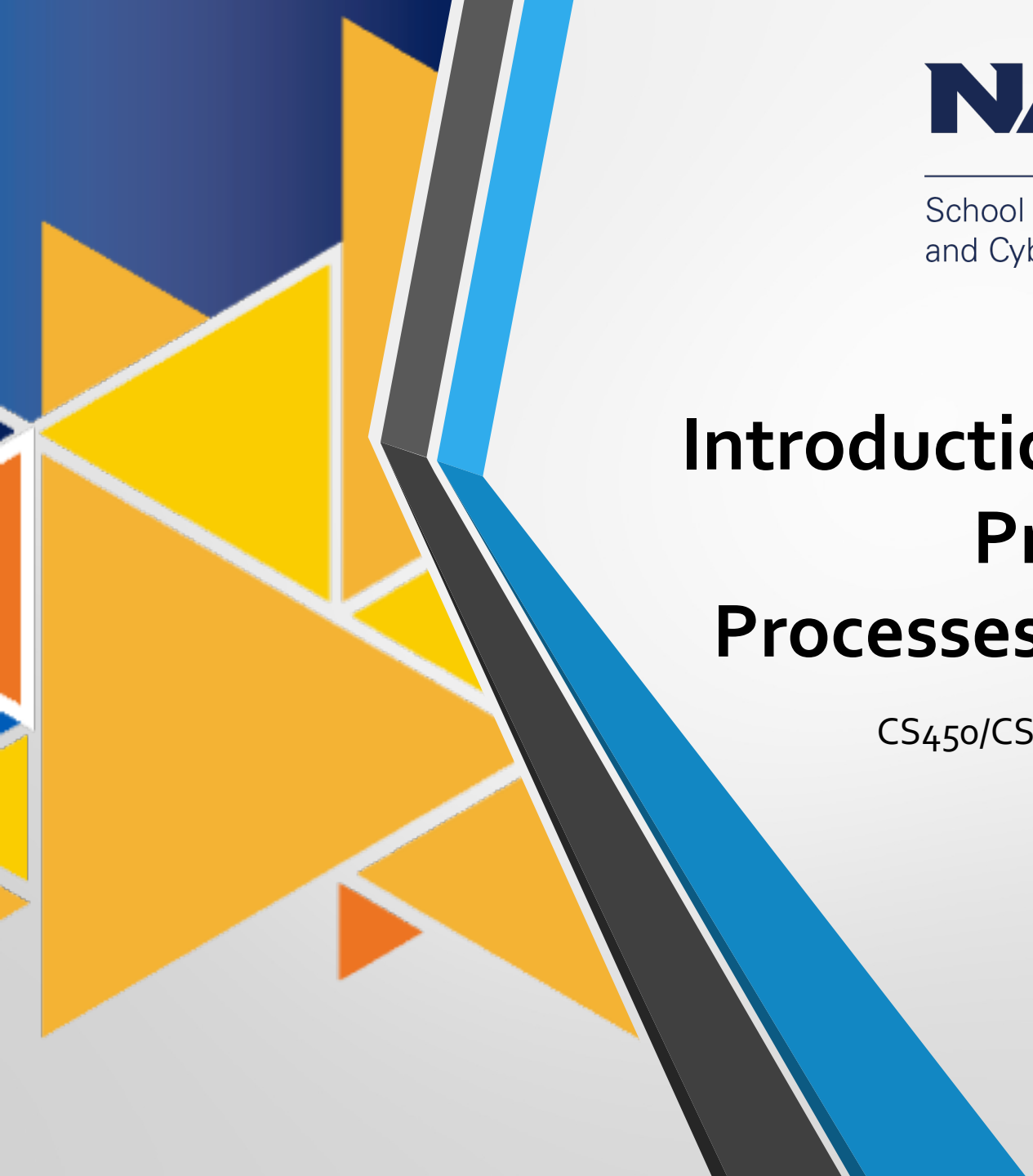


NAU NORTHERN ARIZONA UNIVERSITY

School of Informatics, Computing, and Cyber Systems

Introduction to Parallel Programming. Processes and threads

CS450/CS550: Parallel Programming

Week 2-1

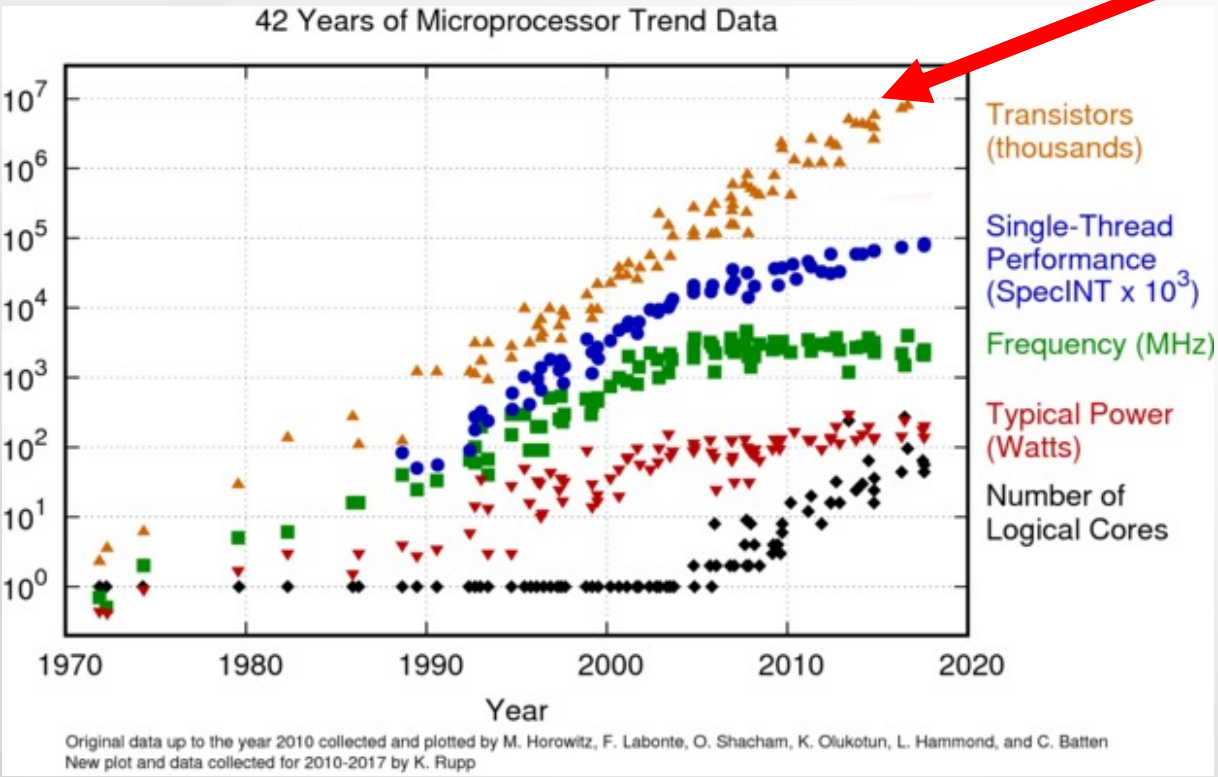
Basic definitions

- **Parallel computing** is the use of *parallel computer* to reduce the time needed to solve a single computational problem
- **Parallel computer** is a multiple-processor computer system supporting *parallel programming*
- **Parallel programming** is programming in a language that allows to explicitly indicate how different portions of the computation may be executed concurrently by different processors

Parallel computers

- Parallel computers are everywhere:
  - All smartphones are parallel computers
  - The iPhone 14's A15 chip has a 6-core CPU
  - Android phones (e.g., Samsung S22) can have even 8 cores
- Commodity machines:
  - First dual core commodity processor IBM POWER4 was introduced in 2001
  - Apple's M2 offers an 8-core CPU
  - Intel Core i9 13th gen. processor has up to 16 cores
  - AMD Epyc Bergamo processor has up to 128 cores

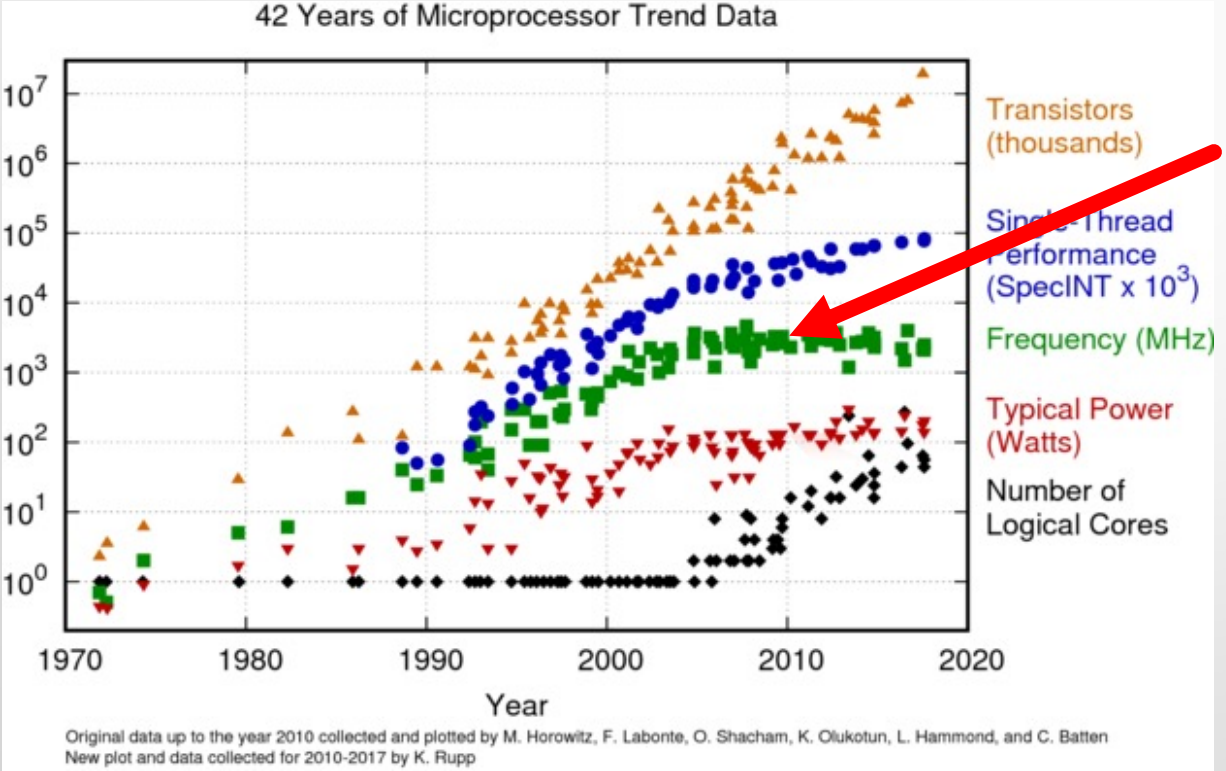
General hardware trends



Moore's Law: the number of transistors on a chip doubles roughly every 2 years (increase in transistor density)

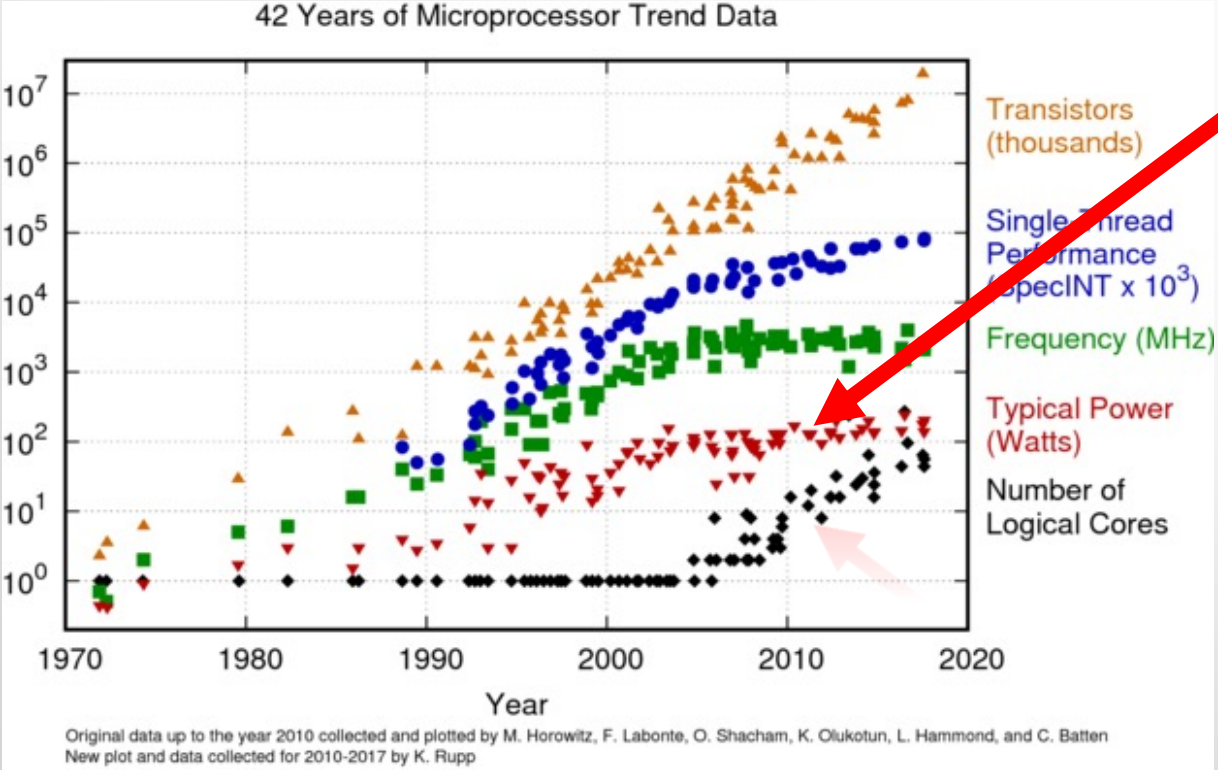
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

