2 – Operating System Structures

2 – Operating System Structures

- Operating system components
- Structures, and various design methodologies.
- Booting an operating system.

From the last chapter ...

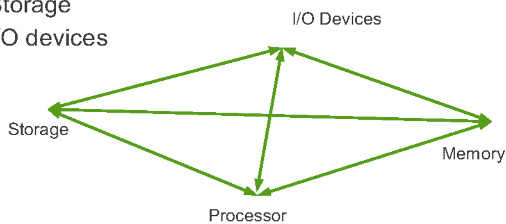
- Linus said "*... So, an operating system never does anything on itself. It's only **waiting** for programs to ask for certain resources or ask for certain files on the disk or ask for the program to connect them to the outside world. And **then the operating system come step in** and try to make it easy for people to write program.*"
- And, again, an operating system has to perform two important functions:
  - Manage and control computing resources (i.e., system hardware) to do whatever user's programs ask for.
  - Control execution of user's programs.

(cont'd.)

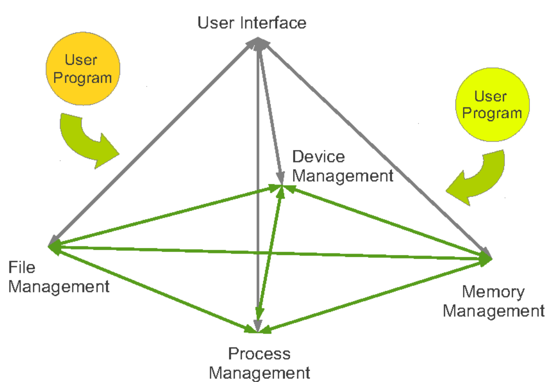
- Let's see how an operating system can be organized to do so.
- And, the start up of a computer system.

Computing Resources ?

- You have already learned about hardware components
  - Processors
  - Memory
  - Storage
  - I/O devices



Operating System Software



System Components

- Process management
- Memory management
- File management
- Device management
- User interfaces
- Networking
- Protection system

Process Management

- A program does nothing unless its instructions are executed by a CPU. So, a *process* can be thought of as a *program in execution*.
- A process needs certain resources, e.g., CPU time, memory, and I/O devices.
  - These resources can be given when the process is created, or allocated while it is running.
- Two processes may be associated with the same program, they are considered two separate execution sequences.
- A set of processes can potentially execute concurrently by multiplexing CPU among them.

(cont'd.)

- An operating system is responsible for many process activities, e.g.,
  - Creating and deleting both system and user processes.
  - Suspending and resuming processes.
  - Providing mechanisms for process synchronization.
  - Providing mechanisms for process communication.
  - Providing mechanisms for deadlock handling.

Memory Management

- For a program to be executed, it must be mapped to *absolute addresses* and loaded into memory.
- As the program executes, it accesses instructions and data from memory by generating these absolute addresses.
- When the program terminates, its memory space is declared available, and the next program can be loaded and executed.

(cont'd.)

- Many memory-management schemes are available.
  - Effectiveness of the different algorithms depends on the particular situation.
  - Selection of a memory-management scheme for a specific system depends on many factors, especially on the hardware design.

(cont'd.)

- An operating system is responsible for the following memory-management activities:
  - Keep track of which parts of memory are currently being used and by whom.
  - Deciding which processes are to be loaded into memory when memory space becomes available
  - Allocating and deallocating memory space as needed.

File Management

- Computer can store information on several different types of physical media.
  - e.g., magnetic disk, magnetic tape, optical disk
  - Each of them has its own characteristics and physical organization.
  - Each medium is controlled by a device that also has unique characteristics.
    - e.g., access speed, capacity, transfer rate, and access method (sequential or random).
- An operating system provides a uniform logical view of information storage – a *file*.
  - The operating system maps files onto physical media, and accesses these files via the storage devices.

(cont'd.)

- A file is a collection of related information defined by its creator.
- Commonly, files represent programs and data.
- Files consist of a sequence of bits, bytes, lines, or records whose meaning are defined by their creators.
- Files are normally organized into directories to ease their use.
- When multiple users have access to the file, the operating system might be able to control by whom and in what way (e.g., read, write, append) files may be accessed.

(cont'd.)

- An operating system is responsible for the following file-management activities:
  - Creating and deleting files
  - Creating and deleting directories
  - Supporting primitives for manipulating files and directories
  - Mapping files onto secondary storage
  - Backing up files on stable (non-volatile) storage media

(cont'd.)

