Lecture 03: Concurrency Sources and Challenges

CS343 – Operating Systems

Branden Ghena – Fall 2020

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

Northwestern

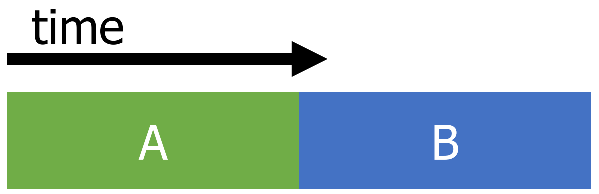
Today’s Goals

- Describe where and why concurrency and parallelism are involved in computing.
- Be disappointed by performance limits on concurrency.
- Understand purpose and challenges of interrupts and signals.
- Introduce concept of data races as a concurrency problem.

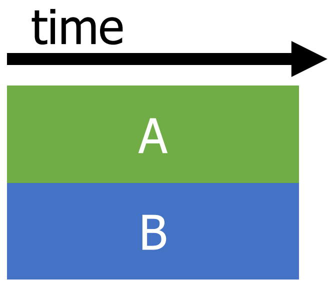
Parallelism versus Concurrency

Two processes A and B

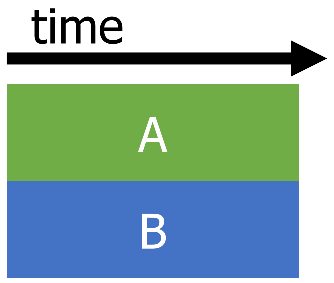
Serial execution



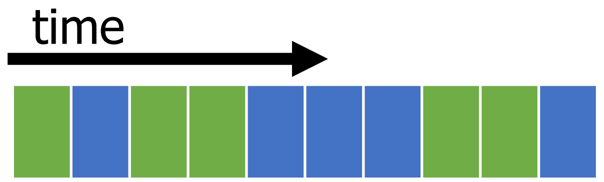
Parallel execution



Concurrent execution



OR

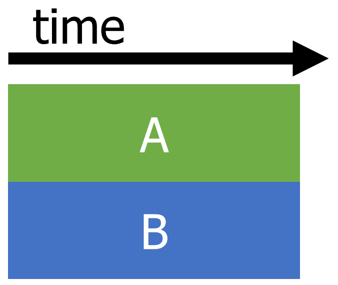


Parallelism versus Concurrency

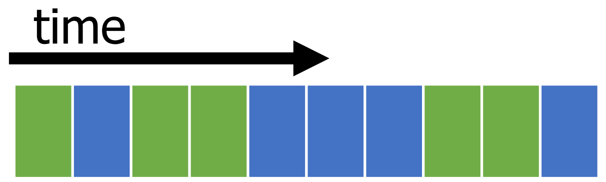
- Parallelism
  - Two things happen strictly simultaneously

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

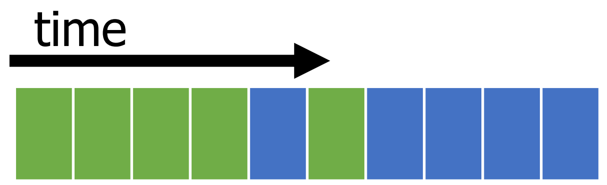
  - Concurrent execution occurs whenever two processes are both active



OR



OR



Outline

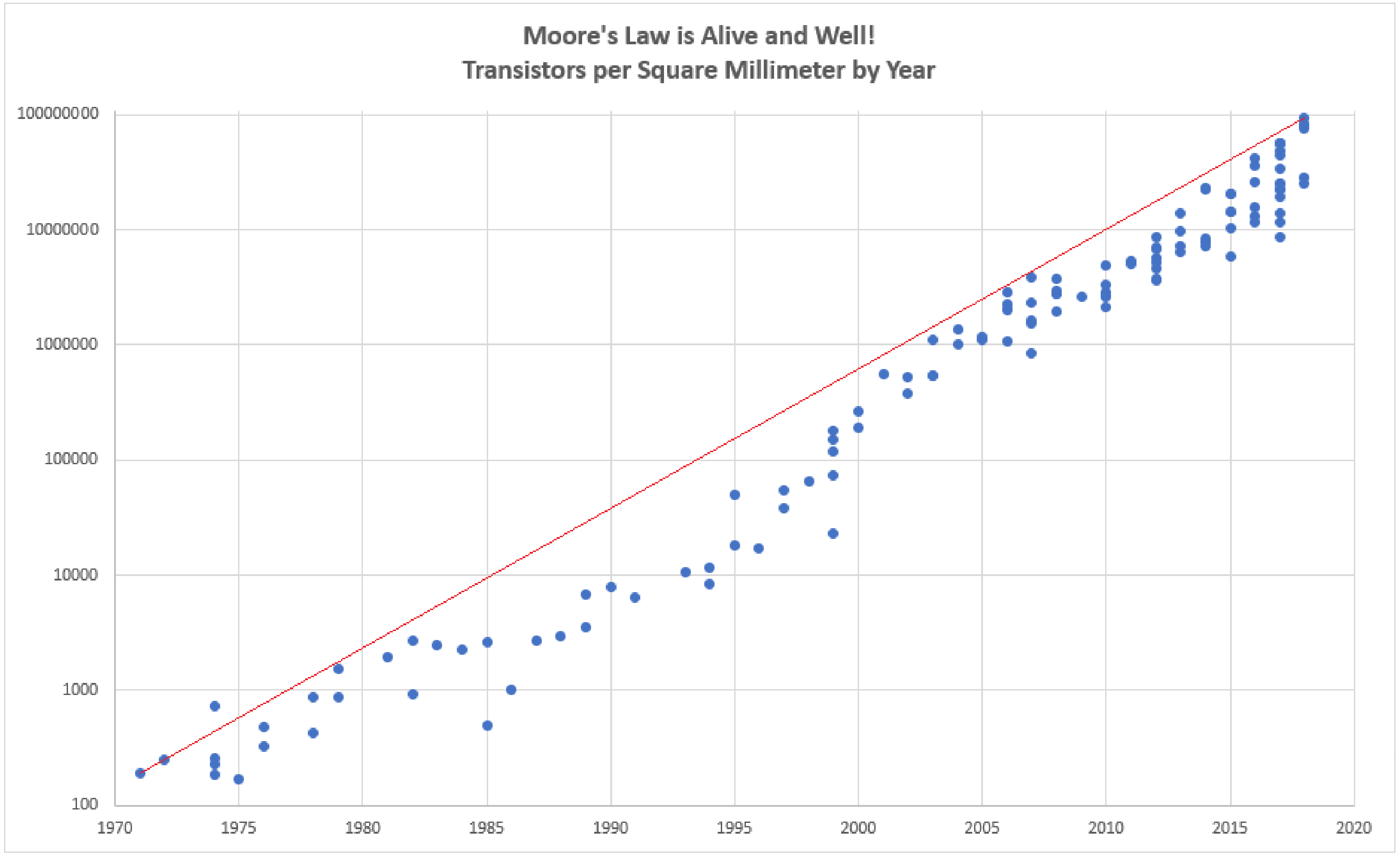
- Performance through concurrency
- Concurrency introduced by the processor
- Amdahl’s Law – limits on performance
- Interrupts and Signals
- A problem with concurrency: data races

