CPSC 457

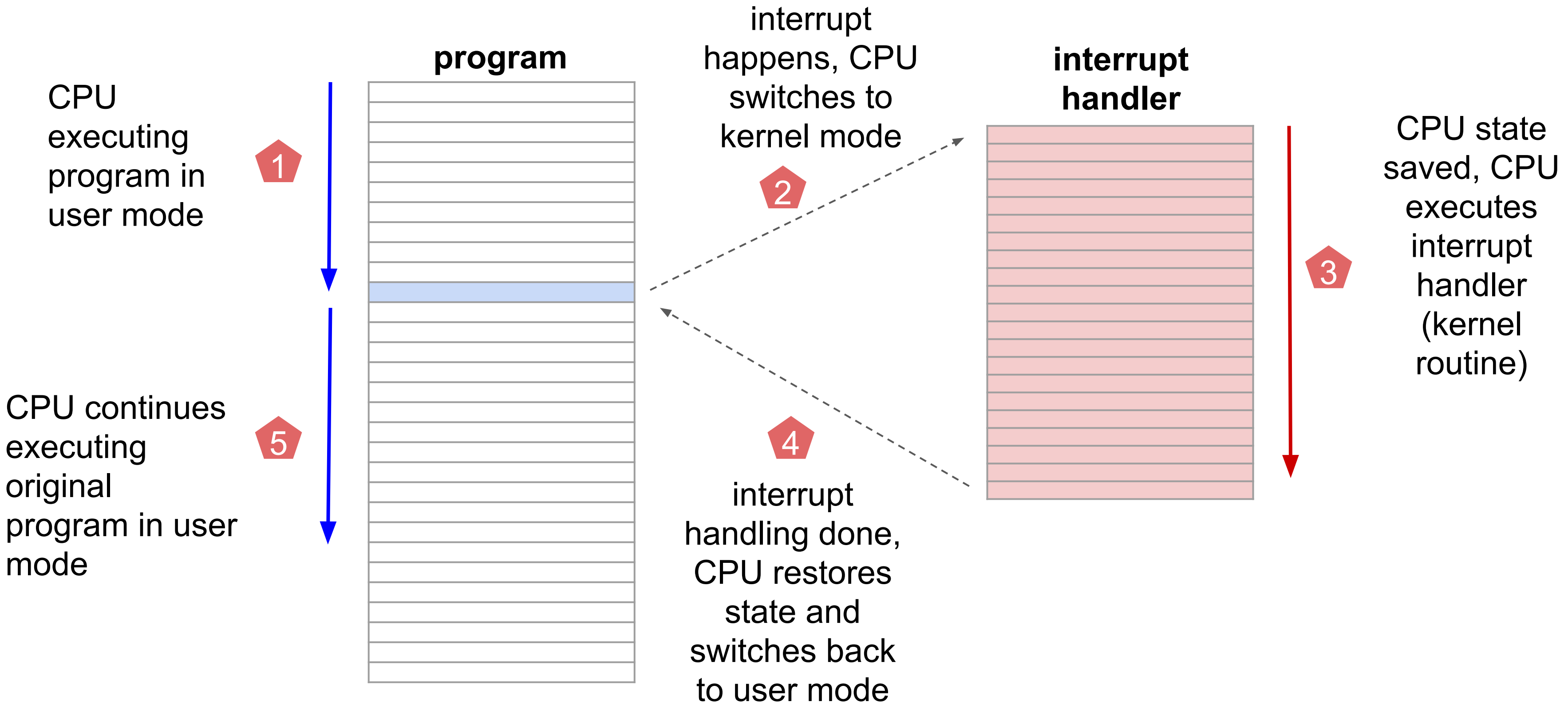
Interrupts, traps, kernel design, VMs

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- traps, interrupts
- kernel designs
- virtual machines

Interrupts



program
interrupt handler
CPU executing program in user mode
1
interrupt happens, CPU switches to kernel mode
2
3
CPU state saved, CPU executes interrupt handler (kernel routine)
4
interrupt handling done, CPU restores state and switches back to user mode
5
CPU continues executing original program in user mode

Software interrupts (exceptions / traps)

