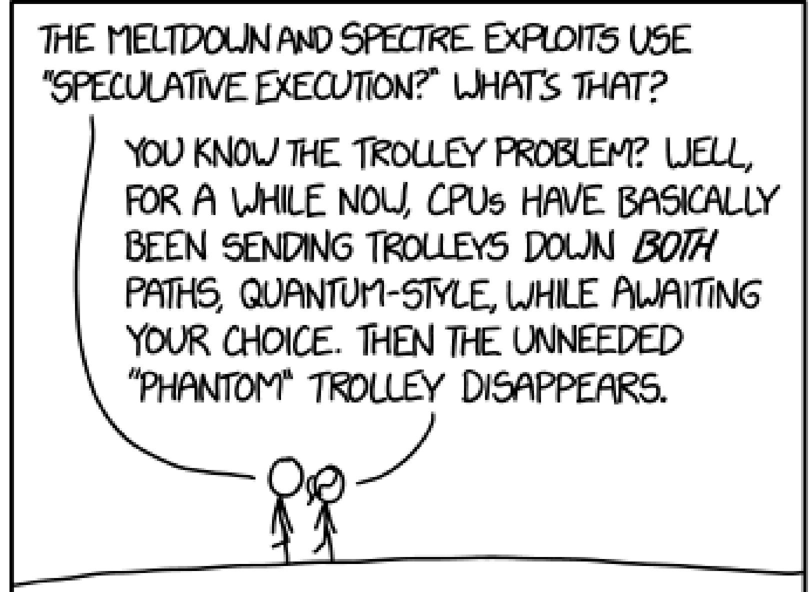
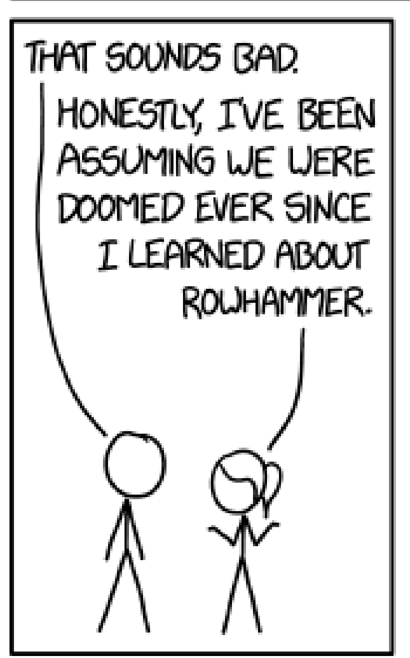
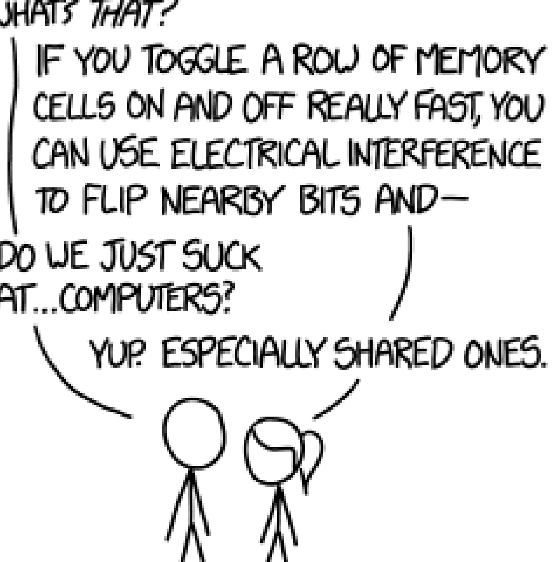
IMPLEMENTING A SEQUENTIAL CPU

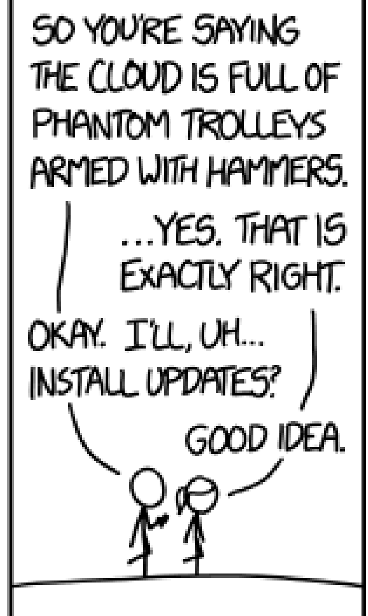
Unit 1

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder









https://xkcd.com/1938/

LEARNING GOALS

- Given the notation from the textbook, describe in plain English and at a high level what an instruction does
- Given a Y86 instruction and its description convert that to the short hand form from that describes what the instruction does
- Given the compiler's assembly output of a C program map between the C and assembly
- Given a small piece of C convert it to Y86
- Convert from x86-64 to Y86
- Describe in plain English what a small piece of x86-64 or Y86 is doing
- Explain the purpose and use of the Y86 assembler directives
- Given the description of a an instruction format for an ISA translate between assembly and machine language byte encodings and vice versa

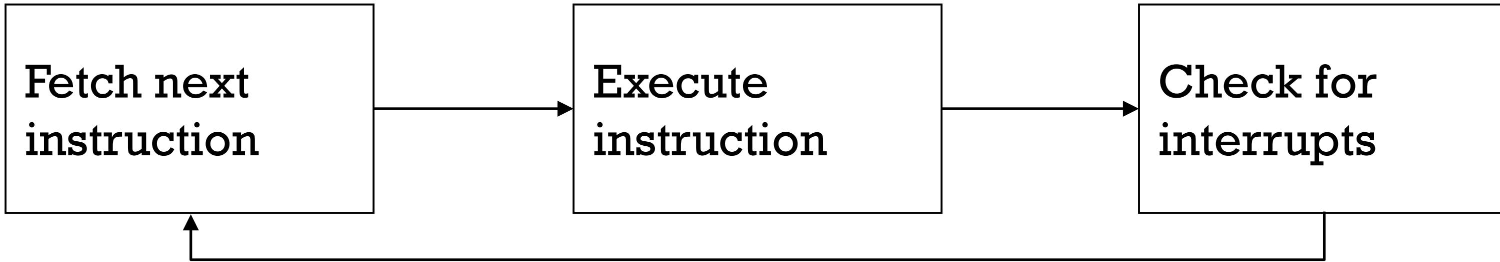
SHORT TERM PLAN

- Look at a new instruction set architecture
  - Goal - understand how it might be implemented
- Study how it is implemented in hardware
  - Goal - Appreciate the implications on program performance and behaviour
- Look at a pipelined implementation
  - Goal - Appreciate the implications on program performance and behaviour

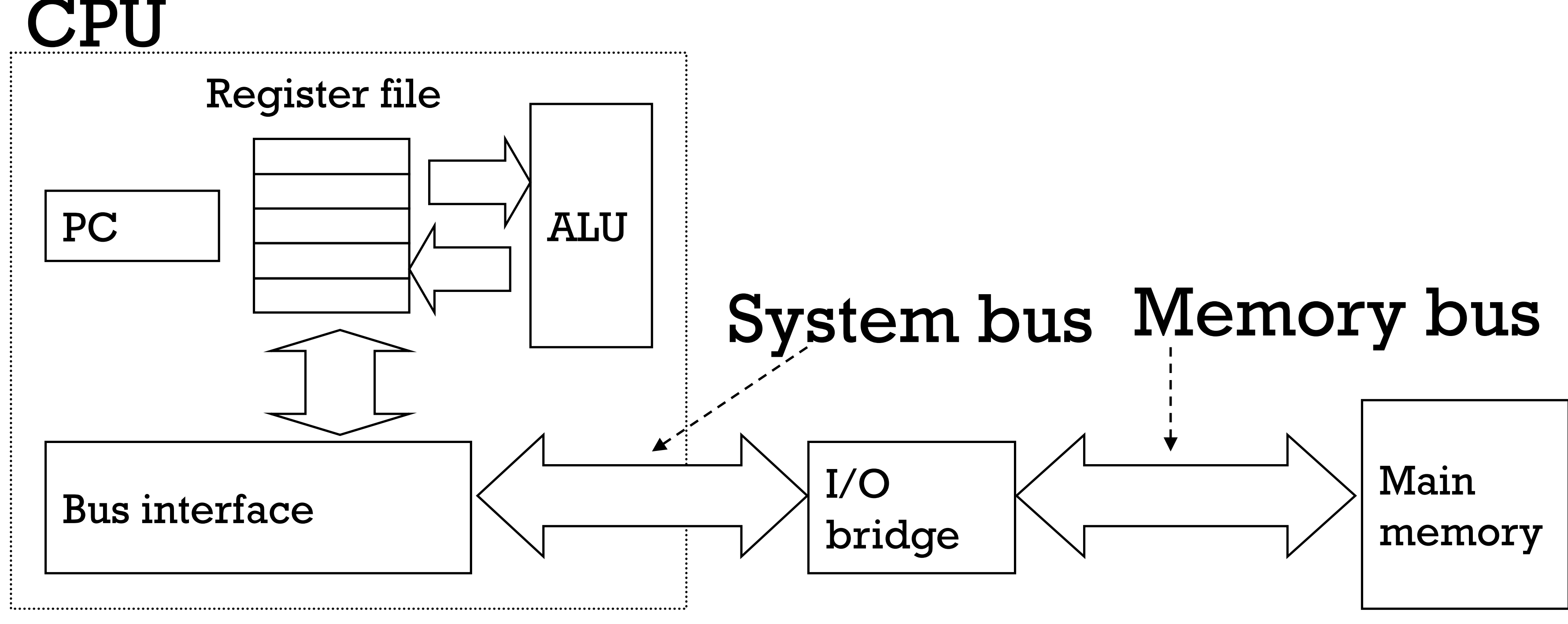
FROM CPEN 211 YOU KNOW

- computers implement an Instruction Set Architecture
  - ISA: Abstract expression of what a machine does
  - Independent of actual implementation
  - Multiple implementations of the same ISA
- compilers translate high-level language programs into sequences of low-level hardware instructions
- CPU hardware executes
  - one instruction at a time – one per cycle (or does it?)
  - cycle is Fetch then Execute
  - instructions are stored in memory
  - execution transforms register and memory contents from one state to another

WHAT DOES THE CPU DO?



FETCH/EXECUTE CYCLE



From *Computer Systems: A Programmer's Perspective*

ISA COMMONALITY

- Most ISAs have in common:
  - Some set of registers to work with
  - Rules for accessing memory
  - A set of instructions for:
    - Manipulating memory
    - Manipulating registers
    - Controlling the flow of instruction execution

INSTRUCTION TYPES

- All ISAs have a few types of instructions
  - Memory based operations (load, store)
  - Register based operations (moves, arithmetic operations)
  - Change program counter (jumps, branches, call, return)
  - Conditional operations (allows loops, if/then/else)
- Some ISAs have instructions that might combine a couple of these operations
  - Example: load from memory and add to register

LEARNING A NEW ASSEMBLY LANGUAGE

- How do you learn a new assembly language?
  - Read a short description
    - http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
  - Read a longer description
    - http://csapp.cs.cmu.edu/public/1e/public/docs/asm64-handout.pdf
    - https://software.intel.com/en-us/node/181178
  - Or use a C compiler and read the assembly language code it produces

X86-64/Y86-64 INTRODUCTION

- Y86-64: invented by the textbook authors as a teaching tool
- A simple subset of the x86-64 (Intel) architecture
  - Less instructions, simpler encoding, simpler addressing modes
- Inspired by RISC (reduced instruction set computer)
- It has
  - 15 general-purpose 64-bit registers
  - 1 program counter (PC)
  - 3 condition codes zero (ZF), sign (SF), overflow (OF)
  - 12 types of instruction

Y86-64 REGISTERS

- Registers (64 bits each):

0	%rax	%rsp	4
1	%rcx	%rbp	5
2	%rdx	%rsi	6
3	%rbx	%rdi	7

8	%r8	%r12	12
9	%r9	%r13	13
10	%r10	%r14	14
11	%r11		

  - Instructions that only need one register use F (15) for the second register.
- Additional flags available: overflow flag (OF), zero flag (ZF), sign flag (SF)
  - on or off depending on result of previous operation
  - only accessible indirectly
- Memory contains 2^{64} addressable bytes
  - all data accesses load/store 64 bit words aligned on an 8 byte boundary

Y86-64 INSTRUCTION TYPES

- register/memory transfer (at most one memory access per instruction)
- arithmetic and register move (no memory access allowed)
- jumps (conditional and unconditional)
- conditional moves (register move only if condition is true)
- stack manipulation
- procedure calls (special kind of jump)
- miscellaneous (halt, nop)

Y86-64 ISA: INSTRUCTIONS

- register/memory transfers:
  - rmmovq rA, D(rB) $M_8[D + R[rB]] \leftarrow R[rA]$
    - Example: rmmovq %rdx, $0x20(%rsi)
  - mrmovq D(rB), rA $R[rA] \leftarrow M_8[D + R[rB]]$
    - Example: mrmovq $0x0A(%rdx), %rsi

Y86-64 ISA: INSTRUCTIONS

- Register manipulation
  - rrmovq rA, rB $R[rB] \leftarrow R[rA]$
  - irmovq V, rB $R[rB] \leftarrow V$
- Arithmetic instructions
  - addq rA, rB $R[rB] \leftarrow R[rB] + R[rA]$
  - subq rA, rB $R[rB] \leftarrow R[rB] - R[rA]$
  - andq rA, rB $R[rB] \leftarrow R[rB] \wedge R[rA]$
  - xorq rA, rB $R[rB] \leftarrow R[rB] \oplus R[rA]$
  - mulq rA, rB $R[rB] \leftarrow R[rB] * R[rA]$
  - divq rA, rB $R[rB] \leftarrow R[rB] / R[rA]$
  - modq rA, rB $R[rB] \leftarrow R[rB] \% R[rA]$

Extensions, not in the book

Y86-64 ISA: INSTRUCTIONS

- Unconditional jump
  - jmp Dest PC $\leftarrow$ Dest
- Conditional jumps
  - jle Dest PC $\leftarrow$ Dest if last result ≤ 0
  - jl Dest PC $\leftarrow$ Dest if last result < 0
  - je Dest PC $\leftarrow$ Dest if last result $= 0$
  - jne Dest PC $\leftarrow$ Dest if last result $\neq 0$
  - jge Dest PC $\leftarrow$ Dest if last result ≥ 0
  - jg Dest PC $\leftarrow$ Dest if last result > 0

Y86-64 ISA: Instructions

- Conditional moves
  - cmovle rA, rB — R[rB] ← R[rA] if last result ≤ 0
  - cmovl rA, rB — R[rB] ← R[rA] if last result < 0
  - cmove rA, rB — R[rB] ← R[rA] if last result = 0
  - cmovne rA, rB — R[rB] ← R[rA] if last result ≠ 0
  - cmovge rA, rB — R[rB] ← R[rA] if last result ≥ 0
  - cmovg rA, rB — R[rB] ← R[rA] if last result > 0

Y86-64 ISA: INSTRUCTIONS

- Stack operations and procedure calls
  - call Dest — $R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M_8[R[\%rsp]] \leftarrow PC;$ $PC \leftarrow Dest;$
  - ret — $PC \leftarrow M_8[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$
  - pushq rA — $R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M_8[R[\%rsp]] \leftarrow R[rA]$
  - popq rA — $R[rA] \leftarrow M_8[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$
- Others
  - halt
  - nop

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

USING ZERO AND SIGN FLAGS (SIMPLIFIED)

Condition	Test	Zero flag		Sign Flag
g	> 0	0	and	0
ge	>= 0			0
e	== 0	1		
ne	!= 0	0		
le	<= 0	1	or	1
l	< 0			1

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

Y86 ASSEMBLY LANGUAGE NOTES

- Labels
  - symbolic names for addresses, assigned using label:
  - used anywhere a number can be used (number replaces label)
- Fixed values
  - .byte X   insert the 8-bit number X
  - .long X   insert the 32-bit number X (in Little Endian)
  - .quad X   insert the 64-bit number X (in Little Endian)
  - .byte X, n / .long X, n / .quad X, n   same as above, repeated n times
- Adjust memory location
  - .pos X   set the address of the next instruction (or directive) to X
  - .align X   ensure that address of next instruction is aligned to X bytes, padding (adding unused space) if necessary

CALLING CONVENTIONS

- Some conventions define register and stack usage when one function calls another
  - This permits compiler to create code for a function without knowing where it will be called from
- Calling conventions are:
  - Compiler dependent
  - Architecture dependent
  - Operating system dependent
- For example:
  - 32-bit, x86, gcc-based calling convention is called cdecl (after C)

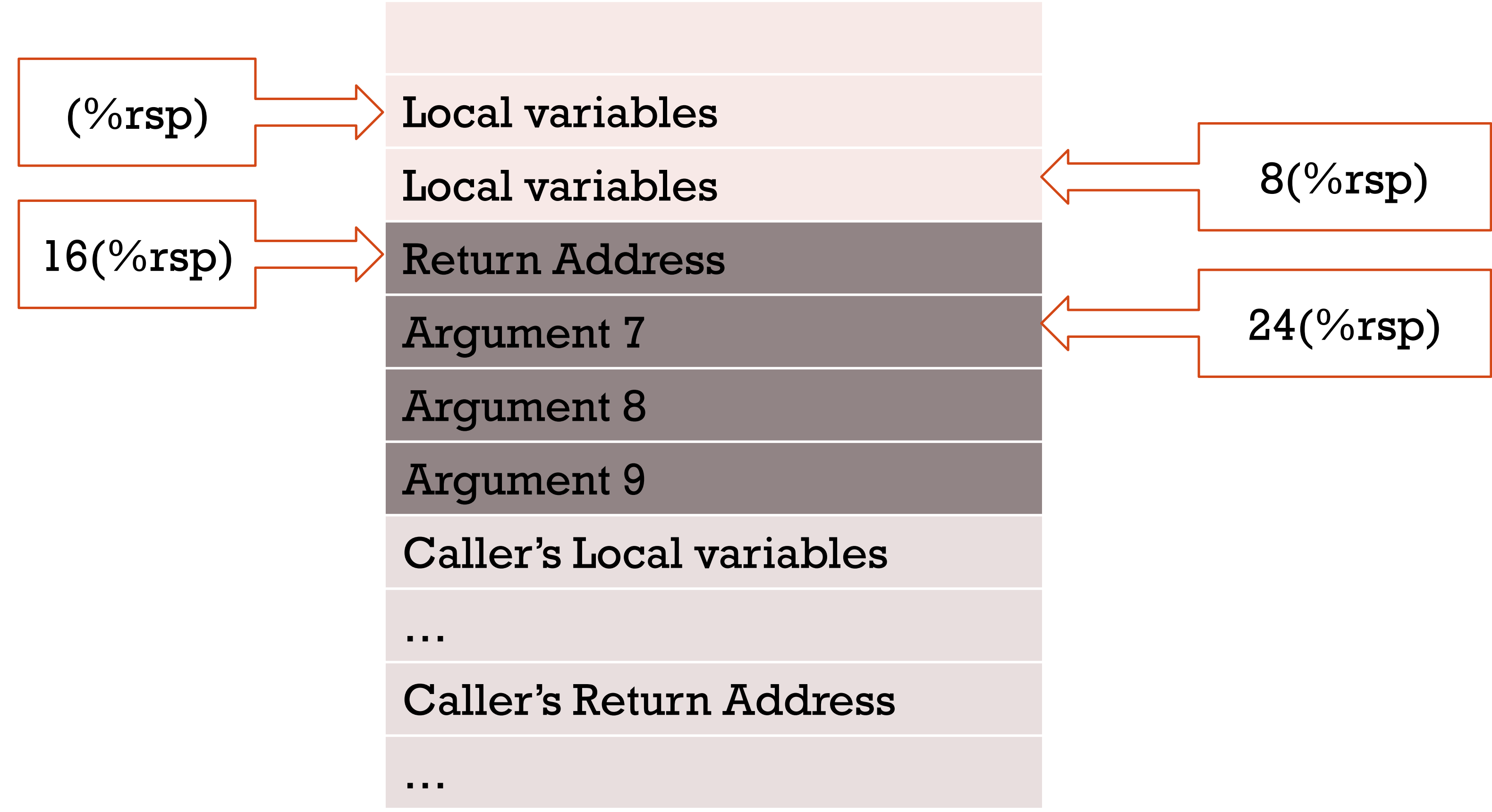
Y86-64 CALLING CONVENTIONS (BASED ON X86-64)

- Stack pointer: %rsp
- Arguments passed in registers, in this order: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Additional arguments passed in stack
- Returning value passed in %rax
- Caller-save registers: %rax, %r10, %r11 + arguments
  - Callee can change their values at will
  - Caller responsible for saving before calling other functions
- Callee-save registers: %rsp, %rbx, %rbp, %r12, %r13, %r14
  - Caller assumes they have same value when function returns
  - Callee must restore value before returning

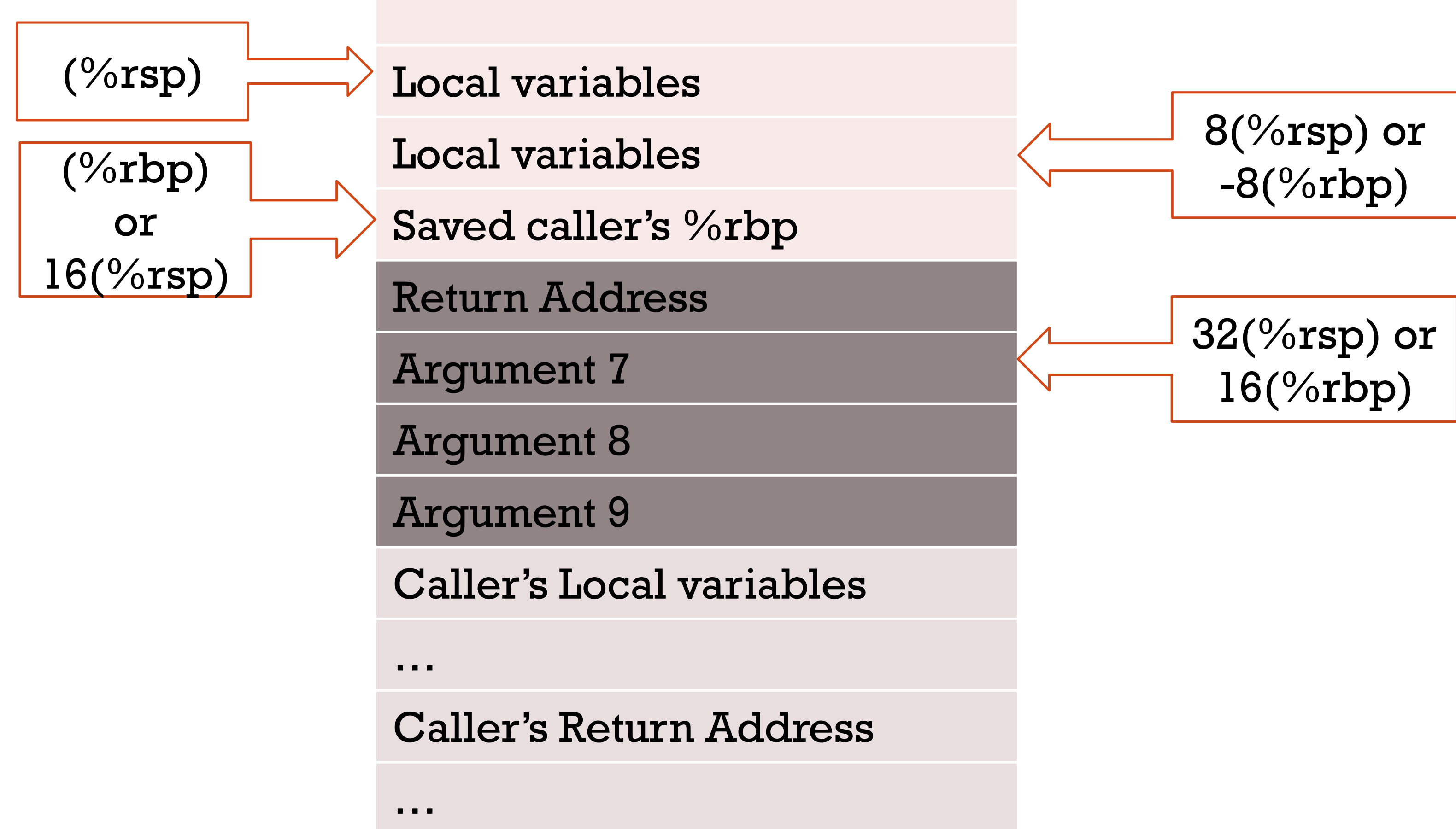
Y86-64 ISA: FUNCTION CALLS AND STACK

- Parameters are passed in registers unless there are too many
- Stack pointer (%rsp) represents the top of the stack
  - %rbp is sometimes used for the base of the frame (where stack was when function started)
- Stack is used for:
  - local variables in functions
  - arguments (if too many)
  - returning address
  - register values before a function call that may change them

STACK STRUCTURE (NO BASE POINTER)



STACK STRUCTURE (WITH BASE POINTER)



X86 VS Y86

- What does x86 have that Y86 doesn't?
- Non-quad versions of instructions and registers:
  - XXXb, XXX, XXXl, XXXq for 8, 16, 32, 64 bits, respectively
  - ah, al, bh, bl, ch, cl, dh, dl: 8-bit registers
  - ax, bx, cx, dx, si, di, sp, bp: 16-bit registers
  - eax, ebx, ecx, edx, esi, edi, esp, ebp: 32-bit registers
- Additional addressing mode:
  - Indexed: e.g., movq 8(%rax, %rbx, 4), %rcx

X86 VS Y86 (CONT.)

- Instructions rmmov, mrmov, irmov, rrmov all called mov
  - Addressing of arguments determines type of instruction
- Additional instructions
  - More ALU operations (inc, dec, or, neg, shl, shr, sal, sar)
  - Floating-point instructions
  - Combination of instructions (e.g., memory load plus ALU)
  - Computation of address (lea)
  - Comparison instructions (test, cmp)
  - Additional conditions: carry (jc/jnc), overflow (jo/jno), sign (js/jns), parity (jp/jnp), unsigned versions (ja/jb/jae/jbe)
  - Special purpose and privileged instructions (iret, xchg)

ASSEMBLY CODE EXAMPLE

- C example:

```c
long start[]  = { 4, 7, 8, 9, 12, 11 };

long sum_function () {
    long sum = 0;
    long count = 6;
    long *str = start;

    while (count) {
        sum += *str;
        str++;
        count--;
    }
    return sum;
}
```

ASSEMBLY CODE EXAMPLE

Compiled to assembly by gcc 4.9.2 with -O1 -S

```c
long start[]  = { 4, 7, 8, 9, 12, 11 };

long sum_function () {
    long sum = 0;
    long count = 6;
    long *str = start;

    while (count) {
        sum += *str;
        str++;
        count--;
    }
    return sum;
}
```

```
sum_function:
        movl    $start, %edx     # long *str = start
        xorl    %eax, %eax       # long sum = 0
.L2:
        addq    (%rdx), %rax     # sum += *str
        addq    $8, %rdx         # str++
        cmpq    $start+48, %rdx  # if str != &start[6]
        jne     .L2              # loop
        rep ret
```

ASSEMBLY CODE EXAMPLE

Converting to y86-64

```
sum_function:
        movl    $start, %edx
        xorl    %eax, %eax
.L2:
        addq    (%rdx), %rax
        addq    $8, %rdx
        cmpq    $start+48, %rdx
        jne     .L2
        rep ret
```

```
sum_function:
        irmovq  start, %rdx     # long *str = start
        xorq    %rax, %rax      # long sum = 0
.L2:    mrmovq  (%rdx), %rbx
        addq    %rbx, %rax      # sum += *str
        irmovq  $8, %rbx
        addq    %rbx, %rdx      # str++
        irmovq  end, %rbx
        subq    %rdx, %rbx      # if str != &start[6]
        jne     .L2             # loop
        ret
```

EXERCISE: TRANSLATE TO Y86

```c
long a = 1830;
long b = 1131;
long gcd;

int main() {
   long r;
   while (b > 0) {
      r = a % b;
      a = b;
      b = r;
   }
   gcd = a;
}
```

Y86 – GCD CODE

```
long a = 1830;
long b = 1131;
long gcd;

int main() {
    long r;
    while (b > 0) {
        r = a % b;
        a = b;
        b = r;
    }
    gcd = a;
}
```

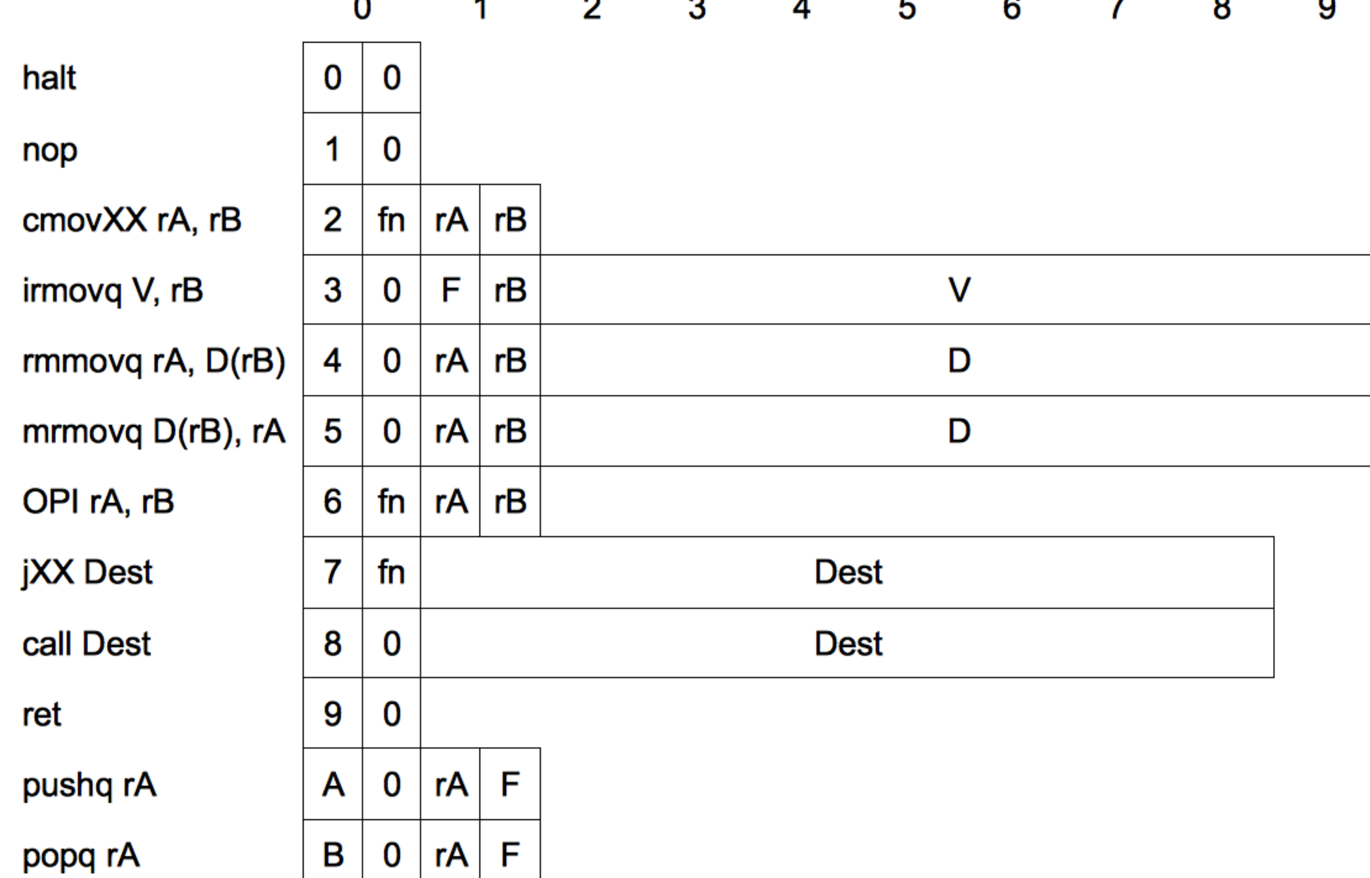
```
.pos 0x100
main:   xorq %rdx, %rdx          # %rdx = 0
        mrmovq a(%rdx), %rax     # %rax = a
        mrmovq b(%rdx), %rbx     # %rbx = b
loop:   subq %rdx, %rbx          # is b <= 0?
        jle     end              # if so, done
        modq %rbx, %rax          # %rax gets remainder
        rrmovq %rax, %rcx        # save remainder
        rrmovq %rbx, %rax        # a = b
        rrmovq %rcx, %rbx        # b = remainder
        jmp     loop
end:    rmmovq %rax, gcd(%rdx) # gcd = a
        rmmovq %rax, a(%rdx)     # update a
        rmmovq %rbx, b(%rdx)     # update b
        halt
.pos 0x1000
a:      .quad 1830
b:      .quad 1131
gcd:    .quad 0
```

WHAT DO WE KNOW SO FAR

- Y86-64 instructions
- Way to specify what an instruction does
- Translation of C to Y86 for simple problems

Now that we know what an instruction does we want to know how an instruction is represented in memory so that the CPU can retrieve it and then execute it.

Y86-64 Memory Representation



Y86-64 Memory Representation

- Instructions format:
  - Arithmetic instructions:
    - addq → fn = 0
    - subq → fn = 1
    - andq → fn = 2
    - xorq → fn = 3
    - mulq → fn = 4
    - divq → fn = 5
    - modq → fn = 6
- Conditional jumps and moves:
  - jmp → fn = 0
  - jle → fn = 1
  - jl → fn = 2
  - je → fn = 3
  - jne → fn = 4
  - jge → fn = 5
  - jg → fn = 6

Y86-64 MEMORY REPRESENTATION

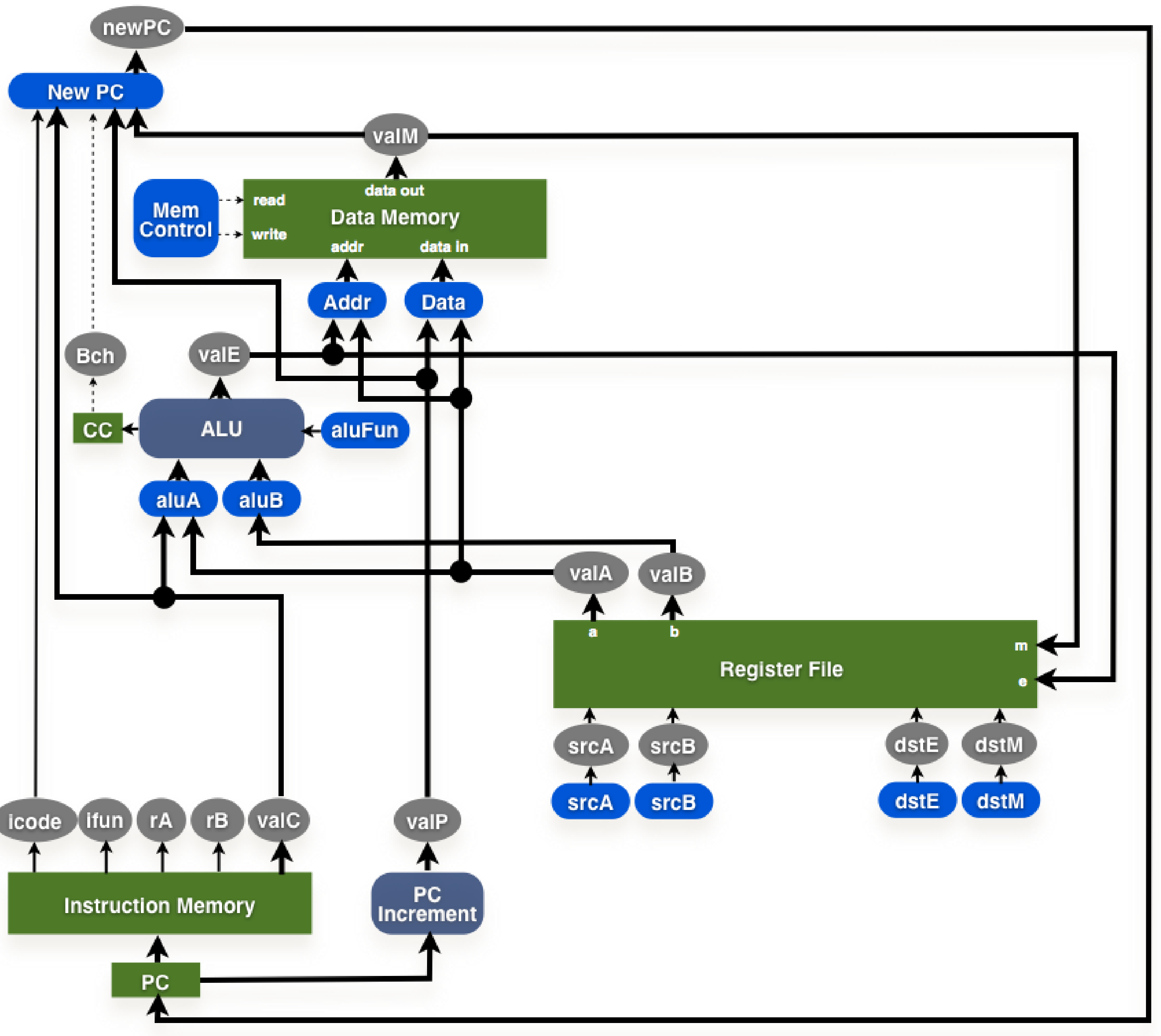
- Translate the following into machine language
  - mrmovq 0x2000(%rax), %rdx
  - xorq %rsi, %rbx
  - jne $0x1234
  - irmovq $0x376, %rax
- Translate the following into assembly language
  - 0x25 0x42
  - 0x80 0x12 0x34 0x56 0x78 0x00 0x00 0x00 0x00

BASIC IMPLEMENTATION – LEARNING GOALS

- Identify CPU stages and the order an instruction goes through them in our Y86 sequential processor
- Describe/explain the general functionality of each instruction execution stage
- Describe/explain, in plain English, what happens in each stage as an instruction is executed
- Use the notation from the text to describe what happens in each stage of the processor as the instruction is executed

BASIC IDEA

- The parts
  - register file
  - PC and instruction registers
  - main memory
  - combinatorial logic (and gates, or gates, etc.)
  - clock
- Describing the combinational logic
  - We will use C/Java-like statements to describe instruction semantics
  - book uses C-like HCL for the same purpose
  - real chips use hardware description languages such as Verilog, VHDL

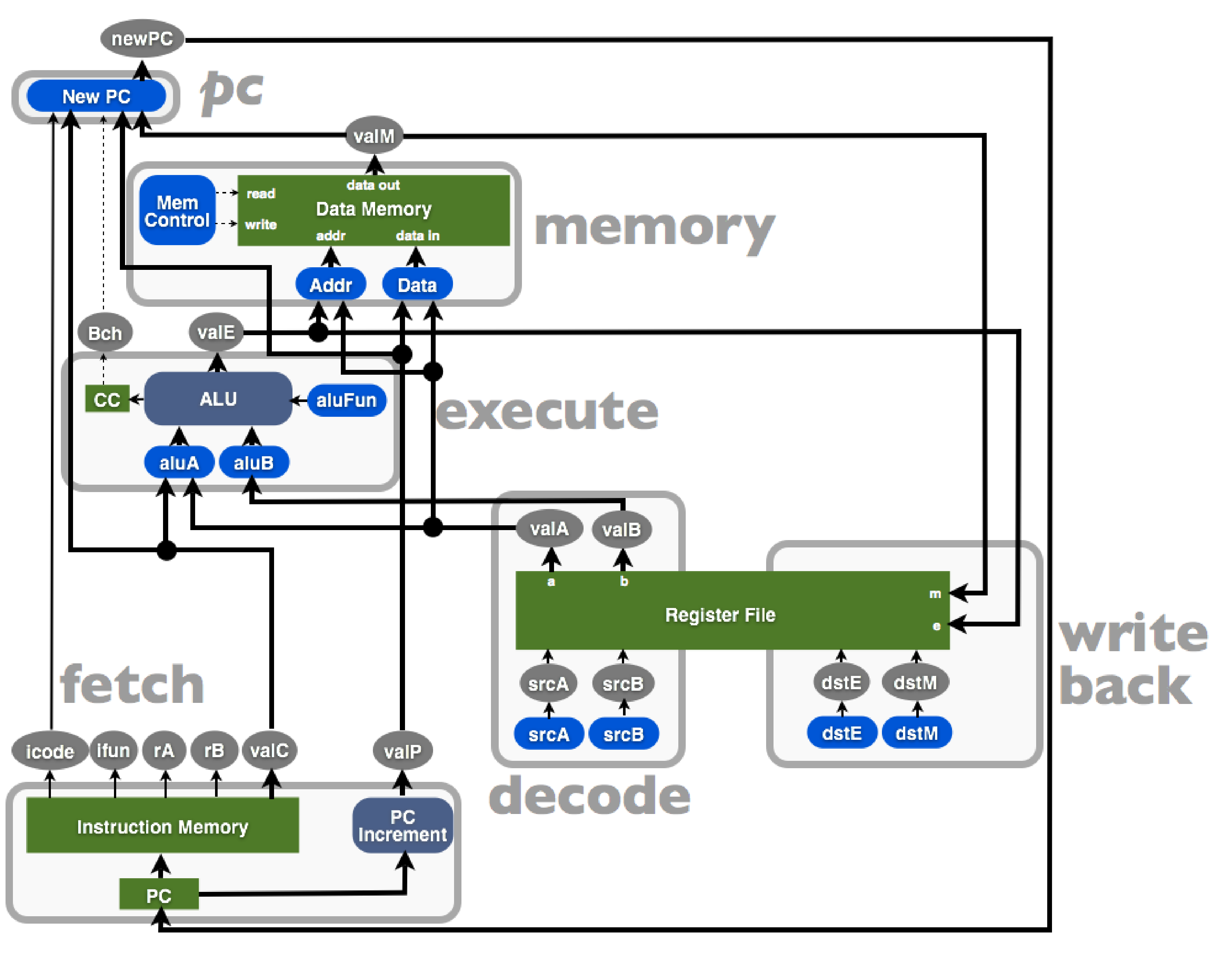

newPC
New PC
valM
Mem Control
read
write
data out
Data Memory
addr
data in
Addr
Data
Bch
valE
CC
ALU
aluFun
aluA
aluB
valA
valB
a
b
m
Register File
e
srcA
srcB
dstE
dstM
srcA
srcB
dstE
dstM
icode
ifun
rA
rB
valC
valP
Instruction Memory
PC Increment
PC

PROCESSOR COMPONENTS

- Memory (green boxes)
  - register file, main memory, program counter (PC) and condition codes (CC)
- Multiplexers (blue boxes)
  - control input selects output from one of several data inputs
  - similar to switch statement in Java
- Special-Purpose Registers (grey boxes)
  - connect memory and multiplexers
  - named to indicate their function
  - later these will store data between CPU stages
  - in Java these are instance variables of stage that contains them

BREAKING THINGS DOWN

- Modularize by dividing instruction execution into 6 stages:
  - *Fetch*: read instruction and decide on new PC value
  - *Decode*: read from registers
  - *Execute*: use the ALU to perform computations
  - *Memory*: read data from or write data to memory
  - *Write-back*: store value(s) into register(s)
  - *PC update*: store the new PC value


newPC
New PC
pc
valM
Mem Control
read
write
data out
Data Memory
addr
data in
memory
Addr
Data
Bch
valE
CC
ALU
aluFun
execute
aluA
aluB
valA
valB
a
b
Register File
m
e
srcA
srcB
srcA
srcB
dstE
dstM
dstE
dstM
write back
fetch
icode
ifun
rA
rB
valC
valP
Instruction Memory
PC Increment
PC
decode

Y86 PROGRAMMER VISIBLE STATE

- The things the programmer sees:
  - Registers -> %rax, %rbx, %rsp …
  - Program counter -> PC
  - Memory -> M_1[addr], M_8[addr] …
  - Status register bits ZF, SF, OF
    - Set when certain instructions execute
    - Indirectly used by certain instructions
    - Sometimes referred to as condition codes (CC) or program status word (PSW)

CAVEAT - PURPOSE OF INSTRUCTIONS

- An instruction precisely describes the actions made on the current programmer visible state to get to the "next" state
- Example: rrmovq rA, rB
  - R[rB] <- R[rA]
  - PC <- PC + 2
- The underlying hardware implementation to achieve the required outcome may be different provided the result conforms to the instruction/architecture specification

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

STAGE FUNCTIONALITY - FETCH

- Reads the instruction code at the address determined by PC
- Based on instruction code, determines the length of the instruction
- Extracts pieces (arguments) of instruction
- Computes the address of the next instruction

STAGE FUNCTIONALITY - DECODE

- Decode:
  - Determines which registers need to be read (e.g., the arguments, or the stack pointer)
  - Reads the values of the registers
  - Determines which registers need to be written to (sets up the signals, but doesn't change registers yet)

STAGE FUNCTIONALITY - EXECUTE

- Depending upon the instruction, produces the value of the computation performed at the execute stage
- This could be:
  - Result of operation (+, -, /, * etc) specified by ifun
  - Effective address for a memory reference
  - Increment or decrement of the stack pointer
- Condition Codes (CC) (i.e. ZF, OF, SF) could be set if it is an arithmetic operation
- Conditional instructions (jXX, cmovXX): evaluates result of corresponding condition
  - Translates previous CC to a flag based on condition

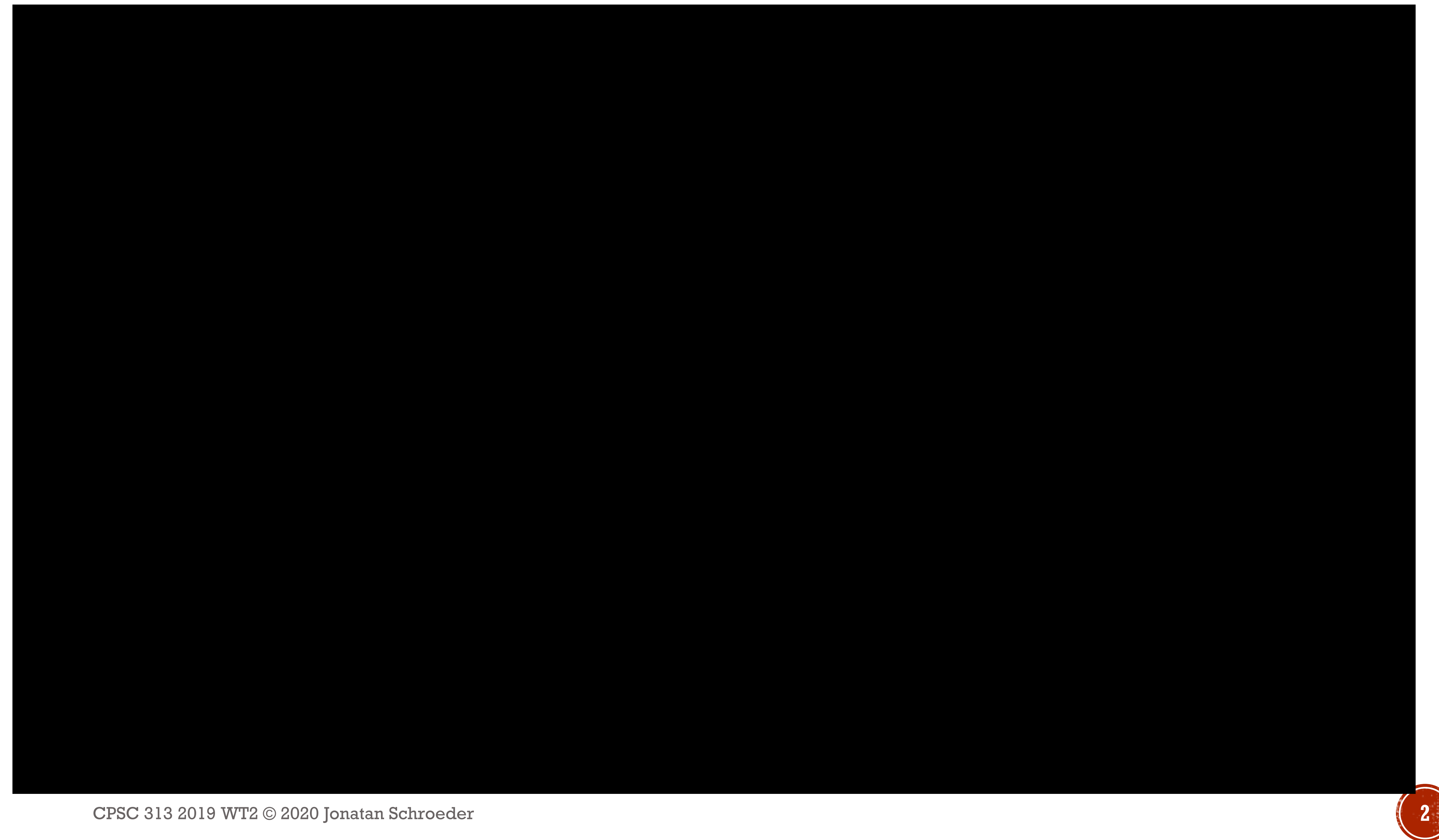
STAGE FUNCTIONALITY

- Memory:
  - Read data from main memory, at an address computed by previous stages
  - Writes data to main memory, with data and address computed by previous stages
- Write back
  - Based on signals computed in decode, changes register values
  - These are typically arguments or the stack pointer
- PC Update
  - Determine the next PC value (following instruction, jump destination, return destination), possibly based on condition

IMPLEMENTING A PIPELINED CPU

Unit 2

CPSC 313 2019 WT2 © 2020 Jonatan Schroeder



PIPELINED Y86 IMPLEMENTATION

- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

MOTIVATION

In a sequential Y86 implementation

- Instr 1:
- Instr 2:
- Instr 3:

Instr 1:	F	D	E	M	W						
Instr 2:						F	D	E	M	W	
Instr 3:											F	D

PIPELINE COMPUTATION

- Represent each stage as a module (fetch, decode, ...)
  - Modules are ordered along the flow of computation
  - One module's output is the input of the next module
- Turn each module into a pipeline stage
  - Add pipeline registers before every stage except the 1st
  - These store the inputs for that stage
  - Stages execute in parallel working on different instructions
- As we will see later this introduces new problems

 © 2020 Jonatan Schroeder

PIPELINE STAGES

How many stages should a pipeline have?

