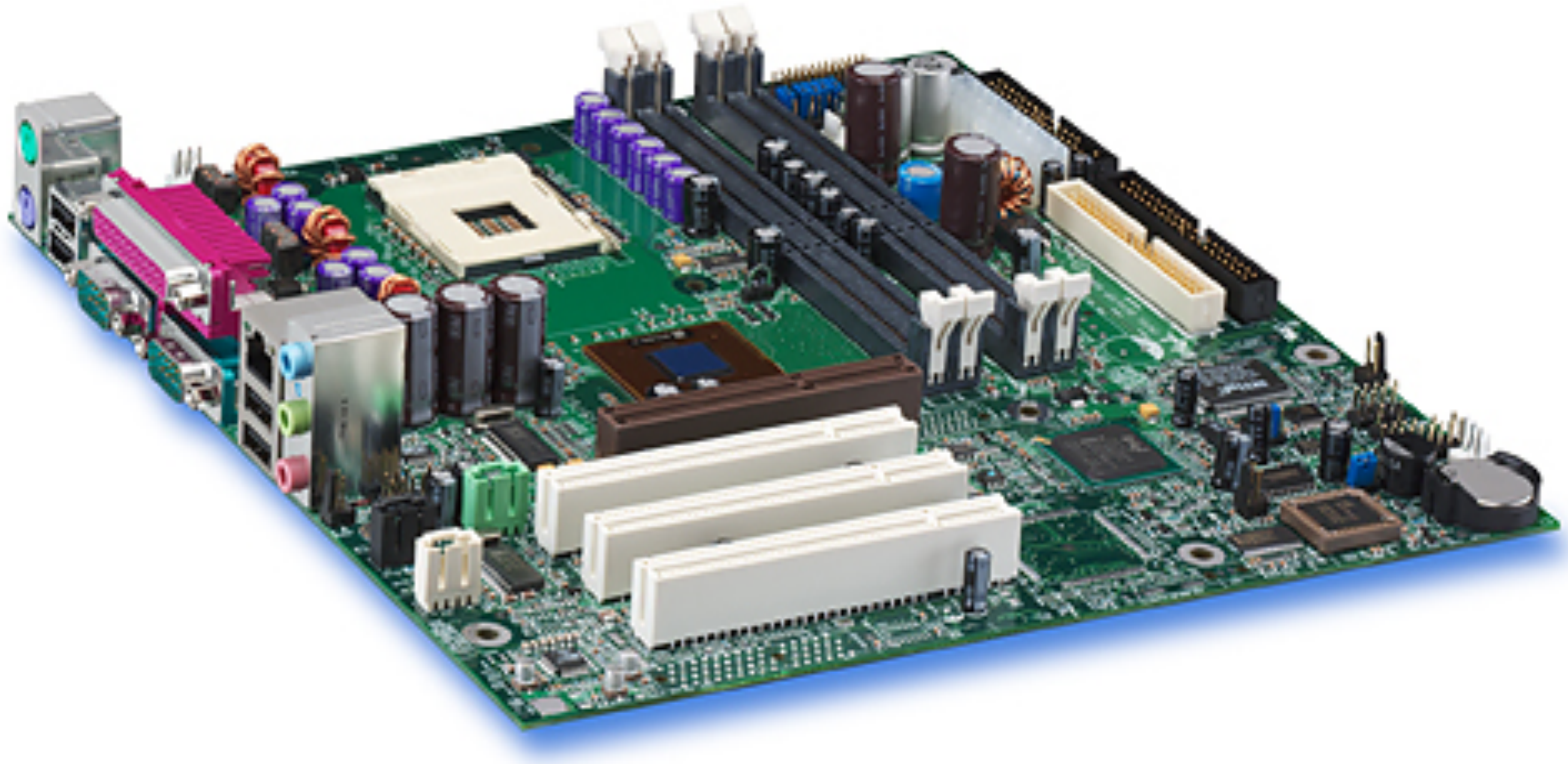
Hardware

- Central Processing Units (CPUs)
- Memory
  (cache(s), RAM, ROM, Flash, …)
- I/O Devices
  (network cards, disks, CD, keyboard, mouse, …)
- Links
  (interconnects, busses, …)

Example:
Intel Hub Architecture (850 Chipset)

Intel D850MD Motherboard:

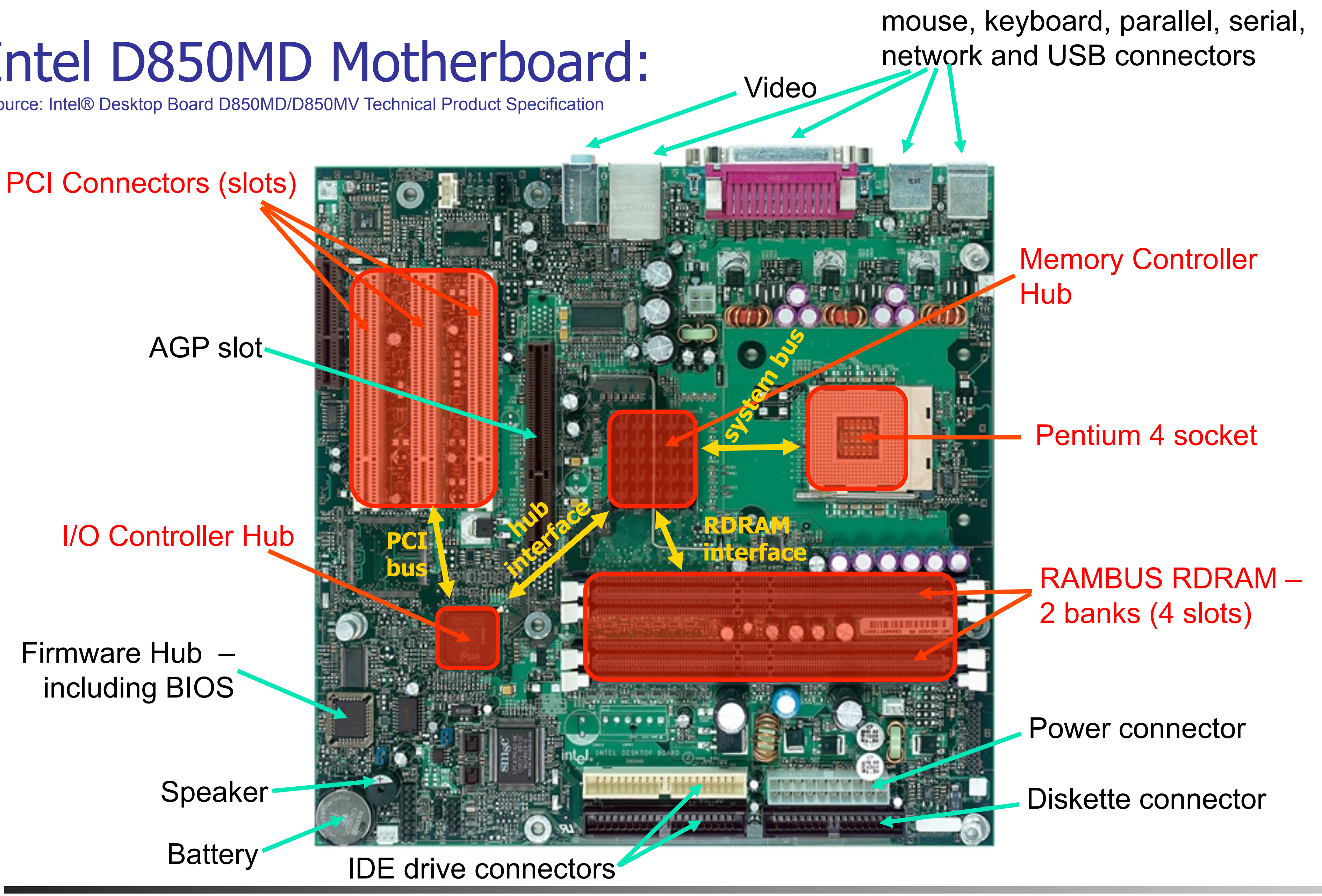
Source: Intel® Desktop Board D850MD/D850MV Technical Product Specification



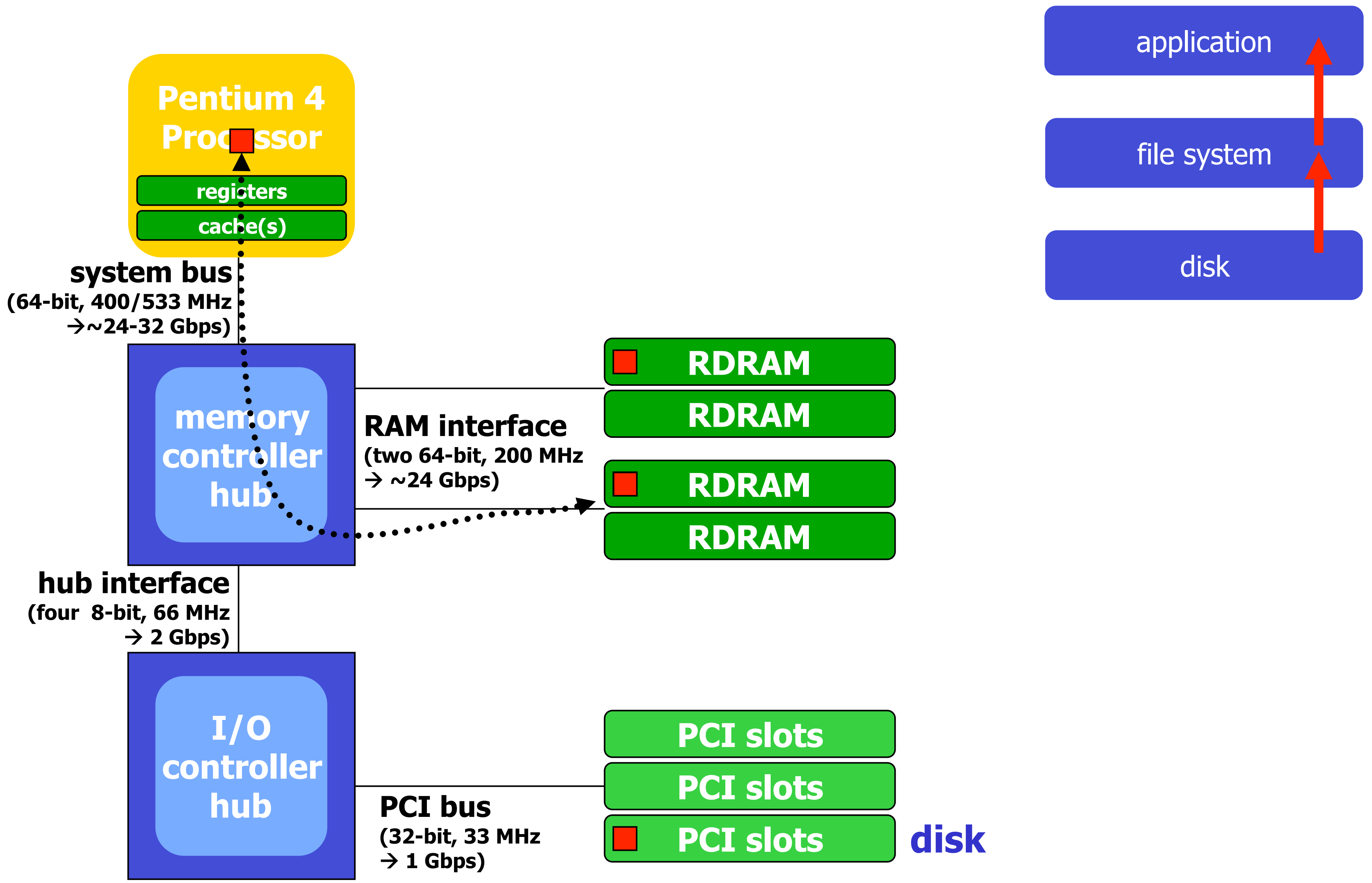
Example:
Intel Hub Architecture (850 Chipset)