- similar to hardware interrupts, but the source of the interrupt is the CPU itself
- software interrupts are handled similarly to hardware interrupts
- two types: unintentional and intentional
- unintentional software interrupts, aka. **exceptions**:
  - occurs when CPU executes 'invalid' instruction
  - eg. accessing non-existent memory, write to read/only memory, division by zero, …
  - used by OS to detect when an application attempts an illegal operation
- intentional software interrupt, aka. **trap**
  - trap occurs (usually) via special instruction, eg. INT
  - the purpose is to execute predefined routine in kernel mode
  - operating systems can use traps to implement system calls

Hardware Interrupts vs Software Interrupt (Traps, Exceptions)

Hardware Interrupts:

- external event delivered to the CPU
- origins: I/O, timer, user input
- asynchronous with the current activity of the CPU
- the time of the event is not known and is not predictable

Software Interrupts:

- internal events, eg. system calls, error conditions (div by zero)
- synchronous with the current activity of the CPU
- occurs as a result of execution of a machine instruction

but both ...

- invoke a kernel routine, defined by the OS
- put the CPU in a kernel mode
- save the current state of the CPU
- eventually resume the original operations when done

I/O

- how does the kernel do I/O?
- option 1: busy waiting / spinning / busy looping

```
cpu → disk : please read a file
loop:
    cpu → disk : are you done yet?
    if yes then break else continue loop
cpu → disk : give me the result
```

- problems
  - the CPU is tied up while the slow I/O completes the operation
  - we are wasting power (so what?)

I/O

- how does the kernel do I/O?
- option 2: busy wait with sleep

```
cpu → disk : please read a file
loop:
    sleep for a while
    cpu → disk : are you done yet?
    if yes then break else continue loop
cpu → disk : give me the result
```

- sleep could be detected by OS, and the CPU could then run another program
- problems:
  - hard to estimate the right amount of sleep
  - program might end up running longer than necessary

I/O

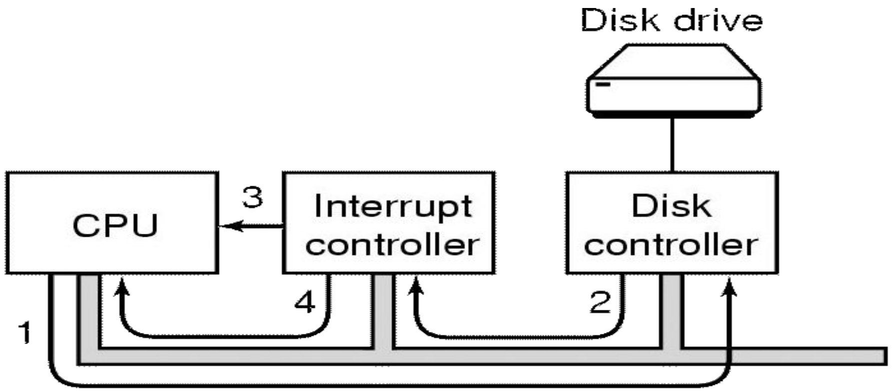
- how does the kernel do I/O?
- option 3: hardware interrupts

```
cpu → disk : please read a file
cpu → disk : when you're done, let me know
cpu: goes to sleep / or executes other program … until an interrupt happens
disk → cpu : I am done (interrupt)
cpu → disk : give me the result
```

- when the I/O device finishes the operation, it generates an interrupt, letting the OS know it's done, or if there was an error
- this approach assumes the I/O device supports interrupts
- most devices support interrupts, and if they don't, they can be connected through controllers that do

Using Interrupts to do I/O

- Kernel talks to the device driver to request an operation.
- The device driver tells the controller what to do by writing into its device registers.
- The controller starts the device and monitors its progress.
- When the device is done its job, the device controller signals the interrupt controller.
- The interrupt controller informs the CPU and puts the device information on the bus.
- The CPU suspends whatever it's doing, and handles the interrupt by executing the appropriate interrupt handler (in kernel mode).
- The CPU then resumes its original operations.



