

Lecture 03: Concurrency Sources and Challenges

CS343 – Operating Systems
Branden Ghena – Fall 2025

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

Administrivia

- Getting Started Lab
 - Let us know if you're having problems with this
 - Should not take long to complete
 - 61/120 of you have finished already
- Office Hours
 - Coming soon...
- PCLab Partnership Survey
 - Out now. Sign up if you do not have a partner

Today's Goals

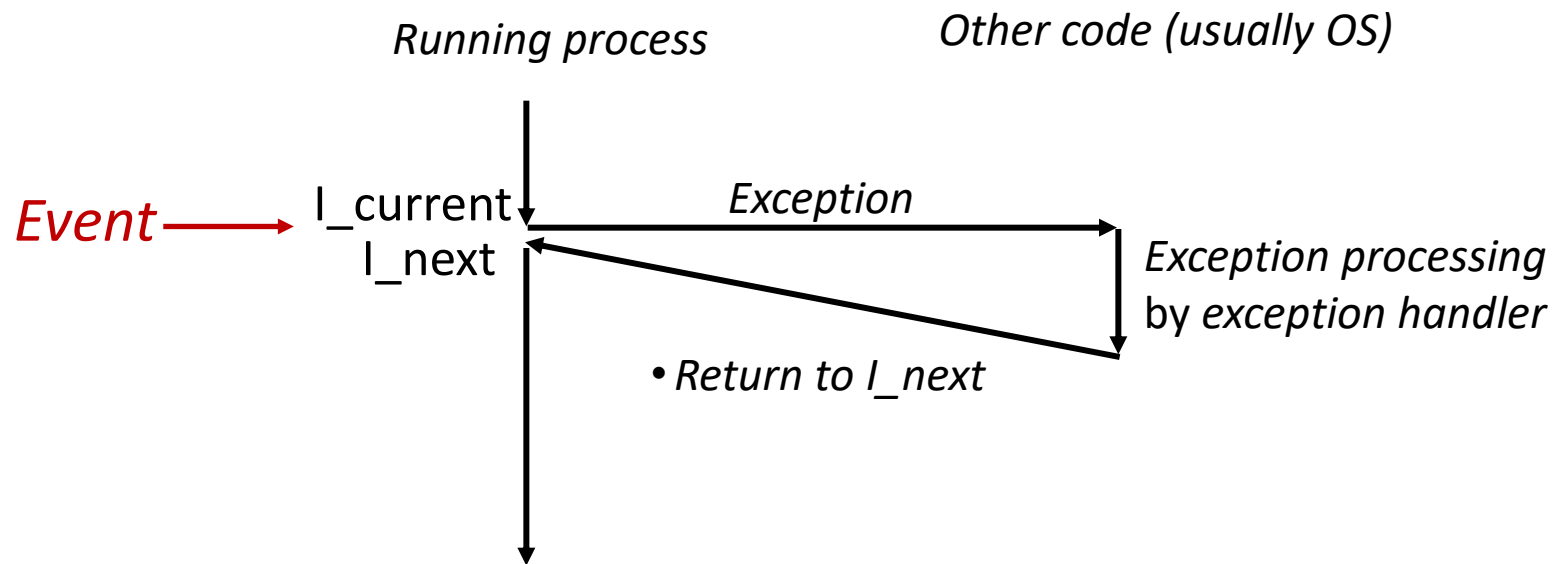
- Understand threads as a software design mechanism.
- Describe where and why concurrency and parallelism are involved in computing.
- Discuss multiple sources of concurrency on computing systems
- Be disappointed by performance limits on concurrency.

Outline

- Processes
- Context Switching
 - Running a Process
 - Exceptions
 - Running the Kernel
- System Calls
- **Signals**
- Threads

Exceptional control flow

- Mechanisms that could cause exceptional control flow
 - Exceptions: events cause execution to jump to OS handler
 - System calls: request from process causes execution to jump to OS
 - Signals: OS causes process to jump to process handler



Alerting processes of events

- How do we let a process know there was an event?
 - Errors
 - Termination
 - User commands (like CTRL-C or CTRL-\)
- Events could happen whenever
 - Need to interrupt process control flow and run an event handler
 - Linux mechanism to do so is called "signals"

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
 - Your child process completed
 - You tried to use an illegal instruction
 - You accessed invalid memory
 - You are terminating now
- In POSIX systems, this idea is called “Signals”

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	...			

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
 - Your child process completed
 - You tried to use an illegal instruction
 - You accessed invalid memory
 - You are terminating now
- In POSIX systems, this idea is called “Signals”

1) SIGHUP	2) STGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	...			

Process Errors

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
 - Your child process completed
 - You tried to use an illegal instruction
 - You accessed invalid memory
 - You are terminating now
- In POSIX systems, this idea is called “Signals”

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGTILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	...			

Process Termination

Sending signals

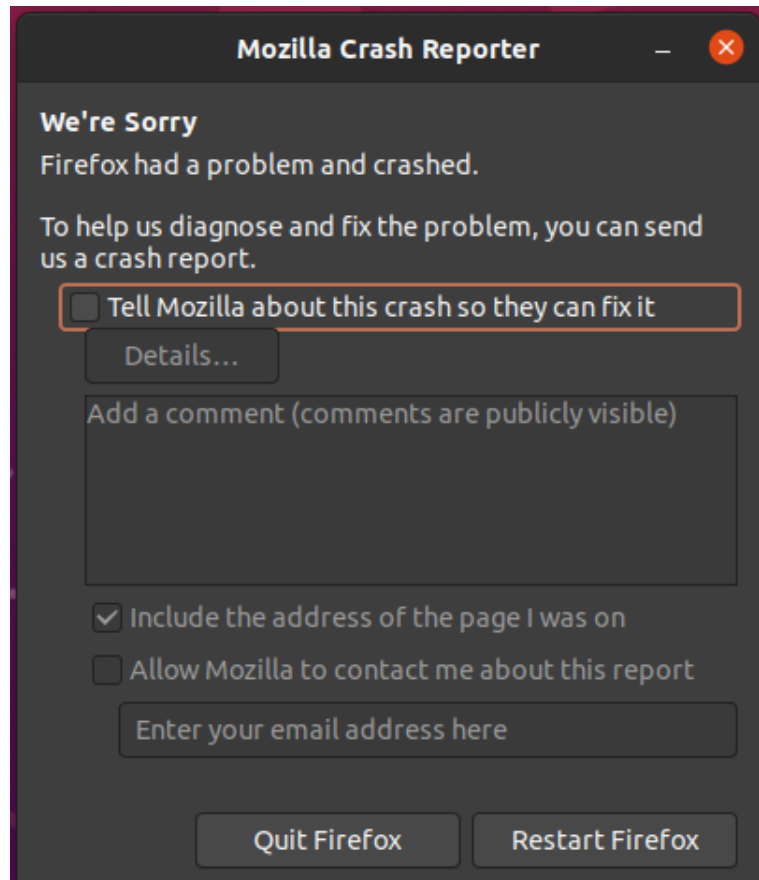
- OS sends signals when it needs to
- Processes can ask the OS send signals with a system call
 - `int kill(pid_t pid, int sig);`
- Users send signals through OS from command line or keyboard
 - Shell command: `kill -9 pid` (SIGKILL)
 - CTRL-C (SIGINT)

Handling signals

- Programs can register a function to handle individual signals
 - `signal(int sig, sighandler_t handler);`
- OS keeps track of signal handlers for each signal
 - Calls that function when a signal occurs
- What is the process supposed to do about it?
 - Do some *quick* processing to handle it
 - That needs to be “reentrant” safe
 - Reset the process and try again
 - Quit the process (default handler)

Examples: sending a signal

> `kill -11 pid` (11 is SIGSEGV – a.k.a segfault)



```
[brghena@ubuntu ~] $ firefox
ExceptionHandler::GenerateDump cloned child 55274
ExceptionHandler::SendContinueSignalToChild sent continue signal to child
ExceptionHandler::WaitForContinueSignal waiting for continue signal...
Exiting due to channel error.
Exiting due to channel error.
Exiting due to channel error.
Segmentation fault (core dumped)
[brghena@ubuntu ~] $
```

Outline

- **Threads**
- Need for Parallelism
- Processor Concurrency
 - Instruction-level parallelism
 - Task parallelism
 - Interrupts
- Concurrency Challenges
 - Amdahl's Law

Software Tasks: Threads

Unit of execution *within* a process

Processes discussed so far have a single thread

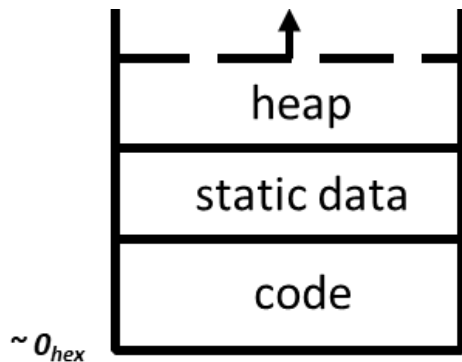
- They “have a single thread of execution”
- They “are single-threaded”

But a single process could have multiple threads

View of a process

- Process: program that is being executed
- Contains code, data, and a thread
 - Thread contains registers, instruction pointer, and stack

