CPSC 457

Interrupts, traps, kernel design, VMs

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- interrupts, traps
- kernel designs
- virtual machines

I/O

- how do we write programs that do I/O?
  - how do we write programs that interact with devices?
  - how does kernel do I/O?
- at low level (eg. assembly, or C without OS)
  - we use special I/O instructions
  - or memory mapped devices
- at higher level we use system calls provided by the OS
  - most OSes have blocking and non-blocking versions of system calls
  - blocking calls will suspend the application until request is finished
  - non-blocking calls will schedule the request and application continues to run

I/O - busy waiting

- I/O using **busy waiting** / spinning / busy looping:
  - CPU repeatedly checks if device is ready

```
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 with busy waiting:
  - CPU is tied up while the slow I/O completes the operation
  - we are wasting power and generating heat (so what?)

I/O - busy waiting with delay

- I/O using busy wait and sleep:
  - same as before, we just add a short delay to reduce CPU usage

```
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 be given to another program
- problems:
  - hard to estimate the right amount of sleep
  - program might end up running longer than necessary

I/O - using interrupts

- I/O using interrupts:

```
cpu → disk : please read a file, and when you're done,
             interrupt me to let me know
```

program can continue to execute, or can sleep
when interrupt happens, some interrupt routine gets executed:

```
disk → cpu : hey, I am done (interrupt)
cpu → disk : give me the result
```

- when the I/O device finishes the operation, it generates an interrupt, letting the CPU 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

