Basic Concepts

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

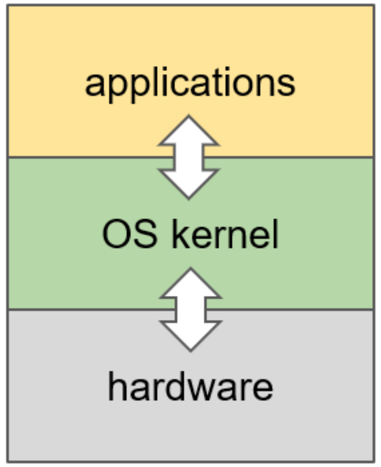
Topics

- Definition/History
- Hardware review
  - Processor
  - Memory & Disks, caching
  - Devices & I/O
  - Buses
- Bootstrapping
  - Traps
  - Kernel mode v.s. user mode
- Virtual machines
- Docker
- Interrupts
  - Interrupts vs. traps
  - DMA
- OS structure
  - Monolithic systems, Microkernel
  - Modular kernels and layered approach

OS?

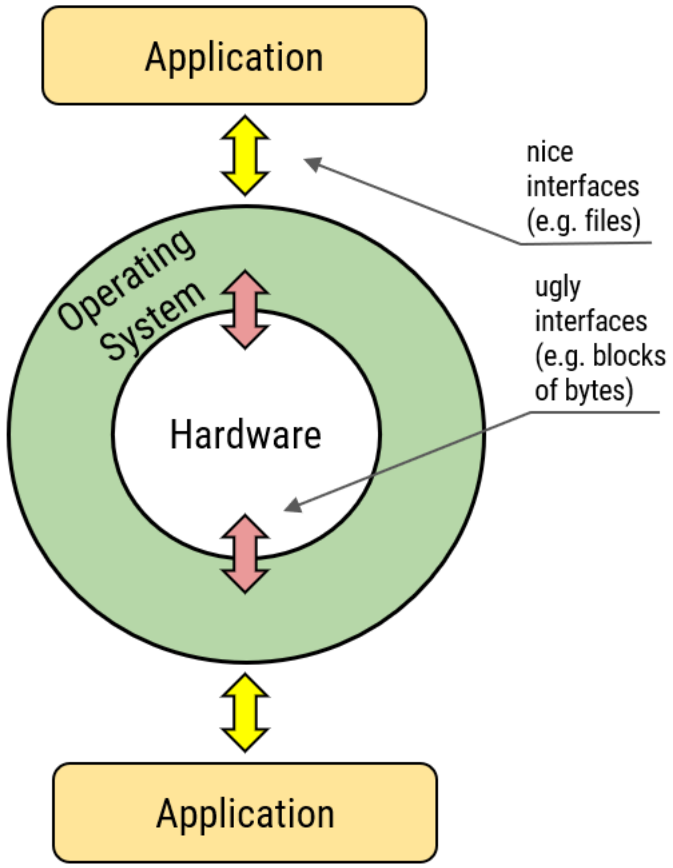
What is an Operating System

- OS is a layer of software that provides application programs with a nicer, simpler, cleaner, model of the computer (hardware)
- it manages all resources
- the central part of OS is the **kernel**
  - **kernel** runs all the time, in unrestricted **CPU mode**
  - all applications must interact with kernel to talk to hardware
  - applications run in restricted **CPU mode**



OS – as an extended machine

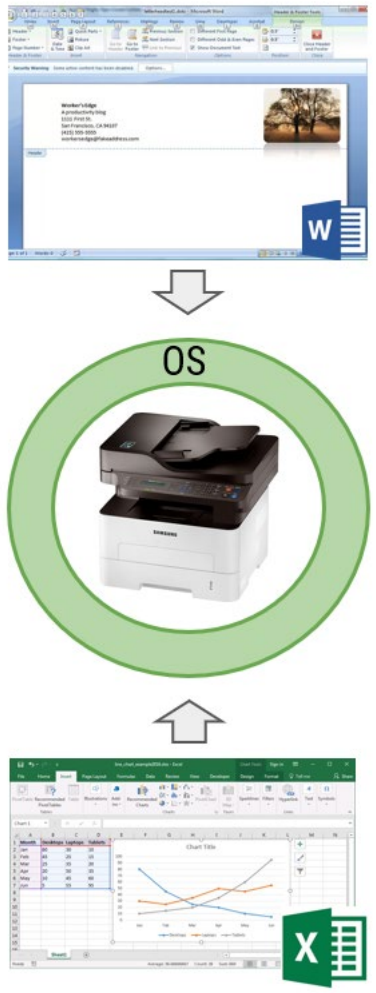
- abstraction/generalization is key to managing complexity
- first we define and implement the abstractions
  - e.g. file is an abstraction of disk storage
- working with files is easier than dealing with raw disk
- we can use these abstractions to write applications and solve problems, e.g. file editor, image viewer
- the abstractions allow us to mask the ugly hardware and provide nice interfaces instead
- many OS concepts are abstractions
  - some similarity to OO programming
  - e.g. interface = filesystem API
    implementation = USB stick


Application
nice interfaces (e.g. files)
Operating System
ugly interfaces (e.g. blocks of bytes)
Hardware
Application

OS – as a resource manager

- **resource allocator**
  - eg. 2 programs trying to print to the same printer (spooling)
  - eg. 2 programs trying to run at the same time (scheduling)
  - eg. 2 programs, each allocating memory
  - manages conflicts among multiple programs or users

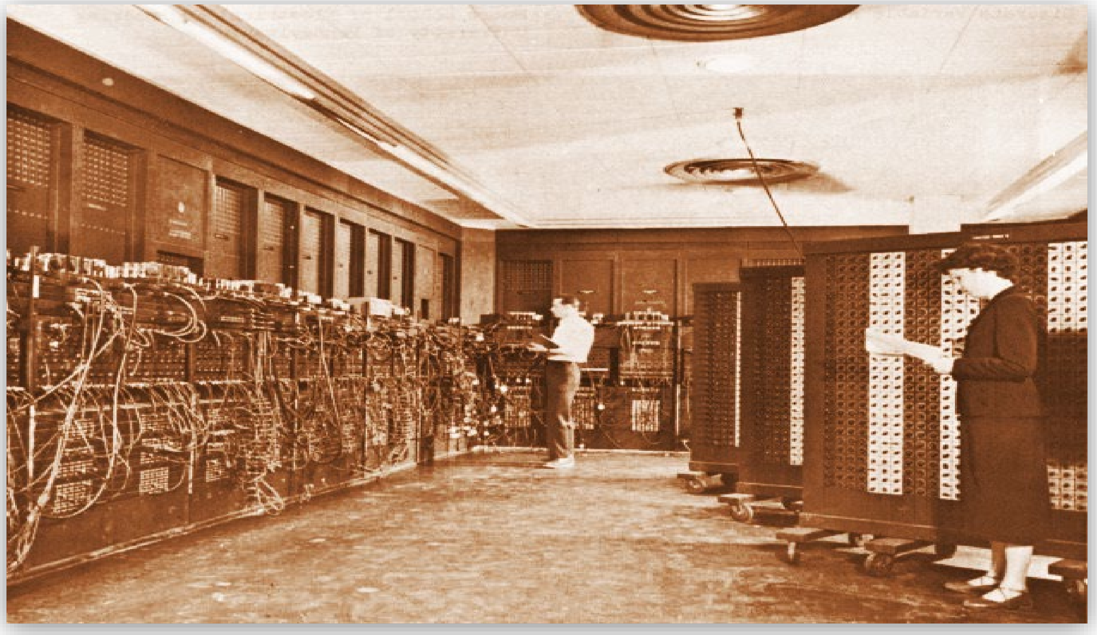
- **control program**
  - controls execution of programs
  - prevents errors and improper use



History

History of operating systems

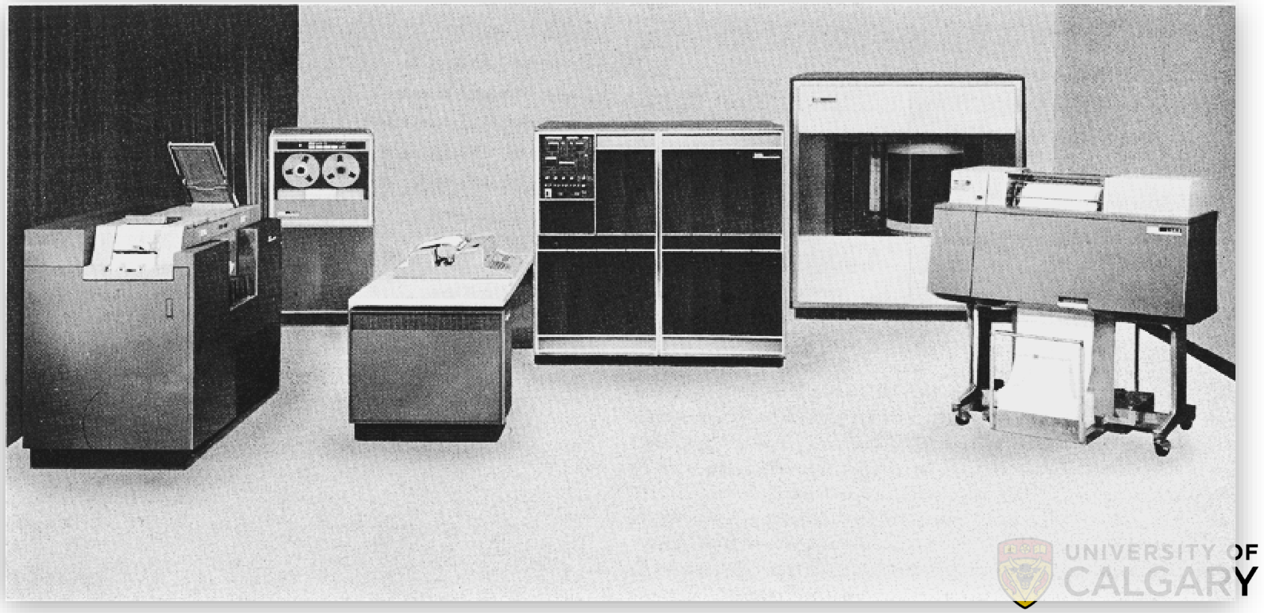
- 1st generation (1945 - 1955): Vacuum Tubes and no OS



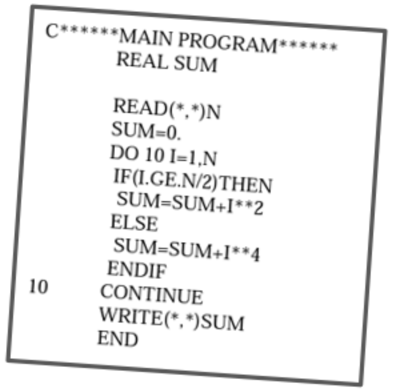
- HW with complicated wiring
- designers = builders = programmers
- programs hard-wired (wires & switches)
- machine language
- only basic numerical calculations
- one user/application at a time
- no OS (no need)

History of operating systems

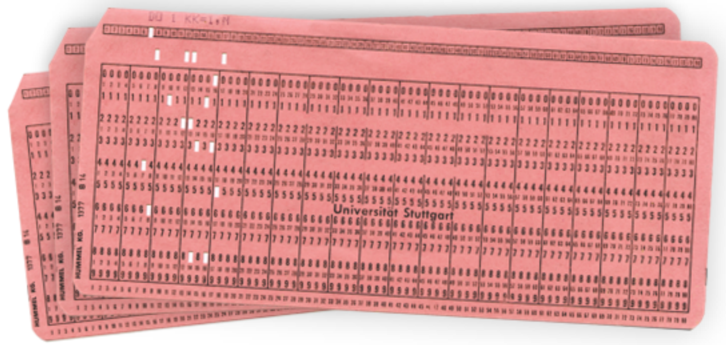
- 2nd generation (1955 - 1965): Transistors and Batch Systems
  - mainframe computers
  - very expensive memory
  - assembly, FORTRAN & COBOL + punch cards
  - OSes: FMS (Fortran Monitor System) and IBSYS (IBM's OS)
  - important concepts: **batch systems**



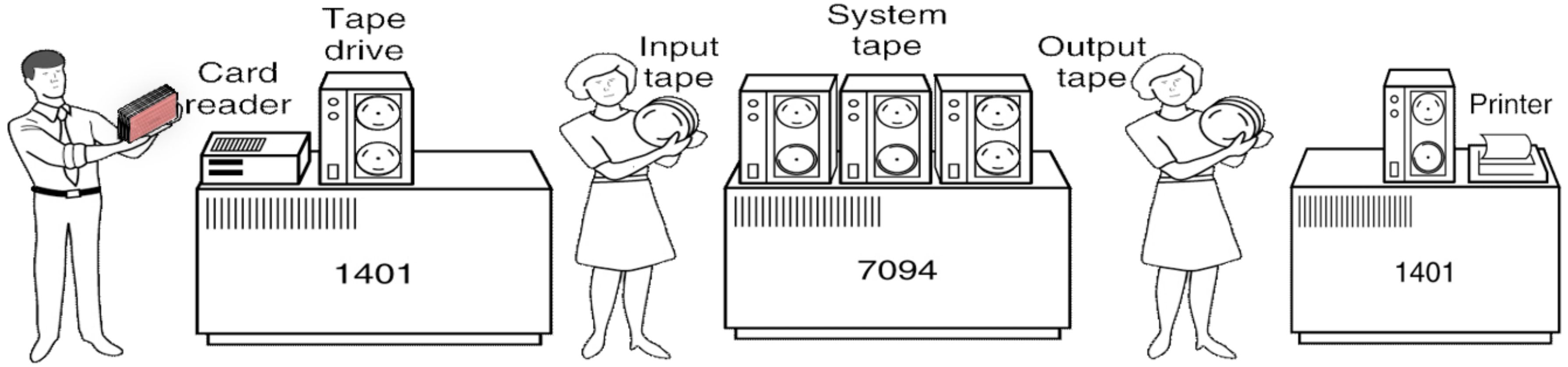
Batch systems



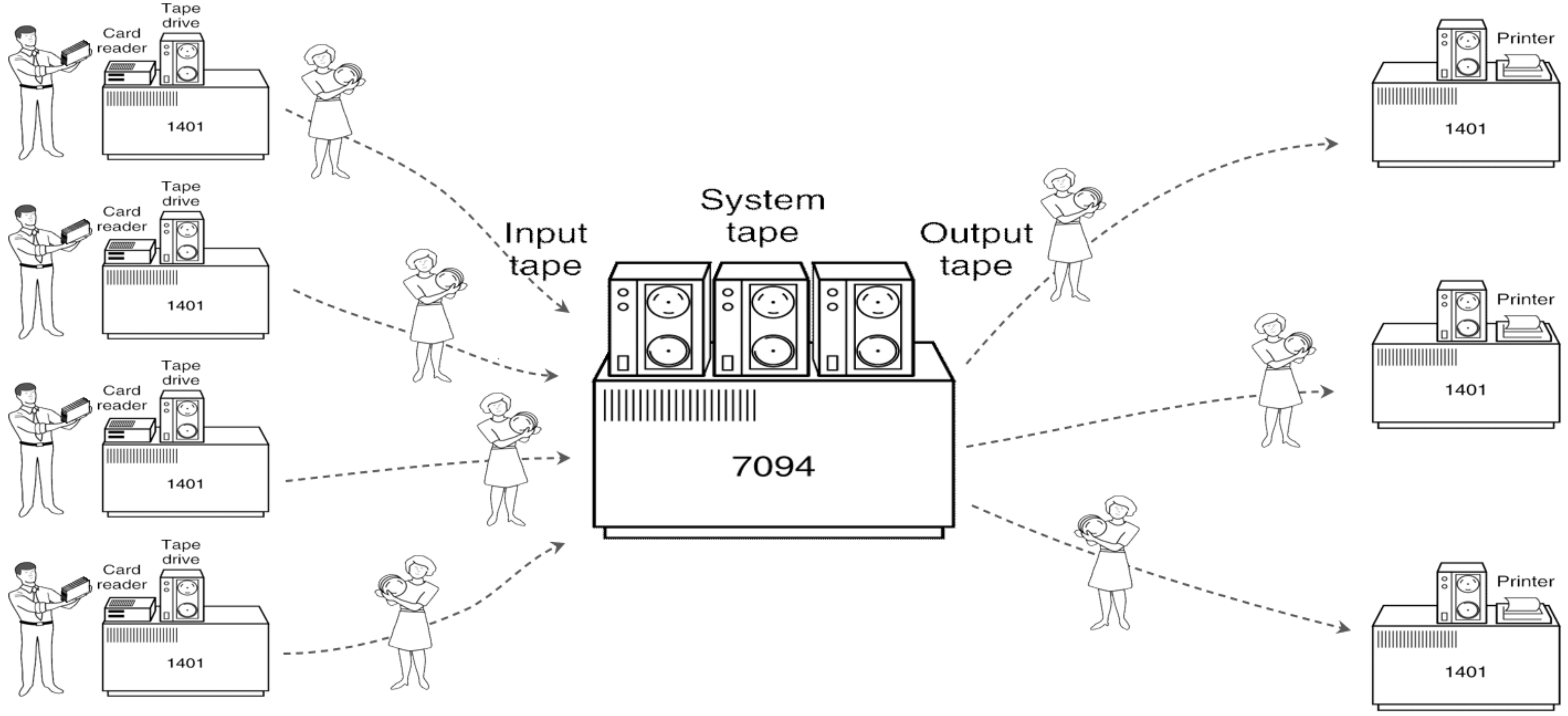
```
C******MAIN PROGRAM******
       REAL SUM

       READ(*,*)N
       SUM=0.
       DO 10 I=1,N
        IF(I.GE.N/2)THEN
         SUM=SUM+I**2
        ELSE
         SUM=SUM+I**4
        ENDIF
10     CONTINUE
       WRITE(*,*)SUM
       END
```



https://commons.wikimedia.org/wiki/File:Punch_card_Fortran_Uni_Stuttgart_%283%29.jpg