- If it has too few stages…
  - We are not exploiting the parallelism present in the program
- If it has too many…
  - There is high overhead and complexity
  - The program may not have enough parallelism to use them well

EXAMPLES

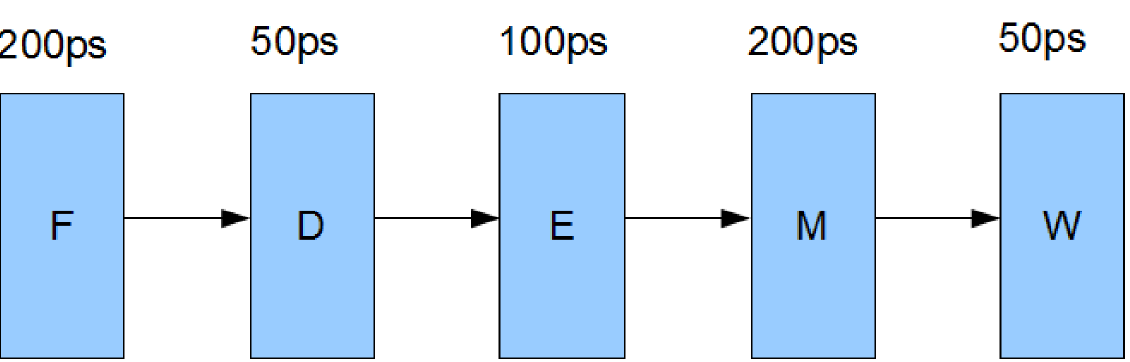
- MIPS processor (1985): first RISC processor, 5 stages
- Sparc, PowerPC processors: 9 pipeline stages
- Intel Pentium IV (late models): ____________
- Intel Core i7 processor: _______________

MEASURING EFFICIENCY

- Latency:
  - How long it takes to execute one instruction from start to finish
  - Will usually not be reduced in a pipeline
- Throughput:
  - The number of instructions we can execute per unit of time
  - This is the only meaningful measure for pipelined CPUs

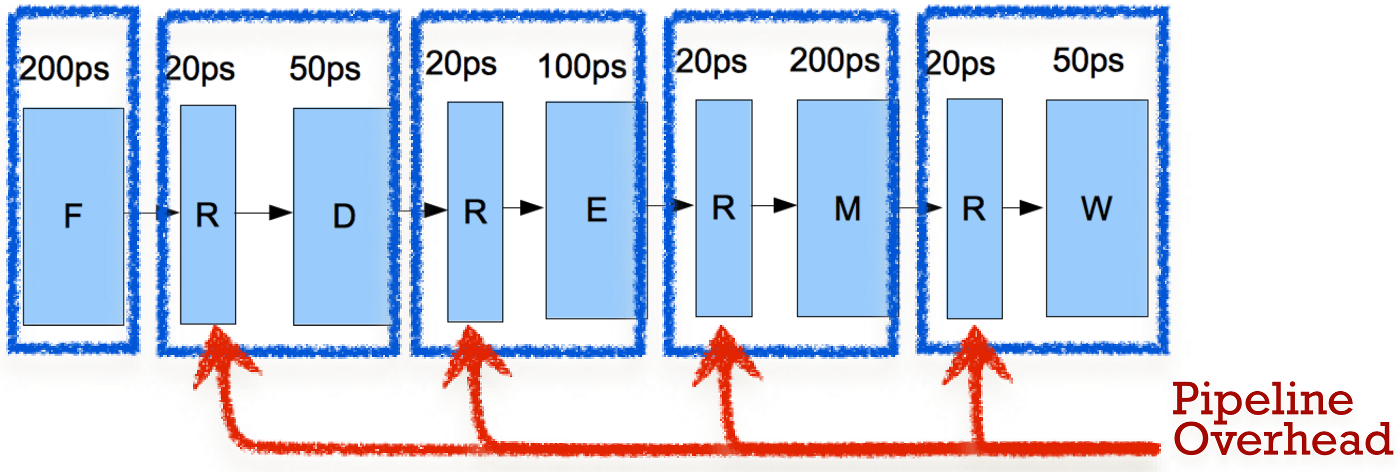
LATENCY & THROUGHPUT

- Sequential Implementation



- Latency: ______________
- Throughput: ____________

LATENCY AND THROUGHPUT



Assuming all stages are in use

- Throughput: ____________________
- Latency: ____________________

LATENCY AND THROUGHPUT

- Generalizing for pipelined CPUs:
  - Stages require an additional overhead
    - Storing and retrieving special registers
    - Latency for one instruction increases
  - New instruction can start executing once first stage is complete
    - Better throughput overall
  - All stages must run in same time slot
    - Can’t move to the next instruction until slowest stage is free

PIPELINED Y86 IMPLEMENTATION

- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

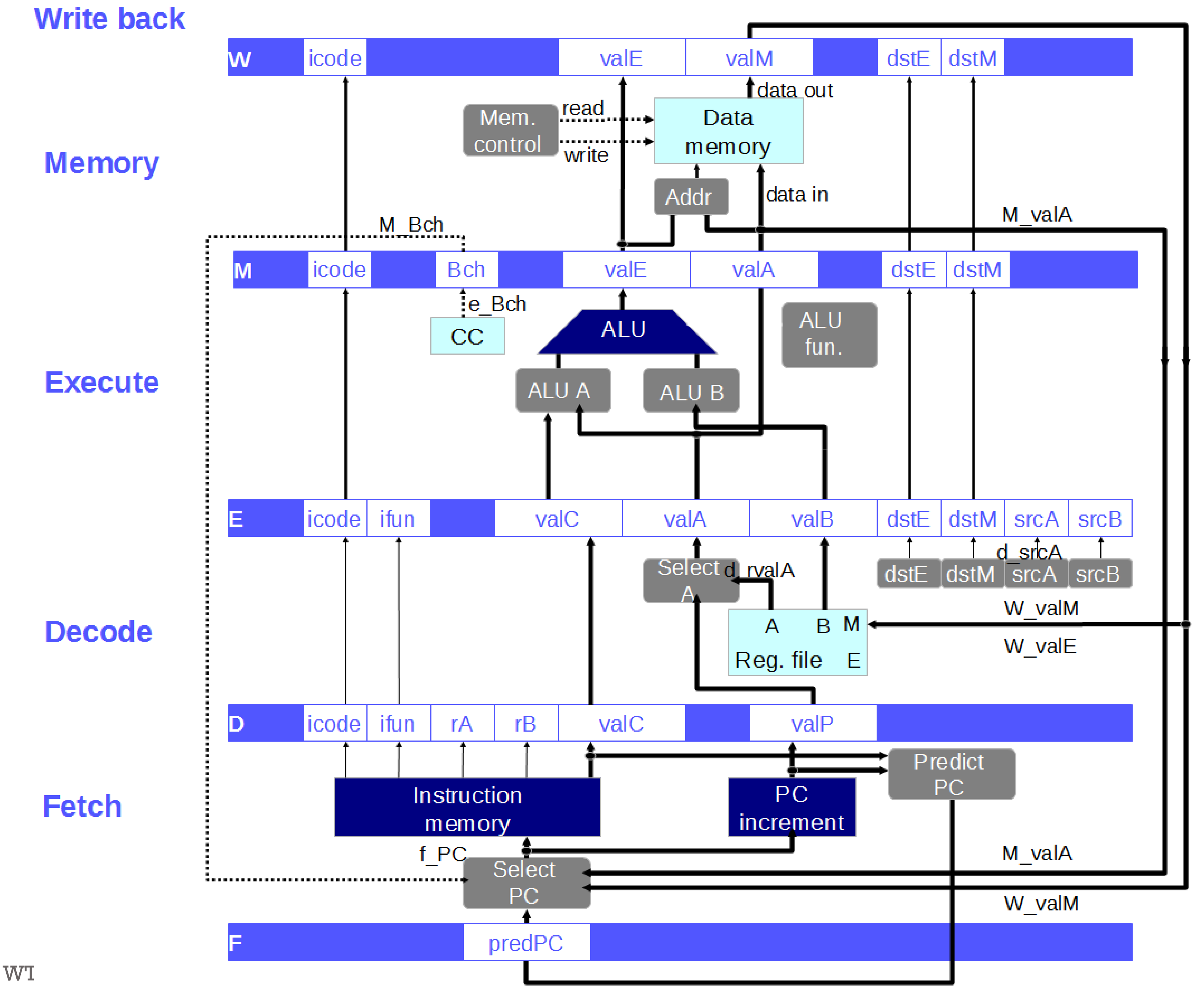
PIPELINED COMPUTATION

- divide execution into modules along flow of computation
  - classic RISC pipeline has five modules
  - modules arranged in order along computation flow
  - all inputs to a module must be computed by an earlier module in flow
- turn modules into pipeline stages
  - add pipeline registers between each stage
  - registers store inputs for that stage
  - each stage executes in parallel working on a different instruction
- observe that
  - a stage has less gate-propagation delay than overall circuit
  - what determines clock rate?

THE RISC PIPELINE

- How many stages?
  - enough to achieve sufficient parallelism
    - must have enough parallelism in the program
  - not too much to add undue overhead or complexity
- Which stages?
  - divide instructions into stages that instruction completes in order
  - then we can execute the stages in parallel on different instructions
- Example: rmmovq rA, D(rB)
  - what are the parts?
  - what order do they need to execute?

CPSC 313 2019 WT2 © 2020 Jonatan Schroeder



Write back
W
icode
valE
valM
dstE
dstM
data out
Mem. control
read
write
Data memory
Addr
data in
Memory
M_Bch
M_valA
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
CC
ALU
ALU fun.
Execute
ALU A
ALU B
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
A
B
M
Reg. file
E
W_valE
Decode
D
icode
ifun
rA
rB
valC
valP
Predict PC
Fetch
Instruction memory
PC increment
f_PC
M_valA
Select PC
W_valM
F
predPC

AN EXAMPLE PROBLEM

- What will each pipeline register contain when the last instruction in this sequence is entering the Fetch stage?

```
irmovq $1, %rax
irmovq $2, %rcx
irmovq $3, %rdx
irmovq $4, %rbx
irmovq $5, %rsi
```

TERMINOLOGY

- an instruction is ***in flight*** *when*
  - it is executing in the pipeline
- an instruction is ***retired*** when
  - it exits the pipeline; i.e., it completes
- on upcoming slides ...
  - ***exploiting*** and ***expressing*** parallelism
  - ***instruction-level parallelism***
  - instruction ***dependencies***
  - ***thread-level parallelism***
  - ***sequential consistency***
  - pipeline ***hazard, stall and bubble***

EXPRESSING VS EXPLOITING PARALLELISM

- Expressing parallelism: it's a mechanism where the programmer tells the system that two pieces of code can execute in parallel
- Exploiting parallelism: it's the system actually executing two pieces of code in parallel
- How do you express parallelism in C or Java?
- Is this mechanism useful for expressing ILP?

EXPLOITING INSTRUCTION-LEVEL PARALLELISM

The problem with instruction-level parallelism is:

- Programming languages like C, C++ and Java are based on the sequential consistency model:
  - The effect of executing the program must be the same as if instructions were executed one by one in the order they are written
- Programmers write code without thinking about parallelism
  - Example:
    a = b + c;   d = a + 1;   e = f + g;
    Can be rewritten as:
    a = b + c;   e = f + g;   d = a + 1;
- Compilers must find this on their own

PIPELINED Y86 IMPLEMENTATION

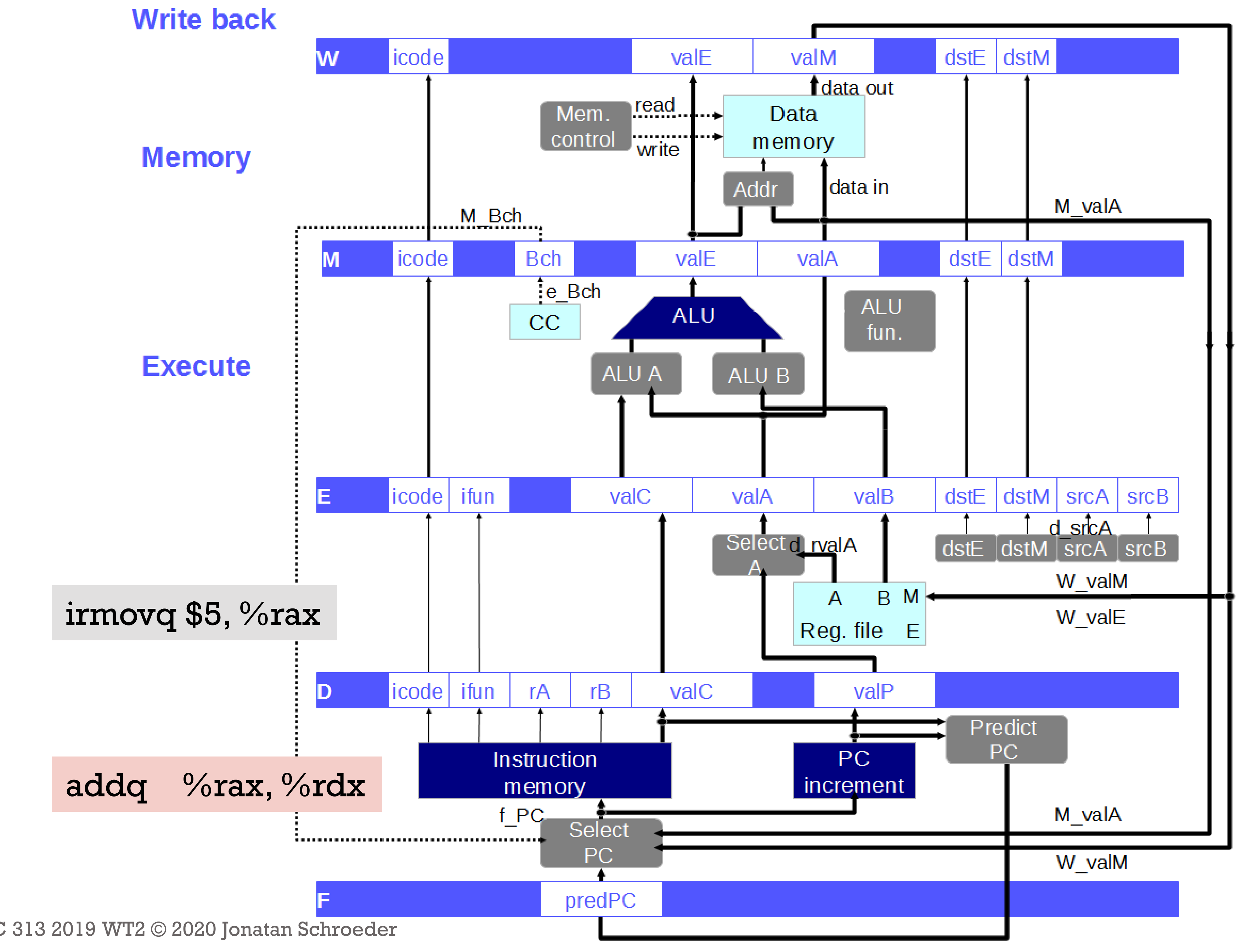
- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

PIPELINE CONSIDERATIONS

- Consider this code:

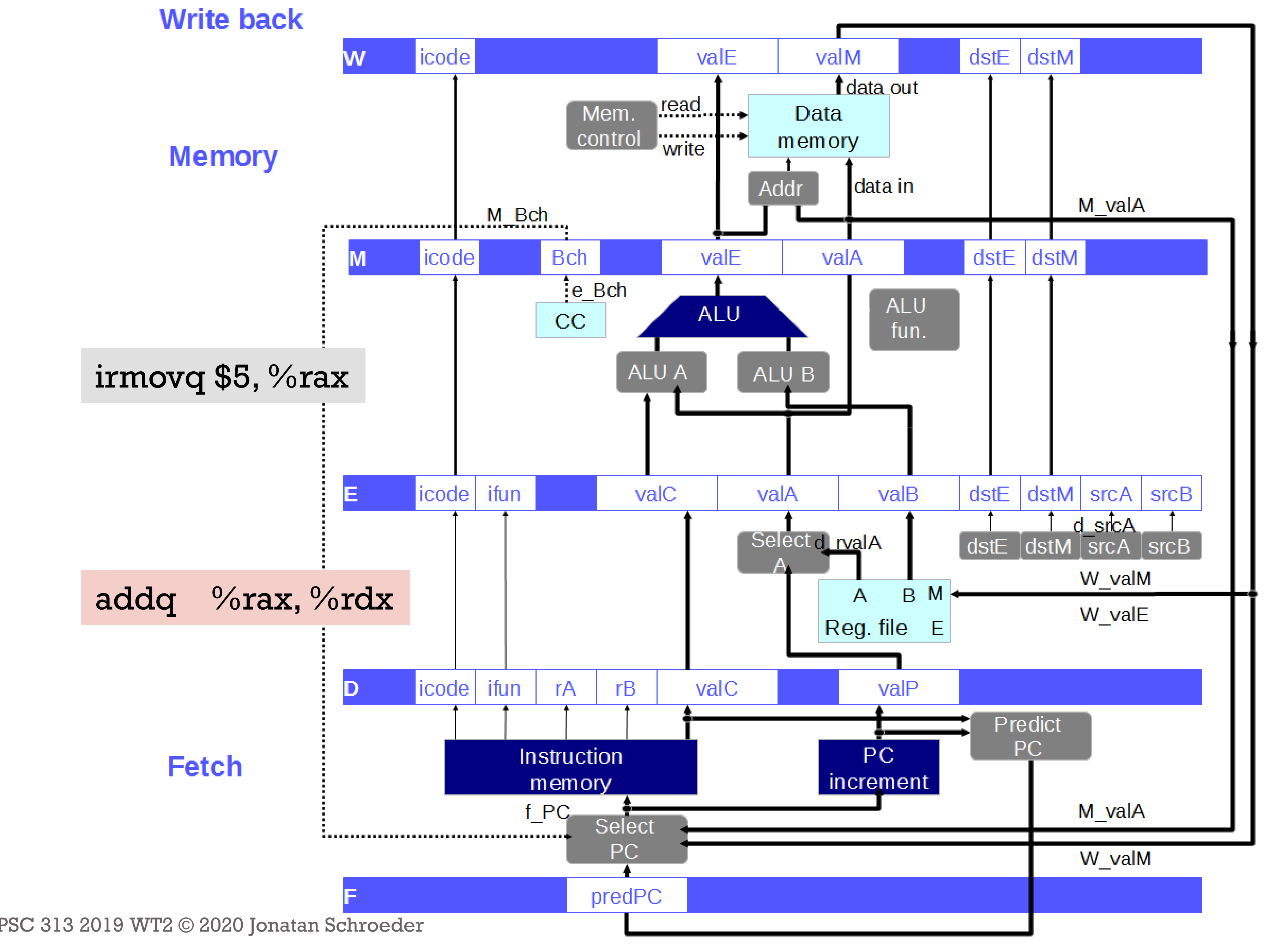
```
irmovq $5, %rax
addq   %rax, %rdx
```

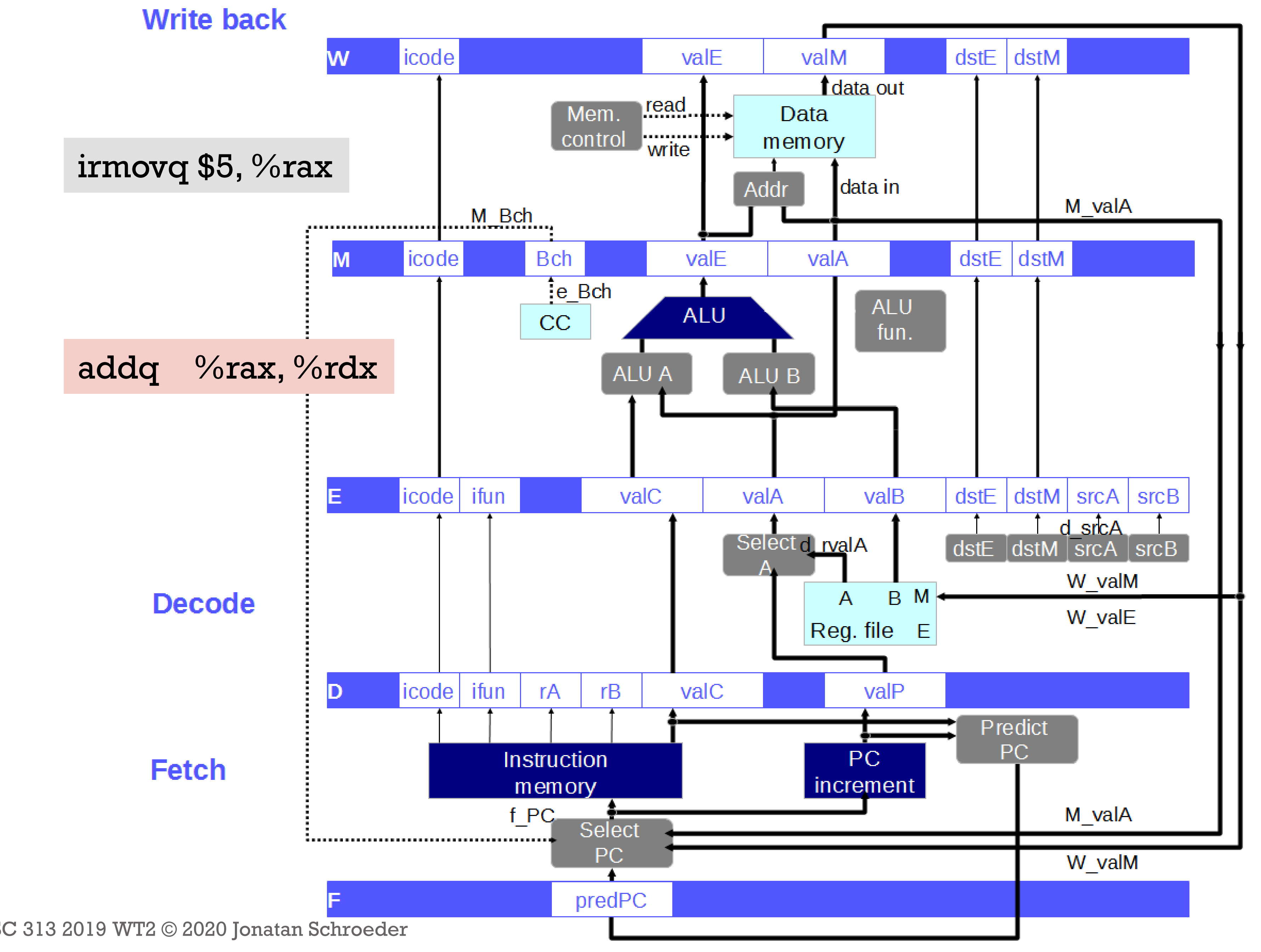
- At what stage is irmovq in when addq needs its result?
  - At what stage is the value of %rax needed in addq?
  - At what stage is the value of %rax updated in irmovq?



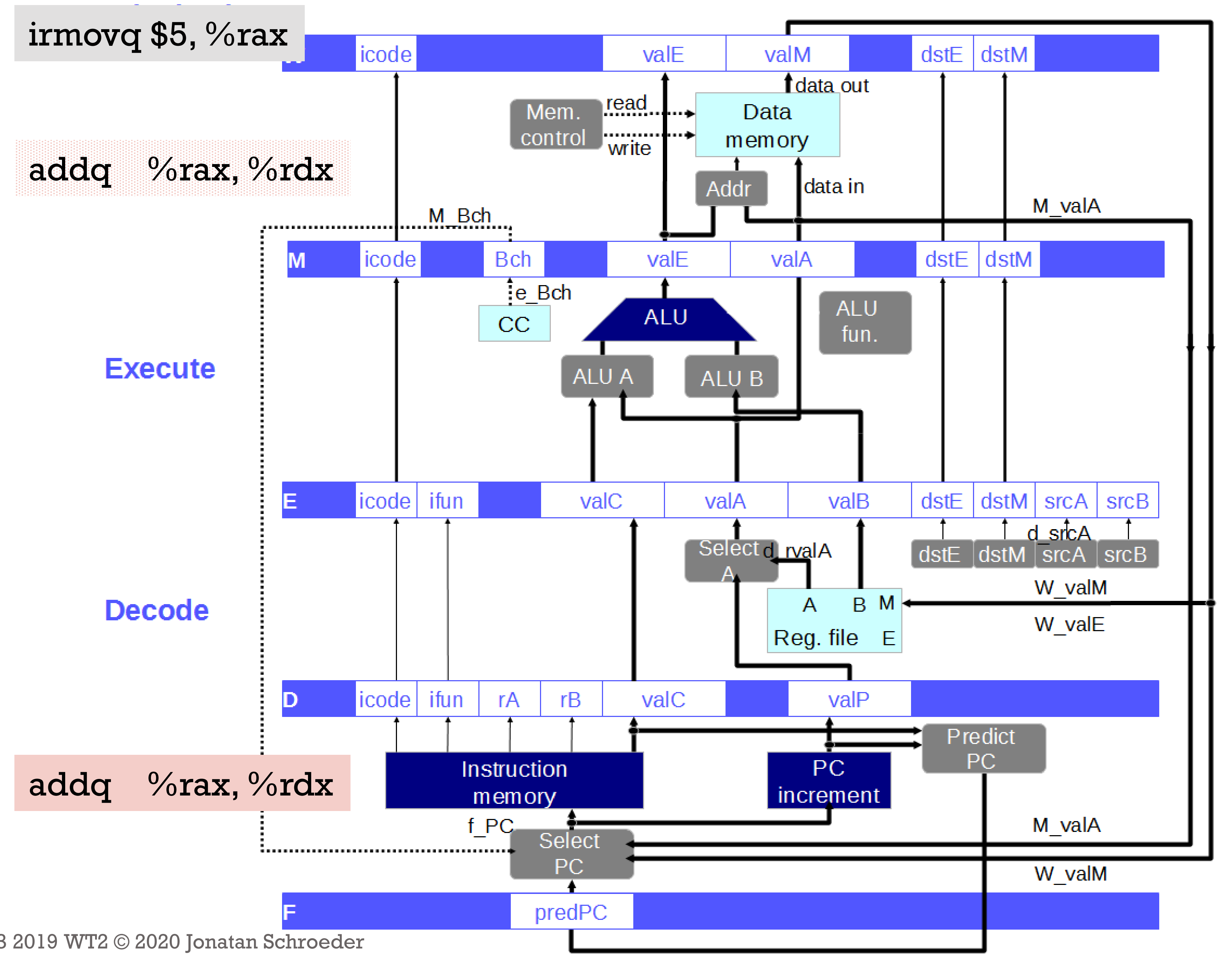
irmovq $5, %rax

addq %rax, %rdx


Write back
Memory
Fetch
irmovq $5, %rax
addq %rax, %rdx
W
icode
valE
valM
dstE
dstM
data out
Mem. control
read
write
Data memory
Addr
data in
M_valA
M_Bch
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
CC
ALU
ALU fun.
ALU A
ALU B
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
Select A
d_rvalA
d_srcA
dstE
dstM
srcA
srcB
W_valM
W_valE
A
B
M
Reg. file
E
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
f_PC
Select PC
M_valA
W_valM
F
predPC



Write back
irmovq $5, %rax
addq %rax, %rdx
Decode
Fetch
W
icode
valE
valM
dstE
dstM
data out
read
Mem. control
Data memory
write
Addr
data in
M_valA
M_Bch
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
CC
ALU
ALU fun.
ALU A
ALU B
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
A
B
M
Reg. file
E
W_valE
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
f_PC
M_valA
Select PC
W_valM
F
predPC


irmovq $5, %rax
addq %rax, %rdx
addq %rax, %rdx
Execute
Decode
W
icode
valE
valM
dstE
dstM
data out
Mem. control
read
write
Data memory
Addr
data in
M_valA
M_Bch
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
CC
ALU
ALU fun.
ALU A
ALU B
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
W_valE
A
B
M
Reg. file
E
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
f_PC
M_valA
Select PC
W_valM
F
predPC

INSTRUCTION-LEVEL PARALLELISM

- Pipeline requires some parallelism
  - multiple in-flight instructions run partly at the same time
  - may even run out-of-order
- However, the program **must** have the same output as if instructions were executed sequentially (*sequential consistency*)

DEPENDENCIES CONSTRAIN PARALLELISM

- Execution of 2 or more instructions has to proceed in strict time order
  - Must be able to produce the same output as if one instruction were completely executed before the next instruction is started
- If no dependency, execution order doesn't matter
- Compilers try to expose as much instruction-level parallelism as they can
  - By separating dependent instructions
  - By grouping them with others on which they don't depend

AVOIDING DEPENDENCIES IN CODE

- Example: how can we rewrite the following code to expose more parallelism?

```
addq %rax, %rbx
addq %r8, %rbx
addq %rdx, %rcx
addq %r9, %rcx
addq %rsi, %rdi
addq %r10, %rdi
```

COMPILERS & INSTRUCTION PARALLELISM

- Compilers can reorder instructions to expose as much instruction-level parallelism as possible.
- However they cannot know every detail of the processor's pipeline (e.g. later Pentium IV's had more stages than earlier ones).
  - How many instructions should be kept between dependencies?
- So the **pipeline** must handle all dependencies correctly
- Combined effort:
  - Reordering reduces the need for dependency control
  - Pipeline handles what can't be handled in software

DEPENDENCY TYPES

- Data dependencies
  - Involve dependencies between registers
  - Example: one instruction needs a register that is written by a previous instruction
- Control dependencies
  - Involve the flow of control (changes in PC)
  - Example: conditional jumps decide PC based on the result of a previous instruction

CLASSIFYING DATA DEPENDENCIES

- if A and B are instructions, then $A \prec B$ means instruction B depends on instruction A
  - Causal: $A \prec B$ if B reads a value written by A
  - Output: $A \prec B$ if B writes to a location written by A
  - Alias (anti): $A \prec B$ if B writes to a location read by A

TYPES OF DATA DEPENDENCY: EXERCISE

a) no dependency

b) causal dependency

c) anti-dependency

d) output dependency

1.	3.
a = 1; b = 2;	a = 1; b = a;
2.	**4.**
a = 1; a = 2;	a = b; b = 2;

TYPES OF DATA DEPENDENCY: EXERCISE

a) no dependency
b) causal dependency
c) anti-dependency
d) output dependency

1.	3.
irmovq $1, %rax rrmovq %rcx, %rax	rrmovq %rax, %rcx irmovq $1, %rax
2.	**4.**
irmovq $1, %rax rrmovq %rax, %rcx	irmovq $1, %rax irmovq $2, %rcx

CONTROL DEPENDENCIES

- Control dependencies determine what code is executed next
- Examples:
  - Whether a branch is taken or not taken (e.g., conditional jumps)
  - When the next instruction is obtained from a register or memory (e.g., return)
  - When an instruction writes to instruction memory (self-modifying code)
    - Will not be explored

WHEN DEPENDENCIES BECOME HAZARDS

- Dependencies are not always a problem
  - If there are enough instructions in between dependent instructions, there is no hazard
- Hazard happens when "normal" pipeline execution violates the dependency
- CPU must change its behaviour
  - By stalling instructions
  - By forwarding values from previous instructions
  - By speculating what instructions must run

PIPELINED Y86 IMPLEMENTATION

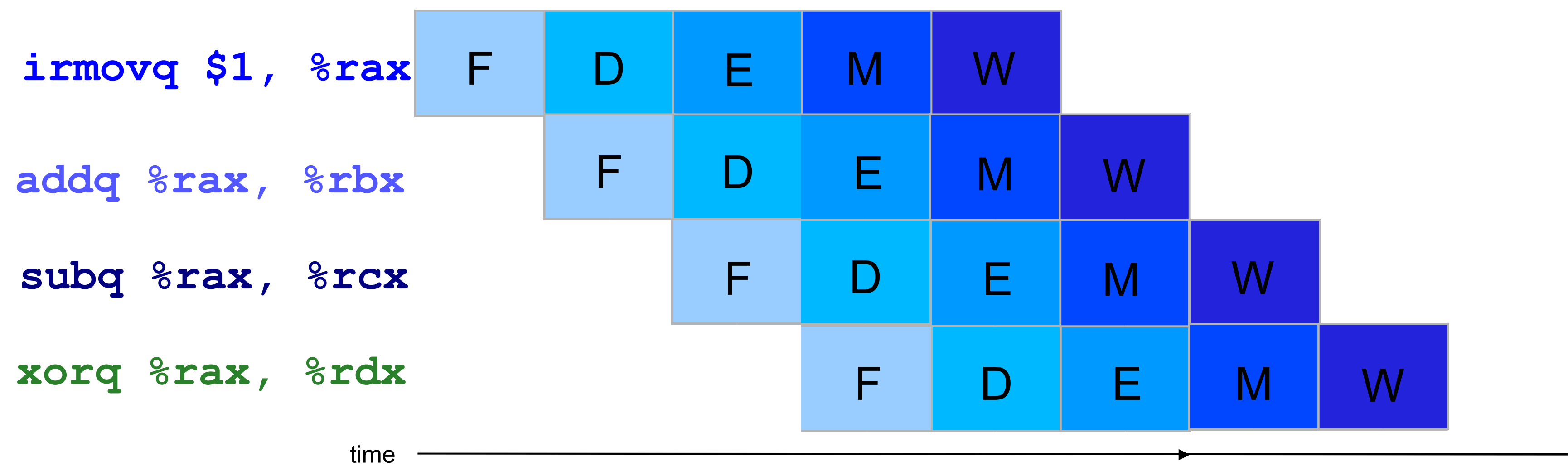
- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

LEARNING GOALS

- Explain how a pipelined processor uses stalling, data forward, and branch prediction to reduce or eliminate hazards.
- Define what is meant by a pipeline bubble and explain why/how a bubble is generated.
- For a sequence of machine language instructions, describe at an arbitrary execution point the state of the pipeline:
  - When there are no hazards
  - When hazards are handled using only stalling
  - When hazards are handled using stalling and/or data forwarding
  - When hazards are handled using stalling, data forwarding, and/or branch prediction

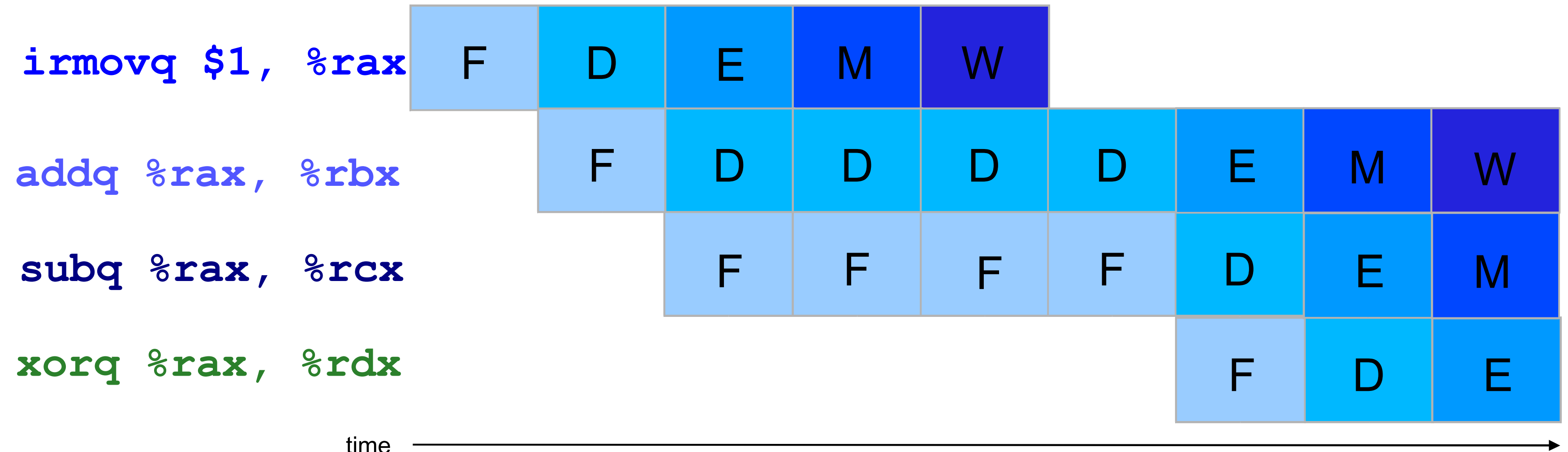
DATA HAZARDS

- What is the problem here?



STALLING

- pipeline stall: hold an instruction in a pipeline stage for extra cycles



BUBBLE

- The term pipeline bubble denotes a pipeline stage that is forced to do nothing to avoid a hazard, because of a stall in a previous stage
  - For example, if decode stalls, in the next cycle execute will have a bubble
- The bubble converts the instruction into a NOP

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**
xorq	subq	addq	**bubble**	**bubble**

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**
xorq	subq	addq	**bubble**	**bubble**
	xorq	subq	addq	**bubble**

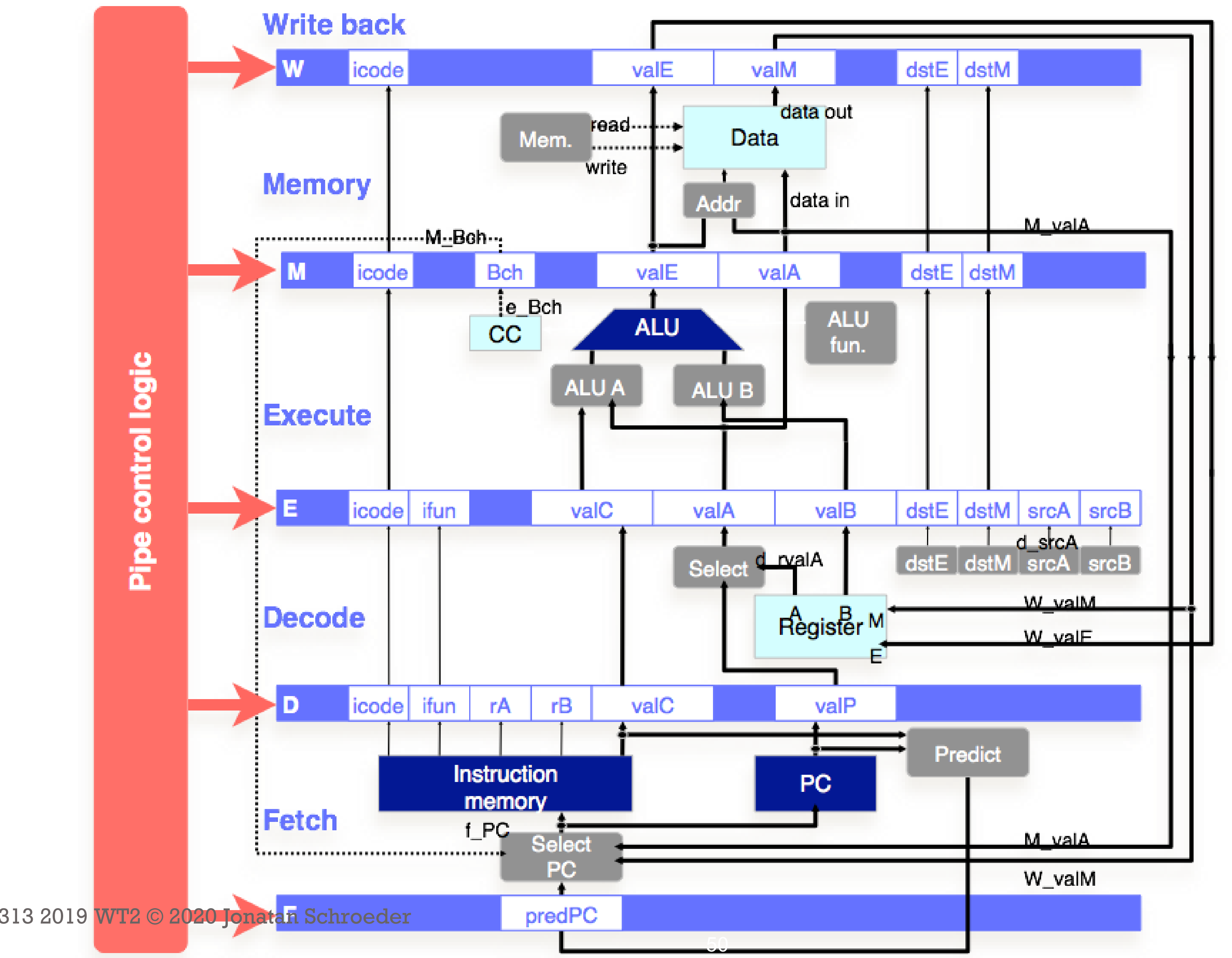
EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**
xorq	subq	addq	**bubble**	**bubble**
	xorq	subq	addq	**bubble**
		xorq	subq	addq

PIPELINE CONTROL UNIT

- The pipeline-control module
  - is a hardware component (circuit) separate from the 5 stages
  - examines values across every stage
  - decides whether stage should stall or bubble
- Each stage register has a control input that determines what happens when the clock ticks:
  - Normal: register's new value is the input value
  - Stall: register's new value is the same as its current value
  - Bubble: register's new value is the same as for NOP

PIPELINE CONTROL UNIT



STALLING WORKS, BUT…

- Stalls should be avoided
  - every stall creates a pipeline bubble
  - every bubble results in a cycle when processor can't retire an instruction
  - retiring a bubble is of no value to program execution
- Bubbles decrease overall throughput

STALLING SUMMARY

- data hazards
  - reading registers in decode that are written by instructions currently in execute, memory or write-back stages
  - stall instruction in decode until writer is retired
  - Write-back only writes register when next clock ticks, so Decode can't run until Write-back finishes
  - how many stall cycles? ______

STALLING FOR CONTROL DEPENDENCIES

- Conditional jump
  - At what stage do we know which PC?
  - At what stage is the PC needed?
  - how many stall cycles? ______
- Return
  - The next PC is available at the end of?
  - how many stall cycles? ______

WHAT ABOUT ANTI AND OUTPUT DEPENDENCIES

- If instructions are executed in-order, they are not a problem (not a hazard)
- Output dependency: first instruction's output can be ignored

Output:
```
irmovq  $1, %rax
irmovq  $2, %rax
```
Anti:
```
addq    %rax, %rdx
irmovq  $1, %rax
```
Causal:
```
irmovq  $2, %rax
addq    %rax, %rdx
```

ANTI AND OUTPUT DEPENDENCIES

For our pipelined Y86 implementation:

Anti-dependency:

```
addq %rax, %rdx
irmovq $1, %rax
```

F	D	E	M	W	
	F	D	E	M	W

Output dependency:

```
addq %rax, %rdx
irmovq $1, %rdx
```

F	D	E	M	W	
	F	D	E	M	W

As long as instructions execute in order, no problem

OTHER DEPENDENCIES

For our pipelined Y86 implementation:

Memory dependency:

rmmovq %rax, (%rbx)

mrmovq (%rbx), %rcx

F	D	E	**M**	W	
	F	D	E	**M**	W

Condition code dependency:

addq %rax, %rdx

cmovle %rcx, %rbx

F	D	**E**	M	W	
	F	D	E	M	**W**

Again, as long as instructions execute in order, no problem

PIPELINED Y86 IMPLEMENTATION

- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

DEALING WITH CAUSAL DEPENDENCIES

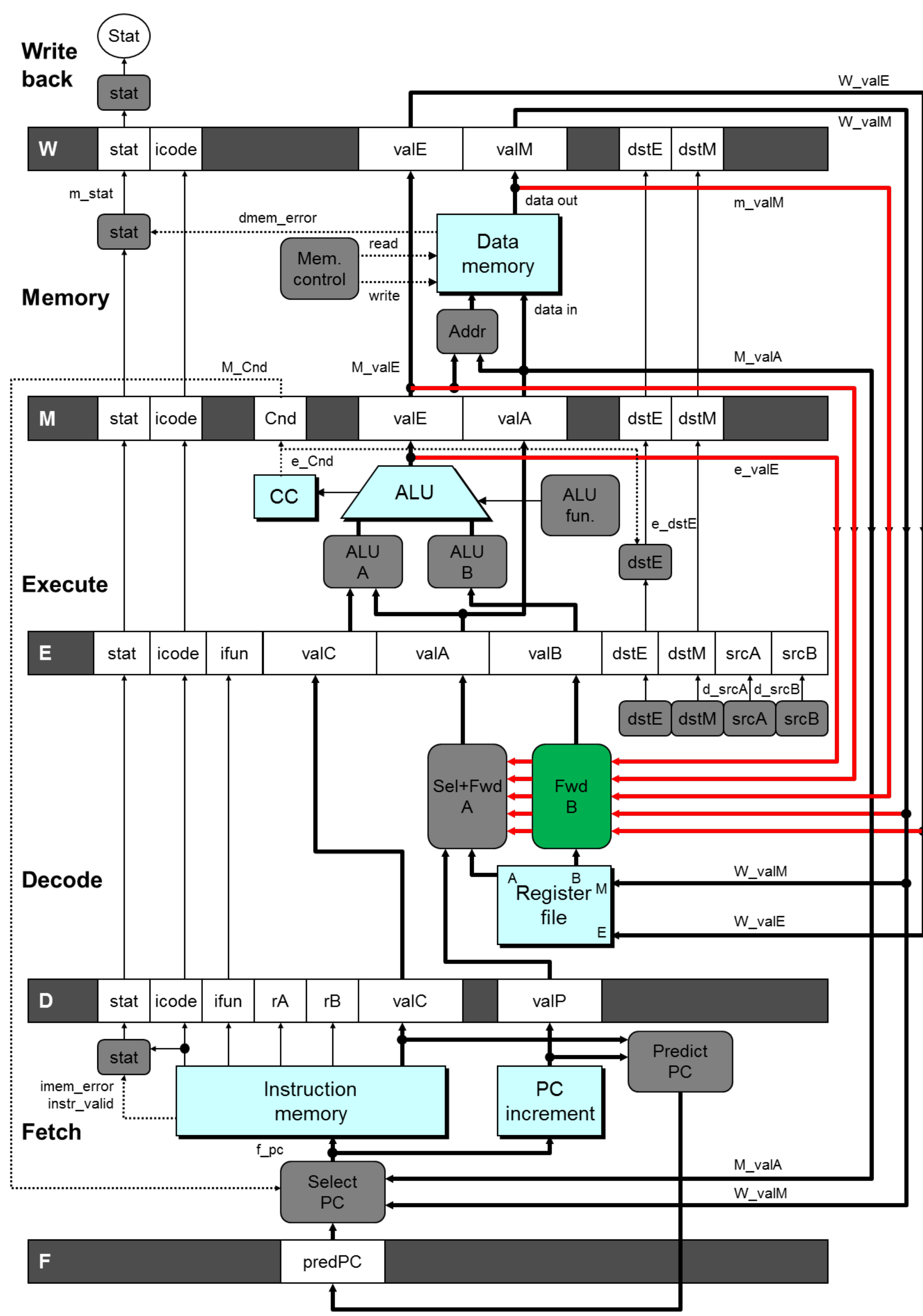
- may have to stall
  - an instruction can't read a value that the processor doesn't know yet
- but in most cases processor has computed value, it just hasn't written it to the register file yet
- Forwarding data will resolve hazard in these cases
  - Data forwarding: mechanism that forwards values from later pipeline stages to earlier ones
  - instruction can read value before it is written back to register file

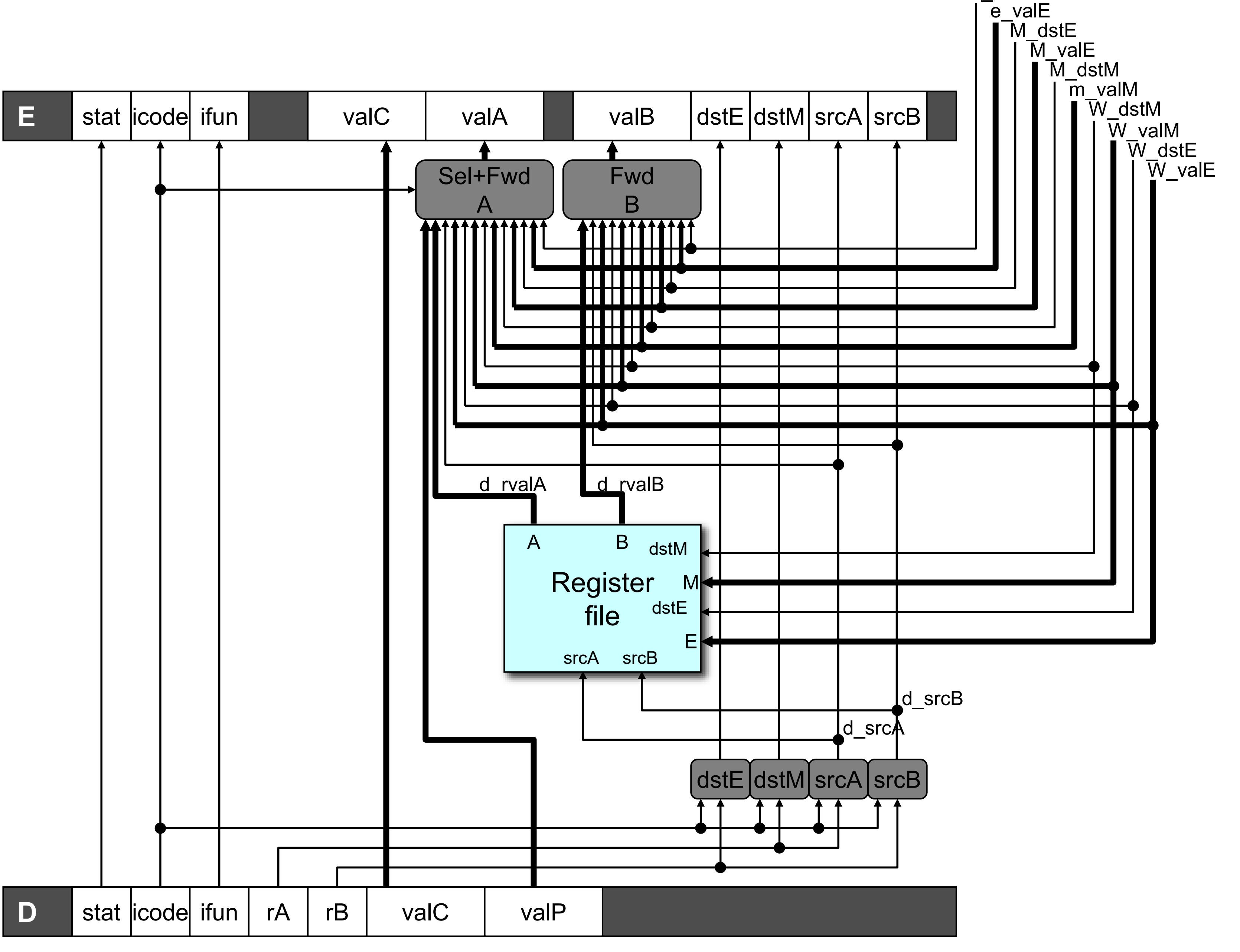
DATA FORWARDING

- In y86, at the end of what stage is output known by CPU?
  - addq, subq, andq, xorq, mulq, divq, modq
  - irmovq, rrmovq
  - cmovXX
  - pushq, popq, call, ret (changed %rsp)
  - mrmovq, popq (target reg)
- in y86, at the end of what stage is register value needed?