General hardware trends



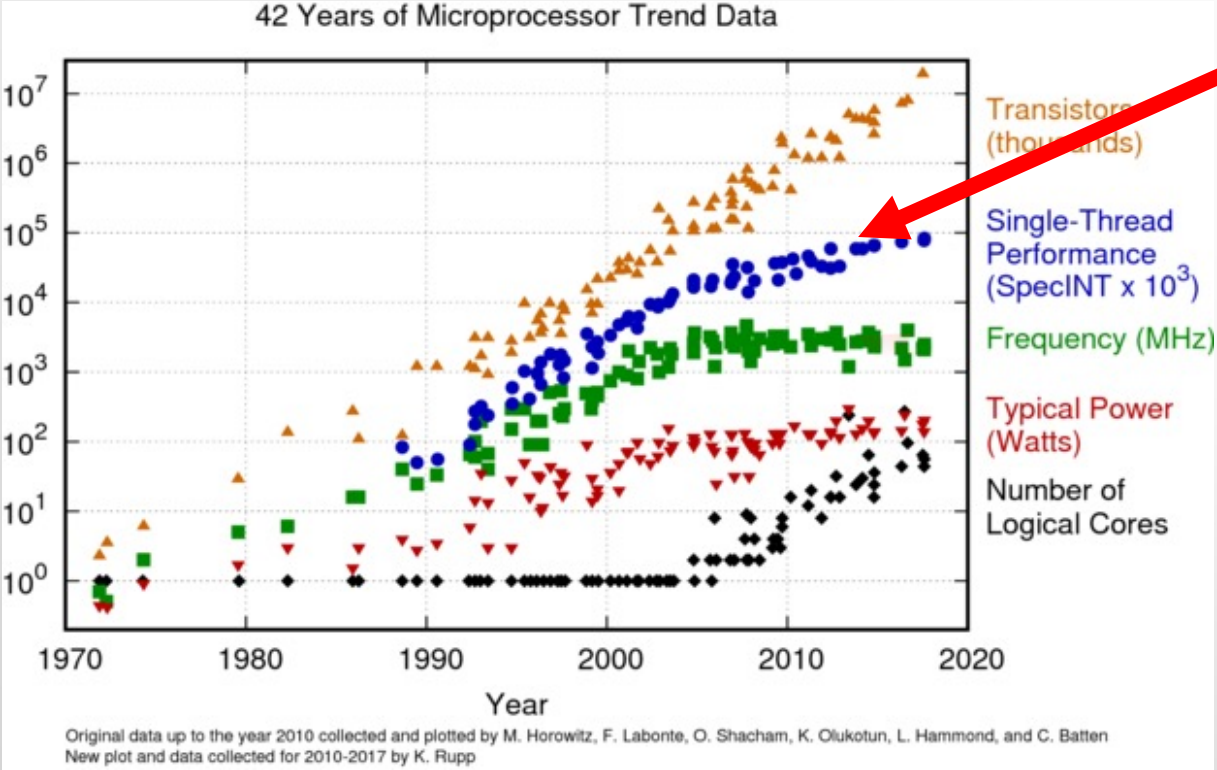
Clock speeds have peaked by mid 2000's

General hardware trends



Power has plateaued since around the same time

General hardware trends



Performance per cycle has plateaued since around 2000: this refers to the optimizations and parallel optimizations that occur in a single chip or core

Summarizing general trends

- Number of transistors are increasing
- Clock speed, power, performance/cycle fixed
- Implications?
  - CPUs must become more power efficient (more transistors for the same power budget)
  - For performance: cannot rely on clock speed or hardware optimizations (performance / cycle improvements)
- Need to use **parallel computing** to improve the performance of the code

Concurrency

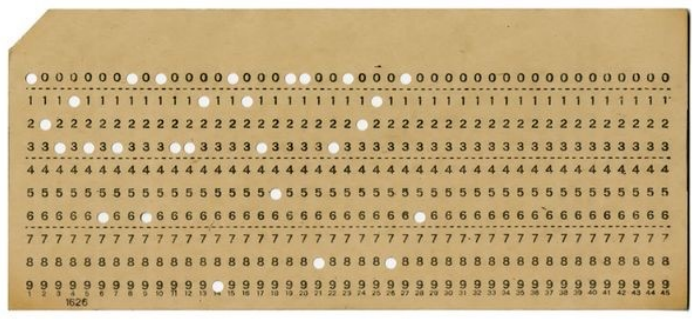
- **Concurrency** is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the outcome
- **Concurrency** allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems
- Concurrency is about **dealing** with lots of things at once, but parallelism is about doing lots of things at once
- Concurrency is about **structure**, parallelism is about execution, concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable

Concurrency

- In a concurrent program, the main idea is to design a program in terms of individual tasks
  - Each of these tasks has a job
  - Tasks may need to communicate between each other
  - These tasks may be in different regions of the code or the same region *at the same time*
- Code is usually decomposed into tasks through objects, functions, etc., for different reasons, e.g., the code's reuse, or increase of the code's readability
- Still, parallel programming requires thinking much differently about the program

Historical retrospective of concurrency

- The first computers only allowed one user to use it at once
- For example: A programmer needs the computer on Monday from 9:00 am to 12:00 pm:
  - he goes into the machine room
  - puts in punch cards
  - there are bugs in the program
  - he debugs the program and gives it the modified punch cards



- The situation is awful, not because of the punch cards, but because when the programmer is debugging and thinking about the problem, the computer is just sitting there doing absolutely nothing

Solution: batch processing

- Instead of reserving the computer for a significant fraction of time, the jobs get submitted to a **job queue**
  - The queue then executes the jobs in some order (some programmers can have a priority over others)
  - When the program stops, from failure or it completes as expected, the next user in the queue gets their job executed
- This makes sure that the machine is at least somewhat active if the job queue isn't empty
  - But the computer has many different functions: some part of the system resources will still be idle, and some will be utilized
- Back then, I/O took forever, while the CPU was idle with nothing to do
  - In certain regards, this is still the case today

History of concurrency

- Multiprogramming (1960s)
  - Multiple programs are stored in memory at once due to increased memory capacity
  - This requires mechanisms to keep the programs separate from each other and interruptions (priority signals to the CPU that the code needs to be interrupted)
- Time sharing (1970s)
  - Multiprogramming, where multiple jobs get executed through time-sharing
  - Process 1 executes for 5 s, then process 2 executes for 5 s, and so on (this is context switching): this concept is similar to round-robin scheduling
- Over time, this trends led to concurrency within the same application
  - That is, you think of your application as several concurrent tasks
  - The system will execute these tasks at the same time
  - This is parallel or concurrent computing

Concurrent programs

- Thinking about what a parallel program does is a lot more challenging than for a sequential program
  - A programmer should foresee, what is one task doing when another task is doing something else
  - This is incredibly difficult for numerous reasons
  - Executing two parallel programs may not execute identically
- Concurrent programs
  - Are more difficult to design to ensure that the program has correct output
  - Very challenging to read
  - And very challenging to debug

Designing the program using tasks

- Instead of thinking of one monolithic program, we should think of a collection of **tasks**
- Ideally, every task would be independent of each other:
  - No communication between tasks
  - Tasks do not rely on each other
- Programming languages often support task-based constructs

Concurrent task abstraction

- The operating system can support the abstraction of concurrent tasks
  - The operating system allows multiple processes to run at the same time
  - Therefore, it can also allow for the tasks inside a program to execute concurrently
- Ways that this can be accomplished:
  - Special libraries
  - System calls from the program to the operating system
  - Java virtual machine (threads scheduled by the VM instead of the OS)
  - A hybrid of these

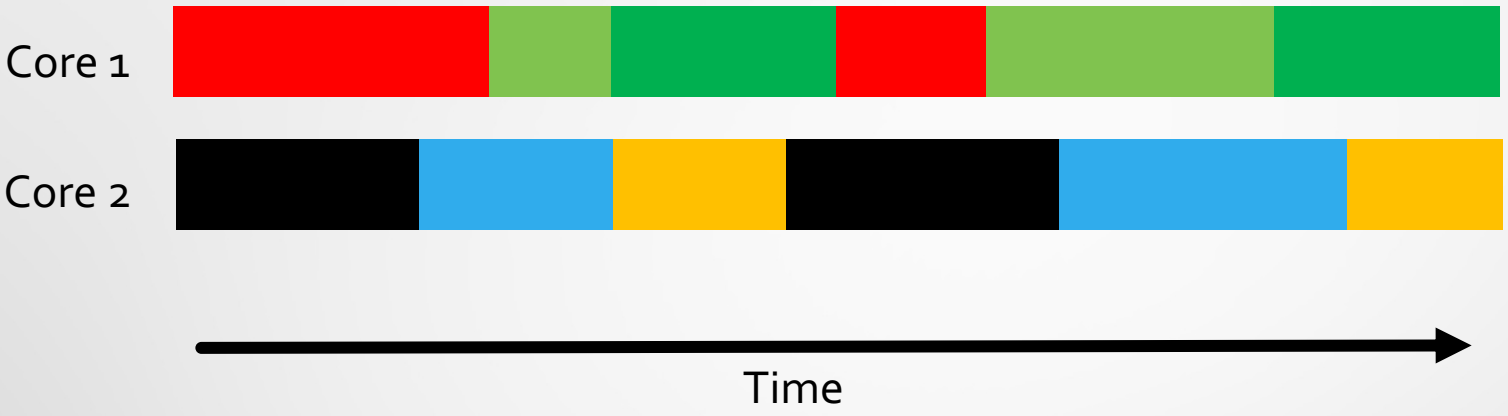
True and false concurrency

- A program has two concurrent tasks, T1 and T2
  - This means they can be executed at the same time
- On a single core machine, only T1 or T2 can utilize the CPU at a given time
  - This uses false concurrency
- On a multi-core machine, each task can execute on a different core
  - The tasks use true concurrency

Makespan of false concurrency on one core

- There are 3 tasks (3 colors)
- The OS will perform context switch between the three tasks
  - The context switching by the OS is very fast, and so this provides the illusion of parallel execution

Makespan of true concurrency on two cores



- There are 6 tasks, 3 per core
- The OS will still perform *context switch* between the three tasks *on each core*
- This is true concurrency between tasks: red and white, blue and purple, green and orange, etc.

Processes

- A **process** is a program that is running
- The OS keeps track of what processes are running in the process table
- Each process has the following characteristics:
  - A process Id
  - A username
  - A state (running, ready, blocked, etc.)
  - A program counter that points to the next instruction
  - A stack (for function calls)
  - File descriptors (open files called by your program to the system)
  - The page table, that maps the process's address space to RAM
  - The process Id of the parent process
  - and others

Processes

- Modern OSs support running multiple active processes
- Each process has three main states
  - **Ready**: I can run when the OS lets me
  - **Running**: I am currently running
  - **Blocked**: I am waiting for a resource and cannot run right now
- The OS determines how long a process in the ready state will run for
  - How the OS makes this determination will affect the performance of the system, and it is most noticeable in terms of responsiveness
  - How the OS determines what runs when is called **scheduling**

List of processes

- To find out what processes are running on your machine in UNIX/Linux
  - Type **ps**
- Let's look at some of the output
  - Default: ps (only controlling terminals)
  - ps –A (all processes)
  - The TTY means the terminal interface
    - So, if you run a process from a terminal, and then close the terminal, the process will end
    - But if you detach the process from a terminal, it won't end. This is shown by the "?" in the TTY column

Processes and memory

- Each process has its **own** memory called an **address space**
- **Virtual memory** is a memory that may be physically larger than RAM
  - The OS will swap pages between disk and RAM
  - This is so the machine doesn't crash if your process requires more memory than you have on the machine
  - Results in terrible performance!
- Virtual memory allows you to develop a program that doesn't depend on the architecture of the machine that runs the program

Process creation

- When you enter a command in the shell, you create a new process
  - The shell creates a process for you
- In the shell program there is a functionality for creating processes
- Processes can be created using the **fork system call**
  - This can be called from C

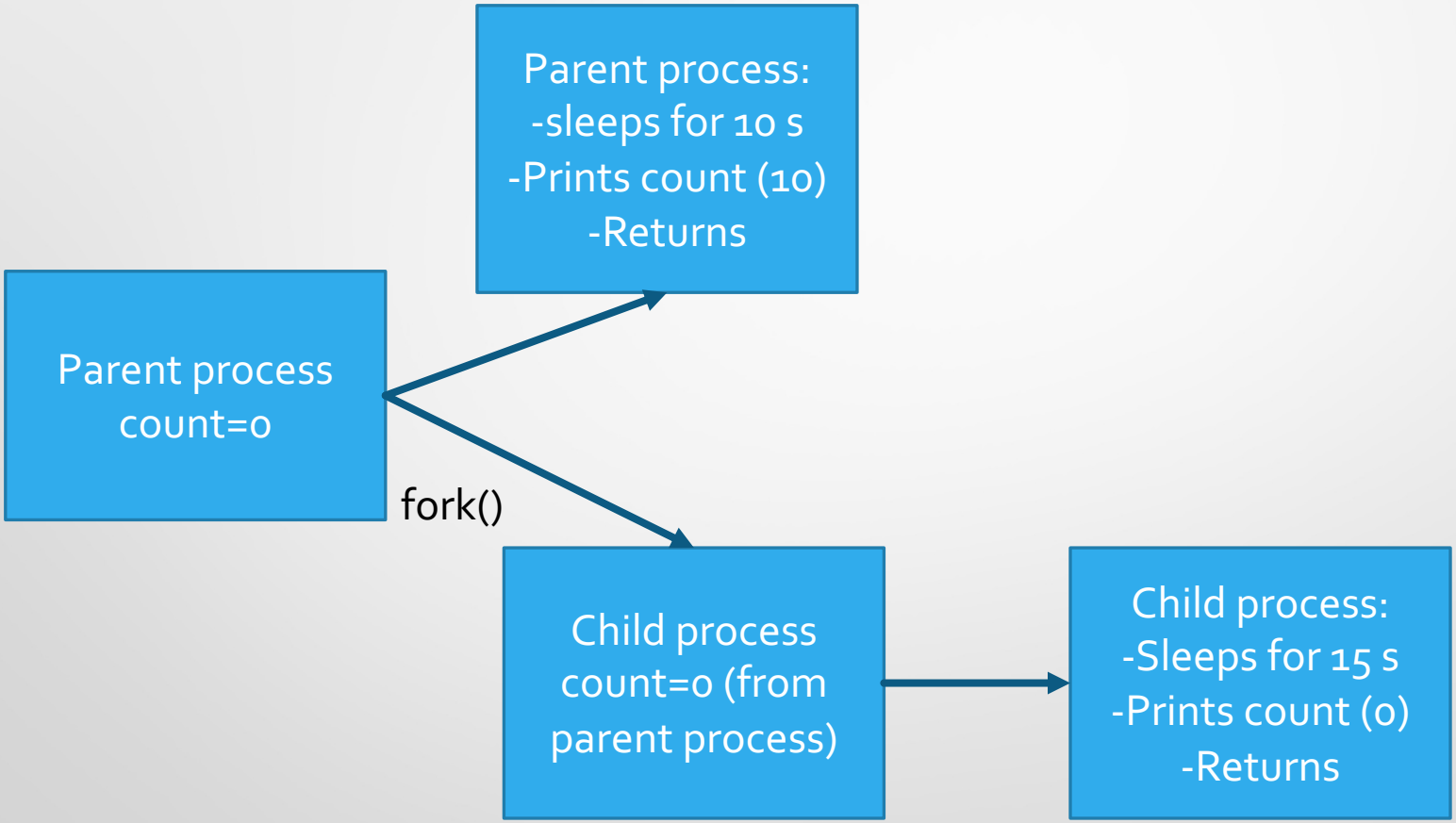
The fork() system call

- Why is it a system call?
  - Since the processes are managed by the OS, only it can create the process, such that it updates the process table
- The fork() system call creates **a copy** of the process that calls it
- After the call, both processes can follow different execution pathways
- The fork() function returns an integer value
  - Id presented in the process table
  - The integer value is the process id (PID) of the new process which is returned to the parent process
  - It returns o to the child process

The fork() system call

- The two processes run independently
- The OS determines when to run the processes
- The two processes have their own address spaces
  - Each process makes a copy of the process at the call to fork()
  - When the parent updated the variable count, the child process is oblivious to that information

The fork() system call



Processes as tasks

- We should change our way of thinking about programs as a set of tasks instead of a single monolithic program
- Can we use processes?
  - They can run concurrently because they are independent, and the OS will schedule them at the same time
  - They can use real concurrency on multi-core processors
  - They are straightforward to implement, it's one line of code
- It's not easy communicating between processes
- Concurrently solving a problem using processes is challenging

Shared memory between processes

- This sounds counter-intuitive
  - A process is supposed to have its own memory address space that it's given by the OS
- But it would be useful to make programming parallel programs easier
- There are ways to share memory between processes
- Linux has a shared memory segment abstraction
  - One process creates an area of memory that can be shared
  - It tells the other process about this memory

Threads

- Threads are used to write concurrent tasks that share memory
- Threads are like processes that share an address space
- Threads are called "lightweight processes"
  - N processes have N page tables, N addresses spaces, N PIDs
  - N threads have 1 page table, 1 address space, 1 PID
- Threads do not share the *program counter and stack*
  - N threads have N program counters
  - N threads have N stacks
- Multiple threads execute different parts of the program at the same time and therefore have their own calling sequences because they are different
- Each thread can be assigned a single task

Processes vs. threads

- Sharing memory with threads is simple: it's the most important advantage of threads
- Threads have lower overhead than processes
  - Less overhead to create a new thread
  - Less overhead to switch between threads
- Threads do not get memory protection like processes
  - You can overwrite one thread's data with another thread
  - But, if this feature is not needed, there's no sense to use threads instead of processes
- Like processes, threads require synchronization as well
  - It seems simple, but it's often challenging

Number of threads per app

- Almost all applications that have concurrent tasks use threads
- Let's find out how many threads PowerPoint uses
  - First: what process ID is PowerPoint?
    - **ps aux | grep PowerPoint**
    - PID the second column
  - Second: how many threads does it use?
    - **ps M -p <PID>**