Intel D850MD Motherboard:

Source: Intel® Desktop Board D850MD/D850MV Technical Product Specification

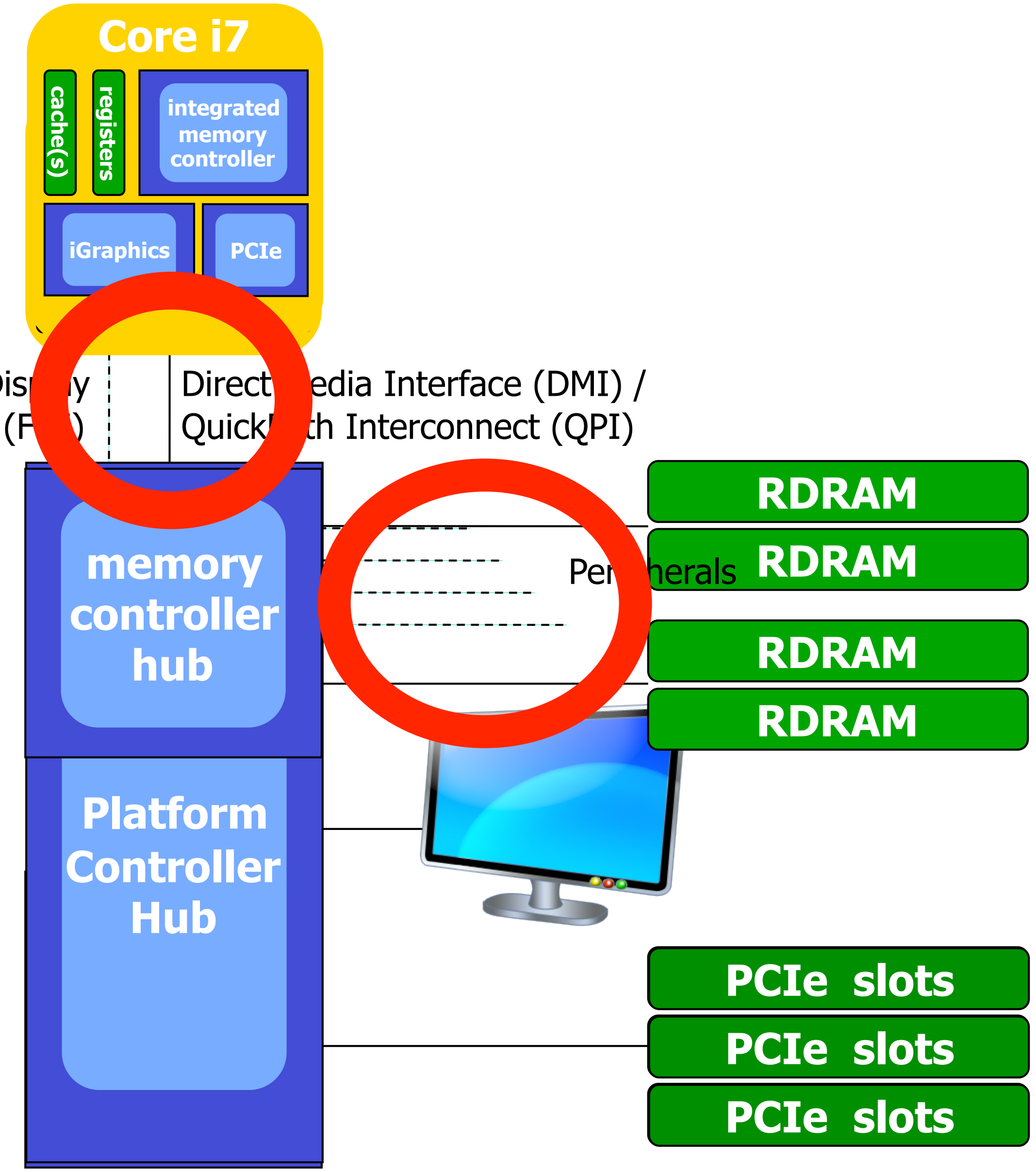


Example:
Intel Hub Architecture (850 Chipset)



Example:
Intel Platform Controller Hub Architecture

Sandy Bridge



Intel 32-bit Architecture (IA32): Basic Execution Environment

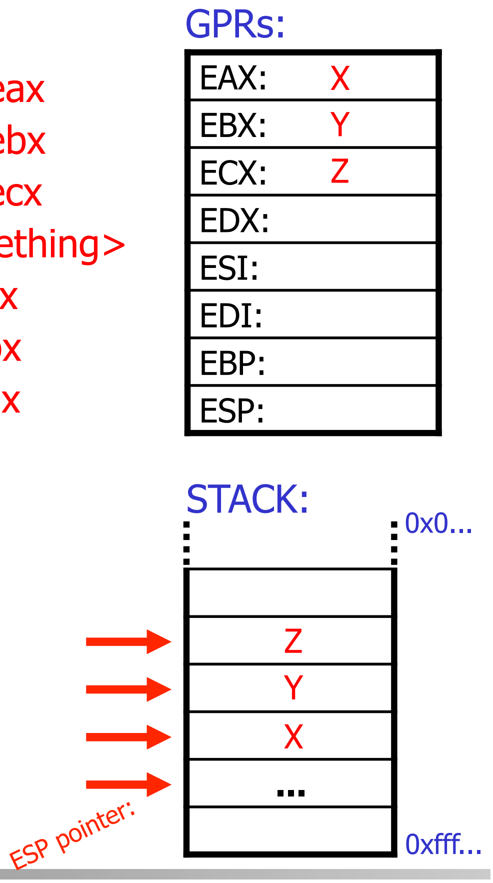
- Address space: 1 – 2^{36} (64 GB),
  each process may have a linear address space of 4 GB (2^{32})

- Basic program execution registers:
  - 8 general purpose registers (data: EAX, EBX, ECX, EDX, address: ESI, EDI, EBP, ESP)
  - 6 segment registers (CS, DS, SS, ES, FS and GS)
  - 1 flag register (EFLAGS)
  - 1 instruction pointer register (EIP)

- Stack – a continuous array of memory locations
  - Current stack is referenced by the SS register
  - ESP register – stack pointer
  - EBP register – stack frame base pointer (fixed reference)
  - PUSH – stack grows, add item (ESP decrement)
  - POP – remove item, stack shrinks (ESP increment)

- Several other registers like Control, MMX/FPU, Memory Type Range Registers (MTRRs), SSE*x* *(XMM)*, performance monitoring, …

```
PUSH %eax
PUSH %ebx
PUSH %ecx
<do something>
POP %ecx
POP %ebx
POP %eax
```



Intel 32-bit Architecture (IA32): Basic Execution Environment

- Example:

```
main (void)
{
    int a = 4, b = 2, c = 0;
    c = a + b;
}
```

objdump -d

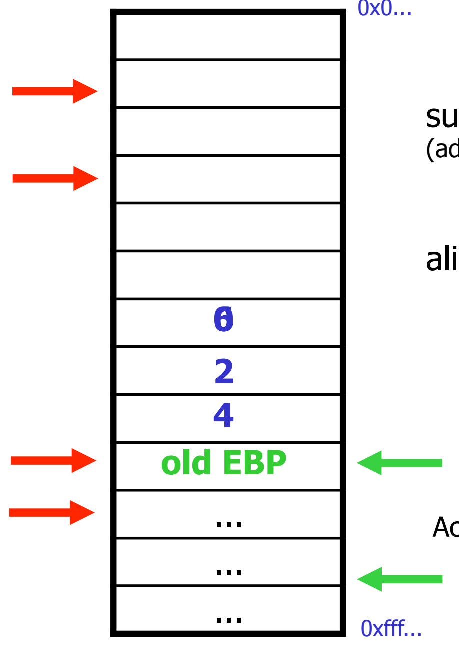
code segment:

```
...
8048314 <main>:
8048314:   push   %ebp
8048315:   mov    %esp,%ebp
8048317:   sub    $0x18,%esp
804831a:   and    $0xfffffff0,%esp
804831c:   mov    $0x0,%eax
8048322:   sub    %eax,%esp
8048324:   movl   $0x4,0xfffffffc(%ebp)
804832b:   movl   $0x2,0xfffffff8(%ebp)
8048332:   movl   $0x0,0xfffffff4(%ebp)
8048339:   mov    0xfffffff8(%ebp),%eax
804833c:   add    0xfffffffc(%ebp),%eax
804833f:   mov    %eax,0xfffffff4(%ebp)
8048342:   leave
8048343:   ret
...
```

insert value 4 in variable a on stack:
0xfffffffc = -(0xffffffff – 0xfffffffc) = **-0x4**

a's memory address = EBP - 4

stack:



sub 24 (0x18) bytes
(add space for 24 bytes)

alignment – sub "X" (here 8) bytes

Accumulator for operands and results data — EAX: 0

EBP: 0xfffffff0 — Pointer to data on stack (base)

Stack pointer — ESP: 0xffffffd8

EPI: 8048339 — Pointer to next instruction to be executed

C Function Calls & Stack

- A calling function does
  - push the parameters into stack in reverse order
  - push return address (current EIP value) onto stack

- When called, a C function does
  - push frame pointer (EBP) into stack - saves frame pointer register and gives easy return if necessary
  - let frame pointer point at the stack top, i.e., point at the saved stack pointer (EBP = ESP)
  - shift stack pointer (ESP) upward (to lower addresses) to allocate space for local variables

- When returning, a C function does
  - put return value in the return value register (EAX)
  - copy frame pointer into stack pointer - stack top now contains the saved frame pointer
  - pop stack into frame pointer (restore), leaving the return program pointer on top of the stack
  - the RET instruction pops the stack top into the program counter register (EIP), causing the CPU to execute from the "return address" saved earlier

- When returned to calling function, it does
  - copy the return value into right place
  - pop parameters – restore the stack

