

Slides/lectures

Contents

Binary	3
Numbers as bits and bytes	7
Media as numbers	7
Bits and bytes in action.....	9
Floating point numbers.....	14
X86-64	15
Binary files	21
Binary exploits.....	23
Bit hacks	26
Processors	29
Digital logic	29
Combinatorial logic circuits	29
Sequential logic circuits	32
A processor Y86-64 Design (ISA).....	33
A Processor: Y86-64 Semantics (SEQ)	39
A processor: Y86-64 implementation (SEQ)	45
Stack, procedure calls.....	47
Buffer overflows	53
Parallelism	56
Notes.....	56
Reading.....	57
Pipelining	57
Heterogeneous computing.....	72
Slides.....	75
Pipelining	75
Vectorization	104
Multicore	106
Heterogeneous computing.....	106
Memory	109
Reading.....	109
Locality	109
Caching.....	112

Virtual memory	120
Slides.....	126
C	152
Reading.....	152
C-to-assembly.....	154
Slides.....	188
C	189
C to assembly.....	209
Optimizations, compiler.....	214
Optimizations manual.....	217
Threads	221
Reading.....	221
Multithreading	221
Slides.....	260
Processes.....	277
Reading.....	277
Slides.....	300
Interrupts	301
Booting.....	314
Processes.....	318
I/O.....	333
Reading.....	333
File systems	333
Slides.....	356
File systems	356
I/O API.....	363
File Systems	365
Aside.....	366
File systems	367
Takeaways: Files.....	370
I/O API.....	370
Takeaways: sockets	379
I/O API: MPI	379
Takeaways: MPI	380
Monitors	381
Reading.....	381

Operating system security.....	381
Access control.....	383
Slides.....	390
Containment	417
Introduction	417
Namespaces.....	417
Example: UTS namespaces.....	418
Commands	419
Demonstration	421
Capabilities.....	422
User namespace overview.....	423
UID and GID mappings	424
User namespaces and capabilities	426
Use cases	429
Extra	430
Heterogeneous computing.....	432
CUDA	455

Binary

Course focus: most important operatives

What's in it for you

- how computers work (*that course at ITU*) "power-programmer"
- computer literacy (shell, Linux, vim, ...)
- general knowledge (history, business, (geo-)politics)
- learn a new programming language performance | security
- experience w/ system programming
- how *computer systems* impact *software design*

want to be a software / data engineer?
want to be a great programmer?
(want to graduate...)

} you must master
this course.

What is a system:

system = set of interconnected **components** w/ well-defined **behavior** at their **interface** (to **environment**)

- modularity, abstraction, layering, hierarchy

processor, memory, keyboard, ...

hardware (HW) = physical computer components

software (SW) = instructions for hardware

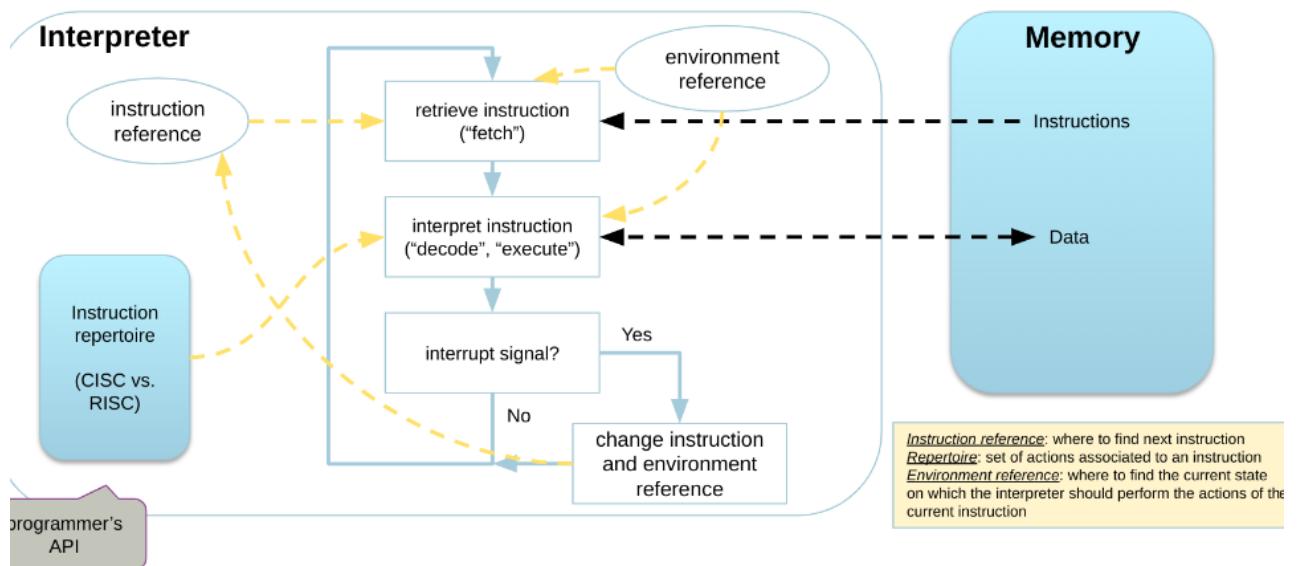
computer system = **hardware + systems software**
working together to run **apps.**

3 fundamental abstractions for computer systems:

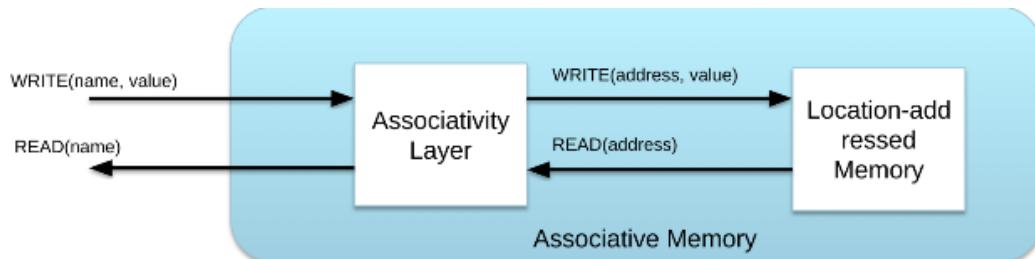
- Interpreter
- Memory
- Communication

Computer systems are complex, we control this complexity with a system.

Interpreter



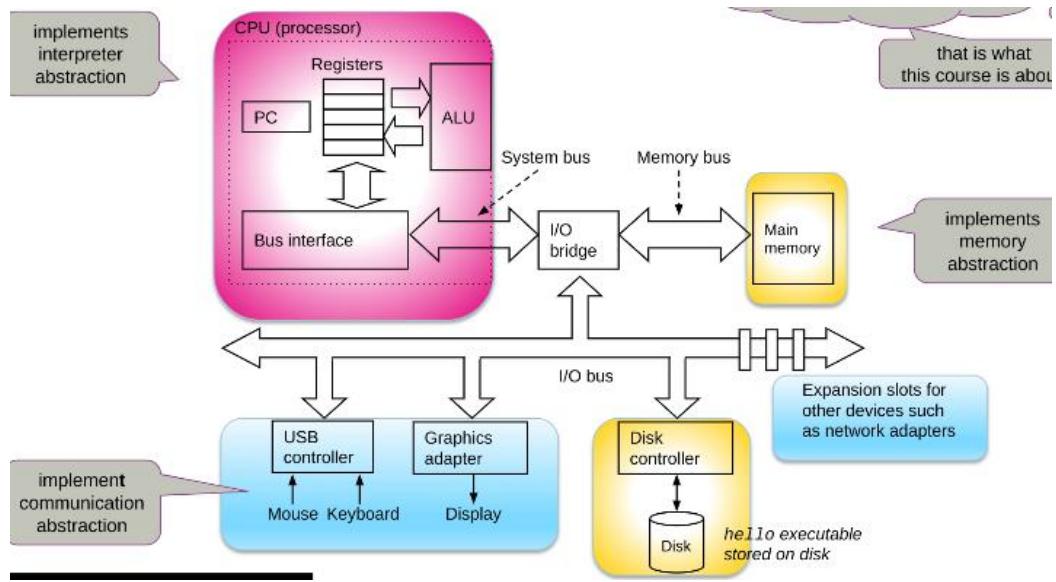
Memory abstraction - you can read and write to



Communication abstraction - receive and send data



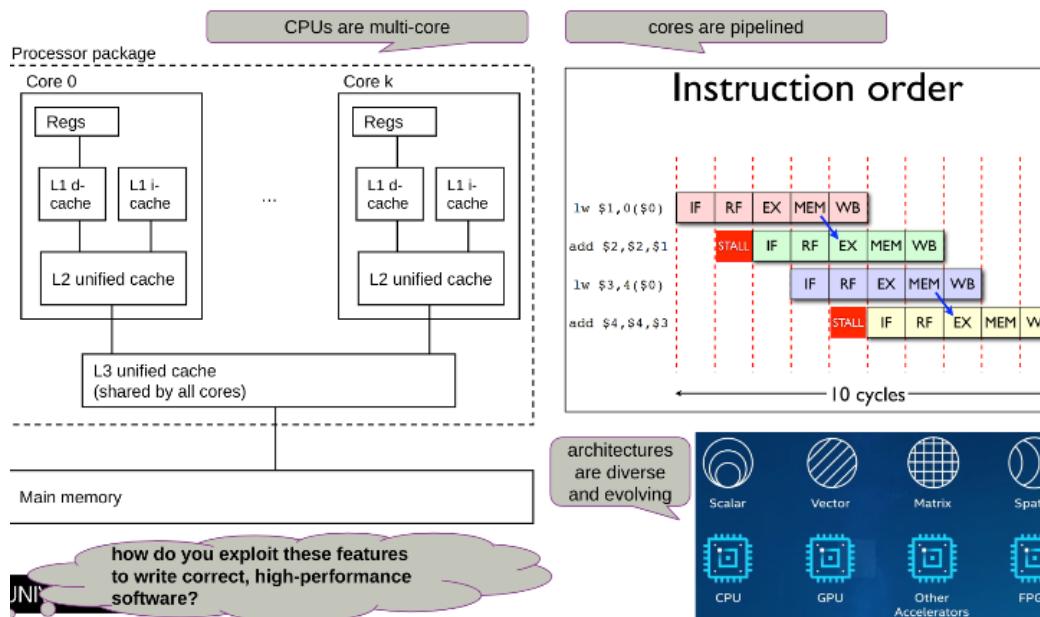
Computer hardware - the computer as a whole



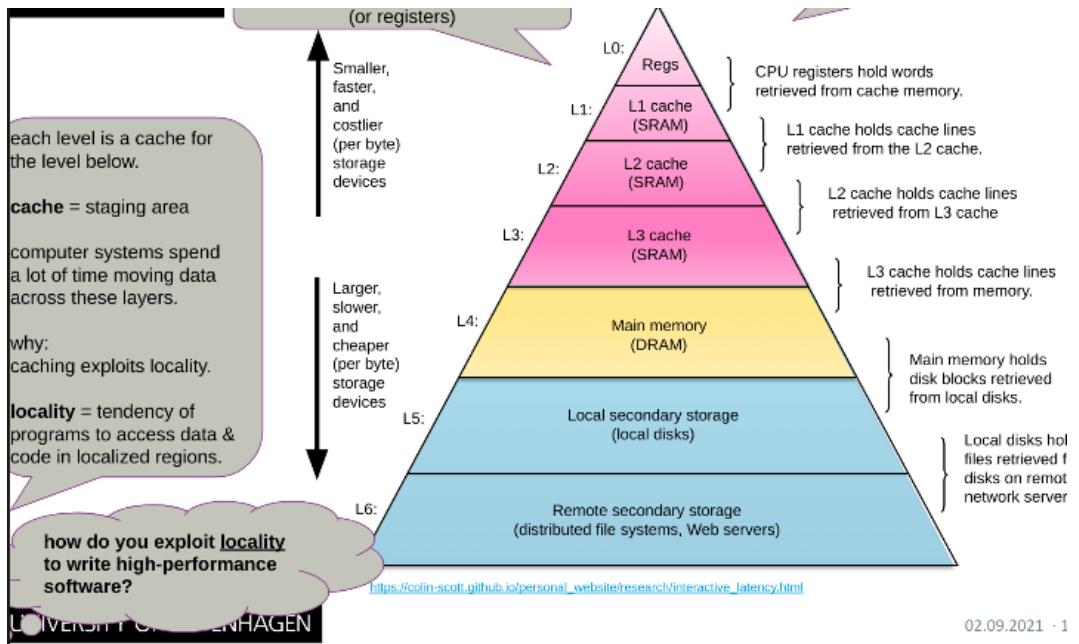
How does the system affect us as programmers?

- What the course is about

Processors -



Memory - not just one (memory mountain) - how different data can be stored away from the

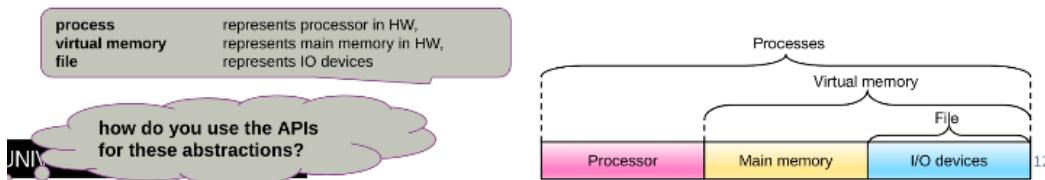


CPU - the longer away, the longer it takes to gain access

Operating system - manages computer hardware

3 fundamental abstractions for operating systems:

- Process
- Virtual Memory
- File



System programming - write programs that manage hardware

writing programs that manage hardware

- OS kernel
- embedded systems
- infrastructure software that must tightly control its use of hardware resources:
 - compilers, database systems, version control,

Programs and data

What are programs (detailed)?

How computers work?

Numbers as bits and bytes

Numbers are normally written in decimal notation (base -10)

numbers, decimal (base-10)

numbers are typically written in **decimal** notation (base-10):

10^5	10^4	10^3	10^2	10^1	10^0
3	1	4	1	0	9

i.e. a { sequence, string, vector } of *decimal digits* { 0, 1, ..., 9 }.

a *place-value* notation; 0th place contributes $9 \cdot 10^0$ to number being denoted (here, 314109_{10} ; the 10 means base-10).

an n -digit decimal number $d_{n-1} \dots d_0$ has value $\sum_{i=0}^{n-1} d_i 10^i$.

Can calculate the value of the string

numbers, binary (base-2)

computers represent numbers in **binary** notation (base-2).

2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1

i.e. a sequence of **binary digits** - **bits**, { 0, 1 }.

an n -digit binary number $b_{n-1} \dots b_0$ has value $\sum_{i=0}^{n-1} b_i 2^i$.

k -bit storage can store 2^k values, ranging from 0 to $2^k - 1$.

1 **byte** is 8 bits. memory is **byte-addressable**.

1 **word** is how many bits a processor handles at a time.

i.e. **register** size. current standard: 8 bytes (64 bits).

Same with base-2

Different as it's binary notation

Why do we care about bytes - the way we work with memory is by bytes

Alternative representation – hexadecimal

Media as numbers

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

The prefix **0x** designates a hex constant.

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Example: **0xDE1DEC2CODE4F00D** is

110111101100001110111100010110000001101111001001111000000001101
 D E C 1 D E 2 C 0 D E 4 F 0 0 D

© 2008-2018 by the MIT 6.172 Lecturers

Each digit needs 4 bits to store

Text can be encoded as bit strings

ASCII

0-	NULL	SIG	STA	ETA	EOT	ENQ	ACK	BELL	"S	^T	Tab	^I	FF	^R	^S0	S1	0-
1-	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESQ	FS	GS	RS	US	1-
2-	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	2-
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	3- *Control
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	4-
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	5-
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	6- *Shift D
7-	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL	7-
-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F		
Transmission									Control								
Format									Extended								
Termination									File								
Control									Record								
Ctrl-Q	02	Null		Ctrl-P	10	Data Link Escape			Ctrl-O	11	Device Control 1						
Ctrl-A	03	Start of Heading		Ctrl-Q	12	Device Control 2			Ctrl-R	13	Device Control 3						
Ctrl-B	02	Start of Text		Ctrl-S	14	Device Control 4			Ctrl-T	15	Negative Acknowledge						
Ctrl-C	03	End of Text		Ctrl-U	16	Synchronous Idle			Ctrl-V	17	End of Transmission Block						
Ctrl-D	04	End of Transmission		Ctrl-X	18	Cancel			Ctrl-Y	19	End of Medium						
Ctrl-E	05	Enquiry		Ctrl-Z	10	Substitute			Ctrl-[10	Escape						
Ctrl-F	06	Acknowledge		Ctrl-\	10	File Separator			Ctrl-]	10	Group Separator						
Ctrl-G	07	Bell	0	Ctrl-^	10	Record Separator			Ctrl-~	10	EOF						
Ctrl-H	08	Backspace															
Ctrl-I	09	Horizontal Tab															
Ctrl-J	0A	Line Feed															
Ctrl-K	0B	Vertical Tab															
Ctrl-L	0C	Form Feed															
Ctrl-M	0D	Carriage Return															
Ctrl-N	0E	Shift In															

Hexadecimal 45 is the letter E

Every character in ASCII takes 1 byte

Sound can be represented by bit numbers

Graphics can be represented by bits - represent a pixel by bytes

Bits and bytes in action

Operations on bit vectors:

operations on bitvectors

for any given bitvector, we can do

- **bitwise** operations
and (`&`), or (`|`), not (`~`), xor (`^`), shift (`<<`, `>>`)
- **logic** operations
(bitvectors interpreted as Booleans; all-0 false, else true)
and (`&&`), or (`||`), not (`!`)
- **arithmetic** operations
(bitvectors interpreted as numbers)
add (`+`), subtract (`-`), multiply (`*`), divide (`/`)

you know the logic operations. let's see arithmetic & shift.

In c you can do logic operations on integers

Bit factors

- We can extend it

extend (zero-)

when you write bitvector **into larger space**, pad 0s at head.

always preserves the numeric value being represented.

example (8 bits to 16-bit storage)

before:

1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

after:

0	0	0	0	0	0	0	1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Truncate
 - Hack off the first byte and throw it away

- Losing some information
- If you truncate some value - equal to doing a modulus operation

truncate

when you write bitvector **into shorter space**, drop bits at head.
easily changes numeric value (if dropped part has non-0 bits).

example (16 bits to 8-bit storage)

before:

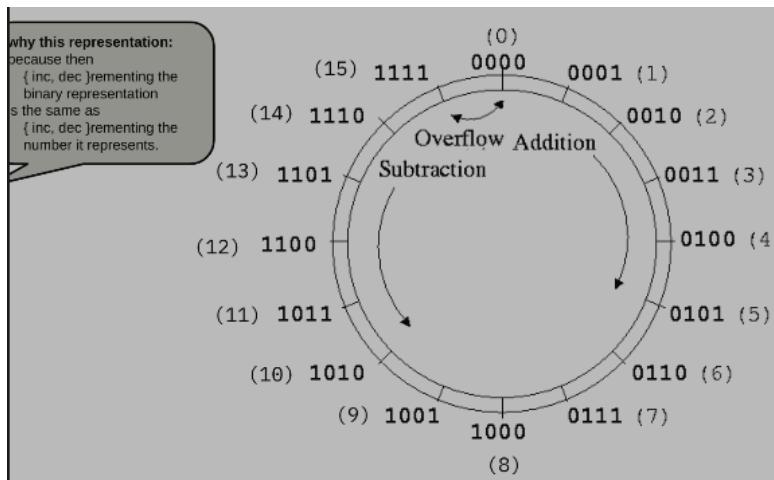
0	1	1	0	0	1	1	0	1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

after:

1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

truncating k to n bits equivalent to modulus op : $k \bmod 2^n$.

Modular arithmetic



addition (unsigned)

like (long-)addition for base-10, just now with bits.

k -bit addition yields $\leq k + 1$ -bit result; truncate to k bits.

example (add 4-bit numbers in base-2)

1	1				← carry bits
		1	1	1	0
+			1	0	1
1	0	0	1	1	

4-bit + **overflowed**; colored bit truncated, resulting in 3.

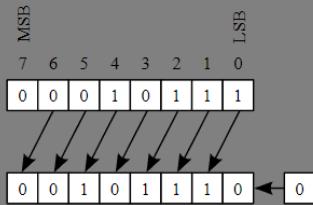
question: per column, which logic op for a) result? b) carry?

Shift

shift (logical-)

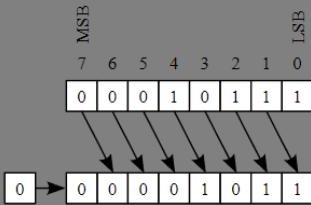
move bits by k spaces, truncate homeless bits, put 0 in empty.

left-shift ($v \ll k$)



multiplication by powers of 2.

right-shift ($v \gg k$)



division by powers of 2.
rounds **towards 0**, not “down”

Move a bit one space - a bit will fall off the left or right side and be forgotten

- Shifting left is multiplying by 2
- Shifting right is division by 2

Multiplication

multiplication (unsigned)

like (long-)multiplication for base-10, just now with bits.

k -bit multiplication yields $\leq 2k$ -bit result; truncate to k bits.

example (multiply 4-bit numbers in base-2)

			← carry bits
		1 1 1	
	*	1 1 1 0	14
		1 0 1	5
		1 1 1 0	14
		0	0
1	1	1 0 0 0	56
1	0	0	70 (truncated: 6)

4-bit * overflowed; colored bit truncated, resulting in 6.

question: which ops needed to implement this?

- Shifting and adding
- Example is shifted twice

Signed and unsigned numbers converting

two's complement

Let $x = \langle x_{w-1}x_{w-2}\dots x_0 \rangle$ be a w -bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k.$$

The prefix **0b** designates a Boolean constant.

For example, the 8-bit word **0b10010110** represents the unsigned value $150 = 2 + 4 + 16 + 128$.

The signed integer (two's complement) value stored in x is

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1}.$$

sign bit

For example, the 8-bit word **0b10010110** represents the signed value $-106 = 2 + 4 + 16 - 128$.

© 2008–2018 by the MIT 6.S72 Lecturers

Negative numbers (sign numbers)

Extend

extend (sign-)

two's complement extension: **pad sign bit at head** instead.

intuition: preserve distance from 0.

example (8 bits to 16-bit storage)

before (for $b \in \{0, 1\}$):

b	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

after:

b	b	b	b	b	b	b	b	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

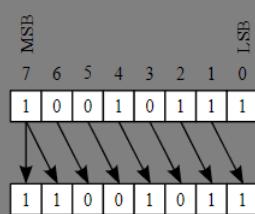
- The distance to zero is restored

shift (arithmetic-)

two's complement right shift: **put sign bit in empty** instead.

two's complement left shift: as in unsigned.

intuition: preserve signage, makes it useful for division by power of 2. (here is where the “towards 0” matters).



addition, multiplication (signed)

compute signed a op b using implementation for unsigned:

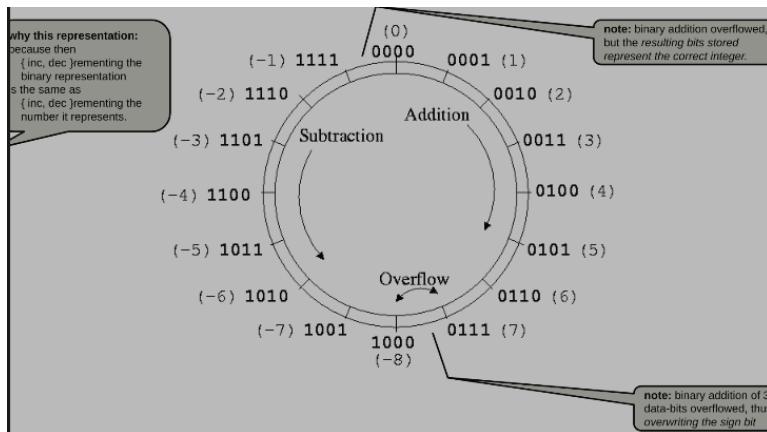
- interpret a and b as unsigned,
- perform op,
- interpret result as signed.

try it on paper for some 4-bit value-pairs.

consequence: need only 1 implementation for both types op.
(isn't two's complement neat?)

We can recycle hardware for both positive(unsigned) and negative(signed) bits

Same clock, different values



0xFFFFFFFF means -1

0x8000000 means the smallest number

Additive inverse

two's complement - additive inverse

Important identity

Since we have $x + \sim x = -1$, it follows that

$$\sim x = \sim x + 1.$$

Example

```
x = 0b011011000
~x = 0b100100111
-x = 0b100101000
```

subtraction (unsigned & signed)

add the additive-inverse: $a - b = a + \sim b$.

works for unsigned numbers, too:

- interpret **a** and **b** as signed,
- add $\sim b$ to **a**,
- interpret result as unsigned.

try it on paper; compute 4-bit **14-5, 14-11, 11-14**.

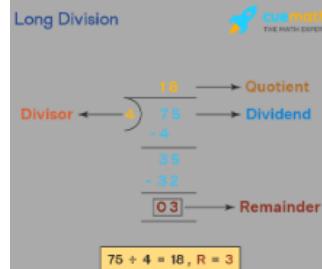
consequence: need only 1 implementation for both types of **-**.

Division

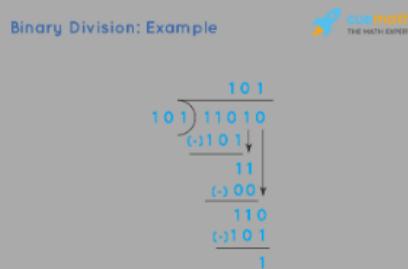
division (unsigned & signed)

like (long-)division for base-10, just now with bits.

example base-10



example base-2



consequence: need only 1 implementation for both types of **/**

question: which previous ops needed to implement this?

Question - subtraction and shifting

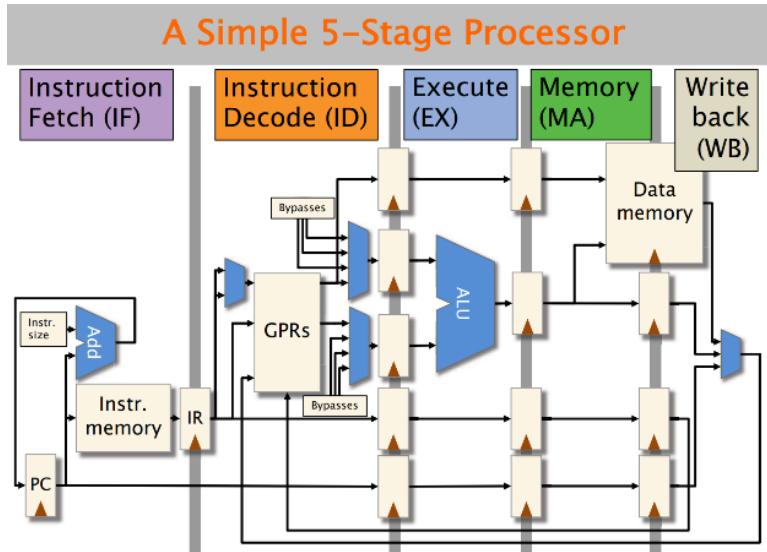
Floating point numbers



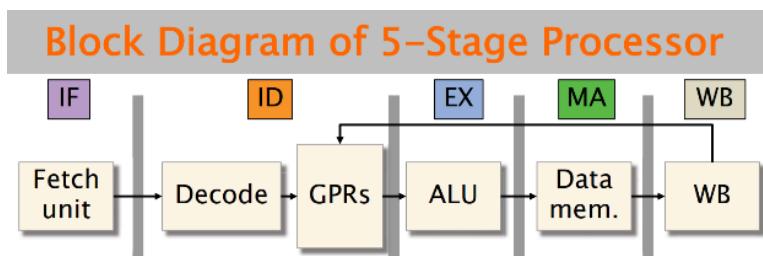
- Sign = whether positive or negative

X86-64

5 stage processor



Instructions are represented by numbers



Each instruction is executed through 5 stages:

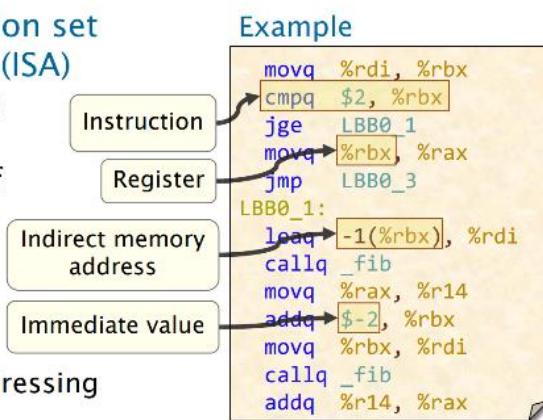
1. **Instruction fetch (IF):** Read instruction from memory.
2. **Instruction decode (ID):** Determine which units to use to execute the instruction, and extract the register arguments.
3. **Execute (EX):** Perform ALU operations.
4. **Memory (MA):** Read/write data memory.
5. **Write back (WB):** Store result into registers.

(his expectations - should recognize language and be able to look the syntax up)

The Instruction Set Architecture

The instruction set architecture (ISA) specifies the syntax and semantics of assembly.

- Registers
- Instructions
- Data types
- Memory addressing modes



Common x86-64 Registers

Number	Width (bits)	Name(s)	Purpose
16	64 (many)		General-purpose registers
6	16	%ss, %[c-g]s	Segment registers
1	64	RFLAGS	Flags register
1	64	%rip	Instruction pointer register
7	64	%cr[0-4,8], %xcro	Control registers
8	64	%mm[0-7]	MMX registers
1	32	mxcsr	SSE2 control register
16	128	%xmm[0-15]	XMM registers (for SSE)
	256	%ymm[0-15]	YMM registers (for AVX)
8	80	%st([0-7])	x87 FPU data registers
1	16	x87 CW	x87 FPU control register
1	16	x87 SW	x87 FPU status register
1	48		x87 FPU instruction pointer register
1	48		x87 FPU data operand pointer register
1	16		x87 FPU tag register
1	11		x87 FPU opcode register

Aliasing

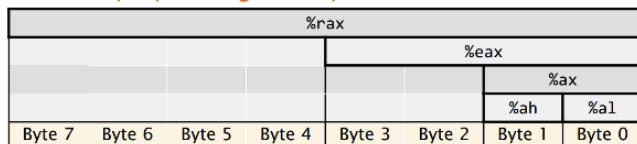
Each register is 64 bits long

How to get different content:

x86-64 Register Aliasing

The x86-64 general-purpose registers are **aliased**: each has multiple names, which refer to overlapping bytes in the register.

General-purpose register layout



Only %rax, %rbx, %rcx, and %rdx have a separate register name for this byte.

x86-64 General-Purpose Registers

64-bit name	32-bit name	16-bit name	8-bit name(s)
%rax	%eax	%ax	%ah, %al
%rbx	%ebx	%bx	%bh, %bl
%rcx	%ecx	%cx	%ch, %cl
%rdx	%edx	%dx	%dh, %dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

AT&T versus Intel Syntax

What does “<op> A, B” mean?

AT&T Syntax	Intel Syntax
B ← B <op> A	A ← A <op> B
movl \$1, %eax	mov eax, 1
addl (%ebx,%ecx,0x2), %eax	add eax, [ebx+ecx*2h]
subq 0x20(%rbx), %rax	sub rax, [rbx+20h]

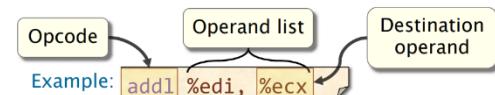
Generated or used by clang, objdump, perf, 6.172 lectures.

Used by Intel documentation.

x86-64 Instruction Format

Format: <opcode> <operand_list>

- <opcode> is a short mnemonic identifying the type of instruction.
- <operand_list> is 0, 1, 2, or (rarely) 3 operands, separated by commas.
- Typically, all operands are sources, and one operand might also be the destination.



Common x86-64 Opcodes

Type of operation	Examples
Data movement	Move
	movs, movz
Arithmetic and logic	Sign or zero extension
	add, sub, mul, imul, div, idiv, lea, sal, sar, shl, shr, rol, ror, inc, dec, neg
	Boolean logic
Conditional jumps	test, cmp
	j<condition>

Note: The subtraction operation “subq %rax, %rbx” computes $\%rbx = \%rbx - \%rax$.

Q means 64 bits

L means 32 bits

Data types

x86-64 Data Types

C declaration	C constant	x86-64 size (bytes)	Assembly suffix	x86-64 data type
char	'c'	1	b	Byte
short	172	2	w	Word
int	172	4	l or d	Double word
unsigned int	172U	4	l or d	Double word
long	172L	8	q	Quad word
unsigned long	172UL	8	q	Quad word
char *	"6.172"	8	q	Quad word
float	6.172F	4	s	Single precision
double	6.172	8	d	Double precision
long double	6.172L	16(10)	t	Extended precision

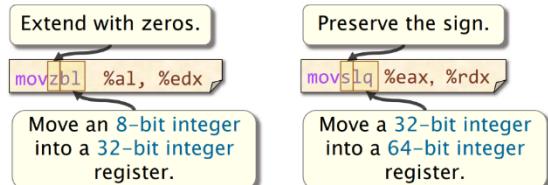
Guide to know the different amount of bits based on suffix

Extensions

Opcode Suffixes for Extension

Sign-extension or zero-extension opcodes use two data-type suffixes.

Examples:



Careful! Results of 32-bit operations are implicitly zero-extended to 64-bit values, unlike the results of 8- and 16-bit operations.

Z is extend with 0

Bl is

Careful - take note to 32 bits

Conditional operations

Jumps and moves

Conditional jumps and conditional moves use a one- or two-character suffix to indicate the condition code.

Example

```
cmpq $4096, %r14
jne .LBB1_1
```

The jump should only be taken if the arguments of the previous comparison are **not equal**.

Register flags

0 - CF - Carry - IF the last ALU....

RFLAGS Register		
Bit(s)	Abbreviation	Description
0	CF	Carry
1		Reserved
2	PF	Parity
3		Reserved
4	AF	Adjust
5		Reserved
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12–63		System flags or reserved

Arithmetic and logic operations update **status flags** in the RFLAGS register.

Decrement %rbx, and set ZF if the result is 0.

Example:

```
decq %rbx
jne .LBB7_1
```

Jump to label .LBB7_1 if ZF is not set.

Bit(s)	Abbreviation	Description
0	CF	Carry
1		Reserved
2	PF	Parity
3		Reserved
4	AF	Adjust
5		Reserved
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12–63		System flags or reserved

The last ALU operation generated a carry or borrow out of the most-significant bit.

The result of the last ALU operation was **0**.

The last ALU operation produced a value whose sign bit was set.

The last ALU operation resulted in arithmetic overflow.

Condition codes

Condition code	Translation	RFLAGS status flags checked
a	if above	CF = 0 and ZF = 0
ae	if above or equal	CF = 0
c	on carry	CF = 1
e	if equal	ZF = 1
ge	if greater or equal	SF = OF
ne	if not equal	ZF = 0
o	on overflow	OF = 1
z	if zero	ZF = 1

Question: Why do the condition codes **e** and **ne** check the zero flag?

Answer: Hardware typically compares integer operands using subtraction.

Addressing modes

The operands of an instruction specify values using a variety of **addressing modes**.

- At most one operand may specify a memory address.

Direct addressing modes

- Immediate:** Use the specified value.
- Register:** Use the value in the specified register.
- Direct memory:** Use the value at the specified memory address.

Examples

```
movq $172, %rdi
```

```
movq %rcx, %rdi
```

```
movq 0x172, %rdi
```

Moving from one place to another - has to go through the CPU

Indirect addressing modes

The x86-64 ISA also supports **indirect addressing**: specifying a memory address by some computation.

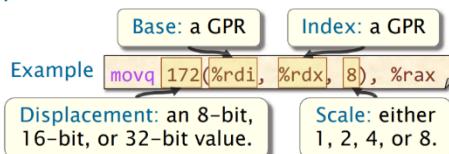
- **Register indirect:** The address is stored in the specified register. **Examples**
`movq (%rax), %rdi`
- **Register indexed:** The address is a constant offset of the value in the specified register. `movq 172(%rax), %rdi`
- **Instruction-pointer relative:** The address is indexed relative to `%rip`. `movq 172(%rip), %rdi`

the address of memory moves the memory address?

Memory address is made up off numbers - as images and so on

Base indexed scale displacement

The most general form of indirect addressing supported by x86-64 is the **base indexed scale displacement** mode.

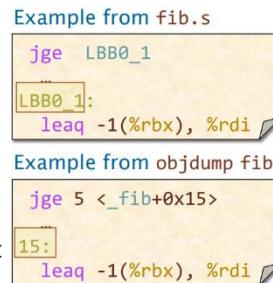


This mode refers to the address **Base + Index*Scale + Displacement**.

If unspecified, **Index** and **Displacement** default to **0**, and **Scale** defaults to **1**.

Jump instructions

The x86-64 jump instructions, `jmp` and `j(condition)`, take a **label** as their operand, which identifies a location in the code.



- Labels can be symbols, exact addresses, or relative addresses.
- An indirect jump takes as its operand an indirect address.

Example: `jmp *%eax`

Take label as operant. Specifies where to jump to - (label: start) where the start of the program is

Assembly idiom 1

The XOR opcode, “`xor A, B`,” computes the bitwise XOR of **A** and **B**.

Question: What does the following assembly do?

`xor %rax, %rax`

Answer: Zeros the register.

Idiom 2

The test opcode, “`test A, B`,” computes the bitwise AND of **A** and **B** and discard the result, preserving the RFLAGS register.

Status flags in RFLAGS		
Bit	Abbreviation	Description
0	CF	Carry
2	PF	Parity
4	AF	Adjust
6	ZF	Zero
7	SF	Sign
11	OF	Overflow

Question: What does the `test` instruction test for in the following assembly snippets?

`test %rcx, %rcx`
`je 400c0a <mm+0xda>`

`test %rax, %rax`
`cmove %rax, %r8`

Answer: Checks to see whether the register is 0.

Idiom 3

The x86-64 ISA includes several no-op (no operation) instructions, including "nop," "nop A," (no-op with an argument), and "data16."

Question: What does this line of assembly do?

```
data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
```

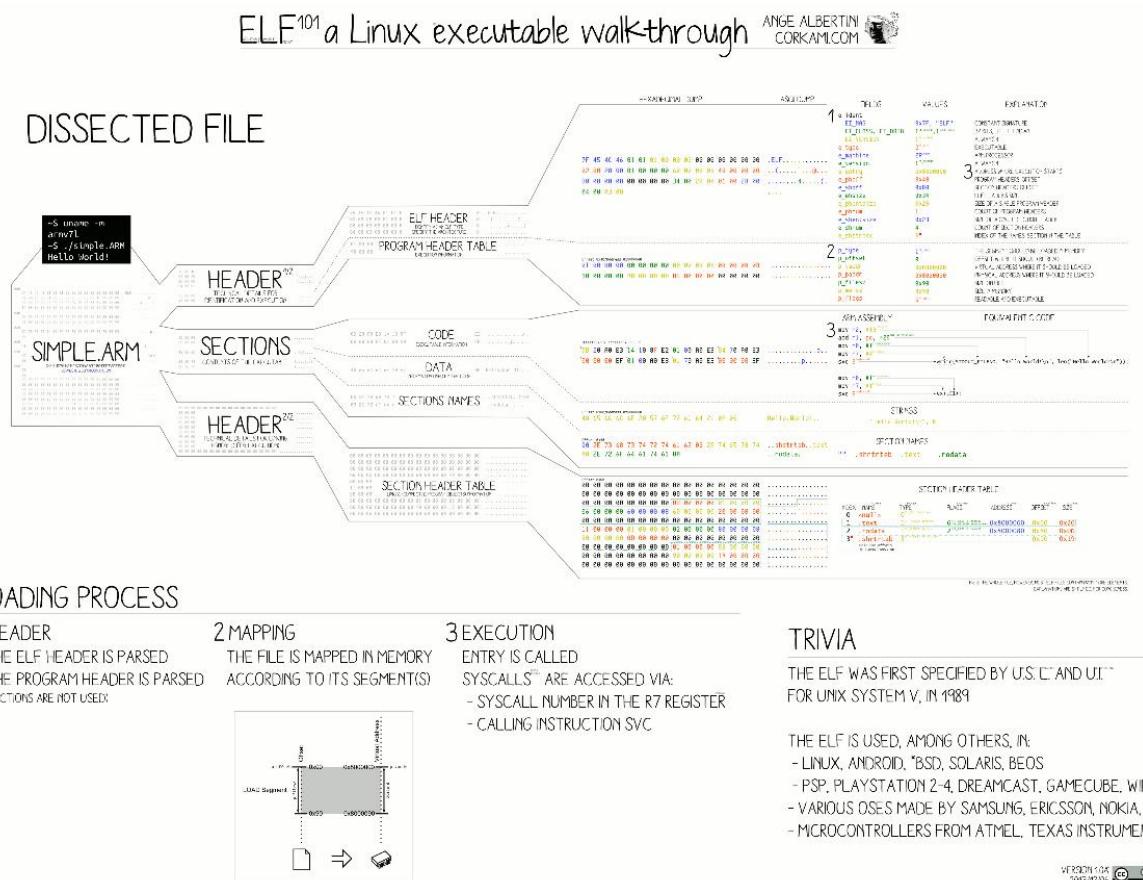
Answer: Nothing!

Question: Why would the compiler generate assembly with these idioms?

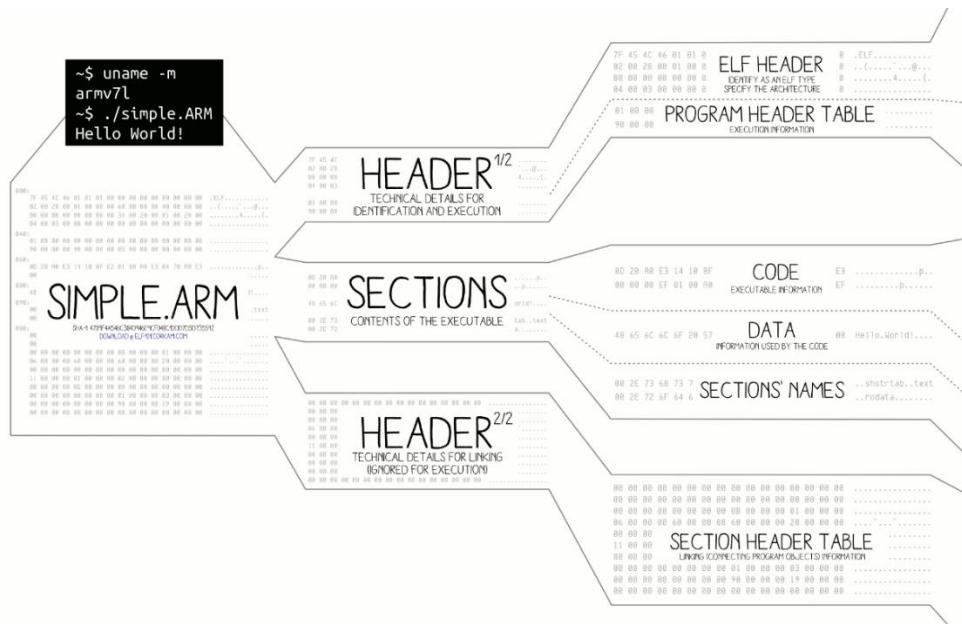
Answer: Mainly, to optimize instruction memory (e.g., code size, alignment).

Will encounter later in Paging (similar to cashing)

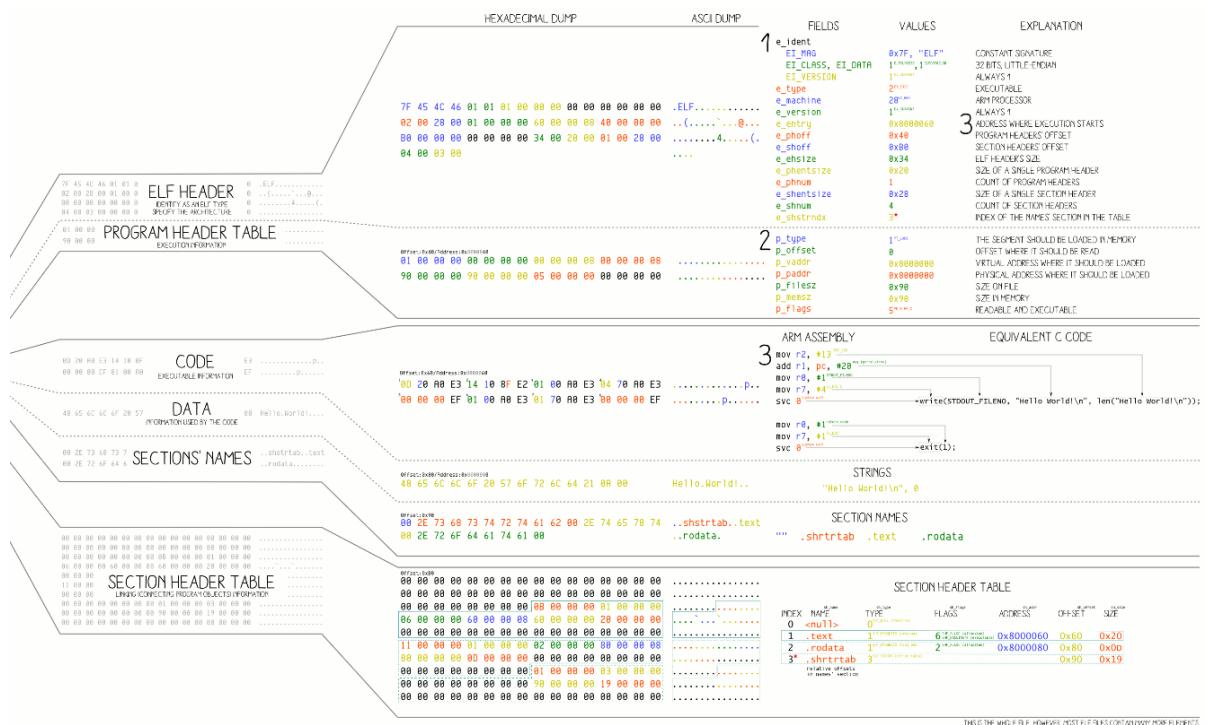
Binary files



Bites consist of header, sections and another header



Header contains meta data about what the operators will do



THIS IS THE WHOLE FILE. HOWEVER, MOST ELF FILES CONTAIN WAY MORE ELEMENTS.

First header - elf header and program header table - made into metadata

- Is it executable, where is the start section etc.

Sections - contains the actual code, data and section names

- Addresses - numbers
- Virtual memory - program under the illusion it has access to the whole memory - if something accesses memory 42, and another does as well, it will be 2 different memory locations

Binary exploits

process
Programs as Data

—

— fascinating.

- Process follows its instructions w/o question.
- Process can rewrite its own instructions!
- Process can perform I/O

what could possibly go wrong?

Buffer overflow attack

Buffer Overflow Attack

Process, Anatomy

—

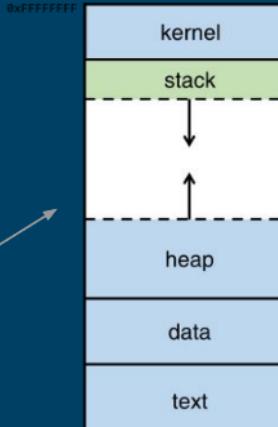
process in memory consists of:

- **text** instructions (the program)
- **data** variables (static size)
- **kernel** command-line parameters
- **heap** large data (`malloc`)

and our main actor:

- **stack** function calls; parameters, return address, function-local variables.

elements arranged as depicted.
stack & heap grow as depicted.



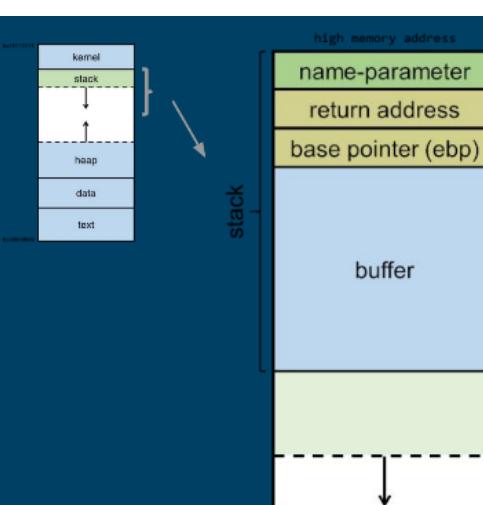
function call allocates a stack frame.

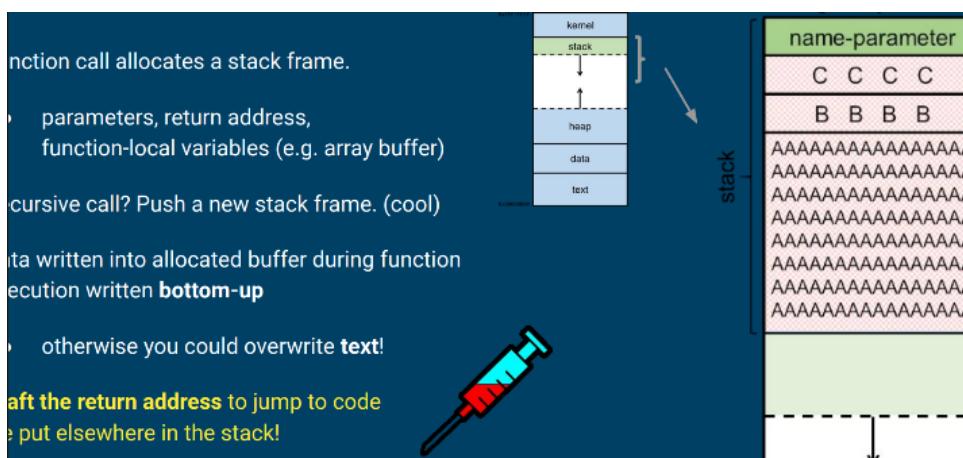
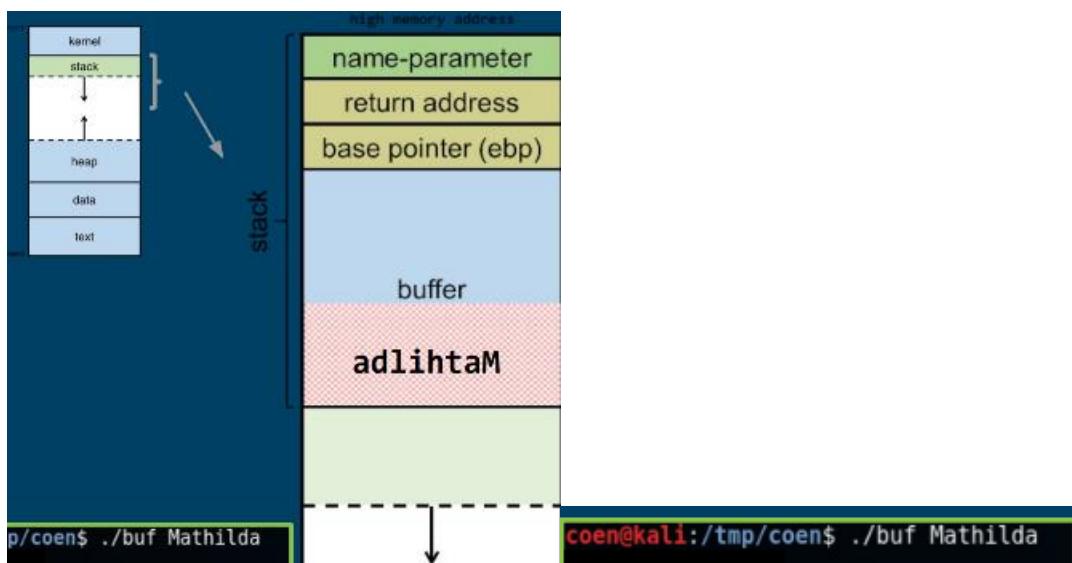
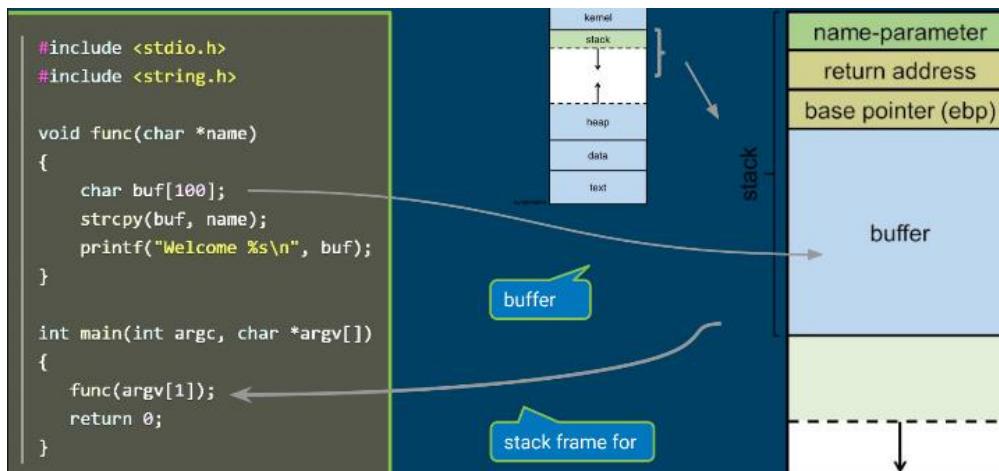
- parameters, return address, function-local variables (e.g. array buffer)

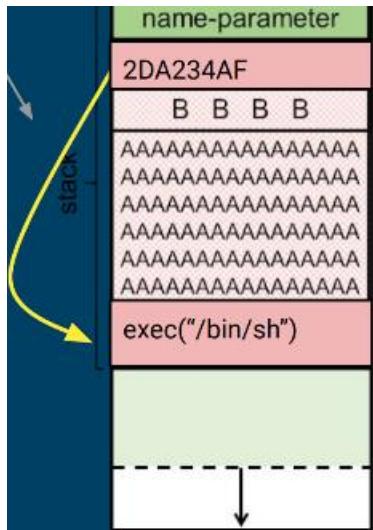
recursive call? Push a new stack frame. (cool)

Data written into allocated buffer during function execution written **bottom-up**

- otherwise you could overwrite **text**!







b) list

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

(gdb) disas func

```
Dump of assembler code for function func:
0x0804841b <+0>: push  %ebp
0x0804841c <+1>: mov   %esp,%ebp
0x0804841e <+3>: sub   $0x64,%esp
0x08048421 <+6>: pushl 0x8(%ebp)
0x08048424 <+9>: lea   -0x64(%ebp),%eax
0x08048427 <+12>: push  %eax
0x08048428 <+13>: call  0x80482f0 <strcpy@plt>
0x0804842d <+18>: add   $0x8,%esp
0x08048430 <+21>: lea   -0x64(%ebp),%eax
0x08048433 <+24>: push  %eax
0x08048434 <+25>: push  $0x80484e0
0x08048439 <+30>: call  0x80482e0 <printf@plt>
0x0804843e <+35>: add   $0x8,%esp
0x08048441 <+38>: nop
0x08048442 <+39>: leave
0x08048443 <+40>: ret
```

allocate 100

```
n $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43")  
program: /tmp/coen/buf $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43")  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB  
received signal SIGSEGV, Segmentation fault.  
43 in ?? ()
```

Segmentation fault: The OS is telling us that the process tried to access something outside of itself (thus, the OS killed it).

How could that happen? Aha! We have overwritten the function **return pointer!** with 'C'; 0x43)

The program is **vulnerable**. Let's craft an attack.

Then look at stack

See the address

(gdb) info registers

eax	0x75	117
ecx	0x75	117
edx	0xb7fb3870	-1208272784
ebx	0x0	0
esp	0xbffffdc4	0xbffffdc4
ebp	0x42424242	0x42424242
esi	0x2	2
edi	0xb7fb2000	-1208279040
eip	0x43434343	0x43434343
eflags	0x10282	[SF IF RF]
cs	0x73	115
ss	0x7b	123
ds	0x7b	123
es	0x7b	123
fs	0x0	0
gs	0x33	51

Attack Scenario

- Port scan target computer with nmap
- Find vulnerable service.
- Buffer overflow, reverse-shell \Rightarrow you are in! but, with few privileges, perhaps? :/
- Find vulnerable binaries on the machine. (that either always run as root, or which are currently running in a process that is running as root)
- Buffer overflow, shell \Rightarrow you are in! with root.

Only programs with unsafe function calls are broken

- Strcpy, strcat, sprint, gets - & array pointer

Compilers and OS gives countermeasures

- OS: memory layout randomization (ASLR), canary...
- HW: Executable space protection
- Compiler: PointGuard, ...

Bit hacks

Set the kth Bit

Problem

Set k th bit in a word x to 1.

Idea

Shift and OR.

```
y = x | (1 << k);
```

Example

$k = 7$

x	1011110101101101
$1 << k$	0000000010000000
$x (1 << k)$	1011110111101101

Clear the kth Bit

Problem

Clear the k th bit in a word x .

Idea

Shift, complement, and AND.

```
y = x & ~(1 << k);
```

Example

$k = 7$

x	101111011101101
$1 << k$	0000000010000000
$\sim(1 << k)$	111111101111111
$x & \sim(1 << k)$	1011110101101101

Extract a Bit Field

Toggle the kth Bit

Problem

Flip the k th bit in a word x .

Idea

Shift and XOR.

```
y = x ^ (1 << k);
```

Example ($0 \rightarrow 1$)

$k = 7$

x	1011110101101101
$1 << k$	0000000010000000
$x ^ (1 << k)$	1011110111101101

```
(x & mask) >> shift;
```

Example

shift = 7

x	1011110101101101
mask	0000011110000000
$x & mask$	0000010100000000
$x & mask >> shift$	00000000000001010

Set a Bit Field

Problem

Set a bit field in a word x to a value y .

Idea

Invert mask to clear, and OR the shifted value.

```
x = (x & ~mask) | (y << shift);
```

Example

shift = 7

x	1011110101101101
y	0000000000000011
mask	0000011110000000
$x & \sim mask$	1011100001101101
$x = (x & \sim mask) (y << shift);$	1011100111101101

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse: $(x \wedge y) \wedge y \Rightarrow x$

Performance

Poor at exploiting *instruction-level parallelism (ILP)*.

No-Branch Minimum

Problem

Find the minimum r of two integers x and y without using a branch.

```
r = y ^ ((x ^ y) & -(x < y));
```

Why it works

- The C language represents the Booleans **TRUE** and **FALSE** with the integers **1** and **0**, respectively.
- If $x < y$, then $-(x < y) \Rightarrow -1$, which is all **1**'s in two's complement representation. Therefore, we have $y \wedge (x \wedge y) \Rightarrow x$.
- If $x \geq y$, then $-(x < y) \Rightarrow 0$. Therefore, we have $y \wedge 0 \Rightarrow y$.

Merging Two Sorted Arrays

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __restrict B,
                  size_t na,
                  size_t nb) {
    while (na > 0 && nb > 0) {
        if (*A <= *B) {
            *C++ = *A++;
            na--;
        } else {
            *C++ = *B++;
            nb--;
        }
    }
    while (na > 0) {
        *C++ = *A++;
        na--;
    }
    while (nb > 0) {
        *C++ = *B++;
        nb--;
    }
}
```

3	12	19	46
4	14	21	23

Branchless

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __restrict B,
                  size_t na,
                  size_t nb) {
    while (na > 0 && nb > 0) {
        long cmp = (*A <= *B);
        long min = *B ^ ((*B ^ *A) & (-cmp));
        *C++ = min;
        A += cmp; na -= cmp;
        B += !cmp; nb -= !cmp;
    }
    while (na > 0) {
        *C++ = *A++;
        na--;
    }
    while (nb > 0) {
        *C++ = *B++;
        nb--;
    }
}
```

This optimization works well on some machines, but on modern machines using `clang -O3`, the branchless version is usually slower than the branching version. Modern compilers can perform this optimization better than you can!

Why Learn Bit Hacks?

- Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly code.
- Because sometimes the compiler doesn't optimize, and you have to do it yourself by hand.
- Because many bit hacks for words extend naturally to bit and word hacks for vectors.
- Because these tricks arise in other domains, and so it pays to be educated about them.
- Because they're fun!

Least-Significant 1

Problem

Compute the mask of the least-significant 1 in word x .

```
r = x & (-x);
```

Example

x	0010000001010000
$-x$	1101111110110000
$x \& (-x)$	00000000000010000

Why it works

The binary representation of $-x$ is $(\sim x) + 1$.

Question

How do you find the index of the bit, i.e., $\lg r$?

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

Bit $\lceil \lg n \rceil - 1$ must be set

```
uint64_t n;
:
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Set bit $\lceil \lg n \rceil$

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000

Populate all bits to the right with 1

Population Count III

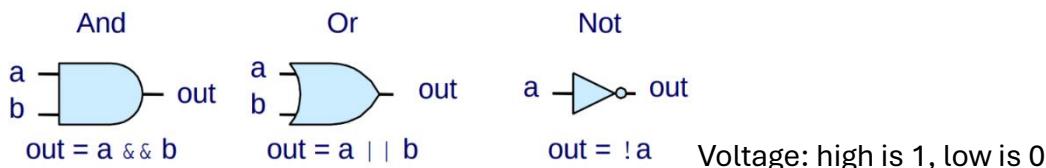
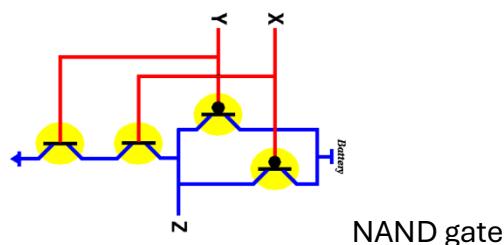
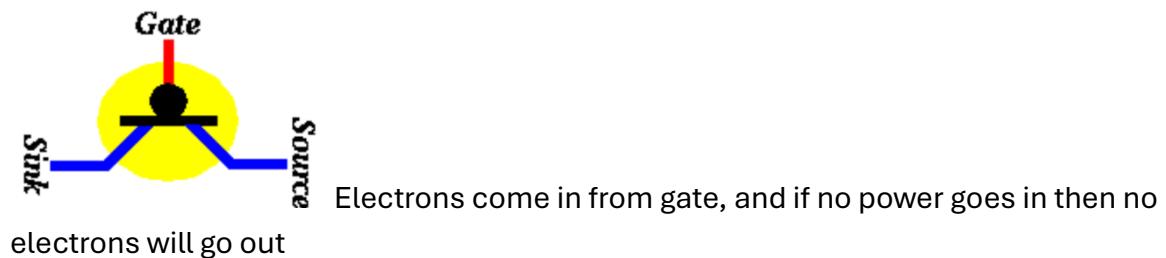
Parallel divide-and-conquer

```
// Create masks
M5 = ~((-1) << 32); // 0^321^32
M4 = M5 ^ (M5 << 16); // (0^161^16)^2
M3 = M4 ^ (M4 << 8); // (0^81^8)^4
M2 = M3 ^ (M3 << 4); // (0^41^4)^8
M1 = M2 ^ (M2 << 2); // (0^21^2)^16
MO = M1 ^ (M1 << 1); // (01)^32
// Compute popcount
x = ((x >> 1) & MO) + (x & MO);
x = ((x >> 2) & M1) + (x & M1);
x = ((x >> 4) + x) & M2;
x = ((x >> 8) + x) & M3;
x = ((x >> 16) + x) & M4;
x = ((x >> 32) + x) & M5;
```

Notation:
 $X^k = \underbrace{XX\dots X}_{k \text{ times}}$

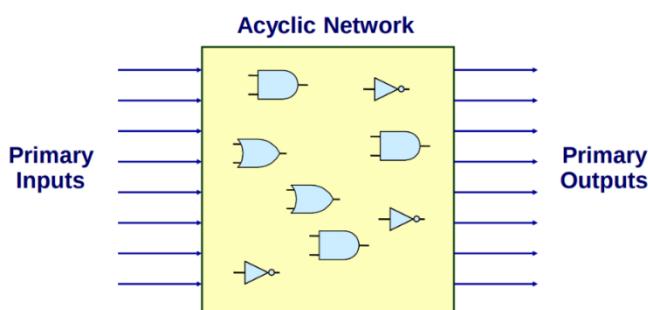
Processors

Digital logic



Combinatorial logic circuits

Combinational Circuits



Changing any of the input will change the outputs as well - solve with register

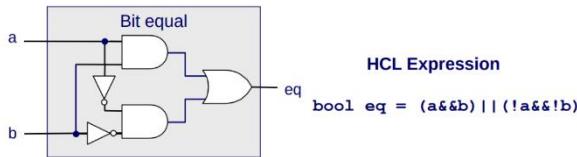
Acyclic Network of Logic Gates

- Continuously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs

How we compare bits are equal

HCL is a language for writing a circuit description

Bit Equality



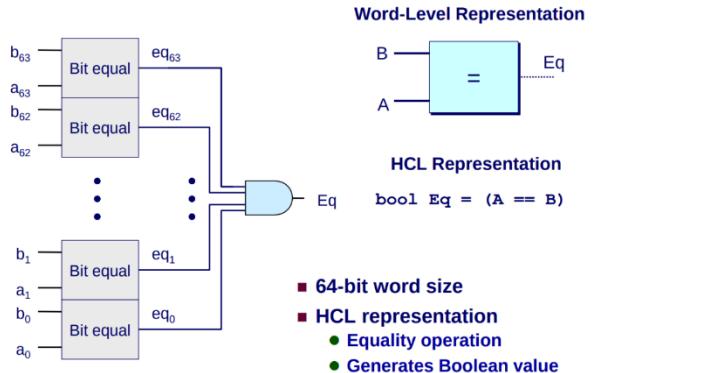
- Generate 1 if a and b are equal

Hardware Control Language (HCL)

- Very simple hardware description language
 - Boolean operations have syntax similar to C logical operations
- We'll use it to describe control logic for processors

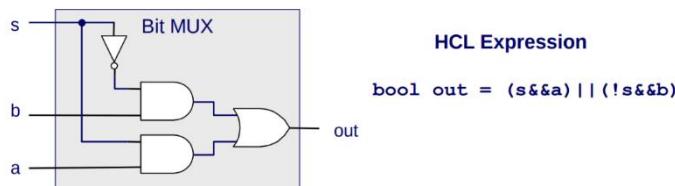
When we can compare bits we can compare words

Word Equality



Now we can compare words

Bit-Level Multiplexor

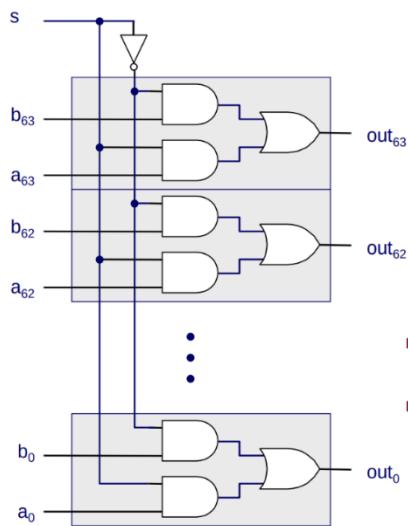


- Control signal s
- Data signals a and b
- Output a when $s=1$, b when $s=0$

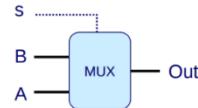
When s is 1 we choose a , so the only way for the output to be one is if a is one

The out is completely determined by a in this case

Word Multiplexor



Word-Level Representation



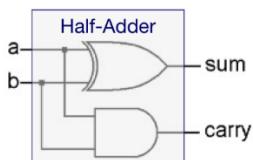
Used for conditional branching

HCL Representation

```
int Out = [
    s : A;
    1 : B;
];
```

- Select input word A or B depending on control signal s
- HCL representation
 - Case expression
 - Series of test : value pairs
 - Output value for first successful test

Adder, Half-

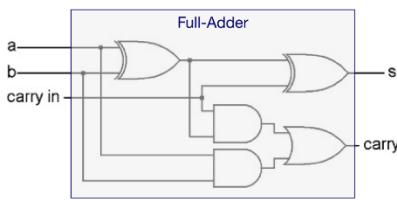


HCL Expression

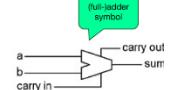
```
bool sum = (a^b)
bool carry = a&b
```

- Output binary-sum to sum, carry-bit to carry.

Adder, Full-

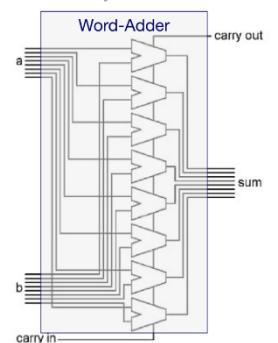


- A Half-adder with a carry-in bit.



Same as before, but has a carry-in

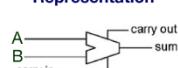
Adder, Word-



- Chain the 1-bit adders, carry-out to carry-in, to get a word-adder.

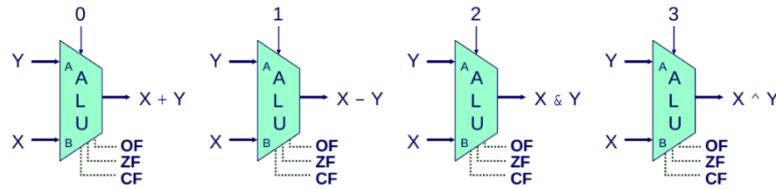
Now we can add words together

Word-Level Representation



Once we have adder's:

Arithmetic Logic Unit

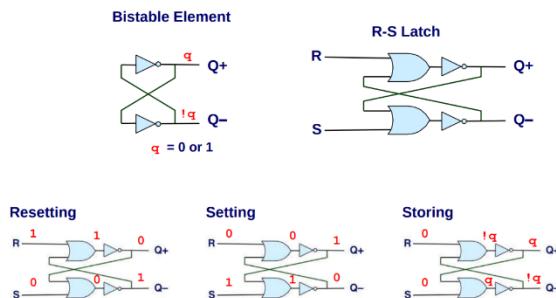


- Combinational logic
 - Continuously responding to inputs
- Control signal selects function computed
 - Corresponding to 4 arithmetic/logical operations in Y86-64
- Also computes values for condition codes

Sum is $x+y$ output bits is OF, ZF and CF

Sequential logic circuits

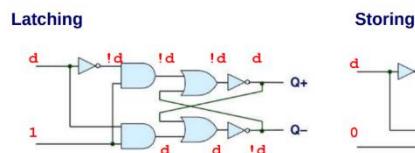
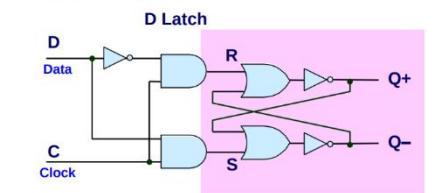
Storing and Accessing 1 Bit



Electrons will be stuck in the latch, which we can read later

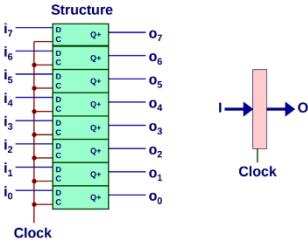
Reading the gate doesn't reenable it, it always has power

1-Bit Latch



If not updating the latch, the value will be what was stuck in the previous part

Registers



- Stores word of data
 - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

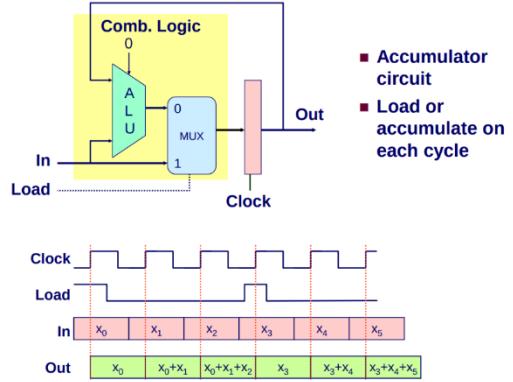
We can store bytes, and whole words now with latches

State Machine Example

Register Operation

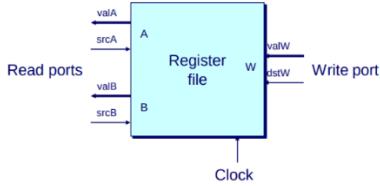


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input



With registers we can implement state machine

Random-Access Memory



- Stores multiple words of memory
 - Address input specifies which word to read or write
- Register file
 - Holds values of program registers
 - %rax, %rsp, etc.
 - Register identifier serves as address
 - » ID 15 (0xF) implies no read or write performed
- Multiple Ports
 - Can read and/or write multiple words in one cycle
 - » Each has separate address and data input/output

Summary

Computation

- Performed by combinational logic
- Computes Boolean functions
- Continuously reacts to input changes

Storage

- Registers
 - Hold single words
 - Loaded as clock rises
- Random-access memories
 - Hold multiple words
 - Possible multiple read or write ports
 - Read word when address input changes
 - Write word as clock rises

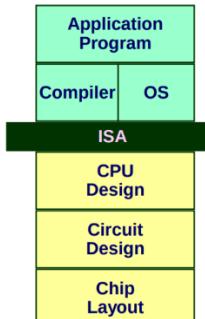
A processor Y86-64

Design (ISA)

Instruction Set Architecture

Assembly Language View

- Processor state
 - Registers, memory, ...
- Instructions
 - `addq, pushq, ret, ...`
 - How instructions are encoded as bytes



Layer of Abstraction

- Above: how to program machine
 - Processor executes instructions in a sequence
- Below: what needs to be built
 - Use variety of tricks to make it run fast
 - E.g., execute multiple instructions simultaneously

-2-

Y86-64 Processor State



- Program Registers
 - 15 registers (omit `%r15`). Each 64 bits
- Condition Codes
 - Single-bit flags set by arithmetic or logical instructions
 - ZF: Zero
 - SF:Negative
 - OF: Overflow
- Program Counter
 - Indicates address of next instruction
- Program Status
 - Indicates either normal operation or some error condition
- Memory
 - Byte-addressable storage array
 - Words stored in little-endian byte order

CS:AF³ -

(

The stack is in memory

Y86-64 Instructions

Format

- 1–10 bytes of information read from memory
 - Can determine instruction length from first byte
 - Not as many instruction types, and simpler encoding than with x86-64
- Each accesses and modifies some part(s) of the program state

Y86-64 Instruction Set #1

Byte	0	1	2	3	4	5	6	7	8	9
halt	[0]	[0]								
nop	[1]	[0]								
cmoveq rA, rB	[2]	[fn]	[rA]	[rB]						
irmovq V, rB	[3]	[0]	[F]	[rB]	V					
rmmovq rA, D(rB)	[4]	[0]	[rA]	[rB]	D					
mrmovq D(rB), rA	[5]	[0]	[rA]	[rB]	D					
OPq rA, rB	[6]	[fn]	[rA]	[rB]						
jXX Dest	[7]	[fn]			Dest					
call Dest	[8]	[0]			Dest					
ret	[9]	[0]								
pushq rA	[A]	[0]	[rA]	[F]						
popq rA	[B]	[0]	[rA]	[F]						

CS:AF

Start at beginning, if its 00 then halt.

Y86-64 Instruction Set #2

Byte	0	1	2	3	4	5	6	
halt	0	0						
nop	1	0						
cmoveXX rA, rB	2	fn	rA	rB				
irmovq V, rB	3	0	F	rB	V			
rmmovq rA, D(rB)	4	0	rA	rB	D			
mrmovq D(rB), rA	5	0	rA	rB	D			
OPq rA, rB	6	fn	rA	rB				
jXX Dest	7	fn			Dest			
call Dest	8	0			Dest			
ret	9	0						
pushq rA	A	0	rA	F				
popq rA	B	0	rA	F				

CS:APP3e

Y86-64 Instruction Set #3

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

CS:APP

Y86-64 Instruction Set #4

Byte	0	1	2	3	4	5	6	7	
halt	0	0							
nop	1	0							
cmoveXX rA, rB	2	fn	rA	rB					
irmovq V, rB	3	0	F	rB	V				
rmmovq rA, D(rB)	4	0	rA	rB	D				
mrmovq D(rB), rA	5	0	rA	rB	D				
OPq rA, rB	6	fn	rA	rB					
jXX Dest	7	fn			Dest				
call Dest	8	0			Dest				
ret	9	0							
pushq rA	A	0	rA	F					
popq rA	B	0	rA	F					

CS:APP3e

Instruction Example

Encoding Registers

Each register has 4-bit ID

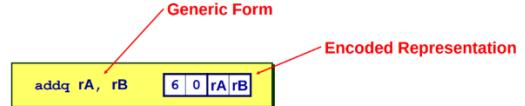
%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

- Same encoding as in x86-64

Register ID 15 (0xF) indicates “no register”

- Will use this in our hardware design in multiple places

Addition Instruction



- Add value in register rA to that in register rB
 - Store result in register rB
 - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
 - e.g., addq %rax, %rsi Encoding: 60 06
- Two-byte encoding
 - First indicates instruction type
 - Second gives source and destination registers

-10-

CS:APP

Move Operations

Arithmetic and Logical Operations

Instruction Code	Function Code
Add	6 0 rA rB
Subtract (rA from rB)	6 1 rA rB
And	6 2 rA rB
Exclusive-Or	6 3 rA rB

- Refer to generically as “Opq”
- Encodings differ only by “function code”
 - Low-order 4 bytes in first instruction word
- Set condition codes as side effect

Register → Register
rrmovq rA, rB 2 0
Immediate → Register
irmovq V, rB 3 0 F rB V
Register → Memory
rmmovq rA, D (rB) 4 0 rA rB D
Memory → Register
mrmovq D (rB), rA 5 0 rA rB D

- Like the x86-64 movq instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

-12-

CS:APP3e

Conditional Move Instructions

Move Unconditionally	rrmovq rA, rB 2 0 rA rB
Move When Less or Equal	cmovele rA, rB 2 1 rA rB
Move When Less	cmove l rA, rB 2 2 rA rB
Move When Equal	cmove e rA, rB 2 3 rA rB
Move When Not Equal	cmove ne rA, rB 2 4 rA rB
Move When Greater or Equal	cmovege rA, rB 2 5 rA rB
Move When Greater	cmoveg rA, rB 2 6 rA rB

- Refer to generically as “cmovxx”
- Encodings differ only by “function code”
- Based on values of condition codes
- Variants of rrmovq instruction
 - (Conditionally) copy value from source to destination register

Least significant byte comes first when translating to hex

Jump Instructions

Jump (Conditionally)

<code>jxx Dest</code>	<code>7 fn</code>	Dest
-----------------------	---------------------	------

- Refer to generically as “jxx”
- Encodings differ only by “function code” fn
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
 - Unlike PC-relative addressing seen in x86-64

Jump Instructions

Jump Unconditionally

<code>jmp Dest</code>	<code>7 0</code>	Dest
-----------------------	--------------------	------

Jump When Less or Equal

<code>jle Dest</code>	<code>7 1</code>	Dest
-----------------------	--------------------	------

Jump When Less

<code>jl Dest</code>	<code>7 2</code>	Dest
----------------------	--------------------	------

Jump When Equal

<code>je Dest</code>	<code>7 3</code>	Dest
----------------------	--------------------	------

Jump When Not Equal

<code>jne Dest</code>	<code>7 4</code>	Dest
-----------------------	--------------------	------

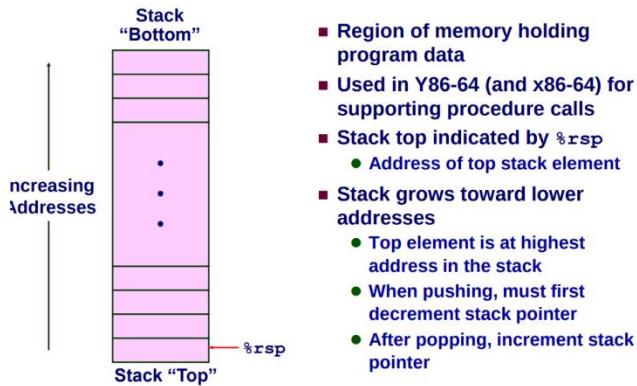
Jump When Greater or Equal

<code>jge Dest</code>	<code>7 5</code>	Dest
-----------------------	--------------------	------

Jump When Greater

<code>jg Dest</code>	<code>7 6</code>	Dest
----------------------	--------------------	------

Y86-64 Program Stack



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by %rsp
 - Address of top stack element
- Stack grows toward lower addresses
 - Top element is at highest address in the stack
 - When pushing, must first decrement stack pointer
 - After popping, increment stack pointer

Stack Operations

<code>pushq rA</code>	<code>A 0 rA F</code>
-----------------------	-----------------------------

- Decrement %rsp by 8
- Store word from rA to memory at %rsp
- Like x86-64

<code>popq rA</code>	<code>B 0 rA F</code>
----------------------	-----------------------------

- Read word from memory at %rsp
- Save in rA
- Increment %rsp by 8
- Like x86-64

Subroutine Call and Return

<code>call Dest</code>	<code>8 0</code>	Dest
------------------------	--------------------	------

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

<code>ret</code>	<code>9 0</code>
------------------	--------------------

- Pop value from stack
- Use as address for next instruction
- Like x86-64

Miscellaneous Instructions

<code>nop</code>	<code>1 0</code>
------------------	--------------------

- Don't do anything

<code>halt</code>	<code>0 0</code>
-------------------	--------------------

- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code	
AOK	1	■ Normal operation
Mnemonic	Code	■ Halt instruction encountered
HLT	2	
Mnemonic	Code	■ Bad address (either instruction or data encountered)
ADR	3	
Mnemonic	Code	■ Invalid instruction encountered
INS	4	

Desired Behavior

- If AOK, keep going
- Otherwise, stop program execution

RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

Fewer, simpler instructions

- Might take more to get given task done
- Can execute them with small and fast hardware

Register-oriented instruction set

- Many more (typically 32) registers
- Use for arguments, return pointer, temporaries

Only load and store instructions can access memory

- Similar to Y86-64 `mrmovq` and `rmmovq`

No Condition codes

- Test instructions return 0/1 in register

Summary

Y86-64 Instruction Set Architecture

- Similar state and instructions as x86-64
- Simpler encodings
- Somewhere between CISC and RISC

How Important is ISA Design?

- Less now than before
 - With enough hardware, can make almost anything go fast

CISC Instruction Sets

- Complex Instruction Set Computer
- IA32 is example

Stack-oriented instruction set

- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

Arithmetic instructions can access memory

- `addq %rax, 12(%rbx,%rcx,8)`
 - requires memory read and write
 - Complex address calculation

Condition codes

- Set as side effect of arithmetic and logical instructions

Philosophy

- Add instructions to perform “typical” programming tasks

CISC vs. RISC

Original Debate

- Strong opinions!
- CISC proponents---easy for compiler, fewer code bytes
- RISC proponents---better for optimizing compilers, can make run fast with simple chip design

Current Status

- For desktop processors, choice of ISA not a technical issue
 - With enough hardware, can make anything run fast
 - Code compatibility more important
- x86-64 adopted many RISC features
 - More registers; use them for argument passing
- For embedded processors, RISC makes sense
 - Smaller, cheaper, less power
 - Most cell phones use ARM processor

A Processor: Y86-64

SEQ State & Flow

State

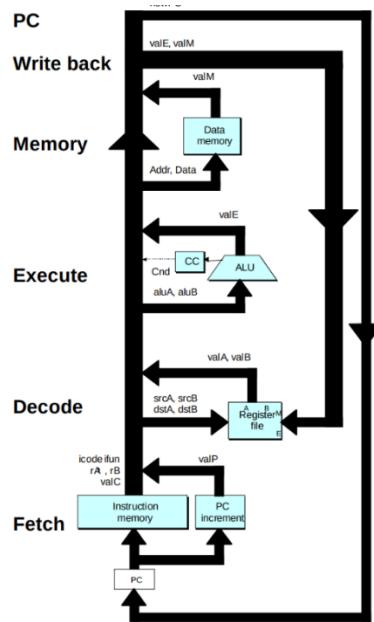
- Program counter register (PC)
- Condition code register (CC)
- Register File
- Memories
 - Access same memory space
 - Data: for reading/writing program data
 - Instruction: for reading instructions

Instruction Flow

- Read instruction at address specified by PC
- Process through stages
- Update program counter

- 10 -

Semantics (SEQ)



SEQ Stages

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

- Read or write data

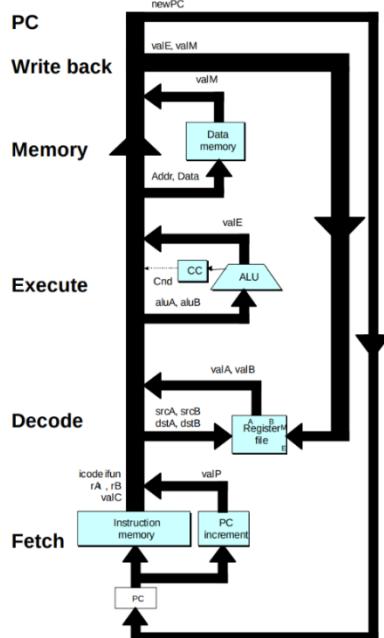
Write Back

- Write program registers

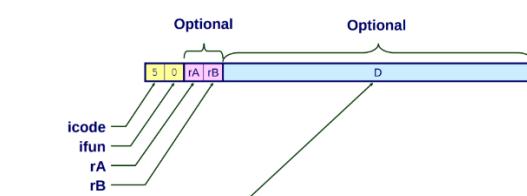
PC

- Update program counter

- 11 -



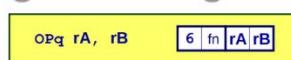
Instruction Decoding



Instruction Format

- Instruction byte icode:ifun
- Optional register byte rA:rB
- Optional constant word valC

Executing Arith./Logical Operation



Fetch

- Read 2 bytes

Decode

- Read operand registers

Execute

- Perform operation
- Set condition codes

Memory

- Do nothing

Write back

- Update register

PC Update

- Increment PC by 2

Increment the pc by bytes

Stage Computation: Arith/Log. Ops

OPq rA, rB	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$
	valP $\leftarrow PC+2$
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$
Execute	valE $\leftarrow valB \text{ OP } valA$ Set CC
Memory	
Write back	R[rB] $\leftarrow valE$
PC update	PC $\leftarrow valP$

Read instruction byte
 Read register byte
 Compute next PC
 Read operand A
 Read operand B
 Perform ALU operation
 Set condition code register
 Write back result
 Update PC

Icode determines operation

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `xmmovq`

<code>xmmovq rA, D(rB)</code>	4 0 rA rB	D
-------------------------------	---------------	---

Fetch	Memory
■ Read 10 bytes	■ Write to memory
Decode	Write back
■ Read operand registers	■ Do nothing
Execute	PC Update
■ Compute effective address	■ Increment PC by 10

Stage Computation: `xmmovq`

<code>xmmovq rA, D(rB)</code>	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valC $\leftarrow M_8[PC+2]$ valP $\leftarrow PC+10$
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$
Execute	valE $\leftarrow valB + valC$
Memory	M ₈ [valE] $\leftarrow valA$
Write back	
PC update	PC $\leftarrow valP$

Read instruction byte
 Read register byte
 Read displacement D
 Compute next PC
 Read operand A
 Read operand B
 Compute effective address
 Write value to memory
 Update PC

- Use ALU for address computation

Executing popq

	<code>popq rA</code>	
Fetch		Memory
	■ Read 2 bytes	■ Read from old stack pointer
Decode		Write back
	■ Read stack pointer	■ Update stack pointer
Execute		■ Write result to register
	■ Increment stack pointer by 8	PC Update
		■ Increment PC by 2

Stage Computation: popq

	<code>popq rA</code>	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$	Read instruction byte Read register byte
Decode	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$	Compute next PC Read stack pointer Read stack pointer
Execute	valE $\leftarrow valB + 8$	Increment stack pointer
Memory	valM $\leftarrow M_8[valA]$	Read from stack
Write back	R[\%rsp] $\leftarrow valE$ R[rA] $\leftarrow valM$	Update stack pointer Write back result
PC update	PC $\leftarrow valP$	Update PC

- Use ALU to increment stack pointer
- Must update two registers
 - Popped value
 - New stack pointer

- 17 -

Executing Conditional Moves

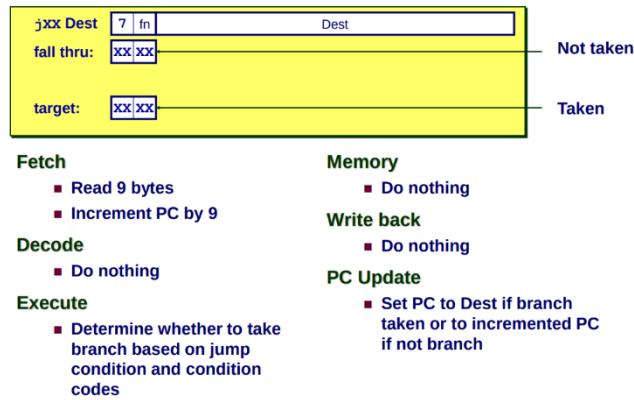
	<code>cmoveXX rA, rB</code>	
Fetch		Memory
	■ Read 2 bytes	■ Do nothing
Decode		Write back
	■ Read operand registers	■ Update register (or not)
Execute		PC Update
	■ If !cnd, then set destination register to 0xF	■ Increment PC by 2

Stage Computation: Cond. Move

	cmoveXX rA, rB
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow 0$
Execute	valE $\leftarrow valB + valA$ If ! Cond(CC,ifun) rB $\leftarrow 0xF$
Memory	
Write back	R[rB] $\leftarrow valE$
PC update	PC $\leftarrow valP$

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
 - If condition codes & move condition indicate no move

Executing Jumps

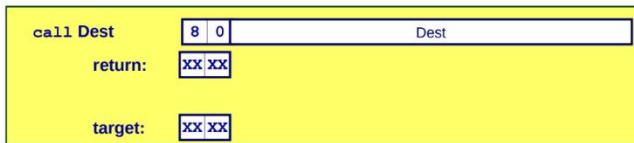


Stage Computation: Jumps

	jXX Dest
Fetch	icode:ifun $\leftarrow M_1[PC]$ valC $\leftarrow M_8[PC+1]$ valP $\leftarrow PC+9$
Decode	
Execute	Cnd $\leftarrow Cond(CC,ifun)$
Memory	
Write back	
PC update	PC $\leftarrow Cnd ? valC : valP$

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Executing call



Fetch	Memory
■ Read 9 bytes	■ Write incremented PC to new value of stack pointer
■ Increment PC by 9	
Decode	Write back
■ Read stack pointer	■ Update stack pointer
Execute	PC Update
■ Decrement stack pointer by 8	■ Set PC to Dest

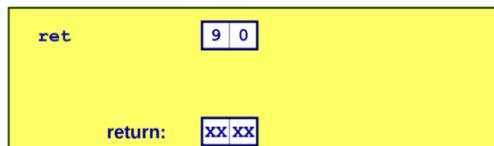
Stage Computation: call

call Dest	
Fetch	icode:ifun $\leftarrow M_1[PC]$
	valC $\leftarrow M_8[PC+1]$
	valP $\leftarrow PC+9$
Decode	valB $\leftarrow R[\%rsp]$
Execute	valE $\leftarrow valB + -8$
Memory	$M_8[valE] \leftarrow valP$
Write back	$R[\%rsp] \leftarrow valE$
PC update	$PC \leftarrow valC$

Read instruction byte
Read destination address
Compute return point
Read stack pointer
Decrement stack pointer
Write return value on stack
Update stack pointer
Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing ret



Fetch	Memory
■ Read 1 byte	■ Read return address from old stack pointer
Decode	Write back
■ Read stack pointer	■ Update stack pointer
Execute	PC Update
■ Increment stack pointer by 8	■ Set PC to return address

Stage Computation: ret

ret	
Fetch	icode:ifun $\leftarrow M_1[PC]$
Decode	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$
Execute	valE $\leftarrow valB + 8$
Memory	valM $\leftarrow M_8[valA]$
Write back	$R[\%rsp] \leftarrow valE$
PC update	PC $\leftarrow valM$

Read instruction byte

Read operand stack pointer

Read operand stack pointer

Increment stack pointer

Read return address

Update stack pointer

Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

Computation Steps

OPq rA, rB		
Fetch	icode,ifun	icode:ifun $\leftarrow M_1[PC]$
	rA,rB	rA:rB $\leftarrow M_1[PC+1]$
	valC	[Read constant word]
	valP	valP $\leftarrow PC+2$
Decode	valA, srcA	valA $\leftarrow R[rA]$
	valB, srcB	valB $\leftarrow R[rB]$
Execute	valE	valE $\leftarrow valB \text{ OP } valA$
	Cond code	Set CC
Memory	valM	
Write back	dstE	$R[rB] \leftarrow valE$
	dstM	[Write back ALU result]
PC update	PC	Update PC

Read instruction byte

Read register byte

[Read constant word]

Compute next PC

Read operand A

Read operand B

Perform ALU operation

Set/use cond. code reg

[Memory read/write]

Write back ALU result

[Write back memory result]

Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

call Dest		
Fetch	icode,ifun	icode:ifun $\leftarrow M_1[PC]$
	rA,rB	[Read register byte]
	valC	Read constant word
	valP	Compute next PC
Decode	valA, srcA	[Read operand A]
	valB, srcB	Read operand B
Execute	valE	Perform ALU operation
	Cond code	[Set /use cond. code reg]
Memory	valM	Memory read/write
Write back	dstE	Write back ALU result
	dstM	[Write back memory result]
PC update	PC	Update PC

Read instruction byte

[Read register byte]

Read constant word

Compute next PC

[Read operand A]

Read operand B

Perform ALU operation

[Set /use cond. code reg]

Memory read/write

Write back ALU result

[Write back memory result]

Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computed Values

Fetch		Execute	
icode	Instruction code	valE	ALU result
ifun	Instruction function	Cnd	Branch/move flag
rA	Instr. Register A	Memory	
rB	Instr. Register B	valM	Value from memory
valC	Instruction constant		
valP	Incremented PC		
Decode			
srcA	Register ID A		
srcB	Register ID B		
dstE	Destination Register E		
dstM	Destination Register M		
valA	Register value A		
valB	Register value B		

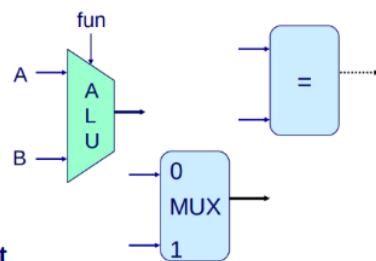
A processor: Y86-64

implementation (SEQ)

Building Blocks

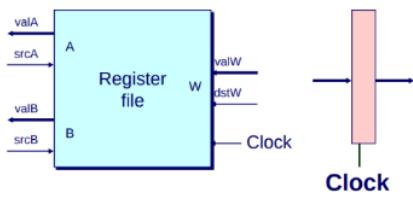
Combinational Logic

- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control



Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises



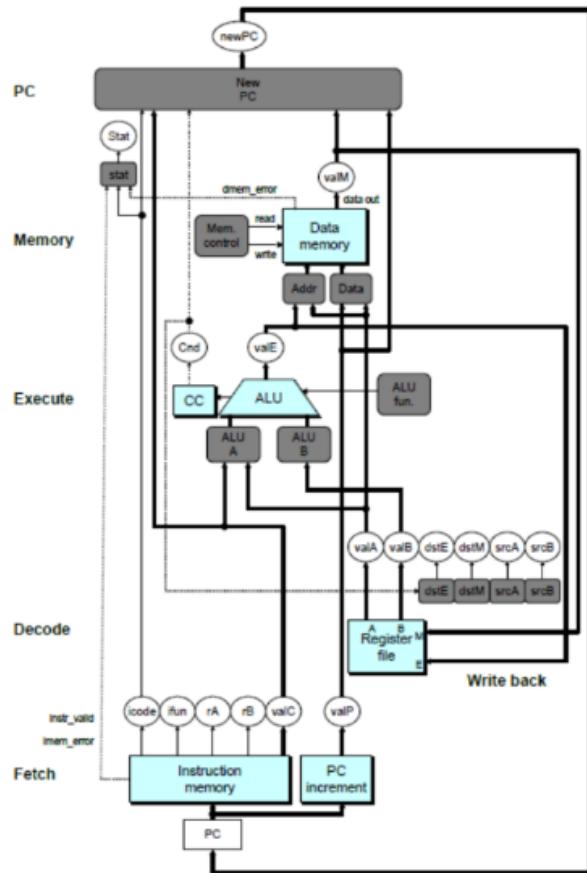
- 6 -

CS:APP3e

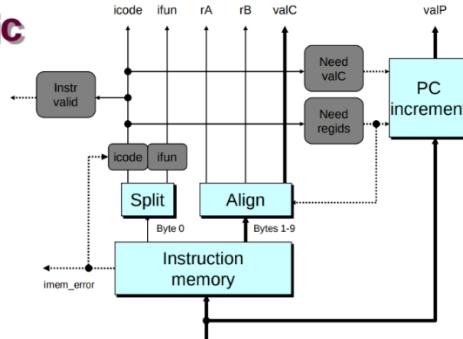
SEQ Hardware

Key

- Blue boxes: predefined hardware blocks
 - E.g., memories, ALU
- Gray boxes: control logic
 - Describe in HCL
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



Fetch Logic



Predefined Blocks

- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
 - Signal invalid address
- Split: Divide instruction byte into icode and ifun
- Align: Get fields for rA, rB, and valC

SEQ Summary

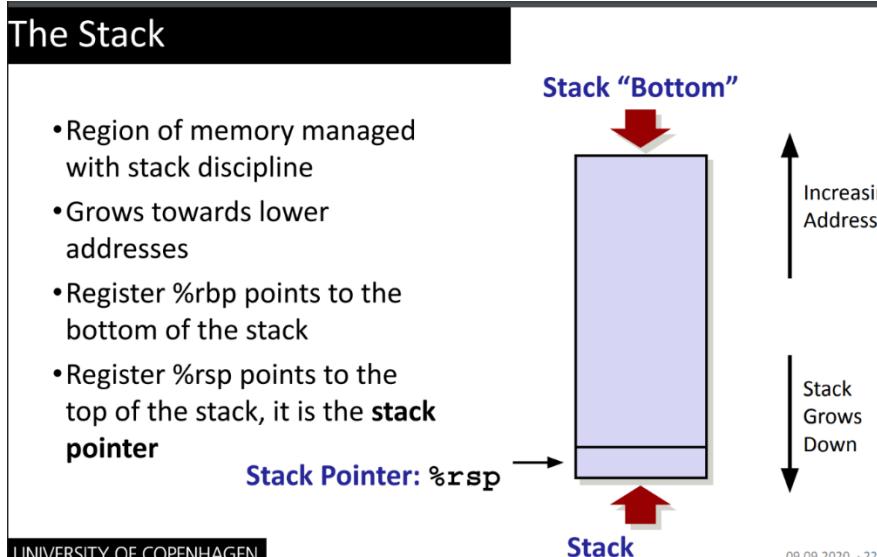
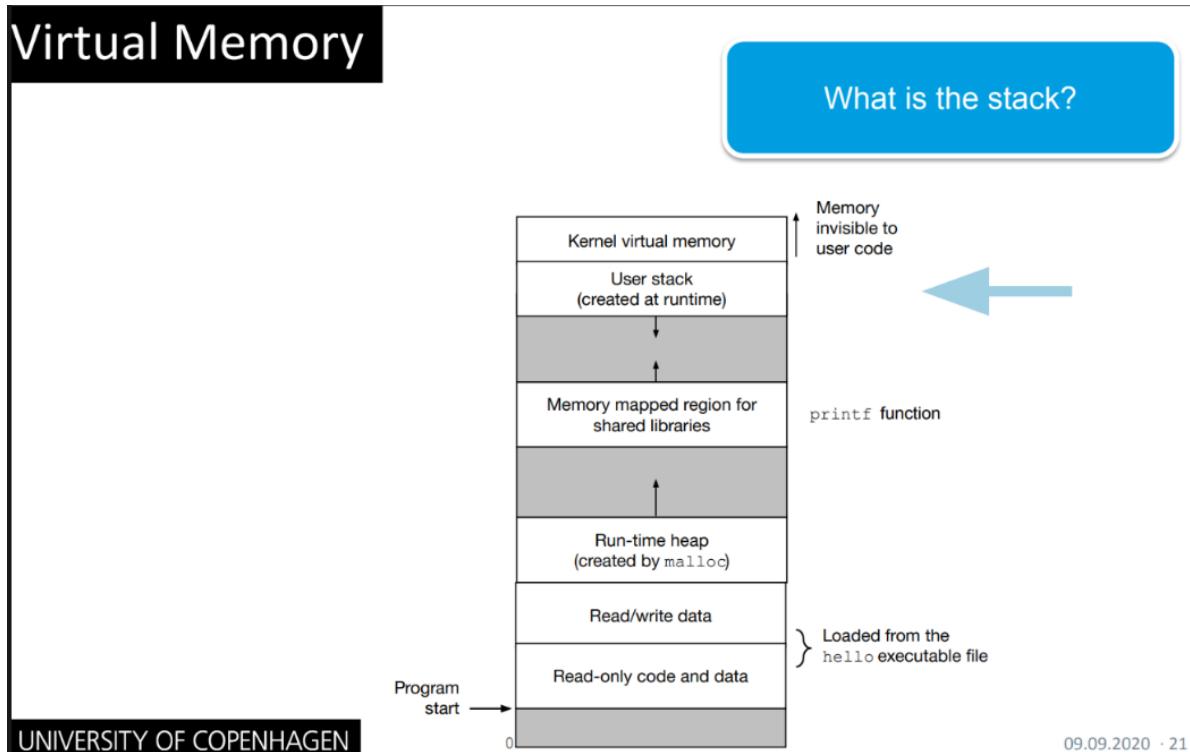
Implementation

- Express every instruction as series of simple steps
- Follow same general flow for each instruction type
- Assemble registers, memories, predefined combinational blocks
- Connect with control logic

Limitations

- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle

Stack, procedure calls

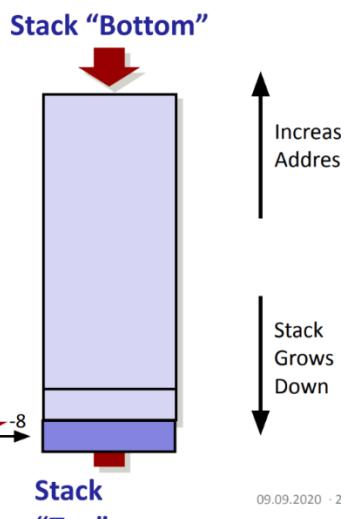


Stack - Push

Operator push
pushq %r10

1. Decrement %rsp by 8
2. Write contents of %r10 at address given by %rsp

Stack Pointer: %rsp  -8



UNIVERSITY OF COPENHAGEN

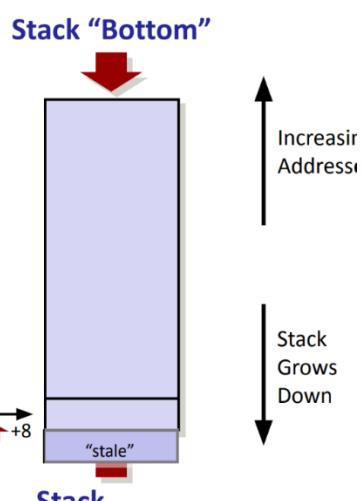
09.09.2020 · 2

Stack - Pop

Operator pop
popq %r10

1. Copy contents at address %rsp to %r10
2. Increment %rsp by 8

Stack Pointer: %rsp  +8



UNIVERSITY OF COPENHAGEN

09.09.2020 · 24

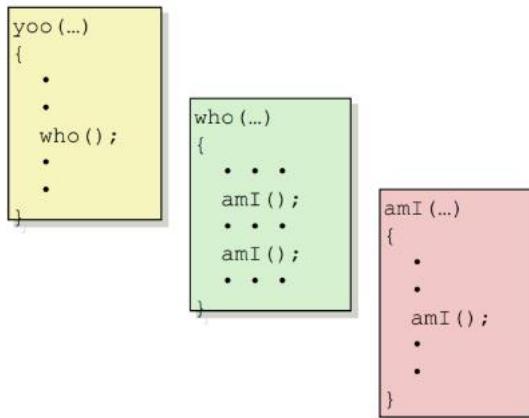
Procedure Call

Instructions:

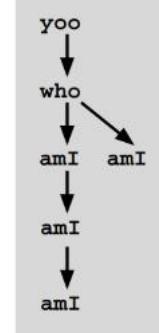
- **call**: push **return address** on stack; jump to label/address
 - Return address is address of instruction right after call instruction
- **ret**: pop address from stack; jump to address

Example

why/how is the stack relevant
when talking about procedure calls?



Example
Call Chain

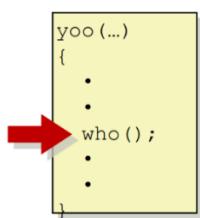


Procedure `amI ()` is recursive

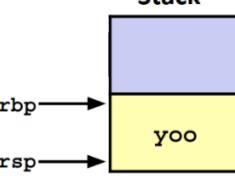
UNIVERSITY OF COPENHAGEN

we use the stack to keep track of deep
(possibly recursive) procedure calls.

Example



Stack



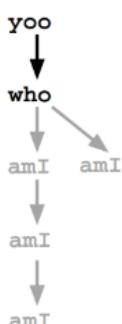
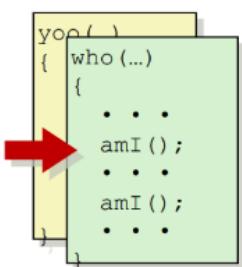
stack frame

- caller's saved registers⁽¹⁾
- local vars
- args
- return address⁽²⁾

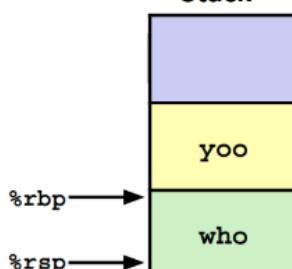
⁽¹⁾: to restore caller state on return
(bottom frame has no caller).

⁽²⁾: pushes it when it calls
(top frame does not need this)

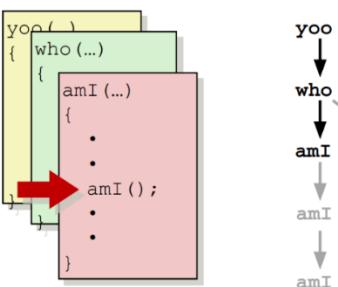
Example



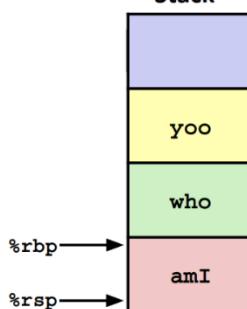
Stack



Example

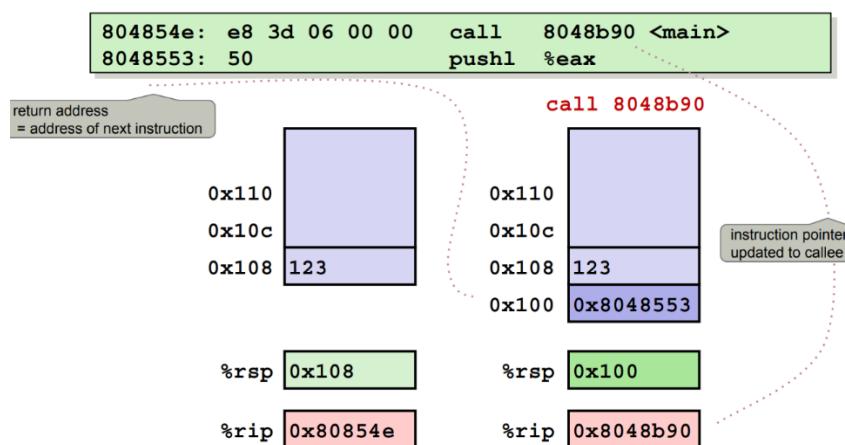


Stack

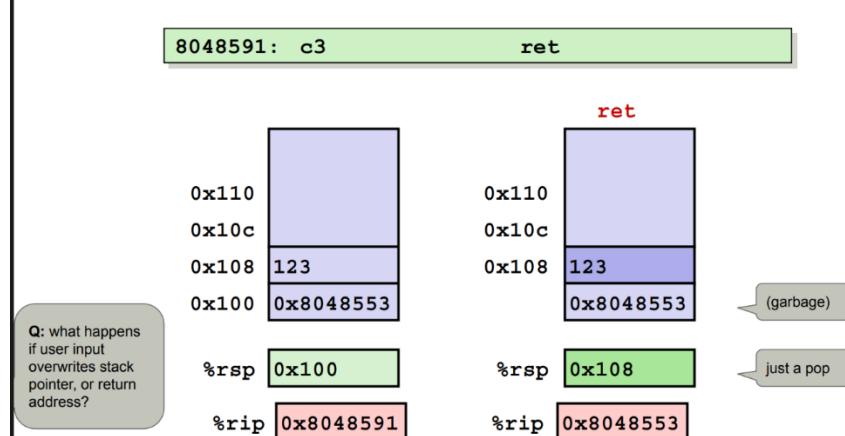


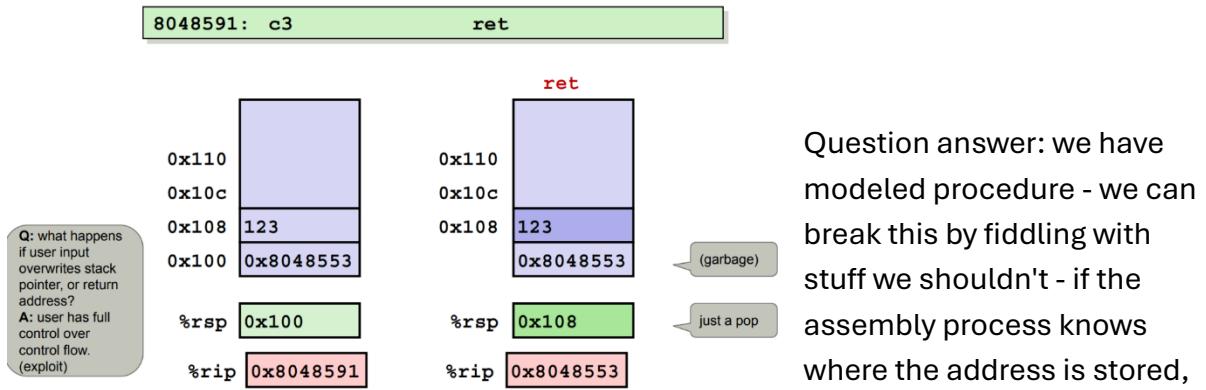
Etc. in slides

Procedure Call Example



Procedure Call Example





- Example of code injection

Calling a function (x86-64)

To call a function, a program:

1. **Places the first six integer or pointer parameters in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`**
2. Pushes onto the stack subsequent parameters and parameters larger than 8B (in order).
3. Executes the call instruction, which:
 - Pushes the return address onto the stack
 - Jumps to the start of the specified function

GP Register Usage during function calls

First six arguments of a function stored in `di, si, dx, cx, r8, r9`

Remaining arguments are on the stack (more later)

Return value is in `rax`

What is the stack?
How is the stack used?

Calling a function preserves `rbp, rbx, r12-15`. The other registers **might be overwritten**.

Executing a function

The C run-time system introduces instruction to set-up and clean-up the stack in each procedure.

Set-up consists in allocation and initialization of a stack-frame. Clean-up: deallocating a stack frame.

A stack-frame is the space needed on the stack by a procedure for storing:

- The return address
- (some) parameters
- Local variables

Stack Frame

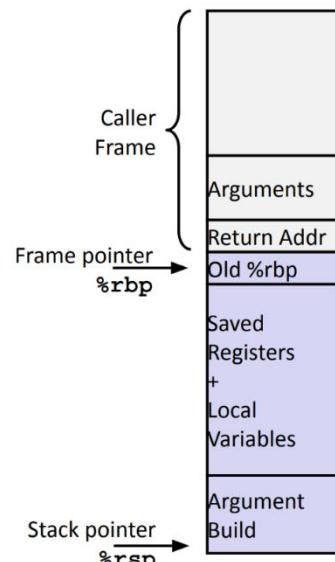
Caller:

- Arguments
 - pushed by program (if needed)
- Return address
 - pushed by call

Callee:

- Previous frame pointer (%rbp)
- Other callee-save registers (%rbx, %r12-15)
- Space for local variables
- Arguments for next function (when about to call another function)

UNIVERSITY OF COPENHAGEN



Example

```
arith.s > arith.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int logical(int x, int y)
5 {
6     int t1 = x^y;
7     int t2 = t1 >> 17;
8     int mask = (1<<13) - 7;
9     int rval = t2 & mask;
10    return rval;
11 }
12
13 int main(int argc, char* argv[])
14 {
15     if (argc != 3) {
16         printf("Usage: arith x y\n");
17         return 1;
18     }
19
20     int x = atoi(argv[1]);
21     int y = atoi(argv[2]);
22     printf("Arguments x: %d, y: %d\n", x, y);
23     printf("Logical returns: %d\n", logical(x,y));
24     printf("\n");
25
26     return 0;
27 }
```

Example

```
5 logical:  
6 .LFB2:  
7     .cfi_startproc  
8     pushq %rbp  
9     .cfi_offset %rbp, -16  
10    movq %rsp, %rbp  
11    .cfi_offset %rbp, 16  
12    movl %edi, -20(%rbp)  
13    movl %esi, -24(%rbp)  
14    movl -20(%rbp), %eax  
15    xorl -24(%rbp), %eax  
16    movl %eax, -16(%rbp)  
17    movl -16(%rbp), %eax  
18    sarl $17, %eax  
19    movl %eax, -12(%rbp)  
20    movl $8185, -8(%rbp)  
21    movl -12(%rbp), %eax  
22    andl -8(%rbp), %eax  
23    movl %eax, -4(%rbp)  
24    movl -4(%rbp), %eax  
25    popq %rbp  
26    .cfi_offset %rbp, 8  
27    ret  
28    .cfi_endproc
```

Set-up:

- Previous stack frame base %rbp pushed on stack
- %rbp is the only callee save register
- Frame pointer re-initialised

Function:

- 4 local variables at positions relative to stack frame base %rbp
 - t1: -16(%rbp)
 - t2: -12(%rbp)
 - mask: -8(%rbp)
 - rval: -4(%rbp)
- %eax holds intermediate results
- %eax holds return value at the end of the function

remember
stack
grows
down

Clean-up:

- Previous stack frame base restored
- ret manipulates %rsp and %rip to return control to return address

Buffer overflows

Vulnerable C code

example of vulnerable C code

```
echo.c ➔  
1 #include <stdio.h>  
2  
3 void echo()  
4 {  
5     char buf[4];  
6     gets(buf);  
7     puts(buf);  
8 }  
9  
10 int main()  
11 {  
12     echo();  
13     return 0;  
14 }  
15  
16 }
```

disable stack protection
(more on that later today)
(so it's not really a problem
today; compilers prevent
problem by default)

```
% gcc -fno-stack-protector echo.c -o echonp
```

```
% ./echonp  
12345678  
12345678  
% ./echonp  
123456789  
123456789  
% ./echonp  
12345678901234567890123  
123456789012345678901234567890123  
% ./echonp  
123456789012345678901234567890123  
123456789012345678901234567890123  
[1] 19108 segmentation fault (core dumped) ./echonp
```

process tried I/O on
kernel virtual memory
(kernel-space).
what's happening?
let's investigate.

Typical hint that there is a buffer overflow

Vulnerable C code

```
objdump -D echonp >> echonp.d
```

The stack is 16B aligned

=>

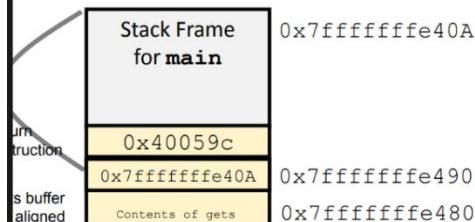
rsp is decreased with 16B
sub \$0x10, %rsp

```
323 0000000000400566 <echo>:
324 400566: 55
325 400567: 48 89 e5
326 40056a: 48 83 ec 10
327 40056e: 48 8d 45 f0
328 400572: 48 89 c7
329 400575: b8 00 00 00 00
330 40057a: e8 d1 fe ff ff
331 40057f: 48 8d 45 f0
332 400583: 48 89 c7
333 400586: e8 a5 fe ff ff
334 40058b: 90
335 40058c: c9
336 40058d: c3
337
338 000000000040058e <main>:
339 40058e: 55
340 40058f: 48 89 e5
341 400592: b8 00 00 00 00
342 400597: e8 ca ff ff ff
343 40059c: b8 00 00 00 00
344 4005a1: 5d
345 4005a2: c3
346 4005a3: 66 2e 0f 1f 84 00 00
347 4005aa: 00 00 00
348 4005ad: 0f 1f 00
349
push %rbp
mov %rsp,%rbp
sub $0x10,%rsp
lea -0x10(%rbp),%rax
mov %rax,%rdi
mov $0x0,%eax
callq 400450 <gets@plt>
lea -0x10(%rbp),%rax
mov %rax,%rdi
callq 400430 <puts@plt>
nop
leaveq
retq
push old base pointer on stack
new base pointer := old stack pointer
new stack pointer := old stack pointer - 16
= %rsp
address of buffer into rax then into 1st arg
initialize return
evict stack frame
```

enough space for buf

Vulnerable code

Before call to gets



so, why vulnerable?

