CS3.301 Operating Systems and Networks

Concurrency - Introduction

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

SERC Software Engineering Research Centre

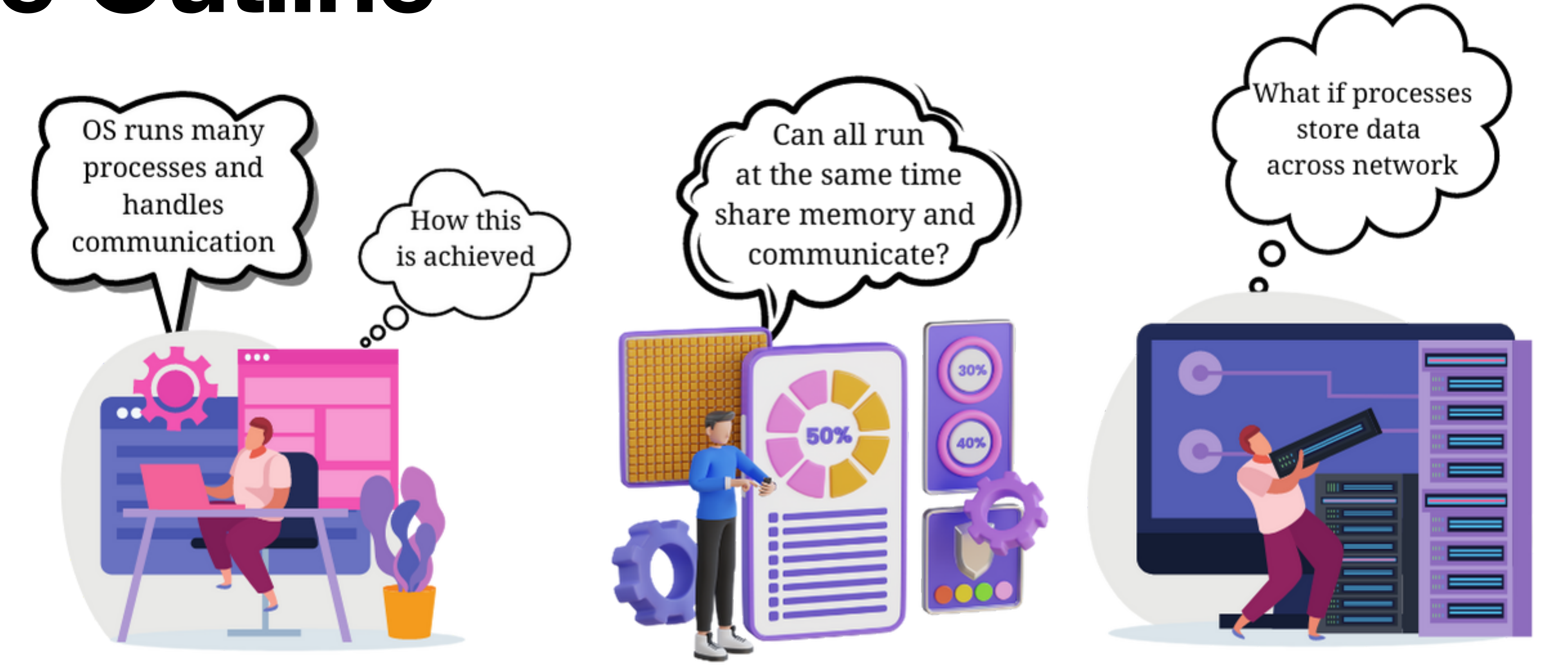
Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in three easy pieces by Remzi et al.

Course Outline



Getting the base right | Building concurrent processes that can communicate | Building a network file system

Timeline →

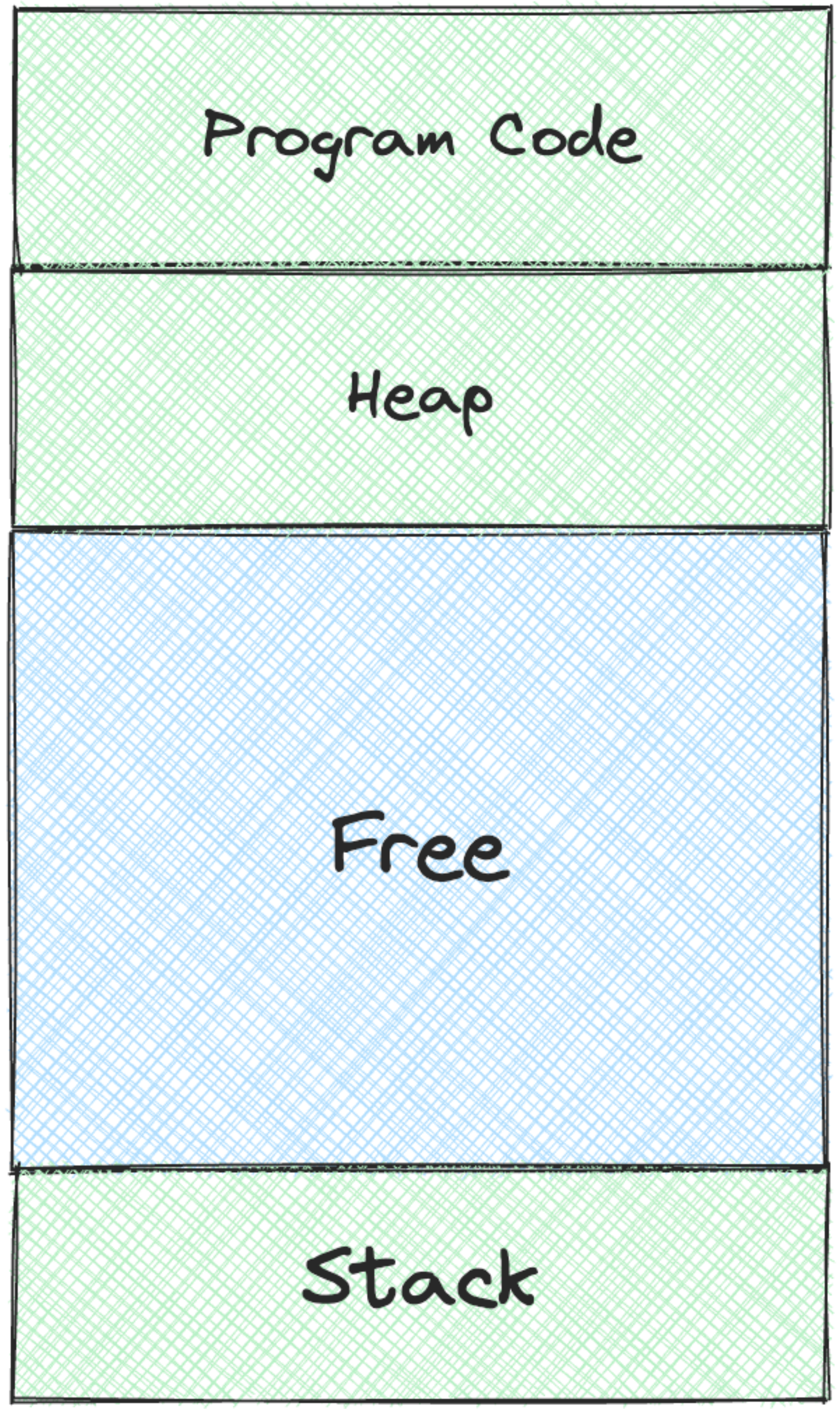
This Course

Process and Memory Virtualization	Concurrency	Persistence
Networking intro	Addressing and Routing	Network file Systems

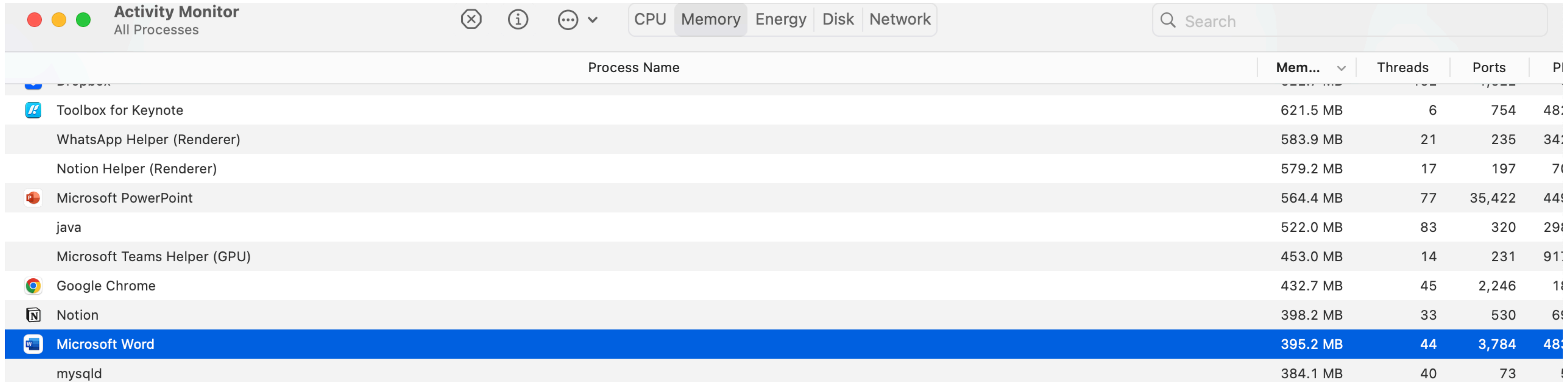
The Type of Process we have seen so far!

Some Recap

- Process during execution
  - Program Counter (PC): Points to the current instruction that is being run
  - Stack Pointer (SP): Points to the current frame of the function call
- What about the memory? - Paging!
- This is a single thread execution
- But in reality process is more than a single thread of execution



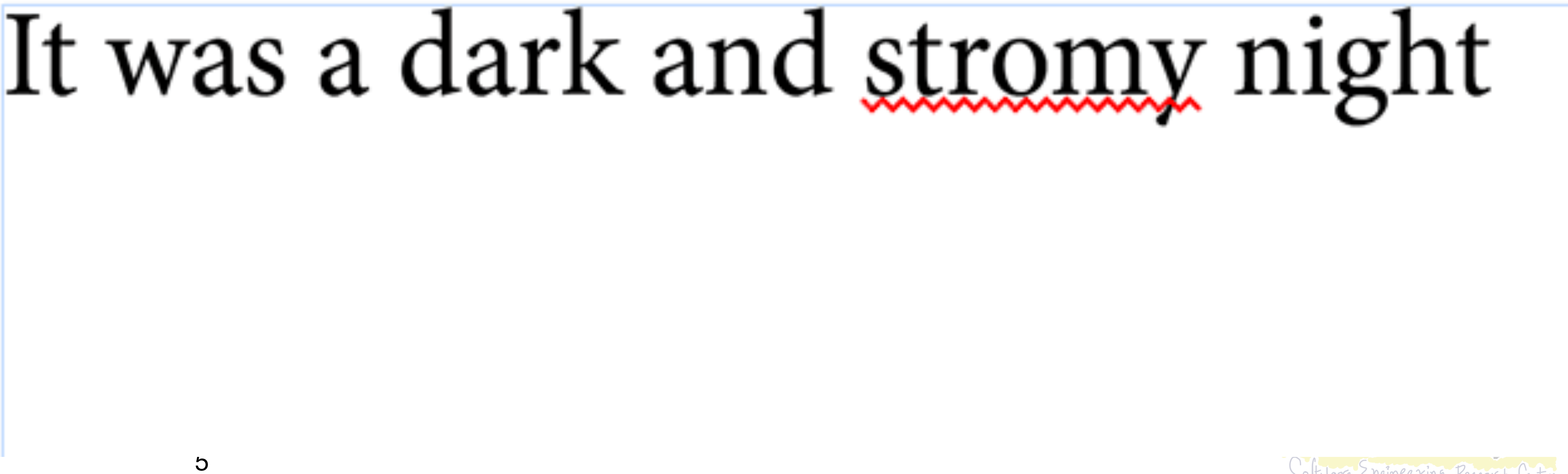
In reality a process does more things!



Check the processes running in your OS

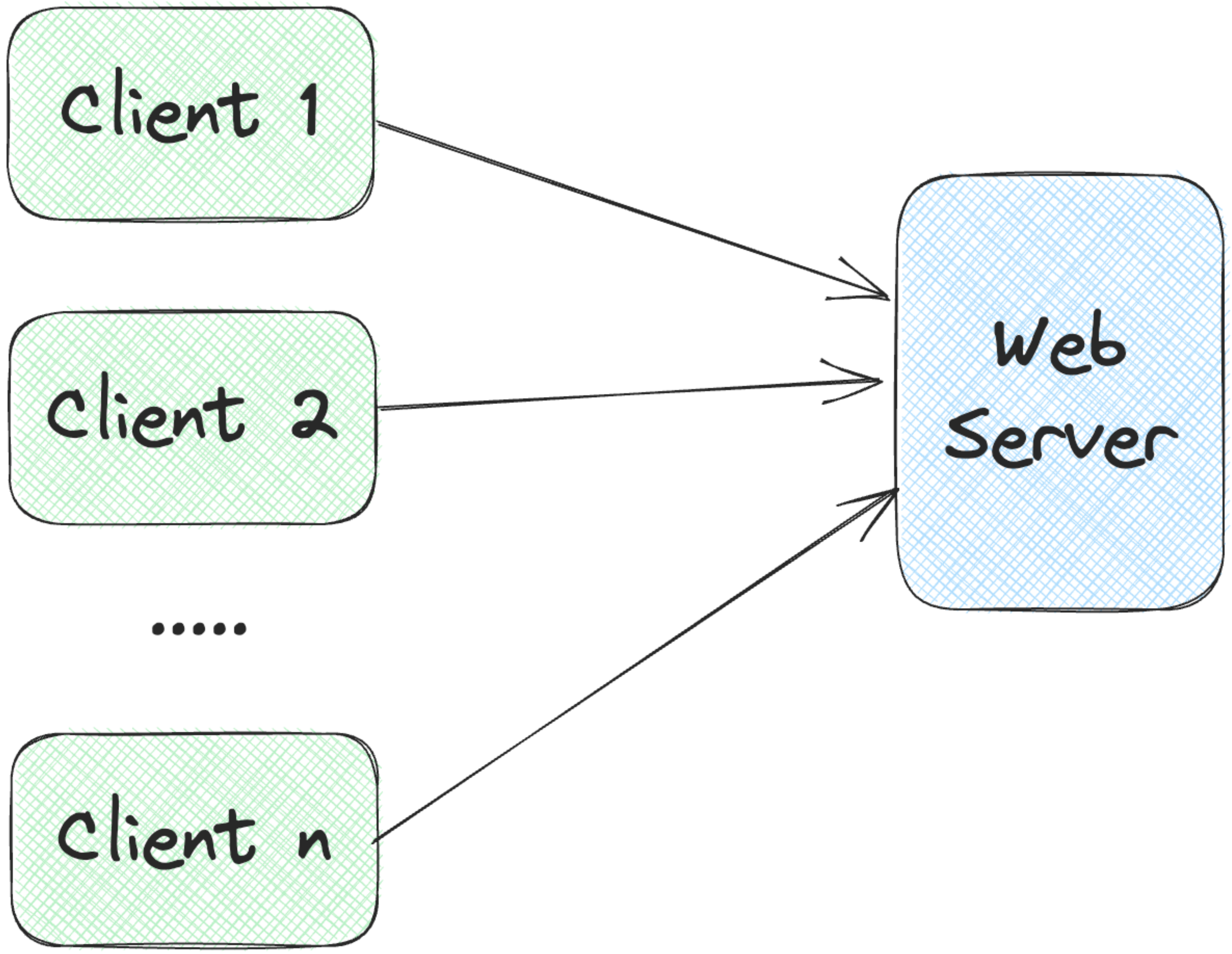
Microsoft Word is a process, while using it:

1. Spell checker works
2. Auto save happens
3. Auto formatting happens
4. ….



Think about a web server

- Web server runs a process to serve the clients
- Multiple clients may sent request to web server at the same time
- If the process handles each client sequentially - What can be an issue?
- How to make it more faster and better performing?
- What mechanism do we need? - Does multiple processes work?



An Analogy: Classrooms and Courses

Two Classrooms, two faculties teaching two different courses

Classroom 1: CS3.315 OS

Classroom 2: CS3.390 Networks

This is very similar to two separate processes

An Analogy: What if two faculties teach one course?

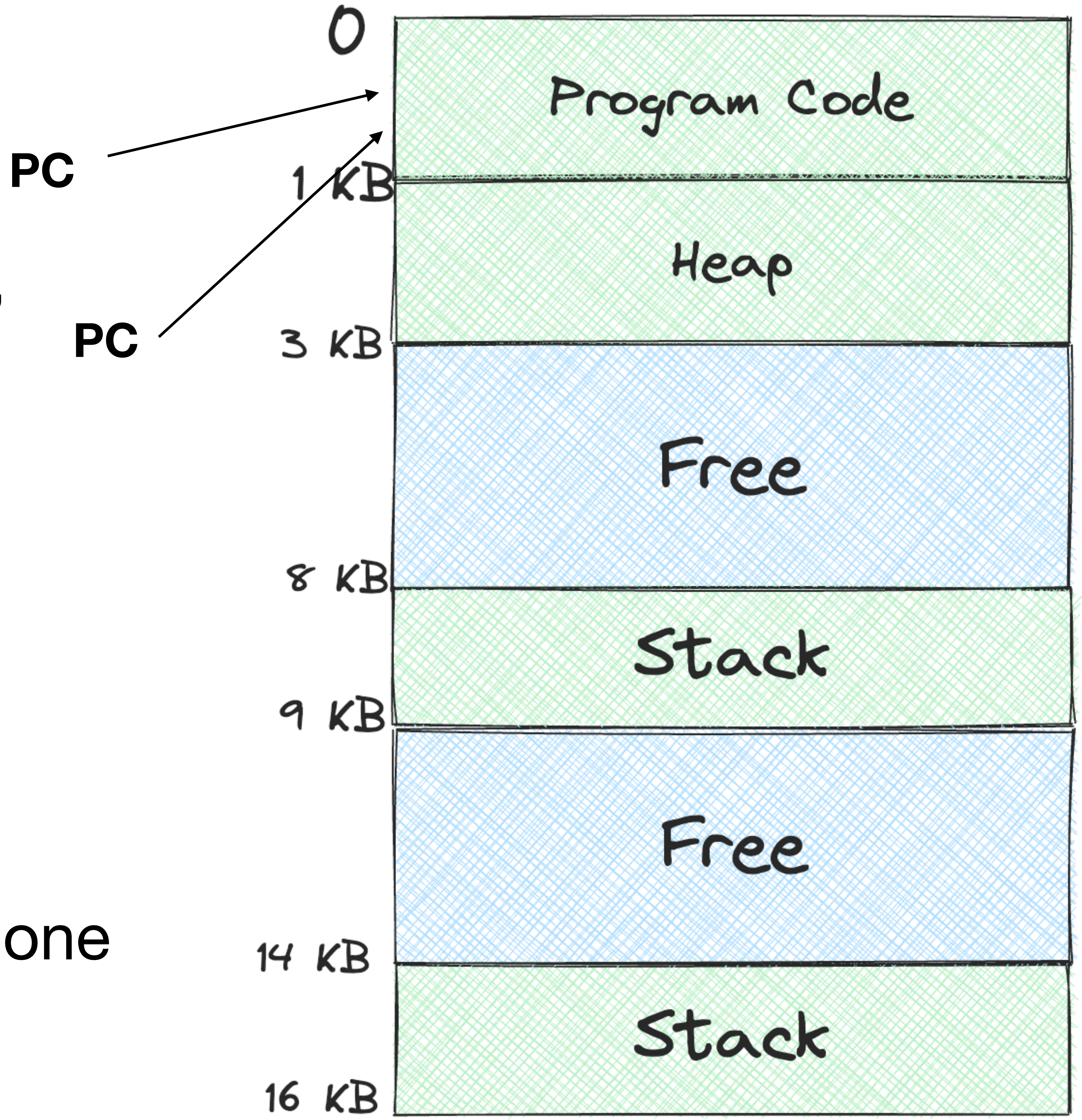
Two faculties teaching one course

Classroom 1: CS3.398 OS and Networks

- Can they teach at the same time?
  - Imagine such a scenario :-D
- Each teacher may take turns
- They may be at the class at the same time as well!
- There is only one attendance sheet, one course ID, one mark sheet
  - **Each faculty teaches in their style**
  - When question paper is set, they may take turns
  - The respective course content may be different
  - Somethings are shared!!

Process can have Threads!

- **Thread:** Another copy of the process that executes independently (lightweight process)
- Threads share the same address space (code, heap)
- Each thread:
  - Has separate Program counter
  - Separate stack for managing independent function calls
- In single thread, it was just about one PC and one stack



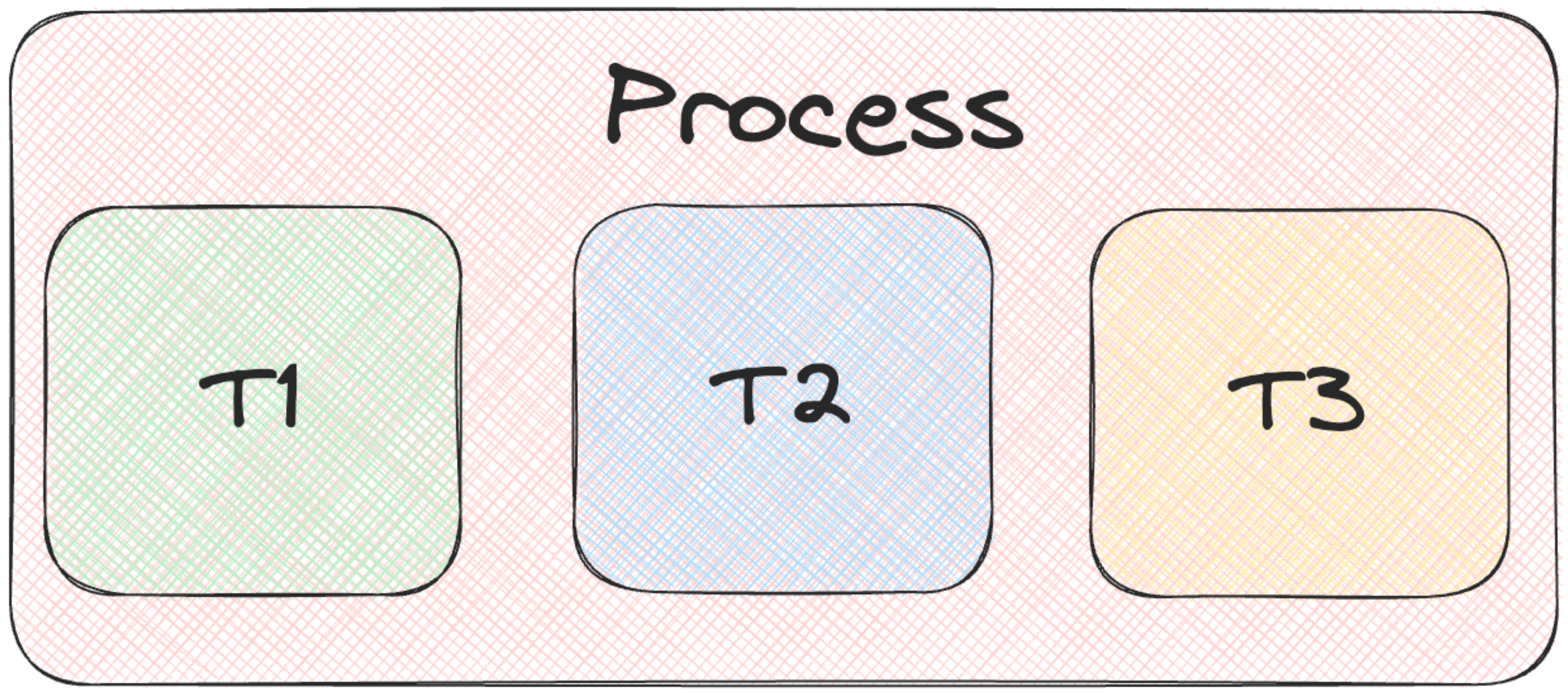
Wait, what about Process vs Threads?

Lets revisit parent and child - forks!

- What happens in a fork?
  - Parent and Child **do not share any memory**
    - Page tables are not shared, shared until changes - **Copy on Write (CoW)**
    - Subtle variations exist to improve efficiency but essentially parent and child are two different process
    - What about exec? - **Think!**
  - If they have to communicate, complicated inter process communication needs to be done (sockets, pipes, etc)
  - Extra copies of data, code, etc needs to be done

Threads

- Threads are another copy of process that executes independently
- Any process (parent process) can have multiple threads
    - Eg: Two threads T1 and T2
    - Both share the same address space - No separate page table, same code and same variables
    - Communication happens through shared variables (global)
    - Smaller memory footprint
- Threads are like separate process but share same address space

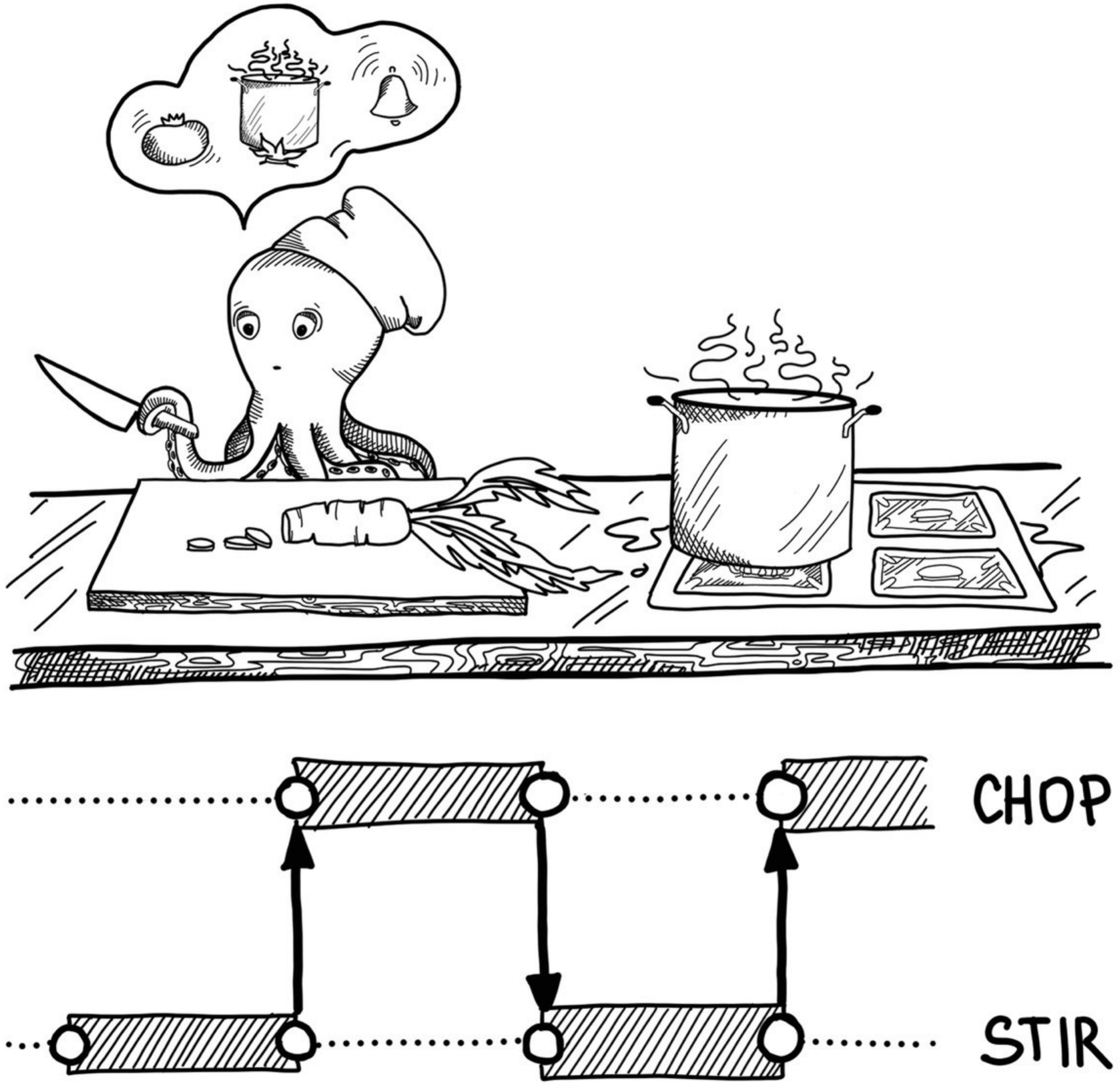


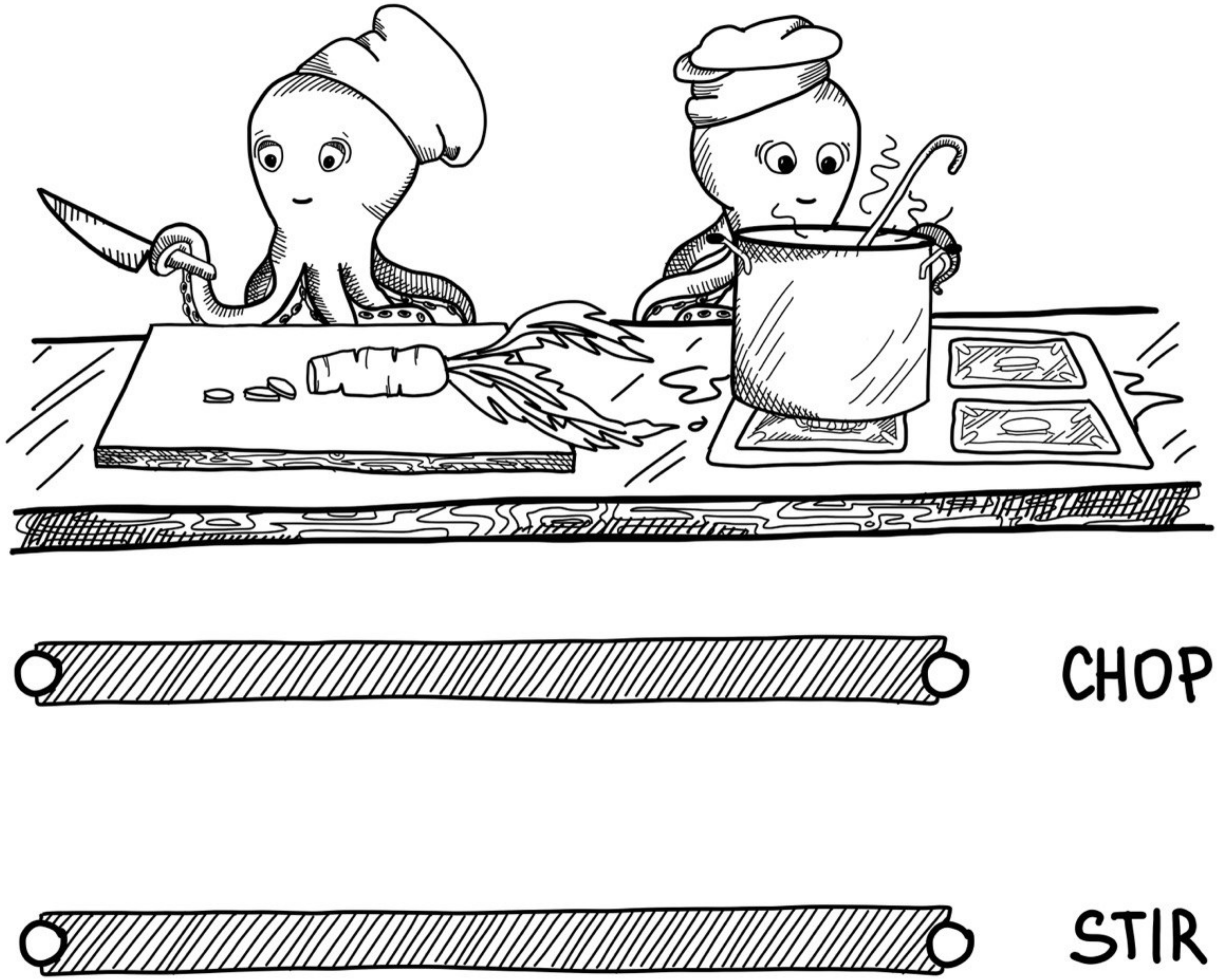
Why to do all these? Why Threads?

- Machine can be single core or multi-core:
  - Single process can effectively use multiple or even single CPU cores
  - Each thread can run independently and call different routines
  - Multi-threaded program has more than one point of execution
  - Within a process: one thread can perform I/O, one can perform computation, etc.
  - Scheduling happens between the threads - **Parallelism?**

Concurrency and Parallelism

What is what?





Source: https://freecontent.manning.com/concurrency-vs-parallelism/

Concurrency Vs Parallelism

Concurrency is about dealing with lot of things at once while parallelism is doing lot of things at once

- **Concurrency:** Running multiple threads/processes at the same time, even on a single CPU by interleaving their executions
- **Parallelism:** Running multiple threads/processes in parallel over different CPU cores
- Concurrent computations can be parallelized without changing correctness of result
- Concurrency by itself does not imply parallelism and vice versa
- Parallelism can be thought of as subclass of concurrency

Scheduling Threads

- OS schedules threads that are **ready** similar to scheduling processes
- The context of thread (PC, registers) is saved into/restored from **Thread Control Block (TCB)**
    - Every PCB can have one or more linked TCBs corresponding to threads
- OS also has kernel level processes, each has threads - **Kernel threads**
    - **Kernel threads** can perform various tasks - system calls, handling interrupts, background tasks, etc. Execute in kernel mode. Eg: Linux pthreads
- **User threads** - managed by user level libraries. Execute in user mode
    - Eg: POSIX threads, anything that need not be managed by kernel

Creating a Thread

- POSIX provides interface for management of threads - pthread.h

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,

void *(*start_routine)(void *), void *arg)

- ***thread:** Pointer to pthread_t variable
- ***attr:** holds the attributes for new thread, stack size, scheduling policy, etc. NULL points to default
- ***start_routine:** Pointer to the function that will be executed by the thread upon execution
  - Takes a single void parameters and returns void value
- ***arg:** Argument that will be passed to the start_routine function
- Returns 0 if thread successfully created

Some Interesting things to be considered

```
void *worker_thread(void *arg)
{
    printf("%s\n", (char *) arg);
}
```

```
Starting the threading demo
thread 1
thread 2
end
```

```
Starting the threading demo
thread 2
thread 1
end
```

- Order of execution can be non deterministic
- Its hard to predict which thread executes first
- Two executions have two different sequence here!!
- So what could have happened?

An Ideal Trace

main	Thread 1	Thread 2
Running prints “Starting the threading demo” Creates T1 Creates T2 Waits for T1		
	Runs Prints “thread 1” Returns	
Waits for T2		Runs Prints “thread 2” Returns
print “end”		

This can also happen!

main	Thread 1	Thread 2
Running prints “Starting the threading demo” Creates T1		
	Runs Prints “thread 1” Returns	
Creates T2		Runs Prints “thread 2” Returns
Waits for T1 Waits for T2 prints “end”		

Shared Data - More Tricky

```
void *worker_thread(void *arg)
{
    int index;
    for (index =0; index<max_index; index++)
    {
        counter++;
    }
}
```

```
Inital value of the counter 0
Final value of the counter 4000
```

```
Inital value of the counter 0
Final value of the counter 3790
```

- Max size: 2000, assume global variable counter initialised to 0
- Desired final result: **4000!, even on a single processor system there is no guarantee**
- **Why does this happen?**

Lets break the code down in assembly

counter = counter + 1

```
mov 0x8049a1c, %eax
add &0x1, %eax
mov %eax, 0x8049a1
```

1. Load memory value to register eax
2. Increment the value in the register by 1
3. Move the value from register back to memory

What can happen?

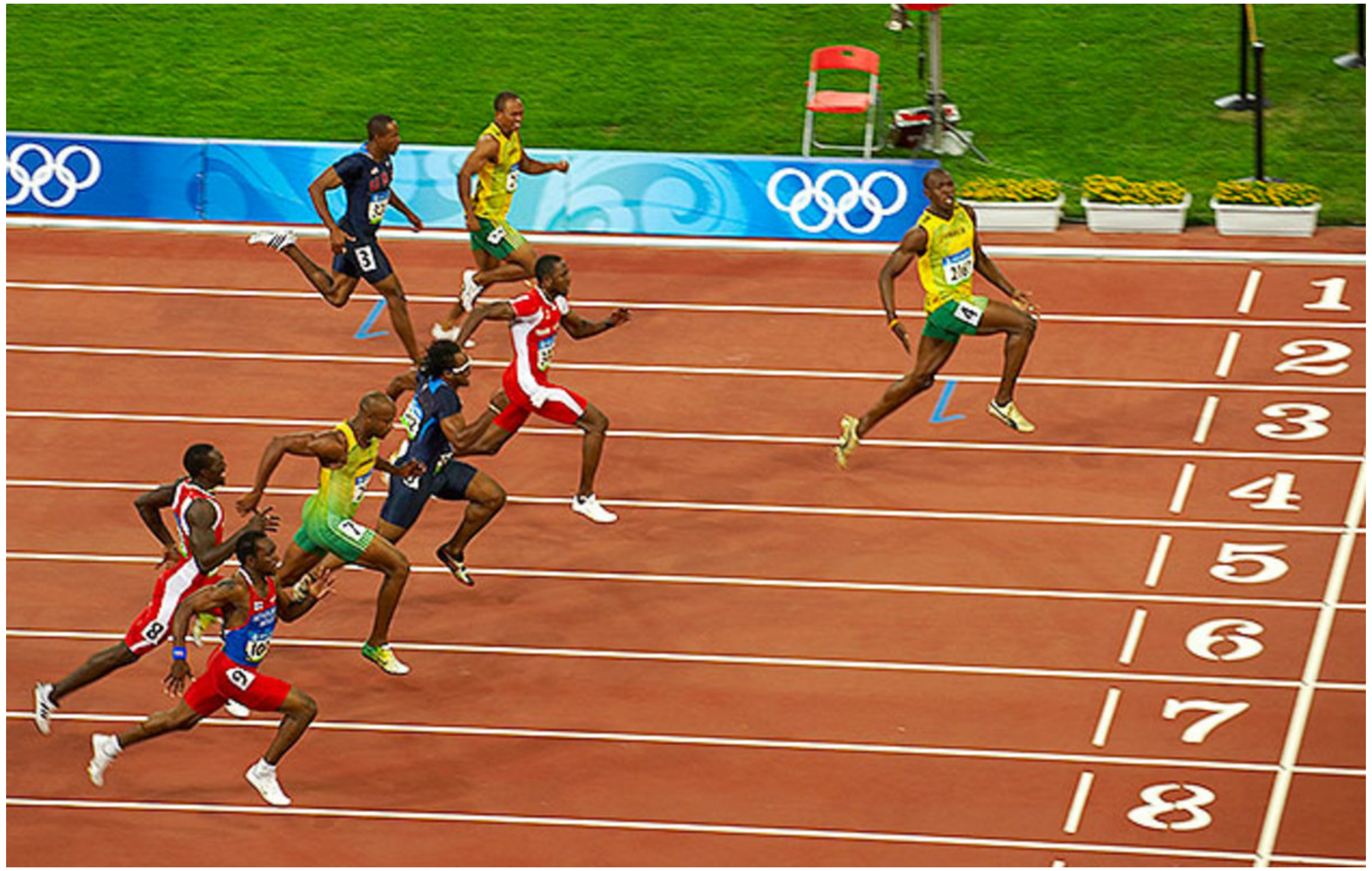
OS	Thread 1	Thread 2	Counter and Register (Initial value of counter = 50
	mov 0x8049a1c, %eax add $0x1, %eax		eax = 51 **counter = 50**
interrupt **Save T1' state** **Restore T2's state**			

		mov 0x8049a1c, %eax
		add $0x1, %eax
		Mov %eax, 0x8049a1c
			eax = 51
			counter = 51

interrupt **Save T2' state** **Restore T1's state**			

Race Condition and Critical Section

- **Race Condition: Condition where**
  - Multiple threads executing concurrently and
  - results depend on order of execution (time)
  - Scheduler can swap threads, also interrupts
  - Non-deterministic results
- **Critical Section:**
  - The section of code that is shared between the threads (leads to race conditions)
  - Shared variables or data



Source: https://www.si.com/olympics/2016/07/14/usain-bolt-2016-rio-olympics

Concurrency is tricky!

