CS 3502
Operating Systems

Operating Systems Overview

Kun Suo

Computer Science, Kennesaw State University

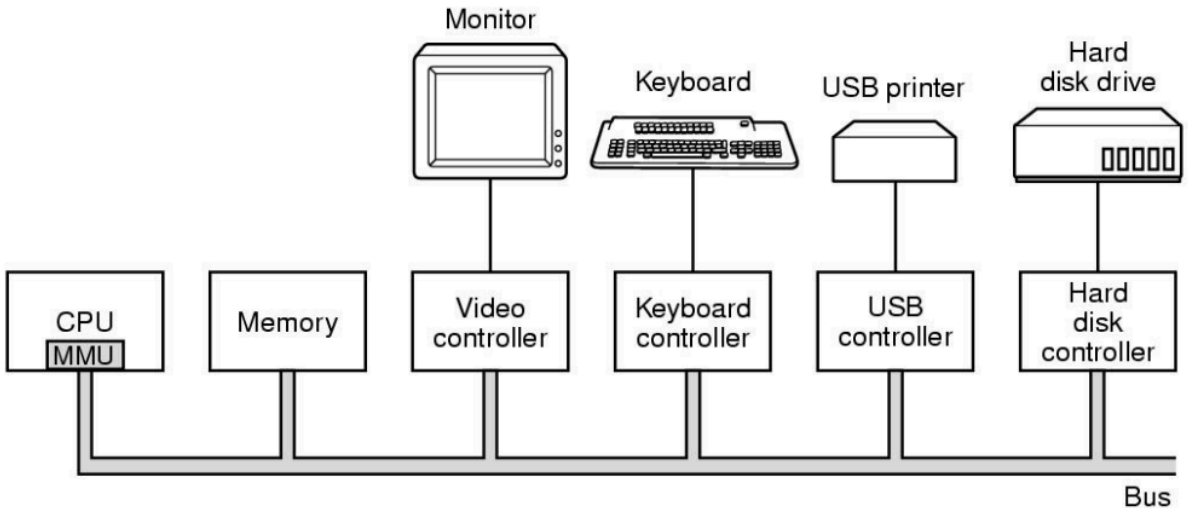
https://kevinsuo.github.io/

Outline

- Hardware Review
- What is an OS?
  - Virtualization
  - Concurrency
  - Persistence
- History of OS
  - Monolithic
  - Microkernel
  - Hybrid

Computer Hardware Review

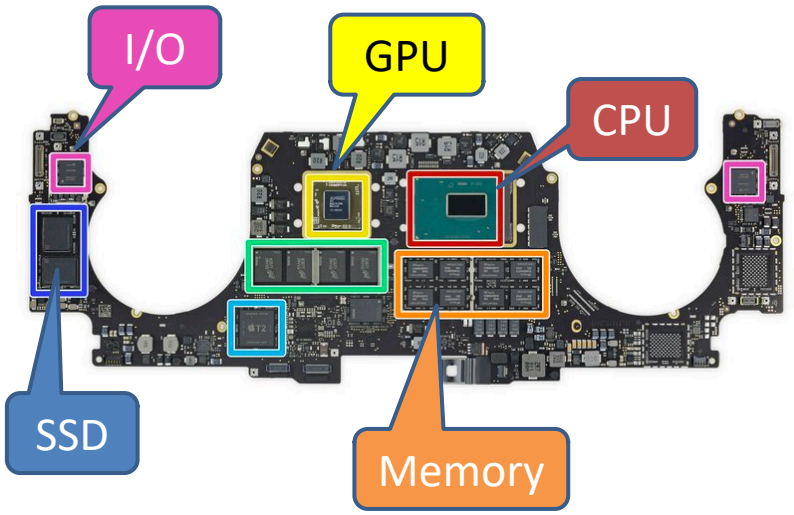
- Basic components of a simple personal computer



- CPU: data processing
- Memory: volatile data storage
- Disk: persistent data storage
- NIC: inter-machine communication
- Bus: intra-machine communication

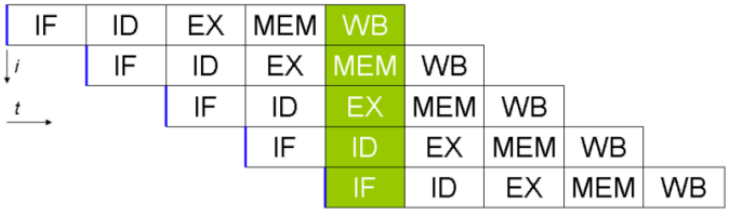
Computer Hardware Review

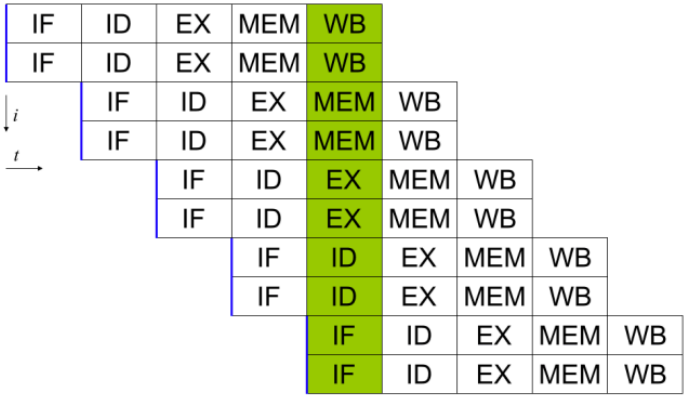
Computer Hardware Review

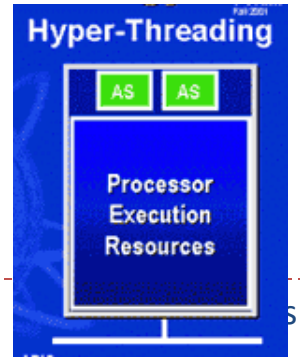


Central Processing Unit (CPU)

- Components
  - Arithmetic Logic Unit (ALU) -> Compute and data
  - Control Unit (CU) -> control device and system
- Clock rate
  - The speed at which a CPU is running
- Data storage
  - General-purpose registers: EAX, EBX …
  - Special-purpose registers: PC (program counter), SP (stack), IR (instruction register) …
- Parallelism
  - Instruction-level parallelism
  - Thread-level parallelism
    - Hyper-threading: duplicate units that store architectural states
    - Replicated: registers. Partitioned: ROB, load buffer… Shared: reservation station, caches

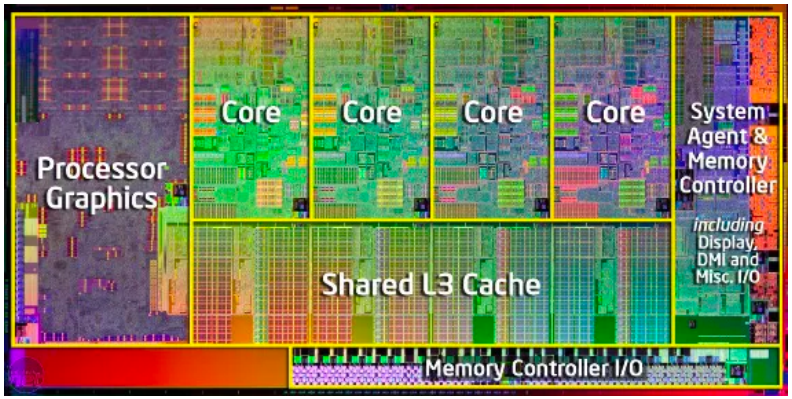


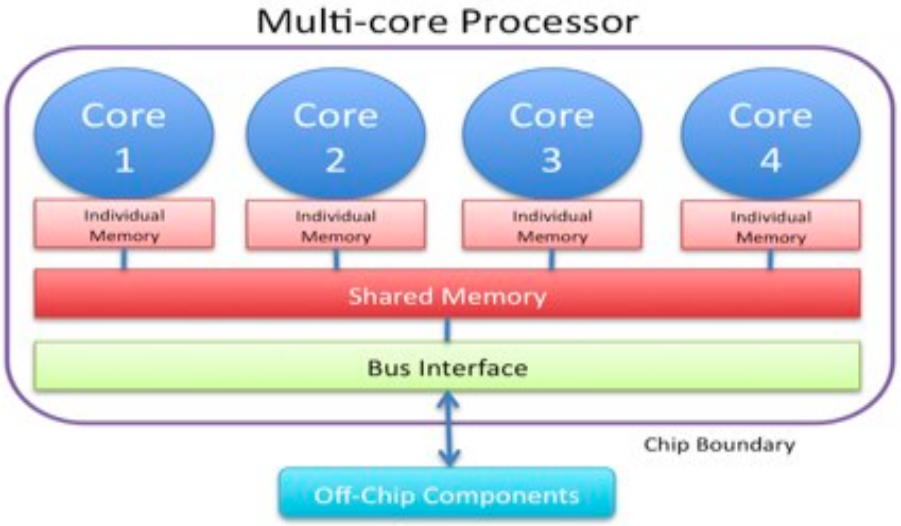




Multi-Core Processors (SMP)

- Multiple CPUs on a single chip

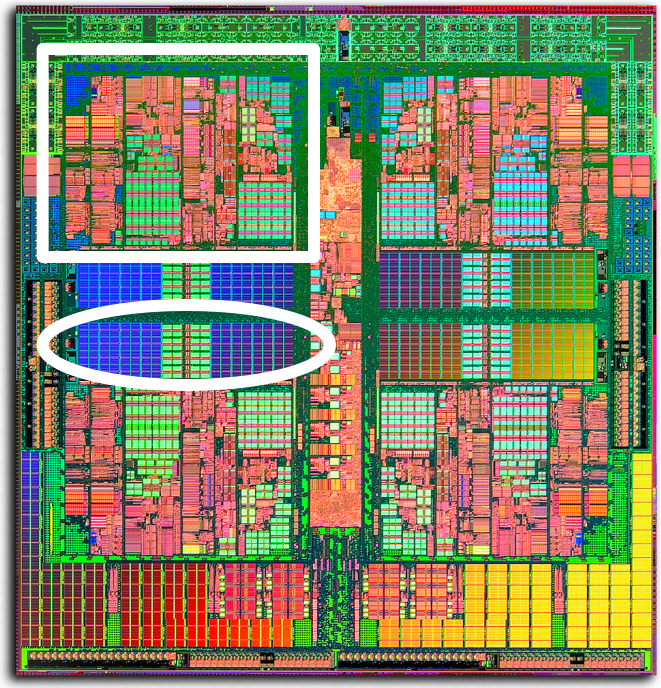


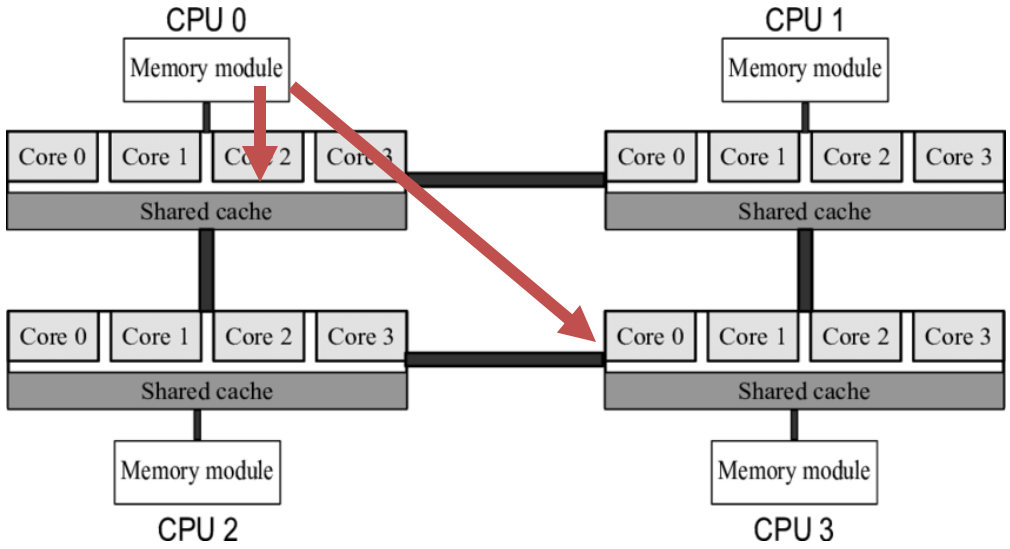


Symmetric multiprocessing (**SMP**)