PIPE (WITH FORWARDING)

- **To** Decode
- **From**
  - W: new register value
  - M: new value read from memory
  - E: new value from ALU
- **By** sending value and name from all three stages back to forwarding logic in decode stage





E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
e_dstE
e_valE
M_dstE
M_valE
M_dstM
m_valM
W_dstM
W_valM
W_dstE
W_valE
Sel+Fwd
A
Fwd
B
d_rvalA
d_rvalB
A
B
dstM
Register
file
M
dstE
E
srcA
srcB
d_srcB
d_srcA
dstE
dstM
srcA
srcB
D
stat
icode
ifun
rA
rB
valC
valP

REGISTER TO REGISTER HAZARD WITH DATA FORWARDING

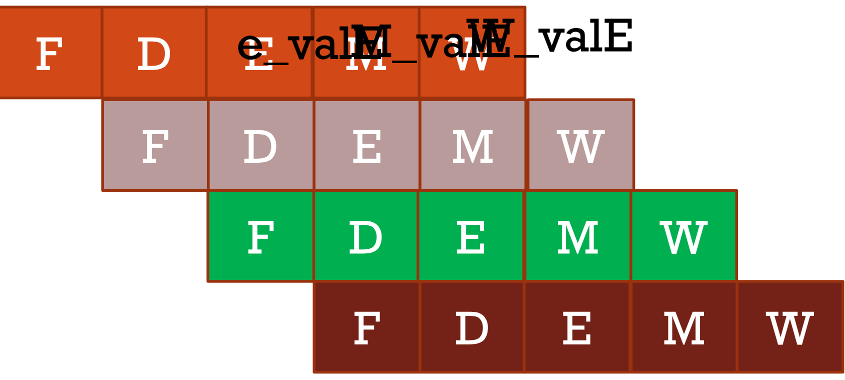
- Instructions:
  - irmovq, addq, subq, andq, xorq
  - pushq, popq, call, ret - all depend upon %rsp
- Forward from ______ to _________

irmovq $10, %rax

addq %rax, %rbx

addq %rax, %rcx

addq %rax, %rdx



Time

LOAD USE HAZARD WITH DATA FORWARDING

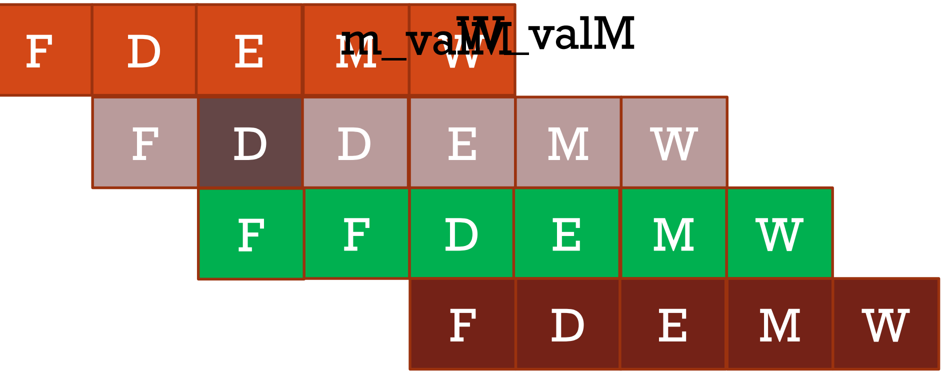
- Instructions: mrmovq, popq
- Stall 1 cycle
- Forward from _____ to _______

mrmovq (%esi), %rax

addq %rax, %rbx

addq %rax, %rcx

addq %rax, %rdx



Time

PIPELINED Y86 IMPLEMENTATION

- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

CONTROL DEPENDENCIES

- Unconditional jumps and procedure calls
  - Hazard? (e.g. jmp foo, call bar)
- Conditional jumps
  - Hazard? (e.g. jle foo)
  - What is the problem?
  - At what stage does the CPU know the answer?
- Return
  - Hazard? (e.g. ret)
  - What is the problem?
  - At what stage does the CPU know the answer?

BRANCH PREDICTION

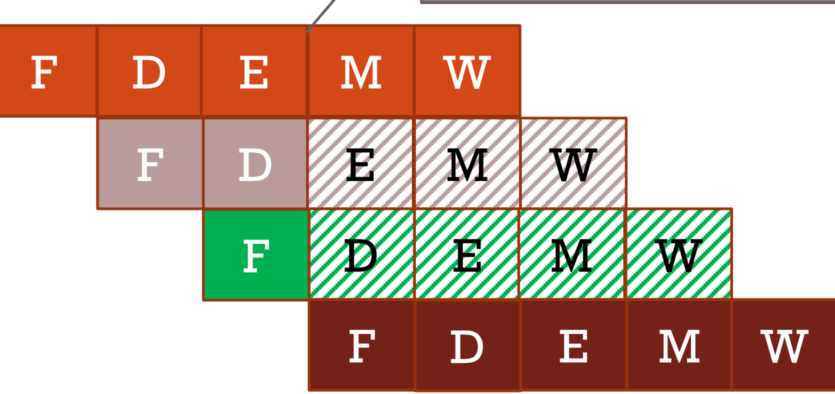
- Problem:
  - We won't know whether or not to jump until end of Execute stage
  - In the fetch stage (i.e. when setting the next PC) we don't know if we should use valP (branch not taken) or valC (branch taken)
- Idea:
  - Guess whether or not branch taken
  - Start speculative execution of instructions
- If guess was wrong
  - Shoot down all speculative instructions by turning into bubbles
  - No instruction reaches a state modification stage (M/W) before we shoot them down, so no lasting effect

BRANCH PREDICTION

- For conditional jumps:
  - No bubble if guess right
  - Two bubbles if guess wrong

```
0x100          jle $0xbadbeef
0xbadbeef      addq %rax, %rbx
0xbadbeef+2    addq %rax, %rcx
0x109          irmovq $0, %rax
```

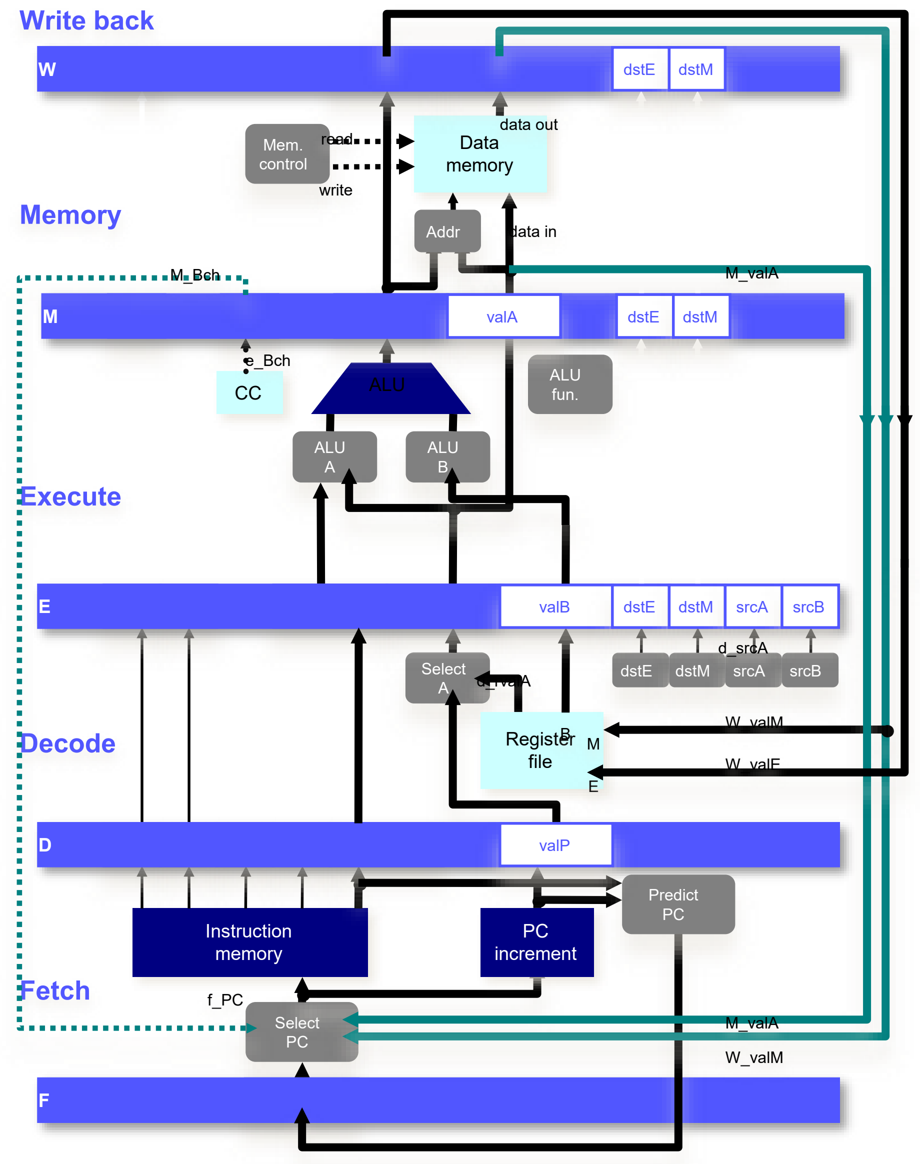
Now we know the guess was wrong



Time

PREDICTING THE NEXT PC IN FETCH STAGE

- unconditional jumps (call, jmp)
  - predPC = valC, available in F
- conditional jumps
  - know if branch taken in E (bch)
  - jump prediction in F
    - e.g., predPC = valC (i.e., taken)
  - misprediction control hazard
    - handled by "Select PC" in F
    - feedback from M
  - no stall if prediction is correct
- return from procedure call
  - know target in M
    - feedback from W to "Select PC" in F
  - y86 stalls 3 cycles



IS IT JUMP PREDICTION OR GUESSING?

- Are both jump directions (taken, not taken) equally likely?
  - If so might as well just guess at random.
- Consider how jumps are used in a program:
  - loops:
    - continue condition at bottom of loop is normally taken.
    - exit condition at top of loop is normally not taken.
  - ifs:
    - if testing for error condition, error handling code normally skipped.
    - if testing for recursion base case, base case normally skipped.

OBSERVATIONS

- Most jumps tend to go one way more often
  - Taken or not taken for that line of code predominates
  - Supported by empirical evidence
- If we can figure this out we can improve a program's performance
  - By guessing the direction taken more often

WHAT THE COMPILER KNOWS

- In many cases the compiler can make a good prediction because:
  - It creates the code
  - For loops it knows if the jump means exit or continue
  - *Might* be able to spot error test for ifs
  - Only has the program text to make decisions on
  - Cannot (usually) use dynamic information from execution

WHAT COMPILER WANTS FROM ISA

- Conditional jumps to be predicted one way or the other
- If ISA rules are explicit, compiler can adjust code
  - Compiler can change type of jump based on its prediction
  - Example: convert
    ```
    if (a < b) c = 1; else d = 2;
    ```
    to
    ```
    if (a >= b) d = 2; else c = 1;
    ```
- Complicated prediction rules are OK as long as they are well-defined

EXAMPLE OF COMPLEX PREDICTION

- Example: predict *backward* jumps are taken and *forward* jumps are not taken
  - backward jumps (that just go a short distance) are almost always loop-continue jumps
  - so they will be mostly taken
  - forward jumps could be anything, so ISA might predict not taken to add flexibility
- what's most important is that the compiler knows what the processor will do

DYNAMIC JUMP PREDICTION

- Sometimes the compiler can't tell, but jump still has a strong tendency one way or another
  - e.g., it's hard for compilers to tell which if branch tends to be executed
- Dynamic Jump Prediction is done by CPU Hardware
  - this is a type of jump where the compiler does not know what will happen
  - hardware bases its prediction on past behaviour for that line of code

IMPLEMENTING DYNAMIC JUMP PREDICTION

- CPU hardware maintains an on-chip cache of recent jump results
  - Caches jump address (key) to bch (value)
- When jump is in Execute stage (bch is computed) it updates this cache
- When jump is in Fetch state, use history as basis for prediction
  - Example: if last jump was taken, assume it's taken again

IMPLEMENTING DYNAMIC JUMP PREDICTION

- Better implementation: remember not just the last execution, but the last few results
  - Example: store branch history as a sequence of bits with 1 indicating taken
  - e.g., 1010 means the jump's recent taken history was: no, yes, no yes
- Use the history as predictor to future jumps
  - Option 1: use the count of the majority of bits as prediction
  - Option 2: try to observe a pattern
  - Trade-off: more complex predictor provides better result, but is slower

DYNAMIC PREDICTION & COMPILER

- Compiler needs to know if jump prediction is used and how the decision is made
  - Compiler can change code to make better use of predictor
- One option
  - One set of branch instructions that use static prediction
  - One set that uses dynamic prediction
- Another option: delay slots
  - Instructions that execute regardless of jump direction
  - Found in older RISC processors (e.g., MIPS)

PIPELINED Y86 IMPLEMENTATION

- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

INDIRECT JUMPS

- Y86-64 only has direct jumps (destination in instruction)
  - Special case: return (discussed later)
- Indirect jumps are common
  - Indirect jump: jmp D(%rax)
    - PC ← D + R[%rax]
  - Double indirect jump: jmp *D(%rax)
    - PC ← M[D + R[%rax]]
  - necessary to support modern programming languages (e.g., polymorphic dispatch in OO languages)

PROBLEM

- Unlike direct jumps, we do not know the target address in Fetch
  - for indirect we know it at the end of Execute
  - for double-indirect we know it at the end of Memory
  - Consequently, we cannot predict it in Fetch, too many targets
- This is really bad
  - virtually every procedure call in object-oriented language is double-indirect

INDIRECT JUMPS: RETURN

- Problem
  - Can't "guess" where return moves to
  - Must read return address from memory
  - but that doesn't happen until M
- Result
  - leads to seemingly inevitable 3 bubbles for each return
  - *returns are common*, because procedures are small
  - a good idea for program readability, but with bad performance consequences

SOLUTION

- Maintain a small stack of return addresses on the CPU Chip
  - Add address on call instruction
  - Retrieve address on return instruction
  - Run retrieved address speculatively
- note that this stack has a finite size, so return address may not be there
  - in this case, we follow the old procedure and stall three times

POLYMORPHIC DISPATCH

- To generate code for the call object.method() (e.g. b.foo())
  - object contains pointer to table with method addresses
  - Index into the table is determined by the method name
  - Assuming %rax is object address, D is method index:
    - mrmovq (%rax), %rbx
      // %rbx is now table address
    - call *D(%rbx)
      // jump to method address stored in table

```
class Base {
   void foo () { ... }
   void bar () { ... }
}

class Sub extends Base {
   void foo () { ... }
}

void zot (Base b) {
   b.foo ();
}
```

PREDICTION FOR INDIRECT JUMPS

- Key observation
  - while compiler does not actually know the class of object, it knows only that the object implements the static type of the variable
  - many variables tend to store objects of the same type over time
- Key idea
  - the hardware (or Java Virtual Machine) remembers object type and target address used the last time call was executed. If next call is to same type, it uses cached target address instead of reading it from memory.
- Implementation
  - maintain a cache of call instructions with address of procedure last called (ID a call instruction by its address)
    - predict that call will call this same procedure again when the same call instruction is executed

PIPELINED Y86 IMPLEMENTATION

- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

PERFORMANCE ANALYSIS

Don't count bubbles

- cycles per instruction (CPI)
  - measures pipeline efficiency
  - Affects effective throughput
- `CPI = totalCycles / instructionRetiredCycles`
  `= 1 + lp + mp + rp`
- bubble penalties (for implementation with data forwarding and branch predition):
  - load/use `lp` = 1 bubble * prob. of occurrence
  - branch misprediction `mp` = 2 bubbles * prob. of occurrence
  - return `rp` = 3 bubbles * prob. of occurrence

PERFORMANCE ANALYSIS EXAMPLE

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Penalty
load/use	lp	25%	20%	1	?
mispredict	mp	20%	40%	2	?
return	rp	2%	100%	3	?
total					?

```
CPI = ?
```

CPSC 313 2019 WT2 © 2020 Jonatan Schroeder

LIMITS TO PIPELINE DEPTH

- Goal is improved effective throughput
  - retired instructions / second
  - 1 / (CPI * clock-period)
- Are deeper pipelines better?
  - Pros
  - Cons

POINTERS AND DYNAMIC ALLOCATION

Unit 3

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

LEARNING GOALS

- At the end of this module, you will be able to:
  - Explain how a C program uses memory
  - Write simple C functions that allocate and deallocate memory dynamically
  - Recognize and fix common errors related to dynamic memory allocation
  - Compare and contrast the implementation issues that must be addressed by every memory allocator
  - Compare and contrast the possible placement algorithms used by memory allocators
  - Describe memory allocator requirements, the trade-offs between them, and how allocators that maintain implicit, explicit and segregated free lists meet these requirements
  - Contrast internal and external fragmentation, and explain briefly how they occur, as well as common techniques employed by memory allocators to minimize their impact

MODULE SUMMARY

- C, memory addresses and pointers
- Dynamic memory allocation
- Common programming mistakes with pointers
- How a process' memory is organized
- A look at the runtime stack
- Implementing a dynamic memory allocator

REVIEW: ADDRESSES AND POINTERS

- What is the value stored in each variable below?

```
int *a, b;
int **c;
…
d = &c;
x = a + 7;
y = c[3];
```

REVIEW: ADDRESSES AND POINTERS

- What does the following C function print?

```
void do_something() {
  char mtl[5];
  char *qc, **cdn = &qc;

  mtl[0] = 'Y';
  mtl[4] = '\0';
  qc = mtl + 2;
  *qc = 'A';
  qc[-1] = 'P';
  *cdn = qc + 1;
  **cdn = 'Z';
  *(qc - 1) = 'E';
  printf("%s\n", mtl);
}
```

REVIEW: ADDRESSES AND POINTERS

- Rewrite the following piece of code using arrays to make it more readable.

```
int confusing(long *p, int n) {
  long *q = p + n;
  while (n > 0 && *p++ == *--q) {
    n -= 2;
  }
  return n <= 0;
}
```

STRINGS IN C

- A string in C is an array of characters (bytes)
  - end of the string is indicated by byte with value 0
  - P.S.: Note that 0 (null) and ‘0’ (digit zero) are different values
- Every string has:
  - array size: size of the array that contains the string
  - maximum length: array size minus one
  - current length: position of the first null byte
- Standard C library has many operations on strings
  - strlen(s) returns the length of a string

MEMORY ALLOCATION

- Allocation: assigning a memory location to store a variable's value
  - Associating the variable to the address of that location
- Global variables: exist before program starts
  - Compiler allocates variables statically (constant address)
  - No dynamic computation required for allocation, they just exist

STATIC ARRAY ACCESS

- Compiler doesn't know address of `a[i]`
  - Unless it knows the value of `i` statically
- Array access is computed from base and index
  - Address of element is **base** plus **offset**
  - **Offset** is **index** times size of each element
- For global arrays:
  - The base address and element size are static
  - The index value is dynamic (`i`'s value can change)

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

PROCESS MEMORY ORGANIZATION

- When a program is executed:
  - Space is allocated for the shared libraries it needs
  - Instructions and initialized data are loaded in memory
  - Space is reserved for uninitialized data
- The stack and the heap are set up
  - The stack is managed by the compiler's code
  - The heap is managed by the user's program
- On a Linux system, we can look at `/proc/`*`pid`*`/maps` to the how memory is used for process *pid*.

PROCESS MEMORY EXAMPLE

```
00400000-00401000 r-xp 00000000 08:08 5244791            /home/patrice/fib
00600000-00601000 r--p 00000000 08:08 5244791            /home/patrice/fib
00601000-00602000 rw-p 00001000 08:08 5244791            /home/patrice/fib
7f86a3122000-7f86a32dd000 r-xp 00000000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a32dd000-7f86a34dd000 ---p 001bb000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a34dd000-7f86a34e1000 r--p 001bb000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a34e1000-7f86a34e3000 rw-p 001bf000 08:02 1308509    /lib/x86-64-linux-gnu/libc-2.19.so
7f86a34e3000-7f86a34e8000 rw-p 00000000 00:00 0
7f86a34e8000-7f86a35ed000 r-xp 00000000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a35ed000-7f86a37ec000 ---p 00105000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a37ec000-7f86a37ed000 r--p 00104000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a37ed000-7f86a37ee000 rw-p 00105000 08:02 1308538    /lib/x86-64-linux-gnu/libm-2.19.so
7f86a37ee000-7f86a3811000 r-xp 00000000 08:02 1308526    /lib/x86-64-linux-gnu/ld-2.19.so
7f86a39e5000-7f86a39e8000 rw-p 00000000 00:00 0
7f86a3a0e000-7f86a3a10000 rw-p 00000000 00:00 0
7f86a3a10000-7f86a3a11000 r--p 00022000 08:02 1308526    /lib/x86-64-linux-gnu/ld-2.19.so
7f86a3a11000-7f86a3a12000 rw-p 00023000 08:02 1308526    /lib/x86-64-linux-gnu/ld-2.19.so
7f86a3a12000-7f86a3a13000 rw-p 00000000 00:00 0
7fffc88a3000-7fffc88c4000 rw-p 00000000 00:00 0          [stack]
7fffc88cf000-7fffc88d1000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

LOCAL VARIABLES AND THE STACK

- Scope
  - Local variables are only accessible within declaring procedure
  - Each execution has its own private copy
- Lifetime
  - Allocated when procedure starts
  - “Freed” when procedure returns (in most languages)

PROCEDURE ACTIVATION

- Activation: execution of a procedure
  - Starts with procedure is called, ands when it returns
  - There can be many activations of same procedure alive at once
- Activation Frame
  - memory that stores activation's state
  - Includes local variables and arguments

ALLOCATING LOCAL VARIABLES

- Order of frame allocation and deallocation is special
  - freed in reverse order of allocation
- Simple allocation for frames:
  - Reserve big chunk of memory for all frames
  - Initial address known
  - Simple, cheap allocation: add or subtract from a pointer
- Questions
  - What data structure is this like?
  - What restriction do we place on lifetime of local variables?

RUNTIME STACK

- Stack of activation frames
  - Stored in memory, grows up from bottom
- Stack pointer
  - Stores base address (address of first byte) of current frame
- Current frame is the “top” of the stack
- First activation is the “bottom” or “base” of the stack
  - Local variables of initial function (e.g., main)

rsp

Stack Top
(current frame)

Stack Bottom
(first frame)

Code

Static data

Heap

Stack

VARIABLE ADDRESSES

- Value of the stack pointer is dynamic
- Local variables and arguments
  - Size of each frame is (usually) static
  - Offset from stack pointer is static
- Each frame is like a struct
  - Top of frame is in rsp (stack pointer)
  - Each variable in procedure is a member of the struct

WHAT IS STORED IN THE STACK?

- Local variables
- Arguments
  - Some architectures use registers for arguments
- Return address
- Other saved registers
  - Called function may change register values
  - Values that must be kept after call are saved

SOME IMPLICATIONS

- What is the value of l in foo when it is active?

```
void goo() { int x = 3; }          goo();
void foo() { int l; }              foo();
```

- What is wrong with this?

```
int *foo() {
  int l;
  return &l;
}
```

INSTANCE VARIABLES

- Variables that are an instance of a class or struct
  - Created dynamically
  - Many instances of the same class/struct can co-exist
- Java vs C
  - Java: objects are instances of non-static variables of a class
  - C: structs are named variable groups, or one of its instances
- Accessing an instance variable
  - requires a reference to a particular object (pointer to a struct)
  - then variable name chooses a variable in that object (struct)

```
Class X
  static int i;
  int j;
```

STRUCTS IN C

- A struct is a collection of variables of arbitrary type
  - allocated and accessed together
- Declaration
  - similar to declaring a Java class without methods
  - name is "struct" plus name provided by programmer
  - static: struct D d0;
  - dynamic: struct D* d1;
- Access:
  - static: d0.e = d0.f;
  - dynamic: d1->e = d1->f;

```
struct D {
  int e;
  int f;
}
```

≈

```
class D {
  public int e;
  public int f;
}
```

STRUCT ALLOCATION

```
struct D {
  int e;
  int f;
}
```

- Static structs are allocated by the compiler

```
struct D d0;
```

Static Memory Layout

```
0x1000: value of d0.e
0x1004: value of d0.f
```

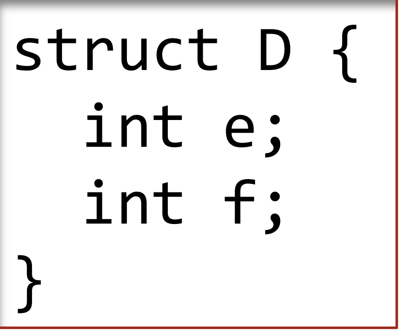
- Dynamic structs are allocated at runtime
  - variable that stores struct pointer may be static or dynamic
  - struct itself is allocated with a call to malloc

```
struct D* d1;
```

Static Memory Layout

```
0x1000: value of d1
  (address of instance)
```

STRUCT ALLOCATION (CONT.)




```
void foo() {
  d1 = malloc(sizeof(struct D));
}
```

- Example: assume malloc returned 0x2000

Static Memory Layout

```
0x1000: 0x2000 # d1
…
0x2000: value of d1->e
0x2004: value of d1->f
```

DYNAMIC ALLOCATION IN C AND JAVA

- Programs can allocate memory dynamically
  - allocation reserves a range of memory for a purpose
- In Java, instances of classes are allocated by the new statement
- In C, byte ranges are allocated by call to malloc procedure
  - these bytes can be used for any type that can fit in them

DYNAMIC ALLOCATION: USAGE EXAMPLES

- Example: array that grows as data grows
  - When adding an element, if array is full, allocate bigger space and copy old data to new space
- Example: linked list
  - Each element is a struct that contains a pointer to next element
  - More details: CPSC 221

DYNAMIC ALLOCATION IN C

- Memory allocation
  - void* malloc(int n);
  - n is the number of bytes to allocate
  - returning type is void*
    - pointer to anything (no specific type assigned)
    - can be cast to/from any other pointer type
    - cannot be dereferenced directly
- Use sizeof to determine number of bytes to allocate
  - struct Foo* f = malloc(sizeof(struct Foo));
  - statically computes number of bytes in type or variable

MEMORY DEALLOCATION

- Wise management of memory requires deallocation
  - Memory is a scarce resource
  - Deallocation frees previously allocated memory for re-use
- In Java:
  - Garbage collection: memory is deallocated when no longer in use
  - Requires keeping track of every reference to an object
- In C:
  - Dynamic memory must be deallocated explicitly by calling free
  - Memory is deallocated immediately, no checks if it's still in use

MEMORY HEAP

- The heap is a large section of memory from which malloc allocates objects
  - malloc finds unused space in the heap, marks as used and returns it
  - free marks space as unused
  - heap may be increased if necessary
- All objects are stored in the heap

PROBLEMS WITH EXPLICIT DEALLOCATION

- What free(x) does
  - deallocates "object" at address x
  - this memory can be reused by subsequent call to malloc
- What free(x) does not do
  - after all call to free, x still points at the freed object
  - other variables may still point there too

EXPLICIT DEALLOCATION EXAMPLE

- What bad things can happen below?

```
struct buffer *create() {
  struct buffer *buf = malloc(sizeof(struct buffer));
  …
  return buf;
}
void destroy(struct buffer *buf) {
  …
  free(buf);
}
```

DANGLING POINTERS

- A dangling pointer is a pointer to an object that is not allocated
  - Often caused by use after free
  - if another malloc has been called since last free, could point to another object
- Why is this a problem?
  - program thinks it's writing to one object, but is writing to another
  - program may be writing to object of another type

DANGLING POINTERS

- Examples of dangling pointers:
  - Uninitialized pointer
  - Multiple pointers to same location, one is freed and the other is still in use
  - Calling free on the same memory twice
  - Function returns pointer to local variable
- Good practices to avoid dangling pointers:
  - Initialize all pointers to valid data (e.g., NULL)
  - If needed, implement reference counting

MEMORY LEAKS

- A memory leak is a dynamic memory object with no pointers pointing to it
  - Usually happens if a program doesn't free object properly
  - May also happen if a pointer to valid object is changed to another value
- Why is this a problem?
  - If object had useful information, it can't be accessed anymore
  - If object is large (or if many memory leaks happen in sequence), program will use too much memory

MEMORY LEAKS

- Memory leak examples:
  - Function allocates memory, then returns without saving or returning memory
  - Function returns dynamically allocated memory, but return value is ignored
  - Last pointer to allocated space is changed to different value

OTHER MISTAKES WITH POINTERS

- Buffer overflow: using more data than allocated
  - Can be a problem with global and local arrays as well
  - Can be caused by off-by-one errors (e.g., not counting the string termination byte)
- Incorrect sizeof parameter
  - Call to malloc with sizeof must match the object you're allocating, not its pointer

AVOIDING MEMORY PROBLEMS IN C

- Avoid the program cases
  - if possible, restrict dynamic allocation/free to single procedure
  - if possible, don't write procedures that return pointers
  - if possible, use local variables instead
    - local variables are allocated on call and freed on return, automatically
- Engineer for memory management
  - define rules for which procedure is responsible for deallocation
  - use explicit reference counting if multiple potential deallocators
  - define rules for which pointers can be stored in data structures
  - use coding conventions and documentation to ensure rules are followed

DETECTING PROBLEMS: VALGRIND

- Valgrind is a program that performs dynamic analysis of the runtime of a program
  - Example: valgrind ./grade
- It runs your program and monitors dynamic allocation and deallocation
- It can tell if your program has:
  - memory leaks
  - use after free (dangling pointers)

IMPLEMENTING A DYNAMIC MEMORY ALLOCATOR

- So far we have been using malloc/calloc and free
- But how are they implemented?
  - How is available memory maintained?
  - How to keep track of freed blocks?
  - When is it ok to reuse a block?

MEMORY ALLOCATOR REQUIREMENTS

- Handling arbitrary sequences of requests
  - The allocator can not control the requests for malloc/free
- Making immediate responses to requests
  - The allocator can not wait to process several requests at once, even if it would be more efficient
- Using only the heap for its data structures
  - We can use a constant amount of additional space only
- Not modifying allocated blocks
  - The user program assumes their contents won't change
  - It also assumes its location won't change

MEMORY ALLOCATOR IMPLEMENTATION

- Implementation issues:
  - *Placement*: when malloc() is called, how to we find a free block that will be used to satisfy the request?
  - *Splitting*: if we only need part of a free block to satisfy a request, what do we do with the rest?
  - *Coalescing*: do we merge a newly free()'d block with adjacent free blocks?
- These issues arise in all implementations

MEMORY ALLOCATION GOALS

- Maximizing throughput
  - We want to respond to requests quickly
  - So we need to use simple data structures
- Maximizing memory utilization
  - We want to avoid internal and external fragmentation

INTERNAL FRAGMENTATION

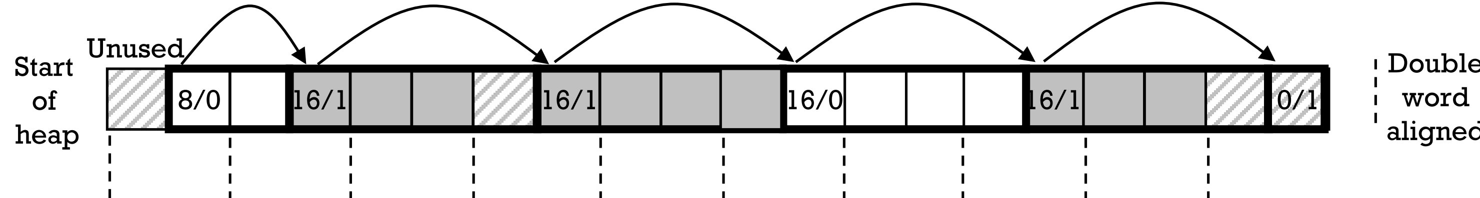
- Most allocators impose a minimum size on the blocks they return to a process
  - Because of alignment requirements (pointer addresses must be a multiple of 4, or 8)
  - Because the allocation needs to store information inside the blocks once they are freed; so the blocks must be large enough
- Hence a request for a very small amounts of memory returns a larger block than that requested
- Some space inside the block is wasted

EXTERNAL FRAGMENTATION

- The block returned by malloc() must consist of consecutive memory locations
- Sometimes there may be enough free space in total, but no free block is large enough to satisfy the request
- The more calls to malloc() and free() have been made with requests for different size blocks, the more external fragmentation is a problem

IMPLICIT FREE LIST

- A simple implementation: the implicit free list.



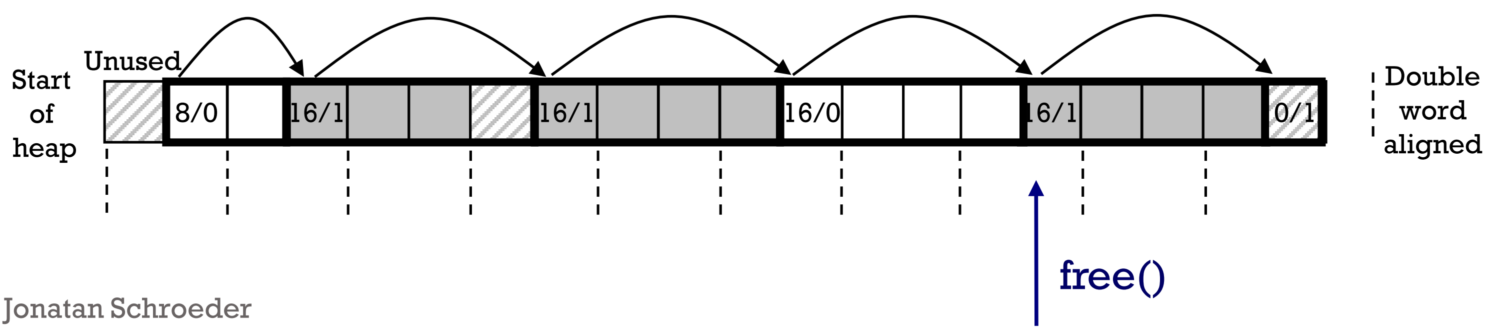
- We have a linked list of blocks.
  - The list contains both occupied and free blocks.
  - Each block contains its size.
  - Each block knows if it's free or occupied:

IMPLICIT FREE LIST: SPLITTING

- If the block used is larger than the requested size, we can
  - use the whole block (increases internal fragmentation)
  - divide the block in two (may end up with many small blocks, which can cause external fragmentation)
- Splitting policy may depend on the algorithm
  - Some options require a minimum block size

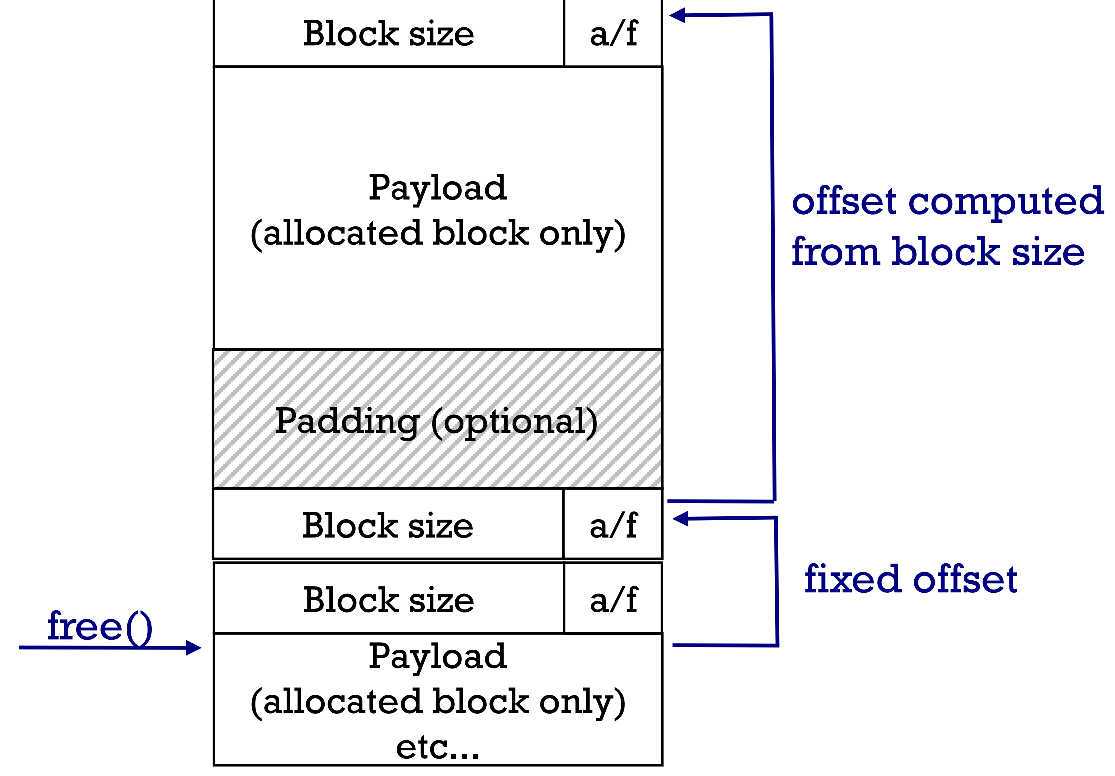
IMPLICIT FREE LIST: COALESCING

- When freeing a block, merge adjacent free blocks
- Avoids ending up with lots of small free blocks all adjacent
- We can do this on every free
  - Advantage: simpler
  - Disadvantage: free operation becomes slower
  - Alternative: wait until an allocation request fails
- Problem: how to find adjacent blocks



IMPLICIT FREE LIST: COALESCING

- We store the block size at the end of the block also:



IMPLICIT FREE LIST: PLACEMENT

- Placement:
  - *First-fit*: return the first free block that is large enough
    - retains large free block near end of the list
    - Disadvantage: search time if too many small blocks
  - *Next-fit*: similar but start searching from the last allocated block
    - Advantage: faster search time
    - Disadvantage: worse memory utilization than first-fit
  - *Best-fit*: find the free block whose size is closest to the requested size
    - Advantage: optimal use of memory
    - Disadvantage: slower

EXPLICIT FREE LIST

- *Explicit free list*: uses the payload in free blocks to point to other free blocks
  - Block search is faster (don't need to check blocks in use)
  - Doubly-linked list
  - Disadvantage: minimum payload size must be enough for two pointers
- Linked-list order (where to pointers point to)
  - Last-in First-out: free blocks go at the beginning of the list
    - free() takes constant time
  - In address order: pointers list blocks in address order
    - free() requires linear time (must search previous/next free blocks)
    - We get slightly better memory utilization (search is in memory order)

SEGREGATED FREE LIST

- *Segregated free list*: one linked list per block size
  - Blocks point to other blocks of similar size
- One approach (segregated fits):
  - malloc() finds a block large enough, splits it if desired, and inserts the other piece in the appropriate free list
  - free() coalesces with adjacent free blocks if possible, stores the new free block in the appropriate free list

SEGREGATED FREE LIST (CONT.)

- Searching for free blocks is more efficient
  - We are only searching part of the heap
- Memory utilization improves
  - First-fit search with a segregated list approximates a best-fit search of the entire heap
- The GNU malloc package, part of the standard C library on all Linux systems, uses segregated free lists

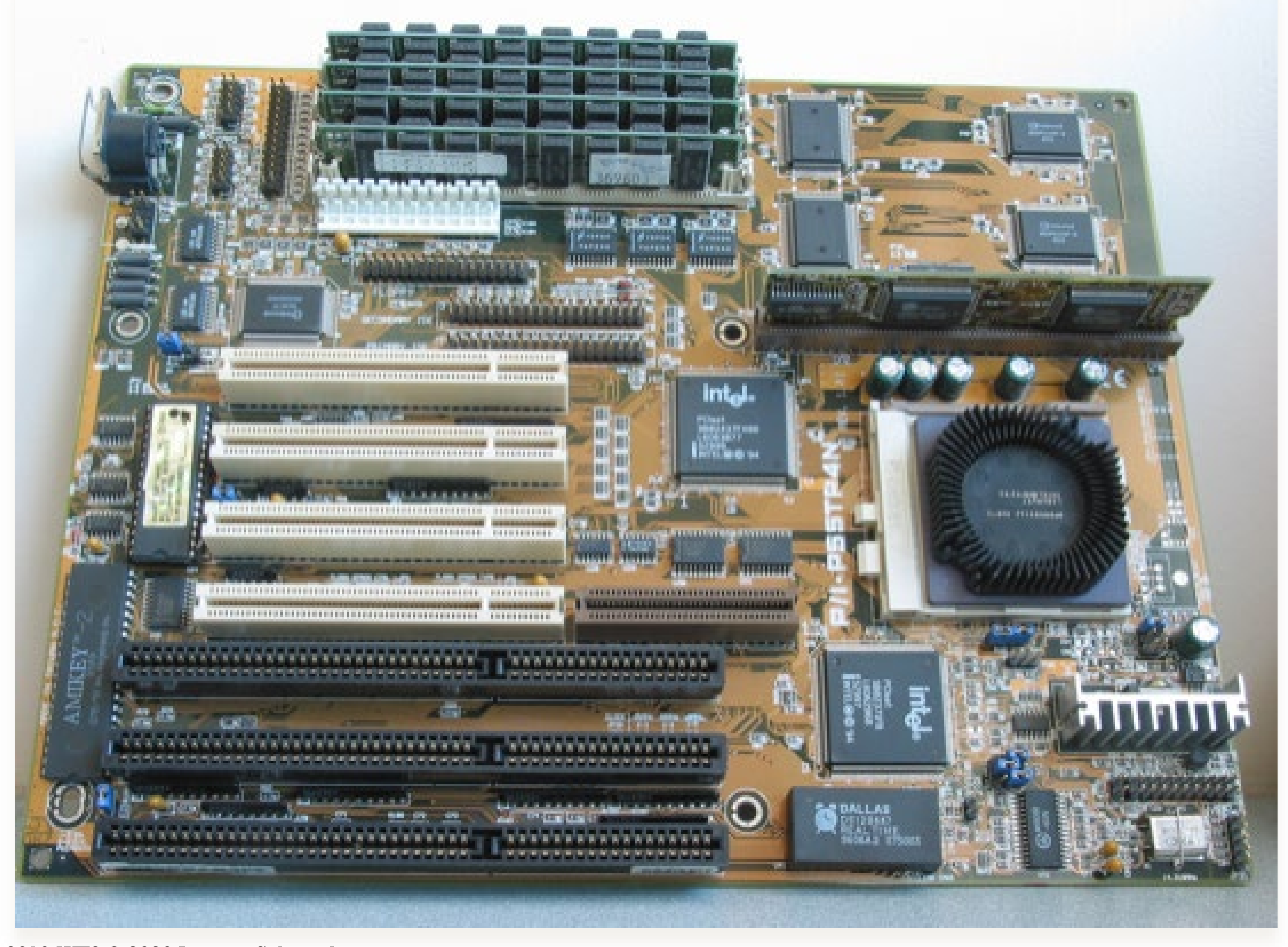
MEMORY HIERARCHY

Unit 3

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

OUTLINE

- memory, types, and the hierarchy
- locality
- cache memories
  - getting data in
  - how much data to bring in at once
  - throwing data away
  - locating data
  - handling writes
- writing cache-friendly code

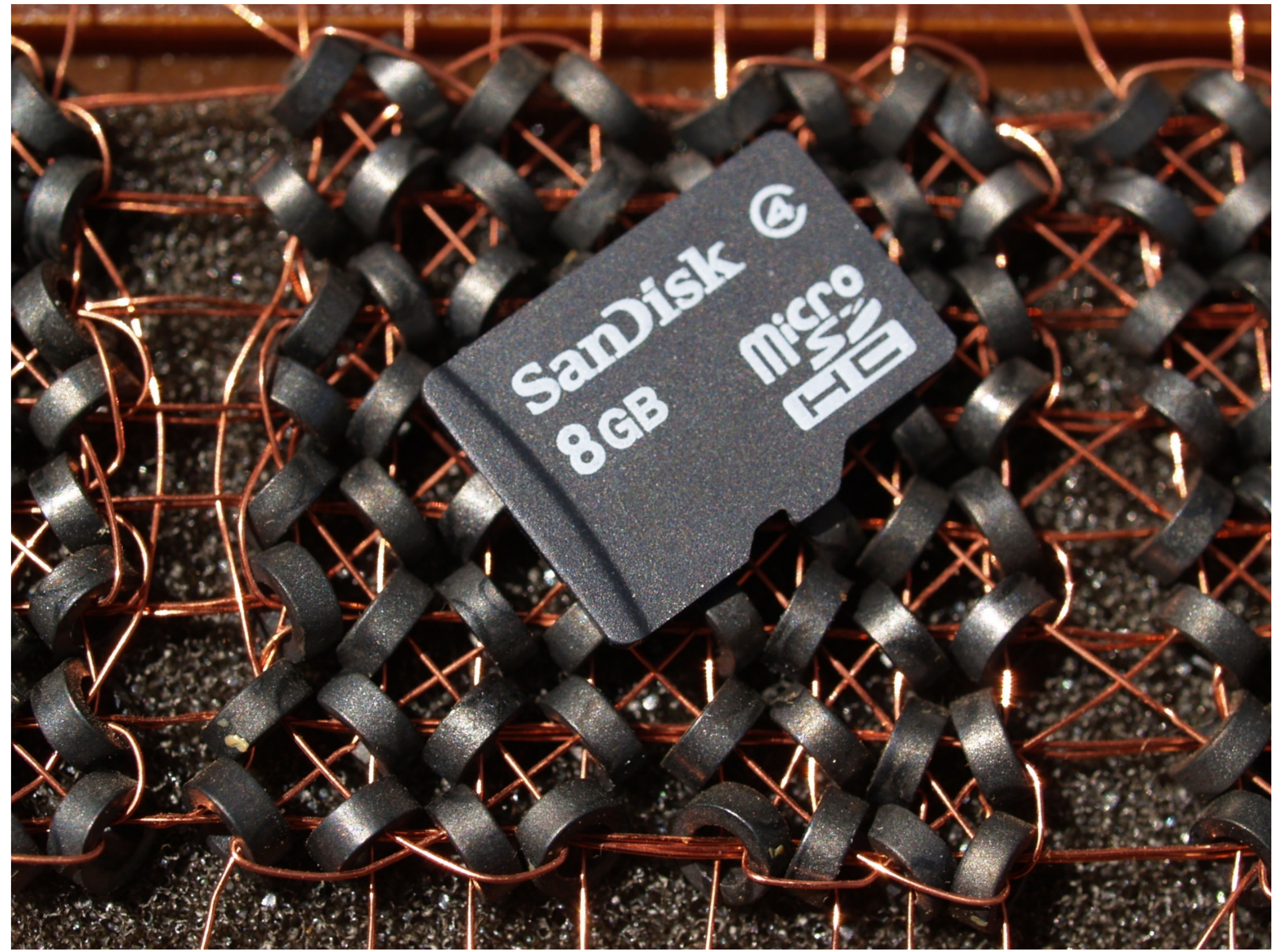

intel
intel
DALLAS
AMIKEY

MEMORY TECHNOLOGY - HISTORY

- Core memory
  - Popular from 1955-1975
  - 32 kilobits per cubic metre by the late 1960s
  - Semiconductor memory started to take over the market in the early 1970s.

https://commons.wikimedia.org/wiki/File:Magnetic-core_memory.JPG Benoitb

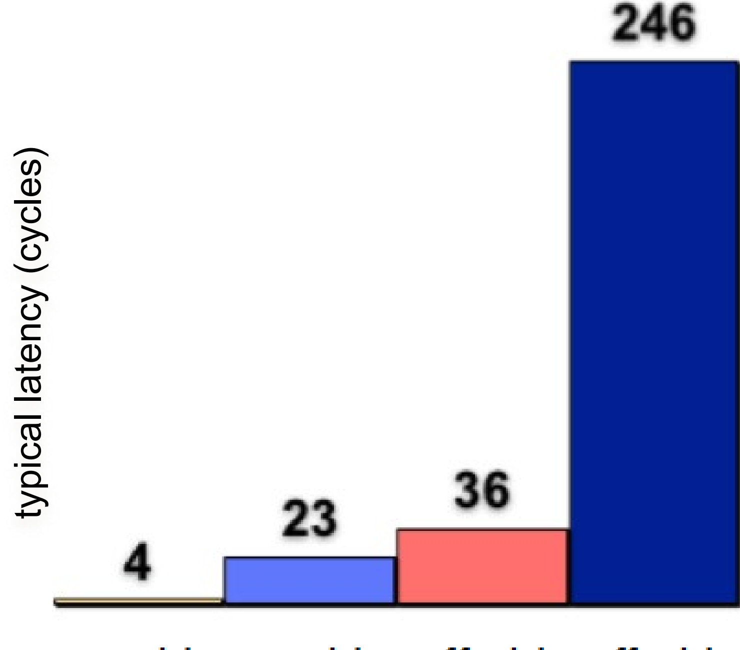
64 BITS VS 8 GBYTES



https://commons.wikimedia.org/wiki/File:8_bytes_vs._8Gbytes.jpg Daniel Sancho

STORING STUFF AND PERFORMANCE

- Static RAM (SRAM)
  - expensive (6 transistors per bit).
  - very fast (10ns access times).
  - retains its value indefinitely as long as it is powered.
  - relatively insensitive to disturbances such as electrical noise.
- Dynamic RAM (DRAM)
  - cheaper (1 transistor and 1 capacitor per bit)
  - slower than SRAM (60ns access times)
  - value must be refreshed periodically (every 10 to 100ms) or it is lost
  - relatively sensitive to disturbances
  - uses a lot less power (and generates less heat) than SRAM.



STORING STUFF AND PERFORMANCE

- Disk
  - even cheaper
  - slowest (10–15 ms access times)
  - data is only accessible in large chunks
- Other types of memory:
  - ROM, PROM, EPROM, EEPROM
  - Read-only (Static, Programmable once, Erasable, Electronically Erasable).
  - usually used for BIOS or device firmware.
- Flash
  - a type of EEPROM used in flash disks, cameras, MP3 players
  - wears out after being reprogrammed too many times

SOME RULES OF THUMB

- How big things are
  - Bigger means slower, typically also cheaper
  - If A is bigger and faster than B (for same cost), there's no point in using B
- Where things are
  - On chip is much faster
  - Off chips is slower but typically much bigger

TRENDS

100,000,000.0
10,000,000.0
1,000,000.0
100,000.0
10,000.0
1,000.0
100.0
10.0
1.0
0.1
0.0

ns

1980 1985 1990 1995 2000 2003 2005 2010

Year

Disk

SSD

DRAM

SRAM

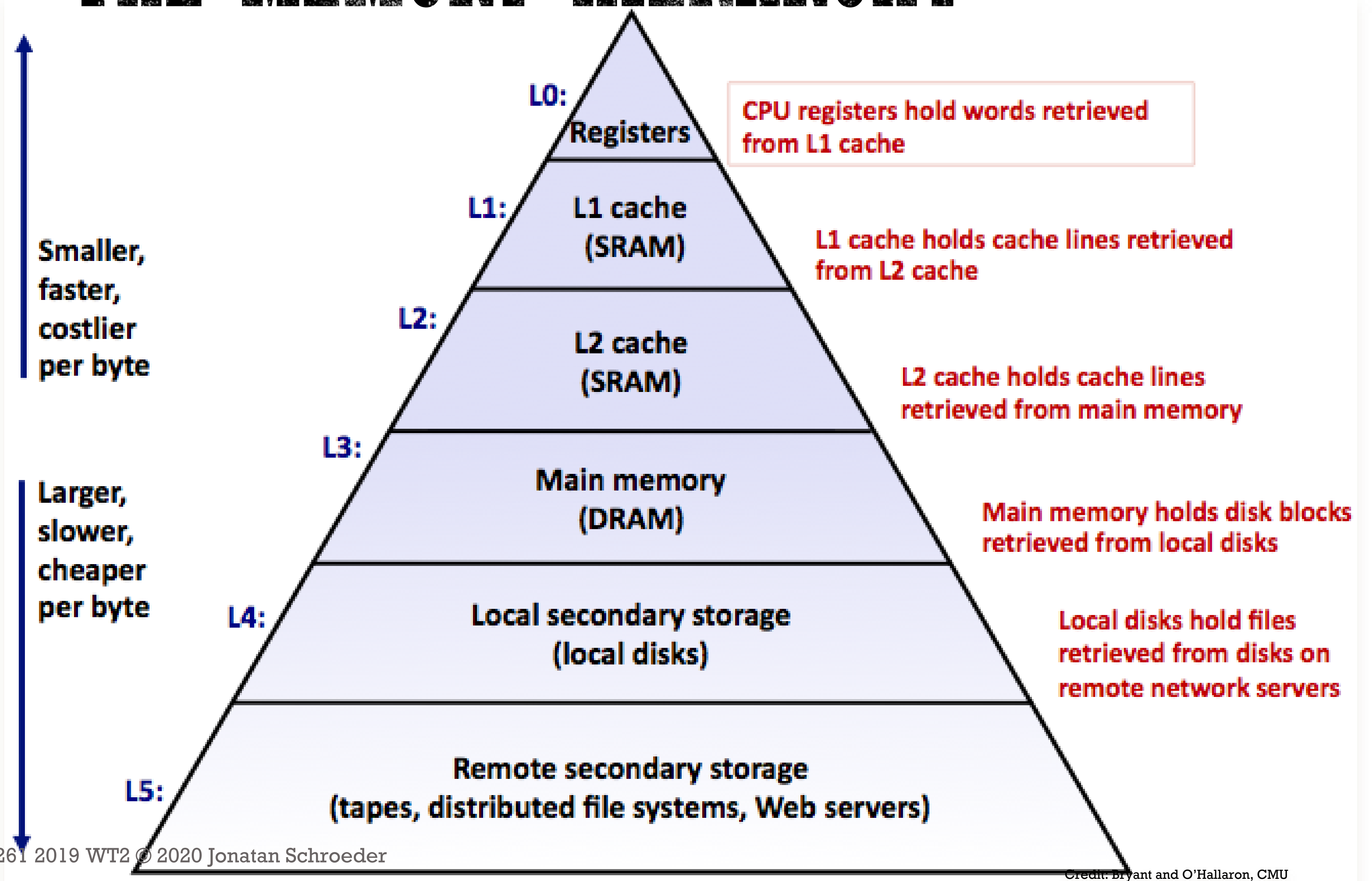
CPU

- Disk seek time
- Flash SSD access time
- DRAM access time
- SRAM access time
- CPU cycle time
- Effective CPU cycle time

MEMORY SUMMARY

- Placement
  - On-chip: high bandwidth but limited capacity
  - Off-chip: is clocked slower than CPU and consequently reduced bandwidth
- Type
  - Disks, flash disks, DRAM, SRAM
- Size
  - Bigger memory is slower, due to capacitance load and bit lines
- Challenge: can we make cheap memory look faster?

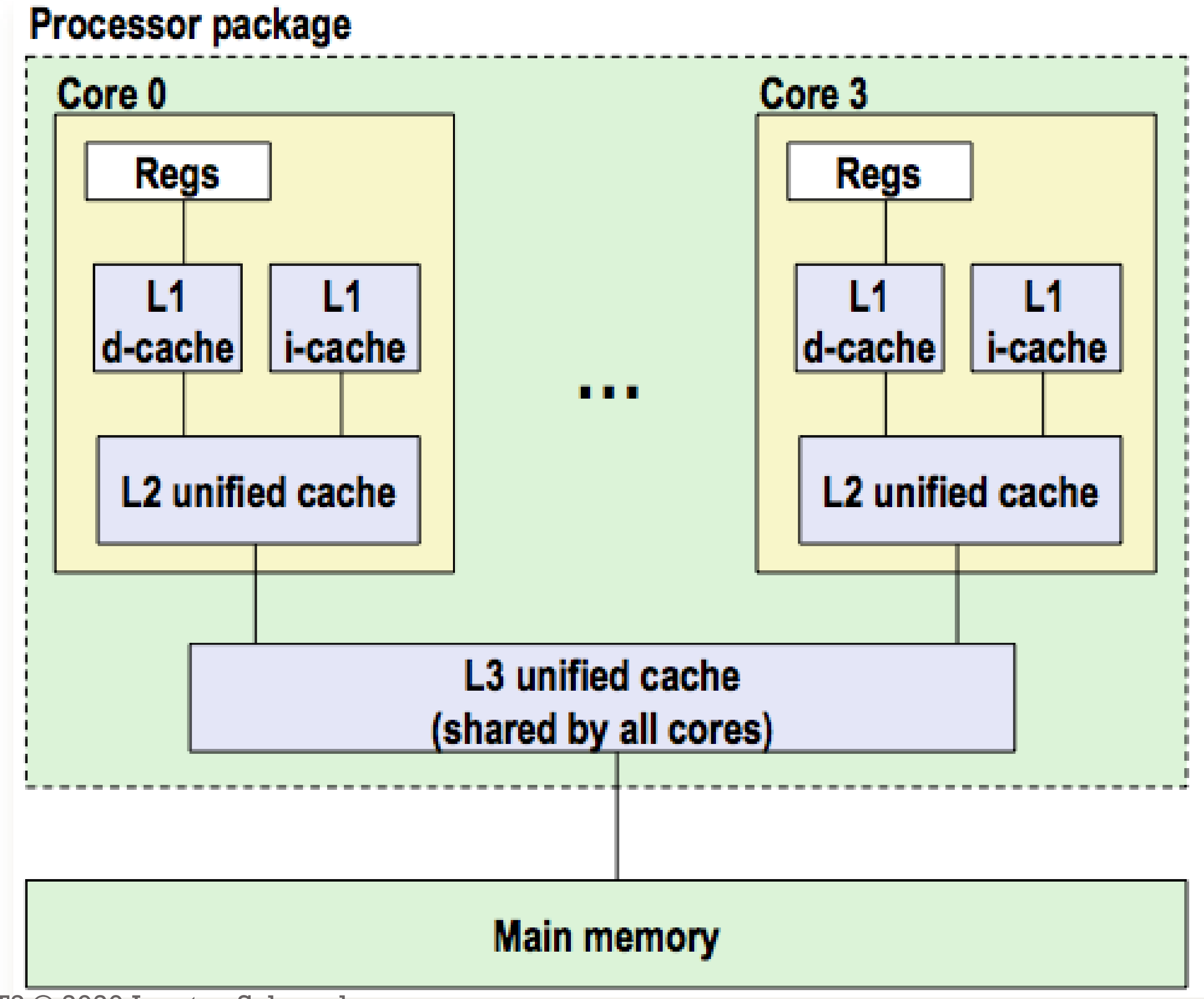
THE MEMORY HIERARCHY



CACHE MEMORIES

- Cache
  - Small storage device
  - Contains a copy of a subset of data from larger, slower storage device
  - each level acts as a cache for the next larger, slower level
  - programs access data/code at level k more often than data/code at > k
  - thus k+1 can be slower than k and so it can be bigger and cheaper
- Examples
  - The OS and applications cache/buffer disk data in main memory
  - web browsers cache web data on your local disk
  - the processor caches memory data in L1, L2, and L3 caches

INTEL CORE I7 MEMORY HIERARCHY



Credit: Bryant and O'Hallaron, CMU

USING THE MEMORY HIERARCHY

- How does data get into fast memory?
  - What naming conventions do we use?
  - What about dynamic information?
  - How long does it stay there?
- Challenge
  - make it **easy** to use this memory
    - i.e., transparent for programmers; single, simple model of memory
  - while getting the **best performance** from fast memories
    - i.e., most data access should come from the smaller, faster memories

REFERENCE LOCALITY AND CACHING

- Reference locality exists when
  - data or code accesses tend to be near each other in time or space
  - recent accesses predict future ones in a simple way
- Types of locality
  - Temporal locality: multiple accesses to same memory location within short interval
  - Spatial locality: accesses to nearby addresses within a short interval

EXPLOITING LOCALITY

