Chapter I Introduction

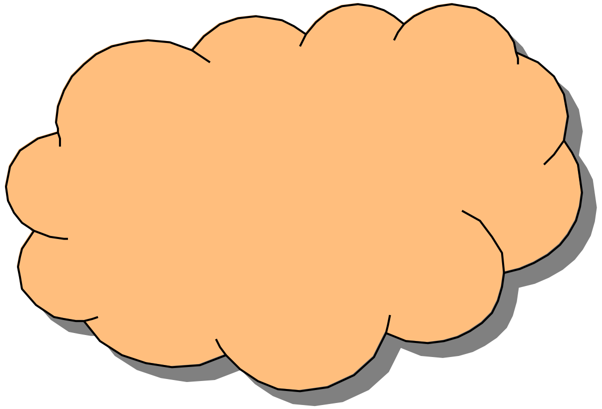
Jehan-François Pâris
jfparis@uh.edu

Chapter Overview

- Defining operating systems
- Major functions of an OS
- Types of operating systems
- Unix
- Kernel organization

What is an operating system?

- "What stands between the user and the bare machine"


A better definition

- The *basic* software required to operate a computer.
- Has a similar role to that of the conductor of an orchestra

Do not belong to OS

- All ***user programs***
- Compilers, spreadsheets, word processors, and so forth
- Most utility programs
  - ***mkdir*** is a user program calling ***mkdir()***
- The ***command language interpreter***
  - Anyone can write his/her Unix shell

The Unix shells

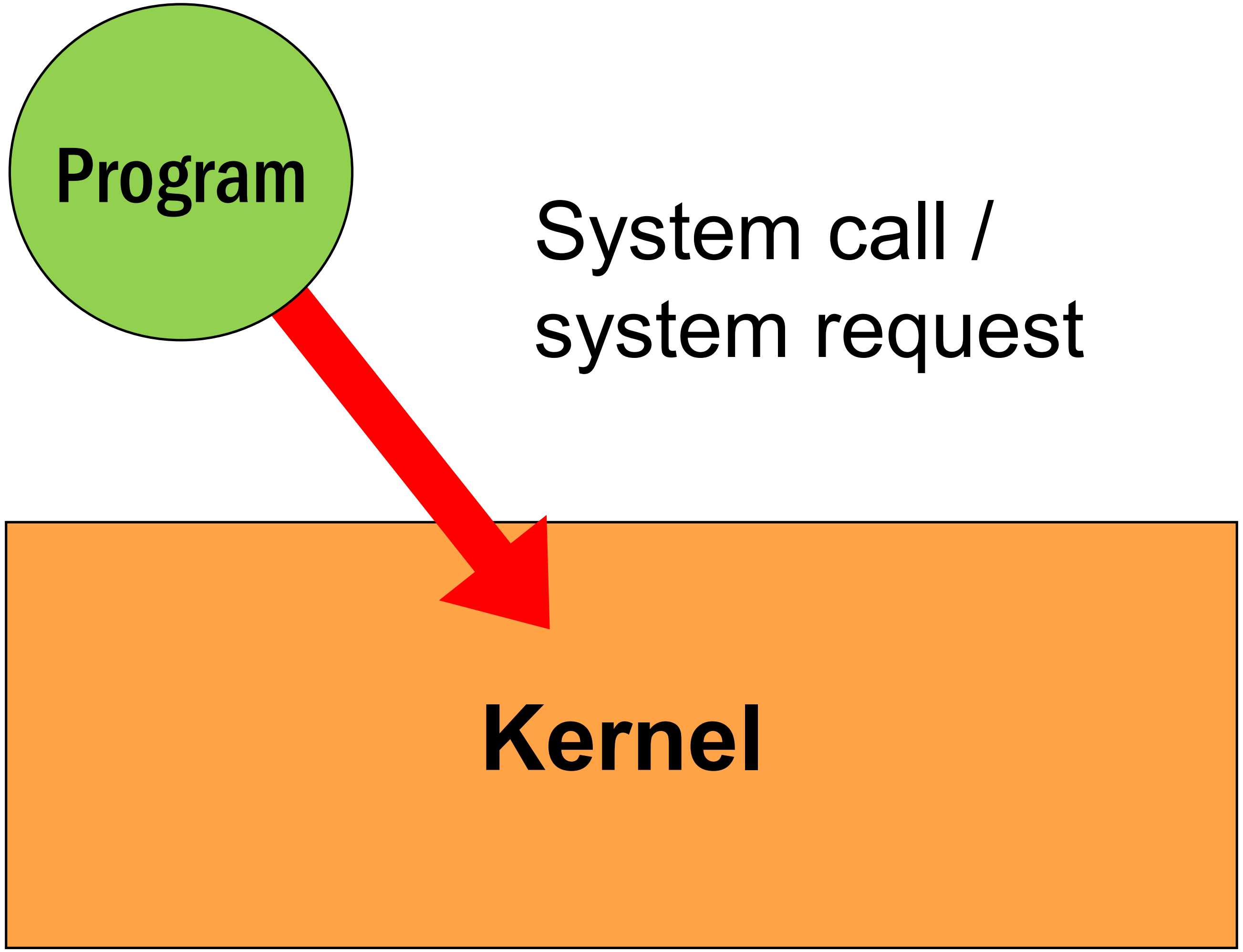
- Unix has several shells
  - ***sh*** (the Bourne shell) is the original Unix shell
  - ***csh*** was developed at Berkeley by Bill Joy
  - ***ksh*** (the Korn shell) was developed by David Korn at AT&T Bell Laboratories
  - ***bash*** (the GNU Bourne-Again shell)

  and the list is far from complete

The core of the OS

- Part that remains in main memory
- Controls the execution of all other programs.
- Known as the ***kernel***
  - Also called ***monitor***, ***supervisor***, ***executive***
- Other programs interact with it through **system calls**

System calls



A question

- Who among you has already used system calls?

The answer

- All of you
  - *All I/O operations are performed through system calls*

The four missions

Missions of an OS

- ***Four*** basic functions
  - To provide a better user interface
  - To manage the system resources
  - To protect users' programs and data
  - To let programs exchange information

A better user interface

- Accessing directly the hardware would be very cumbersome
- Must enter manually the code required to read into main memory each program
  - ***boot strapping***

How it was done (I)

PDP 8

- Early 70’s
- 12-bit machine
  - 4K RAM!

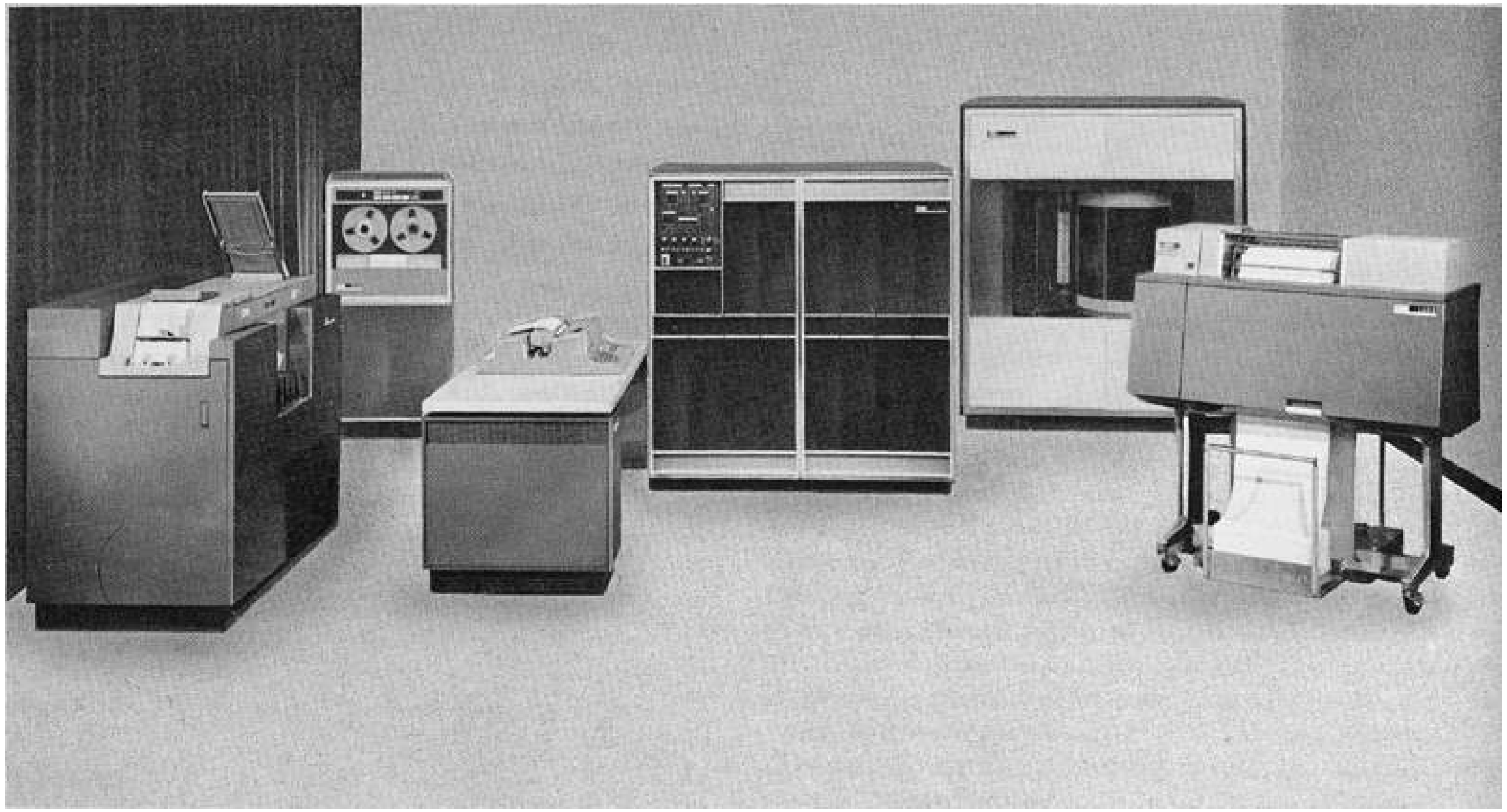
How it was done (II)

Toggle switches in front panel were used to enter the bootstrap code

Batch systems

- Allow users to submit a batches of requests to be processed in sequence
- Include a command language specifying what to do with the inputs
  - Compile
  - Link edit
  - Execute and so forth

An IBM 1401



Interactive systems

- Came later
- Allow users to interact with the OS through their terminals:
- Include an ***interactive*** command language
  - Unix shells, Windows PowerShell
  - Can also be used to write scripts

Time sharing

- Lets several interactive users to access a single computer at the same time
- Standard solution when computers were expensive

Graphical user interfaces

- Called GUIs (pronounced *goo-eys*):
  Macintosh, Windows, X-Windows, Linux
  - Require a dedicated computer for each user
  - Pioneered at Xerox Palo Alto Research Center (Xerox PARC)
  - Popularized by the Macintosh
  - Dominated the market with MS Windows

The Xerox Alto

Xerox PARC (I)

- Founded by XEROX in 1970
- Invented
  - Laser printing
  - Ethernet
  - The GUI paradigm
  - Object-oriented programming (Smalltalk)

Xerox PARC (II)

- All their inventions were brought to market by other concerns
- Popular belief is that Xerox management blew it
- In reality
  - Alto workstations were very expensive
  - Smalltalk was very slow
  - Group was too small to deliver a full system

Smart phones

- Convergence of trends
  - Better cellular connectivity
  - Cheaper LCD displays
  - Solid-State Storage (SSD)
  - Inexpensive wireless networks (WiFi)

History repeats itself

- First successful devices introduced by Apple
  - iPod, iPhone, iPad, …
    - First iPad was underpowered
- Competition soon grows
  - Cheaper Android devices

With a difference!

- Apple did not "steal" the concept from anyone
- iPods, iPhones, iPads were an instant success
  - Reasonably priced

Two models

- ***Apple:***
  - Closed ecosystem (***walled garden***)
  - Strict controls on app market
  - Missing features
    - No file system

- ***Android:***
  - Just the opposite
  - Lax controls on app market
  - Can access Linux/Android shell

Is a walled garden the paradise?

