CS 3502
Operating Systems

Operating Systems Overview

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

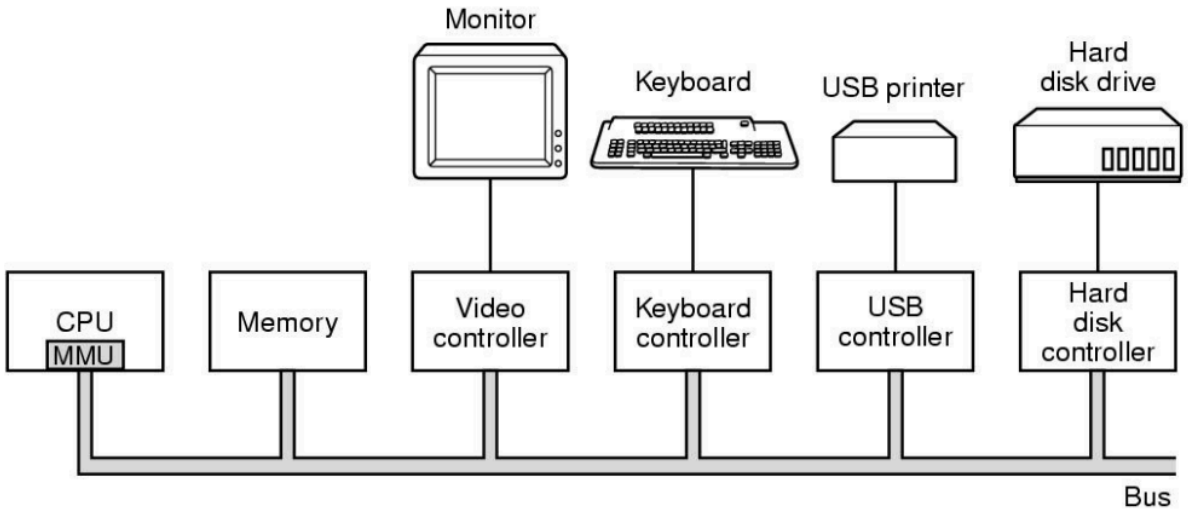
Outline

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

Computer Hardware Review

Computer Hardware Review

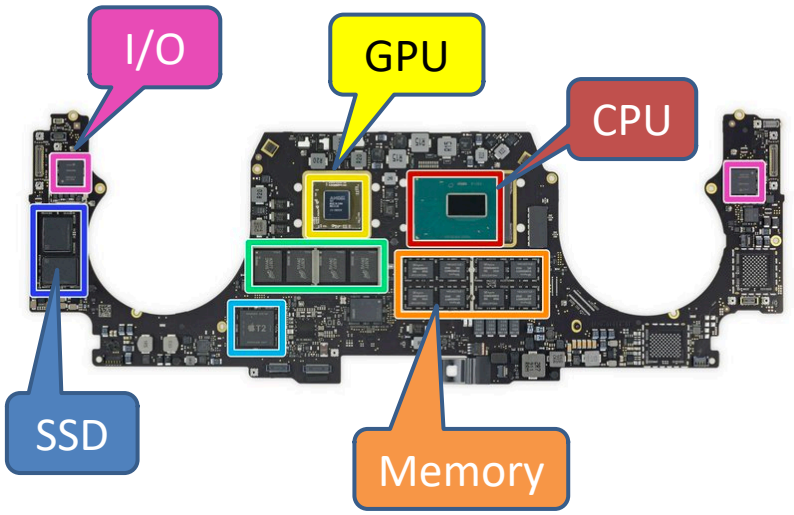
- Basic components of a simple personal computer



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

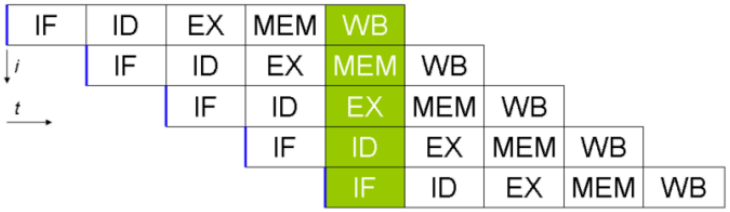
Computer Hardware Review

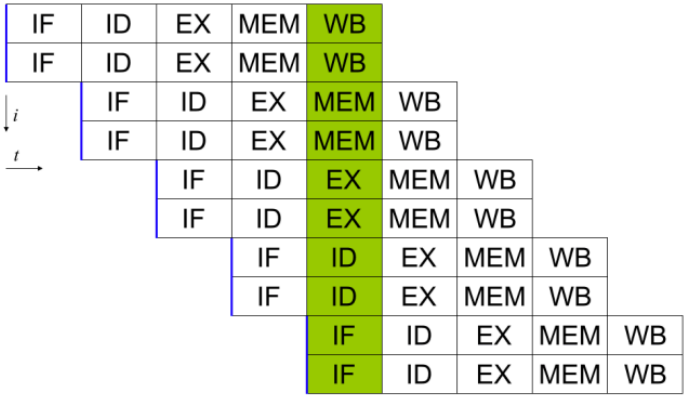
Computer Hardware Review

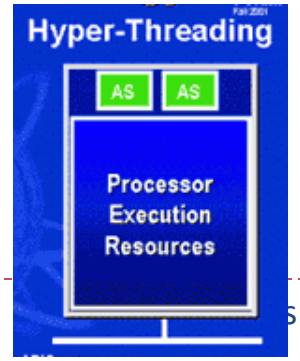


Central Processing Unit (CPU)

- 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

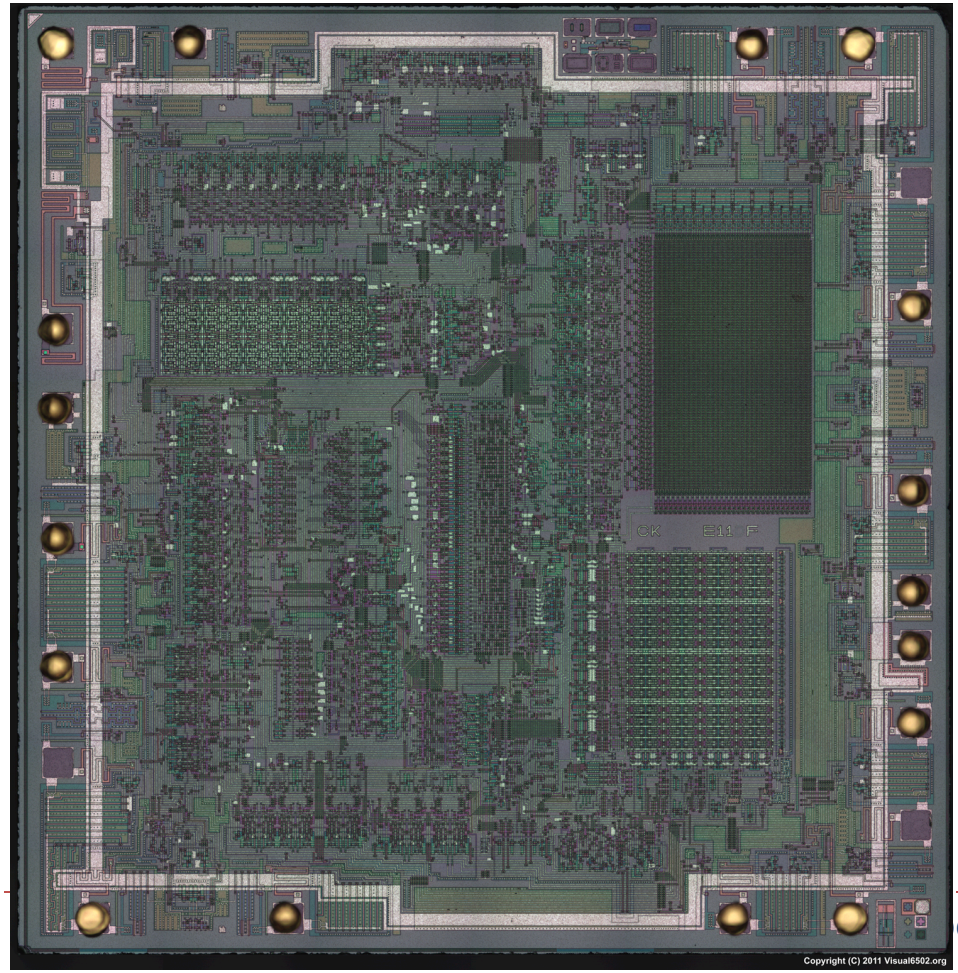






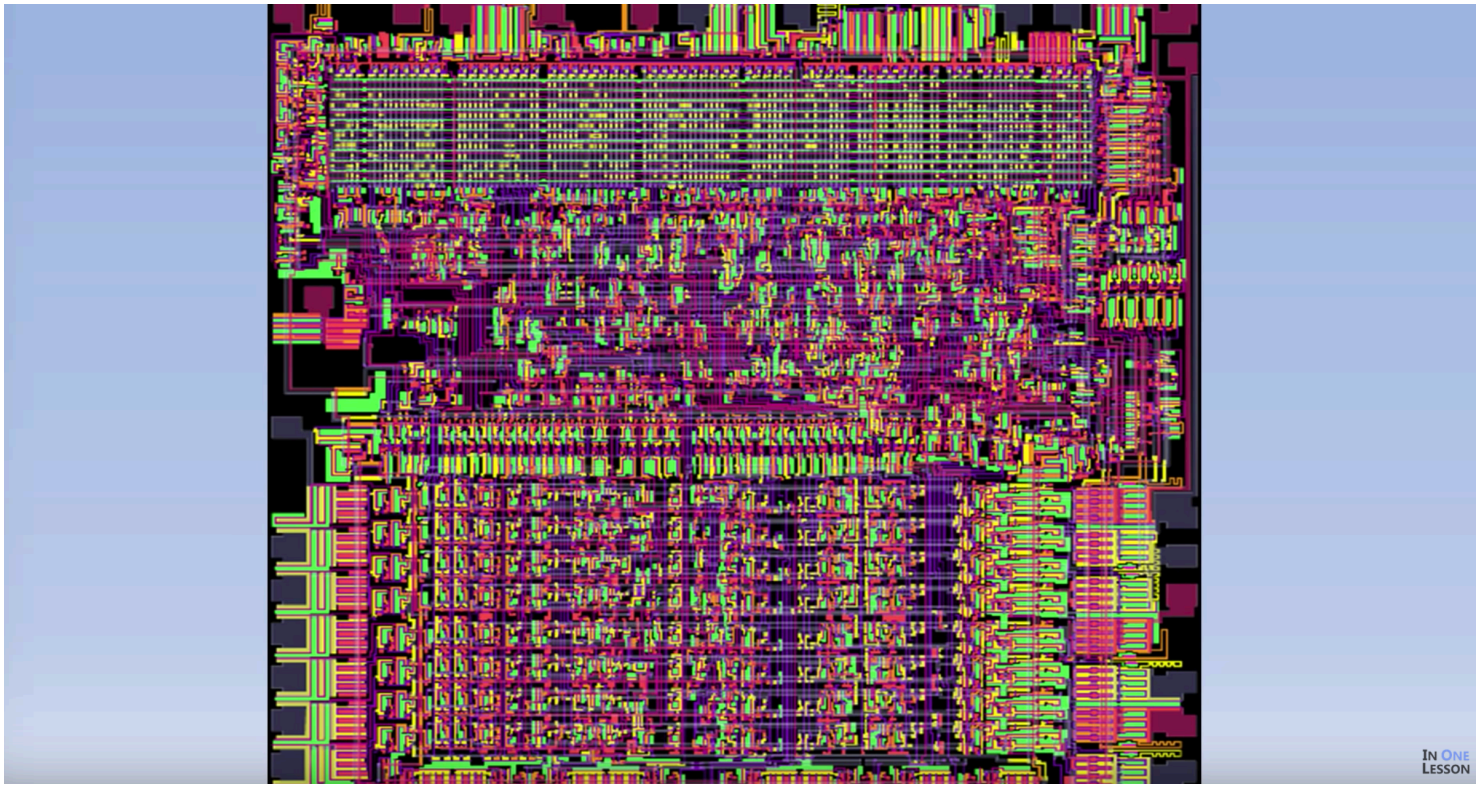
What's inside of CPU?

- http://www.visual6502.org/

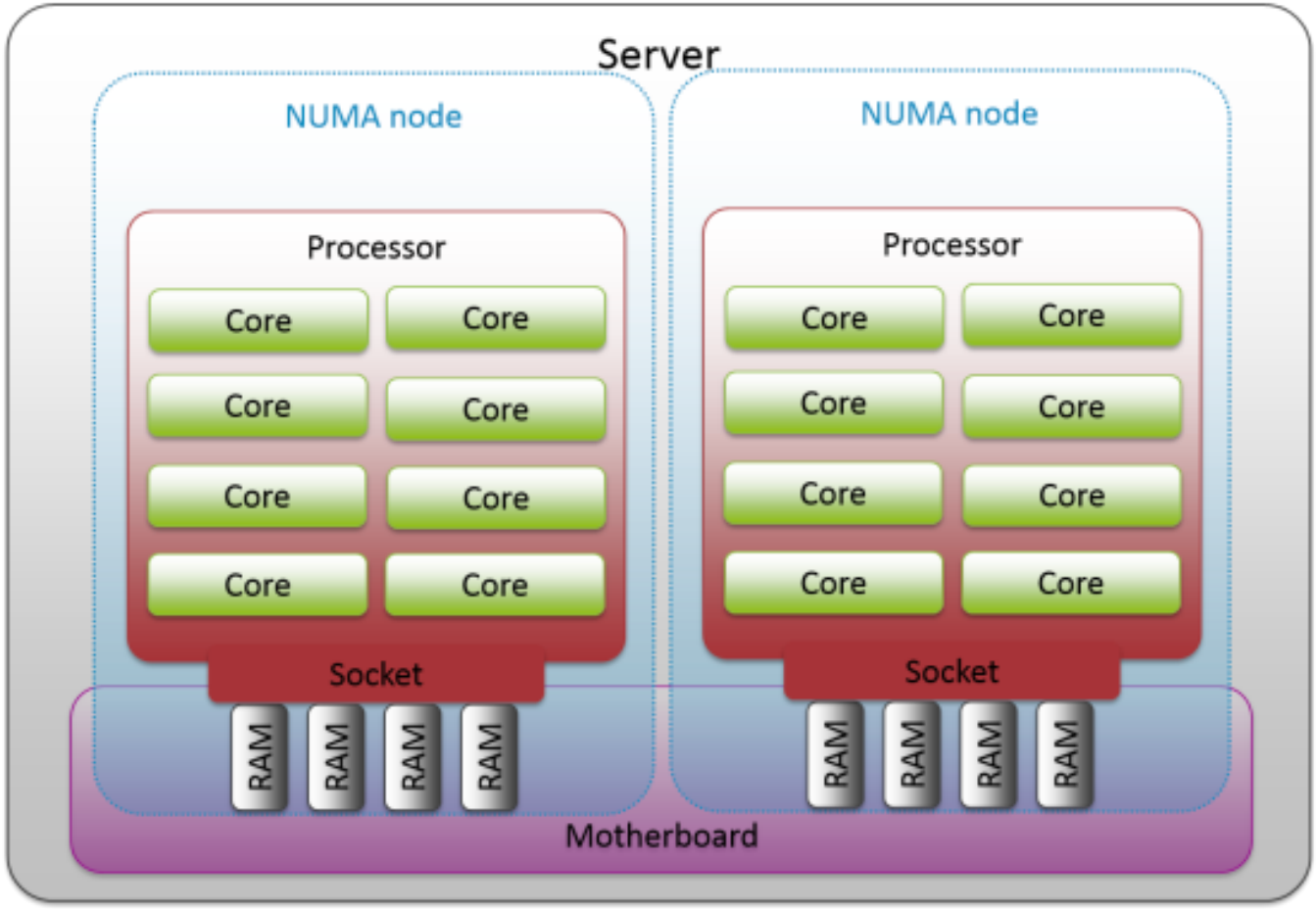


How CPU works?

