Operating Systems

Introduction, Processes and Virtualisation

Lecture Overview

- Housekeeping (Assessment)
- Why Operating Systems?
- Processes
- Virtualisation

Housekeeping

Administration!

The Team

Course Coordinator

- Me!
- IW 5.44
- bernard.evans@adelaide.edu

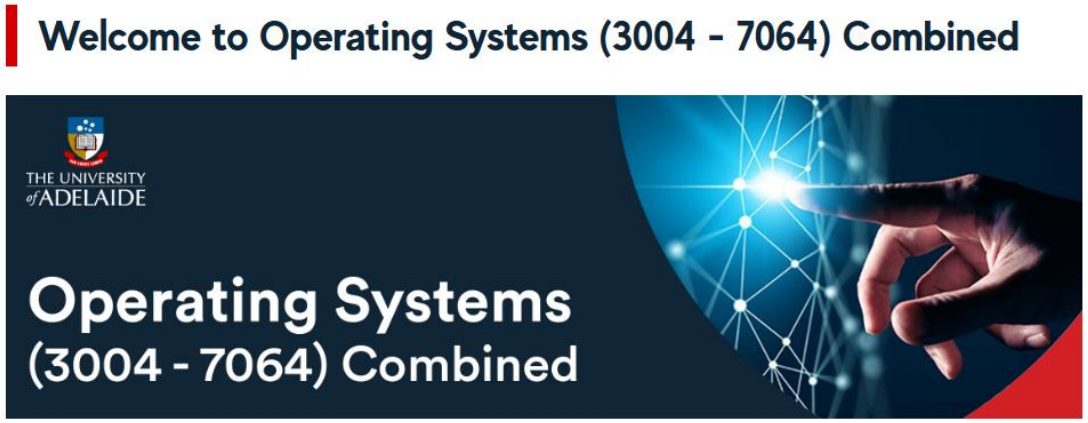
Workshop Supervisors

- Anubhav Gupta
- Jyothis Joy
- Nithin Dharanu
- Arjun Sharma

Resources

My Uni

- Echo360
- Lecture Notes
- Piazza



Prerequisites and Assumed Knowledge

Pre-requisite

Algorithm Design and Data Structures

OR

Programming for IT Specialists

OR

Foundations of Computer Science B

Means

Coding, stacks, queues, trees etc

Assumed Knowledge

Computer Systems

- Logic gates
- Registers, Program Counters, Memory
- Stack
- Assembly
- Compilation

Prerequisites and Assumed Knowledge

Pre-requisite

Algorithm Design and Data Structures

OR

Programming for IT Specialists

OR

Foundations of Computer Science B

Means

Coding, stacks, queues, trees etc

Assumed Knowledge (Kind of)

Systems Programming

- Processes
- Pipes
- File Descriptors
- System Calls
- C-Code

Course Learning Outcomes

- Explain the role of the operating system as a **high level interface to the hardware**.
- Use OS as a **resource manager** that supports **multiprogramming**
- Explain the low level implementation of **CPU dispatch**.
- Explain the low level implementation of **memory management**.
- Explain the **performance trade-offs** inherent in OS implementation

Course Learning Outcomes

Operating Systems

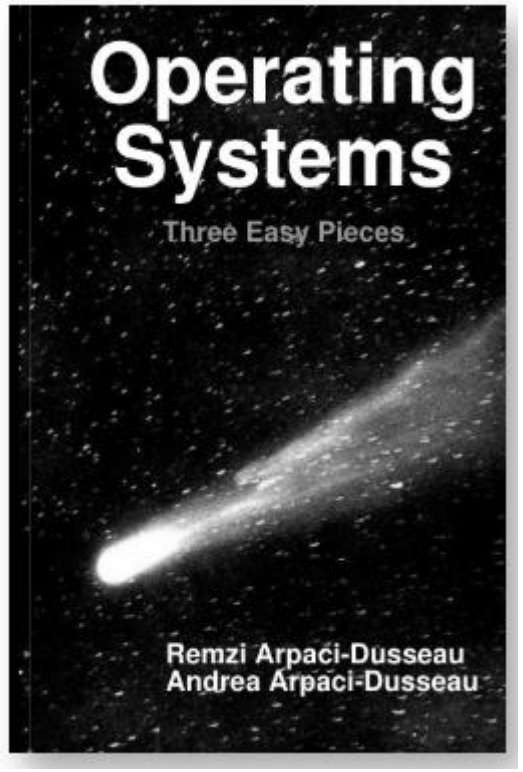
- Manages processes and hardware
- Allows multiprocessing
- CPU Dispatch
- Memory Management
- Performance Trade-offs

Every part of an Operating System is a design decision. There are **choices**.

Hints

Suggestions from previous students

- Read the textbook
- Start early, work consistently
- Practice the theoretical/numerical questions!



Hints

Suggestions from me

- **Read the textbook!**
- Watch and take notes during lectures.
- Do all the assessments.
- Ask for Help!
- Have fun!

Accurate representation of OS
(AI Generated, NightCafe Studio)

Assessments

How to gain marks

Assessment Components

Quizzes	**Total:** 6%
Tutorials	**Total:** 6%
Assignments	**Total:** 38%
Exam	**Total:** 50%

Quizzes

Online!

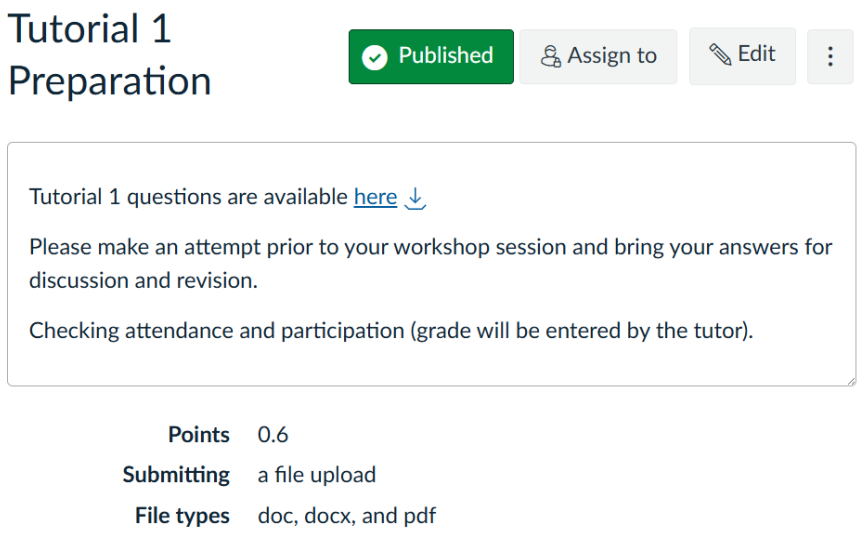
- There are four **quizzes**
- They are worth 6% (only your best three results will count)

Tutorials

Weeks: 3, 5, 7, 9, 11

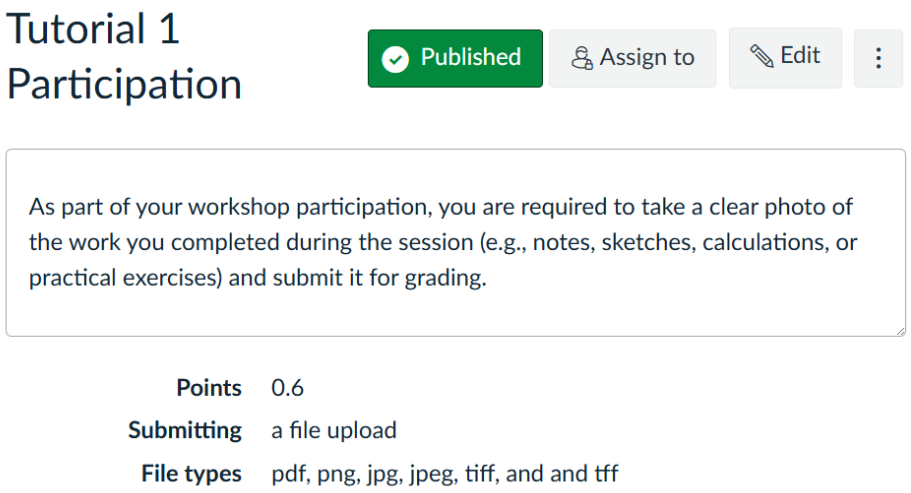
Preparation

- Online (must be done the week before the tutorial)



Participation

- In Person (uploaded during your tutorial)



Assignment #1: System Calls

Objectives

- C-coding
- System Calls
- Processes and **Signals**
- **Gradescope** submission
- Worth **8%** of final grade

- Due **Week 4**

Task #1: Handling Signals

Task #2: Improve a Shell

Assignment #2: Scheduling

Objectives

- Coding Activity
- Report
- Worth **15%** of final grade

- Due **Week 8**

Task #1: Implement Schedulers

Task #2: Evaluate Schedulers

Assignment #3: TBD

Objectives

TBD!

- Worth **15%** of final grade

- Due **Week 12**

Motivation

Why are we studying OS?

Motivation

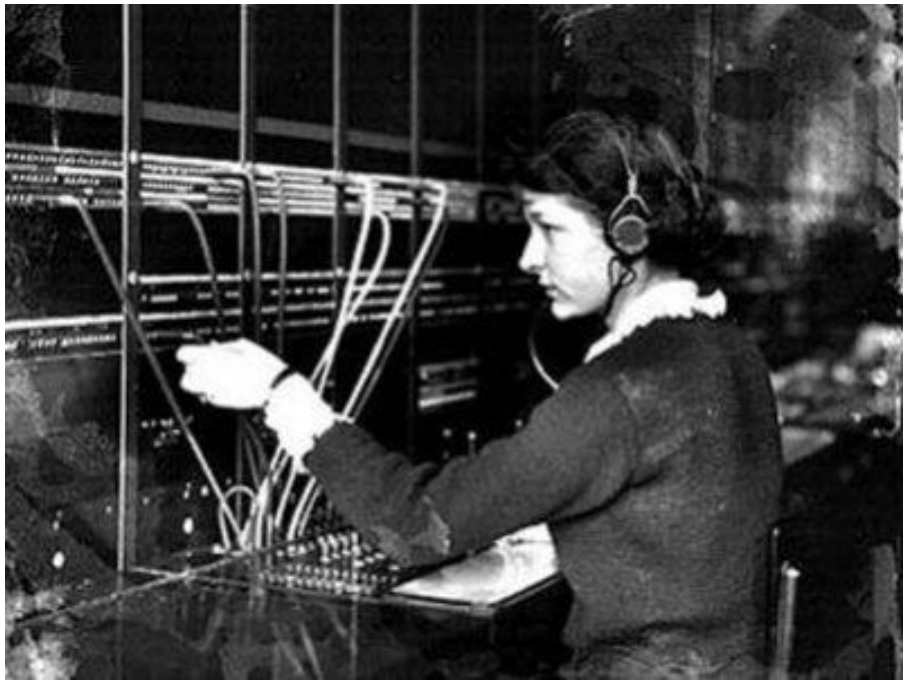
- ~~Learning C~~
- ~~Learning how to program~~

- Understanding how stuff actually happens
  - Memory
  - Processes
  - Registers
  - Concurrency

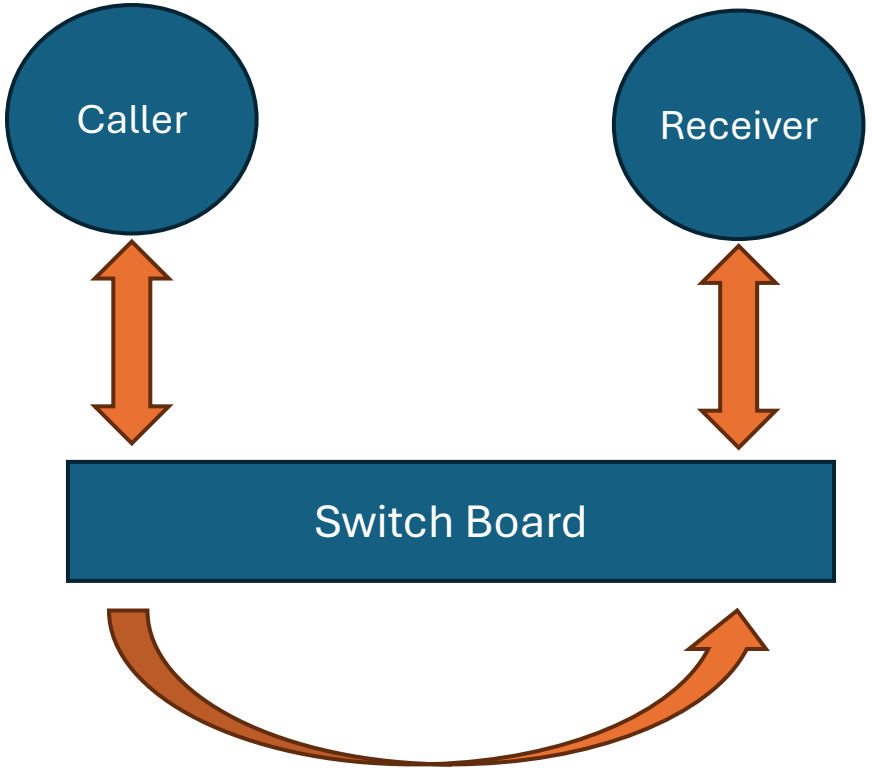
Definitions

What is an operating system?