```
% gdb ./echonp
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./echonp...done.
(gdb) break echo
Breakpoint 1 at 0x40056e: file echo.c, line 6.
(gdb) r
Starting program: /home/phbo/Class/C/TMP/echonp

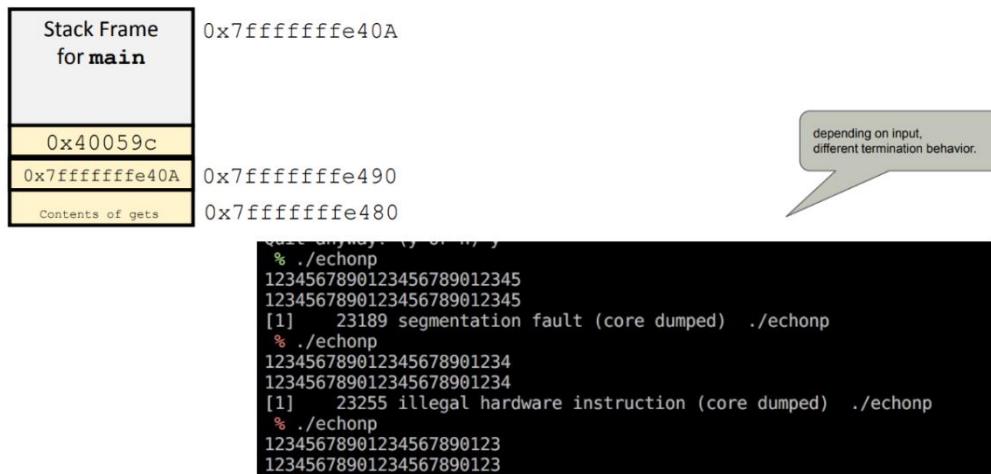
Breakpoint 1, echo () at echo.c:6
6       gets(buf);
(gdb) p/x $rbp
$1 = 0x7fffffe490
(gdb) p/x *(unsigned long*)$rbp
$2 = 0x7fffffe4a0
(gdb) p/x *(unsigned long*)$rbp+1
$3 = 0x40059c
```

%rbp+1 is return instruction

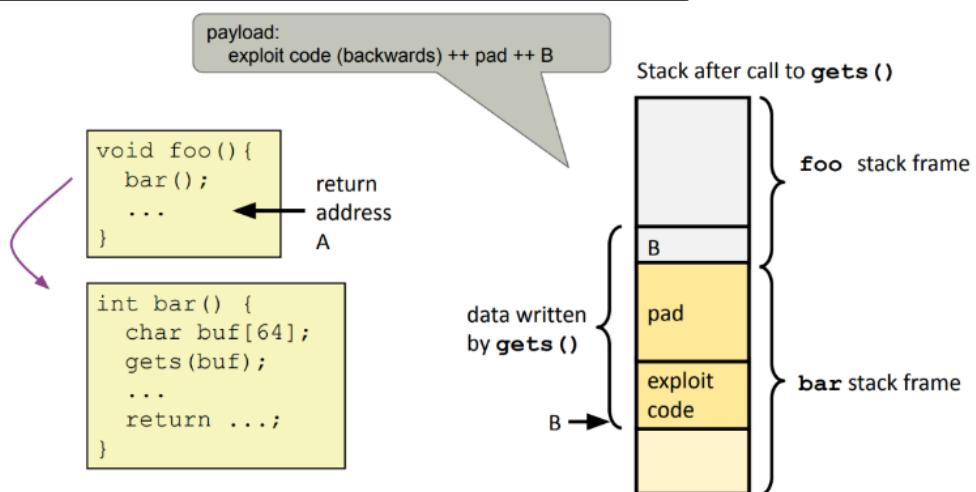
```
342 400597: e8 ca ff ff ff    callq 400566 <echo>
343 40059c: b8 00 00 00 00    mov    $0x0,%eax
```

Buffer Overflow Stack Example

Before call to gets



Malicious Use of Buffer Overflow



Input string contains byte representation of executable code
Overwrite return address with address of exploit code
When `bar()` executes `ret`, will jump to exploit code

Exploits Based on Buffer Overflows

Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines

Internet worm

Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:

- `finger droh@cs.cmu.edu`

Morris worm, 1988

Worm attacked fingerd server by sending phony argument:

- `finger "exploit-code padding new-return-address"`

- exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

UNIVERSITY OF COPENHAGEN

how to stop this:

- want to make it impossible to execute code on the stack.
- want to make it hard to figure out where exploit code starts
- want to protect return address.

Parallelism

Notes

Prediction strategy

- Instructions that don't transfer control
 - Predict next PX to be ValP (increment)
 - Always reliable
- Call and unconditional jumps
 - Predict next PC to be valC (destination)
- Conditional jumps
 - Predict next PC to be valC (destination)
 - Only correct if branch is taken
 - Typically right 60% of time
- Return instruction
 - Don't try to predict

Reading

Pipelining

Dive into systems (5.7-5.8)

5.7 pipelining: making the CPU faster

Four-staged CPU takes four cycles to execute one instruction

First cycle: fetches instruction from memory

Second cycle: decode instruction and read operands from the register file

Third cycle: The ALU execute the operation

Fourth cycle: write back the ALU result to a register

A sequence of N instructions takes $4N$ clock cycles.

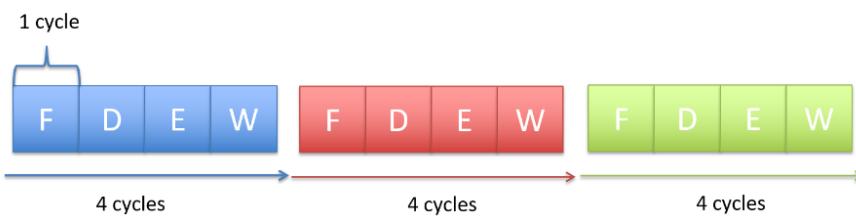


Figure 84. Executing three instructions takes 12 total cycles.

Figure results in a CPI of 4

CPI: average numbers of cycles to execute an instruction

CPU pipelining idea:

- Start the execution of the next instruction before the current instruction has completed its execution
- (if we start on instruction 1, and execute F, then when we move to D we start instruction 2 etc.)
- Executes in order

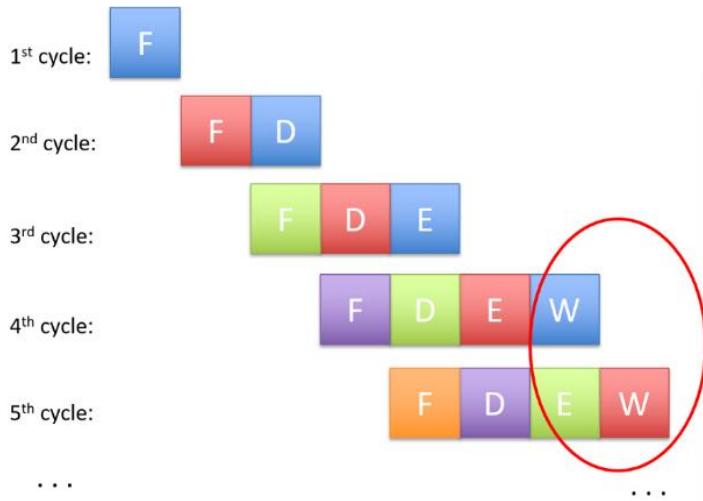


Figure 85. Pipelining: overlapping instruction execution to achieve one instruction completed per cycle. The circle indicates that the CPU has reached the steady state of completing one instruction every cycle.

Pipelining is more complex

- Complexity is almost always worth the improvement

Modern microprocessors implement pipelined execution

ADD instruction example:

To execute the load AND store between memory and register: five-stages pipeline

Fetch - decode - execute - memory - writeback

(processors may have fewer or more pipelines)

- Intel core i7 has a 14 stage pipeline

5.8 advanced pipelined instruction considerations

Fetch (F): reads an instruction from memory (pointed to by the program counter).

Decode (D): reads source registers and sets control logic.

Execute (E): executes the instruction.

Memory (M): reads from or writes to data memory.

WriteBack (W): stores a result in a destination register.

Assembly code (instructions) - each instruction operates on one or more operands (registers, memory or constant values)

- Not all requires the same number of pipeline stages to execute

Pipeline stall - when any instruction is forced to wait for another to finish executing before it can continue

5.8.1 pipelining consideration: data hazards

Data hazard - occurs when two instructions tries to access common data in a pipeline

```
MOV M[0x84], Reg1      # move value at memory address 0x84 to register Reg1
ADD 2, Reg1, Reg1      # add 2 to value in Reg1 and store result in Reg1
```

MOV M[0x84], Reg1
Add 2, Reg1, Reg1



Problem



Partial Solution: use a "bubble" (NOP)

MOV - requires 5 stages since it has to access memory

ADD - requires 4

Processor prevents the scenario by forcing every instruction to take 4 stages to execute (add gets a "no-operation" (NOP/bubble) instruction)

However: mov needs to finish writing before add can access and write

```
MOV 4, Reg2          # copy the value 4 to register Reg2
ADD Reg2, Reg2, Reg2 # compute Reg2 + Reg2, store result in Reg2
```

MOV 4, Reg2
ADD Reg2, Reg2, Reg2



Problem: ADD doesn't have the proper value of Reg2!



Solution (suboptimal): more bubbles!



Operand forwarding: read and use result from previous operation

Figure 87. The processor can reduce the damage caused by pipeline hazards by forwarding operands between instructions.

Adding more bubbles is suboptimal - stalls the pipeline

Operand forwarding - pipeline reads the result from previous operation

5.8.2 pipelining hazards: control hazards

Pipeline is optimized for instructions that occur one after another

If statements and loops can affect the performance

```

int result = *x; // x holds an int
int temp = *y; // y holds another int

if (result <= temp) {
    result = result - temp;
}
else {
    result = result + temp;
}
return result;

```

Reads integer data from 2 pointers, compares and then calculate

C code in assembly instructions:

```

MOV M[0x84], Reg1      # move value at memory address 0x84 to register Reg1
MOV M[0x88], Reg2      # move value at memory address 0x88 to register Reg2
CMP Reg1, Reg2          # compare value in Reg1 to value in Reg2
JLE L1<0x14>            # switch code execution to L1 if Reg1 less than Reg2
ADD Reg1, Reg2, Reg1    # compute Reg1 + Reg2, store result in Reg1
JMP L2<0x20>            # switch code execution to L2 (code address 0x20)
L1:
SUB Reg1, Reg2, Reg1  # compute Reg1 - Reg2, store in Reg1
L2:
RET                   # return from function

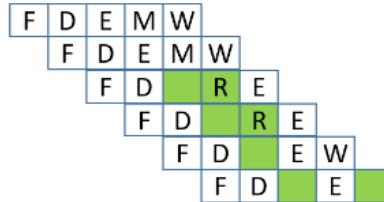
```

If statement: CMP and JLE

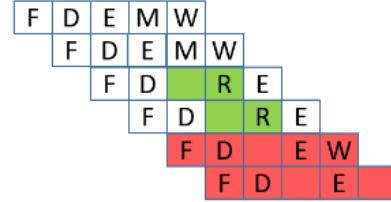
```

MOV M[0x84], Reg1
MOV M[0x88], Reg2
CMP Reg1, Reg2
JLE L1<0x14>
ADD Reg1, Reg2, Reg1
JMP L2<0x18>
L1:
    SUB Reg1, Reg2, Reg1
L2:
    RET

```



If branch is not taken...



If branch is taken, we have "junk" in the pipeline that needs to be flushed!

Control hazard occurs when pipeline encounters a branch (or conditional) instruction.

- Then it has to GUESS whether the branch will be taken
- If not then it continues the next one

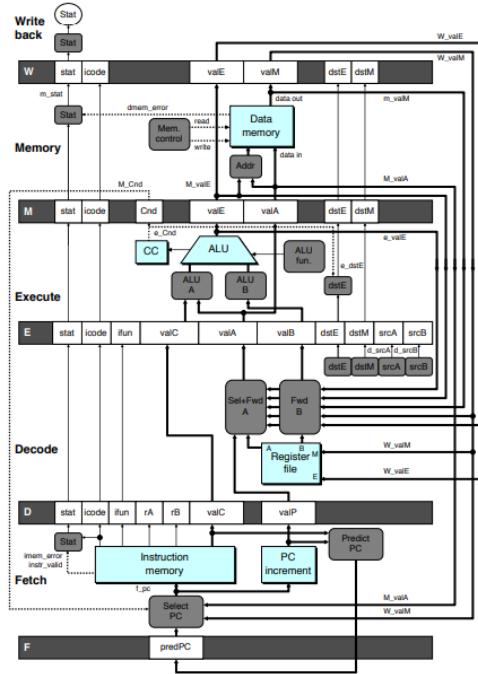
To help

Stall the pipeline - if there's a branch, then lots of bubbles are added and will eventually lead to performance hit

Branch prediction - predicts based on previous executions. Recently caused security vulnerabilities

Eager execution - CPU executes both sides of the branch- enables to continue execution, however not all code is capable (can be dangerous)

Y86-64 microprocessor 2.3



Five stage pipeline called PIPE - implements Y86-64 instruction set

PIPE has pipeline registers - enable up to 5 instructions to flow through simultaneously each in a different state

STD Is illustrated in the figure

Figure 5: Hardware structure of PIPE, the pipelined implementation to be verified. Some of the connections are not shown. (From [5, Fig. 4.52].)

STD

- Data hazards are handled by forwarding into the decode stage.
- A one-cycle stall in the decode stage is required for load/use hazards
- A three-cycle stall is required for return instruction
- Branches are predicted as taken. Up to two instructions cancelled when misprediction

FULL

- Implements the iaddq instruction

STALL

- No data forwarding
- Instructions stall in decode for up to 3 cycles whenever an instruction further down imposes data hazard

NT

- Predicts that branches will not be taken unless they are conditional
- Up to two instructions cancelled when misprediction

BTFNT

- Similar to NT
- Branches to lower addresses are predicted as being taken
- Those to higher are predicted not to be taken unless unconditional
- Up to two instructions cancelled when misprediction

LF

- Additional forward path between data memory output and pipeline register that is feeding the data memory input
- Allows forms of load/use hazards to be resolved by data forwarding rather than stalling

SW

- Simplifies register file to have 1 write port
- Multiplexer splitting the 2 sources
- Requires popq instruction executed in 2 cycles (update stack pointer - read from memory)

Introduction to intel (page 1-5 and 16-21)

VEX-prefix

Two or three-byte prefix

Designed to clean up the current and future instruction encoding

SIMD instructions

Allows processing multiple pieces of data in a single step

Floating point value in 32-bit length (single precision)

Floating point value in 64-bit length (double precision)

IEEE-753 standard defining reproducible, robust floating point operation

The older, related Intel® SSE instructions also support various signed and unsigned integer sizes, including signed and unsigned byte (B, 8-bit), word (W, 16-bit), doubleword (DW, 32-bit), quadword (QW, 64-bit), and doublequadword (DQ, 128-bit) lengths.

Bits 0–5 represent invalid operation, denormal, divide by zero, overflow, underflow, and precision, respectively

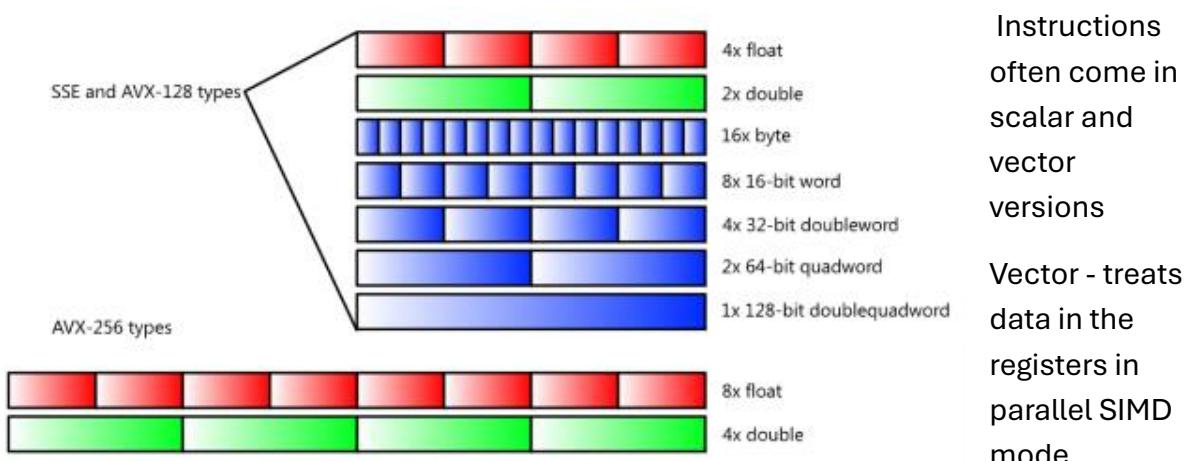


Figure 2. Intel® AVX and Intel® SSE data types

Scalar - only operates on one entry in each register

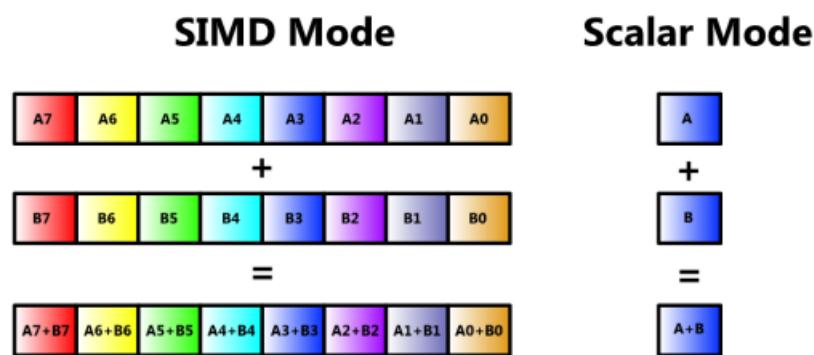


Figure 3. SIMD versus scalar operations

Data is memory aligned when the data to be operated upon as an n-byte chunk is stored on an n-byte memory boundary.

For intel@SSE operations - memory alignment required unless explicitly stated

Instruction set

Entries in square brackets ([]) are required; entries in parentheses (()) are optional.

Examples:

- (V)ADD[P/S][D/S] is the addition of packed or scalar, double or single, with eight possible forms—VADDPD, VADDPS, VADDSD, VADDSS, and versions without the leading V.
- (V)[MIN/MAX][P/S][D/S] represents 16 different instructions for a min or max of packed or scalar of double or single precision.

O = ordered, U = unordered, S = signaling, and Q = non-signaling

Ordered/unordered

- Tells whether the comparison is false or true if one operand is not a floating point number (NaN)

Signaling/non-signaling

- States whether an exception is fired when at least one operand is quiet not a number (QNaN)

Type	Flavors	Meaning
EQ	(QQ), UQ, QS, US	Equal
LT	(QS), QQ	Less than
LE	(QS), QQ	Less than or equal to
UNORD	(Q), S	Tests for unordered (NaN)
NEQ	(UQ), US, QQ, QS	Not equal
NLT	(US), UQ	Not less than
NLE	(US), UQ	Not less than or equal to
ORD	(Q), S	Tests for ordered (not NaN)
NGE	US, UQ	Not greater than or equal to

Type	Flavors	Meaning
NGT	US, UQ	Not greater than
FALSE	UQ, OS	Comparison is always false
GE	OS, OQ	Greater than or equal to
GT	OS, OQ	Greater than
TRUE	UQ, US	Comparison is always true

Finally, here are all the Intel® AVX instructions:

Arithmetic	Description
(V) [ADD/SUB/MUL/DIV] [P/S] [D/S]	Add/subtract/multiply/divide packed/scalar double/single
(V) ADDSUBPD [D/S]	Packed double/single add and subtract alternating indices
(V) DPDP [D/S]	Dot product, based on immediate mask
(V) HADDPD [D/S]	Horizontally add
(V) [MIN/MAX] [P/S] [D/S]	Min/max packed/scalar double/single
(V) MOVMSKPD [D/S]	Extract double/single sign mask
(V) PMOVMSKB	Make a mask consisting of the most significant bits
(V) MPSADBN	Multiple sum of absolute differences
(V) PABSV [B/W/D]	Packed absolute value on bytes/words,doublewords
(V) P [ADD/SUB] [B/W/D/Q]	Add/subtract packed bytes/words,doublewords,quadwords
(V) PADD [S/U] S [B/W]	Add packed signed/unsigned with saturation bytes/words
(V) PAvg [B/W]	Average packed bytes/words
(V) PCMLMULQDQ	Carry-less multiplication quadword
(V) PH [ADD/SUB] [W/D]	Packed horizontal add/subtract word/doubleword
(V) PH [ADD/SUB] SW	Packed horizontal add/subtract with saturation
(V) PHMINPOSUN	Min horizontal unsigned word and position
(V) PMADDWD	Multiply and add packed integers
(V) PMADDUBSW	Multiply unsigned bytes and signed bytes into signed words
(V) P [MIN/MAX] [S/U] [B/W/D]	Min/max of packed signed/unsigned integers
(V) PMUL [H/L] [S/U] W	Multiply packed signed/unsigned integers and store high/low result
(V) PMULHRSW	Multiply packed unsigned with round and shift

Arithmetic	Description
(V) PMULHH	Multiply packed integers and store high result
(V) PMULL [W/D]	Multiply packed integers and store low result
(V) PMUL (U) DQ	Multiply packed (un)signed doubleword integers and store quadwords
(V) PSADBN	Compute sum of absolute differences of unsigned bytes
(V) PSIGN [B/W/D]	Change the sign on each element in one operand based on the sign in the other operand
(V) PS [L/R] LDQ	Byte shift left/right amount in operand
(V) SL [L/AR/LR] [W/D/Q]	Bit shift left/arithmetic right/logical right
(V) PSUB (U) S [B/W]	Packed (un)signed subtract with (un)signed saturation
(V) RCP [P/S] S	Compute approximate reciprocal of packed/scalar single precision
(V) RSQRT [P/S] S	Compute approximate reciprocal of square root of packed/scalar single precision
(V) ROUND [P/S] [D/S]	Round packed/scalar double/single
(V) SQRT [P/S] [D/S]	Square root of packed/scalar double/single
VEZERO [ALL/UPPER]	Zero all/upper half of YMM registers

Comparison	Description
(V) CMP [P/S] [D/S]	Compare packed/scalar double/single
(V) COMIS [S/D]	Compare scalar double/single, set ZFLAGS
(V) PCMP [EQ/GT] [B/W/D/Q]	Compare packed integers for equality/greater than
(V) PCMP [E/I] STR [I/M]	Compare explicit/implicit length strings, return index/mask

Control	Description
V[LD/ST] MXCSR	Load/store MXCSR control/status register
XSAVEOPT	Save processor extended states optimized

Conversion	Description
(V) CVTx2y	Convert type <i>x</i> to type <i>y</i> , where <i>x</i> and <i>y</i> are chosen from DQ and P[D/S], [P/S]S and [P/S]D, or

| S[D/S] and SI.

Load/store	Description
VBROADCAST [SS/SD/F128]	Load with broadcast (loads single value into multiple locations)
VEXTRACTF128	Extract 128-bit floating-point values
(V) EXTRACTPS	Extract packed single precision
VINSERTF128	Insert packed floating-point values
(V) INSERTPS	Insert packed single-precision values
(V) PINSR [B/W/D/Q]	Insert integer
(V) LDDQU	Move quad unaligned integer
(V) MASKMOVQDQ	Store selected bytes of double quadword with NT Hint
VMASKMOVP [D/S]	Conditional SIMD packed load/store
(V) MOV [A/D] P [D/S]	Move aligned/unaligned packed double/single
(V) MOV [D/Q]	Move doubleword/quadword
(V) MOVDQ [A/U]	Move double to quad aligned/unaligned
(V) MOV [HL/LH] P [D/S]	Move high-to-low/low-to-high packed double/single
(V) MOV [H/L] P [D/S]	Move high/low packed double/single
(V) MOVTNT [DQ/DD/PS]	Move packed integers/doubles/singles using a non-temporal hint
(V) MOVNTQA	Move packed integers using a non-temporal hint, aligned
(V) MOVS [D/S]	Move or merge scalar double/single
(V) MOVS [H/L] DUP	Move single odd/even indexed singles
(V) PACK [U/S] SW [B/W]	Pack with unsigned/signed saturation on bytes/words
(V) PALIGNR	Byte align
(V) PEXTR [B/W/D/Q]	Extract integer
(V) PMOV [S/Z] X [B/W/D] [W/D/Q]	Packed move with sign/zero extend (only up in length, DD, DW, etc. disallowed)

Logical	Description
(V) [AND/ANON/OR] P [D/S]	Bitwise logical AND/AND NOT/OR of packed double/single values
(V) RAND (H)	Logical AND (NOT)
(V) P [OR/XOR]	Bitwise logical OR/exclusive OR

Logical	Description
(V) PTEST	Packed bit test, set zero flag if bitwise AND is all 0
(V) UCOMIS [D/S]	Unordered compare scalar doubles/singles and set EFLAGS
(V) XORP [D/S]	Bitwise logical XOR of packed double/single

Shuffle	Description
(V) BLENDP [D/S]	Blend packed double/single; selects elements based on mask
(V) BLENDVP [D/S]	Blend values
(V) MOVDQUF	Copies even values to all values
(V) PBLENDB	Variable blend packed bytes
(V) PBLENDD	Blend packed words
VPERMILP [D/S]	Permute double/single values
VPERM2F128	Permute floating-point values
(V) PSHUF [B/D]	Shuffle packed bytes/doublewords based on immediate value
(V) PSHUF [H/L] N	Shuffle packed high/low words
(V) FUNPCK [H/L] [BM/WD/DQ/QDQ]	Unpack high/low data
(V) SHUFF [D/S]	Shuffle packed double/single
(V) UNPCK [H/L] P [D/S]	Unpack and interleave packed/scalar doubles/singles

AES	Description
AESENC/AESENCLAST	Perform one round of AES encryption
AESDEC/AESDECLAST	Perform one round of AES decryption
AESIMC	Perform the AES InvMixColumn transformation
AESKEYGENASSIST	AES Round Key Generation Assist

Future Instructions	Description
[RD/NR] [F/G] SEASK	Read/write FS/GS register
RDRAND	Read random number (into r16, r32, r64)
VCVTPH2PS	Convert 16-bit floats to single precision floating-point values
VCVTPS2PH	Convert single-precision values to 16-bit floating-point values

FMA	Each $\{\}$ is the string 132 or 213 or 231, giving the order the operands A,B,C are used in: 132 is A=AC+B 213 is A=AB+C 231 is A=BC+A
VFMADD [z] [P/S] [D/S]	Fused multiply add A = r1 * r2 + r3 for packed/scalar of double/single
VFMADDSUB [z] P [D/S]	Fused multiply alternating add/subtract of packed double/single A = r1 * r2 + r3 for odd index, A = r1 * r2-r3 for even
VFMSUBADD [z] P [D/S]	Fused multiply alternating subtract/add of packed double/single A = r1 * r2-r3 for odd index, A = r1 * r2+r3 for even
VFMSub [z] [P/S] [D/S]	Fused multiply subtract A = r1 * r2-r3 of packed/scalar double/single
VFNMAAdd [z] [P/S] [D/S]	Fused negative multiply add of packed/scalar double/single A = - r1 * r2+r3
VFNMSub [z] [P/S] [D/S]	Fused negative multiply subtract of packed/scalar double/single A = - r1 * r2-r3

Multicore

Dive into systems (5.9)

Looking ahead: CPU's today

CPU pipelining is 1 example of **instruction-level parallelism (ILP)**

IPC - average number of Instructions per cycle

- Commonly used to describe microarchitectures performance
- A large IPC indicates that the processor achieves a high sustained degree of simultaneous instruction execution

Transistors are building blocks for all circuitry

- Implement the storage circuits used in a CPU register and in fast on-chip cache memory

Moore's law

- The number of transistors per integrated circuit doubles about every two years

Means that computer architects can design a new chip with twice as much space (roughly doubling its power)

5.9.1 instruction-level parallelism

Support parallel execution of a single programs instructions on a single processor.

ILP techniques are transparent to the programmer

Vector processor

- Architecture that implements ILP through special vector instructions that take one-dimensional arrays(vectors) of data as their operands
- Instructions are executed in parallel by a vector processor on multiple execution units
- Each unit performs an arithmetic operation on single elements from the vector operand
- Appear primarily in accelerator devices (graphics processing units - GPU)

Superscalar

- Single processor with multiple execution units AND pipelines
- Fetches a set of instructions from a sequential programs instruction stream, then breaks them up into multiple independent streams of instructions
- Then executed in parallel by its execution units.
- Superscalar processor is out-of-order processor or one that executes out of the order they appear in
- Requires identifying sequences without dependencies so they can safely be executed in parallel.
- Contains functionality to dynamically create the multiple streams of independent instructions to feed through multiple units.
- Functionality must perform dependency analysis to ensure correct ordering for instructions that have dependency

Very long instruction word (VLIW)

- Compiler is responsible for constructing multiple independent instruction streams executed in parallel
- Compiler analyses the instructions to statically construct a VLIW instruction (consists of multiple instructions, one from each independent stream)
- Leads to simpler processor design than superscalar (does not need to perform dependency analysis to construct multiple streams)
- Just needs added circuitry to fetch the next VLIW instruction.
- Requires specialized compilers to achieve good performance

Dependencies limit the ability to keep all pipelines full

5.9.2 multicore and hardware multithreading

Moore's law could only be followed until the early 2000's

- Then the clock speed could no longer increase without the power consumption increasing

Hardware multithreading

- Single processor design
- Thread: independent stream of execution

Can be implemented on either scalar or superscalar type microprocessors

Minimum

- Needs to support fetching instructions from multiple separate instruction streams
- Have separate register sets for each threads stream

Explicitly multithreaded - unlike superscalar, execution streams are independently scheduled by the system to run a separate logical sequence of instructions.

Hardware multithreaded microarchitecture based on superscalar processors have multiple pipelines and units.

- Can then execute instructions from several hardware threads simultaneously (in parallel)
- Giving an IPC value greater than 1

Multithreaded architectures based on simple scalar implement Interleaved multithreading

- Typically share a pipeline and always share the processor's single ALU
- The CPU switches between threads on the ALU
- Cannot achieve IPC greater than 1

Hardware threading based on superscalar is called Simultaneous multithreading (SMT)

- SMT is often used to refer to both types of hardware multithreading

Multicore processors

- Contain multiple complete CPU cores
- Each core is independently scheduled by the OS (each core is a FULL CPU core)
- Each core could be scalar, superscalar or hardware multithreaded

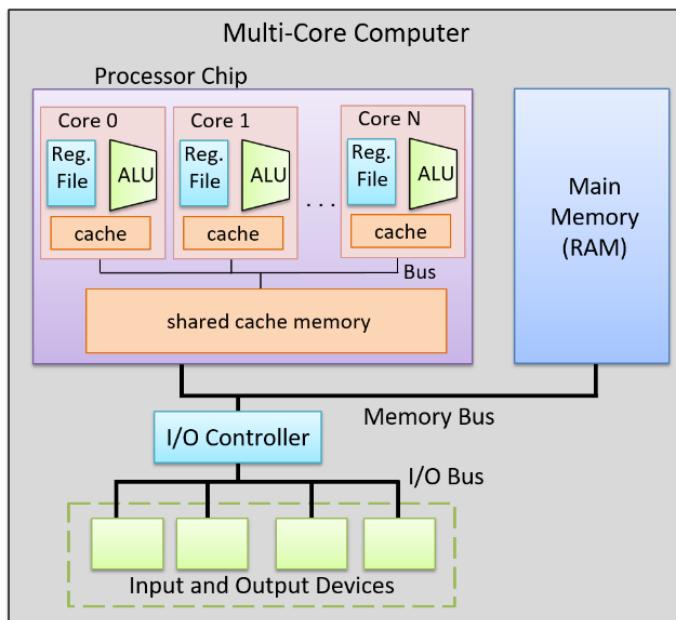


Figure 90. A computer with a multicore processor. The processor contains multiple complete CPU cores, each with its own private cache memory. The cores communicate with each and share a larger shared cached memory via on-chip buses.

Multicore microprocessor design

- Primary way where processor architectures can keep pace with Moore's law without increasing the clock rate
- Can run several sequential programs

Heterogeneous computing

Dive into systems (15.1)

Heterogeneous computing: hardware accelerators, GPGPU computing, and CUDA

Heterogeneous computing is computing using multiple, different processing units found in a computer.

Typically means support for parallel computing using the computer's CPU cores and one or more of its accelerator units such as GPU or FPGA's (field programmable gate arrays).

Common to implement to large, data-intensive and computation-intensive problems.

15.1.1 hardware accelerators

Computers have special-purpose hardware like

- FPGA's
- Cell processors
- GPU's

FPGA's

Integrated circuit - consists of gates, memory and interconnection components

They are reprogrammable - can be reconfigured to implement specific functionality

Typically require less power than a full CPU

Low-latency devices that can be directly connected to a system buses.

Reprogramming takes a long time and use is limited

GPU's and cell processors

Cell processor is a multicore processor

- Has one general purpose processor and multiple co-processors
- Co-processors are specialized to accelerate a specific type of computation

GPU's perform computer graphics computations

- Operate on image data to enable high-speed graphic rendering and image processing.
- Writes its result to a frame buffer, that delivers data to the display

15.1.2. GPU architecture overview

The hardware execution model implemented by GPU's is: single instruction/multiple thread (SIMT), a variation of SIMD.

- A single instruction is executed in lockstep by multiple threads running on the processing units.
- The total number of threads can be larger than the total number of processing units.

Lockstep

- Each thread in a warp executes the same instruction each cycle but on different data
- Example - if an application is changing a colour image to greyscale, each thread in a warp executes the same sequence of instructions at the same time to set a pixel's RGB value to its grayscale equivalent. Each thread executes these instructions on a different pixel data value, resulting in different pixels being updated in parallel.

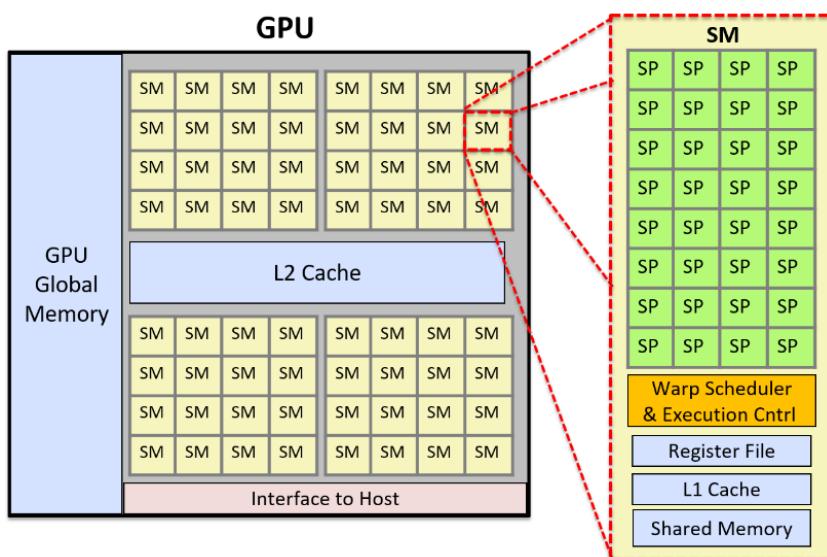


Figure 216. An example of a simplified GPU architecture with 2,048 cores. This shows the GPU divided into 64 SM units, and the details of one SM consisting of 32 SP cores. The SM's warp scheduler schedules thread warps on its SPs. A warp of threads executes in lockstep on the SP cores.

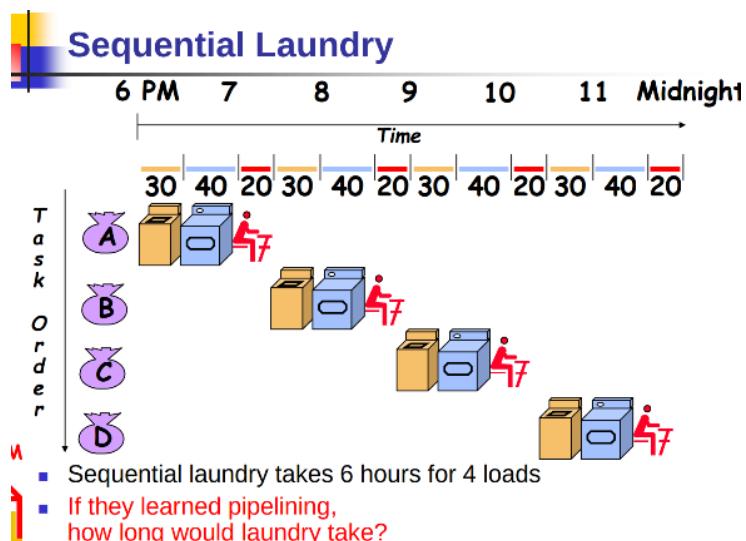
Slides

Taxonomy of parallel architectures

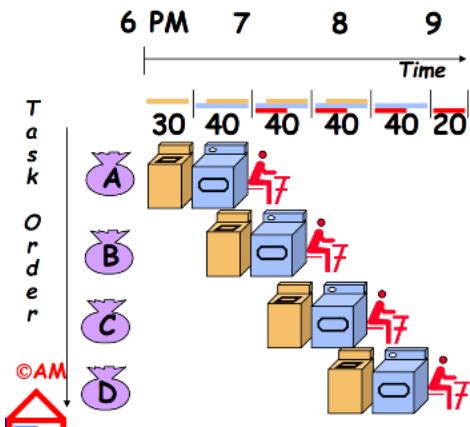
	Horizontal	Vertical
ILP	Superscalar / VLIW	Pipelined
TLP	Multi-core SMT	Interleaved / switch-on-event multithreading
DLP	SIMD / SIMT	Vector / temporal SIMD

Pipelining

Pipelined is faster than sequential



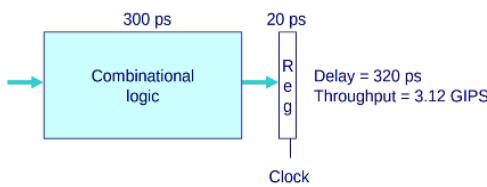
Pipelining Lessons



Billions of instructions ==> throughput is what matters!

- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate is limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" reduce speedup

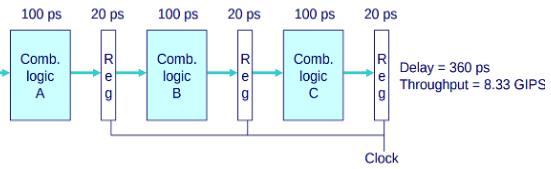
Computational Example



System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle of at least 320 ps

3-Way Pipelined Version

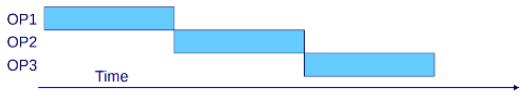


System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
 - Begin new operation every 120 ps
- Overall latency increases
 - 360 ps from start to finish

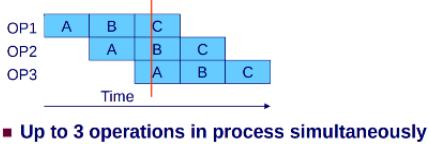
Pipeline Diagrams

Unpipelined



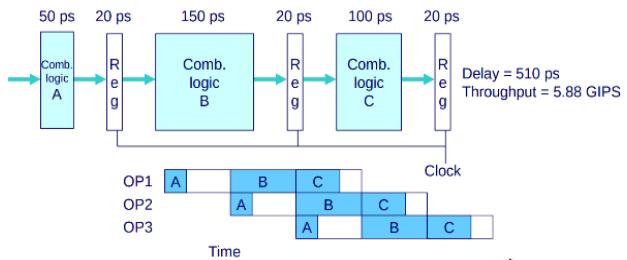
- Cannot start new operation until previous one completes

3-Way Pipelined



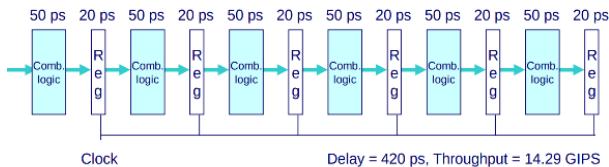
- Up to 3 operations in process simultaneously

Limitations: Nonuniform Delays



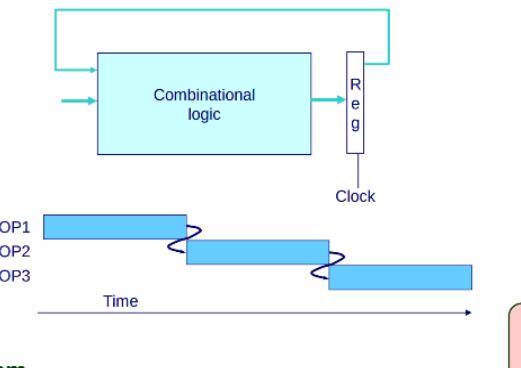
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
 - 1-stage pipeline: 6.25%
 - 3-stage pipeline: 16.67%
 - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Data Dependencies



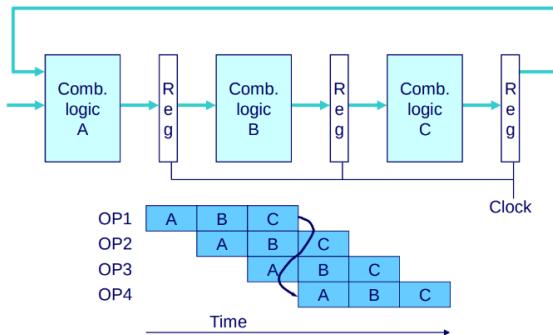
System

- Each operation depends on result from preceding one

THE inhibitor of parallelization.

Can be mitigated in SW and HW. Today: HW

Data Hazards



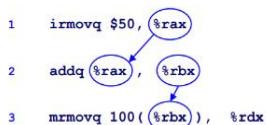
- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

- 14 -

Important criteria:
pipelining ==> "same behavior, just faster".

CS:APP

Data Dependencies in Processors



- Result from one instruction used as operand for another
 - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
 - Get correct results
 - Minimize performance impact

recall:

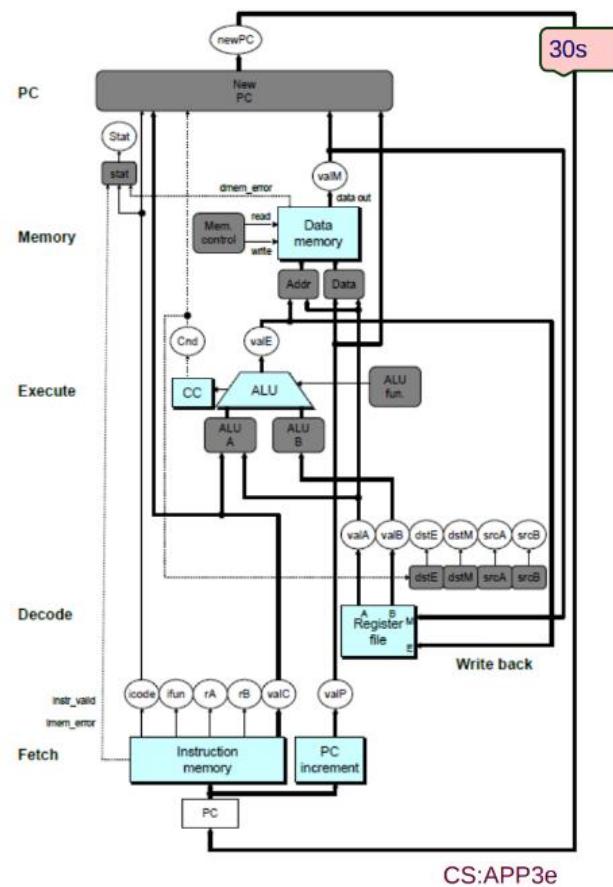
SEQ Hardware

- Stages occur in sequence
- One operation in process at a time

note how this computes what's mentioned in the **semantics**.

Now: let's pipeline this.
issue: branching (jump, ret);
which instruction is next?
predict next instruction
(speculative execution).
let's be optimistic for now
(address misprediction later)

- 17 -



SEQ+ Hardware

- Still sequential implementation
- Reorder PC stage to put at beginning

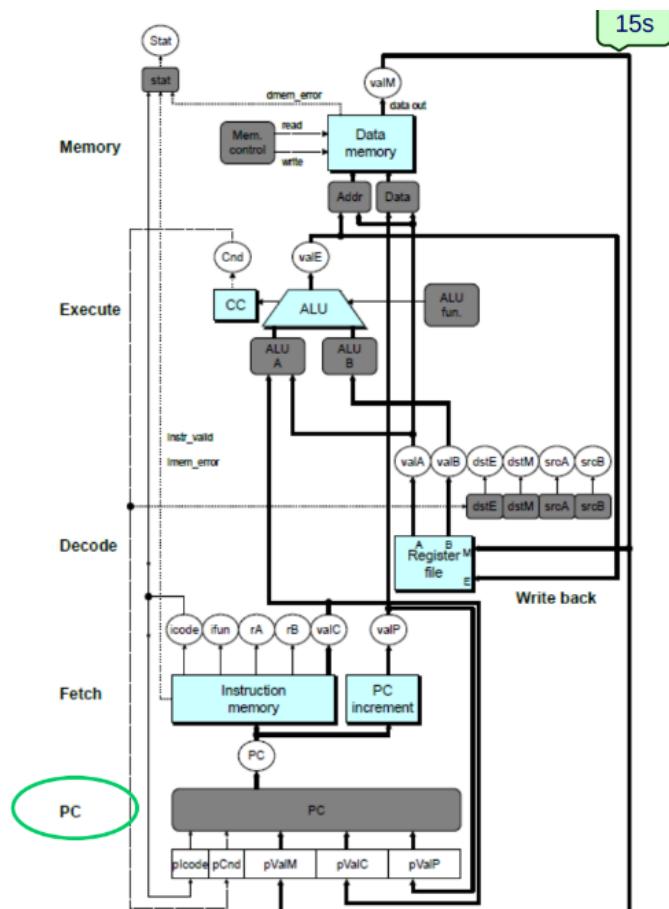
PC Stage

- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information

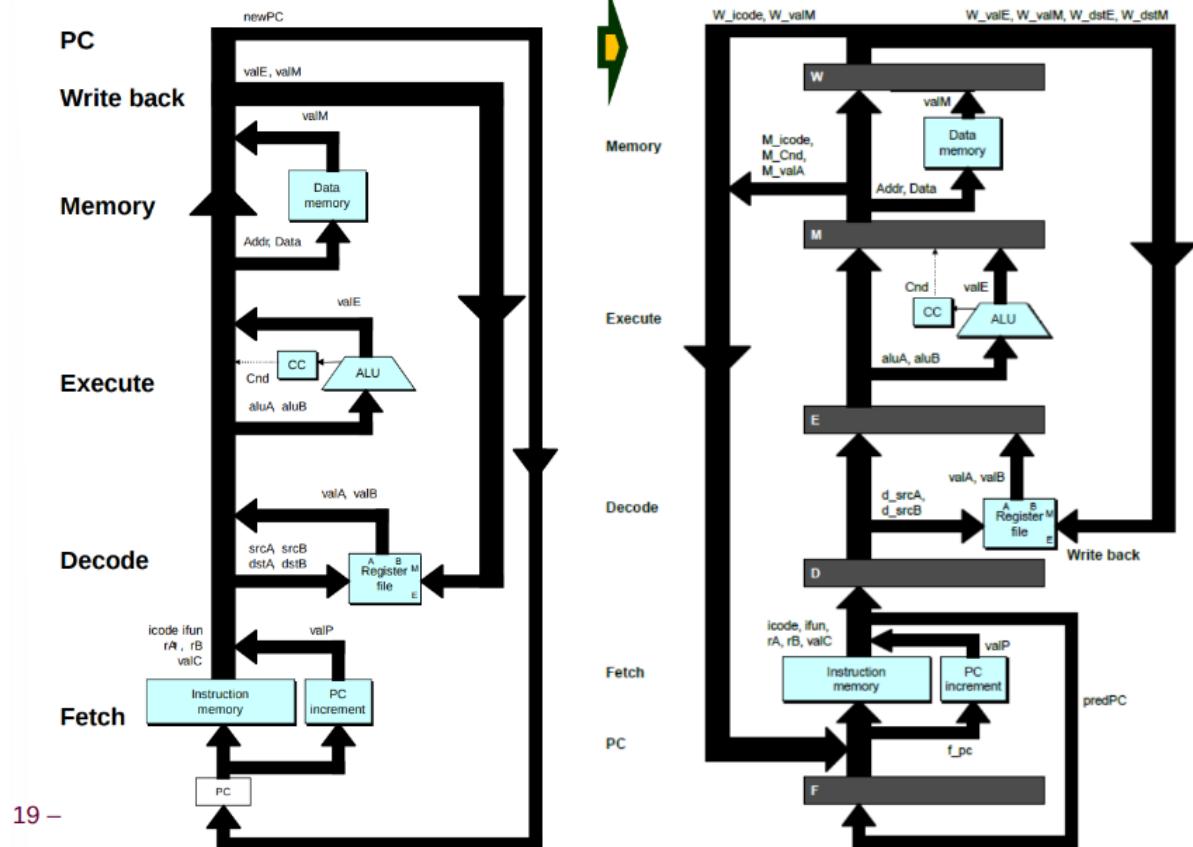
- 18 -



Adding Pipeline Registers

at each stage

15s



Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

- Operate ALU

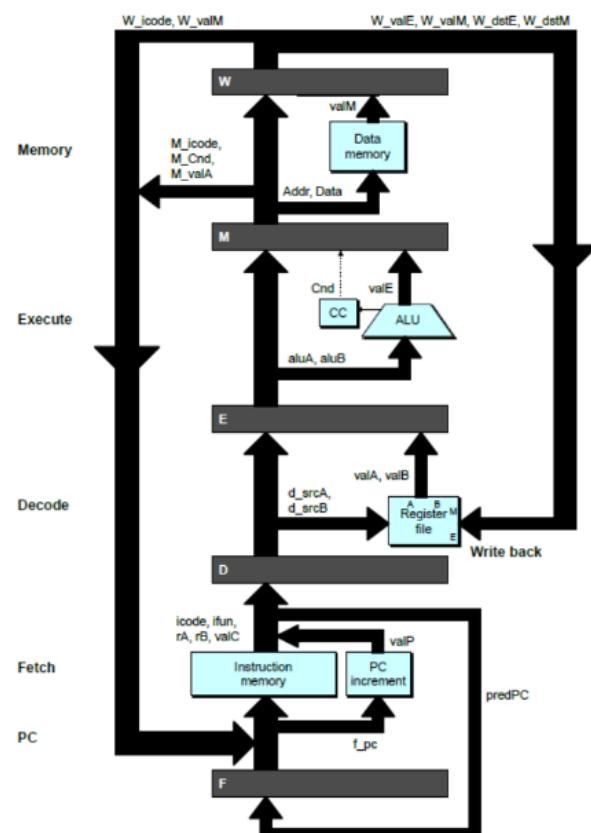
Memory

- Read or write data memory

Write Back

- Update register file

- 20 -



Signal Naming Conventions

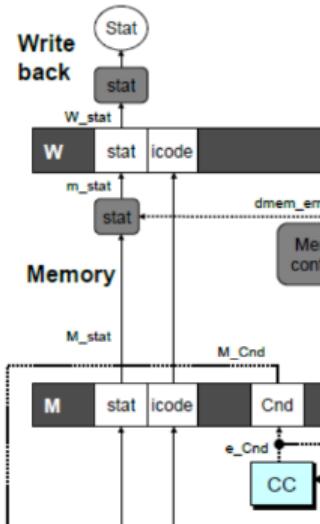
S_Field

- Value of Field held in stage S pipeline register

s_Field

- Value of Field computed in stage S

sharp edge: store
rounded edge: logic



in its glory (w/ combinational logic drawn):

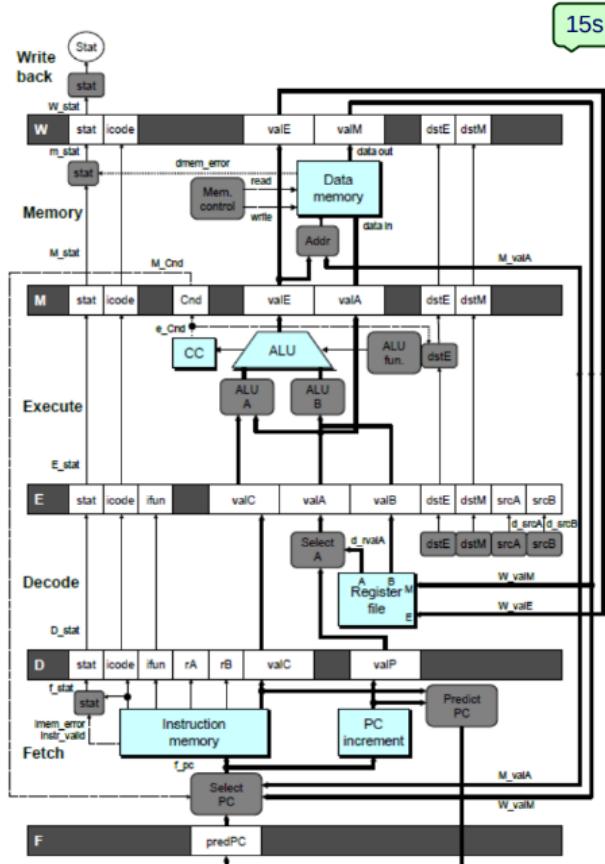
PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution

Forward (Upward) Paths

- Values passed from one stage to next
- Cannot jump past stages
 - e.g., valC passes through decode

behaves as explained in the previous slide



- 21 -

highlight:

Feedback Paths

Predicted PC

- Guess value of next PC

Branch information

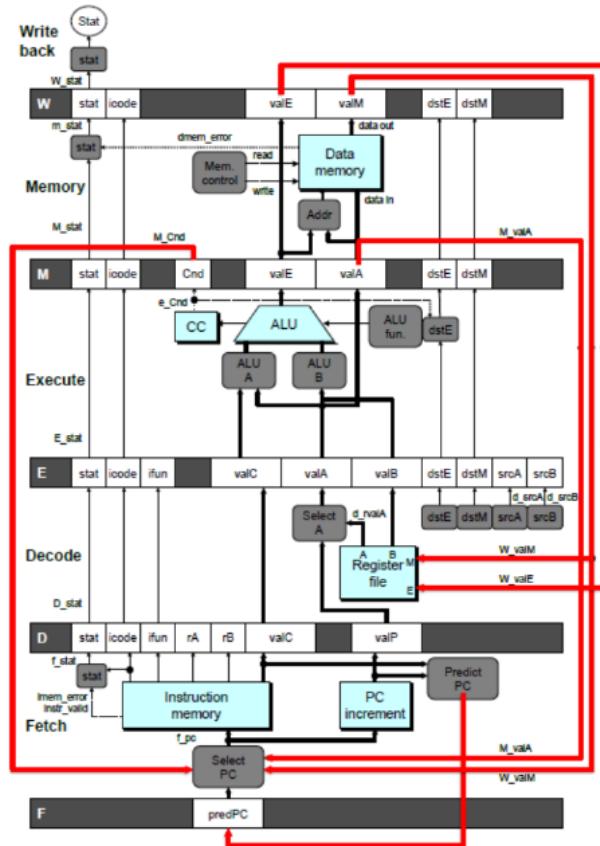
- Jump taken/not-taken
- Fall-through or target address

Return point

- Read from memory

Register updates

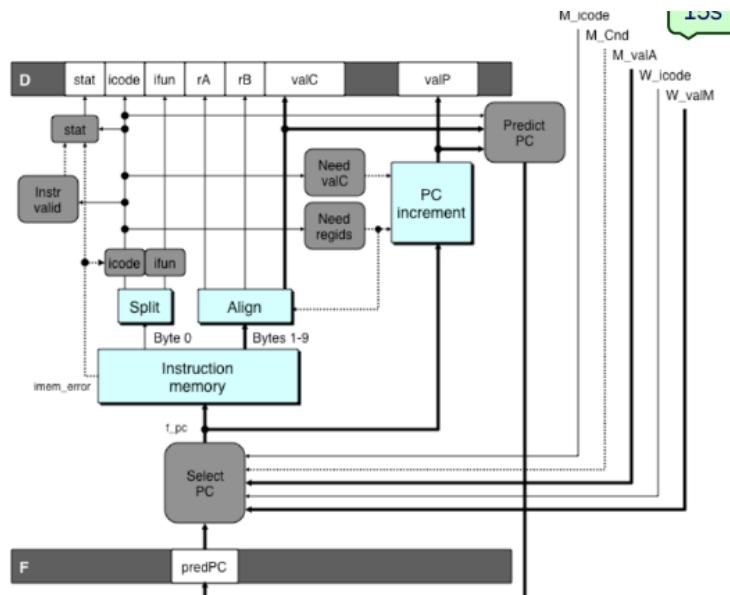
- To register file write ports



22

Predicting the PC

detailed view of the fetch-logic



- Start fetch of new instruction after current one has completed fetch stage
 - Not enough time to reliably determine next instruction
- Guess which instruction will follow
 - Recover if prediction was incorrect

remainder of PIPE slides
are about this
branch prediction & recovery.

23 –

Our Prediction Strategy

Instructions that Don't Transfer Control

- Predict next PC to be valP (increment)
- Always reliable

remember semantics

Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

remember semantics

Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
 - Typically right 60% of time

cond. assumed true

Return Instruction

- Don't try to predict

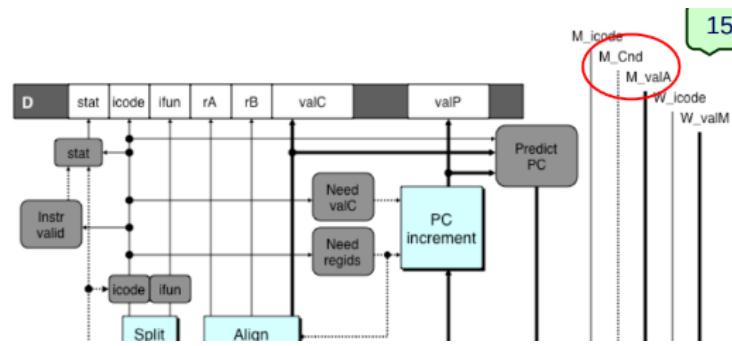
have nothing to go on
in our registers; target
addr. is on the stack

- 24 -

recovery? (later)

CS:APP3e

Recovering from PC Misprediction



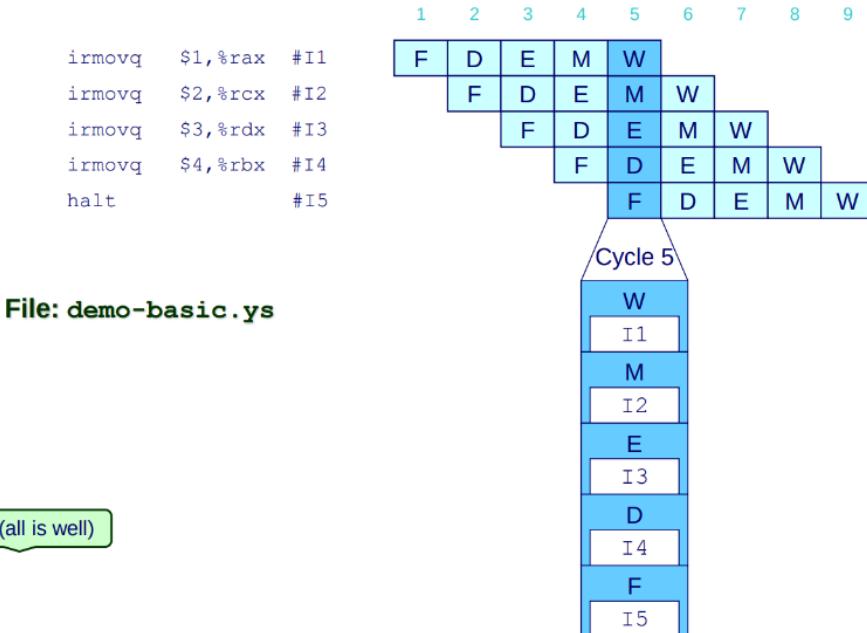
- Mispredicted Jump
 - Will see branch condition flag once instruction reaches M stage
 - Can get fall-through PC from valA (value M_valA)
- Return Instruction
 - Will get return PC when ret reaches write-back (W) stage (W_valM)

-

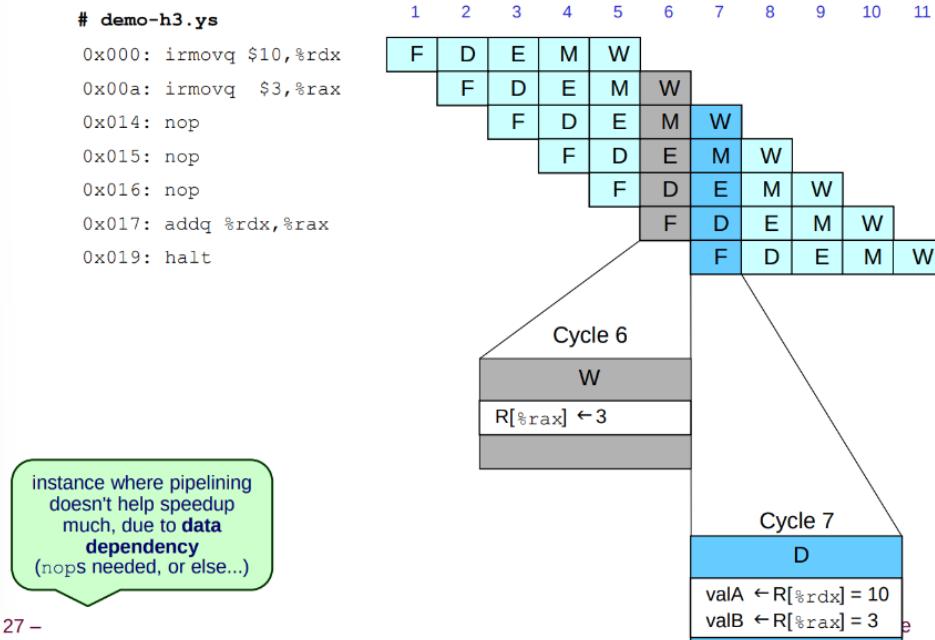
should give some idea of cost of misprediction.

CS:APP3e

Pipeline Demonstration



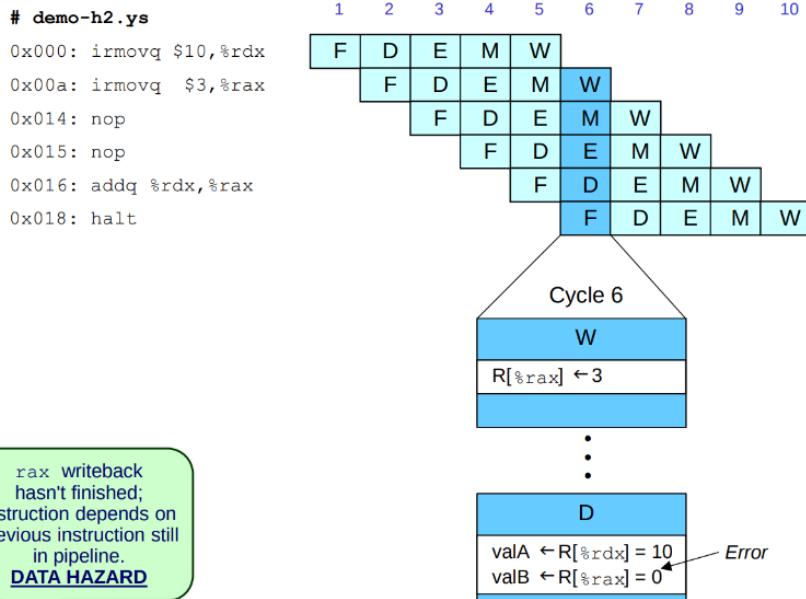
Data Dependencies: 3 Nop's



- 27 -

Data Dependencies: 2 Nop's

15s

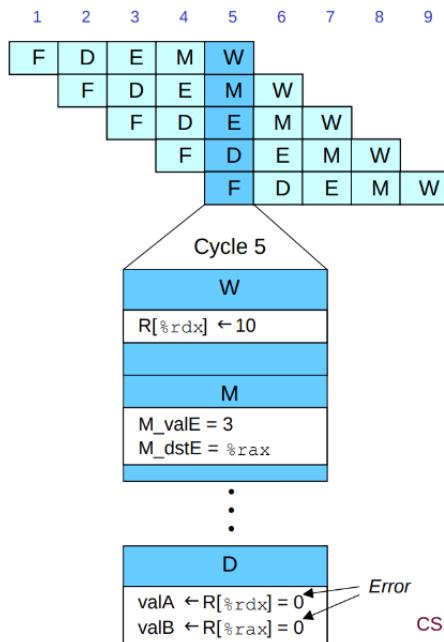


28 -

Data Dependencies: 1 Nop

5s

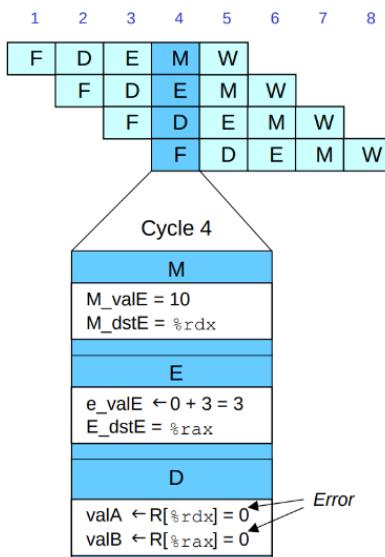
```
# demo-h1.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



CS:APP3e

Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Branch Misprediction Example

demo-j.ys

```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target (Should not execute)
0x023: irmovq $4, %rcx  # Should not execute
0x02d: irmovq $5, %rdx  # Should not execute
```

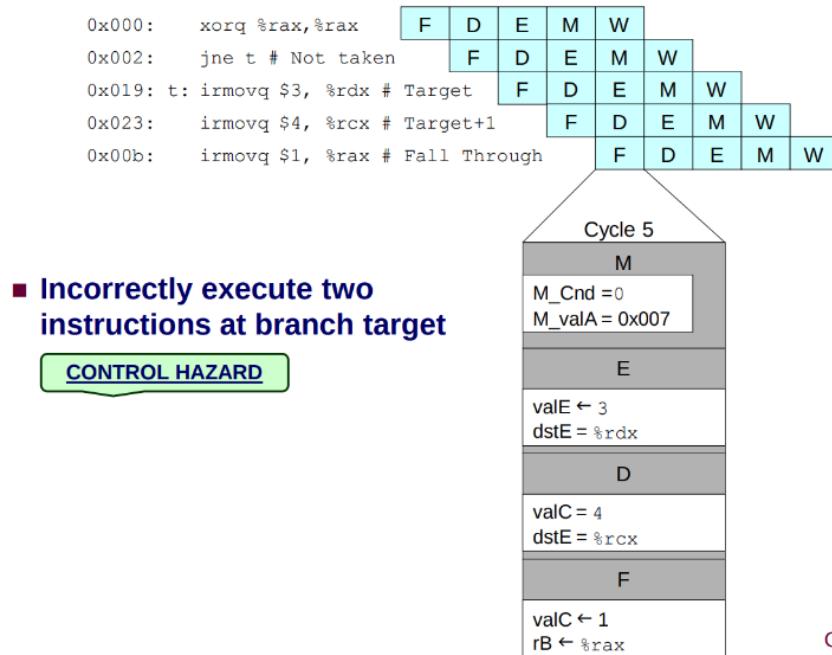
- Should only execute first 8 instructions

another instance where
pipelining doesn't help
speedup:
branch misprediction
(jump)

31 -

CS:APP3e

Branch Misprediction Trace



Return Example

demo-ret.ys

```
0x000:    irmovq Stack,%rsp      # Initialize stack pointer
0x00a:    nop                  # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p              # Procedure call
0x016:    irmovq $5,%rsi      # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p:    nop              # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax      # Should not be executed
0x02e:    irmovq $2,%rcx      # Should not be executed
0x038:    irmovq $3,%rdx      # Should not be executed
0x042:    irmovq $4,%rbx      # Should not be executed
0x100: .pos 0x100
0x100: Stack:                # Initial stack pointer
```

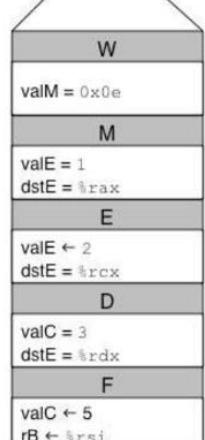
- Require lots of nops to avoid data hazards

another instance where
pipelining doesn't help
speedup:
branch misprediction
(return)

Incorrect Return Example

demo-ret

```
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



- Incorrectly execute 3 instructions following ret

Pipeline Summary, so far

Concept

- Break instruction execution into 5 stages
- Run instructions through in pipelined mode

Limitations

- Can't handle dependencies between instructions when instructions follow too closely
- Data dependencies
 - One instruction writes register, later one reads it
- Control dependency
 - Instruction sets PC in way that pipeline did not predict correctly
 - Mispredicted branch and return

How to handle these? coming right up...

spoiler: dynamically
inject the right number of
nops

Fixing the Pipeline

Make the pipelined processor work!

Data Hazards

- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

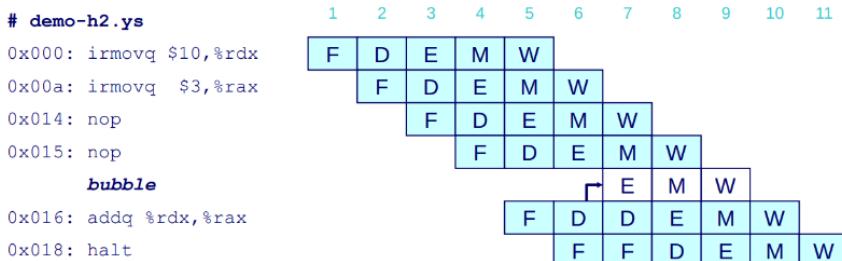
Control Hazards

- Mispredict conditional branch
 - Our design predicts all branches as being taken
 - Naïve pipeline executes two extra instructions
- Getting return address for `ret` instruction
 - Naïve pipeline executes three extra instructions

Making Sure It Really Works

- What if multiple special cases happen simultaneously?

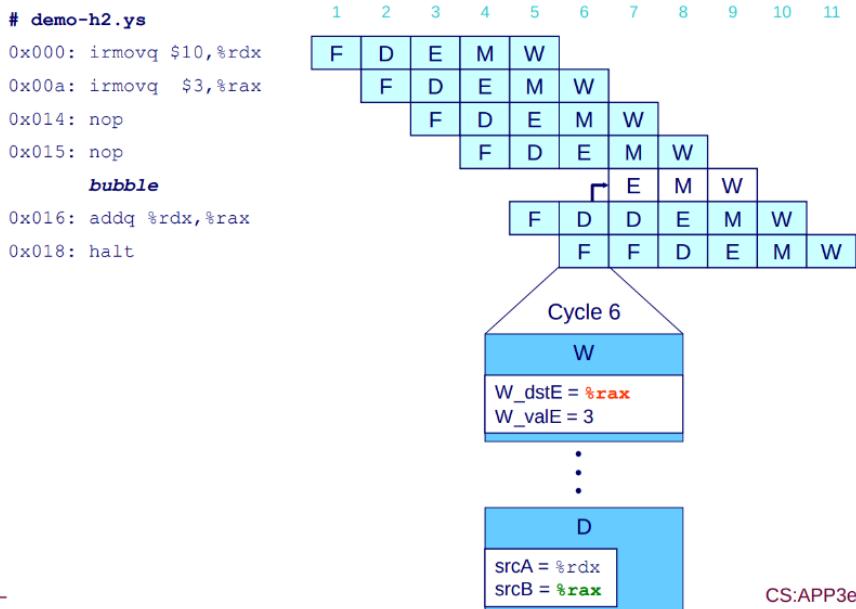
Stalling for Data Dependencies



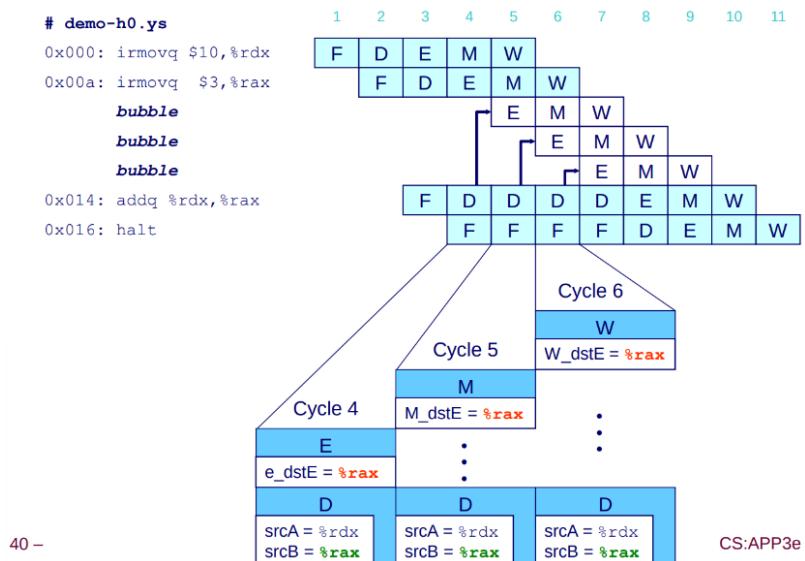
- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

(a bubble is such a dynamically injected nop instruction)

Detecting Stall Condition



Stalling X3



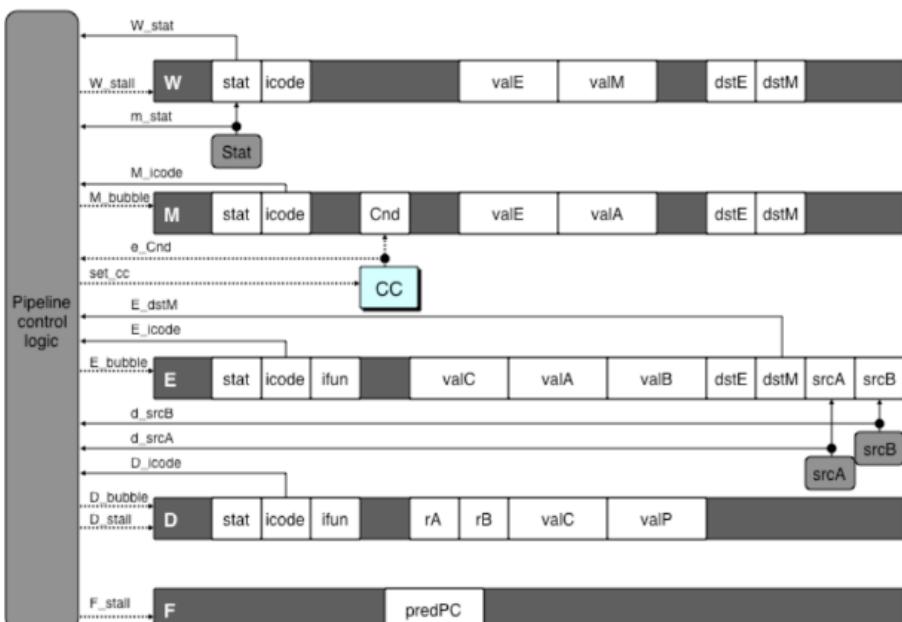
What Happens When Stalling?

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8	
Write Back	<i>bubble</i>
Memory	<i>bubble</i>
Execute	0x014: addq %rdx,%rax
Decode	0x016: halt
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
 - Like dynamically generated nop's
 - Move through later stages

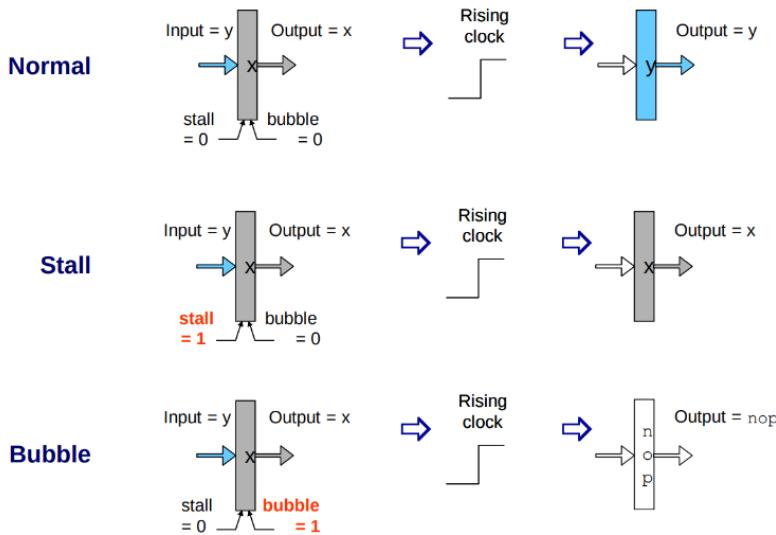
Implementing Stalling



Pipeline Control

- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

Pipeline Register Modes



Data Forwarding

Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
 - Needs to be in register file at start of stage

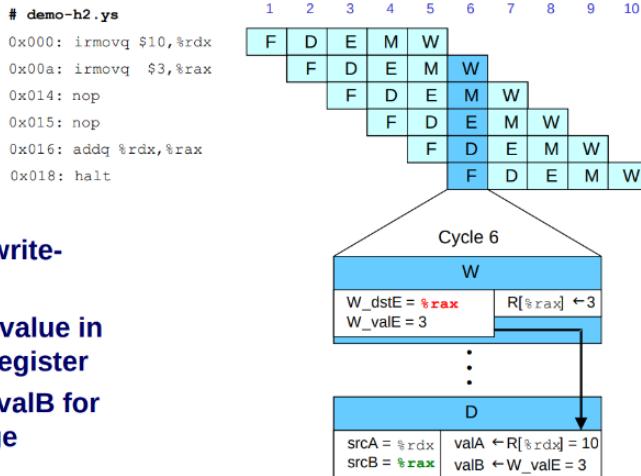
Observation

- Value generated is already in execute or memory stage

Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

Data Forwarding Example



implemented by:

Bypass Paths

Decode Stage

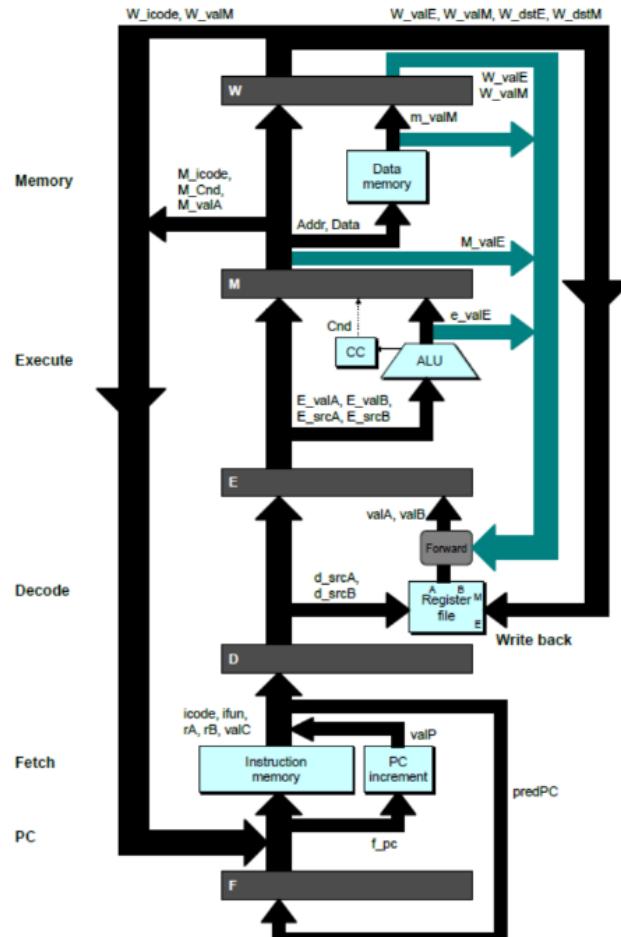
- Forwarding logic selects valA and valB
- Normally from register file
- Forwarding: get valA or valB from later pipeline stage

Forwarding Sources

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM

& logic

- 46 -



Data Forwarding Example #2

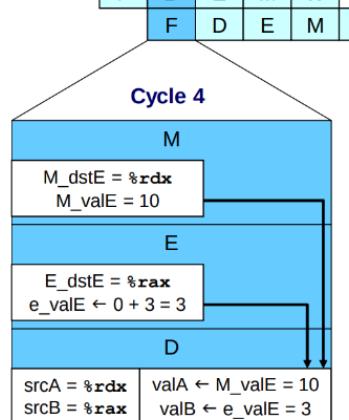
# demo-h0.ys	1	2	3	4	5	6	7	8
0x000: irmovq \$10,%rdx	F	D	E	M	W			
0x00a: irmovq \$3,%rax		F	D	E	M	W		
0x014: addq %rdx,%rax			F	D	E	M	W	
0x016: halt				F	D	E	M	W

Register %rdx

- Generated by ALU during previous cycle
- Forward from memory as valA

Register %rax

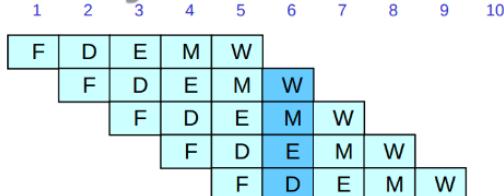
- Value just generated by ALU
- Forward from execute as valB



Forwarding Priority

demo-priority.ys

```
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```

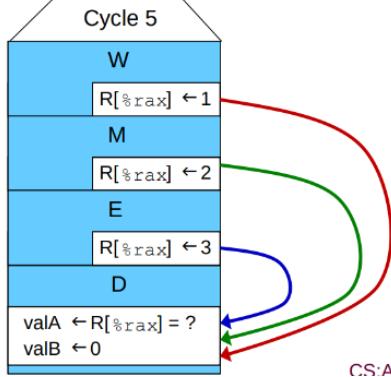


Multiple Forwarding Choices

- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

- DR -

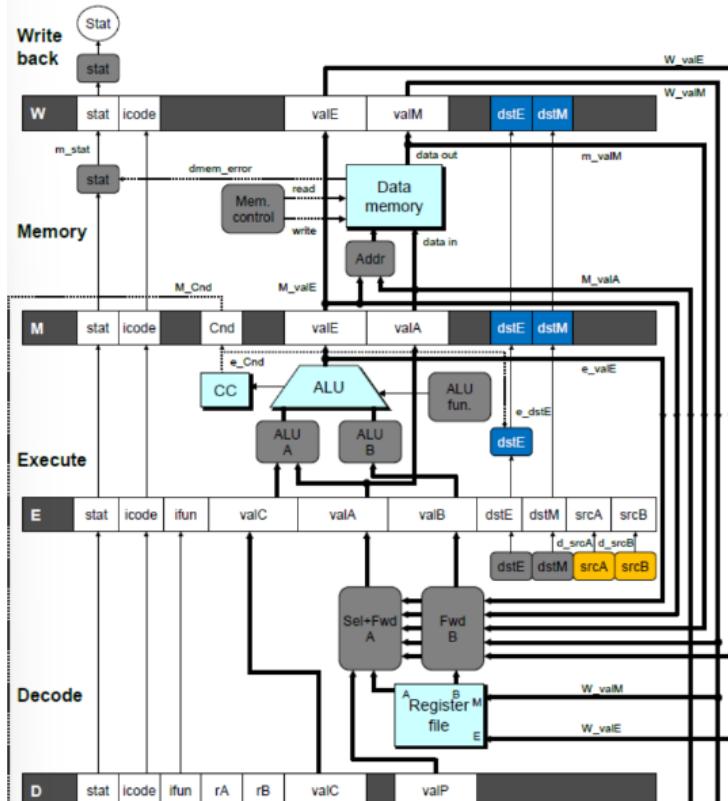
==> most up-to-date value



Implementing Forwarding

- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage

15s



Limitation of Forwarding

```
# demo-luh.ys          1 2 3 4 5 6 7 8 9 10 11
0x000: irmovq $128,%rdx   F D E M W
0x00a: irmovq $3,%rcx    F D E M W
0x014: rmmovq %rcx, 0(%rdx) F D E M W
0x01e: irmovq $10,%rbx   F D E M W
0x028: mrmovq 0(%rdx),%rax # Load %rax F D E M W
0x032: addq %rbx,%rax # Use %rax F D E M W
0x034: halt             F D E M W
```

Load-use dependency

- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8

mrmovq read from memory hasn't happened yet; cannot forward it.

— 51 —

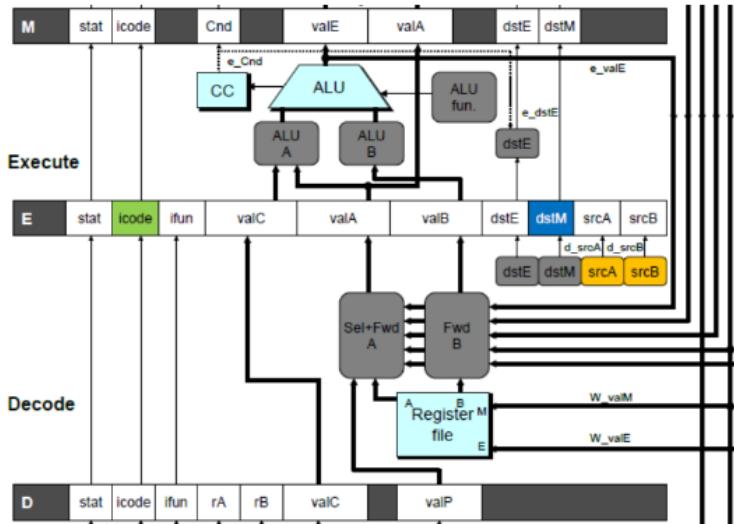
Avoiding Load/Use Hazard

```
# demo-ruh.ys          1 2 3 4 5 6 7 8 9 10 11 12
0x000: irmovq $128,%rdx   F D E M W
0x00a: irmovq $3,%rcx    F D E M W
0x014: rmmovq %rcx, 0(%rdx) F D E M W
0x01e: irmovq $10,%rbx   F D E M W
0x028: mrmovq 0(%rdx),%rax # Load %rax F D E M W
      bubble
0x032: addq %rbx,%rax # Use %rax F D D E M W
0x034: halt             F F D E M W
```

- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

52 —

Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	<code>E_icode in { TMRRMOVO, IPOPOQ } && E_dstM in { d_srcA, d_srcB }</code>

Control for Load/Use Hazard

```
# demo-luh.ys
      1   2   3   4   5   6   7   8   9   10  11  12
0x000: irmovq $128,%rdx [F D E M W]
0x00a: irmovq $3,%rcx [F D E M W]
0x014: rmrmovq %rcx, 0(%rdx) [F D E M W]
0x01e: irmovq $10,%ebx [F D E M W]
0x028: mrmovq 0(%rdx),%rax # Load %rax [F D E M W]
      bubble [F D E M W]
0x032: addq %ebx,%rax # Use %rax [F D D E M W]
0x034: halt [F F D E M W]
```

- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

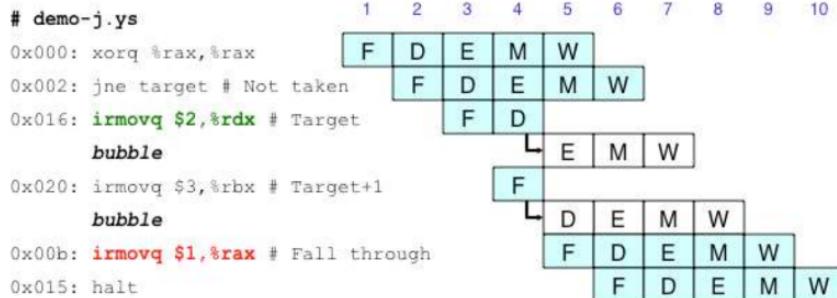
Branch Misprediction Example

demo-j.ys

```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target
0x023: irmovq $4, %rcx # Should not execute
0x02d: irmovq $5, %rdx # Should not execute
```

- Should only execute first 8 instructions

Handling Misprediction



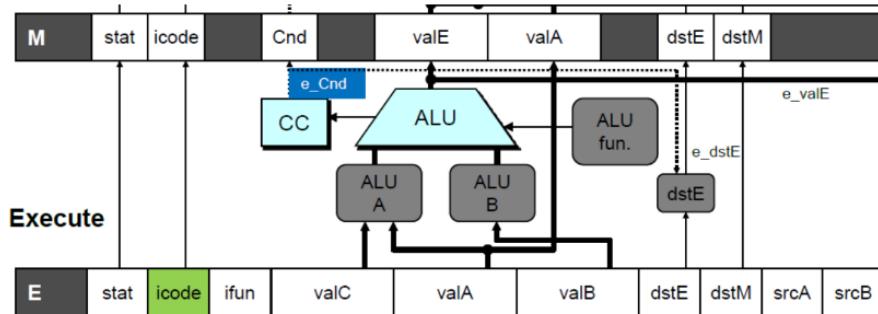
Predict branch as taken

- Fetch 2 instructions at target

Cancel when mispredicted

- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects yet (bad Ws haven't happened)

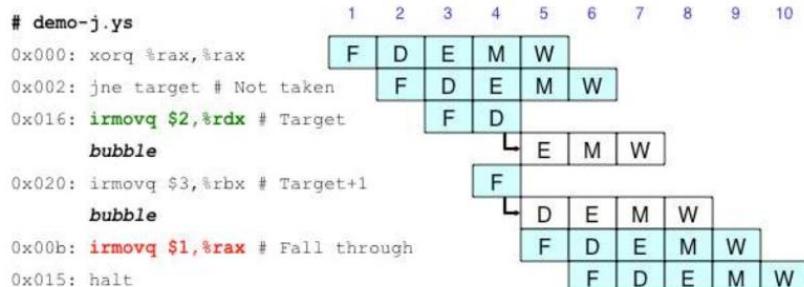
Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	$E_icode = IJXX \& !e_Cnd$

it's a jump, and it's false
(we predict true)

Control for Misprediction



Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

Return Example

demo-retb.ys

```

# demo-j.ys
0x000: xorq %rax,%rax      # Intialize stack pointer
0x00a: call p                # Procedure call
0x013: irmovq $5,%rsi       # Return point
0x01d: halt
0x020: .pos 0x20
0x020: p: irmovq $-1,%rdi   # procedure
0x02a:    ret
0x02b:    irmovq $1,%rax    # Should not be executed
0x035:    irmovq $2,%rcx    # Should not be executed
0x03f:    irmovq $3,%rdx    # Should not be executed
0x049:    irmovq $4,%rbx    # Should not be executed
0x100: .pos 0x100
0x100: Stack:             # Stack: Stack pointer

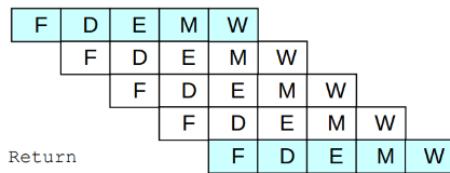
```

- Previously executed three additional instructions

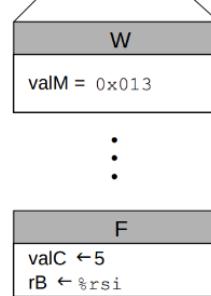
Correct Return Example

demo-retb

```
0x026:    ret
bubble
bubble
bubble
0x013:    irmovq $5,%rsi # Return
```



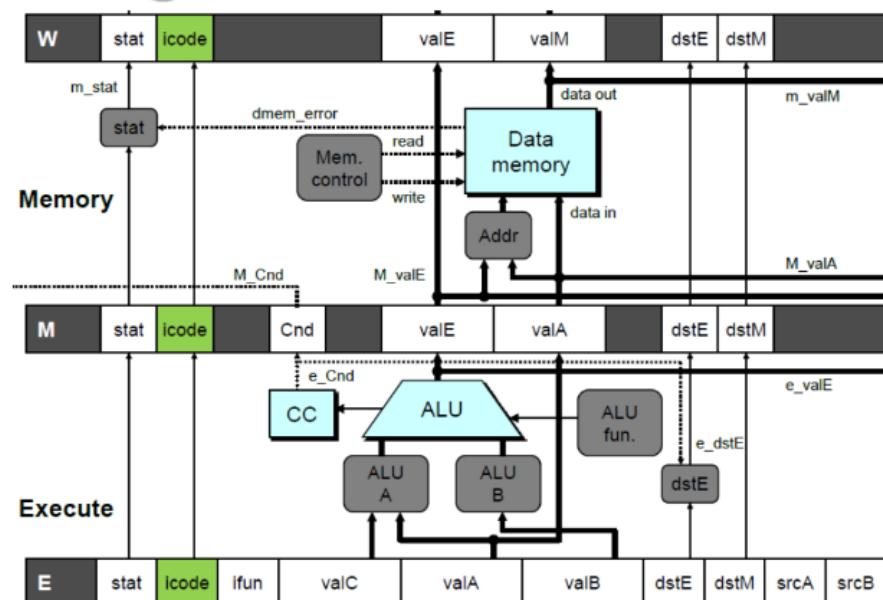
- As `ret` passes through pipeline, stall at fetch stage
 - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage



basically stop pipelining until we've read the return address.

60 -

Detecting Return

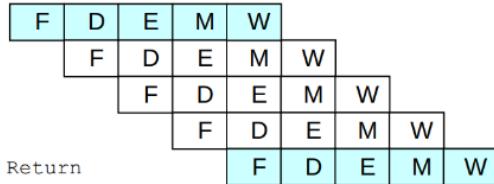


Condition	Trigger
Processing <code>ret</code>	<code>IRET</code> in { <code>D_icode</code> , <code>E_icode</code> , <code>M_icode</code> }

Control for Return

demo-retb

```
0x026:    ret
           bubble
           bubble
           bubble
0x014:    irmovq $5,%rsi # Return
```



Condition	F	D	E	M	W
Processing <code>ret</code>	stall	bubble	normal	normal	normal

Summary: Special Control Cases

Detection

Condition	Trigger
Processing <code>ret</code>	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & Ie_Cnd

Action (on next cycle)

Condition	F	D	E	M	W
Processing <code>ret</code>	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

Performance Metrics

Clock rate

- Measured in Gigahertz
- Function of stage partitioning and circuit design
 - Keep amount of work per stage small

Rate at which instructions executed

- CPI: cycles per instruction
- On average, how many clock cycles does each instruction require?
- Function of pipeline design and benchmark programs
 - E.g., how frequently are branches mispredicted?

CPI for PIPE

CPI ≈ 1.0

- Fetch instruction each clock cycle
- Effectively process new instruction almost every cycle
 - Although each individual instruction has latency of 5 cycles

CPI > 1.0

- Sometimes must stall or cancel branches

Computing CPI

- C clock cycles
- I instructions executed to completion
- B bubbles injected ($C = I + B$)
$$CPI = C/I = (I+B)/I = 1.0 + B/I$$
- Factor B/I represents average penalty due to bubbles

CPI for PIPE (Cont.)

$$B/I = LP + MP + RP$$

	Typical Values
■ LP: Penalty due to load/use hazard stalling	
● Fraction of instructions that are loads	0.25
● Fraction of load instructions requiring stall	0.20
● Number of bubbles injected each time	1
⇒ $LP = 0.25 * 0.20 * 1 = 0.05$	
■ MP: Penalty due to mispredicted branches	
● Fraction of instructions that are cond. jumps	0.20
● Fraction of cond. jumps mispredicted	0.40
● Number of bubbles injected each time	2
⇒ $MP = 0.20 * 0.40 * 2 = 0.16$	
■ RP: Penalty due to ret instructions	
● Fraction of instructions that are returns	0.02
● Number of bubbles injected each time	3
⇒ $RP = 0.02 * 3 = 0.06$	
■ Net effect of penalties $0.05 + 0.16 + 0.06 = 0.27$	
⇒ $CPI = 1.27$ (Not bad!)	

72 -

CS:APP:

Pipelining Summary

Data Hazards

- Most handled by forwarding: no performance penalty
- Load/use hazard requires one cycle stall

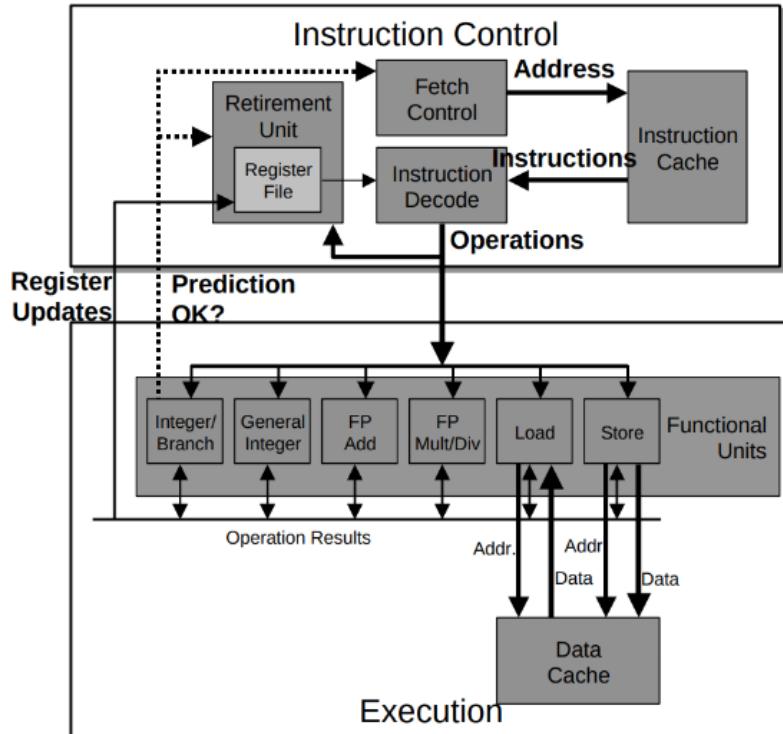
Control Hazards

- Cancel instructions when detect mispredicted branch
 - Two clock cycles wasted
- Stall fetch stage while `ret` passes through pipeline
 - Three clock cycles wasted

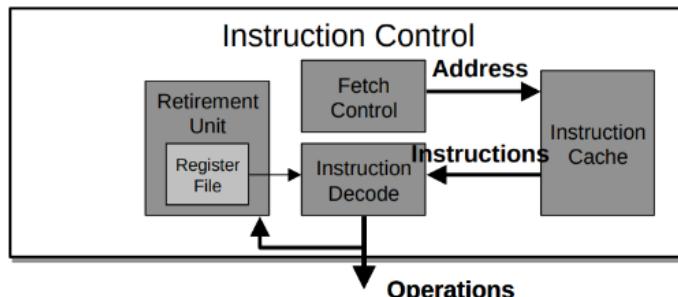
Control Combinations (Corner Cases)

- Must analyze carefully; first version subtle bug

Modern CPU Design



Instruction Control



Grabs Instruction Bytes From Memory

- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into Operations

- Primitive steps to perform instruction
- Typical instruction requires 1–3 operations

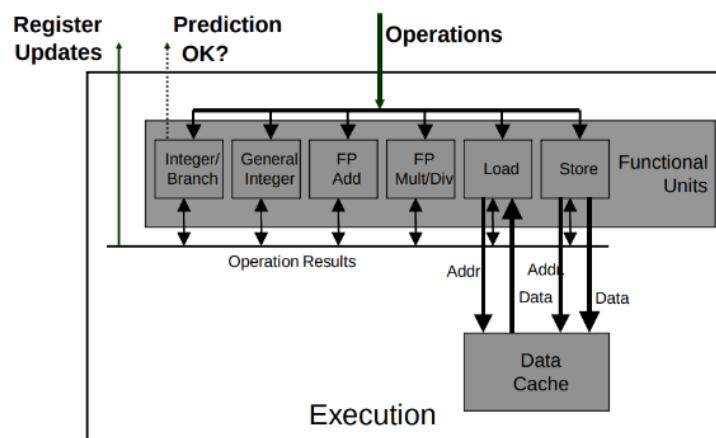
this is called **microcode**
shallower logic ==> more parallelizable
ex: (add instruction)
T4: IR_out(addr part), MAR_in
T5: read
T6: ACC_out, aluadd
T7: TEMP_out, ACC_in, reset to T0
(bonus: later, add instructions to ISA!)

Converts Register References Into Tags

- Abstract identifier linking destination of one operation with sources of later operations

this technique is called **register renaming**.
eliminates false data dependencies.
 $r1 := m[1024]$ $r1 := m[1024]$
 $r1 := r1 + 2$ $r1 := r1 + 2$
 $m[1032] := r1$ $m[1032] := r1$
 $r1 := m[2048]$ $r2 := m[2048]$
 $r1 := r1 + 4$ $r2 := r1 + 4$
 $m[2056] := r1$ $m[2056] := r2$

Execution Unit



- Multiple functional units
 - Each can operate independently
- Operations performed as soon as operands available
 - Not necessarily in program order
 - Within limits of functional units
- Control logic
 - Ensures behavior equivalent to sequential program execution

this is called **out-of-order execution**.
CPU reorders instructions, to e.g.
Avoid need to stall/bubble if there's a data-independent instruction nearby.

CPU Capabilities of Intel Haswell

Multiple Instructions Can Execute in Parallel

- 2 load
- 1 store
- 4 integer
- 2 FP multiply
- 1 FP add / divide

Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/Issue
Load / Store	4	1
Integer Multiply	3	1
Integer Divide	3—30	3—30
Double/Single FP Multiply	5	1
Double/Single FP Add	3	1
Double/Single FP Divide	10—15	6—11

Haswell Operation

Translates instructions dynamically into “Uops”

- ~118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine

- Uop executed when
 - Operands available
 - Functional unit available
- Execution controlled by “Reservation Stations”
 - Keeps track of data dependencies between uops
 - Allocates resources

High-Performance Branch Prediction

Critical to Performance

- Typically 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
 - Can recognize repeated patterns, e.g., alternating taken–not taken

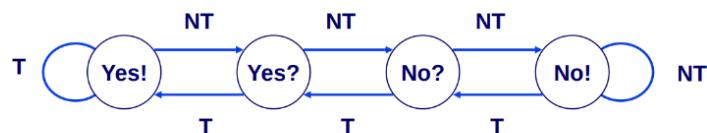
Handling BTB misses

- Detect in ~cycle 6
- Predict taken for negative offset, not taken for positive
 - Loops vs. conditionals

Example Branch Prediction

Branch History

- Encode information about prior history of branch instructions
- Predict whether or not branch will be taken



State Machine

- Each time branch taken, transition to right
- When not taken, transition to left
- Predict branch taken when in state Yes! or Yes?

Processor Summary

Design Technique

- Create uniform framework for all instructions
 - Want to share hardware among instructions
- Connect standard logic blocks with bits of control logic

Operation

- State held in memories and clocked registers
- Computation done by combinational logic
- Clocking of registers/memories sufficient to control overall behavior

Enhancing Performance

- Pipelining increases throughput and improves resource utilization
- Must make sure to maintain ISA behavior

All this... to understand an old CPU?

all these ideas are implemented in the Intel 8086 (1976) (gave rise to x86)



since: aggressively optimizing this design, incorporating breakthroughs in chip manufacturing

"Hey, I ran on airplanes, the Nintendo Gameboy, and all sorts of things!"