C Function Calls & Stack

- Example:

```
int add (int a, int b)
{
    return a + b;
}

main (void)
{
    int
    c =
}
```

objdump -d

1. Pop return instruction pointer into the EIP register
2. Release parameters (ESP)
3. Resume caller execution

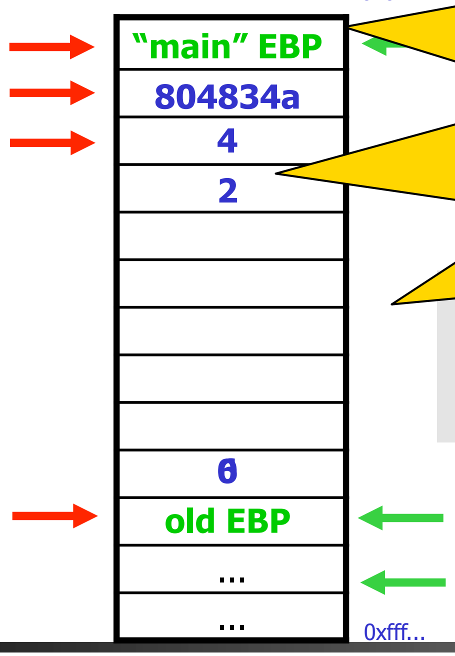
code segment:

```
…
8048314 <add>:
8048314:   push    %ebp
8048315:   mov     %esp,%ebp
8048317:   mov     0xc(%ebp),%eax
804831a:         d  0x8(%ebp),%eax
8048        p       %eb
8           t



                      p
                  , %esp
                  ,%esp
                    $0x0,%eax
      2d:  sub      %eax,%esp
     32f:  movl     $0x0,0xfffffffc(%ebp)
8048336:   movl     $0x2,0x4(%esp)
804833e:   movl     $0x4,(%esp)
8048345:   call     8048314 <add>
804834a:   mov      %eax,0xfffffffc(%ebp)
804834d:   leave
804834e:   ret
804834f:   nop
…
```

Wouldn't it be nice if this could be automatically managed!!??
→ **operating system**

1. Push EIP
2. Loads the set of the called procedure in the EIP register
3. Begin execution

stack:

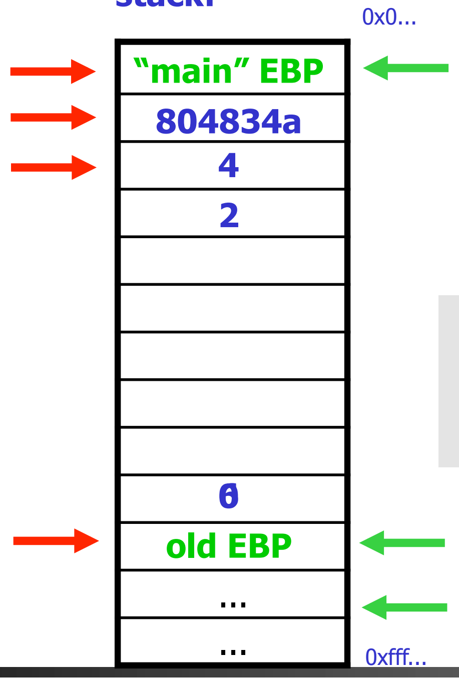


C Function Calls & Stack

extra copy for handout

```
int add (int a, int b)
{
    return a + b;
}

main (void)
{
    int c = 0;
    c = add(4 , 2);
}
```

stack:



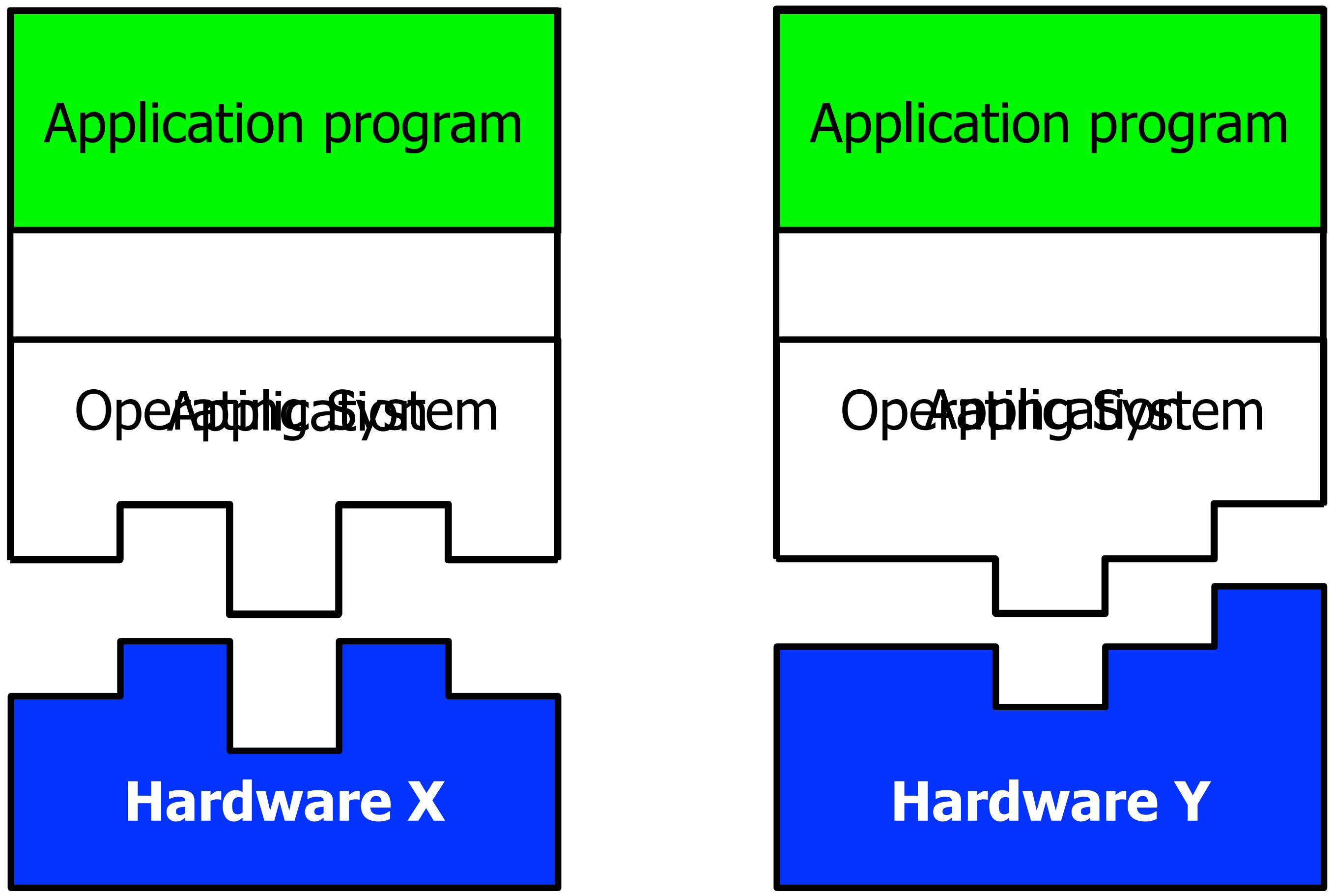
1. Pop return instruction pointer into the EIP register
2. Release parameters (ESP)
3. Resume caller execution

1. Push EIP register
2. Loads the offset of the called procedure in the EIP register
3. Begin execution

code segment:

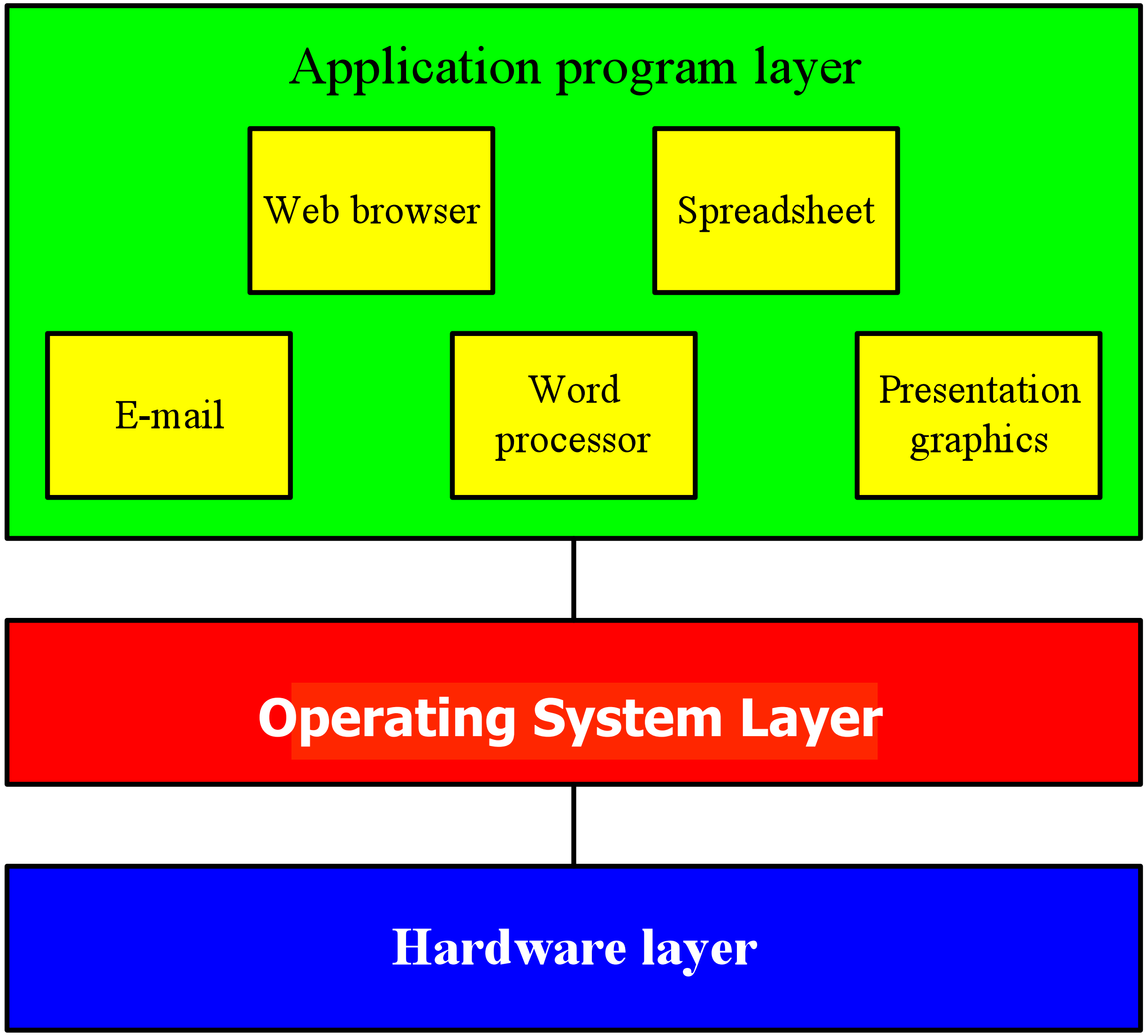
```
…
8048314 <add>:
8048314:  push   %ebp
8048315:  mov    %esp,%ebp
8048317:  mov    0xc(%ebp),%eax
804831a:  add    0x8(%ebp),%eax
804831d:  pop    %ebp
804831e:  ret
804831f <main>:
804831f:  push   %ebp
8048320:  mov    %esp,%ebp
8048322:  sub    $0x18,%esp
8048325:  and    $0xfffffff0,%esp
8048328:  mov    $0x0,%eax
804832d:  sub    %eax,%esp
804832f:  movl   $0x0,0xfffffffc(%ebp)
8048336:  movl   $0x2,0x4(%esp)
804833e:  movl   $0x4,(%esp)
8048345:  call   8048314 <add>
804834a:  mov    %eax,0xfffffffc(%ebp)
804834d:  leave
804834e:  ret
804834f:  nop
…
```

Different Hardware



Many Concurrent Tasks

- Better utilization
  - many concurrent processes
    - performing different tasks
    - using different parts of the machine
  - many concurrent users
- Challenges
  - "concurrent" access
  - protection/security
  - fairness
  - …

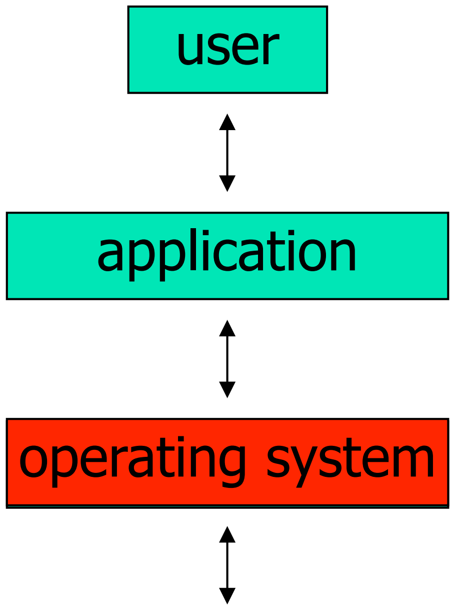


What is an Operating System (OS)?