Summary

- ***Six major steps***
  - Bare bone machine
  - Batch systems
  - Timesharing
  - Personal computer
  - Personal computer with GUI
  - Smart phone/tablet

File systems

- Let users create and delete files without having to worry about disk allocation
  - Users lose the ability to specify how their files are stored on the disk
  - Database designers prefer to bypass the file system
- Some file systems tolerate disk failures (RAID)

Managing system resources

- ***Focus of the remainder of the course***
- ***Not an easy task***
  - Enormous gap between CPU speeds and disk access times

The memory hierarchy (I)

Level	Device	Access Time
1	Fastest registers (2 GHz)	**0.5 ns**
2	Main memory	**10-70 ns**
3	Secondary storage (flash)	**35-100 µs**
4	Secondary storage (disk)	**3-12 ms**
5	Mass storage (off line)	**a few s**

The memory hierarchy (II)

- To make sense of these numbers, let us consider an analogy

Writing a paper (I)

Level	Resource	Access Time
1	Open book on desk	1 s
2	Book on desk	
3	Book in UH library	
4	Book in another library	
5	Book very far away	

Writing a paper (II)

Level	Resource	Access Time
1	Open book on desk	**1s**
2	Book on desk	**20-140 s**
3	Book in UH library	
4	Book in another library	
5	Book very far away	

Writing a paper (III)

Level	Resource	Access Time
1	Open book on desk	**1s**
2	Book on desk	**20-140s**
3	Book in UH library	**20-55h**
4	Book in another library	
5	Book very far away	

Writing a paper (IV)

Level	Resource	Access Time
1	Open book on desk	**1 s**
2	Book on desk	**20-140 s**
3	Book in UH library	**20-55 h**
4	Book in another library	**70-277 days**
5	Book very far away	

Writing a paper (V)

Level	Resource	Access Time
1	Open book on desk	**1 s**
2	Book on desk	**20s-140 s**
3	Book in UH library	**20-55 h**
4	Book in another library	**70-277 days**
5	Book very far away	**> 63 years**

Will the problem go away?

- New storage technologies
  - Cheaper than main memory
  - Faster than disk drives
- Flash drives
- Optane memory

Flash drives

- Offspring of EEPROM memories
- Fast reads
  - Block-level
- Slower writes
  - Whole page of data must be erased then rewritten
- Can only go through a finite number of program /erase cycles

Optane memory

- Byte-addressable non-volatile memory (BNVM)
- Simpler design
  - Bits are stored as resistivity levels of a secret alloy
  - No transistors (≠ SRAM and DRAM)
- Faster than flash
  - 100-300 ns
- Dropped by Intel last year

Optimizing disk accesses

- Two main techniques
  - Making disk accesses more efficient
  - Doing something else while waiting for an I/O operation

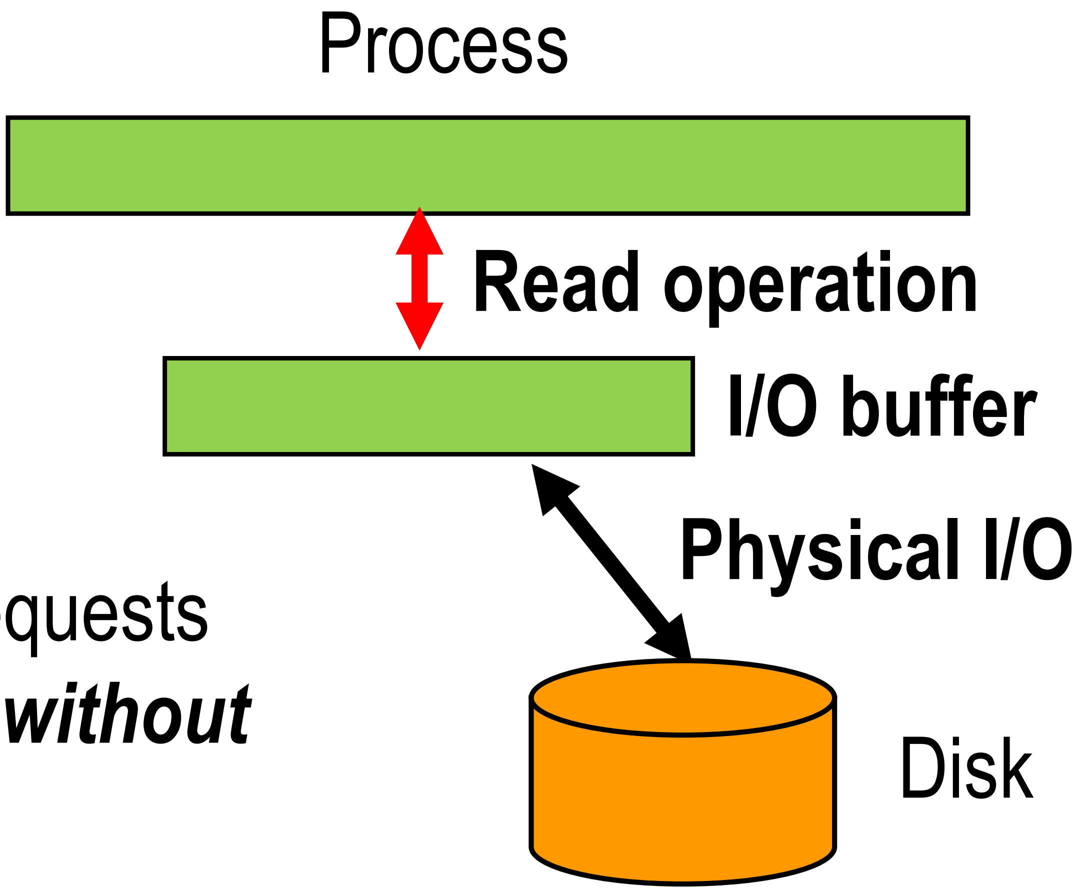
- *Not very different from what we are doing in our every day's lives*

Optimizing read accesses (I)

- *When we shop in a market that's far away from our home, we plan ahead and buy food for several days*

- The OS will read as many bytes as it can during each disk access
  - In practice, entire blocks (4KB or more)
  - Blocks are stored in the I/O buffer

Optimizing read accesses (II)



- Most small read requests can be completed ***without any disk access***

Optimizing read accesses (III)

- Buffered reads work quite well
  - Most systems use it
- Major limitation
  - Cannot read ***too much ahead*** of the program
    - Could end bringing into main memory data that would ***never be used***

Optimizing read accesses (IV)

- Can ***also keep*** in a buffer recently accessed blocks hoping they will be accessed again
  - ***Caching***
- Works very well because we keep accessing again and again the data we are working with
- ***Caching is a fundamental technique of OS and database design***

Optimizing write accesses (I)

- *If we live far away from a library, we wait until we have several books to return before making the trip*
- The OS will ***delay writes*** for a few seconds then write an entire block
  - Since most writes are sequential, most small writes will not require any disk access

Optimizing write accesses (II)

- ***Delayed writes*** work quite well
  - Most systems use it
- ***Major drawback***
  - We will ***lose data*** if the system or the program crashes
    - After the program issued a write but
    - Before the data were saved to disk
  - Unless we use NVRAM

Doing something else

- *When we order something on the web, we do not remain idle until the goods are delivered*

- The OS can implement ***multiprogramming*** and let the CPU run another program while a program waits for an I/O

Advantages (I)

- Multiprogramming is very important in business applications
  - Many of these applications use the peripherals much more than the CPU
  - For a long time the CPU was the most expensive component of a computer
  - ***Multiprogramming*** was invented to keep the CPU busy

Advantages (II)

- Multiprogramming made ***time-sharing*** possible
- Multiprogramming lets your PC run several applications at the same time
  - MS Word and MS Outlook

Multiprogramming (I)

- Multiprogramming lets the CPU divide its time among different tasks:
  - One tenth of a second on a program, then another tenth of a second on another one and so forth
- Each core of your CPU will still be working on ***one single task*** at any given time

Multiprogramming (II)

- The CPU does not waste any time waiting for the completion of I/O operations
- From time to time, the OS will need to regain control of the CPU
  - Because a task has exhausted its fair share of the CPU time
  - Because something else needs to be done.
- This is done through ***interrupts*****.**

Interrupts (I)

- Request to interrupt the flow of execution the CPU
- Detected by the CPU hardware
  - ***After*** it has executed the current instruction
  - ***Before*** it starts the next instruction.

A very schematic view (I)

- A very basic CPU would execute the following loop:

```
forever {
    fetch_instruction();
    decode_instruction();
    execute_instruction();
}
```

- Pipelining makes things more complicated
  - And CPU much faster!

A very schematic view (II)

- We add an extra step:

```
forever {
    check_for_interrupts();
    fetch_instruction();
    decode_instruction();
    execute_instruction();
}
```

Interrupts (II)

- When an interrupt occurs:

a. The ***current state of the CPU*** (program counter, program status word, contents of registers, and so forth) is saved, normally on the top of a stack

b. A ***new CPU state*** is fetched

Interrupts (III)

- New state includes a new ***hardware-defined*** value for the program counter
  - Cannot “hijack” interrupts
- Process is totally transparent to the task being interrupted
  - A process ***never*** knows whether it has been interrupted or not

Types of interrupts (I)

- ***I/O completion interrupts***
  - Notify the OS that an I/O operation has completed,
- ***Timer interrupts***
  - Notify the OS that a task has exceeded its quantum of core time

Types of interrupts (II)

- ***Traps***
  - Notify the OS of a ***program error*** (division by zero, illegal op code, illegal operand address, ...) or a *hardware failure*
- ***System calls***
  - Notify OS that the running task wants to submit a request to the OS

A surprising discovery

- ***Programs do interrupt themselves!***

Context switches

- Each interrupt will result into ***two context switches***:
  - One when the running task is interrupted
  - Another when it regains the CPU
- Context switches are ***not cheap***
- The overhead of any simple system call is
  two context switches

Remember that!

Prioritizing interrupts (I)

- Interrupt requests may occur while the system is processing another interrupt
- All interrupts are not equally urgent (as it is also in real life
  - Some are more urgent than other
  - *Also true in real life*

Prioritizing interrupts (II)

- The best solution is to *prioritize* interrupts and assign to each source of interrupts a ***priority level***
  - New interrupt requests will be allowed to interrupt lower-priority interrupts but will have to wait for the completion of all other interrupts
- Solution is known as ***vectorized interrupts***.

Example from real life

- Let us try to prioritize
  - Phone is ringing
  - Washer signals end of cycle
  - Dark smoke is coming out of the kitchen
  - …
- With vectorized interrupts, a phone call will never interrupt another phone call

The solution

- Smoke in the kitchen
- Phone is ringing
- End of washer cycle
- More low-priority stuff

Disabling Interrupts

- We can ***disable*** interrupts
- OS does it before performing short critical tasks that cannot be interrupted
  - Works only for single-threaded kernels
- User tasks ***must*** be prevented from doing it
  - Too dangerous

DMA

- Disk I/O poses a special problem
  - CPU will have to transfer large quantities of data between the disk controller's buffer and the main memory
- ***Direct memory access*** (***DMA***) allows the disk controller to read data from and write data to main memory without any CPU intervention
  - Controller “steals” memory cycles from CPU

Protecting users' data (I)

- Unless we have an isolated single-user system, we must prevent users from
  - Accessing
  - Deleting
  - Modifying

  without authorization other people's programs and data

Protecting users’ data (II)

- Two aspects
  - Protecting user's files on disk
  - Preventing programs from interfering with each other
- Two solutions
  - Dual-mode CPUs
  - Memory protection

Historical Considerations

- Earlier operating systems for personal computers did not have any protection
  - They were single-user machines
  - They typically ran one program at a time

- Windows 2000, Windows XP, Vista and MacOS X are protected

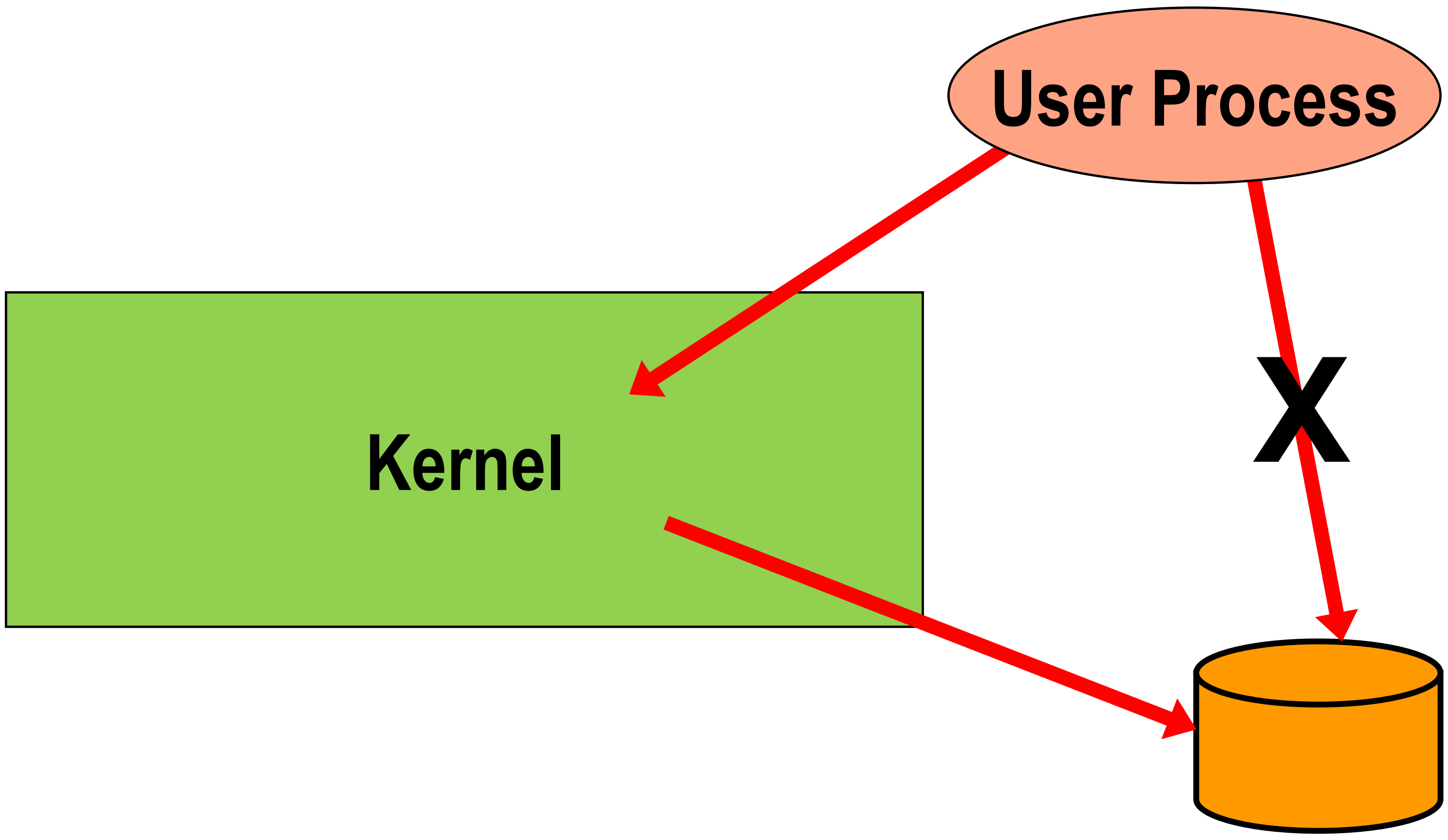
Protecting users' files

- Key idea is to prevent users' programs from directly accessing the disk
- Will require I/O operations to be performed by the kernel
- Make them ***privileged instructions***
  - Only the kernel can execute

Privileged instructions

- Require a ***dual-mode CPU***
- Two CPU modes
  - ***Privileged mode*** or ***executive mode***
    - Allows CPU to execute all instructions
  - ***User mode***
    - Allows CPU to execute only safe unprivileged instructions
- State of CPU is determined by a ***special bit***

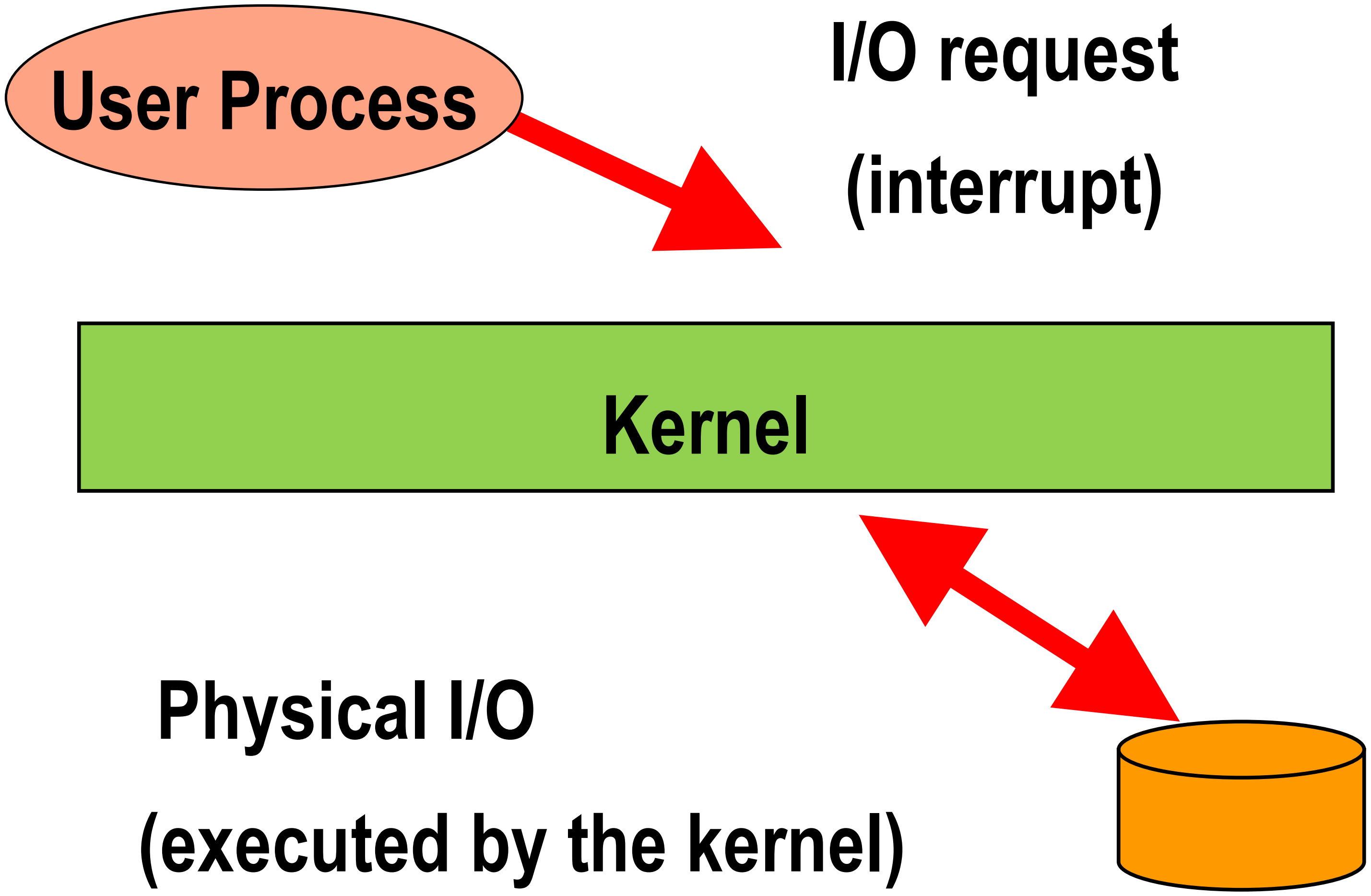
All disk/SSD accesses must go through the kernel



Switching between states

- User mode will be the default mode for all programs
  - Only the kernel can run in supervisor mode
- Switching from user mode to supervisor mode is done through an interrupt
  - Safe because the jump address is at a well-defined location in main memory

Performing an I/O



An analogy (I)

- Most UH libraries are ***open stacks***
  - Anyone can consult books in the stacks and bring them to checkout
- National libraries and the Library of Congress have ***closed stack collections***
  - Users fill a request for a specific document
  - A librarian will bring the document to the circulation desk

An analogy (II)

- ***Open stack collections***
  - Let users browse the collections
  - Users can misplace or vandalize books
- ***Closed stack collections***
  - Much slower access
  - Much safer

More trouble

- Having a dual-mode CPU is ***not enough*** to protect user's files
- Must also prevent rogue users from tampering with the kernel
  - Same as a rogue customer bribing a librarian in order to steal books
- Done through ***memory protection***

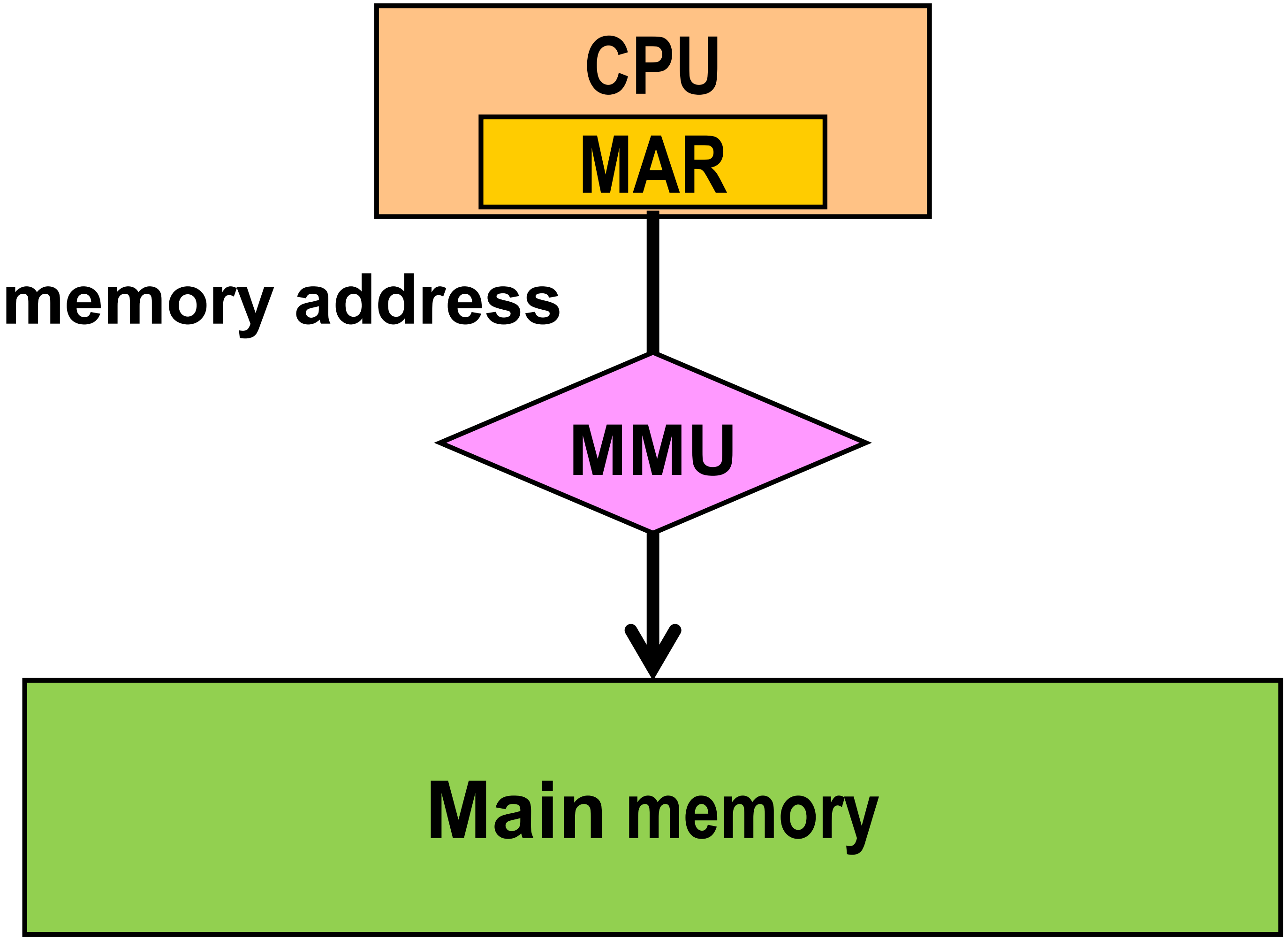
Memory protection (I)

- Prevents programs from accessing any memory location outside their own ***address space***
- Requires special ***memory protection hardware***
  - ***Memory Management Unit*** **(MMU)**
- Memory protection hardware
  - Checks ***every*** reference issued by program
  - Generates an interrupt when it detects a protection violation

Memory protection (II)

- Has additional advantages:
  - Prevents programs from corrupting address spaces of other programs
  - Prevents programs from crashing the kernel
    - Not true for device drivers which are inside the kernel
- Required part of any multiprogramming system

Memory protection (III)



Even more trouble

- Having both a dual-mode CPU and memory protection is not enough to protect user's files
- Must also prevent rogue users from booting the system with a ***doctored kernel***
  - ***Example:***
    - Can run Linux from a "live" CD Linux
    - Linux will read all NTFS files ignoring all restrictions set up by Windows

Inter-process communication

- Has become very important over the last thirty years
- Two techniques
  - Message passing
    - General but not very easy to use
  - Shared memory
    - Less general, easier to use but requires inter-process synchronization

ANOTHER VIEW

- Arpaci-Dusseau & Arpaci-Dusseau
  - Focus on services provided by OSes
- Three themes
  - Virtualization
  - Concurrency
  - Persistence

Virtualization

- The process abstraction
- Virtualizing the CPU:
  - Process scheduling
- Virtualizing the memory:
  - Memory management

Concurrency

- Threads
- Locks
- Semaphores

We will cover threads in the chapter on processes because they are essential to the client-server model

Persistence

- The file system

Types of operating systems

Overview

- Already discussed:
  - Batch systems
  - Time-sharing systems
- Will now introduce
  - Real-Time systems
  - Operating systems for multiprocessors
  - Distributed systems

Real-time systems

- Designed for applications with ***strict real-time constraints*** :
  - ***Process control***
  - ***Guidance systems***
  - Most ***multimedia applications***
- Must guarantee that critical tasks will ***always*** be performed within a specific time frame.

Hard RT systems

- Must guarantee that all deadlines will always be met
- ***Any failure*** could have ***catastrophic consequences***:
  - The reactor could overheat and explode
  - The rocket could be lost

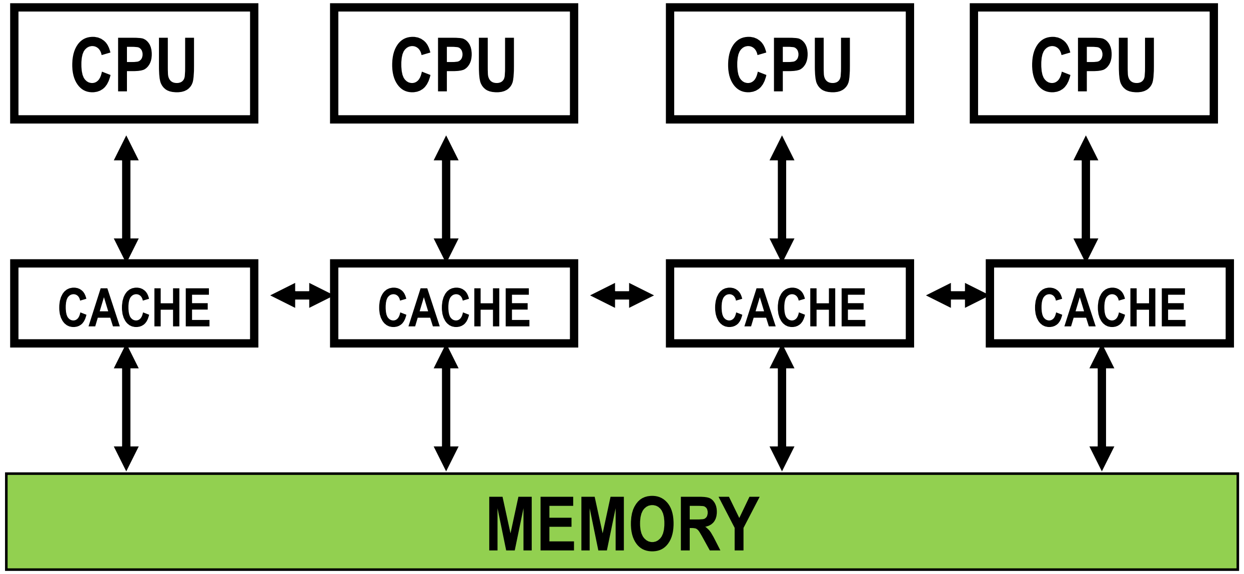
Soft RT systems

- Guarantee that most deadlines will be met
- A DVD decoder that miss a deadline will spoil our viewing pleasure for a fraction of a second

Observations

- Hard RT applications normally run on special RT OSes
- Soft RT applications can run on a regular OS
  - If the OS supports them
- Interactive and time-sharing systems are ***not*** RT systems
  - They attempt to provide a fast response time but do not try to meet specific deadlines

Multiprocessor operating systems



- Designed for ***multiprocessor architectures***
  - Several processors share the same memory

Leader/follower multiprocessing

- Single copy of OS runs on a dedicated core/processor
  - ***Leader*** (previously called *master*)
- Other cores/processors can only run applications
  - ***Followers*** (previously called *slaves*)
- Major advantage is ***simplicity***
  - Requires few changes
- Major disadvantage is ***lack of scalability***
  - Single copy of OS can become a ***bottleneck***

Symmetric multiprocessing

- Any core/processor can perform all functions
  - There can be multiple copies of the OS running in parallel
- Must prevent them from interfering with each other
  - Disabling interrupts will not work
  - Must add ***locks*** to all critical sections

The state of the art

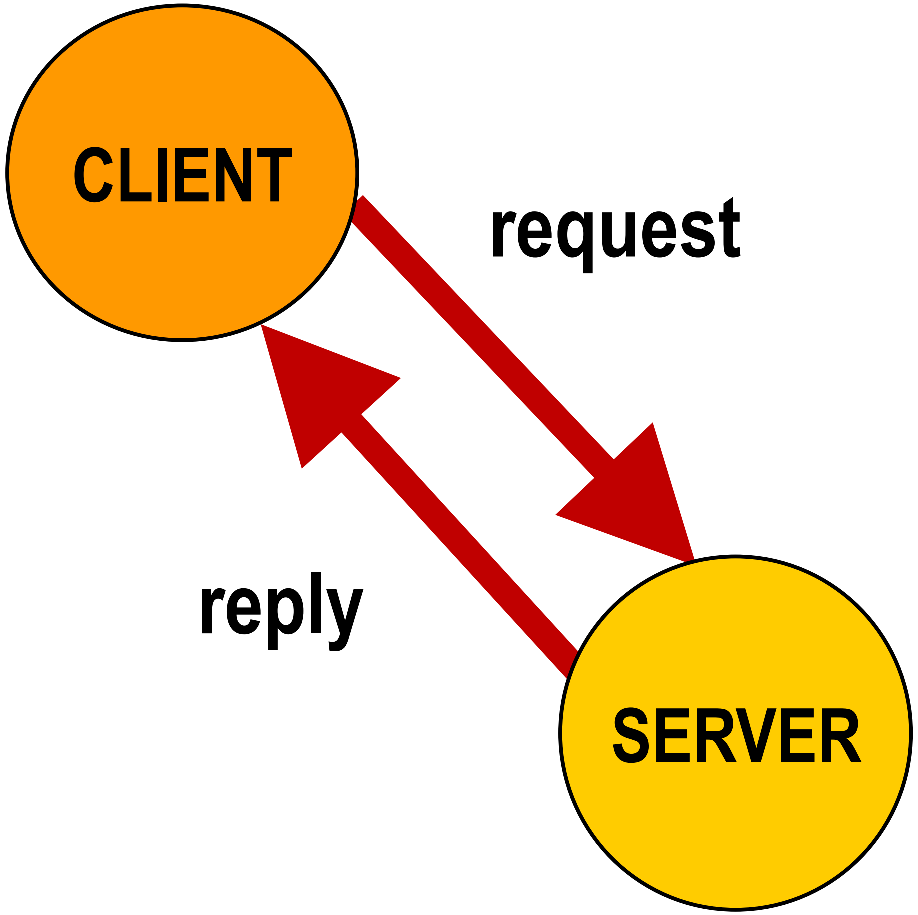
- Most computers now have ***multicore CPUs***
  - Sole practical way to increase CPU power
- Many have powerful GPUs
  - Highly parallel
- Using multicore architectures in an effective way is a huge challenge

Distributed systems

- Integrated networks of computers
  - ***Workstations*** sharing common resources (file servers, printers, …)
- Current trend is to leave systems very loosely coupled
  - Each computer has its own OS

Client /Server Model

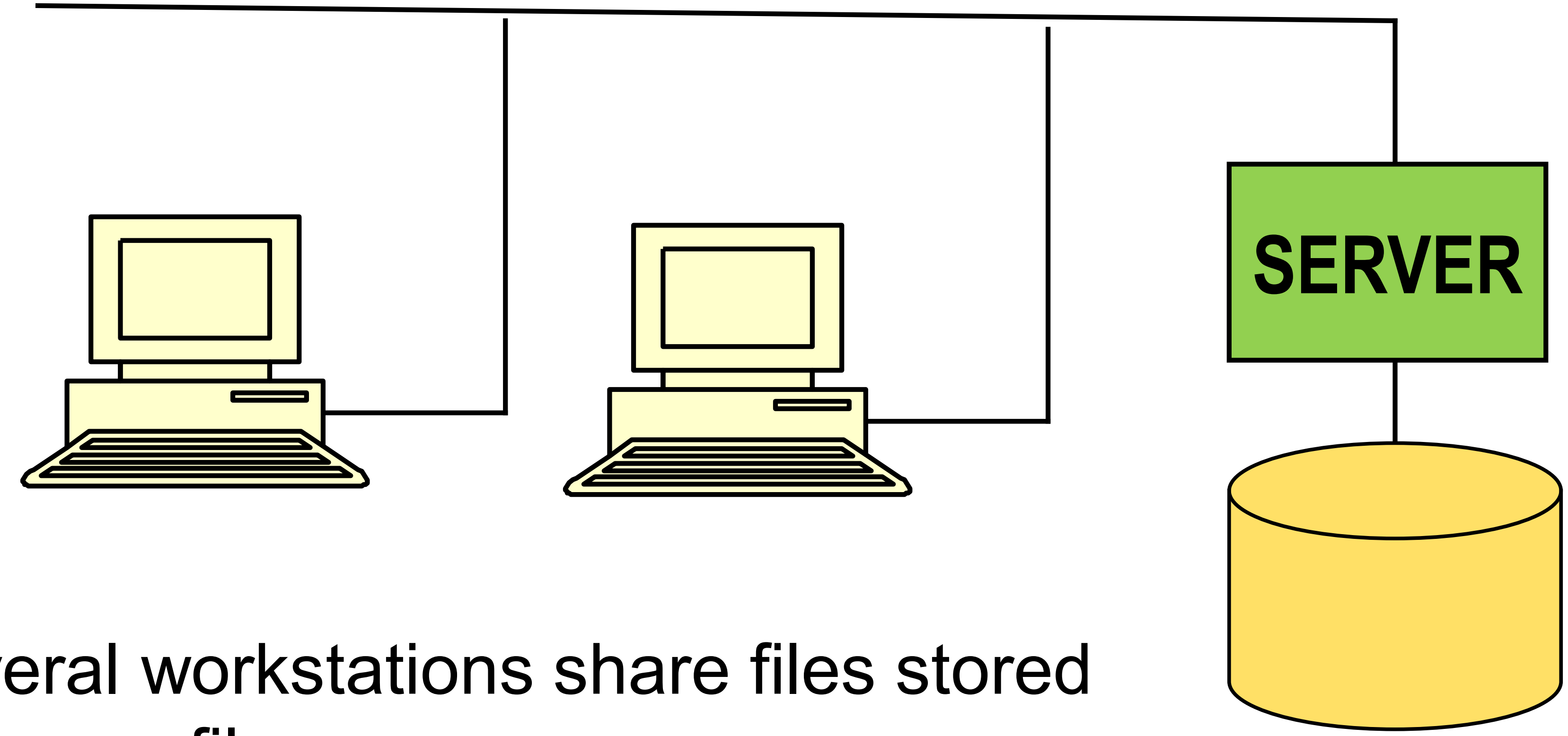
- Servers wait for requests from clients and process them
  - File servers
  - Print servers
  - Authentication servers



A typical sequential server

```
for (;;){
   //wait for request
   get_request(…);
   // process it
   process_request(…);
   // send reply
   send_reply(…);
} // forever
```

Network file system

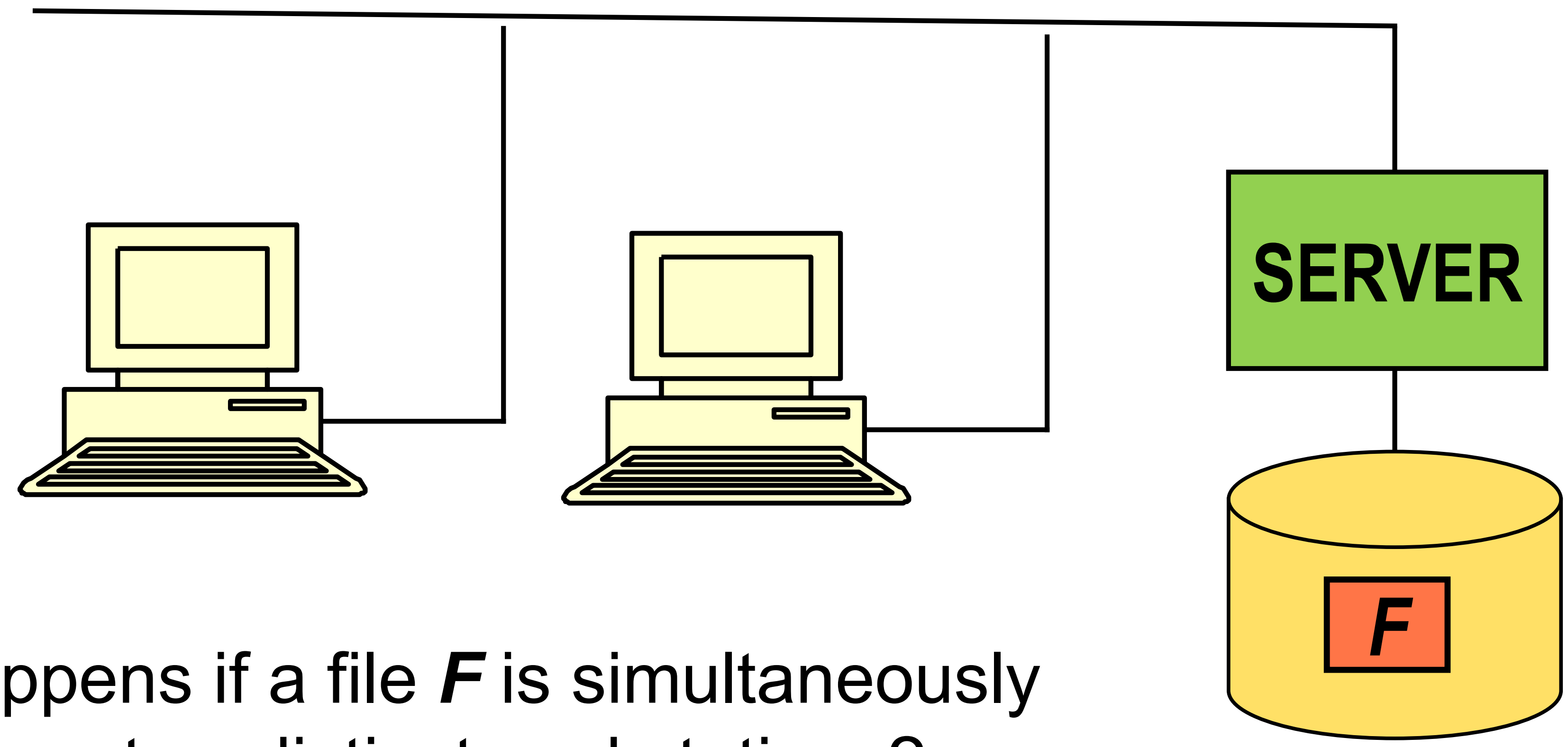


- Lets several workstations share files stored on a common file server

Performance Issues

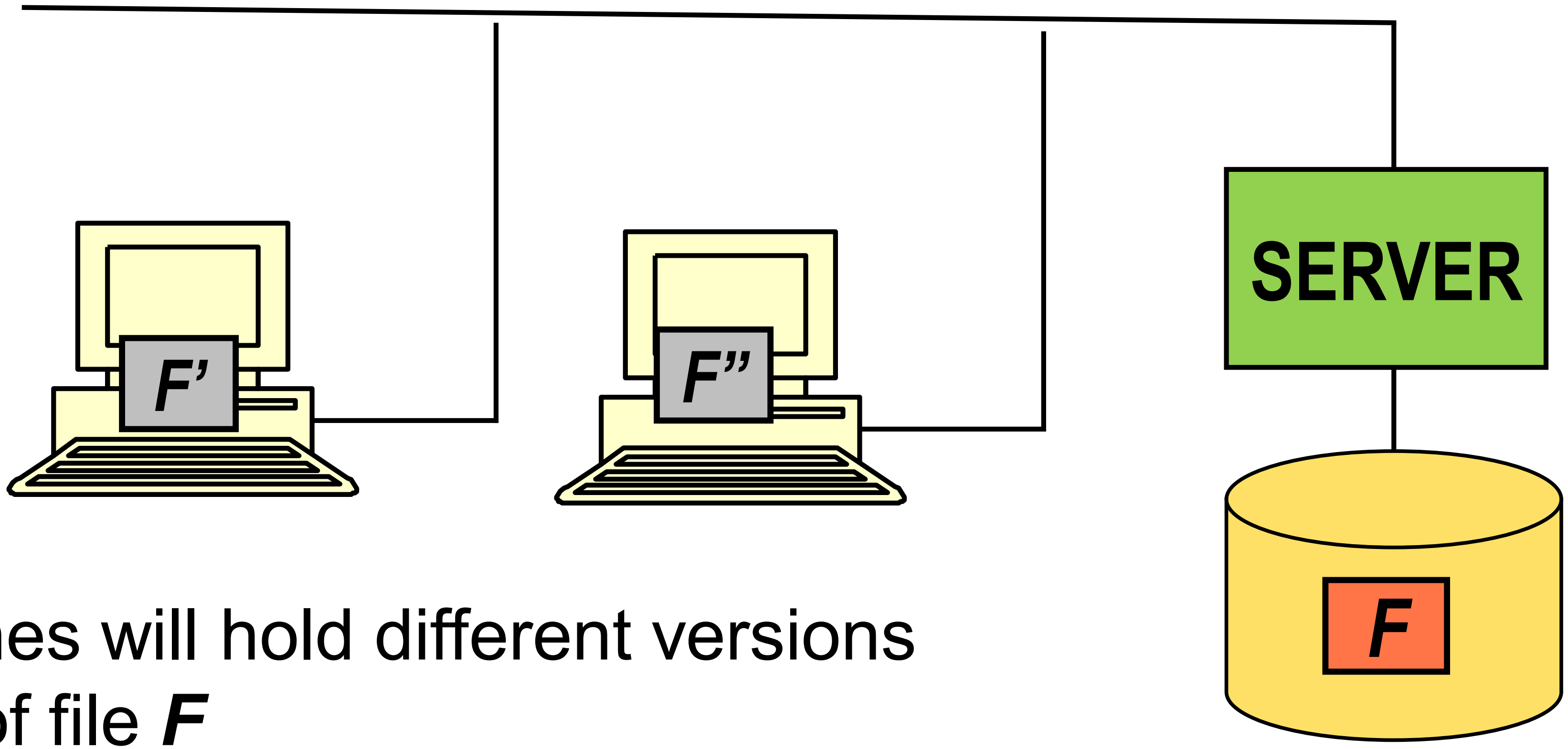
- ***Response time*** is the main issue
  - ***Network latency*** is now added to ***disk latency***
- Will attempt to mask these two latencies
  - Extensive ***client caching***
  - ***Works very well***

File consistency issues (I)



- What happens if a file ***F*** is simultaneously modified on two distinct workstations?

File consistency issues (II)



- Client caches will hold different versions ***F'*** and ***F''*** of file ***F***

File Consistency Issues (III)

- Maintaining file consistency is a very important issue in distributed/networked file system design
- Different systems use different approaches
  - NFS from Sun Microsystems
  - AFS/Coda from CMU
  - …

Other distributed systems issues

- **Authenticating** users
  - *A problem in open networks*
- Making distributed systems as *reliable* as stand-alone systems
  - ***Replication*** of data and services
- Keeping the clocks of the machines more or less synchronized.

Unix and Linux

Unix (I)

- Started at Bell Labs in the early 70's as an attempt to build a sophisticated time-sharing system on a very small minicomputer.
- First OS to be almost entirely written in C
- Ported to the VAX architecture in the late 70’s at U. C. Berkeley:
  - Added virtual memory and networking

The fathers of Unix

Ken Thompson and Denis Ritchie

Unix (II)

- Became the standard operating systems for ***workstations***
  - Selected by Sun Microsystems
- Became less popular because
  - Too many variants
    - Berkeley BSD, ATT System V, …
  - PCs displaced workstations
  - Windows is easier to use
    - Especially by newbies!

Unix Today

- Several ***free versions*** exist (FreeBSD, Linux):
  - Free access to source code
    - Ideal platform for OS research
- ***Apple OS X*** runs on the top of an updated version of BSD
- ***Android*** runs on top of a heavily customized Linux kernel
- ***Chrome*** runs on top of a vanilla Linux OS

A Rapid Tour

- Unix kernel is the core of the system and handles the system calls
- Unix has several shells: **sh, csh**, **ksh**, **bash**
- On-line command manual:
  - **man xyz**
    displays manual page for command **xyz**
  - **man 2 xyz**
    displays manual page for *system call* **xyz(…)**

Most Lasting Impact

- First OS that
  - Run efficiently on very different platforms
  - Had its source code made available to its users
- File system inspired most more recent OSes
- Remains the best platform for OS research

Kernel organizations

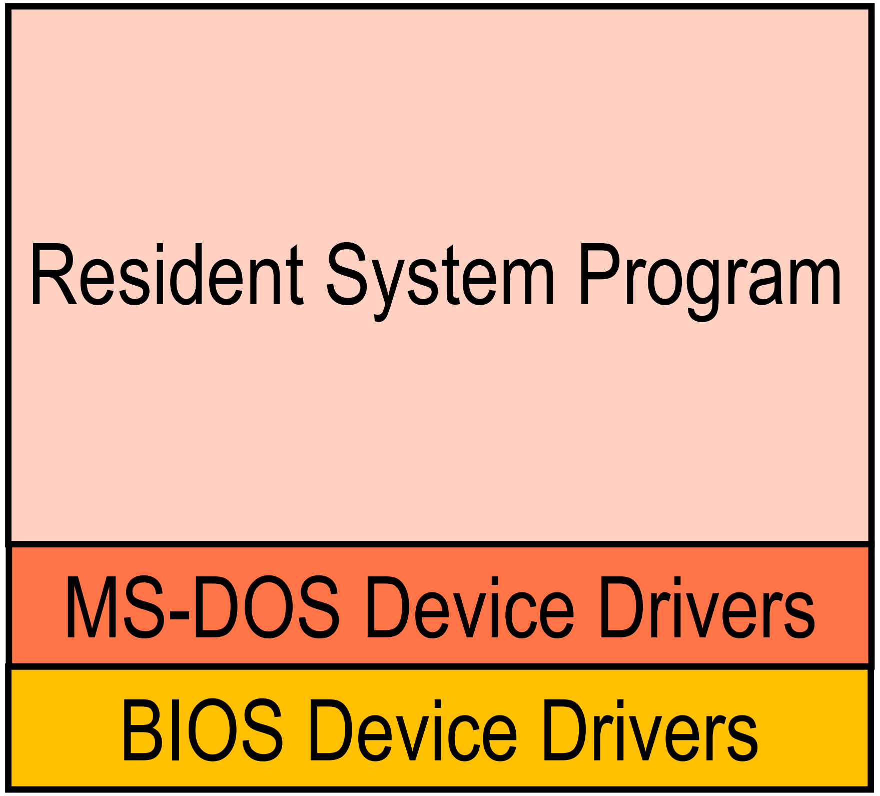
Three basic organizations

- ***Monolithic kernels***:
  - The default
- ***Layered kernels:***
  - A great idea that did not work
- ***Microkernels:***
  - Hurt by the high cost of context switches

Monolithic kernels

- No particular organization
  - All kernel functions share the same address space
  - This includes ***device drivers*** and other ***kernel extensions***
- ***Fastest***
- Lack of internal organization makes the kernel ***hard to manage, extend, and debug***

MS-DOS


Resident System Program
MS-DOS Device Drivers
BIOS Device Drivers

The BIOS

- Basic Input-Output System
- Stored on a chip
  - First ROM, now EEPROM
- Takes control of CPU when system is turned on
  - Identifies system components
  - Initiates booting of operating system
- Also provides low-level I/O access routines

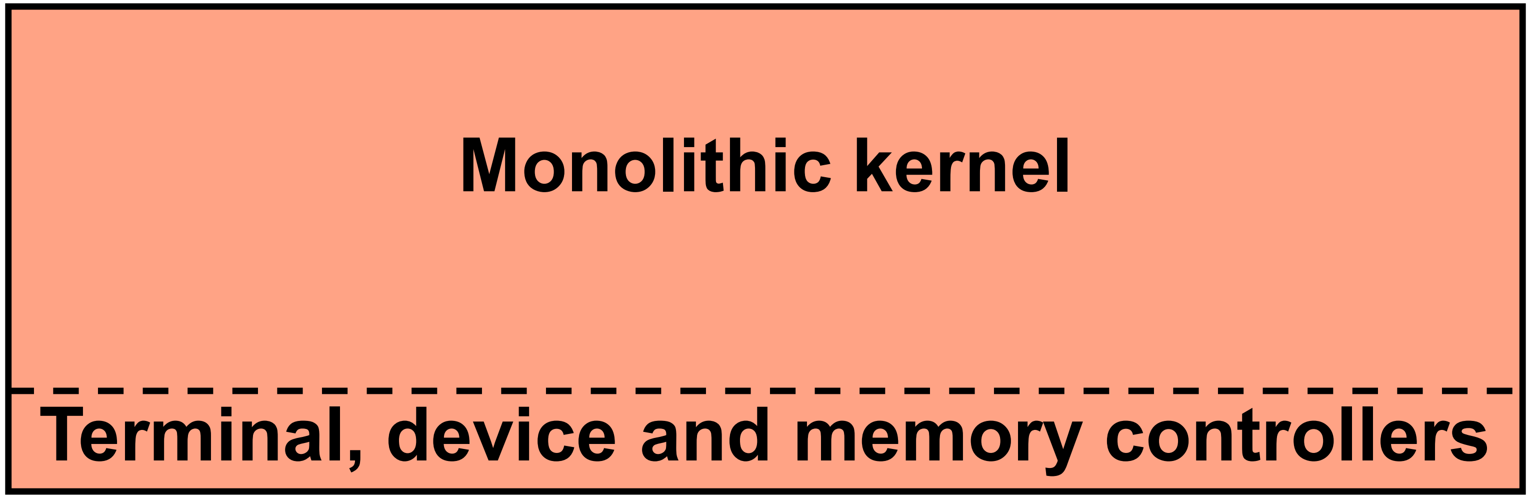
The “curse”

- Hardware lacked dual mode and hardware memory protection
  - Nothing prevented application programs from accessing directly the BIOS
  - Program accessing disk files through BIOS I/O routines assumed a given disk organization
    - Changing it became impossible

Its impact

- For a long time, Microsoft could not make radical changes to its FAT-16 disk organization
- Windows XP and all modern operating systems prevent user programs from bypassing the kernel.

Unix



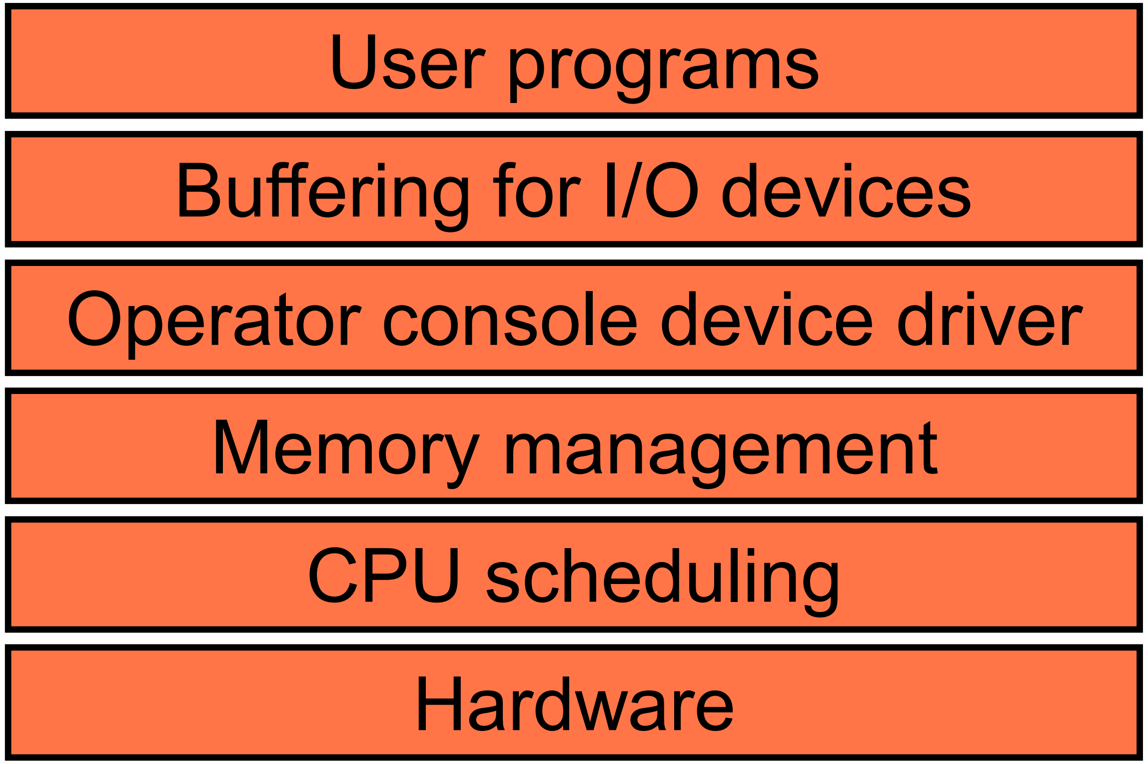
- Monolithic kernel contains everything that is ***not device-specific*** including file system, networking code, and so forth.

Layered kernel

- Proposed by Edsger Dijkstra
- Implemented as a hierarchy of **layers**:
- Each layer defines a new data object
  - Hiding from the higher layers some functions of the lower layers
  - Providing some new functionality

THE operating system kernel

- (named after Dutch initials of T. U. Eindhoven)



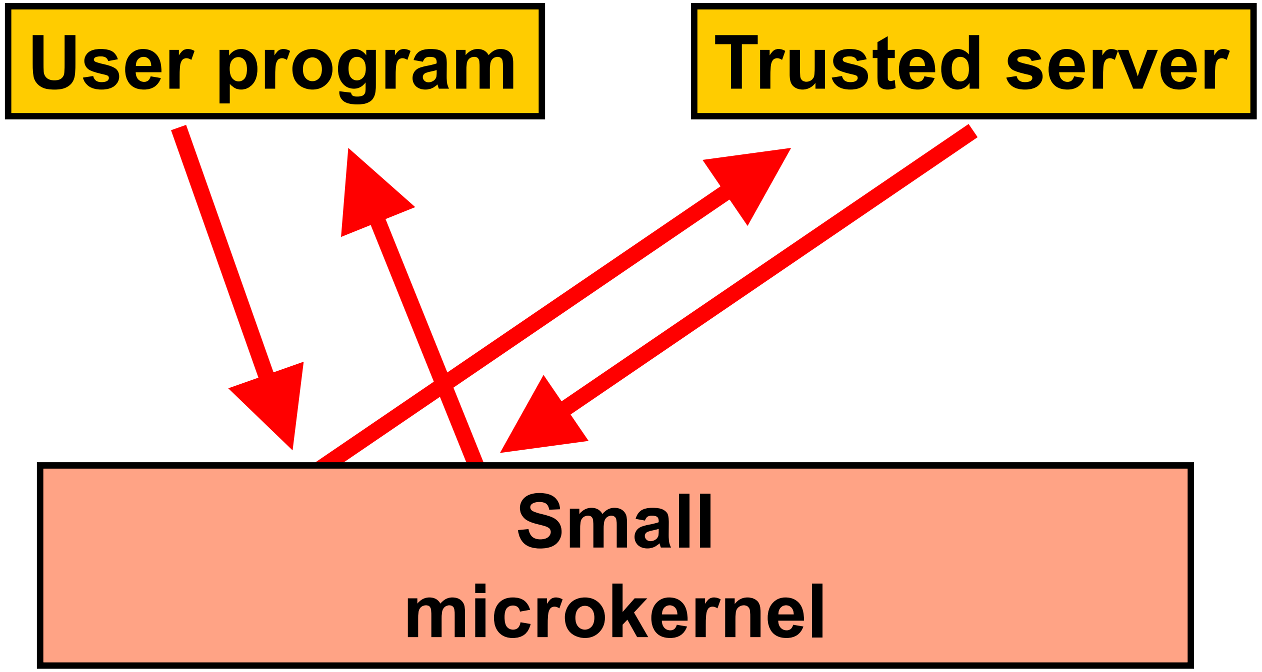
Limitations

- Layered design works extremely well for ***networking code***
  - Each layer offers its own functionality
- Much less successful for kernel design
  - No clear ordering of layers
    - Memory management uses file system features and vice versa

Microkernels

- A reaction against “bloated” monolithic kernels
  - Hard to manage, extend, debug and ***secure***
- Key idea is making kernel ***smaller*** by delegating ***non-essential tasks*** to ***trusted user-level servers***
  - *Same idea as **subcontracting***
- Microkernel keeps doing what cannot be delegated:
  - Security, short-term scheduling, …

How it works (I)



How it works (II)

- Microkernel
  - Receives request from user program
  - Decides to forward it to a user-level server
  - Waits for reply for server
  - Forwards it to user program
- ***Trusted servers*** run outside the kernel
  - Cannot execute privileged instructions

Advantages

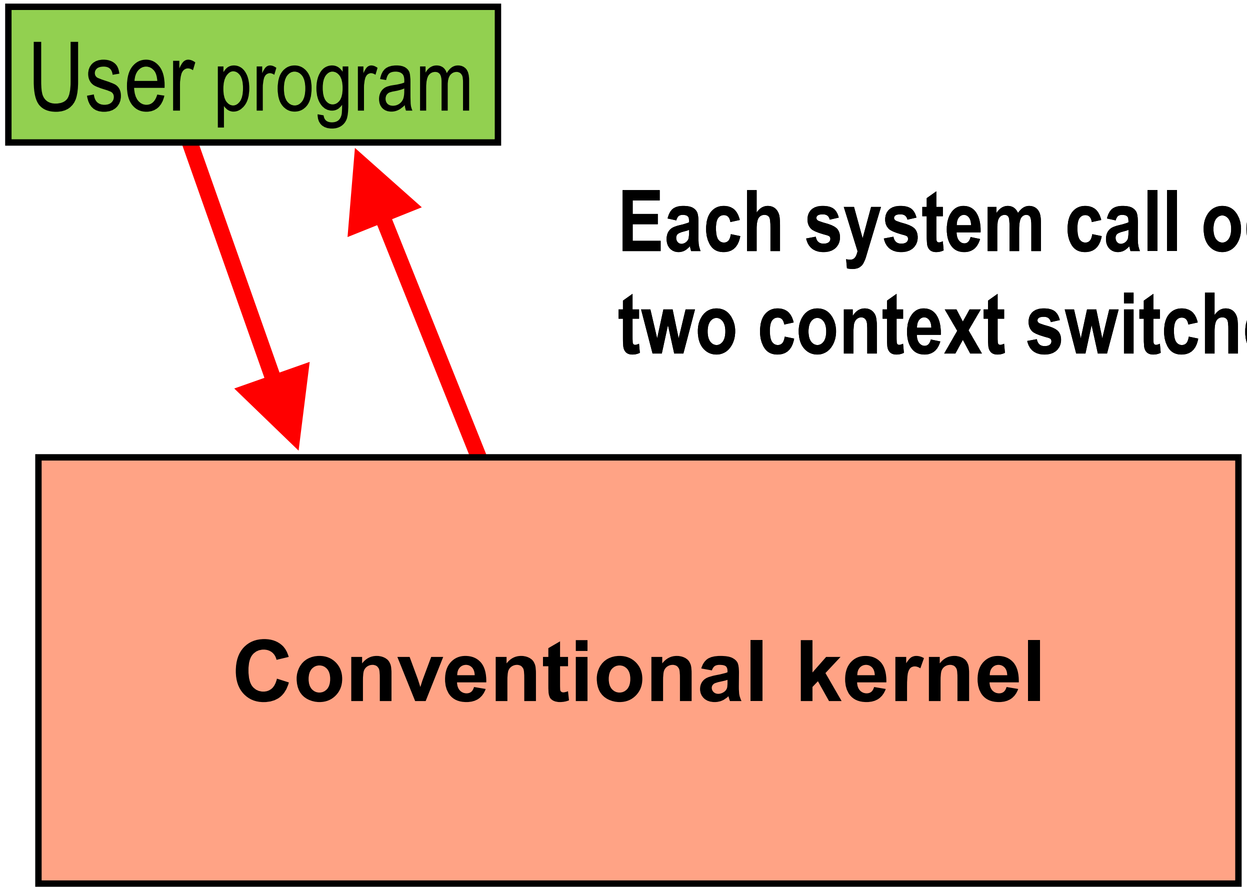
- Kernel is smaller, easier to secure and manage
- Servers run outside of the kernel
  - Cannot crash the kernel
  - Much easier to extend kernel functionality
    - Adding new servers
    - Adding an NTFS server to Unix microkernel

Major disadvantage

- ***Too slow***
  - Four context switches instead of two

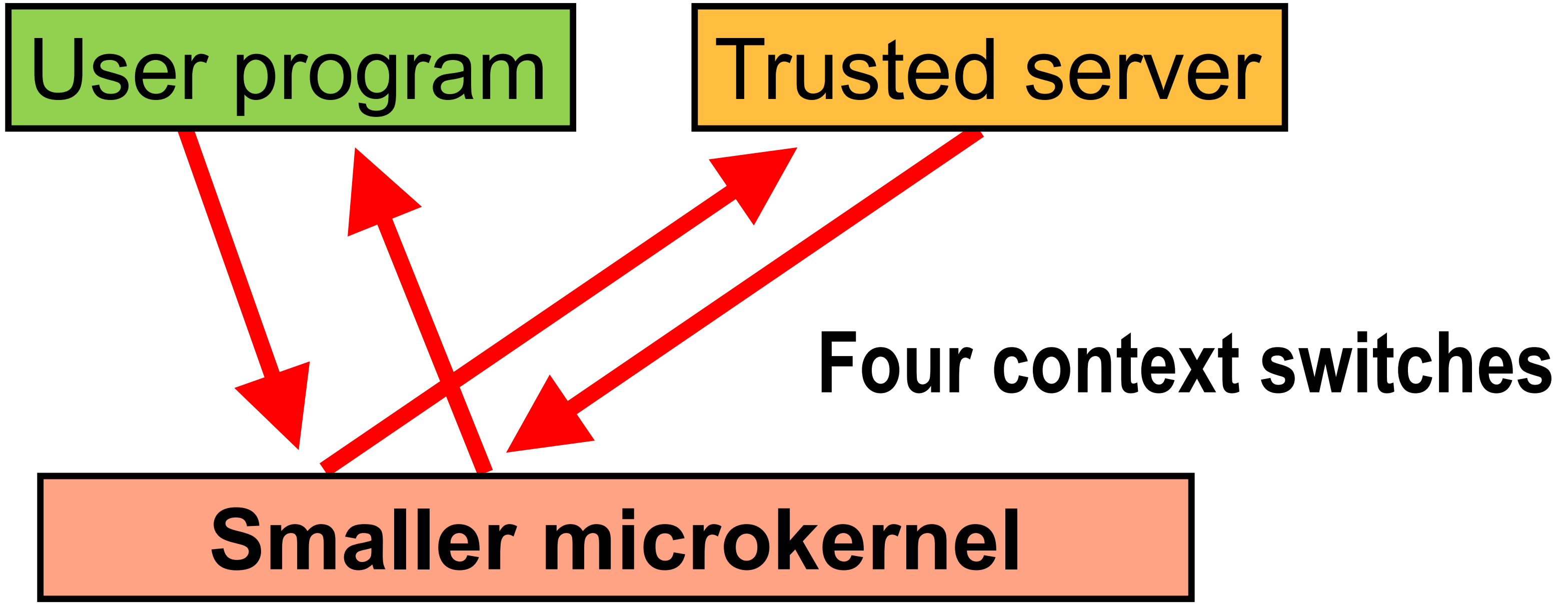
> - ***Speed remains an essential concern***
> - ***We don't like to trade speed for safety (or anything else)***

A conventional kernel



Each system call occasions two context switches

A microkernel



Mach

- Designed in mid 80's to replace Unix kernel
- New kernel with different system calls
  - Unix system calls are routed to an ***emulation server***
- Emulation server was designed to run in user space
  - Slowed down the system
  - Server ended inside the kernel

MINIX 3

- MINIX 1 was designed for teaching OS internals
  - Predates Linux
- Now aimed at high reliability (embedded) applications
  - *More willing to trade space for reliability*
- Runs on x86 and ARM processors
- Compatible with NetBSD

MINIX 3 microkernel

- "Tiny" (12,700 lines) microkernel
  - Handles ***interrupts*** and ***message*** passing
  - Only code running in kernel mode
- Other OS functions are handled by ***isolated***, ***protected***, ***user-mode*** processes
  - Each device driver is a separate user-mode process
  - System automatically restarts ***crashed drivers***

Modular kernels

- Linux, Windows
- Modules are object files whose contents can be linked to—and unlinked from—the kernel at any time
  - Run inside the kernel address space
  - *Used to add to the kernel **device drivers** for new devices*

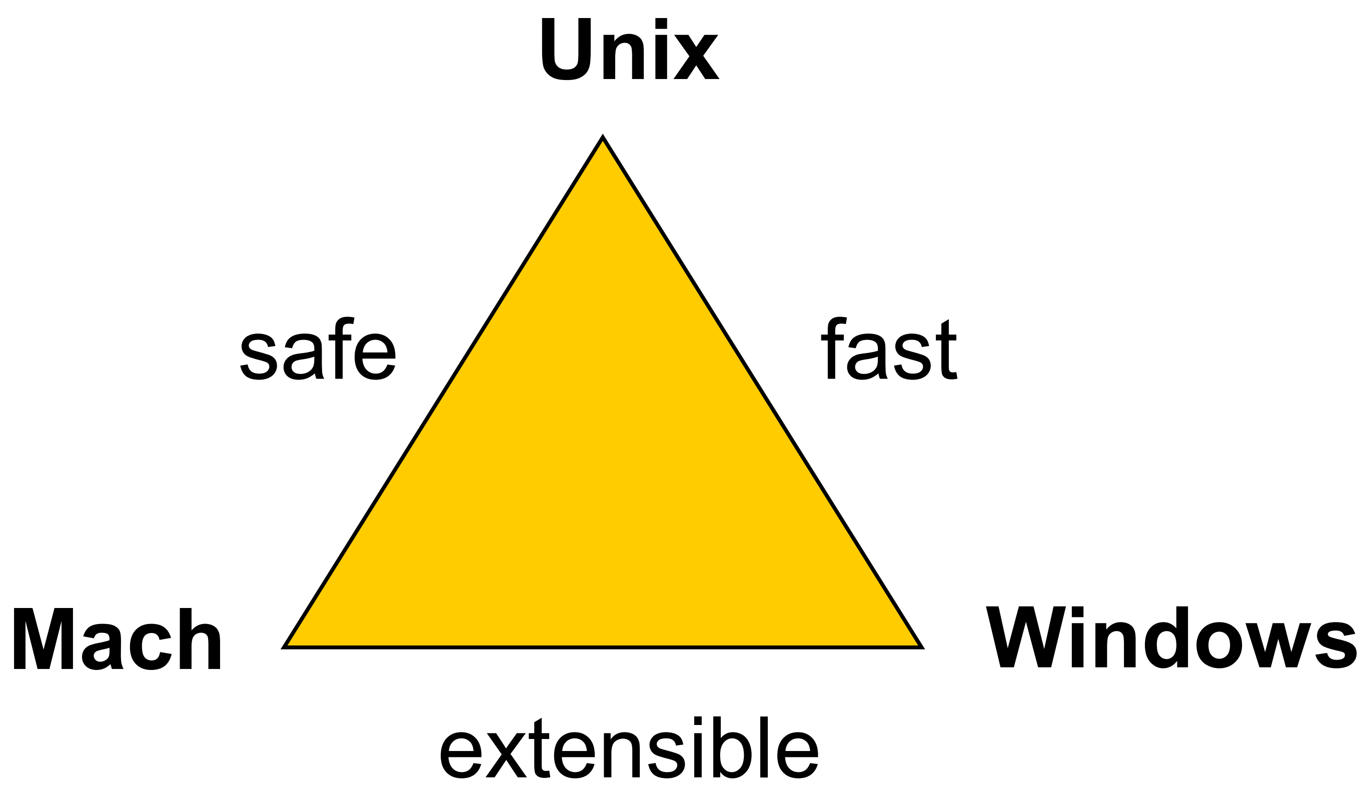
Advantages of modular kernels

- ***Extensibility:***
  - Can add new features the kernel
  - In many cases, the process is completely transparent to the user
- ***Lack of performance penalty:***
  - Modules run in the kernel address space

Their disadvantages

- ***Lower reliability***
  - A bad module can corrupt the whole kernel and crash the system.
- ***Serious problem***
  - Many device drivers are poorly written
  - Device drivers account for 85% of reported failures of Windows XP

Current state of the art



Why?

- Unix has a monolithic kernel (which makes it fast) and does not allow extensions (which makes it both safe and non-extensible)
- Windows has a monolithic kernel (which makes it fast) and allows extensions (which makes it both extensible and unsafe)
- Mach allows extensions in user space (which makes it extensible, safe and slow)

Virtual machines

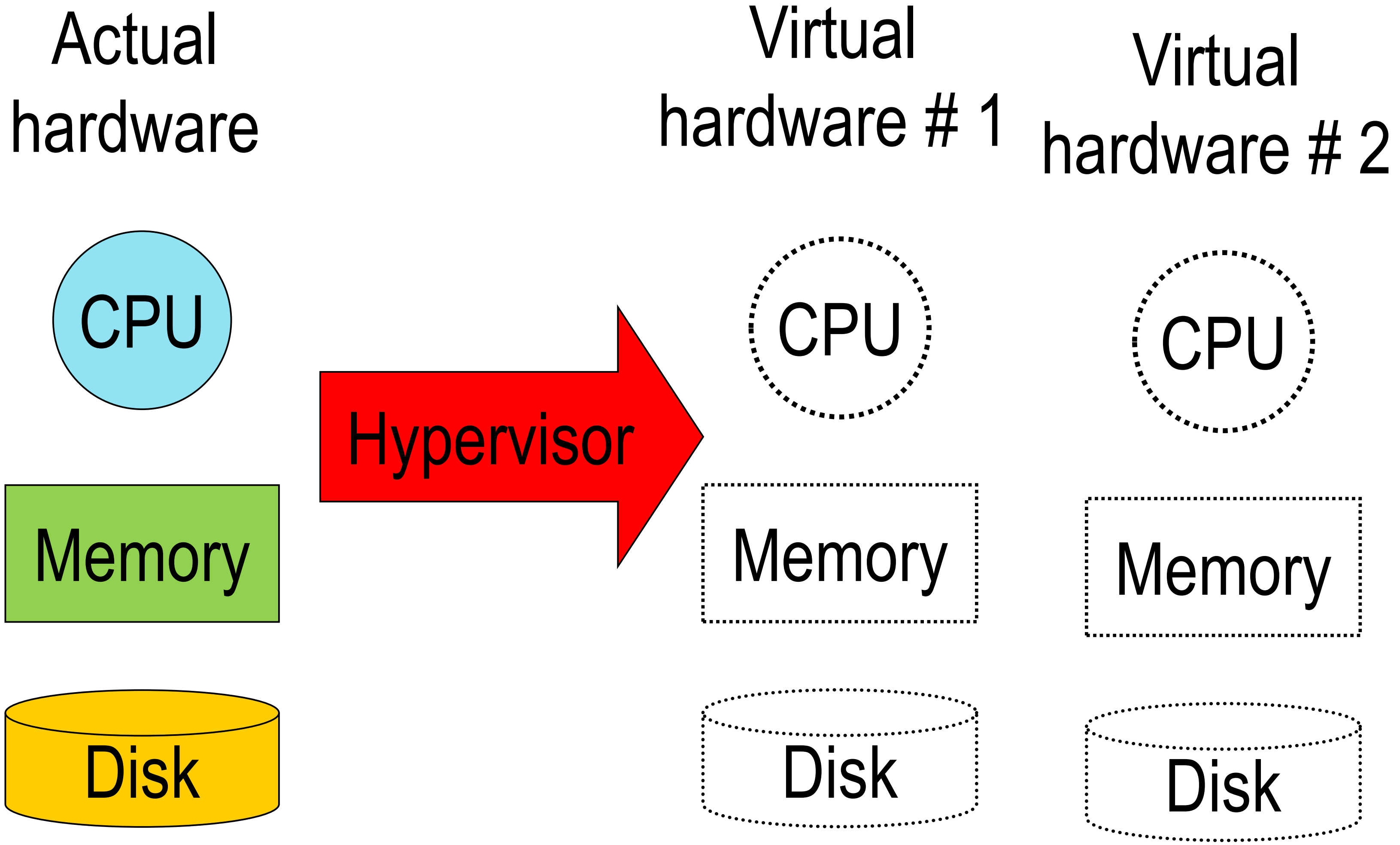
The main idea

- Let different operating systems run at the same time on a single computer
  - Windows, Linux and Mac OS
  - A real-time OS and a conventional OS
  - A production OS and a new OS being tested

How it is done

- A ***hypervisor /VM monitor*** defines two or more virtual machines
  - Each virtual machine has
    - Its own virtual CPU
    - Its own virtual physical memory
    - Its own virtual disk(s)
- Can also install VM on top of a ***host OS***
  - ***VMware, Virtual Box, Parallels, QEMU***

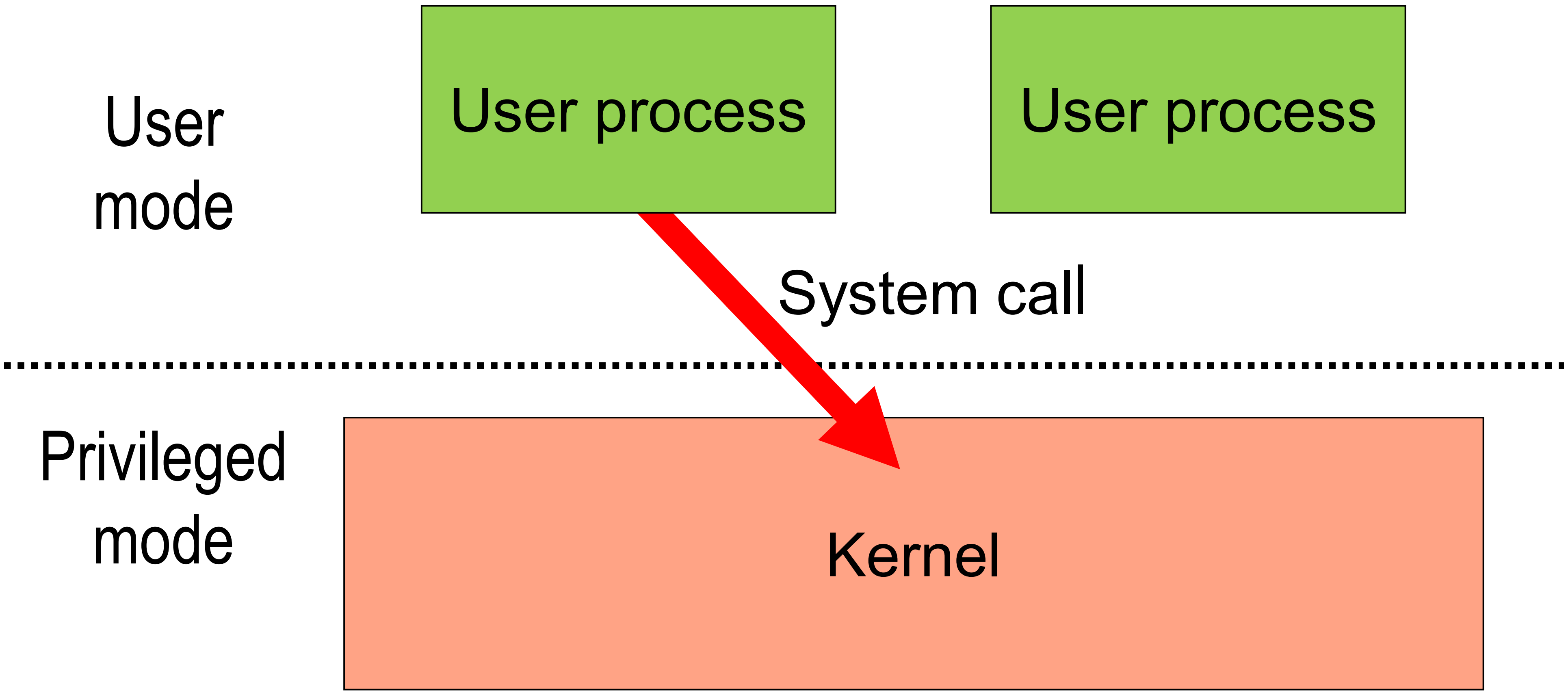
The virtualization process



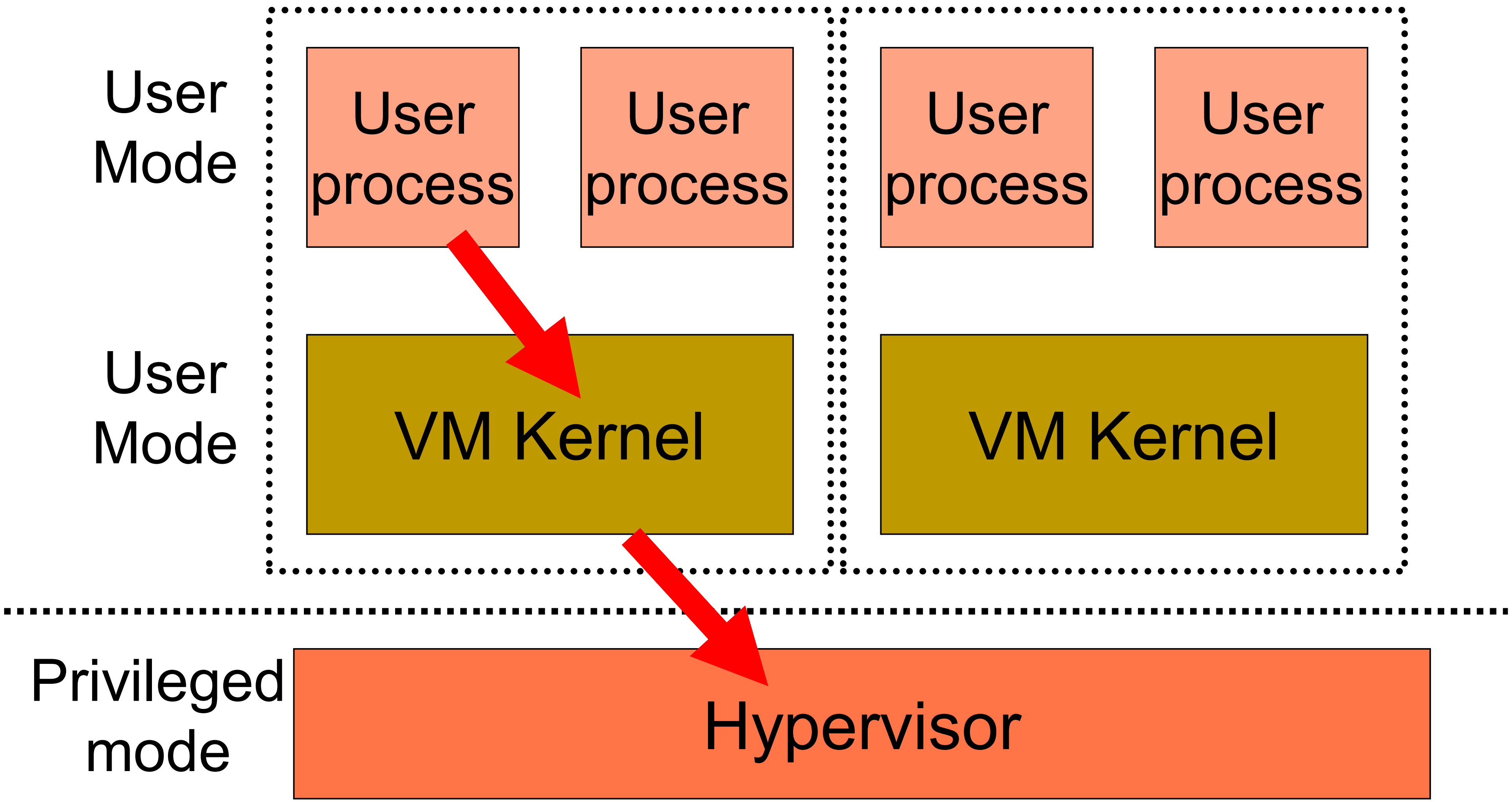
Reminder

- In a conventional OS,
  - Kernel executes in ***privileged/supervisor mode***
    - Can do virtually everything
  - User processes execute in ***user mode***
    - Cannot modify their page tables
    - Cannot execute privileged instructions

A conventional architecture



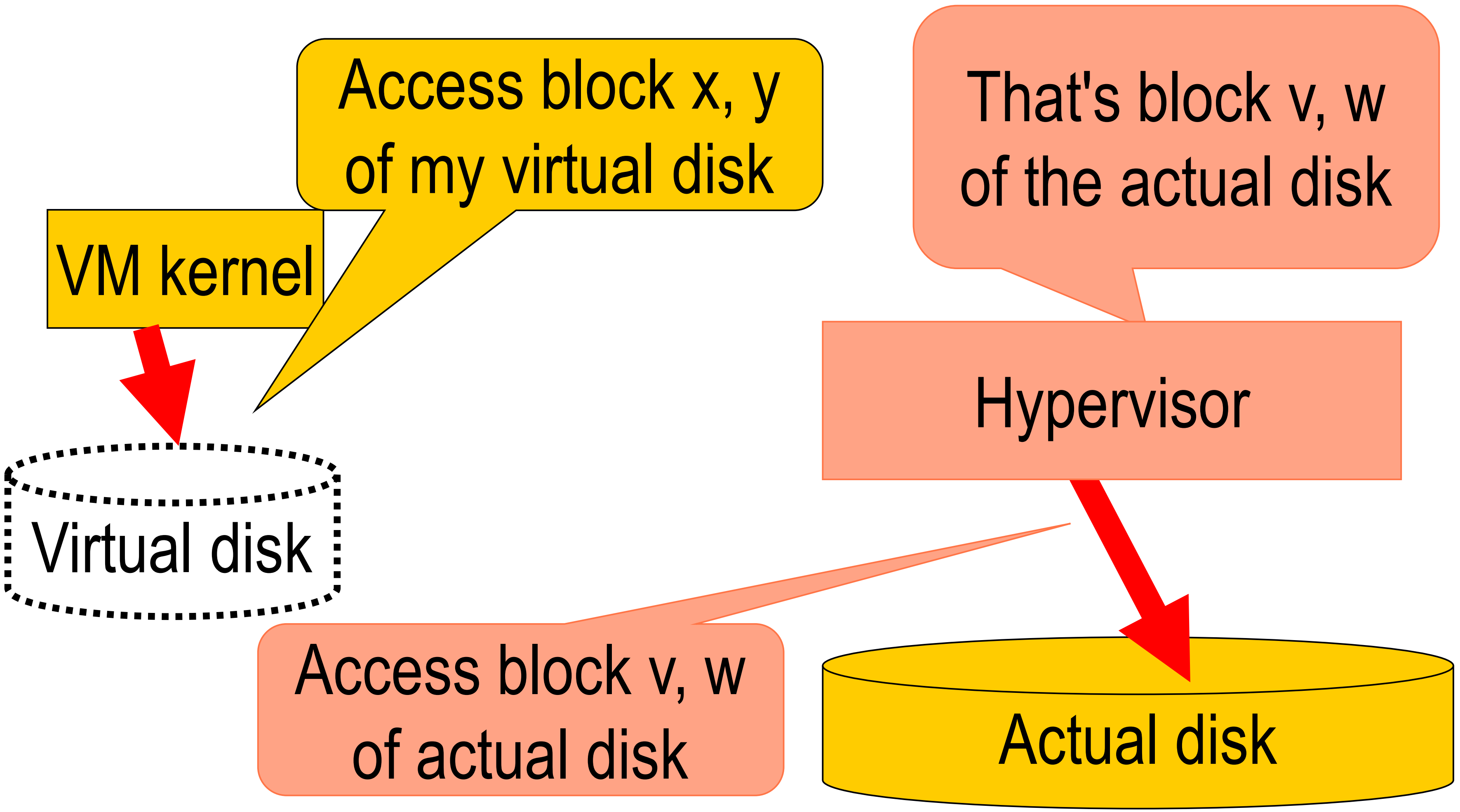
Two virtual machines



Explanations (II)

- Whenever the kernel of a VM issues a privileged instruction, an interrupt occurs
  - The hypervisor takes control and do the physical equivalent of what the VM attempted to do:
    - Must convert virtual RAM addresses into physical RAM addresses
    - Must convert virtual disk block addresses into physical block addresses

Translating a block address



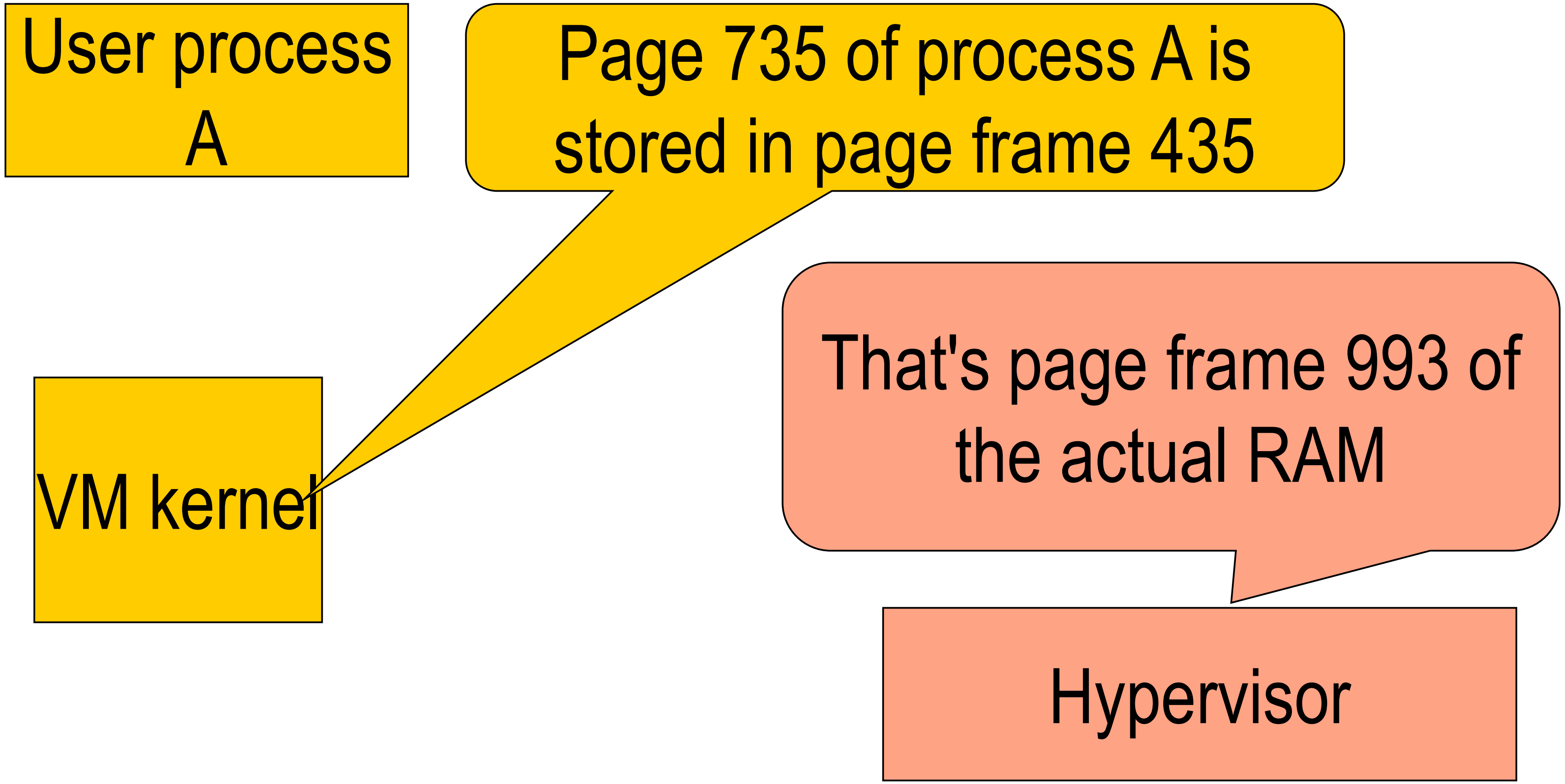
Handling I/Os

- Difficult task because
  - Wide variety of devices
  - Some devices may be shared among several VMs
    - Printers
    - Shared disk partition
      - *Want to let Linux and Windows access the same files*

Virtual Memory Issues

- Each VM kernel manages its own memory
  - Its page tables map program virtual addresses into ***what it believes to be physical addresses***

The dilemma


User process A
Page 735 of process A is stored in page frame 435
VM kernel
That's page frame 993 of the actual RAM
Hypervisor

Nastiest Issue

- The whole VM approach assumes that a kernel executing in user mode will behave exactly like a kernel executing in privileged mode except that privileged instructions will be trapped
- ***Not true for all architectures!***
  - ***Intel x86 Pop flags (POPF) instruction***
  - …

The Virtual Box Solution

- VMware pioneered the approach
- Code Scanning and Analysis Manager (CSAM)
  - Scans privileged code recursively before its first execution to identify problematic instructions
  - Calls the Patch Manager (PATM) to perform *in-situ* patching

The Xen solution

- Modify the guest kernel to eliminate badly behaving instructions such as POPF
  - ***Paravirtualization***
  - Faster but less flexible
    - Requires open-source kernel

User programs are not affected
- **Only the kernel**

Containers

- Each VM runs its own copy of the kernel
  - Takes memory space
- Containers provide isolated user-space instances that share the same kernel
  - Less overhead
  - Less flexibility
- Docker, LYXC