... until ca. 2005; we hit a limit.
since: multicore (next lecture).

Vectorization

SIMD Vector Instructions in a Nutshell

■ What are these instructions?

- Extension of the ISA. Data types and instructions for parallel computation on short (2-16) vectors of integers and floats



■ Why are they here?

- Useful: Many applications (e.g., multi media) feature the required fine grain parallelism – code potentially faster
- Doable: Chip designers have enough transistors available, easy to implement

What are the new registers?

Carnegie Mellon

Electrical & Computer
ENGINEERING

XMM/YMM/ZMM Relationship

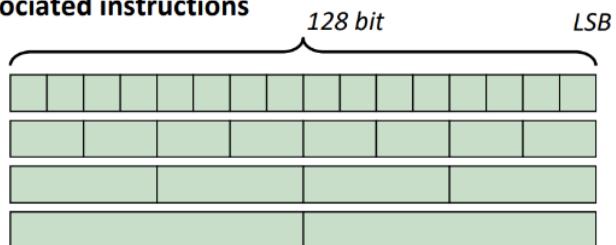


Detail: XMM/SSE2/3/4 Register

- Different data types and associated instructions

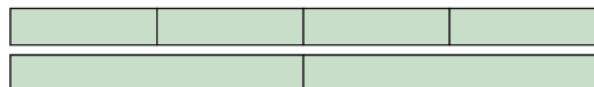
- Integer vectors:**

- 16-way byte
- 8-way 2 bytes
- 4-way 4 bytes
- 2-way 8 bytes



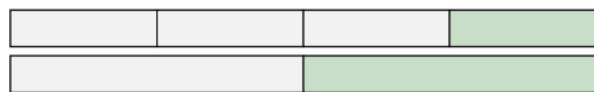
- Floating point vectors:**

- 4-way single (since SSE)
- 2-way double (since SSE2)

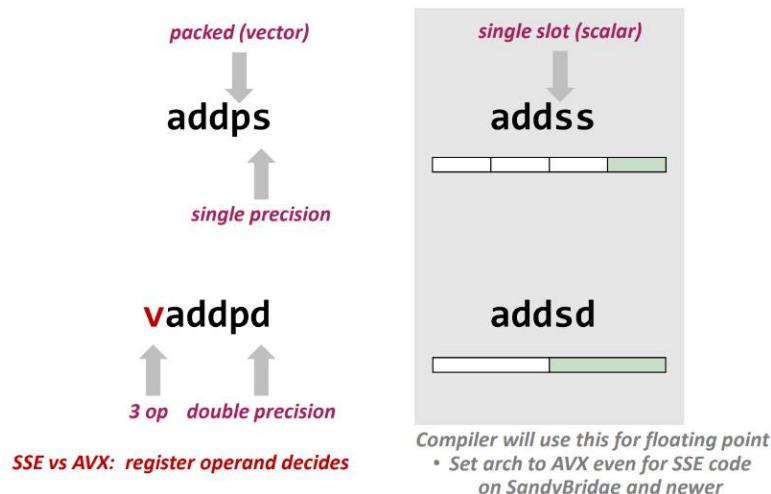


- Floating point scalars:**

- single (since SSE)
- double (since SSE2)

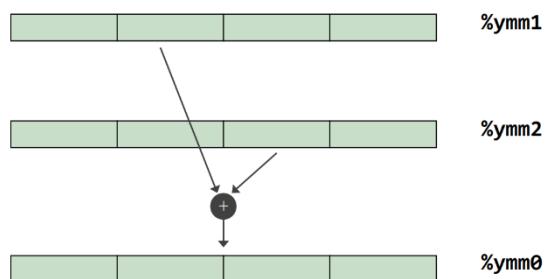


Instruction Names



AVX Instructions: Examples

- Double precision 4-way vector add: vaddpd %ymm0, %ymm1, %ymm2



Since AVX: Intel has 3-operand instructions and added AVX-style SSE instructions

SSE/AVX: How to Take Advantage?



- Necessary: fine grain parallelism
- Options (ordered by effort):
 - Use vectorized libraries (easy, not always available)
 - Compiler vectorization (good option)
 - Use intrinsics (this lecture)
 - Write assembly
- We will focus on 4-way (SSE single and AVX double)

Multicore

Two Laws of CS

30

Moore's law

“... the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.”

Dennard scaling

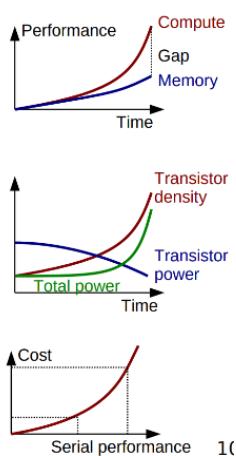
“... as transistors get smaller their power density stays constant, so that the power use stays in proportion with area.”

One way to do moores = increase number of cores

Heterogeneous computing

Technology evolution

- Memory wall
 - ♦ Memory speed does not increase as fast as computing speed
 - ♦ Harder to hide memory latency
- Power wall
 - ♦ Power consumption of transistors does not decrease as fast as density increases
 - ♦ Performance is now limited by power consumption
- ILP wall
 - ♦ Law of diminishing returns on Instruction-Level Parallelism
 - ♦ Pollack rule: cost \approx performance²

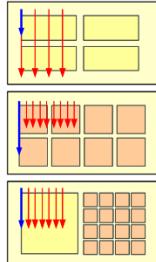


Why heterogeneous architectures?

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Time to run sequential portions Time to run parallel portions

- Latency-optimized multi-core (CPU)
 - Low efficiency on parallel portions: spends too much resources
- Throughput-optimized multi-core (GPU)
 - Low performance on sequential portions
- Heterogeneous multi-core (CPU+GPU)
 - Use the right tool for the right job
 - Allows aggressive optimization for latency **or** for throughput



M. Hill, M. Marty. Amdahl's law in the multicore era. IEEE Computer, 2008.

14

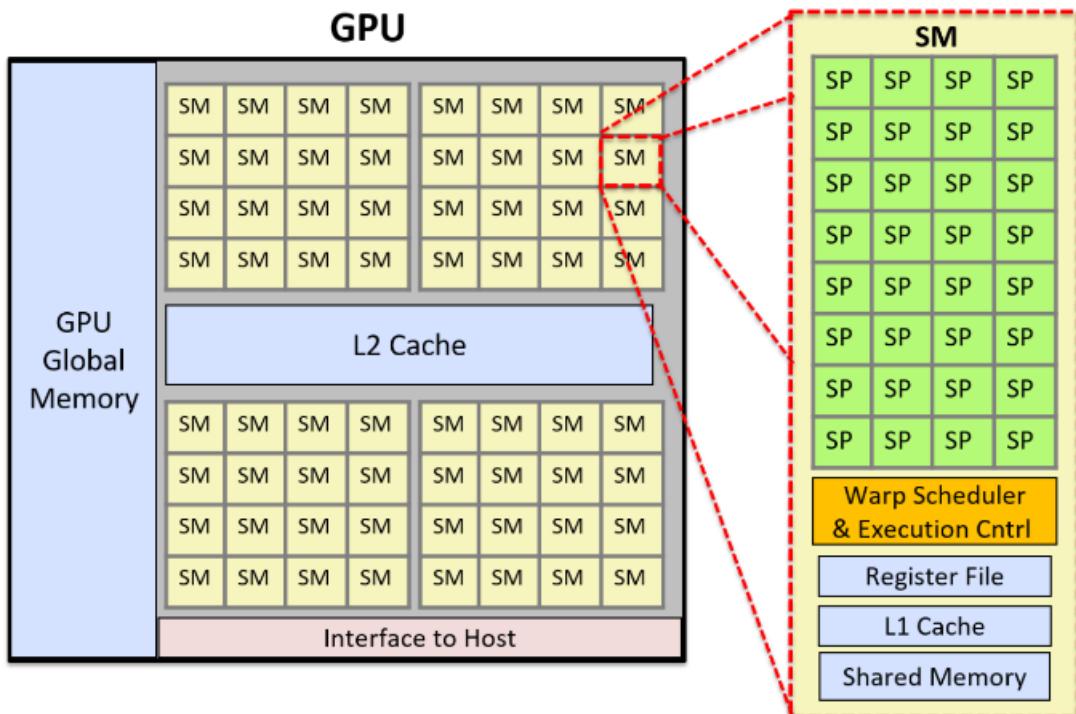


Figure 216. An example of a simplified GPU architecture with 2,048 cores. This shows the GPU divided into 64 SM units, and the details of one SM consisting of 32 SP cores. The SM's warp scheduler schedules thread warps on its SPs. A warp of threads executes in lockstep on the SP cores.

5.1.2. GPU architecture overview

GPU hardware is designed for computer graphics and image processing. Historically, GPU development has been driven by the video game industry. To support more detailed graphics and faster frame rendering, a GPU device consists of thousands of special-purpose processors, specifically designed to efficiently manipulate image data, such as the individual pixel values of a two-dimensional image, in parallel.

The hardware execution model implemented by GPUs is **single instruction/multiple thread** (SIMT), a variation of SIMD. IMT is like multithreaded SIMD, where a single instruction is executed in lockstep by multiple threads running on the processing units. In SIMT, the total number of threads can be larger than the total number of processing units, requiring the scheduling of multiple groups of threads on the processors to execute the same sequence of instructions.

As an example, NVIDIA GPUs consist of several streaming multiprocessors (SMs), each of which has its own execution control units and memory space (registers, L1 cache, and shared memory). Each SM consists of several scalar processor (SP) cores. The SM includes a warp scheduler that schedules **warps**, or sets of application threads, to execute in lockstep on its SP cores. In lockstep execution, each thread in a warp executes the same instruction each cycle but on different data. For example, if an application is changing a color image to grayscale, each thread in a warp executes the same sequence of instructions at the same time to set a pixel's RGB value to its grayscale equivalent. Each thread in the warp executes these instructions on a different pixel data value, resulting in multiple pixels of the image being updated in parallel. Because the threads are executed in lockstep, the processor design can be simplified so that multiple cores share the same instruction control units. Each unit contains cache memory and multiple registers that it uses to hold data as it's manipulated in lockstep by the parallel processing cores.

Figure 216 shows a simplified GPU architecture that includes a detailed view of one of its SM units. Each SM consists of multiple SP cores, a warp scheduler, an execution control unit, an L1 cache, and shared memory space.

4. CUDA

(Compute Unified Device Architecture)³ is NVIDIA's programming interface for GPGPU computing on its GPUs. CUDA is designed for heterogeneous computing in which some program functions run on the host CPU, and others run on the GPU device. Programmers typically write CUDA programs in C or C++ with annotations that specify kernel functions, and they make calls to CUDA library functions to manage GPU device memory. A CUDA **kernel** is a function that is executed on the GPU, and a CUDA **thread** is the basic unit of execution in a CUDA program. Threads are scheduled in warps that execute in lockstep on the GPU's SMs, executing CUDA kernel code on the data stored in GPU memory. Kernel functions are annotated with `global` to distinguish them from host functions. `device` functions are helper functions that can be called from a CUDA kernel function.

Memory space of a CUDA program is separated into host and GPU memory. The program must explicitly allocate host memory space to store program data manipulated by CUDA kernels. The CUDA programmer must either directly copy data to and from the host and GPU memory, or use CUDA unified memory that presents a view of memory that is directly shared by the GPU and host. Here is an example of CUDA's basic memory allocation, memory deallocation, and explicit memory copy functions:

```
returns" through pass-by-pointer param dev_ptr GPU memory of size bytes
returns cudaSuccess or a cudaError value on error

Malloc(void **dev_ptr, size_t size);
```

There are in fact two different CUDA assembly languages.

PTX is a machine-independent assembly language that is compiled down to SASS, the actual opcodes executed on a particular GPU family. If you build .cubins, you're dealing with SASS. CUDA runtime applications use PTX, since this enables them to run on GPUs released after the original application.

Also, function pointers have been in CUDA for a while if you're targeting sm_20 (Fermi/GTX series).

Memory

Reading

Locality

Dive into systems: 11.1-11.3

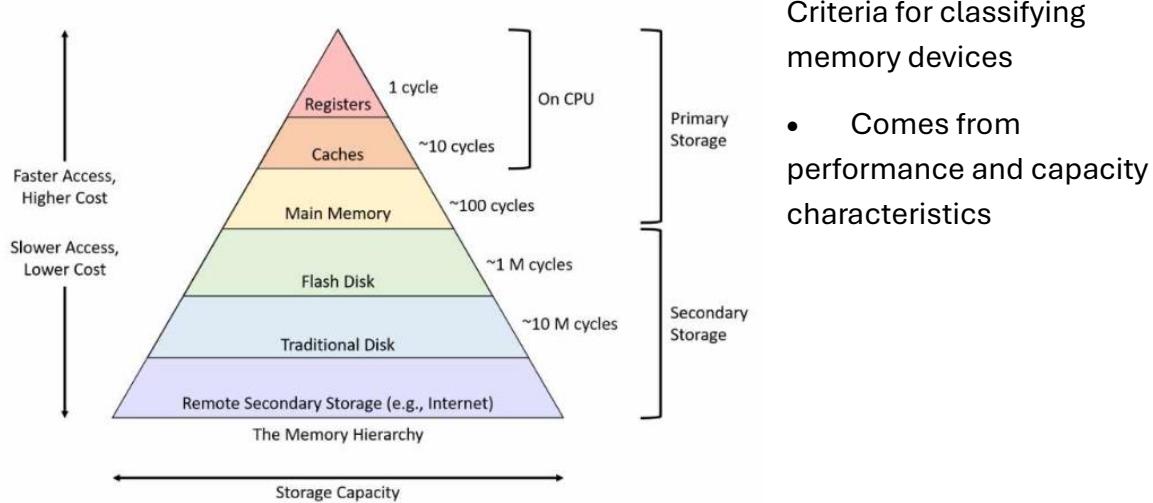


Figure 146. The memory hierarchy

The 3 most interesting measure:

- Capacity
 - The amount of data a device can store
 - Usually measured in bytes
- Latency
 - Amount of time it takes for a device to respond with data, after it has been instructed to perform a data retrieval operation
 - Typically measured in either fractions of a second (milliseconds, nanoseconds) or in CPU cycles
- Transfer rate
 - Amount of data that can be moved between the device and main memory over some interval of time
 - Also known as: throughput
 - Typically measured in bytes per second

The performance variance primarily arises from two factors: distance and variations in the technologies used to implement the devices.

Performance variance arises from 2 factors: distance and variations in the technology used to implement the devices

Distance

Any data that a program wants to use, must be available to the CPU and the ALU for processing

CPU designers place registers close to the ALU to minimize the time it takes for a signal to move between the two.

Since the registers can only store a few bytes, the values are available to the ALU immediately.

Secondary storage devices like disks transfer data to memory through controller devices, that are connected with long wires, the extra distance and intermediate processing slows down the storage.

Underlying technology also affects devices performance.

Registers and caches are built from relatively simple circuits, that consist of only a few logic gates.

Their small size and minimal complexity ensures that electrical signals can move through them quickly, reducing latency.

Traditional harddisks contain spinning magnetic platters, that store hundreds of gigabytes - they offer dense storage, their access latency is pretty high

"Computing devices need to be small in order to be fast"

Types of ram

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Table 108. Primary Storage Device Characteristics of a Typical 2020 Workstation

Device	Capacity	Approx. latency	RAM type
Register	4 - 8 bytes	< 1 ns	SRAM
CPU cache	1 - 32 megabytes	5 ns	SRAM
Main memory	4 - 64 gigabytes	100 ns	DRAM

SRAM stores data in a small electrical circuit.

- Typically faster
- Is integrated directly into a CPU to build registers and caches
- Expensive to build, operate and in the amount of spaces it occupies.

DRAM

- Stores in main memory

Modern Secondary Storage

Table 109. Secondary Storage Device Characteristics of a Typical 2020 Workstation

Device	Capacity	Latency	Transfer rate
Flash disk	0.5 - 2 terabytes	0.1 - 1 ms	200 - 3,000 megabytes / second
Traditional hard disk	0.5 - 10 terabytes	5 - 10 ms	100 - 200 megabytes / second
Remote network server	Varies considerably	20 - 200 ms	Varies considerably

Table 109 characterizes the secondary storage devices commonly available to workstations today. Figure 149 displays how the path from secondary storage to main memory generally passes through several intermediate device controllers. For example, a typical hard disk connects to a Serial ATA controller, which connects to the system I/O controller, which in turn connects to the memory bus. These intermediate devices make disks easier to use by abstracting the disk communication details from the OS and programmer. However, they also introduce transfer delays as data flows through the additional devices.

programmer. However, they also introduce transfer delays as data flows through the additional devices.

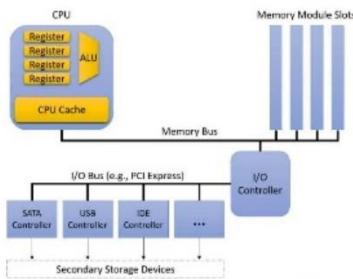


Figure 149. Secondary storage and I/O bus architecture

The two most common secondary storage devices today are **hard disk drives** (HDDs) and flash-based **solid-state drives** (SSDs). A hard disk consists of a few flat, circular platters made from a material that allows for magnetic recording. The platters rotate quickly, typically at speeds between 5,000 and 15,000 revolutions per minute. As the platters spin, a small mechanical arm with a disk head at the tip moves across the platter to read or write data on concentric tracks (regions of the platter located at the same diameter).

11.3.1. Locality Examples in Code

Fortunately, common programming patterns exhibit both forms of locality quite frequently. Take the following function, for example:

```
/* Sum up the elements in an integer array of length len. */
int sum_array(int *array, int len) {
    int i;
    int sum = 0;

    for (i = 0; i < len; i++) {
        sum += array[i];
    }

    return sum;
}
```

Systems primarily exploit two forms of locality:

1. **Temporal locality:** Programs tend to access the same data repeatedly over time. That is, if a program has used a variable recently, it's likely to use that variable again soon.
2. **Spatial locality:** Programs tend to access data that is nearby other, previously accessed data. "Nearby" here refers to the data's memory address. For example, if a program accesses data at addresses N and $N+4$, it's likely to access $N+8$ soon.

In this code, the repetitive nature of the `for` loop introduces temporal locality for `i`, `len`, `sum`, and `array` (the base address of the array), as the program accesses each of these variables within every loop iteration. Exploiting this temporal locality allows a system to load each variable from main memory into the CPU cache only once. Every subsequent access can be serviced out of the significantly faster cache.

Caching

Dive into systems - 11.4

If data is in the cache = cache hit

If data is not in the cache = cache miss

Important design questions

- Which subsets of a program's memory should the cache hold?
- When should the cache copy a subset of a program's data from main memory to the cache, or vice versa?
- How can a system determine whether a program's data is present in the cache?
 - metadata

Direct mapped caches

Divide storage into cache lines

Each line is independent

Contains 2 types of important information

- a cache data block

Cache data block

- Subset of data from memory
- Takes advantage of spatial locality

information: a *cache data block* and *metadata*.

1. A **cache data block** (often shortened to **cache block**) stores a subset of program data from main memory. Cache blocks store multibyte chunks of program data to take advantage of **spatial locality**. The size of a cache block determines the unit of data transfer between the cache and main memory. That is, when loading a cache with data from memory, the cache always receives a chunk of data the size of a cache block.

Cache designers balance a trade-off in choosing a cache's block size. Given a fixed storage budget, a cache can store more smaller blocks or fewer larger blocks. Using larger blocks improves performance for programs that exhibit good spatial locality, whereas having more blocks gives a cache the opportunity to store a more diverse subset of memory. Ultimately, which strategy provides the best performance depends on the workload of applications. Because general-purpose CPUs can't assume much about a system's applications, a typical CPU cache today uses middle-of-the-road block sizes ranging from 16 to 64 bytes.

2. **Metadata** stores information about the contents of the cache line's data block. A cache line's metadata does *not* contain program data. Instead, it maintains bookkeeping information for the cache line (for example, to help identify which subset of memory the cache line's data block holds).

Loading cached data

- First determine which cache lines to check
- Then identify contents

[Figure 152](#) shows how memory addresses map to cache lines in a small direct-mapped cache with four cache lines and a 32-byte cache block size. Recall that a cache's block size represents the smallest unit of data transfer between a cache and main memory. Thus, every memory address falls within one 32-byte range, and each range maps to one cache line.

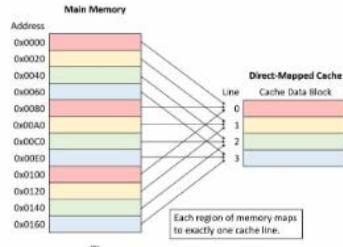


Figure 152. An example mapping of memory addresses to cache lines in a four-line direct-mapped cache with 32-byte cache blocks

A cache maps a memory address to a cache line using a portion of the bits in the memory address. To spread data more evenly among cache lines, caches use bits taken from the *middle* of the memory address, known as the **index** portion of the address, to determine which line the address maps to. The number of bits used as the index (which varies) determines how many lines a cache will hold. [Figure 153](#) shows the index portion of a memory address referring to a cache line.

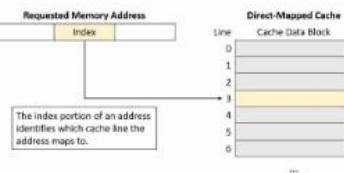


Figure 153. The middle index portion of a memory address identifies a cache line.

Important questions:

- Does this cache line hold a valid subset of memory?
- If so, which of the many subsets of memory that map to this cache line does it currently hold?

Each metadata contains a valid bit and a tag. The bit states if the cache line is storing a valid part of memory (1 if yes). Tag identifies which subset of memory

For a cache lookup to produce a hit, the tag field stored in the cache line must exactly match the tag portion (upper bits) of the program's requested memory address. A tag mismatch indicates that a cache line's data block does not contain the requested memory, even if the line stores valid data. [Figure 154](#) illustrates how a cache divides a memory address into a tag and an index, uses the index bits to select a target cache line, verifies a line's valid bit, and checks the line's tag for a match.

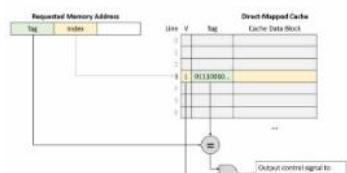


Figure 154. After using the requested memory address's index bits to locate the proper cache line, the cache simultaneously verifies the line's valid bit and checks its tag against the requested address's tag. If the line is valid with a matching tag, the lookup succeeds as a hit.

Retrieving Cached Data

Finally, after using the program's requested memory address to find the appropriate cache line and verifying that the line holds a valid subset of memory containing that address, the cache sends the requested data to the CPU's components that need it. Because a cache line's data block size (for example, 64 bytes) is typically much larger than the amount of data that programs request (for example, 4 bytes), caches use the low-order bits of the requested address as an **offset** into the cached data block. [Figure 155](#) depicts how the offset portion of an address identifies which bytes of a cache block the program expects to retrieve.



Figure 155. Given a cache data block, the offset portion of an address identifies which bytes the program wants to retrieve.

The *index* portion of the address begins immediately to the left of the offset. To determine the number of index bits, consider the number of lines in the cache, given that the index needs enough bits to uniquely identify every cache line. Using similar logic to the offset, a cache with 1,024 lines needs 10 bits for the index ($\log_2 1,024 = 10$). Likewise, a cache that uses 12 bits for the index must have 4,096 lines ($2^{12} = 4,096$).

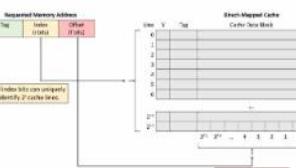


Figure 156. The index portion of an address uniquely identifies a cache line, and the offset portion uniquely identifies a position in the line's data block.

Write to cache

1. In a **write-through cache**, a memory write operation modifies the value in the cache and simultaneously updates the contents of main memory. That is, a write operation *always* synchronizes the contents of the cache and main memory immediately.
2. In a **write-back cache**, a memory write operation modifies the value stored in the cache's data block, but it does *not* update main memory. Thus, after updating the cache's data, a write-back cache's contents differ from the corresponding data in main memory.

Cache misses and associative designs

Real caches can't expect to hit on every access for a variety of reasons

- Compulsory misses / cold start misses
 - If the program has never accessed a memory location (or anyone near it) it has little hope of finding the data in the cache
 - Programs cannot often avoid cache missing when accessing new memory addresses
- Capacity misses
 - A cache stores a subset of main memory, and exactly the subset of memory that the program is actively using
 - If it's using more memory than what fits in the cache, it can't find ALL of the data it wants in the cache, which leads to misses
- Conflict misses
 - To reduce complexity of finding data, some designs limit where in the cache data can reside, this can lead to misses.
 - If a direct-mapped cache is not 100% full, the program might end up with the addresses of 2 frequently used variables that map to the same cache location.
 - Each access to one variable evicts the other from the cache, as they compete for the same line

Direct-mapped caches are less complex but suffer most from conflicts

- Alternative: associative cache

Fully associative

- Allows any memory region to occupy any location
- Offer most flexibility
- Has highest lookup and eviction complexity
 - Every location needs to be considered at the same time during any operation
- Unfit for general-purpose CPU

Set associative

- Occupy the middle ground between direct-mapped and fully associative designs
- Well suited for general-purpose CPU's
- Every memory region maps to exactly one cache set, each set stores multiple cache lines
 - Usually 2-8 lines per set

Set associative cache

If any of a set's valid lines contains a tag that matches the address's tag portion, the matching line completes the lookup. When the lookup narrows the search to just one cache line, it proceeds like a direct-mapped cache: the cache uses the address's *offset* to send the desired bytes from the line's cache block to the CPU's arithmetic components.

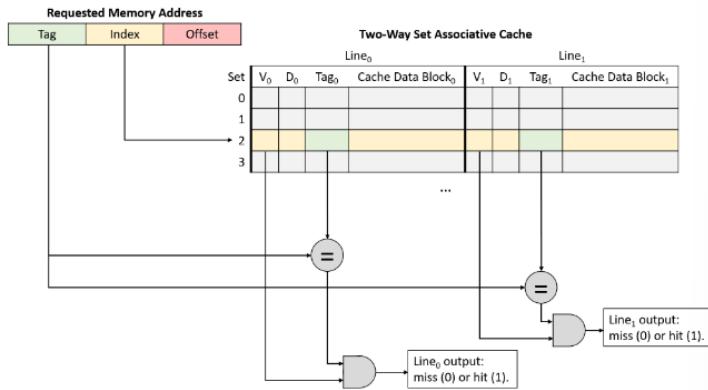


Figure 165. Valid bit verification and tag matching in a two-way set associative cache

Implements temporal locality

Figure 166 illustrates a two-way set associative cache, meaning each set contains two lines. With just two lines, each set requires one LRU metadata bit to keep track of which line was least recently used. In the figure, an LRU value of zero indicates the leftmost line was least recently used, and a value of one means the rightmost line was least recently used.

LRU (least recently used) = cache replacement policy

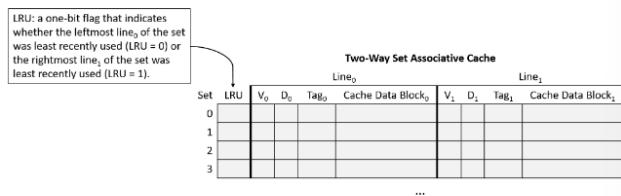


Figure 166. A two-way set associative cache in which each set stores one bit of LRU metadata to inform eviction decisions

Dive into systems - 11.6-11.7

Looking ahead: caching on multicore processors

Typically, each core maintains its own private cache memory at the highest level(s) of the memory hierarchy and shares a single cache with all cores at lower levels

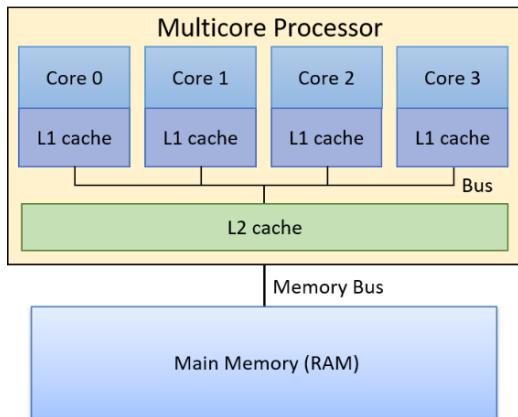


Figure 174. An example memory hierarchy on a multicore processor. Each of the four cores has its own private L1 cache, and all four cores share a single L2 cache that they access through a shared bus. The multicore processor connects to RAM via the memory bus.

L1 is smaller and faster than an L2 cache

L2 is a cache of RAM contents, and each L1 is a cache of L2 contents.

Giving each core its own L1 cache, allows it to store copies of the data and instructions away from the instruction stream

- Yields a higher hit rate in each core's private L1

Unified caches

- Store both program data and instructions

Cache coherency

Multiple L1 caches can result in cache coherency problems.

They arise when the value of a copy of a block of memory is different than a value of a copy of the same block on another L1's cache.

Multicore processors implement: cache-coherence protocol

MSI protocol (a cache coherence protocol)

MSI protocol = Modified, Shared, Invalid

Adds 3 flags(bits) to each cache line.

- The **M** flag that, if set, indicates the block has been modified, meaning that this core has written to its copy of the cached value.
- The **S** flag that, if set, indicates that the block is unmodified and can be safely shared, meaning that multiple L1 caches may safely store a copy of the block and read from their copy.
- The **I** flag that, if set, indicates if the cached block is invalid or contains stale data (is an older copy of the data that does not reflect the current value of the block of memory).

The MSI protocol is triggered on read and write accesses to cache entries.

On a read access:

- If the cache block is in the M or S state, the cached value is used to satisfy the read (its copy's value is the most current value of the block of memory).
- If the cache block is in the I state, the cached copy is out of date with a newer version of the block, and the block's new value needs to be loaded into the cache line before the read can be satisfied.

If another core's L1 cache stores the new value (it stores the value with the M flag set indicating that it stores a modified copy of the value), that other core must first write its value back to the lower level (for example, to the L2 cache). After performing the write-back, it clears the M flag with the cache line (its copy and the copy in the lower-level are now consistent) and sets the S bit to indicate that the block in this cache line is in a state that can be safely cached by other cores (the L1 block is consistent with its copy in the L2 cache and the core read the current value of the block from this L1 copy).

The core that initiated the read access on an line with the I flag set can then load the new value of the block into its cache line. It clears the I flag indicating that the block is now valid and stores the new value of the block, sets the S flag indicating that the block can be safely shared (it stores the latest value and is consistent with other cached copies), and clears the M flag indicating that the L1 block's value matches that of the copy stored in the L2 cache (a read does not modify the L1 cached copy of the memory).

On a write access:

- If the block is in the M state, write to the cached copy of the block. No changes to the flags are needed (the block remains in the M state).
- If the block is in the I or the S state, notify other cores that the block is being written to (modified). Other L1 caches that have the block stored in the S state, need to clear the S bit and set the I bit on their block (their copies of the block are now out of date with the copy that is being written to by the other core). If another L1 cache has the block in the M state, it will write its block back to the lower level, and set its copy to I. The core writing will then load the new value of the block into its L1 cache, set the M flag (its copy will be modified by the write), and clear the I flags (its copy is now valid), and write to the cached block.

[Figure 175](#) through [Figure 177](#) step through an example of the MSI protocol applied to ensure coherency of read and write accesses to a block of memory that is cached in two core's private L1 caches. In [Figure 175](#) our example starts with the shared data block copied into both core's L1 cache with the S flag set, meaning that the L1 cached copies are the same as the value of the block in the L2 cache (all copies store the current value of the block, 6). At this point, both core 0 and core 1 can safely read from the copy stored in their L1 caches without triggering coherency actions (the S flag indicates that their shared copy is up to date).

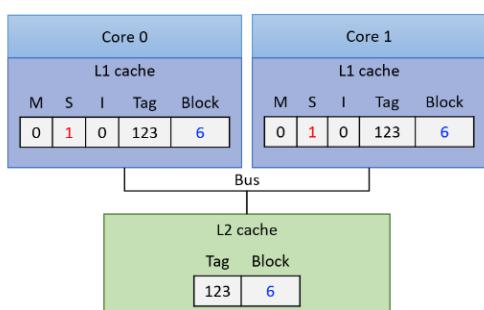


Figure 175. At the start, both cores have a copy of the block in their private L1 caches with the S flag set (in Shared mode)

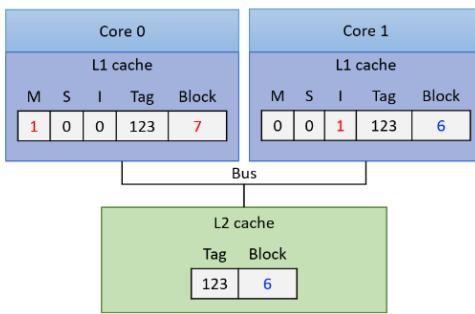


Figure 176. The resulting state of the caches after Core 0 writes to its copy of the block

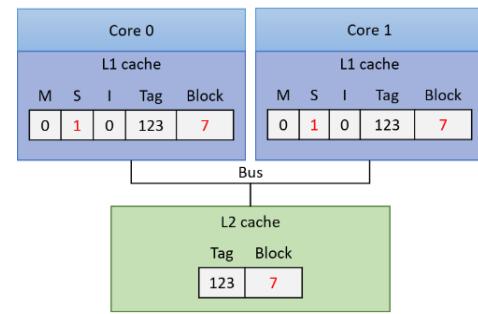


Figure 177. The resulting state of the caches after Core 1 next reads the block

To implement cache coherency protocol

- Snooping on a bus that is shared by all L1 caches
 - To learn other cores content

MSI and other similar protocols such as MESI and MOESI are write-invalidate protocols; that is, protocols that invalidate copies of cached entries on writes. Snooping can also be used by write-update cache coherency protocols, where the new value of a data is snooped from the bus and applied to update all copies stored in other L1 caches.

Virtual memory

13.3

Virtual memory is an abstraction that gives each process its own private, logical address space in which its instructions and data are stored. Each process's virtual address space can be thought of as an array of addressable bytes, from address 0 up to some maximum address.

Each process has its own copy

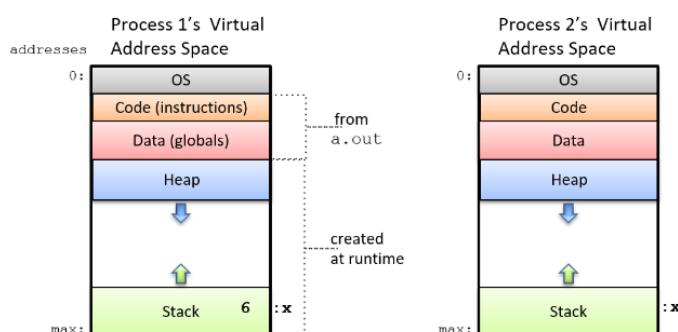


Figure 192. Two executions of *a.out* results in two processes, each running isolated instances of the *a.out* program. Each process has its own private virtual address space, containing its copies of program instructions, global variables, and stack and heap memory space. For example, each may have a local variable *x* in the stack portion of their virtual address spaces.

A process's virtual address space is divided into several sections, each of which stores a different part of the process's memory. The top part (at the lowest addresses) is reserved for the OS and can only be accessed in kernel mode. The text and data parts of a process's virtual address space are initialized from the program executable file (a.out). The text section contains the program instructions, and the data section contains global variables (the data portion is actually divided into two parts, one for initialized global variables and the other for uninitialized globals).

Stack space grows in response to the process making function calls, and shrinks as it returns from functions. Heap space grows when the process dynamically allocates memory space (via calls to malloc), and shrinks when the process frees dynamically allocated memory space (via calls to free).

Virtual addresses refer to storage locations in a process's virtual address space, and **physical addresses** refer to storage locations in physical memory (RAM).

The CPU cannot directly access other storage devices

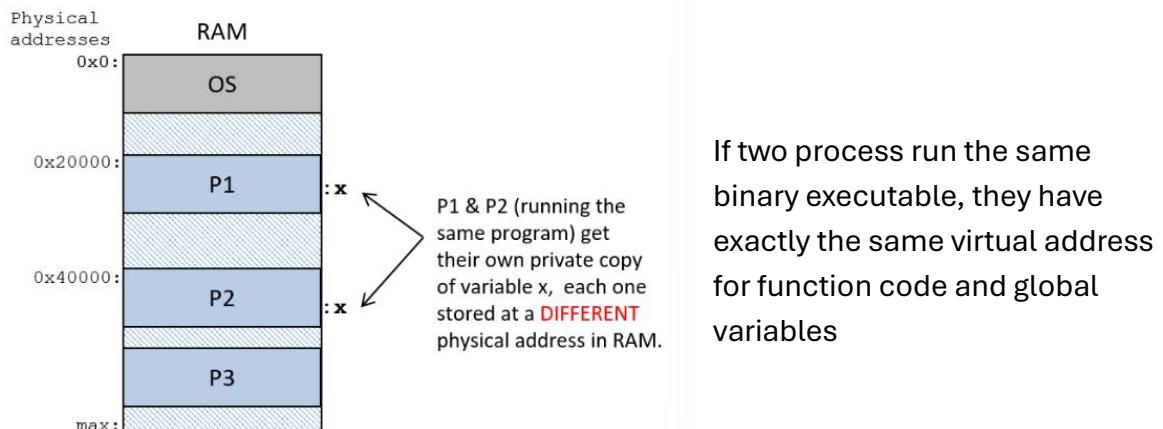


Figure 193. Example RAM contents showing OS loaded at address 0x0, and processes loaded at different physical memory addresses in RAM. If P1 and P2 are running the same a.out, P1's physical address for x is different from P2's physical address for x.

If two process run the same binary executable, they have exactly the same virtual address for function code and global variables

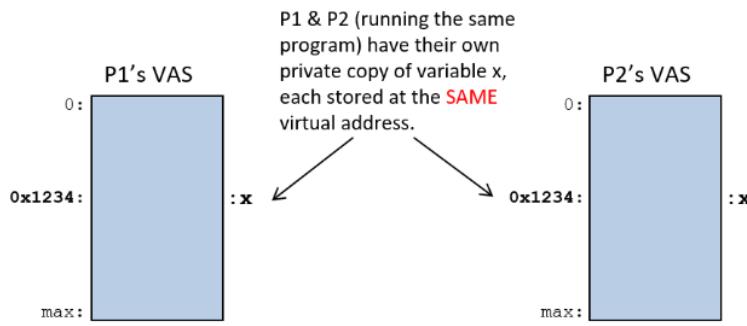


Figure 194. Example virtual memory contents for two processes running the same *a.out* file. P1 and P2 have the same virtual address for global variable x.

The **memory management unit (MMU)** is the part of the computer hardware that implements address translation. Together, the MMU hardware and the OS translate virtual to physical addresses when applications access memory.

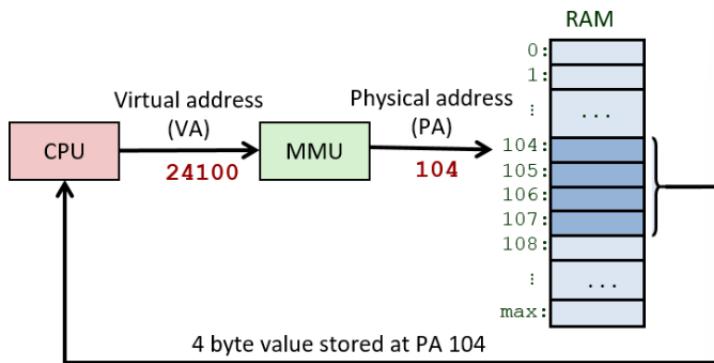


Figure 195. The memory management unit (MMU) maps virtual to physical addresses. Virtual addresses are used in instructions executed by the CPU. When the CPU needs to fetch data from physical memory, the virtual address is first translated by the MMU to a physical address that is used to address RAM.

In a **paged virtual memory** system, the OS divides the virtual address space of each process into fixed-sized chunks called **pages**

Physical pages do the same with Frames.

In a paging system:

- Pages and frames are the same size, so any page of virtual memory can be loaded into (stored) in any physical frame of RAM.
- A process's pages do not need to be stored in contiguous RAM frames (at a sequence of addresses all next to one another in RAM).
- Not every page of virtual address space needs to be loaded into RAM for a process to run.

Paged virtual memory systems divide the bits of a virtual address into two parts: the high-order bits specify the **page number** on which the virtual address is stored, and the low-order bits correspond to the **byte offset** within the page (which byte from the top of the page corresponds to the address).

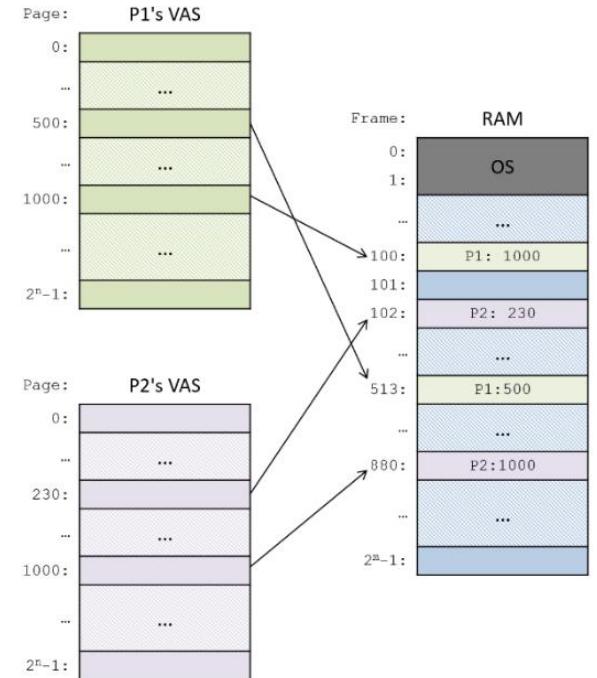


Figure 196. Paged virtual memory. Individual pages of a process's virtual address space are stored in RAM frames. Any page of virtual address space can be loaded into (stored at) any frame of physical memory. In this example, P1's virtual page 1000 is stored in physical frame 100, and its page 500 resides in frame 513. P2's virtual page 1000 is stored in physical frame 880, and its page 230 resides in frame 102.

Virtual Address Space of 2^n bytes, Page size 2^k bytes, VA bits:



Physical Address Space of 2^m bytes, Page size 2^k bytes, PA bits:

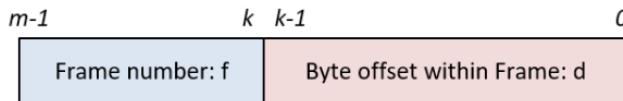


Figure 197. The address bits in virtual and physical addresses

Virtual Address (16 bits): 1010100101011101		Physical Address (14 bits): 00000101011101	
bit: 15	$3 \ 2 \ 1 \ 0$	bit: 13	$3 \ 2 \ 1 \ 0$
1010100101011101	101	00000101011101	101
Page number 13 bits	byte offset 3 bits	Frame number 11 bits	byte offset 3 bits
on virtual page 5419 at byte offset 5		in physical frame 43 at byte offset 5	

Figure 198. Virtual and physical address bit divisions in an example system with 16-bit virtual addresses, 14-bit physical addresses, and a page size of 8 bytes.

In the example in Figure 198, virtual address 43357 (in decimal) has a byte offset of 5 (0b101 in binary), the low-order 3 bits of the address, and a page number of 5419 (0b1010100101011), the high-order 13 bits of the address. This means that the virtual address is at byte 5 from the top of page 5419.

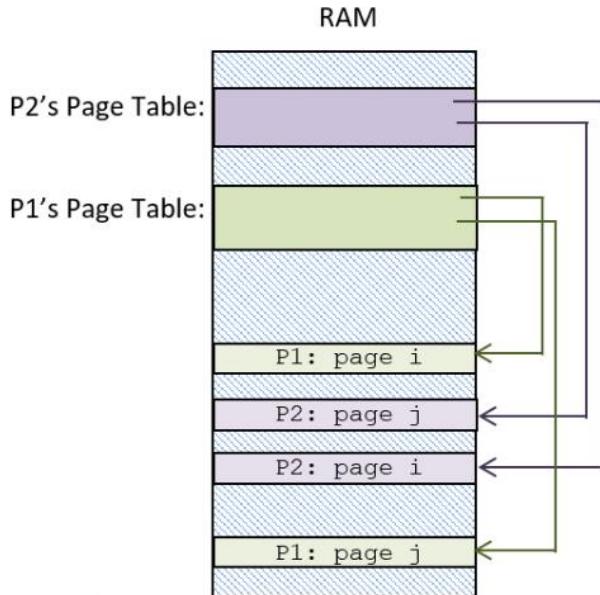
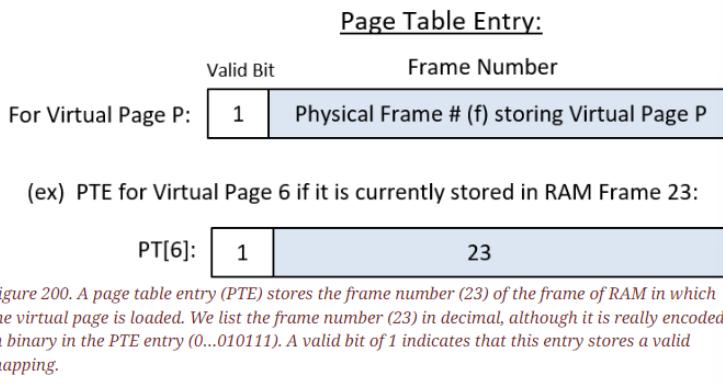


Figure 199. Every process has a page table containing its virtual page to physical frame mappings. Page tables, stored in RAM, are used by the system to translate process's virtual addresses to physical addresses that are used to address locations in RAM. This example shows the separate page tables stored in RAM for processes P1 and P2, each page table with its own virtual page to physical frame mappings.

PTE = page table entry



4 steps to translate a virtual address to a physical address

1. First, the MMU divides the bits of the virtual address into two parts: for a page size of 2^k bytes, the low-order k bits (VA bits $(k-1)$ to 0) encode the byte offset (d) into the page, and the high-order $n-k$ bits (VA bits $(n-1)$ to k) encode the virtual page number (p).
2. Next, the page number value (p) is used by the MMU as an index into the page table to access the PTE for page p . Most architectures have a **page table base register** (PTBR) that stores the RAM address of the running process's page table. The value in the PTBR is combined with the page number value (p) to compute the address of the PTE for page p .
3. If the valid bit in the PTE is set (is 1), then the frame number in the PTE represents a valid VA to PA mapping. If the valid bit is 0, then a page fault occurs, triggering the OS to handle this address translation (we discuss the OS page fault handling later).
4. The MMU constructs the physical address using the frame number (f) bits from the PTE entry as the high-order bits, and the page offset (d) bits from the VA as the low-order bits of the physical address.

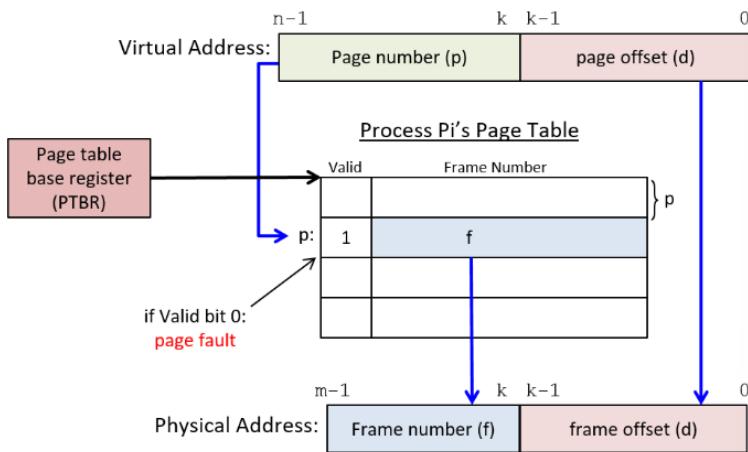


Figure 201. A process's page table is used to perform virtual to physical address translations. The PTBR stores the base address of the currently running process's page table.

Page fault

- Try to access a page that is not stored in RAM

In virtual memory systems, however, processes sometimes try to access a page that is currently not stored in RAM (causing a **page fault**). When a page fault occurs, the OS needs to read the page from disk into RAM before the process can continue executing. The MMU reads a PTE's valid bit to determine whether it needs to trigger a page fault exception. When it encounters a PTE whose valid bit is zero, it traps to the OS, which takes the following steps:

1. The OS finds a free frame (e.g., frame j) of RAM into which it will load the faulted page.
2. It next issues a read to the disk to load the page from disk into frame j of RAM.
3. When the read from disk has completed, the OS updates the PTE entry, setting the frame number to j and the valid bit to 1 (this PTE for the faulted page now has a valid mapping to frame j).
4. Finally, the OS restarts the process at the instruction that caused the page fault. Now that the page table holds a valid mapping for the page that faulted, the process can access the virtual memory address that maps to an offset in physical frame j.

A **translation look-aside buffer** (TLB) is a hardware cache that stores (page number, frame number) mappings. It is a small, fully associative cache that is optimized for fast lookups in hardware. When the MMU finds a mapping in the TLB (a TLB hit), a page table lookup is not needed, and only one RAM access is required to execute a load or store to a virtual memory address. When a mapping is not found in the TLB (a TLB miss), then an additional RAM access to the page's PTE is required to first construct the physical address of the load or store to RAM. The mapping associated with a TLB miss is added into the TLB. With good locality of memory references, the hit rate in the TLB is very high, resulting in fast memory accesses in paged virtual memory—most virtual memory accesses require only a single RAM access. [Figure 202](#) shows how the TLB is used in virtual-to-physical address mappings.

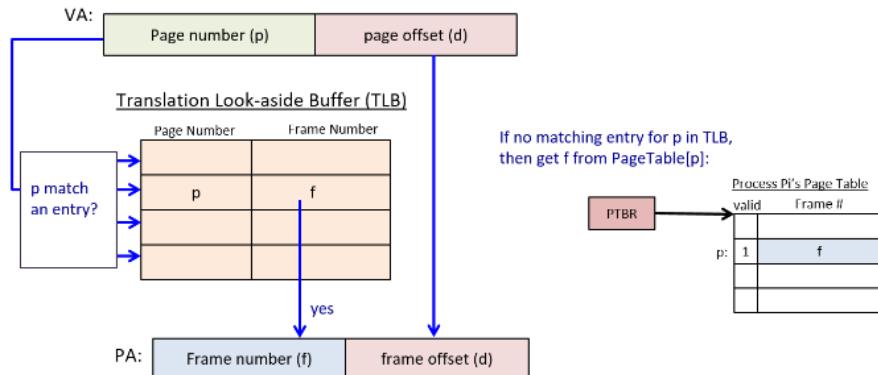


Figure 202. The translation look-aside buffer (TLB) is a small hardware cache of virtual page to physical frame mappings. The TLB is first searched for an entry for page p. If found, no page table lookup is needed to translate the virtual address to its physical address.

Slides

Multicore

Moore's law and Dennard scaling

- Want to keep CPU small

Moore's law

“... the observation that the number of transistors in a dense integrated circuit doubles approximately every two years.”

Adding more cores to the CPU (explicit parallelism)

Multisocket multicores

Dennard scaling

“... as transistors get smaller their power density stays constant, so that the power use stays in proportion with area.”

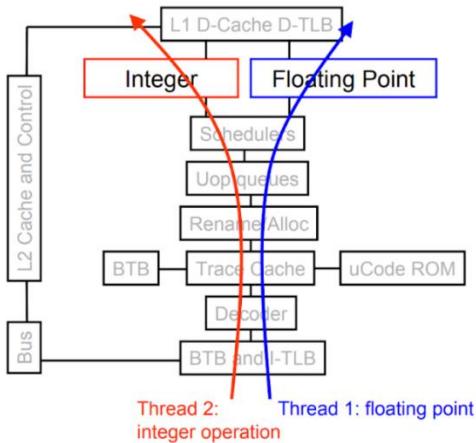
Simultaneous multithreading

- Complementary to multicore

SMT processor

- Both threads can run concurrently
- Not a true parallel processor
- Still speeds up

SMT processor: both threads can run concurrently



SMT not a “true” parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Compare to multi-core: each core has its own copy of resources

21

Pitfall - amadahl's law

- You cannot parallelise everything

fun, fantastic, but) don't expect magic

Pitfall: Amdahl's Law

Execution time after improvement =

$$\frac{\text{affected execution time}}{\text{amount of improvement}} + \text{execution time unaffected}$$

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

analogies:

- “too many cooks spoil the broth”
- project management joke:
it takes 1 woman 9 months to grow 1 baby
⇒ takes 9 women 1 months to grow 1 baby

takeaway: diminishing returns.

embarrassingly parallel: task that can be divided cleanly and split off to threads of execution. (ideal)
reality: threads need to communicate & coordinate; incurs overhead.

UNIVERSITY OF COPENHAGEN

slide by Hakim Weatherspoon, from CS 3410 course at Cornell University

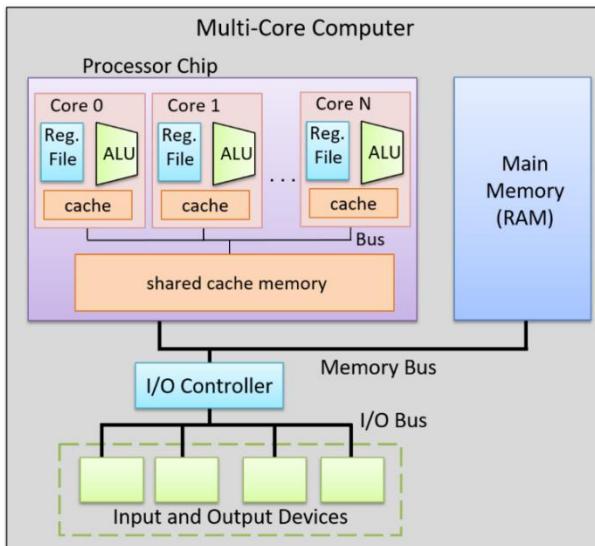


Figure 90. A computer with a multicore processor. The processor contains multiple complete CPU cores, each with its own private cache memory. The cores communicate with each other and share a larger shared cache memory via on-chip buses.

Memory bus

```

:l multicore (?):
//stackoverflow.com/a/991191
er boots: core 0 processor 0 - the bootstrap core - starts executing code at 0xffffffff0.
her cores are "sleeping", in a Wait-for-SIPI state.
trap core can send inter-processor interrupt (IPI) - a startup IPI (SIPI) -
he advanced programmable interrupt controller (APIC).
PI contains the address from which that core should start executing code.
ode can therefore be a kernel (instance specific to that core).
ore's kernel gets context-switched into at regular intervals (just like bootstrap-core).
ore's kernel can then schedule another thread to run.
all cores' kernels share memory, they can coordinate (e.g. which thread runs on what kerne
can announce to each other (by updating memory) that they are operational).
hread (on a core) wants to start a new thread, it makes a SYSTEM CALL;
(on that core) takes care of the rest
ing the thread in mem, which can then be picked up by any core's kernel for execution).

```

The computer is a single core when it starts (the others are sleeping)

Heterogeneous computing

Memory wall

Power wall

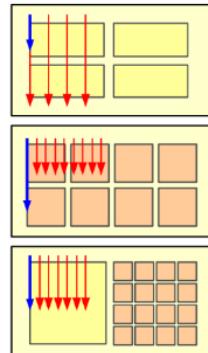
ILP wall

Why heterogeneous architectures?

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Time to run sequential portions Time to run parallel portions

- Latency-optimized multi-core (CPU)
 - ♦ Low efficiency on parallel portions: spends too much resources
- Throughput-optimized multi-core (GPU)
 - ♦ Low performance on sequential portions
- Heterogeneous multi-core (CPU+GPU)
 - ♦ Use the right tool for the right job
 - ♦ Allows aggressive optimization for latency **or** for throughput



M. Hill, M. Marty. Amdahl's law in the multicore era. IEEE Computer, 2008.

14

You can make one that is optimized for specific features

- Could be for latency
- Or optimized for multicore

Heterogeneous mean

- Both

Writing programs that utilize both

The hardware execution model implemented by GPUs is **single instruction/multiple thread** (SIMT), a variation of SIMD. SIMT is like multithreaded SIMD, where a single instruction is executed in lockstep by multiple threads running on the processing units. In SIMT, the total number of threads can be larger than the total number of processing units, requiring the scheduling of multiple groups of threads on the processors to execute the same sequence of instructions.

As an example, NVIDIA GPUs consist of several streaming multiprocessors (SMs), each of which has its own execution control units and memory space (registers, L1 cache, and shared memory). Each SM consists of several scalar processor (SP) cores. The SM includes a warp scheduler that schedules **warps**, or sets of application threads, to execute in lockstep on its SP cores. In lockstep execution, each thread in a warp executes the same instruction each cycle but on different

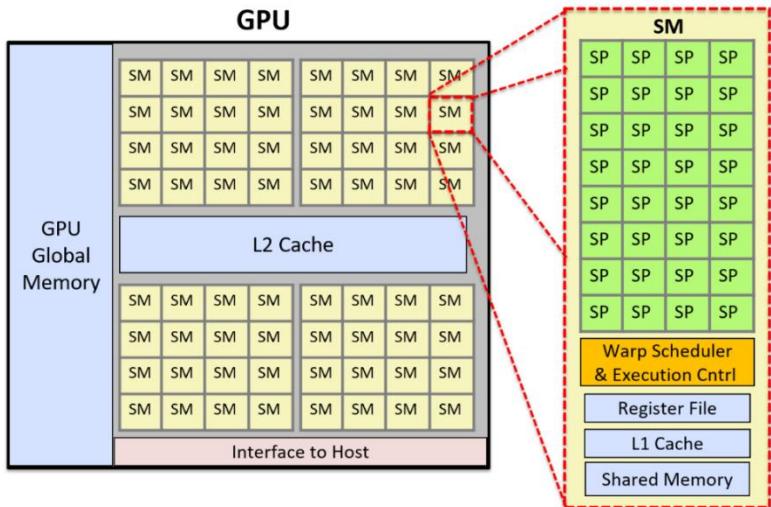


Figure 216. An example of a simplified GPU architecture with 2,048 cores. This shows the GPU divided into 64 SM units, and the details of one SM consisting of 32 SP cores. The SM's warp scheduler schedules thread warps on its SPs. A warp of threads executes in lockstep on the SP cores.

kernel functions, and they make calls to CUDA library functions to manage GPU device memory. A **CUDA kernel** is a function that is executed on the GPU, and a **CUDA thread** is the basic unit of execution in a CUDA program. Threads are scheduled in warps that execute in lockstep on the GPU's SMs, executing CUDA kernel code on threads stored in GPU memory. Kernel functions are annotated with *global* to distinguish them from host functions.

When running this function - each of the streams run the same code on different processors.

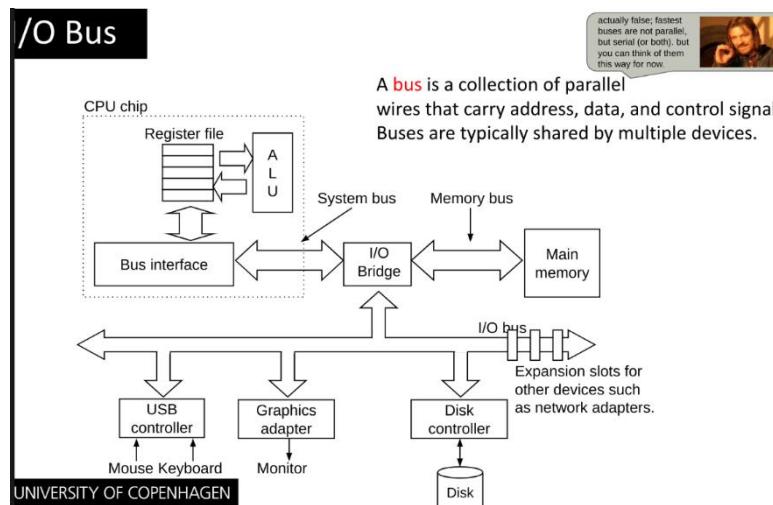
Each SM will one the first instruction, on its own portion of the data

- Make sure the data can be split in this way

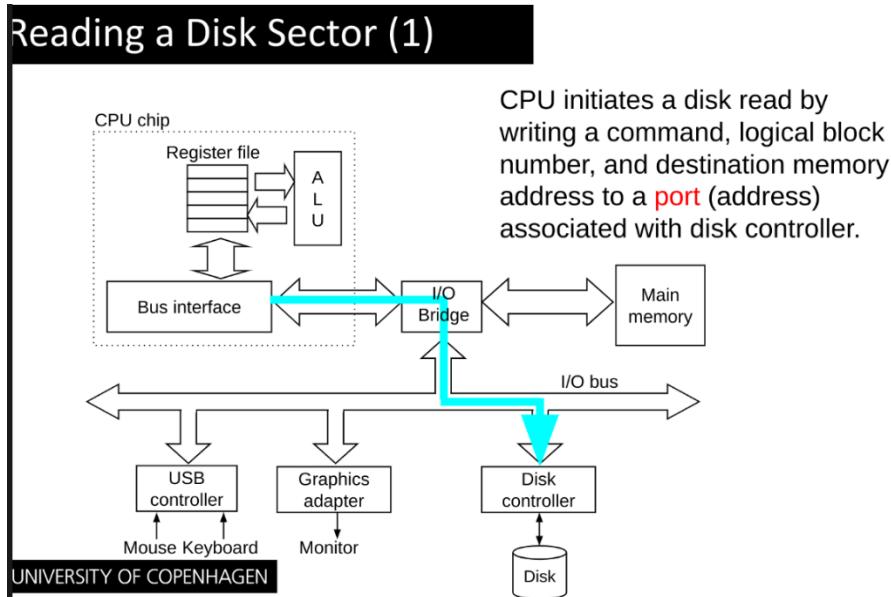
We will be working with this in C - that has keywords which will help us

Memory

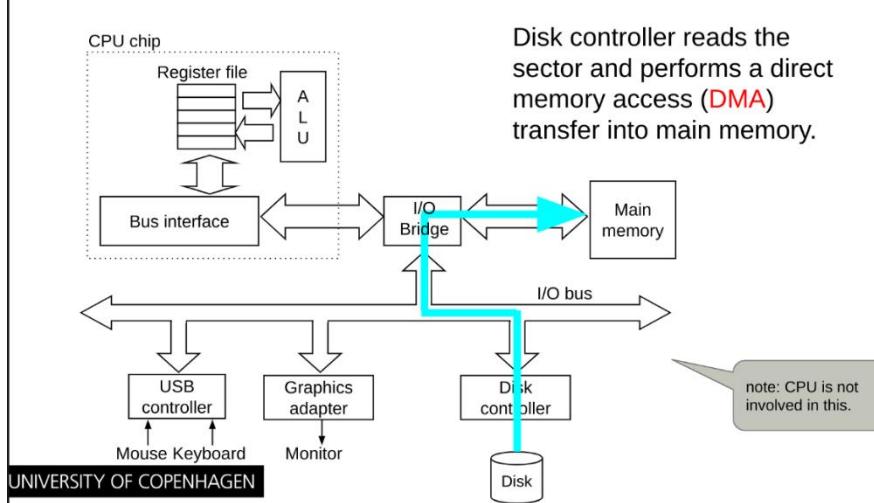
I/O Bus



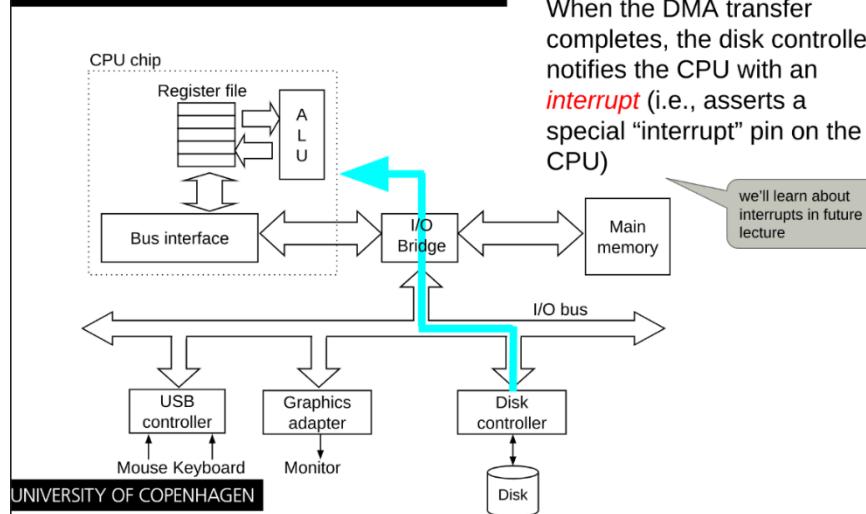
Reading a disk sector (1)



Reading a Disk Sector (2)



Reading a Disk Sector (3)



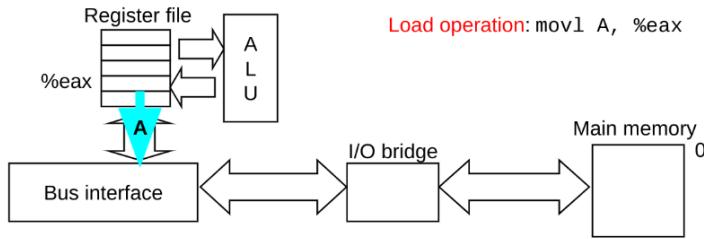
The disk pokes the CPU (hey, I need this)

The CPU will stop running what it's doing, and then do what it's asking?

Memory read

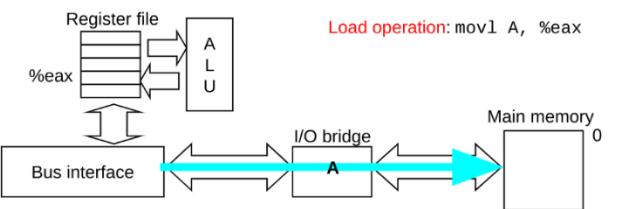
Memory Read Transaction (1)

CPU places address A on the memory bus.



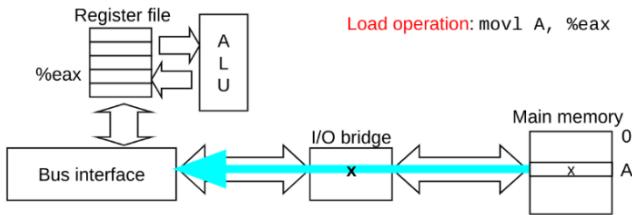
Memory Read Transaction (2)

Main memory reads A from the memory bus.



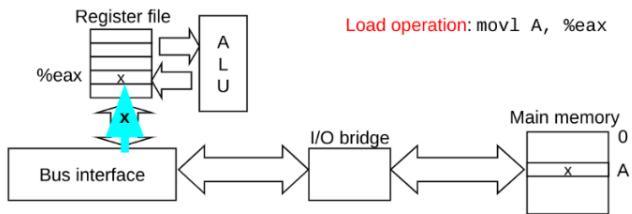
Memory Read Transaction (3)

Main memory retrieves word at address A (that's x),
and places it on the bus.



Memory Read Transaction (4)

CPU reads word from the bus (that's x),
and copies it into %eax.

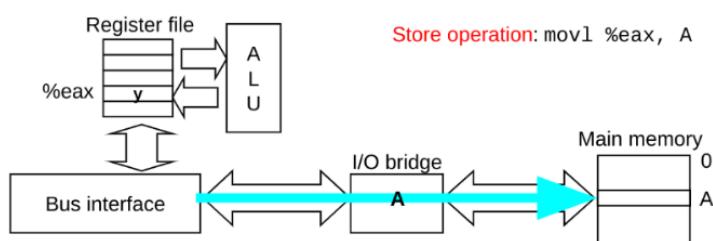


main memory will figure out where that address belongs

Memory write

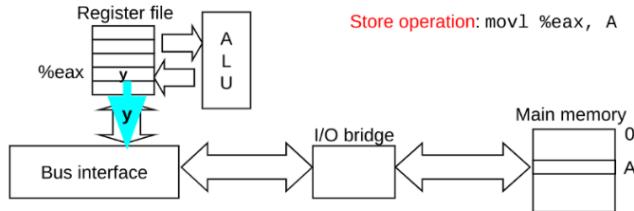
Memory Write Transaction (2)

Main memory reads A from the memory bus,
and waits for the corresponding data to arrive.



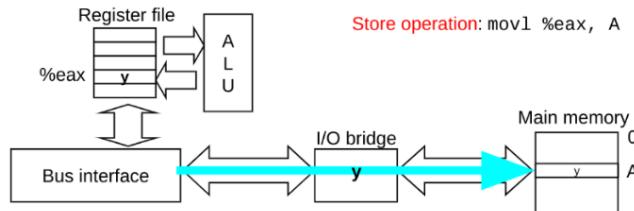
Memory Write Transaction (3)

CPU places contents of %eax on the memory bus
(that's word y).



Memory Write Transaction (4)

Main memory reads y from the memory bus,
and stores it at address A.



Locality

Use data and instructions with addresses near or equal to those used recently

Locality example

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Data references

Reference array elements in succession
(stride-1 reference pattern).

Spatial locality

Reference variable sum each iteration.

Temporal locality

Instruction references

Reference instructions in sequence.
Cycle through loop repeatedly.

Spatial locality

Temporal locality

Qualitative estimates of locality

Qualitative Estimates of Locality

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality wrt. array a?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Q: which is faster?

UNIVERSITY OF COPENHAGEN

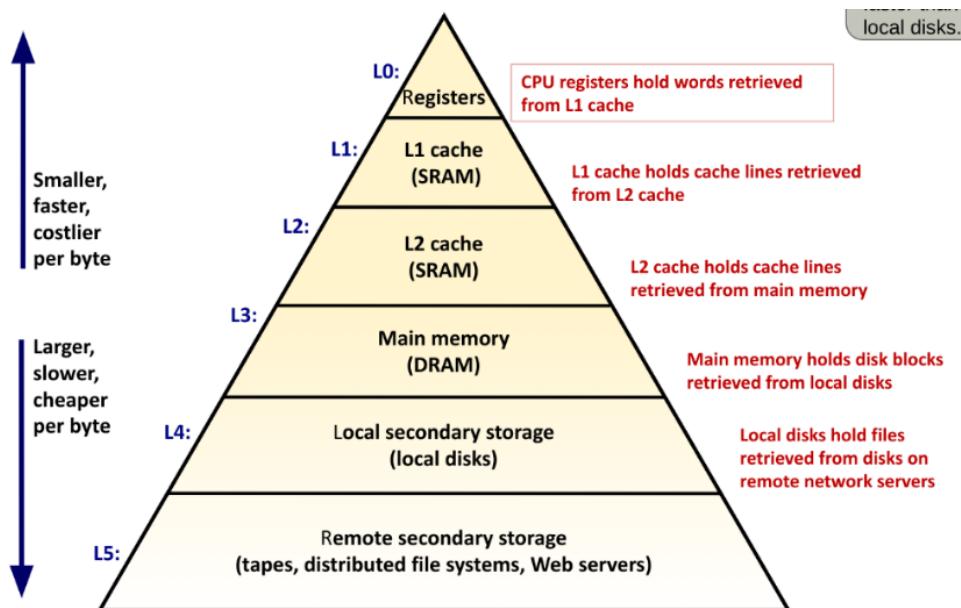
Question answer

- Rows

- An array of arrays (the end of the array will be in the beginning in the array)

Memory hierarchies

- Some fundamental and enduring properties of HW & SW:
 - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
 - The gap between CPU and main memory *speed* is widening.
 - Well-written programs tend to exhibit good locality.
- These fundamental properties complement each other beautifully. They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.



(local network is faster than some local disks)

Caching

Caches

Caches

car mechanic analogy

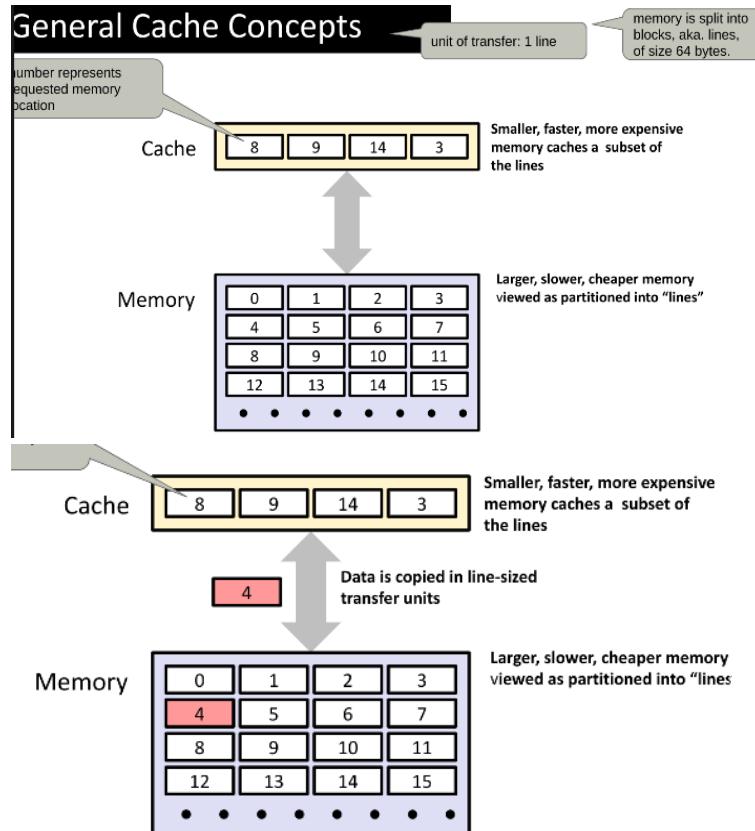
- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- **Fundamental idea of a memory hierarchy:**
 - For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.
- Why do memory hierarchies work? Because **locality**.
 - Programs tend to access the data at level k more often than they access the data at level $k+1$.
 - Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.
- **Big Idea:** Memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but serves data to programs at the rate of the fast storage near the top.

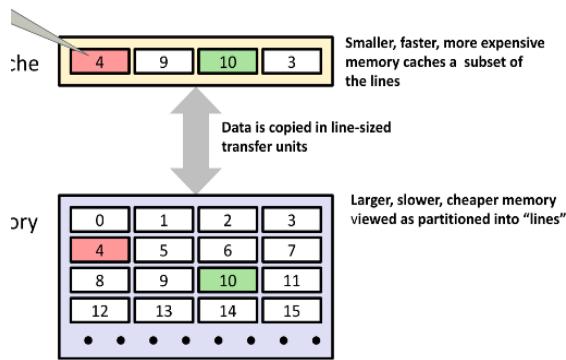
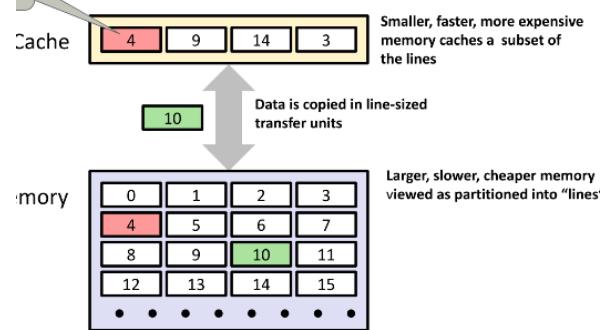
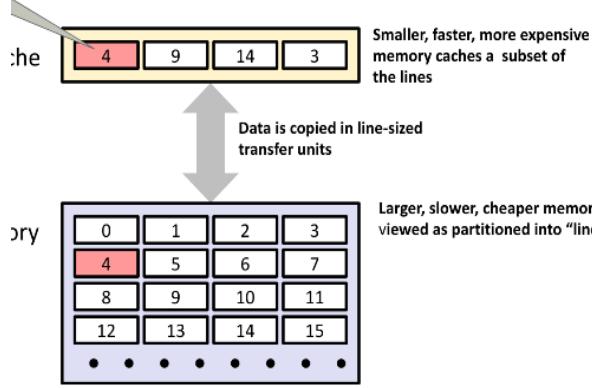
Cache hit and miss

Types of miss

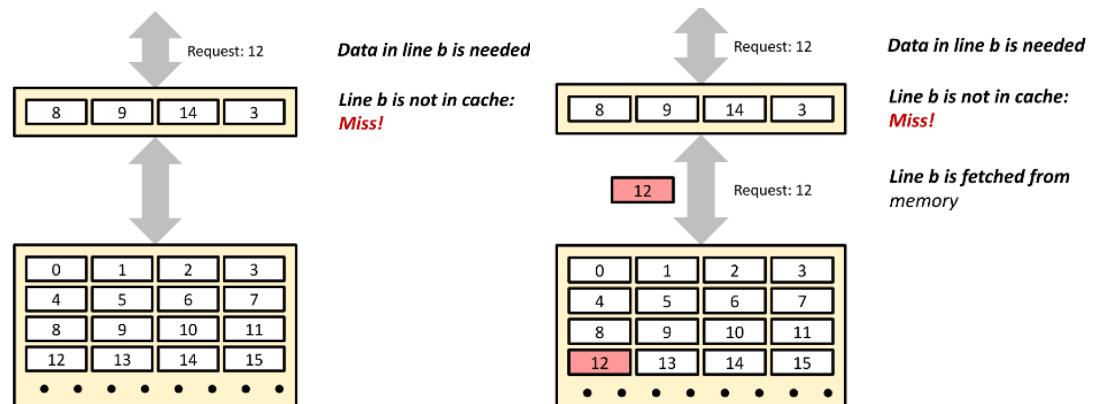
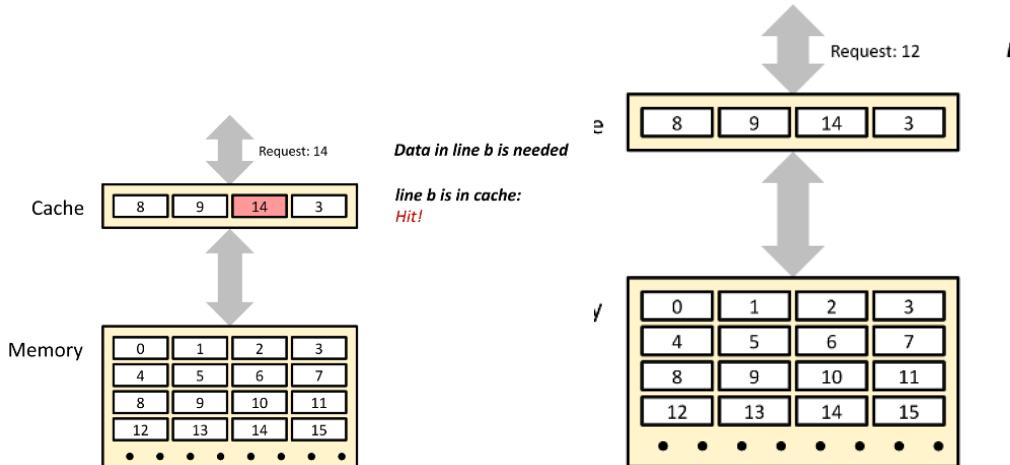
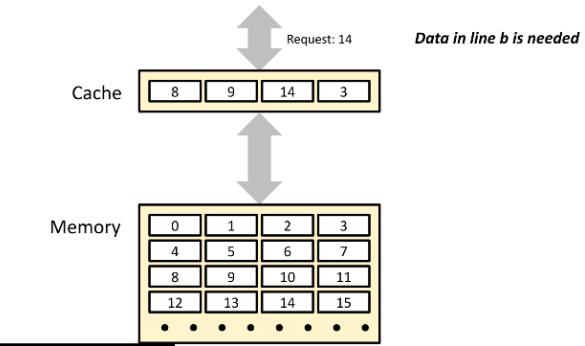
Thrashing = every access misses

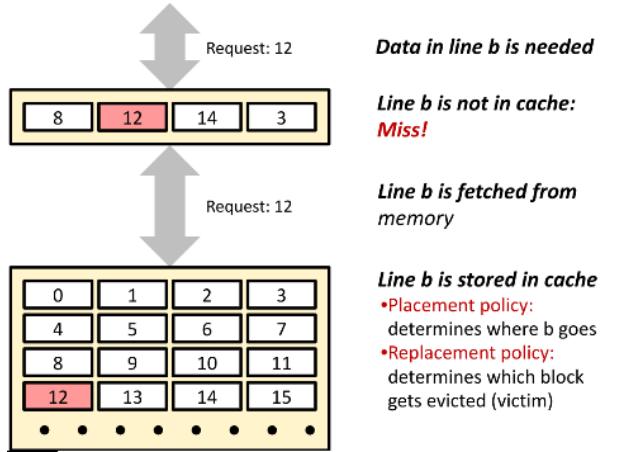
General Cache Concepts





General Cache Concepts: Hit





Cold (compulsory) miss

Cold misses occur because the cache is empty.

Conflict miss

Most caches limit lines at level $k+1$ to a small subset (sometimes a singleton) of the line positions at level k .

- E.g. Line i at level $k+1$ must be placed in line $(i \bmod 4)$ at level k .

Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k line.

- E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

Capacity miss

Occurs when set of active cache lines (**working set**) is larger than the cache.

UNIVERSITY OF COPENHAGEN

placement policy
avoids conflict misses
replacement policy care, to
avoid capacity misses.

thrashing = every
access misses.

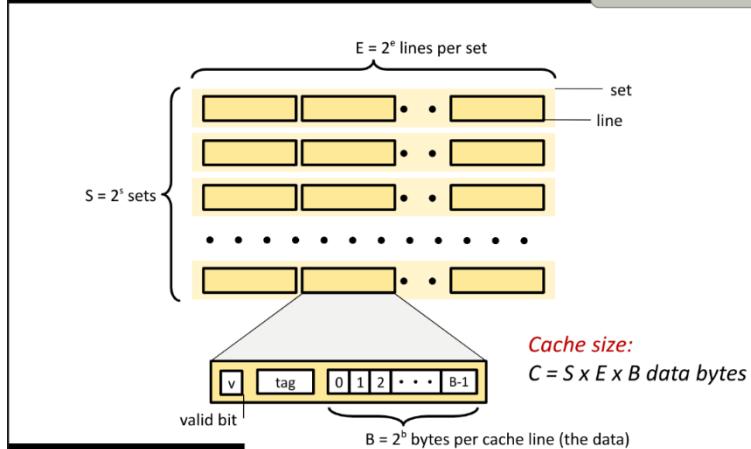
Examples of caching

Examples of Caching in the Hierarchy

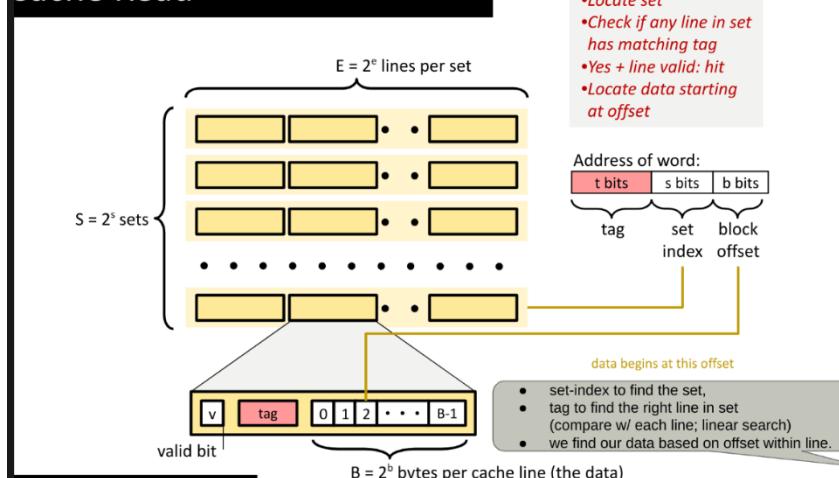
ex: access to memory
100x more expensive
than L1 cache.

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes line	On-Chip L1	1	Hardware
L2 cache	64-bytes line	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

General Cache Organization (S, E, B)

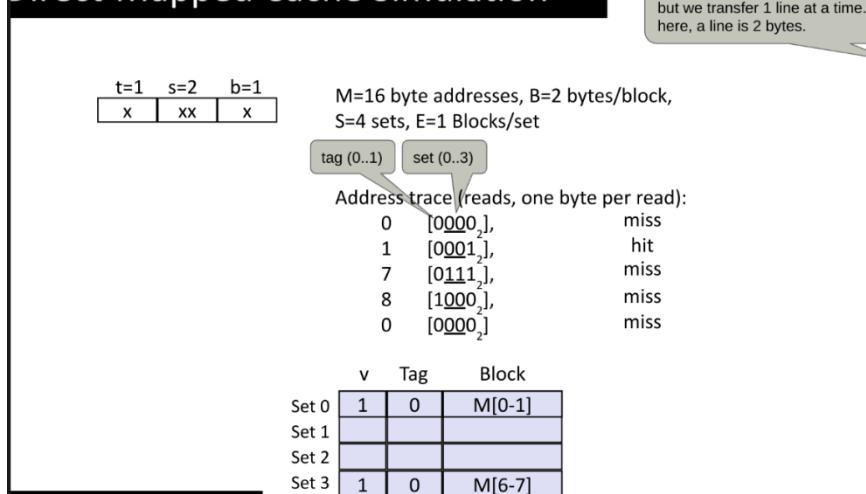


Cache Read

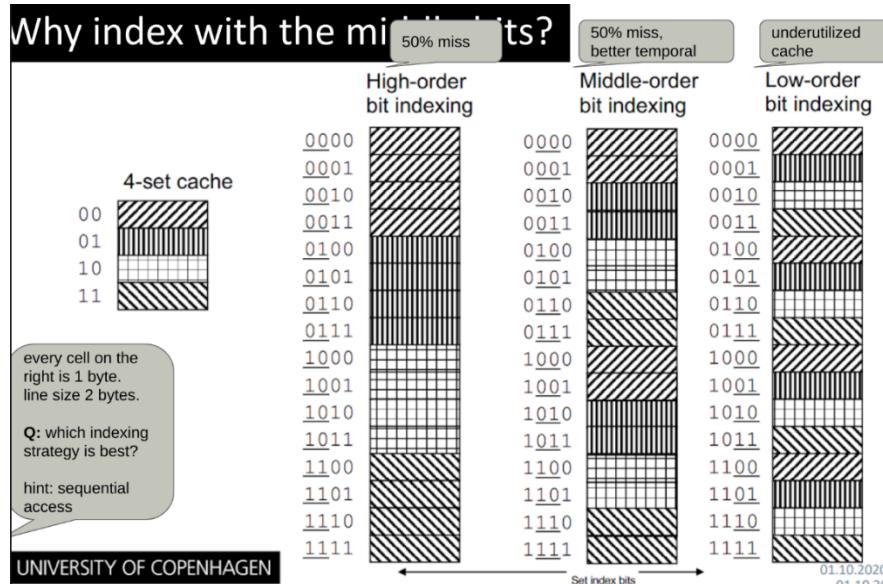


Direct-mapped cache example

Direct-Mapped Cache Simulation



Why we order by the middle bits



We need to have subsets in L1 taken from main memory, so we need a little bit of something, and not the whole of main memory

Writes

What about writes?

Multiple copies of data exist:

L1, L2, Main Memory, Disk

What to do on a write-hit?

Write-through (write immediately to memory)

faster

Write-back (defer write to memory until replacement of line)

- Need a dirty bit (line different from memory or not)

What to do on a write-miss?

Write-allocate (load into cache, update line in cache)

- Good if more writes to the location follow

No-write-allocate (writes immediately to memory)

Typical

Write-through + No-write-allocate

Write-back + Write-allocate

No-write allocate is faster, since it doesn't need to access the cache

Performance of a cache

Cache Performance Metrics

10s

- **Miss Rate**
 - Fraction of memory references not found in cache (misses / accesses)
= $1 - \text{hit rate}$
 - Typical numbers (in percentages):
 - 3-10% for L1
 - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
 - Time to deliver a line in the cache to the processor
 - includes time to determine whether the line is in the cache
 - Typical numbers:
 - 1-2 clock cycle for L1
 - 5-20 clock cycles for L2
- **Miss Penalty**
 - Additional time required because of a miss
 - typically 50-200 cycles for main memory (Trend: increasing!)

Hit rate scenario

Huge difference between a hit and a miss

Could be 100x, if just L1 and main memory

Would you believe 99% hits is twice as good as 97%?

Consider:
cache hit time of 1 cycle
miss penalty of 100 cycles

Average access time:
97% hits: 1 cycle + $0.03 * 100$ cycles = **4 cycles**
99% hits: 1 cycle + $0.01 * 100$ cycles = **2 cycles**

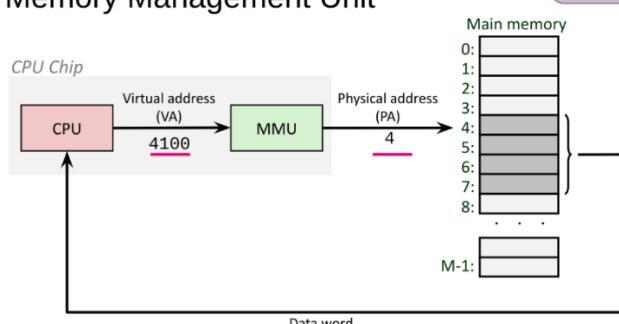
This is why “miss rate” is used instead of “hit rate”

Virtual memory

A System Using Virtual Addressing

MMU: Memory Management Unit

level of indirection:
maps virtual addresses
to physical addresses



Used in all modern servers, desktops, and laptops

One of the great ideas in computer science

MMU can do some key clever things. let's walk through that.

UNIVERSITY OF COPENHAGEN

When the CPU makes a request, and the MMU translates it into a physical address

Virtual memory

- Uses main memory efficiently **performance**
 - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
 - Each process gets the same uniform linear address space (from 0 and up)
- Isolates address spaces **security**
 - One process can't interfere with another's memory
 - User program cannot access privileged kernel information

Address space

An address space is an ordered set of contiguous addresses (non-negative integers)

Physical address space \Rightarrow associated to RAM

Virtual address space \Rightarrow associated to each **process**

UNIVERSITY OF COPENHAGEN

from point of view of process, it has access to whole memory, and that memory belongs to it. in actuality, it has access to parts of memory, some shared w/ other processes.

How does this enable caching?

VM is a Tool for Caching

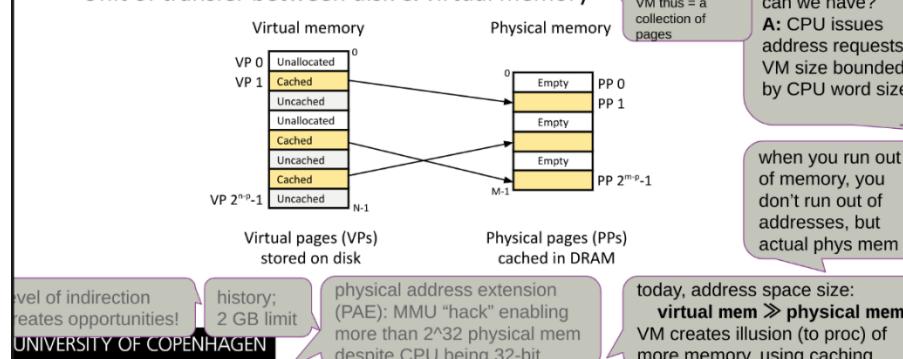
(i.e. for organizing how physical memory is used)

Virtual memory is an array of N contiguous bytes stored **on disk**.

The contents of the array on disk are **cached** in **physical memory (DRAM cache)**

These cache blocks are called **pages** (size (quanta) $P = 2^p$ bytes)

Unit of transfer between disk & virtual memory



Blocks of lines = pages

DRAM cache organization

DRAM Cache Organization

DRAM for phys. mem,
SRAM for registers

DRAM cache organization driven by the enormous miss penalty

DRAM is about **10x** slower than SRAM

Disk is about **10,000x** slower than DRAM

Consequences:

Large page (block) size: typically 4-8 KB, sometimes 4 MB

Fully associative

- Any VP can be placed in any PP
- Need a “large” mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms
- Too complicated and open-ended to be implemented in hardware

Write-back rather than write-through

Cannot use the (simple) mechanism we saw in CPU caching

UNIVERSITY OF COPENHAGEN

solution: page table (next slide)

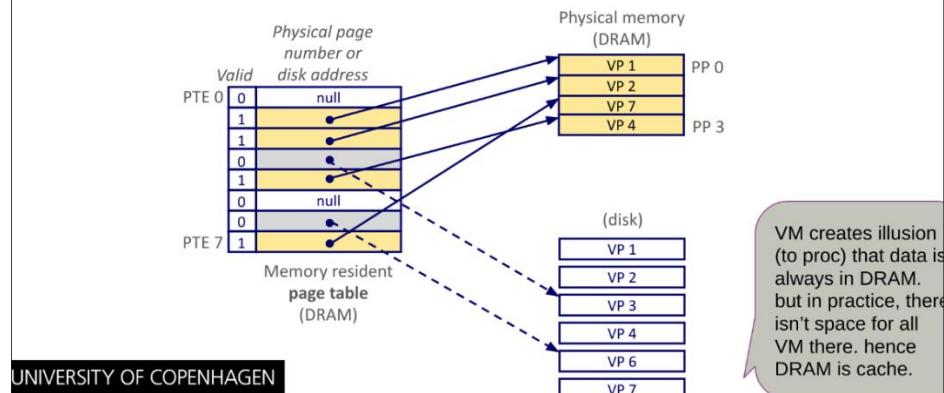
Page table

Page Tables

serves as our virt-phys address mapping

A **page table** is an array of page table entries (PTEs) that maps **virtual pages to physical pages**.

A per-process **kernel data structure** in DRAM



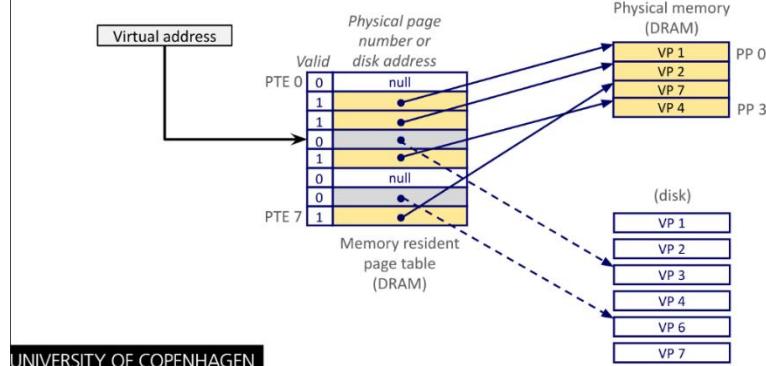
Array of page table entries (pages)

Virtual to physical mapping

Page Fault

what happens on a miss?
(bring the page to the cache)
implemented w/ exceptions.

Page fault: reference to VM word that is not in physical memory (DRAM cache miss)



Why does it work? Locality

Virtual memory **works** because of **locality**

At any point in time, programs tend to access a set of active virtual pages called the **working set**

Programs with better temporal locality will have smaller working sets

If (working set size < main memory size)

Good performance for one process after compulsory misses

If (SUM(working set sizes) > main memory size)

Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously (to/from disk; "swapping"; slow)

Address translation with page table

Address Translation With a Page Table

example now where we go through the whole translation.

(2) given a virt addr., part of it will be virt. page number (idx to page table)

Have a pointer to the page table

Offset is the same in virtual and physical

The page number in virtual points to the physical number

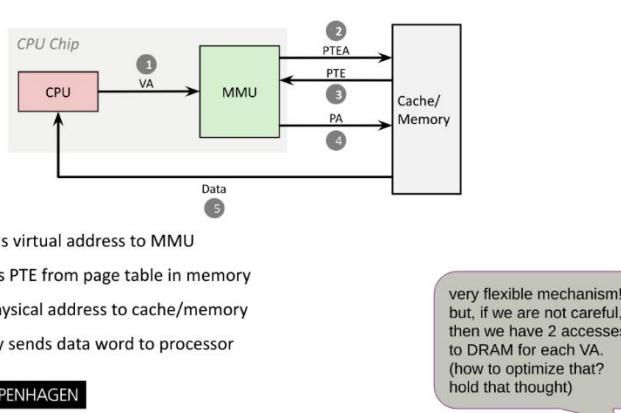
UNIVERSITY OF COPENHAGEN

(remember, byte-addressable)

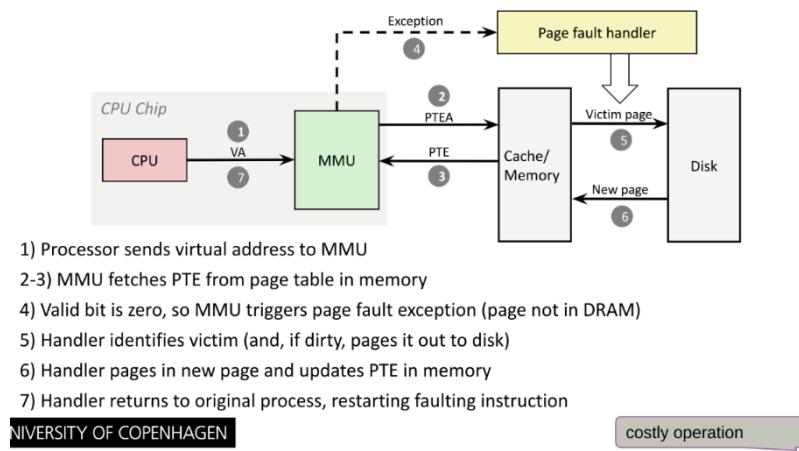
Page hit

Page fault

- Access something that is not in physical memory, but on disk



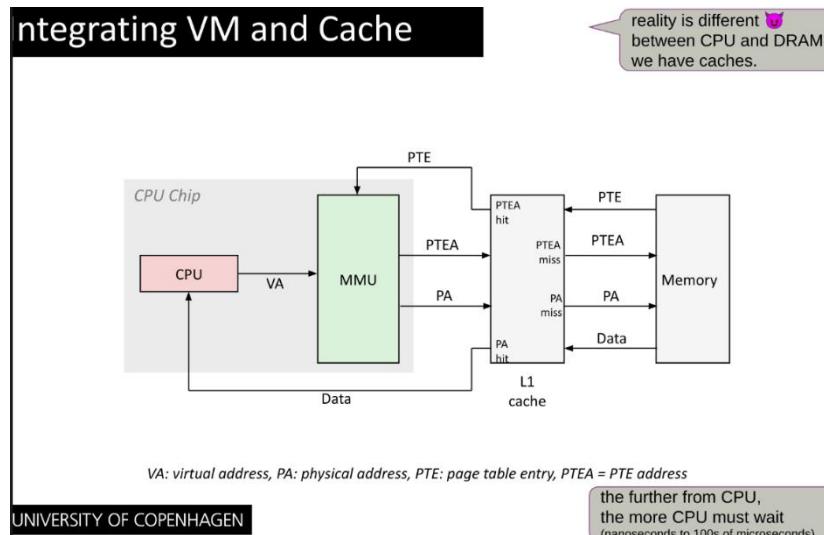
Page fault



Wants to access a byte, the MMU asks memory is the page there, no its not. Then page fault: exception gets generation - page fault handler - bring the right page from disk to memory and then the process that caused the page fault, will be redone. That access will

always succeed since its added to memory

Integrating VM and cache



There is a cache between MMU and memory

- Cache is very limited

Speeding up translation

Page table entries (PTEs) are cached in L1 like any other memory word

PTEs may be evicted by other data references

PTE hit still requires a small L1 delay.

we don't want to pollute the L1 cache w/ PTEs

Solution: *Translation Lookaside Buffer* (TLB)

Small hardware cache in MMU

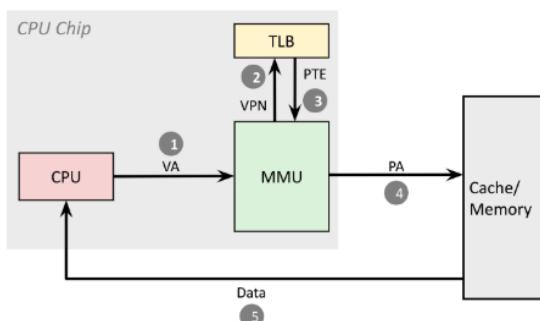
Maps virtual page numbers to physical page numbers

Contains complete page table entries for small number of pages

Store a certain number of pages near the CPU

TLB stores a subset

TLB hit



Ask the TLB whether the page is in memory or not

A TLB hit eliminates a memory access

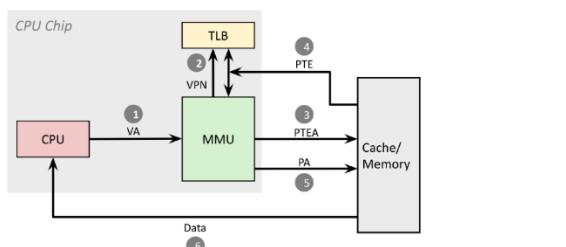
Y OF COPENHAGEN

nice and fast :-)

TLB miss

IISS

a way to quantify performance of your program is to monitor nr. of TLB misses. add locality to reduce



A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

Y OF COPENHAGEN

TLB misses are rare! why: locality

Take-aways

You should be able to describe:

- Address Space
- Physical and Virtual Memory
- MMU (its role, how work split between it and OS)
- Pages
- Page Table
- Translation Lookaside Buffer (TLB)

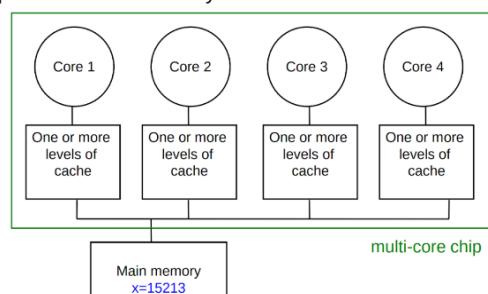
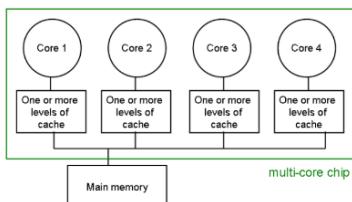
Virtual Memory as a tool for caching, memory management and memory protection.

Cache coherence

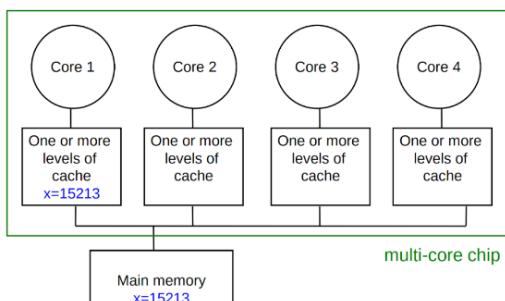
The Cache Coherence Problem

- Since we have private caches:
How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores

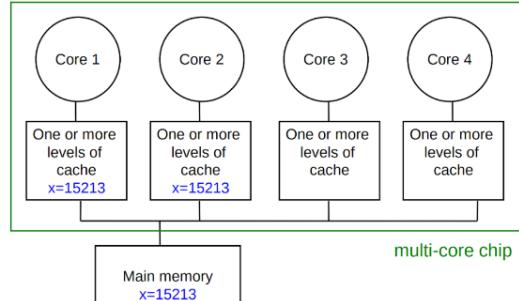
Suppose variable x initially contains 15213



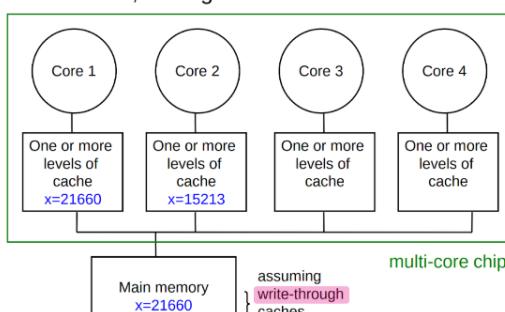
Core 1 reads x



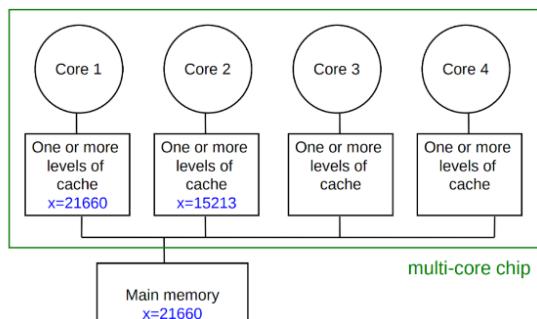
Core 2 reads x



Core 1 writes to x , setting it to 21660

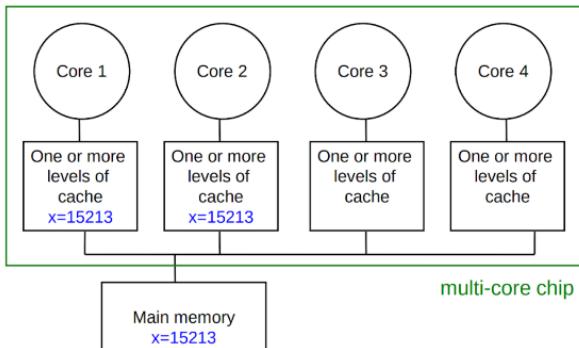


Core 2 attempts to read x ... gets a stale copy

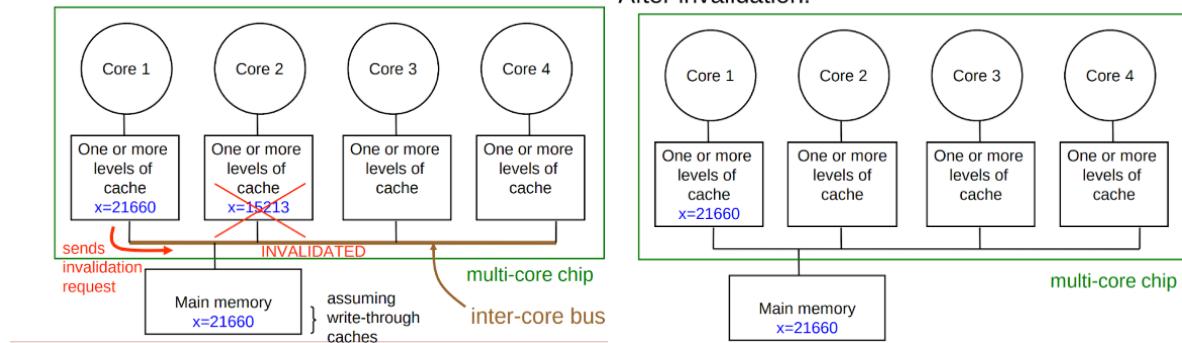


Invalidation Based Cache Coherence Protocol

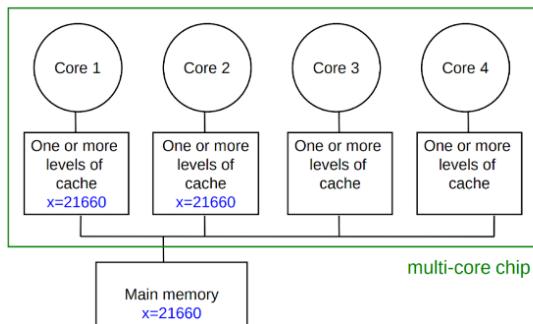
Revisited: Cores 1 and 2 have both read x



Core 1 writes to x, setting it to 21660

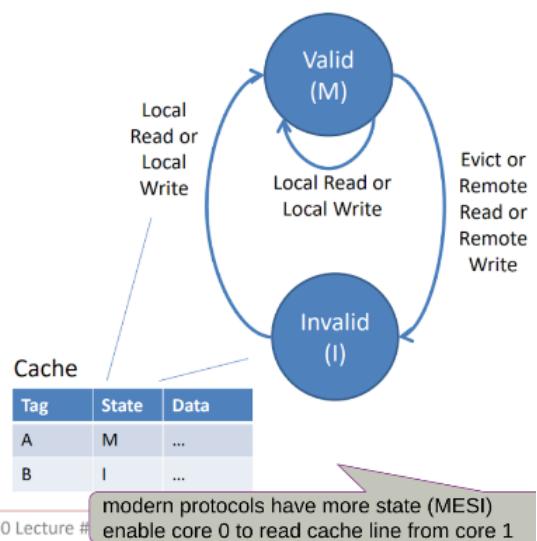


Core 2 reads x. Cache misses, and loads the new copy.



Minimal Coherence Protocol (Write-Back Cache)

- Blocks are always private or exclusive
- State transitions:
 - Local read: I->M, fetch, invalidate other copies
 - Local write: I->M, fetch, invalidate other copies
 - Evict: M->I, write back data
 - Remote read: M->I, write back data
 - Remote write: M->I, write back data



25/2017 (© J.P. Shen)

18-600 Lecture #

Prefetching

Have data at hand before it is needed

Hide the Memory Latency

- Many techniques have been proposed to further hide/tolerate the increasing memory latency.
 - For example
 - Caches
 - Locality optimization
 - Pipelining
 - Out-of-order execution
 - Multithreading
- Prefetching is one of the well studied techniques to hide memory latency.
 - Some prefetching schemes have been adopted in commercial processors.

How Prefetching Works?

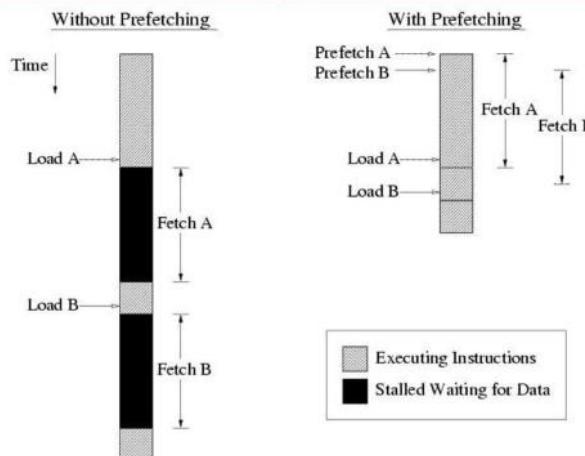


Figure 1.4: Illustration of how prefetching tolerates memory latency.