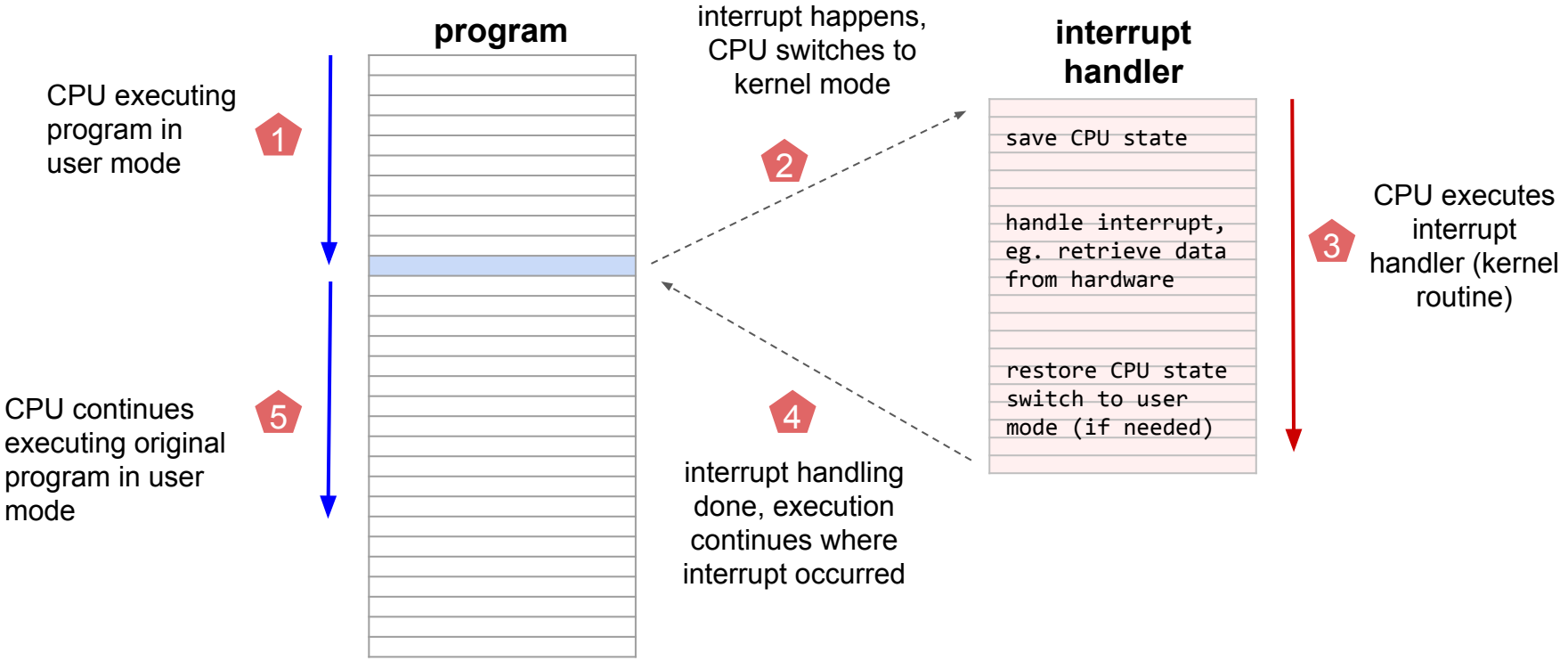
Interrupts

- a mechanism to let the CPU know something "important" happened

  eg. a printer finished printing, new data arrived on a network card, a disk finished saving data, a program performed illegal instruction, program requested access to some hardware

- the CPU usually responds to interrupt immediately
- the CPU temporarily suspends its current activity
  - eg. by saving all registers in memory
- the CPU then executes a predefined routine (interrupt handler or **i**nterrupt **s**ervice **r**outine)
  - the CPU switches to kernel mode (privileged mode)
  - which routine[s] gets executed is configured by OS
- eventually CPU restores the saved state and resumes original execution

Interrupts


program
interrupt happens, CPU switches to kernel mode
interrupt handler
CPU executing program in user mode
1
2
save CPU state
handle interrupt, eg. retrieve data from hardware
3
CPU executes interrupt handler (kernel routine)
CPU continues executing original program in user mode
5
4
interrupt handling done, execution continues where interrupt occurred
restore CPU state
switch to user mode (if needed)

Interrupts

- most CPUs support multiple different interrupts, numbered 0..N
- most CPUs support having different handlers for each interrupt
- a common mechanism is an interrupt vector table
  - for each interrupt it contains an address of a service routine
  - eg. x86 has 256 different interrupts, so its IVT has 256 entries (addresses)
- depending on the source of the interrupt, we have:
  - hardware interrupts
  - software interrupts
- they are handled the same way

Hardware interrupts

- the source of the interrupt is another device
  - eg. printer, hard-drive, mouse, network card
- the precise timing of a HW interrupt is unpredictable - it can happen any time
- an interrupt can happen even while servicing a previous interrupt
- most CPUs allow defining interrupt priorities
- most CPUs allow disabling interrupts for short periods

Software interrupts (exceptions / traps)

- similar to hardware interrupts, but the source of the interrupt is the CPU itself
  - otherwise they are handled similarly to hardware interrupts
- two types: unintentional and intentional
- unintentional software interrupts, aka. exceptions:
  - occurs when CPU executes "invalid" or "forbidden" instruction
  - eg. accessing non-existent memory, write to read-only memory, division by zero, …
  - used by OS to detect when an application misbehaves, OS usually terminates it
- intentional software interrupt, aka. traps (in this course *)
  - trap occurs as a result of executing a special instruction, eg. `INT`
  - the purpose is to execute a predefined routine in kernel mode
  - some operating systems 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:

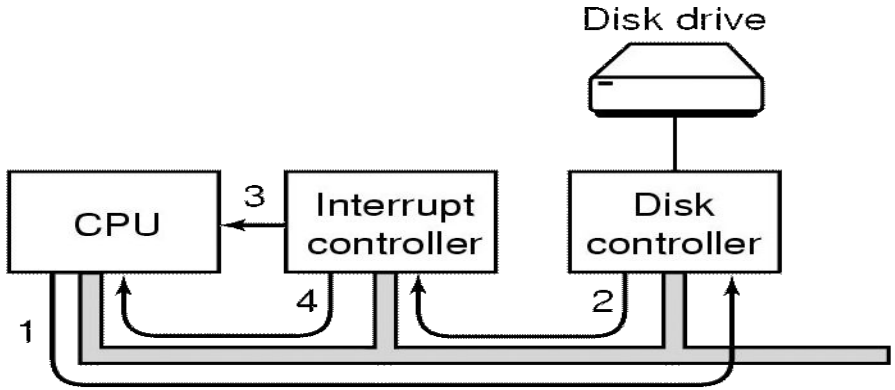
- internal events, eg. system calls, error conditions (div by zero)
- synchronous with the current activity of the CPU
- occurs as a result of execution of a machine instruction

but both ...