Outline

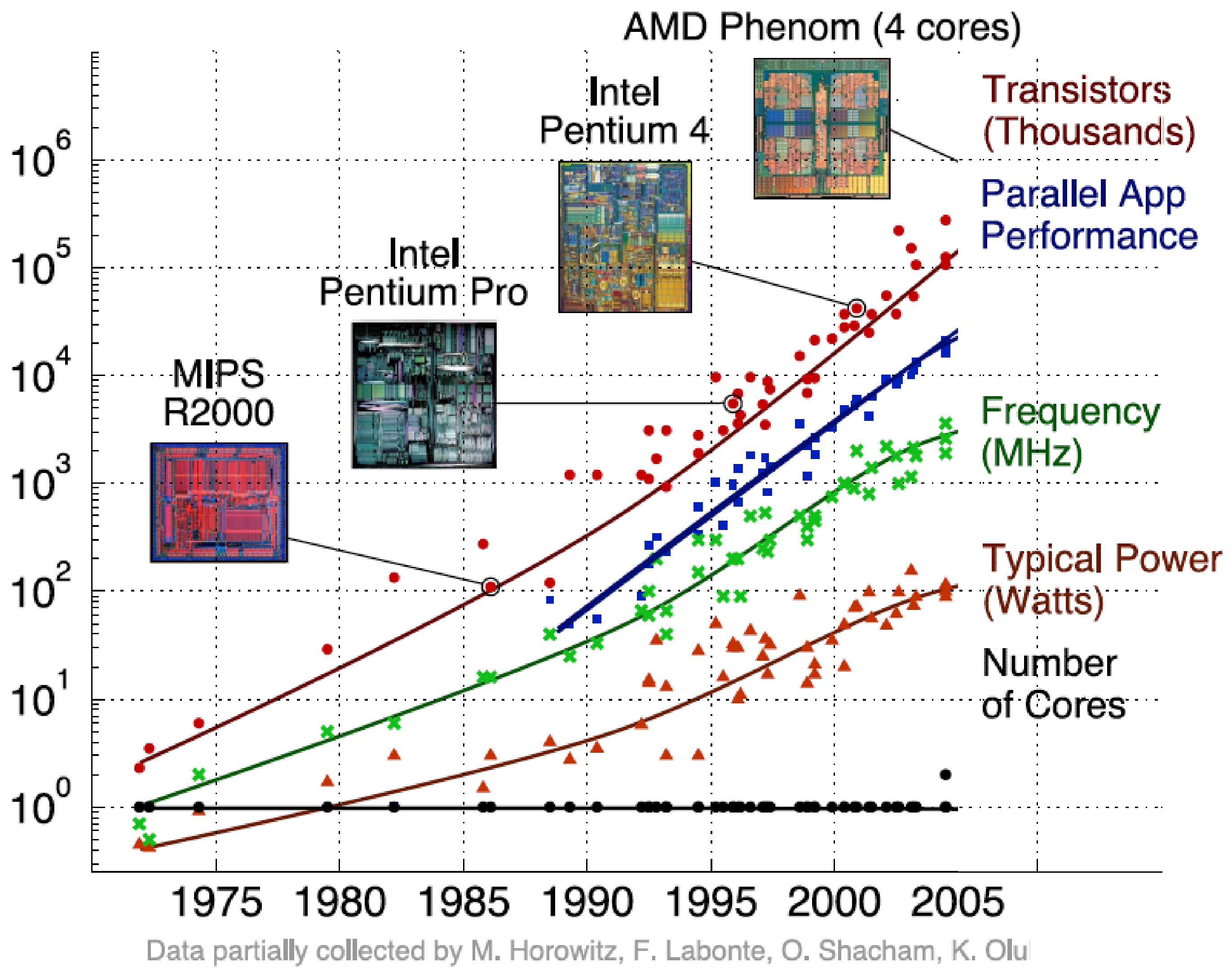
- **Performance through concurrency**
- Concurrency introduced by the processor
- Amdahl’s Law – limits on performance
- Interrupts and Signals
- A problem with concurrency: data races

Moore’s Law

“Every two years, the number of transistors on a chip of a fixed size doubles”



Processors kept getting faster too



Denard Scaling

- Moore’s Law corollary: As transistors get smaller, the power density stays the same.
  - If Moore’s Law holds true, we also get a doubling of “performance per watt” every two years!
  - *Manufacturers could raise the clock frequency between generations without more power consumption*

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

You need to double the speed of Excel in two years time or you’re fired.

What do you do?

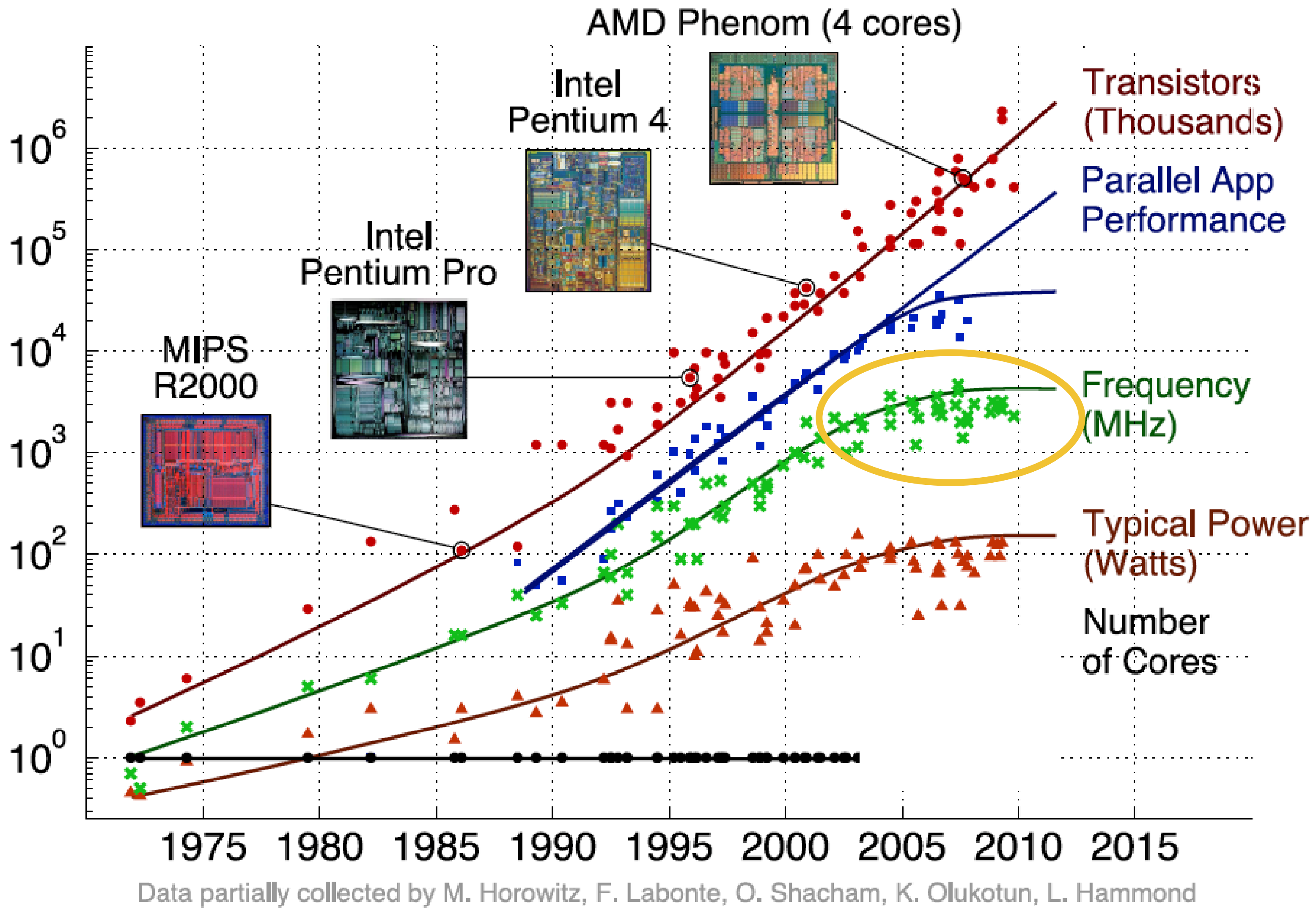
It’s the mid 1990s and you work at Microsoft.

You need to double the speed of Excel in two years time or you’re fired.

What do you do?

Take a vacation

Then they stopped getting faster



~2006: Leakage current becomes significant

Now smaller transistors don't mean lower power

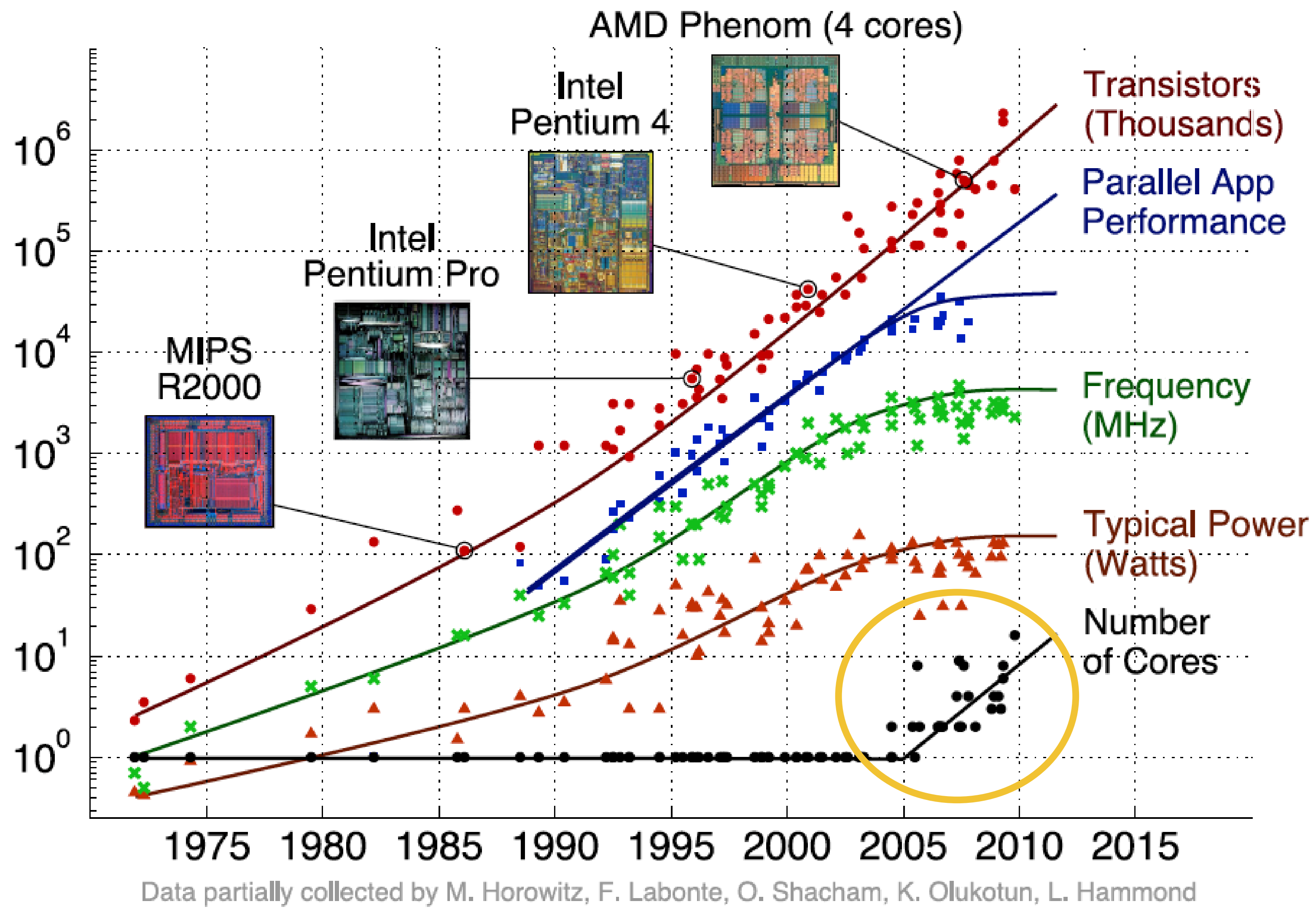
So… now what?