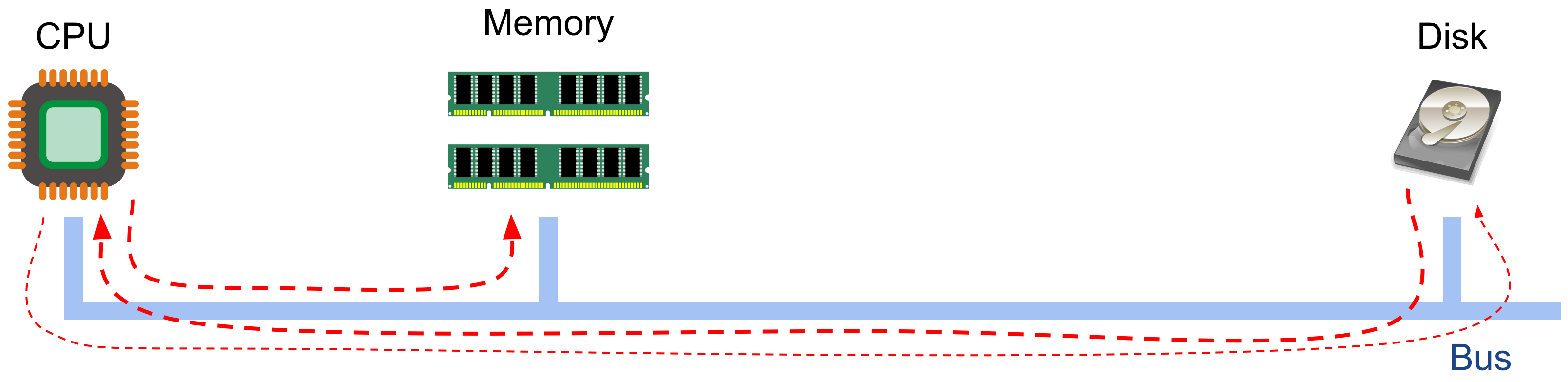
Limits of interrupts

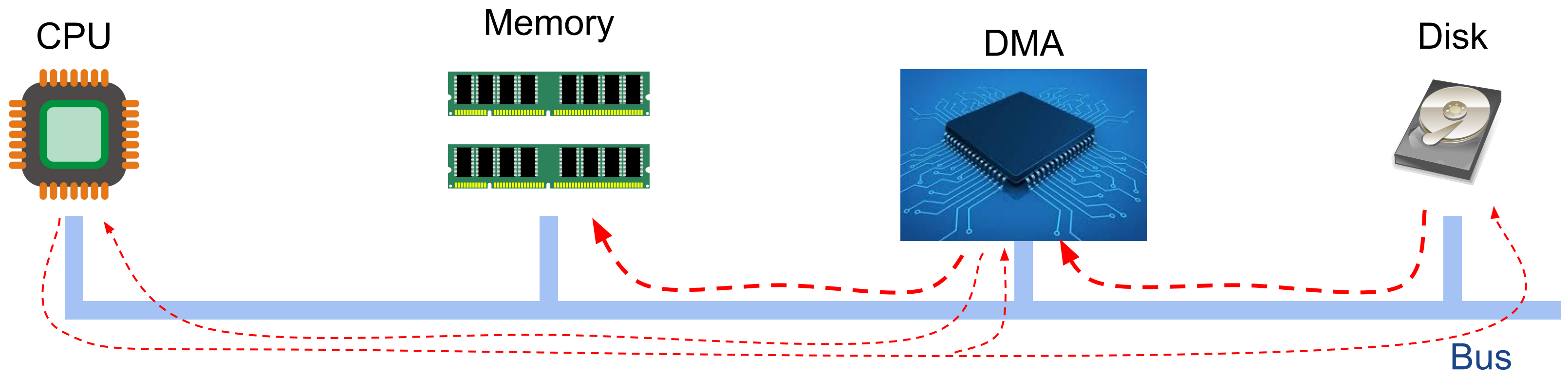
- CPU can run other programs while waiting for I/O
- but the CPU could be interrupted for every single byte of I/O
  - many devices/controllers have limited memory
  - these devices could generate an interrupt for every single byte
  - interrupts take many CPU cycles to save/restore CPU state
  - useful work often a single instruction - to store the data in memory
- better solution - introduce a dedicate hardware to deal with interrupts (DMA)
  - DMA absorbs most interrupts
  - DMA can save data directly into memory, without CPU even knowing
  - result is less interrupts for CPU