Race conditions can result in fatal issues

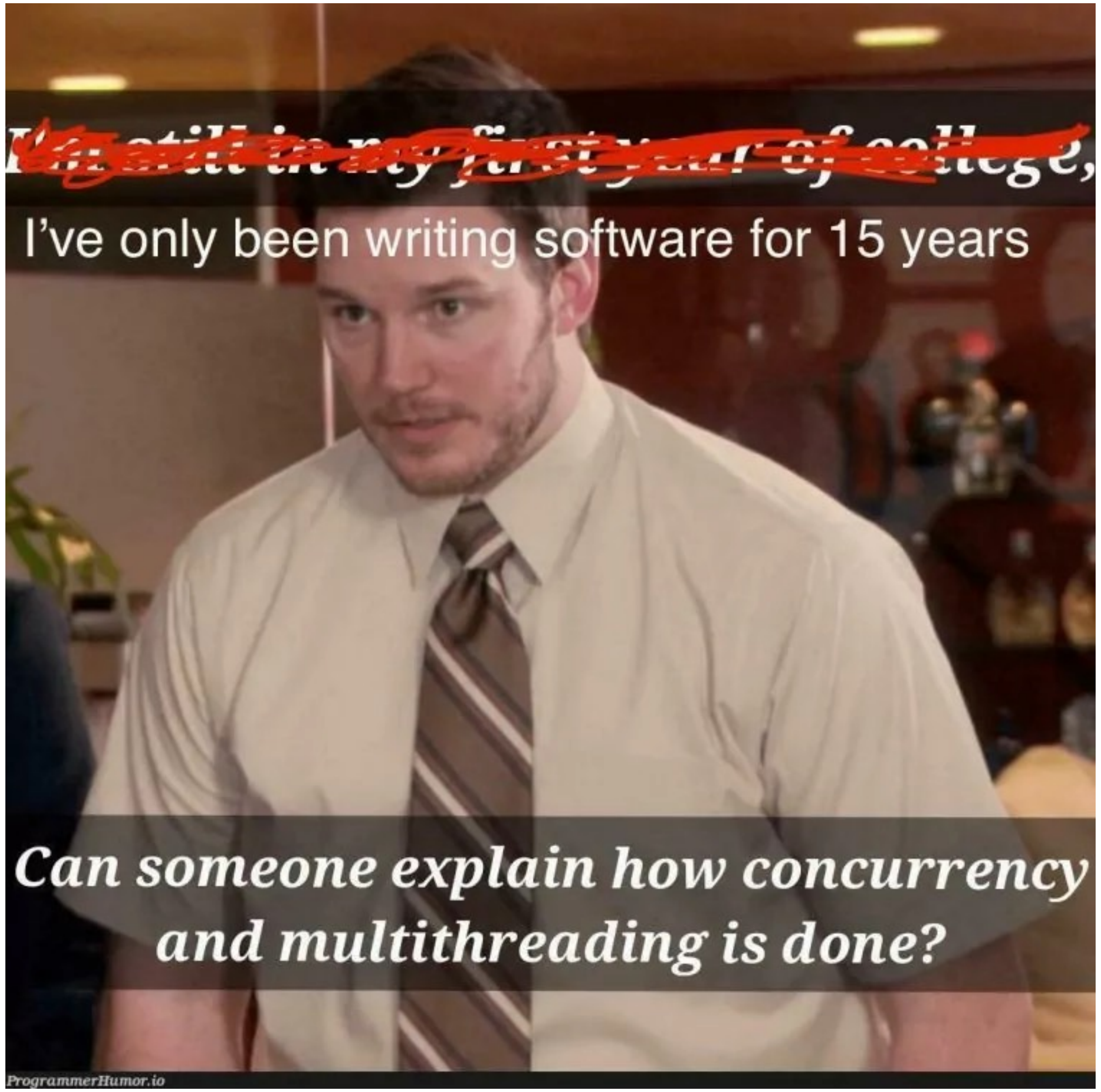
Therac 25

Northeast Blackout of 2003, US

https://www.everydayshouldbesaturday.com/2018/8/14/17687734/flashback-the-blackout-of-2003

Source: https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

Concurrency can be tricky!



Source: programmerhumor.io

What can be done?

Bring in Atomicity

- What we want here is mutual exclusion!
  - When one thread is accessing critical section, others should wait
  - No two threads should access critical section at the same time
- In other words, **atomicity** needs to be provided
  - What if there was one instruction in assembly:
    - memory-add 0x8049a1c, &0x1 - Reality is not this!!

```
mov 0x8049a1c, %eax
add &0x1, %eax
mov %eax, 0x8049a1
```

This should execute atomically!!

What we need?

- Need to build synchronization primitives
  - Hardware + software support
  - Ensure that critical section is accessed in synchronised and controlled manner
- **One part:** Build some primitives for synchronisation
  - This will ensure atomicity (avoid race conditions)
- **Second part:** Ensure every thread gets access!
  - No one should starve

Some Issues needs to be addressed

- What support do we need from hardware?
- What support is needed from software?
- How to build primitives correctly and effectively?
- How can programs use these primitives?

SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere