Lecture 05 - Intro to OS

ENSF461 - Applied Operating Systems

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)
Slides by Lorenzo De Carli

Review of previous lecture

What is C?

- **Programming language** first proposed by Dennis Ritchie in the early ‘70s
- Why “C”? It was derived from an earlier language called “B”
- Characteristics:
  - **General-purpose**
  - **Compiled**
  - **Strongly-typed**

C’s basic syntax

There are four program sections you need to worry about

```c
#include <stdio.h>

void do_stuff();

int main() {
  do_stuff();
}

void do_stuff() {
  printf("Hello, world!");
}
```

Preprocessor directives

Functions/globals declarations

Main function (entry point)

Function definitions

Compiling C programs with gcc: the basics

```
gcc  -o  <executable file name>  <source file(s)>
```

Compiler command

Set the name of the generated executable

Source file(s) to be compiled

Pointers & memory access in C

```
#include <stdio.h>

void value_or_reference(int val, int* ref) {
    val = 42;
    *ref = 42;
}

int main() {
    char myvec[4] = {'a', 'b', 'c', 'd'};

    printf("Position 0 of myvec is %c\n", myvec[0]);
    printf("Position 1 of myvec is %c\n", *(myvec+sizeof(char)));

    int byvalue = 2;
    int byref = 2;
    value_or_reference(byvalue, &byref);

    printf("Variable passed by value: %d\n", byvalue);
    printf("Variable passed by references: %d\n", byref);

    return 0;
}
```

But back to the lecture at hand

Goals for this semester

- Learn basic **OS concepts**:
  - Processes and process execution
  - Virtual memory
  - Concurrency
  - **Additional topics:** Storage, I/O, Networking, Security
- (Personal stretch goal: finish Tears of the Kingdom)

What does an OS do?

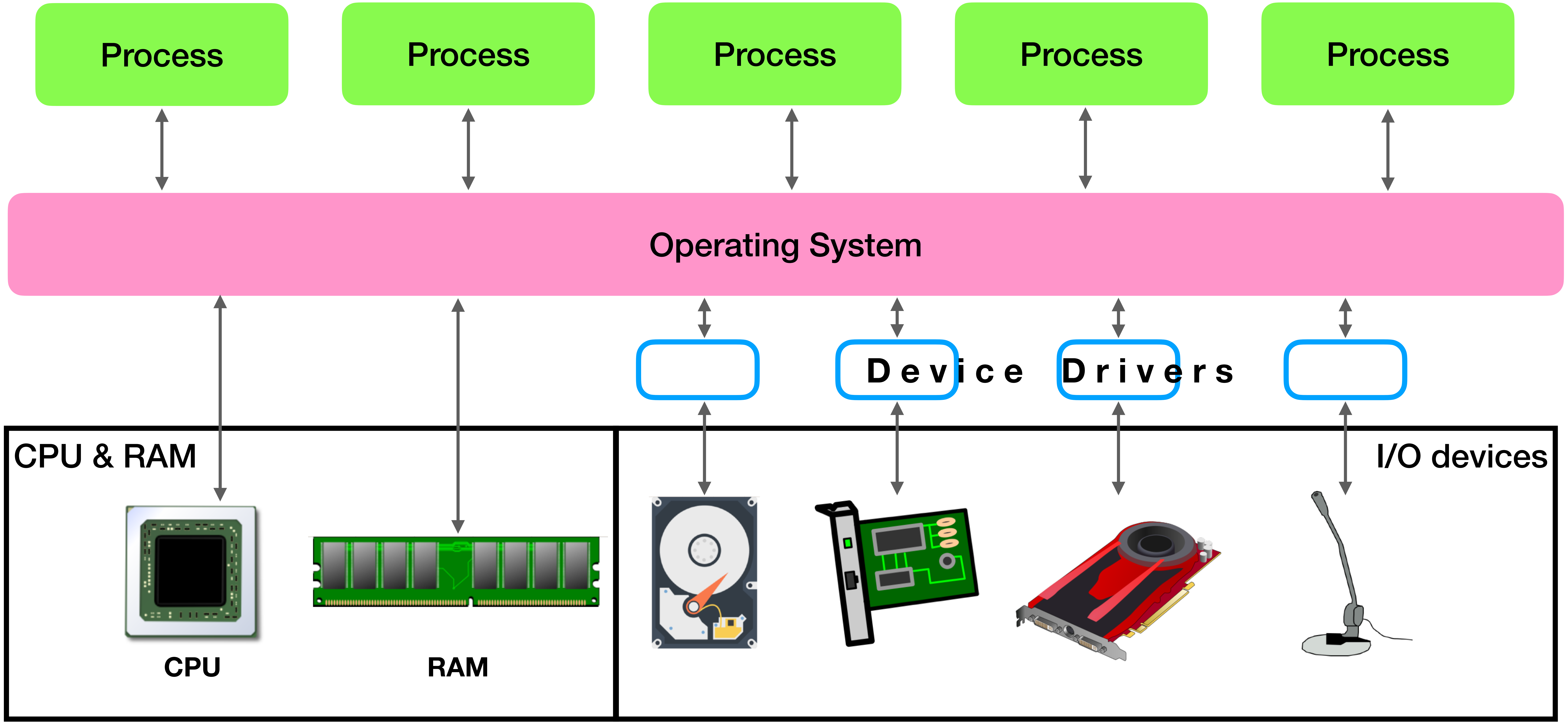
Let’s hear some ideas

What does an OS do?

Let's hear some ideas

- There are really three main things:
  - **Virtualize resources**, so every process can have the illusion of a dedicated machine w/ CPU and memory
  - **Manage I/O** so multiple processes can access devices (disk, network card, etc.) concurrently without interfering
  - **Make the above easy** - define simple, unambiguous interfaces (syscalls)

Conceptually, how does an OS look like?



What’s missing from the previous slide?

Efficiency!

- One goal of the issue is to **mediate** access to machine resources so…
  - Processes **do not interfere** with each other
  - Processes **can’t interfere** with each other (security!)
- But also, mediation should be in such a way that the OS creates **as little overhead as possible** (cannot directly intercept every I/O operation, CPU instruction, memory access)
- Figuring out how to do this properly took decades of engineering!

A bit of history
Early 1950s

- Early machines were mostly used for **scientific computing** and/or **processing large dataset**
- **Batch mode**: submit a large computation, wait for result, submit another one
- Basically a very powerful calculator used by **one operator at a time** - no need for OS

IBM T04 - https://images.nasa.gov/details/LRC-1957-B701_P-00989

A bit of history/2

- Well, truth to be told even early machines had **something** which is now considered part of any OS
    - **Libraries of functions** implementing **common I/O functions**
- The reason was really **convenience** - preventing everyone from having to implement tedious and tricky code
- **No virtualization** - still one program at a time
- **No mediation** - any program could still do whatever they wanted

A bit of history/3

Early 1960s

- The next breakthrough in OS design was **separation**
- Idea: the stuff that an OS does is **sensitive**
  - Just giving the **option** to programs to use an OS does not seem enough
  - Each program **should go through the OS** to access hardware
  - **But how to implement this?**