Direct memory access (DMA)

- special piece of hardware on most modern systems
- used for bulk data movement such as disk I/O
    - usually used with slow devices, so that CPU can do other useful things
    - but could be used with extremely fast devices that could overwhelm the CPU
- device controller transfers an entire block of data directly to the main memory without CPU intervention
- only one interrupt is generated per-block – to tell the device driver that the operation has completed
- used for device → memory, memory → device and even memory → memory transfers

DMA (without and with comparison)





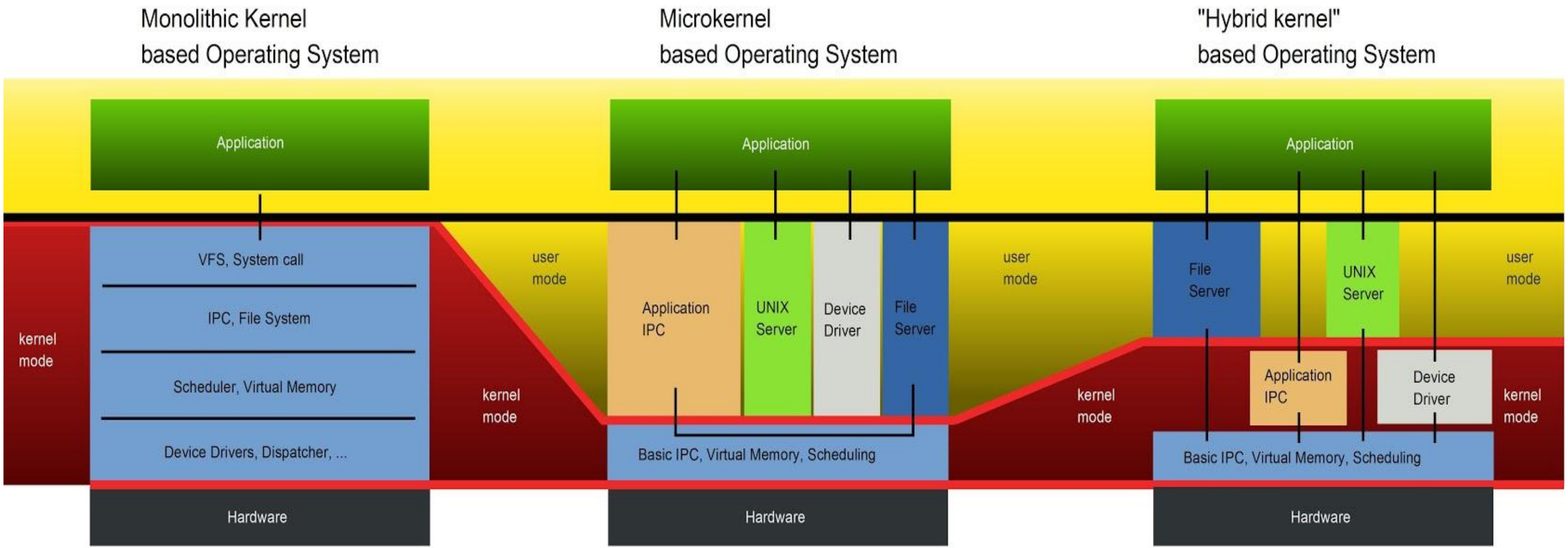
Kernel designs

- what goes into a kernel and what does not?
- trade-offs to consider:
  - code in kernel runs faster, but
  - big kernels have more bugs → higher system instability

Kernel designs

- **monolithic kernels** (e.g., MS-DOS, Linux)
    - the entire OS runs as a single program in kernel mode
    - faster, but more prone to bugs, harder to port, potentially less stable
- **microkernels** (e.g., Mach, QNX)
    - only essential components in kernel — running in kernel mode
        - essential = code that must run in kernel mode
    - the rest is implemented in user mode
    - less bugs, easier to port, easier to extend, more stable, but slower
- many modern OSes are **hybrid kernels**
    - trying to balance the cons/pros of monolithic kernels and microkernels