- Additionally, an operating system is responsible for disk management activities, e.g.,
  - Free-space management
  - Storage allocation
  - Disk scheduling

Device Management

- One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user.
  - e.g., UNIX I/O subsystem
- The I/O subsystem consists of
  - Memory management
    - Buffering, caching, spooling
  - General device-driver interface
  - Drivers for specific hardware devices

(cont'd.)

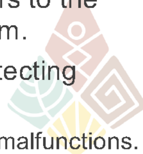
- An operating system is responsible for
  - Request and release device
  - Read, write, relocation
  - Get/set device attributes
  - Logical attach/detach devices

Networking

- In a distributed system, processors are connected through a communication network.
  - Heterogeneous
  - Different configurations
  - Fully or partially connected
- Operating systems usually generalize network access as a form of file access.
- Network protocol implementations are a part of modern operating systems.

Protection System

- In multiuser environment, a computer system allows multiple users to execute multiple processes concurrently.
  - File, memory, CPU and other resources must be operated on by only those processes that have gained proper authorization from the operating system.
- Protection is any mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.
- Protection can improve reliability by detecting errors of component subsystems.
  - Early detection can often prevent system malfunctions.



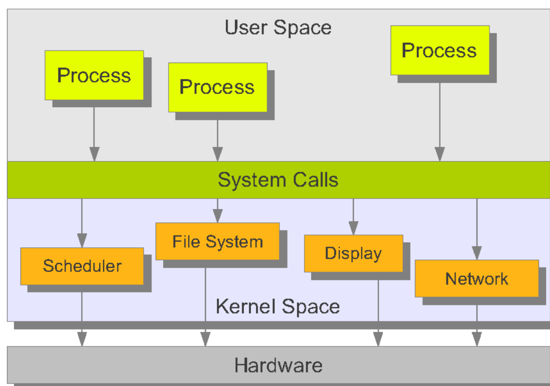
User Interfaces

- The interface between the user and the operating system.
- Usually, it is a special program that reads and interprets control statements.
  - Often known as the *shell*.
- The shell can be
  - Command-line interface
  - Graphical user interface

System Calls

- To provide computing resources and other services to user's programs, an operating system provides a set of APIs commonly known as *system calls*.
  - A collection of common operations.
- Pretty much similar to a procedure or function call, but the call enters *kernel space* (or executes in *kernel mode*).
- So, it splits the software layer into two (logical) spaces/mode
  - User space/mode ~ run user's programs.
  - Kernel space/mode ~ run operating system's code.

(cont'd.)



(cont'd.)

- This allows an operating system to get all requests from user's processes and manage how available resources can be provided.
  - Efficiency, consistency, stability, security.
- For developers, it is easier to develop programs.
  - They just need to call the syscall. Then, OS could handle the rest.

(cont'd.)

- Different operating systems have different sets of system calls.
  - Even UNIX ones.
- IEEE has published Std 1003.1-1988 POSIX to standardize Unix system calls
  - Was IEEE 1003 and ISO/IEC 9954.
  - There are extension standards, e.g.,
    - 1003.1b-1993 – Real-time extension
    - 1003.1c-1995 – Thread extension
  - This means source code of user's program can be portable.
    - Solaris, *BSD, Linux (through LSB), Mac OS X, etc.

(cont'd.)

- System calls are very *machine-dependent*.
  - Most, if not all, expressed in assembly code.
  - So, implementations can be (very) different.
    - Even those POSIX-compliant.
- System calls are generally available as assembly-language instructions.
- C, C++, Perl, etc. allow system calls to be made directly.
  - GNU C library
  - Win32 API

(cont'd.)

- System calls can be grouped roughly into the components:
  - Process management
    - load, execute, abort, ...
  - File management
    - create, delete, open, close, ...
  - Device management
    - read, write, ...
  - Information maintenance
    - date, time, system attributes, ..
  - Communication
    - send, receive, status, ...

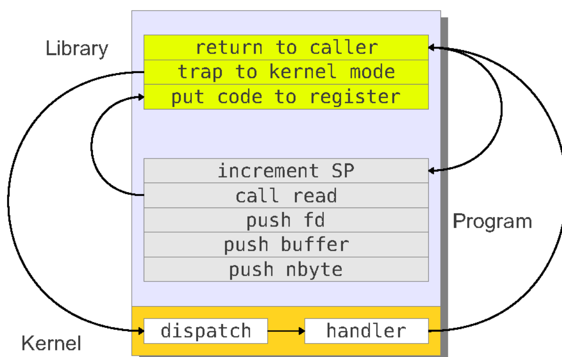
Example: `read` system call

```
count = read(fd, buffer, nbyte)
```

- Push parameters onto the stack
- Call the system call
- Put the code to register
- Execute TRAP to enter the kernel mode
- Start execution in the kernel mode
- Dispatch the system call handler
- Execute the handler
- Return to the caller
- Clean up the stack

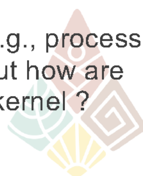
(cont'd.)

```
count = read(fd, buffer, nbyte)
```



System Structure

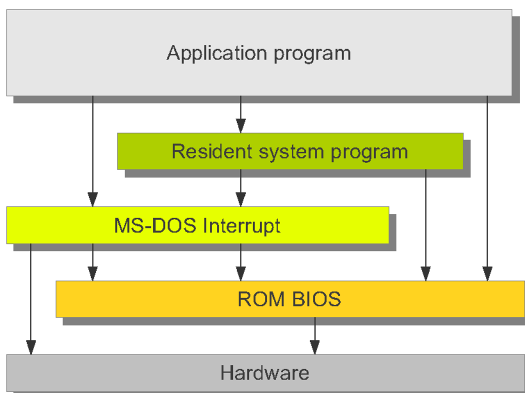
- Modern operating systems are so complex, they must be engineered carefully.
- A common approach is to partition a particular task into small components rather than one monolithic system.
  - Each component must be well-defined
    - Input, output, function of the component
- We already know those components, e.g., process management, file management, etc., but how are they interconnected and melted into a kernel ?



Simple System Structure

- Many operating systems started as a small, simple, and limited system.
  - Then, they grew beyond their original scope.
- e.g., MS-DOS, it was originally designed and implemented by a few people who had no idea that it would become so popular.
  - Limited hardware, so it was designed to provide most functionality in the least space.
  - Not divided into modules carefully.

MS-DOS System Structure



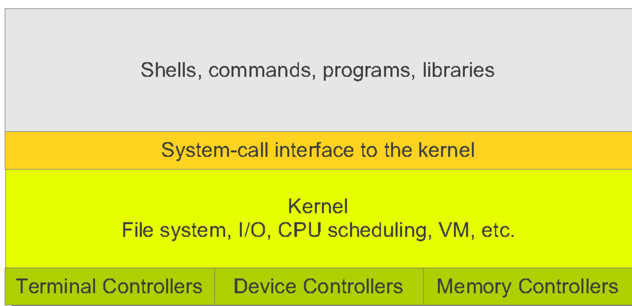
Layered Approach

- Break the operating system into a number of layers or levels, each built on top lower layers.
  - The lowest layer, layer 0, is the hardware.
  - The highest layer, layer n, is the user interface.
- Encapsulation of data and operations
  - Each layer hides the existence of certain data structures, operations, and hardware from higher-layers.
- The major difficulty of this approach is the definition of the layers.
  - A layer can use only layers below it, e.g., disk management must be at a lower level than the memory management because of the VM.

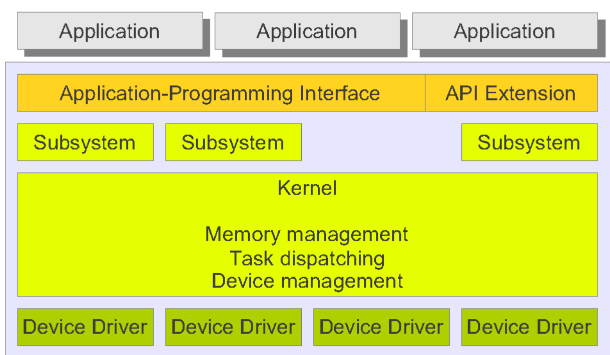
(cont'd.)

- Layered implementations tend to be less efficient than the other types.
  - Passing control layer to layer
  - Processing overhead in each layer
- Fewer layers with more functionality are being designed.
  - Provide modularized code while avoid the problems of layer definition and interaction.

UNIX System Structure



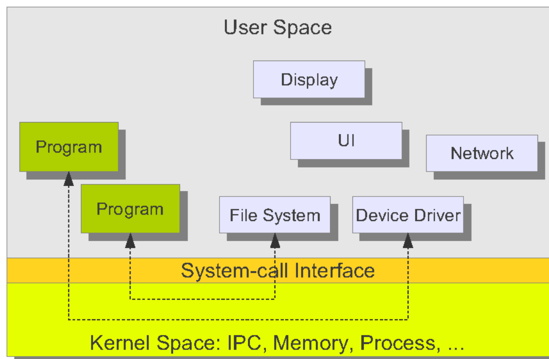
OS/2 System Structure



Microkernels

- As the UNIX expanded, the *monolithic* kernel became large and difficult to maintain.
- In mid 80s, researchers at CMU developed an operating system called *"Mach"* that modularized the kernel using the *microkernel* approach.
  - Remove all unnecessary components from the kernel, and implement them as system-level and user-level programs.
- The main function of microkernel is to provide a message passing facility between the client program and the various services that are running in user space.