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

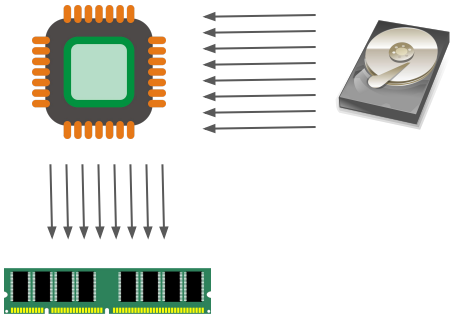
Using Interrupts to do I/O

- Kernel talks to a 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.

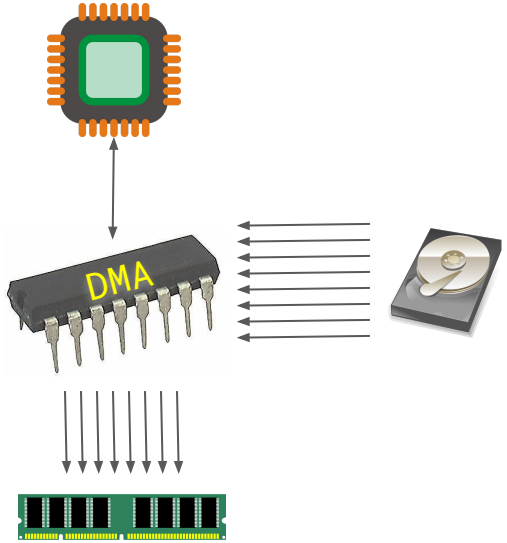


Limits of interrupts

- 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
  - such 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 – a dedicated 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

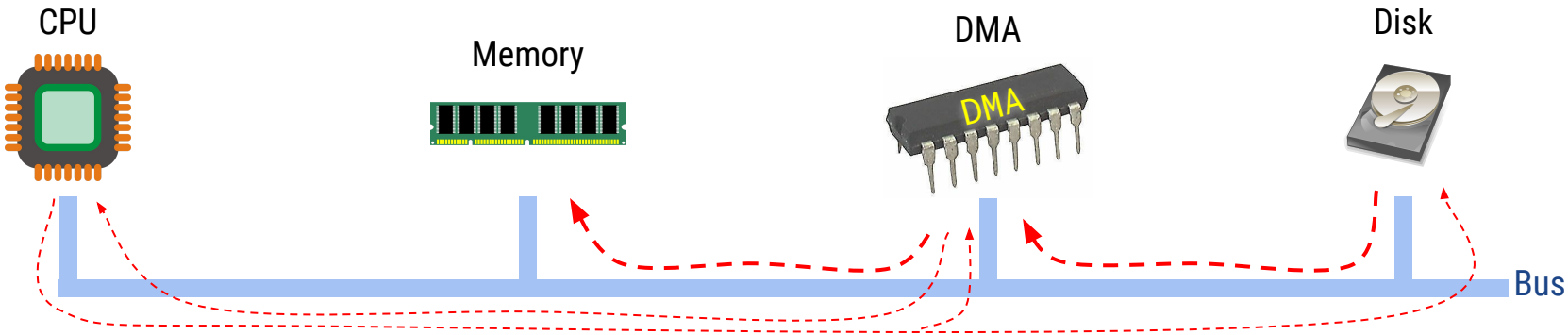


Direct memory access (DMA)



- 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 can be also used with very fast devices
    that could overwhelm the CPU
- DMA 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)



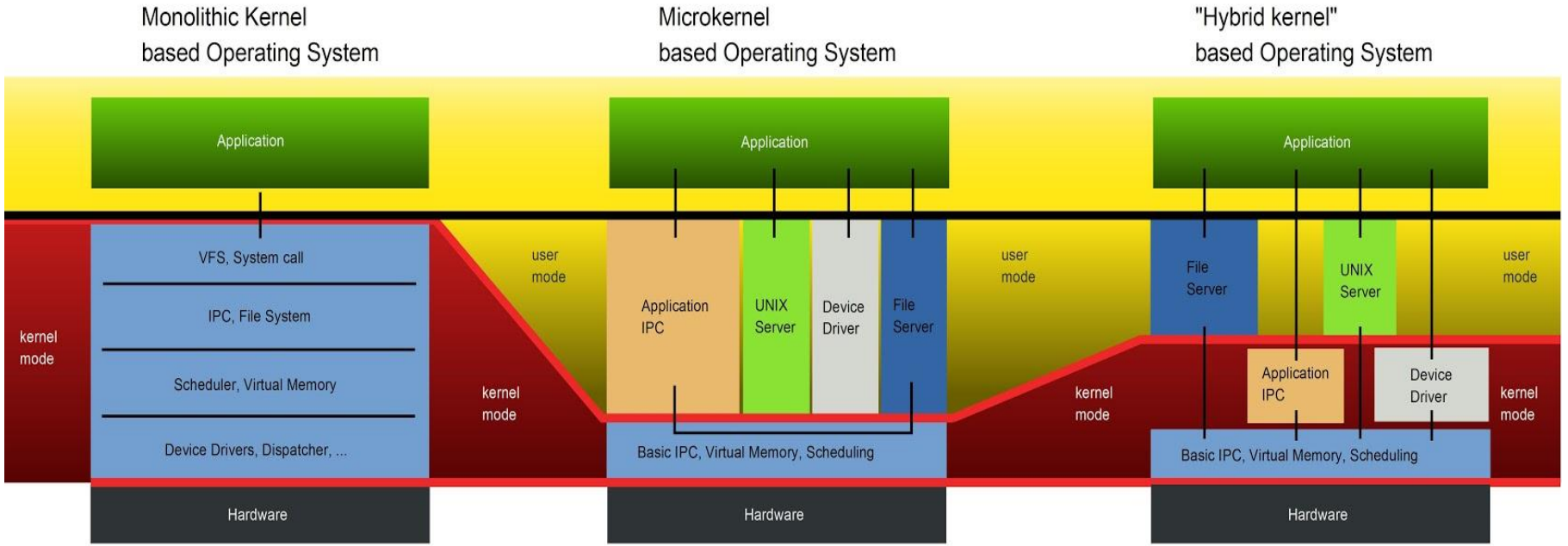
Kernel designs

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

Kernel designs

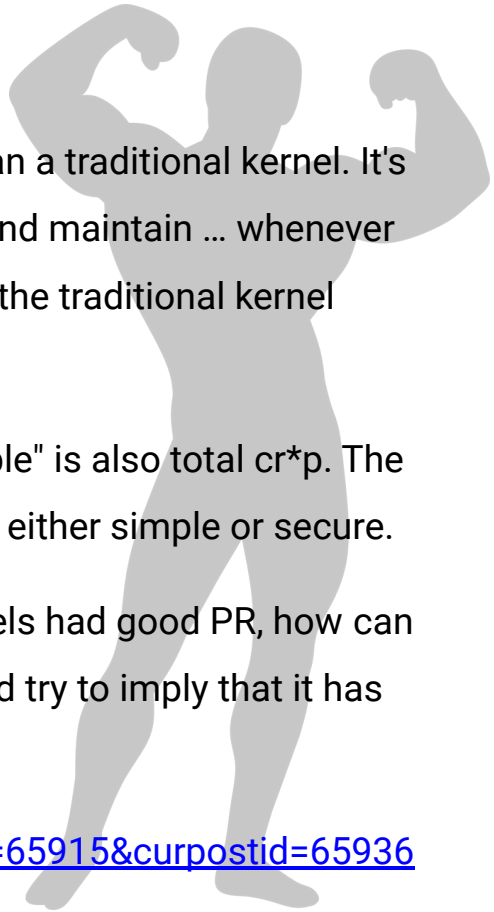
- monolithic kernels (e.g., MS-DOS, Linux)
  - the entire OS runs as a single program in kernel mode
  - fastest, more prone to bugs, potentially less stable, harder to port
- microkernels (e.g., Mach, QNX)
  - only essential components in kernel — running in kernel mode
    - essential = code that must run in kernel mode
  - the rest is implemented in user mode
  - less bugs, easier to port, more stable, but slower
- some modern OSes claim to be 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 designs – theory vs practice



Linus (2006):

It's ludicrous how micro-kernel proponents claim that their system is "simpler" than a traditional kernel. It's not. It's much much more complicated … Microkernels are much harder to write and maintain … whenever you compare the speed of development of a microkernel and a traditional kernel, the traditional kernel wins. By a huge amount, too...

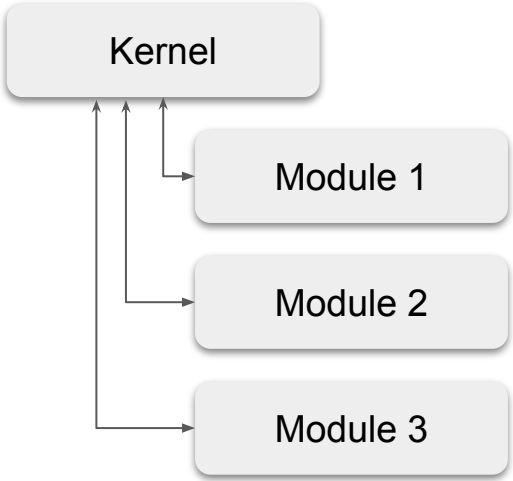
The whole argument that microkernels are somehow "more secure" or "more stable" is also total cr*p. The fact that each individual piece is simple and secure does not make the aggregate either simple or secure.

As to the whole "hybrid kernel" thing - it's just marketing. It's "oh, those microkernels had good PR, how can we try to get good PR for our working kernel? Oh, I know, let's use a cool name and try to imply that it has all the PR advantages that that other system has"

https://www.realworldtech.com/forum/?threadid=65915&curpostid=65936

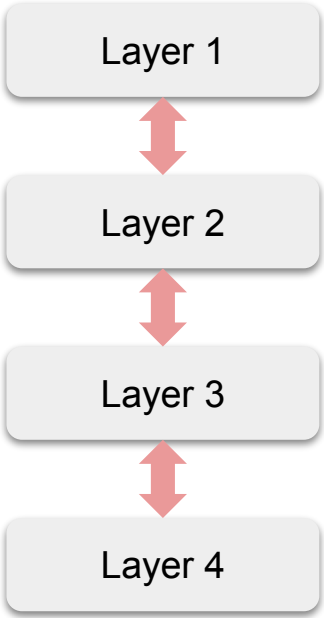
Kernel modules

- modular kernels (could be used to implement 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, but some may run in user mode
- OS can come with many drivers, but only the needed ones are actually loaded, resulting in faster boot time
- no kernel recompile/reboot necessary to activate a module



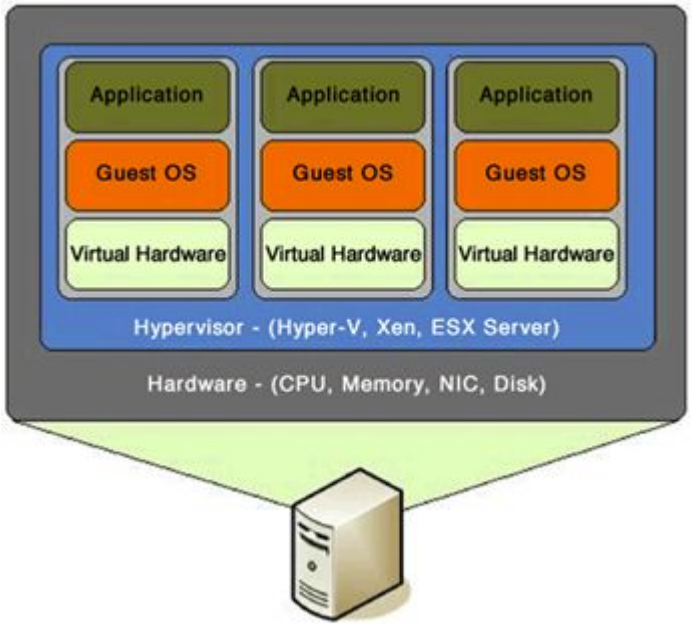
Layered approach

- 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)



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

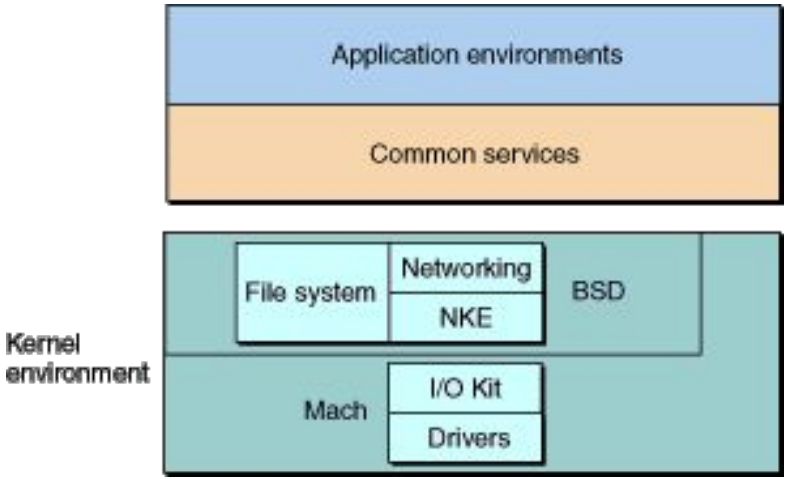
- the host system is protected from the VMs
  - eg. run unsafe programs in VM
- VMs are isolated & safe from each other
  - eg. can run conflicting applications in separate VMs
- multiple different OSes or versions can be running on the same computer concurrently
  - eg. host is Windows, VMs are Linux and MacOS
- 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

Questions?

Review

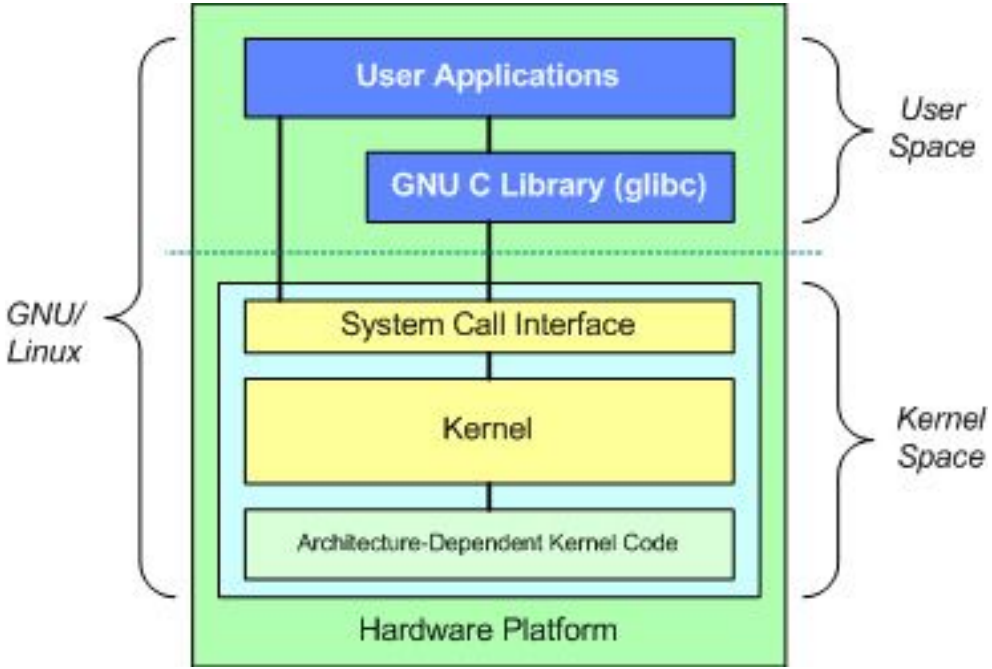
- Why do modern OSs move away from the standard monolithic system structure?
- List some benefits of virtual machines:
  - from a user's perspective
  - from a developer's perspective
  - from a company's perspective
  - from a system administrator's perspective