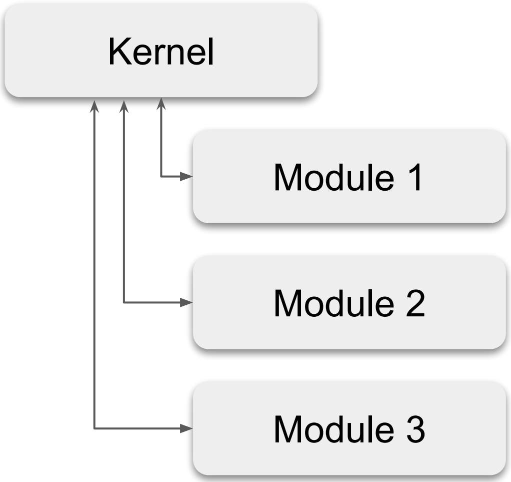
Monolithic / Microkernel / Hybrid



https://commons.wikimedia.org/w/index.php?curid=4397379

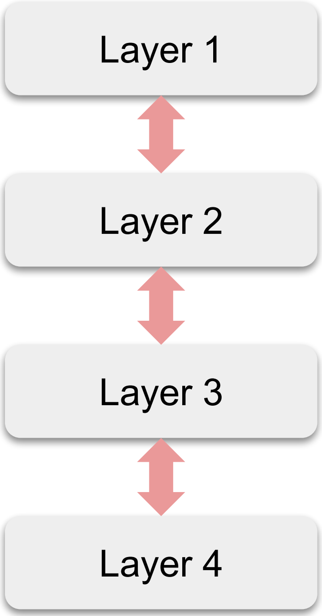
Kernel modules

- modular kernels (type of hybrid kernel)
- smaller kernel with only essential components, plus non-essential, dynamically loadable kernel parts (**kernel modules**)
- drivers are often implemented as modules (Linux)
- modules loaded on demand, when needed or requested
  - could be at boot time, eg. loading a driver for a video-card
  - or could be done later, eg. when user plugs in a USB device
  - modules usually run in kernel mode
- OS can come with many drivers, but only the needed ones are actually loaded, leads to faster boot time
- no kernel recompile/reboot necessary to activate a module

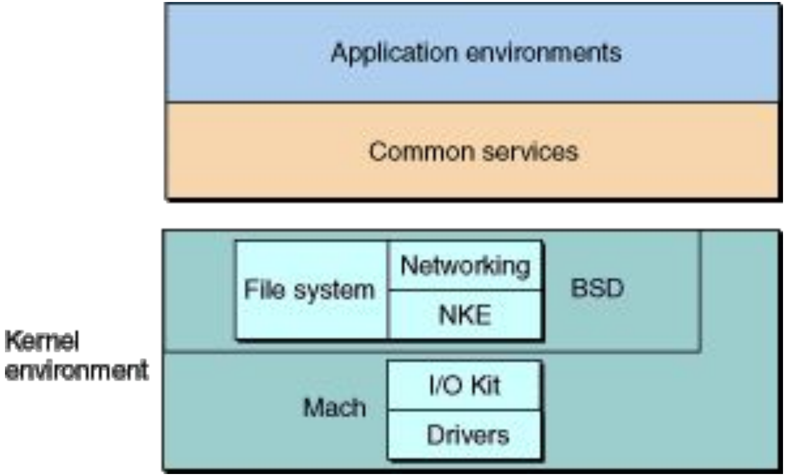


Layered approach

- kernel components organized into a hierarchy of layers
- layers above constructed upon the ones below it
- sounds great in theory, but…
  - hard to define layers, needs careful planning
  - less efficient since each layer adds overhead to communication
  - not all problems can be easily adapted to layers
  - some parts of Linux implemented via layers (eg. VFS)