- https://youtu.be/cNN_tTXABUA?t=494



NUMA node vs Socket vs Core relationship



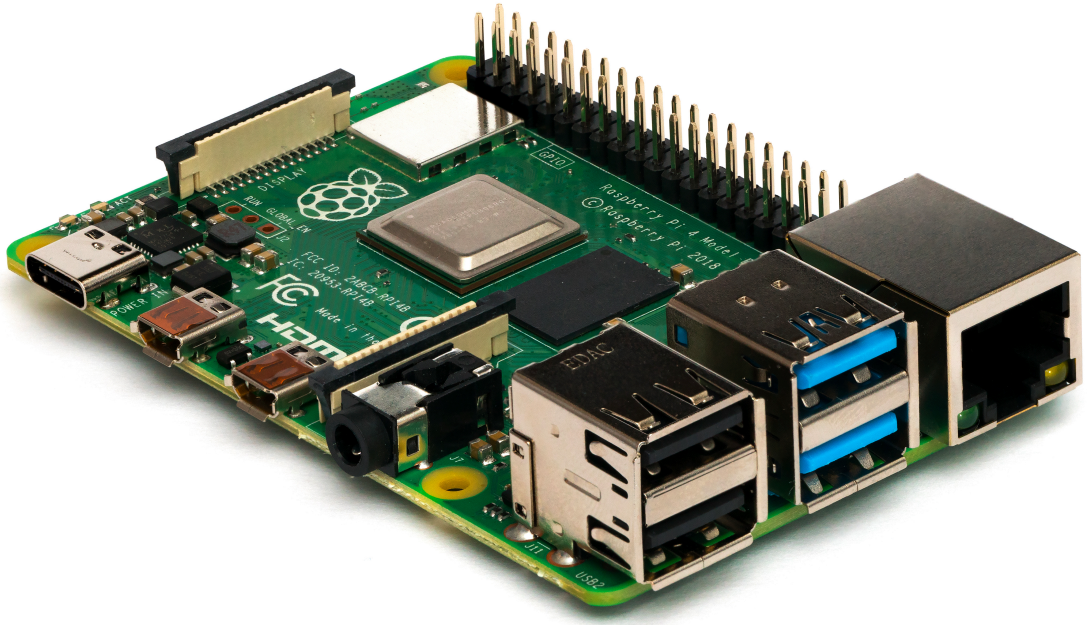
CPU information

- lscpu


```
administrator@ubuntuvm-1604 ~> lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
Stepping:              1
CPU MHz:               2199.998
BogoMIPS:              4399.99
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              51200K
NUMA node0 CPU(s):     0,1
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep m
all nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology ts
id sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave av
lt invpcid_single pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1
 flush_l1d arch_capabilities
```

CPU information

- lscpu



```
pi@raspberrypi:~ $ lscpu
Architecture:            armv7l
Byte Order:              Little Endian
CPU(s):                  4
On-line CPU(s) list:     0-3
Thread(s) per core:      1
Core(s) per socket:      4
Socket(s):               1
Vendor ID:               ARM
Model:                   3
Model name:              Cortex-A72
Stepping:                r0p3
CPU max MHz:             1500.0000
CPU min MHz:             600.0000
BogoMIPS:                108.00
Flags:                   half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
```

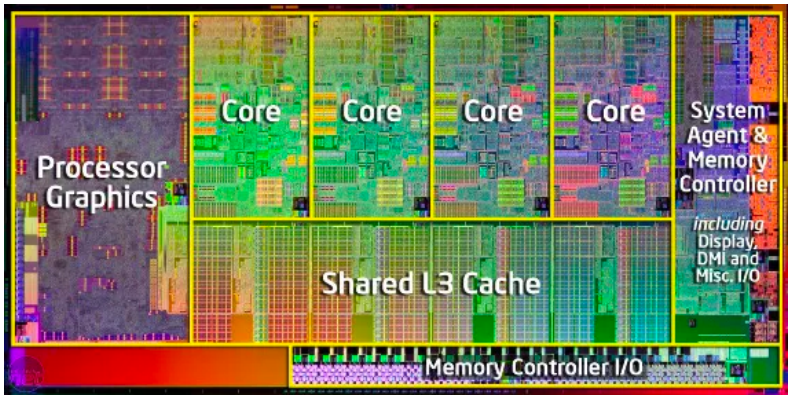
CPU information

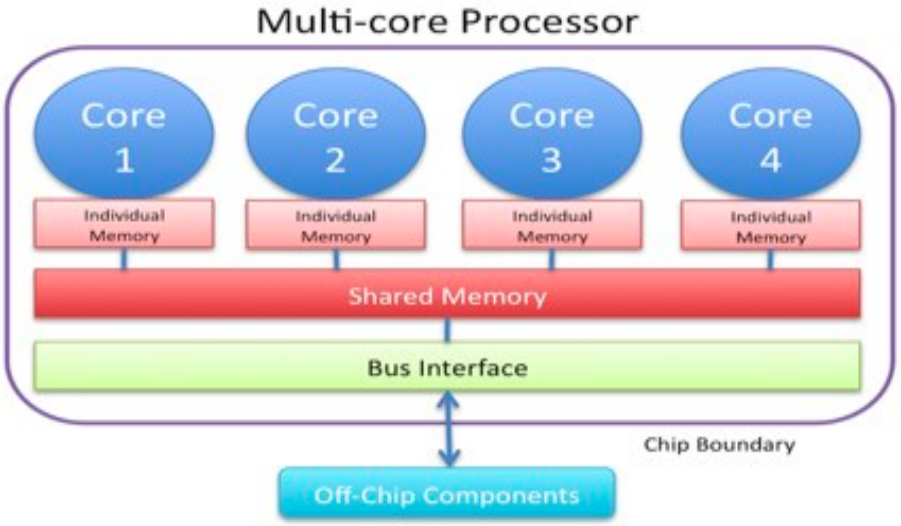
- $ cat /proc/cpuinfo

```
administrator@ubuntuvm-1604 ~> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
stepping        : 1
microcode       : 0xb000036
cpu MHz         : 2199.998
cache size      : 51200 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
x pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology t
e4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave a
vpcid_single pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1
h_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_stor
bogomips        : 4399.99
clflush size    : 64
cache_alignment : 64
address sizes   : 42 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
stepping        : 1
microcode       : 0xb000036
cpu MHz         : 2199.998
cache size      : 51200 KB
physical id     : 2
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 2
initial apicid  : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
x pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology
e4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
vpcid_single pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi
h_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_st
bogomips        : 4399.99
clflush size    : 64
cache_alignment : 64
address sizes   : 42 bits physical, 48 bits virtual
power management:
```

Multi-Core Processors (SMP)

- Multiple CPUs on a single chip

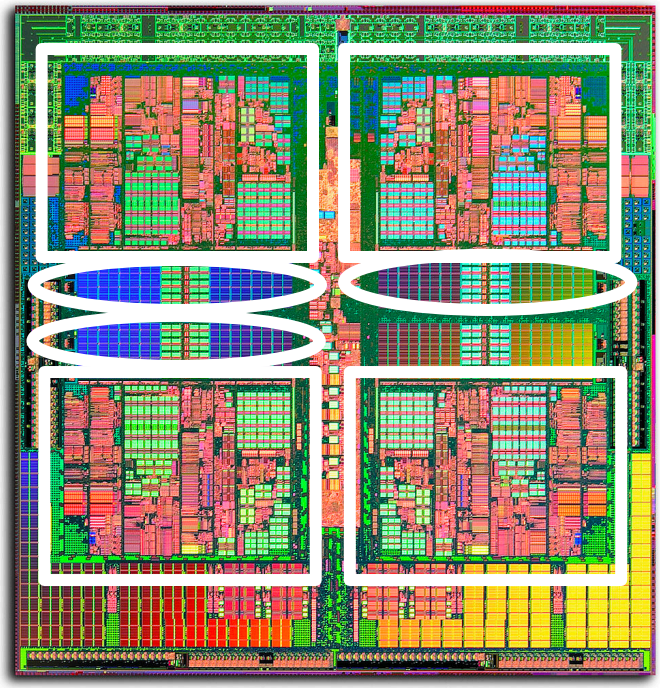


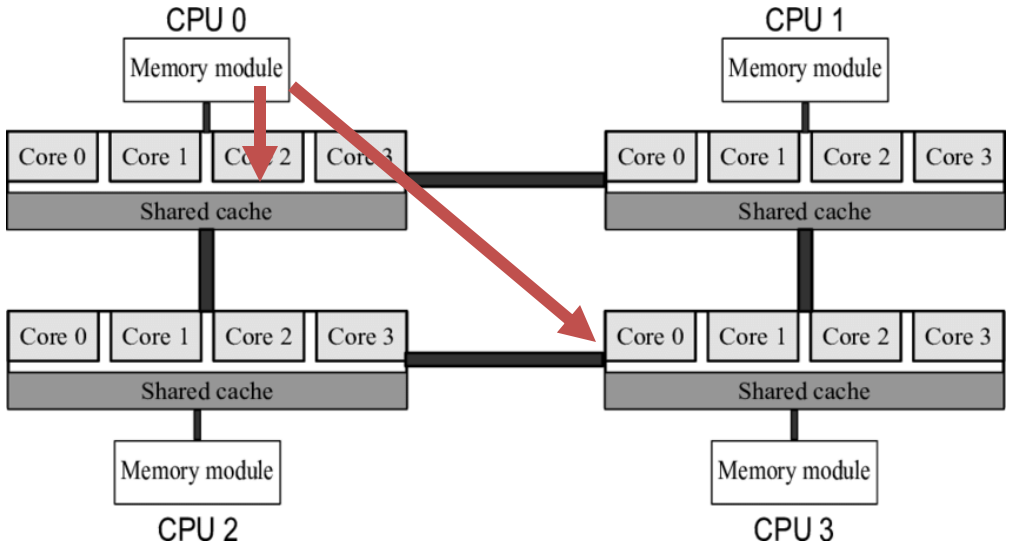


Symmetric multiprocessing (**SMP**)

Multi-Core Processors (NUMA)

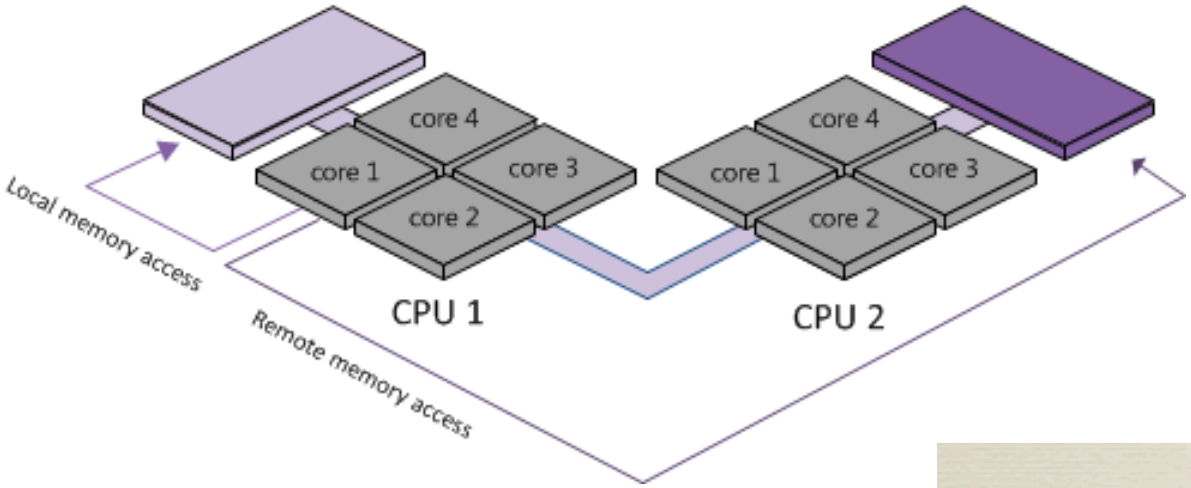
- Multiple CPUs on a single chip

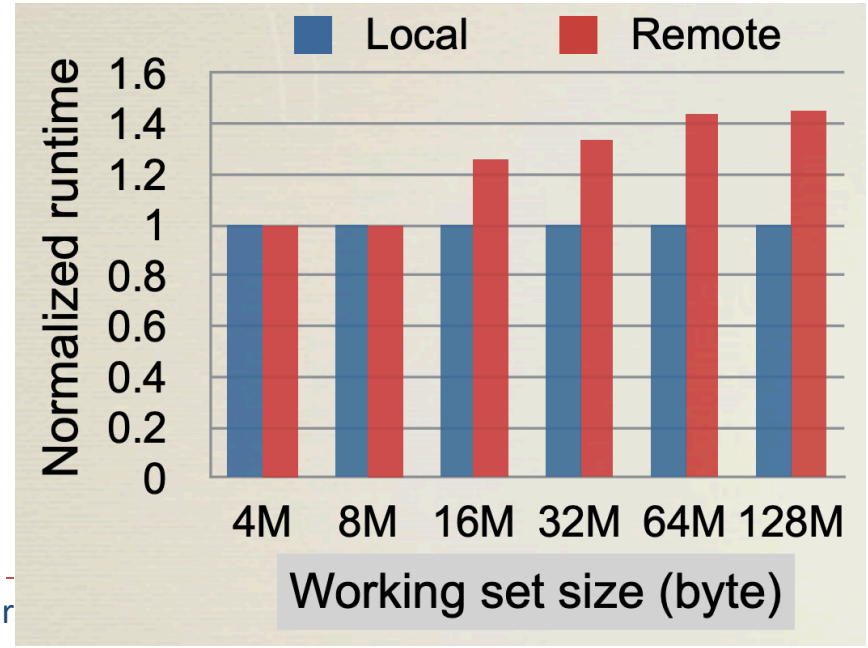




Non-uniform memory access (**NUMA**)

Multi-Core Processors (NUMA)





Check CPU topology

- $ likwid-topology –g

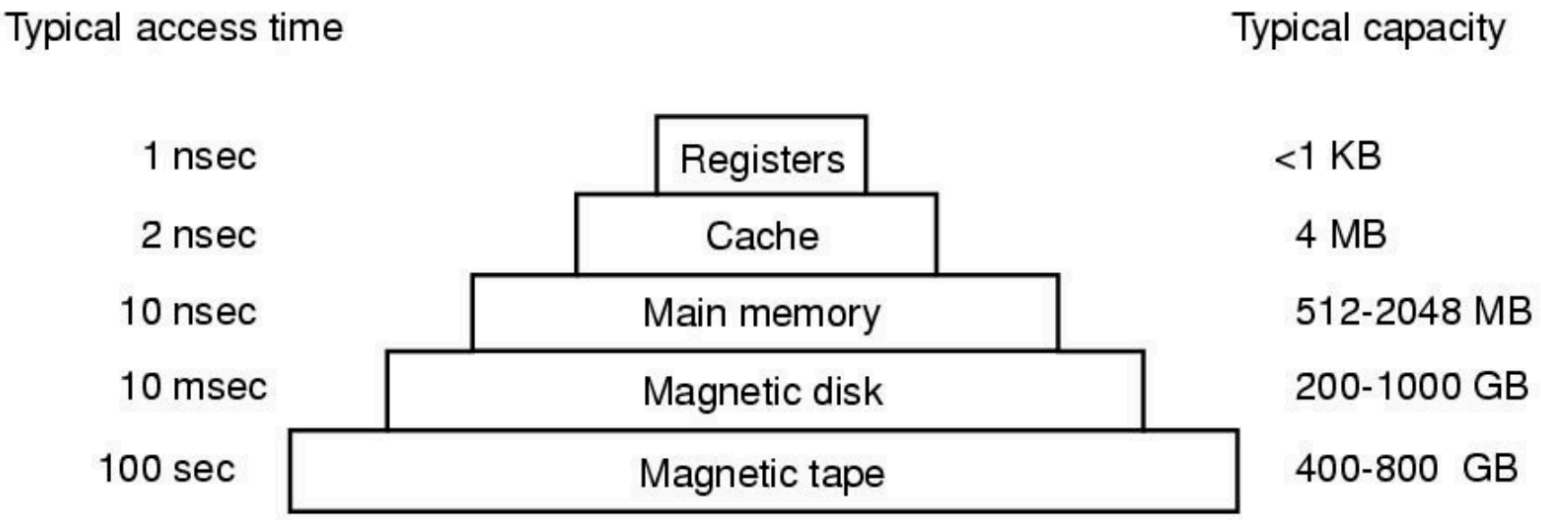
```
ksuo@ksuo-VirtualBox ~/likwid-5.0.0> likwid-topology -g
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz
CPU type:       Intel Coffeelake processor
CPU stepping:   13
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       1
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 1 2 3 )
--------------------------------------------------------------------------------
********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+-------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ |
| |    0   | |    1   | |    2   | |    3   | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| |  32 kB | |  32 kB | |  32 kB | |  32 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| | 256 kB | | 256 kB | | 256 kB | | 256 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| |  16 MB | |  16 MB | |  16 MB | |  16 MB | |
| +--------+ +--------+ +--------+ +--------+ |
+-------------------------------------------+
```