Some History

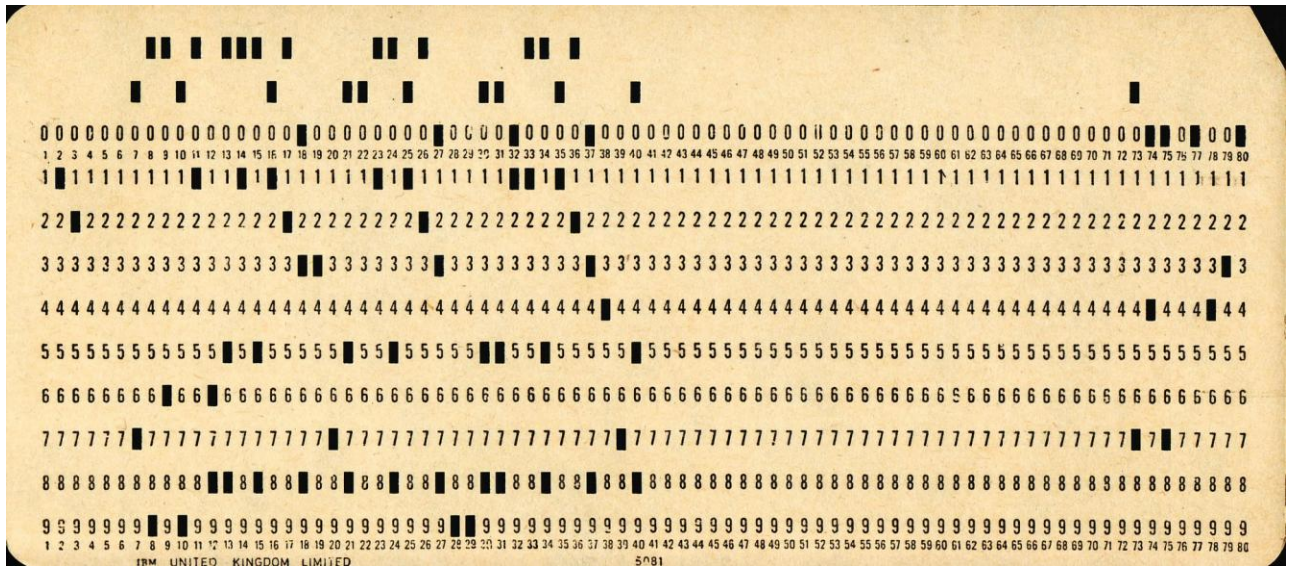


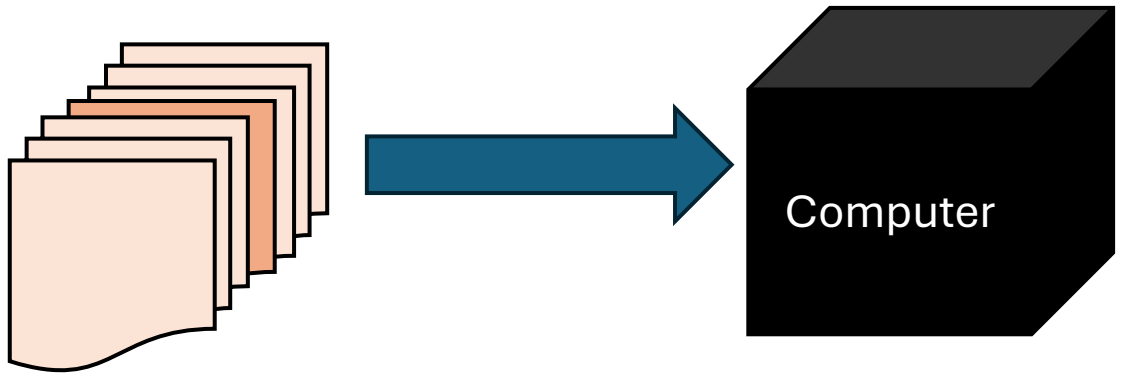
Switchboard Operator



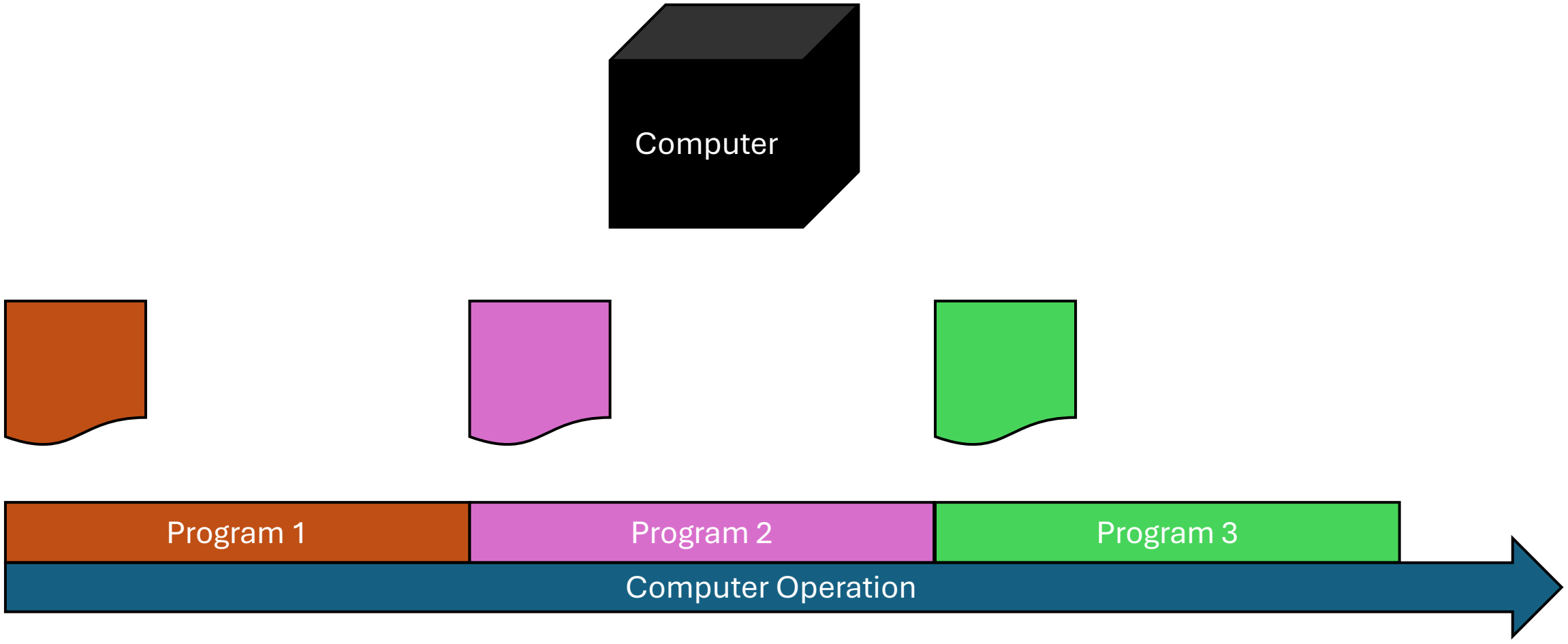
Some History

Computer Operator

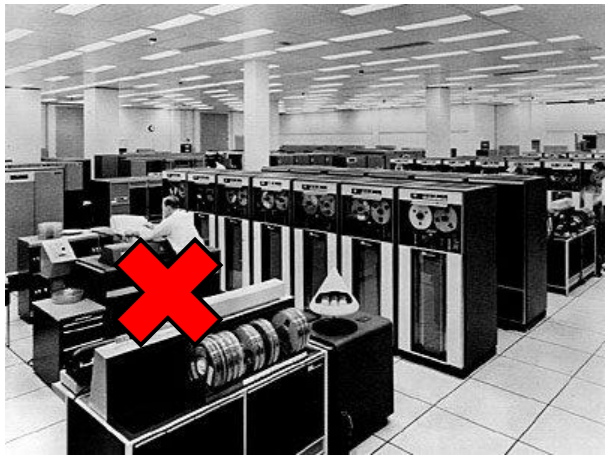




Some History



Some History



What do Operating Systems do?

Theoretically

- They abstract a human operator

What does a Human Operator Do?

- Optimises
  - CPU usage
  - Memory usage
  - Time
  - Security/reliability

Main Operating Systems

VMS

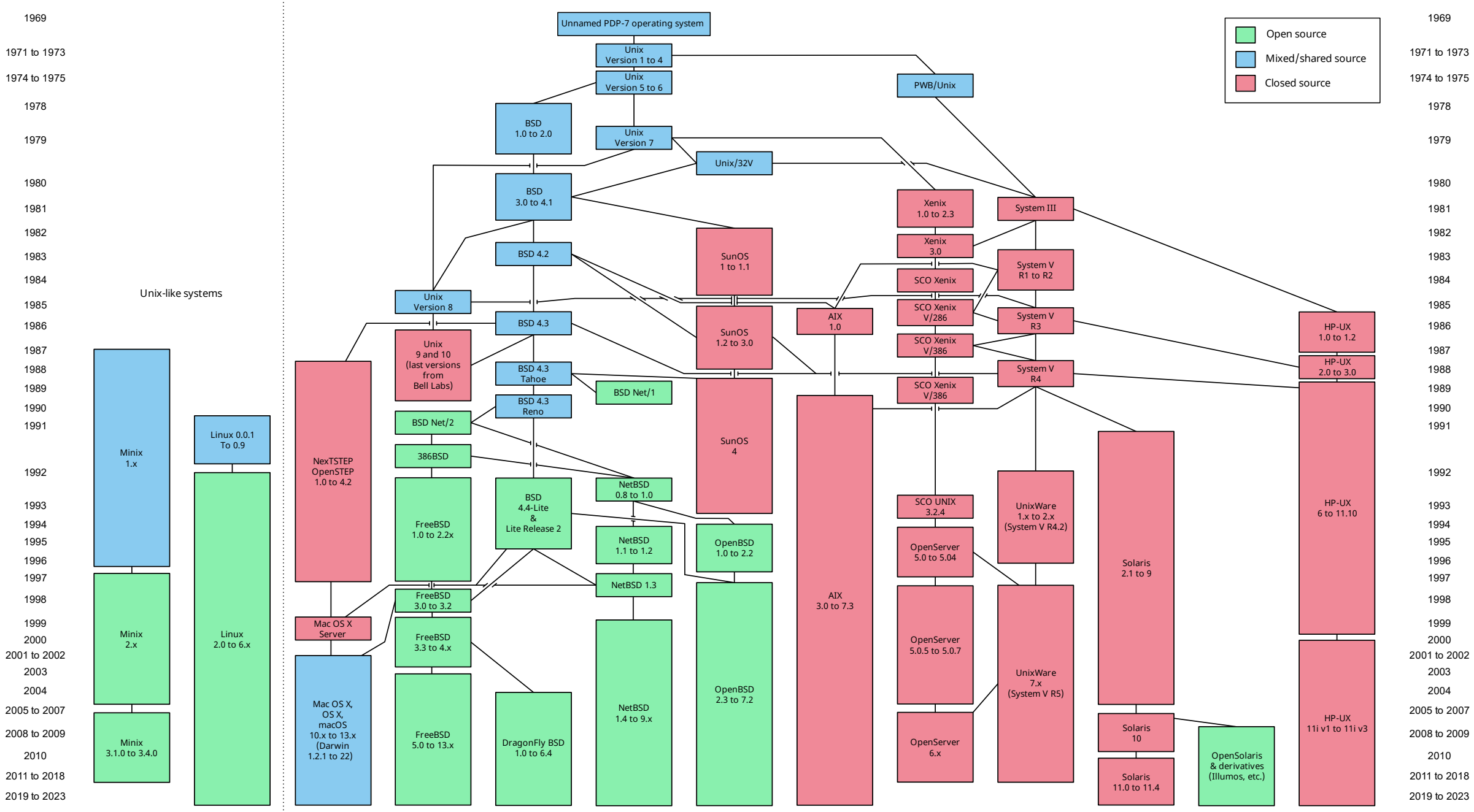
- VAX 11
- **V**irtual **A**ddress **E**xtension of the PDP-**11**
- Dave Culter
  - Architect of Windows NT

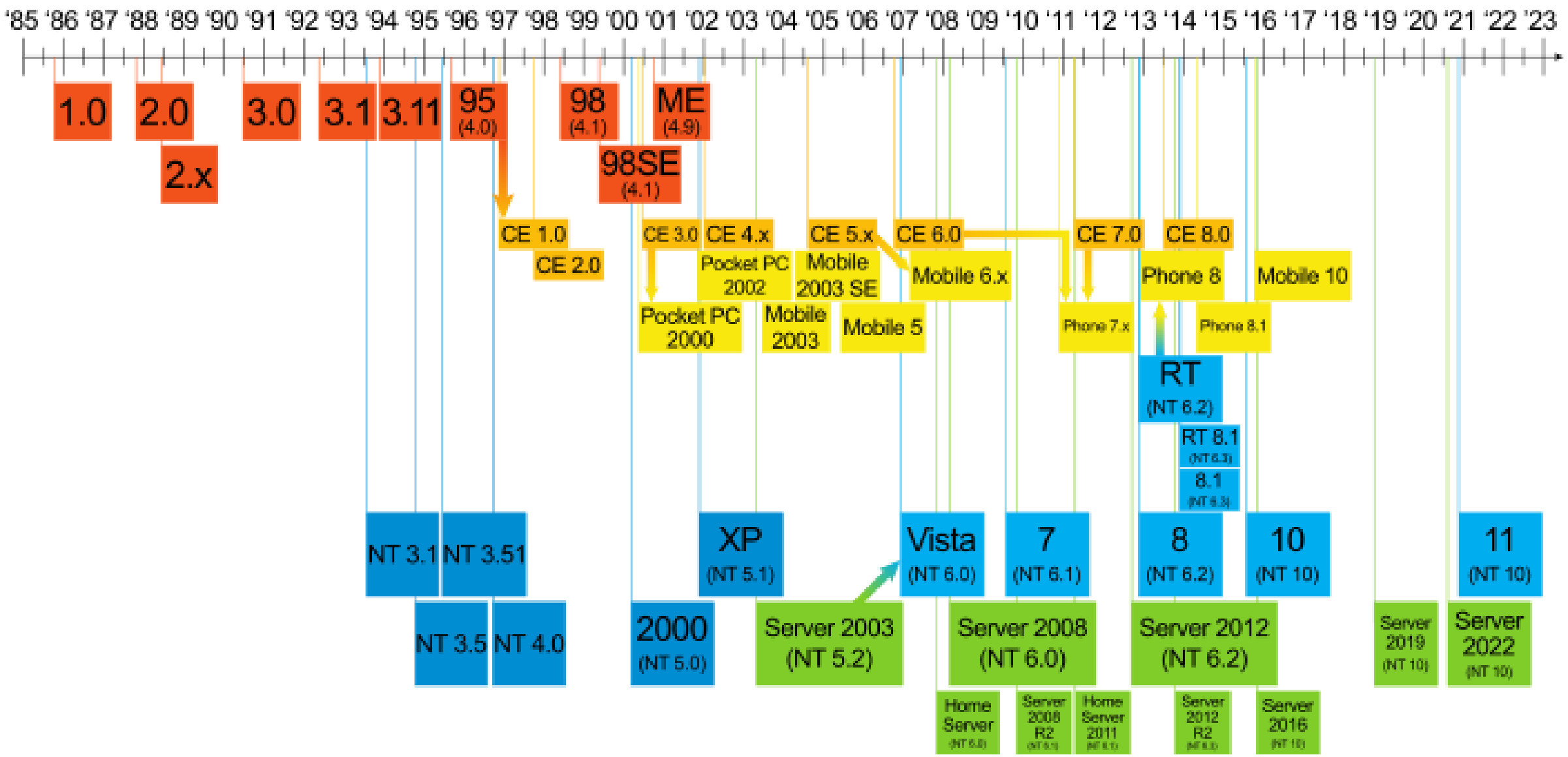
Unix

- Written on a PDP-11
- Berkely development added
  - Virtual Memory
  - Network
- Ken Thompson and Dennis Ritchie

iOS & Mac OS

- Microkernel Architecture
- Carnegie Mellon University
  - OS provides POSIX API
- Avie Trevanian


Unix-like systems
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2002
2003
2004
2005 to 2007
2008 to 2009
2010
2011 to 2018
2019 to 2023
Open source
Mixed/shared source
Closed source
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
Unix Version 7
PWB/Unix
BSD 1.0 to 2.0
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
Xenix 3.0
System III
BSD 4.2
SunOS 1 to 1.1
SCO Xenix
System V R1 to R2
Unix Version 8
SCO Xenix V/286
System V R3
BSD 4.3
SunOS 1.2 to 3.0
AIX 1.0
SCO Xenix V/386
HP-UX 1.0 to 1.2
Unix 9 and 10 (last versions from Bell Labs)
HP-UX 2.0 to 3.0
BSD 4.3 Tahoe
BSD Net/1
System V R4
BSD 4.3 Reno
Minix 1.x
Linux 0.0.1 To 0.9
NextSTEP OpenSTEP 1.0 to 4.2
BSD Net/2
386BSD
SunOS 4
AIX 3.0 to 7.3
HP-UX 6 to 11.10
FreeBSD 1.0 to 2.2x
BSD 4.4-Lite & Lite Release 2
NetBSD 0.8 to 1.0
SCO UNIX 3.2.4
UnixWare 1.x to 2.x (System V R4.2)
Solaris 2.1 to 9
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
NetBSD 1.3
FreeBSD 3.0 to 3.2
OpenServer 5.0.5 to 5.0.7
UnixWare 7.x (System V R5)
Minix 2.x
Linux 2.0 to 6.x
Mac OS X Server
FreeBSD 3.3 to 4.x
Solaris 10
OpenSolaris & derivatives (Illumos, etc.)
Mac OS X, OS X, macOS 10.x to 13.x (Darwin 1.2.1 to 22)
NetBSD 1.4 to 9.x
OpenBSD 2.3 to 7.2
OpenServer 6.x
HP-UX 11i v1 to 11i v3
Minix 3.1.0 to 3.4.0
FreeBSD 5.0 to 13.x
DragonFly BSD 1.0 to 6.4
Solaris 11.0 to 11.4


'85 '86 '87 '88 '89 '90 '91 '92 '93 '94 '95 '96 '97 '98 '99 '00 '01 '02 '03 '04 '05 '06 '07 '08 '09 '10 '11 '12 '13 '14 '15 '16 '17 '18 '19 '20 '21 '22 '23
1.0
2.0
2.x
3.0
3.1
3.11
95 (4.0)
98 (4.1)
98SE (4.1)
ME (4.9)
CE 1.0
CE 2.0
CE 3.0
CE 4.x
CE 5.x
CE 6.0
CE 7.0
CE 8.0
Pocket PC 2002
Mobile 2003 SE
Mobile 6.x
Phone 8
Mobile 10
Pocket PC 2000
Mobile 2003
Mobile 5
Phone 7.x
Phone 8.1
RT (NT 6.2)
RT 8.1 (NT 6.3)
8.1 (NT 6.3)
NT 3.1
NT 3.51
XP (NT 5.1)
Vista (NT 6.0)
7 (NT 6.1)
8 (NT 6.2)
10 (NT 10)
11 (NT 10)
NT 3.5
NT 4.0
2000 (NT 5.0)
Server 2003 (NT 5.2)
Server 2008 (NT 6.0)
Server 2012 (NT 6.2)
Server 2019 (NT 10)
Server 2022 (NT 10)
Home Server (NT 6.0)
Server 2008 R2 (NT 6.1)
Home Server 2011 (NT 6.1)
Server 2012 R2 (NT 6.3)
Server 2016 (NT 10)

Virtualisation

To make a thing, like a thing, but not actually a thing, but seem like a thing

Overview

- Abstraction
- Process
- Access
- Process vs Thread
- Kernel vs User Mode
- System Calls and Interrupts

Abstraction

My Program

<code>
<code>
<code>
<code>
<code>
<use printer>

Abstraction

My Program

<code>
<code>
<code>
<code>
<code>
<use EPSON Stylus S22>

Abstraction

My Program

<code>
<code>
<code>
<code>
<code>
if printer == **EPSON Stylus S22**
<use EPSON Stylus S22>
else if printer == ???
else if printer == ???

Abstraction

My Program

<code>
<code>
<code>
<code>
<code>
<use printer>

Print

Operating System

Print

Abstraction

My Program #1

<code>
<code>
<code>
<code>
<code>
<use printer>

Print

My Program #2

<code>
<code>
<code>
<code>
<code>
<use printer>

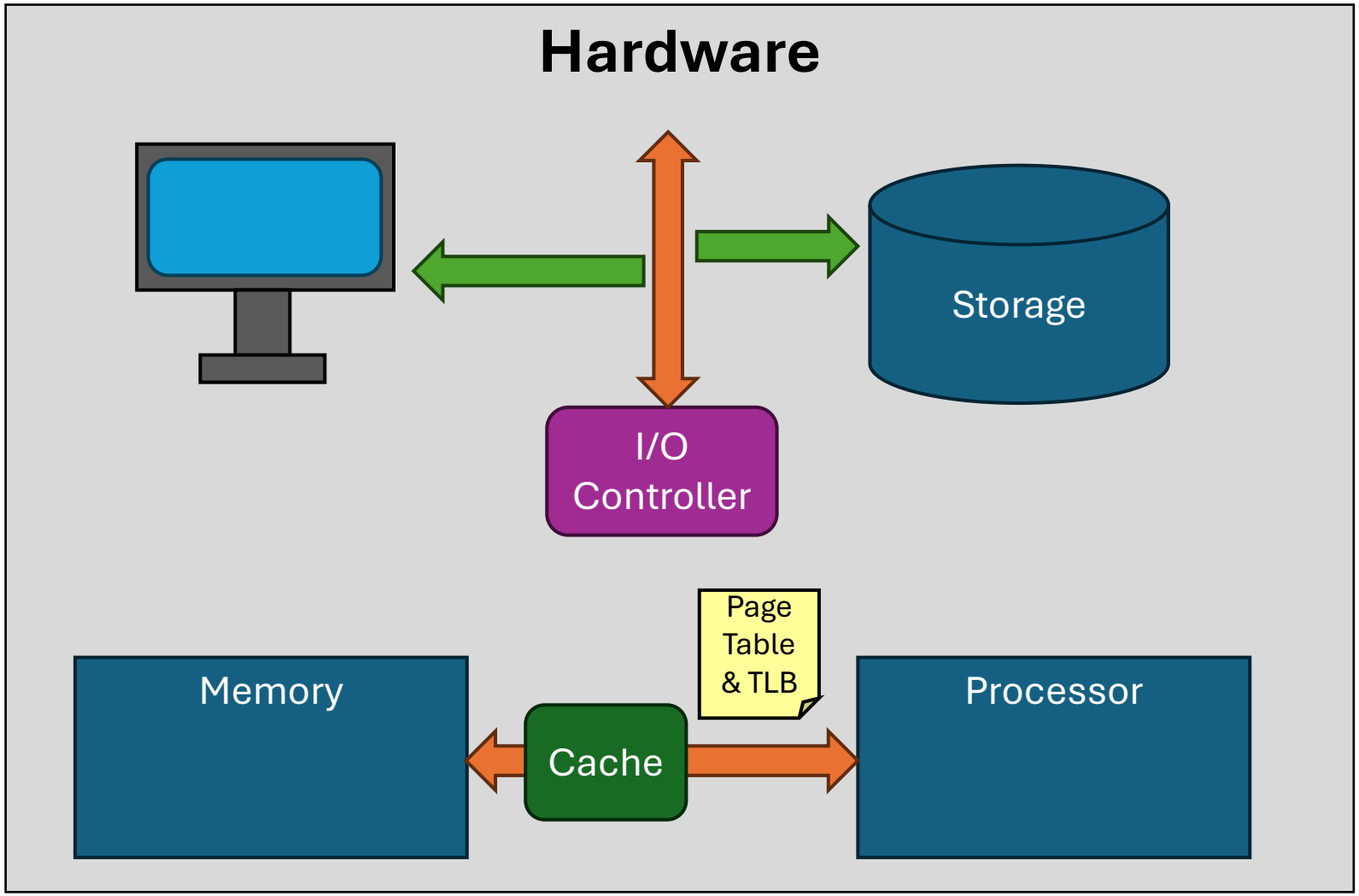
Print

Operating System

Print

Hardware Software Interface

My Program #1
<code>
<code>
<code>
<code>
<code>
<use printer>

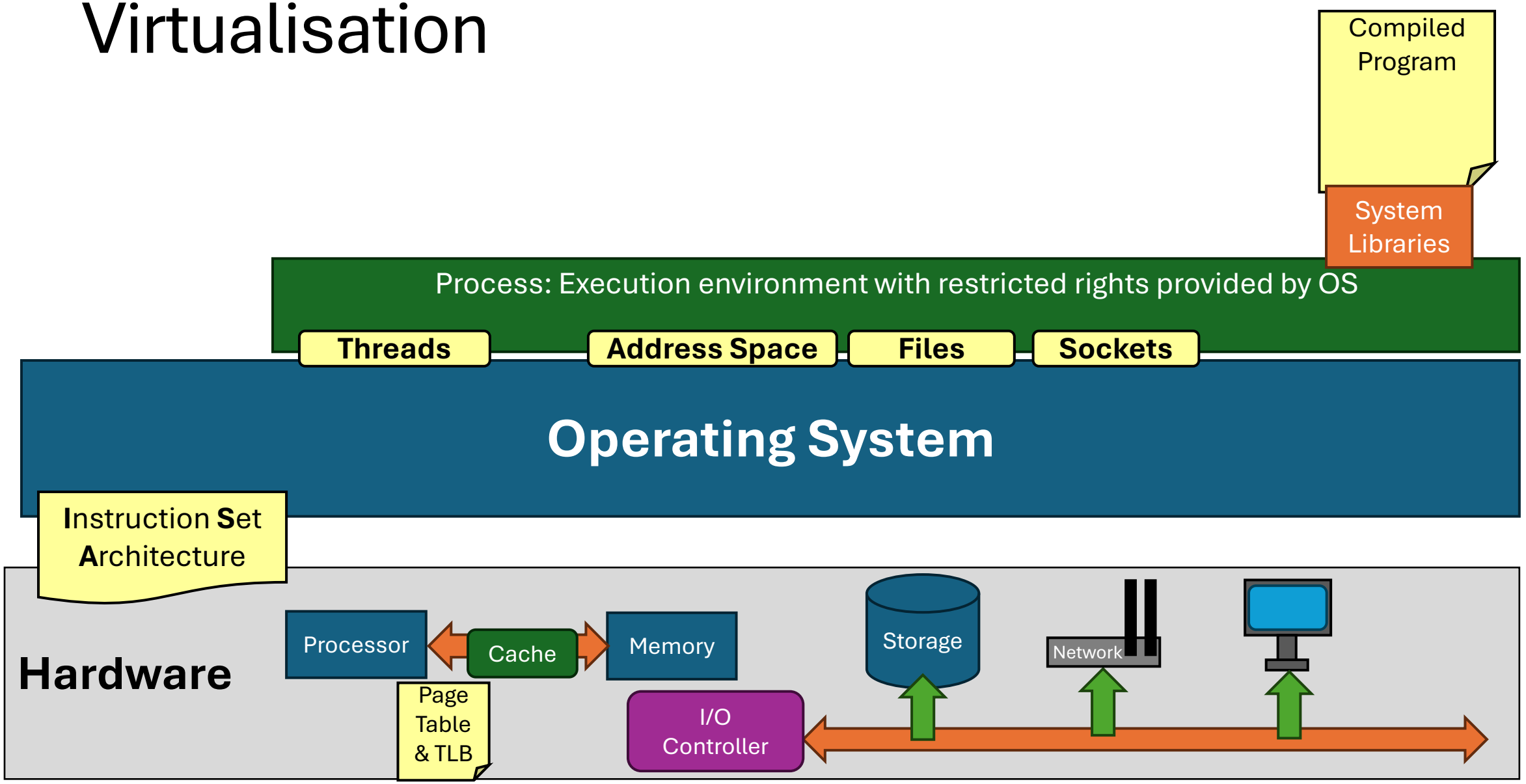


Hardware Abstraction

Hardware		Software
Processor	⟺	Thread
Memory	⟺	Address Space
Disks, SSDs	⟺	Files
Networks	⟺	Sockets
Machines	⟺	Processes

The OS creates an **illusion**

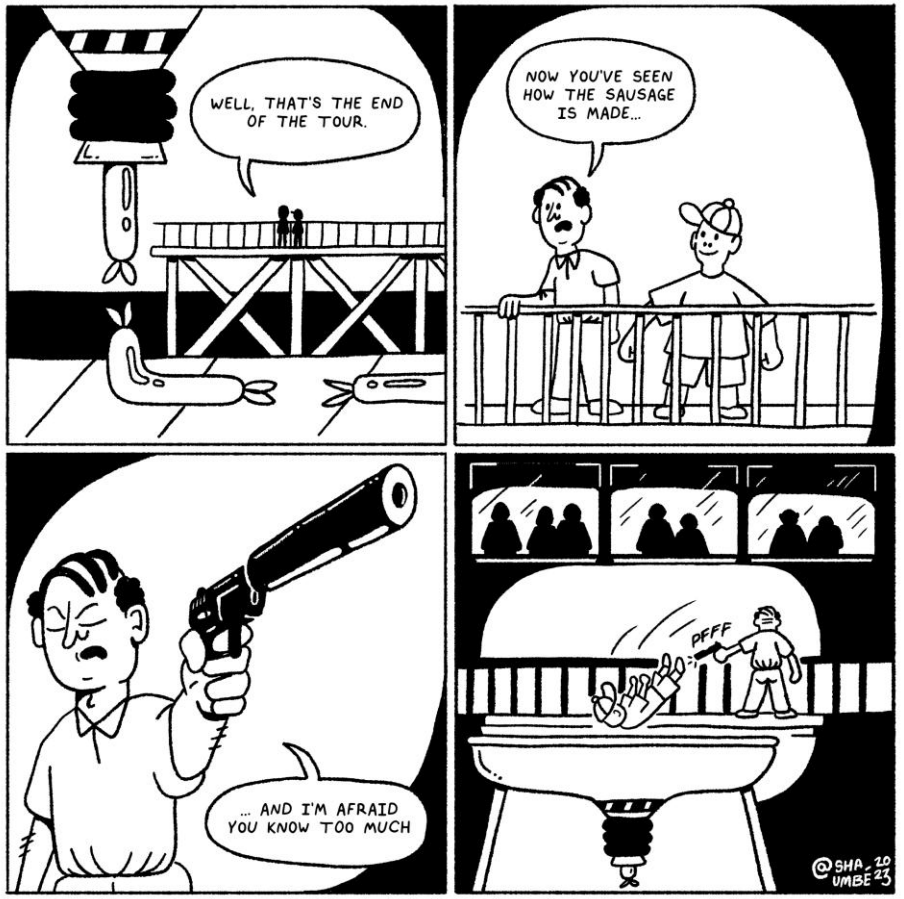
Virtualisation



Virtualisation

Programs

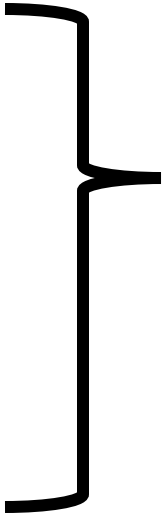
- Only see the interface
  - System calls, address space etc.
- Each program runs its own process



Virtualisation: System Resources

Hardware

Processor
Memory
Disks, SSDs
Networks
Machines



System Resources

Virtualisation

- Each process thinks it is in its own universe? I.e., it has all the resources.
  - Requires time-sharing
  - Requires space-sharing

Abstraction: Summary

Advantages

- Code reuse
- Unifies interface for many devices
- Allows higher-level functionality

Challenges

- What is the best abstraction?
- *Functionality* vs *Generality*
- How much hardware do you see?
- How much does hardware determine the abstraction?

Protection

Protection from what? Zee germans?

Protection

My Program #1

<code>
<code>
<code>
<code>
<code>
<use printer>

Print

My Program #2

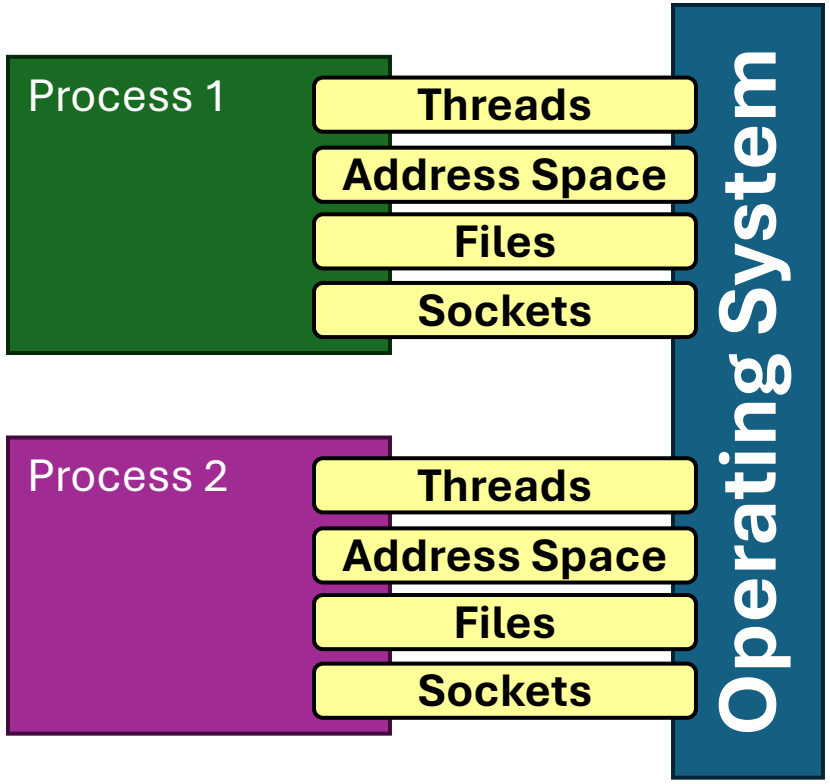
<code>
<code>
<code>
<code>
<code>
<use printer>

Print

Operating System

Print

Protection



Process 1 requires

- Isolation from Process 2
- Isolation from the OS

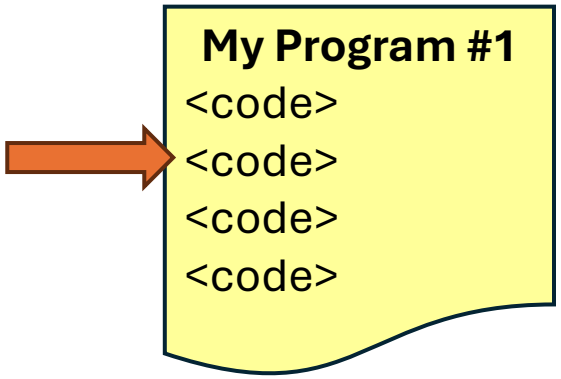
Process

What is it?

Process

Definition?

- Thread of control
- Not a program (sort of)

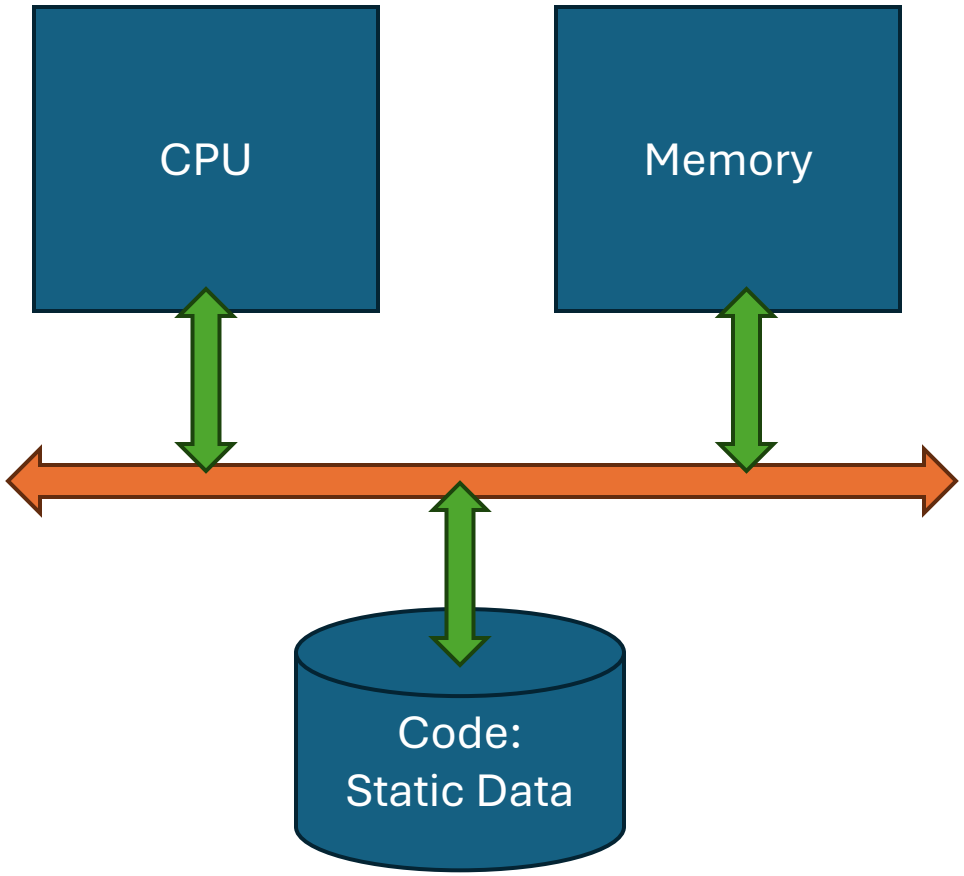


Parts

- Registers
  - Program Counter
  - Stack Pointer
- Visible address space
- Open files
- Open communication channels

Process: Initiating a Process

- Where is the program?
- Is it just copying?

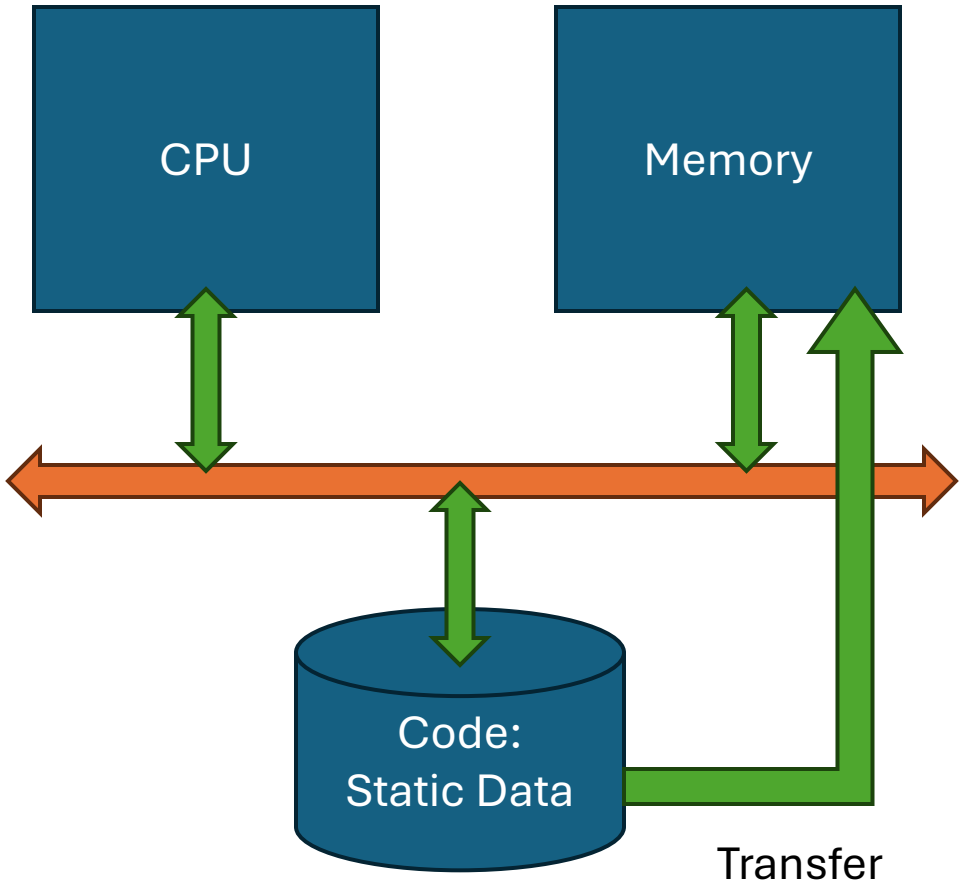


Process: Initiating a Process

- Where is the program?
- Is it just copying?

Loader loads code into memory.

- Overwrite code?
- Fix pointers?
- Stack layout?
- Libraries?



Process vs Thread

Process

- Program Counter
- Stack
- Registers
- Memory Space

Thread

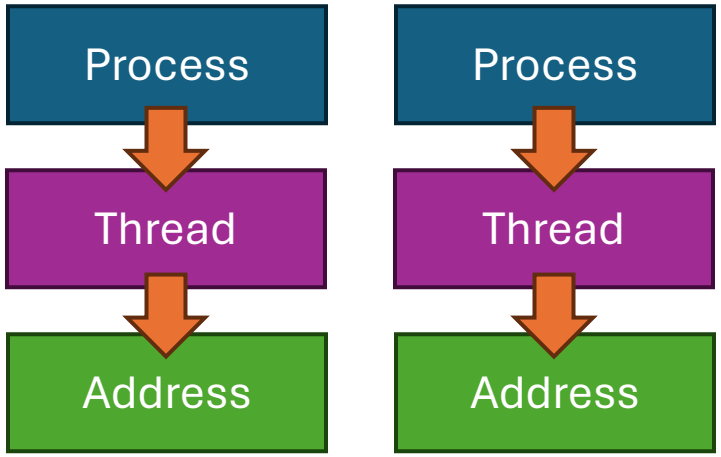
- Program Counter
- Stack
- Registers
- **Shared** Memory Space

In a technical sense, every **process** is running at least one (and usually one) **thread**.

Process vs Thread

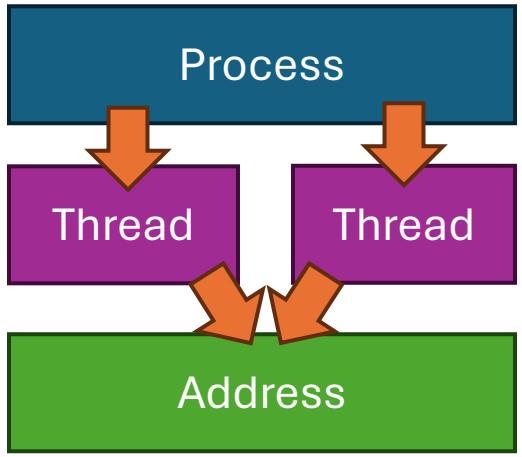
Process

- When examining an address in its address space will see a **different** value to another process addressing the same address.



Thread

- When examining an address in its address space will see the **same** value to another process addressing the same address.



This can be **very** bad

Performance vs Protection

How to balance the two

Direct Execution

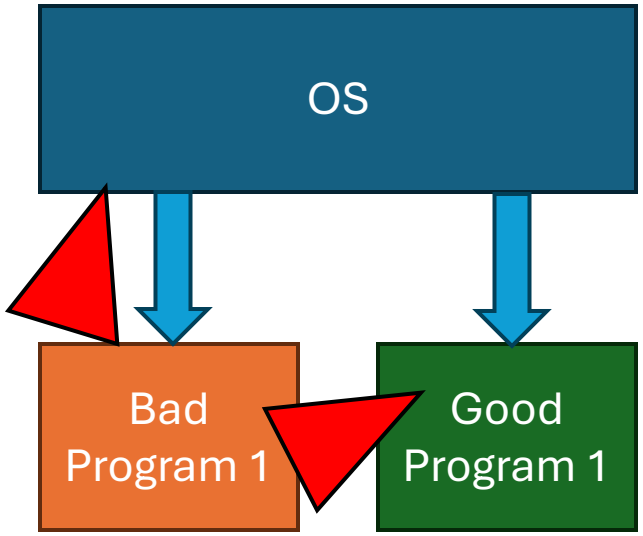
What is it?

- OS functions are very efficient and cleverly built
- *Direct Execution* wants to run CPU instructions directly on the CPU

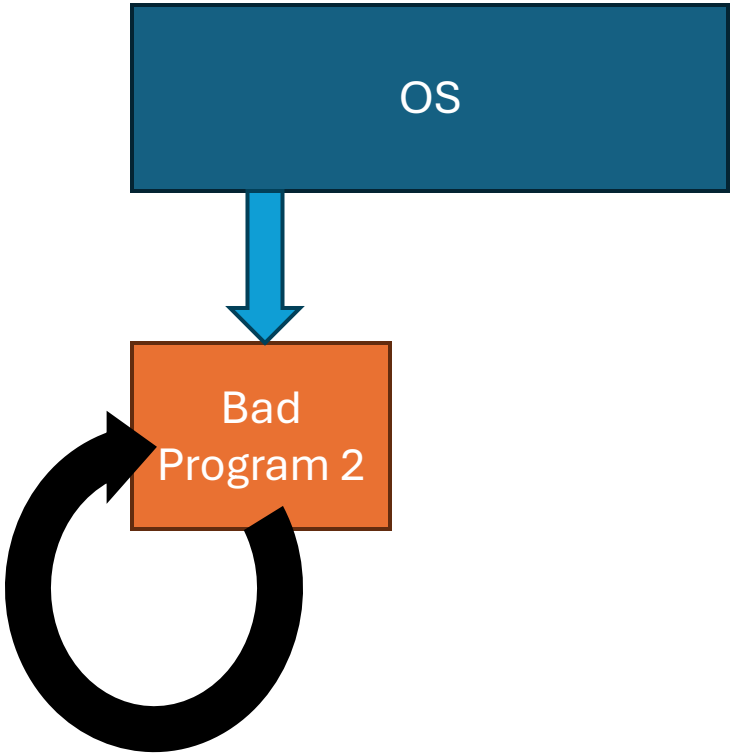
Disasters waiting to happen

- We need to restrict access
- We need to avoid ‘infinite programs’
- Slow

Shared Resources: Protection..



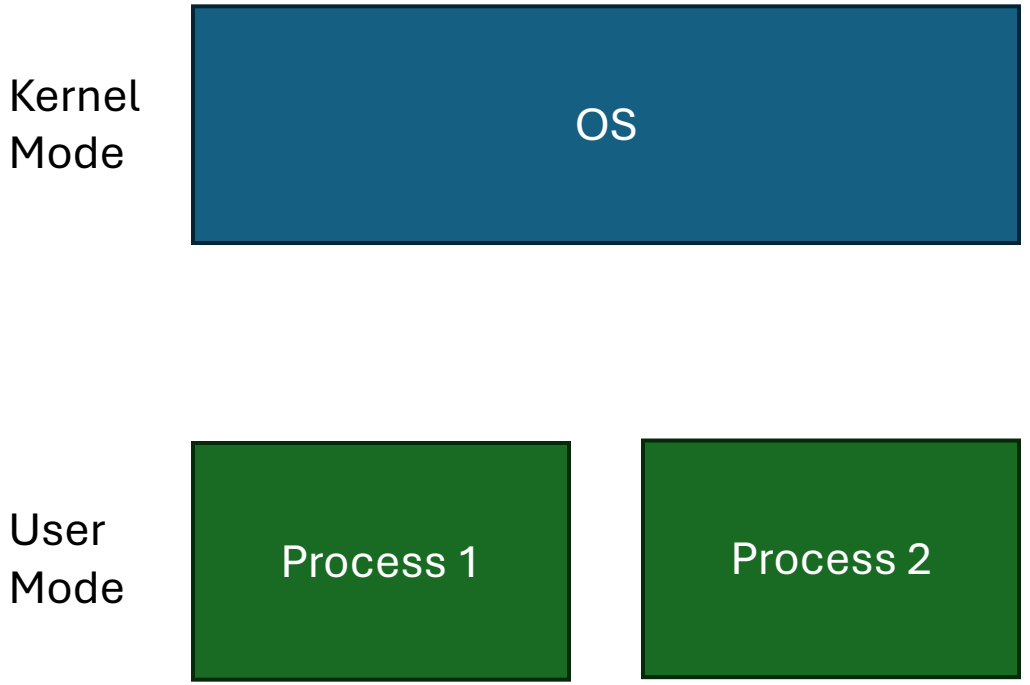
Destroy the OS and every other program



Limited Direct Execution

Modes

- User mode:
  - Not all instructions are available
- Kernel mode:
  - Device instructions
  - Can have multiple levels of privilege

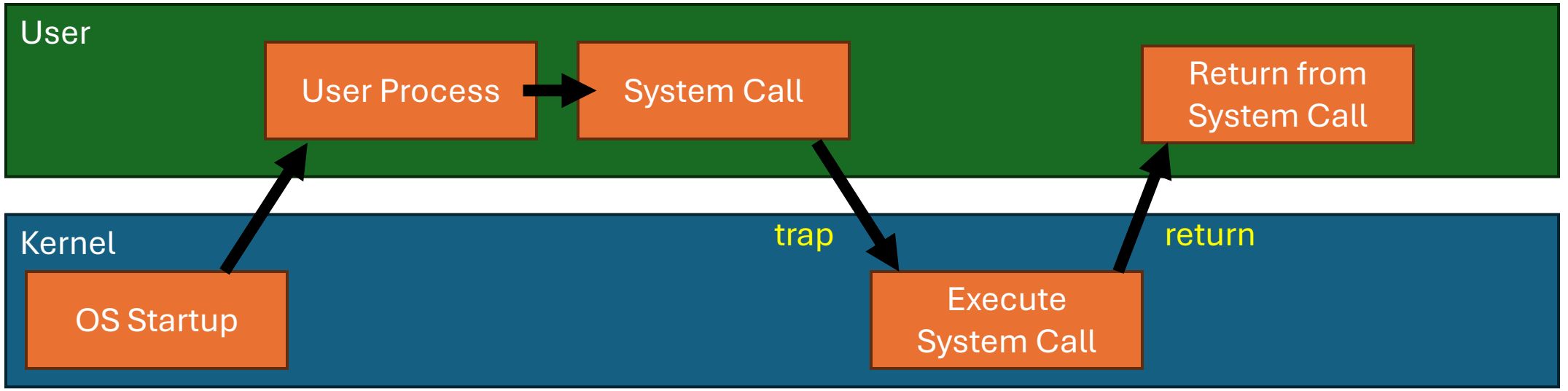


Limited Direct Execution

System Calls (Traps)

- User tries to call **System**
- OS is given control to do it on your behalf

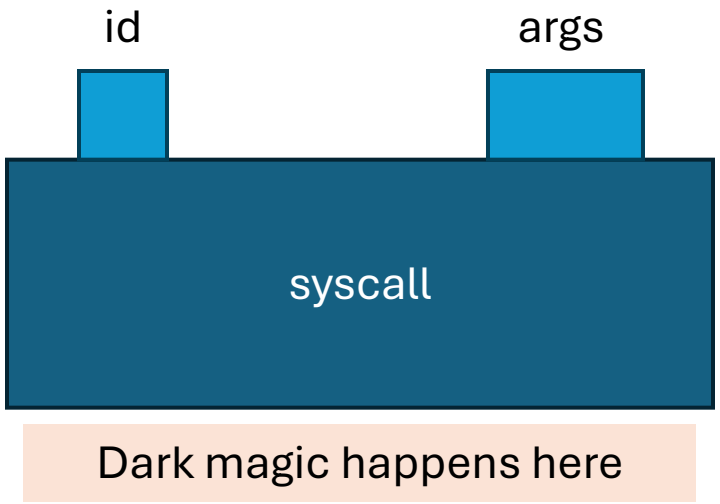
The user/kernel mode is controlled by a ‘mode bit’.
user = 1
kernel = 0



Kernel Mode Transfer: Types

Syscall

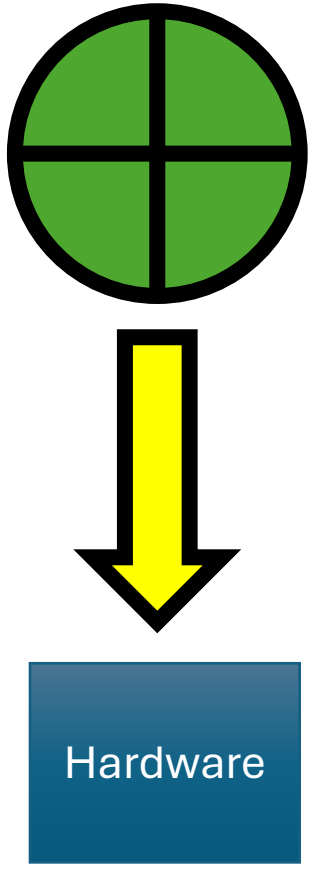
- Process requests a **system service** (i.e., exit)
- Behaves like a **function call**, but runs outside a process
- Does **not** have the address of the system function to call
- Marshall the syscall **id** and **args** in registers and exec **syscall**



Kernel Mode Transfer: Types

Interrupts

- **External asynchronous** event triggers context switch
- Timer, I/O device, etc.
- Maskable
- Independent of user process



Kernel Mode Transfer: Types

Exceptions

- **Internal synchronous** event in process triggers context switch
- Examples
  - Protection violation (seg fault)
  - Divide-by-zero

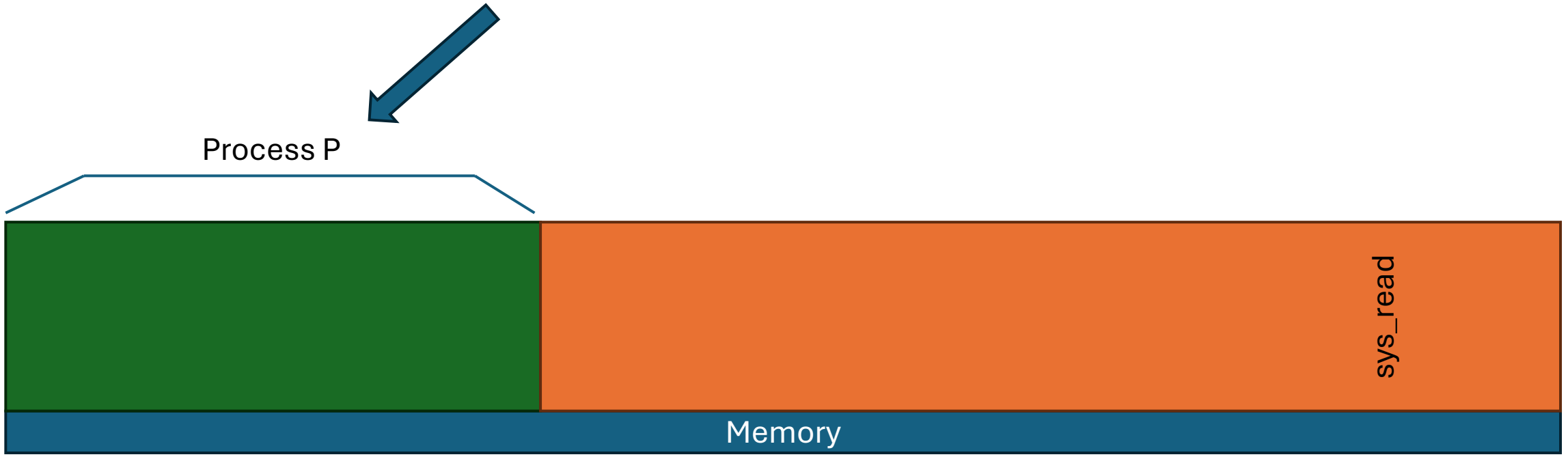
Interrupt vs Exception

Exceptions refer to what you (internal) are doing now (synchronous) and how it is wrong (usually)

Interrupts can be unrelated to what you are doing now (asynchronous), are triggered by something else (asynchronous) and are more about handling normal behaviour (usually)

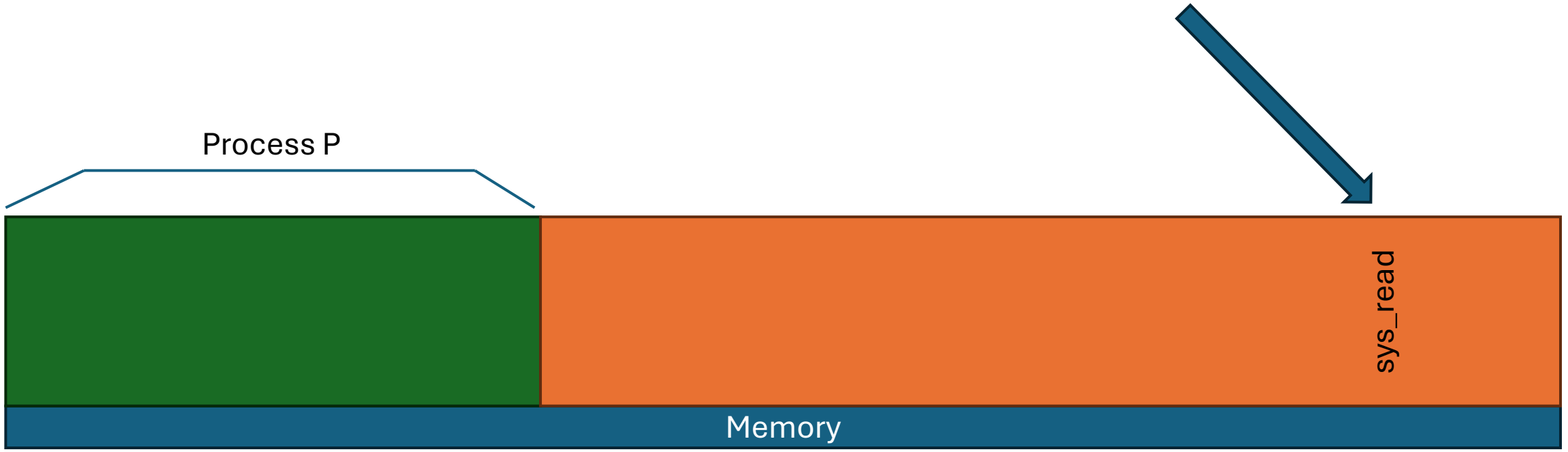
System Calls

We curtail where **Process P** can fiddle in the memory to avoid malicious programs

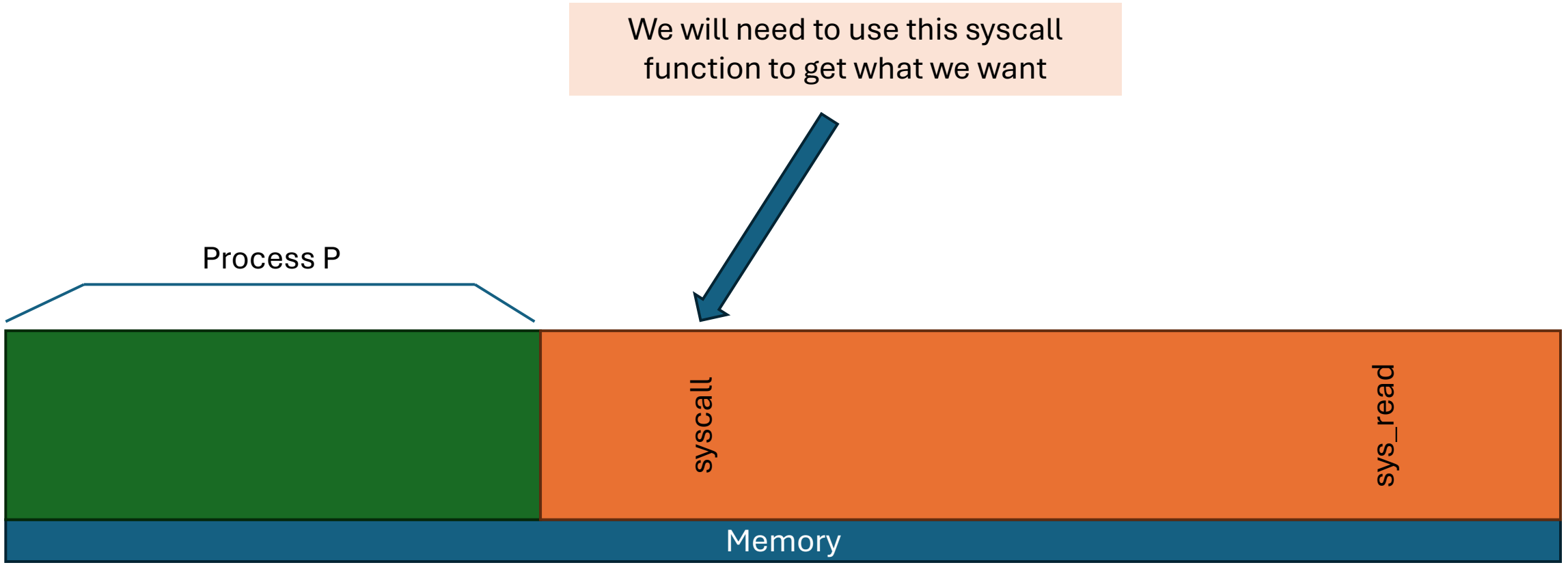


System Calls

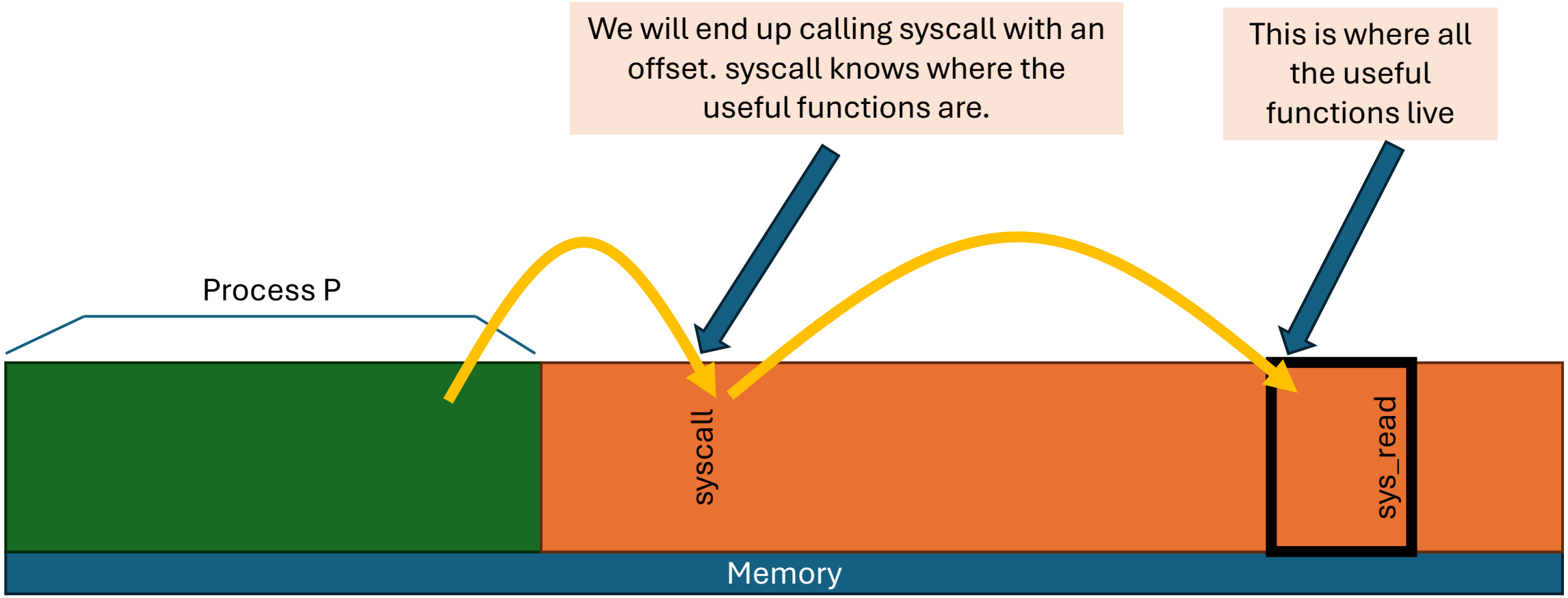
We want to use this particular instruction which is really just about reading something



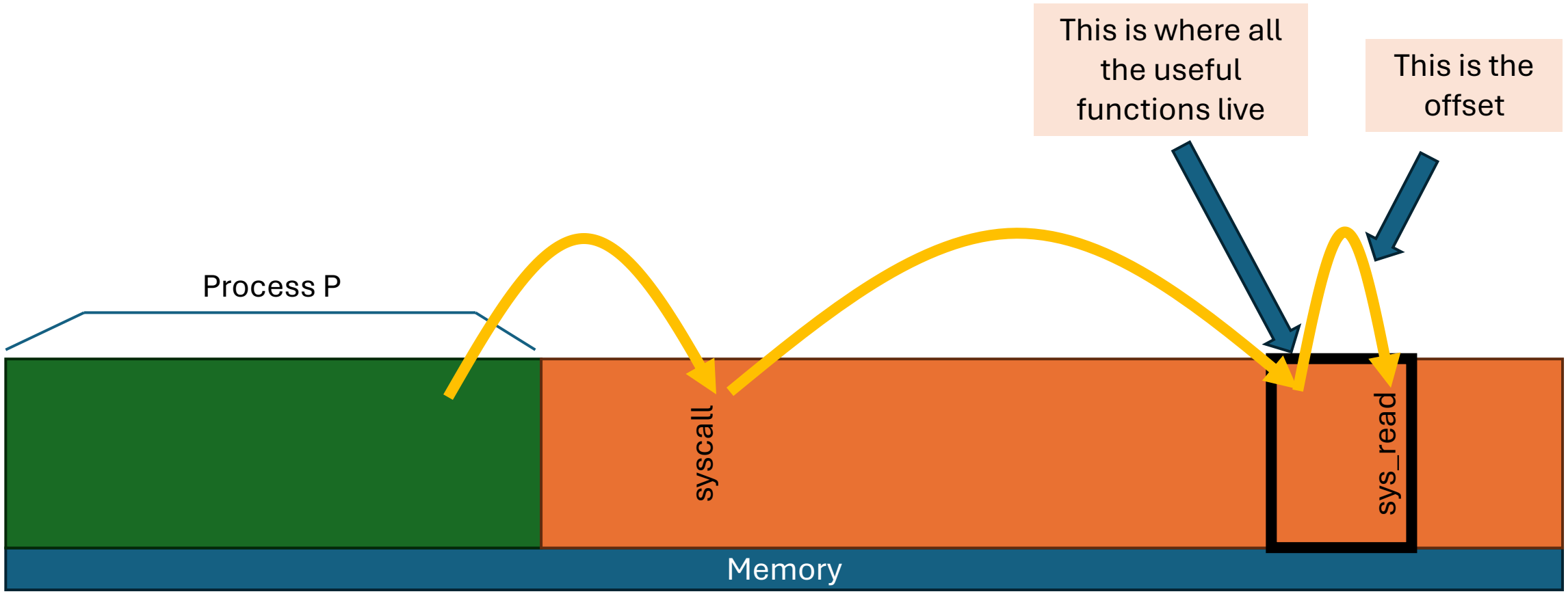
System Calls



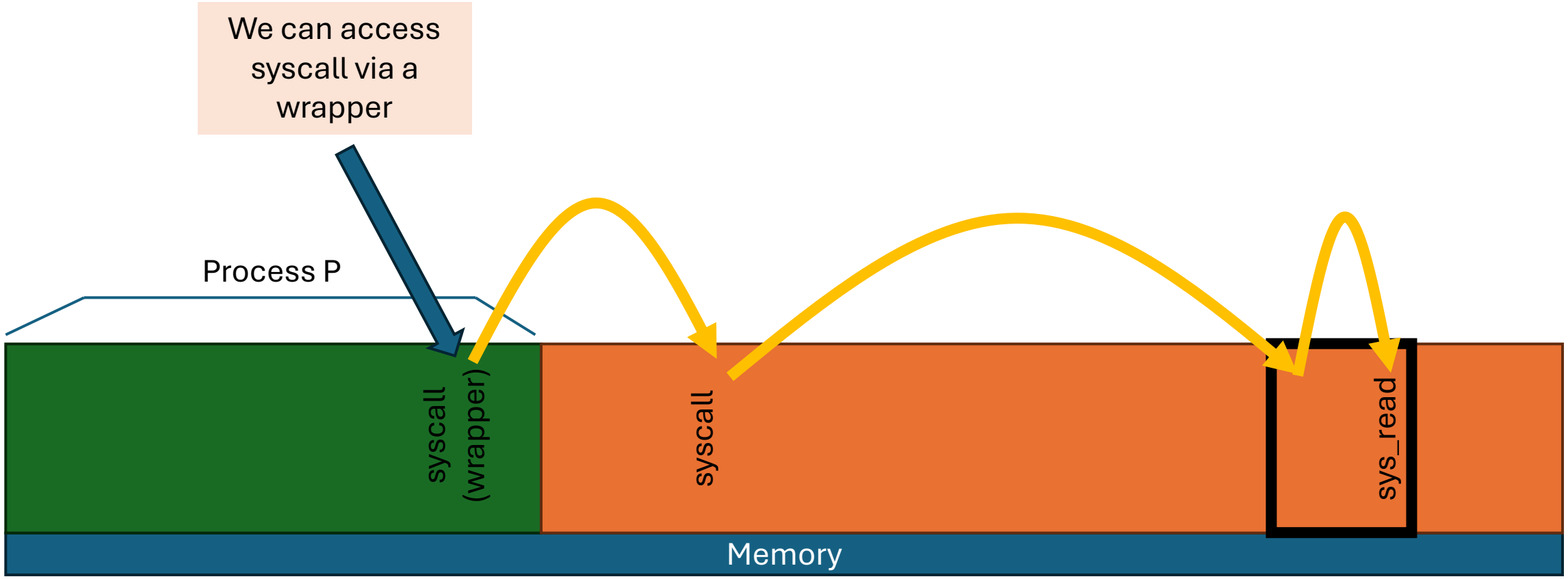
System Calls



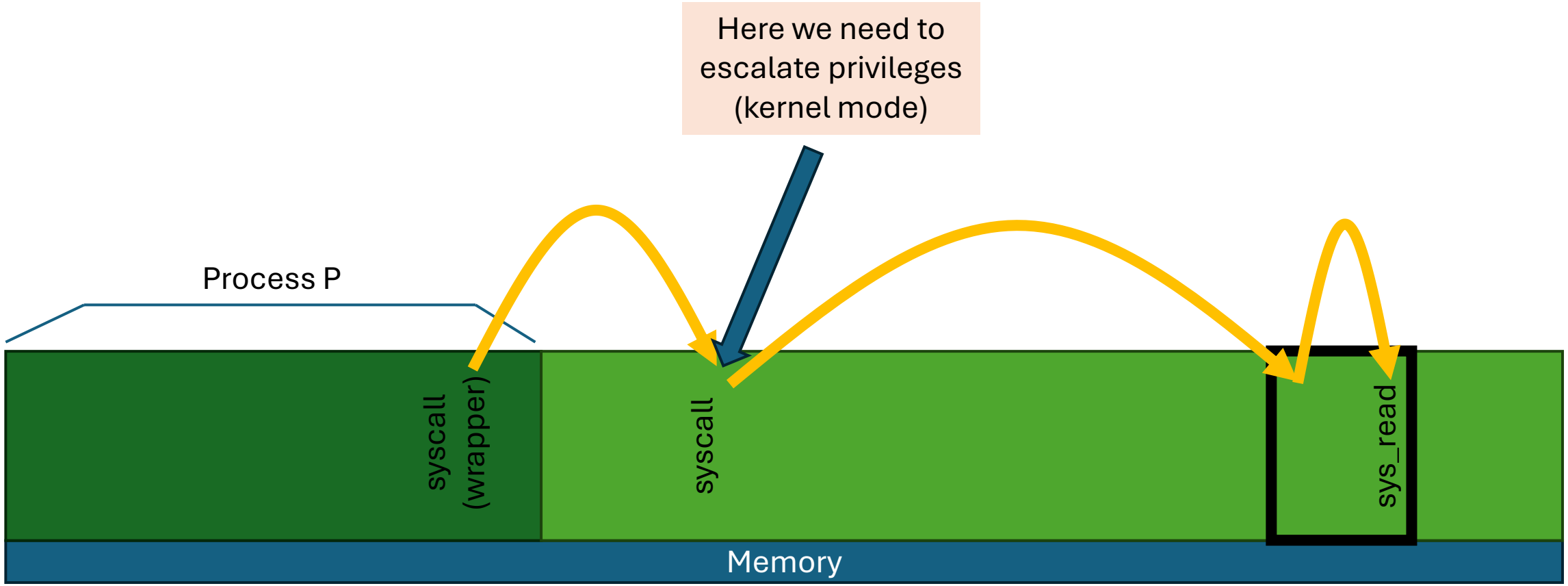
System Calls



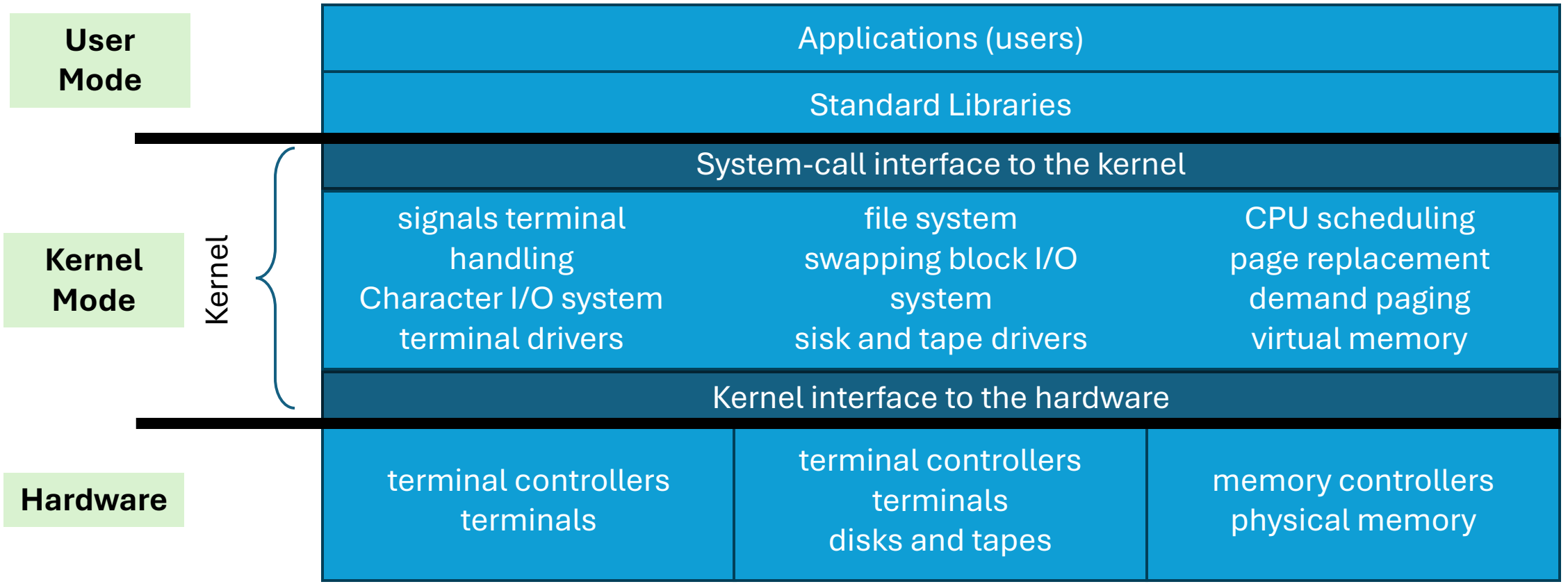
System Calls



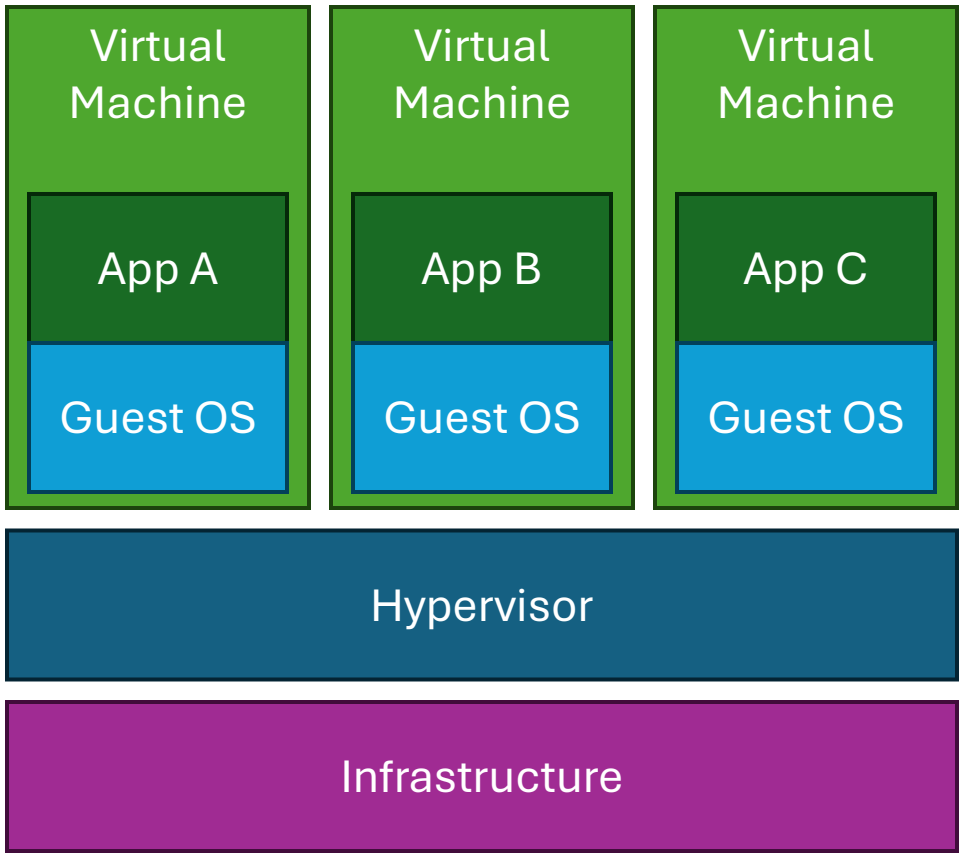
System Calls

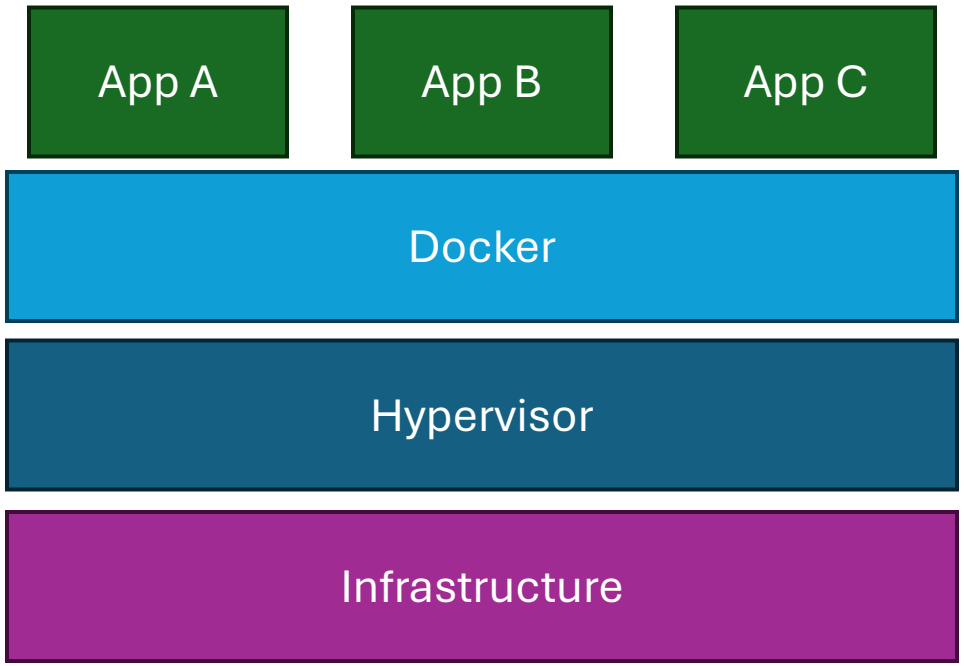


Unix System Structure



Alternative Structures

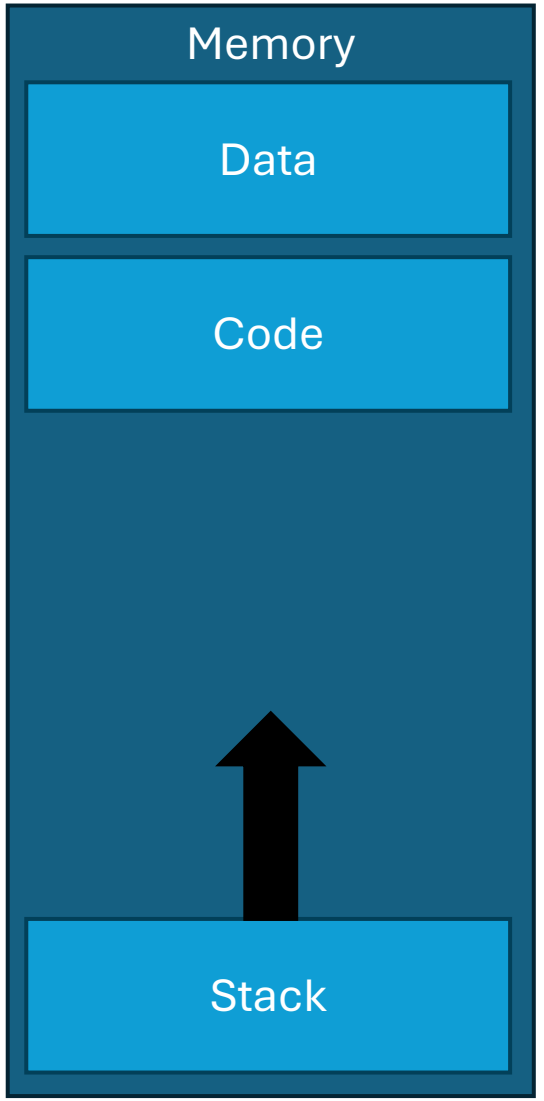




Revision: Stack Machine

Each process is assigned memory:

- Data (global data)
- Code (the actual bit of code)
- Stack (what is currently executing)



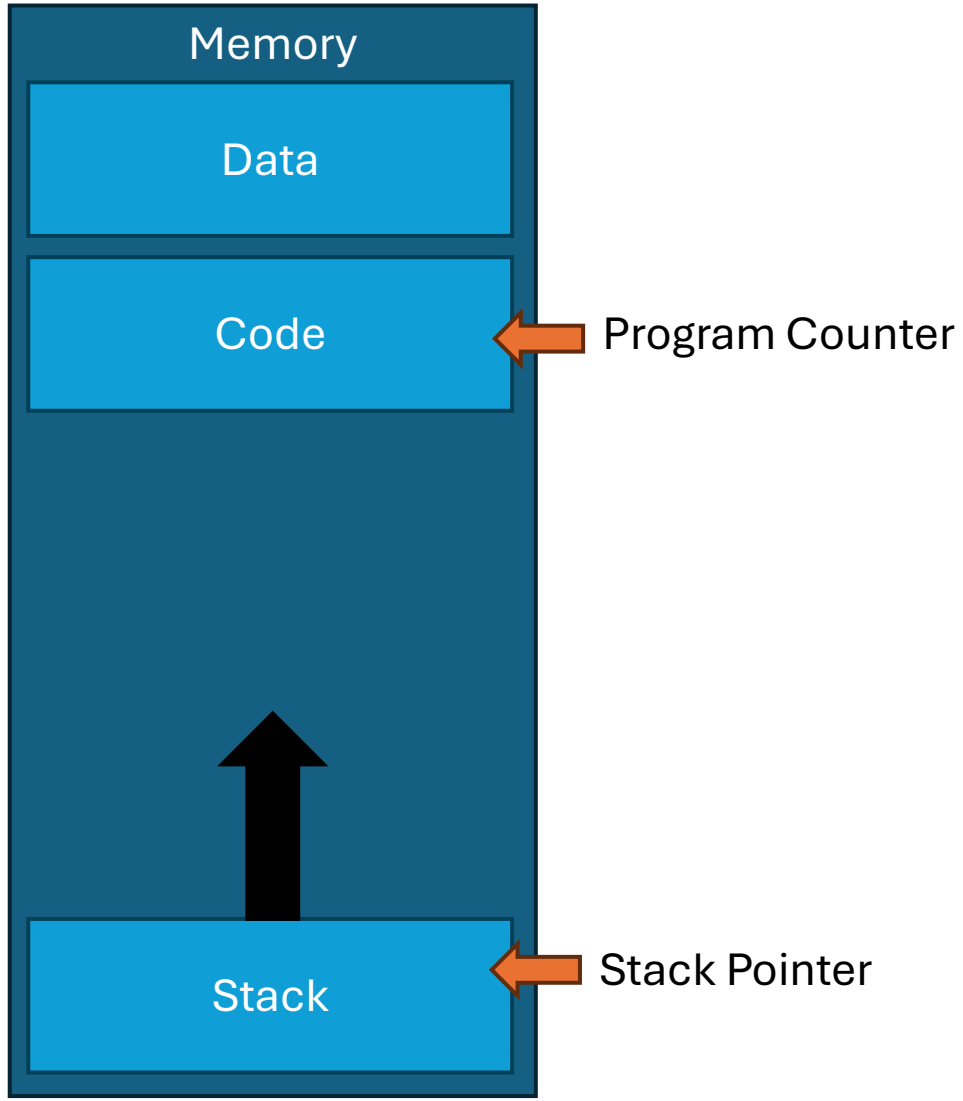
Revision: Stack Machine

Each process is assigned memory:

- Data (global data)
- Code (the actual bit of code)
- Stack (what is currently executing)

Registers

- Where we are in the code?
- Where we are in the stack?

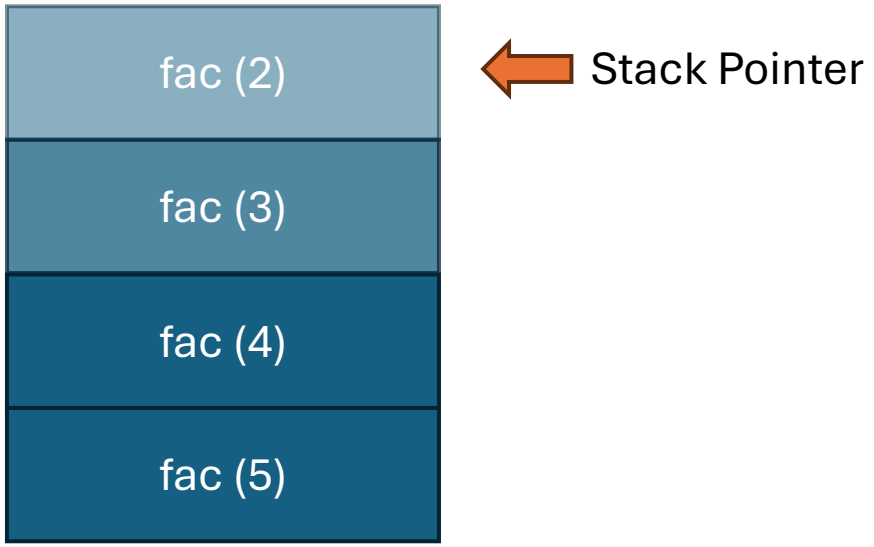


Program Counter vs Stack Pointer

```
fac(int n)
{
        if (n ≤ 0)
                return 0;
        else
                return n * fac(n-1);
}
```

Program Counter

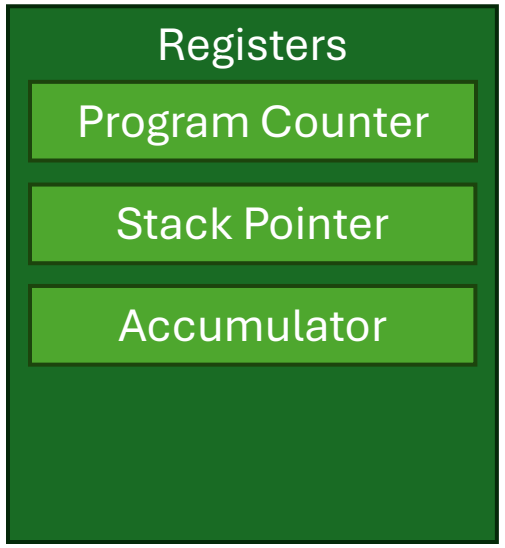
fac (5)

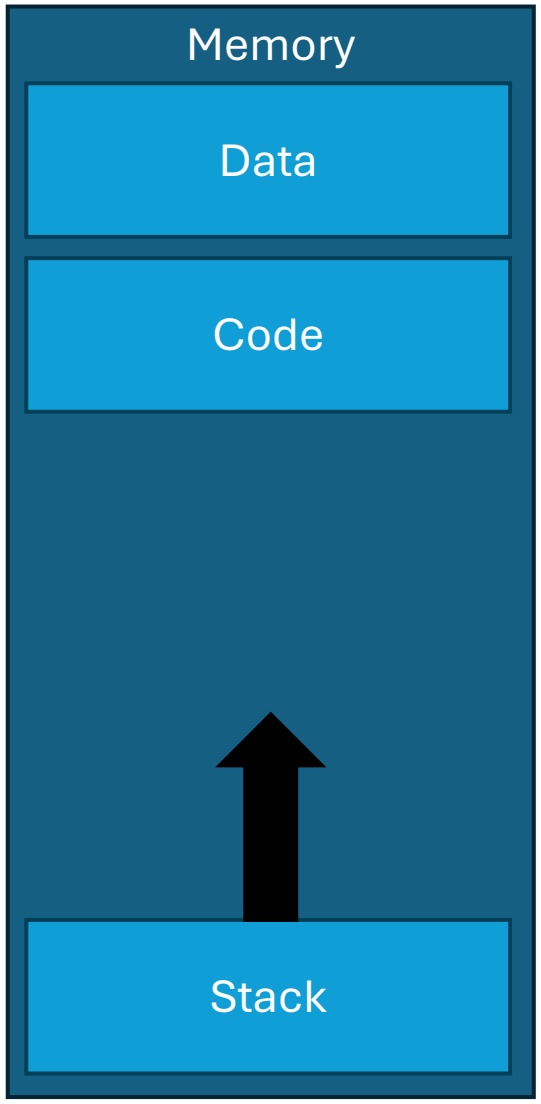


Accumulator

Accumulator Register

- Stores interim results
- Old machines had one register
- Superceded by general purpose registers



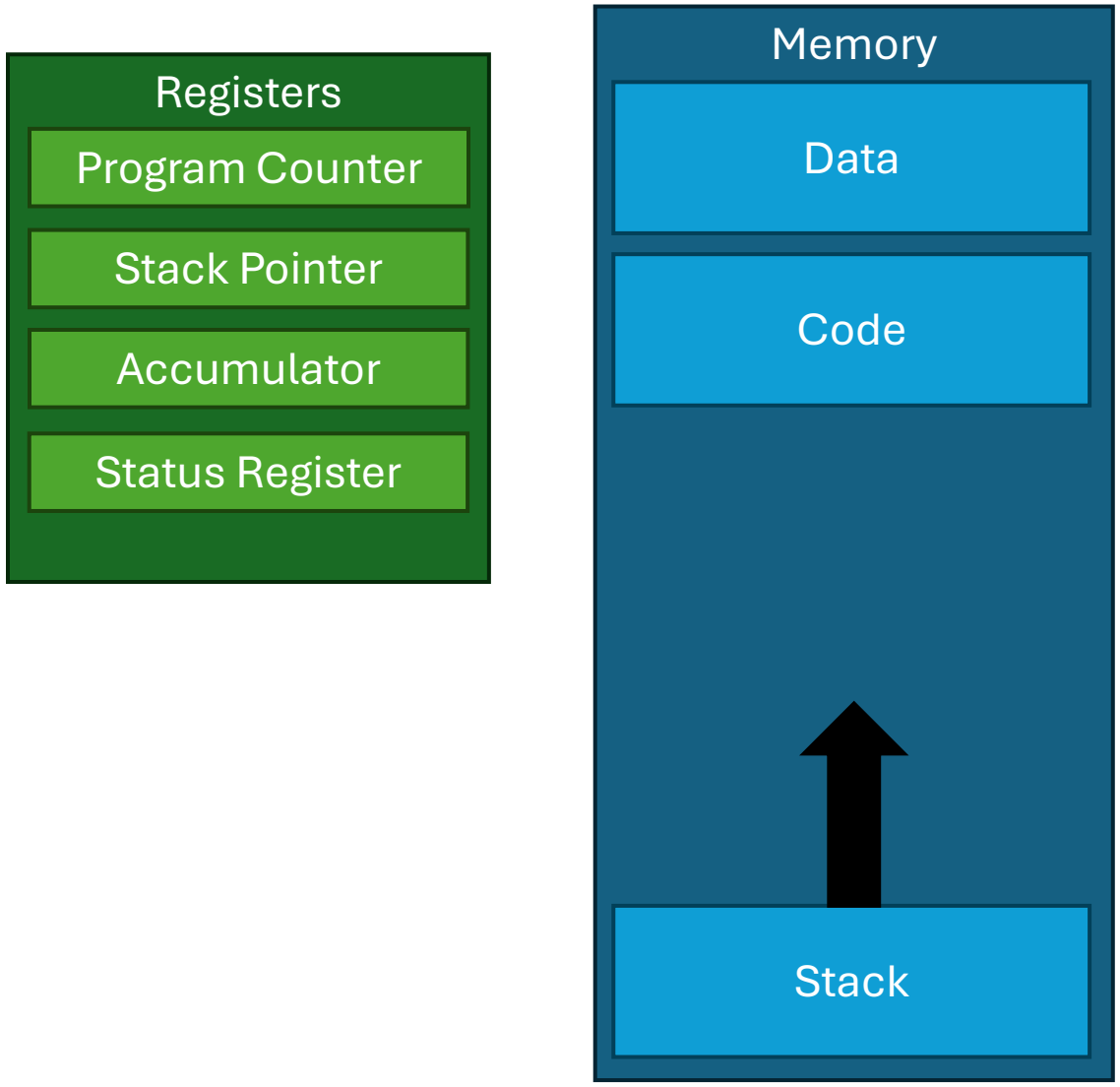



```
LOAD A      ; Load the value at memory location A into the accumulator
ADD B       ; Add the value at memory location B to the accumulator
STORE C     ; Store the accumulator's result into memory location C
```

Status

Status Register

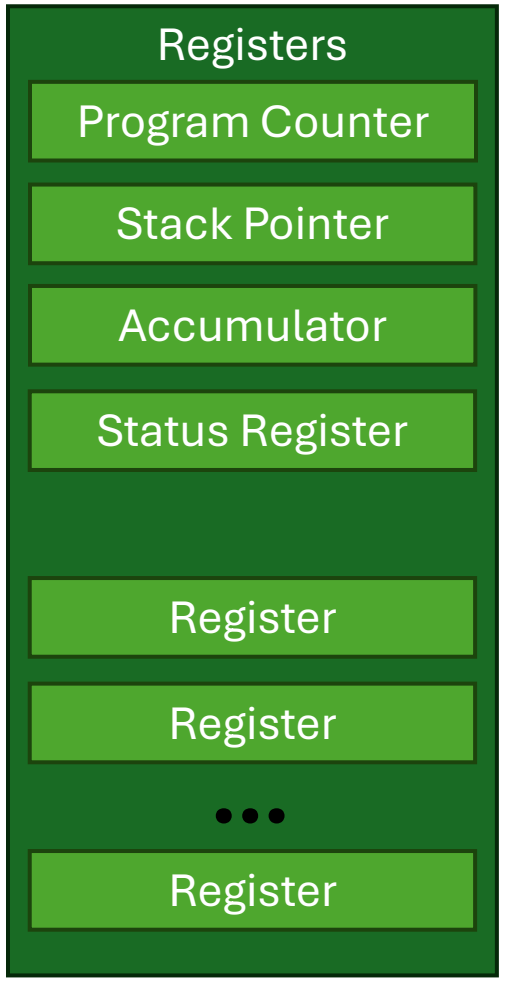
- Stores information…
- Flags
  - Are interrupts enabled?
  - Was there an overflow?
  - Was the result zero?

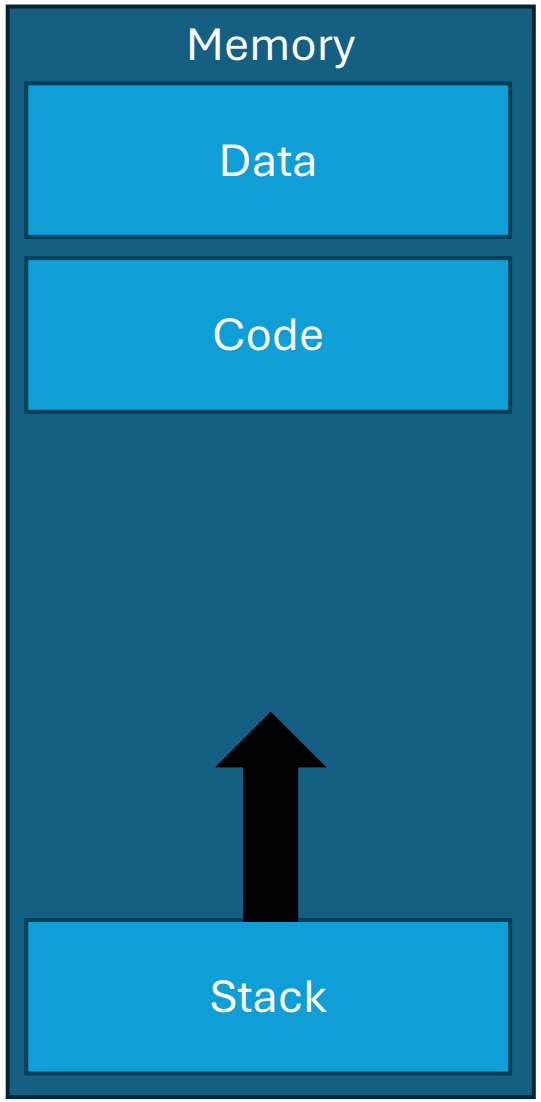


More Registers

General Purpose Registers

- Operands
- Results
- Pointers
- Specific Roles



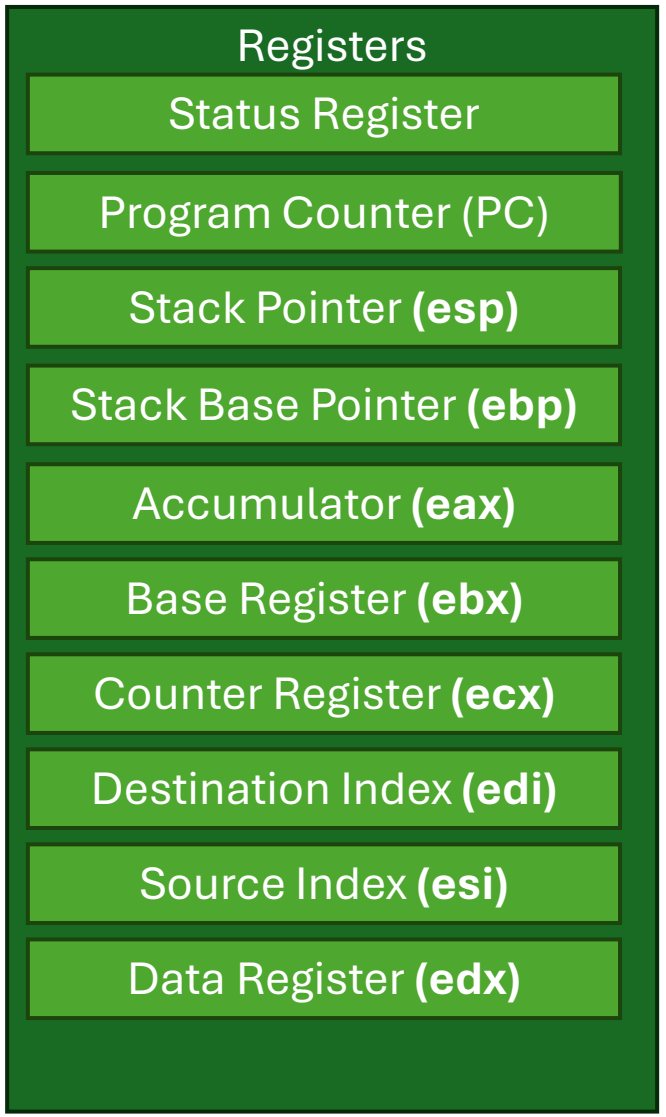


Hardware – x86 – 32 bit

Once you have a lot registers, it is common to start labelling them for similar purposes.

- These names are mostly arbitrary

Instruction	Source	Destination
movl	%0xabc123	eax



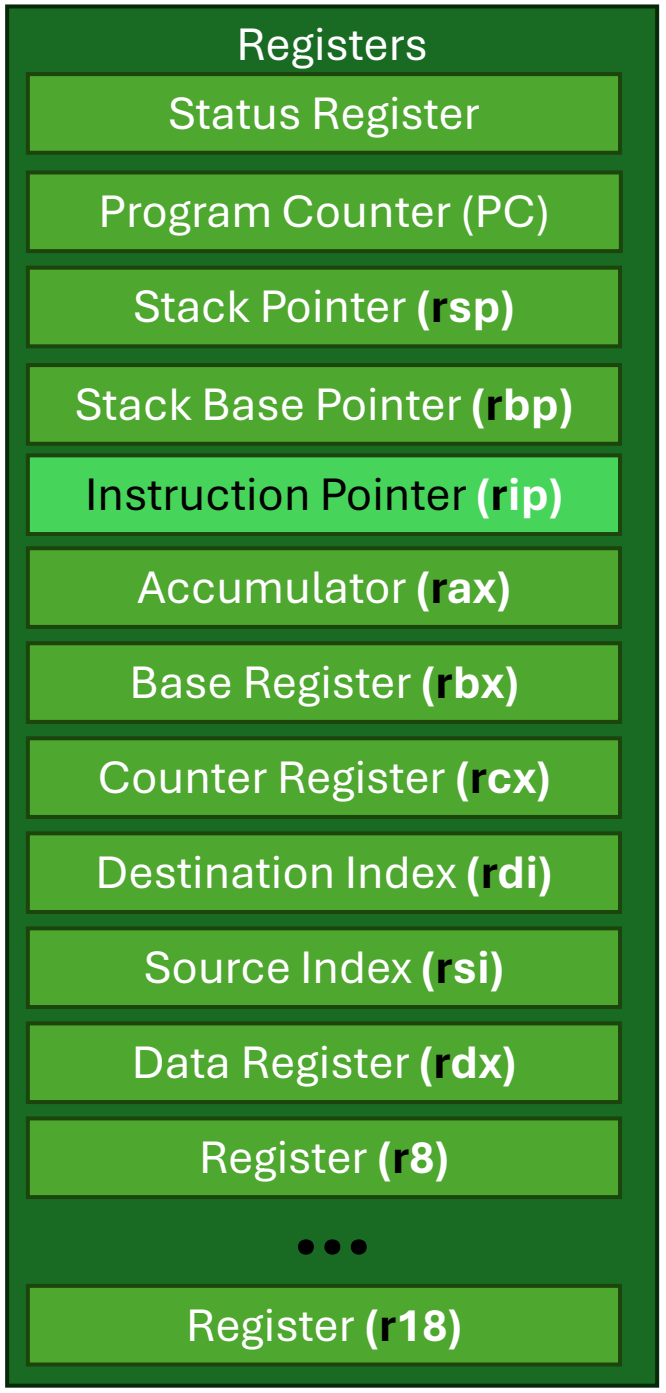
Hardware – x86 – 64 bit

Registers are renamed…

- For reasons

More general purpose registers.

Significant performance improvement.



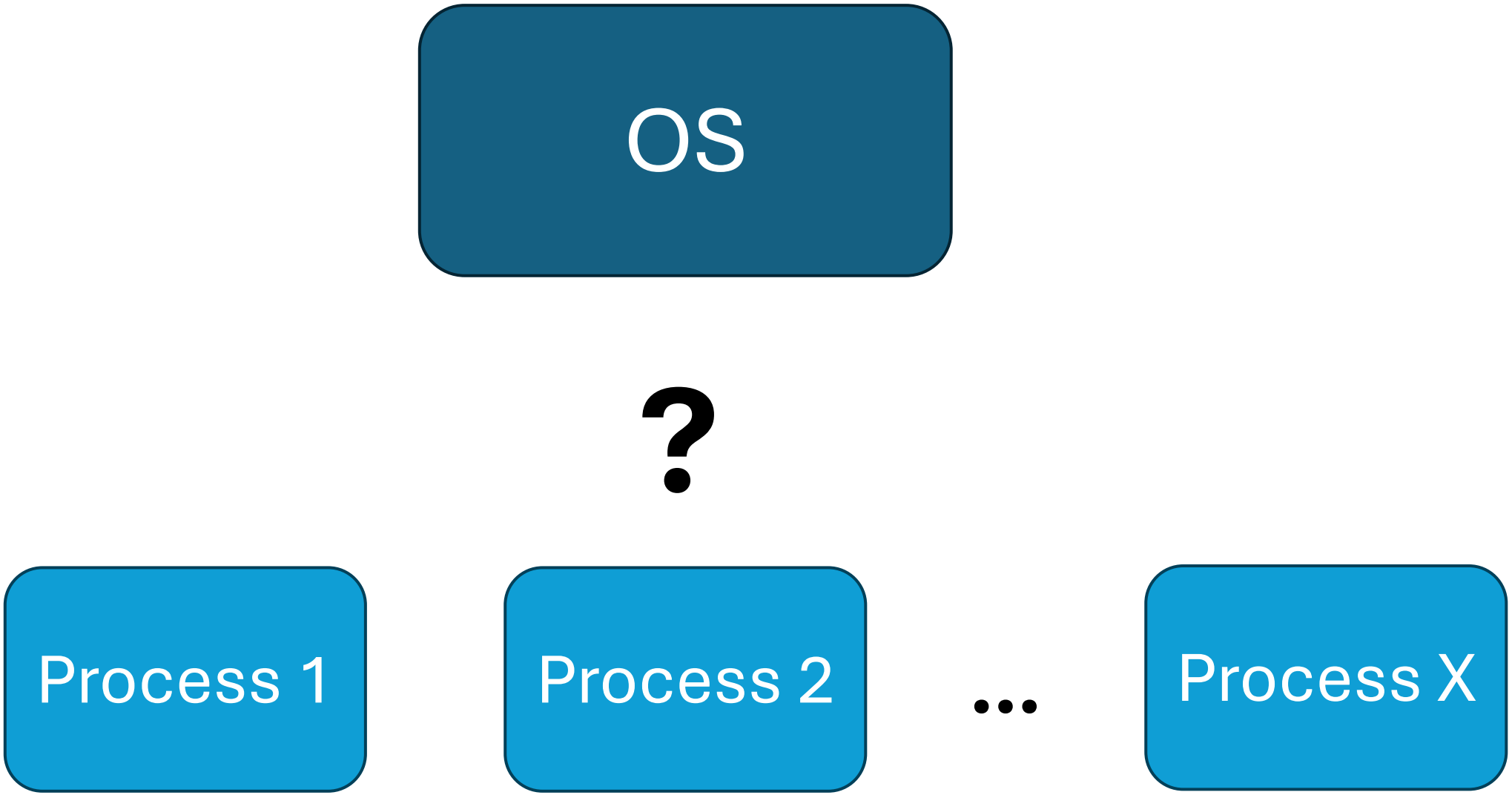
Assembly Code in Textbook

```
Instruction   Source          Destination
movl          %0xabc123       eax
push          eax
pop           eax

call          %0xff1234
call          table[ebx]
movl          [ebx]           eax
```

OS Dispatch

Running Processes

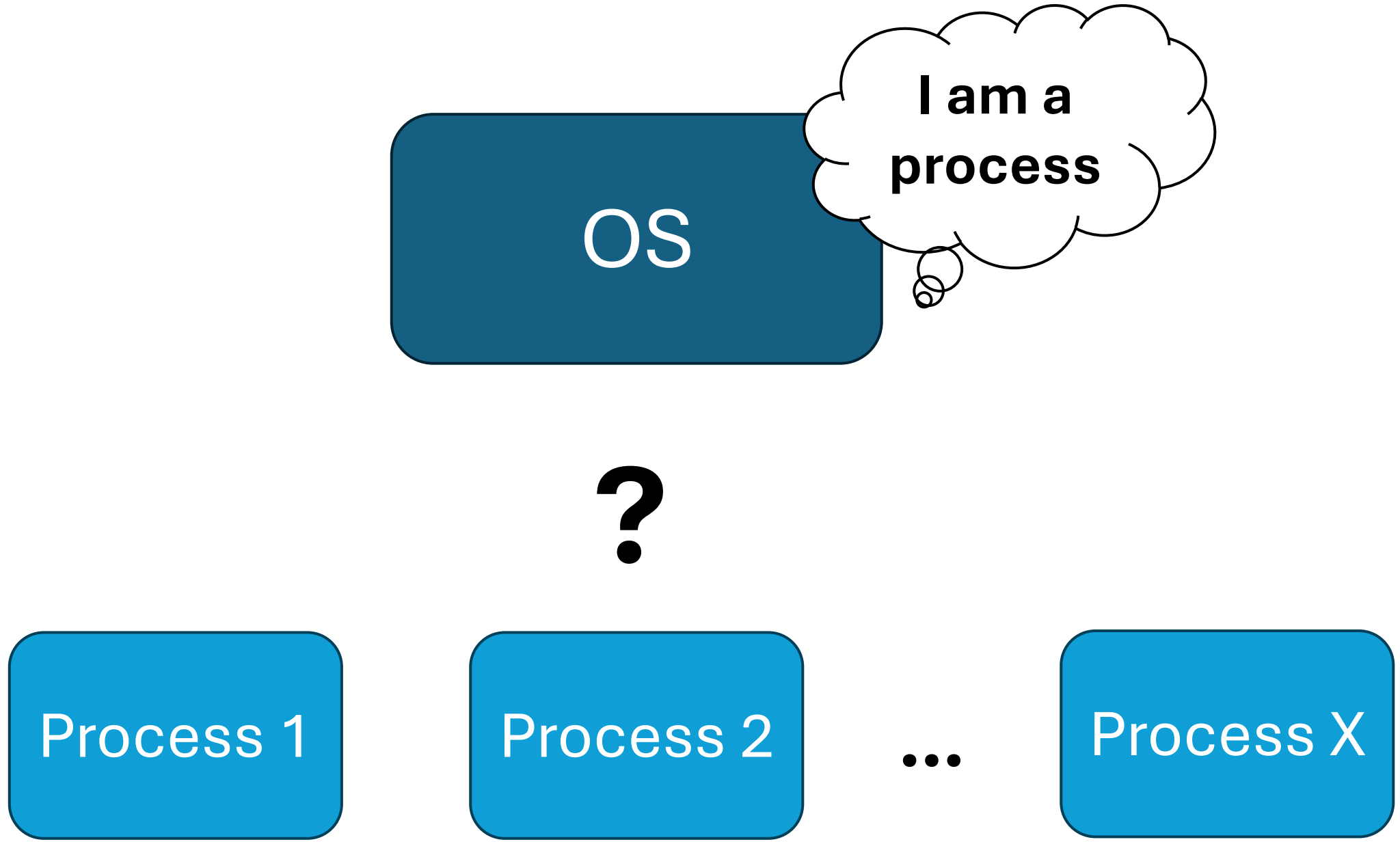

OS
?
Process 1
Process 2
...
Process X

A few questions

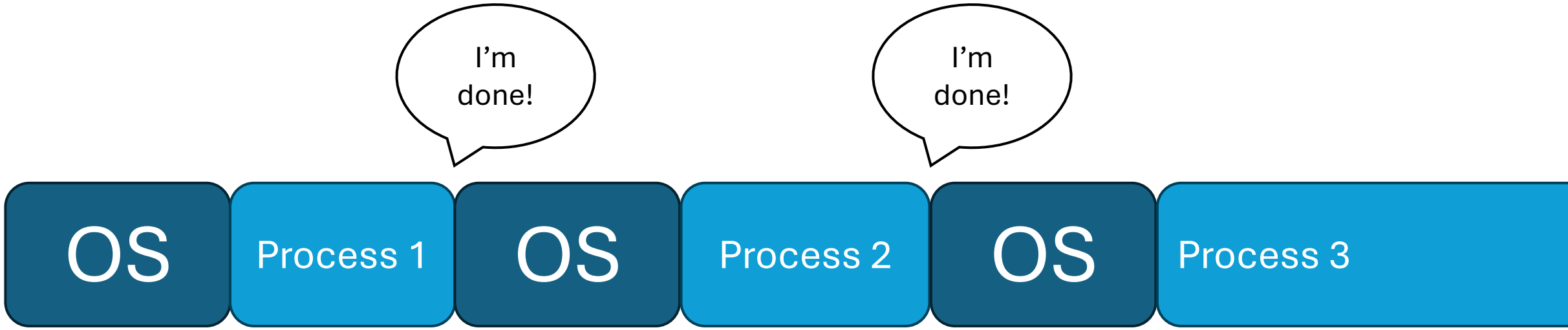
I am an OS. I control the processes.