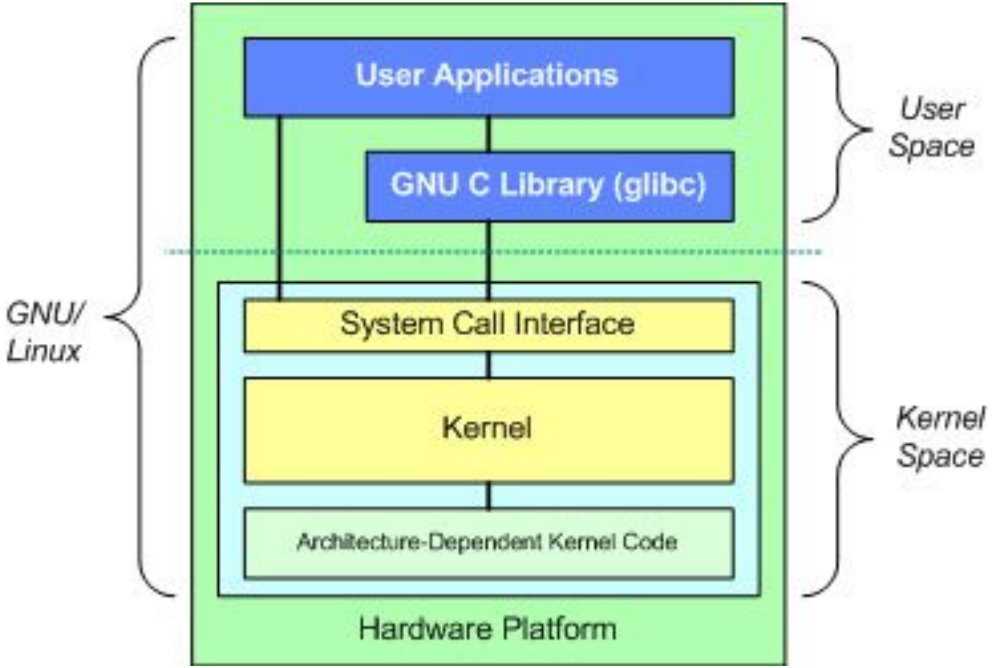


Mac OS X structure



- hybrid kernel "XNU"
- Mach microkernel: memory management, RPC, IPC, thread scheduling
- BSD kernel: BSD command line interface, networking, file systems, POSIX APIs