Multi-Core Processors (NUMA)

- Multiple CPUs on a single chip

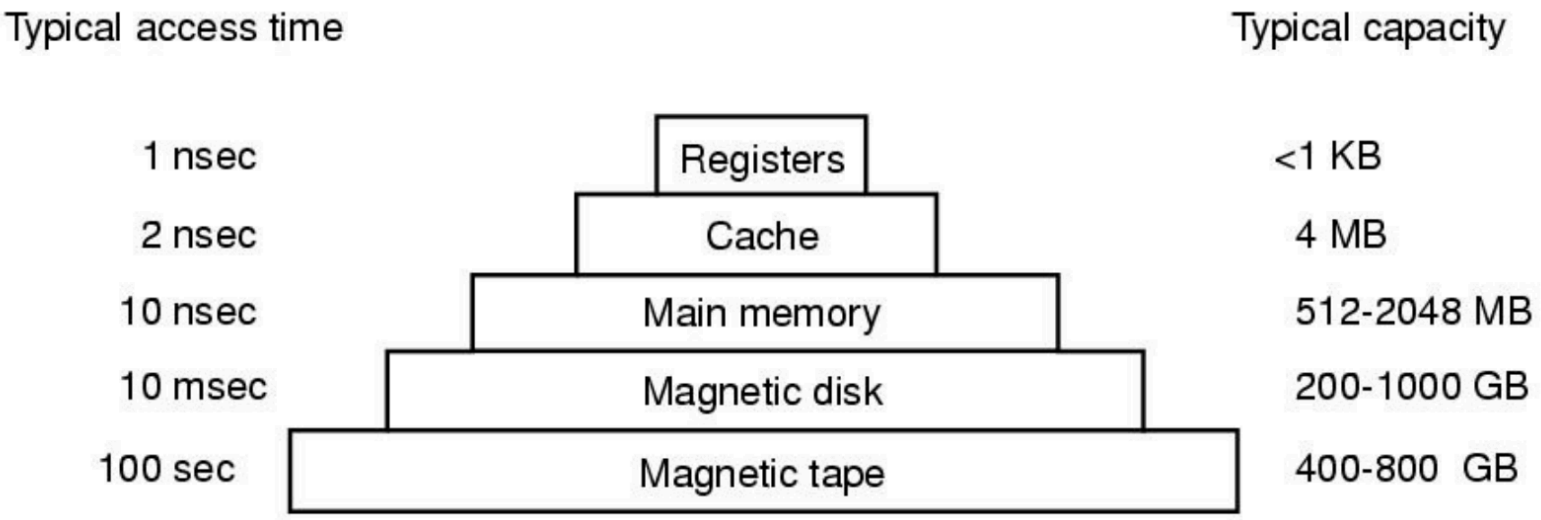




Non-uniform memory access (**NUMA**)

Memory

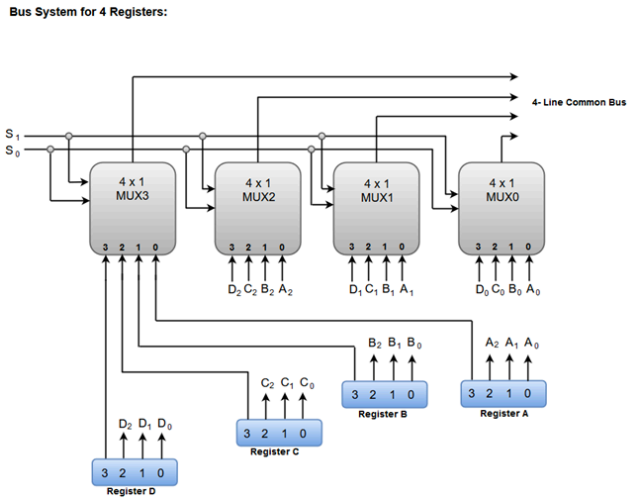
- A typical memory hierarchy

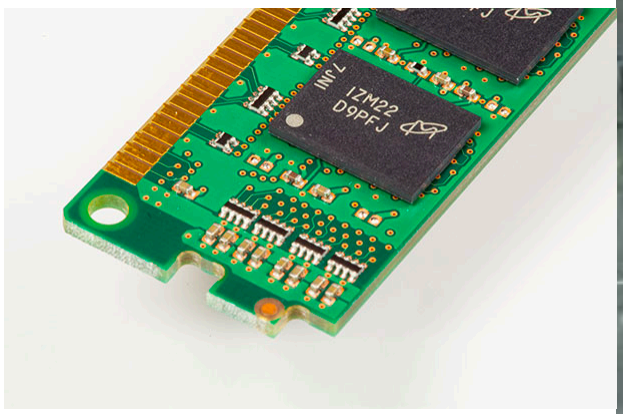


Minimize the access time vs. Cost

Memory

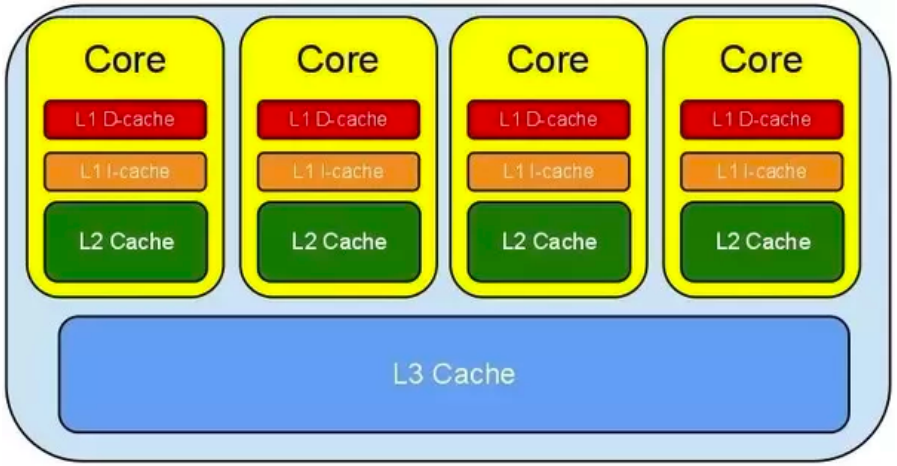
- A typical memory hierarchy

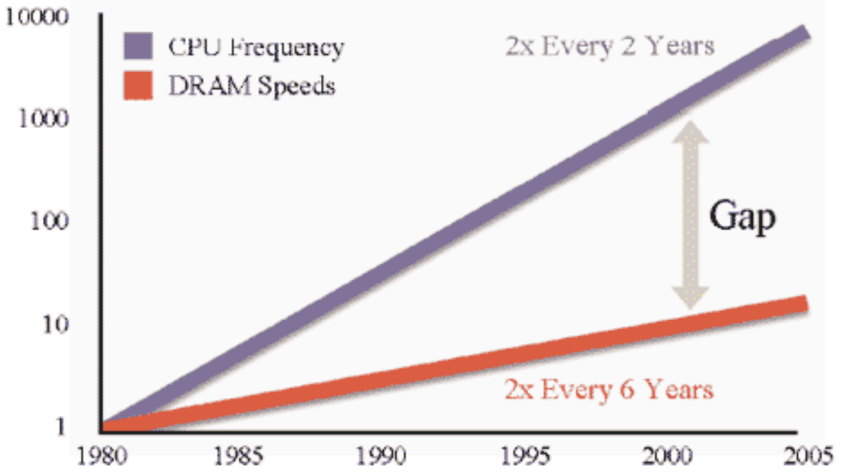




Cache

- Why Cache is important?





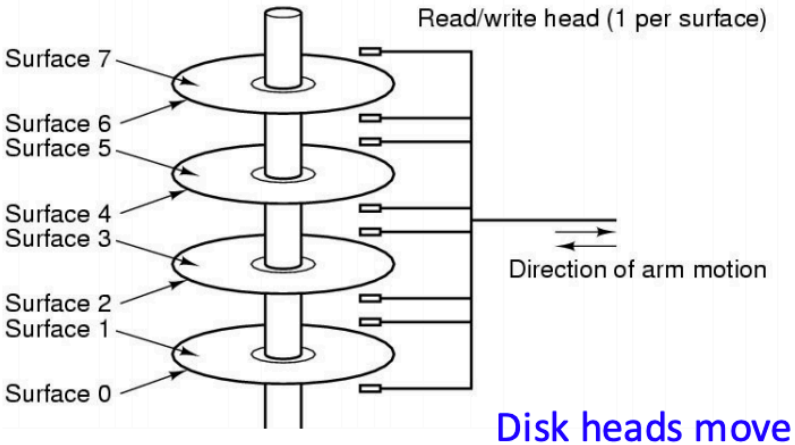
A larger size than registers
A much faster speed than memory
Tradeoff between performance and cost

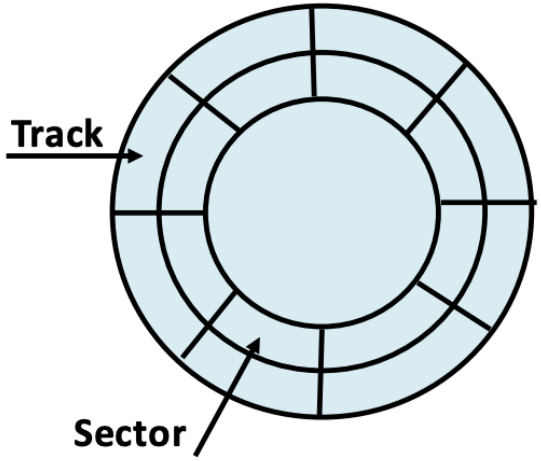
MacBook Pro

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro15,1
Processor Name:	Intel Core i9
Processor Speed:	2.3 GHz
Number of Processors:	1
Total Number of Cores:	8
L2 Cache (per Core):	256 KB
L3 Cache:	16 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB
Boot ROM Version:	220.270.99.0.0 (iBridge: 16.16.6568.0.0,0)
Serial Number (system):	C02YR4JHLVCJ
Hardware UUID:	DCC2D30A-9630-57B5-89A2-5F2B85254DC1