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



still considered

monolithic kernel,

but with some layers,

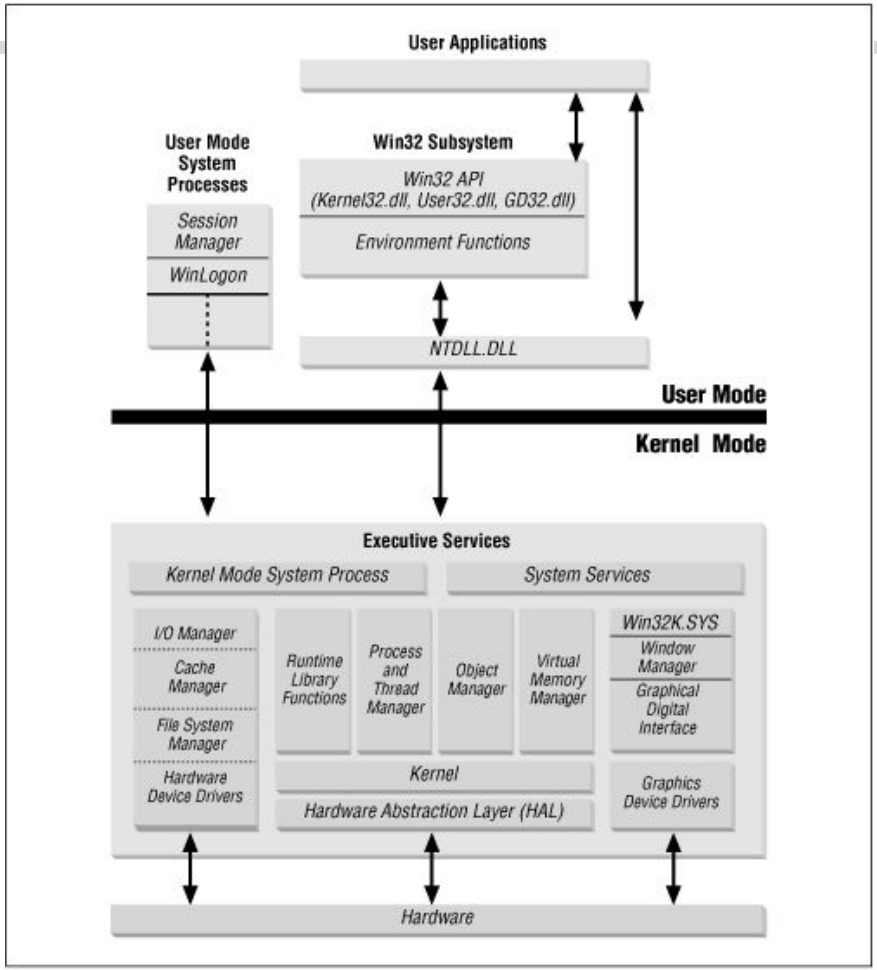
and dynamically loadable

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