Array Prefetching Approaches

Basic Questions

1. When to initiate prefetches?

- Timely
 - Too early → replace other useful data (cache pollution) or be replaced before being used
 - Too late → cannot hide processor stall

2. Where to place prefetched data?

- Cache or dedicated buffer

3. What to be prefetched?

● Software-based

- Explicit "fetch" instructions
- Additional instructions executed

● Hardware-based

- Special hardware
- Unnecessary prefetchings (w/o compile-time information)

Side Effects and Requirements

● Side effects

- Prematurely prefetched blocks → possible "cache pollution"
- Removing processor stall cycles (increase memory request frequency);
- Unnecessary prefetchings → higher demand on memory bandwidth

● Requirements

- Timely
- Useful
- Low overhead

Hardware Data Prefetching

- No need for programmer or compiler intervention
- No changes to existing executables
- Take advantage of run-time information

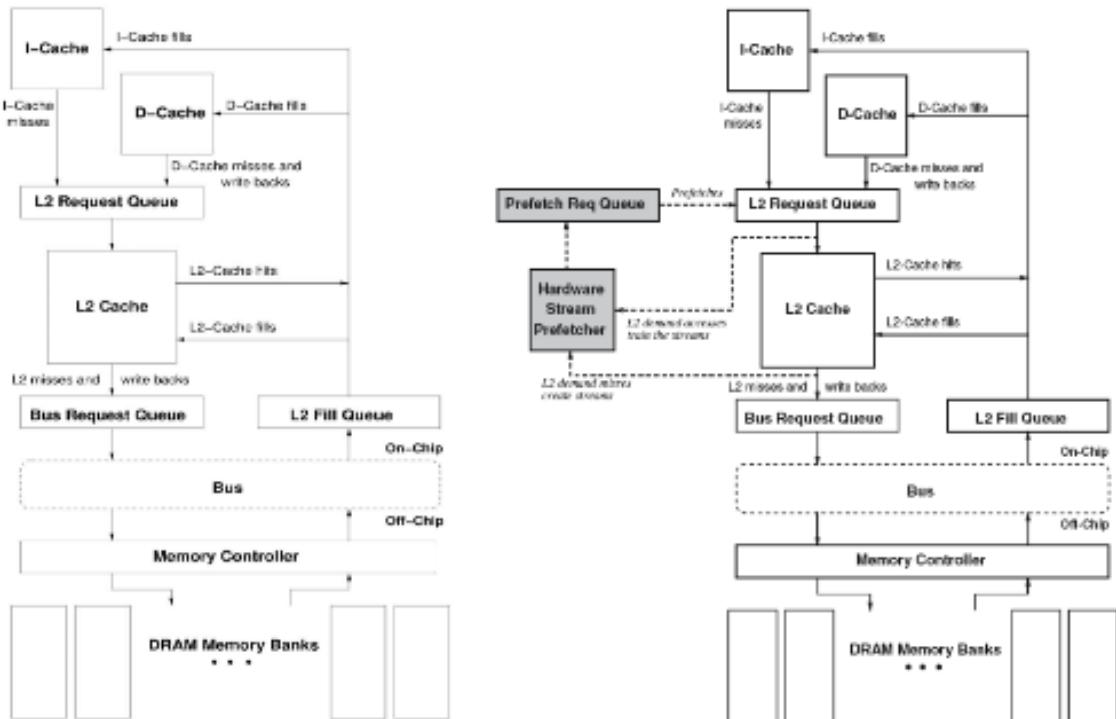
● E.g.,

- Alpha 21064 fetches 2 blocks on a miss
- Extra block placed in "[stream buffer](#)"
- On miss check stream buffer

● Works with data blocks too:

- Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
- Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-wav set associative caches

How a Prefetcher Fits in the Memory System



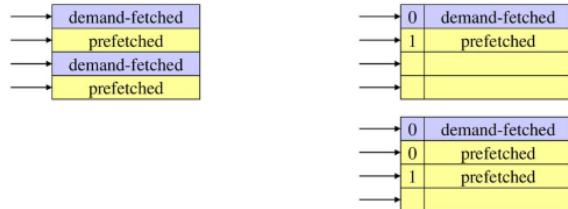
OBL Approaches

Sequential Prefetching

- Take advantage of spatial locality
- One block lookahead (OBL) approach**
 - Initiate a prefetch for block $b+1$ when block b is accessed
 - Prefetch-on-miss
 - Whenever an access for block b results in a cache miss
 - Tagged prefetch
 - Associates a tag bit with every memory block
 - When a block is demand-fetched or a prefetched block is referenced for the first time next block is fetched.
 - Used in HP PA7200

Prefetch-on-miss

Tagged prefetch



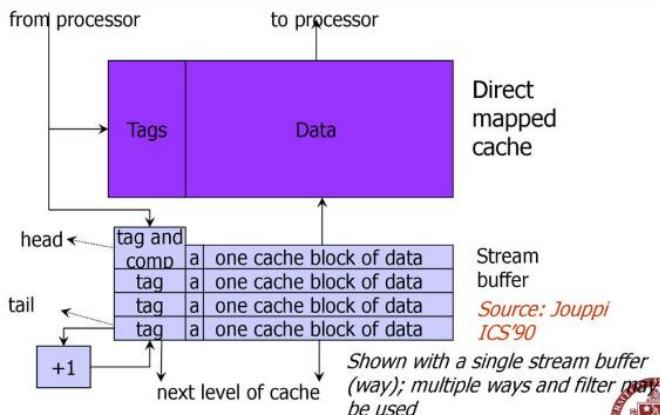
Degree of Prefetching

- OBL may not initiate prefetch far enough to avoid processor memory stall**
- Prefetch $K > 1$ subsequent blocks**
 - Additional traffic and cache pollution
- Adaptive sequential prefetching**
 - Vary the value of K during program execution
 - High spatial locality \rightarrow large K value
 - Prefetch efficiency metric
 - Periodically calculated
 - Ratio of useful prefetches to total prefetches

Stream Buffer

- K prefetched blocks \rightarrow FIFO stream buffer**
- As each buffer entry is referenced**
 - Move it to cache
 - Prefetch a new block to stream buffer
- Avoid cache pollution**

Stream Buffer Diagram



Sequential Prefetching

- Pros:**
 - No changes to executables
 - Simple hardware
- Cons:**
 - Only applies for good spatial locality
 - Works poorly for nonsequential accesses
 - Unnecessary prefetches for scalars and array accesses with large stride

Reference Prediction Table (RPT)

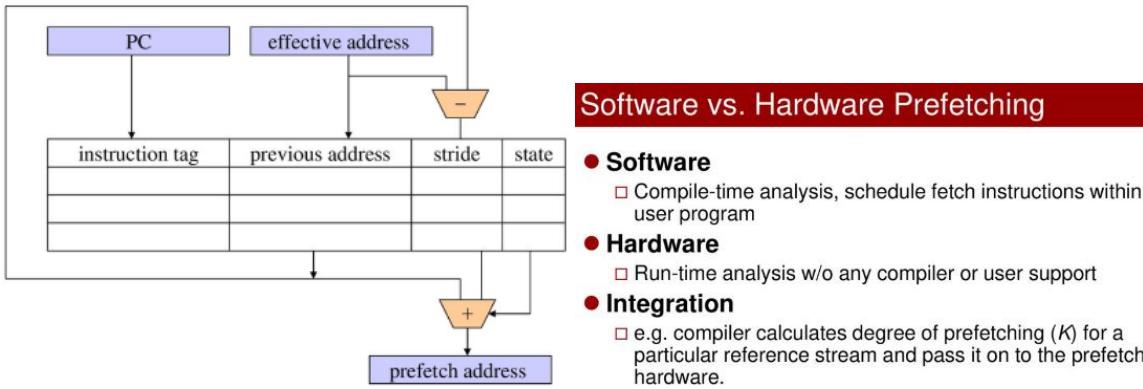
Prefetching with Arbitrary Strides

- Employ special logic to monitor the processor's address referencing pattern
- Detect constant stride array references originating from looping structures
- Compare successive addresses used by load or store instructions

- Hold information for the most recently used memory instructions to predict their access pattern**

- Address of the memory instruction
- Previous address accessed by the instruction
- Stride value
- State field

Organization of RPT



C

Reading

C guide and references

Size of a type into bytes

- Primitives

Pointers

```
int *x; /* points to an int */  
char *s; /* points to a char */  
float **w; /* points to a pointer that points to a float */  
float ***a; /* points to a pointer that points to a pointer that points to a float */
```

All pointers have the same size

Compound types

- Struct
- Array

We declare variables exactly as we would use them

You can do bad things

C does not try to prevent you from doing bad things.

```
float x = 123.567f; /* A floating-point number */
int y = *((int *)&x); /* An integer made from the same bytes as the floating-point number */

int z[4]; /* An array of 4 integers */
int w = z[254]; /* An integer made from the contents of memory 1000 bytes after the end of z */

const char *s = "hi"; /* compiler makes the string in memory the OS won't allow us to change */
char *t = (char *)s; /* we get a pointer to that memory that C will allow us to change */
t[0] = 'H'; /* we try to change that memory (the OS will crash our program) */
```

C's general attitude is "every rule has an exception" and "the programmer knows best". It might make you do some complicated casting to do things, but it won't stop you if you are determined.

Inside a function

- Declaring a local variable as static makes it a global variable that only that one function can see.

Declaring a global variable as static makes it visible only inside that .c file.

Declaring a global variable as extern tells the compiler "assume it exists, don't create it. It will be supplied by a different .c file. Connect the two during linking."

Compiler intrinsics

What is the synonym of intrinsic?

built-in, constitutional, inbuilt, inherent, integral. existing as an essential constituent or characteristic. inner, internal, intimate. innermost or essential.

The intrinsics are required on 64-bit architectures where inline assembly is not supported.

- In computer programming, an inline assembler is a feature of some compilers that allows low-level code written in assembly language to be embedded within a program

Some intrinsics, such as __assume and __ReadWriteBarrier, provide information to the compiler, which affects the behaviour of the optimizer.

Some intrinsics are available only as intrinsics, and some are available both in function and intrinsic implementations

`#pragma function(intrinsic-function-name-list)`

C-to-assembly

Dive into Systems - ch 7

(tables 31-34, §7.4-7.9(skip 7.5.2))

Table 31. Example Instruction Suffixes

Suffix	C Type	Size (bytes)
b	char	1
w	short	2
l	int or unsigned	4
s	float	4
q	long, unsigned long, all pointers	8
d	double	8

Table 32. Most Common Instructions

Instruction	Translation
mov S, D	S → D (copies value of S into D)
add S, D	S + D → D (adds S to D and stores result in D)
sub S, D	D - S → D (subtracts S from D and stores result in D)

Table 33. Stack Management Instructions

Instruction	Translation
push S	Pushes a copy of S onto the top of the stack. Equivalent to: <code>sub \$0x8, %rsp mov S, (%rsp)</code>
pop D	Pops the top element off the stack and places it in location D. Equivalent to: <code>mov (%rsp), D add \$0x8, %rsp</code>

Table 34. Common Arithmetic Instructions.

Instruction	Translation
add S, D	$S + D \rightarrow D$
sub S, D	$D - S \rightarrow D$
inc D	$D + 1 \rightarrow D$
dec D	$D - 1 \rightarrow D$
neg D	$-D \rightarrow D$
imul S, D	$S \times D \rightarrow D$
idiv S	$\%rax / S: \text{quotient} \rightarrow \%rax, \text{remainder} \rightarrow \%rdx$

Conditionals and loops

Preliminaries

Basic instructions associated with conditional control

Table 38. Conditional Control Instructions

Instruction	Translation
cmp R1, R2	Compares R2 with R1 (i.e., evaluates $R2 - R1$)
test R1, R2	Computes $R1 \& R2$

Test performs bitwise AND.

test %rax, %rax

In this example, the bitwise AND of %rax with itself is zero only when %rax contains zero. In other words, this is a test for a zero value and is equivalent to:

```
cmp $0, %rax
```

Cmp and test does not modify destination register, only a series of single-bit values: condition code flags

cmp will modify condition code flags based on whether the value R2 - R1 results in a positive (greater), negative (less), or zero (equal) value

Table 39. Common Condition Code Flags.

Flag	Translation
ZF	Is equal to zero (1: yes; 0: no)
SF	Is negative (1: yes; 0: no)
OF	Overflow has occurred (1:yes; 0: no)
CF	Arithmetic carry has occurred (1: yes; 0: no)

The SF and OF flags are used for comparison operations on signed integers, whereas the CF flag is used for comparisons on unsigned integers

Jump instructions

A jump instruction enables a program's execution to "jump" to a new position in the code

\$rip always points to the next instruction in program memory

Table 40. Direct Jump Instructions

Instruction	Description
jmp L	Jump to location specified by L
jmp *addr	Jump to specified address

L is a symbolic label -> identifiers for the programs object file

Table 41. Conditional Jump Instructions; Synonyms Shown in Parentheses

Signed Comparison	Unsigned Comparison	Description
je (jz)		jump if equal (==) or jump if zero
jne (jnz)		jump if not equal (!=)
js		jump if negative
jns		jump if non-negative
jg (jnle)	ja (jnbe)	jump if greater (>)
jge (jnl)	jae (jnb)	jump if greater than or equal (>=)
jl (jnge)	jb (jnae)	jump if less (<)
jle (jng)	jbe (jna)	jump if less than or equal (<=)

Table 42 - letters commonly found in jump

Table 42. Jump Instruction Suffixes.

Letter	Word
j	jump
n	not
e	equal
s	signed
g	greater (signed interpretation)
l	less (signed interpretation)
a	above (unsigned interpretation)
b	below (unsigned interpretation)

Goto statement

Reverse conditionals and loops in assembly back to C

The goto statement is a C primitive that forces program execution to switch to another line in the code. The assembly instruction associated with the goto statement is jmp.

The goto statement consists of the goto keyword followed by a **goto label**, a type of program label that indicates where execution should continue. So, goto done means that the program execution should jump to the line marked by label done

Table 43. Comparison of a C function and its associated goto form.

Regular C version	Goto version
<pre>int getSmallest(int x, int y) { int smallest; if (x > y) { //if (conditional) smallest = y; //then statement } else { smallest = x; //else statement } return smallest; }</pre>	<pre>int getSmallest(int x, int y) { int smallest; if (x <= y) { //if (!conditional) goto else_statement; } smallest = y; //then statement goto done; else_statement: smallest = x; //else statement done: return smallest; }</pre>

If statements

Let's take a look at the `getSmallest` function in assembly. For convenience, the function is reproduced below.

```
int getSmallest(int x, int y) {
    int smallest;
    if (x > y) {
        smallest = y;
    }
    else {
        smallest = x;
    }
    return smallest;
}
```

The corresponding assembly code extracted from GDB looks similar to the following:

```
(gdb) disas getSmallest
Dump of assembler code for function getSmallest:
0x40059a <+4>: mov    %edi,-0x14(%rbp)
0x40059d <+7>: mov    %esi,-0x18(%rbp)
0x4005a0 <+10>: mov    -0x14(%rbp),%eax
0x4005a3 <+13>: cmp    -0x18(%rbp),%eax
0x4005a6 <+16>: jle    0x4005b0 <getSmallest+26>
0x4005a8 <+18>: mov    -0x18(%rbp),%eax
0x4005ae <+24>: jmp    0x4005b9 <getSmallest+35>
0x4005b0 <+26>: mov    -0x14(%rbp),%eax
0x4005b9 <+35>: pop    %rbp
0x4005ba <+36>: retq
```

- The first mov instruction copies the value located in register %edi (the first parameter, x) and places it at memory location %rbp-0x14 on the call stack. The instruction pointer (%rip) is set to the address of the next instruction, or 0x40059d.
- The second mov instruction copies the value located in register %esi (the second parameter, y) and places it at memory location %rbp-0x18 on the call stack. The instruction pointer (%rip) updates to point to the address of the next instruction, or 0x4005a0.
- The third mov instruction copies x to register %eax. Register %rip updates to point to the address of the next instruction in sequence.
- The cmp instruction compares the value at location %rbp-0x18 (the second parameter, y) to x and sets appropriate condition code flag registers. Register %rip advances to the address of the next instruction, or 0x4005a6.
- The jle instruction at address 0x4005a6 indicates that if x is less than or equal to y, the next instruction that should execute should be at location <getSmallest+26> and that %rip should be set to address 0x4005b0. Otherwise, %rip is set to the next instruction in sequence, or 0x4005a8.

We can then annotate the preceding assembly as follows:

```

0x40059a <+4>:  mov %edi,-0x14(%rbp)      # copy x to %rbp-0x14
0x40059d <+7>:  mov %esi,-0x18(%rbp)      # copy y to %rbp-0x18
0x4005a0 <+10>: mov -0x14(%rbp),%eax       # copy x to %eax
0x4005a3 <+13>: cmp -0x18(%rbp),%eax       # compare x with y
0x4005a6 <+16>: jle 0x4005b0 <getSmallest+26> # if x<=y goto <getSmallest+26>
0x4005a8 <+18>: mov -0x18(%rbp),%eax       # copy y to %eax
0x4005ae <+24>: jmp 0x4005b9 <getSmallest+35> # goto <getSmallest+35>
0x4005b0 <+26>: mov -0x14(%rbp),%eax       # copy x to %eax
0x4005b9 <+35>: pop %rbp                    # restore %rbp (clean up stack)
0x4005ba <+36>: retq                      # exit function (return %eax)

```

Translating this back to C code yields:

Table 44. Translating `getSmallest()` into goto C form and C code.

goto Form	Translated C code
<pre>int getSmallest(int x, int y) { int smallest; if (x <= y) { goto assign_x; } smallest = y; goto done; assign_x: smallest = x; done: return smallest; }</pre>	<pre>int getSmallest(int x, int y) { int smallest; if (x <= y) { smallest = x; } else { smallest = y; } return smallest; }</pre>

Table 45. Standard if statement format and its equivalent goto form.

C if statement	Compiler's equivalent goto form
<pre>if (condition) { then_statement; } else { else_statement; }</pre>	<pre>if (!condition) { goto else; } then_statement; goto done; else: else_statement; done:</pre>

Conditional move

Cmp, test and jmp

The use of C's **ternary expression** often results in the compiler generating a cmov instruction in place of jumps. For the standard if-then-else statement, the ternary expression has the form:

```
result = (condition) ? then_statement : else_statement;
```

Then we can rewrite code, that functions exactly the same as before

```
int getSmallest_cmov(int x, int y) {

    return x > y ? y : x;
```

}

```

0x4005d7 <+0>: push %rbp      #save %rbp
0x4005d8 <+1>: mov %rsp,%rbp    #update %rbp
0x4005db <+4>: mov %edi,-0x4(%rbp) #copy x to %rbp-0x4
0x4005de <+7>: mov %esi,-0x8(%rbp) #copy y to %rbp-0x8
0x4005e1 <+10>: mov -0x8(%rbp),%eax #copy y to %eax
0x4005e4 <+13>: cmp %eax,-0x4(%rbp) #compare x and y
0x4005e7 <+16>: cmovle -0x4(%rbp),%eax #if (x <=y) copy x to %eax
0x4005eb <+20>: pop %rbp      #restore %rbp
0x4005ec <+21>: retq        #return %eax

```

It has no jumps

Table 46. The cmov Instructions.

Signed	Unsigned	Description
cmove (cmovz)		move if equal (==)
cmovne (cmovnz)		move if not equal (!=)
cmovs		move if negative
cmovns		move if non-negative
cmovg (cmovnle)	cmova (cmovnbe)	move if greater (>)
cmovge (cmovnl)	cmovae (cmovnb)	move if greater than or equal (>=)
cmovl (cmovng)	cmovb (cmovnae)	move if less (<)
cmovle (cmovng)	cmovbe (cmovna)	move if less than or equal (<=)

Table 47. Two functions that attempt to increment the value of integer x .

C code	C ternary form
<pre> int incrementX(int *x) { if (x != NULL) { //if x is not NULL return (*x)++; //increment x } else { //if x is NULL return 1; //return 1 } } </pre>	<pre> int incrementX2(int *x){ return x ? (*x)++ : 1; } </pre>

It is tempting to think that `incrementX2` uses a `cmove` instruction since it uses a ternary expression. However, both functions yield the exact same assembly code:

```
0x4005ed <+0>: push %rbp
0x4005ee <+1>: mov %rsp,%rbp
0x4005f1 <+4>: mov %rdi,-0x8(%rbp)
0x4005f5 <+8>: cmpq $0x0,-0x8(%rbp)
0x4005fa <+13>: je 0x40060d <incrementX+32>
0x4005fc <+15>: mov -0x8(%rbp),%rax
0x400600 <+19>: mov (%rax),%eax
0x400602 <+21>: lea 0x1(%rax),%ecx
0x400605 <+24>: mov -0x8(%rbp),%rdx
0x400609 <+28>: mov %ecx,(%rdx)
0x40060b <+30>: jmp 0x400612 <incrementX+37>
0x40060d <+32>: mov $0x1,%eax
0x400612 <+37>: pop %rbp
0x400613 <+38>: retq
```

Loops in assembly

```
int sumUp(int n) {
    //initialize total and i
    int total = 0;
    int i = 1;

    while (i <= n) { //while i is less than or equal to n
        total += i; //add i to total
        i++; //increment i by 1
    }
    return total;
}
```

Compiling this code and disassembling it using GDB yields the following assembly code:

```
Dump of assembler code for function sumUp:
0x400526 <+0>: push %rbp
0x400527 <+1>: mov %rsp,%rbp
0x40052a <+4>: mov %edi,-0x14(%rbp)
0x40052d <+7>: mov $0x0,-0x8(%rbp)
0x400534 <+14>: mov $0x1,-0x4(%rbp)
0x40053b <+21>: jmp 0x400547 <sumUp+33>
0x40053d <+23>: mov -0x4(%rbp),%eax
0x400540 <+26>: add %eax,-0x8(%rbp)
0x400543 <+29>: add $0x1,-0x4(%rbp)
0x400547 <+33>: mov -0x4(%rbp),%eax
0x40054a <+36>: cmp -0x14(%rbp),%eax
0x40054d <+39>: jle 0x40053d <sumUp+23>
0x40054f <+41>: mov -0x8(%rbp),%eax
0x400552 <+44>: pop %rbp
0x400553 <+45>: retq
```

Table 48. Translating sumUp into goto C form.

Assembly	Translated goto Form
<pre> <sumUp>: <+0>: push %rbp <+1>: mov %rsp,%rbp <+4>: mov %edi,-0x14(%rbp) <+7>: mov \$0x0,-0x8(%rbp) <+14>: mov \$0x1,-0x4(%rbp) <+21>: jmp 0x400547 <sumUp+33> <+23>: mov -0x4(%rbp),%eax <+26>: add %eax,-0x8(%rbp) <+29>: add \$0x1,-0x4(%rbp) <+33>: mov -0x4(%rbp),%eax <+36>: cmp -0x14(%rbp),%eax <+39>: jle 0x40053d <sumUp+23> <+41>: mov -0x8(%rbp),%eax <+44>: pop %rbp <+45>: retq </pre>	<pre> int sumUp(int n) { int total = 0; int i = 1; goto start; body: total += i; i += 1; start: if (i <= n) { goto body; } return total; } </pre>

The primary loop in the `sumUp` function can also be written as a `for` loop:

```

int sumUp2(int n) {
    int total = 0;           //initialize total to 0
    int i;
    for (i = 1; i <= n; i++) { //initialize i to 1, increment by 1 while i<=n
        total += i;          //updates total by i
    }
    return total;
}

```

This version yields assembly code identical to our `while` loop example. We repeat the assembly code below and annotate each line with its English translation:

```

Dump of assembler code for function sumUp2:
0x400554 <+0>: push %rbp      #save %rbp
0x400555 <+1>: mov %rsp,%rbp   #update %rbp (new stack frame)
0x400558 <+4>: mov %edi,-0x14(%rbp) #copy %edi to %rbp-0x14 (n)
0x40055b <+7>: movl $0x0,-0x8(%rbp) #copy 0 to %rbp-0x8 (total)
0x400562 <+14>: movl $0x1,-0x4(%rbp) #copy 1 to %rbp-0x4 (i)
0x400569 <+21>: jmp 0x400575 <sumUp2+33> #goto <sumUp2+33>
0x40056b <+23>: mov -0x4(%rbp),%eax #copy i to %eax [loop]
0x40056e <+26>: add %eax,-0x8(%rbp) #add i to total (total+=i)
0x400571 <+29>: addl $0x1,-0x4(%rbp) #add 1 to i (i++)
0x400575 <+33>: mov -0x4(%rbp),%eax #copy i to %eax [start]
0x400578 <+36>: cmp -0x14(%rbp),%eax #compare i with n
0x40057b <+39>: jle 0x40056b <sumUp2+23> #if (i <= n) goto loop
0x40057d <+41>: mov -0x8(%rbp),%eax #copy total to %eax
0x400580 <+44>: pop %rbp      #prepare to leave the function
0x400581 <+45>: retq       #return total

```

loop representation

for (<initialization>; <boolean expression>; <step>) {

<body>

}

<initialization>

while (<boolean expression>) {

<body>

```
<step>
```

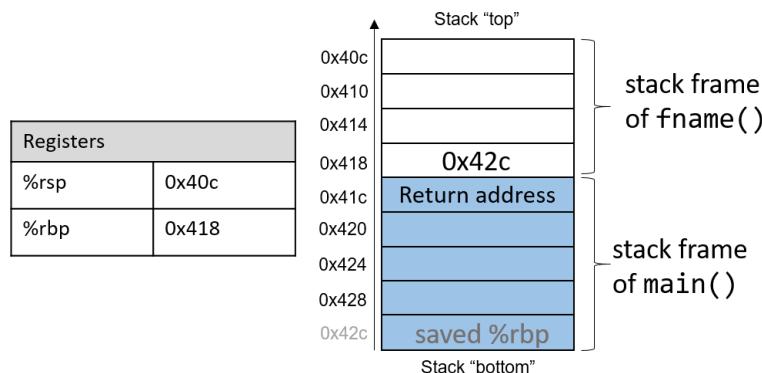
```
}
```

Table 49. Equivalent ways to write the sumUp function.

For loop	While loop
<pre>int sumUp2(int n) { int total = 0; int i = 1; for (i; i <= n; i++) { total += i; } return total; }</pre>	<pre>int sumUp(int n){ int total = 0; int i = 1; while (i <= n) { total += i; i += 1; } return total; }</pre>

Functions in assembly

%rsp is the **stack pointer** and always points to the top of the stack. The register %rbp represents the base pointer (also known as the **frame pointer**) and points to the base of the current stack frame. The **stack frame** (also known as the **activation frame** or the **activation record**) refers to the portion of the stack allocated to a single function call. The currently executing function is always at the top of the stack, and its stack frame is referred to as the **active frame**. The active frame is bounded by the stack pointer (at the top of stack) and the frame pointer (at the bottom of the frame). The activation record typically holds local variables for a function.



The return address points to code segment memory, not stack memory

Recall that the call stack region (stack memory) of a program is different from its code region (code segment memory). While `%rbp` and `%rsp` point to addresses in the stack memory, `%rip` points to an address in *code* segment memory. In other words, the return address is an address in code segment memory, not stack memory:

Parts of Program Memory

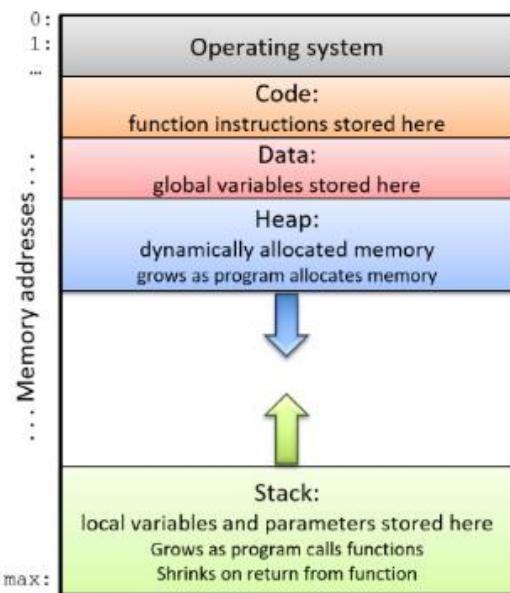


Figure 95. The parts of a program's address space

Table 50. Common Function Management Instructions

Instruction	Translation
<code>leaveq</code>	Prepares the stack for leaving a function. Equivalent to: <code>mov %rbp, %rsp pop %rbp</code>
<code>callq addr <fname></code>	Switches active frame to callee function. Equivalent to: <code>push %rip mov addr, %rip</code>
<code>retq</code>	Restores active frame to caller function. Equivalent to: <code>pop %rip</code>

Table 51. Locations of Function Parameters.

Parameter	Location
Parameter 1	%rdi
Parameter 2	%rsi
Parameter 3	%rdx
Parameter 4	%rcx
Parameter 5	%r8
Parameter 6	%r9
Parameter 7+	on call stack

[Figure 96](#) shows the execution stack immediately prior to the execution of `main`.

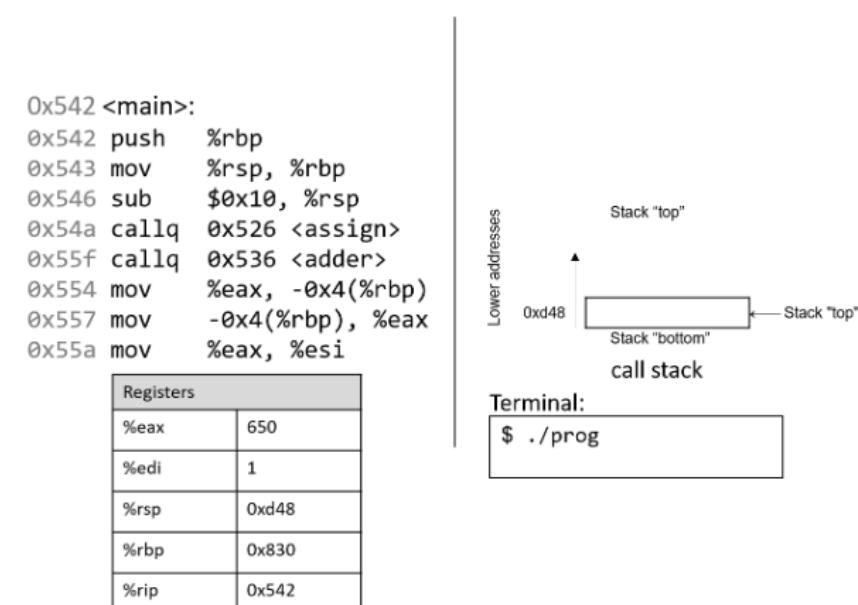


Figure 96. The initial state of the CPU registers and call stack prior to executing the main function

Stack grows toward lower addresses.

Walkthrough of calling `main` in 7.5.3.

Recursion

Table 52. Iterative version (`sumDown`) and recursive version (`sumr`)

Iterative	Recursive
<pre>int sumDown(int n) { int total = 0; int i = n; while (i > 0) { total += i; i--; } return total; }</pre>	<pre>int sumr(int n) { if (n <= 0) { return 0; } return n + sumr(n-1); }</pre>

The base case in the recursive function `sumr` accounts for any values of n that are less than one. The recursive step calls `sumr` with the value $n-1$ and adds the result to n prior to returning. Compiling `sumr` and disassembling it with GDB yields the following assembly code:

```
Dump of assembler code for function sumr:
0x400551 <+0>: push %rbp          # save %rbp
0x400552 <+1>: mov %rsp,%rbp      # update %rbp (new stack frame)
0x400555 <+4>: sub $0x10,%rsp      # expand stack frame by 16 bytes
0x400559 <+8>: mov %edi,-0x4(%rbp) # move first param (n) to %rbp-0x4
0x40055c <+11>: cmp $0x0,-0x4(%rbp) # compare n to 0
0x400560 <+15>: jg 0x400569 <sumr+24> # if (n > 0) goto <sumr+24> [body]
0x400562 <+17>: mov $0x0,%eax      # copy 0 to %eax
0x400567 <+22>: jmp 0x40057d <sumr+44> # goto <sumr+44> [done]
0x400569 <+24>: mov -0x4(%rbp),%eax    # copy n to %eax (result = n)
0x40056c <+27>: sub $0x1,%eax      # subtract 1 from %eax (result -= 1)
0x40056f <+30>: mov %eax,%edi      # copy %eax to %edi
0x400571 <+32>: callq 0x400551 <sumr>   # call sumr(result)
0x400576 <+37>: mov %eax,%edx      # copy returned value to %edx
0x400578 <+39>: mov -0x4(%rbp),%eax    # copy n to %eax
0x40057b <+42>: add %edx,%eax      # add sumr(result) to n
0x40057d <+44>: leaveq             # prepare to leave the function
0x40057e <+45>: retq               # return result
```

Table 53. C goto form and translation of `sumr` assembly code

C goto form	C version without goto statements
<pre>int sumr(int n) { int result; if (n > 0) { goto body; } result = 0; goto done; body: result = n; result -= 1; result = sumr(result); result += n; done: return result; }</pre>	<pre>int sumr(int n) { int result; if (n <= 0) { return 0; } result = sumr(n-1); result += n; return result; }</pre>

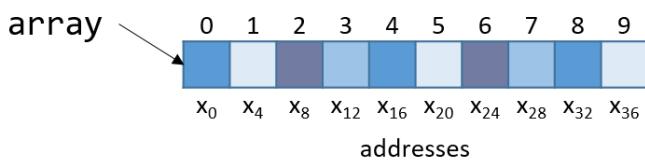
7.6.1 contains animation for call stack changes

Arrays

Table 54. Common Array Operations and Their Corresponding Assembly Representations

Operation	Type	Assembly Representation
<code>x = arr</code>	<code>int *</code>	<code>mov %rdx, %rax</code>
<code>x = arr[0]</code>	<code>int</code>	<code>mov (%rdx), %eax</code>
<code>x = arr[i]</code>	<code>int</code>	<code>mov (%rdx, %rcx, 4), %eax</code>
<code>x = &arr[3]</code>	<code>int *</code>	<code>lea 0xc(%rdx), %rax</code>
<code>x = arr+3</code>	<code>int *</code>	<code>lea 0xc(%rdx), %rax</code>
<code>x = *(arr+5)</code>	<code>int</code>	<code>mov 0x14(%rdx), %eax</code>

10-integer array in memory, each x represents four bytes



To compute the address of element 3, the compiler multiplies the index 3 by the data size of the integer type (4) to yield an offset of 12 (or 0xc). Sure enough, element 3 in [Figure 97](#) is located at byte offset x12.

```
int sumArray(int *array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];
    }
    return total;
}
```

0x400686 <+0>:	push %rbp # save %rbp
0x400687 <+1>:	mov %rsp,%rbp # update %rbp (new stack frame)
0x40068a <+4>:	mov %rdi,-0x18(%rbp) # copy array to %rbp-0x18
0x40068e <+8>:	mov %esi,-0x1c(%rbp) # copy length to %rbp-0x1c
0x400691 <+11>:	movl \$0x0,-0x4(%rbp) # copy 0 to %rbp-0x4 (total)
0x400698 <+18>:	movl \$0x0,-0x8(%rbp) # copy 0 to %rbp-0x8 (i)
0x40069f <+25>:	jmp 0x4006be <sumArray+56> # goto <sumArray+56>
0x4006a1 <+27>:	mov -0x8(%rbp),%eax # copy i to %eax
0x4006a4 <+30>:	cltq # convert i to a 64-bit integer
0x4006a6 <+32>:	lea 0x0(%rax,4),%rdx # copy i*4 to %rdx
0x4006ae <+40>:	mov -0x18(%rbp),%rax # copy array to %rax
0x4006b2 <+44>:	add %rdx,%rax # compute array+i*4, store in %rax
0x4006b5 <+47>:	mov (%rax),%eax # copy array[i] to %eax
0x4006b7 <+49>:	add %eax,-0x4(%rbp) # add %eax to total
0x4006ba <+52>:	addl \$0x1,-0x8(%rbp) # add 1 to i (i+=1)
0x4006be <+56>:	mov -0x8(%rbp),%eax # copy i to %eax

```

0x4006c1 <+59>:    cmp -0x1c(%rbp),%eax    # compare i to length
0x4006c4 <+62>:    jl 0x4006a1 <sumArray+27> # if i<length goto <sumArray+27>
0x4006c6 <+64>:    mov -0x4(%rbp),%eax    # copy total to %eax
0x4006c9 <+67>:    pop %rbp        # prepare to leave the function
0x4006ca <+68>:    retq          # return total

```

Let's take a closer look at the five instructions between locations `<sumArray+32>` and `<sumArray+49>`:

```

<+32>: lea 0x0(%rax,4),%rdx      # copy i*4 to %rdx
<+40>: mov -0x18(%rbp),%rax      # copy array to %rax
<+44>: add %rdx,%rax            # add i*4 to array (i.e. array+i) to %rax
<+47>: mov (%rax),%eax          # dereference array+i*4, place in %eax
<+49>: add %eax,-0x4(%rbp)       # add %eax to total (i.e. total+=array[i])

```

Matrices

Two dimensional array

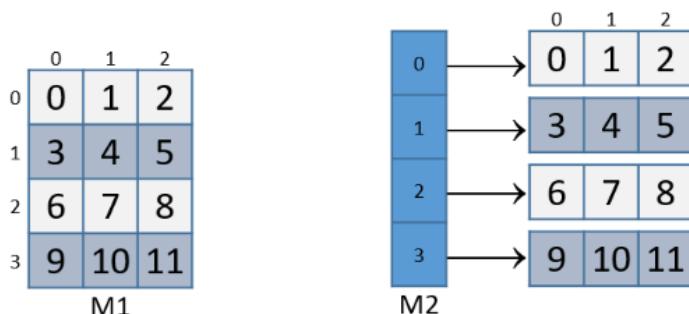
```

//statically allocated matrix (allocated on stack)
int M1[4][3];

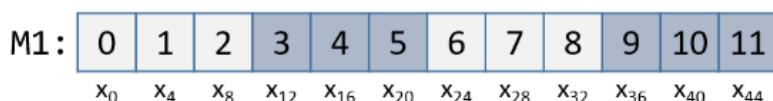
//dynamically allocated matrix (programmer friendly, allocated on heap)
int **M2, i;
M2 = malloc(4 * sizeof(int *));
for (i = 0; i < 4; i++) {
    M2[i] = malloc(3 * sizeof(int));
}

```

In the case of the dynamically allocated matrix, the main array contains a contiguous array of `int` pointers. Each integer pointer points to a different array in memory. [Figure 98](#) illustrates how we would normally visualize each of these matrices.



[Figure 98. Illustration of a statically allocated \(M1\) and a dynamically allocated \(M2\) 3x4 matrix](#)



[Figure 99. Matrix M1's memory layout in row-major order](#)

```

int sumMat(int *m, int rows, int cols) {
    int i, j, total = 0;
    for (i = 0; i < rows; i++){
        for (j = 0; j < cols; j++){
            total += m[i*cols + j];
        }
    }
    return total;
}

```

Here is the corresponding assembly. Each line is annotated with its English translation:

```

Dump of assembler code for function sumMat:
0x400686 <+0>: push %rbp          # save rbp
0x400687 <+1>: mov %rsp,%rbp      # update rbp (new stack frame)
0x40068a <+4>: mov %rdi,-0x18(%rbp) # copy m to %rbp-0x18
0x40068e <+8>: mov %esi,-0x1c(%rbp) # copy rows to %rbp-0x1c
0x400691 <+11>: mov %edx,-0x20(%rbp) # copy cols parameter to %rbp-0x20
0x400694 <+14>: movl $0x0,-0x4(%rbp) # copy 0 to %rbp-0x4 (total)
0x40069b <+21>: movl $0x0,-0xc(%rbp) # copy 0 to %rbp-0xc (i)
0x4006a2 <+28>: jmp 0x4006e1 <sumMat+91> # goto <sumMat+91>
0x4006a4 <+30>: movl $0x0,-0x8(%rbp) # copy 0 to %rbp-0x8 (j)
0x4006ab <+37>: jmp 0x4006d5 <sumMat+79> # goto <sumMat+79>
0x4006ad <+39>: mov -0xc(%rbp),%eax # copy i to %eax
0x4006b0 <+42>: imul -0x20(%rbp),%eax # mult i with cols, place in %eax
0x4006b4 <+46>: mov %eax,%edx # copy i*cols to %edx
0x4006b6 <+48>: mov -0x8(%rbp),%eax # copy j to %eax
0x4006b9 <+51>: add %edx,%eax # add i*cols with j, place in %eax
0x4006bb <+53>: cltq # convert %eax to a 64-bit int
0x4006bd <+55>: lea 0x0(%rax,4),%rdx # mult (i*cols+j) by 4.put in %rdx
0x4006c5 <+63>: mov -0x18(%rbp),%rax # copy m to %rax
0x4006c9 <+67>: add %rdx,%rax # add m to (i*cols+j)*4.put in %rax
0x4006cc <+70>: mov (%rax),%eax # copy m[i*cols+j] to %eax
0x4006ce <+72>: add %eax,-0x4(%rbp) # add m[i*cols+j] to total
0x4006d1 <+75>: addl $0x1,-0x8(%rbp) # add 1 to j (j++)
0x4006d5 <+79>: mov -0x8(%rbp),%eax # copy j to %eax
0x4006d8 <+82>: cmp -0x20(%rbp),%eax # compare j with cols
0x4006db <+85>: jl 0x4006ad <sumMat+39> # if (j < cols) goto <sumMat+39>
0x4006dd <+87>: addl $0x1,-0xc(%rbp) # add 1 to i
0x4006e1 <+91>: mov -0xc(%rbp),%eax # copy i to %eax
0x4006e4 <+94>: cmp -0x1c(%rbp),%eax # compare i with rows
0x4006e7 <+97>: jl 0x4006a4 <sumMat+30> # if (i < rows) goto <sumMat+30>
0x4006e9 <+99>: mov -0x4(%rbp),%eax # copy total to %eax
0x4006ec <+102>: pop %rbp # clean up stack
0x4006ed <+103>: retq # return total

```

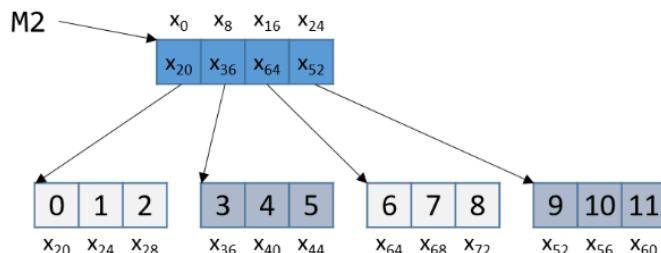


Figure 101. Matrix M2's noncontiguous layout in memory

```

int sumMatrix(int **matrix, int rows, int cols) {
    int i, j, total=0;

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            total += matrix[i][j];
        }
    }
    return total;
}

```

Dump of assembler code for function sumMatrix:

```

0x4006ee <+0>: push %rbp # save rbp
0x4006ef <+1>: mov %rsp,%rbp # update rbp (new stack frame)
0x4006f2 <+4>: mov %rdi,-0x18(%rbp) # copy matrix to %rbp-0x18
0x4006f6 <+8>: mov %esi,-0x1c(%rbp) # copy rows to %rbp-0x1c
0x4006f9 <+11>: mov %edx,-0x20(%rbp) # copy cols to %rbp-0x20
0x4006fc <+14>: movl $0x0,-0x4(%rbp) # copy 0 to %rbp-0x4 (total)
0x400703 <+21>: movl $0x0,-0xc(%rbp) # copy 0 to %rbp-0xc (i)
0x40070a <+28>: jmp 0x40074e <sumMatrix+96> # goto <sumMatrix+96>
0x40070c <+30>: movl $0x0,-0x8(%rbp) # copy 0 to %rbp-0x8 (j)
0x400713 <+37>: jmp 0x400742 <sumMatrix+84> # goto <sumMatrix+84>
0x400715 <+39>: mov -0xc(%rbp),%eax # copy i to %eax
0x400718 <+42>: cltq # convert i to 64-bit integer
0x40071a <+44>: lea 0x0(%rax,8),%rdx # mult i by 8, place in %rdx
0x400722 <+52>: mov -0x18(%rbp),%rax # copy matrix to %rax
0x400726 <+56>: add %rdx,%rax # put i*8 + matrix in %rax
0x400729 <+59>: mov (%rax),%rax # copy matrix[i] to %rax (ptr)
0x40072c <+62>: mov -0x8(%rbp),%edx # copy j to %edx
0x40072f <+65>: movslq %edx,%rdx # convert j to 64-bit integer
0x400732 <+68>: shl $0x2,%rdx # mult j by 4, place in %rdx
0x400736 <+72>: add %rdx,%rax # put j*4 + matrix[i] in %rax
0x400739 <+75>: mov (%rax),%eax # copy matrix[i][j] to %eax
0x40073b <+77>: add %eax,-0x4(%rbp) # add matrix[i][j] to total
0x40073e <+80>: addl $0x1,-0x8(%rbp) # add 1 to j (j++)
0x400742 <+84>: mov -0x8(%rbp),%eax # copy j to %eax
0x400745 <+87>: cmp -0x20(%rbp),%eax # compare j with cols
0x400748 <+90>: jl 0x400715 <sumMatrix+39> # if j<cols goto<sumMatrix+39>
0x40074a <+92>: addl $0x1,-0xc(%rbp) # add 1 to i (i++)
0x40074e <+96>: mov -0xc(%rbp),%eax # copy i to %eax
0x400751 <+99>: cmp -0x1c(%rbp),%eax # compare i with rows
0x400754 <+102>: jl 0x40070c <sumMatrix+30> # if i<rows goto<sumMatrix+30>
0x400756 <+104>: mov -0x4(%rbp),%eax # copy total to %eax
0x400759 <+107>: pop %rbp # restore %rbp
0x40075a <+108>: retq # return total

```

Uses pointers, its an array of int pointers. The element at matrix[i] is itself an int pointer.

Structs in assembly

Let's revisit `struct studentT` from Chapter 1:

```
struct studentT {
    char name[64];
    int age;
    int grad_yr;
    float gpa;
};

struct studentT student;
```

[Figure 103](#) shows how `student` is laid out in memory. Each x_i denotes the address of a particular field.

student:	name[0]	name[1]	...	name[63]	age	grad_yr	gpa
	x_0	x_1	...	x_{63}	x_{64}	x_{68}	x_{72}

Figure 103. The memory layout of a struct studentT

To understand how the compiler generates assembly code to work with a `struct`, consider the function `initStudent`:

```
void initStudent(struct studentT *s, char *nm, int ag, int gr, float g) {
    strncpy(s->name, nm, 64);
    s->grad_yr = gr;
    s->age = ag;
    s->gpa = g;
}
```

The `initStudent` function uses the base address of a `struct studentT` as its first parameter, and the desired values for each field as its remaining parameters. The following listing depicts this function in assembly:

```
Dump of assembler code for function initStudent:
0x4006aa <+0>: push %rbp          #save rbp
0x4006ab <+1>: mov %rsp,%rbp      #update rbp (new stack frame)
0x4006ae <+4>: sub $0x20,%rsp     #add 32 bytes to stack frame
0x4006b2 <+8>: mov %rdi,-0x8(%rbp) #copy 1st param to %rbp-0x8 (s)
0x4006b6 <+12>: mov %rsi,-0x10(%rbp) #copy 2nd param to %rbp-0x10 (nm)
0x4006ba <+16>: mov %edx,-0x14(%rbp) #copy 3rd param to %rbp-0x14 (ag)
0x4006bd <+19>: mov %ecx,-0x18(%rbp) #copy 4th param to %rbp-0x18 (gr)
0x4006c0 <+22>: movss %xmm0,-0x1c(%rbp) #copy 5th param to %rbp-0x1c (g)
0x4006c5 <+27>: mov -0x8(%rbp),%rax   #copy s to %rax
0x4006c9 <+31>: mov -0x10(%rbp),%rcx  #copy nm to %rcx
0x4006cd <+35>: mov $0x40,%edx       #copy 0x40 (or 64) to %edx
0x4006d2 <+40>: mov %rcx,%rsi        #copy nm to %rsi
0x4006d5 <+43>: mov %rax,%rdi        #copy s to %rdi
0x4006d8 <+46>: callq 0x400460 <strncpy@plt> #call strncpy(s->name, nm, 64)
0x4006dd <+51>: mov -0x8(%rbp),%rax   #copy s to %rax
0x4006e1 <+55>: mov -0x18(%rbp),%edx  #copy gr to %edx
0x4006e4 <+58>: mov %edx,0x44(%rax)    #copy gr to %rax+0x44 (s->grad_yr)
0x4006e7 <+61>: mov -0x8(%rbp),%rax   #copy s to %rax
0x4006eb <+65>: mov -0x14(%rbp),%edx  #copy ag to %edx
0x4006ee <+68>: mov %edx,0x40(%rax)    #copy ag to %rax+0x40 (s->age)
0x4006f1 <+71>: mov -0x8(%rbp),%rax   #copy s to %rax
0x4006f5 <+75>: movss -0x1c(%rbp),%xmm0 #copy g to %xmm0
0x4006fa <+80>: movss %xmm0,0x48(%rax) #copy g to %rax+0x48
0x400700 <+86>: leaveq                #prepare stack to exit function
0x400701 <+87>: retq                  #return (void func, %rax ignored)
```

Consider the following modified declaration of `struct studentT`:

```
struct studentTM {
    char name[63]; //updated to 63 instead of 64
    int age;
    int grad_yr;
    float gpa;
};

struct studentTM student2;
```

The size of the `name` field is modified to be 63 bytes, instead of the original 64. Consider how this affects the way the `struct` is laid out in memory. It may be tempting to visualize it as in [Figure 104](#).

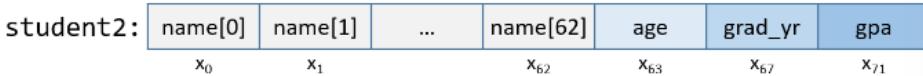


Figure 104. An incorrect memory layout for the updated struct studentTM. Note that the struct's "name" field is reduced from 64 to 63 bytes.

In this depiction, the `age` field occurs in the byte immediately following the `name` field. But this is incorrect. [Figure 105](#) depicts the actual layout in memory.

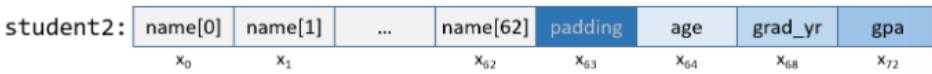


Figure 105. The correct memory layout for the updated struct studentTM. Byte x₆₃ is added by the compiler to satisfy memory alignment constraints, but it doesn't correspond to any of the fields.

Optimization

Dive into systems - ch 12

Constant Folding, Constant Propagation, Dead Code Elimination, Simplifying Expressions, Strength Reduction, Aliasing, Code Motion, Inlining, Loop Unrolling, Loop Interchange.

Code optimization.

GCC C compiler implements wide variety of optimization flags = gives direct access to a subset of the implemented optimization.

For example, the following command compiles a sample program with level 1 optimizations:

```
$ gcc -O1 -o program program.c
```

The level 1 (-O1 or -O) optimizations in GCC perform basic optimizations to reduce code size and execution time while attempting to keep compile time to a minimum. Level 2 (-

O2) optimizations include most of GCC's implemented optimizations that do not involve a space-performance trade-off. Lastly, level 3 (-O3) performs additional optimizations (such as function inlining, discussed later in this chapter), and may cause the program to take significantly longer to compile

What compilers already do

Constant folding

- Constants are evaluated at compile time to reduce the number of instructions

For example, in the code snippet that follows, **macro expansion** replaces the statement `int debug = N-5` with `int debug = 5-5`. **Constant folding** then updates this statement to `int debug = 0`.

```
#define N 5
int debug = N - 5; //constant folding changes this statement to debug = 0;
```

Constant propagation

- Replace variables with a constant value, if its known in compile time

```
int debug = 0;

//sums up all the elements in an array
int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (debug) {
            printf("array[%d] is: %d\n", i, array[i]);
        }
    }
    return 2 * total;
}
```

A compiler employing constant propagation will change `if (debug)` to `if (0)`.

Dead code elimination

- Unused variables, assignments or statements
- Makes it faster by shrinking code size and associated set of instructions

```

int debug = 0;

//sums up all the elements in an array
int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (0) { //debug replaced by constant propagation by compiler
            printf("array[%d] is: %d\n", i, array[i]);
        }
    }
    return 2 * total;
}

```

A compiler employing dataflow analysis recognizes that the `if` statement always evaluates to false and that the `printf` statement never executes. The compiler therefore eliminates the `if` statement and the call to `printf` in the compiled executable. Another pass also eliminates the statement `debug = 0`.

Simplifying expressions

- Some instructions are more expensive than others
- Can be reduced by simplifying mathematical operations when possible

```

//declaration of debug removed through dead-code elimination

//sums up all the elements in an array
int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        //if statement removed through data-flow analysis
    }
    return total + total; //simplifying expression
}

```

Likewise, the compiler will transform code sequences with bit-shifting and other bitwise operators to simplify expressions. For example, the compiler may replace the expression `total * 8` with `total << 3`, or the expression `total % 8` with `total & 7` given that bitwise operations are performed with a single fast instruction.

What compilers cannot always do: benefits of learning code optimization

Algorithmic strength reduction is impossible

- Top reason for bad code performance
 - Bad data structures and algorithms
- Compilers cannot fix these decisions

Compiler optimization flags are not guaranteed to make code "optimal" (or consistent)

- Increasing the level from 2 to 3 may not always decrease runtime
- Updating can slow down or changes nothing
- Or it can yield errors
- GCC debug (-g) flag is incompatible with optimization flags

Pointers can prove problematic

- If a transformation risks changing the behaviour of the program, the compiler will not make the transformation
- Memory aliasing
 - Two different pointers point to the same address

Table 111. Comparison of two functions that multiplies the first number by 10 and adds the second to it. Available at [this link](#).

Unoptimized Version	Optimized Version
<pre>void shiftAdd(int *a, int *b){ *a = *a * 10; //multiply by 10 *a += *b; //add b }</pre>	<pre>void shiftAddOpt(int *a, int *b){ *a = (*a * 10) + *b; }</pre>

The `shiftAddOpt` function optimizes the `shiftAdd` function by removing an additional memory reference to `a`, resulting in a smaller set of instructions in the compiled assembly. However, the compiler will never make this optimization due to the risk of memory aliasing. To understand why, consider the following `main` function:

First steps: Code profiling

Premature optimization

- Occurs when a programmer attempts to optimize based on "gut feelings"
- Not based on data

Hot spots

- Focus on areas with the most loops
- Verify with benchmarking tools before attempting optimization

Table 114. Loop execution components (assuming k iterations)

Initialization statement	Boolean expression	Step expression	Loop body
1	$k+1$	k	k

Loop invariant code motion

- Optimization technique that moves static computations that occur inside of a loop to outside the loop without affecting the loops behaviour.
- the `-fmove-loop-invariants` compiler flag in GCC (enabled at level `-O1`) attempts to identify examples of loop-invariant code motion and move them outside their respective loop.
- Cannot always be detected in the case of function calls
 - Since they cause side effects
 - We know \sqrt{x} always yield the same result, but the compiler does not, and will not always call it
 - If the compiler doesn't know that it always returns the same result, it will not move it outside of the loop

Other optimizations: loop unrolling and function inlining

Fractional performance gains are not worth the git to code readability.

This is already implemented by compilers

Function inlining

Replaces calls to a function with the body of the function

Table 116. Example of compiler inlining the `allocateArray` function.

Original Version	Version with <code>allocateArray</code> in-lined
<pre> int main(int argc, char **argv) { // omitted for brevity // some variables shortened for space considerations int lim = strtol(argv[1], NULL, 10); // allocation of array int *a = allocateArray(lim); // generates sequence of primes int len = genPrimeSequence(a, lim); return 0; } </pre>	<pre> int main(int argc, char **argv) { // omitted for brevity // some variables shortened for space considerations int lim = strtol(argv[1], NULL, 10); // allocation of array (in- lined) int *a = malloc(lim * sizeof(int)); // generates sequence of primes int len = genPrimeSequence(a, lim); return 0; } </pre>

Enable compiler to eliminate excessive calls and easier to identify potential improvements

- Constant propagation, constant folding and dead code elimination

Should generally avoid doing manually.

Carries high risk of reducing readability.

Loop unrolling

Reduce the impact of wrong guesses

- Reduce the number of iterations of a loop by a factor n, by increasing the workload that each iteration performs by a factor of n.
- When the loop is unrolled by a factor of 2, the number of iterations in the loop is cut by half, where the amount of work performed per iteration is doubled
- Manual example in 12.2.2

Memory considerations

Compiler cannot always improve on a programs memory use

Loop interchange

Switch order of inner and outer loops in nested loops to maximize cache locality

Compiler flag -floop-interchange exists but is currently not available by default

Table 118. Loop interchange on the `matrixVectorMultiply()` function.

Original Version (matrixVector.c)	Loop interchange version (matrixVector2.c)
<pre>void matrixVectorMultiply(int **m, int *v, int **res, int row, int col) { int i, j; //cycles through every matrix //in inner-most loop (inefficient) for (j = 0; j < col; j++){ for (i = 0; i < row; i++){ res[i][j] = m[i][j] * v[j]; } } }</pre>	<pre>void matrixVectorMultiply(int **m, int *v, int **res, int row, int col) { int i, j; //cycles through every row of //matrix //in inner-most loop for (i = 0; i < row; i++){ for (j = 0; j < col; j++){ res[i][j] = m[i][j] * v[j]; } } }</pre>

Table 119. Time in Seconds to Perform Matrix Multiplication on $10,000 \times 10,000$ Elements

Version	Program	Unoptimized	-01	-02	-03
Original	<code>matrixVector</code>	2.01	2.05	2.07	2.08
With Loop Interchange	<code>matrixVector2</code>	0.27	0.08	0.06	0.06

Algorithms for modern hardware

Branchless Programming, Throughput Programming, Prefetching, Alignment, Packing, Horizontal Summation, Auto-Vectorization

3.2-3.5

The cost of branching

When CPU encounters conditional jump or any other branching, it starts speculatively executing the branch that seems most likely.

It computes statistics and recognizes common patterns.

The true "cost" of a branch depends on how well it can be predicted

- If 50/50 then there will be a control hazard

Concrete example in book.

You can hint the likeliness of branches by putting `[[likely]]`

Branchless programming

Remove if statements?

There are no Boolean types in assembly.

Eliminating branching using predication comes at the cost of evaluating BOTH branches.

It eliminates a control hazard but introduces data hazard.

Only do when the cost of computing both branches instead of just one, outweighs the penalty for the potential branch misprediction.

Instruction tables

Interleaving stages of execution is not only on the main CPU pipeline, but also on the separate instructions and memory.

There are 2 different costs for instructions here

- Latency: how many cycles are needed to receive the results of an instruction
- Throughput: how many instructions can be, on average, executed per cycle
 - The bigger the number the more is passed through

We get these from instruction tables.

Some comments:

- If a certain instruction is especially frequent, its execution unit could be duplicated to increase its throughput — possibly to even more than one, but not higher than the [decode width](#).
- Some instructions have a latency of 0. This means that these instruction are used to control the scheduler and don't reach the execution stage. They still

have non-zero reciprocal throughput because the [CPU front-end](#) still needs to process them.

- Most instructions are pipelined, and if they have the reciprocal throughput of nn , this usually means that their execution unit can take another instruction after nn cycles (and if it is below 1, this means that there are multiple execution units, all capable of taking another instruction on the next cycle). One notable exception is [integer division](#): it is either very poorly pipelined or not pipelined at all.

Some instructions have variable latency, depending on not only the size, but also the values of the operands. For memory operations (including fused ones like add), the latency is usually specified for the best case (an L1 cache hit).

Throughput computing

Optimizing for latency is different than optimizing for throughput

- Data structures, queries or small one-time/branchy algorithms
 - Look up latencies of instructions
 - Mentally construct the execution graph of the computation
 - Then try to reorganize it so that the critical path is shorter
- Hot loops and large dataset algorithms
 - Look up the throughput of their instructions
 - Count how many times each one is used per iteration
 - Determine which is the bottleneck
 - Try to reconstruct the loop so that its used less often

When there is some interdependency between consecutive iterations, there may potentially be a pipeline stall caused by a data hazard as the next iterations is waiting for the previous to complete

9.6-9.7

Prefetching

Prefetch data that is likely to be accessed next

- Easy with do data of control hazards

If memory locations aren't in the instruction stream, they can still be predicted

- Explicitly
 - Separately reading the next data word or any of the bytes in the same cache line, so that is is lifted in the cache hierarchy
- Implicitly
 - Using simple access patterns such as linear iteration, which are detectable by the memory hardware that can start prefetching automatically

Hardware prefetching

- Make the CPU request consecutive cache lines when iterating over the permutation, but still accessing the elements inside a cache line in random order

Software prefetching

- Load any byte in the cache line with mov or any other memory instruction
- CPU's have separate prefetch instruction that lifts a cache line without doing anything with it

Alignment and packing

Memory is partitioned into 64B cache lines, makes it difficult to operate on data words that cross a cache lines boundary

- Primitive types as a 32-bit integer should be located on a single cache line
- Receiving two lines require more memory bandwidth

Aligned allocation

- By default, when you allocate an array of some primitive type, you are guaranteed that the addresses of all elements are a multiple of their size, which ensures that they only span a single cache line

Structure alignment

- Structure alignment similarly ensures that the address of all its member primitive types (char, int, float*, etc) are multiples of their size, which automatically guarantees that each of them only spans one cache line. It achieves that by:
 - *padding*, if necessary, each structure member with a variable number of blank bytes to satisfy the alignment requirement of the next member;
 - setting the alignment requirement of the structure itself to the maximum of the alignment requirements of its member types, so that when an array of the structure type is allocated or it is used as a member type in another structure, the alignment requirements of all its primitive types are satisfied.

Optimizing member order

- Padding is only inserted before a not-yet-aligned member or at the end of the structure. By changing the ordering of members in a structure, it is possible to change the required number of padding bytes and the total size of the structure.

Structure packing

- If you know what you are doing, you can disable structure padding and pack your data as tight as possible.
- You have to ask the compiler to do it, as such functionality is not a part of neither C nor C++ standard yet

Bit fields

- Packing along with bit fields
- Allow you to explicitly fix the size of a member in bits

10.3-10.6

Reductions

(folding in F#)

The simplest example of reduction is calculating the sum an array:

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

The naive approach is not so straightforward to vectorize, because the state of the loop (sum s on the current prefix) depends on the previous iteration. The way to overcome this is to split a single scalar accumulator s into 8 separate ones, so that s_i would contain the sum of every 8th element of the original array, shifted by i :

$$s_i = \sum_{j=0}^{n/8} a_{8j+i}$$

If we store these 8 accumulators in a single 256-bit vector, we can update them all at once by consecutive 8-element segments of the array. With [vector extensions](#), this is straightforward:

```
int sum_simd(v8si *a, int n) {
    //           ^ you can just cast a pointer normally, like with any other pointer type
    v8si s = {0};

    for (int i = 0; i < n / 8; i++)
        s += a[i];

    int res = 0;

    // sum 8 accumulators into one
    for (int i = 0; i < 8; i++)
        res += s[i];

    // add the remainder of a
    for (int i = n / 8 * 8; i < n; i++)
        res += a[i];
```

Can be used to find minimum or the xor-sum of an array.

Instruction-level parallelism.

Our implementation matches what the compiler produces automatically, but it is actually suboptimal: when we use just one accumulator, [we have to wait](#) one cycle

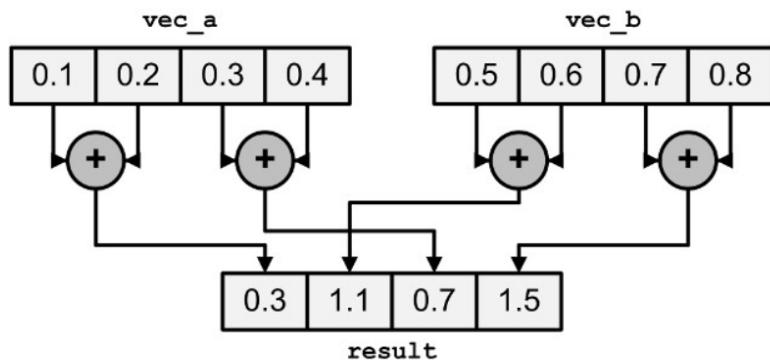
between the loop iterations for a vector addition to complete, while the [throughput](#) of corresponding instruction is 2 on this microarchitecture.

If we again divide the array in $B \geq 2B \geq 2$ parts and use a *separate* accumulator for each, we can saturate the throughput of vector addition and increase the performance twofold.

Horizontal summation

The part where we sum up the 8 accumulators stored in a vector register into a single scalar to get the total sum is called “horizontal summation.”

Although extracting and adding every scalar one by one only takes a constant number of cycles, it can be computed slightly faster using a [special instruction](#) that adds together pairs of adjacent elements in a register.



Horizontal summation in SSE/AVX. Note how the output is stored: the (a b a b) interleaving is common for reducing operations

Masking and blending

Options for control flow are limited

- The operations you apply to a vector are the same for all its elements
- Problems usually solved with if or any other type of branching is harder
 - Need to be dealt with by branchless programming

Masking

Main way to make branchless is predication

- Computing the results of both branches and then using either some arithmetic trick or special “conditional move” instruction

```

for (int i = 0; i < N; i++)
    a[i] = rand() % 100;

int s = 0;

// branch:
for (int i = 0; i < N; i++)
    if (a[i] < 50)
        s += a[i];

// no branch:
for (int i = 0; i < N; i++)
    s += (a[i] < 50) * a[i];

// also no branch:
for (int i = 0; i < N; i++)
    s += (a[i] < 50 ? a[i] : 0);

```

To vectorize this loop, we are going to need two new instructions:

- `_mm256_cmpgt_epi32`, which compares the integers in two vectors and produces a mask of all ones if the first element is more than the second and a mask of full zeros otherwise.
- `_mm256_blendv_epi8`, which blends (combines) the values of two vectors based on the provided mask.

```

const reg c = _mm256_set1_epi32(49);
const reg z = _mm256_setzero_si256();
reg s = _mm256_setzero_si256();

for (int i = 0; i < N; i += 8) {
    reg x = _mm256_load_si256( (reg*) &a[i] );
    reg mask = _mm256_cmpgt_epi32(x, c);
    x = _mm256_blendv_epi8(x, z, mask);
    s = _mm256_add_epi32(s, x);
}

```

This is how predication is usually done in SIMD, but it isn't always the most optimal approach. We can use the fact that one of the blended values is zero, and use bitwise `and` with the mask instead of blending:

```

const reg c = _mm256_set1_epi32(50);
reg s = _mm256_setzero_si256();

for (int i = 0; i < N; i += 8) {
    reg x = _mm256_load_si256( (reg*) &a[i] );
    reg mask = _mm256_cmpgt_epi32(c, x);
    x = _mm256_and_si256(x, mask);
    s = _mm256_add_epi32(s, x);
}

```

This loop is slightly faster

- The vector "and" takes one cycle less than "blend"

Searching

Array: Find a specific value and return its position

- To vectorize (Example in book)

- Compare a vector of its elements with the searched value for equality
- Producing a mask
- And then somehow check if this mask is zero
- If not, the element is somewhere within the block of 8

Checking if a vector is zero is a common operation

In-register shuffles

Masking lets you apply operations to only a subset of vector elements

- Effective and frequently used data manipulation technique
- Often need to perform more advanced operations (permuting values inside a vector register) instead of just blending them in with other vectors.

We can add 1 general permutation instruction that takes the indices of a permutation and produces these indices using precomputed lookup tables.

Population count (the hamming weight)

- The count of 1 bits in a binary string

Naïve way is go through it bit by bit

- Vectorization is faster

Filter

- Data processing primitive
- Takes an array as input and writes out only the elements that satisfy a given predicate
 - In original order

Auto-vectorization and SPMD

embarrassingly parallel computations: the kinds where all you do is apply some elementwise function to all elements of an array and write it back somewhere else

Potential problems with only relying on auto-vectorization

- Array size
- Memory aliasing
- Alignment
- Checking if vectorization happened

Slides

C: key features

imperative language

statically typed (albeit permissively)

minimal language:

- efficient mapping to assembly code (this lecture)
- all interesting stuff done by libraries (even printf)

minimal run-time support:

- explicit memory management (pointers, malloc)
- explicit threads programming (next lecture)
- efficient mapping to assembly code (this lecture)

Programming Language Concepts,
Ch. 7 (esp. 7.5)
"Programs as Data" course

C language itself very minimal.
you must be acquainted w/ C libraries.

C: spirit

- trust the programmer.
- do not prevent the programmer from doing what needs to be done.
- keep the language small and simple.
- provide only one way to do an operation.
- make it fast, even if not guaranteed to be portable.
- make support for safety & security demonstrable

C primer, not a lecture/tutorial

we assume you are comfortable with another imperative programming language, so we assume familiarity with:

- expressions, operators, numbers, characters, strings, arrays,
- statements, blocks, conditionals, loops, variables, scope,
- functions, parameters, modules, libraries, ...

in other words, that you can easily grasp what is going on over there →

UNIVERSITY OF COPENHAGEN

```
#include <stdio.h>
#define N 20

int fibs[N];

void computefibs() {
    fibs[0] = 0;
    fibs[1] = 1;
    for ( int i = 2; i < N; i++ ) {
        fibs[i] = fibs[i-1] + fibs[i-2];
    }
}

void printfibs() {
    for ( int i = 0; i < N; i++ ) {
        printf("%d\n", fibs[i]);
    }
}

int main() {
    computefibs();
    printfibs();
    return 0;
}
```

-UU-----F1 fibs.c All (1,4)

C

ble 2. C Numeric Types

Type name	Usual size	Values stored	How to declare
char	1 byte	integers	char x;
short	2 bytes	signed integers	short x;
int	4 bytes	signed integers	int x;
long	4 or 8 bytes	signed integers	long x;
long long	8 bytes	signed integers	long long x;
float	4 bytes	signed real numbers	float x;
double	8 bytes	signed real numbers	double x;

also provides *unsigned* versions of the integer numeric types (`char`, `short`, `int`, `long`, and `long long`). To declare a variable as *unsigned*, add the keyword `unsigned` before the type name. For example:

```
int x;          // x is a signed int variable
unsigned int y; // y is an unsigned int variable
```

C pointers

that's really all.
small level of abstraction on top of mov:

"A pointer is a variable that contains [an] address."

- K & R

We have already
encountered them in
assembly

useful for:

- explicit memory management (data placement)
- share data w/o copies
- indirection

(i.e. "give me the next byte")
(great power over memory access)
(great responsibility; might not have access)

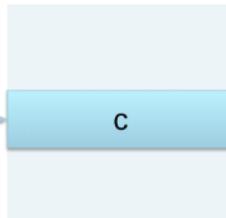
Notation

& gives the address of
that variable

p is a char pointer. It contains an address
(of a char variable).

```
char *p;  
char c;  
p = &c;      (char *) &c
```

address-of



Q: how many bytes (or bits) to represent a pointer?

UNIVERSITY OF COPENHAGEN

can also write `char* p;`
Linux: next to var name

Exercise

Int** q -> since it's a pointer
pointing to another pointer

write type declarations for the following variables: char* t

- q : a pointer to an integer pointer
- t : a pointer to a byte
- u : a pointer to a byte array

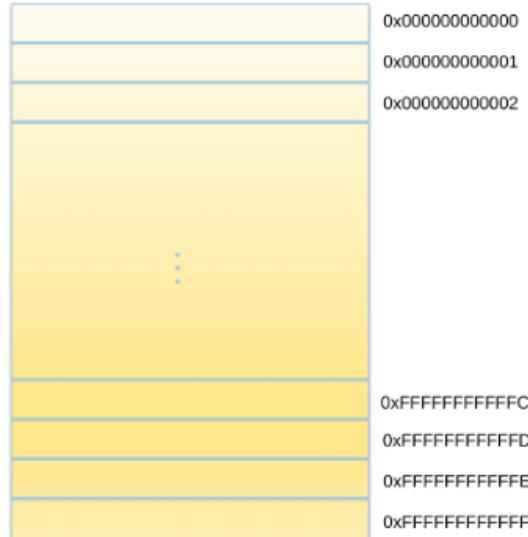
Q: spend 1-2 mins on this

How does main memory work?

- **sequence of bytes.**
1 byte = 8 bits
- each byte has unique address.
- address space is linear.

technology:

- DRAM, SRAM: transient
- 3D Xpoint: persistent

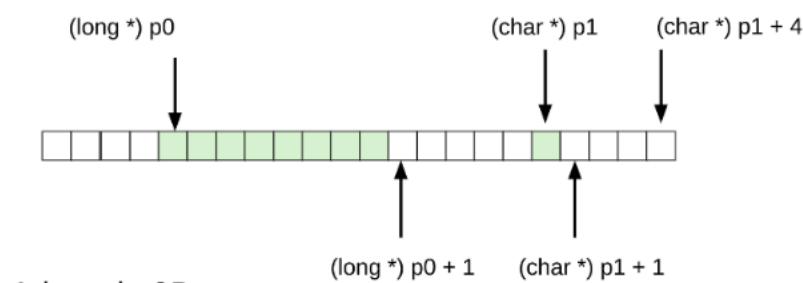


UNIVERSITY OF COPENHAGEN

02.09.2021 · 1

Pointer Arithmetic

recall: 1 cell is 1 byte.
memory is byte-addressable.



A long is 8B
A char is 1B

Compiler will
know that p0 is
a long

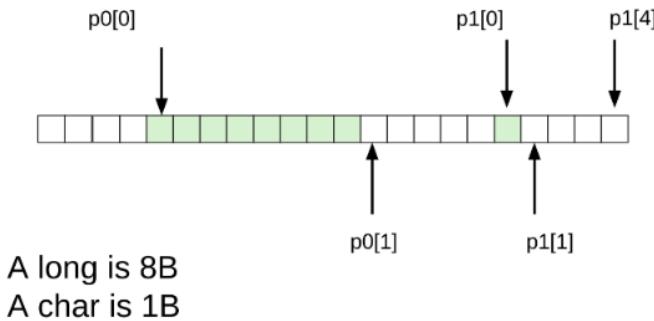
you can do arith. on addr.
how far you skip, depends on
type of pointer.

UNIVERSITY OF COPENHAGEN

16.09.2020 · 1

Pointer Arithmetic

equivalent to this;
arrays are just pointers
(and a little more)



Notation:
 $a[i]$ is equivalent to $*(a+i)$

UNIVERSITY OF COPENHAGEN

16.09.2020 · 13

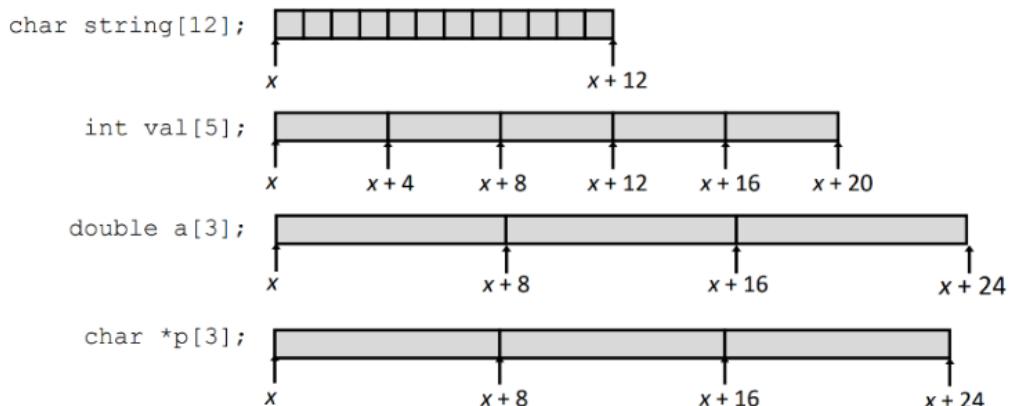
Array Allocation

Basic Principle

$T \mathbf{A}[L];$

A is an Array of data type T and length L

Contiguously allocated region of $L * \text{sizeof}(T)$ bytes



A pointer occupies 8 bytes?? 64 bit???

Array Access

```
int A[5] = {0, 1, 2, 3, 4};  
Array of data type int and length 5  
Identifier A can be used as a pointer to array element 0: Type int*
```

Array's are basically
pointers without types?

Reference	Type	Value
val[4]	int	4
val	int *	x
val+1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
* (val+1)	int	1
val + i	int *	x + 4i

Pointers are NOT arrays

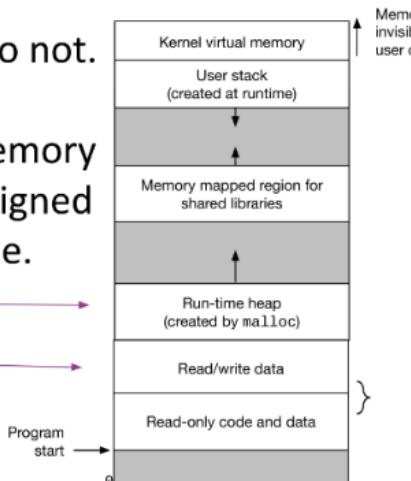
(1) arrays have size; pointers do not.

(2) arrays assigned address in memory at compile time; pointers are assigned an address in memory at run time.

```
int* a;  
int b[10];  
a = b;
```

a now points to &b[0],
size info lost

must specify array size, so
compiler can allocate space at
compile time



16.09.2020 · 1

Restrictions

You can't have:

- A function that returns a function
Never `foo()`
- A function that returns an array
Never `foo()`
- An array of function
Never `foo[]()`

(type system doesn't allow it,
despite conceptually making sense)

Instead...

But you *can* have

- A function returning a pointer to a function
`*fun()`
- A function returning a pointer to an array
`*fun[]()`
- An array of function pointers
`*foo[]()`

(due to restrictions in C type system,
we use pointers as an indirection-level)

Type specifiers: struct

struct: a bunch of data items grouped together
(in memory)

```
struct tag {  
    type_1 identifier_1;  
    type_2 identifier_2;  
    ...  
    type_N identifier_N;  
};  
struct tag variable_name;
```

Type specifier: struct

data items accessed through dot operator.
when using *pointer to struct*,
data items accessed through arrow operator.

```
/* struct that points to the next struct */  
struct node_tag {  
    int datum;  
    struct node_tag *next;  
};  
struct node_tag a,b;  
a.next = &b;  
a.next->next=NULL;
```

foo->bar
shorthand for
(*foo).bar

UNN shorthand for
(*(a.next)).next

16.09.2020 · 11

Type specifier: struct

giving names to
bits inside a
struct.

structs can have bit fields, unnamed fields, and
word-aligned fields.

```
/* process ID info */  
struct pid_tag {  
    unsigned short int inactive :1; /* 1 bit of padding */  
    unsigned short int :1;  
    unsigned short int refcount :6;  
    unsigned short int :8; /* pad to short length */  
    short pid_id;  
    struct pid_tag *link;  
};
```

naming its bits

Type specifier: struct

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

struct {
    unsigned int age : 3;
} Age;

int main( ) {

    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );

    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );

    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );

    return 0;
}
```

```
phbo@parallels-vm ~/C/C/Lectures> ./l3ex2
Sizeof(Age): 4
Age.age: 4
Age.age: 7
Age.age: 0
```

UNIVERSITY OF COPENHAGEN

Q: if I assign 10, and print, what gets printed?

16.09.2020 · 2

A: because of overflow

Type specifier: struct, the beauty of

```
struct s_tag { int a[100]; };
struct s_tag orange, lime, lemon;
struct s_tag twofold (struct s_tag s) {
    int j;
    for (j=0;j<100;j++) s.a[j] *= 2;
    return s;
}

main() {
    int i;
    for (i=0;i<100;i++) lime.a[i] = 1;
    lemon = twofold(lime);
    orange = lemon; /* assigns entire struct */
}
```

array as input

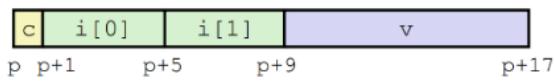
function cannot return an array,
but
function can return a struct.
(cf. returning a pointer)

this will **copy** the entire
structure!
(if that's not what you want,
then use pointers)

UNIVERSITY OF COPENHAGEN

Structures & Alignment

Unaligned Data

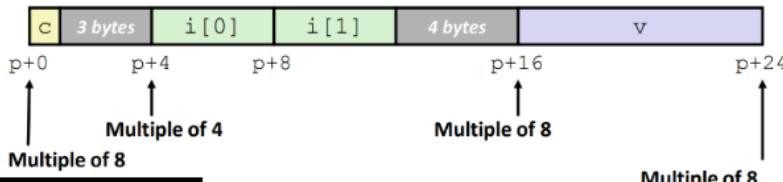


```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

Aligned Data

Primitive data type requires K bytes

Address must be multiple of K



UNIVERSITY OF COPENHAGEN

Specific Cases of Alignment (x86-64)

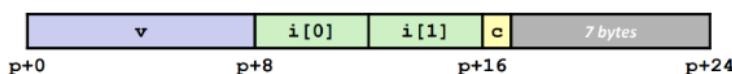
- 1 byte: `char`, ...
no restrictions on address
- 2 bytes: `short`, ...
lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, ...
lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `char *`, ...
lowest 3 bits of address must be 000_2

Meeting Overall Alignment Requirement

For largest alignment requirement K

Overall structure must be multiple of K

```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Type specifier: union

union: like struct, except:
the storage for the individual members is overlaid:
only one member at a time can be stored there.

```
union bits32_tag {  
    int whole; /* a 4B value */  
    struct {char c0,c1,c2,c3;} byte; /* 4 * 1B values */  
}
```

example use: TCP frames

Type specifier: enum

enums (enumerated types) just a way of associating a series of names w/ a series of integer values.

```
enum sizes { small=7, medium, large=10, humongous };
```

8 (7+1)

11

sizes.small etc. are **constants**

Type specifier: void

void is the type of a function that does not return a result.

unit type
(but cannot use it as freely
as other types,
hence this wording)

void *

void * defines a pointer to data of unspecified type.

Type qualifier: const

const qualifies a read-only variable; one that cannot be a left value in assignment (except variable declaration).

```
const.c
int main()
2 {
3     const int i;
4     i = 12;
5
6     return 0;
7 }

% make const
cc -c const.c -o const
const.c:4:4: error: cannot assign to variable 'i' with const-qualified type 'const int'
      i = 12;
           ^
const.c:3:12: note: variable 'i' declared const here
    const int i;
                   ^
1 error generated.
make: *** [const] Error 1
```

combination of const and *: "I am giving you a pointer to this thing, but you may not change the pointer."

```
int * const p;
// p cannot be left value in an assignment
```

but you may overwrite what's at the other end of the pointer.

UNIVERSITY OF COPENHAGEN

Type qualifier: const

(note: cannot have)
const int limit;
limit = 10;

```
const int limit = 10;
const int * limitp = &limit;
int i=27;
limitp = &i;

int limit = 10;
int * const limitp = &limit;
int i=27;
limitp = &i;
```



pointer to constants. can point to other constants (i.e. something that cannot be on lhs of assignment).



constant pointer. cannot point to other things.
(W: but can overwrite pointee; e.g. update limit)

Type qualifier: volatile

volatile qualifies a variable that might be modified outside the program. (by another thread, device, etc.).

(program reads it twice $\not\Rightarrow$ same value read)

```
struct devregs{
    unsigned short volatile csr;
    unsigned short const volatile data;
};
```

can be initialized outside your program.

Type conversions

Explicit:

(type) expr
(int)'y' e.g.

a value of one type is explicitly cast to another type.
bits do not change. higher bits that do not fit truncated.

Implicit:

1. a value of one type is assigned to a variable of a different type
2. an operator converts the type of its operands
3. **a value is passed as argument to a function, or when a value is returned from a function**

Unsigned and signed

Same bit level representation, different interpretations

If there is mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*

be very careful!
(recall kernel mem copy)

Pointer conversions, Rules

- A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage.
- A pointer to an object can be converted to a pointer to an object whose type requires less or equally strict storage alignment, and back again without change.
- A pointer to void can be converted to or from a pointer to any type, without restriction or loss of information. If the result is converted back to the original type, the original pointer is recovered.
- If a pointer is converted to another pointer with the same type but having different or additional qualifiers, the new pointer is the same as the old except for restrictions imposed by the new qualifier.

A pointer value can also be converted to an integral value.

The conversion path depends on the size of the pointer and the size of the integral type:

- If the size of the pointer is greater than or equal to the size of the integral type, the pointer behaves like an unsigned value. It cannot be converted to a floating value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type.

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).
- If the size of the integral type is different from the size of the pointer type, the integral type is first extended or truncated to fit the size of the pointer. It is then treated as a pointer value.

Pointer conversions, Example (common use)

Name

malloc, free, calloc, realloc - allocate and free dynamic memory

Synopsis

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);

/* j is a pointer to an array of 20 char *,
char (*j)[20];
j = (char (*)[20]) malloc( 20 );
```

Declarations and definitions

Definition: specifies

what a function does or where a variable is stored.

Declaration: describes type/name of variable/function.

No space is allocated.

Variables and functions are defined exactly once,
but may be declared several times.

W: think of dec as "intent",
and def as "enact"

def fun: what it does
def var: how it's stored
dec: just a signature.

dec: x exists,
def: dec + allocate mem for x

Scope of variables

A variable defined in a function is local to that function. It is an **automatic** variable. It does not retain its value across function calls (lives in stack frame).

A variable defined outside any function is an *external* variable. It is a **global** variable.

Before a global variable can be accessed in other files, it must be declared with the **extern** prefix.

A global variable does not need to be declared in the file where it is defined.

typically, these are collected in a header file.

UNIVERSITY OF COPENHAGEN

The scope of a global variable can be restricted to the file where it is defined with the **static** prefix.

static and **extern** are mutually exclusive.

An automatic variable can retain its value across calls to a function when it is defined with **static**.

Recap so far

What is an automatic variable?

A: local to function

How is a global variable defined when it can be accessed by all C files contributing to an executable?

A: in files where not def, it must be dec w/ extern

How is a global variable defined when it can only be accessed from the C file where it is defined?

A: static

How is a global variable declared outside the file where it is defined?

A: extern

Example

```
scope.c
1 #include<stdio.h>
2 int fun()
3 {
4     static int count = 0;
5     count++;
6     return count;
7 }
8
9 int main()
10 {
11     printf("%d ", fun());
12     printf("%d ", fun());
13     return 0;
14 }
```

function definition; we are specifying what function does

local to function, but retains values across calls.

```
% make scope
cc scope.c -o scope
% ./scope
1 2
```

Static inside a function, means it will remember the previous values

```
scope2.c
1 extern int var;
2 int main(void)
3 {
4     var = 10;
5     return 0;
6 }
```

```
% make scope2
cc scope2.c -o scope2
Undefined symbols for architecture x86_64:
 "_var", referenced from:
   _main in scope2-4b8294.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
make: *** [scope2] Error 1
```

solution:
define it in a header file. (next)

Complaining that the variable haven't been defined

The diagram illustrates the compilation process. On the left, a terminal window shows the command `% make scope2` followed by the compiler command `cc scope2.c -o scope2`. Above this, two code snippets are shown. The first is `scope2.c` containing:

```

1 #include "scope2.h"
2
3 extern int var;
4 int main(void)
5 {
6     var = 10;
7     return 0;
8 }

```

The second is `scope2.h` containing:

```

int var;

```

A callout bubble points from the `var` declaration in `scope2.h` to the `var` variable in the `main` function of `scope2.c`, with the text "int var implicitly defined (to 0)".

C Declarations, what they mean

1. Declarations are read by starting with the name (of the variable, function or type)
2. The following precedence rules apply:
 - A. Parentheses grouping together part of the declaration
 - B. The postfix operators
 - Parenthesis indicating a function
 - Square brackets indicating an array
 - C. The prefix operator
 - * denoting a pointer to
3. If a const or volatile is next to a type specifier it qualifies it, otherwise const or volatile applies to the * on its immediate left

Unscrambling C declarations, Example

```
char* const *(*next)();
```

UNIVERSITY OF COPENHAGEN

Q: so, what is this? (brief pause; next)

Jnscrambling C declarations, Example

```
char* const *(*next)();
```

- Next is (1)
- a pointer to (2A)
- a function returning (2B)
- a pointer to (2C)
- a constant pointer to (3)
- char

UNIVERSITY OF COPENHAGEN

that's one expressive type system, from the 70s!

There are tools to help understand this

- C deco?

Gotcha-s

Security, Revisited

```
l2ex1.c+ 12 #define MSIZE 528
13
14
15 void getstuff() {
16     char mybuf[MSIZE];
17     copy_from_kernel(mybuf, -MSIZE);
18     ...
19 }

l2ex1.c+ 1 /* Kernel memory region holding user-accessible data */
2 #define KSIZE 1024
3 char kbuf[KSIZE];
4
5 /* Copy at most maxlen bytes from kernel region to user buffer */
6 int copy_from_kernel(void *user_dest, int maxlen) {
7     /* Byte count len is minimum of buffer size and maxlen */
8     int len = KSIZE < maxlen ? KSIZE : maxlen;
9     memcpy(user_dest, kbuf, len);
10    return len;
11 }
```

memcpy

```
MEMCPY(3)          Linux Programmer's Manual          MEMCPY(3)

NAME
    memcpy - copy memory area

SYNOPSIS
    #include <string.h>
    void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
    The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas must not overlap. Use memmove(3) if the memory areas do overlap.

RETURN VALUE
    The memcpy() function returns a pointer to dest.
```

From [GNU glibc manual /Appendix A – Language Features /Important Data Types:](#)

Data Type: size_t

This is an unsigned integer type used to represent the sizes of objects. The result of the `sizeof` operator is of this type, and functions such as `malloc` (see [Unconstrained Allocation](#)) and `memcpy` (see [Copying Structures and Arrays](#)) accept arguments of this type to specify object sizes. On systems using the GNU C Library, this will be `unsigned int` or `unsigned long int`.

Usage Note: `size_t` is the preferred way to declare any arguments or variables that hold the size of an object.

Size t is an unsigned int

Security - Woops

```
12 #define MSIZE 528
13
14
15 void getstuff() {
16     char mybuf[MSIZE];
17     copy_from_kernel(mybuf, -MSIZE);
18     ...
19 }

ex1.c+ ┶
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

-528 in two's complement:

0xFFFFFDFO



Reinterpreted as unsigned within
memcpy: 4294966768 (decimal)

UNIVERSITY OF COPENHAGEN

09.09.2021 3C

ALWAYS be mindful/careful about unsigned integers

Consider the following code:

Exercise

Consider the following code:

```
le3ex1.c
1 #include <stdio.h>
2
3 int main() {
4     int *ptr;
5     *ptr = 20;
6     printf("%d\n", *ptr);
7     return 0;
8 }
```

What is wrong?

```
le3ex1.c
1 #include <stdio.h>
2
3 int main() {
4     int *ptr;           not initialized (indeterminate pointer);
5     *ptr = 20;          where to store the 20?
6     printf("%d\n", *ptr);
7     return 0;
8 }
```

What is wrong?

(compiler *should* complain,
but compiler *could* do anything)

Exercise

Consider the

```
l3ex1.c:5:3: warning: variable 'ptr' is uninitialized when used here [-Wuninitialized]
    *ptr = 20;
    ^
l3ex1.c:4:10: note: initialize the variable 'ptr' to silence this warning
    int *ptr;
            ^
            = NULL
1 warning generated.
```

What is wrong?

```
le3ex1.c
1 #include <stdio.h>
2
3 int main() {
4     int *ptr;
5     *ptr = 20;
6     printf("%d\n", *ptr);
7     return 0;
8 }
```

gcc decided to
initialize to null.
other compilers, or
later versions of gcc,
might do different.
(segfault, write to
device memory, ...)

```
(gdb) p ptr
$2 = (int *) 0x0
(gdb) p &ptr
$3 = (int **) 0x7fffffff3a8
```

Pointers are valid, null or indeterminate.

A pointer is null when assigned 0

Null pointers evaluate to false in logical expressions

Dereferencing indeterminate pointers leads to **undefined behaviour**

Always initialize pointers!

Overloading, of keywords

From expert C programming

Symbol	Meaning
<code>static</code>	Inside a function, <i>retains its value between calls</i> At the function level, <i>visible only in this file</i> [1]
<code>extern</code>	Applied to a function definition, <i>has global scope</i> (and is redundant) Applied to a variable, <i>defined elsewhere</i>
<code>void</code>	As the return type of a function, <i>doesn't return a value</i> In a pointer declaration, the type of a generic pointer In a parameter list, <i>takes no parameters</i>

UNIVERSITY OF COPENHAGEN

can be used for different things.
(just need to be aware of the
overloading)

16.09.2020

Overloading, of symbols

From expert C program

lol

lol

lol

lol

*	The multiplication operator Applied to a pointer, indirection In a declaration, a pointer
&	Bitwise AND operator Address-of operator
=	Assignment operator
==	Comparison operator
<=	Less-than-or-equal-to operator
<<=	Compound shift-left assignment operator
<	Less-than operator
<	Left delimiter in <code>#include</code> directive
()	Enclose formal parameters in a function definition Make a function call Provide expression precedence Convert (cast) a value to a different type Define a macro with arguments Make a macro call with arguments Enclose the operand of the <code>sizeof</code> operator when it is a typename

UNIVERSITY OF COPENHAGEN

16

Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type){list}</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code>	For relational operators <code><</code> and <code>≤</code> respectively	
7	<code>> >=</code>	For relational operators <code>></code> and <code>≥</code> respectively	
8	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	
9	<code>&</code>	Bitwise AND	
10	<code>^</code>	Bitwise XOR (exclusive or)	
11	<code> </code>	Bitwise OR (inclusive or)	
12	<code>&&</code>	Logical AND	
13 [note 1]	<code> </code>	Logical OR	
14	<code>? :</code> <code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><<= >>=</code> <code>&= ^= =</code>	Ternary conditional [note 2] Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-Left
15	<code>,</code>	Comma	Left-to-right

Kernighan and Ritchie.

when in doubt:
use parentheses!
circumvents
precedence
rules

Precedence problem	Expression	What People Expect	What They Actually Get
. is higher than *	<code>*p.f</code>	the f field of what p points to <code>(*p).f</code>	take the f offset from p, use it as a pointer <code>*(p.f)</code>
[] is higher than *	<code>int *ap[]</code>	ap is a ptr to array of ints <code>int (*ap) []</code>	ap is an array of ptrs-to-int <code>int * (ap[])</code>
function () higher than *	<code>int *fp()</code>	fp is a ptr to function returning int <code>int (*fp) ()</code>	fp is a function returning ptr-to-int <code>int * (fp())</code>
<code>==</code> and <code>!=</code> higher precedence than bitwise operators	<code>(val&mask != 0)</code>	<code>(val&mask) != 0</code>	<code>val & (mask != 0)</code>
<code>==</code> and <code>!=</code> higher precedence than assignment	<code>c=getchar() != EOF</code>	<code>(c=getchar()) != EOF</code>	<code>c=(getchar() != EOF)</code>
arithmetic higher precedence than shift	<code>msb<<4 + lsb</code>	<code>(msb<<4)+lsb</code>	<code>msb<<(4+lsb)</code>
, has lowest precedence of all operators	<code>i = 1,2;</code>	<code>i= (1,2);</code>	<code>(i=1, 2;</code>

C: Take-Aways

You should remember:

1. A pointer is a variable that contains the address of a variable
2. The nature of structs
3. The meaning of const and volatile
4. When type conversions takes place
5. The difference between type specifier and qualifier
6. The difference between declaration and definition
7. The scope of variables (automatic / global)
8. What happens when signed and unsigned are mixed
9. Beware operator precedence
10. Use cdecl when in doubt about a declaration

C to assembly

Use compiler explorer to visualize the assembly code

The screenshot shows the Compiler Explorer interface. On the left is the C source code: `foo (void) { }`. On the right is the generated assembly code:

```
1 foo:  
2     pushq  %rbp  
3     movq  %rsp, %rbp  
4     nop  
5     popq  %rbp  
6     ret
```

rbp is the base pointer, which points to the base of the current stack frame, and rsp is the stack pointer, which points to the top of the current stack frame

The screenshot shows the Compiler Explorer interface. On the left is the C source code: `foo () { return 42; }`. On the right is the generated assembly code:

```
1 foo:  
2     pushq  %rbp  
3     movq  %rsp, %rbp  
4     movl  $42, %eax  
5     popq  %rbp  
6     ret
```

"eax" is the destination of the move, also known as the "destination operand". It's a register, register number 0, and it happens to be 32 bits wide, so this is a 32-bit move. 5 is the source of the moved data, also known as the "source operand".

The screenshot shows the Compiler Explorer interface. On the left is the C source code: `foo (long a) { }`. On the right is the generated assembly code:

```
1 foo:  
2     pushq  %rbp  
3     movq  %rsp, %rbp  
4     movq  %rdi, -8(%rbp)  
5     nop  
6     popq  %rbp  
7     ret
```

```

foo ( int a ) {
}

```

```

A   ⚙️   ⚓   ⚔   ⚕   ⚖   ⚗   ⚑   ⚒   ⚔   ⚕   ⚖   ⚐
1  foo:
2      pushq  %rbp
3      movq  %rsp, %rbp
4      movl  %edi, -4(%rbp)
5      nop
6      popq  %rbp
7      ret

```

```

foo ( int x ) {
    return x + 7;
}

```

```

A   ⚙️   ⚓   ⚔   ⚕   ⚖   ⚐   ⚑   ⚒   ⚔   ⚕   ⚖   ⚐
1  foo:
2      pushq  %rbp
3      movq  %rsp, %rbp
4      movl  %edi, -4(%rbp)
5      movl  -4(%rbp), %eax
6      addl  $7, %eax
7      popq  %rbp
8      ret

```

```

foo ( int x ) {
    return x + 3;
}

bar () {
    return foo (7);
}

```

```

A   ⚙️   ⚓   ⚔   ⚕   ⚖   ⚐   ⚑   ⚒   ⚔   ⚕   ⚖   ⚐
1  foo:
2      pushq  %rbp
3      movq  %rsp, %rbp
4      movl  %edi, -4(%rbp)
5      movl  -4(%rbp), %eax
6      addl  $3, %eax
7      popq  %rbp
8      ret
9  bar:
10     pushq  %rbp
11     movq  %rsp, %rbp
12     movl  $7, %edi
13     call  foo
14     popq  %rbp
15     ret

```

```

foo ( int a1, int a2, int a3, int a4,
      int a5, int a6, int a7, int a8 ) {
    return a6;
}

bar () {
    return foo ( 1, 2, 3, 4, 5, 6, 7, 8 );
}

```

```

A   ⚙️   ⚓   ⚔   ⚕   ⚖   ⚐   ⚑   ⚒   ⚔   ⚕   ⚖   ⚐
1  foo:
2      pushq  %rbp
3      movq  %rsp, %rbp
4      movl  %edi, -8(%rbp)
5      movl  %esi, -8(%rbp)
6      movl  %edx, -12(%rbp)
7      movl  %ecx, -16(%rbp)
8      movl  %r8d, -20(%rbp)
9      movl  %rd, -24(%rbp)
10     movl  -24(%rbp), %eax
11     popq  %rbp
12     ret
13  bar:
14     pushq  %rbp
15     movq  %rsp, %rbp
16     pushq  $8
17     pushq  $7
18     movl  $6, %r9d
19     movl  $5, %r8d
20     movl  $4, %ecx
21     movl  $3, %edx
22     movl  $2, %esi
23     movl  $1, %edi
24     call  foo
25     addq  $16, %rsp
26     leave
27     ret

```

```

foo ( ) {
    int x = 42;
    return x;
}

```

```

A   ⚙️   ⚓   ⚔   ⚕   ⚖   ⚐   ⚑   ⚒   ⚔   ⚕   ⚖   ⚐
1  foo:
2      pushq  %rbp
3      movq  %rsp, %rbp
4      movl  $42, -4(%rbp)
5      movl  -4(%rbp), %eax
6      popq  %rbp
7      ret

```

foo () { int a[4] = { 13, 14, 15, 16 }; return a[2];	A- ⚡ ▾ + ↻ 1 foo: 2 pushq %rbp 3 movq %rsp, %rbp 4 movl \$13, -16(%rbp) 5 movl \$14, -12(%rbp) 6 movl \$15, -8(%rbp) 7 movl \$16, -4(%rbp) 8 movl -8(%rbp), %eax 9 popq %rbp 10 ret
foo (int* x) { return *x + 7;	A- ⚡ ▾ + ↻ 1 foo: 2 pushq %rbp 3 movq %rsp, %rbp 4 movq %rdi, -8(%rbp) 5 movq -8(%rbp), %rax 6 movl (%rax), %eax 7 addl \$7, %eax 8 popq %rbp 9 ret
foo (int* x) { return x[0] + 7;	A- ⚡ ▾ + ↻ 1 foo: 2 pushq %rbp 3 movq %rsp, %rbp 4 movq %rdi, -8(%rbp) 5 movq -8(%rbp), %rax 6 movl (%rax), %eax 7 addl \$7, %eax 8 popq %rbp 9 ret
foo (int* x) { return *(x + 3) + 7;	A- ⚡ ▾ + ↻ 1 foo: 2 pushq %rbp 3 movq %rsp, %rbp 4 movq %rdi, -8(%rbp) 5 movq -8(%rbp), %rax 6 addq \$12, %rax 7 movl (%rax), %eax 8 addl \$7, %eax 9 popq %rbp 10 ret
foo (int* x) { return x[3] + 7;	A- ⚡ ▾ + ↻ 1 foo: 2 pushq %rbp 3 movq %rsp, %rbp 4 movq %rdi, -8(%rbp) 5 movq -8(%rbp), %rax 6 addq \$12, %rax 7 movl (%rax), %eax 8 addl \$7, %eax 9 popq %rbp 10 ret

```

struct rat {
    int nom;
    int den;

    foo ( struct rat r ) {
        return r.nom + r.den;
    }
}

```

```

1   foo:
2       pushq  %rbp
3       movq  %rsp, %rbp
4       movq  %rdi, -8(%rbp)
5       movl  -8(%rbp), %edx
6       movl  -4(%rbp), %eax
7       addl  %edx, %eax
8       popq  %rbp
9       ret

```

Struct

```

struct rat {
    int nom;
    int den;

    foo ( struct rat* r ) {
        return r->nom + r->den;
    }
}

```

```

1   foo:
2       pushq  %rbp
3       movq  %rsp, %rbp
4       movq  %rdi, -8(%rbp)
5       movq  -8(%rbp), %rax
6       movl  (%rax), %edx
7       movq  -8(%rbp), %rax
8       movl  4(%rax), %eax
9       addl  %edx, %eax
10      popq  %rbp
11      ret

```

```

foo ( int a ) {
    int x;
    if ( a ) {
        x = 42;
    } else {
        x = 7;
    }
    return x;
}

```

```

1   foo:
2       pushq  %rbp
3       movq  %rsp, %rbp
4       movl  %edi, -20(%rbp)
5       cmpl  $0, -20(%rbp)
6       je    .L2
7       movl  $42, -4(%rbp)
8       jmp   .L3
9   .L2:
10      movl  $7, -4(%rbp)
11   .L3:
12      movl  -4(%rbp), %eax
13      popq  %rbp
14      ret

```

```

foo ( int a ) {
    int x;
    if ( a ) {
        x = 42;
    } else {
        x = 7;
    }
    return x;
}

```

```

1   foo:
2       pushq  %rbp
3       movq  %rsp, %rbp
4       movl  %edi, -20(%rbp)
5       cmpl  $0, -20(%rbp)
6       je    .L2
7       movl  $42, -4(%rbp)
8       jmp   .L3
9   .L2:
10      movl  $7, -4(%rbp)
11   .L3:
12      movl  -4(%rbp), %eax
13      popq  %rbp
14      ret

```

Compares the first source operand with the second source operand and sets the flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

foo (int a) {
int x;
if (a) {
 x = 42;
} else {
 x = 7;
}
return x;

A:

```

1 Checks the state of one or more of the status flags in the EFLAGS register (CF
2 SF, and ZF) and, if the flags are in the specified state (condition), performs a ju
3 get target instruction specified by the destination operand. A condition code (cc) is
4 used with each instruction to indicate the condition being tested for. If the condition is
5 satisfied, the jump is not performed and execution continues with the instruction
6 after the Jcc instruction.
7 More information available in the context menu.
8     je      .L2
9     movl    $42, -4(%rbp)
10    jmp     .L3
11    .L2:
12    movl    $7, -4(%rbp)
13    .L3:
14    movl    -4(%rbp), %eax
15    popq    %rbp
16    ret

```

foo (int a) {
int x;
if (a) {
 x = 42;
} else {
 x = 7;
}
return x;

A:

```

1 foo:
2     pushq  %rbp
3     movq   %rsp, %rbp
4     movl   %edi, -20(%rbp)
5     cmpl   $0, -20(%rbp)
6     je     .L2
7     movl   $42, -4(%rbp)
8     jmp    .L3
9     .L2:
10    movl   $7, -4(%rbp)
11    .L3:
12    movl   -4(%rbp), %eax
13    popq    %rbp
14    ret

```

foo (int a) {
int x;
if (a) {
 goto L0;
} else {
 goto L1;
}

x = 42;
goto L2;

x = 7;

return x;

A:

```

1 foo:
2     pushq  %rbp
3     movq   %rsp, %rbp
4     movl   %edi, -20(%rbp)
5     cmpl   $0, -20(%rbp)
6     je     .L7
7     nop
8     movl   $42, -4(%rbp)
9     jmp    .L5
10    .L7:
11    nop
12    movl   $7, -4(%rbp)
13    .L5:
14    movl   -4(%rbp), %eax
15    popq    %rbp
16    ret

```

foo (int a) {
int x;
while (a) {
 x++;
 a--;
}
return x;

A:

```

1 foo:
2     pushq  %rbp
3     movq   %rsp, %rbp
4     movl   %edi, -20(%rbp)
5     jmp    .L2
6     .L3:
7     addl   $1, -4(%rbp)
8     subl   $1, -20(%rbp)
9     .L2:
10    cmpl   $0, -20(%rbp)
11    jne    .L3
12    movl   -4(%rbp), %eax
13    popq    %rbp
14    ret

```

Optimizations, compiler

Performance Realities

big-O, etc.

There's more to performance than asymptotic complexity

- Constant factors matter too! order of magnitude
 - Easily see 10:1 performance range depending on how code is written
 - Must optimize at multiple levels:
 - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
 - How programs are **compiled** and **executed**
 - How modern **processors** + **memory** systems operate
 - How to measure program **performance** and identify **bottlenecks**
 - How to improve performance without destroying code **modularity** and **generality**

Optimizing Compilers, Limitations

compiler is conservative

- Operate under fundamental constraint
 - Must **not** cause any **change** in program **behavior**
 - Except, possibly when program making use of nonstandard language features
 - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
 - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
 - Whole-program analysis is too expensive in most cases
 - Newer versions of GCC do interprocedural analysis within individual files
 - But, not between code in different files
- Most analysis is based only on *static* information
 - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

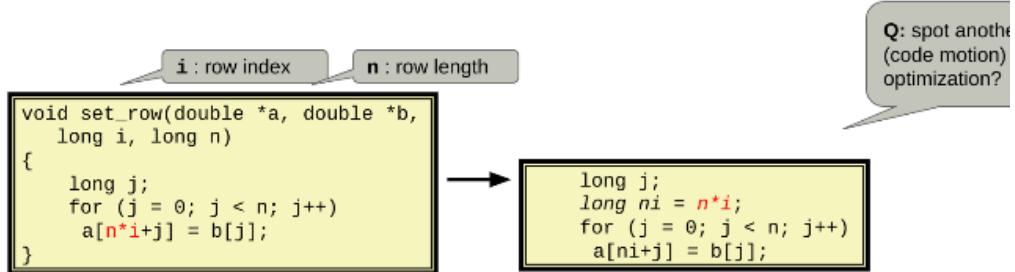
Dead Code Elimination

despite this, compiler can help.

```
int junk ( int n ) {  
    int k = 0;  
    for (int i = 0; i <= n; i++){  
        k += i;  
    }  
    return 4;  
}
```

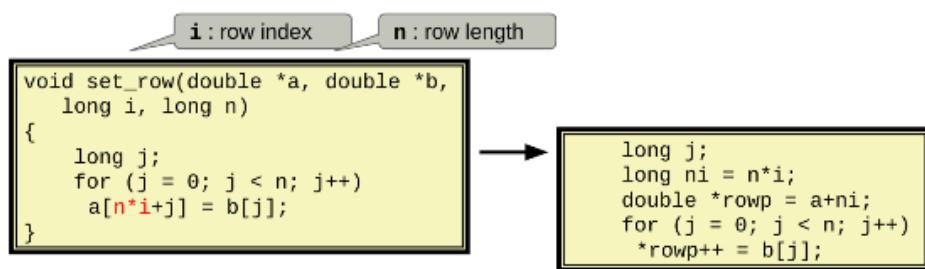
Code Motion

- Reduce frequency with which computation is performed
 - If it will always produce same result
 - Especially moving code out of loop



Code Motion

- Reduce frequency with which computation is performed
 - If it will always produce same result
 - Especially moving code out of loop



Strength Reduction

shift (1 cycle
is way more
efficient than
multiplication
(3 cycles)

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
 - Utility machine dependent
 - Depends on cost of multiply or divide instruction
 - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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

→

```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n;  
}
```

(think of
multiplication
as a
sequence of
additions)

UNIVERSITY OF COPENHAGEN

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with -O1

```
/* Sum neighbors of i,j */  
up = val[(i-1)*n + j];  
down = val[(i+1)*n + j];  
left = val[i*n + j-1];  
right = val[i*n + j+1];  
sum = up + down + left + right;
```

3 multiplications: $i \cdot n$, $(i-1) \cdot n$, $(i+1) \cdot n$

```
long inj = i*n + j;  
up = val[inj - n];  
down = val[inj + n];  
left = val[inj - 1];  
right = val[inj + 1];  
sum = up + down + left + right;
```

1 multiplication: $i \cdot n$

```
leaq 1(%rsi), %rax # i+1  
leaq -1(%rsi), %r8 # i-1  
imulq %rcx, %rsi # i*n  
imulq %rcx, %rax # (i+1)*n  
imulq %rcx, %r8 # (i-1)*n  
addq %rdx, %rsi # i*n+j  
addq %rdx, %rax # (i+1)*n+j  
addq %rdx, %r8 # (i-1)*n+j
```

```
imulq%rcx, %rsi # i*n  
addq %rdx, %rsi # i*n+j  
movq %rsi, %rax # i*n+j  
subq %rcx, %rax # i*n+j-n  
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

Optimizations manual

Calling Strlen

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

Q: lower is quite slow. why?