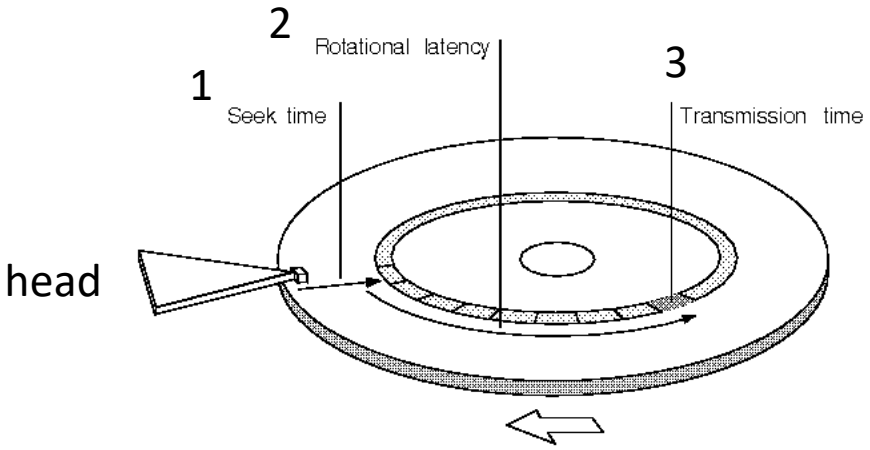
Disk





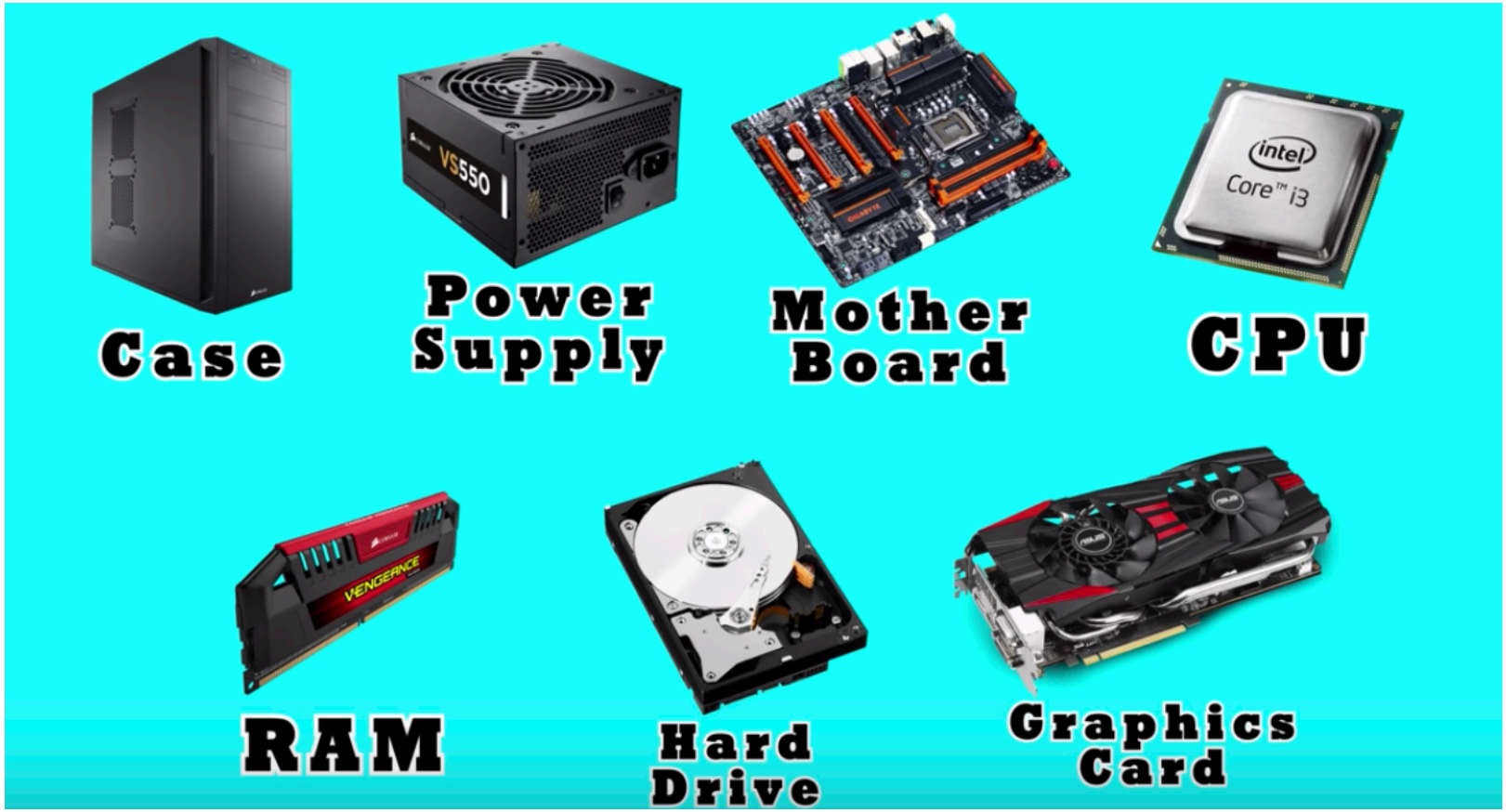
- A stack of platters, a surface with a magnetic coating
- Typical numbers (depending on the disk size):
  - 500 to 2,000 tracks per surface
  - 32 to 128 sectors per track
    - A sector is the smallest unit that can be read or written
- Originally, all tracks have the same number of sectors

Disk



- Disk head: each side of a platter has separate disk head
- Read/write data is a three-stage process:
  - Seek time: position the arm over the proper track
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Average seek time as reported by the industry:
  - Typically in the range of 8 ms to 15 ms

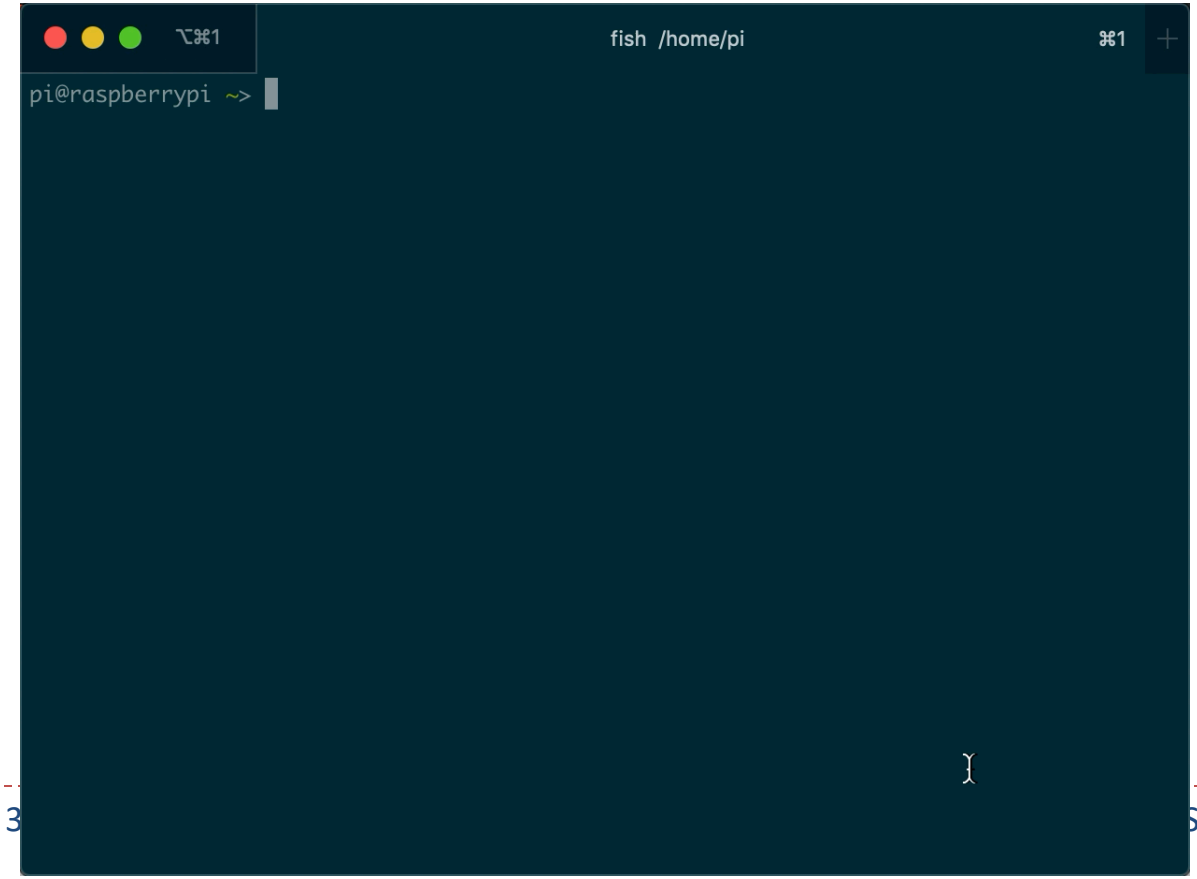
An interesting video introducing hardware