GNU/Linux structure

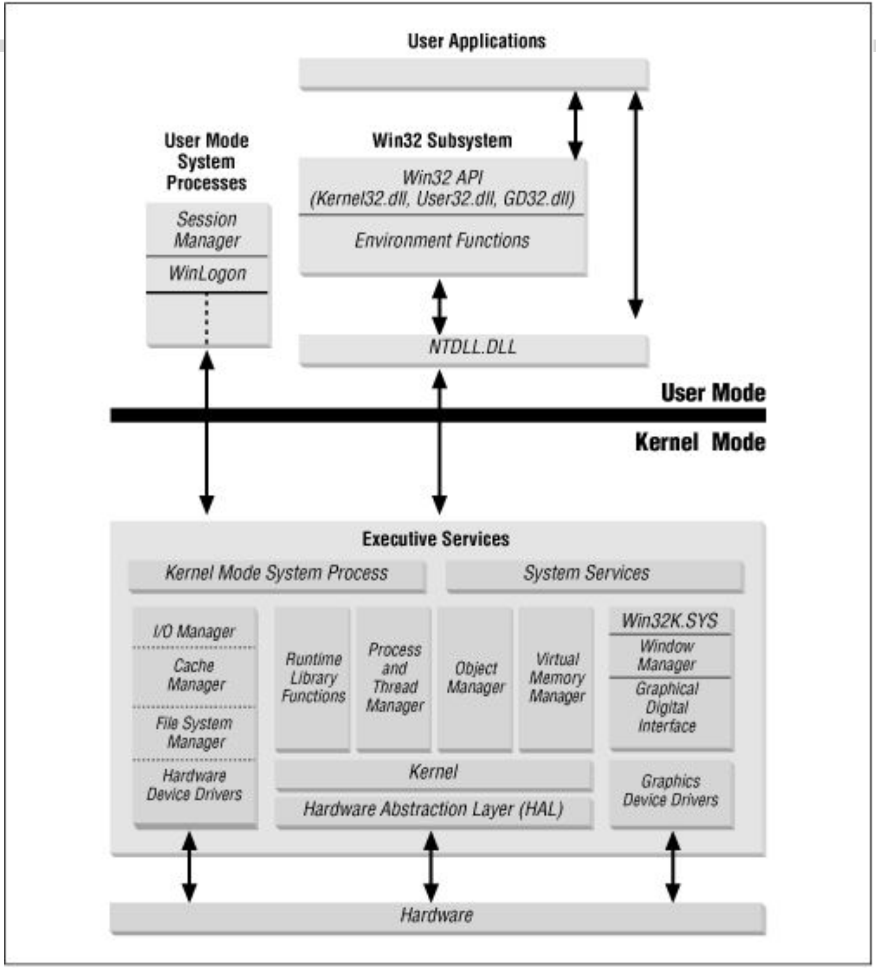


- monolithic kernel
- with some layers
- and dynamically loadable modules

www.ibm.com/developerworks/linux/library/l-linux-kernel/

Win NT structure

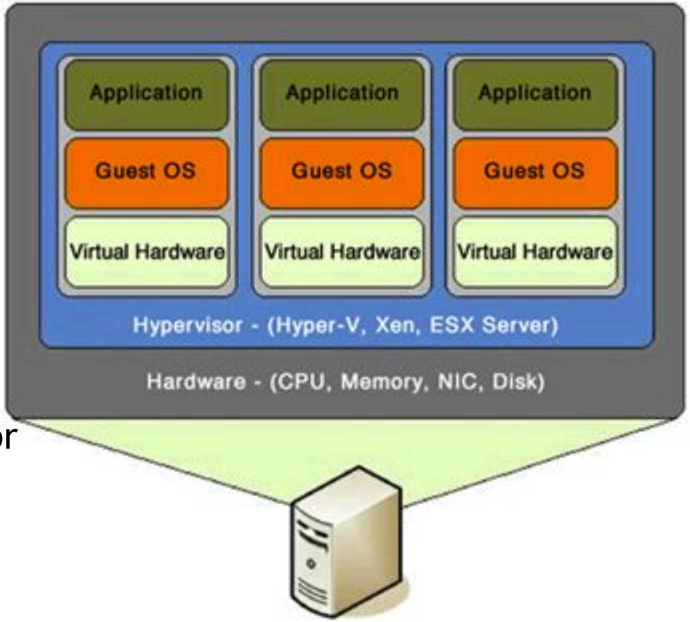
- hybrid kernel
- modules & layers



technet.microsoft.com/en-us/library/cc768129.aspx

Virtual machines

- virtual machines (VMs) emulate computer systems
  - in software, or in specialized hardware, or both
- host machine creates illusion that each guest machine has its own processor and its own hardware
- hypervisor - software or hardware that manages VMs
  - bare-metal - runs directly on hardware
    - usually on big servers, fastest
    - XEN, VMWare ESX
  - hosted - runs on top of another OS
    - usually on desktops, slower
    - VMWare Player, VirtualBox, Docker (kind of)
  - hybrid - eg. Linux kernel can function as a hypervisor through a KVM module
- also possible - OS virtualization, eg. docker, lxc



Benefits of VMs

- the host system is protected from the VMs
  - eg. run unsafe programs in VM
- VMs are isolated & safe from each other
  - eg. can run conflicting applications in separate VMs
- multiple different OSes or versions can be running on the same computer concurrently
- perfect vehicle for OS research and development
  - normal system operation seldom needs to be disrupted from system development
- system consolidation
  - can potentially save lot of money – buy one big server, instead of many smaller ones
  - system administrators love it

Review

- Why do modern OSs move away from the standard monolithic system structure?
- Benefits of virtual machines?

Summary

- Interrupts
  - Interrupts vs. traps
  - DMA
- OS structure
  - Monolithic systems, Microkernel
  - Modular kernels and layered approach
- Virtual machines

Reference: 1.3, 1.7 (Modern Operating Systems)

1.2, 1.4, 2.6-2.8 (Operating System Concepts)

Questions?