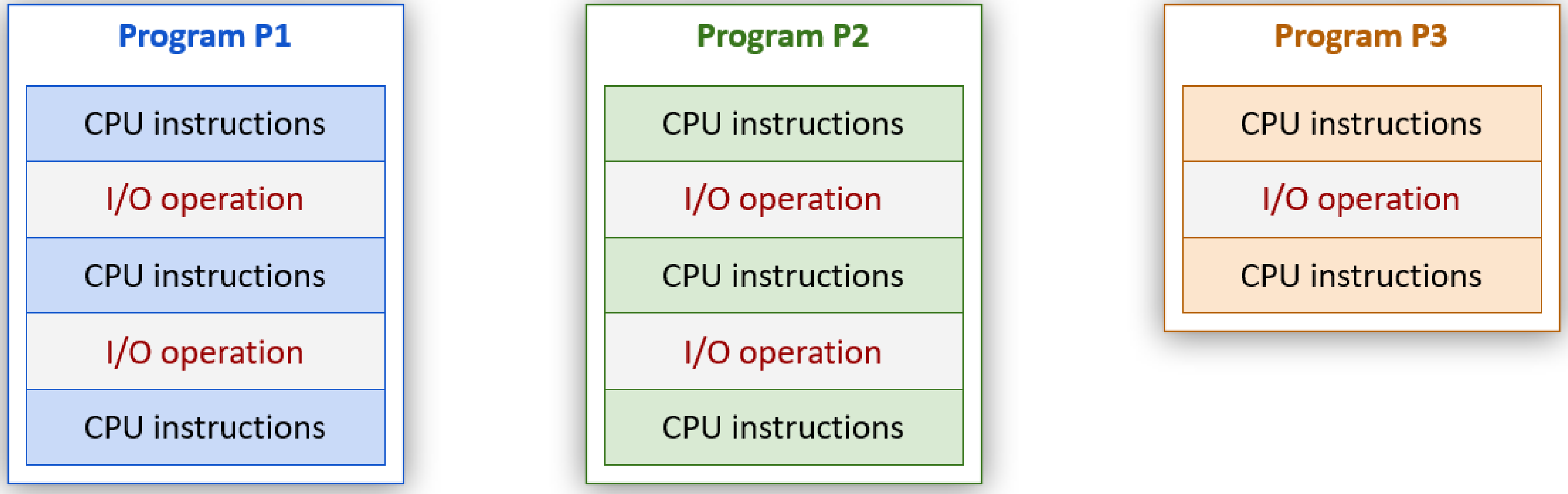
Batch systems



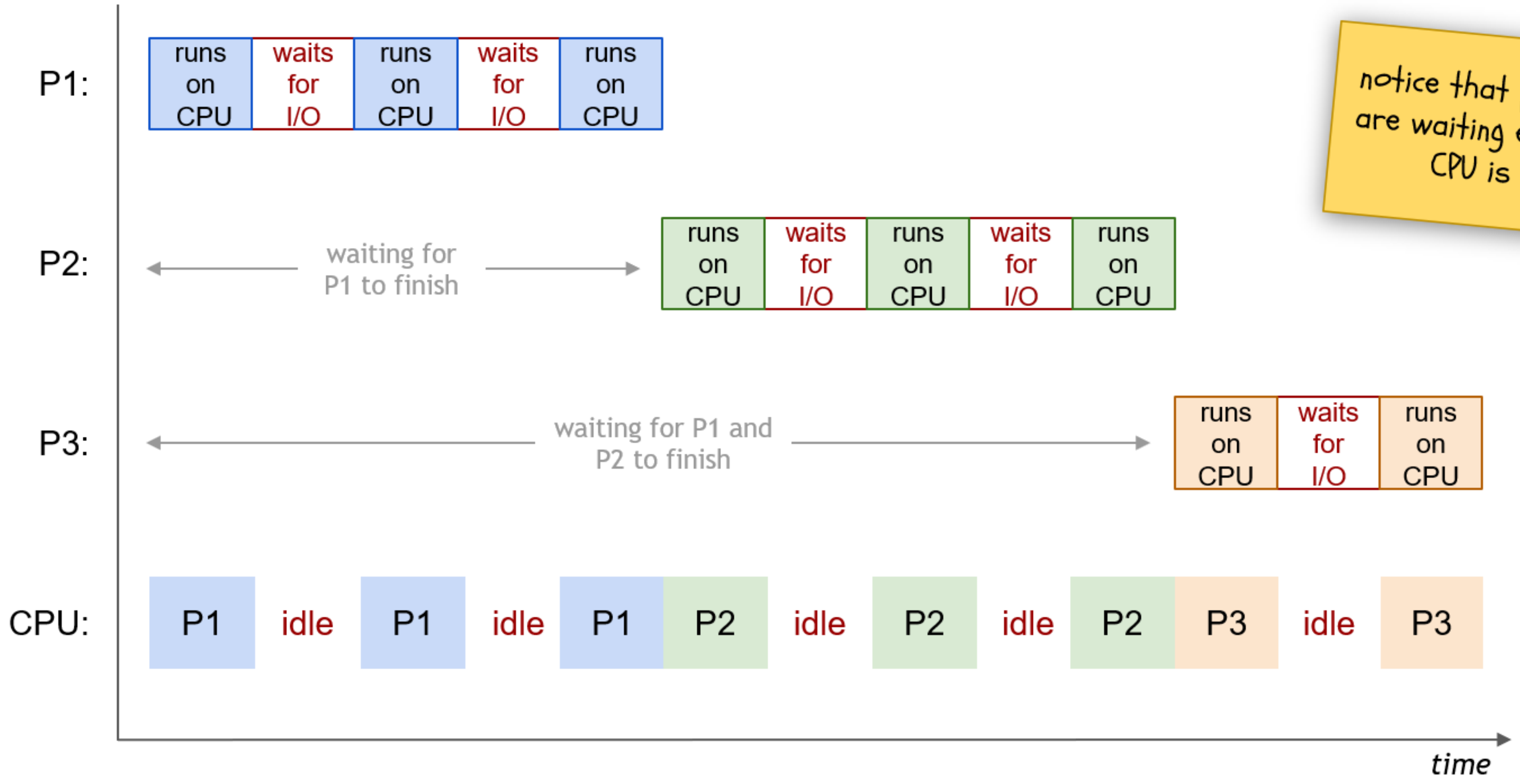
History of operating systems

- 3rd generation (1965 - 1980): ICs and Multiprogramming
  - integrated circuits & much cheaper memory
  - OSes: IBM OS/360, CTSS (by MIT), MULTICS (complicated, but influential), UNIX (inspired by MULTICS), and eventually Linux (90's)
  - several important concepts:
    1. **multiprogramming**: multiple programs loaded into memory, when one program waiting for I/O, CPU executes next program
    2. **spooling:** mechanism for dealing with slow devices
    3. **time-sharing**: multiple users using one computer simultaneously & interactively

Running multiple programs

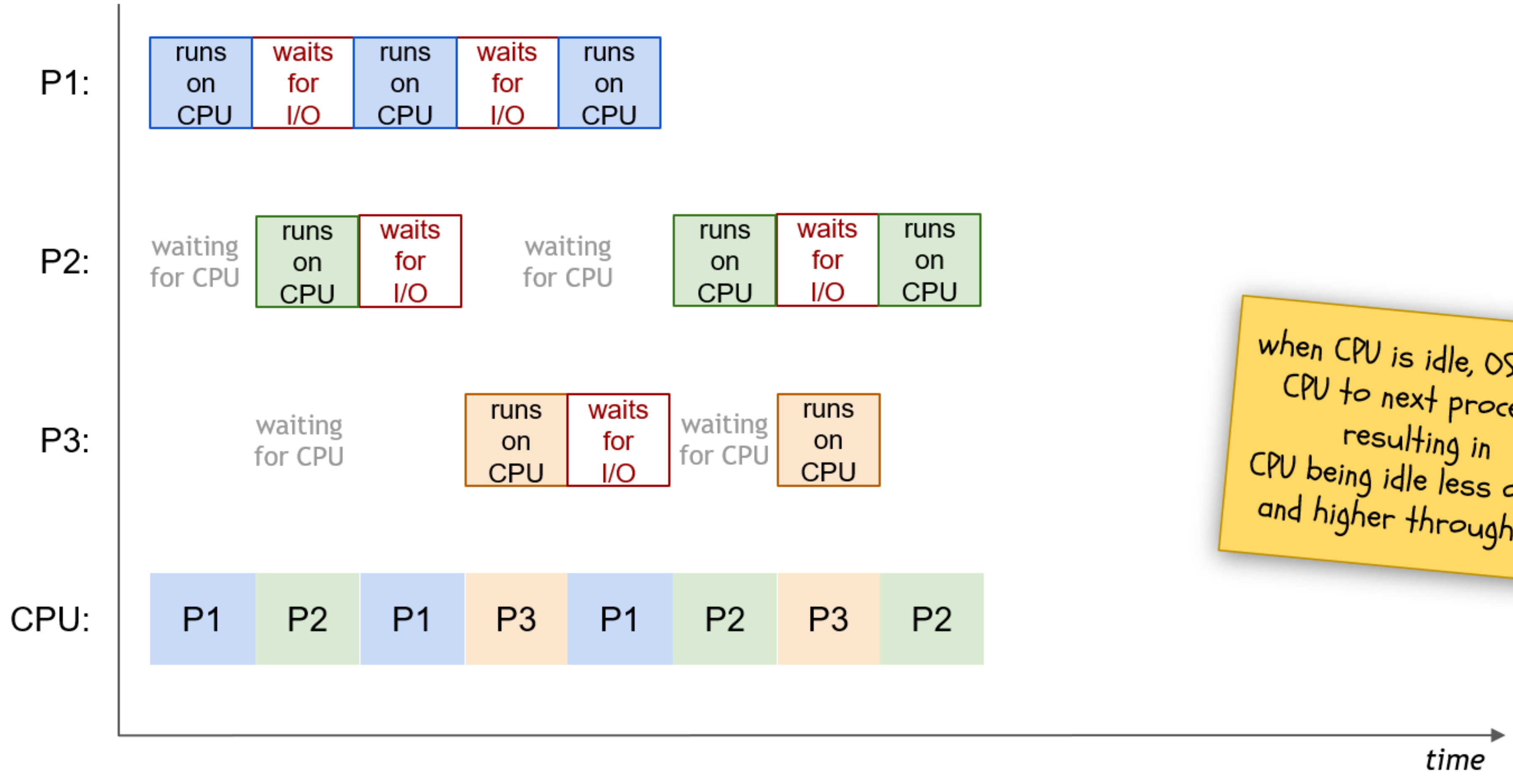


Running one program at a time



notice that programs are waiting even when CPU is idle

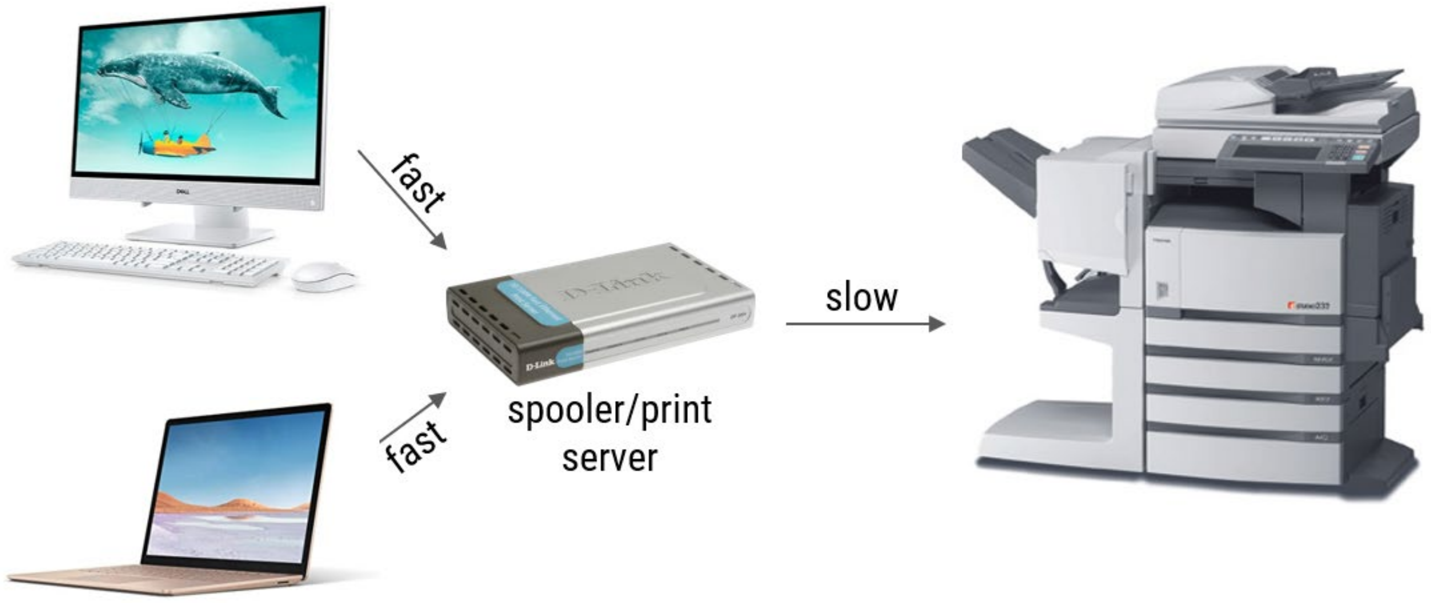
Multiprogramming



when CPU is idle, OS gives CPU to next process, resulting in CPU being idle less often, and higher throughput