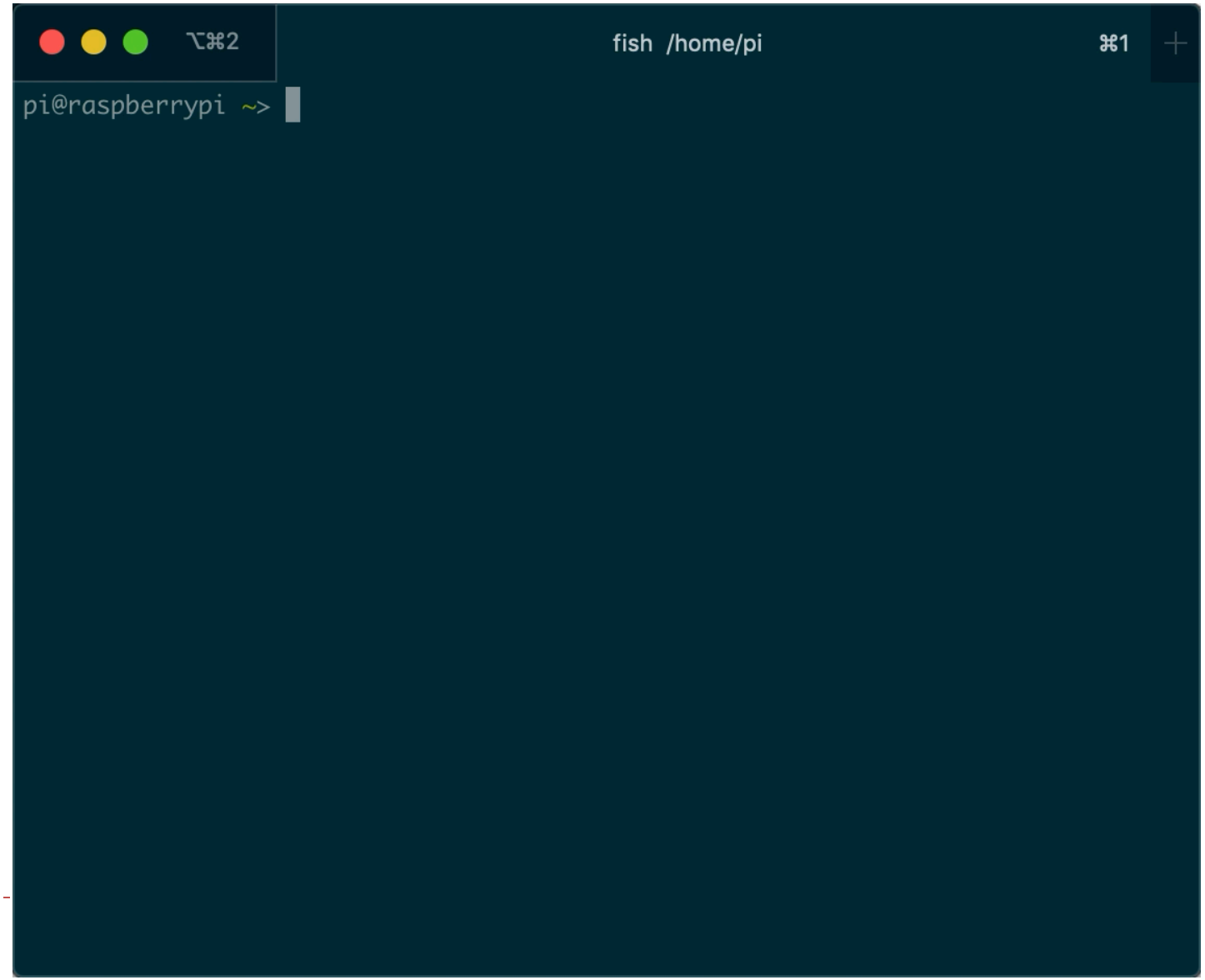
https://www.youtube.com/watch?v=ExxFxD4OSZ0

How to get my machine spec?

- Inxi: https://www.tecmint.com/inxi-command-to-find-linux-system-information/



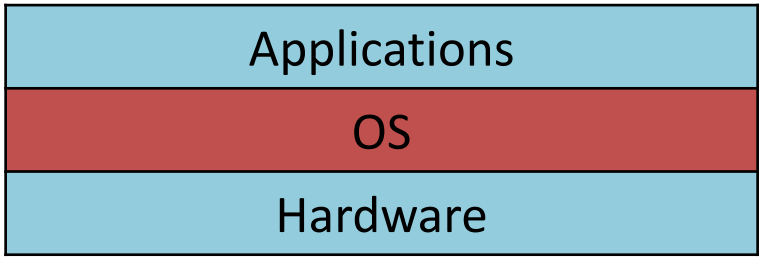
How to get my machine spec? One command for All!



Outline

- Hardware Review
- What is an OS?
  - Virtualization
  - Concurrency
  - Persistence
- History of OS
  - Monolithic
  - Microkernel
  - Hybrid

What is an OS?

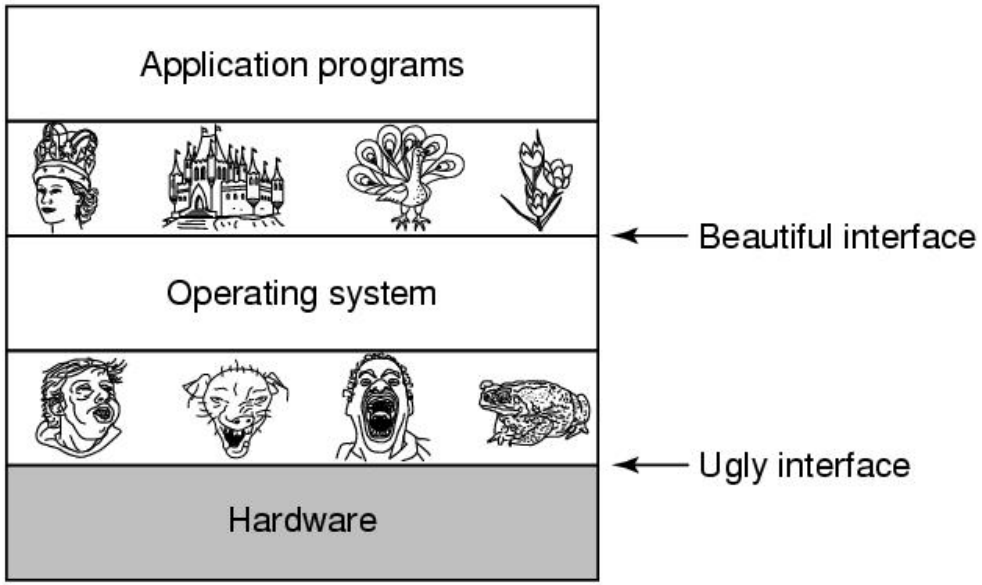


- A software layer between the hardware and the application programs/users which provides a **virtualization** interface.
- A resource manager that allows **concurrent** programs/users to share the hardware resources
- And it ensures information stored **persistently** in the computer.

Three themes of operating systems: virtualization, concurrency and persistence.

Theme 1: Virtualization

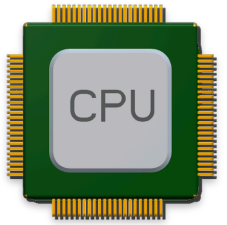
- Making a physical resource look like something else (virtual).
- Why virtualize?
  - To make the computer easier to use and program.



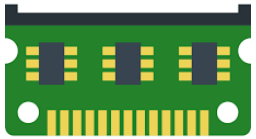

```
fprintf(fd, "%d", data);
        ↓
write(fd, buffer, count) ;
        ↓
file->f_op->write(file, buf,
count, pos);
        ↓
```

Examples

- Make one physical CPU look like multiple virtual CPUs
  - One or more virtual CPUs per process



- Make physical memory (RAM) and look like multiple virtual memory spaces
  - One or more virtual memory spaces per process



- Make physical disk look like a file system
  - Physical disk = raw bytes.
  - File system = user's view of data on disk. It is an extended machine

Virtualization example – Virtualizing the CPU

- cpu.c (simple code that loops and prints)

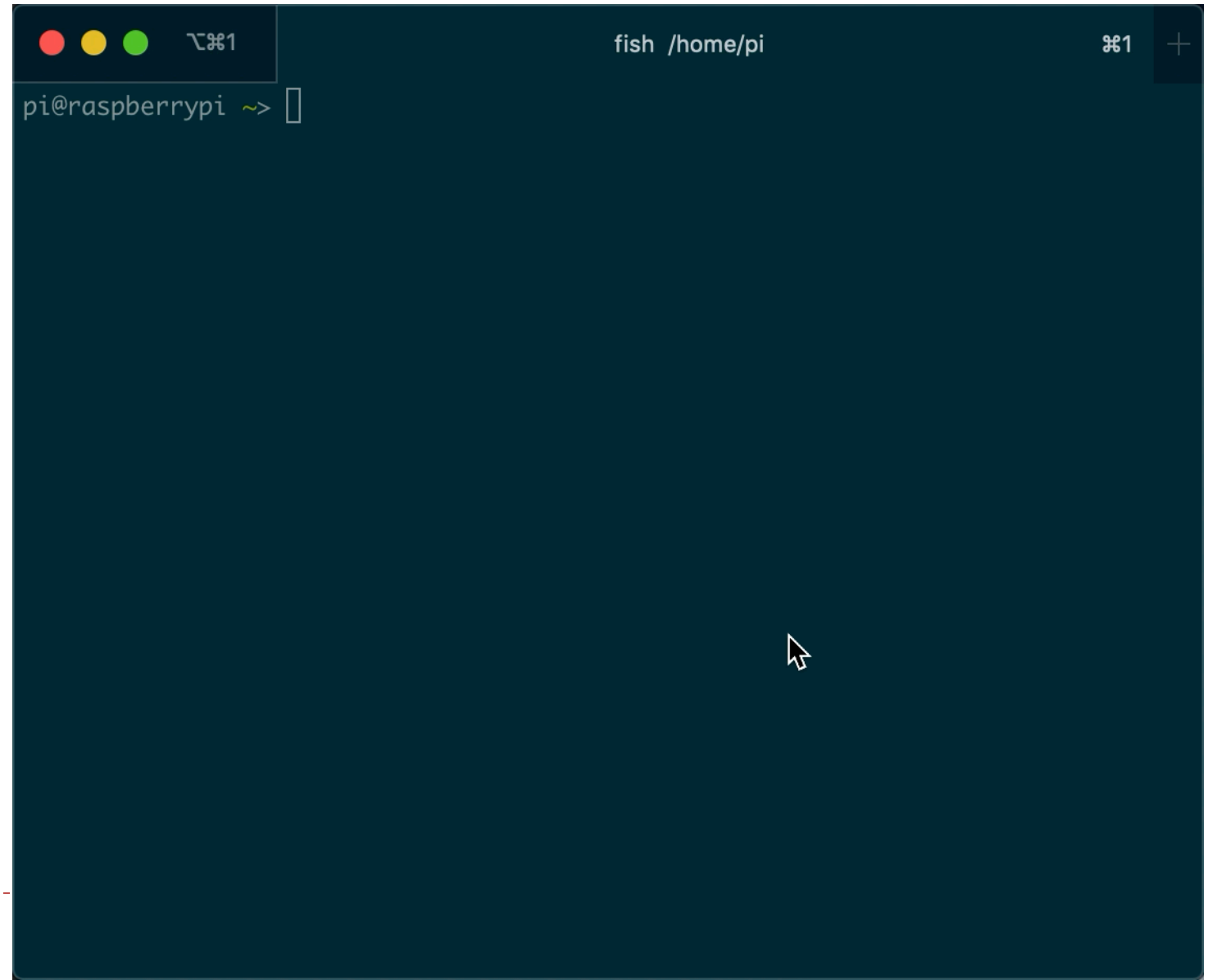
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

input

Infinite loop

print

Virtualization example – Virtualizing the CPU



Virtualization example – Virtualizing the CPU

- cpu.c (simple code that loops and prints)

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Run one instance of "cpu" program on a single core CPU.
The CPU runs the program and outputs to the terminal the messages that the program wants to print.