NUMA example:

https://github.com/RRZE-HPC/likwid/wiki/TutorialNUMA

Memory

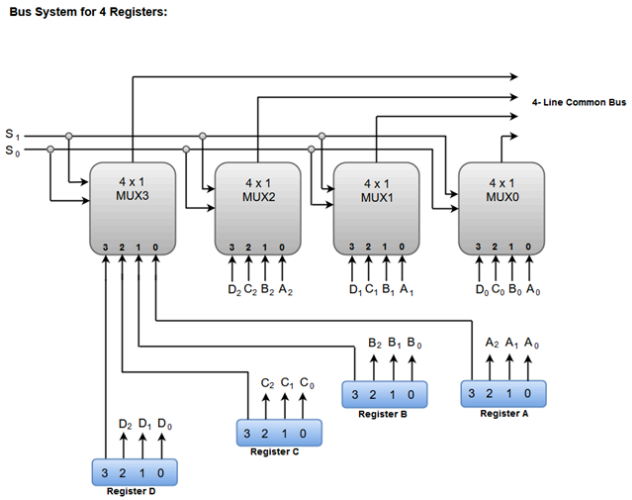
- A typical memory hierarchy

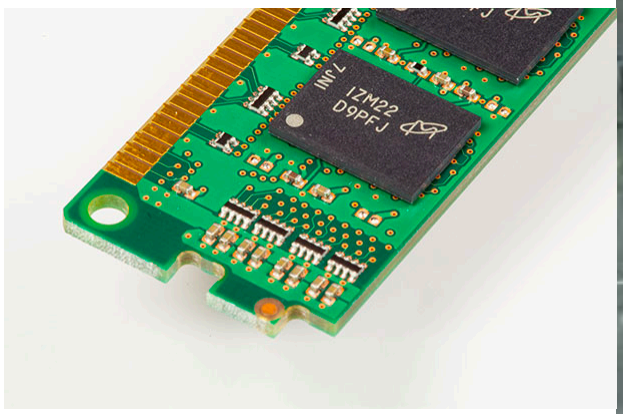


Minimize the access time vs. Cost

Memory

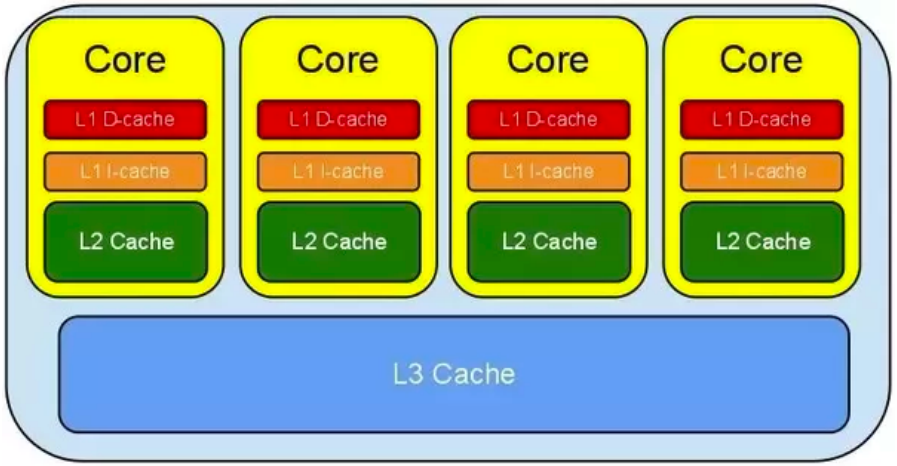
- A typical memory hierarchy

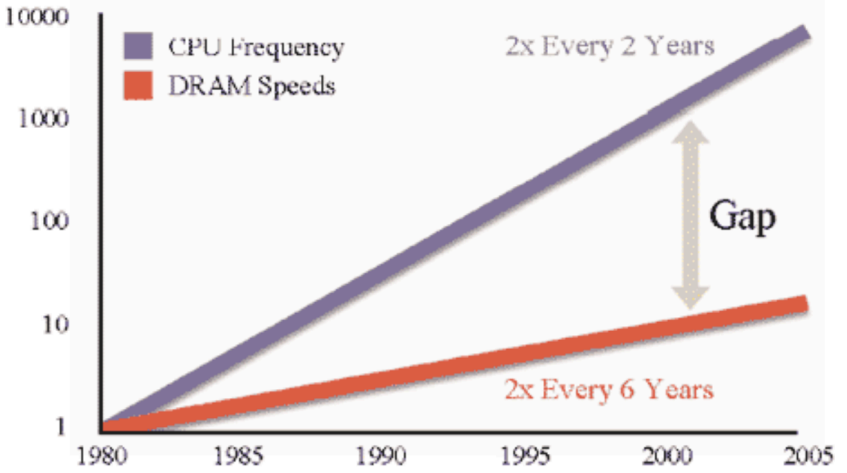




Cache

- 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

Check Cache info

- $ likwid-topology –g
- $ lscpu | grep cache

```
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   16 MB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 )
Distances:              10
Free memory:            1967.79 MB
Total memory:           3942.19 MB
--------------------------------------------------------------------------------
```

```
ksuo@ksuo-VirtualBox ~/likwid-5.0.0> lscpu | grep cache
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              16384K
```

Memory information

- free

```
administrator@ubuntuvm-1604 ~> free
              total        used        free      shared  buff/cache   available
Mem:        8168756     1348160     3031888       97424     3788708     6383036
Swap:        998396           0      998396
```

- The free command displays:
  - ✓ Total amount of free and used physical memory
  - ✓ Total amount of swap memory in the system
  - ✓ Buffers and caches used by the kernel

Memory information

- cat /proc/meminfo
  - MemTotal
  - MemFree
  - MemAvailable
  - Buffers
  - Cached
  - SwapCached
  - SwapTotal
  - SwapFree

```
administrator@ubuntuvm-1604 ~> cat /proc/meminfo
MemTotal:        8168756 kB
MemFree:         3034596 kB
MemAvailable:    6385744 kB
Buffers:          287636 kB
Cached:          3188812 kB
SwapCached:            0 kB
Active:          2523764 kB
Inactive:        2177968 kB
Active(anon):    1226312 kB
Inactive(anon):    96392 kB
Active(file):    1297452 kB
Inactive(file):  2081576 kB
Unevictable:          48 kB
Mlocked:              48 kB
SwapTotal:       998396 kB
SwapFree:        998396 kB
Dirty:              264 kB
Writeback:            0 kB
AnonPages:       1225344 kB
Mapped:           304212 kB
Shmem:             97424 kB
Slab:             312268 kB
SReclaimable:     278104 kB
SUnreclaim:        34164 kB
KernelStack:        8896 kB
PageTables:        31004 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     5082772 kB
Committed_AS:    5183160 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      135040 kB
DirectMap2M:     5107712 kB
DirectMap1G:     5242880 kB
```

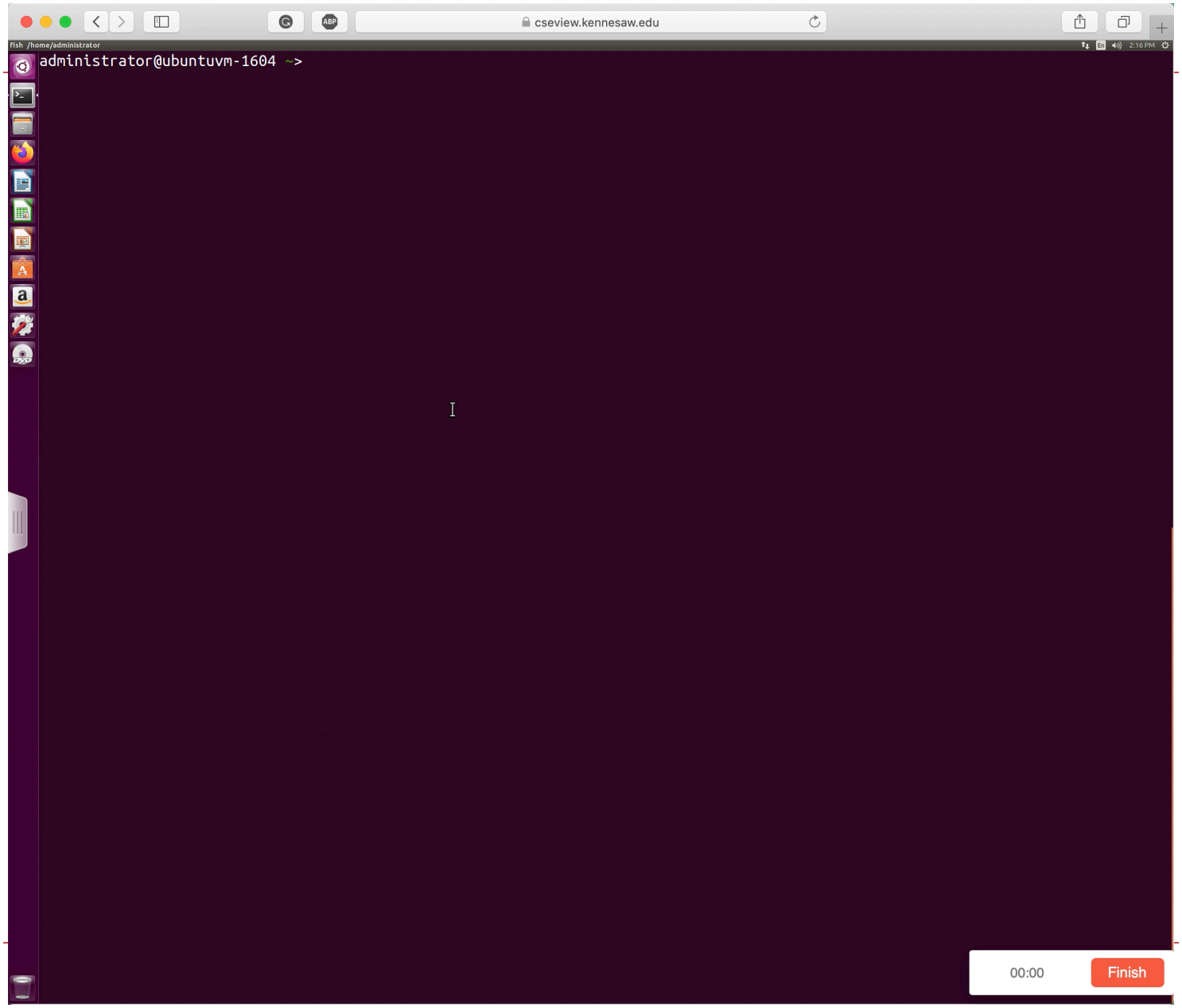
Memory information

- lstopo: show the CPU cache and logical CPU layout

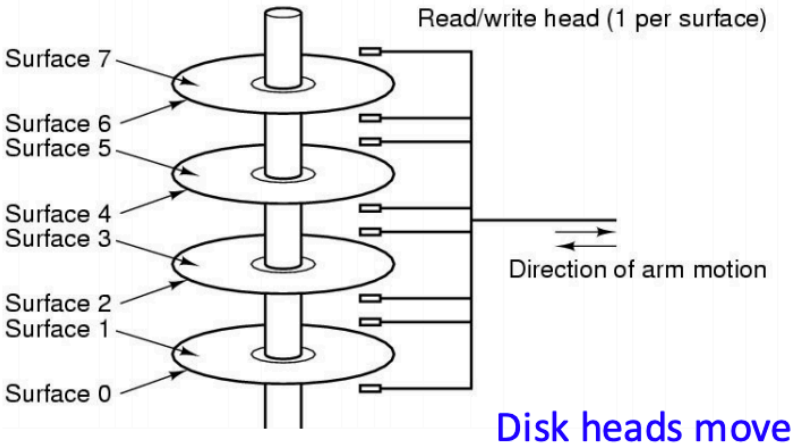
- Intall: $ sudo apt-get install hwloc

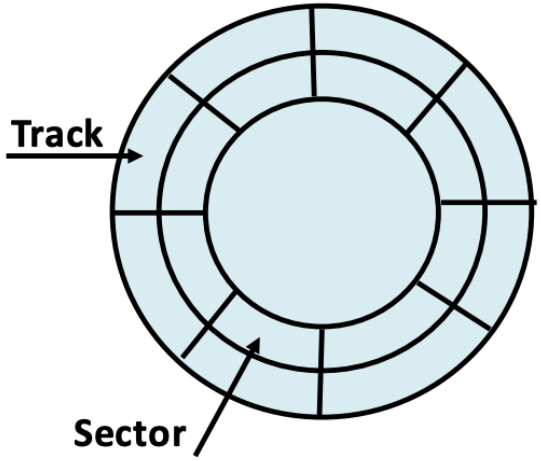
- Run: $ lstopo

https://youtu.be/65c5yEC_ZlA



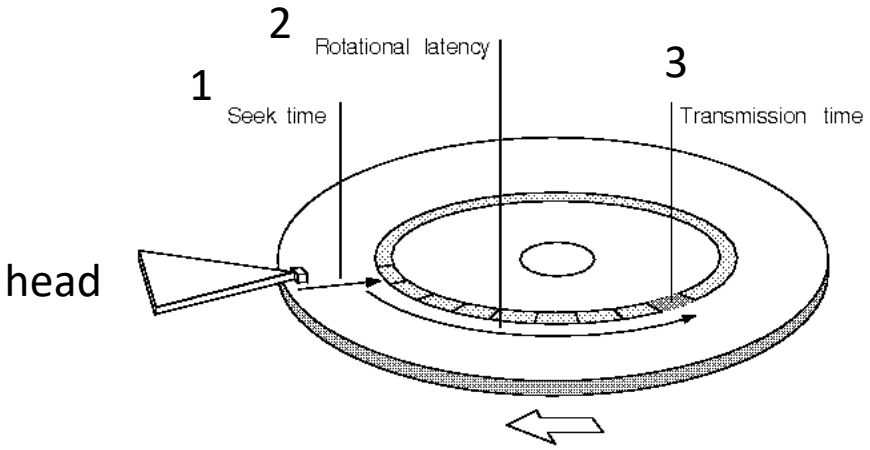
Disk





- 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

Disk information

- lsblk
  - Lists out all the storage blocks, which includes disk partitions and optical drives. Details include the total size of the partition/block and the mount point if any.

```
administrator@ubuntuvm-1604 ~> lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sr0     11:0    1  1.5G  0 rom  /media/administrator/Ubuntu 16.04.5 LTS amd64
sda      8:0    0   40G  0 disk
├─sda2   8:2    0    1K  0 part
├─sda5   8:5    0  975M  0 part [SWAP]
└─sda1   8:1    0   39G  0 part /
```

Disk information

- df
  - prints out details about only mounted file systems. The list generated by df even includes file systems that are not real disk partitions.

```
administrator@ubuntuvm-1604 ~> df
Filesystem     1K-blocks    Used Available Use% Mounted on
udev             4054632       0   4054632   0% /dev
tmpfs             816876   42148    774728   6% /run
/dev/sda1       40168028 7350596  30753972  20% /
tmpfs            4084376    1044   4083332   1% /dev/shm
tmpfs               5120       0      5120   0% /run/lock
tmpfs            4084376       0   4084376   0% /sys/fs/cgroup
/dev/sr0         1610928 1610928         0 100% /media/administrator/Ubuntu 16.04.5 LTS amd64
tmpfs             816876      56    816820   1% /run/user/1000
```

Disk information

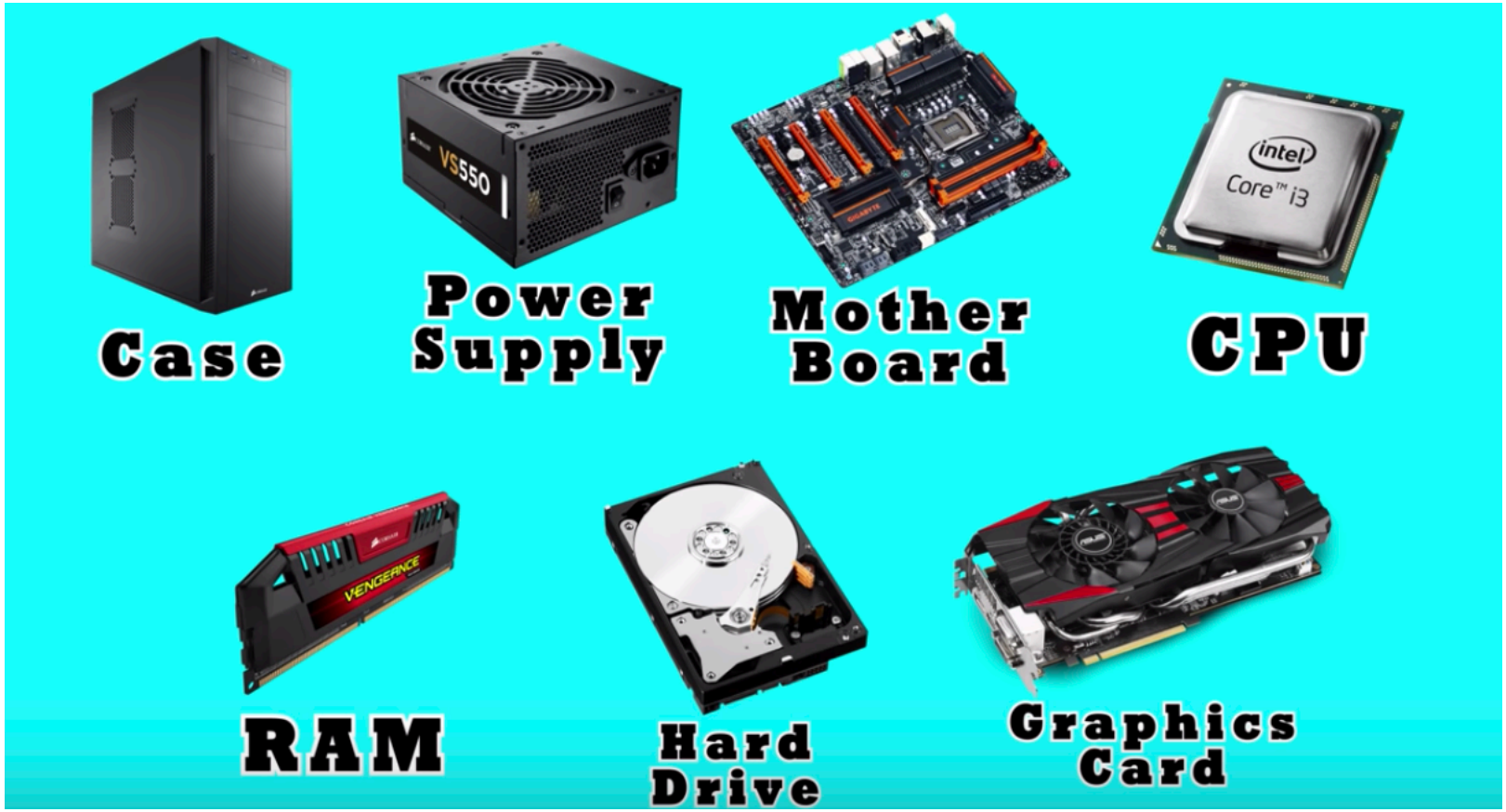
- fdisk
  - display the partitions and details like file system type

```
administrator@ubuntuvm-1604 ~> sudo fdisk -l
[sudo] password for administrator:
Disk /dev/sda: 40 GiB, 42949672960 bytes, 83886080 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x6e6dc012

Device     Boot    Start      End  Sectors  Size Id Type
/dev/sda1  *        2048 81885183 81883136   39G 83 Linux
/dev/sda2       81887230 83884031  1996802  975M  5 Extended
/dev/sda5       81887232 83884031  1996800  975M 82 Linux swap / Solaris
```