(cont'd.)



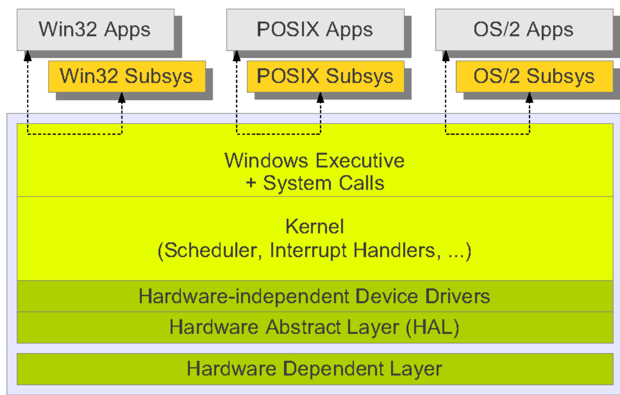
(cont'd.)

- The client program and the service never interact directly. They exchange messages with the microkernel.
- So, the operating system can be extended easily.
  - New services can be added without modification of the kernel.
  - The kernel is small, if it is modified, the changes tend to be fewer.
    - Easier to port from one hardware design to another.
- It is more secure and reliable because most services are running in user space rather than kernel space.

(cont'd.)

- Many contemporary operating systems have used the microkernel approach.
  - Tru64 UNIX
    - DEC OSF/1, then Digital UNIX, then Compaq, then HP
  - Mac OS X
    - Open source core codename “Darwin”
    - Developed from those in NEXTSTEP and FreeBSD.
  - QNX, Minix, L4, ...
- Performance is an issue in microkernel.
  - IPC bottleneck
  - Two-mode switches

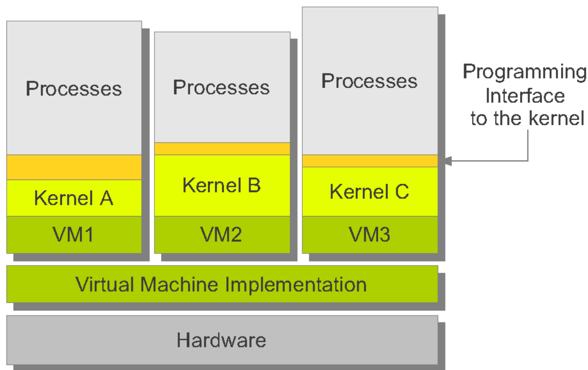
Microsoft Windows NT



Virtual Machines

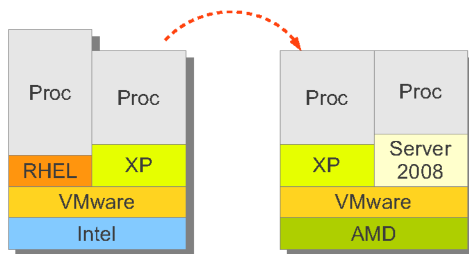
- By using CPU scheduling and virtual memory technique, and operating system can create an illusion that a process has its own processor with its own memory.
- The virtual-machine approach does not provide any additional functionality, but rather provides an interface that is identical to the underlying bare hardware.
- Advantages
  - Security
  - System development

(cont'd.)



(cont'd.)

- On 2008-11-06, Red Hat and AMD have just demonstrated live migrations of *running* VMs across *physical* machines of *different* vendors.



System Design Goals

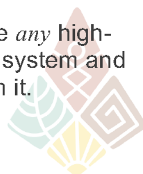
- At the highest level – choices of hardware and types of system: batch, time shared, single user, multiuser, distributed, real time, ...
- Beyond that ...
  - User goals: convenient and easy to use, learn, reliable, safe, and fast.
  - System goals: easy to design, create, maintain, flexible, reliable, error-free, efficient
- **No** general solution. That's why there are various kinds of operating systems to provide variety of solutions for different environments.

Implementations

- Traditionally, operating systems have been written in assembly language.
- Later, they are often written in higher-level languages.
  - Easier to port from one system to another.
  - First system not written in assembly language was probably the Master Control Program (MCP), which is written in a variant of ALGOL.
  - MULTICS was written in PL/1
  - UNIX, OS/2, MS Windows are mainly written in C.

(cont'd.)