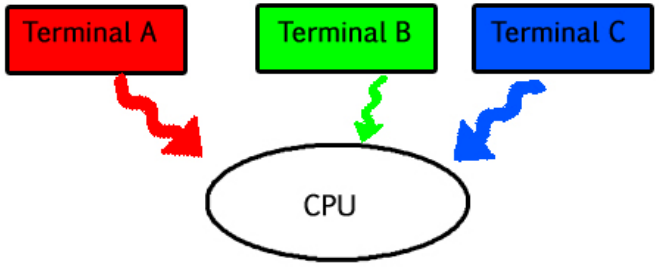
```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
```

While loop?

Run many instances of "cpu" program on a single core CPU. Even though we have only one processor, somehow all four of these programs seem to be running at the same time! How? The operating system, with some help from the hardware, turns (or virtualizes) a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once → virtualizing the CPU.

Virtualization example – Virtualizing the CPU

- cpu.c (simple code that loops and prints)



Run many instances of "cpu" program on a single core CPU. Even though we have only one processor, somehow all four of these programs seem to be running at the same time! How? The operating system, with some help from the hardware, turns (or virtualizes) a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once → virtualizing the CPU.

Virtualization example – Virtualizing the Memory

- mem.c (print pid and the address held in p)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                    // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);                             // a2
    *p = 0;                                          // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);        // a4
    }
    return 0;
}
```

What is the variable p stored?

the address held in p

the value held in p

Virtualization example – Virtualizing the Memory

Run a single instance of the "mem" program.

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

Run two instances of the "mem" program.

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

For the two instances case, each running program has allocated memory at the same address (00x200000), and yet each seems to be updating the value at 00x200000 **independently**! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs.

Virtualization example – Virtualizing the Memory

Run a single instance of the "mem" program.

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

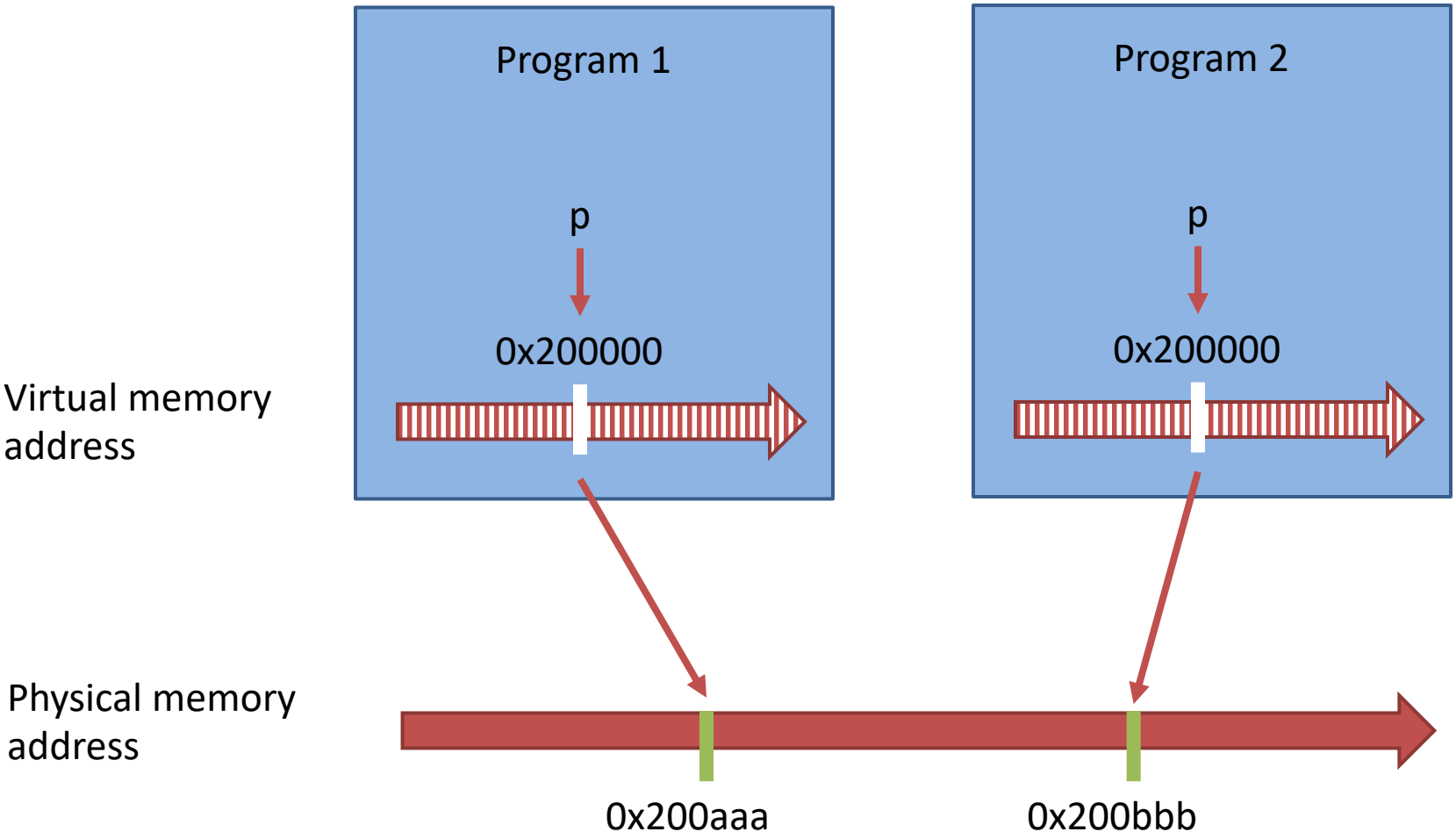
Run two instances of the "mem" program.

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Virtualizing memory:

- Each process accesses its own private virtual memory address space (sometimes just called its address space), which the OS somehow maps onto the physical memory of the machine.
- A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself.

Virtualization example – Virtualizing the Memory



Theme 2: concurrency

- Virtualization allows multiple concurrent programs to share the virtualized hardware resources, which brings out another class of topics: concurrency.
- Examples
  - One physical CPU runs many processes
  - One process runs many threads
  - One OS juggles process execution, system calls, interrupts, exceptions, CPU scheduling, memory management, etc.
- There's a LOT of concurrency in modern computer systems.
- And it's the source of most of the system complexity.

Concurrency Example

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Expected output?

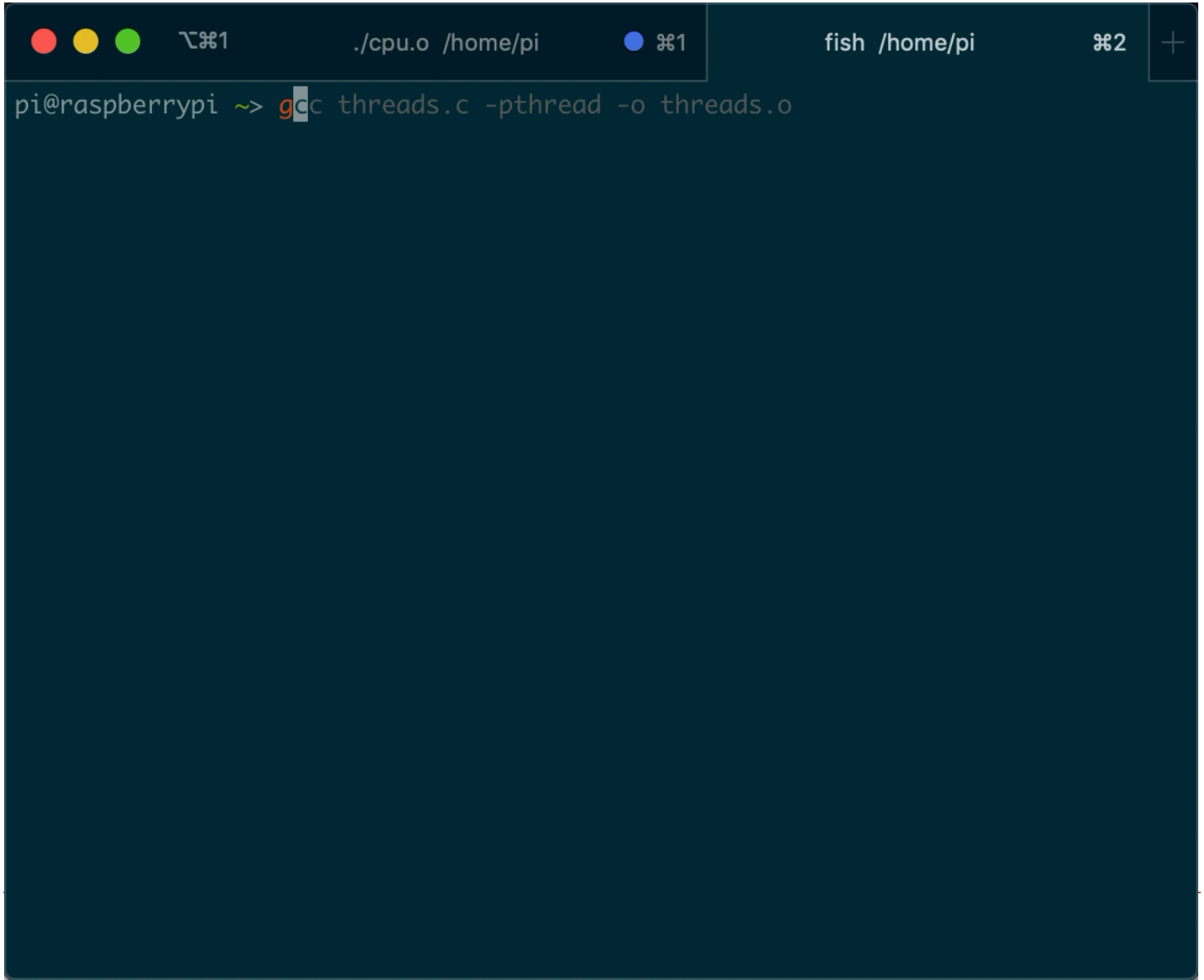
```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

Counter value: before

Two threads increase a counter

Counter value: after

Concurrency Example



Concurrency Example

Reality?

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

A key part of the program above, where the shared counter is incremented, takes three instructions:

- one to load the value of the counter from memory into a register,
- one to increment it, and
- one to store it back into memory.

Because these three instructions do not execute **atomically** (all at once), strange things can happen.