• Code and Data



• Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

• Instruction Pointer

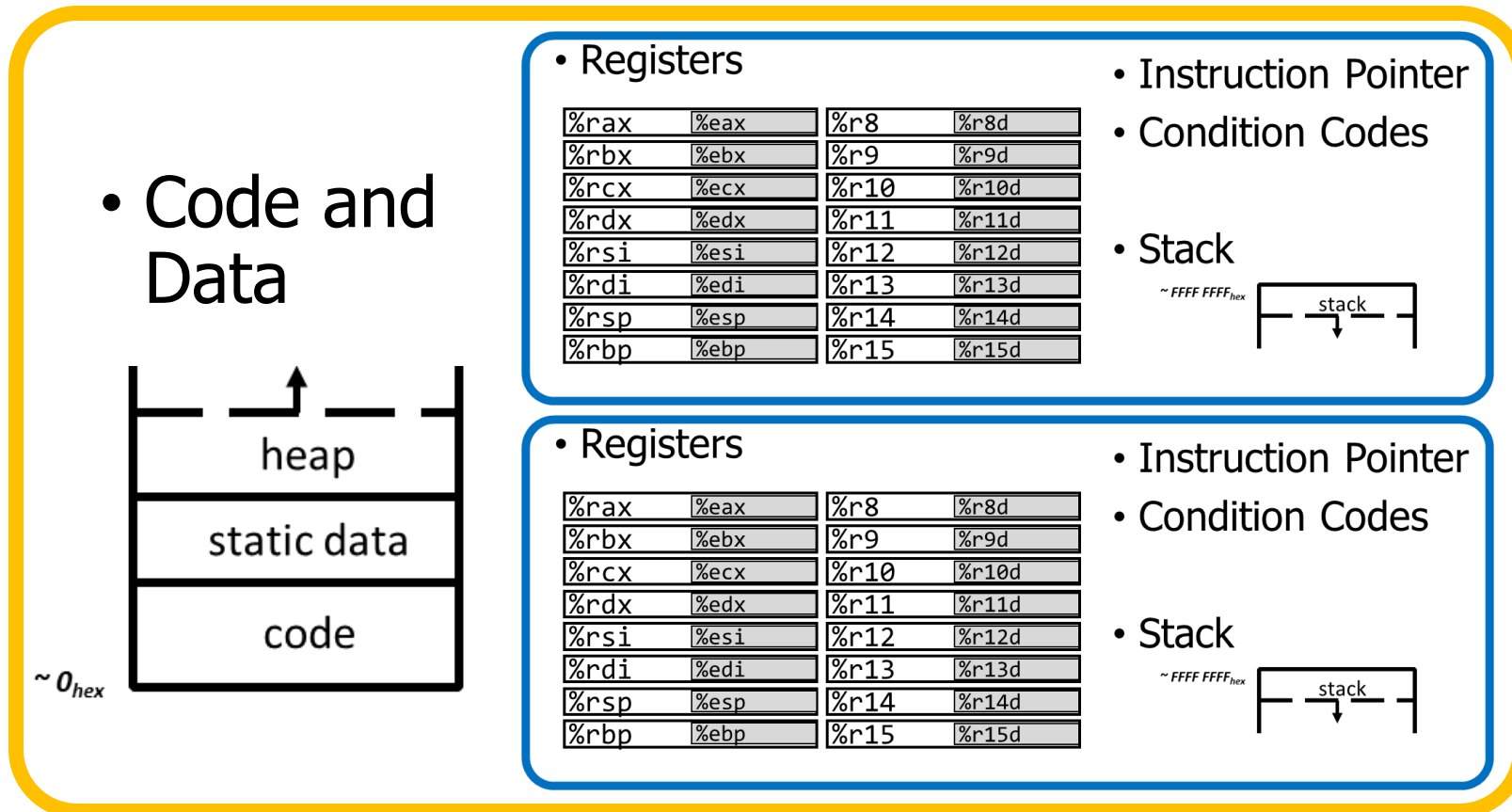
• Condition Codes

• Stack



Processes and threads

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



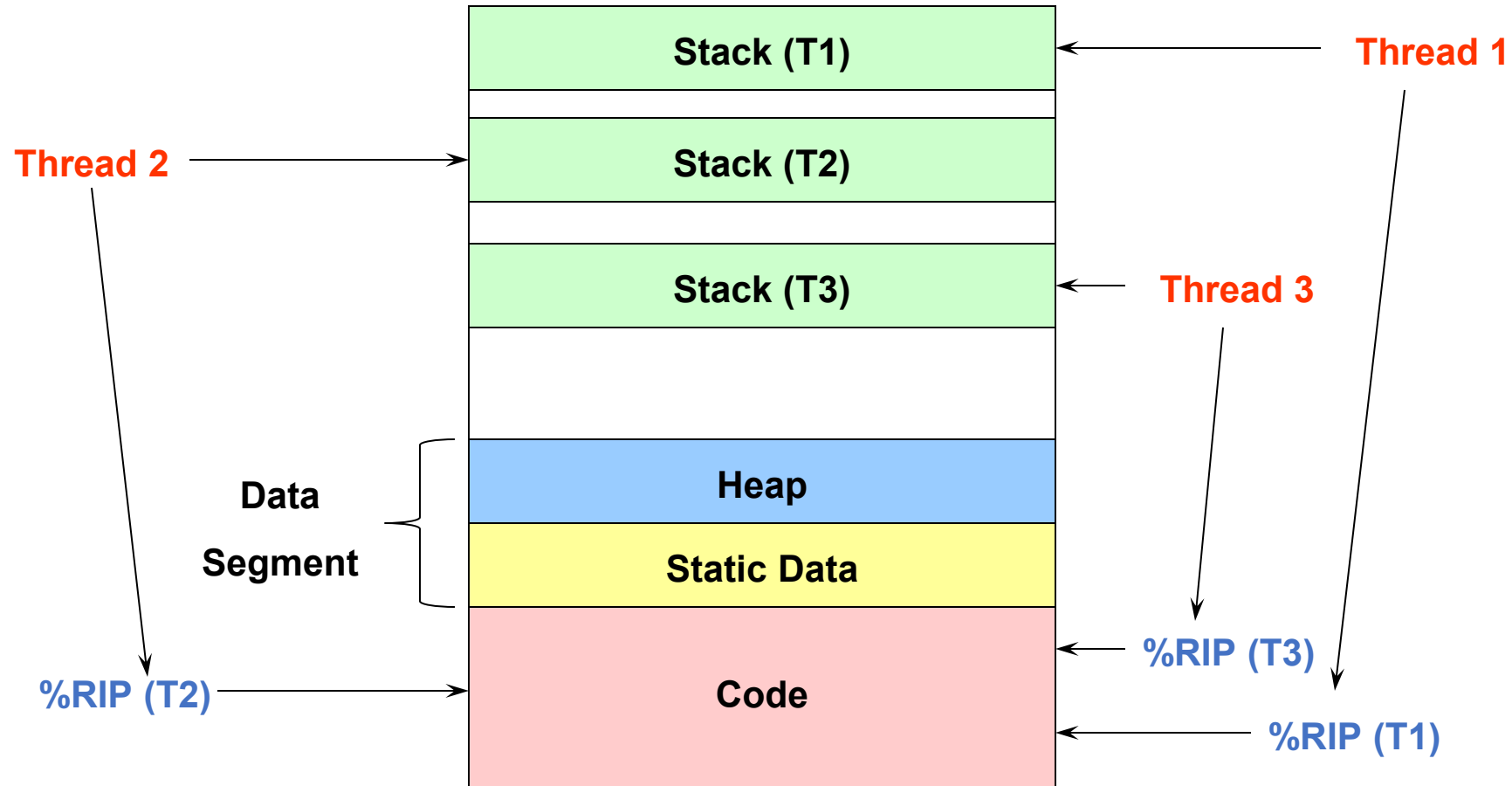
Threads have separate:

- Instruction Pointer
- Registers
- Stack Memory
- Condition Codes

Threads share:

- Code
- Global variables

Process address space with threads



Thread use case: web browser

Let's say you're implementing a web browser:

You want a tab for each web page you open:

- The same code loads each website (shared code section)
- The same global settings are shared by each tab (shared data section)
- Each tab does have separate state (separate stack and registers)

Disclaimer: Actually, modern browsers use separate processes for each tab for a variety of reasons including performance and security. But they used to use threads.

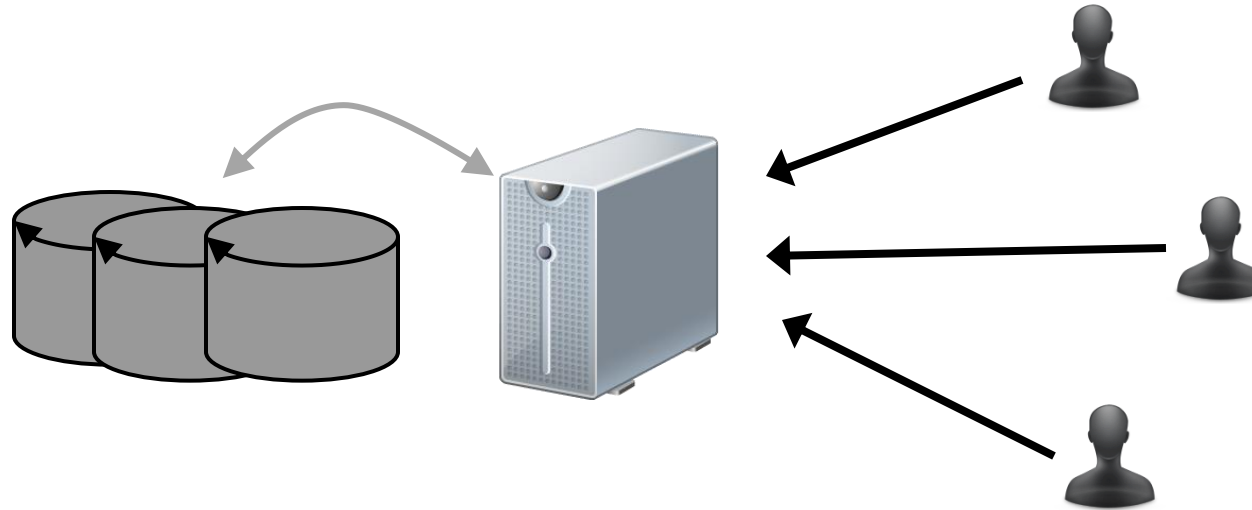
Thread use case: user interfaces

- Even if there is only a single processor core, threads are useful
- Single-threaded User Interface
 - While processing actions, the UI is frozen

```
main() {  
    while(true) {  
        check_for_UI_interactions();  
        process_UI_actions(); // UI freezes while processing  
    }  
}
```

Thread use case: web server

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



Web server option 1: handle one request at a time

Request 1 arrives

Server reads in request 1

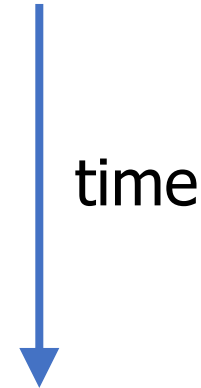
Server starts disk I/O for request 1

Request 2 arrives

Disk I/O for request 1 finishes

Server responds to request 1

Server reads in request 2



- Easy to program, but slow
 - Can't overlap disk requests with computation
 - Can't overlap either with network sends and receives

Web server option 1: event-driven model

- Issue I/Os, but don't wait for them to complete

Request 1 arrives

Server reads in request 1

Server starts disk I/O for request 1

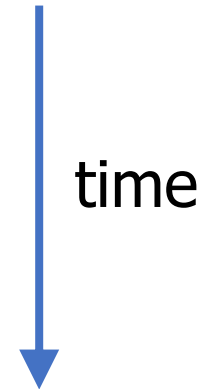
Request 2 arrives

Server reads in request 2

Server starts disk I/O for request 2

Disk I/O for request 1 completes

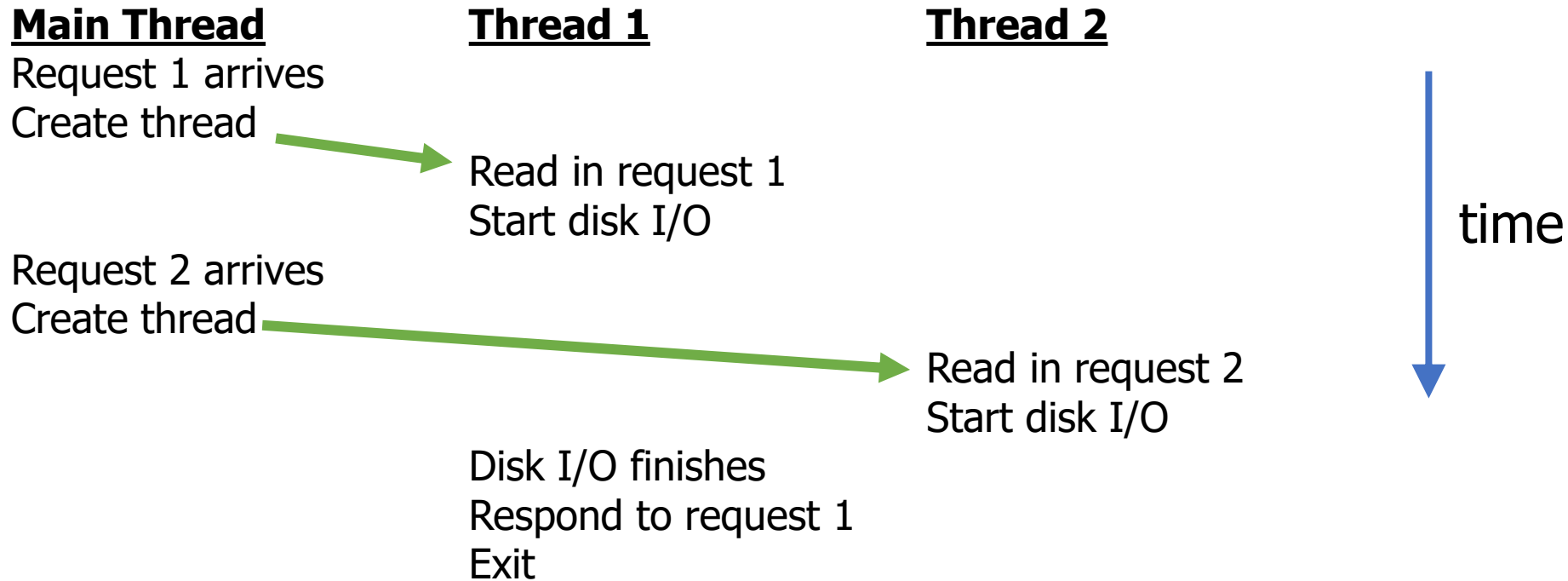
Server responds to request 1



- Fast, but hard to program
 - Must remember which requests are in flight and which I/O goes where
 - Lots of extra state

Web server option 3: multi-threaded web server

- One thread per request. Thread handles only that request.



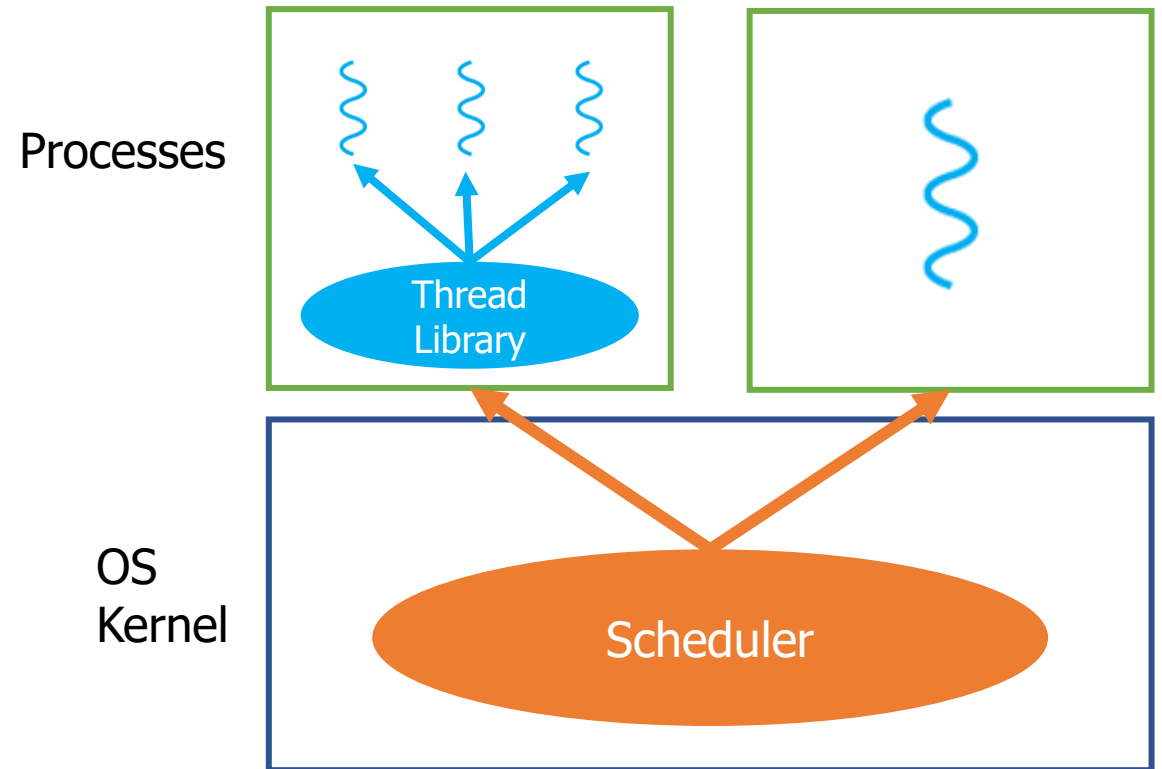
- Easy to program (maybe), and fast!
 - State is stored in the stacks of each thread and the thread scheduler
 - Simple to program if they are independent...

How are threads implemented?

- Two major possibilities
 - User Threads
 - Kernel Threads
- (There are other options that mix these)

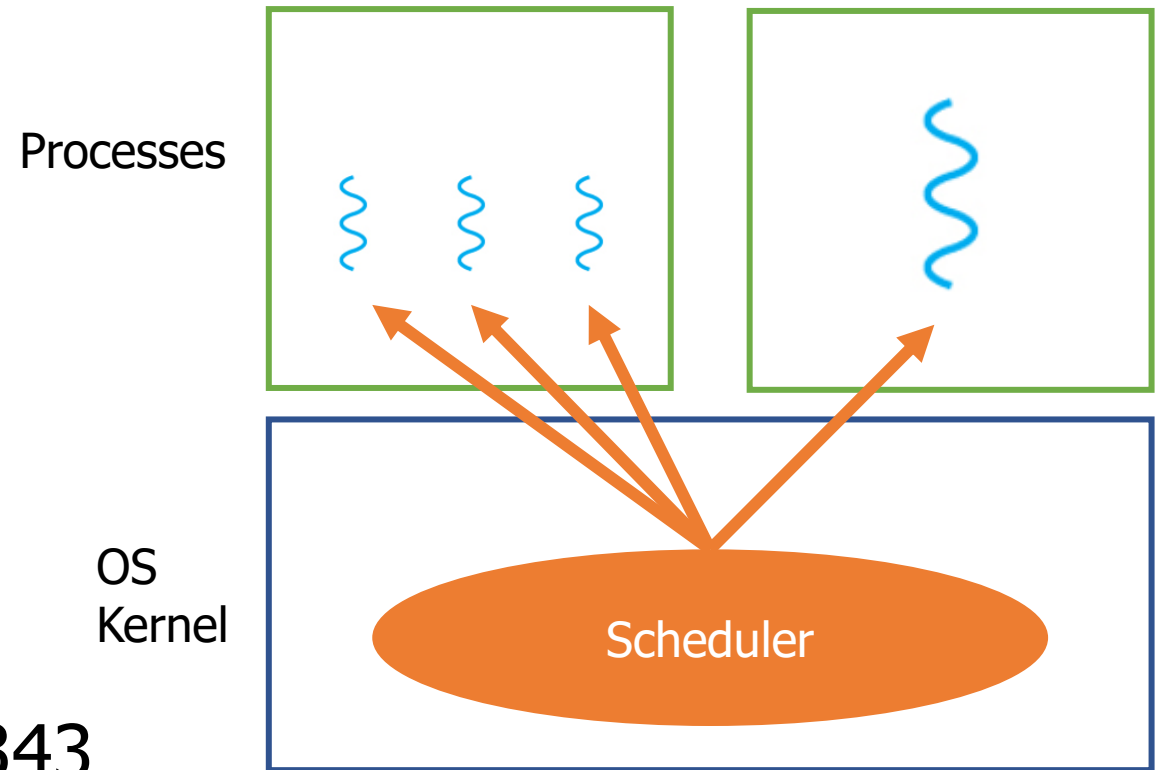
Models for thread libraries: **User Threads**

- Thread scheduling is implemented within the process
 - OS only knows about the process, not the threads
- Upsides
 - Works on any hardware or OS
 - Performance is better when creating and switching
- Downsides
 - A system call in any thread **blocks all threads**



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 ← clone (56) syscall on Linux  
    Put args into registers  
    Special trap instruction
```

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
 - Thread number
 - Memory address of its stack
 - Memory address of global variable
 - Global value, starting at 162, which it increments
- 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)
```

```
#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

```
#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?

```
#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!**

```
#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
 - Instruction-level parallelism
 - Task parallelism
 - Interrupts
- Concurrency Challenges
 - Amdahl's Law