- *"An operating system (OS) is a collection of programs that acts as an intermediary between the hardware and its user(s), providing a high-level interface to low level hardware resources, such as the CPU, memory, and I/O devices. The operating system provides various facilities and services that make the use of the hardware convenient, efficient and safe"*

Lazowska, E. D.: Contemporary Issues in Operating Systems , in: Encyclopedia of Computer Science, Ralston, A., Reilly, E. D. (Editors), IEEE Press, 1993, pp.980

- It is an extended machine (top-down view)
  - Hides the messy details
  - Presents a virtual machine, easier to use
- It is a resource manager (bottom-up view)
  - Each program gets time/space on the resource



Where do we find OSes?

Computers

Phones

Game Boxes

Cars

cameras,
other vehicles/crafts,
set-top boxes,
watches,
sensors,
…

Operating System Categories

- Single-user, single-task:
  historic, and rare (only a few PDAs use this)
- Single-user, multi-tasking:
  PCs and workstations may be configured like this
- Multi-user, multi-tasking:
  used on large, old mainframes; and handhelds, PCs, workstations and servers today
- Distributed OSes:
  support for administration of distributed resources
- Real-time OSes:
  support for systems with real-time requirements like cars, nuclear reactors, etc.
- Embedded OSes:
  built into a device to control a specific type of equipment like cellular phones, micro waves, etc.

History

- OSes have evolved over the last 60 years

- Early history ('40s and early '50s):
  - first machines did not include OSes
  - programmed using mechanical switches or wires

- Second generation ('50s and '60s):
  - transistors introduced in mid-'50s
  - batch systems
  - card readers

History

- Third generation (mid-’60s to the ’80s)
  - integrated circuits and simple multiprogramming
  - timesharing
  - graphical user interface
  - UNIX (’69-’70)
  - BSD (’77)
- Newer times (’80s to present)
  - personal computers & workstations
  - MS-DOS (’82), Win (’85), Minix (’87), Linux (’91), Win95, …

Why Study OSes?

- Understand how computers work under the hood
  - "you need to understand the system at all abstraction levels or you don't" (Yale Patt)
  - ⇨ Easier to do things right and efficient if one knows what happens

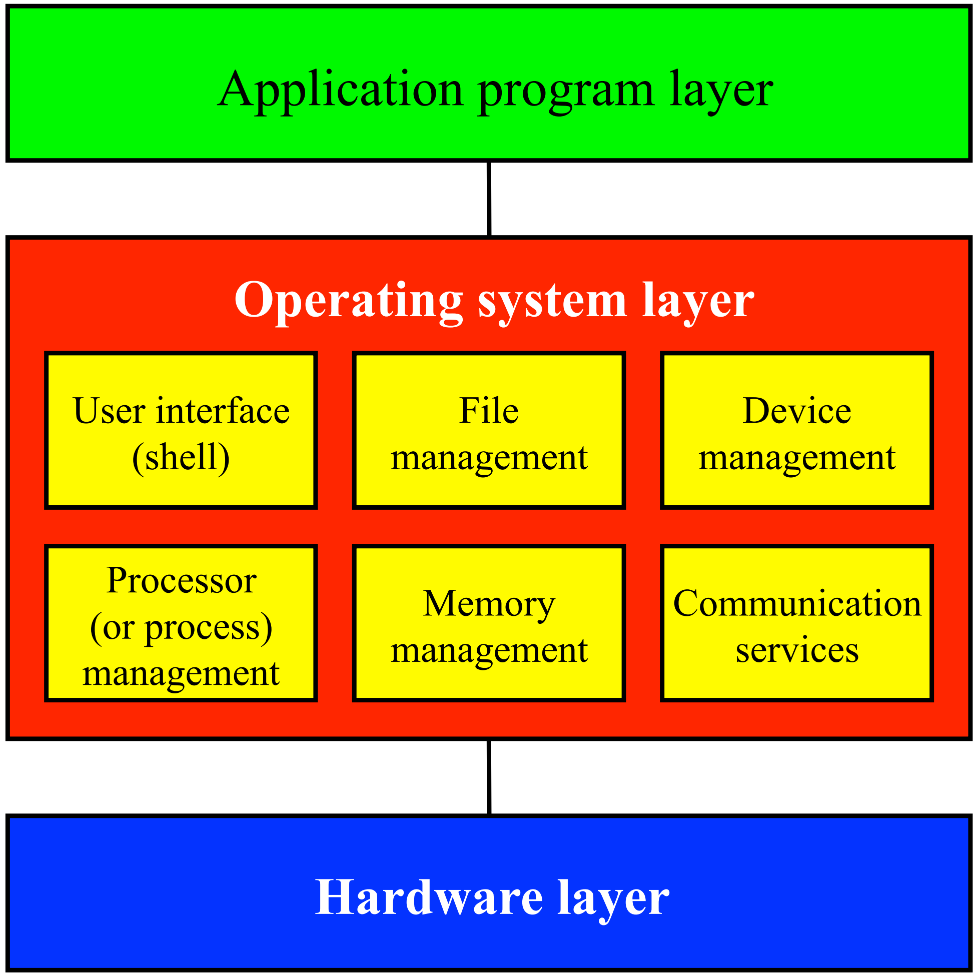
- Magic to provide infinite CPU cycles, memory, devices and networked computing

- Tradeoffs between performance and functionality, division of labor between HW and SW

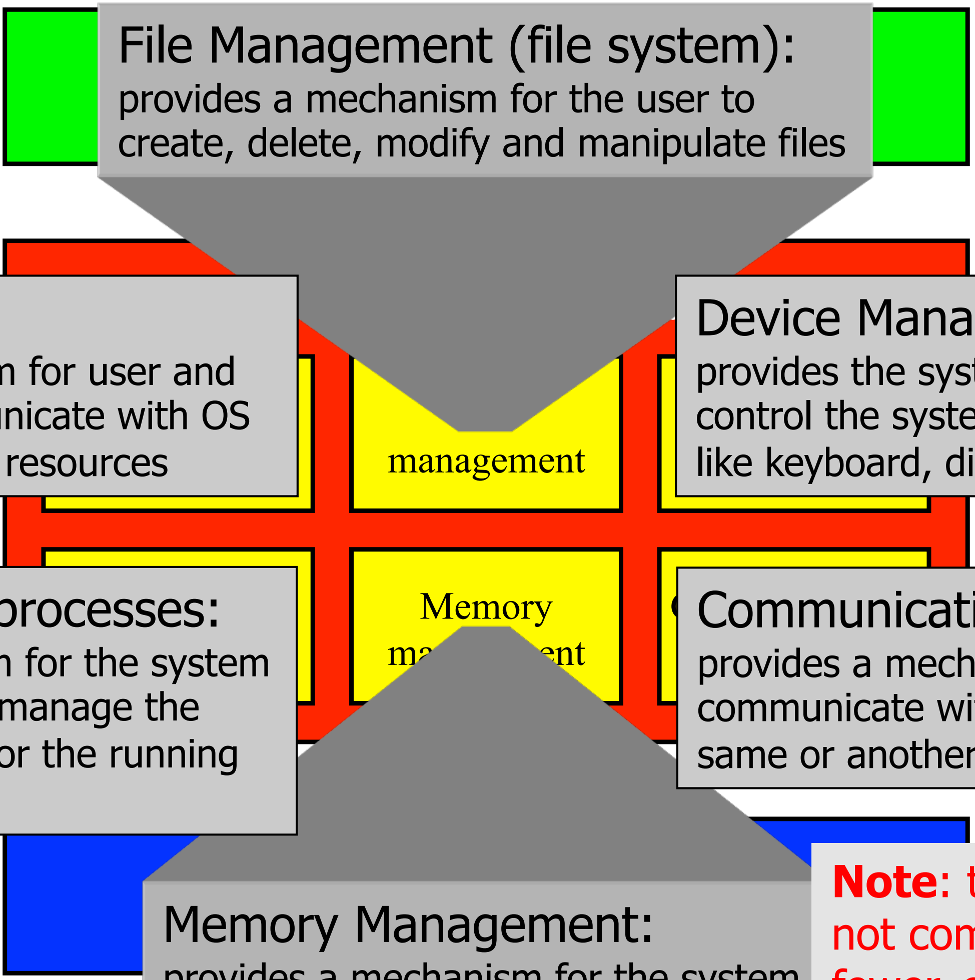
- *An OS is a key component in many systems*

Primary Components

- "Visible" to user
  - Shell
  - File system
  - Device management
- "Transparent"
  - Processor management
  - Memory management
  - Communication services



Primary Components



File Management (file system):
provides a mechanism for the user to create, delete, modify and manipulate files

User Interface:
provides a mechanism for user and application to communicate with OS and use the machine resources

Device Management:
provides the system with means to control the systems peripheral devices like keyboard, display, printer and disk

Management of processes:
provides a mechanism for the system to efficiently and fair manage the machine CPU cycles for the running processes

Communication:
provides a mechanism for the system communicate with other processes (on same or another machine)

Memory Management:
provides a mechanism for the system to efficiently manage the system's memory recourses – allocating space to processes

Note: this list of components is not complete. Some OSes have fewer, others more. Some have sub-components