Theme 3: Persistence

- Storing data “forever”.
  - On hard disks, SSDs, CDs, floppy disks, tapes, phono discs, paper!
- But its not enough to just store raw bytes
- Users want to
  - Organize data (via file systems)
  - Share data (via network or cloud)
  - Access data easily (via user interface)
  - Protect data from being stolen (via security)

Above example code

- Code example:

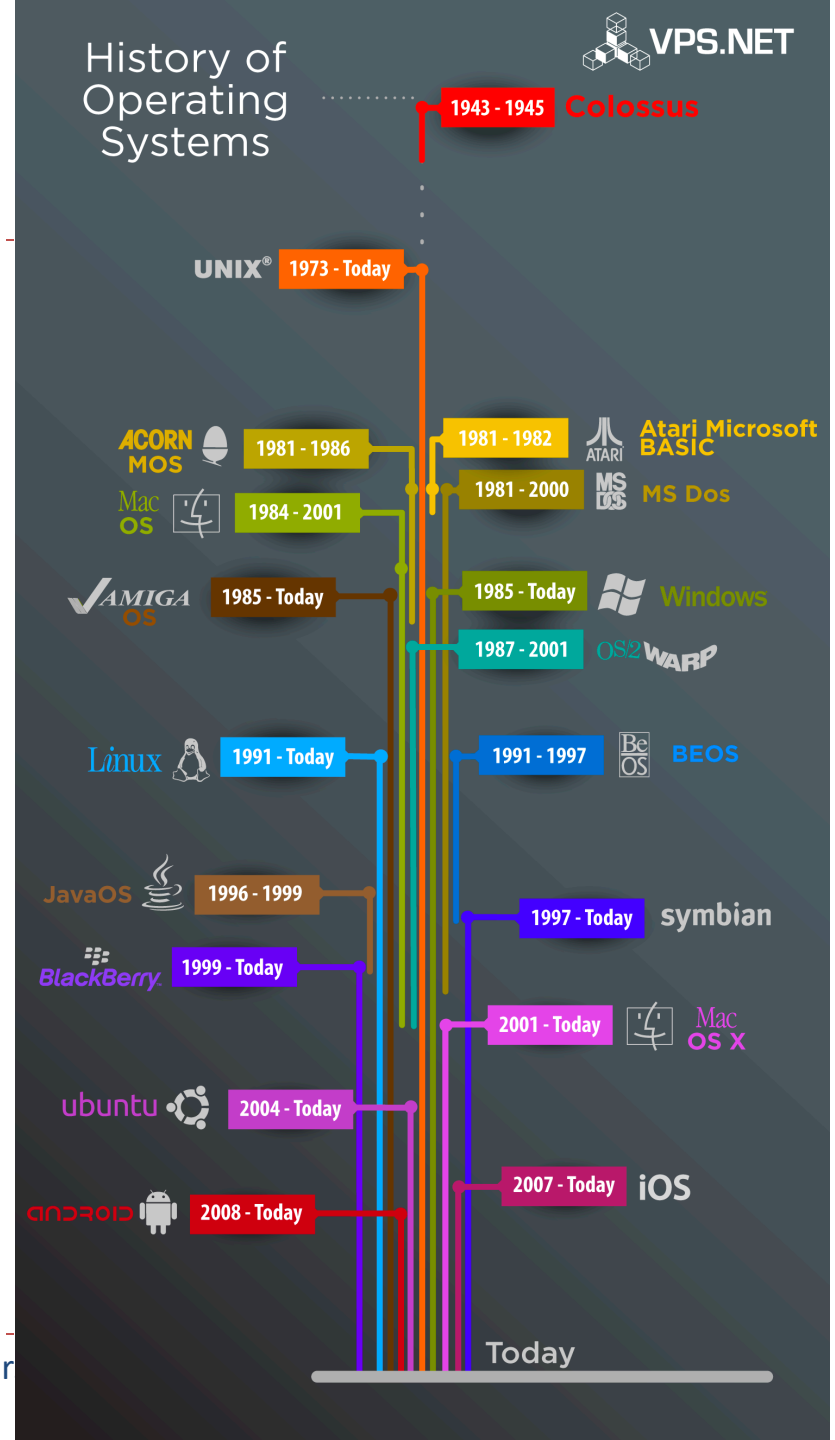
https://github.com/remzi-arpacidusseau/ostep-code/tree/master/intro

Outline

- Hardware Review
- What is an OS?
  - Virtualization
  - Concurrency
  - Persistence
- History of OS
  - Monolithic
  - Microkernel
  - Hybrid

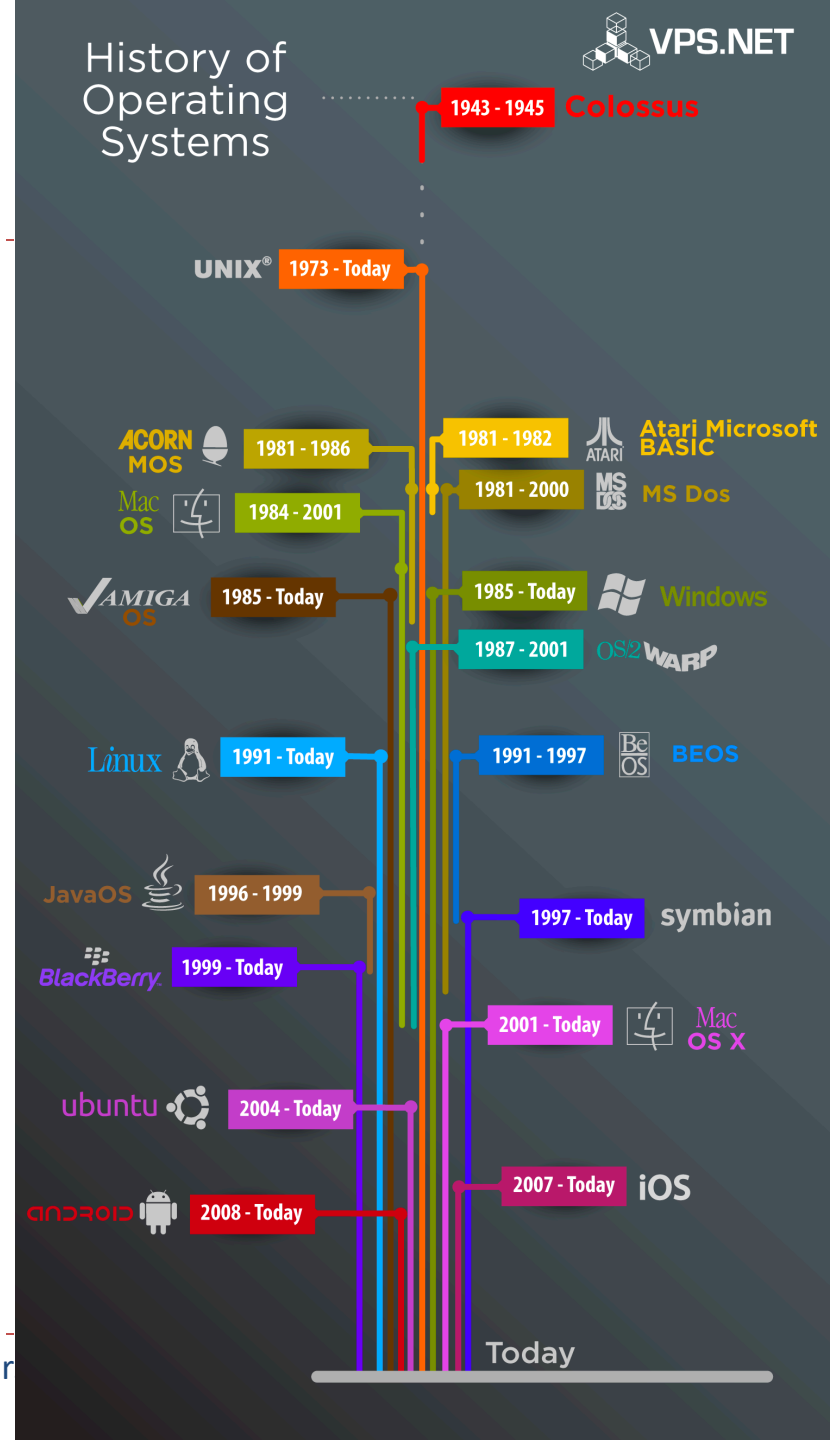
History of OS

- 1950s and 1960s: Early operating systems were simple batch processing systems
  - Users provided their own “OS” as libraries.
- 1960s and 1970s: Multi-programming on mainframes
  - Concurrency, memory protection, Kernel mode, system calls, hardware privilege levels, trap handling
  - Earliest Multics hardware and OS on IBM mainframes
  - Which led to the first UNIX OS which pioneered file systems, shell, pipes, and the C language.

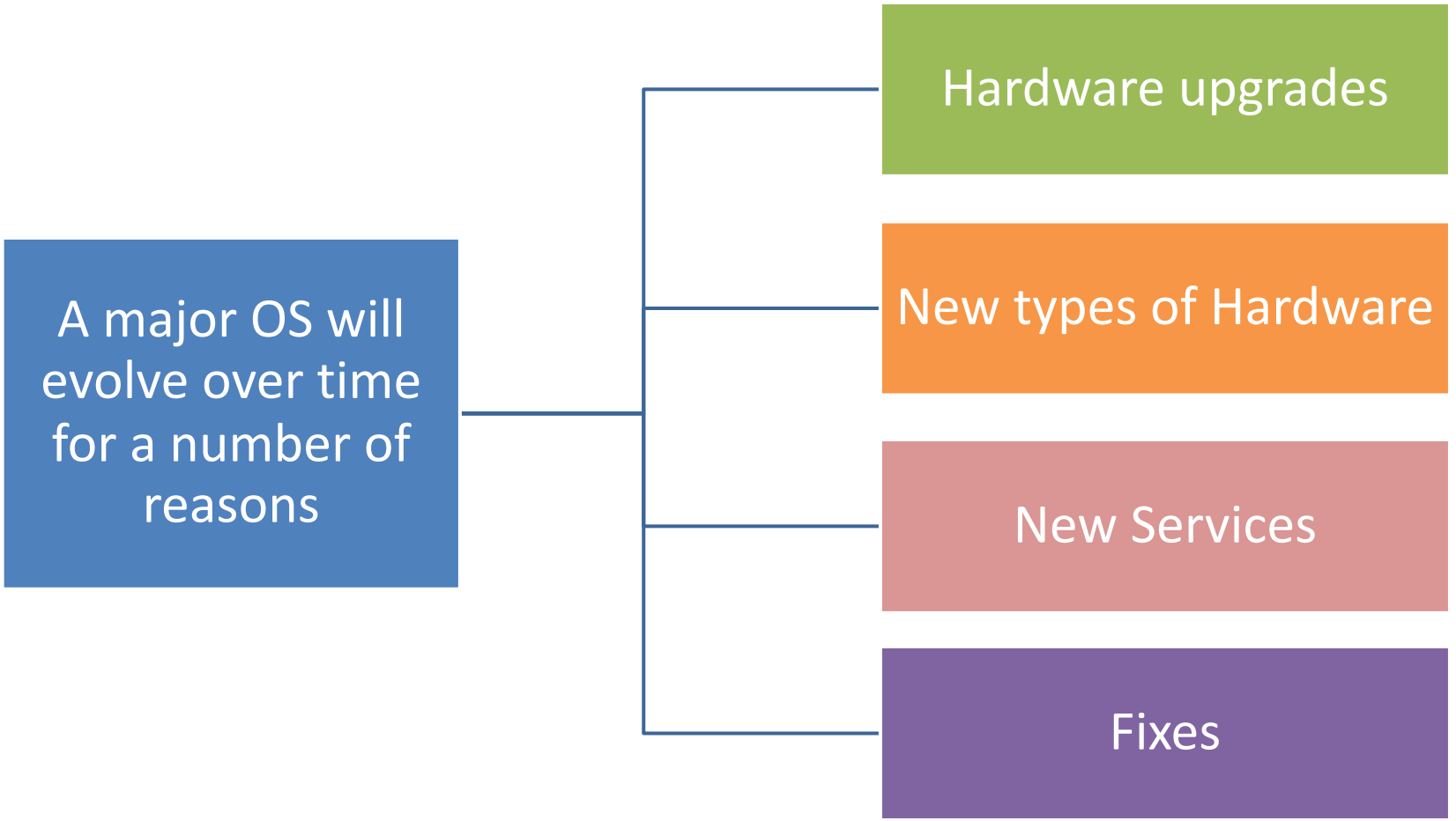


History of OS

- 1980s: Personal computing era
  - MacOS, IBM PC and its DOS, Windows, and so forth
  - Each big company had its own version of OS (IBM, Apple, HP, SUN, SGI, NCR, AT&T….)
- 1990s: Then came BSD and Linux
  - Open source.
  - Led the way to modern OSes and cloud platforms
  - wider adoption of threads and parallelism
- 2000 and beyond: Mobile device OS and hypervisors
  - Android, iOS, VMWare ESX, Xen, Linux/KVM etc.



Evolution of Operating Systems



User space vs. Kernel space

- **Kernel space** is strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers.
- **User space** is the area where application software and some drivers execute.

<table>
<tr><td rowspan="3">User mode</td><td>User applications</td><td colspan="4">For example, bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, etc.</td></tr>
<tr><td>Low-level system components:</td><td>System daemons:
systemd, runit, logind, networkd, PulseAudio, ...</td><td>Windowing system:
X11, Wayland, SurfaceFlinger (Android)</td><td>Other libraries:
GTK+, Qt, EFL, SDL, SFML, FLTK, GNUstep, etc.</td><td>Graphics:
Mesa, AMD Catalyst, ...</td></tr>
<tr><td>C standard library</td><td colspan="4"><code>open()</code>, <code>exec()</code>, <code>sbrk()</code>, <code>socket()</code>, <code>fopen()</code>, <code>calloc()</code>, ... (up to 2000 subroutines)
glibc aims to be POSIX/SUS-compatible, musl and uClibc target embedded systems, bionic written for Android, etc.</td></tr>
<tr><td rowspan="3">Kernel mode</td><td rowspan="3">Linux kernel</td><td colspan="4"><code>stat</code>, <code>splice</code>, <code>dup</code>, <code>read</code>, <code>open</code>, <code>ioctl</code>, <code>write</code>, <code>mmap</code>, <code>close</code>, <code>exit</code>, etc. (about 380 system calls)
The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)</td></tr>
<tr><td>Process scheduling subsystem</td><td>IPC subsystem</td><td>Memory management subsystem</td><td>Virtual files subsystem</td><td>Network subsystem</td></tr>
<tr><td colspan="4">Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter
Linux Security Modules: SELinux, TOMOYO, AppArmor, Smack</td></tr>
<tr><td colspan="6">Hardware (CPU, main memory, data storage devices, etc.)</td></tr>
</table>

User mode vs. Kernel mode

- What is the difference between kernel and user mode?
  - Most modern CPUs provide two modes of execution (i.e., supported directly by the hardware): kernel mode and user mode.
  - The CPU can execute every instruction in its instruction set and use every feature of the hardware when executing in kernel mode.
  - However, it can execute only a subset of instructions and use only subset of features when executing in the user mode.

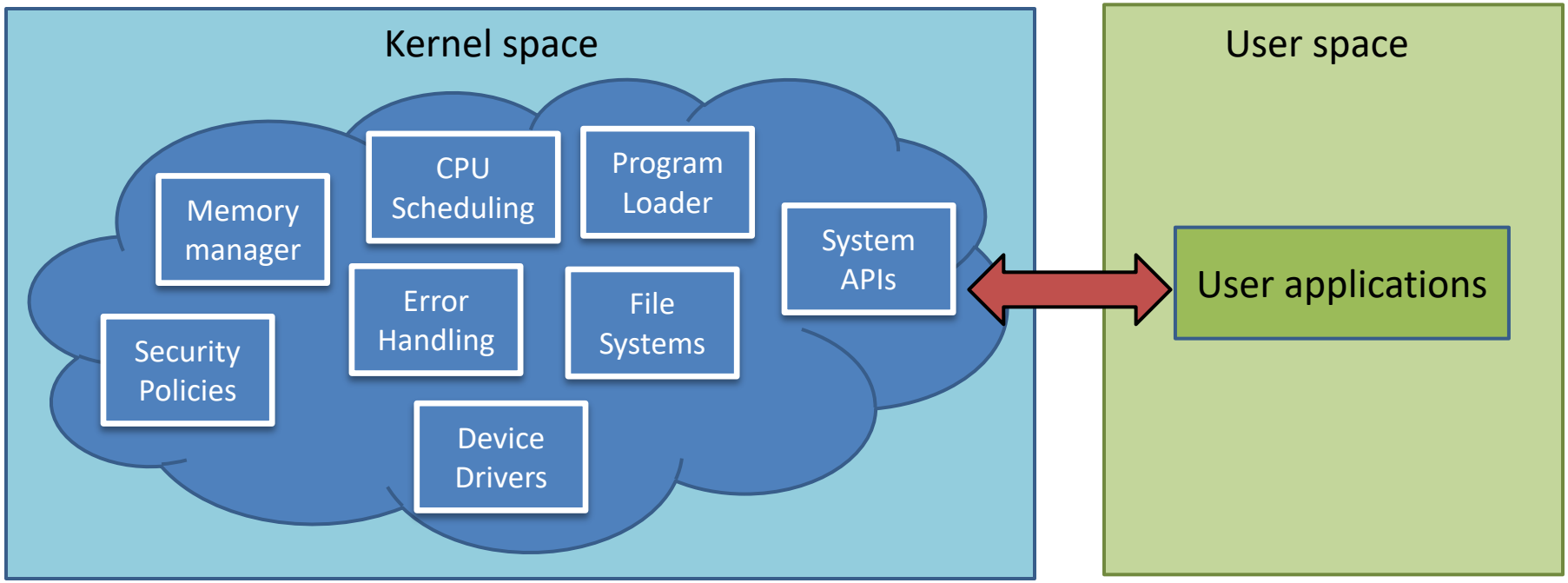
- What is the purpose of having these two modes?
  - Purpose: protection – to protect critical resources (e.g., privileged instructions, memory, I/O devices) from being misused by user programs.

Architecting Kernels -- Three basic approaches

- Monolithic kernels
  - All functionalities are compiled together
  - All code runs in privileged kernel-space
- Microkernels
  - Only essential functionalities are compiled into the kernel
  - All other functionalities run in unprivileged user space
- Hybrid kernels
  - Most functionalities are compiled into the kernel
  - Some functions are loaded dynamically
  - Typically, all functionality runs in kernel-space

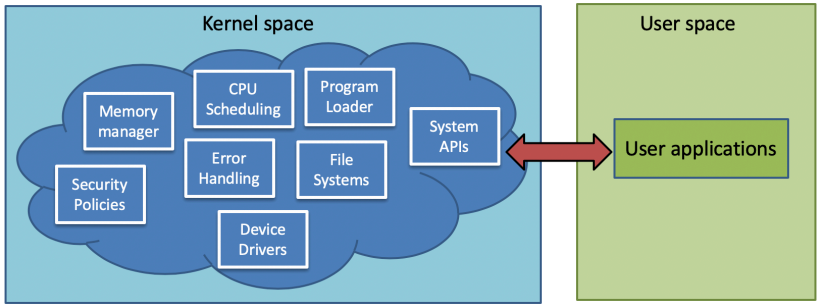
Monolithic Kernel

- Monolithic kernels
  - All functionalities are compiled together
  - All code runs in privileged kernel-space



Pros/Cons of Monolithic Kernels

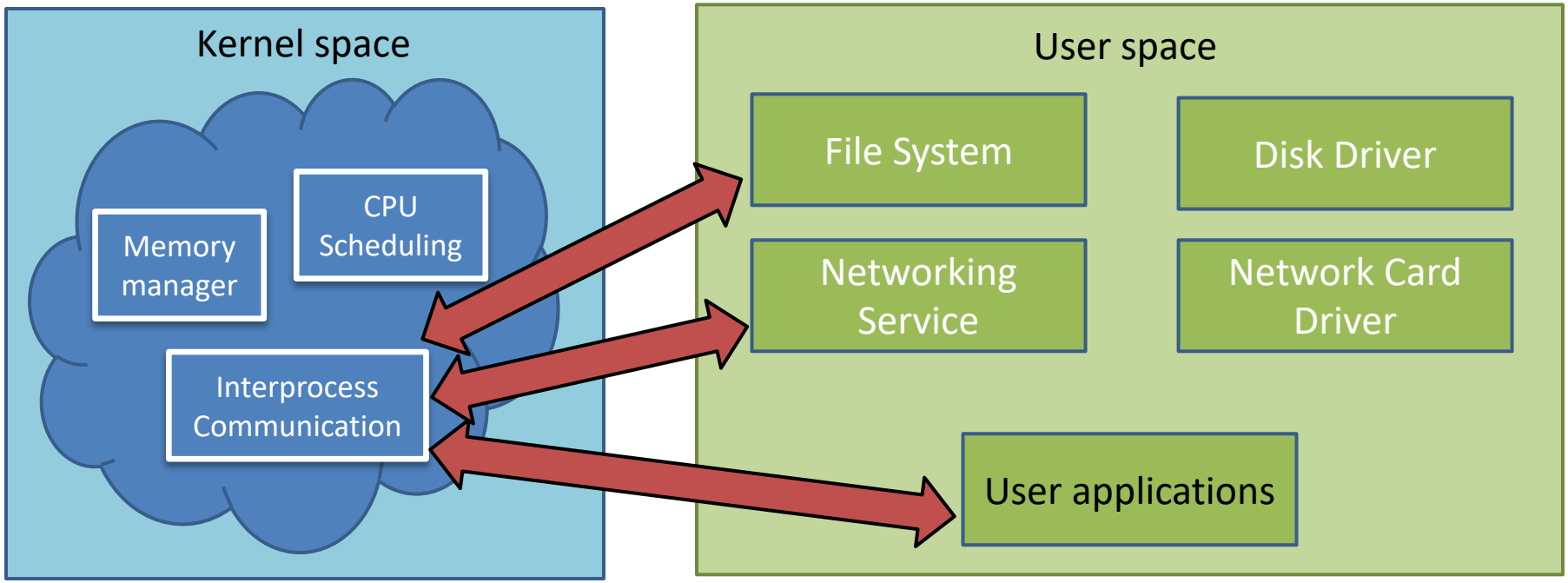
- Advantages
  - Single code base eases kernel development, simple