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**

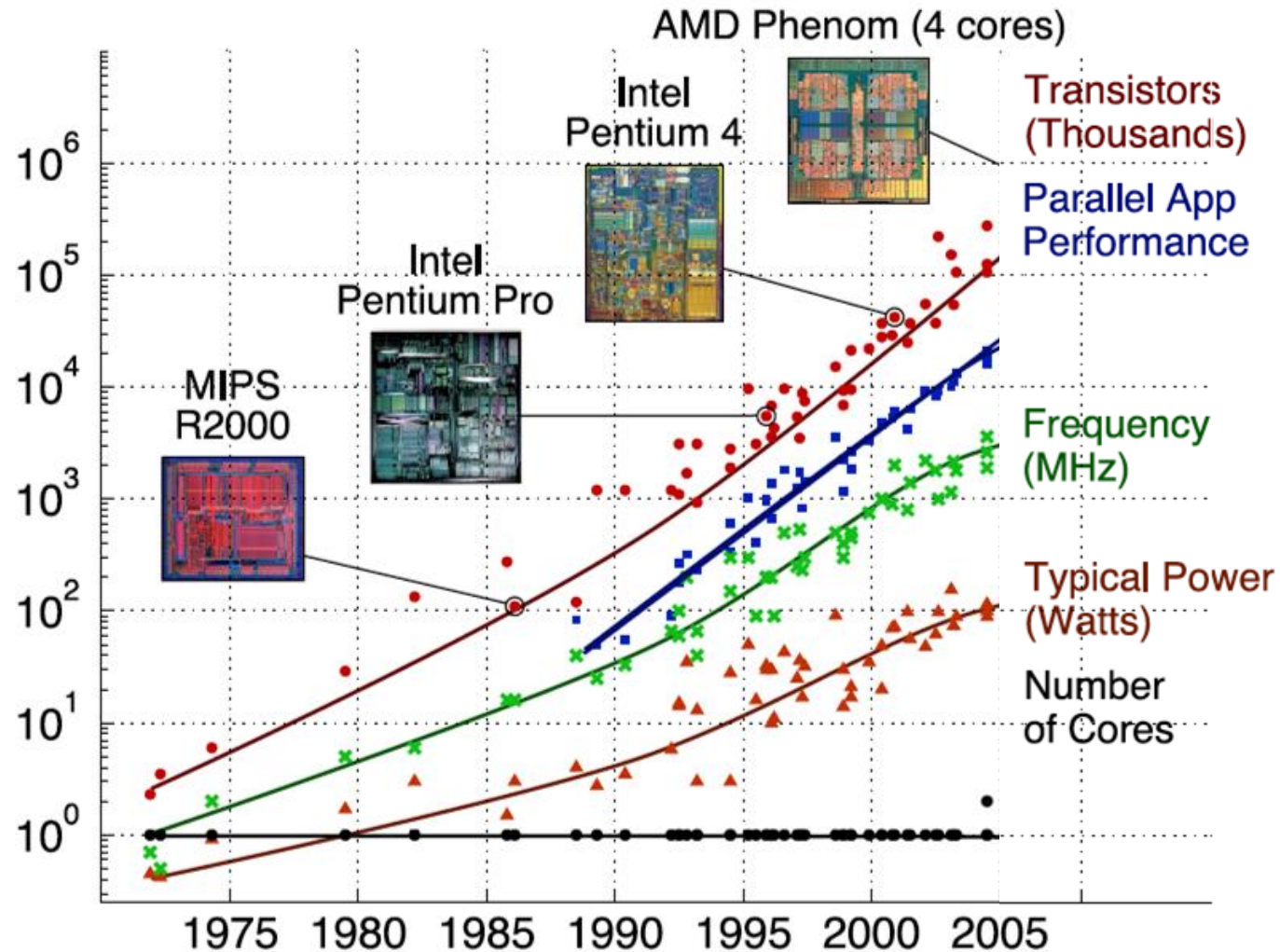
Transistor count



How? Transistors are getting exponentially smaller!

How small? Today: $< 7\text{nm}$!
(maybe smaller, kind of complicated)
 $< \frac{1}{2}$ the size of most viruses!

Processors kept getting faster too



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olu

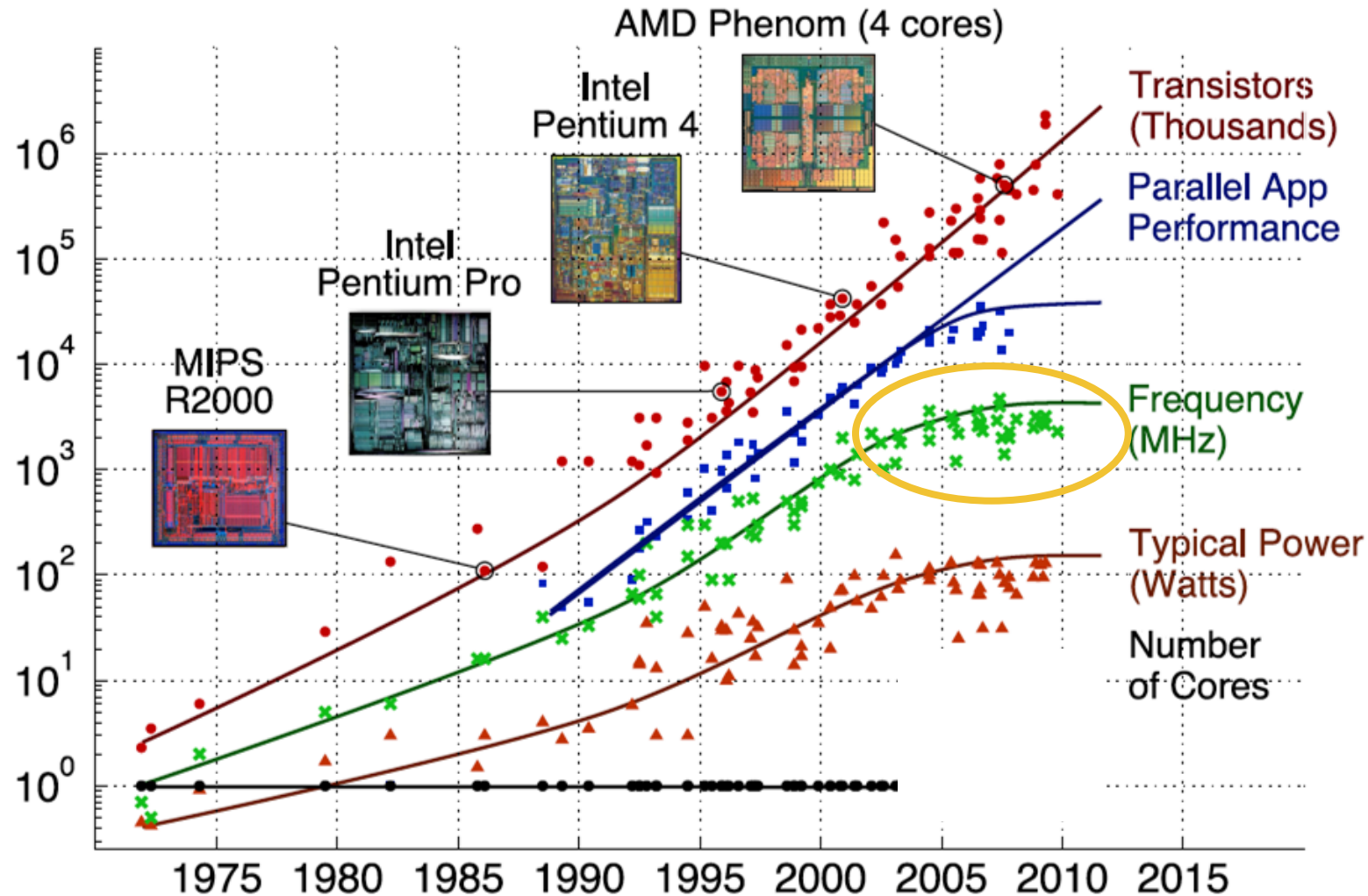
Power is a major limiting factor on speed

- 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: Denard 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



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

~2006: Leakage current becomes significant

Now smaller transistors doesn't mean lower power

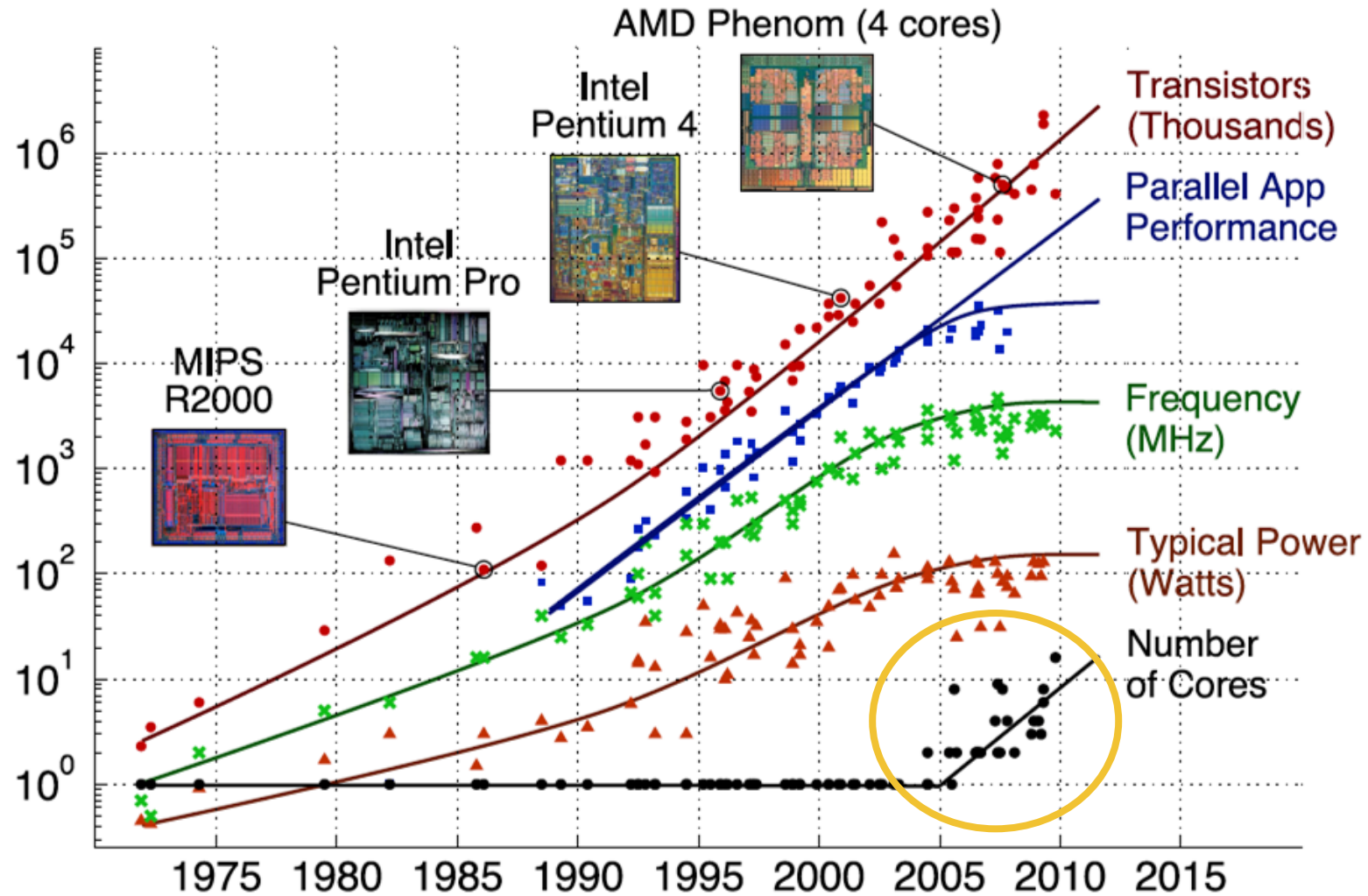
So... now what?

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?