In summary:

- We can't make transistors faster due to current leakage,
- and because of that, *we can't reliably make performance better by waiting for clock speeds to increase,*
- but we have literally billions of transistors available to use.

How do we continue to get better performing computation?

Exploit parallelism!



In reality the cause-and-effect of parallelism isn't so simple.

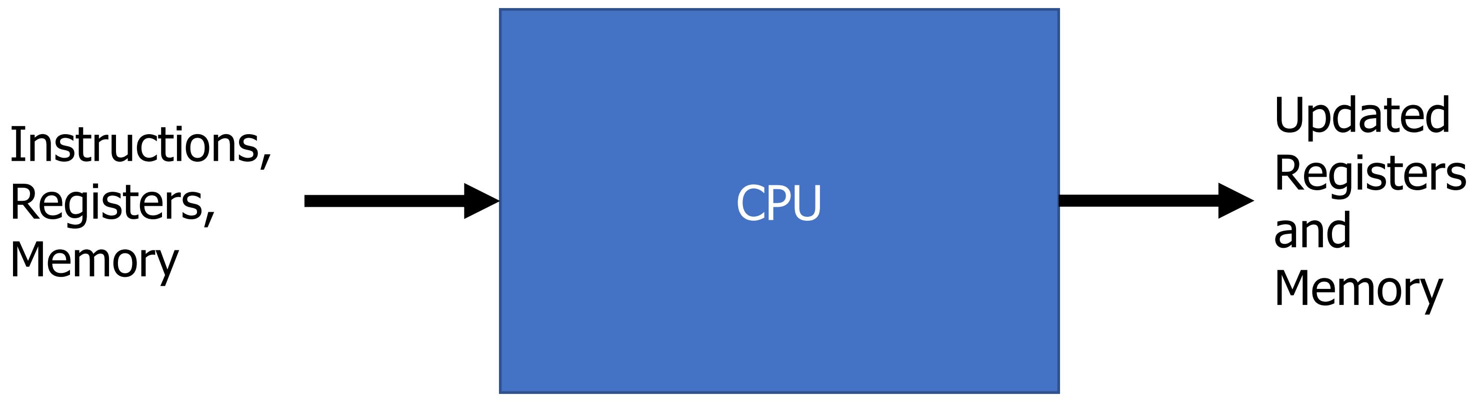
Key points:

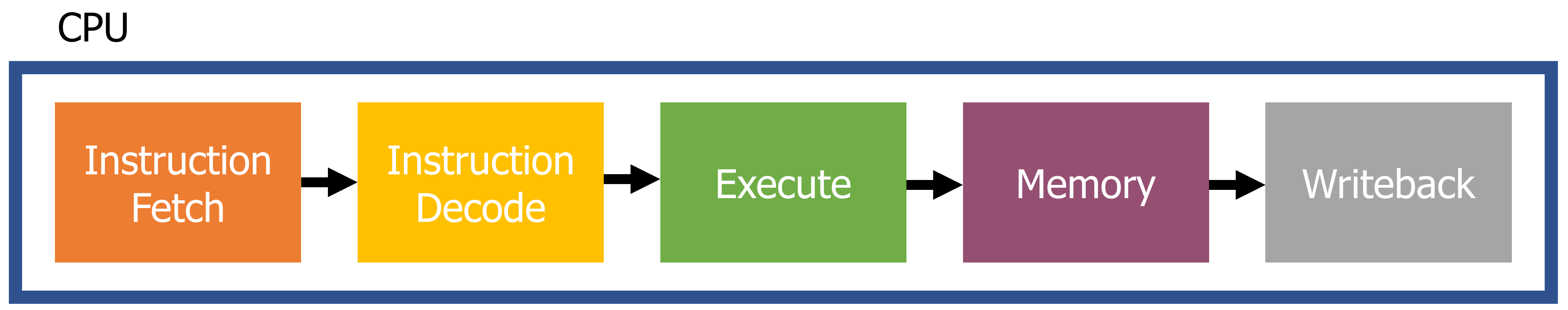
- we can't just increase frequency
- we do have a lot of transistors available
- parallelism is one approach to improve performance again

Outline

- Performance through concurrency
- **Concurrency introduced by the processor**
- Amdahl’s Law – limits on performance
- Interrupts and Signals
- A problem with concurrency: data races

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.