Implementing separation

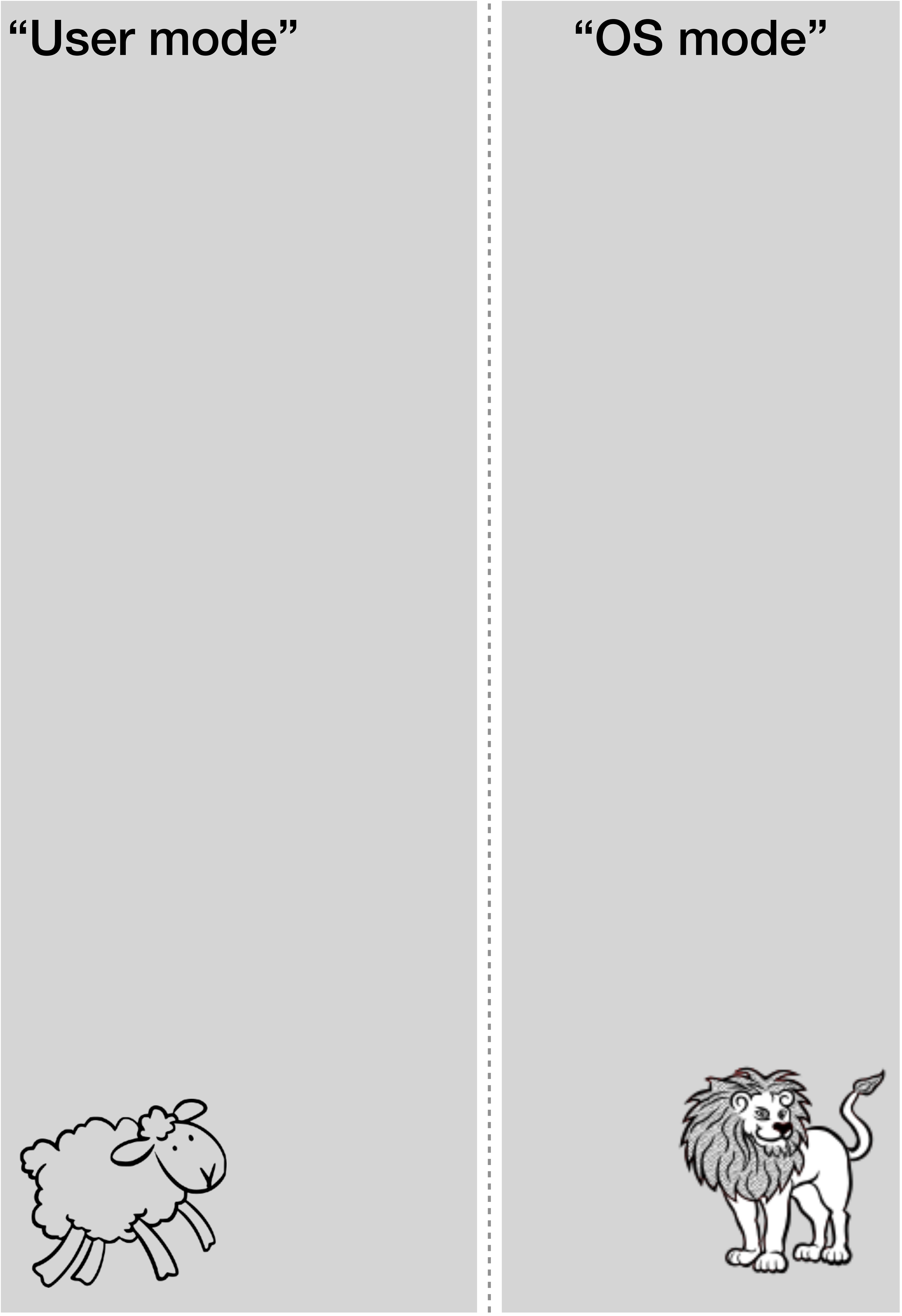
- The idea is to give the CPU (which in practice controls the machine) at least **two execution modes**:
  - An **unprivileged execution mode** for regular processes
  - A **privileged execution mode** for OS code
- **How to switch between the two? How to prevent misuse?**

Implementing separation/2

- Typically, the CPU architecture defines some type of **trap** - an instruction to switch into OS code, and back
- **To be clear:** this allows a process to "jump" into and continue execution into OS code
  - **It does not** allow a process to run its own code as OS
  - The app can pass **execution and parameters to OS**, but cannot define the code that the OS is going to execute

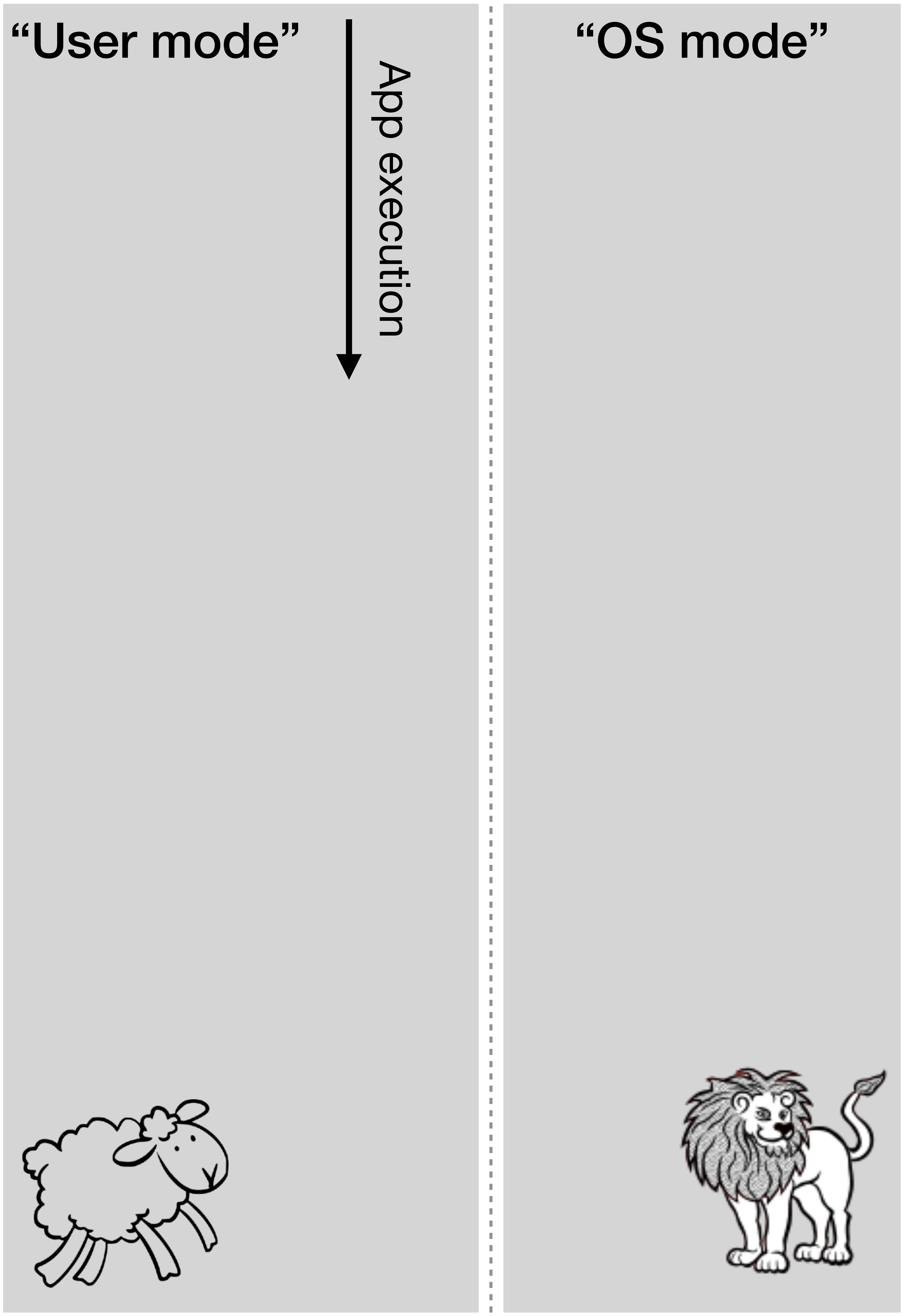
Implementing separation/2

- Typically, the CPU architecture defines some type of **trap** - an instruction to switch into OS code, and back
- **To be clear:** this allows a process to “jump” into and continue execution into OS code
  - **It does not** allow a process to run its own code as OS
  - The app can pass **execution and parameters to OS**, but cannot define the code that the OS is going to execute