```
int a[NUM_ELEMENTS];

int sumArray (int* a, int size) {
  int sum=0;
  for (int i=0; i<size; i++)
    sum = sum + a[i];
  return sum;
}
```

- Caching is a technique for exploiting locality
- Temporal locality: keep recently access data/code in fast-memory for a while
- Spatial locality: fetch accessed data/code into fast memory in multi-word blocks

LOCALITY

```
int a[NUM_ELEMENTS];

int sumArray (int* a, int size) {
  int sum=0;
  for (int i=0; i<size; i++)
    sum = sum + a[i];
  return sum;
}
```

- Data locality
  - Reference array elements in succession:
  - Reference sum each iteration:
- Instruction locality
  - Reference instructions in sequence:
  - Cycle through loop repeatedly:

DATA LOCALITY EXAMPLE

- Does this function have good locality?

```
int m[NUM_ROWS][NUM_COLUMNS];

int sumMatrix (int* m, int rows, int cols) {
  int sum=0;
  for (int c=0; c<cols; c++)
    for (int r=0; r<rows; r++)
      sum = sum + m[r][c];
  return sum;
}
```

HOW THE ARRAY IS STORED

- How are 2D arrays stored in memory? Consider the following matrix:

A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]

- If the array is stored in row-major order, the values are stored as follows:

A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]

- Or in column-major order, as follows:

A[0][0]	A[1][0]	A[0][1]	A[1][1]	A[0][2]	A[1][2]

WHAT ABOUT MATRIX SUM?

- does this code exhibit spatial locality on the array?
  - it depends on how the array is stored in memory and how it is accessed
  - Elements are accessed in column-major order

```
int m[NUM_ROWS][NUM_COLUMNS];

int sumMatrix (int* m, int rows, int cols) {
  int sum=0;
  for (int c=0; c<cols; c++)
    for (int r=0; r<rows; r++)
      sum = sum + m[r][c];
  return sum;
}
```

m[0][0]
m[0][1]
...
m[1][0]
m[1][1]
...
m[2][0]
m[2][1]
...

0
rows+0
stride
1
rows+1
2

A BETTER VERSION

- Spatial Locality is good
  - loop's stride through memory is low
  - each step accesses an element 4 bytes away, not columns*4 bytes away

```
int m[NUM_ROWS][NUM_COLUMNS];

int sumMatrix (int* m, int rows, int cols) {
  int sum=0;
  for (int r=0; r<rows; r++)
    for (int c=0; c<cols; c++)
      sum = sum + m[r][c];
  return sum;
}
```

```
m[0][0]
m[0][1]
...
m[1][0]
m[1][1]
...
m[2][0]
m[2][1]
...
```

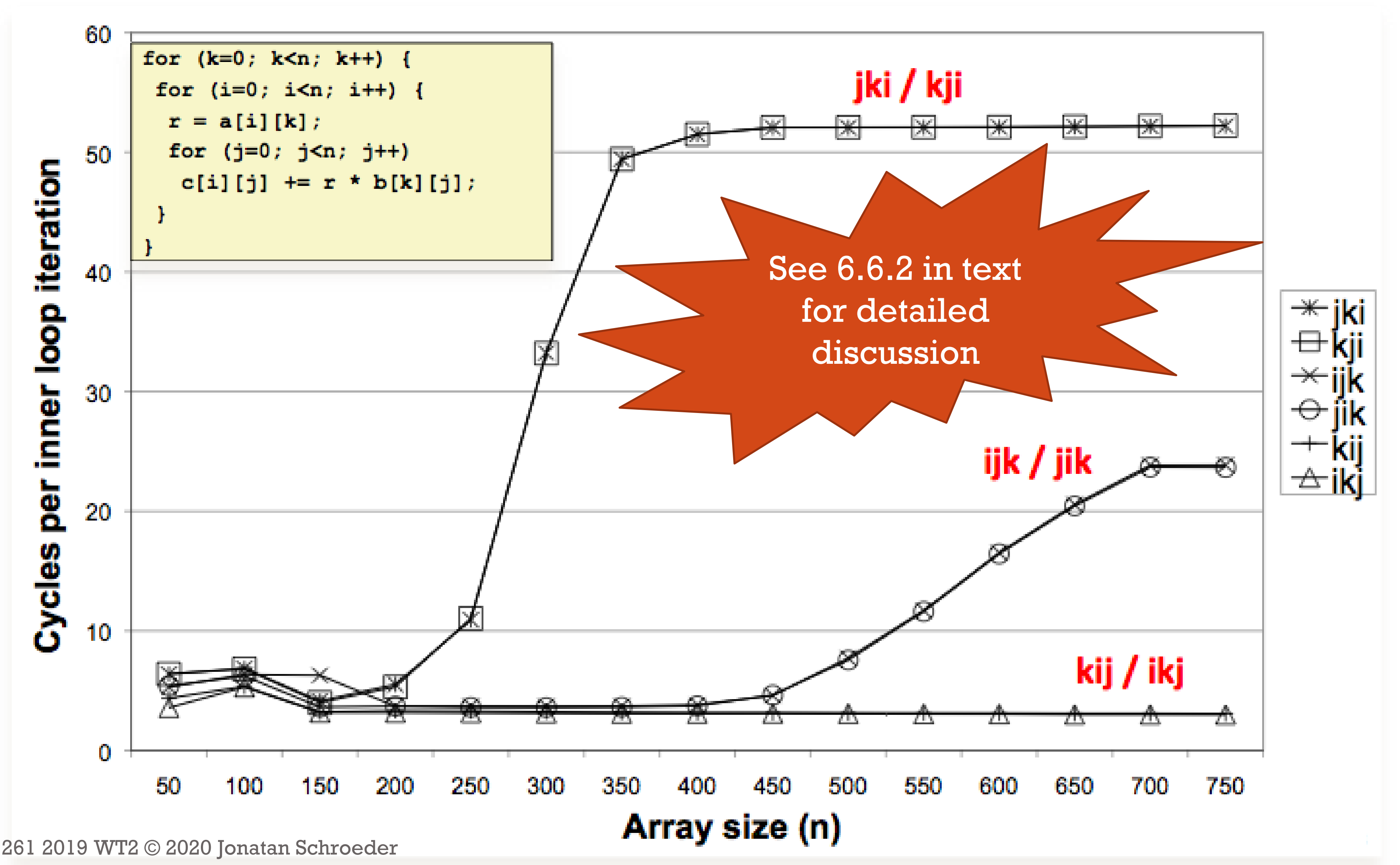
0
stride
1
cols+0
cols+1

MATRIX MULTIPLICATION

- Can we do better than this?

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

ACCESS ORDER AND PERFORMANCE



Credit: Bryant and O'Hallaron, CMU

DOES CACHING WORK?

- It depends on workload
  - some workloads exhibit lots of locality and other's don't
- Amdahl's Law
  - for cache to matter, it must improve performance for part of program that executes most of the time
  - factor S speedup of code that runs for x% of total time of program
- problems
  - some workloads exhibit little locality
  - some workloads with locality are too big to fit in cache

AMDAHL'S LAW AS APPLIED TO CACHES

- Amdahl's law
  - Suppose we speed up some part of a system, in this case through caching
  - Overall speedup is then a function of the amount of speedup and the amount of time that that part of the system executes for
  - Suppose system takes T_{old} time to execute and that part of the program takes α proportion of time and is sped-up by a factor of k
  - New time after speed up is

$$T_{new} = (1-\alpha)T_{old} + \frac{\alpha T_{old}}{k} = T_{old}\left((1-\alpha) + \frac{\alpha}{k}\right)$$

$$\text{Speedup} = \frac{T_{old}}{T_{new}} = \frac{1}{(1-\alpha) + {}^{\alpha}/_{k}}$$

EXAMPLE

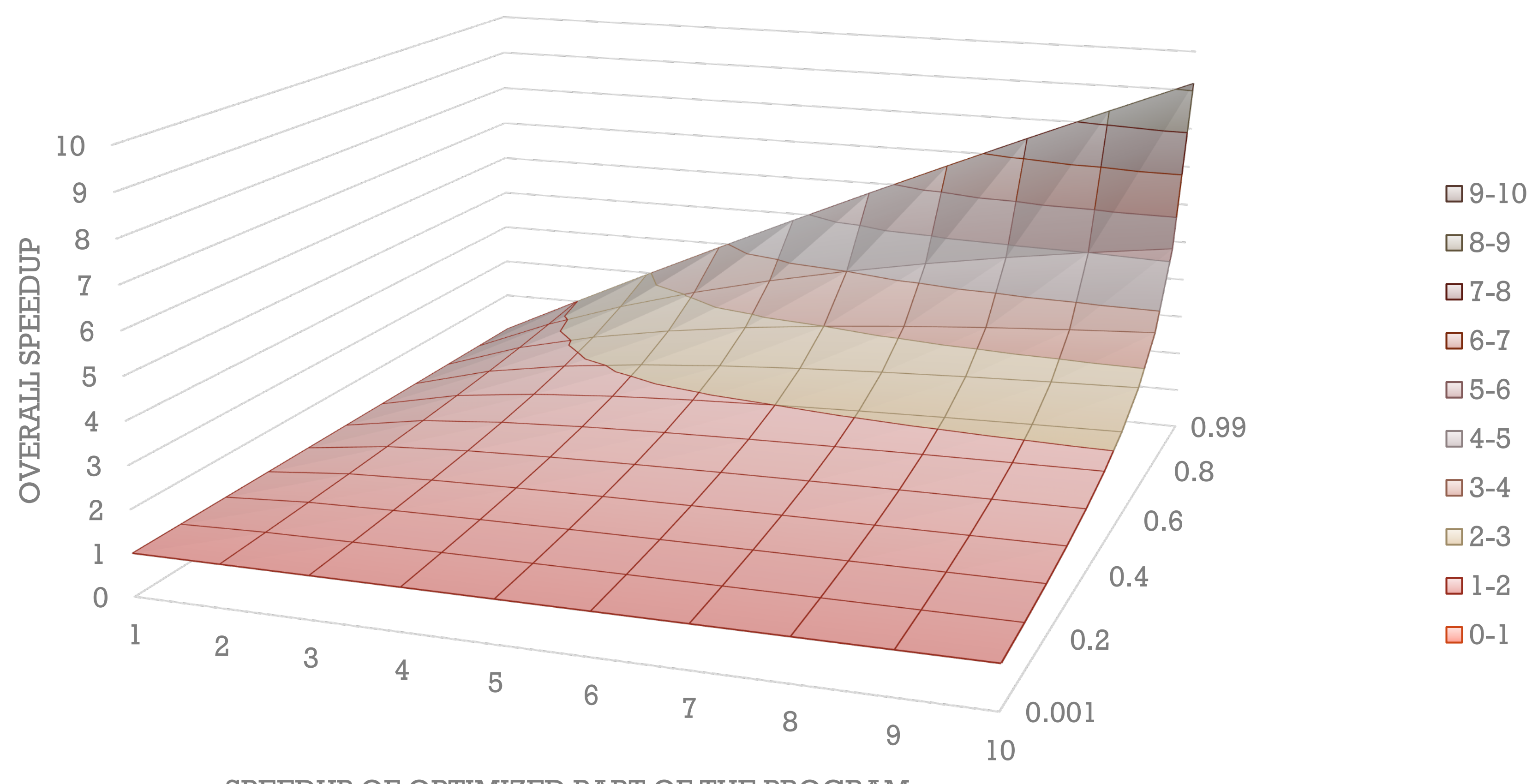
- Suppose a piece of the code that is run 10% of the time has its performance improved by 100 times:

$$\frac{1}{(1-\alpha)+\frac{\alpha}{k}} = \frac{1}{(1-.1)+\frac{.1}{100}} = \frac{1}{.9+.001} = \frac{1}{.901} \cong 1.11$$

- Suppose code runs 95% of the time is made 4 times faster ($\alpha = .95, k = 4$)

$$\frac{1}{(1-\alpha)+\frac{\alpha}{k}} = \frac{1}{(1-.95)+\frac{.95}{4}} \cong \frac{1}{.05+.238} \cong \frac{1}{.288} \cong 3.48$$

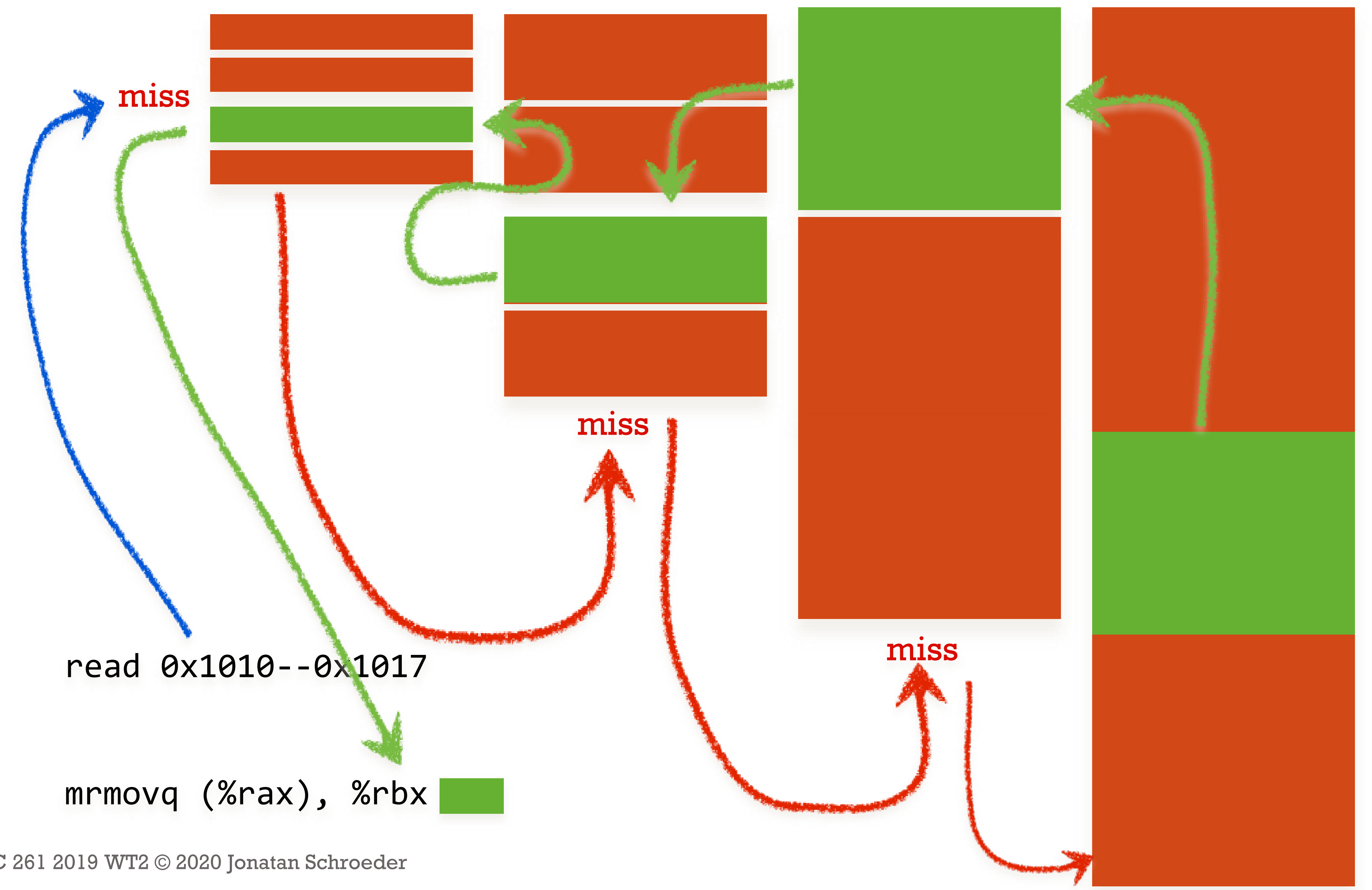
PLOT OF AMDAHL'S LAW



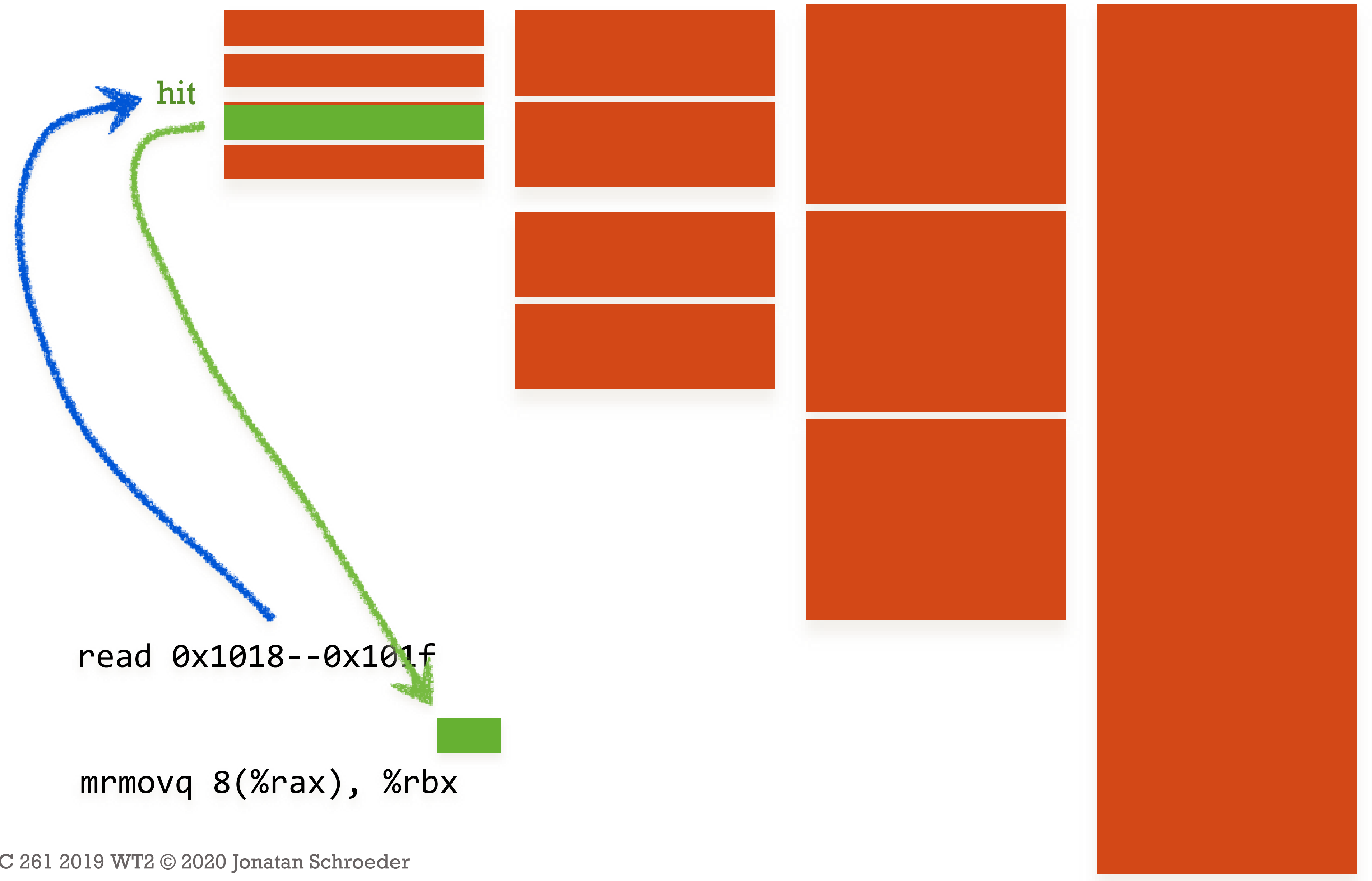
ISSUES FOR CACHE DESIGN

- getting data in
- how much data to bring in at once
- locating data
- throwing data out
- handling writes
- caching instructions

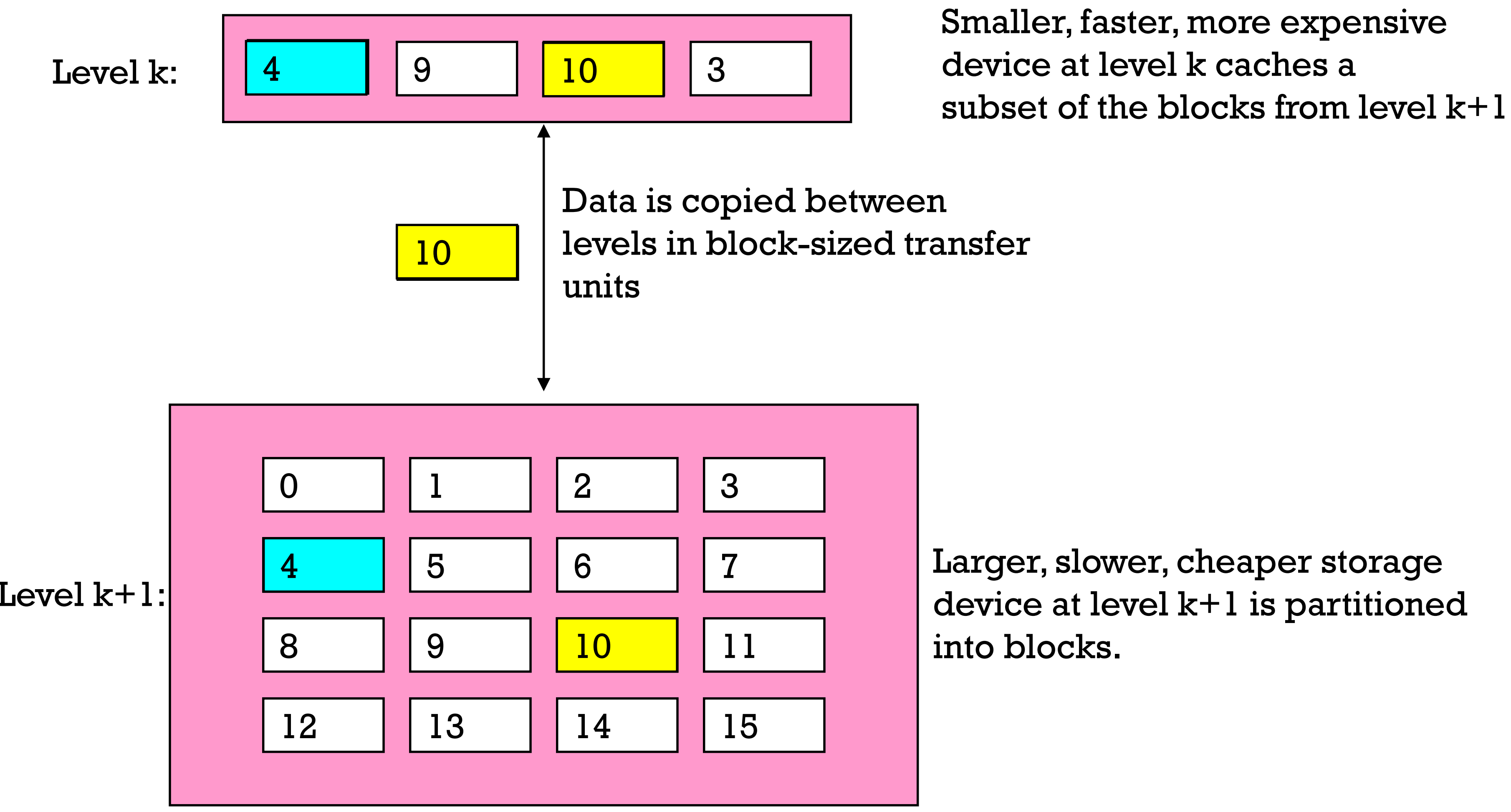
EXAMPLE: READ ACCESS MISS



EXAMPLE: READ ACCESS HIT

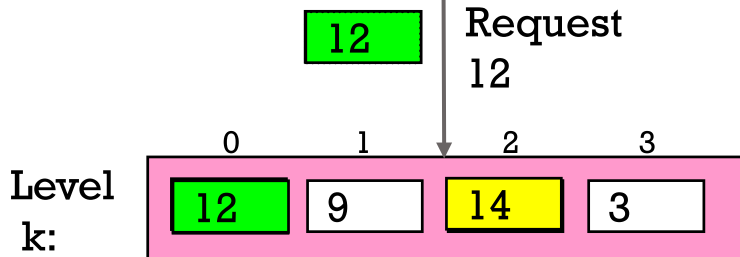


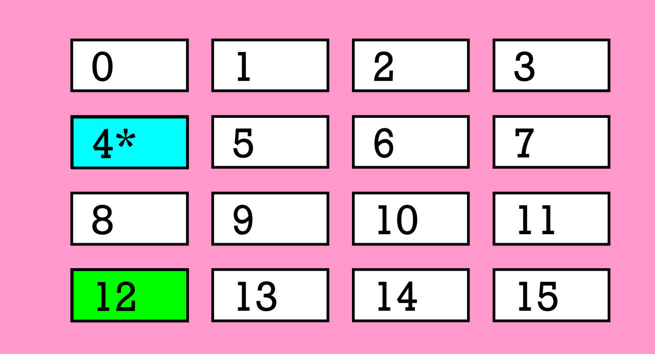
CACHING IN A MEMORY HIERARCHY



CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

GENERAL CACHING CONCEPTS





- Program needs object d, which is stored in some block b.
- Cache hit
  - Program finds b in the cache at level k. E.g., block 14.
- Cache miss
  - b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
  - If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
    - Placement policy: where can the new block go? e.g., b mod 4
    - Replacement policy: which block should be evicted? e.g., LRU

SUMMARY: CACHE HIT OR MISS

- cache hit
  - is an access that can be satisfied by a cache
- cache miss
  - is an access that cannot be satisfied by a cache
- handling a miss
  - propagate down the hierarchy toward main memory
  - stops at hit in "lower-level" cache or main memory
    - "lower-level" here means lower in the hierarchy (slow on bottom, fast on top) so L2 is "lower" than L1
  - propagate data back up hierarchy to CPU
  - (usually) add data to each cache on the way

TYPES OF CACHE MISSES

- Types of cache misses:
  - Cold (compulsory) miss
    - Cold misses occur because the cache is empty
  - Conflict miss
    - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same cache line
    - e.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time if both 8 and 0 mapped to the same line
  - Capacity miss
    - Occurs when the set of active cache blocks (working set) is larger than the cache

BLOCK SIZE

- cache block is
  - granularity of data transfers into some cache
  - power-of-two number of bytes (why?)
  - static partitioning of physical address space
- in the cache hierarchy
  - typically higher-level caches (faster/smaller) have smaller blocks
- picking block size is a design tradeoff
  - increasing size improves ______________________
  - decreasing size improves ______________________

LOCATING DATA

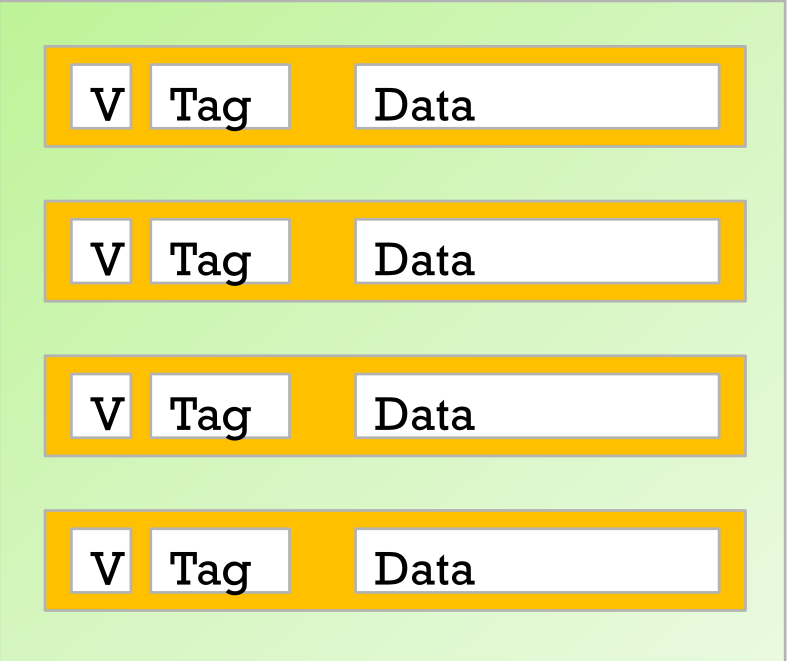
- naming
  - data is named by its numeric physical address
  - examples: condition codes, registers, main memory
- data is located using its name (address)

DIRECT-MAPPED ADDRESSING

- Data item is located
  - directly using bits of its address as an index
  - address index names a unique location in memory
  - fastest way to access a data item
- Main memory, register file, condition codes
  - always use direct-mapped addressing
  - natural because data have a unique location in memory
- In cache:
  - Pigeonhole principle: too few holes, too many addresses

DIRECT MAPPED CACHE – HOW IT WORKS

- each cache line contains:
  - valid bit
  - tag: prefix of address range
  - block (value)
- finding an item
  - use some address bits as index to the cache line
  - check valid bit
  - ensure that the rest of the bits match the tag



LOCATING THE CACHE LINE

- Values of interest:
  - Size of the cache block
    - must be power of 2
    - determines how many bits are used for the offset
  - Number of cache lines
    - also a power of 2
    - *Cache index*: determines which cache line to check
  - Tag is the number of bits needed to check if proper address range in cache.

DIRECT MAPPED CACHE – EXAMPLE

- Assume memory is named using 12 bits
  - Each byte has a different address
  - Total memory space: 4096 bytes
- Assume a cache with 8 lines, each for a block of 16 bytes

EXAMPLE (CONT.)

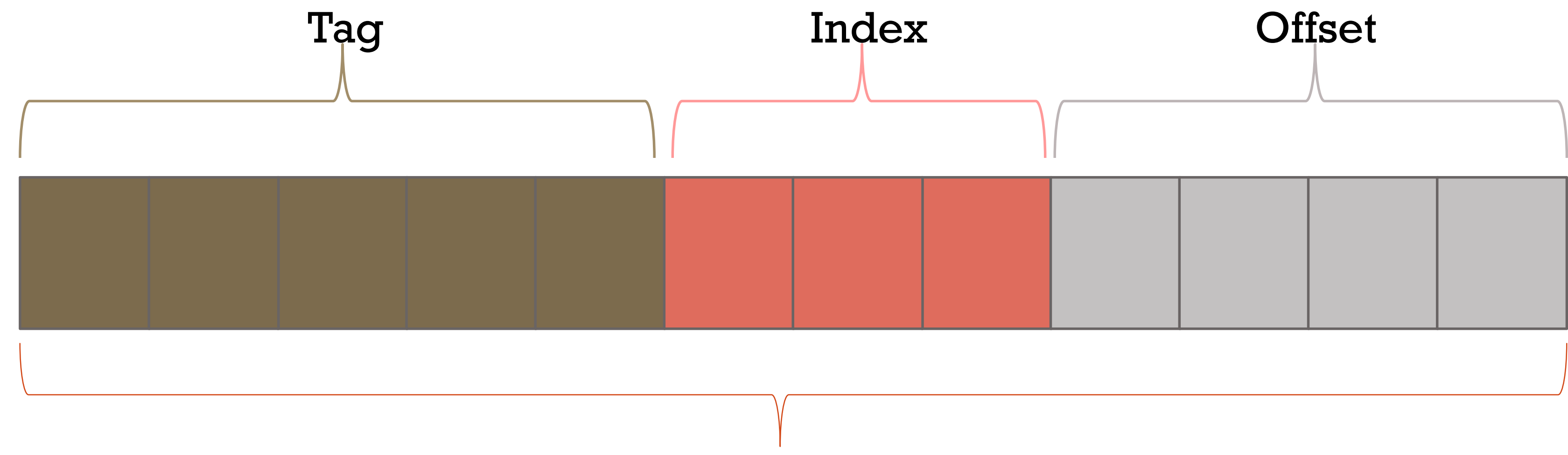
- All possible addresses are divided into blocks:
  - First block: addresses 000000000000, 000000000001, 000000000010, 000000000011, … 000000001111
  - Second block: addresses 000000010000, 000000010001, … 000000011111
  - …
  - Last block: addresses 111111110000, 111111110001, … 111111111111
- We can use the address as follows:
  - First 8 bits: range address (block number)
  - Last 4 bits: block offset

EXAMPLE (CONT.)

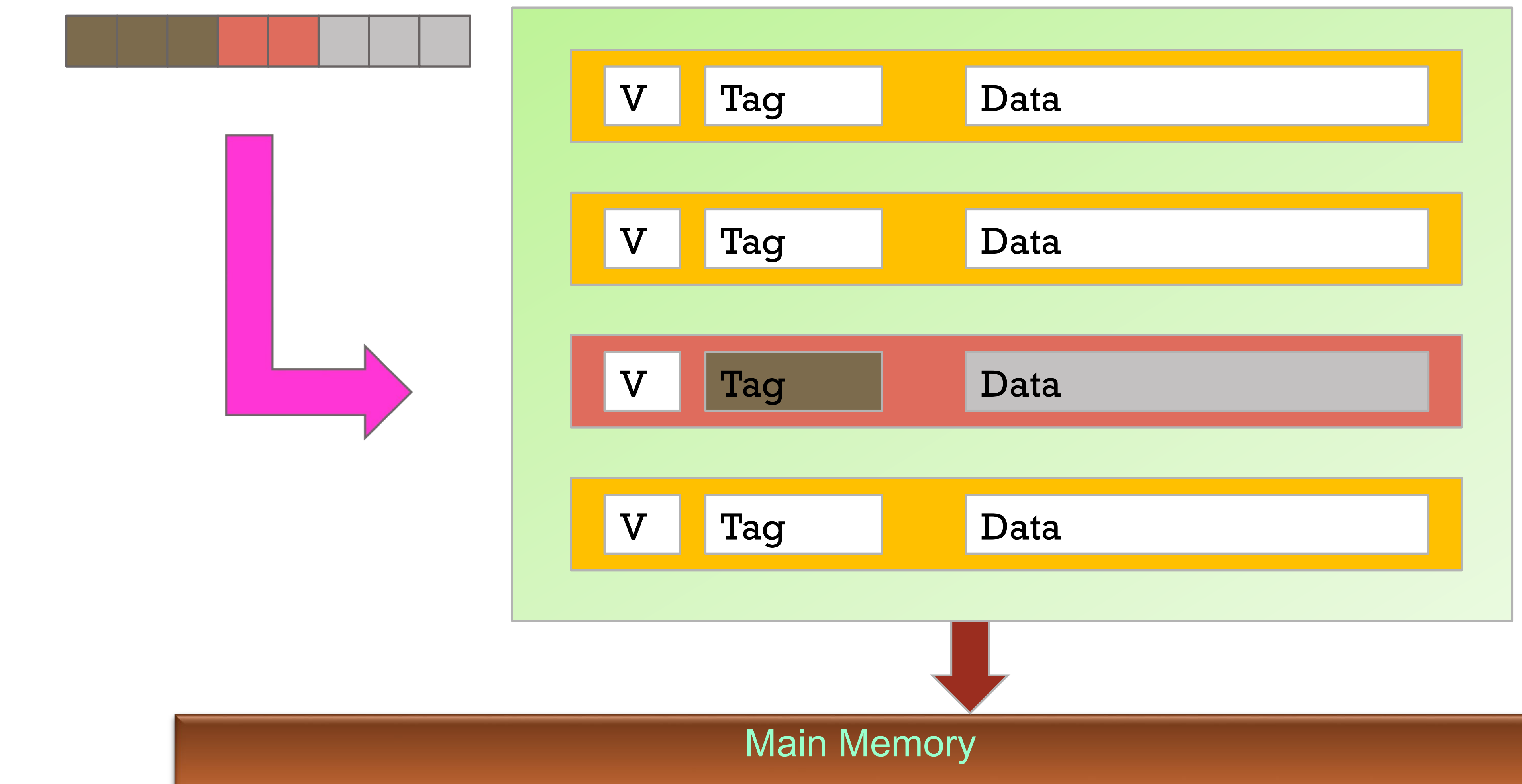
- Cache has 8 lines: 000, 001, 010, 011, 100, 101, 110, 111
- Where should we store block 01111010?
- Assume we just read address 01111010|1111
  - Spatial locality: we are likely to read the following address soon: 01111011|0000
  - Keeping both in cache simultaneously is beneficial
  - So they should be in different cache lines
  - Idea: keep 01111010 in line 010, and 01111011 in line 011

EXAMPLE (CONT.)

- So the 12-bit address will be divided into (from least to most significant):
  - 4 bits for the offset within a block
  - 3 bits for the index of the cache line to be used
  - 5 bits for the cache tag

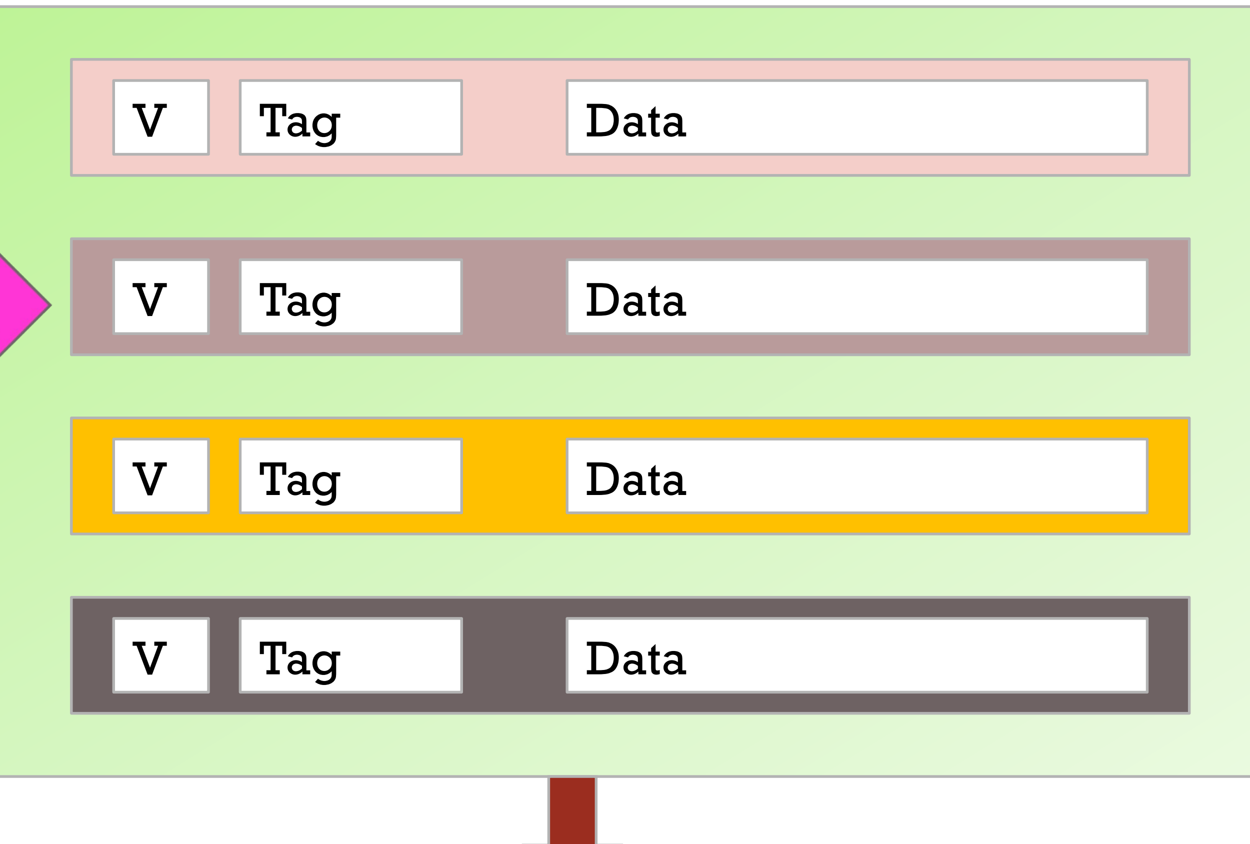


ASSUME 8 BIT ADDRESS



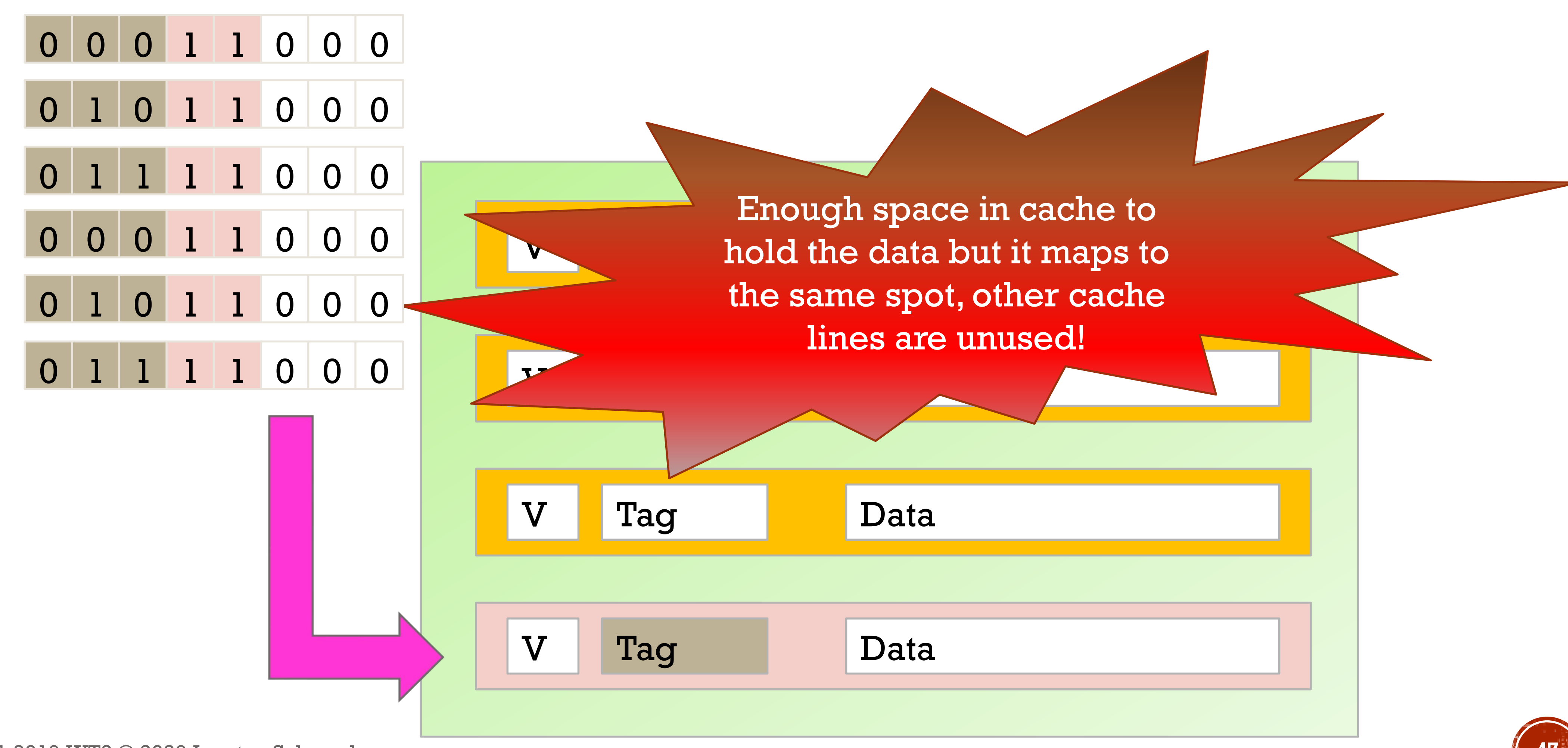
0 1 0 0 1 0 0 1

LOOKUP SUMMARY



1. Extract index bits
2. Go to cache line identified
3. Verify that block is valid
4. Check Tag
5. Return bytes starting at offset if tag matches

SUPPOSE WE ACCESS DATA IN THIS PATTERN



SOLUTION: ASSOCIATIVE MAPPING

- data item is located by searching for its name (TAG)
  - searching cache for a tag in hardware is conceptually simple
    - check every cache line in parallel at once (one comparator per line)
  - any block can be stored at any location in cache within the cache set
  - each block stores a tag and value (just like in direct mapped cache)
- associative caches
  - fully-associative
  - E (or N)-way associative

FULLY ASSOCIATIVE CACHE

- Fully Associative Cache: item can be stored anywhere
  - only the tag is used to identify a block
- problem:
  - increasing associativity requires more circuits, thereby decreasing clock frequency
  - tag needs to be larger, requires more comparison

SET ASSOCIATIVE MAPPING

- set-associative
  - Half-way between direct mapped and fully associative
  - divide cache into multiple fully associative caches called sets
  - address bits pick one set of cache lines
  - associative compares then picks matching tag from lines in set (if there is a match)

E-WAY SET ASSOCIATIVE CACHE (HERE: E = 2)

Step 1 – find the set

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

t bits	0…01	100

find set

v tag 0 1 2 3 4 5 6 7 | v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7 | v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7 | v tag 0 1 2 3 4 5 6 7
……
v tag 0 1 2 3 4 5 6 7 | v tag 0 1 2 3 4 5 6 7

From CMU slide set

E-WAY SET ASSOCIATIVE CACHE (HERE: E = 2)

Step 2 – look for tag match in parallel

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

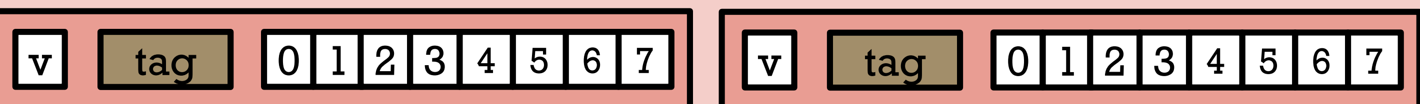
t bits	0…01	100

compare both

valid? + match: yes = hit

v	tag	0	1	2	3	4	5	6	7	v	tag	0	1	2	3	4	5	6	7
v	tag	0	1	2	3	4	5	6	7	v	tag	0	1	2	3	4	5	6	7
v	tag	0	1	2	3	4	5	6	7	v	tag	0	1	2	3	4	5	6	7

• •

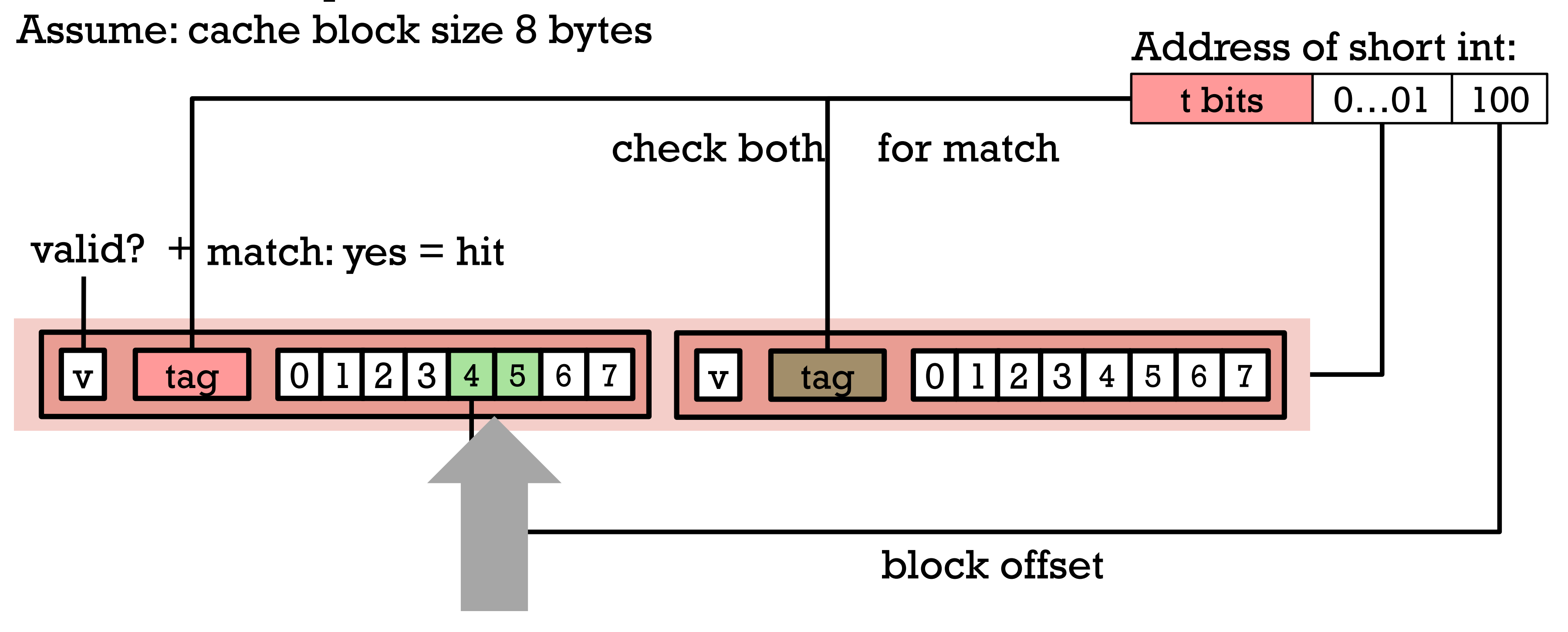


From CMU slide set

E-WAY SET ASSOCIATIVE CACHE (HERE: E = 2)

Step 3 – assuming hit, extract bytes requested

E = 2: Two lines per set
Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

E-WAY, SET-ASSOCIATIVE CACHES

- For a given amount of cache memory what should E be
  - increasing E improves ______________________________
  - decreasing E improves ______________________________

SUMMARY: PARAMETERIZING A CACHE

- description
  - cache contains a collection of direct-mapped sets
  - each set contains associatively-matched lines
  - each line contains valid bit, tag and data block
- Tuples to parameterize a cache: (S, E, B, m)
  - S # of sets – must be power of 2
  - E # of lines per set
  - B # of bytes per cache block – must be power of 2
  - m # of physical address bits
- address-bit breakdown
  - s -> # of set-index bits $= \log_2 S$
  - b -> # of block-offset bits $= \log_2 B$
  - t -> # of tag bits $= m - (s + b)$

BITS AND SIZES

- Number of bits needed to represent each of
  - Tag, Sets, and cache block size sums to number of address bits
- Cache size
  - Number of sets * Cache associativity * Cache block size (in bytes)

EXAMPLE

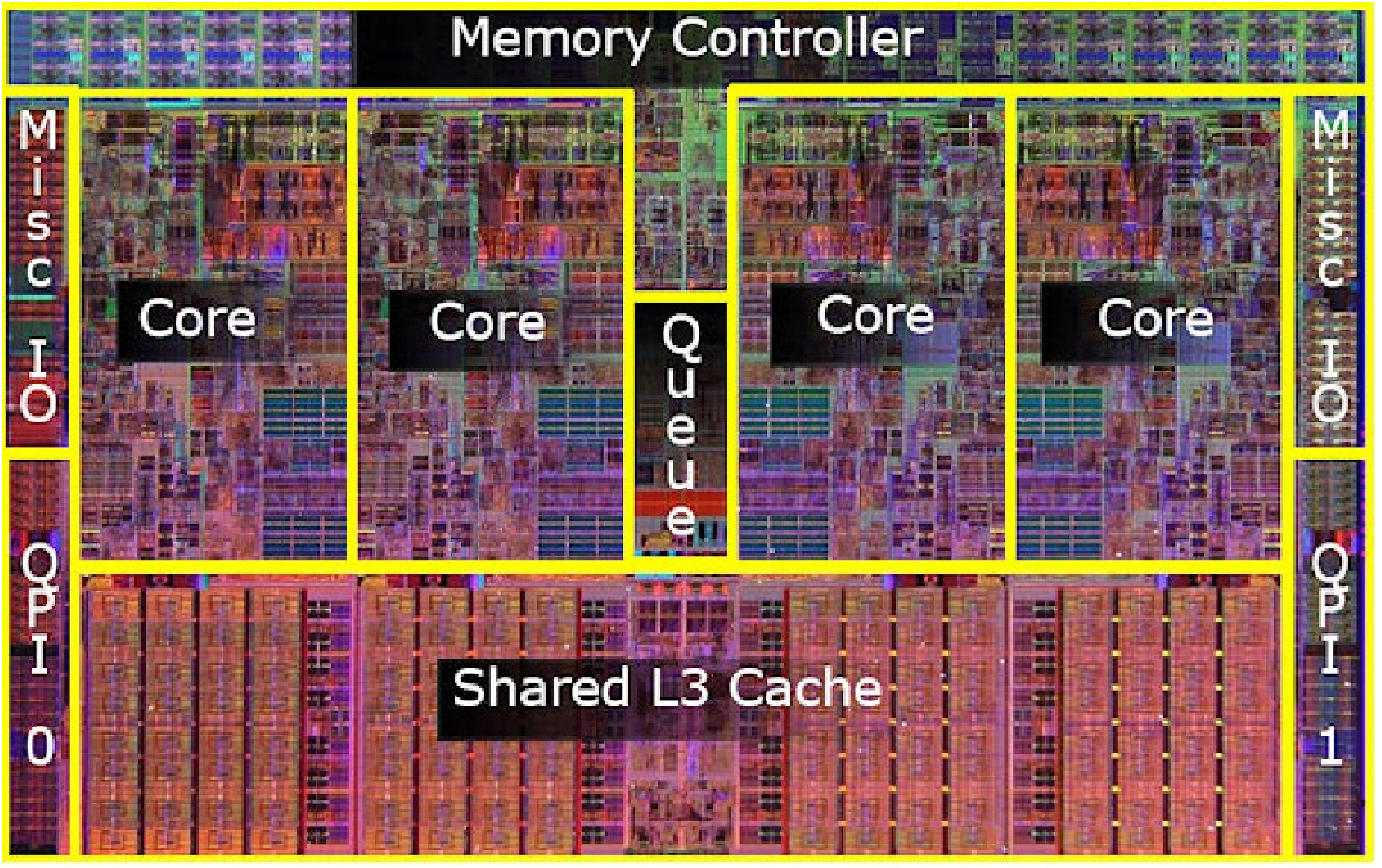
- example: 2MB cache, 64-byte blocks, 32-bit physical address
  - direct mapped
  - fully associative
  - 8-way set associative

	m	Tag (Bits)	Sets S	(lines/set) E	Bytes/ block
Direct Mapped	32 bits				
Fully Associative	32 bits				
8-Way	32 bits				

INTEL CORE I7 (QUAD CORE)

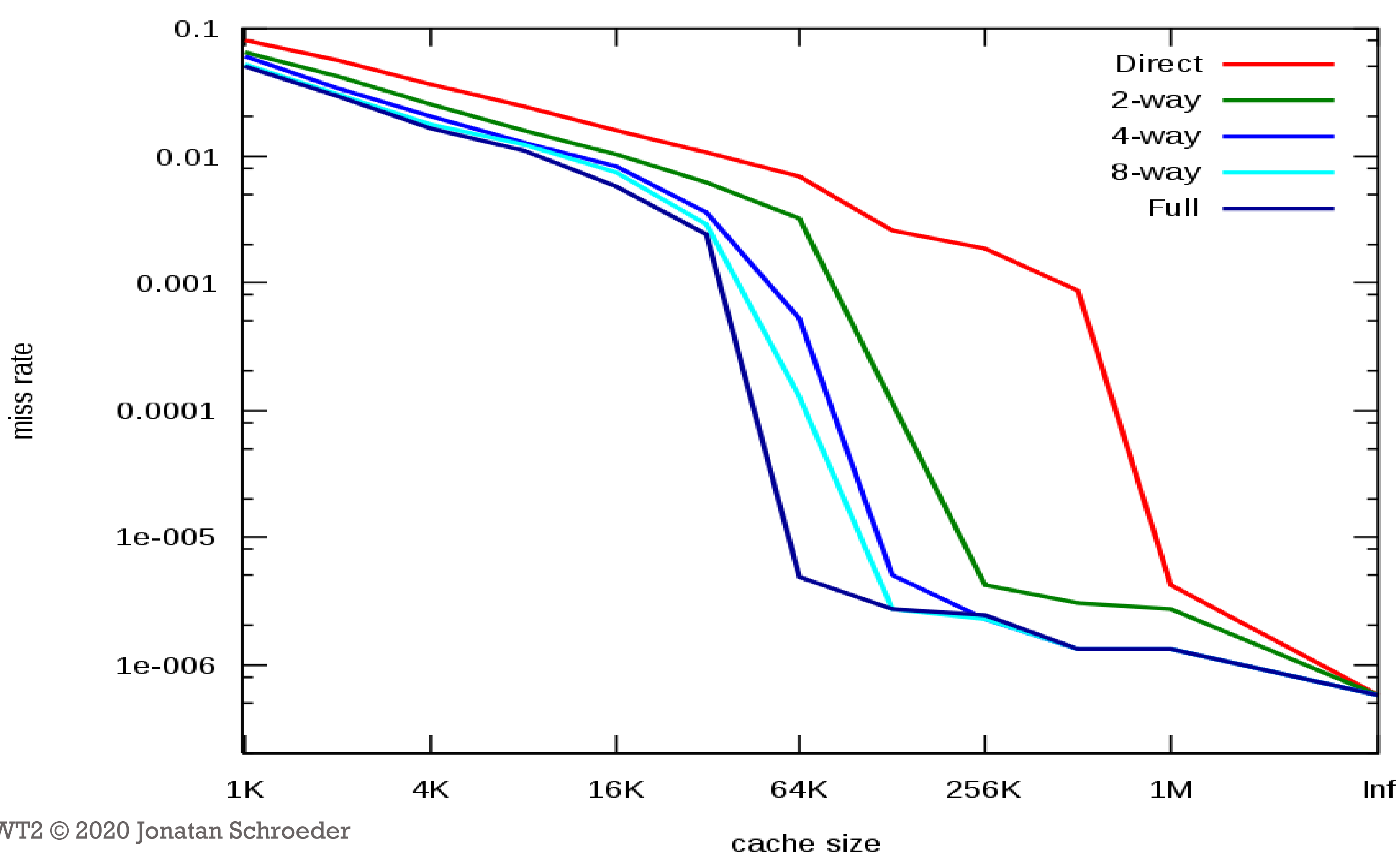
- L1 I-cache (per core): 32KB, 4-way set associative
- L1 D-cache (per core): 32KB, 8-way set associative.
- L2 unified cache (per core): 256KB, 8-way set associative.
- L3 unified cache (shared): 8MB, 16-way set associative.

CORE I7



The Core i7 die and major components. Source: Intel

DOES CACHING WORK?



PREFETCHING

- key idea
  - fetch data into the cache before it is accessed
  - alternative to caching for low-locality workloads (or that don't fit in cache)
    - typically an addition to caching
  - hide memory latency by overlapping fetch with computation
- problem
  - must know or guess what will be accessed in the future
  - predicting the future is hard

PREFETCHING TECHNIQUES

- Software-based: modify ISA to add prefetch instruction (IA32: prefetcht0)
  - its like a "load" (e.g., mrmovq), but without destination or stalling for completion
  - compiler can exploit static properties of program
    - e.g., loops, database queries
- Hardware-based: use recent access patterns to predict future accesses
  - hardware can exploit dynamic properties of program
  - simple patterns are easy to detect dynamically
    - e.g., sequential w/ fixed stride (forward or reverse) - instruction cache

SUMMARY

- caching
  - data loaded into cache implicitly as side effect of normal access
  - exploit temporal and spatial locality when it exists
  - keep recently-access data in fast memory for a while
  - fetch data into fast memory in multi-word blocks
- prefetching
  - data loaded into cache explicitly by program or hardware predictor
  - fetch data into fast memory before it is accessed
  - thus overlap memory latency with computation

REPLACEMENT POLICIES

- Throwing data out: what do we do when a set is full?
- Direct-mapped caches:
  - No choice (only one location can be used)
- Set-associative or fully associative caches:
  - Need to select a block to throw out
  - What would be the optimal decision for a block to be removed?
  - Alternatives: random, Least-recently used (LRU)
- The right method depends on the level of the cache
  - Caches lower in the memory hierarchy have bigger miss penalty: we can spend more time to decide

HANDLING WRITES

- Multiple copies of the data exist (L1, L2, L3, memory, maybe even disk)
- Case 1: the location being written to is in the cache (write-hit)
  - *Write-through*: write the value back to the next level immediately
  - *Write-back*: defer write to next level until cache line is replaced
  - Trade-off:

HANDLING WRITES

- Case 2: the location being written to is not in the cache (write miss)
  - *No-write-allocate*: write directly to memory, ignoring the cache
  - *Write-allocate*: load location into the cache before writing
  - Trade-off:
- Typically:
  - Write-through is used with No-write-allocate
  - Write-back is used with write-allocate
- The choice may vary according to the level in the memory hierarchy

INSTRUCTIONS AND DATA

- instruction and data caches
  - L1 caches typically split instructions from data
  - data cache (d-cache) stores only data
  - instruction cache (i-cache) stores only instructions
    - what's different about instructions?
- unified cache
  - stores instructions and data together
  - does not distinguish between them
  - L2 (and L3) caches are typically unified

MULTI-CORE ISSUES

- Choice
  - one cache per core
  - one shared cache for all cores
- What are the tradeoffs involved in choice?
  - advantages of per-core cache:
  - advantages of shared cache:
- What about cache coherence?

DESIGNING CACHES FOR PERFORMANCE

- workload dependent
  - study - use access traces
  - some traces collected from real programs
  - others generated synthetically
- simulate cache behaviour
  - pick a cache configuration
  - run access trace through simulator
  - look at what hits where
- things to measure
  - miss rate for each type of miss (cold, conflict, capacity)
  - hit rate
  - miss penalty for each type of miss

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

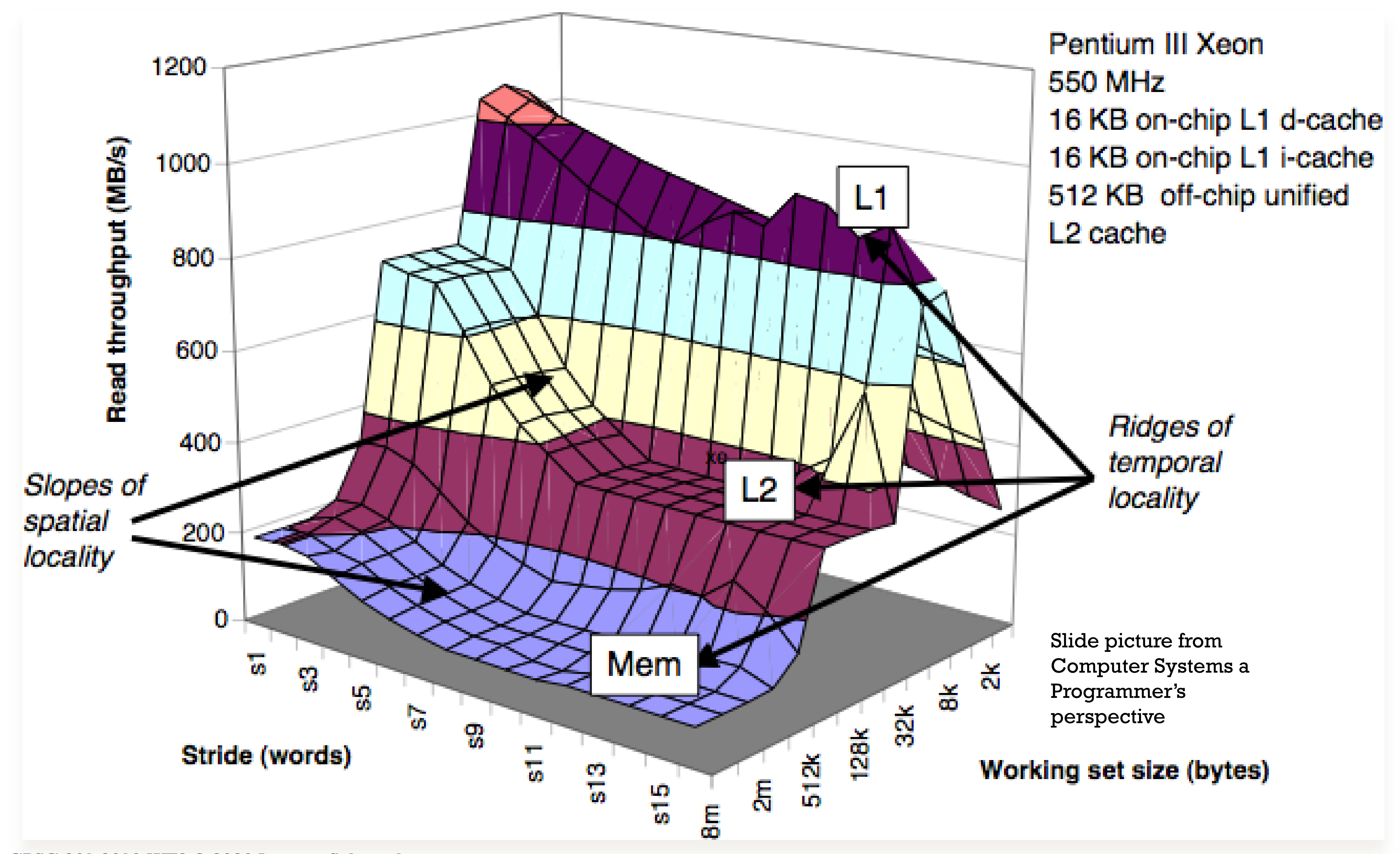
MEMORY MOUNTAIN

- To get an overall idea of the performance of a cache system in terms of:
  - Cache levels (how each cache performs)
  - Cache sizes
  - Block sizes
  - Effects of locality
- We want to have a basic idea through a graph: memory mountain

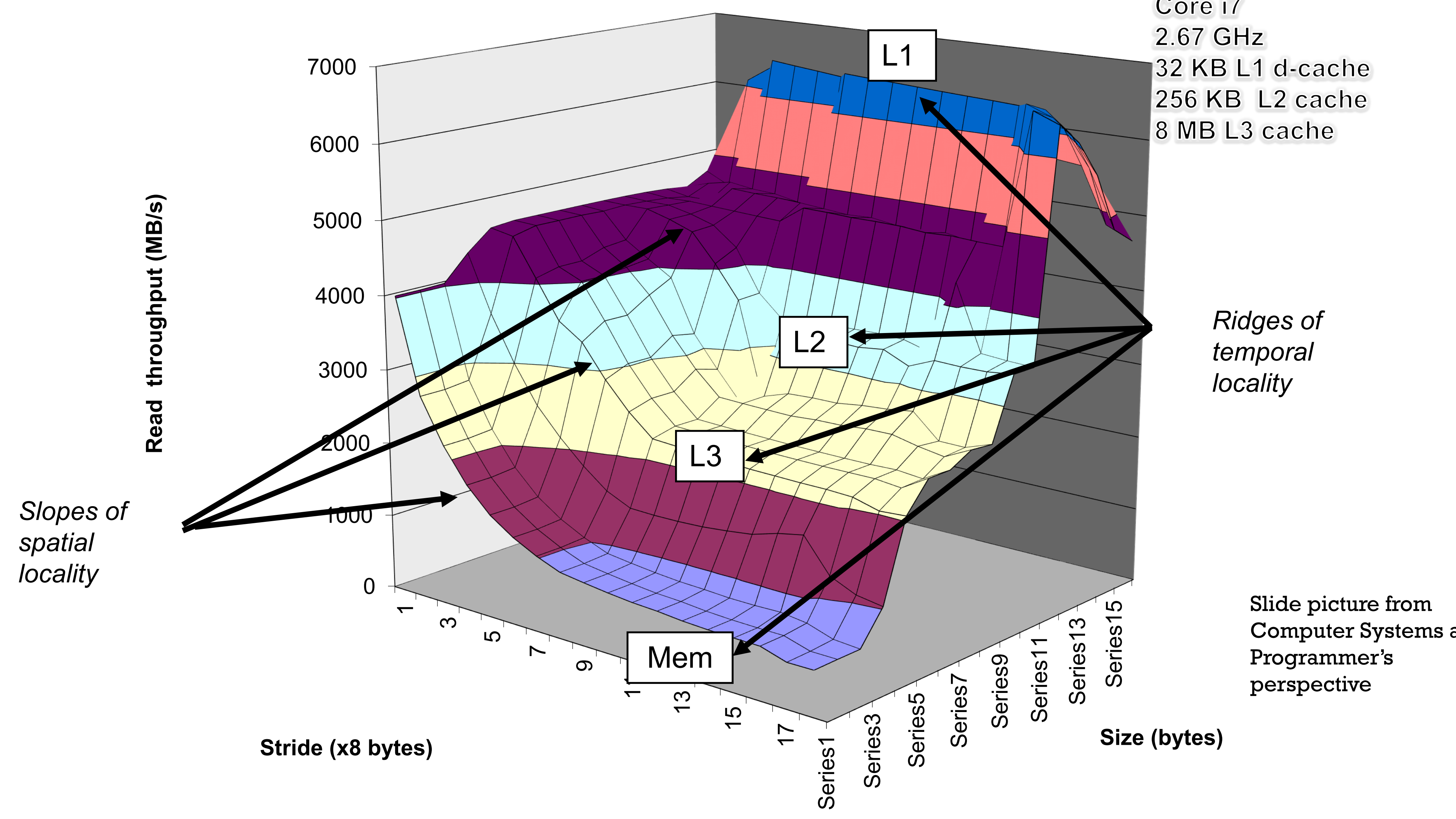
MEMORY MOUNTAIN TEST PROGRAM

- A program to show how the read throughput depends upon temporal (size) and spatial (stride) locality
- Run once to warm cache; then run again to measure performance

```
void test(int size, int stride) {
  int i, res = 0;
  volatile int sink;
  for (i = 0; i < size; i += stride) {
    res += data[i];
  }
  sink=res; // stop compiler from deleting code.
}
```

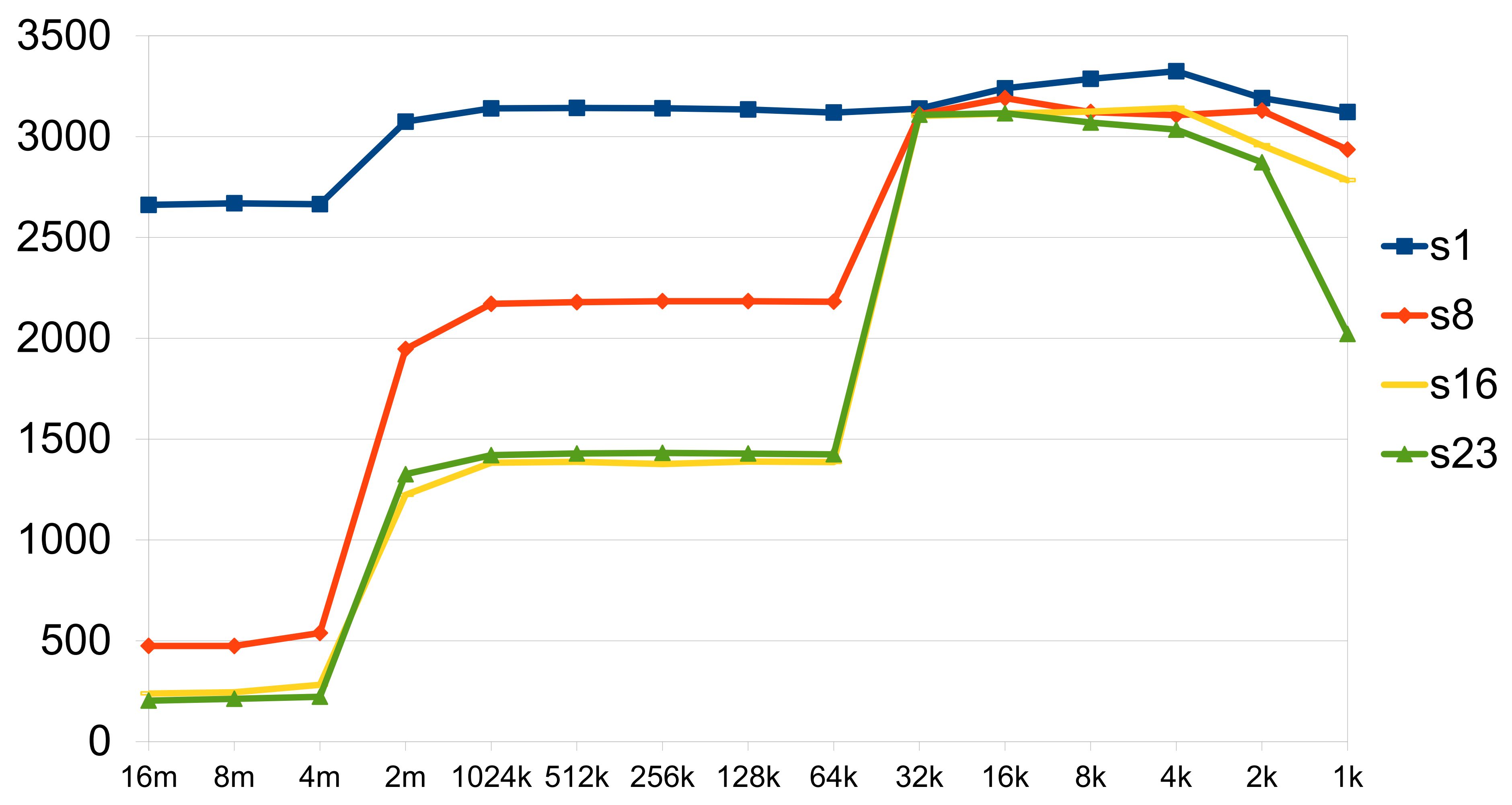


Credit: Bryant and O'Hallaron, CMU



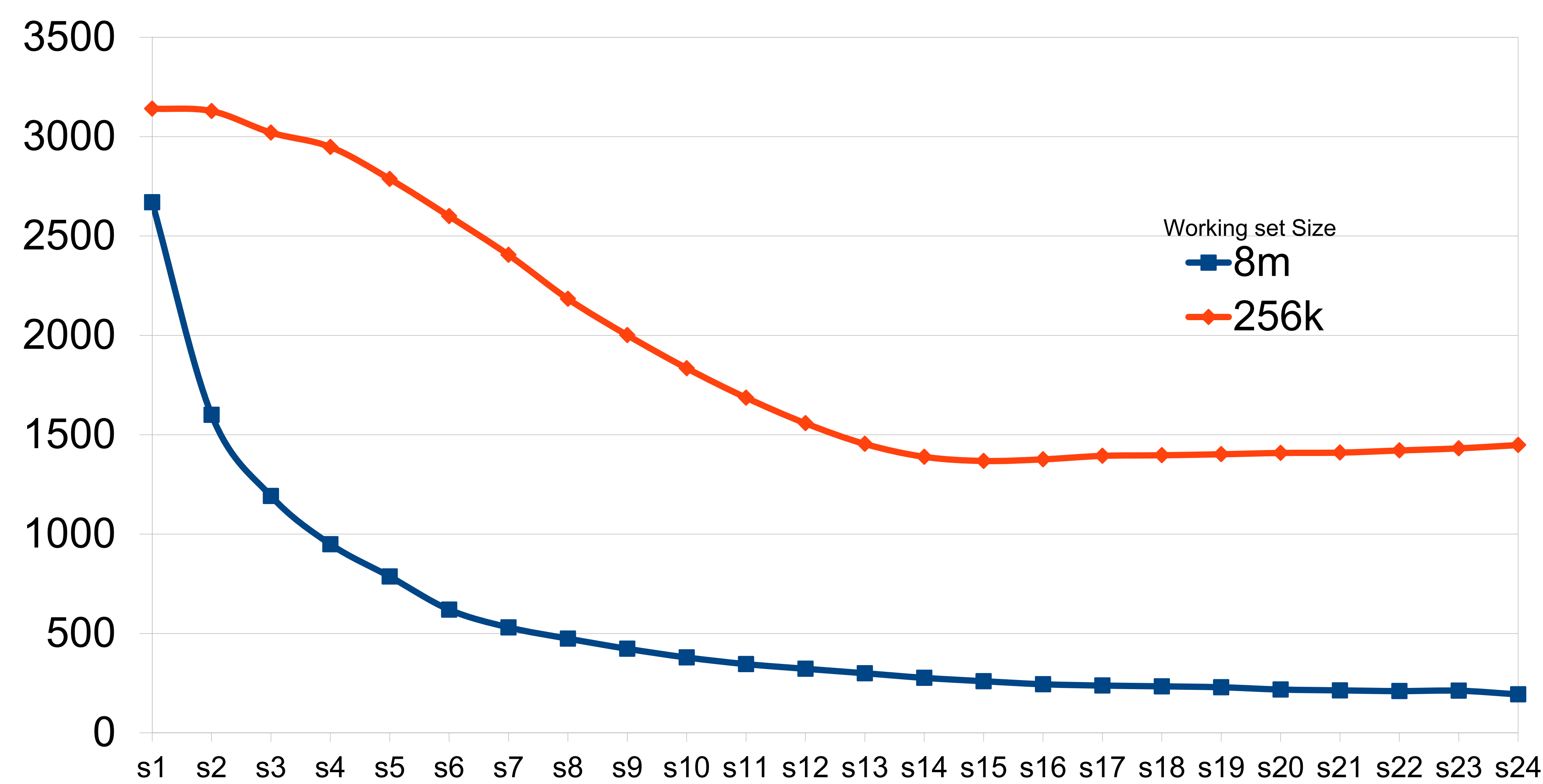
Slide picture from Computer Systems a Programmer's perspective

1.83 GHz CoreDuo – Temporal Locality



CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

1.83 GHz CoreDuo – Spatial Locality



WRITING CACHE-FRIENDLY CODE

- profile and measure
  - find out what makes program slow
  - determine whether cache performance is a problem
- optimize
  - Cold, capacity, conflict misses
  - increase temporal locality
  - increase spatial locality
  - consider prefetching

APPLYING IT INTO PROGRAMMING

- Run access traces. Evaluate the proportion of accesses that hit the cache in specific functions
  - particularly those dealing with large amounts of data
- Partition your data, and deal with the subsets one at a time, instead of dealing with the whole data at once
- Figure out where data ended up in memory, and whether or not it could be relocated to decrease the # of conflict misses
- Increase temporal locality by keeping the working set small
- Increase spatial locality by keeping stride small

FILE SYSTEMS

Unit 5

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

OUTLINE

- Introduction
- Disks characteristics
- Virtual File Systems
- File System implementation and layout
  - ISO 9660 (CD-ROMs), MS-DOS, Linux

VIRTUALIZATION

- Two tools to handle complexity:
  - Modularity: break a system down into smaller pieces
  - Abstraction: hide the details inside each module so the rest of the system does not need to know about them
- Examples of modularity:
  - Functions in C, C++ or assembly language
  - Files in C or assembly language
  - Classes in Java, Python

SOFT MODULARITY

- Problem: functions are what we call soft modularity
  - One module may easily make mistakes that affects other modules
  - Example:

```
test2:
  leaq (%rdi, %rdi, 2), %r15
  movq (%rsi, %r15, 8), %rdi
  call test4
  addq %r15, %rax
  ret
```

VIRTUALIZATION

- Another example:

