Input output

Polling vs interrupt, Interrupt controllers, interrupt descriptor table, interrupt handlers, Direct memory access, hard disks

Abhilash Jindal

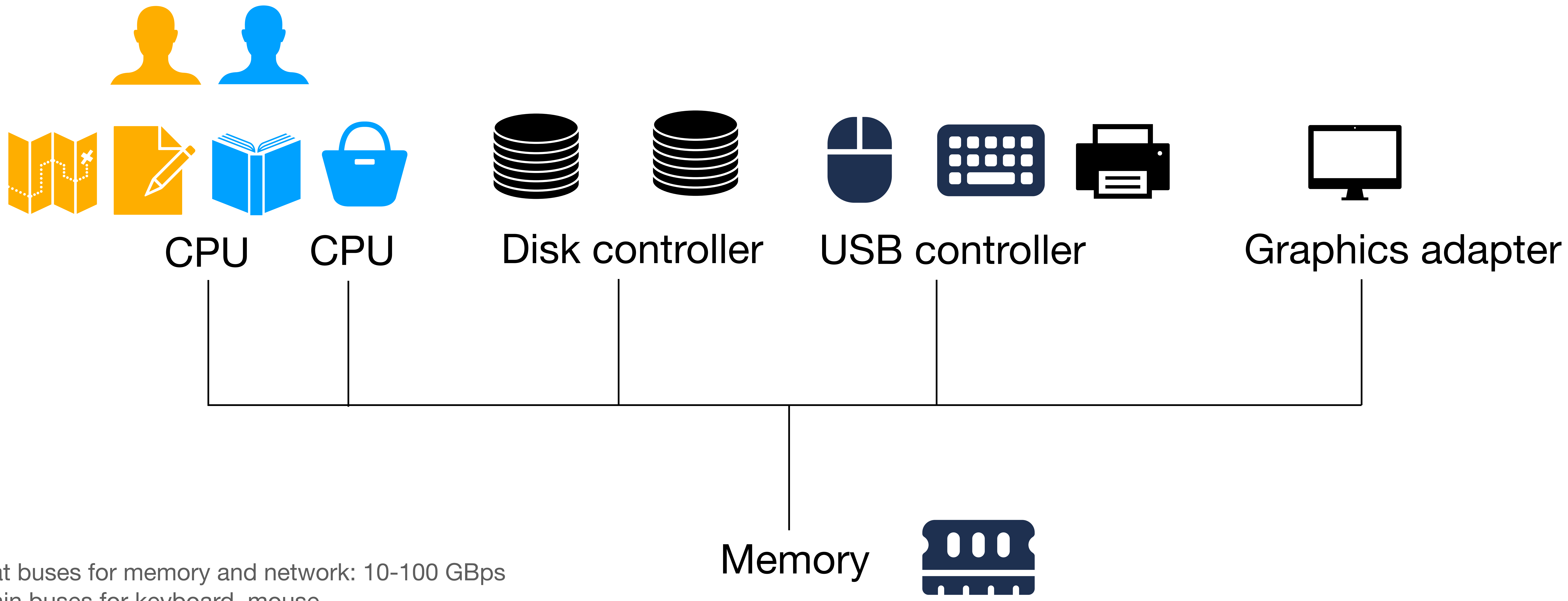
Agenda

- Overview of IO devices (OSTEP Ch. 36): Polling, Interrupts, Direct memory access
- Interrupt handling (xv6 Ch. 3): interrupt controllers, interrupt descriptor table
- Hard disk drives (OSTEP Ch. 37): disk geometry, disk scheduling
- Redundant Array of Inexpensive Disks (OSTEP Ch. 38): improve capacity, throughput, fault tolerance
- Disk driver, Buffer cache (xv6 Ch. 6)

Overview of IO devices

OSTEP Ch. 36

Computer organization

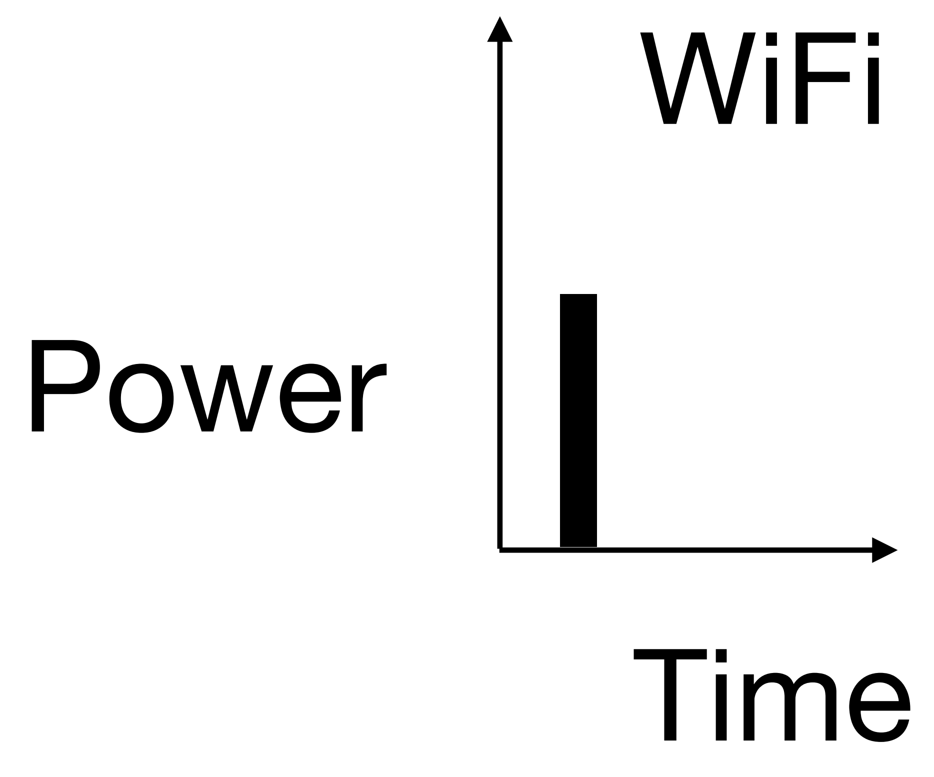


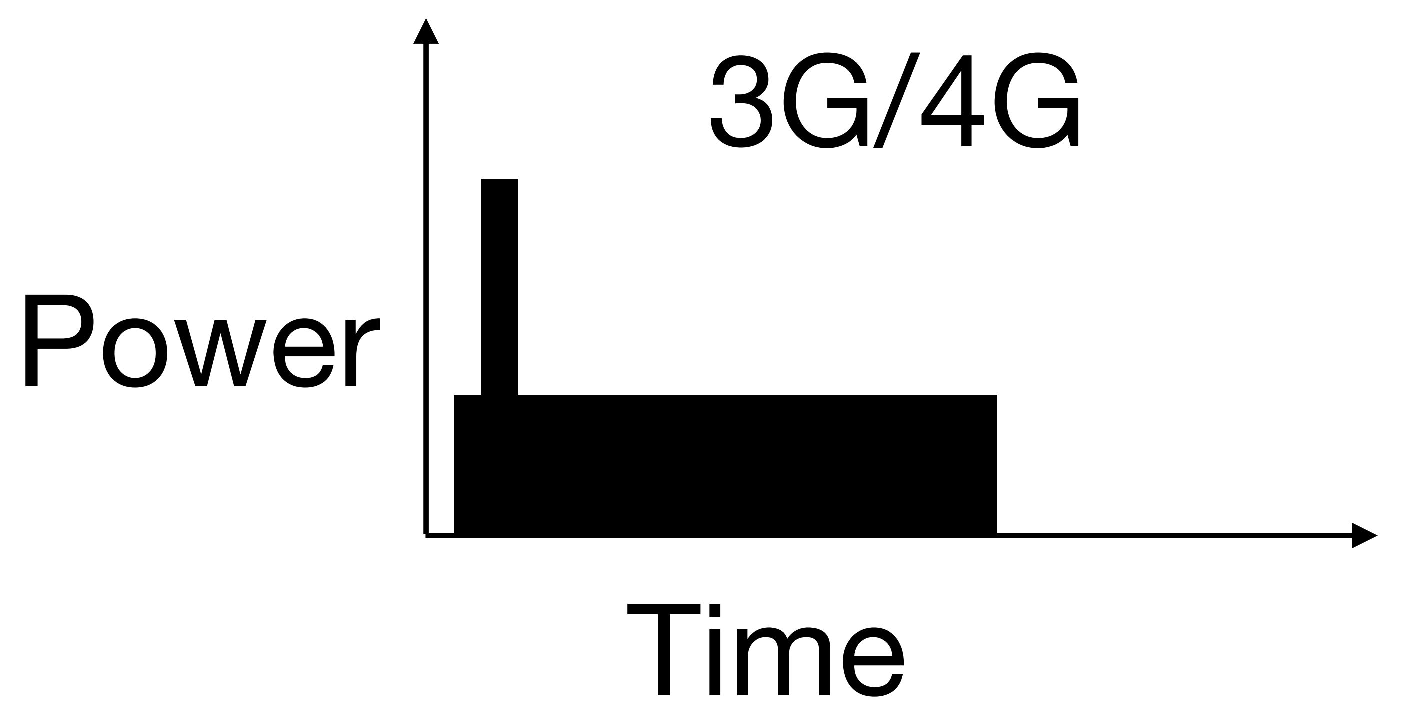
Fat buses for memory and network: 10-100 GBps
Thin buses for keyboard, mouse

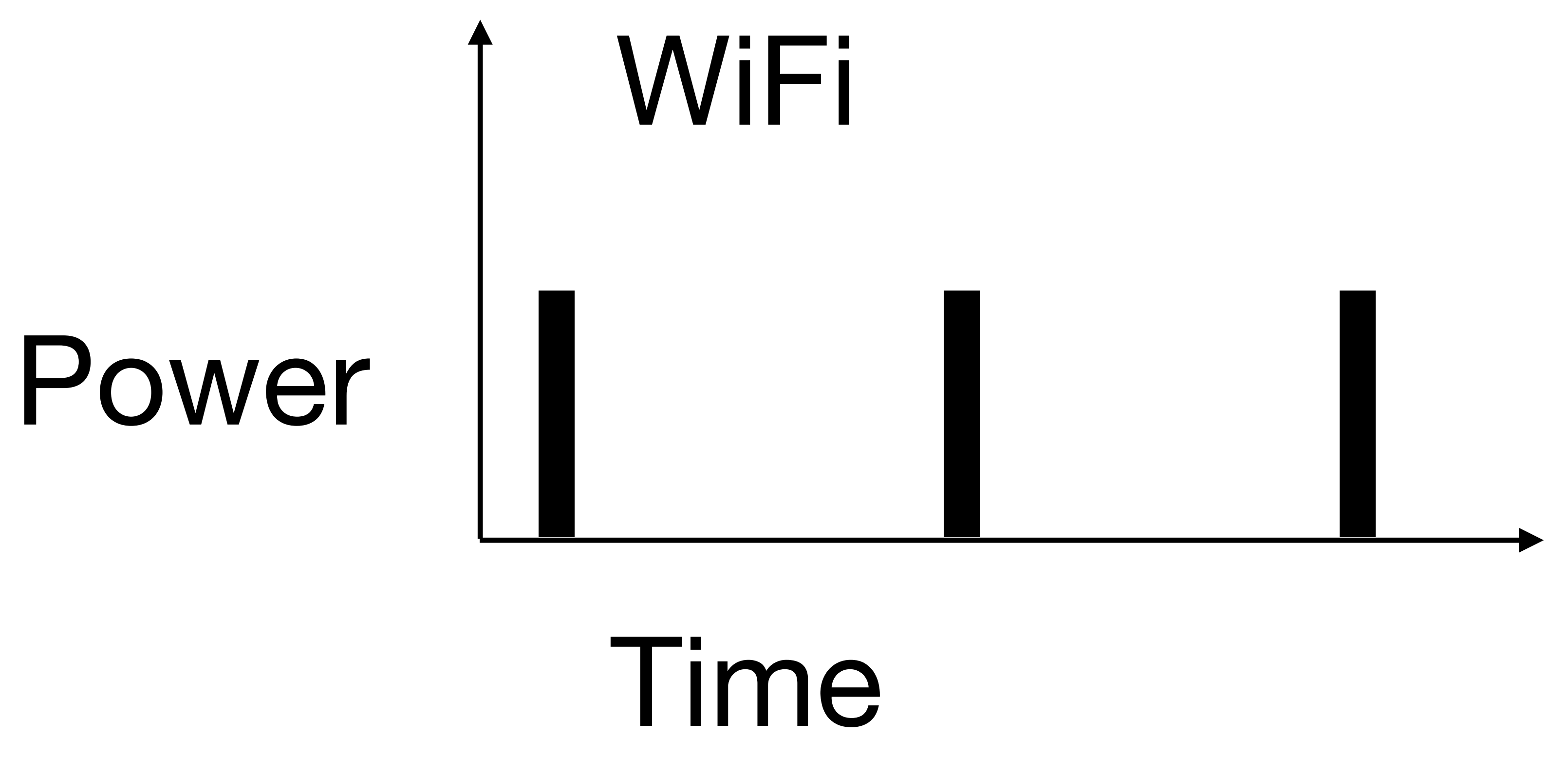
Fitting into the OS

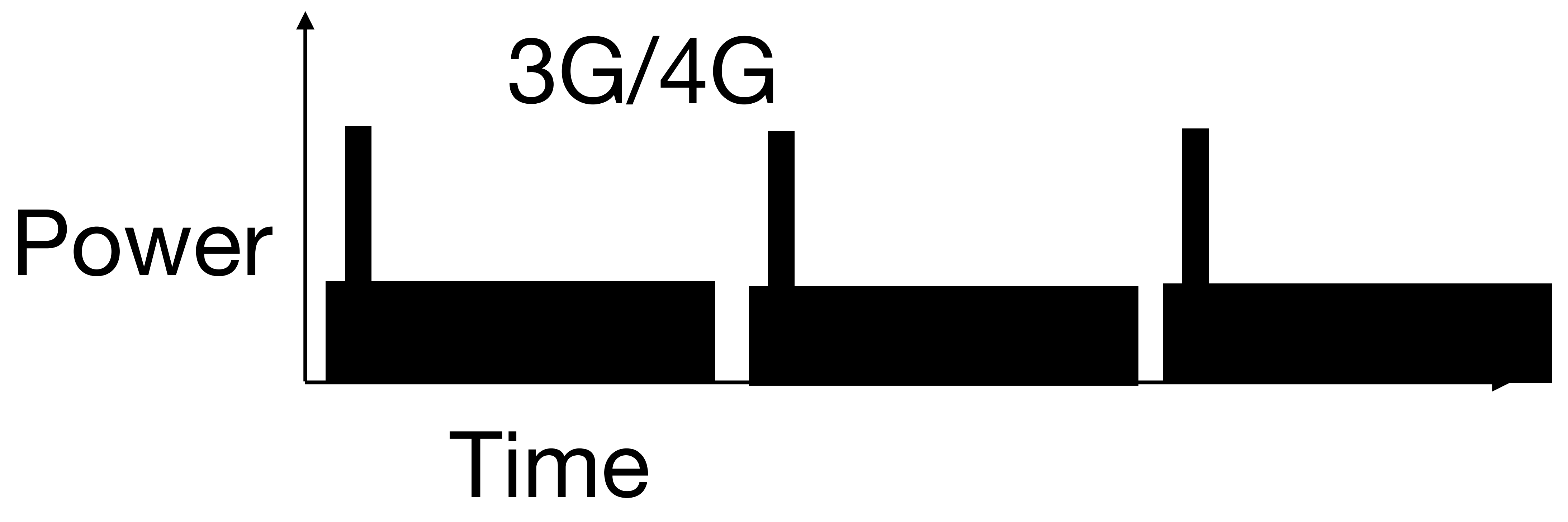
Hide device specific details in device driver

- Abstraction allows OS and applications to stay device-neutral
- Abstraction can hurt.
  - Example: 3G/4G are inefficient for small periodic pings
- > 70% of OS code is device drivers. Tend to have most number of bugs









Memory-mapped IO and Port-mapped IO

- Memory mapped:
  - Regular memory access instructions
  - Reads and writes are routed to appropriate device
  - Does not behave like memory! Reading same location twice can change due to external events (declare volatile)
- Port mapped:
  - Special IN and OUT instructions

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

Canonical protocol

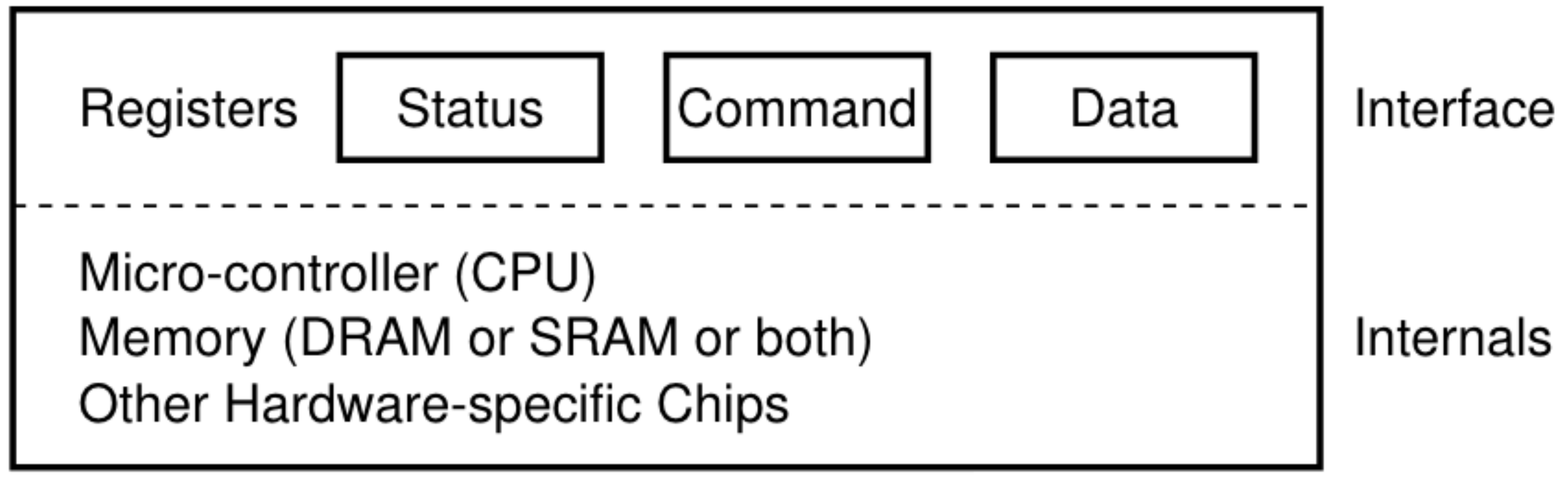


Figure 36.3: **A Canonical Device**

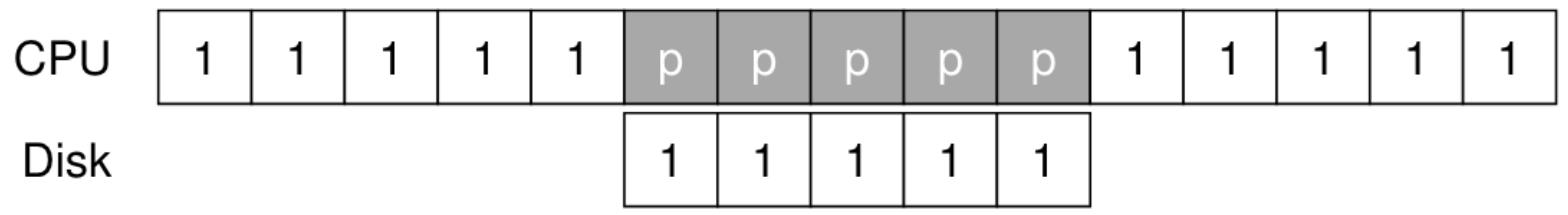
- Poll device until it is ready
- CPU cannot do anything else.
- Example: CPU needs to spend ~1 million instructions waiting for disk
- Ok for bootloader. It does not have anything else to do.
- Not ok for OS. It can run other processes.

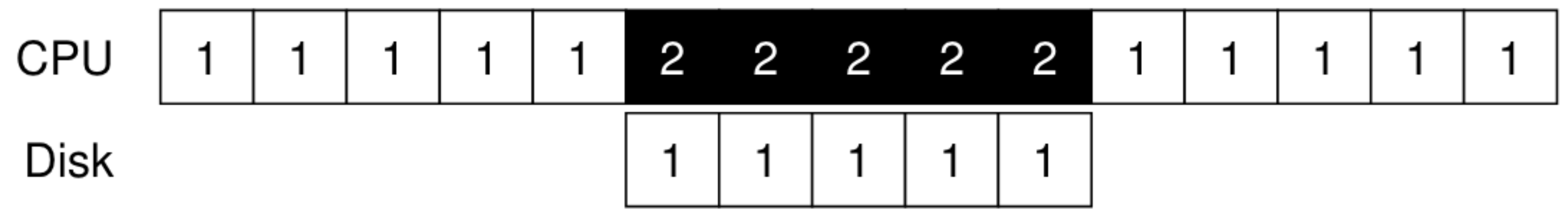
bootmain.c

```
void waitdisk(void){
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40);
}

// Read a single sector at offset into dst.
void readsect(void *dst, uint offset) {
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
   ..
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

Lowering CPU overheads with interrupts

- Device sends an interrupt that it is ready
- CPU runs another process in the meantime
- Better CPU utilisation
- Not a good idea if device is fast.
    - If first poll finds that the device is ready, unnecessary overhead of switching processes





More efficient data movement

Direct Memory Access (DMA)

CPU	1	1	1	2	2	c	c	c	1
Disk				1	1				

CPU	1	1	1	2	2	2	2	2	1
DMA						c	c	c	
Disk				1	1				

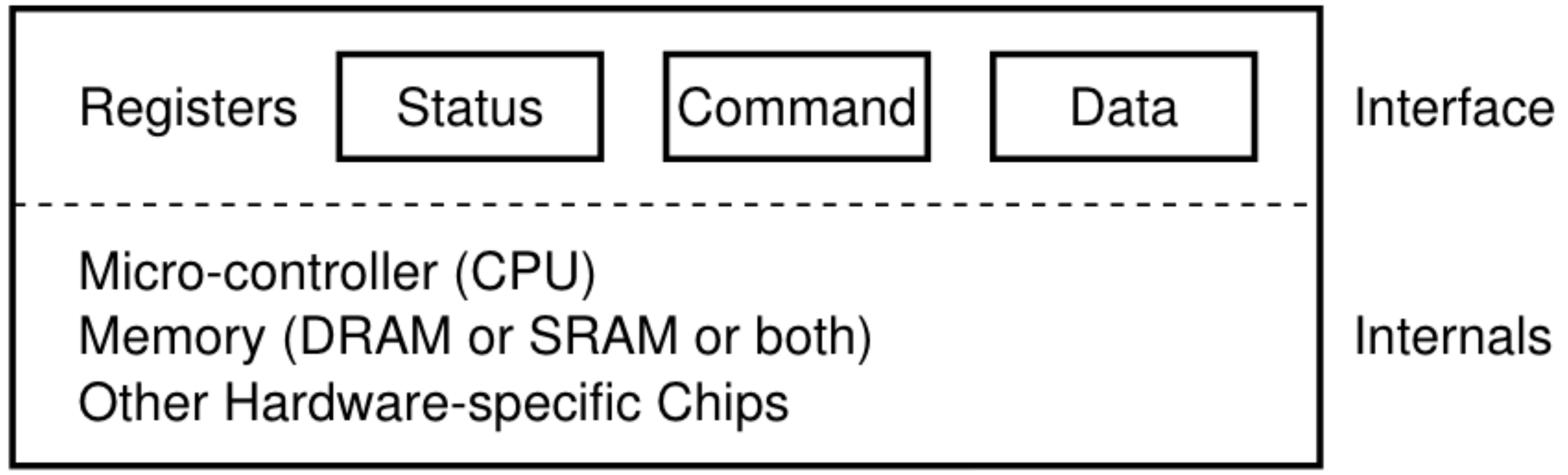


Figure 36.3: **A Canonical Device**

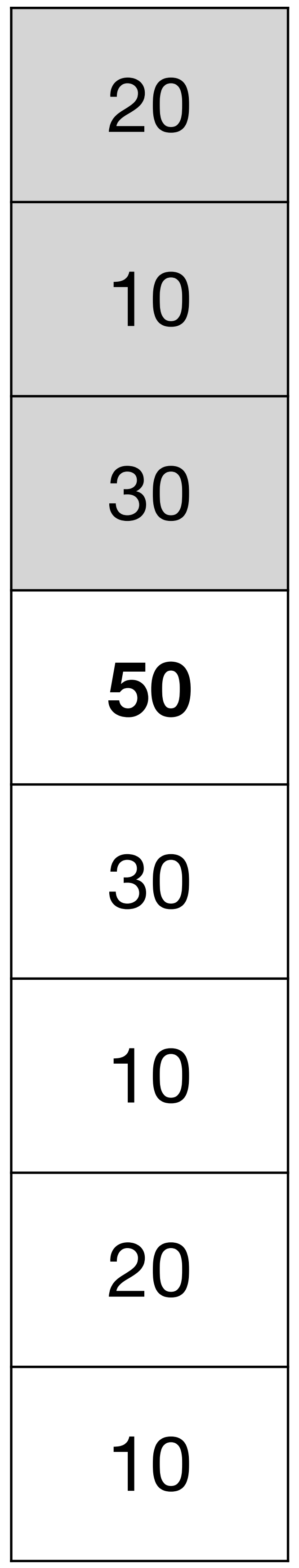
bootmain.c

```
void waitdisk(void){
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40);
}

// Read a single sector at offset into dst.
void readsect(void *dst, uint offset) {
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
   ..
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

Interrupt controllers, interrupt handling

xv6 Ch. 3 “Code: interrupts”

Calculator analogy



- 2 0 = (move pointer to 10)
- + 1 0 = (move pointer to 30)
- + 3 0 = (move pointer to 50)

Interrupt ←

Give me the calculator!

- 3*2 = 6

End of Interrupt ←

Ok, you can have it back

- + 5 0 = (move pointer to 30)
- + 3 0 = (move pointer to 10)
- + 1 0 = (move pointer to 20)
- + 2 0 = (move pointer to 10)

Programmable interrupt controllers (PIC)

Example: Intel 8259A

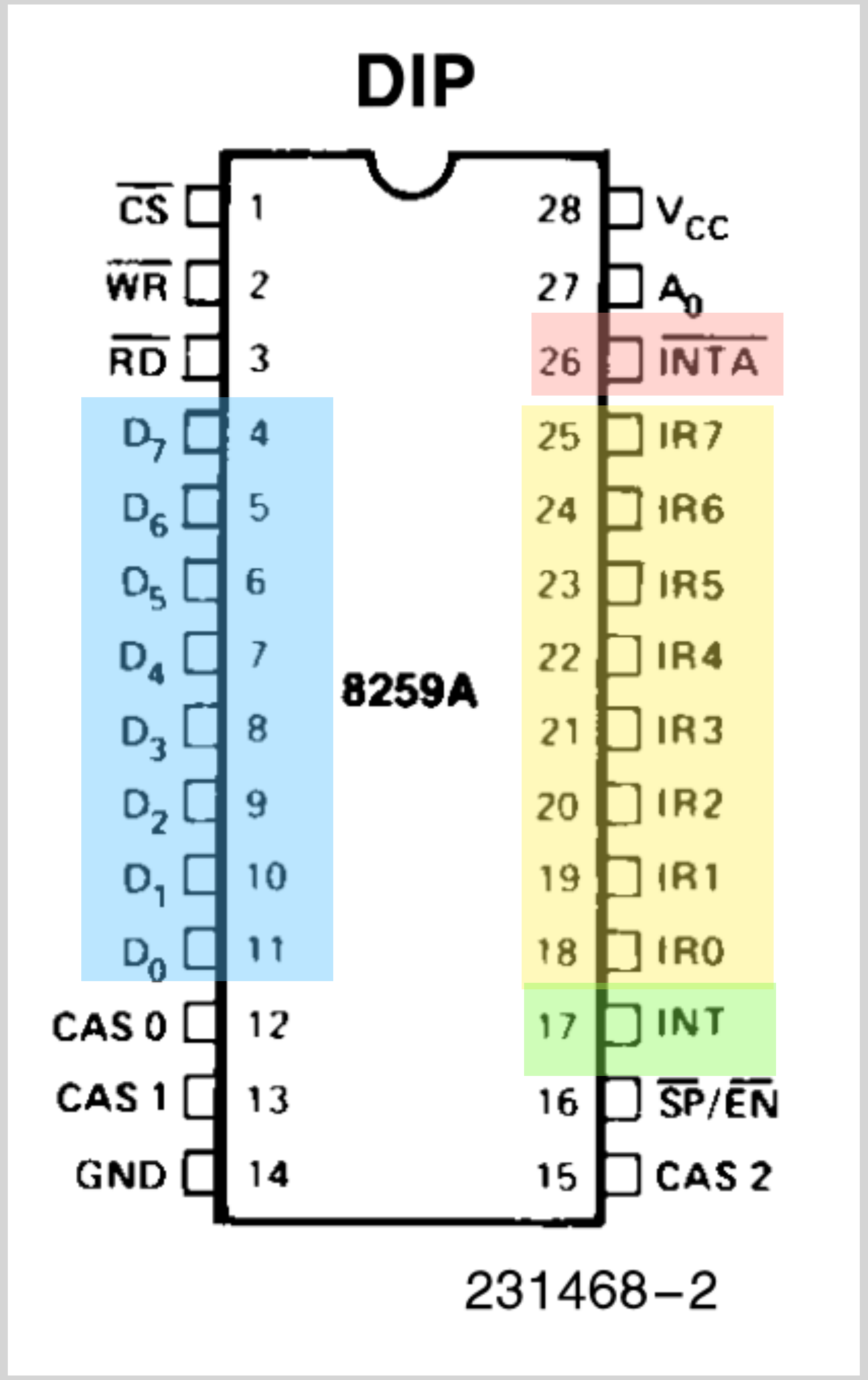
- Devices connect to IR0-IR7 pins.
  Device enables its pin to raise interrupt

- INT pin connects to CPU.

- PIC sends an 8-bit "interrupt vector" to CPU via D0-D7 pins

- CPU acknowledges that it is now working on interrupt on INTA pin

- CPU acknowledges "end-of-interrupt" on INTA pin



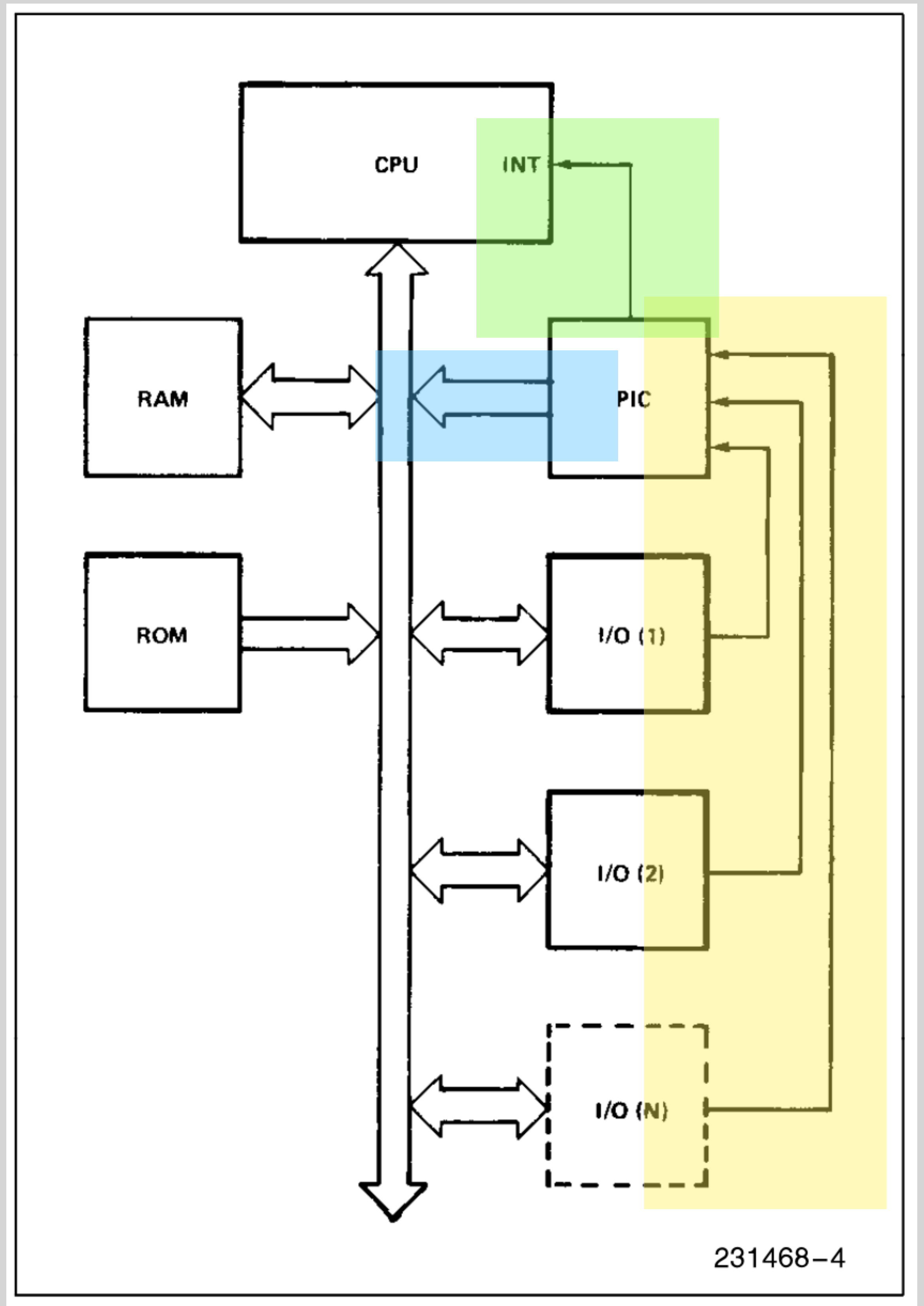
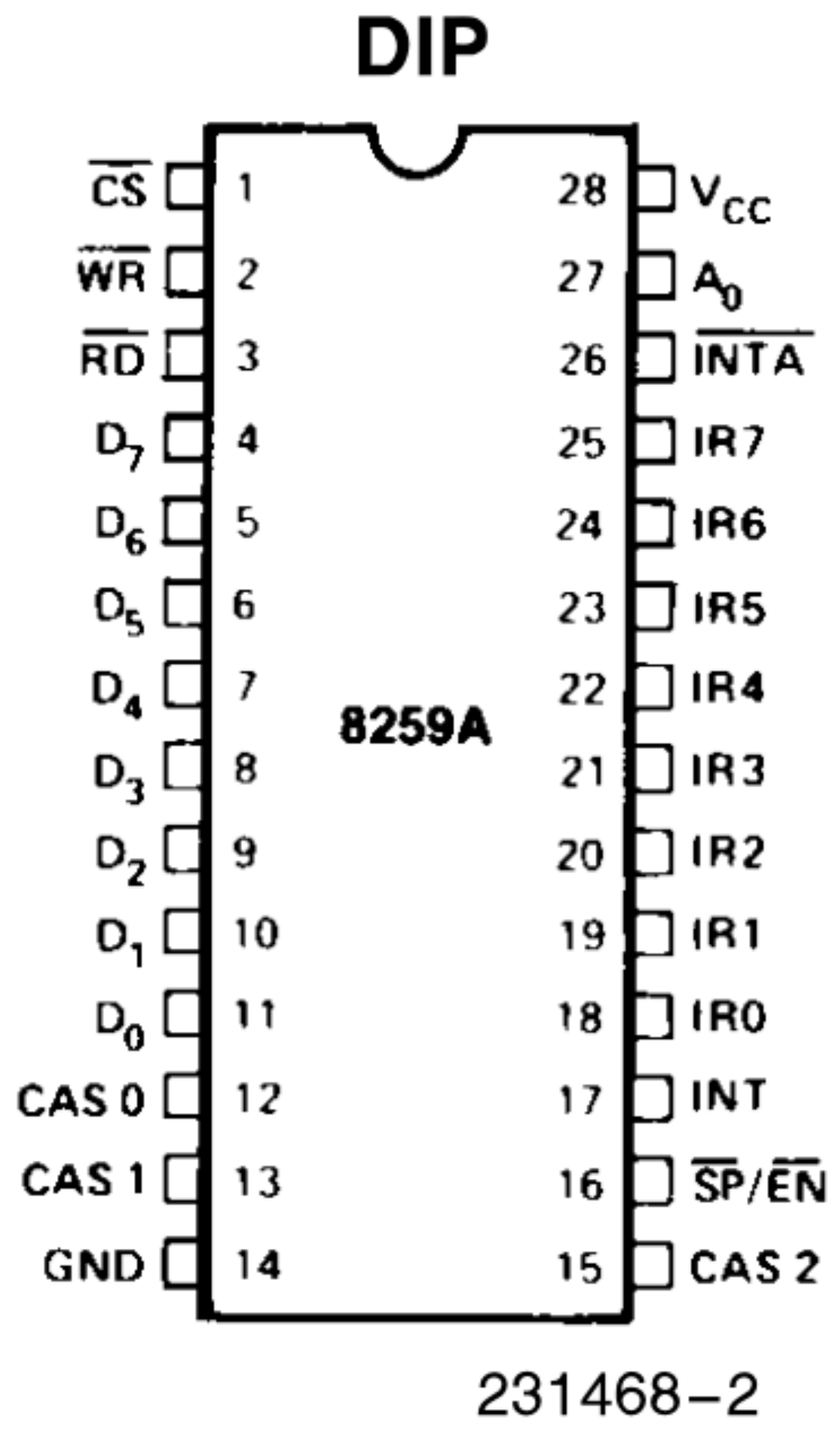


Figure 3b. Interrupt Method

PIC Problems

- No support for multiple CPUs
  - Route disk interrupts to CPU-0, keyboard interrupts to CPU-1
  - Different CPUs are working on different interrupts



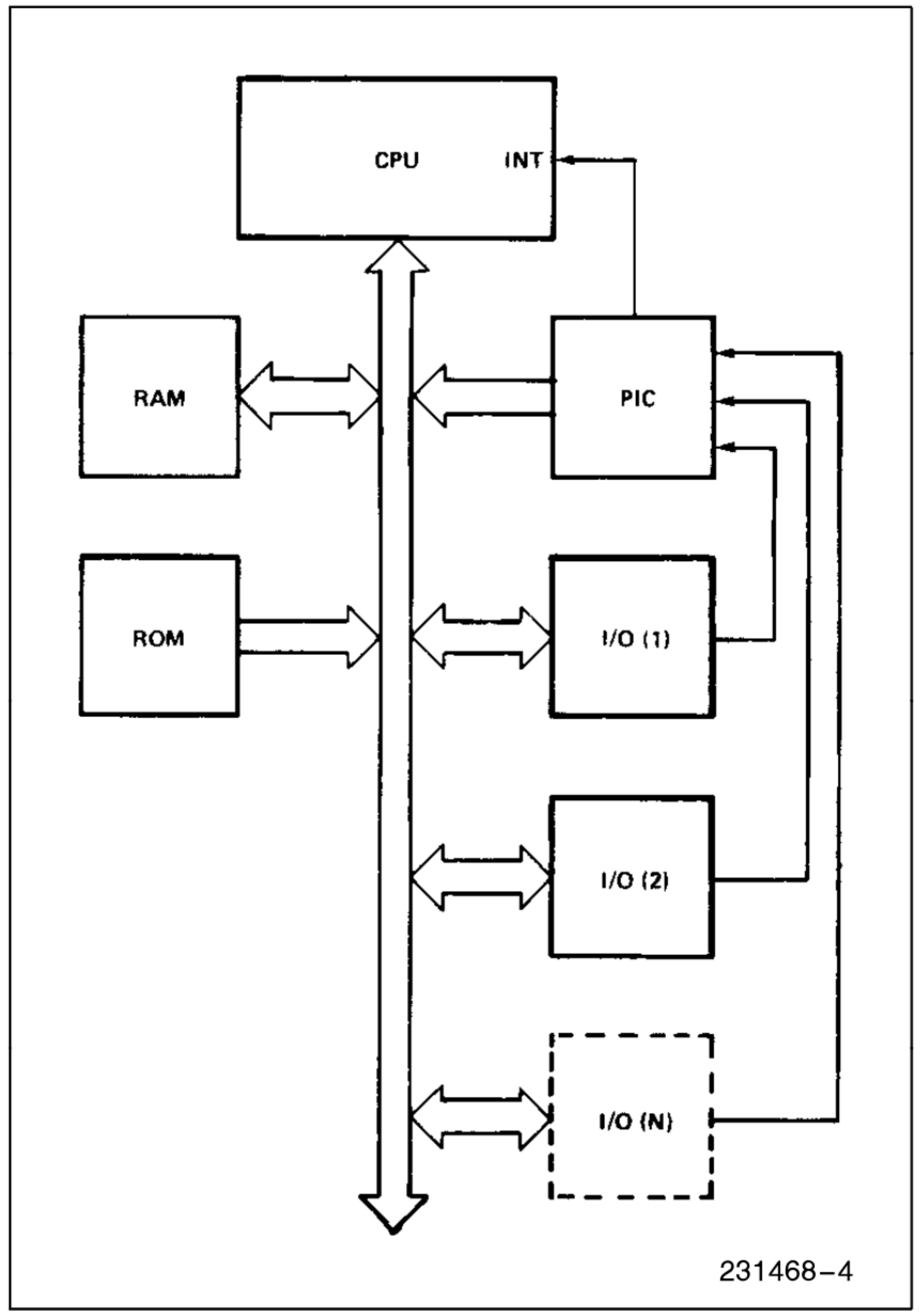
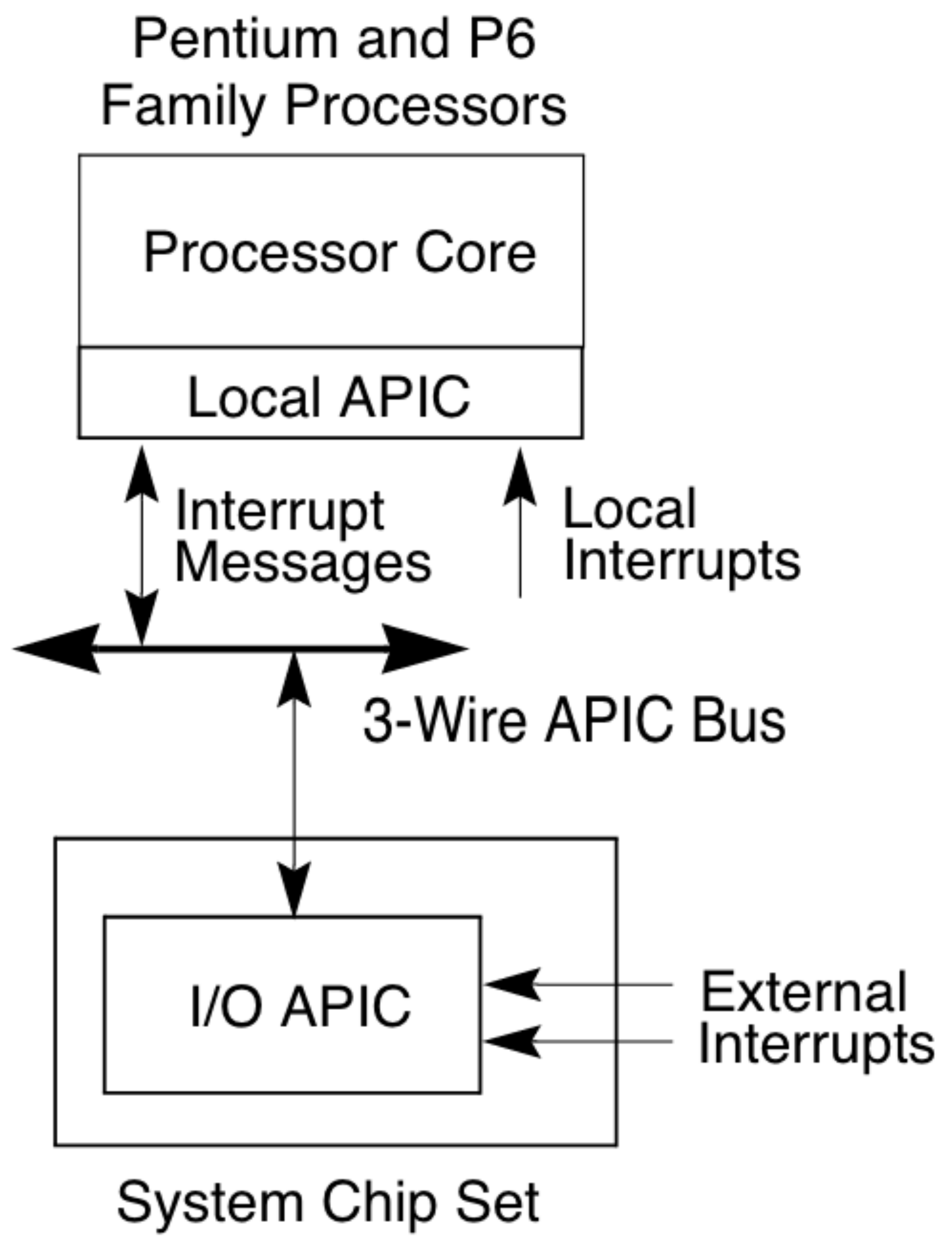


Figure 3b. Interrupt Method

Advanced programmable interrupt controllers (APIC)

- Each CPU can have local APICs for handling *local interrupts* like timer, thermal sensor, etc.
- A separate IO APIC receives external interrupts like keyboard, mouse, disk, etc and forwards it to a particular CPU
  - Example: Route keyboard interrupts to CPU-0, disk interrupts to CPU-1



p3-pic

Code walkthrough

- main.c calls mpinit, lapicinit, picinit, ioapicinit
- mpinit initialises lapic and ioapic addresses
- lapicinit enables timer interrupt at every 10ms. lapicw is just writing to memory location (MMIO)
- picinit just disables PIC using outb instructions (PMIO)
- ioapicinit initialises IO APIC with MMIO
- Bootloader had disabled interrupt with cli. We will not receive interrupts yet.

LAPIC/IOAPIC are MMIO

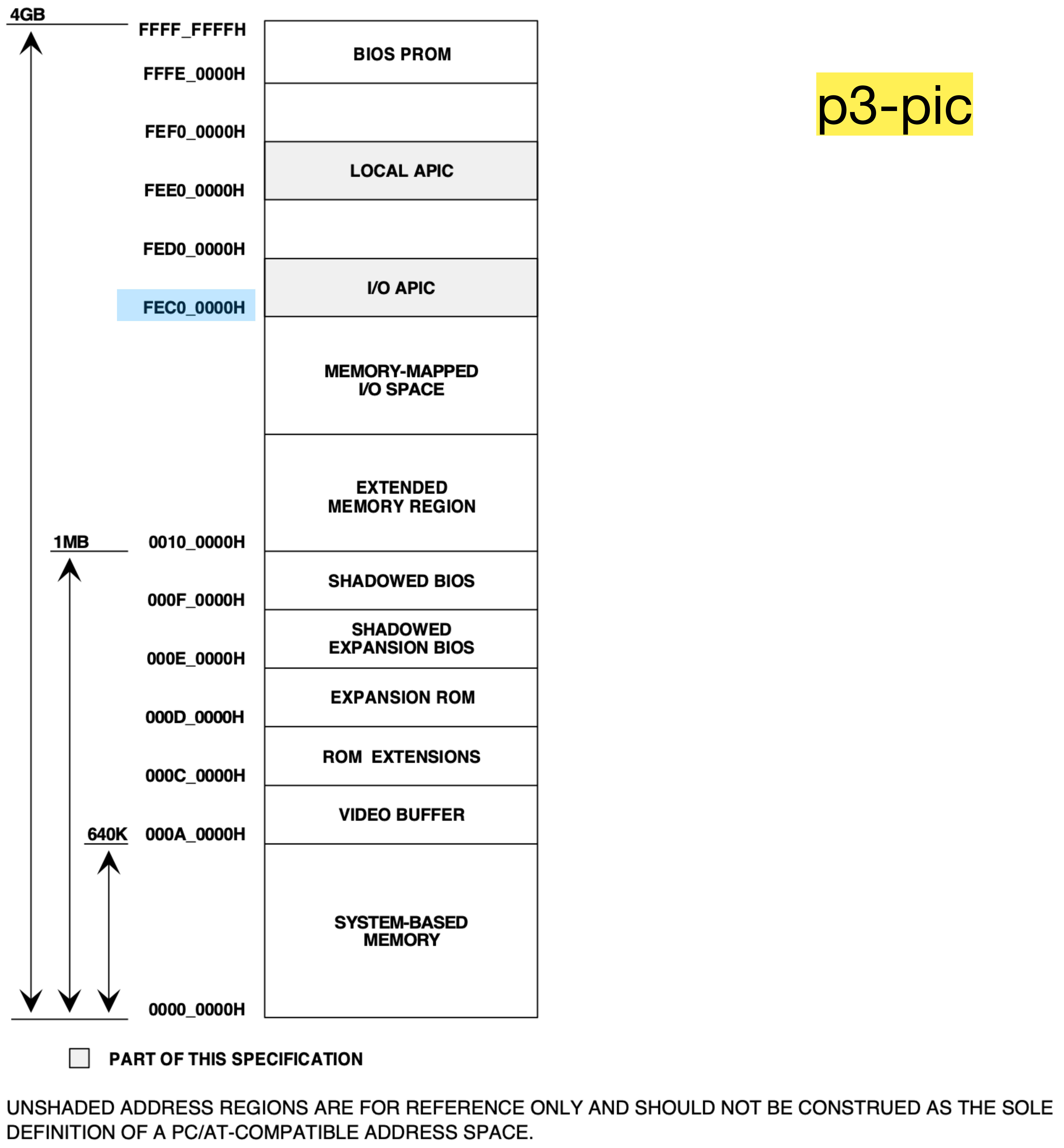
p3-pic

xv6 initializes these regions as volatile.

LAPIC base address can move within the region. It has to be *found* at bootup

```
volatile uint *lapic;


volatile struct ioapic *ioapic;
#define IOAPIC  0xFEC00000
ioapic = (volatile struct ioapic*)IOAPIC;
```

From Intel Multiprocessor Specification



UNSHADED ADDRESS REGIONS ARE FOR REFERENCE ONLY AND SHOULD NOT BE CONSTRUED AS THE SOLE DEFINITION OF A PC/AT-COMPATIBLE ADDRESS SPACE.

Figure 3-1. System Memory Address Map

Discovering Multiprocessor configuration table

- main() calls mpinit()
- mpinit() scans different memory regions to search for the “MP floating structure”

```
if(memcmp(p, "_MP_", 4) == 0 …)
  return (struct mp*)p;
…
conf = (struct mpconf*) (uint) mp->physaddr;
…
conf = mpconfig(&mp)
```

From Intel Multiprocessor Specification

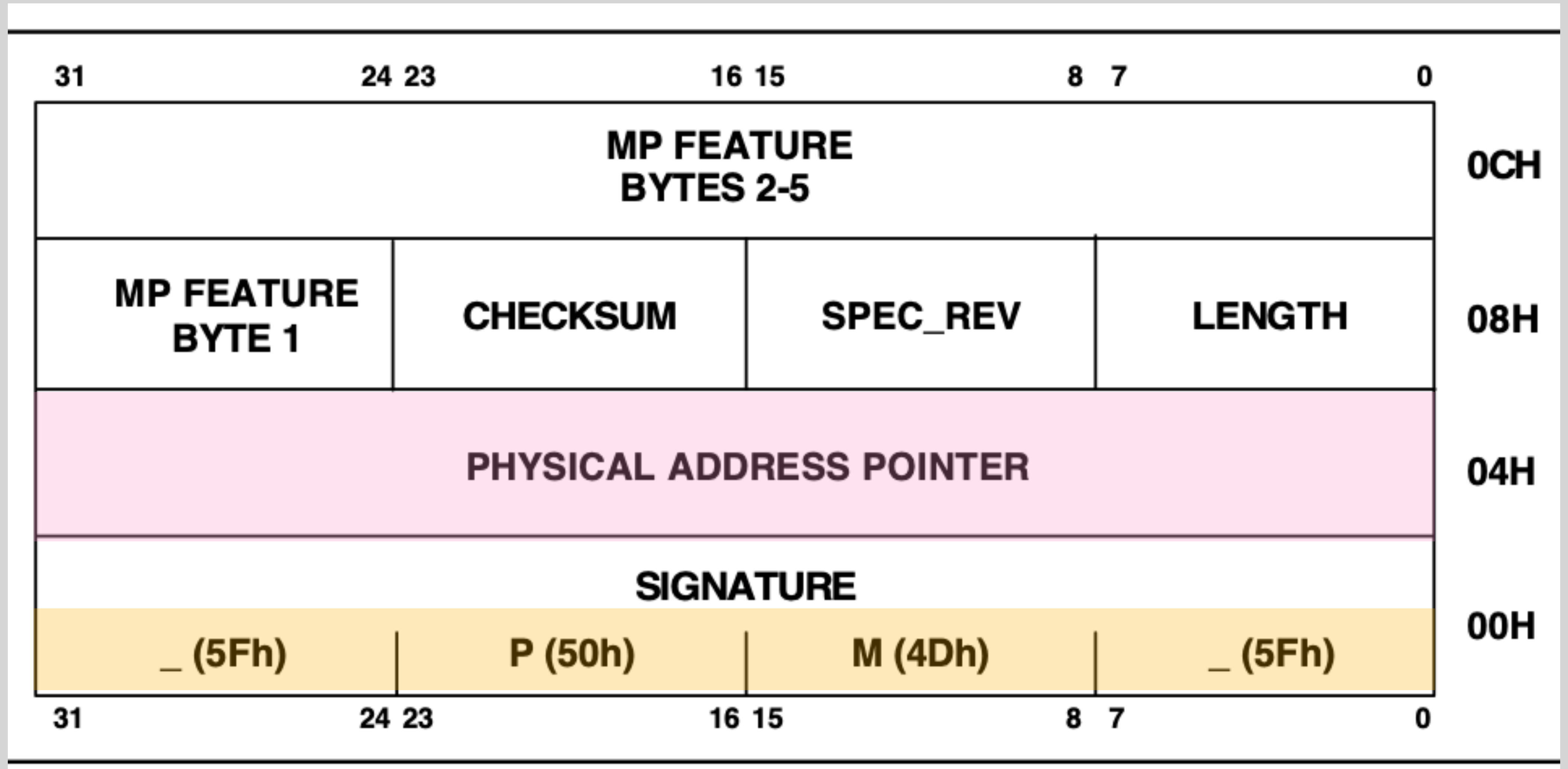


Figure 4-2. MP Floating Pointer Structure

Discovering processors

- Header mentions how many entries are there
- Entries in Multiprocessor Configuration table describes each processor, IOAPIC

From Intel Multiprocessor Specification

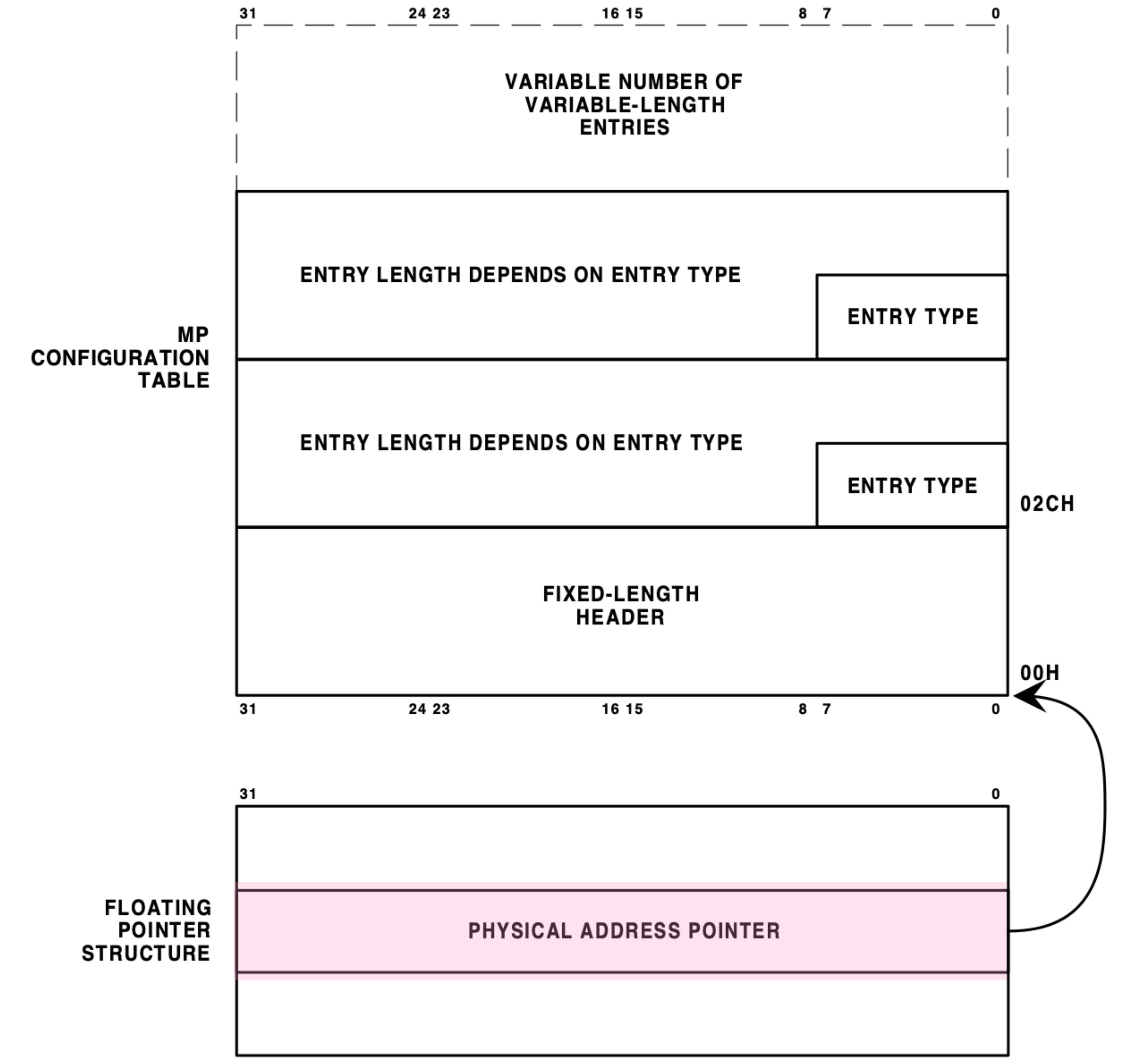

31
24 23
16 15
8 7
0
VARIABLE NUMBER OF VARIABLE-LENGTH ENTRIES
ENTRY LENGTH DEPENDS ON ENTRY TYPE
ENTRY TYPE
ENTRY LENGTH DEPENDS ON ENTRY TYPE
ENTRY TYPE
02CH
MP CONFIGURATION TABLE
FIXED-LENGTH HEADER
00H
FLOATING POINTER STRUCTURE
PHYSICAL ADDRESS POINTER

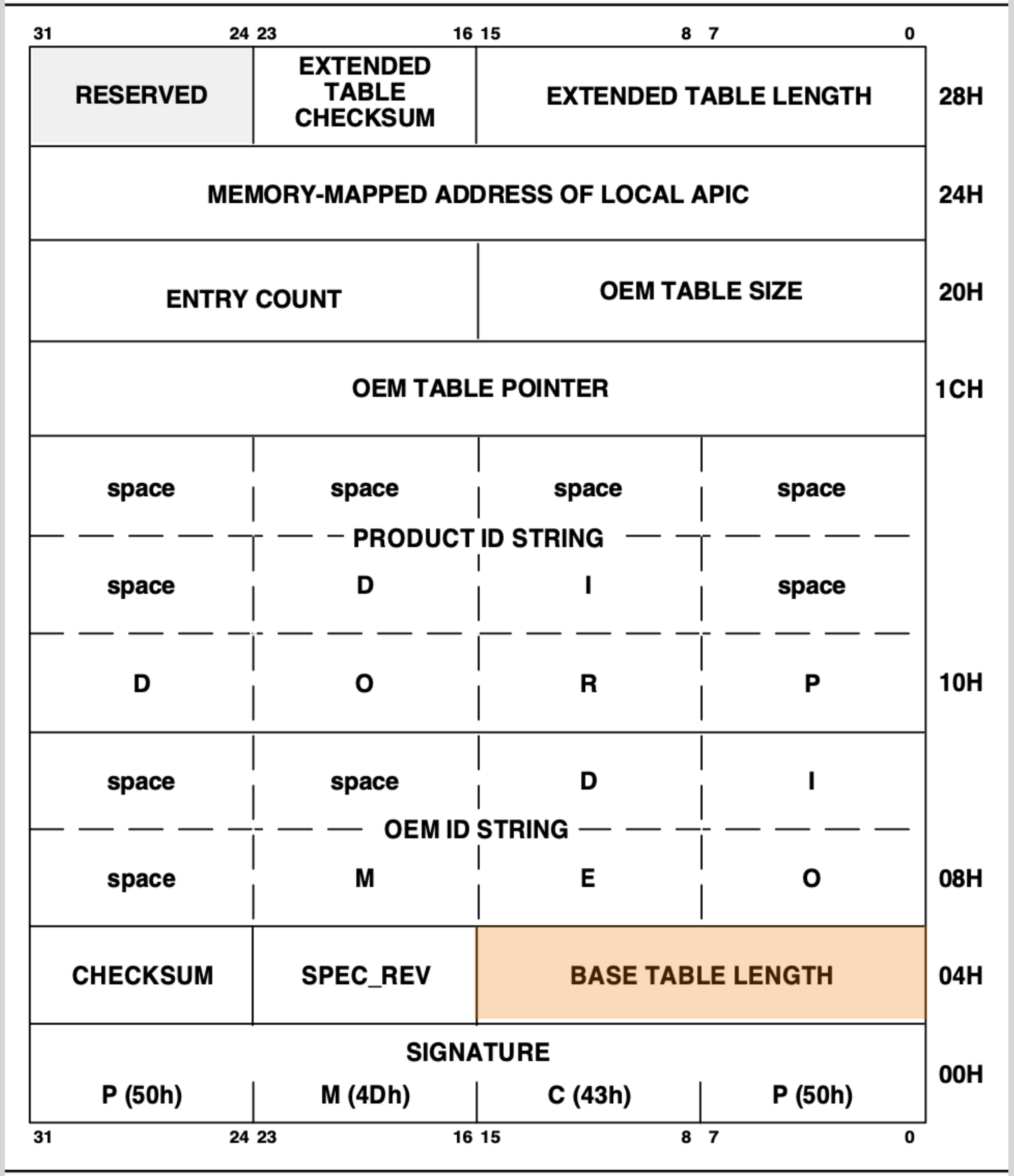
Figure 4-1. MP Configuration Data Structures

How many entries?

- mpinit() iterates over MP configuration table entries

```
for(p=(uchar*)(conf+1),
      e=(uchar*)conf+conf->length; p<e; ){
```

From Intel Multiprocessor Specification



31 24	23 16	15 8	7 0	
RESERVED	EXTENDED TABLE CHECKSUM	EXTENDED TABLE LENGTH		28H
MEMORY-MAPPED ADDRESS OF LOCAL APIC				24H
ENTRY COUNT		OEM TABLE SIZE		20H
OEM TABLE POINTER				1CH
space	space	space	space	
PRODUCT ID STRING				
space	D	I	space	
D	O	R	P	10H
space	space	D	I	
OEM ID STRING				
space	M	E	O	08H
CHECKSUM	SPEC_REV	BASE TABLE LENGTH		04H
SIGNATURE				
P (50h)	M (4Dh)	C (43h)	P (50h)	00H
31 24	23 16	15 8	7 0	

Figure 4-3. MP Configuration Table Header

Discovering processors

```
#define MPPROC    0x00  // One per processor

for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
switch(*p){
  case MPPROC:
    proc = (struct mpproc*)p;
    if(ncpu < NCPU) {
      // Each CPU has a unique APICID
      cpus[ncpu].apicid = proc->apicid;
      ncpu++;
    }
    p += sizeof(struct mpproc);
    continue;
```

From Intel Multiprocessor Specification

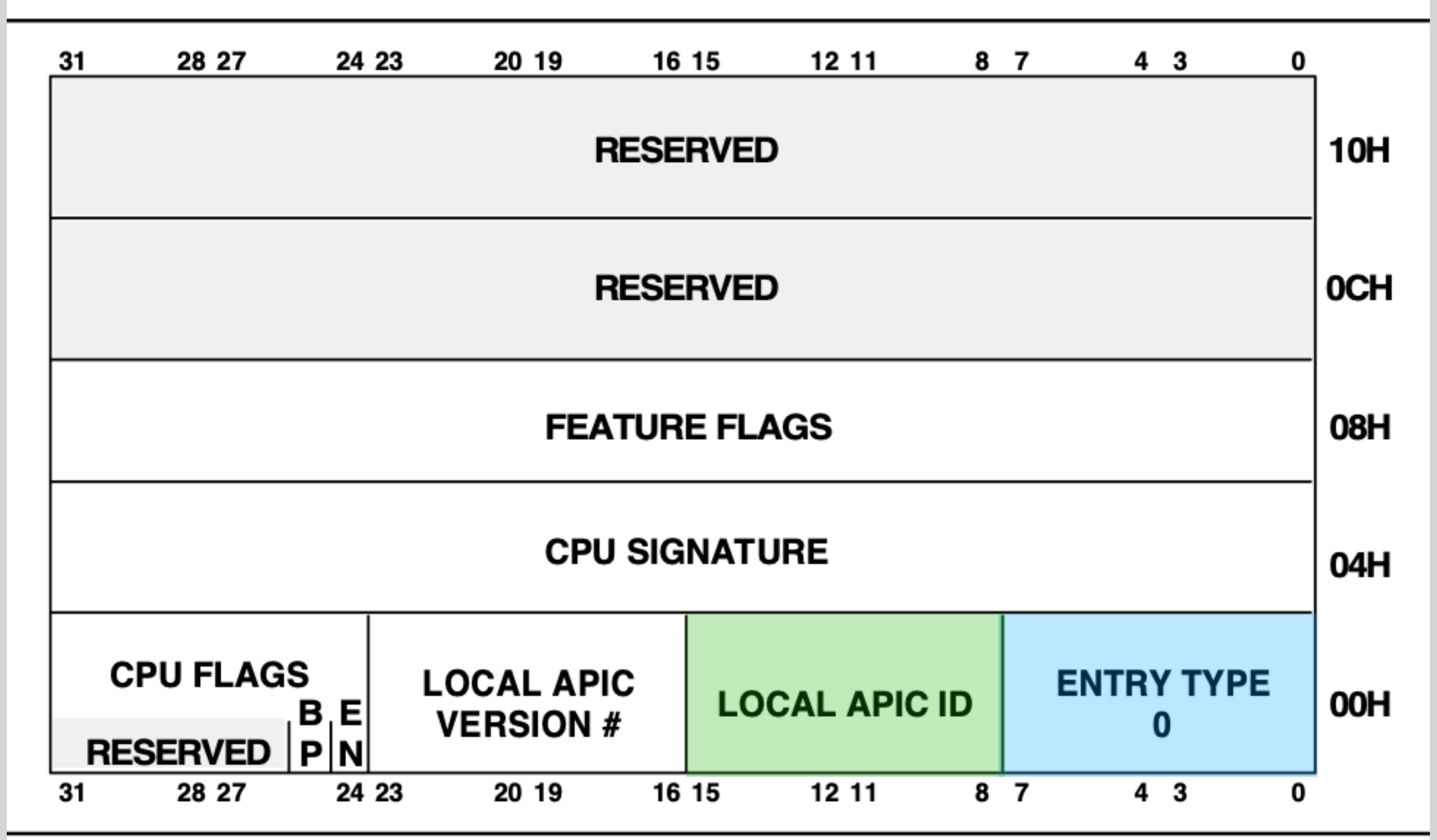


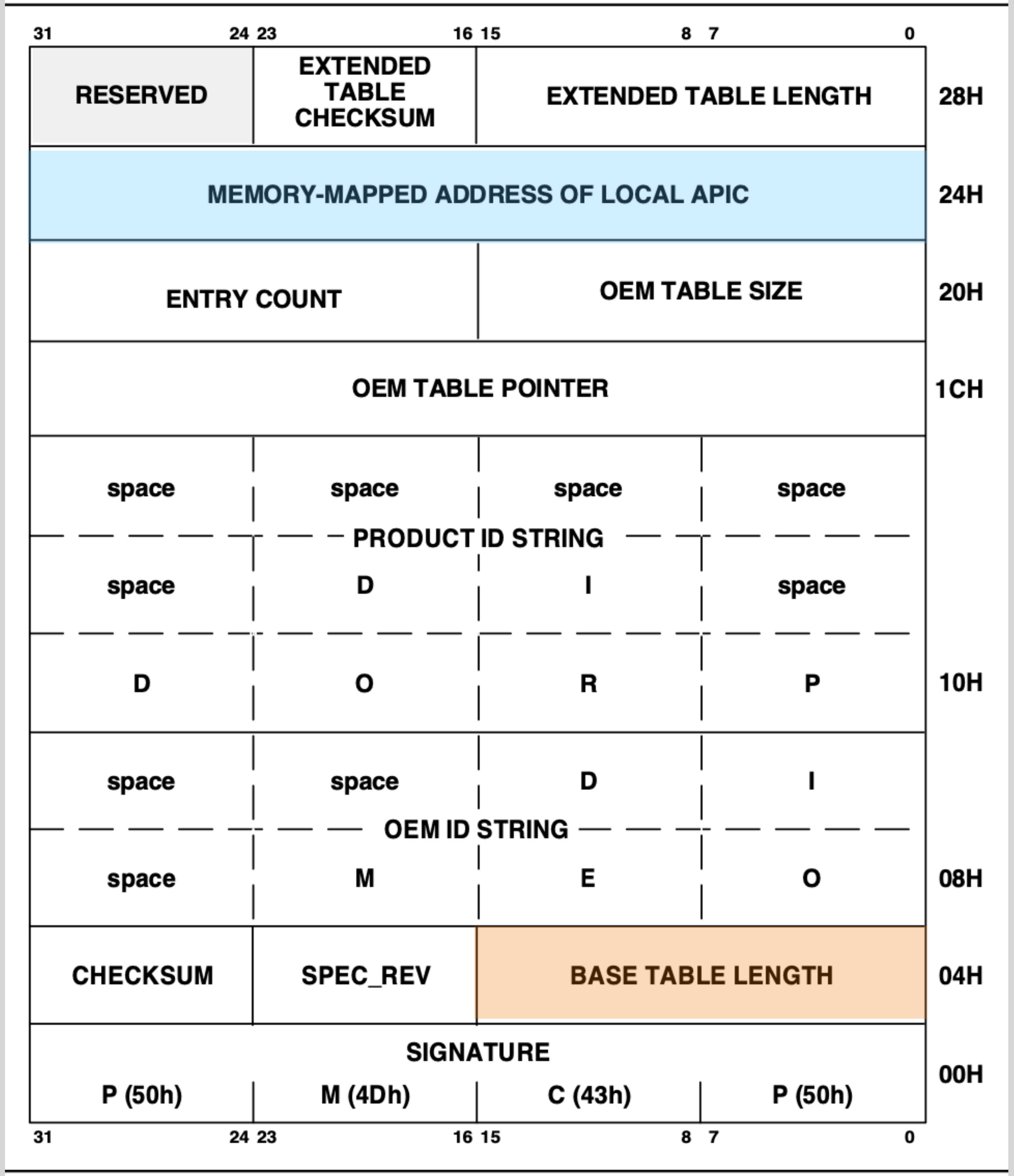
Figure 4-4. Processor Entry

Where is LAPIC mapped?

- mpinit() initializes lapic address from MP configuration header

```
lapic = (uint*)conf->lapicaddr;
```

From Intel Multiprocessor Specification



31–24	23–16	15–8	7–0	
RESERVED	EXTENDED TABLE CHECKSUM	EXTENDED TABLE LENGTH		28H
MEMORY-MAPPED ADDRESS OF LOCAL APIC				24H
ENTRY COUNT		OEM TABLE SIZE		20H
OEM TABLE POINTER				1CH
space	space	space	space	
space	D	I	space	
D	O	R	P	10H
space	space	D	I	
space	M	E	O	08H
CHECKSUM	SPEC_REV	BASE TABLE LENGTH		04H
SIGNATURE P (50h)	M (4Dh)	C (43h)	P (50h)	00H

31
24 23
16 15
8 7
0
PRODUCT ID STRING
OEM ID STRING

Figure 4-3. MP Configuration Table Header

Set periodic timers in LAPIC

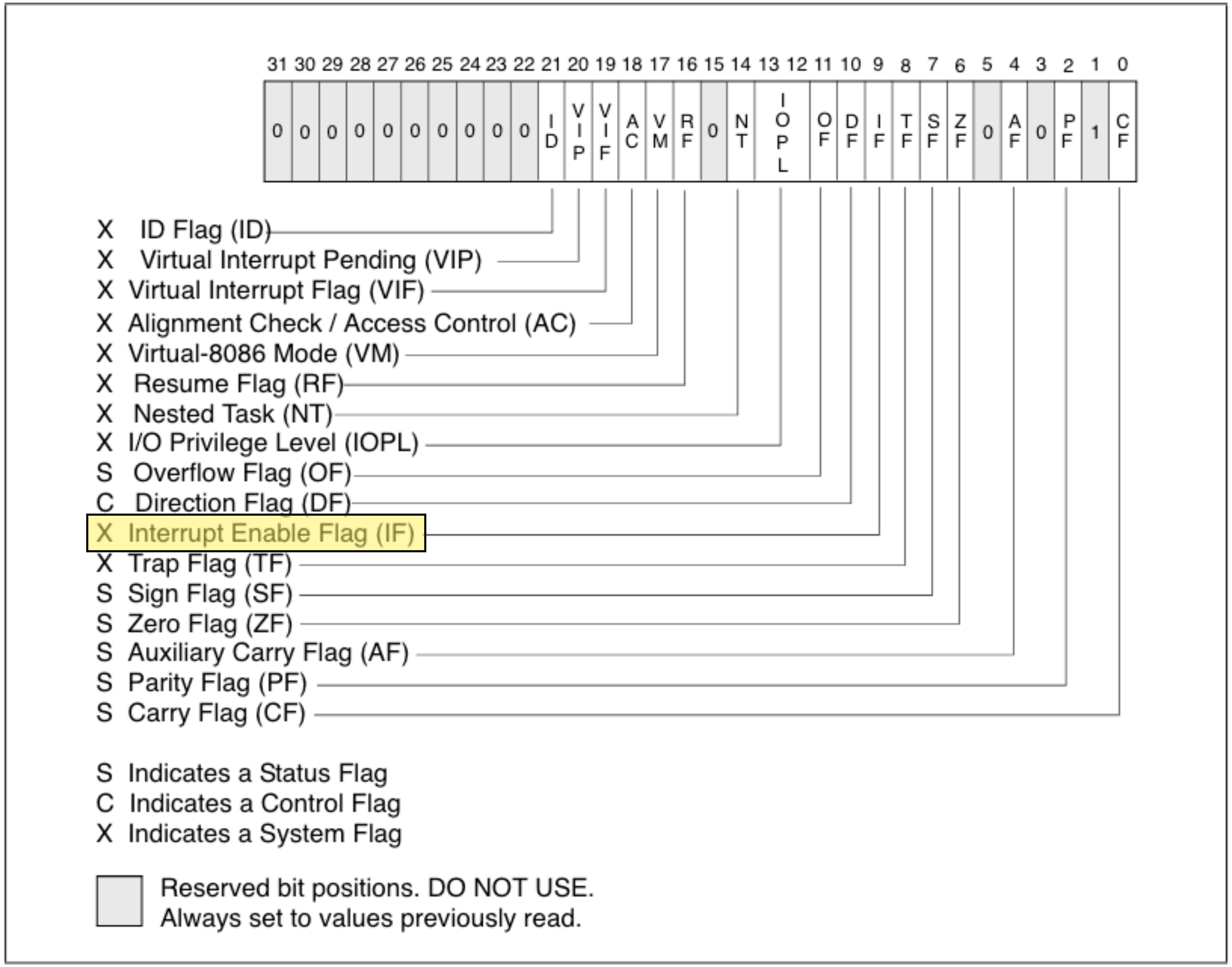
p3-pic

```
volatile uint *lapic;
static void lapicw(int index, int value){
  lapic[index] = value;
  lapic[ID];  // wait for write to finish, by reading
}
void lapicinit(void) {
  …
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 10000000);
}
```

LAPIC registers are described in Intel SDM Volume 3A, Table 10-1 in Section 10.4

Interrupt enable flag

- cli: Clear interrupt flag
    - PICs are not allowed to interrupt
- sti: Set interrupt flag

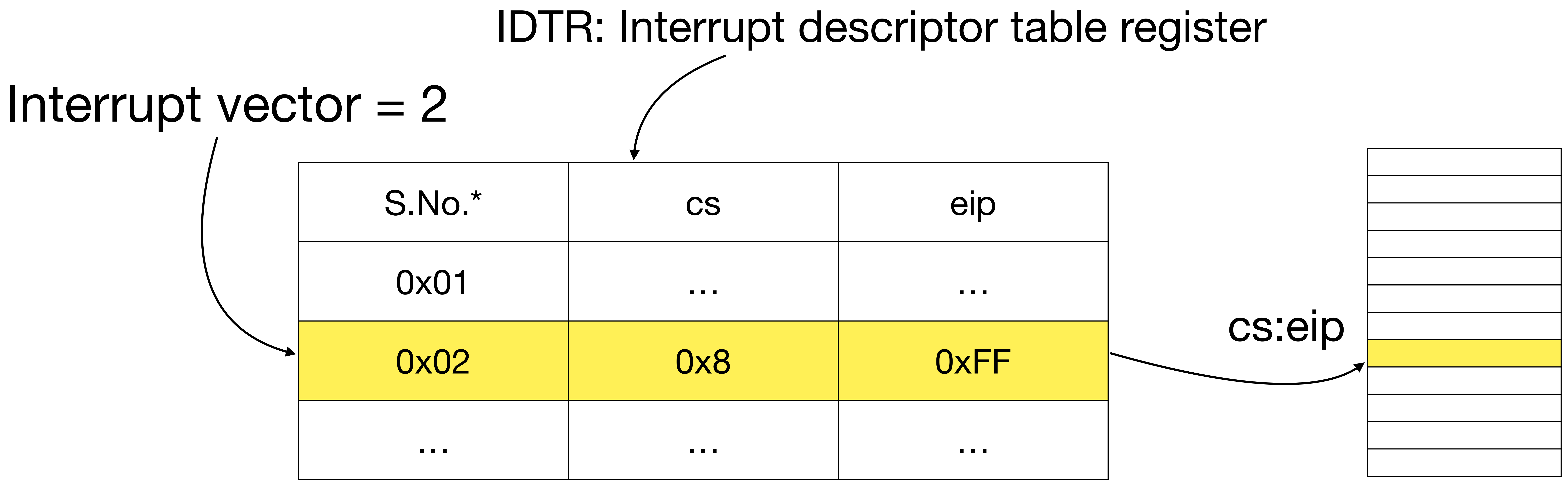


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE. Always set to values previously read.

Figure 3-8. EFLAGS Register

Interrupt handling in a nutshell

- OS sets up “interrupt descriptor table” (IDT)
- Points IDTR to IDT using LIDT instruction
- When interrupt occurs, jump %eip to interrupt handler, handle interrupt, tell LAPIC about end of interrupt, resume what we were doing



S.No.*	cs	eip
0x01	…	…
0x02	0x8	0xFF
…	…	…

Interrupt descriptor table

- Interrupt descriptor table register (IDTR) points to interrupt descriptor table in memory
- OS sets up IDT and initialises IDTR using LIDT instruction
- Interrupt descriptor table has one entry for each interrupt vector (upto 2^8=256)

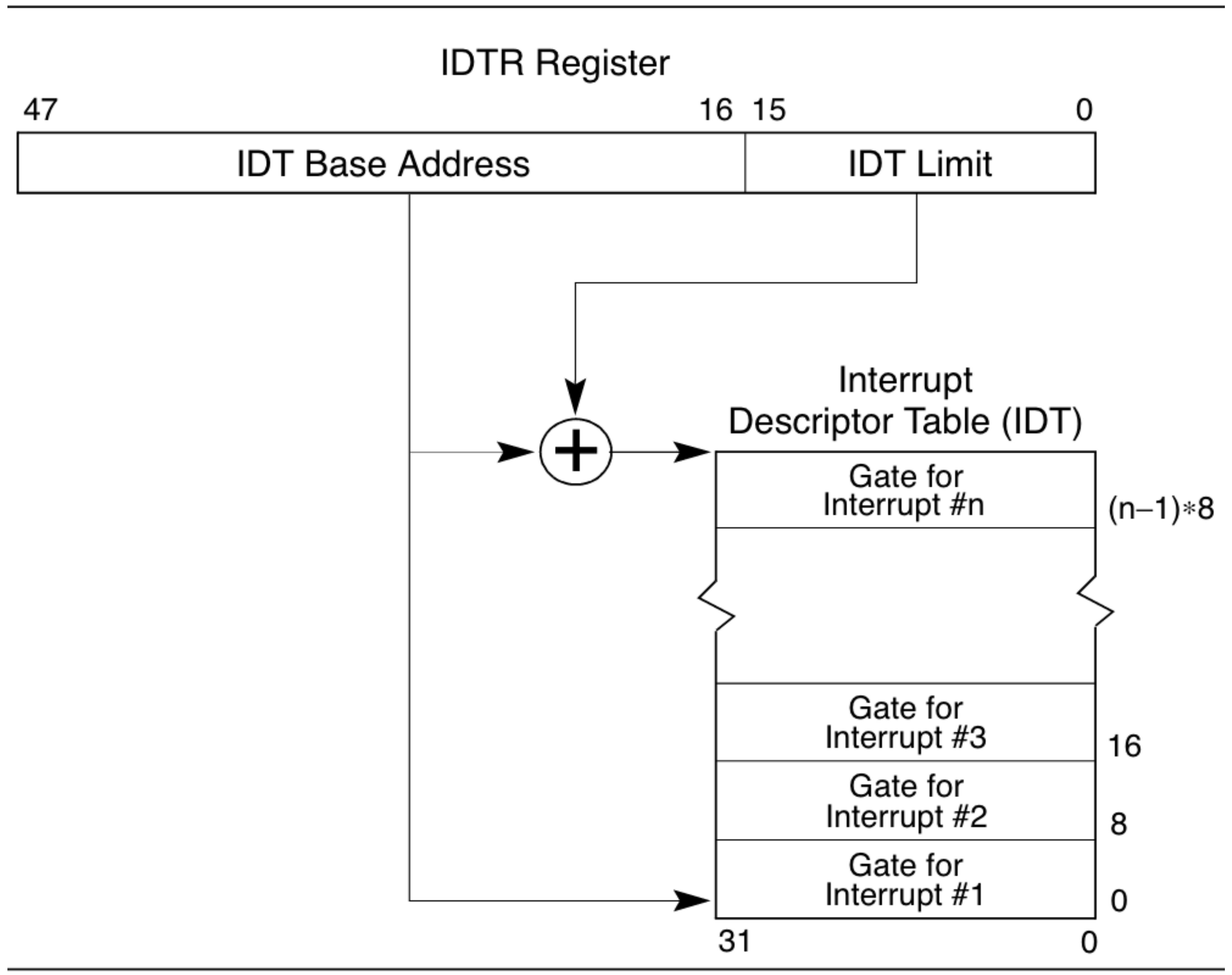
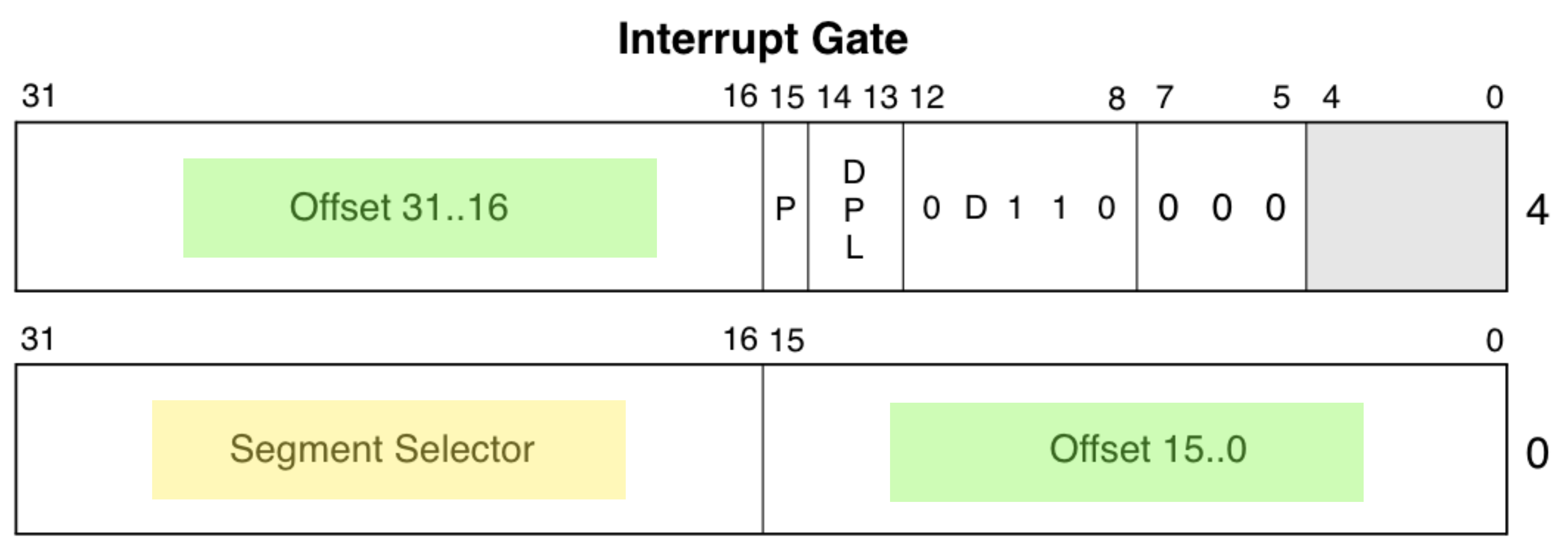


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor table (2)

- Each IDT entry is 64-bits. Contains code segment and eip
- When interrupt appears, hardware changes CS and EIP to the one pointed by IDT entry



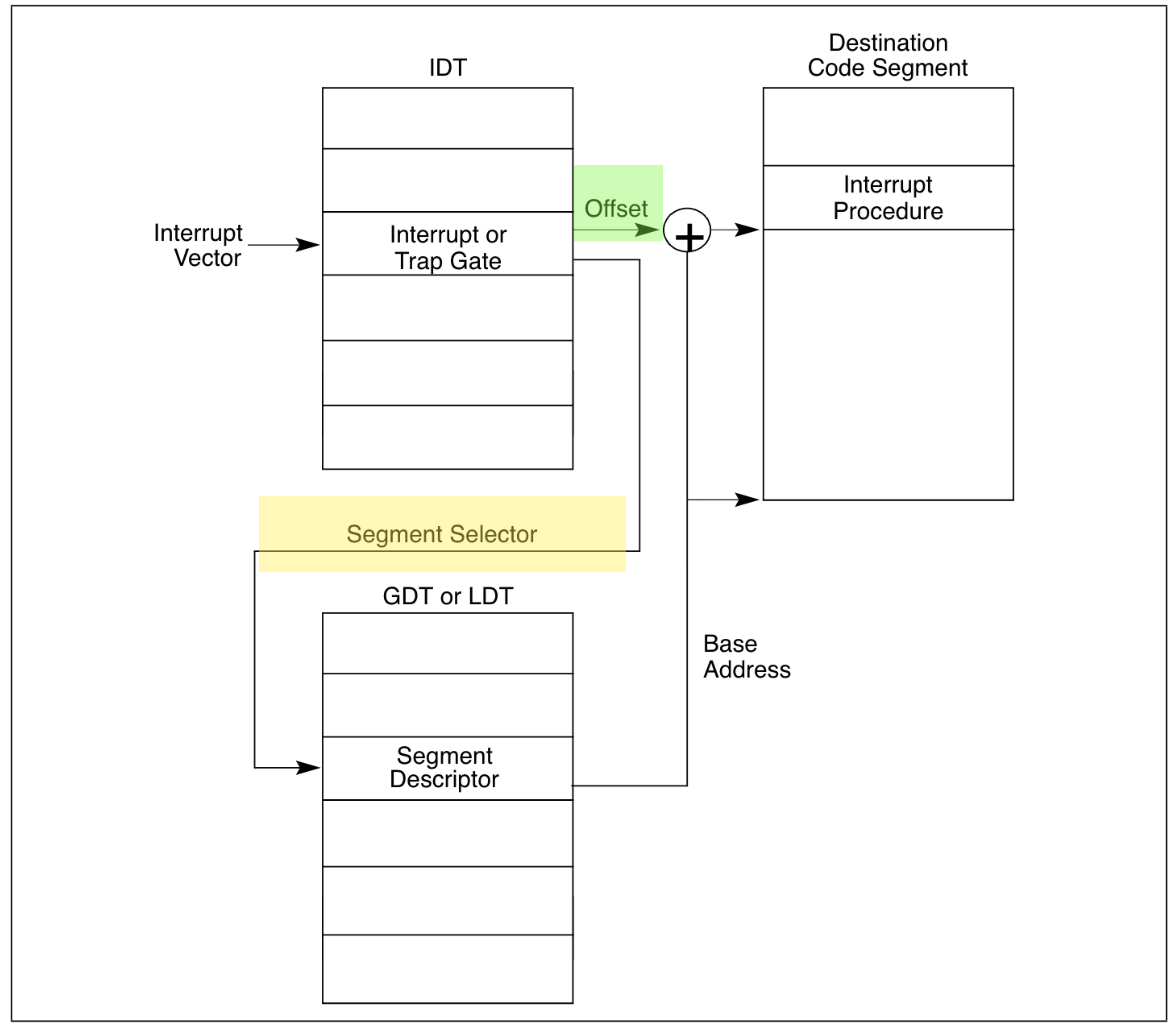
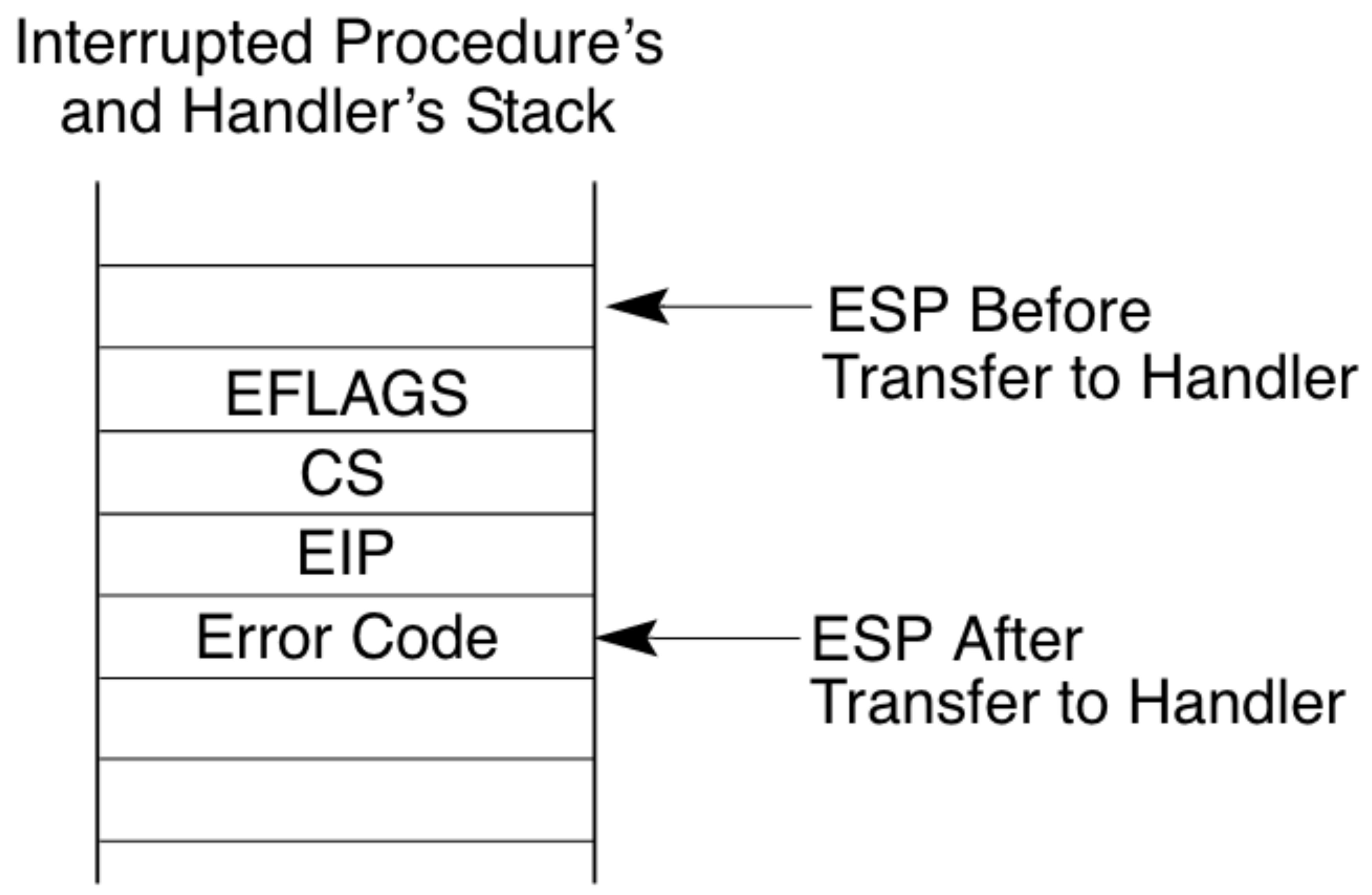


Figure 6-3. Interrupt Procedure Call

Interrupt handling

- On an interrupt, hardware pushes old EFLAGS, CS and EIP on the stack
- Jumps CS and EIP according to IDT
- IRET instruction (similar to RET instruction) restores CS, EIP, EFLAGS, ESP
- Interrupt handler may push more registers, like eax etc. on the stack.


Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler

Code walkthrough

p4-traps

- Setting up IDT:
  - vectors.pl creates 256 IDT entries. ‘i’th entry write ‘i’ on top of the stack and jumps to ‘alltraps’
  - main.c calls tvinit and idtinit to setup interrupt descriptor table to populate the 256 entries and point IDTR to IDT. It calls sti to receive interrupts.
- Handling interrupts
  - ‘alltraps’ in trapasm.S runs ‘pushal’ to save general purpose registers. Then it calls ‘trap’ with the trapframe.
  - ‘trap’ in ‘trapasm.S’ reads trapno saved by vectors.S to find out which interrupt occurred. It handles timer and spurious interrupts. It signals EOI to LAPIC when it is done with interrupt.
  - trapasm recovers registers with popal, backs up esp above err code and trap number, executes IRET to jump back to whatever OS was doing earlier

Setting up IDT

- vectors.S creates 256 interrupt handlers. All of them jump to alltraps

- vectors.S creates a vectors array containing the addresses of these 256 interrupt handlers

vectors.S

```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
 …

.data
.globl vectors
vectors:
  .long vector0
  .long vector1
```

Setting up IDT

- vectors.S creates a vectors array containing the addresses of these 256 interrupt handlers

```
$ vim kernel.sym
00100cb1 vector0
00100cba vector1
00102000 vectors
```

```
$ vim kernel.asm
00100cb1 <vector0>:
100cb1: 6a 00             push $0x0
100cb3: 6a 00             push   $0x0
100cb5: e9 e6 fe ff ff   jmp    100ba0 <alltraps>
```

```
$ objdump -s --start-address=0x00102000 --stop-address=0x00102100 kernel
kernel:     file format elf32-i386

Contents of section .data:
 102000 b10c1000 ba0c1000 c30c1000 cc0c1000
```

Setting up IDT

- tvinit sets up interrupt descriptor table. In ith IDT entry,
    - set cs=SEG_KCODE<<3
    - eip = vectors[i]
- idtinit sets IDTR
- sti enables interrupts

trap.c

```
struct gatedesc idt[256];
// in vectors.S: array of 256 entry pointers
extern uint vectors[];
void tvinit(void) {
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
}
void idtinit(void) {
  lidt(idt, sizeof(idt));
}
```

main.c

```
int main(void) {
  …
  tvinit();          // trap vectors
  idtinit();         // load idt register
  sti();
```

Visualizing interrupt handling

eip →

main.c

```
for(;;)
  wfi();
```

trap.c

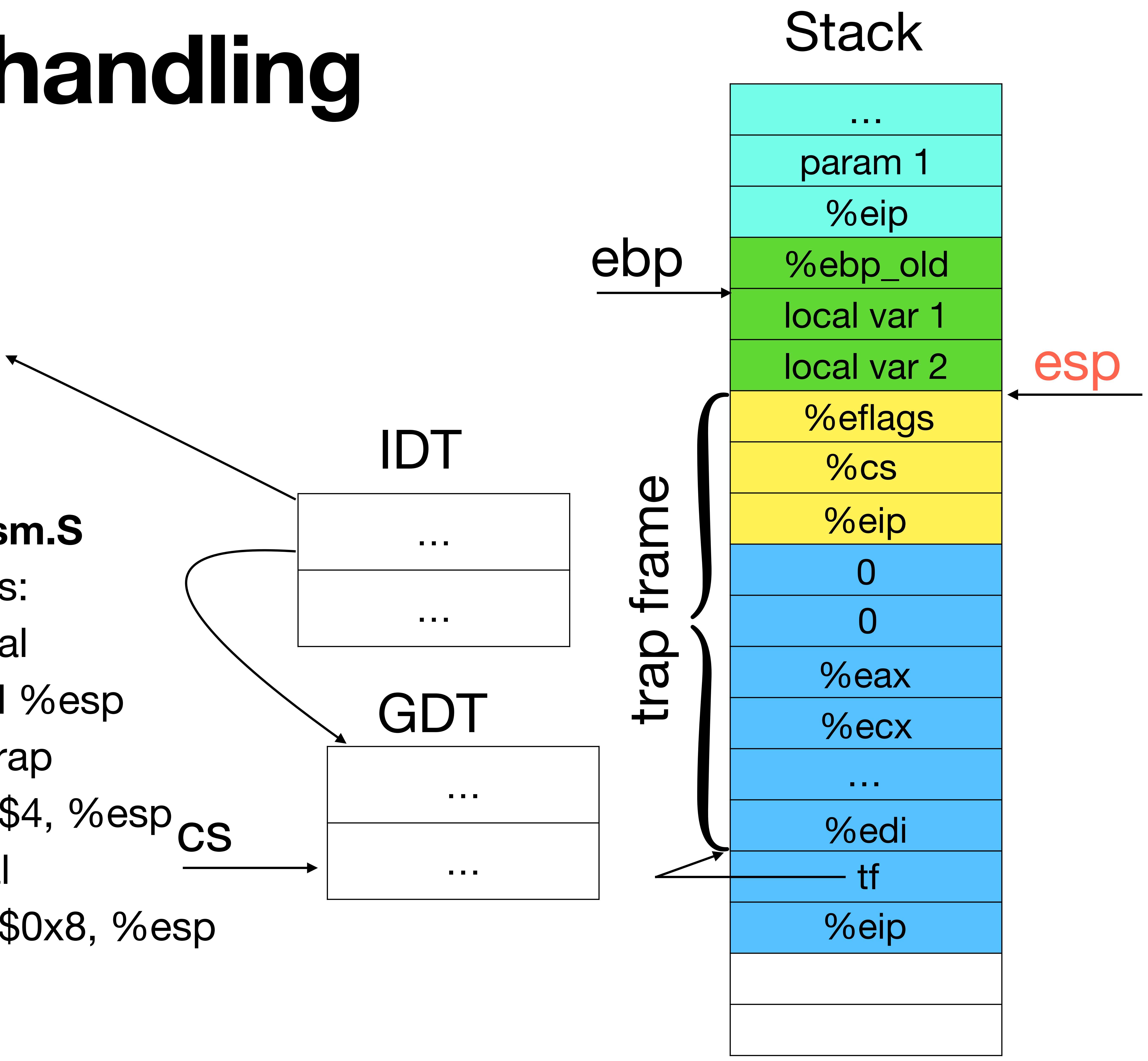
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

vectors.S

```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trapasm.S

```
alltraps:
  pushal
  pushl %esp
  call trap
  addl $4, %esp
  popal
  addl $0x8, %esp
  iret
```



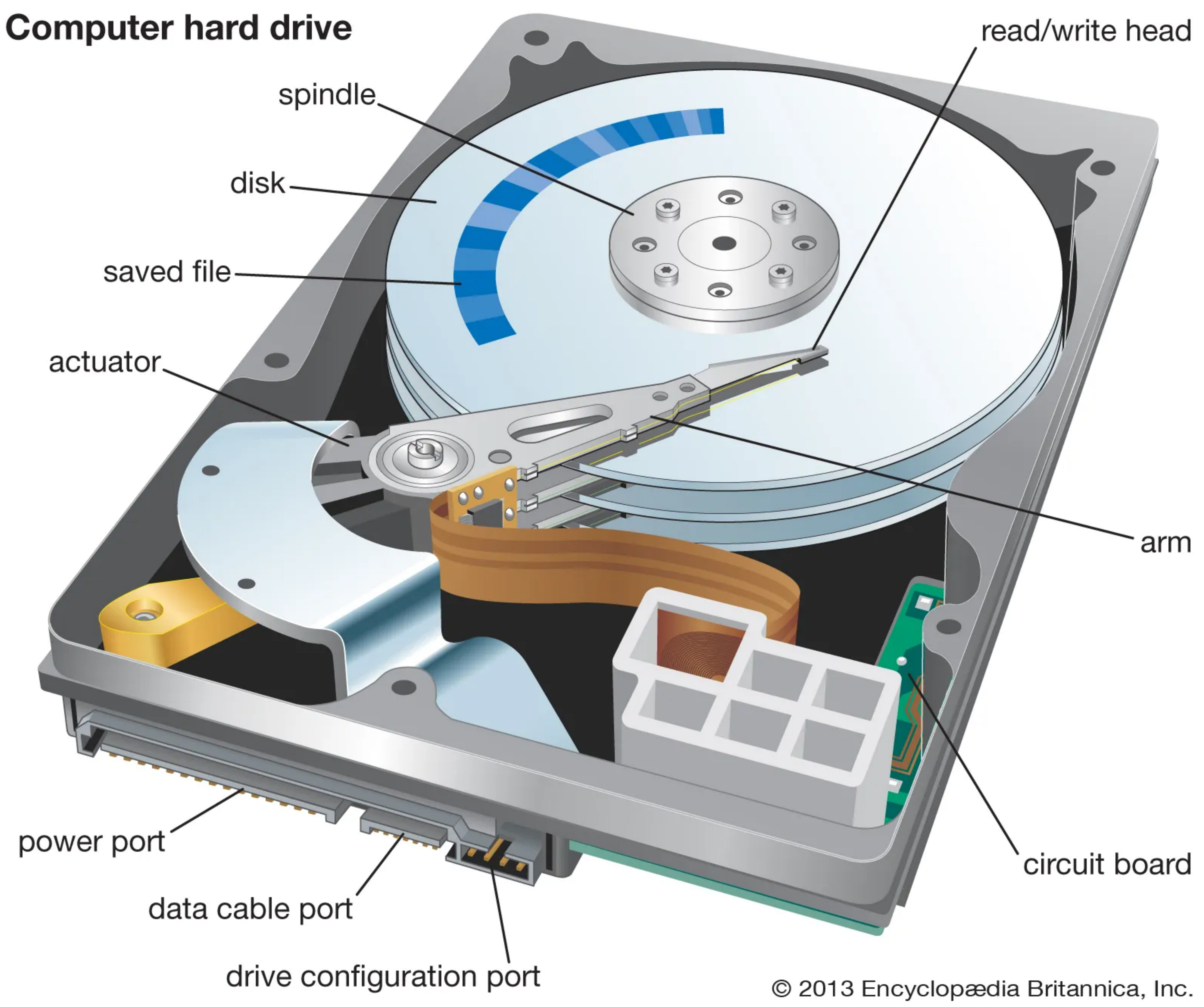
Hard disk drive

Ch. 37 OSTEP book

Understanding disk behaviour is important to write a performant file system

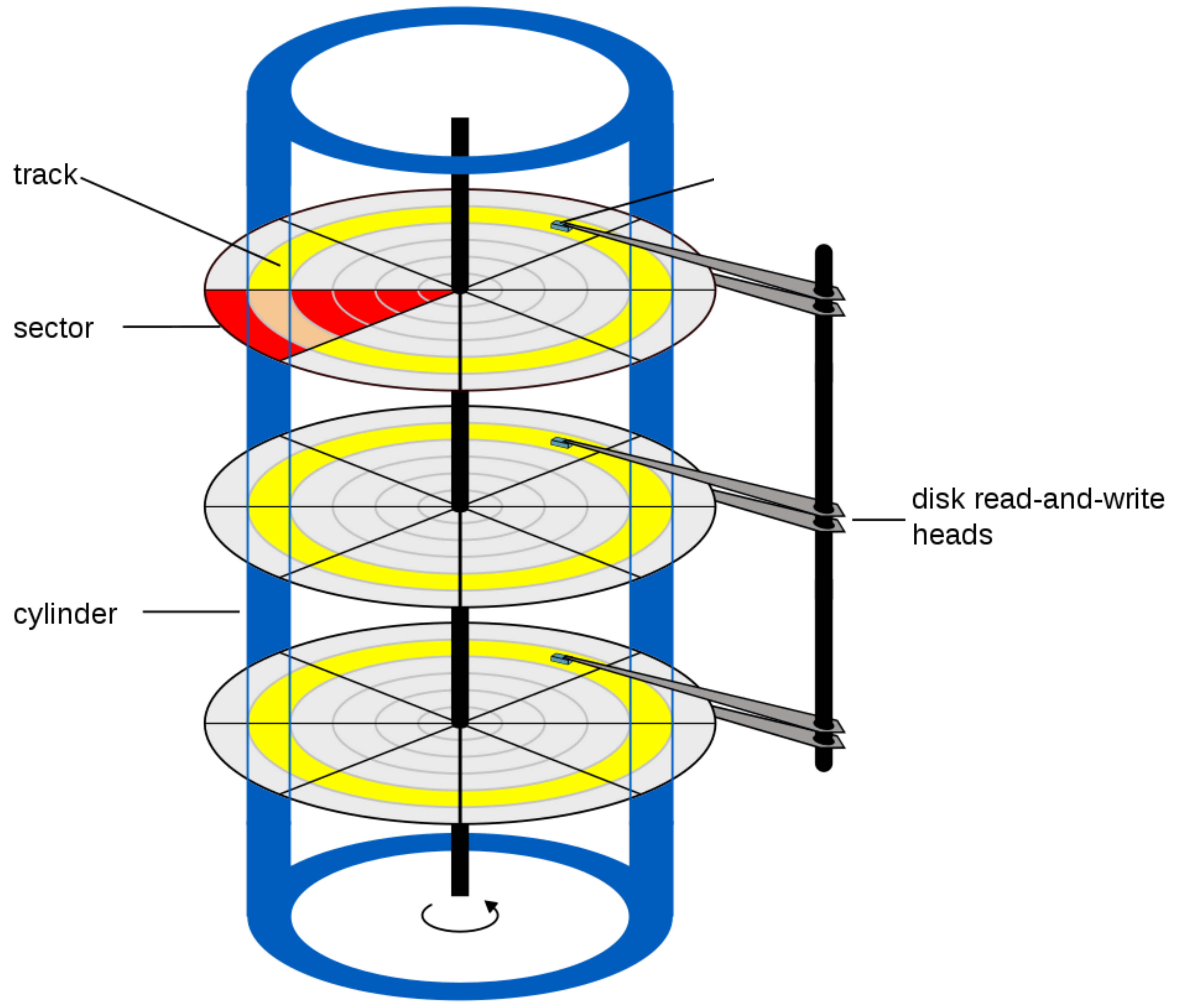
Disk geometry

- Many platters spinning on a spindle (~10,000 RPM)
- Each platter has two disk heads, one for each surface
- Disk heads are controlled by actuator
- One circle is called a track. Data is stored in sectors
- When the head is above a sector, it can read/write data



CPU-disk interface: Cylinder-head-sector (CHS) addressing

- C: cylinder number. 1024 cylinders.
- H: head number. 255 heads (tracks).
- S: sector number. 63 sectors per track.
- 512 bytes in each sector
- Example: read 40th cylinder's 26th sector using 7th head.



Example of reads

	Cheetah 15K.5
Capacity	300 GB
RPM	15,000
Average Seek	4 ms
Max Transfer	125 MB/s
Platters	4
Cache	16 MB

- Seek delay (4ms)
- Rotation delay: (60*1000/15,000)/2 = 2ms
- Transfer delay
  - 125MBps = 125 Bytes per us.
  - Time take to read 4KB: 4096/125 ~ 30us
- 4KB random read: 4ms (seek) + 2ms (rotation) + 30us (transfer) ~ 6ms

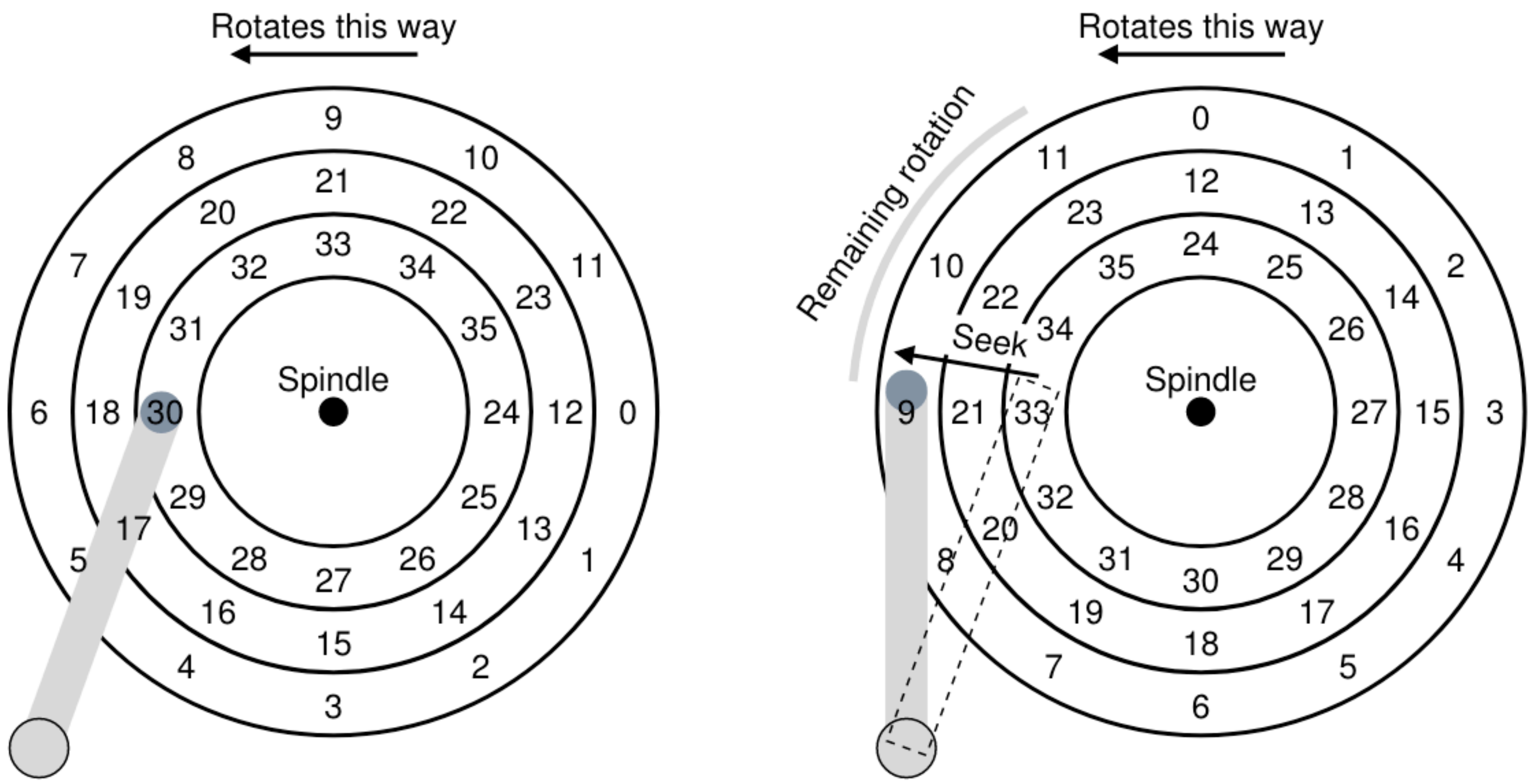


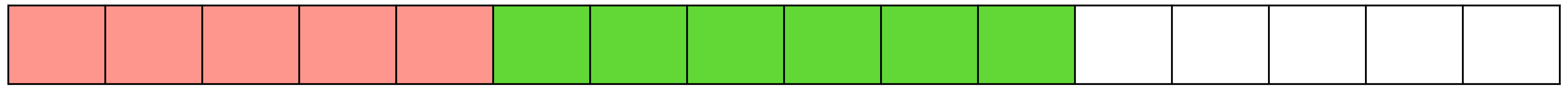
Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

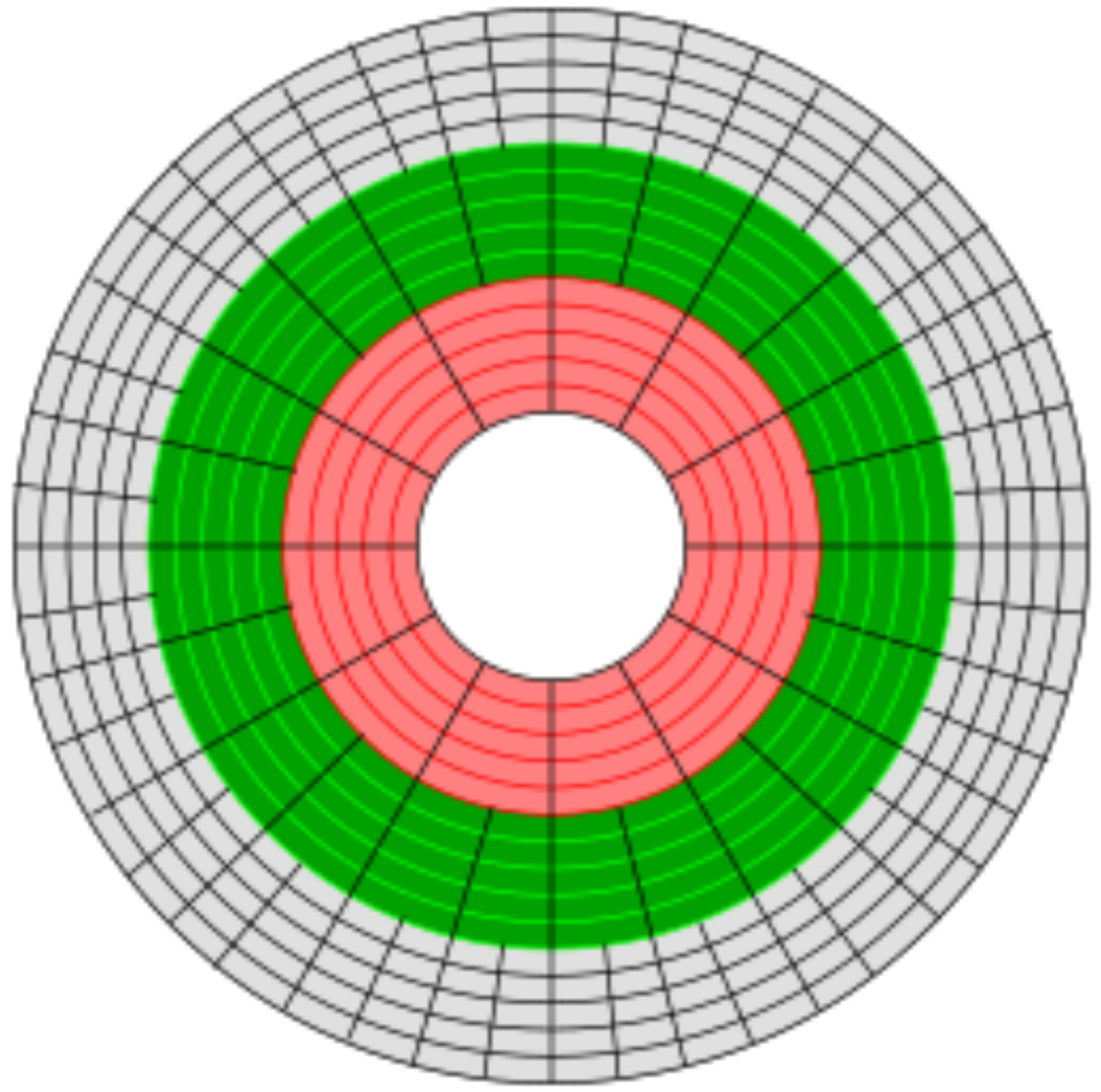
Random rws are ~100x slower than sequential rws!

- Random read: 4ms (seek time) + 2ms (rotation time) + 30us (transfer time) ~ 6ms
- Sequential read: 30us (transfer time)

CPU-disk interface: logical block addressing (LBA)

- CHS made addressing cumbersome. Strongly ties to disk geometry (not a good abstraction!).
    - Outer tracks can have more sectors
- Disk controller abstracts out such details
    - LBA interface: read block number 293
- Close block numbers are close on disk





Disk scheduling problem

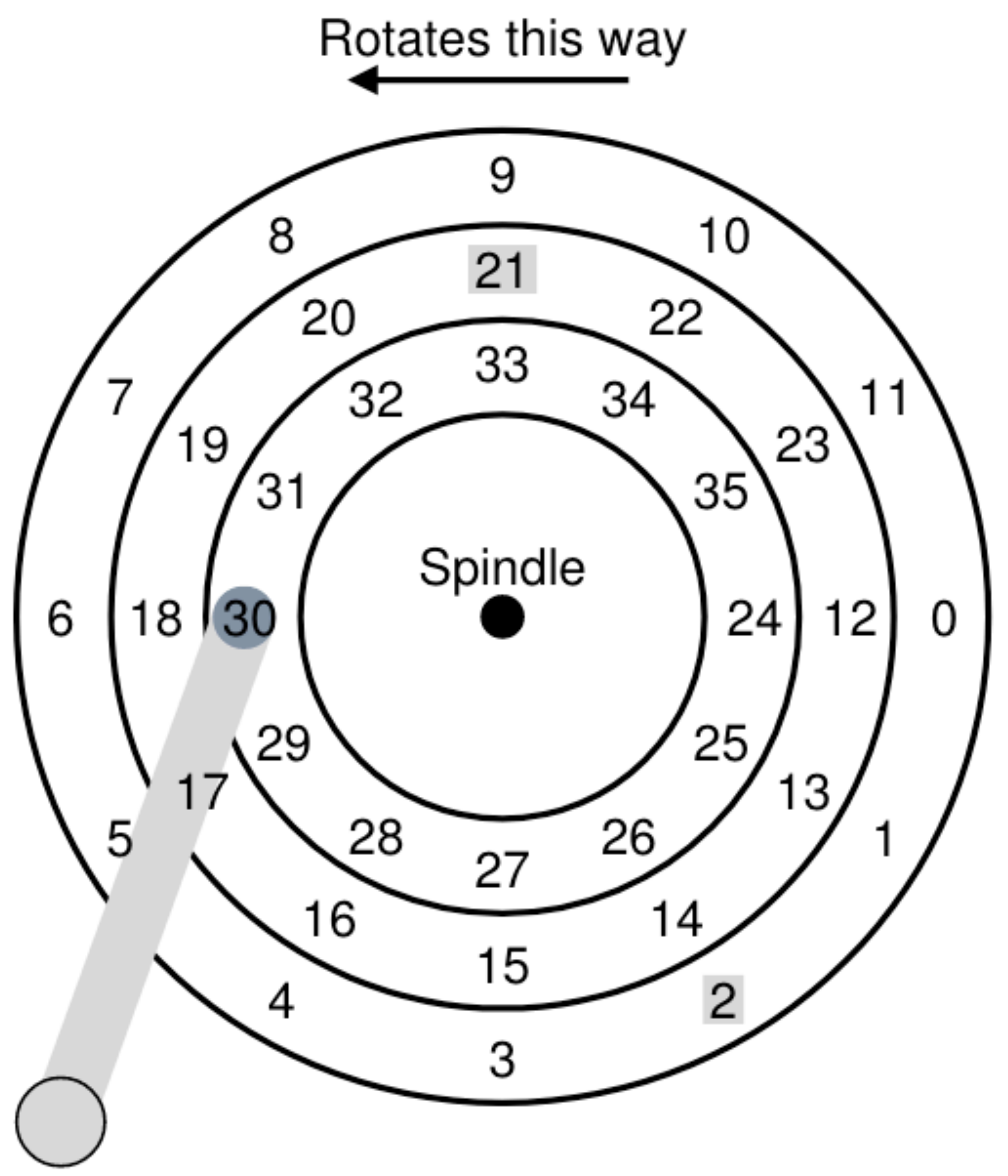
- python3 disk.py -a 10,15,32,11,33,16 -G

Total:1395

- python3 disk.py -a 10,11,15,16,32,33 -G

Total: 465

- Given a sequence of requests, reorder requests to service them quicker



Shortest job first

Greedy algorithm to minimize average waiting time

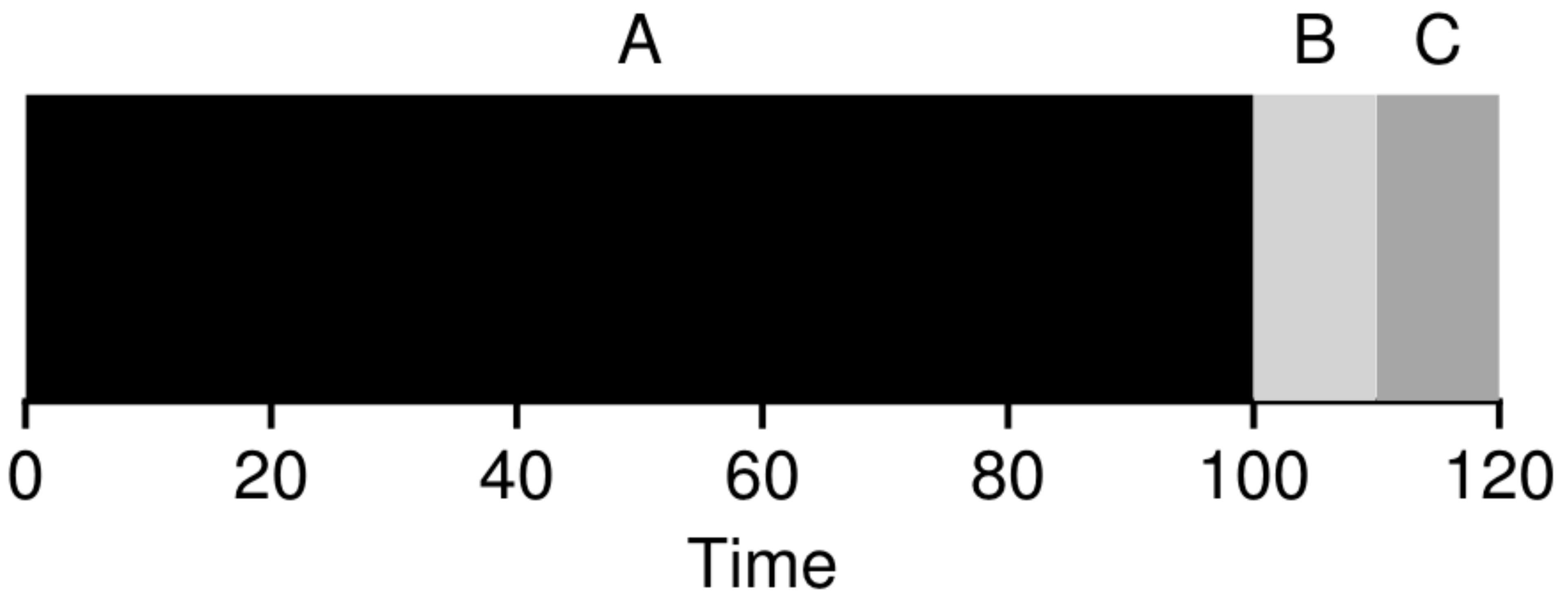


Figure 7.2: **Why FIFO Is Not That Great**

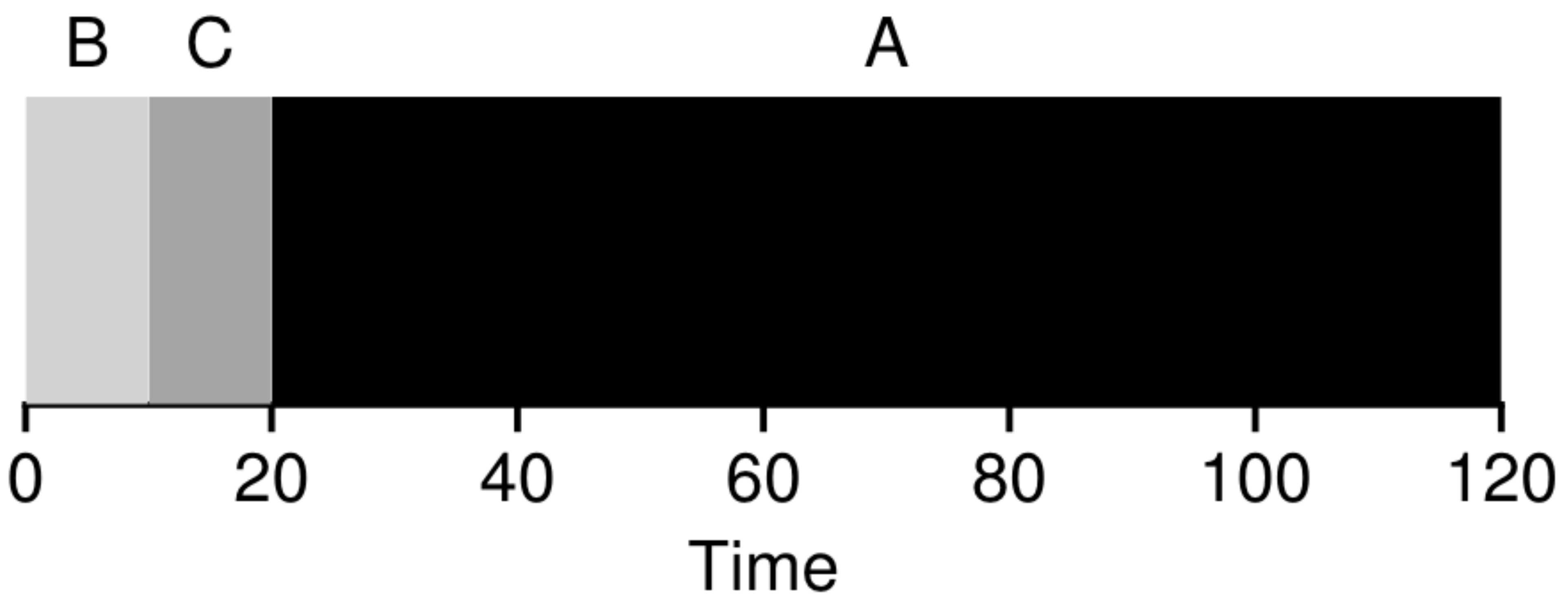


Figure 7.3: **SJF Simple Example**

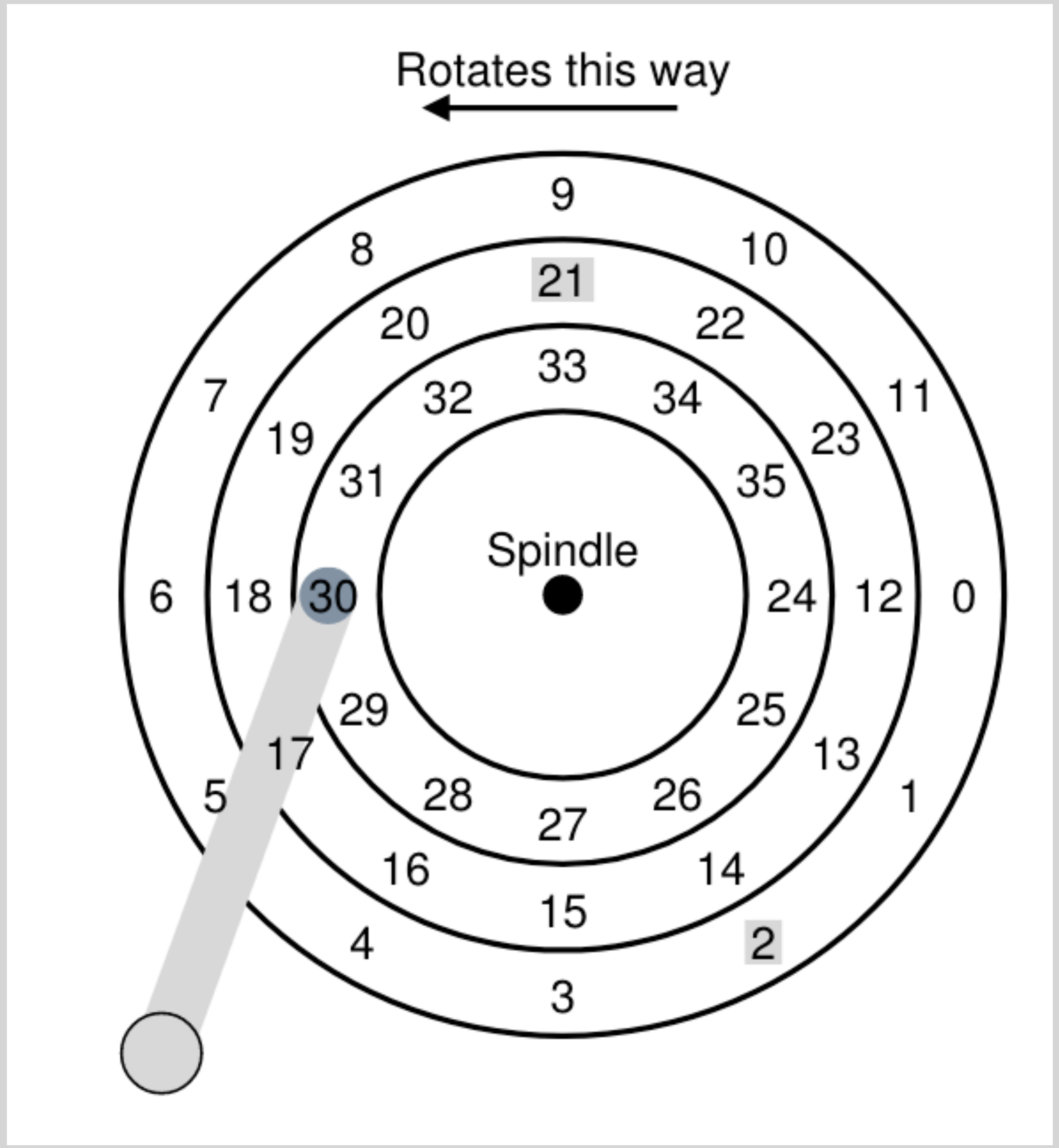
Strategy	Average waiting time
FIFO	(100 + 110 + 120)/3 = 110
SJF	(10 + 20 + 120)/3 = 50

Shortest seek time first

- Closer tracks first.
- python3 disk.py -a 10,15,32,11,33,16 -p SSTF —G
- Starvation problem

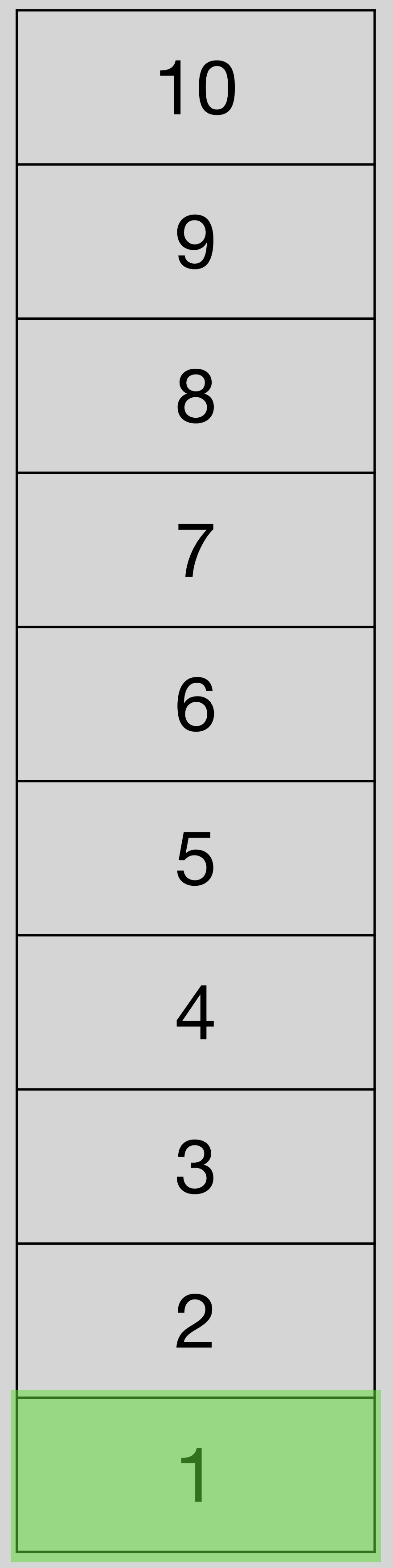
$\longrightarrow$ Time

🟩 31								
2								
21	🟩							
	34	🟩						
		23	🟩					
			24	🟩				
				26	🟩			
					15	🟩		
						28	🟩	
							17	🟩



Elevator analogy

- You get on, press 10
- At floor 8, P1 gets on, P1 presses 7
- At floor 7, P1 gets down, P2 gets on, P2 presses 5
- At floor 5, P2 gets down, P3 gets on, P3 presses 1



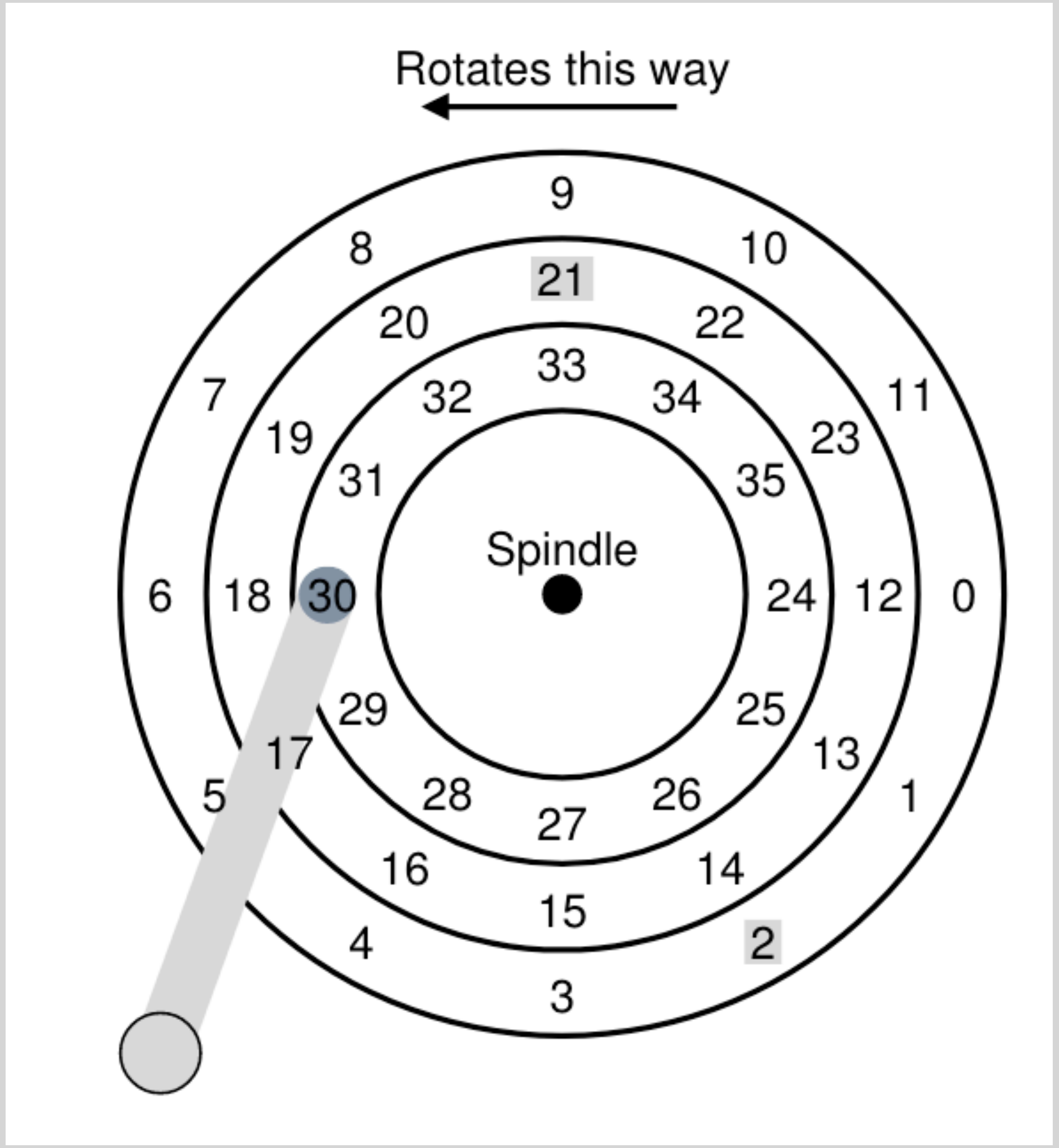
Elevator algorithm

Fix starvation

- Closer tracks first but sweep end-to-end

→ Time

🟩 31									
2		🟩							
21	🟩								
	34					🟩			
		23	🟩						
			24	🟩					
				26	🟩				
					15				🟩
						28	🟩		
							17	🟩	

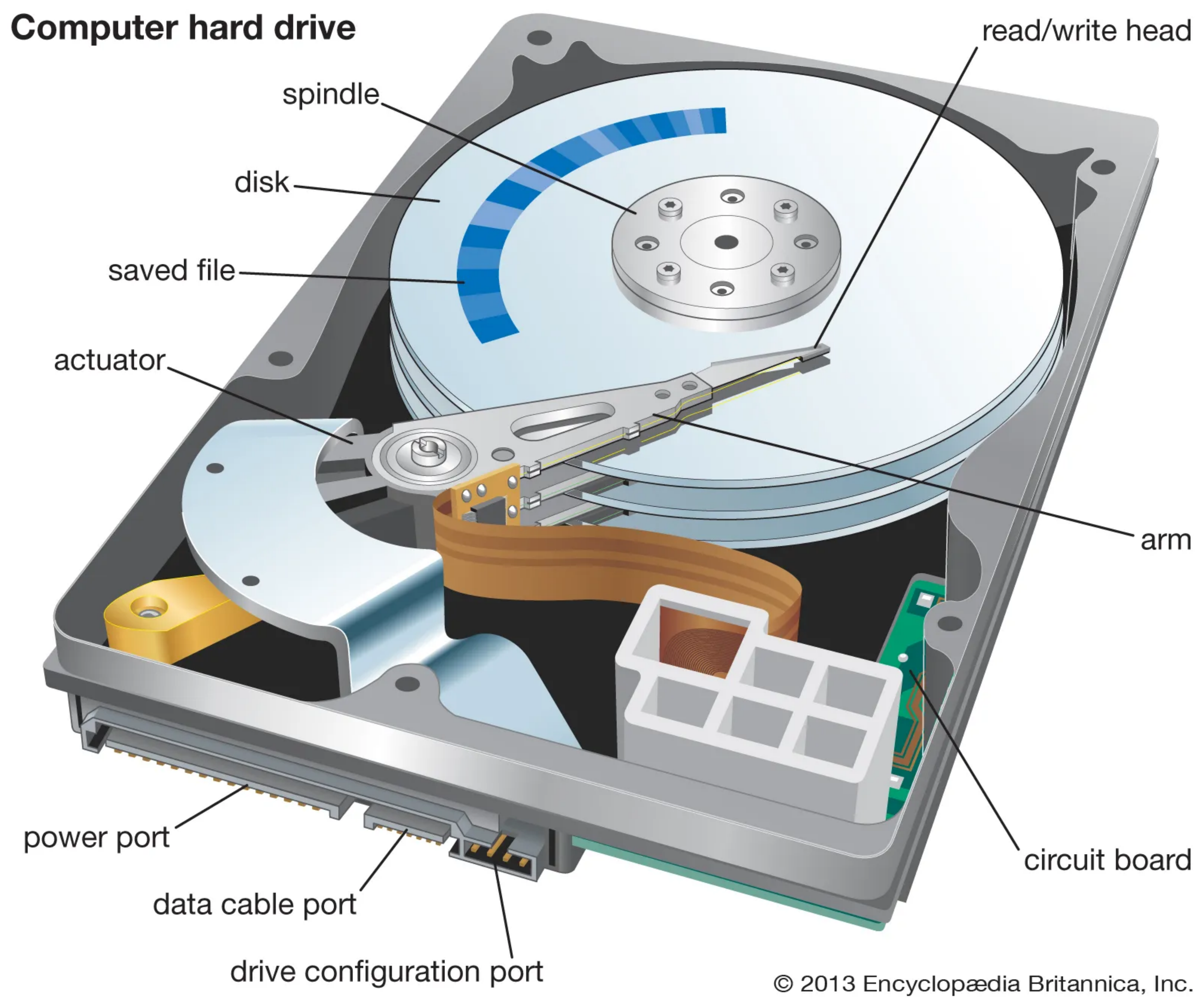


Disk scheduling

- OS does not know where the disk head is etc.,
  OS will send multiple outstanding read/write requests
- Disk controller will do disk scheduling.
  OS will do bookkeeping on which request is complete.
- Xv6 will send one request at a time in FIFO manner. No out-of-order request bookkeeping.

Disk problems

- Disks are slow
  - ~100MBps compared to memory ~100GBps
- Disks can fail.
  - Fail stop model
- Disks have limited capacity
  - Not true for medium scale. True for data centers.



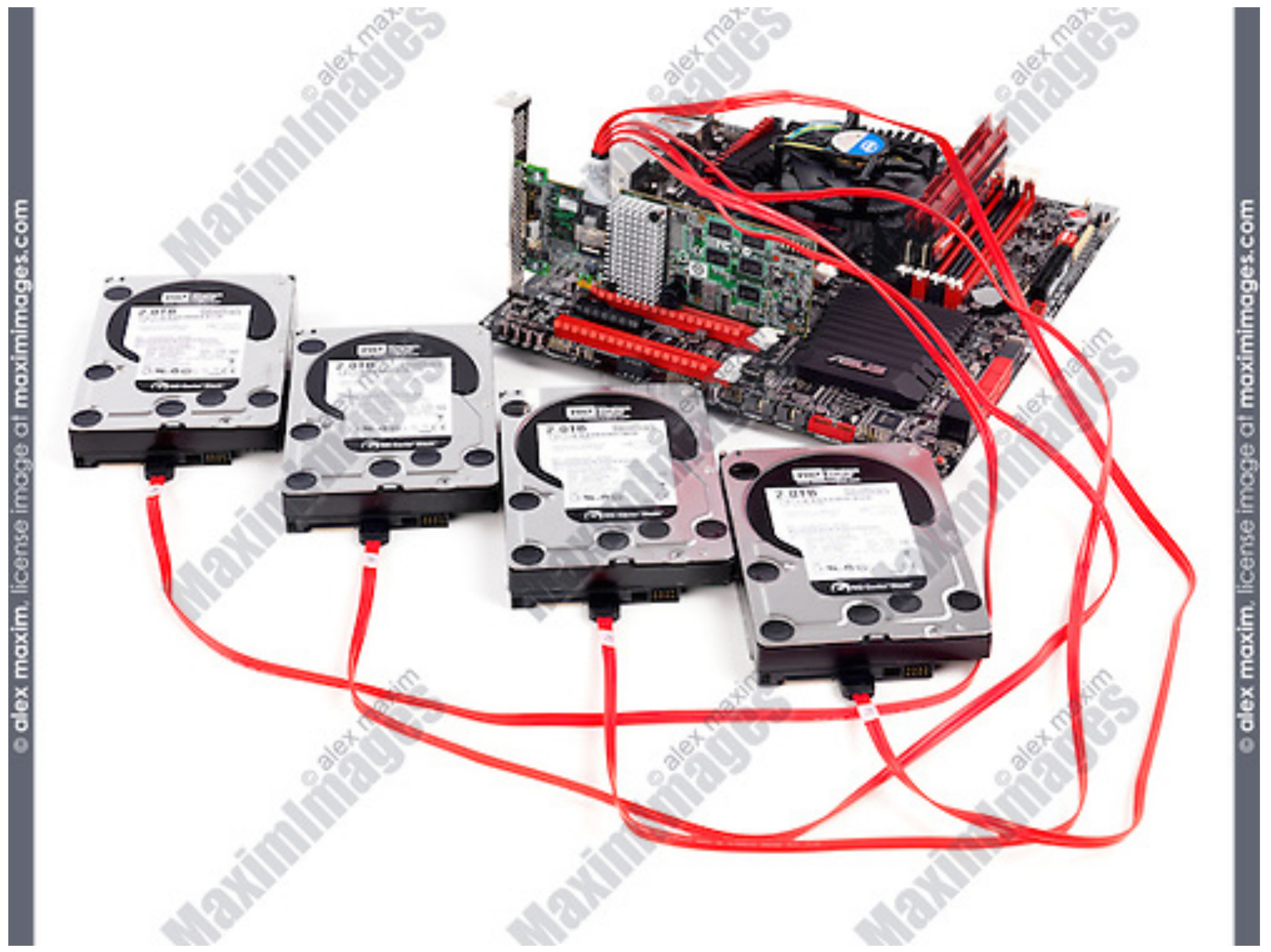
Redundant Array of Inexpensive disks (RAID)

I think I am talking to a single disk

CPU

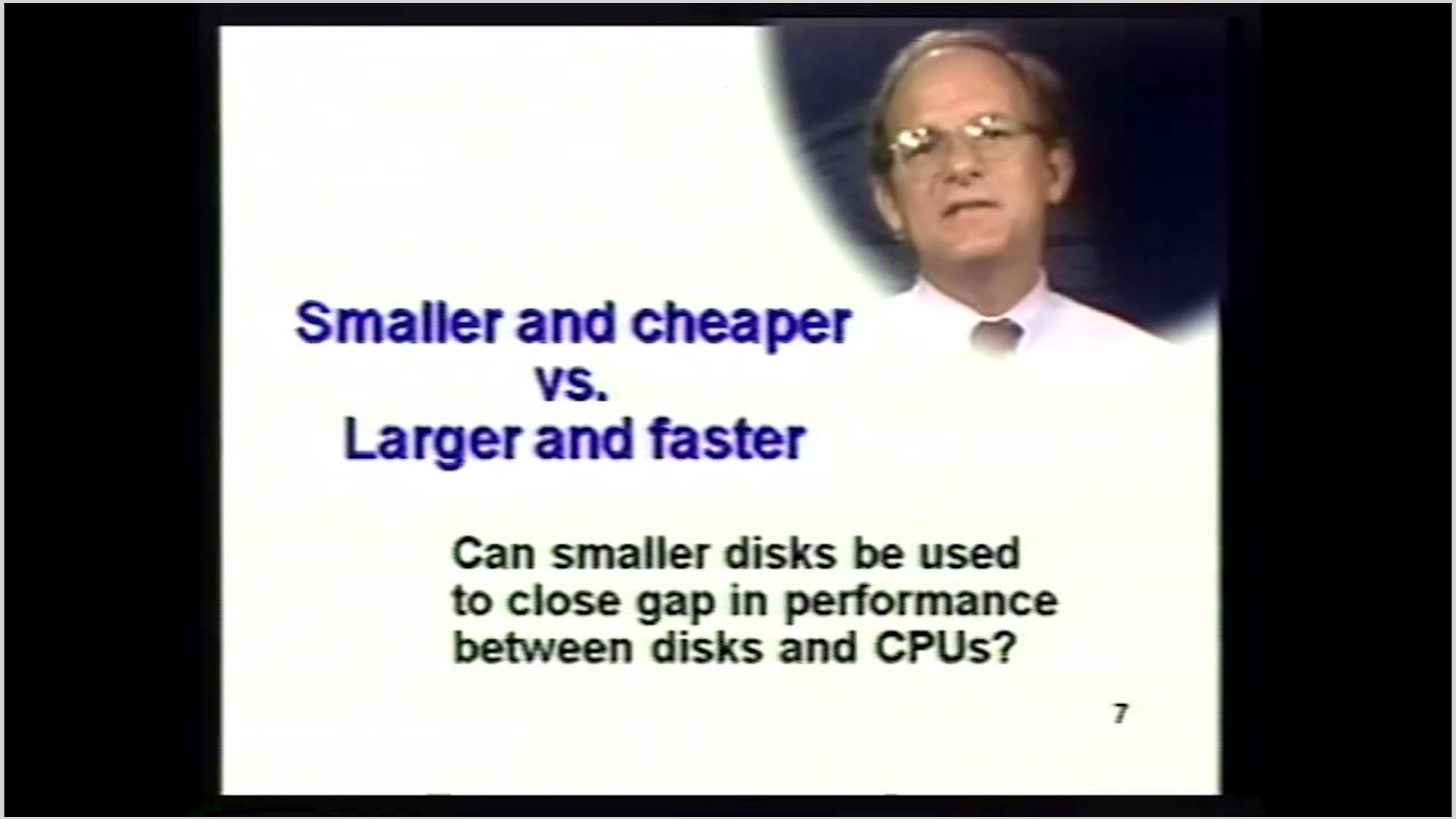
read block number 293

- Use multiple disks. Expose like a single disk
- Deployment principle: Need minimal changes to existing setup



Redundant array of inexpensive disks (RAID)

Ch. 38 OSTEP book



Smaller and cheaper
vs.
Larger and faster

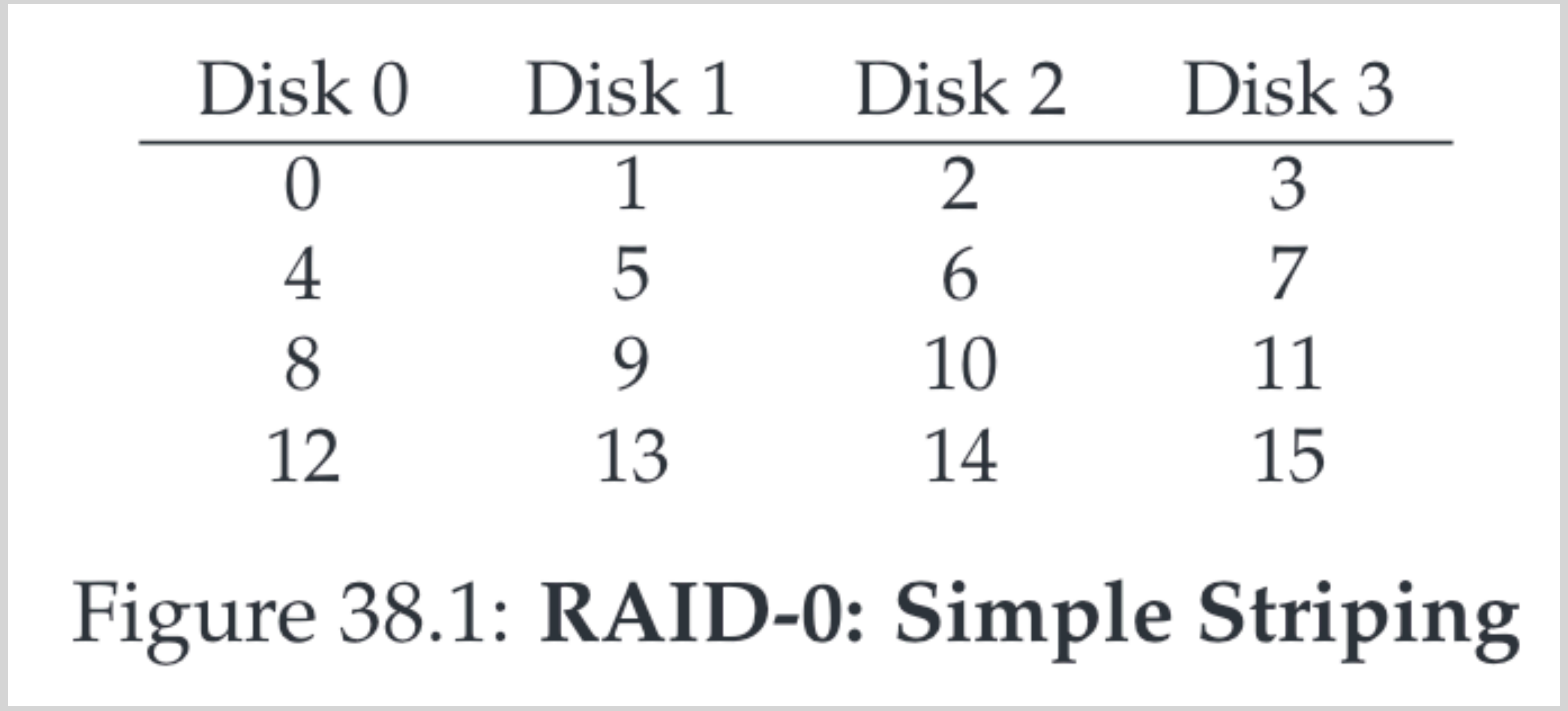
Can smaller disks be used
to close gap in performance
between disks and CPUs?

7

RAID-0 striping

Capacity	N * B
Fault tolerance	0 disk crashes

- Assume N disks. Each disk has
  - Capacity = B
  - Sequential read/write throughput=S
  - Random read/write throughput=R

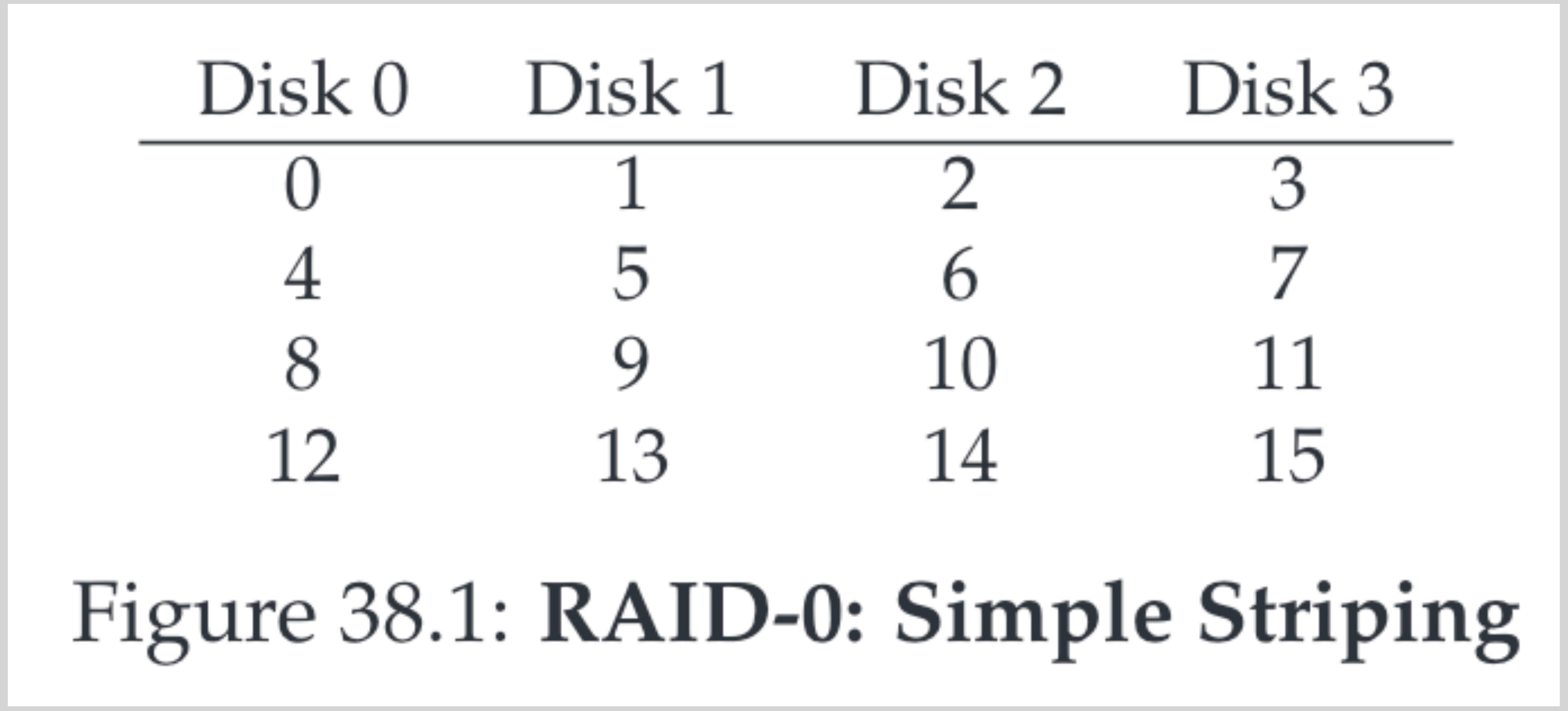


Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

Writing to RAID-0

- RAID controller
  - rw block X
  - Ask disk# (X%N) to rw block# $\lceil X/N \rceil$
- Sequential (random) rw become sequential (random) rw to disks
  - Throughput: NS, NR

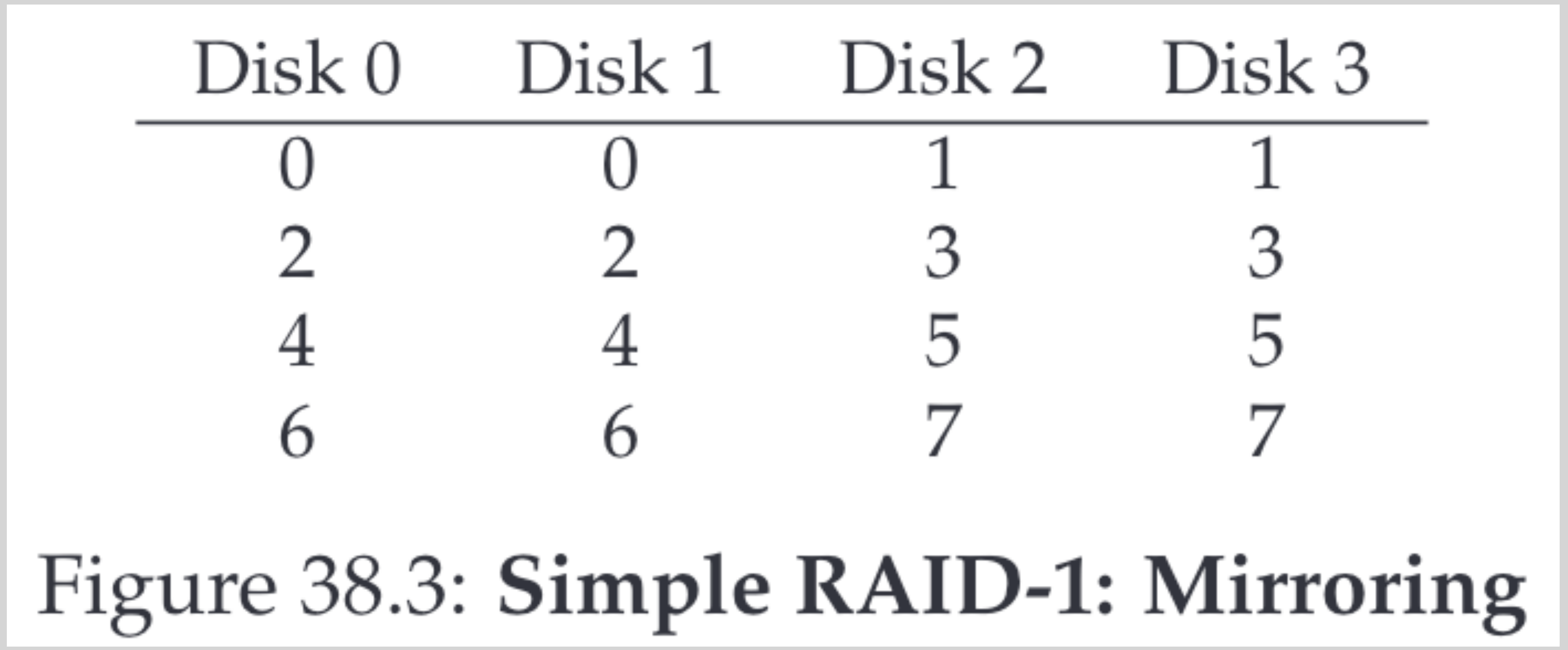


Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

RAID-1 mirroring

Capacity	N/2 * B
Fault tolerance	Definitely tolerate 1. Tolerate upto N/2 failures

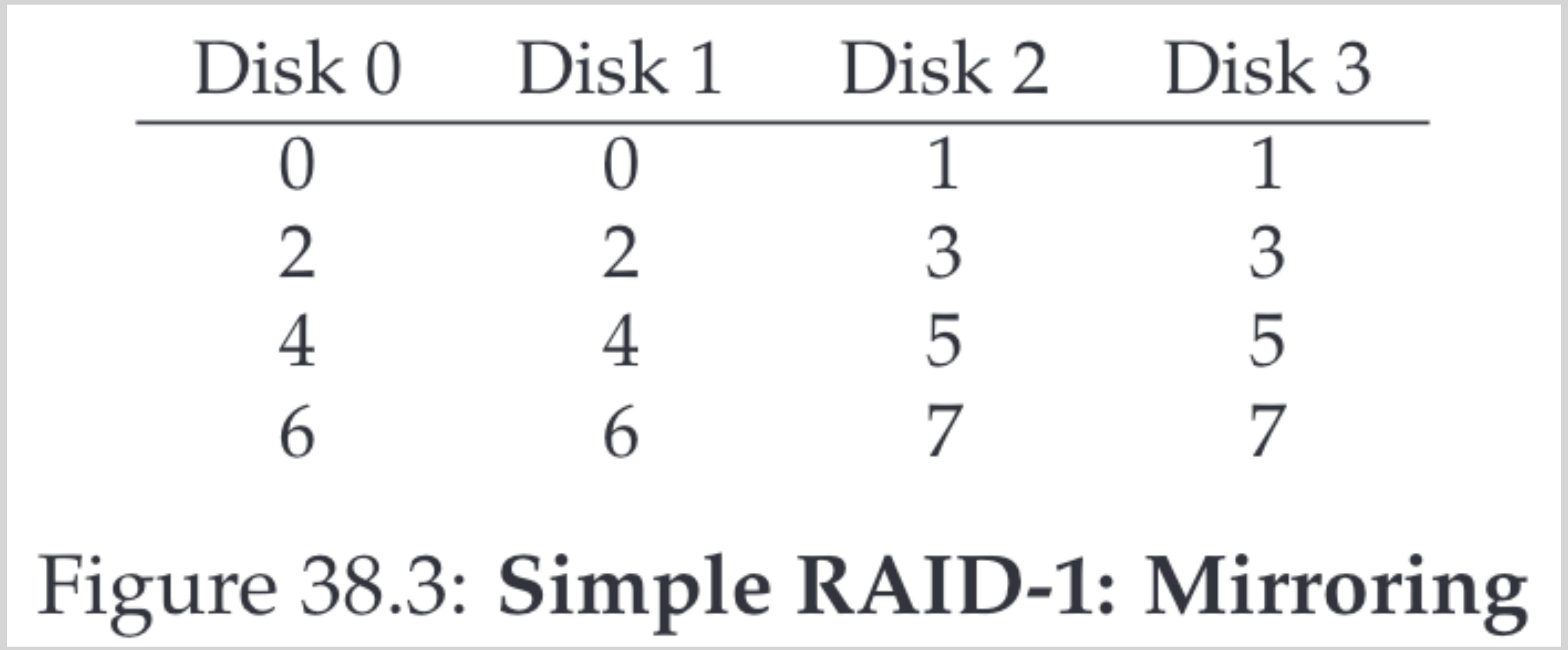


Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

Writing to RAID-1

- RAID controller
  - Write block X
  - Write to both disks
- Sequential write throughput: N/2 S
- Random write throughput: N/2 R

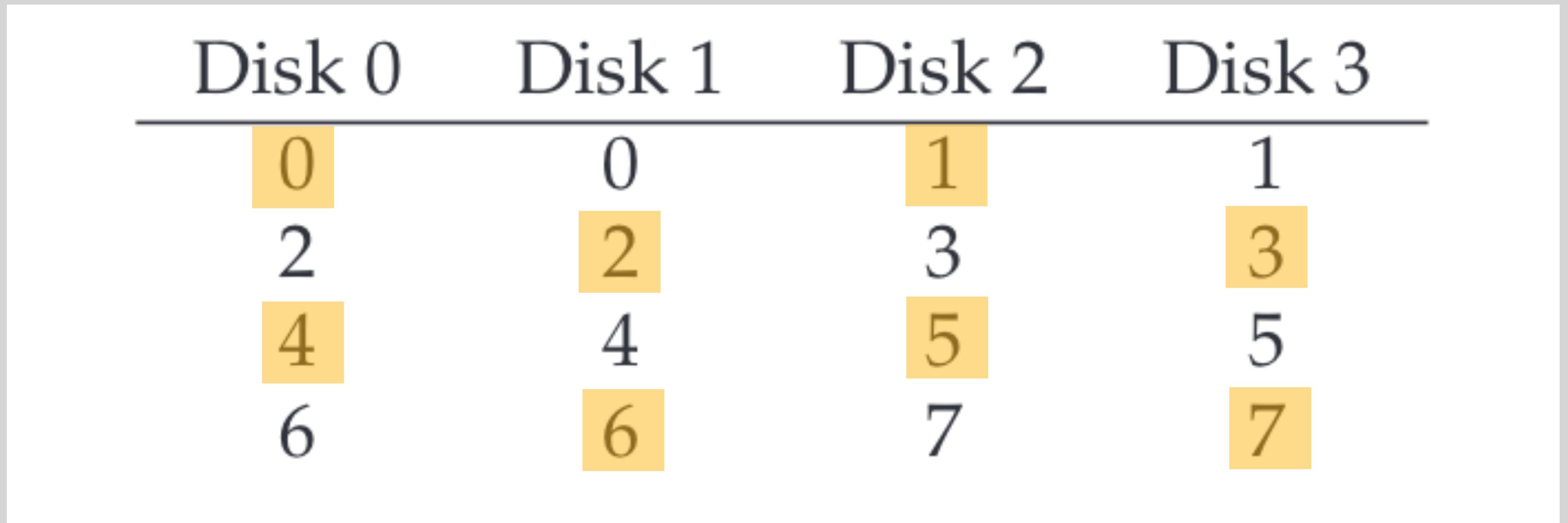


Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

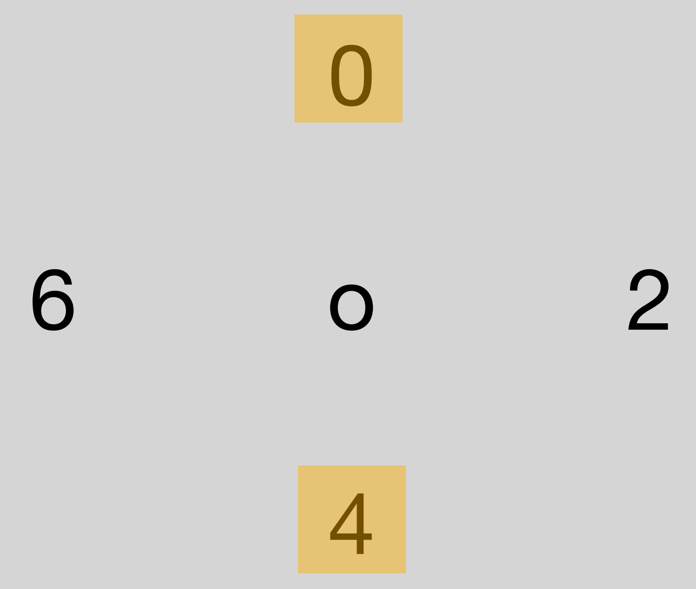
Reading from RAID-1

- RAID controller
  - Read block X
  - Choose either of the two disk
  - Forward read
- Random read throughput: NR
- Sequential read throughput: N/2 S
  - Disk 0 still has to pass over block 2 without serving read



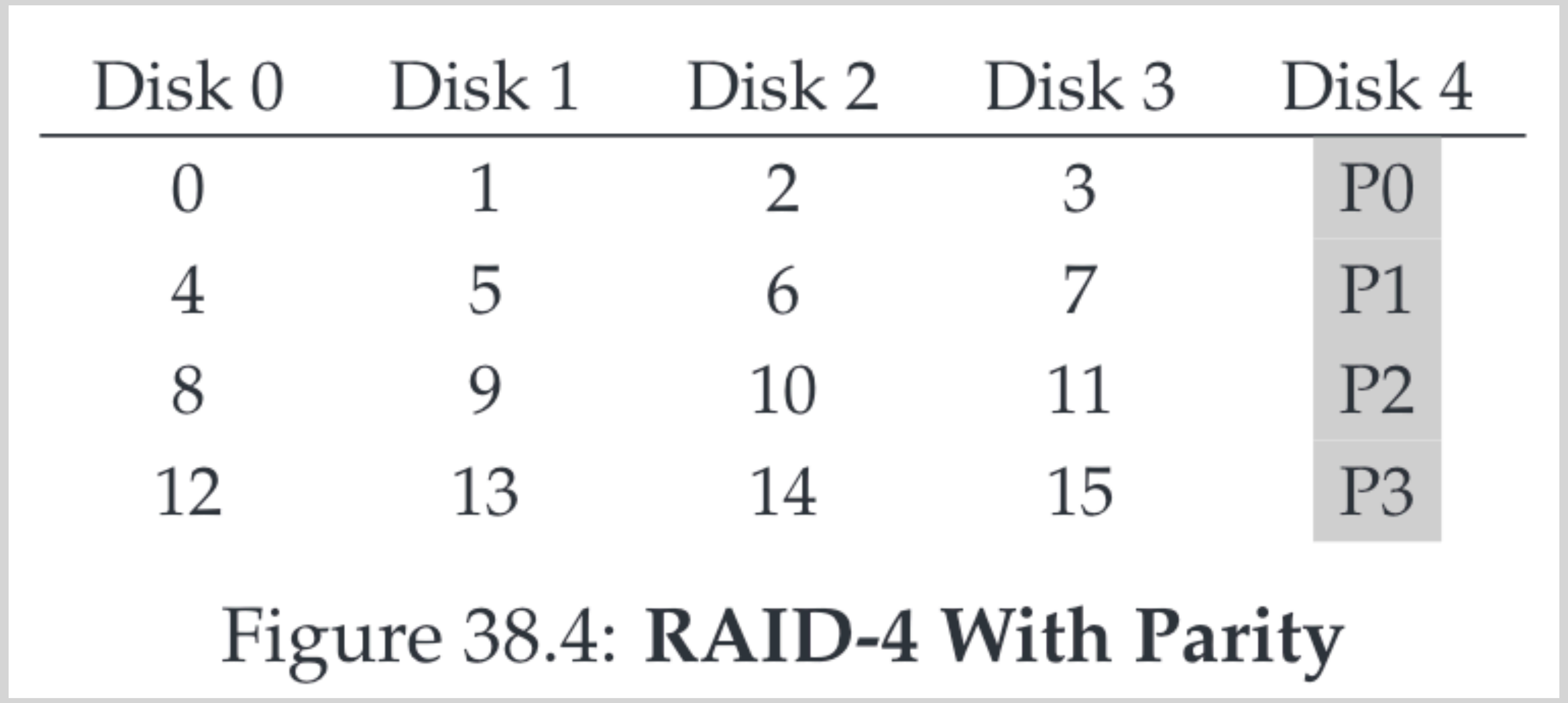
Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**



RAID-4 parity

- Parity: XOR bits
- Recovery: XOR bits
- $a = b \oplus c \implies a \oplus c = b$



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

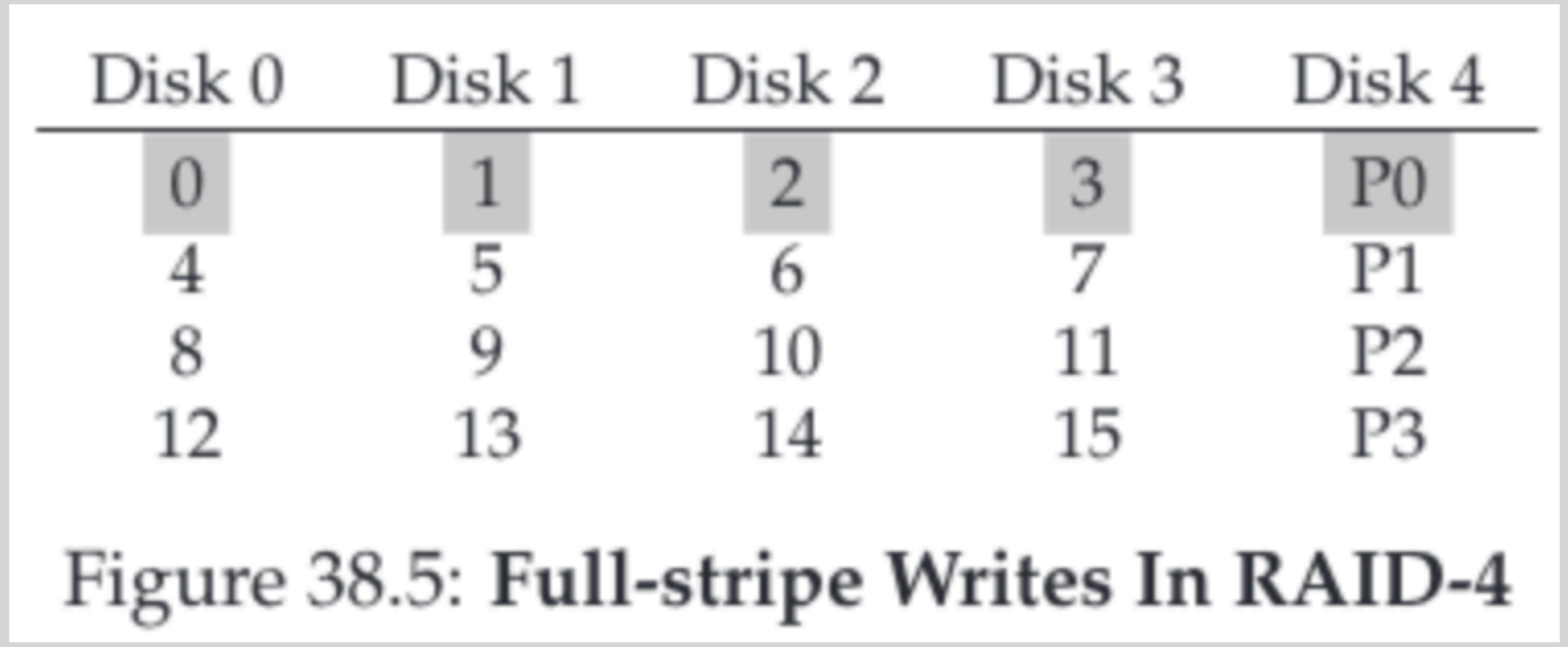
Figure 38.4: **RAID-4 With Parity**

C0	C1	C2	C3	P0
00	10	11	10	11

Capacity	(N-1)* B
Fault tolerance	1

Reading/Writing RAID-4

- Behaves like N-1 disk RAID-0 for reads
  - Sequential throughput: (N-1)S
  - Random throughput: (N-1)R
- Sequential write:
  - Compute $P0 = C0 \oplus C1 \oplus C2 \oplus C3$
  - 5 IO requests for 4 writes
  - Throughput = (N-1)S

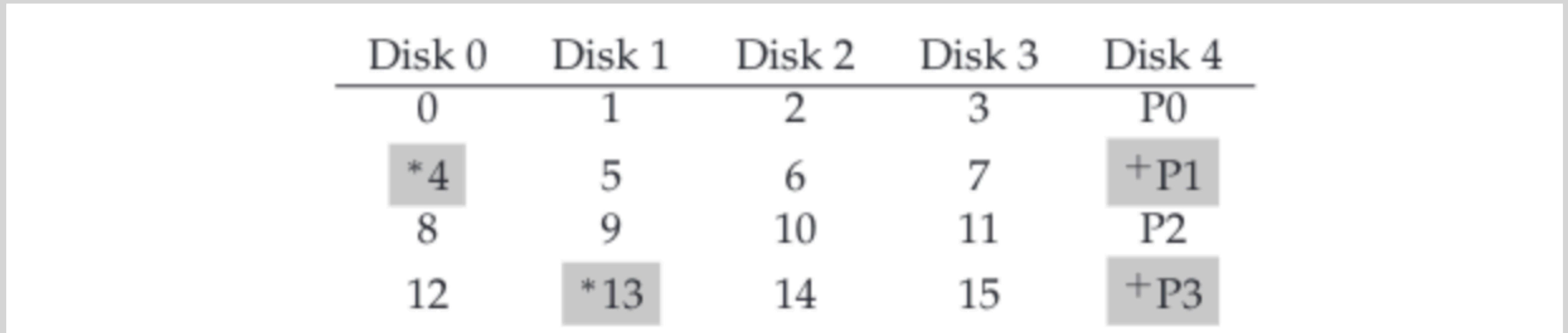


Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: **Full-stripe Writes In RAID-4**

Writing to RAID-4: Random write

- Read all blocks (4, 5, 6, 7) in stripe
- Compute parity P1
- Write 4, P1
- Total 6 IO requests for 1 write!
- $P1_{old} = C4_{old} \oplus C5_{old} \oplus C6_{old} \oplus C7_{old}$
- $C4_{old} \oplus P1_{old} = C5_{old} \oplus C6_{old} \oplus C7_{old}$
- $P1_{new} = C4_{new} \oplus C5_{old} \oplus C6_{old} \oplus C7_{old}$
- $P1_{new} = C4_{new} \oplus C4_{old} \oplus P1_{old}$

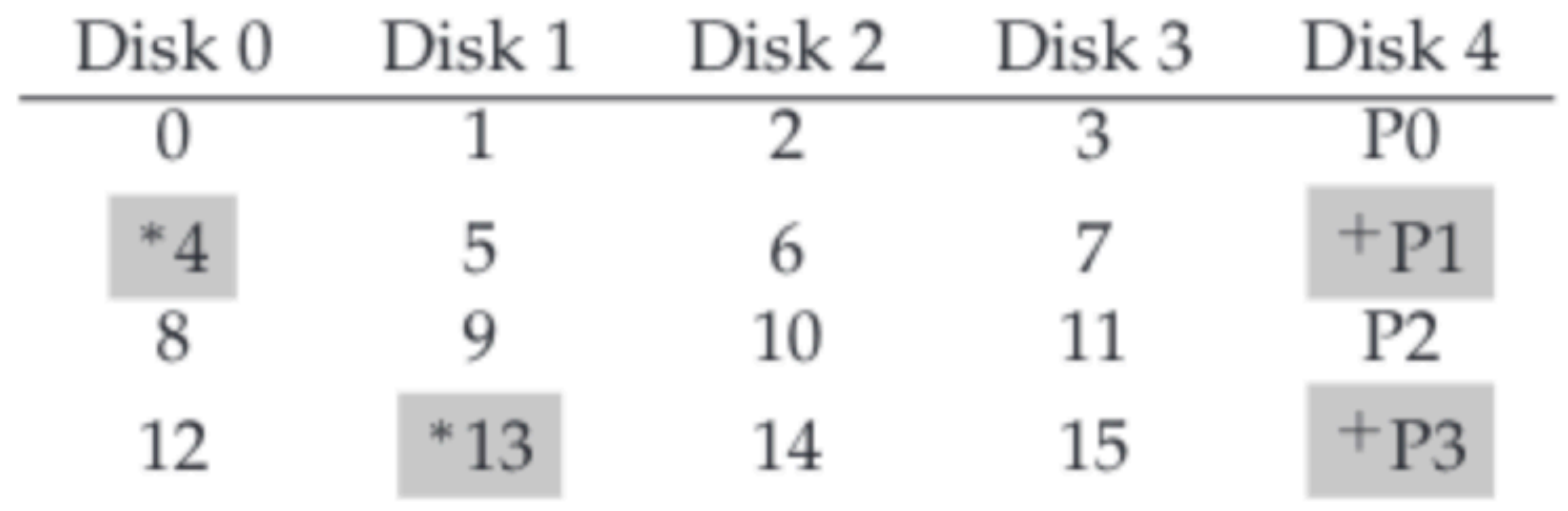


Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Figure 38.6: **Example: Writes To 4, 13, And Respective Parity Blocks**

Writing to RAID-4: Random write (2)

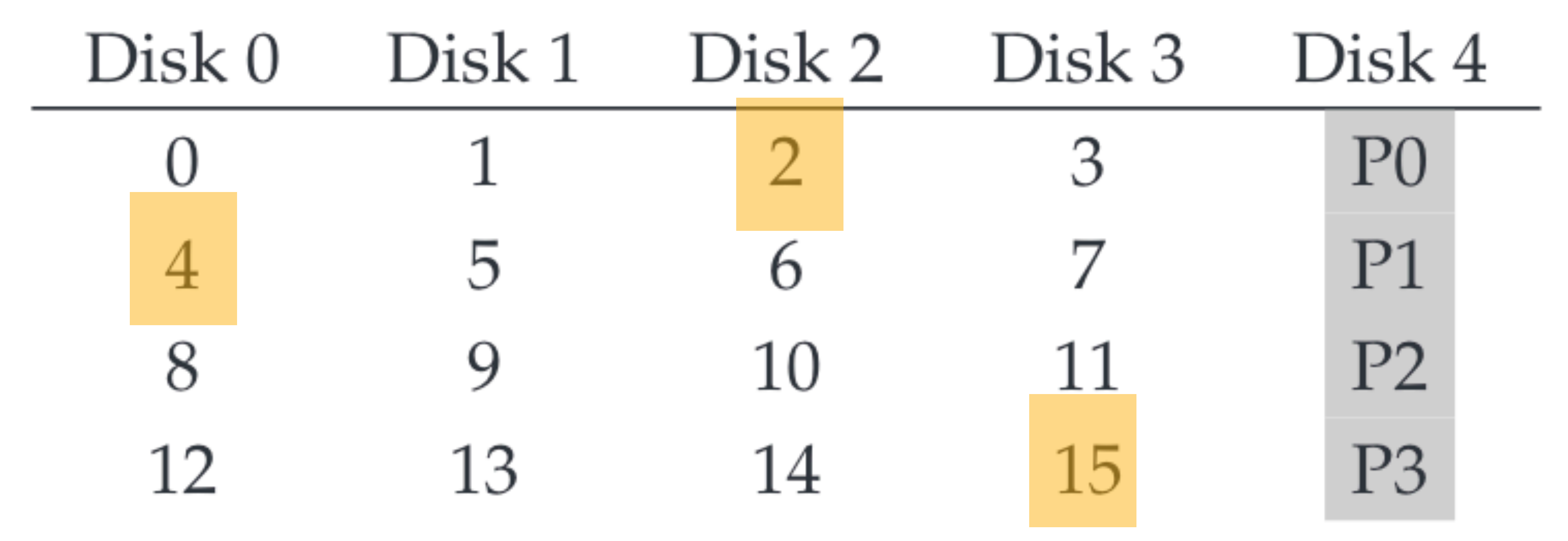
- Read 4, P1
- Compute parity
  $P1_{new} = C4_{new} \oplus C4_{old} \oplus P1_{old}$
- Write 4, P1
- Total ~~6~~ 4 IO requests for 1 write



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Figure 38.6: **Example: Writes To 4, 13, And Respective Parity Blocks**

Writing to RAID-4: Random write (3)



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

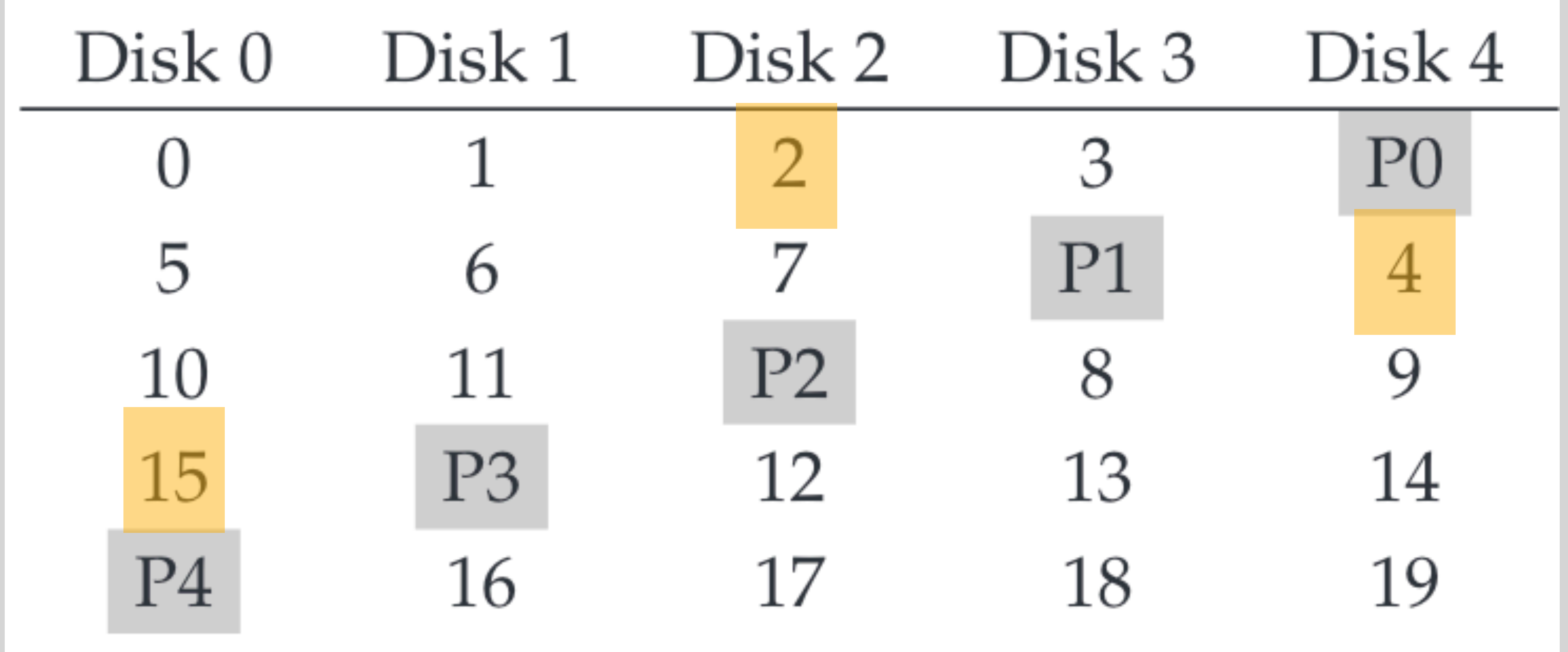
Figure 38.4: **RAID-4 With Parity**

- Random writes 2, 4, 15
- Read 2, P0, 4, P1, 15, P3
- Compute parity
  $P1_{new} = C4_{old} \oplus C4_{new} \oplus P1_{old}$, etc
- Write 2, P0, 4, P1, 15, P3
- For 3 random write requests: did 3 reads from, 3 writes to parity disk
- Random write throughput: R/2

Capacity	(N-1)* B
Fault tolerance	1
Random write bandwidth	1/2 * R

RAID-5 rotated parity

- Random writes 2, 4, 15
- Read 2, P0, 4, P1, 15, P3
- Compute parity
  $P1_{new} = C4_{old} \oplus C4_{new} \oplus P1_{old}, \ldots$
- Write 2, P0, 4, P1, 15, P3
- One random write leads to 4 random rw requests.
  But no single parity disk bottleneck
- Random write throughput = NR/4



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

RAID levels

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N*B	N/2*B	(N-1)*B	(N-1)* B
Fault tolerance	0	1. Upto N/2	1	1
Sequential rw throughput	N*S	N/2 * S	(N-1)*S	(N-1) * S
Random read throughput	N*R	N*R	(N-1)*R	N * R
Random write throughput	N*R	N/2 * R	1/2 * R	N/4 * R
Read Latency	T	T	T	T
Sequential write latency	T	T	T	T
Random write latency	T	T	2T	2T

Buffer cache: Code walkthrough

p6-ide

- Disk is slow: cache disk blocks in memory
- `make fs` prepares a “file system” with two blocks. First block has “welcome.txt”. Second block is zero. main.c prints first block and increments the value in the second block.
- buf.h defines buffers. Each buffer has a refcnt to indicate how many callers have reference to the buffer. Buffers also have flags. Valid buffers have been read from the disk. Dirty buffers have been modified, but not yet written to disk.
- bio.c has a linked list of buffers. bget finds an unused buffer and increments refcnt. brelease decrements refcnt.
- ide.c maintains a queue of disk requests. iderw starts a disk request and waits for the buffer to be ready. idestart asks disk to raise interrupt and sends request. ideintr updates buffer flags.

Preparing a file system image

Makefile

- `make fs` prepares a “file system” with two blocks. First block has “welcome.txt”. Second block is zero.

Makefile

```
fs:
  dd if=/dev/zero of=fs.img count=2
  dd if=welcome.txt of=fs.img conv=notrunc
```

fs.img

```
                                                         ###
#    #  ######  #        ####    ####   #    #  ######   ###
#    #  #       #       #    #  #    #  ##  ##  #        ###
#    #  #####   #       #       #    #  # ## #  #####     #
# ## #  #       #       #       #    #  #    #  #
##  ##  #       #       #    #  #    #  #    #  #        ###
#    #  ######  ######   ####    ####   #    #  ######   ###
```

Reading/writing/releasing buffers

main.c

- main.c prints first block and increments the value in the second block.

```
struct buf *b1 = bread(1, 1);

cprintf("Rebooted %d times\n", b1->data[0]);

b1->data[0] = b1->data[0] + 1;

bwrite(b1);

brelse(b1);
```

Buffer cache

buf.h

- Doubly linked list in LRU order
- Can be in disk queue waiting to be read/ written to disk
- Mark blocks as valid when they are read, dirty when they are written
- refcnt: how many callers have reference to the buffer

```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

Buffer cache

bio.c

- bio.c has a linked list of buffers.
- bget finds if block is already in the buffer cache, otherwise finds an unused buffer, and increments refcnt.
- brelse decrements refcnt. When zero, can be used to cache other disk blocks

```
static struct buf* bget(uint dev, uint blockno) {
  struct buf *b;
  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++; return b;
    }
  }
   // Not cached; recycle an unused buffer.
   for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
        if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0) {
          b->refcnt = 1; …
        }
   }
}
void brelse(struct buf *b) {
  B->refcnt--;
   ..
}
```

Buffer cache

bio.c

- bread returns the buffer if block is already in cache, otherwise asks disk driver to read it
- bwrite marks the buffer as dirty and asks disk driver to write it

```
struct buf* bread(uint dev, uint blockno) {
  struct buf *b;
  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0)
    iderw(b);
  return b;
}

void bwrite(struct buf *b) {
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Disk driver

ide.c

- Enable disk interrupts at init
- iderw queues the request and waits for buffer to become valid

```
void ideinit(void) {
  ioapicenable(IRQ_IDE, ncpu - 1);
  ..
}


void iderw(struct buf *b) {
  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    noop();
}
```

Disk driver

ide.c

- Interrupt handler reads data if needed and updates flags

```
// Interrupt handler.
void ideintr(void){
  struct buf *b;
  ..

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, BSIZE/4);

  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  ..
}
```