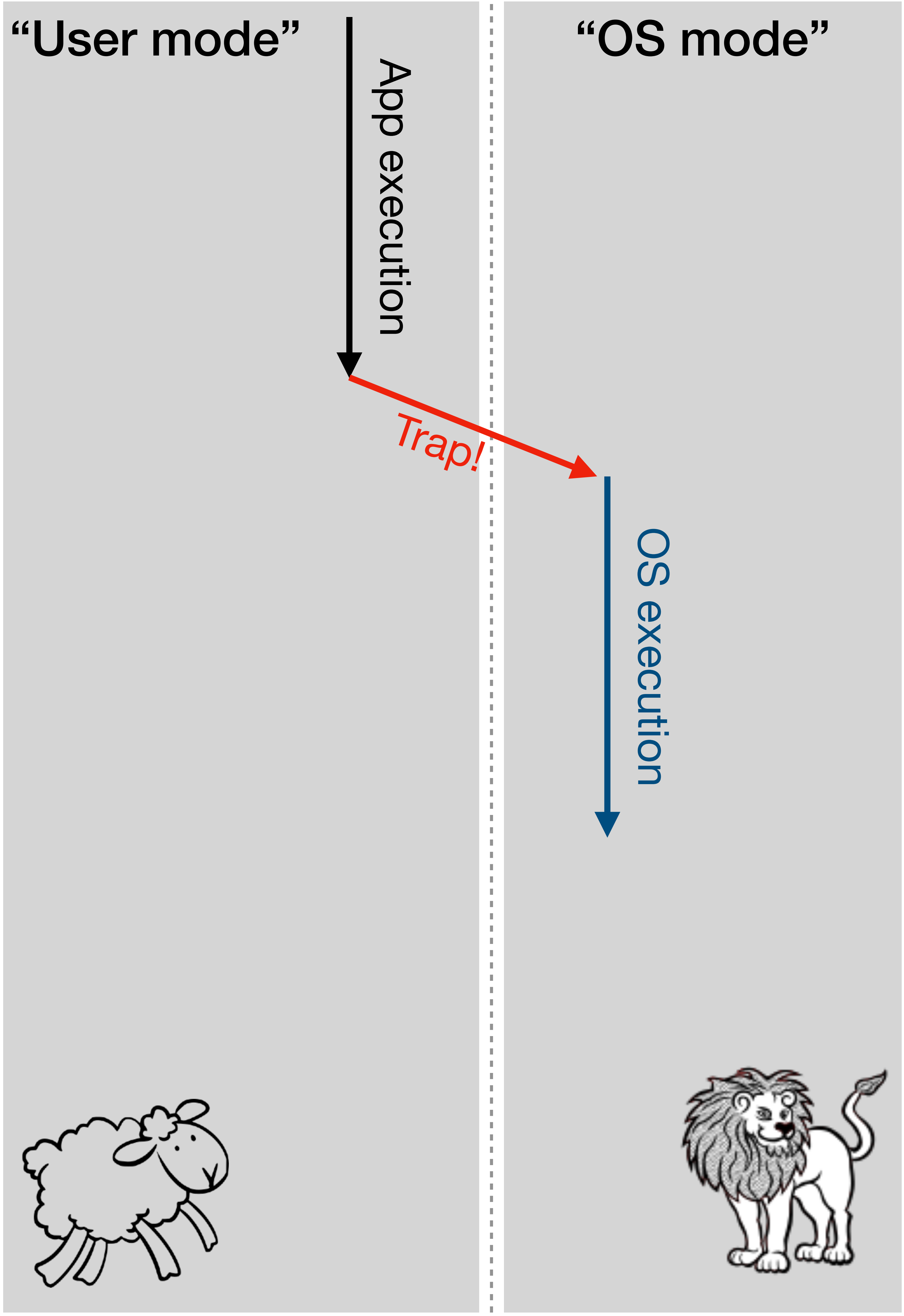
Implementing separation/2

- Typically, the CPU architecture defines some type of **trap** - an instruction to switch into OS code, and back
- **To be clear:** this allows a process to “jump” into and continue execution into OS code
  - **It does not** allow a process to run its own code as OS
  - The app can pass **execution and parameters to OS**, but cannot define the code that the OS is going to execute



Implementing separation/2

- Typically, the CPU architecture defines some type of **trap** - an instruction to switch into OS code, and back
- **To be clear:** this allows a process to "jump" into and continue execution into OS code
    - **It does not** allow a process to run its own code as OS
    - The app can pass **execution and parameters to OS**, but cannot define the code that the OS is going to execute



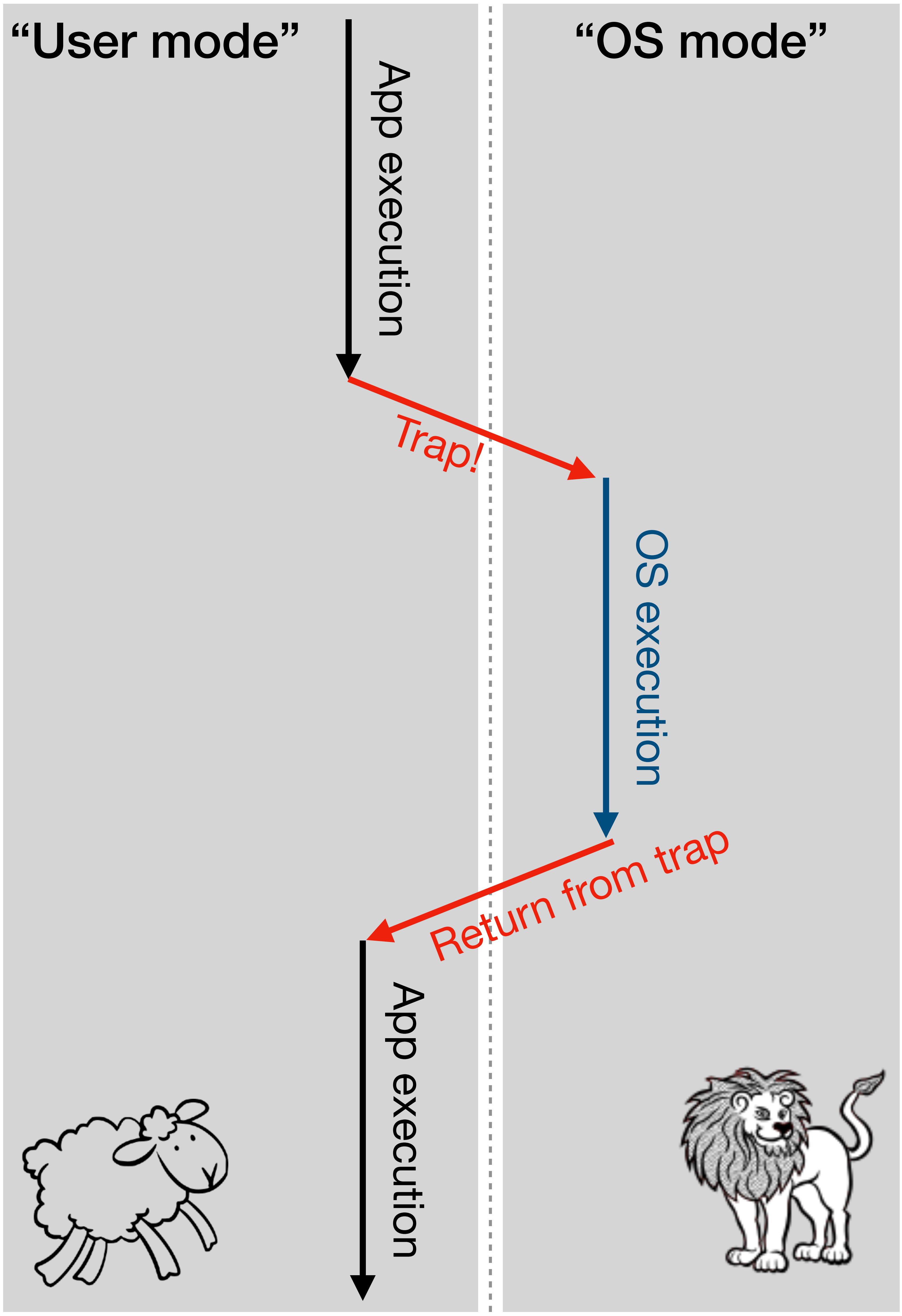
Implementing separation/2

- Typically, the CPU architecture defines some type of **trap** - an instruction to switch into OS code, and back

- **To be clear:** this allows a process to “jump” into and continue execution into OS code

  - **It does not** allow a process to run its own code as OS

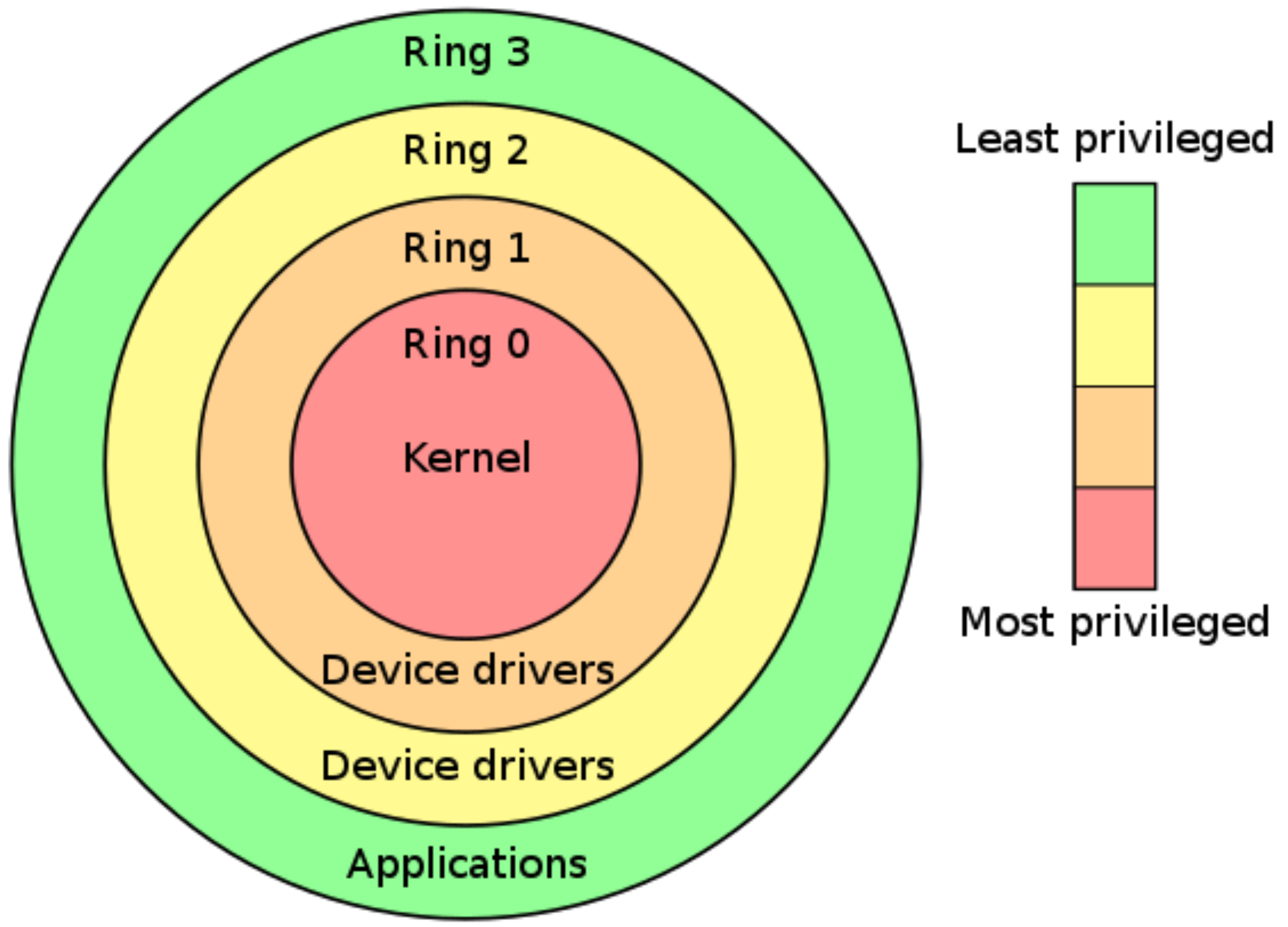
  - The app can pass **execution and parameters to OS**, but cannot define the code that the OS is going to execute



While limiting ourselves to two modes?

The ring of fire

- x86 for examples defines four **rings**
- **Goal:** more flexibility/security
- In practice (Windows, Linux):
  - Use **Ring 0** for **OS** (kernel)
  - Use **Ring 3** for **processes** (applications)
- **Why do you think is that?**



https://en.wikipedia.org/wiki/Protection_ring

More about separation

Sharing storage resources

- Another reason to implement OS as a privileged layer is **storage**
- The OS-as-library abstraction only works if **one process** accesses the **disk**
- If multiple processes write to disk without coordination, **mayhem**!
- For that reason, it is best that **something coordinate access to disk** (and other resources to - e.g., network)

A bit of history/4

Multiprogramming - we are now in the 1970s

- Many processing tasks were (and still are) I/O bound
    - **Can you explain that to me?**
- If you only run **one process at a time**, your (very expensive) CPU is **sitting idle** most of the time
- Solution: **let multiple processes run at the same time,** switch between them when appropriate (e.g., when one is idle waiting for data to be loaded)

So, how do we do this?

This one weird trick enables multiprogramming

- Several things, but the most important is probably **memory protection**
- **Harder than it looks**
  - You could just force processes to “stay” in different memory areas… but this creates more problems than it solves
  - Solution: **virtual memory** (much, much, much more on this later)
- **UNIX** was the first widely available OS which (eventually) implemented storage (file system), time sharing (splitting CPU time), etc.

Cue in the 1980s

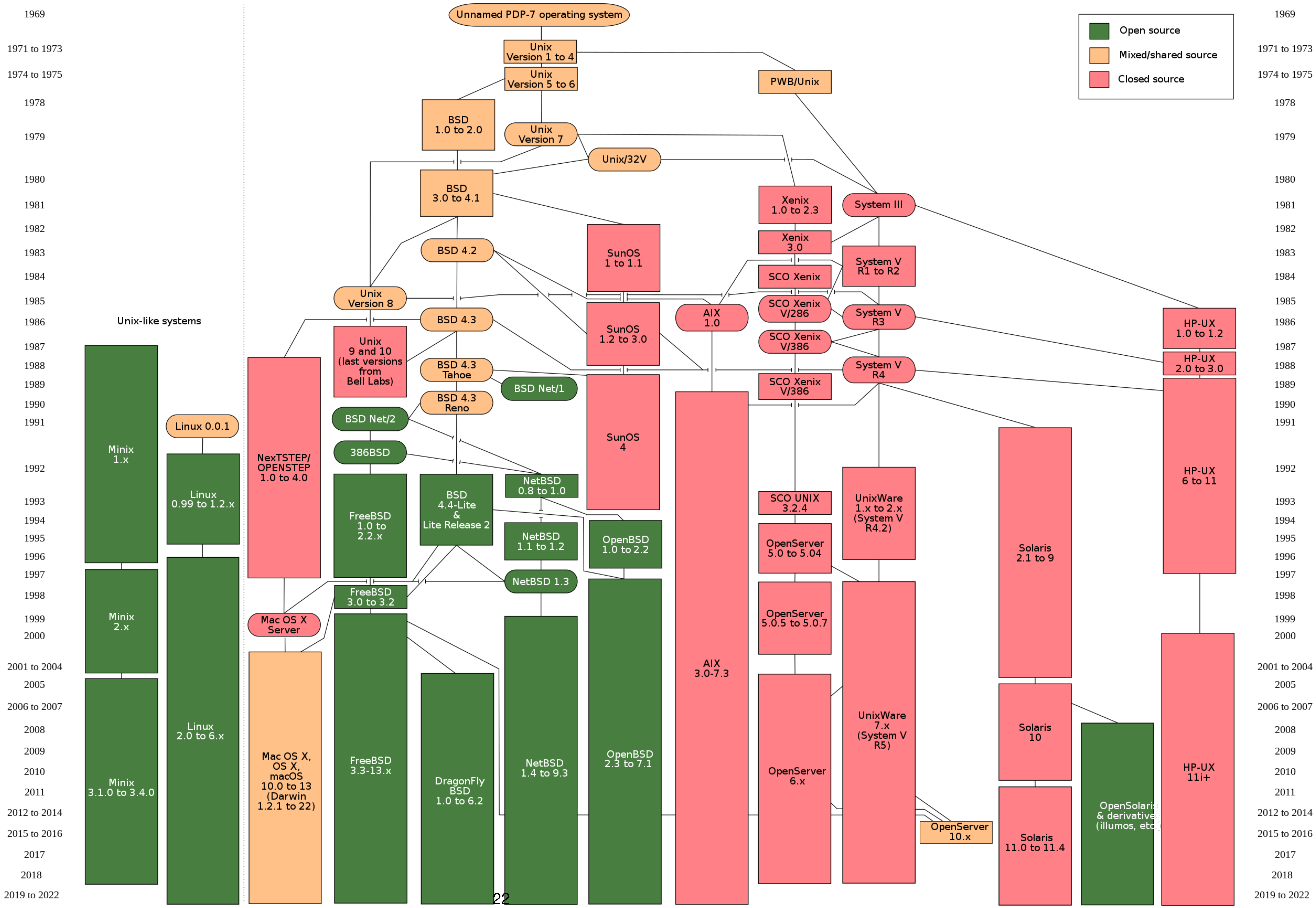
Personal Computers

- Starting in the late 70s/early 80s, companies like IBM and Apple started selling **personal computers** - small machines intended for home/office use
- Those shipped with very simple OS'es, oftentimes **lacking proper protection** and/or **multiprogramming capabilities**
- Examples: **MS DOS** - no memory protection; **Apple Mac OS (the old one)** - cooperative threads (a stuck program means a stuck machine)
- The "tradition" of poorly designed consumer OS'es lasted well into the 1990s

1990s to today

- Starting in the 1990s, and partly thanks to the influence of Linux (a Open-Source UNIX descendant), **mainstream OSes improved significantly**:
  - True **memory protection** capabilities
  - True **multiprocessing/threading** (the OS controls how processes share the CPU)
  - Robust, performant **file systems**
- **Curiosity:** MacOS, Linux, Android all come from the **UNIX** family tree. Windows is a bit of a different beast.

UNIX family tree



Quiz time!

D2L-> ENSF461 -> Assessment -> Quizzes -> Quiz 05

CPU virtualization

CPU virtualization

…or the art of “slicing” a CPU

- An OS should offer the ability to:
  - **Run** a specific program
  - **Stop** a program
- It should also:
  - Decide **which program should run** at any given time
  - **Divide CPU time** across programs “**fairly**” (**what does it mean?**)

Example (from the book)

Let’s see it in action…

```
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;
}
```

A couple of points

- First, ask yourself how the two programs can run concurrently (yes, your laptop has > 1 CPU cores, but this will work even on a single-core machine)
- Also, let’s talk about that `#include “common.h”`

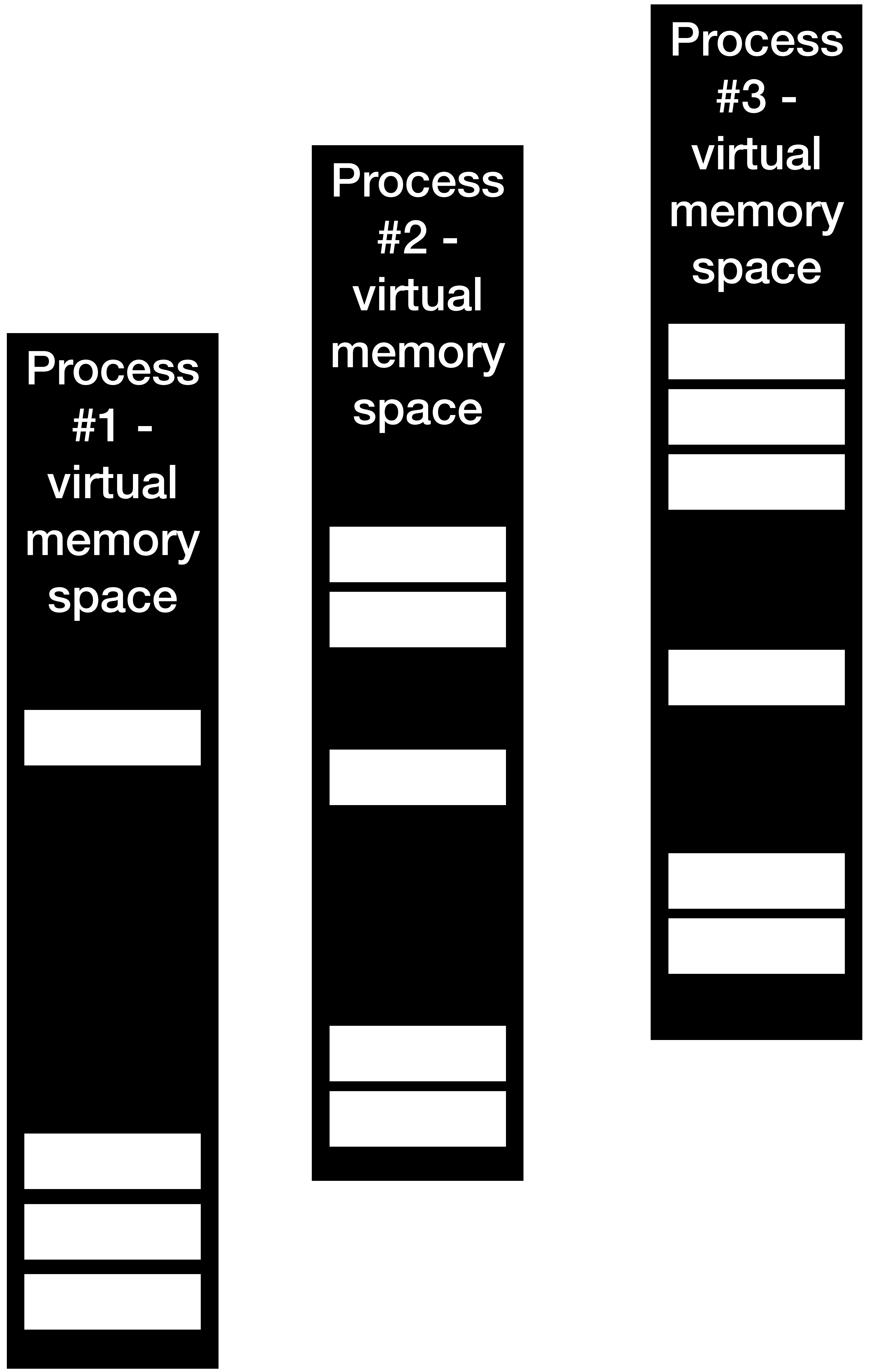
Memory virtualization

Memory virtualization

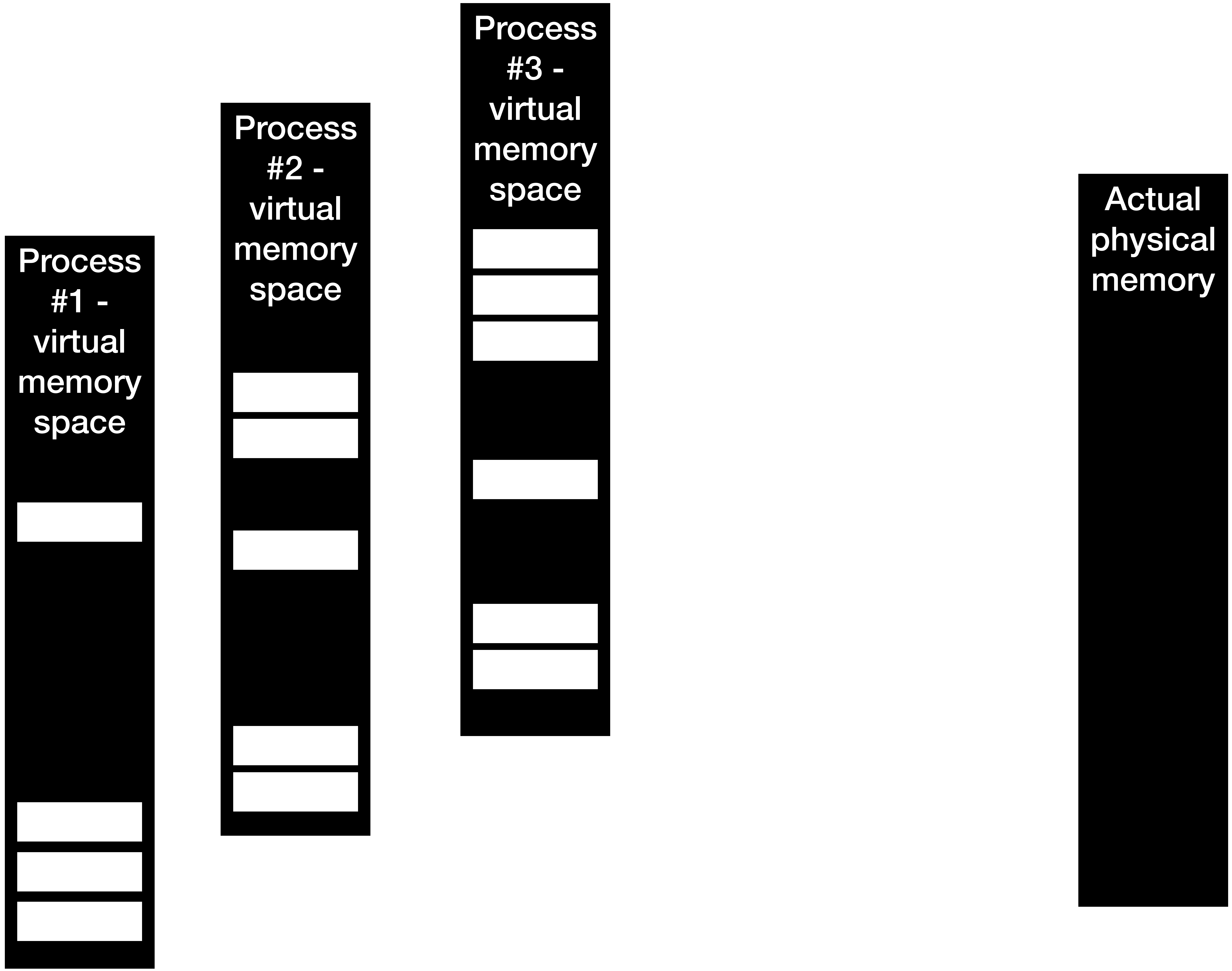
What is this sorcery?

- **Virtualizing memory** is actually (suprisingly?) difficult
  - Arguably more than virtualizing CPU
- Memory virtualization give every process the same **virtual address space**
  - In other words, each process gets the illusion of having memory to itself (the **size** of the virtual address space depends on **OS/architecture**)
- The virtual address space is somehow **mapped** to the **physical space**

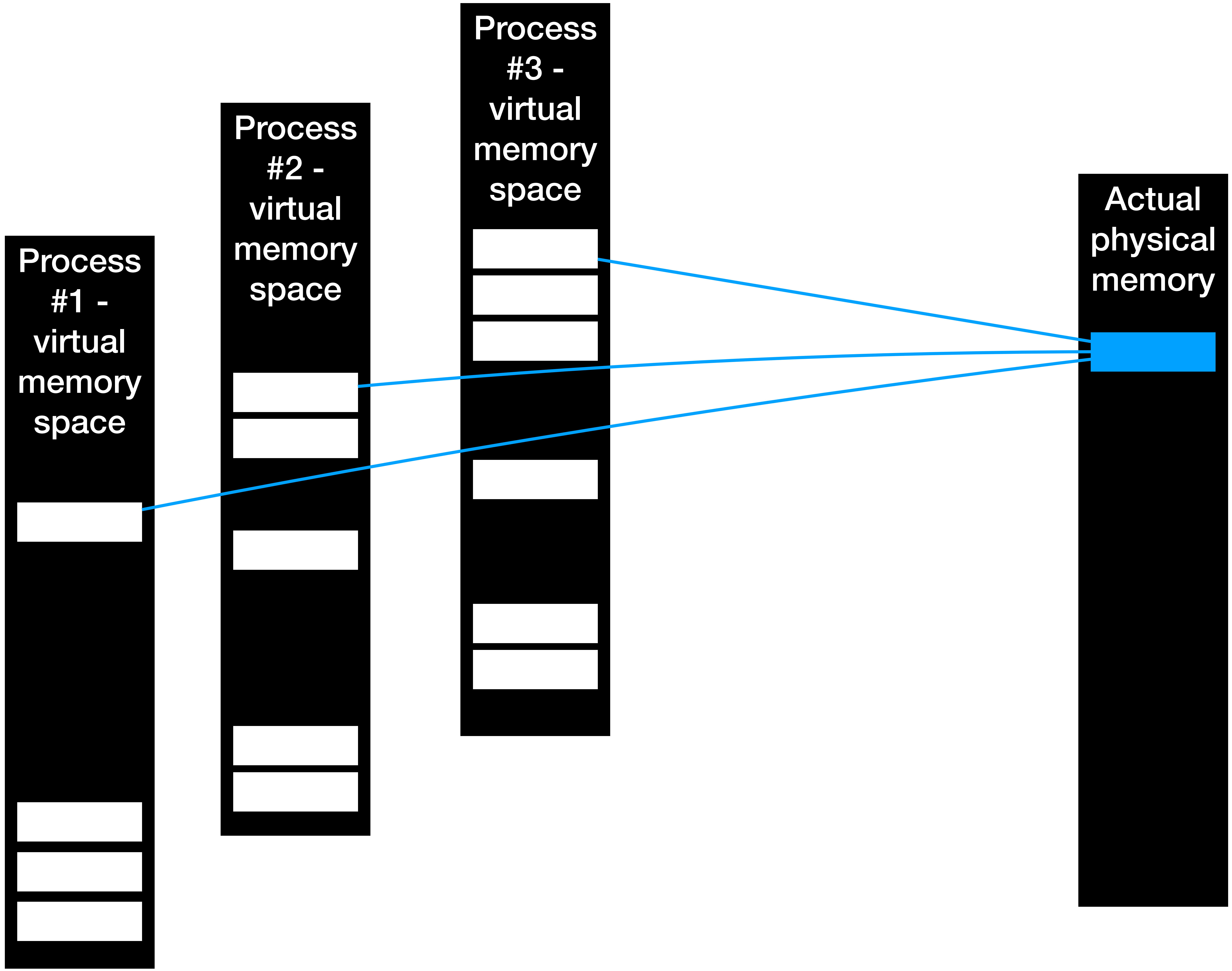
Memory virtualization /2



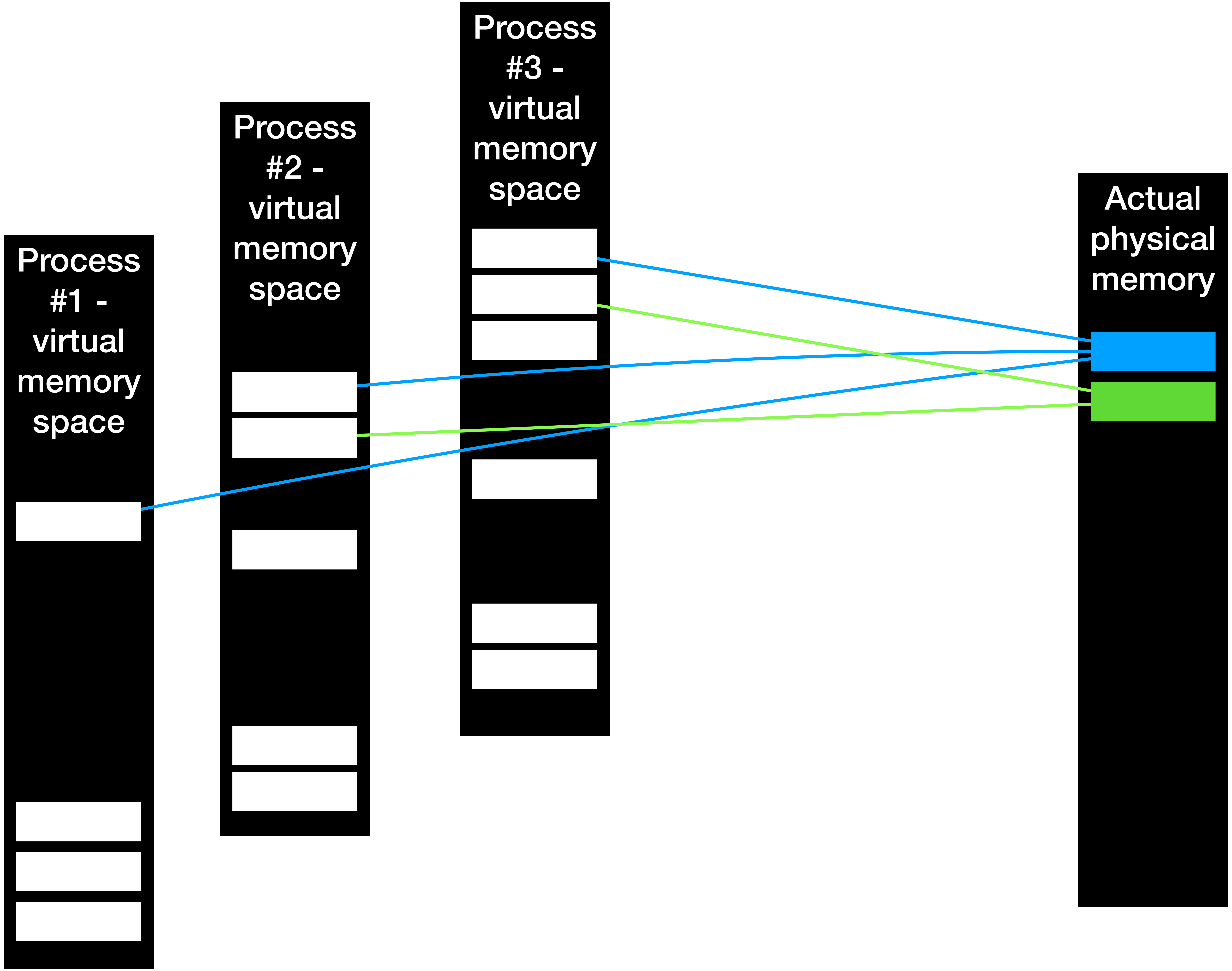
Memory virtualization /2



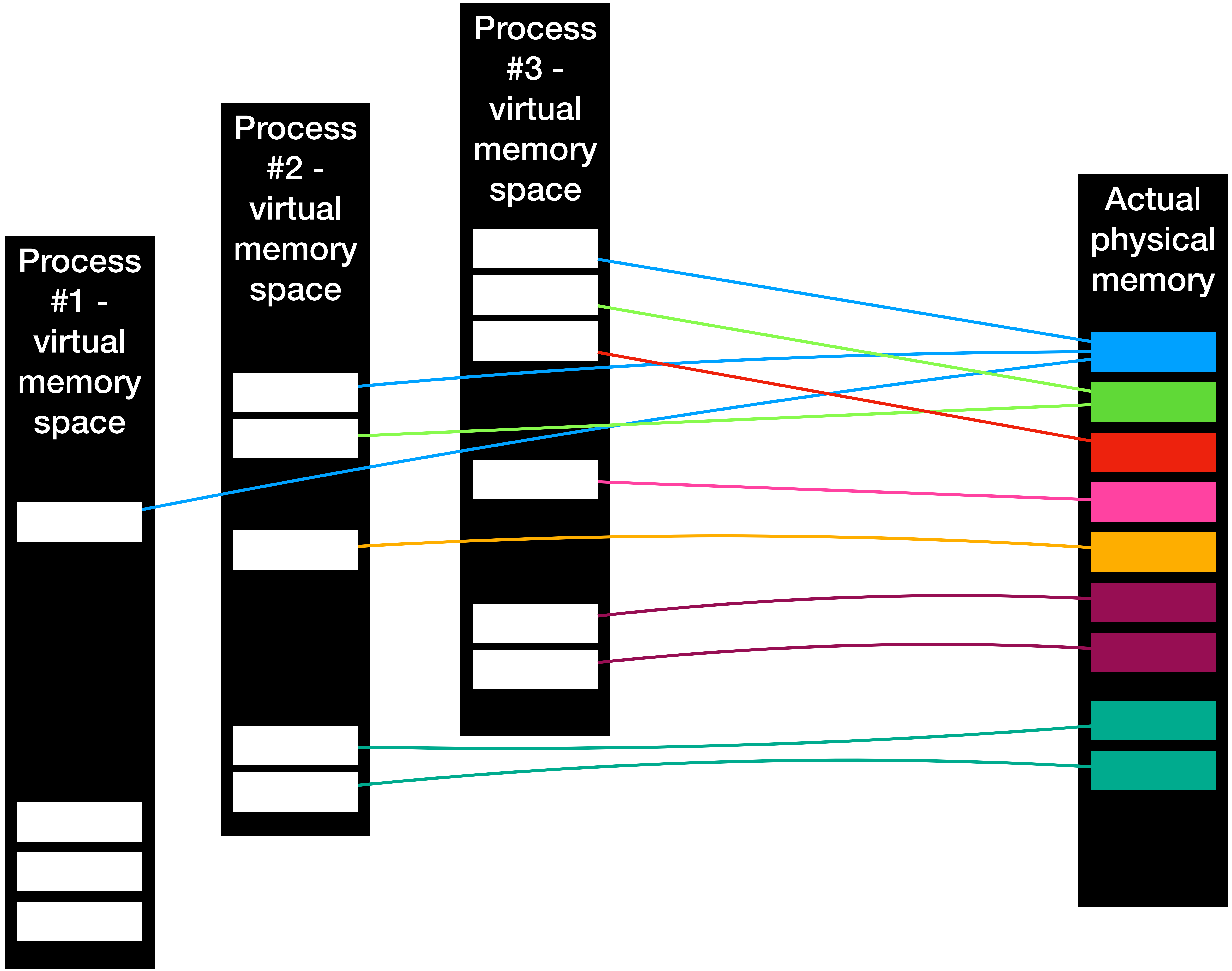
Memory virtualization /2



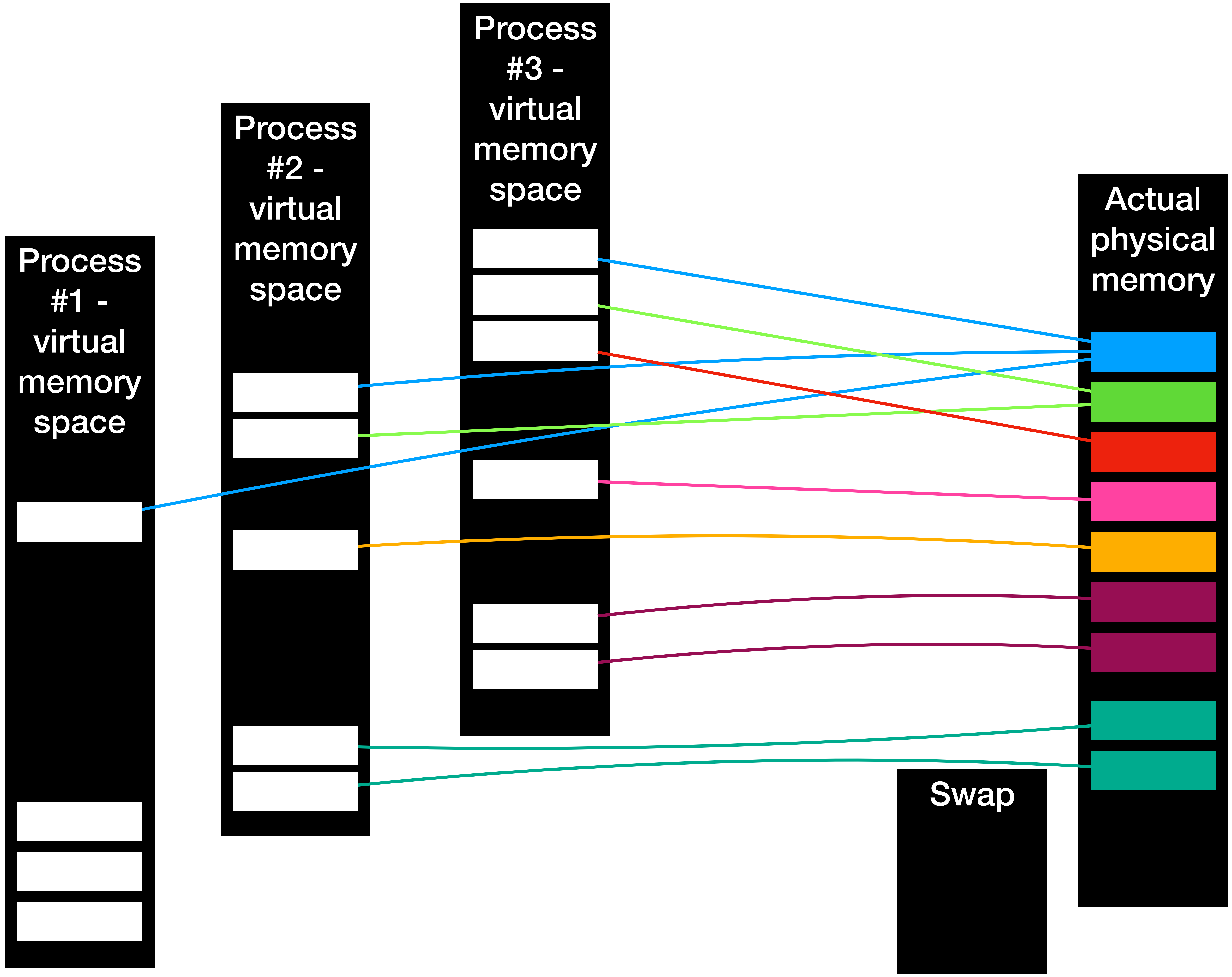
Memory virtualization /2



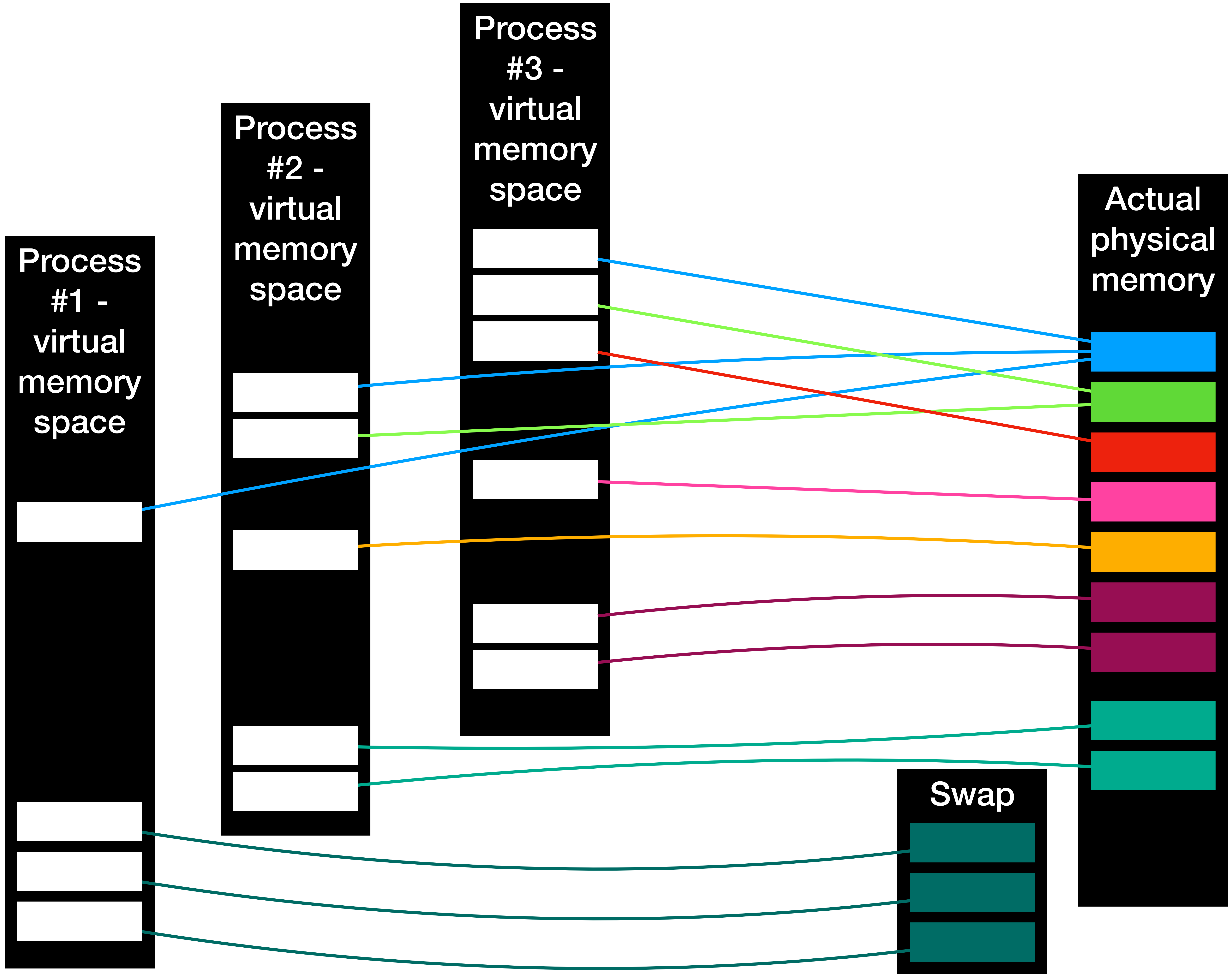
Memory virtualization /2



Memory virtualization /2



Memory virtualization /2



Memory virtualization…

is not static!

- Mapping between virtual and physical memory pages can **change**
  - For example, virtual memory pages can be **swapped**
- Mapping is also **not necessarily 1:1**
  - Example: **resources used by multiple processes**

Example (from the book)

```
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;
}
```

Let’s talk about malloc

Do you remember what it does?

Let’s talk about malloc

Do you remember what it does?

- **Dynamic memory allocation**
- **Allocates a bunch of memory** and **returns a pointer** to the beginning of that region
    - **Physical or virtual?**
- The memory should be freed using **`free()`** when no longer needed
- You may also want to make the acquaintance of their cousins **`calloc()`** and **`realloc()`**

Concurrency

What is concurrency?

And how is it different from sharing CPU?

- **CPU virtualization** just means that multiple independent processes can share the CPU
- In many cases, you want **different parts** of your program **cooperate** to **complete a task**
  - **Can you think of some examples?**
  - **What would be necessary to enable this?** (assume the OS already give you the capability to run multiple programs on the same CPU)

Concurrency /2

What are the requirements?

- **Concurrency** (commonly) requires **two capabilities**:
  - The ability for different “parts” of the program to **communicate**
  - The ability to **synchronize** access to **shared data**
  - (The second is really a specialized case of the first)

Concurrency /3

- Partly for historical reasons, “parts” of a concurrent program can take **two forms**:
  - Different communicating **processes** (separate address spaces)
  - **Threads** within the same process (shared address space)
  - We’ll talk much more about this stuff later

Example (from the book)

(Not very significant, as threads do not communicate)

```
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;
}
```

Let’s talk about this

You have seen an example of **threads**

- **`pthread_create()`**: creates a new execution threads, pass the execution to a specified function
- **`pthread_join()`**: wait until a specified thread terminates

Is there more to OS’es?

You bet!

Persistence

AKA storage

- Processes typically need to **receive some pre-existing** inputs and **save some outputs**
- This is accomplished using **persistent storage (disks, SSD)**
- No virtualization abstraction (it does not make sense to give each program the abstraction of a “dedicated disk”, as **programs often share data**)
- Instead, **shared disk** using **files** and **directories** (foldes) **abstraction**

Compare the two abstractions

Virtual memory VS Shared disk

- **Virtual memory:** works because processes mostly need their own memory space, and only occasionally need to share memory data
- **Shared disk:** works because it is very common for programs to use or share disk data

But wait! There is more

- Disk is not the only I/O device the OS must manage
- **Some other examples:**
  - **Graphics output** (window-based UI, Vulkan/Direct3D/Metal, etc.)
  - **Network interface devices** (e.g., WiFi)
- OS is also responsible to **enforce security**. This means many things, e,g.:
  - Ensure processes **do not interfere**
  - Ensure no user process can **execute stuff** with **OS privileges**

That’s all!