  - Robust APIs for application developers, stable APIs
  - No need to find separate device drivers, easy for apps
  - Fast performance due to tight coupling
- Disadvantages
  - Large code base, hard to check for correctness
  - Bugs crash the entire kernel (and thus, the machine)

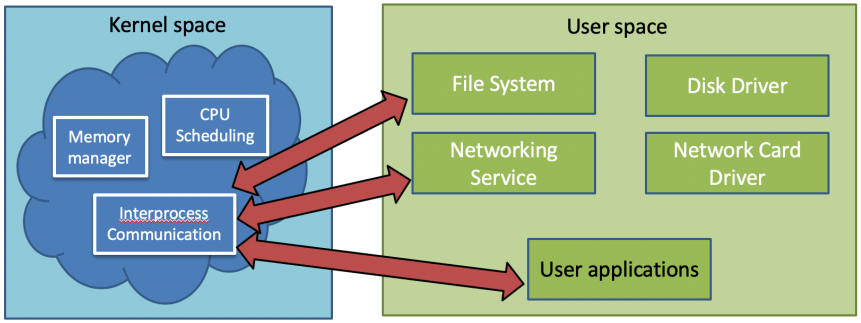
Microkernel

- Microkernels
  - Only essential functionalities are compiled into the kernel
  - All other functionalities run in unprivileged user space



Pros/Cons of Microkernels

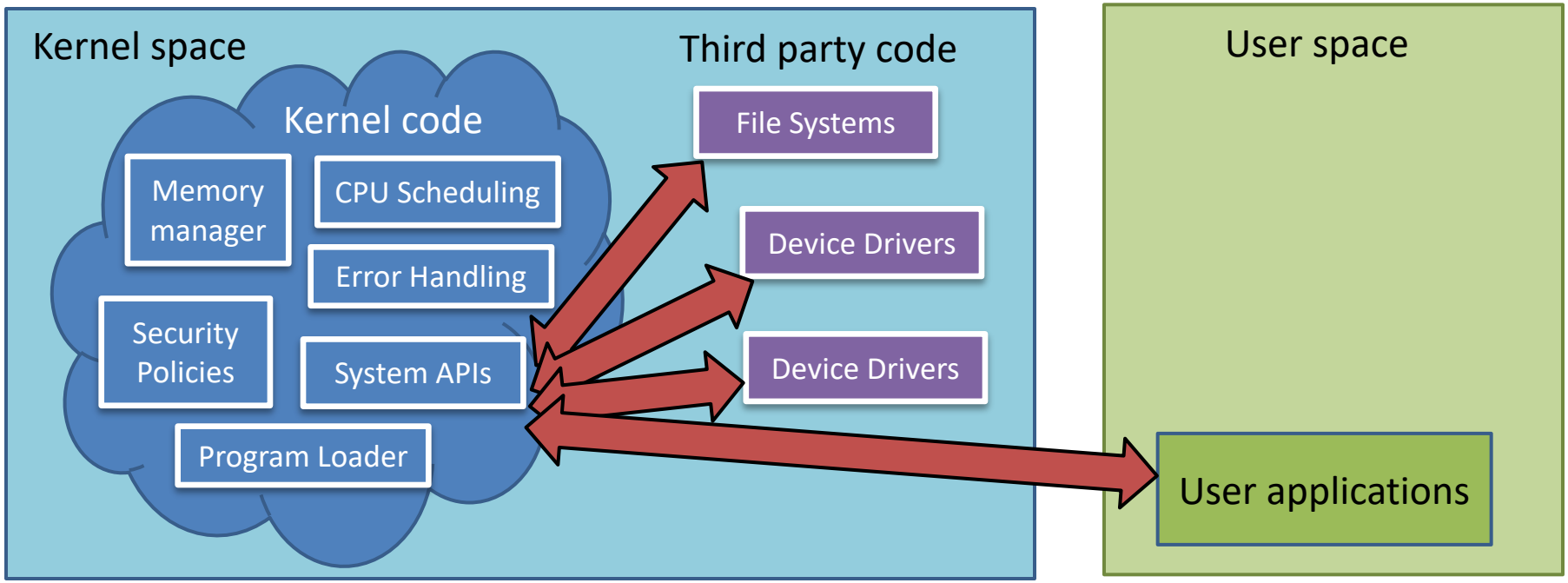
- Advantages
  - Small code base, easy to check for correctness
    - Excellent for high-security systems
  - Extremely modular and configurable
    - Choose only the pieces you need for embedded systems
    - Easy to add new functionality (e.g. a new file system)
  - Services may crash, but the system will remain stable



- Disadvantages
  - Performance is slower: many context switches
  - No stable APIs, more difficult to write applications

Hybrid Kernel

- Hybrid kernels
  - Most functionalities are compiled into the kernel
  - Some functions are loaded dynamically
  - Typically, all functionality runs in kernel-space



Examples

iOS

Research Kernels: Mach L4 GNU Hurd

Kernels for Embedded System: QNX

Monolithic Kernels:
Huge code base,
Many features

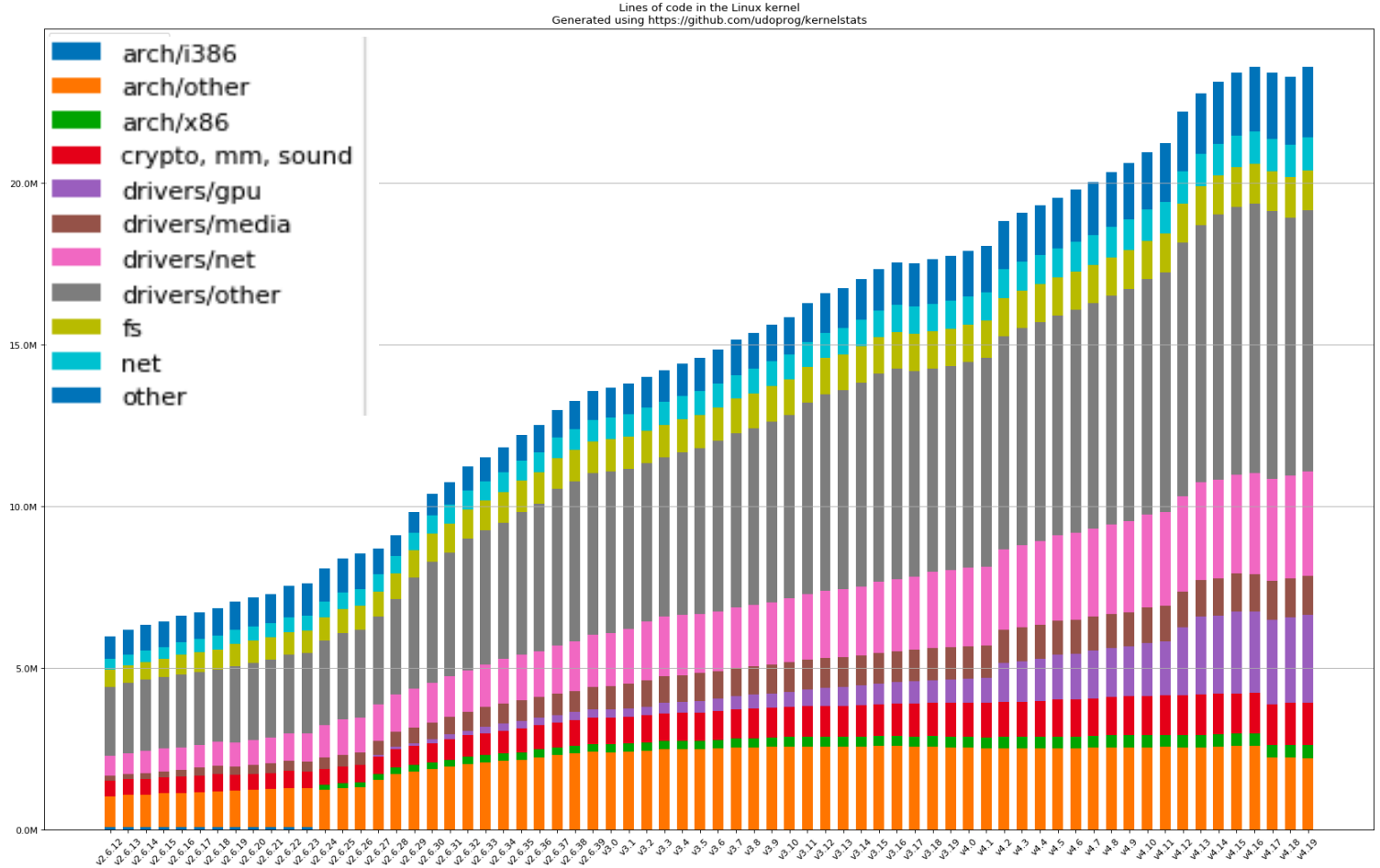
Hybrid Kernels:
Pretty large code base,
Some features delegated

Microkernels:
Small code base,
Few features

Examples

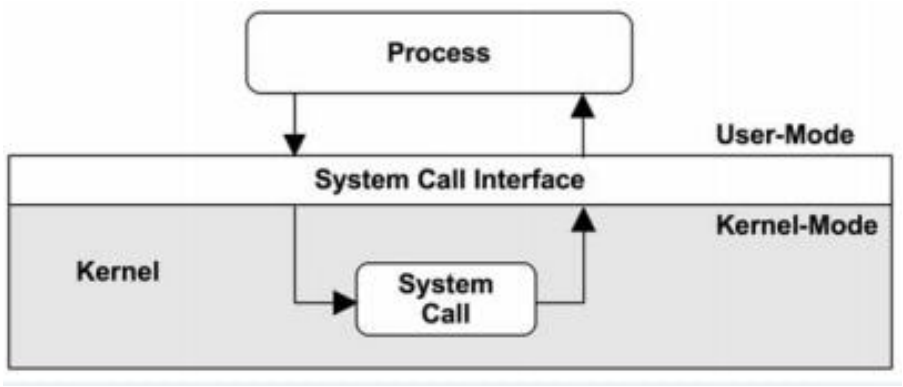
- The most complex software
  - ~ 20+ million lines of code in Linux

https://elixir.bootlin.com/linux/latest/source



User<-->Kernel: System Call

- A system call is a way for programs to interact with the operating system
- A computer program makes a system call when it makes a request to the operating system's kernel
- System call provides the services of the operating system to the user programs via Application Program Interface(API)



System call example:
Scheduling,
Memory allocation,
Read/write,
…

Linux Syscall Table

https://filippo.io/linux-syscall-table/

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	poll	sys_poll	fs/select.c
8	lseek	sys_lseek	fs/read_write.c
9	mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	mprotect	sys_mprotect	mm/mprotect.c
11	munmap	sys_munmap	mm/mmap.c
12	brk	sys_brk	mm/mmap.c

Conclusion

- Hardware Review
- What is an OS?
  - Virtualization
  - Concurrency
  - Persistence
- History of OS
  - Monolithic
  - Microkernel
  - Hybrid