When do I do this?

How do I do this?


OS
I am a process
?
Process 1
Process 2
...
Process X

Bad Option #1: Let the processes decide



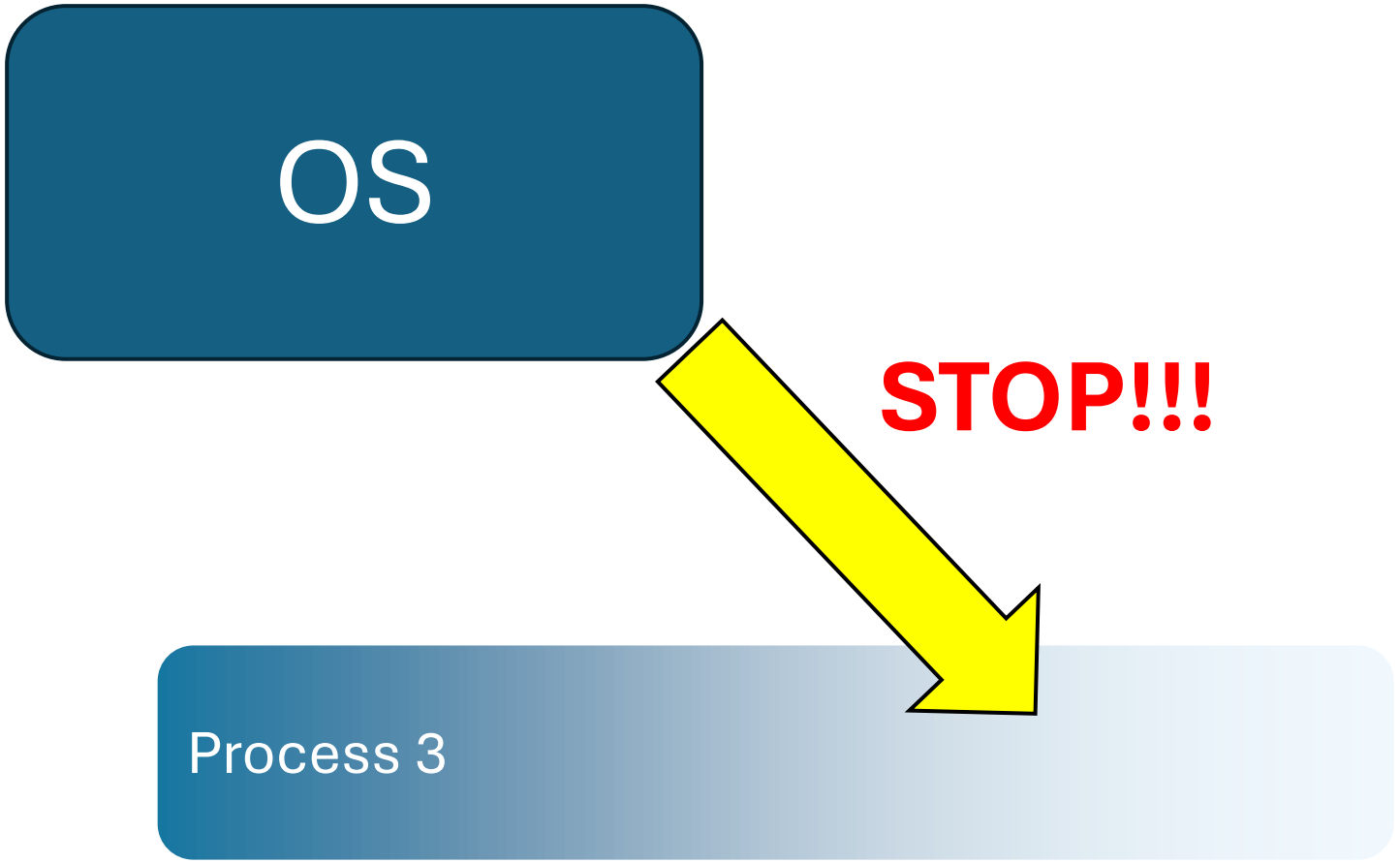
What happens when a process never wants to finish?

Bad Option #1

Doesn’t work for:

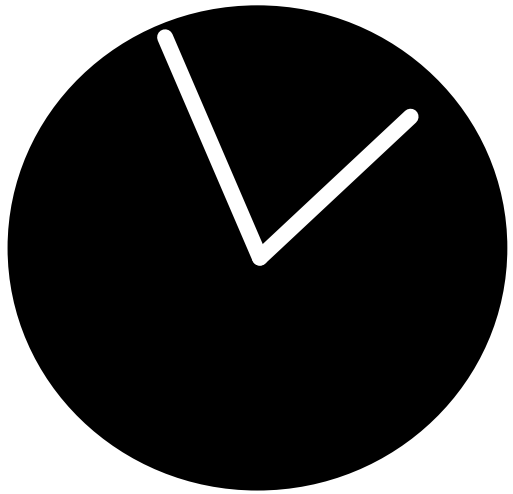
- Greedy processes — Selfish users
- Badly written processes — Bugs
- Malicious processs — Evil users

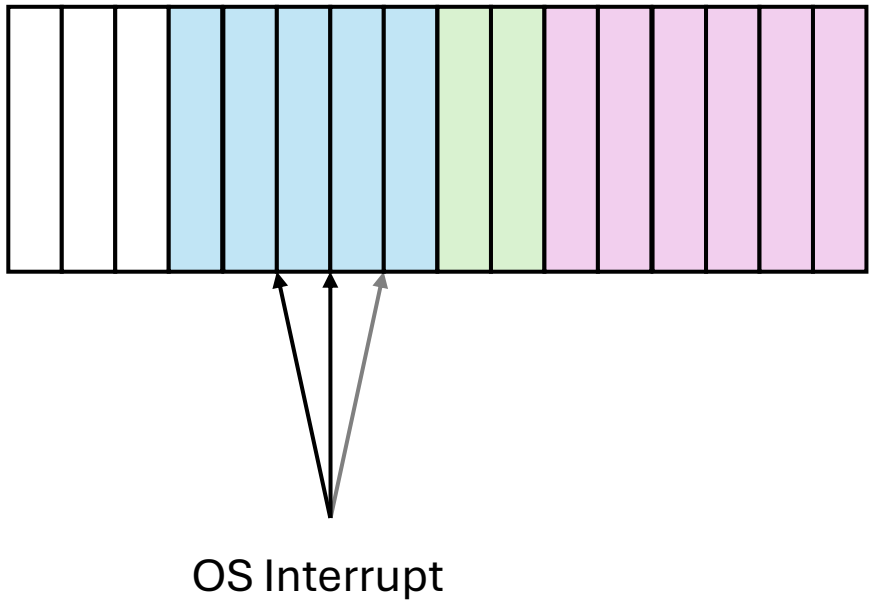
Problems to fix



The OS must be able to stop processes without the permission of the process

When do I do this?





The OS can use an external timer to regain control **periodically**

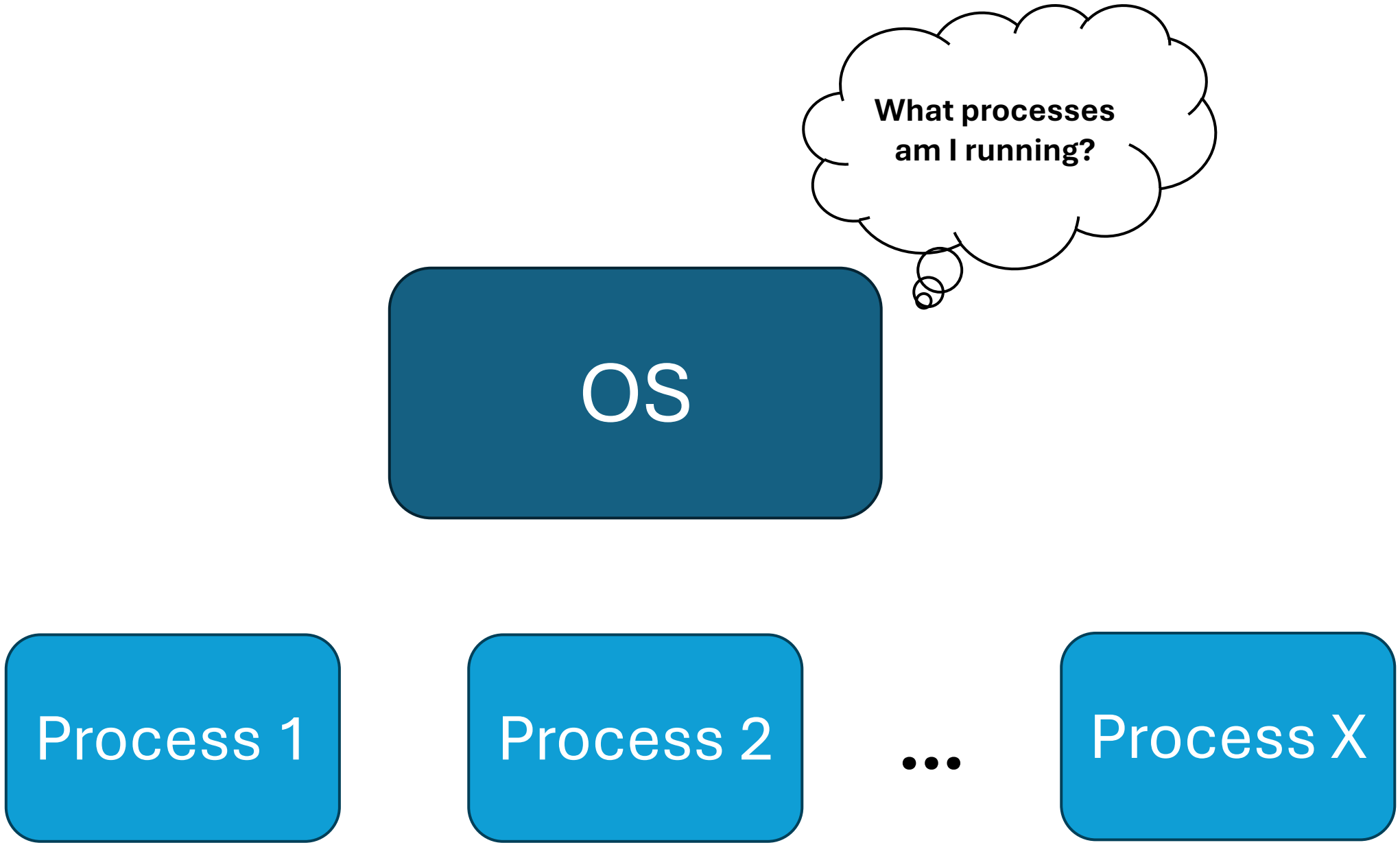
Multi-Tasking Clock

Time Slices

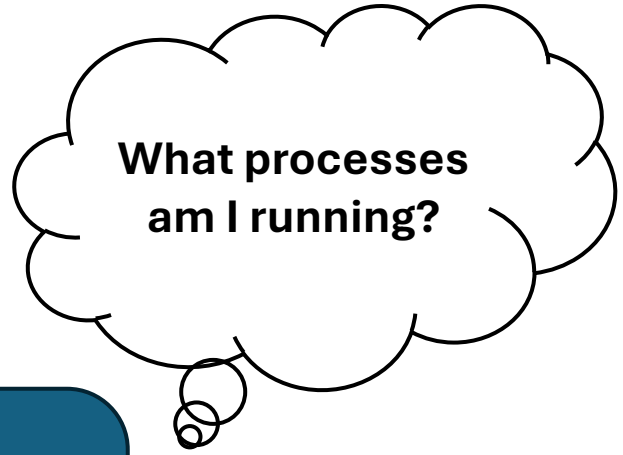
- Do a bit
- Interrupt
- Do a bit more
- Interrupt
- Do a bit more
- …
- Done

Practicalities

- Requires an external hardware interrupt
- Interrupt may not be ignored


What processes am I running?
OS
Process 1
Process 2
...
Process X

Process Control Block



OS

Process 1
PID
Process State
...

Process 2

...

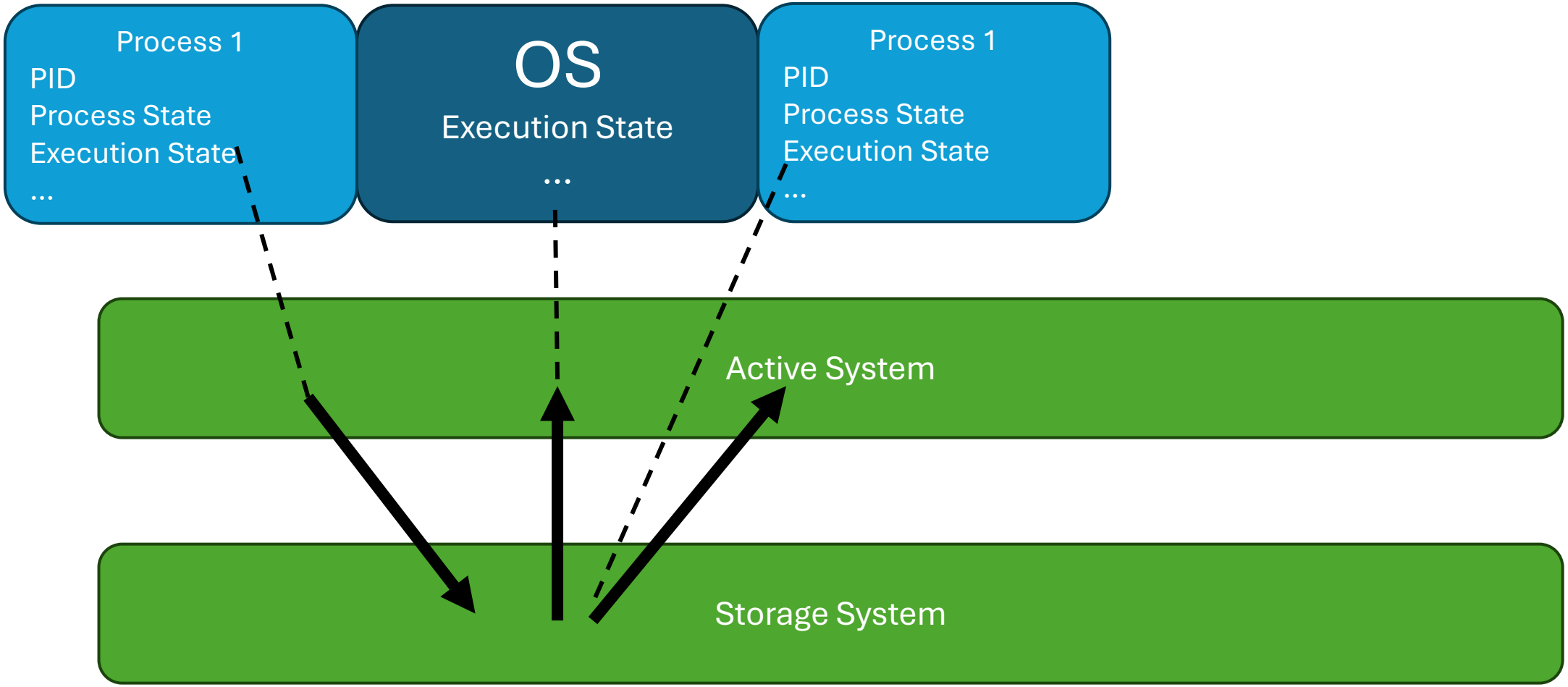
Process X

Process Control Block

What do we need?

- Process ID (PID)
- Process State (running, ready, blocked)
- Execution state (registers, program counter, stack pointer
- Scheduling priority
- Accounting information (parent/child processes)
- Credentials (who and what can it access?)
- Pointers (open files etc.)

Execution State



Switch Tasks

OS (Switch)

- Save registers for process (A) to proc-struct (A)
- Load registers for process (B) from proc-struct (B)
- Switch to k-stack (B)
- Return-from-trap (into B)

Hardware

- Trigger timer interrupt
- Change to kernel mode (i.e. OS process)
- …
- Restore the registers (B)
- Change to user mode

Process Status

Revision

- Running: On the CPU
- Ready: Waiting for the CPU
- Blocked: Asleep (waiting for something)

Runnable Processes Queue

Process X	Process M	…				

Queues

Runnable Processes Queue

Process X	Process M	...				

Waiting Queue (for I/O)

Process Y	Process Z	...				

Waiting Queue (for I/O #2)

Process W	Process R	...				

What is an Operating System?

What is it?

- A program/process
  - Controls other processes
  - Simplifies computer operation
  - Fairly distributes resources

What does it do?

- Interfaces software and hardware (**Abstraction**)
- Allocates resources

Summary

- Operating Systems
- Virtualisation
- Processes
- System Calls