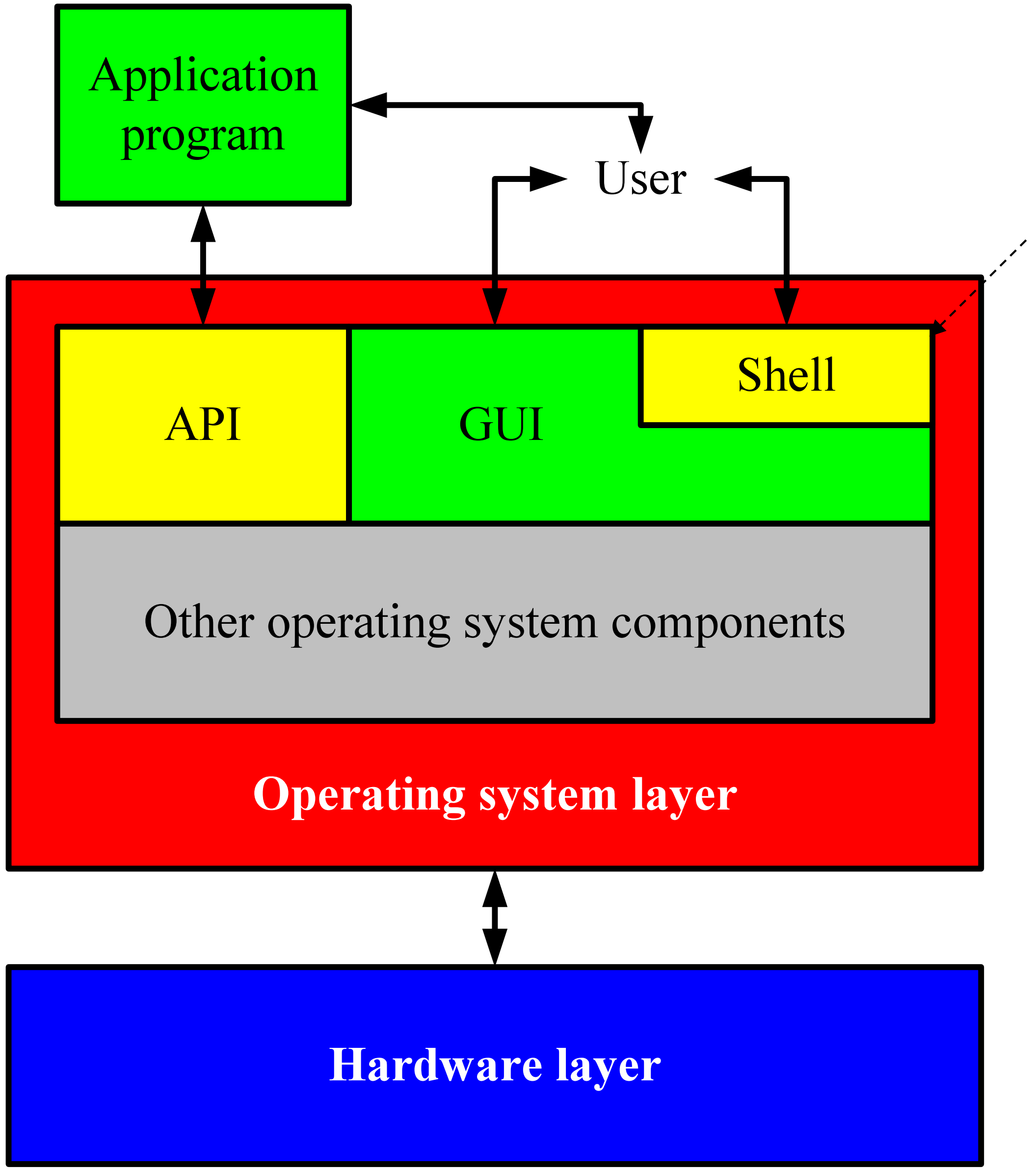
Device Management

- The OS must be able to control pheripal devices such as disk, keyboard, network cards, screen, speakers, mouse, memory sticks, ...
- Device controllers often have registers to hold status, give commands, ...
- Each type is different and requires device-spesific software
- The software talking to the controller and giving commands is often called a device driver
  - usually running within the kernel
  - mostly provided by the device vendors
  - translating device-independent commands, e.g.,
    read from file on disk: logical block number → device, cylinder, head, sector(s)
- A huge amount of code (95% of the Linux code!!??)

Interfaces

- A point of connection between components
- The OS incorporates logic that support interfaces with both hardware and applications, e.g.,
  - command line interface, e.g., a shell
  - graphical user interface (GUI)
    - interface consisting of windows, icons, menus and pointers
    - often not part of the OS (at least not kernel), but an own program
  - …
- Example: X (see `man X`)
  - network transparent window system running on most ANSI C and POSIX (portable OS interface for UNIX) compliant systems
  - uses inter-process communication to get input from and send output to various client programs
  - xdm (X Display Manager) – usually set by administrator to run automatically at boot time
  - xinit – manually starting X (`startx`, `x11`, `xstart`, …)

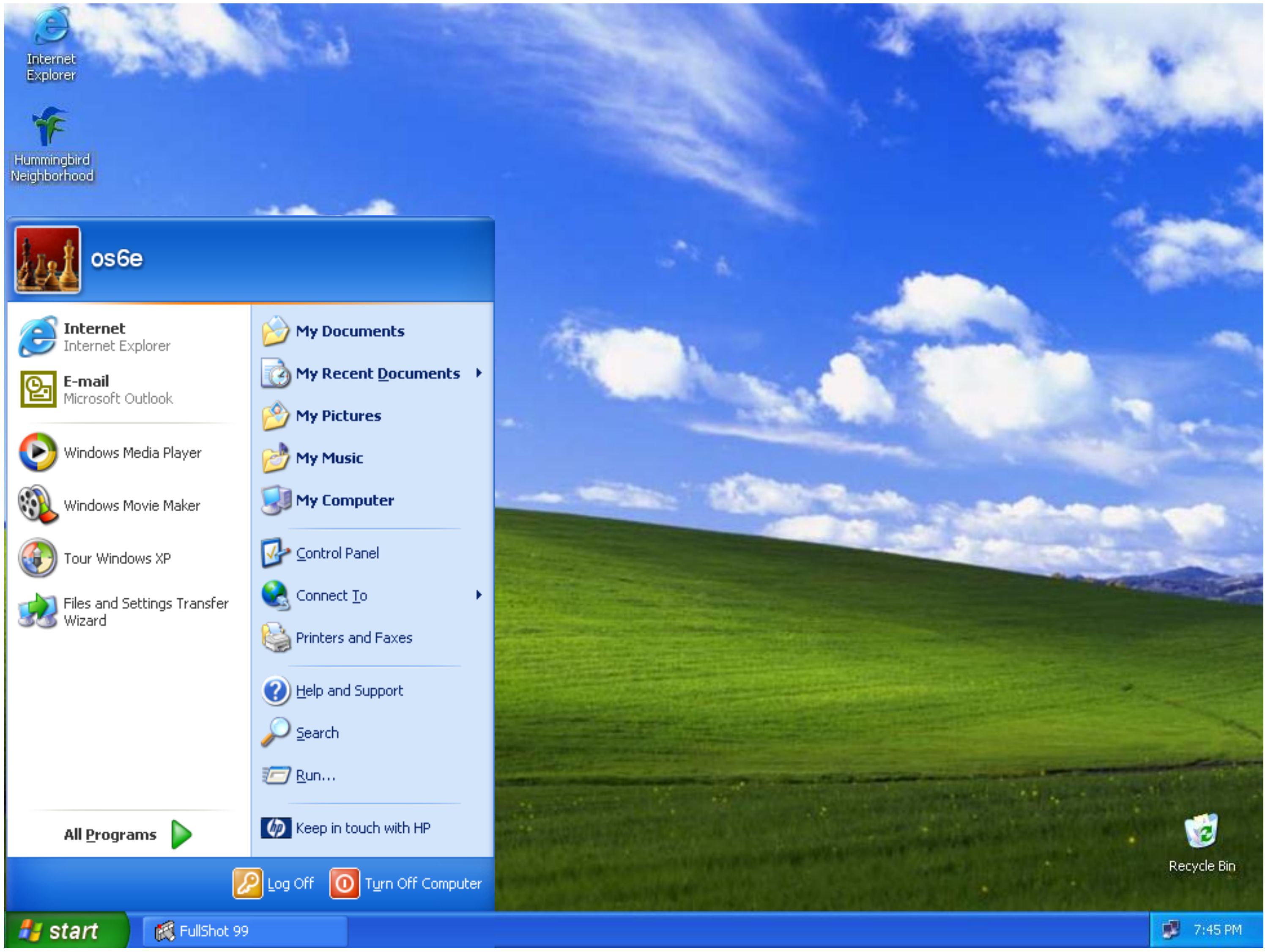
Windows Interfaces



The GUI incorporates a command line shell similar to the MS-DOS interface

Applications access HW through the API consisting of a set of routines, protocols and other tools

The WinXP Desktop Interface



Start button

Taskbar

Notification area

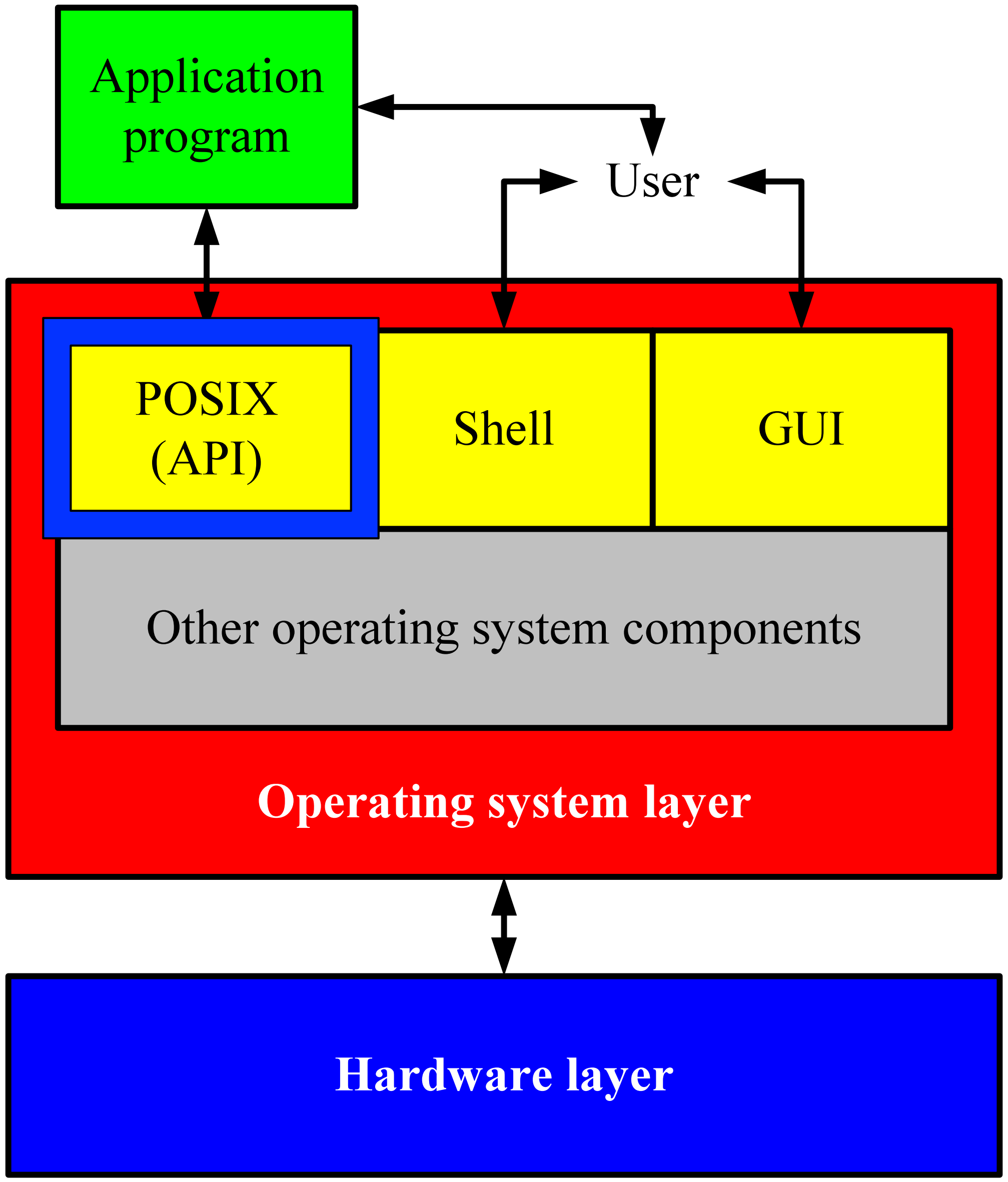
UNIX Interfaces

Applications are accessed HW through the API consisting of a set of routines, protocols and other tools (e.g., POSIX – portable OS interface for UNIX)