Disk R/W Process

https://www.youtube.com/watch?v=3owqvmMf6No

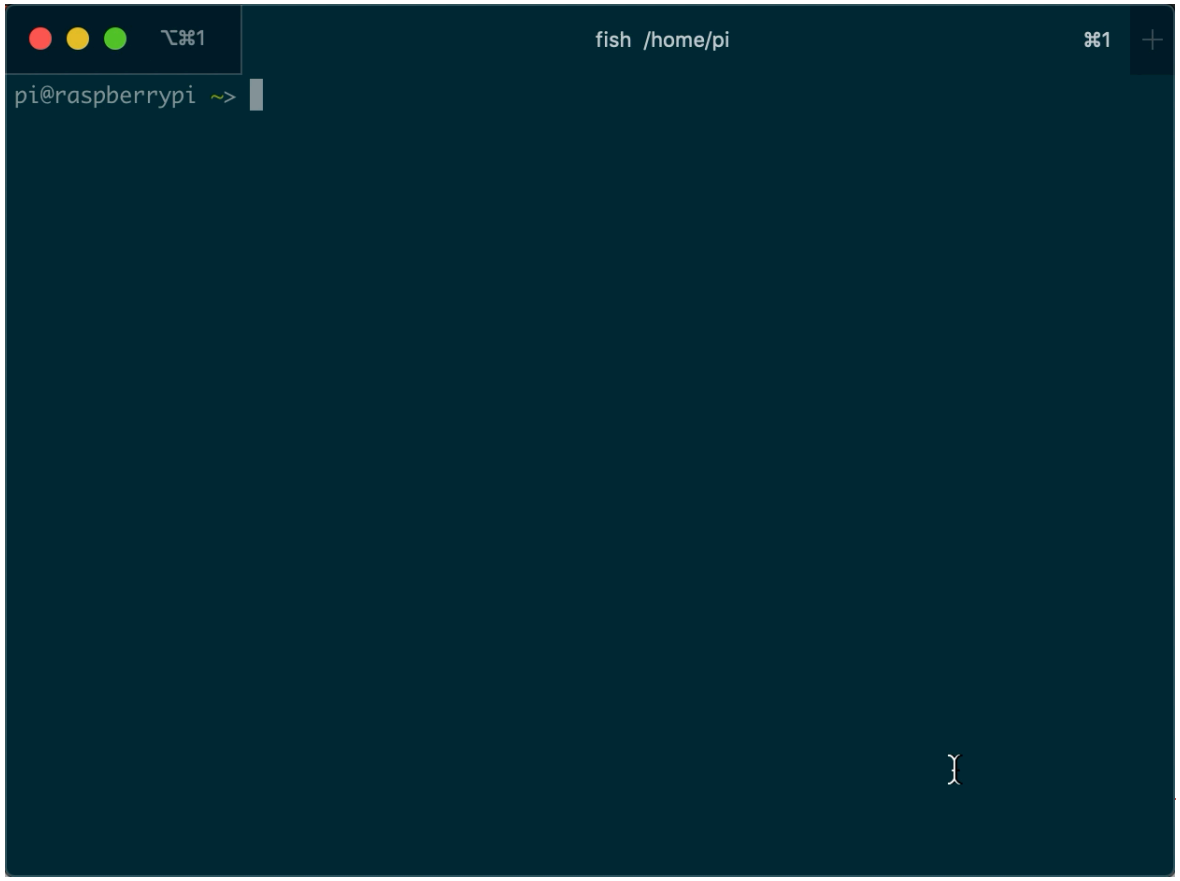
An interesting video introducing hardware



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

How to get my machine spec?

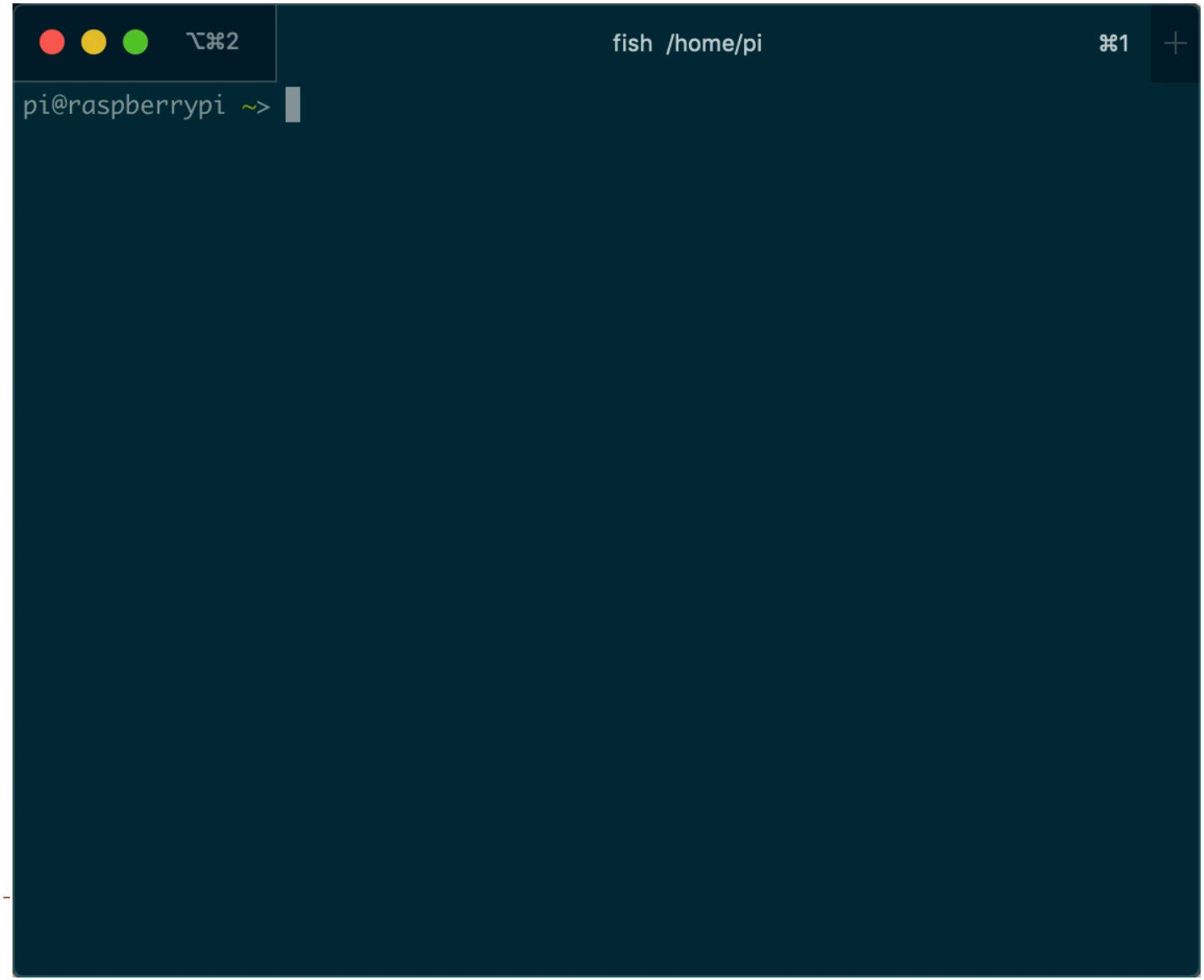
- Inxi: https://www.tecmint.com/inxi-command-to-find-linux-system-information/



https://youtu.be/aaO8b_ebWpM

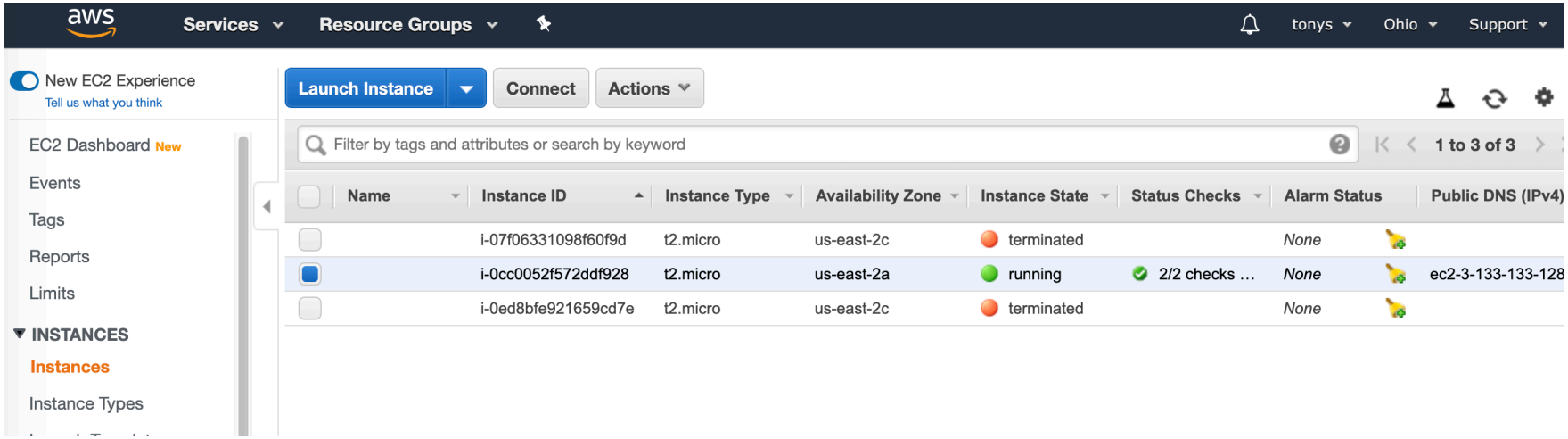
How to get my machine spec? One command for All!

https://youtu.be/LMbBO8k2vOE



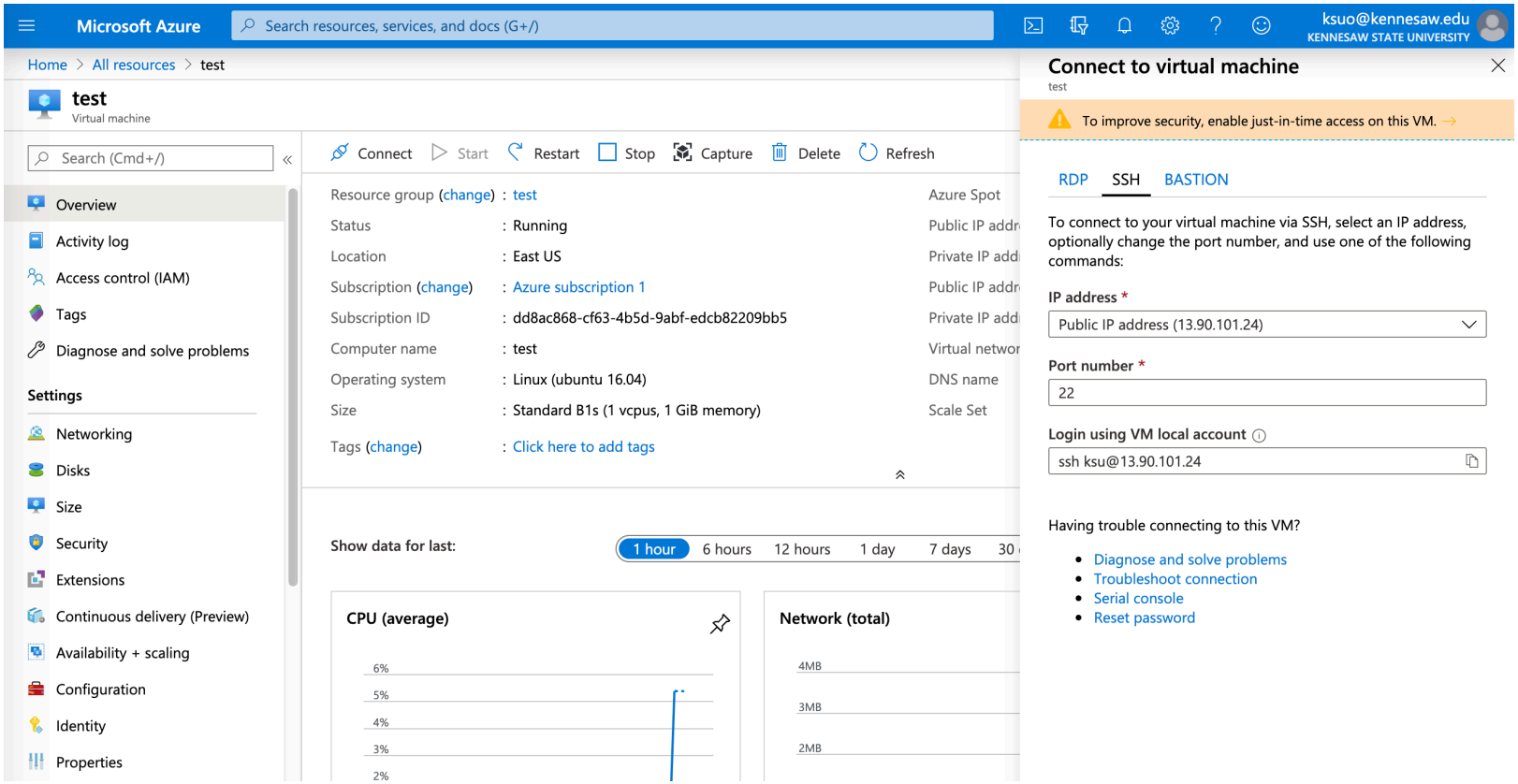
Test 1: Amazon Web Service VM

- ssh -i "osclass.pem" ubuntu@ec2-3-133-133-128.us-east-2.compute.amazonaws.com



Test 2: Microsoft Azure VM

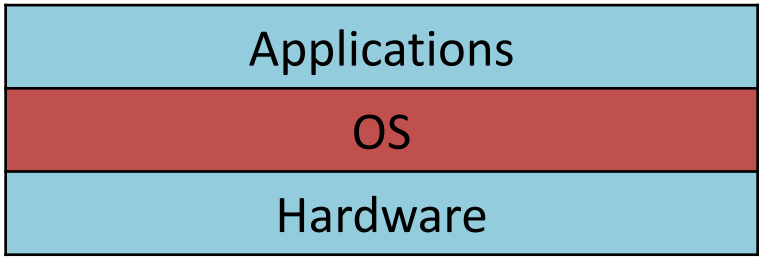
- ssh ksu@13.90.101.24



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.

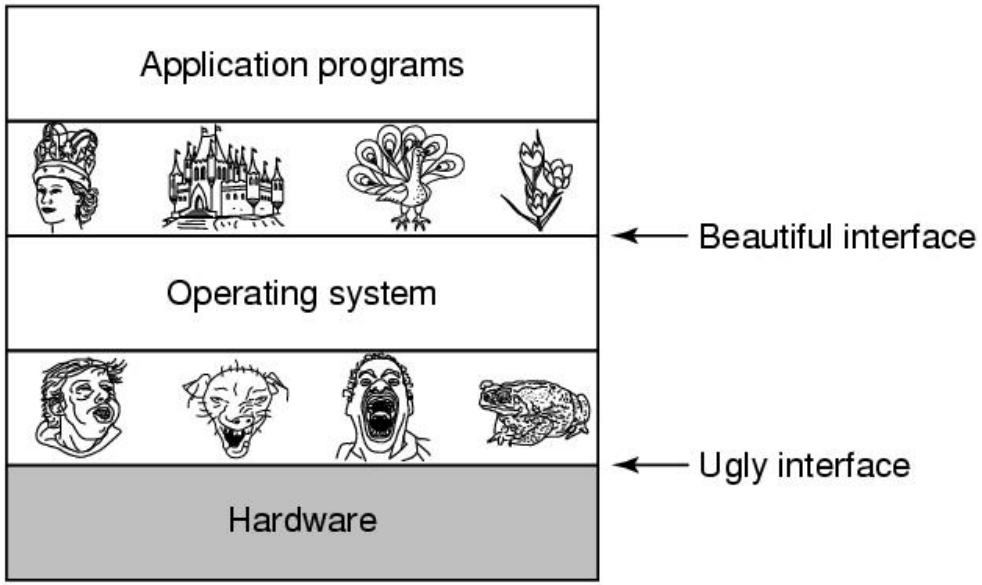
Three themes of operating systems: virtualization, concurrency and persistence.

What is an OS?

- Theme 1: Virtualization
- Theme 2: Concurrency
- Theme 3: Persistency

Theme 1: Virtualization

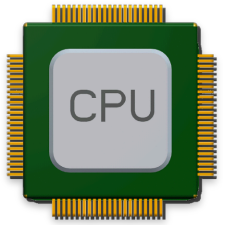
- Making a physical resource look like something else (virtual).
- Why virtualize?
  - To make the computer easier to use and program.



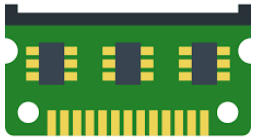

```
fprintf(fd, "%d", data);
        ↓
write(fd, buffer, count) ;
        ↓
file->f_op->write(file, buf,
count, pos);
        ↓
```

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 (What does this program do?)

```
#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;
}
```

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;
}
```

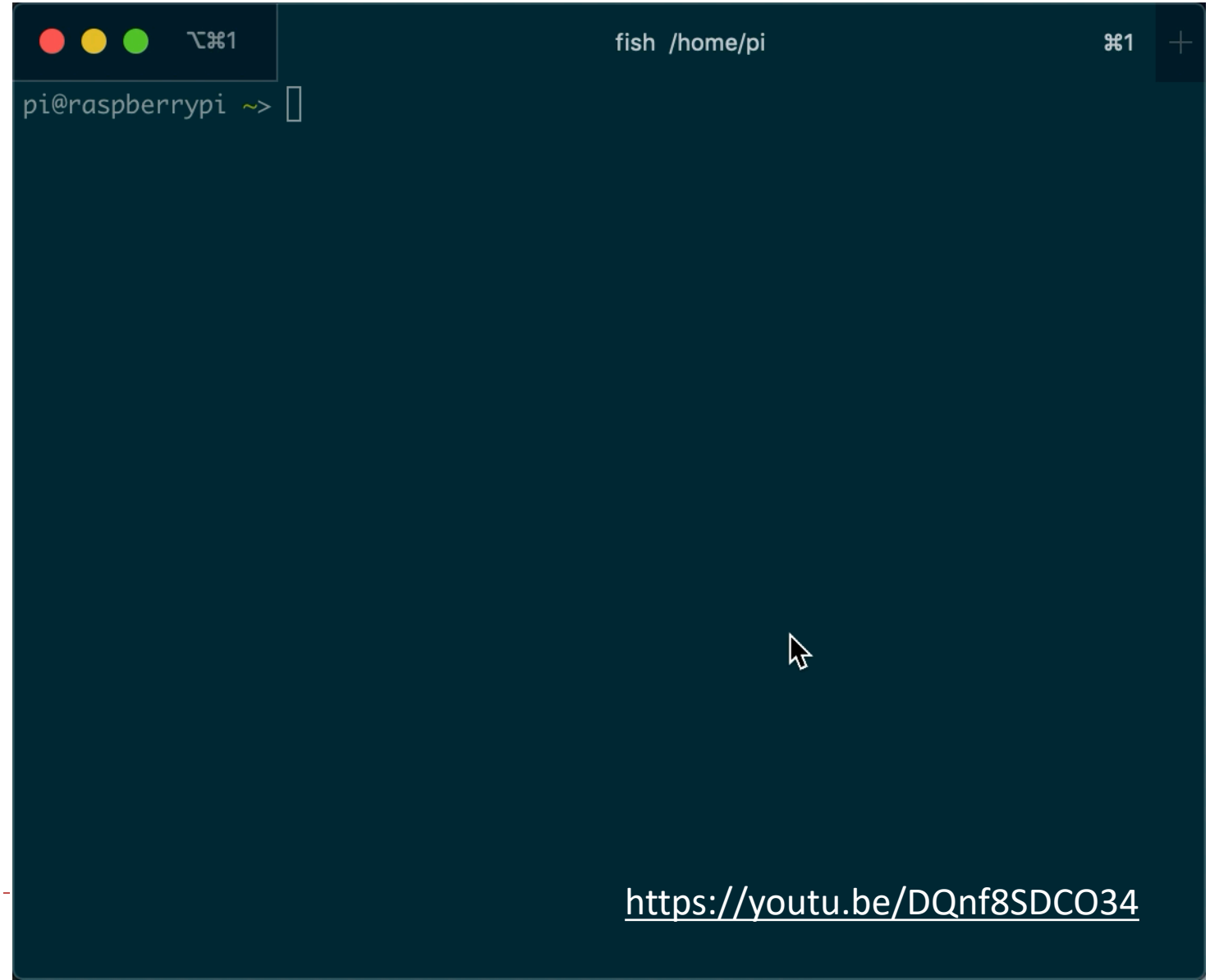
If parameter number is not 2, print error message

input

Infinite loop

print

Virtualization example – Virtualizing the CPU



https://youtu.be/DQnf8SDCO34

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?

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.

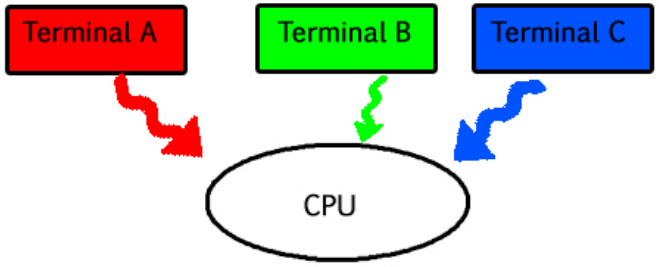
```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
```

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 (What does this program do?)

```
#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;
}
```

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

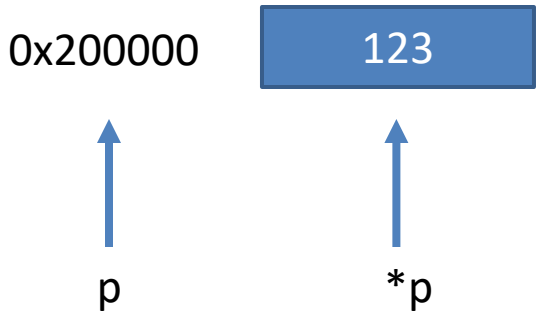
the value held in p

Virtualization example – Virtualizing the Memory

- 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));
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
           getpid(), p);
    *p = 0;
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);
    }
    return 0;
}
```

What is the variable p stored?



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
```

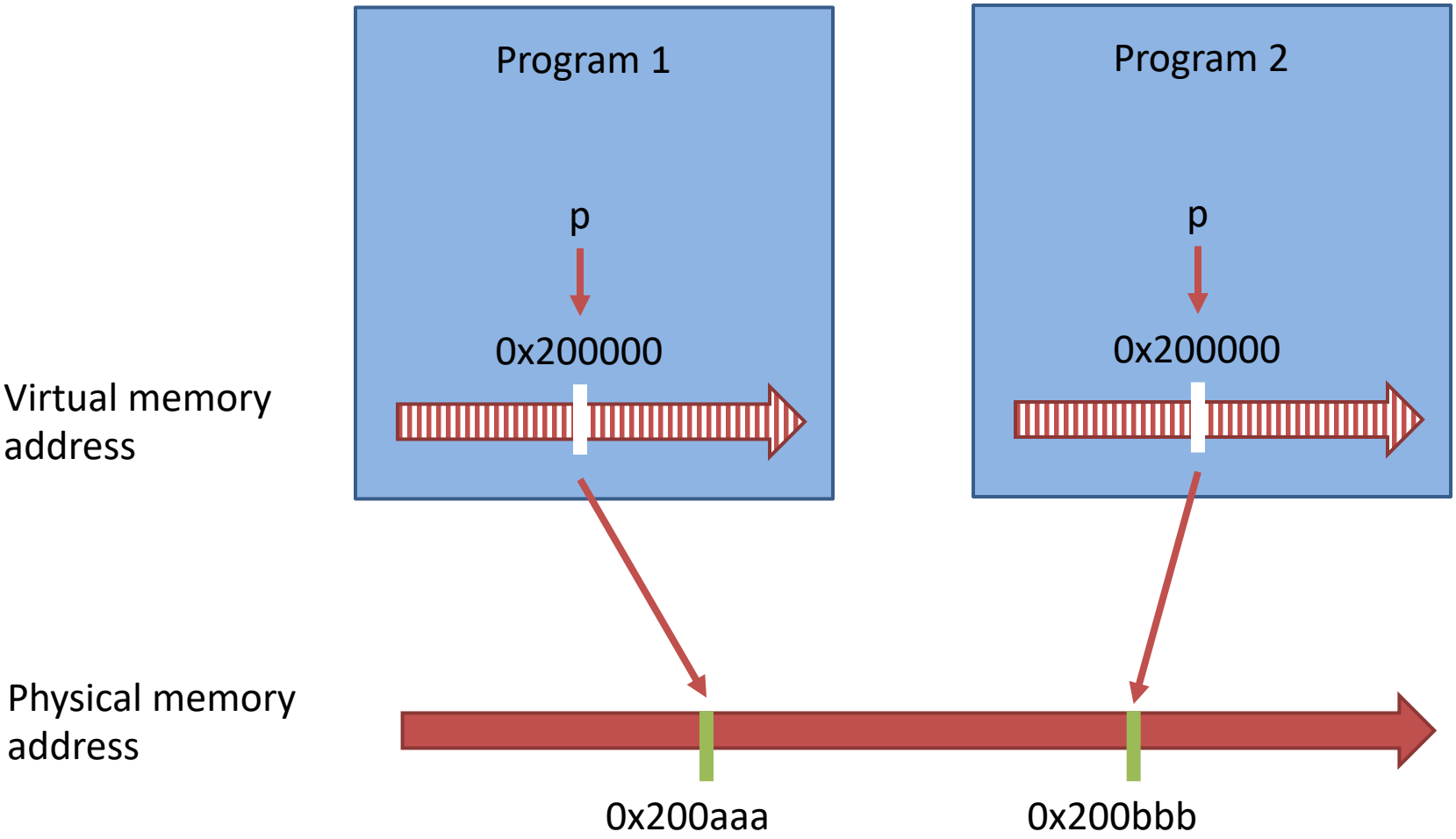
Run two instances of the "mem" program.

```
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;
}
```

- thread.c (What does this program do?)

Expected output?

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

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?

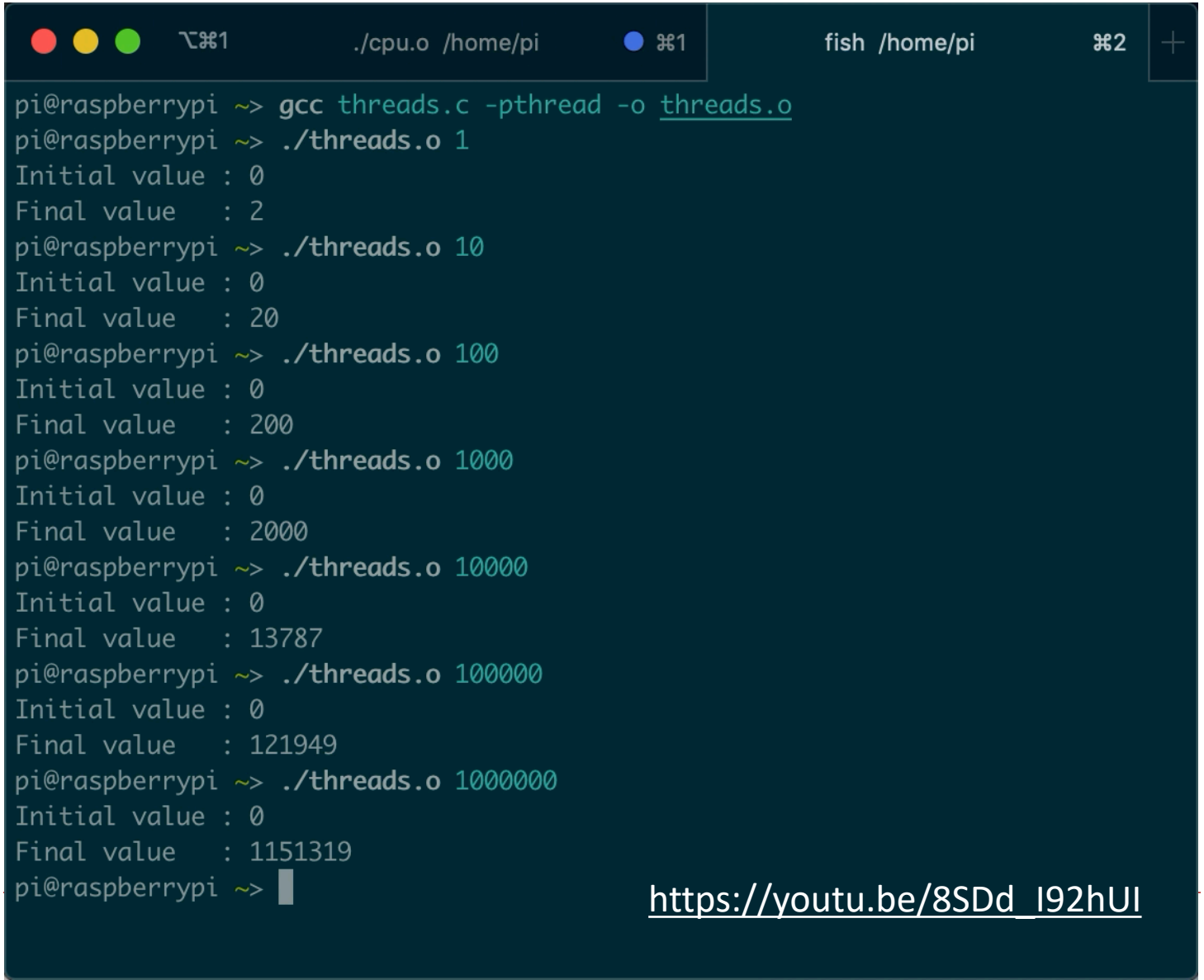
```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

Counter value: before

Two threads increase a counter

Counter value: after

Concurrency Example



https://youtu.be/8SDd_I92hUI

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)

Persistence Example

https://github.com/kevinsuo/CS3502/blob/master/storage.c

```c
#include <stdio.h>
#define SIZE 5
typedef struct
{
        char name[20];
        char num[15];
        int age;
}student;
student stu[SIZE],buf[SIZE];


int main()
{
        FILE *fp;
        char filename[50],ch;
        printf("input your name, num, and age:\n");
        fscanf(stdin,"%s %s %d",&stu[0].name,&stu[0].num,&stu[0].age);

        printf("Please input your read filename:");
        scanf("%s",filename);
        if(!(fp=fopen(filename,"w")))
        {
                printf("Can not open %s",filename);
                return 1;
        }
        fprintf(fp,"name:%s num:%s age:%d",stu[0].name,stu[0].num,stu[0].age);
        fclose(fp);

        return 0;
}
```

Where is OS in this program?

fscanf: read data from input source

fprintf: write data into destination file

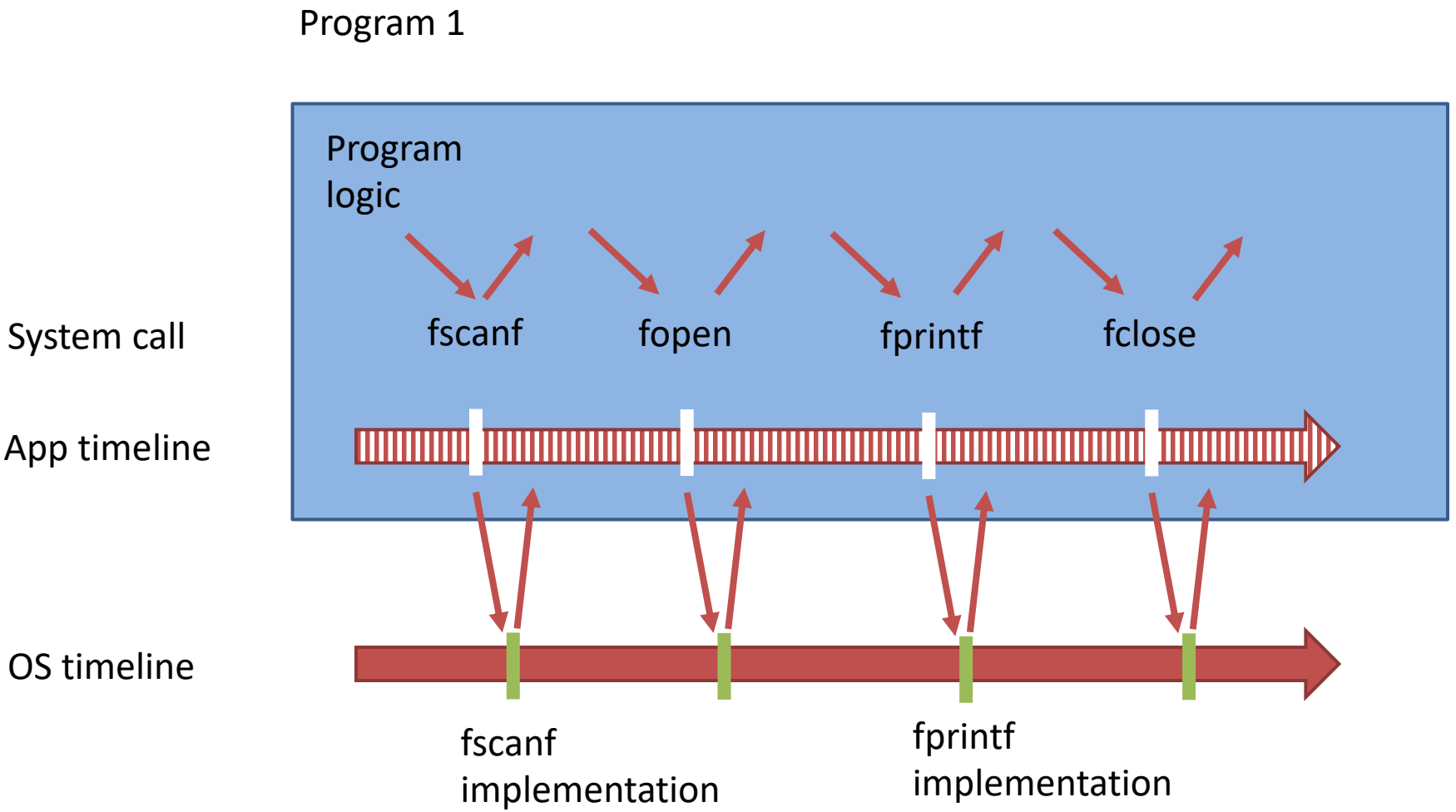
Persistence Example

```c
#include <stdio.h>
#define SIZE 5
typedef struct
{
        char name[20];
        char num[15];
        int age;
}student;
student stu[SIZE],buf[SIZE];


int main()
{
        FILE *fp;
        char filename[50],ch;
        printf("input your name, num, and age:\n");
        fscanf(stdin,"%s %s %d",&stu[0].name,&stu[0].num,&stu[0].age);

        printf("Please input your read filename:");
        scanf("%s",filename);
        if(!(fp=fopen(filename,"w")))
        {
                printf("Can not open %s",filename);
                return 1;
        }
        fprintf(fp,"name:%s num:%s age:%d",stu[0].name,stu[0].num,stu[0].age);
        fclose(fp);

        return 0;
}
```

Where is OS in this program?

Persistence Example

Where is OS in this program?



