Lecture 05: Concurrency Sources and Challenges

CS343 – Operating Systems

Branden Ghena – Spring 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), and UC Berkeley CS61C and CS162

Administrivia

- Get started on Scheduling Lab right away!
  - Getting Started lab was NOT an accurate representation of workload
  - Don’t plan on being able to get schedulers working at the last minute

- Scheduling Lab debugging tip:
  - Make your own workloads, do the math on their metrics, then test on them
    - That’s the only way to know for sure that your scheduler is right

  - We will test on many workloads that have not been provided to you

Today's Goals

- Describe where and why concurrency and parallelism are involved in computing.

- Be disappointed by performance limits on concurrency.

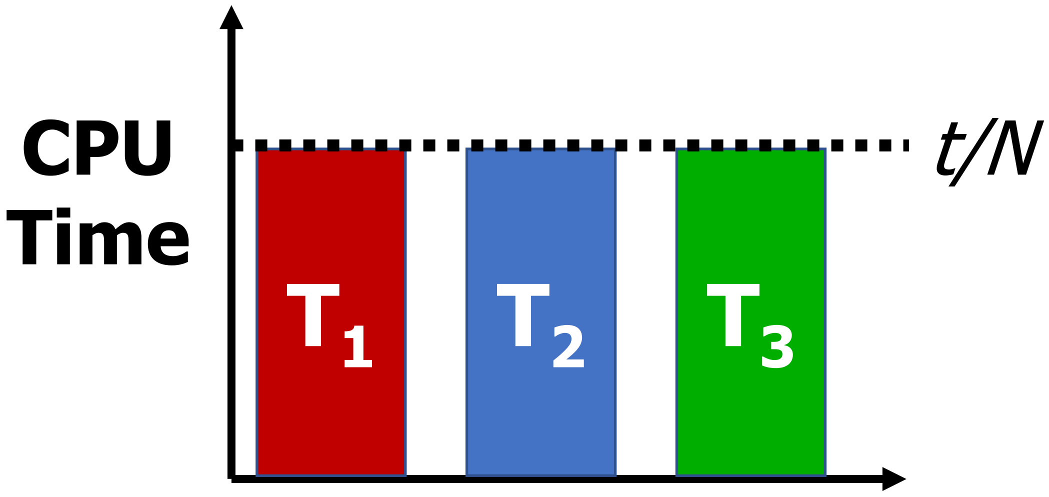
- Introduce concept of data races as a concurrency problem.

Outline

- Real Time Operating Systems
  - Earliest Deadline First scheduling
  - Rate Monotonic scheduling
- **Modern Operating Systems**
  - Linux O(1) scheduler
  - Lottery and Stride scheduling
  - Linux Completely Fair Scheduler

Proportional-share scheduling is impossible instantaneously

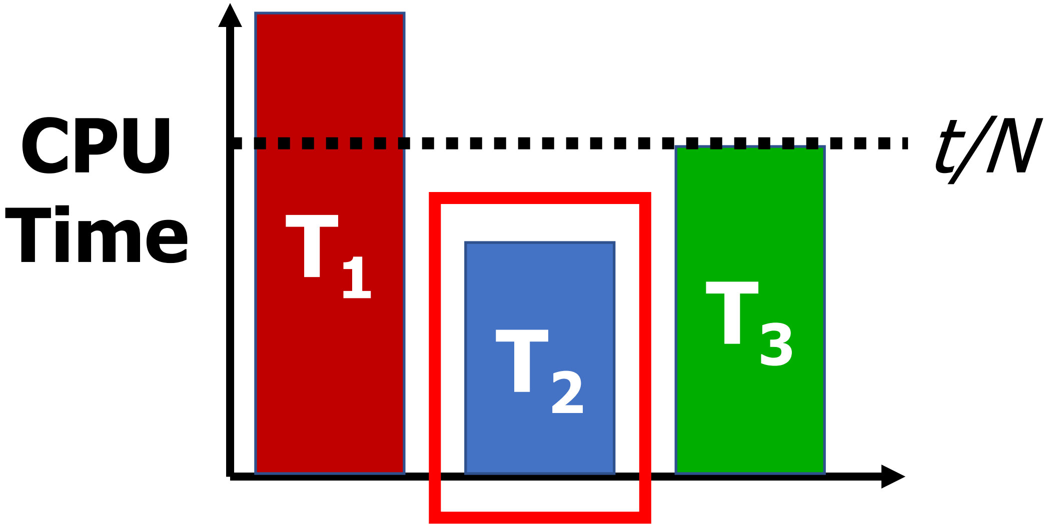
At *any* time *t*
we want to observe:



- Goal: each process gets an equal share of processor
- N threads "simultaneously" execute on 1/N^th^ of processor
- Doesn't work in the real world
  - Jobs block on I/O
  - OS needs to give out timeslices

Linux Completely Fair Scheduler (CFS) (2007-2023)

What if we make shares proportional over a longer period?



- Track processor time given to job so far
- Scheduling decision
  - Choose thread with minimum processor time to schedule
  - "Repairs" illusion of fairness
- Update processor time when the scheduling occurs again
  - Timeslice expiration is a big update
  - Blocking I/O results in maintaining small processor time

Modern scheduling is not easy

- Getting scheduling right on multicore can be difficult
  - No way to know whether a process will be more I/O or CPU bound in the future
  - Want to keep threads on the same core, but also not waste cores

- In 2016, researchers found issues in Linux scheduler implementation that lead to 13%+ slowdown in jobs
  - https://blog.acolyer.org/2016/04/26/the-linux-scheduler-a-decade-of-wasted-cores/

Modern scheduling challenges

- Fair sharing of CPU time is insufficient
  - Maximize cache usage
  - Maximize processor affinity
  - Reduce energy consumption
  - Hybrid systems with heterogeneous processing capabilities

- Particular focus: latency requirements
  - Some processes need to respond quickly to new data
  - They don’t need more processing *time*. They need the time more quickly
  - Heuristic shortcuts were added to CFS to allow some jobs to jump the queue

Earliest Eligible Virtual Deadline First (EEVDF) (2023-Present)

- Algorithm first described in a 1995 research paper
  - Run job with earliest "virtual deadline"
  - TLDR: share processor time proportionally, but schedule within that based on latency

- Still divides processor time equally between jobs, like CFS
  - Biased by priority of the job. Higher priority means larger share

- Calculate "lag" for each job
  - Measurement of how far it's behind a fair share of processor time
  - Negative lag means a job has run more than its fair share already
    - Job won't be eligible to run until lag >= 0
    - Lag increases automatically as other jobs run. So time until lag >= 0 can be calculated

- Virtual deadline for job: time until lag >= 0, plus duration it should run for
  - Now + timeslice for any jobs below fair share of processor time
  - Future + timeslice for any jobs above fair share of processor time
  - Where timeslices vary by priority of the job

https://lwn.net/Articles/925371/

Multicore scheduling

- *Affinity scheduling*: once a thread is scheduled on a CPU, OS tries to reschedule it on the same CPU
  - Cache reuse
  - Grouping threads could help or hurt…
- Soft affinity: make this a goal of the system
  - Not always achieved
- Hard affinity: allow some jobs to demand guaranteed affinity
  - Process/Thread can tell the kernel which core it should be run on

Single queue multicore scheduling

- Simplest approach to multicore scheduling
  - Keep one global queue of all jobs to be run
  - Whenever a processor becomes idle, grab the next job from the queue
  - Essentially, exactly the single-core strategy, but with more cores

- Downsides
  - Odds are that a job will run on a different core next time
    - So state for the job won’t be in the per-core cache

  - Only one core can modify the queue at a time
    - See data race issues we’ll talk about today

Multi-queue multicore scheduling

- Keep one queue of jobs for each core
  - Ensures that jobs stay on a single core when running

- Need to balance work among cores somehow
  - Might have one core with many jobs and another that's idle
  - Work stealing: when a processor is idle, look at other processor's queues and take a job from them
    - Still undermines affinity goals, but hopefully worth it?
    - Complicated to scale out to many cores

Summary on schedulers

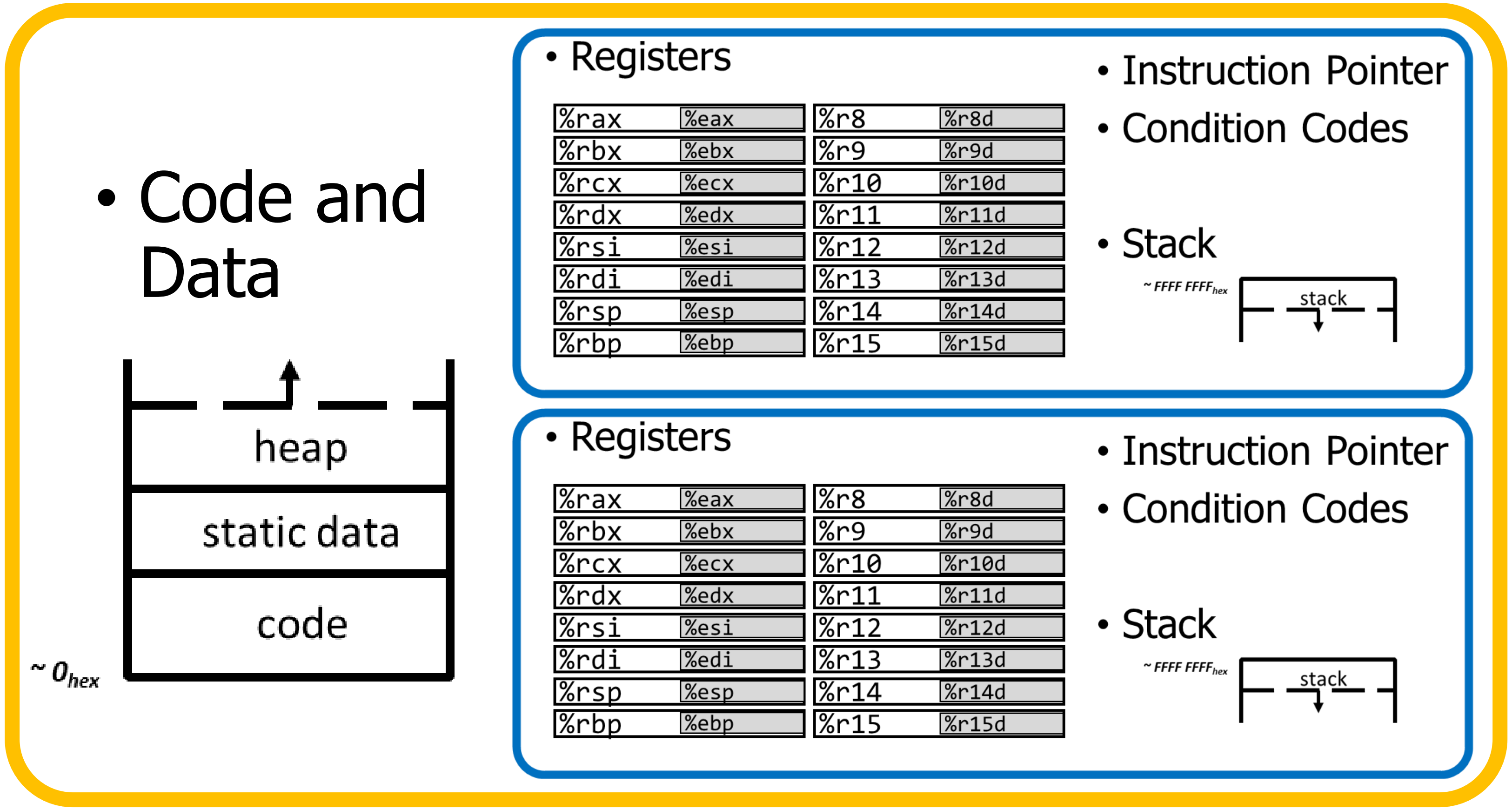
If You care About:	Then Choose:
CPU Throughput	First-In-First-Out
Average Turnaround Time	Shortest Remaining Processing Time
Average Response Time	Round Robin
Favoring Important Tasks	Priority
Fair CPU Time Usage	Linux CFS or EEVDF
Meeting Deadlines	EDF or RMS