- It is true that assembly code is generally smaller and more efficient, but modern compiler could optimize and generate excellent code.
- Major performance improvements are more likely to be the result of *better data structures and algorithms* than of excellent assembly-language code.
  - Of course, this does **not** mean we can use *any* high-level language to implement an operating system and always achieves a good performance from it.

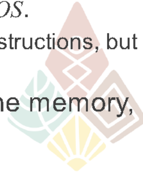


Loading an OS

- In any case, an operating system has to be loaded into the memory, then start to run to provide services as soon as the system is on.
  - This is more complex than it seems.
- To load an operating system, one must locate for a kernel image file.
  - File ?!! Hardware components do not know anything about file !
    - In fact, they don't even have any knowledge about file systems. Hardware alone cannot process files !
  - Sure, we can write a program load the image, but again, how can we load and run that program ?

Loading an OS ~ Linux on x86

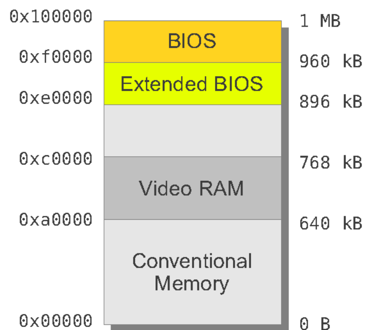
- So, the program must be immediately available when the system is on. In other words:
  - It must be retained even the power is off.
  - It cannot be in a secondary storage (e.g., a disk) because loading anything from those storages is too complex at a very beginning.
  - So, we put the program in a *non-volatile memory*. For PCs, this is called a *Basic I/O System* or *BIOS*.
    - BIOS does not only contain the start up instructions, but also provides low-level service routines.



- Eh ?? We still have to load BIOS into the memory, don't we ?
  - Well, for x86 architecture, we don't.

(cont'd.)

- The BIOS is *mapped* into the memory. So, the CPU can access BIOS directly.



(cont'd.)

- In 8086 era, upon reset or power on, the code segment (CS) register and instruction pointer (IP) register are set to `f000:fff0`.
  - That can be translated to the 20-bit physical address of (CS * 16) + IP = `ffff0`
    - The last 16-byte block of the BIOS.
  - On Linux, we may dump the BIOS by

```
# dd if=/dev/mem | hd -v | less
```

    and search for the offset `000ffff0`.

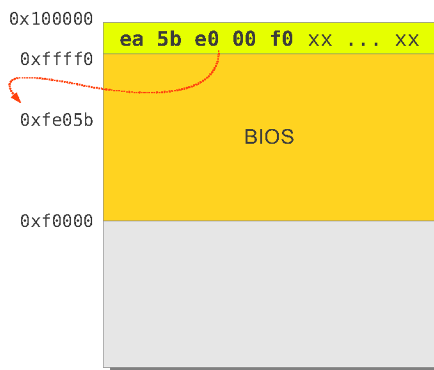
(cont'd.)

- It contains `ea 5b e0 00 f0`

`ea` `5b e0 00 f0`

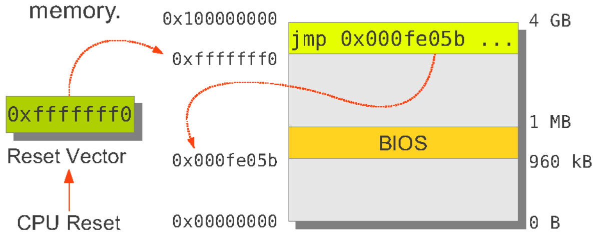
  - ea = far jump (4-byte operand = absolute address)
  - `5be000f0` specifies 20-bit physical address of `0xfe05b` or CS:IP of `f000:e05b`.
  - Again, we can try dd and search for the offset `000fe050`.

(cont'd.)



(cont'd.)

- In i386 and later, with 32-bit address space, the *reset vector* is `0xfffffff0`. The system would execute whatever the instruction is at the address.
  - It's the latest 16-bit block of the 32-bit address space, but this works even in a system with only 1 MB of memory.

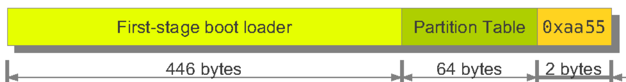


Hardware Boot-Up Sequence

- First, the BIOS tries to get CPU to run.
  - This also includes the chipset.
  - In case of multiprocessor or multi-core system, one of them will be used, the others will be halted.
- CPU starts executing BIOS instructions. The first is to perform *Power-On Self Test (POST).*
  - Check video adapter, get it shows some boot up output.
  - Check memory, and all required components.
  - Report error by (audio) code.

(cont'd.)