Persistence Example

```
ksuo@ksuo-VirtualBox ~> gcc a.c -o a.o
a.c: In function ‘main’:
a.c:17:17: warning: format ‘%s’ expects argument of type ‘char *’, but argument 3 has type ‘char (*)[2
  fscanf(stdin,"%s %s %d",&stu[0].name,&stu[0].num,&stu[0].age);
                ~^        ~~~~~~~~~~~~
a.c:17:20: warning: format ‘%s’ expects argument of type ‘char *’, but argument 4 has type ‘char (*)[1
  fscanf(stdin,"%s %s %d",&stu[0].name,&stu[0].num,&stu[0].age);
                  ~^                  ~~~~~~~~~~~
ksuo@ksuo-VirtualBox ~> ./a.o
input your name, num, and age:
alice 123 20
Please input your read filename:a.txt
```

https://youtu.be/8GkhGP-bUbw

Above example code

- Code example:

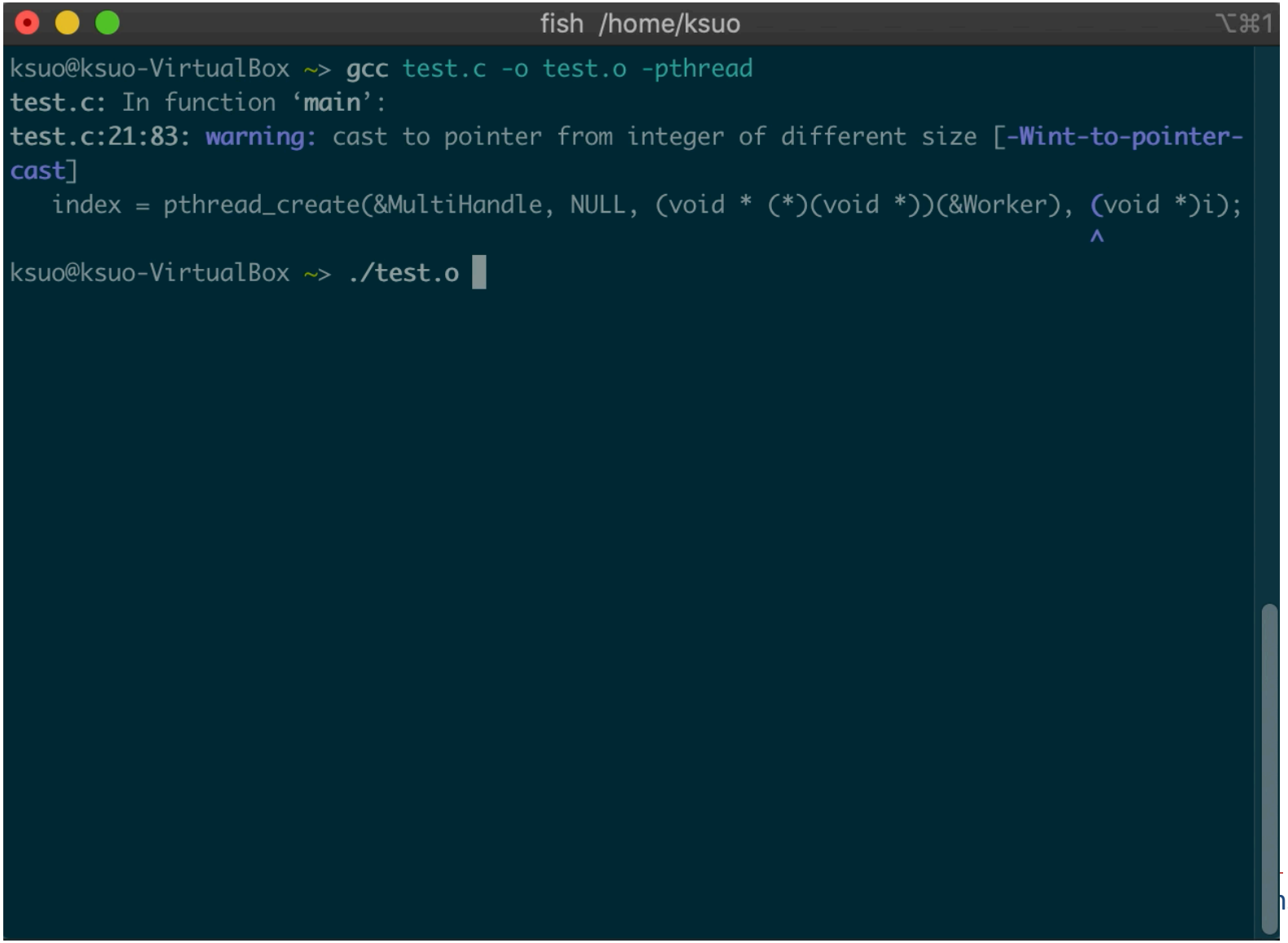
https://github.com/remzi-arpacidusseau/ostep-code/tree/master/intro

Test 1: concurrency

https://github.com/kevinsuo/CS3502/blob/master/test.c

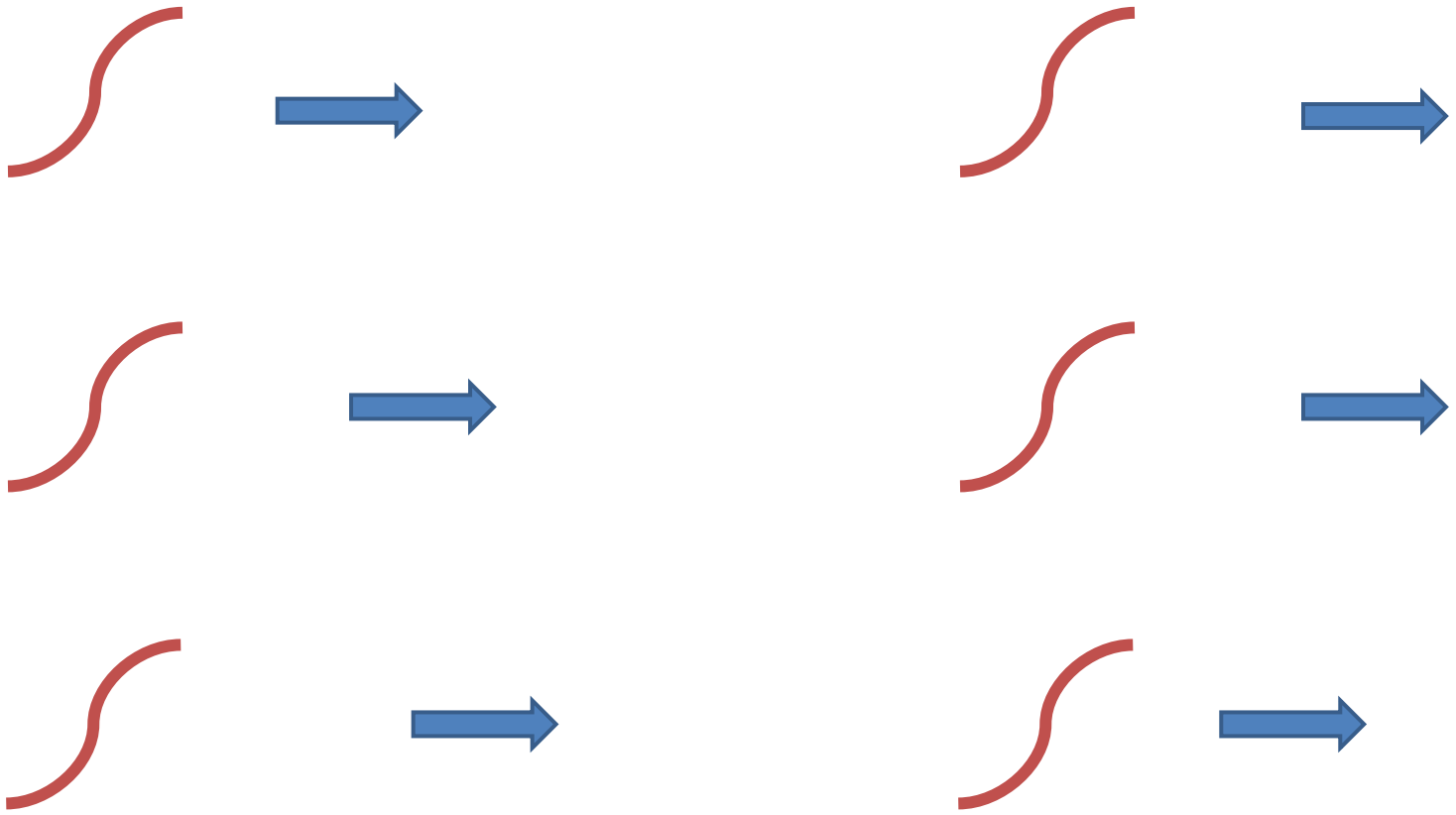
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

unsigned int X = 0;

void Worker(void *pParam)
{
    X++;
}


int main()
{
    pthread_t MultiHandle       = 0;
    unsigned int    i           = 0;
    signed   int    index       = 0;

    for (i = 0; i < 100; i ++)
    {
        index = pthread_create(&MultiHandle, NULL, (void * (*)(void *))(&Worker), (void *)i);
        if (0 != index)
        {
            printf("Create Worker %d failed!\n", i);
            return -1;
        }
    }

    printf("In main, X = %d\n", X);

    return 0;
}
```

Test 1: concurrency



Why?

Test 1: concurrency

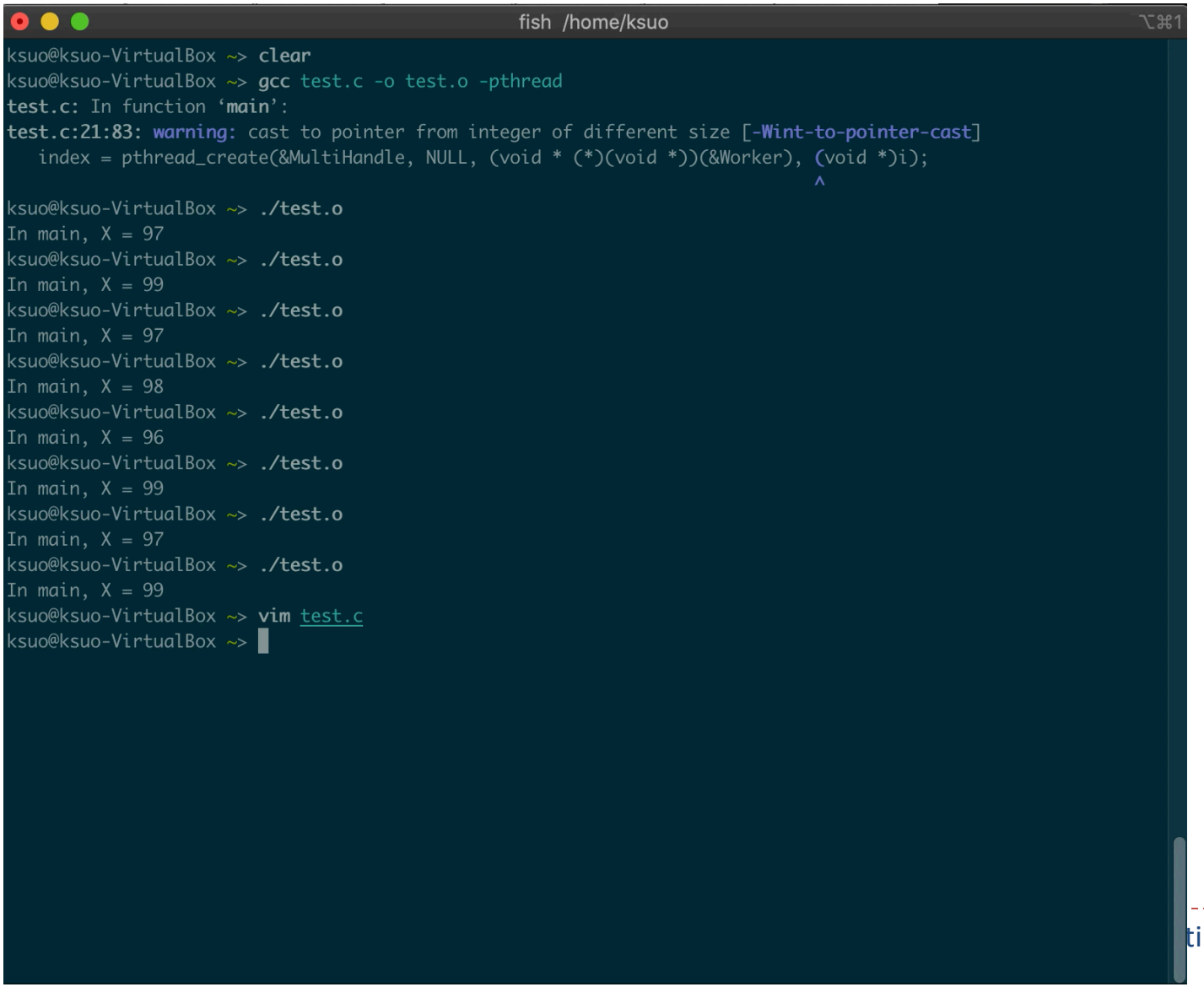


Test 1: concurrency

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

unsigned int X = 0;

void Worker(void *pParam)
{
    X++;
}


int main()
{
    pthread_t MultiHandle       = 0;
    unsigned int    i           = 0;
    signed   int    index       = 0;

    for (i = 0; i < 100; i ++)
    {
        index = pthread_create(&MultiHandle, NULL, (void * (*)(void *))(&Worker), (void *)i);
        if (0 != index)
        {
            printf("Create Worker %d failed!\n", i);
            return -1;
        }
    }

    sleep(3);

    printf("In main, X = %d\n", X);

    return 0;
}
```

Sleep for 3 seconds

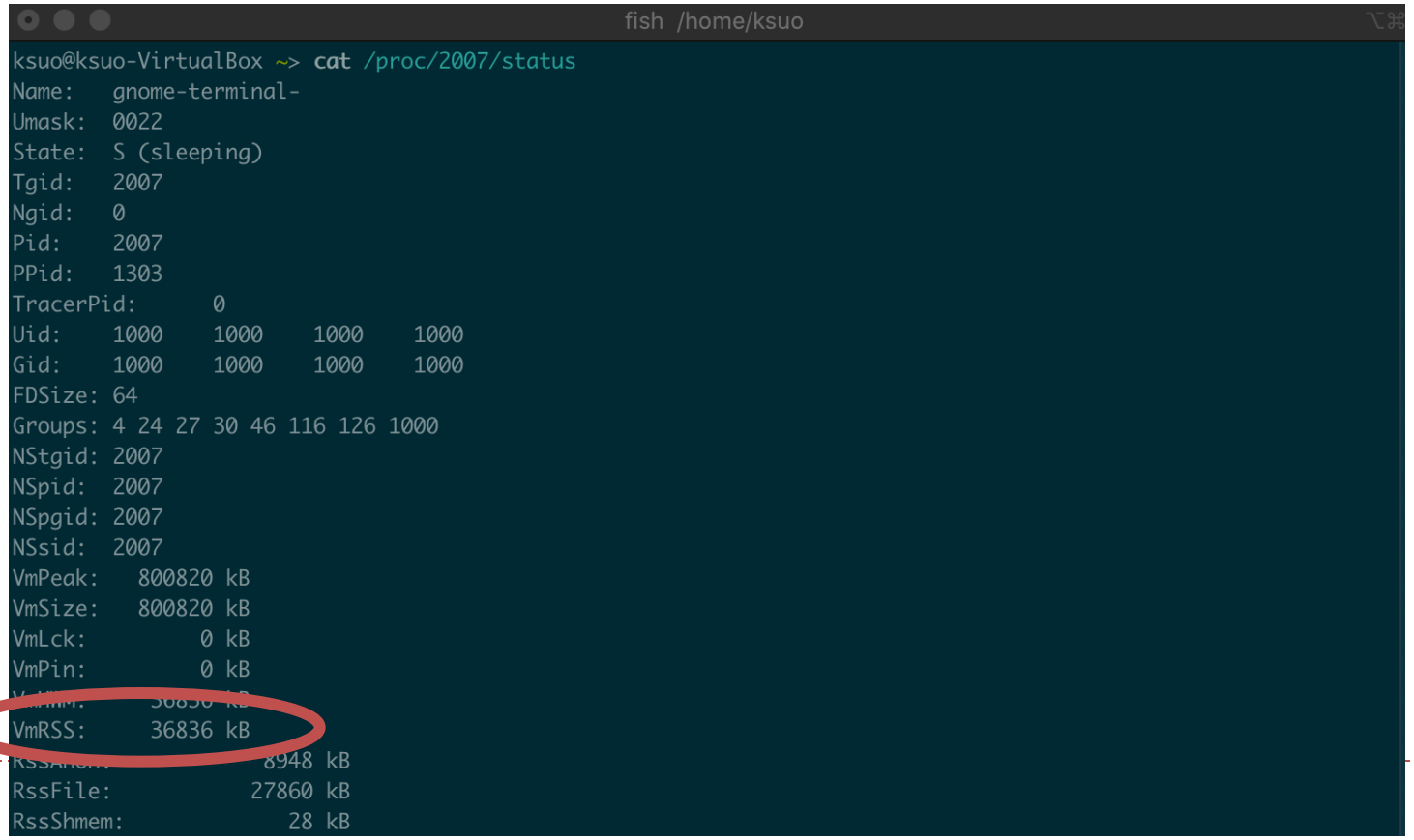
Test 1: concurrency



Test 2: virtualization

https://github.com/kevinsuo/CS3502/blob/master/test2.cpp

- /proc/[pid]/status
- VmRSS: Resident set size. Note that the value here is the sum of RssAnon, RssFile, and RssShmem.



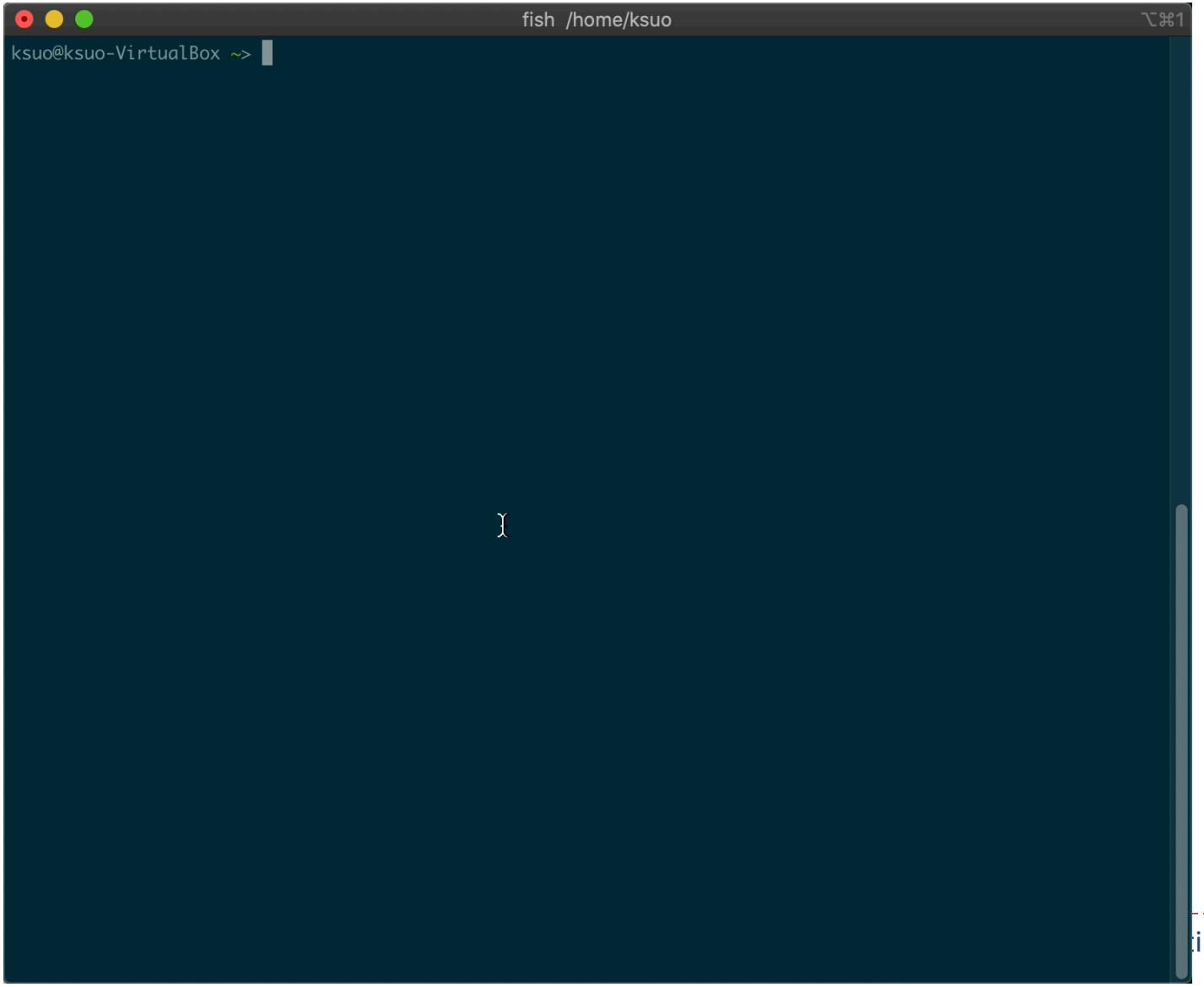
Test 2: virtualization

vim /home/ksuo

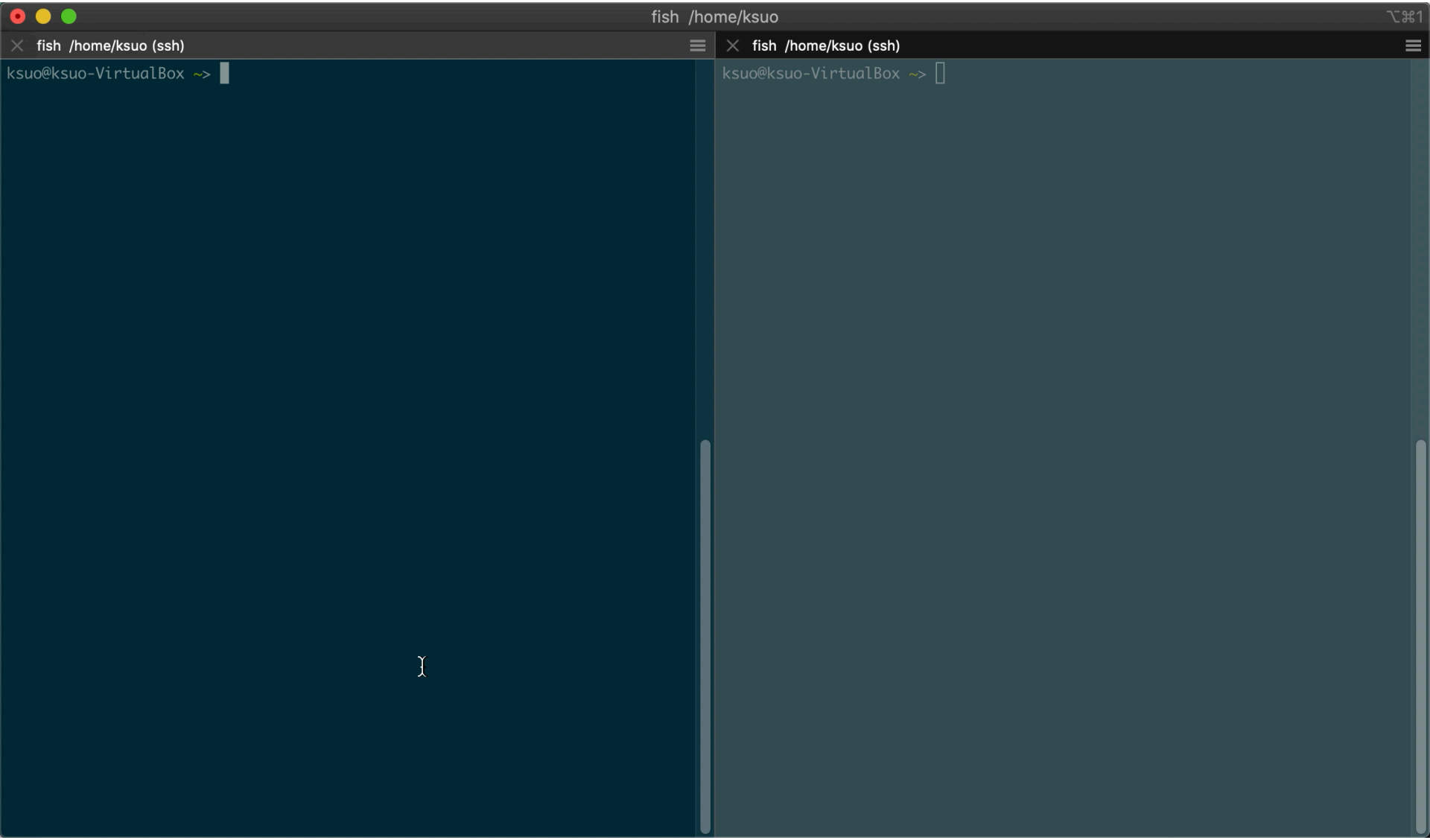
```
#include <iostream>
#include <unistd.h>
#include <string.h>
#include <cstdlib>
using namespace std;

size_t physical_memory_used_by_process()
{
    FILE* file = fopen("/proc/self/status", "r");
    int result = -1;
    char line[128];

    while (fgets(line, 128, file) != nullptr) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            int len = strlen(line);

            const char* p = line;
            for (; std::isdigit(*p) == false; ++p) {}

            line[len - 3] = 0;
            result = atoi(p);

            break;
        }
    }
    fclose(file);

    return result;
}

int main()
{
        int *p = new int;
        *p = 100;
        while(1) {
                cout << getpid() << ", Current memory usage: " << physical_memory_used_by_process()
                        << ", " << p << ", " << *p << endl;
                sleep(1);
        }
    return 0;
}
```

- test.c (What does this program do?)

Test 2: virtualization



Test 2: virtualization

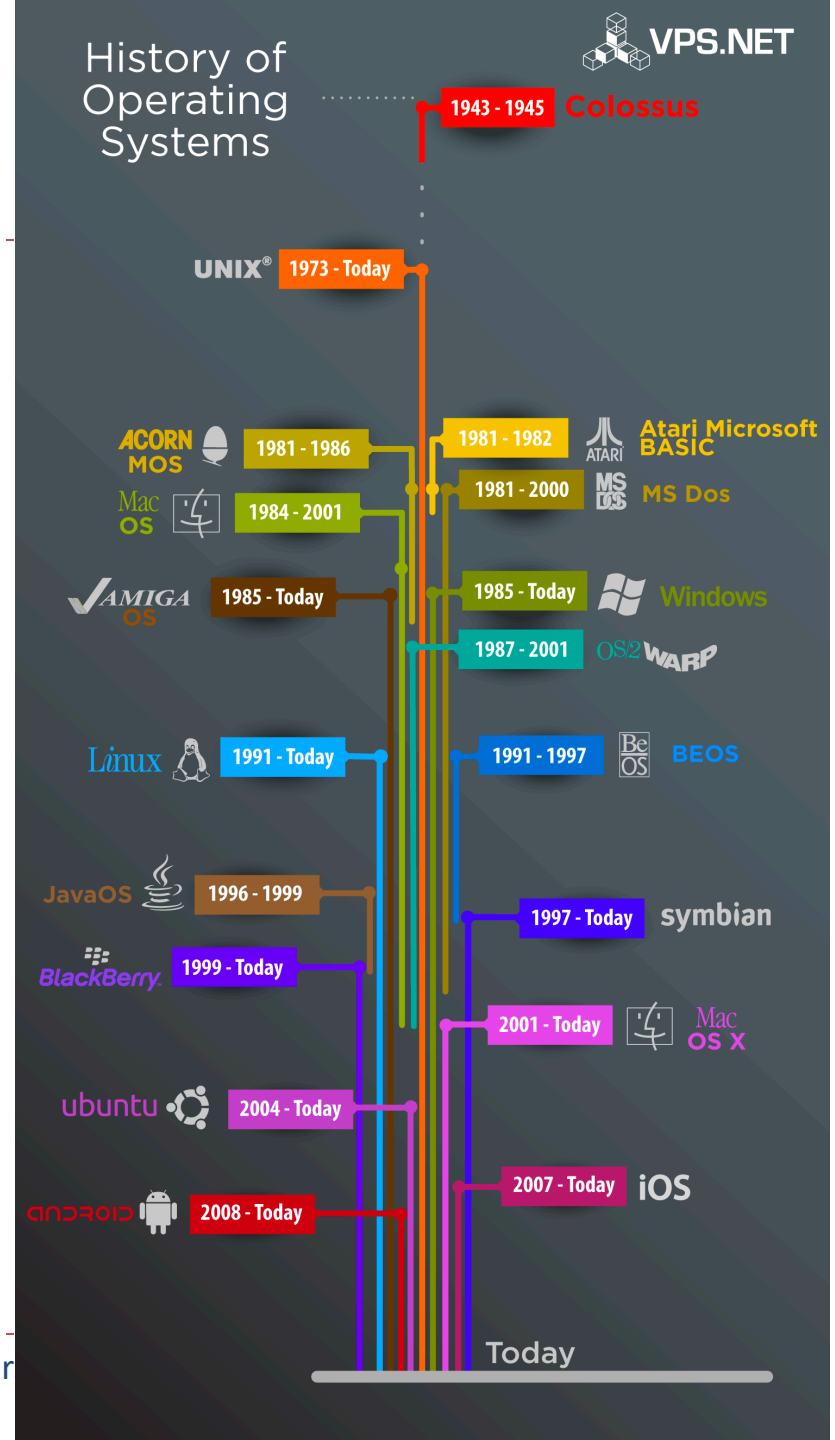


Outline

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

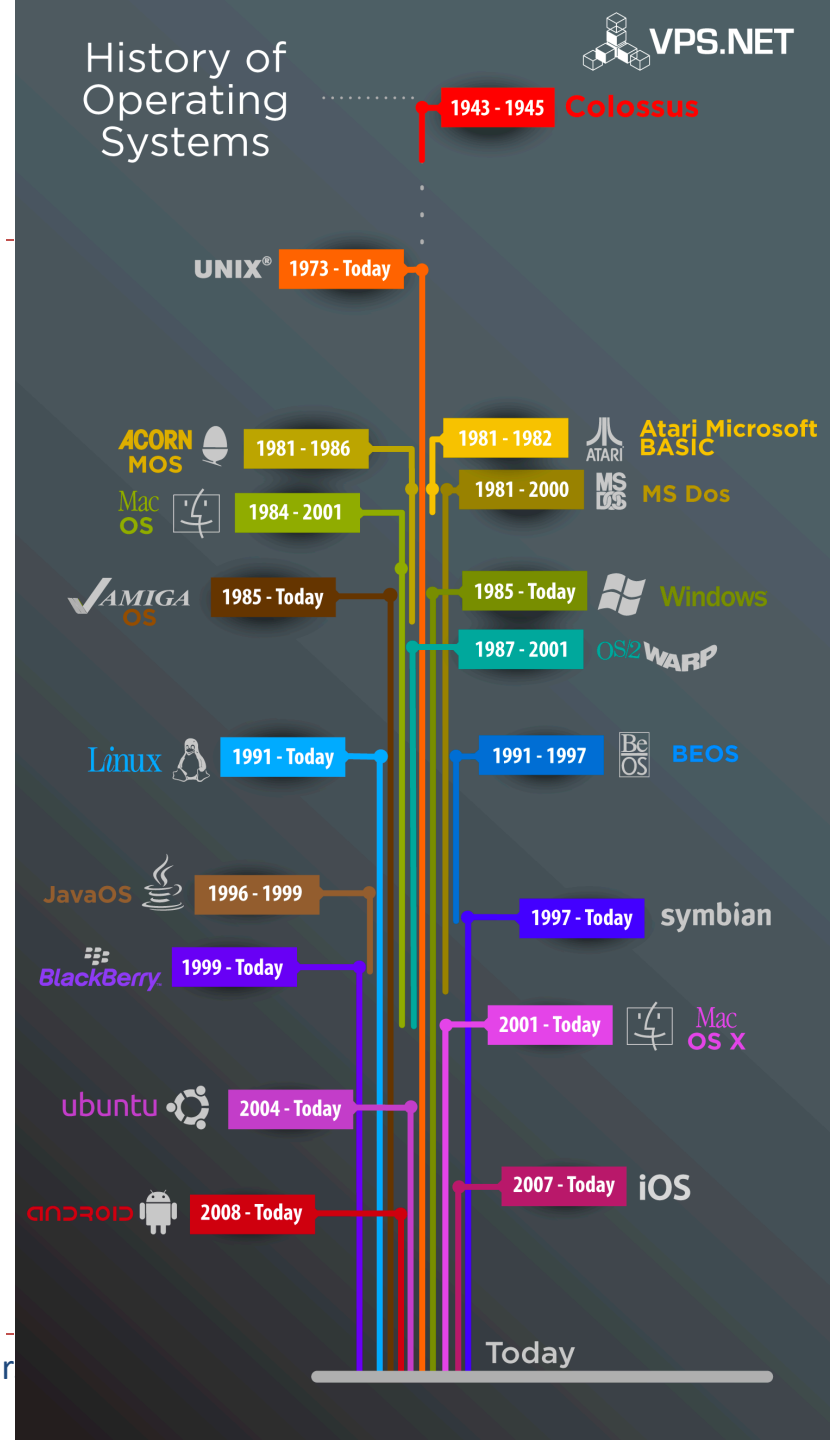
History of OS

- 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.

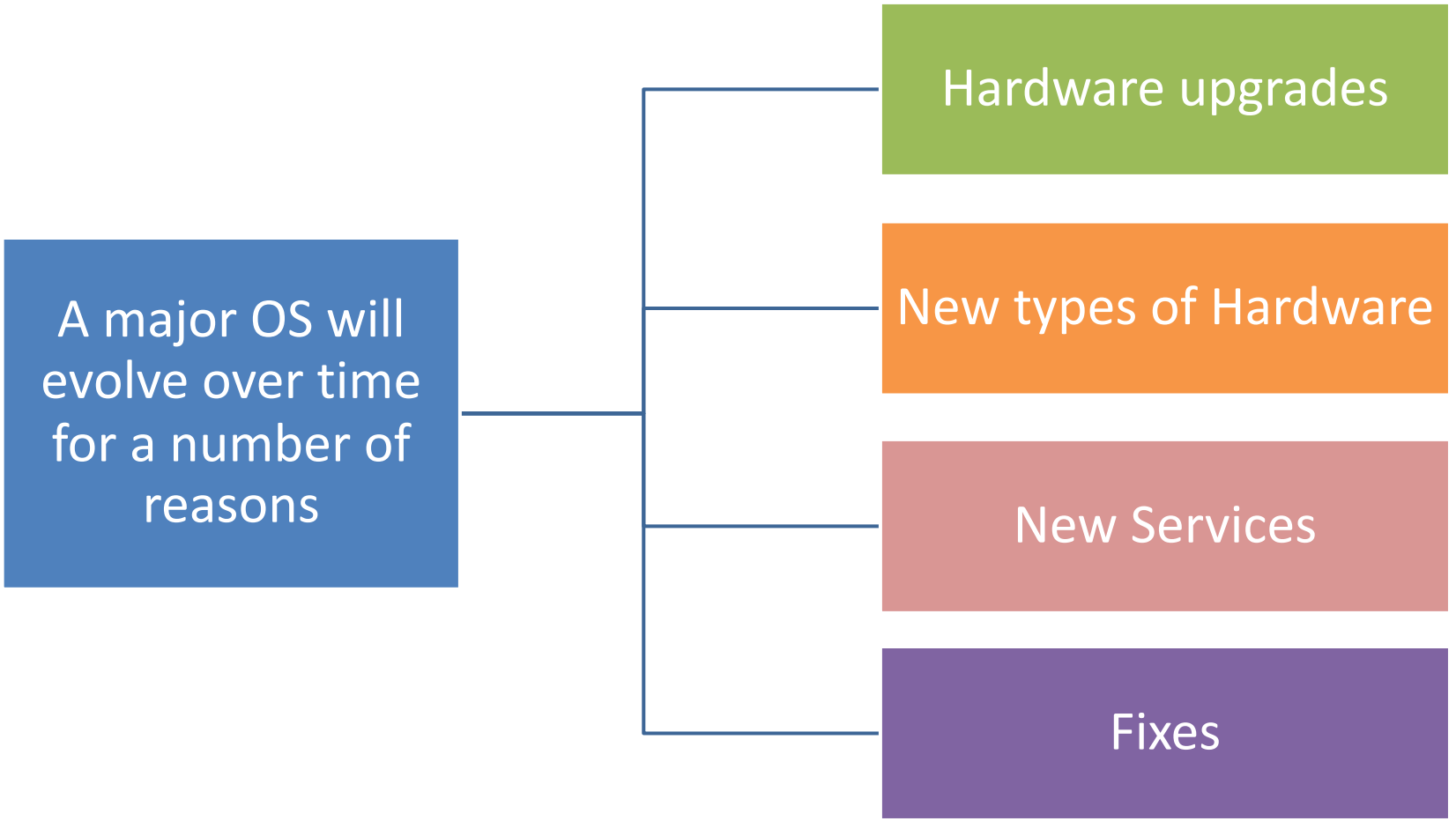


History of OS

- 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.