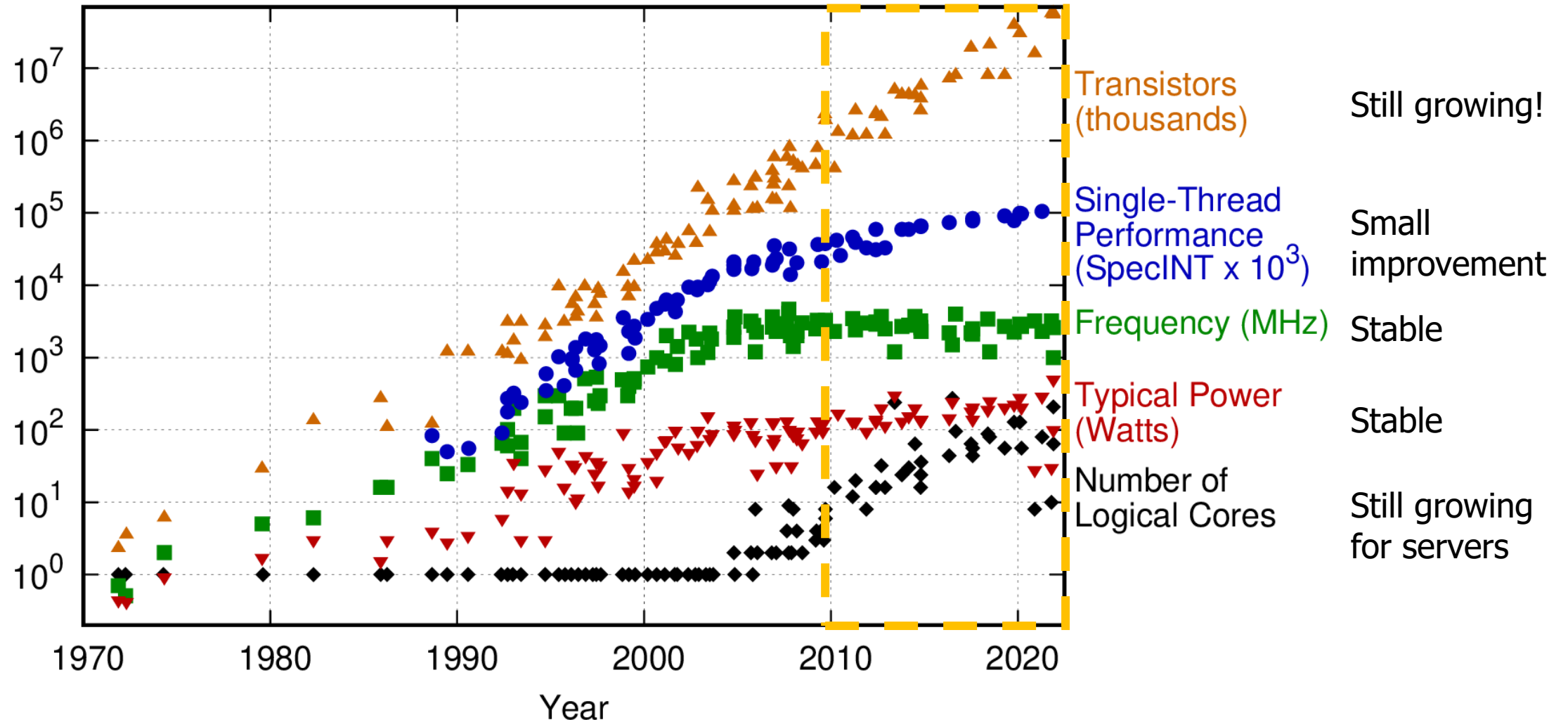
Exploit parallelism!



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Update: 2010-2021

50 Years of Microprocessor Trend Data



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

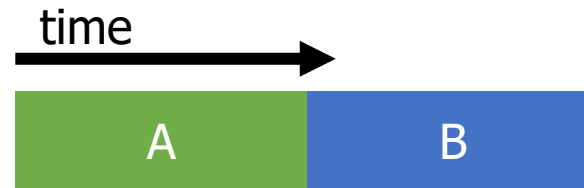
- I can get 99 friends to help me
- Whenever one result doesn't depend on another, doing the task in parallel can be a big win!

Parallelism versus Concurrency

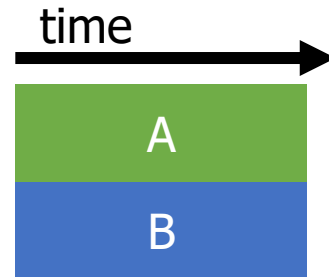
Two processes A and B



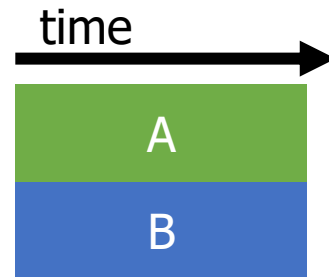
Serial execution



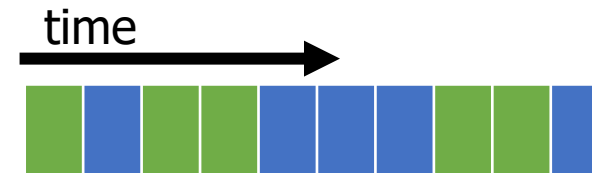
Parallel execution



Concurrent execution

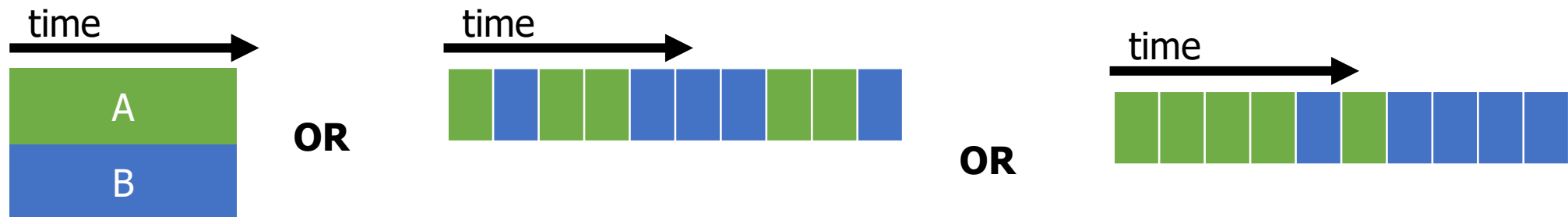


OR



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



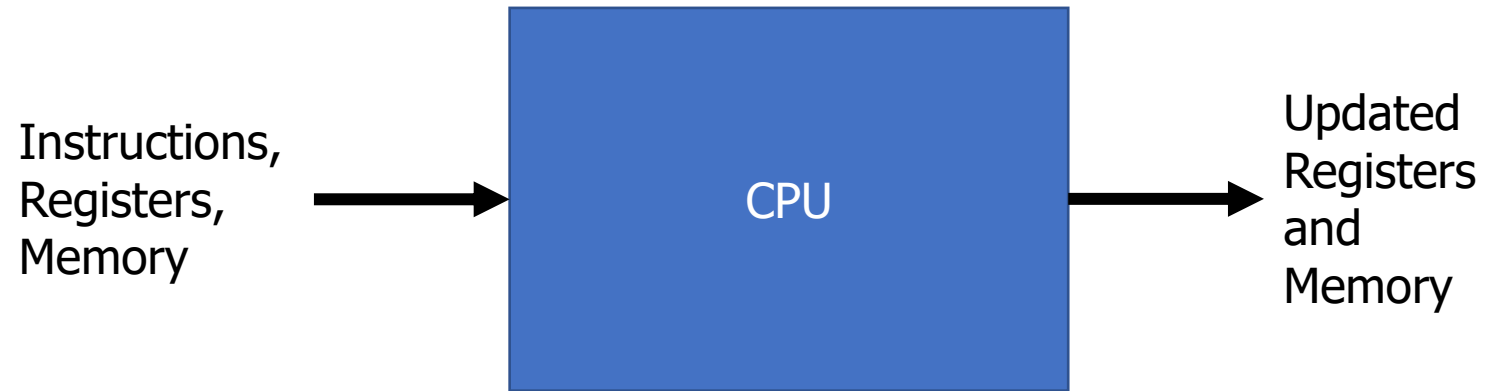
Outline

- Threads
- Need for Parallelism
- **Processor Concurrency**
 - **Instruction-level parallelism**
 - Task parallelism
 - Interrupts
- Concurrency Challenges
 - Amdahl's Law

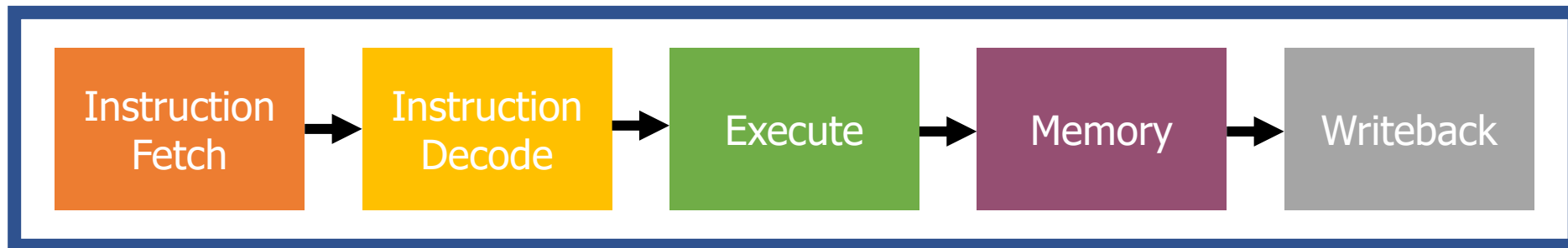
Hardware sources of concurrency

- Instruction-level parallelism
- Task parallelism
 - Multiple processes
 - Multiple threads
- Interrupts

Model of a processor



CPU



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

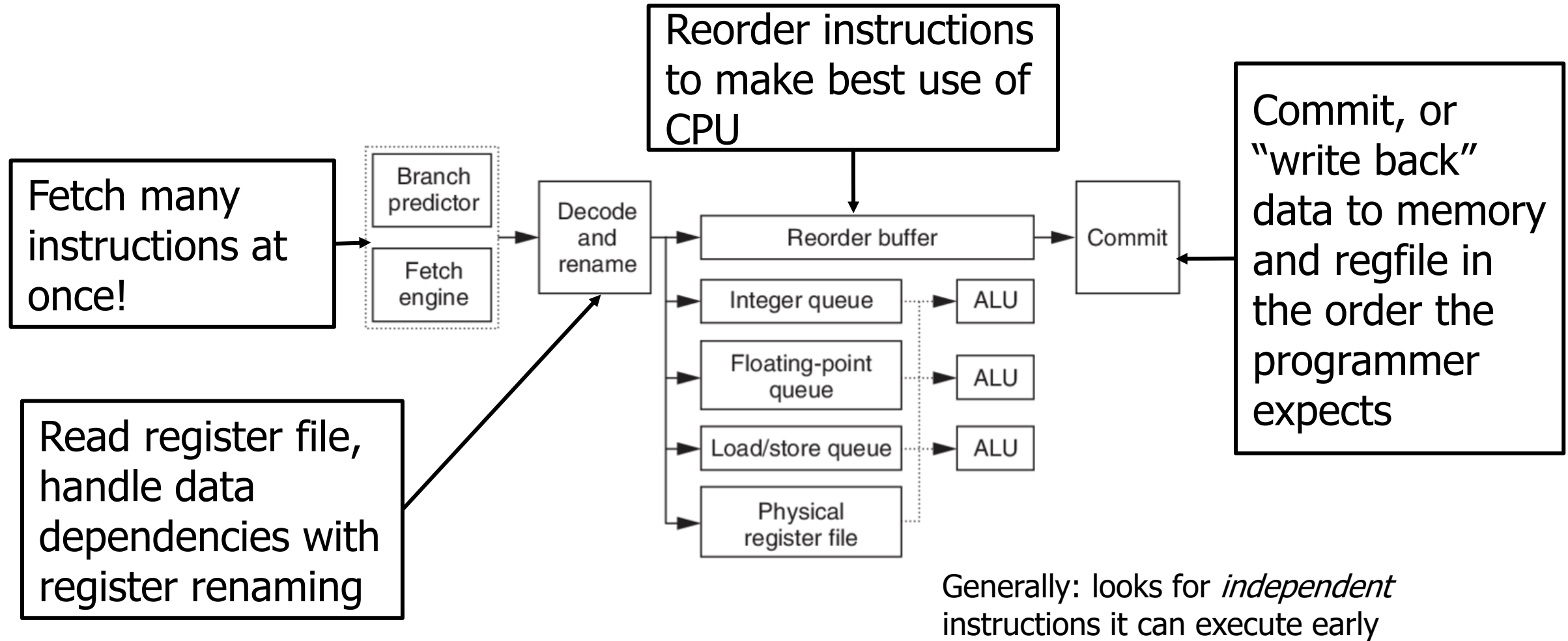
```
movq    (%rdi), %rax  
movq    (%rsi), %rdx  
movq    %rdx, (%rdi)  
movq    %rax, (%rsi)  
addq    %rcx, %rbx
```

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



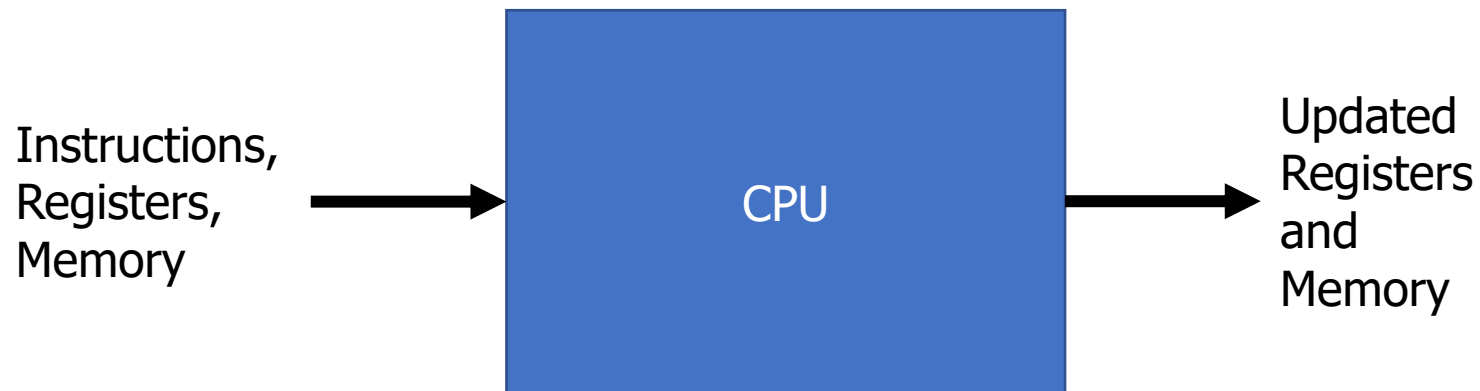
We can apply the multiprocessing approach of executing this
addq while the movq is waiting on memory.

Out-of-order processors



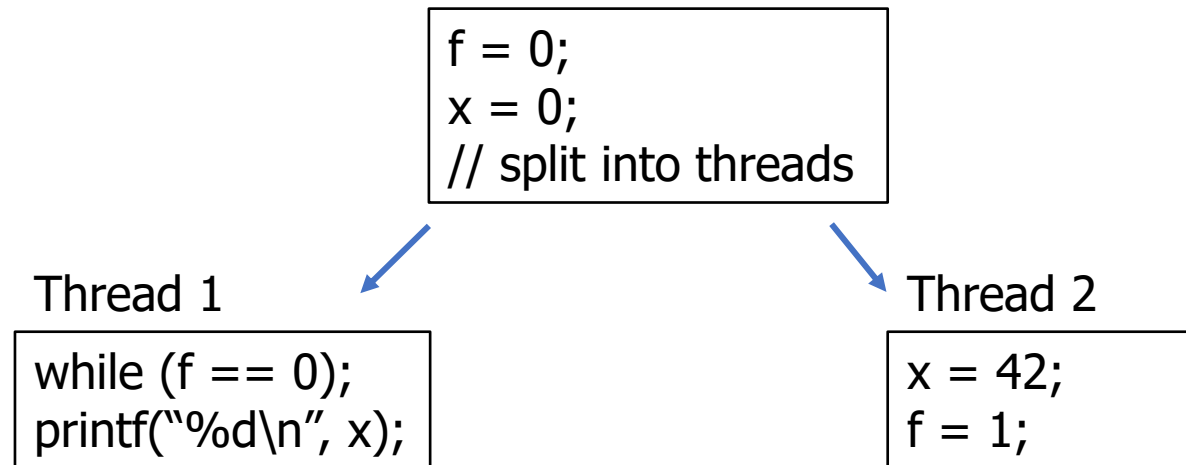
Out-of-order processors obey normal execution results

- 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

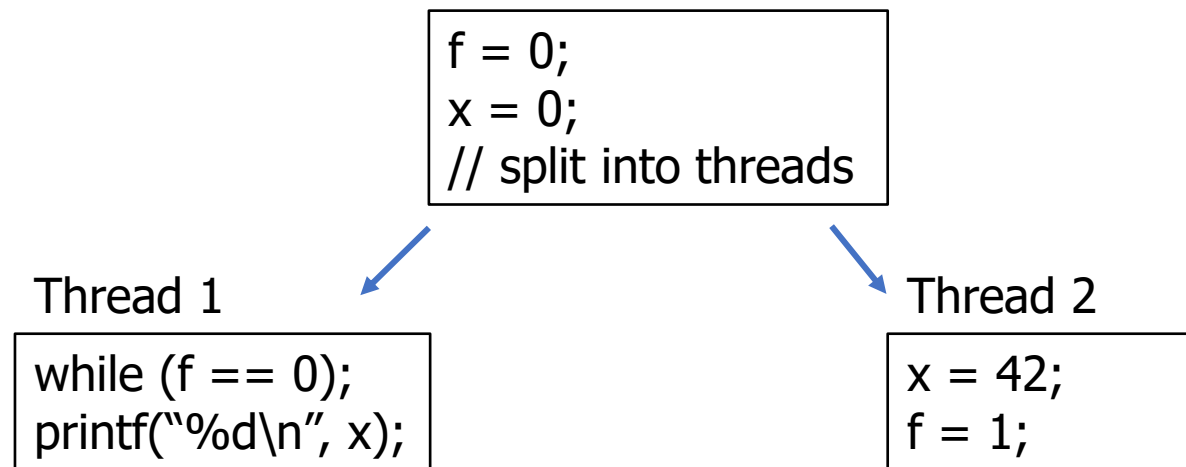
- The processor can't account for multiple threads though
- If memory results are shared by two threads, the processor might mess something up for you.



- 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.



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

This can be addressed with memory barriers

Outline

- Threads
- Need for Parallelism
- **Processor Concurrency**
 - Instruction-level parallelism
 - **Task parallelism**
 - Interrupts
- Concurrency Challenges
 - Amdahl's Law

Task parallelism use case

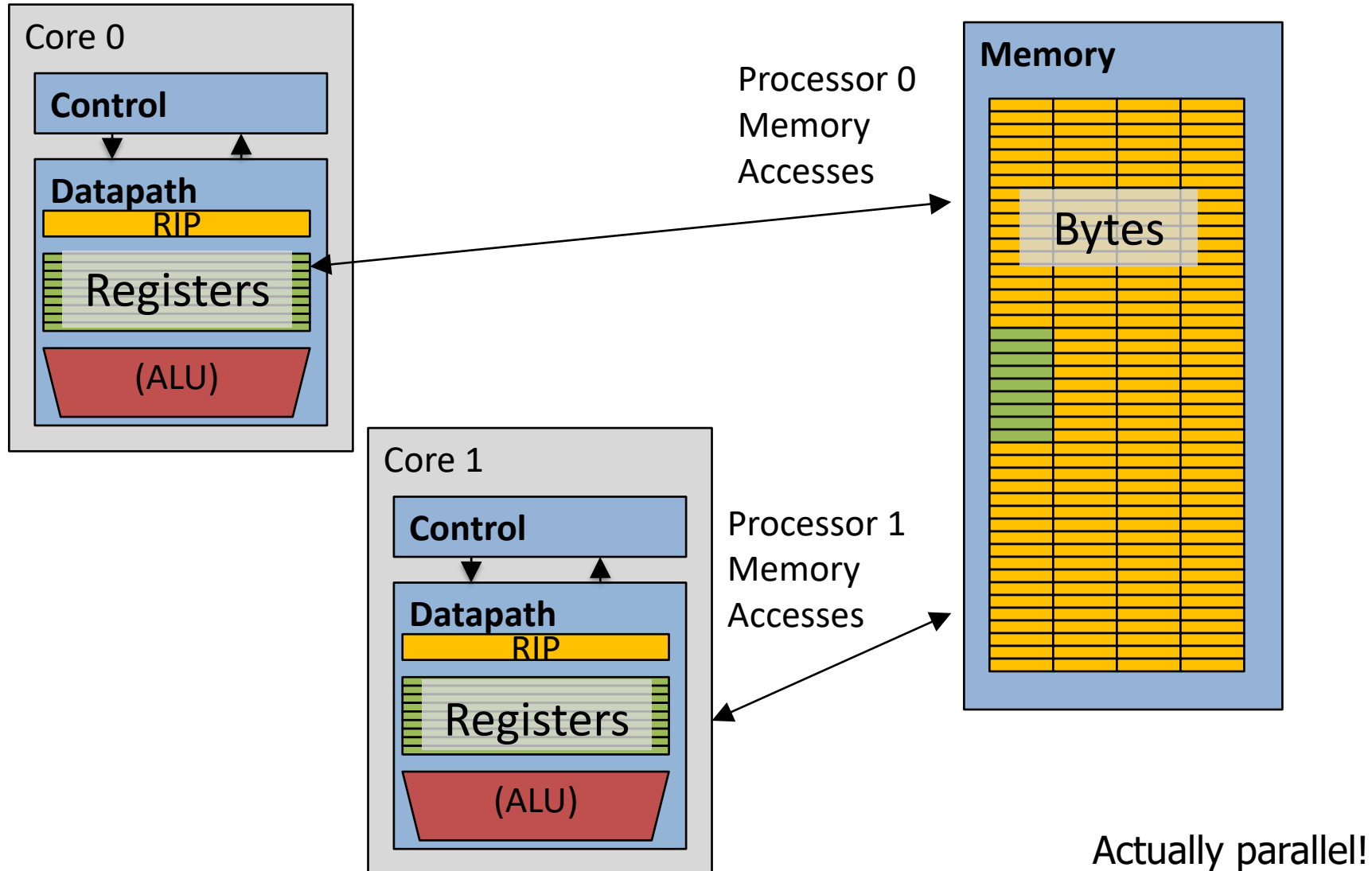
Run Chrome and Spotify simultaneously

- 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)



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

Multithreading processors

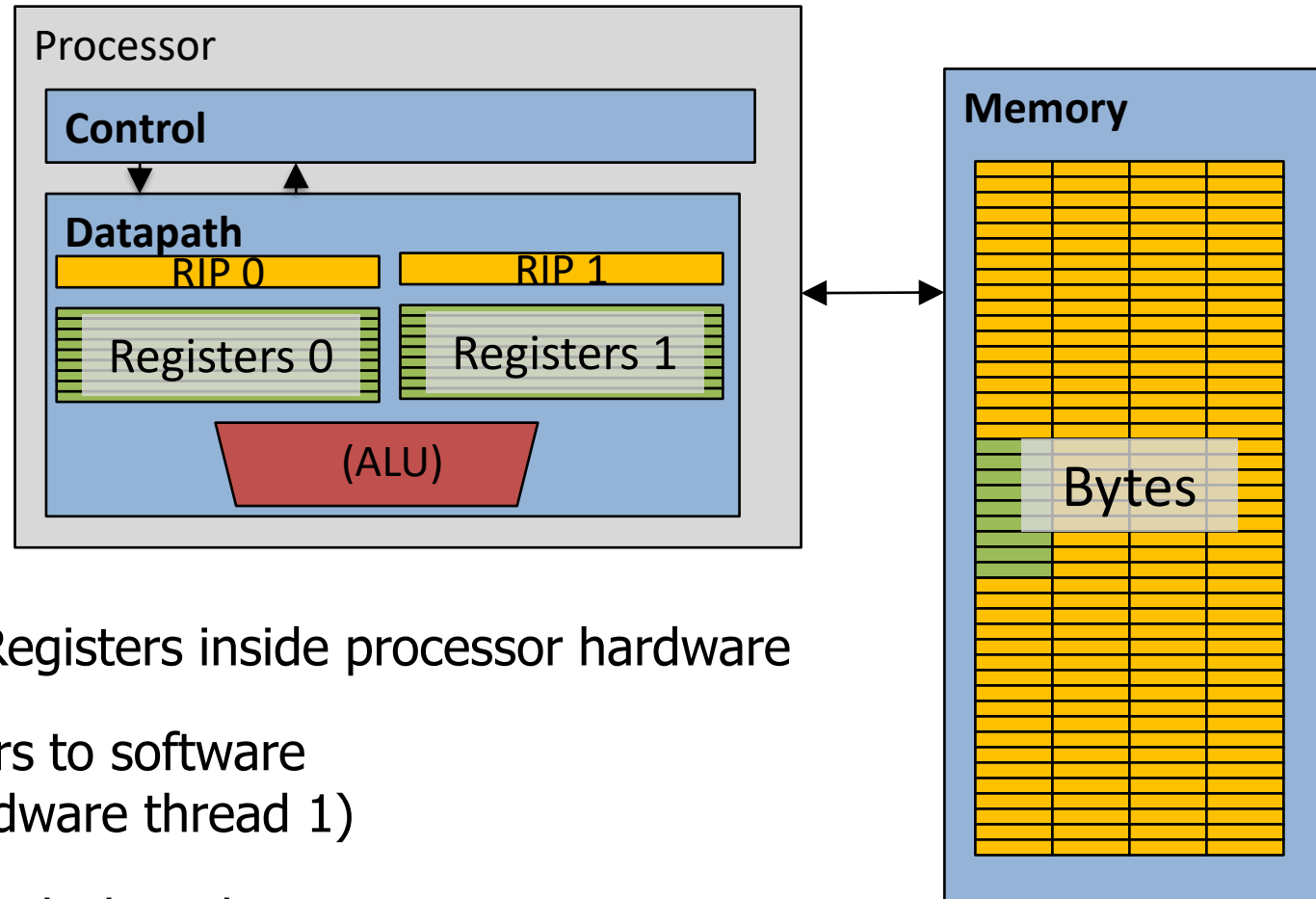
Basic idea: Processor resources are expensive and should not be left idle

- So have multiple threads ready to run at all times

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

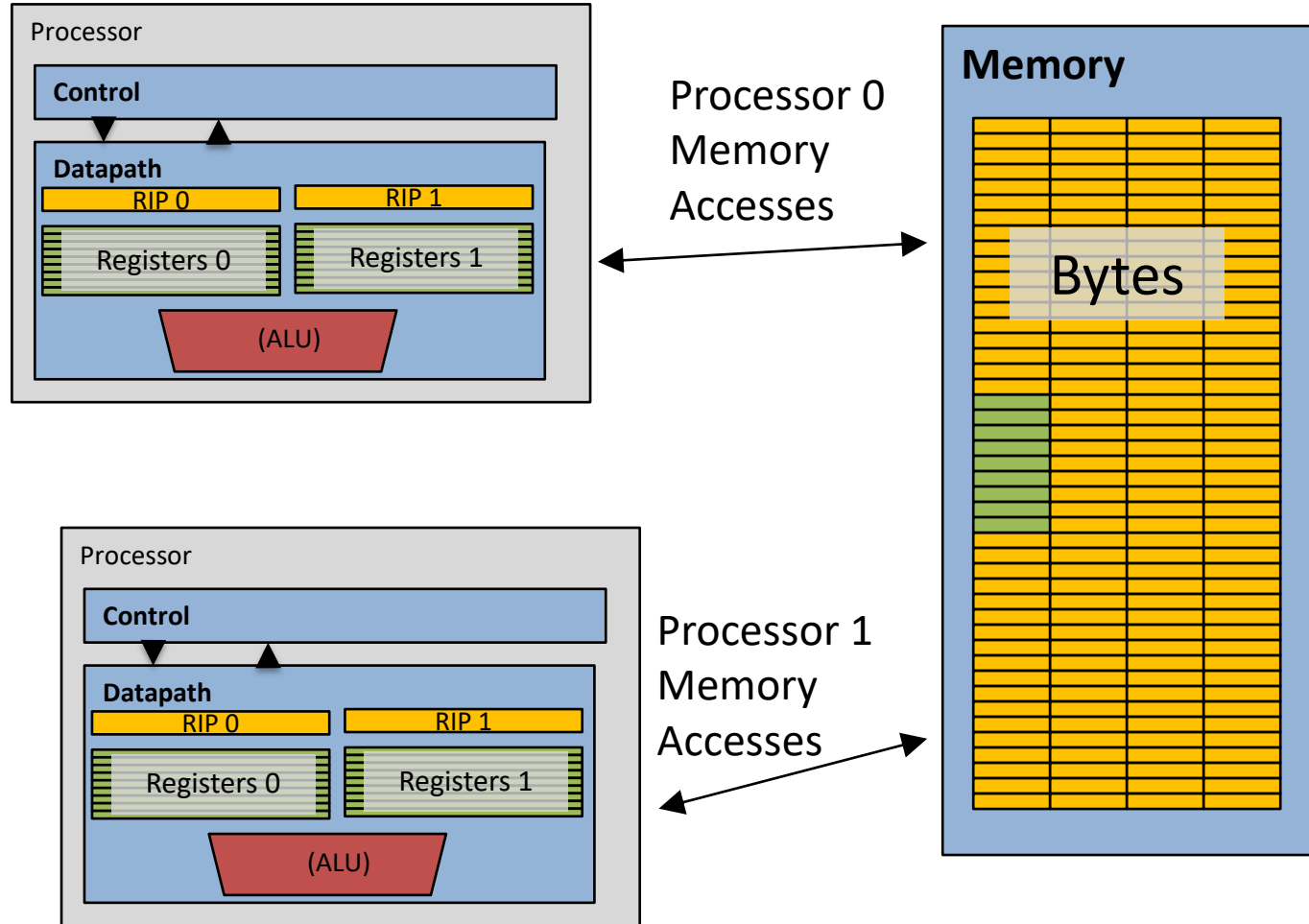


- Two copies of RIP and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next (concurrent, but NOT parallel)

Multithreading versus Multicore

- Multithreading => Better utilization
 - $\approx 5\%$ more hardware for $\approx 1.3x$ better performance?
 - Gets to share ALUs, caches, memory controller
- Multicore => Duplicate cores
 - $\approx 50\%$ more hardware for $\approx 2x$ 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

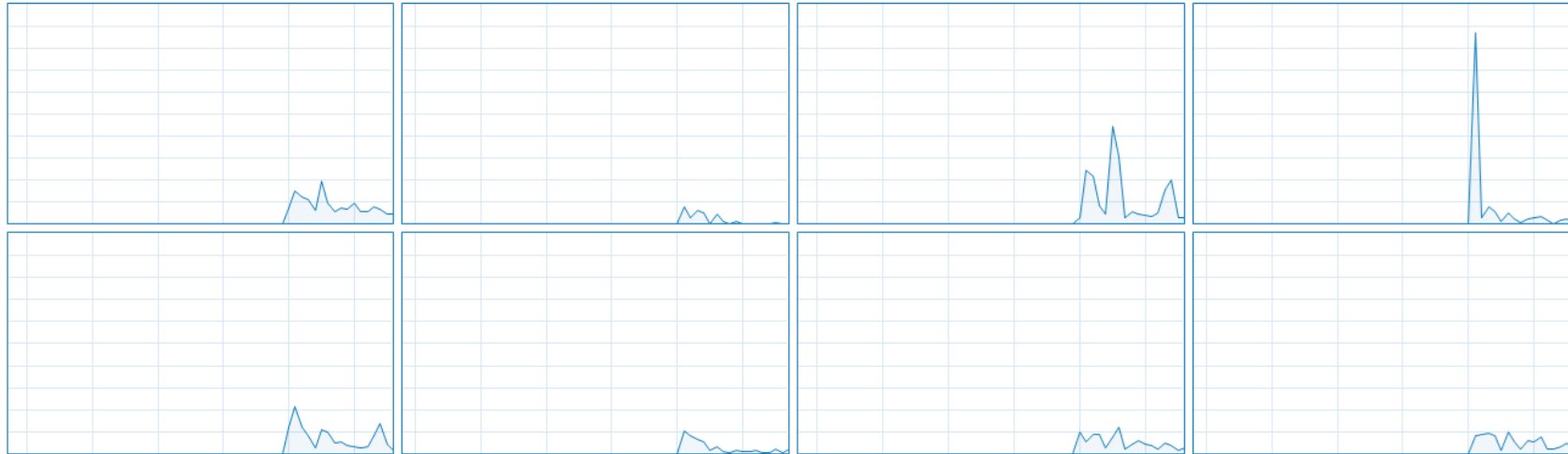
- Core: computation unit within the CPU
 - ALU, Registers, etc.
 - Capable of running one or more threads
- CPU (processor): the chip that goes in your computer
 - Contains one or more cores
 - Computers could have multiple CPU chips as well
- Sometimes people equate processors and cores, which is confusing
 - I'll definitely do it by mistake at some point if I haven't already. Sorry!