- strlen performance
 - Only way to determine length of string is to scan its entire length, looking for null character
- Overall performance, string of length N
 - N calls to strlen
 - Require times N, N-1, N-2, ..., 1
 - Overall $O(N^2)$ performance

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

UNIVERSITY OF COPENHAGEN

Strlen takes linear time

Improving Performance

```
void lower2(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

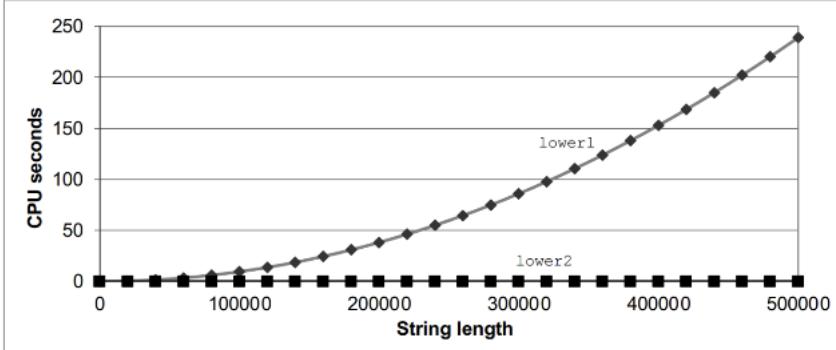
Move call to strlen outside of loop

Since result does not change from one iteration to another

Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker: Procedure Calls

Here is what an optimizing compiler does to this code

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

UNIVERSITY OF COPENHAGEN

- Compiler won't move `strlen` out of inner loop.

Why won't it?

Procedure may have side effects

- Alters global state each time called

Function may not return same value for given arguments

- Depends on other parts of global state

- Procedure `lower` could interact with `strlen`

- **Warning:**

Compiler treats procedure call as a black box

Weak optimizations near them

- **Remedies:**

Use of inline functions

- GCC does this with `-O1`

Within single file

Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    lencnt += length;
    return length;
}
```

Jnnecessary movs?

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
    movsd  (%rsi,%rax,8), %xmm0      # FP load
    addsd  (%rdi), %xmm0             # FP add
    movsd  %xmm0, (%rsi,%rax,8)      # FP store
    addq   $8, %rdi
    cmpq   %rcx, %rdi
    jne    .L4
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

why doesn't the compiler keep the intermediate results in a register, and write register to mem when done?
(would be 100x faster)

UNIVERSITY OF COPENHAGEN

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
{ 0, 1, 2,
  4, 8, 16,
 32, 64, 128};

double *B = A+3;
sum_rows1(A, B, 3);
```

Q: first suppose we had
double B[3] = {42, 42, 42};
what are final values in B?

final: [3, 28, 224]

Q: now, for B as defined on left,
what are intermediate & final B?

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

we just updated part of A

reading & writing to same row

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

A and b might be overlapping

Optimization Blocker: Memory Aliasing

Aliasing

Two **different memory references** specify **single location**

Easy to have happen in C

- Since allowed to do address arithmetic
- Direct access to storage structures

Get in habit of introducing **local variables**

- **Accumulating within loops**
- **Your way of telling compiler not to check for aliasing**

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 27, 16]

i = 2: [3, 22, 224]

```
# sum_rows2 inner loop
.L10:
    addsd (%rdi), %xmm0 # FP load + add
    addq $8, %rdi
    cmpq %rax, %rdi
    jne .L10
```

no mov instruction;
100x faster.

Now **val** cannot be an alias for cells in **a**. (in inner loop)

No need to store intermediate results

Take-aways

- Understand compiler optimizations
⇒ You can help the compiler do them for you.
- Don't do anything stupid
Watch out for hidden algorithmic inefficiencies
Write compiler-friendly code
 - Watch out for optimization blockers:
procedure calls & memory references
- Look carefully at innermost loops (where most work is done)
- Tune code for machine
Exploit instruction-level parallelism
Avoid unpredictable branches
Make code cache friendly (see last week => **blocking**)

Threads

Reading

Multithreading

Dive into systems - 14.1-14.2? (not 14.7)

Programming multicore systems

Each process executes in its own virtual address space

The operating system schedules processes for execution on the PCU

- A context switch occurs when the CPU changes which process is currently executes

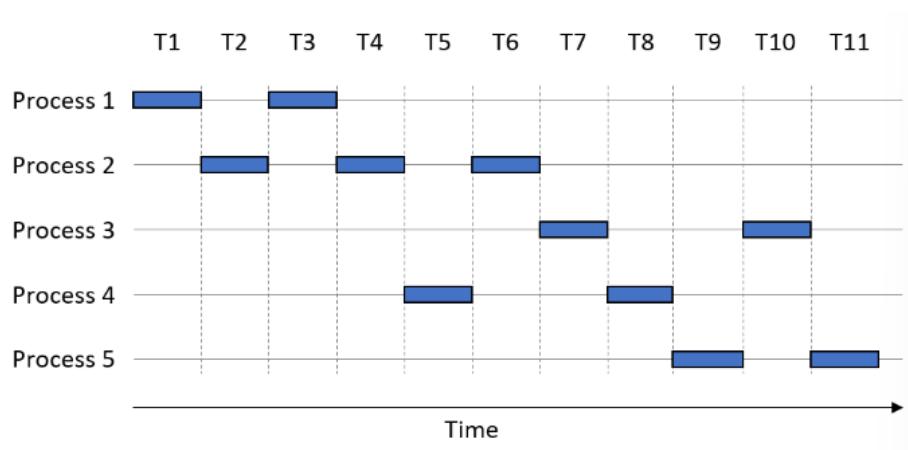


Figure 206. An execution time sequence for five processes as they share a single CPU core

Box is a single core CPU.

CPU time

- Measures the amount of time a process takes to execute on a CPU

Walk-clock time

- Measures the amount of time a human perceives a process to take to complete

Process 1

- CPU time: 2 time units
- Wall-clock time: 3 time units

Process 1 and 2 run concurrently

- Overlap at time points T2-T4

Multicore

- Allows execution simultaneously

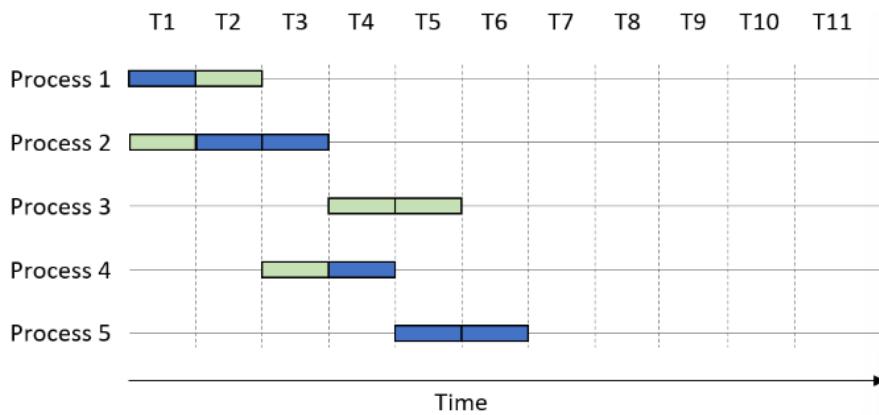


Figure 207. An execution time sequence for five processes, extended to include two CPU cores (one in dark blue, the other in light green).

Same order as before, but enables them to execute sooner.

Multicore increases the throughput of process execution.

Speed up execution by decomposing it into lightweight independent execution flows: threads

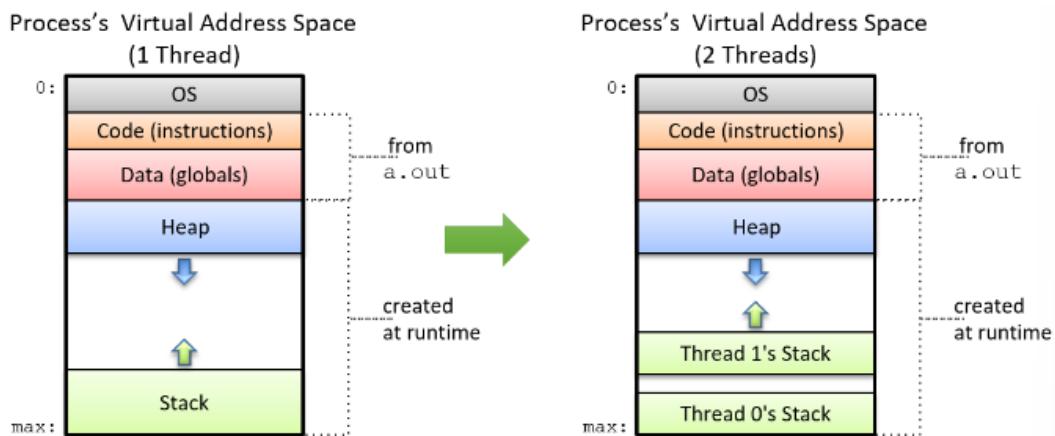


Figure 208. Comparing the virtual address space of a single-threaded and a multithreaded process with two threads

Operating system schedules threads in the same manner as it does processes

- Run on separate cores
- Max of threads is equal to the number of physical cores

- o If it exceeds they must wait to execute

A serial implementation of a scalar multiplication function follows:

```
void scalar_multiply(int * array, long length, int s) {
    int i;
    for (i = 0; i < length; i++) {
        array[i] = array[i] * s;
    }
}
```

Suppose that `array` has N total elements. To create a multithreaded version of this application with t threads, it is necessary to:

1. Create t threads.
2. Assign each thread a subset of the input array (i.e., N/t elements).
3. Instruct each thread to multiply the elements in its array subset by s .

To make 4 threads -> assign each thread one fourth of the total input array

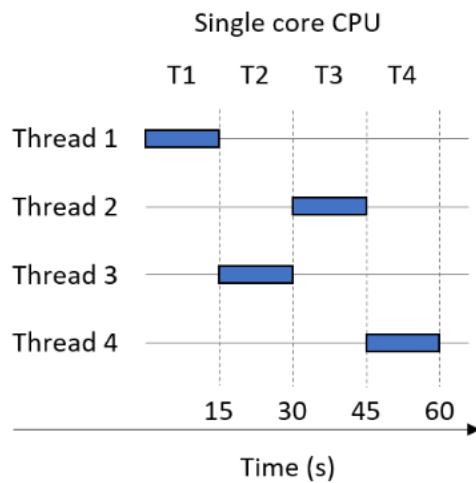


Figure 209. Running four threads on a single-core CPU

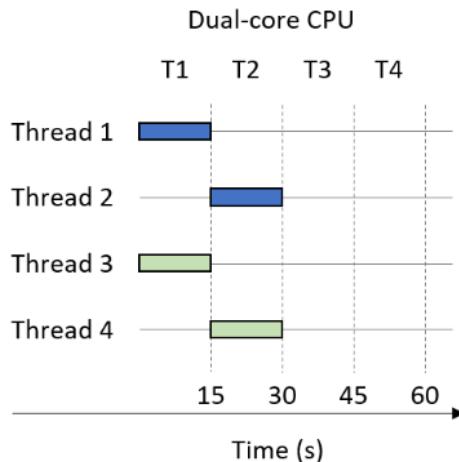


Figure 210. Running four threads on a dual-core CPU

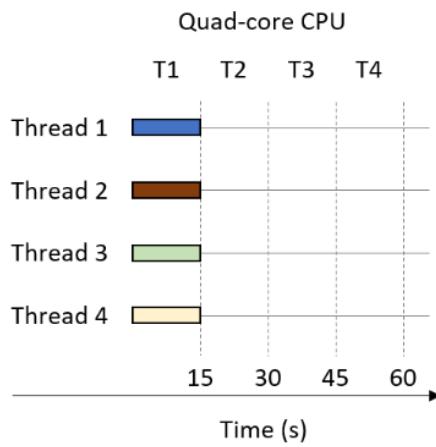


Figure 211. Running four threads on a quad-core CPU

Hello threading! Writing your first multithreaded program

Posix pthreads library

Posix -> portable operating system interface

```

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

/* The "thread function" passed to pthread_create. Each thread executes this
 * function and terminates when it returns from this function. */
void *HelloWorld(void *id) {

    /* We know the argument is a pointer to a long, so we cast it from a
     * generic (void *) to a (long *). */
    long *myid = (long *) id;

    printf("Hello world! I am thread %ld\n", *myid);

    return NULL; // We don't need our threads to return anything.
}

int main(int argc, char **argv) {
    int i;
    int nthreads; //number of threads
    pthread_t *thread_array; //pointer to future thread array
    long *thread_ids;

    // Read the number of threads to create from the command line.
    if (argc != 2) {
        fprintf(stderr, "usage: %s <n>\n", argv[0]);
        fprintf(stderr, "where <n> is the number of threads\n");
        return 1;
    }
    nthreads = strtoll(argv[1], NULL, 10);

    // Allocate space for thread structs and identifiers.
    thread_array = malloc(nthreads * sizeof(pthread_t));
    thread_ids = malloc(nthreads * sizeof(long));

    // Assign each thread an ID and create all the threads.
    for (i = 0; i < nthreads; i++) {
        thread_ids[i] = i;
        pthread_create(&thread_array[i], NULL, HelloWorld, &thread_ids[i]);
    }

    /* Join all the threads. Main will pause in this loop until all threads
     * have returned from the thread function. */
    for (i = 0; i < nthreads; i++) {
        pthread_join(thread_array[i], NULL);
    }

    free(thread_array);
    free(thread_ids);

    return 0;
}

```

Void*: anonymous pointer

- Allows to write thread functions that deals with arguments and return values of different types

After all the preliminary variables are allocated and initialized, the main thread executes the two major steps of multithreading:

- The **creation** step, in which the main thread spawns one or more worker threads. After being spawned, each worker thread runs within its own execution context concurrently with the other threads and processes on the system.
- The **join** step, in which the main thread waits for all the workers to complete before proceeding as a single-thread process. Joining a thread that has

terminated frees the thread's execution context and resources. Attempting to join a thread that *hasn't* terminated blocks the caller until the thread terminates, similar to the semantics of the [wait function for processes](#).

The Pthreads library offers a pthread_create function for creating threads and a pthread_join function for joining them

The OS schedules the execution of each created thread; the user cannot make any assumption on the order in which the threads will execute.

Avoid global variables in c

Dive into systems - 15.1 (c examples)

Heterogeneous computing: hardware accelerators, GPGPU computing and CUDA

```
/* "returns" through pass-by-pointer param dev_ptr GPU memory of size bytes
 * returns cudaSuccess or a cudaError value on error
 */
cudaMalloc(void **dev_ptr, size_t size);

/* free GPU memory
 * returns cudaSuccess or cudaErrorInvalidValue on error
 */
cudaFree(void *data);

/* copies data from src to dst, direction is based on value of kind
 * kind: cudaMemcpyHostToDevice is copy from cpu to gpu memory
 * kind: cudaMemcpyDeviceToHost is copy from gpu to cpu memory
 * returns cudaSuccess or a cudaError value on error
 */
cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind);
```

CUDA threads are blocks, and blocks are grids

Two dimensional block and grid dimensions:

```
dim3 blockDim(16,16); // 256 threads per block, in a 16x16 2D arrangement
dim3 gridDim(20,20); // 400 blocks per grid, in a 20x20 2D arrangement
```

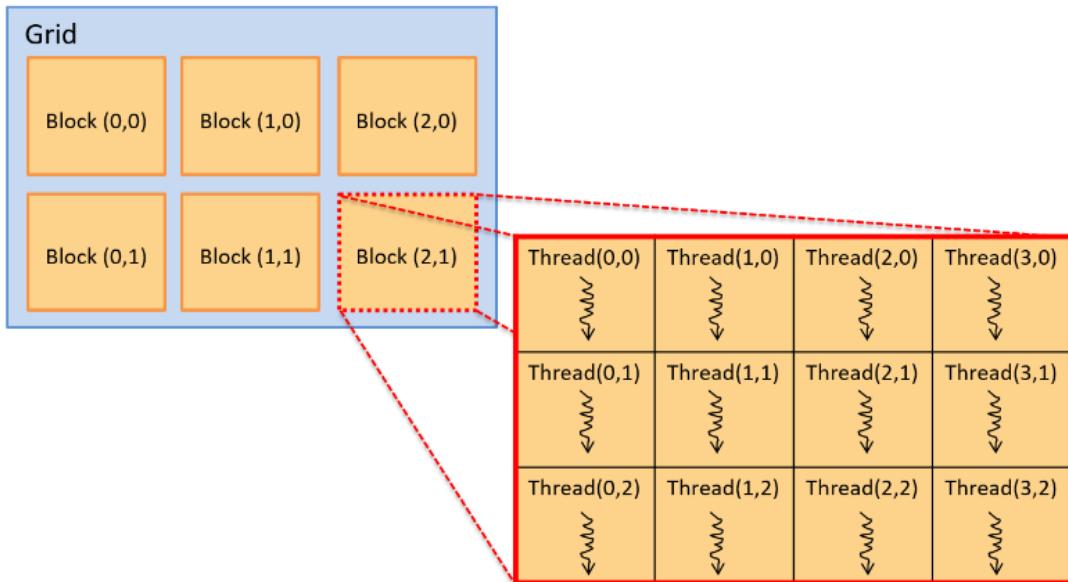


Figure 217. The CUDA thread model. A grid of blocks of threads. Blocks and threads can be organized into one-, two-, or three-dimensional layouts. This example shows a grid of two-dimensional blocks, 3 × 2 blocks per grid, and each block has a two-dimensional set of threads, 4 × 3 threads per block).

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

As an example, consider a CUDA program that performs scalar multiplication of a vector:

```
x = a * x    // where x is a vector and a is a scalar value
```

Because the program data comprises one-dimensional arrays, using a one-dimensional layout of blocks/grid and threads/block works well. This is not necessary, but it makes the mapping of threads to data easier.

When run, the main function of this program will do the following:

1. Allocate host-side memory for the vector `x` and initialize it.
2. Allocate device-side memory for the vector `x` and copy it from host memory to GPU memory.
3. Invoke a CUDA kernel function to perform vector scalar multiply in parallel, passing as arguments the device address of the vector `x` and the scalar value `a`.
4. Copy the result from GPU memory to host memory vector `x`.

The main function of the CUDA³ program performs the four steps listed above:

```
#include <cuda.h>

#define BLOCK_SIZE      64      /* threads per block */
#define N              10240    /* vector size */

// some host-side init function
void init_array(int *vector, int size, int step);

// host-side function: main
int main(int argc, char **argv) {

    int *vector, *dev_vector, scalar;

    scalar = 3;      // init scalar to some default value
    if(argc == 2) { // get scalar's value from a command line argument
        scalar = atoi(argv[1]);
    }

    // 1. allocate host memory space for the vector (missing error handling)
    vector = (int *)malloc(sizeof(int)*N);

    // initialize vector in host memory
    // (a user-defined initialization function not listed here)
    init_array(vector, N, 7);

    // 2. allocate GPU device memory for vector (missing error handling)
    cudaMalloc(&dev_vector, sizeof(int)*N);

    // 3. call the CUDA scalar_multiply kernel
    // specify the 1D layout for blocks/grid (N/BLOCK_SIZE)
    // and the 1D layout for threads/block (BLOCK_SIZE)
    scalar_multiply<<<(N/BLOCK_SIZE), BLOCK_SIZE>>>(dev_vector, scalar);

    // 4. copy device vector to host memory (missing error handling)
    cudaMemcpy(vector, dev_vector, sizeof(int)*N, cudaMemcpyDeviceToHost);

    // ...(do something on the host with the result copied into vector)

    // free allocated memory space on host and GPU
    cudaFree(dev_vector);
    free(vector);

    return 0;
}
```

Optimization

Dive into systems - 14.4-14.5

Measuring performance of parallel programs

Basics

Speedup

A program takes T_c time to execute on c cores, a serial version would take T_1 time

The speedup of the program on c cores is then expressed by the equation:

$$Speedup_c = \frac{T_1}{T_c}$$

Example:

Serial program takes 60 seconds.

Parallel version takes 30 seconds on 2 cores

Speedup is 2

Ideal: running on n cores with n threads has speedup of n

If greater than 1, it has yielded improvement

A program can have speedup greater than n = superlinear speedup

Efficiency

Speedup doesn't take metric on cores = program on 60 seconds, that takes 30 on 4 cores still has speedup 2

To measure the speedup per core, use efficiency:

$$Efficiency_c = \frac{T_1}{T_c \times c} = \frac{Speedup_c}{c}$$

Varies from 0 - 1

1: cores are being used perfectly

Greater than 1 = 1 superlinear speedup

Critical path

- The longest set of dependencies in a program

Reducing it improves performance

Running a function from an algorithm with different amount of cores

```
$ ./countElems_p_v3 100000000 0 1
Time for Step 1 is 0.331831 s

$ ./countElems_p_v3 100000000 0 2
Time for Step 1 is 0.197245 s

$ ./countElems_p_v3 100000000 0 4
Time for Step 1 is 0.140642 s

$ ./countElems_p_v3 100000000 0 8
Time for Step 1 is 0.107649 s
```

Table 130. Performance Benchmarks

Number of threads	2	4	8
Speedup	1.68	2.36	3.08
Efficiency	0.84	0.59	0.39

While we have 84% efficiency with two cores, the core efficiency falls to 39% with eight cores. Notice that the ideal speedup of 8 was not met. One reason for this is that the overhead of assigning work to threads and the serial update to the `counts` array starts dominating performance at higher numbers of threads. Second, resource contention by the eight threads (remember this is a quad-core processor) reduces core efficiency.

Amdahl's law

Maximum speedup that a program can achieve is limited by the size of its necessary serial component.

For every program, there is a component that can be sped up, and one that cannot be sped up.

Consider a program that executes on one core in time T_1 . Then, the fraction of the program execution that is necessarily serial takes $S \times T_1$ time to run, and the parallelizable fraction of program execution ($P = 1 - S$) takes $P \times T_1$ to run.

Maximum improvement for the parallel processor with c cores to run the same job

$$T_c = S \times T_1 + \frac{P}{c} \times T_1$$

More cores will become dominated by the serial part.

Table 131. The Effect of Amdahl's Law on a 10-Second Program that is 90% Parallelizable

Number of cores	Serial time (s)	Parallel time (s)	Total Time (T_c s)	Speedup (over one core)
1	1	9	10	1
10	1	0.9	1.9	5.26
100	1	0.09	1.09	9.17
1000	1	0.009	1.009	9.91

A more formal way to look at this requires incorporating Amdahl's calculation for T_c into the equation for speedup:

$$Speedup_c = \frac{T_1}{T_c} = \frac{T_1}{S \times T_1 + \frac{P}{c} \times T_1} = \frac{T_1}{T_1(S + \frac{P}{c})} = \frac{1}{S + \frac{P}{c}}$$

Advanced topics

Gustafson-barsis law

Critical assumption in Amdahl's law that's not always true

The number of compute cores c and the fraction of a program that is parallelizable P are independent of each other.

- Virtually never the case

Assume run time and not problem size to be constant

- The amount of work that can be done in parallel varies linearly with the number of processors

Consider a *parallel* program that takes time T_c to run on a system with c cores.

Let S represent the fraction of the program execution that is necessarily serial and takes $S \times T_c$ time to run. Thus, the parallelizable fraction of the program execution, $P = 1 - S$, takes $P \times T_c$ time to run on c cores.

$$SSpeedup_c = \frac{T_1}{T_c} = \frac{S \times T_c + P \times T_c \times c}{T_c} = \frac{T_c(S + P \times c)}{T_c} = S + P \times c$$

Scaled speedup increases linearly with the number of compute units

Consider our prior example in which 99% of a program is parallelizable (i.e., $P = 0.99$). Applying the scaled speedup equation, the theoretical speedup on 100 processors would be 99.01. On 1,000 processors, it would be 990.01. Notice that the efficiency stays constant at P .

Scalability

Scalable:

- If performance improves as we add more resources
 - Cores
 - Processors

Strong scaling

- Increasing the number of resources on a fixed problem size yields an improvement in performance
- Run on n cores the speedup is also n

Weak scaling

- Increasing size of data at the same rate as cores result in constant or an improvement of performance
- Improvement n If the work per core is scaled up by a factor of n

General advice

Run the program multiple times to get the correct benchmark.

Be careful executing multiple threads on the same core.

Cache coherence and false sharing

Basic concepts in cache design

- Data/instructions are not transported *individually* to the cache. Instead, data is transferred in *blocks*, and block sizes tend to get larger at lower levels of the memory hierarchy.

- Each cache is organized into a series of sets, with each set having a number of lines. Each line holds a single block of data.
- The individual bits of a memory address are used to determine which set, tag, and block offset of the cache to which to write a block of data.
- A **cache hit** occurs when the desired data block exists in the cache. Otherwise, a **cache miss** occurs, and a lookup is performed on the next lower level of the memory hierarchy (which can be cache or main memory).
- The **valid bit** indicates if a block at a particular line in the cache is safe to use. If the valid bit is set to 0, the data block at that line cannot be used (e.g., the block could contain data from an exited process).
- Information is written to cache/memory based on two main strategies. In the **write-through** strategy, the data is written to cache and main memory simultaneously. In the **write-back** strategy, data is written only to cache and gets written to lower levels in the hierarchy after the block is evicted from the cache.

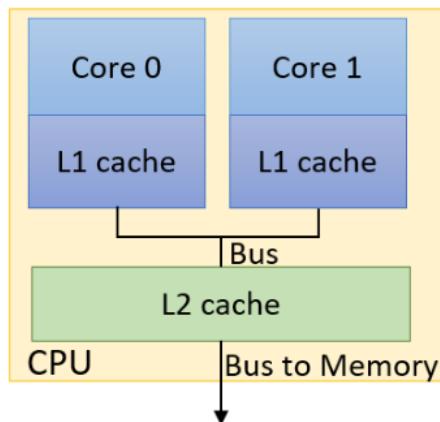


Figure 213. An example dual-core CPU with separate L1 caches and a shared L2 cache

Cache coherence should prevent shared variables from being updated inconsistently

Table 132. Problematic Data Sharing Due to Caching

Time	Core 0	Core 1
0	$g = 5$	(other work)
1	(other work)	$y = g^4$
2	$x += g$	$y += g^2$

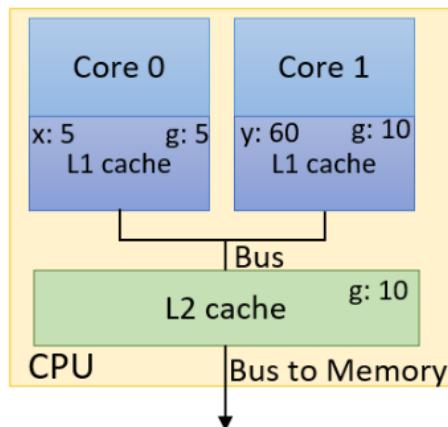


Figure 214. A problematic update to caches that do not employ cache coherency

Common implementation of MSI

- Snoopy cache
 - Snoops on the memory bus for possible write signals

Cache coherence guarantees correctness but can harm performance

Snoopy cache

- Invalidates not only g, but the ENTIRE cache line that g is a part of

Can cause thrashing

- Repeated conflicts in the cache causes a series of misses

One way to fix

- Pad the array so it doesn't fit in a single cache line

However can waste memory

Better way

- Threads write to local storage

The memory that is local to a thread

So then only triggered if write to shared values in memory, then ensure only one thread updates the shared value at a time

What every programmer should know about memory - 6.4.1 & 6.4.3

Concurrency optimizations

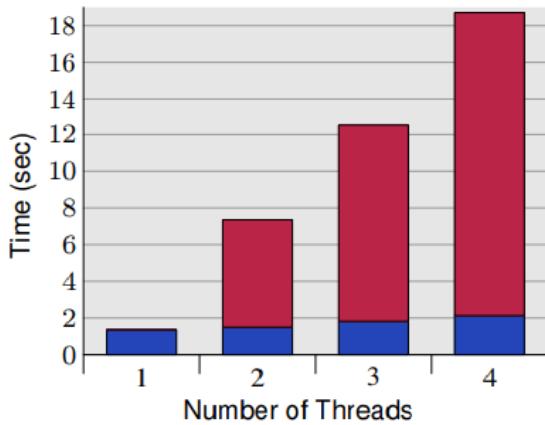


Figure 6.10: Concurrent Cache Line Access Overhead

Blue is running with individual cache lines

Red is when everything Is on 1 cache line

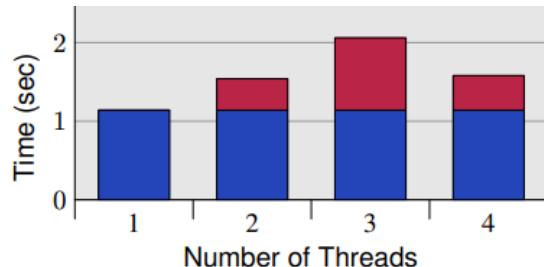


Figure 6.11: Overhead, Quad Core

Running on a single processor, even with its two different L2 caches, it does not show any scalability issues

Slight overhead when doing it more than once, but does not increase with the number of cores

Putting every variable in its own cache line is a fix, but the footprint of the application would increase a lot

Constants (those that are never only initialized once and or never written to) can be shared in the cache state S.

You can create thread-local variables using the `_thread` keyword

- When a thread is created it spends time of setting up the variables
- Requires time and memory

Advice:

- Separate at least read-only (after initialization) and read-write variables. Maybe extend this separation to read-mostly variables as a third category.
- Group read-write variables which are used together into a structure. Using a structure is the only way to ensure the memory locations for all of those variables are close together in a way which is translated consistently by all gcc versions..
- Move read-write variables which are often written to by different threads onto their own cache line. This might mean adding padding at the end to fill a remainder of the cache line. If combined with step 2, this is often not really wasteful.
- If a variable is used by multiple threads, but every use is independent, move the variable into TLS.

Bandwidth considerations

Multiple processors might share the same bus to memory

Even in perfect condition, it cannot fulfil all load and store requests without waiting

Divide available bandwidth by number of cores, hyper-threads and processors sharing a connection

One solution

Get a faster computer

- Better to work on the program than this

After optimizing for cache misses

- Achieve better bandwidth utilization
- Place threads better on available cores

Scheduler does not know anything about workload

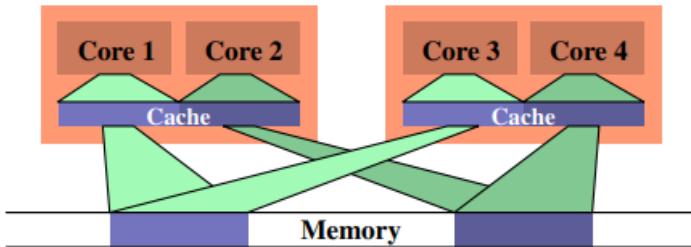


Figure 6.13: Inefficient Scheduling

Each dataset has to be read twice from memory

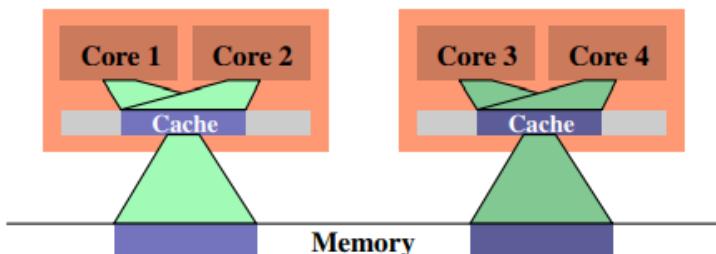


Figure 6.14: Efficient Scheduling

Total cache size in use is reduced

Thread affinity

- Assigning a thread to one or more cores

Multithreaded programs: the individual threads have no process ID

Working set of two threads overlaps such that having both threads on the same core makes sense

- If they work on separate sets, it would be a problem to have them work on the same core

Solution: sort the affinity of the threads so they cannot be scheduled on the same core

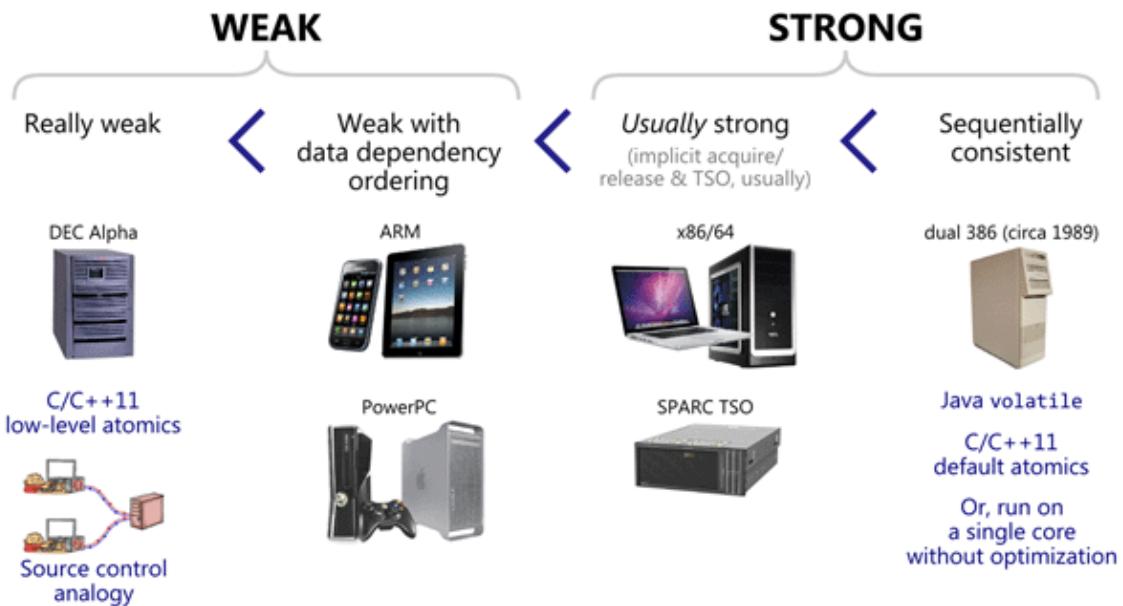
Concurrency control (atomics and locks)

Weak vs strong memory models

A **memory model** tells you, for a given processor or toolchain, exactly what types of memory reordering to expect at runtime relative to a given source code listing.

- Can only be observed when lock-free programming are used

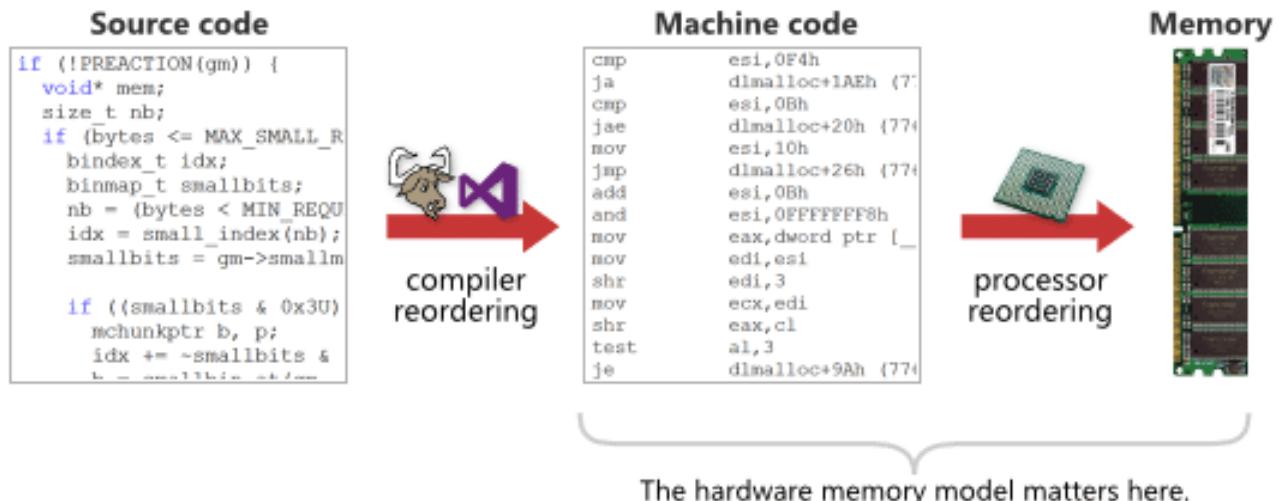
Each model guarantees for the one to the left



Each depicts a hardware memory model

Hardware memory model

- Kind of memory ordering to expect at runtime relative to an assembly code listing



Weak memory models

Possible to experience all four types of memory reordering

Any load or store operation can effectively be reordered with any other load or store operation, as long as it would never modify the behavior of a single, isolated thread

Weak hardware memory model

- Weakly ordered
- Weak ordering
- Relaxed memory model

Example: DEC Alpha - weakly ordered processor

The C11 and C++11 programming languages expose a weak software memory model which was in many ways influenced by the Alpha. When using low-level atomic operations in these languages, it doesn't matter if you're actually targeting a strong processor family such as x86/64

(C11 is C from 2011)

Weak with data dependency ordering

- **ARM**, which is currently found in hundreds of millions of smartphones and tablets, and is increasingly popular in multicore configurations.
- **PowerPC**, which the Xbox 360 in particular has already delivered to 70 million living rooms in a multicore configuration.
- **Itanium**, which Microsoft no longer supports in Windows, but which is still supported in Linux and found in HP servers.

However, they maintain data dependency ordering

- If we write A -> B, we are guaranteed the load value of B

Strong memory models

A **strong hardware memory model** is one in which every machine instruction comes implicitly with [acquire and release semantics](#). As a result, when one CPU core performs a sequence of writes, every other CPU core sees those values change in the same order that they were written.

X86/64 is usually strongly-ordered

Sequential consistency

In a sequential consistent memory model there is no memory reordering.

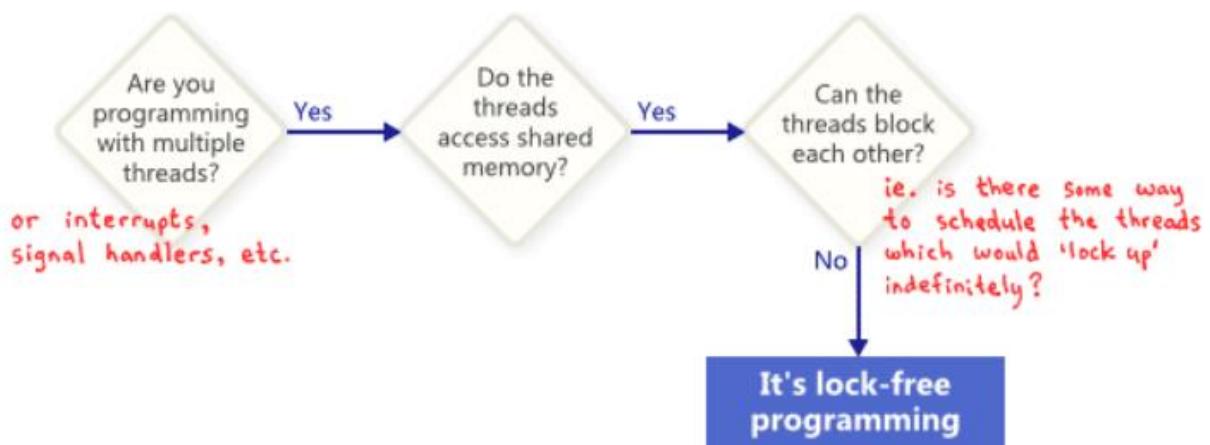
Interesting when working in higher-level programming languages.

An introduction to lock-free programming

What is it?

Programming without mutexes(locks)

Lock-free is a property used to describe some code, without saying too much about how that code was actually written



Operation which contains no mutexes, but is still not lock-free

```

while (X == 0)
{
    X = 1 - X;
}

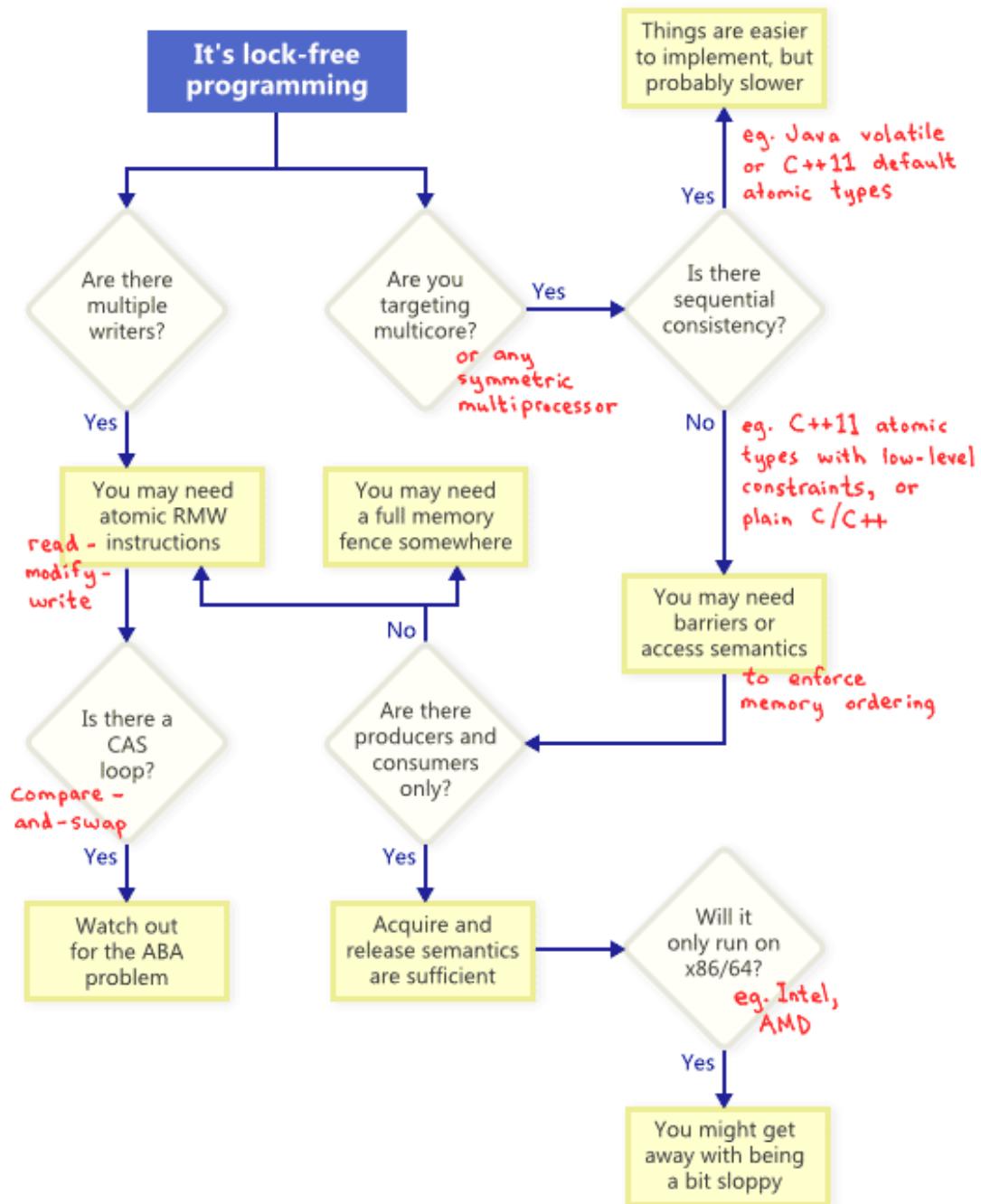
```

“In an infinite execution, infinitely often some method call finishes.” In other words, as long as the program is able to keep *calling* those lock-free operations, the number of *completed* calls keeps increasing, no matter what. It is algorithmically impossible for the system to lock up during those operations.

If you suspend a single thread, it will never prevent other threads from making progress as a group.

Lock-free programming techniques

Atomic operations, memory barriers, avoiding the ABA problem etc



Atomic read-modify-write operations

Manipulate memory in a way that appears invisible

- No thread can observe the operation half-complete

Useful when a lock-free algorithm must support

- Multiple writers
- Since when multiple threads attempt on the same address they line up in a row

- Execute one at a time

Atomic RMW's are necessary part of lock-free programming even on single-processor systems.

Without atomicity, a thread could be interrupted halfway through the transaction, possibly leading to an inconsistent state.

Compare-and-swap loops

Most often discussed RMW operation

Typically

- Copying a shared variable to a local variable
- Performing some speculative work
- Attempt to publish the changes using CAS

```
void LockFreeQueue::push(Node* newHead)
{
    for (;;)
    {
        // Copy a shared variable (m_Head) to a local.
        Node* oldHead = m_Head;

        // Do some speculative work, not yet visible to other threads.
        newHead->next = oldHead;

        // Next, attempt to publish our changes to the shared variable.
        // If the shared variable hasn't changed, the CAS succeeds and we return.
        // Otherwise, repeat.
        if (_InterlockedCompareExchange(&m_Head, newHead, oldHead) == oldHead)
            return;
    }
}
```

If the test fails for one thread, it means It must have succeeded for another

- Still lock-free

Sequential consistency

Impractical way

- Force all threads to run on 1 processor

Memory ordering

Consider how to prevent memory reordering.

3 categories - prevent compiler reordering AND processor reordering

- Lightweight sync or fence instruction
- Full memory fence instruction
- Memory operations which provide acquire or release semantics

Acquire semantics prevent memory reordering of operations which follow it in program order, and release semantics prevent memory reordering of operations preceding it.

These semantics are particularly suitable in cases when there's a producer/consumer relationship, where one thread publishes some information and the other reads it

Different processors have different memory models

at the x86/64 instruction level, every load from memory comes with acquire semantics, and every store to memory provides release semantics – at least for non-SSE instructions and non-write-combined memory. As a result, it's been common in the past to write lock-free code which works on x86/64, but [fails on other processors](#).

Is parallel programming hard, and, if so, what can you do about it? - c.1-c.5

Cache structure

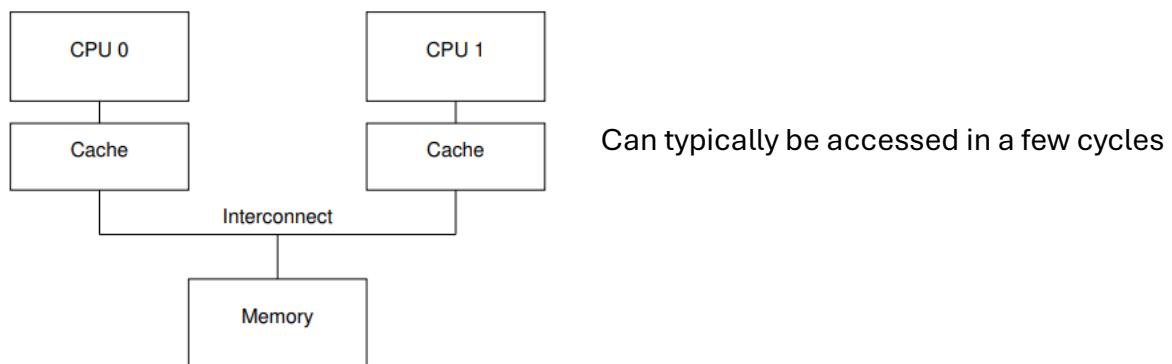


Figure C.1: Modern Computer System Cache Structure

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

large caches are implemented as hardware hash tables with fixed-size hash buckets (or “sets”, as CPU designers call them) and no chaining.

C.2 has 16 sets and 2 ways for 32 lines

- Hardware parlance: two-way set-associative cache
- Software hash table: 16 buckets

Figure C.2: CPU Cache Structure

The size and associativity is called the caches geometry

Cache-coherence protocols

Manage cache-line states to prevent inconsistent or lost data

Four-state MESI cache-coherence protocol

MESI

- Modified
- Exclusive
- Shared
- Invalid

Is given a cache line

Caches using this maintain a two-bit state "tag" on each line

Caches in the modified state

- Subject to a recent memory store
- Owned by the CPU
- Cache holds the only up to date copy, responsible for writing it back or handing off to another cache

Caches in exclusive state

- Similar to modified
 - Cache line has not YET been modified
- Can discard its data without writing back

Cache in shared state

- Line might be replicated in at least one other CPU's cache
- Not permitted to store to the line without consulting other CPI's
- Can discard data without writing back

Cache in invalid

- Is empty = holds not data
- When new data comes, its put into a line from invalid

Protocol provides messages that coordinate the movement of lines through system

MESI messages

Read

- Contains physical address of the line to be read

Read response

- Contains data requested by earlier read
- May be supplied by memory or other caches

Invalidate

- Contains physical address of line to be invalidated
- Other caches must remove the data from their caches and respond

Invalidate acknowledge

- CPU receiving Invalidate must respond after removing the data

Read invalidate

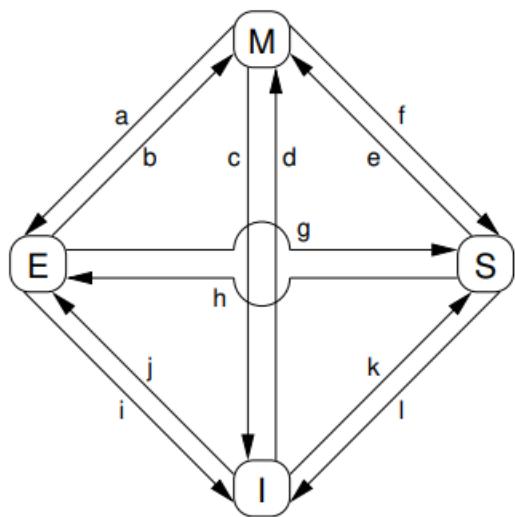
- Contains physical address of the line to be read, at the same directing other caches to remove the data
- Combination of read and invalidate

- Requires both read response and a set of invalidate acknowledge

Writeback

- Contains address and data to be written back to memory

MESI state diagram



A = line written back to memory, but CPU retains it in its cache, and retains the right to modify it - requires writeback message

B = CPU writes to the line it already had exclusive access to - does not require any message

C = CPU receives read invalidate, must invalidate its own copy - respond with both read response and invalidate acknowledge

Figure C.3: MESI Cache-Coherency State Diagram

D = CPU does atomic read-modify-write on a data item not present in its cache. - transmits read invalidate and receive the data via read response, can complete once received full set of invalidate acknowledge responses

E = atomic read-modify-write on an item that was previously read-only in its cache - transmit invalidate and wait for full set of invalidate acknowledge responses

F = other CPU reads cache line and is supplied from this CPU's cache, which retains a read-only copy, maybe writing back to memory - receive a read message, and this CPU responds with read response

G = other CPU reads data in its cache line and is supplied from this CPU's cache or from memory. Retains a read-only copy - receive read and sends read response containing requested data

H = CPU realizes it soon needs to write to some item in its cache line - transmits invalidate. Cannot complete until full set of invalidate acknowledge, making sure no other CPU has this cache line in its cache

I = other CPU does atomic read-modify-write on item in a cache line held only in this CPU's cache, so this CPU invalidates it from its cache - receive read invalidate and respond with read response and invalidate acknowledge

J = CPU does a store to item in line that was not in cache - transmits read invalidate.
Cannot complete until receive read response and full set of invalidate acknowledge.
Transition to modified state via b as soon as complete

K = CPU loads item in line that was not in its cache - transmits read and completes
when receiving read response

L = other CPU does a store to item in its cache line, but holds this line in read-only state
due to it being held in other CPU's caches (such as the current CPU's cache - receive
invalidate and this CPU responds with invalidate acknowledge)

Stores result in unnecessary stalls

Cache structure in C1 is bad in performance for the first write to a cache line.

Since CPU 0 must wait for the cache line to arrive before it can write to it, CPU 0 must
stall for an extended period of time.

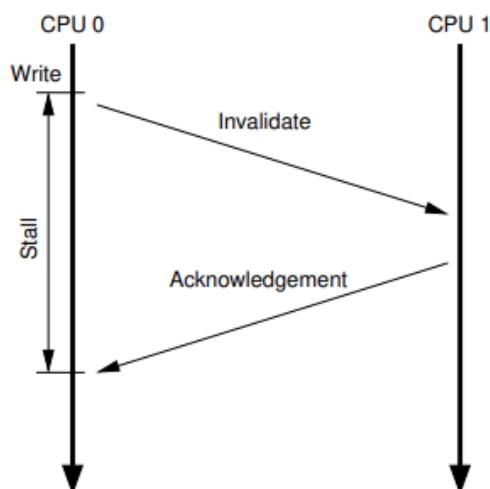


Figure C.4: Writes See Unnecessary Stalls

Regardless of what data happens to be in the cache line that CPU 1 sends it, CPU 0 is
going to unconditionally overwrite it

Store buffers

Preventing unnecessary stalling of write

- Between each CPU and its cache

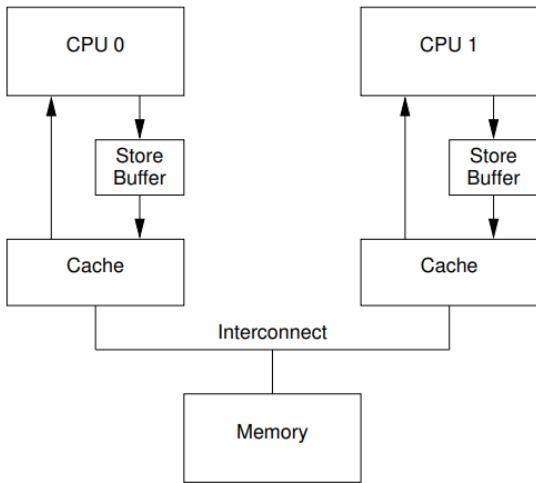


Figure C.5: Caches With Store Buffers

CPU 0 can record its write in its store buffer and continue executing

When cache line makes its way from 1 to 0 the data will be moved from the store buffer to the cache line

Store buffer entry need only contain the value stored, not the other data contained in the corresponding cache line

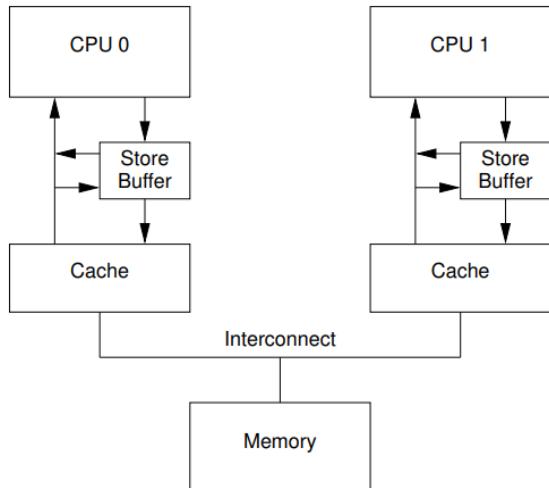
- The CPU doing the store has no idea what the other data might be

A CPU can only access the store buffer assigned to it

- Simplifies hardware by separating concerns

Improves performance for consecutive writes.

Store forwarding



Problem: Having two copies of a variable, one in cache one in buffer

Figure C.6: Caches With Store Forwarding

Store buffers and memory barriers

Problem: not knowing which variables are related, or how they are related

```

1 void foo(void)
2 {
3     a = 1;
4     b = 1;
5 }
6
7 void bar(void)
8 {
9     while (b == 0) continue;
10    assert(a == 1);
11 }
```

Provide memory-barrier instructions to allow the software to tell the CPU about the relations

```

1 void foo(void)
2 {
3     a = 1;
4     smp_mb();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0) continue;
11    assert(a == 1);
12 }
```

Store sequence Result in unnecessary stalls

Each store buffer must be relatively small

- Allows CPU to fill the buffer way to quickly

Can be improved by making invalidate acknowledge messages arrive more quickly

Invalidate queues

The messages take long since they have to wait for the cache line to be invalidated, it can be delayed if the cache is busy

However it does no need to invalidate the cache line before sending the acknowledgement

- Could queue the message with the understanding it will be processed before any further messages is send

Invalidate queues and invalidate acknowledge

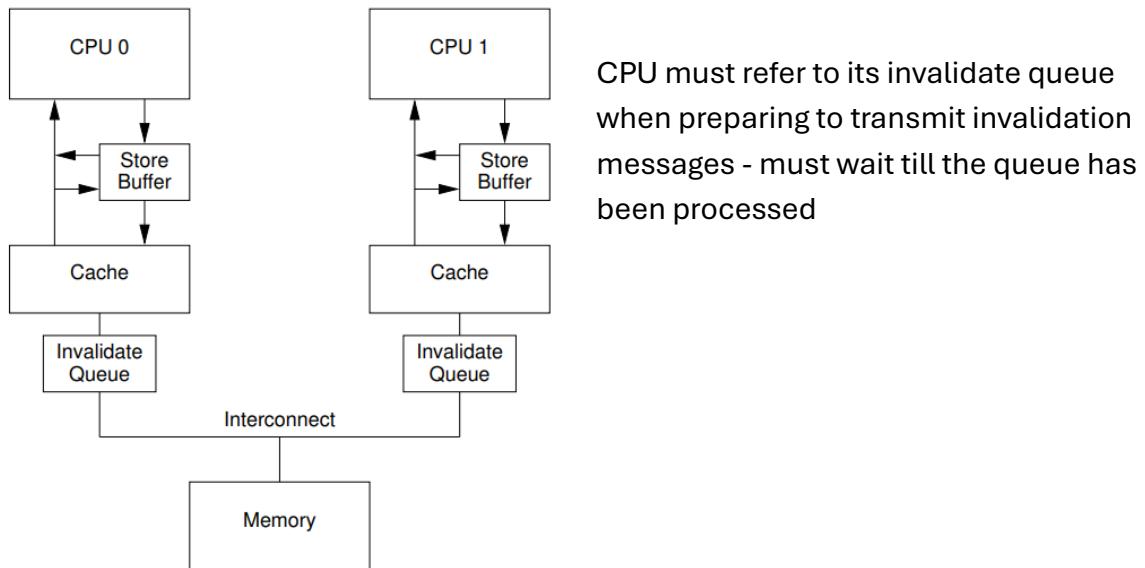


Figure C.7: Caches With Invalidate Queues

Invalidate queues and memory barriers

CPU's send the invalidation requests to a queue

- Minimizes cache-invalidation latency

No point in accelerating invalidation responses if doing so causes memory barriers to effectively be ignored

However, the memory-barrier instructions can interact with the invalidate queue, so that when a given CPU executes a memory barrier, it marks all the entries currently in its invalidate queue, and forces any subsequent load to wait until all marked entries have been applied to the CPU's cache

Be careful with cache-coherence optimizations

Read and write memory barriers

Weaker memory-barrier instructions.

Read memory barriers orders only loads on the CPU that executes it.

- All loads preceding will appear to have completed before any load following the read memory barrier

A full-fledged memory barrier orders both loads and stores

What does the LOCK instruction mean in x86 assembly?

I saw some x86 assembly in Qt's source:

```

q_atomic_increment:
    movl 4(%esp), %ecx
    lock
    incl (%ecx)
    mov $0,%eax
    setne %al
    ret

.align 4,0x90
.type q_atomic_increment,@function
.size   q_atomic_increment,.-q_atomic_increment

```

1. From Googling, I knew `lock` instruction will cause CPU to lock the bus, but I don't know when CPU frees the bus?
2. About the whole above code, I don't understand how this code implements the `Add` ?

1. `LOCK` is not an instruction itself: it is an instruction prefix, which applies to the following instruction. That instruction must be something that does a read-modify-write on memory (`INC`, `XCHG`, `CMPXCHG` etc.) --- in this case it is the `incl (%ecx)` instruction which increments the `long` word at the address held in the `ecx` register.

The `LOCK` prefix ensures that the CPU has exclusive ownership of the appropriate cache line for the duration of the operation, and provides certain additional ordering guarantees. This may be achieved by asserting a bus lock, but the CPU will avoid this where possible. If the bus is locked then it is only for the duration of the locked instruction.

2. This code copies the address of the variable to be incremented off the stack into the `ecx` register, then it does `lock incl (%ecx)` to atomically increment that variable by 1. The next two instructions set the `eax` register (which holds the return value from the function) to 0 if the new value of the variable is 0, and 1 otherwise. The operation is an **increment**, not an add (hence the name).

What every programmer should know about memory - 6.4.2

Atomicity optimizations

If multiple threads modify the same memory location concurrently, processors do not guarantee any specific result

If a memory location is in the ‘S’ state and two threads concurrently have to increment its value, the execution pipeline does not have to wait for the cache line to be available in the ‘E’ state before reading the old value from the cache to perform the addition. Instead it reads the value currently in the cache and, once the cache line is available in state ‘E’, the new value is written back.

If the two cache reads in the threads happen simultaneously, one addition will be lost

When concurrent operations can happen - processors provide atomic operations

- Does not read old value until its clear that the addition to memory location appears atomic
- Some signal atomic operations for addresses to other devices on motherboard

This makes atomic operations slower

Bit test

Set or clear a bit atomically and return a status of whether the bit was set before or not

Load lock/store conditional (LL/SC)

A pair, load instruction is used to start transaction, final store will only succeed if not modified

Success or failure so program can repeat if necessary

Compare-and-swap (CAS)

Ternary operation

Writes a value (parameter) into address (second parameter) if the current value is the same as the third.

Atomic arithmetic

Only on x86 and x86-64

Perform arithmetic and logic operations on memory locations

Architecture can have EITHER LL/SC OR CAS, not both

- They are equivalent

Atomic addition

```
int curval;
int newval;
do {
    curval = var;
    newval = curval + addend;
} while (CAS(&var, curval, newval));
```

CAS indicates succeed or not

- Failure: its run again

LL/SC looks almost the same

```
int curval;
int newval;
do {
    curval = LL(var);
    newval = curval + addend;
} while (SC(var, newval));
```

Uses special load instruction (LL) and does not pass current value of memory location to SC, since it know if it's been modified in the meantime

Important to select the proper operation to achieve best result

3 different ways to implement atomic increment operation, all produce different code - huge performance differences

```

for (i = 0; i < N; ++i)    for (i = 0; i < N; ++i)    for (i = 0; i < N; ++i) {
    __sync_add_and_fetch(&var, 1);    __sync_fetch_and_add(&var, 1);    long v, n;
}                                }                                do {
}                                }                                v = var;
                                         n = v + 1;
                                         | while (!__sync_bool_compare_and_swap(&var,
                                         v, n));
                                         |
1. Add and Read Result      2. Add and Return Old Value      3. Atomic Replace with New Value

```

Figure 6.12: Atomic Increment in a Loop

Execution time for 1 million increment by four concurrent threads

1. Exchange Add	2. Add Fetch	3. CAS
0.23s	0.21s	0.73s

Cost when using CAS is much more expensive

- There are two memory operations
- CAS itself is more complicated and requires conditional operation
- Whole operation has to be done in a loop
 - In case two concurrent accesses cause the CAS call to fail

Complexity is usually hidden

- Makes programs simpler

Execution of just 2 threads, each on its own core

Thread #1	Thread #2	var Cache State
v = var		'E' on Proc 1
n = v + 1	v = var	'S' on Proc 1+2
CAS(var)	n = v + 1	'E' on Proc 1
	CAS(var)	'E' on Proc 2

Cache line status changes at least 3 times

- 2 changes are RFO's
- Second will fail, so repeat

When done, processor can keep the load and store operations

- Ensure that concurrently-issued cache line requests are blocked until atomic operation is done

Dive into systems - 14.3 & 14.6

Synchronizing threads

A threads ability to share data with other threads is one of its main features

All the threads of a multithreaded process share the heap common to the process

Thread synchronization

- Forcing threads to execute in a particular order
- Even though this can add to the runtime, it is often necessary to ensure program correctness

Data races

Execution of any thread can be pre-empted at any time by the OS, which means that each thread could be running different instructions of a particular function at any given time

To better illustrate what is going on, we translated the line $\text{counts}[\text{val}] = \text{counts}[\text{val}] + 1$ into the following sequence of equivalent instructions:

1. **Read** $\text{counts}[\text{val}]$ and place into a register.
2. **Modify** the register by incrementing it by one.
3. **Write** the contents of the register to $\text{counts}[\text{val}]$.

This is read-modify-write

Table 128. A Possible Execution Sequence of Two Threads Running countElems

Time	Thread 0	Thread 1
i	Read counts[1] and place into Core 0's register	...
$i+1$	Increment register by 1	Read counts[1] and place into Core 1's register
$i+2$	Overwrite counts[1] with contents of register	Increment register by 1
$i+3$...	Overwrite counts[1] with contents of register

An operation is defined as being **atomic** if a thread perceives it as executing without interruption (in other words, as an "all or nothing" action)

Not all executions of the program can cause a race condition

Table 129. Another Possible Execution Sequence of Two Threads Running countElems

Time	Thread 0	Thread 1
i	Read counts[1] and place into Core 0's register	...
$i+1$	Increment register by 1	...
$i+2$	Overwrite counts[1] with contents of register	...
$i+3$...	Read counts[1] and place into Core 1's register
$i+4$...	Increment register by 1
$i+5$...	Overwrite counts[1] with contents of register

To fix a data race we

- Isolate critical section or the subset of code that must execute atomically
- In threaded programs, blocks of code that update a shared resource are identified as critical sections

Mutual exclusion

Mutual exclusion lock (mutex)

- Synchronization primitive that ensures only one thread enters and executes the code inside the critical section at any given time

Before using it, the program must first

- Declare the mutex in memory that's shared by threads (global variable)
- Initialize the mutex before the threads need to use it (typically in main)

The Pthreads library defines a `pthread_mutex_t` type for mutexes

Initial state of mutex is unlocked

- It's immediately usable by any thread
- To enter critical section, a thread must acquire a lock

When it has the lock

- No other thread can enter the critical section until the thread with the lock releases

In addition to protecting the critical section to achieve correct behavior, an ideal solution would use the lock and unlock functions as little as possible, and reduce the critical section to the smallest possible size.

Deadlock, multiple synchronization constructs are incorrectly applied.

Semaphores

Semaphores are commonly used in operating systems and concurrent programs where the goal is to manage concurrent access to a pool of resources. When using a semaphore, the goal isn't *who* owns what, but *how many* resources are still available. Semaphores are different from mutexes in several ways:

- Semaphores need not be in a binary (locked or unlocked) state. A special type of semaphore called a *counting semaphore* can range in value from 0 to some r , where r is the number of possible resources. Any time a resource is produced, the semaphore is incremented. Any time a resource is being used, the semaphore is decremented. When a counting semaphore has a value of 0, it means that no resources are available, and any other threads that attempt to acquire a resource must wait (e.g., block).

- Semaphores can be locked by default.

While a mutex and condition variables can simulate the functionality of a semaphore, using a semaphore may be simpler and more efficient in some cases. Semaphores also have the advantage that *any* thread can unlock the semaphore (in contrast to a mutex, where the calling thread must unlock it).

Barriers

A **barrier** is a type of synchronization construct that forces *all* threads to reach a common point in execution before releasing the threads to continue executing concurrently.

Condition variables

Condition variables force a thread to block until a particular condition is reached. This construct is useful for scenarios in which a condition must be met before the thread does some work. In the absence of condition variables, a thread would have to repeatedly check to see whether the condition is met, continuously utilizing the CPU. Condition variables are always used in conjunction with a mutex.

Thread safety

Not all functions in the C library are **thread safe**, or capable of being run by multiple threads while guaranteeing a correct result without unintended side effects. To ensure that the programs we write are thread safe, it is important to use [synchronization primitives](#) like mutexes and barriers to enforce that multithreaded programs are consistent and correct regardless of how the number of threads varies.

All thread safe code is re-entrant; however, not all re-entrant code is thread safe. A function is **re-entrant** if it can be re-executed/partially executed by a function without causing issue. By definition, re-entrant code ensures that accesses to the global state of a program always result in that global state remaining consistent. While re-entrancy is often (incorrectly) used as a synonym for thread safety, there are special cases for which re-entrant code is not thread safe.

Slides

Concurrent Programming's Goals

1. Performance
Effective use of hardware
2. Productivity
Effective use of Software Dev's time
3. Generality
To lower the cost of low-level concurrency and parallelism

to make our apps utilize multicore,
we use multithreading.

concurrent programming is a way
to manage explicit parallelism

THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."

HEY! GET BACK
TO WORK!

COMPILING!

OH, CARRY ON.

<https://xkcd.com/303/>

cleaner in Go, in fact, Go was
created because of this.

IT UNIVERSITY OF COPENHAGEN

18.11.2020 · 1

Concurrent Programming

caveats, gotchas, & head-scratches

tough, maddening,
fun, \$\$\$

**today: quick overview of
basic synchronization primitives**

to master concurrent programming,
(i.e. to utilize modern HW well),
you need a solid understanding of
basic sync primitives offered by HW.

want more: MSc courses on this.

Practical Concurrent and Parallel Programming (Y1)
Performance of Computer Systems (Data Systems)

IT UNIVERSITY OF COPENHAGEN

18.11.2020 · 2

Outline

C-way of handling things.
concurrency / parallelism is **not** a
feature of C; it's a feature of libc.
(recall: C isn't much; all interesting
stuff is libraries)

- The necessity of concurrent programming
- **The problem with concurrent programming**
- Threads
- Synchronization

Concurrent vs. Parallel Programming

Parallel computing: many calculations, or execution of processes, are carried out **simultaneously**.

Concurrent computing: several processes are in progress at the same time (concurrently) instead of one completing before next starts (sequentially)

to drive home the **difference**:

- concurrent computing is the **illusion** of parallel computing; processes are actually *interleaved*.
- parallel computing **requires** HW support (multiple cores).

important to **understand the difference** (often debated, frequently asked)

why a lecture on concurrent (not parallel): parallel is an optimization of concurrent.

What Makes Concurrent Programming Hard?

1. Identify Parallelizable Tasks:

Identify areas that can be divided into concurrent tasks (ideally independent).

2. Balance:

Tasks should perform equal work of equal value.

3. Data Splitting:

How to split data that is accessed by separate tasks?

4. Data Dependency:

If data dependencies between different tasks =>
Synchronization needed.

5. Testing, Debugging:

Many different execution paths possible, testing & debugging become more difficult.

Concurrency Anomalies

Classical problem classes of concurrent programs:

Race: outcome depends on arbitrary scheduling decisions elsewhere in the system

Example: who gets the last seat on the airplane?

Example: concurrent writes to a global variable

Deadlock: improper resource allocation prevents progress

Example: traffic gridlock

Livelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress

Example: hallway dance (livelock)

Example: people always jump in front of you in line

Outline

- The necessity of concurrent programming
- The problem with concurrent programming
- **Threads**
- Synchronization

Concurrency in C

- Processes (libc)
 - Hard to share resources: Easy to avoid unintended sharing
 - High overhead in adding/removing children
- Threads (libc)
 - Easy to share resources
 - Medium overhead
 - Not much control over scheduling policies
 - Difficult to debug
 - Event orderings not repeatable
- I/O Multiplexing
 - Tedious and low level
 - Total control over scheduling
 - Very low overhead
 - Cannot create as fine grained a level of concurrency
 - Does not make use of multi-core

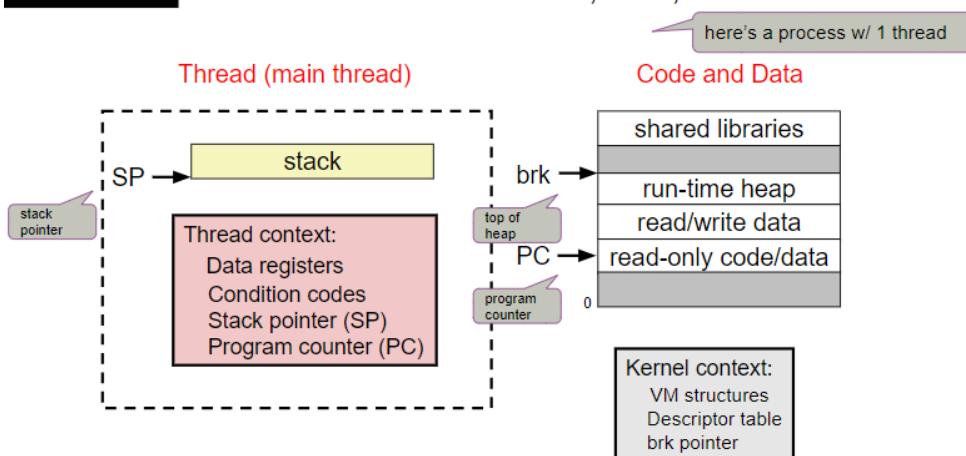
we will talk about these in a later lecture (fork, parent, children, synchronization (wait for each other), sharing across processes)

lighter form of process. instead of 2 processes w/ separate address spaces, you now have 1 process, w/ multiple threads that share address space. (separate stacks*, though)

in Linux, same data structures & mechanisms for these two

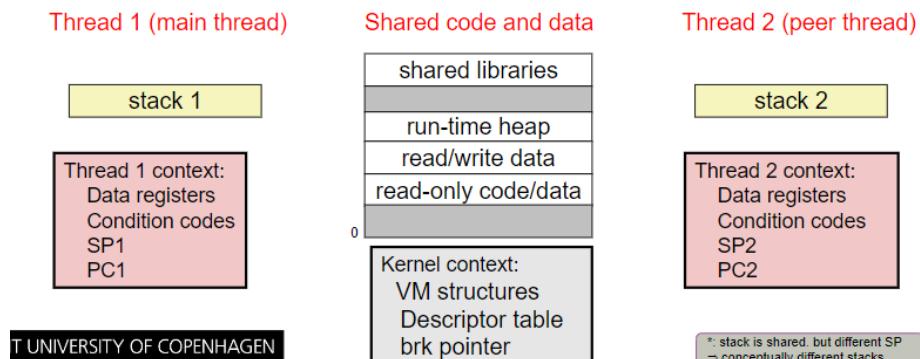
Threads

Process = thread + code, data, and kernel context



Threads

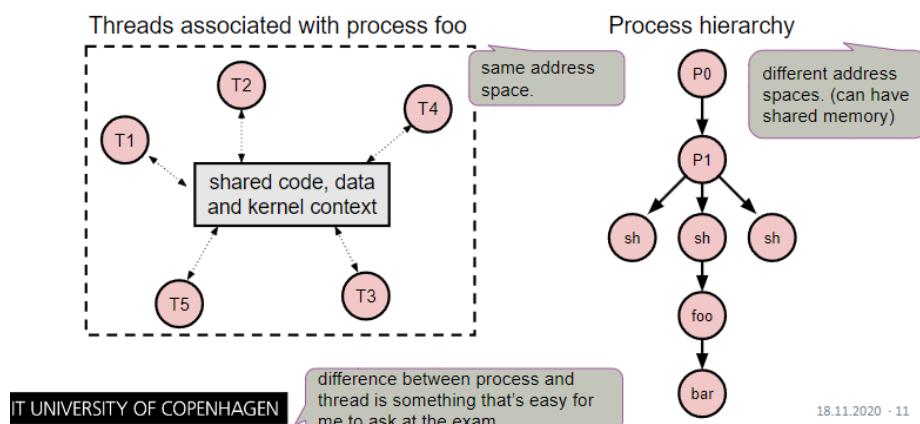
- can have multiple threads in a process
- Multiple threads can be associated with a process
- Each thread has its own logical control flow
- Each thread shares the same code, data, and kernel context
- Share common virtual address space (stack*)
- Each thread has its own thread id (TID)



Threads

Threads associated with process form a pool of peers

Unlike processes which form a tree hierarchy



Thread Execution

illustrating the difference between concurrency and parallelism.

Single Core Processor

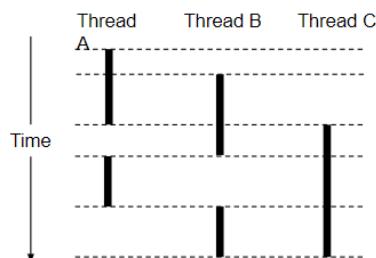
Simulate concurrency by time slicing



IT UNIVERSITY OF COPENHAGEN

Multi-Core Processor

Parallel execution



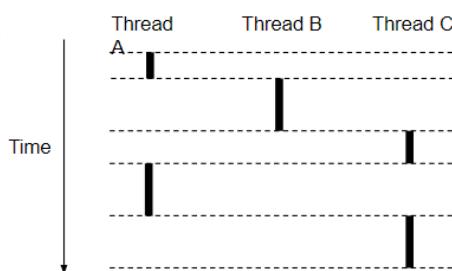
Run 3 threads on 2 cores

Logical Concurrency

- Two threads are (logically) concurrent if their flows overlap in time (otherwise, sequential)

- Examples:

- Concurrent: A & B, A&C
- Sequential: B & C



UNIVERSITY OF COPENHAGEN

Posix Threads (Pthreads) Interface

thread interface in C given by Posix standard.

30s

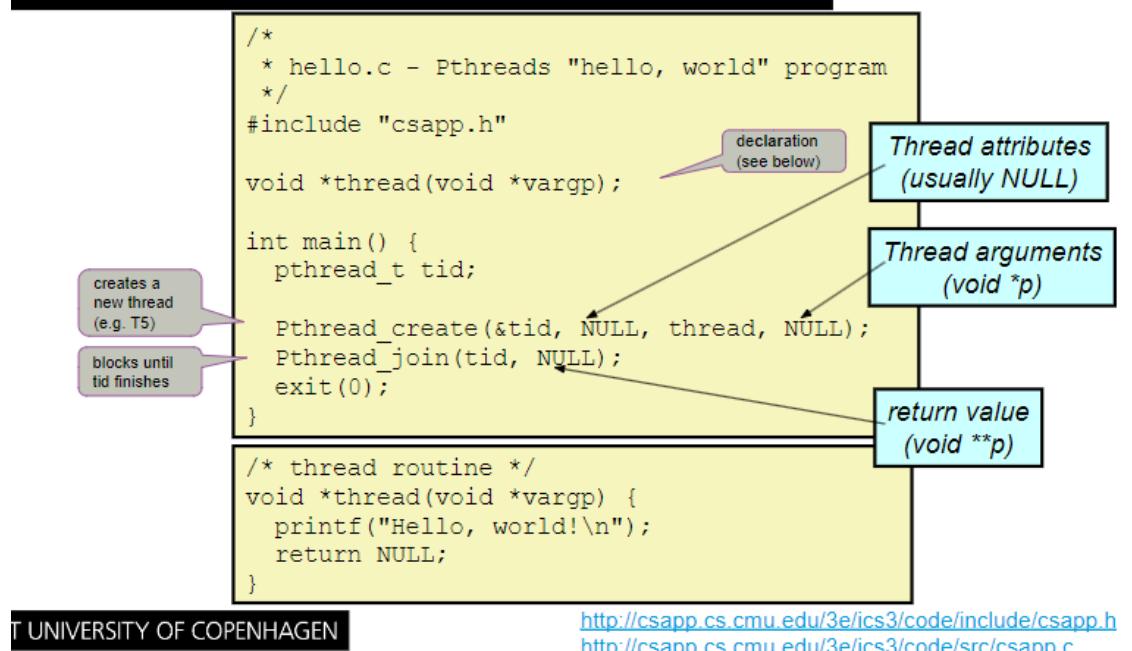
- *Pthreads* library: Standard interface of ~60 functions to manipulate threads from C.
 - Creating and reaping threads
 - `pthread_create()`
 - `pthread_join()`
 - Determining your thread ID
 - `pthread_self()`
 - Terminating threads
 - `pthread_cancel()`
 - `pthread_exit()`
 - `exit()` [terminates all threads], `RET` [terminates current thread]
 - Synchronizing access to shared variables
 - `pthread_mutex_init`
 - `pthread_mutex_[un]lock`
 - `pthread_cond_init`
 - `pthread_cond_[timed]wait`

one criticism of C:
threads are not a
beautiful concept,
but a "fix".

UNIVERSITY OF COPENHAGEN

https://www.gnu.org/software/libc/manual/html_mono/libc.html#POSIX-Threads

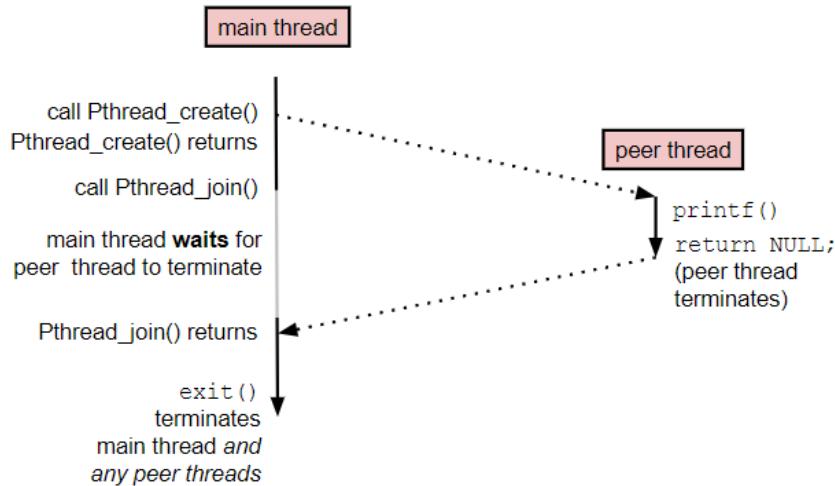
The Pthreads "hello, world" Program



IT UNIVERSITY OF COPENHAGEN

<http://csapp.cs.cmu.edu/3e/ics3/code/include/csapp.h>
<http://csapp.cs.cmu.edu/3e/ics3/code/src/csapp.c>

Execution of Threaded "hello, world"



IT UNIVERSITY OF COPENHAGEN

Shared Variables in Threaded C Programs

Threads must *synchronize* on shared data.

more on this in a bit.
for now:

Question: Which variables in a threaded C program are shared?

The answer is not as simple as
“*global variables are shared*” and
“*stack variables are private*”

Requires answers to the following questions:

- What is the *memory model* for threads?
- How are *instances of variables mapped to memory*?
- How *many threads might reference each of these instances*?

Def: A variable x is *shared* if and only if
multiple threads reference some instance of x .

UNIVERSITY OF COPENHAGEN

Threads Memory Model

Conceptual model:

- Multiple threads run within the context of a single process
- Each thread has its own separate thread context
 - Thread ID, **stack**, **stack pointer**, **PC**, condition codes, GP registers
- All threads share the remaining process context
 - Code, data, heap,
shared library segments of the process virtual address space
 - Open files and installed handlers

Operationally, this model is not strictly enforced:

- Register values are truly separate and protected...
- ... but any thread can read and write the stack of any other thread

*The mismatch between the **conceptual** and **operation** model
is a source of confusion and errors*

Mapping Variable Instances to Memory

Global variables

Def: Variable declared outside of a function

Virtual memory contains exactly one instance of any global variable

Local variables

Def: Variable declared inside function without `static` attribute

Each thread stack contains one instance of each local variable

Local static variables

Def: Variable declared inside function with the `static` attribute

Virtual memory contains exactly one instance of any local static variable.

Example Program to Illustrate Sharing

what is shared?
not obvious.

```
char **ptr; /* global */  
  
int main()  
{  
    int i;  
    pthread_t tid;  
    char *msgs[2] = {  
        "Hello from foo",  
        "Hello from bar"  
    };  
    ptr = msgs;  
  
    for (i = 0; i < 2; i++)  
        Pthread_create(&tid,  
                       NULL,  
                       thread,  
                       (void *)i);  
    Pthread_exit(NULL);  
}
```

```
/* thread routine */  
void *thread(void *vargp)  
{  
    int myid = (int) vargp;  
    static int cnt = 0;  
  
    printf("[%d]: %s (svar=%d)\n",  
          myid, ptr[myid], ++cnt);  
}
```

Peer threads reference main thread's stack
indirectly through global ptr variable

UNIVERSITY OF COPENHAGEN

Mapping Variable Instances to Memory

Global var: 1 instance (ptr [data])

```
char **ptr; /* global */  
  
int main()  
{  
    int i;  
    pthread_t tid;  
    char *msgs[2] = {  
        "Hello from foo",  
        "Hello from bar"  
    };  
    ptr = msgs;  
  
    for (i = 0; i < 2; i++)  
        Pthread_create(&tid,  
                       NULL,  
                       thread,  
                       (void *)i);  
    Pthread_exit(NULL);  
}
```

Local vars: 1 instance (i.m, msgs.m)

Local var: 2 instances (
myid.p0 [peer thread 0's stack],
myid.p1 [peer thread 1's stack])

```
/* thread routine */  
void *thread(void *vargp)  
{  
    int myid = (int)vargp;  
    static int cnt = 0;  
  
    printf("[%d]: %s (svar=%d)\n",  
          myid, ptr[myid], ++cnt);  
}
```

Local static var: 1 instance (cnt [data])

UNIVERSITY OF COPENHAGEN

Shared Variable Analysis

Which variables are shared?

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

when we have
shared variables,
we need
synchronization.

Answer: A variable x is shared iff multiple threads reference some instance of x . Thus:

- **ptr, cnt, and msgs are shared**
- **i and myid are not shared**

IT UNIVERSITY OF COPENHAGEN

Thread Local Storage (TLS)

https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Thread_002dLocal.html#Thread_002dLocal

New storage class keyword: `__thread`

modern version of
libc, new keyword

One instance of the variable per thread

```
__thread int i;  
extern __thread struct state s;  
static __thread char *p;
```

recommendation:
to make clear
*what should be
shared and
what should not,*
use thread-local
storage.

IT UNIVERSITY OF COPENHAGEN

18.11.2020 · 23

Reentrant Functions

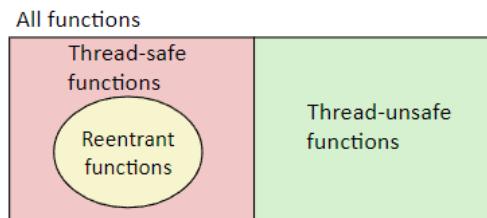
cf. **referential transparency**:
output of function always the same
for a given input.

Def: A function is *reentrant* iff

sanity

it accesses no shared variables when called by multiple threads.

- Important subset of thread-safe functions.
- Require no synchronization operations.



UNIVERSITY OF COPENHAGEN

there is another definition of reentrant which makes it a subset of both thread-safe and thread-unsafe. we will be using the above definition.

Outline

- The necessity of concurrent programming
- The problem with concurrent programming
- Threads
- **Synchronization**

Def: special shared variable that guarantees that a data structure can only be accessed atomically

- Doorbell, Mutex, Conditional variable, Semaphore

HW synchronization

thread synchronization primitives

UNIVERSITY OF COPENHAGEN

18.11.2020 · 26

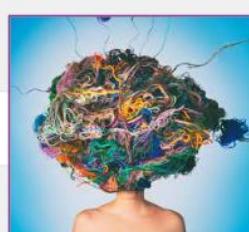
Synchronization Issues

Ancient Greek “ἄτομος”
(atomos, “indivisible”)

Thread 1:

this is just a few assignments to two variables! think about full programs.

```
func foo() {  
    x++;  
    y = x;  
}
```



sequential	ffbb x=10, y= 8 bbff x=10, y=10
bffb	x=10, y= 7
fbbf	x=10, y=10
fbfb	x=10, y= 7

Thread 2:

```
func bar() {  
    y++;  
    x+=3;  
}
```

concurrent (w/ atomic statements)	bffb x=10, y= 7 fbbf x=10, y=10 fbfb x=10, y= 7 bfbf x=10, y=10
concurrent (w/ nonatomic statements) x= 7, y= 7 x=10, y= 1 x= 9, y= 7 ...

If the initial state is x = 6, y = 0,
what happens after these threads finish running?

Q: what are possible final values of x and y?

example of **data race**
aka. **race condition**
(notoriously hard to debug!)

UNIVERSITY OF COPENHAGEN

18.11.2020 · 27

Synchronization Issues

30s

Many things that look like “one step” operations take several steps under the hood:

```
func foo() {  
    eax = mem[x];  
    inc eax;  
    mem[x] = eax;  
    ebx = mem[x];  
    mem[y] = ebx;  
}  
  
func bar() {  
    eax = mem[y];  
    inc eax;  
    mem[y] = eax;  
    eax = mem[x];  
    add eax, 3;  
    mem[x] = eax;  
}
```

to update `mem[x]`: (RMW)
1. read `mem[x]` into register,
2. op on register,
3. write from register to `mem[x]`.
this is **multi-step (non-atomic)**.
`foo` can be in midst, while
`bar` completes the three steps.
⇒ `foo` has stale `mem[x]`
in its register.
(cache coherence (across cores) won't help)
to understand synchronization
issues, must know how code is
mapped to assembly.

When we run a multithreaded program, we don't know what order threads run in, nor do we know when they will be interrupted.

IT UNIVERSITY OF COPENHAGEN

18.11.2020 - 28

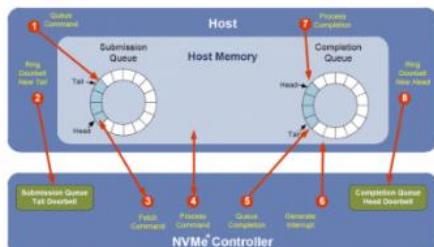
Synchronization is needed when data structures are shared

HW synchronization: PCIe/NVMe Doorbell

side note (low level sync)

30s

doorbell is a boolean register



host software
notifies
storage device
that data is ready
- SQ doorbell
- CQ doorbell

submission queue
completion queue

now, on to
thread synchronization primitives

18.11.2020 - 30

IT UNIVERSITY OF COPENHAGEN

Thread Synchronization

read:
problems

how do threads
even synchronize at the
lowest level?

main reference: →

solutions are **opaque**,
solutions vary
between processors.

I'll give the gist of
common cases.

UNIVERSITY OF COPENHAGEN



section
8.1
(quite opaque)

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 3A:
System Programming Guide, Part 1

NOTE: The Intel® 64 and IA-32 Architectures Software Developer's Manual consists of ten volumes: Basic Architecture, Order Number 253665; Instruction Set Reference A-L, Order Number 253666; Instruction Set Reference M-U, Order Number 253667; Instruction Set Reference V-Z, Order Number 326018; Instruction Set Reference, Order Number 334569; System Programming Guide, Part 1, Order Number 253668; System Programming Guide, Part 2, Order Number 253669; System Programming Guide, Part 3, Order Number 326019; System Programming Guide, Part 4, Order Number 332831; Model-Specific Registers, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

central concepts:

- bus locking
- memory consistency
- cache coherence
(next lecture)

18.11.2020 · 31

Bus Locking & Atomicity

cores share buses.

Q: core 1, 2 do an op
simultaneously;
what happens? (need: **atomicity**)

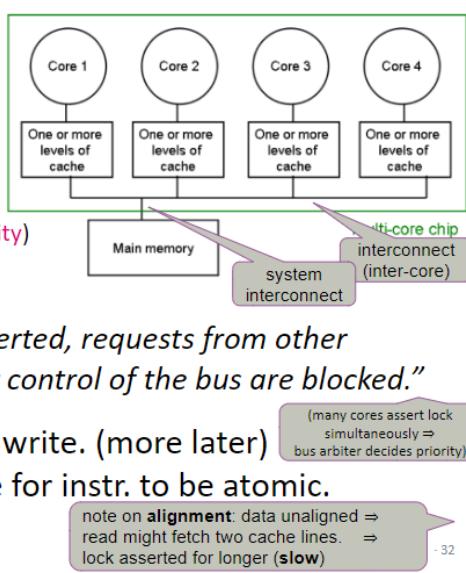
bus locking prevents this.

"While [LOCK#] signal is asserted, requests from other
processors or bus agents for control of the bus are blocked."

guaranteed atomic: read, write. (more later)

can otherwise state desire for instr. to be atomic.

UNIVERSITY OF COPENHAGEN



(many cores assert lock
simultaneously ⇒
bus arbiter decides priority)

note on alignment: data unaligned ⇒
read might fetch two cache lines. ⇒
lock asserted for longer (slow)

- 32

LOCK instruction prefix, example



I saw some x86 assembly in Qt's source:

```
88    q_atomic_increment:
        movl $1,%ecx
        lock
        incl (%ecx)
        movl $0,%eax
        setne %al
        ret

        .align 4,0x90
.type  q_atomic_increment,@function
.size  q_atomic_increment,-q_atomic_increment
```

1. From Googling, I knew `lock` instruction will cause CPU to lock the bus, but I don't know when CPU frees the bus?

2. About the whole above code, I don't understand how this code implements the `Add`?



1. `LOCK` is not an instruction itself: it is an **instruction prefix**, which applies to the following instruction. That instruction must be something that does a **read-modify-write** on memory (`INC`, `XCHG`, `CMPXCHG` etc.) --- in this case it is the `incl (%ecx)` instruction which increments the `long` word at the address held in the `ecx` register.

The `LOCK` prefix ensures that the CPU has exclusive ownership of the appropriate cache line for the duration of the operation, and provides certain additional ordering guarantees. This may be achieved by asserting a bus lock, but the CPU will avoid this where possible. If the bus is locked then it is only for the duration of the locked instruction.

2. This code copies the address of the variable to be incremented off the stack into the `ecx` register, then it does `lock incl (%ecx)` to atomically increment that variable by 1. The next two instructions set the `eax` register (which holds the return value from the function) to 0 if the new value of the variable is 0, and 1 otherwise. **The operation is an increment, not an add** (hence the name).

IT UNIVERSITY OF COPENHAGEN

Instruction reordering due to bus locks

out-of-order execution

ex: access to memory is
100x more expensive
than L1 cache.

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes line	On-Chip L1	1	Hardware
L2 cache	64-bytes line	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

IT UNIVERSITY OF COPENHAGEN

core 1 asserts bus lock to access mem, core 2 wants access to mem
⇒ core 2 must wait for a **really** long time. *next instructions need mem?*

Memory [Consistency] Models

other models: sequential consistency, acquire/release, relaxed.
x86 has a **strong memory model**, w/ a wee bit of reordering.

which instructions reorders can take place?

weak memory model: R/Ws can be reordered arbitrarily as long as behavior of **isolated thread** unaffected.

- compiler, CPU core (\leftarrow weak HW memory model)

sometimes order matters.

ex: NVMe I/O
ex (silly): DMA to robotic surgeon

```
Thread #1 Core #1:  
while (f == 0);  
// Memory fence required here  
print x;  
Thread #2 Core #2:  
x = 42;  
// Memory fence required here  
f = 1;
```

to prevent reordering (when important):

memory barriers. (sync)

volatile keyword in C prevents statement from being reordered / skipped.
(anecdote: password in Windows)

UNIVERSITY OF COPENHAGEN

- 35

Atomic CPU Operations

& Posix

synchronization mechanisms are based on **shared variables** and **atomic instructions**.

- Atomic CPU instructions:
 - Fetch and Add
 - Compare and Swap
 - Test and Set
 - Memory Barrier: operations placed before the barrier are guaranteed to execute before operations placed after the barrier. (aka. "fence")
- In GCC:
<https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html>
 - `__sync_fetch_and_{sub, or, and, xor, nand}()`
 - `__sync_{bool, val}_compare_and_swap()`
 - `__sync_lock_test_and_set, __sync_lock_release`
 - `__sync_synchronize()`

you have abstractions for the above

Mutex, Implementation?

API for providing mutually exclusive access to a resource

Thread 1:

```
critical section {  
    void foo() {  
        mutex.lock();  
        x++;  
        y = x;  
        mutex.unlock();  
    }  
}
```

Thread 2:

```
critical section {  
    void bar() {  
        mutex.lock();  
        y++;  
        x+=3;  
        mutex.unlock();  
    }  
}
```

Global mutex guards access to x & y.

that's nice. how to implement?

IT UNIVERSITY OF COPENHAGEN

Can we do something like this? (easy?)

```
static unsigned int lockvar = 0;  
static void lock() { // acquire  
    while (lockvar) {}  
    lockvar = 1;  
}  
static void unlock() { // release  
    lockvar = 0;  
}
```

No; threads race on `lockvar` (double-acquire)

Need: **Hardware support** to guarantee that operations on synchronization primitives are **atomic**.

instance of **readers-writers problem**:
“no thread may R or W the shared resource while another thread is W-ing to it.”

Mutex

Thread 1:

```
void foo() {  
    mutex.lock();  
    x++;  
    y = x;  
    mutex.unlock();  
}
```

Thread 2:

```
void bar() {  
    mutex.lock();  
    y++;  
    x+=3;  
    mutex.unlock();  
}
```

Global mutex guards access to x & y.

In C:
`pthread_mutex_t lock;`

lock variable

```
pthread_mutex_lock(&lock);  
pthread_mutex_unlock(&lock);
```

(implementation on next slide)

Mutex, sample implementation (w/ spinlock)

now you see
why it's called a
spinlock

thread yields
the core;
someone else
takes over
(hopefully the
thread that
held the lock)

important if you
only have a
single core!

```
static inline void _lock(unsigned int *lock) {
    while (1) {
        int i;
        for (i=0; i < 10000; i++) {
            if (__sync_bool_compare_and_swap(lock, 0, 1)) {
                return;
            }
        }
        sched_yield();
    }
}

static inline void _unlock(unsigned int *lock) {
    __sync_bool_compare_and_swap(lock, 1, 0);
}
```

lock is a datastructure (unsigned int)
which is set to 0 or 1 w/ compare and
swap (atomic).

area of memory

lock is a datastructure (unsigned int)
which is set to 0 or 1 w/ compare and
swap (atomic).

usage: you take the lock before
accessing the shared variable. when
done, you unlock.

guarantee: only 1 thread gets the
lock. (guaranteed by CPU instr.)

Condition Variable

signal threads that a condition is true.
(implemented using mutex)

```
// safely examine the condition, prevent other threads from
// altering it
pthread_mutex_lock (&lock);
while ( SOME-CONDITION is false)
    pthread_cond_wait (&cond, &lock);

// Do whatever you need to do when condition becomes true
do_stuff();
pthread_mutex_unlock (&lock);
```

block the thread. will unblock when
1) signal received on cond, 2) mutex unlocked
after which, the thread will hold the mutex.

```
// ensure we have exclusive access to whatever comprises the condition
pthread_mutex_lock (&lock);
```

ALTER-CONDITION

```
// Wakeup at least one of the threads that are waiting on the condition (if any)
pthread_cond_signal (&cond);

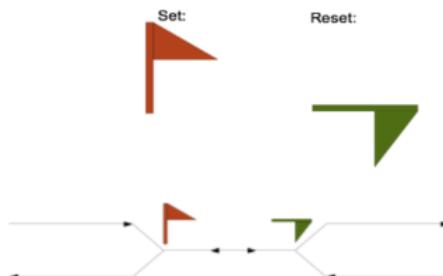
// allow others to proceed
pthread_mutex_unlock (&lock)
```

pthread_cond_broadcast() function shall unblock
all threads currently blocked on the specified
condition variable *cond*.

Semaphore

limit how many threads can access resource at a time: semaphore

- A semaphore is a flag that can be raised or lowered in one step.
- Semaphores were flags that railroad engineers would use when entering a shared track.



For more see [Edsger W. Dijkstra: Cooperating sequential processes](#).

Semaphore

- Semaphore restricts the number of simultaneous threads accessing a shared resource.
 - Semaphore = counter + mutex + wait_queue
- For a binary semaphore (= mutex + conditional variable)
 - **wait()** and **signal()** can be thought of as lock() and unlock()
 - Calls to lock() when the semaphore is already locked cause the thread to block.
- Pitfalls:
 - Must "bind" semaphores to particular objects; must remember to unlock correctly
 - Mutex can only be unlocked by thread that locked it, semaphore can be signaled from any thread => used for synchronization.

Take-Aways

concurrency: **illusion** of parallel

Concurrent Programming is a **necessity** on today's hardware.

Concurrency is **not** a first-class citizen in C; as opposed to languages based on communicating sequential processes (e.g., golang), actor languages (e.g., erlang).

Concurrency in C is based on **multi-threading**.

Communication necessary across threads:

- message passing, shared memory.

Classical problems of concurrent programs:

- races, deadlocks, starvation.

Synchronization primitives needed to avoid problems in concurrent programs:

- mutex, semaphore, conditional variable.

Synchronization primitives require hardware support:

- fetch-and-add, compare-and-swap, test-and-set, memory-barrier.

Other important concepts:

- reentrant, memory model, cache coherence, bus locking, thrashing, critical section

Processes

Reading

Dive into systems - 13.1

How the OS works and how it runs

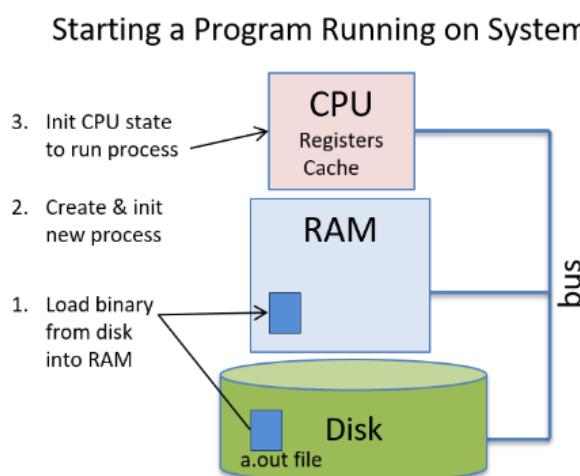


Figure 180. Steps the OS takes to start a new program running on the underlying hardware

OS is also a software that runs on computer hardware, like user programs.

- It requires special software

It then needs help to start running

OS booting

Process of OS loading and initializing itself = booting

(pulls itself up by its bootstraps?)

To initiate the OS code to start running, code stored in computer firmware (nonvolatile memory in the hardware) runs when the computer first powers up; **BIOS** (Basic Input/Output System) and **UEFI** (Unified Extensible Firmware Interface) are two examples of this type of firmware

Getting the OS to do something: interrupts and traps

Are it finishes booting, its ready.

Most OS are -> interrupt-driven systems

- The OS doesn't run until some entity needs it to do something
- Interrupted from its sleep to handle a request

When an applications wants to write to a file, it makes a system call to the OS - wakes up the OS

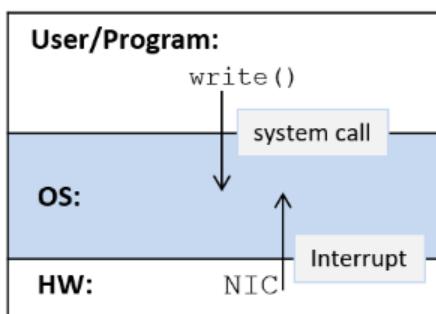


Figure 181. In an interrupt-driven system, user-level programs make system calls, and hardware devices issue interrupts to initiate OS actions.

Hardware interrupts / interrupts = come from hardware layer. could be when a NIC (network interface card) receives data from the network)

Traps = interrupts that come from software layer as a result of instructions executions (could be when a system call is made)

Both can interrupt the OS

For example, a hard disk drive may interrupt the OS if a read fails due to a bad disk block, and an application program may trigger a trap to the OS if it executes a divide instruction that divides by zero.