Spooling

- **spooling** is typically used to deal with slow devices / peripherals, e.g. printers:



- **spooling** can be used (somewhat) to deal with **deadlocks** in concurrent programming by making non-shareable resource a shareable

History of operating systems

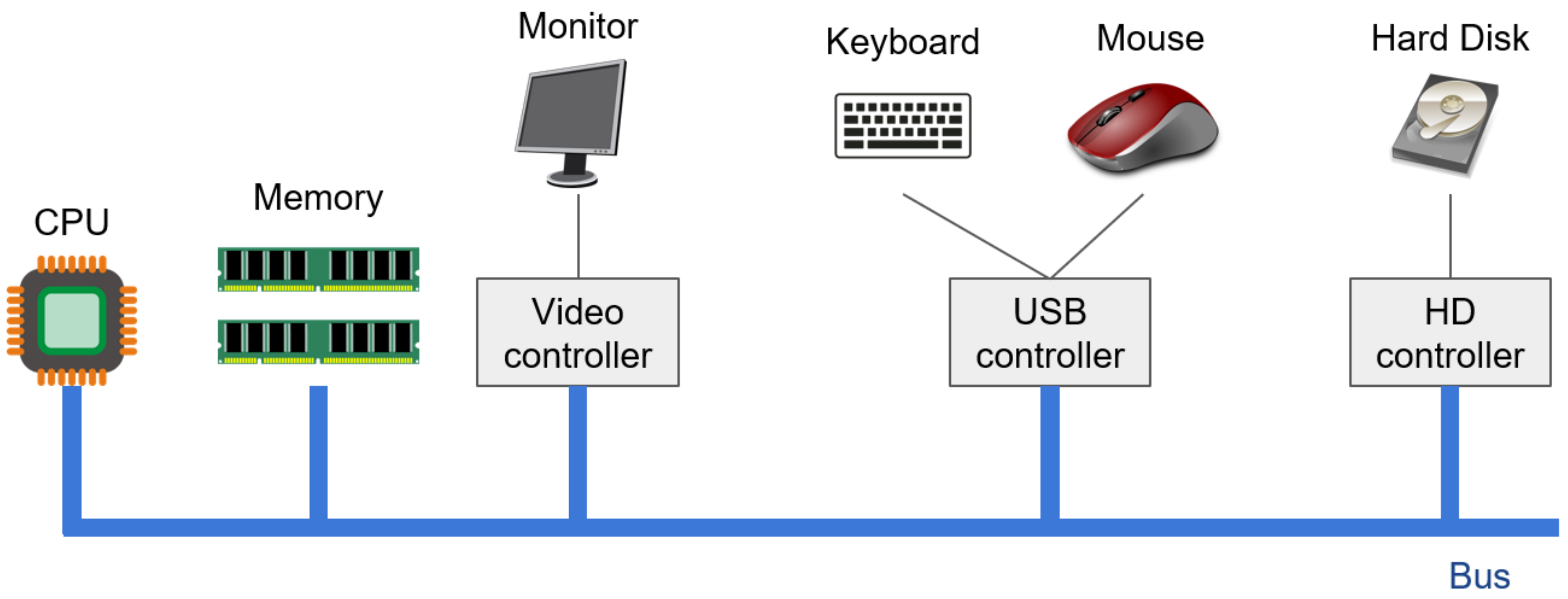
- 4th generation (1980 - present): personal computers
  - cheap mass-produced computers
  - GUI shells on top of OS
  - Windows, Mac OS, Linux + GNOME / KDE

- 5th generation (1990 - present): mobile computers

Hardware

UNIVERSITY OF CALGARY

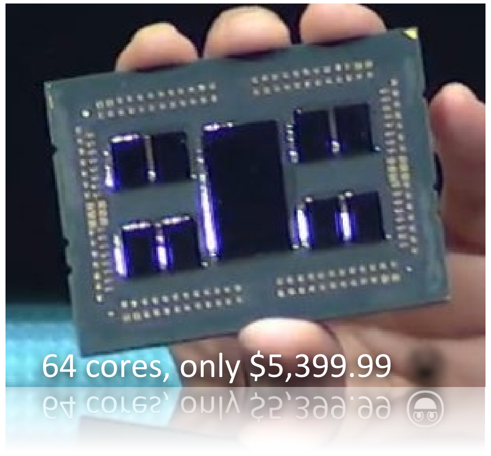
Hardware review



common components of a desktop computer

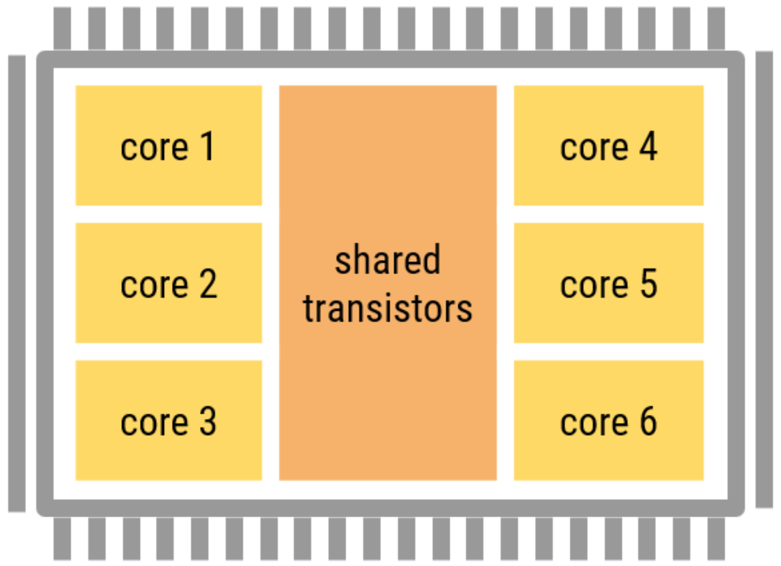
CPU

- **Central Processing Unit**
- the “brain” of the computer
- on-board **registers** for faster computation
  - instead of accessing memory for every instruction
  - accessing information in registers is much faster than memory
  - general purpose registers:
    - data & addresses
  - special purpose registers:
    - **program counter**: contains memory address of the next instruction to be fetched
    - **stack pointer**: points to the top of the current stack in memory
    - **status register**: interrupt flag, privilege mode, zero flag, carry flag, …
  - other (floating point, vector, internal, machine specific, etc)



Multicore CPUs

- nearly all modern CPUs contain multiple cores
- a core = "mini CPU"
- each core can execute code in parallel with other cores
  - e.g. one core running YouTube,
    another core running Minecraft,
    both at full speed
- cores typically share some hardware, e.g. cache(s)



Instructions

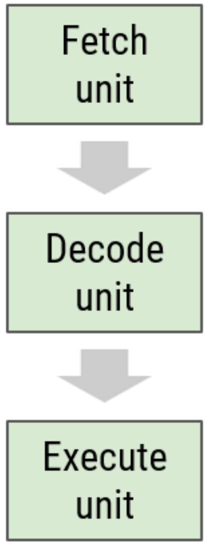
UNIVERSITY OF CALGARY

Instruction cycle

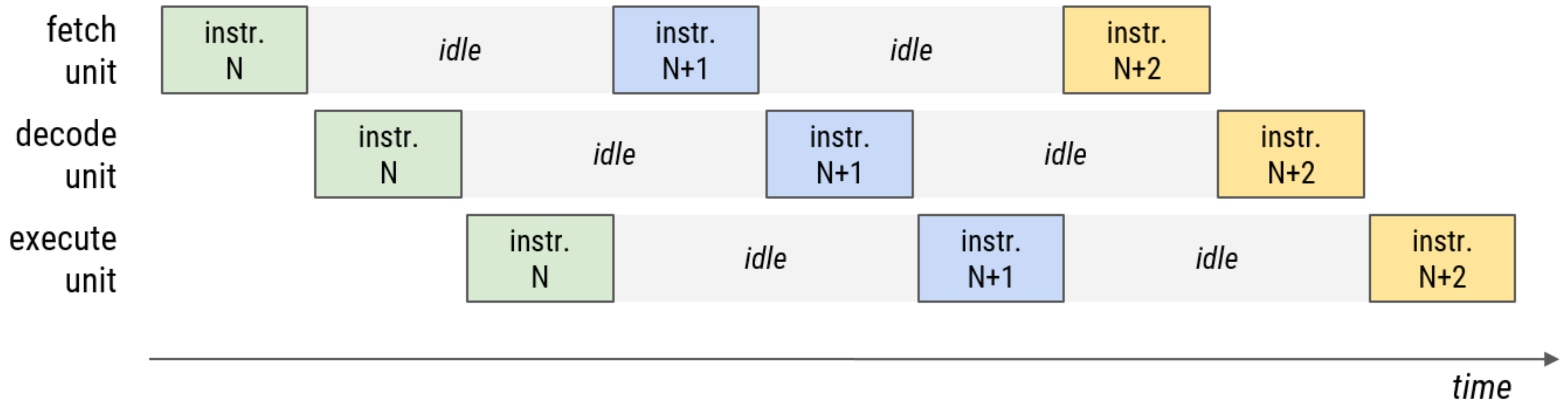
- a simple CPU cycle when stages operate sequentially:

    1. **fetch** an instruction from memory
    2. **decode** it to determine its type and operands
    3. **execute** it
    4. repeat

- fetch is usually the longest operation



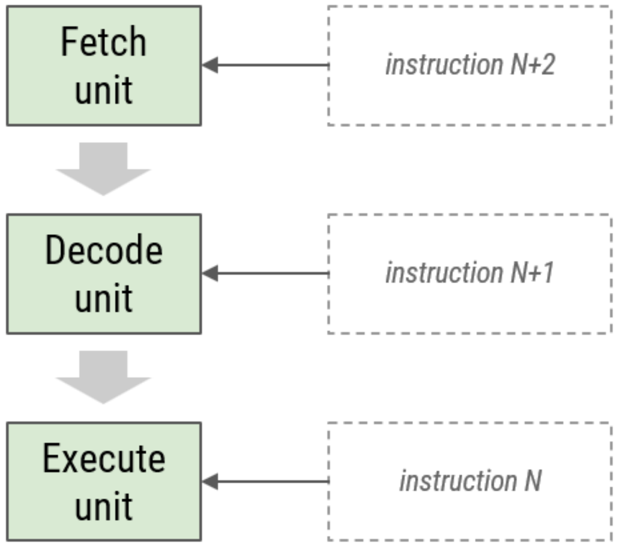
Instruction cycle - sequential



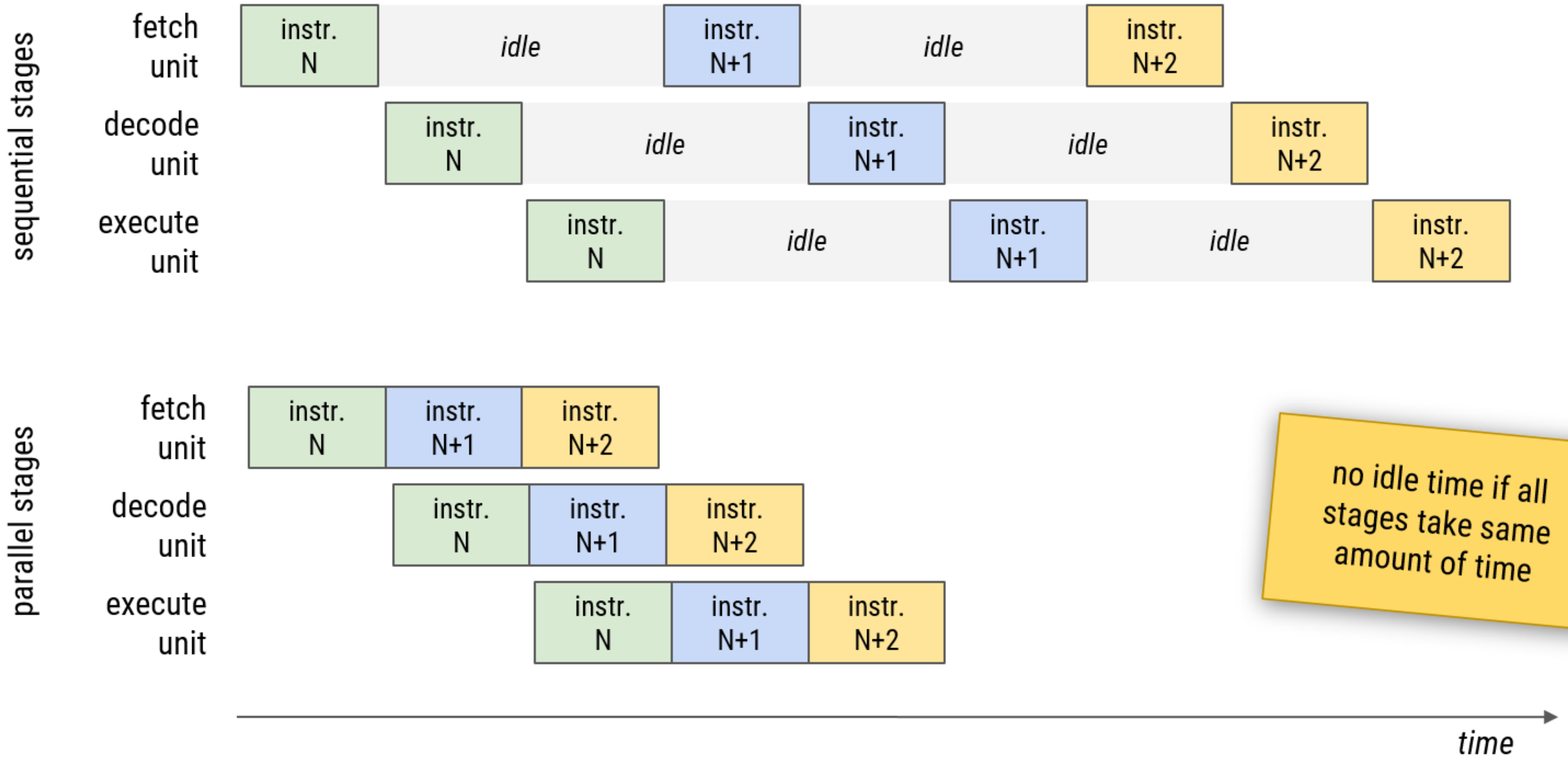
- assuming all stages take the same amount of time
- notice that the units are often idle
- we can improve the performance by letting the units operate in parallel

Instruction pipelining

- we can let the stages work in parallel
- while executing instruction N,
  the CPU could be decoding instr. N+1 and
  fetching instr. N+2