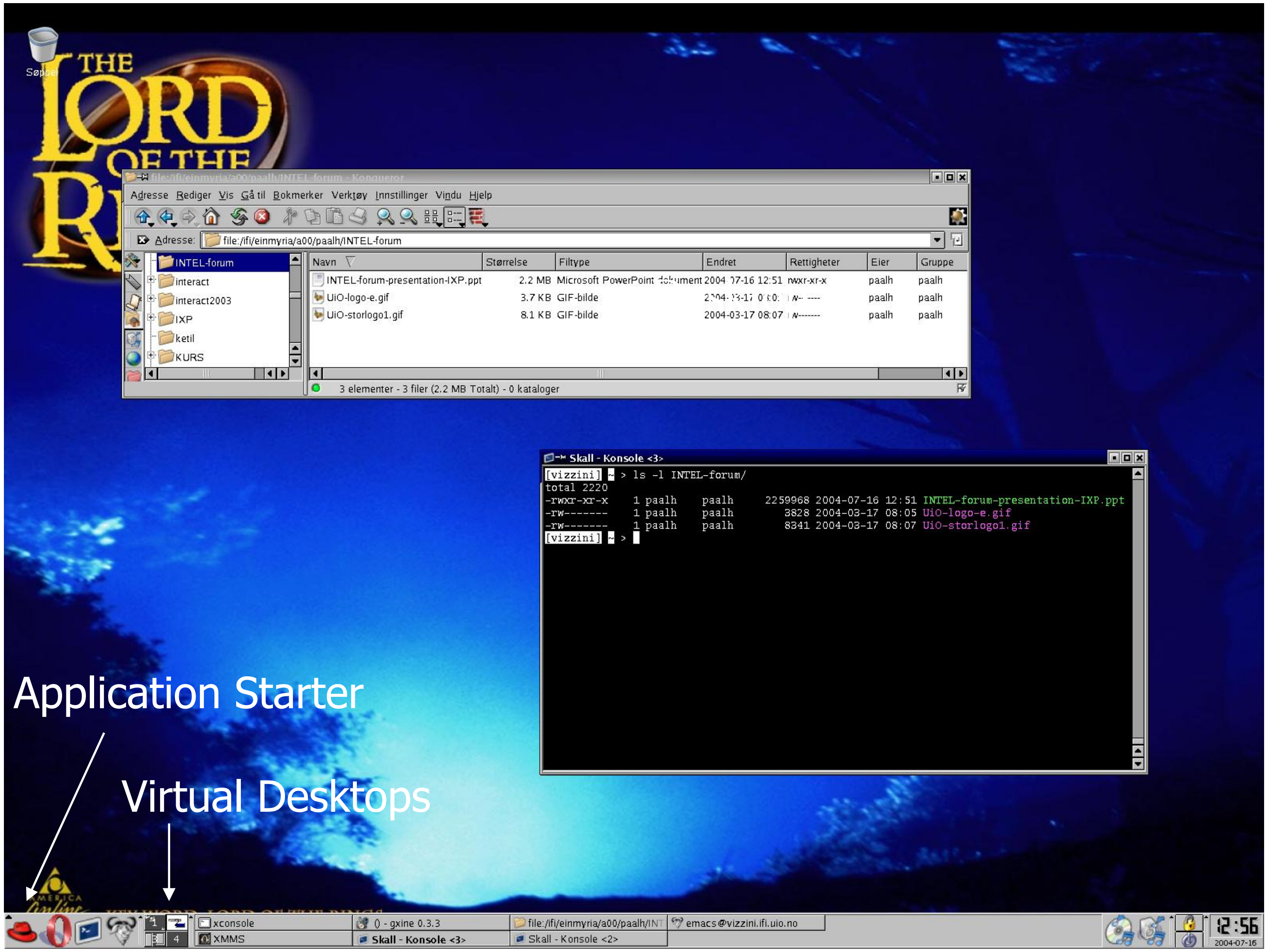
A user can interact with the system through the application interface or using a command line prosessed by a shell (not really a part of the OS)

A plain command line interface may be hard to use. Many UNIX systems therefore have a standard graphical interface (X Windows) which can run a desktop system (like KDE, Gnome, Fvwm, Afterstep, …)

Windows is more or less similar…



A Linux (KDE) Desktop Interface



file:/ifi/einmyria/a00/paalh/INTEL-forum

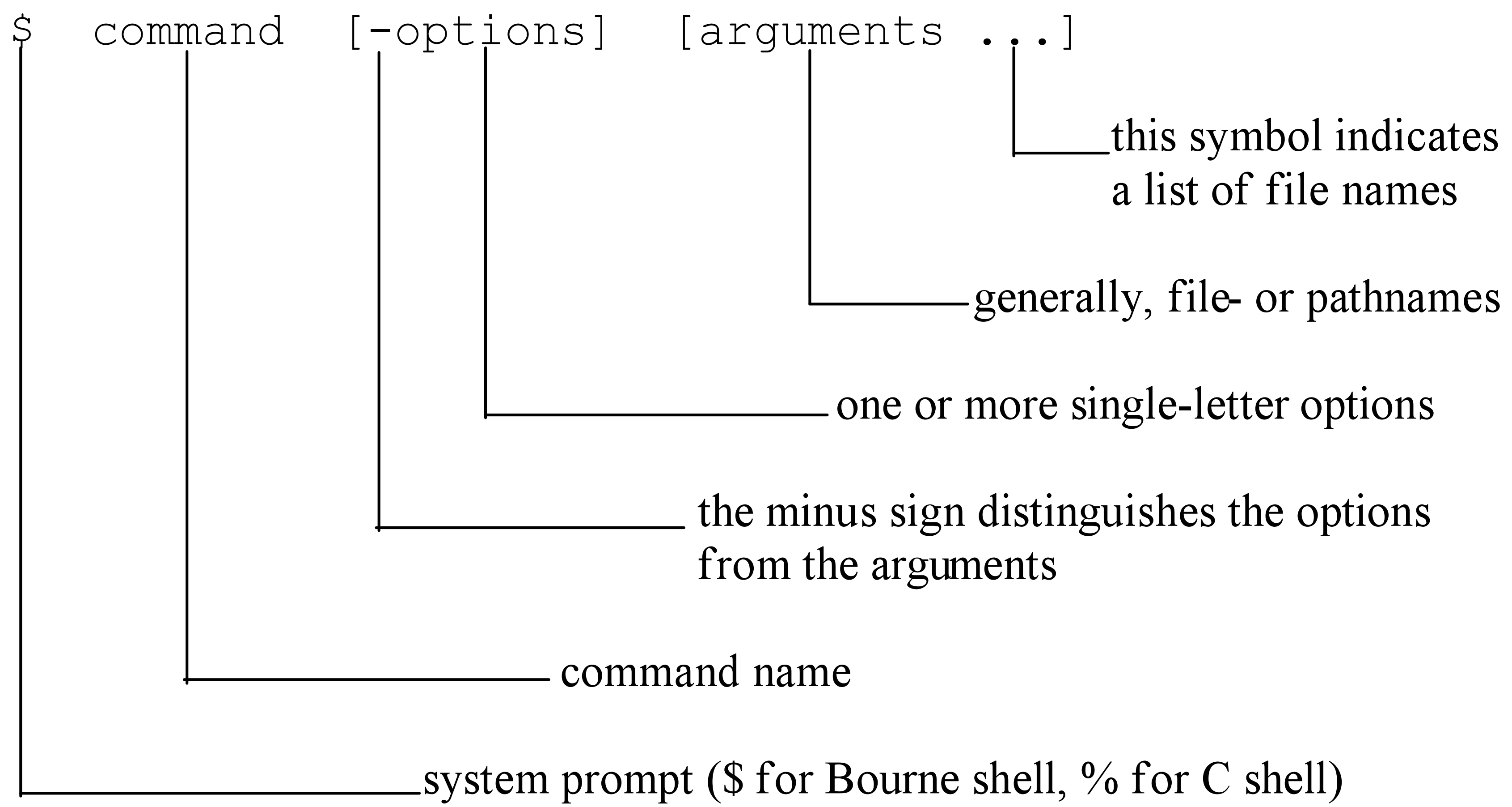
Navn	Størrelse	Filtype	Endret	Rettigheter	Eier	Gruppe
INTEL-forum-presentation-IXP.ppt	2.2 MB	Microsoft PowerPoint dokument	2004-07-16 12:51	rwxr-xr-x	paalh	paalh
UiO-logo-e.gif	3.7 KB	GIF-bilde	2004-03-17 08:05	rw-------	paalh	paalh
UiO-storlogo1.gif	8.1 KB	GIF-bilde	2004-03-17 08:07	rw-------	paalh	paalh

3 elementer - 3 filer (2.2 MB Totalt) - 0 kataloger

Skall - Konsole <3>

```
[vizzini] ~ > ls -l INTEL-forum/
total 2220
-rwxr-xr-x    1 paalh    paalh     2259968 2004-07-16 12:51 INTEL-forum-presentation-IXP.ppt
-rw-------    1 paalh    paalh        3828 2004-03-17 08:05 UiO-logo-e.gif
-rw-------    1 paalh    paalh        8341 2004-03-17 08:07 UiO-storlogo1.gif
[vizzini] ~ > 
```

Typical (UNIX) Line Commands



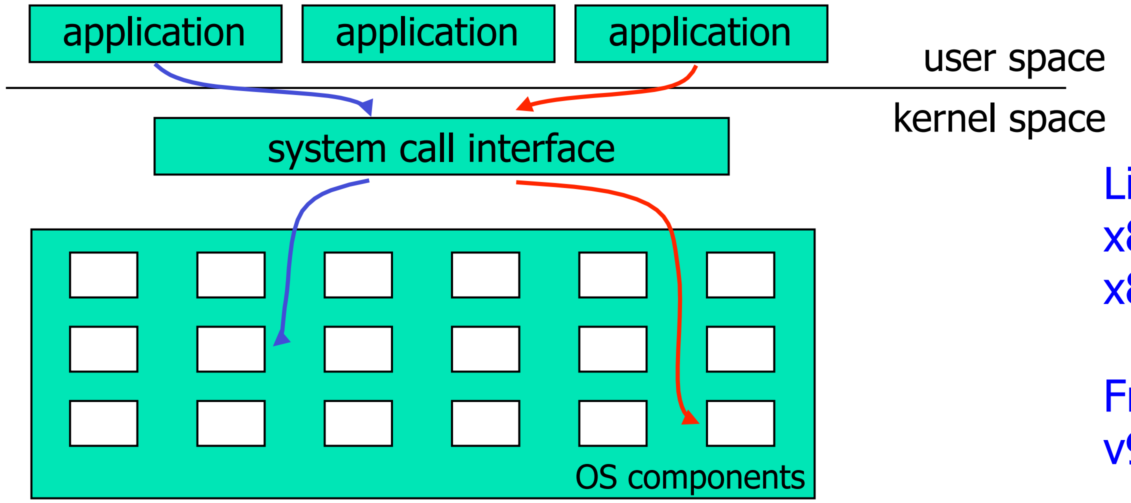
```
$   command   [-options]   [arguments ...]
```

this symbol indicates a list of file names
generally, file- or pathnames
one or more single-letter options
the minus sign distinguishes the options from the arguments
command name
system prompt ($ for Bourne shell, % for C shell)