- It checks configurations and parameters from NVRAM (e.g, CMOS) to configure the system.
  - One of the configuration is the boot devices.
- It checks whether there exists the *master boot record* (MBR) in the configured boot device.
  - The first sector (sector 0) of the device with the MBR signature of `0xaa55`. This is also called a *boot sector*.



(cont'd.)

- On Linux, try:

  ```
  # dd if=<disk-device> bs=512 count=1 | hd
  ```

  It should be the same as, e.g., GRUB's stage1. This binary is called the *first-stage boot loader*.
- BIOS just puts the boot sector into memory at `0x00007c00` then executes whatever the code is.
- The first-stage boot loader now takes control of the system. This ends the hardware boot-up sequence.

The Bootstrap Loader

- The first-stage boot loader is too small to put the whole OS loader into it. So, most OSes use the first-stage boot loader to load another program called the *second-stage boot loader*.
- The second-stage boot loader can reside in a specific disk partition. It is capable to load a kernel image into memory and start running it.
  - e.g., `lilo`, `grub`, or `c:\NTLDR`.
  - Since it accesses a disk partition, the boot loader must know about the file system.
    - This is why it cannot fit the MBR.
    - GRUB may does this in stage 1.5 if not stage 2

(cont'd.)

- The second-stage boot loader then reads its configurations (e.g., `grub.conf`, `boot.ini`) and runs accordingly.
  - The boot loader may allow users to choose what OS to load from an interactive menu.
- The boot loader finally loads a kernel image.
  - For Linux, this can be one of the `vmlinuz`.
  - For recent versions of MS Windows, the image is `c:\windows\system32\ntoskrnl.exe`.

Loading a Kernel Image

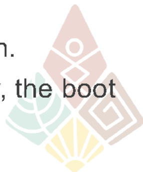
- During the second-stage, the system must run in the *real mode* of the x86 processor.
  - **Real mode** ~ 20-bit address space, software can access to BIOS functions directly (through a set of BIOS interrupts).
- It needs to use disk service routines in BIOS to read a kernel image from a disk partition.
- The problem is that the 20-bit address space is only 1 MB (only 640 kB is available), kernel images are typically larger than that.

(cont'd.)

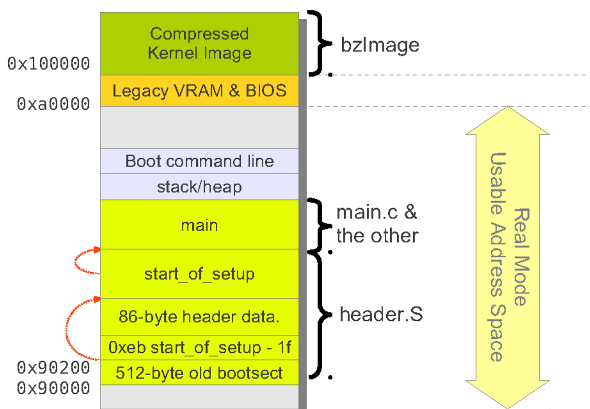
- Switch to the *protected mode* would help.
  - **Protected mode** ~ 24/32-bit address space with various features to support modern OSes.
  - But, all service routines in BIOS (including disk access) cannot be used in the protected mode.
- We need both BIOS functions in the real mode and a large address space of the protected mode at the same time.
  - How ?

(cont'd.)

- Some boot loader, e.g. GRUB, load a kernel image in the *unreal* mode.
  - This mode does not exist in any x86, so it is unreal. :)
  - It is actually a technique to access memory beyond 1 MB in the real mode.
    - A popular technique used by many DOS games during 1990 - 1995.
    - DOS Extender/DOS4GW
- The others may use `LOADALL` instruction.
- After the kernel image is in the memory, the boot loader transfers control to the kernel.



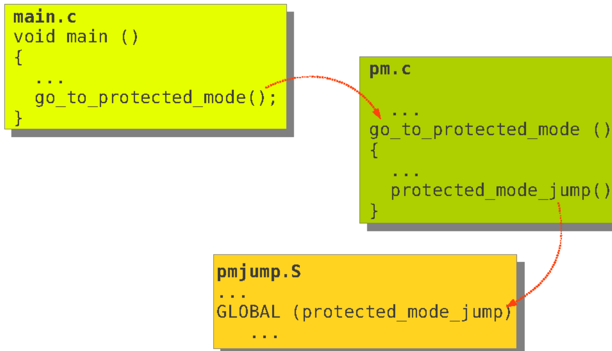
(cont'd.)