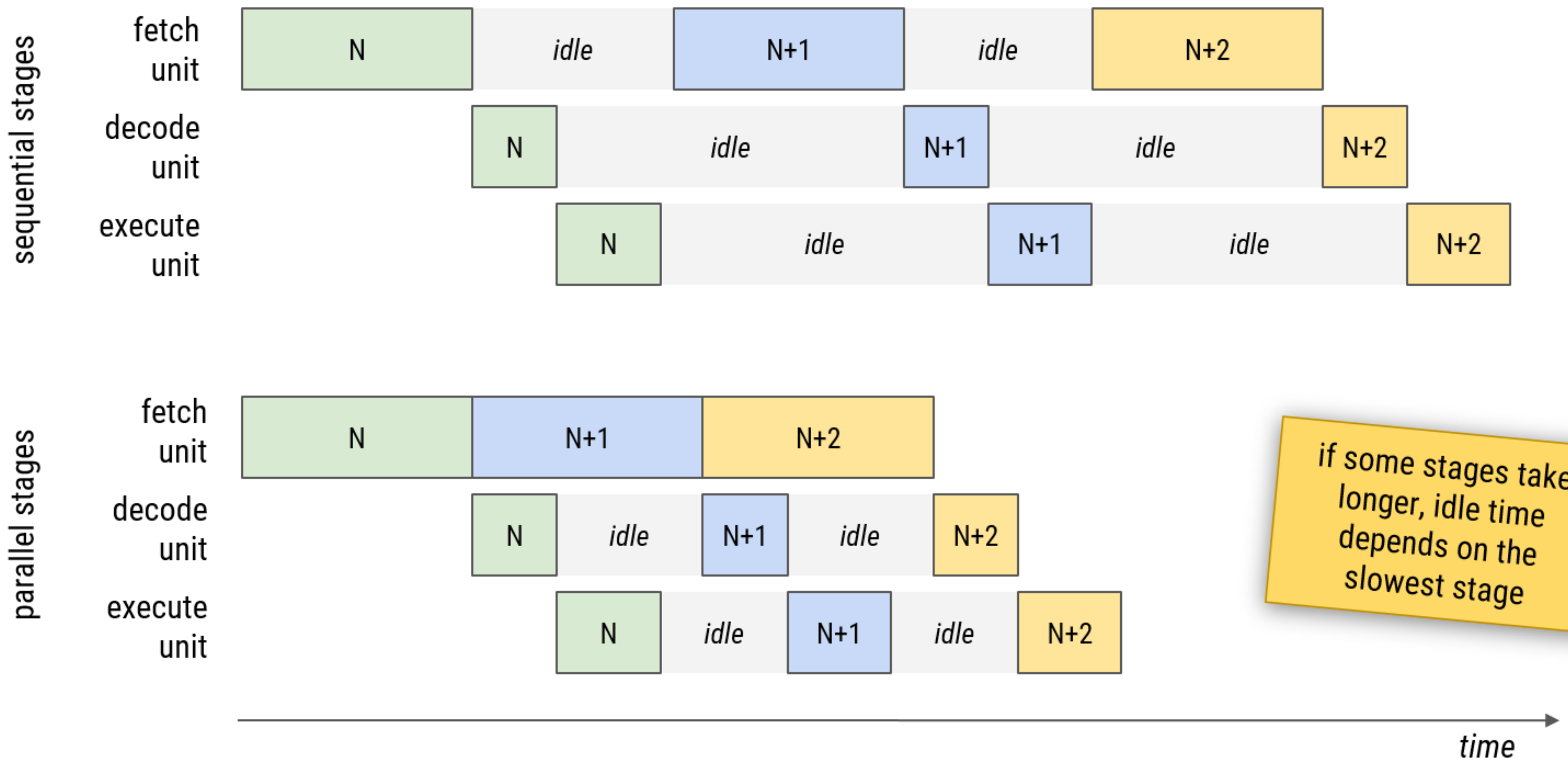


Instruction cycle – sequential vs parallel



no idle time if all stages take same amount of time

Instruction cycle – sequential vs parallel



if some stages take longer, idle time depends on the slowest stage

Instruction pipelining

- benefits of pipelining:
  - the CPU can work on more than one instruction at a time
  - this allows the CPU to mask some of the memory access time
- cons:
  - more complexity
  - have to deal with invalidated stages

```
reg1 = reg1+1
reg2 = reg1
```

Instruction pipeline – example 1

- consider a CPU with a 3 stage instruction pipeline, where each stage takes 1/1000s
- how many instructions per second can this CPU execute?

- if the stages are executed sequentially:
  every instruction takes 0.003s, so on **average** the CPU executes ~333 instructions/s

- if the stages are executed in parallel:
  1st instruction will take 0.003s to finish executing…
  2nd instruction will be done 0.001s after first
  3rd instruction will be done 0.001s after 2nd
  etc.
  so over **long run** the CPU will execute 1000 instructions/s

Instruction pipeline – example 2

- consider a CPU with a 3 stage instruction pipeline,
  fetch takes 10ns, decode takes 3ns and execute takes 2ns

- if the stages are executed sequentially:
  each instruction takes 10ns+3ns+2ns = 15ns
  on **average** the CPU executes 1instr./15ns ⟶ ~66,666,667 instructions/s

- if the stages are executed in parallel:

  first instruction will take 10ns+3ns+2ns = 15ns to execute
  but **over long** run the CPU will execute 1 instr./**10ns** = 100,000,000 instructions/s

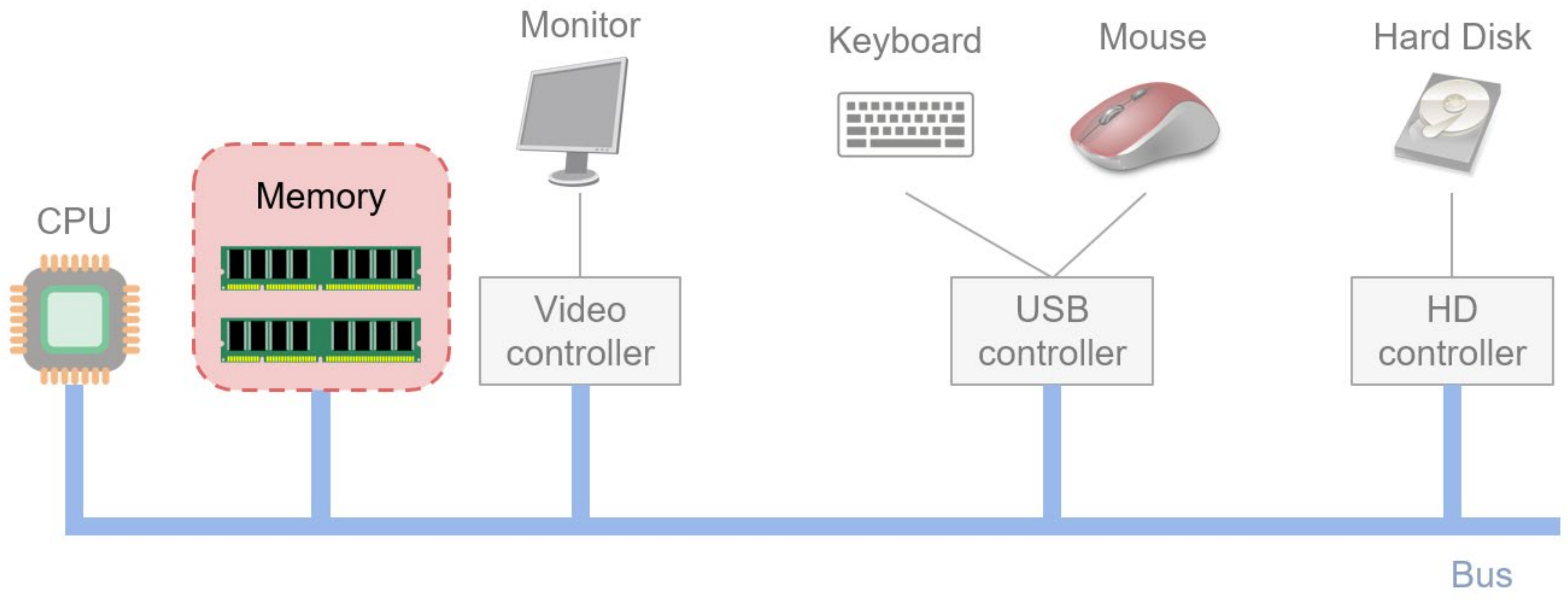
  in other words, the pipeline is as slow as its slowest stage

Memory

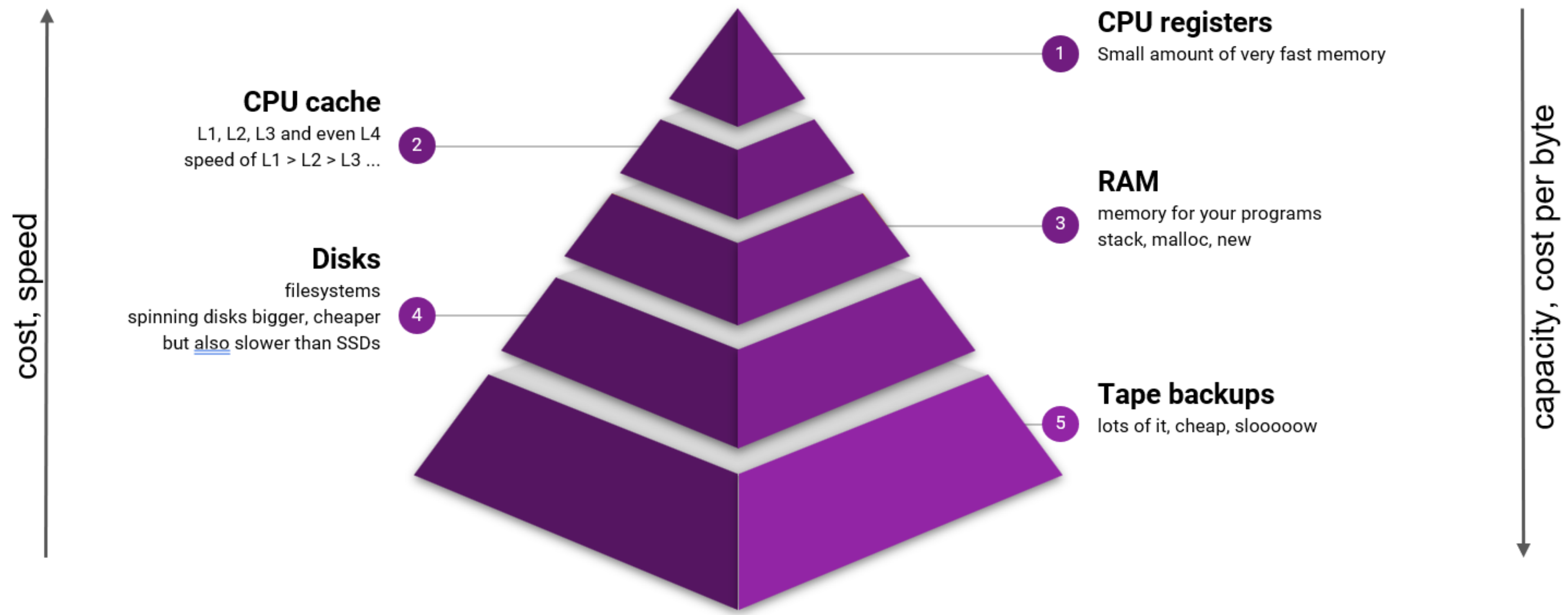
UNIVERSITY OF CALGARY

Memory

- ideally, memory should be **fast**, **large** and **cheap**
- in practice, **we can get 2 of the 3, but not all three**



Typical memory hierarchy



CPU registers
Small amount of very fast memory

CPU cache
L1, L2, L3 and even L4
speed of L1 > L2 > L3 ...

RAM
memory for your programs
stack, malloc, new

Disks
filesystems
spinning disks bigger, cheaper
but also slower than SSDs

Tape backups
lots of it, cheap, slooooow

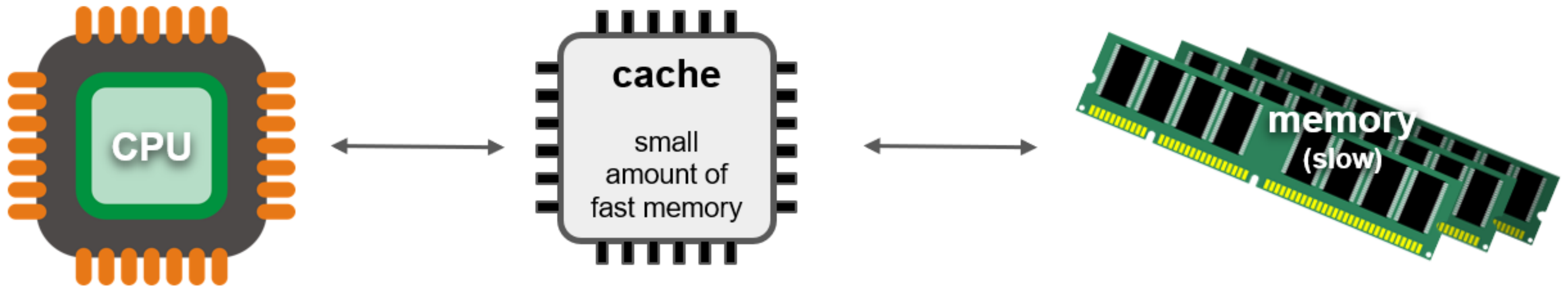
Memory

- main memory: **Random-Access Memory (RAM)**
- consists of an array of words, and each word has its own address (memory address)
- typical memory operations:
  - `load <address>,<register>` – load a word from memory into CPU register
  - `store <register>,<address>` – stores contents of register in memory
- both are slow operations compared to the speed of the CPU

Caching

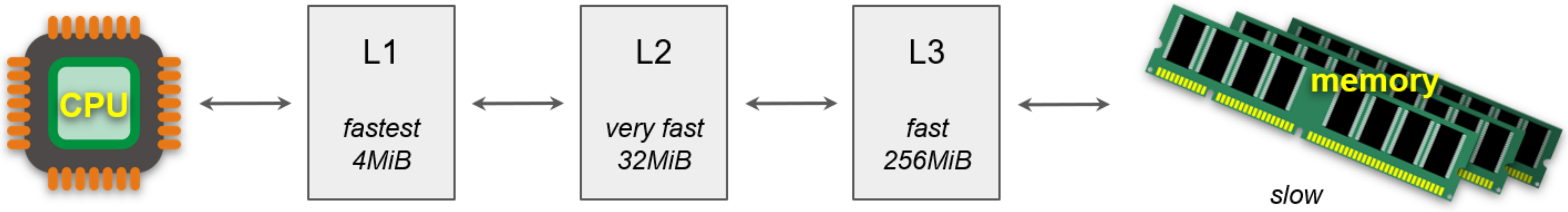
Caching

- CPU caching
  - most heavily used data from memory is kept in a high-speed cache located inside or very close to the CPU
  - when CPU needs to get data from memory, it first checks the cache
  - **cache hit**: the data needed by the CPU is in the cache
    cache miss: CPU needs to fetch the data from main memory

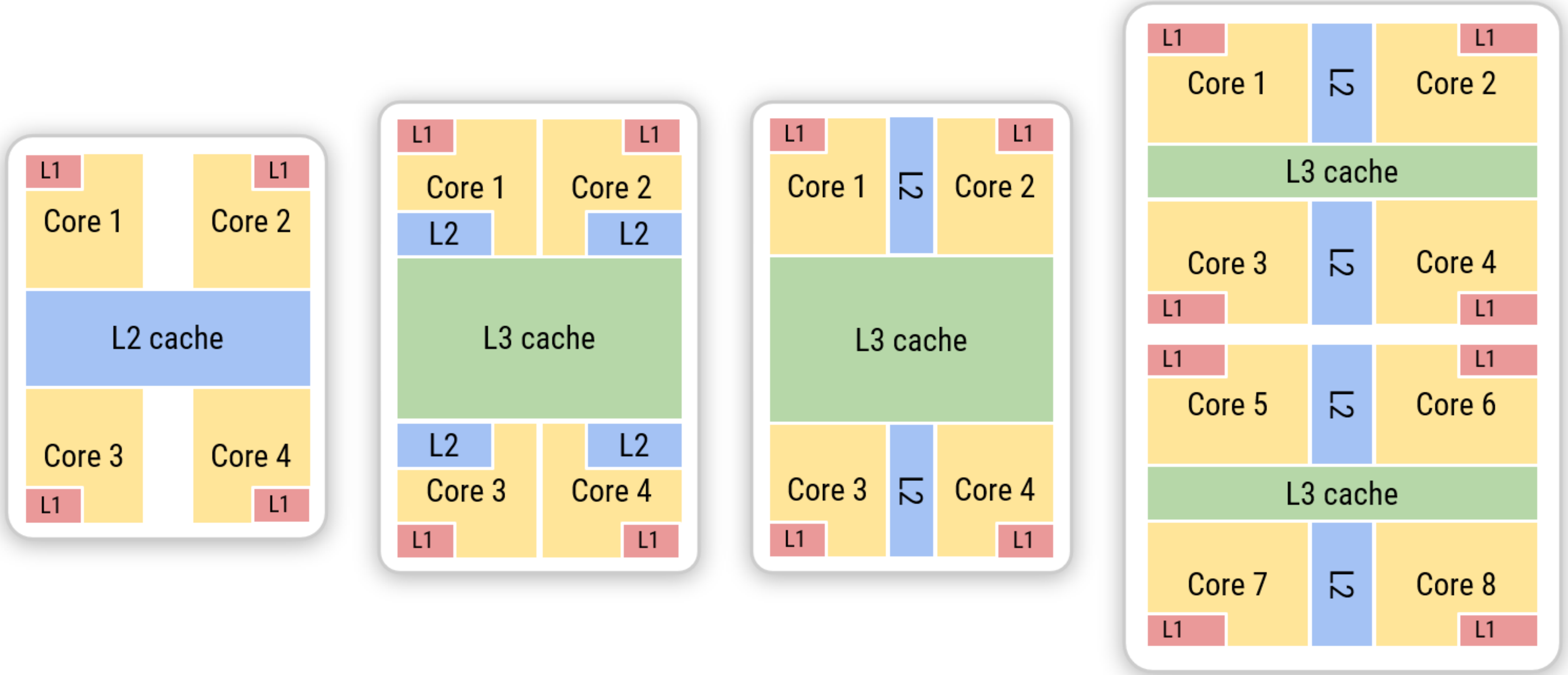


Cache hierarchy (multilevel caches)

- L1 cache (~32KiB): fastest, feeds decoded instructions into CPU execution engine, private per core
- L2 cache (½ MiB): stores recently used memory, slower than L1, may be shared by multiple cores
- L3 (x MiB): faster than memory, slower than L2, usually shared by all cores, or a group of cores
- L1, L2 and L3 are usually on the same chip as the CPU
- some CPUs have L4 cache ...