```
long do_something(long x) {
    volatile long a[1];
    a[2] = 0x0123456789ABCDEF;
    return x + 7;
}
```

- How can we isolate modules better?

VIRTUALIZATION

- One option: run modules on separate computers
  - Pros:
    - very strong separation
    - simple to reason about
  - Cons:
    - slow
    - expensive
    - passing values gets complicated

VIRTUALIZATION

- Another option: virtualization
  - Instead of using a physical device, we provide another device with the same interface
  - The user interacts with the device in the same way
  - No need to know the implementation is different
  - When we virtualize a resource, we can either
    - Provide the exact same interface as for the physical resource, or
    - Modify it to make it more general, easier to use, easier to implement, etc.

VIRTUALIZATION: WHAT TO VIRTUALIZE

- Processors: a machine runs multiple programs, each of which assumes it has full control of the CPU
- Memory: each program assumes it's the only one using memory
  - It ensures programs can only access the memory they are authorized to access
  - It allows programs to access more "memory" than physically available
  - We will discuss virtual memory in details later

VIRTUALIZATION: WHAT TO VIRTUALIZE

- Secondary storage (disks, removable devices, etc.)
  - Access using OS-specific primitives (file system)
    - Typically abstracted by programming language
  - Allows transparent access to different types of devices
    - Similar primitives for any type of storage device
    - Virtual disks stored in memory, multiple devices providing redundancy, etc.
- Communication
  - several types of virtual communication links
  - send/receive primitives

VIRTUALIZATION TECHNIQUES

- Emulation: use software to simulate a physical resource
  - Virtual file systems
  - Java virtual machine
- Multiplexing: a single physical resource is shared between multiple applications
  - processors and memory (for individual programs)
  - web sites (one machine can run multiple web servers)

VIRTUALIZATION TECHNIQUES

- Aggregation: several resources residing on different machines are grouped together transparently
  - A file system may be divided between disks located on multiple file servers
  - Most popular web services actually consist of many machines (e.g., www.google.com)
  - Anycast: a network request is sent to any server that responds to it (used by some DNS servers)

FILE SYSTEMS

- File systems virtualize storage devices:
  - Work directly with the view of the device provided by the controller
  - Provide a device-independent view
- Applications see further levels of abstraction on the file system

Application
System I/O
File System
Disk

INTRODUCTION

- Why do we use file systems?
  - They manage the device (so application don't need to)
  - They hide complexities of different disk drive types (e.g., rotating disks vs SSD)
  - They organize the data on the disks
  - They protect the data against unauthorized and/or incorrect access

INTRODUCTION

- File system properties:
  - persistence: must survive reboots
  - robustness: must be recoverable after a crash occurs
  - efficiency: must be fast and make good use of disk space
  - correctness: their state must reflect the operations performed

INTRODUCTION

- What issues are relevant to the design of a file system?
  - How files are named
  - Where information about a file is stored
  - How to find a file's data, given its name
  - How space for new files is located
  - How to recover from hardware and software failures

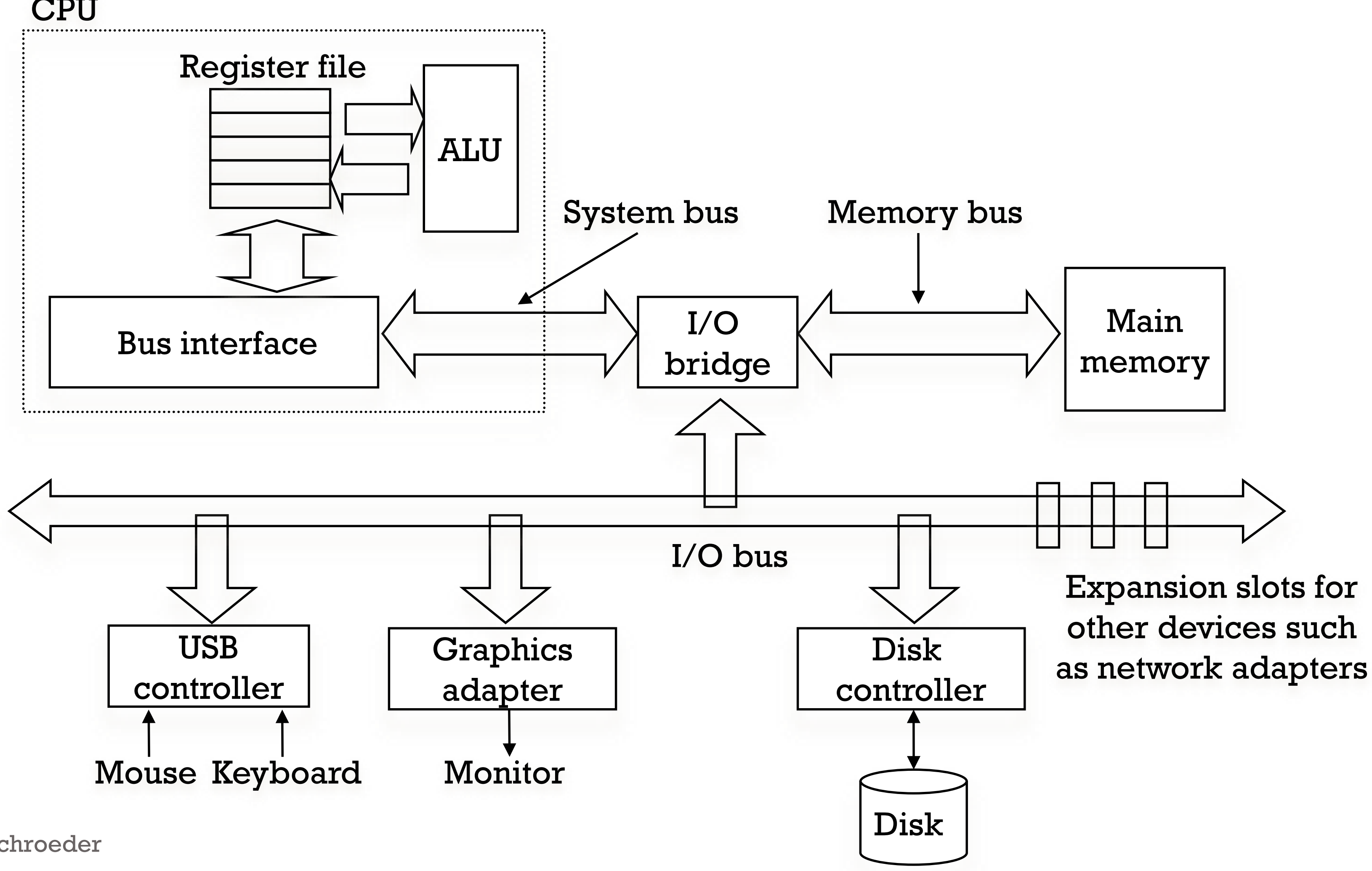
MEMORY VS DISK

- Memory access is different than disk access
  - memory locations are accessible individually
  - data on disk can only be accessed one chunk (block) at a time
    - block size used to be 512 bytes, now most disks use 4096 byte blocks
    - blocks are usually a virtualization of one of the physical properties of the disk (for rotating disks: sectors)

MEMORY VS DISK

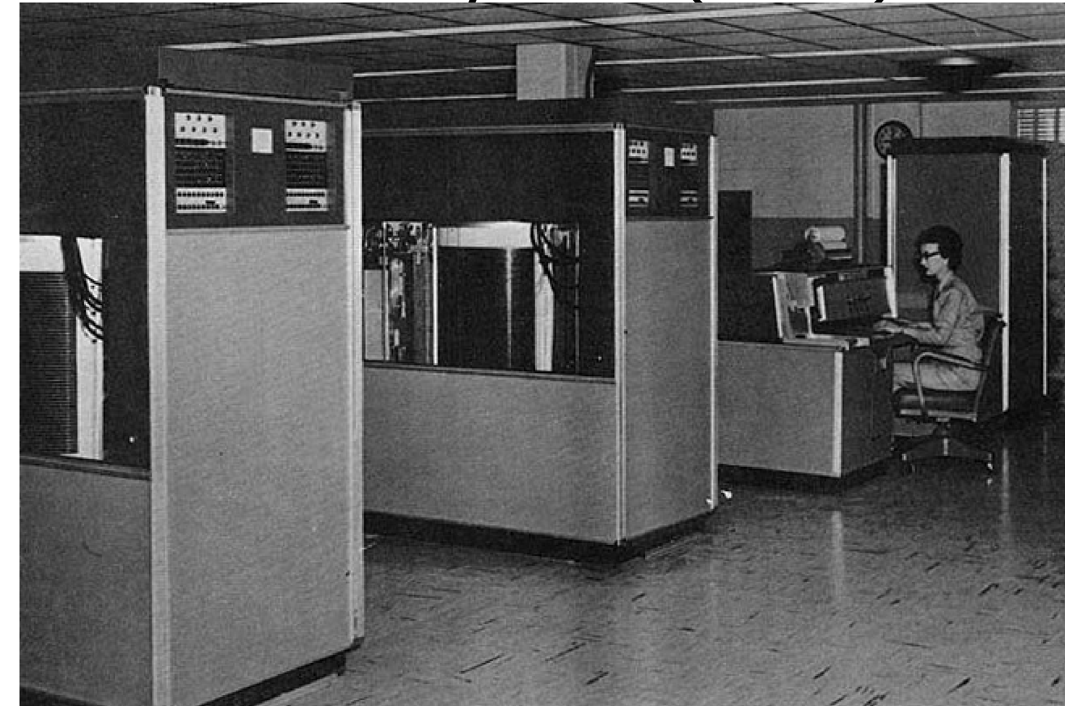
- Memory naming is different than disk naming
  - variables are accessed using their address
  - data on disk is normally accessed through a file name (path)
    - the file system translates the name and offset into a logical block number
    - the disk controller maps that number to the location on disk

SYSTEM ARCHITECTURE



ROTATING DISKS CHARACTERISTICS

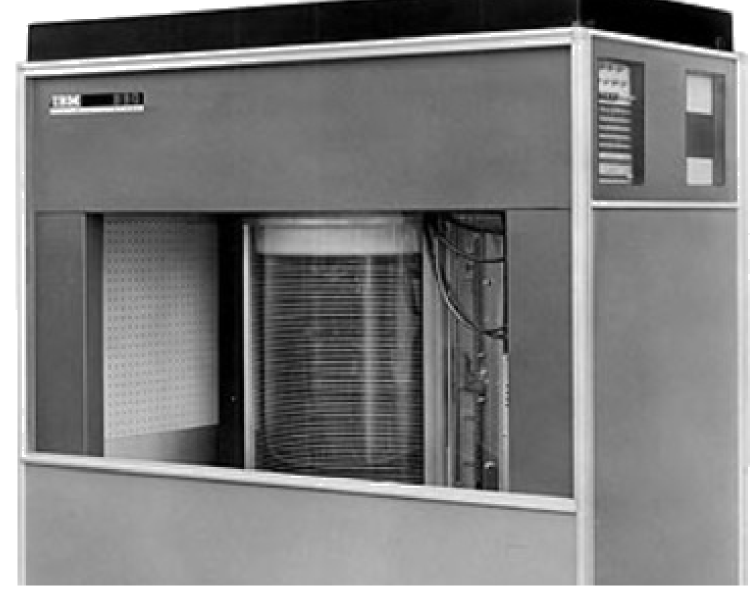
- The IBM 350 Disk system (1956)



From: http://en.wikipedia.org/wiki/File:BRL61-IBM_305_RAMAC.jpeg

ROTATING DISKS CHARACTERISTICS

- The IBM 350 Disk system specifications (1956)
  - 3.3 MB
  - 50 platters @ 1200 RPMs
  - 24 inch diameter platters
  - 20 tracks per inch
  - 100 bits per inch on each track
  - 50000 sectors per drive
  - 100 6-bit chars per sector
  - IBM 350 + disk leased for $3200 /month (equivalent to $27,287 in 2016 dollar).
  - Weighed over 1 ton

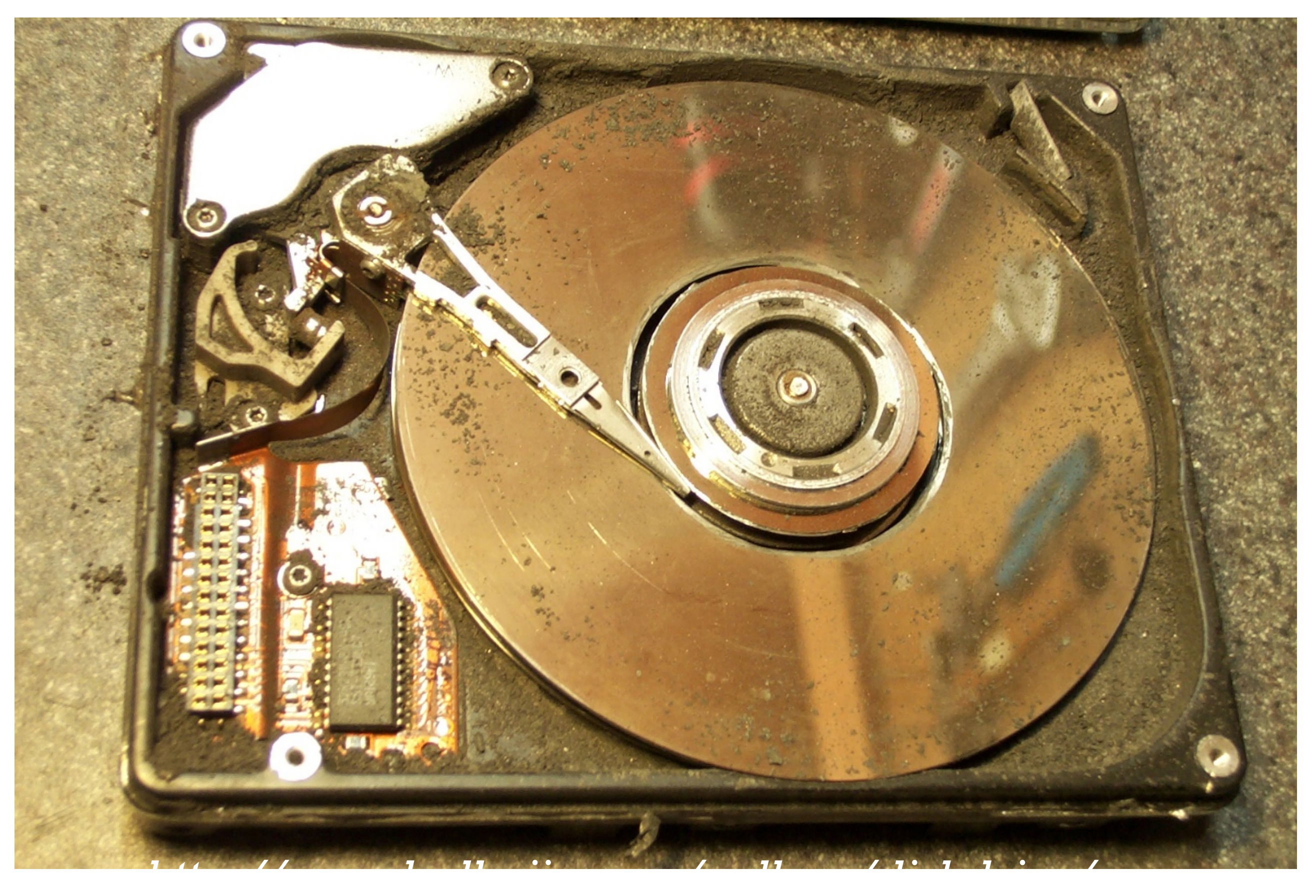


From http://www-03.ibm.com/ibm/history/exhibits/storage/storage_350.html

DISK DRIVE

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

HEAD CRASH



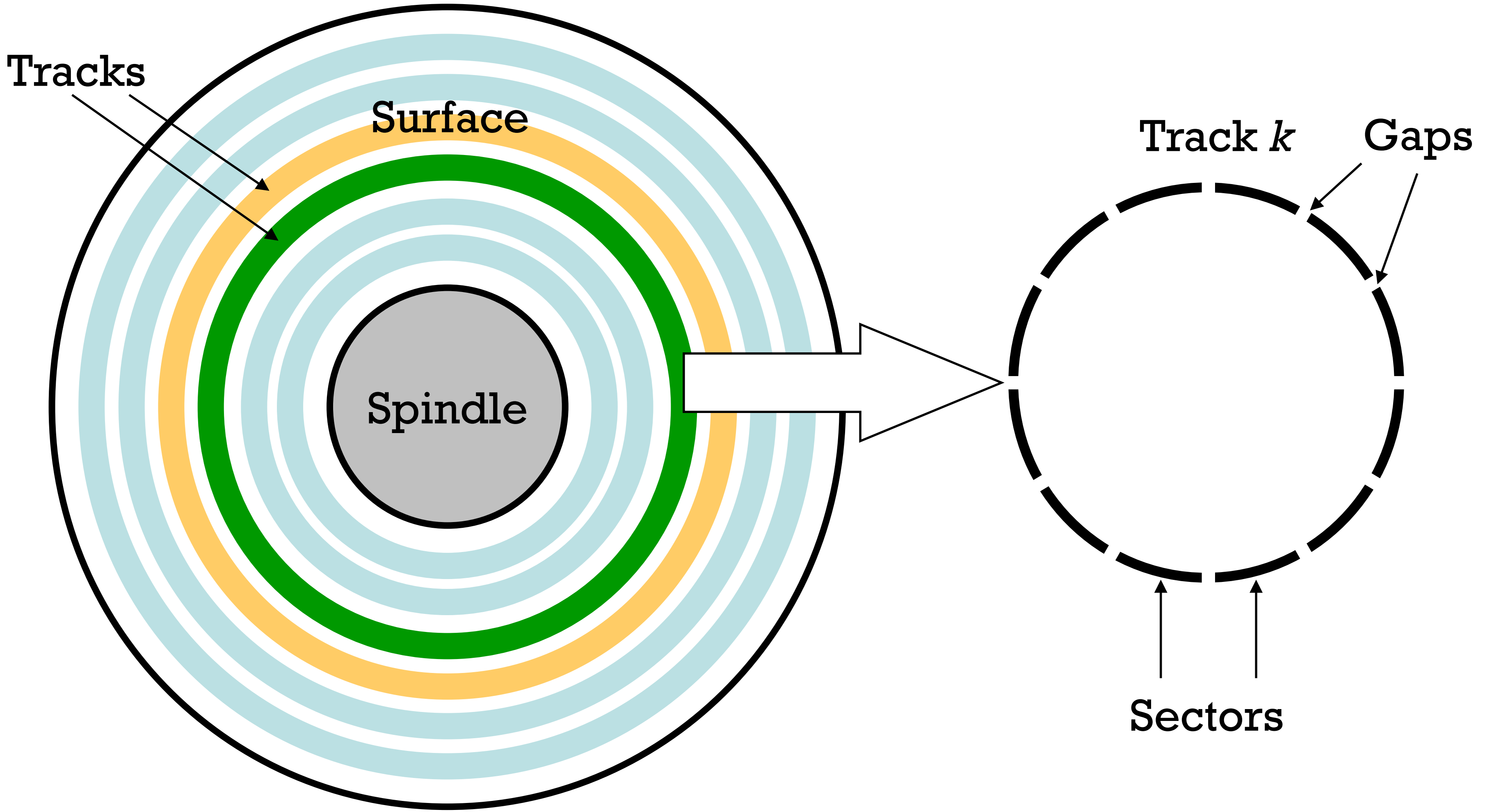
ROTATING DISKS CHARACTERISTICS

- Why are we studying disk characteristics?
  - To understand the issues affecting file system performance
  - To use this information to improve the performance of the programs we write
- File systems are constrained by the medium they are stored on (disks)
- To help inform our understanding of how file systems work, we need to know how disks are structured

ROTATING DISKS CHARACTERISTICS

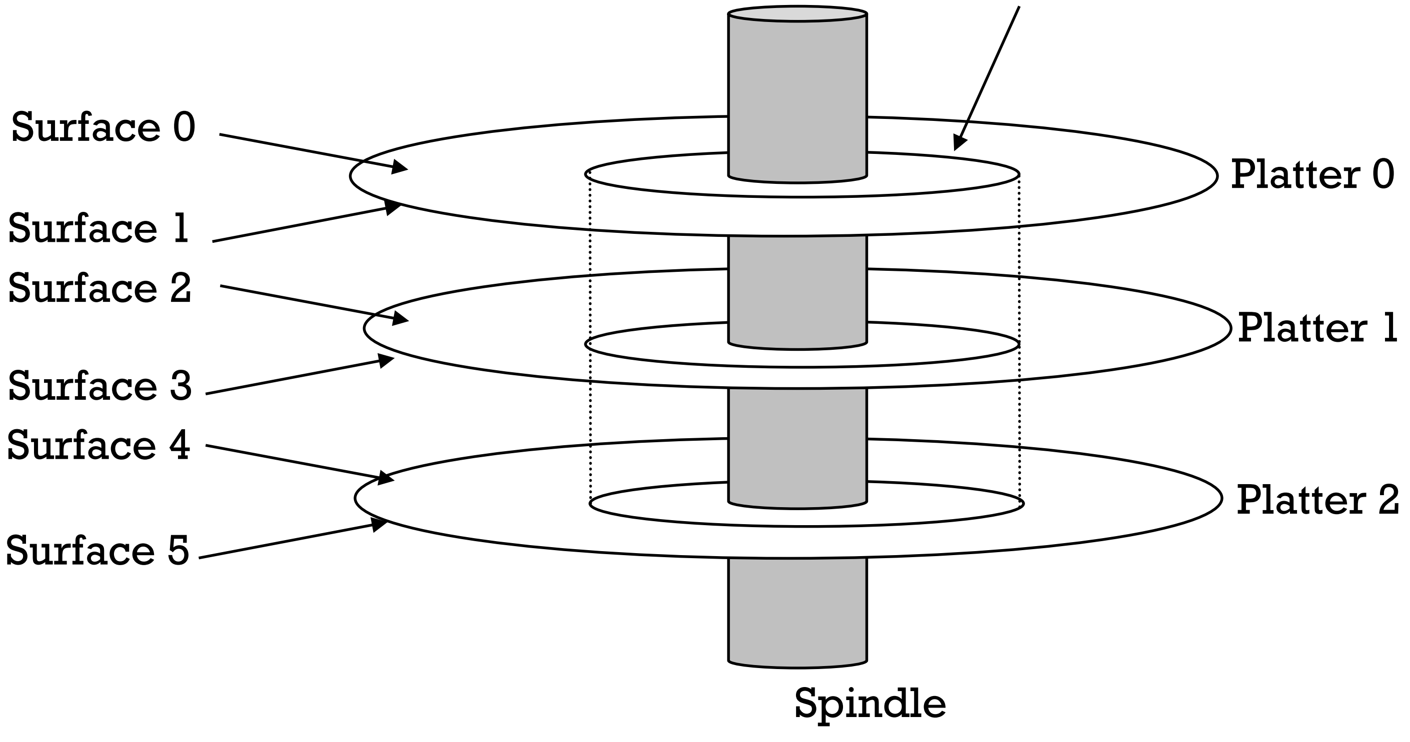
- Terminology:
  - Platter: the circular medium on which data is stored.
  - Surface: each platter has two surfaces (sides).
  - Track: each surface is divided in concentric tracks.
  - Sector: each track is divided into sectors. A sector is the smallest amount of data that can physically be read from or written to disk.
  - Cylinder: the corresponding tracks on all of the disk's surfaces.

ROTATING DISKS CHARACTERISTICS



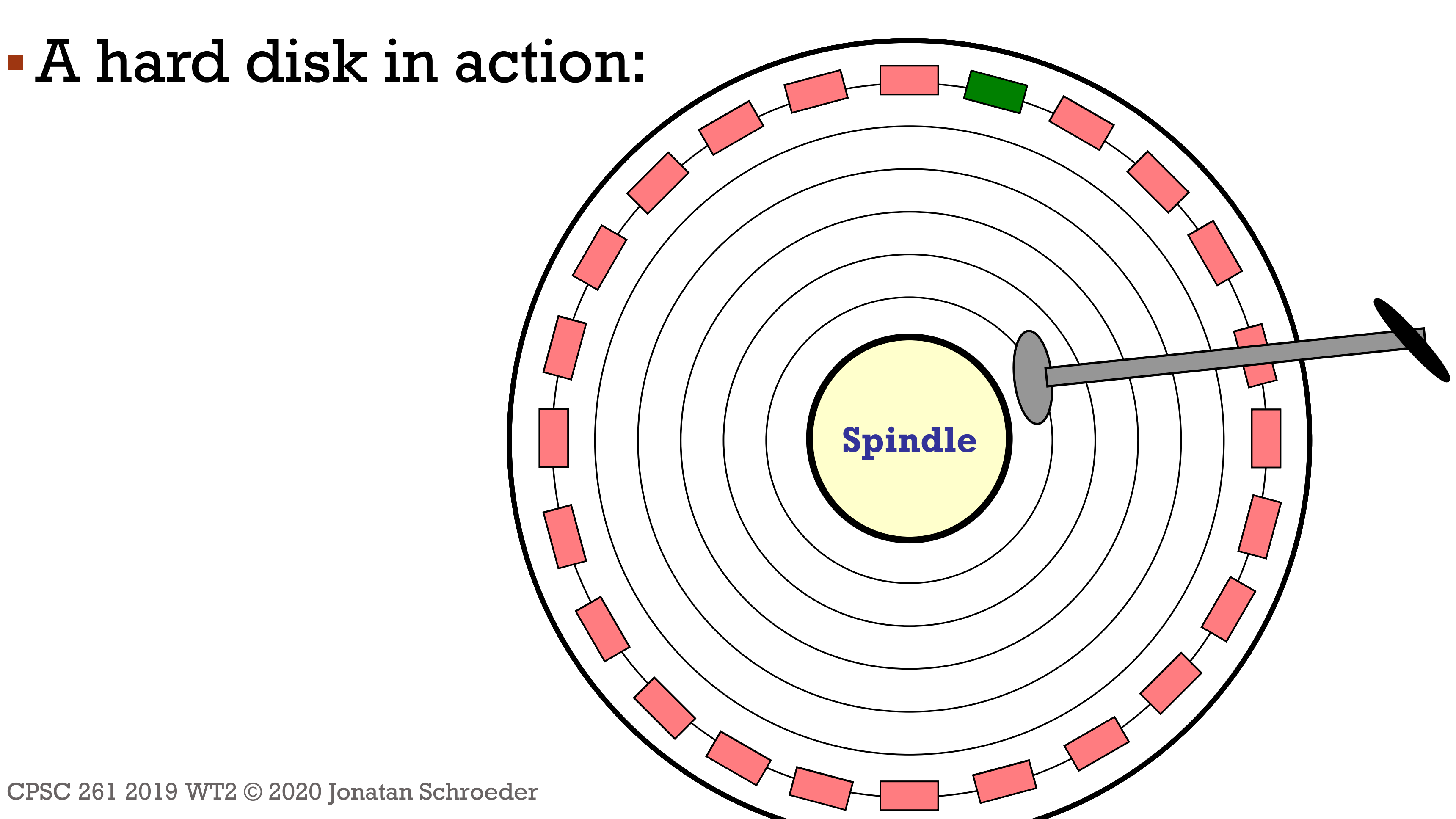
ROTATING DISKS CHARACTERISTICS

- Hard disks: platter view (from the side)



ROTATING DISKS CHARACTERISTICS

- A hard disk in action:



ROTATING DISKS CHARACTERISTICS

ROTATING DISKS CHARACTERISTICS

- What affects the time needed to retrieve data from a hard disk?
  - Seek delay: how long it takes to move the head to the appropriate track
  - Rotational delay: how long it takes for the disk to rotate and bring the data under the heads
  - Transfer delay: how fast the data can be read off the disk

ROTATING DISKS CHARACTERISTICS

- Example: a disk with
  - 12000 rotations per minute
  - 500 sectors per track
  - 1024 bytes per sector
  - average seek time 6ms
- What is the average time needed to read one 4096 byte block?

COMPUTING DISK DELAY

- What is the average time needed to read one 4096 byte block?
  - Seek delay: 6ms
  - Rotation delay: 12000RPM = 200 RPS = 5ms/rotation, average will be half, so 2.5ms
  - Transfer delay: We need to read 4 sectors = 4/500 of a track = 4/500 of 5ms = 0.04ms
  - Total time: 6 + 2.5 + 0.04 = 8.54ms

COMPUTING DISK DELAY

- What is the average time needed to read one 8192 byte block?
  - Seek delay: 6ms
  - Rotation delay: 12000RPM = 200 RPS = 5ms/rotation, average will be half, so 2.5ms
  - Transfer delay: We need to read 8 sectors = 8/500 of a track = 8/500 of 5ms = 0.08ms
  - Total time: 6 + 2.5 + 0.08 = 8.58ms

HOW DOES THIS SCALE?

Block Size (KB)	average seek+rot (ms)	transfer (ms)	average access per block (ms)	Throughput (KB/s)
4	8.50	0.04	8.54	468
8	8.50	0.08	8.58	932
16	8.50	0.16	8.66	1847
32	8.50	0.32	8.82	3628
64	8.50	0.64	9.14	7002
128	8.50	1.28	9.78	13088
256	8.50	2.56	11.06	23146
500 (full track)	8.50	5.00	13.50	37037

BLOCK SIZE VS SECTOR SIZE

- Disks have high per-access overhead
- What is the implication on performance?

- How do blocks help?

- What does that mean for disk locality?

ROTATING DISKS CHARACTERISTICS

- Increasing block size:
  - Higher throughput, Less external fragmentation
- Decreasing block size:
  - More efficient use for small files
  - Less internal fragmentation of disk and memory
  - Faster access time per block
- Compromise:
  - Choose a small enough block to minimize fragmentation
  - Keep related data in contiguous blocks (locality)

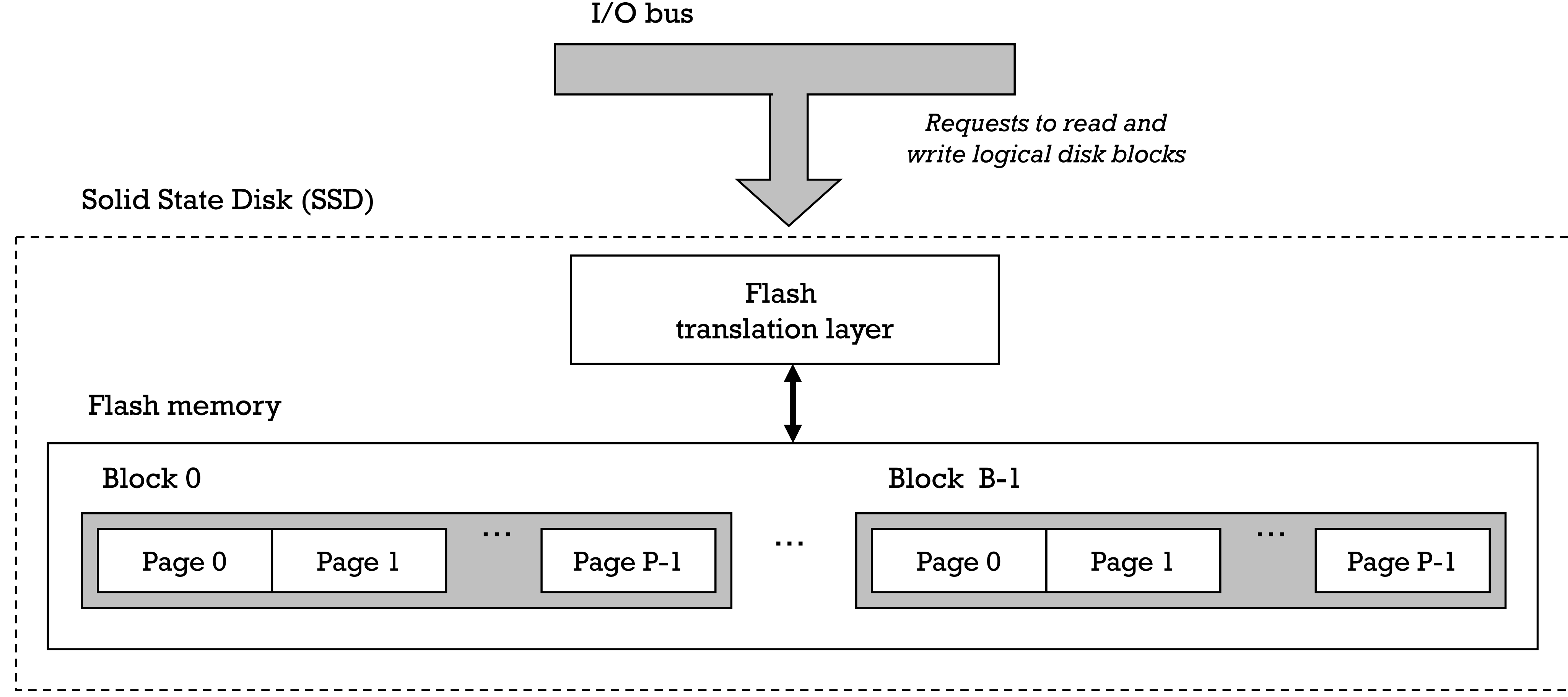
SOLID STATE DISK CHARACTERISTICS

- Flash memory: used in USB sticks, digital cameras, iPods, etc.
- Contains B blocks of P pages each
  - Page size is around 512B to 4KB
  - Each block contains from 32 to 128 pages
- To write to a page that has already been written to:
  - First need to erase the entire block (1ms)
  - Then we can write the page
  - However we can write pages to a new block and rely on the flash translation layer for the remapping

SOLID STATE DISK CHARACTERISTICS

- Blocks wear out after repeated writes
  - Can be erased only about 100,000 times
  - The disk will eventually stop working
- Flash translation layer uses wear leveling logic:
  - Tries to spread erasures evenly across blocks
  - Remapping: where each logical block is in flash memory changes with time
  - Mapping is internal to disk: transparent to CPU

SOLID STATE DISK



SSD PERFORMANCE INCREASING RAPIDLY

- Intel SSD DC P3700 Series (2TB):
  - Sequential read throughput: 2,800MB/s
  - Random read throughput: 1,800MB/s
  - Sequential write throughput: 2,000MB/s
  - Random write throughput: 700MB/s
- Compare to average hard drive disk: ~100MB/s

FILE SYSTEM IMPLEMENTATION

- A file system must be able to:
  - Given a name, locate the file contents
  - Keep track of which blocks are free
  - Determine which blocks belong to a particular file
  - Maintain administrative data like file permissions, creation and modified times
  - Find the list of free blocks, root directory, and some "other stuff" when the system is started
  - Determine which files are "disk based" and which ones are special

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

FILES

- A file is a sequence of bytes
  - This concept typically is the same in most OSs
- Content is given meaning by user programs
- Type of data (and associated program) determined by:
  - Using the file name (e.g., Windows)
  - Looking at the first few bytes (e.g., Unix)

FILE METADATA

- Attributes are associated with each file:
  - These vary depending on the operating system
- Examples:
  - File size
  - File owner and group
  - Location of the file's data
  - Time of creation/last access/last update
  - File permissions (who can read/write/execute it)
  - Assorted flags (hidden/system/archive/lock/etc)
  - Extended attributes (user- or application-defined)

POSIX FILES

- In POSIX systems (systems based on Unix), every sequence of bytes can be a file:
  - “Regular” binary and text files containing data
  - Directories
  - I/O devices
    - /dev/sda8 (disk partition)
    - /dev/pts/4 (terminal)
    - /dev/cdrom (a CD ROM drive)
    - /dev/mem (the computer's main memory)
    - named pipes, sockets, semaphores

FILE NAMING

- Each file is identified by its name
- Rules for names depend on the operating system and file system
  - MS-DOS/FAT (1981 to 2000)
    - 8 ASCII characters, followed by "." and 3 characters extension
    - Case insensitive (that is MYFILE.DOC == myfile.doc)
  - ISO 9660 CD-ROM (1988)
    - Same as for MS-DOS (to support the lowest common denominator)
    - Extensions support longer file names

FILE NAMING (CONT.)

- Rules for names
  - Windows NT to 10/NTFS (1993 to current)
    - 255 Unicode characters (some exceptions), case sensitive (can be switched off)
    - Many Windows tools are case insensitive
  - Unix/Linux
    - 255 ASCII characters (except NULL and /), case sensitive
    - UTF-8 can be used with all current versions of Linux

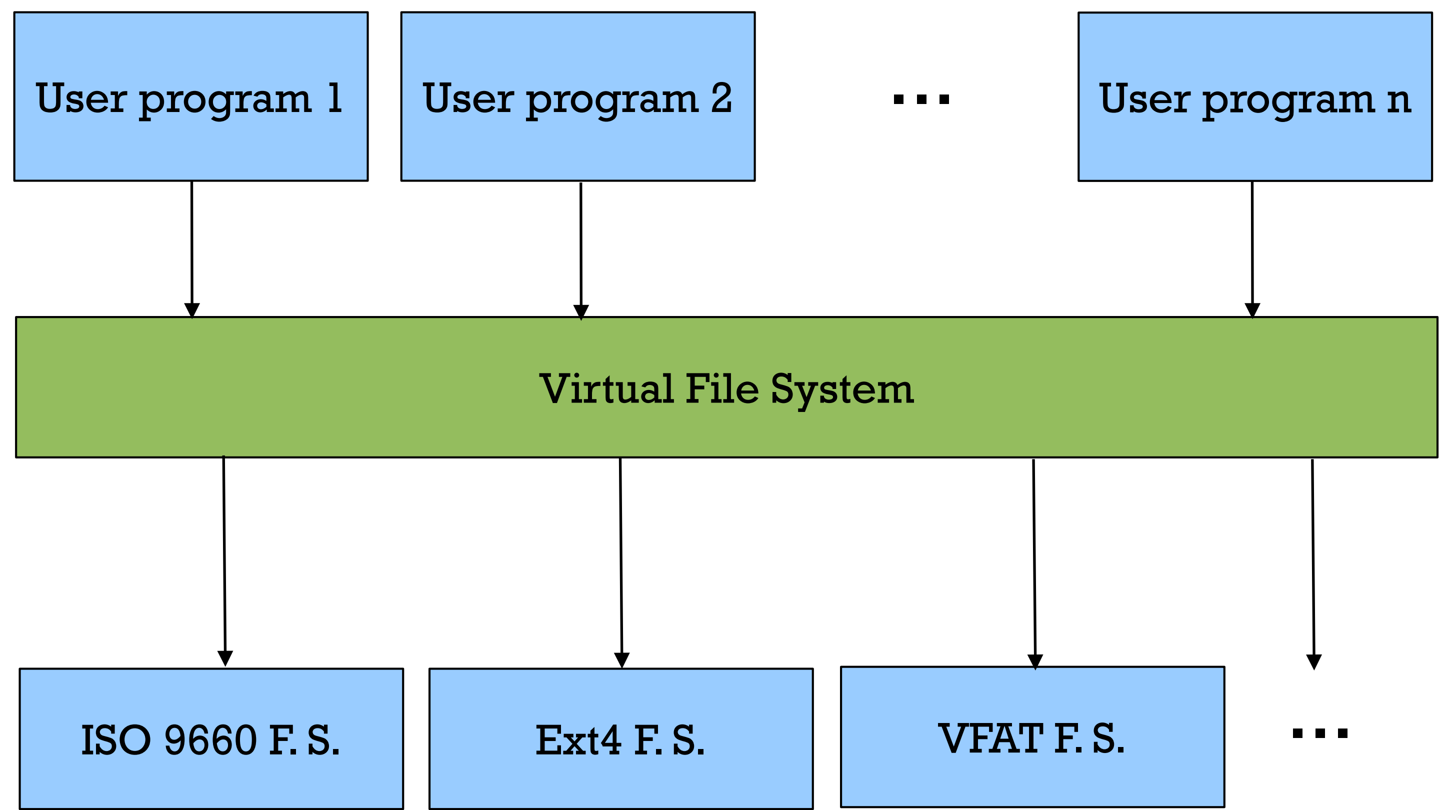
DIRECTORIES

- A directory is just a file whose data contains a list of entries
  - Each entry contains information about one file or subdirectory
- Every file must be an entry in some directory
  - That includes directories, except for the top-level directory

VIRTUAL FILE SYSTEMS

- File System Selection:
  - User programs make system calls to access various operations
  - A layer called the Virtual File System performs the parts of the operations that are common to all file systems
  - The virtual file system calls low-level functions to accomplish specific tasks
  - Each file system must implement these low-level functions appropriately

VIRTUAL FILE SYSTEMS



FILE SYSTEM IDENTIFICATION

- MS-DOS, Windows
  - Each file system is assigned a letter name
    - A:\ : floppy disk
    - C:\ : primary hard disk
    - Z:\ : drive on a server somewhere on the network
  - This letter is used to decide which file system to pass the request to
  - Hence the user must know which file system contains the file he/she wants to access

FILE SYSTEM IDENTIFICATION

- Unix/Linux
  - There is a root file system "/" at the top of the hierarchy
  - Every other file system appears as a subdirectory in that file system
  - The user need not even be aware that multiple file systems are involved

FILE RESOLUTION PROCESS

- Given a path:
  - Find the file system associated to the path
  - In the file system's root directory, find the entry corresponding to the first component of the path (after the drive or mountpoint)
  - In the directory found above, find the entry corresponding to the next component of the path
  - Repeat until the last component of the path is found
  - Return the file pointed to by the last entry

FILE RESOLUTION PROCESS EXAMPLE

- Example: if the path is "/media/cdrom/docs/grades/a1.csv"
  - Find the file system (e.g., /media/cdrom is associated to the CD-ROM device)
  - In the CD-ROM root, find the entry named "docs"
  - In the directory found above, find the entry named "grades"
  - In the directory found above, find the entry named "a1.csv"
  - Return the file pointed to by the entry above

ISO-9660 FILE SYSTEM LAYOUT

- CD-ROMs are read-only
  - The file system layout is simpler
  - Files can always be stored in contiguous blocks
- Directory entry links name to specific location in disk
  - Also includes attributes like size, date/time and flags

MS-DOS (FAT) FILE SYSTEM LAYOUT

- No longer used normally with computers, but in
  - Most digital cameras
  - MP3 players
  - iPods (if formatted on a Windows machine)
- Simple, easy to implement
- Cons:
  - Uses a lot of memory for allocation table
  - Random access to file data is slow
  - Prone to fragmentation

DISK ORGANIZATION

- Disk organized in sectors starting at 0
  - Equivalent to one big array of sectors
- Given the sector number, the drive determines
  - which cylinder to seek to
  - which head to use
  - which sector to read

Boot block Partition info	FAT1	FAT2 (duplicate)	Root Directory	Sectors (blocks/clusters) Used for the files and directories

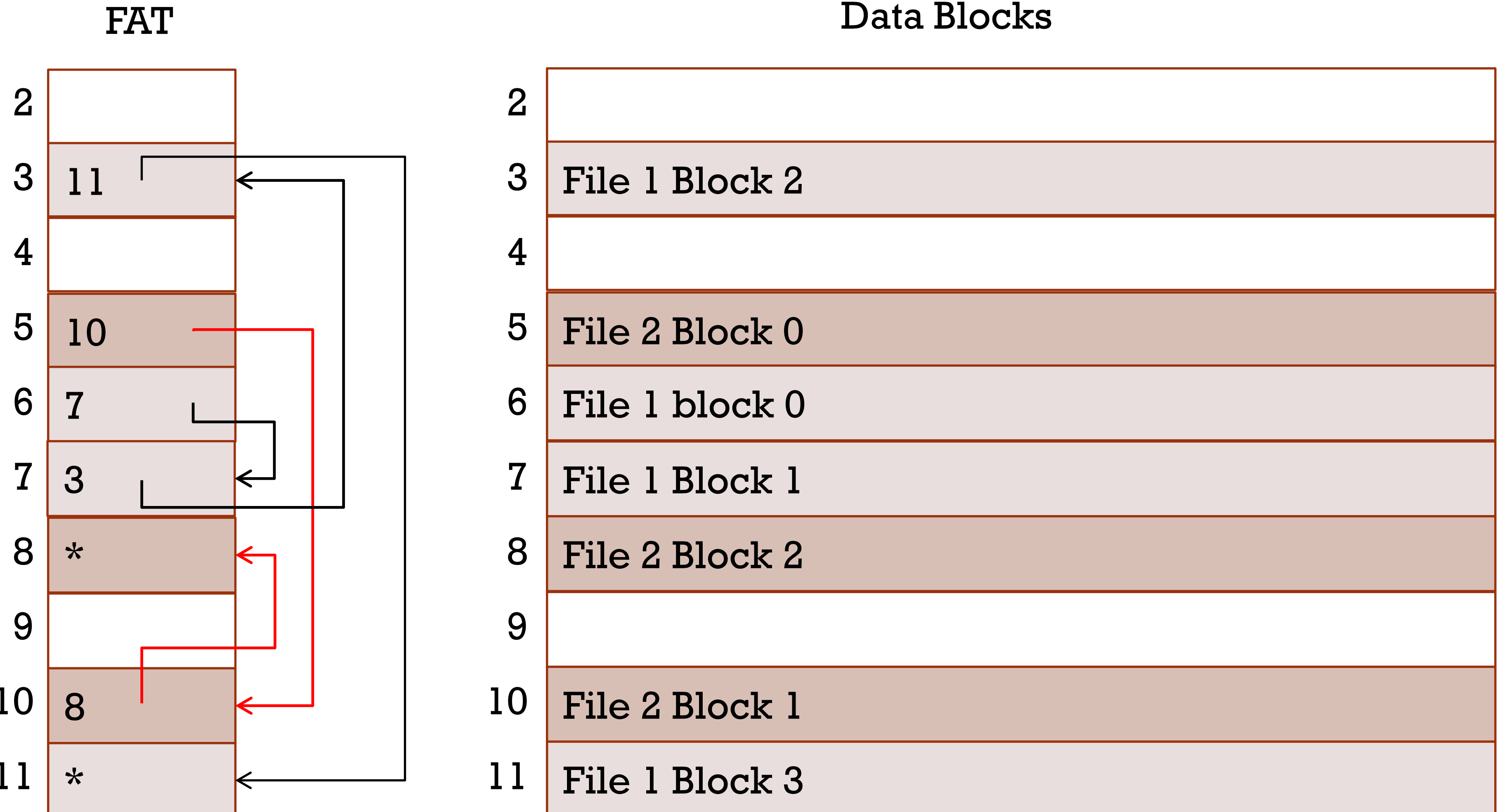
MS-DOS FILE SYSTEM LAYOUT

- Root directory stored in fixed location
- Files and non-root directories are stored in general cluster area
- Directory entry links name to first block (cluster) number
  - Also includes metadata like date/time, size and flags

FILE ALLOCATION TABLE (FAT)

- Space is managed using a File Allocation Table (FAT)
  - Each block represented by a 12, 16 or 32-bit word
  - A word contains the number of the next block in file
    - In other words: each file is a linked list of blocks
  - Typically two copies of the FAT are stored on disk (redundancy)
  - A copy is always kept in memory

FAT-32 SYSTEM

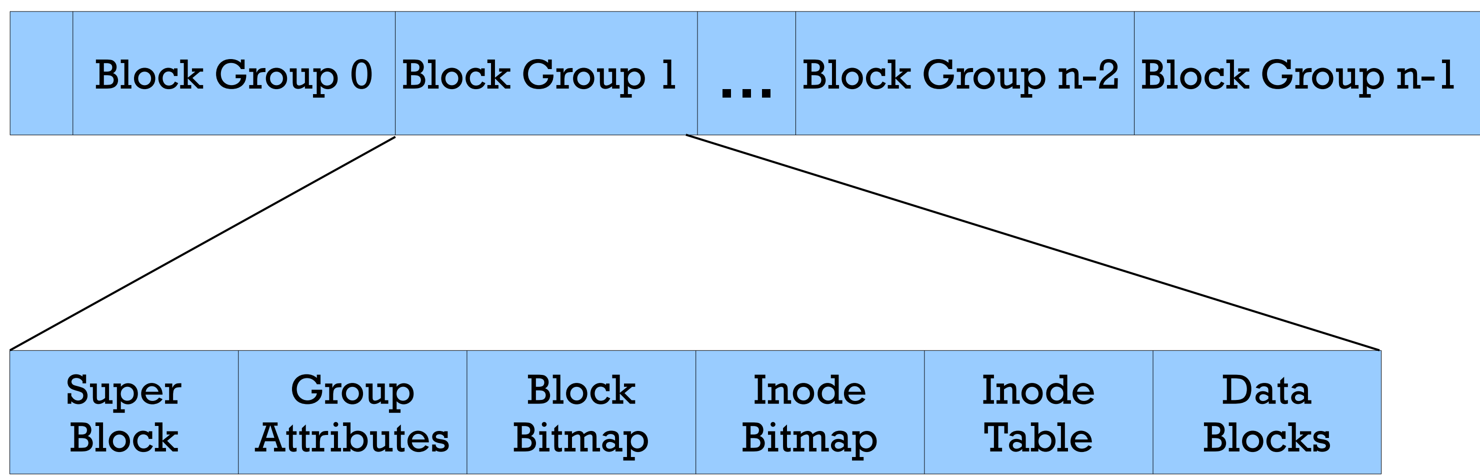


FRAGMENTATION IN MS-DOS FILE SYSTEM

- Moving the hard disk's head from one position to another takes a long time (comparatively)
- Hence file operations are much faster if all of the file's data is stored in neighbouring blocks
- Unfortunately repeated use of the file system may scatter some files all over the disk

LINUX FILE SYSTEM LAYOUT

- File system organized in groups
- Information about free/occupied blocks is kept separate from the information used to locate data



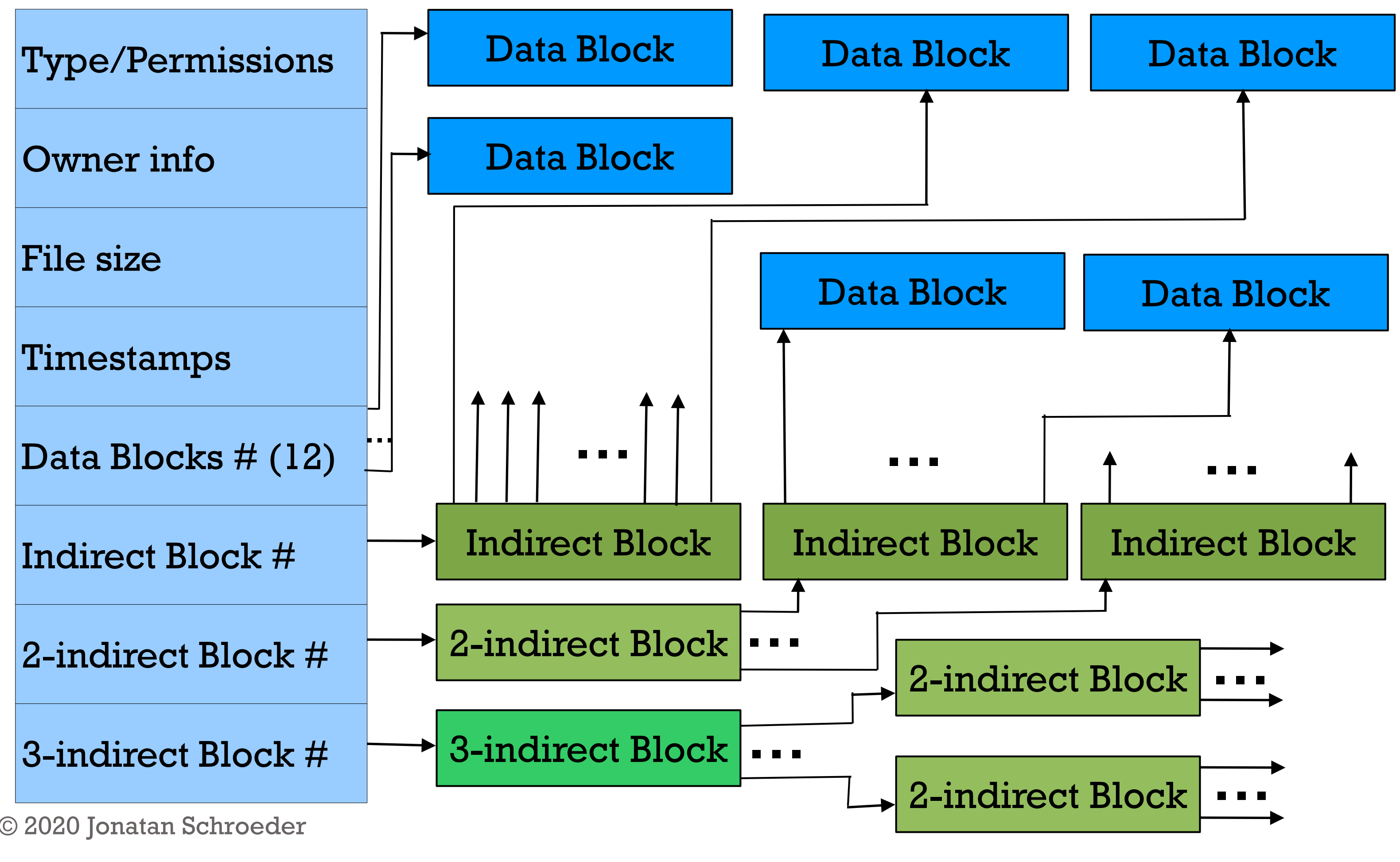
LINUX FILE SYSTEM: SUPERBLOCK

- Superblock is located at the start of the disk
- Contains global file system information
  - How many inodes and data blocks the disk holds
  - How many inodes and data blocks are free
  - How many inodes and data blocks each group has
  - Dirty flag (was the system shut down cleanly?)
  - Etc.
- A lost superblock is a disaster
  - So each block group holds a redundant copy of the superblock

FILES IN LINUX FILE SYSTEM

- A file is represented by an inode
  - Contains the file's attributes (but not its name)
- File's data is stored in data blocks
  - Inode points directly to first 12 blocks
  - Remaining blocks organized in a tree-like fashion
    - An *indirect block* points to some data blocks
    - A *double-indirect block* points to some indirect blocks

LINUX FILE SYSTEM LAYOUT



LINUX FILE SYSTEM: DIRECTORIES

- A directory contains file names, and an inode number for each file name.
  - Metadata is not stored in directory entry (it is stored in inode)
- The first entry of every directory is . : a reference to the directory itself.
- The second entry of every directory is .. : a reference to the parent directory.

LINUX FILE SYSTEM: RESOLUTION

- To read data from a file:
  - Find the inode for file (usual path resolution)
  - If the offset we need is within the first 12 blocks, read the block pointed to by the corresponding direct block
  - If the offset is in the range of the 1-indirect block, read the indirect block to find the block number we need, and then read the block identified by that number
  - If offset is in 2- or 3-indirect block, do the same but with additional indirection
- Random access to large files is much faster than for the MS-DOS file system

LINUX FILE SYSTEM: HARD LINKS

- *Hard links*: several directory entries may refer to one inode
  - This is the case for . and ..
- Can be used to give a program several names
  - The program behaviour may depend on the name used.
  - Example:

```
ls -ali /bin
1308482 -rwxr-xr-x  3 root root   31112 2010-09-11 06:48 bunzip2*
1308482 -rwxr-xr-x  3 root root   31112 2010-09-11 06:48 bzcat*
1308482 -rwxr-xr-x  3 root root   31112 2010-09-11 06:48 bzip2*
```

  - All files must belong to the same file system (why?)

LINUX FILE SYSTEM: SOFT LINKS

- Unix/Linux also support symbolic (soft) links
  - A file whose contents is the name of another file
  - Example:

```
ls -al /lib
-rw-r--r--  1 root root  534832 2010-10-21 19:02 libm-2.12.1.so
lrwxrwxrwx  1 root root      14 2010-10-22 18:42 libm.so.6 -> libm-2.12.1.so
```

  - The second file may be on a different file system
    - In fact it does not even need to exist

FRAGMENTATION

- Unlike the MS-DOS file system, modern file systems (ext*fs, NTFS) try to keep files together
- For Linux:
  - files are kept within a block group if possible
  - files in same directory are kept within a block group if possible
  - large files are written to large free areas, whereas small files are stored in smaller free areas
- Fragmentation still happens, but much more slowly
  - normally only becomes a problem if the file system is very full

THE ROLE OF UNIX I/O

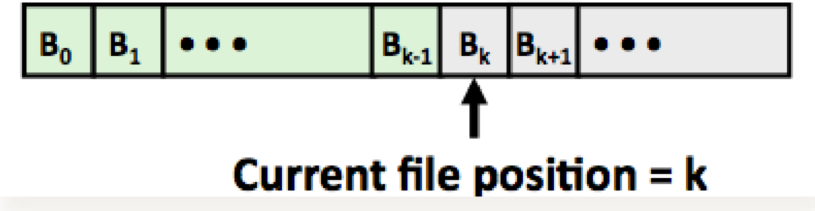
- I/O: the process of copying data between memory and external devices
  - File system works at the block level
  - Applications work at the byte level
  - Unix I/O converts the byte level access to block level operations
- Why we study it?
  - helps understand how I/O functions provided by programming languages work
  - high level I/O functions provided by programming languages are not suitable for some applications
  - helps understand system structuring concepts

Application
Unix I/O
File System
Disk Drive

File System Layering

BASIC UNIX I/O OPERATIONS

- Basic Unix I/O operations (system calls):
  - Opening and closing files
    - open() and close()
  - Reading and writing a file
    - Based on current location: read() and write()
    - Random location: pread() and pwrite()
  - Changing the current file position (seek)
    - Indicates next offset into file to read or write
    - Reading or writing a file implicitly changes the file offset.
    - lseek()



EXAMPLE CODE

```
char* path;  // file name
...
int source_fd;
if ((source_fd = open(path, O_RDONLY)) < 0) {
  perror("Open source failed:");
  exit(2);
}
char buf[512];
int chars_read;
chars_read = read(source_fd, buf, sizeof(buf));
while (chars_read > 0) {
  // Do something
  chars_read = read(source_fd, buf, sizeof(buf));
}
```

An integer >= 0 on success, negative on failure

File descriptor passed in and used by kernel (OS) to determine what file to read the data from

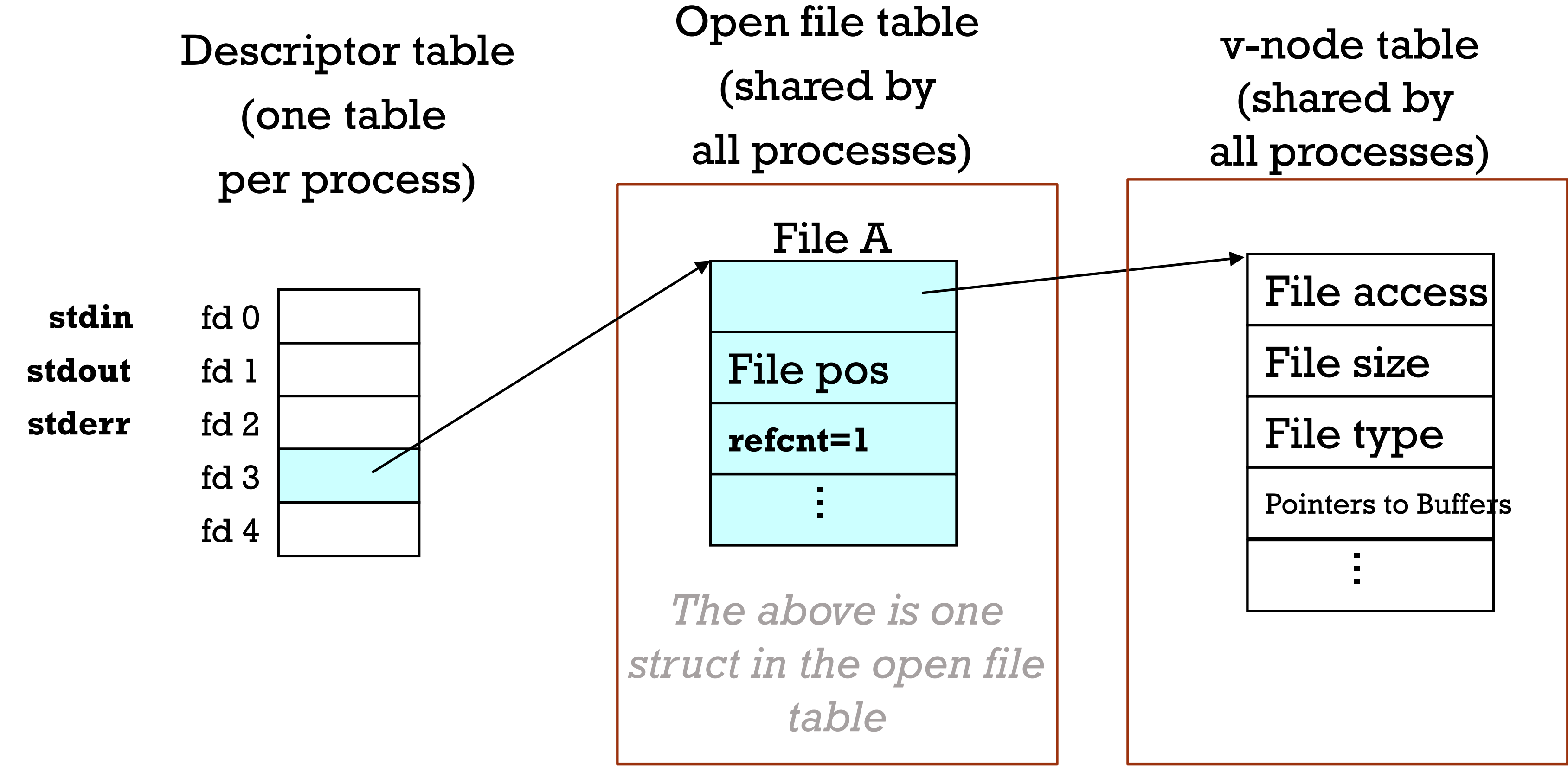
FILE DESCRIPTOR - SUMMARY

- Open returns a small integer called a file descriptor
- Application passes this value back to the kernel in subsequent requests to work with a file
- Each process created starts with three open files:
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

FILE DESCRIPTOR: KERNEL'S VIEW

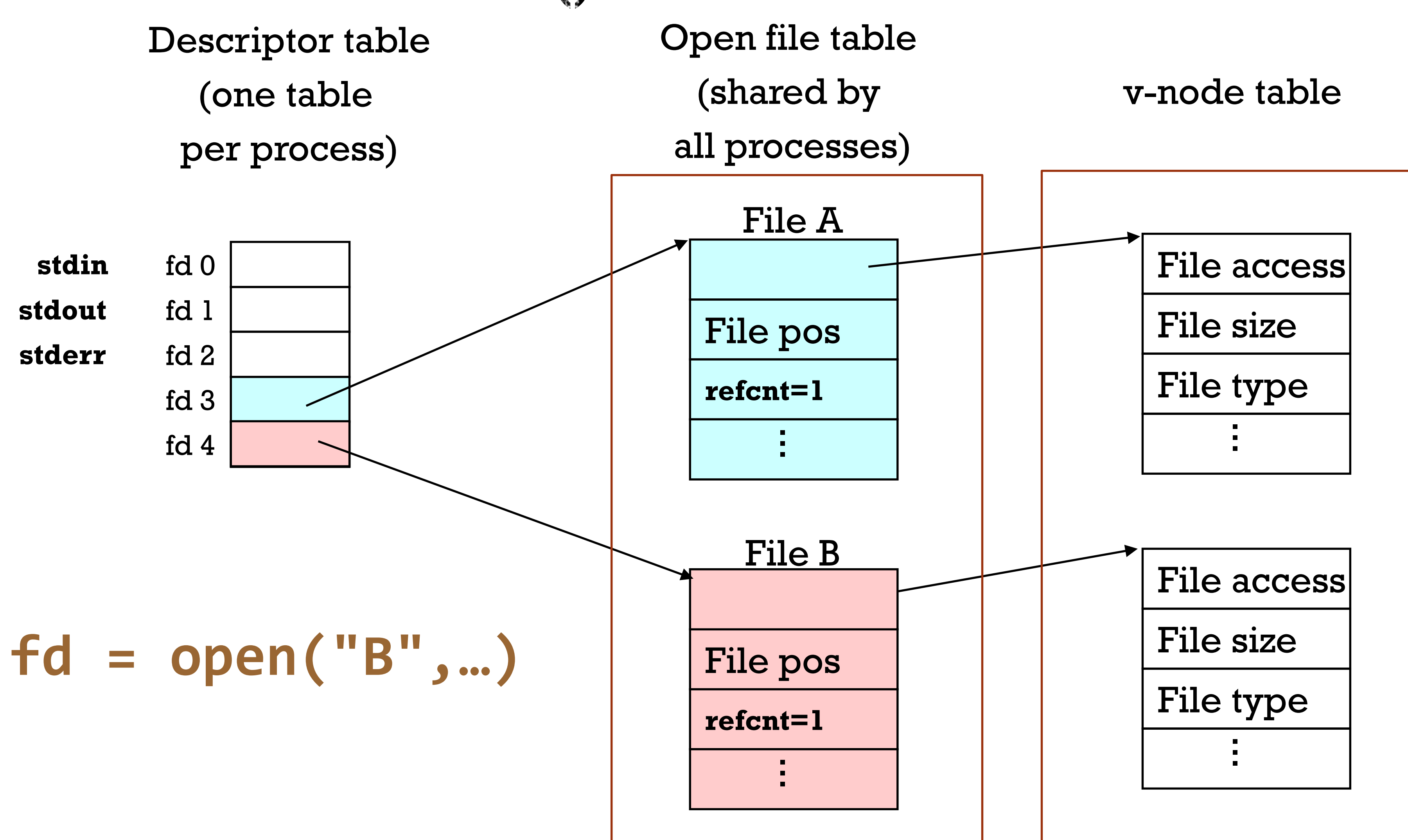
- Each process has associated with it a fixed size file descriptor table
  - The file descriptor is just the index into this table
- Each active entry in the table identifies an entry in a shared system-wide open file table
- Entries are created in the open file table (and a process FD table) each time open() succeeds

THE KERNEL VIEW



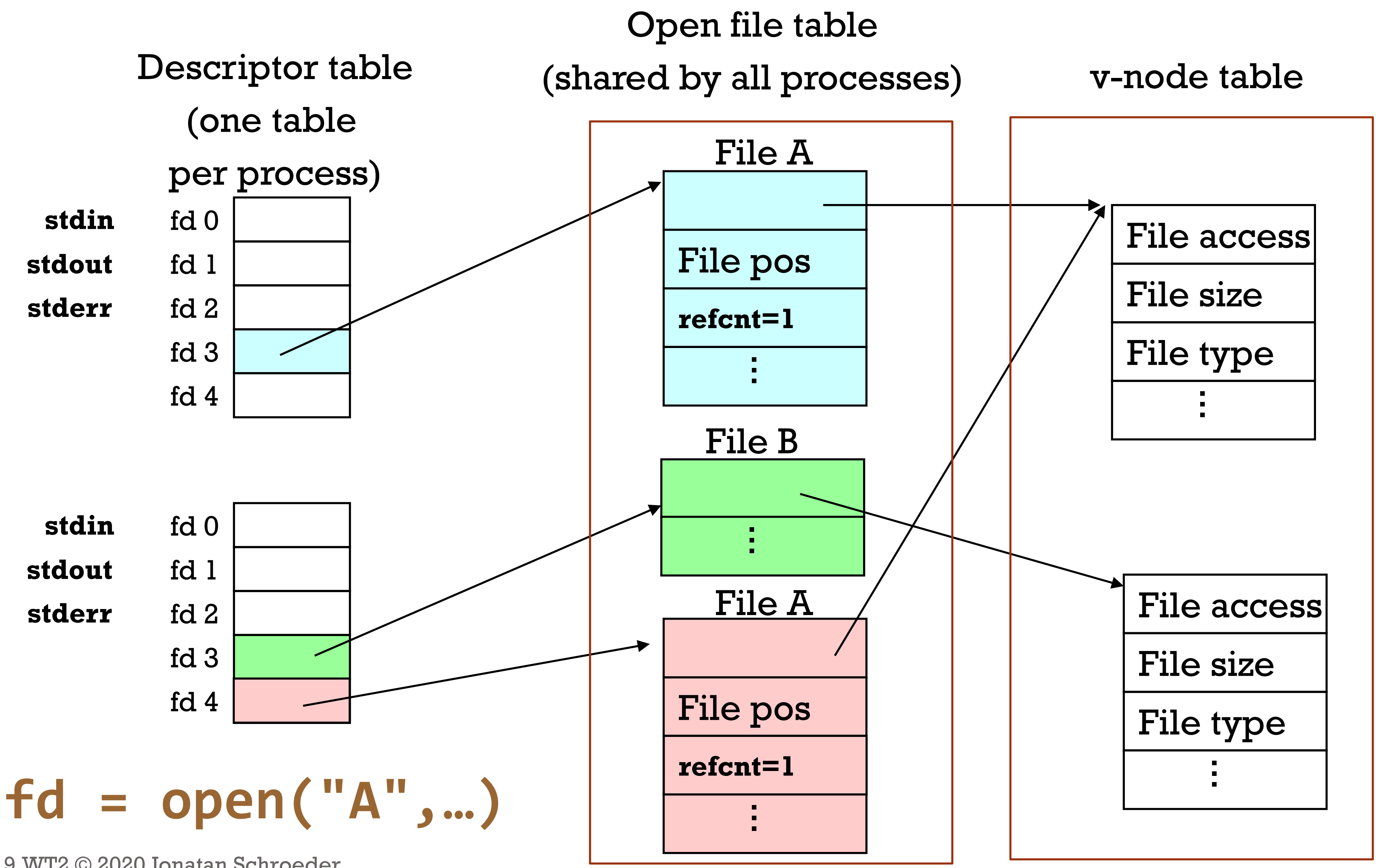
Adapted from: *Computer Systems: A Programmer's Perspective*

ACTIONS ON OPEN()



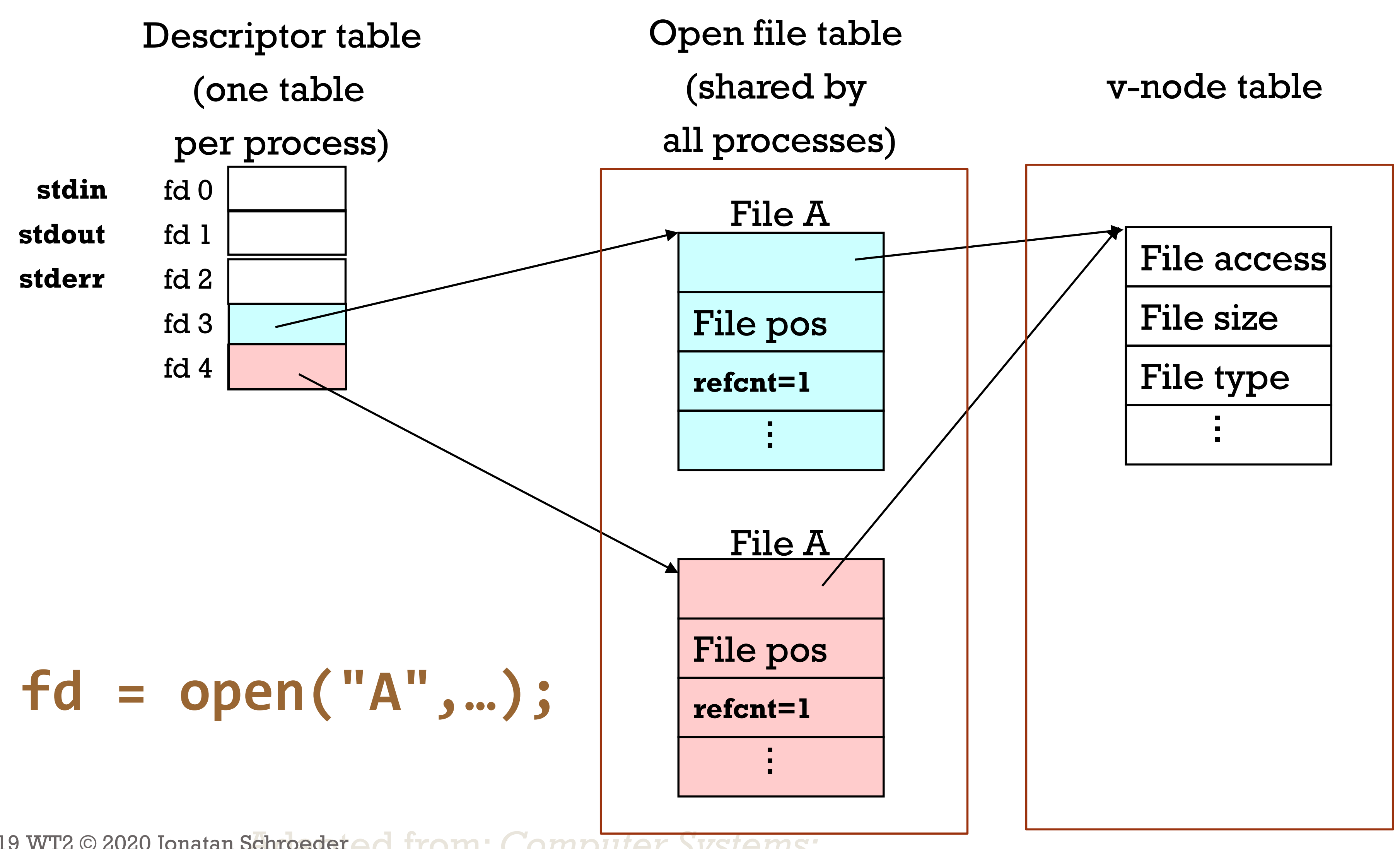

```
fd = open("B",…)
```

SAME FILE DIFFERENT PROCESS




```
fd = open("A",…)
```

SAME FILE SAME PROCESS



HOW CAN WE IMPROVE PERFORMANCE?

- Given what we know, are there interesting things we can do at the application layer to speed things up?
- A system call is several orders of magnitude more expensive than a function call

Application
Unix I/O
File System
Disk Drive

File System Layering

CACHING IN THE APPLICATION

- Applications can use caching to improve performance just like the kernel
- Most I/O has both
  - Spatial locality
  - Temporal locality
- Application level functions in the Standard I/O library of C take advantage of this
- All these functions are declared in the header stdio.h

This is buffered

Application

Unix I/O

File System

Disk Drive

File System Layering

STANDARD I/O FUNCTIONS IN C

- The C standard library (libc.so) contains a collection of higher-level standard I/O functions
  - Different OS architectures may adapt these to their own I/O interface
- Examples of standard I/O functions:
  - Opening and closing files (fopen and fclose)
  - Reading and writing bytes (fread and fwrite)
  - Reading and writing text lines (fgets and fputs)
  - Formatted reading and writing (fscanf and fprintf)

 © 2020 Jonatan Schroeder

STDIO

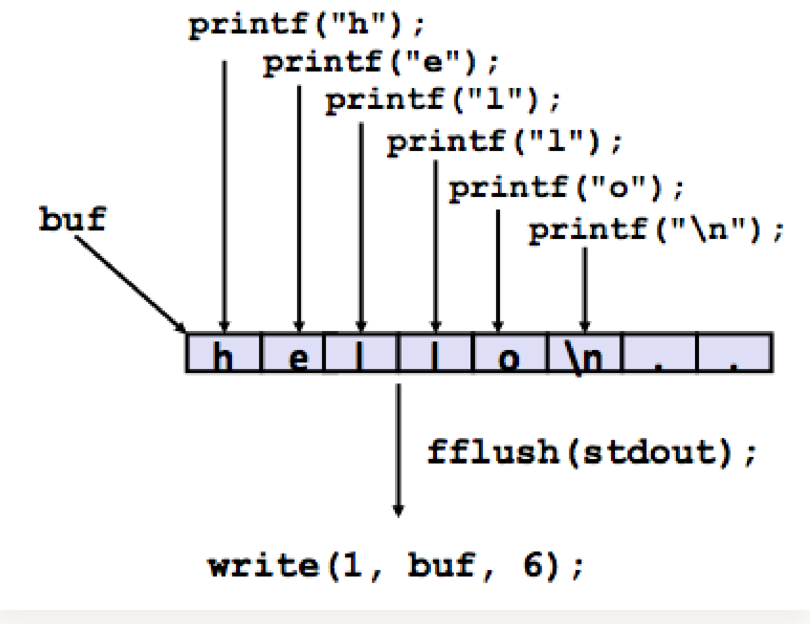
- Instead of returning a file descriptor, fopen() returns a FILE *
- The FILE struct contains:
  - actual file descriptor
  - pointer to a buffer
  - position in buffer
  - other bookkeeping information

HOW IT WORKS - WRITES

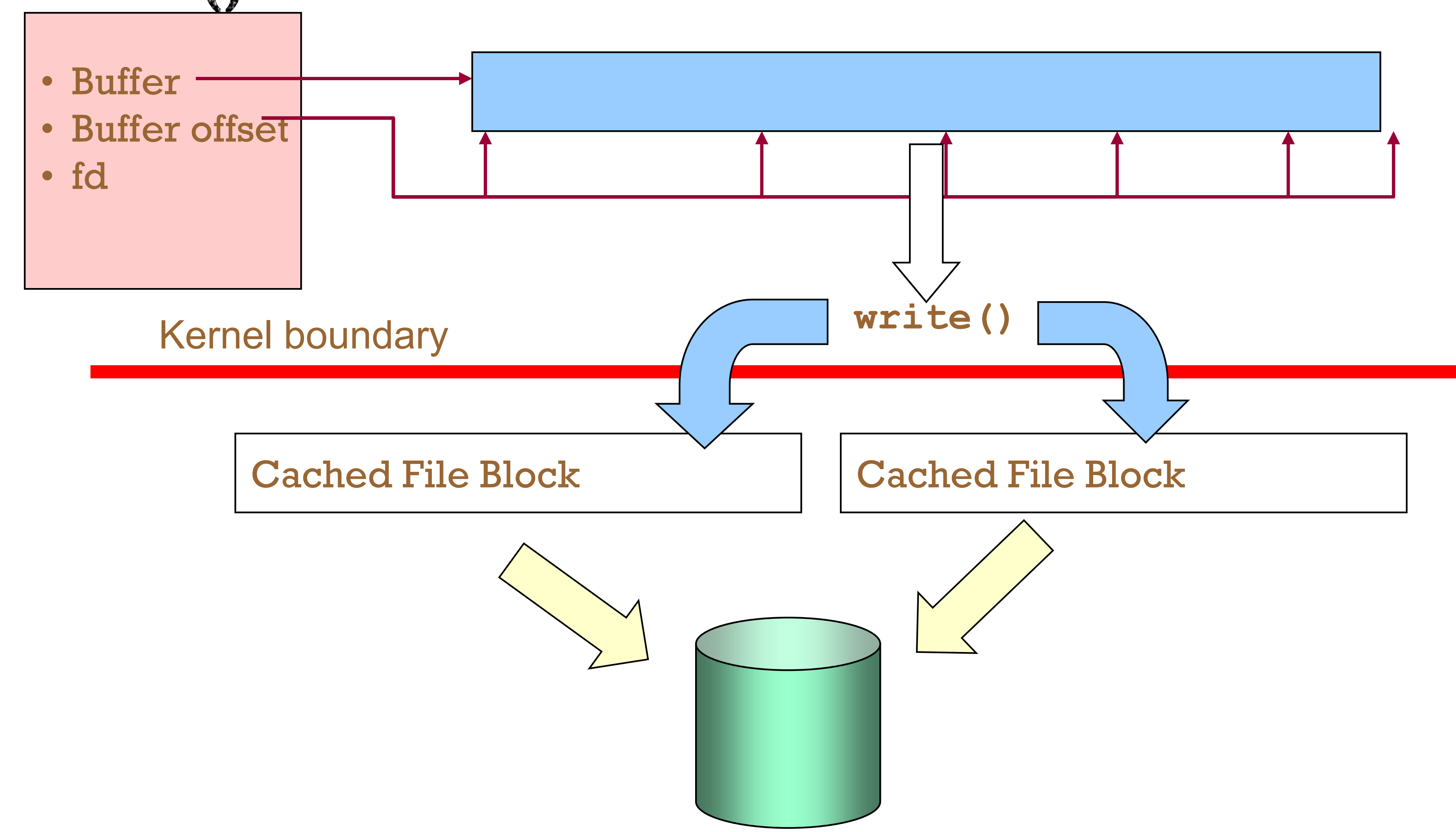
- When fwrite() is called, bytes are copied to the stream buffer
- If the stream buffer fills during the fwrite()
  - write() called to "write" the stream buffer
  - Stream buffer cleared

BUFFERING IN STANDARD I/O

- Standard I/O functions use buffered I/O
- Buffer flushed to output fd on:
  - fflush() call
  - Buffer flushing mode (setvbuf function):
    - Unbuffered (_IONBF): no buffer used, flushed on every write
    - Line-buffered (_IOLBF): on line break or buffer full (default for terminal)
    - Fully buffered (_IOFBF): on buffer full (default for file)



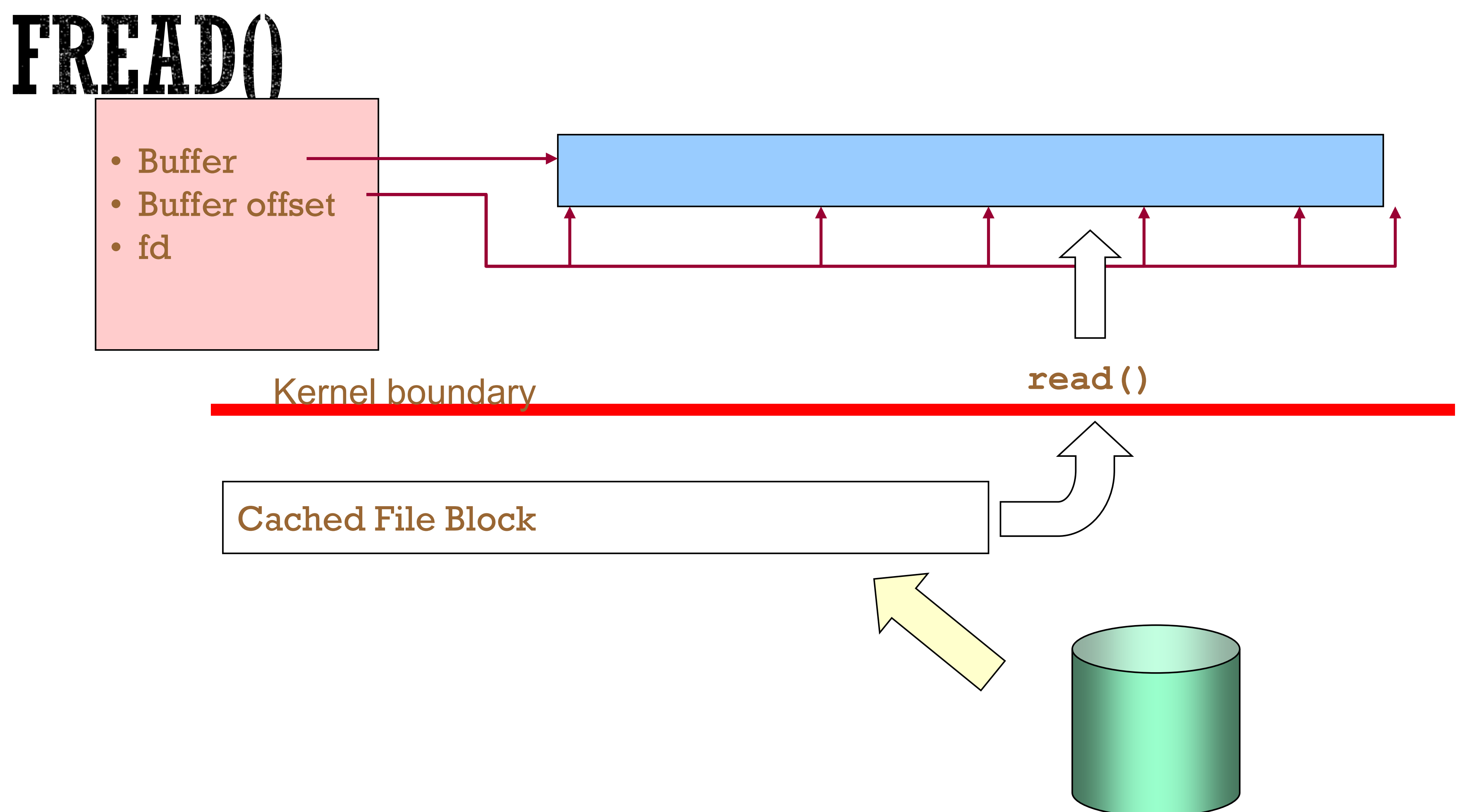
FWRITE()



HOW IT WORKS - READS

- When fread() is called, bytes are copied from the stream buffer to the application designated location
- If the stream buffer empties during the fread()
  - read() called to refill the stream buffer
  - Position in stream buffer reset

FREAD()



ANALYSIS

- Costs over doing a system call
  - Need extra buffer space
  - One extra set of copies
  - Bookkeeping to ensure the stream buffer exactly matches real file location
  - I/O to random locations can be inefficient
- Advantage over system call
  - If application I/O requests are much smaller than what the OS disk buffer holds then greatly reduces the number of system calls
  - System calls are very expensive

PROS AND CONS OF UNIX I/O

- Pros
  - Unix I/O is the most general and lowest "overhead" form of I/O
  - All other I/O packages are implemented using Unix I/O functions
  - Unix I/O provides functions for accessing file metadata
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers
- Cons
  - Dealing with short counts is tricky and error prone
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone
- Both of these issues are addressed by the standard I/O and RIO packages
  - RIO package is described in text but you don't have to know it

PROS AND CONS OF STANDARD I/O

- Pros:
  - Buffering improves efficiency by decreasing the number of read and write system calls
  - Short counts are handled automatically
- Cons:
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers
  - Standard I/O is not appropriate for input and output on network sockets
    - Simultaneous read and write of same file descriptor requires seeking, not supported by sockets

WORKING WITH BINARY FILES

- Binary file examples
  - Object code, Images (JPEG, GIF), Documents (DOC, PDF)
- Functions you shouldn't use on binary files
  - Line-oriented I/O such as fgets, scanf, printf, rio_readlineb
- Different systems interpret a line break differently:
  - Linux and Mac OS X: LF(0x0a) ['\n']
  - HTTP servers & Windows: CR+LF(0x0d 0x0a) ['\r\n']
- String functions
  - strlen, strcpy
  - Interprets byte value 0 (end of string) as special

PROCESSES AND THREADS

Unit 6

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

OPERATING SYSTEMS: OUTLINE

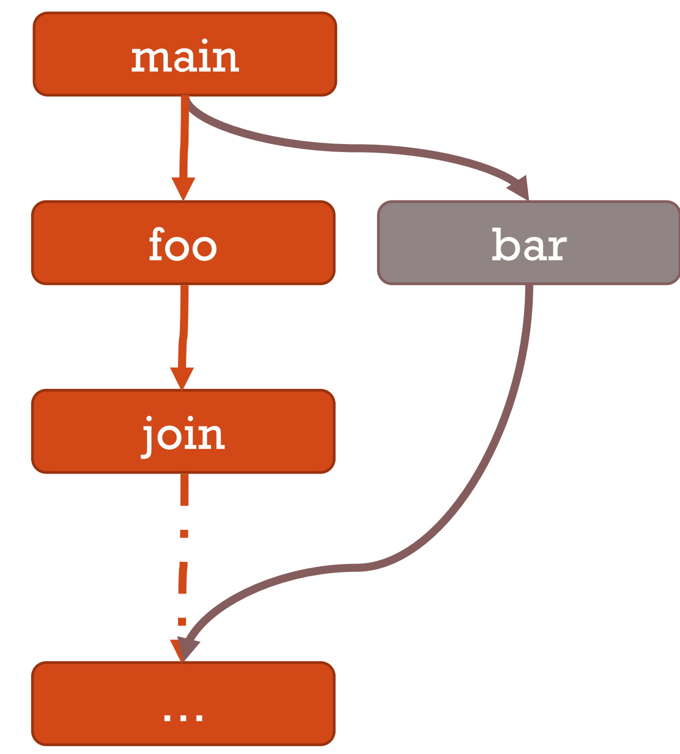
- Asynchronous Programming
  - Threads
  - Synchronization principles
  - Mutual exclusion locks
  - Condition variables
  - Semaphores
- Operating Systems and Processes
  - Exceptions
  - User and kernel modes
  - Processes and process control

VIRTUALIZATION OF THE CPU

- Modern systems require multiple processes to run simultaneously
- Some processes may require the CPU to be idle for some time
  - Something else can be done in the mean time
- Some CPUs have multiple cores
- Sometimes processes need to share use of a CPU core

THREAD

- Abstraction for execution
  - For programmer, looks like sequential flow of execution (private CPU)
  - Can be stopped and started (may be running or not)
  - Physical CPU multiplexes multiple threads at different times



VIRTUAL PROCESSES (THREADS)

- Each thread should have the illusion of isolation in CPU
- Each thread has:
  - Own variables (stack, registers)
  - Own *synchronous* control flow (program counter)
- Threads can resume other threads
  - Do they have to?

THREAD OPERATIONS

- Creating and starting a thread
  - Like an asynchronous procedure call
  - Starts a new thread of control to execute a procedure
- Stopping (blocking) a thread
  - Save thread's state and switch to a different thread
- Re-starting (unblocking) a thread
  - Restore thread's state and continue running
- Joining with a thread
  - Block current thread until a target thread completes
  - Can obtain the return value of the target thread
  - Turns CPU back into a synchronous procedure call

POSIX THREADS

- Linux support threads through a standard (POSIX) interface knows as pthreads
- To compile, we need to give gcc the -lpthread flag:

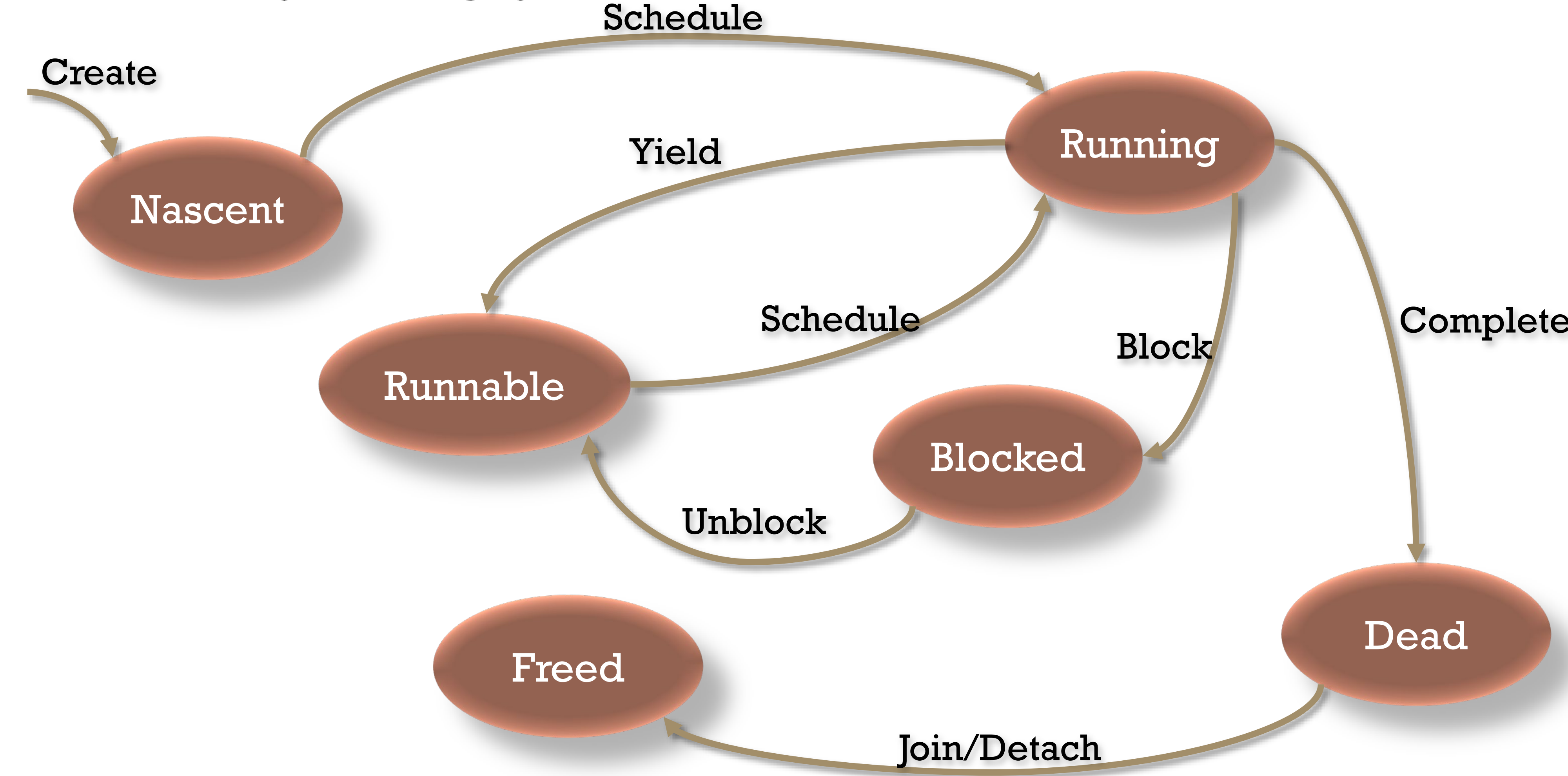
```
gcc -o pingpong pingpong.c -lpthread
```

- All threads for a process share
  - The same memory space
  - The same global variables and heap
  - The same file descriptors (for open files).
- Each thread has its own state and stack

COMMON PTHREAD OPERATIONS

- pthread_create: create and start a thread
  - Receives a function as parameter, function contains code to execute in thread
  - Caller function continues executing without waiting for thread to complete
- pthread_join: join a thread
  - Block until thread completes
- pthread_detach: detach a thread
  - Allows thread to free its resources as soon as it completes
  - Signals that a join is not expected to be called
- pthread_yield: yields to a different thread

THREAD STATUS



THREAD SCHEDULING

- Scheduling a thread is:
  - Deciding which threads should run, and when
  - When there are more runnable threads than processors
  - Involves a ***policy*** and a ***mechanism***
- Thread Scheduling Policy
  - Set of rules that determines which threads should be running
  - Do some threads have higher ***priority***?
  - Should threads get ***fair*** access to the processor?
  - Should threads be guaranteed to ***make progress***?
  - Should one thread be able to ***preempt*** another?

PRIORITY ROUND-ROBIN SCHEDULING POLICY

- Priority: number assigned to each thread
  - Thread with highest priority goes first
- When choosing the next thread to run
  - Run the highest priority runnable thread
  - When threads have the same priority, run thread that has waited the longest
- Implementation (mechanism)
  - Organize Ready Queue as a priority queue
    - Highest priority first, FIFO (first in first out) otherwise

PREEMPTION

- Preemption occurs when
  - a "yield" is forced upon the current running thread
  - current thread is stopped to allow another thread to run
- Priority-based preemption
  - when a thread is made runnable (e.g., created or unblocked) with higher priority than current-running thread, it preempts that thread
- Quantum-based preemption
  - each thread is assigned a runtime "quantum" (time slot)
  - thread is preempted at the end of its quantum
  - How long should quantum be?

PROBLEMS WITH THREADS

- Threads have been introduced to:
  - Exploit parallelism: use different processors to run things at the same time
  - Manage asynchrony: do something else while waiting for I/O Controller (or other blocking condition)
- But they introduce other problems:
  - Coordinating access to memory shared among threads (e.g., static variables)
  - Unpredictable control flow transfers among threads (preemption)

SYNCHRONIZATION

- Ensure mutual exclusion of critical sections
  - Threads may share data structure
  - Operations involve multiple memory accesses
  - Accesses can be arbitrarily interleaved
- Wait for and signal the occurrence of events
  - One thread may require the completion of a task performed by another
  - This task is not completion (so pthread_join is not suitable)

SHARED DATA STRUCTURE

- Assume the code on the right
  - Stack implemented as an array
- Could it cause problems?

```
int n;
int array [SIZE];
void push (int i) {
  if (n < SIZE) {
    array [n] = i;
    n++;
  }
}
int pop () {
  if (n > 0) {
    n--;
    return array [n];
  } else
    return -1;
}
```

A SEQUENTIAL TEST WORKS...

```
void pop_driver (int c) {
  int e;
  while (c--) {
    do {
      e = pop ();
    } while (e != -1);
    /* consume value of e */
  }
}
```

```
void push_driver (int c) {
  while (c--)
    push (/* some value */);
}

int main (void) {
  push_driver (100);
  pop_driver (100);
  assert (n==0);
}
```

A CONCURRENT TEST MAY NOT WORK

```
int main (void) {
  …
  pthread_create(&et, NULL, push_driver, 100);
  pthread_create(&dt, NULL, pop_driver, 100);
  pthread_join(et, NULL);
  pthread_join(dt, NULL);
  assert (top==0);
}
```

THE PROBLEM

- Shared data
  - Data structure that could be accessed by multiple threads
  - Concurrent access to shared data can lead to inconsistencies
- Critical sections
  - Sections of code that access shared data
- Race condition
  - Simultaneous access to critical section by multiple threads
  - Conflicting operations on shared data structure are arbitrarily interleaved
  - Unpredictable (non-deterministic) program behaviour
  - Usually hard to debug as well

EXAMPLE: A BOUNDED BUFFER

- Consider the following situation:
  - We have one or more producers that generate values
  - We have one or more consumers that use them
  - We need to send the values from the producers to the consumers
- A bounded buffer
  - has a fixed capacity (for example, N = 4)
  - producers write values to it; if it's full they wait
  - consumers read values from it; if it's empty they wait
- Example application: video playback (e.g., Youtube)
  - Producer loads frames from source (e.g., streaming service)
  - Consumer shows frames on the screen

EXAMPLE: A BOUNDED BUFFER

- Check example file, threads/boundedbuf.c
- How do we know our implementation is correct?
  - Does it have any race conditions?
- Observation:
  - send() updates buf->in, but not buf->out
  - receive() updates buf->out, but not buf->in
- Single-writer principle: if each variable only has one writer, then communication becomes easier

EXAMPLE: A BOUNDED BUFFER

- The correctness of our algorithm depends on all of:
  - A single writer for every shared (global) variable
  - Each thread has a [virtual] processor
    - They both have the ability to continue running
  - Memory has read/write coherence
  - Memory write operations are seen in program order
  - The values buf->in and buf->out never overflow
  - The values buf->in and buf->out have before-and-after atomicity

EXAMPLE: A BOUNDED BUFFER

- What if we had more producers/consumers?
  - Race conditions on buf->in and buf->out
- How do we fix this?
  - We need to make a group of operations before-or-after atomic

THREAD SYNCHRONIZATION

- Consider the following code (pingpong.c)

```
void *counting_thread(void *arg) {
    char *name = arg;
    int i;
    for (i = 0; i < LIMIT; i++) {
        counter++;
    }
    return 0; /* Success */
}
```

- The counter is unprotected
  - Other threads can read its value in the middle of the operation

MUTUAL EXCLUSION

- Mechanism to ensure critical sections are executed by one thread at a time
- Usually implemented in software, with some special hardware support
- Reading and writing may be handled differently
  - Two threads reading don't interfere with each other
  - Two threads writing may cause race condition
  - A thread writing can cause inconsistent read in other threads
  - More on that later

MUTUAL EXCLUSION USING LOCKS

- Lock semantics
  - A ***lock*** is either ***held*** by a thread or ***available***
  - At most one thread can hold a lock at a time
  - A thread attempting to ***acquire*** a lock that is already held is forced to wait
- Lock operations (primitives)
  - ***lock***: acquire lock, wait if necessary
  - ***unlock***: release lock, if other threads are waiting allow one of them to run

USING LOCKS FOR THE SHARED STACK

```
void push_cs (int e) {
  lock(&aLock);
  push(e);
  unlock(&aLock);
}
```

```
int pop_cs () {
  int e;
  lock(&aLock);
  e = pop();
  unlock(&aLock);
  return e;
}
```

IMPLEMENTING SIMPLE LOCKS

- Naïve implementation:
  - Shared global variable for synchronization
  - Lock loops until variable is 0, then sets it to 1
  - Unlock sets variable to 0

```
void lock (int *lock) {
  while (*lock) {}
  *lock = 1;
}
void unlock (int *lock) {
  *lock = 0;
}
```

SIMPLE LOCK PROBLEM

- There is still a race condition
  - If two threads lock at the same time, they could check that lock is zero at the same time
  - Two threads would "acquire" the lock
- Race happens even in machine-code

```
loop:    movq  (%rsi), %rax # assume %rsi has &lock
         testq %rax, %rax
         jne   loop
         movq  $1, (%rsi)
```

DEKKER'S ALGORITHM

```
p0:
   entrance_intents[0] = true;
   while (entrance_intents[1]) {
    if (turn ≠ 0) {
     entrance_intents[0] = false;
     while (turn ≠ 0) {
      // busy wait
     }
     entrance_intents[0] = true;
    }
   }
   // insert critical section here
   turn = 1;
   entrance_intents[0] = false;
```

```
p1:
   entrance_intents[1] = true;
   while (entrance_intents[0]) {
    if (turn ≠ 1) {
     entrance_intents[1] = false;
     while (turn ≠ 1) {
      // busy wait
     }
     entrance_intents[1] = true;
    }
   }
   // insert critical section here
   turn = 0;
   entrance_intents[1] = false;
```

DEKKER'S ALGORITHM

- Advantages
  - It does not require any support from the CPU
- Disadvantages
  - The process spends a lot of time looping (live-lock)
  - Optimizing compilers don't know about concurrency
    - they may store turn in a register before the loop
    - they may remove writes to entrance_intents from the loop
    - we need to declare them as volatile to avoid this
  - The CPU may perform write operations out of order
    - We need to declare them as atomic (C11)
  - Code doesn't scale for more than 2 critical sections very well

ATOMIC MEMORY EXCHANGE INSTRUCTION

- We need a special purpose Assembly instruction
  - Read and write in a single instruction
  - No intervening access from other threads
  - Atomicity: operations are performed in single, indivisible unit
- Test and Set Operation
  - Change a register, but keep track of old value (usually in a condition code/flag)
- Atomic Memory Exchange
  - Group a load and store together atomically
  - Exchange value of register and memory location

SPINLOCK

- Spinlock: lock where waiter ***spins***, looping on memory read until lock is acquired
  - Also called a ***busy-waiting*** lock
- Implementation using atomic exchange
  - Attempt to acquire lock
  - Simultaneously read old value
  - Lock acquired when old value is free
- Problem: exchange operation is expensive

SPINLOCK OPTIONS

```
_taslock:
    xorq %rax, %rax
loop:
    lock
    btsq %rax, (%rdi)
    jc   loop
    ret
```

```
_caslock:
    movq    $1, %rsi
loop:
    xorq    %rax, %rax
    lock
    cmpxchg %rsi, (%rdi)
    jne     loop
    ret
```

PROBLEMS WITH ATOMIC EXCHANGE

- Cannot be implemented in CPU alone
  - Must synchronize across multiple CPUs
  - Multiple cores accessing same location at the same time
- Typically implemented by memory bus
  - Memory bus synchronizes every CPU's access to memory
  - Bus couples both parts of exchange (read and write)
  - Bus ensures no other transaction intervenes
  - Higher overhead, slower than normal read and write

ARE SPINLOCKS A GOOD IDEA?

- Spinlocks are necessary and ok if spinner only waits for a short time
- Spinlocks waste CPU cycles (polling)
  - CPU is busy waiting for a thread that may take a long time to run
  - If running in a single CPU, other threads are not able to progress
- Alternative: block threads when they cannot run
  - A thread waiting for an event should ***block*** so that other threads may run
  - It should be ***unblocked*** when the event happens

BLOCKING LOCKS

- Blocking locks for mutual exclusion
  - Attempting to acquire a held lock blocks calling thread
  - Blocked thread's TCB stored in lock's ***waiting queue***
  - Releasing a lock unblocks first thread in waiting queue
    - Removes it from waiting queue, adds it to ready queue
- Blocking locks for event notification
  - Wait for event by blocking (waiting queue for event)
  - Event signal unblocks

SPINLOCKS VS BLOCKING LOCKS

- Spinlocks:
  - Uncontended lock has low overhead
  - Waiting for lock has high overhead and wastes resources
- Blocking locks:
  - Lock has fixed overhead
  - Higher than spinlocks' uncontended lock, but less wait if contended

WHICH LOCK SHOULD I USE?

- Use blocking locks when:
  - Lock may be held for a long time
  - High contention expected
- Use spinlocks when:
  - Small critical section, or short wait for event
  - Minimal contention expected
  - To implement blocking locks

MONITORS AND CONDITION VARIABLES

- Monitors and condition variables:
  - basis for synchronization primitives in Unix, Java, etc.
- Monitors (mutex) provide mutual exclusion
  - Blocking lock to guarantee mutual exclusion
  - Basic operations: ***lock*** and ***unlock***
- Condition variables provide inter-thread synchronization
  - Threads can synchronize events with each other
  - ***wait***: blocks until another thread signals
  - ***signal***: unblocks a thread currently waiting, if one exists
  - ***broadcast***: unblocks all threads currently waiting
  - Associated to a mutex, must have mutex lock held

MUTEX AND CONDITION VARIABLES

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&lock, NULL);
pthread_mutex_lock(&lock);          /* Acquire */
pthread_mutex_unlock(&lock);        /* Release */

pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_cond_init(&cv, NULL);
pthread_cond_wait(&cv, &mutex);
pthread_cond_signal(&cv);
pthread_cond_broadcast(&cv);
```

USING CONDITION VARIABLES: WAIT

- Condition variables are often used to signal that a desired state has been reached
  - One thread checks for condition, and waits if it is not reached
  - Another thread establishes the condition and signals the waiter
- Wait operation
  - Must only be called if mutex is held (why?)
  - Releases the mutex before blocking
  - Reaquires the mutex when unblocked (waits if necessary)
  - Note: other threads may have acquired mutex in mean time

USING CONDITION VARIABLES: SIGNAL

- Signal operation
  - Waiter does not run until signaller releases the mutex explicitly
  - Waiter must recheck wait condition (why?)
  - If no threads are waiting, then calling signal has no effect
- Broadcast operation
  - Wakes up all threads waiting for condition
  - May wake up too many threads: ok since threads recheck condition

CONSUMER WITH CONDITION VARIABLES

```
void consumer() {
  while (1) {
    uthread_mutex_lock(mx);
    while (buffer_is_empty())
      uthread_cond_wait(has_items);
    item = dequeue();
    uthread_cond_signal(has_space);
    uthread_mutex_unlock(mx);
    consume_item(item);
  }
}
```

PRODUCER WITH CONDITION VARIABLES

```
void producer() {
  while (1) {
    item = create_item();
    uthread_mutex_lock(mx);
    while (buffer_is_full())
      uthread_cond_wait(has_space);
    enqueue(item);
    uthread_cond_signal(has_items);
    uthread_mutex_unlock(mx);
  }
}
```

EVENT ORDERING EXERCISE

- Assume two threads running concurrently
  - Thread 0 calls procedure a
  - Thread 1 calls procedure b
- We need to ensure that b is not called until a returns
  - How?

WAIT-SIGNAL RACE

- The problem:
  - Wait condition check/trigger and CV wait are not atomic
  - Signal could occur before wait
  - Waiter could miss signal and wait forever
- The solution:
  - Ensure that condition check/trigger and wait are atomic
  - Particularly, signal cannot happen between check/trigger and wait
  - Waiter code is not atomic if signal isn't inside monitor
  - Mutual exclusion: locks for both wait and signal

NAKED NOTIFY

- Naked signal/notify:
  - A signal called outside of a monitor
  - Should be avoided (can cause wait-signal race)
- It's sometimes necessary
  - e.g., when blocking is not allowed

READER-WRITER MONITORS

- Critical sections could be classified as:
  - Readers: only read the shared data, do not modify it
  - Writers: update the shared data
- We could then weaken mutual exclusion constraint
  - Writers require exclusive access
  - Multiple readers can access monitor concurrently
- Reader-Writer Monitors
  - Can be free, held for reading or held for writing
  - If held for reading, multiple readers can access simultaneously

READER-WRITER MONITOR OPERATIONS

- mutex_lock(): lock for writing
  - Only acquires lock if it is free
  - Sets state to ***held for writing***
- mutex_lock_read_only():
  - If lock is free, set its state to ***held for reading***
  - Increments a ***reader count*** (or set)
- mutex_unlock():
  - If held for writing, set state to free
  - If held for reading: decrement reader count
    - Set state to free if count is zero

FAIR ACCESS TO READER-WRITER LOCK

- Policy question
  - If monitor state is held for reading
  - Writer thread attempts lock, blocks waiting for release
  - Are new readers accepted?
- Disallowing new readers while writer is waiting
  - Affects a thread that could be running and isn't
  - Provides fair access to monitor (writer has been waiting longer)
- Allowing new readers while writer is waiting
  - Increases concurrency, allows more threads to run
  - Writer may need to wait for a long time to get access (starvation)
- Solution: may depend on application

SEMAPHORES

- Introduced by Edsger Dijkstra (THE System, ~1968)
- A semaphore is a non-negative atomic counter
  - Attempts to make counter negative block calling thread
  - No operation to read value, only to change it
- P(s) (or wait):
  - Dutch: *prober te verlagen*
  - Atomic: blocks until s>0, then decrements s
- V(s) (or signal, or post):
  - Dutch: *verhogen*
  - Atomic: increase s and unblocks waiting threads if necessary

SEMAPHORES IN C

- The semaphore library provides the following functions:

```
sem_t sem;
sem_init(&sem, shared, initial_value);
sem_wait(&sem);
sem_post(&sem);
```

MUTUAL EXCLUSION USING SEMAPHORES

- Implementing a mutex using semaphores:
  - Create semaphore with initial value 1 (free)
  - lock is P()/wait()
  - unlock is V()/post()
- Implementing condition variables using semaphores
  - Not as easy as it looks
    - In condition variables, signals with no wait have no effect
    - In semaphores, signals can unlock a future wait
  - Replacing one for the other requires revising code
  - Further reading: *Andrew D. Birrell. "Implementing Condition Variables with Semaphores", 2003.*

ORDERING TWO THREADS

- If thread B must wait for thread A to finish:
  - Initialize semaphore to 0
  - Thread A: `sem_post(&b);`
  - Thread B: `sem_wait(&b);`
- If both threads need to wait for each other (barrier):
  - Initialize two semaphores with 0
  - Thread A:

```
sem_post(a);
sem_wait(b);
```

  - Thread B:

```
sem_post(b);
sem_wait(a);
```

THE DINING PHILOSOPHERS PROBLEM

- Formulated by Dijkstra around 1965
  - As an exam problem
- Problem:
  - 5 philosophers sit at a round table with forks in between each pair
  - If they are thinking, they do nothing
  - If they want to eat, they grab 2 adjacent forks and eat
  - If another philosopher has fork, wait

DINING PHILOSOPHERS: DEADLOCK

- Assume philosophers always start with left fork
- Assume all philosophers decide to start at the same time
  - All philosophers are able to get the left fork
  - All philosophers must wait for the right fork
  - DEADLOCK!!!

DINING PHILOSOPHERS: LIVELOCK

- Assume that, if philosophers can't get the second fork, they release the first fork, then wait on the second
  - If all of them do it at the same time, they will now hold the right fork
  - But none can proceed because they can't get the left
- If the process is repeated, and all philosophers are synchronized
  - Philosophers will repeatedly get one fork at a time
  - All are busy, but they are unable to eat
  - LIVELOCK!!!

PROBLEMS WITH CONCURRENCY

- Race condition
  - Competing, unsynchronized access to shared variable
  - From multiple threads
  - At least one thread is typically updating the variable
  - Solved with synchronization
  - Language does not always help, can be hard

PROBLEMS WITH CONCURRENCY

- Deadlock
  - Multiple competing actions wait for each other
  - All actions are prevented from completing

```
void foo() {
  lock1.lock();
  lock2.lock();
  // …
  lock2.unlock();
  lock1.unlock();
}
```

```
void bar() {
  lock2.lock();
  lock1.lock();
  // …
  lock1.unlock();
  lock2.unlock();
}
```

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

DEADLOCK

OTHER PROBLEMS WITH CONCURRENCY

- Livelock
  - Threads respond to actions by other threads, but other threads also respond
  - Threads are unable to make progress

```
while (true) {
  lock1.lock();
  if (!lock2.tryLock()) {
    lock1.unlock();
    lock2.lock();
    if (!lock1.tryLock()) {
      lock2.unlock();
      continue;
    }
  }
  break;
}
// …
```

OTHER PROBLEMS WITH CONCURRENCY

- Starvation
  - A thread is unable to gain regular access to a shared resource
  - "Greedy" threads may lock resource for long time
  - Threads with higher priority may skip ahead
  - Reader threads may acquire reader-writer lock in front of writer

PROBLEMS WITH RECURSION

- What's the problem with the following code?

```
void foo (int n) {
  uthread_mutex_lock (mx);
  count -= n;
  if (count >= n)
    foo(n-1);
  uthread_mutex_unlock (mx);
}
```

PROBLEMS WITH RECURSION

- If lock is acquired again in recursive call
  - Lock is held, so thread blocks
  - Thread is waiting for lock to be released
  - But same thread is actually holding the lock
- If we release lock before calling recursively
  - Could break critical section's protection

SOLUTION: REENTRANT MUTEX

- Reentrant mutex: allows lock to be acquired more than once
  - Only if it is acquired again ***by the same thread***
- Unlock only releases the lock if called as many times as lock was called
- Each lock operation increments counter
  - Unlock decrements it, and releases lock when counter is zero

AVOIDING DEADLOCKS

- If you don't need them, don't use multiple threads
- If you don't need them, don't use shared variables
  - Use local variables and parameters whenever possible
- Avoid unnecessary locks
  - When possible, use atomic data structures and lock-free synchronization
  - Be careful with livelock situations
- Evaluate scope of lock
  - Could deadlock be avoided by reducing portion of code that needs lock?
- If possible, use only one lock at a time
  - Deadlock only happens if thread holding a lock needs to wait

AVOIDING DEADLOCKS (CONT.)

- Organize locks into precedence hierarchy
  - Locks must always be acquired in same order
- Limit wait time on locks
  - Provide an alternative action if lock cannot be acquired
- Detect and destroy
  - If possible, identify when a deadlock has occurred (or is about to occur)
  - Break deadlock by interrupting threads

HARD MODULARITY

- Modularity between threads is soft
  - They share the same memory
  - Which means one thread can affect the others
- How can we make it harder?
  - We could run them on different machines
  - Or simply make sure they don't share memory
    - We will see how when we discuss virtual memory

OPERATING SYSTEMS

- Modern computers handle multiple processes simultaneously
  - Kernel does not trust processes
  - Processes do not trust each other
- Operating systems aim to:
  - Provide a single-system illusion for each process
  - Coordinate sharing and isolation
  - Provide primitives for transparent access to resources

ABSTRACTIONS

- Some resources are provided as abstractions
  - File systems
  - Processes, threads, synchronization
  - Communication
  - Authentication
- Shared resources abstract isolated access
  - Processor, Memory, I/O access
  - Access to resources is limited
  - Interaction between processes is limited
  - Modularity and security are enforced

OPERATING SYSTEM ABSTRACTIONS

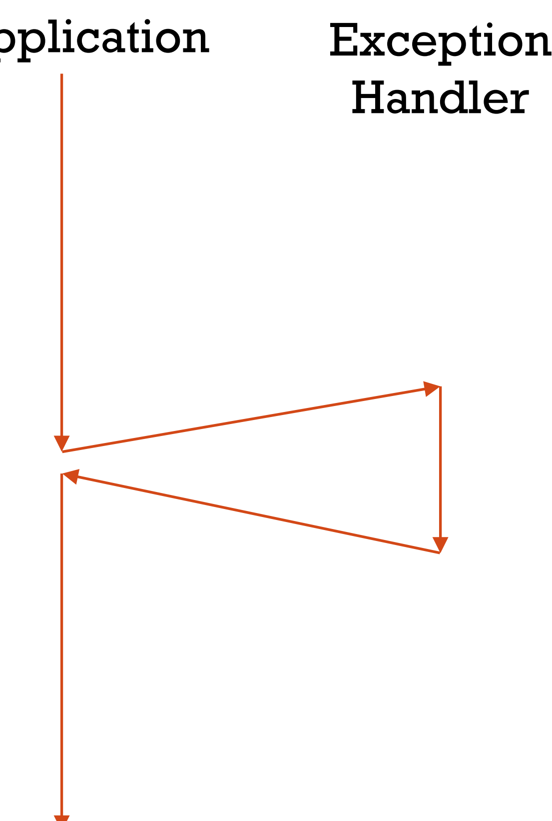
- OS memory is protected mostly using virtual memory
- Hardware operates in two modes: *user* and *kernel*
  - Regular processes operate in user mode
  - Kernel mode required for privileged operations/instructions
  - Additional modes may be used in contexts like virtualization
- Some instructions are illegal when in user mode

KERNEL MODE

- What you can do only in Kernel mode
  - Change value of special registers
    - Page Table Base, Exception Table Base
  - Change behaviour of interrupts
  - Access memory using physical address
  - Halt the processor
  - Talk to I/O devices
    - I/O addresses are mapped to memory that only kernel can access

EXCEPTIONS

- Exceptional Control Flow implemented partly by hardware, partly by OS
- Change in control flow in response to change in CPU state
  - E.g.: I/O requests, invalid instruction
  - Causes CPU to move to kernel mode



CLASSES OF EXCEPTIONS

- Interrupt
  - Asynchronous, caused by signal from I/O device
- Trap
  - Caused by specific instruction (e.g., system call)
- Fault
  - Caused by instruction with potentially recoverable error
  - e.g., access to invalid memory
  - Same instruction is repeated, if possible
- Abort
  - Caused by non-recoverable error
  - Control does not return to application

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: CRITICAL_PROCESS_DIED

EXCEPTION HANDLING

- Each type of possible exception is assigned an exception number
  - Hardware defined: divide by zero, memory violations, overflows
  - OS defined: system calls, I/O signals
- At boot time:
  - OS sets up Exception Table in memory
    - Jump table with addresses of OS procedures (function pointers)
  - Special register (exception table base register) has start address of exception table
- When exception happens:
  - Switch to kernel mode
  - Indirect call to entry in exception table

KERNEL MODE TRANSFER

- Getting into Kernel mode from User mode
  - Trap instruction: `syscall`
  - Interrupt from I/O device
  - Fault (illegal memory access, divide by zero, etc.)
- Switching back to User Mode
  - Instruction `iret`

THE OPERATING SYSTEM KERNEL

- Kernel: Collection of modules running in kernel (supervisor) mode
- Ideally it must be small
  - A programming error in kernel mode is fatal
  - Microkernel: any non-essential services removed from kernel
  - Disadvantage: slow (too much overhead)

PROCESSES

- Instance of program in execution
  - Independent logical control flow
  - Illusion that process has exclusive use of CPU
- Program runs in a context (state):
  - Code and data stored in memory
  - Contents of registers, status registers, stack pointer
  - Program counter
  - OS specific information (e.g., open files, environment variables)
- Private address space
  - Will be discussed later, with virtual memory

PROCESSES VS THREADS

- Each process can have multiple threads
- Threads in the same process share:
  - Same memory space (code and variables)
  - Same resources (e.g., open files, environment variables)
- Threads in the same process have different:
  - Register values, stack pointer, condition codes
  - Flow of control (program counter)

LOGICAL CONTROL FLOW

- Abstraction: process runs as if exclusively
- Concurrency (multitasking)
  - Execution of logical flow is interleaved
  - Processes take turns using the CPU
- Periodically processes are *pre-empted*
  - Pre-emption process triggers context switch
  - Typically caused by interrupt (may be caused by other events)

CONTEXT SWITCH

- When a CPU switches between processes, the state (context) needs to be switched too
- Caused by:
  - timer (through interrupts)
  - system call block (e.g., I/O wait, sleep)
- Context switch:
  - switch to kernel mode
  - save the context of current process
  - restore the context of scheduled process
  - set PC to scheduled process in user mode

PROCESS CONTROL: FORK

- Fork is a system call used in Unix-based systems to create a new process
- After a call to fork:
  - The current process continues running as usual
  - A new process with same state is created
  - All process memory is “copied” into new process
- Returning value of fork distinguishes “parent” process from “child” process
  - 0 (zero): current process is the child process
  - positive: current process is the parent process
    - value corresponds to process ID of the child
  - -1: fork failed, new process was not created

EXAMPLE:

```
switch(fork()) {
case -1:
    perror("fork");  /* something went wrong */
case 0:
    printf("This is the child process!\n");
    /* … */
default:
    printf("This is the parent process!\n");
    /* … */
    wait(NULL);
}
```

SIGNALS

- Signal: message that notifies a process that an event has occurred
- Kernel notifies process of specific event:
  - System event detected by kernel (e.g., fault)
  - Another process requested a signal being sent (system call "kill")
- Process can either:
  - Terminate
  - Handle it through a pre-determined handler function
  - Ignore the signal

SIGNAL EXAMPLES

- Ctrl-C in terminal:
  - Triggers SIGINT (by default terminates process)
- Kill command:
  - Sends a signal to process (by default SIGTERM)
- I/O ready:
  - Triggers SIGIO (allows process to proceed with read/write)
- Child process terminates (fork):
  - Triggers SIGCHLD

PIPES

- Pipes are a unidirectional method of communication between processes
- A pipe contains two file descriptors:
  - Data written to side 1 is read from side 0
- Example: calling a process with "|":
  - ls -l | wc
- fifo: named pipe
- socketpair: similar, but bidirectional

MEMORY MAPPED FILES

- An open file can be mapped to a region in memory
  - Reads from the region correspond to data in the file
  - Writes to the region are written back to the file
- PTE points directly to file itself
- Can be shared between different processes

MEMORY SHARING

- A shared memory object is similar to a file, but saved in memory
- Memory mapping allows multiple processes to access same object

VIRTUAL MEMORY

Unit 7

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

OUTLINE

- Motivation: Isolation, protection and sharing
- Address spaces, paging and memory translation
- Caching
- Demand paging and optimization

VIRTUAL MEMORY: MOTIVATION

- Problem 1: memory space
  - Each process behaves as if it's the only one using memory
  - Each process assumes there is, say, 2^{32} bytes of memory
  - There are many processes running concurrently
  - Physical memory (RAM) is limited

VIRTUAL MEMORY: MOTIVATION

- Problem 2: isolation
  - Programs have bugs, pointers can point to wrong memory address
    - Unintentional (e.g., bugs)
    - Intentional (e.g., Trojan horse)
  - One process could access memory that belongs to another process

VIRTUAL MEMORY: MOTIVATION

- Problem 3: sharing
  - Processes may want to share memory
  - Example: Programming libraries
    - Code should not be copied to all processes
  - Sometimes there are restrictions (e.g., read-only)

CPSC 261 2019 WT2 © 2020 Jonatan Schroeder

VIRTUAL MEMORY

- Memory translation
  - Program accesses instructions and data using *virtual address*
  - MMU (Memory Management Unit) translates it into *physical address*
  - Physical address is used to access memory
- MMU must be in hardware
  - Why?

ADDRESS SPACE

- Address space: collection of valid addresses
  - Usually linear: consecutive integers from 0 to $2^n - 1$.
  - Segmented space (early Intel CPUs)
    - Each address consisted of two parts
- Virtual address space: used by the user process
- Physical address space: address used by actual physical memory

VIRTUAL MEMORY

- Virtual and physical addresses are divided into pages
  - Pages have fixed size
  - Each virtual page can be linked to one physical page (page table)
- Translation is performed by MMU
- Each process has its own page table

VIRTUAL MEMORY

- Sharing problem:
  - Virtual addresses in two processes may point to same physical address
  - Page table entry may mark page as read-only
- Isolation problem:
  - Same virtual address in different processes leads to different physical addresses
- Memory space problem:
  - Unused virtual addresses do not point to an actual physical address
  - Some pages may be stored in disk
    - Usually separate partition (swap)

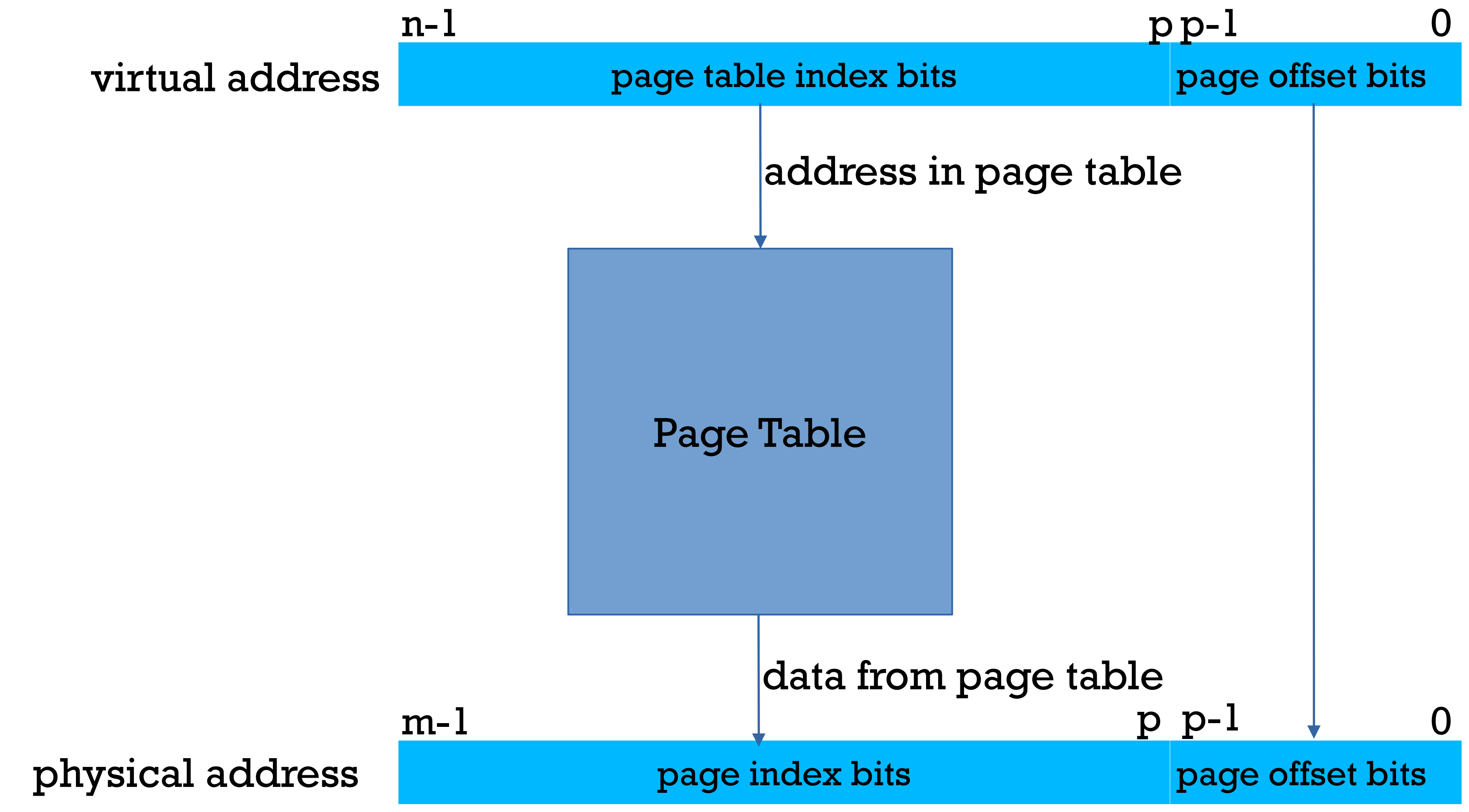
PAGE TABLE

- Page table is stored in main memory
  - Address is only accessible by the kernel
  - Most frequently used blocks end up in cache
- Each process has separate page table
- Base address stored in special register: PTBR (Page Table Base Register)
  - In x86: CR3
  - Value of register changed during context switch

PAGE TABLE

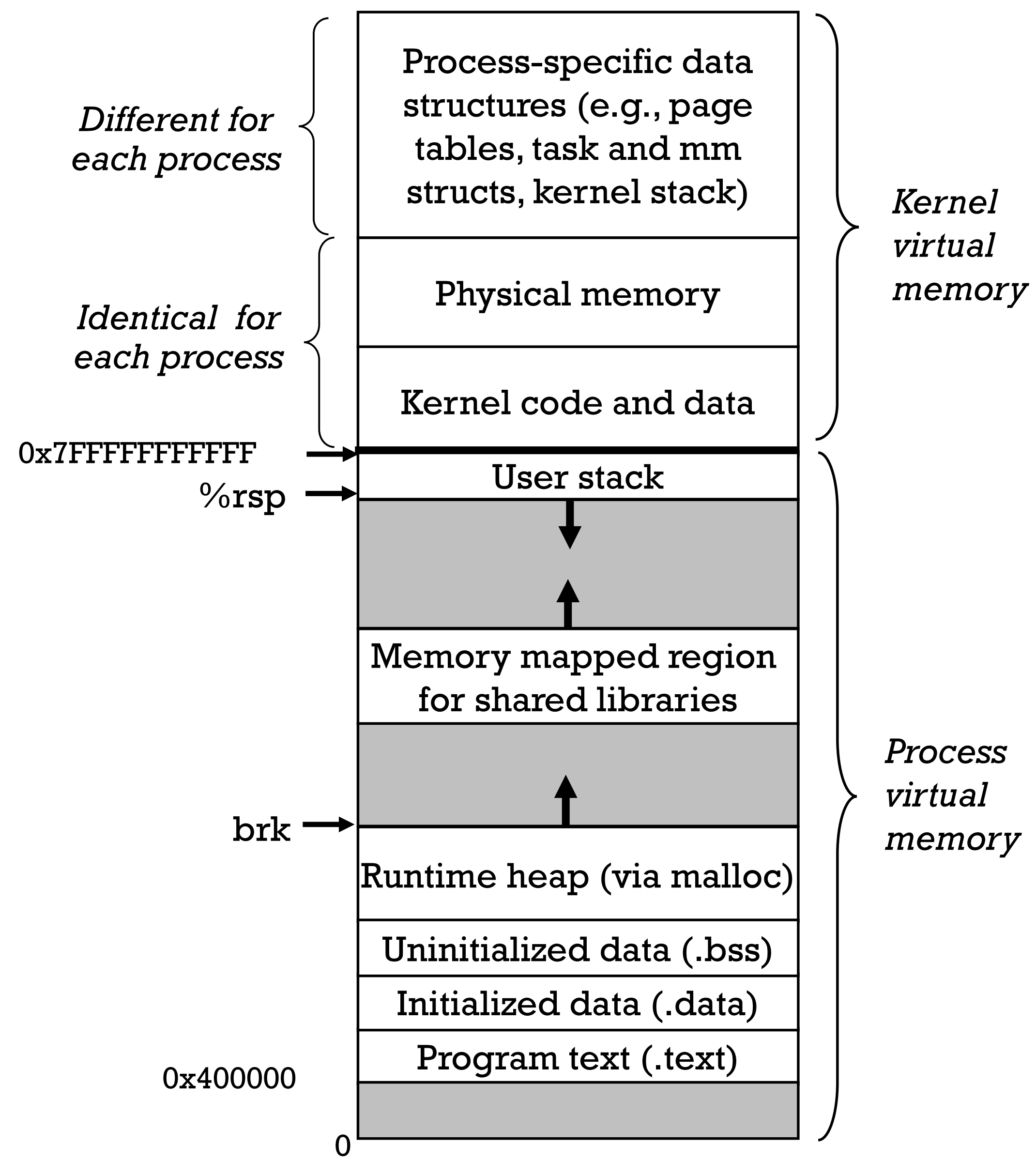
- Virtual and physical addresses are both divided into pages of 2^p addresses each
  - Least significant p bits determine the offset within the page
  - Remaining bits of virtual address (virtual page number) are used as index into a page table
  - Value stored at the index determines the physical page number
- A virtual address is converted by:
  - Extracting the virtual page number and offset
  - Finding the corresponding entry in page table
  - Combining physical page number with offset

VIRTUAL MEMORY TRANSLATION



MEMORY SPACE

- Linux processes use convention for memory address use
- OBS: Addresses here are listed from bottom to top



MEMORY MAPPING FOR PROCESSES

- Each process will map regions into different areas (mappings)
  - Text region (code): backed by executable in disk
  - Read-only data (e.g., strings): backed by executable in disk
  - Global variables: initially fault on read
    - when read, exception handler copies from executable
  - Heap, swap: initially fault on read
    - when read, exception handler copies from zeroed page
    - global variables initialized to zero may use this approach
  - Shared libraries: backed by library files
    - OS will try to share with other processes

PAGE TABLE ENTRY

- Cached page: page is currently in physical memory
- Uncached (non-resident) page: page is on disk, but not in memory
- Unallocated page: page does not exist yet
  - Memory space that has not been used or initialized yet

PAGE TABLE ENTRY

- Each entry of the page table contains (may vary per architecture):
  - SUP bit: if set, can only be accessed in kernel mode
    - example: memory used to access devices
  - READ bit: if set, process can read the page
  - WRITE bit: if set, process can write to the page
  - EXECUTE bit: if set, address can be used as instruction
  - INMEM bit: if set, page is stored in physical memory
    - if not set, page may be stored in disk, or not stored at all
  - The physical page number (address)

VIRTUAL MEMORY FOR PROTECTION

- Process can only access physical memory mapped by its page table
  - OS controls the page table, so OS controls access
- What about kernel memory for a process?
  - Included in the page table
  - Kernel acts as a library
  - Kernel pages have SUP bit set

PAGE FAULTS

- MMU may detect a problem (page fault):
  - Accessing a page process is not allowed to access
  - Reading/writing to page with no read/write permission
  - Accessing SUP page in user mode
  - Accessing unallocated page
  - Accessing uncached page
- When MMU detects a page fault
  - Fault triggers kernel mode
  - Kernel may fix problem, or abort

PAGE FAULTS

- Accessing a page process is not allowed to access
  - Usually a programming error
  - Example: NULL pointer
  - Operating system will terminate process, possible save "core" (copy of memory contents) and issue a warning

PAGE FAULTS

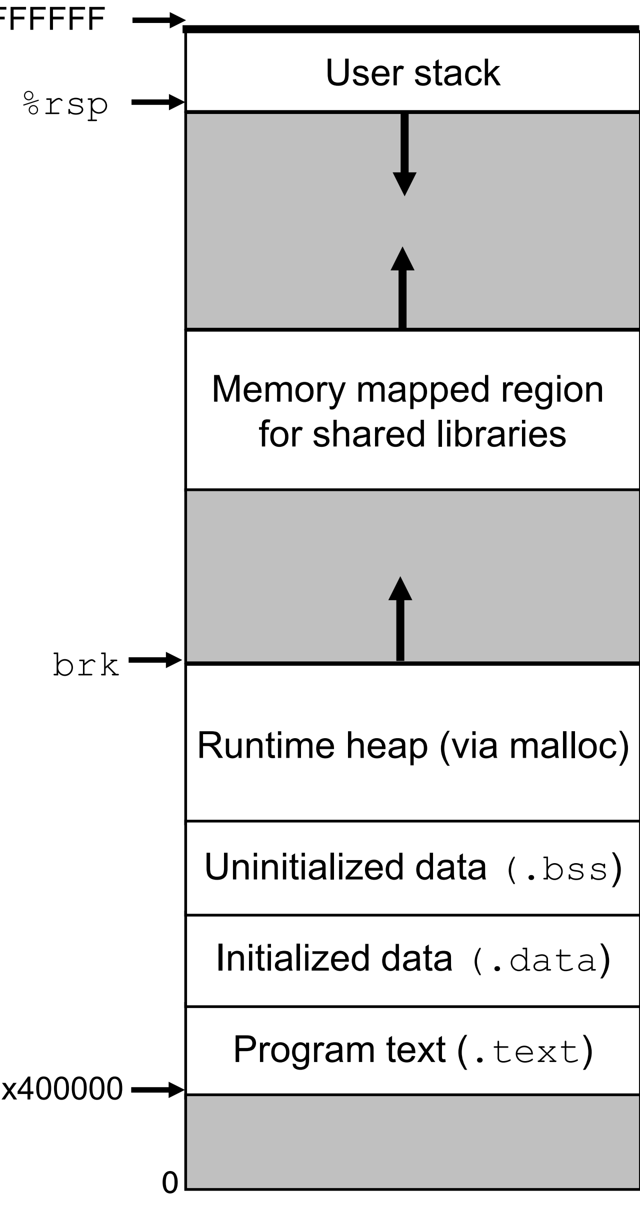
- Accessing a page whose read or write bit is unset
  - Process may only be allowed to read
    - shared library, instruction memory
    - OS terminates process
  - Page may have not been updated yet
    - OS may delay writing to a page until it is accessed
    - Pages may block until I/O provides its data
  - Page has never been accessed yet
    - Call to malloc will create page, but not allocate memory space

PAGE FAULTS

- Accessing an uncached page
  - Page needs to be retrieved from disk
  - Physical memory is treated as fully-associative cache of disk
    - More sophisticated cache policy
    - Write-back policy (write evicted page to disk)
  - Process may be suspended while page fault is handled
- If too many pages need to be retrieved from disk
  - Very little work is accomplished
  - Thrashing
  - Typically happens when working set is bigger than RAM

MEMORY SPACE

- Virtual memory just below %rsp is marked as unallocated
  - When accessed, allocation is automatic
- Virtual memory above heap limit is marked as invalid
  - Segmentation fault on access
- System call brk changes size of heap
  - Library functions brk/sbrk



MEMORY AS CACHE FOR DISK

- Which page is evicted?
  - Penalty for bad replacement policy is high
  - True LRU is too expensive to be done in hardware
- Optimal: Béládу's algorithm (aka Clairvoyent's algorithm)
  - Furthest in the future
  - Impossible in practice
- LRU is not an option
  - There are too many pages
  - LRU variations will require too much computation and/or too much space

PAGE REPLACEMENT POLICY

- NRU (Not Recently Used) policy:
  - Keep two bits per page: one for read, one for write
    - Set them every time there is an access
    - Clear them (or decrement counter) periodically
  - Choose pages to evict based on recent access:
    - First, pages not modified or accessed
    - Then, pages modified but not accessed
    - Then, pages accessed but not modified
    - Then, pages accessed and modified

PAGE REPLACEMENT POLICY

- Clock replacement policy
  - On access, set reference bit
  - For each page in round-robin:
    - If current bit has reference bit set, clear bit
    - Otherwise, evict page

TIME OPTIMIZATION

- As is, with virtual memory, every memory access in CPU requires two reads from physical memory
  - If page table is in L1 cache, impact is reduced, but this affects locality of actual data
- How can we make it faster?
  - A dedicated cache

TRANSLATION LOOKASIDE BUFFER (TLB)

- MMU uses a small, high-associativity cache only for virtual to physical address translation
- TLB structure:
  - Each entry in TLB corresponds to one page
  - Virtual page number is composed of set index and tag (similar to L1 cache)
  - TLB entry has same data as page table entry (e.g., read/write/SUP bits, physical address)

VIRTUAL MEMORY EXAMPLE

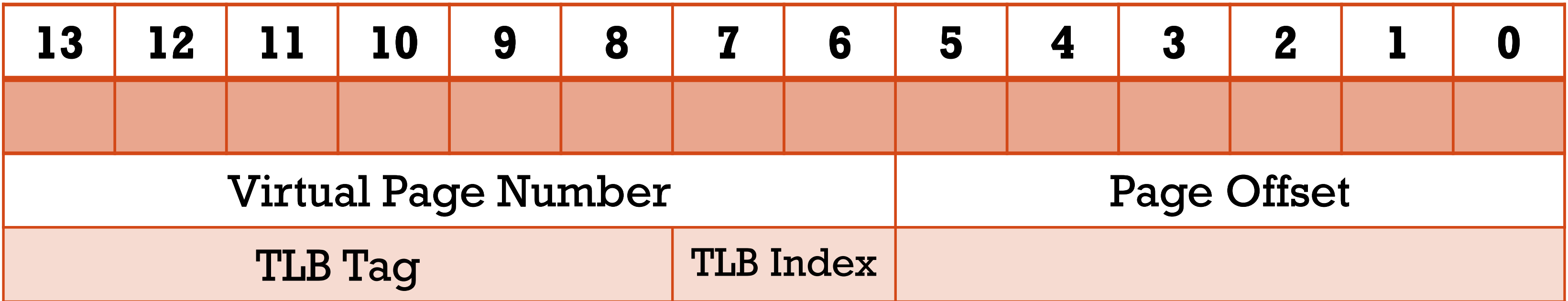
- Consider a memory system where:
  - Memory is byte addressable
  - Virtual addresses are 14-bits wide
  - Physical addresses are 12-bits wide
  - Page size is 64 bytes
  - TLB is 4-way set associative with 16 total entries
  - L1 d-cache is direct mapped (via physical address), with 8-byte block size and 16 lines

VIRTUAL MEMORY EXAMPLE (CONT.)

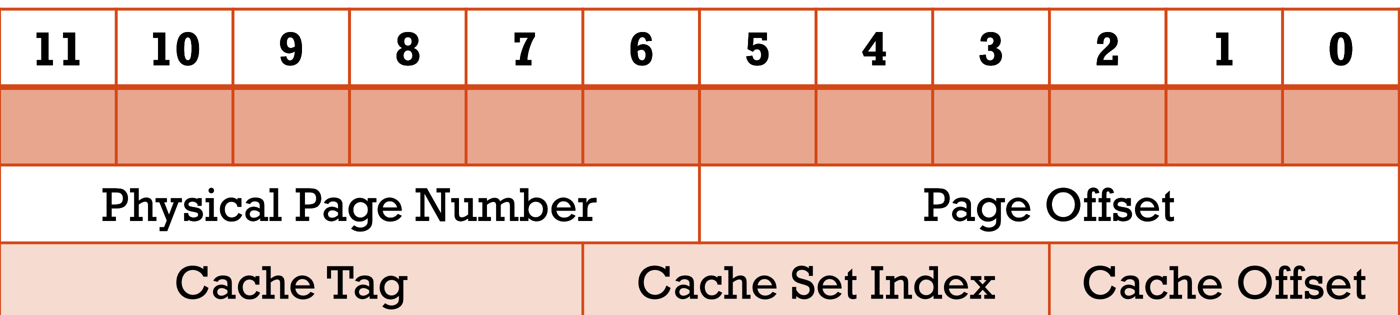
- Conclusions:
  - Page size is 64 bytes: $\log_2 64 = 6$ bits for page offset
  - Virtual page number: $14 - 6 = 8$ bits long
  - Physical page number (aka page frame number): $12 - 6 = 6$ bits long

VIRTUAL MEMORY EXAMPLE (CONT.)

- Virtual address:



- Physical address:



VIRTUAL PAGE TRANSLATION SUMMARY

- Divide Virtual Address into Virtual Page Number (VPN) and Offset
- Check TLB for VPN, if miss:
  - Compute physical address of page table entry (PTE)
    - PTE physical address = PTBR + VPN * PTE_SIZE (e.g., 8)
  - Read PTE from memory using this address
- Check PTE valid bit, throw exception if invalid
- Read Page Frame Number (PFN) from PTE to compute Physical Address (PA)
  - PA = PFN * PAGE_SIZE + Offset
- Read/Write target PA from/to memory
  - Start with L1, then L2…

SPACE OPTIMIZATION

- Imagine a scenario:
  - Virtual address: 48 bits
  - Page size: 4KB (2^{12})
  - 64-bit page table entry
- How many pages are there in total?
- How much space do you need to store the virtual page of one process?
- What if we have 200 processes running simultaneously?

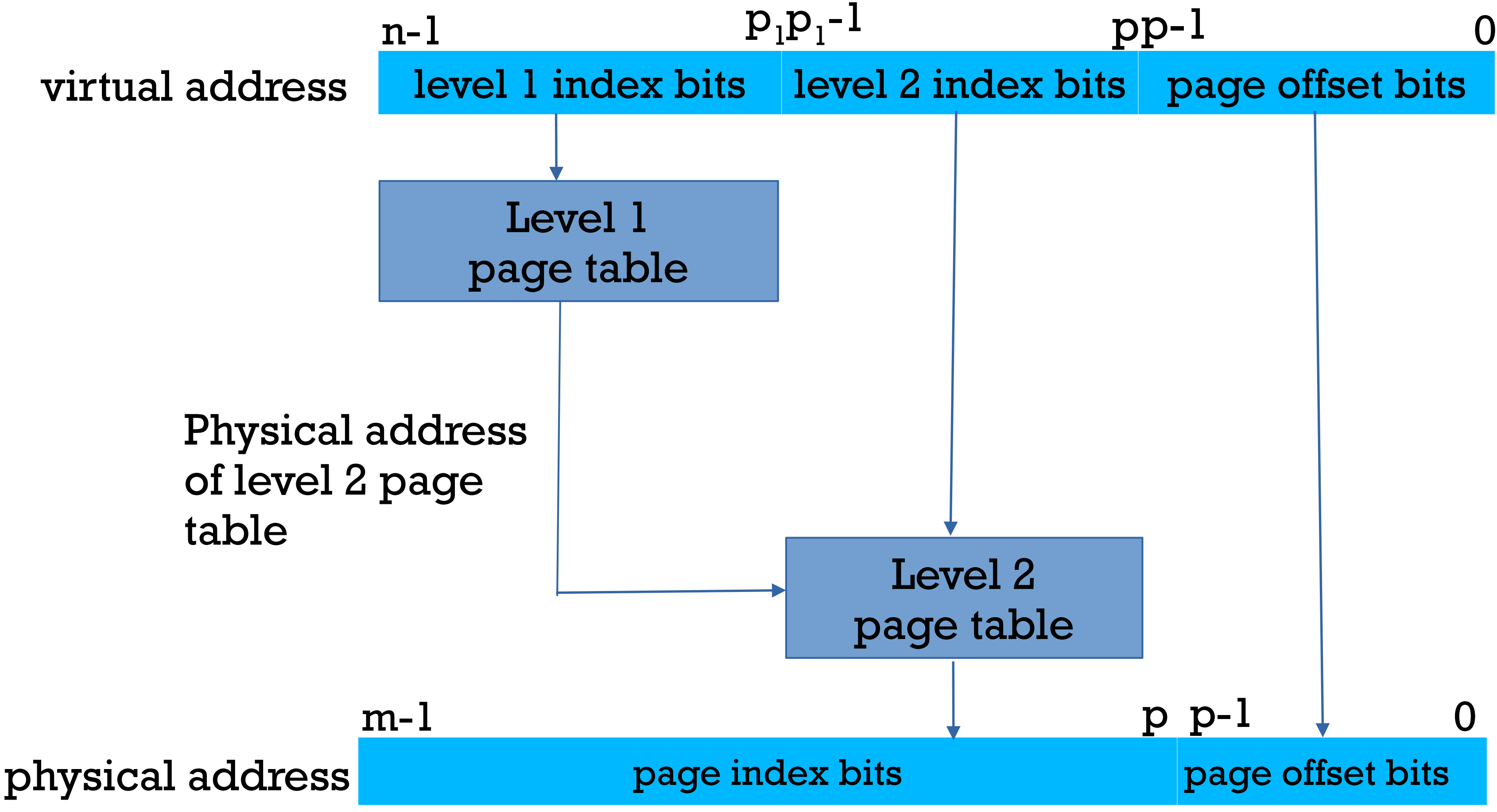
SPACE OPTIMIZATION

- What if we make pages bigger?
  - Page size: 4GB
- What is the size of each virtual page?
- What is the impact on memory usage?

SPACE OPTIMIZATION

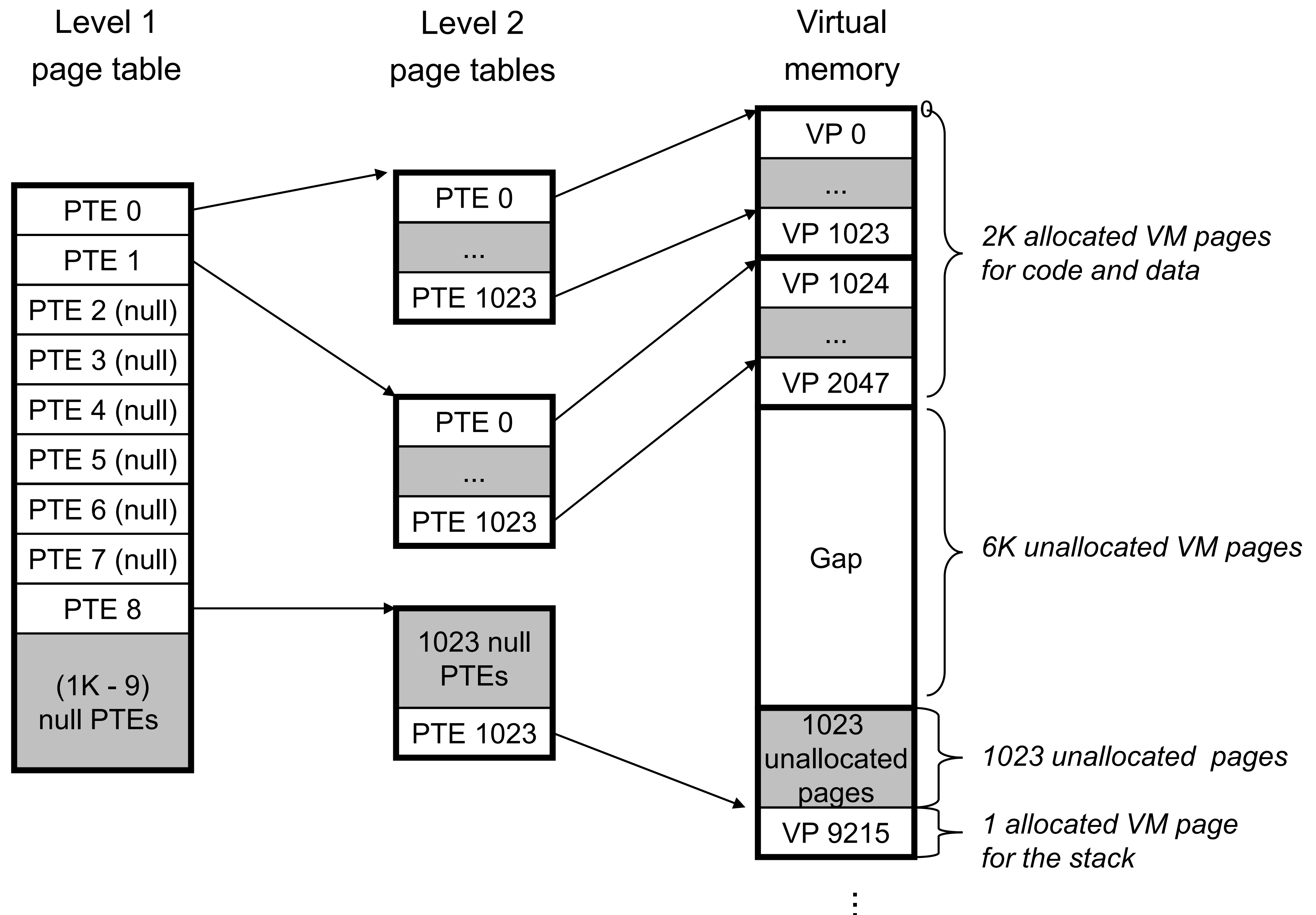
- What if page table is divided into groups?
  - Each group has, say, 1024 entries
  - Each group is stored in its own table
  - Another "master" table has pointers to individual tables
  - Groups where all addresses are unused doesn't need to exist
- This only affects the actual page table, not TLB

MULTI-LEVEL PAGE TABLES

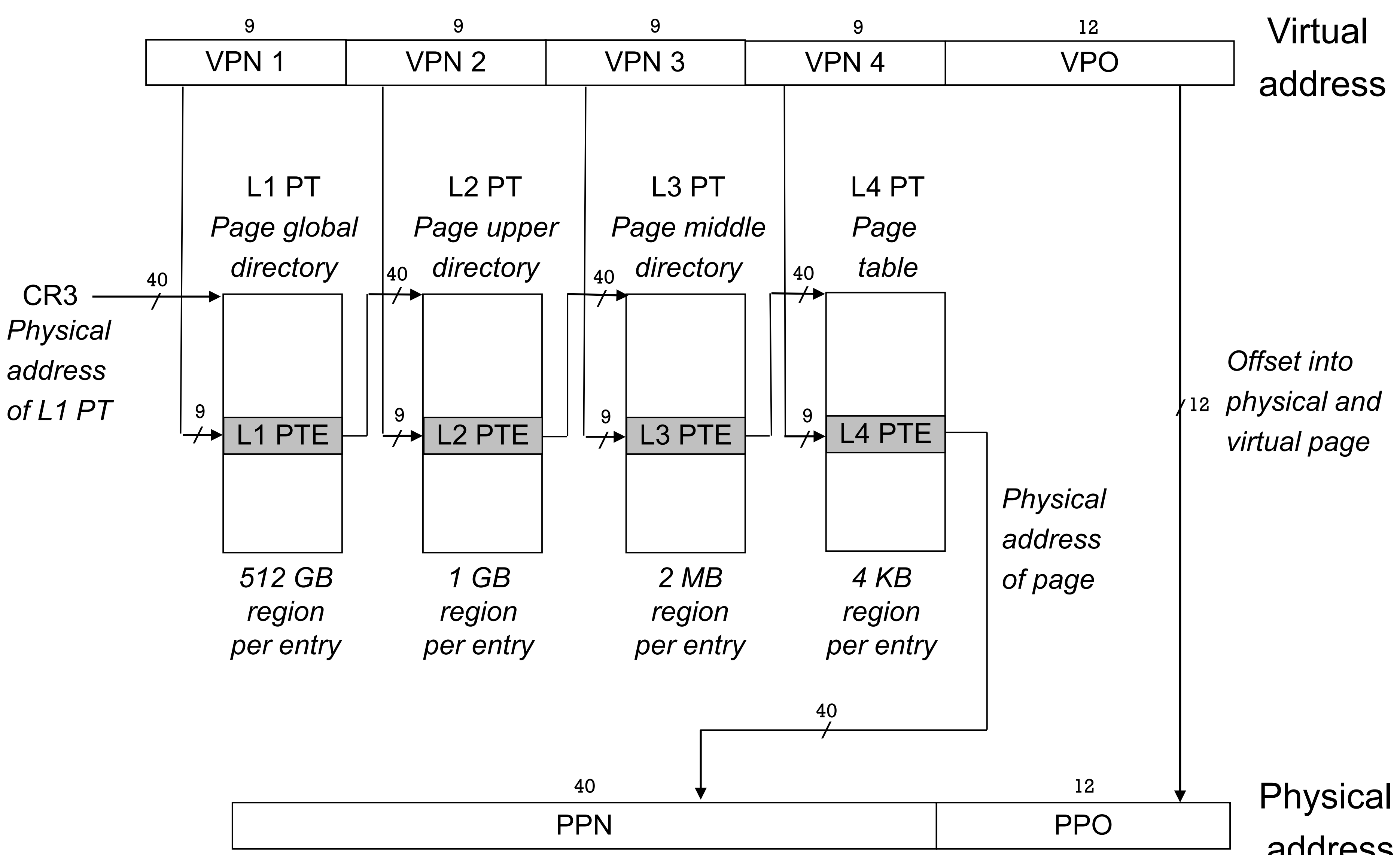


MULTI-LEVEL PAGE TABLES

- How is space saved?
  - If all page entries linked by the first level are unused, the second level table doesn't need to exist
  - If a second level table isn't used often, it can be stored in disk
- When TLB is properly used, impact of extra access time is minimized
- We are not limited to two levels


Level 1
page table
Level 2
page tables
Virtual
memory
PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8
(1K - 9)
null PTEs
PTE 0
...
PTE 1023
PTE 0
...
PTE 1023
1023 null
PTEs
PTE 1023
0
VP 0
...
VP 1023
VP 1024
...
VP 2047
Gap
1023
unallocated
pages
VP 9215
2K allocated VM pages
for code and data
6K unallocated VM pages
1023 unallocated pages
1 allocated VM page
for the stack

INTEL CORE I7 ADDRESS TRANSLATION



INTEL CORE I7 ADDRESS TRANSLATION