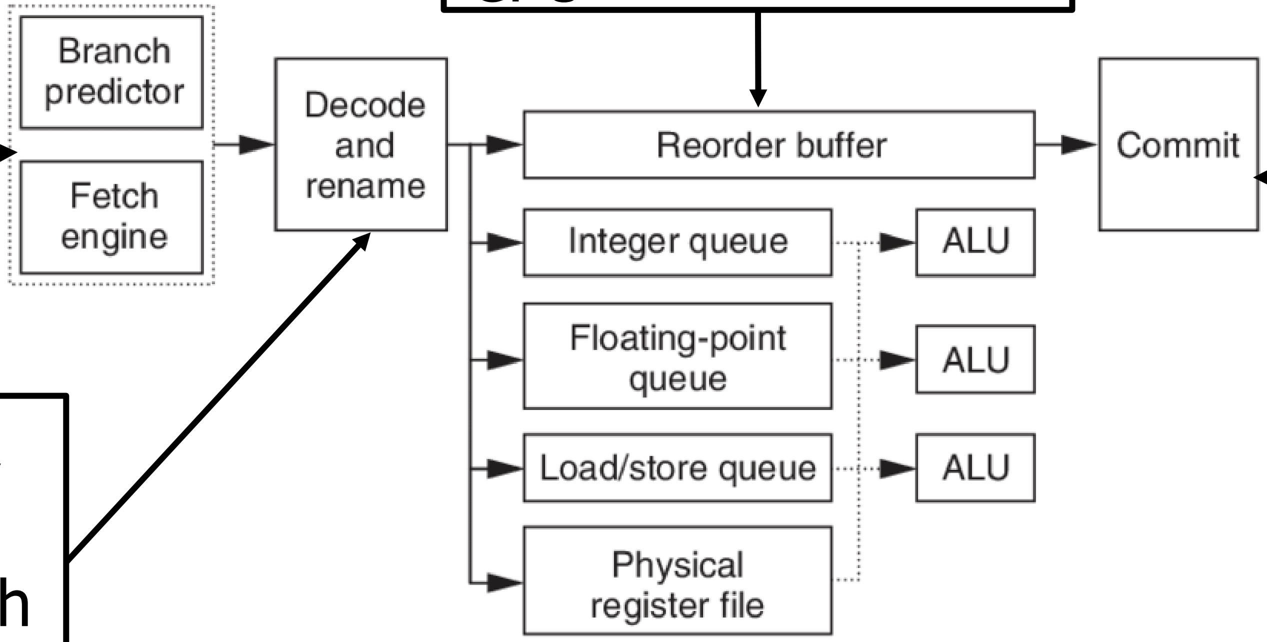
Out-of-order machines

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

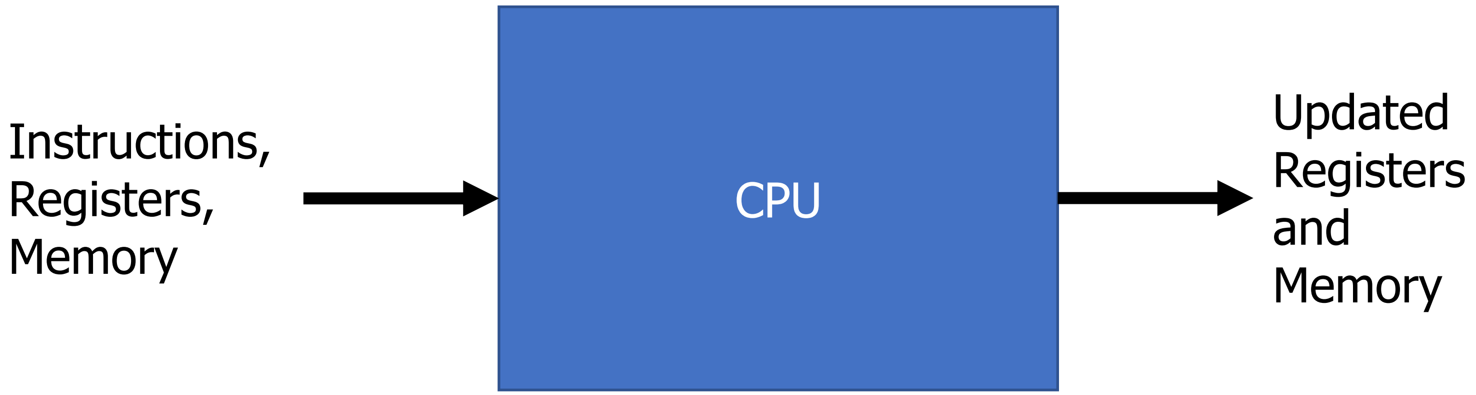


Generally: looks for *independent* instructions it can execute early

Out-of-order processors obey normal execution results

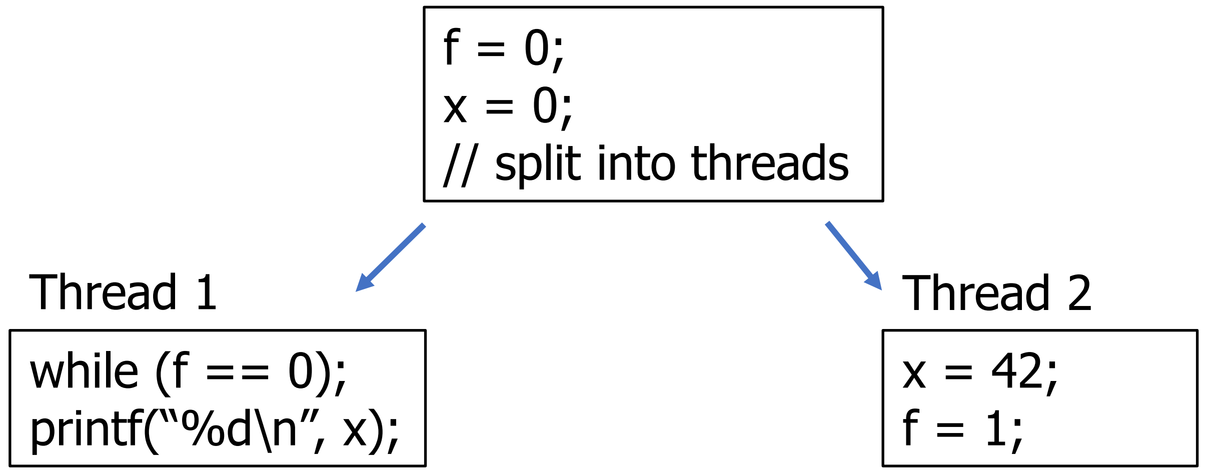
- Initial thoughts on out-of-order execution
  - 😱
  - The processor could be executing my program in order it feels like?!!
  - How do I possibly reason about anything?