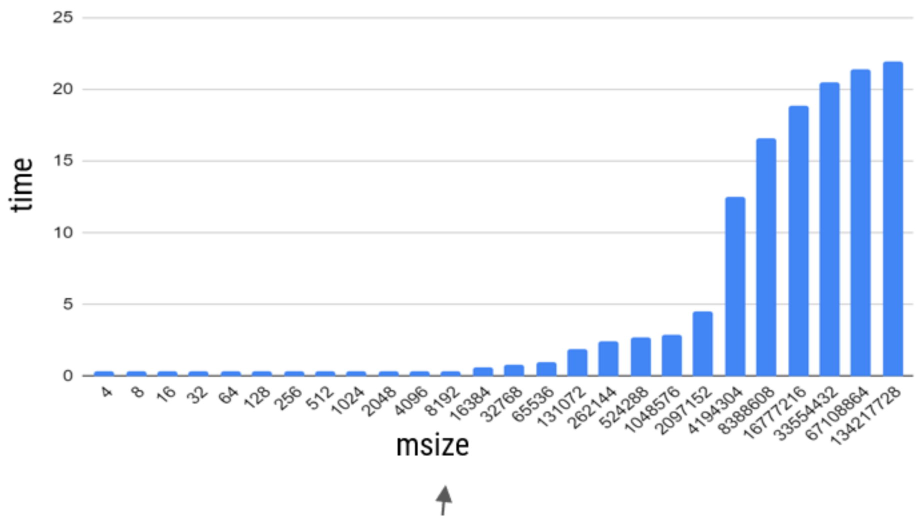


Shared vs separate caches on multicore CPUs



CPU cache and C++

```
auto start_time = clk::now(); // start timer
int sum = 0;
for( int i = 0 ; i < N ; i ++) {
  sum += arr[arr[i % msize]];
}
auto end_time = clk::now();    // end timer
double dt = duration_cast<duration<double>>(
            end_time - start_time).count();
std::cout << dt << "\n";
```



- N is a constant, but we can change msize
- by changing the variable msize, the loop runs fast or slow
- this is weird, since the loop contains the same number of instructions, no matter what msize is
- https://replit.com/@jonathanwhudson/cpu-cache-test#main.cpp (copy & run on linuxlab)

Caching

- the goal of caching is to increase performance of slower memory/storage by adding a small amount of faster memory/storage, called cache
- cache can improve **read performance**:
  - keep copy of information obtained from slow storage in cache
  - next time we need the information, check the cache first
- cache can also improve **write performance**:
  - write info to fast storage, and eventually (delayed) write to slow storage
  - delay reduces number of writes if data is overwritten multiple times
  - can replace multiple small writes with fewer big writes (buffering)
- caching is a useful concept in general
- many uses: disk cache, DNS, database

Caching

- cache storage is fast but expensive, so it's usually much smaller than the slow storage
- some general caching issues:
  - when and where to put a new item into the cache
  - which item to remove from the cache when cache is full
  - what to do with evicted item
  - multiple cache synchronization
  - how long does the cached data stay valid (expiration)
- answers depend on the application

Memoization

- similar concept to caching
- optimization technique used to speed up programs,
  by remembering (storing) results of expensive computations
- general idea:

```
slow_computation(p):
  … /* some slow computation */
  return result
```

$\Rightarrow$

```
cache = global variable, initially empty
fast_computation(p):
  if p not in cache:
    cache[p]= slow_computation(p)
  return cache[p]
```

- works best if the set of possible parameters is small

Memoization

- easy to use even on recursive functions

$O(2^n)$

```
def fib_slow(n):
  if n < 2:
    res = n
  else:
    res = fib_slow(n-1)
          + fib_slow(n-2)
  return res
```

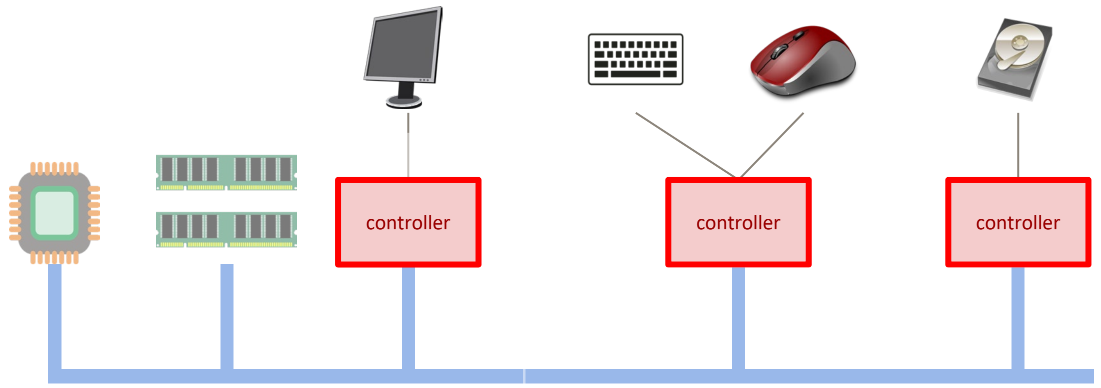
$O(n)$

```
cache = {}

def fib_fast(n):
  if n not in cache.keys():
    if n < 2:
      cache[n] = n
    else:
      cache[n] = fib_fast(n-1)
                 + fib_fast(n-2)
  return cache[n]
```

Devices

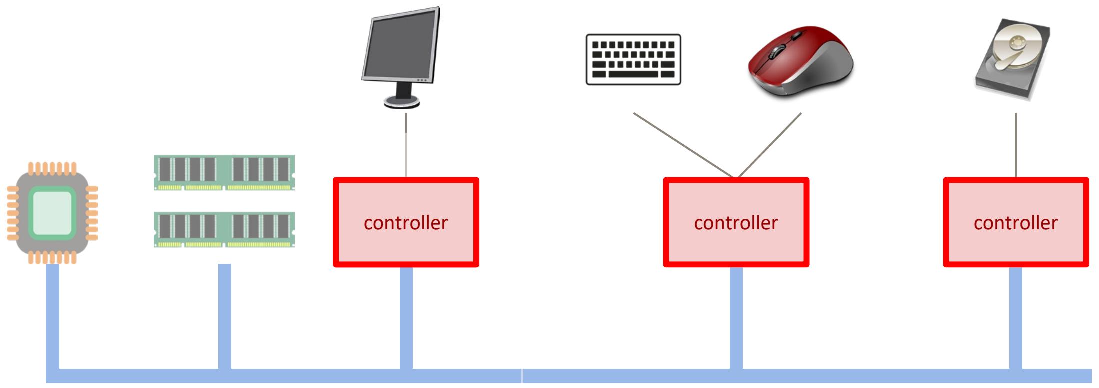
Devices

- I/O devices usually connected to computer via device controller
- **device controller**
  - a chip or a set of chips that physically control the device
  - controlling the device is complicated, and CPU could be doing other things, so the controller presents a simpler interface to the OS
  - there are many different types of controllers
- **device**
  - connects to the computer through the controller
  - follows some agreed standard for communication



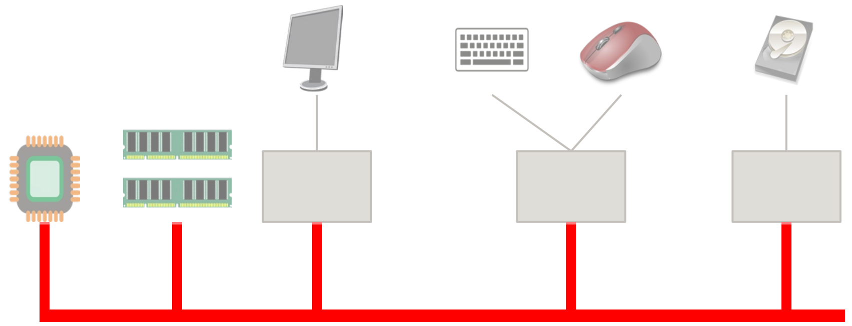
Devices

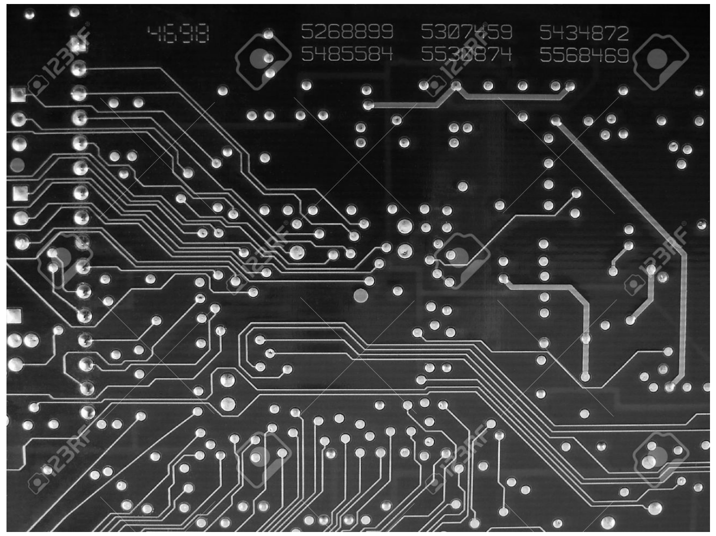
- **device driver**
  - device specific software that kernel needs to talk to a device
  - typically run in unrestricted mode
  - typically written by the controller/device manufacturer, following some abstraction defined by OS
  - often implemented as kernel modules, loaded on demand



Buses

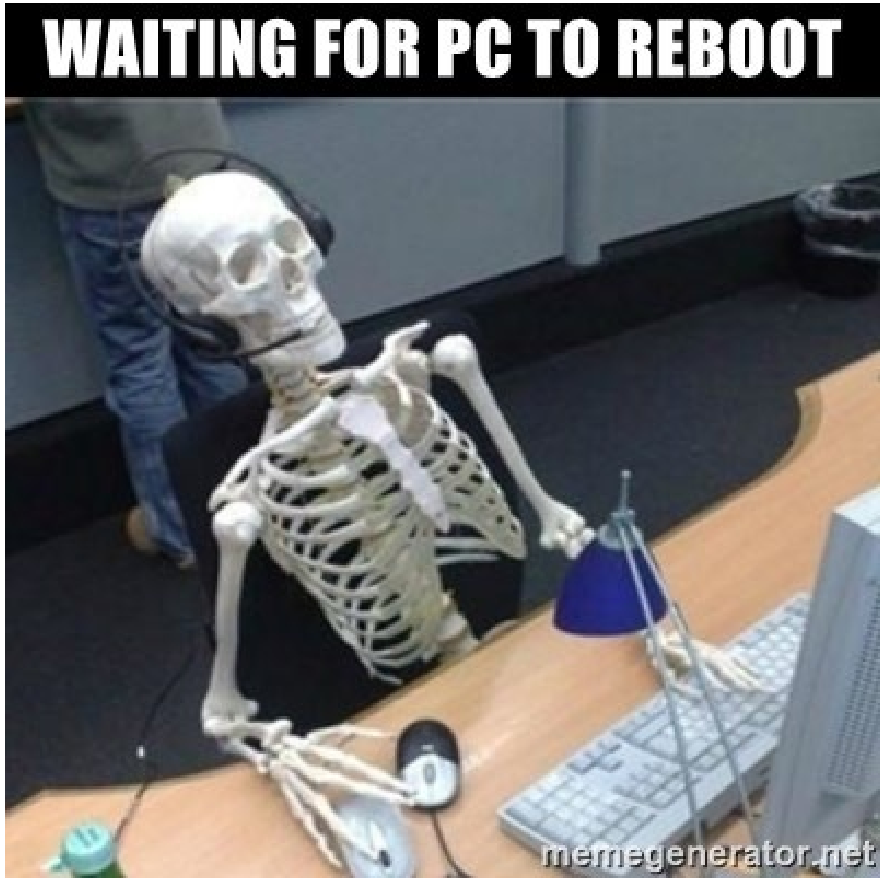
- a communication system for transferring data between different computer components
- modern computer systems have multiple busses, eg. cache, memory, PCI, ISA, etc
- each has a different transfer rate and function
- OS must be aware of all of them for configuration and management
- for example, collecting information about the I/O devices
- assigning interrupt levels and I/O addresses
- much of this is done during the boot process





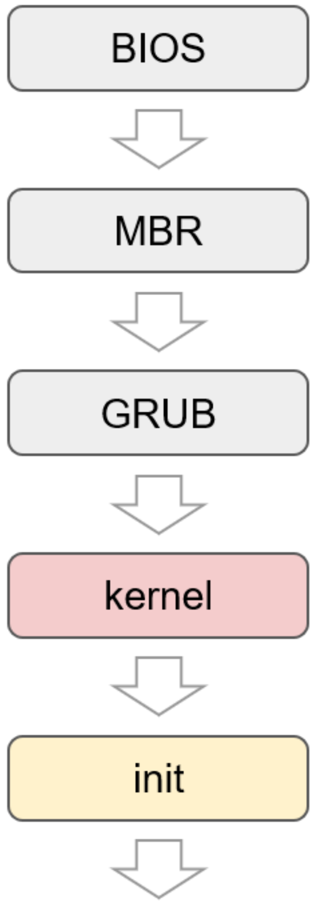
Booting

Booting



Booting a (Linux) system

- when the computer is booted, the BIOS is started
  (Basic Input Output System) is a program stored on motherboard
- check the RAM, keyboard, other devices by scanning the ISA and PCI buses
- record interrupt levels and I/O addresses of devices, or configure new ones
- determine the boot device (ie. try list of devices stored in CMOS)
- read & run primary boot loader program from first sector of boot device
- read & run secondary boot loader from potentially another device
- read in the OS kernel from the active partition and start it
- OS queries the BIOS to get the configuration information and initialize all device drivers in the kernel
- OS creates a device table, and necessary background processes, then waits for I/O events