```
/* C code */
ret = write(fd, buff, size);

# IA32 translation
write:

...          # set up state and parameters for OS to perform write
movl $4, %eax # load 4 (unique ID for write) into register eax
int $0x80    # trap instruction: interrupt the CPU and transition to the OS
addl $8, %ebx # an example instruction after the trap instruction
```

Hardware interrupts are delivered on an interrupt bus - system calls are not

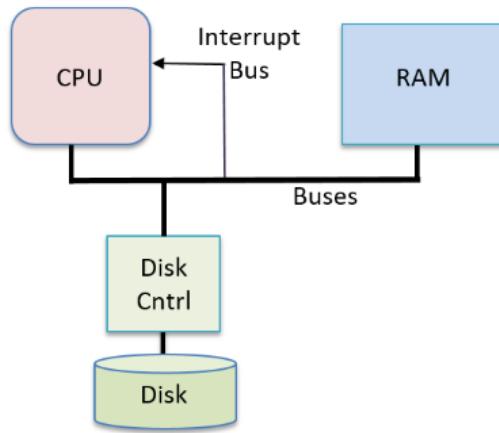


Figure 182. A hardware device (disk) sends a signal to the CPU on the interrupt bus to trigger OS execution on its behalf.

Execution modes

1. In **user mode** a CPU executes only user-level instructions and accesses only the memory locations that the operating system makes available to it. The OS typically prevents a CPU in user mode from accessing the OS's instructions and data. User mode also restricts which hardware components the CPU can directly access.
2. In **kernel mode**, a CPU executes any instructions and accesses any memory location (including those that store OS instructions and data). It can also directly access hardware components and execute special instructions.

When OS code is run on the CPU, the system runs in kernel mode, and when user-level programs run on the CPU, the system runs in user mode. If the CPU is in user mode and receives an interrupt, the CPU switches to kernel mode, fetches the interrupt handler routine, and starts executing the OS handler code. In kernel mode, the OS can access hardware and memory locations that are not allowed in user mode. When the OS is done handling the interrupt, it restores the CPU state to continue executing user-level code at the point at which the program left off when interrupted and returns the CPU back to user mode (see [Figure 183](#)).

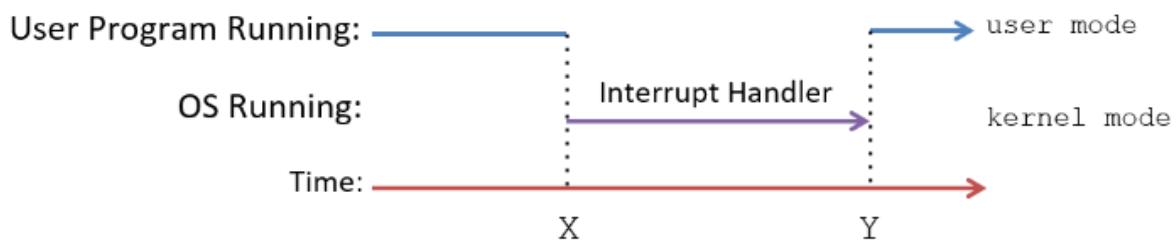


Figure 183. The CPU and interrupts. User code running on the CPU is interrupted (at time X on the time line), and OS interrupt handler code runs. After the OS is done handling the interrupt, user code execution is resumed (at time Y on the time line).

In an interrupt-driven system, interrupts can happen at any time

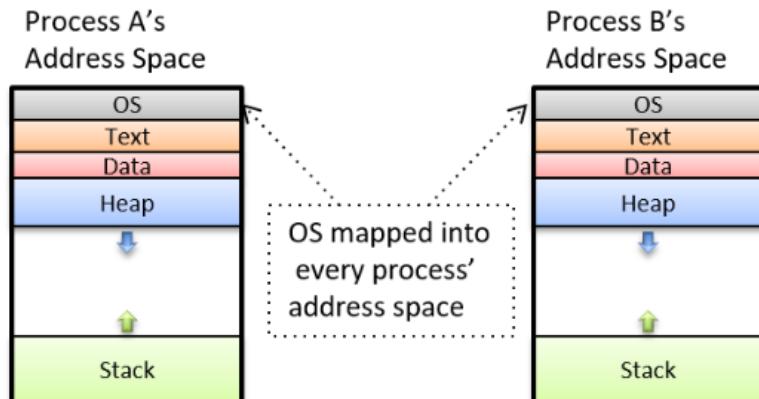


Figure 184. Process address space: the OS kernel is mapped into the top of every process's address space.

What is the difference between Trap and Interrupt?

Traps are a type of exception, and exceptions are similar to interrupts

Vectored Events (**interrupts** and **exceptions**) cause the processor to jump into an interrupt handler after saving much of the processor's state

Both have an id (vector) = determines which interrupt handler the processor jumps to

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT n instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults.

A **trap** is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.

A **fault** is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the

processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.

An **abort** is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

Booting

System initialization of intel x86 with BIOS Firmware

Is the process of bringing an intel x86 system up from a cold start

Cold start

On the x86 architecture, the reset vector is very near the bottom of the memory map. Thus, actual location moves around based upon the width of the system's Instruction Pointer:

CPU	Address Width	Reset Vector Address	Comment
Intel 8080	16 bits	0xffff0	The first 16 bytes below 64KiB.
Intel 8086	20 bits	0xfffff0	The first 16 bytes below 1MiB.
Intel 80286	24 bits	0xfffffff0	The first 16 bytes below 16MiB.
Intel 80386	32 bits	0xfffffffff0	The first 16 bytes below 4GiB.

Boot strap stage

Reset vector first word must be a valid instruction

Power-on self test (POST)

Performs following checks

- Verifies the CPU registers are functional.
- Verifies the integrity of the BIOS code, typically with a simple checksum.

- Initialize RAM, prior to this step the CPU may only have its registers and CPU cache available.
- Verifies interrupts and DMA are working as expected.
- Initialize the chipset.
- Initialize the system bus.
- Fetches the contents of [CMOS](#) (a small amount of battery backed RAM) restoring the system configuration.

Once the BIOS has been initialized, RAM below 1MiB looks like this:

Start	Stop	Size	Description
0x000000000	0x000003ff	1KiB	Interrupt Vector Table (IVT) in Protected Mode.
0x00000400	0x000004ff	256B	BIOS Data Area (BDA).
0x00000500	0x00007bff	29.75KiB	Conventional memory available to programs.
0x00007c00	0x00007dff	512B	Reserved for Boot Sector.
0x00007e00	0x0007ffff	480.5KiB	Conventional memory available to programs.
0x00080000	0x0009ffff	128KiB	Extended BIOS Data Area (EBDA).
0x000a0000	0x000bffff	128KiB	Video memory.
0x000c0000	0x000c7fff	32KiB	Video card BIOS.
0x000c8000	0x000effff	160KiB	BIOS Extensions.
0x000f0000	0x000fffff	64KiB	Motherboard BIOS

X86 interrupt system

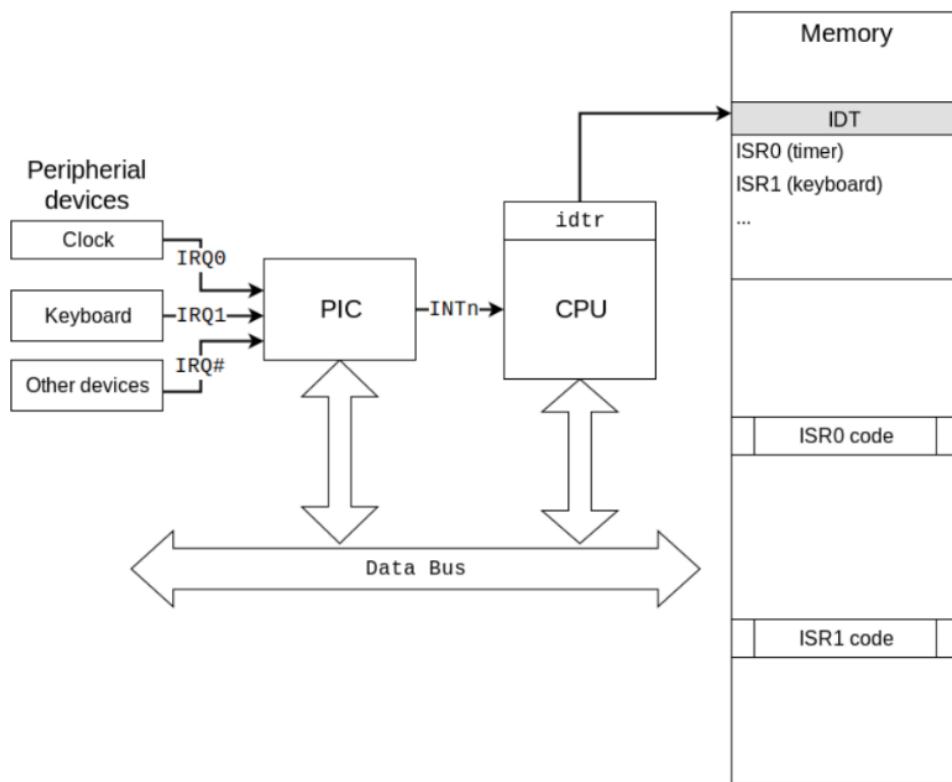
Interrupts

There are 3 sources or types of interrupts:

- Hardware interrupts - comes from hardware devices like keyboard or network card.
- Software interrupts - generated by the software int instruction. Before introducing SYSENTER/SYSEXIT system calls invocation was implemented via the software interrupt int \$0x80.
- Exceptions - generated by CPU itself in response to some error like “divide by zero” or “page fault”.

x86 interrupt system is tripartite in the sense of it involves 3 parts to work conjointly:

- **Programmable Interrupt Controller (PIC)** must be configured to receive interrupt requests (IRQs) from devices and send them to CPU.
- CPU must be configured to receive IRQs from PIC and invoke correct interrupt handler, via gate described in an **Interrupt Descriptor Table (IDT)**.
- Operating system kernel must provide **Interrupt Service Routines (ISRs)** to handle interrupts and be ready to be preempted by an interrupt. It also must configure both PIC and CPU to enable interrupts.



Programmable interrupt controller (PIC)

PIC chaining = 2 PIC's give 15 interrupt lines

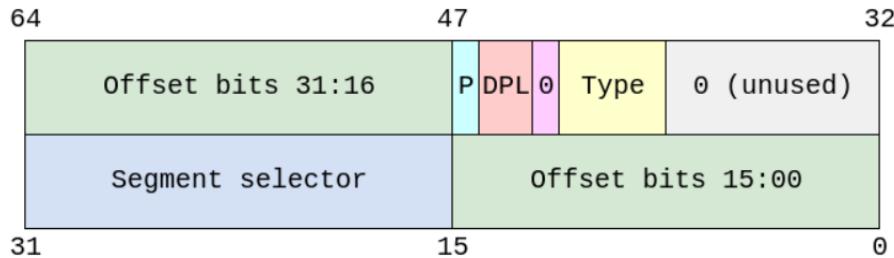
Queues interrupts

Interrupt descriptor table (IDT)

Table that holds descriptors for ISR's or interrupt handlers.

In real mode, there is an IVT (interrupt vector table) which is located by the fixed address 0x0 and contains “interrupt handler pointers” in the form of CS and IP registers values.

IDT can only be used in protected mode



- P - Segment present
- DPL - Descriptor privilege level

Offset is a main part

- A pointer to an ISR within code segment chosen by segment selector

Segment selector

- Index in GDT table, table indicator and request privilege level (RPL)

Type = gate type (task, trap or interrupt).

The main purpose of IDT is to store pointers to ISR that will be automatically invoked by CPU on interrupt receive.

Interrupt service routines (ISR)

If an interrupt occurred in userspace (actually in a different privilege level), CPU does the following¹:

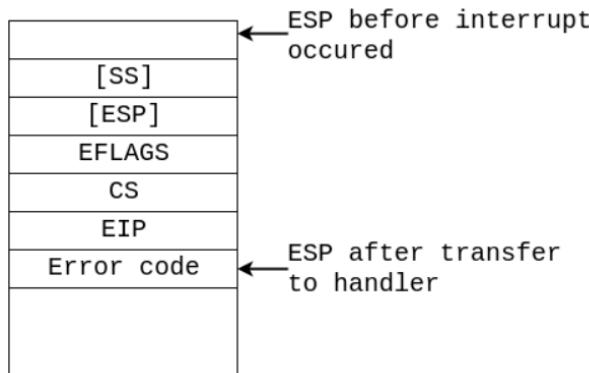
- Temporarily saves (internally) the current contents of the SS, ESP, EFLAGS, CS and EIP registers.
- Loads the segment selector and the stack pointer for the new stack (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
- Pushes the temporarily saved SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure’s stack onto the new stack.
- Pushes an error code on the new stack (if appropriate).

- Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
- If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
- Begins execution of the handler procedure at the new privilege level.

When an interrupt occurs in kernel mode, CPU will:

- Push the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
- Push an error code (if appropriate) on the stack.
- Load the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
- Clear the IF flag in the EFLAGS, if the call is through an interrupt gate.
- Begin execution of the handler procedure.

Interrupt handlers stack on start



Pusha saves general purpose registers onto the stack.

Here is the basic ISR algorithm:

- Save the state of interrupted procedure
- Save previous data segment
- Reload data segment registers with kernel data descriptors
- Acknowledge interrupt by sending EOI to PIC

- Do the work
- Restore data segment
- Restore the state of interrupted procedure
- Enable interrupts
- Exit interrupt handler with iret

Everything together

Now to complete the picture let's see how keyboard press is handled:

- Setup interrupts:
 1. Create IDT table
 2. Set IDT entry #9 [2](#) with interrupt gate pointing to keyboard ISR
 3. Load IDT address with lidt
 4. Send interrupt mask 0xfd (11111101) to PIC1 to unmask (enable) IRQ1
 5. Enable interrupts with sti
- Human hits keyboard button
- Keyboard controller raises interrupt line IRQ1 in PIC1
- PIC checks if this line is not masked (it's not) and send interrupt number 9 to CPU
- CPU checks if interrupts disabled by checking IF in EFLAGS (it's not)
 - (Assume that currently we're executing in kernel mode)
- Push EFLAGS, CS, and EIP on the stack
- Push an error code from PIC (if appropriate) on the stack
- Look into IDT pointed by idtr and fetch segment selector from IDT descriptor 9.
- Check privilege levels and load segment selector and ISR address into the CS:EIP
- Clear IF flag because IDT entries are interrupt gates
- Pass control to ISR

- Receive interrupt in ISR:
 1. Disable interrupt with cli (just in case)
 2. Save interrupted procedure state with pusha
 3. Push current DS value on the stack
 4. Reload DS, ES, FS, GS from kernel data segment
- Acknowledge interrupt by sending EOI (0x20) to master PIC (I/O port 0x20)
- Read keyboard status from keyboard controller (I/O port 0x64)
- If status is 1 then read keycode from keyboard controller (I/O port 0x60)
- Finally, print char via VGA buffer or send it to TTY
- Return from interrupt:
 1. Pop from stack and restore DS
 2. Restore interrupted procedure state with popa
 3. Enable interrupts with sti
 4. iret

Processes

Dive into systems - 13.2

A process represents an instance of a program running in the system, which includes the program's binary executable code, data, and execution **context**. The context tracks the program's execution by maintaining its register values, stack location, and the instruction it is currently executing.

Processes are necessary abstractions in **multiprogramming** systems, which support multiple processes existing in the system at the same time. The process abstraction is used by the OS to keep track of individual instances of programs running in the system, and to manage their use of system resources.

The OS provides each process with a "lone view" abstraction of the system. That is, the OS isolates processes from one another and gives each process the illusion that it's controlling the entire machine. In reality, the OS supports many active processes and manages resource sharing among them. The OS hides the details of sharing and accessing system resources from the user, and the OS protects processes from the actions of other processes running in the system.

Multiprogramming and context switching

when a process running on the CPU needs to access data that are currently on disk, rather than have the CPU sit idle waiting for the data to be read into memory, the OS can give the CPU to another process and let it run while the read operation for the original process is being handled by the disk. By using multiprogramming, the OS can mitigate some of the effects of the memory hierarchy on its program workload by keeping the CPU busy executing some processes while other processes are waiting to access data in the lower levels of the memory hierarchy.

Timesharing = the OS schedules each process to take turns executing on the CPU for short time durations (known as a **time slice** or **quantum**). When a process completes its time slice on the CPU, the OS removes the process from the CPU and lets another run.

On multi-programmed and timeshared systems, processes run concurrently

Context switching

Determine how the OS swaps one process running with another

The OS performs **context switching**, or swapping process state on the CPU, as the primary mechanism behind multiprogramming (and timesharing). There are two main steps to performing a CPU context switch:

- The OS saves the context of the current process running on the CPU, including all of its register values (PC, stack pointers, general-purpose register, condition codes, etc.), its memory state, and some other state (for example the state of system resources it uses, like open files).
- The OS restores the saved context from another process on the CPU and starts the CPU running this other process, continuing its execution from the instruction where it left off.

Process state

The OS maintains information about each process

- A **process id** (PID), which is a unique identifier for a process.
The ps command lists information about processes in the system, including their PID values.
- The address space information for the process.
- The execution state of the process (e.g., CPU register values, stack location).
- The set of resources allocated to the process (e.g., open files).
- The current **process state**, which is a value that determines its eligibility for execution on the CPU.

Sees which processes that are candidates for being scheduled on the CPU

The set of process execution states are:

- **Ready:** The process could run on the CPU but is not currently scheduled (it is a candidate for being context switched on to the CPU). Once a new process is created and initialized by the OS, it enters the ready state (it is ready for the CPU to start executing its first instruction). In a timesharing system, if a process is context switched off the CPU because its time slice is up, it is also placed in the *ready* state (it is ready for the CPU to execute its next instruction, but it used up its time slice and has to wait its turn to get scheduled again on the CPU).
- **Running:** The process is scheduled on the CPU and is actively executing instructions.
- **Blocked:** The process is waiting for some event before it can continue being executed. For example, the process is waiting for some data to be read in from disk. Blocked processes are not candidates for being scheduled on the CPU. After the event on which the process is blocked occurs, the process moves to the *ready* state (it is ready to run again).
- **Exited:** The process has exited but still needs to be completely removed from the system. A process exits due to its completing the execution of its program instructions, or by exiting with an error (e.g., it tries to divide by zero), or by receiving a termination request from another process. An exited process will never run again, but it remains in the system until final clean-up associated with its execution state is complete.

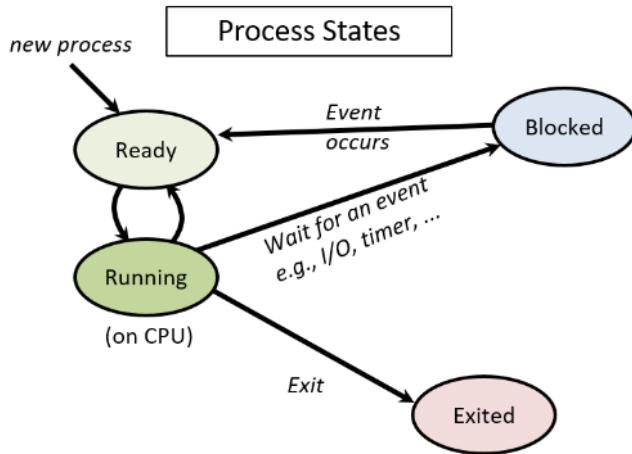


Figure 185. The states of a process during its lifetime

Total **wall time** (or wall-clock time). Wall time is the duration between the start and completion of a process

total **CPU time** (or process time). CPU time measures just the amount of time the process spends in the Running state executing its instructions on the CPU. CPU time does not include the time the process spends in the Blocked or Ready states.

Creating and destroying processes

An OS creates a new process when an existing process makes a system call requesting it to do so. In Unix, the **fork** system call creates a new process. The process calling fork is the **parent** process and the new process it creates is its **child** process

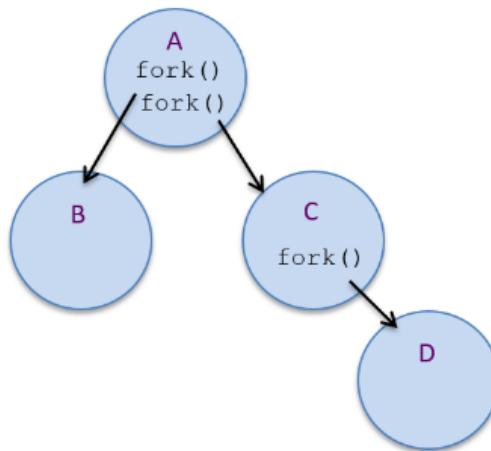


Figure 186. An example process hierarchy created by a parent process (A) calling fork twice to create two child processes (B and C). C's call to fork creates its child process, D. To list the process hierarchy on Linux systems, run `pstree`, or `ps -Aef --forest`.

The fork system call is used to create a process. At the time of the fork, the child inherits its execution state from its parent. The OS creates a *copy* of the calling (parent) process's execution state at the point when the parent calls fork. This execution state

includes the parent's address space contents, CPU register values, and any system resources it has allocated (e.g., open files). The OS also creates a new **process control struct**, an OS data structure for managing the child process, and it assigns the child process a unique PID.

Both parent and child run concurrent.

When the child process is first scheduled by the OS to run on the CPU, it starts executing at the point at which its parent left off — at the return from the fork call

- Child has a copy of parents execution state

The child process always receives a return value of 0, whereas the parent receives the child's PID value (or -1 if fork fails).

```
pid_t pid;  
  
pid = fork(); /* create a new child process */  
  
printf("pid = %d\n", pid); /* both parent and child execute this */
```

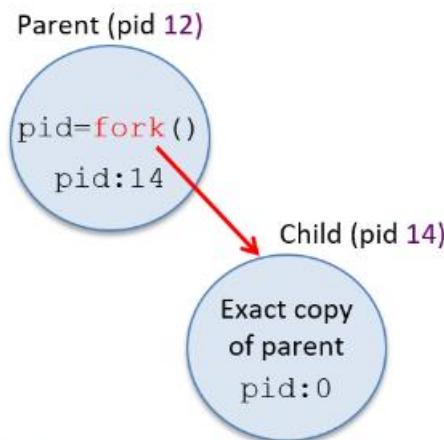


Figure 187. A process (PID 12) calls fork to create a new child process. The new child process gets an exact copy of its parent's address and execution state, but gets its own process identifier (PID 14). fork returns 0 to the child process and the child's PID value (14) to the parent.

```

pid_t pid;

pid = fork(); /* create a new child process */

if (pid == 0) {
    /* only the child process executes this code */
    ...
} else if (pid != -1) {
    /* only the parent process executes this code */
    ...
}

```

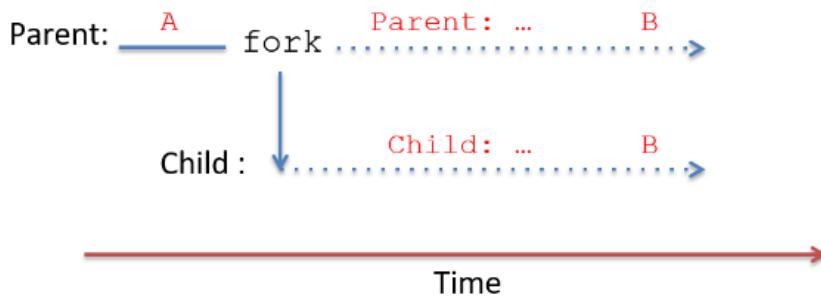


Figure 188. The execution time line of the program. Only the parent process exists before the call to `fork`. After `fork` returns, both run concurrently (shown in the dotted lines).

Exec

While `fork` creates the new child process, it does not cause the child to run `a.out`. To initialize the child process to run a new program, the child process calls one of the **`exec`** system calls.

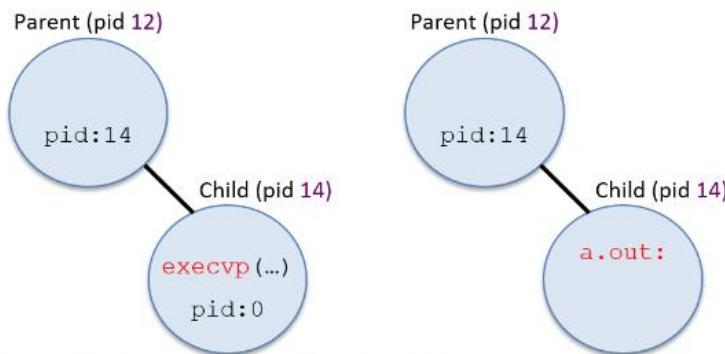


Figure 189. When the child process calls `execvp` (left), the OS replaces its image with `a.out` (right) and initializes the child process to start running the `a.out` program from its beginning.

Exit and wait

To terminate, a process calls the `exit` system call, which triggers the OS to clean up most of the process's state. After running the `exit` code, a process notifies its parent process

that it has exited. The parent is responsible for cleaning up the exited child's remaining state from the system.

Processes can be triggered to exit in several ways. First, a process may complete all of its application code. Returning from its main function leads to a process invoking the exit system call. Second, a process can perform an invalid action, such as dividing by zero or dereferencing a null pointer, that results in its exiting. Finally, a process can receive a **signal** from the OS or another process, telling it to exit (in fact, dividing by zero and NULL pointer dereferences result in the OS sending the process SIGFPE and SIGSEGV signals telling it to exit).

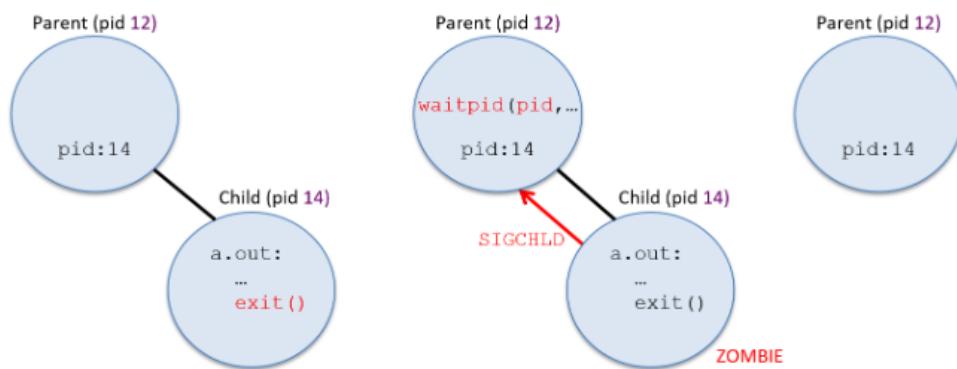


Figure 190. Process exit. Left: The child process calls the exit system call to clean up most of its execution state. Middle: After running exit, the child process becomes a zombie (it is in the Exited state and cannot run again), and its parent process is sent a SIGCHLD signal, notifying it that its child is exited. Right: The parent calls waitpid to reap its zombie child (cleans up the rest of the child's state from the system).

A programmer can also design the parent process code so that it will never block waiting for a child process to exit.

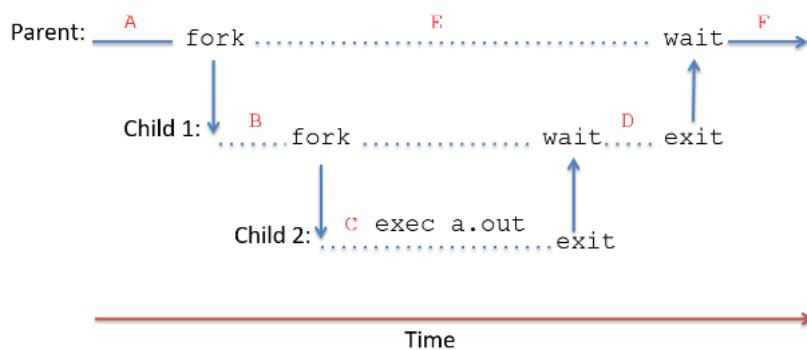


Figure 191. The execution time line for the example program, showing a possible sequence of fork, exec, wait, and exit calls from the three processes. Solid lines represent dependencies in the order of execution between processes, and dotted line concurrent execution points. Parent is the parent process of Child 1, and Child 1 is the parent of Child 2.

Operating systems: three easy pieces - ch 6 and ch 8

Ch 6

limited direct execution. The “direct execution” part of the idea is simple: just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point (i.e., the main() routine or something similar), jumps to it, and starts running the user’s code.

Problems:

- if we just run a program, how can the OS make sure the program doesn’t do anything that we don’t want it to do, while still running it efficiently?
- when we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?

Ch 8

The multi-level feedback queue

- Queues have different priority levels

the first two basic rules for MLFQ:

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn’t).
- Rule 2: If Priority(A) = Priority(B), A & B run in RR.

workload: a mix of interactive jobs that are short-running (and may frequently relinquish the CPU), and some longer-running “CPU-bound” jobs that need a lot of CPU time but where response time isn’t important.

allotment is the amount of time a job can spend at a given priority level before the scheduler reduces its priority

Here is our first attempt at a priority-adjustment algorithm:

- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

- Rule 4a: If a job uses up its allotment while running, its priority is reduced (i.e., it moves down one queue).
- Rule 4b: If a job gives up the CPU (for example, by performing an I/O operation) before the allotment is up, it stays at the same priority level (i.e., its allotment is reset).

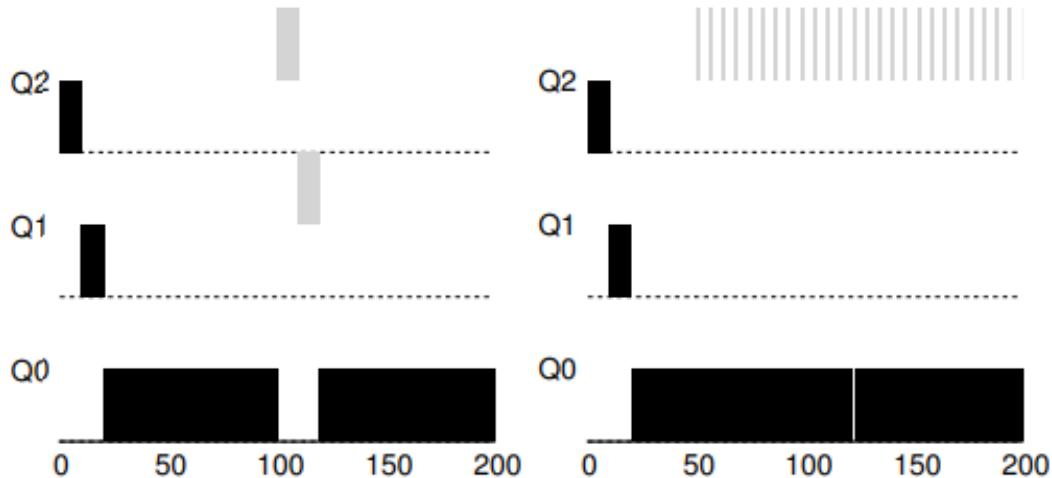


Figure 8.3: Along Came An Interactive Job: Two Examples

Job A = black = long-running job

Job B = grey = short-running job

Problems = starvation (to many jobs consume ALL cpu time), open for attack to relinquish the CPU (remain in the same queue and gain higher percentage of cpu time), program may change behaviour

Starvation and behaviour solution = boost priority of jobs in the system

- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

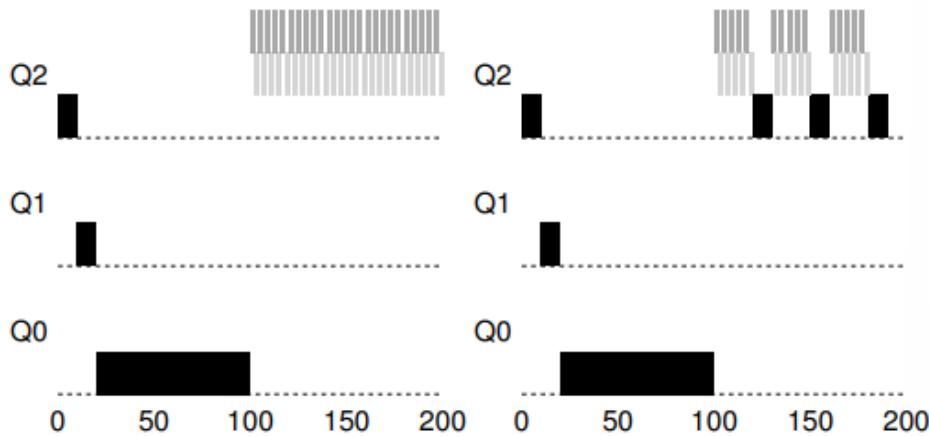


Figure 8.4: Without (Left) and With (Right) Priority Boost

Scheduler solution = perform better accounting of CPU time at each level (once its used its allotment, its demoted)

- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

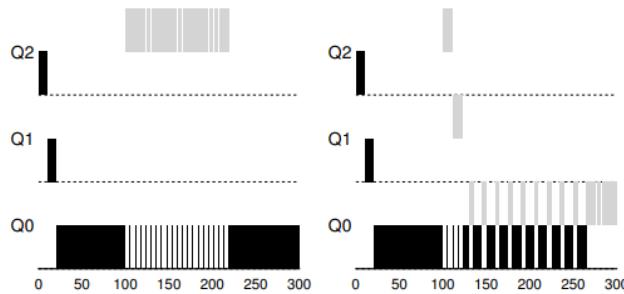


Figure 8.5: Without (Left) and With (Right) Gaming Tolerance

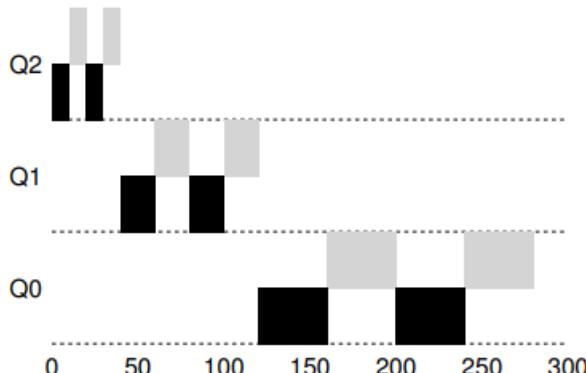


Figure 8.6: Lower Priority, Longer Quanta

Inter-process communication

Dive into systems - 13.4

Interprocess communication

Private virtual address spaces are an important abstraction in multiprogrammed systems, and are one way in which the OS prevents processes from interfering with one another's execution state. However, sometimes a user or programmer may want their application processes to communicate with one another (or to share some of their execution state) as they run.

Operating systems typically implement support for several types of interprocess communication, or ways in which processes can communicate or share their execution state. **Signals** are a very restricted form of interprocess communication by which one process can send a signal to another process to notify it of some event. Processes can also communicate using **message passing**, in which the OS implements an abstraction of a message communication channel that is used by a process to exchange messages with another process. Finally, the OS may support interprocess communication through **shared memory** that allows a process to share all or part of its virtual address space with other processes. Processes with shared memory can read or write to addresses in shared space to communicate with one another.

Signals

A software interrupt via the OS.

Signals are similar to hardware interrupts and traps but are different from both. Whereas a trap is a synchronous software interrupt that occurs when a process explicitly invokes a system call, signals are asynchronous — a process may be interrupted by the receipt of a signal at any point in its execution. Signals also differ from asynchronous hardware interrupts in that they are triggered by software rather than hardware devices.

Amount of different signals is a set size

- Linux offers 32

Table 127. Example Signals Used for Interprocess Communication

Signal Name	Description
SIGSEGV	Segmentation fault (e.g., dereferencing a null pointer)
SIGINT	Interrupt process (e.g., Ctrl-C in terminal window to kill process)
SIGCHLD	Child process has exited (e.g., a child is now a zombie after running <code>exit</code>)
SIGALRM	Notify a process a timer goes off (e.g., <code>alarm(2)</code> every 2 secs)
SIGKILL	Terminate a process (e.g., <code>pkill -9 a.out</code>)
SIGBUS	Bus error occurred (e.g., a misaligned memory address to access an <code>int</code> value)
SIGSTOP	Suspend a process, move to Blocked state (e.g., Ctrl-Z)
SIGCONT	Continue a blocked process (move it to the Ready state; e.g., <code>bg</code> or <code>fg</code>)

When a process receives a signal, one of several default actions can occur:

- the process can terminate
- the signal can be ignored
- the process can be blocked
- the process can be unblocked

Message passing

message passing — by sending and receiving messages to one another. Message passing allows programs to exchange arbitrary data rather than just a small set of predefined messages like those supported by signals.

The message passing interprocess communication model consists of three parts:

- Processes allocate some type of message channel from the OS. Example message channel types include *pipes* for one-way communication, and *sockets* for two-way communication. There may be additional connection setup steps that processes need to take to configure the message channel.
- Processes use the message channel to send and receive messages to one another.
- Processes close their end of the message channel when they are done using it.

Pipe = one way communication channel (2 processes on the same machine)

- Bash pipe example: \$ cat foo.c | grep factorial

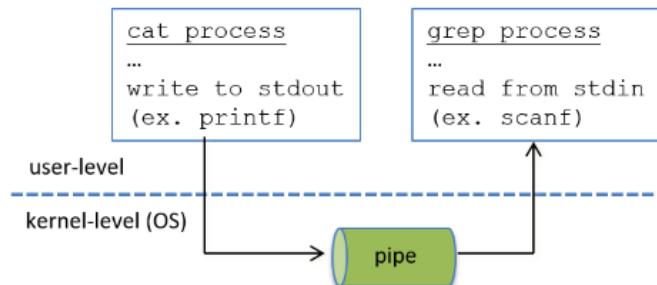


Figure 203. Pipes are unidirectional communication channels for processes on the same system. In this example, the cat process sends the grep process information by writing to the write end of the pipe. The grep process receives this information by reading from the read end of the pipe.

Socket = two way communication

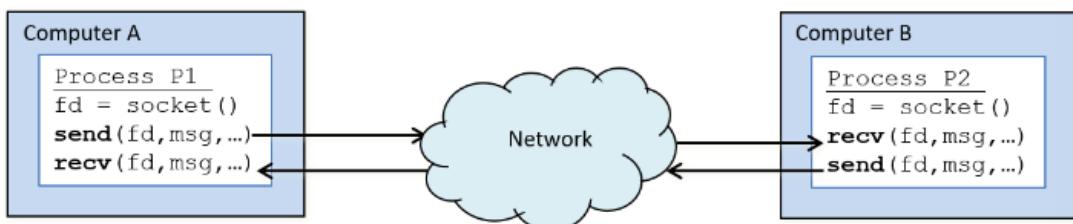


Figure 204. Sockets are bidirectional communication channels that can be used by communicating processes on different machines connected by a network.

Shared memory

when two processes are running on the same machine, they can take advantage of shared system resources to communicate more efficiently than by using message passing.

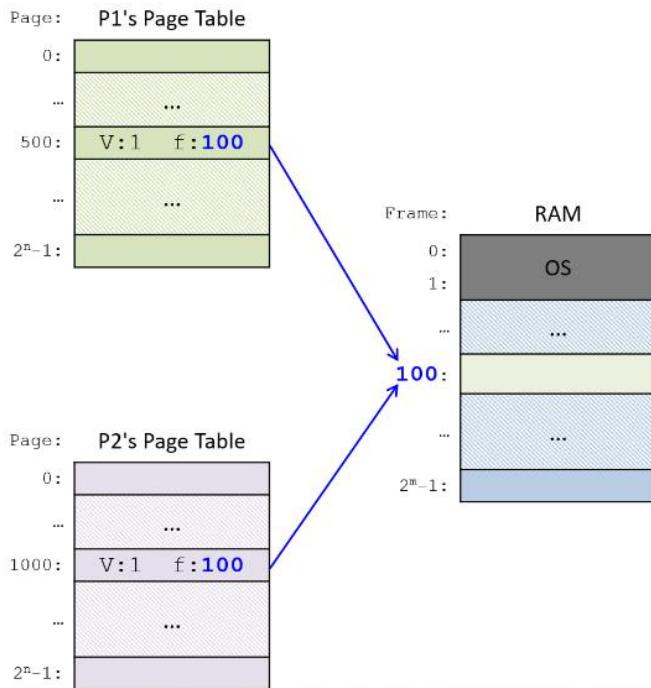


Figure 205. The OS can support sharing pages of virtual address space by setting entries in the page tables of sharing processes to the same physical frame number (e.g., frame 100). Note that processes do not need to use the same virtual address to refer to the shared page of physical memory.

Slides

Recap

recall: “operating system manages processes & hardware”

...how? we haven’t *really* seen the OS **do anything** yet.

kernel is a collection of **code** and **data structures** (residing in RAM) that are used to manage processes and hardware. but a kernel is not a process/thread. so,

how does the OS do anything? what makes this code execute?

interrupts.

Topics

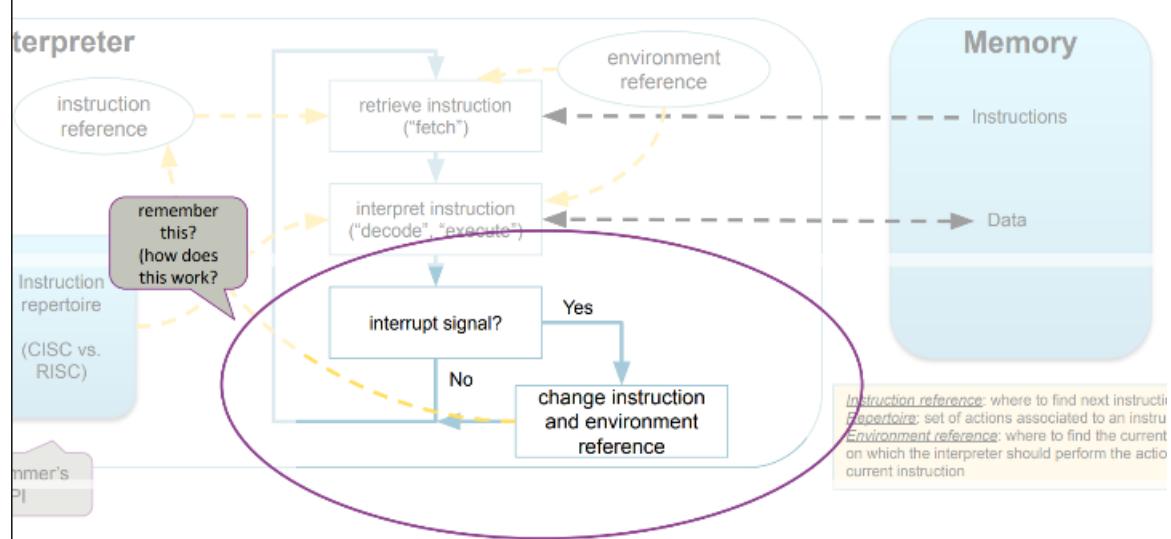
- Interrupts
 - X86 Interrupt System (w/ example `write` syscall in xv6)
 - Interrupts (async), Exceptions (sync; Traps, Faults, Aborts)
- Booting
- Processes
 - API (`fork`, `exit`, `wait`, Inter-Process Communication)
 - Kernel State, Context Switching
 - Scheduling

Interrupts

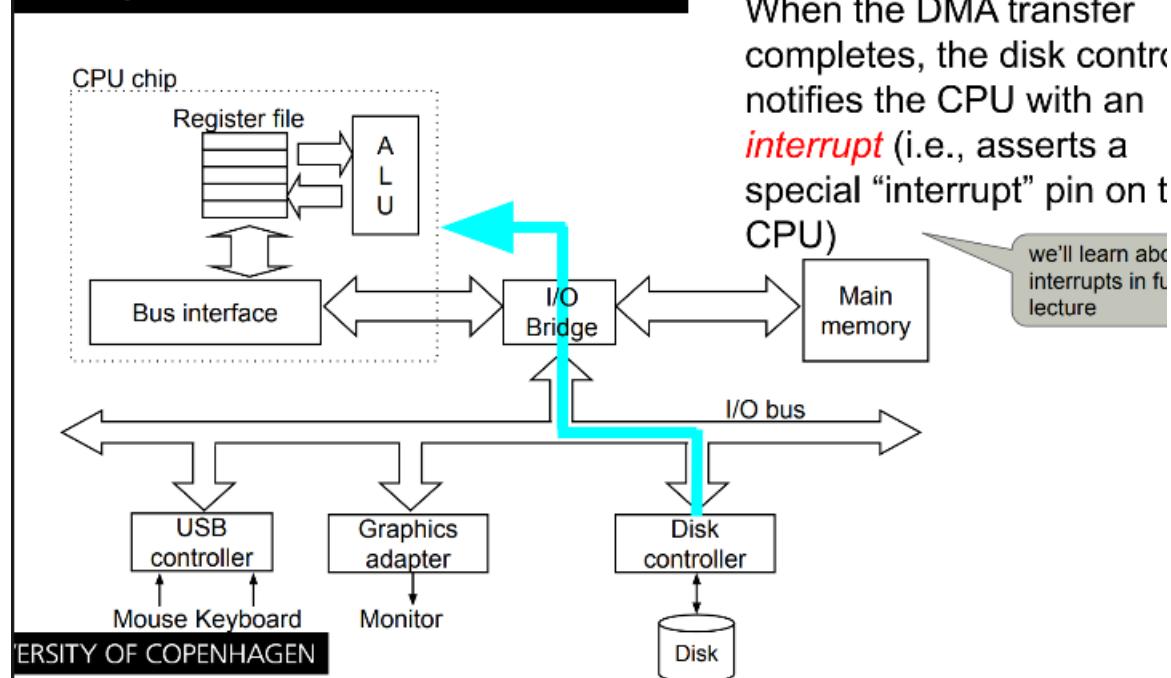
What makes the OS spring into action

Interpreter abstraction

Source: Saltzer and Kaa

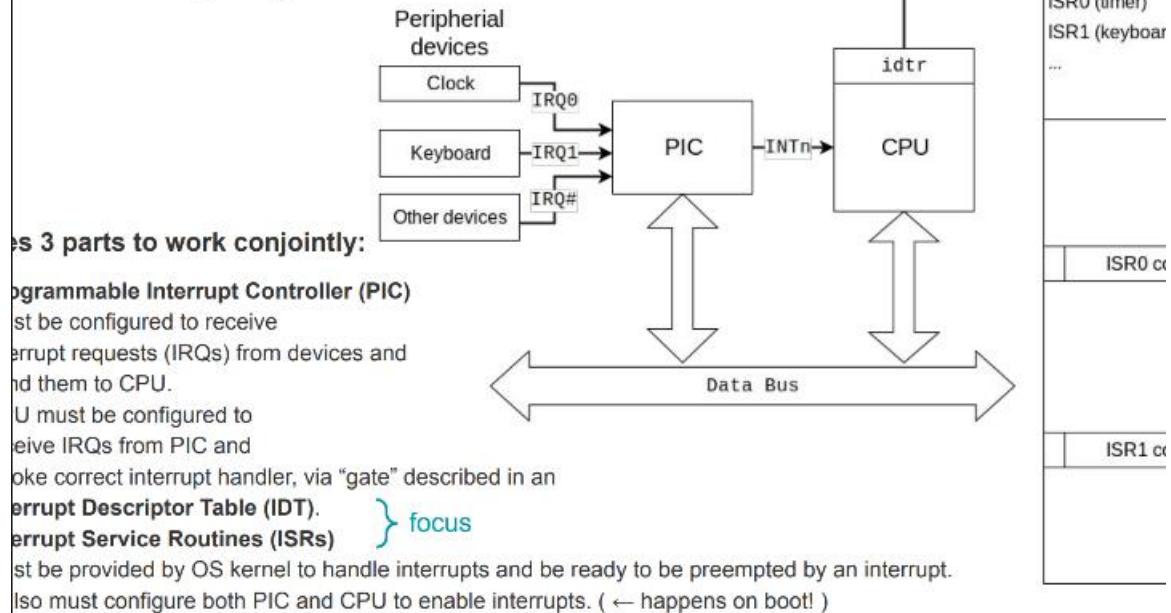


Reading a Disk Sector (3)



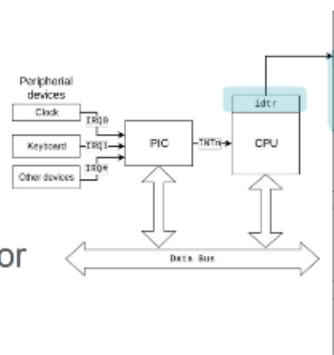
When the DMA transfer completes, the disk controller notifies the CPU with an **interrupt** (i.e., asserts a special "interrupt" pin on the CPU)

86 interrupt system



interrupt descriptor table (IDT)

Role in memory (pointed by idtr).
Created by OS (lidt instruction.). holds "gate descriptors" for interrupt service routines (ISRs), aka. *interrupt handlers*.



"Gate descriptors":

Address to interrupt handler (ISRs)

Flags, protection levels

Execution semantics:

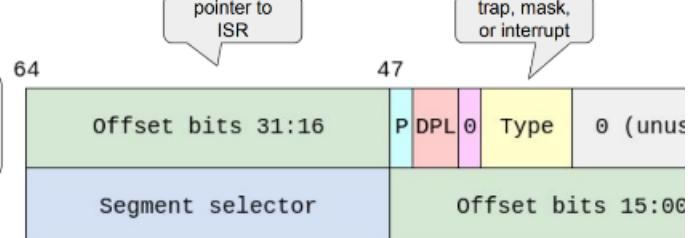
Processor automatically

invokes the handler

registered for the interrupt.

e.g. in Fig, for IRQ1, it's ISR1)

assuming
interrupts
have been
enabled
(sti)



- P - Segment present
- DPL - Descriptor privilege level

giving an interrupt

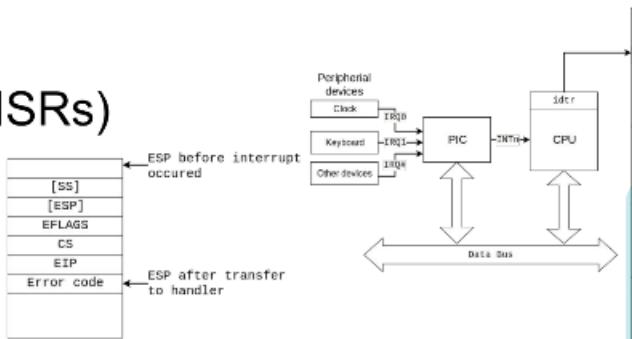
interrupt service routines (ISRs)

-space:

Temporarily **saves** (internally) the current contents of the SS, ESP, EFLAGS, CS and EIP registers.

Loads the segment selector and the stack pointer for the **new stack** (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.

Pushes the temporarily saved registers SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack **onto the new stack**.
Pushes an error code on the new stack (if appropriate).
Loads the segment selector for the new code segment and the **new instruction pointer** (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
Begins execution of the handler procedure at the new privilege level.



kernel-space: (main diff: doesn't switch stacks)

1. **Push the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.**
2. Push an error code (if appropriate) on the stack.
3. **Load** the segment selector for the new code segment and the **new instruction pointer** (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. Clear the IF flag in the EFLAGS, if the call is through an interrupt gate.
5. **Begin execution of the handler procedure**

what should the handler do? (next slide)

Handling an interrupt

interrupt service routines (ISRs)

Here is the basic ISR algorithm:

```
Save the state of interrupted procedure (pusha) ←  
Save previous data segment  
Reload data segment registers with kernel data descriptors  
Acknowledge interrupt by sending EOI to PIC  
[ Do the work ]  
Restore data segment  
Restore the state of interrupted procedure (popa)  
Enable interrupts (sti)  
Exit interrupt handler (with iret)
```



example: keyboard press

Setup interrupts:

- Create IDT table
- Set IDT entry #9 with interrupt gate pointing to keyboard ISR
- Load IDT address with lidt
- Send interrupt mask 0xd (11111101) to PIC1 to unmask (enable) IRQ1
- Enable interrupt with sti

Human hits keyboard button

Keyboard controller raises interrupt line IRQ1 in PIC1

PIC checks if this line is not masked (it's not) and send interrupt number 9 to CPU

CPU checks if interrupts disabled by checking IF in EFLAGS (it's not)

(Assume that currently we're executing in kernel mode)

Push EFLAGS, CS, and EIP on the stack

Push an error code from PIC (if appropriate) on the stack

Look into IDT pointed by idtr and fetch segment selector from IDT descriptor 9.

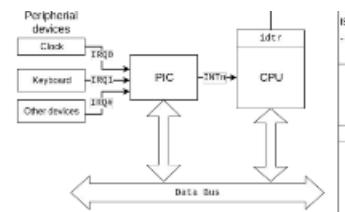
Check privilege levels and load segment selector and ISR address into the CS:EIP

Clear IF flag because IDT entries are interrupt gates

Pass control to ISR

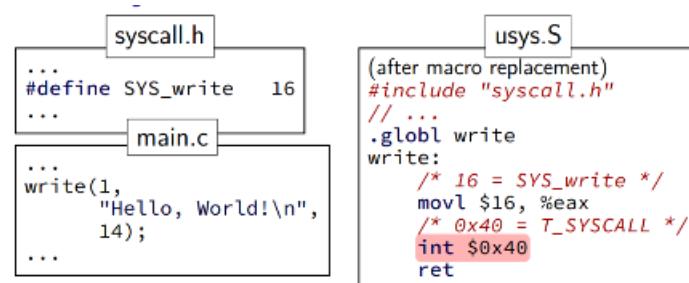
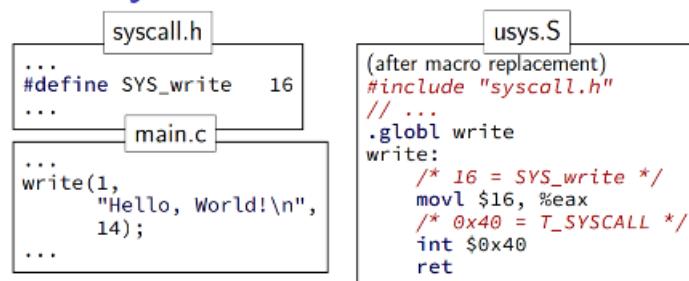
the handler

keyboard's IRQ pre-set to 9 by BIOS
(is reconfigurable)

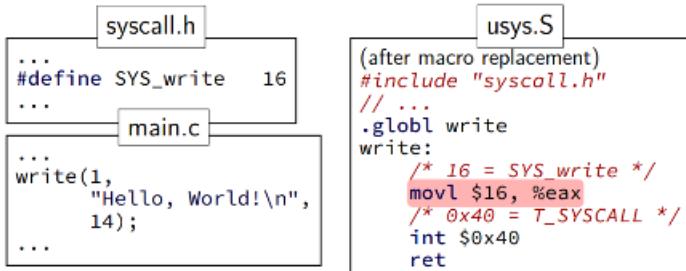


- Receive interrupt in ISR:
 - Disable interrupt with cli (just in case)
 - Save interrupted procedure state with pusha
 - Push current DS value on the stack
 - Reload DS, ES, FS, GS from kernel data segm
- Acknowledge interrupt by sending EOI (0x20) to master (I/O port 0x20)
- Read keyboard status from keyboard controller (I/O port 0x64)
- If status is 1 then read keycode from keyboard controller (I/O port 0x60)
- Finally, print char via VGA buffer or send it to TTY
- Return from interrupt:
 - Pop from stack and restore DS
 - Restore interrupted procedure state with popa
 - Enable interrupts with sti
 - iret

write syscall in xv6: user mode



interrupt — trigger an exception similar to a keypress
parameter (0x40 in this case) — type of exception



xv6 syscall calling convention:
 eax = syscall number
 otherwise: same as 32-bit x86 calling convention
 (arguments on stack)

write syscall in xv6: interrupt table setup

```

trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
  
```

```

trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
  
```

lidt —
 function (in x86.h) wrapping lidt instruction
 sets the *interrupt descriptor table*
 table of *handler functions* for each interrupt type

```

trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
  
```

(from mmu.h):
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap gate, 0 for an interrupt gate.
// - interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
// the privilege level required for software to invoke
// this interrupt/trap gate explicitly using an int instruction.

```

#define SETGATE(gate, istrap, sel, off, d) \
  
```

```

trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
  
```

set the T_SYSCALL (= 0x40) interrupt to
 be callable from user mode via **int** instruction
 (otherwise: triggers fault like privileged instruction)

```
trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

set it to use the kernel "code segment"
 meaning: run in kernel mode
 (yes, code segments specifies more than that — nothing we care about)

```
trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

1: do not disable interrupts during syscalls
 e.g. keypress handling can interrupt slow syscall
 con: makes writing system calls safely more complicated
 pro: slow system calls don't stop timers, keypresses, etc. from working

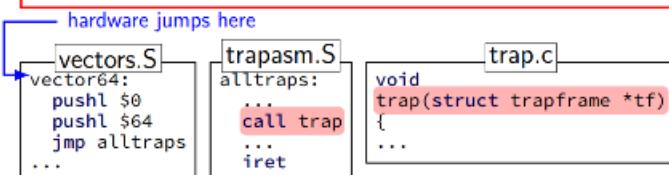
xv6 choice: interrupts are disabled during non-syscall exception handling
 (e.g. don't worry about keypress being handled while timer being handled)

```
trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

vectors[T_SYSCALL] — OS function for processor to run
 set to pointer to assembly function vector64

```
trap.c (run on boot)
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

vectors[T_SYSCALL] — OS function for processor to run
 set to pointer to assembly function vector64



write syscall in xv6: the trap function

```
trap.c
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    ...
}
```

```
trap.c
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    ...
}
```

struct trapframe — set by assembly
interrupt type, application registers, ...
example: `tf->eax` = old value of eax

```
trap.c
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    ...
}
```

myproc() — pseudo-global variable
represents currently running process
much more on this later in semester

```
trap.c
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    ...
}
```

myproc() — pseudo-global variable
represents currently running process
much more on this later in semester

```
trap.c
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    ...
}
```

syscall() — actual implementations
uses `myproc()->tf` to determine
what operation to do for program

write syscall in xv6: the syscall function

```
syscall.c
static int (*syscalls[])(void) = {
...
[SYS_write] sys_write,
...
};

...
void
syscall(void)
{
...
    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    } else {
...
}
```

```
syscall.c
static int (*syscalls[])(void) = {
...
[SYS_write] sys_write,
...
};

...
array of functions — one for syscall
'[number] value': syscalls[number] = value
void
syscall(void)
{
...
    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    } else {
...
}
```

```
syscall.c
static int (*syscalls[])(void) = {
...
[SYS_write] sys_write,
...
};

...
(if system call number in range)
call sys_...function from table
store result in user's eax register
void
syscall(void)
{
...
    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    } else {
...
}
```

```
syscall.c
static int (*syscalls[])(void) = {
...
[SYS_write] sys_write,
...
};

...
(result assigned to eax
(assembly code this returns to
copies tf->eax into %eax))
void
syscall(void)
{
...
    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        curproc->tf->eax = syscalls[num]();
    } else {
...
}
```

write syscall in xv6: sys_write

```
int
sys_write(void)
{
    struct file *f;
    int n;
    char *p;

    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
        return -1;
    return filewrite(f, p, n);
}
```

```
int
sys_write(void)
{
    struct file *f;
    int n;
    char *p;

    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
        return -1;
    return filewrite(f, p, n);
}
```

utility functions that read arguments from user's stack
returns -1 on error (e.g. stack pointer invalid)
(more on this later)

(note: 32-bit x86 calling convention puts all args on stack)

```
int
sys_write(void)
{
    struct file *f;
    int n;
    char *p;

    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
        return -1;
    return filewrite(f, p, n);
}
```

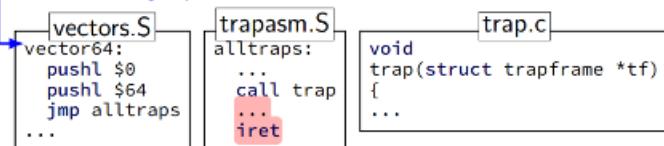
actual internal function that implements writing to a file
(the terminal counts as a file)

write syscall in xv6: interrupt table setup

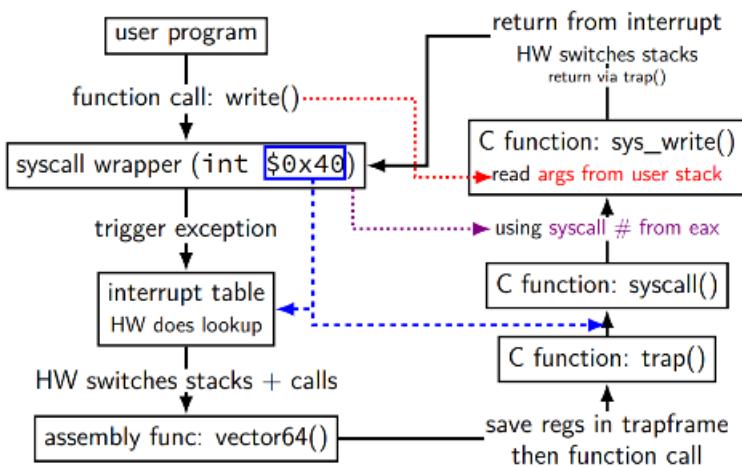
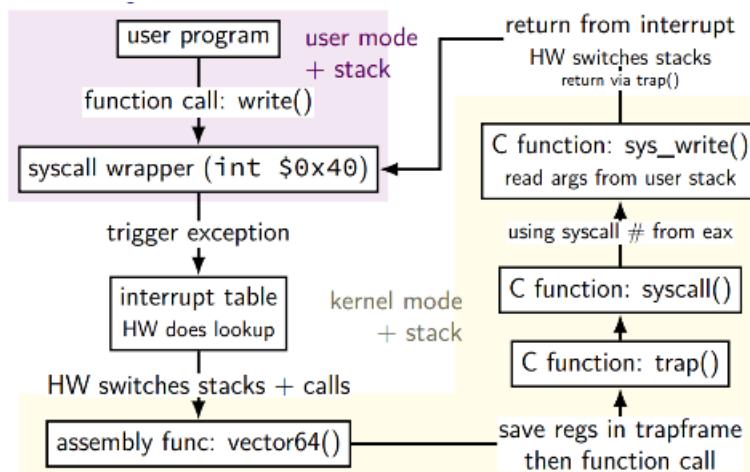
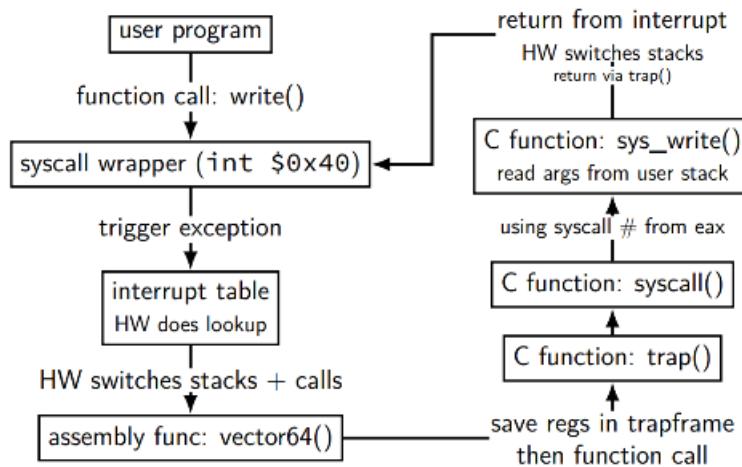
```
... lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

trap returns to alltraps
alltraps restores registers from tf, then returns to user-mode

hardware jumps here



write syscall in xv6

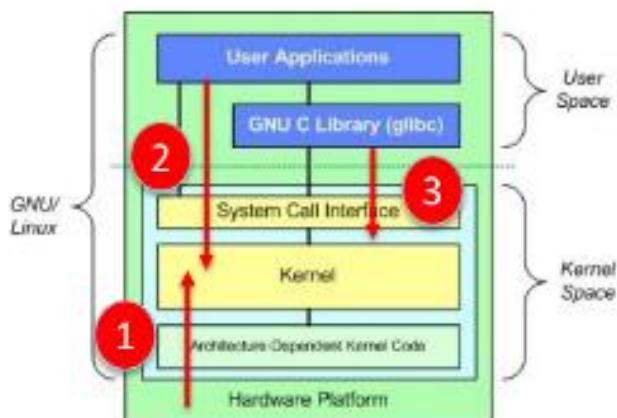


Exceptional Control Flow

Interruption event:

- 1 A device needs attention
- 2 The user program did something illegal
- 3 The user program asks the OS kernel for a service through a system call

In these cases, the flow of control is transferred from the user program to the OS kernel.



Asynchronous Events

Managed in hardware

- Caused by events external to the processor
 - Indicated by setting the processor's interrupt pin
 - **Handler returns to “next” instruction**
- Examples:
 - I/O interrupts
 - hitting Ctrl-C at the keyboard
 - arrival of a packet from a network
 - arrival of data from a disk
 - Hard reset interrupt
 - hitting the reset button
 - Soft reset interrupt
 - hitting Ctrl-Alt-Delete on a PC

• Advanced Programmable Interrupt Controller (APIC) is a more modern interrupt controller than the earlier 82C59 (see below). It supports multiprocessor/multicore interrupt management by allowing interrupts to be directed to a specific processor. The I/O APIC in the PCH can support up to 24 interrupt vectors and works in conjunction with I/O APICs in other devices to help eliminate the need for share interrupts among multiple devices.

PCH: Platform Controller Hub

Asynchronous Events

Hardware hands it out to software

When an interruption event occurs, hardware saves the minimum processor state required to enable software to resolve the event and continue. The state saved by hardware is held in a set of interruption resources, and together with the interruption vector gives software enough information to either resolve the cause of the interruption, or surface the event to a higher level of the operating system. Software has complete control over the structure of the information communicated, and the conventions between the low-level handlers and the high-level code. Such a scheme allows software rather than hardware to dictate how to best optimize performance for each of the interruptions in its environment. The same basic mechanisms are used in all interruptions to support efficient IA-64 low-level fault handlers for events such as a TLB fault, speculation fault, or a key miss fault.

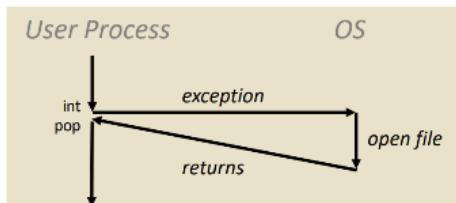
Synchronous Events

- Caused by events that occur as a result of executing an instruction:
 - **Traps**
 - Intentional
 - Examples: **system calls**, breakpoint traps, special instructions
 - Returns control to “next” instruction
 - **Faults**
 - Unintentional but possibly recoverable
 - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
 - Either re-executes faulting (“current”) instruction or aborts
 - **Aborts**
 - unintentional and unrecoverable
 - Examples: parity error, machine check
 - Aborts current program

Trap Example: System call

User calls: open(filename, options)
Function open executes **system call instruction int**

```
0804d070 <__libc_open>:  
  . . .  
 804d082: cd 80          int    $0x80  
 804d084: 5b             pop    %ebx  
  . . .
```



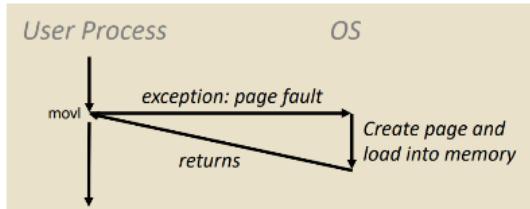
OS must find or create file, get it ready for reading or writing
Returns integer file descriptor

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d  movl $0xd,0x8049d10
```

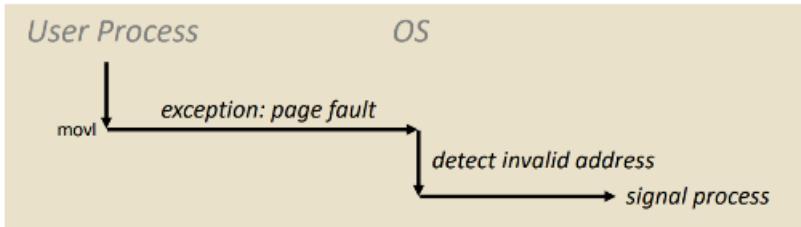


- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Abort Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d  movl $0xd,0x804e360
```



Page handler detects invalid address
Sends SIGSEGV signal to user process
User process exits with "segmentation fault"

Taxonomy

interrupts, exceptions, traps, events, often conflated.

two kinds of vectored events:

- interrupt : asynchronous events
 - consequence of HW, e.g. keypress, timer, disk, ...
 - handed off to SW (**interrupt handlers**) for processing.
- exception : synchronous events
 - consequence of SW executing an instruction.
 - traps, faults, aborts are exceptions.
 - int, syscall are traps, handled by **interrupt handlers**.

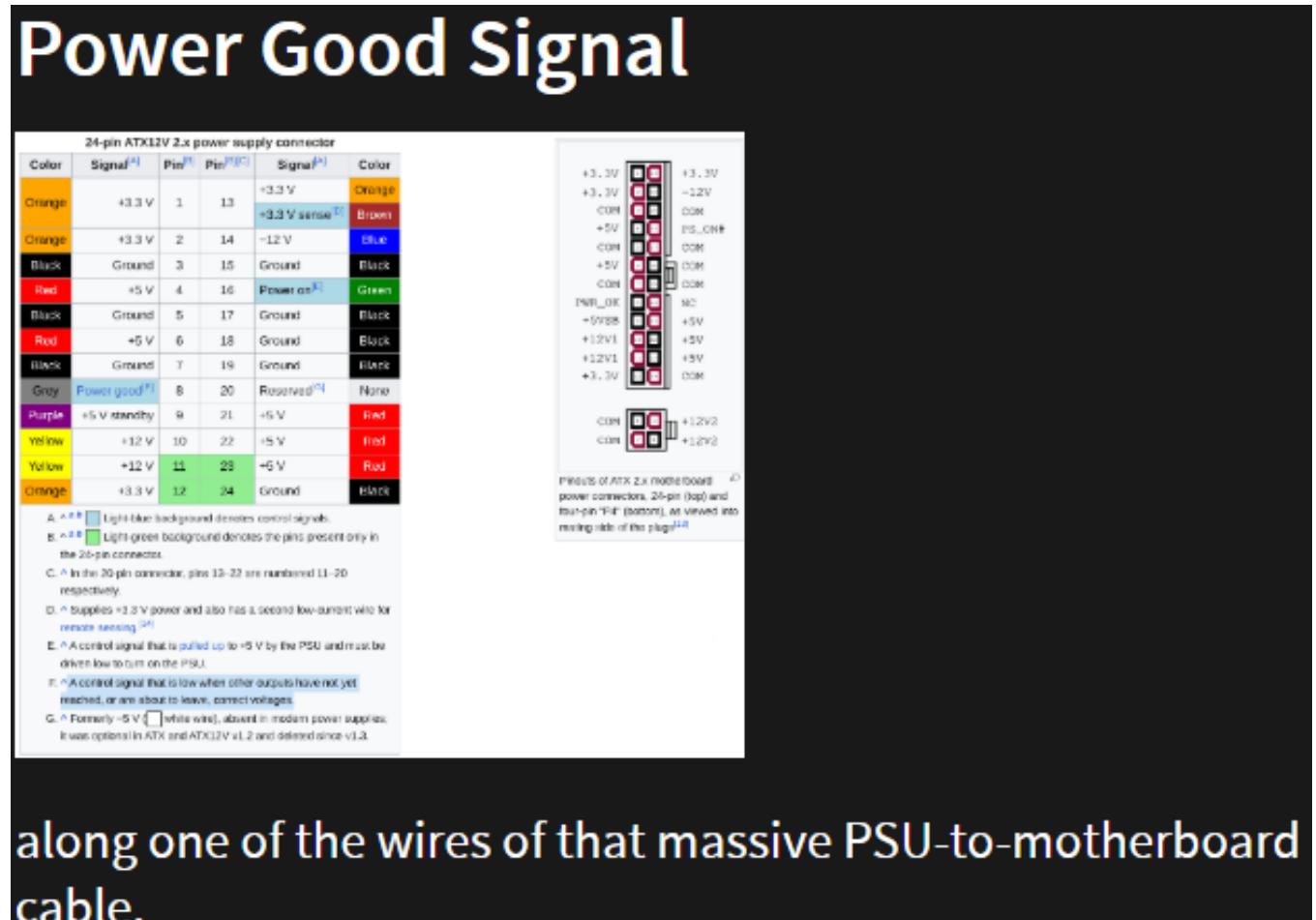
Booting

What puts the OS in place

Signal received by PSU

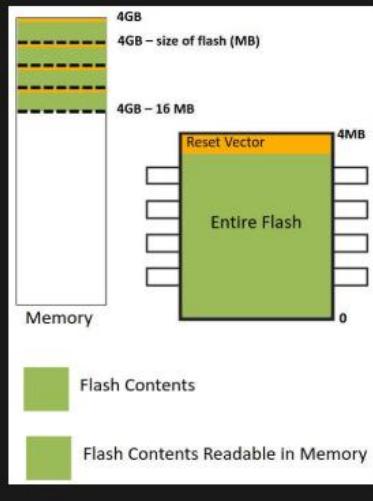
Power supply unit (PSU) performs self-test and checks that its output has stabilized.

Once it has, sends power good signal to motherboard.

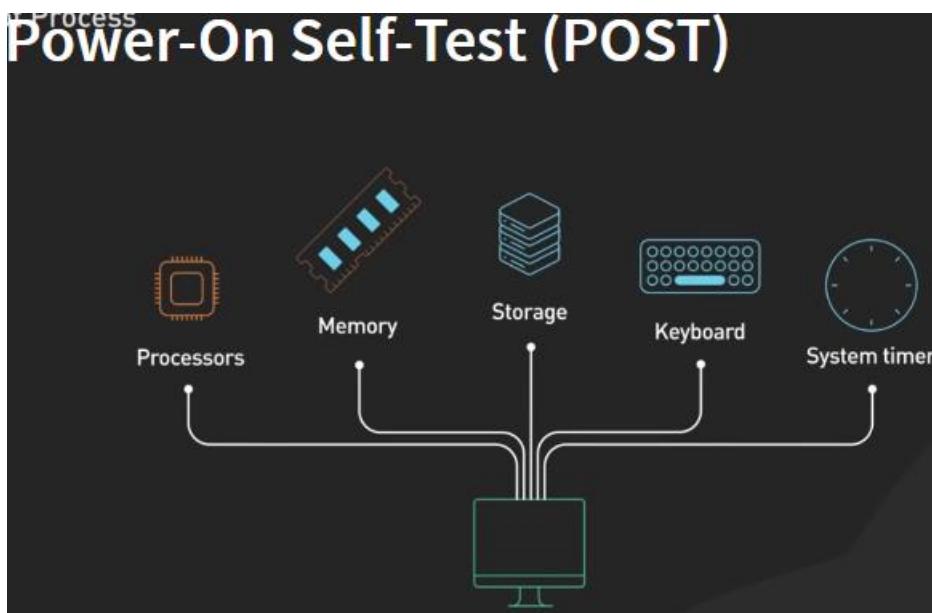


Execute System Firmware from ROM

read-only memory (ROM) on motherboard contains (system) firmware (e.g. BIOS, UEFI) its contents is memory-mapped. first instruction CPU executes, is one in ROM's last word: the reset vector: a `jmp` into firmware. (RAM is not configured yet, hence we execute from (slow) ROM) sets up interrupt handlers (for keyboard, system timer, etc.).



Power-On Self-Test (POST)



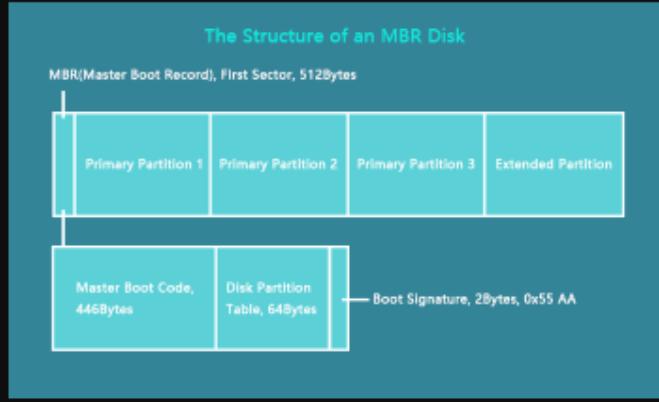
Power-On Self-Test (POST)

check all important devices are functioning.



if not, booting halts, and:

Boot Device Houses A Bootloader



Bootloader in MBR “Hello, World!”

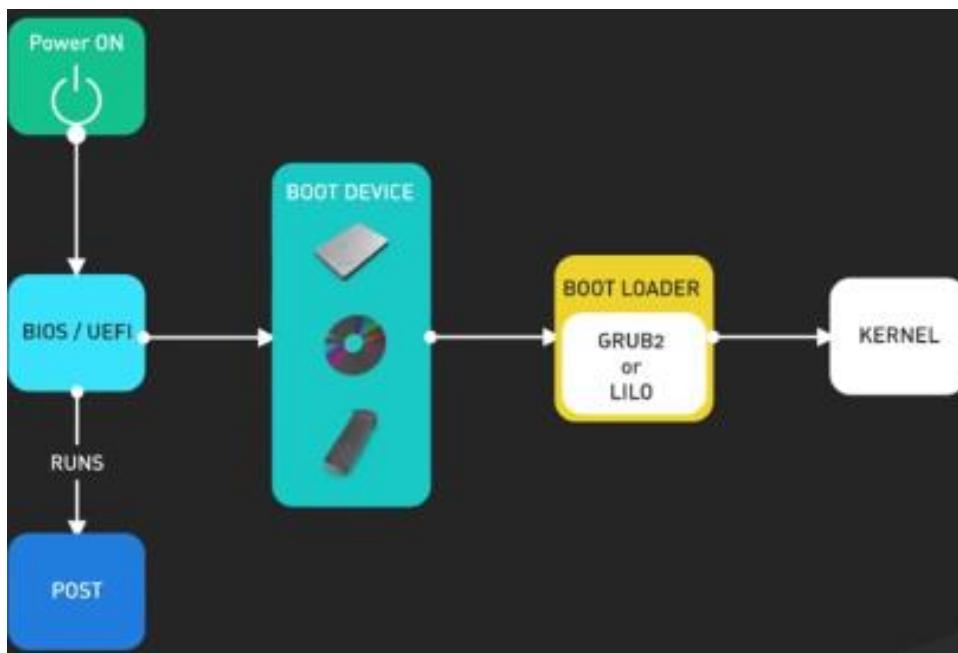
```
1 ;
2 ; Note: this example is written in Intel Assembly syntax
3 ;
4 [BITS 16]
5
6 boot:
7     mov al, '!'
8     mov ah, 0x0e
9     mov bh, 0x00
10    mov bl, 0x07
11
12    int 0x10
13    jmp $ 
14
15 times 510-($-$) db 0
16
17 db 0x55
18 db 0xaa
```

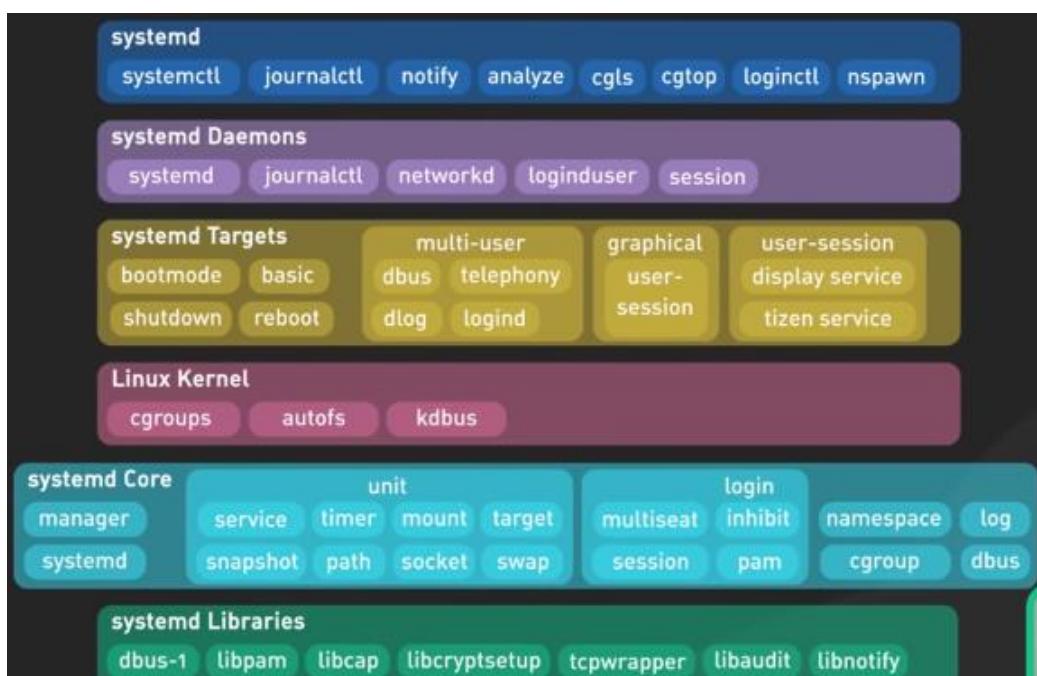
(see the two magic bytes at the end; cf. to MBR slide)

(that interrupt is handled by the system firmware)

The key jobs for the boot loader are:

1. Locate the operating system kernel on the disk
2. Load the kernel into the computer's memory
3. Start running the kernel code





Processes

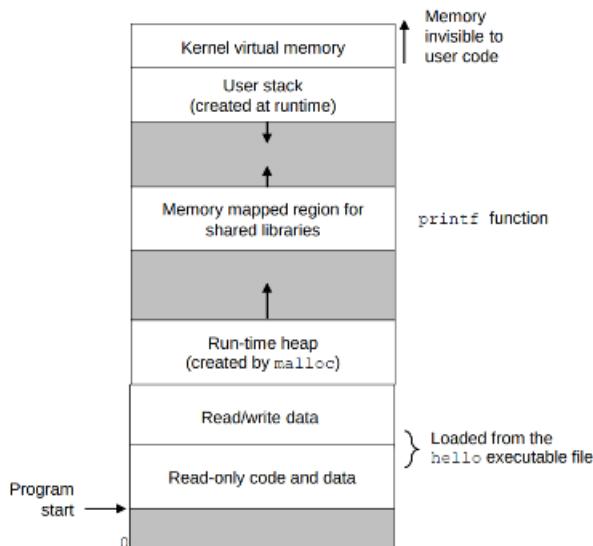
Tasks being managed by the OS

26 Processes

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized hierarchically. Each process has a *parent process* which explicitly arranged to create it. The processes created by a given parent are called its *child processes*. A child inherits many of its attributes from the parent process.

Virtual Memory

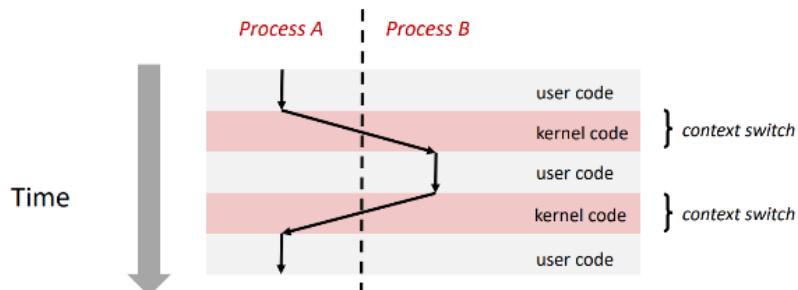


- Process creation/termination/control defined in standard C library (`unistd.h`)
- Transferring the thread of control from one process to another is called *context switching*. It is managed by the OS kernel.

Context Switching

Control flow passes from one process to another via a *context switch*

Important: the kernel is not a separate process, but rather runs as part of some user process

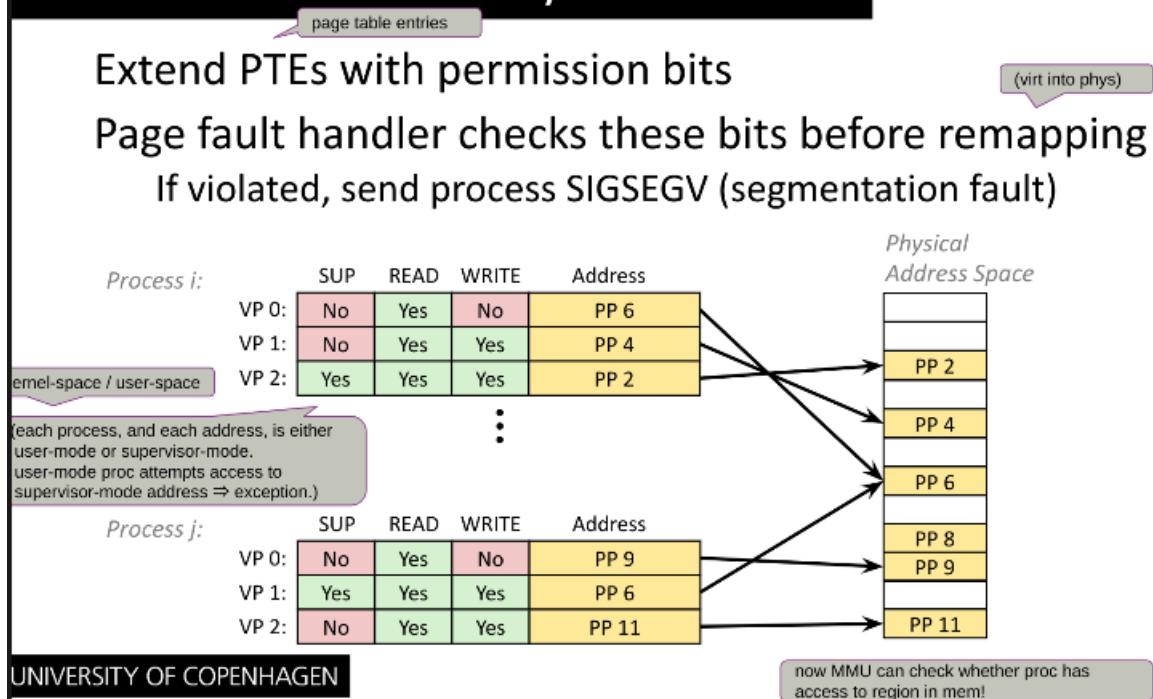


Processor modes:

- Supervisor vs. user modes: "Supervisor mode may provide access to different peripherals, to memory management hardware or to different memory address spaces. It is also capable of interrupt enabling, disabling, returning and loading of processor status."
- Supervisor mode entered on system call

When a context switch is made the scheduler marks the task as interruptible, saves the process's `task_struct` and replaces the current tasks pointer with a pointer to the new process's `task_struct`, marked as running, restoring its memory access and register context.

VM as a Tool for Memory Protection



The question is broad, but I can give some general information and links referencing the [Instruction Set Architecture \(ISA\)](#) which describes all the instructions.

You can't [MOV](#) or [POP](#) a value into CS (on 286+¹) so you are prevented from modifying CS that way. For example for [POP](#) there is a rule:

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

And the rule for [MOV](#) is similar:

The MOV instruction cannot be used to load the CS register.

You can modify CS indirectly through [syscall](#), [sysenter](#), [FAR jmp](#) (via call gate), [FAR call](#) (via call gate), [iret](#), [retf](#) (FAR return) or [int](#). You can review the [ISA](#) for each instruction and what privilege level checks are applied. You can't arbitrarily change CPL if you don't have the privilege and access rights to do so.

Under most circumstances if you have the privilege to affect a change to CPL it will be changed. If you don't have the required privileges you get an exception (privilege level checks usually involve [RPL](#), [CPL](#), [DPL](#)). If using [conforming code segments](#) (a different topic) you can request to execute code using a code segment with a higher privileged DPL but the CPL will remain unchanged. It is a case where the CPL and DPL (Descriptor privilege level) of CS can be different while code is executing.

Footnotes

¹You were allowed to modify CS via [POP](#) and [MOV](#) on 8088/8086 processors.

Share Improve this answer Follow

edited Sep 2, 2019 at 19:00

answered Sep 2, 2019 at 17:27

 Michael Petrich
47.8k • 9 • 116 • 212

Process State (kernel)

Process context:

- 8KB / process in kernel space to store process descriptor `task_struct` ([/linux/include/linux/sched.h](#)).

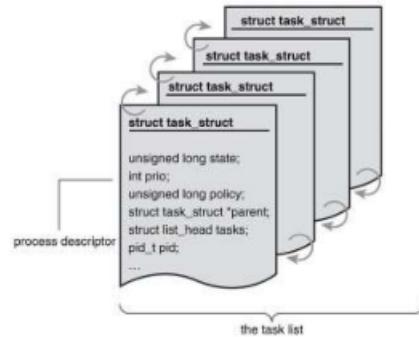
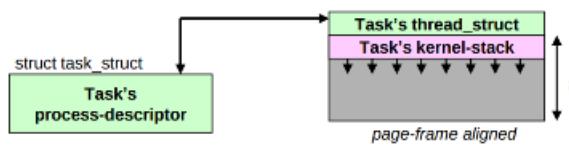
State:

```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE 4
#define TASK_STOPPED 8
```

Process ID

+ virtual memory info, file system info, open files, signal handlers, ...

- The thread of execution `thread_struct` ([/linux/arch/x86/include/asm/processor.h](#))
PC, registers, Fault info,



Process Management (libc)

- Spawning process: `fork()`
- Terminating process: `exit()`
- Waiting for process: `wait()`
- Executing a program within a process: `execve()`

On cos: /usr/include/unistd.h

fork: Creating New Processes

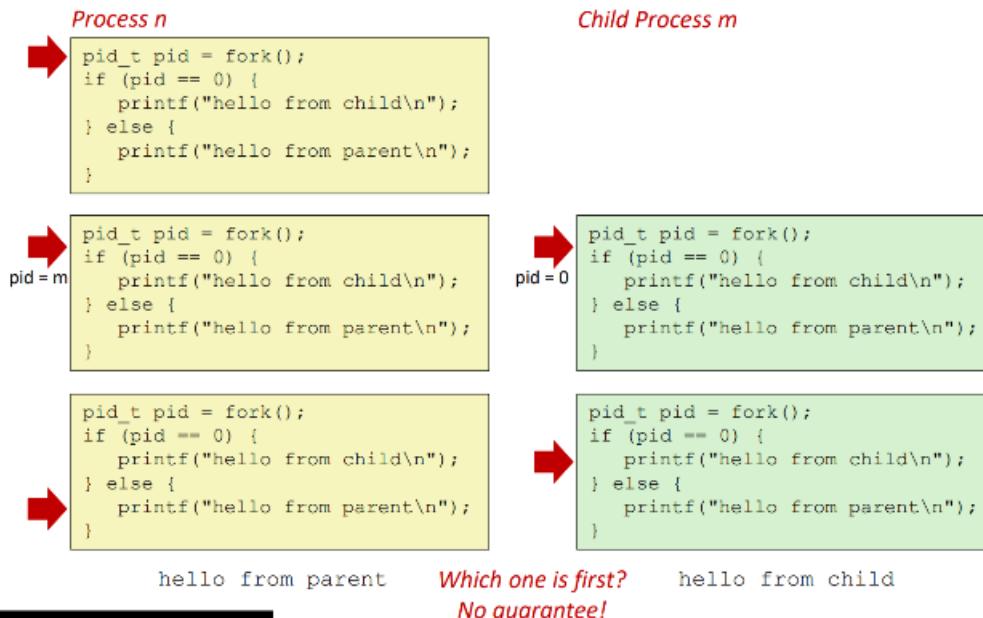
```
int fork(void)
```

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's `pid` to the parent process

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Understanding fork



Fork Example #1

Parent and child both run same code

Distinguish parent from child by return value from `fork`

Start with same state, but each has private copy

Including shared output file descriptor

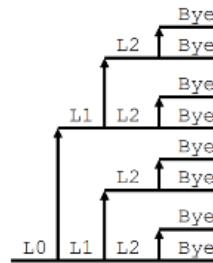
Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

Both parent and child can continue forking

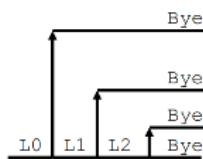
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



exit: Ending a process

```
void exit(int status)
    exits a process
• Normally return with status 0
atexit() registers functions to be executed upon
    exit
```

```
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    exit(0);
}
```

Zombies

- Idea
 - When process terminates, still consumes system resources
 - Various tables maintained by OS
 - Called a “zombie”
 - Living corpse, half alive and half dead
- Reaping
 - Performed by parent on terminated child
 - Parent is given exit status information
 - Kernel discards process
- What if parent doesn't reap?
 - If any parent terminates without reaping a child, then child will be reaped by `init` process
 - So, only need explicit reaping in long-running processes
 - e.g., shells and servers

Example

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY      TIME CMD
 6585 ttys9    00:00:00 tcsh
 6639 ttys9    00:00:03 forks
 6640 ttys9    00:00:00 forks <defunct>
 6641 ttys9    00:00:00 ps
linux> kill 6639
[1]  Terminated
linux> ps
  PID TTY      TIME CMD
 6585 ttys9    00:00:00 tcsh
 6642 ttys9    00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

ps shows child process as “defunct”

Killing parent allows child to be
reaped by init

Nonterminating Child Example

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY      TIME CMD
 6585 tttyp9    00:00:00 tcsh
 6676 tttyp9    00:00:06 forks
 6677 tttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY      TIME CMD
 6585 tttyp9    00:00:00 tcsh
 6678 tttyp9    00:00:00 ps
```

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

Child process still active even though parent has terminated

Must kill explicitly, or else will keep running indefinitely

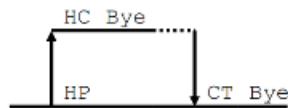
wait: Synchronizing with Children

```
int wait(int *child_status)
suspends current process until one of its children terminates
return value is the pid of the child process that terminated
if child_status != NULL, then the object it points to will be set to a status indicating why the child process terminated
```

wait: Synchronizing with Children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}
```



wait() Example

If multiple children completed, will take in arbitrary order
Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

waitpid(): Waiting for a Specific Process

waitpid(pid, &status, options)

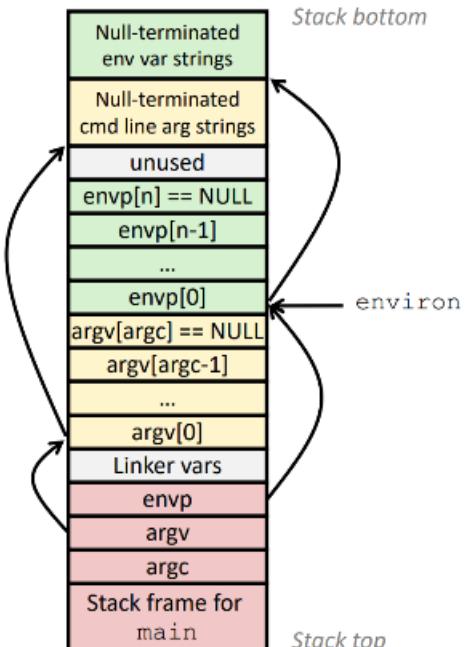
suspends current process until specific process terminates

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

execve: Loading and Running Programs

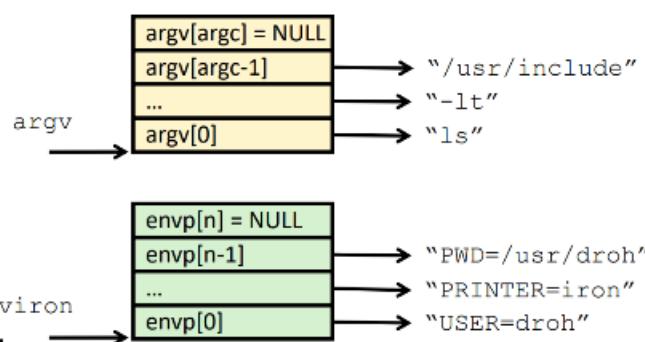
```
int execve(
    char *filename,
    char *argv[],
    char *envp[])
)
Loads and runs in current process:
Executable filename
With argument list argv
And environment variable list envp
Does not return (unless error)
Overwrites code, data, and stack
keeps pid, open files and signal context
Environment variables:
"name=value" strings
getenv and putenv
```

UNIVERSITY OF COPENHAGEN



execve Example

```
if ((pid = fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



Outline

- Process Management
- Signals

Signals

A **signal** is a small message that notifies a process that an event of some type has occurred in the system

- akin to exceptions and interrupts
- sent from the kernel (sometimes at the request of another process) to a process
- signal type is identified by small integer ID's (1-30)
- only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Sending a Signal

- Kernel **sends** (delivers) a signal to a **destination process** by updating some state in the context of the destination process
- Kernel sends a signal for one of the following reasons:
 - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
 - Another process has invoked the **kill** system call to explicitly request the kernel to send a signal to the destination process

Receiving a Signal

A destination process **receives** a signal. It is forced by the kernel to react in some way to the delivery of the signal

Three possible ways to react:

- Ignore** the signal (do nothing)
Terminate the process (with optional core dump)
Catch the signal by executing a user-level function called **signal handler**

Akin to a hardware exception handler being called in response to an asynchronous interrupt

Signal Concepts

Kernel maintains pending and blocked bit vectors in the context of each process

- **pending**: represents the set of pending signals
 - Kernel sets bit k in **pending** when a signal of type k is delivered
 - Kernel clears bit k in **pending** when a signal of type k is received
- **blocked**: represents the set of blocked signals
 - Can be set and cleared by using the **sigprocmask** function

Sending Signals with /bin/kill Program

/bin/kill program sends arbitrary signal to a process or process group

Examples

/bin/kill -9 24818

Send SIGKILL to process 24818

/bin/kill -9 -24817

Send SIGKILL to every process in process group 24817

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY      TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY      TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Example of ctrl-c and ctrl-z

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w
  PID TTY      STAT      TIME COMMAND
27699 pts/8    Ss       0:00   -tcsh
28107 pts/8    T        0:01   ./forks 17
28108 pts/8    T        0:01   ./forks 17
28109 pts/8    R+      0:00   ps w
bluefish> fg
./forks 17
<types ctrl-c>
bluefish> ps w
  PID TTY      STAT      TIME COMMAND
27699 pts/8    Ss       0:00   -tcsh
28110 pts/8    R+      0:00   ps w
```

STAT (process state) Legend:

First letter:

S: sleeping

T: stopped

R: running

Second letter:

s: session leader

+: foreground proc group

See "man ps" for more details

Receiving Signals

Suppose kernel is returning from an exception handler and is ready to pass control to process *p*

Kernel computes *pnb* = pending & ~blocked

The set of pending nonblocked signals for process *p*

If (*pnb* == 0)

Pass control to next instruction in the logical flow for *p*

Else

Choose least nonzero bit *k* in *pnb* and force process *p* to *receive* signal *k*

The receipt of the signal triggers some *action* by *p*

Repeat for all nonzero *k* in *pnb*

Pass control to next instruction in logical flow for *p*

Default Actions

Each signal type has a predefined **default action**, which is one of:

- The process terminates
- The process terminates and dumps core
- The process stops until restarted by a SIGCONT signal
- The process ignores the signal

Installing Signal Handlers

The `signal` function modifies the default action associated with the receipt of signal `signum`:

```
handler_t *signal(int signum, handler_t *handler)
```

Different values for `handler`:

`SIG_IGN`: ignore signals of type `signum`

`SIG_DFL`: revert to the default action on receipt of signals of type `signum`

Otherwise, `handler` is the address of a **signal handler**

- Called when process receives signal of type `signum`
- Referred to as "**installing**" the handler
- Executing handler is called "**catching**" or "**handling**" the signal
- When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Signal Handling Example

```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == -1)
            while(1); /* ch: */
        else
            kill(pid[i], SIGINT);
    for (i = 0; i < N; i++)
        printf("Killing proc %d\n", pid[i]);
    for (i = 0; i < N; i++)
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", i, wpid);
        else
            printf("Child %d still alive\n", i);
}
```

linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>

UNIVERSITY OF COPENHAGEN

VM as a Tool for Memory Management

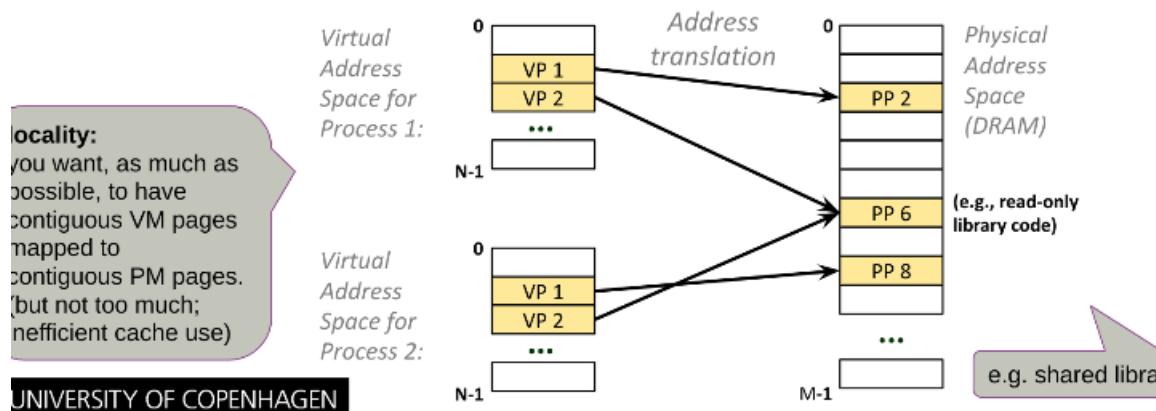
Memory allocation

Each virtual page can be mapped to any physical page

A virtual page can be stored in different physical pages at different times

Sharing code and data among processes

Map virtual pages to the same physical page (here: PP 6)



Shared Memory in Linux

Shared Memory in Linux

10s

10s

Shared Memory Segment

Memory Mapped Segment/File

<https://stackoverflow.com/questions/5656530/how-to-use-shared-memory-with-linux-in-c>

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 3K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key */
    if ((key = ftok("Hello.txt", 'R')) == -1) /* where the file must exist */
    {
        perror("ftok");
        exit(1);
    }

    /* create the segment */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
        perror("shmget");
    exit(1);
}

void* create_shared_memory(size_t size) {
    // Our memory buffer will be readable and writable
    int protection = PROT_READ | PROT_WRITE;

    // The buffer will be shared (meaning other processes can access it), but
    // anonymous (meaning third-party processes cannot obtain an address for it),
    // so only this process and its children will be able to use it.
    int visibility = MAP_SHARED | MAP_ANONYMOUS;

    // The remaining parameters to 'mmap()' are not important for this use case,
    // but the manpage for 'mmap()' explains their purpose.
    return mmap(NULL, size, protection, visibility, -1, 0);
}
```

UNIV

old UNIX way of sharing
between processes.

fork; child shares
mem w/ parent

like VP2/PP6 before

Linux default: no
sharing between
processes. If you want
sharing (shared
memory), then you have
to work for it.
There are primitives in
the C std lib for this.

Inter-Process Communication, More

We will see more IPC in the next lecture

- File I/O
- Socket I/O

back Queue (MLFQ). Hopefully you can now see why it is called : it has *multiple levels* of queues, and uses *feedback* to determine the priority of a given job. History is its guide: pay attention to how jobs ave over time and treat them accordingly.

The refined set of MLFQ rules, spread throughout the chapter, are reduced here for your viewing pleasure:

- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S , move all the jobs in the system to the topmost queue.

Take-Aways

A vectored event is a transfer of control to the OS in response to some event (*asynchronous vs. synchronous (trap, fault, abort)*).

Process has own address space and thread of control. Libs defines primitives for spawning/terminating processes, waiting for processes and executing programs within a process. The kernel takes care of context switching.

Hardware interrupts first handled in hardware then in software (kernel) through interrupt vector. Signals as process-level interrupt handling.

I/O

Reading

File systems

In unix everything is a file

Get a common abstraction for all ressources = use the same basic commands to all to read/write

fundamental concept is actually two-fold:

- In UNIX everything is a stream of bytes
- In UNIX the filesystem is used as a universal name space

In unix everything is a stream of bytes

From the programmer and the user perspectives, UNIX exposes:

- Documents stored on a hard-drive
- Directories
- Links
- Mass storage devices (e.g. hard-drive, CD-ROM, Tape, USB Key)
- Inter-process communication (e.g. Pipes, Shared Memory, UNIX Sockets)
- Network connections
- Interactive terminals
- Almost all other devices (e.g, Printers, Graphic Card)

as a stream of bytes that you can:

- read
- write
- lseek
- close

The filesystem as a universal name space

UNIX filesystem paths provide a consistent and global scheme to label resources, regardless of their nature. For instance you can reference a local directory with /usr/local, a file with /home/joe/memo.pdf, a CD-ROM with /mnt/cdrom, a directory on a network drive with /usr, a hard disk partition with /dev/sda1, a UNIX domain socket with /tmp/mysql.sock, a terminal with /dev/tty0 or even a mouse with /dev/mouse.

The namespace is hierarchical and all resources can be referenced from the root directory (/). You can access multiple filesystems within the same namespace: you just “attach” a device or a filesystem (let’s say an external hard-drive) at a specific location in the namespace (say /backups). In UNIX jargon, this action is called *mounting* a filesystem, and the namespace location where you attach the filesystem is called a *mount point*. You can reference all the resources of a mounted filesystem as a part of the global namespace by prefixing them with the mount point (say the file /backups/myproject-Oct07.zip)

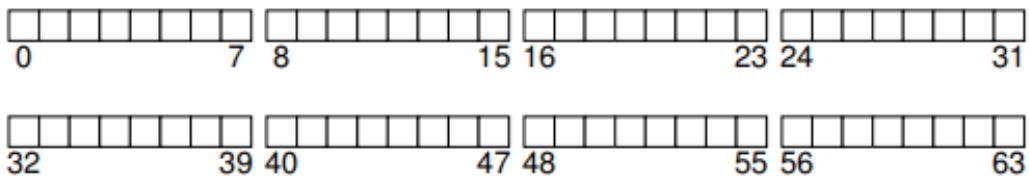
pseudo filesystems that behave like normal file systems but can be used to access resources that are not directly related to a traditional filesystem. For instance you can use a pseudo filesystem to query and control processes, access kernel internals or establish TCP connections. These pseudo filesystems provide filesystem semantics as a convenient way to represent hierarchical information and to offer uniform access to a wide variety of objects. Pseudo filesystems, sometimes also referred to as virtual filesystems, typically have no physical presence and backing storage at all, they are memory based.

Example of pseudo filesystems are:

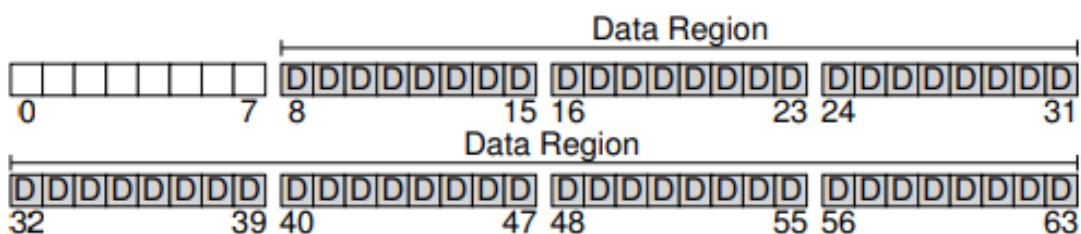
- **procfs** (/proc): The proc filesystem contains a hierarchy of special files which can be used to query or control running processes or peek into the kernel internals through standard file entries (mostly text based).
- **devfs** (/dev or /devices): Devfs presents all devices on the system as a dynamic filesystem namespace. Devfs also manages this namespace and interfaces directly with kernel device drivers to provide intelligent device management – including device entry registration/unregistration.
- **tmpfs** (/tmp): Temporary filesystem whose content disappear on reboot. Tmpfs is designed for speed and efficiency with features such as dynamic filesystem size as well as memory storage with transparent fallback to swap space.
- **portalsfs** (/p): With the BSD portal filesystem you can attach a server process to the filesystem global namespace. This can be used to provide transparent access to network services through the filesystem. For instance an application could interact with the SMTP server hosted by ph7spot.com just by opening a regular <file:/p/tcp/ph7spot.com/smtp>. The Portal filesystem is somewhat magical in that it provides socket semantic in the filesystem which can be piped and leveraged by standard UNIX tools (e.g. cat, grep, awk, etc.) – even from the shell!
- **ctfs** (/system/contract): The contract filesystem acts as a file based interface to Solaris contract subsystem. A Solaris contract defines the behavior of a process or process group for various types of event and failures – e.g. restart it if it dies. Solaris contracts provide very advanced capabilities for software management and monitoring in environments such as clustering fail-over software, batch queuing systems, and grid computing engines.

Operating systems: three easy pieces ch 40

Small disk:

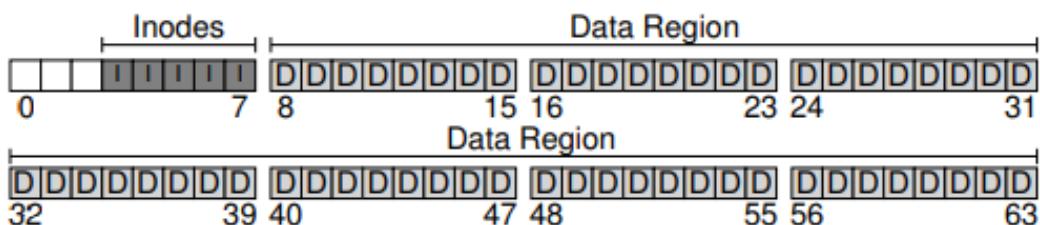


Most of the space in any file system should be user data



Metadata (which data blocks comprise file, the size, owner, access rights, access and modify times etc. -> inode)

- We have inode table (array of on-disk inode)



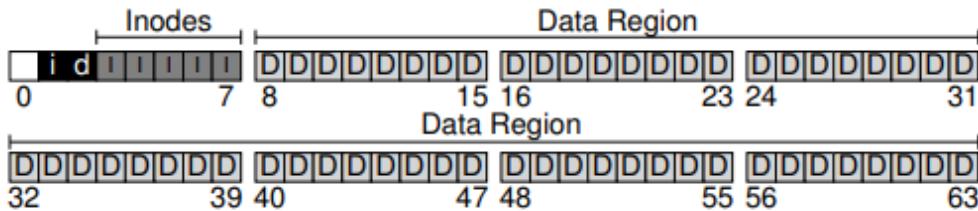
Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes = our contains 80 total inodes = amount of files that can be

- We can however allocate more space

Allocation structures to track whether inodes or data blocks are free or allocated

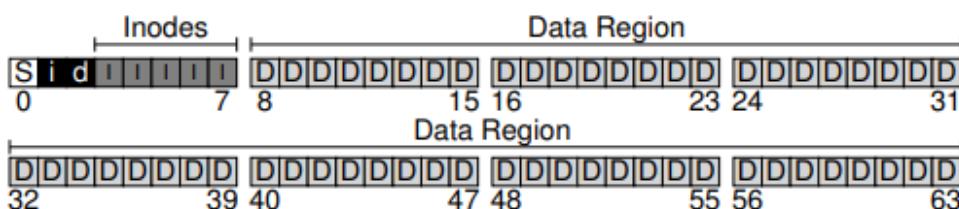
- Free list (points to the first free block etc)
- Bitmap (each bit indicate whether the corresponding block is free = 0 or in-use = 1)

Bitmap for data and one for inode



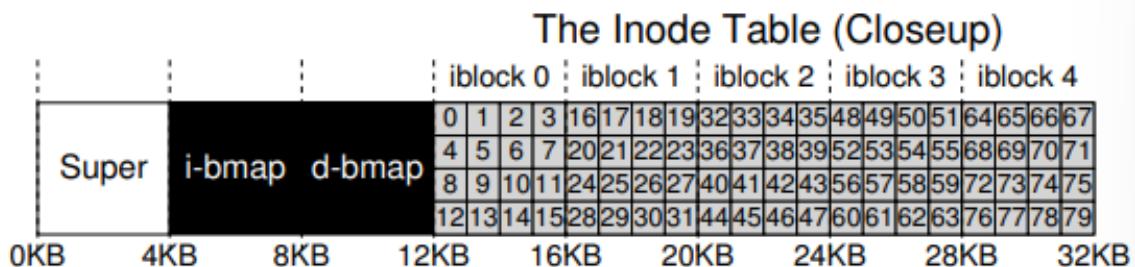
Superblock (information about file system)

- Amount of inodes and data blocks (here is 80 and 56)
- Where inode table begins (here is block 3)
- Etc
- Identifier for system type



Inode

= index node



To read inode 32

- Calculate offset $32 * \text{sizeof(inode)} \text{ or } 8192?$
- Add to start address of inode table on disk $\text{inodeStartAddr} = 12\text{KB}$
- Arrive at correct address 20KB

the sector address sector of the inode block can be calculated as follows:

```
blk = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Inode information about a file

- Type (regular file, directory etc.)
- Size (number of blocks)
- Protection information (who owns it, who can access)
- Time information (when created, modified and last accessed)
- Where data blocks are (pointers)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: Simplified Ext2 Inode

Multi-level index

Indirect pointer = points to a block that contains more pointers

Direct pointer = points to one disk block that belongs to the file

- Is limited, cannot have a large number since they take up space

Can have a fixed number of direct pointers and a single indirect pointer

- If file is large then use an indirect

Extents

- Disk pointer plus a length

pointer-based approaches are the most flexible but use a large amount of metadata per file (particularly for large files). Extent-based approaches are less flexible but more

compact; in particular, they work well when there is enough free space on the disk and files can be laid out contiguously (which is the goal for virtually any file allocation policy anyhow).

Double indirect pointer = pointer to a block that contains pointers to indirect blocks, which contain pointers to a block of data

Triple indirect pointer

Those are used for scaling larger and larger systems

Most files are small = we optimize for this case

Most files are small	~2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contain lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: File System Measurement Summary

Directory organization

Has inodes as well

Free space management

Important to know which inodes and blocks that are free

Access paths: reading and writing

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data	bar data	bar data
						[0]	[1]	[2]		
open(bar)			read			read				
				read			read			
					read					
read()							read			
read()								read		
read()									read	

Figure 40.3: File Read Timeline (Time Increasing Downward)

Traverse pathname to find inode

Most systems inode = 2

Looks for foo entry

Final step of open is to put bar's inode in memory

When open we can read

First read at offset 0 unless lseek() reads first block from file

Updates file descriptor -> so next read will be on the next block

Amount of I/O is proportional to the length of the pathname

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read		read	read			
					read write			write		
					write	read				
write()	read write							write		
					write	read				
write()	read write							write		
					write	read				
write()	read write								write	
					write					

Figure 40.4: File Creation Timeline (Time Increasing Downward)

Open() then we can write

Writing may allocate a block (unless its being overwritten)

May need to update bitmap and inode

Each write logically generates 5 I/O

- One reads bitmap (which is then updated)
- One write to bitmap
- Two to read and then write the inode
- One to write the block

A new file also need to allocate space

- Read bitmap to find free inode

- Write to inode bitmap
- Write to the new inode to initialize
- Data of the directory - link them together
- Read and write to directory to update

= 10 I/O's

Caching and buffering

How to reduce file system I/O costs

Use DRAM (system memory) to cache important blocks

Use buffer to schedule subsequent I/O's

by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes.

In case of databases and other that care about not losing writes, they force write

To FUSE or not to FUSE 1-2

FUSE = popular user-space file system framework

FUSE = Filesystem in UserSpace

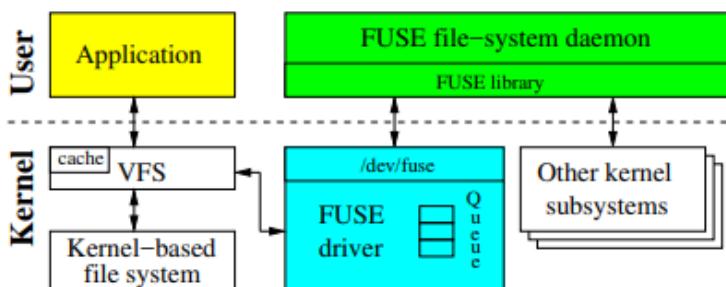


Figure 1: FUSE high-level architecture.

Group (#)	Request Types
Special (3)	INIT, DESTROY, INTERRUPT
Metadata (14)	LOOKUP, FORGET, BATCH_FORGET, CREATE, UNLINK, LINK, RENAME, RENAME2, OPEN, RELEASE, STATES, FSYNC, FLUSH, ACCESS
Data (2)	READ, WRITE
Attributes (2)	GETATTR, SETATTR
Extended Attributes (4)	SETXATTR, GETXATTR, LISTXATTR, REMOVEXATTR
Symlinks (2)	SYMLINK, READLINK
Directory (7)	MKDIR, RMDIR, OPENDIR, RELEASEDIR, REaddir, REaddirplus, FSYNCDIR
Locking (3)	GETLK, SETLK, SETLKW
Misc (6)	BMAP, FALLOCATE, MKNOD, IOCTL, POLL, NOTIFY_REPLY

User-kernel protocol

Kernel driver communicates with user-space daemon = fuse request structure.

Table 1: FUSE request types, by group (whose size is in parenthesis). Requests we discuss in the text are in bold.

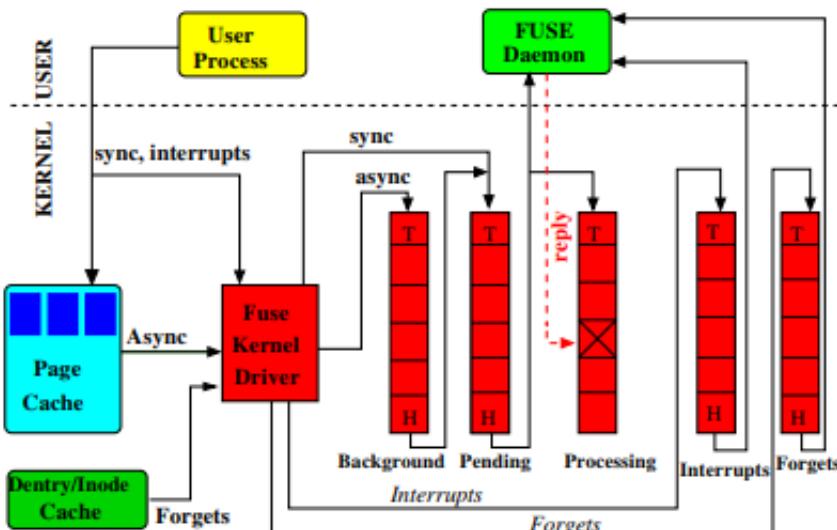


Figure 2: The organization of FUSE queues marked with their Head and Tail. The processing queue does not have a tail because the daemon replies in an arbitrary order.

I/O: files

Operating systems: three easy pieces ch 39-39.15

Files and directories

Persistent storage = hard disk drive or modern solid-state storage device

- Unlike memory, which is lost on power loss

Files and directories

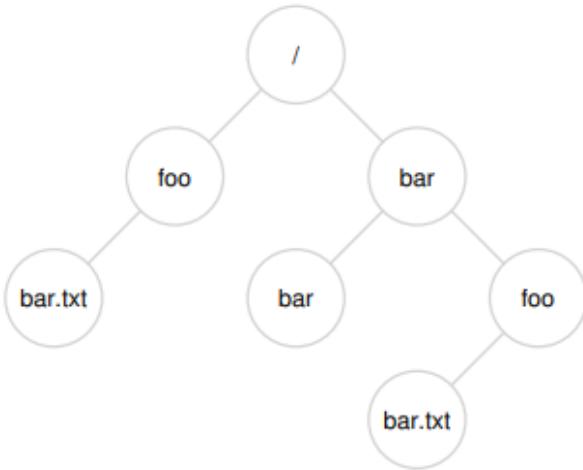


Figure 39.1: An Example Directory Tree

File = linear array of bytes

Directory = (user-readable name , low level name)

- Example = ("foo", "10")

Creating files

Creating a file called foo

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

OR

```
// option: add second flag to set permissions
```

```
int fd = creat("foo");
```

Reading and writing files

How cat works

```

prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)              = 6
write(1, "hello\n", 6)                = 6
hello
read(3, "", 4096)                   = 0
close(3)                            = 0
...
prompt>

```

Reading and writing, but not sequentially

Use lseek() to find the file with the offset.

```
off_t lseek(int fildes, off_t offset, int whence);
```

Behind whence:

If whence is SEEK_SET, the offset is set to offset bytes.

If whence is SEEK_CUR, the offset is set to its current location plus offset bytes.

If whence is SEEK_END, the offset is set to the size of the file plus offset bytes.

System Calls	Return Code	Current Offset
fd = open("file", O_RDONLY);	3	0
read(fd, buffer, 100);	100	100
read(fd, buffer, 100);	100	200
read(fd, buffer, 100);	100	300
read(fd, buffer, 100);	0	300
close(fd);	0	-

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
fd1 = open("file", O_RDONLY);	3	0	-
fd2 = open("file", O_RDONLY);	4	0	0
read(fd1, buffer1, 100);	100	100	0
read(fd2, buffer2, 100);	100	100	100
close(fd1);	0	-	100
close(fd2);	0	-	-

OFT = open file table

System Calls	Return Code	Current Offset
fd = open("file", O_RDONLY);	3	0
lseek(fd, 200, SEEK_SET);	200	200
read(fd, buffer, 50);	50	250
close(fd);	0	-

Shared file table entries: fork and dup

Mapping of file descriptor to an entry in the OFT is a one-to-one mapping

If both processes share a file table entry, only when both processes has closed the file will the entry be removed

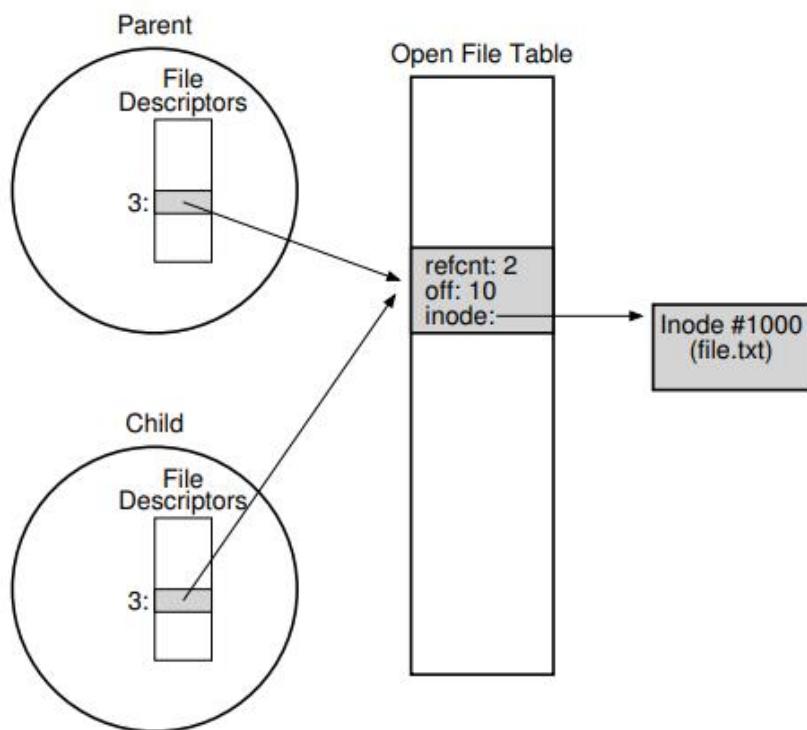


Figure 39.3: Processes Sharing An Open File Table Entry

Dup(), dup2() and dup3()

Dup creates a new file descriptor that refers to the same underlying open file as an existing descriptor

Fsync() (force write)

Force write if we can't afford to lose the action on power loss

```

int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);

```

May need to force write the directory as well as the file

Getting information about files

```

struct stat {
    dev_t      st_dev;      // ID of device containing file
    ino_t      st_ino;      // inode number
    mode_t     st_mode;     // protection
    nlink_t    st_nlink;    // number of hard links
    uid_t      st_uid;      // user ID of owner
    gid_t      st_gid;      // group ID of owner
    dev_t      st_rdev;     // device ID (if special file)
    off_t      st_size;     // total size, in bytes
    blksize_t  st_blksize;  // blocksize for filesystem I/O
    blkcnt_t   st_blocks;   // number of blocks allocated
    time_t     st_atime;    // time of last access
    time_t     st_mtime;    // time of last modification
    time_t     st_ctime;    // time of last status change
};

Figure 39.5: The stat structure.

```

Here is the output on Linux:

```

prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6  Blocks: 8  IO Block: 4096  regular file
Device: 811h/2065d Inode: 67158084  Links: 1
Access: (0640/-rw-r----)  Uid:  (30686/remzi)
          Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500

```

All of this is metadata

Remove files

```

prompt> strace rm foo
...
unlink("foo")                                = 0
...

```

Making directories

```
prompt> strace mkdir foo
...
mkdir("foo", 0777) = 0
...
prompt>
```

Reading directories

Use ls

Deleting

Rmdir() on an empty directory

Hard links

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Create a way to refer to the same file

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

Prints inode number

Use unlink() to remove the link

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

Cant create one to a directory

- May create cycles in the tree

Symbolic/soft links

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

Symbolic link is a file itself

```
prompt> stat file
... regular file ...
prompt> stat file2
... symbolic link ...
```

A = regular files

D = directories

L = soft links

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 .
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ..
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

File size is because of the pathname??

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi 6 May 3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 ->
               alongerfilename

prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

I/O: sockets

Beej's guide to network programming ch 2-5

What is a socket?

A file can be

- Network connection
- FIFO
- A PIPE
- A terminal
- A real on-the-disk file
- Anything else

Sockets has a socket descriptor, which we can communicate through (it's a file descriptor)

Internet sockets (internet addresses)

unix sockets (path names on a local node)

x.25 sockets (addresses that we can ignore?)

Two types of internet sockets

There are more than 2 types

- Raw sockets are also powerful

Stream sockets and datagram sockets(connectionless sockets)

Stream socket

- Two way connected communication stream
- Are error free
- Gives the original order of messages

Ssh uses stream sockets

HTTP uses stream sockets to get pages

Stream sockets use TCP (the transmission control protocol)

Datagram sockets

- May arrive out of order
- If it arrives the packet is error-free

Use UDP (user datagram protocol)

Don't have to maintain an open connection

When a packet is sent, the receiver will send a packet back that states it got it (ACK packet)

UDP is a good choice for games = delivers speed

Low level nonsense and network theory



Figure 2.1: Data Encapsulation.

Layered Network Model (aka “ISO/OSI”)

- write sockets programs that are exactly the same without caring how the data is physically transmitted (serial, thin Ethernet, AUI)

- Lower level programs deal with this

Layers

- Application (users interact)
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical (ethernet etc)

Layered model with unix

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*Ethernet, wi-fi, or whatever*)

IP Addresses, structs, and data munging

IPv4 = 192.0.2.111

Run out of space -> use IPv6

2001:0db8:c9d2:ae5:73e3:934a:a5ae:9551

Big-endian = network byte order

Little-endian = host byte order (intel 89x86)

Function	Description
htons()	host to network short
htonl()	host to network long
ntohs()	network to host short
ntohl()	network to host long

Firewall using NAT translates the computers IP address to another address

Jumping from IPv4 to IPv6

- First of all, try to use [getaddrinfo\(\)](#) to get all the struct sockaddr info, instead of packing the structures by hand. This will keep you IP version-agnostic, and will eliminate many of the subsequent steps.
- Any place that you find you're hard-coding anything related to the IP version, try to wrap up in a helper function.
- Change AF_INET to AF_INET6.
- Change PF_INET to PF_INET6.
- Change INADDR_ANY assignments to in6addr_any assignments, which are slightly different:
`struct sockaddr_in sa;`
`struct sockaddr_in6 sa6;`

```
sa.sin_addr.s_addr = INADDR_ANY; // use my IPv4 address  
sa6.sin6_addr = in6addr_any; // use my IPv6 address
```

Also, the value IN6ADDR_ANY_INIT can be used as an initializer when the struct in6_addr is declared, like so:

```
struct in6_addr ia6 = IN6ADDR_ANY_INIT;
```

- Instead of struct sockaddr_in use struct sockaddr_in6, being sure to add "6" to the fields as appropriate (see [structs](#), above). There is no sin6_zero field.
- Instead of struct in_addr use struct in6_addr, being sure to add "6" to the fields as appropriate (see [structs](#), above).
- Instead of inet_aton() or inet_addr(), use inet_pton().
- Instead of inet_ntoa(), use inet_ntop().
- Instead of gethostbyname(), use the superior getaddrinfo().
- Instead of gethostbyaddr(), use the superior getnameinfo() (although gethostbyaddr() can still work with IPv6).
- INADDR_BROADCAST no longer works. Use IPv6 multicast instead.

System calls or bust

Getaddrinfo()

- Does DNS and service name lookups
- Fills out structs

Socket()

- Type of IP, type of socket, TCP or UDP
- Return socket descriptor

Bind()

- Port matches a packet to a socket descriptor
- Socket file descriptor, pointer to information about addresses(port and IP address), length of bytes to that address

Connect()

- Connect to remote host
- Socket file descriptor, port and IP address, length of server address structure
- Only care about where we are going

Listen()

- Wait for connections then accept
- Socket file descriptor, amount of connections allowed in queue

Bind needs to happen before listen, need to be on a port

Accept()

- Accept one from the queue
- Gives a whole new socket file descriptor for the 1 connection
- Listen socket descriptor, pointer to a local struct storage (information about incoming connection), size of the struct

Sockets and networking - wire protocols

Set of messages represented as a byte sequence

Rules for designing

- Keep the number of different messages **small**. It's better to have a few commands and responses that can be combined rather than many complex messages.
- Each message should have a well-defined purpose and **coherent** behavior.
- The set of messages must be **adequate** for clients to make the requests they need to make and for servers to deliver the results.

Platform independence

- **Safe from bugs**
 - The protocol should be easy for clients and servers to generate and parse. Simpler code for reading and writing the protocol (whether written with a parser generator like ANTLR, with regular expressions, etc.) will have fewer opportunities for bugs.
 - Consider the ways a broken or malicious client or server could stuff garbage data into the protocol to break the process on the other end.
Email spam is one example: when we spoke SMTP above, the mail server asked *us* to say who was sending the email, and there's nothing in SMTP to prevent us from lying outright. We've had to build systems on top of SMTP to try to stop spammers who lie about From: addresses.
Security vulnerabilities are a more serious example. For example, protocols that allow a client to send requests with arbitrary amounts of data require careful handling on the server to avoid running out of buffer space, [or worse](#).
- **Easy to understand:** for example, choosing a text-based protocol means that we can debug communication errors by reading the text of the client/server exchange. It even allows us to speak the protocol "by hand" as we saw above.
- **Ready for change:** for example, HTTP includes the ability to specify a version number, so clients and servers can agree with one another which version of the protocol they will use. If we need to make changes to the protocol in the future, older clients or servers can continue to work by announcing the version they will use.

Use a grammar

Specifications:

- **What are the preconditions of a message?** For example, if a particular field in a message is a string of digits, is any number valid? Or must it be the ID number of a record known to the server?
Under what circumstances can a message be sent? Are certain messages only valid when sent in a certain sequence?
- **What are the postconditions?** What action will the server take based on a message? What server-side data will be mutated? What reply will the server send back to the client?

I/O: message-passing

Dive into systems - 15.3

To exascale and beyond: cloud computing, big data, and the future of computing.

Sophisticated multinode supercomputers form the foundation of high-performance computing (HPC).

High-end data analysis (HDA) systems

Slides

Topics

- File Systems
 - implementation, conceptually
 - implementation, in Linux (VFS)
 - virtual file systems
- I/O APIs
 - File
 - Socket
 - Message-Passing Interface (MPI)

File systems

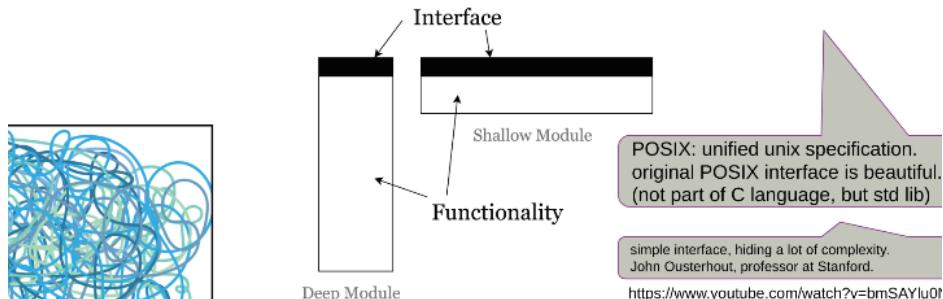
Implementation, conceptually

Files

like memory!

A file is an array of bytes

File interface: create/delete, open/close, read/write



File System, Conceptual Overview

- **path**, hierarchy of
- **directory**, collection of
- **filename**, user-friendly name of a
- **inode**, identifies a (& holds metadata on)
- **file**, collection of
- **block**

let's start with **blocks**.

Block Layer

we start with:

30s

A **block device** is an **array of blocks**.

To each block is associated a number,
a Logical Block Address (LBA)

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block  
return device[b]
```

sound familiar?
virtual memory! pages!
(quantized data)
block is a unit of transfer.
(associativity layer maps block
number to actual block)

hard disk drives today
(incl. SSDs) are block devices.

File Layer

How to represent files?
Each file is a **collection of disk blocks**
(more abstractly (haha), an **array of bytes**)

```
structure inode {
    integer block_numbers[N]; // the numbers of the blocks that constitute the file
    integer size;           // the size of the file in bytes
}

procedure INDEX_TO_BLOCK_NUMBER (instance of inode i, integer index) returns integer{
    return i.block_numbers[index];
}

procedure INODE_TO_BLOCK (integer offset, instance of inode i) returns instance of block
{
    o ← offset / BLOCKSIZE;
    b ← INDEX_TO_BLOCK_NUMBER(i, o);
    return BLOCK_NUMBER_TO_BLOCK(b);
}
```

inode ("index node") is a collection of block numbers (associated to the file), and their collective size.
(we need this level of indirection)

node Name Layer

File system state
inode_table

How to avoid carrying inodes around?

level of indirection

number the inodes!
inode_number to inode table (map),
carry this table around.

```
procedure INODE_NUMBER_TO_INODE(integer inode_number) returns instance of inode{
    return inode_table[inode_number];
}

procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
    returns instance of block {
    structure inode i ← INODE_NUMBER_TO_INODE (inode_number);
    o ← offset / BLOCKSIZE;
    b ← INDEX_TO_BLOCK_NUMBER (i, o);
    return BLOCK_NUMBER_TO_BLOCK (b);
}
```

File Name Layer

Representing directories

File system stat
inode_table

```
structure inode{  
    integer block_numbers[N]; // the numbers of the blocks that constitute the file  
    integer size;           // the size of the file in bytes  
    integer type;          // type of file: regular file, directory,...  
}
```

directory represented as inode.
(now have 2 types of inodes).

directory contents represented:
each block stores inode nums

User-friendly names

File name	Inode number
program	10
Paper	12

when you work with files, you
don't work with inode numbers
you work with filenames.
need mapping from filename to
inode number.

in dir, we store,
alongside an inode number,
the *filename* of that inode.

UNIVERSITY OF COPENHAGEN

File Name Layer

Directory lookup

File system stat
inode_table

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer {  
    return LOOKUP (filename, dir);  
}
```

(inode number)

```
procedure LOOKUP (character string filename, integer dir) returns integer {  
    instance of block b;  
    instance of inode i ← INODE_NUMBER_TO_INODE (dir);  
    if i.type ≠ DIRECTORY then return FAILURE;  
    for offset from 0 to i.size - 1 do {  
        b ← INODE_NUMBER_TO_BLOCK (offset, dir);  
        if STRING_MATCH (filename, b) then {  
            return INODE_NUMBER (filename, b); // return inode number for filename  
        }  
        offset ← offset + BLOCKSIZE; // increase offset by block size  
    }  
    return FAILURE;  
}
```

if filename occurs in b,

then return the inode num
that's written next to the
filename

STRING_MATCH, INODE_NUMBER
implementation not shown

Path Name Layer

File system state:
inode_table

Hierarchy of Directories

```
procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer{
    if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir);
    else {
        dir ← LOOKUP (FIRST (path), dir);
        path ← REST (path);
        return PATH_TO_INODE_NUMBER (path, dir);
    }
}
```

Absolute Path Name Layer

File system state:
inode_table
Process state:
wd

Change working directory

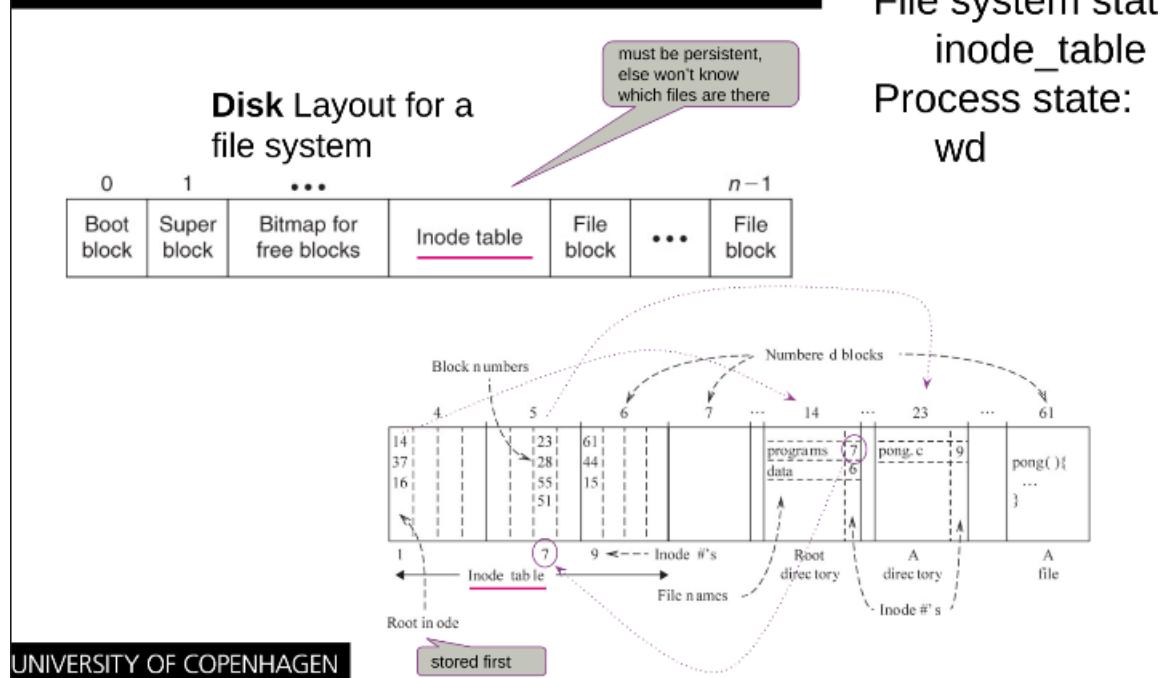
```
procedure CHDIR (path character string) { wd ← PATH_TO_INODE_NUMBER (path, wd); }
```

How to name a file regardless of the current working directory?

```
procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer{
    if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1);
    else return PATH_TO_INODE_NUMBER(path, wd);
}
```

(root inode number)

Unix File System Naming Scheme



UNIVERSITY OF COPENHAGEN

File system stat
inode_table
Process state:
wd

Symbolic Link Layer

we can have multiple paths
to the same inode.

How about flexible management of files?

```
LINK (from_name, to_name);
UNLINK (from_name);
```

```
structure inode{
    integer block_numbers[N];
    integer size;
    integer type;
    integer refcnt;
}
```

once no path refers to inode
it can be garbage collected.

Symbolic Link Layer

in Linux, you can mount a file system

10s

How to attach new disks to a file system?

MOUNT (""/dev/fd1", "/floppy")

(block) device
(should contain file system)

mount point
(where root of mounted file system
is to be accessible)

UNIVERSITY OF COPENHAGEN

"In UNIX, Everything is a File"
block devices (e.g. discovered
during bootup, or e.g. when you
plug in a USB) represented as file
in the root file system on boot.

Naming Layers in Unix File System

Layer	Names	Values	Context	Name-mapping algorithm	
Symbolic link	Path names	Path names	The directory hierarchy	PATHNAME_TO_GENERAL_PATH	
Absolute path name	Absolute path names	Inode numbers	The root directory	GENERALPATH_TO_INODE_NUMBER	
Path name	Relative path names	Inode numbers	The working directory	PATH_TO_INODE_NUMBER	
File name	File names	Inode numbers	A directory	NAME_TO_INODE_NUMBER	
Inode number	Inode numbers	Inodes	The inode table	INODE_NUMBER_TO_INODE	↑ user-oriented names ↓
File	Index numbers	Block numbers	An inode	INDEX_TO_BLOCK_NUMBER	machine-user interface
Block	Block numbers	Blocks	The disk drive	BLOCK_NUMBER_TO_BLOCK	↑ machine- oriented names ↓

API: State

Which files are in use?
file_table

File name	Inode number	cursor
program	10	64
Paper	12	0

Cursor is the first byte that will be accessed by the next read or write operation.

Which files is each process using?
fd_table

Mapping from file descriptors into the **file_table**.
(file descriptors are per-process, natural numbers; 0 is stdin, 1 is stdout, 2 is stderr, ...)

Multiple processes can have a file open with different cursors, and

Multiple processes can have a file open sharing a cursor (fork; **fd_table** shared)

File system state:

inode_table

file_table

Process state:

fd_table

wd

API: inode

```
structure inode {
    integer block_numbers[N]; // the number of blocks that constitute the file
    integer size; // the size of the file in bytes
    integer type; // type of file: regular file, directory, symbolic link
    integer refcnt; // count of the number of names for this inode
    integer userid; // the user ID that owns this inode
    integer groupid; // the group ID that owns this inode
    integer mode; // inode's permissions
    integer atime; // time of last access (READ, WRITE,...)
    integer mtime; // time of last modification
    integer ctime; // time of last change of inode }
```

we did not talk about
e.g. access control
(but we will!)

I/O API

File

I/O API : File

recall: **create/delete, open/close, read/write, link/unlink.**

in Linux:

```
1 int creat  ( const char *pathname, mode_t mode);
2 int remove /* calls `unlink` for files, `rmdir` for directories (refcnt...)
3
4 int link   ( const char *oldpath, const char *newpath);
5 int unlink ( const char *pathname);
6
7 int open   ( const char *pathname, int flags, ... /* mode_t mode */ );
8 int close  ( int fd);
9
10 ssize_t read ( int fd, void buf[.count], size_t count);
11 ssize_t write ( int fd, const void buf[.count], size_t count);
```

with conceptual details above, you can imagine how these work; see **man-pages** for details.

will show **open** and **read** (conceptually) shortly.

Linux I/O System Calls

- creat, open, read, write, close, lseek
- fsync
- link, unlink
- stat, lstat, fstat
- access, umask, chmod, chown, utime
- ioctl

libraries for this

there are also *async I/O* system calls. (e.g. aio_read)
and ways to batch system calls (io_submit, ...)

API Calls: Open

(skippable)
File system state:
inode_table
file_table
Process state:
fd_table
wd

```
procedure OPEN (character string filename, flags, mode) {
    inode_number ← PATH_TO_INODE_NUMBER (filename, wd);
    if inode_number = FAILURE and flags = O_CREATE then { // Create the file?
        inode_number ← CREATE (filename, mode);           // Yes, create it.
    } else return FAILURE;
    inode ← INODE_NUMBER_TO_INODE (inode_number);
    if PERMITTED (inode, flags) then { // Does this user have the required permissions?
        file_index ← INSERT (file_table, inode_number);
        fd ← FIND_UNUSED_ENTRY (fd_table); // Yes, find entry in file descriptor table
        fd_table[fd] ← file_index;       // Record file index for the file descriptor
        return fd;                      // Return fd
    } else return FAILURE;           // No, return a failure
}
```

API Calls: Read

File name	Inode number	cursor
program	10	64
Paper	12	0

(skippable)
File system state:
 inode_table
 file_table
Process state:
 fd_table
 wd

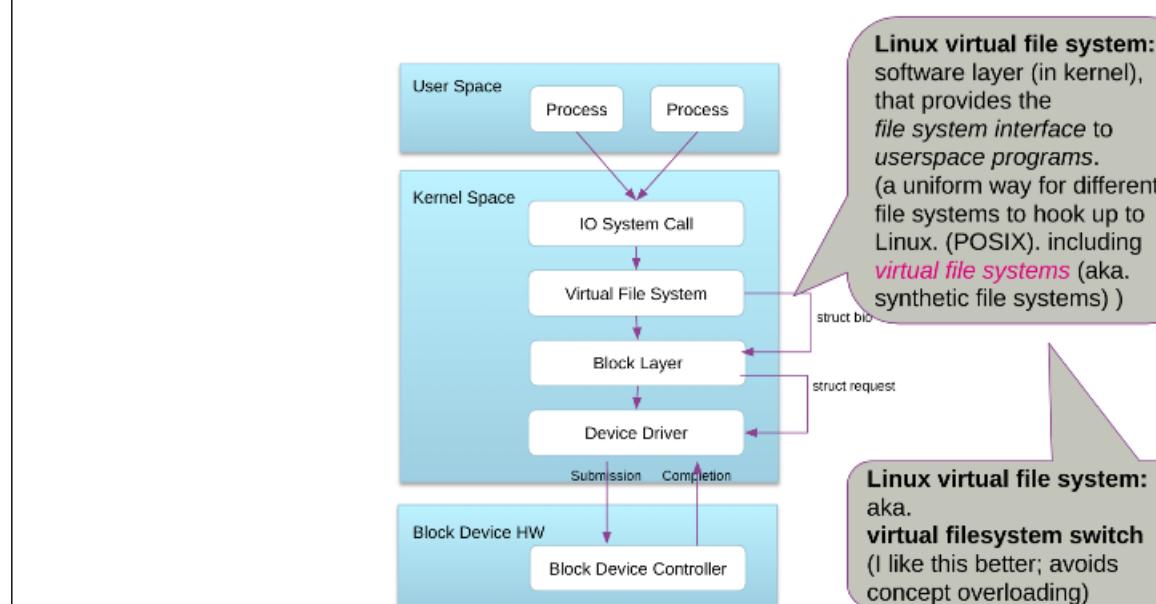
```

procedure READ (fd, reference buf, n) {
    file_index ← fd_table[fd];
    cursor ← file_table[file_index].cursor;
    inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number);
    m = MINIMUM (inode.size - cursor, n);
    atime of inode ← NOW ();
    if m = 0 then return END_OF_FILE;
    for i from 0 to m - 1 do {
        b ← INODE_NUMBER_TO_BLOCK (i, inode_number);
        COPY (b, buf, MINIMUM (m - i, BLOCKSIZE));
        i ← i + MINIMUM (m - i, BLOCKSIZE);
    }
    file_table[file_index].cursor ← cursor + m;
    return m;
}
  
```

File Systems

Implementation, in linux

Linux File System



Linux Virtual File System

The virtual file system defines the generic file system interface and data structures:

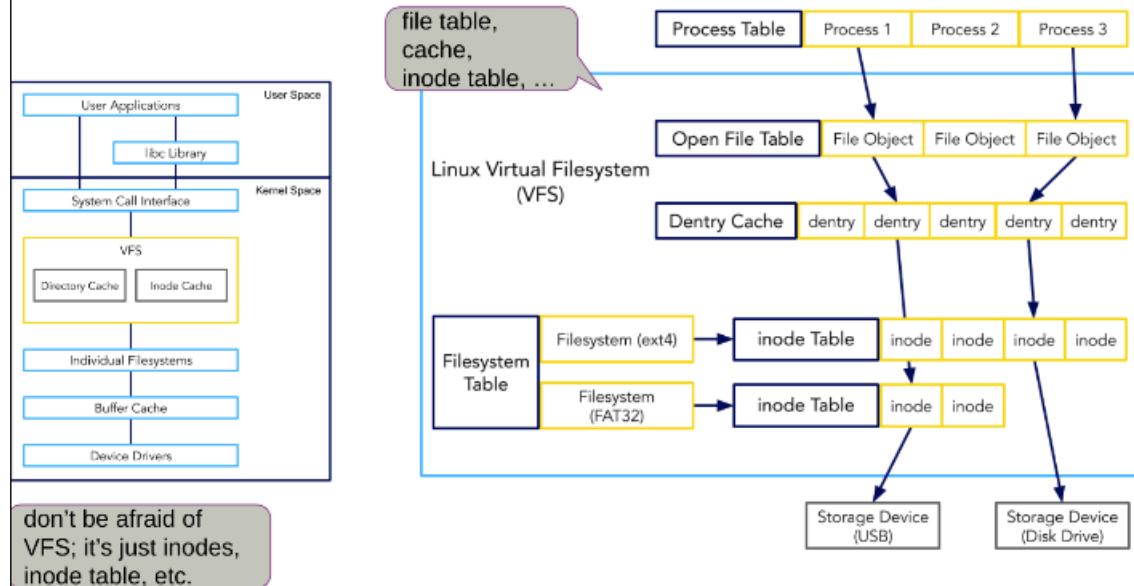
file, dentry, inode, vfsmount, super_block.

Each specific file system provides a specific implementation:

block-based FS (ext4, btrfs), network FS (NFS, ceph),
stackable FS, pseudo FS (sysfs),
special purpose FS (tmpfs)

Linux VFS

they all respect the VFS setup.



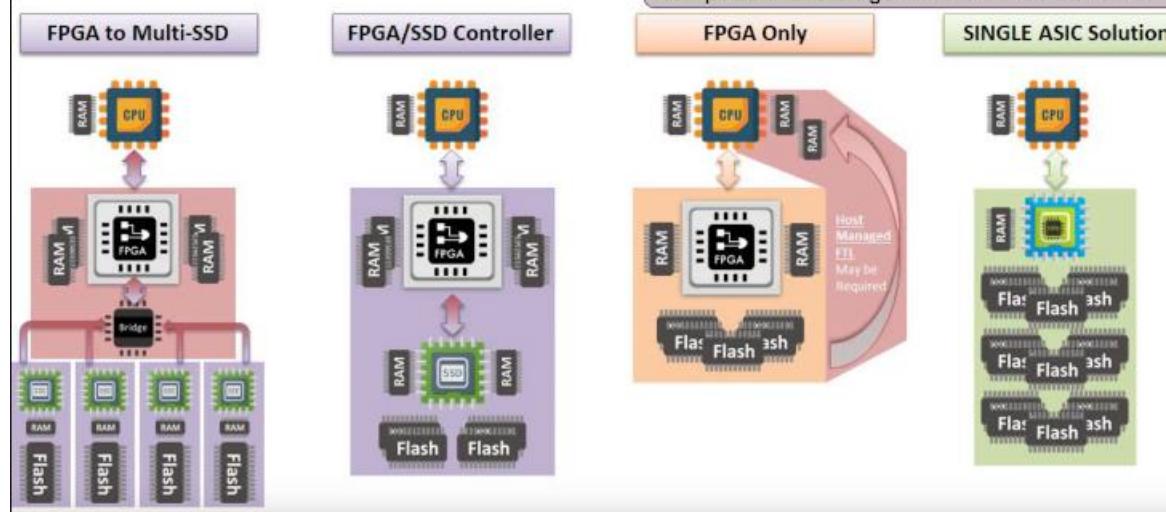
Aside

Computational storage

Computational storage = computation on the IO path

Computational storage

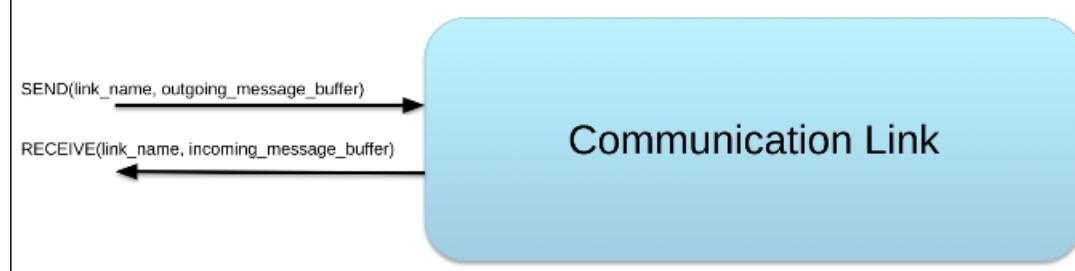
<https://www.youtube.com/watch?v=3CeO1Y1PO-Y>
storage industry has defined an architecture for computational storage. NVMe will be standard.



Specialized storage interface + functionality offload

Communication Abstraction

what we get to... is a **communication abstraction**.
not just blocks, but e.g. 8MB object, transactions, ..



File systems

Virtual file systems

In Unix, Everything is a File

originated in Plan 9 mid-80s. concept is two-fold

- *in UNIX, everything is a stream of bytes*
- *in UNIX, the filesystem is used as a universal namespace*

global file system as unified namespace for heterogeneous resources. (that is very convenient).

in nutshell: have stream of bytes $\Rightarrow \Rightarrow$ can implement the file I/O API, s.t. invocations of the file I/O API (through the global file system, on a mount-point for the byte-stream) will do whatever your file I/O API implementation decides to said byte stream.

ple of pseudo filesystems are:

(/proc): The proc filesystem contains a hierarchy of special files which can be used to query or control running processes or peek into the kernel internals through standard file entries (mostly text based).

(/dev OR /devices): Devfs presents all devices on the system as a dynamic system namespace. Devfs also manages this namespace and interfaces directly with kernel device drivers to provide intelligent device management – including device registration/unregistration.

(/tmp): Temporary filesystem whose content disappear on reboot. Tmpfs is tuned for speed and efficiency with features such as dynamic filesystem size as well as memory storage with transparent fallback to swap space.

fs (/p): With the BSD portal filesystem you can attach a server process to the system global namespace. This can be used to provide transparent access to network services through the filesystem. For instance an application could interact with an MTP server hosted by ph7spot.com just by opening a regular file: `p/ph7spot.com/smtp`. The Portal filesystem is somewhat magical in that it provides a semantic in the filesystem which can be piped and leveraged by standard UNIX tools (e.g. cat, grep, awk, etc.) – even from the shell!

2.1 High-Level Architecture

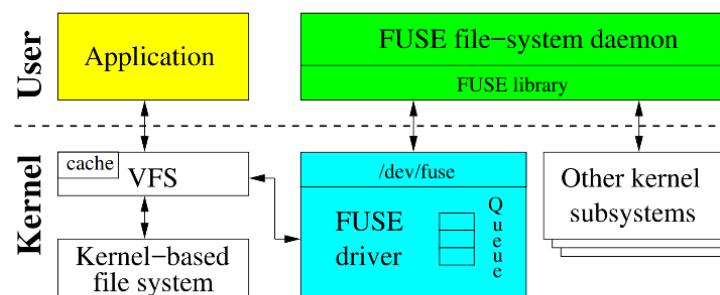


Figure 1: FUSE high-level architecture.

FUSE consists of a kernel part and a user-level daemon. The kernel part is implemented as a Linux kernel module that, when loaded, registers a *fuse* file-system driver with Linux's VFS. This *Fuse* driver acts as a proxy for various specific file systems implemented by different user-level daemons.

In addition to registering a new file system, FUSE kernel module also registers a `/dev/fuse` block device. This device serves as an interface between user-space FUSE daemons and the kernel. In general, a daemon reads FUSE requests from `/dev/fuse`, processes them, and then writes replies back to `/dev/fuse`.

FUSE: Example implementations

- [gcsfuse](#) - A user-space file system for interacting with Google Cloud Storage. Language: Golang.
- [sshfs](#) - File system based on the SSH File Transfer Protocol; same authors as [osxfuse](#). Language: C.
- [sshfs](#) - A network filesystem client to connect to SSH servers; same authors as [libfuse](#). Language: C.
- [go-ipfs](#) - IPFS implementation in go. Language: Golang.
- [mirrdfs](#) - Go filesystem project using bazil/fuse. Language: Golang.
- [gocryptfs](#) - Encrypted overlay filesystem written in Go. Language: Golang.
- [tahoe-lafs](#) - The Tahoe-LAFS decentralized secure filesystem. Language: Python.
- [btfs](#) - A bittorrent filesystem based on FUSE. Language: C++.
- [google-drive-ocamlfuse](#) - FUSE filesystem over Google Drive. Language: OCaml.
- [mp3fs](#) - FUSE-based transcoding filesystem from FLAC to MP3. Language: C++.
- [encfs](#) - An Encrypted Filesystem for FUSE. Language: C++.
- [GDriveFS](#) - An innovative FUSE wrapper for Google Drive; Language: Python.
- [pachyderm](#) - Containerized Data Analytics. Language: Golang.
- [camlistore](#) - Personal storage system for life: a way of storing, syncing, sharing, modelling and backing up content. Language: Golang.
- [svfs](#) - The Swift Virtual File System. Language: Golang.
- [restic](#) - restic backup program. Language: Golang.
- [unionfs-fuse](#) - union filesystem using fuse. Language: C.
- [GlusterFS](#) - Storage for your Cloud. Language: C.
- [LoggedFS](#) - Filesystem monitoring with Fuse. Language: C++.
- [go-mtpfs](#) - Mount MTP devices over FUSE. Language: Go.
- [ntfs](#) - Filesystem that stores your data in nt. Language: C.
- [s3fs](#) - FUSE-based file system backed by Amazon S3. Language: C++.

Could be Minecraft

Takeaways: Files

Take-Aways

File abstraction is one of Unix enduring contribution. Beautiful example of a deep module.

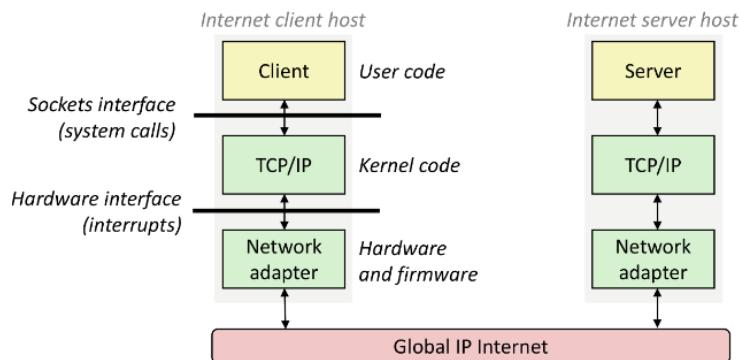
You should be able to describe the data structures and name mapping steps involved in file system operations.

With computational storage, storage devices are moving from a memory to a communication abstraction.

I/O API

Sockets

Hardware and Software Organization of an Internet Application



Global IP Internet (upper case)

Most famous example of an internet

Based on the TCP/IP protocol family

IP (Internet Protocol) :

Provides **basic naming scheme** and unreliable **delivery capability** of packets (datagrams) from **host-to-host**

UDP (Unreliable Datagram Protocol)

Uses IP to provide **unreliable** datagram delivery from **process-to-process**

TCP (Transmission Control Protocol)

Uses IP to provide **reliable** byte streams from **process-to-process** over **connections**

Accessed via a mix of Unix file I/O and functions from the **sockets interface**

A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit *IP addresses*

130.226.140.95 (2003)

- We are trying to fully replace 32-bit addresses with 64-bit addresses. It has not happened yet (2022)

2. The set of IP addresses is mapped to a set of identifiers called Internet *domain names*

130.226.140.95 is mapped to cos.itu.dk

3. A process on one Internet host can communicate with a process on another Internet host over a *connection*

(1) IP Addresses

32-bit IP addresses are stored in an *IP address struct*

IP addresses are always stored in memory in *network byte order* (big-endian byte order)

True in general for any integer transferred in a packet header from one machine to another.

E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    uint32_t s_addr; /* network byte order (big-endian) */
};
```

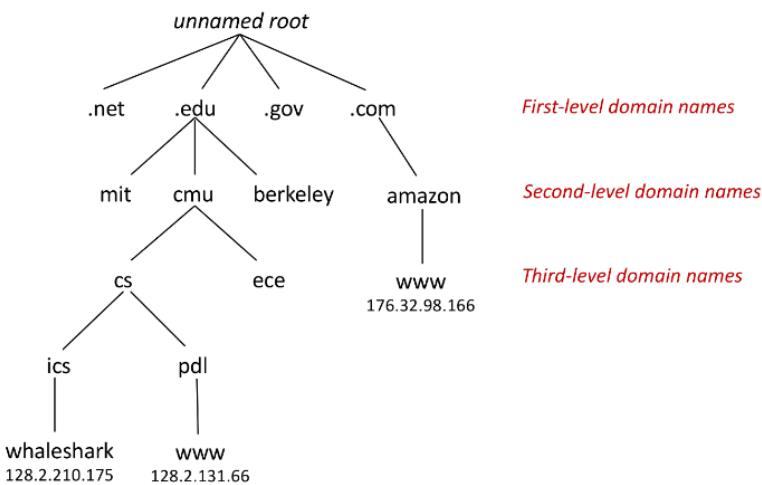
Dotted Decimal Notation

By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period

IP address: 0x**8002C2F2** = **128.2.194.242**

Use `getaddrinfo` and `getnameinfo` functions (described later) to convert between IP addresses and dotted decimal format.

(2) Internet Domain Names



Domain Naming System (DNS)

The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called **DNS**

Conceptually, programmers can view the DNS database as a collection of millions of **host entries**.

Each host entry defines the mapping between a set of domain names and IP addresses.

In a mathematical sense, a host entry is an equivalence class of domain names and IP addresses.

3) Internet Connections

Clients and servers communicate by sending streams of bytes over **connections**. Each connection is:

Point-to-point: connects a pair of processes.

Full-duplex: data can flow in both directions at the same time,

Reliable: stream of bytes sent by the source is eventually received by the destination in the same order it was sent.

A **socket** is an endpoint of a connection

Socket address is an **IPaddress : port** pair

A **port** is a 16-bit integer that identifies a process:

Ephemeral port: Assigned automatically by client kernel when client makes a connection request.

Well-known port: Associated with some **service** provided by a server (e.g., port 80 is associated with Web servers)

Well-known Ports and Service

Popular services have permanently assigned *well-known ports* and corresponding *well-known service names*:

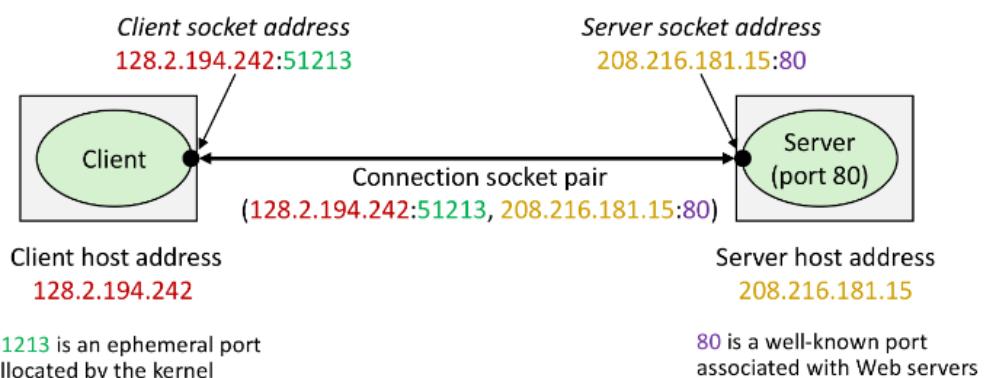
- echo server: 7/echo
- ssh servers: 22/ssh
- email server: 25/smtp
- Web servers: 80/http

Mappings between well-known ports and service names is contained in the file `/etc/services` on each Linux machine.

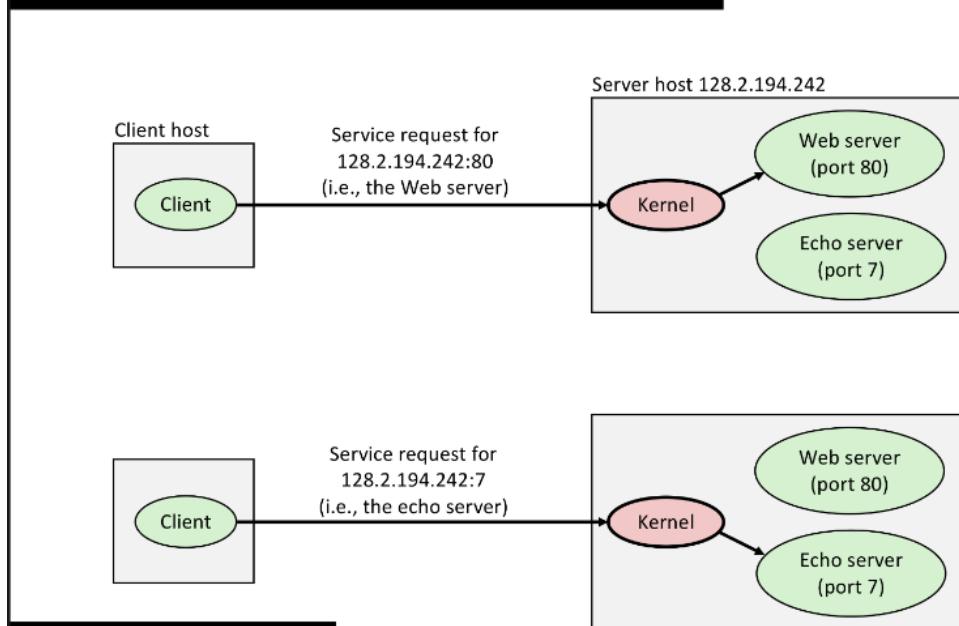
Anatomy of a Connection

A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)

$(\text{cliaddr}:\text{cliport}, \text{servaddr}:\text{servport})$



Using Ports to Identify Services



Sockets Interface

Set of system-level functions used in conjunction with Unix I/O to build network applications.

Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

Available on all modern systems

Unix variants, Windows, OS X, IOS, Android, ARM

Sockets

What is a socket?

To the kernel, a socket is an endpoint of communication

To an application, a socket is a file descriptor that lets the application read/write from/to the network

Remember: All Unix I/O devices, including networks, are modeled as files

Clients and servers communicate with each other by reading from and writing to socket descriptors



The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors

Socket Address Structures

Generic socket address:

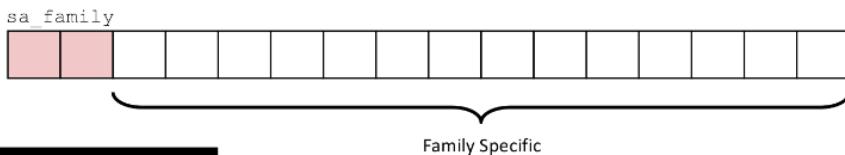
For address arguments to `connect`, `bind`, and `accept`

Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed

For casting convenience, we adopt the Stevens convention:

```
typedef struct sockaddr SA;
```

```
struct sockaddr {  
    uint16_t sa_family; /* Protocol family */  
    char     sa_data[14]; /* Address data. */  
};
```

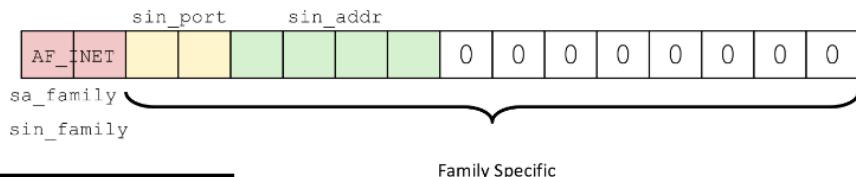


Socket Address Structures

Internet-specific socket address:

Must cast `(struct sockaddr_in *)` to `(struct sockaddr *)` for functions that take socket address arguments.

```
struct sockaddr_in {  
    uint16_t      sin_family; /* Protocol family (always AF_INET) */  
    uint16_t      sin_port;   /* Port num in network byte order */  
    struct in_addr sin_addr; /* IP addr in network byte order */  
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */  
};
```



Host and Service Conversion: getaddrinfo

```
int getaddrinfo(const char *host,          /* Hostname or address */
                const char *service,        /* Port or service name
*/
                const struct addrinfo *hints, /* Input parameters */
                struct addrinfo **result); /* Output linked list */

void freeaddrinfo(struct addrinfo *result); /* Free linked list */

const char *gai_strerror(int errcode);      /* Return error msg */
```

Given host and service, `getaddrinfo` returns result that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.

Helper functions:

- `freeaddrinfo` frees the entire linked list.
- `gai_strerror` converts error code to an error message.

Sockets Interface: socket

Clients and servers use the `socket` function to create a *socket descriptor*:

Example:

```
int socket(int domain, int type, int protocol)
```

```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using
32-bit IPv4 addresses

Indicates that the socket
will be the end point of a
connection

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Sockets Interface: bind

A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.

Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

Sockets Interface: listen

By default, kernel assumes that descriptor from socket function is an **active socket** that will be on the client end of a connection.

A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

```
int listen(int sockfd, int backlog);
```

Converts sockfd from an active socket to a **listening socket** that can accept connection requests from clients.

backlog is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.

Sockets Interface: accept

Servers wait for connection requests from clients by calling accept:

```
int accept(int listenfd, SA *addr, int *addrlen);
```

Waits for connection request to arrive on the connection bound to listenfd, then fills in client's socket address in addr and size of the socket address in addrlen.

Returns a **connected descriptor** that can be used to communicate with the client via Unix I/O routines.

Sockets Interface: connect

A client establishes a connection with a server by calling connect:

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

Attempts to establish a connection with server at socket address addr

If successful, then clientfd is now ready for reading and writing.

Resulting connection is characterized by socket pair

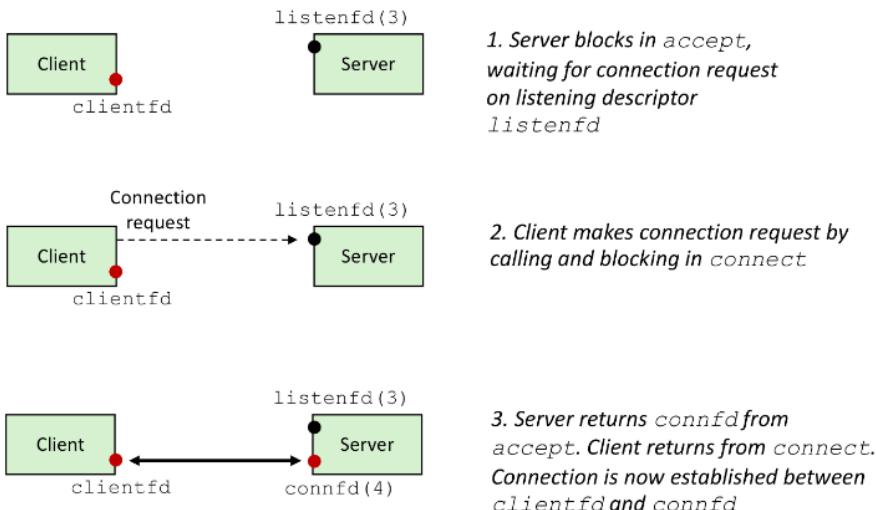
(x:y, addr.sin_addr:addr.sin_port)

x is client address

y is ephemeral port that uniquely identifies client process on client host

Best practice is to use getaddrinfo to supply the arguments addr and addrlen.

accept Illustrated



Connected vs. Listening Descriptors

Listening descriptor

End point for client connection requests
Created once and exists for lifetime of the server

Connected descriptor

End point of the connection between client and server
A new descriptor is created each time the server accepts a connection request from a client
Exists only as long as it takes to service client

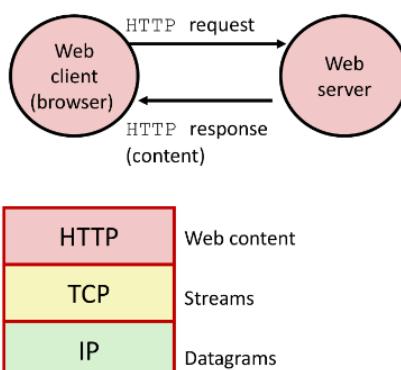
Why the distinction?

Allows for concurrent servers that can communicate over many client connections simultaneously
E.g., Each time we receive a new request, we fork a child to handle the request

Web Server Basics

Clients and servers communicate using the HyperText Transfer Protocol (HTTP)

Client and server establish TCP connection
Client requests content
Server responds with requested content
Client and server close connection (eventually)
Current version is HTTP/1.1
RFC 2616, June, 1999.



Takeaways: sockets

Take-Aways

Network as a strictly layered system: physical (ethernet), kernel (IP/TCP), applications

Protocol is a set of rules that governs how hosts and routers should cooperate when they transfer data from network to network via naming scheme and delivery mechanism.

Socket as communication abstraction. To the kernel, a socket is an endpoint of communication. To an application, a socket is a file descriptor that lets the application read/write from/to the network. Client connects to a server via a socket. Servers bind sockets to address:port, listen and accept incoming connections from clients. Thereafter, clients and servers can read and write.

I/O API: MPI

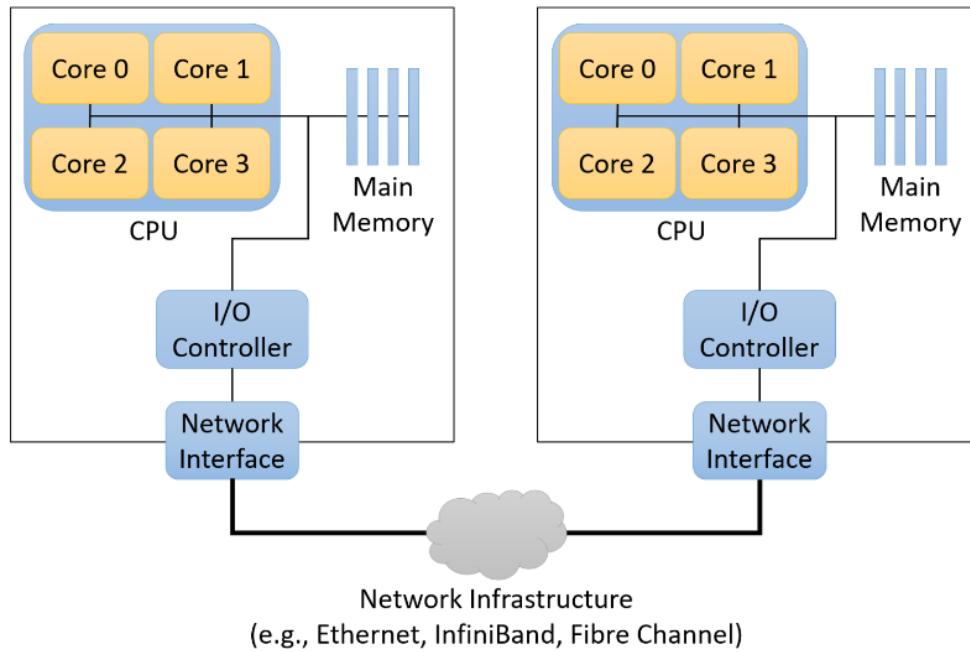


Figure 218. The major components of a shared-nothing distributed memory architecture built from two compute nodes

Parallel & Distributed Processing

models:

- client/server
- pipeline (each worker processes data independently)
- boss/worker (one worker process orchestrates)
- peer-to-peer (coordination/consensus algorithm needed)

communication: via. message-passing, following a protocol.

example paradigm: scatter/gather (example in book); boss scatters array across workers, gathers result from them.

15.2.4. MPI Hello World

As an introduction to MPI programming, consider the "hello world" program ([hello_world_mpi.c](#)) presented here:

```
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc, char **argv) {
    int rank, process_count;
    char hostname[1024];

    /* Initialize MPI. */
    MPI_Init(&argc, &argv);

    /* Determine how many processes there are and which one this is. */
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Determine the name of the machine this process is running on. */
    gethostname(hostname, 1024);

    /* Print a message, identifying the process and machine it comes from. */
    printf("Hello from %s process %d of %d\n", hostname, rank, process_count);

    /* Clean up. */
    MPI_Finalize();

    return 0;
}
```

When starting this program, MPI simultaneously executes multiple copies of it as independent processes across one or more computers. Each process makes calls to MPI to determine how many total processes are executing (with

Takeaways: MPI

Takeaways

- distributed processing is challenging
 - sharing
 - failure
 - races
 - consensus
- fortunately, frameworks (MPI) exist (based on decades of research) that make this easier.

Monitors

Reading

Operating system security

Operating system security - ch 1

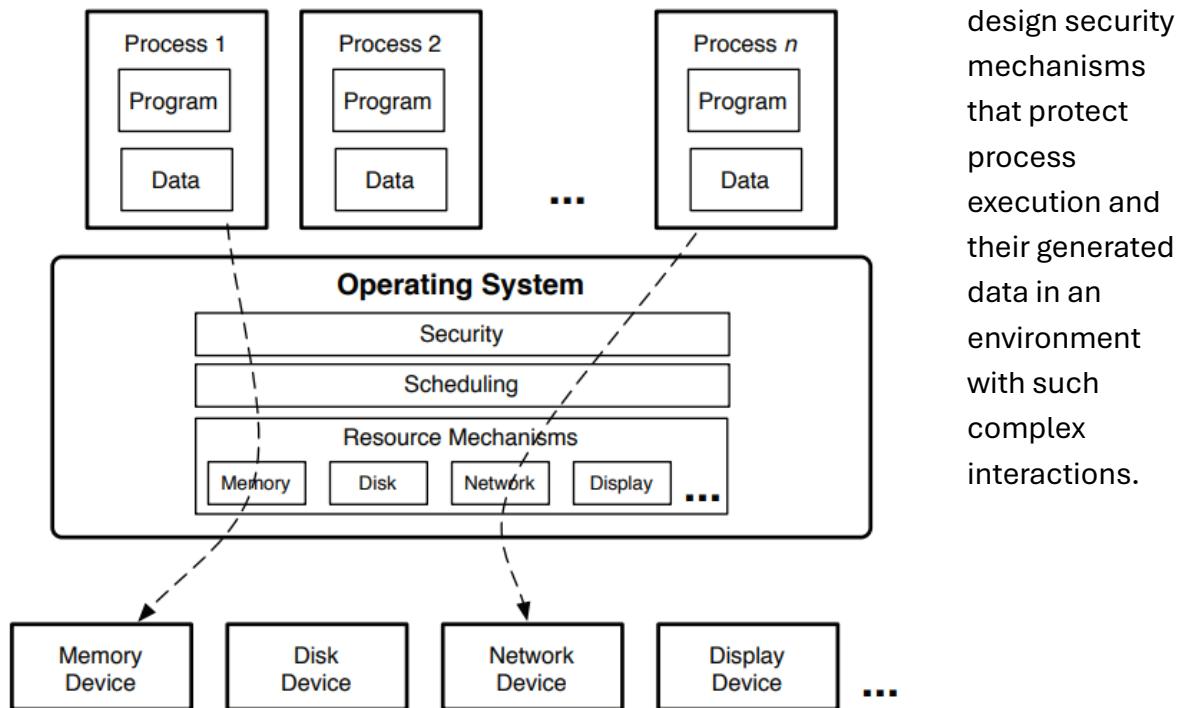


Figure 1.1: An operating system runs *security, scheduling, and resource mechanisms* to provide *processes* with access to the computer system's resources (e.g., CPU, memory, and devices).

the current state of operating systems security takes two forms:

- (1) constrained systems that can enforce security goals with a high degree of assurance and
- (2) general-purpose systems that can enforce limited security goals with a low to medium degree of assurance.

A secure operating system provides security mechanisms that ensure that the system's security goals are enforced despite the threats faced by the system.

- We aim for multi-fold

Security goals

System goals under a set of threats (threat model)

Satisfy: secrecy, integrity and availability

Subject: processes and users

Operations: read and write

Objects: files and sockets

Which subjects perform which operations on which objects

Simple-security property - bell-LaPadula model

- Process cannot read an object whose secrecy classification is higher than the process's

Principle of least privilege

- Process only has a set of operations necessary for its execution

Trust model

A system's trust model defines the set of software and data upon which the system depends for correct enforcement of system security goals. For an operating system, its trust model is synonymous with the system's trusted computing base (TCB).

Minimal amount of software

Must prove they have a viable trust model

- (1) the system TCB must mediate all security-sensitive operations
- (2) verification of the correctness of the TCB software and its data
- (3) verification that the software's execution cannot be tampered by processes outside the TCB.

Verifying is complex

Threat model

A threat model defines a set of operations that an attacker may use to compromise a system.

Assume the attacker will do everything possible

Access control

Operating system security - ch 2

An access enforcement mechanism authorizes requests (e.g., system calls) from multiple subjects (e.g., users, processes, etc.) to perform operations (e.g., read, write, etc.) on objects (e.g., files, sockets, etc.).

Protection system

Definition 2.1. A protection system consists of a protection state, which describes the operations that system subjects can perform on system objects, and a set of protection state operations, which enable modification of that state.

Lampsons access matrix

Definition 2.2. An access matrix consists of a set of subjects $s \in S$, a set of objects $o \in O$, a set of operations $op \in OP$, and a function $ops(s, o) \subseteq OP$, which determines the operations that subject s can perform on object o . The function $ops(s, o)$ is said to return a set of operations corresponding to cell (s, o) .

	File 1	File 2	File 3	Process 1	Process 2
Process 1	Read	Read, Write	Read, Write	Read	-
Process 2	-	Read	Read, Write	-	Read

Figure 2.1: Lampson's Access Matrix

Definition 2.3. A protection domain specifies the set of resources (objects) that a process can access and the operations that the process may use to access such resources.

One representation stores the protection state using individual object columns, describing which subjects have access to a particular object. This representation is called an access control list or ACL.

The other representation stores the other dimension of the access matrix, the subject rows. In this case, the objects that a particular subject can access are stored. This representation is called a capability list or C-List.

ACL approach, the set of subjects and the operations that they can perform are stored with the objects, making it easy to tell which subjects can access an object at any time.

C-Lists store the set of objects and operations that can be performed on them are stored with the subject, making it easy to identify a process's protection domain

Mandatory protection systems

Matrix problem: untrusted processes can tamper with the protection system using protection state operations.

A protection system that permits untrusted processes to modify the protection state is called a discretionary access control (DAC) system.

They must not be able to leak the file through the permissions available to them.

Matrix is not secure.

Protection systems that can enforce secrecy and integrity goals must enforce the requirement of security: *where a system's security mechanisms can enforce system security goals even when any of the software outside the trusted computing base may be malicious.*

Definition 2.4. A mandatory protection system is a protection system that can only be modified by trusted administrators via trusted software, consisting of the following state representations:

- A mandatory protection state is a protection state where subjects and objects are represented by labels where the state describes the operations that subject labels may take upon object labels;
- A labeling state for mapping processes and system resource objects to labels;
- A transition state that describes the legal ways that processes and system resource objects may be relabeled.

subjects and objects in an access matrix are represented by system-defined label

Labels are tamperproof because: (1) the set of labels is defined by trusted administrators using trusted software and (2) the set of labels is immutable.

mandatory access control (MAC)

- Trusted subjects define the access that subjects of particular labels can perform

A labeling state assigns labels to new subjects and objects.

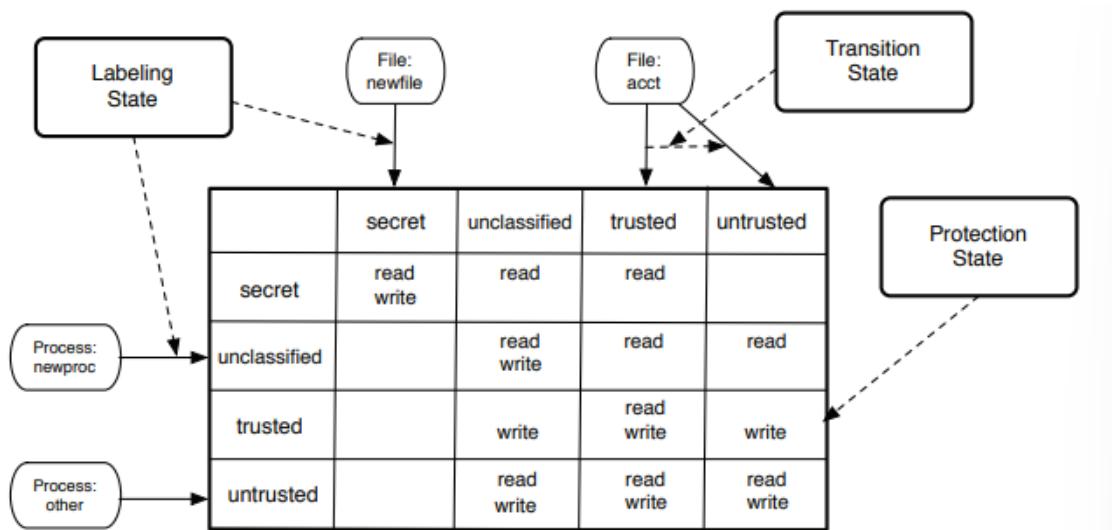


Figure 2.2: A Mandatory Protection System: The *protection state* is defined in terms of labels and is immutable. The immutable *labeling state* and *transition state* enable the definition and management of labels for system subjects and objects.

Reference monitor

Access enforcement mechanism

(1) its interface; (2) its authorization module; and (3) its policy store

Reference Monitor Interface The reference monitor interface defines where protection system queries are made to the reference monitor. In particular, it ensures that all security-sensitive operations are authorized by the access enforcement mechanism. By a security-sensitive operation, we mean an operation on a particular object (e.g., file, socket, etc.) whose execution may violate the system's security requirements.

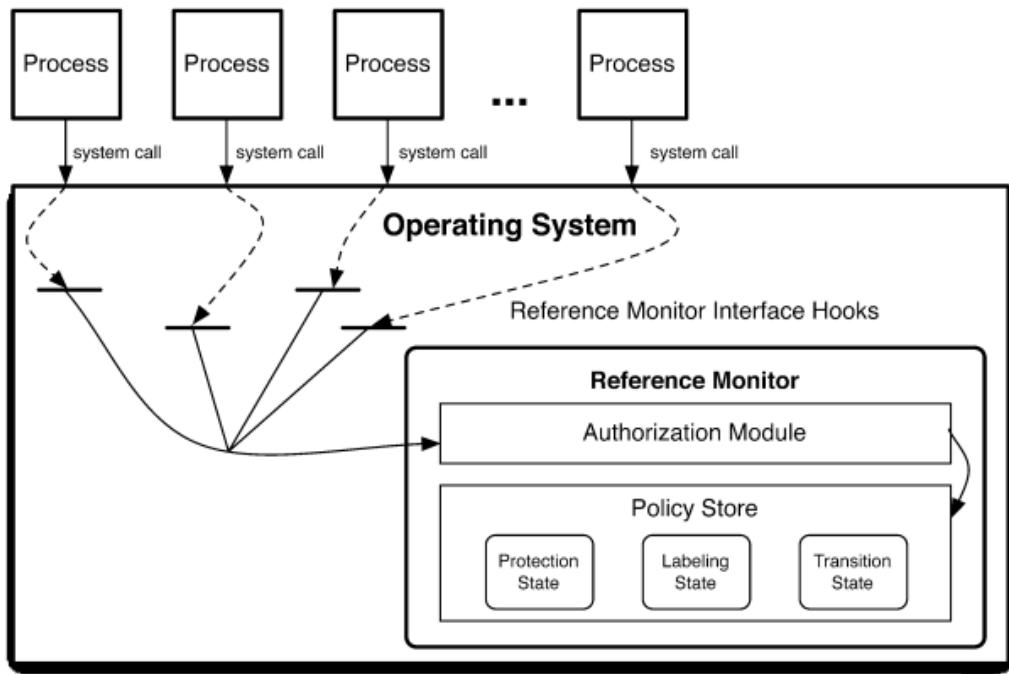


Figure 2.3: A *reference monitor* is a component that authorizes access requests at the *reference monitor interface* defined by individual *hooks* that invoke the reference monitor's *authorization module* to submit an authorization query to the *policy store*. The policy store answers authorization queries, labeling queries, and label transition queries using the corresponding states.

Interface: determines where enforcement is necessary and the information it needs to authorize

Authorization module: takes interfaces inputs and converts to a query for the policy store

- Map the process identity to a subject label
- Object references to an object label
- Determine operations to authorize

Policy store: database for states (protection, labeling and transition). Returns binary authorization reply.

Secure operating system definition

Definition 2.5. A secure operating system is an operating system where its access enforcement satisfies the reference monitor concept

Definition 2.6. The reference monitor concept defines the necessary and sufficient properties of any system that securely enforces a mandatory protection system, consisting of three guarantees:

1. Complete Mediation: The system ensures that its access enforcement mechanism mediates all security-sensitive operations.
2. Tamperproof: The system ensures that its access enforcement mechanism, including its protection system, cannot be modified by untrusted processes.
3. Verifiable: The access enforcement mechanism, including its protection system, “must be small enough to be subject to analysis and tests, the completeness of which can be assured”. That is, we must be able to prove that the system enforces its security goals correctly

Reference monitor defines the necessary and sufficient requirements for access control

Assessment criteria

1. Complete Mediation: How does the reference monitor interface ensure that all security sensitive operations are mediated correctly? In this answer, we describe how the system ensures that the subjects, objects, and operations being mediated are the ones that will be used in the security-sensitive operation.
2. Complete Mediation: Does the reference monitor interface mediate security-sensitive operations on all system resources? We describe how the mediation interface described above mediates all security-sensitive operations.
3. Complete Mediation: How do we verify that the reference monitor interface provides complete mediation? We describe any formal means for verifying the complete mediation described above.
4. Tamperproof: How does the system protect the reference monitor, including its protection system, from modification? In modern systems, the reference monitor and its protection system are protected by the operating system in which they run. The operating system must ensure that the reference monitor cannot be modified and the protection state can only be modified by trusted computing base processes.
5. Tamperproof: Does the system’s protection system protect the trusted computing base programs? The reference monitor’s tamperproofing depends on the integrity of the entire trusted computing base, so we examine how the trusted computing base is defined and protected.

6. Verifiable: What is basis for the correctness of the system's trusted computing base? We outline the approach that is used to justify the correctness of the implementation of all trusted computing base code.
7. Verifiable: Does the protection system enforce the system's security goals?
Finally, we examine how the system's policy correctly justifies the enforcement of the system's security goals. The security goals should be based on the models in Chapter 5, such that it is possible to test the access control policy formally.

Linux security

Operating system security - 4.2

Unix protection system

UNIX implements definition 2.1 and not definition 2.4

= DAC system

Nowadays, users run a variety of processes, some of which may be supplied by attackers and others may be vulnerable to compromise from attackers, so the user will have no guarantee that these processes will behave consistently with the user's security goals. As a result, a secure operating system cannot use discretionary access control to enforce user security goals.

A secure protection system requires a mandatory labeling state, so this is another reason that UNIX systems cannot satisfy the requirements of a secure operating system.

UNIX authorization

- Runs in kernel

Does not implement a reference monitor.

Does not meet any of the security criterias.

Rootkits -> difficult to detect an attacker.

TCB programs may be vulnerable to any input value supplied by an untrusted process, such as malicious input arguments.

Time-of-check-to-time-of-use attacks

Operating system security - ch 9

Linux

- The reference monitor interface must be truly generic, such that “using a different security model is merely matter of loading a different kernel module”
- The reference monitor interfaces must be “conceptually simple, minimally invasive, and efficient”
- Must support the POSIX.1e capabilities mechanism as an “optional security module”

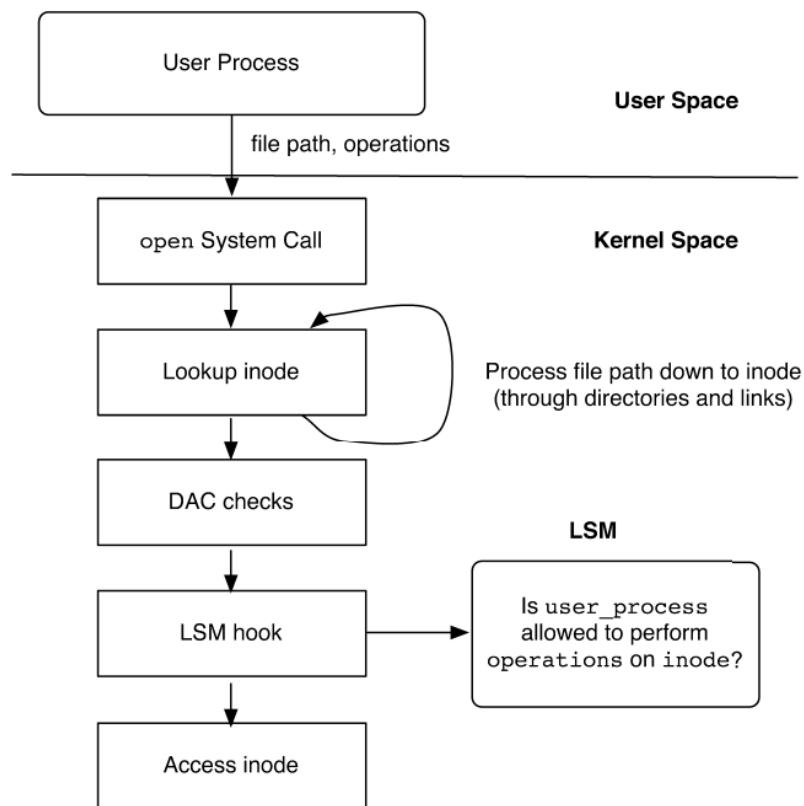


Figure 9.1: LSM Hook Architecture

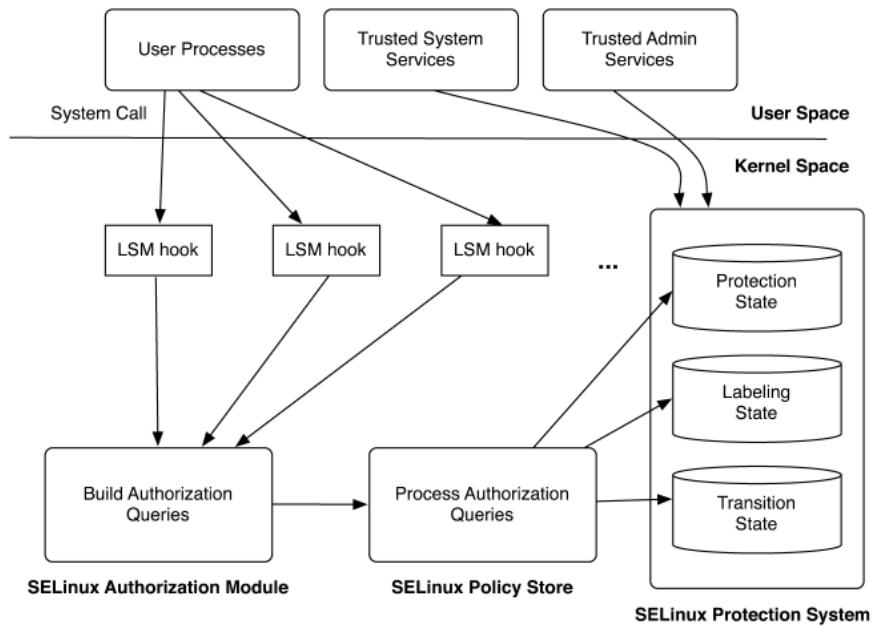


Figure 9.2: SELinux System Architecture

Slides

Bad Code

PENNSTATE

- Adversary may control the code that you run
- Examples
 - **Classical:** Viruses, Worms, Trojan horses, ...
 - **Modern:** Client-side scripts, Macro-viruses, Email, Ransomware, ...
- Easier to update/add software (malware) than ever
- **What are the problems with adversary code on your machine?**

Systems and Internets Infrastructure Security (SIS) Laboratory

Bad Code - Example



- You run an adversary-controlled program
 - What can an adversary do?
- Anything you can do
 - Do you have anything you would want to protect?
 - Secret data on your computer
 - Communications you make with your computer
- Well, at least these are only “user” processes
 - They do not *directly* compromise the host
 - Beware “local exploits”

Systems and Internet Infrastructure Security (SIS) Laboratory

Page 18

Bad Code - Defenses



- What can you do to **avoid executing adversary-controlled code**?
- Defenses
 - Only run “approved” code
 - How do you know?
 - Use automated installers or predefined images
 - Let someone else manage it
 - “Sandbox” code you are uncertain of
 - How do you do that?

Systems and Internet Infrastructure Security (SIS) Laboratory

Page 18

Good Code



- Fortunately, most code is not adversary controlled
 - I think...
- What is the problem with running code from **benign sources**?
 - Not really designed to defend itself from a determined, active adversary
- Functions performed by benign code may be exploited – i.e., have vulnerabilities

Systems and Internet Infrastructure Security (SIS) Laboratory

Page 20

Vulnerabilities

- A program **vulnerability** consists of three elements:
 - ▶ A flaw
 - ▶ Accessible to an adversary
 - ▶ Adversary has the capability to exploit the flaw
- Often focus on a subset of these elements
 - ▶ But all conditions must be present for a true vulnerability

Good Code – Goes Bad

- Classic flaw: **Buffer overflow**
- If adversary can access, exploits consist of two steps usually
 - ▶ (1) Gain control of execution – IP or stack pointer
 - ▶ (2) Choose code for performing exploitation
- Classic attack:
 - ▶ (1) Overwrite return address
 - ▶ (2) Write code onto stack and execute that

Good Code – Defenses

- Preventing either of these two steps prevents a vulnerability from being exploited
- How to prevent overwriting the return address?
 - ▶ ???
- How to prevent code injection onto the stack?
 - ▶ ???
- Are we done?
 - ▶ End the semester early...

Good Code – Evading Defenses



- Unfortunately, no
- (1) Adversaries gain access to the control flow in multiple ways
 - ▶ Function pointers, other variables, heap variables, etc.
 - ▶ Or evade defenses – e.g., disclosure attacks
- (2) Adversaries may perform desired operations without injecting code
 - ▶ Return-to-libc
 - ▶ Return-oriented attacks

Systems and Internet Infrastructure Security (SII) Laboratory

Page 24

Good Code – Confused Deputy



- And an adversary may accomplish her goals without any memory errors
 - ▶ Trick the program into performing the desired, malicious operations
- Example “confused deputy” attacks
 - ▶ SQL injection
 - ▶ Resource access attacks
 - ▶ Bypass attacks
 - ▶ Race condition attacks (TOCTTOU)

Systems and Internet Infrastructure Security (SII) Laboratory

Page 25

Result



- Adversaries have a variety of ways to try to get code under their control running on your computer
- Software defenses may not prevent exploitation
 - ▶ And still lots of room for improvement
- Malware and intrusion detection is a hard problem
 - ▶ How do we know whether code is bad or good?
- Systems security is about blocking damage or limiting damage from adversary-controlled execution
 - ▶ Not doing well enough yet

Control Bad Code

- What mechanism does an OS use to restrict the rights of processes (i.e., running code) from system resources?

Access Control

- System makes a decision to grant or reject an access request
 - ▶ from an already authenticated subject
 - ▶ based on what the subject is authorized to access

- Access request
 - ▶ Object: System resource
 - ▶ Operations: One or more actions to be taken
 - ▶ Subject: Process that initiated the request
- Access Control Mechanisms enforce Access Control Policies to make such decisions

Access Matrix

PENNSTATE

- Lampson formalizes the model of access control in his 1970 paper "Protection"
- Called **Access Matrix**
 - ▶ Rows are **subjects**
 - ▶ Columns are **objects**
 - ▶ Authorized **operations** listed in cells
- To determine if S_i has right to access object O_j , compare the request ops to the appropriate cell

	O	O	O
S	Y	Y	N
S	N	Y	N
S	N	Y	Y

UNIX Access Control

- On Files
 - ▶ All objects are files
 - ▶ Not exactly true
- Classical Protection System
 - ▶ Limited access matrix
 - ▶ Discretionary protection state operations
- Practical model for end users
 - ▶ Still involves some policy specification

UNIX Mode Bits

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

Access Matrix

PENNSTATE


- Using the Access Matrix
- (1) Suppose J wants to prevent other users' processes from reading/writing her **private key** (object O₁)
- (2) Suppose J wants to prevent other users' processes from writing her **public key** (object O₂)
- Design the access matrix
- Are these the rights on your host to your SSH public and private keys?

	o	o	o
J	?	?	?
S	?	?	?
S	?	?	?

Access Matrix

PENNSTATE


- Using the Access Matrix
- (1) Suppose J wants to protect a **private key** (object O₁) from being leaked to or modified by others
- (2) Suppose J wants to prevent a **public key** (object O₂) from being modified by others
- Design the access matrix
- Will this access matrix protect the keys' secrecy and integrity?

	o	o	o
J	?	?	?
S	?	?	?
S	?	?	?

Consider Bad Code Again

- Claim: Any code you run may be able to compromise either of the key files
- For the private key
 - ▶ Any process running under your user id can read and leak your private key file
- For the public key
 - ▶ Any process running under your user id may modify the public key file

- Often people make the public key file read-only even to the owner
- Is that enough?
 - No. Processes running on behalf of the owner may change perms

Bad Code - Examples

- Suppose you download and run adversary-controlled code (e.g., Trojan horse)
 - ▶ It will run with all your permissions
 - ▶ Even can modify the permissions of any files you own
- Suppose you run benign code that is compromised by an adversary – becoming bad
 - ▶ Is effectively the same as above if adversary can choose code to execute (e.g., return-oriented attack)
 - ▶ Adversaries can also trick victims into performing operations on their behalf (e.g., confused deputy attack)

Protection vs. Security

- Protection
 - ▶ Secrecy and integrity met under benign processes
 - ▶ Protects against an error by a non-malicious entity
- Security
 - ▶ Security goals met under potentially malicious processes
 - ▶ Enforces requirements even if adversary is in complete control of the process
- Hence, for J: Non-malicious process shouldn't leak the private key by accident to a specific file owned by others
- A potentially malicious process may contain a Trojan horse that can write the private key to files chosen by adversaries

Fundamentally Flawed

- Conventional operating system mechanisms enforce protection rather than security

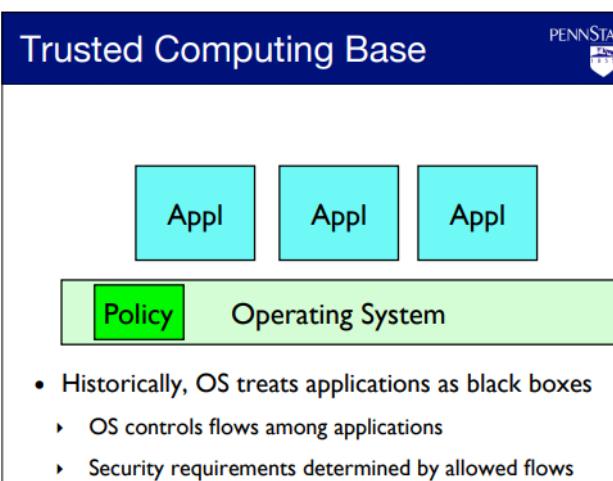
- ▶ Protection is fundamentally incapable of defending from an active and determined adversary

Secrecy

- Process secrecy requires that the process not communicate with unauthorized parties
 - ▶ But what about a process that services requests?
 - ▶ This is a very difficult requirement to meet
- Suppose a benign process can write to a file controlled by an adversary
- Unless the process is trusted to contain no vulnerabilities then the process could be compromised (is potentially malicious)

Integrity

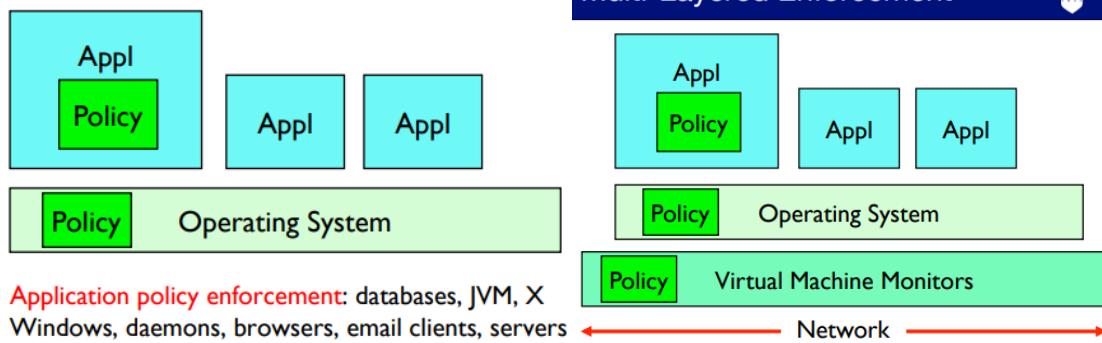
- Process integrity requires that the process not depend on adversary input
 - ▶ What does “depend on” mean?
 - ▶ This is a very difficult requirement to meet
- Suppose a benign process can read from a file controlled by an adversary
- Unless the process is trusted to contain no vulnerabilities then the process could be compromised (is potentially malicious)



Policy Enforcement in Apps



Multi-Layered Enforcement



Security Enforcement

- Application policy enforcement: databases, JVM, X Windows, daemons, browsers, email clients, servers
- Several applications include access control
 - ▶ Databases, window servers, web servers, browsers, ...
- Some programming systems include access control to system resources
 - ▶ Java, Safe-Tcl, Ruby, Python, Perl – Jif, Flow Caml (information flow);
- Some systems recognize that programs may contribute to access control
 - ▶ User-level policy server for SELinux
 - ▶ Information Flow Control
- Requirement: Ensure that all layers are using their authority in a manner consistent with system security goals

Questions for This Class

- How do we keep bad code off our systems?
- How do we keep benign code from becoming bad code?
- How do we prevent benign code from being tricked into being a confused deputy?
- How do we restrict code that may be/go bad from propagating damage?
- How can we leverage the myriad of system defenses to control code efficiently?
- How do we know what we configured works?

Take Away

- Traditional OS access control
 - ▶ Is for protection, not security

- So it cannot confine an active adversary
 - ▶ Build attacks that work despite access control
 - ▶ They can change the access control policies
- Access control is enforced in many places now
 - ▶ Can we utilize them comprehensively and efficiently?

Protection System

- Manages the authorization policy for a system
 - ▶ It describes what operations each subject (via their processes) can perform on each object
- Consists of
 - ▶ State: Protection state
 - ▶ State Ops: Protection state operations

Access Matrix Protection System

- Protection State
 - ▶ Current state of matrix
- Can modify the protection state
 - ▶ Via protection state operations
 - ▶ E.g., can create objects
 - ▶ E.g., owner can add a subject, operation mapping for their objects
- Lampson's "Protection" paper
 - ▶ Can even delegate authority to perform protection state ops

Protection System

- Why is Protection State insufficient to enforce security?
- Goal: a protection state in which we can determine whether an unauthorized operation will ever be allowed (Safety)

Protection System Problems

- Protection system approach is inadequate for security
 - ▶ Suppose a process runs bad code
- Processes can change their own permissions
 - ▶ Processes may become untrusted, but can modify policy
- Processes, files, etc. are created and modified
 - ▶ Cannot predict in advance (safety problem)
- What do we need to achieve necessary controls?

Define and Enforce Goals

- Claim: If we can define and enforce a security policy that ensures security goals, then we can prevent such attacks
- How do we know what policy will be enforced?
- How do we know the enforcement mechanism will enforce policy as expected?
 - ▶ Look into this today
- How do we know the policy expresses effective goals?
 - ▶ Will look into this in depth later

Mandatory Protection System

- Is a protection system that can be modified only by trusted administration that consists of
 - ▶ A mandatory protection state where the protection state is defined in terms of an immutable set of labels and the operations that subject labels can perform on object labels
 - ▶ A labeling state that assigns system subjects and objects to those labels in the mandatory protection state
 - ▶ A transition state that determines the legal ways that subjects and objects may be relabeled
- MPS is immutable to user-space process

Mandatory Protection State

- Immutable table of
 - ▶ Subject labels
 - ▶ Object labels
 - ▶ Operations authorized for former upon latter
- How can you use an MPS to control use of bad code?
 - ▶ E.g., Prevent modification of kernel memory?

Mandatory Protection State

- Immutable table of
 - ▶ Subject labels
 - ▶ Object labels
 - ▶ Operations authorized for former upon latter
- How can you use an MPS to control use of bad code?
 - ▶ E.g., Prevent modification of kernel memory?
 - ▶ Subject labels for all subjects running “bad code” are not allowed modify kernel memory
 - Or that may run “bad code” (be compromised)
 - ▶ How do subjects (processes) get their labels?

Labeling State

- Immutable rules mapping
 - ▶ Subjects to labels (in rows)
 - ▶ Objects to labels (in columns)
- How can you use labeling state to control bad code?
 - ▶ E.g., Prevent modification of kernel memory?
 - ▶ Assign all processes that may run bad code ...
 - ▶ With a label that cannot modify kernel memory

- ▶ What about objects created by these processes?

Protecting Good Code

- How can you use labeling state to prevent good code from going bad?

Protecting Good Code

- How can you use labeling state to prevent good code from going bad?
 - ▶ E.g., Prevent dependence on untrusted input?
 - ▶ Assign object labels to all objects that may be adversary controlled
 - ▶ Do not grant subject labels that should run good code access to those labels
 - ▶ Verify that you are running good code (how?) and assign to one of these protected subject labels
 - ▶ What integrity model does this approximate?

Protecting Good Code

- What if good code needs to access some adversary-controlled resources?

Mandatory Protection State

- What if good code needs to access some adversary controlled resources?
 - ▶ (1) if a process reads adversary-controlled object label, remove privileged permissions (e.g., to modify kernel memory)
 - ▶ (2) if a process reads adversary-controlled object label, remove permission to write to any object that may be accessed by a subject whose label grants privileged permissions
- How do we achieve this change with the MPS?

Transition State

- Immutable rules mapping
 - ▶ Subject labels to conditions that change their subject labels

- ▶ Object labels to conditions that change their object labels
- How can you use labeling state to control bad code?
 - ▶ E.g., Achieve (1) and (2)
 - ▶ Change subject label of subject accessing adversary controlled resources to remove these permissions
 - ▶ What integrity model does this approximate?

Transition State

- Is it possible to launch processes with more permissions than the invoker with MPS?

Managing MPS

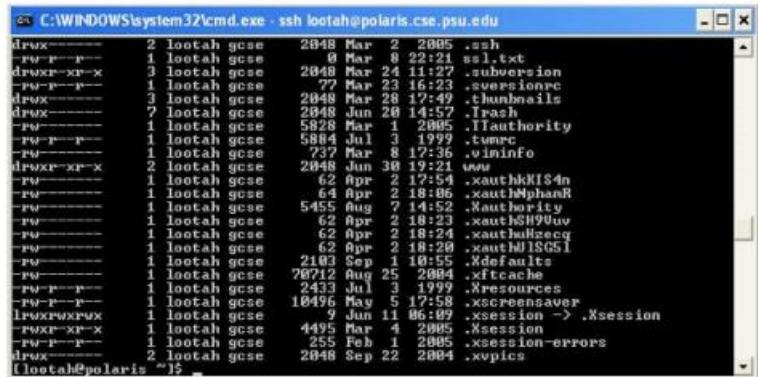
- Challenge
 - ▶ Determining how to set and manage an MPS in a complex system involving several parties
- Parties
 - ▶ What does programmer know about deploying their program securely?
 - ▶ What does an OS distributor know about running a program in the context of their system?
 - ▶ What does an administrator know about programs and OS?
 - ▶ Users?

Managing MPS

- Current methods use dynamic analysis to setup MAC policies – run the program and collect the permissions used
 - ▶ Really a functional policy

Linux Authorization circa 2000

- Linux implements discretionary access control



A screenshot of a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe - ssh lootah@polaris.cse.psu.edu'. The window displays a file listing from a Linux system. The files listed include .ssh, .subversion, .thumbnails, .Trash, .tuncrc, .viminfo, .www, .xauthhkIS4n, .xauthNphanR, .xauthority, .xauthSH9uv, .xauthulfzecq, .xauthW1SG1, .xfccache, .xresources, .xscreensaver, .xsession, and .xsession-errors. The files are owned by 'lootah' and have various permissions like drwxr-xr-x, -rw-r--r--, and -rwxr--r--.

Linux Security circa 2000



- Patches to the Linux kernel
 - Enforce different access control policy
 - Restrict root processes
 - Some hardening
- Argus PitBull
 - Limited permissions for root services
- RSBAC
 - MAC enforcement and virus scanning
- grsecurity
 - RBAC MAC system
 - Auditing, buffer overflow prevention, /tmp race protection, etc
- LIDS
 - MAC system for root confinement

What is the right solution?

The Answer

- The solution to all computer science problems
- Add another layer of indirection

Linux Security Modules Was Born



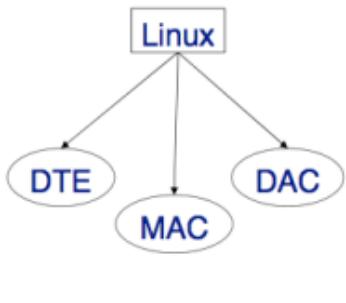
- “to allow Linux to support a variety of security models, so that security developers don't have to have the ‘my dog's bigger than your dog’ argument, and users can choose the security model that suits their needs.”, Crispin Cowan

= <http://mail.wirex.com/pipermail/linux-security-module/2001-April/0005.html>

Linux Before and After

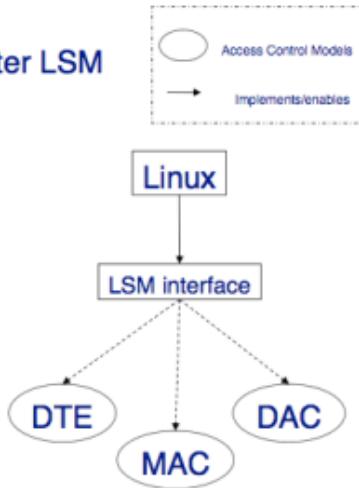


Before LSM



Access control models implemented as
Kernel patches

After LSM



Access control models implemented as
Loadable Kernel Modules

LSM Requirements



- LSM needs to reach a balance between kernel developer and security developers requirements.
LSM needs to unify the functional needs of as many security projects as possible, while minimizing the impact on the Linux kernel.
 - Truly generic
 - conceptually simple
 - minimally invasive
 - Efficient
 - Support for POSIX capabilities
 - Support the implementation of as many access control models as Loadable Kernel Modules

LSM – A Reference Monitor

- To enforce mandatory access control
 - ▶ We need to develop an authorization mechanism that satisfies the reference monitor concept
- How do we do that?
 - ▶ And satisfy all the other goals?

LSM – Complete Mediation

- First requirement is complete mediation

- Add security hooks to mediate various operations in the kernel
 - ▶ These hooks invoke functions defined by the chosen module
- These hooks construct “authorization queries” that are passed to the module
 - ▶ Subject, Object, Operations

LSM Hooks

PENN

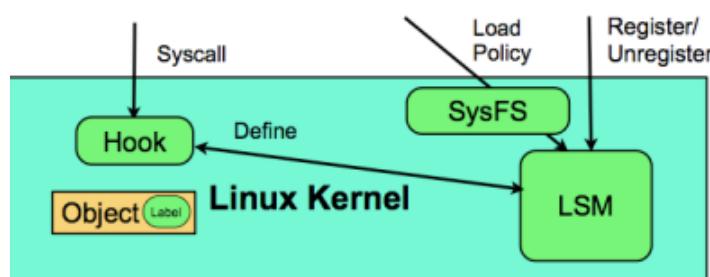
- Function calls that can be overridden by security modules to manage security fields and mediate access to Kernel objects.
- Hooks called via function pointers stored in `security->ops` table
- Hooks are primarily “restrictive”

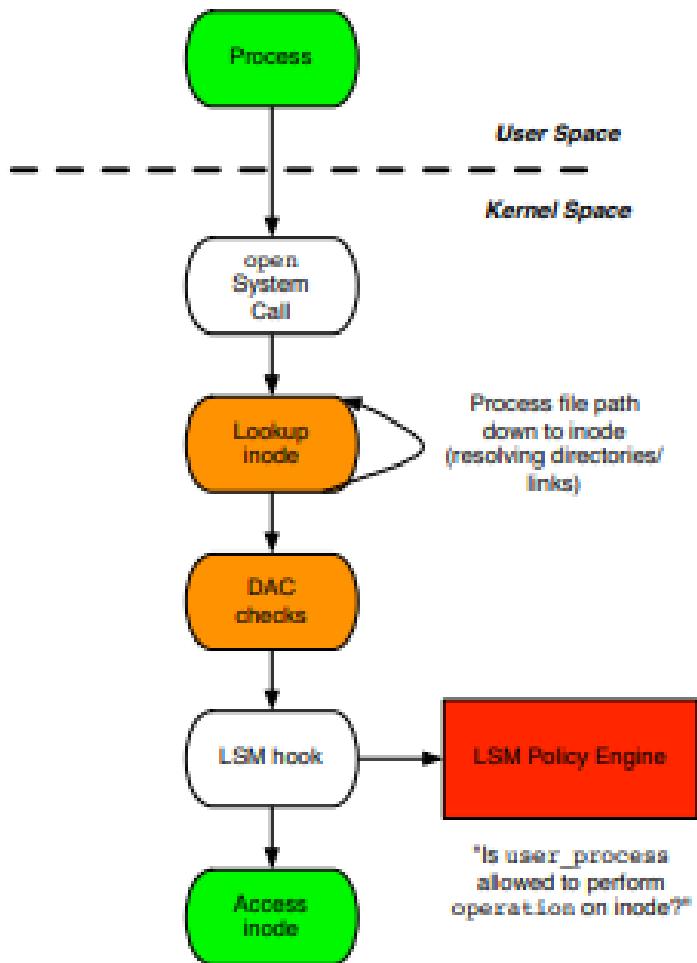
LSM Hooks

PENN



LSM Hook Architecture





LSM – Complete Mediation

- First requirement is complete mediation
- Enables authorization by module
- Linux extends “sensitive data types” with opaque security fields
 - ▶ Modules manage these fields – e.g., store security labels
- Which Linux data types are sensitive?

- Enable security modules to associate security information to Kernel objects
- Implemented as void* pointers
- Completely managed by security modules
- What to do about object created before the security module is loaded?

LSM – Complete Mediation

- First requirement is complete mediation
- How do we know LSM implements complete mediation?
- Asked one of the lead developers (Cowan)
 - ▶ His reply?
 - “We don’t”

LSM – Tamperproof

- Second requirement is tamperproof
- Prevent adversaries from modifying the reference monitor code or data
- How is LSM code protected?
- How is LSM data protected?

LSM – Tamperproof

- Second requirement is tamperproof
- Add functions to register and unregister Linus Security Modules
 - ▶ Implemented as a set of function pointers defined at registration time
- LSM module defines code
- LSM function pointers define targets of hooks
- ▶ These are data – modifiable

- Implications?

LSM – Tamperproof

- Adversaries could modify the code executed by Linux by modifying these function pointer data values

- ▶ Some people opposed this idea and refused to participate
- ▶ Eventually changed to require compiled-in LSM modules

LSM Tasks



- Linux Kernel modified in 5 ways:
 - Opaque security fields added to certain kernel data structures
 - Security hook function calls inserted at various points with the kernel code
 - A generic security system call added
 - Function to allow modules to register and unregister as security modules
 - Move capabilities logic into an optional security module

Hook Details



- Difference from discretionary controls
- More object types
 - 29 different object types
 - Per packet, superblock, shared memory, processes
 - Different types of files
- Finer-grained operations
 - File: ioctl, create, setattr, setattr, lock, append, unlink,
- System labeling
 - Not dependent on user
- Authorization and policy defined by module
 - Not defined by the kernel

LSM Performance

PENNSTATE

- Microbenchmark: LMBench
 - Compare standard Linux Kernel 2.5.15 with Linux Kernel with LSM patch and a default capabilities module
 - Worst case overhead is 5.1%
- Macrobenchmark: Kernel Compilation
 - Worst case 0.3%
- Macrobenchmark: Webstone
 - With Netfilter hooks 5-7%
 - Uni-Processor 16%
 - SMP 21% overhead

LSM Use

- Available in Linux 2.6
 - Packet-level controls upstreamed in 2.6.16
- Modules
 - POSIX Capabilities module
 - SELinux module
 - Domain and Type Enforcement
 - Openwall, includes grsecurity function
 - LIDS
 - AppArmor
- Not everyone is in favor
 - How does LSM impact system hardening?

Take Away

- Aiming for mandatory controls in Linux
 - ▶ But everyone had their own approach
- Linux Security Modules is a general interface for any* authorization module
 - ▶ Much finer controls – interface is union of what everyone can do
- What does this effort say about
 - Achieving complete mediation?
 - Whether complete mediation should be policy-dependent?

Reference Monitor for Linux

- LSM provides a reference monitor interface for Linux
 - ▶ Complete Mediation

- You need a module and infrastructure to achieve the other two goals
 - ▶ Tamperproofing
 - ▶ Verifiability
- SELinux is a comprehensive reference validation mechanism aiming at reference monitor guarantees

SELinux History

- Origins go back to the Mach microkernel retrofitting projects of the 1980s
 - ▶ DTMach (1992)
 - ▶ DTOS (USENIX Security 1995)
 - ▶ Flask (USENIX Security 1999)
 - ▶ SELinux (2000-...)
- Motivated by the security kernel design philosophy
 - ▶ But, practical considerations were made

The Rest of the SELinux Story

- Tamperproof
 - ▶ Protect the kernel
 - ▶ Protect the trusted computing base
 - ▶ Use MPS to provide tamperproofing of TCB?
- Verifiability
 - ▶ Code correctness
 - ▶ Policy satisfy a security goal
 - ▶ Use MPS to express secrecy and integrity requirements

Design MPS

- Do not believe that classical integrity is achievable in practice
 - ▶ Too many exceptions

- ▶ Commercial systems will not accept constraints of classical integrity
- Instead, focus on providing comprehensive control of access aiming for
 - ▶ Confining root processes (tamperproof)
 - ▶ Least privilege in general (verifiability)
- How does ‘least privilege’ affect security?

SELinux Policy Rules

- SELinux Rules express an MPS
 - ▶ Protection state – ALLOW subject-label object-label ops
 - ▶ Labeling state – TYPE_TRANSITION subject-label objectlabel new-label (at create – objects)
 - Default is to label to same state as creator
 - ▶ Transition state – TYPE_TRANSITION subject-label objectlabel new-label (at exec – processes)
- Tens of thousands of rules are necessary for a standard Linux distribution
 - ▶ Protect system processes from user processes
 - ▶ User data can be protected by MLS

SELinux Transition State

- For user to run passwd program
 - ▶ Only passwd should have permission to modify /etc/shadow
- Need permission to execute the passwd program
 - ▶ allow user_t passwd_exec_t:file execute (user can exec /usr/bin/passwd)
 - ▶ allow user_t passwd_t:process transition (user gets passwd perms)
- Must transition to passwd_t from user_t
 - ▶ allow passwd_t passwd_exec_t:file entrypoint (run w/ passwd perms)
 - ▶ type_transition user_t passwd_exec_t:process passwd_t
- Passwd can perform the operation

- ▶ allow passwd_t shadow_t:file {read write} (can edit passwd file)

SELinux Deployment

- You've configured your SELinux policy
 - ▶ Now what is left?
- Surprisingly, a lot
 - ▶ Many services must be aware of SELinux
 - ▶ Got to get the policy installed in the kernel
 - ▶ Got to manage all this policy
- And then there is the question of getting the policy to do what you want

User-space Services

- What kind of security decisions are made by user-space services?
 - ▶ Authentication (e.g., sshd)
 - ▶ Access control (e.g., X windows, DBs (servers), browsers (middleware), etc.)
 - ▶ Configuration (e.g., policy build and installation)
- Also, many services need to be aware of SELinux to enable usability
 - ▶ E.g., Listing files/processes with SELinux contexts (ls/ps)

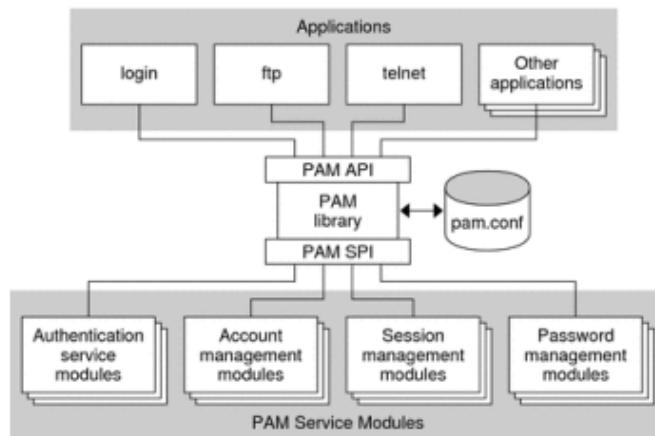
User-space Services

- Authentication
 - ▶ Various authentication services need to create a "SELinux subject context" on a user login
 - ▶ Like login in general, except we set an SELinux context and a UID for the generated shell
- How do you get all these ad hoc authentication services to interact with SELinux?

Authentication for SELinux

- **Pluggable Authentication Modules**

- ▶ There is a module for SELinux that various authentication services use to create a subject context



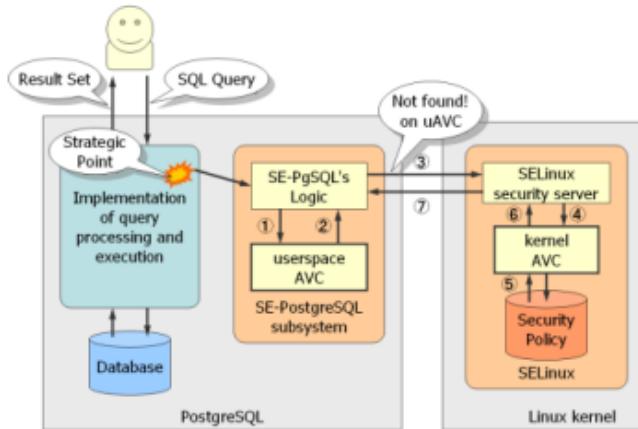
User-space Services

- Access Control
 - ▶ Many user-space services are shared among mutually untrusting clients
 - Problem: service may leak one client's secret to another
 - If your SELinux policy allows multiple, mutually untrusting clients to talk to the same service, what can SELinux do to prevent exploits?

User-space Services

PENN

- Add SELinux support to the service
 - X Windows, postgres, dbus, gconf, telephony server
- E.g., Postgres with the SELinux user-space library



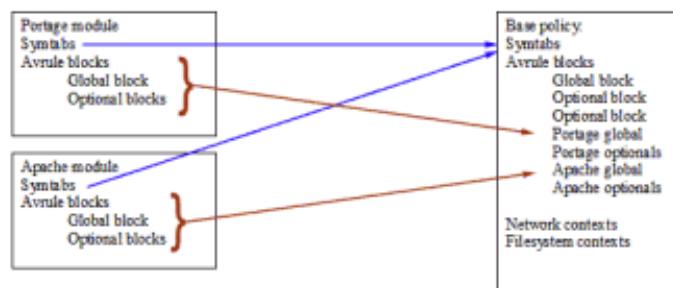
User-space Services

- Configuration
 - You need to get the SELinux policy constructed and loaded into the kernel
 - Without allowing attacker to control the system policy
 - And policy can change dynamically
- How to compose policies?
- How to install policies?

Compose Policies



- The SELinux policy is modular
 - Although not in a pure, object-oriented sense
 - Too much had been done
- Policy management system composes the policy from modules, linking a module to previous definitions and loads them



Take Away

- Problem: Turn the SELinux policy into a working, usable reference monitor
 - Work with user-space services
 - Design the policy that you want
- There are many requirements for user-space services to provide authentication, access control, and policy configuration itself
 - PAM, Policy Mgmt, User-space access, Network support
- Design of MPS can only be semi-automated
 - Prevent network threats and design for app integrity

Containment

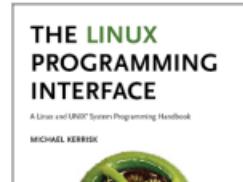
Introduction

Why is this interesting?

- User namespaces are cornerstone of unprivileged containers
 - But also many other Linux tools
 - Flatpak / Snap
 - Firejail
 - Modern browser sandboxes
 - Etc.

Who?

- Linux *man-pages* project
 - <https://www.kernel.org/doc/man-pages/>
 - Approx. 1060 pages documenting syscalls and C library
 - Contributor since 2000
 - Maintainer 2004-2020
 - Comaintainer 2020-2021
- I wrote a book
- Trainer/writer/engineer
<http://man7.org/training/>
- mtk@man7.org, @mkerrisk



Time is short

- Normally, I would spend several hours on this topic
- Many details left out, but I hope to convey the big picture
- We'll go fast

Namespaces

Namespaces

- Before looking specifically at user namespaces, what is a namespace (NS) more generally?
- A namespace "wraps" some global system resource to provide resource isolation
- Linux supports multiple NS types
 - Eight currently, and counting...

Each NS isolates some kind of resource(s)

- Each NS type isolates some kind of resource(s):
 - **UTS** NSs: isolate system identifiers (e.g., hostname)
 - **Mount** NSs: isolate mount point list
 - **IPC** NSs: isolate interprocess communication resources
 - **PID** NSs: isolate PID number space
 - **Network** NSs: isolate NW resources
 - Firewall & routing rules, socket port numbers, `/proc/net`, `/sys/class/net`, ...
 - And so on....

Namespaces

- For each NS type:
 - Multiple **instances** of NS may exist on a system
 - At system boot, there is one instance of each NS type—the **initial namespace**
 - A process resides in one NS instance (of each of NS types)
 - To processes inside NS instance, it appears that only they can see/modify corresponding global resource
 - (They are unaware of other instances of resource)
- This is a bit abstract so far; let's look at concrete example...

Example: UTS namespaces

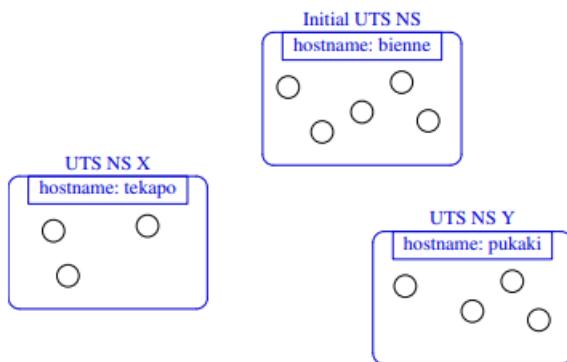
UTS namespaces

- UTS NSs are simple, and so provide an easy example
- Isolate two system identifiers returned by `uname(2)`
 - **nodename**: system hostname (set by `sethostname(2)`)
 - **domainname**: NIS domain name (set by `setdomainname(2)`)
- Container configuration scripts might tailor their actions based on these IDs
 - E.g., nodename could be used with DHCP, to obtain IP address for container
- “UTS” comes from `struct utsname` argument of `uname(2)`
 - Structure name derives from “UNIX Timesharing System”

UTS namespaces

- Running system may have multiple UTS NS instances
- Processes within single instance access (get/set) same *nodename* and *domainname*
- Each NS instance has its own *nodename* and *domainname*
 - Changes to *nodename* and *domainname* in one NS instance are invisible to other instances

UTS namespace instances



Each UTS NS contains a set of processes (the circles) which see/modify same hostname (and domain name, not shown)

Commands

Some “magic” symlinks

- Each process has some symlink files in `/proc/PID/ns`

```
/proc/PID/ns/cgroup      # Cgroup NS instance
/proc/PID/ns/ipc          # IPC NS instance
/proc/PID/ns/mnt          # Mount NS instance
/proc/PID/ns/net          # Network NS instance
/proc/PID/ns/pid          # PID NS instance
/proc/PID/ns/time         # Time NS instance
/proc/PID/ns/user         # User NS instance
/proc/PID/ns/uts          # UTS NS instance
```

- One symlink for each of the NS types

Some “magic” symlinks

- Target of symlink tells us which NS instance process is in:

```
$ readlink /proc/$$/ns/uts  
uts:[4026531838]
```

- Content has form: *ns-type : [magic-inode-#]*
 - (*inode-#* comes from internally mounted NS filesystem)

- Various uses for these symlinks, including:

- If processes show same symlink target, they are in same NS

The *unshare(1)* and *nsenter(1)* commands

There are shell commands for working with namespaces...

- *unshare(1)* creates new NSs and executes a command in those NSs:

```
unshare [options] [command [arg...]]
```

- *command* defaults to *sh*

- *nsenter(1)* steps into already existing NS(s) and executes a command:

```
nsenter [options] [command [arg...]]
```

- *command* defaults to *sh*

The *unshare(1)* and *nsenter(1)* commands

unshare(1) and *nsenter(1)* have options for specifying NS types:

```
unshare [options] [command [arguments]]  
-C Create new cgroup NS  
-i Create new IPC NS  
-m Create new mount NS  
-n Create new network NS  
-p Create new PID NS  
-T Create new time NS  
-u Create new UTS NS  
-U Create new user NS
```

```
nsenter [options] [command [arguments]]  
-t PID PID of process whose NSs should be entered  
-C Enter cgroup NS of target process  
-i Enter IPC NS of target process  
-m Enter mount NS of target process  
-n Enter network NS of target process  
-p Enter PID NS of target process  
-T Enter time NS of target process  
-u Enter UTS NS of target process  
-U Enter user NS of target process  
-a Enter all NSs of target process
```

Demonstration

Demo

- Start two terminal windows (*sh1*, *sh2*) in initial UTS NS

```
sh1$ hostname      # Show hostname in initial UTS NS  
bienne
```

```
sh2$ hostname  
bienne
```

- In *sh2*, create new UTS NS, and change hostname

```
$ SUDO_PS1='sh2# ' sudo unshare -u bash --norc  
sh2# hostname langwied      # Change hostname  
sh2# hostname              # Verify change  
langwied
```

- sudo(8)* because we need privilege (**CAP_SYS_ADMIN**) to create a UTS NS

 *man7.ora*
• We set `SUDO_PS1` so shell has a distinctive prompt. Setting this environment variable causes *sudo(8)* to set `PS1` for the command that it executes. (`PS1` defines the prompt displayed by the shell.) The `bash --norc` option prevents the execution of shell start-up scripts that might modify `PS1`.

- In *sh1*, verify that hostname is unchanged:

```
sh1$ hostname  
bienne
```

- Compare `/proc/PID/ns/uts` symlinks in two shells

```
sh1$ readlink /proc/$$/ns/uts  
uts:[4026531838]
```

```
sh2# readlink /proc/$$/ns/uts  
uts:[4026532855]
```

- The two shells are in different UTS NSs

- Discover the PID of *sh2*:

```
sh2# echo $$  
5912
```

- From *sh1*, use *nsenter(1)* to create a new shell that is in same NS as *sh2*:

```
sh1$ SUDO_PS1='sh3# ' sudo nsenter -t 5912 -u  
sh3# hostname  
langwied  
sh3# readlink /proc/$$/ns/uts  
uts:[4026532855]
```

- Comparing the symlink values, we can see that this shell (*sh3#*) is in the second (*sh2#*) UTS NS

Capabilities

(Traditional) superuser and set-UID-*root* programs

- We need a brief understanding of capabilities...
- Traditional UNIX privilege model divides users into two groups:
 - **Normal users**, subject to privilege checking based on UIDs and GIDs
 - **Superuser** (UID 0) bypasses many of those checks
- Traditional mechanism for giving privilege to unprivileged users is **set-UID-*root* program**

```
# chown root prog  
# chmod u+s prog
```

- When executed, **process assumes UID of file owner**
 - ⇒ process gains privileges of superuser
- Powerful... but dangerous



The traditional privilege model is a problem

- Coarse granularity of traditional privilege model is a problem:
 - E.g., say we want to give a program the power to change system time
 - Must also give it power to do **everything else** *root* can do
 - ⇒ **No limit on possible damage** if program is compromised
- **Capabilities** are an attempt to solve this problem

Background: capabilities

- Capabilities: **divide power of superuser into small pieces**
 - 41 capabilities as at Linux 6.4 (see [capabilities\(7\)](#))
- Examples:
 - **CAP_DAC_OVERRIDE**: bypass all file permission checks
 - **CAP_SYS_ADMIN**: do (too) many different sysadmin tasks
 - **CAP_SYS_TIME**: change system time
- Instead of set-UID-*root* programs, have programs with one/a few attached capabilities
 - Attached using [setcap\(8\)](#)
 - When program is executed ⇒ process gets those capabilities
- Program is **weaker** than set-UID-*root* program
 - ⇒ **less dangerous if compromised**



Background: capabilities

- **Summary:**

- Processes can have capabilities (**subset** of power of *root*)
- Programs can have attached capabilities, which are given to processes that executes those programs
- Privileged programs/processes using capabilities are less dangerous if compromised

User namespace overview

What do user namespaces do?

- Allow per-namespace **mappings** of UIDs and GIDs
 - I.e., process's UIDs and GIDs inside NS may be different from IDs outside NS
- Interesting use case: process has nonzero UID outside NS, and UID of 0 inside NS
 - Process has *root* privileges **for operations inside user NS**
 - Understanding what that means is our goal...

Relationships between user namespaces

- User NSs have a **hierarchical relationship**:
 - Each user NS (except initial user NS) has a parent user NS
- **Parent of a user NS** == user NS of process that created this user NS
- Parental relationship determines some rules about how capabilities work
 - (End slides)

The first process in a new user NS has *root* privileges

- When a new user NS is created, first process in NS has **all** capabilities
 - Creation is done using *unshare(1)*, *clone(2)*, or *unshare(2)*
- That process has superuser powers!
- ... but only inside the user NS

UID and GID mappings

UID and GID mappings

- One of first steps after creating a user NS is to define **UID and GID mappings** for NS
- Defined by writing to 2 files: `/proc/PID/uid_map` and `/proc/PID/gid_map`
- For security reasons, there are **many rules** governing:
 - **How / when** files may be updated
 - **Who** can update the files
 - Way too many details to cover here...
 - See `user_namespaces(7)`

UID and GID mappings

- Records written to/read from `uid_map` and `gid_map` have the form:

<code>ID-inside-ns</code>	<code>ID-outside-ns</code>	<code>length</code>
---------------------------	----------------------------	---------------------

 - `ID-inside-ns` and `length` define range of IDs inside user NS that are to be mapped
 - `ID-outside-ns` defines start of corresponding mapped range in "outside" user NS
- Commonly these files are initialized with a single line containing "root mapping":

<code>0 1000 1</code>

 - I.e., UID 0 inside NS maps to unprivileged UID in outer NS

Example: creating a user NS with "root" mappings

- `unshare -U -r` creates user NS with root mappings
- Create a user NS with root mappings running new shell, and examine map files:

```
$ id # Show credentials in current shell
uid=1000(mtk) gid=1000(mtk) ...

$ PS1='uns2$ ' unshare -U -r bash
uns2$ cat /proc/$$/uid_map
0 1000 1
uns2$ cat /proc/$$/gid_map
0 1000 1
```

- (`$$` is PID of the shell)

Example: creating a user NS with “root” mappings

- Examine credentials of new shell:

```
uns2$ id  
uid=0(root) gid=0(root) groups=0(root) ...
```

- Examine capabilities of new shell:

```
uns2$ grep -E 'CapPrm|CapEff' /proc/$$/status  
CapPrm: 000001ffffffffffff          # Hex bit mask  
CapEff: 000001ffffffffffff
```

- `0xffffffffffff` is bit mask with all capability bits set
- `getpcaps` gives same info more readable:

```
uns2$ getpcaps $$  
21135: =ep
```

- '`=ep`' means all permitted and effective capabilities

Example: creating a user NS with “root” mappings

- Discover PID of shell in new user NS:

```
uns2$ echo $$  
21135
```

- From a shell in **initial user NS**, examine credentials of that PID:

```
$ ps -o 'uid,gid,pid' 21135  
UID   GID   PID  
1000  1000  21135
```

I'm superuser, right?

- From the shell in new user NS, let's try to change the hostname

- Requires `CAP_SYS_ADMIN`

```
uns2$ hostname langwied  
hostname: you must be root to change the host name
```

- What went wrong?

- After all, that shell has **all** capabilities

- The new shell is in new user NS, but **still resides in initial UTS NS**

- (Remember: hostname is isolated/governed by UTS NS)
 - Let's look at this more closely...

User namespaces and capabilities

User namespaces and capabilities

- Kernel grants **all** capabilities to initial process in new user NS of capabilities
- But, those capabilities are available **only for operations on objects governed by the new user NS**
 - But what does that mean?

User namespaces and capabilities

- We've already seen that:
 - There are a number of NS types
 - Each NS type governs some global resource(s); e.g.:
 - UTS: hostname
 - Mount: mount list
 - Network: IP routing tables, port numbers, `/proc/net`, ...
- Adding to this: **each nonuser NS instance is owned by some user NS instance**
 - When creating new nonuser NS, kernel marks that NS as owned by **user NS of process creating the new NS**
 - If a process operates on resources governed by nonuser NS:
 - Permission checks are done according to that **process's capabilities in user NS that owns the nonuser NS**

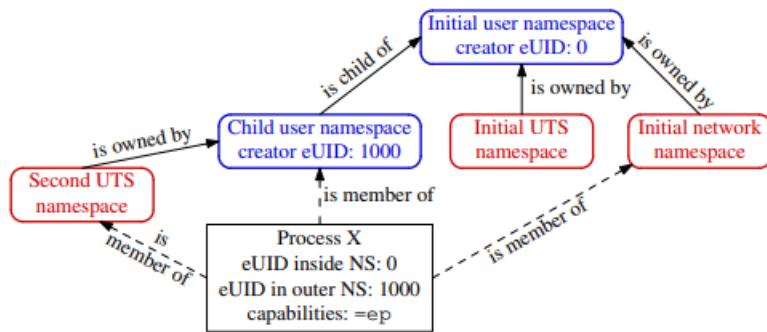
User namespaces and capabilities

- To illustrate, let's look at set-up resulting from command:

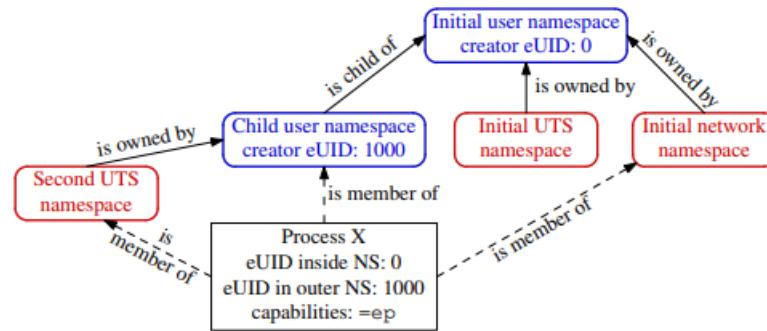
```
unshare -Ur -u <prog>
```

(Create process running `prog` in new user NS
with root mappings + new UTS NS)

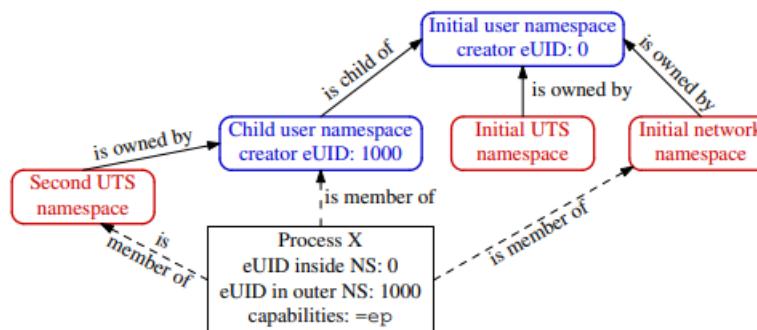
User namespaces and capabilities—an example



- X is in new user NS, with root mappings, has all capabilities
- X is in a new UTS NS, which is owned by new user NS
- X is in initial instance of all other NS types (e.g., NW NS)

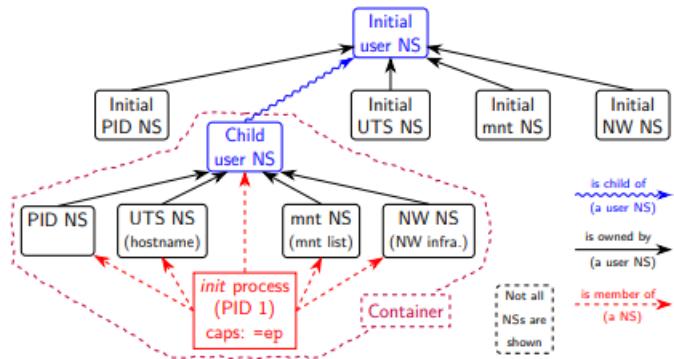


- Suppose X tries to change hostname (`CAP_SYS_ADMIN`)
- X is in second **UTS** NS
- Privilege checked according to X's capabilities in user NS that owns that UTS NS ⇒ succeeds (X has capabilities in that user NS)



- Suppose X tries to turn NW device up/down (`CAP_NET_ADMIN`)
- X is in initial **network** NS
- Privilege checked according to X's capabilities in user NS that owns network NS ⇒ attempt fails (no capabilities in initial user NS)

Containers and namespaces



- “Superuser” process in a container has **root power over resources governed by non-user NSs owned by container’s user NS**
- And does **not** have privilege in outside user NS
 - (E.g., can’t change mounts seen by processes outside container)

Discovering namespace relationships

- There are APIs to discover:
 - Parental relationships between user NSs
 - Ownership relationships between user NSs
 - See `ioctl_ns(2)`
- Code example: `namespaces/namespaces_of.go`
 - Shows NS memberships of specified processes, in context of user NS hierarchy
 - Better example: <https://github.com/TheDive0/lxkns>

Demo: effect of capabilities in a user NS

- Create a shell in new user and UTS NSs:

```
$ unshare -Ur -u bash
# getpcaps $$

353: ep          # Shell has all capabilities in its user NS
```

- Since this shell has all capabilities in user NS that owns its UTS NS, we can change hostname:

```
# hostname
bienne
# hostname langwied
# hostname
langwied
```

- But, this shell is in a network NS owned by **initial** user NS, and so can’t turn a NW device down:

```
# ip link set dev lo down
RTNETLINK answers: Operation not permitted
```

Discovering namespace relationships

- Inspect with namespaces/namespaces_of.go program:

```
$ echo $$      # PID of a shell in initial user NS
327
$ go run namespaces_of.go --namespaces=net,uts 327 353
user {4 4026531837} <UID: 0>
    [ 327 ]
net {4 4026532008}
    [ 327 353 ]
uts {4 4026531838}
    [ 327 ]
user {4 4026532760} <UID: 1000>
    [ 353 ]
uts {4 4026532761}
    [ 353 ]
```

- Indentation indicates user NS ownership / parental relationship between user NSs
- Shells are in same network NS, but different UTS+user NSs
- Second UTS NS is owned by second user NS
- `{...}` shows unique NS identifier (device ID + inode #)



Use cases

Applications of user namespaces

User NSs permit many interesting applications; for example:

- **Running Linux containers without `root` privileges**
 - Docker, LXC
- **Chrome-style sandboxing of browser renderer process**
 - Sandbox renderer process, because it is an attack target
 - Formerly, use of set-UID-`root` helpers was required
 - <https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md>
- User NS with single UID identity mapping ⇒ no superuser possible!
 - E.g., `uid_map: 1000 1000 1`

Applications of user namespaces

- **Firejail**: namespaces + seccomp + capabilities for generalized, **simplified sandboxing** of any application
 - Predefined sandboxing profiles exist for 1000+ common apps (Chrome, LibreOffice, VLC, *tar*, *vim*, *emacs*, ...)
 - <https://firejail.wordpress.com/>, <https://lwn.net/Articles/671534/>
- **Flatpak**: namespaces + seccomp + capabilities + cgroups for **application packaging** / sandboxing
 - Allows upstream project to provide packaged app with all necessary runtime dependencies
 - No need to rely on packaging in downstream distributions
 - Package once; run on any distribution
 - Desktop applications run seamlessly in GUI
 - <http://flatpak.org/>, <https://lwn.net/Articles/694291/>
 - Ubuntu *Snap* is a similar concept

Further information

- My LWN.net article series *Namespaces in operation*
 - <https://lwn.net/Articles/531114/>
 - Many example programs and shell sessions...
- Manual pages:
 - *namespaces(7)*, *user_namespaces(7)*, etc.
 - *unshare(1)*, *nsenter(1)*
 - *capabilities(7)*
 - *clone(2)*, *unshare(2)*, *setns(2)*, *ioctl_ns(2)*
- “Linux containers in 500 lines of code”
 - <https://blog.lizzie.io/linux-containers-in-500-loc.html>
 - (But note: uses cgroups v1)

Extra

What are the rules that determine the capabilities that a process has in a given user namespace?

User namespace hierarchies

- User NSs exist in a hierarchy
 - Each user NS has a parent, going back to initial user NS
- Parental relationship is established when user NS is created:
 - Parent of a new user NS is user NS of process that created new user NS
- Parental relationship is significant because it plays a part in determining capabilities a process has in user NS

User namespaces and capabilities

- Whether a process has a capability inside a user NS depends on several factors:
 - Whether the capability is present in the process's (effective) capability set
 - Which user NS the process is a member of
 - The (effective) process's UID
 - The (effective) UID of the process that created the user NS
 - At creation time, **kernel records eUID of creator** as "owner UID" of user NS
 - The parental relationship between user NSs
 - (The `namespaces/ns_capable.c` program encapsulates the rules shown on next slide—it answers the question, does process P have capabilities in namespace X?)

Capability rules for user namespaces

- ① A process has a capability in a user NS if:
 - it is a **member of the user NS**, and
 - **capability is present in its effective set**
- ② A process that has a capability in a user NS **has the capability in all descendant user NSs** as well
 - I.e., members of user NS are not isolated from effects of privileged process in parent/ancestor user NS
- ③ Any process in **parent** user NS that has **same eUID** as eUID of creator of user NS have all capabilities in the NS
 - At creation time, **kernel records eUID of creator** as "owner UID" of user NS
 - By virtue of previous rule, process also has capabilities in all descendant user NSs

Combining user namespaces and other namespace types

- Earlier, we noted that `CAP_SYS_ADMIN` is needed to create nonuser NSs
- So, why can unprivileged user do the following?

```
$ unshare -U -u -r bash
```
- Can do this, because kernel first creates user NS, giving process all privileges, so that UTS NS can also be created
- Equivalent to following, but without intervening child process:

```
$ unshare -U -r bash # Child in new user NS
$ unshare -u bash    # Grandchild in new UTS NS
```

What about resources not governed by namespaces?

- Some privileged operations relate to resources/features not (yet) governed by any namespace
 - E.g., system time, kernel modules
- Having capabilities in a noninitial user NS doesn't grant power to perform operations on features not currently governed by any NS
 - E.g., can't change system time or load/unload kernel modules

But what about accessing files (and other resources)?

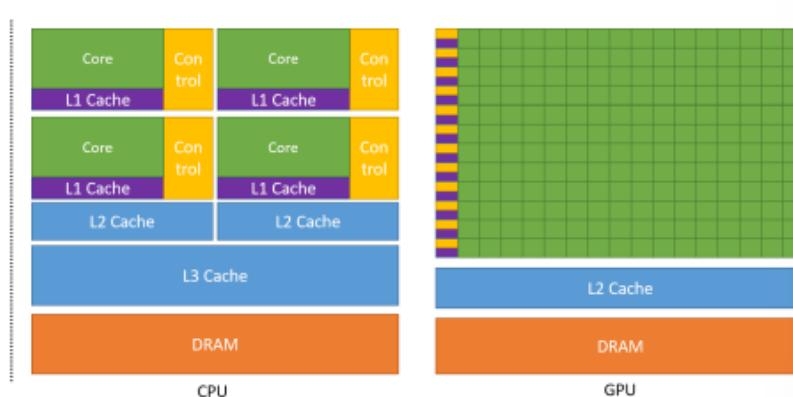
- Suppose UID 1000 is mapped to UID 0 inside a user NS
- What happens when process with UID 0 inside user NS tries to access file owned by ("true") UID 0?
- When accessing files, IDs are mapped back to values in initial user NS
 - There is a chain of user NSs starting at NS of process and going back to initial NS
 - Examining the mappings in this chain allows kernel to know "true" UID and GID of a process
 - Same principle for checks on other resources that have UID+GID owner
 - E.g., various IPC objects

Heterogeneous computing

Gpu's, FPGA, Accelerator, tradeoff processors

GPUs

- Parallel Processor
- Throughput-oriented
- Low Working Frequency



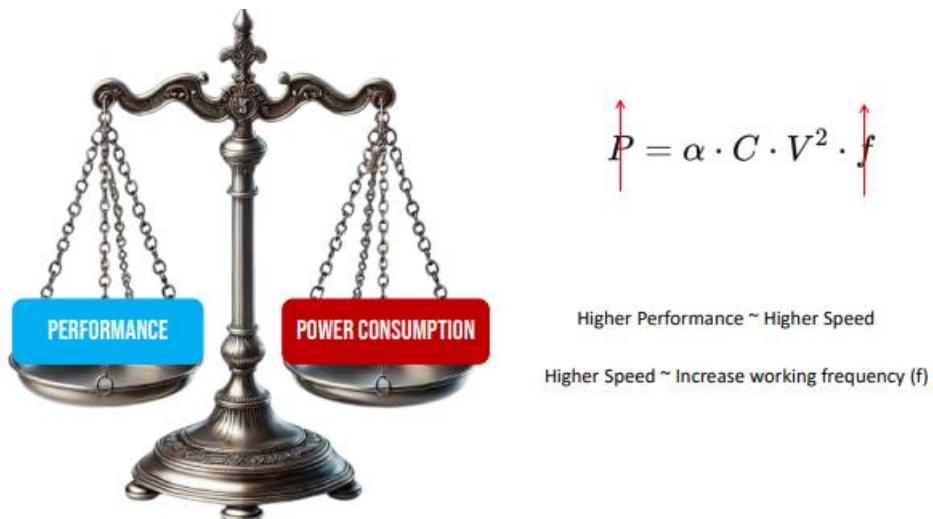
GPUs

$$P = \alpha \cdot C \cdot V^2 \cdot f$$

- Parallel Processor
- Throughput-oriented
- Low Working Frequency

where:

- P : Total power consumption of the processor.
- α : Activity factor, representing the fraction of transistors actively switching in each clock cycle. It ranges from 0 to 1.
- C : Capacitance of the processor's circuits, a measure of how much charge the circuits can hold.
- V : Supply voltage, or operating voltage, applied to the processor.
- f : Operating frequency (clock speed) of the processor, which directly influences the speed of operations.



GPU & Parallelism



GPUs, or Graphics Processing Units, are parallel processors designed to handle multiple tasks simultaneously. Unlike CPUs, which focus on sequential processing with a few powerful cores, GPUs have thousands of smaller cores optimized for parallel execution, allowing them to process large blocks of data at once. For a program to benefit from GPU power, it must have a parallel nature or be parallelizable, meaning tasks can be divided into smaller, independent operations that can run concurrently across GPU cores.

Where GPUs Excel: Example 1



Image and Video Processing

GPUs are optimized for manipulating large matrices, making them ideal for image and video processing. When rendering high-resolution images or processing video frames, GPUs can handle the parallel nature of pixel and frame-based operations, such as color transformation, shading, and rendering effects. By executing operations for thousands of pixels simultaneously, GPUs achieve faster processing times, which is critical in real-time rendering for video games, 3D modeling, and visual effects.

Where GPUs Excel: Example 2



Deep Learning and AI

Neural networks, especially in deep learning, involve repetitive matrix multiplications across multiple layers of neurons. GPUs excel at these tasks by parallelizing matrix operations—specifically, tensor computations, which involve linear algebra at a massive scale. Libraries like CUDA and cuDNN (from NVIDIA) are optimized to leverage GPU cores for deep learning frameworks such as TensorFlow and PyTorch, enabling accelerated training times and efficient model inference. For example, training a convolutional neural network (CNN) on image data can be significantly faster on a GPU than on a CPU due to the ability to process multiple convolution operations in parallel.

12

Where CPUs Win: Example 1



General-Purpose Computing

CPUs are designed to handle a wide range of tasks with complex branching logic, which is essential for running operating systems, applications, and user-driven tasks. CPUs feature fewer but more powerful cores optimized for executing instructions in sequence, making them better suited for varied operations that require frequent decision-making and task-switching, such as file management, internet browsing, and office applications.

Where CPUs Win: Example 2



Low-Latency Requirements

CPUs are optimized for low-latency tasks where the system must respond quickly to input. For example, database servers and real-time analytics require low-latency responses to user queries or data updates. With their large cache memory and high single-threaded performance, CPUs provide immediate access to data and can handle I/O operations more effectively than GPUs, which are optimized for large, parallelizable tasks rather than quick task switching.

Where CPUs Win: Example 3

Single-Threaded Performance

Some applications, like code compilation and certain scientific algorithms, are inherently sequential and cannot be parallelized. CPUs have higher clock speeds and superior single-thread performance compared to GPUs, making them ideal for these single-threaded tasks. For instance, certain steps in data analysis, such as indexing or data sorting, are faster on a CPU as they benefit from its focus on sequential processing rather than parallel execution.

GPUs as Throughput-Oriented Processors



GPUs are often described as "throughput-oriented processors," meaning they are designed to maximize the volume of data processed over time, rather than focusing on minimizing the time taken for each individual operation. This high throughput is achieved by utilizing a large number of simpler cores that operate in parallel, enabling GPUs to handle substantial amounts of data simultaneously. Unlike CPUs, which are optimized for low latency and excel in sequential processing tasks, GPUs are optimized for throughput, making them particularly effective for applications that require processing large datasets in parallel. This design makes GPUs highly suitable for tasks such as image processing, scientific simulations, and deep learning, where handling many operations at once is essential for performance.

Why GPUs Operate at Lower Frequencies



GPUs are considered "low working frequency processors," meaning they operate at lower clock speeds compared to CPUs. While CPUs are designed with high clock frequencies to execute individual tasks quickly, GPUs focus on parallelism rather than speed per core. By using lower clock frequencies, GPUs consume less power per core, allowing them to support thousands of cores running simultaneously. This design makes them highly efficient for tasks requiring massive parallel computation, such as rendering graphics, running machine learning models, and processing large data arrays, where the collective processing power of many cores is more beneficial than the high-speed performance of a few cores.

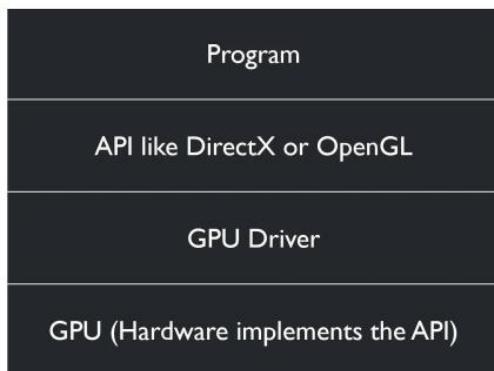
Story of NVIDIA GPUs

- Early GPUs: specific-function co-processors (ASICs) for graphics
- Programmability: Graphics APIs, e.g., DirectX and OpenGL
- Programmability required:
 - Expertise in Computer Graphics
 - Data Transformation Techniques

ASICs: Application Specific Integrated Circuits



An ASIC (Application-Specific Integrated Circuit) is a type of hardware designed to perform a specific task or set of tasks very efficiently. Unlike general-purpose processors, like CPUs, which are versatile and can handle a wide variety of instructions, ASICs are custom-built to perform only one particular function or application. Because of this specialization, ASICs can operate at much higher speeds and lower power consumption than general-purpose processors when handling their designated tasks. ASICs are commonly used in devices that need to execute repetitive operations very quickly and with high efficiency, such as image processing, network devices, audio processing, and specialized parts of smartphones. For example, in a smartphone, an ASIC might be dedicated to handling image signal processing (ISP) for the camera, allowing it to quickly enhance photos by adjusting colors, brightness, and noise with minimal power consumption. However, the downside of ASICs is their lack of flexibility—once manufactured, they cannot be repurposed for other tasks, and designing them is costly and time-consuming. This trade-off makes ASICs ideal for high-volume applications where their speed and efficiency justify the cost of custom design.



Pre-requisites Terminology

Shader: a computer program performing graphics-related tasks

vertex: a data structure describing a certain attribute, e.g., the position of a point in 2D or 3D space, or multiple points on a surface.

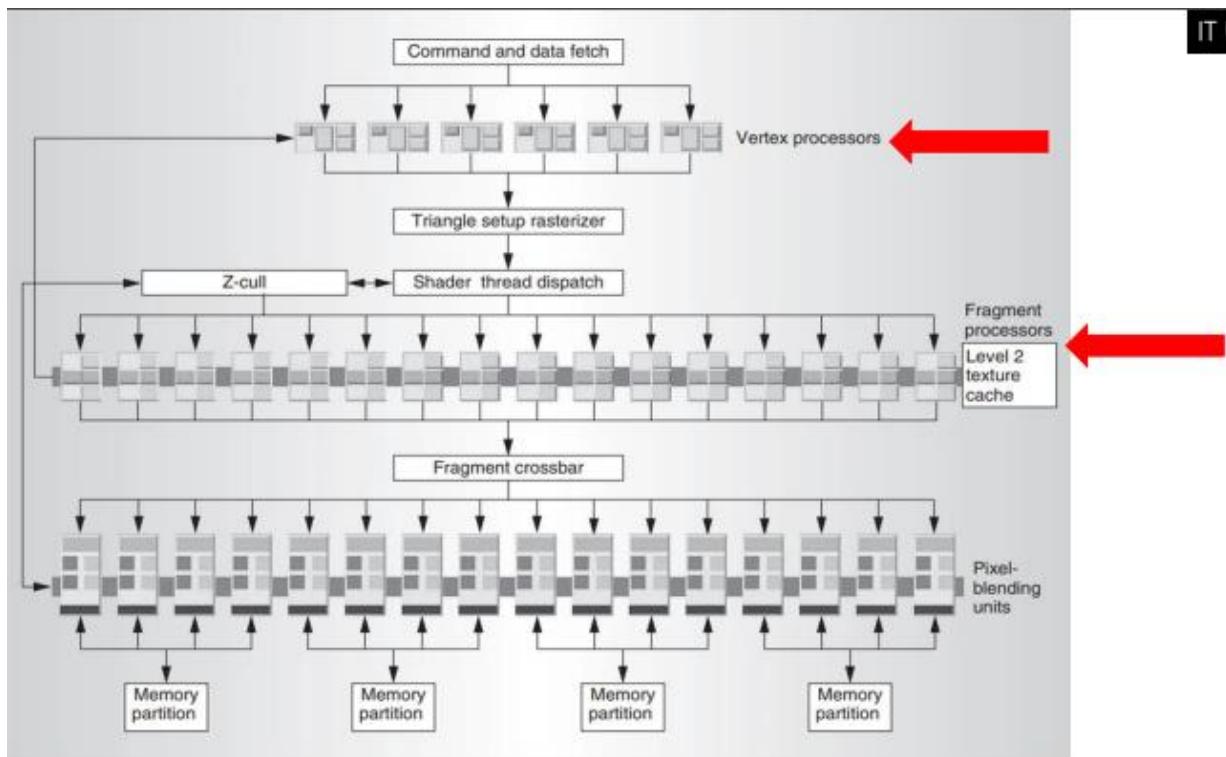
Vertex shader: a program that transforms each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen.

Pixel or fragment shader: a program that computes the color, brightness, contrast, and other attributes of a single pixel or fragment.

Early GPUs

- Early GPUs (before 2007) architecture:
 - Separate stages of vertex and pixel processors

EARLYGPUS: THE ERA BEFORE CUDA UNLEASHED GENERAL-PURPOSE POWER



Montrym, J. & Moreton, Henry, "The GeForce 6800," Micro (2005), IEEE. 25. 41–51. 10.1109/MM.2005.37

Early GPUs



IT UNIVERSITY

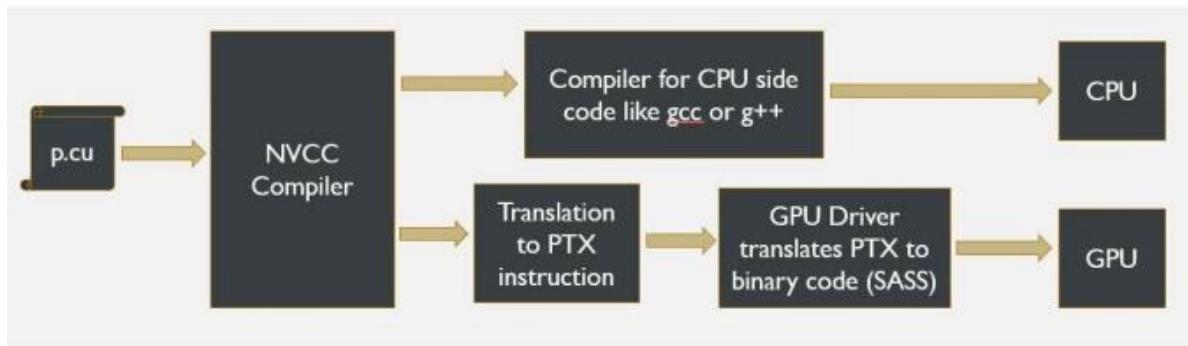
uArch.	Year	CUDA?	#Trans	Fab. Tech.	OpenGL	DirectX	GPU Cards
Fahrenheit	1998	No!	7-15M	350-240nm	1.2	5.0, 6.0	Vanta, Riva Models
Celsius	1999	No!	20-29M	180-150nm	1.5	7.0	GeForce 256 GeForce 2 series
Kelvin	2001	No!	36-57M	150nm	1.5	8.0	GeForce 3 and 4 series
Rankine	2003	No!	~125M	150-130nm	2.1	9.0a	GeForce 5 (or FX) series
Curie	2004	No!	~220M	130-80nm	2.1	9.0c	GeForce 6 and 7 series

MOORE'S LAW EFFECT IS EVIDENT IN THE TABLE ON HOW THE NUMBER OF TRANSISTORS INCREASES OVER THE YEARS.

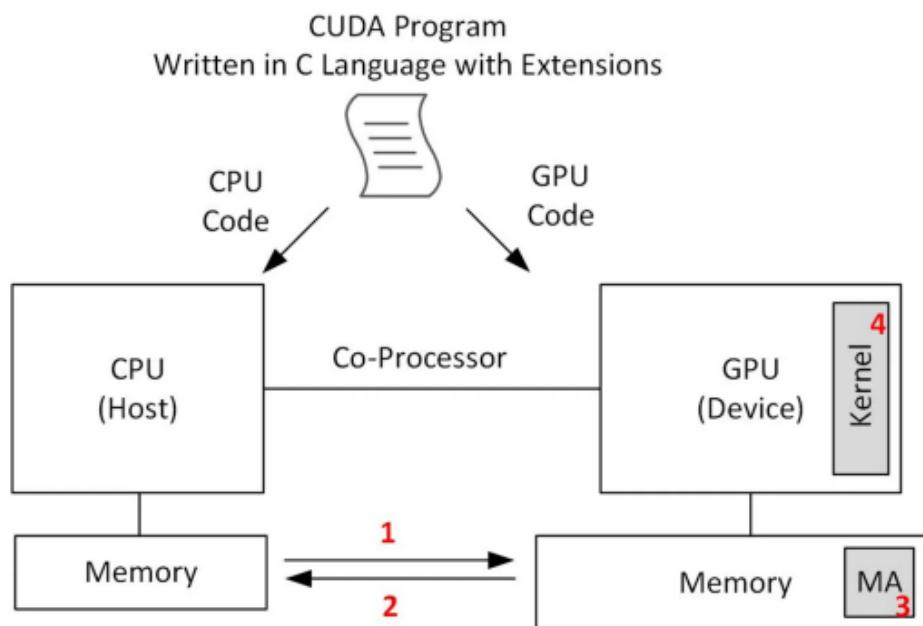
The Big Shift: CUDA Revolutionizes GPU Computing

- **CUDA:** Compute Unified Device Architecture
- A parallel computing **framework platform** and **API** allowing software to use certain types of NVIDIA GPUs for general-purpose computing
- CUDA programming is possible for C/C++ programmers can use ‘CUDA C/C++’ for programming NVIDIA GPUs

From Code to Execution: The CUDA Process



```
$ nvcc my_program.cu -o my_program
```



Microarchitecture Innovations Enabling the CUDA Revolution

- At 2006, NVIDIA introduced **Tesla** Microarchitecture
- Implementing a **UNIFIED SHADER MODEL**
 - (remember Separate stages of vertex and pixel processors)
- Benefits of the Unifications
 - Better of Management of Hardware Resources
 - Load Balancing between Vertex and Pixel (fragment) stage
 - Simpler GPU Design
- Cores became: (1) Sequential (2) Scalar – meaning being able to work on only one computation task at a time



Microarchitecture Innovations Enabling the CUDA Revolution

- Changed their names to: CUDA cores
- They grouped together in Streaming Multiprocessors (SM), which replaced split stages of vertex and fragment units
- Each SM receives Thread Blocks that are composed of groups of threads in the number of 32, which are called **warp**.
- All threads in a warp execute the same instruction at the same time but on different data (**SIMT: Single Instruction Multiple Thread**).

31

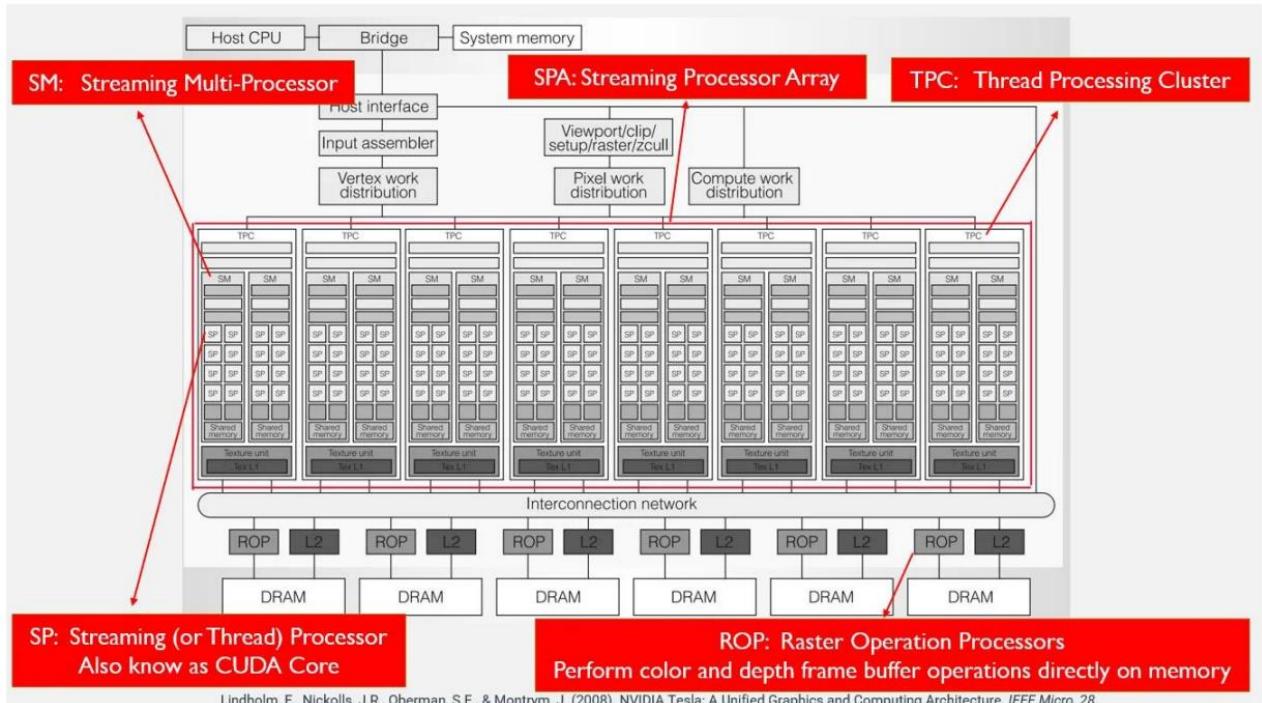
Microarchitecture Innovations Enabling the CUDA Revolution

- Jonah Alben: the senior vice president of GPU engineering at NVIDIA

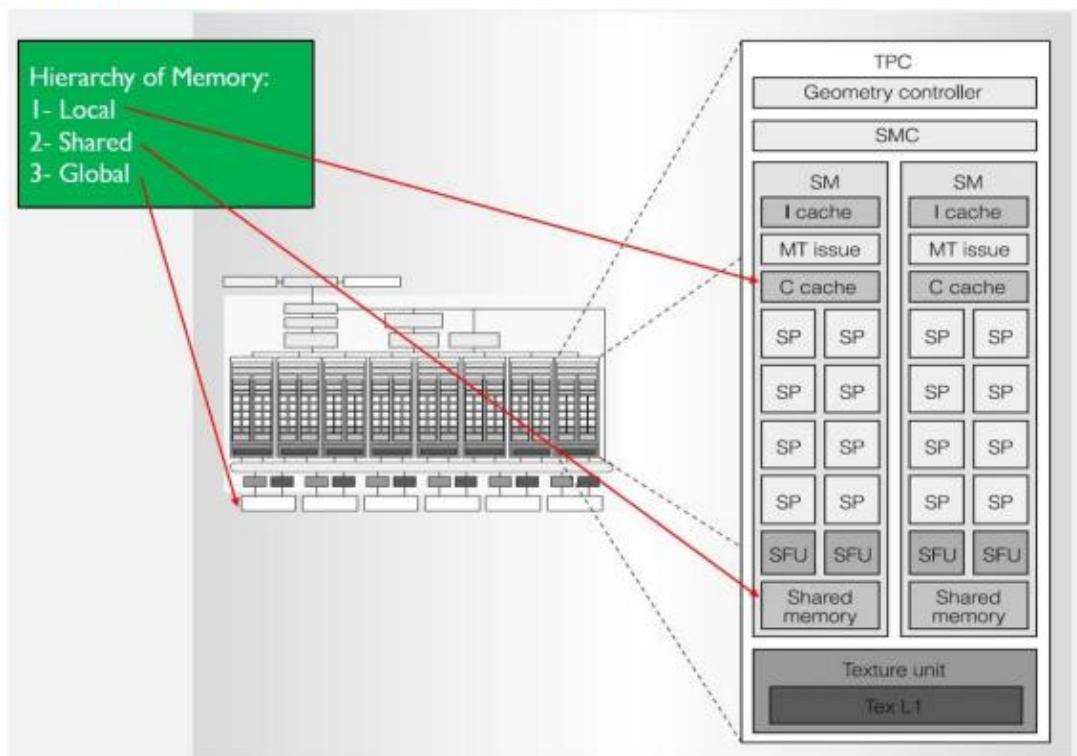


We pretty much threw out the entire shader architecture from NV30/NV40 and made a new one from scratch with a new general processor architecture (SIMT), that also introduced new processor design methodologies.

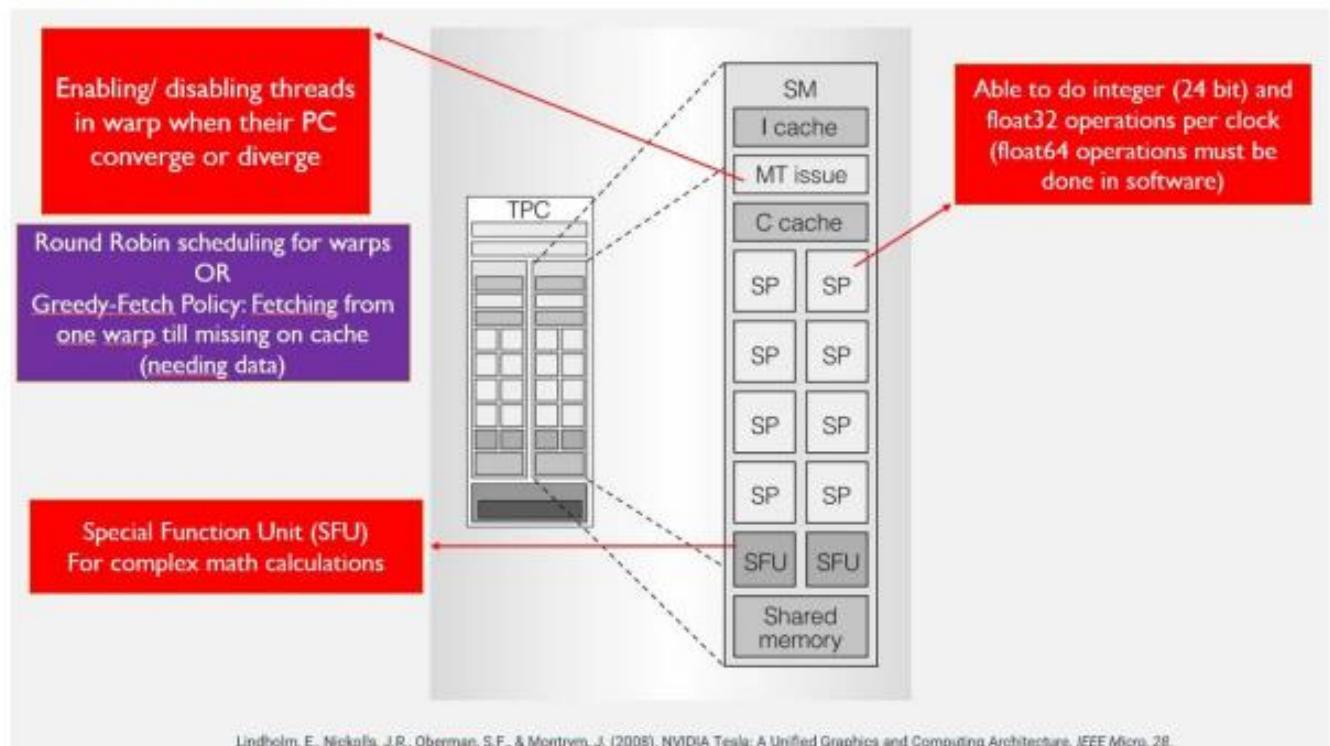
NVIDIA Tesla 2006 Architecture



Inside a TPC

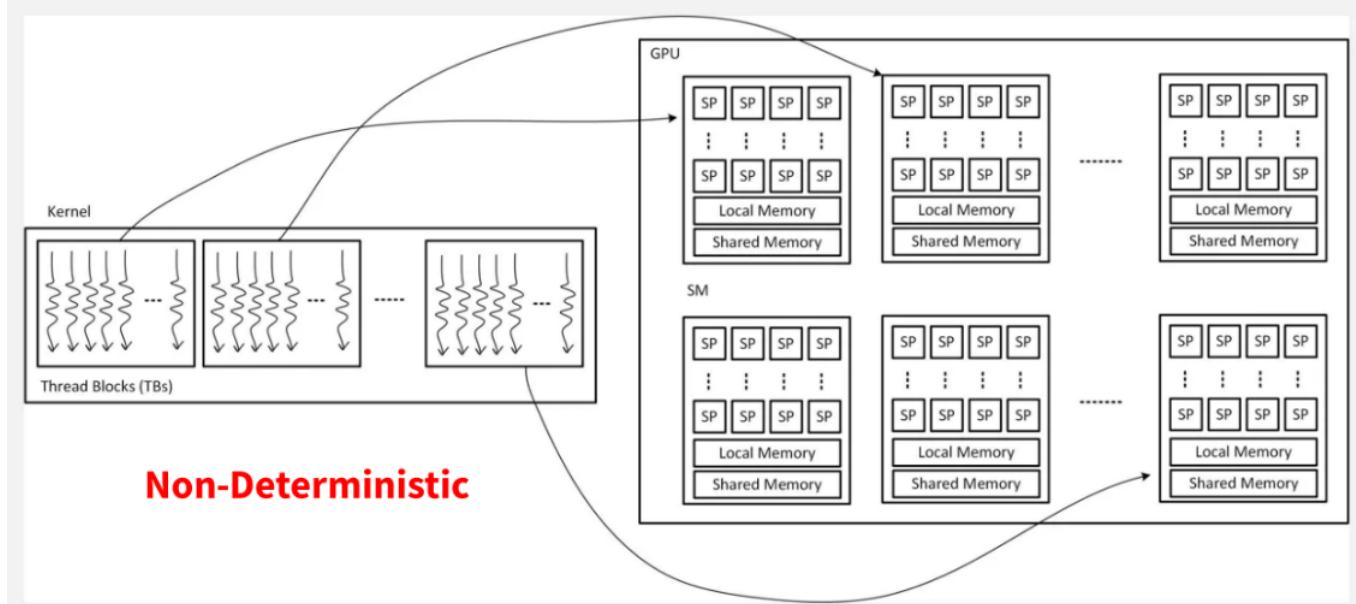


Inside a SM



Lindholm, E., Nickolls, J.R., Oberman, S.F., & Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28.

THE EXECUTION MODEL!



<https://github.com/ehsanyousefzadehasl/PCwGPGPUs>

FPGAs: Field Programmable Gate Array

- A hardware that can be configured after being manufactured
- **Initial Purpose:**
 - **Prototyping** digital circuits and testing Logic Designs (before being manufactured as an ASIC)
 - Implementing small, custom digital circuits (using the FPGA as that circuit)
- **Flexibility** was considered more important, not performance or power
- Early Challenges:
 - A **deep understanding of hardware design**
 - **Low-level hardware description languages (HDLs)** like **Verilog** and **VHDL**
- Early FPGA work demanded a solid grasp of
 - **Digital logic** concepts,
 - **Timing analysis**, and **circuit behavior**
 - Hardware Experts (Digital Electronic Engineers)

40

FPGAs Big Players



- **Xilinx and Altera**
 - Developed many of the innovations in FPGA technology
 - Advanced Architecture
 - High-level Programming Tools
- **Intel acquired Altera**
 - integrating Altera's FPGA technology into Intel's product lineup
- **AMD acquired Xilinx in 2022**
 - To enhance its data center and high-performance computing offerings with Xilinx's FPGA technology

Integration with CPUs

- Both Intel and AMD
 - **integrate FPGAs with CPUs on a single chip** or in multi-chip modules
 - combines the **flexibility** and **parallelism** of FPGAs with the general-purpose processing power of CPUs
 - **Heterogeneous Computing**
- **Benefits**
 - Sharing Same Memory Space
 - Reducing Latency between CPU/FPGA
 - FPGA acceleration for tasks, like:
 - Encryption
 - Compression
 - Machine Learning Inference
 - Data Filtering



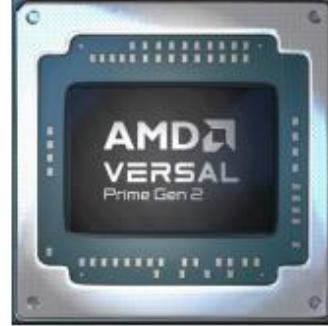
43

Integration with CPUs



AI Edge Series Gen 2

Next-generation AI Engines, high-performance integrated CPUs, and programmable logic enabling preprocessing, AI inference, and postprocessing for AI-driven embedded systems—all in a single device.



Prime Series Gen 2

High-performance integrated CPUs, programmable logic, and 8K video processing for next-level classic embedded systems across a wide range of markets.

<https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal.html>

44

Applications of FPGAs



- **Initial Applications:** Early on, FPGAs were mainly used in **telecommunications**, **signal processing**, and **embedded systems** where custom digital logic was required but dedicated ASICs weren't economical.
- **Modern Applications:**
 - **Data Centers:** For accelerating tasks like **AI inference**, **data processing**, and **high-speed networking**.
 - **5G and Telecommunications:** For **signal processing**, **packet processing**, and handling data transfer in base stations.
 - **Automotive:** Used in advanced driver-assistance systems (ADAS) and autonomous vehicle applications, where reconfigurability allows for updates as new features are developed.
 - **Financial Services:** In high-frequency trading, where FPGAs can execute complex algorithms with extremely low latency.
 - **Aerospace and Defense:** For **radar processing**, **encryption**, and **secure communications**, where real-time processing and reliability are critical.

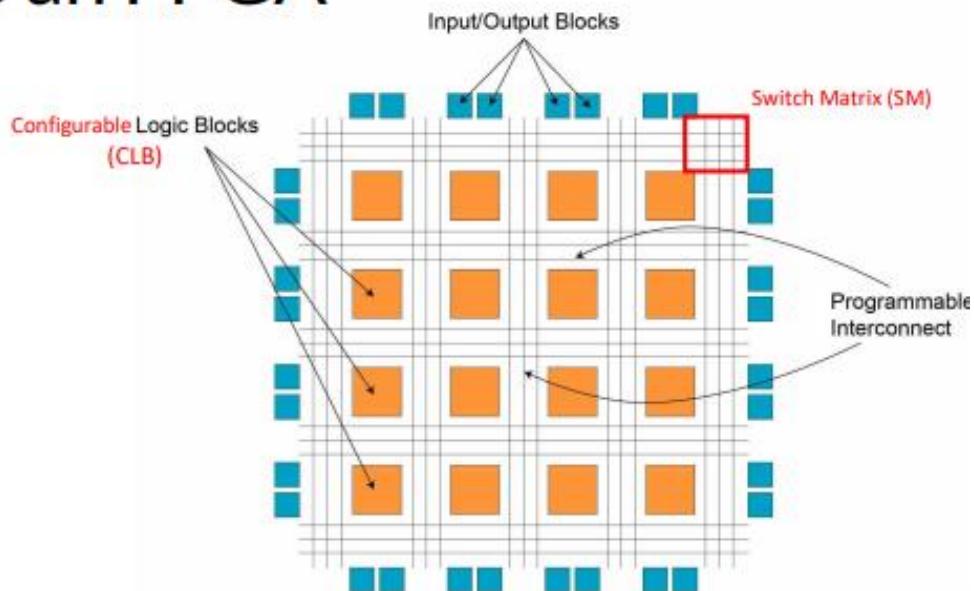
Current State and Modern Applications



- **State of FPGAs:** FPGAs are now key components in **data center** and **edge computing** infrastructures, used for both **prototyping** and **production** systems. With continued investment from Intel and AMD, FPGAs are becoming more tightly integrated with CPUs and GPUs, enhancing performance for a wider range of applications (**Heterogenous Computing**).
- **Modern Applications:** FPGAs are focusing on accelerating complex, compute-heavy tasks in areas such as:
 - **Machine Learning and AI:** FPGA-based AI inference accelerators are growing in popularity due to their **low latency** and ability to be **reconfigured** for different models or algorithms.
 - **Real-Time Data Processing:** FPGAs are ideal for tasks that require real-time, high-throughput processing, such as **video streaming**, **IoT data analysis**, and **sensor data processing**.
 - **Cybersecurity:** FPGAs can be used for encryption, decryption, and secure processing, particularly in industries that require **high security** and **flexibility**.
 - **Cloud Computing:** FPGAs are now available as part of cloud platforms like AWS (with F1 instances) and Microsoft Azure, providing hardware acceleration for users without needing physical access to an FPGA device.

Inside an FPGA

IT UNI



<https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/>

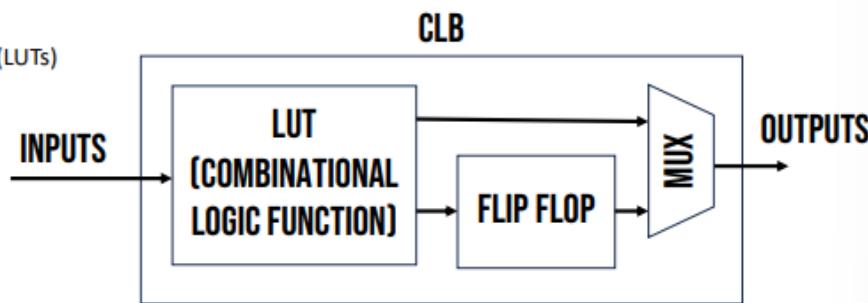
Inside an FPGA

- Configurable Logic Block (CLBs): Processing Units of an FPGA

- Can be configured to perform
 - Basic logic functions like AND, OR, NOT, XOR
 - More complex combinational and sequential logic

- Inside CLBs:

- Look-Up Tables (LUTs)
- Flip-Flops (FFs)



Inside an FPGA

- Interconnects

- Connecting CLBs, allowing signals to flow between different parts of the FPGA
 - Switch Matrix

- I/O Blocks

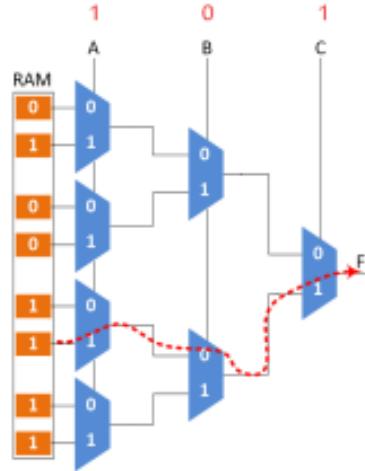
- To interface with external components

- Clocking resources

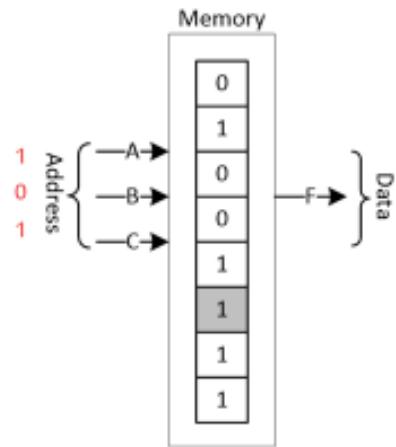
- ensures consistent timing throughout the chip, providing synchronized operation for flip-flops and other sequential logic components



Inside an LUT



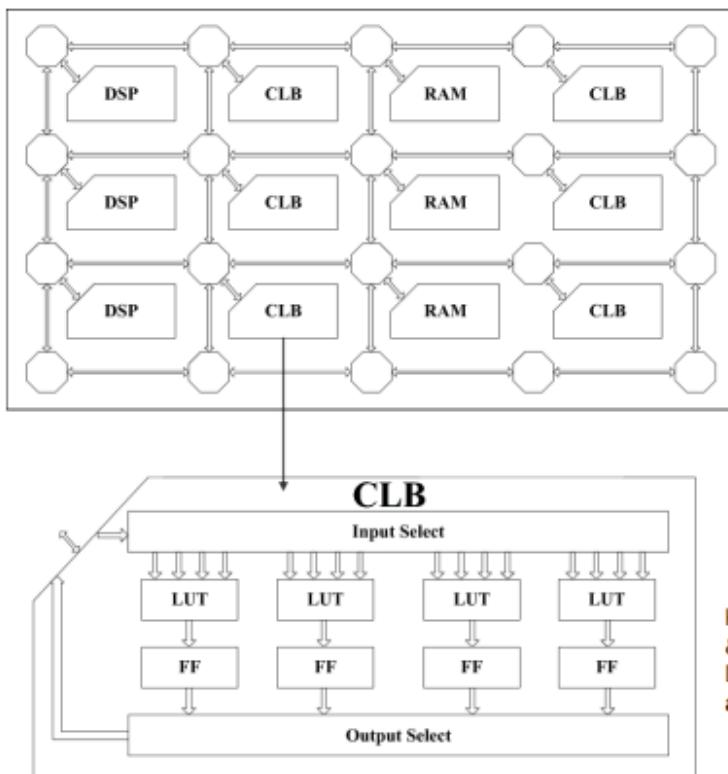
LUT in FPGA



Look-up table (LUT)

<https://fpgatek.com/what-is-an-fpga/>

50



IT UNIVERSITY OF COPENHAGEN

Kastner, Ryan & Jung, Chair & Cho, Uk & Rao, Bhaskar & Sherwood, Timothy & Swanson, Steven & Tullsen, Dean. **GUSTO: General architecture design Utility and Synthesis Tool for Optimization.**

51

Programming an FPGA

- Fundamentally different from programming a CPU or GPU
- Programming is writing instructions for a processor to execute
- Using a Hardware Description Language (HDL) means:
 - Describing the hardware circuitry
- HDL example (Low Level):
 - Verilog (**VER**ification and **LOG**ic)
 - VHDL (**VHSIC** **H**ardware **D**escription **L**anguage)
 - VHSC: Very High-Speed Integrated Circuit
- High Level Synthesis (HLS) tools:
 - Vivado HLS

Process of programming an FPGA

1. Design Description

- The developer describes the intended logic

2. Synthesis

- The HDL is synthesized into a **netlist**, which is a description of the logic gates and interconnections needed to implement the design

3. Place and Route

- The FPGA design tool assigns specific CLBs, LUTs, and interconnects on the FPGA to the gates and connections in the netlist. This process defines which blocks perform each function and how they are wired.

Low Level programming with Verilog

BEHAVIORAL LEVEL

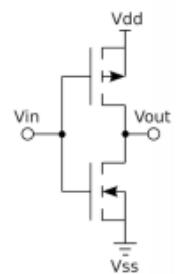
```
not n1(not_a, a);  
not n2(not_b, b);  
and a0(oa0, D, not_a, not_b);  
in);
```

REGISTER TRANSFER LEVEL (RTL)

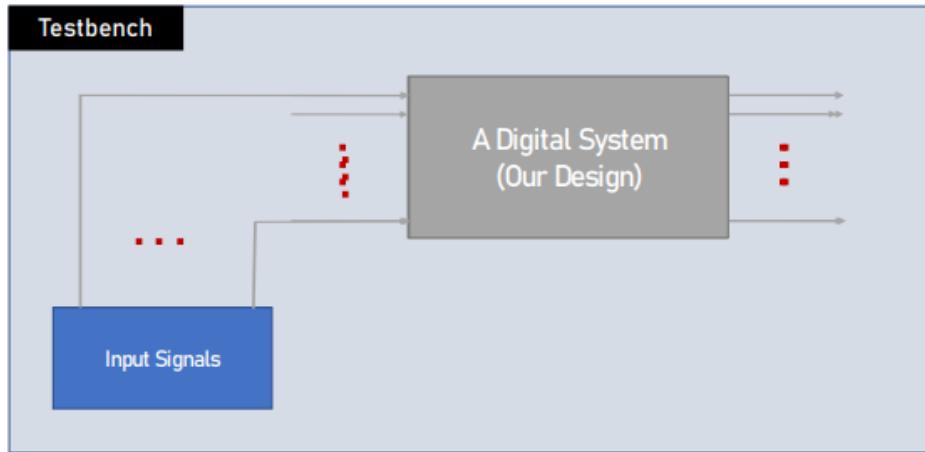
```
cmos_inverter.v  
`timescale 1ns/100ps  
module cmos_inverter(out,in);  
    output out;  
    input in;  
    supply0 GND;  
    supply1 PWD;  
  
    // out, in, control  
    pmos p0(out, PWD, in);  
    nmos n0(out, GND, in);  
endmodule
```

GATE LEVEL

SWITCH LEVEL



Verilog Example



55

Design: a 4-bit counter

```
`timescale 1ns/100ps
module counter(count, clk, reset, enable);
    input clk, reset, enable;
    output [3:0] count;
    reg [3:0] count;
    always @(posedge clk) begin
        if(reset == 1'b1) begin
            count <= 0;
        end else if(enable == 1'b1) begin
            count <= count + 1;
        end
    end
endmodule
```

Testbench

```
`timescale 1ns/100ps
module counter_tb();
    reg CLK, RESET, ENABLE;
    wire [3:0] OUTPUT;
    // instantiating from counter module
    counter C0(.count(OUTPUT), .clk(CLK), .reset(RESET), .enable(ENABLE));
    initial begin
        CLK = 0; RESET = 1; ENABLE = 0;
        #10 RESET = 0; // 10 nanoseconds delay
        #5 ENABLE = 1;
    end
    always
        #5 CLK = !CLK;
endmodule
```

High Level Synthesis (HLS), MM example

```
#include <iostream>

// Define matrix size
#define SIZE 4

// Top-Level function for HLS synthesis
void matrix_multiply(float A[SIZE][SIZE], float B[SIZE][SIZE], float C[SIZE][SIZE]) {
    // Perform matrix multiplication
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            C[i][j] = 0;
            for (int k = 0; k < SIZE; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

59

Accelerator

- Accelerator accelerates (compared to CPU)!
- Accelerator is a flexible term
 - AI Accelerator: chips like TPUs and Cerebras
 - General Purpose Accelerators like GPUs
 - Customn Accelerator like ASICs or FPGAs configured for a specific application

Cerebras Wafer-Scale Engine (WSE)



- **Cerebras WSE is an entire silicon wafer used as a single massive chip**
- The large surface area enables an extensive number of processing cores, memory, and interconnections in a single chip, reducing the need for multiple, smaller chips communicating across a board.
- The WSE includes **over 40 GB of on-chip SRAM memory** spread across the wafer. This proximity to the processing cores provides **high bandwidth** and **low latency** memory access. This eliminates the need for external memory modules, reducing delays and energy consumption associated with off-chip data transfer, which is a common bottleneck in traditional AI accelerators.
- Cerebras uses a **high-speed, low-latency interconnect fabric** that connects all cores on the chip, enabling efficient data flow across the entire wafer.
- **Optimization for AI Workloads:**
 - The Cerebras WSE is specifically optimized for **deep learning and AI workloads**, handling large matrix multiplications and tensor operations at a scale that traditional chips struggle with.
 - The architecture allows for **model parallelism**, where different parts of a neural network can run concurrently, making it highly efficient for training and inference tasks on large models.

65

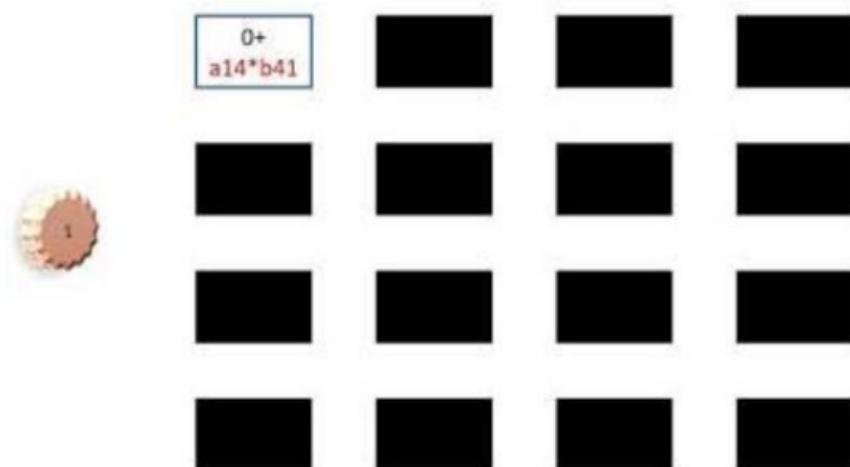
Google Tensor Processing Unit (TPU)

- A custom-designed accelerator developed by Google specifically for AI and machine learning tasks.
- The core of the TPU architecture is: Matrix Multiply Unit (**MXU**)
 - Inside MXUs, there are **128x128 systolic arrays**
 - thousands of multiplications and additions happen in parallel, significantly speeding up tensor operations.
- Unlike CPUs, and GPUs, TPUs designed specially for AI tasks
 - They avoid unnecessary caches and complex control units
- High Bandwidth Memory (HBM)
- Programming with TensorFlow and JAX library (Google's **XLA compiler** is used to further optimize TensorFlow code for TPU hardware, by fusing operations and minimizing memory usage, making TPU execution more efficient.)

6

How Systolic Arrays work!

IT UNIVERS



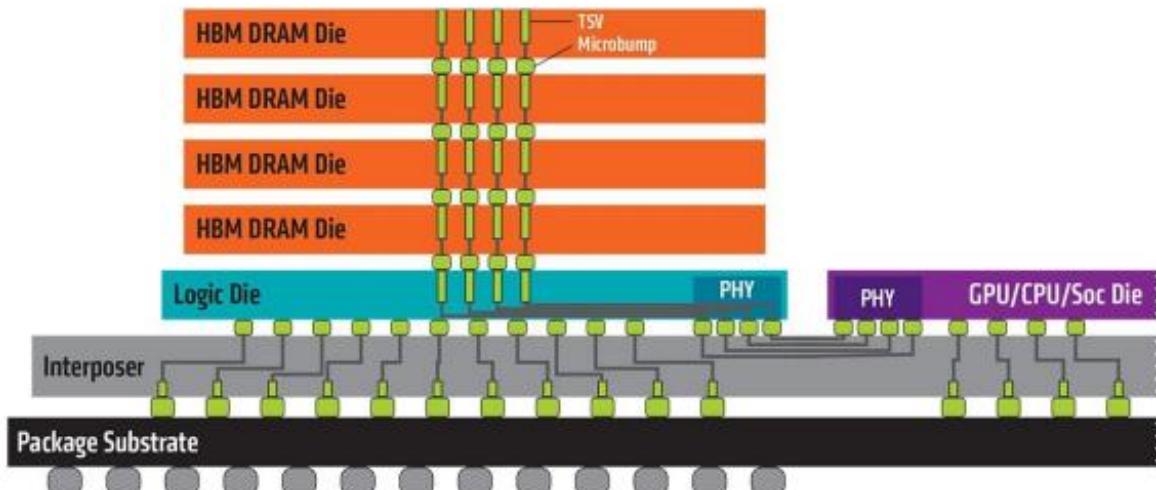
High Bandwidth Memory (HBM)



- **High Bandwidth Memory (HBM)** is an advanced type of memory technology designed to provide faster data transfer rates and higher memory bandwidth than traditional memory types, such as DDR (Double Data Rate) memory. HBM achieves this efficiency through a unique design where memory chips are **stacked vertically** and connected using a technology called **through-silicon vias (TSVs)**. This stacking allows multiple layers of memory to be connected closely, enabling data to travel shorter distances compared to traditional memory layouts.
- HBM's vertical stacking also places it much closer to the processor, typically on the same package or die. This proximity reduces **latency**—the delay before data transfer begins—and significantly **increases bandwidth**, or the rate at which data can move between memory and the processor. HBM can handle **massive amounts of data** simultaneously, making it especially beneficial for **high-performance computing tasks** like AI, deep learning, and graphics rendering, where large datasets need to be processed quickly.
- The efficiency of HBM comes not only from its high bandwidth but also from its **power efficiency**. By keeping memory closer to the processor and enabling faster data access, HBM reduces the energy needed to transfer data, resulting in lower overall power consumption. This combination of high bandwidth, low latency, and energy efficiency makes HBM an ideal choice for modern accelerators, such as GPUs and TPUs, where fast, efficient memory access is crucial for performance.

IT UNIVER

HBM



Tradeoff!

Processor	Programmability	Goal	Flexibility (Different Programs)	Promised Performance
CPU	Easy (use any programming language you know)	Latency Oriented	Super High	Fair
GPU	Medium (learning CUDA!)	Throughput Oriented	High	Medium
FPGA	Hard (Learn Verilog + Digital electronics basics)	Both	Low	High
Accelerator	Very Hard (read docs and learn concepts of the field like AI)	Both	Super Low	Super High

Modern CPUs

- They fetch and execute more than one instruction (a windows of instruction)
 - Higher throughput
- Advanced Hardware Execution Mechanisms to execute faster
- Employ Cache Hierarchy to fill the Memory-Processor performance gap
 - Temporal/ Spatial Locality
- They have several cores (parallel computing)



Hennessy and Patterson



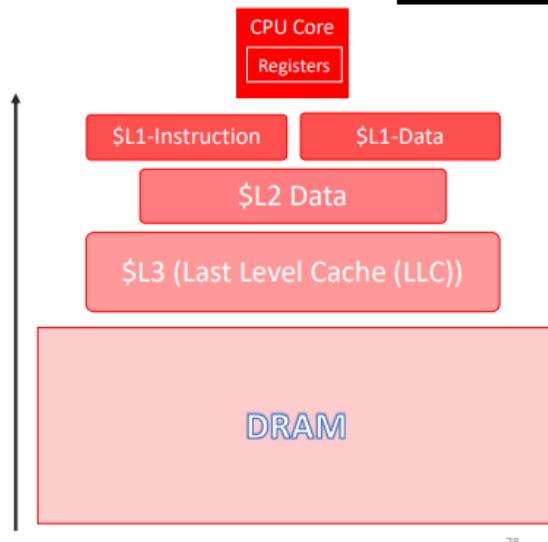
Tomasulo



Yale Patt

Cache Hierarchy

**Less Access latency
More Data Locality**
**Less Storage Capacity
More Expensive per bit**

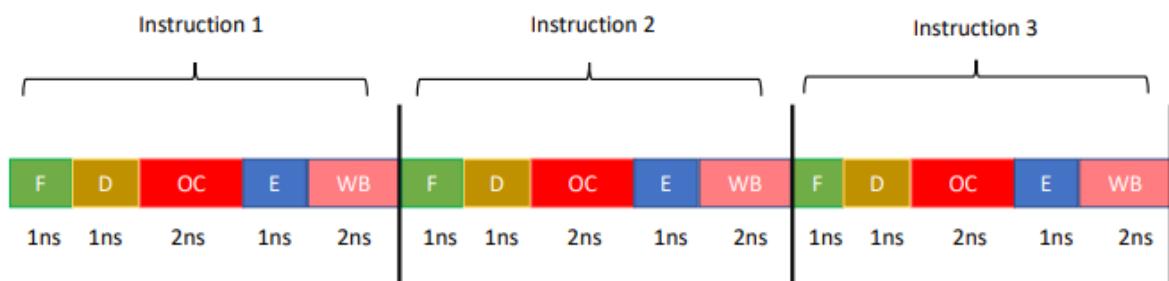
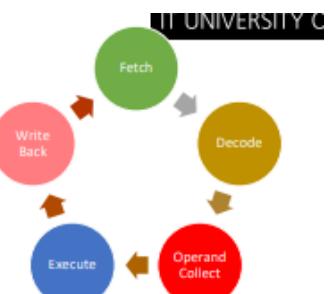


78

Pipelining

- Basic Processor

Inst1	O11, O12
Inst2	O21, O22
Inst3	O31, O32

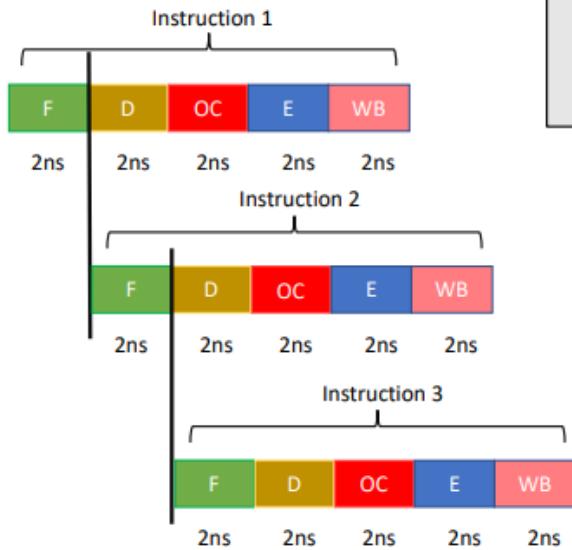


Overall Time of Executing three Instruction: $3 * (1ns + 1ns + 2ns + 1ns + 2ns) = 3 * 7ns = 21ns$

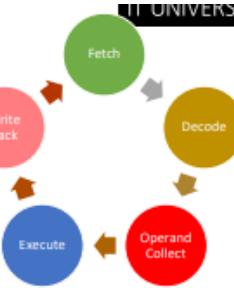
79

Pipelining

- Pipelined Basic Processor



Inst1	O11, O12
Inst2	O21, O22
Inst3	O31, O32



Overall Time of Executing three Instruction:

$$\begin{aligned}
 &= (2\text{ns} + 2\text{ns} + 2\text{ns} + 2\text{ns} + 2\text{ns}) + 2\text{ns} + 2\text{ns} \\
 &= 10\text{ns} + 2\text{ns} + 2\text{ns} = 14\text{ns}
 \end{aligned}$$

Implicit Parallelism
Instruction-Level Parallelism (ILP)
Անսկուլպանական զարգացման համար

80

CUDA

Content

- Why GPUs?
- CUDA and Heterogeneous Computing
- CUDA Concepts and Hands-on Examples
- Learning how to use the University's HPC Cluster

Why GPUs?



GPUs offer much **higher instruction throughput** and **memory bandwidth** than CPUs within a similar price and power envelope, making them ideal for applications that require high parallelism. Unlike CPUs, which excel at executing a few threads sequentially, GPUs are designed to handle **thousands of threads simultaneously**, achieving greater throughput by focusing more on data processing rather than data caching and flow control.

Input: an array, Output: squared



```
length_of_array = 1024;  
for(int i = 0; i < length_of_array; i++) {  
    out[i] = in[i] * in[i];  
}
```

```
__global__ void square(float * d_out, float * d_in) {  
    int idx = threadIdx.x; // threadIdx is a cuda built-in variable  
    float f = d_in[idx];  
    d_out[idx] = f * f;  
}
```

Execution time = $1024 \times 2 \text{ ns} = 2048 \text{ ns}$

Execution Time = $10 \text{ ns} + 2 \times (\text{Data Transfer Overhead}) + (\text{Kernel Launch Overhead})$

Programming Assignment #1



- Get the given assignment code running, which are for CPU, and GPU

[CUDA_for_ITU/assignments/01-CPU_GPU_difference at main · ehsanyousefzadehasl/CUDA_for_ITU](#)

- Read the code carefully to understand its functionality. Then, experiment with different input arguments and observe the differences in execution time between the CPU and GPU implementations.

Number of elements in the input array	CPU Time	GPU Time

CUDA

- Compute Unified Device Architecture
- CUDA C/C++
 - Based on standard C/C++
 - Set of extensions enabling heterogeneous programming
- Pre-requisites:
 - Experience with C/C++

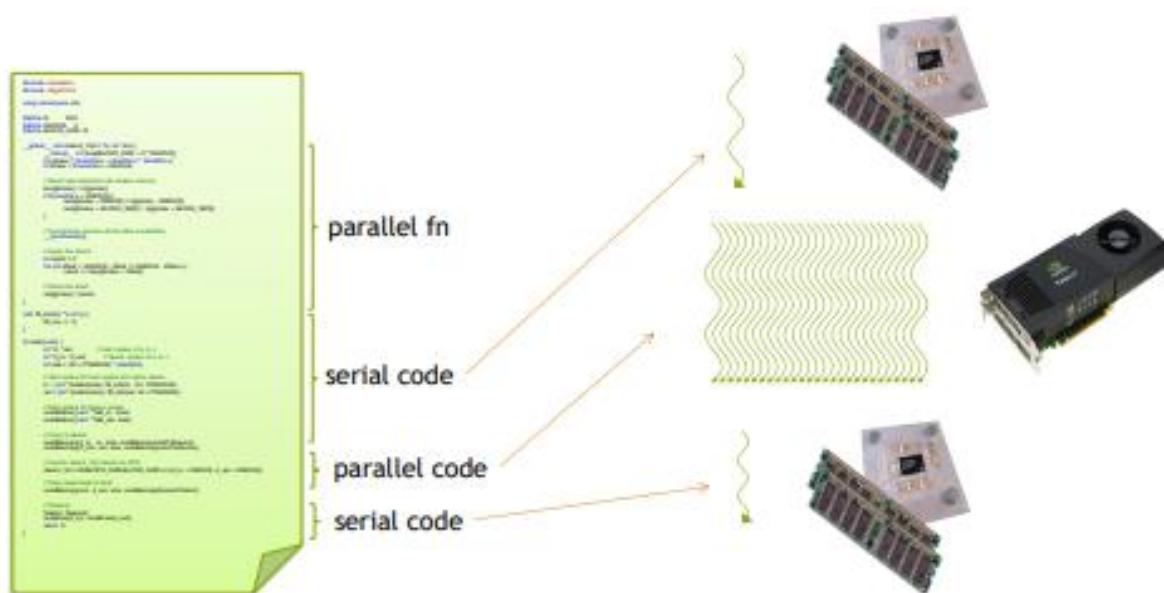
Heterogeneous Computing

- Terminology:

- **Host** The CPU and its memory (host memory)
- **Device** The GPU and its memory (device memory)

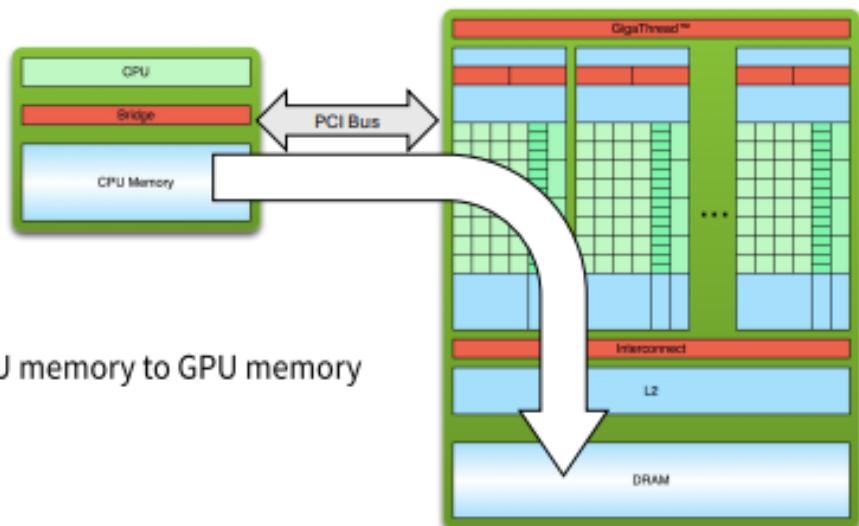


Heterogeneous Computing



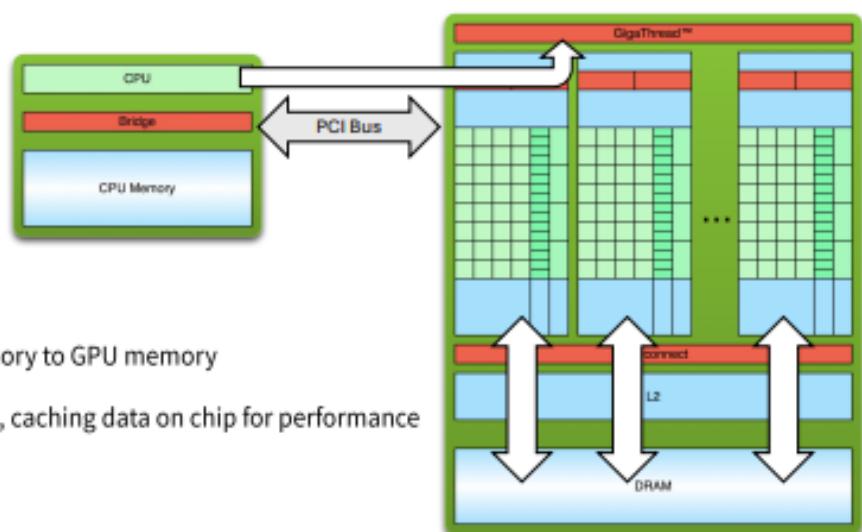
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory

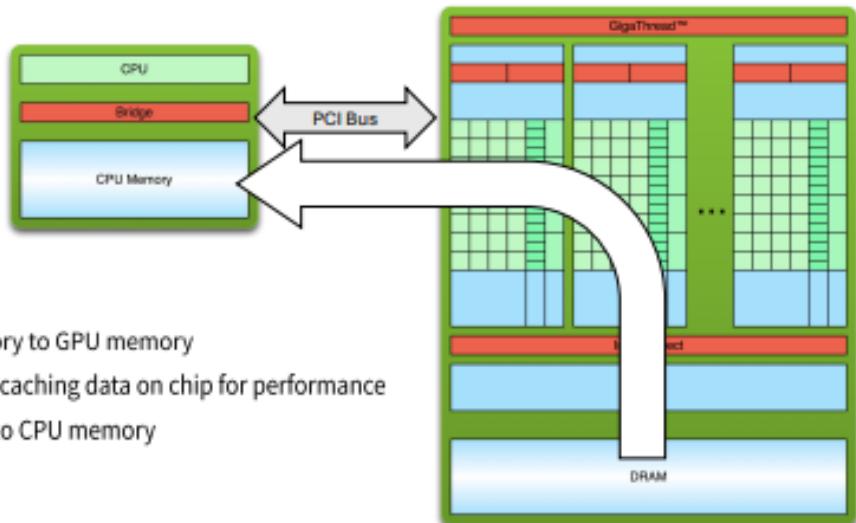


Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance



Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Hello World!



- NVIDIA compiler (`$nvcc`) can be used to compile programs with no device code
- Standard C "hello world" program!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

```
$ nvcc 00-hello_world.cu -o 00-hello_world
```

```
__global__ void helloWorld(void) {
    printf("Hello from thread %d from block %d\n",
        threadIdx.x, blockIdx.x);
}

int main(void) {
    mykernel<<<10,100>>>();

    return 0;
}
```

Hello World! with Device Code

- CUDA C/C++ keyword `_global_` indicates a function that:
 - Runs on the device
 - Is called from host code
- nvcc separates source code into host and device components
 - Device functions (e.g. `helloWorld()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler

Hello World! with Device Code

```
helloWorld<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - Parameters (1,1) indicate the number of thread block and the number of threads in each thread block
- That’s all that is required to execute a function on the GPU!

Adding two Arrays

- We have two input arrays: A, and B
- Goal: calculate $C = A + B$ with GPU
- How do we usually do it with CPU?
 - Loops
 - SIMD instructions

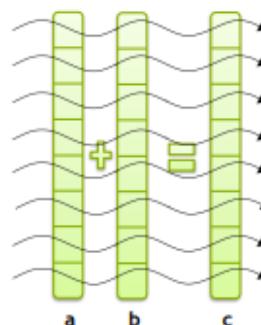


CPU SIMD Instructions



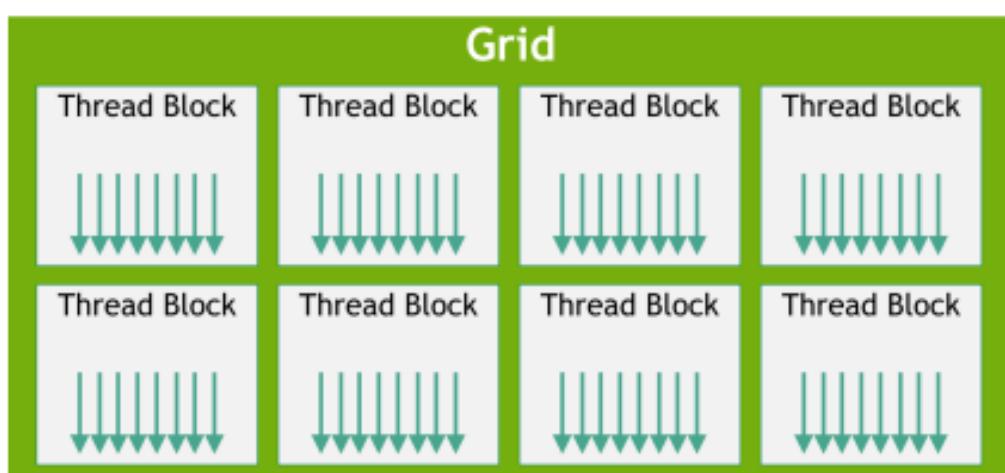
SIMD (Single Instruction, Multiple Data) Instructions: SIMD is a type of parallel processing in CPUs where a single instruction is executed **simultaneously** on multiple pieces of data. This is particularly useful for operations like **vector** and **matrix computations**, **image processing**, or any tasks that involve performing the same operation on large datasets. In SIMD, data is stored in vectors (arrays of elements), and special SIMD registers process multiple elements in parallel. For example, a CPU with 256-bit SIMD registers can process eight 32-bit numbers or sixteen 16-bit numbers at once. This allows significant performance improvements by leveraging **data-level parallelism**. Modern CPUs provide SIMD extensions like **Intel's SSE and AVX**, or **ARM's NEON**, which are designed to optimize workloads in fields like **scientific computing**, **multimedia processing**, and **machine learning**. SIMD helps CPUs handle tasks that require **high throughput**, enabling **faster computations** compared to processing data sequentially.

Adding two Arrays



- Having an independent thread
 - For each add operation
1. Arrays on the CPU (host) memory
 2. Allocate memory on GPU (device) memory
 3. Copy arrays to device memory
 4. Launch the kernel (pass it the arguments)
 5. Copy back the results from device memory to host memory

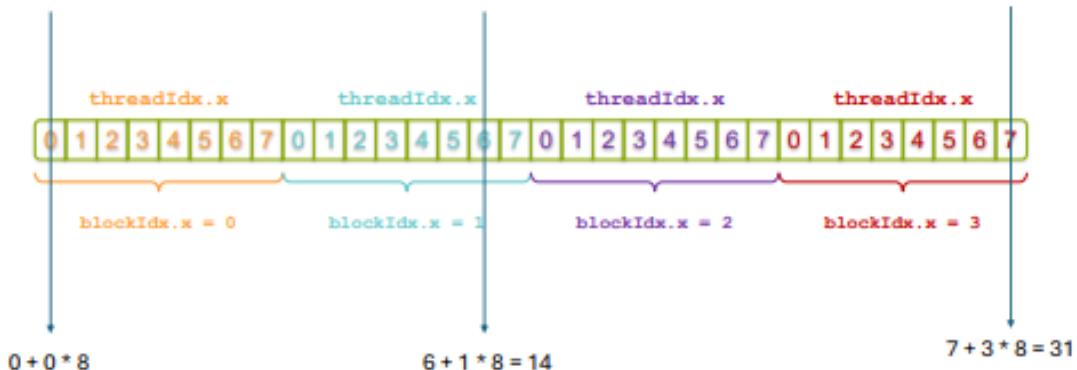
Terminology



The logic behind the add we checked!

- Assume $\text{add} \lll 4, 8 \ggg (\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{N})$

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```



Basic device (GPU) memory management

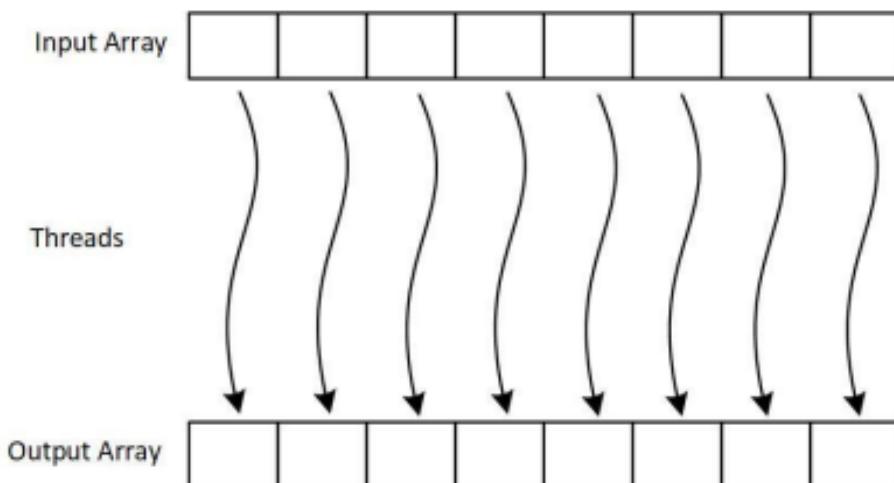
- `cudaMalloc()`
- `cudaMemcpy()`
- `cudaFree()`

Parallel Communication Patterns

- Parallel computing is about many threads solving a problem by working together. The key to working together is **communication**. In CUDA, communication takes place through **memory**.
- There are different kinds of parallel communication patterns and they are about how to map tasks (threads) and memory.
- Some of the important patterns: **map**, **gather**, **scatter**, **stencil**, **transpose**

Map

The example of calculating the squared value of each element.



Map



- The map parallel pattern is a fundamental concept in parallel programming where the same operation is independently applied to every element of a dataset. It is an embarrassingly parallel pattern, meaning there are no dependencies between elements, so all operations can be executed in parallel without requiring communication between threads.
- In CUDA programming, the map pattern is often implemented by assigning one thread to process one or more elements of the dataset. The map pattern is particularly efficient because it allows maximum utilization of GPU cores by distributing work evenly across threads.

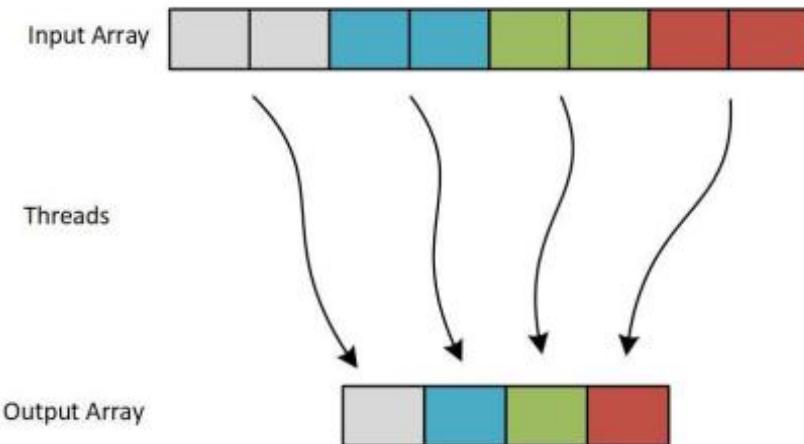
Applications of Map



- **Image Processing:** Applying filters (e.g., grayscale conversion, brightness adjustment) where each pixel can be processed independently.
- **Scientific Simulations:** Computing mathematical functions (e.g., sine, cosine, or exponential) for large arrays of input values.
- **Data Transformation:** Converting or normalizing datasets, such as scaling numerical values or applying logarithmic transformations.
- **Graphics Rendering:** Transforming vertex coordinates or applying color transformations in GPU-accelerated graphics.
- **Machine Learning:** Element-wise activation functions (e.g., ReLU, sigmoid) applied to neural network layers.

The map pattern's simplicity and lack of inter-thread communication make it a highly efficient and scalable approach for parallelizing independent computations.

Gather

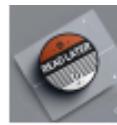


Gather



The **gather parallel pattern** is a common approach in parallel programming where data is **collected from multiple memory locations into a single output dataset**. Each thread retrieves data from one or more indices of the input dataset and performs operations to produce its corresponding result. Unlike the map pattern, the gather pattern often involves **non-contiguous memory accesses**, as threads may need to access scattered input locations. In CUDA, implementing an efficient gather pattern requires **careful memory management** to minimize uncoalesced global memory access.

Applications of Gather

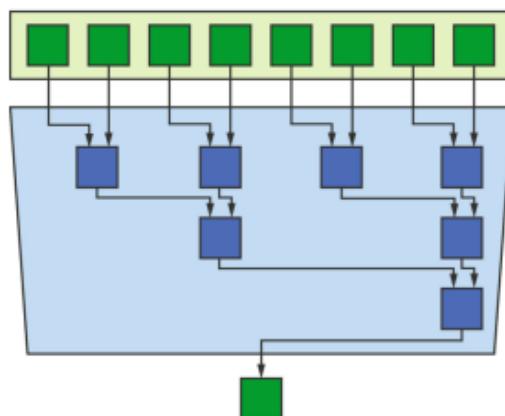


The gather pattern is frequently used in tasks where specific data needs to be extracted or rearranged:

1. **Matrix Operations:** Extracting rows, columns, or specific elements for sub-matrix computations.
2. **Image Processing:** Sampling data from scattered pixel locations (e.g., texture mapping, image warping).
3. **Scientific Simulations:** Collecting data points from irregular grids or domains for further processing.
4. **Data Rearrangement:** Reorganizing datasets (e.g., shuffling, grouping, or sorting by specific criteria).
5. **Graphics and Rendering:** Gathering vertex or texture data for 3D transformations or rendering pipelines.

The gather pattern's flexibility makes it ideal for handling irregular or scattered datasets, though optimizing memory access patterns is crucial to achieve high performance on GPUs.

Reduce



Reduce



The **reduce parallel pattern** involves **combining elements of a dataset into a single result** using a specified operation, such as summation, multiplication, or finding the maximum. In CUDA, this pattern is typically implemented by assigning threads to process parts of the dataset and then performing a **hierarchical reduction** in shared memory, where partial results are iteratively combined until only one result remains. Reduction is a key pattern in parallel programming as it efficiently aggregates data while minimizing global memory accesses.

Applications of Reduce

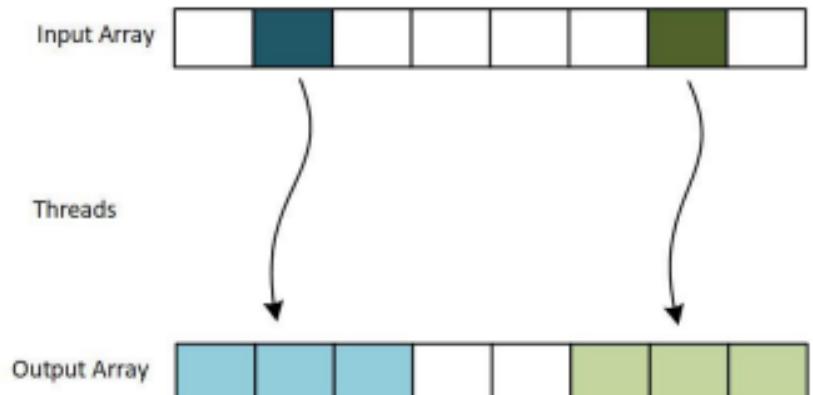


1. **Scientific Simulations:** Summing physical quantities (e.g., energy, mass) or calculating averages over large datasets.
2. **Data Analytics:** Computing metrics like totals, maximums, minimums, or variance across datasets.
3. **Machine Learning:** Summing gradients during backpropagation or aggregating results in distributed computations.
4. **Graphics:** Calculating light intensity or pixel averages in rendering pipelines.
5. **Financial Modeling:** Aggregating transaction data for totals or risk analysis.

The reduce pattern is essential for summarizing large datasets efficiently, with shared memory and synchronization ensuring optimal performance on GPUs.

Scatter

- Tasks compute where to write output.
- An example is **sorting numbers** in an array because each thread will compute where to write its output.
- Note that just two threads are shown to clearly demonstrate the pattern.



Scatter



The **scatter parallel pattern** distributes data from a single input dataset to **specific locations in an output dataset**. Each thread takes a portion of the input and writes it to one or more indices in the output, often based on a mapping or index array. Unlike the gather pattern, which collects data, scatter focuses on **placing data in non-contiguous or irregular memory locations**. In CUDA, efficient scatter implementations require careful handling of memory writes to avoid **race conditions** and ensure coalesced access when possible.

Applications of Scatter



1. **Sorting Algorithms:** Writing elements to their correct positions in a sorted array.
2. **Sparse Matrix Representations:** Distributing values into specific non-zero locations in sparse matrices.
3. **Graphics and Rendering:** Assigning texture data or transforming vertices into frame buffers.
4. **Data Partitioning:** Splitting datasets into groups or buckets based on a condition or key.
5. **Simulation and Modeling:** Distributing particle properties (e.g., position, velocity) into spatial grids.

Scatter is vital for tasks requiring flexible data placement, with synchronization and memory optimization critical for high performance in GPU implementations.

Race Conditions



A race condition occurs when multiple threads in a parallel program attempt to access and modify the same memory location simultaneously, leading to undefined or incorrect results. This happens because threads execute independently, and without proper synchronization, one thread may overwrite the changes made by another. In CUDA programming, race conditions are common when threads in a block write to shared memory or when multiple threads across blocks write to the same global memory address. To avoid race conditions, developers can use atomic operations, which ensure only one thread modifies a memory location at a time, or synchronization mechanisms like `__syncthreads()` within blocks to control access to shared resources.

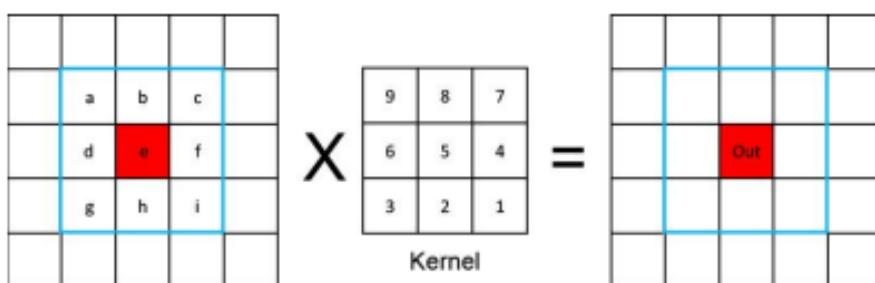
Coalesced Memory Access



Coalesced memory access occurs when threads in a warp access consecutive memory addresses, allowing the GPU to combine those requests into a single transaction with global memory. This is highly efficient because it minimizes the number of memory transactions and maximizes memory bandwidth utilization. Coalesced access is especially important in CUDA programming since uncoalesced accesses (e.g., scattered or misaligned memory requests) result in higher latency and slower performance. Ensuring coalesced access often involves structuring data in memory and aligning thread-to-data mappings so that each thread accesses a unique, contiguous address in global memory. This optimization is crucial for achieving high performance on GPUs.

Stencil

- Tasks read input from a fixed neighborhood in an array.
- Convolution operations in Convolution Neural Networks (CNNs).



Stencil



The stencil parallel pattern is used for computations where each element of an output dataset depends on a specific element in the input dataset and its neighbors. Threads independently process their assigned element and access neighboring values, often defined by a fixed radius. This pattern typically involves regular grid structures and is highly parallelizable, making it well-suited for GPU acceleration. Efficient stencil implementations use shared memory to cache data and reduce redundant global memory accesses.

Applications of Stencil



- 1. Image Processing:** Applying convolution filters (e.g., Gaussian blur, edge detection) where each pixel is updated based on its neighbors.
 - 2. Scientific Simulations:** Modeling physical phenomena like heat diffusion, wave propagation, or fluid dynamics.
 - 3. Finite Difference Methods:** Solving partial differential equations (PDEs) using neighbor-based calculations.
 - 4. Computational Physics:** Updating grid cells in simulations like cellular automata or particle interaction grids.
 - 5. Weather and Climate Models:** Simulating environmental conditions using grid-based data, such as temperature or pressure updates.
- The stencil pattern is essential for tasks involving local interactions, with shared memory and efficient boundary handling playing key roles in optimizing performance.

1D Stencil Example

- Consider applying a 1D stencil to a 1D array of elements
 - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:

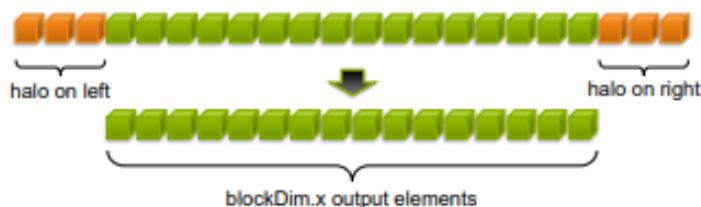


1D Stencil Example

- Each thread processes one output element
 - `blockDim.x` elements per block
- Input elements are read several times
 - With radius 3, each input element is read seven times
- Within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block. Data is not visible to threads in other blocks

1D Stencil Example

- Cache data in shared memory
 - Read ($\text{blockDim.x} + 2 * \text{radius}$) input elements from global memory to shared memory
 - Compute blockDim.x output elements
 - Write blockDim.x output elements to global memory
 - Each block needs a **halo** of radius elements at each boundary



Shared Memory



Shared memory is a small, fast memory space shared by all threads in a block. It is significantly faster than global memory because it resides on-chip (inside the GPU core). Shared memory allows threads within a block to collaborate by sharing data, which can lead to substantial performance improvements by:

- **Reducing Redundant Memory Accesses:**
 - If multiple threads need the same data, they can share it in shared memory instead of each thread fetching it separately from global memory.
- **Minimizing Global Memory Latency:**
 - Global memory access is slow compared to shared memory. Using shared memory avoids repeatedly accessing global memory for frequently used data.
- **Enabling Thread Collaboration:**
 - Threads can share intermediate results via shared memory, making it easier to implement cooperative algorithms.

Programming Assignment #2



- Develop a CUDA kernel that performs the same **1D stencil operation** without using shared memory. Next, design and execute an experiment to compare the performance of the shared memory version with the non-shared memory version. Highlight the advantages of shared memory by demonstrating reduced global memory accesses and improved execution time in the shared memory approach, especially for large input sizes (compare the time).

Where to experiment and learn?

- **Option 1:** If you have NVIDIA GPU on your laptop or PC at home
 - Do your experiment and exploration on it, it is faster
- **Option 2:** ITU's HPC cluster
 - You need to submit tasks to it, and you might also wait!

ITU's HPC Cluster



- The cluster is a shared computing resource used by **students** and **staff** for running computational tasks, simulations, and data processing workloads, etc. efficiently. It is designed to handle **multiple users simultaneously** while ensuring **fair access** to resources.
- The cluster uses **SLURM** (Simple Linux Utility for Resource Management) as the scheduler. SLURM is responsible for: **Assigning tasks to available compute nodes** in the cluster. **Managing queues** to ensure tasks are executed in the appropriate order based on **priority** and **resource availability**.

[ITU HPC](#)

HPC.ITU.DK ITU HPC Documentation

How to login!

Step 1:

```
$ ssh your_username@hpc.itu.dk
```

Step 2:

Enter your password:

EXPLORE AROUND AND SEE WHAT WE HAVE ON THE CLUSTER!

How to submit a task!



```
#!/bin/bash

#SBATCH --job-name=cuda_test_job_name           # Job name
#SBATCH --output=cuda_test_output_name          # output file name
#SBATCH --cpus-per-task=1                        # Schedule 8 cores (includes hyperthreading)
#SBATCH --gres=gpu                               # Schedule a GPU, it can be on 2 gpus like gpu:2
#SBATCH --time=00:05:00                           # Run time (hh:mm:ss) - run for one hour max
#SBATCH --partition=scavenge                     # Run on either the Red or Brown queue

module load CUDA/12.1.1

nvcc test_cuda.cu -o test_cuda
./test_cuda
```

Programming Assignment #3

- Compute the dot product of two vectors using CUDA and compare it to the one available in the repository that uses shared memory (add the timing to both source codes). Experiment with different length of the input array, different block size.
- Write your reflection on how using shared memory improves the performance.

[ehsanyousefzadehasl/CUDA_for_ITU](#)