Running the Kernel

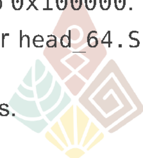
- Linux kernel has two parts:
  - Small real-mode part, starting at `0x90200`
  - Large protected-mode part, starting at `0x100000`
- The main code in real-mode part does many hardware checks, and prepares things before leave the real mode.
  - See `arch/x86/boot/main.c`.
  - The last instruction is `go_to_protected_mode()`

(cont'd.)



(cont'd.)

- Now, the protected mode part takes control.
  - Either `head_32.S` or `head_64.S` in `arch/x86/boot/compressed/` relocates the `bzImage` to end of the allocated buffer.
  - Then, it decompresses the `bzImage` to `0x100000` by calling `decompress_kernel` in `misc.c`.
    - Show “`Decompressing Linux ...`” message.
  - With “`Booting the kernel.`”, it jumps to `0x100000`.
- The `startup_32()` of either `head_32.S` or `head_64.S` in `arch/x86/kernel/` takes control.
  - Setup the system to run the kernel process.
  - Finally, it calls `start_kernel()`.



(cont'd.)

- The `start_kernel()` in `init/main.c` initializes nearly all kernel components, e.g.,
  - Scheduler
  - Memory zones
  - Memory allocator
  - Traps and IRQs
  - Timers
  - Console
  - CPU features
  - etc.
- It calls `kernel_init()` that, in turn, calls `init_post()` in the last step.

(cont'd.)

- The `init_post()` runs `/sbin/init` as the first user-space process.
  - The other processes will be created thereafter.
- On completion, the kernel just waits for an *event* to occur.

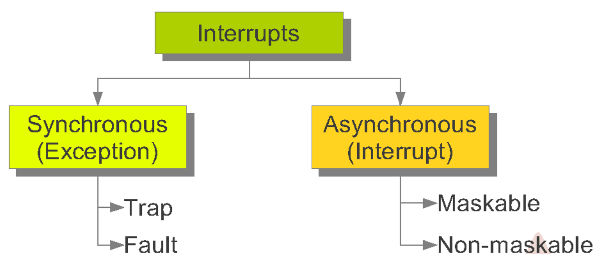
Events ??

- The occurrence of an event is signaled by an *interrupt*.
  - Both hardware and software are capable to generate an interrupt.
- Interrupts are generally divided into
  - **Synchronous** ~ generated by the control unit of a processor while executing an instruction. This occurs *only* after termination of executing of a instruction.
  - **Asynchronous** ~ generated by other hardware devices at any clock cycles.
- Intel manuals call these *exceptions* and *interrupts*, respectively.

(cont'd.)

- Some may also refine exceptions to:
  - **Faults** ~ generated in abnormal cases of execution that are recoverable, e.g., page fault.
  - **Traps** ~ generated when execution cannot be progressed any further, e.g., divided by zero.
- Also, interrupts may be classified into:
  - **Maskable** ~ can be ignored as long as it is masked, e.g., IRQs issued by I/O devices.
  - **Non-maskable** ~ critical events that cannot be ignored, e.g., hardware failures.

(cont'd.)

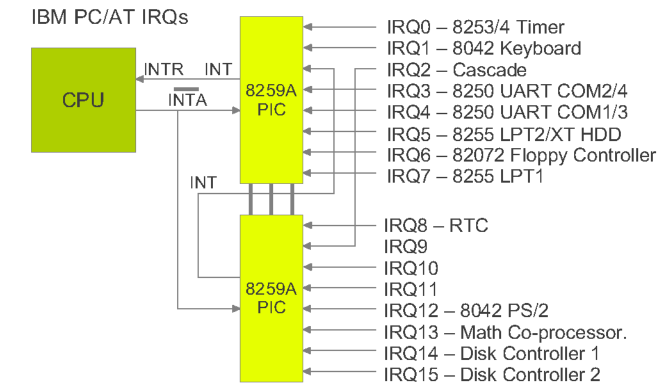


- How can operating system handle these ?
  - First, we should learn about interrupts in x86.

Hardware Interrupts

- Hardware devices are capable to issue interrupt requests through an output line designated as *Interrupt Request* (*IRQ*) line.
  - Some hardware, e.g., PCI, may have several IRQ lines.
- All IRQ lines are connected to the input pins of the *Programmable Interrupt Controller*.
  - e.g., Intel 8259A.
  - 8259A has 8 IRQ input lines, used in PC/XT.
  - Two 8259A can be cascaded to support up to 15 IRQ lines, used in PC/AT.

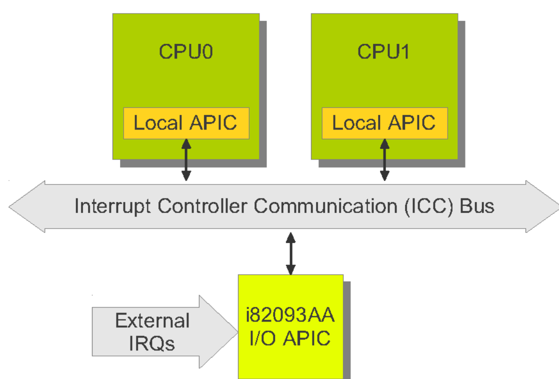
(cont'd.)



(cont'd.)

- PIC monitors IRQ input lines. If two or more lines are raised, the lowest pin is chosen.
- If it is an IRQ signal, PIC converts the signal to a 8-bit unsigned integer called a *vector*.
- The vector is put in PIC I/O port. PIC signals the processor through `INTR` pin.
- It waits until the processor acknowledges, then clear the `INTR` line.
- For Pentium III and later, there is a new chip called *I/O Advanced Programmable Interrupt Controller* (*I/O APIC*) to support SMP.

(cont'd.)



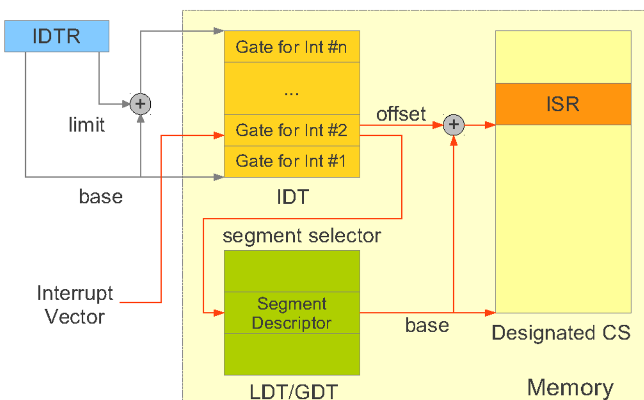
Interrupt Handlers

- In x86 architecture, the vector is 8-bit unsigned integer, so there are 256 interrupt vectors.
  - The vectors of NMIs and all exceptions are fixed.
  - Intel reserved vector 0 – 31
    - 0 – 19 has been defined.
    - 20 – 31 are reserved.
  - The maskable ones can be changed by program those PIC or APIC.
  - See: Intel Architecture Software Developer's Manual Volume 3: System Programming.

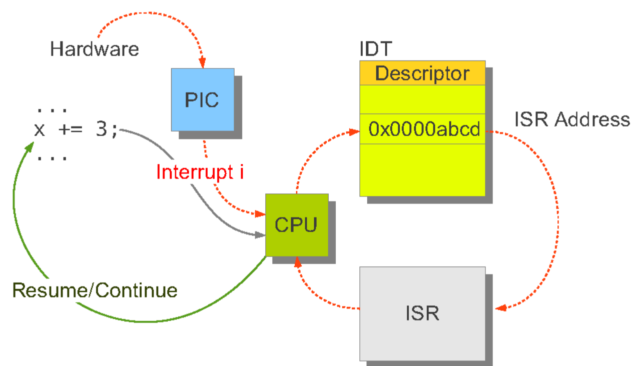
(cont'd.)

- Interrupt must be handled quickly.
  - That's why only a predefined number of interrupts is possible.
  - Since the number of vector is fixed, data structure can be a simple array.
- This array is called the *Interrupt Descriptor Table* (*IDT*).
  - 256 entries x 8 bytes per entry
  - Each entry is either a task-gate, interrupt-gate, or trap-gate descriptor.
  - Those interrupt gate and trap gate descriptors contain offset to the service routine.

(cont'd.)



(cont'd.)



(cont'd.)

- When the CPU is interrupted, it *stops* what it is doing, follows the interrupt vector, and immediately transfer execution to the interrupt service routine.
- The interrupt service routine executes. On completion, the CPU *resumes* the interrupted computation.

Implementing System Calls

- When a user-mode process invokes a system call, the processor has to switch to kernel mode.
- On Linux, user programs do not directly access system calls, but rather through *wrapper routines* in a library, e.g., `libc`.
- The wrapper routine, in turn, calls the *system call handler*. The execution is transfered to the kernel mode by using the interrupt call.
  - Older ~ interrupt vector 128 ~ `int $0x80`
  - Linux 2.6 (with Pentium® II or later) ~ `sysenter`

(cont'd.)

- The system call handler calls the actual *system call service routine*.
  - See `arch/x86/kernel/syscall_table_32.S`
  - See `arch/x86/kernel/entry_32.S`
- On completion, `iret` or `sysexit` is issued to exit the kernel mode.

(cont'd.)