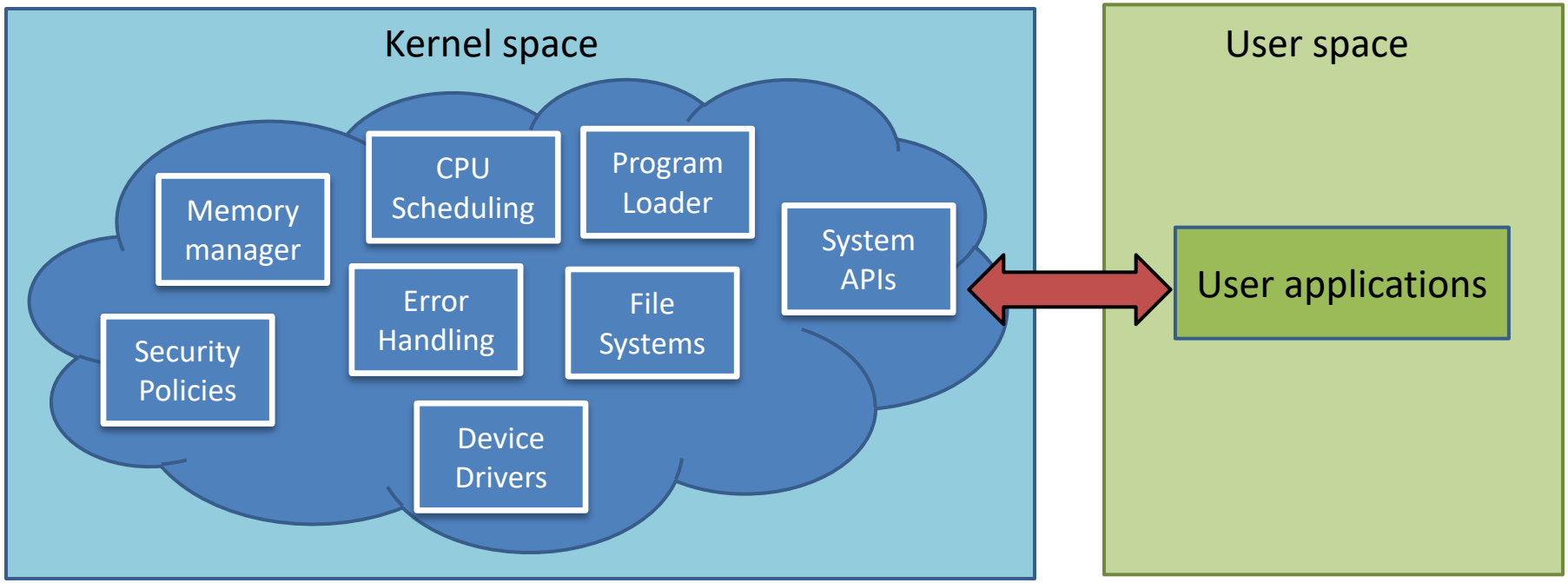
- What is the purpose of having these two modes?
  - Purpose: protection – to protect critical resources (e.g., privileged instructions, memory, I/O devices) from being misused by user programs.

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

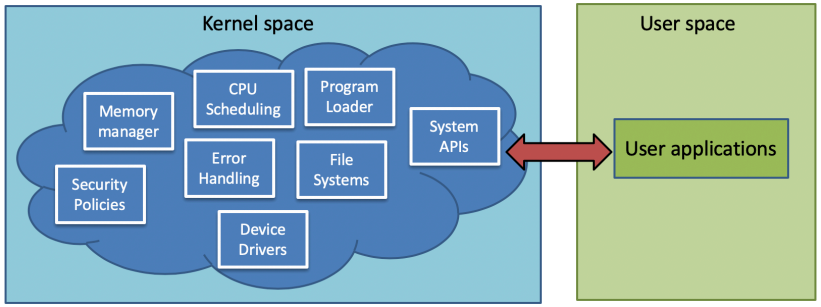
Monolithic Kernel

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



Pros/Cons of Monolithic Kernels

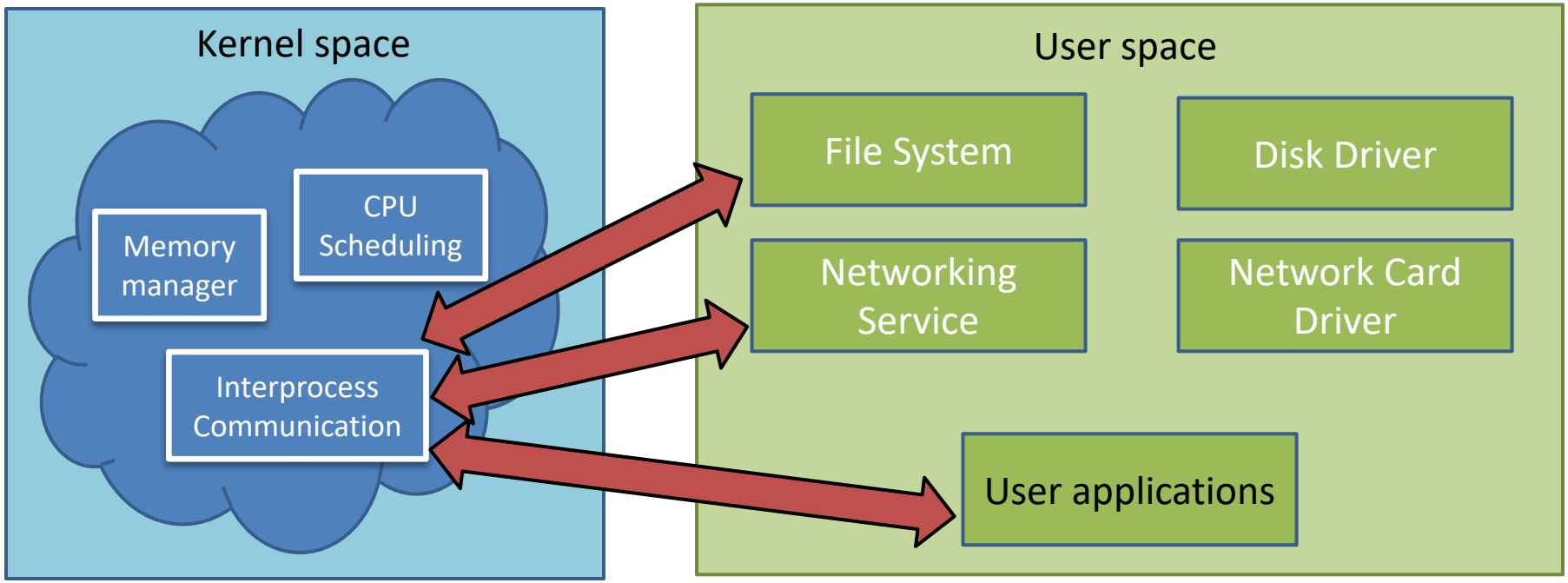
- Advantages
  - Single code base eases kernel development, simple



  - 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)

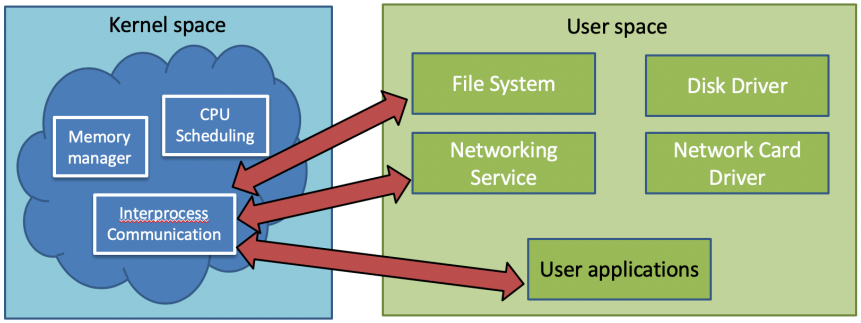
Microkernel

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



Pros/Cons of Microkernels

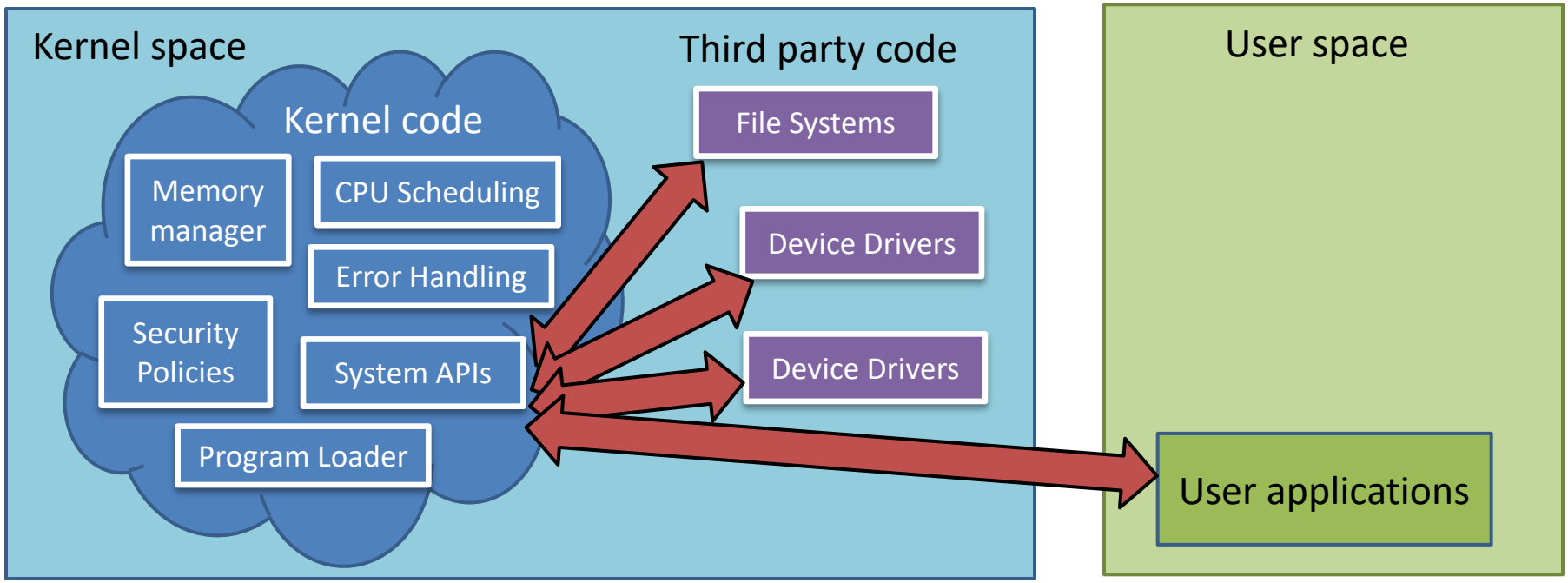
- Advantages
  - Small code base, easy to check for correctness
    - Excellent for high-security systems
  - Extremely modular and configurable
    - Choose only the pieces you need for embedded systems
    - Easy to add new functionality (e.g. a new file system)
  - Services may crash, but the system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

Google fuchsia

Research Kernels:
Mach L4 GNU
Hurd

Kernels for
Embedded System:
QNX

Monolithic Kernels:
Huge code base,
Many features

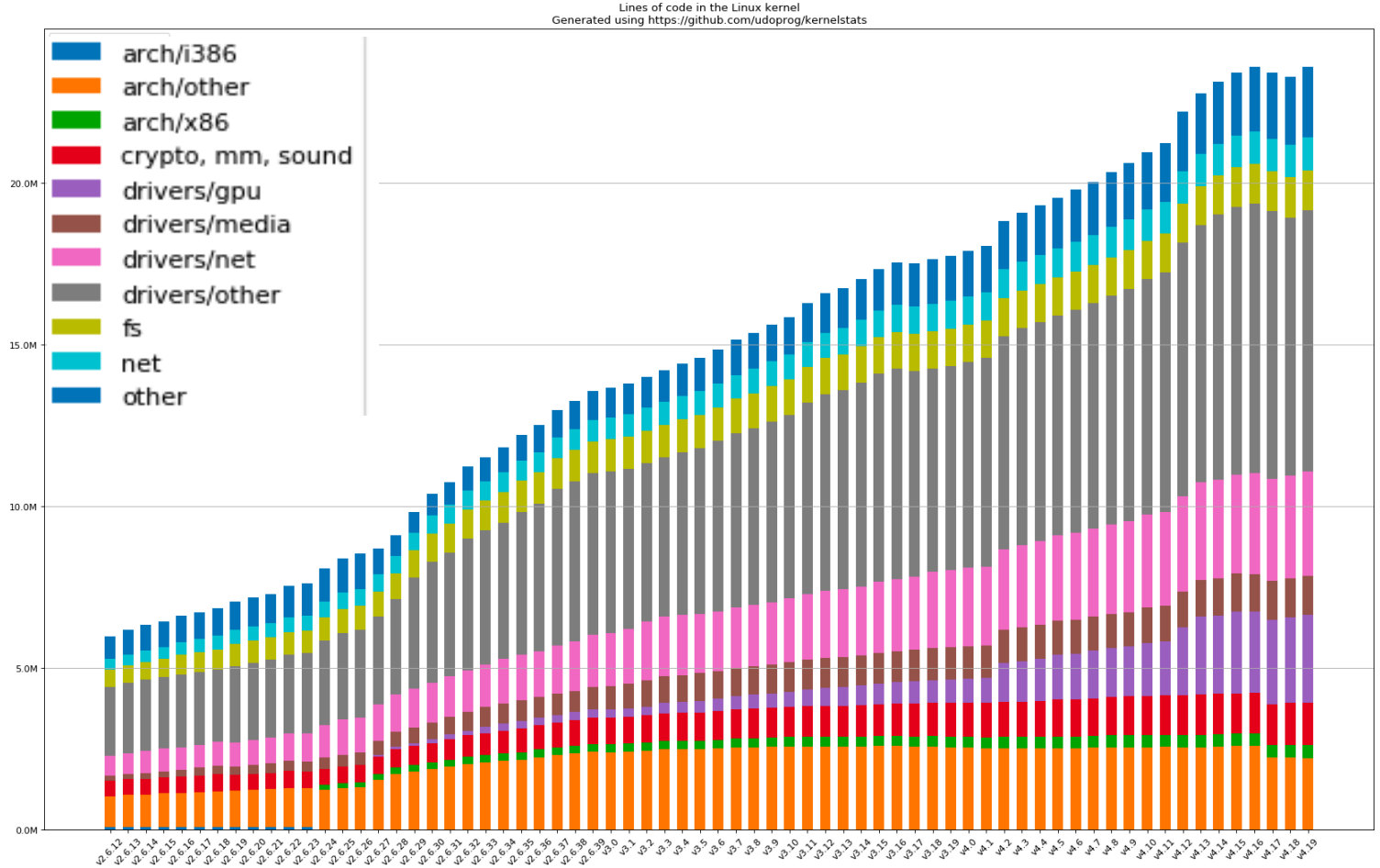
Hybrid Kernels:
Pretty large code base,
Some features delegated

Microkernels:
Small code base,
Few features

Examples

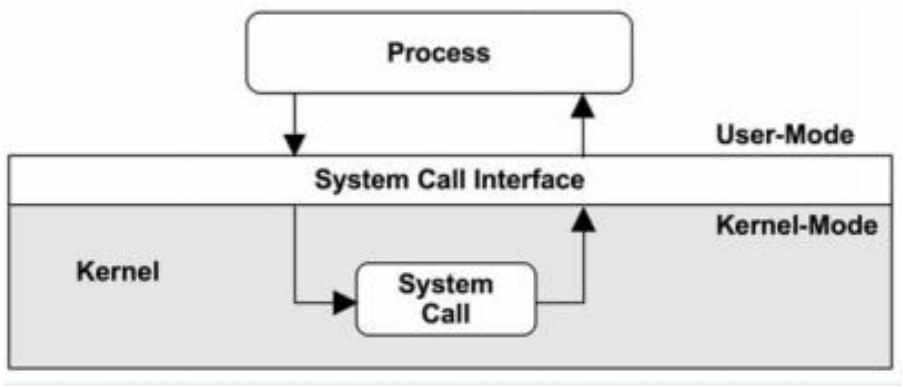
- The most complex software
  - ~ 20+ million lines of code in Linux

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