My desktop computer

CPU

Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz

% Utilization over 60 seconds



Utilization	Speed		Base speed:	3.60 GHz
2%	4.08 GHz		Sockets:	1
			Cores:	4
Processes	Threads	Handles	Logical processors:	8
236	2909	111153	Virtualization:	Enabled
Up time			L1 cache:	256 KB
12:02:28:40			L2 cache:	1.0 MB
			L3 cache:	8.0 MB

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

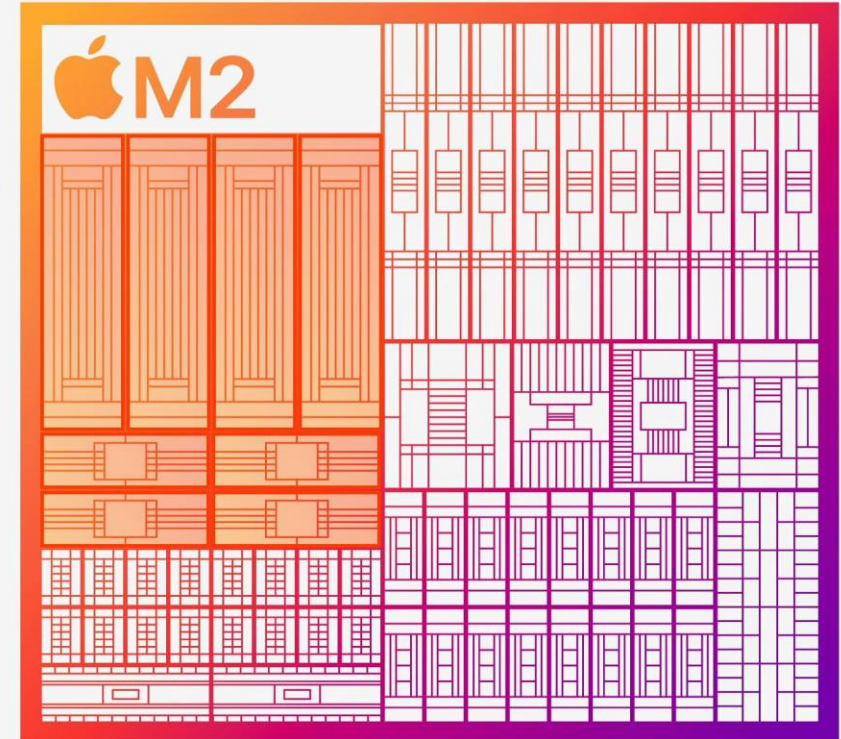
- Heterogeneous multicore
 - Not all cores are necessarily identical
- Enables scheduler to make complicated choices of performance or energy savings
 - At the cost of a complicated scheduler...

4 high-performance cores

Ultrawide microarchitecture
192KB instruction cache
128KB data cache
Shared 16MB cache

4 high-efficiency cores

Wide microarchitecture
128KB instruction cache
64KB data cache
Shared 4MB cache



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**
 - Instruction-level parallelism
 - Task parallelism
 - **Interrupts**
- Concurrency Challenges
 - Amdahl's Law

Interrupts

- An event that the processor handles by running special OS handler code
 - Timer expiration, Keyboard event, Network packet, etc.
 - Necessary for asynchronous event handling
 - Don't wait around for the event, just handle it whenever it happens
- Very similar to Exceptions
 - Might be synonyms, depending on the system
- A system call is a way to generate a software interrupt

Interrupt handlers

- Interrupt context
 - Running code in a special mode
 - Pauses whatever was running previously (kernel or process) until finished
- Handler code
 - Execute some *quick* processing to deal with the interrupt
 - Return so the hardware can bring us back to our normal operation
 - Cannot pause to wait for something else to finish first because the entire core jumped to handling this interrupt
- Handled by the operating system kernel
 - Processes are interrupted, but otherwise not normally involved

Differences from system calls

- Interrupts and exceptions are basically (maybe literally) the same hardware mechanism
- But when we performed a system call:
 - We knew it was about to happen
 - Set up our registers in advance
 - Performed what looked sort of like a function call
 - And we were always switching from process to kernel
- Interrupts can happen *whenever*.
 - This can get extremely complicated on modern systems with out-of-order execution, multiple cores and threads, and caches

Why are interrupts important to concurrency?

- Interrupts are a case where the kernel could have a data race with itself!!
 - Imagine being in the middle of an operation on a device
 - When an interrupt comes in for that same device
 - Data structures for the device could end up messed up
- Takeaway: concurrency isn't just about processes and threads
 - Many different software designs need to deal with it

OS outcomes

1. Threads are a useful programming model
 - Whether concurrent or parallel
 2. Parallelism is necessary for performance improvement
 3. Hardware introduces concurrency
-
- Result: OS is going to have to deal with concurrency
 - Both interleaving and true parallelism

Outline

- Threads
- Need for Parallelism
- Processor Concurrency
 - Instruction-level parallelism
 - Task parallelism
 - Interrupts
- **Concurrency Challenges**
 - **Amdahl's Law**

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?

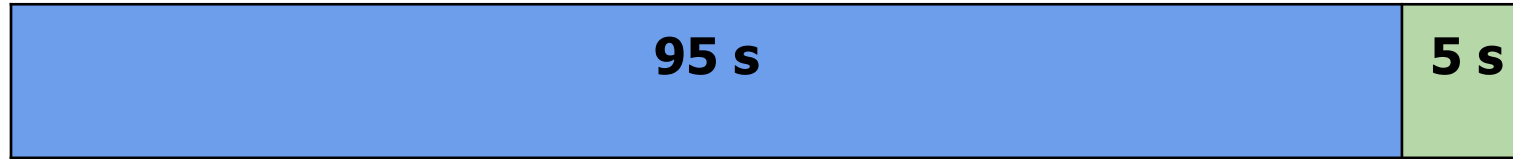
Challenges to concurrency

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



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

$$5 \text{ s} \rightarrow 2.5 \text{ s: Speedup} = 100/97.5 = 1.026$$

$$5 \text{ s} \rightarrow 1 \text{ s: Speedup} = 100/96 = 1.042$$

$$5 \text{ s} \rightarrow 0.001\text{s: Speedup} = 100/95.001 = 1.053$$

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 $\rightarrow$ 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

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

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

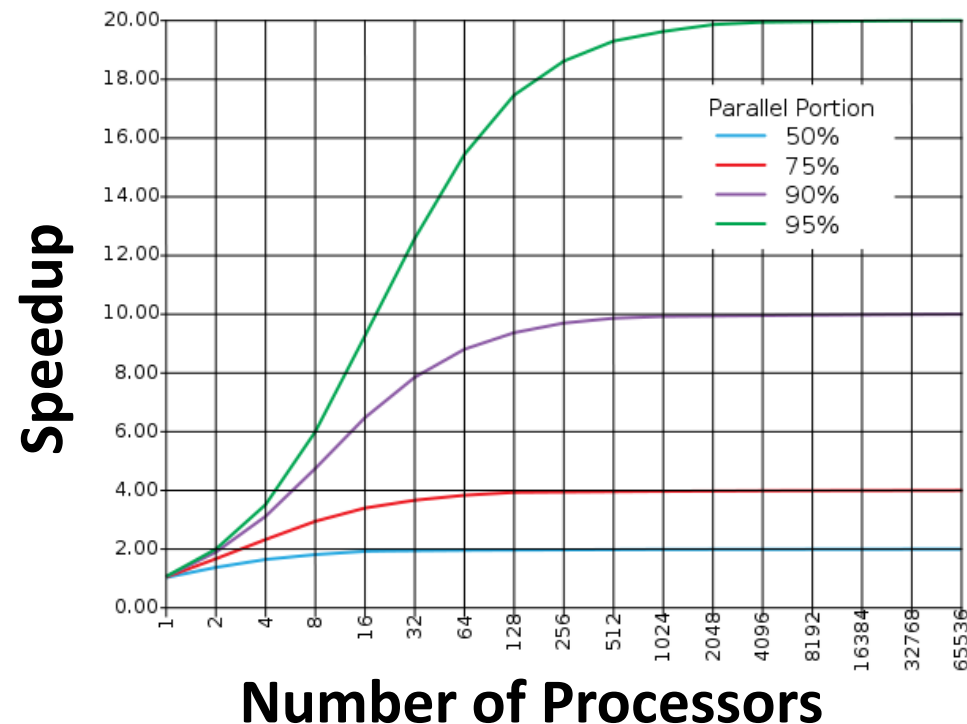
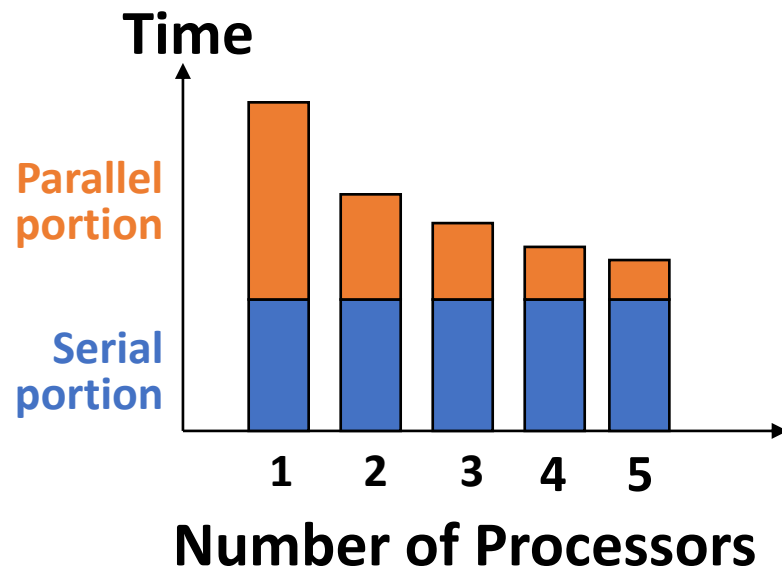
$$\text{Speedup with improvement} = \frac{1}{(0.75) + (0.25/20)} = 1.311$$

**Nowhere near
20x speedup!**



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



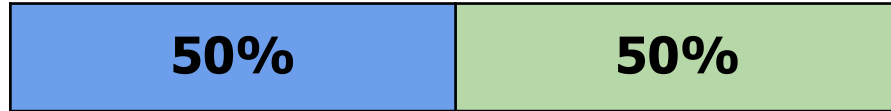
Amdahl's (heartbreaking) Law (in words)

- 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

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

Break + Question

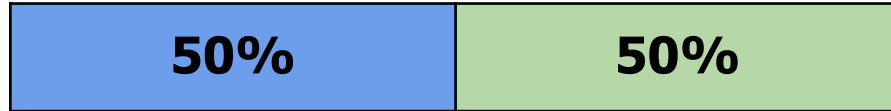
$$\text{Speedup with improvement} = \frac{1}{(1 - F) + (F/S)}$$



- 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

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?

$$\text{Speedup} = 1 / [(1 - F) + (F/S)]$$

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

$$S = 0.5 / ((1/2) - 0.5) = \infty$$

(A) 10

(B) 20

(C) 100

(D) None of the above

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

Outline

- Threads
- Need for Parallelism
- Processor Concurrency
 - Instruction-level parallelism
 - Task parallelism
 - Interrupts
- Concurrency Challenges
 - Amdahl's Law