Kernel

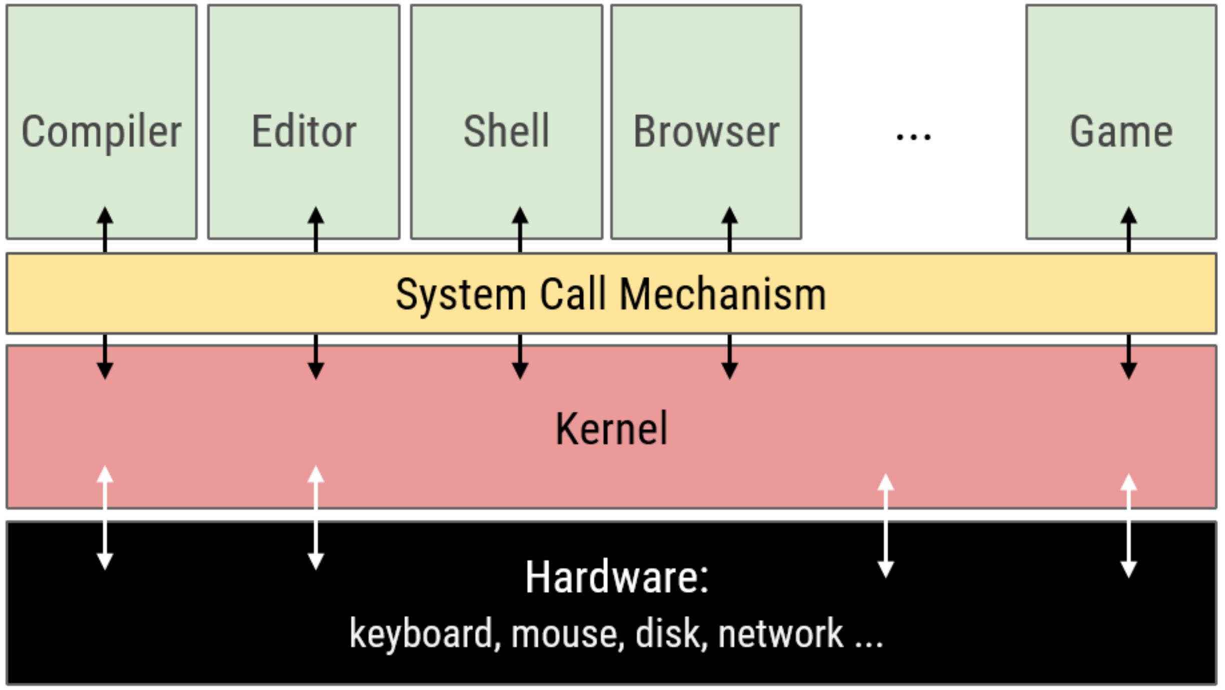
Kernel

- the central part, or the "heart" of the OS
    - located and started by a bootstrap program (boot loader)
    - the only software that can talk directly to hardware
    - provides services to applications via system calls
    - much of the kernel is a set of routines
        - some invoked in response to interrupts - when devices need attention
        - others when applications request services
    - kernel is "running" at all times on the computer

Kernel and user mode

- most modern CPUs support at least two privilege levels: kernel mode and user mode
- the mode can be switched by special instructions
- when CPU is in kernel mode (a.k.a. unrestricted, privileged, supervisor mode):
  - all instructions are allowed
  - all I/O operations are allowed
  - all memory can be accessed
  - only kernel runs in kernel mode
- when CPU is in user mode (aka restricted mode):
  - only some operations are allowed, the rest are disallowed
  - e.g. switching to kernel mode is disallowed (of course),
    I/O instructions not allowed, access to some parts of memory not allowed, …
  - illegal instructions result in traps (exceptions)
  - all applications run in user mode (including ones that came with the OS)

Kernel vs. user mode



user mode - no access to hardware, limited instruction set

kernel mode - full access to hardware, full instruction set

User mode & system calls

- if all applications run in user mode, how do they talk to hardware?
  - how do they read/write files? how do they display information on a monitor? ...
- applications ask the kernel to perform I/O via system calls
  - system call = mechanism for calling a kernel routine
  - system call needs to include transition from user mode to kernel mode

    i.e. it cannot be a simple subroutine call
- system calls are usually implemented using a special instruction
  - the instruction allows the switch from user to kernel mode to be safe
    i.e. only predefined kernel routines can be invoked
  - common mechanism is by invoking a trap (software interrupt)
    e.g. `SWI n`, `INT n`, …
  - when the kernel routine is done, application resumes in user mode

Kernel designs

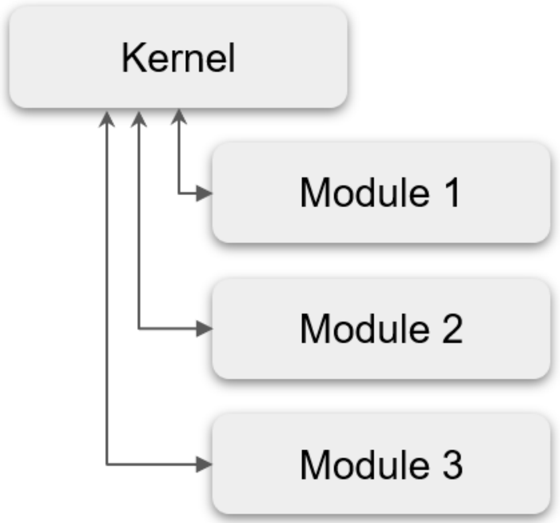
- what goes into a kernel and what does not?
- important trade-off to consider: stability vs speed
- code in kernel runs faster, but big kernels have more bugs → higher system instability

Kernel designs

- monolithic kernels (e.g., MS-DOS, Linux)
  - the entire OS runs as a single program in kernel mode
  - pros: fastest
  - cons*: 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
  - pros*: less bugs, easier to port, more stable
  - cons: slower
- some OSes claim to have hybrid kernels
  - trying to balance the cons/pros of monolithic kernels and microkernels

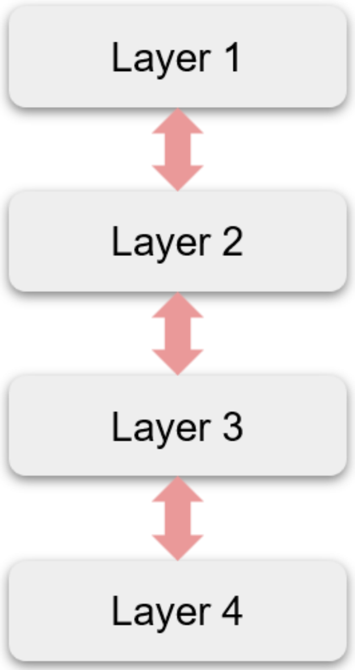
Kernel modules

- some parts of kernel implemented as modules
- 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, e.g. loading a driver for a video-card
  - or could be done later, e.g. after USB device is plugged in
  - 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

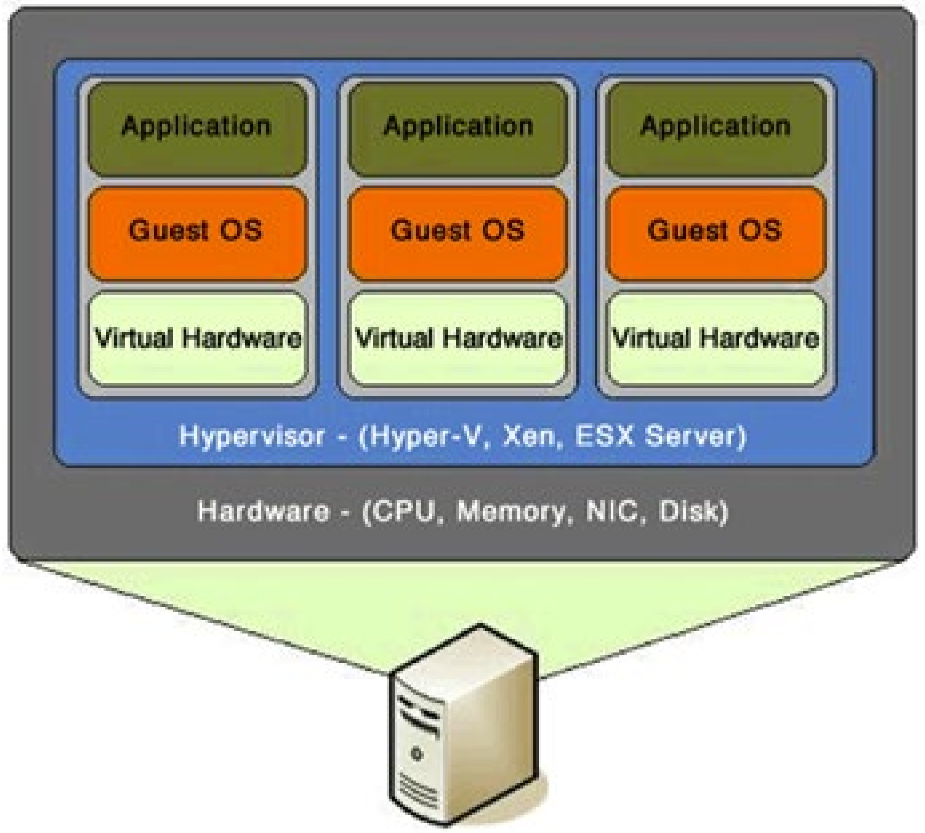
- kernel components organized into a hierarchy of layers
- layers above constructed upon the ones below it
- sounds great in theory, but…
  - hard to define layers, needs careful planning
  - less efficient since each layer adds overhead to communication
  - not all problems can be easily adapted to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



Virtual Machines

Virtual machines

- virtual machines (VMs) emulate computer systems
  - in software, with some H/W support from CPU
  - create 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 ESXi
  - 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, e.g. Docker, LXC



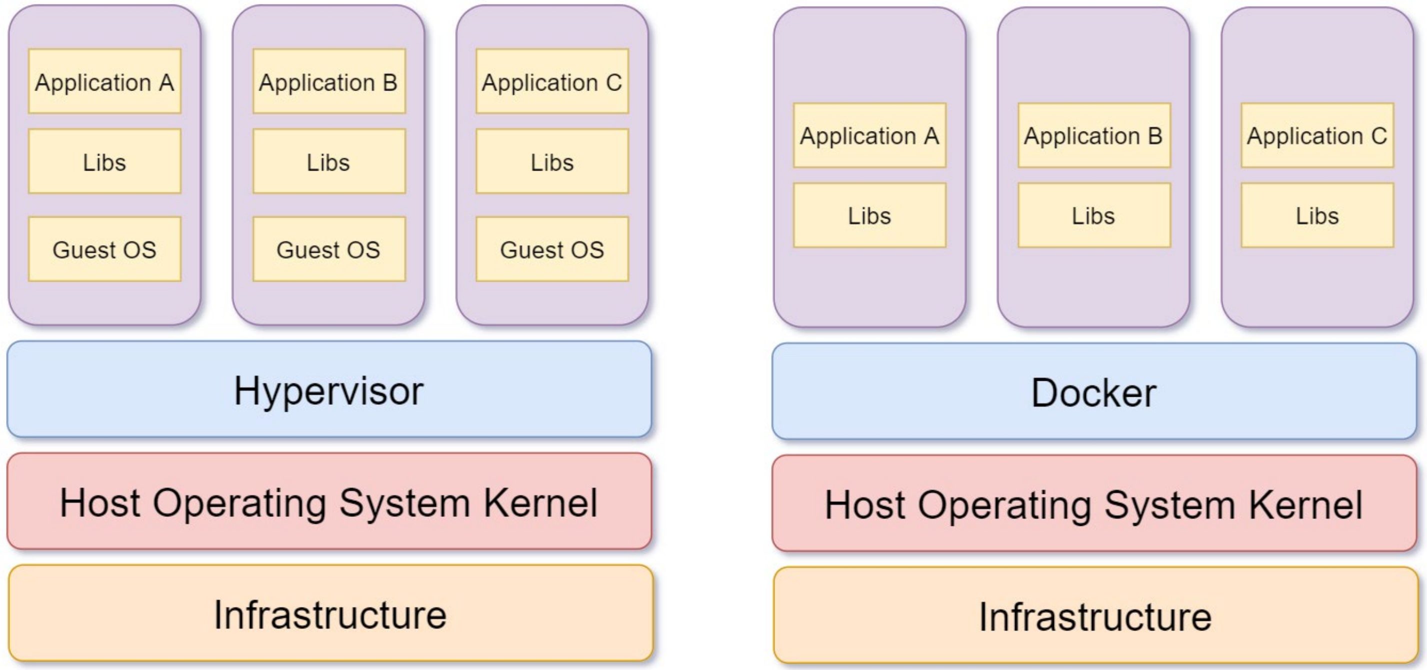
Benefits of VMs

- working on CPSC 457 assignments under windows

Docker

Container versus VM

- Instead of having a full OS in a VM, a container instead access host OS through the limiting containerization environment (Docker)
- Still maintains isolation like a VM



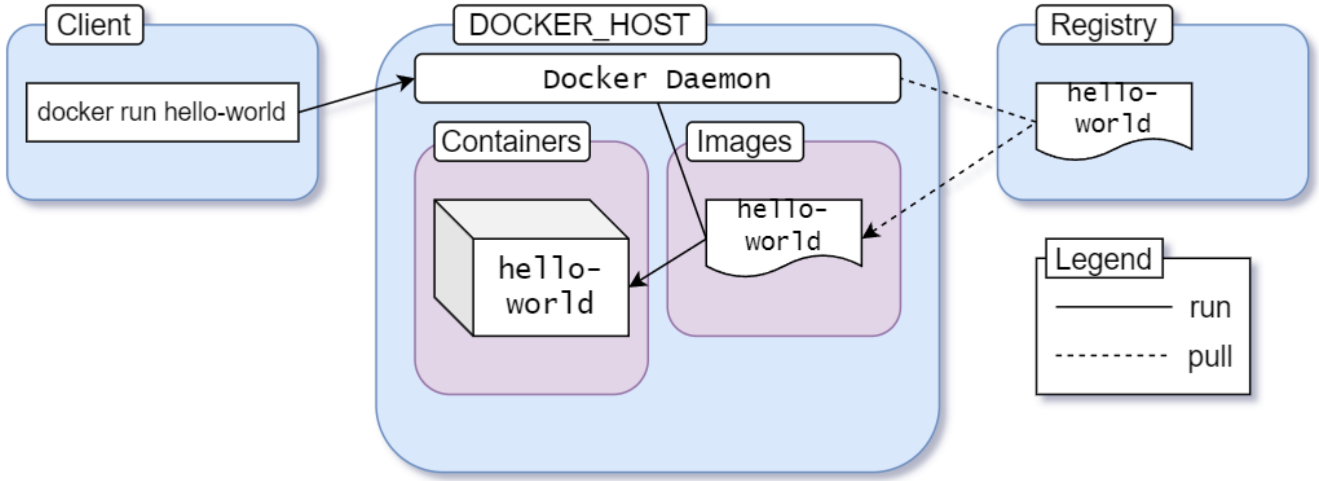
UNIVERSITY OF CALGARY

Terminology

- Image
  - Multi-layered files that act as templates to make containers
  - Frozen-read only copies of a container
  - OCI (open container initiative) as standardized this
- Containers
  - Image in a running state (writable layer on top of read-only image)
- Registry
  - Stores images (DockerHub), can download freely
  - Example there are Data Science images hosted that install 10s/100s of common packages
  - Instead of managing each individual computer install I could register a common image for a course and have everyone use it with the required tools

Terminology

- Docker Daemon
  - Sits around in background waiting for commands to manage containers
- Docker Client
  - Takes commands from user
- REST API
  - Bridge between client and daemon



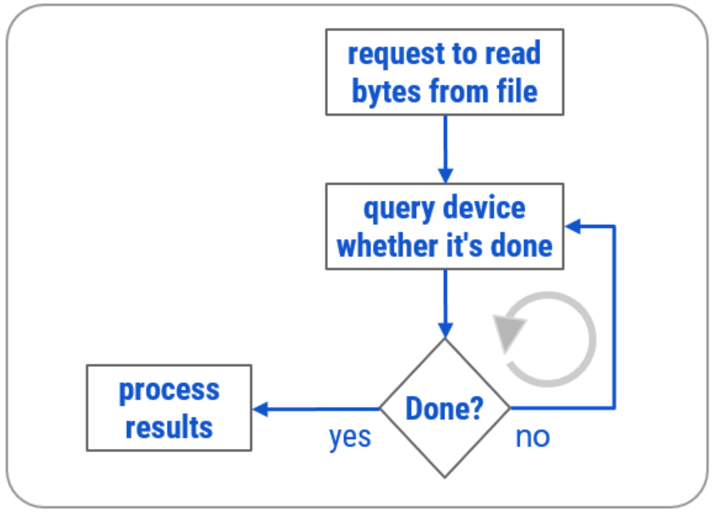
I/O

I/O

- how do we write programs that interact with devices?
- in absence of OS
  - we can use special instructions or memory mapped devices
  - we need to use assembly or C/C++ with some `asm {}`
  - these are usually non-blocking mechanisms, i.e.
    CPU continues execution with next instruction
- with OS installed, we use system calls
  - most OSes have blocking and non-blocking versions of system calls
  - blocking calls will suspend execution until request is finished (e.g. until `printf()` returns)
  - non-blocking calls schedule the request and application continues to run,
    it is up to the application to detect when request is finished

Low level I/O - using busy waiting

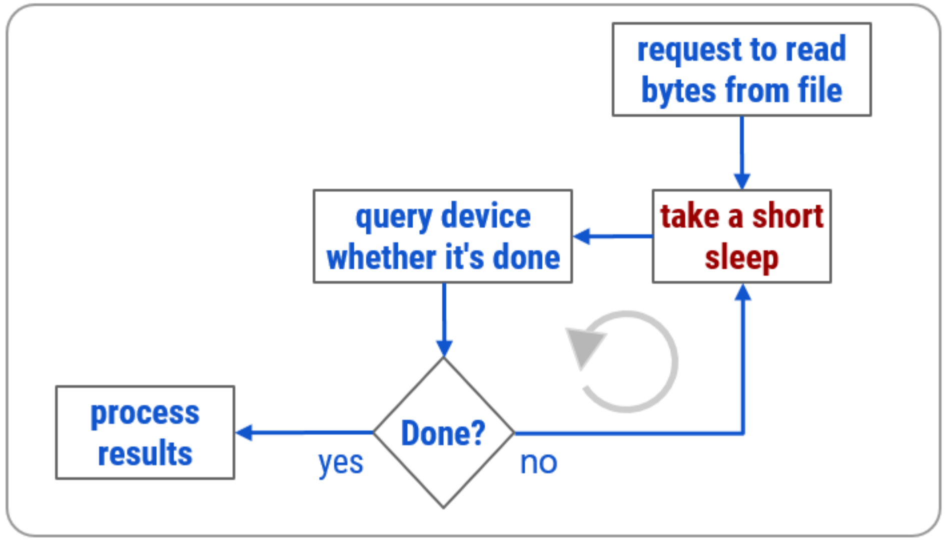
- I/O using **busy waiting** / spinning / busy looping:
  CPU repeatedly checks if device is ready


request to read bytes from file
query device whether it's done
Done?
yes
no
process results

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

I/O - busy waiting with delay

- I/O using busy wait and sleep
  similar to busy waiting, with a short delay to reduce CPU usage


request to read bytes from file
take a short sleep
query device whether it's done
Done?
no
yes
process results

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

I/O Interrupts

I/O - using interrupts

- I/O using interrupts:

```
request reading a file,
but ask device to send interrupt when done
# your program continues to execute, or can choose to sleep
...
...
...
# when interrupt happens, an interrupt handler gets executed:
interrupt_handler:
    process results
```

- 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 device (or controller) supports interrupts

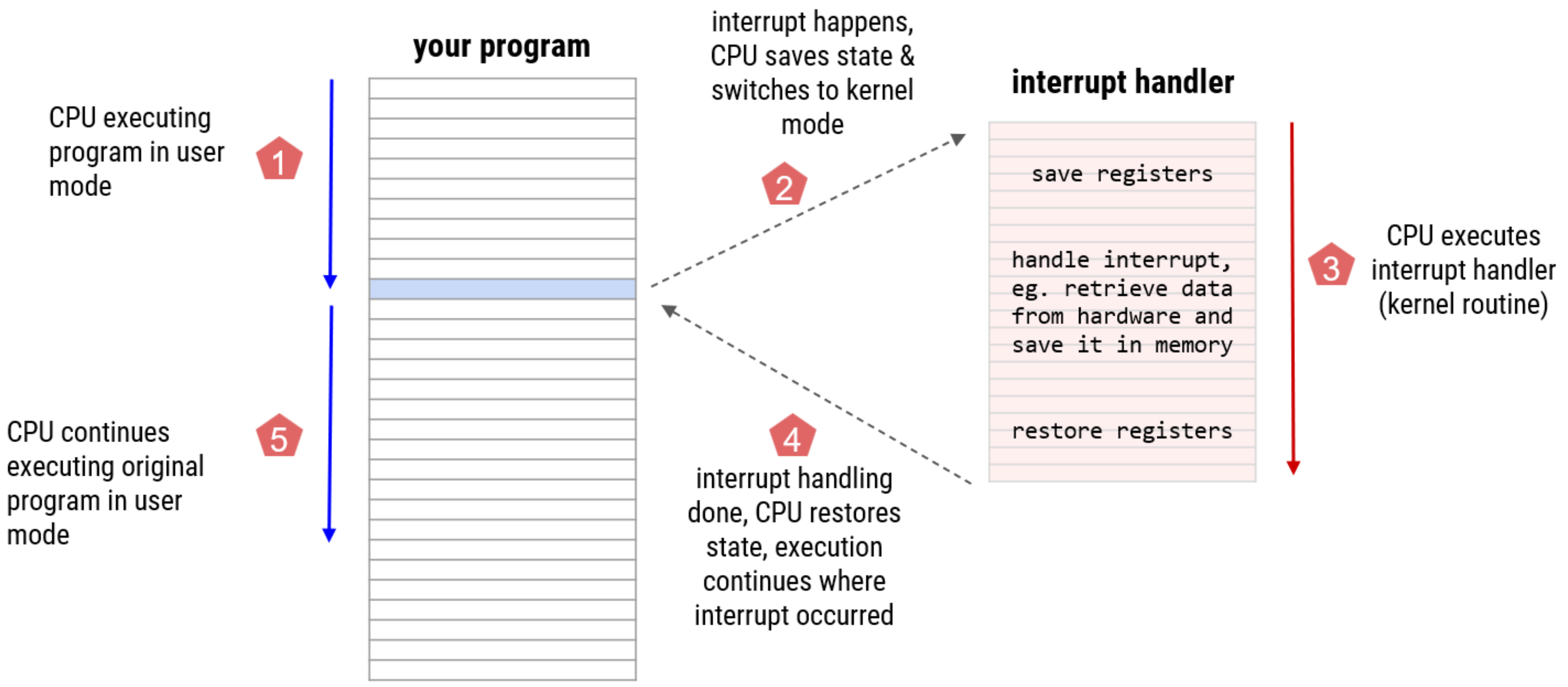
Interrupts

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

  e.g. a printer finished printing, new data arrived on a network card, a disk finished saving data, a program performed illegal instruction

- CPU usually responds to interrupt immediately
  - CPU temporarily suspends current activity
  - CPU saves its state somewhere, typically in memory, such as stack
  - CPU switches to kernel mode if needed
  - CPU executes a predefined routine (interrupt handler or **i**nterrupt **s**ervice **r**outine)
    - which routine[s] gets executed is configurable
    - kernel makes sure all interrupts are handled by kernel code
  - CPU restores user mode if needed
  - eventually CPU restores saved state and resumes original execution

Interrupts


your program
interrupt handler
CPU executing program in user mode
1
interrupt happens, CPU saves state & switches to kernel mode
2
save registers
handle interrupt, eg. retrieve data from hardware and save it in memory
restore registers
3
CPU executes interrupt handler (kernel routine)
4
interrupt handling done, CPU restores state, execution continues where interrupt occurred
5
CPU continues executing original program in user mode

Hardware interrupts

- the source of the interrupt is another device
    - e.g. 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
- modern CPUs allow defining interrupt priorities, and even allow temporarily disabling interrupts

Software interrupts (exceptions / traps)

- similar to hardware interrupts, but the source of the interrupt is the CPU itself
  - handled similarly to hardware interrupts
- SW interrupts can be 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 misbehaved application, OS usually terminates it
- intentional software interrupt, aka. **traps** (in this course *)
  - trap occurs as a result of executing a special instruction, e.g. `INT`
  - the purpose is to execute a predefined routine in kernel mode
  - some operating systems use traps to implement system calls

Hardware vs Software Interrupts

Hardware Interrupts:

- external event delivered to the CPU
- origins: disk, 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
- occur as a result of execution of a machine instruction

but both ...

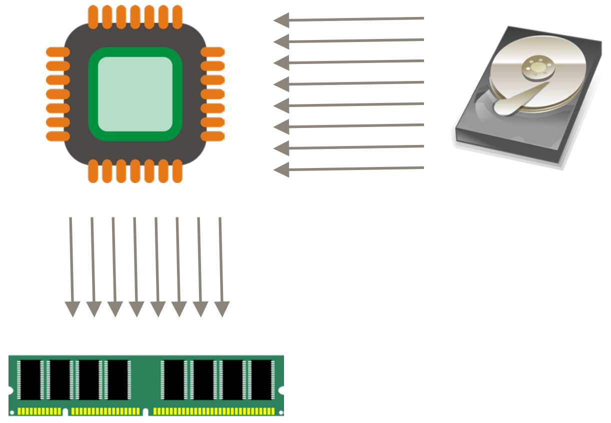
- put the CPU in a kernel mode
- save the current state of the CPU
- invoke a kernel routine, defined by the OS
- resume the original operations when done, restoring user mode

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

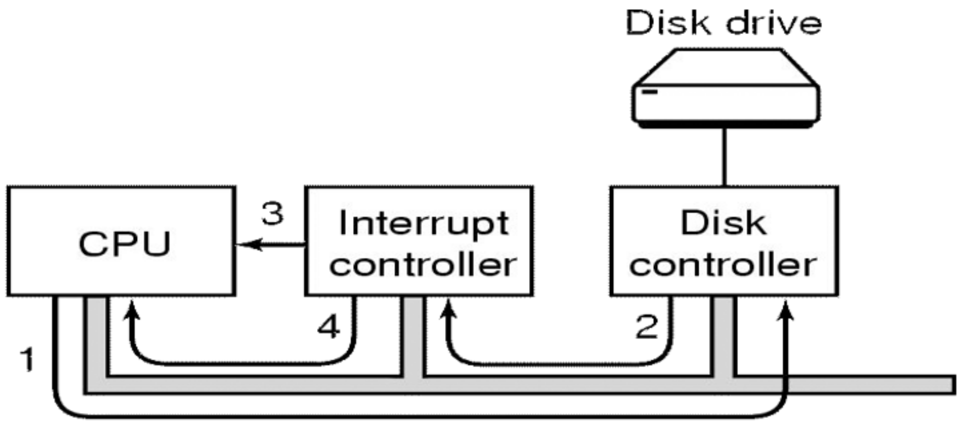
Limits of interrupts

- CPU can run other programs while waiting for I/O, but …
  - 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



Using Interrupts to do I/O

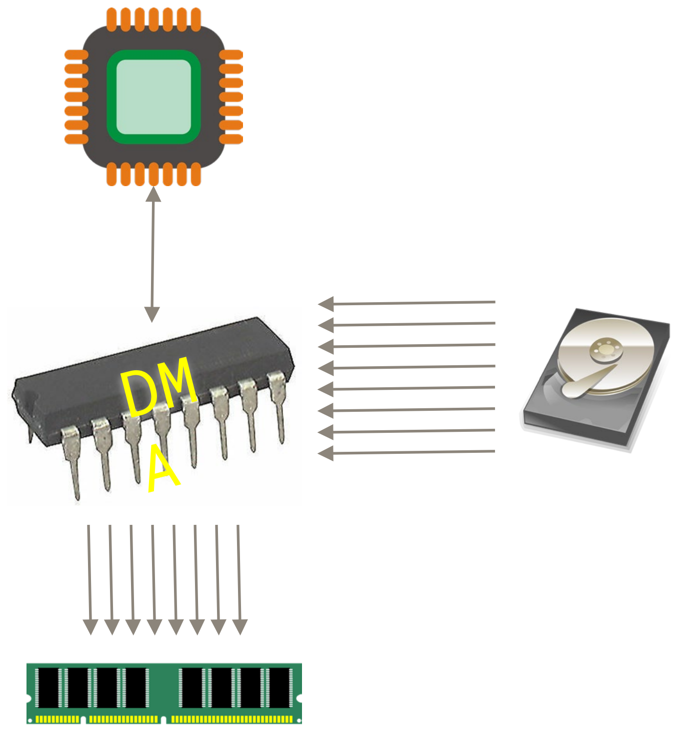
- Kernel talks to a device driver to request an operation.
- The device driver tells the controller what to do by writing into its device registers.
- The controller starts the device and monitors its progress.
- When the device is done its job, the device controller signals the interrupt controller.
- 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.