System Calls

Linux system calls (2.4.19):

- The interface between the OS and users is defined by a set of system calls
- Making a system call is similar to a procedure/function call, but system calls enter the kernel:



[illegible]

Linux:
x86 v2.4.19 *entry.S* → **242**
x86 v3.0-rc4 *syscall_table_32.S* → **347**

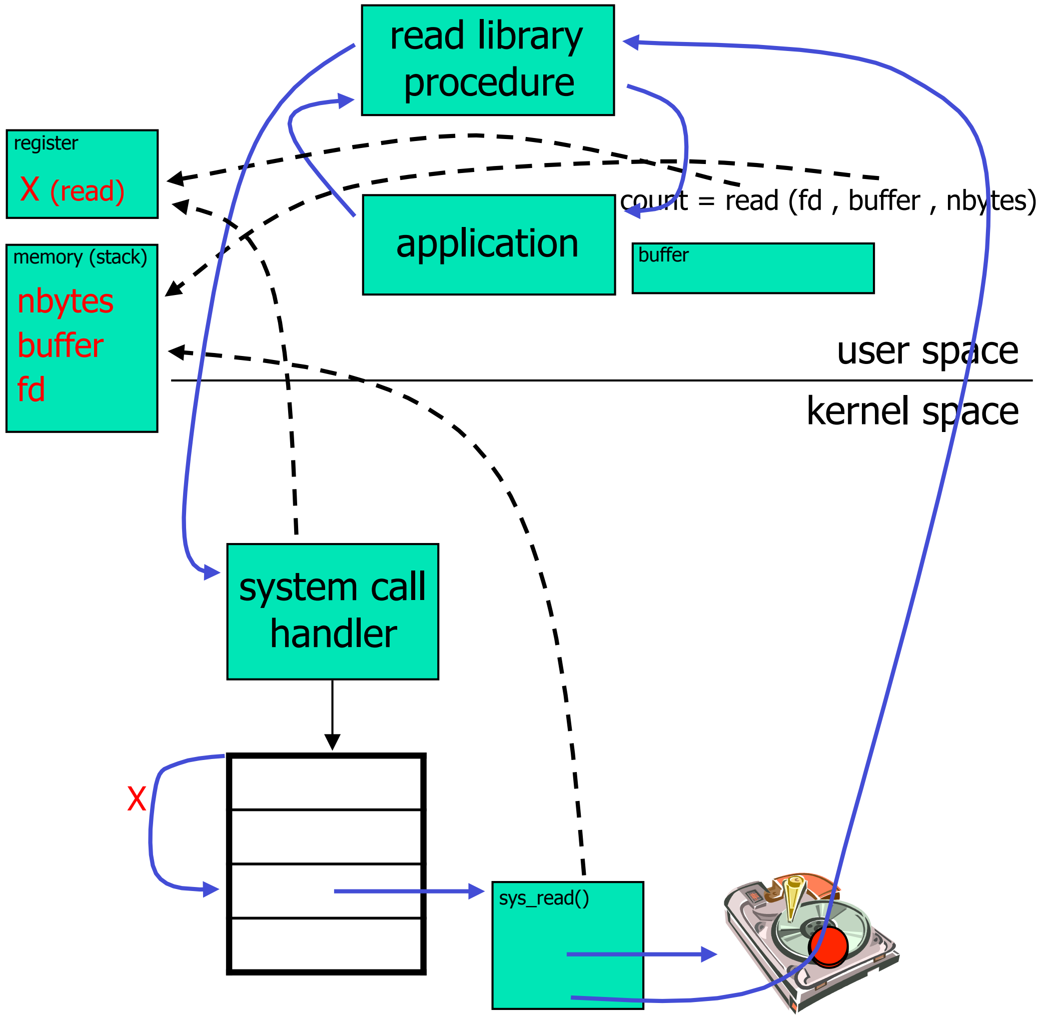
FreeBSD:
v9 *syscalls.c* → **531**

sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *addr)
sys_shmdt (char *shmaddr)

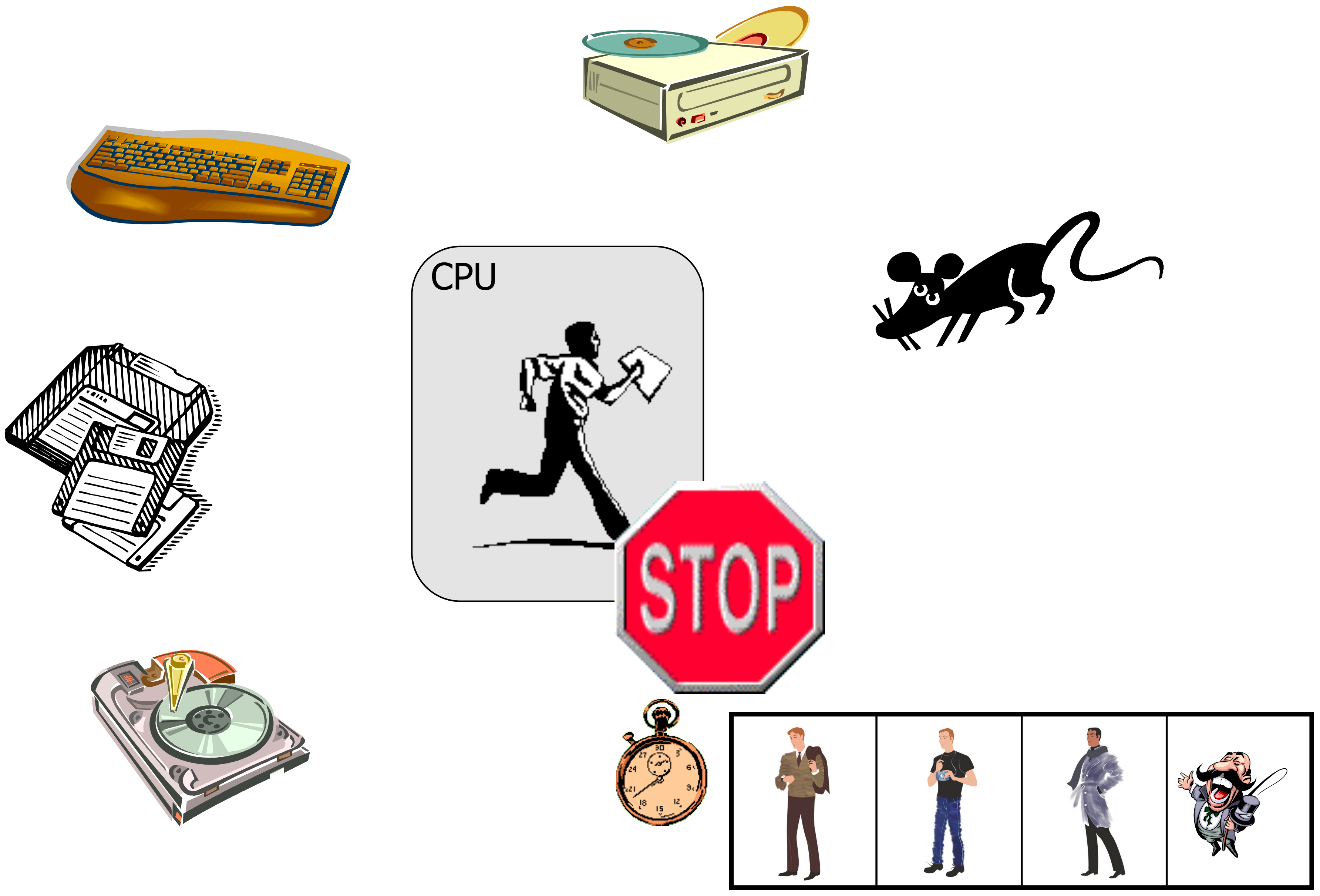
System Calls: `read`

- C example:
  `count = read(fd,buffer,nbyte)`

1. push parameters on stack
2. call library code
3. put system call number in register
4. call kernel (TRAP)
   - ✓ kernel examines system call number
   - ✓ finds requested system call handler
   - ✓ execute requested operation
5. return to library and clean up
   - ✓ increase instruction pointer
   - ✓ remove parameters from stack
6. resume process



Interrupt Program Execution



Interrupts

- Interrupts are electronic signals that (usually) result in a forced transfer of control to an interrupt handling routine
  - alternative to polling
  - caused by *asynchronous* events like finished disk operations, incoming network packets, expired timers, …
  - an interrupt descriptor table (IDT) associates each interrupt with a code descriptor (pointer to code segment)
  - can be disabled or masked out

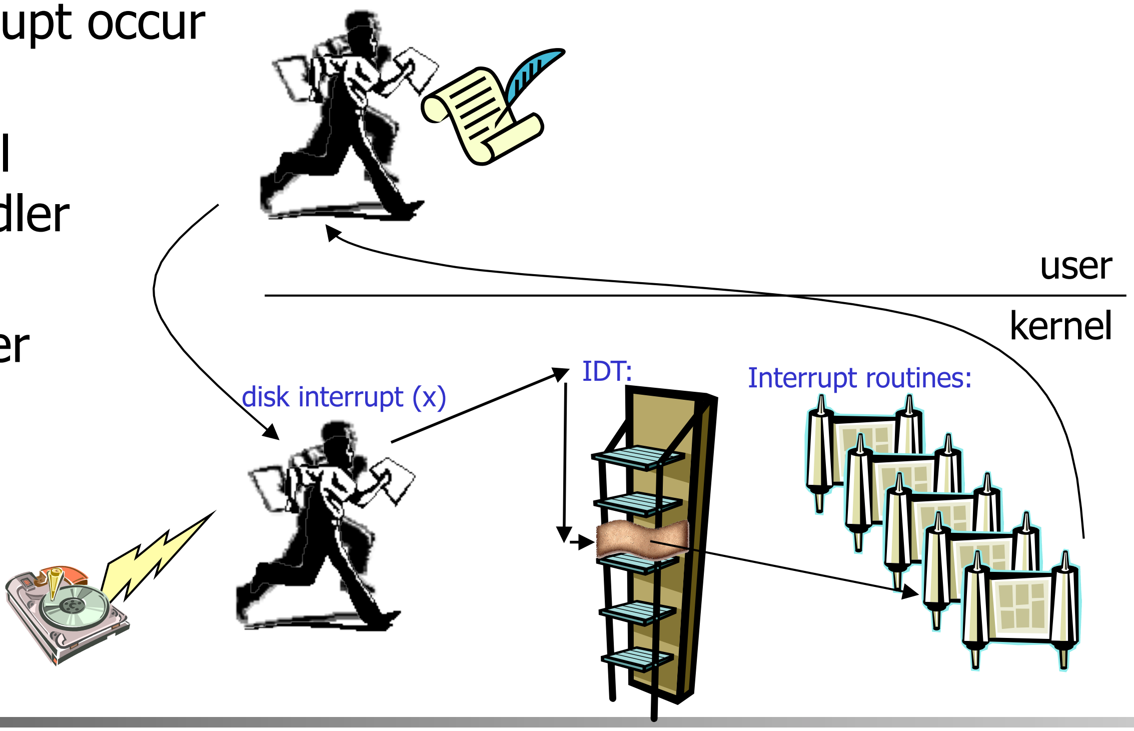
Exceptions

- Another way for the processor to interrupt program execution is exceptions
  - caused by *synchronous* events generated when the processor detects a predefined condition while executing an instruction
  - TRAPS: the processor reaches a condition the exception handler can handle (e.g., overflow, break point in code like making a system call, …)
  - FAULTS: the processor reaches a fault the exception handler can correct (e.g., division by zero, wrong data format, …)
  - ABORTS: terminate the process due to an unrecoverable error (e.g., hardware failure) which the process itself cannot correct
  - the processor responds to exceptions (i.e., traps and faults) essentially as for interrupts