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



Multiple processes might rely on memory ordering

- The processor can’t account for multiple processes 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);
```

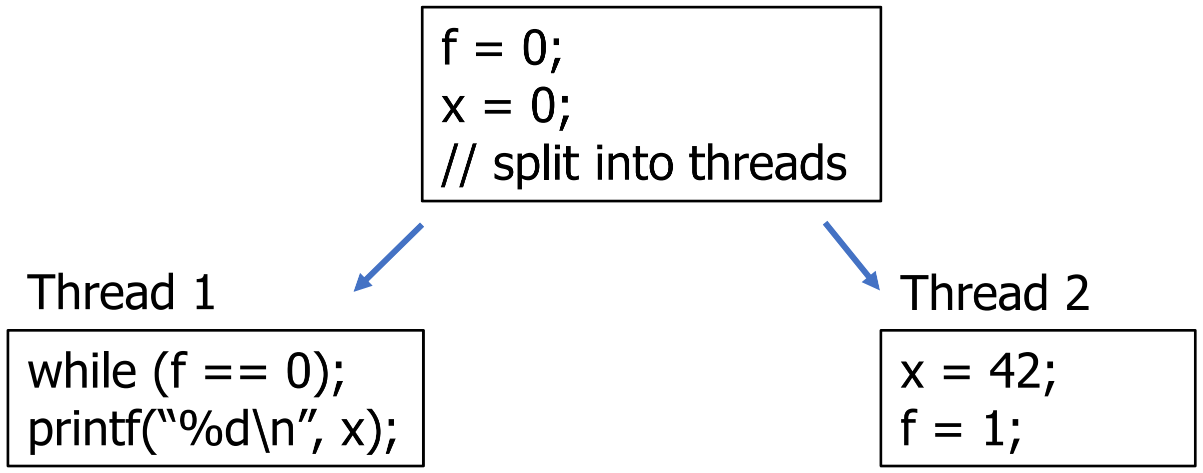
Thread 2

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

- What will Thread 1 print?

Multiple processes might rely on memory ordering

- The processor can't account for multiple processes 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

How else do processors employ concurrency?

Goal: Make computer faster by performing multiple tasks

Solutions:

1. Use multiple cores to run multiple tasks in parallel
2. Run multiple tasks on a single core concurrently

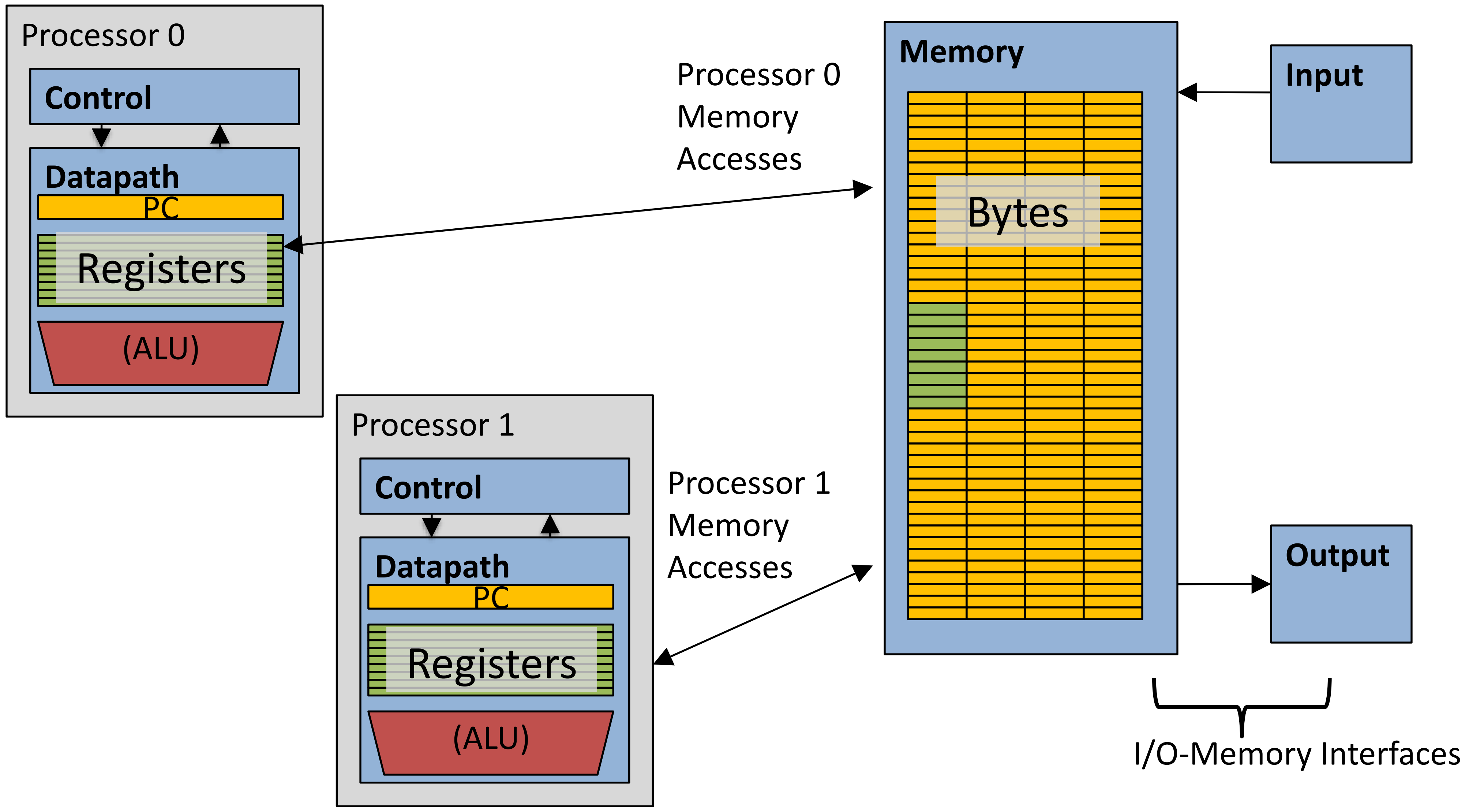
How else do processors employ concurrency?

Goal: Make computer faster by performing multiple tasks

Solutions:

1. **Use multiple cores to run multiple tasks in parallel**
2. Run multiple tasks on a single core concurrently

Multiprocessor Systems (in pictures)



Multiprocessor Systems (in words)

- A computer system with at least 2 processors or *cores*
    - Each core has its own PC and registers
    - Each core executes independent instruction streams
    - Processors share the same system memory
        - But use different parts of it
    - Communication possible through loads and stores to a common location

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

Multiprocessor Example

Run Chrome and Minecraft 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

How else do processors employ concurrency?

Goal: Make computer faster by performing multiple tasks

Solutions:

1. Use multiple cores to run multiple tasks in parallel
2. **Run multiple tasks on a single core concurrently**

Multithreading processors

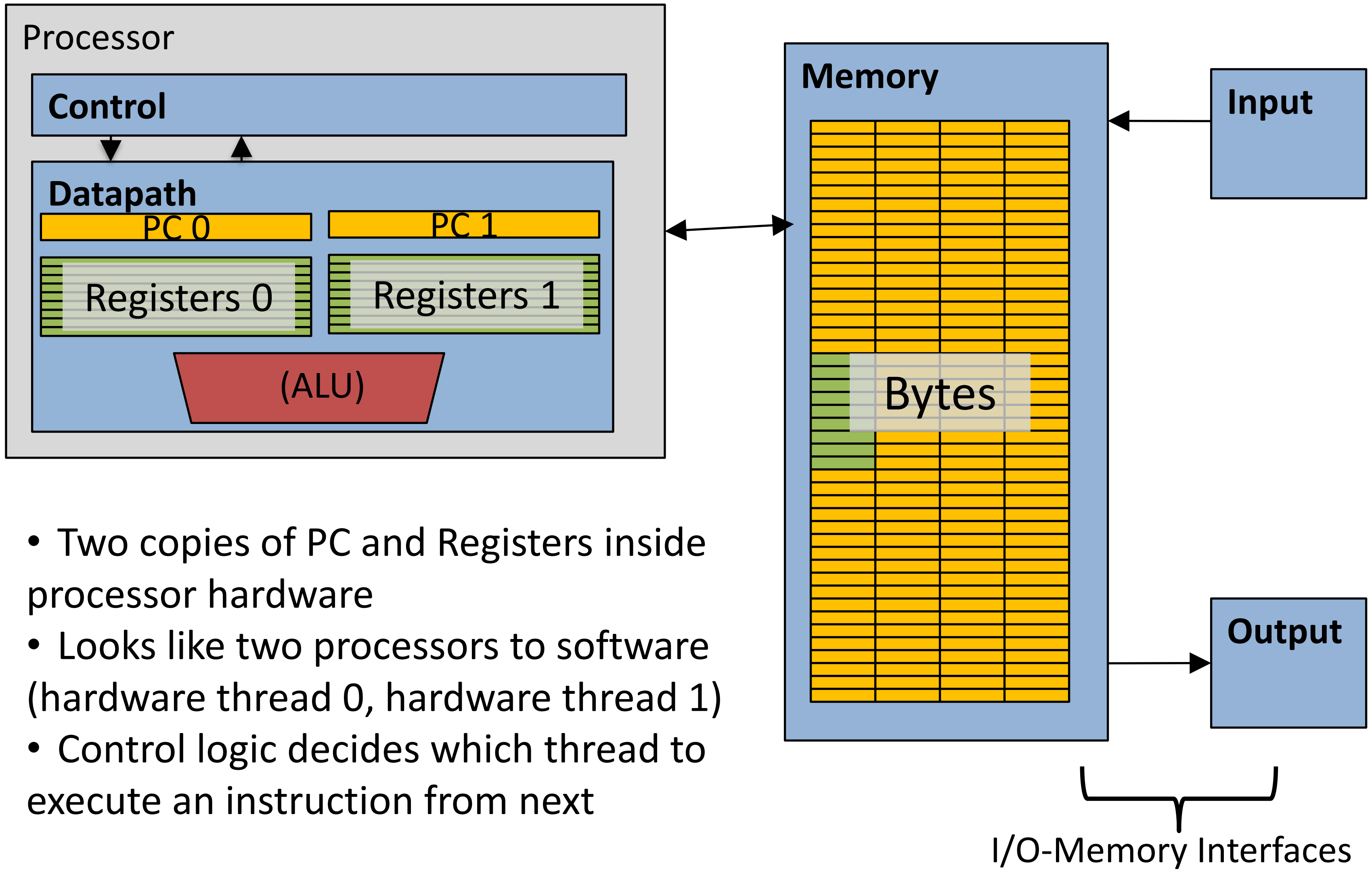
Basic idea: Processor resources are expensive and should not be left idle

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

Hardware support for multithreading

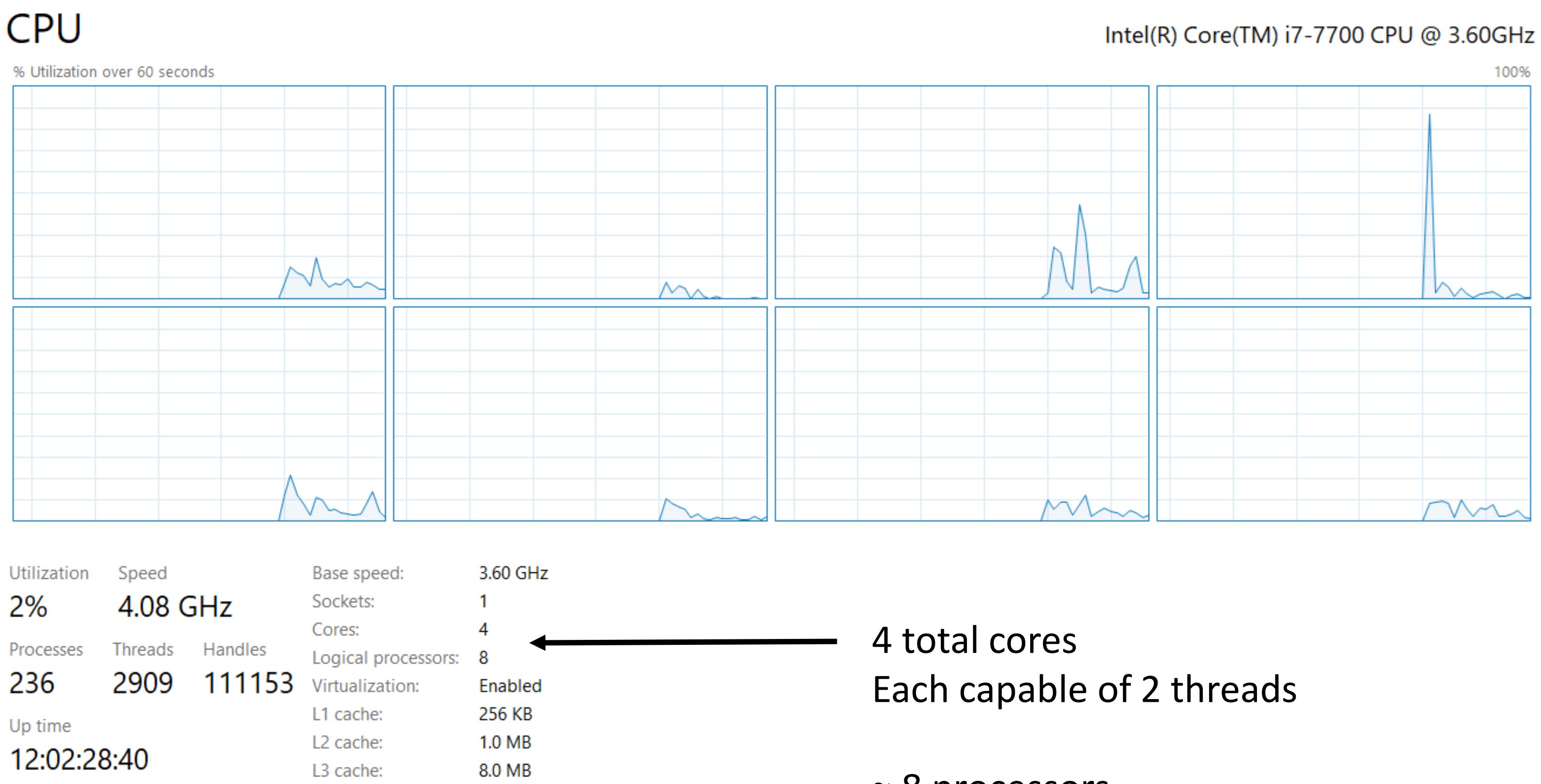


- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

Multithreading versus Multicore

- Multithreading => Better utilization
  - ≈5% more hardware for ≈1.3x better performance?
  - Gets to share ALUs, caches, memory controller
- Multicore => Duplicate processors
  - ≈50% more hardware for ≈2x better performance?
  - Share some caches (L2 cache, L3 cache), memory controller
- Modern machines do both
  - multiple cores with multiple threads per core

My desktop computer


CPU
Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
% Utilization over 60 seconds
100%

Utilization 2%
Speed 4.08 GHz
Processes 236
Threads 2909
Handles 111153
Up time 12:02:28:40

Base speed:	3.60 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	8.0 MB

4 total cores
Each capable of 2 threads

≈ 8 processors

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

Back up to the OS perspective

- Modern operating systems must manage concurrency
    - Both parallel operation and interleaving operations

- Concurrency is worth it
    - Performance gains are the reason

Outline

- Performance through concurrency
- Concurrency introduced by the processor
- **Amdahl’s Law – limits on performance**
- Interrupts and Signals
- A problem with concurrency: 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?

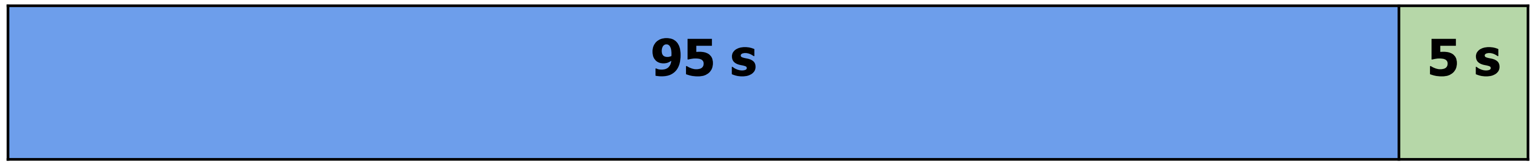
Challenges to concurrency

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?

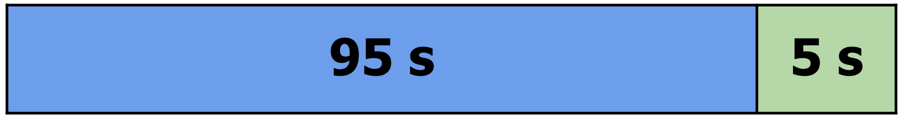
Speedup Example



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

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

Non-speed-up part

Speed-up 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

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

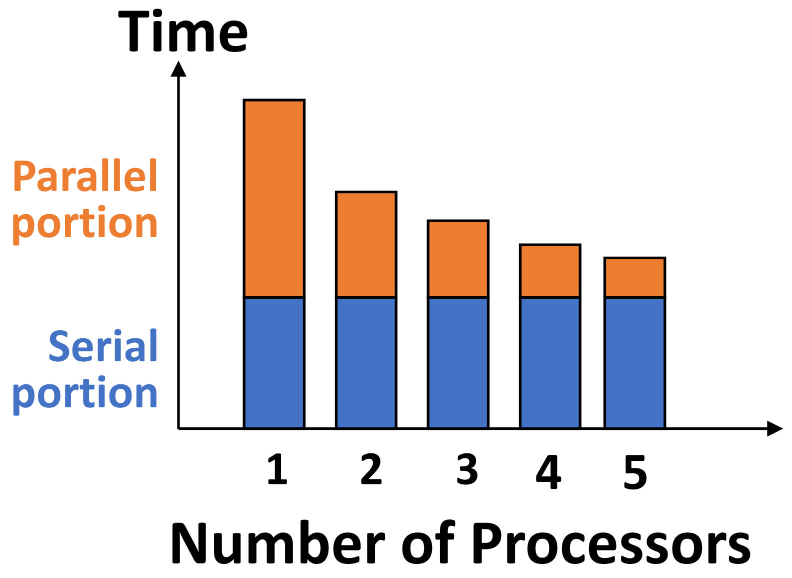
Nowhere near 20x speedup!

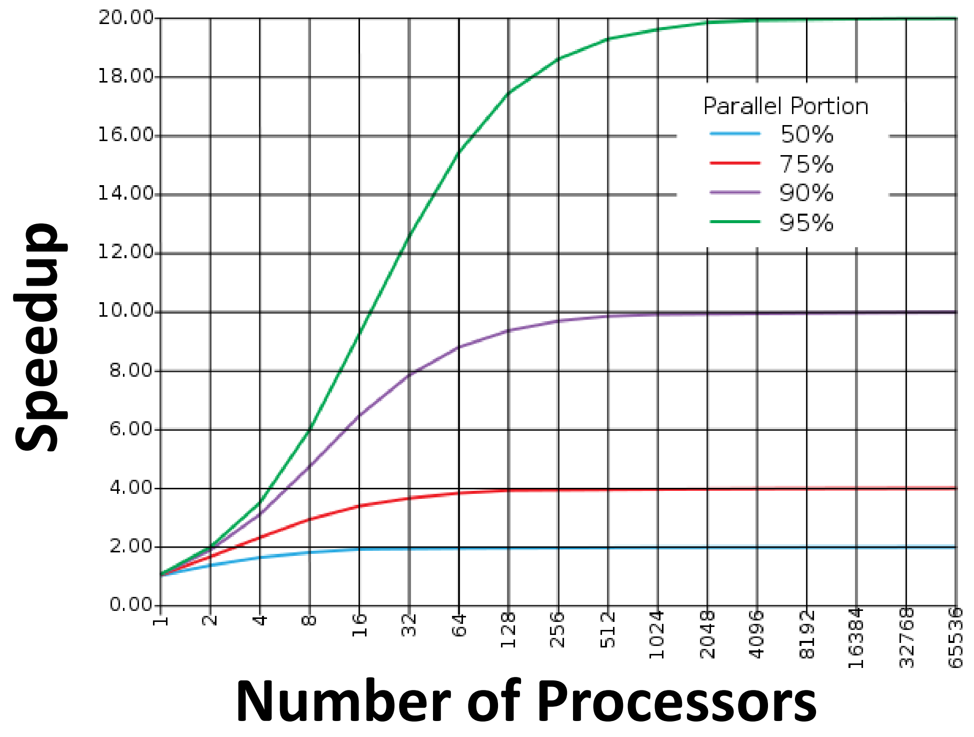
- What if it’s usable 25% of the time?

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

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!





Amdahl’s (heartbreaking) Law (in words)

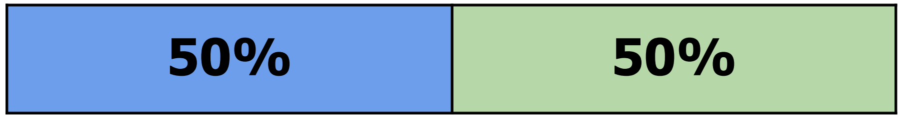
- Amdahl’s Law tells us that to achieve linear speedup with more processors:
    - *none* of the original computation can be serial (non-parallelizable)

- 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

Speedup = 1/(.001 + .999/100) = 90.99

Check your understanding

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

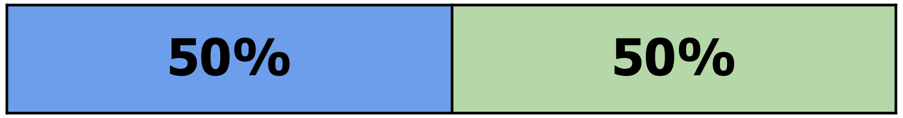


- Suppose a program spends 50% of its time in a square root routine.
- How much must you speed up square root to make the program run 2x faster?

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

Check your understanding

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]

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

- Performance through concurrency
- Concurrency introduced by the processor
- Amdahl’s Law – limits on performance
- **Interrupts and Signals**
- A problem with concurrency: data races

Where else does concurrency come from?

- Processors introduce it for performance reasons by running multiple processes and threads
- Interactions with the outside world introduce it because events occur whenever they feel like it
  - Network request arriving
  - User presses a key
  - Motion sensor triggers
- Also, we need some way to deal with errors the occur when executing instructions
  - No pathway for returning an error from an instruction

Interrupts

A way for the CPU to be, well, *interrupted*.

- CPU hardware switches to privileged mode
  - Now any instruction can be executed, including privileged ones.
- Execution jumps to a predefined location
  - Handler specified in the CPU’s interrupt vector table
  - Lets the kernel deal with whatever the event was
- Used to support asynchronous I/O
  - Lets a hardware device tell the CPU that some data is ready
  - Remember that a disk operation is millions of times slower than an *add*.
- CPU has an electrical pin for hardware interrupts.
- There is also an instruction for *software* interrupts (like traps!)

Interrupt Vector Table

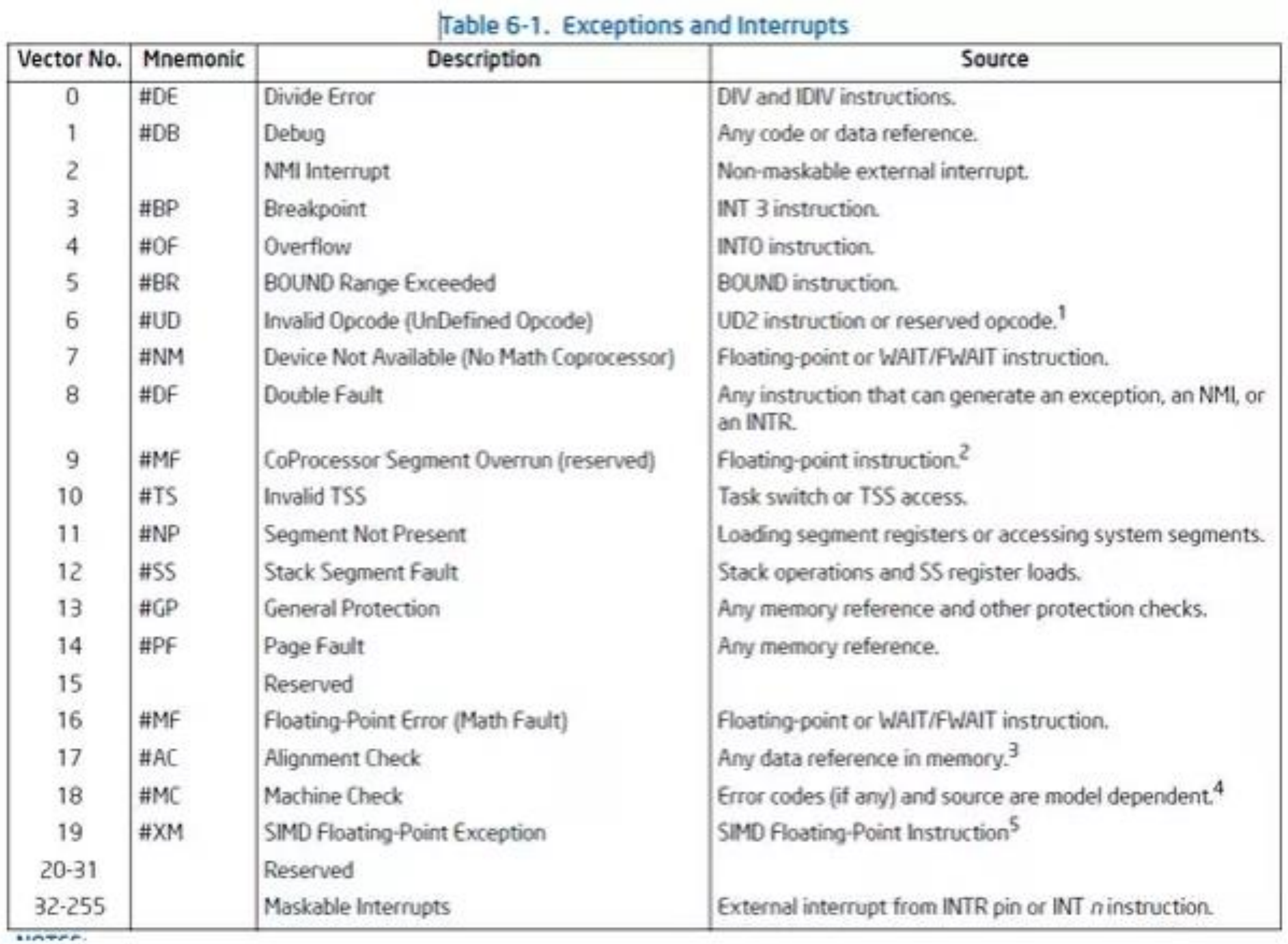


Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Table actually lives in memory somewhere, with function pointers for each vector number

```rust
match interrupt {
    nvic::ASTALARM => ast::AST.handle_interrupt(),

    nvic::USART0 => usart::USART0.handle_interrupt(),
    nvic::USART1 => usart::USART1.handle_interrupt(),
    nvic::USART2 => usart::USART2.handle_interrupt(),
    nvic::USART3 => usart::USART3.handle_interrupt(),

    nvic::PDCA0 => dma::DMA_CHANNELS[0].handle_interrupt(),
    nvic::PDCA1 => dma::DMA_CHANNELS[1].handle_interrupt(),
    nvic::PDCA2 => dma::DMA_CHANNELS[2].handle_interrupt(),
    nvic::PDCA3 => dma::DMA_CHANNELS[3].handle_interrupt(),
    nvic::PDCA4 => dma::DMA_CHANNELS[4].handle_interrupt(),
    nvic::PDCA5 => dma::DMA_CHANNELS[5].handle_interrupt(),
    nvic::PDCA6 => dma::DMA_CHANNELS[6].handle_interrupt(),
    nvic::PDCA7 => dma::DMA_CHANNELS[7].handle_interrupt(),
    nvic::PDCA8 => dma::DMA_CHANNELS[8].handle_interrupt(),
    nvic::PDCA9 => dma::DMA_CHANNELS[9].handle_interrupt(),
    nvic::PDCA10 => dma::DMA_CHANNELS[10].handle_interrupt(),
    nvic::PDCA11 => dma::DMA_CHANNELS[11].handle_interrupt(),
    nvic::PDCA12 => dma::DMA_CHANNELS[12].handle_interrupt(),
    nvic::PDCA13 => dma::DMA_CHANNELS[13].handle_interrupt(),
    nvic::PDCA14 => dma::DMA_CHANNELS[14].handle_interrupt(),
    nvic::PDCA15 => dma::DMA_CHANNELS[15].handle_interrupt(),
```

Example from Tock for SAM4L chip (in Rust)

Interrupt Vector Table

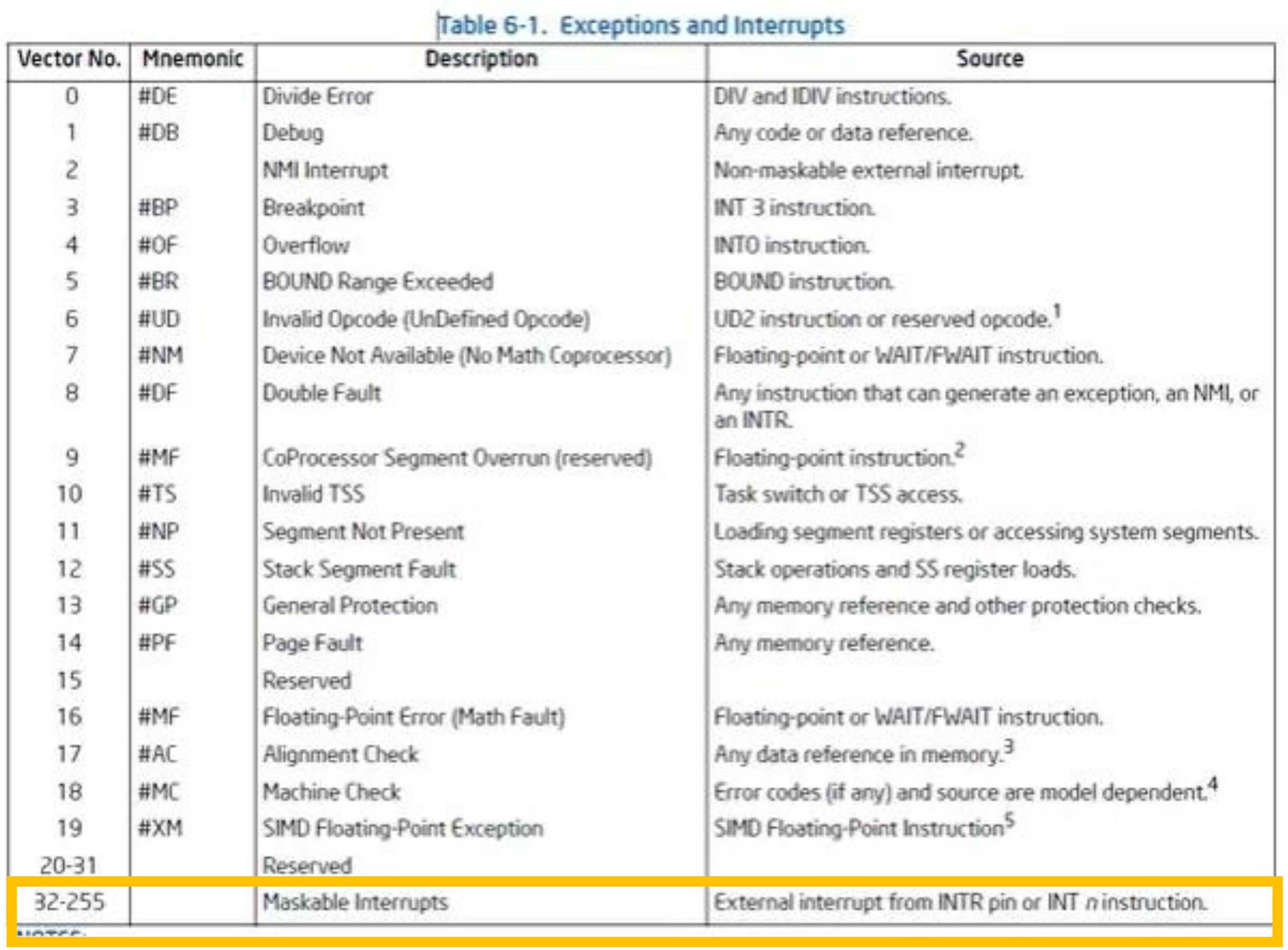


Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Table actually lives in memory somewhere, with function pointers for each vector number

```
match interrupt {
    nvic::ASTALARM => ast::AST.handle_interrupt(),

    nvic::USART0 => usart::USART0.handle_interrupt(),
    nvic::USART1 => usart::USART1.handle_interrupt(),
    nvic::USART2 => usart::USART2.handle_interrupt(),
    nvic::USART3 => usart::USART3.handle_interrupt(),

    nvic::PDCA0 => dma::DMA_CHANNELS[0].handle_interrupt(),
    nvic::PDCA1 => dma::DMA_CHANNELS[1].handle_interrupt(),
    nvic::PDCA2 => dma::DMA_CHANNELS[2].handle_interrupt(),
    nvic::PDCA3 => dma::DMA_CHANNELS[3].handle_interrupt(),
    nvic::PDCA4 => dma::DMA_CHANNELS[4].handle_interrupt(),
    nvic::PDCA5 => dma::DMA_CHANNELS[5].handle_interrupt(),
    nvic::PDCA6 => dma::DMA_CHANNELS[6].handle_interrupt(),
    nvic::PDCA7 => dma::DMA_CHANNELS[7].handle_interrupt(),
    nvic::PDCA8 => dma::DMA_CHANNELS[8].handle_interrupt(),
    nvic::PDCA9 => dma::DMA_CHANNELS[9].handle_interrupt(),
    nvic::PDCA10 => dma::DMA_CHANNELS[10].handle_interrupt(),
    nvic::PDCA11 => dma::DMA_CHANNELS[11].handle_interrupt(),
    nvic::PDCA12 => dma::DMA_CHANNELS[12].handle_interrupt(),
    nvic::PDCA13 => dma::DMA_CHANNELS[13].handle_interrupt(),
    nvic::PDCA14 => dma::DMA_CHANNELS[14].handle_interrupt(),
    nvic::PDCA15 => dma::DMA_CHANNELS[15].handle_interrupt(),
```

Example from Tock for SAM4L chip (in Rust)

Differences from traps

- 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.

- Interrupts can happen *whenever.*
  - This can get extremely complicated on modern systems with out-of-order execution, multiple cores and threads, and caches

Interrupt handlers

- Interrupt context
  - Can’t just enter the kernel like we did with system calls
  - Interrupt could have occurred while we were in the kernel

- 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
  - Processes are interrupted, but otherwise not normally involved

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) 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	      ...
```

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) 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 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

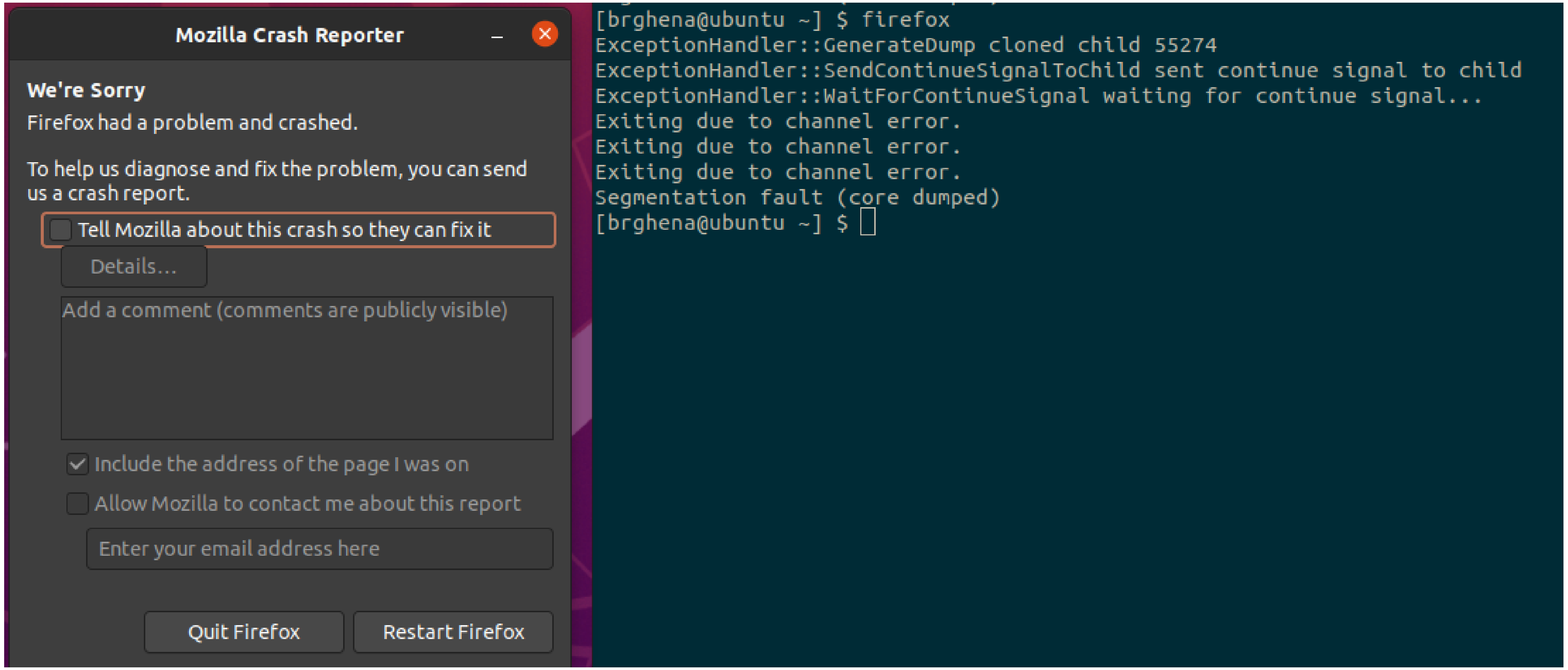
- Programs can register a function to handle individual signals
  - `signal(int sig, sighandler_t handler);`

- What are you supposed to do about it?
  - Do some *quick* processing to handle it
    - Be careful, not all functions are safe to call here (re-entrant functions only)
  - Reset the process and try again
  - Quit the process (default handler)

Signals Examples

Examples: sending a signal

> kill -11 *pid*


Mozilla Crash Reporter
We're Sorry
Firefox had a problem and crashed.
To help us diagnose and fix the problem, you can send us a crash report.
Tell Mozilla about this crash so they can fix it
Details...
Add a comment (comments are publicly visible)
Include the address of the page I was on
Allow Mozilla to contact me about this report
Enter your email address here
Quit Firefox
Restart Firefox

```
[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 ~] $
```

Example: catching a signal

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
```

```
void sighandler (int signum) {
   printf("HA HA You can't kill me!\n");
}
int main (void) {
   signal(SIGINT, sighandler);
   printf("Starting\n");
   while(1) {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
   return(0);
}
```

Example: catching a segfault

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
```

```
int* pointer = 0x00000000;

void sighandler (int signum) {

   printf("Oops, that pointer wasn't valid. Let's try a different one\n");

   pointer++;

   printf("About to read from pointer 0x%08lX\n", (long)pointer);

}

int main (void) {

   signal(SIGSEGV, sighandler);

   printf("About to read from pointer 0x%08lX\n", (long)pointer);

   int test = *pointer;

   return(0);

}
```

Systems software needs to deal with concurrency

- Reliable processes need to handle signals
- OS kernels need to handle interrupts

- This time it’s not just about performance
    - It’s also about handling errors
    - And interacting with the outside world

- Challenges are similar
    - Processor concurrency: shared state between multiple processes/threads
    - Interrupts/Signals: shared state between a process and interrupt context

Outline

- Performance through concurrency
- Concurrency introduced by the processor
- Amdahl’s Law – limits on performance
- Interrupts and Signals
- **A problem with concurrency: 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`

Thread 1:

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

Thread 2:

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

Assuming "count" is in memory location 0x8049a1c

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

These instructions could be interleaved in any way.

Data race example

Assuming “count” is in memory location 0x8049a1c

Time ↓

Thread 1	Thread 2
`mov 0x8049a1c, %eax`	
`add $0x1, %eax`	
`mov %eax, 0x8049a1c`	
	`mov 0x8049a1c, %eax`
	`add $0x1, %eax`
	`mov %eax, 0x8049a1c`

Final value of count: 2

Thread 1	Thread 2
`mov 0x8049a1c, %eax`	
	`mov 0x8049a1c, %eax`
	`add $0x1, %eax`
	`mov %eax, 0x8049a1c`
`add $0x1, %eax`	
`mov %eax, 0x8049a1c`	

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 += 1;
}
```

Thread 2:

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

Thread 3:

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

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 += 1;
}
```

Thread 2:

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

Thread 3:

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

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

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

Outline

- Performance through concurrency
- Concurrency introduced by the processor
- Amdahl’s Law – limits on performance
- Interrupts and Signals
- A problem with concurrency: data races