

Exercise Session 14

Systems Programming and Computer Architecture

Devices

Fall Semester 2024

Disclaimer

- **Website:** n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub:** github.com/falkbe
- **Kahoots:** now on website n.ethz.ch/~falkbe/
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Overview

- Lecture Recap: Devices
- SPCA in a nutshell
- Exam Remarks
- SPCA in perspective
- Questions

Devices

Devices and Device Drivers

Devices

- **Devices:** Device Registers/Dealing with caches
- **Device Driver:** Operating System Part of the device

Devices

- **Devices:** Device Registers/Dealing with caches
- **Device Driver:** Operating System Part of the device

SPCA Devices

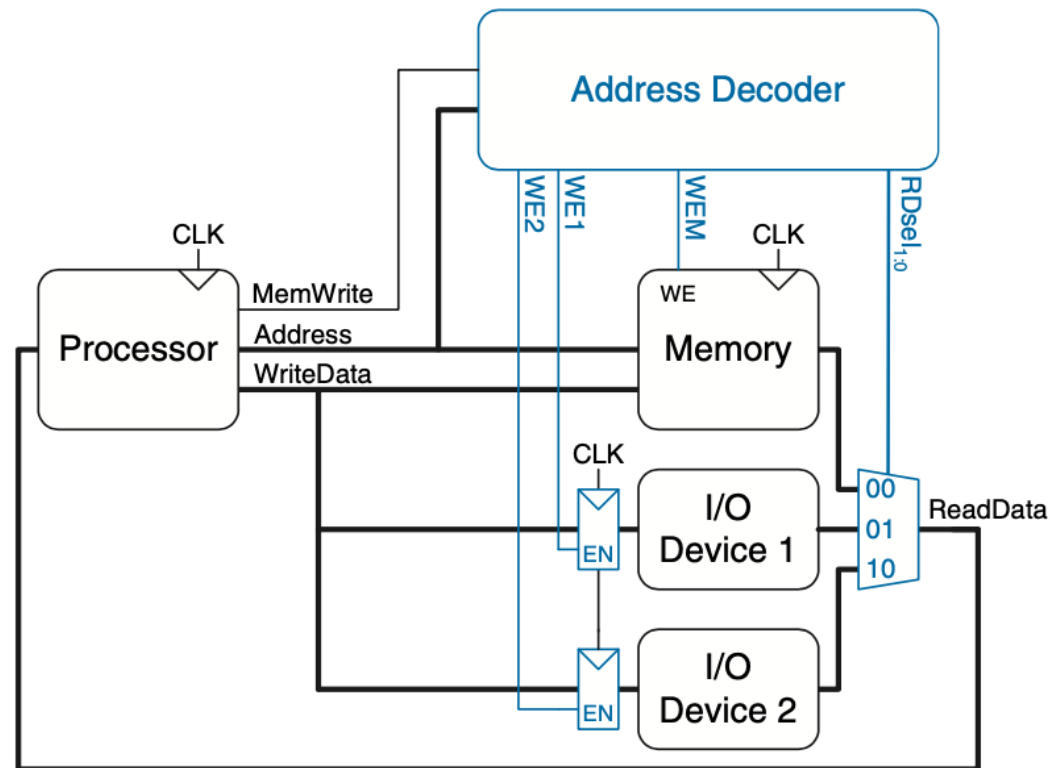
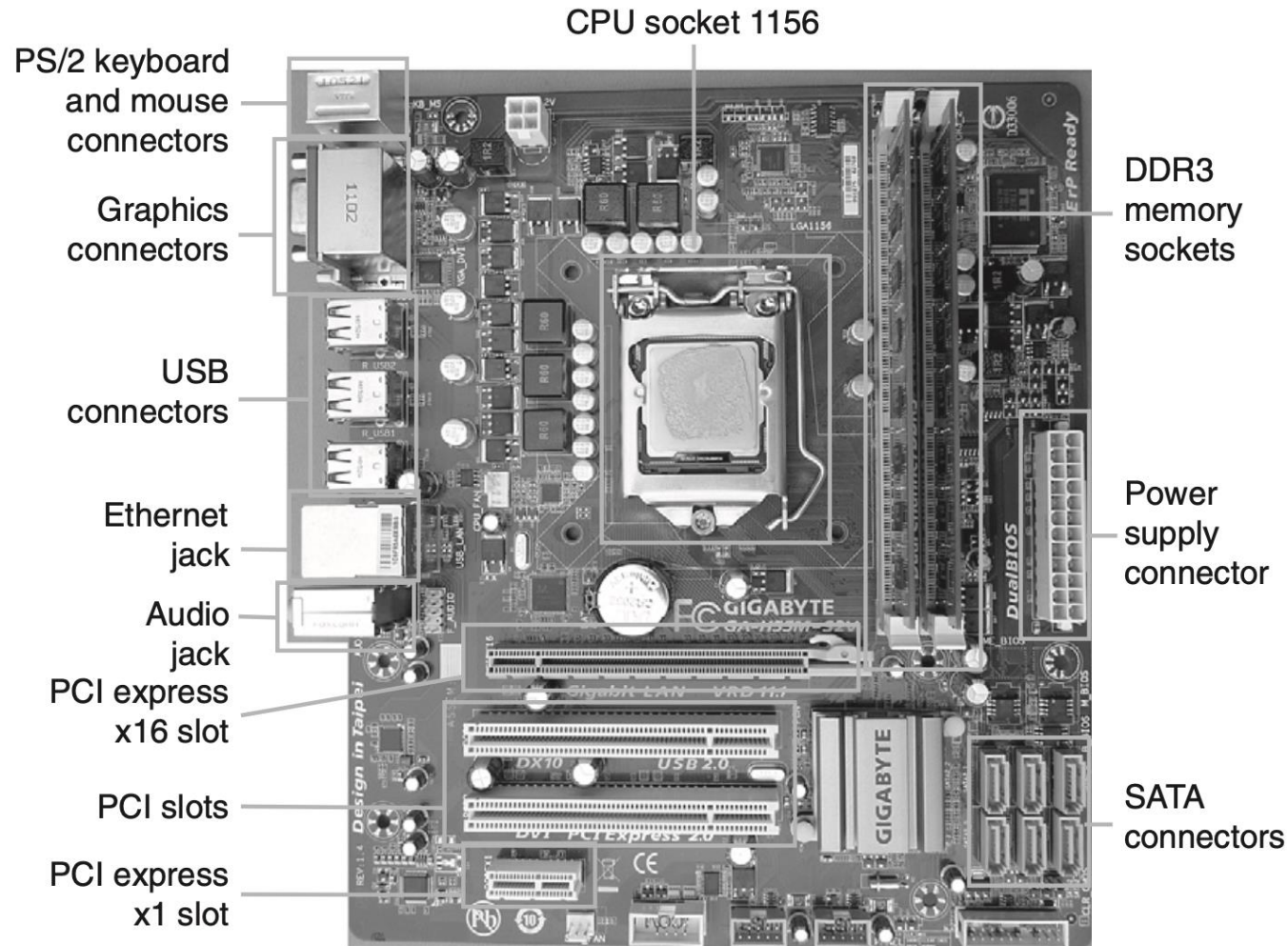


Figure 8.28 Support hardware for memory-mapped I/O

SPCA Devices



Devices

- **Input/Output (I/O) systems** are used to connect a computer with external devices called **peripherals** (keyboards, monitors, etc.)
- Processor accesses an I/O device using the address/data busses the same way as it accesses memory
- Part of **address space** is **dedicated to I/O** devices rather than memory: a store sends data to the device, with a load we receive data from the device
- => This method of communicating is called **memory mapped I/O**

Devices

What is a device?

Specifically, to an OS programmer:

- Piece of hardware visible from software
- Occupies some location on a **bus**
- Set of **registers**
 - Memory mapped or I/O space
- Source of **interrupts**
- May initiate **Direct Memory Access** transfers

Devices

Registers

- CPU can **load from** device registers:
 - Obtain status info
 - Read input data
- CPU can **store to** device registers:
 - Set device state and configuration
 - Write output data
 - Reset states

Devices

Addressing registers

1. Memory mapped:

- Registers appear as memory locations
- Access using loads/stores (`movb/movw/movl/movq`)

2. “I/O instructions”:

- Different (16 bit) address space for older I/O devices
- Specific (these days) to x86 architecture
- Special instructions: `inb`, `outb`, etc.

Devices

Registers are not memory

Device registers don't behave like RAM!

- Register contents change without writes from CPU
 - Status words
 - Incoming data
- Writes to registers are used to trigger actions
 - Sending data
 - Resetting state machines

Devices

Dealing with caches

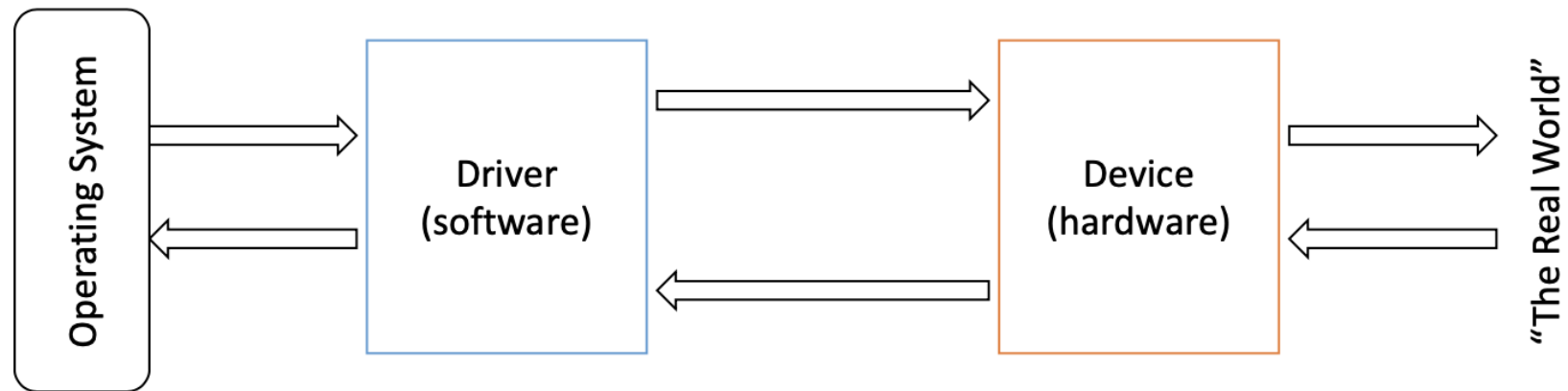
- Reads can't come from the cache
 - Register value changes \Rightarrow cache becomes **inconsistent**
 - Write-back caches (and write buffers) cause problems
 - You don't know when the line will be written
 - Reads and writes cannot be combined into cache lines
 - Registers might require single word or byte writes only
 - Line-size writes stomp on other registers
 - Even spurious reads trigger device state changes
- \Rightarrow Device register access **must** bypass the cache
- Handled in the MMU: PTEs have "no cache" flag
 - I/O space access isn't cached anyway

Devices

- **Devices:** Device Registers/Dealing with caches
- **Device Driver:** Operating System Part of the device

Devices

Basic model



Devices

Very simple UART driver

```
#define UART_BASE 0x3f8
#define UART_THR (UART_BASE + 0)
#define UART_RBR (UART_BASE + 0)
#define UART_LSR (UART_BASE + 5)

void serial_putc(char c)
{
    // Wait until FIFO can hold more chars
    while( (inb(UART_LSR) & 0x20) == 0);
    // Write character to FIFO
    outb(UART_THR, c);
}

char serial_getc()
{
    // Wait until there is a char to read
    while( (inb(UART_LSR) & 0x01) == 0);
    // Read from the receive FIFO
    return inb(UART_RBR);
}
```

Register addresses from
data sheet
0x3f8: location on a PC

Send a character (wait
until we can first)

Read a character (spin
waiting until one is there
to read)

Devices

Very simple UART driver

- Actually, far too simple!
 - But this is how the first version always looks...
- No initialization code, no error handling.
- Uses **Programmed I/O** (PIO)
 - CPU explicitly reads and writes all values to and from registers
 - All data must pass through CPU registers
- Uses **polling**
 - CPU polls device register waiting before send/receive
 - Tight loop!
 - Can't do anything else in the meantime
 - Although could be extended with threads and care...
 - Without CPU polling, no I/O can occur

Devices

- Each device operates differently: Different control register layouts, Unique set of commands, etc.
- **Device driver hides** these details from the OS: **provides a standardised API** (e.g. `send_packet()`, `read_block()`) for OS
- The OS (the OS's device driver) is responsible for initiating DMA transfers: allocates memory, configures the DMA controller

Devices

Other challenges

1. How to avoid polling all the time?
 - How does the CPU know when the device is ready, or finished?
 - Solution: **interrupts**
2. How to avoid the CPU copying all the data?
 - Can the CPU get on with something else?
 - Solution: **direct memory access** (DMA)
3. Where do these register locations come from?
 - How can the OS find devices in the physical address space?
 - How are the physical addresses allocated?
 - Solution: **discoverable buses** (e.g. PCI)

Devices

DMA, Shared Memory, PCIe

Devices

- **DMA** (Direct Memory Access): Copies data for CPU
- **Shared Memory**: Buffer/Descriptor Rings
- **PCIe** (Peripheral Component Interconnect): Finding address space for the devices

Devices

- **DMA** (Direct Memory Access): Copies data for CPU
- **Shared Memory**: Buffer/Descriptor Rings
- **PCIe** (Peripheral Component Interconnect): Finding address space for the devices

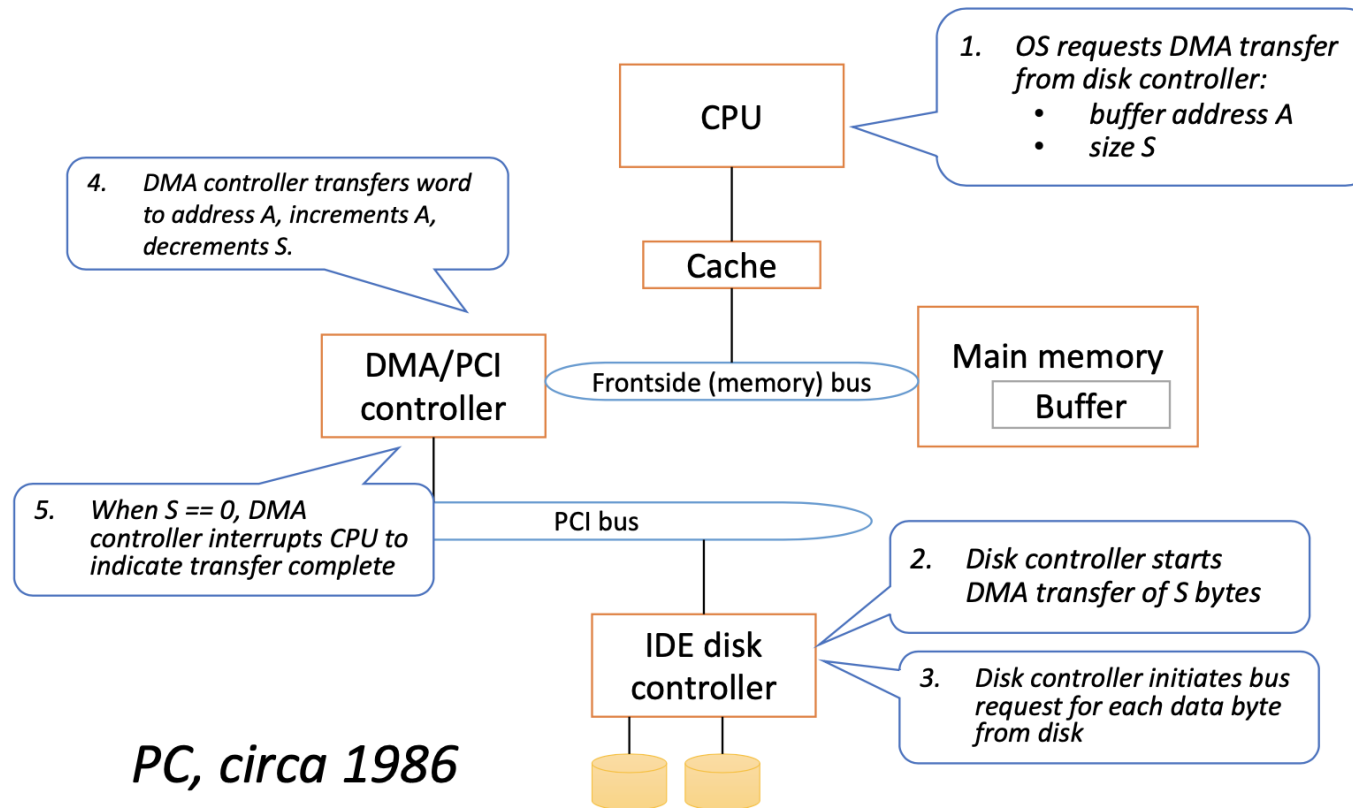
Devices

Direct Memory Access

- Avoid *programmed I/O* for lots of data
 - E.g. fast network or disk interfaces
- Requires *DMA controller*
 - Generally built-in these days
- Bypasses CPU to transfer data directly between I/O device and memory
 - Doesn't take up CPU time
 - Can save memory bandwidth
 - Only one interrupt per transfer

Devices

Very simple DMA transfer



PC, circa 1986

Systems Programming 2023 Ch. 21: Devices

Devices

DMA and Caches

- DMA means memory becomes **inconsistent** with CPU caches
- Options:
 1. CPU can map DMA buffers **non-cacheable**
⇒ large hit – probably wants to process data anyway
 2. Cache can “**snoop**” DMAC bus transactions
(but doesn't scale beyond small SMP systems)
 3. OS can **explicitly flush/invalidate** cache regions
⇒ cache management important part of device drivers!
- **Idea:** add the DMA device to our cache coherency protocol

Devices

DMA and Virtual Memory

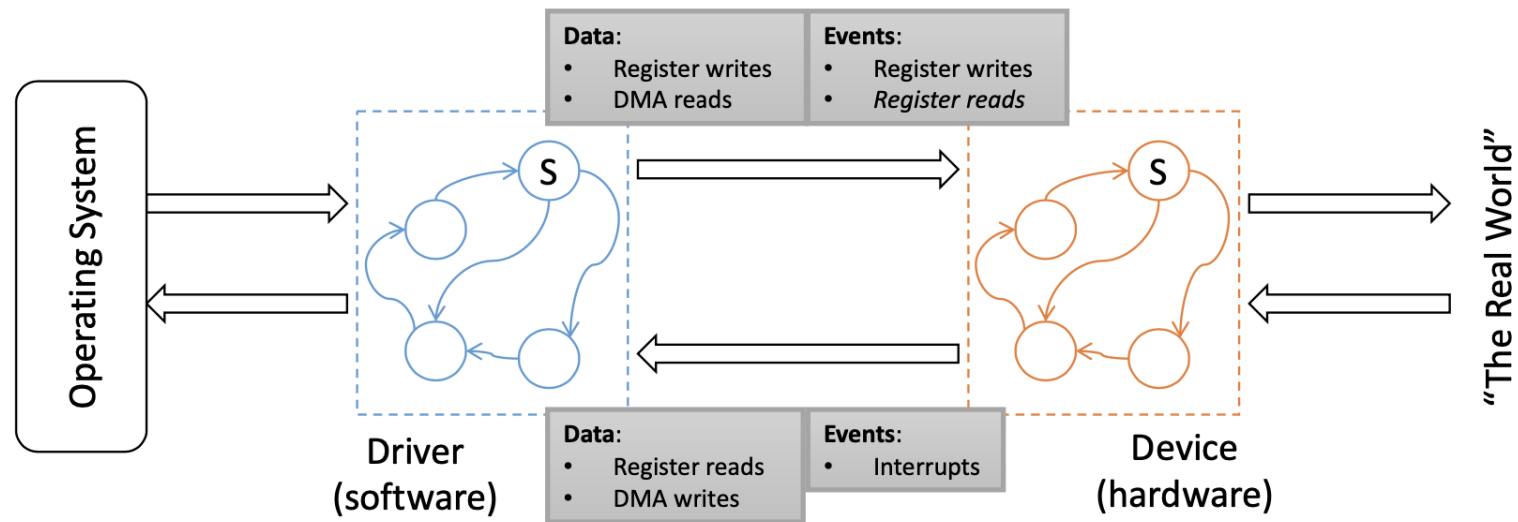
- DMA addresses are **physical**
 - Appear on external bus
- User and OS code deal with **virtual** addresses (mostly)
- OS (and device drivers) must manually translate virtual ↔ physical addresses when programming DMA controllers
 - This can require more than just a hardware page table!
 - DMA of a single virtual address region might **not be contiguous** in physical address space
 - **Scatter-gather** DMA controllers: DMA to/from a list of regions
- Newer systems: provide an **IOMMU**
 - Works like an MMU, but for DMA writes from devices
 - Must still be programmed by OS to match MMU state
 - Has all kinds of other interesting uses – beyond scope of this course!

Devices

- **DMA** (Direct Memory Access): Copies data for CPU
- **Shared Memory**: Buffer/Descriptor Rings
- **PCIe** (Peripheral Component Interconnect): Finding address space for the devices

Devices

Basic model

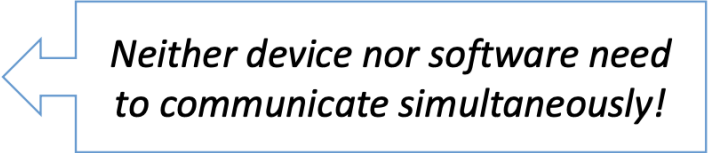


- Driver and device are both *state machines*
- *Data* must be transferred between them
- *Events* signal state transitions

Devices

Device \leftrightarrow CPU communication

1. Writing a device register
 - CPU \rightarrow device, synchronous
2. Reading a device register
 - CPU \leftrightarrow device, synchronous
3. Device requests interrupt
 - Device \rightarrow CPU, synchronous
4. **Shared memory**
 - CPU writes to memory, DMA reads
 - DMA writes to memory, CPU reads
 - **Asynchronous**

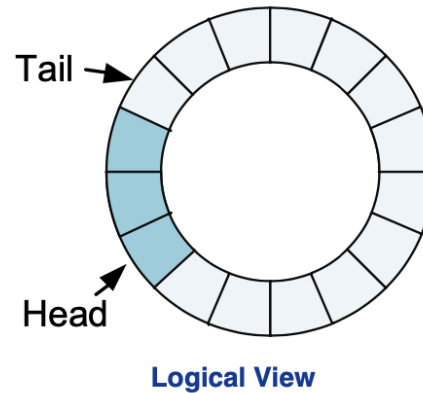


Neither device nor software need to communicate simultaneously!

Devices

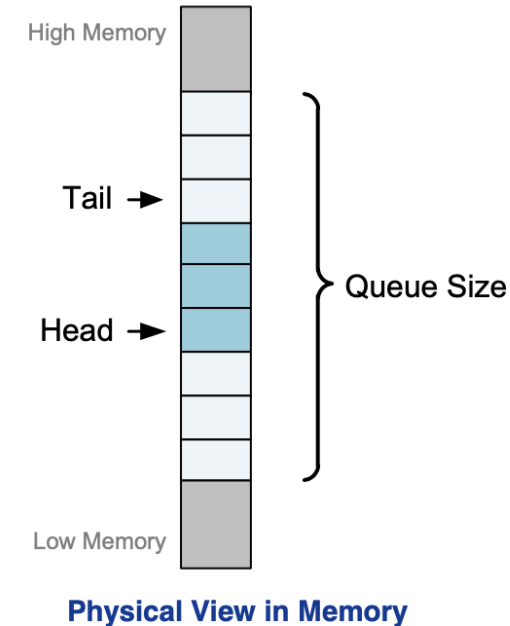
Buffer (or descriptor) rings

Example for transmit ring:
The device reads from head
and the OS adds to the tail.



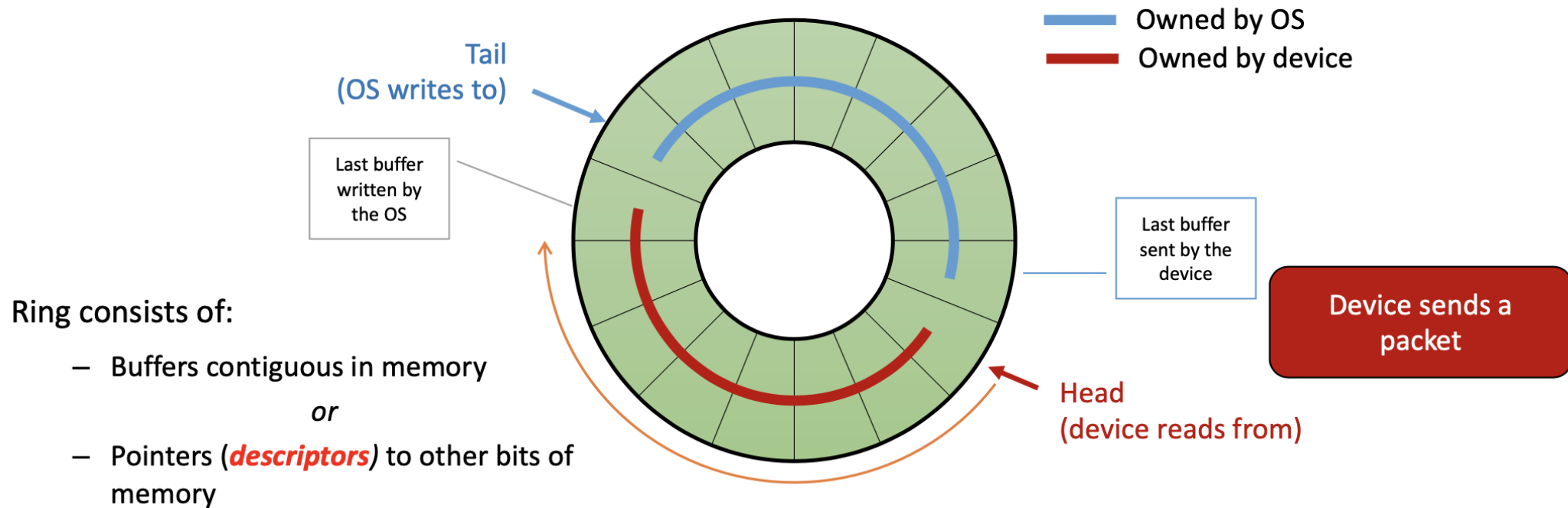
Ring consists of:

- Buffers contiguous in memory
- or
- Pointers (**descriptors**) to other bits of memory



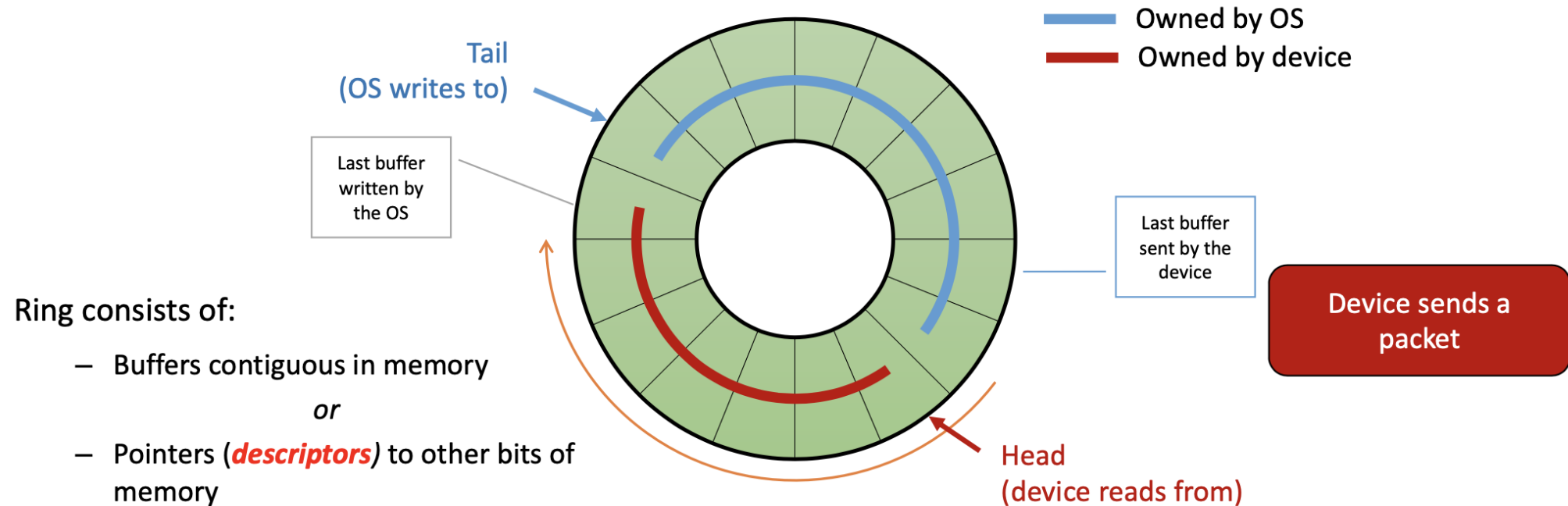
Devices

Buffer (or descriptor) rings (for transmit)



Devices

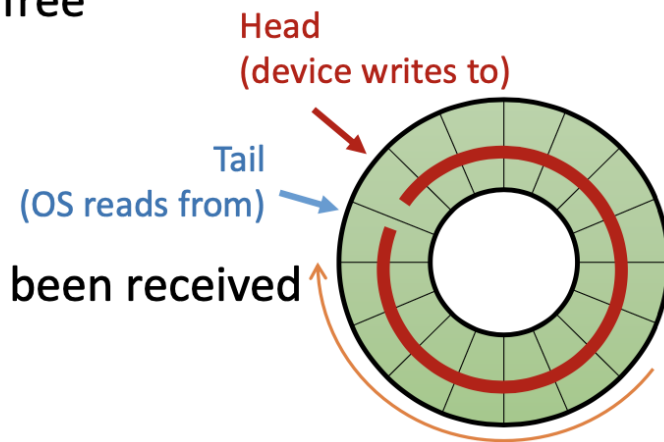
Buffer (or descriptor) rings (for transmit)



Devices

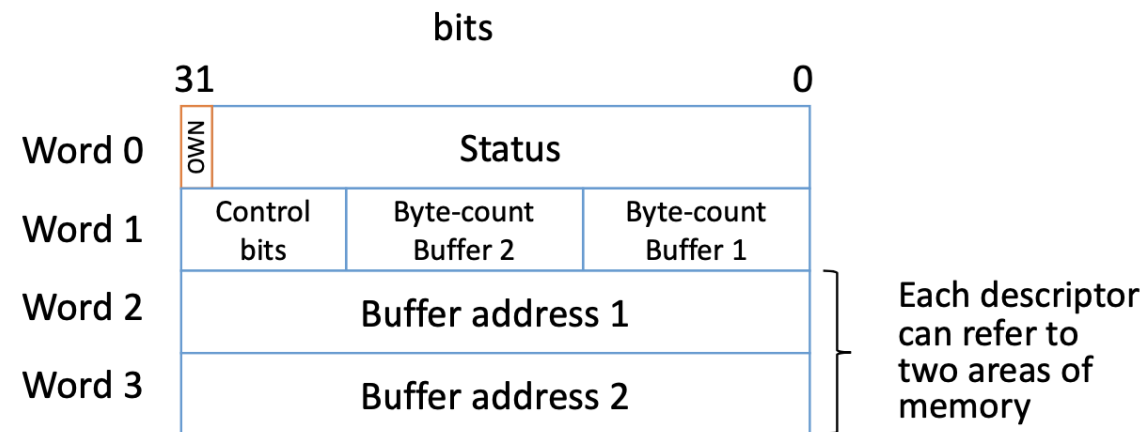
Overruns and underruns (receive)

- Device has no buffers for received packets
⇒ starts discarding packets
 - Not as bad as it sounds
 - Will start copying them to memory when a buffer is free
 - Signals that it's lost some in a status register
- CPU reads all received packets ⇒ it must wait
 - Can spin polling, but inefficient
 - Signals device to interrupt it when a new packet has been received
 - Goes off to do something else



Devices

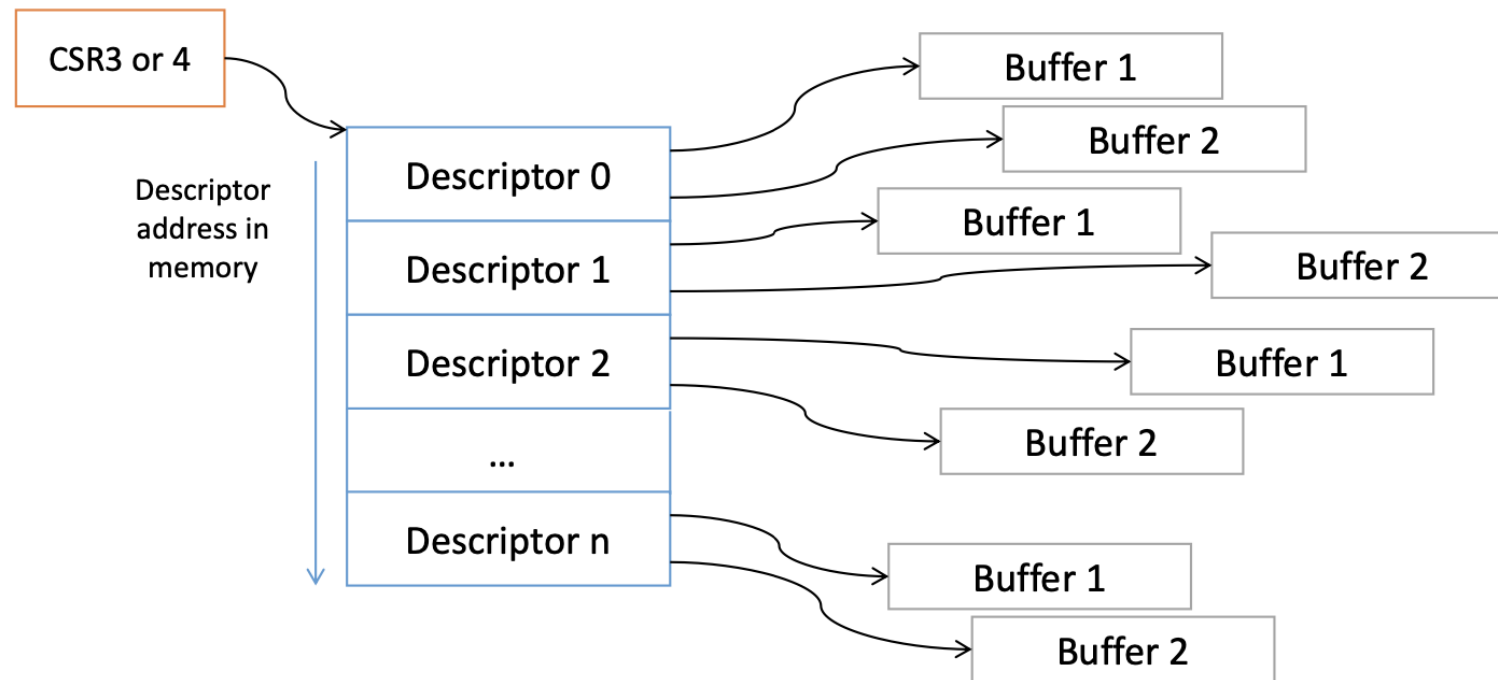
Tulip descriptors



The Tulip has 2 rings of descriptors in main memory -
One for transmit, one for receive

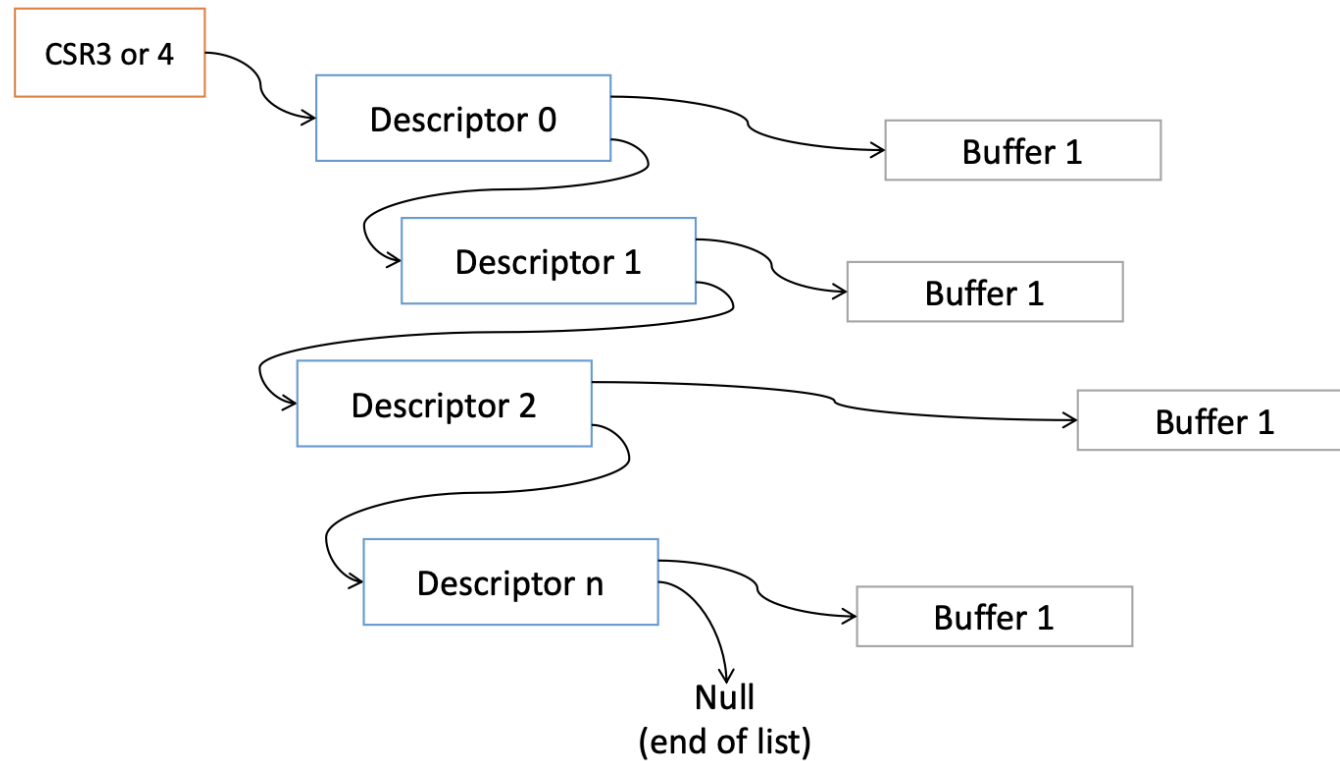
Devices

Descriptor rings



Devices

Descriptor rings – chain mode



Devices

- **DMA** (Direct Memory Access): Copies data for CPU
- **Shared Memory**: Buffer/Descriptor Rings
- **PCIe** (Peripheral Component Interconnect): Finding address space for the devices

Devices

PCI is...

Peripheral **C**omponent **I**nterconnect

- An electrical standard for connecting devices
 - As is PCMCIA, PCI-X, PCI-Express, etc.
- A standard for physical connectors
- A set of “bus protocols” for communication between devices
- A software-visible interface to I/O hardware

PCIe has succeeded PCI, but extends the same software-visible interface

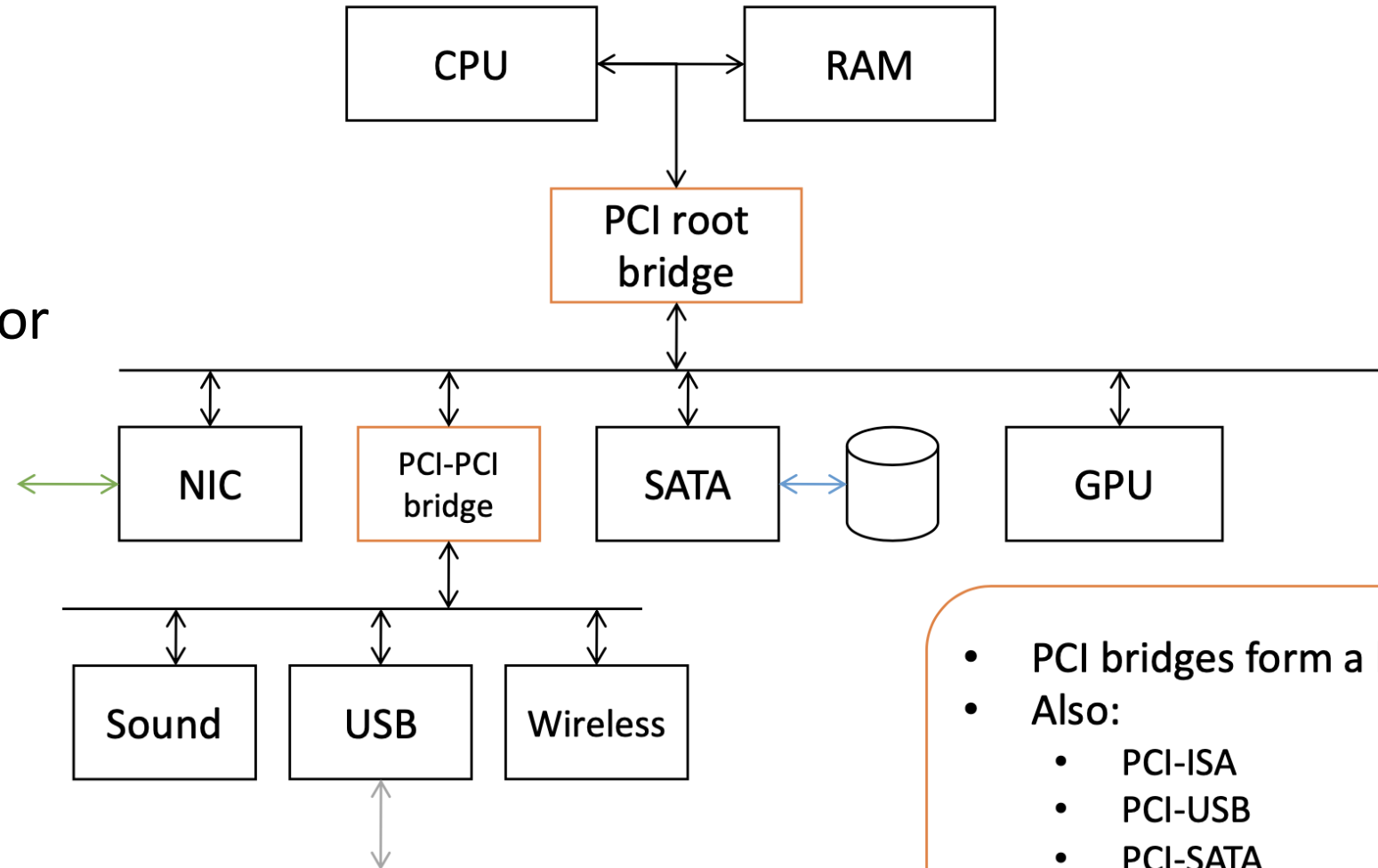
Devices

PCI tries to solve many problems:

- Device discovery
 - Finding out which devices are in the system
- Address allocation
 - Which addresses should each device's registers appear at?
- Interrupt routing
 - Which interrupt signals from the device should map to which exception vectors?
- Intelligent DMA
 - “Bus mastering” devices no longer need a DMA controller

Devices

- **PCI Address space is flat:** each device asks for set of address range
- **Result:** each device appears as a set of contiguous address ranges in memory space

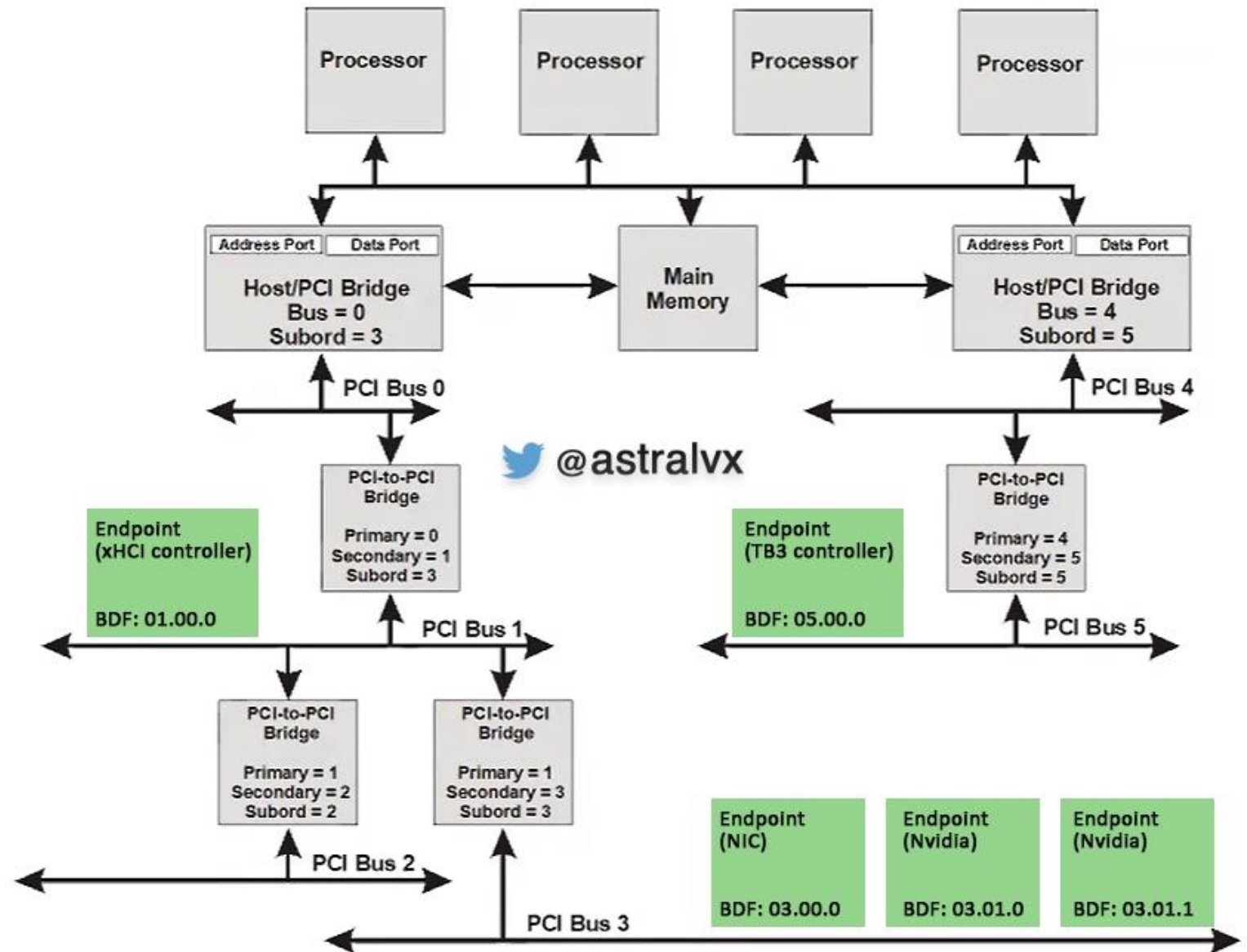


- PCI bridges form a hierarchy
- Also:
 - PCI-ISA
 - PCI-USB
 - PCI-SATA
 - Etc.

Devices

- **Host Bridge:** Connects CPU („host“) to PCI bus
=> It controls the **address mapping**
- **PCI-PCI Bridge:** Connects one PCI Bus with another one
- **BFD (Bus-Device-Function):** Unique identifier for a device on a PCI bus
- **Example:** BDF 00:1f.0
 - **Bus:** 00, **Device:** 1f, **Function:** 0

- **Primary:** The bus the system starts with
- **Secondary:** New bus created
- **Subord:** Highest numbered bus in secondary



Devices

- **Note:** The memory mapped regions from PCI for devies are for the **control registers** (registers != buffer/descriptor rings)
- **Why a tree structure?** In PCI, we have **PCI buses** which are broadcast
=> This yields issues such as **bandwith limitation, electrical limtations** (signal degredation and reflactions of noise): use multiple buses, connected via PCI-PCI bridge

Devices

PCI devices are self-describing

- Each device has a configuration header
 - Accessed through parent bridge, initially
- Some of the fields:

Bits	Description
16	Manufacturer ID (identifies Intel, 3Com, NVidia, etc.)
16	Model ID (specific to manufacturer)
24	Class code (what kind of device is this?)
8	Version identifier

- Plus:
 - Allocated/required address ranges (BAR values)
 - Interrupts
 - Electrical information
 - Etc.

SPCA in a nutshell

SPCA in a nutshell

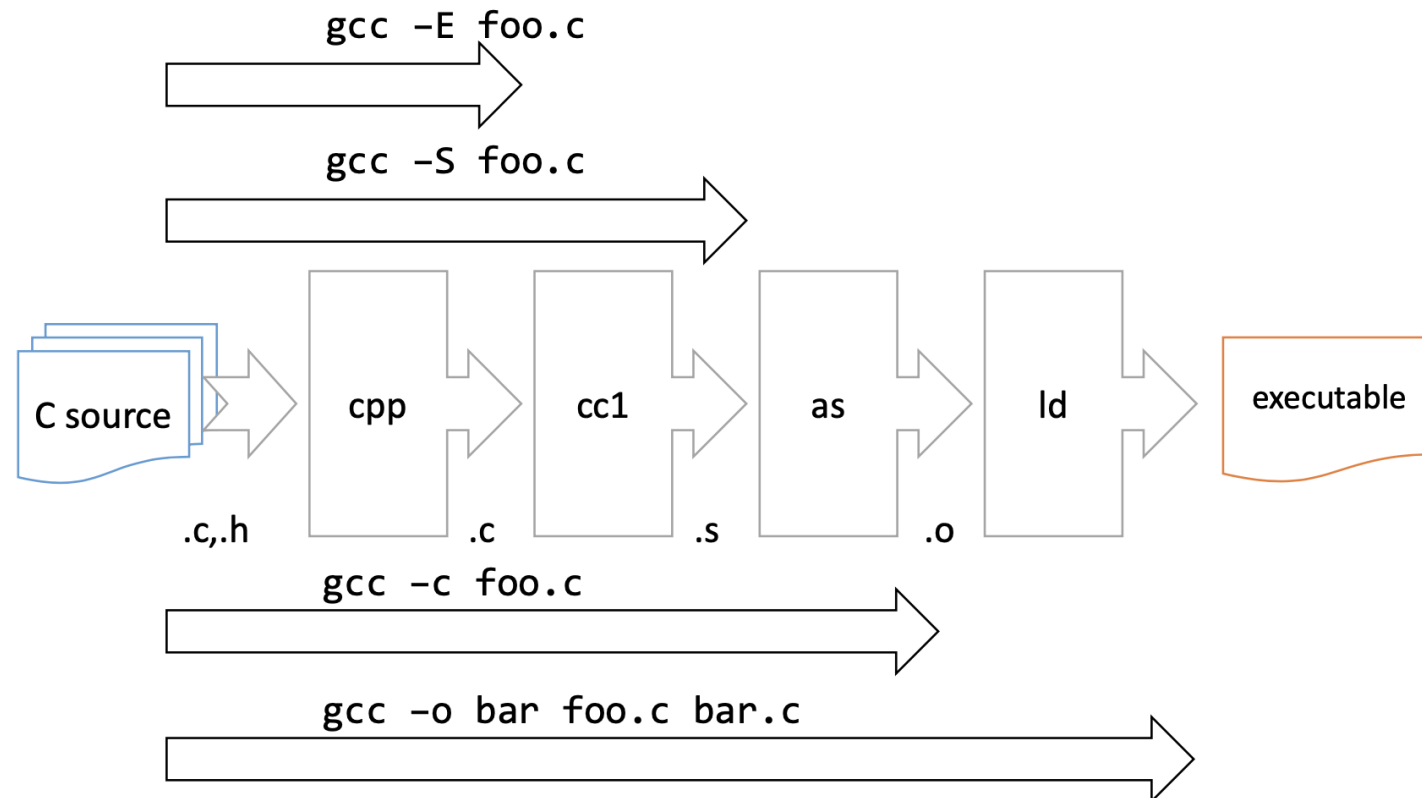
- **1. Programming Language C: Ch. 1-7**
- **2. Assembly x86-64, Compiling, Linking and Loading: Ch. 8-15**
- **3. Computer Arch, Exceptions, Virtual Memory, Devices: Ch. 16-21**

SPCA in a nutshell

- **1. Programming Language C:** Ch. 1-7
- **2. Assembly x86-64, Compiling, Linking and Loading:** Ch. 8-15
- **3. Computer Arch, Exceptions, Virtual Memory, Devices:** Ch. 16-21

SPCA in a nutshell

GNU gcc Toolchain



SPCA in a nutshell

Control flow statements
(like Java or C# or C++)

```
if (Boolean expression) Statement_when_true  
    else Statement_when_false
```

```
switch (Integer expression) {  
    case Constant_1: Statement; break;  
    case Constant_2: Statement; break;  
    ...  
    case Constant_n: Statement; break;  
    default: Statement; break;  
}
```

```
return (Expression)
```

Systems Programming 2024 Ch. 2: Introduction to C

Control flow statements
(like Java or C# or C++)

```
for (Initial; Test; Increment) Statement
```

```
while (Boolean expression) Statement
```

```
do Statement while (Boolean expression)
```

SPCA in a nutshell

Encoding integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;  
short int y = -15213;
```

Sign
Bit

- A C short is 2 bytes long:

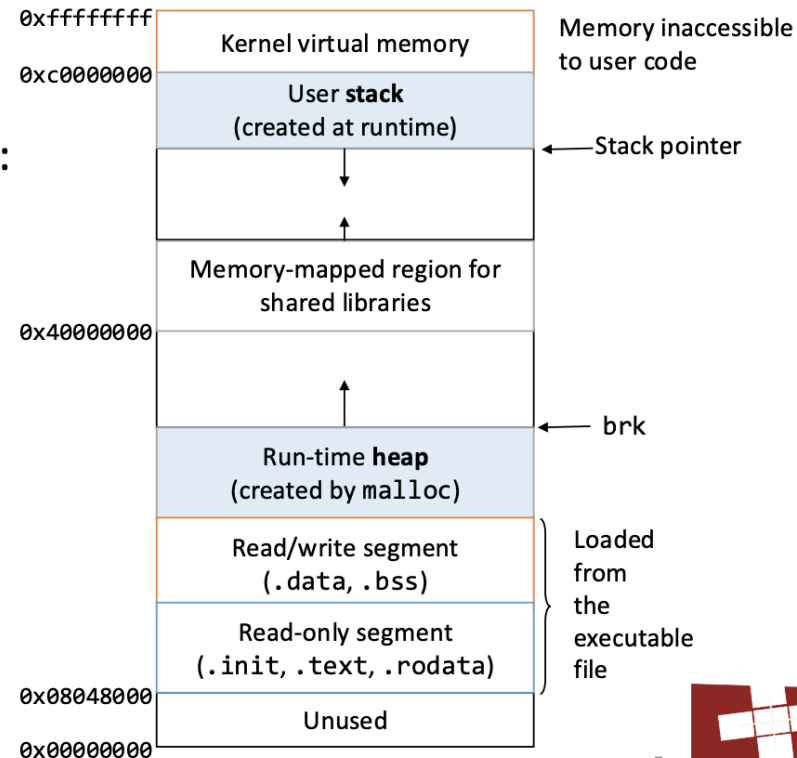
	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign bit
 - For 2's complement, most significant bit = 1 indicates negative

SPCA in a nutshell

Loading

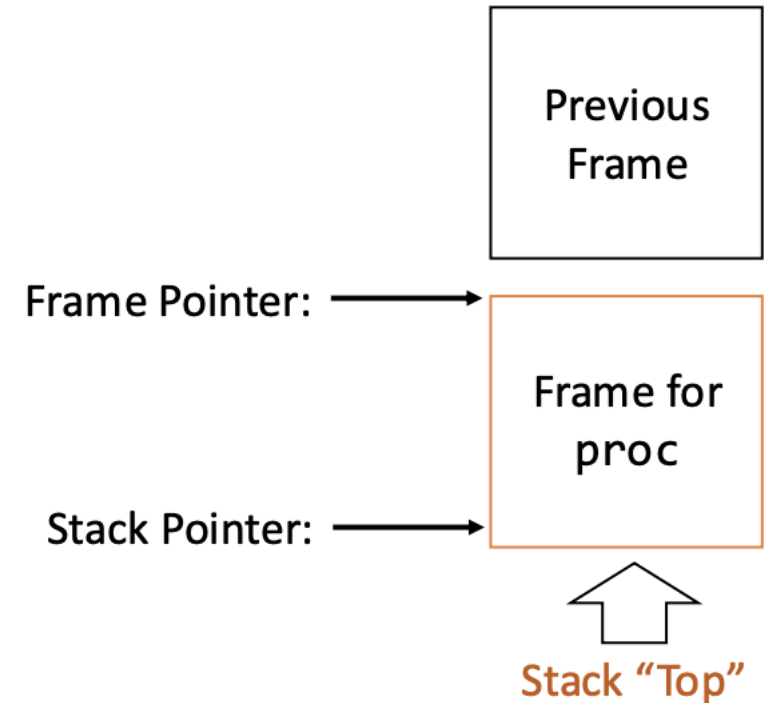
- When the OS loads a program, it:
 - creates an **address space**
 - inspects the **executable file** to see what's in it
 - (lazily) copies **regions** of the file into the right place in the address space
 - does any final **linking, relocation**, or other needed preparation



5



ire



SPCA in a nutshell

Dynamic memory allocation

- Memory allocator?
 - VM h/w and kernel allocate pages
 - Application objects typically small
 - Allocator manages objects with blocks
- Allocation
 - A memory allocator does out memory in blocks
 - “block”: contiguous range of bytes
 - of any size, in this context

External fragmentation

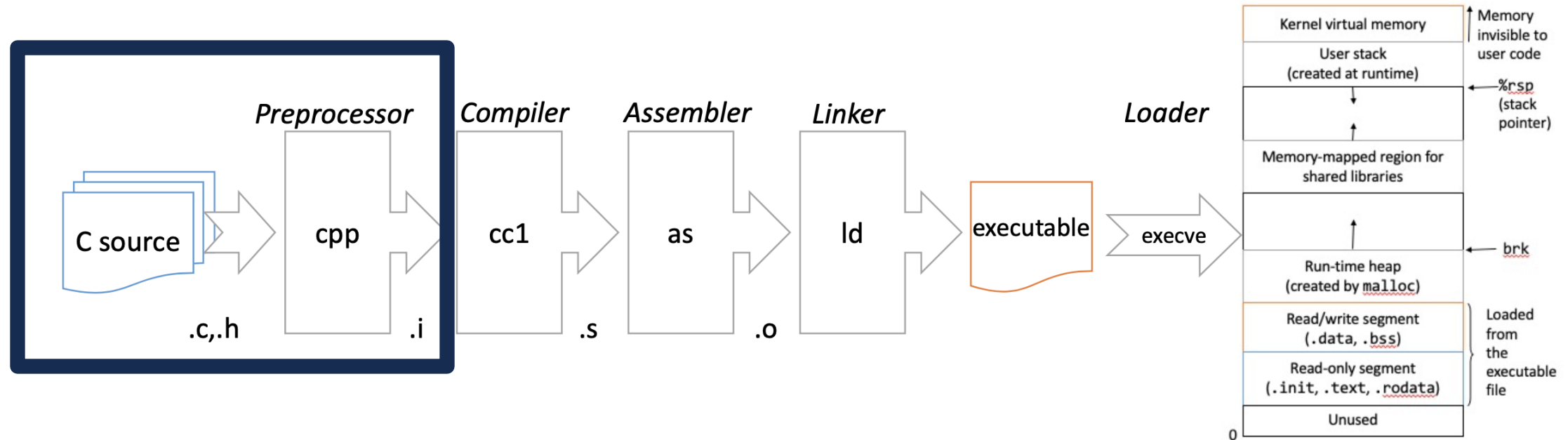
- Occurs when there is enough aggregate heap memory, but no single free block is large enough



- Depends on the pattern of future requests
 - Thus, difficult to measure

SPCA in a nutshell

Recall: how C code runs as a process on CPU



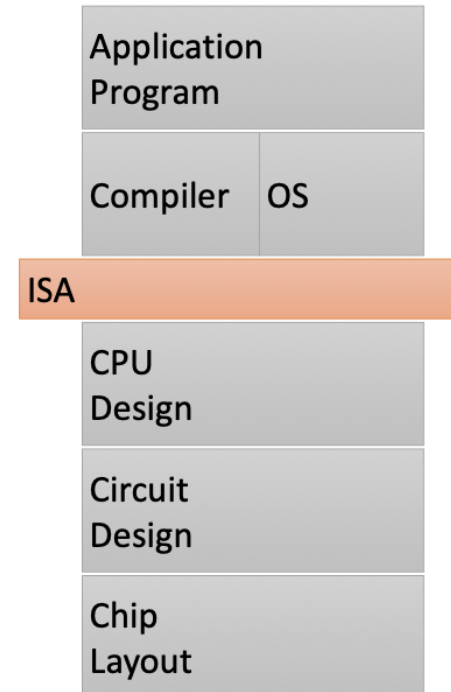
SPCA in a nutshell

- **1. Programming Language C: Ch. 1-7**
- **2. Assembly x86-64, Compiling, Linking and Loading: Ch. 8-15**
- **3. Computer Arch, Exceptions, Virtual Memory, Devices: Ch. 16-21**

SPCA in a nutshell

Instruction Set Architecture

- **Assembly Language View**
 - Processor state
 - Registers, memory, ...
 - Instructions
 - addl, movq, leal, ...
 - 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



SPCA in a nutshell

Moving data

`movx Source, Dest:`

- **Operand Types**

- **Immediate:** Constant integer data
 - Example: `$0x400`, `$-533`
 - Like C constant, but prefixed with ``$'`
 - Encoded with 1, 2, 4, 8 bytes
- **Register:** One of 16 integer registers
 - Example: `%eax`, `%r14d`
 - Note some (e.g. `%rsp`, `%rbp`) reserved for special use
 - Others have special uses for particular instructions
- **Memory:** 1,2,4, or 8 consecutive bytes of memory at address given by register
 - Simplest example: `(%rax)`
 - Various other “address modes”

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

SPCA in a nutshell

Source code:

Compiling into assembly

C code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Obtain with command

```
gcc -O0 -S code.c
```

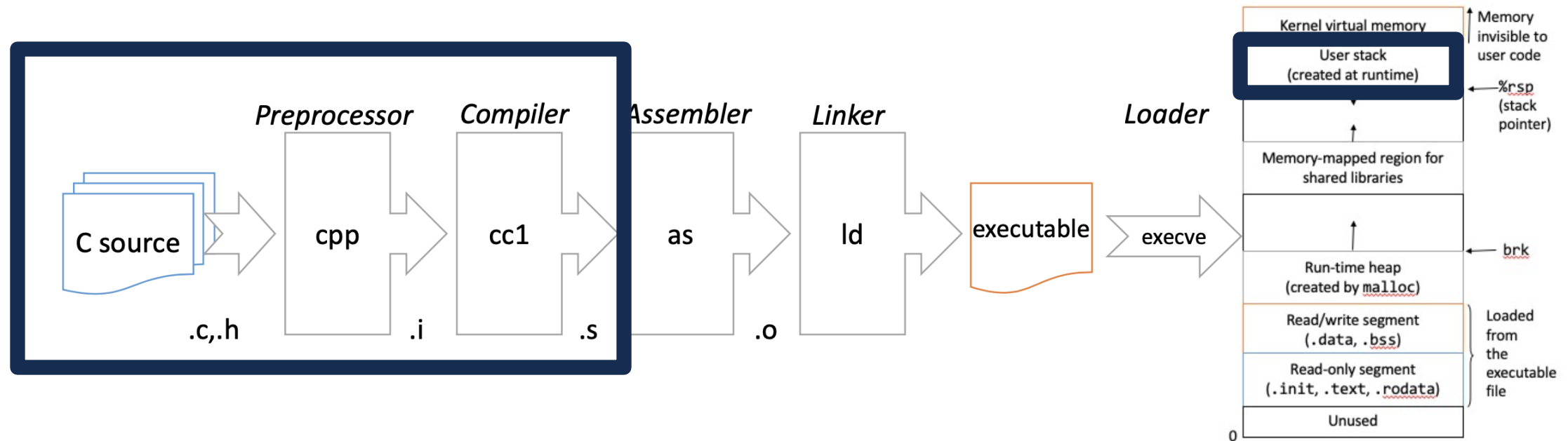
Produces file code.s

Generated x86 assembly

```
sum:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %edx, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
```

SPCA in a nutshell

Recall: how C code runs as a process on CPU



SPCA in a nutshell

Object code

- **Assembler**
 - Translates .s into .o
 - Binary encoding of each instruction
 - Nearly-complete image of executable code
 - Missing linkages between code in different files
- **Linker**
 - Resolves references between files
 - Combines with static run-time libraries
 - E.g., code for malloc, printf
 - Some libraries are dynamically linked
 - Linking occurs when program begins execution

Code for sum.o

```
<sum>:
 0: f3 0f 1e fa
 4: 55
 5: 48 89 e5
 8: 89 7d ec
 b: 89 75 e8
 e: 8b 55 ec
11: 8b 45 e8
14: 01 d0
16: 89 45 fc
19: 8b 45 fc
1c: 5d
1d: c3
```

- Total of 26 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

SPCA in a nutshell

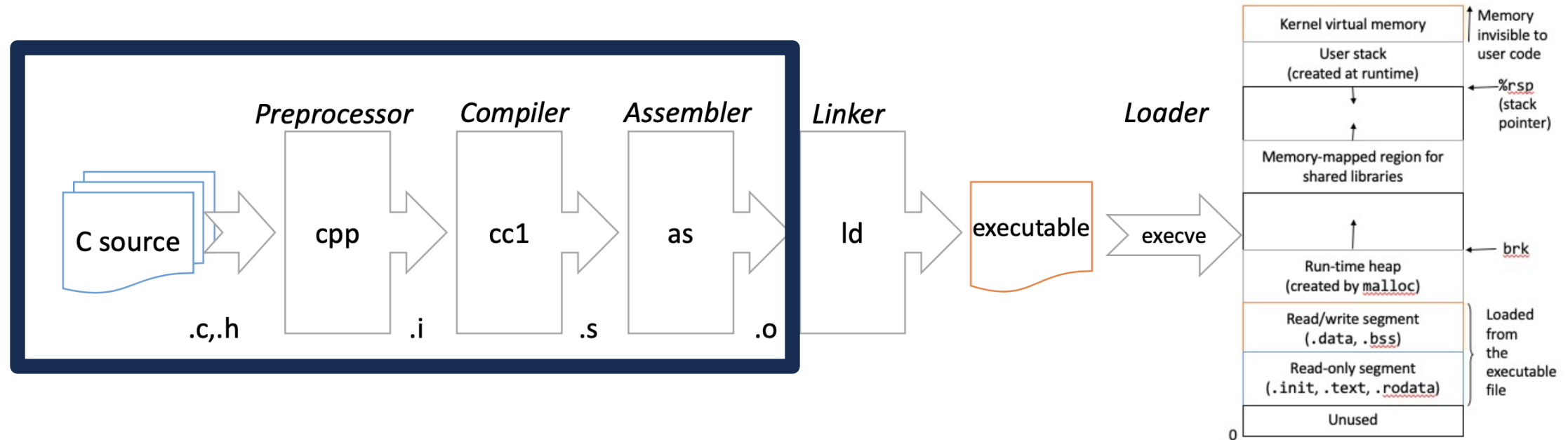
Alternate disassembly

Within gdb debugger:

```
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x0000000000000000 <+0>:      endbr64
   0x0000000000000004 <+4>:      push    %rbp
   0x0000000000000005 <+5>:      mov     %rsp,%rbp
   0x0000000000000008 <+8>:      mov     %edi,-0x14(%rbp)
   0x000000000000000b <+11>:     mov     %esi,-0x18(%rbp)
   0x000000000000000e <+14>:     mov     -0x14(%rbp),%edx
   0x0000000000000011 <+17>:     mov     -0x18(%rbp),%eax
   0x0000000000000014 <+20>:     add     %edx,%eax
   0x0000000000000016 <+22>:     mov     %eax,-0x4(%rbp)
   0x0000000000000019 <+25>:     mov     -0x4(%rbp),%eax
   0x000000000000001c <+28>:     pop     %rbp
   0x000000000000001d <+29>:     retq
End of assembler dump.
```


SPCA in a nutshell

Recall: how C code runs as a process on CPU



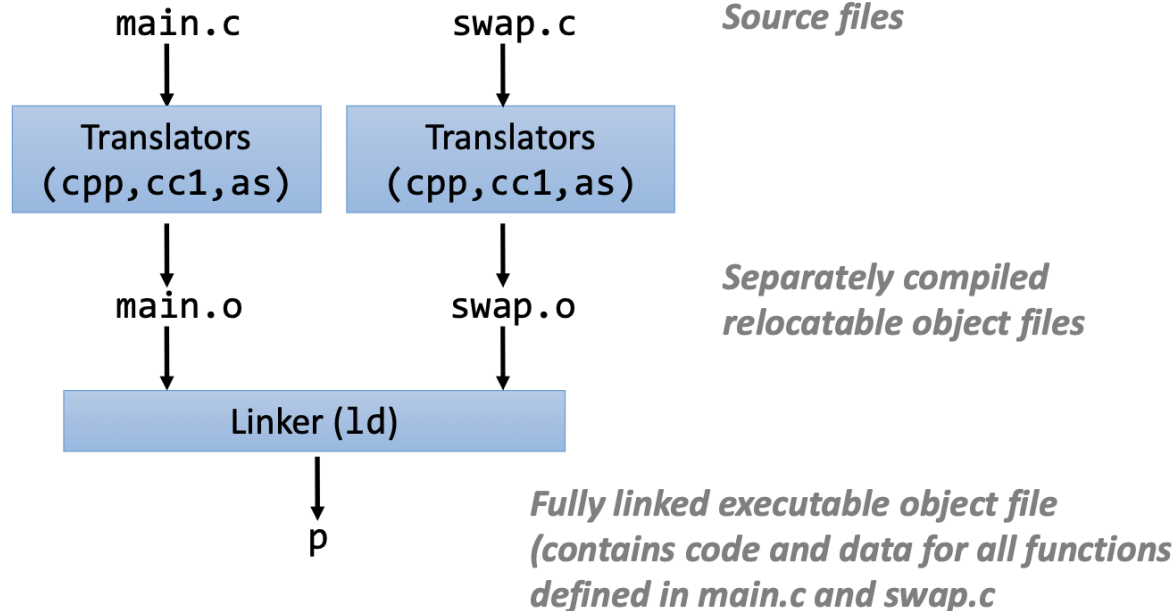
SPCA in a nutshell

Static linking

- Programs are translated and linked using a *compiler driver*:

```
unix> gcc -O2 -g -o p main.c swap.c
```

```
unix> ./p
```



Systems Programming 2024 Ch. 12: Linking

Disassembly of section .data:

```
0000000000000000 <bufp0>:
  0:  00 00 00 00 00 00 00 00
                        0: R_X86_64_64  buf
```

Disassembly of section .bss:

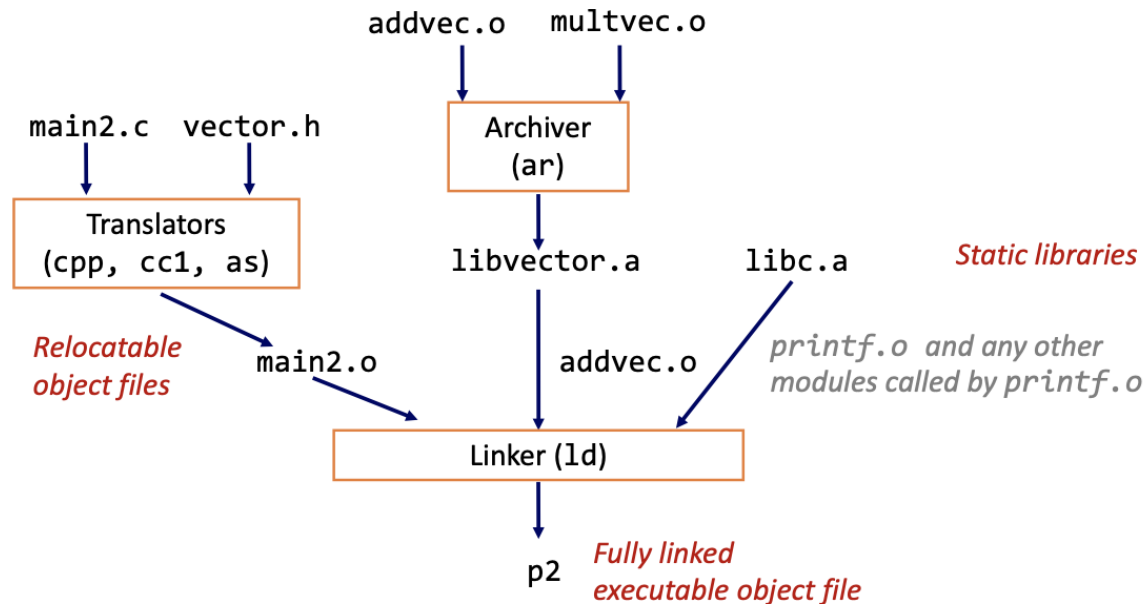
```
0000000000000000 <bufp1>:
  0:  00 00 00 00 00 00 00 00
```

```
push    %rbp
mov     %rsp,%rbp
movq    $0x60103c,0x200b3c(%rip) # 601050 <bufp0>

mov     0x200b25(%rip),%rax      # 601040 <bufp0>
mov     (%rax),%eax
mov     %eax,-0x4(%rbp)
mov     0x200b19(%rip),%rax      # 601040 <bufp0>
mov     0x200b22(%rip),%rdx      # 601050 <bufp0>
mov     (%rdx),%edx
mov     %edx,(%rax)
mov     0x200b17(%rip),%rax      # 601050 <bufp0>
mov     -0x4(%rbp),%edx
mov     %edx,(%rax)
```

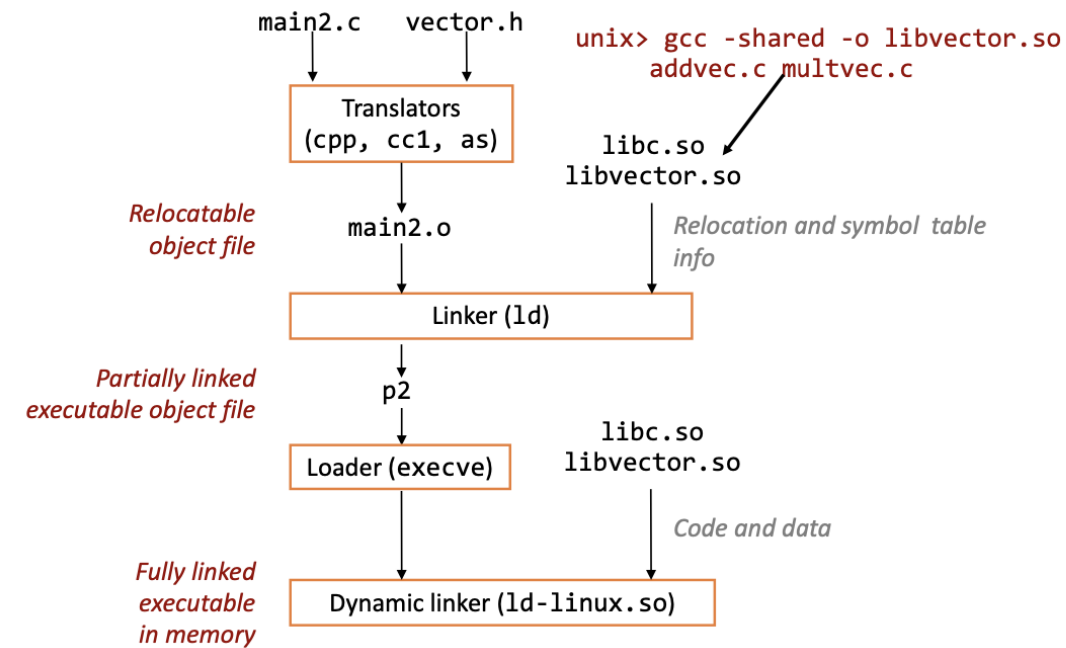
SPCA in a nutshell

Linking with static libraries



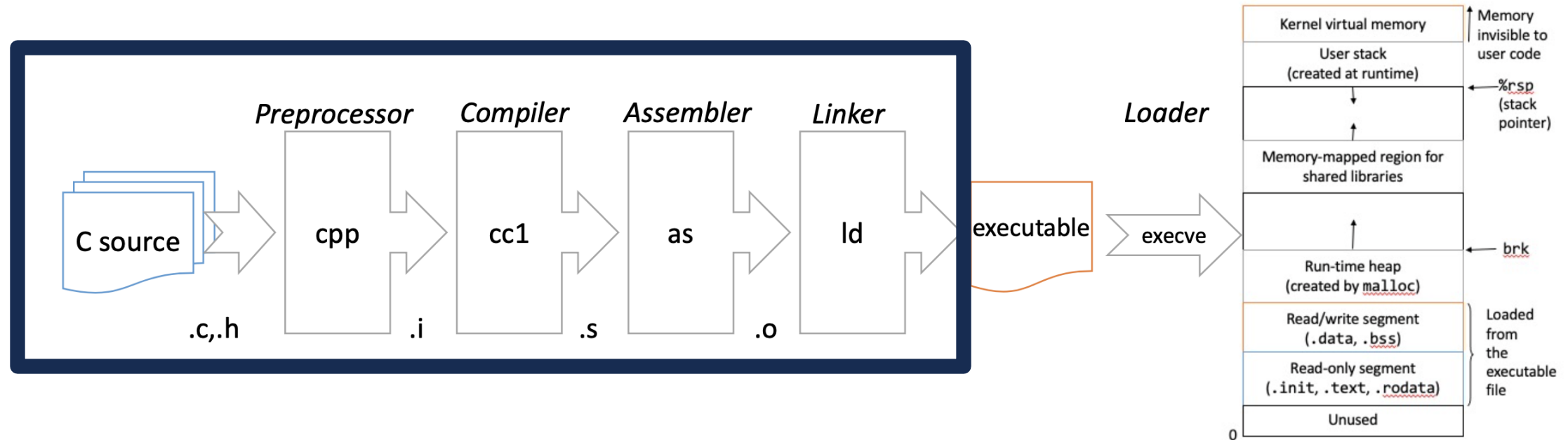
Systems Programming 2024 Ch. 12: Linking

Dynamic linking at load-time



SPCA in a nutshell

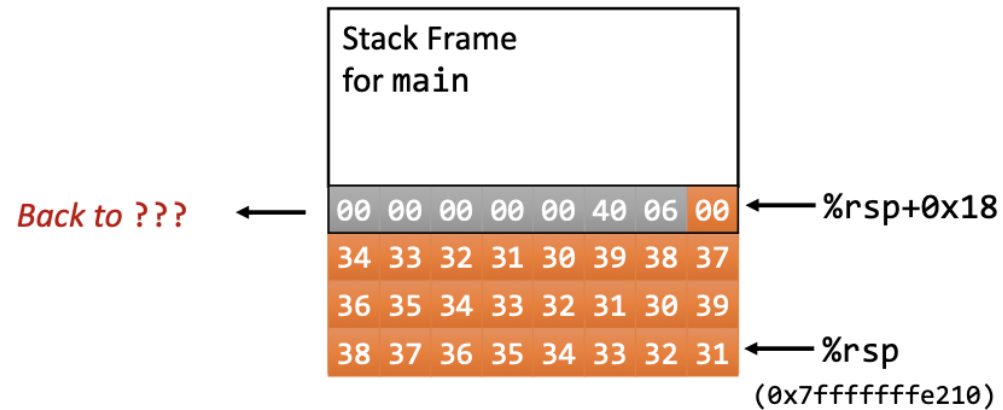
Recall: how C code runs as a process on CPU



SPCA in a nutshell

Buffer overflow stack

Input: 1234567901234567901234



What about something more interesting than crashing?

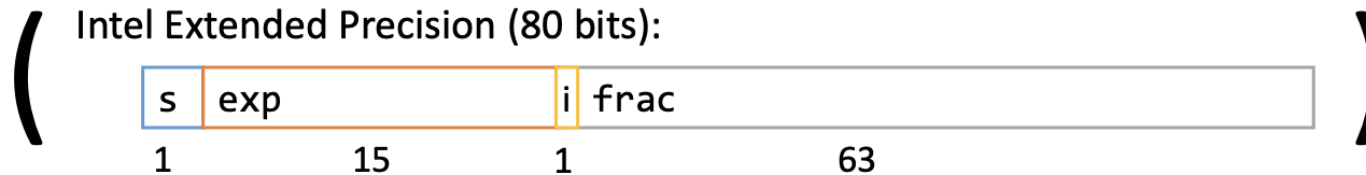
SPCA in a nutshell

Original precisions

IEEE 754 Single Precision (32 bits):



IEEE 754 Double Precision (64 bits):



SPCA in a nutshell

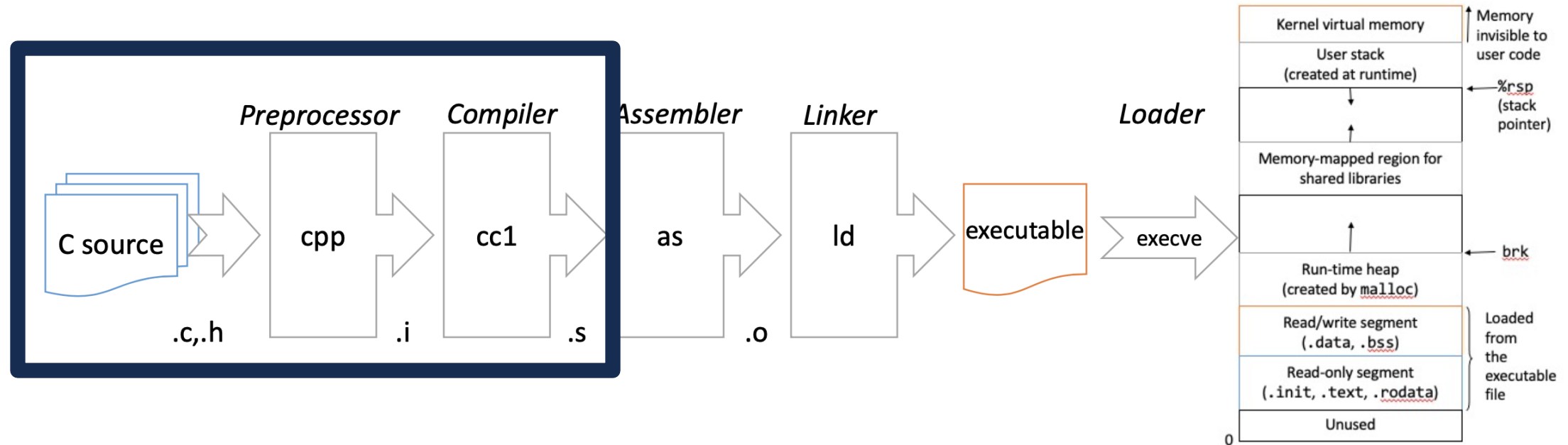
Optimizing compilers

- Use optimization flags, default can be no optimization (-O0)!
- Good choices for gcc:
-O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
 - icc is often faster than gcc



SPCA in a nutshell

Recall: how C code runs as a process on CPU



SPCA in a nutshell

- C Program: from source code to final executable

```
#include <stdio.h>
#include "functions.h"

int main(int argc, char** argv){
    printf("hello, world\n");
    printf("square of 3: %d\n", square(3));
    return 0;
}
```

```
.section      __TEXT,__text,regular,pure_instructions
.build_version macos, 15, 0      sdk_version 15, 0
.globl _main
.p2align     4, 0x90
_main:
    .cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq     $16, %rsp
    movl     $0, -4(%rbp)
    movl     %edi, -8(%rbp)
    movq     %rsi, -16(%rbp)
    leaq     L_.str(%rip), %rdi
    movb     $0, %al
    callq    _printf
    movl     $3, %edi
    callq    _square
    movl     %eax, %esi
    leaq     L_.str.1(%rip), %rdi
    movb     $0, %al
    callq    _printf
    xorl     %eax, %eax
    addq     $16, %rsp
    popq     %rbp
    retq
    .cfi_endproc

                                ## -- End function
L_.str:
    .section      __TEXT,__cstring,cstring_literals
                                ## @.str
    .asciz    "hello, world\n"

L_.str.1:
                                ## @.str.1
    .asciz    "square of 3: %d\n"

.subsections_via_symbols
```

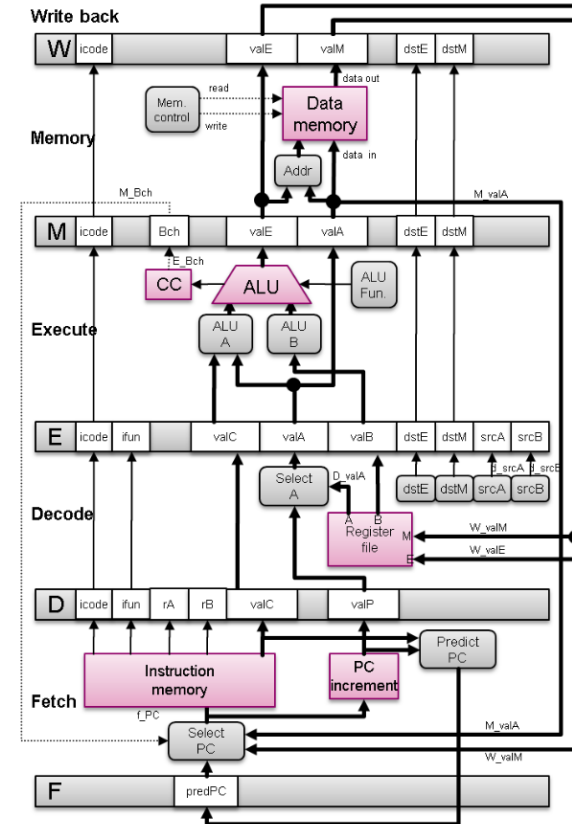
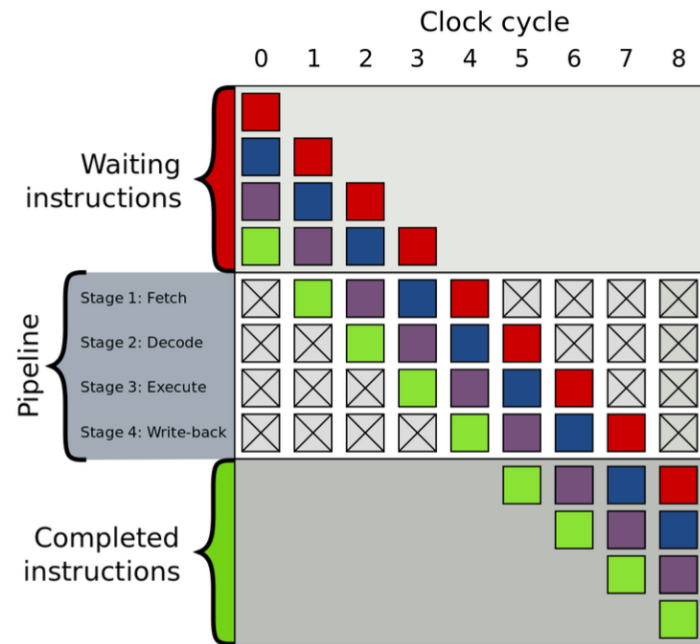
```
student-net-hg-1584:make-demo bened
00000000: 11001111 11111010 11101110
00000006: 00000000 00000001 00000001
0000000c: 00000001 00000000 00000000
00000012: 00000000 00000000 00001000
00000018: 00000000 00100000 00000000
0000001e: 00000000 00000000 00011000
00000024: 10001000 00000001 00000000
0000002a: 00000000 00000000 00000000
00000030: 00000000 00000000 00000000
00000036: 00000000 00000000 00000000
0000003c: 00000000 00000000 00000000
00000042: 00000000 00000000 00000000
00000048: 00101000 00000010 00000000
0000004e: 00000000 00000000 11001000
00000054: 00000000 00000000 00000000
0000005a: 00000000 00000000 00000110
00000060: 00000100 00000000 00000000
00000066: 00000000 00000000 01011111
0000006c: 01111000 01110100 00000000
00000072: 00000000 00000000 00000000
00000078: 01011111 01011111 01010100
0000007e: 00000000 00000000 00000000
00000084: 00000000 00000000 00000000
0000008a: 00000000 00000000 00000000
00000090: 01000110 00000000 00000000
00000096: 00000000 00000000 00101000
0000009c: 00000100 00000000 00000000
000000a2: 00000000 00000000 00000100
000000a8: 00000000 00000100 00000000
000000ae: 00000000 00000000 00000000
000000b4: 00000000 00000000 00000000
000000ba: 01100011 01110011 01110100
```

SPCA in a nutshell

- **1. Programming Language C: Ch. 1-7**
- **2. Assembly x86-64, Compiling, Linking and Loading: Ch. 8-15**
- **3. Computer Arch, Exceptions, Virtual Memory, Devices: Ch. 16-21**

SPCA in a nutshell

Pipelined hardware



SPCA in a nutshell

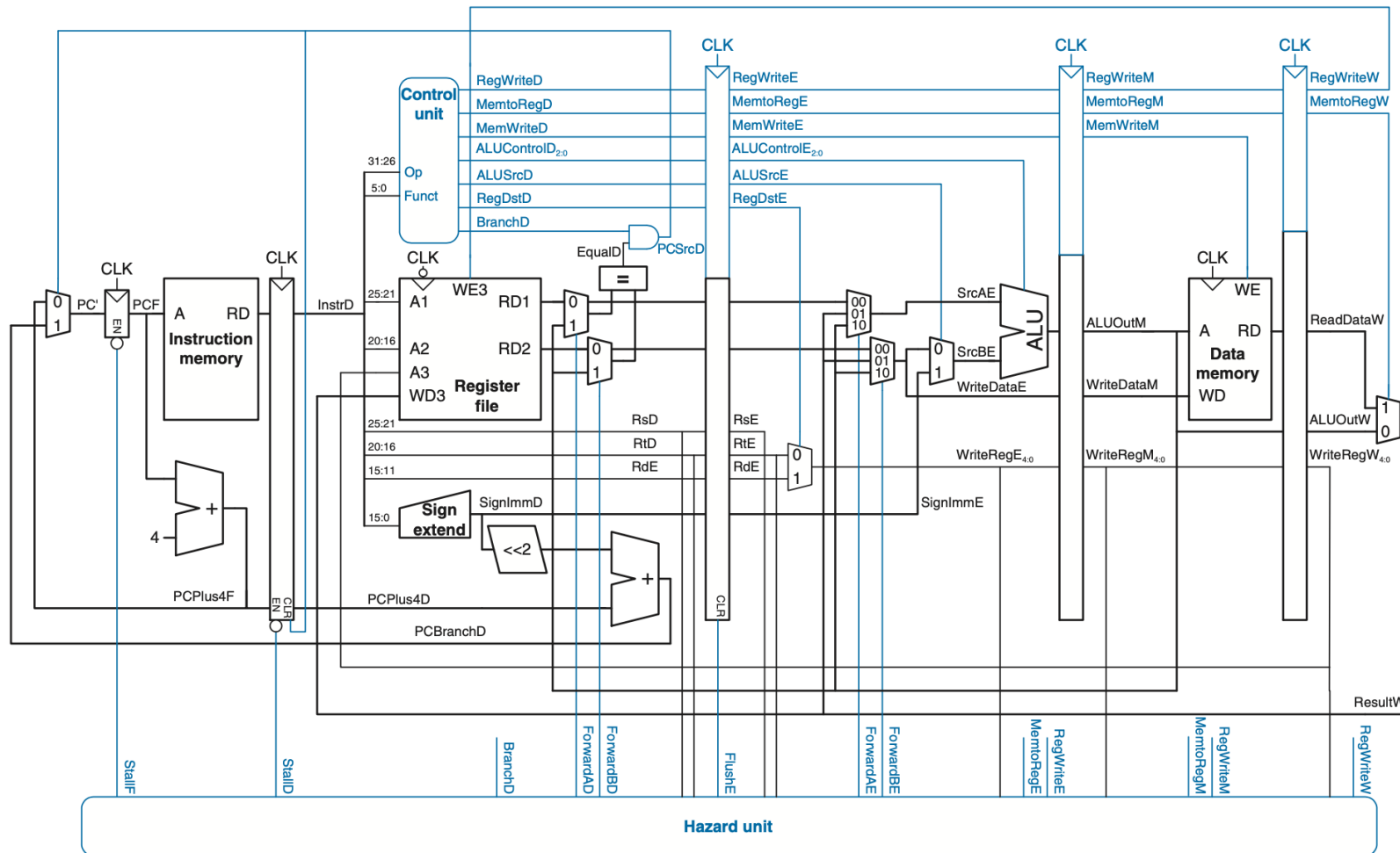
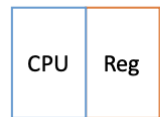


Figure 7.58 Pipelined processor with full hazard handling

SPCA in a nutshell

Historical Problem: Processor-memory bottleneck

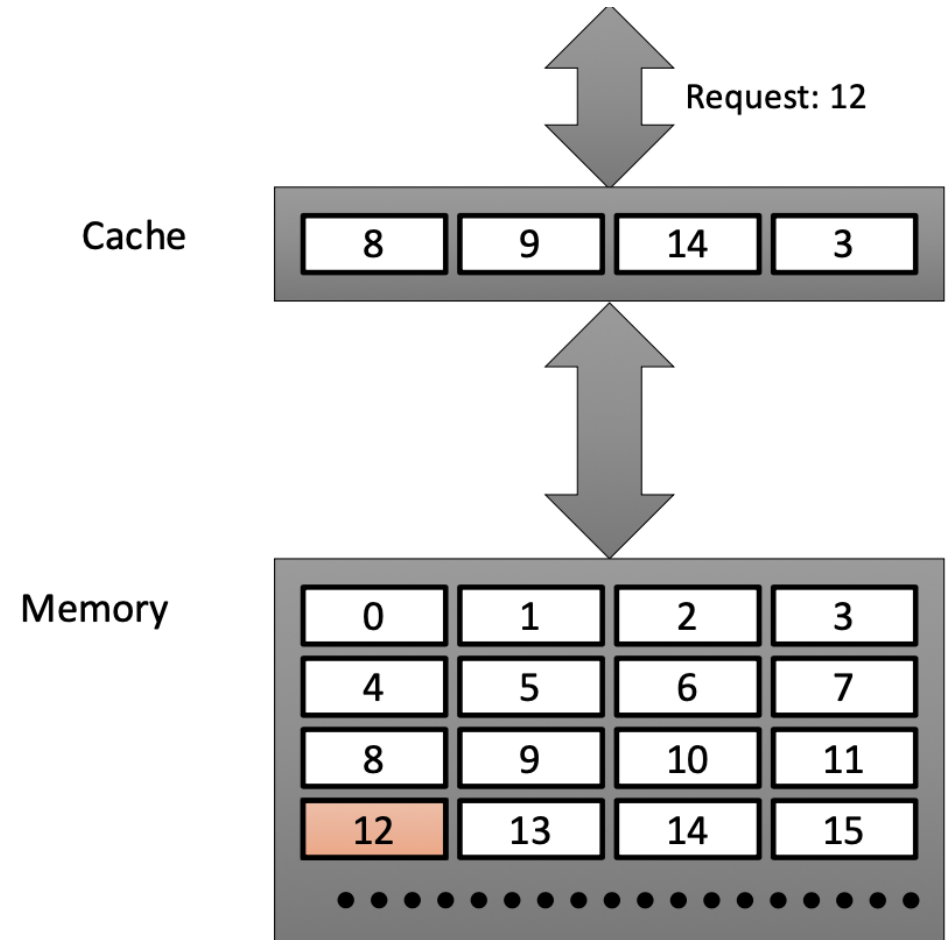
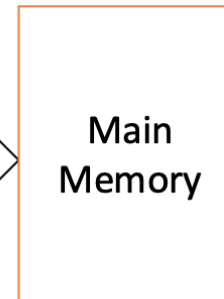
Processor performance
doubled about
every 18 months



Intel Haswell:
Can process at least
512 Bytes/cycle
(1 SSE two operand add and mult)

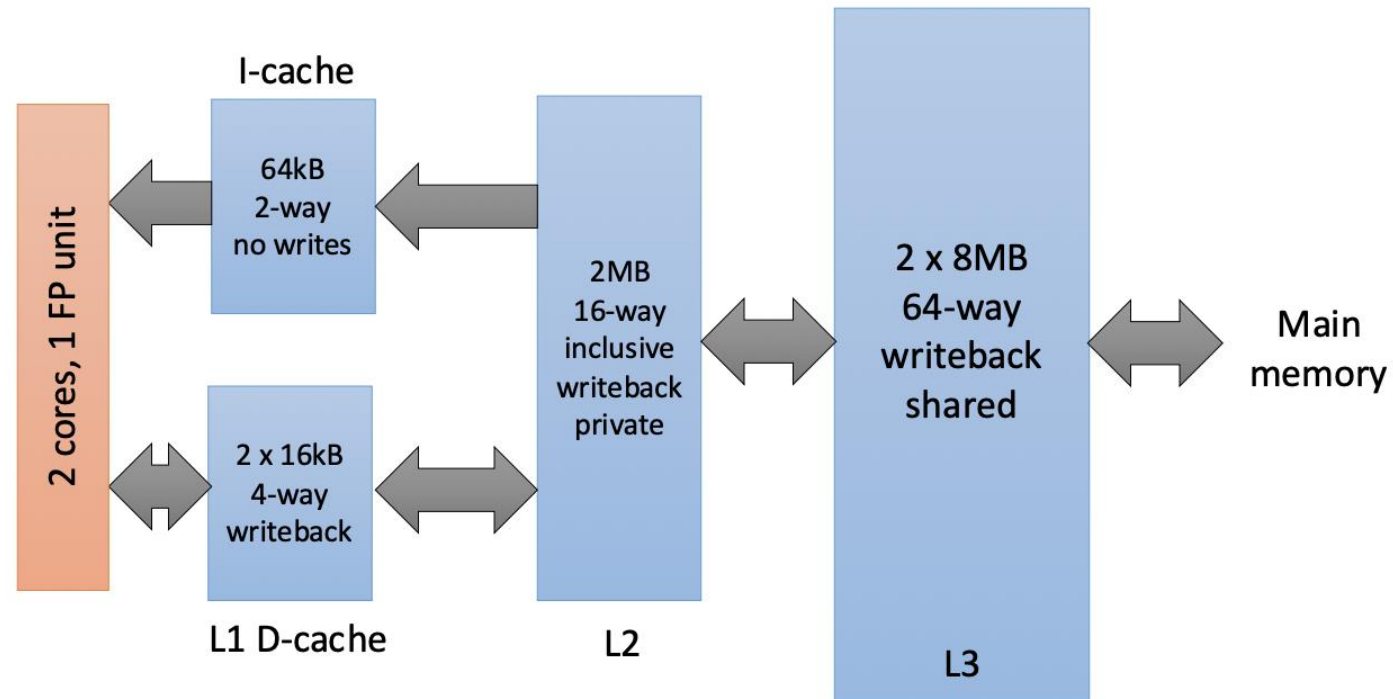
Bus bandwidth
evolved much slower

Intel Haswell:
Bandwidth: 10 Bytes/cycle
Latency: 100 cycles



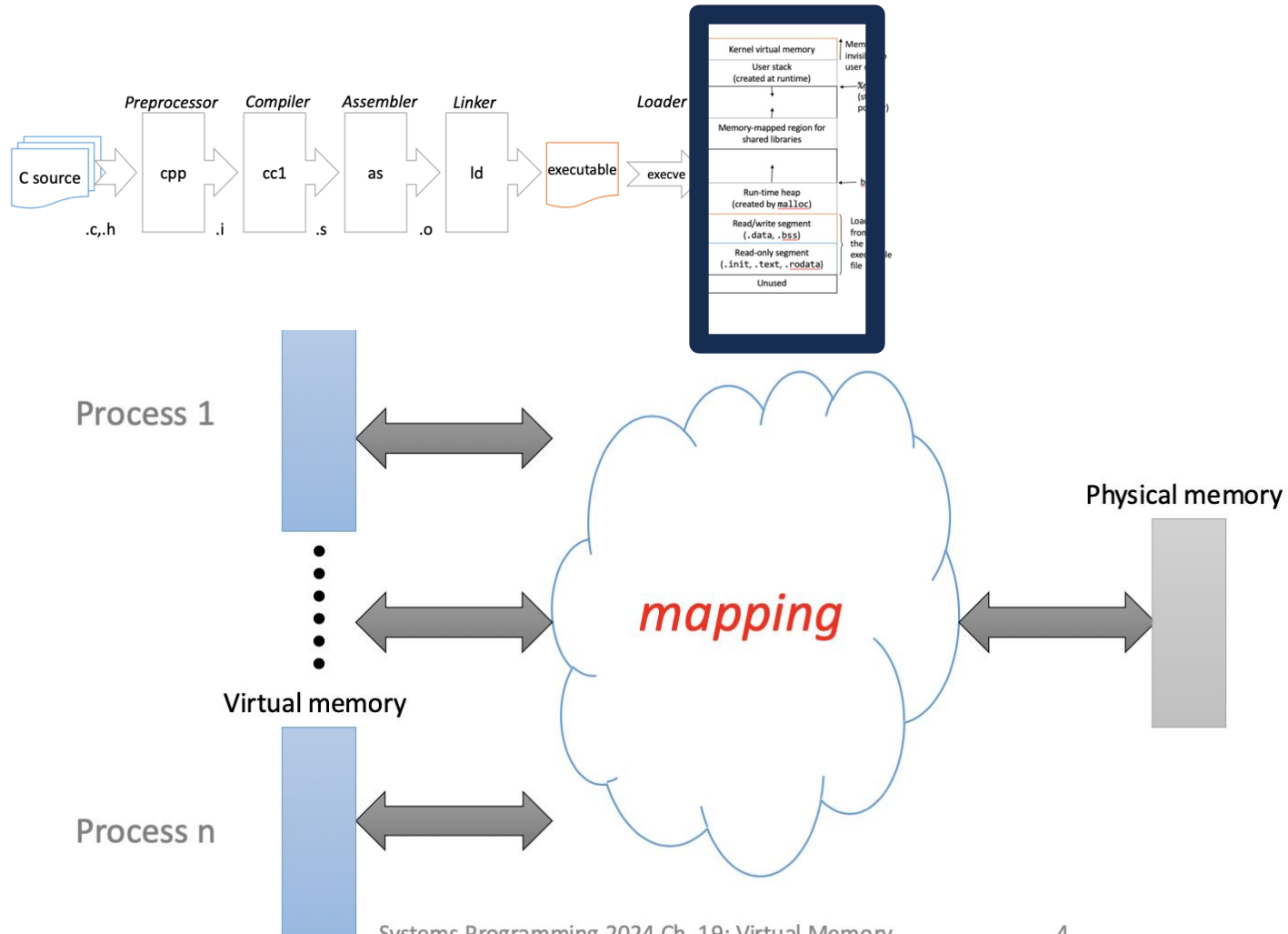
SPCA in a nutshell

Other caches: AMD Warsaw (Opteron 6380)



SPCA in a nutshell

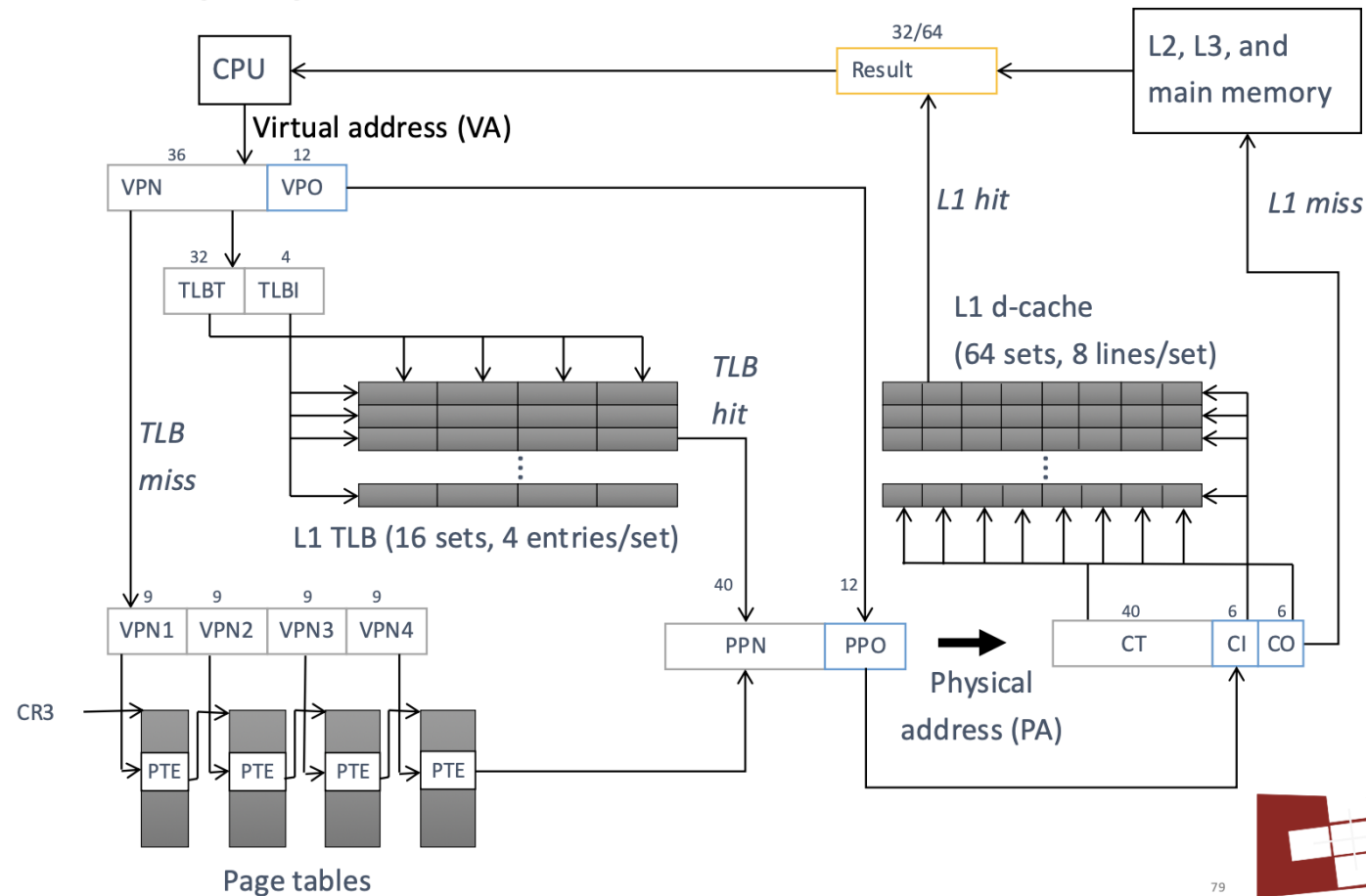
Recall: how C code runs as a process on CPU



CDCA in a nutshell

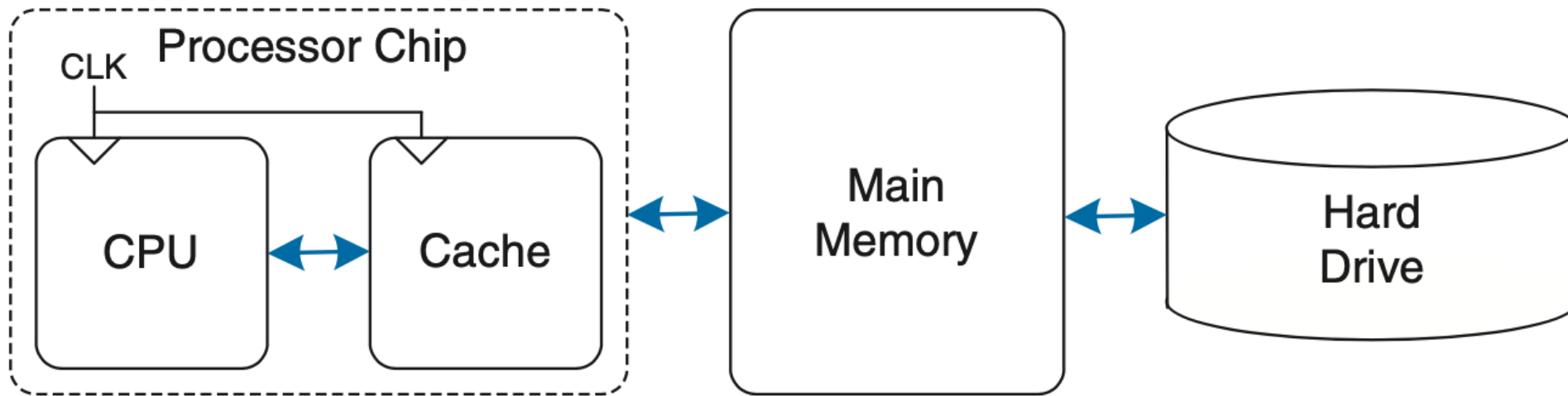
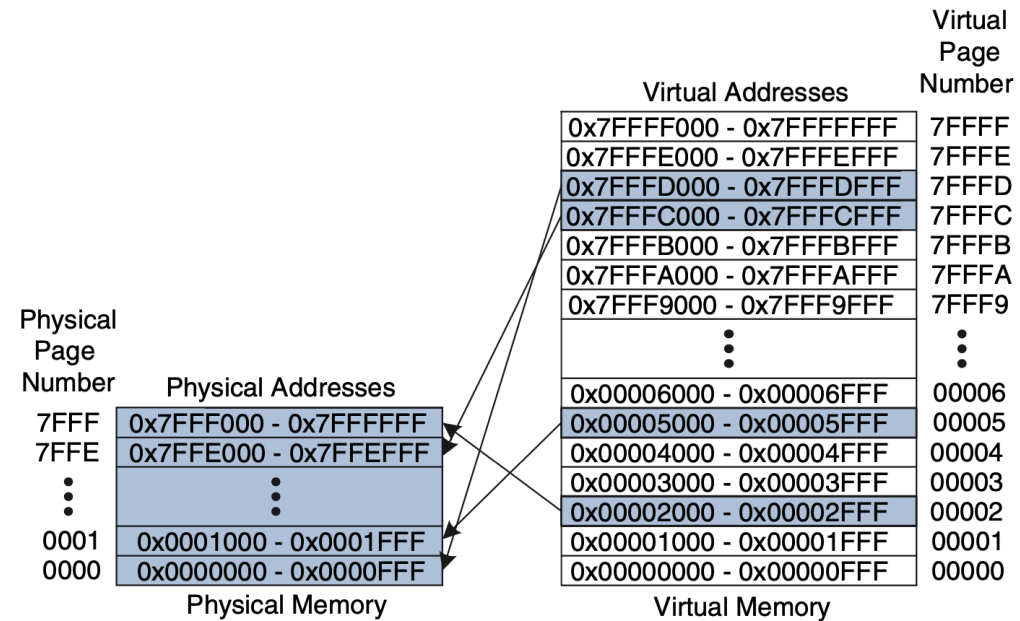
Core i7 memory system*

*A bit simplified



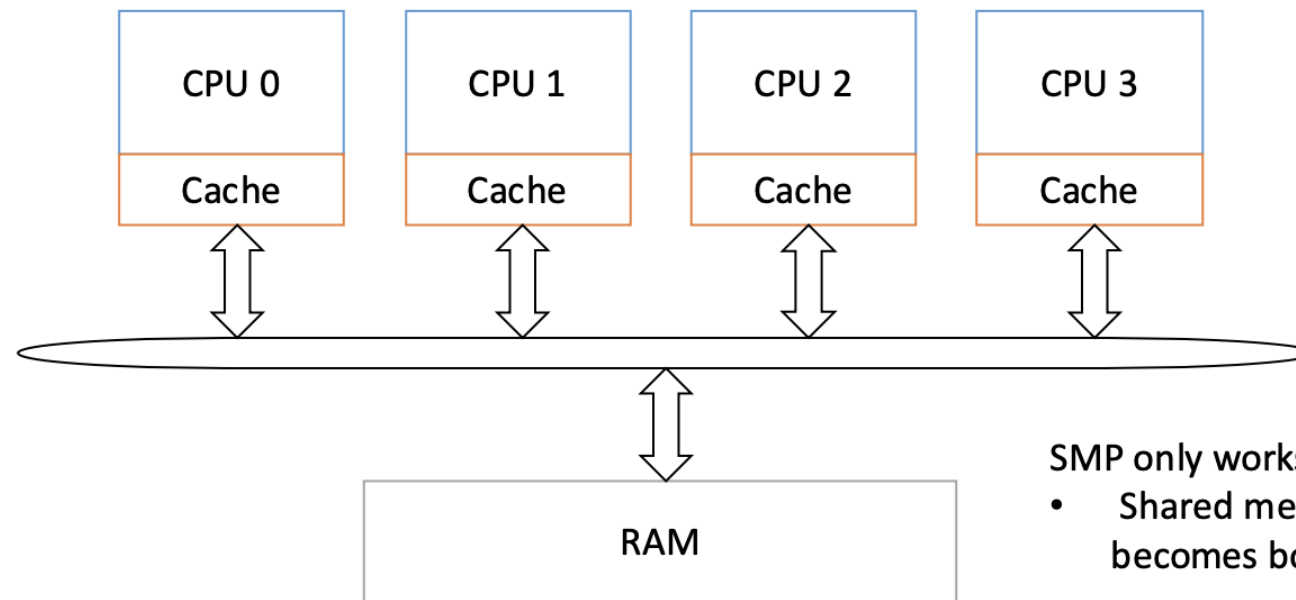
SPCA in a nutshell

- Caches and Virtual Memory



SPCA in a nutshell

Symmetric multiprocessing (SMP)

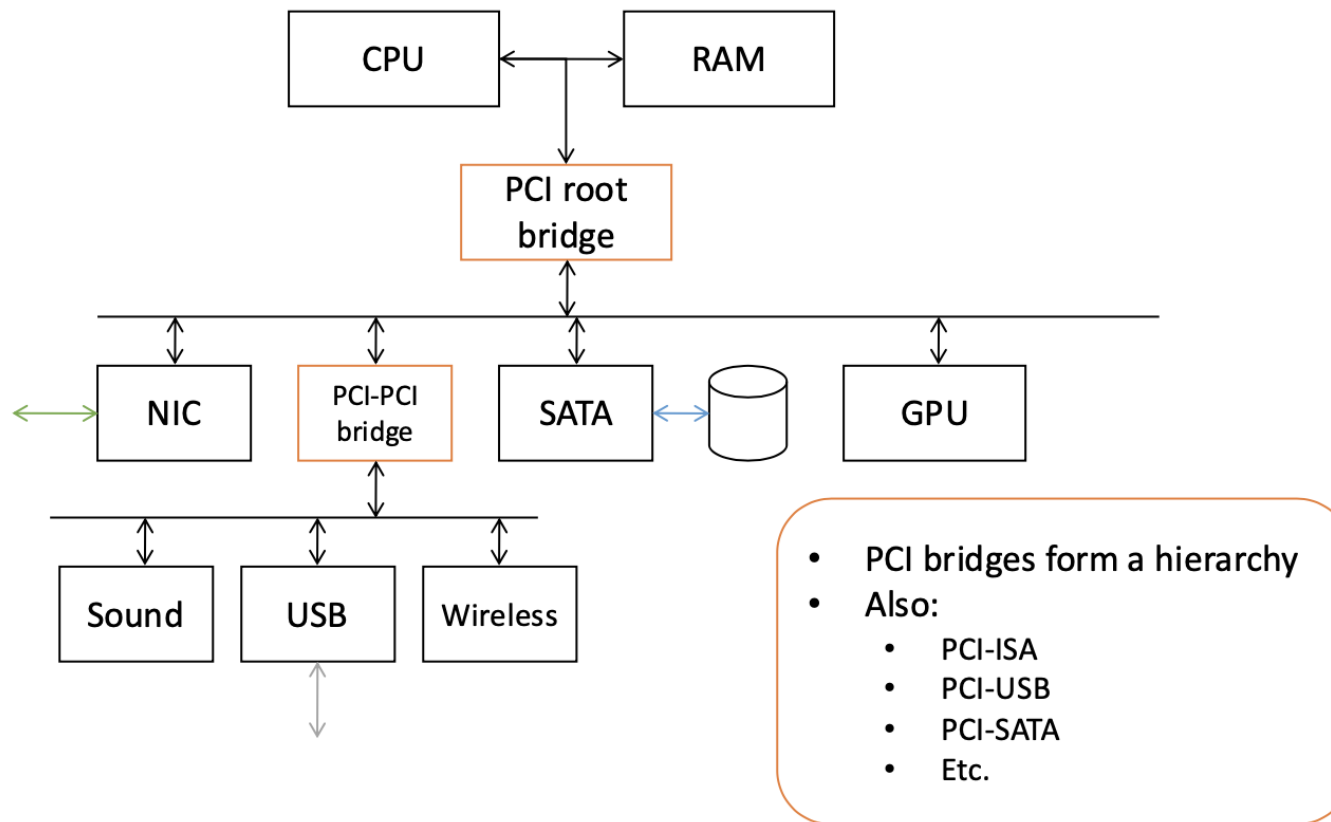


SMP only works because of caches!

- Shared memory rapidly becomes bottleneck

SPCA in a nutshell

Physical connections: PCI is a tree



SPCA in a nutshell

- Devices in combination with the processor

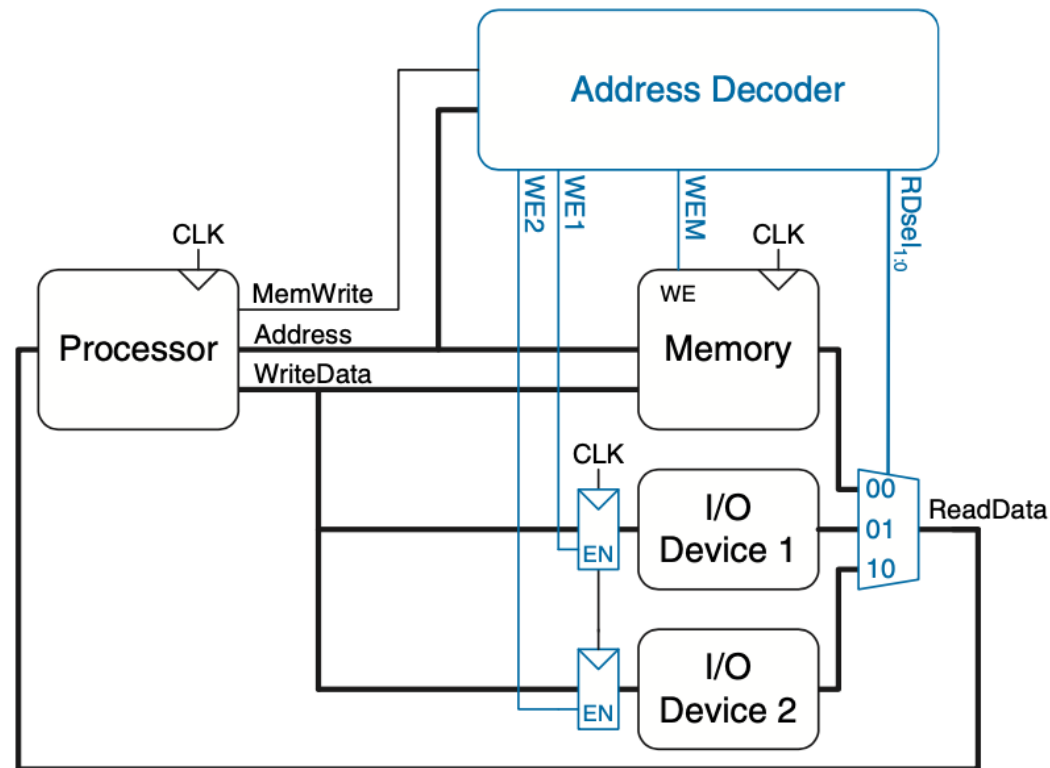


Figure 8.28 Support hardware for memory-mapped I/O



Questions?

SPCA Exam Remarks

SPCA Exam Remarks

- Know the theory well
 - Solve old pen&paper exams
 - Solve the exercise sheets again
- You will need to know some stuff by heart (How to calculate FP bias etc.)

SPCA Exam Remarks

- **C Programming:** most of the exam will be programming
- You will **not need** GDB: i.e. it will not be „bomb lab“ style
- **Training:** do all the C code experts
- **Some of the Labs**
 - 1. FP Lab (If issues with bits, then also: Bit lab)
 - 2. Paging Lab
 - 3. Cache, Attack and Bomb Lab: help with understanding but most likely not directly applicable

SPCA in perspective

SPCA in perspective

Grundlagenfächer

- **Computer Networks:** How does the internet work
- **Data Modelling and Data Bases:** How to store huge amount of data

Kernfächer

- **Computer Systems:** Operating- and Distributed Systems
- **Compiler Design:** How to build a compiler

SPCA in perspective

Grundlagenfächer

- **Computer Networks:** How does the internet work
- **Data Modelling and Data Bases:** How to store huge amount of data

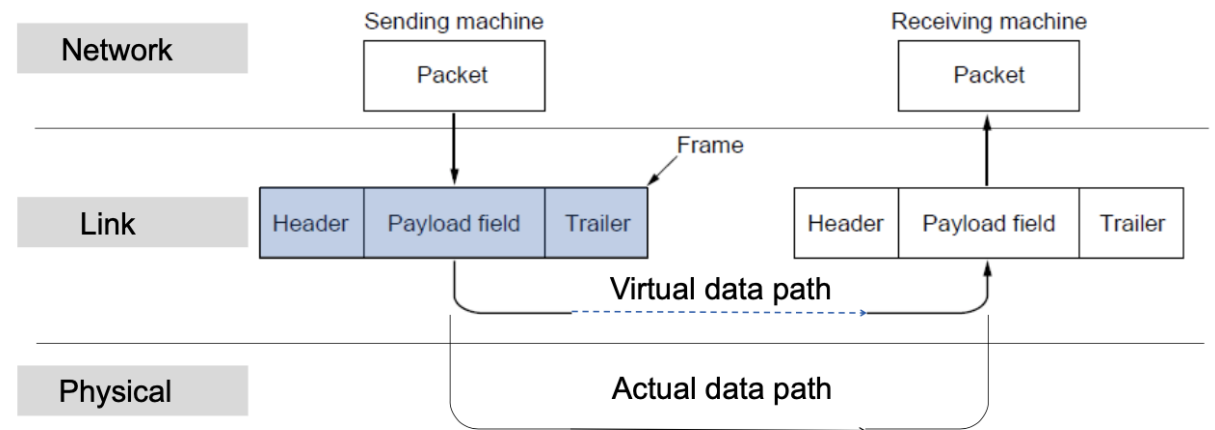
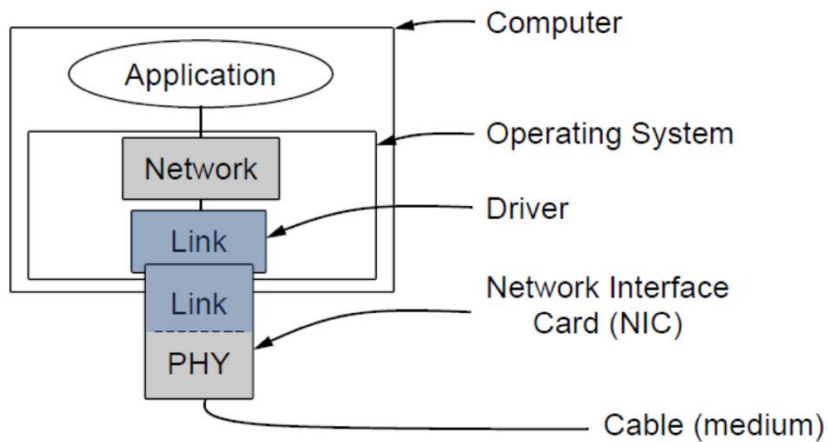
Kernfächer

- **Computer Systems:** Operating- and Distributed Systems
- **Compiler Design:** How to build a compiler

SPCA in perspective

Computer Networks: Network Stack

	layer	protocol
L5	Application	HTTP, SMTP, FTP, SIP, ...
L4	Transport	TCP, UDP, SCTP
L3	Network	IP
L2	Link	Ethernet, Wifi, (A/V)DSL, WiMAX, LTE, ...
L1	Physical	Twisted pair, fiber, coaxial cable, ...

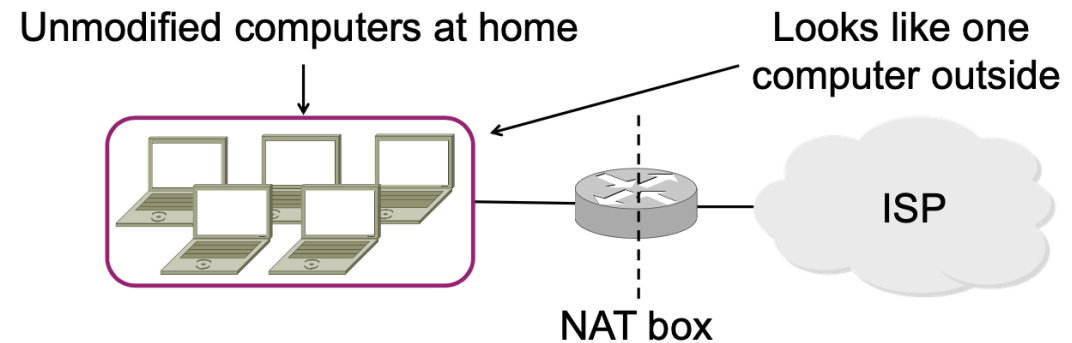
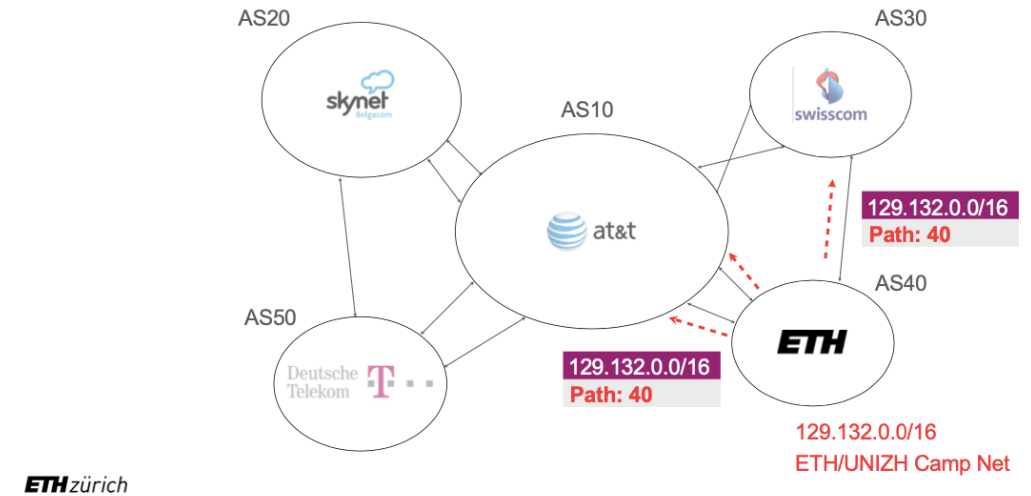


SPCA in perspective

Computer Networks:

From intra- and interdomain routing protocols (OSPF, IS-IS, BGP) to protocols on LANs like ARP, DHCP etc.

BGP announcements carry complete AS-level path information instead of distances



SPCA in perspective

Grundlagenfächer

- **Computer Networks:** How does the internet work
- **Data Modelling and Data Bases:** How to store huge amount of data

Kernfächer

- **Computer Systems:** Operating- and Distributed Systems
- **Compiler Design:** How to build a compiler

SPCA in perspective

Data Modelling and

Databases: Database

systems are as complex

as operating systems

today



```
dvdrental=# select title, release_year, length, replacement_cost from film
dvdrental=#   where length > 120 and replacement_cost > 29.50
dvdrental=#   order by title desc;
      title      | release_year | length | replacement_cost
-----+-----+-----+-----
West Lion        | 2006         | 159    | 29.99
Virgin Daisy     | 2006         | 179    | 29.99
Uncut Suicides   | 2006         | 172    | 29.99
Tracy Cider      | 2006         | 142    | 29.99
Song Hedwig      | 2006         | 165    | 29.99
Slacker Liaisons | 2006         | 179    | 29.99
Sassy Packer     | 2006         | 154    | 29.99
River Outlaw     | 2006         | 149    | 29.99
Right Cranes     | 2006         | 153    | 29.99
Quest Mussolini  | 2006         | 177    | 29.99
Poseidon Forever | 2006         | 159    | 29.99
Loathing Legally | 2006         | 140    | 29.99
Lawless Vision   | 2006         | 181    | 29.99
Jingle Sagebrush | 2006         | 124    | 29.99
Jericho Mulan    | 2006         | 171    | 29.99
Japanese Run     | 2006         | 135    | 29.99
Gilmore Boiled   | 2006         | 163    | 29.99
Floats Garden    | 2006         | 145    | 29.99
Fantasia Park    | 2006         | 131    | 29.99
Extraordinary Conqueror | 2006         | 122    | 29.99
Everyone Craft   | 2006         | 163    | 29.99
Dirty Ace        | 2006         | 147    | 29.99
Clyde Theory     | 2006         | 139    | 29.99
Clockwork Paradise | 2006         | 143    | 29.99
Ballroom Mockingbird | 2006         | 173    | 29.99
(25 rows)
```

SPCA in perspective

Grundlagenfächer

- **Computer Networks:** How does the internet work
- **Data Modelling and Data Bases:** How to store huge amount of data

Kernfächer

- **Computer Systems:** Operating- and Distributed Systems
- **Compiler Design:** How to build a compiler

SPCA in perspective

Computer Systems (OS

Part): What is an

Operating System: how
to execute processes

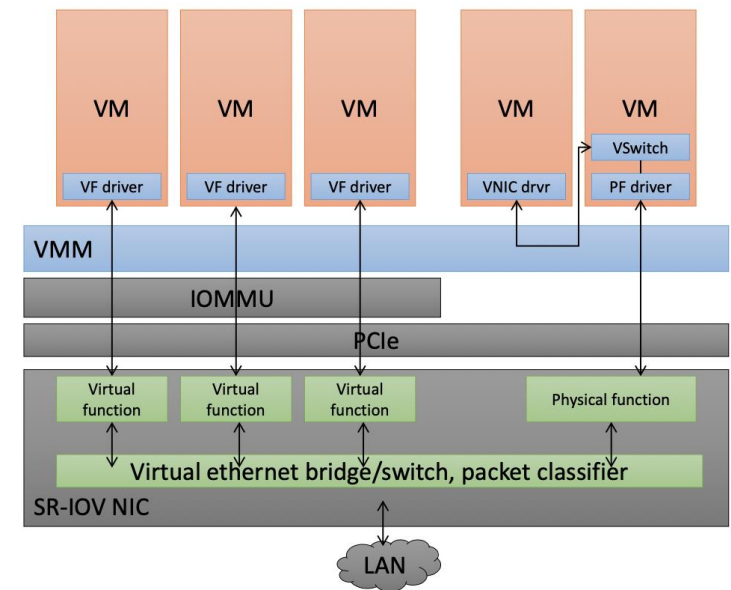
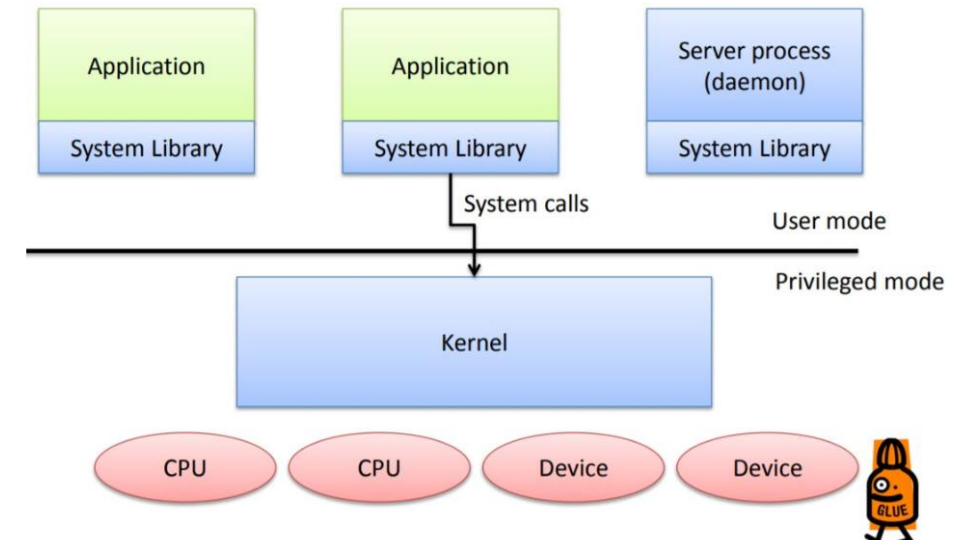
(Kernel, Processes,

Scheduling, IO, VM, File

Systems, Network

Stack, VMs)

General model of OS structure



SPCA in perspective

Computer Systems (DS

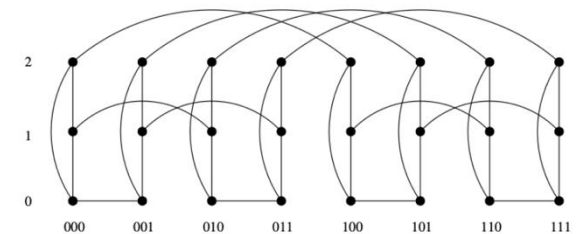
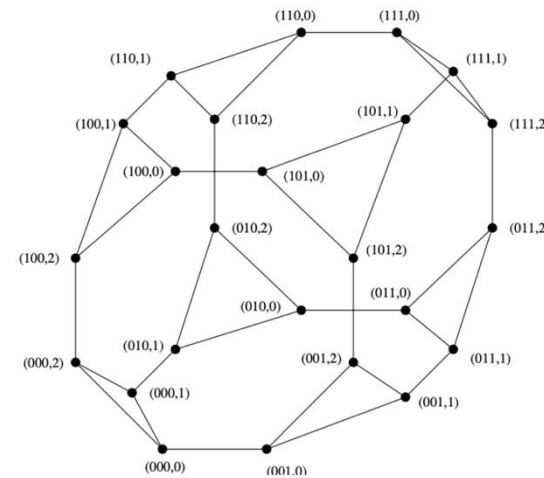
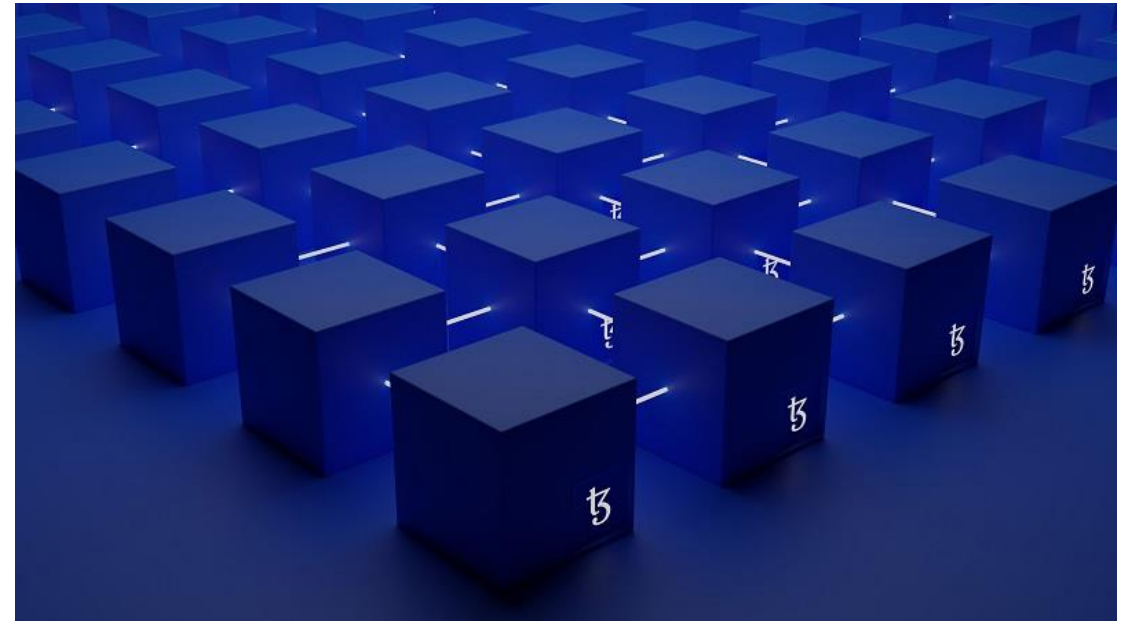
Part): Byzantine

Agreement, Quorum

Systems, Game Theory,

Advanced Blockchain

(Bitcoin, Ethereum)



SPCA in perspective

Grundlagenfächer

- **Computer Networks:** How does the internet work
- **Data Modelling and Data Bases:** How to store huge amount of data

Kernfächer

- **Computer Systems:** Operating- and Distributed Systems
- **Compiler Design:** How to build a compiler

SPCA in perspective

Compiler Design: Build your own
Compiler for the OAT language

```
#include <stdio.h>
#include "functions.h"

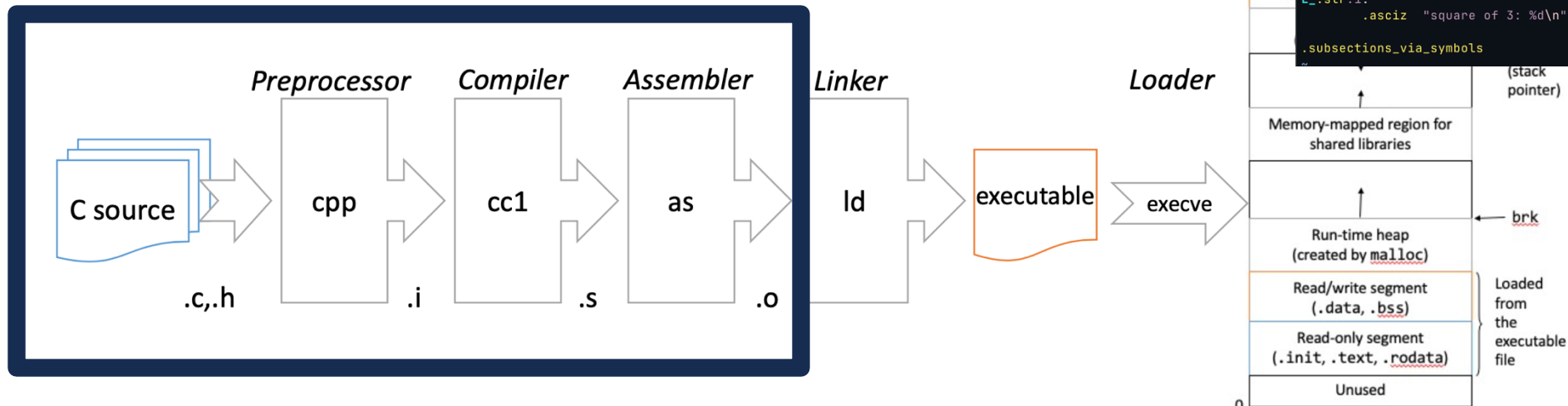
int main(int argc, char** argv){
    printf("hello, world\n");
    printf("square of 3: %d\n", square(3));
    return 0;
}
```

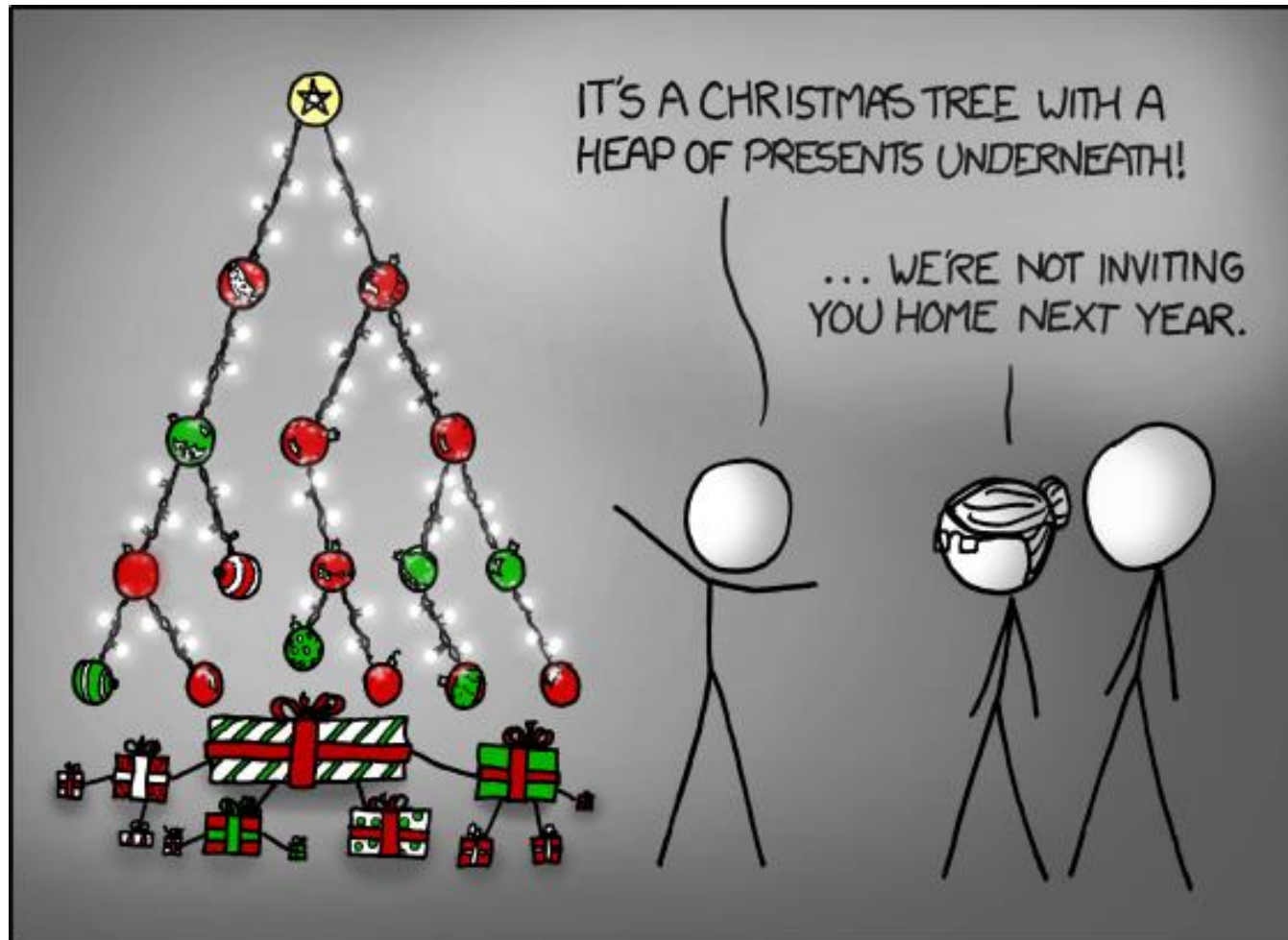
```
.section      __TEXT,__text,regular,pure_instructions
.build_version macos, 15, 0      sdk_version 15, 0
.globl _main                      ## -- Begin function main
.p2align     4, 0x90
_main:
.cfi_startproc
## %bb.0:
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
movl    %edi, -8(%rbp)
movq    %rsi, -16(%rbp)
leaq    L_.str(%rip), %rdi
movb    $0, %al
callq   _printf
movl    $3, %edi
callq   _square
movl    %eax, %esi
leaq    L_.str.1(%rip), %rdi
movb    $0, %al
callq   _printf
xorl    %eax, %eax
addq    $16, %rsp
popq    %rbp
retq
.cfi_endproc
## -- End function

.section      __TEXT,__cstring,cstring_literals
L_.str:
.asciz    "hello, world\n"
## @.str

L_.str.1:
.asciz    "square of 3: %d\n"
## @.str.1

.subsections_via_symbols
```



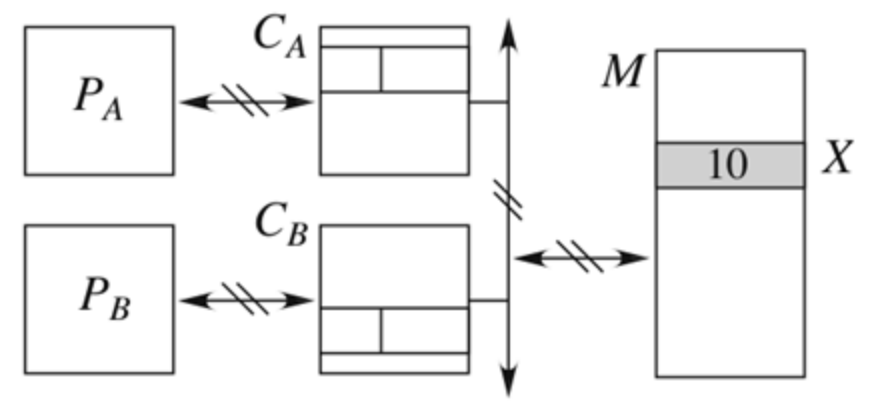


Merry Christmas
and all the best in
the new year!

Overview

- Assignment 11
- Devices Recap
- Hints for Assignment 12
- Exam Strategies

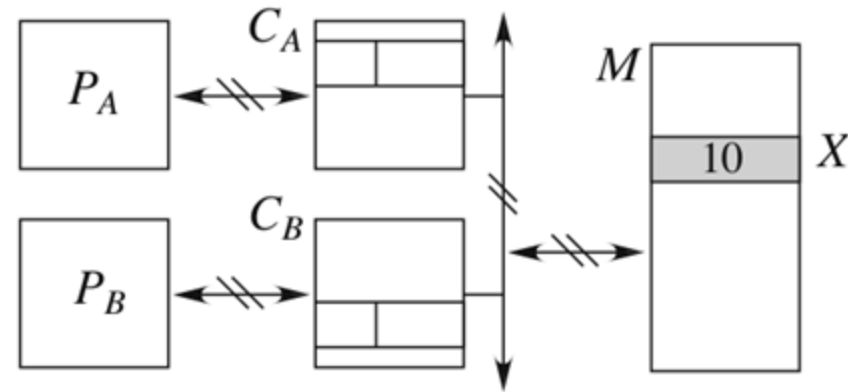
Question 1a: MSI



n	P_A	P_B	Comment
1	mov (X),r1		P_A : r1 := (X)
2		mov (X),r4	P_B : r4 := (X)
3	mov \$0,(X)		P_A : (X) := 0
4		mov (X),r5	P_B : r5 := (X)
5		mov \$20,(X)	P_B : (X) := 20

n	State C_A	Value C_A	State C_B	Value C_B	Value M
1					
2					
3					
4					
5					

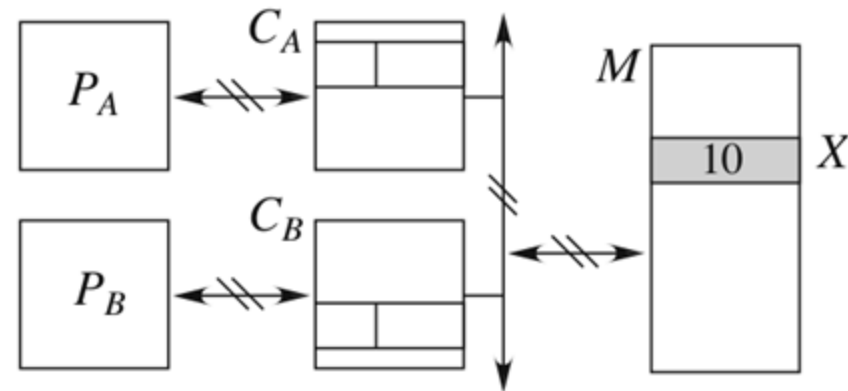
Question 1a: MSI



n	P_A	P_B	Comment
1	mov (X),r1		P_A : r1 := (X)
2		mov (X),r4	P_B : r4 := (X)
3	mov \$0,(X)		P_A : (X) := 0
4		mov (X),r5	P_B : r5 := (X)
5		mov \$20,(X)	P_B : (X) := 20

n	State C_A	Value C_A	State C_B	Value C_B	Value M
1	S	10	I	–	10
2	S	10	S	10	10
3	M	0	I	–	10
4	S	0	S	0	0
5	I	–	M	20	0

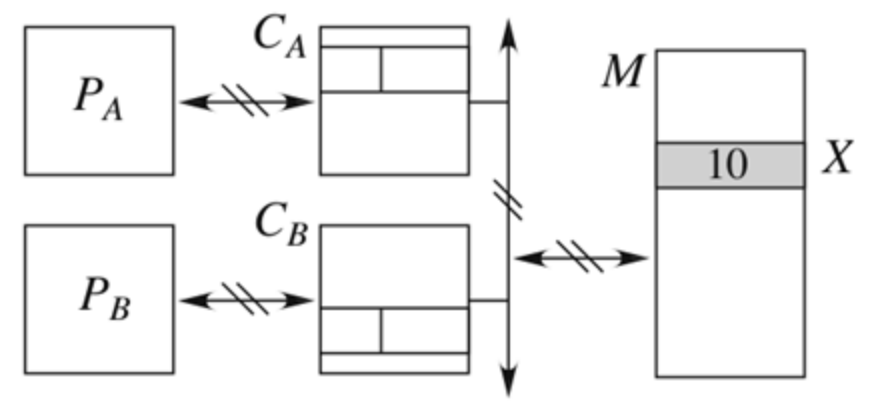
Question 1b: MESI



n	P_A	P_B	Comment
1	mov (X),r1		P_A : r1 := (X)
2		mov (X),r4	P_B : r4 := (X)
3	mov \$0,(X)		P_A : (X) := 0
4		mov (X),r5	P_B : r5 := (X)
5		mov \$20,(X)	P_B : (X) := 20

n	State C_A	Value C_A	State C_B	Value C_B	Value M
1					
2					
3					
4					
5					

Question 1b: MESI



n	P_A	P_B	Comment
1	mov (X),r1		P_A reads X
2		mov (X),r4	P_B reads X
3	mov \$0,(X)		P_A writes 0 to X
4		mov (X),r5	P_B reads X
5		mov \$20,(X)	P_B writes 20 to X

Only difference between MESI and MSI

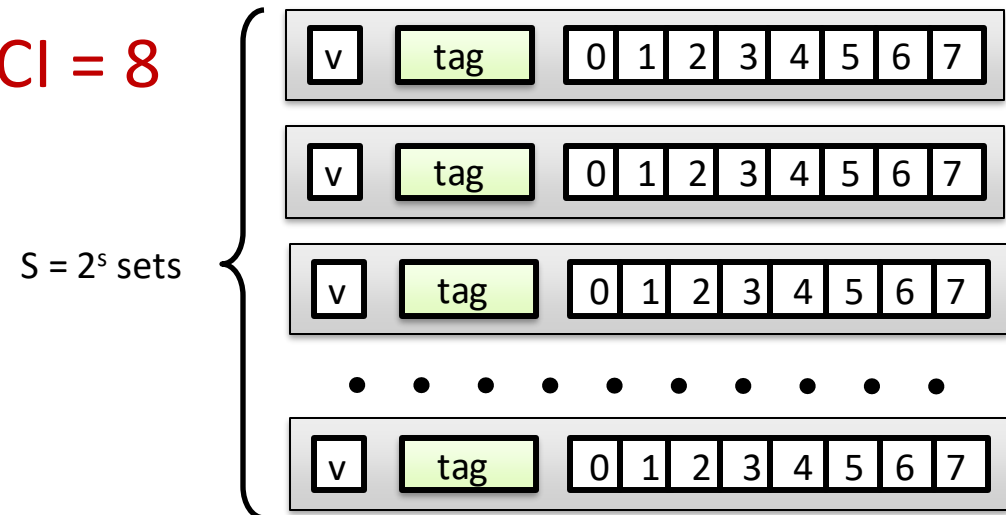
n	State C_A	Value C_A	State C_B	Value C_B	Value M
1	E	10	I	—	10
2	S	10	S	10	10
3	M	0	I	—	10
4	S	0	S	0	0
5	I	—	M	20	0

Question 2a

- 3 cores
- 128B direct-mapped write-back cache, 8B cache line size
- $X = 0xA0C0$

Q2a: Which cache line will be used by the three processors?

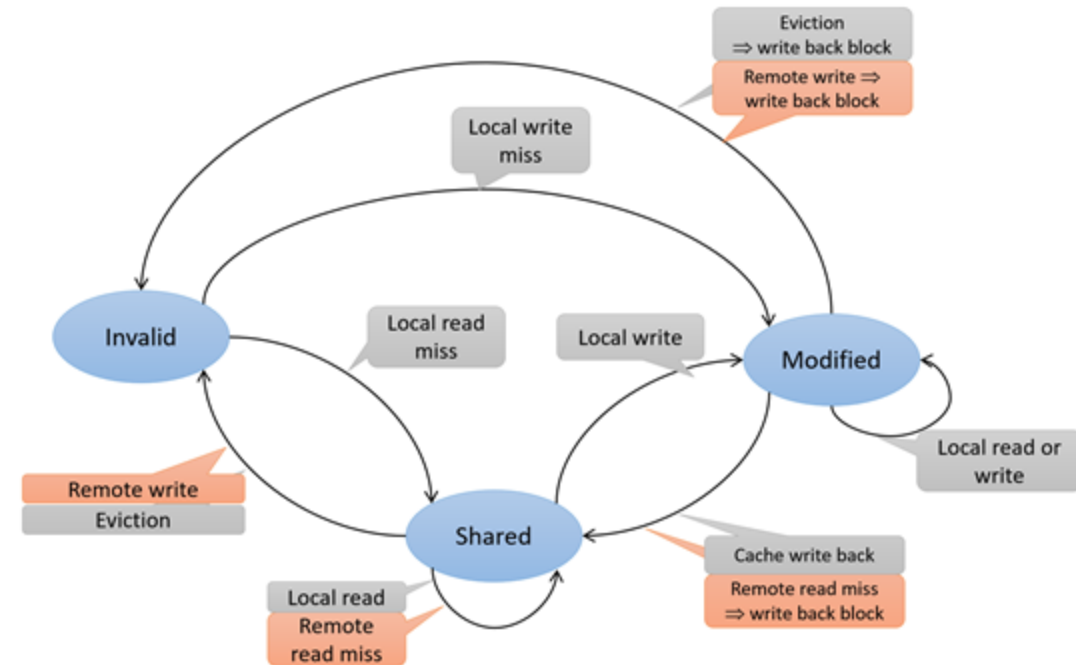
- 8B line size \Rightarrow CO = 3 bits
- 128B direct mapped \Rightarrow 16 sets/lines total \Rightarrow CI = 4 bits
- CI = bits [6:3]
- $0xA0C0 = 0b1010\ 0000\ 1100\ 0000$
- **CI = 8**



Question 2b

- 3 cores
- 128B direct-mapped write-back cache, 8B cache line size
- $X = 0xA0C0$
- Two-byte loads and stores in order:

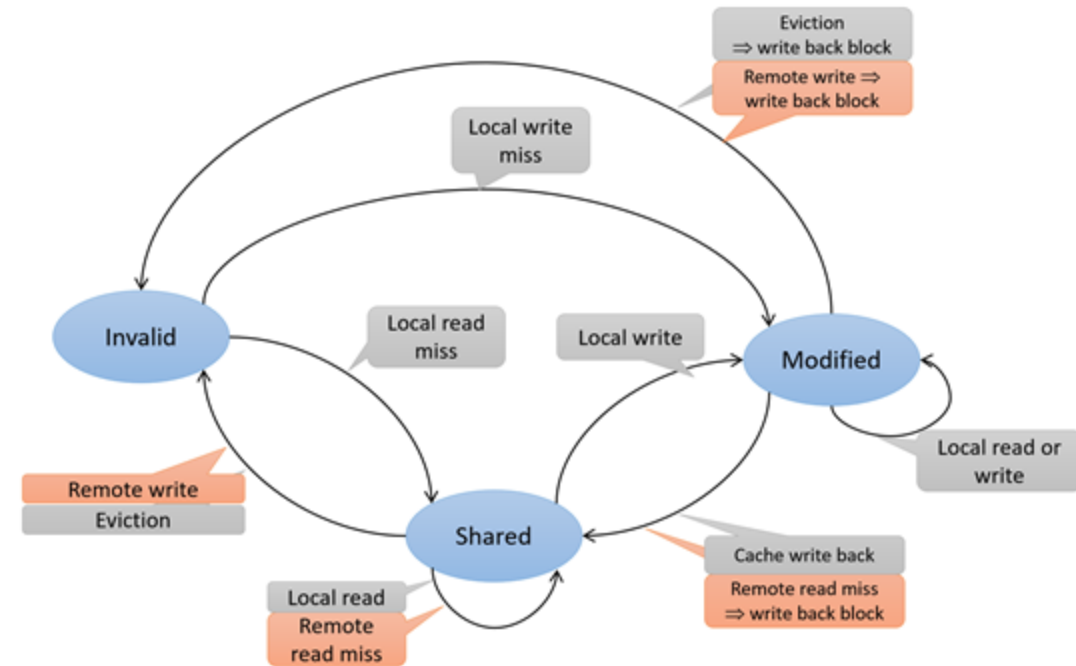
n	P_1	P_2	P_3
1	ld r1, [X]
2	add r1, 1
3	st r1, [X]
4	...	ld r2, [X+2]	...
5	...	and r2, 0FH	...
6	...	st r2, [X+2]	...
7	ld r3, [X+6]
8	sub r3, r1, r5
9	st r3, [X+6]



n	State C1	State C2	State C3	Remarks
1				
2				
3				
4				
5				
6				
7				
8				
9				

Question 2b

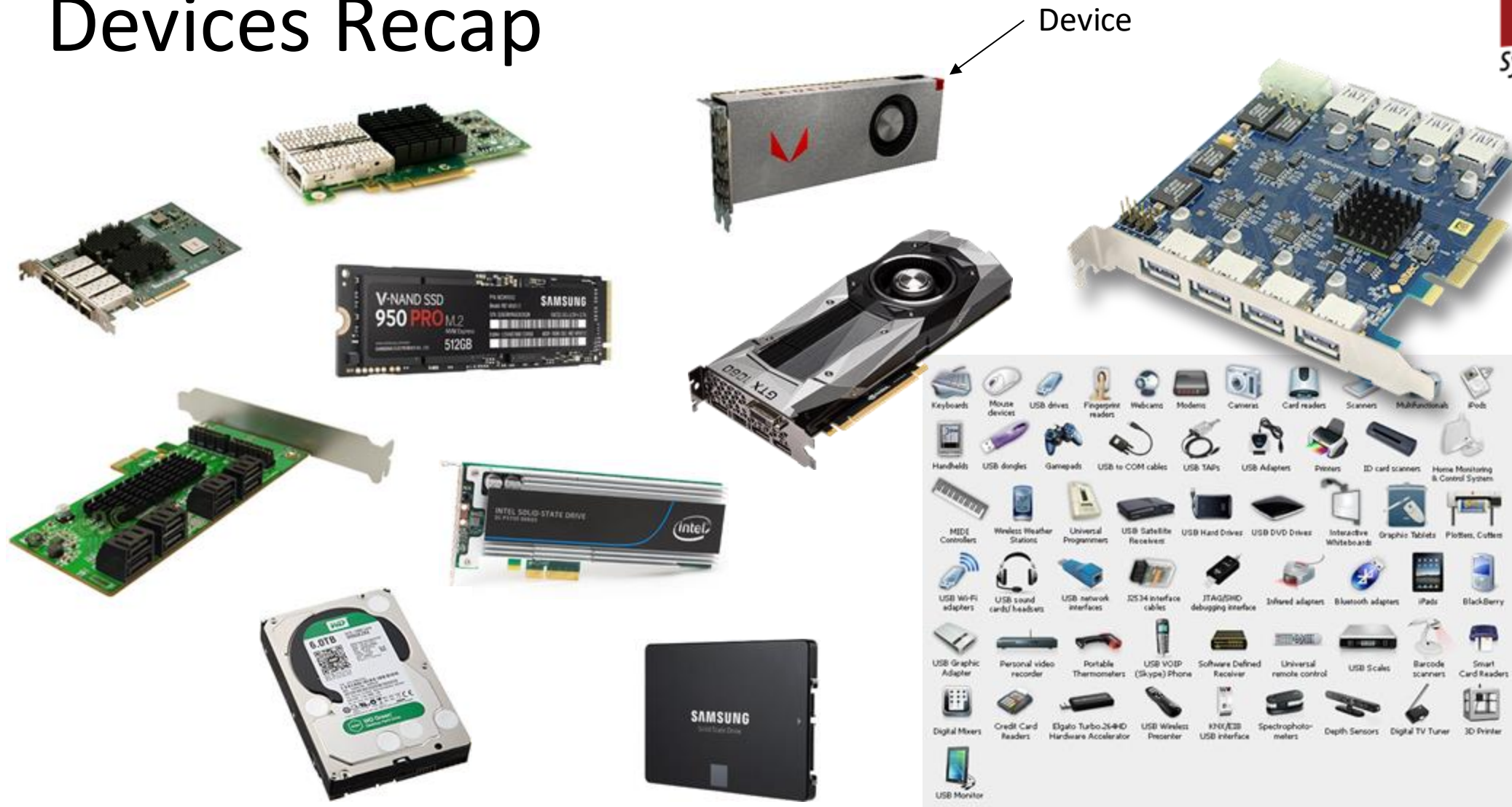
- 3 cores
- 128B direct-mapped write-back cache, 8B cache line size
- $X = 0xA0C0$
- Two-byte loads and stores in order:



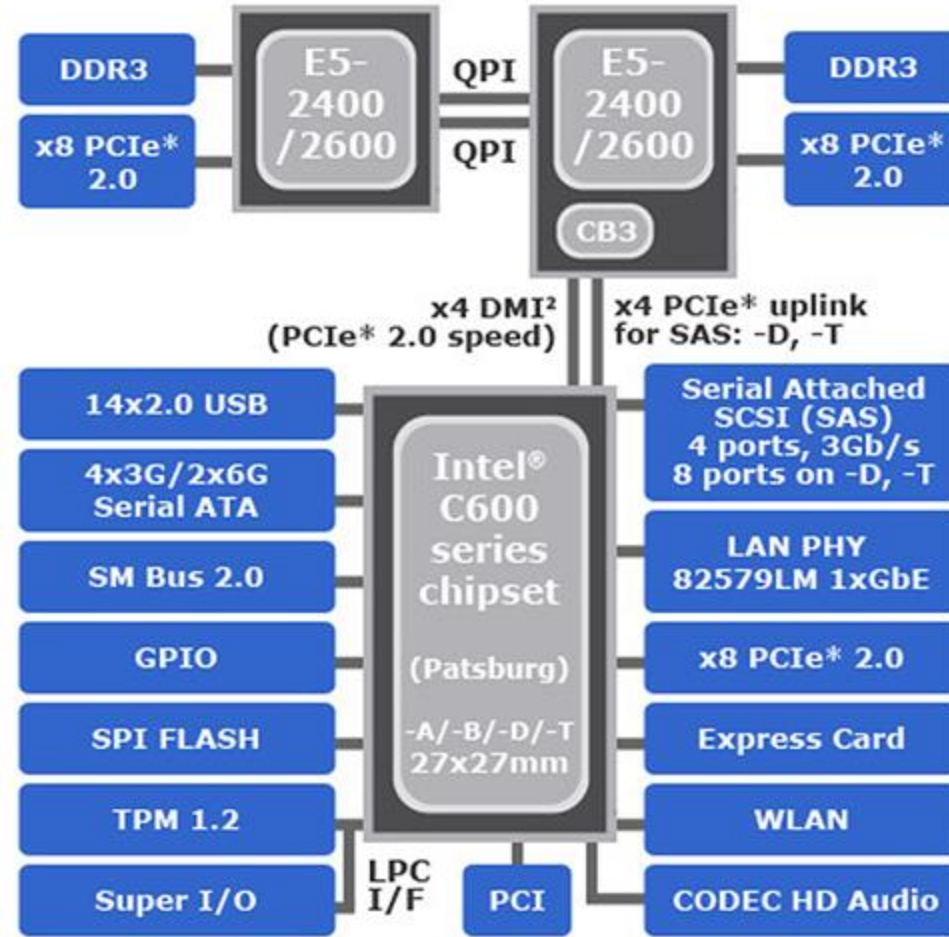
n	P_1	P_2	P_3
1	ld r1,[X]
2	add r1,1
3	st r1,[X]
4	...	ld r2,[X+2]	...
5	...	and r2,0FH	...
6	...	st r2,[X+2]	...
7	ld r3,[X+6]
8	sub r3,r1,r5
9	st r3,[X+6]

n	State C1	State C2	State C3	Remarks
1	S	I	I	
2	S	I	I	
3	M	I	I	
4	S	S	I	P1 writes back, P2 waits for writeback
5	S	S	I	
6	I	M	I	
7	I	S	S	P2 writes back, P3 waits for writeback
8	I	S	S	
9	I	I	M	

Devices Recap

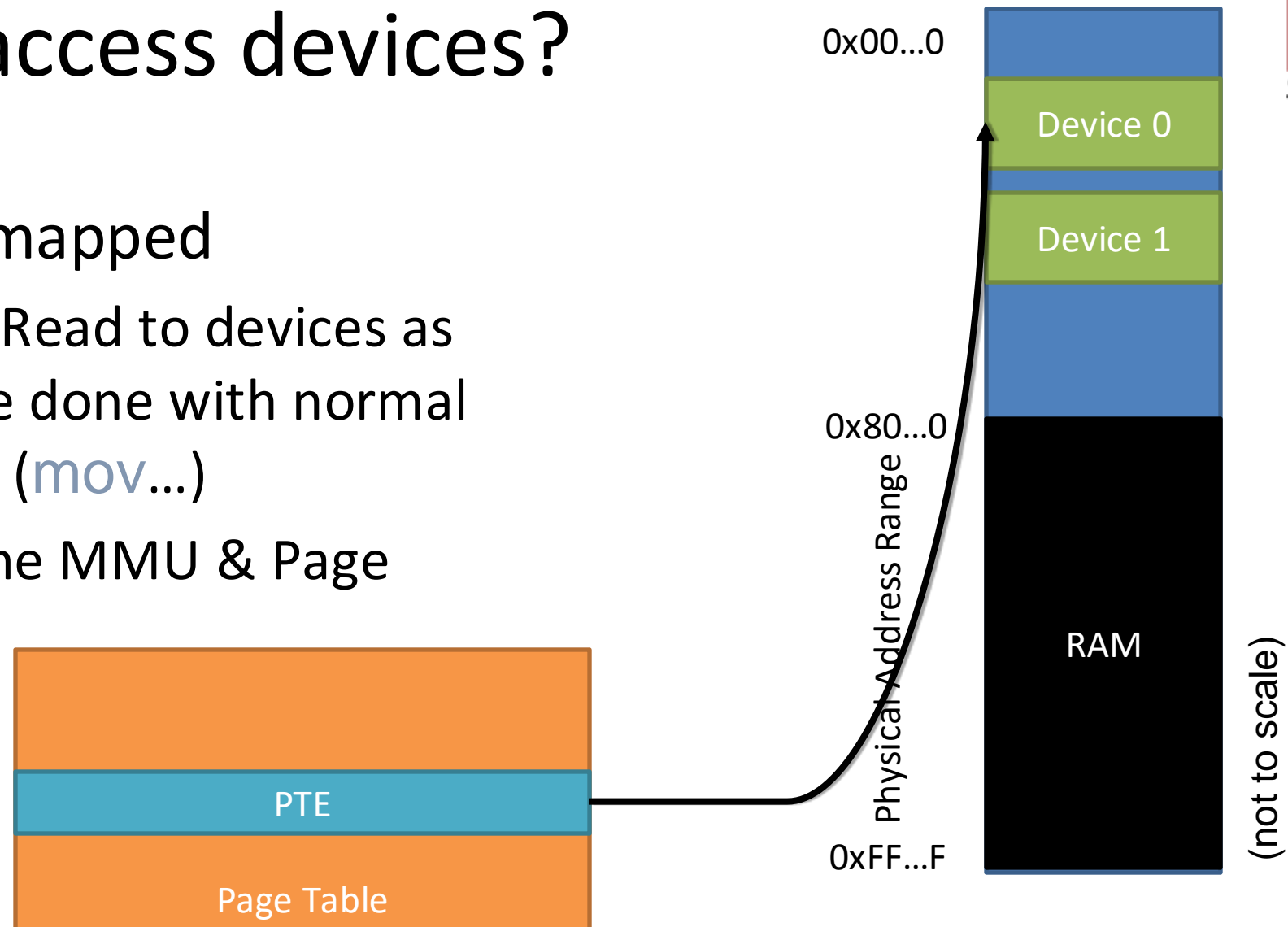


Devices



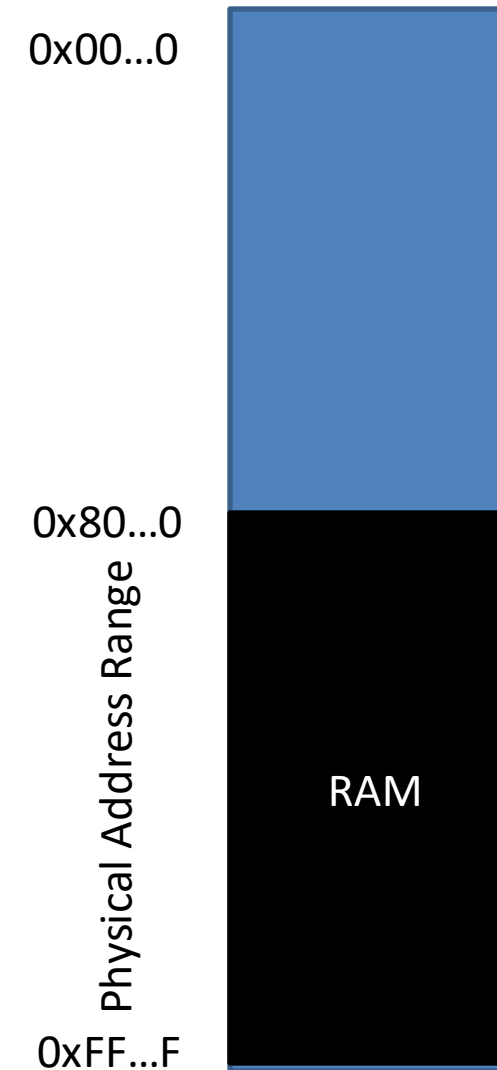
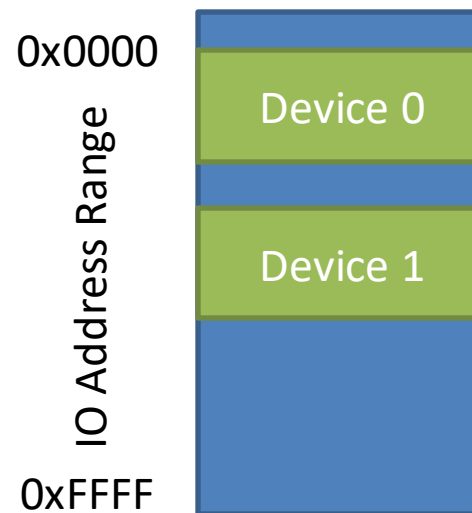
How to access devices?

- Memory mapped
 - Write & Read to devices as you have done with normal memory (`mov...`)
 - Use of the MMU & Page Tables



How to access devices?

- IO Ports
 - Special address space (16 bit)
 - Write to an IO port
 - special instructions
inb, outb



Memory Mapped Devices

- Device represented as [Base Address, Length]
 - Base address refers to the physical address where the device starts
 - cf: your beginning of the stack frame
 - agreed upon by OS and device driver upon device startup
 - Length refers to the total size of the memory region occupied by the device
 - cf: the size of your stack frame
 - includes devices registers and if required descriptor ring
 - Set of registers within Base, Base+Length
 - cf: your variables on the stack
 - used to set control bits etc.

Devices are **NOT** memory

- Contents of device registers may change unexpectedly from view of CPU
 - Data received
- Writing to a register may trigger actions
 - Shutdown device/machine
 - Perform reset

ns16550 registers (each 8 bits)

Addr.	Name	Description	Notes
0	RBR	Receive Buffer Register (read only)	DLAB=0
0	THR	Transmit Holding Register (write only)	DLAB=0
1	IER	Interrupt Enable Register	DLAB=0
2	IIR	Interrupt Identification Register (read only)	
2	FCR	FIFO Control Register (write only)	
3	LCR	Line Control Register	
4	MCR	MODEM Control Register	
5	LSR	Line Status Register	
6	MSR	MODEM Status Register	
7	SCR	Scratch Register	
0	DLL	Divisor Latch (LSB)	DLAB=1
1	DLM	Divisor Latch (MSB)	DLAB=1

DLAB = bit 7 of the LCR register

Too simple UART driver

```
1. #define UART_BASE 0x12345000
2. #define UART_THR (UART_BASE + 0)
3. #define UART_RBR (UART_BASE + 4)
4. #define UART_LSR (UART_BASE + 8)
```

```
1. void serial_putc(char c)
2. {
3.     char *lsr = (char *) UART_LSR;
4.     char *thr = (char *) UART_THR;

1.     // Wait until FIFO can hold more chars
2.     while((*lsr & 0x20) == 0);
3.
4.     *thr = c
5. }
```

What's the problem here?

Too simple UART driver

```
1. #define UART_BASE 0x12345000
2. #define UART_THR (UART_BASE + 0)
3. #define UART_RBR (UART_BASE + 4)
4. #define UART_LSR (UART_BASE + 8)

1. void serial_putc(char c)
2. {
3.     volatile char *lsr = (char *) UART_LSR;
4.     volatile char *thr = (char *) UART_THR;

1.     // Wait until FIFO can hold more chars
2.     while((*lsr & 0x20) == 0);
3.
4.     *thr = c
5. }
```

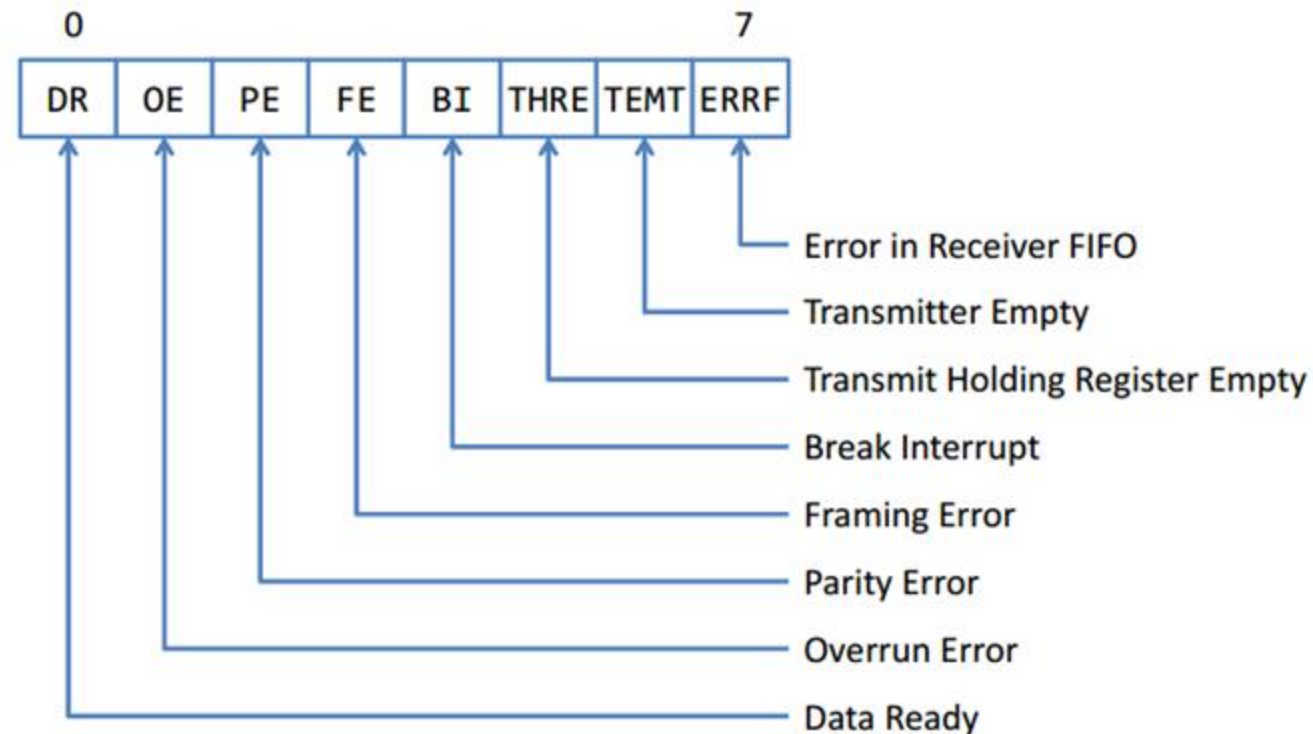
What's the problem here?

- Compiler does not know this is a device register!
- Loop gets optimized away!
- Add: **volatile** keyword

Again: this volatile is not the same as in java

Device Register Contents

- Each bit may have a different meaning
- Lots of configurations possible!



Device Drivers

- Writing device drivers is tedious
 - Setting a single bit wrong and the device does not work
 - Debugging is hard: Likely to set a bit wrong due to a wrong shift & mask
 - Device manuals not always available



Devices and Caches

- Device registers cannot be cached due to inconsistency problem, i.e. register content changes without CPU write!
- What about cache lines? Would overwrite other register values when writing back
- Set the “no-cache” flag in the page table entry

Interrupts

```
1. #define UART_BASE 0x12345000
2. #define UART_THR (UART_BASE + 0)
3. #define UART_RBR (UART_BASE + 4)
4. #define UART_LSR (UART_BASE + 8)

1. void serial_putc(char c)
2. {
3.     volatile char *lsr = (char *) UART_LSR;
4.     volatile char *thr = (char *) UART_THR;

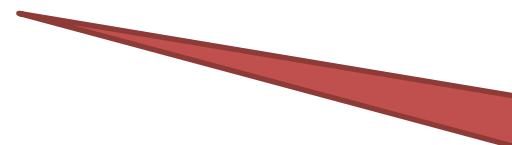
1.     // Wait until FIFO can hold more chars
2.     while((*lsr & 0x20) == 0);
3.
4.     *thr = c
5. }
```

What's the problem with this code?

Efficiency problem: Polling...

- CPU can't do any useful work while busy waiting

Solution: Use **Interrupts**, i.e. tell device to “call this function when data ready”

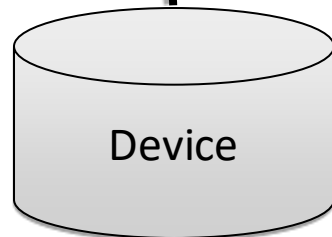
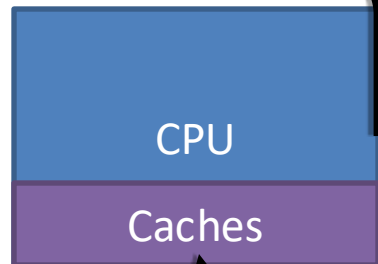


Register polling
=
wasted CPU cycles

Data transfer

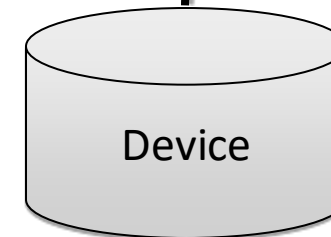
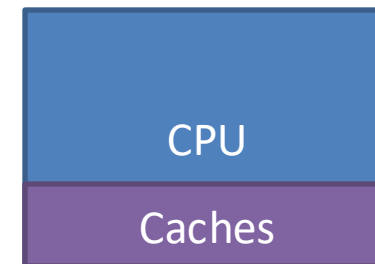
- Don't want to waste CPU cycles just copying data
- Use Direct Memory Access instead
- Now CPU and DMA can work in parallel
- need to make sure we maintain consistency with CPU caches!

Direct Memory Access (DMA)



CPU writes data to memory i.e. also cache

Data path goes Through CPU

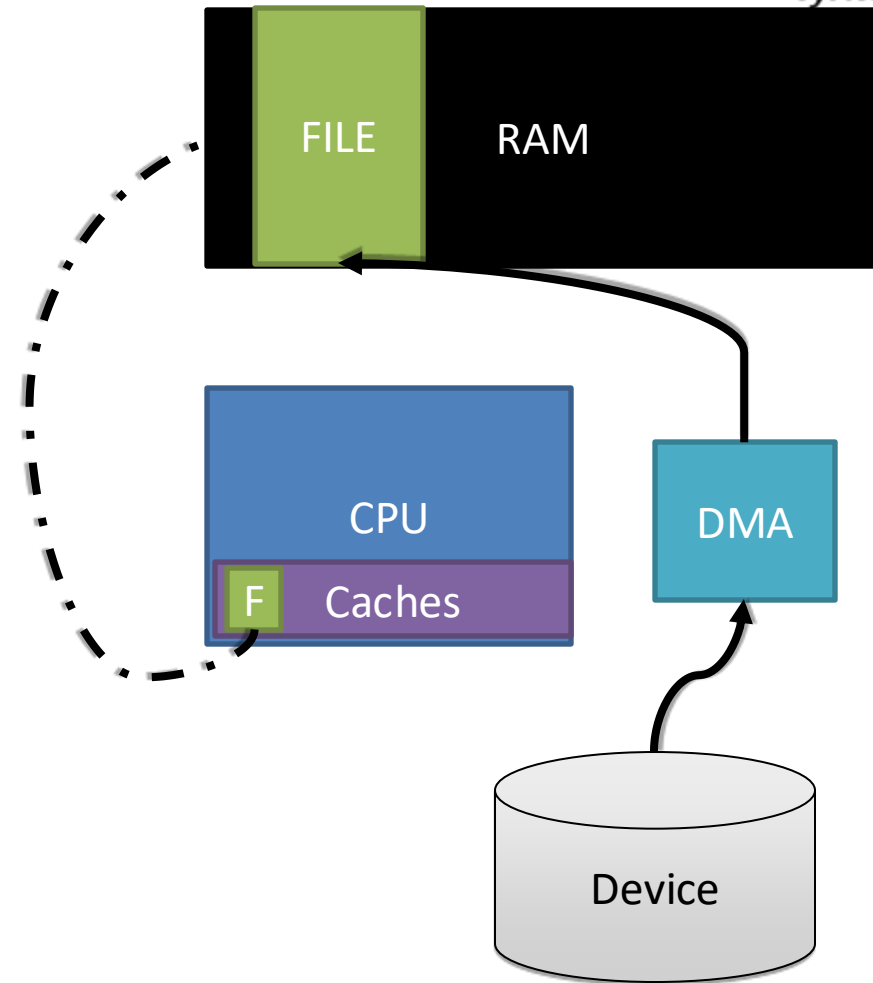


DMA controller autonomously transfers data to/from devices, skipping CPU entirely

Any Problems with DMA?

DMA and Caches

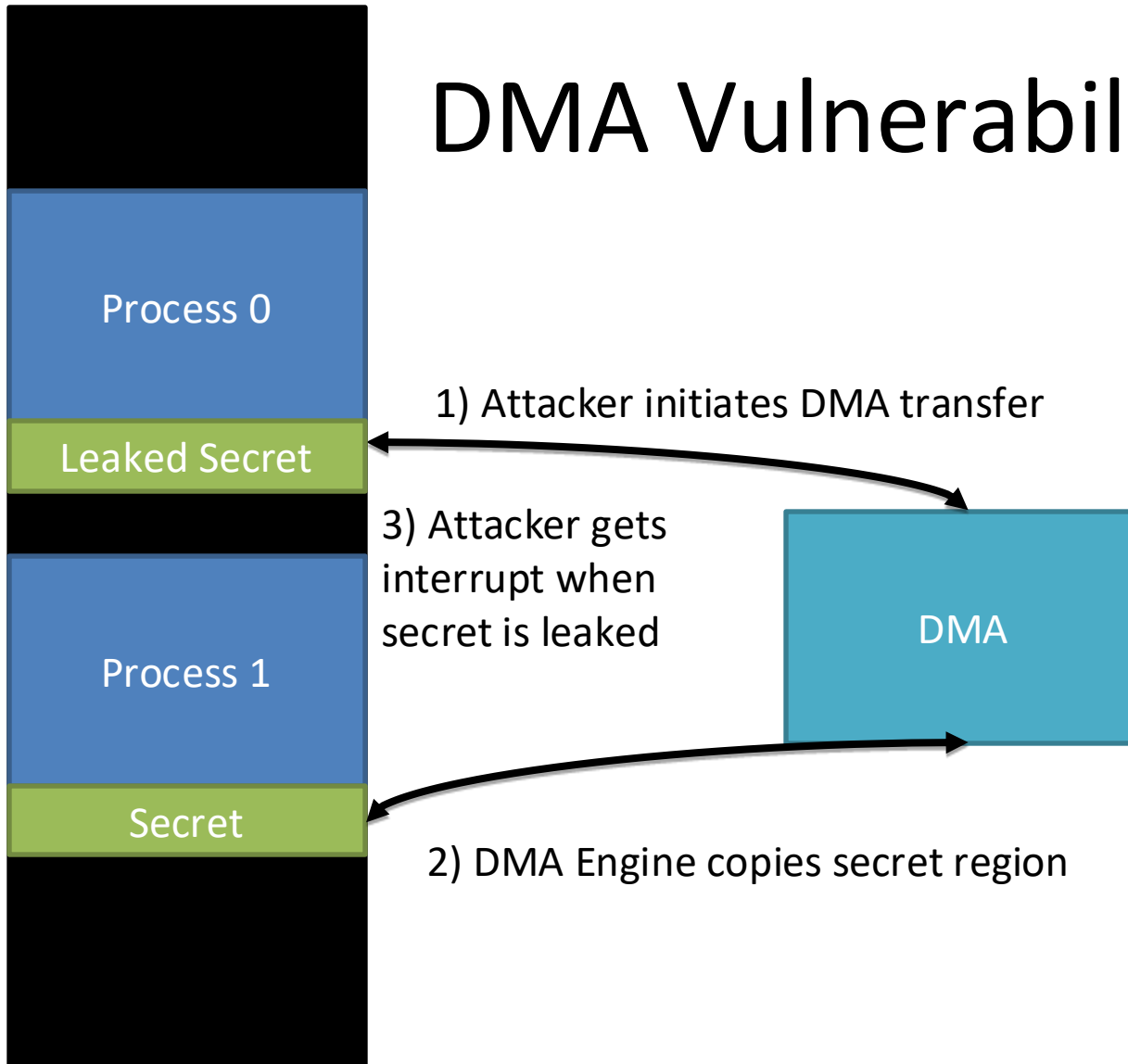
- DMA is like device registers!
Changes the contents w/o CPU write.
- Cannot “no-cache”:
Bad performance since large data
- Need to explicitly invalidate cache!



DMA Addressing

- Deal with physical addresses only!
 - Any problems with that?
- Programs deal with virtual addresses: need translation!
- Physical Range may not be contiguous
 - Scatter-gather DMA controllers: DMA to/from a list of regions
- How about security?

DMA Vulnerability



Attack done by
external devices i.e.
firewire / thunderbolt

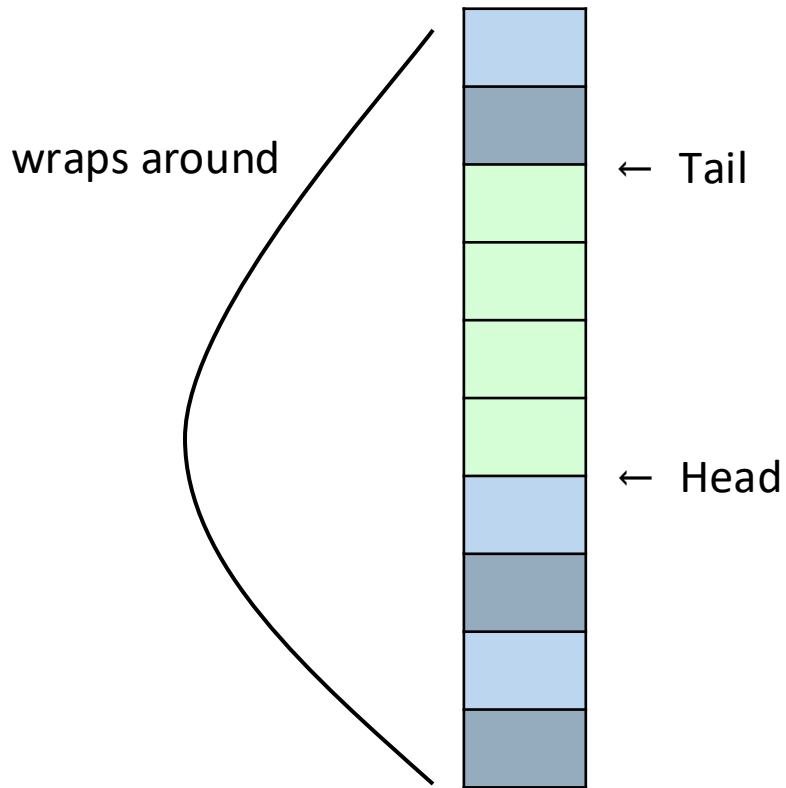
https://en.wikipedia.org/wiki/DMA_attack

DMA and Cache Problems

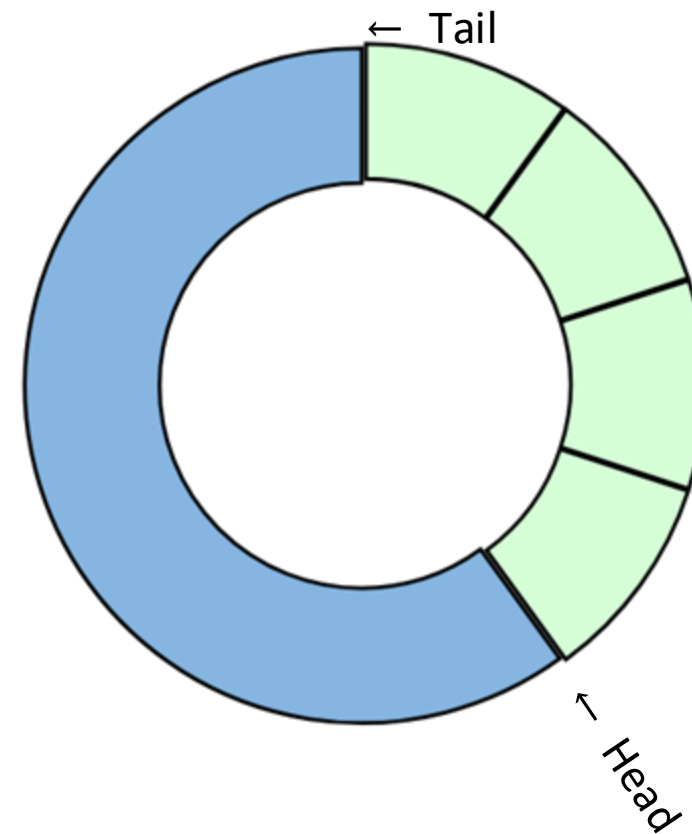
- On DMA read:
 - Before: Flush (= write back all) the cache to update main memory
 - After: Invalidate (= clear all) the cache to avoid old values seen
- On DMA write:
 - Before: flush or invalidate the cache to update main memory
 - After: invalidate CPU cache

Buffer/Descriptor Rings

Actual View in Memory:



Logical View:



Buffer/Descriptor Rings

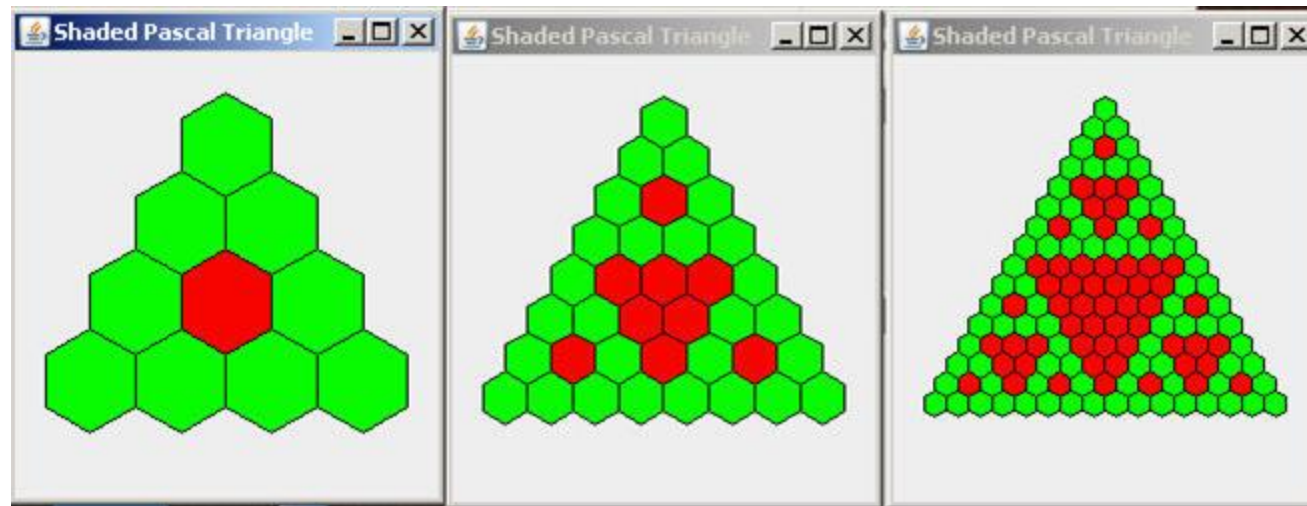
Buffer Ring

- ring holds metadata and actual data
- data must be fixed size
- data is contiguous

Descriptor Ring

- ring holds metadata and pointer (**descriptor**) to actual data
 - data can be variable sized
 - data doesn't have to be contiguous
- => can make descriptor chain

X-Max Assignment 12



An unknown MPFC (?) device appeared



☐ Ignore

☒ Write Driver

Task

- You will write a device driver for a MPFC device
- Device Specs
 - Uses single descriptor ring
 - Uses DMA transfers
 - Uses interrupts for “new item” or “no buffer” signals
 - Memory Mapped Device

Assignment TO-DO

- Initialize the device (reset) and setting up the in memory data structures
- Start the device
- Start issuing DMA requests to the device
- Activate interrupt and go to sleep
- Do not terminate! Hand back the buffers to the device once used

Hints: Interrupts

- You will need to register a handler which is called for received interrupts
- Interrupt received:
 - Wake the sleeping thread
 - Acknowledge the interrupt

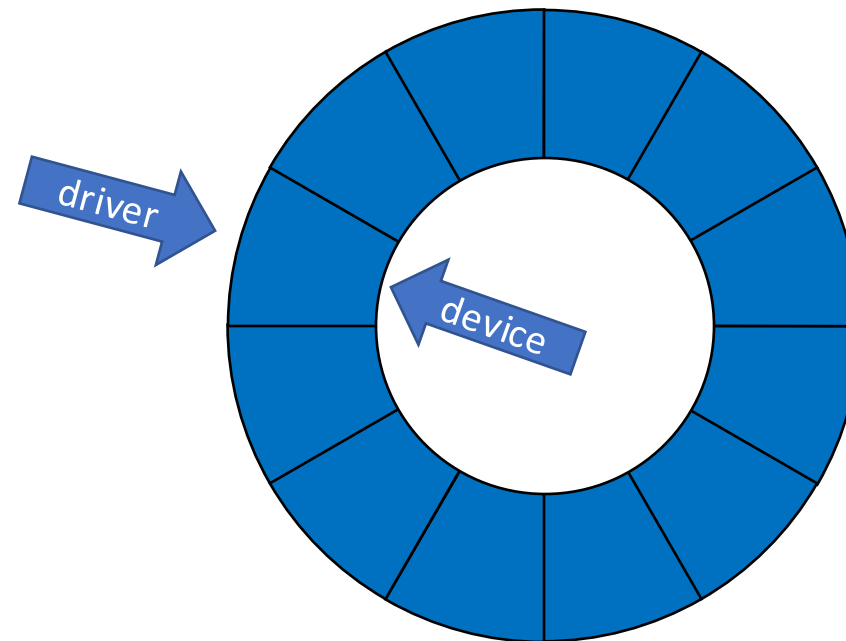
Hints: Buffer Ring

- Allocate enough memory
- Keep track of who owns the buffer
- Tell the device where to find the buffer rings!
 - (Normally you would have to give the physical address, but we stay virtual this time)

Hints: Buffer Ring

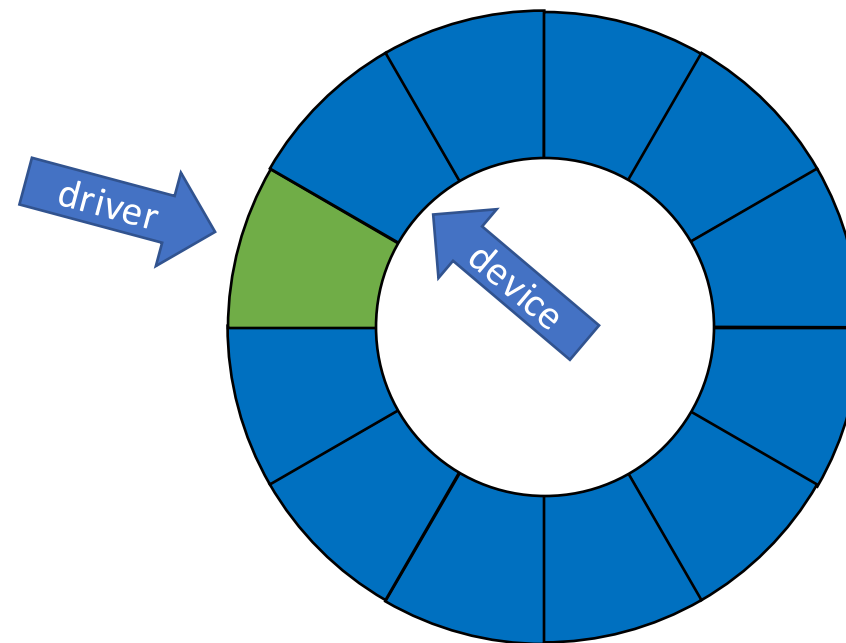
```
typedef struct {  
    size_t size;  
    void *buffer;  
    char owned;  
} mpfc_desc;
```

For receive:
Driver = read-pointer
Device = write-pointer



Hints: Buffer Ring

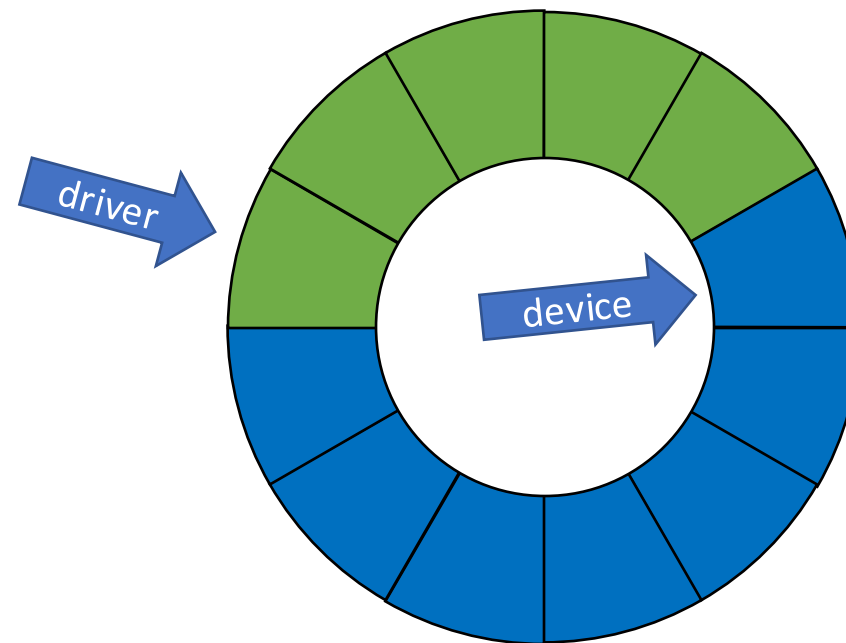
```
typedef struct {  
    size_t size;  
    void *buffer;  
    char owned;  
} mpfc_desc;
```



- Owned by driver
- Owned by device

Hints: Buffer Ring

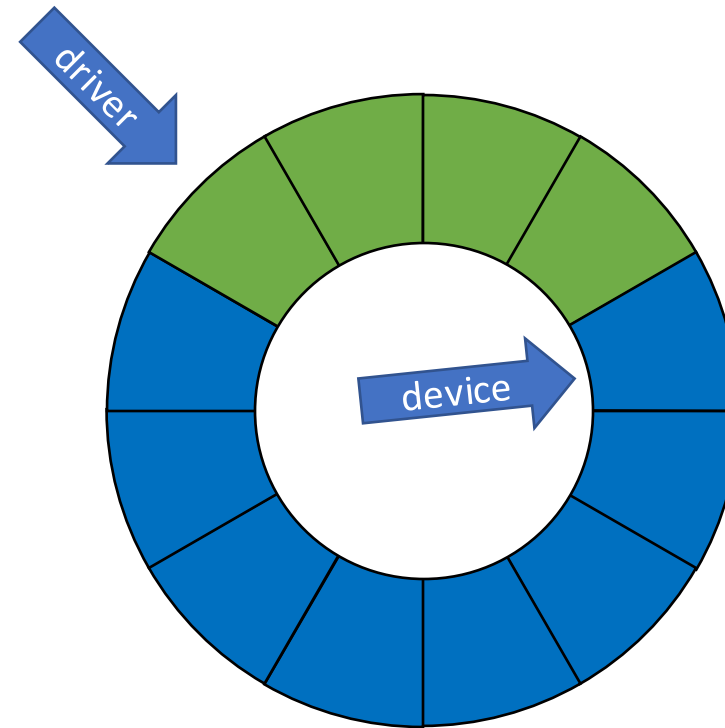
```
typedef struct {  
    size_t size;  
    void *buffer;  
    char owned;  
} mpfc_desc;
```



- Owned by driver
- Owned by device

Hints: Buffer Ring

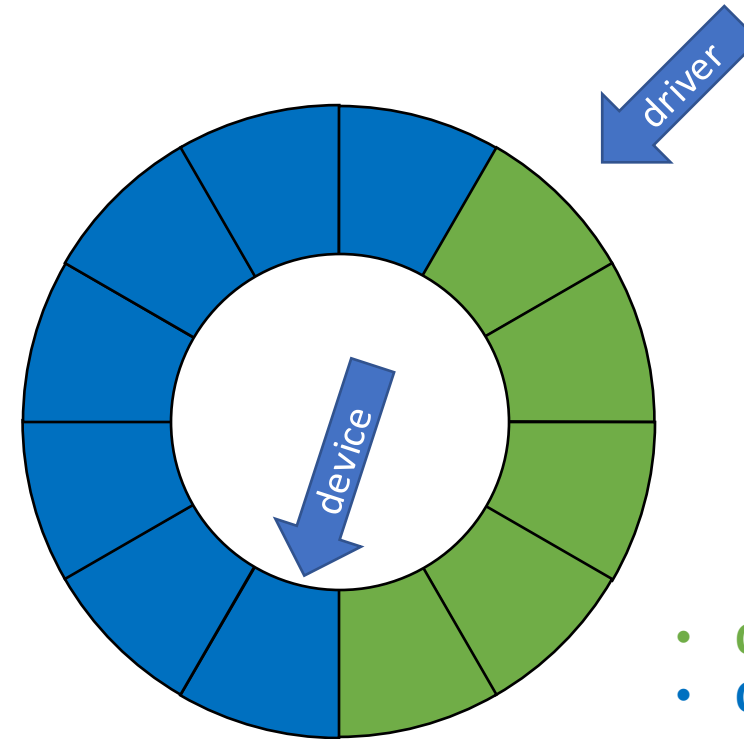
```
typedef struct {  
    size_t size;  
    void *buffer;  
    char owned;  
} mpfc_desc;
```



- Owned by driver
- Owned by device

Hints: Buffer Ring

```
typedef struct {  
    size_t size;  
    void *buffer;  
    char owned;  
} mpfc_desc;
```

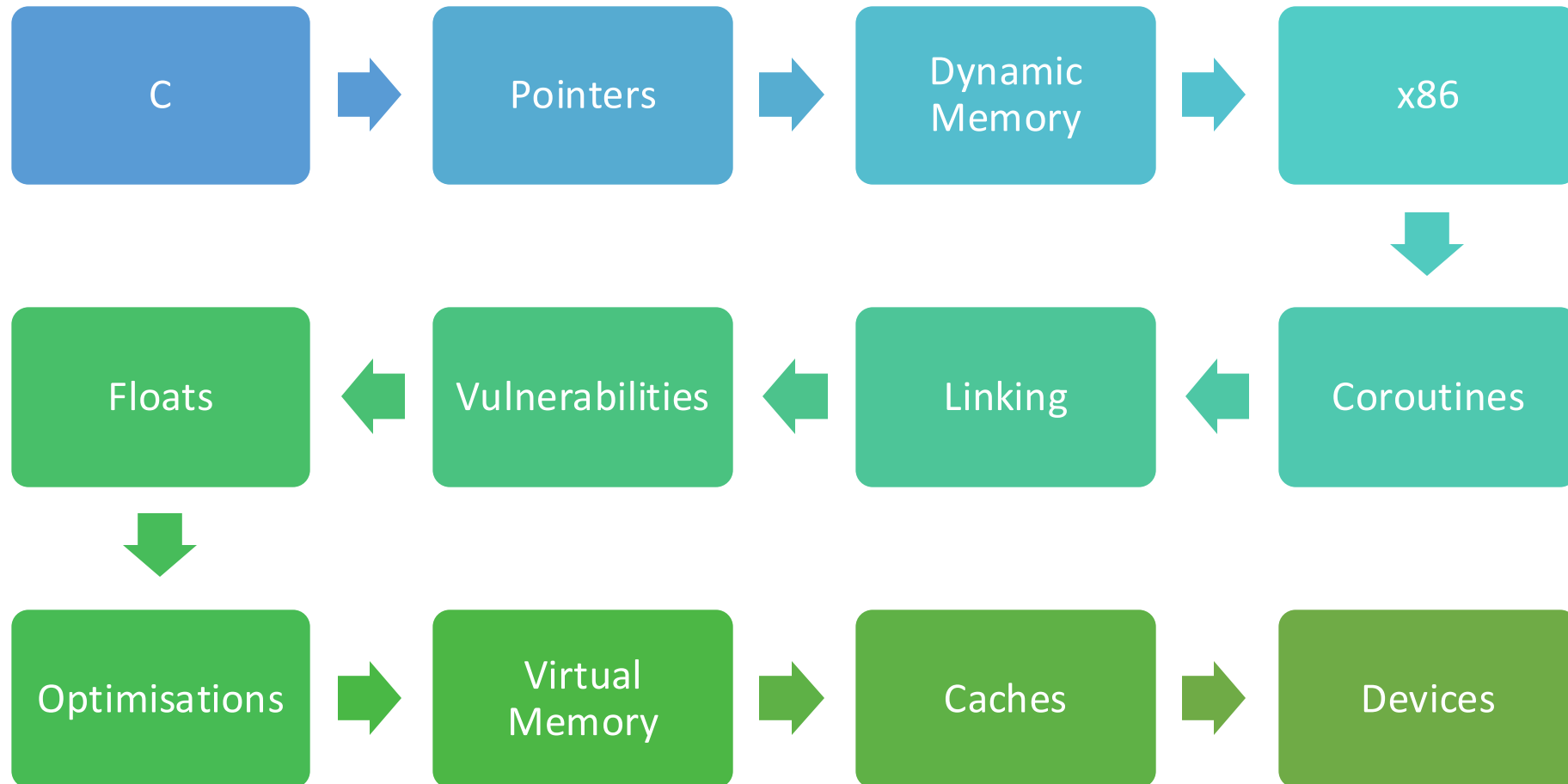


- Owned by driver
- Owned by device

Hints: Buffer Ring

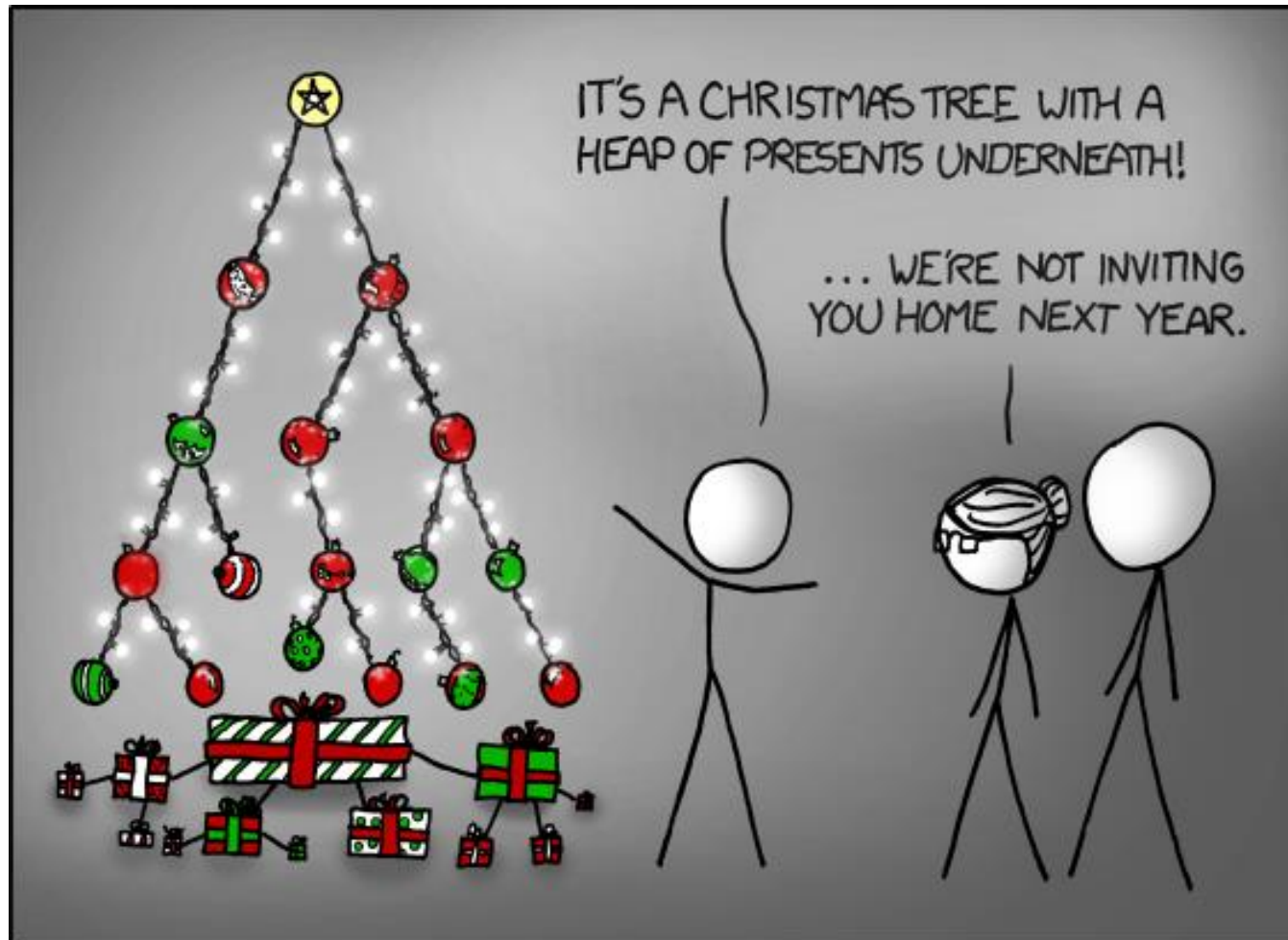
- Overruns and underruns (receive)
 - Device has no buffers for received packets
 - Then it starts discarding packets and signals that to CPU
 - CPU reads all received packets
 - Spin poll or tell device to do an interrupt when more data ready
- Overruns and underruns (transmit)
 - Device has no more packets to send
 - It must wait and either poll memory or tell CPU to wake it up
 - CPU has no more slots to send packets
 - Must wait until more slots are free, again either polling or tell the device to send interrupt

Semester Recap



Exam Strategy

- 1 point \approx 1 minute
- read through all the exam first => **start with what you know!**
- practice C coding
 - C intuition is built through practice
 - labs, code expert, old exercises in C, AoC in C, etc ...
- Stay calm :)



Merry Christmas
and all the best in
the new year!