Interrupt (and Exception) Handling

- The IA-32 has an interrupt description table (IDT) with 256 entries for interrupts and exceptions
  - 32 (0 - 31) predefined and reserved
  - 224 (32 - 255) is "user" (operating system) defined

- Each interrupt is associated with a code segment through the IDT and a unique index value giving management like this:
  1. process running while interrupt occur
  2. capture state, switch control and find right interrupt handler
  3. execute the interrupt handler
  4. restore interrupted process
  5. continue execution



Booting

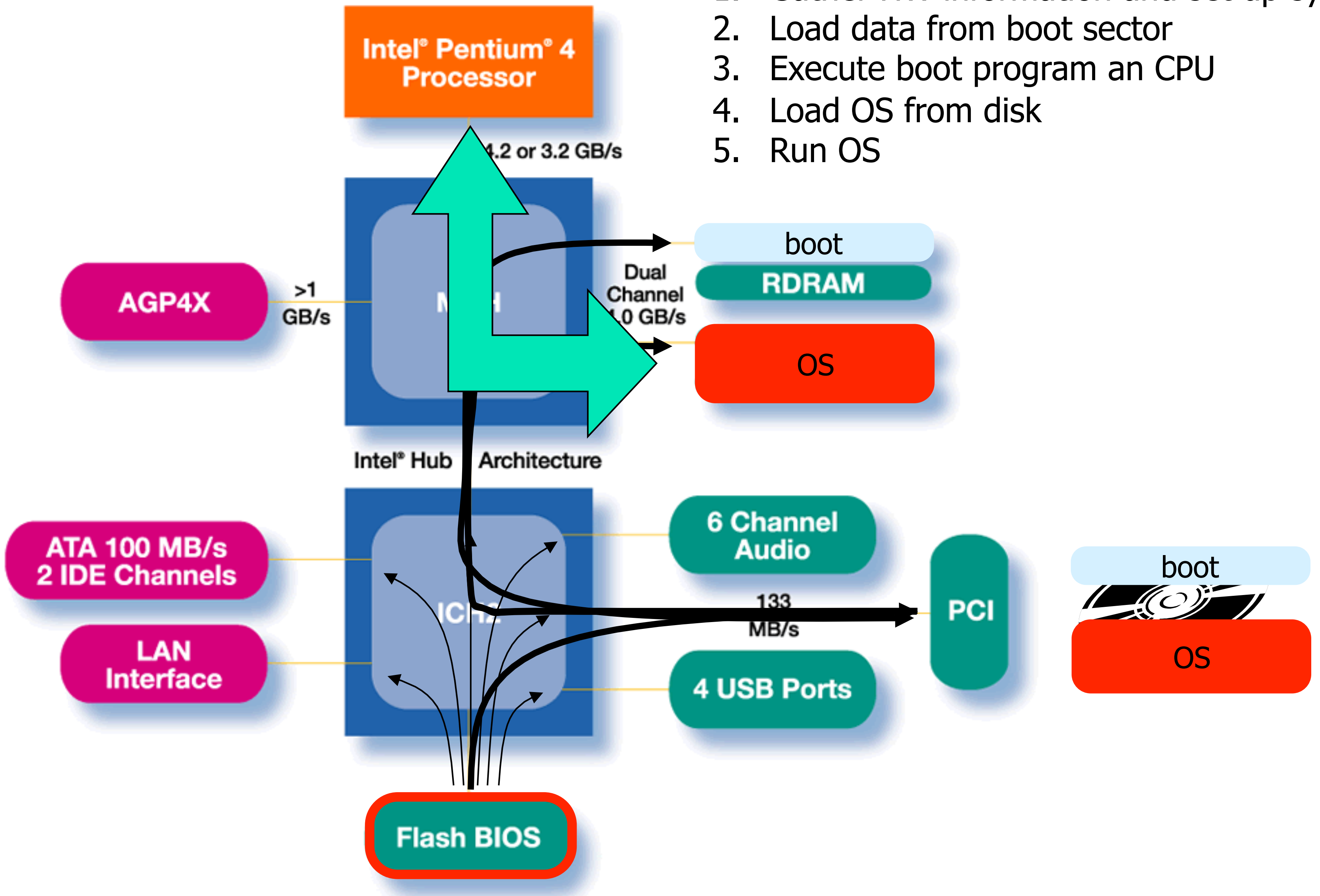
- Memory is a volatile, limited resource: OS usually on disk

- Most motherboards contain a basic input/output system (BIOS) chip (often flash RAM) – stores instructions for basic HW initialization and management, and initiates the …

- ... bootstrap: loads the OS into memory
  - read the `boot` program from a known location on secondary storage typically first sector(s), often called master boot record (MBR)
  - run `boot` program
    - read root file system and locate file with OS kernel
    - load kernel into memory
    - run kernel

Booting

1. Gather HW information and set up system
2. Load data from boot sector
3. Execute boot program an CPU
4. Load OS from disk
5. Run OS

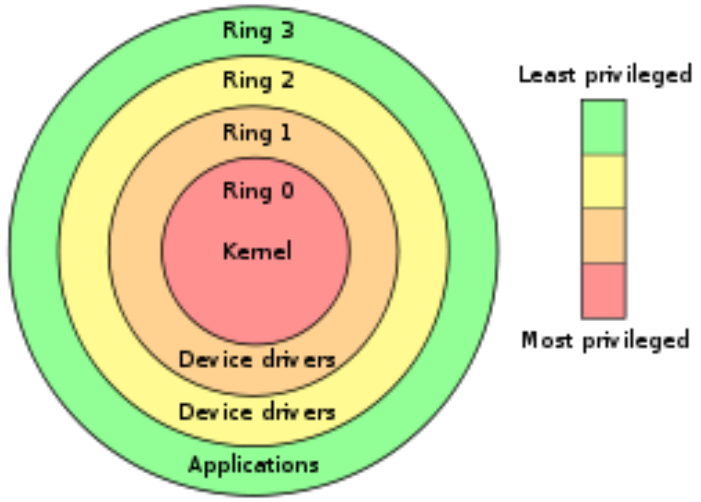


User Level vs. Kernel Level (Protection)

- Many OSes distinguish user and kernel level, i.e., due to security and protection

- Usually, applications and many sub-systems run in user mode (pentium level 3)
  - protected mode
  - not allowed to access HW or device drivers directly, only through an API
  - access to assigned memory only
  - limited instruction set

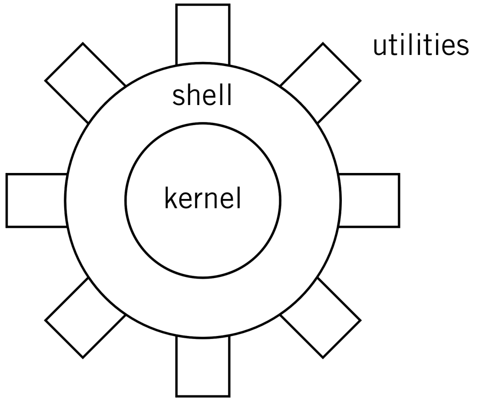
- OSes run in kernel mode (under the virtual machine abstraction, pentium level 0)
  - real mode
  - access to the entire memory
  - all instructions can be executed
  - bypass security



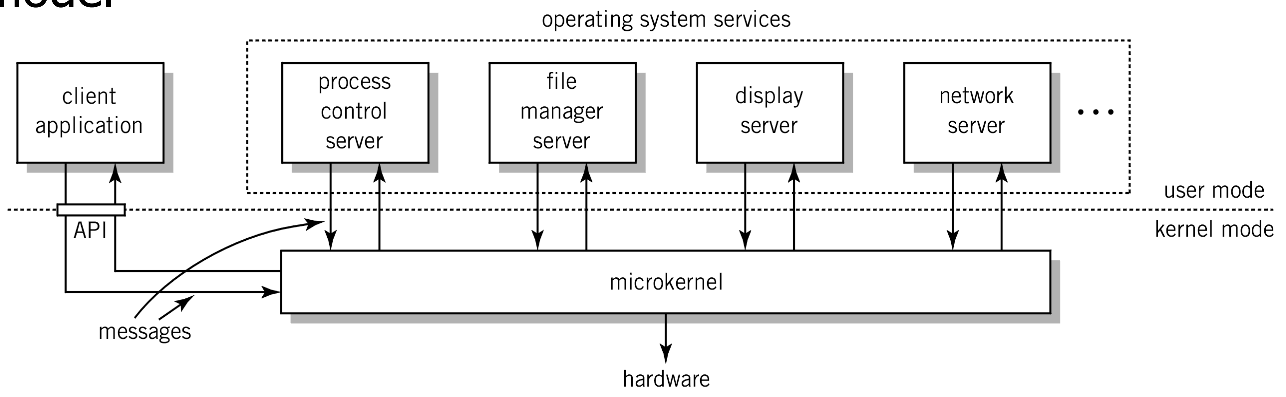
OS Organization

- No standard describing how to organize a kernel (as it is for compilers, communication protocols, etc.) and several approaches exist, e.g.:

- Monolithic kernels ("the big mess"):
  - written as a collection of functions linked into a single object
  - usually efficient (no boundaries to cross)
  - large, complex, easy to crash
  - UNIX, Linux, …



- Micro kernels
  - kernel with minimal functionality (managing interrupts, memory, processor)
  - other services are implemented in server processes running in user space used in a client-server model
  - lot of message passing (inefficient)
  - small, modular, extensible, portable, …
  - MACH, L4, Chorus, …



Summary

- OSes are found “everywhere” and provide virtual machines and work as a resource managers
- Many components providing different services
- Users access the services using an interface like system calls
- In the next lectures, we look closer at some of the main components and abstractions in an OS
  - processes management
  - memory management
  - storage management
  - local inter-process communication

  - inter-computer network communication is covered in the last part of the course