Outline

- **Threads Review**
- Need for Parallelism
- Processor Concurrency
- Concurrency Challenges
  - Amdahl's Law
  - Data Races

Processes and threads

- A process could have multiple threads
  - Each with its own registers and stack

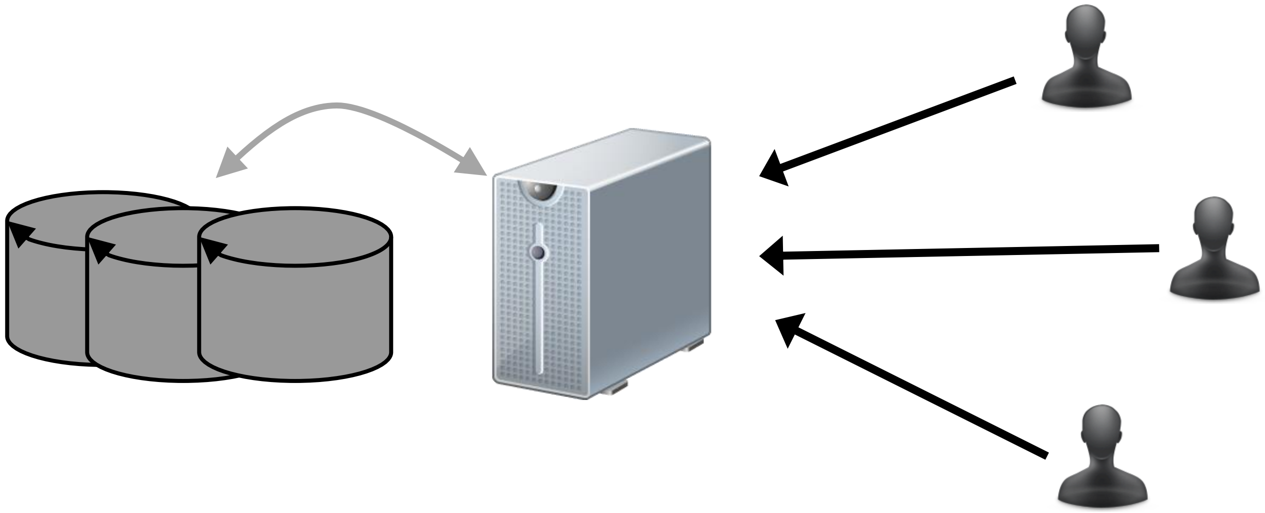


Threads have separate:
- Instruction Pointer
- Registers
- Stack Memory
- Condition Codes

Threads share:
- Code
- Global variables

Thread use case: web server

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



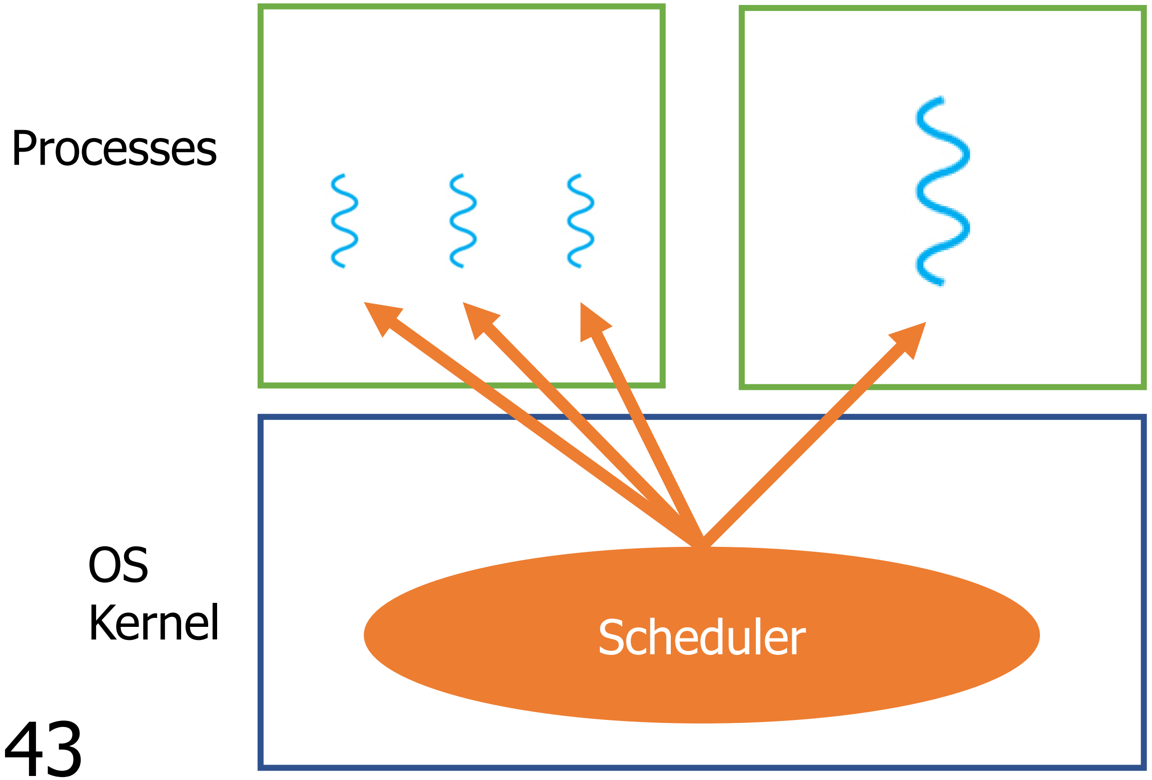
Models for thread libraries: **Kernel Threads**

- Thread scheduling is implemented by the operating system
  - OS manages the threads within each process

- Upsides
  - Other threads can continue while one blocks on I/O
  - No additional scheduler

- Downsides
  - Higher overhead

- This is what we'll focus on in CS343



POSIX Threads Library: pthreads

- https://man7.org/linux/man-pages/man7/pthreads.7.html

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

- thread is created executing *start_routine* with *arg* as its sole argument.
- return is implicit call to pthread_exit

```
void pthread_exit(void *value_ptr);
```

- terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- suspends execution of the calling thread until the target *thread* terminates.
- On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

Pthread system call example

- What happens when pthread_create() is called in a process?

```
Library:
  int pthread_create(…) {
    Do some work like a normal function
    Put syscall number into register
    Put args into registers
    Special trap instruction
```

← `clone` (56) syscall on Linux

```
Kernel:
    Get args from regs
    Do the work to spawn the new thread
    Store return value in %eax
```

```
    Get return values from regs
    Do some more work like a normal function
  };
```

Threads versus Processes

Threads

- `pthread_create()`
  - Creates a thread
  - ***Shares*** all memory with all threads of the process.
  - Scheduled independently of parent
- `pthread_join()`
  - Waits for a particular thread to finish
- Can communicate by reading/writing (shared) global variables.

Processes

- `fork()`
  - Creates a single-threaded process
  - ***Copies*** all memory from parent
    - Can be quick using copy-on-write
  - Scheduled independently of parent
- `waitpid()`
  - Waits for a particular child process to finish
- Can communicate by setting up shared memory, pipes, reading/writing files, or using sockets (network).

Threads Example

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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

Threads Example

- Reads N from process arguments
- Creates N threads
- Each one prints a number, then increments it, then exits
- Main process waits for all of the threads to finish

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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

Threads Example

```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread  */
}
```

Threads Example

```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

- Left: Every thread has its own stack

- Right: Every thread shares global memory

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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread  */
}
```

Break + Check your understanding

```
[(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

1. How many threads are in this program?
2. Does the main thread join with the threads in the same order that they were created?
3. Do the threads exit in the same order they were created?
4. If we run the program again, could the result change?

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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread  */
}
```

Break +
Check your understanding

```
[(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

1. How many threads are in this program? **Five**
2. Does the main thread join with the threads in the same order that they were created? **Yes**
3. Do the threads exit in the same order they were created? **Maybe??**
4. If we run the program again, could the result change? **Possibly!**

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

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread  */
}
```

Outline

- Threads
- **Need for Parallelism**
- Processor Concurrency
- Concurrency Challenges
  - Amdahl’s Law
  - Data Races

It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years.

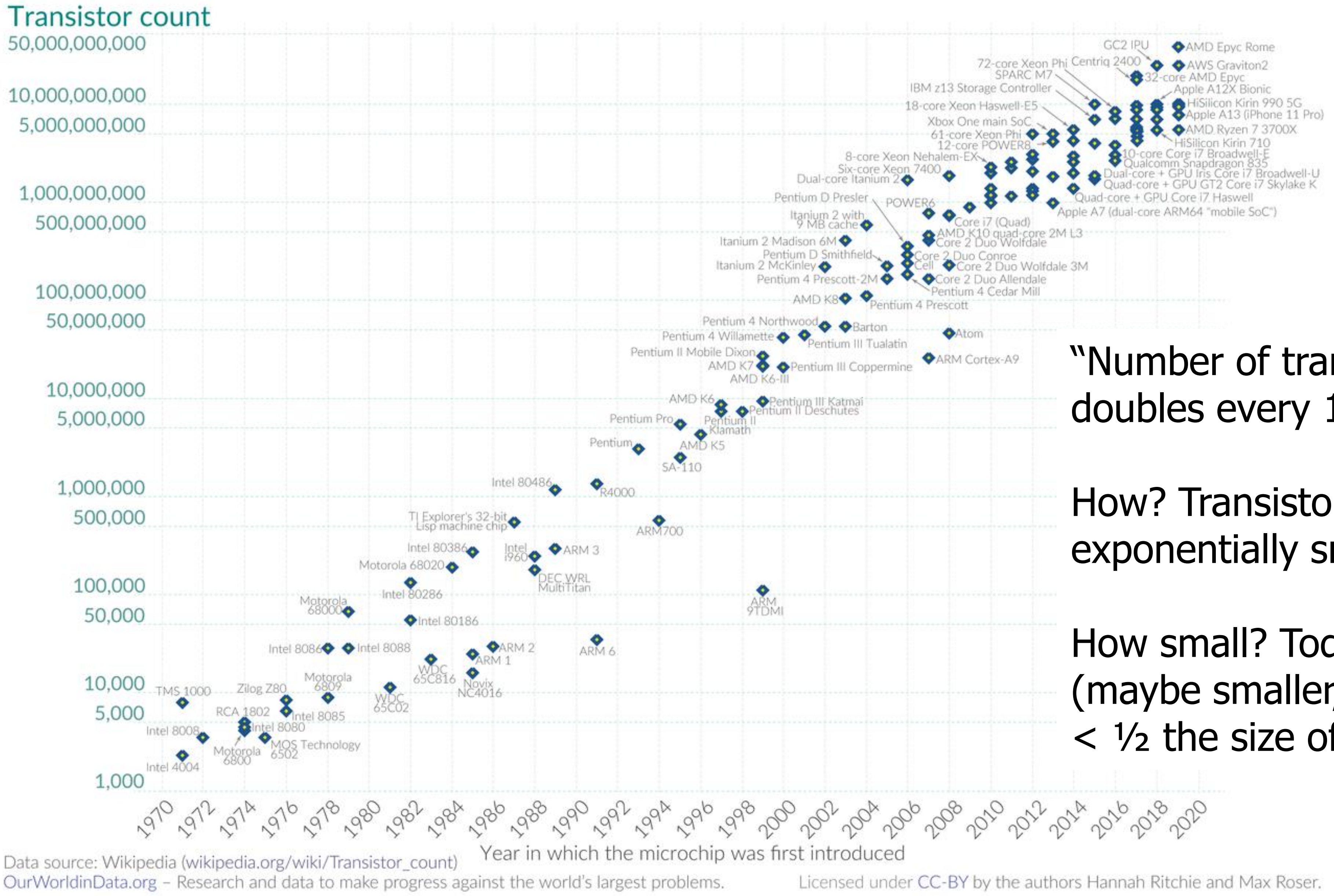
What do you do?

It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years.

What do you do? **Take a vacation**

Moore’s Law – CPU transistors counts

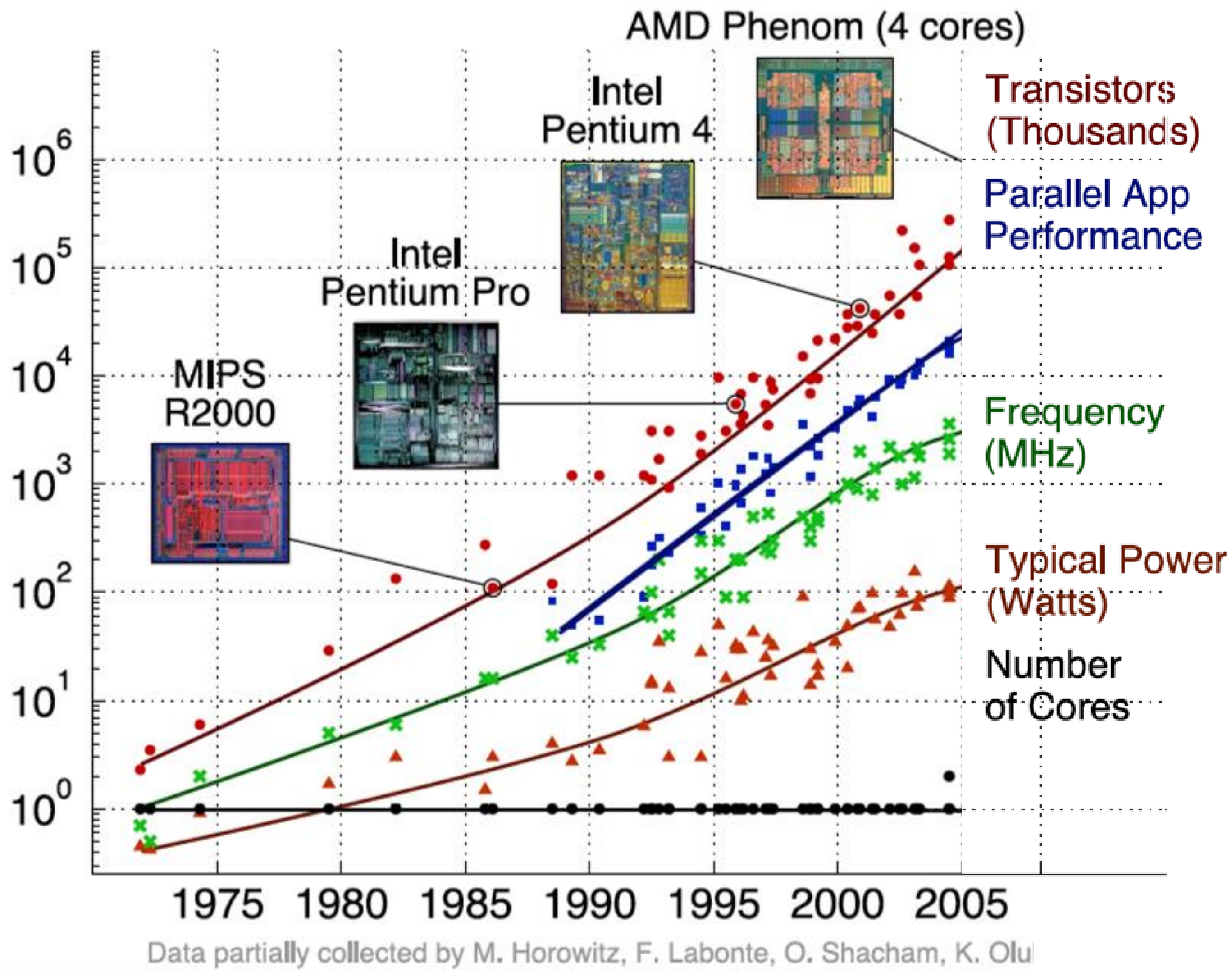


“Number of transistors in a chip doubles every 18 months”

How? Transistors are getting exponentially smaller!

How small? Today: <7nm!
(maybe smaller, kind of complicated)
< ½ the size of most viruses!

Processors kept getting faster too



Power is a major limiting factor on speed

- We could make processors go very fast
  - But doing so uses more and more power
- More power means more heat generated
  - And chips typically work up to around 100°C
  - Hotter than that and things stop working
- We add heat sinks and fans and water coolers to keep chips cool
  - But it's hard to remove heat quickly enough from chips
- So, power consumption ends up limiting processor speed

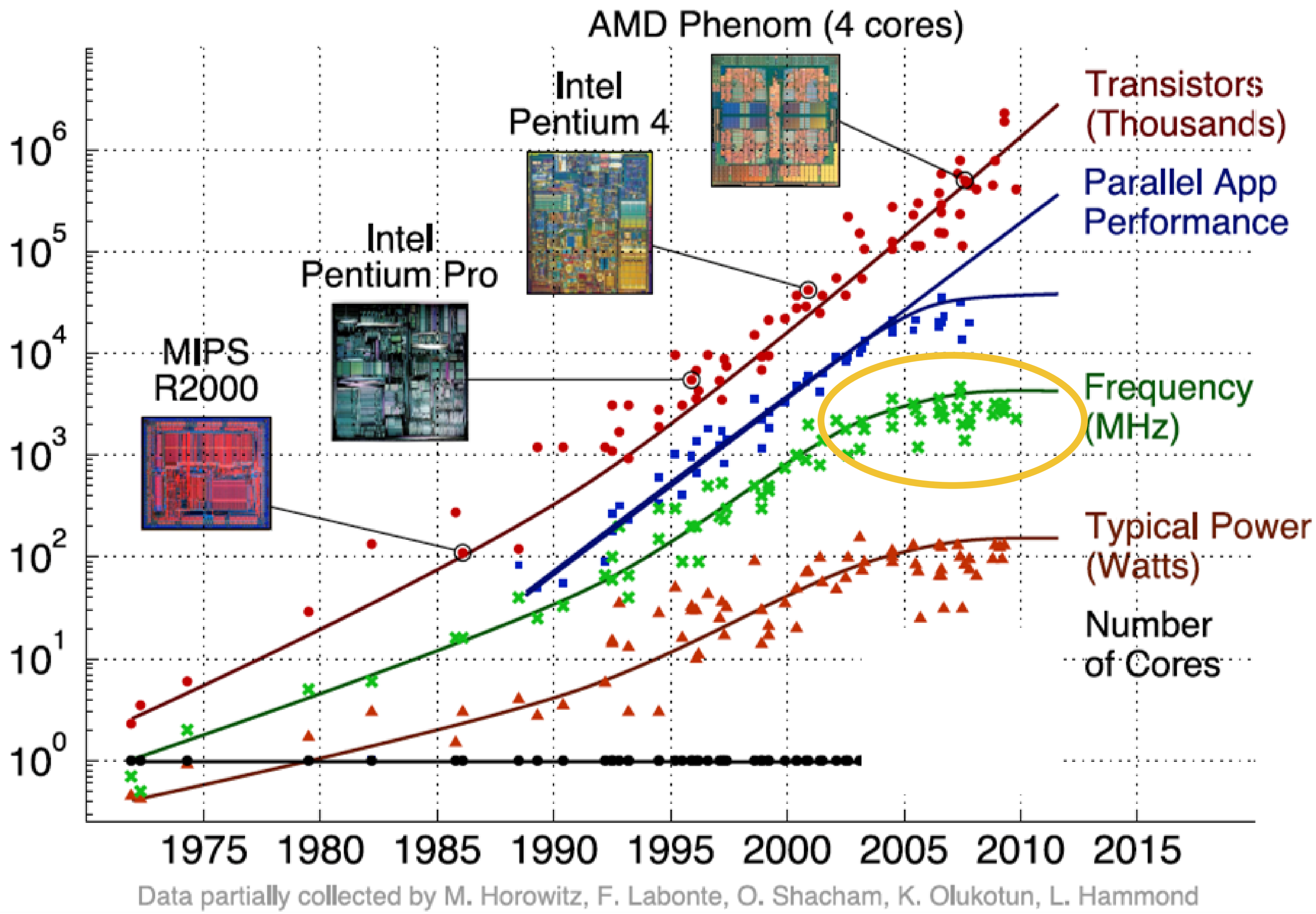
Denard Scaling

- Moore's Law corollary: Denar Scaling
  - As transistors get smaller, the power density stays the same
  - Which is to say that the power-per-transistor decreases!

- Making the processor clock speed faster uses more power
  - But the two balance out for roughly net even power
  - So not only do we get *more* transistors, but chip speed can be *faster* too

- From our Excel example:
  - In two years, new hardware would run the existing software twice as fast

Then they stopped getting faster



~2006: Leakage current becomes significant

Now smaller transistors doesn't mean lower power

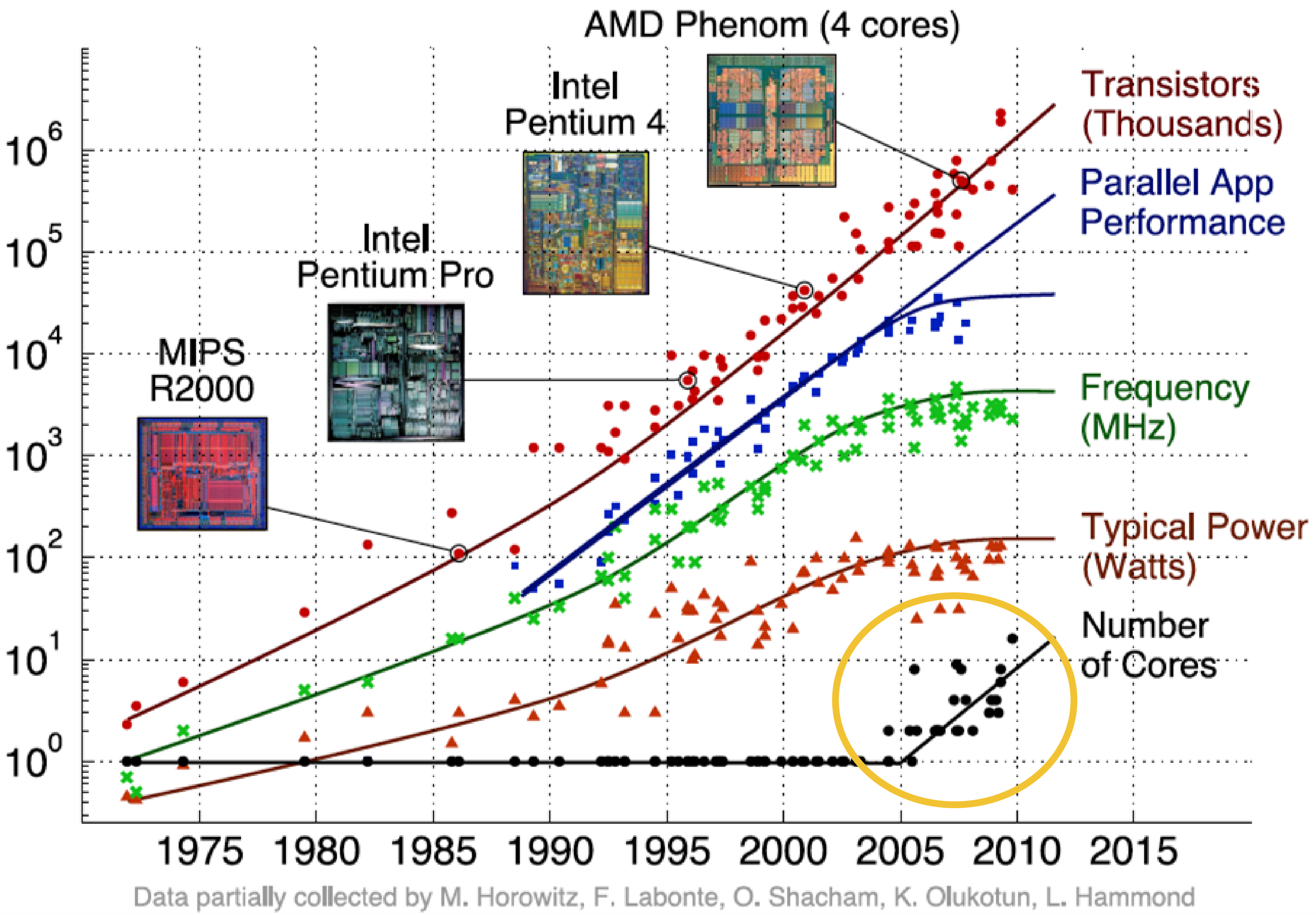
So… now what?

In summary:

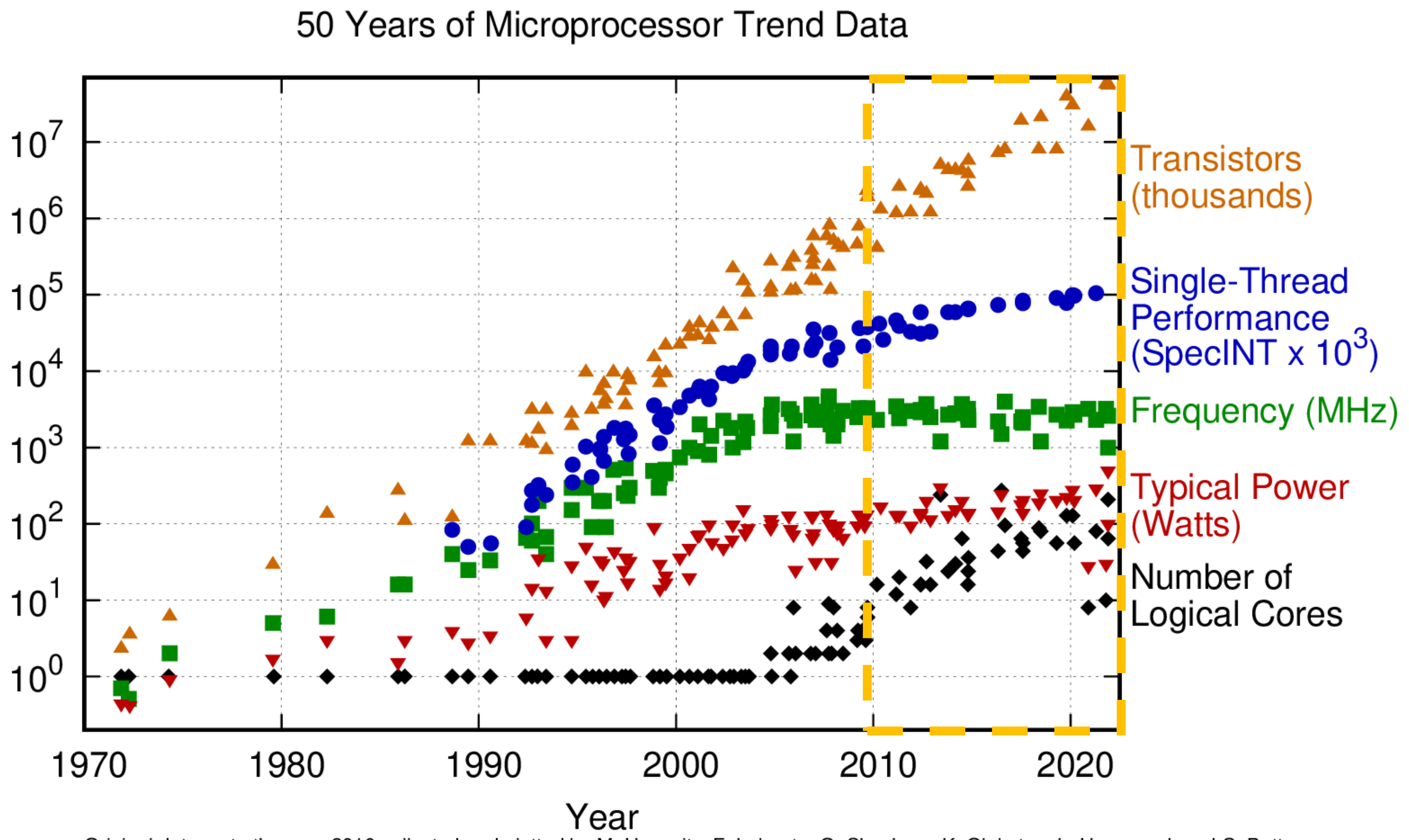
- Making transistors smaller doesn't make them lower power,
- so if we were to make them faster, they would take more power,
- which will eventually lead to our processors melting…
- and because of that, *we can't reliably make performance better by waiting for clock speeds to increase.*

How do we continue to get better computation performance?

Exploit parallelism!



Update: 2010-2021



Still growing!

Small improvement

Stable

Stable

Still growing for servers

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Key question: how do we use all these cores?

Break + Parallelism Analogy

- I want to peel 100 potatoes as fast as possible:
  - I can learn to peel potatoes faster

  OR

  - I can get 99 friends to help me
- Whenever one result doesn't depend on another, doing the task in parallel can be a big win!

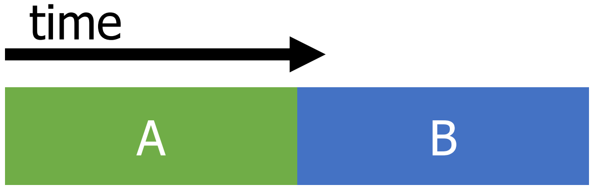
Outline

- Threads
- Need for Parallelism
- **Processor Concurrency**
- Concurrency Challenges
    - Amdahl's Law
    - Data Races

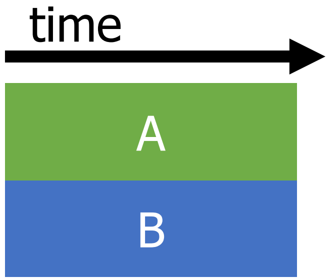
Parallelism versus Concurrency

Two processes A and B

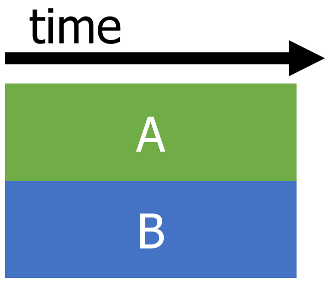
Serial execution



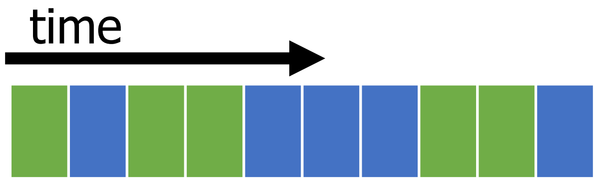
Parallel execution



Concurrent execution



OR

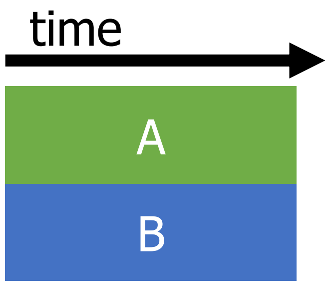


Parallelism versus Concurrency

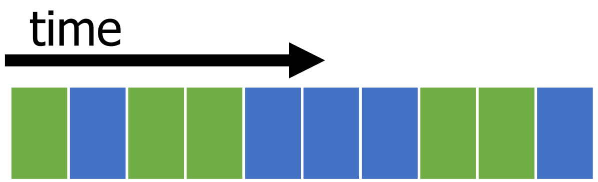
- Parallelism
  - Two things happen strictly simultaneously

- Concurrency
  - More general term
  - Two things happen in the same time window
    - Could be simultaneous, could be interleaved

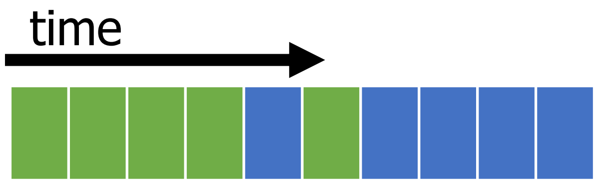
  - Concurrent execution occurs whenever two processes are both active



OR



OR



Hardware sources of concurrency

- Instruction-level parallelism

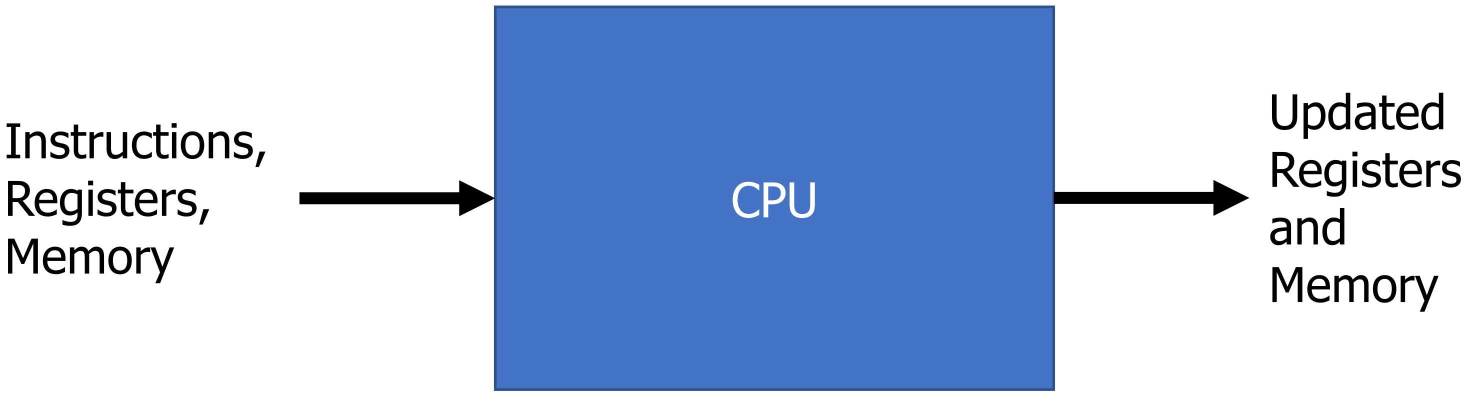
- Task parallelism
  - Multiple processes
  - Multiple threads

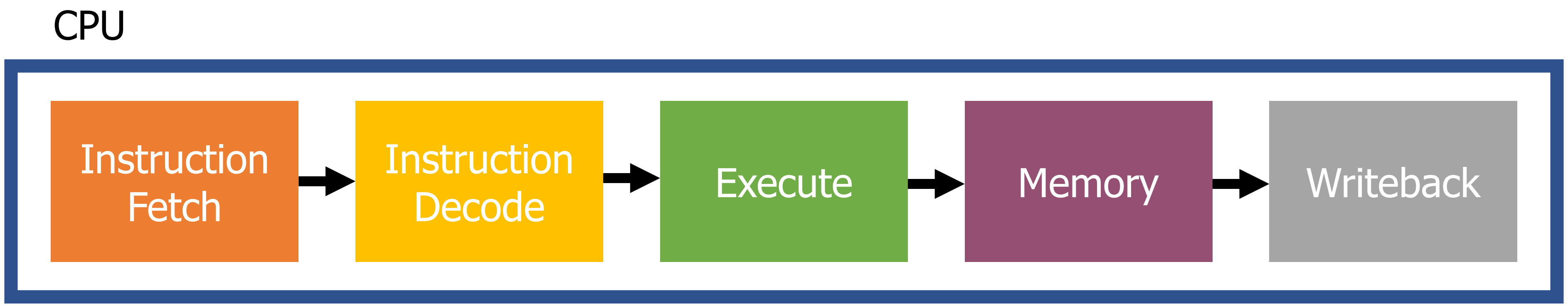
Hardware sources of concurrency

- **Instruction-level parallelism**

- Task parallelism
  - Multiple processes
  - Multiple threads

Model of a processor





But instructions don’t always have to be executed in order

```
movq  (%rdi), %rax
movq  (%rsi), %rdx
movq  %rdx, (%rdi)
movq  %rax, (%rsi)
addq  %rcx, %rbx
```

← Doesn’t have to go after the movq instructions because it uses different registers

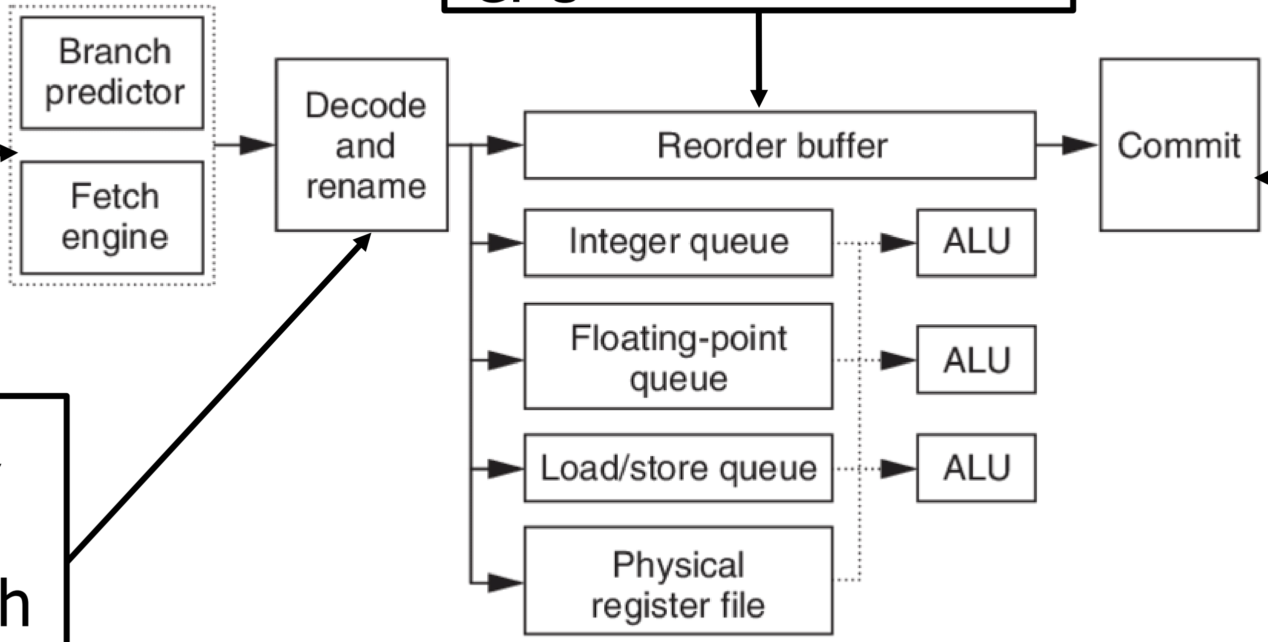
We can apply the multiprogramming approach of executing this `addq` while the `movq` is waiting on memory.

Out-of-order processors

Fetch many instructions at once!

Read register file, handle data dependencies with register renaming

Reorder instructions to make best use of CPU



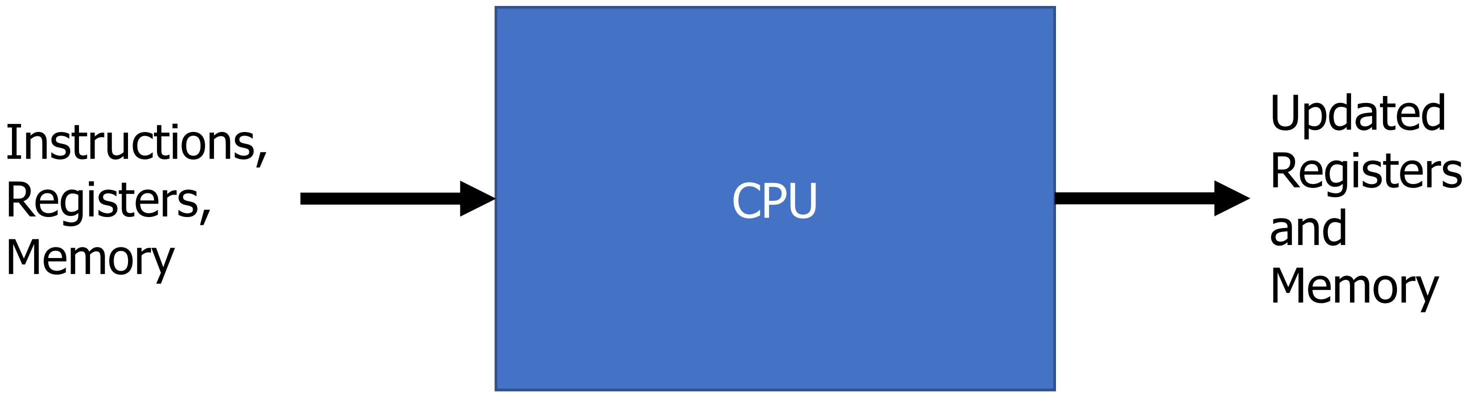
Commit, or "write back" data to memory and regfile in the order the programmer expects

Generally: looks for *independent* instructions it can execute early

Out-of-order processors obey normal execution results

- Initial thoughts on out-of-order execution
  - 😱
  - The processor could be executing my program in order it feels like?!!
  - How do I possibly reason about anything?

- Answer: the processor promises to have the same results as if things were done in the normal order.



Multiple threads might rely on memory ordering

- The processor can't account for multiple threads though
- If memory results are shared by two threads, the processor might mess something up for you.

```
f = 0;
x = 0;
// split into threads
```

Thread 1

```
while (f == 0);
printf("%d\n", x);
```

Thread 2

```
x = 42;
f = 1;
```

- What will Thread 1 print?

Multiple threads might rely on memory ordering

- The processor can’t account for multiple threads though
- If memory results are shared by two threads, the processor might mess something up for you.

```
f = 0;
x = 0;
// split into threads
```

Thread 1

```
while (f == 0);
printf(“%d\n”, x);
```

Thread 2

```
x = 42;
f = 1;
```

- What will Thread 1 print? **Could be 42. Could be 0.**

This can be addressed with memory barriers

Hardware sources of concurrency

- Instruction-level parallelism

- **Task parallelism**
  - **Multiple processes**
  - Multiple threads

Task parallelism use case

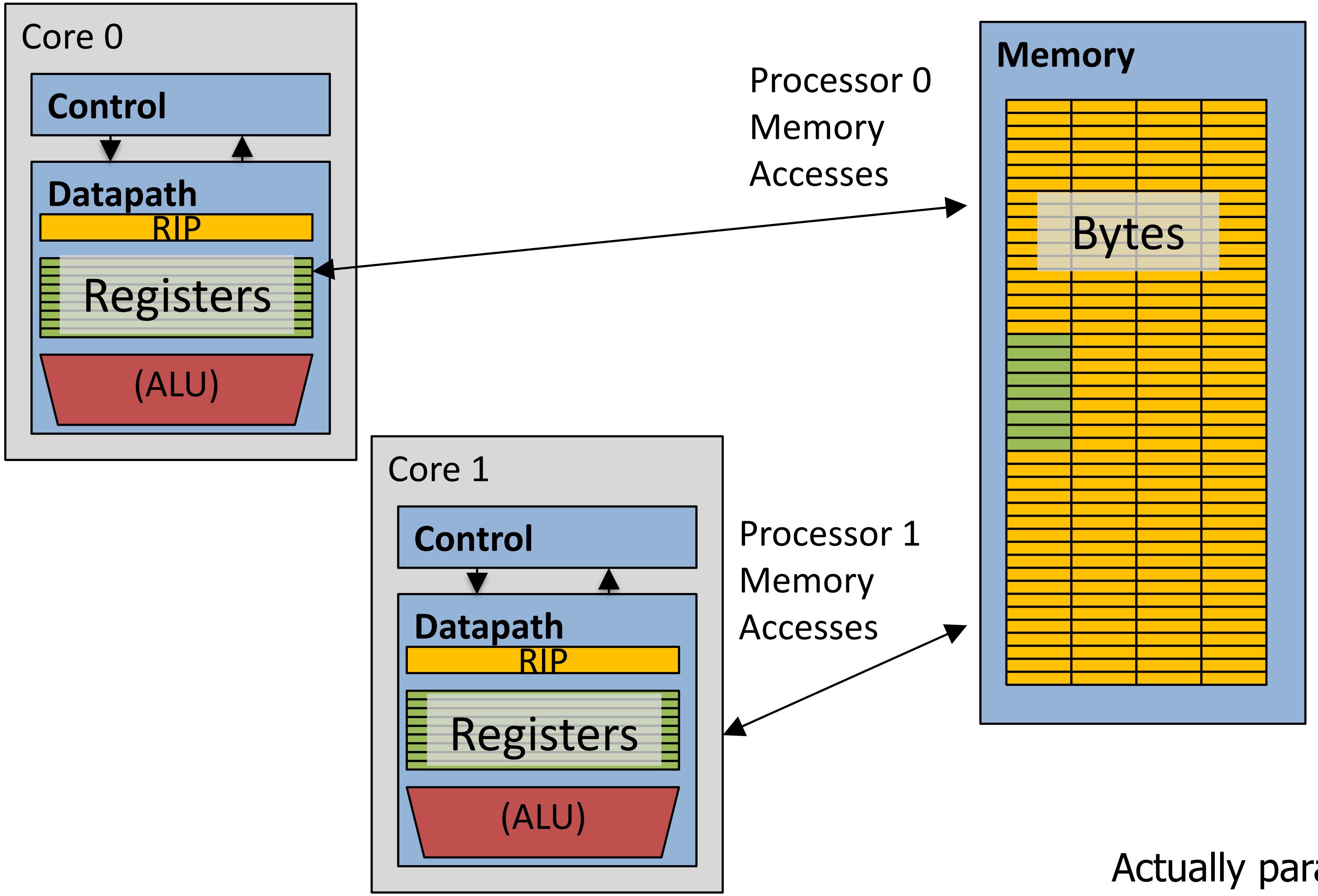
Run Chrome and Spotify simultaneously

- Each are separate programs
- Each has a different memory space
- Each can run on a separate core

Don’t even need to communicate...

Note: OS can fake this by interleaving processes,
but hardware can make it actually simultaneous

Multicore Systems (in pictures)



Actually parallel!

Multicore Systems (in words)

- A computer system with at least 2 processor cores
  - Each core has its own registers
  - Each core executes independent instruction streams
  - Cores share the same system memory
    - But usually use different parts of it
  - Communication possible through memory accesses

- Deliver high throughput for independent jobs via task-level parallelism

Hardware sources of concurrency

- Instruction-level parallelism

- **Task parallelism**
  - Multiple processes
  - **Multiple threads**

Multithreading processors

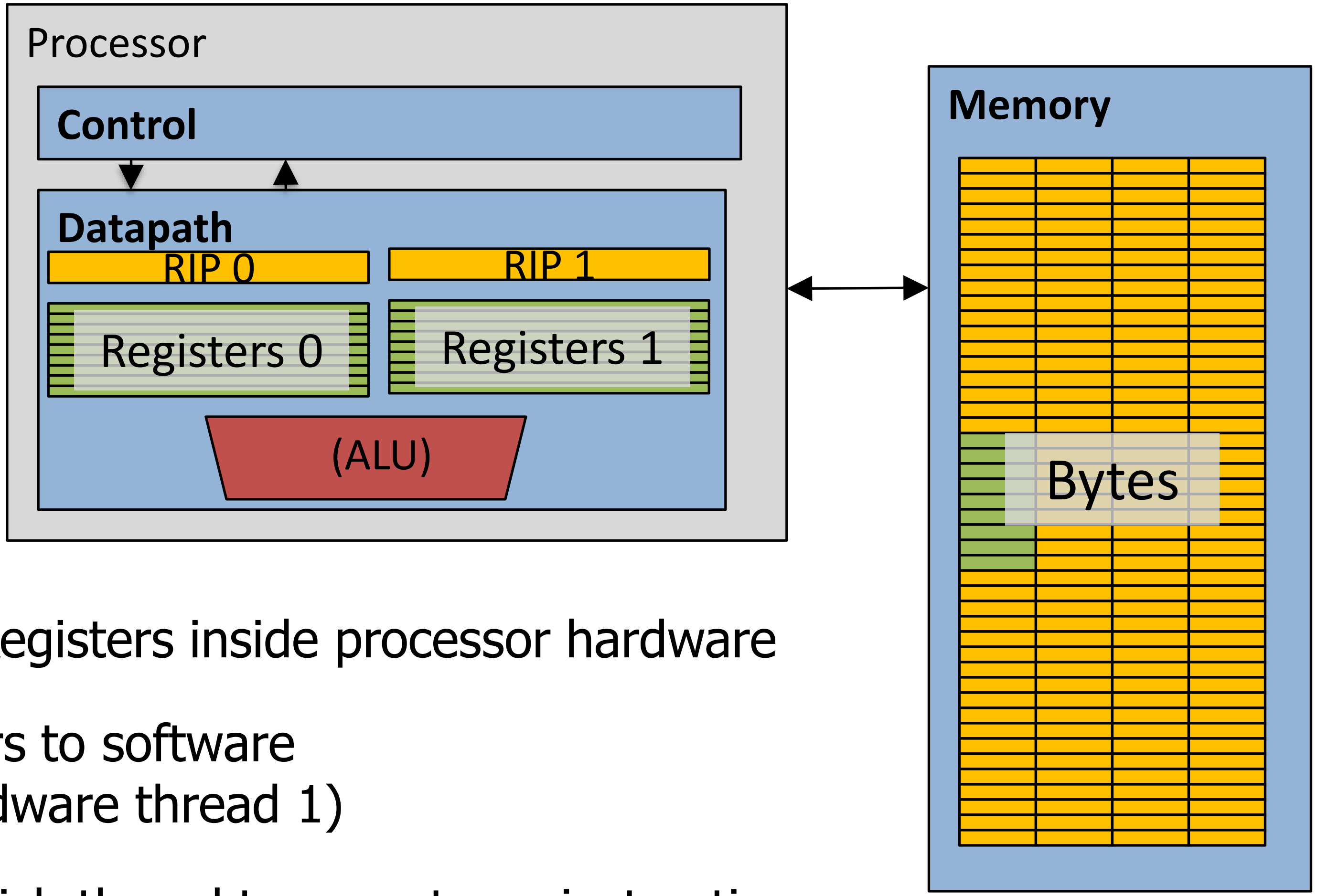
Basic idea: Processor resources are expensive and should not be left idle

Long memory latency to memory on cache miss?

- Hardware switches threads to bring in other useful work while waiting for cache miss
- Cost of thread context switch must be much less than cache miss latency

- Switching threads is less expensive than processes because they share memory
  - Cache is still valid
  - Page Table for virtual memory doesn't have to change

Multithreading processor

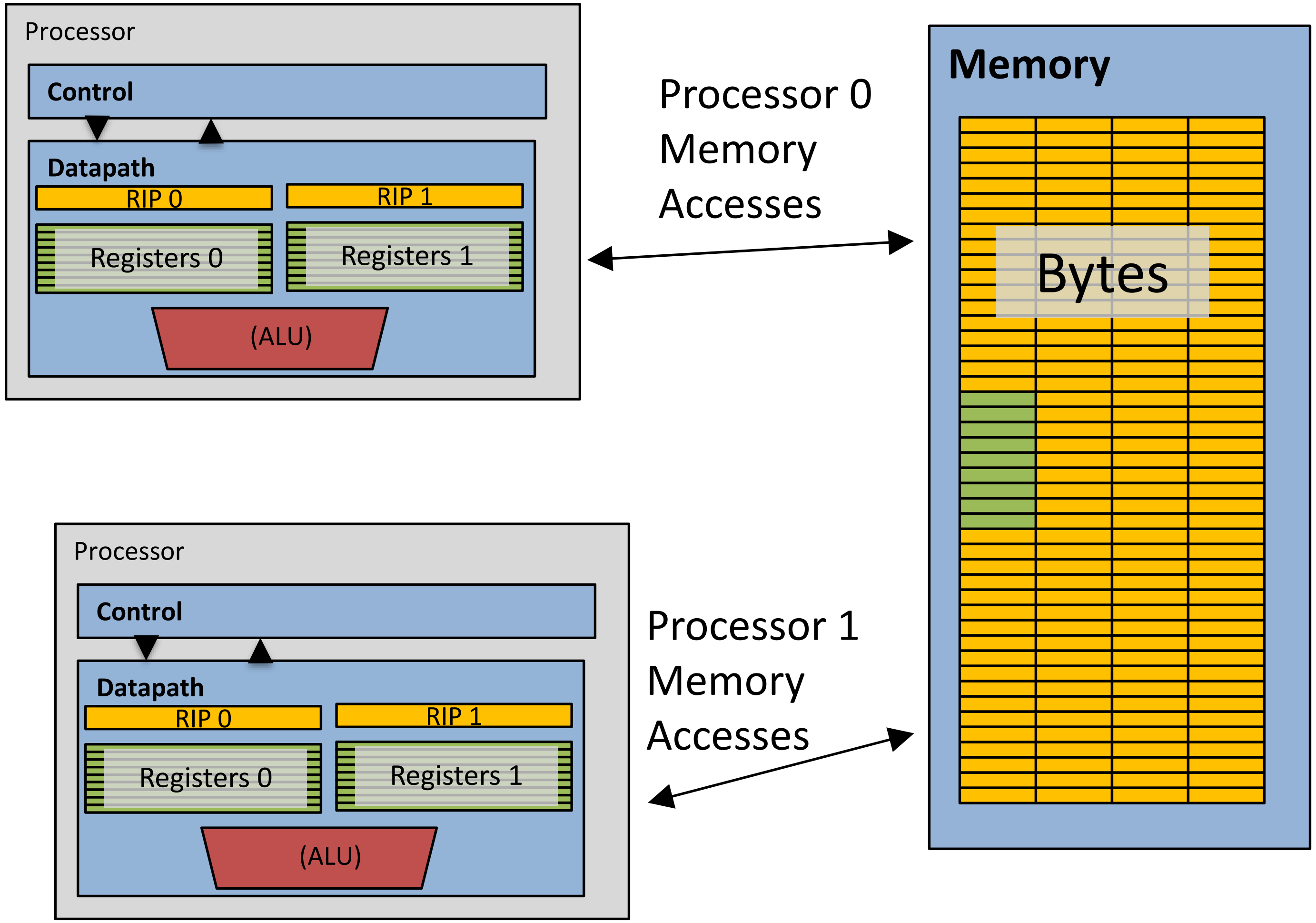


- Two copies of RIP and Registers inside processor hardware
- Looks like two processors to software
  (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next (concurrent, but NOT parallel)

Multithreading versus Multicore

- Multithreading => Better utilization
  - ≈5% more hardware for ≈1.3x better performance?
  - Gets to share ALUs, caches, memory controller
- Multicore => Duplicate cores
  - ≈50% more hardware for ≈2x better performance?
  - Share some caches (L2 cache, L3 cache), memory controller
- Modern processors might do both!
  - Multiple cores with multiple threads per core
  - Not all do though, some focus on better single-thread performance

Multithreading, multicore processors



- Combine capabilities of both designs
- Run two processes each with two threads
- Or run one process with four threads

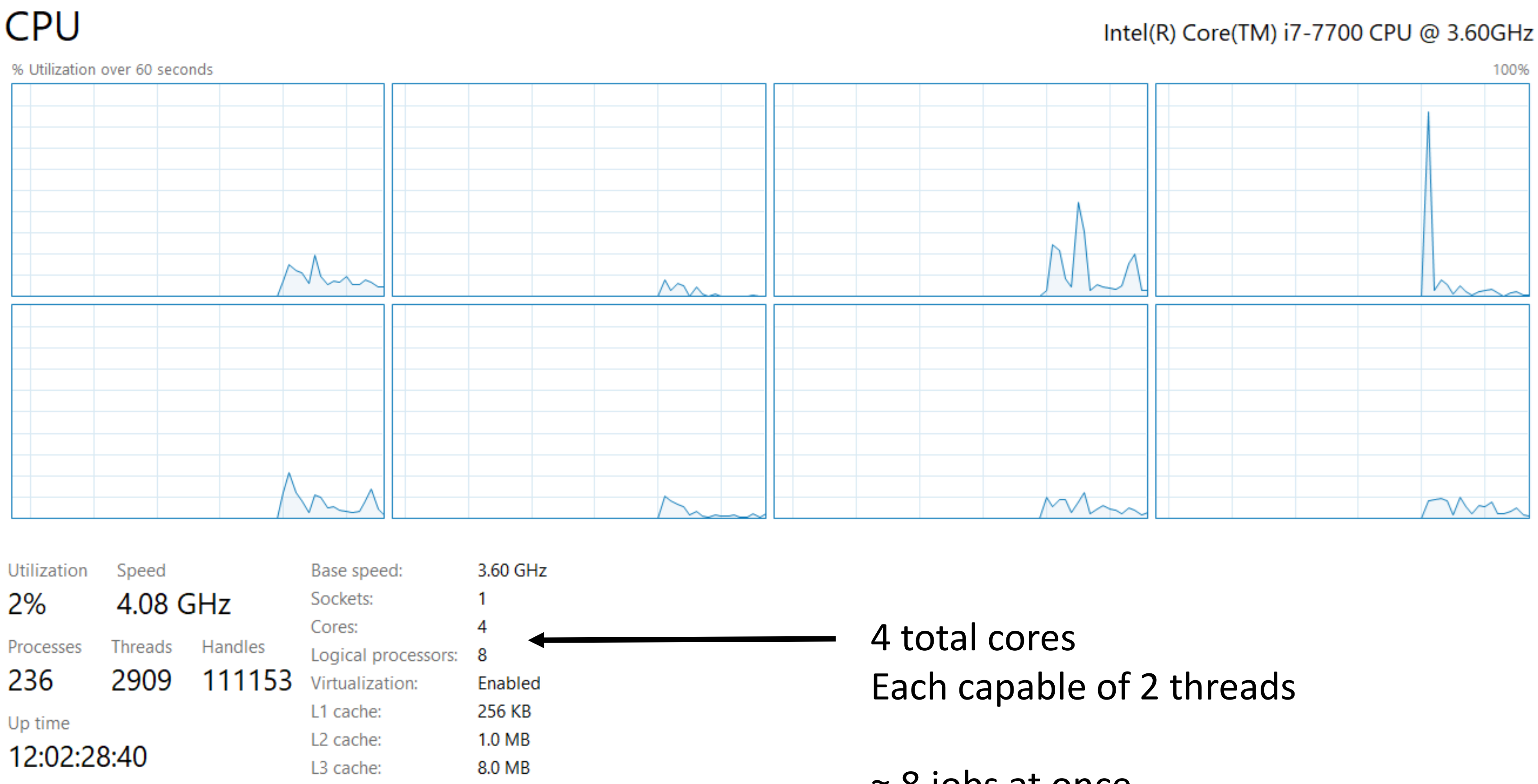
Clearing up vocabulary

- Core: computation unit within the CPU
  - ALU, Registers, etc.
  - Capable of running one or more threads

- CPU (processor): the chip that goes in your computer
  - Contains one or more cores
  - Computers could have multiple CPU chips as well

- Sometimes people equate processors and cores, which is confusing
  - I’ll definitely do it by mistake at some point if I haven’t already. Sorry!

My desktop computer



4 total cores
Each capable of 2 threads

≈ 8 jobs at once

Raspberry Pi 4

Quad core processor

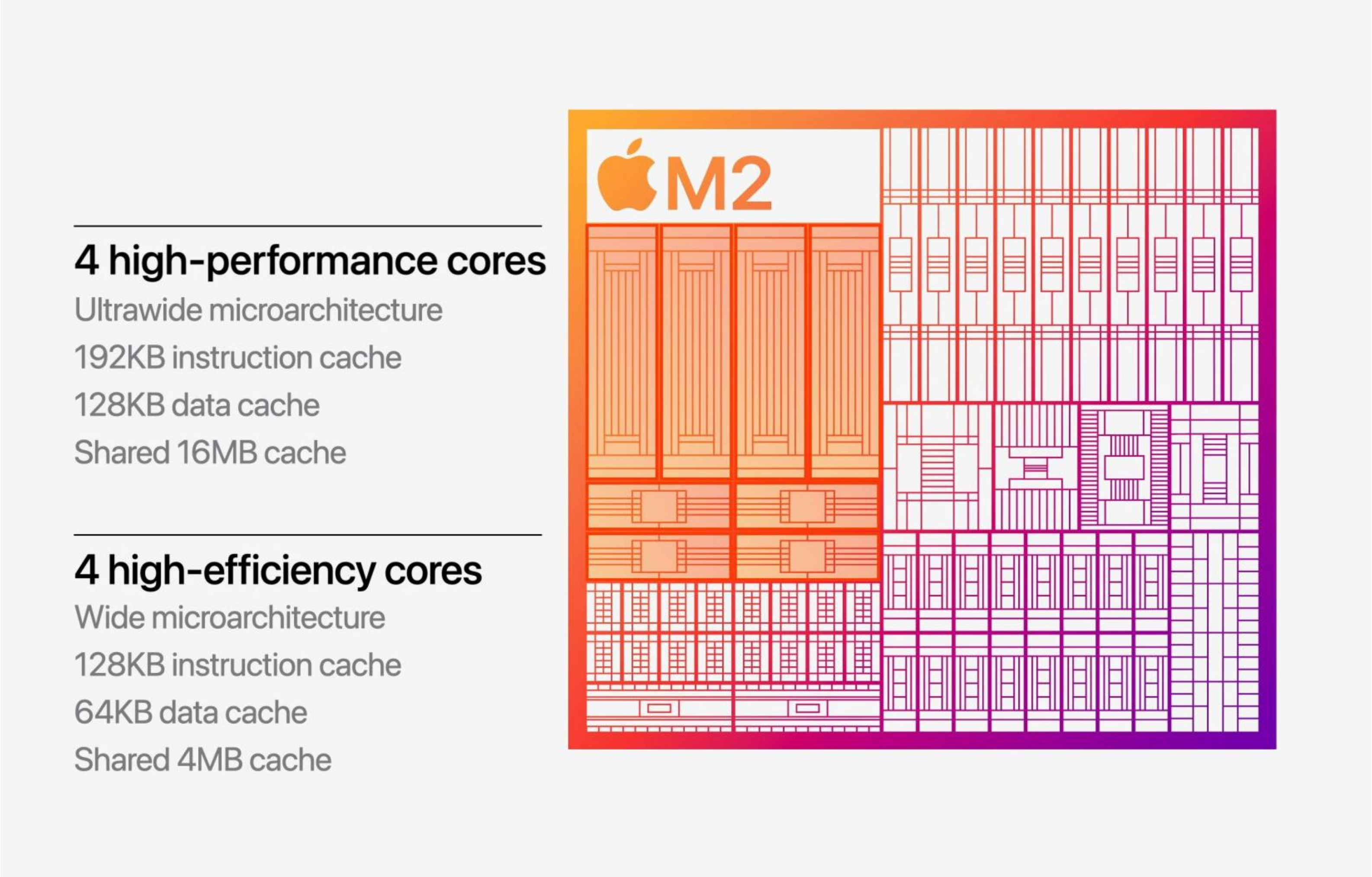
- One thread per core
- 3-way superscalar pipeline
- L1 Cache
  - 32 KiB 2-way set associative data cache
  - 48 KiB 3-way set associative instruction cache
  - Per core
- L2 Cache
  - 512 KiB to 4 MiB (shared)
- RAM 1-4 GB

$35

Literally all computers are doing parallelism these days

Other modern multicore designs

- Heterogeneous multicore
  - Not all cores are necessarily identical
- Enables scheduler to make complicated choices of performance or energy savings
  - At the cost of a complicated scheduler…



Back up to the OS perspective

- Modern operating systems must manage concurrency
  - Both parallel operation and interleaving operations
- Concurrency is valuable
  - Performance gains are the reason

Break + Real-world Connection

- How many cores/threads does your processor support?
  - Windows: Task Manager -> Performance -> CPU
  - MacOS: About this Mac -> System Report -> Hardware
    - Apple ARM M processors only do 1 thread per core
  - Linux: In terminal: lscpu
  - Android/iOS: You’ll need to google it

Outline

- Threads
- Need for Parallelism
- Processor Concurrency
- **Concurrency Challenges**
  - **Amdahl's Law**
  - Data Races

Challenges to concurrency

Concurrency is great! We can do so many things!!

But what’s the downside…?

1. How much speedup can we get from it?
2. How hard is it to write parallel programs?

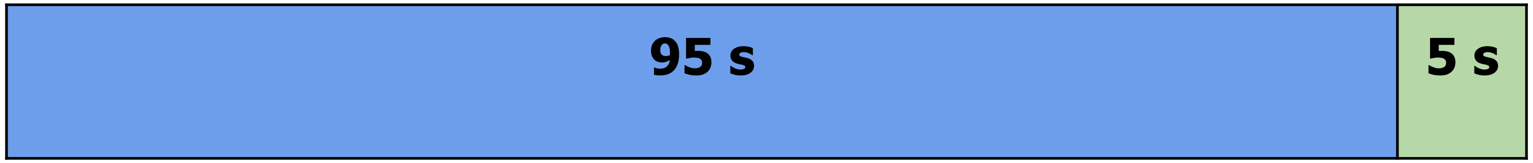
Challenges to concurrency

Concurrency is great! We can do so many things!!

But what's the downside…?

1. **How much speedup can we get from it?**
2. How hard is it to write parallel programs?

Speedup Example



Imagine a program that takes 100 seconds to run

- 95 seconds in the blue part
- 5 seconds in the green part

Speedup from improvements

$$\text{Speedup with Improvement} = \frac{\text{Execution time without improvement}}{\text{Execution time with improvement}}$$

5 s -> 2.5 s: Speedup = 100/97.5 = 1.026

5 s -> 1 s: Speedup = 100/96 = 1.042

5 s -> 0.001s: Speedup = 100/95.001 = 1.053

The impact of a performance improvement is relative to the importance of the part being improved!

Amdahl's Law

Equivalent to prior equation

$$\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Not improved part

Improved part

F = Fraction of execution time speed up

S = Scale of improvement

Example: 2x improvement to 25% of the program

$$\frac{1}{0.75 + \frac{0.25}{2}} = \frac{1}{0.75 + 0.125} = 1.14$$

Parallel speedup example

$$\text{Speedup with improvement} = \frac{1}{(1-F)+(F/S)}$$

- Consider an improvement which runs 20 times faster but is only usable 15% of the time

$$\text{Speedup with improvement} = \frac{1}{(0.85)+(0.15/20)} = 1.166$$

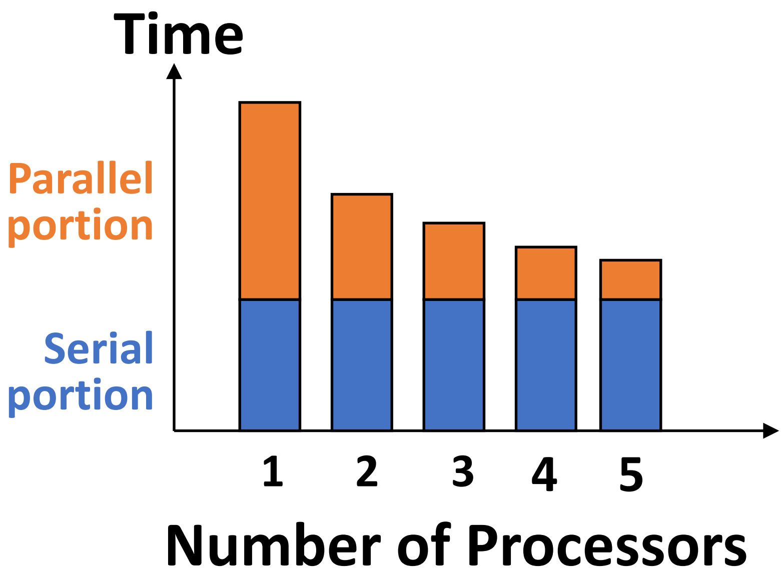
Nowhere near 20x speedup!

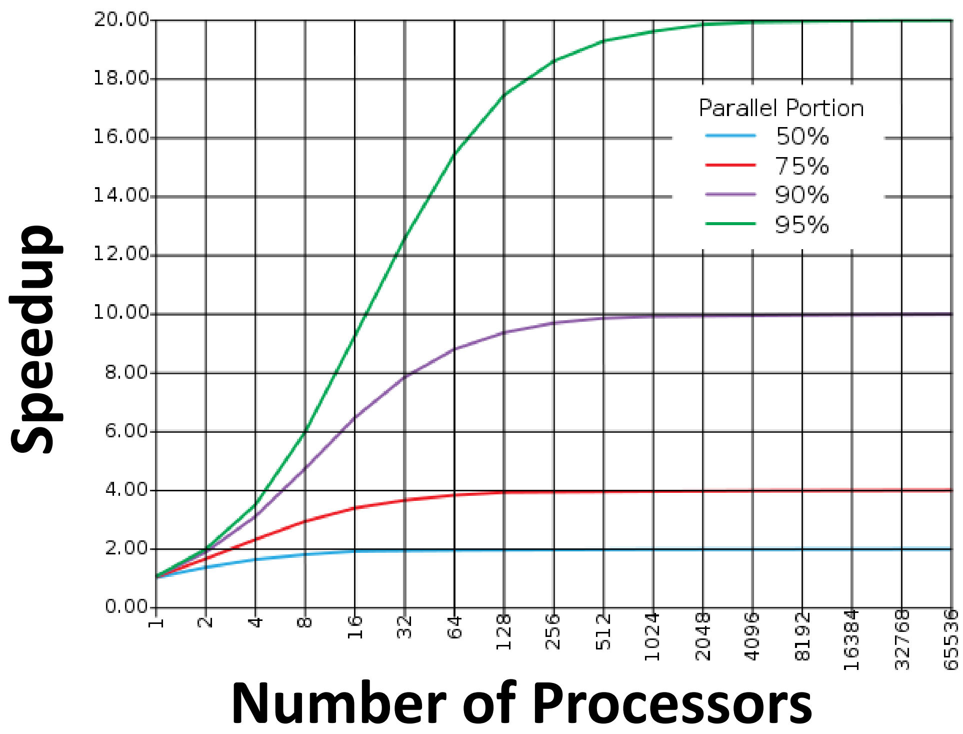
- What if it's usable 25% of the time?

$$\text{Speedup with improvement} = \frac{1}{(0.75)+(0.25/20)} = 1.311$$

Amdahl's (heartbreaking) Law (in pictures)

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!
  - And every program has at least *some* non-parallel parts





Amdahl's (heartbreaking) Law (in words)

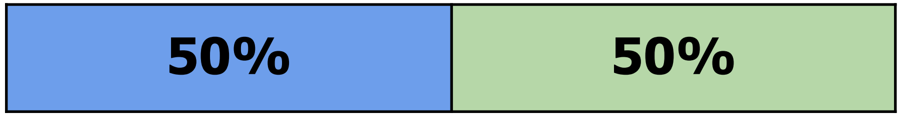
- Amdahl's Law tells us that to achieve linear speedup with more processors:
  - *none* of the original computation can be serial (non-parallelizable)

- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

$$\text{Speedup} = 1/(.001 + .999/100) = 90.99$$

Break + Question

Speedup with improvement $= \dfrac{1}{(1-F)+(F/S)}$



- Suppose a program spends 50% of its time in a `square root` routine.
- How much must you speed up `square root` to make the program run 2x faster?

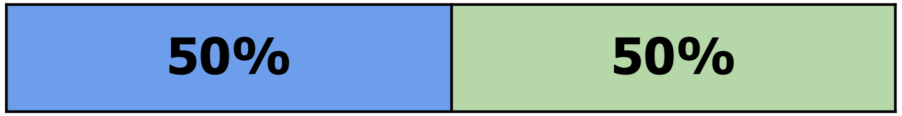
(A) 10

(B) 20

(C) 100

(D) None of the above

Break + Question



$$\text{Speedup with improvement} = \frac{1}{(1-F)+(F/S)}$$

- Suppose a program spends 50% of its time in a `square root` routine.
- How much must you speed up `square root` to make the program run 2x faster?

(A) 10
(B) 20
(C) 100
(D) **None of the above**

Speedup = 1 / [(1 - F) + (F/S)]

2 = 1 / [(1 - 0.5) + (0.5/S)]

S = 0.5 / ((1/2) – 0.5) = ∞

The square root would need to decrease to nothing before you got 2x speedup

Outline

- Threads
- Need for Parallelism
- Processor Concurrency
- **Concurrency Challenges**
  - Amdahl's Law
  - **Data Races**

Challenges to concurrency

Concurrency is great! We can do so many things!!

But what's the downside…?

1. How much speedup can we get from it?
2. **How hard is it to write parallel programs?**

Concurrency problem: data races

Consider two threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 1;
}
```

Thread 2:

```
void main(){
  count += 1;
}
```

`count` could end up with a final value of 1 or 2. How?

Concurrency problem: data races

Consider two threads with a shared global variable: `int count = 0`

Assuming "count" is in memory location 0x8049a1c

Thread 1:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

Thread 2:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

`count` could end up with a final value of 1 or 2. How?

These instructions could be interleaved in any way.

Data race example – Count = 2

Before this code starts

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	???

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming “count” is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	1

Assuming “count” is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
%eax	1

Thread 2

Register	Value
%eax	2

Memory

Variable	Value
count	1

Assuming "count" is in memory location pointed to by **%edi**

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	2

Memory

Variable	Value
`count`	2

Assuming "count" is in memory location pointed to by **`%edi`**

Theads do not have guaranteed ordering

BUT, there’s no guarantee that the instructions occur in that order!

Since the two threads are running in parallel, the instructions could be interleaved in any way
(both threads are really running simultaneously)

Data race example – Count = 1

Remember, each thread has its own separate registers!

Before this code starts

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	???

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming “count” is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	0

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by `%edi`

Data race comparison

Assuming "count" is in memory location pointed to by **`%edi`**

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Final value of count: 2

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Final value of count: 1

Data race explanation

- Thread scheduling is **non-deterministic**
  - There is no guarantee that any thread will go first or last or not be interrupted at any point

- If different threads write to the same variable
  - The final value of the variable is also non-deterministic
  - This is a *data race*

Check your understanding: data races with multiple threads

Consider three threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 2;
}
```

Thread 2:

```
void main(){
  count -= 2;
}
```

Thread 3:

```
void main(){
  count += 3;
}
```

What are the possible values of count?

Check your understanding: data races with multiple threads

Consider three threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 2;
}
```

Thread 2:

```
void main(){
  count -= 2;
}
```

Thread 3:

```
void main(){
  count += 3;
}
```

What are the possible values of count? **-2, 0, 1, 2, 3, 5**

How are you supposed to reason about this?!
Need mechanisms for sharing memory.

Outline

- Threads
- Need for Parallelism
- Processor Concurrency
- Concurrency Challenges
  - Amdahl's Law
  - Data Races