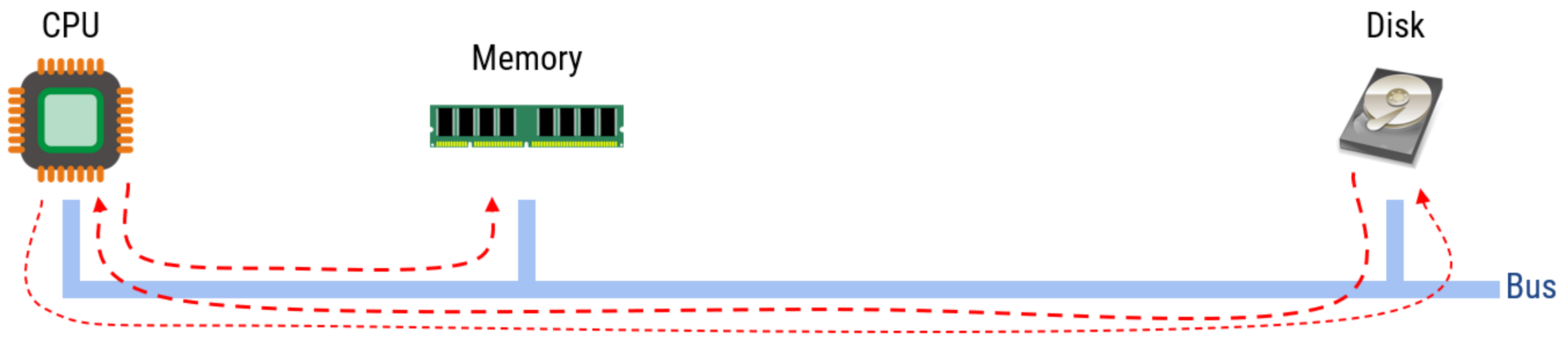
DMA

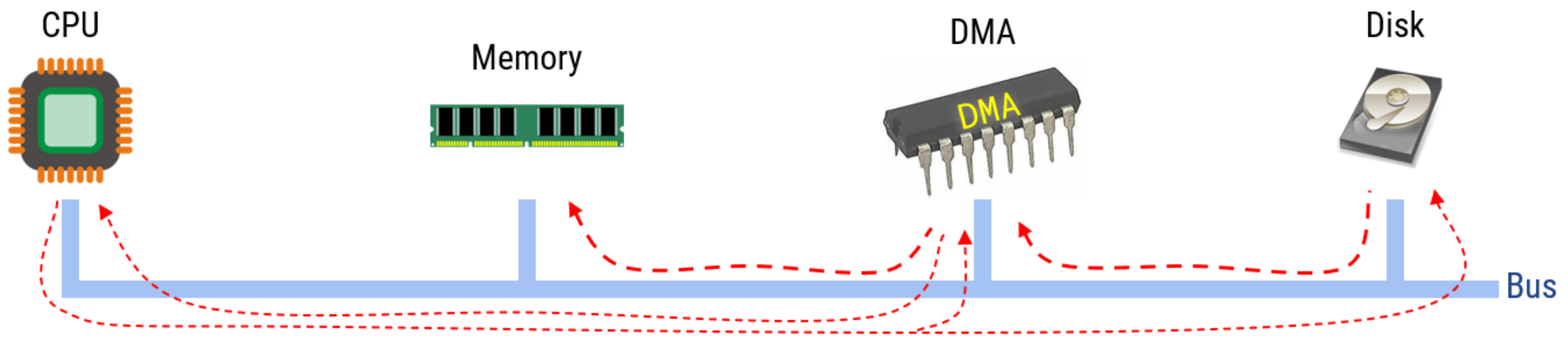
Direct memory access (DMA)

- 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)

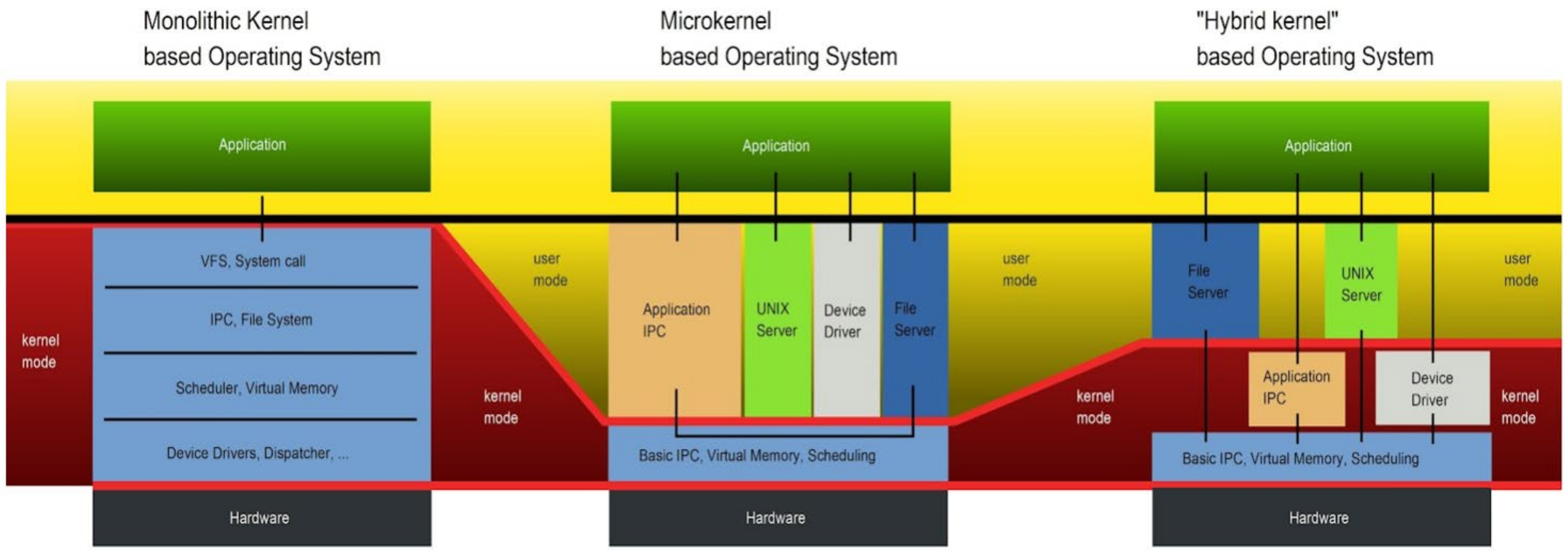




Kernel Designs

UNIVERSITY OF CALGARY

Monolithic / Microkernel / Hybrid



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

* Kernel designs – theory vs practice

Linus (2006):

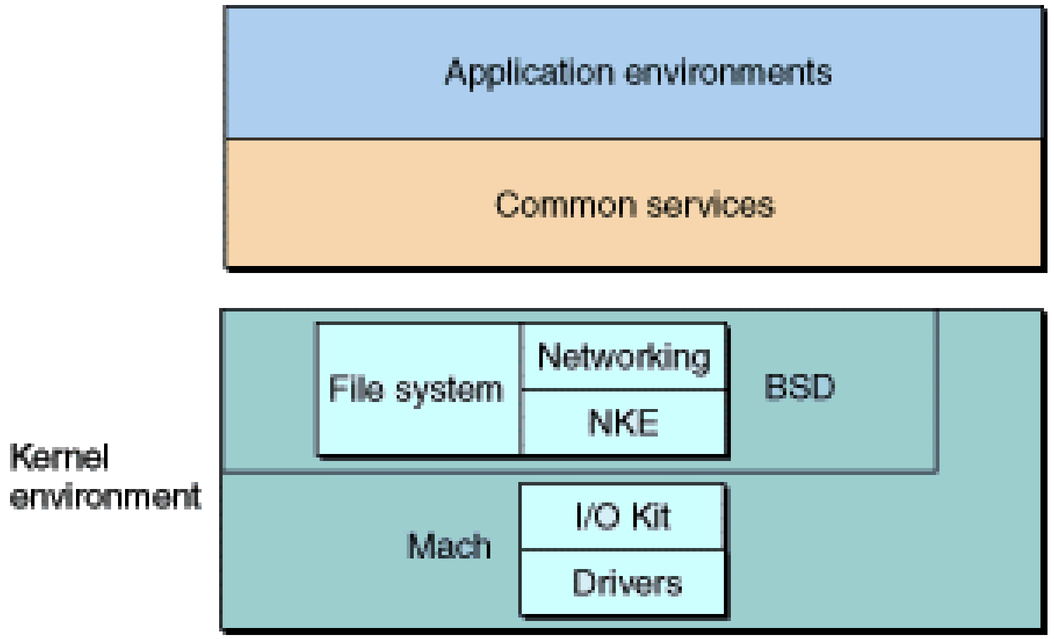
It's ludicrous how micro-kernel proponents claim that their system is "simpler" than a traditional kernel. It's not. It's much much more complicated … Microkernels are much harder to write and maintain … whenever you compare the speed of development of a microkernel and a traditional kernel, the traditional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

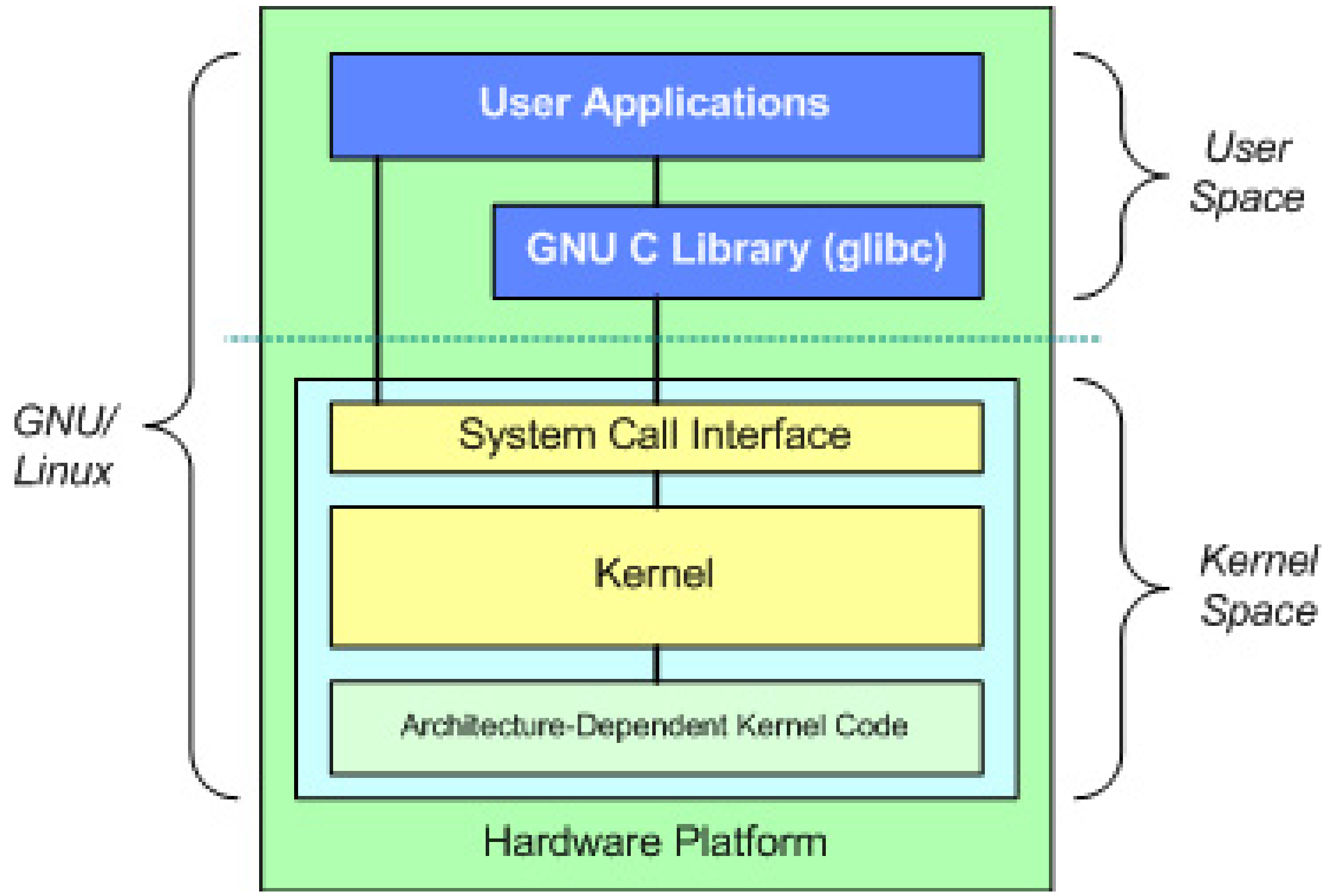
https://www.realworldtech.com/forum/?threadid=65915&curpostid=65936

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

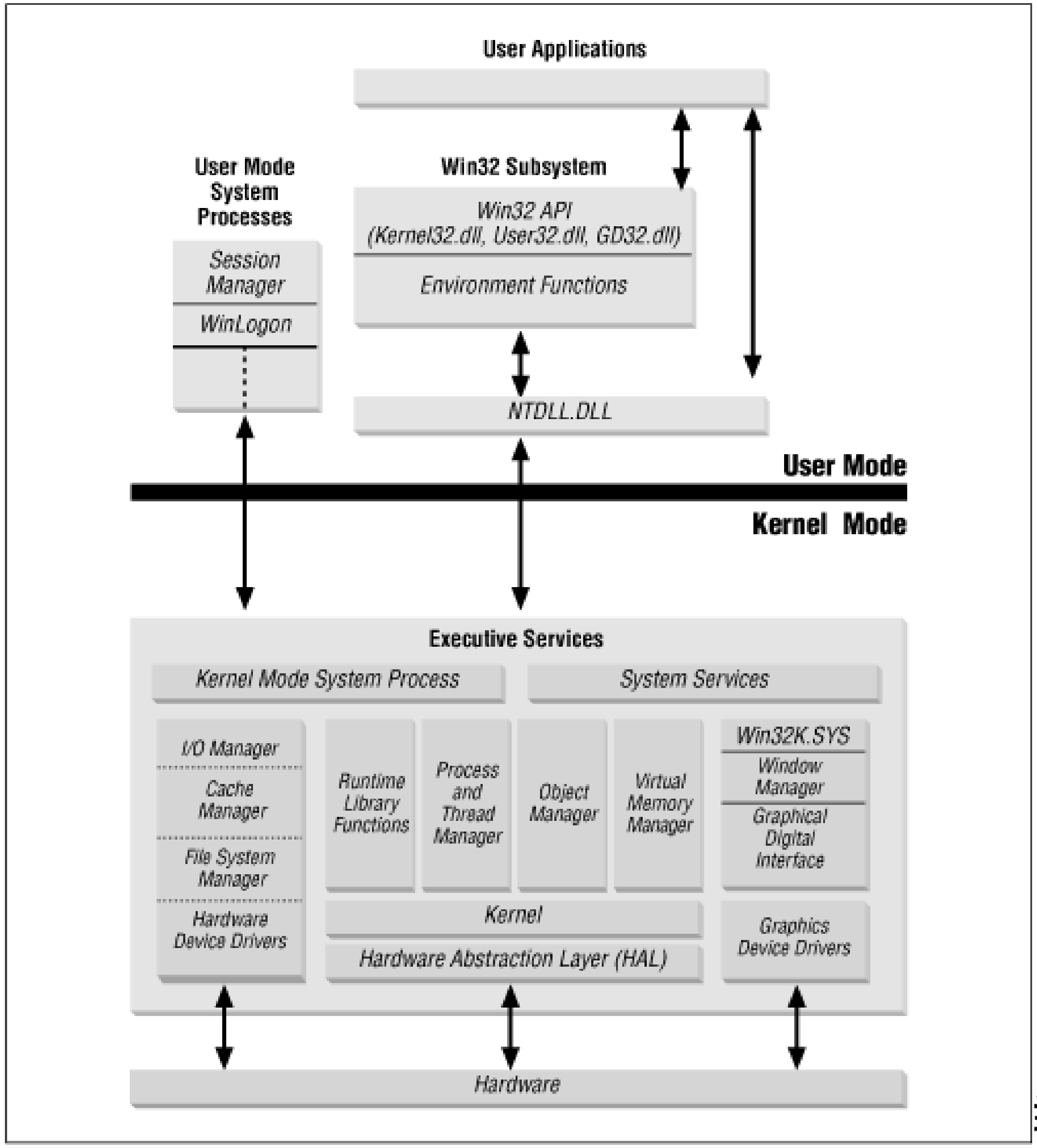


- still considered
  monolithic kernel,
  but with some layers,
  and dynamically loadable modules

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

Win NT structure

- hybrid kernel
- modules & layers



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

Review

Summary

- Definition/History
- Hardware review
  - Processor
  - Memory & Disks, caching
  - Devices & I/O
  - Buses
- Bootstrapping
  - Traps
  - Kernel mode v.s. user mode
- Virtual machines
- Interrupts
  - Interrupts vs. traps
  - DMA
- OS structure
  - Monolithic systems, Microkernel
  - Modular kernels and layered approach

UNIVERSITY OF CALGARY

Review

- Which of the following concepts introduced interactive service for multiple users?
  - batch system
  - multiprogramming
  - spooling
  - Timesharing
- Invoking a system call will cause a trap.
  - True or False
- Applications run in user mode.
  - True or False
- Device drivers run in kernel mode.
  - True or False

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

Onward to … System Calls

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY