Input output

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

Abhilash Jindal

Agenda

Agenda

- Overview of IO devices (OSTEP Ch. 36): Polling, Interrupts, Direct memory access

Agenda

- Overview of IO devices (OSTEP Ch. 36): Polling, Interrupts, Direct memory access
- Interrupt handling (xv6 Ch. 3): interrupt controllers, interrupt descriptor table

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

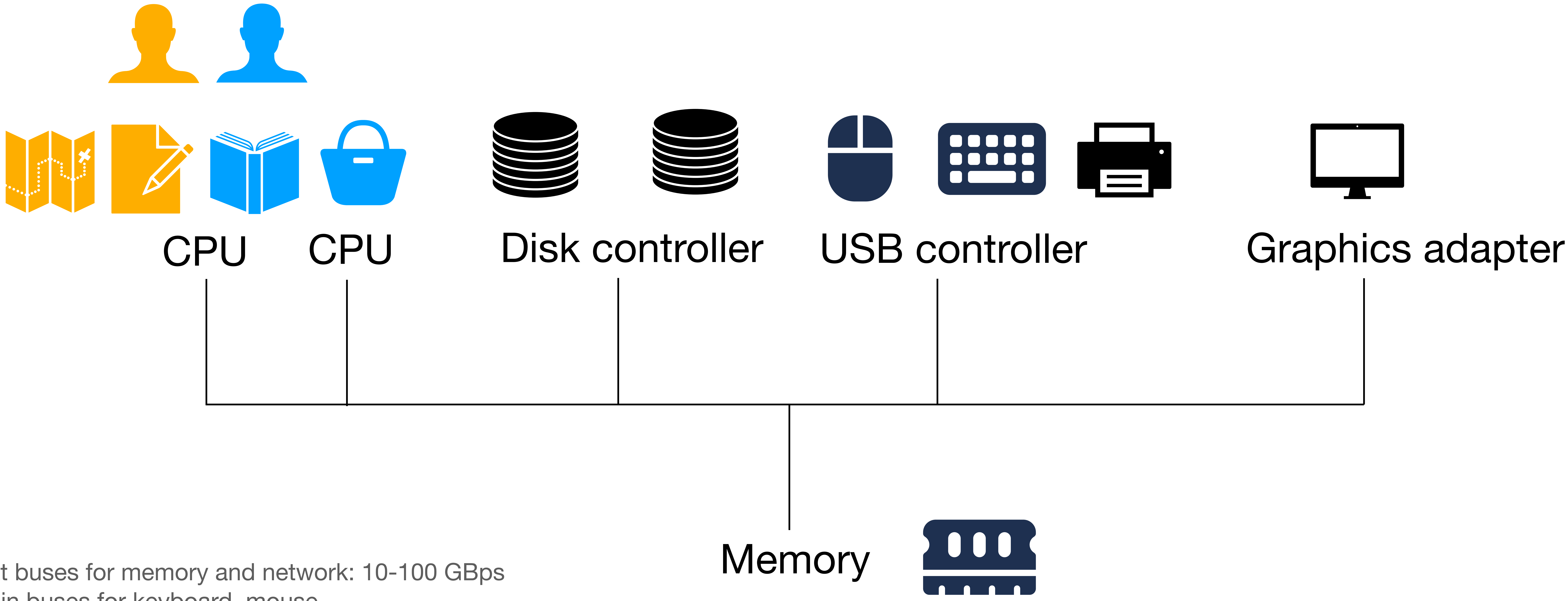
Agenda

- 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

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

Fitting into the OS

Hide device specific details in device driver

- Abstraction allows OS and applications to stay device-neutral

Fitting into the OS

Hide device specific details in device driver

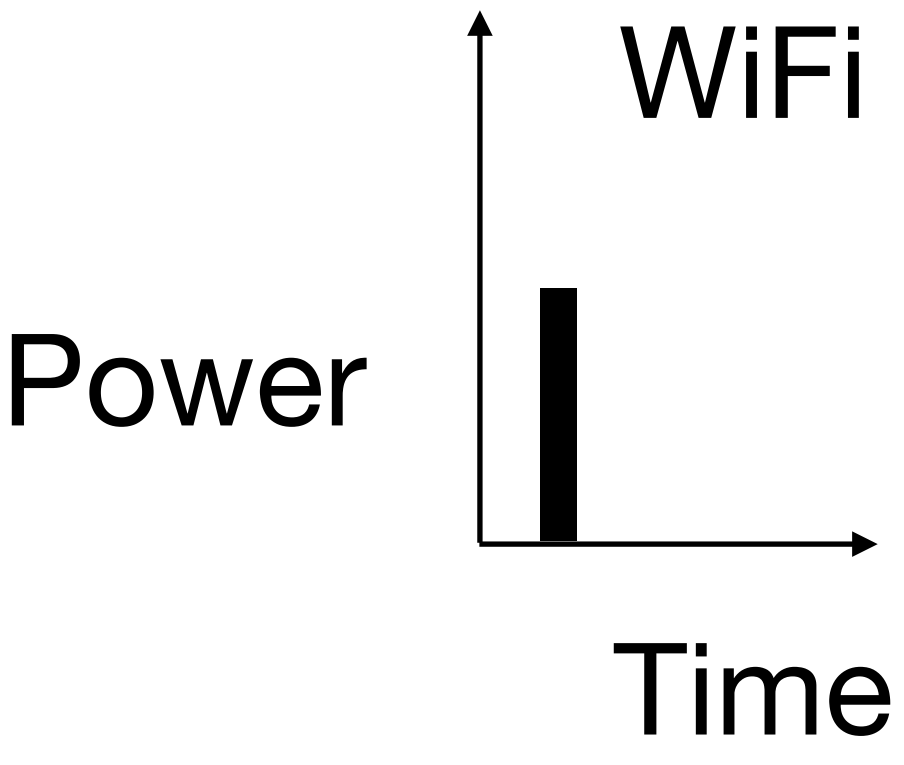
- Abstraction allows OS and applications to stay device-neutral
- Abstraction can hurt.

Fitting into the OS

Hide device specific details in device driver

- Abstraction allows OS and applications to stay device-neutral

- Abstraction can hurt.

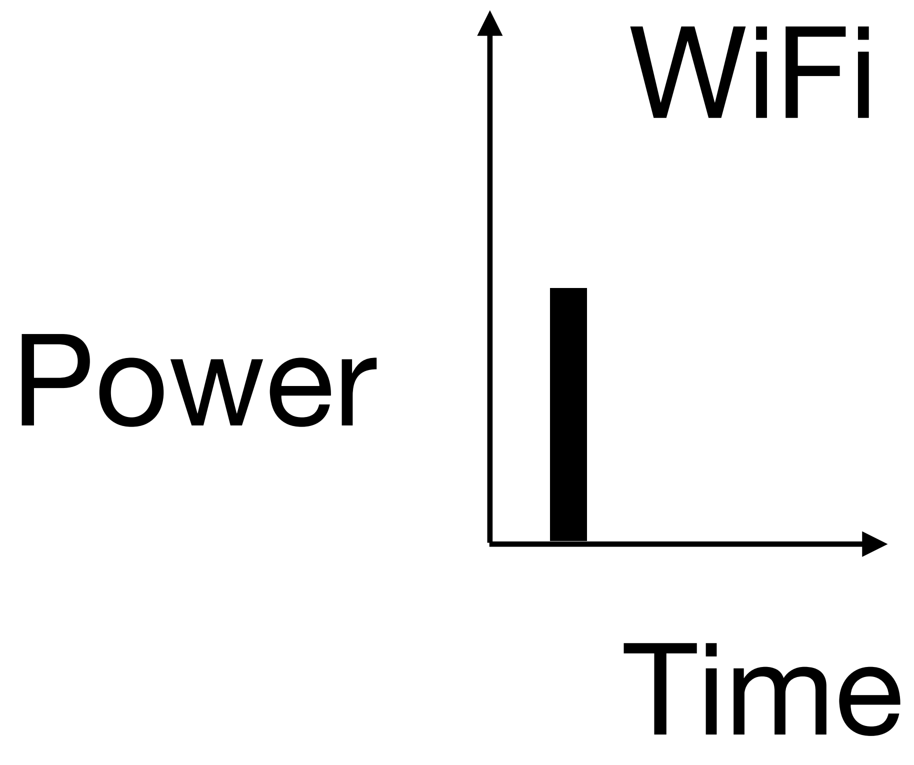


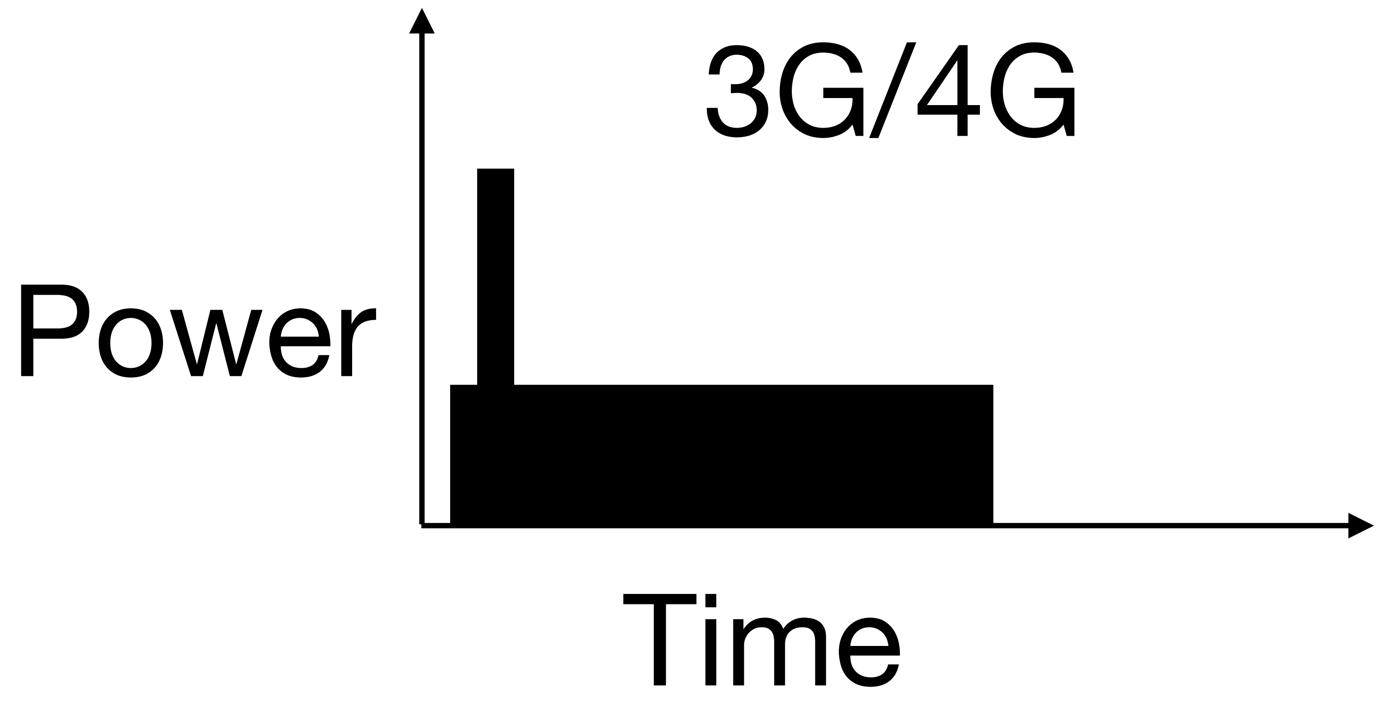
Fitting into the OS

Hide device specific details in device driver

- Abstraction allows OS and applications to stay device-neutral

- Abstraction can hurt.



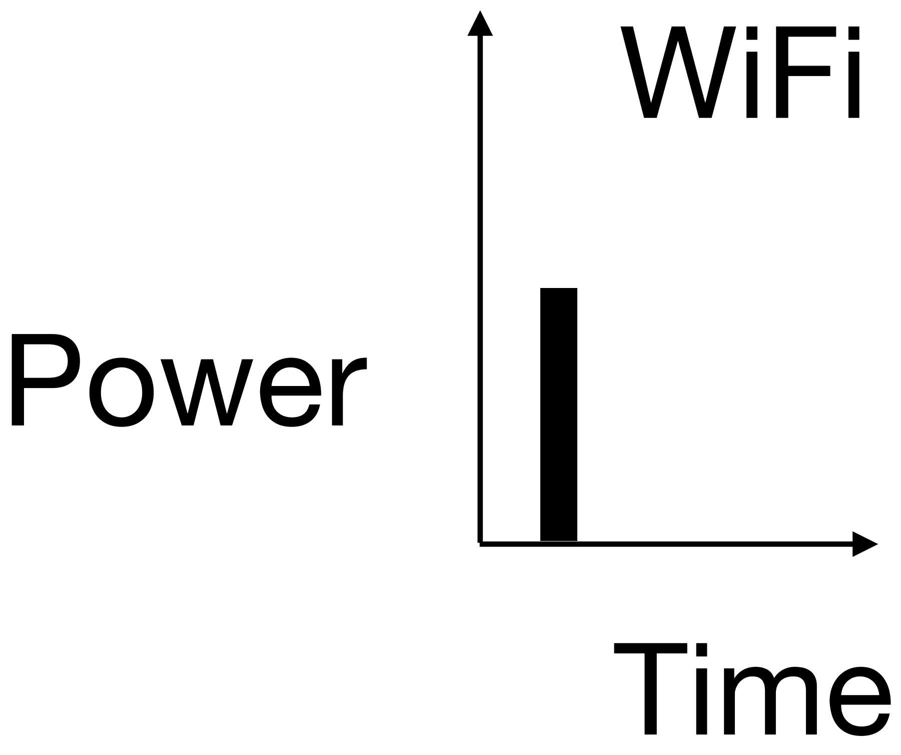


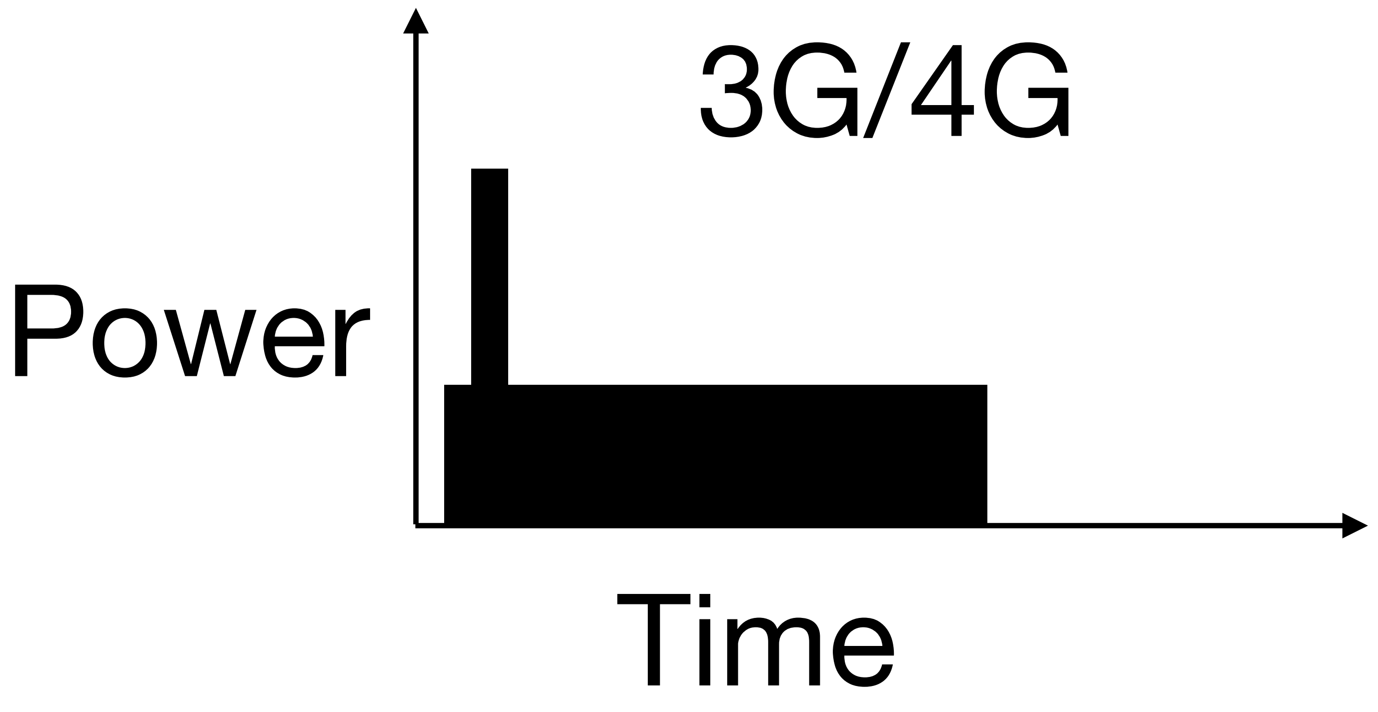
Fitting into the OS

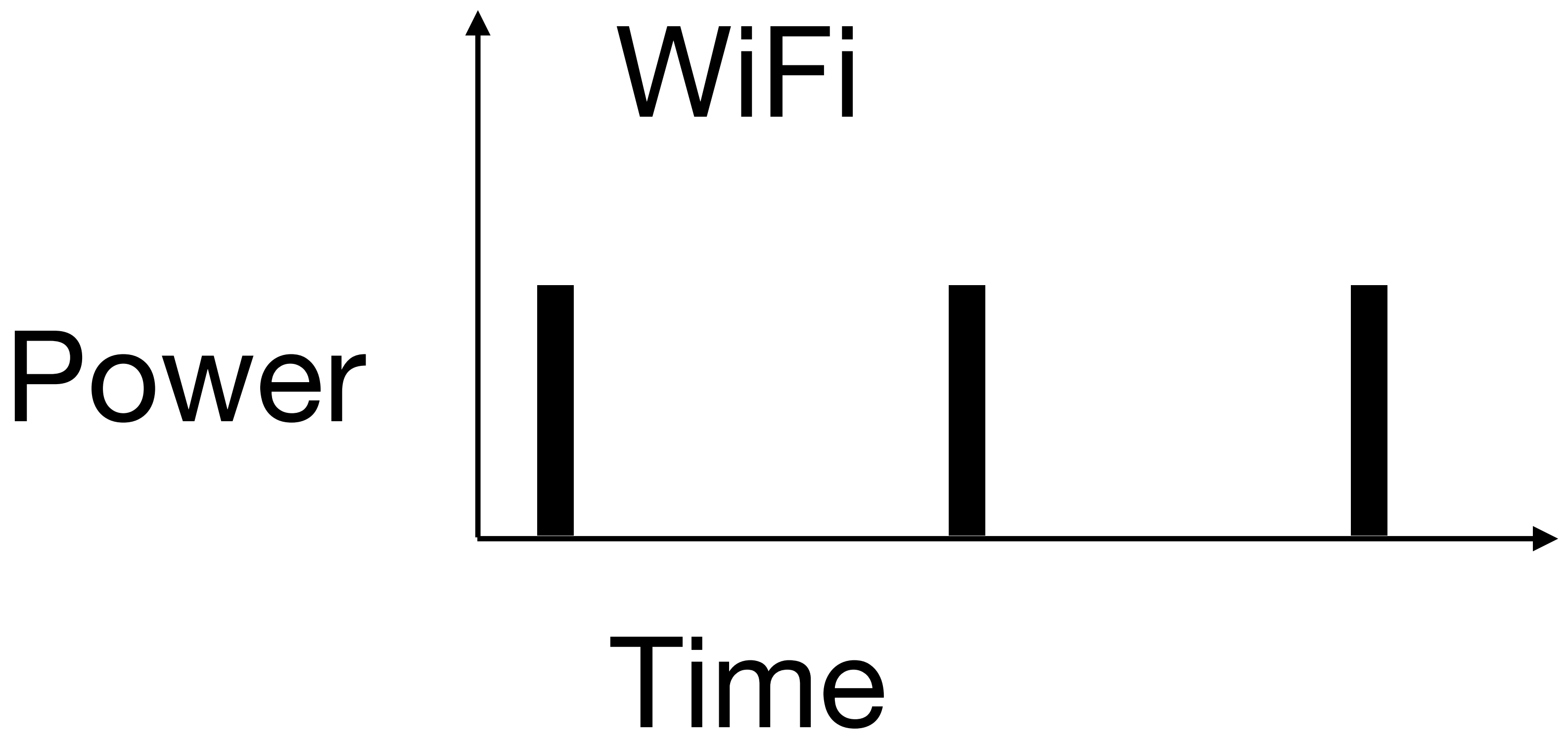
Hide device specific details in device driver

- Abstraction allows OS and applications to stay device-neutral

- Abstraction can hurt.





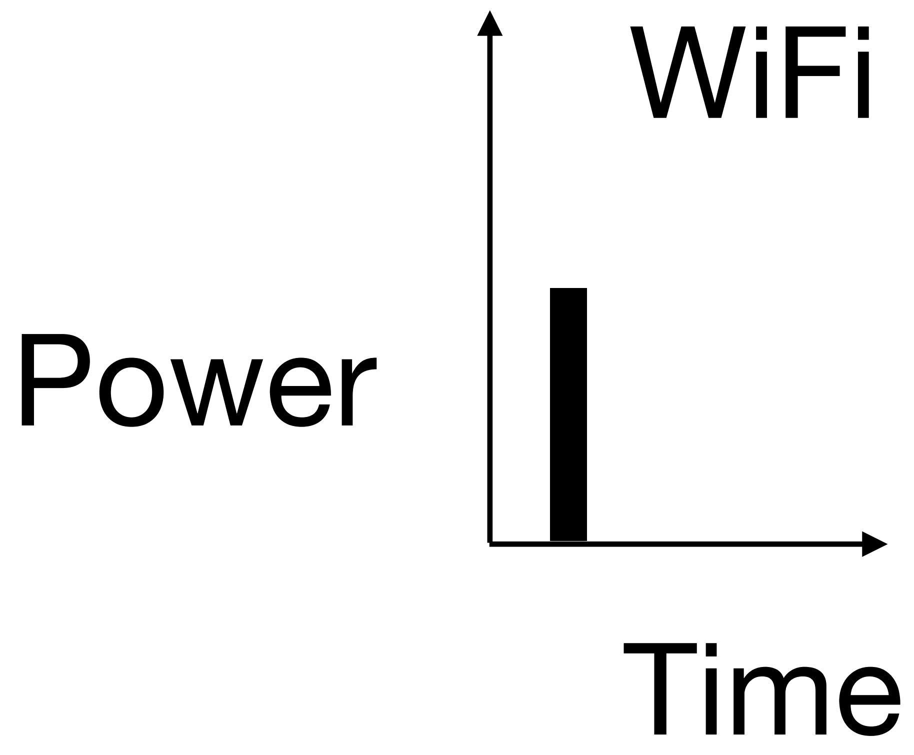


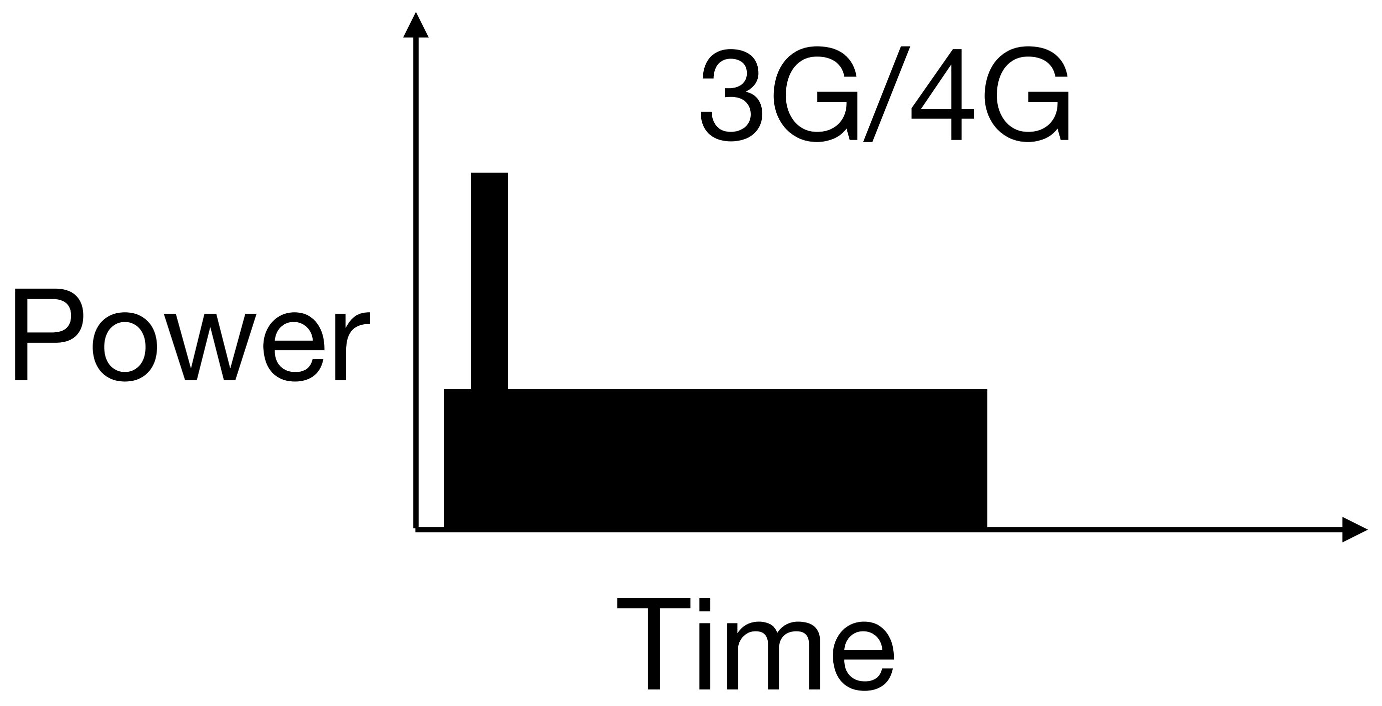
Fitting into the OS

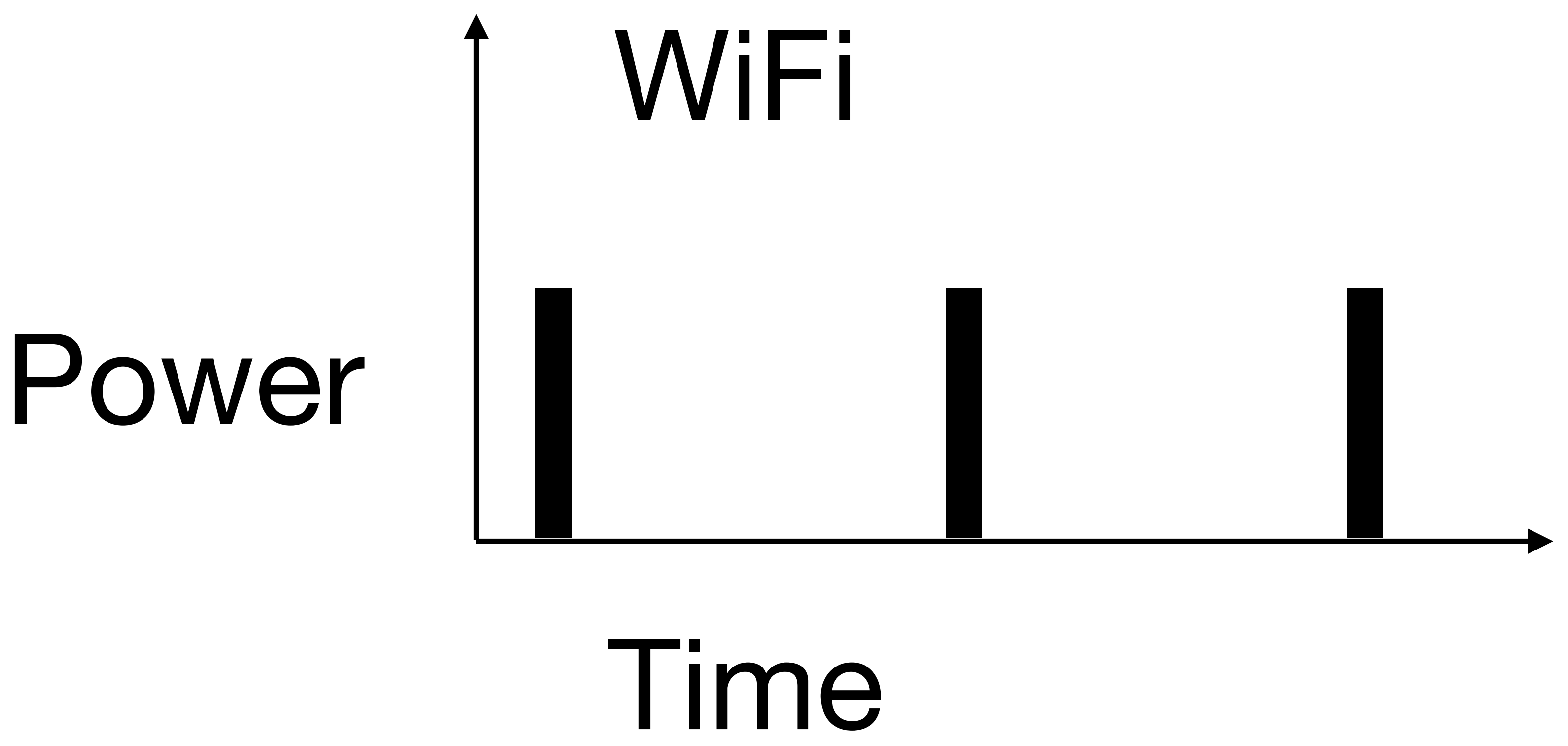
Hide device specific details in device driver

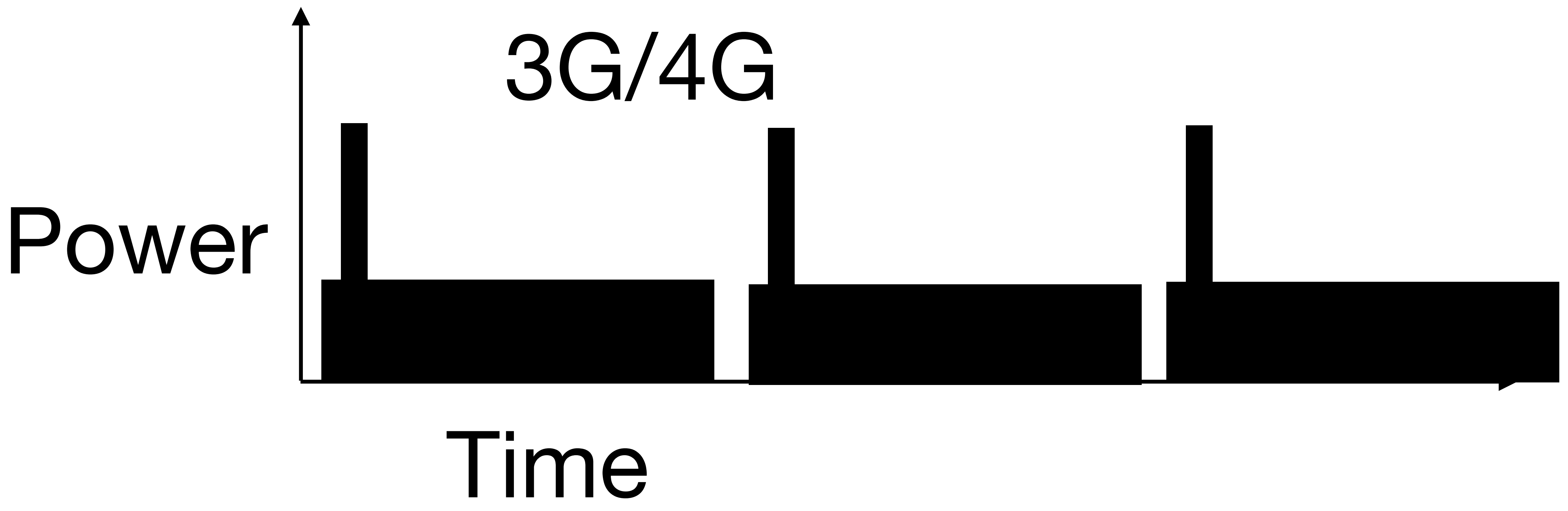
- Abstraction allows OS and applications to stay device-neutral

- Abstraction can hurt.





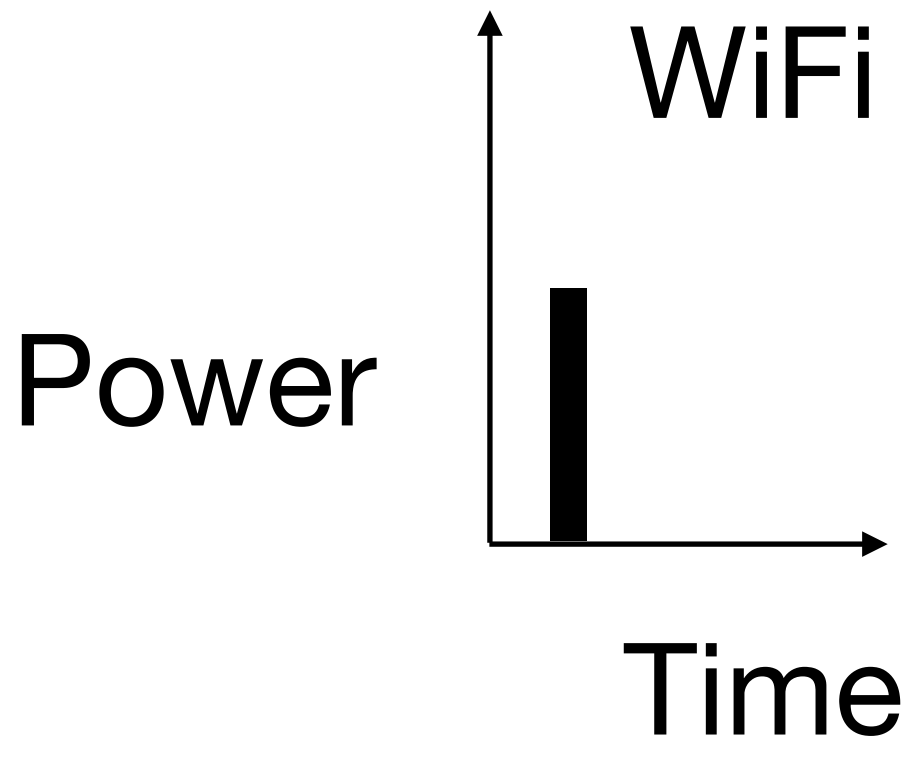


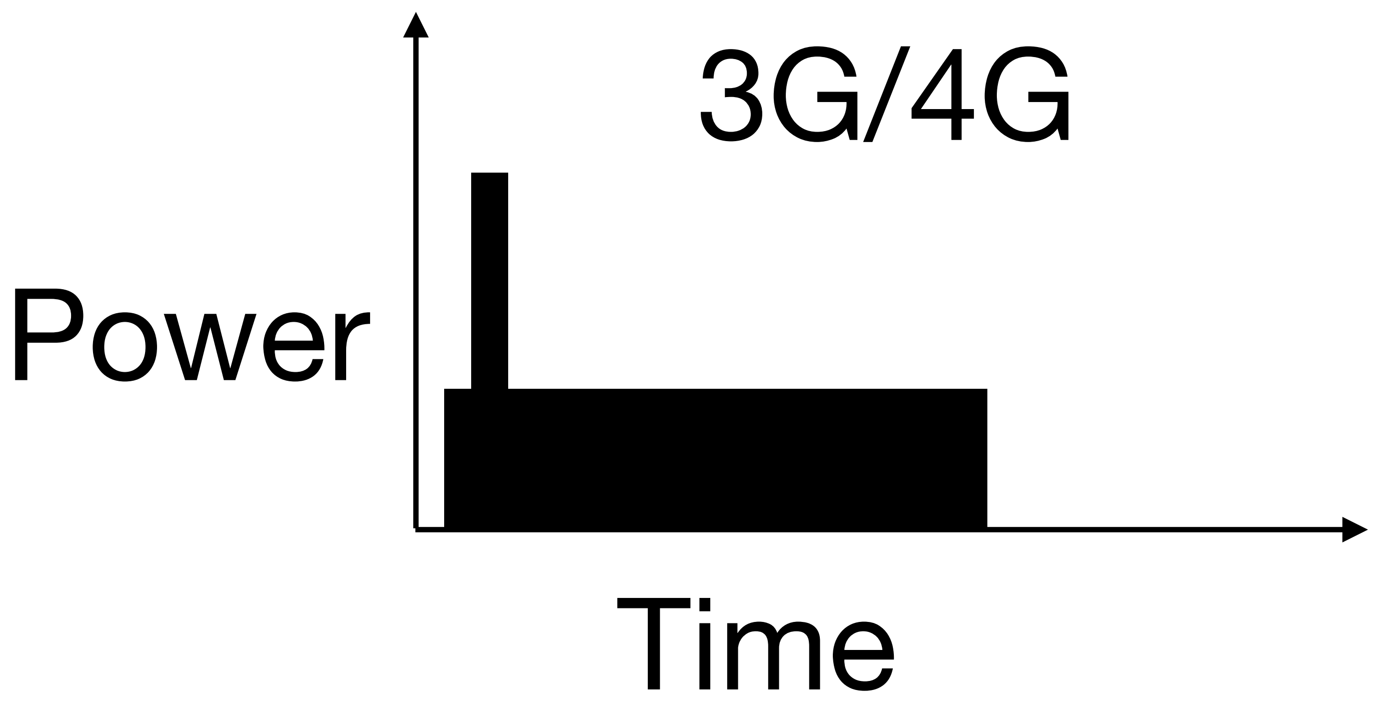


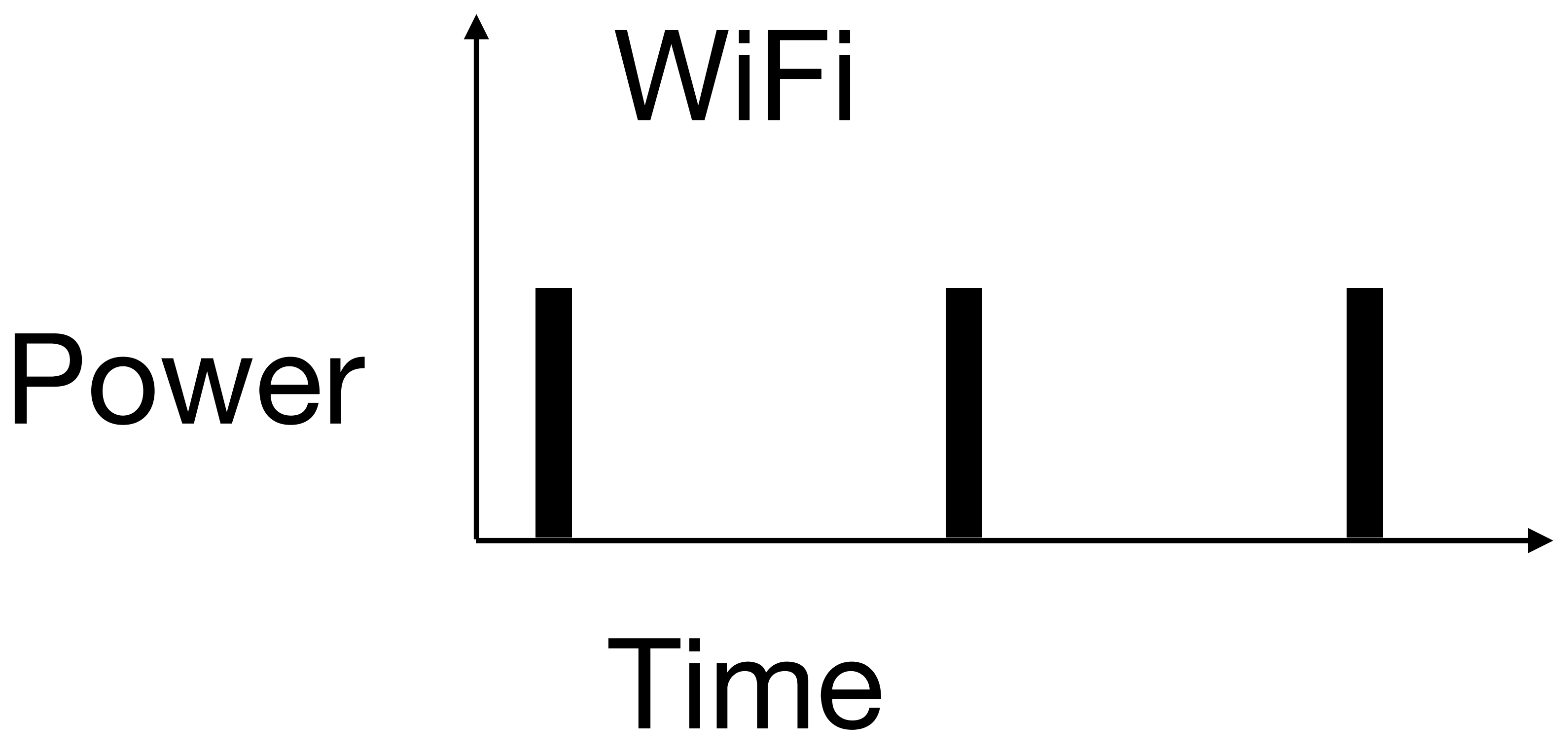
Fitting into the OS

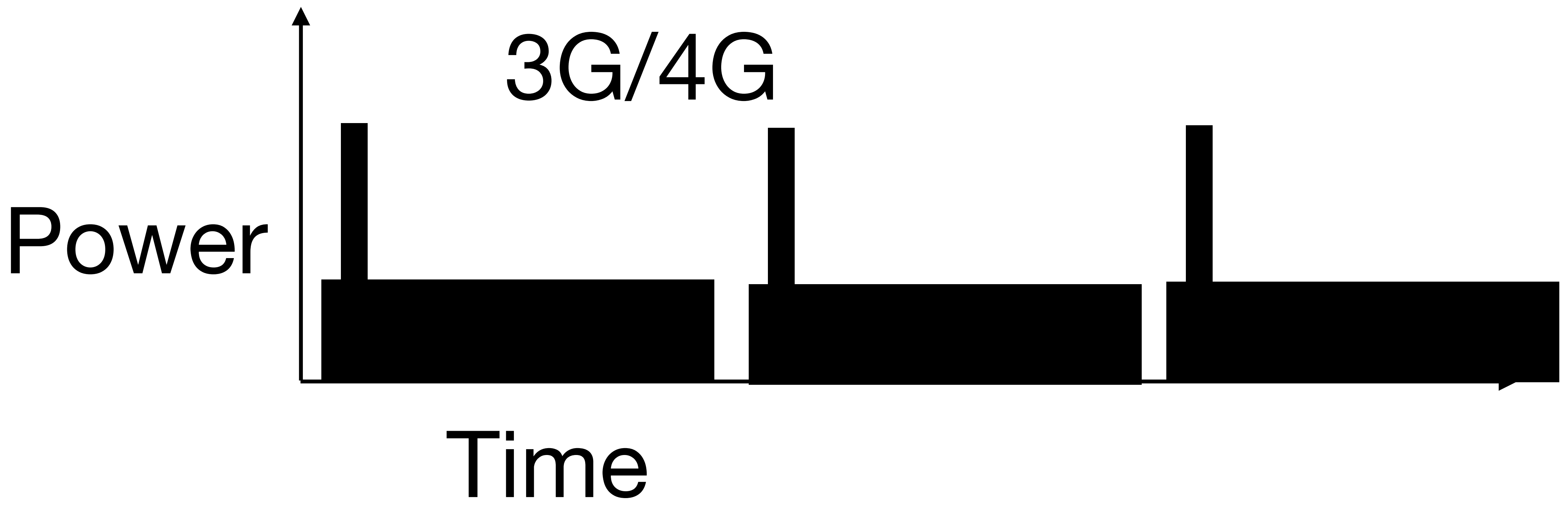
Hide device specific details in device driver

- Abstraction allows OS and applications to stay device-neutral
- Abstraction can hurt.
    - Example: 3G/4G are inefficient for small periodic pings





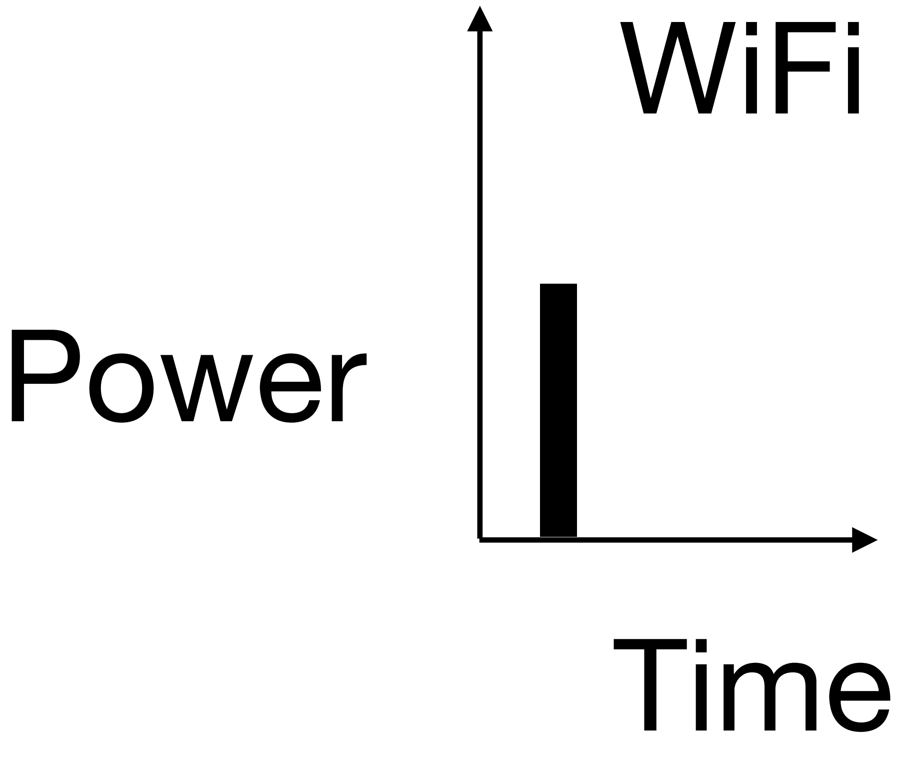


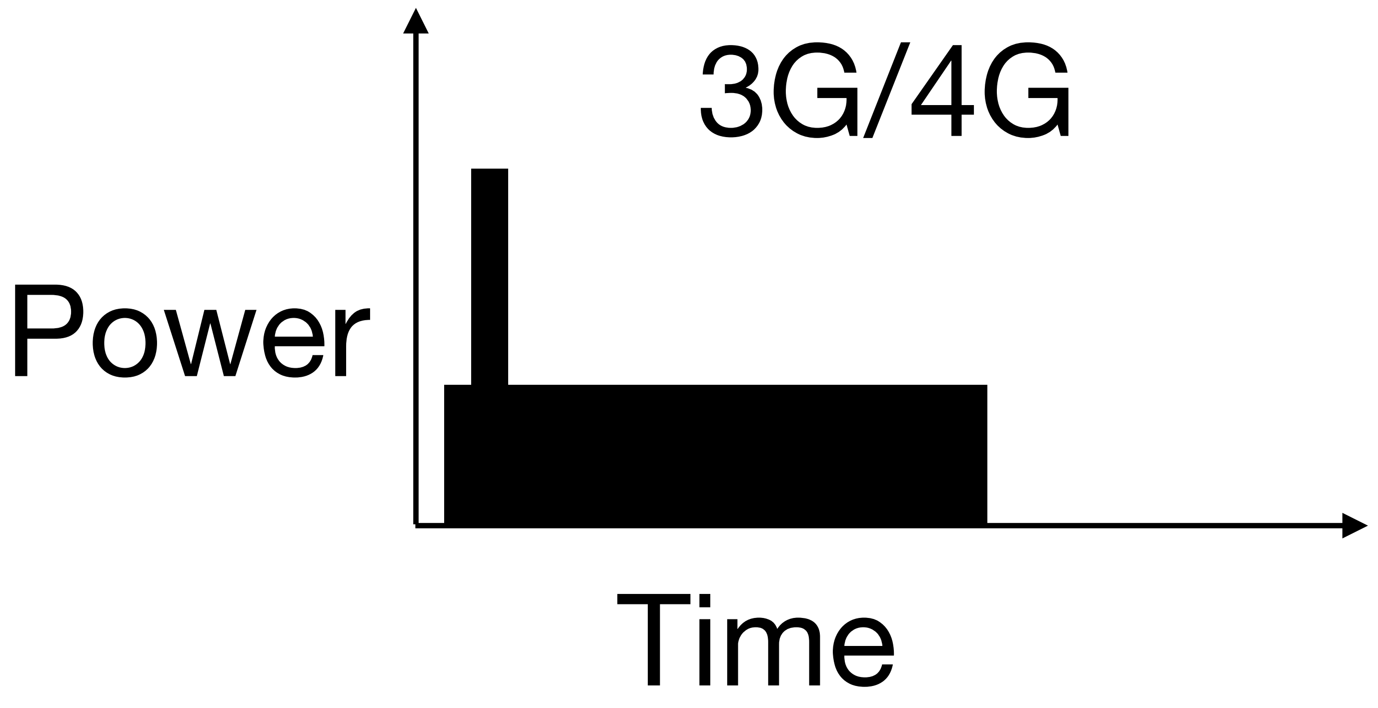


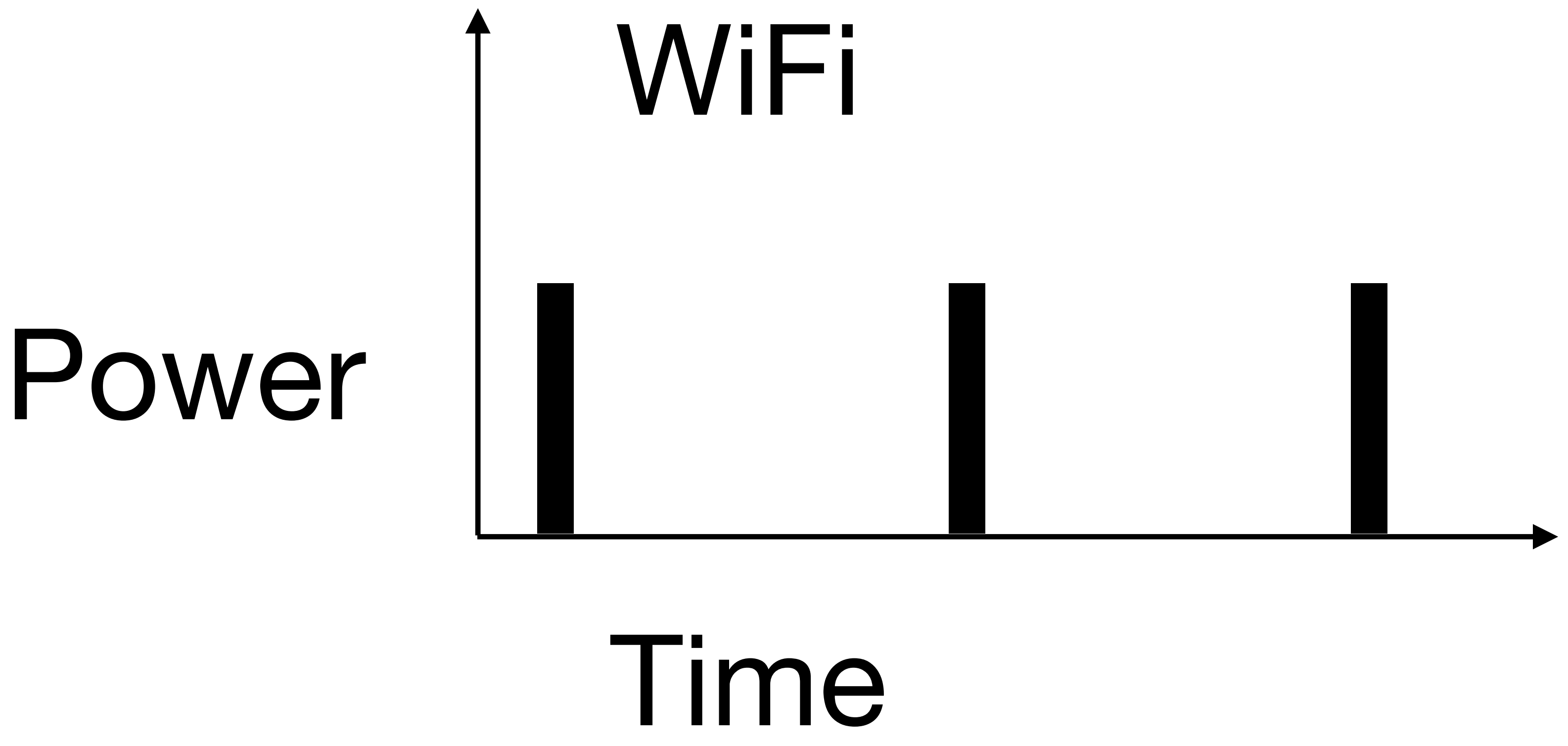
Fitting into the OS

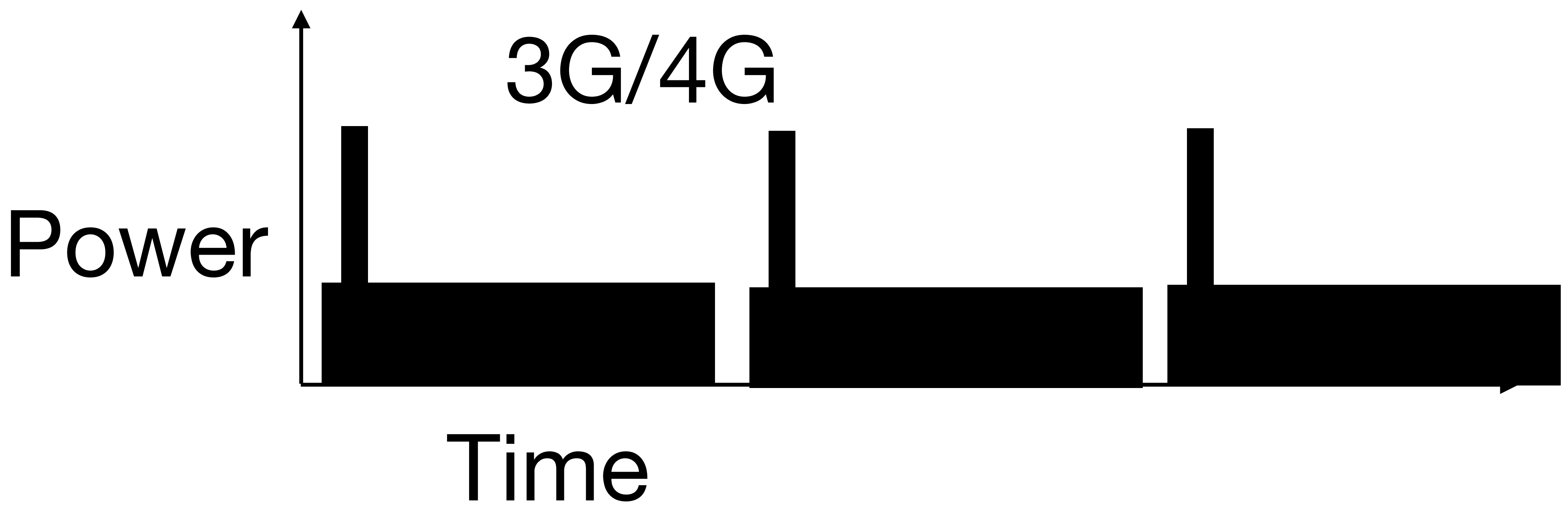
Hide device specific details in device driver

- 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

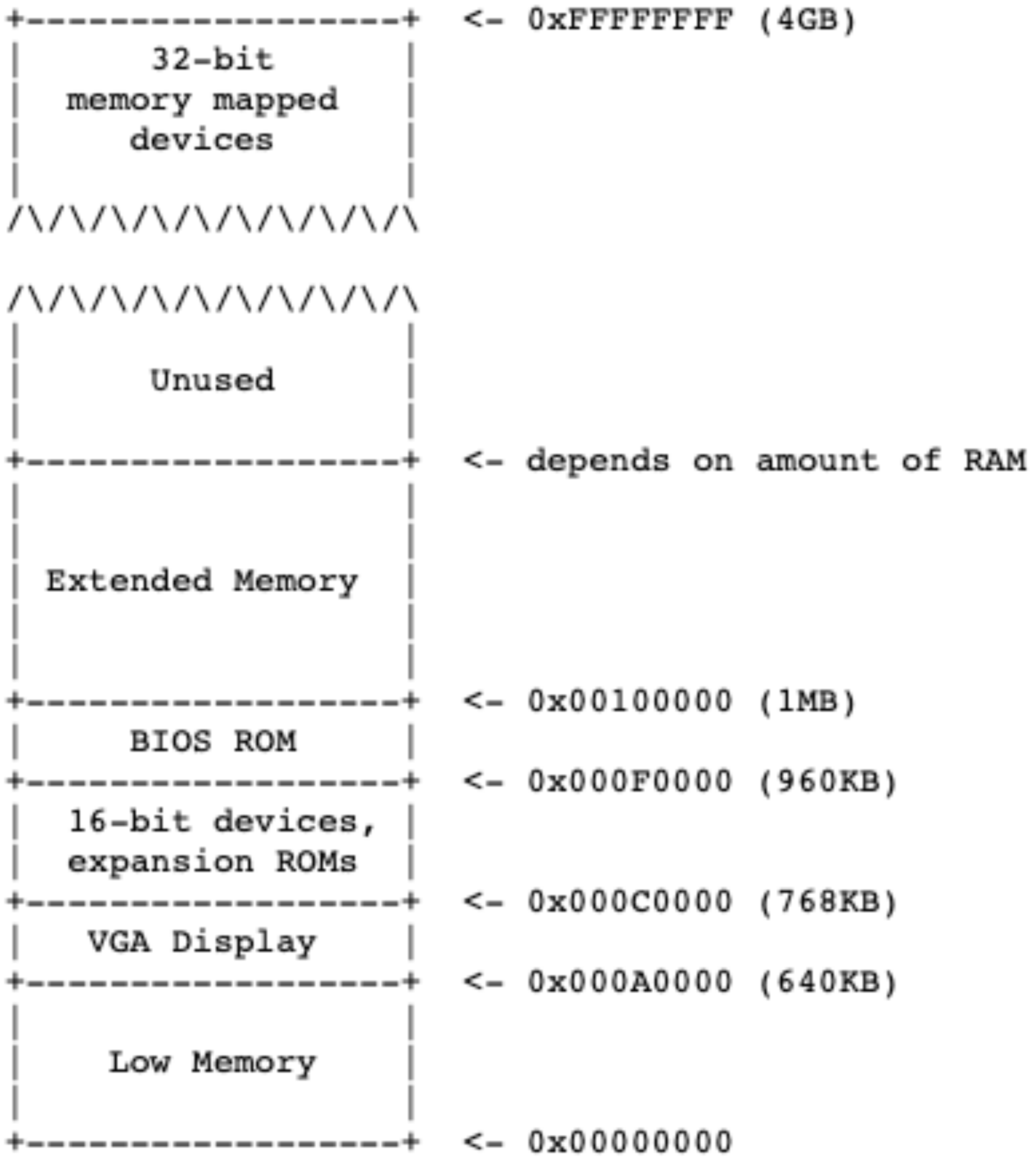








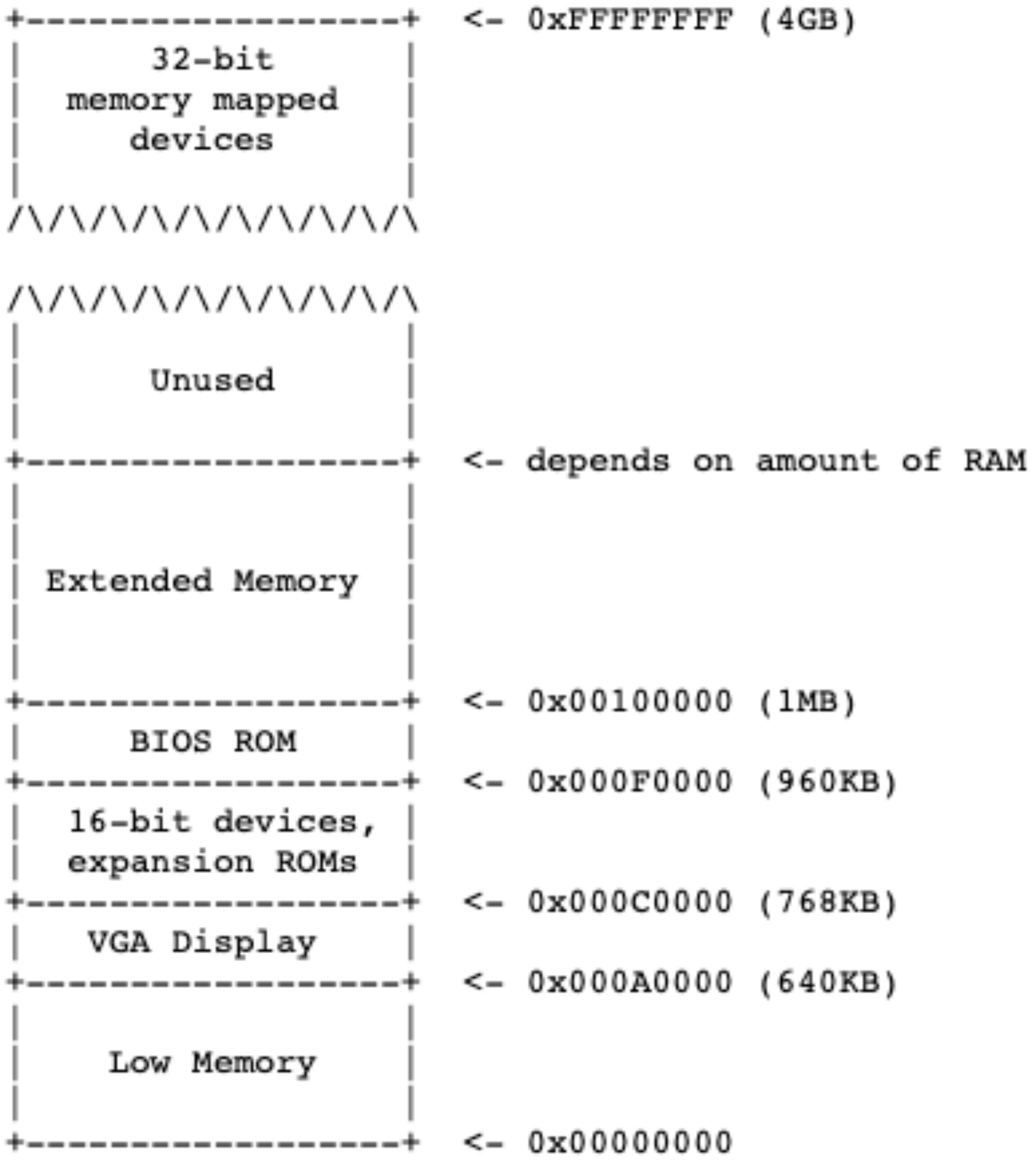
Memory-mapped IO and Port-mapped IO



```
+------------------+  <- 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
```

Memory-mapped IO and Port-mapped IO

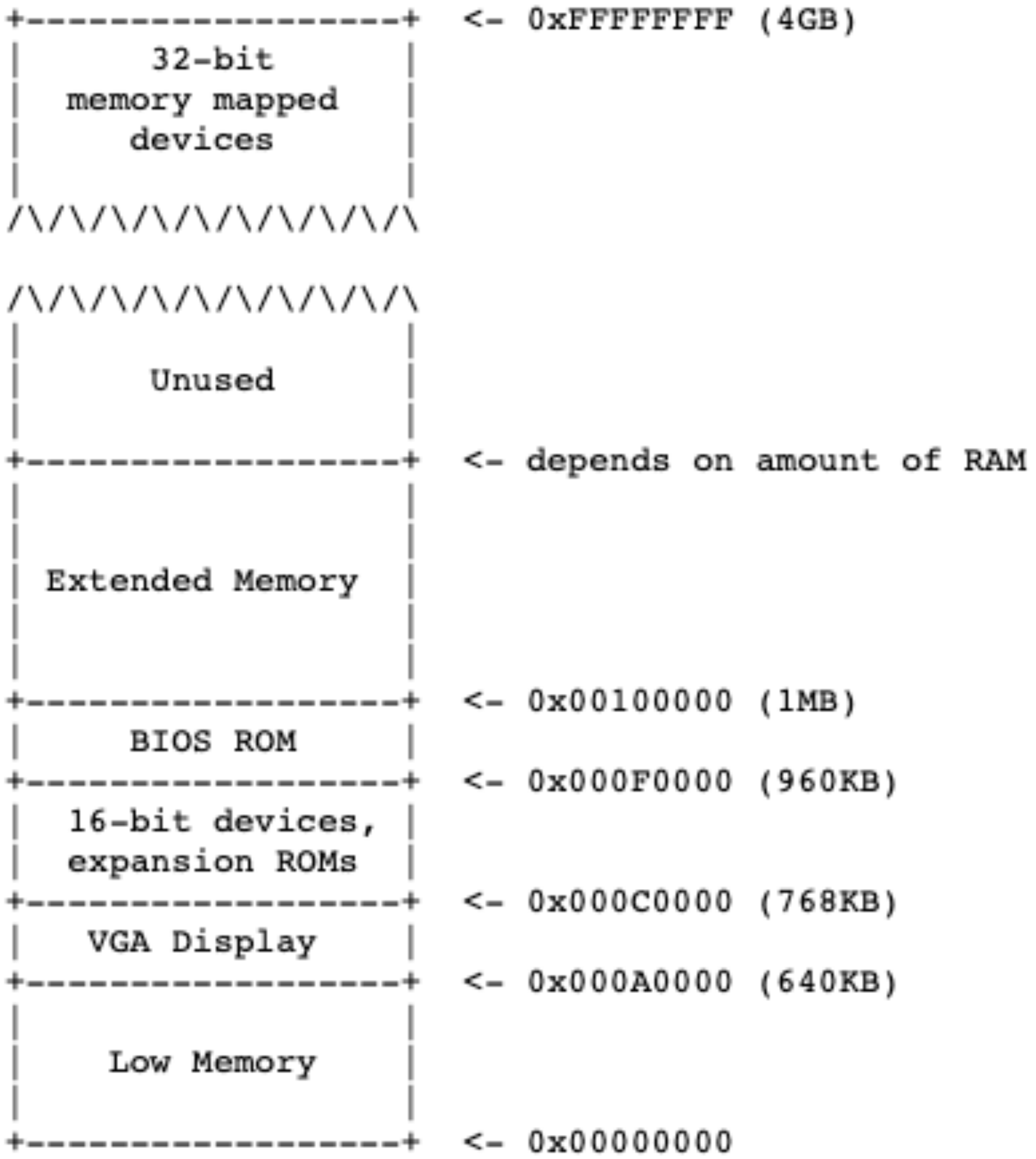
- Memory mapped:



```
+------------------+  <- 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
```

Memory-mapped IO and Port-mapped IO

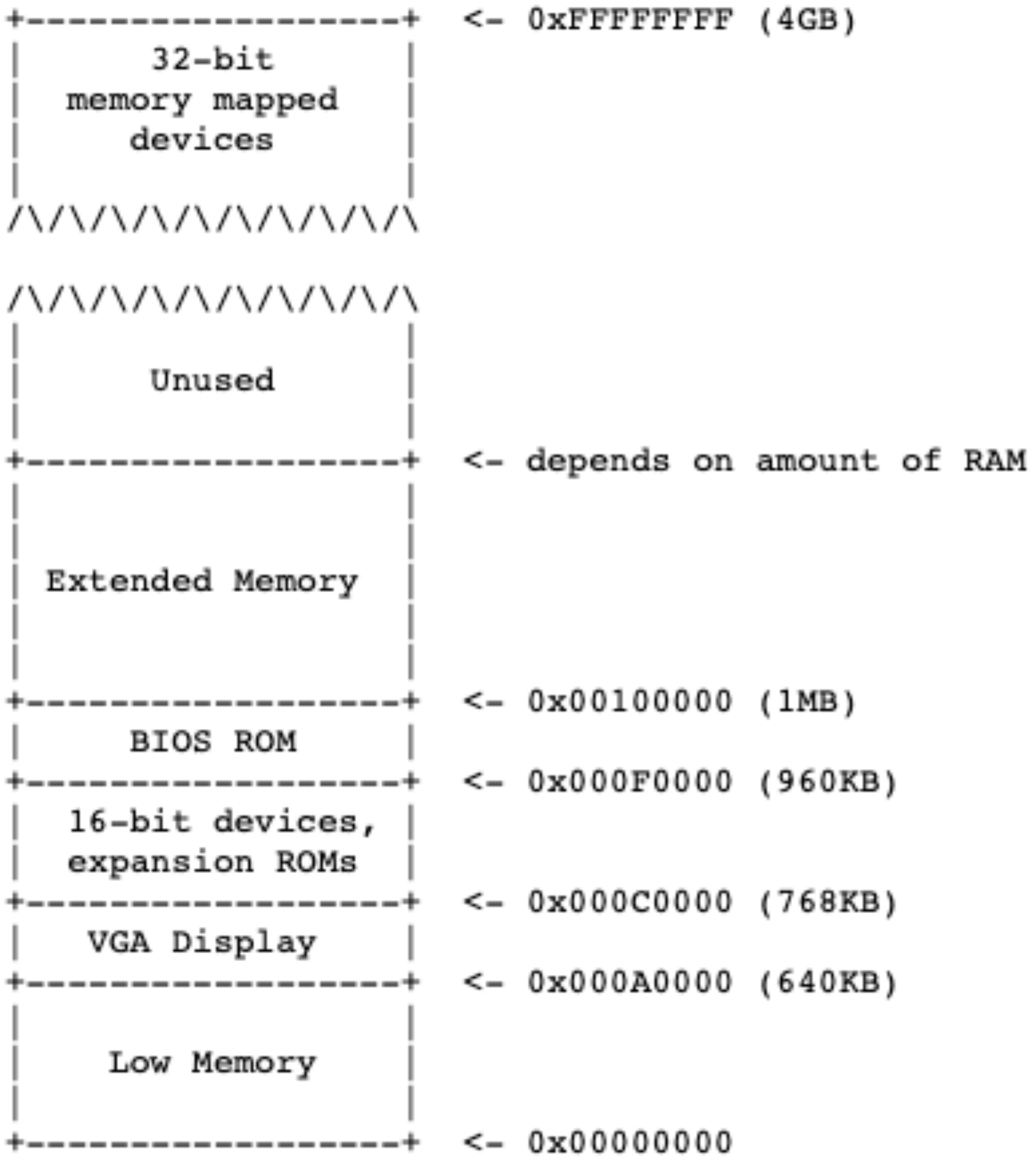
- Memory mapped:
  - Regular memory access instructions



```
+------------------+  <- 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
```

Memory-mapped IO and Port-mapped IO

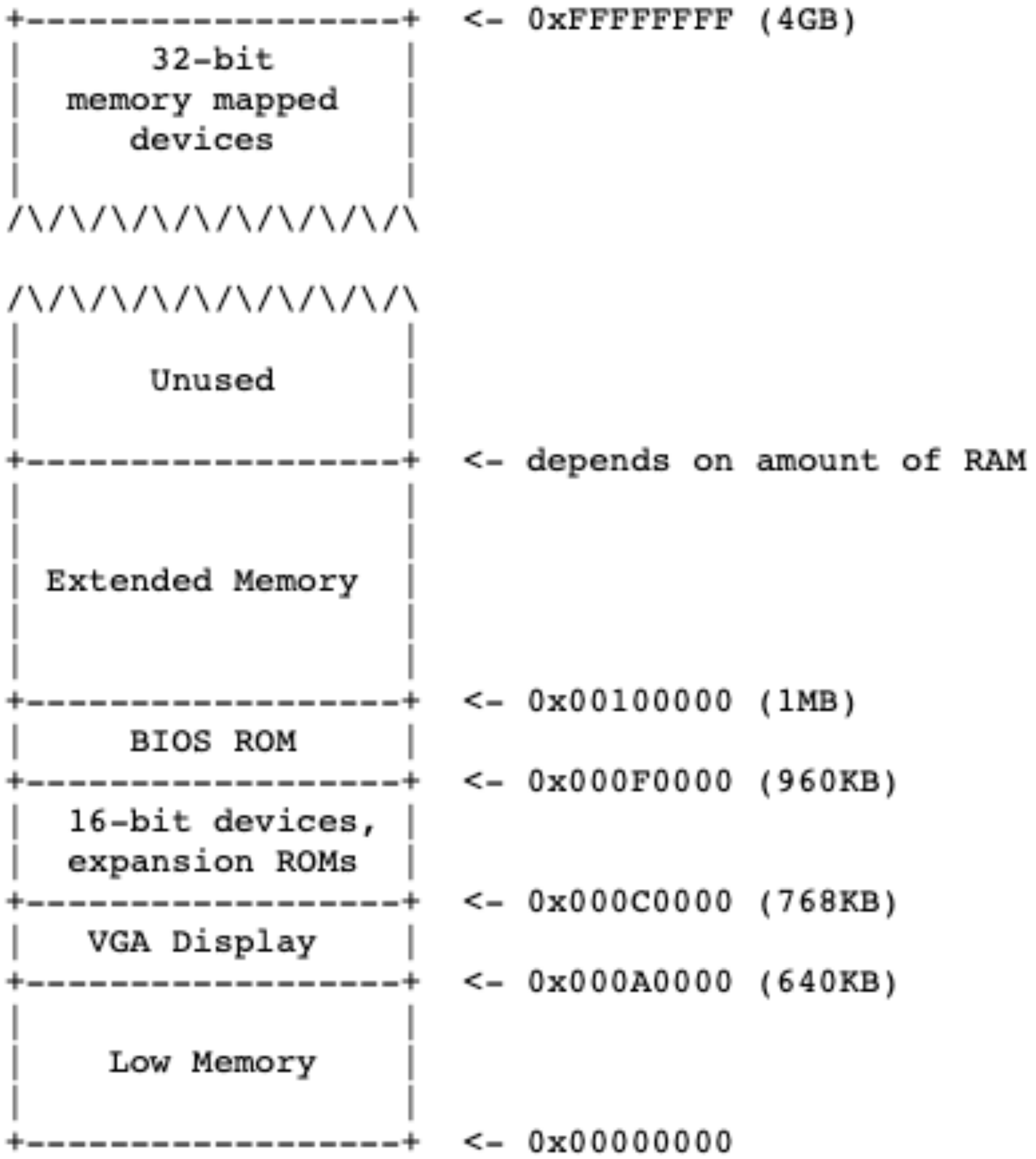
- Memory mapped:
  - Regular memory access instructions
  - Reads and writes are routed to appropriate device



```
+------------------+  <- 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
```

Memory-mapped IO and Port-mapped IO

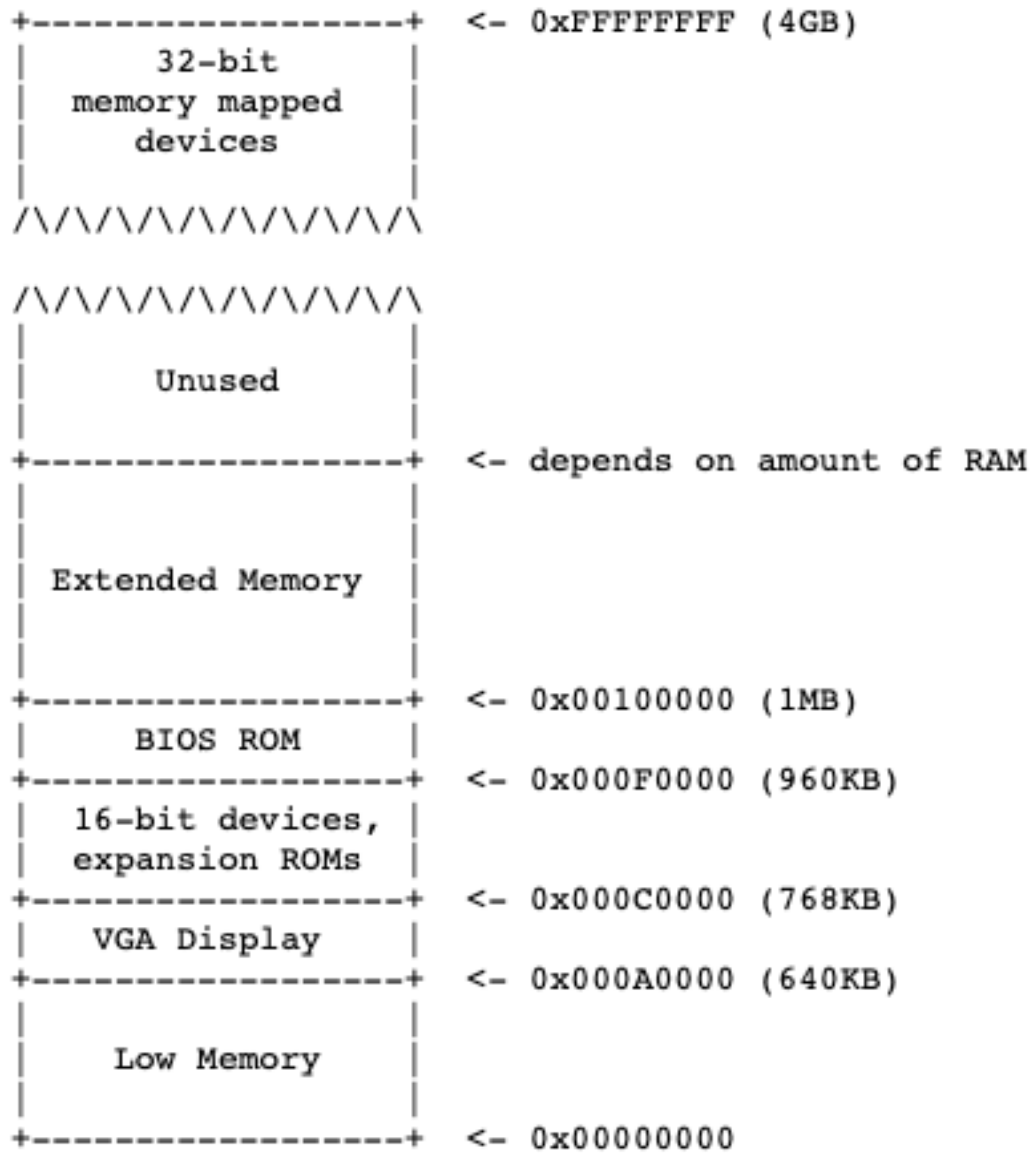
- Memory mapped:
  - Regular memory access instructions
  - Reads and writes are routed to appropriate device
  - Does not behave like memory! Reading same location twice can change due to external events



```
+------------------+  <- 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
```

Memory-mapped IO and Port-mapped IO

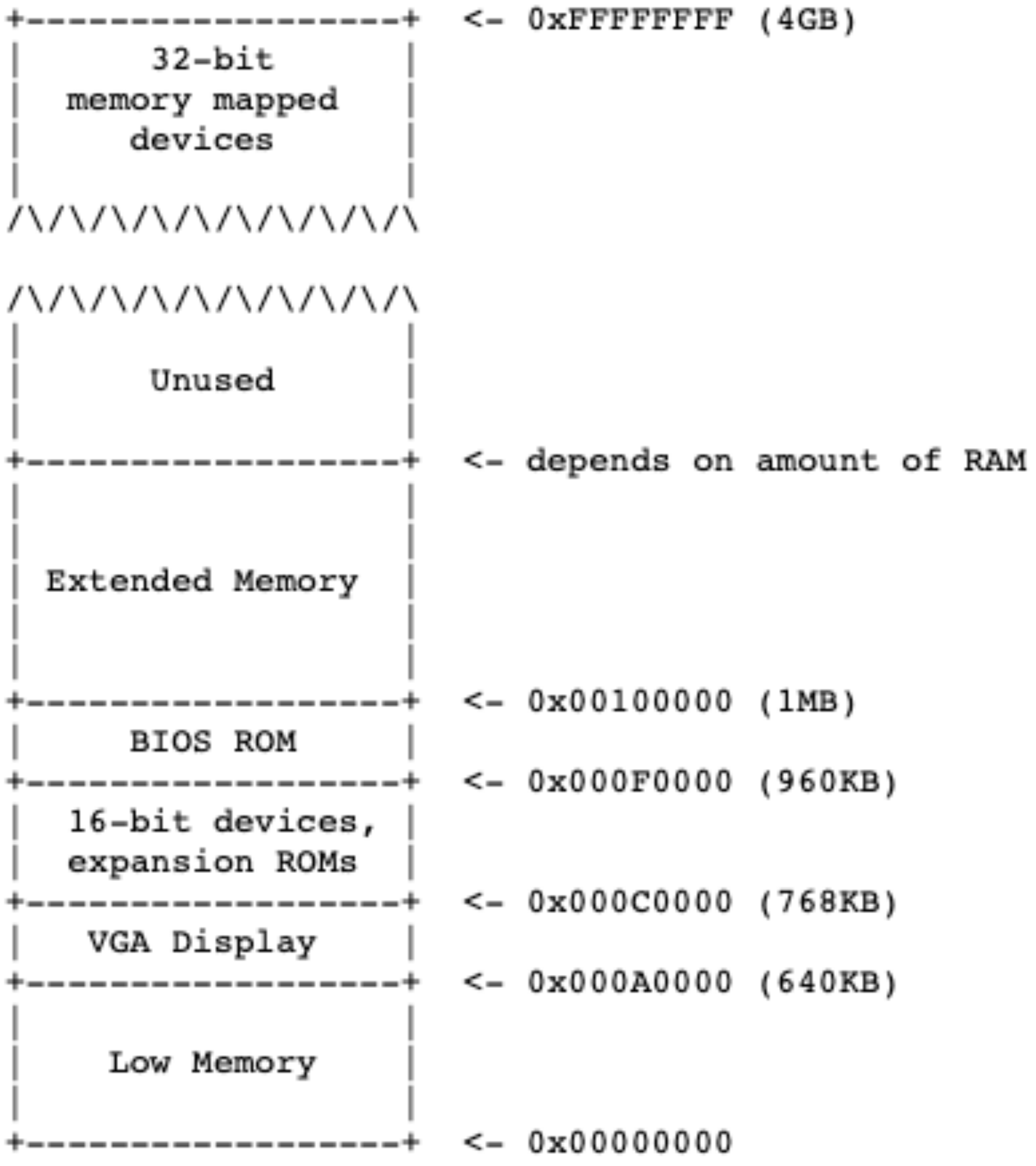
- Memory mapped:
    - Regular memory access instructions
    - Reads and writes are routed to appropriate device
    - Does not behave like memory! Reading same location twice can change due to external events
- Port mapped:



```
+------------------+  <- 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
```

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
- 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

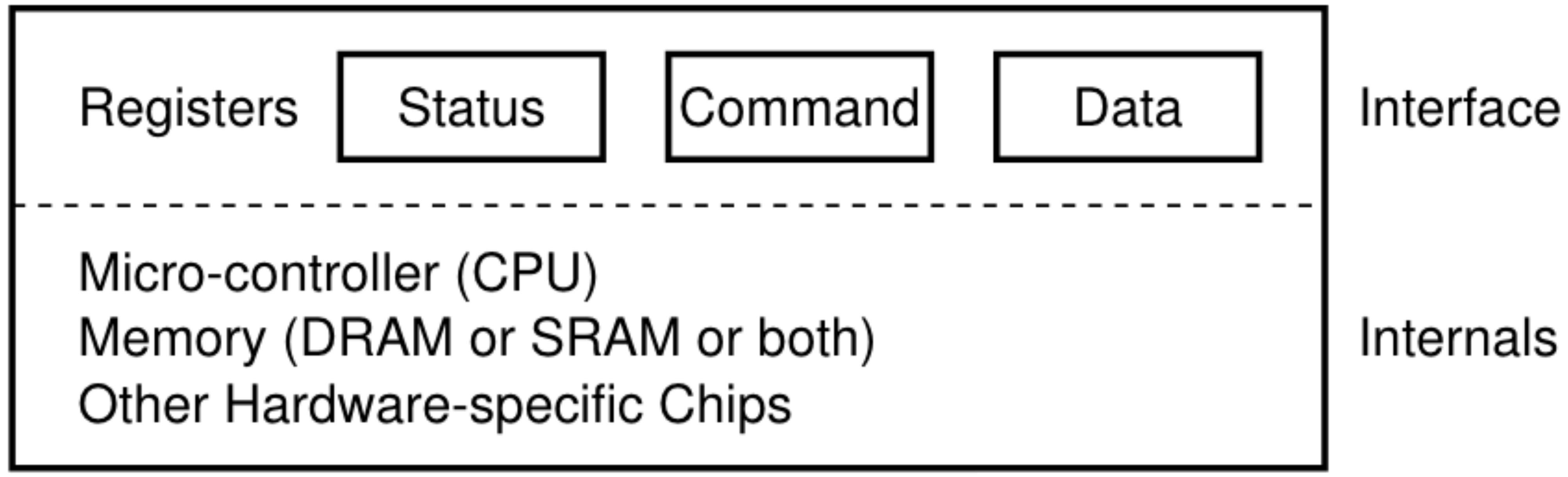


Figure 36.3: **A Canonical Device**

Canonical protocol

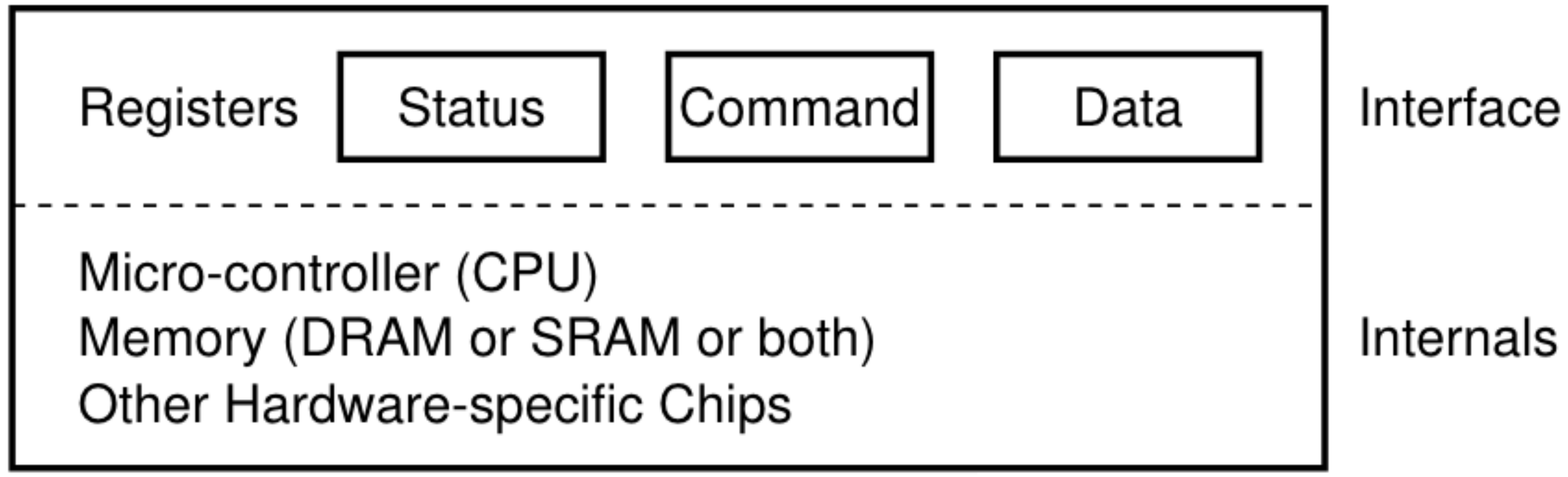


Figure 36.3: **A Canonical Device**

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);
}
```

Canonical protocol

- Poll device until it is ready

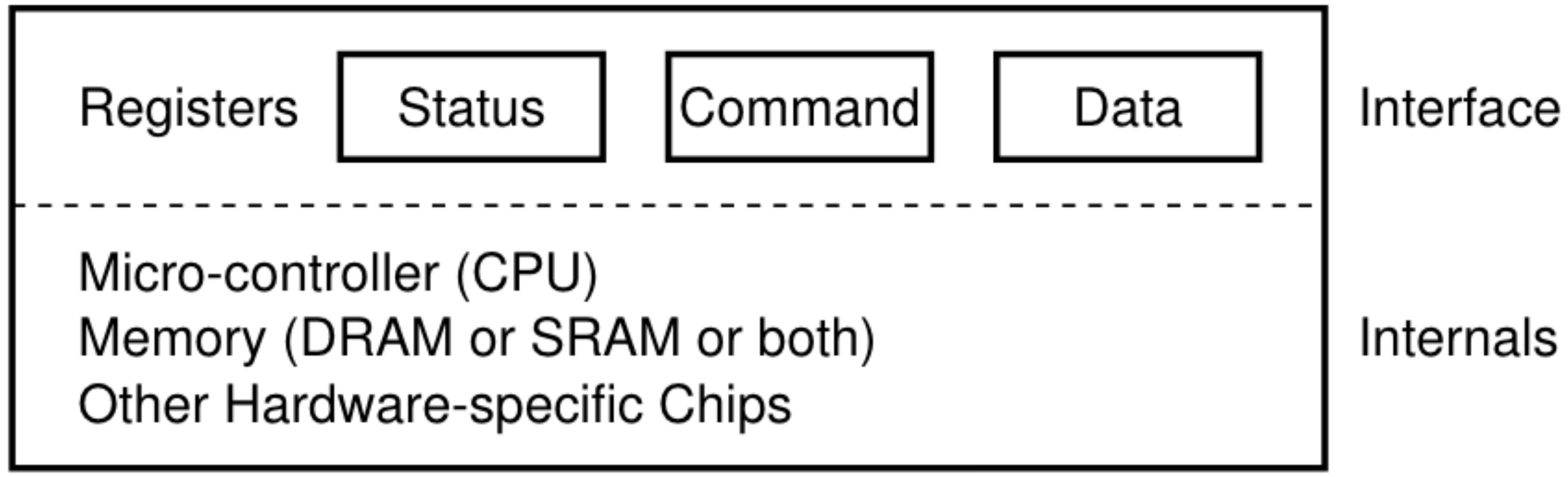


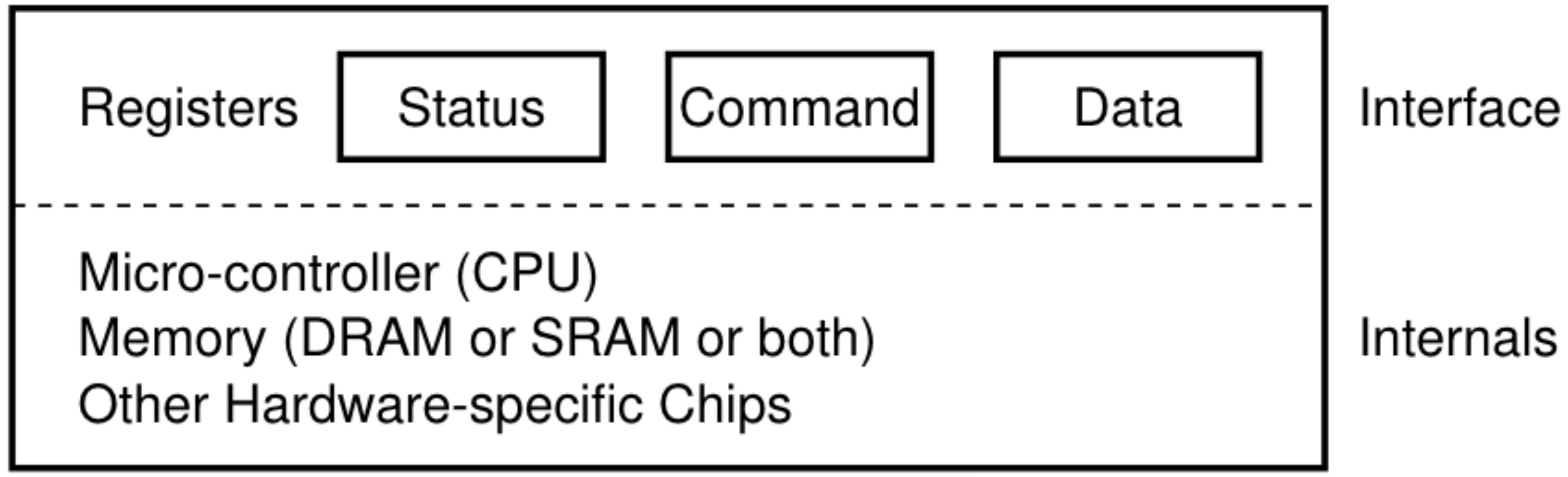
Figure 36.3: **A Canonical Device**

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);
}
```

Canonical protocol

- Poll device until it is ready
- CPU cannot do anything else.



Registers

Status

Command

Data

Interface

Micro-controller (CPU)

Memory (DRAM or SRAM or both)

Other Hardware-specific Chips

Internals

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);
}
```

Canonical protocol

- Poll device until it is ready
- CPU cannot do anything else.
- Example: CPU needs to spend ~1 million instructions waiting for disk

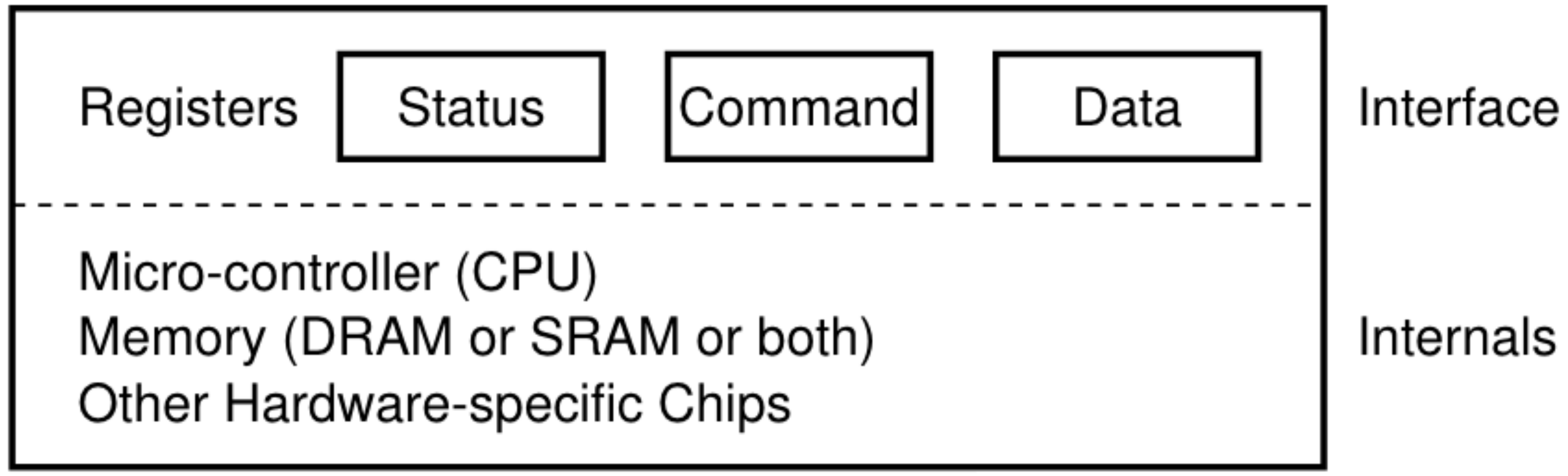


Figure 36.3: **A Canonical Device**

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);
}
```

Canonical protocol

- 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.

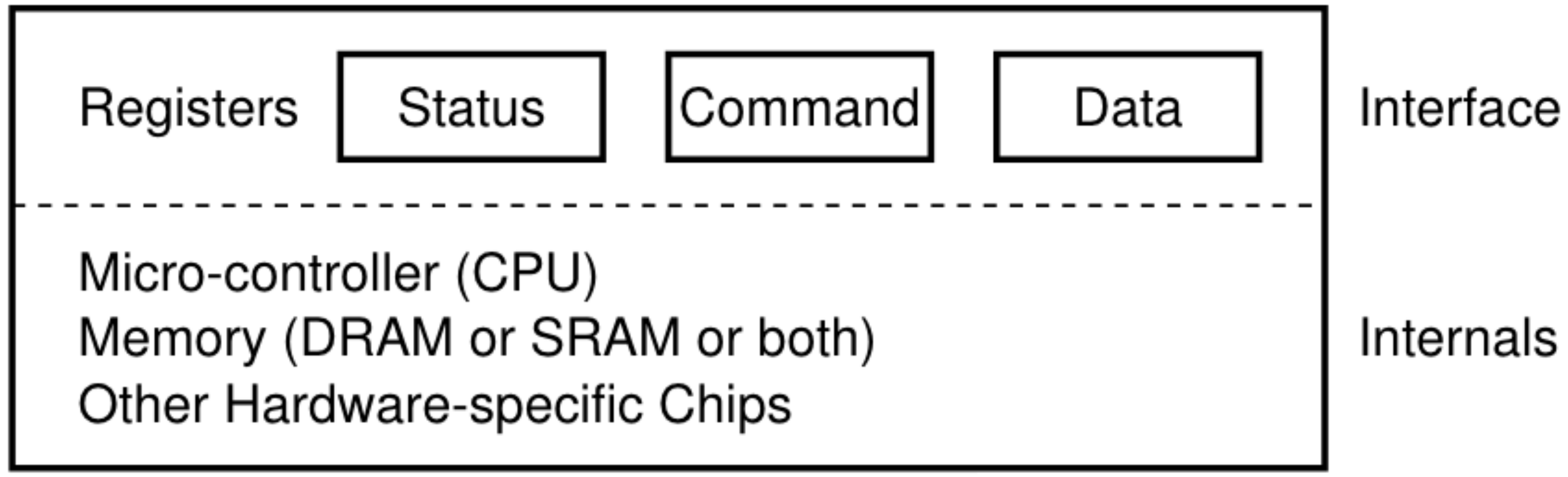


Figure 36.3: **A Canonical Device**

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);
}
```

Canonical protocol

- 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.

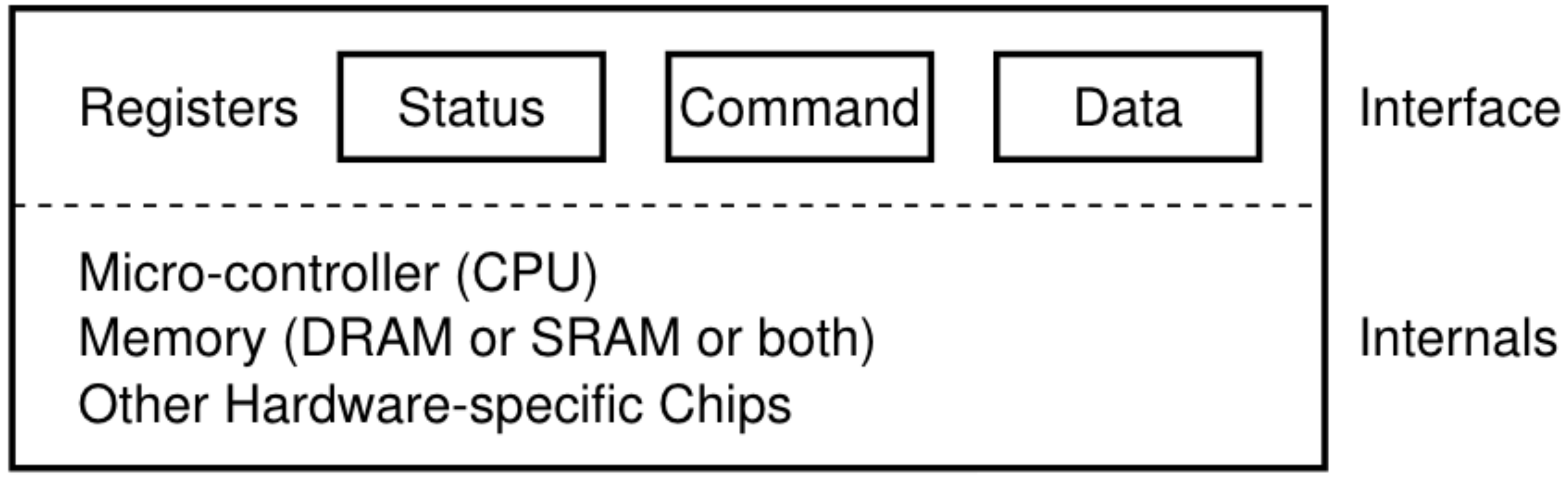
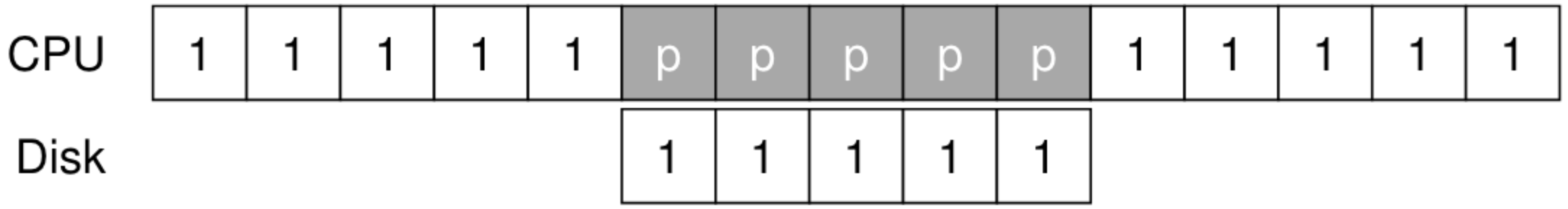


Figure 36.3: **A Canonical Device**

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);
}
```

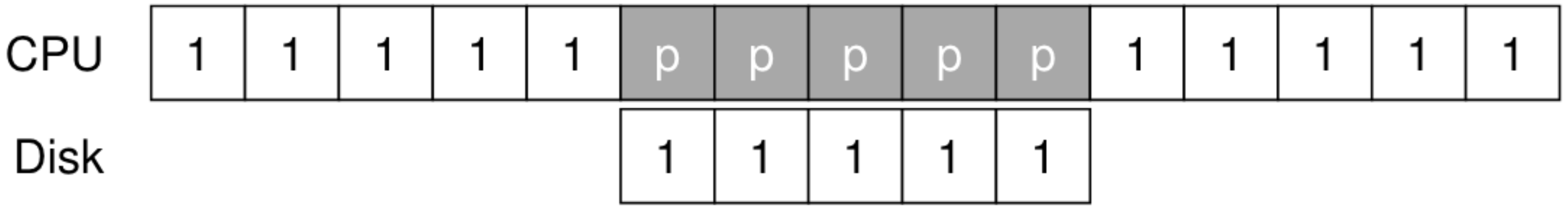
Lowering CPU overheads with interrupts



CPU	1	1	1	1	1	p	p	p	p	p	1	1	1	1	1
Disk						1	1	1	1	1					

Lowering CPU overheads with interrupts

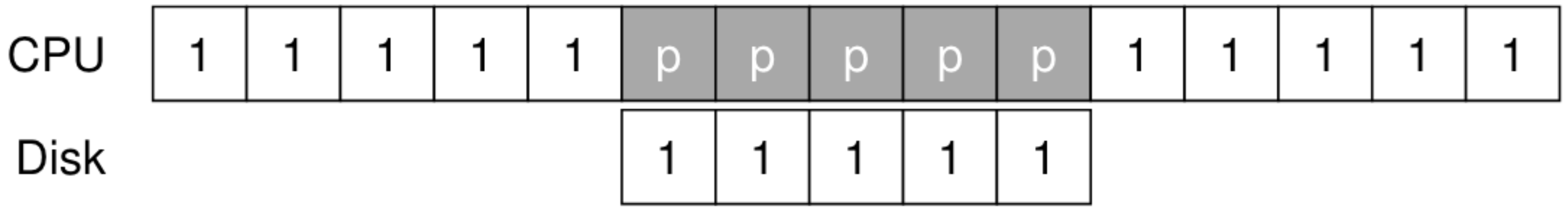
- Device sends an interrupt that it is ready



CPU	1	1	1	1	1	p	p	p	p	p	1	1	1	1	1
Disk						1	1	1	1	1					

Lowering CPU overheads with interrupts

- Device sends an interrupt that it is ready
- CPU runs another process in the meantime

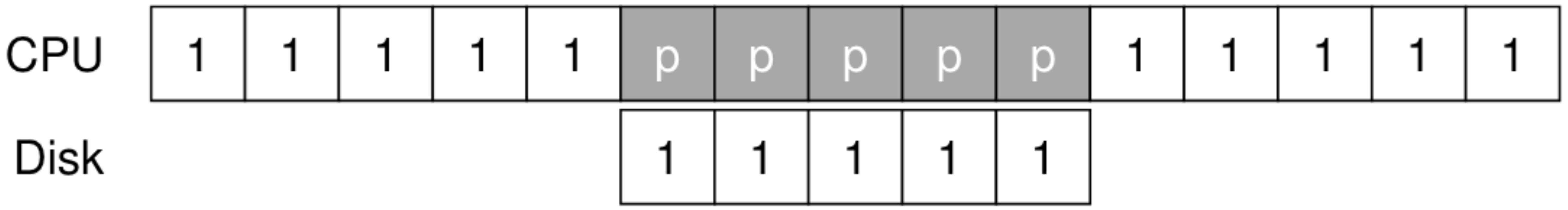


CPU	1	1	1	1	1	p	p	p	p	p	1	1	1	1	1
Disk						1	1	1	1	1					

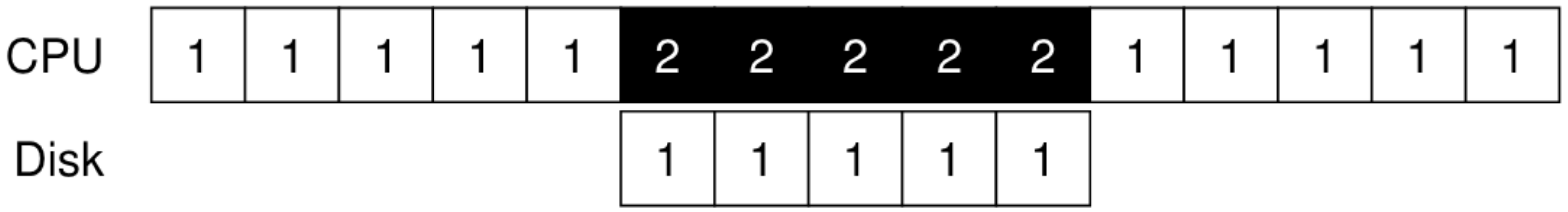
Lowering CPU overheads with interrupts

- Device sends an interrupt that it is ready

- CPU runs another process in the meantime



CPU	1	1	1	1	1	p	p	p	p	p	1	1	1	1	1
Disk						1	1	1	1	1					



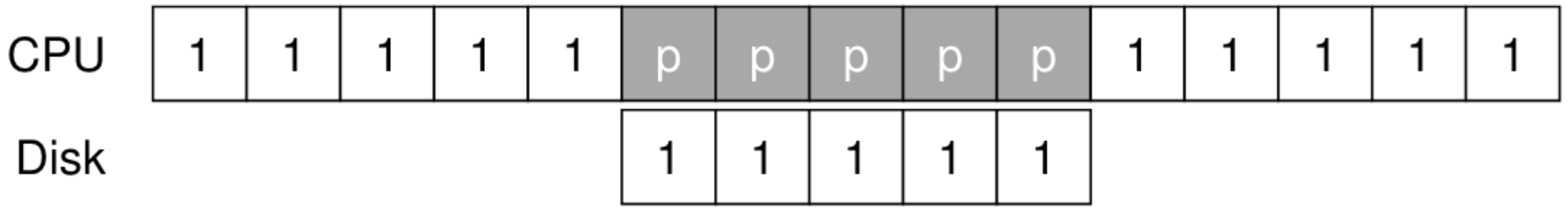
CPU	1	1	1	1	1	2	2	2	2	2	1	1	1	1	1
Disk						1	1	1	1	1					

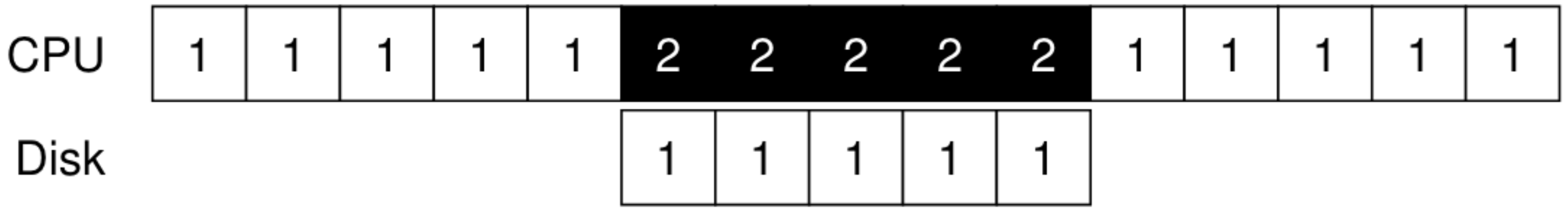
Lowering CPU overheads with interrupts

- Device sends an interrupt that it is ready

- CPU runs another process in the meantime

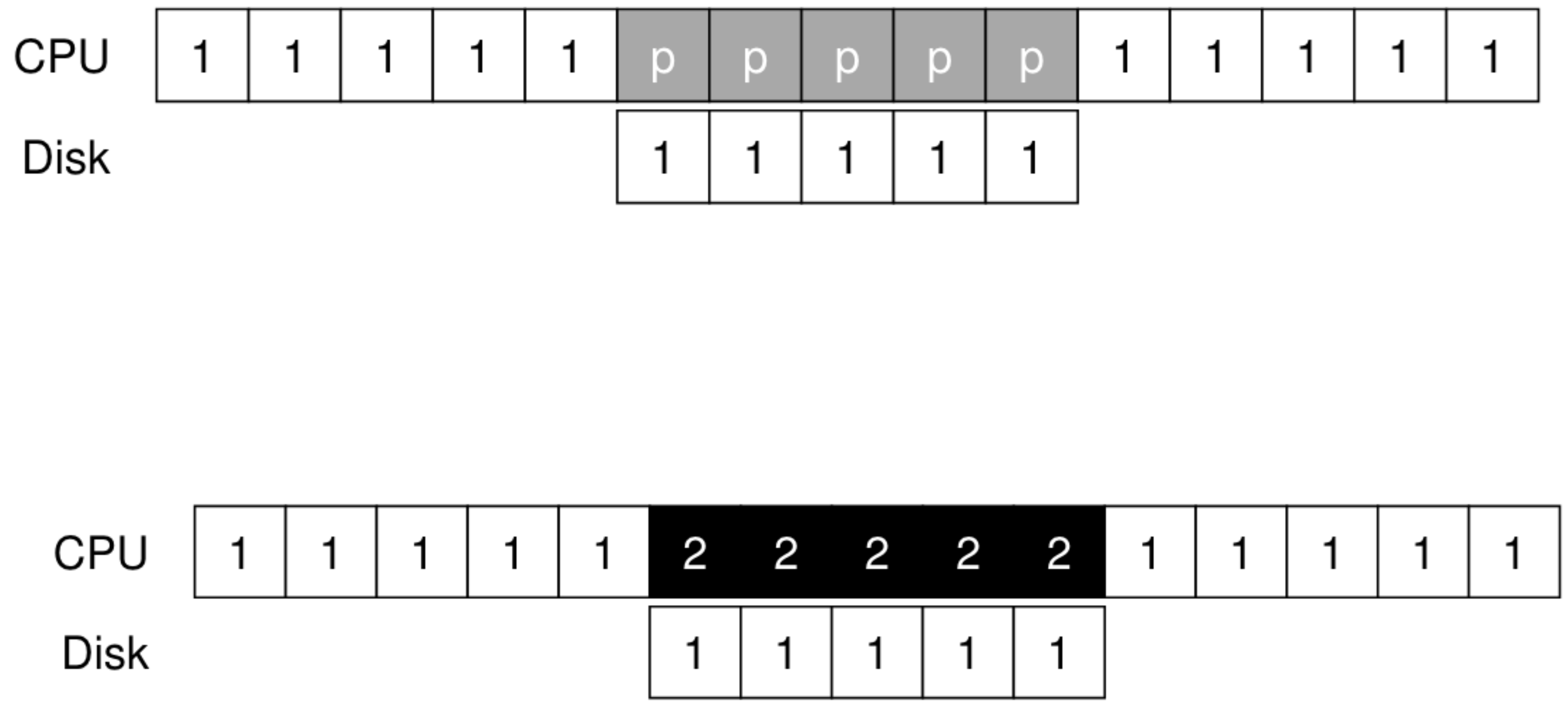
- Better CPU utilisation





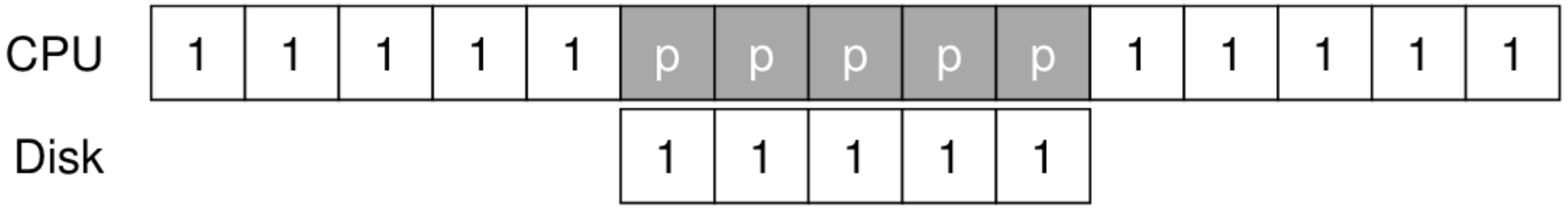
Lowering CPU overheads with interrupts

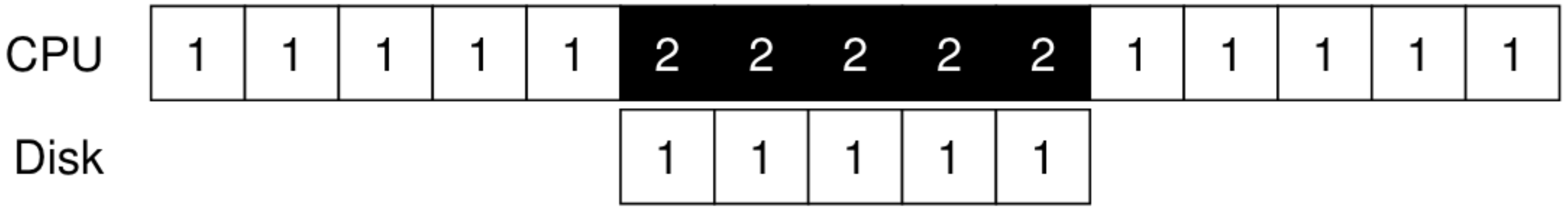
- Device sends an interrupt that it is ready
- CPU runs another process in the meantime
- Better CPU utilisation
- Not a good idea if device is fast.



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)

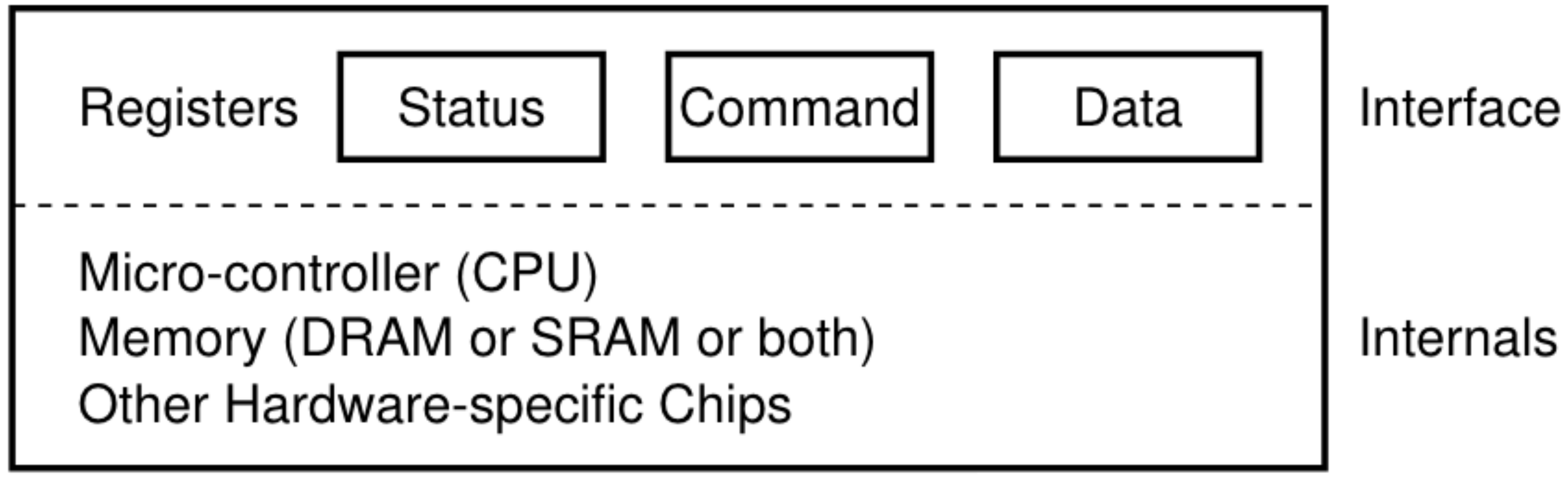


Figure 36.3: **A Canonical Device**

More efficient data movement

Direct Memory Access (DMA)

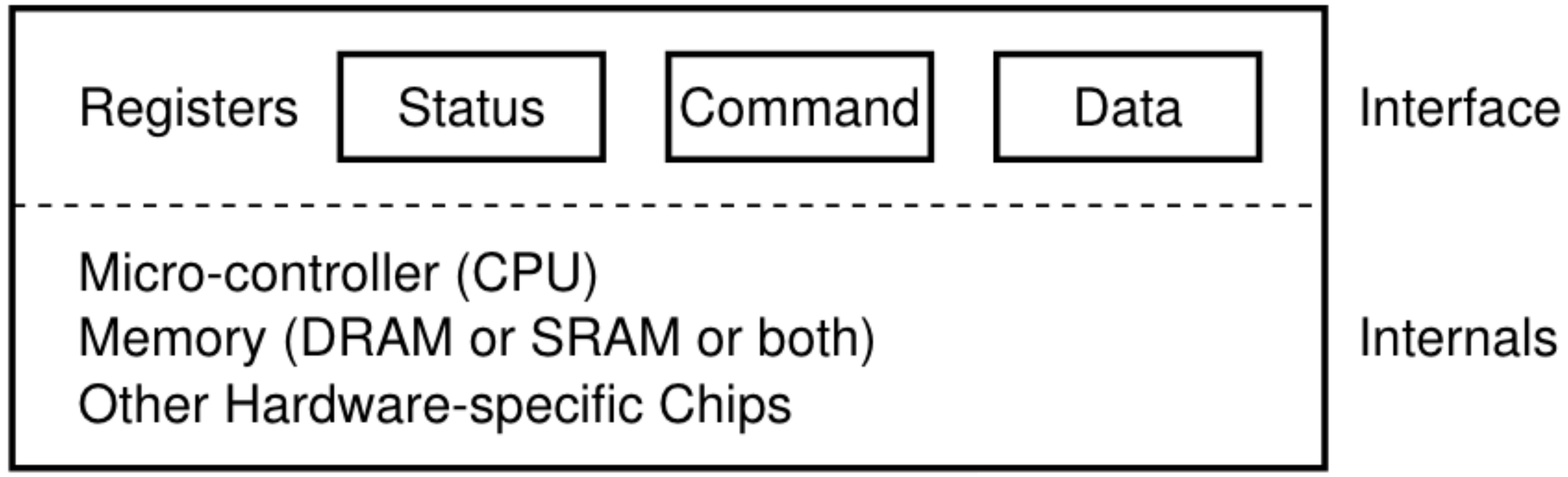


Figure 36.3: **A Canonical Device**

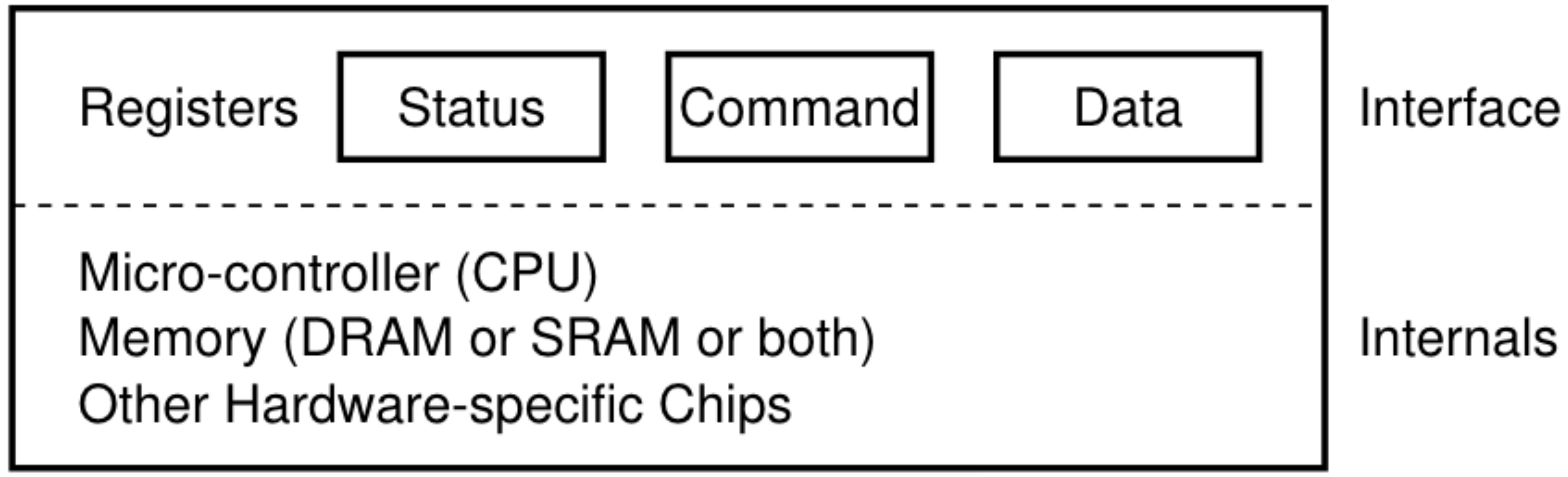
bootmain.c

```
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);
}
```

More efficient data movement

Direct Memory Access (DMA)

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



Registers Status Command Data

Interface

Micro-controller (CPU)
Memory (DRAM or SRAM or both)
Other Hardware-specific Chips

Internals

Figure 36.3: **A Canonical Device**

bootmain.c

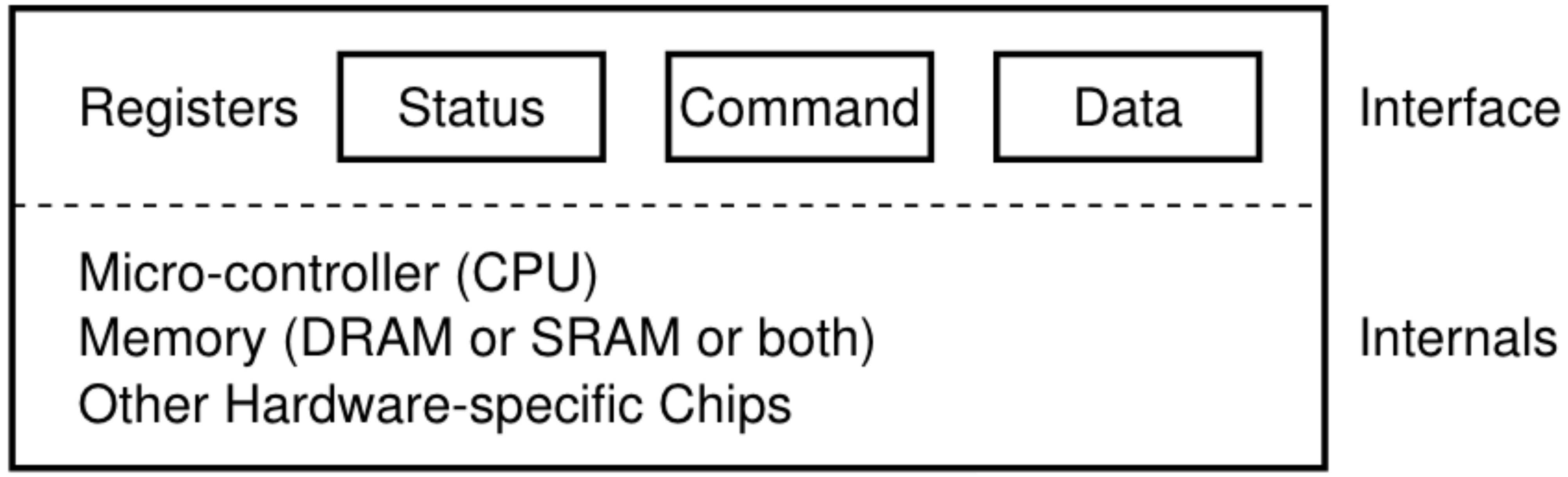
```
void waitdisk(void){
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40);
}

// 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);
}
```

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				



Registers Status Command Data

Interface

Micro-controller (CPU)
Memory (DRAM or SRAM or both)
Other Hardware-specific Chips

Internals

Figure 36.3: **A Canonical Device**

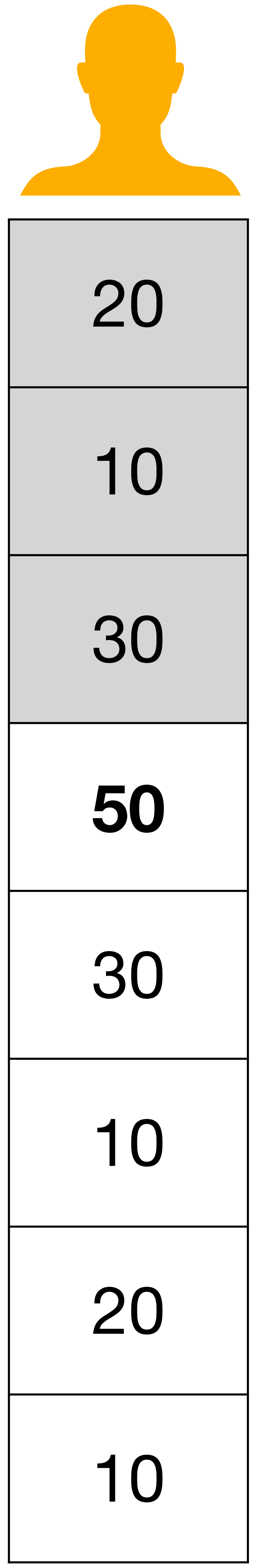
bootmain.c

```
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



20

10

30

50

30

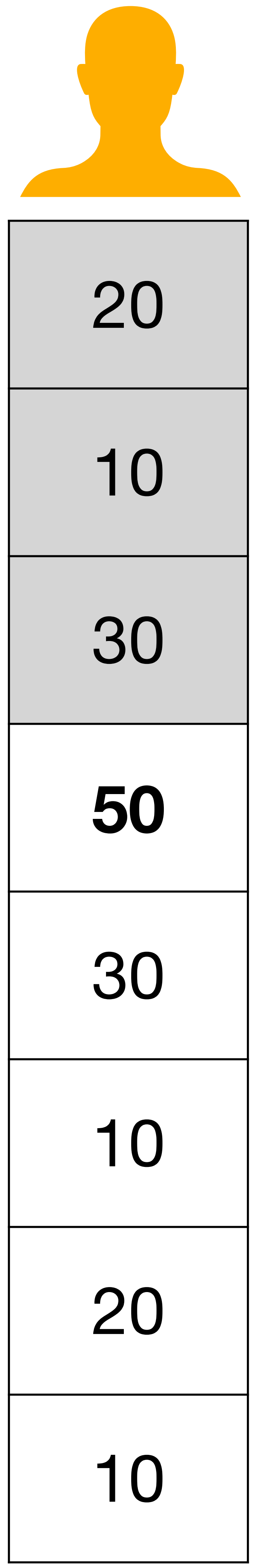
10

20

10

Calculator analogy

- 2 0 = (move pointer to 10)



20

10

30

50

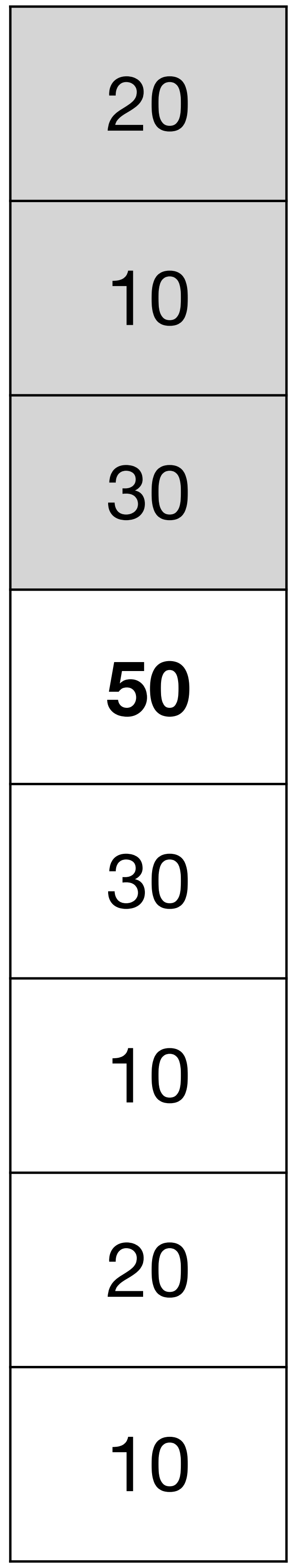
30

10

20

10

Calculator analogy



20

10

30

50

30

10

20

10

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

Calculator analogy

20
10
30
50
30
10
20
10

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

Calculator analogy

20
10
30
50
30
10
20
10

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

Interrupt ←

Give me the calculator!

Calculator analogy

20
10
30
50
30
10
20
10

- 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

Calculator analogy

20
10
30
50
30
10
20
10

- 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

Calculator analogy

20
10
30
50
30
10
20
10

- 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)

Calculator analogy

20
10
30
50
30
10
20
10

- 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)

Calculator analogy

20
10
30
50
30
10
20
10

- 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)

Calculator analogy

20
10
30
50
30
10
20
10

- 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

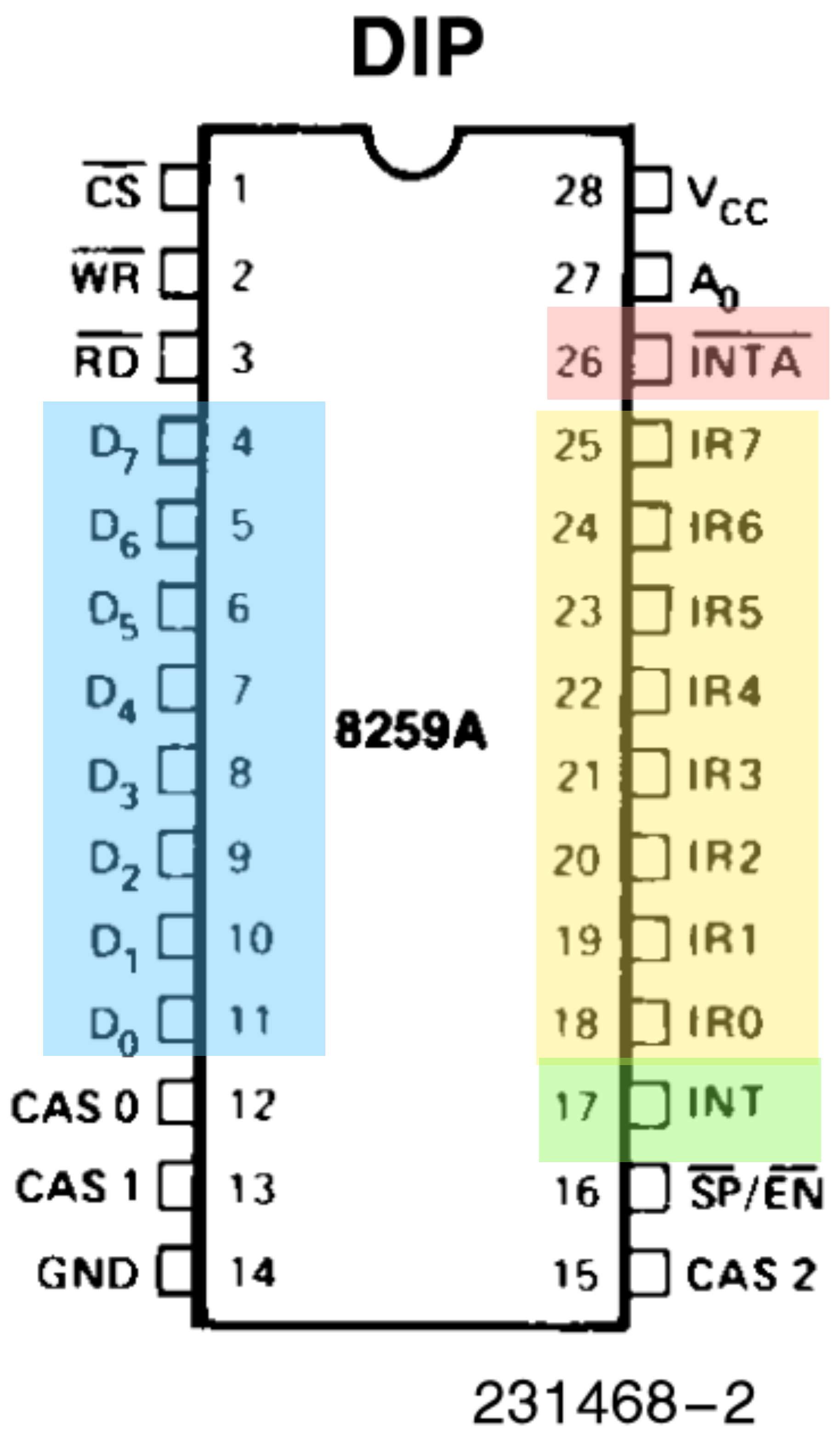
End of Interrupt ←

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



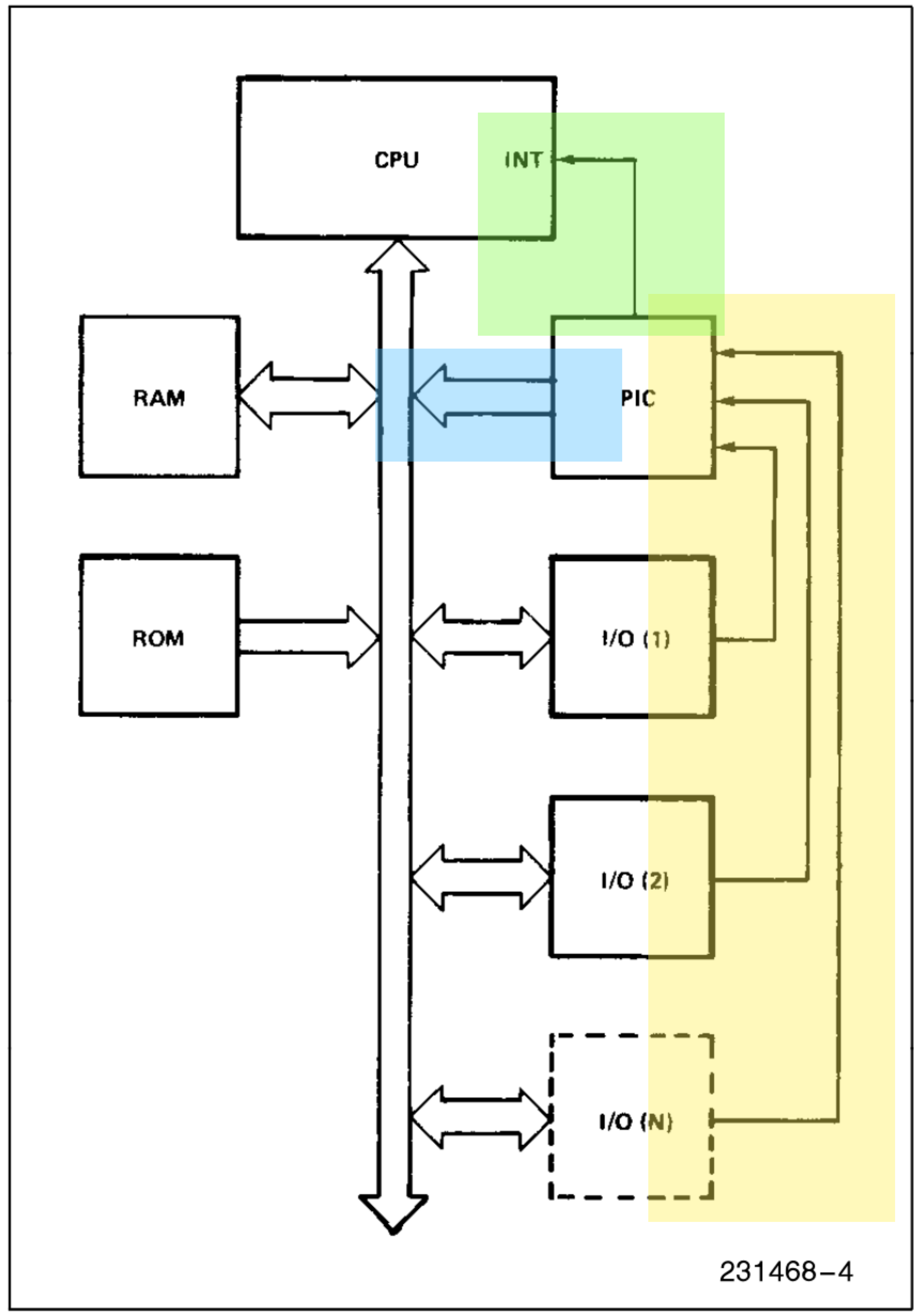


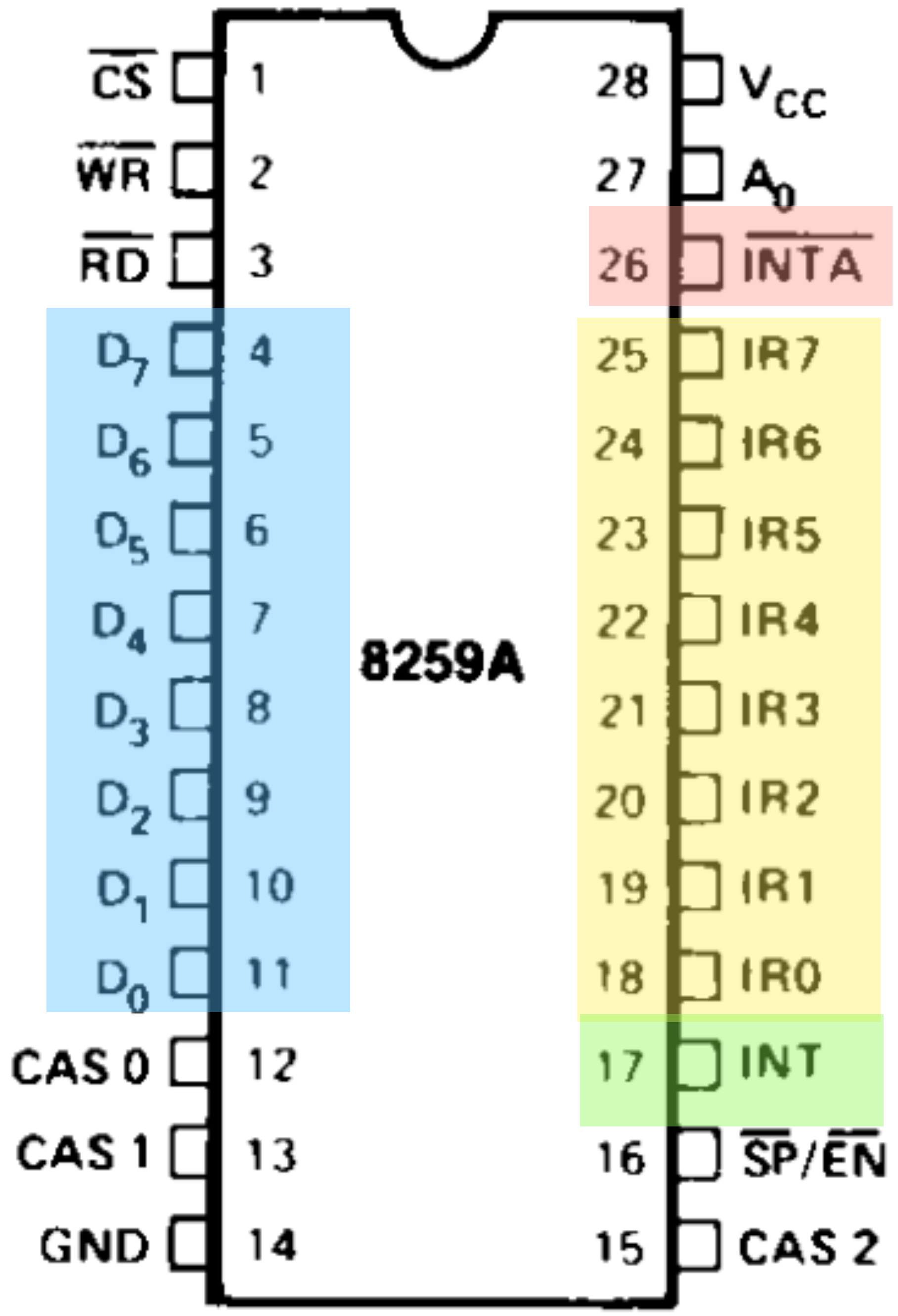
Figure 3b. Interrupt Method

Programmable interrupt controllers (PIC)

Example: Intel 8259A

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

DIP



231468–2

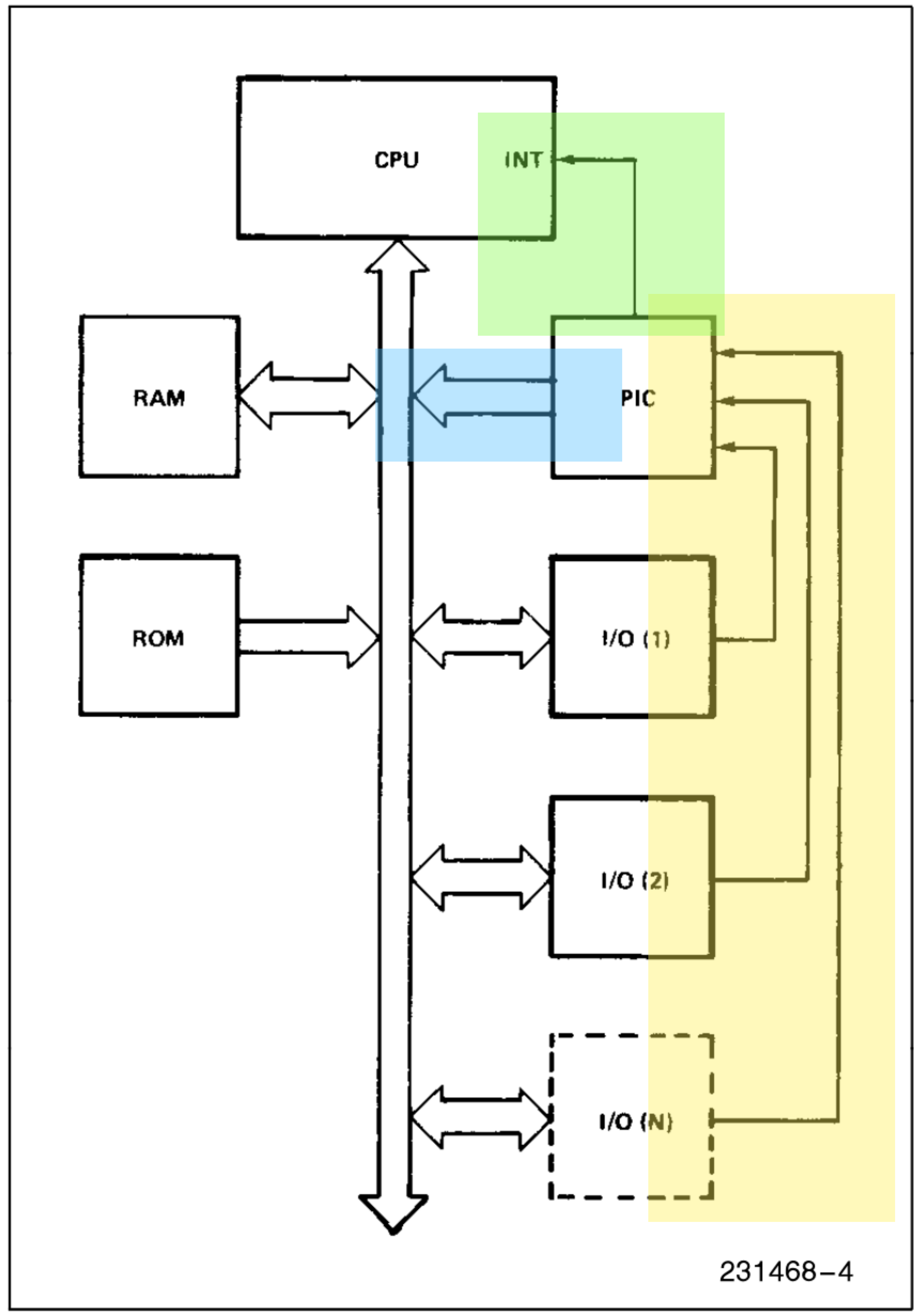


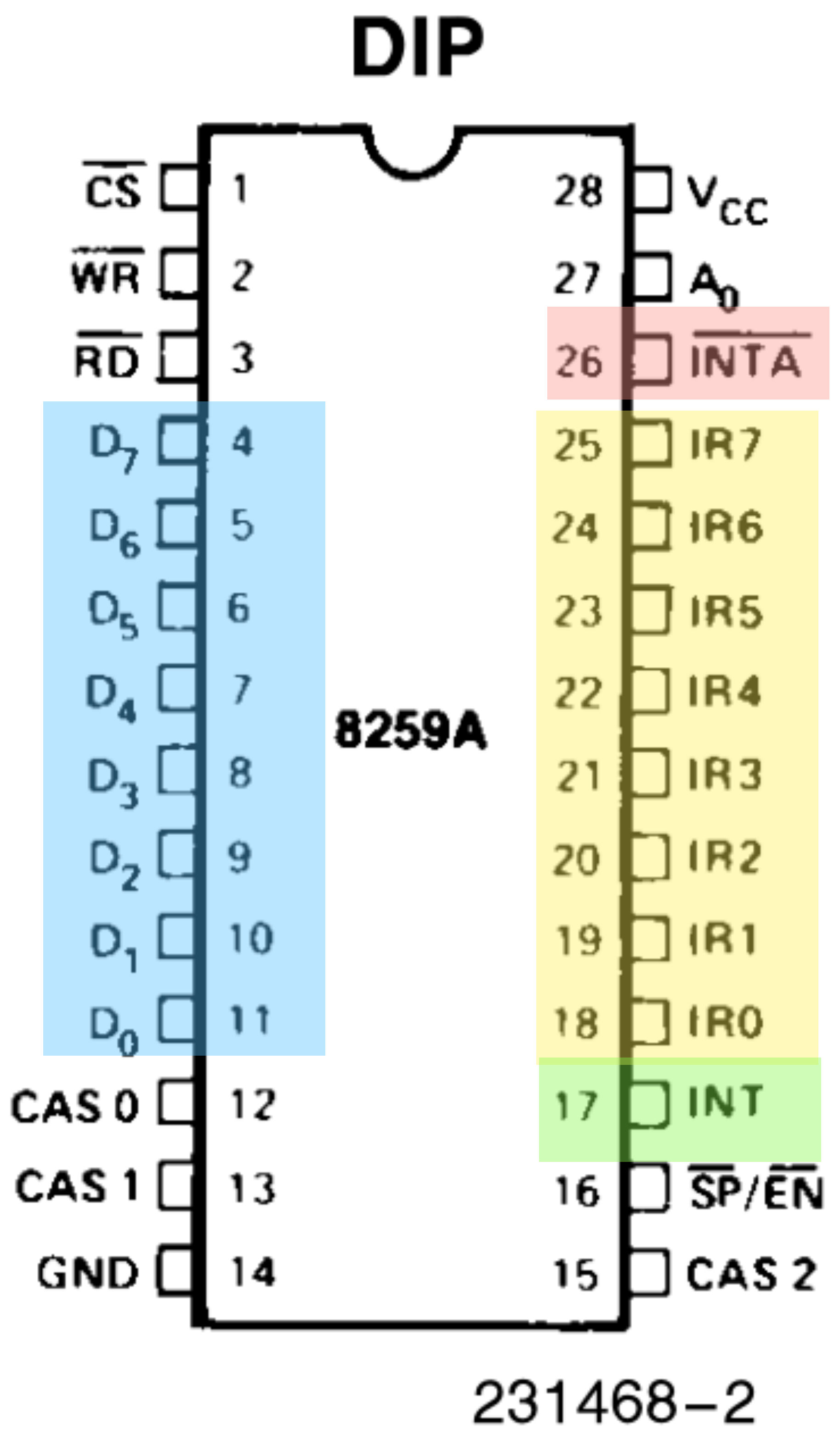
Figure 3b. Interrupt Method

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.



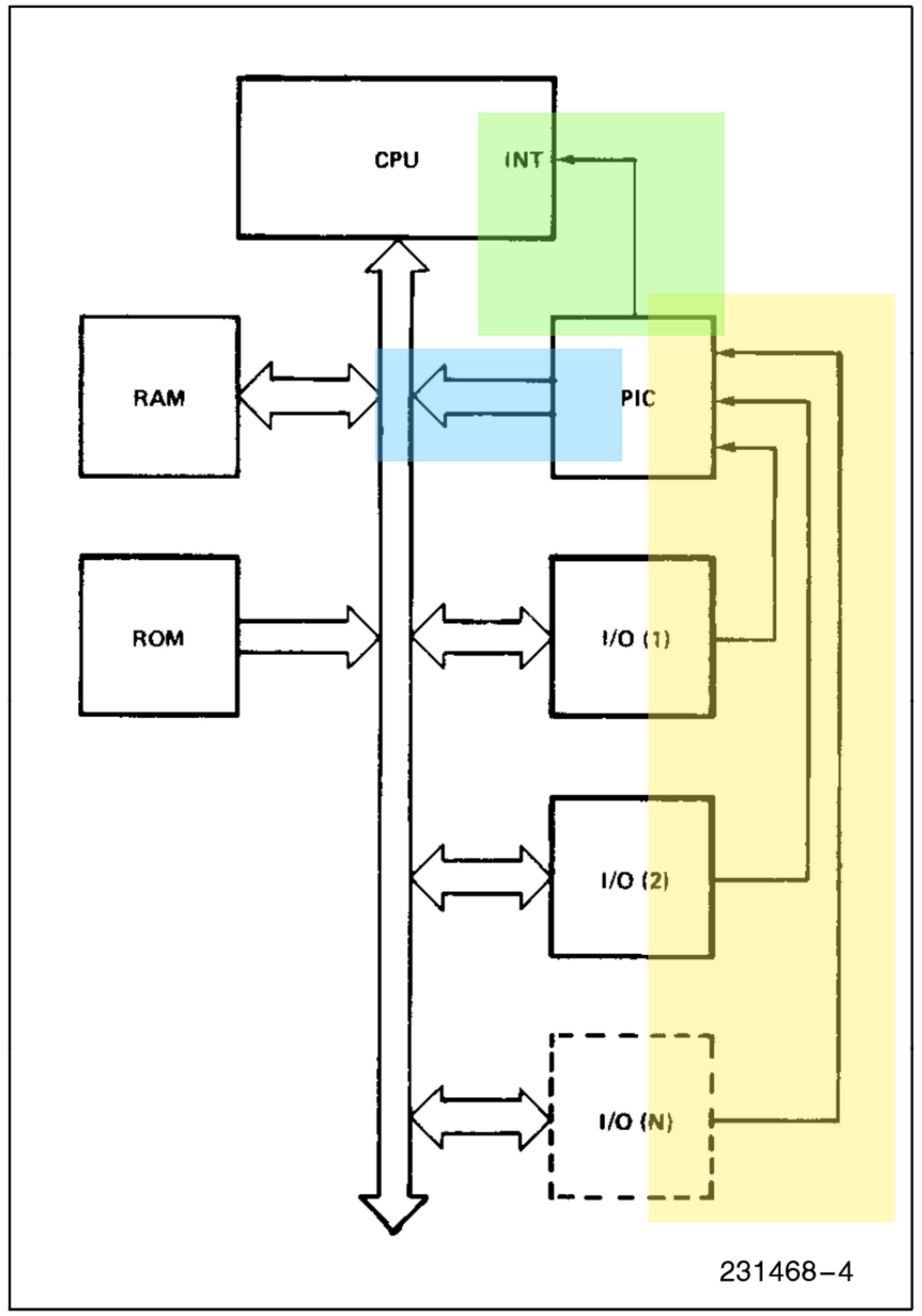


Figure 3b. Interrupt Method

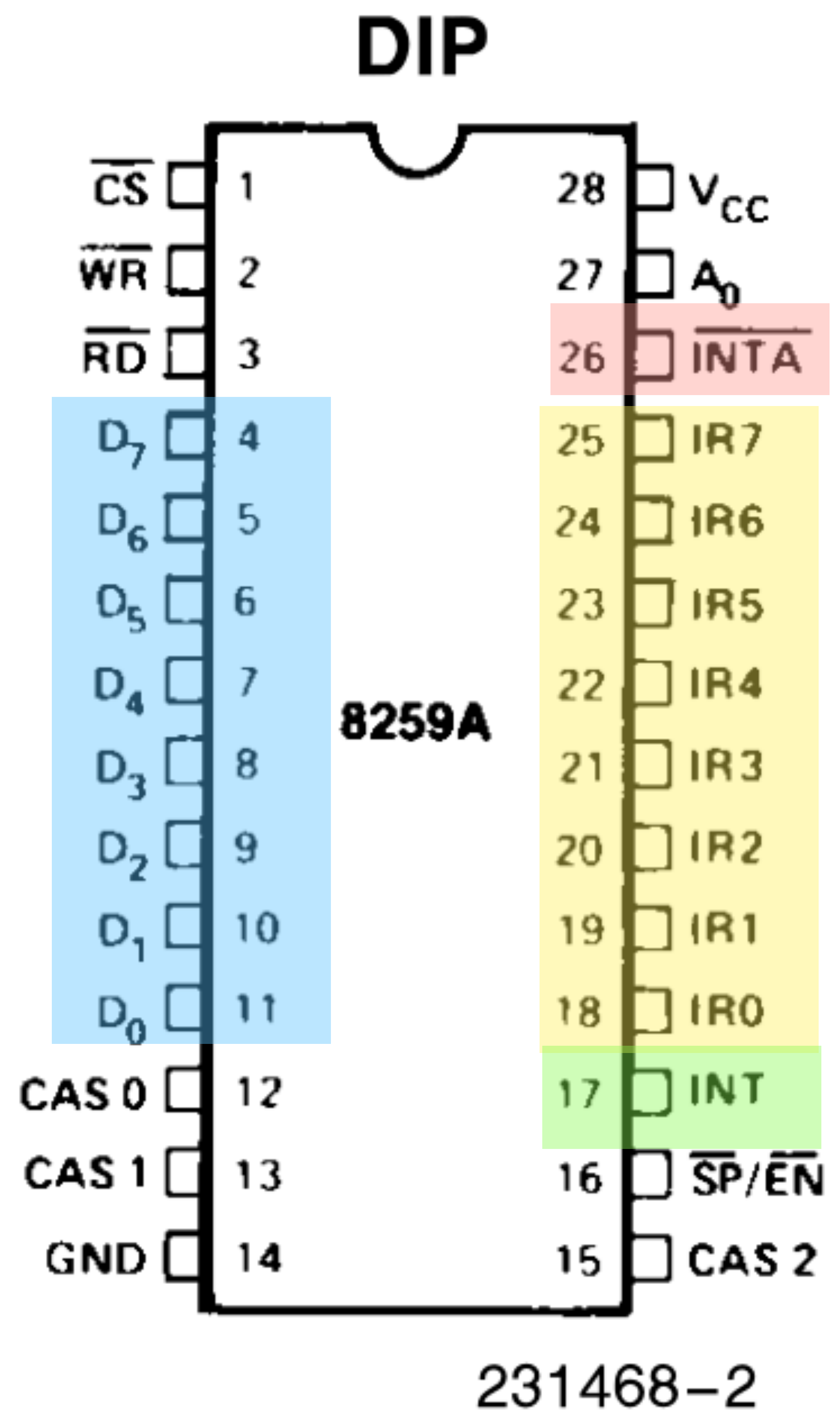
Programmable interrupt controllers (PIC)

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



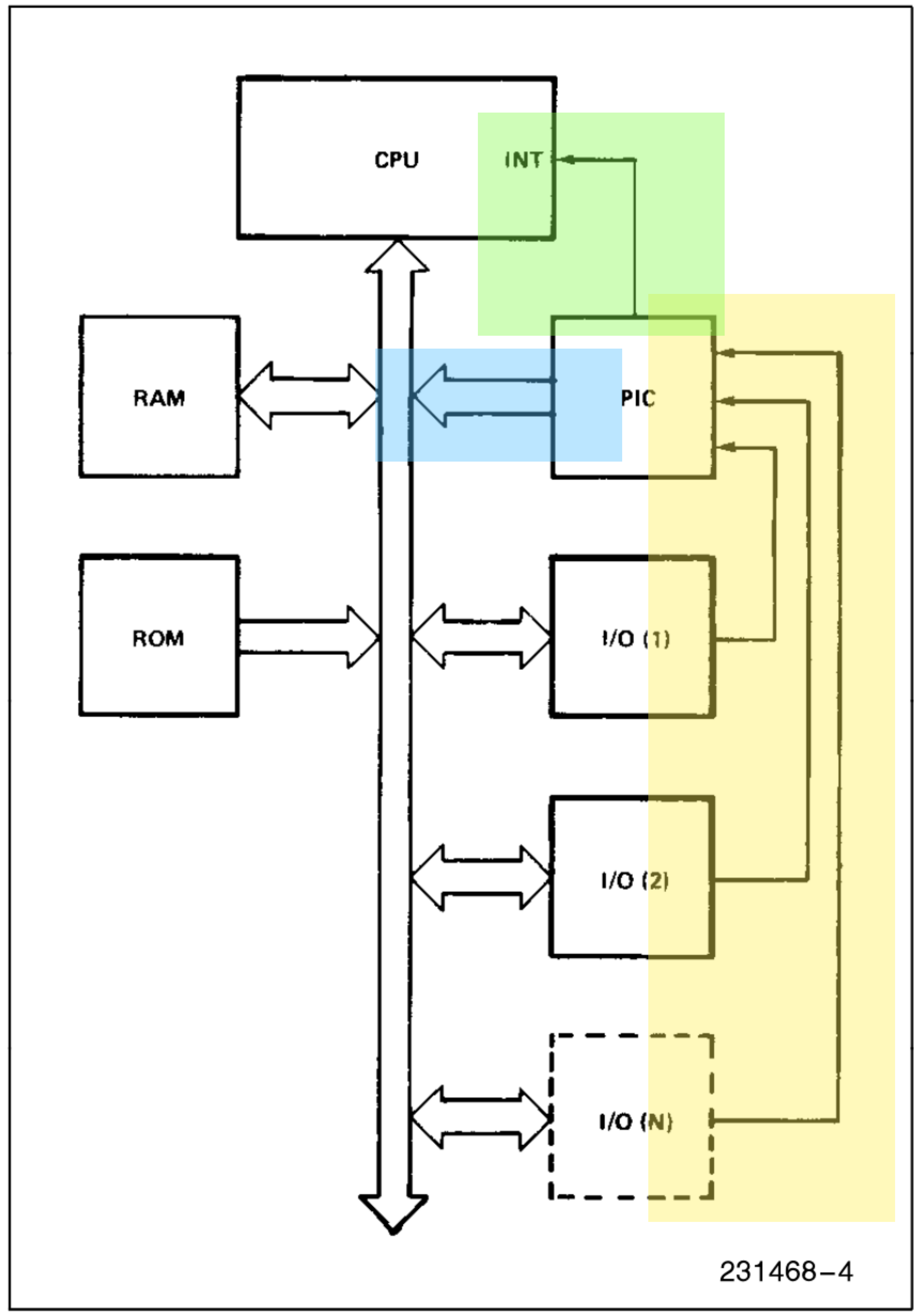
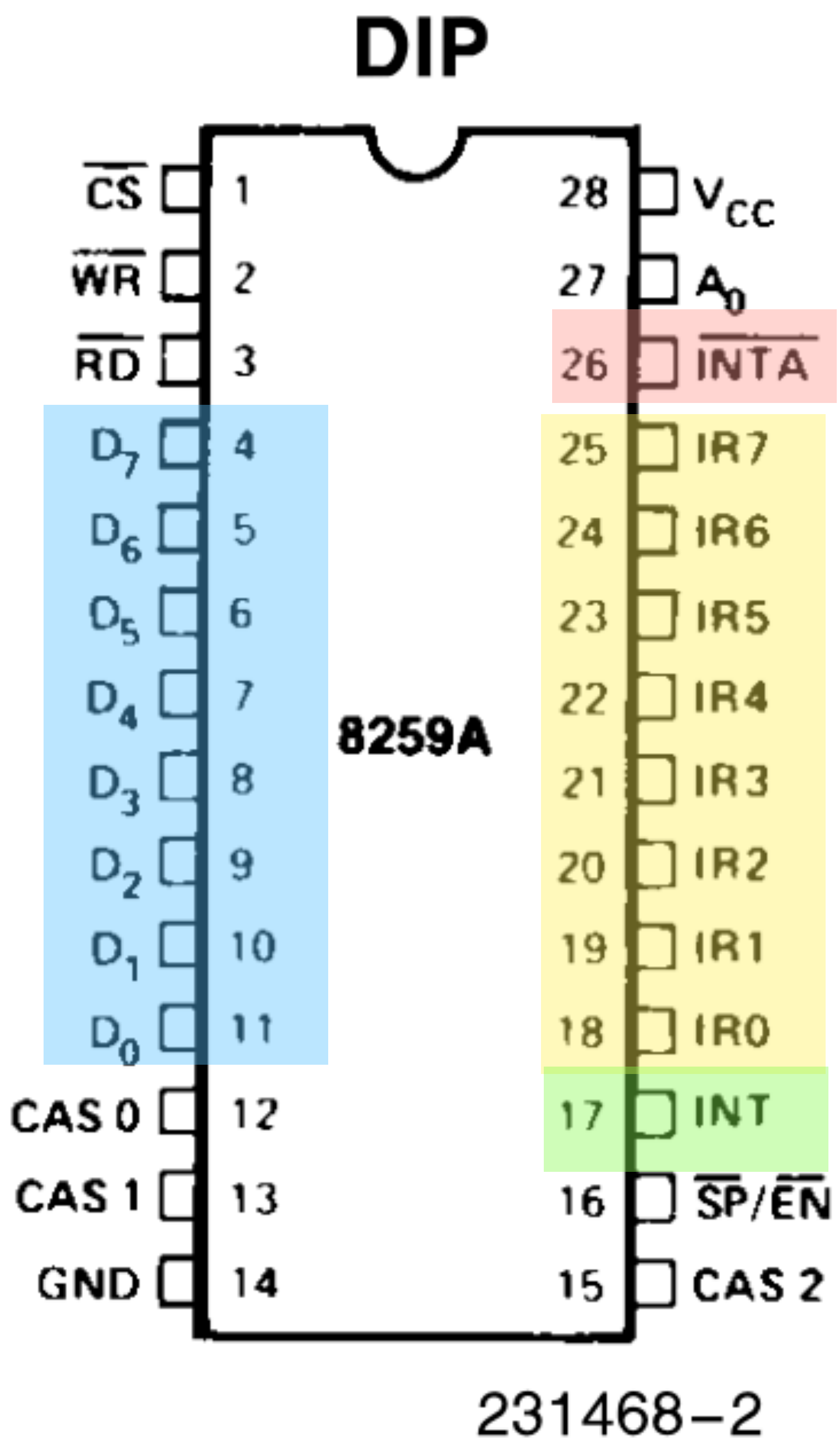


Figure 3b. Interrupt Method

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



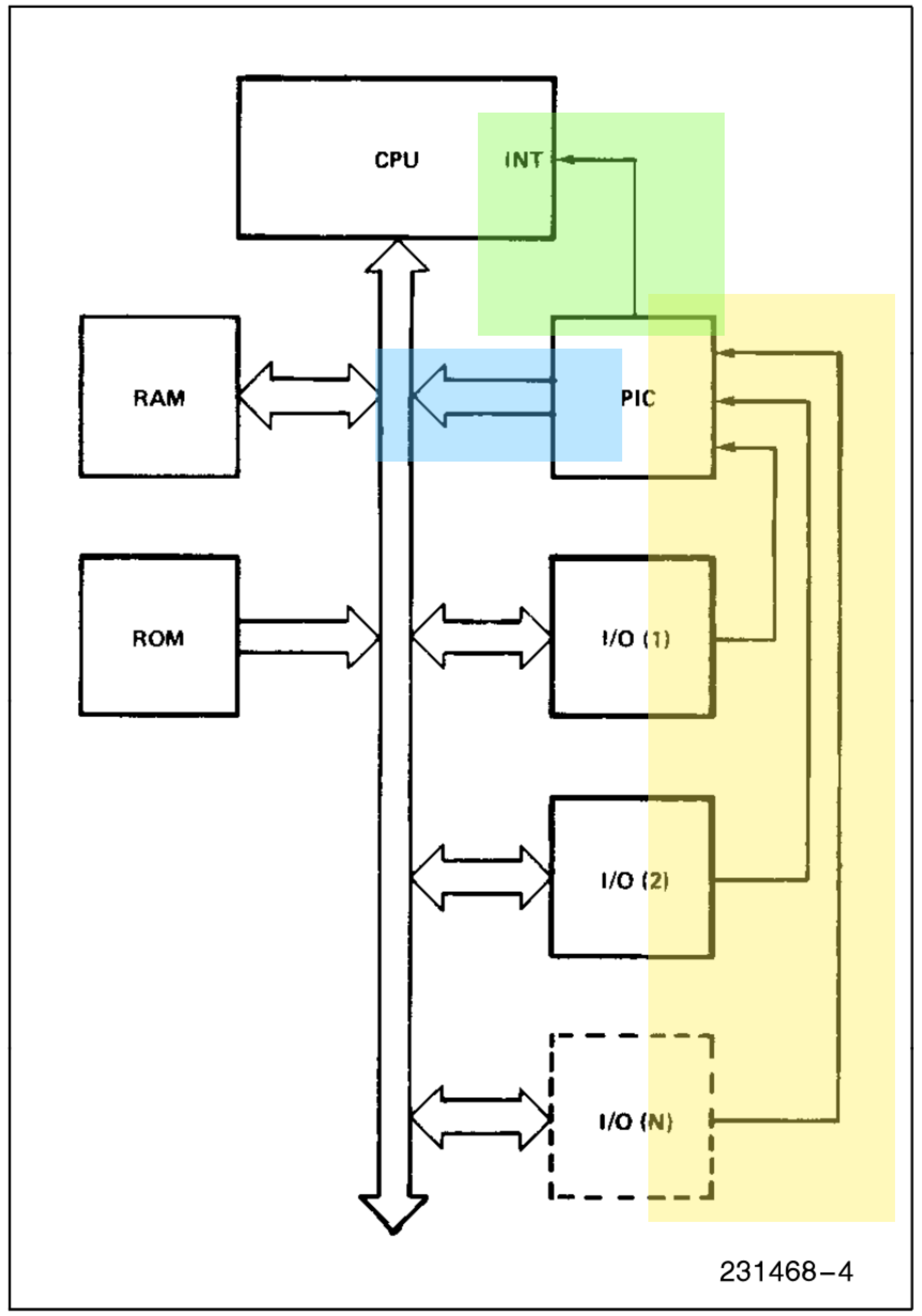
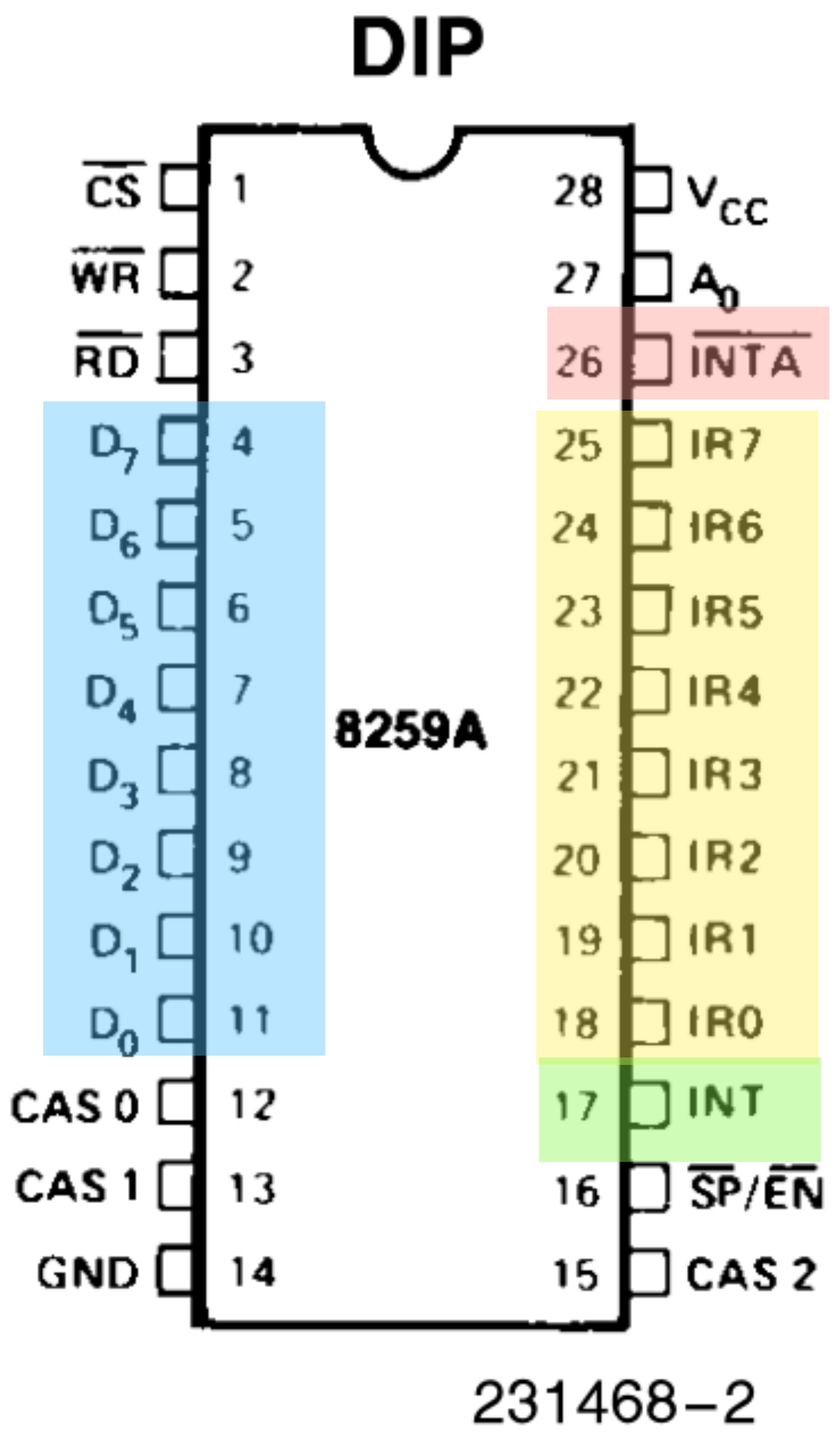


Figure 3b. Interrupt Method

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



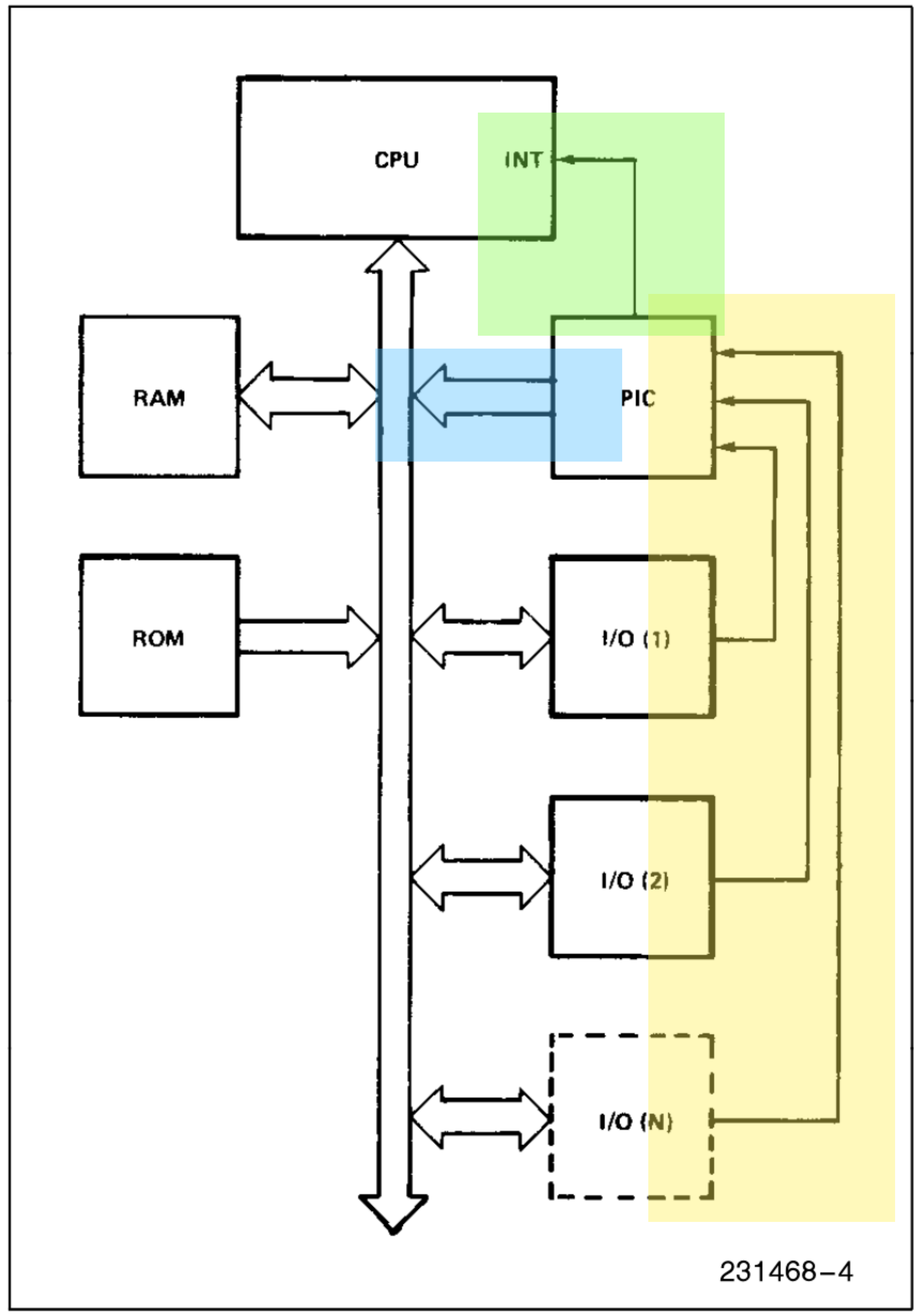
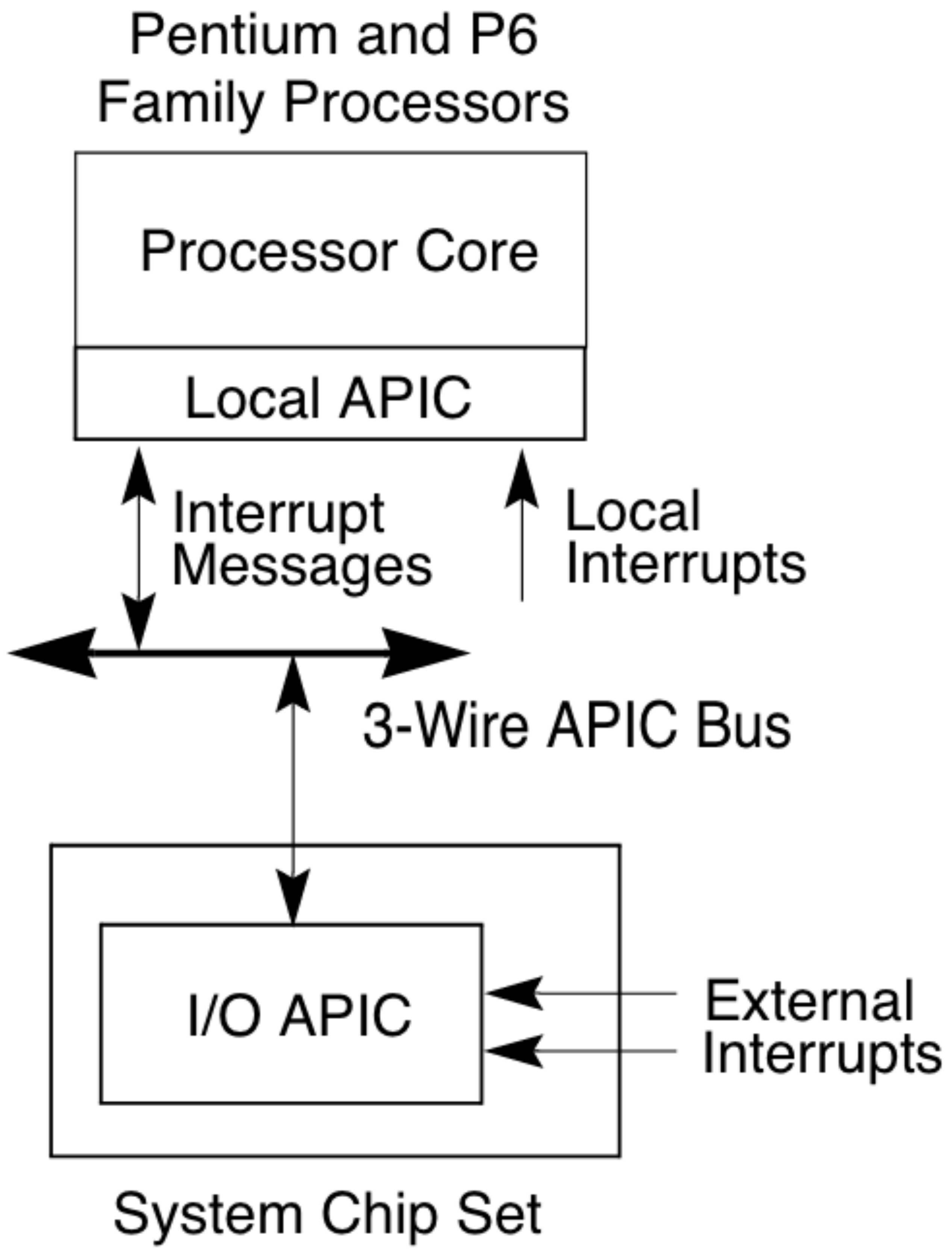


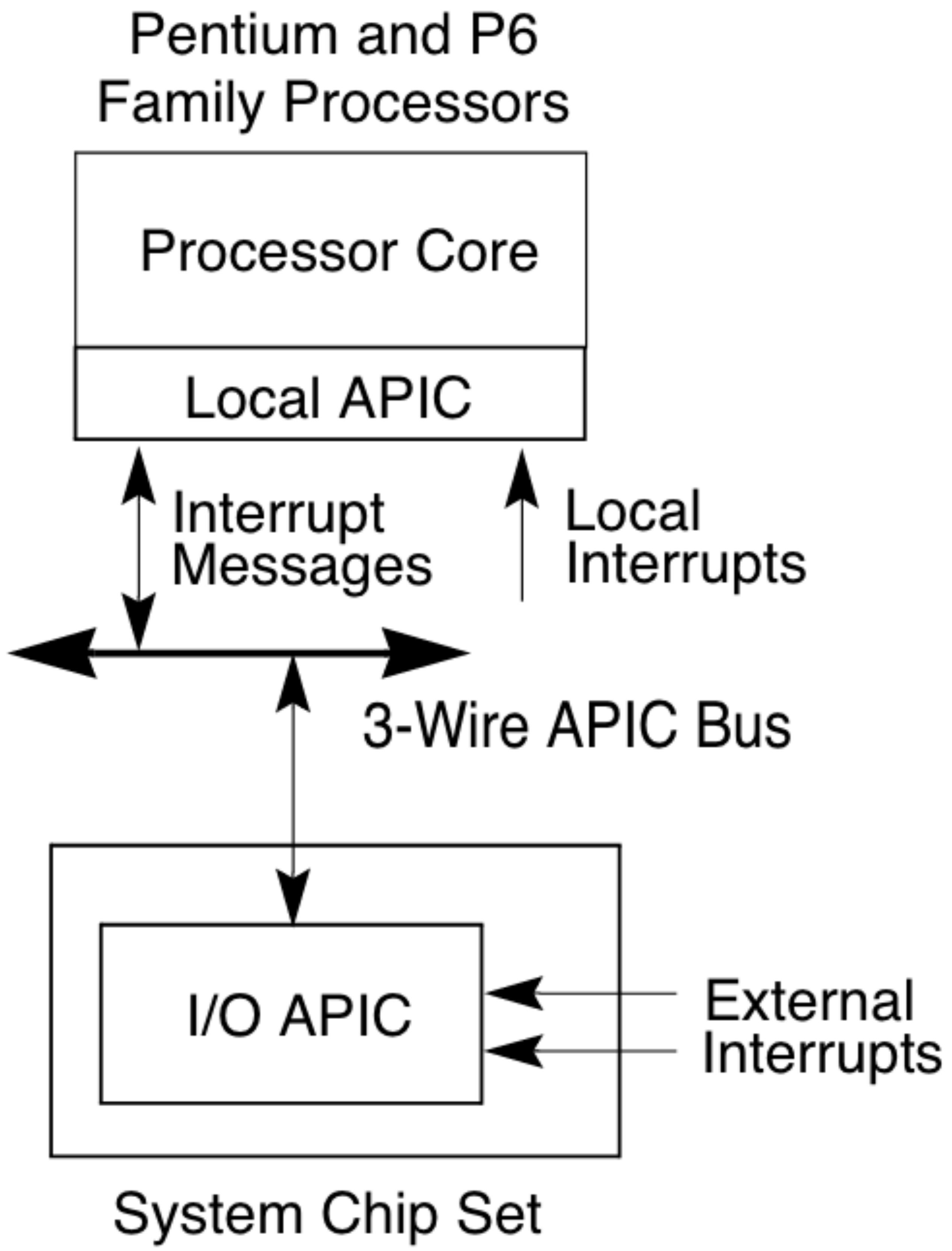
Figure 3b. Interrupt Method

Advanced programmable interrupt controllers (APIC)



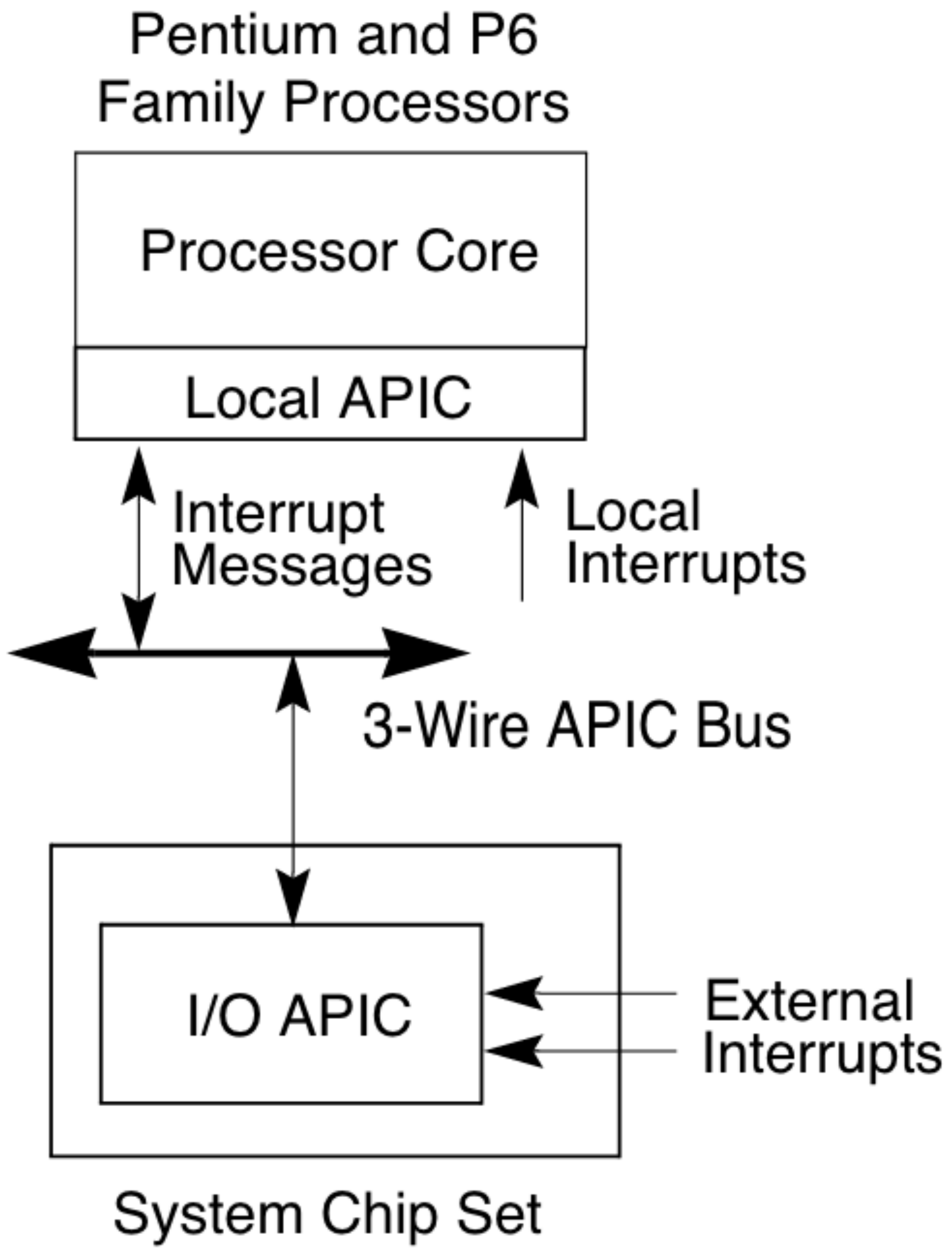
Advanced programmable interrupt controllers (APIC)

- Each CPU can have local APICs for handling *local interrupts* like timer, thermal sensor, etc.



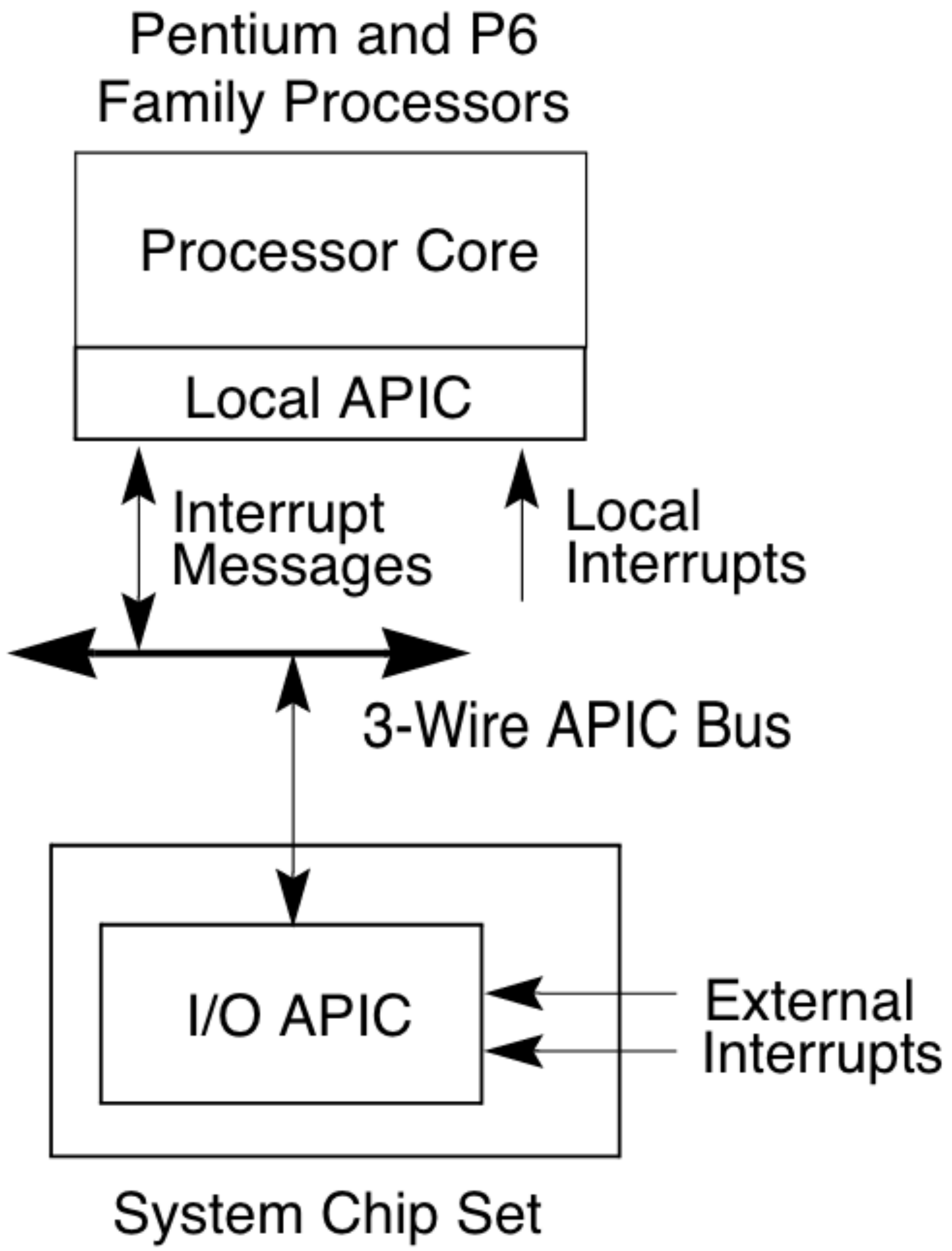
Advanced programmable interrupt controllers (APIC)

- 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



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

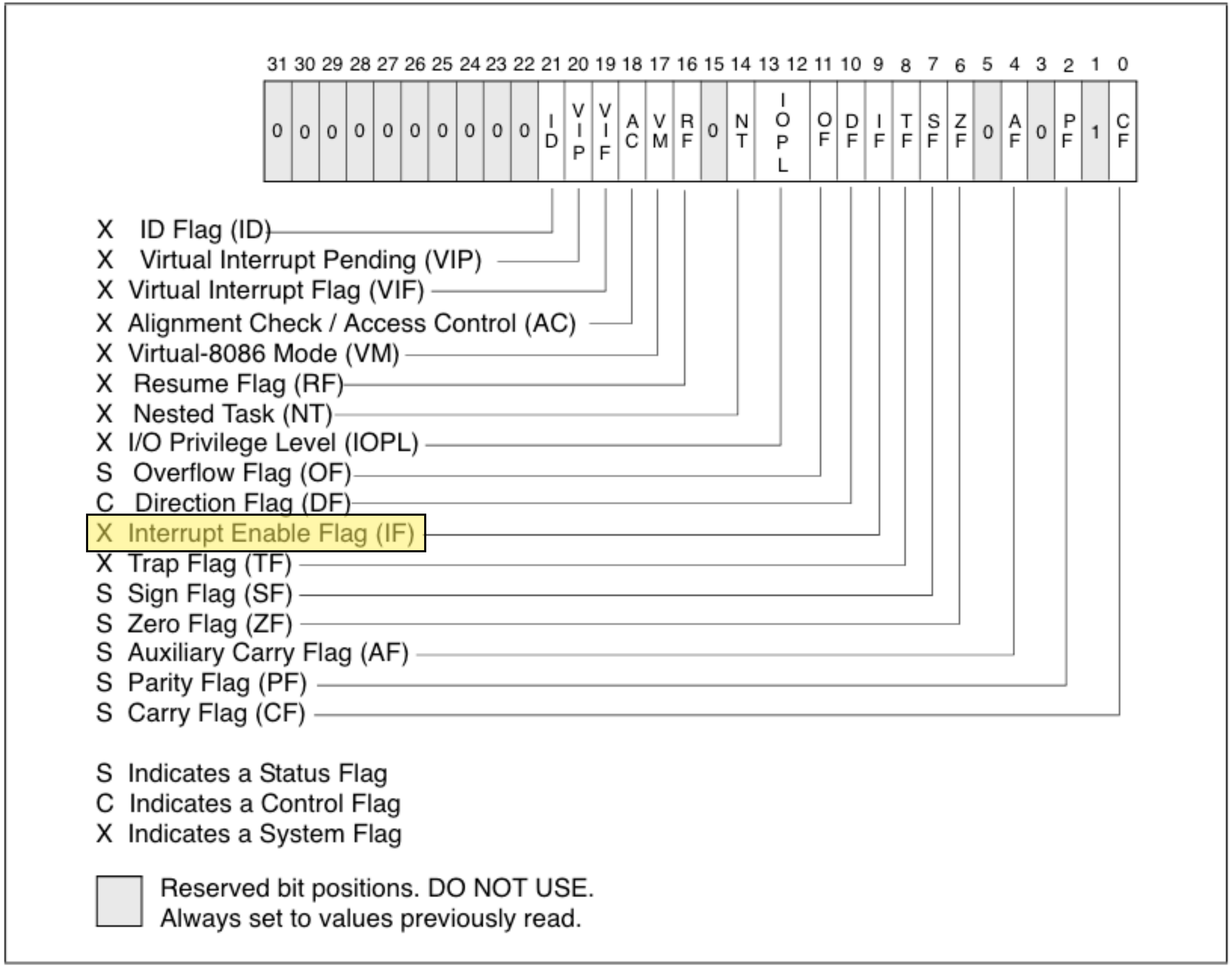


Code walkthrough

p3-pic

- main.c calls lapicinit, picinit, ioapicinit
- 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.

Interrupt enable flag



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)

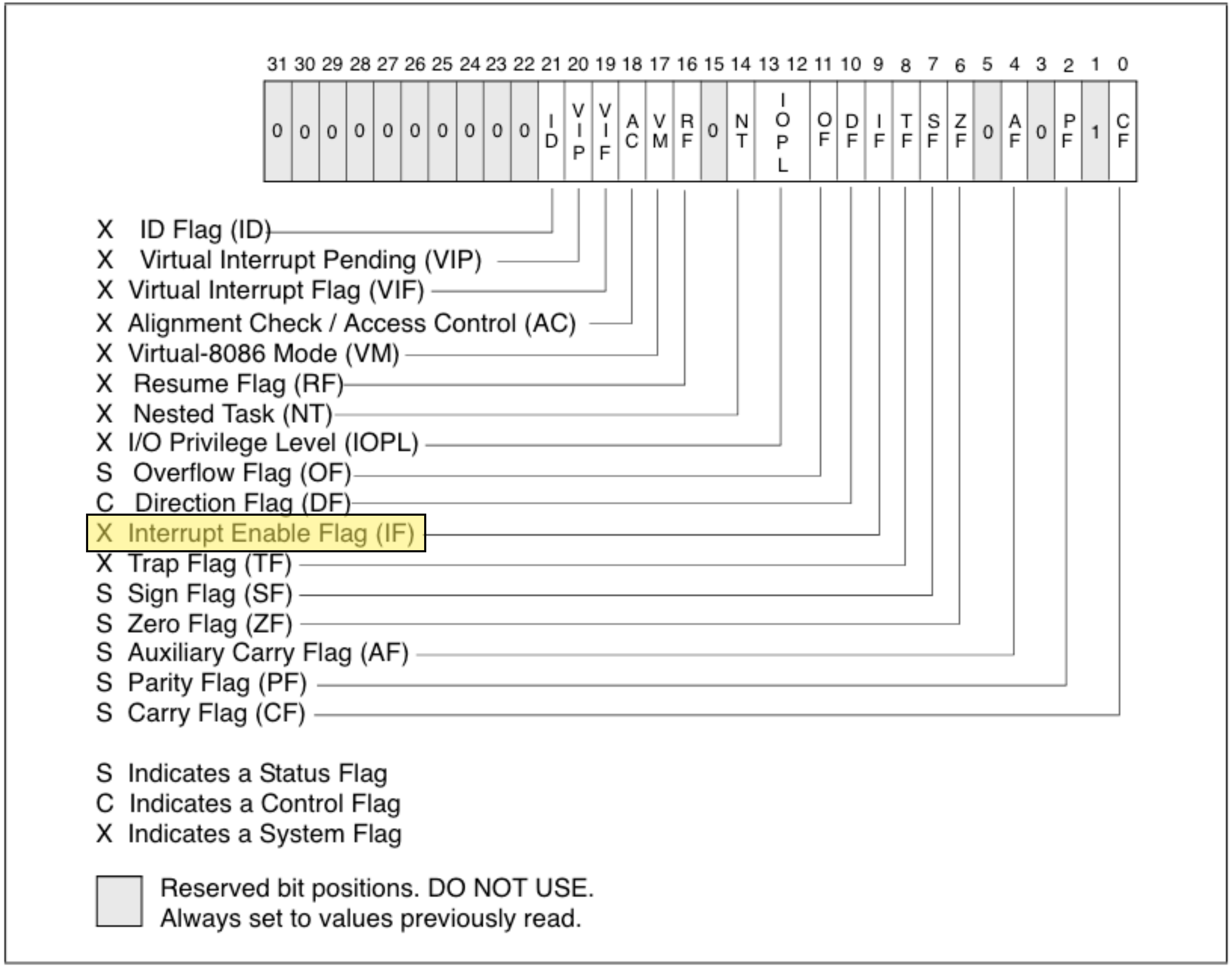
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

▨ Reserved bit positions. DO NOT USE.
Always set to values previously read.

Figure 3-8. EFLAGS Register

Interrupt enable flag

- cli: Clear 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)

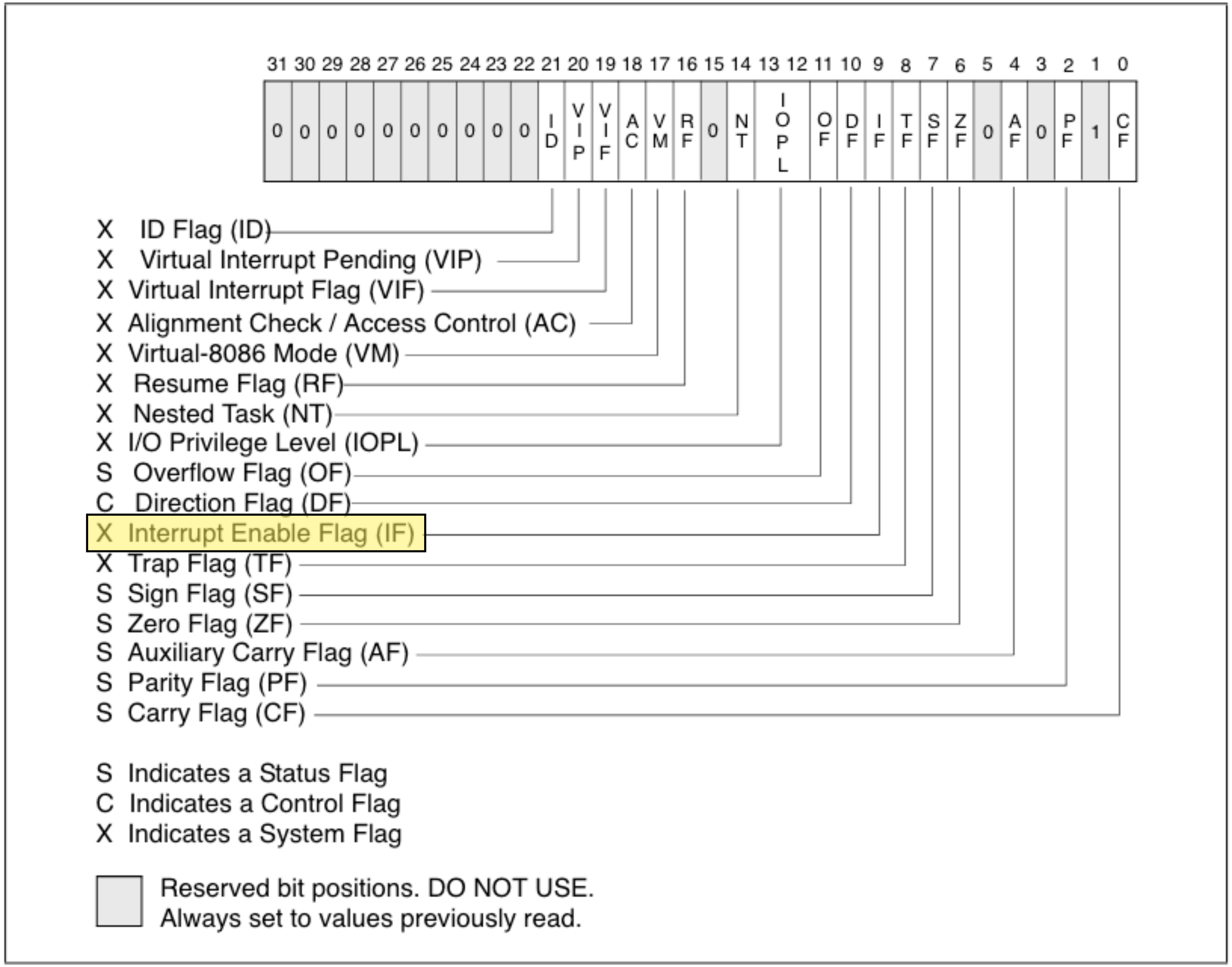
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

▨ Reserved bit positions. DO NOT USE.
Always set to values previously read.

Figure 3-8. EFLAGS Register

Interrupt enable flag

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

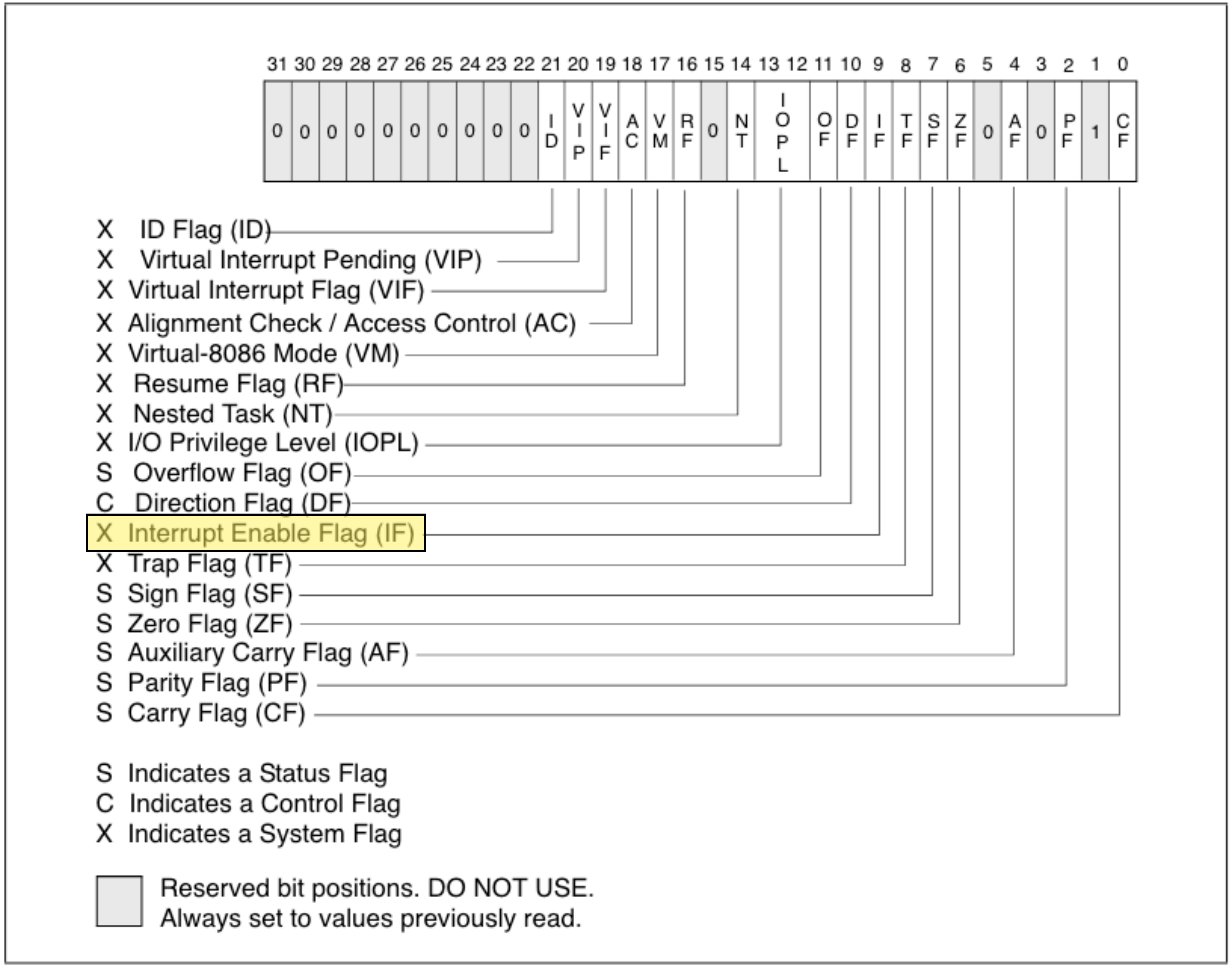


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 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 enable flag

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



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

Interrupt handling in a nutshell

- OS sets up “interrupt descriptor table” (IDT)

Interrupt handling in a nutshell

- OS sets up “interrupt descriptor table” (IDT)
- Points IDTR to IDT using LIDT instruction

Interrupt handling in a nutshell

- OS sets up “interrupt descriptor table” (IDT)
- Points IDTR to IDT using LIDT instruction

IDTR: Interrupt descriptor table register

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

Interrupt handling in a nutshell

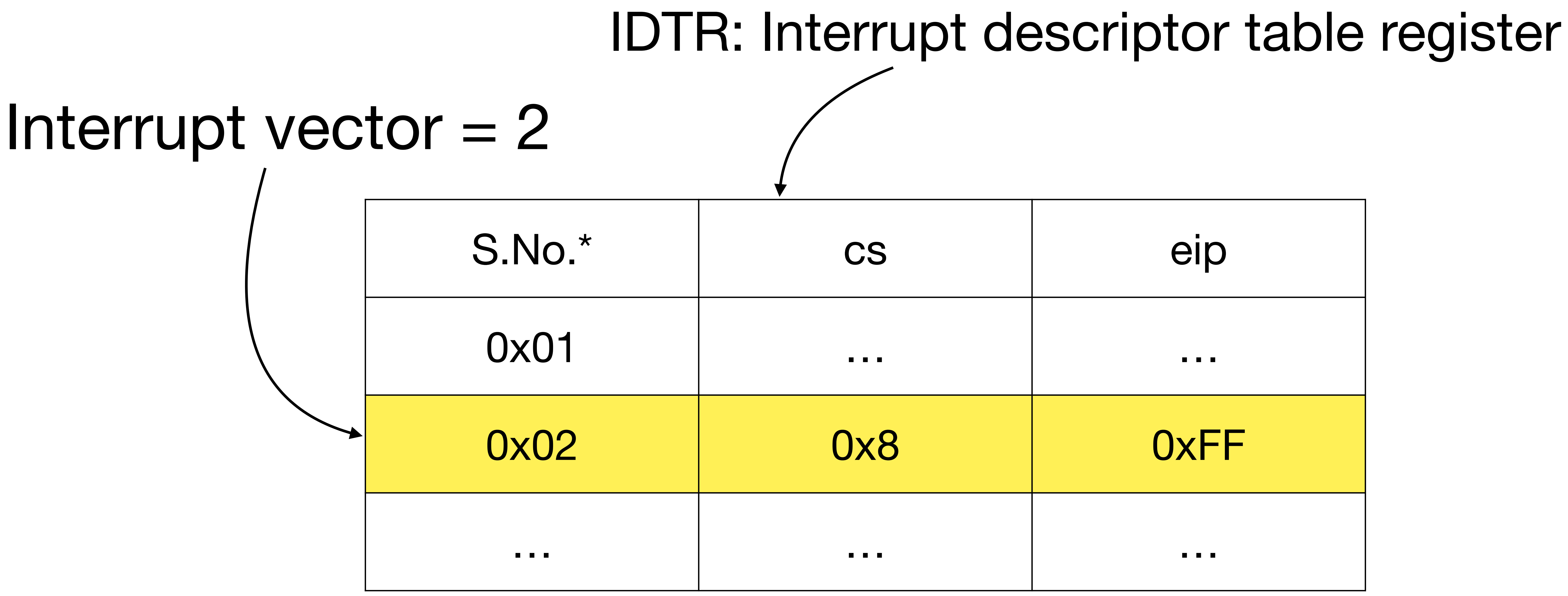
- OS sets up “interrupt descriptor table” (IDT)
- Points IDTR to IDT using LIDT instruction
- When interrupt occurs, jump %eip to interrupt handler, handle interrupt, tell LAPIC about end of interrupt, resume what we were doing

IDTR: Interrupt descriptor table register

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

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



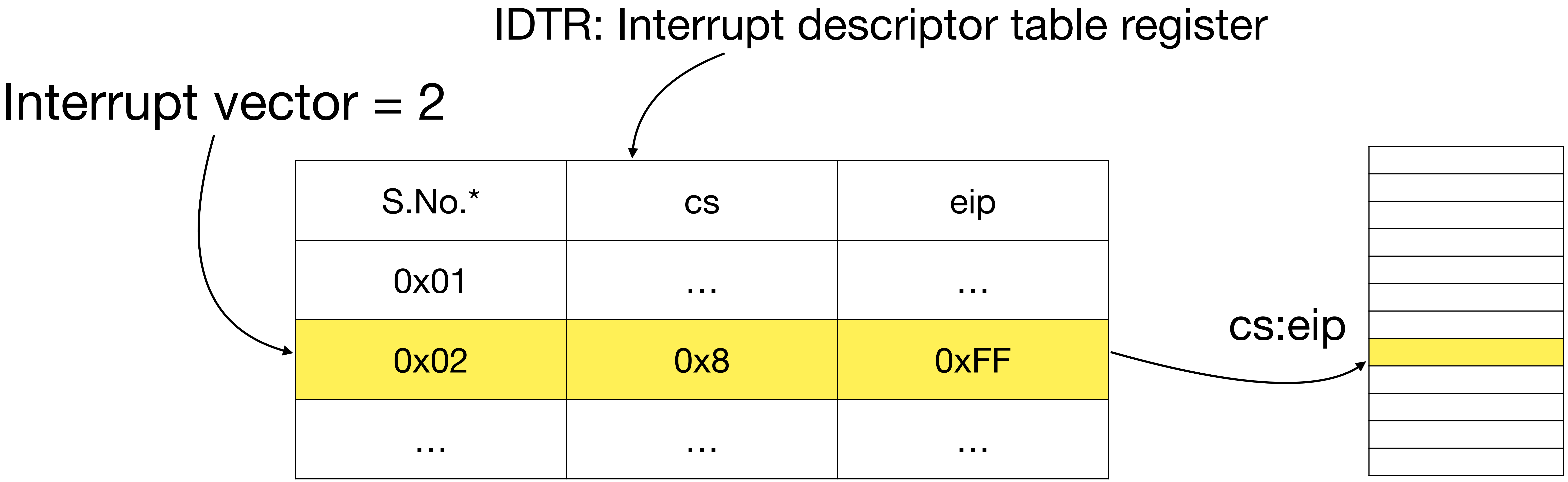
IDTR: Interrupt descriptor table register

Interrupt vector = 2

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

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

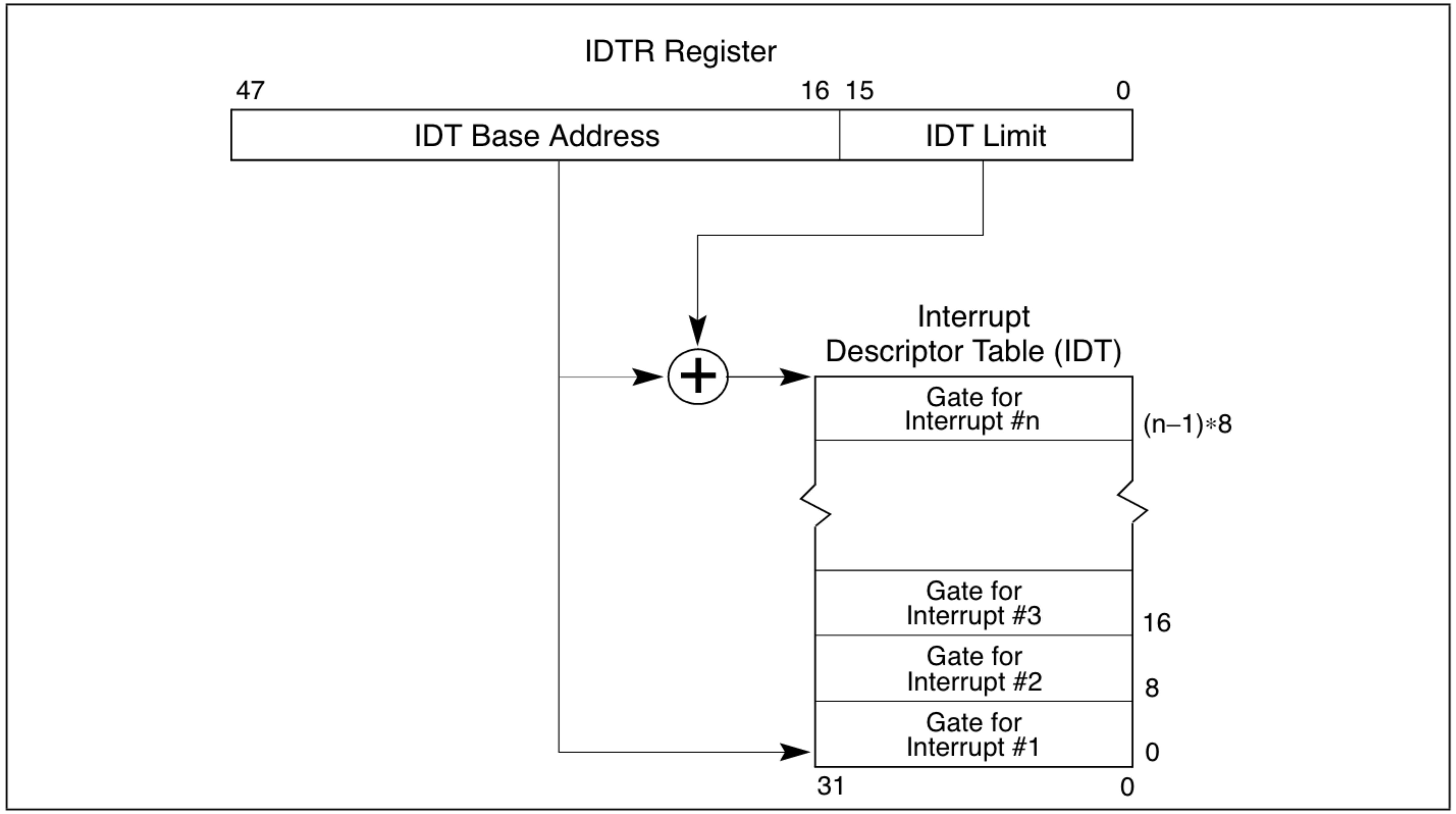


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor table

- Interrupt descriptor table register (IDTR) points to interrupt descriptor table in memory

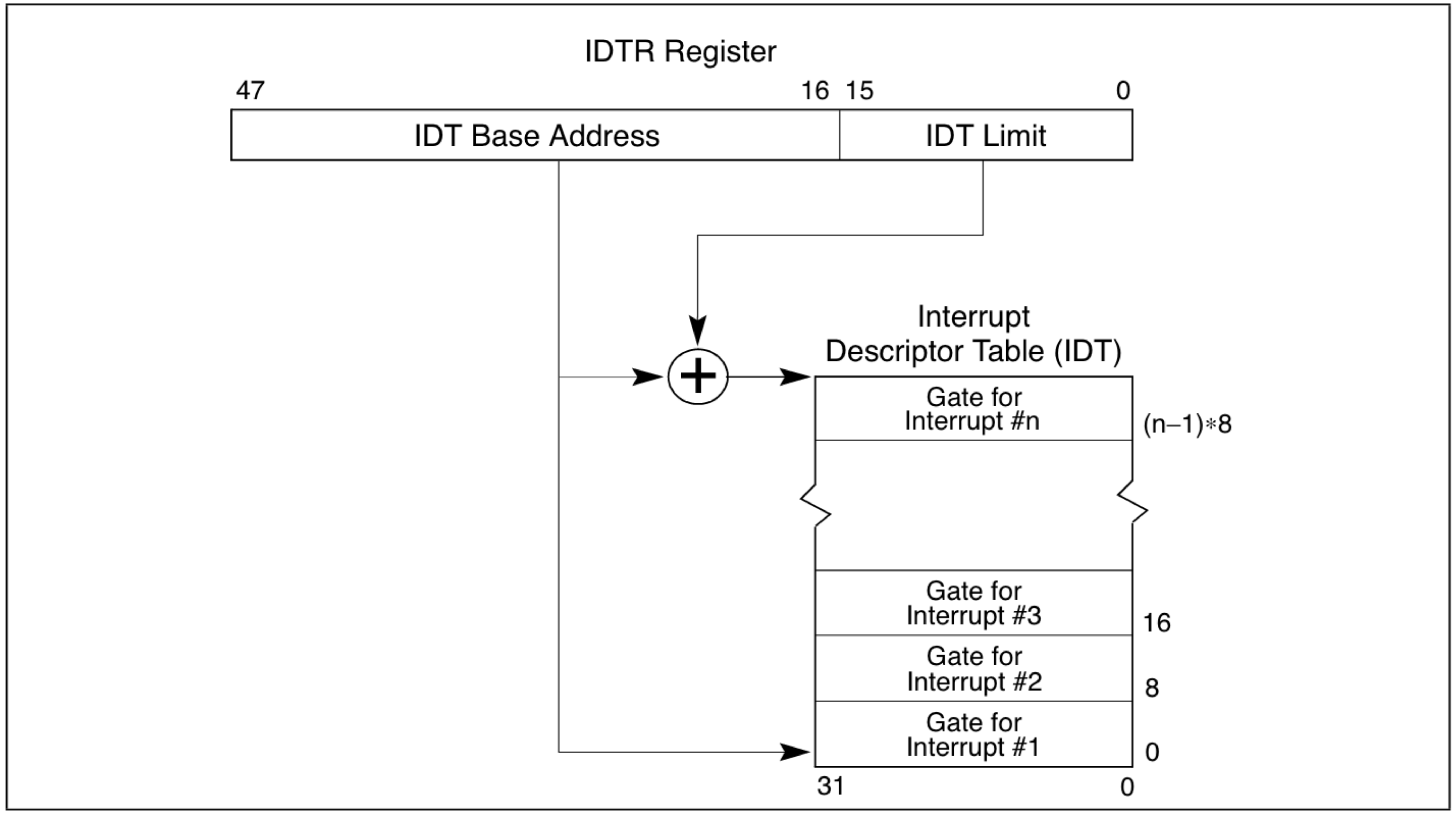


Figure 6-1. Relationship of the IDTR and IDT

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

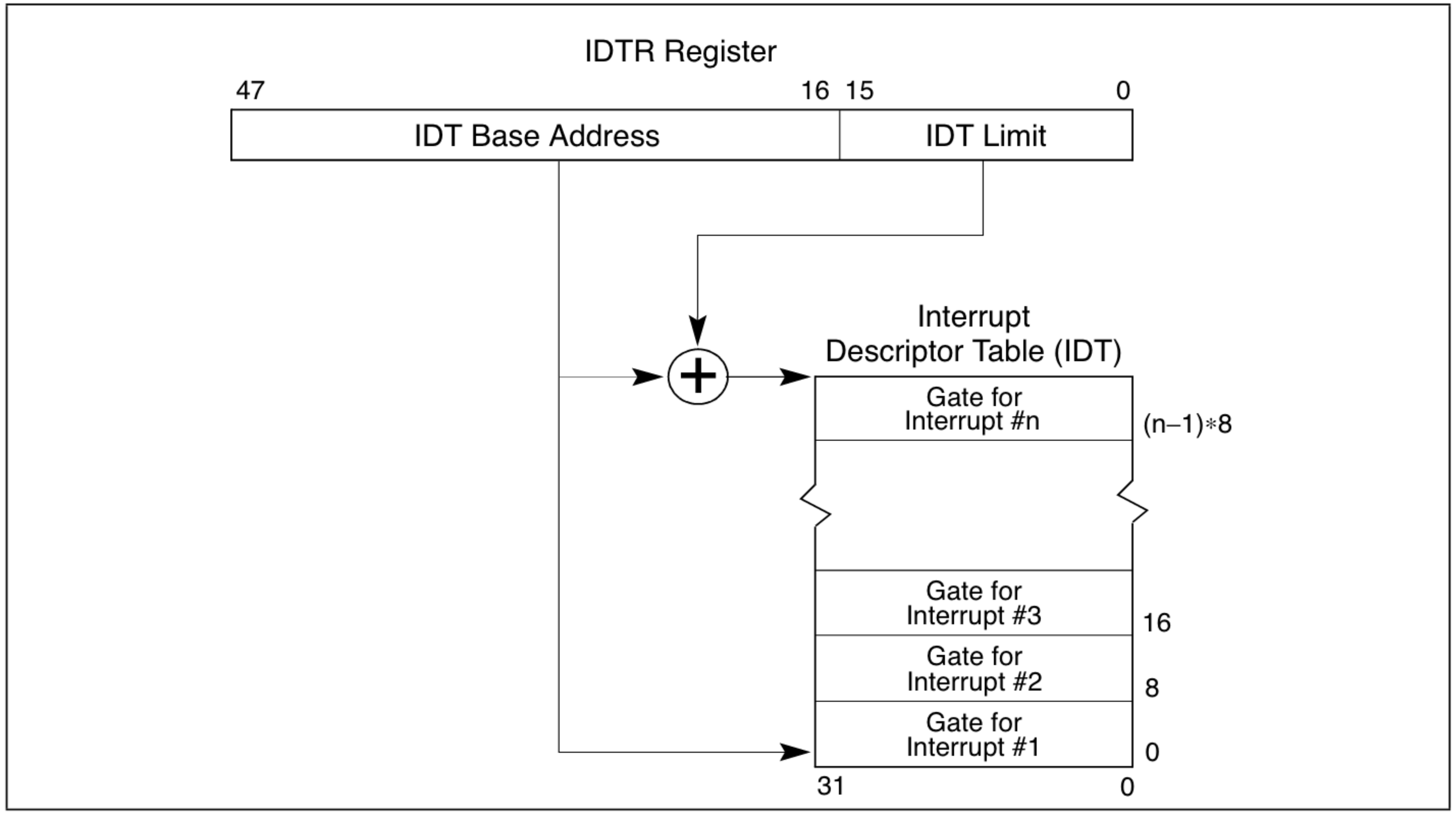


Figure 6-1. Relationship of the IDTR and IDT

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)

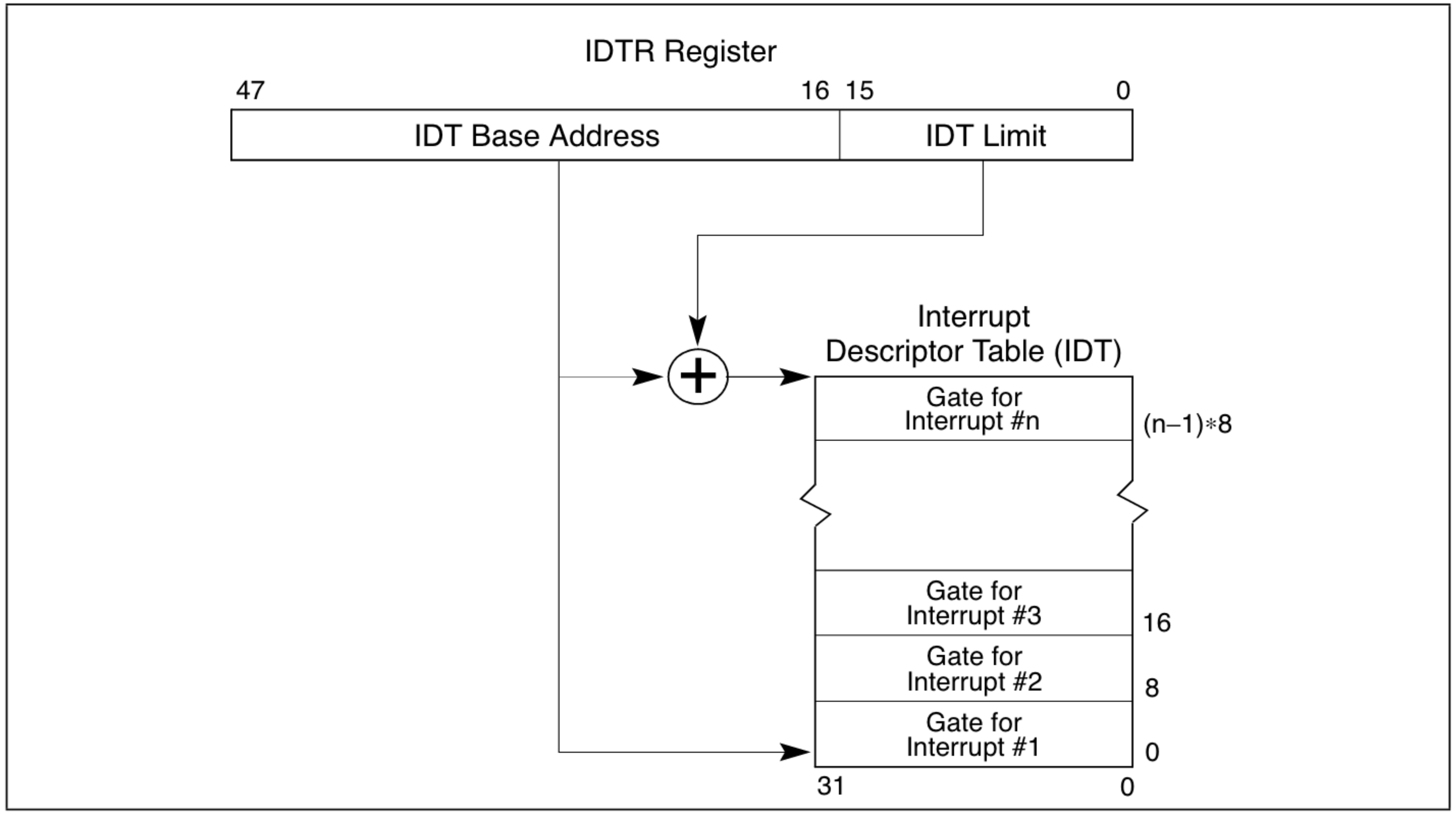
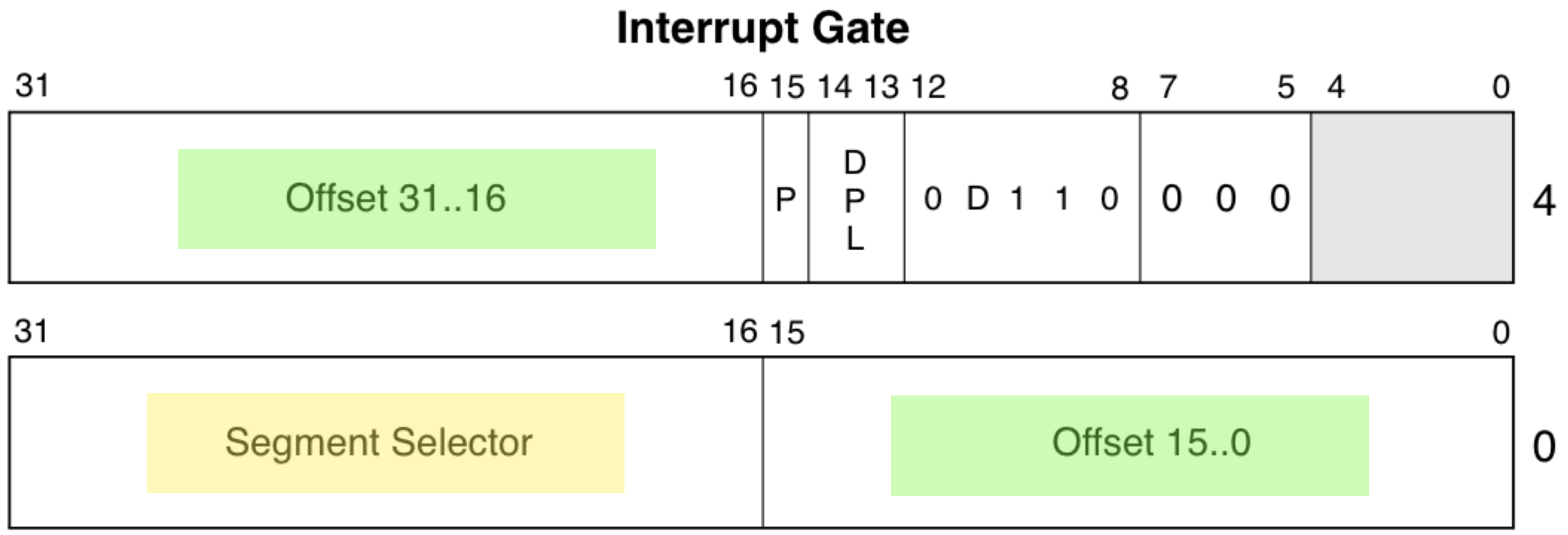


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor table (2)



Interrupt Gate

31 16 15 14 13 12 8 7 5 4 0

Offset 31..16

P

DPL

0 D 1 1 0

0 0 0

4

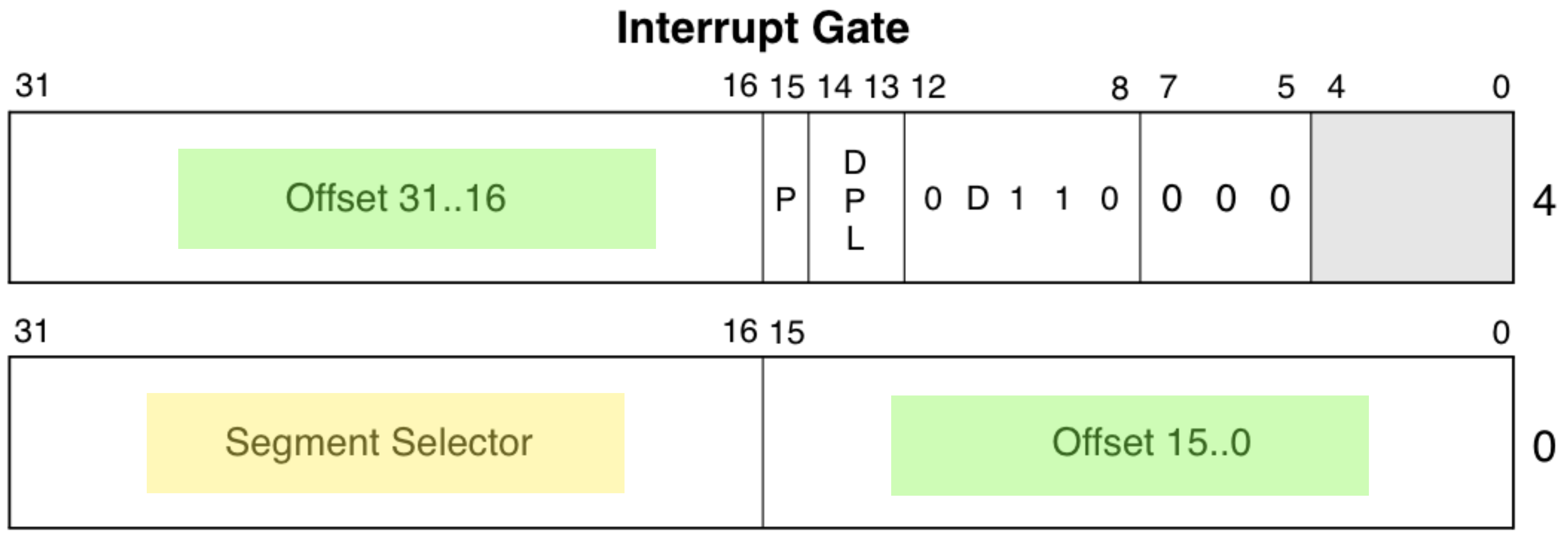
31 16 15 0

Segment Selector

Offset 15..0

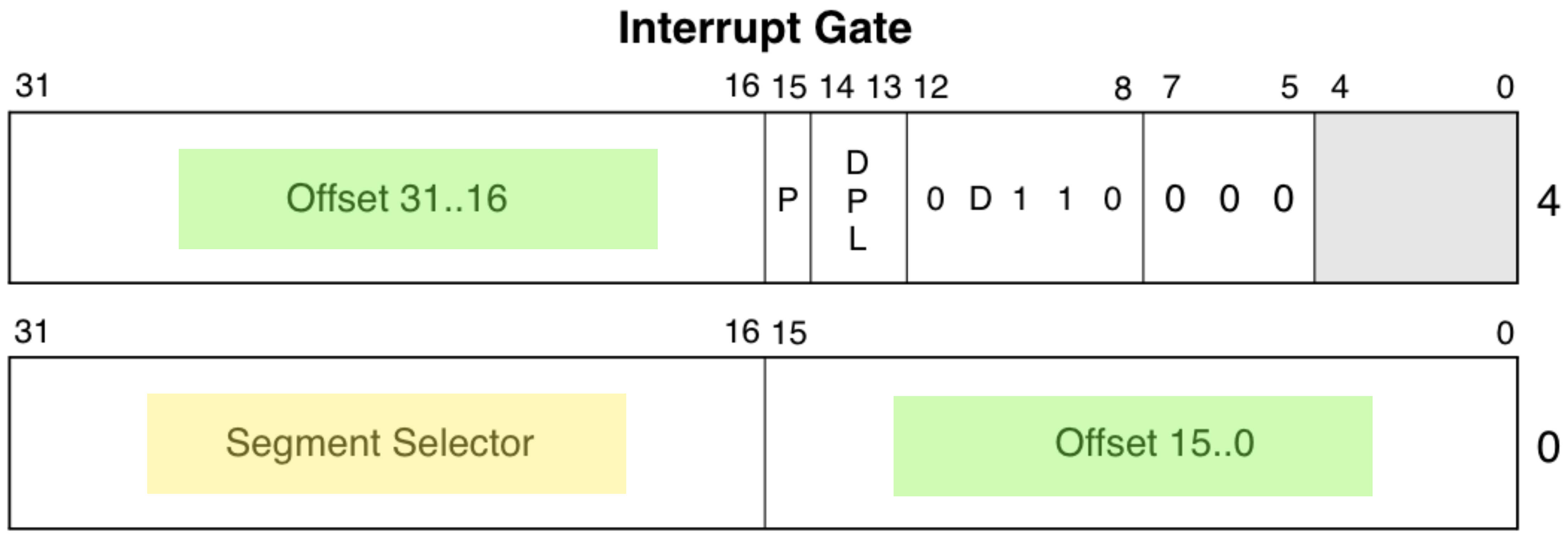
0

Interrupt descriptor table (2)



- Each IDT entry is 64-bits. Contains code segment and eip

Interrupt descriptor table (2)

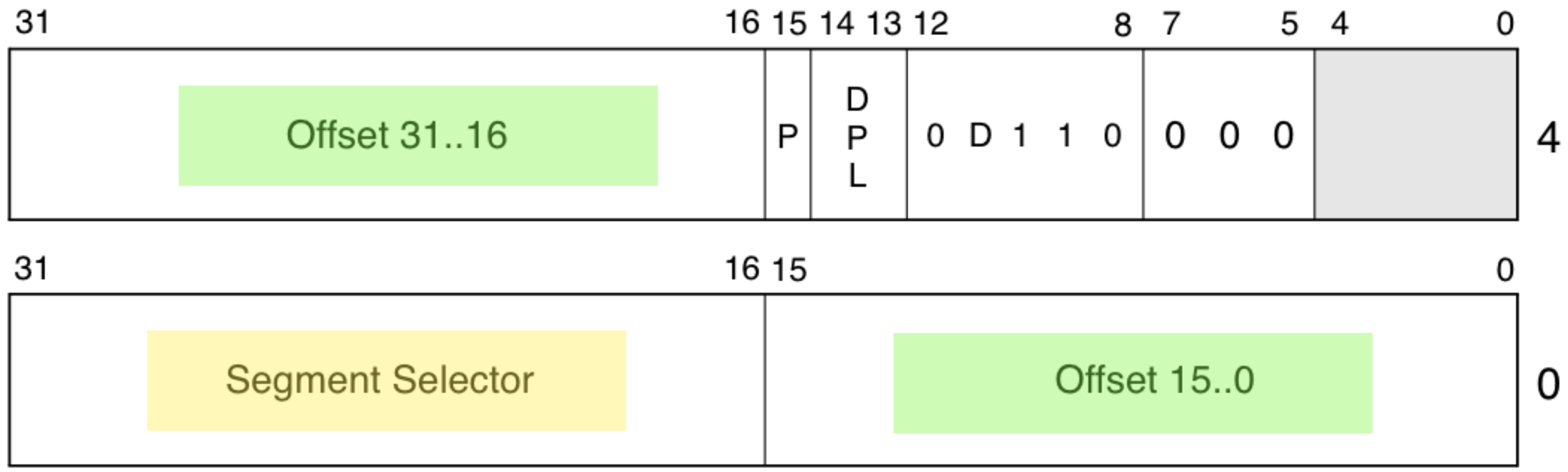


- Each IDT entry is 64-bits. Contains code segment and eip
- When interrupt appears, hardware changes CS and EIP to the one pointed by IDT entry

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

Interrupt Gate



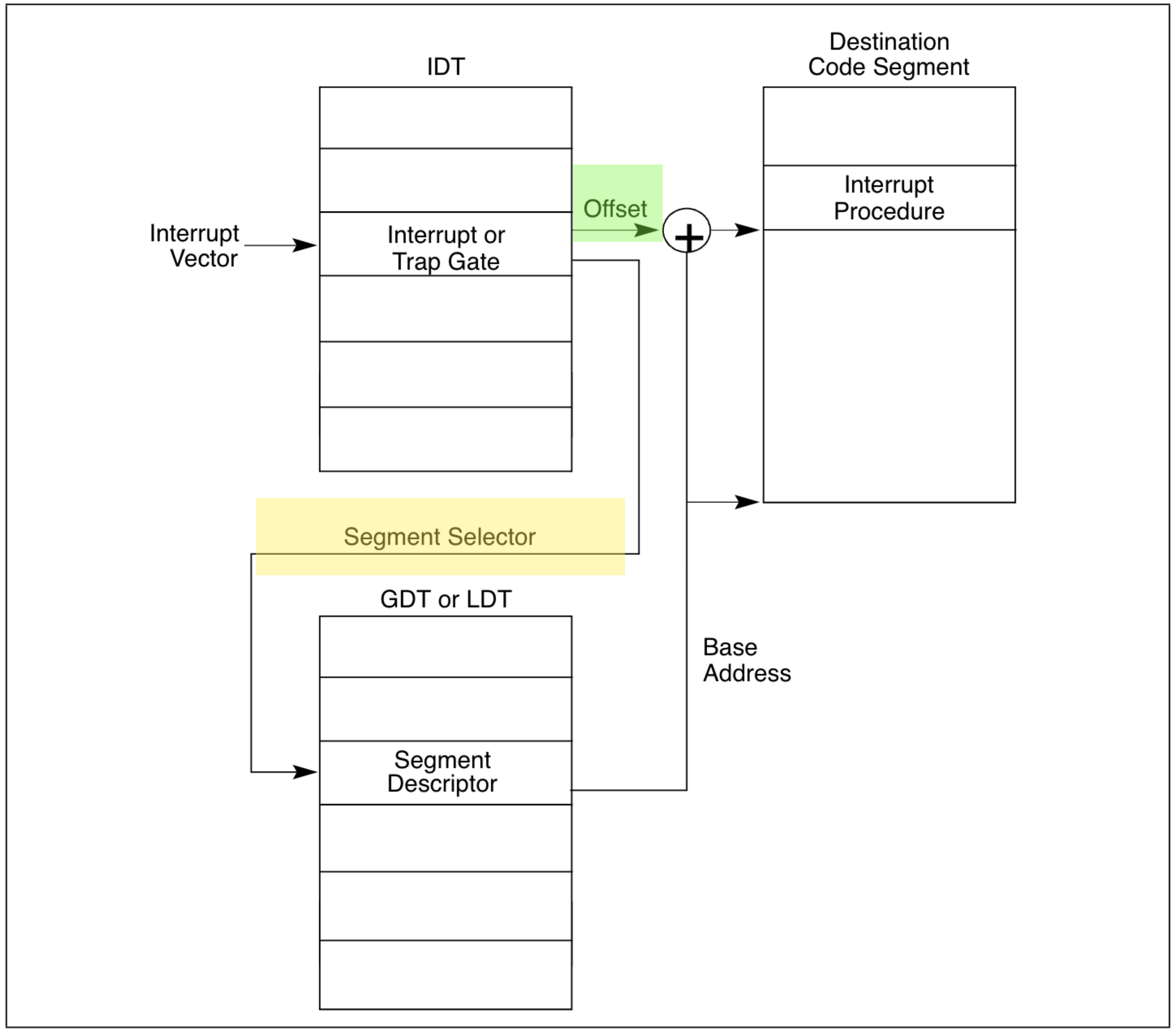
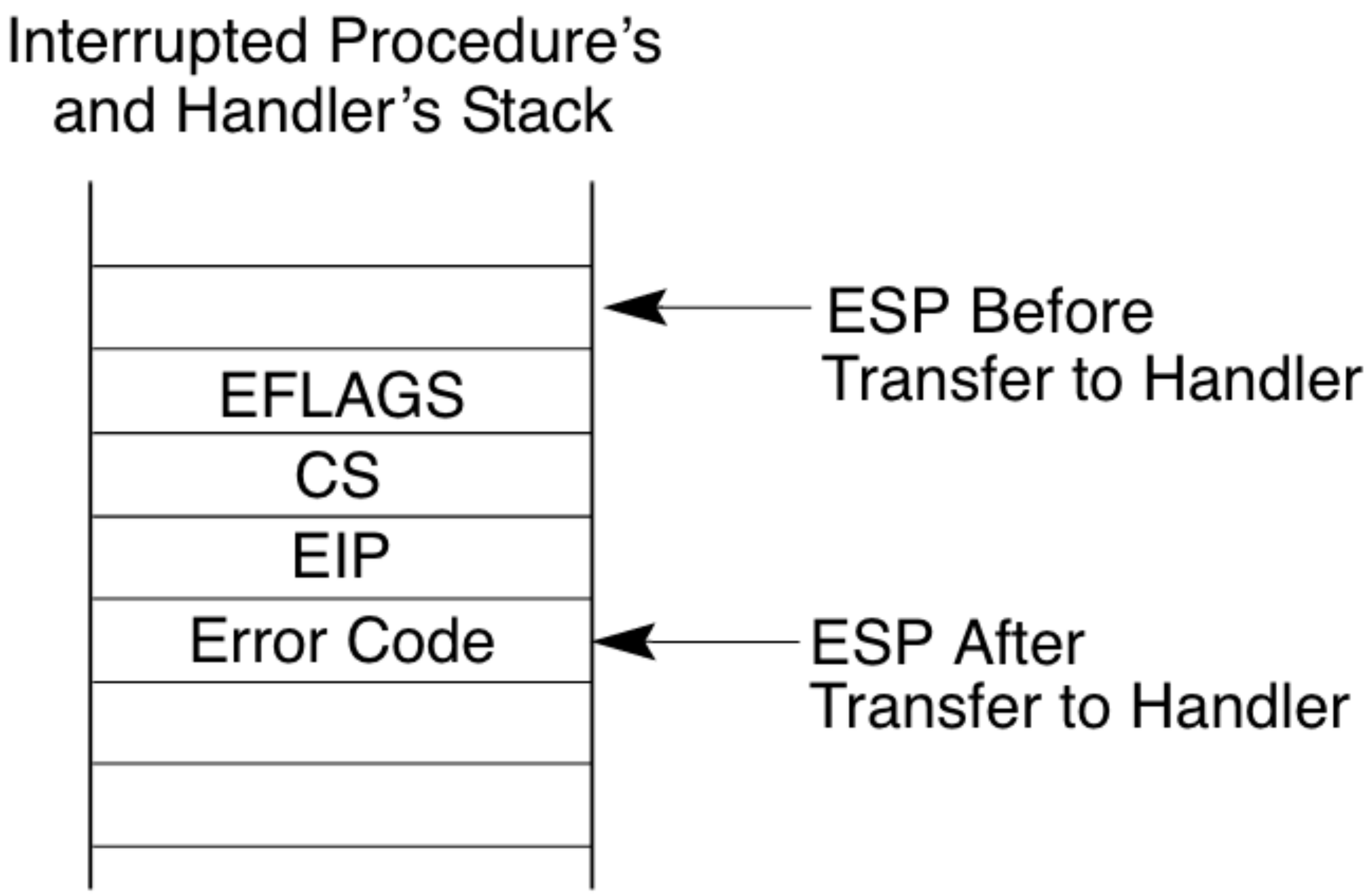


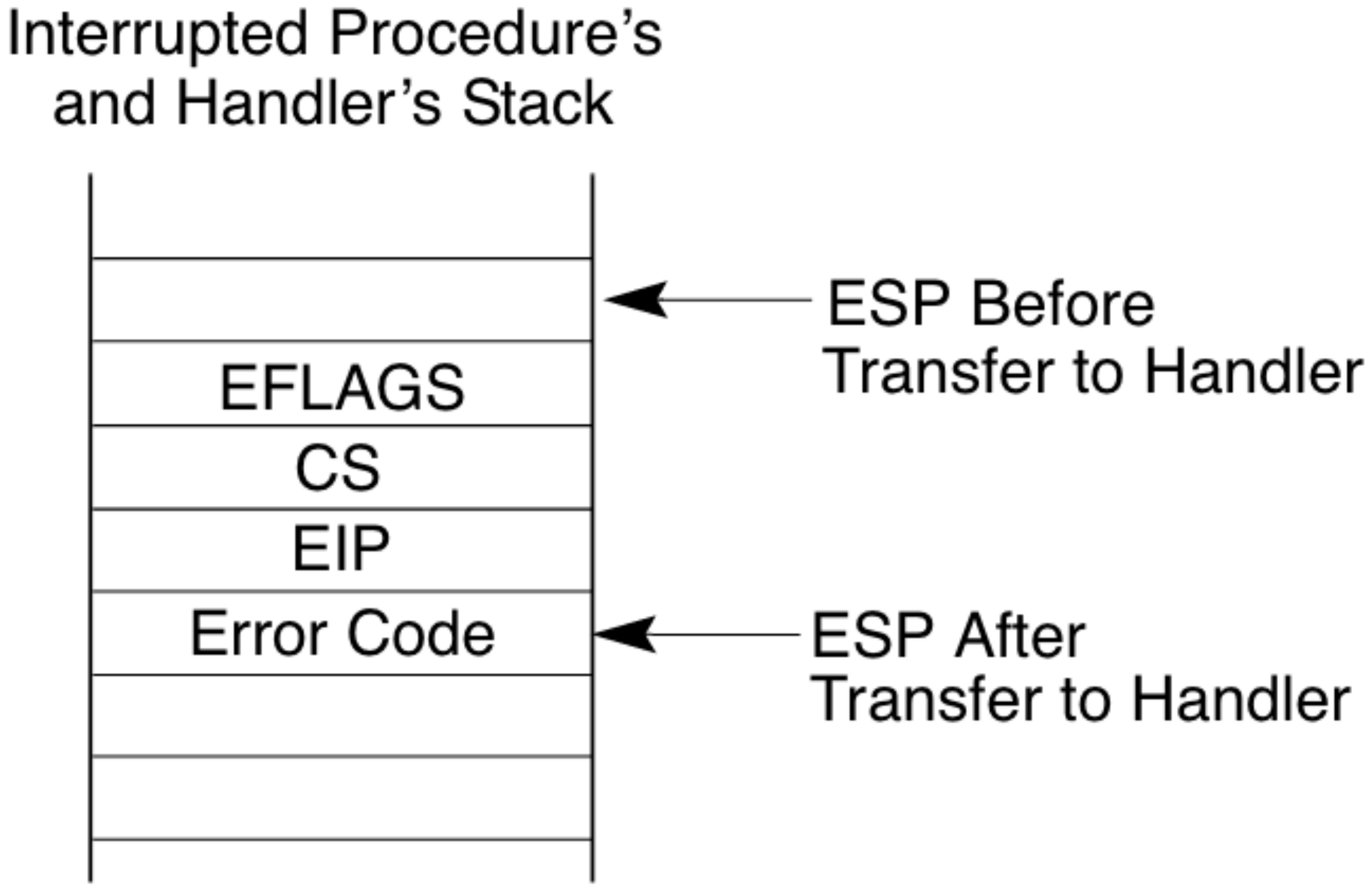
Figure 6-3. Interrupt Procedure Call

Interrupt handling



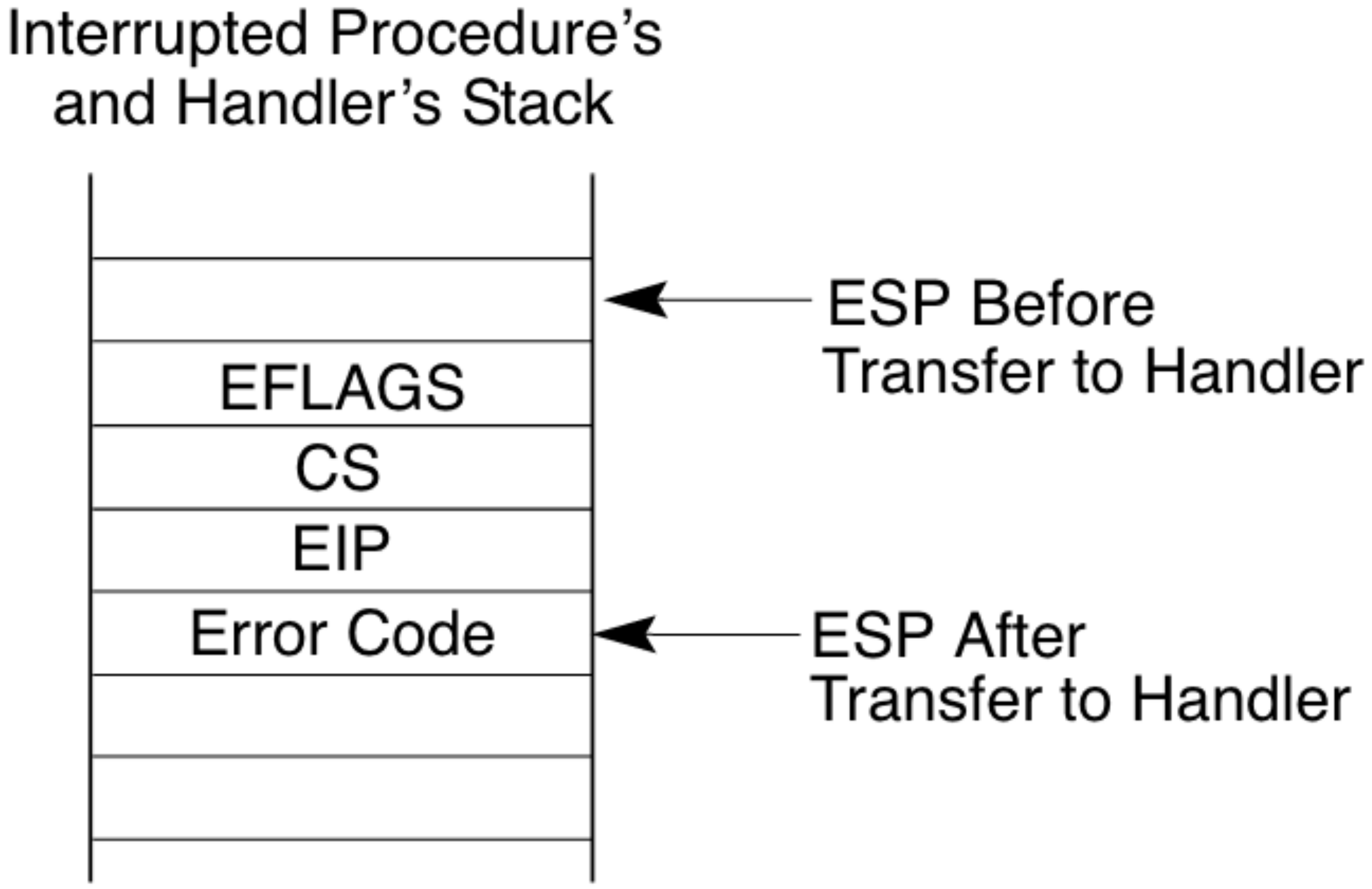
Interrupt handling

- On an interrupt, hardware pushes old EFLAGS, CS and EIP on the stack



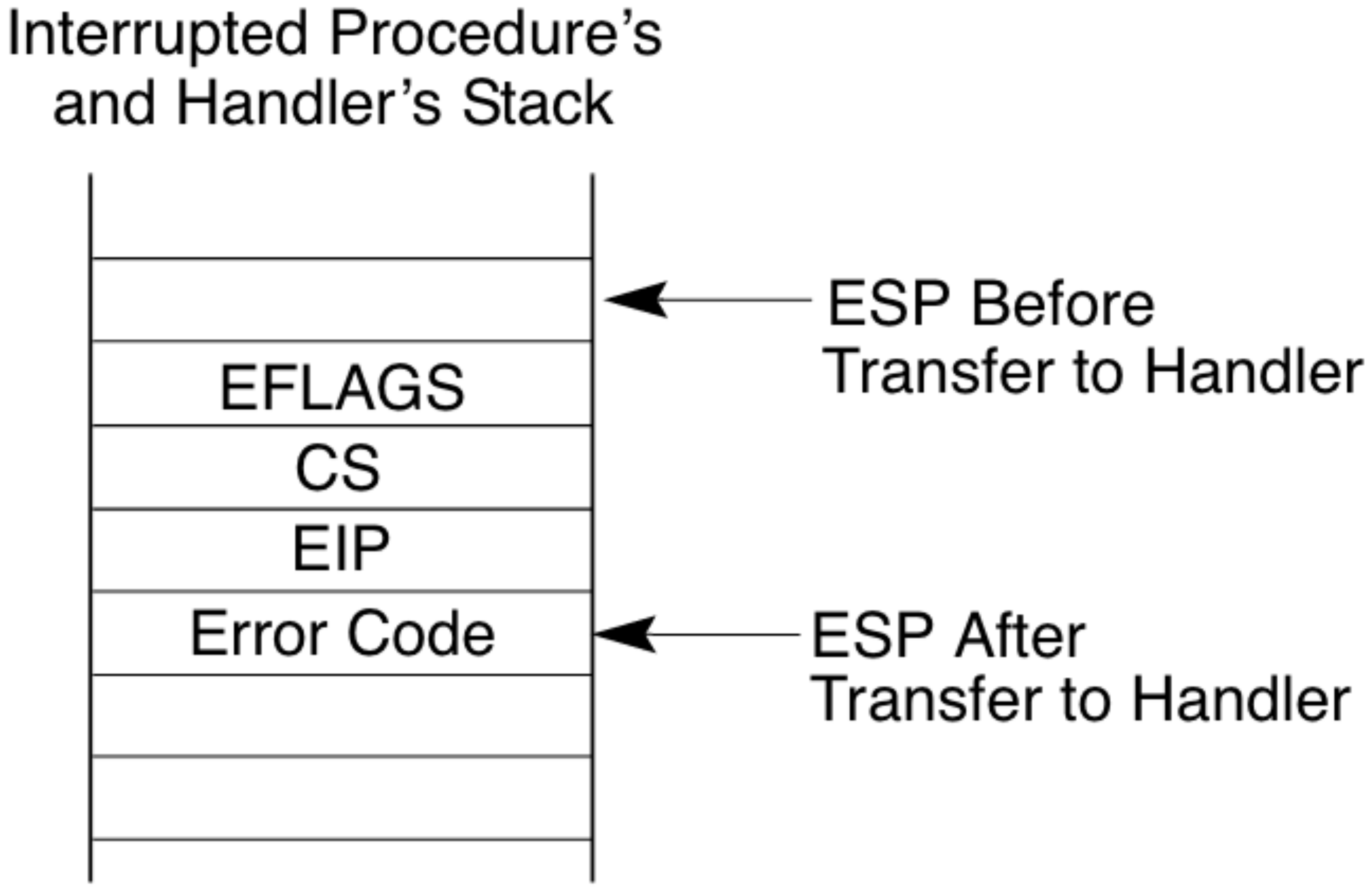
Interrupt handling

- On an interrupt, hardware pushes old EFLAGS, CS and EIP on the stack
- Jumps CS and EIP according to IDT


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

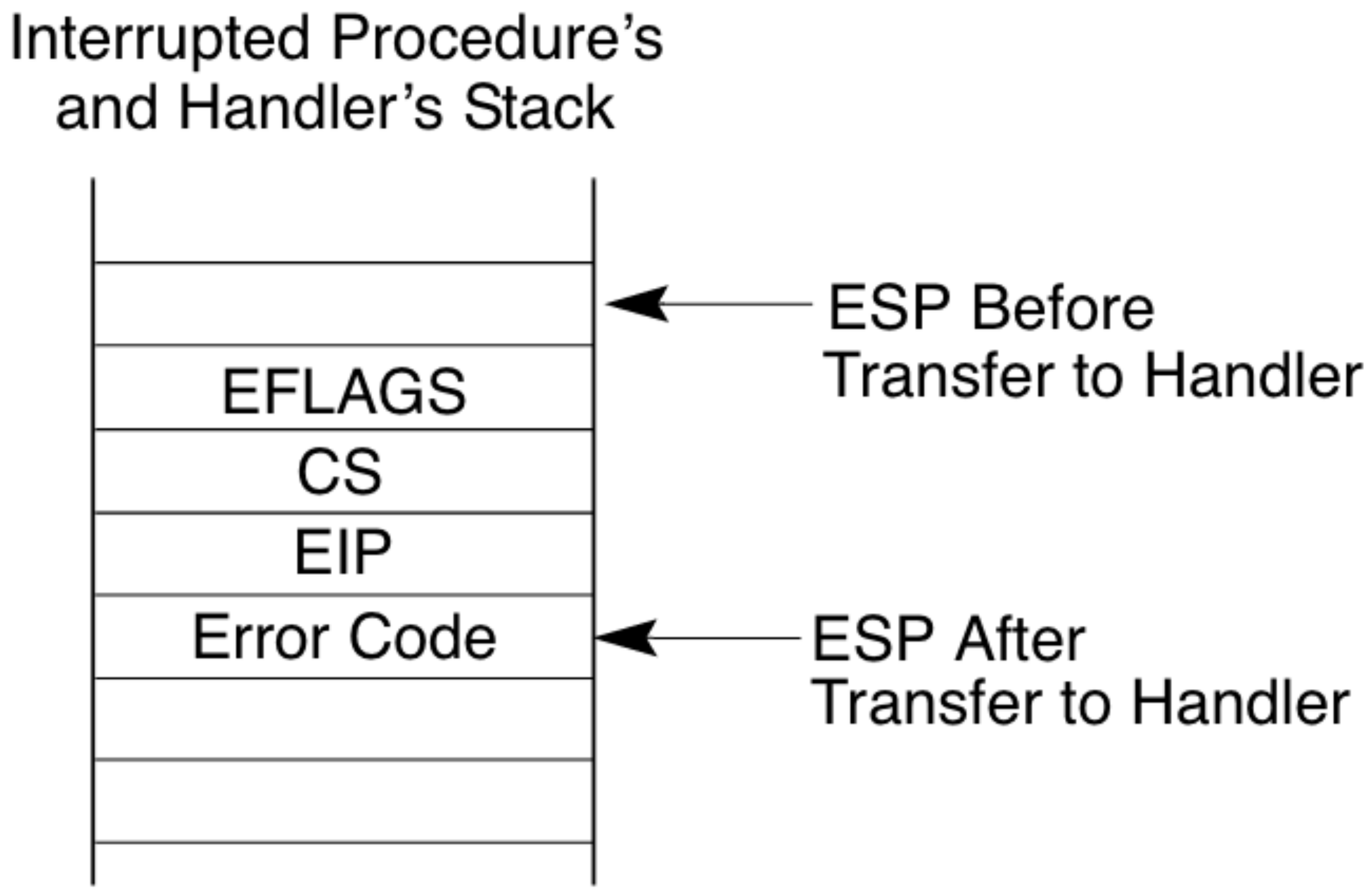
Interrupt handling

- 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


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

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

- 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.

- ‘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

Visualizing interrupt handling

eip

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

vectors.S

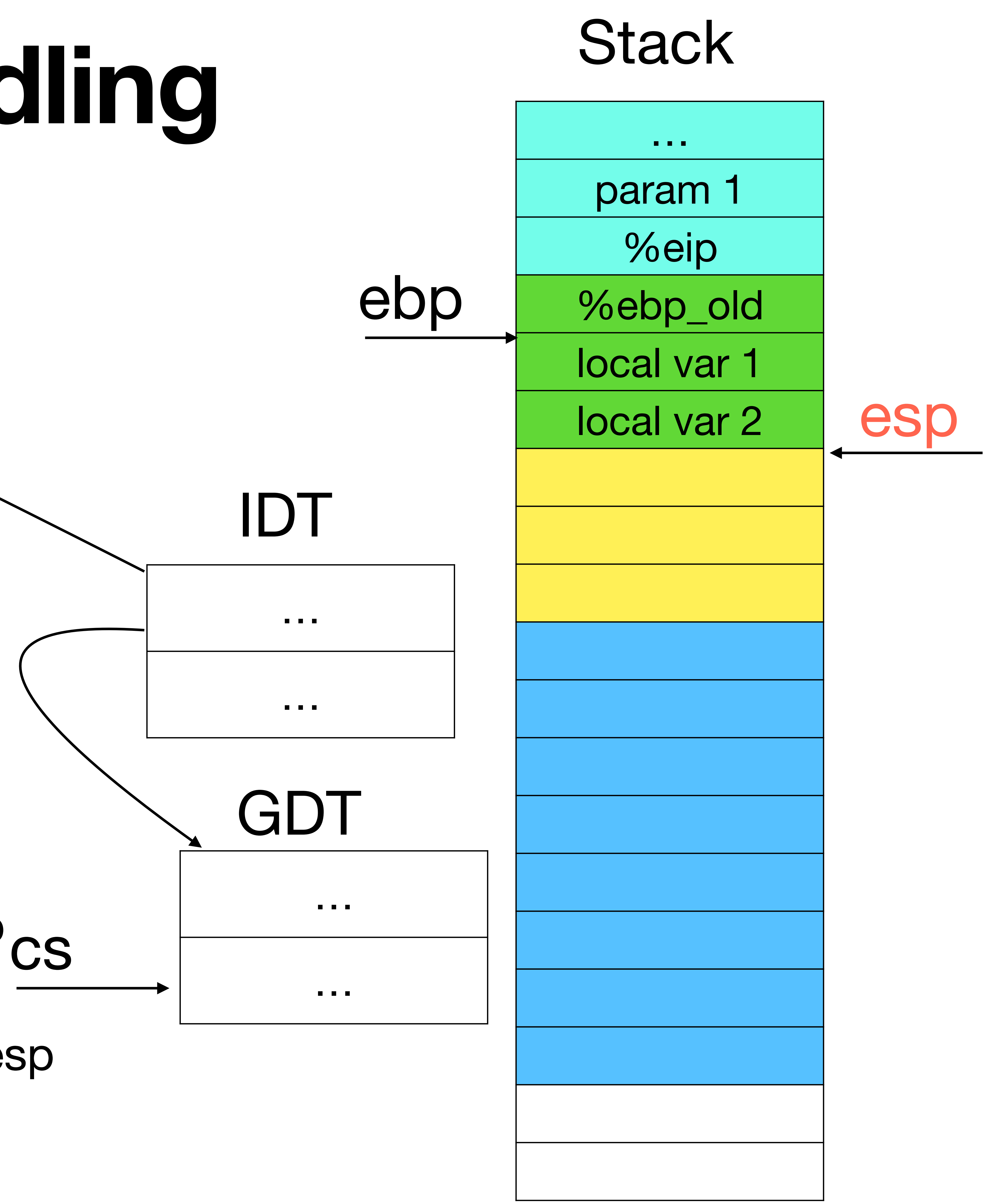
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2

ebp

esp

IDT

...
...

GDT

...
...

cs

Visualizing interrupt handling

eip

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

vectors.S

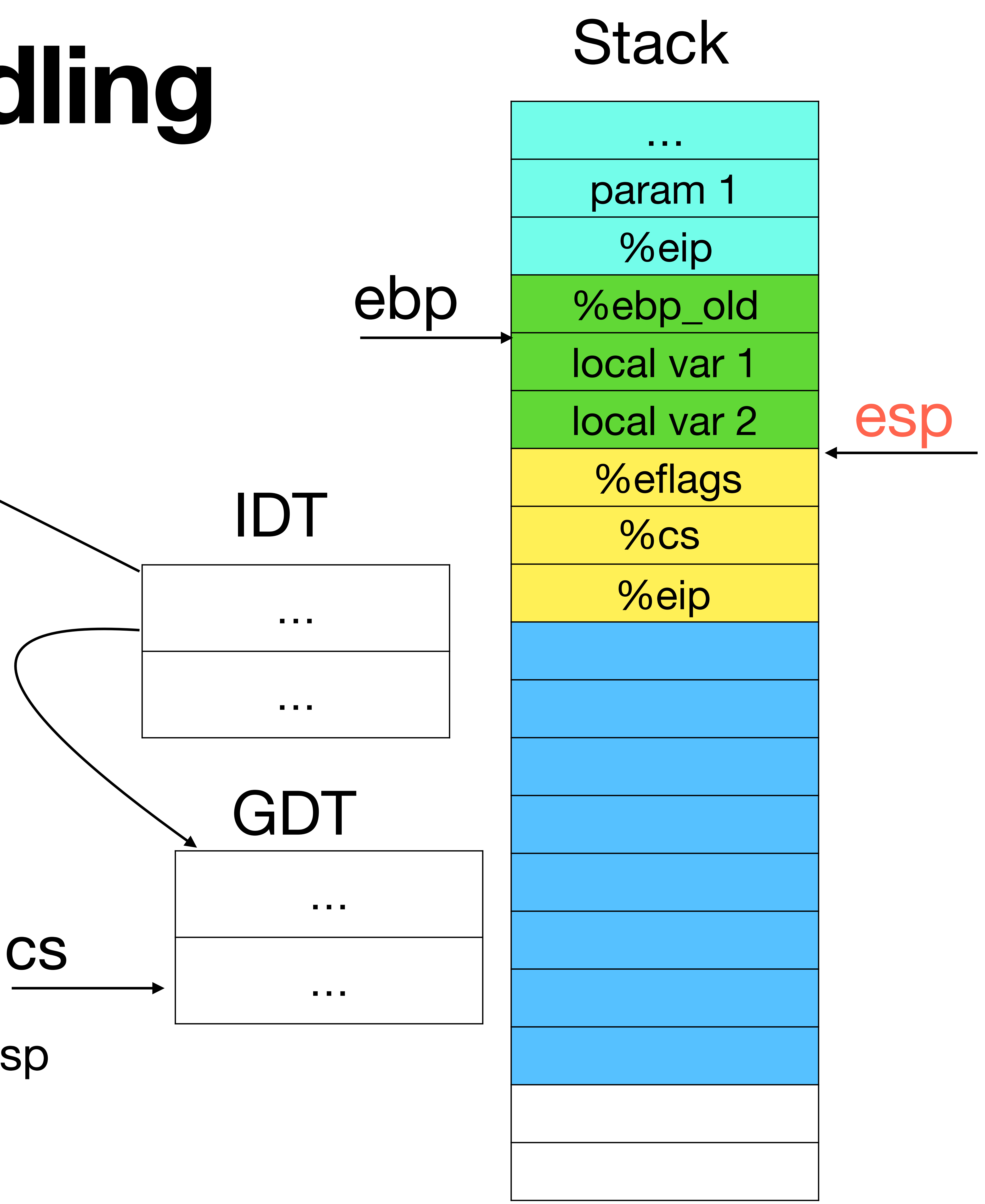
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Visualizing interrupt handling

eip

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

trap.c

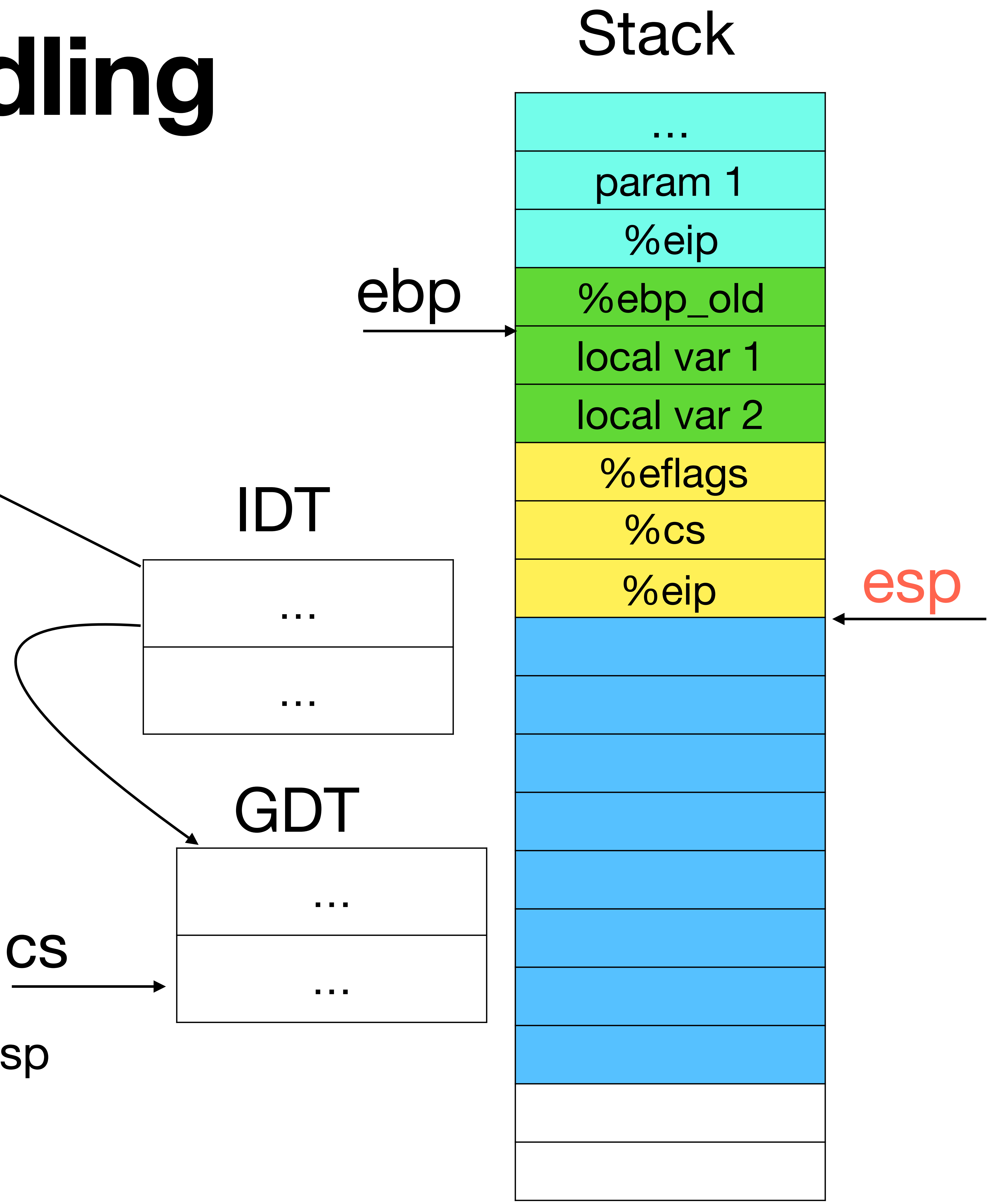
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip

Visualizing interrupt handling

eip

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

vectors.S

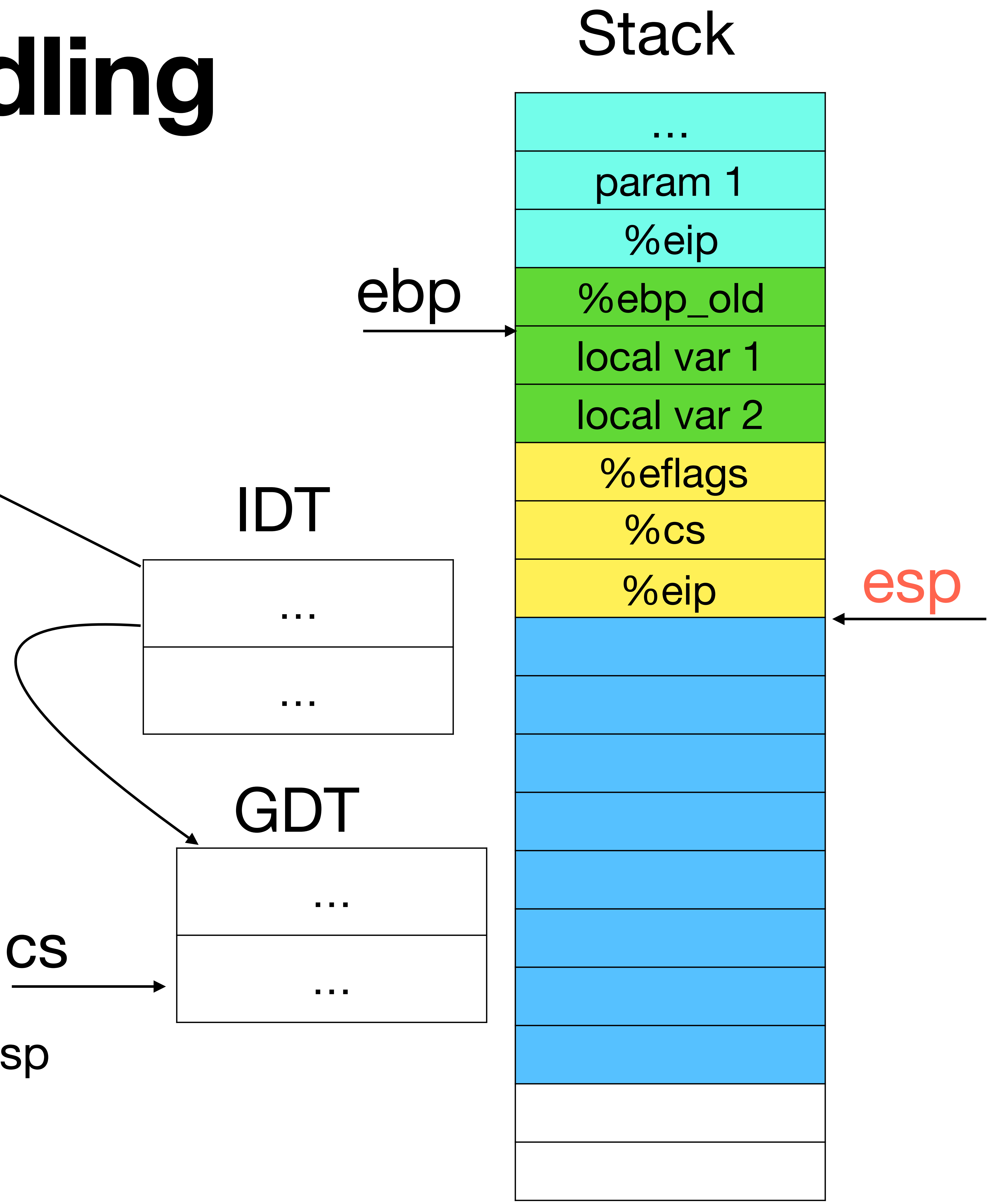
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Visualizing interrupt handling

eip

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

vectors.S

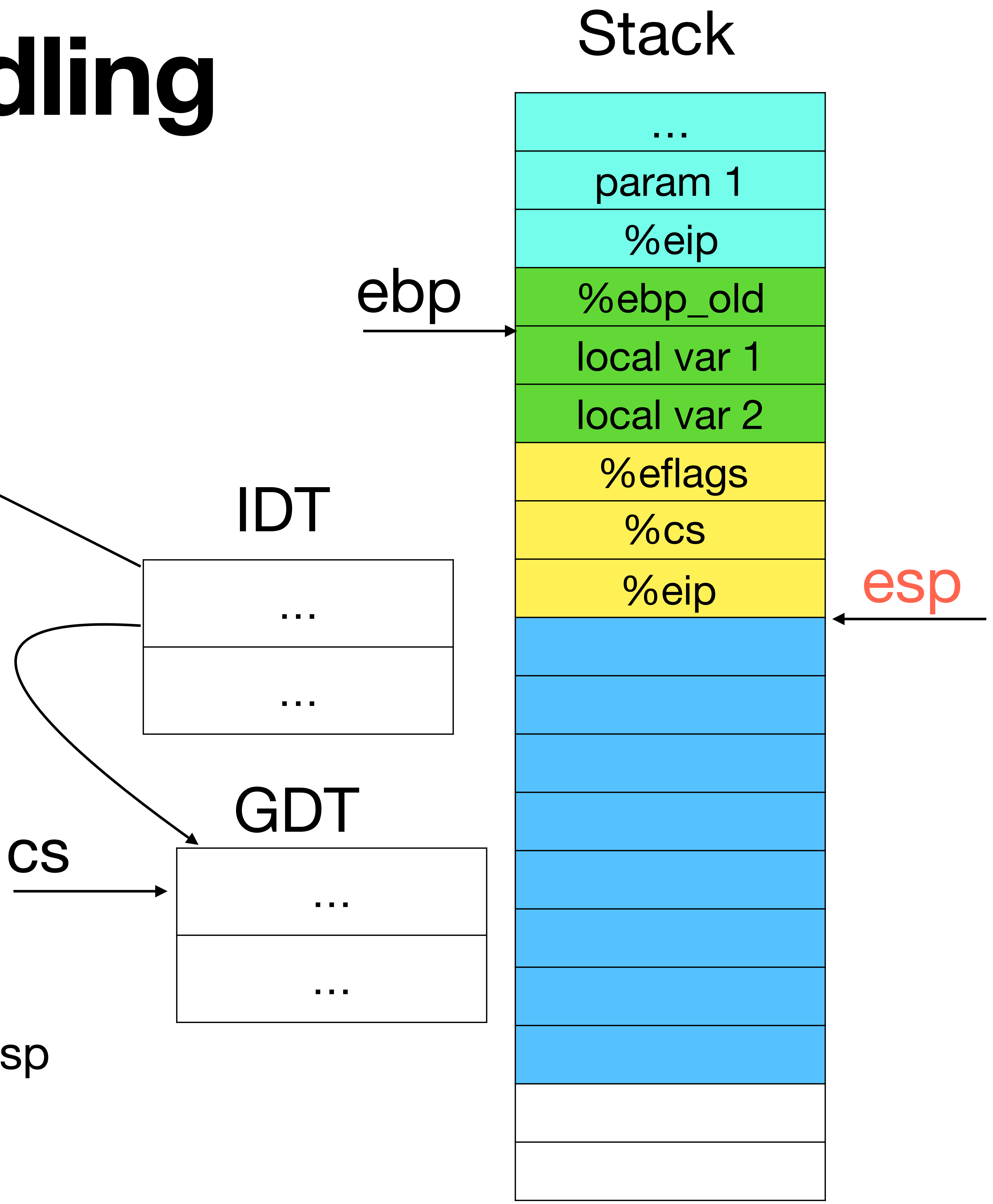
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip

Visualizing interrupt handling

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

vectors.S

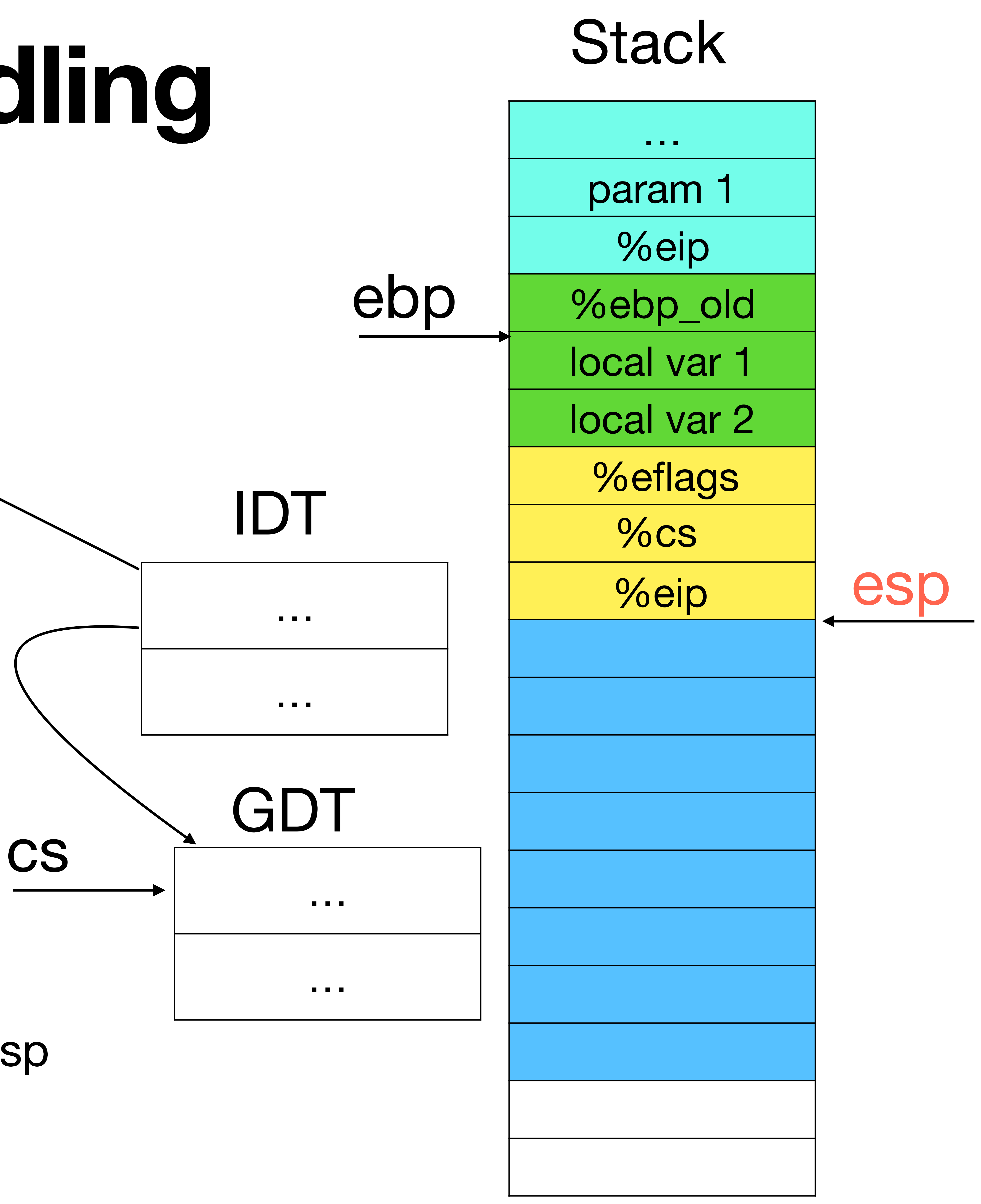
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Visualizing interrupt handling

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

vectors.S

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

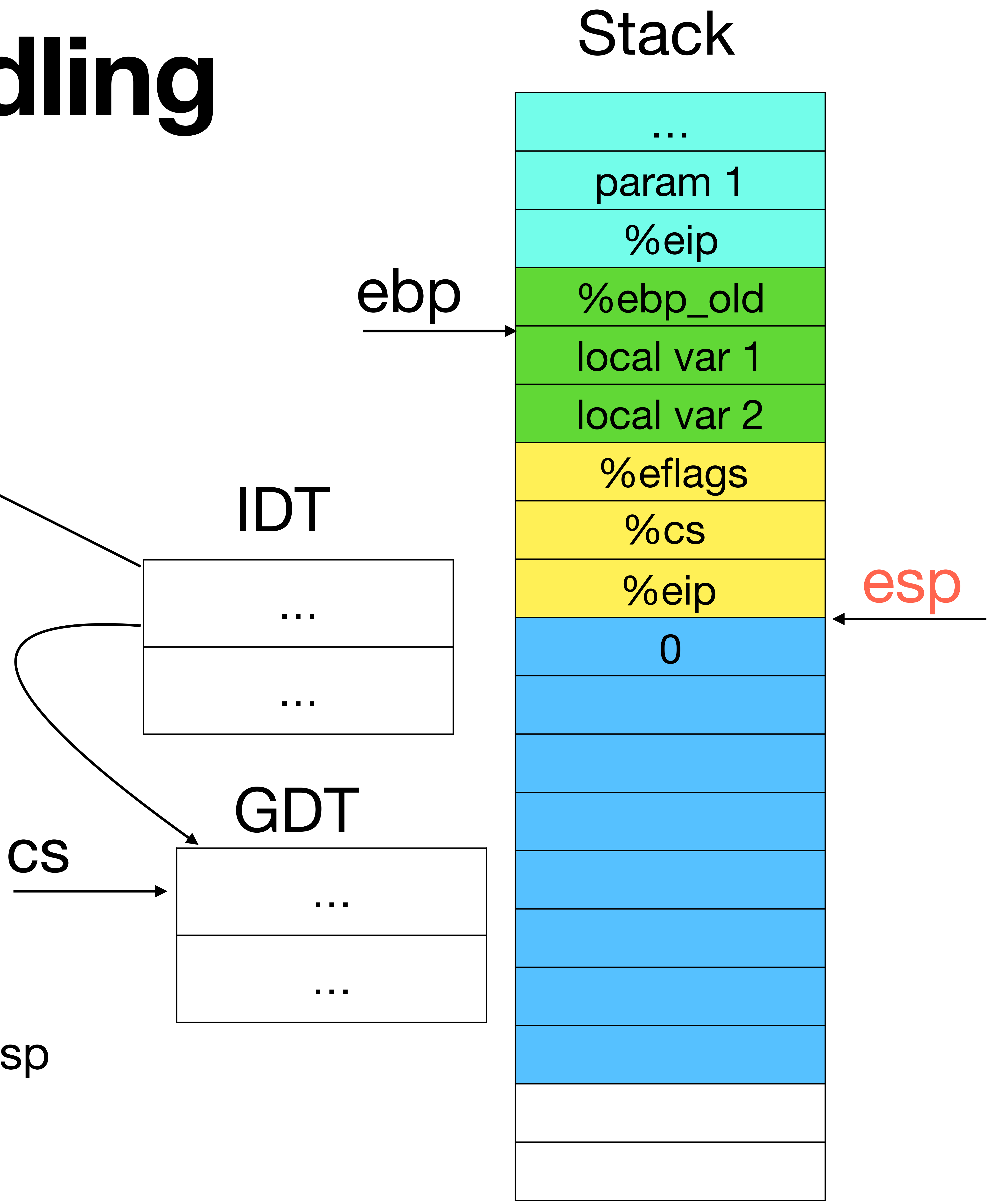
eip

trap.c

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

trapasm.S

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



Visualizing interrupt handling

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

vectors.S

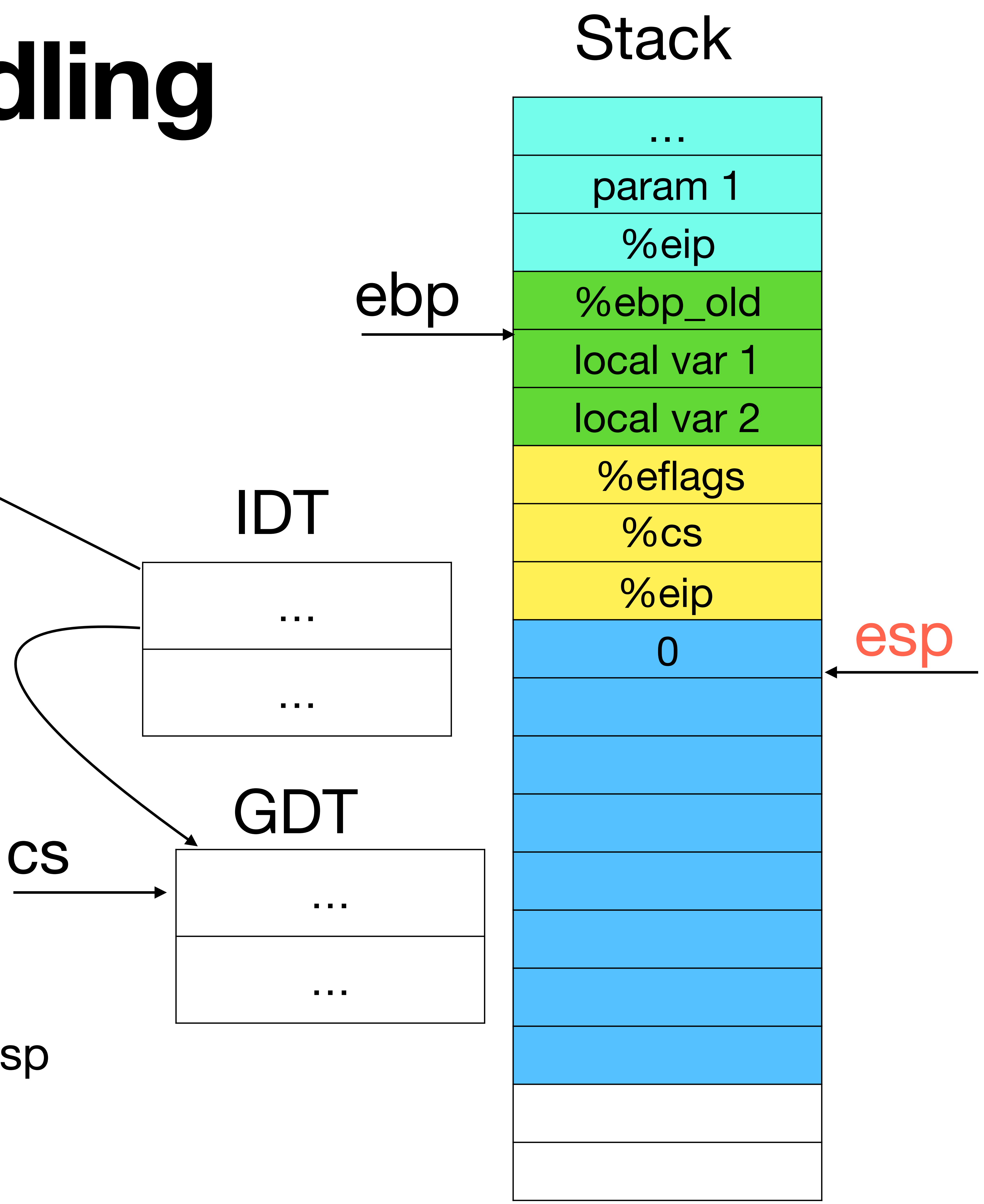
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Visualizing interrupt handling

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

eip →

vectors.S

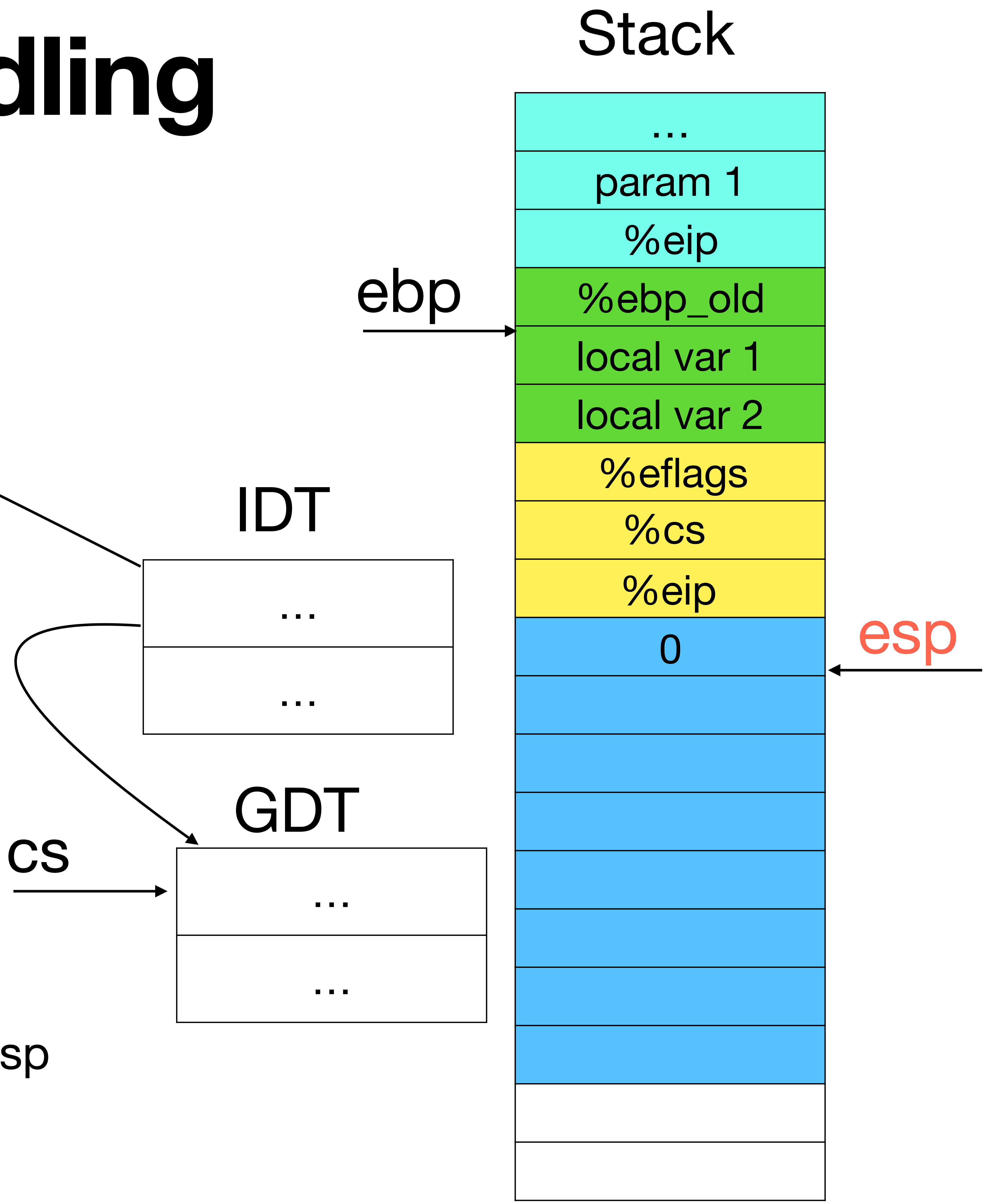
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



IDT

...

...

GDT

CS →

...

...

Stack

ebp →

esp →

Stack
...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0

Visualizing interrupt handling

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

eip

vectors.S

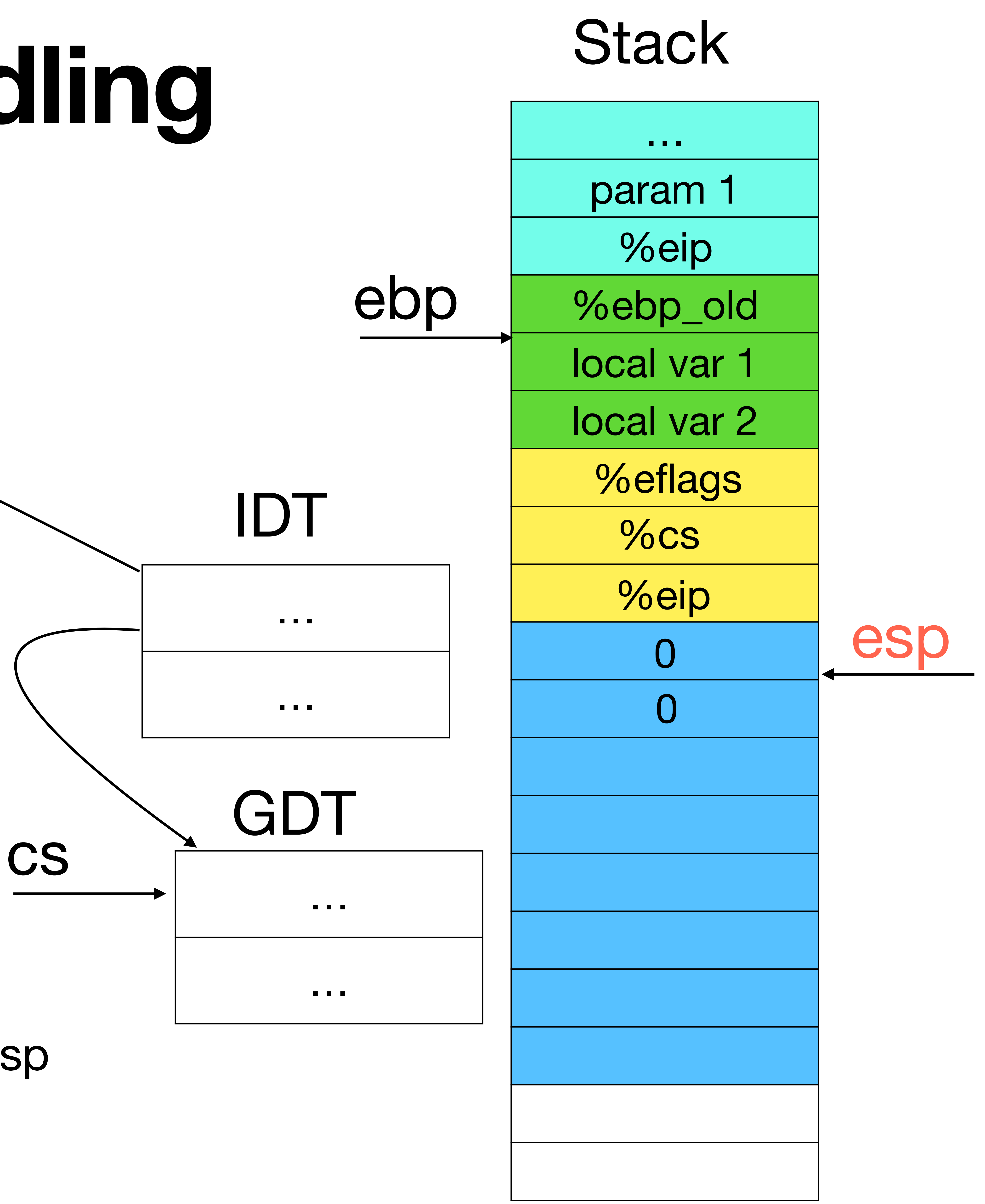
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



IDT:

…
…

GDT:

…
…

Stack:

…
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0

Visualizing interrupt handling

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

trap.c

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

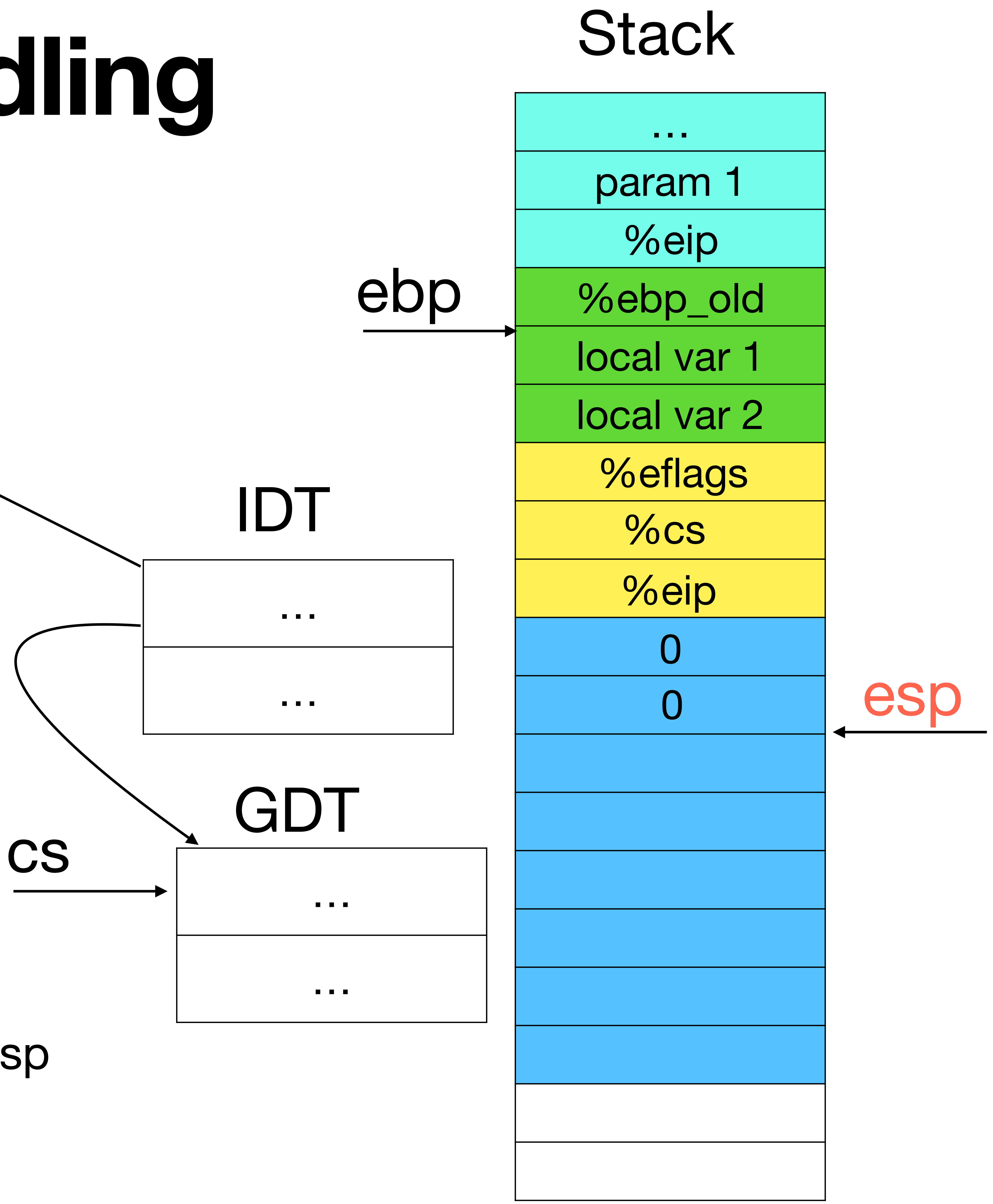
vectors.S

eip →

```
.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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0

Visualizing interrupt handling

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

trap.c

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

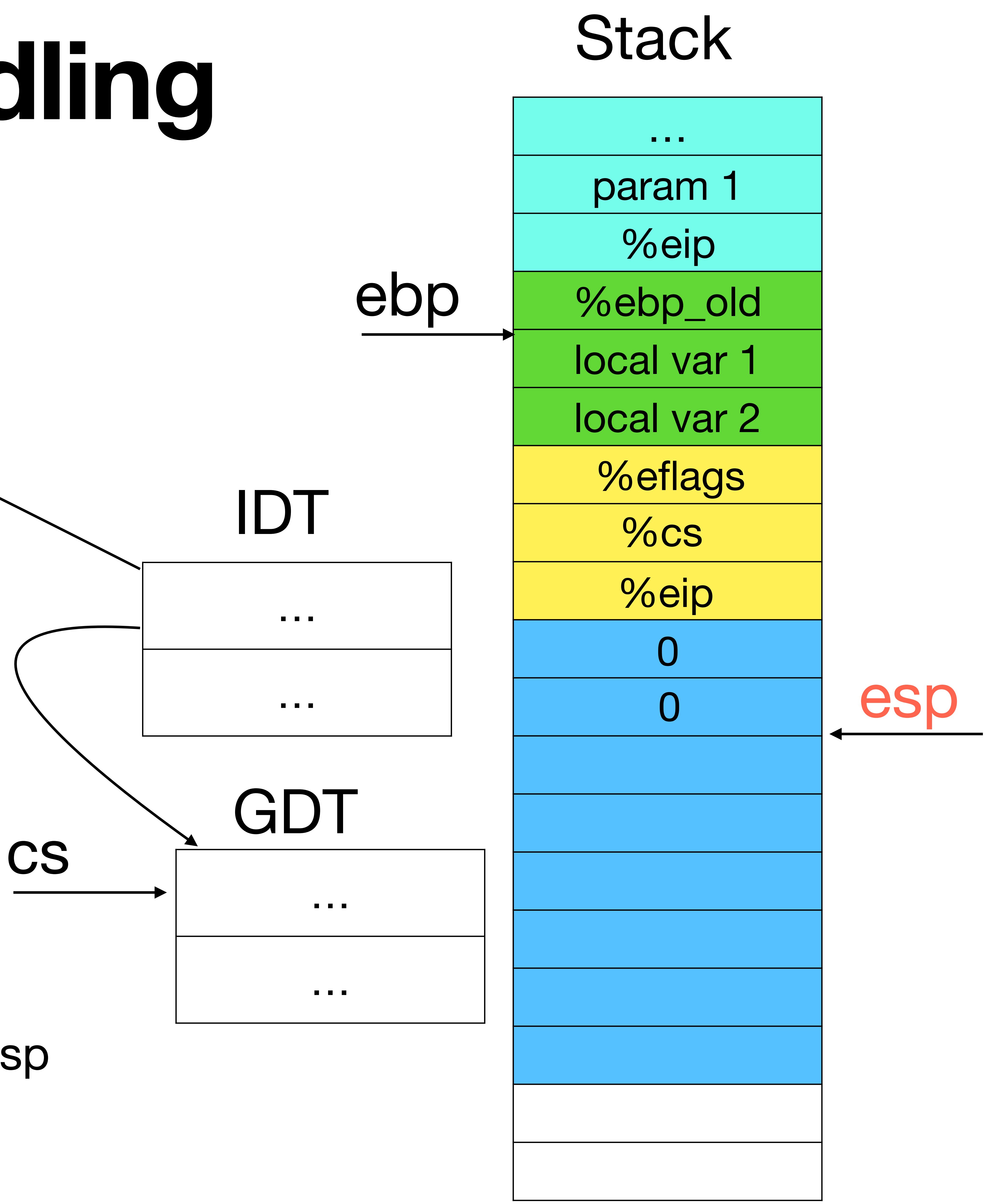
vectors.S

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

eip

trapasm.S

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



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0

Visualizing interrupt handling

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

trap.c

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

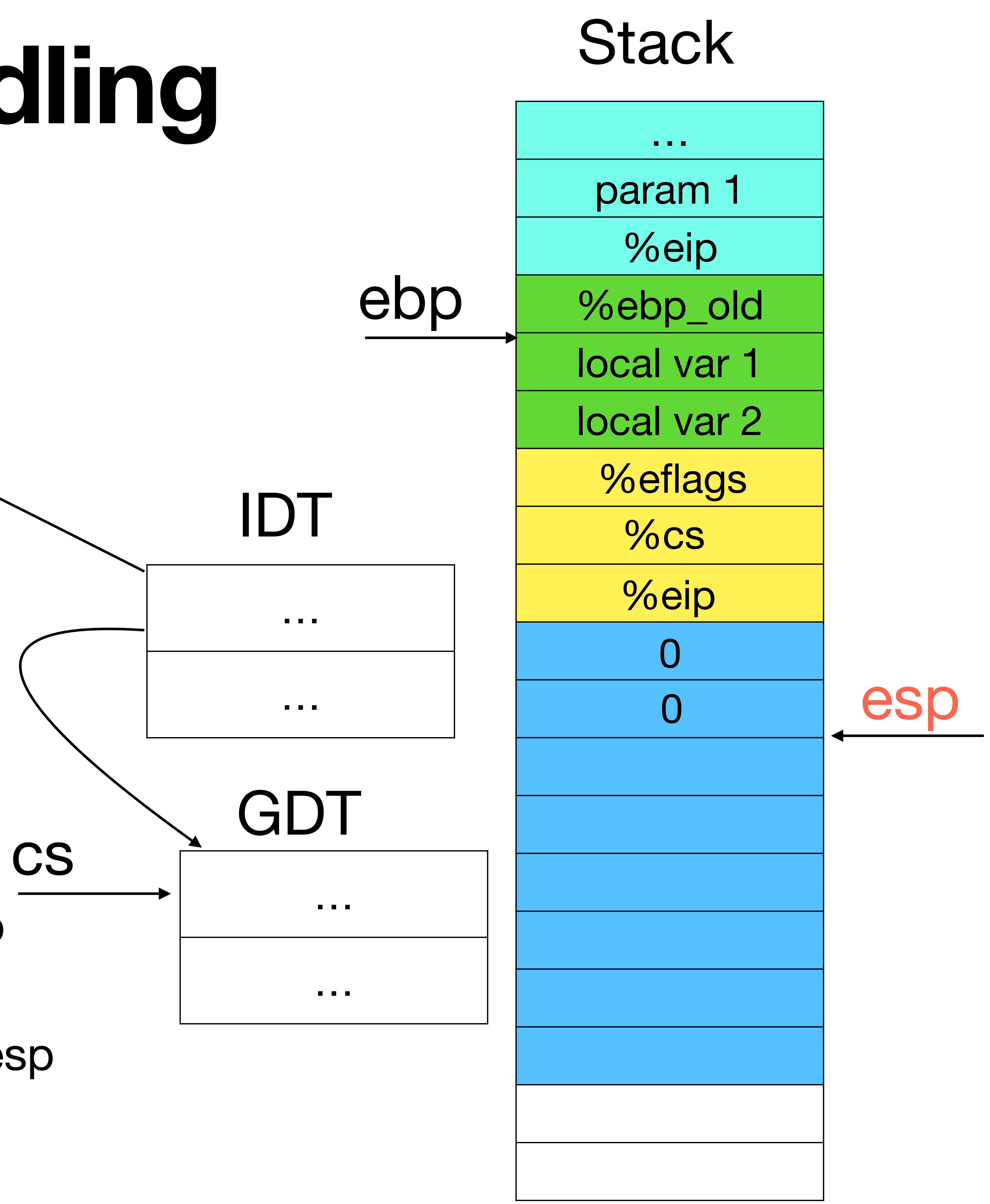
vectors.S

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

trapasm.S

eip

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



Visualizing interrupt handling

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

trap.c

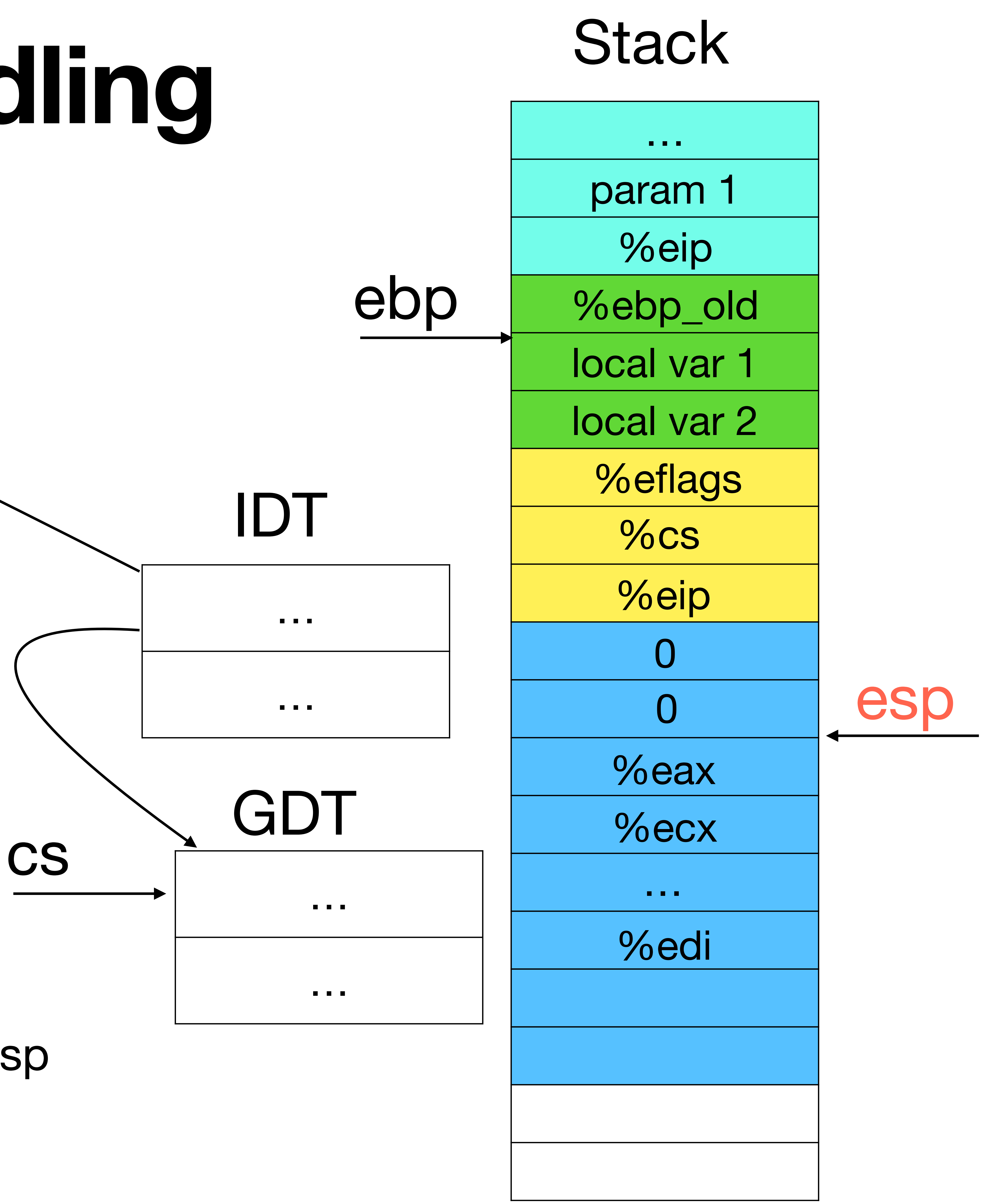
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi

Visualizing interrupt handling

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

trap.c

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

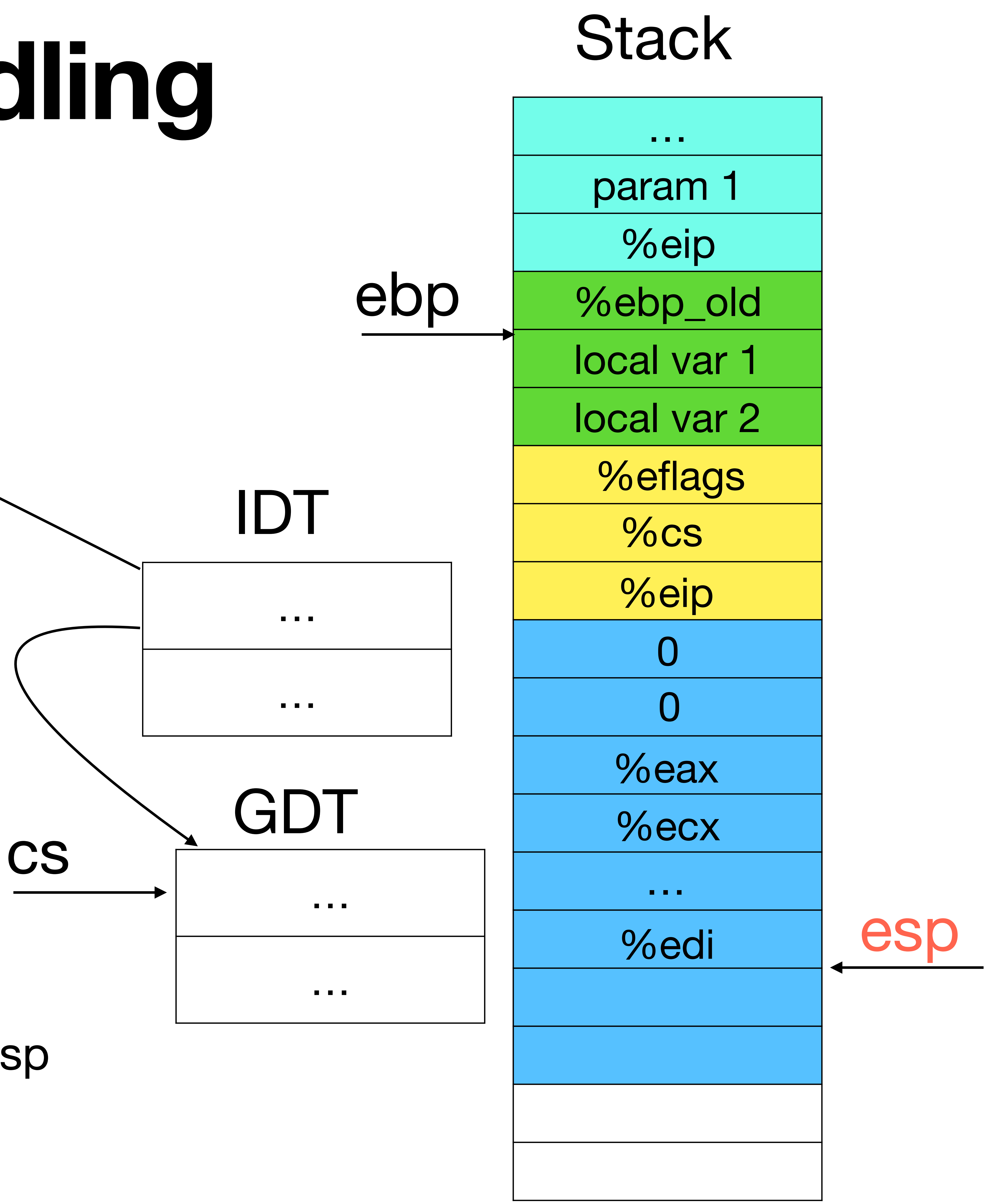
vectors.S

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

trapasm.S

eip

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



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi

Visualizing interrupt handling

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

trap.c

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

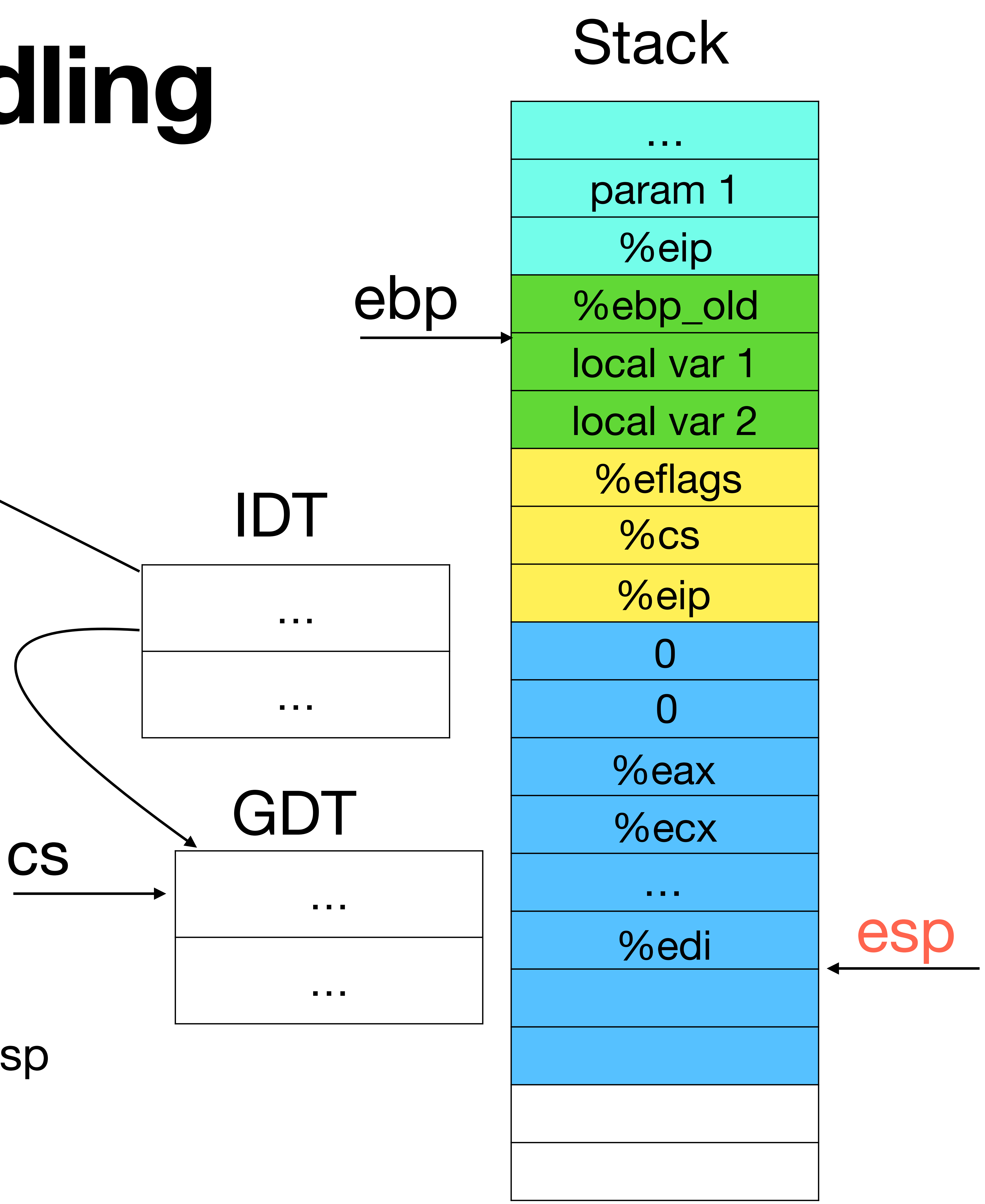
vectors.S

```
.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
```

eip



Stack

- …
- param 1
- %eip
- %ebp_old
- local var 1
- local var 2
- %eflags
- %cs
- %eip
- 0
- 0
- %eax
- %ecx
- …
- %edi

Visualizing interrupt handling

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

trap.c

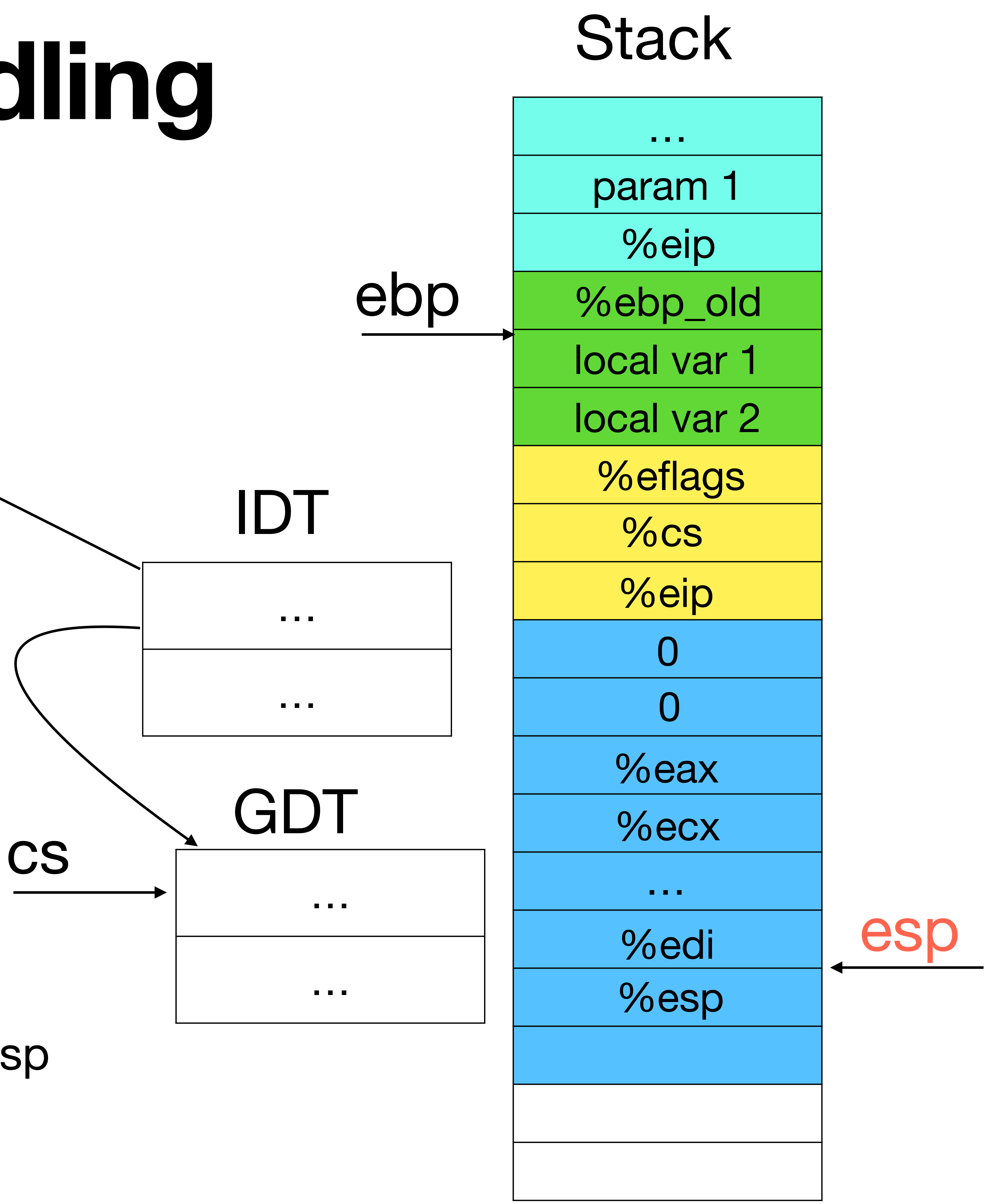
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp

Visualizing interrupt handling

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

trap.c

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

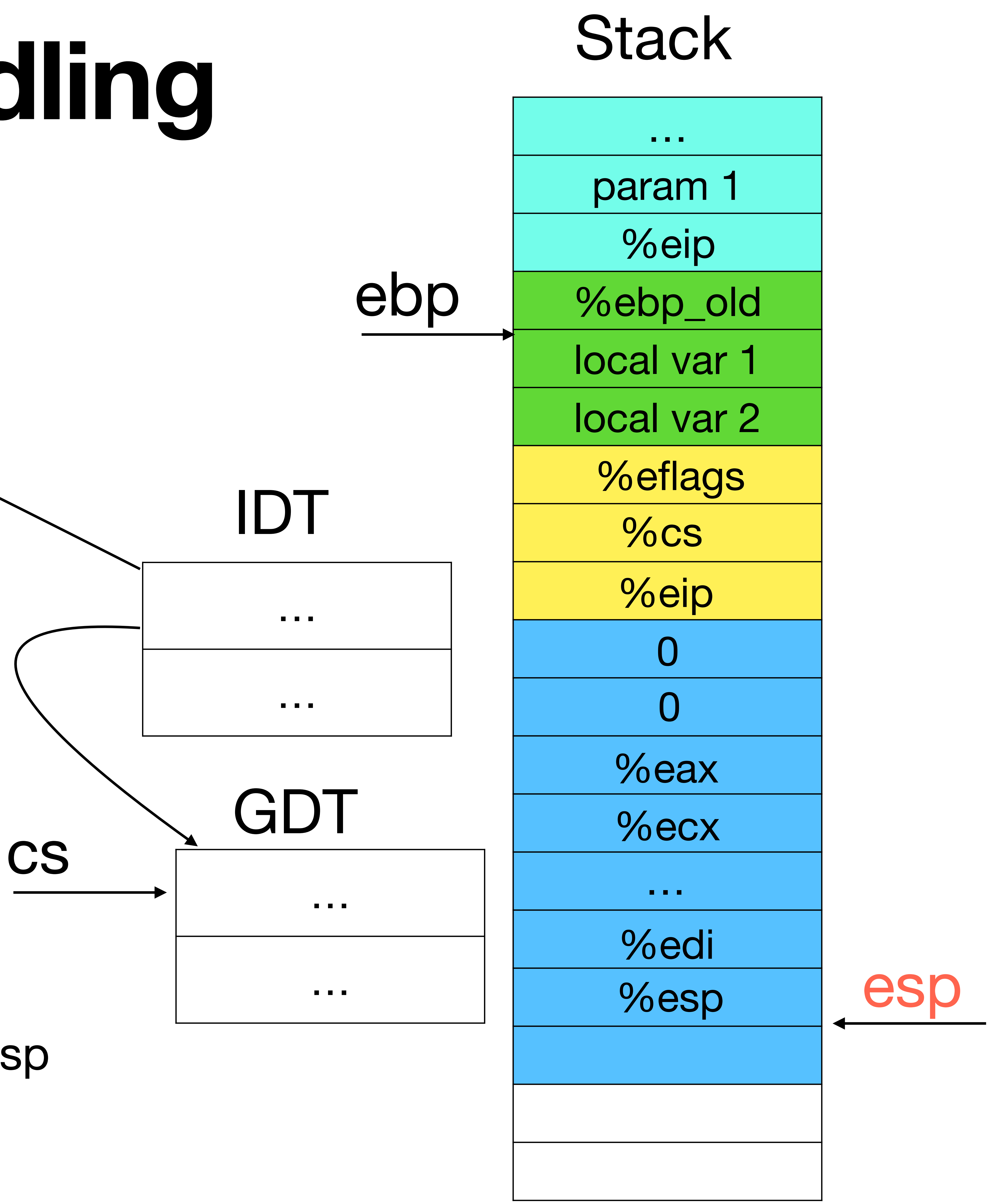
vectors.S

```
.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
```

eip



Stack
...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp

Visualizing interrupt handling

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

trap.c

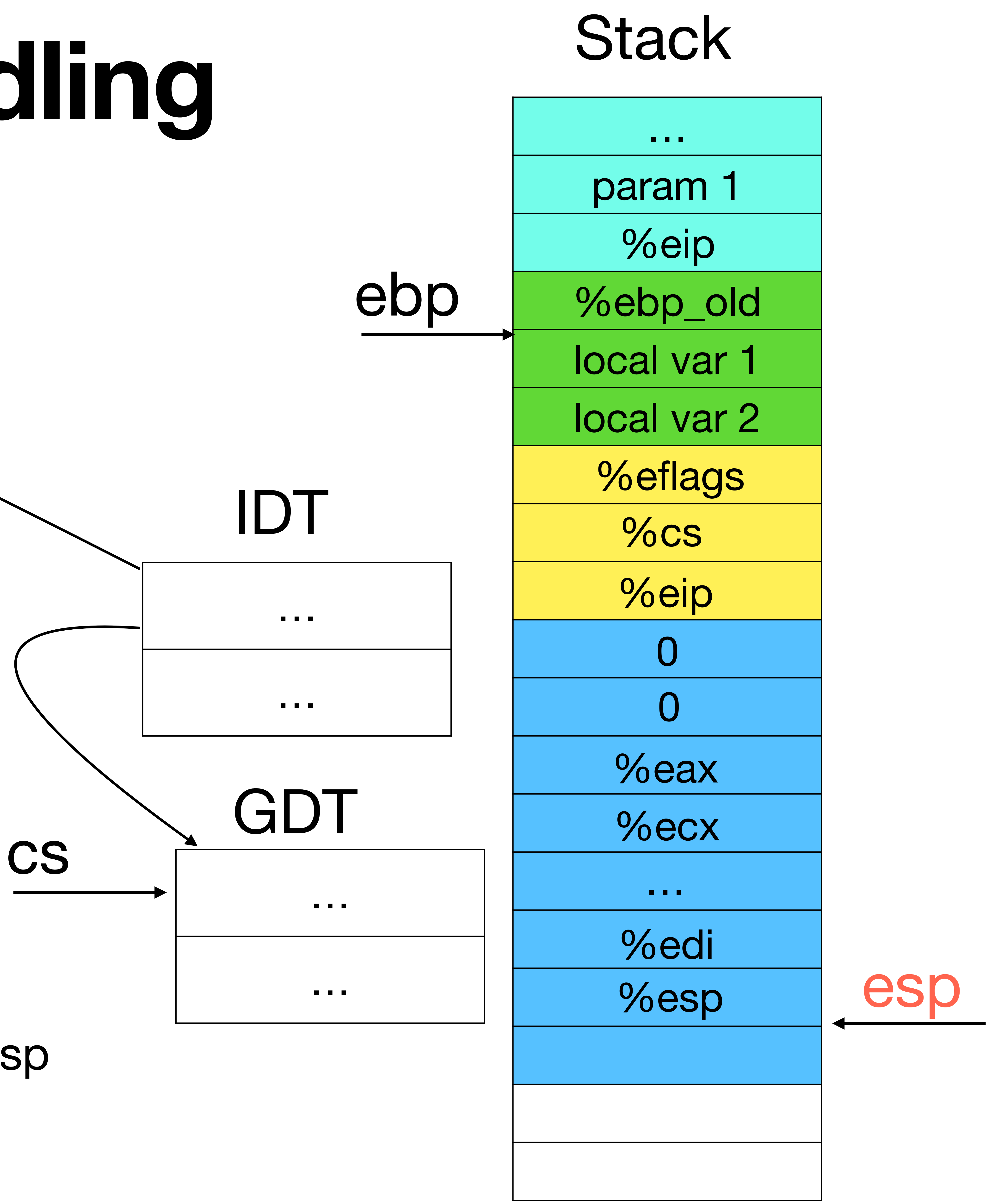
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp

Visualizing interrupt handling

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

vectors.S

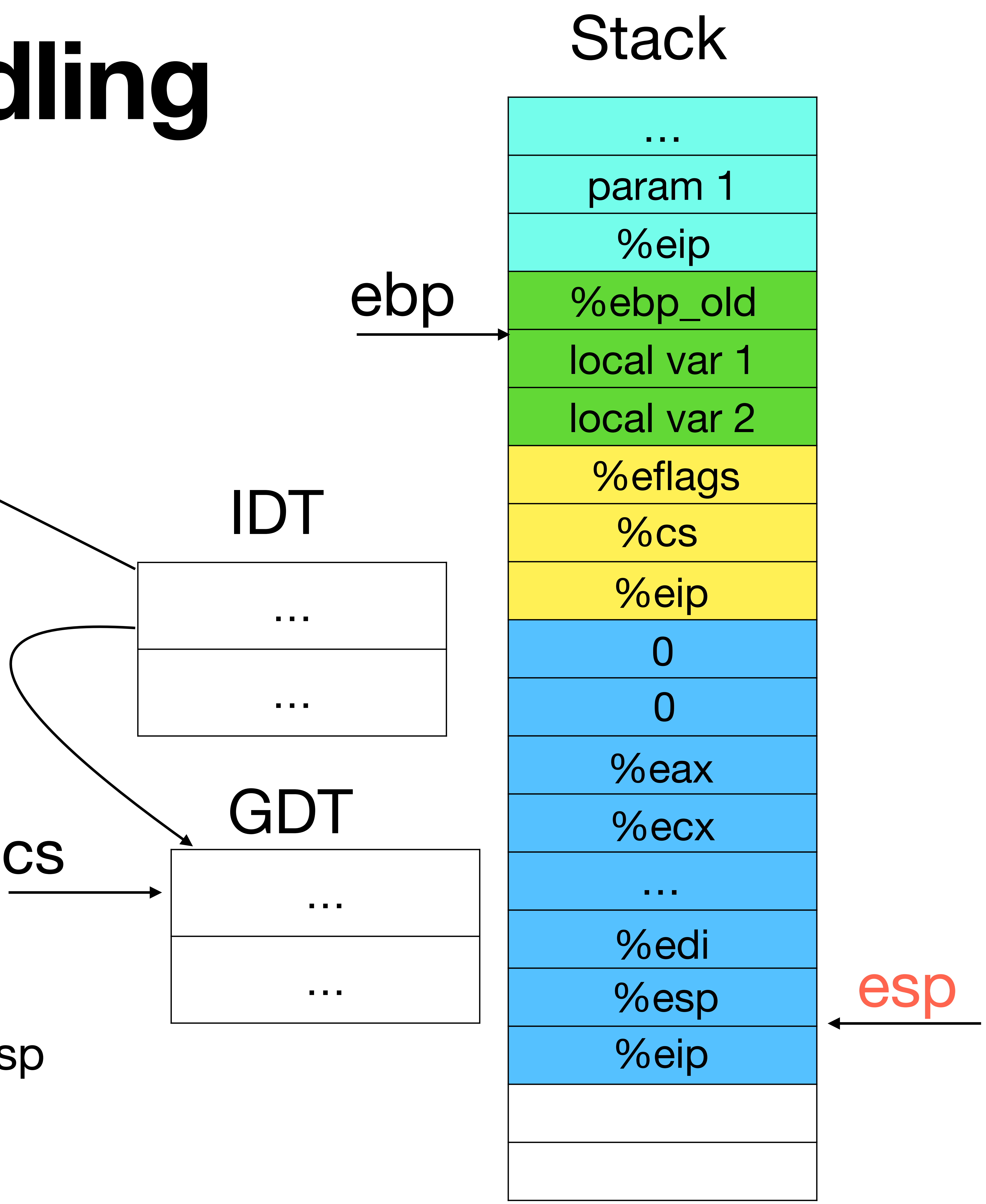
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



…
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
…
%edi
%esp
%eip

Visualizing interrupt handling

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

vectors.S

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

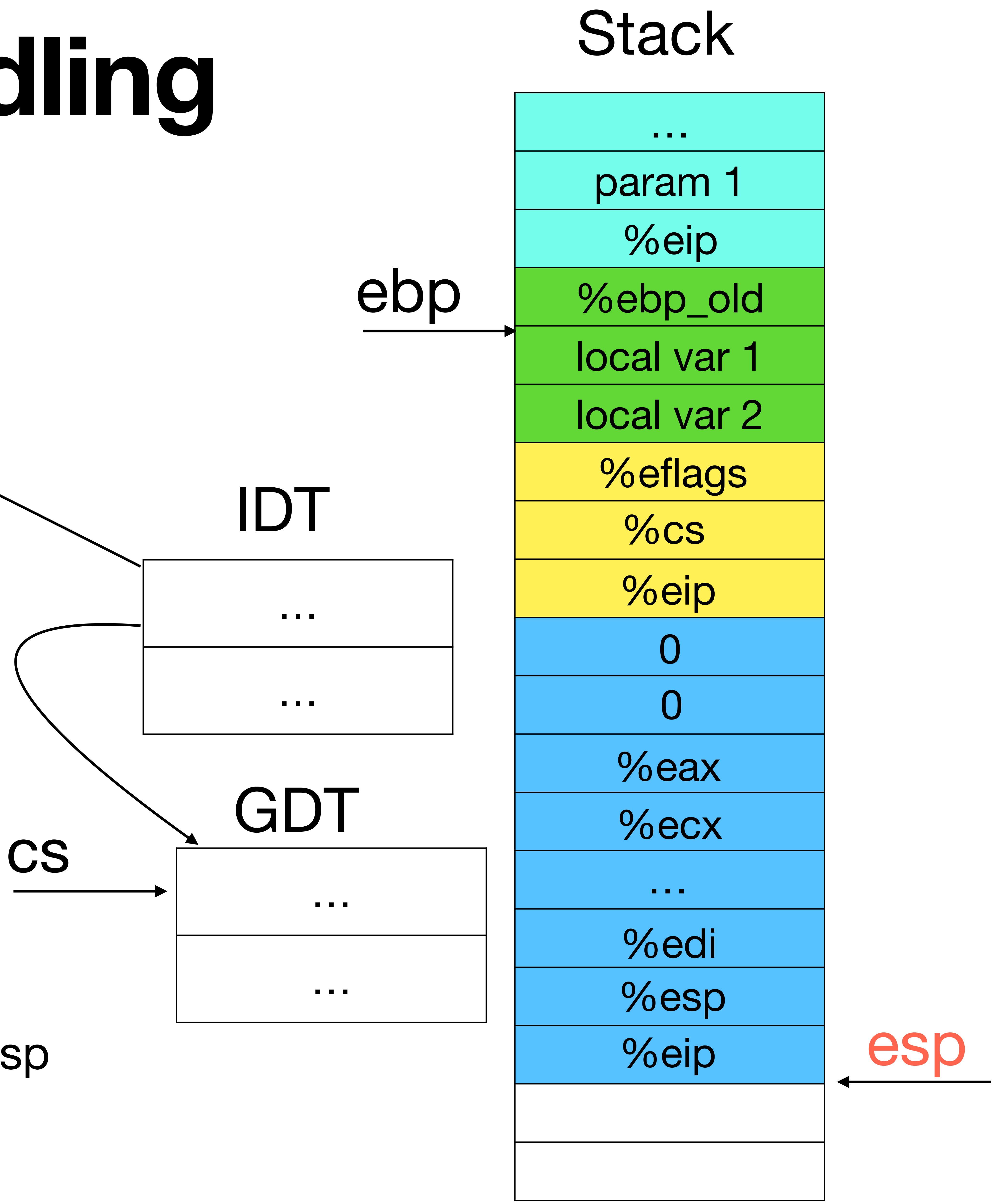
trap.c

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

trapasm.S

eip

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



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

Visualizing interrupt handling

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

trap.c

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

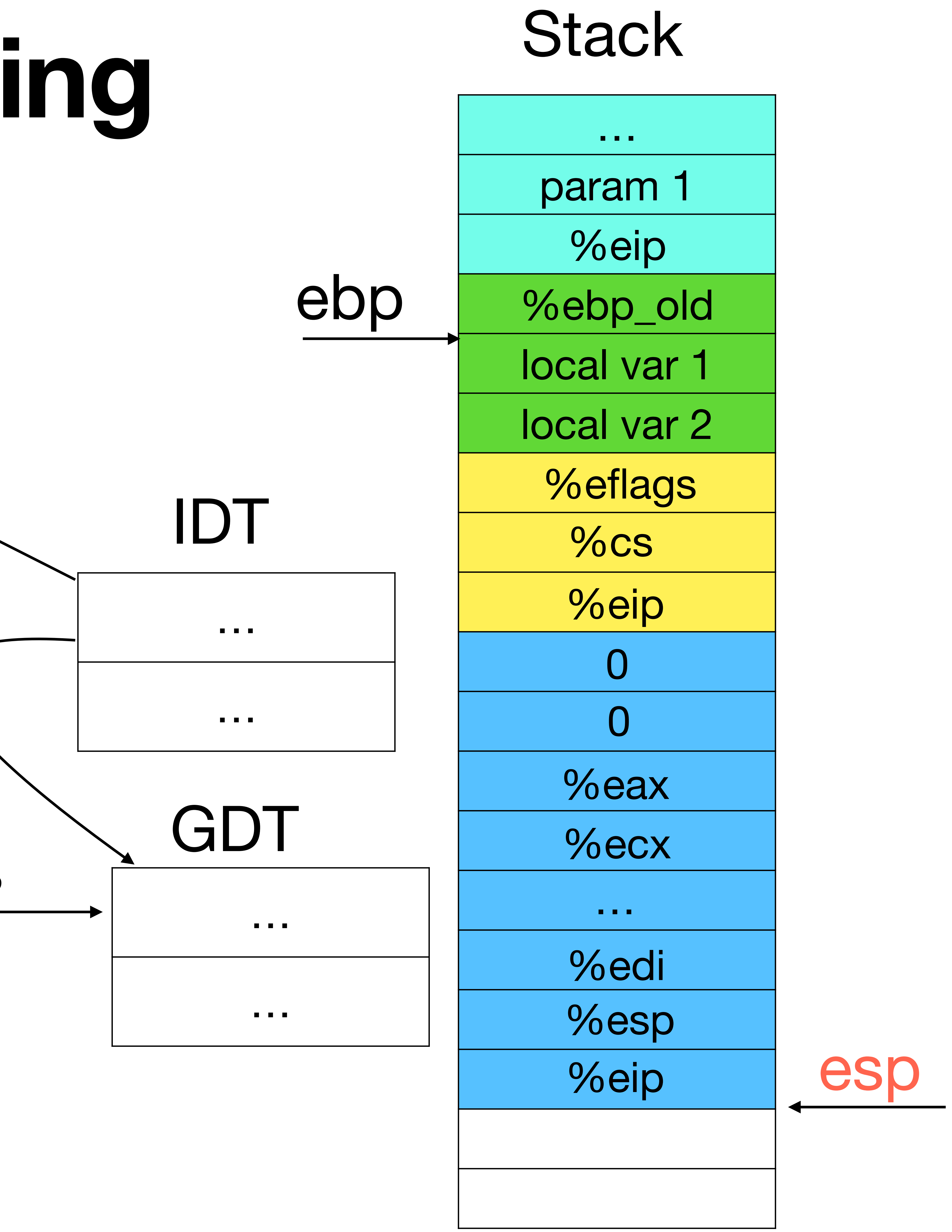
eip

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
```



Visualizing interrupt handling

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

trap.c

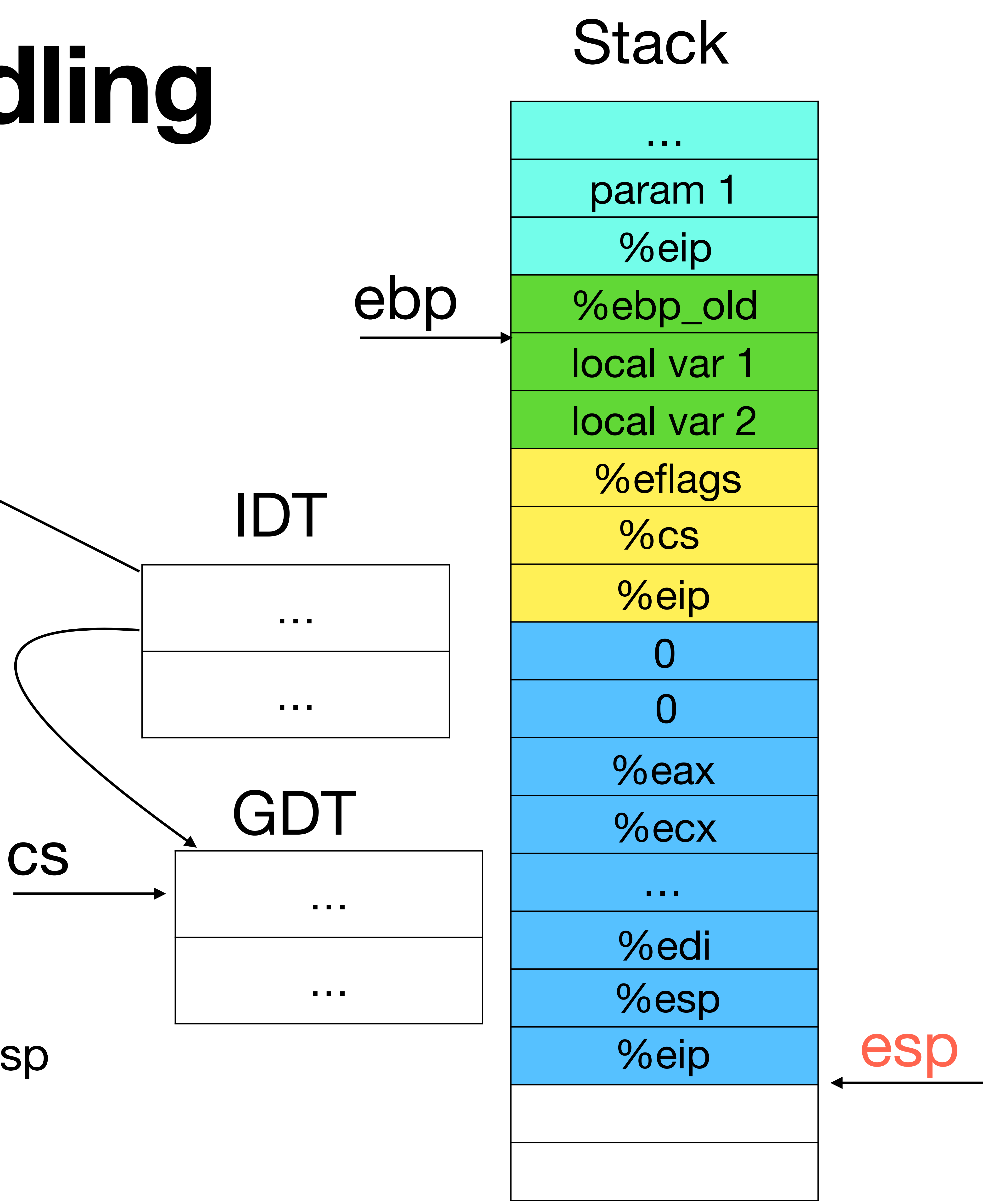
```
void
trap(struct trapframe *tf)
{
eip
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Visualizing interrupt handling

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

trap.c

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

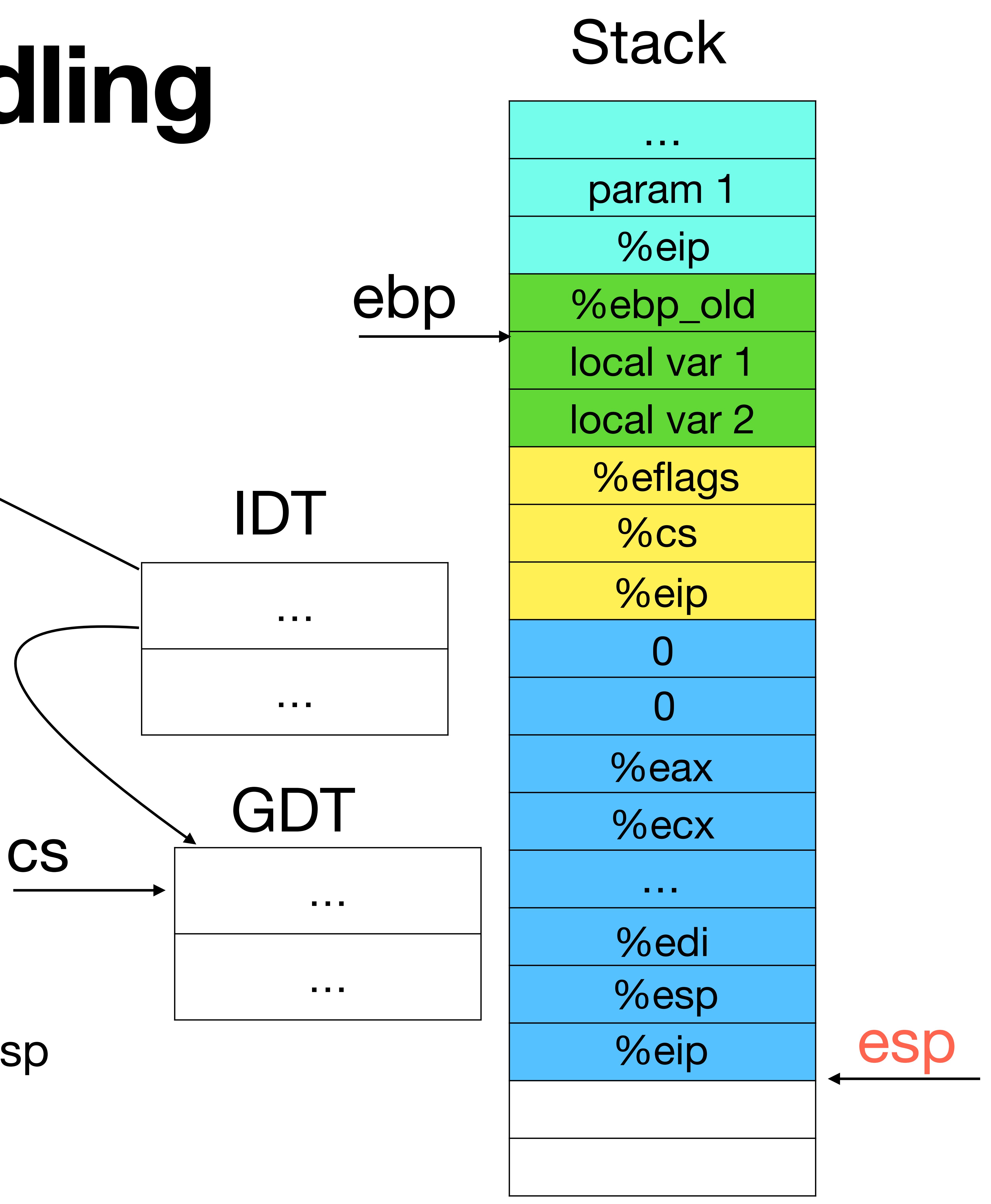
eip

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
```



Visualizing interrupt handling

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

vectors.S

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

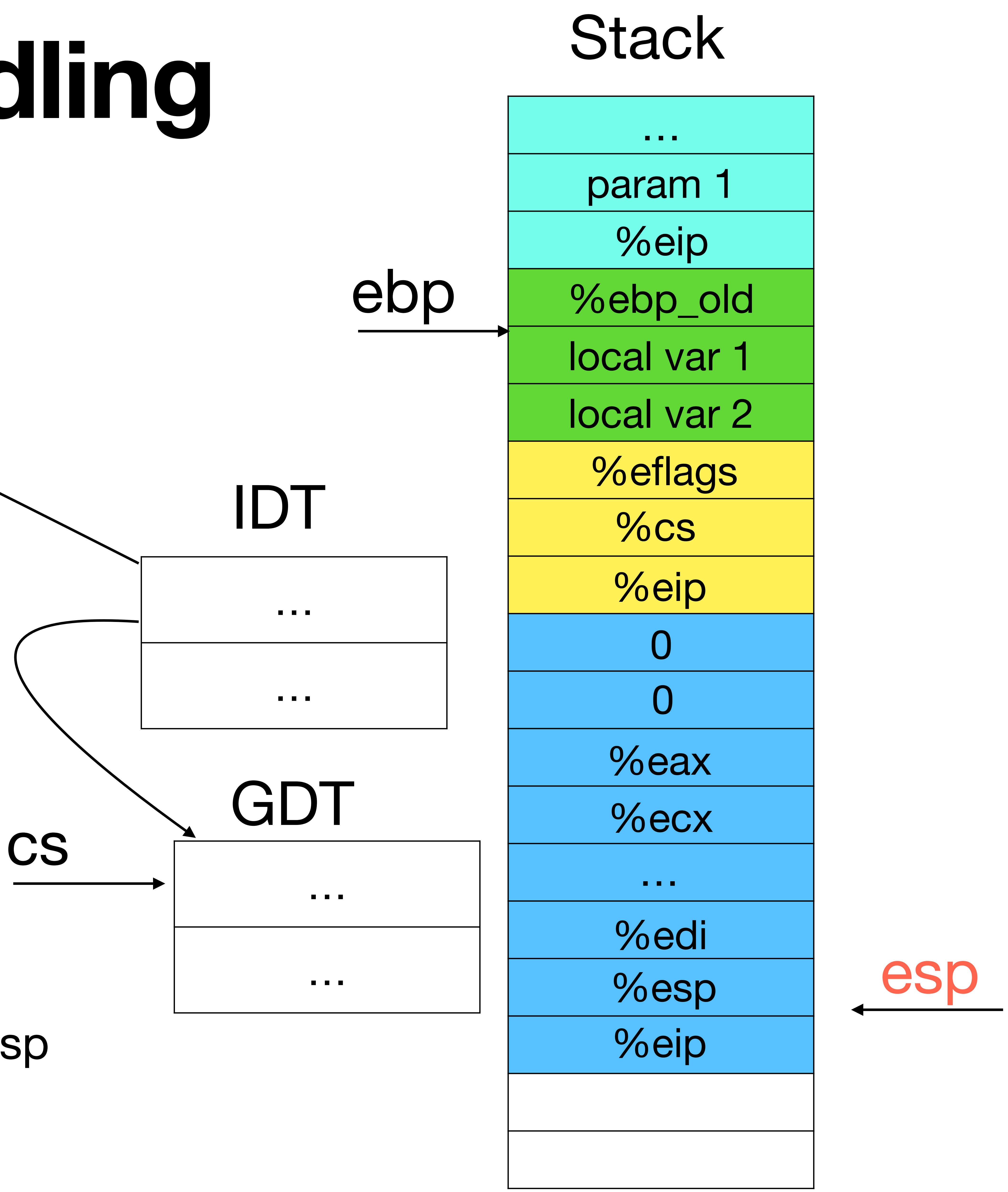
trap.c

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

eip

trapasm.S

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



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

ebp

esp

IDT

...
...

GDT

...
...

cs

Visualizing interrupt handling

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

trap.c

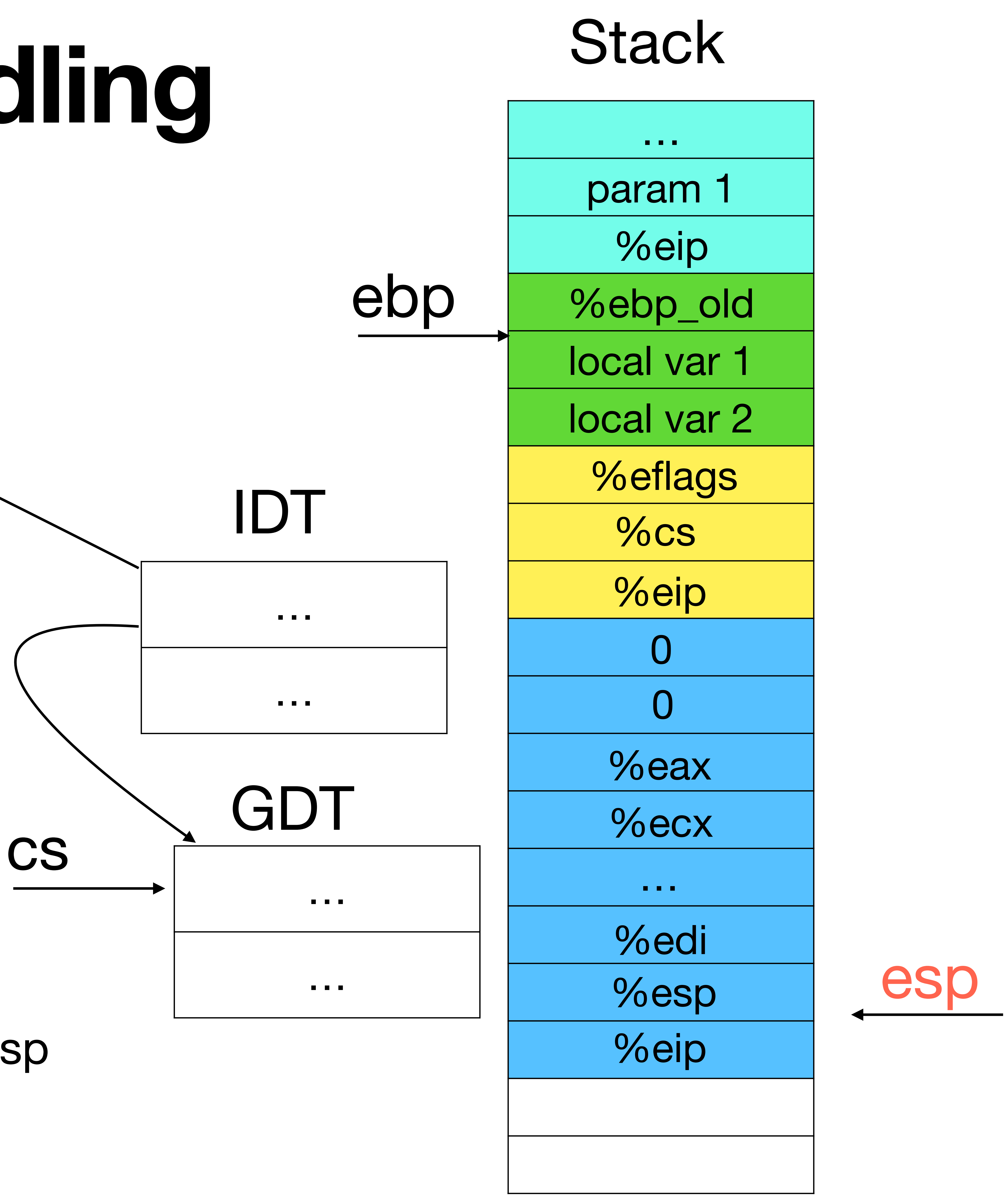
```
void
trap(struct trapframe *tf)
{
 switch(tf->trapno){
 case T_IRQ0 + IRQ_TIMER:
  ticks++;
  cprintf("Tick! %d\n\0", ticks);
  lapiceoi();
..
return
```

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
```



Visualizing interrupt handling

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

trap.c

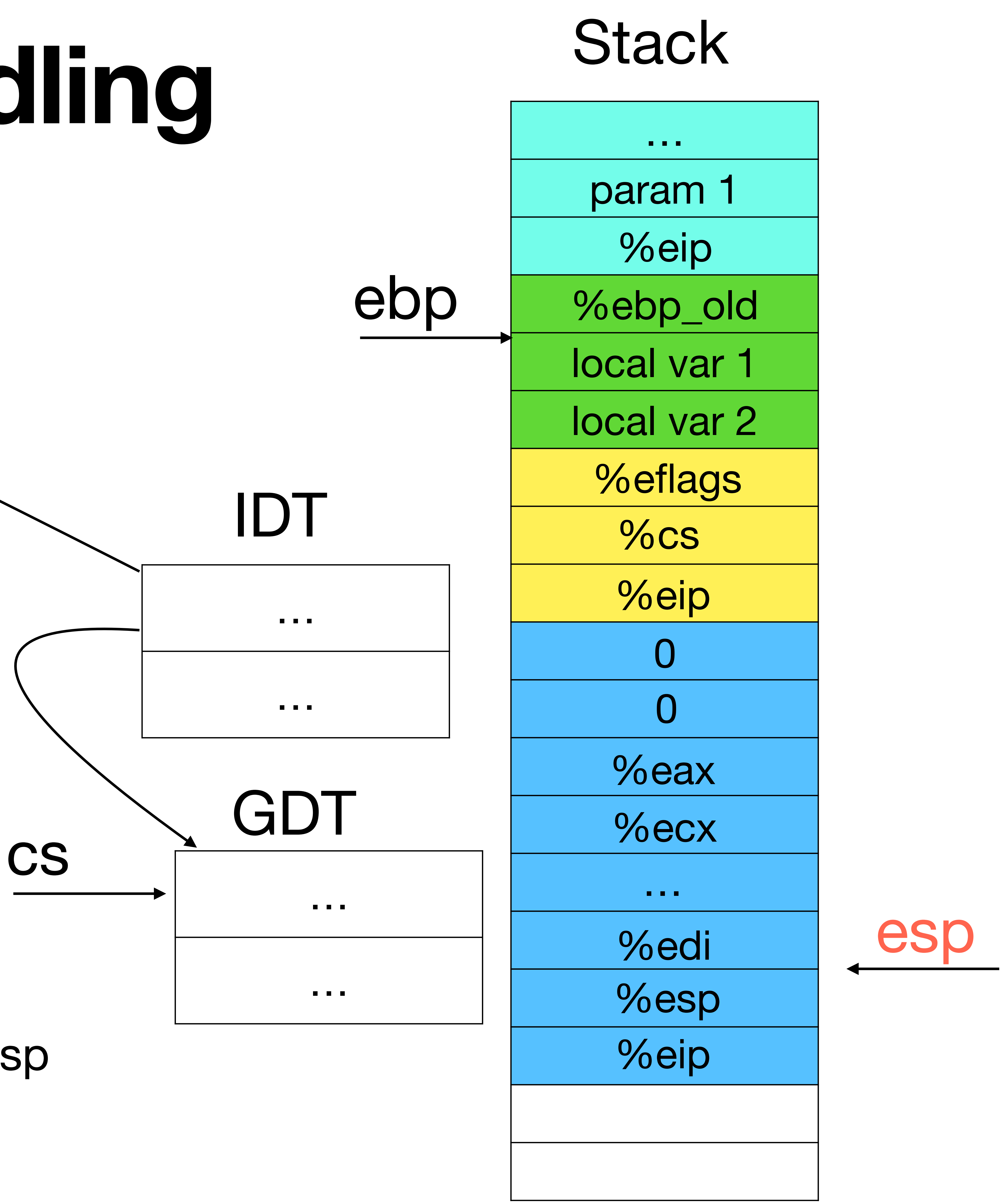
```
void
trap(struct trapframe *tf)
{
 switch(tf->trapno){
 case T_IRQ0 + IRQ_TIMER:
   ticks++;
   cprintf("Tick! %d\n\0", ticks);
   lapiceoi();
..
return
```

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
```



Visualizing interrupt handling

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

trap.c

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

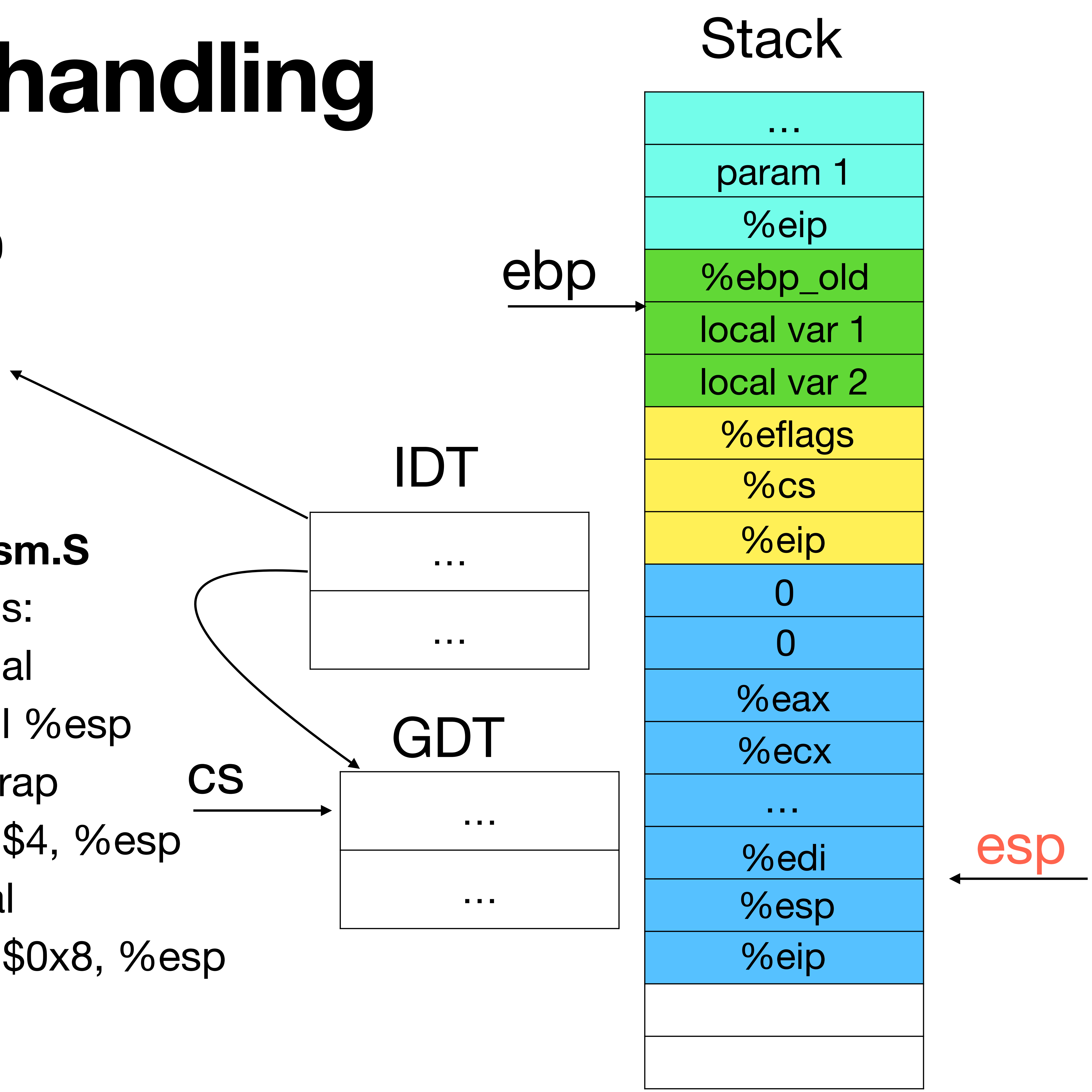
vectors.S

```
.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
```

eip



Visualizing interrupt handling

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

vectors.S

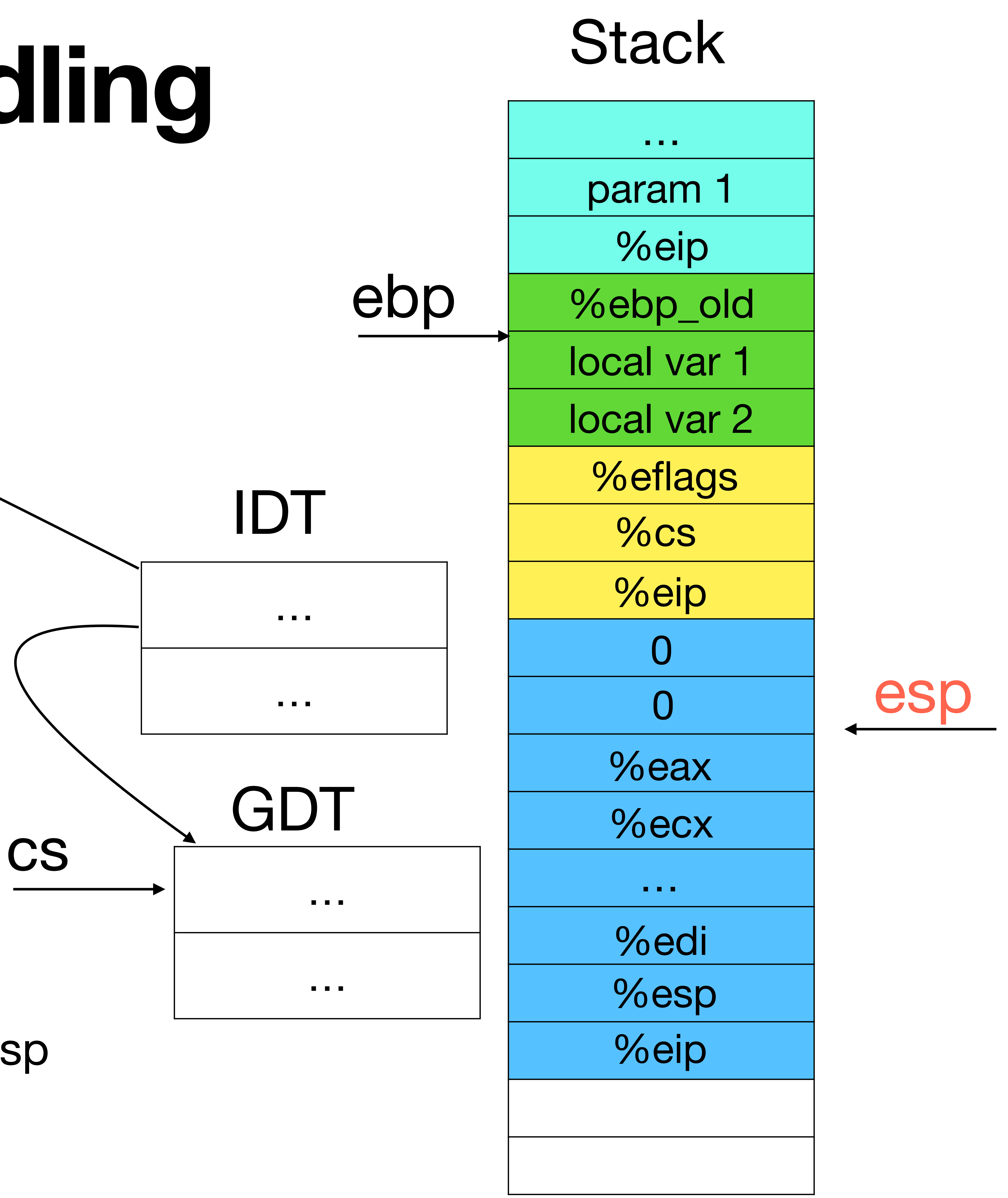
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

trap.c

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

trapasm.S

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



Stack

…
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
…
%edi
%esp
%eip

Visualizing interrupt handling

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

trap.c

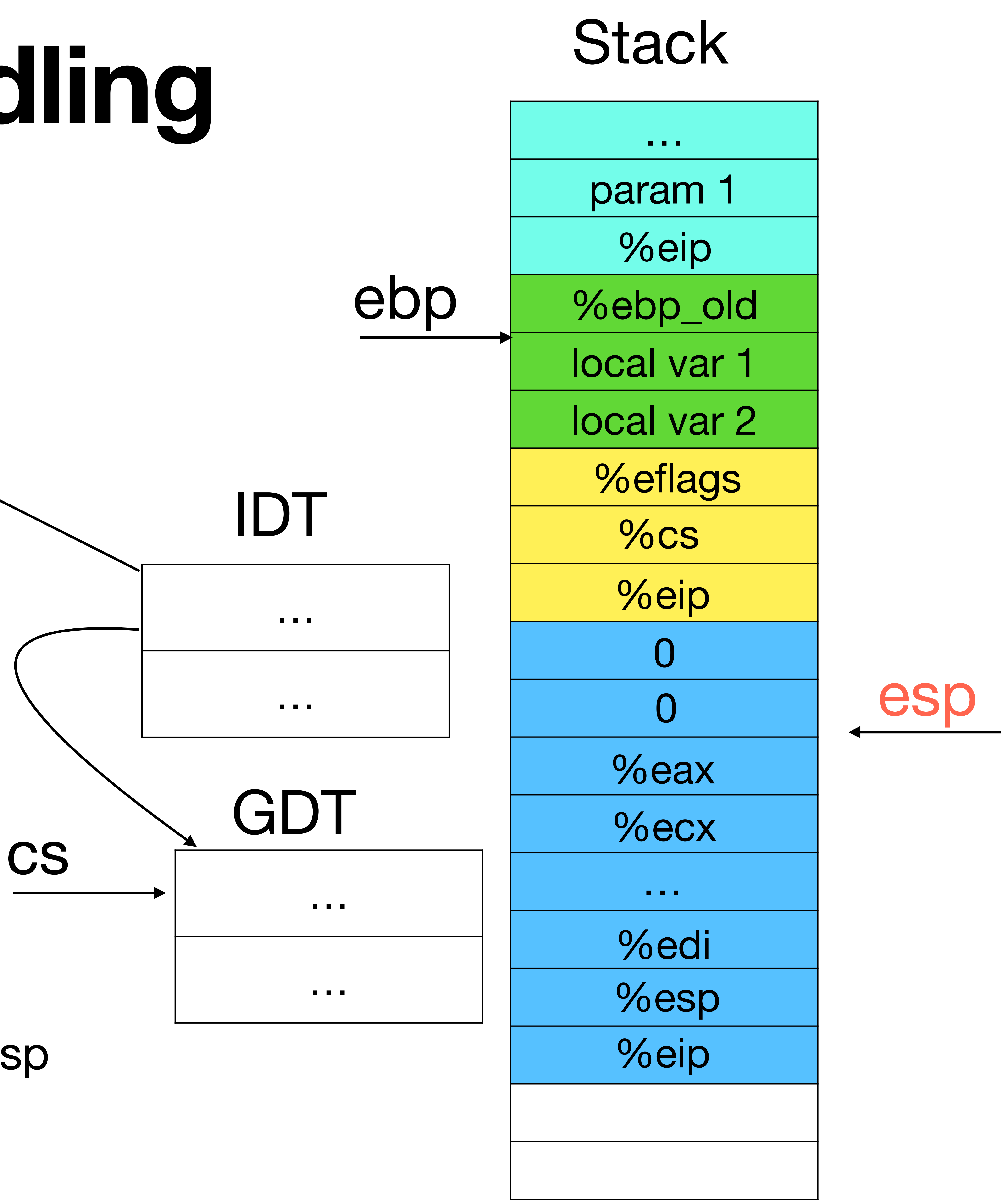
```
void
trap(struct trapframe *tf)
{
 switch(tf->trapno){
 case T_IRQ0 + IRQ_TIMER:
   ticks++;
   cprintf("Tick! %d\n\0", ticks);
   lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

Visualizing interrupt handling

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

trap.c

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

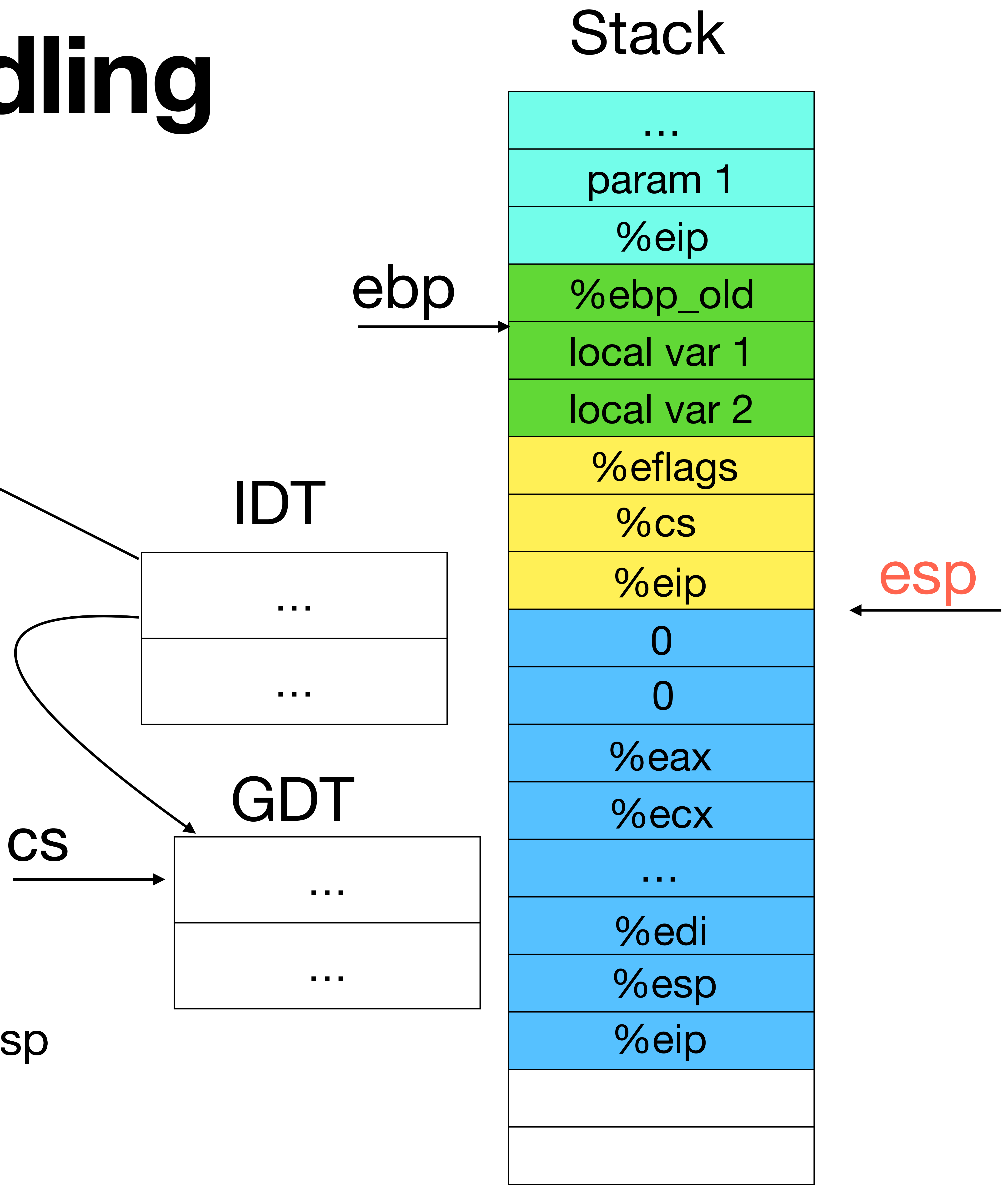
vectors.S

```
.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
```

eip



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

Visualizing interrupt handling

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

trap.c

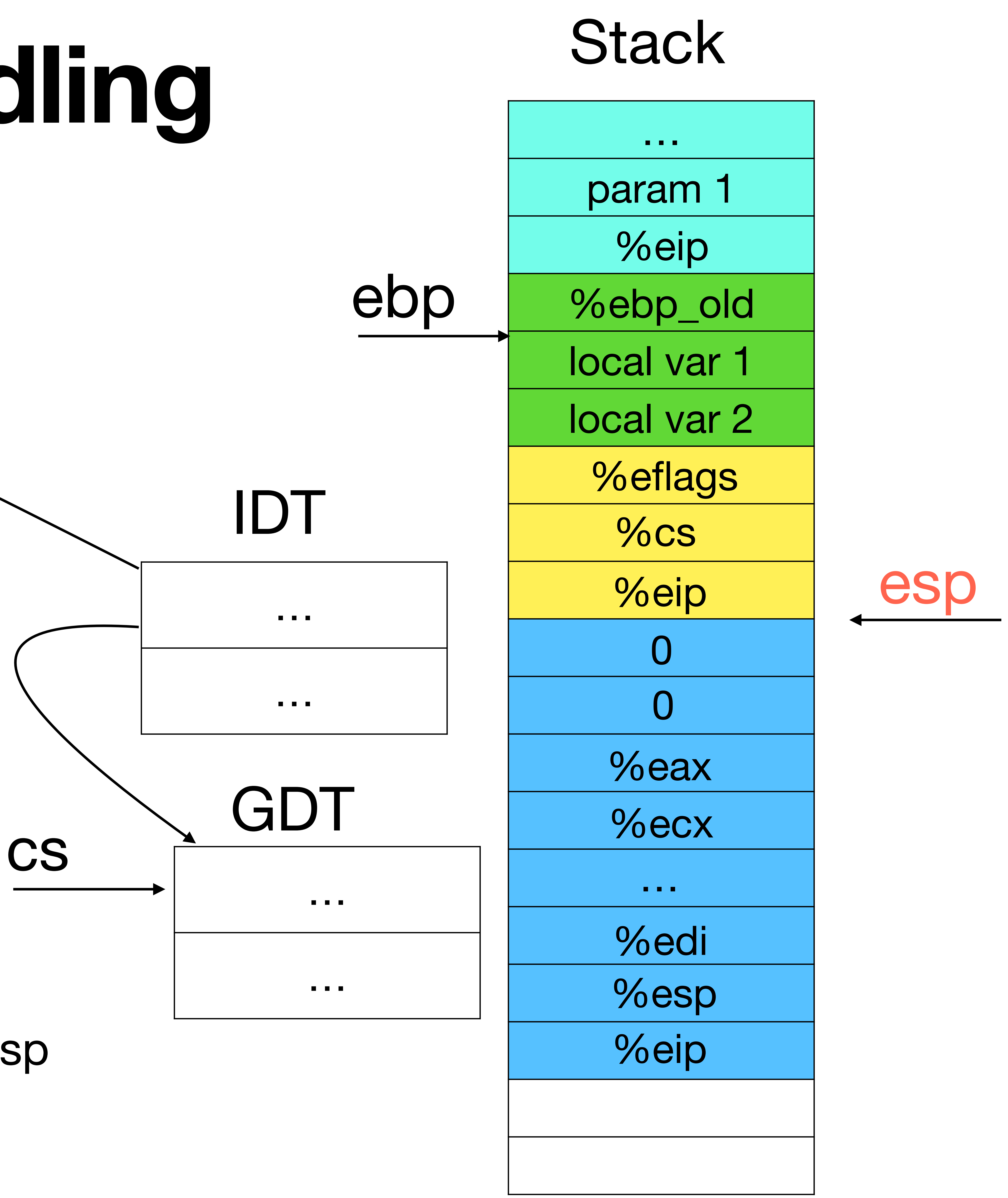
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

Visualizing interrupt handling

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

trap.c

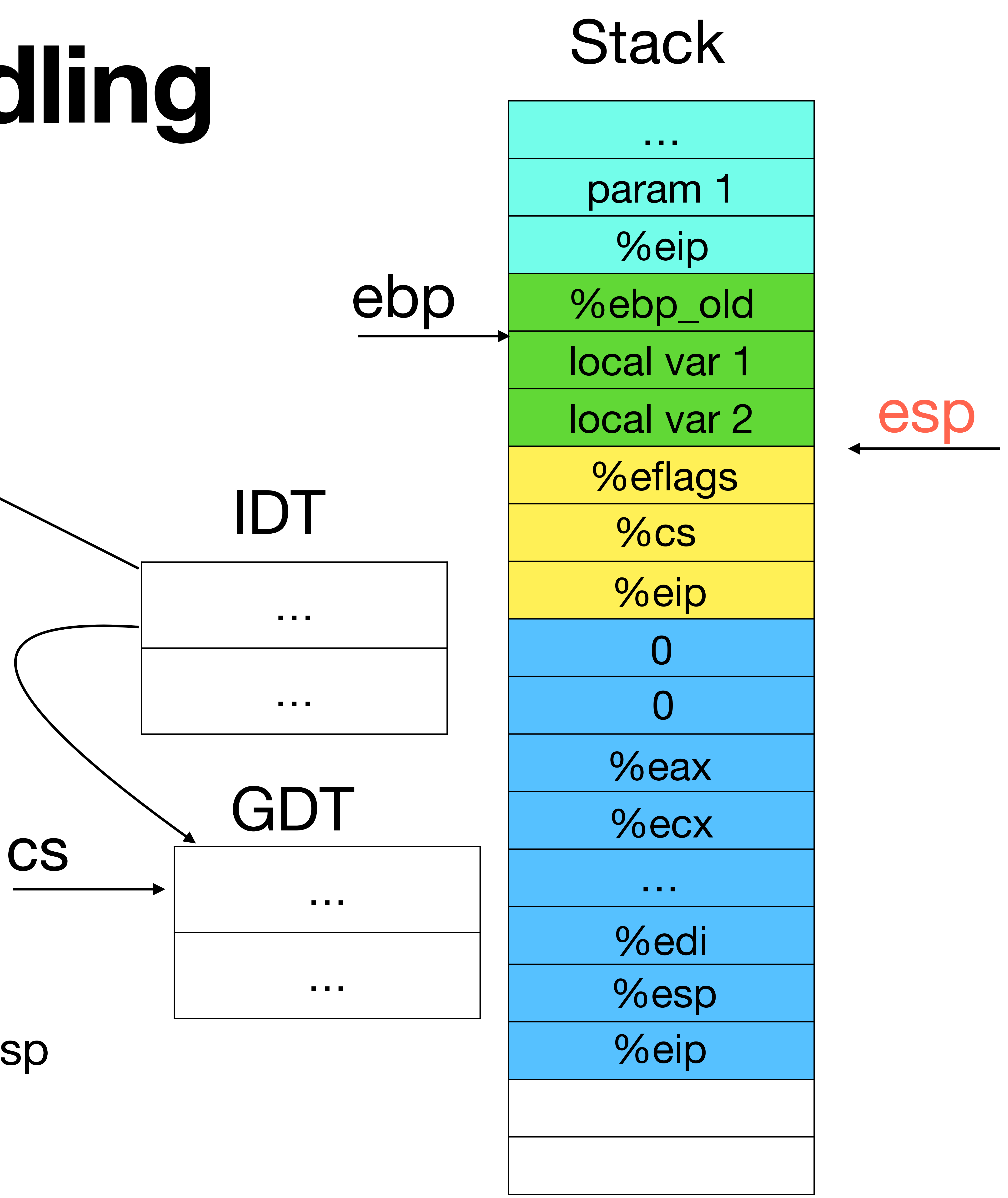
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

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
```



Stack

...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

Visualizing interrupt handling

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

trap.c

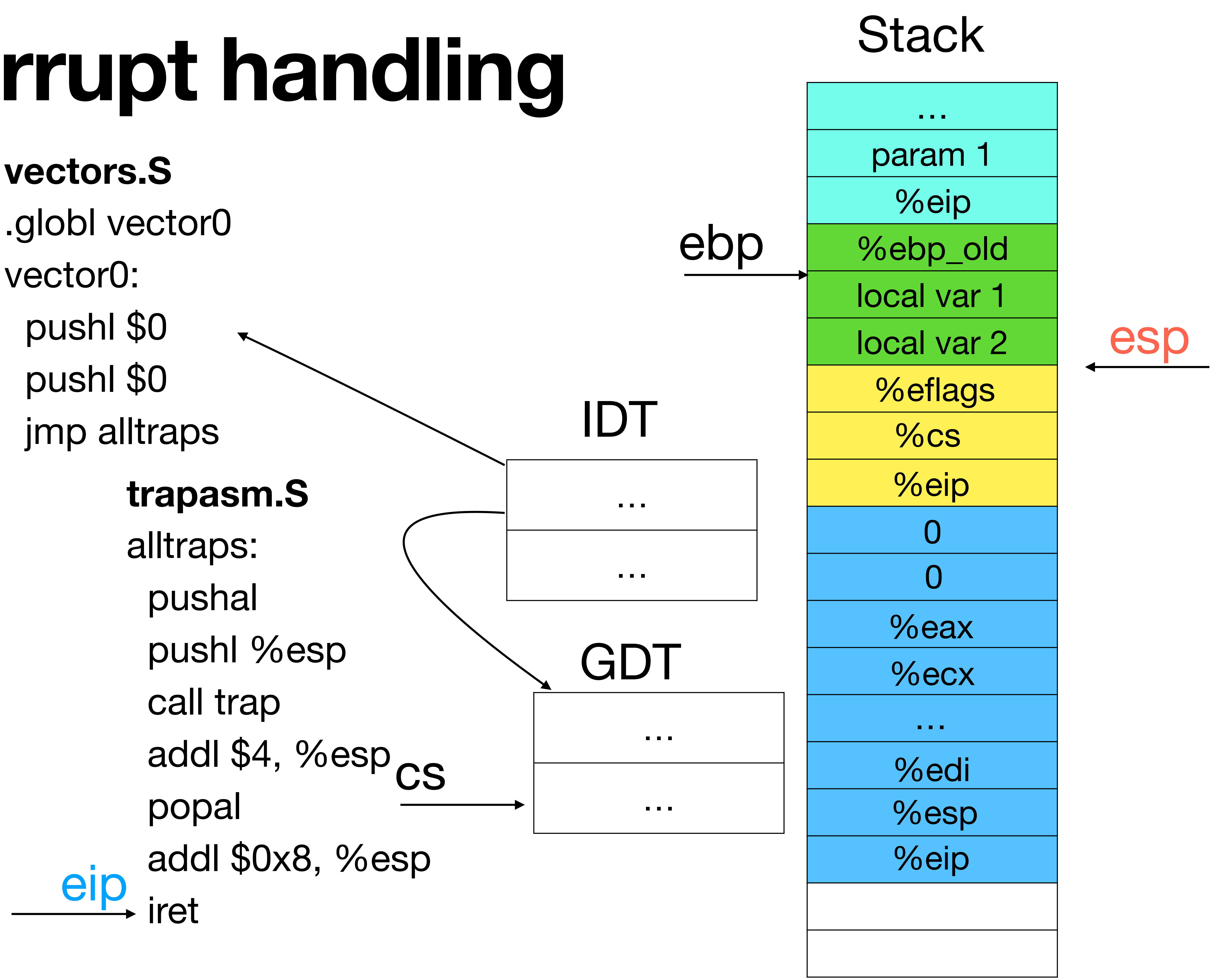
```
void
trap(struct trapframe *tf)
{
 switch(tf->trapno){
 case T_IRQ0 + IRQ_TIMER:
   ticks++;
   cprintf("Tick! %d\n\0", ticks);
   lapiceoi();
..
return
```

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
```



Stack
...
param 1
%eip
%ebp_old
local var 1
local var 2
%eflags
%cs
%eip
0
0
%eax
%ecx
...
%edi
%esp
%eip

Visualizing interrupt handling

eip

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

trap.c

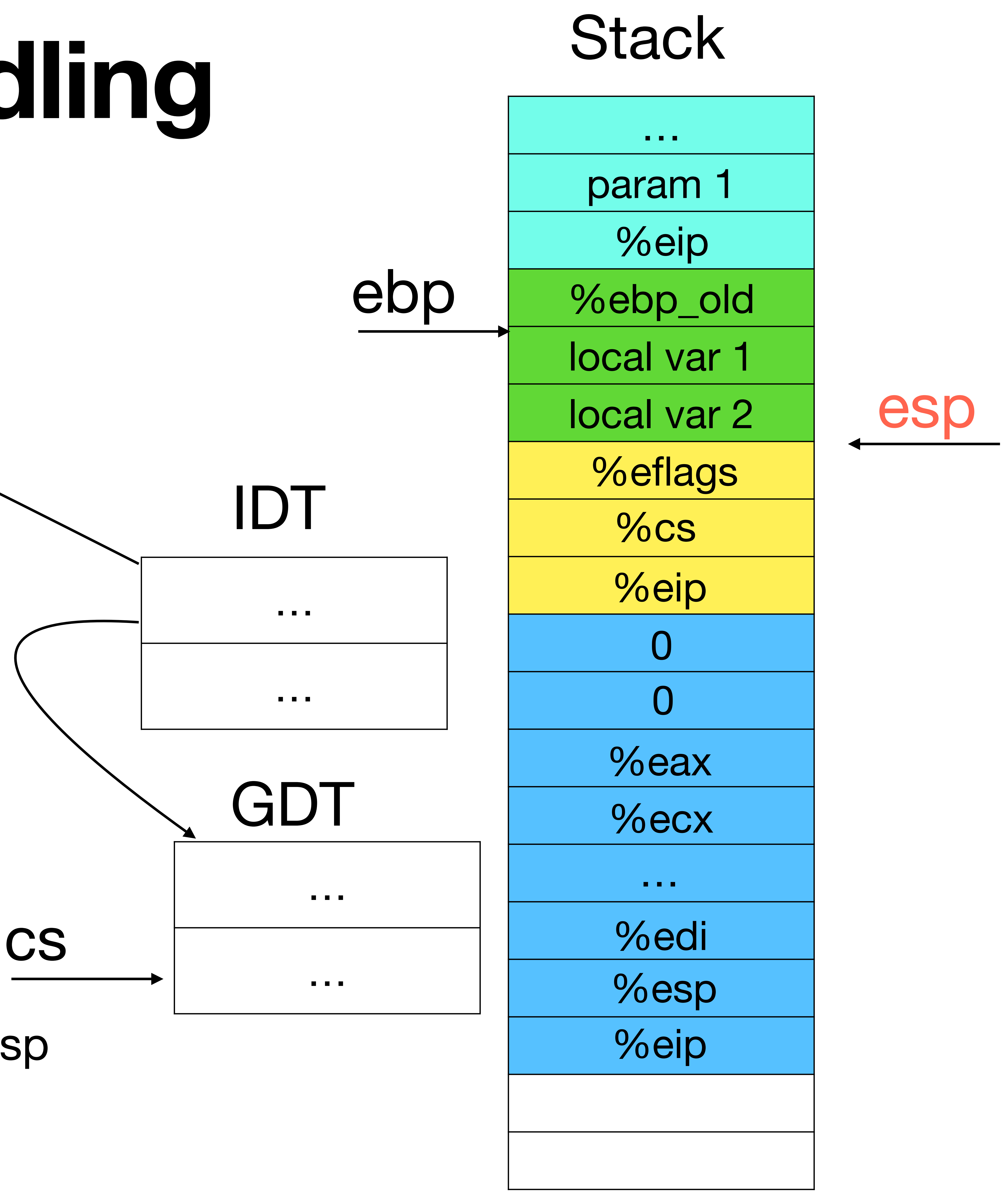
```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

vectors.S

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

trapasm.S

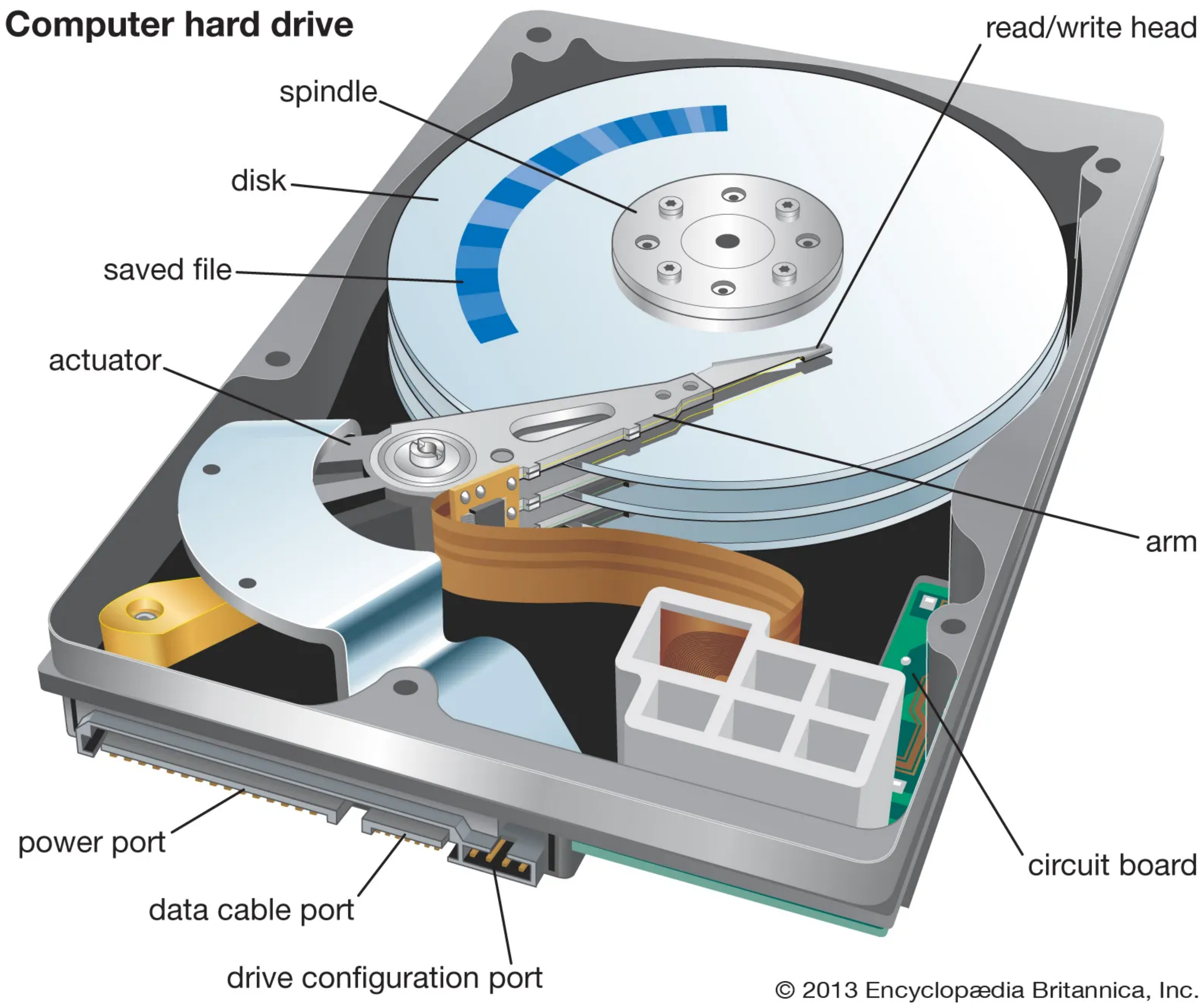
```
alltraps:
  pushal
  pushl %esp
  call trap
  addl $4, %esp
  popal
  addl $0x8, %esp
  iret
```



Hard disk drive

Ch. 37 OSTEP book

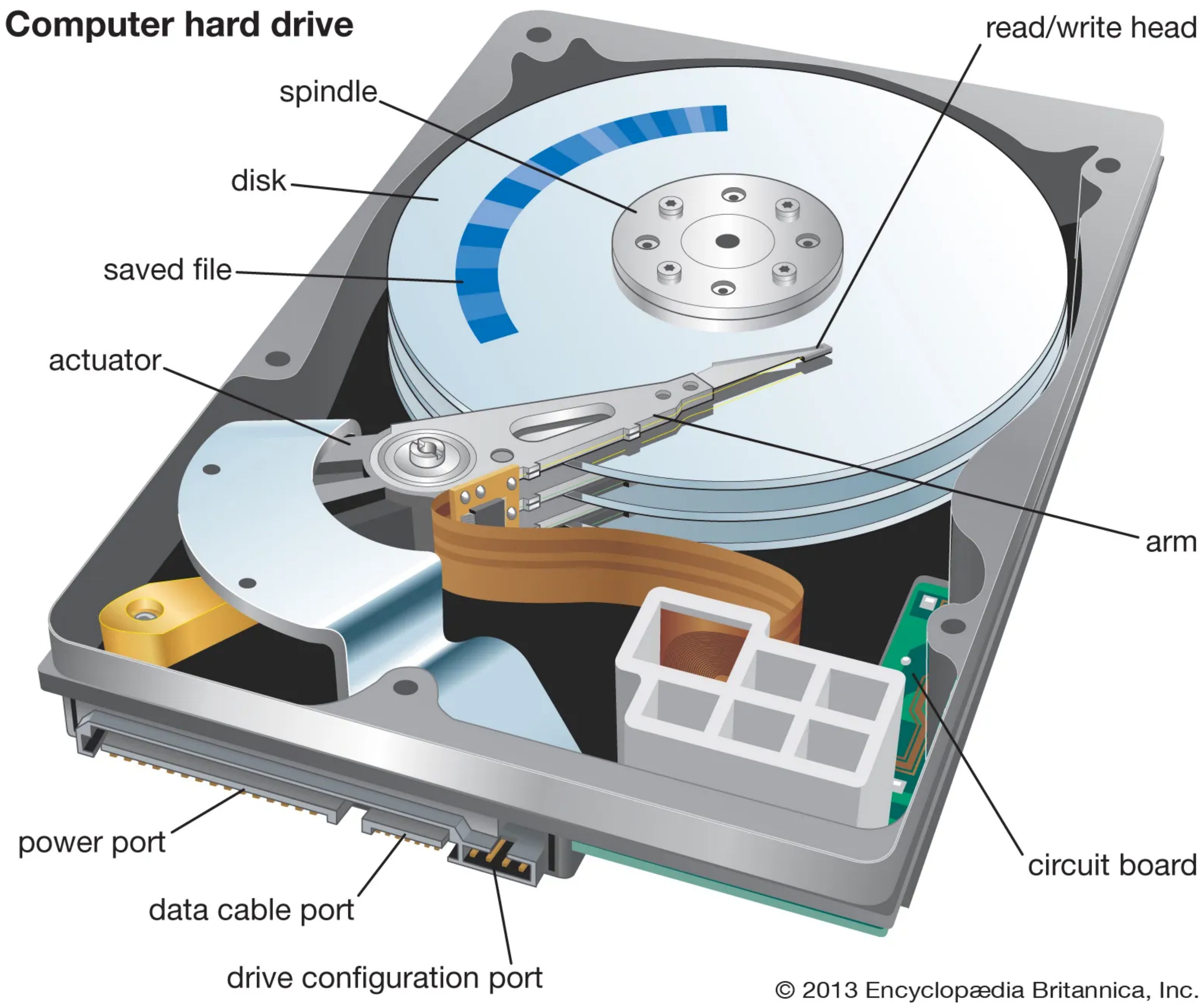
Disk geometry



© 2013 Encyclopædia Britannica, Inc.

Disk geometry

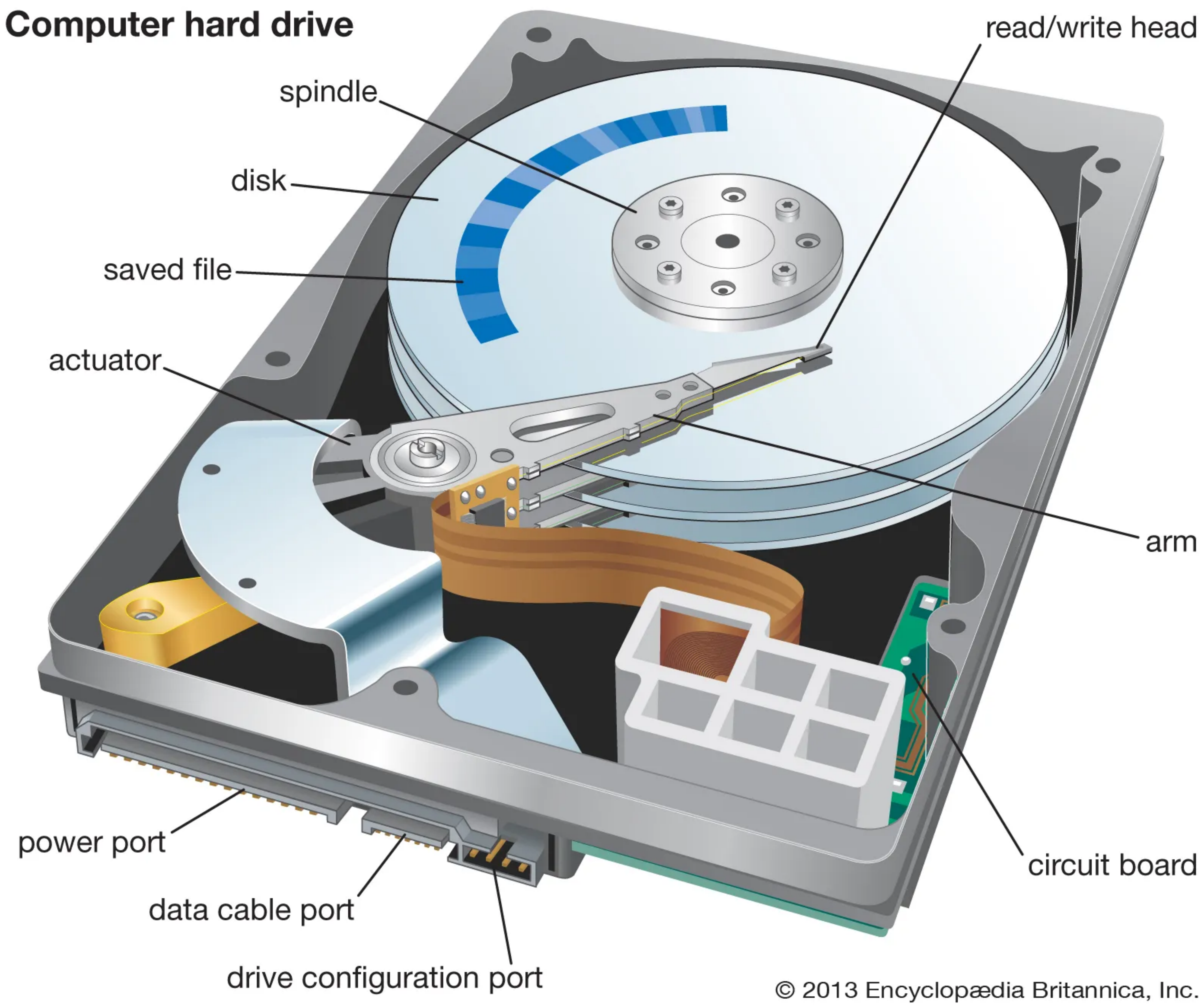
- Many platters spinning on a spindle (~10,000 RPM)



© 2013 Encyclopædia Britannica, Inc.

Disk geometry

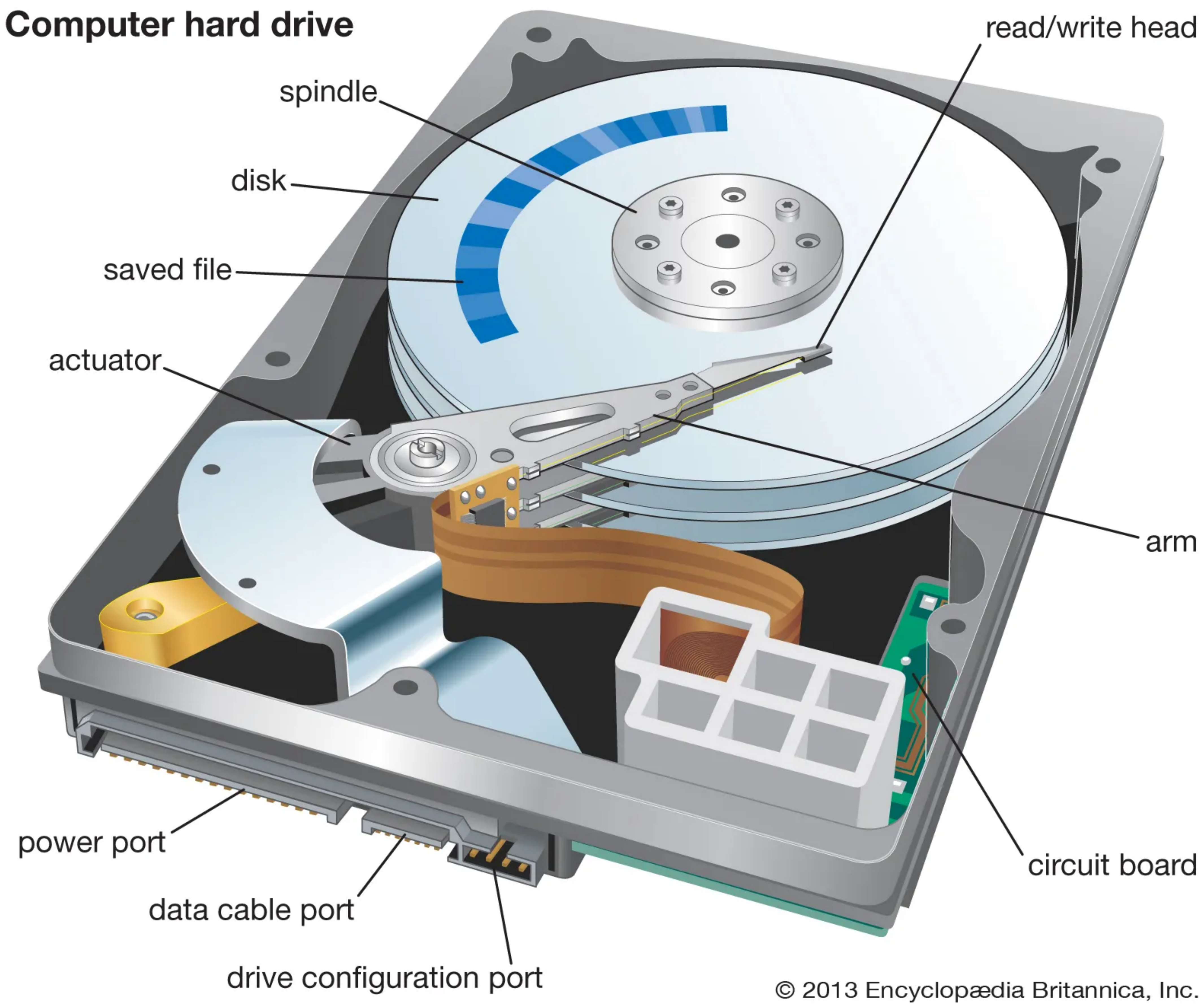
- Many platters spinning on a spindle (~10,000 RPM)
- Each platter has two disk heads, one for each surface



© 2013 Encyclopædia Britannica, Inc.

Disk geometry

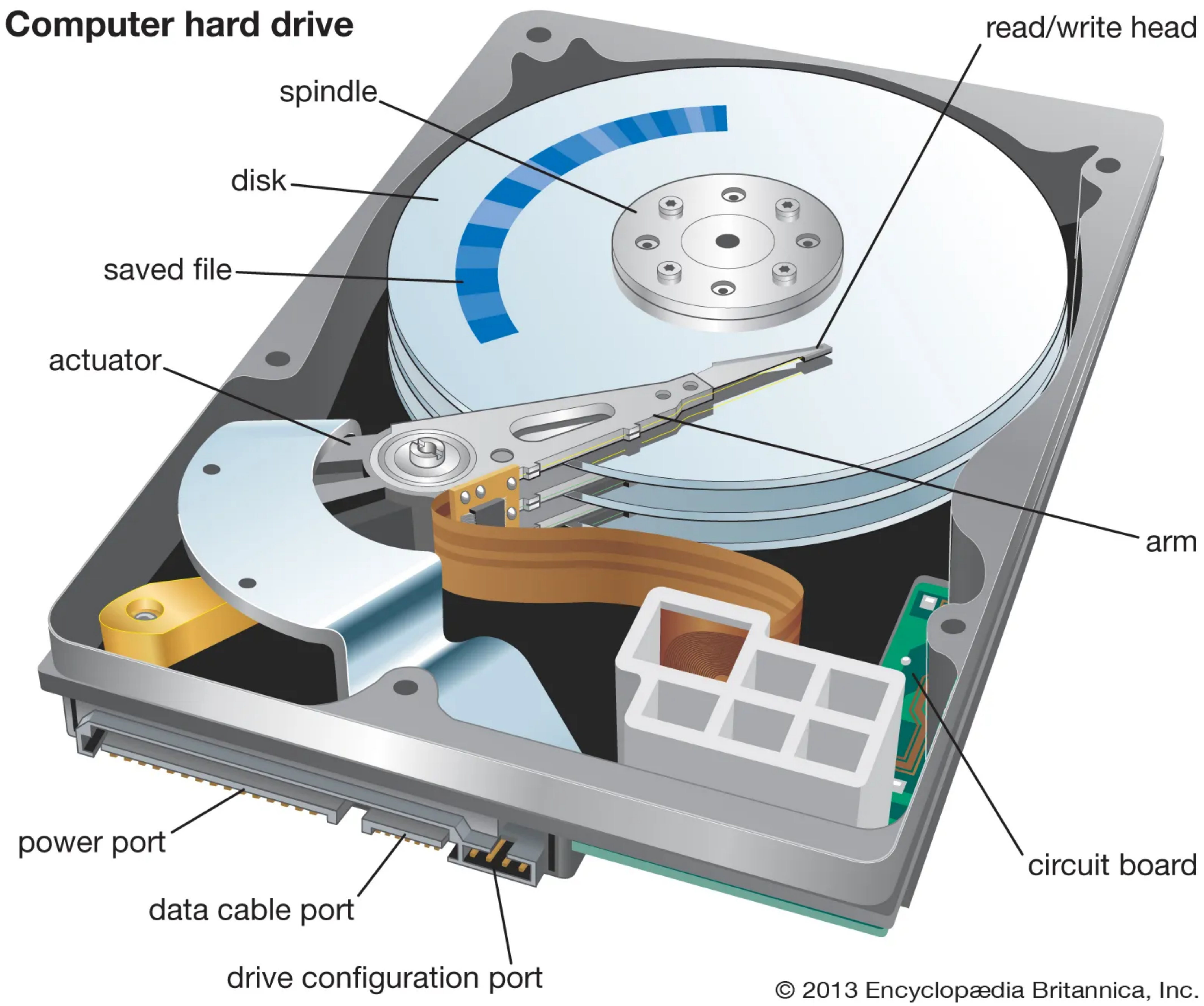
- Many platters spinning on a spindle (~10,000 RPM)
- Each platter has two disk heads, one for each surface
- Disk heads are controlled by actuator



© 2013 Encyclopædia Britannica, Inc.

Disk geometry

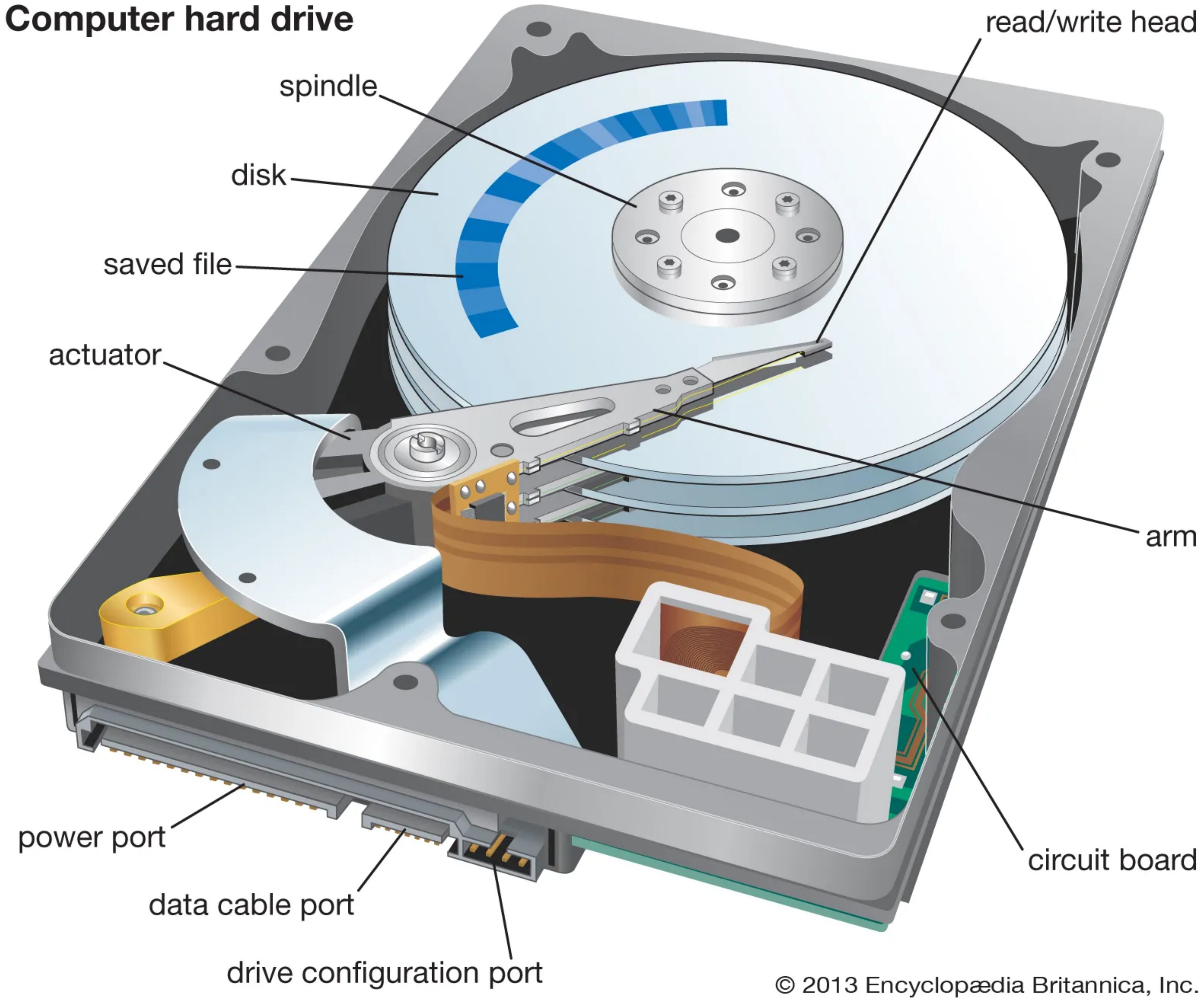
- Many platters spinning on a spindle (~10,000 RPM)
- Each platter has two disk heads, one for each surface
- Disk heads are controlled by actuator
- One circle is called a track. Data is stored in sectors



© 2013 Encyclopædia Britannica, Inc.

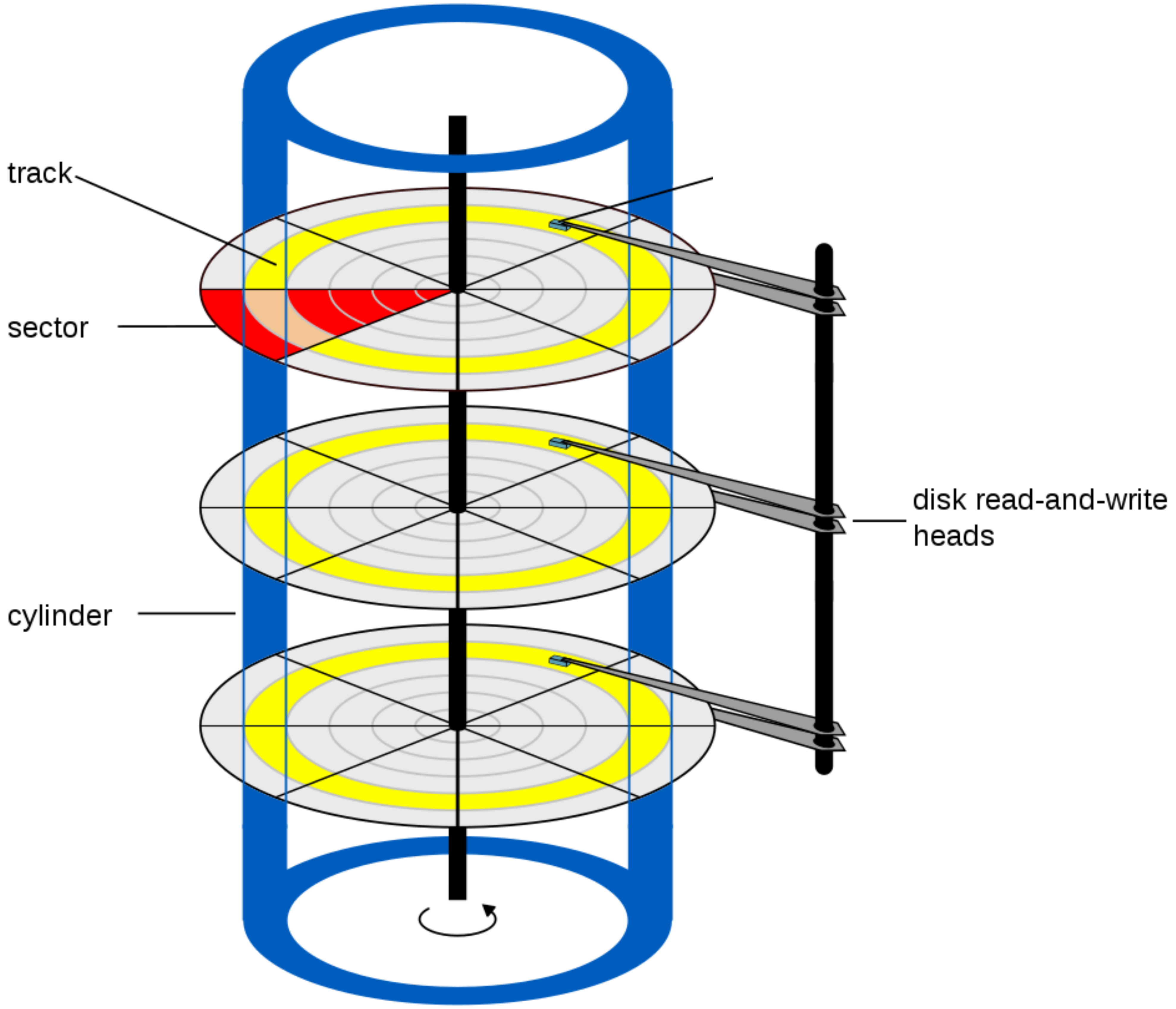
Disk geometry

- 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



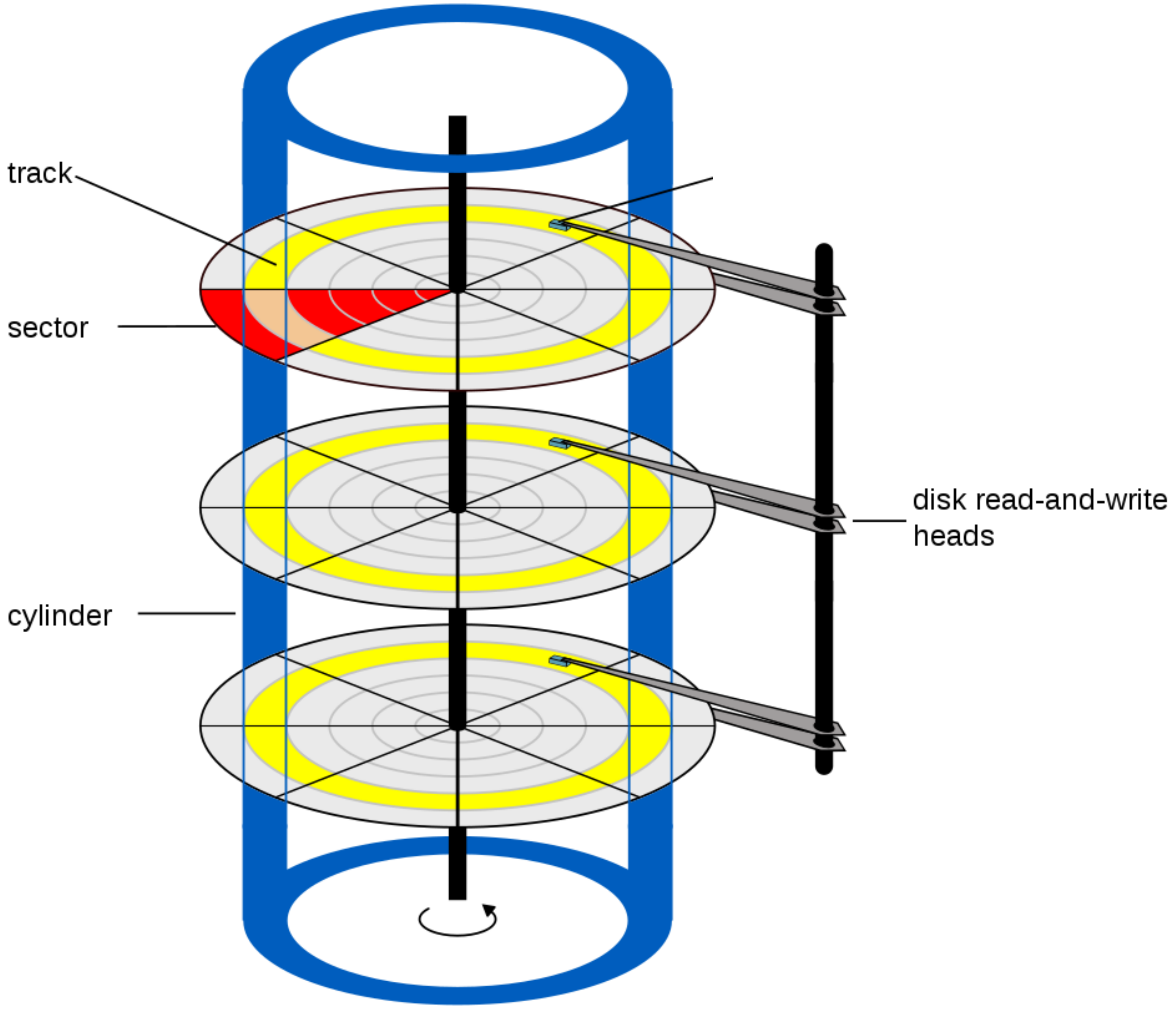
© 2013 Encyclopædia Britannica, Inc.

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



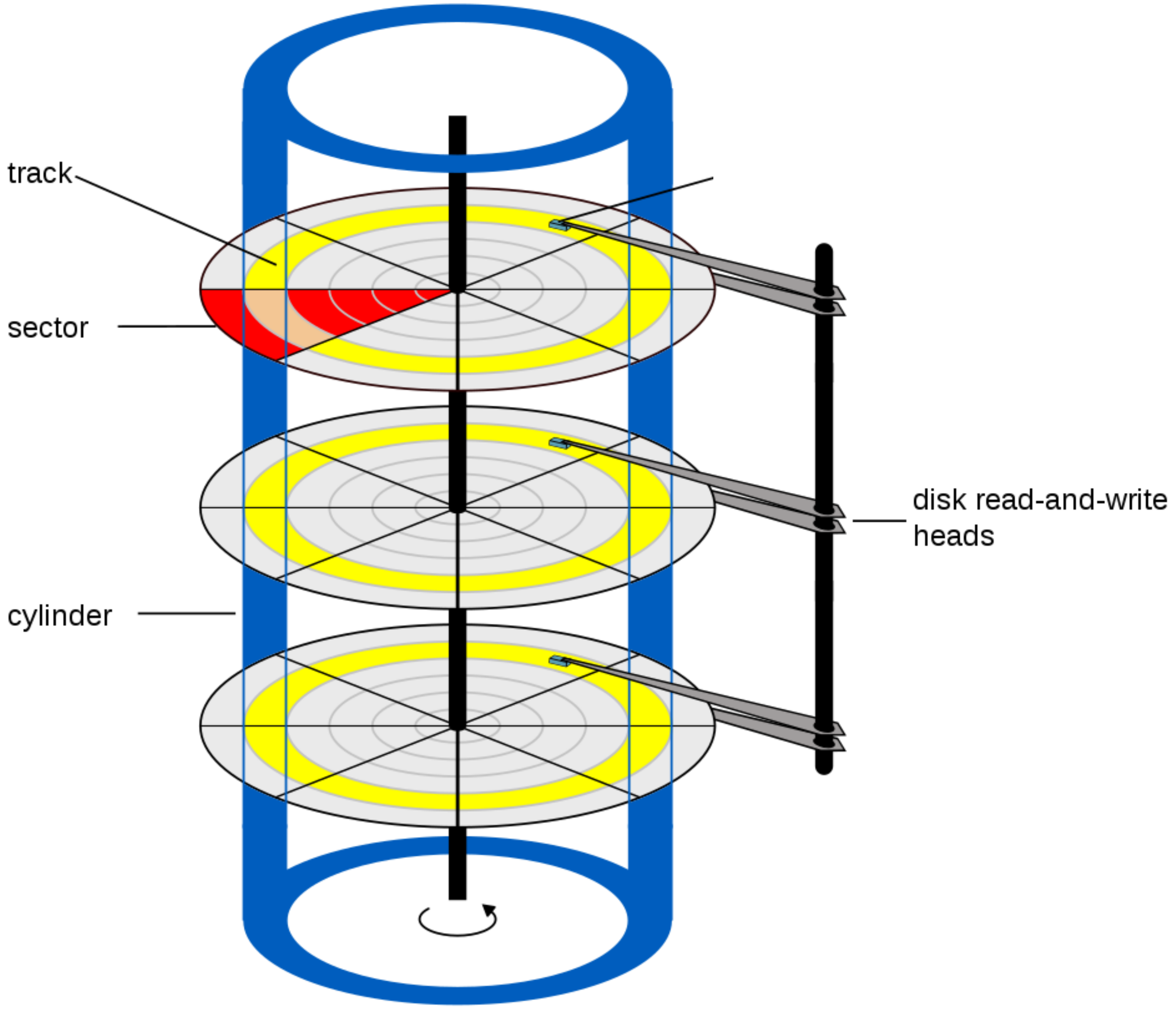
CPU-disk interface: Cylinder-head-sector (CHS) addressing

- C: cylinder number. 1024 cylinders.



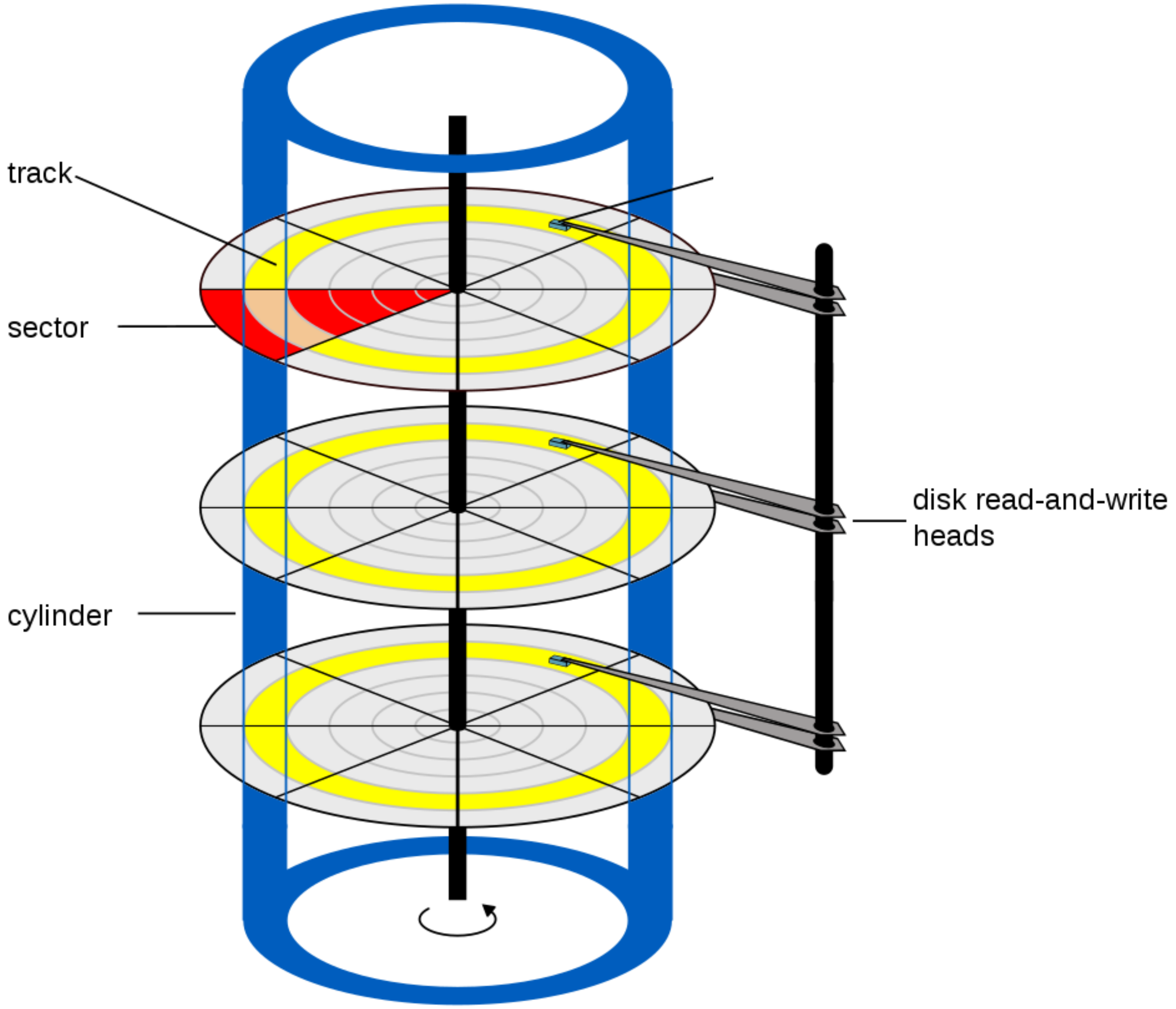
CPU-disk interface: Cylinder-head-sector (CHS) addressing

- C: cylinder number. 1024 cylinders.
- H: head number. 255 heads.



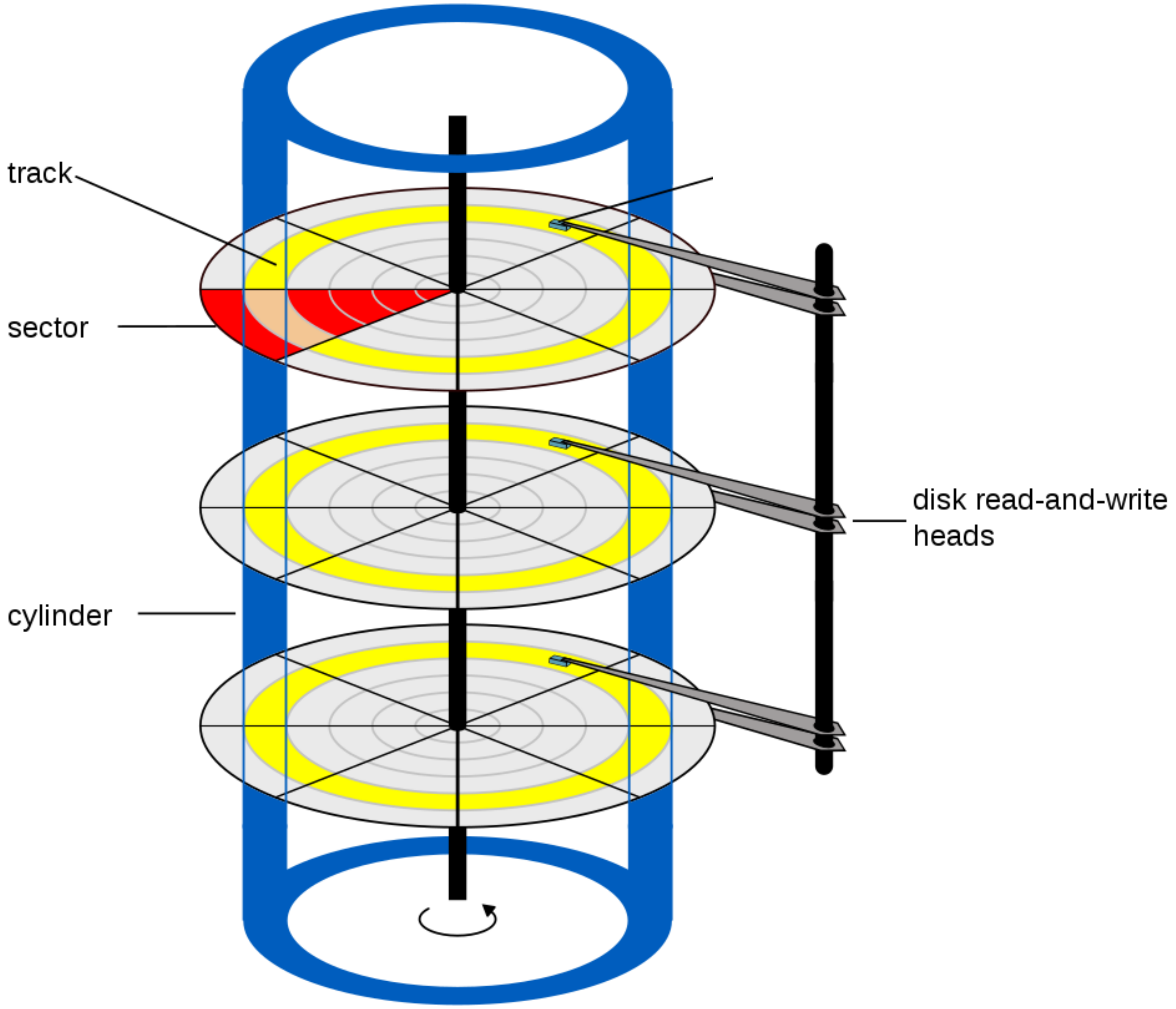
CPU-disk interface: Cylinder-head-sector (CHS) addressing

- C: cylinder number. 1024 cylinders.
- H: head number. 255 heads.
- S: sector number. 63 sectors per track.



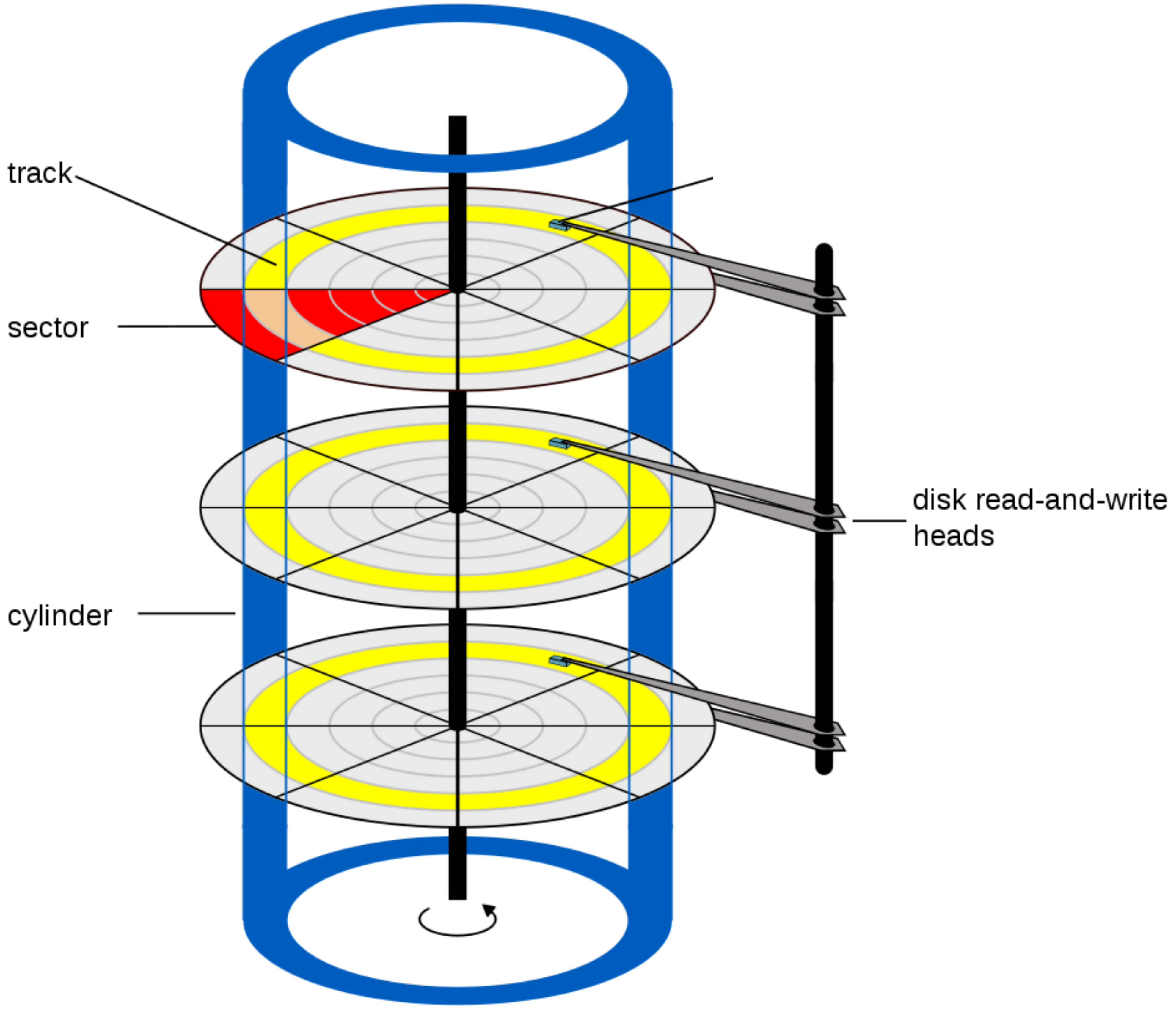
CPU-disk interface: Cylinder-head-sector (CHS) addressing

- C: cylinder number. 1024 cylinders.
- H: head number. 255 heads.
- S: sector number. 63 sectors per track.
- 512 bytes in each sector

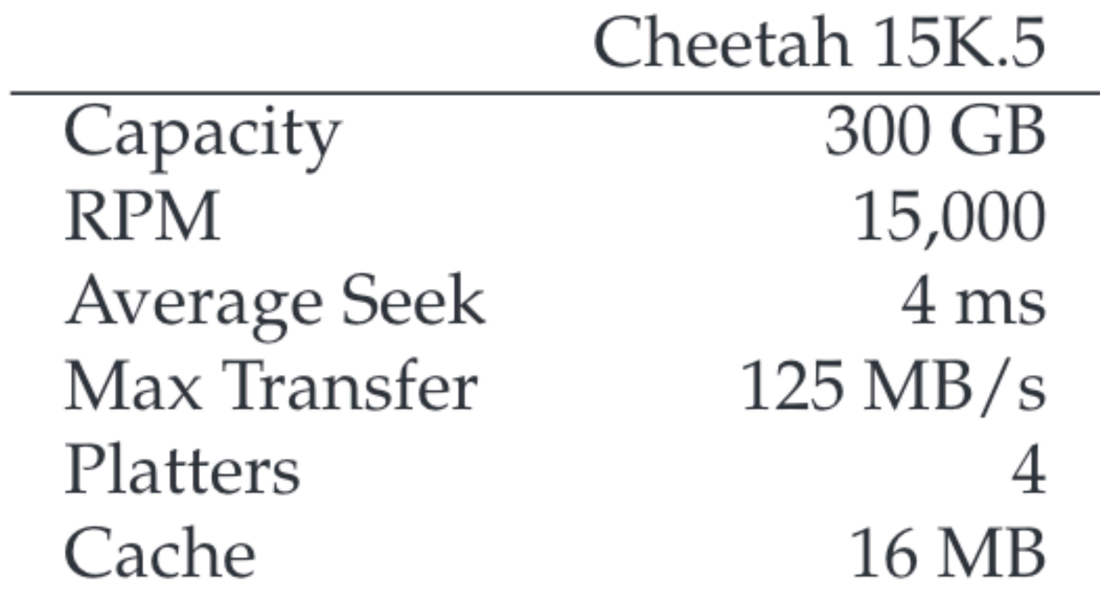


CPU-disk interface: Cylinder-head-sector (CHS) addressing

- C: cylinder number. 1024 cylinders.
- H: head number. 255 heads.
- 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

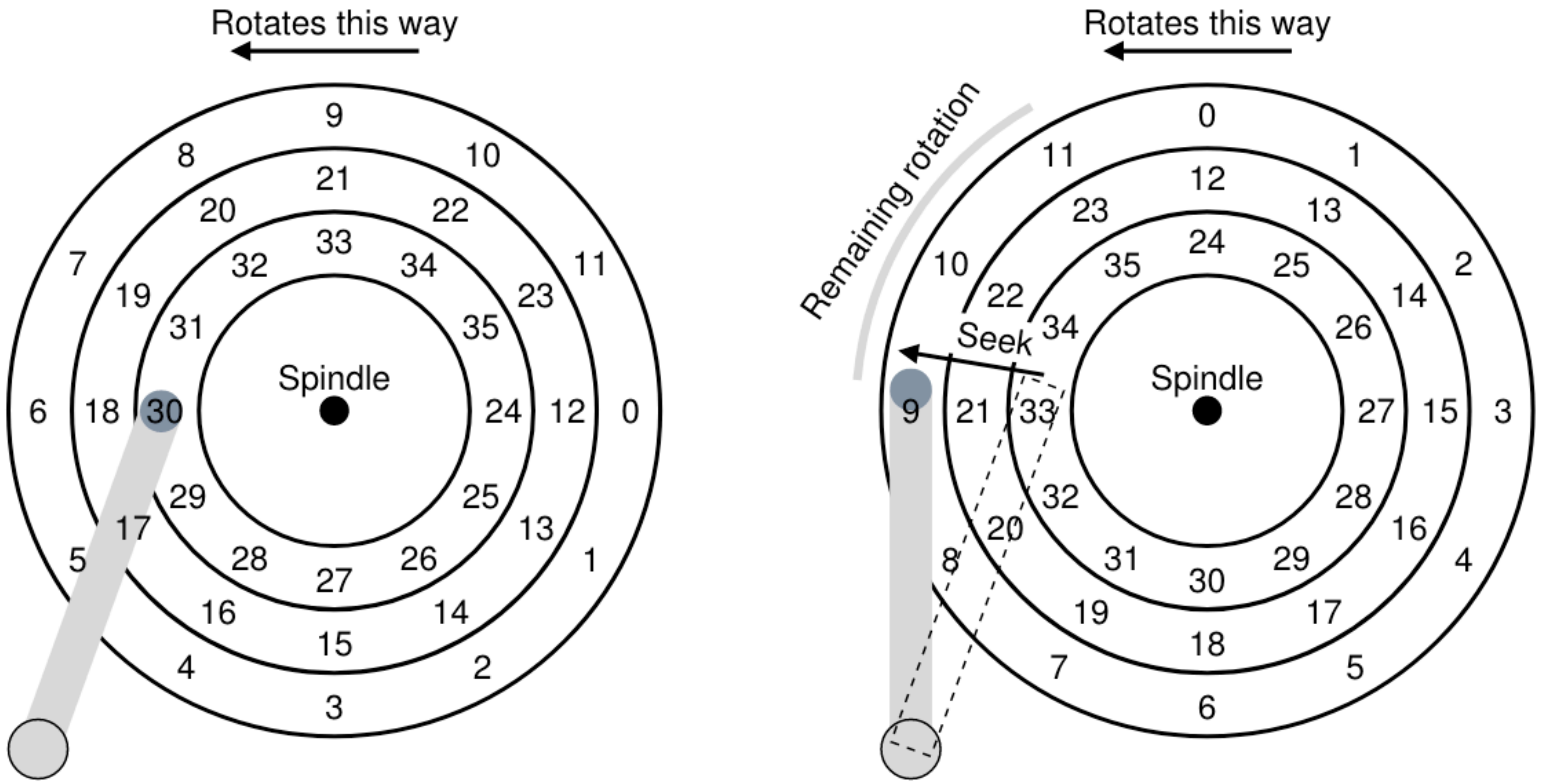
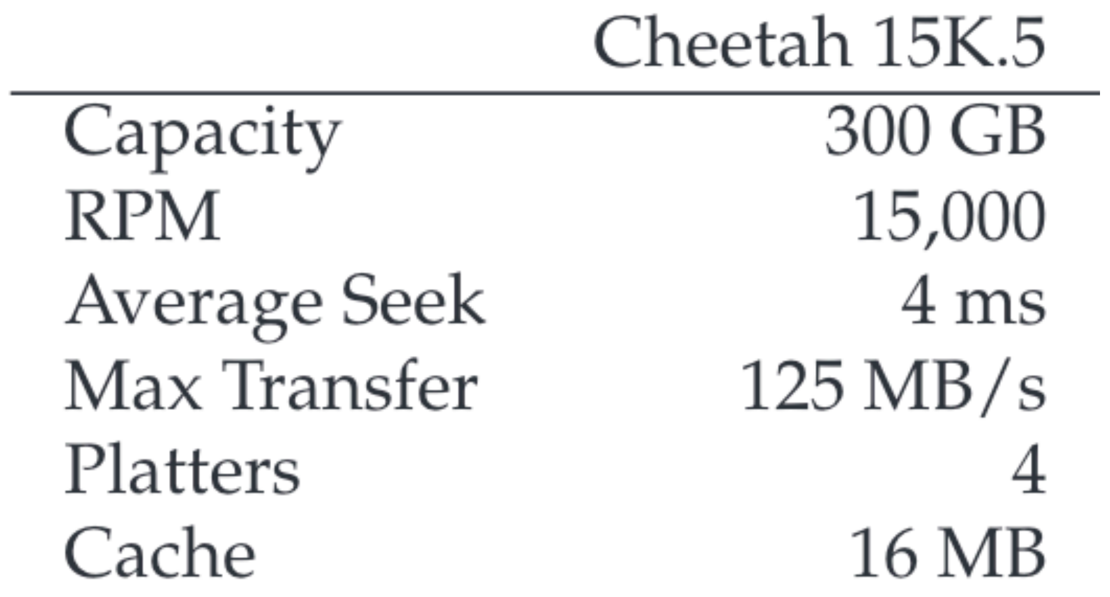


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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)

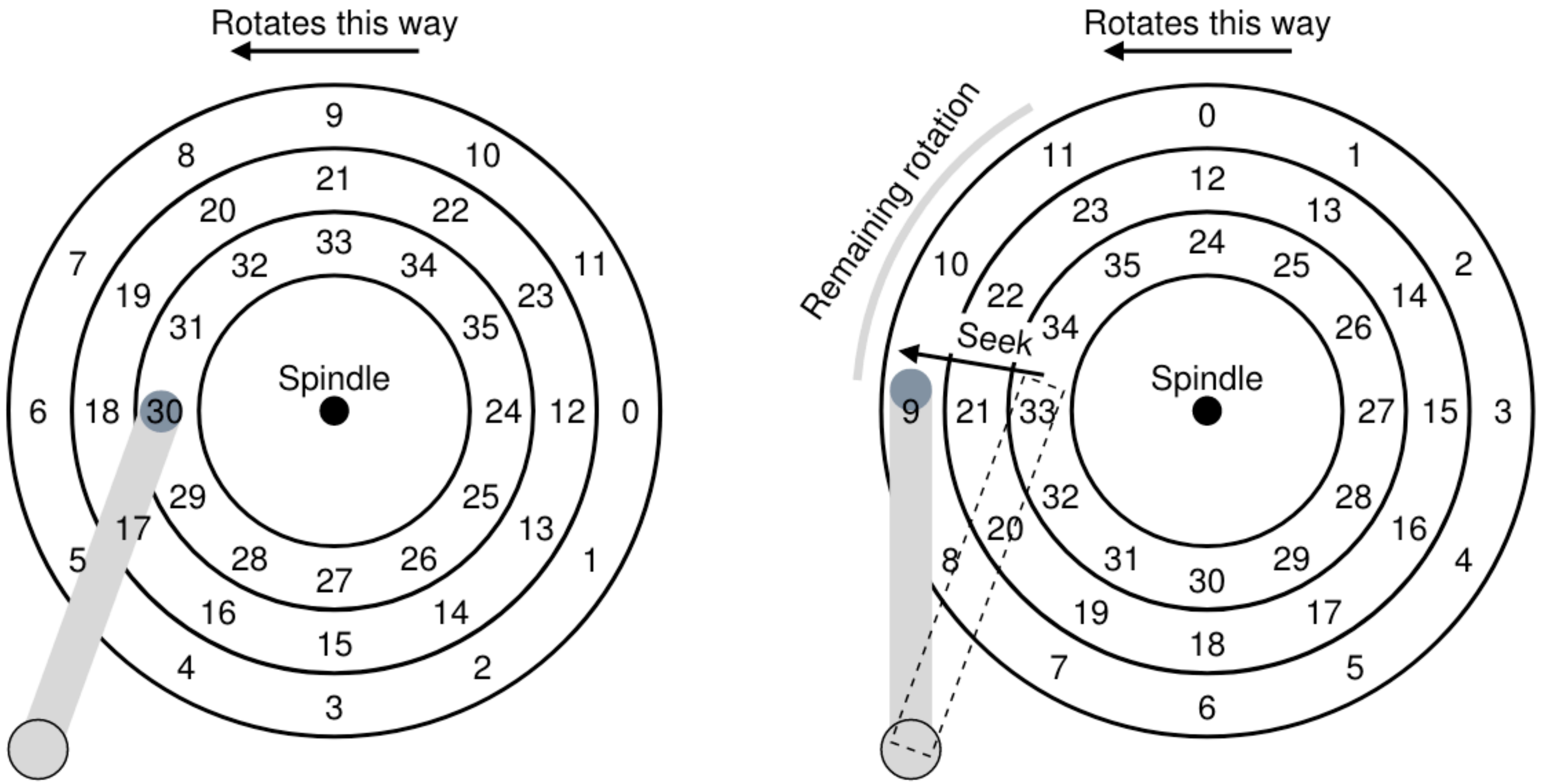
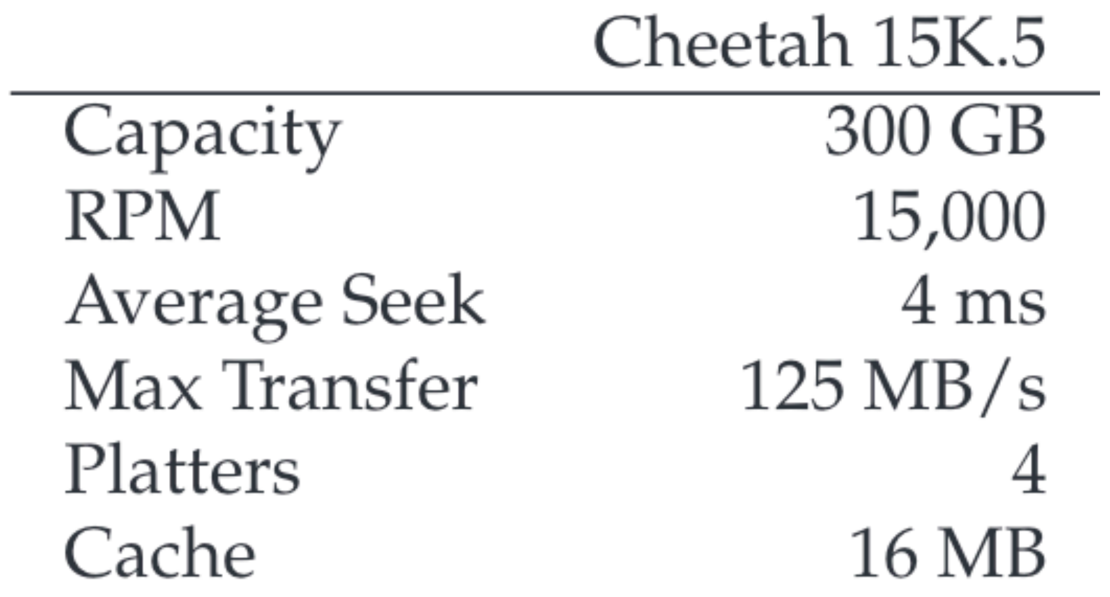


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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

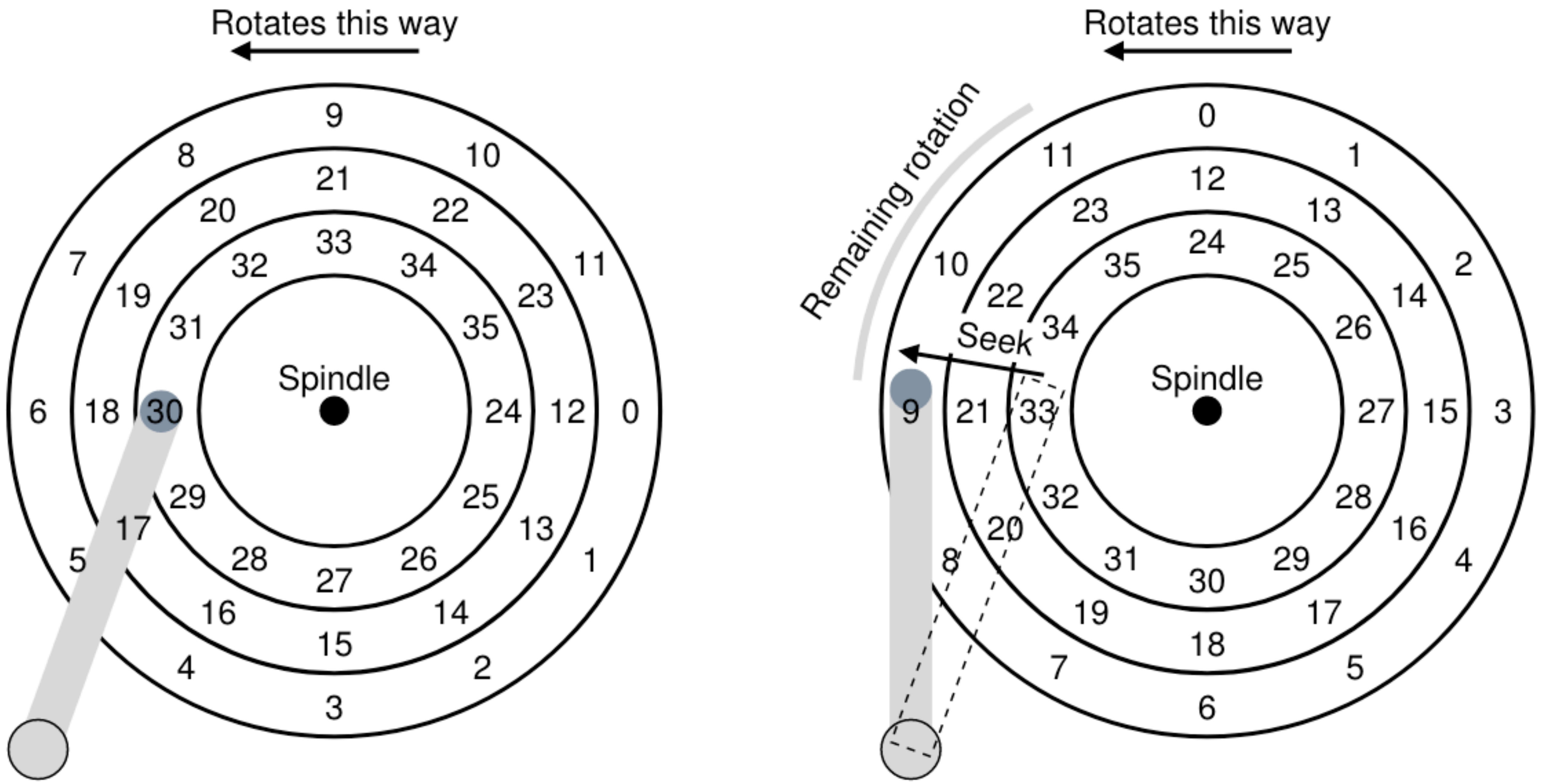
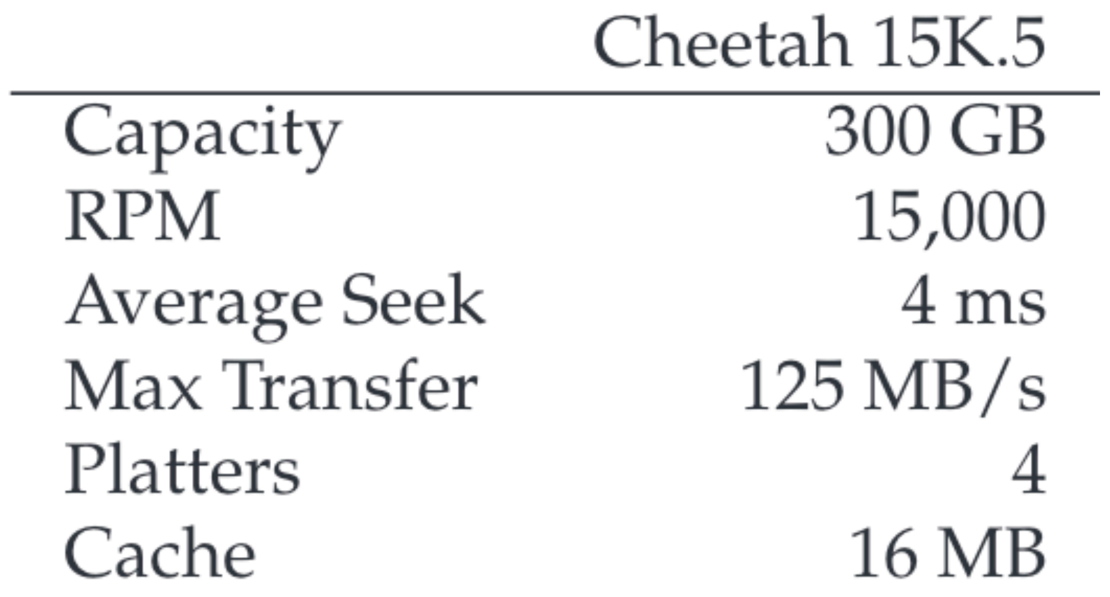


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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

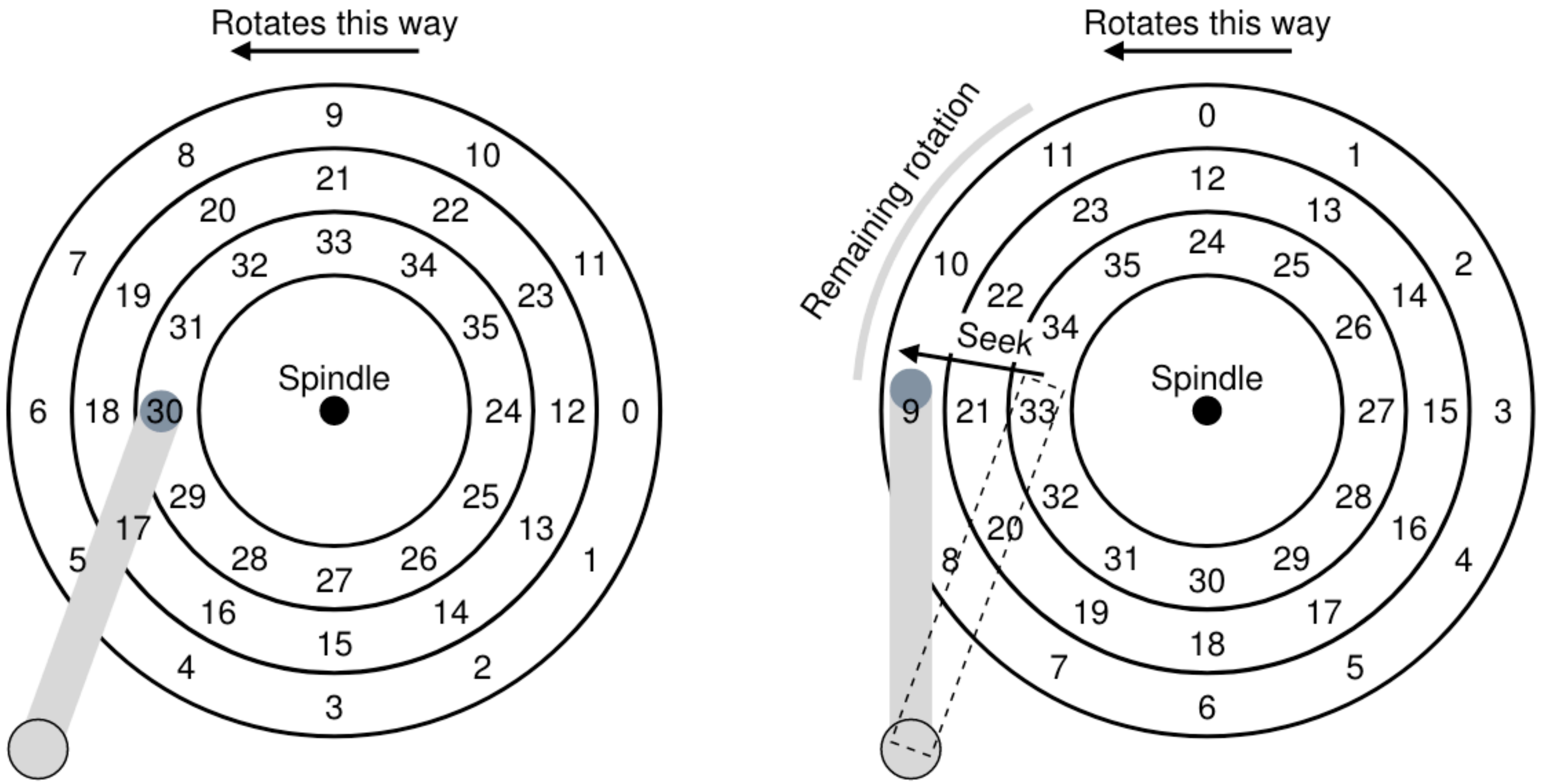
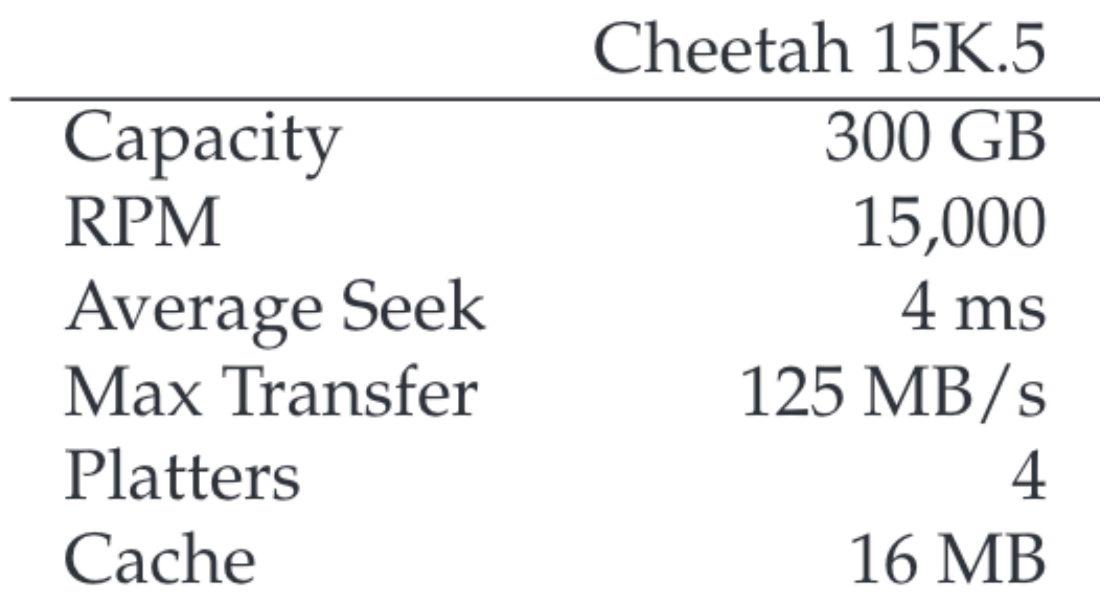


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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.

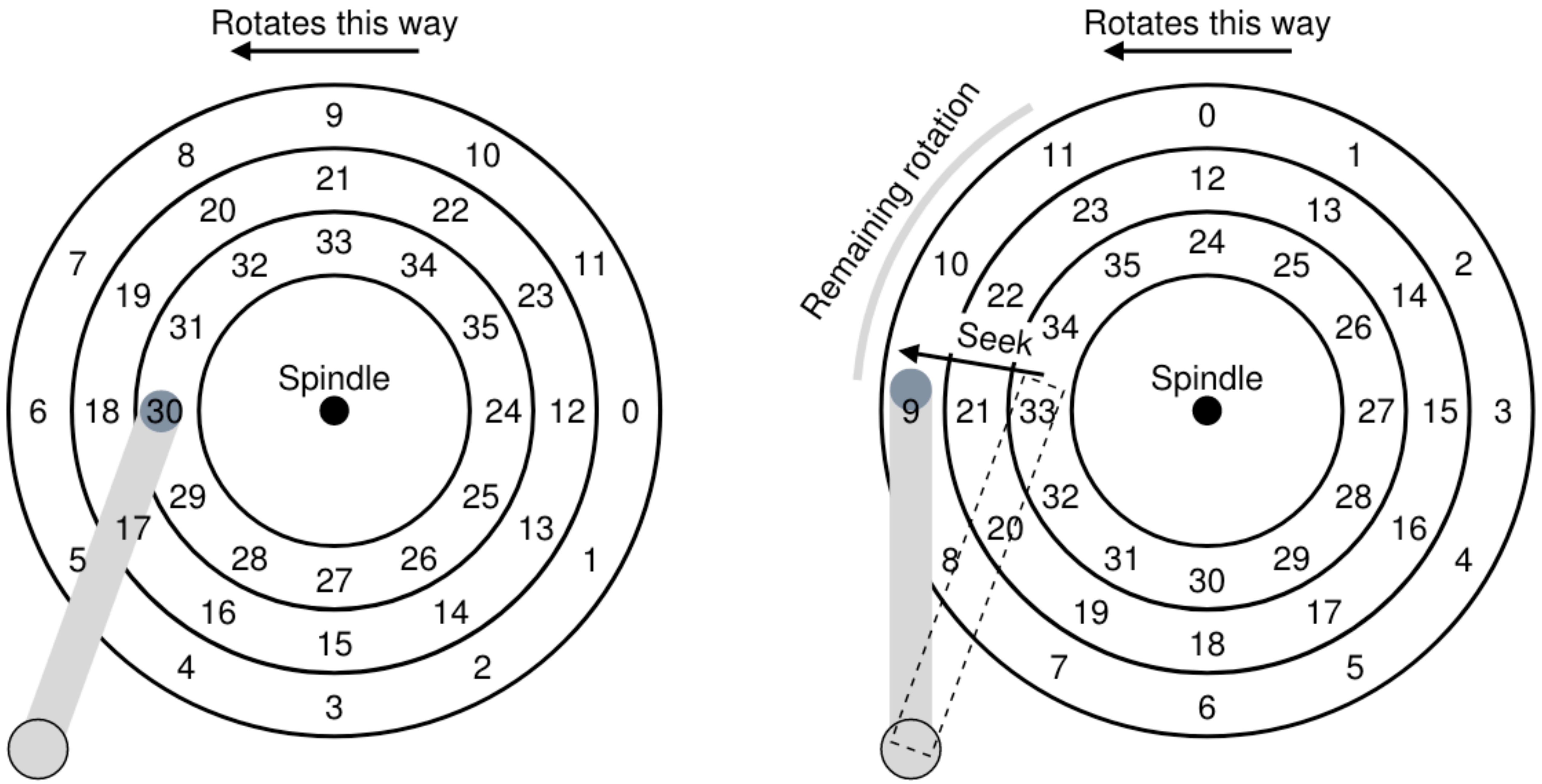
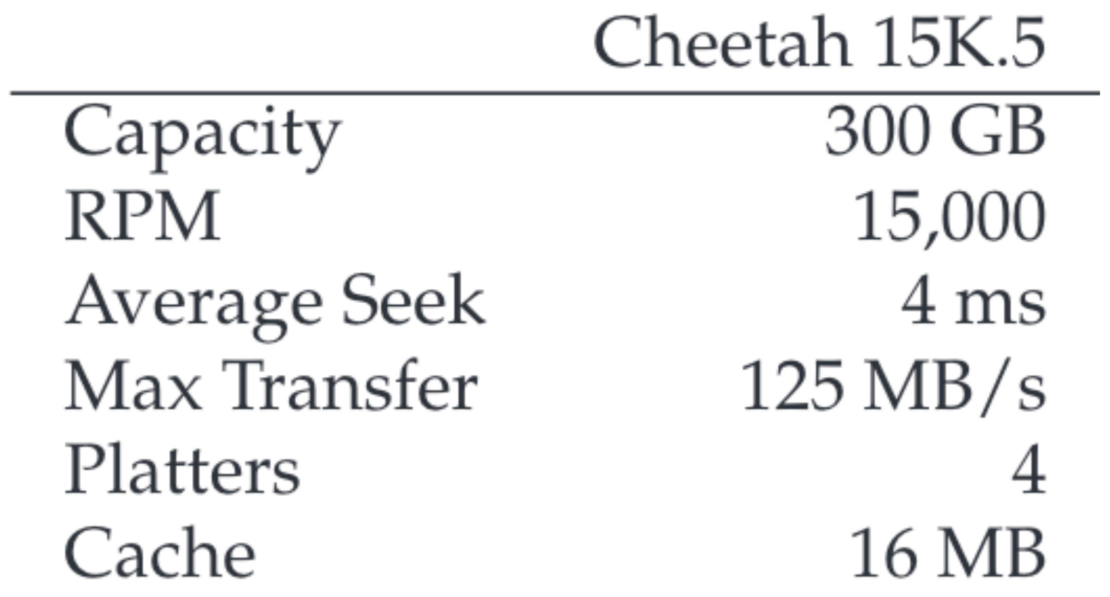


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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

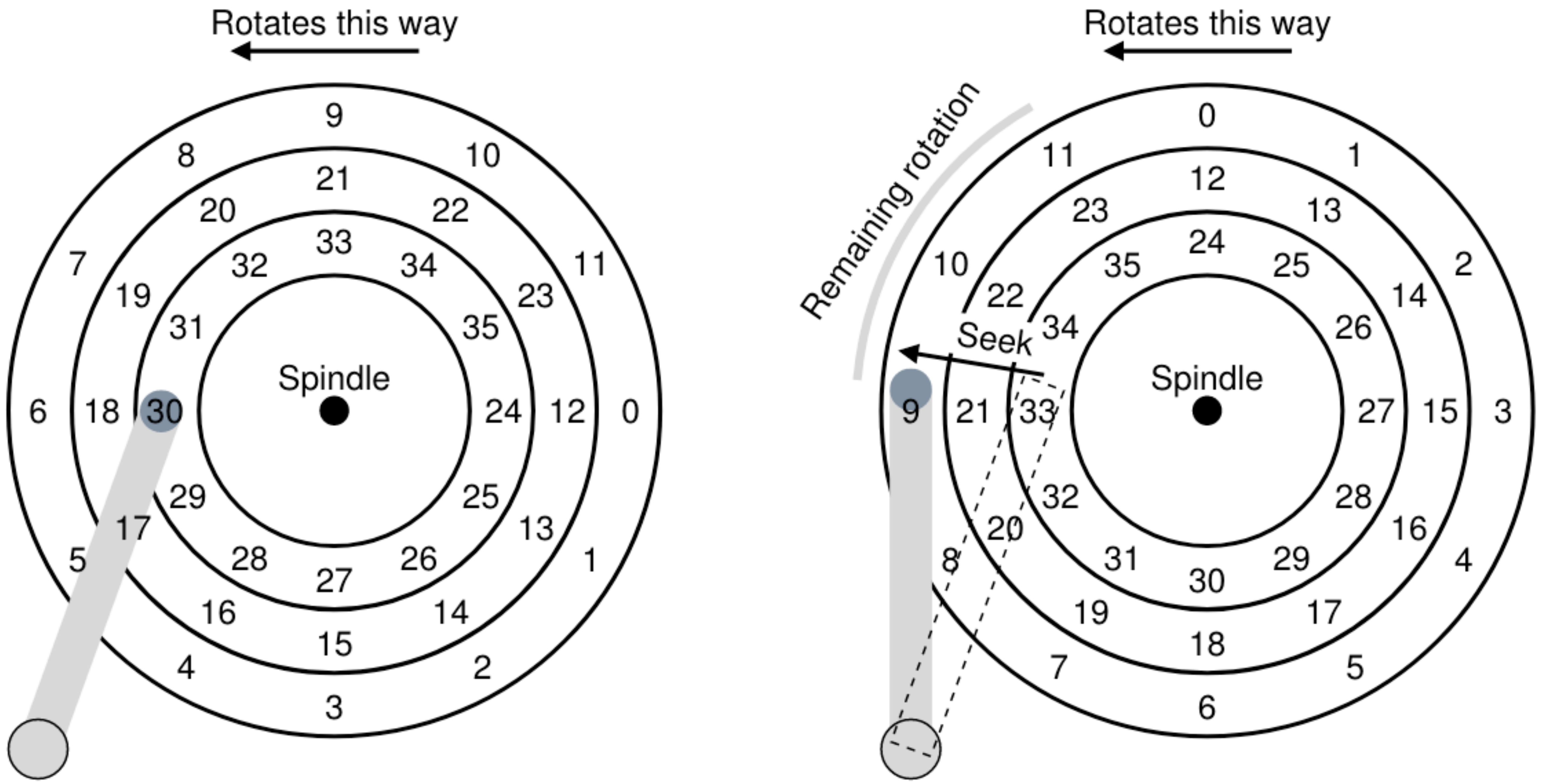
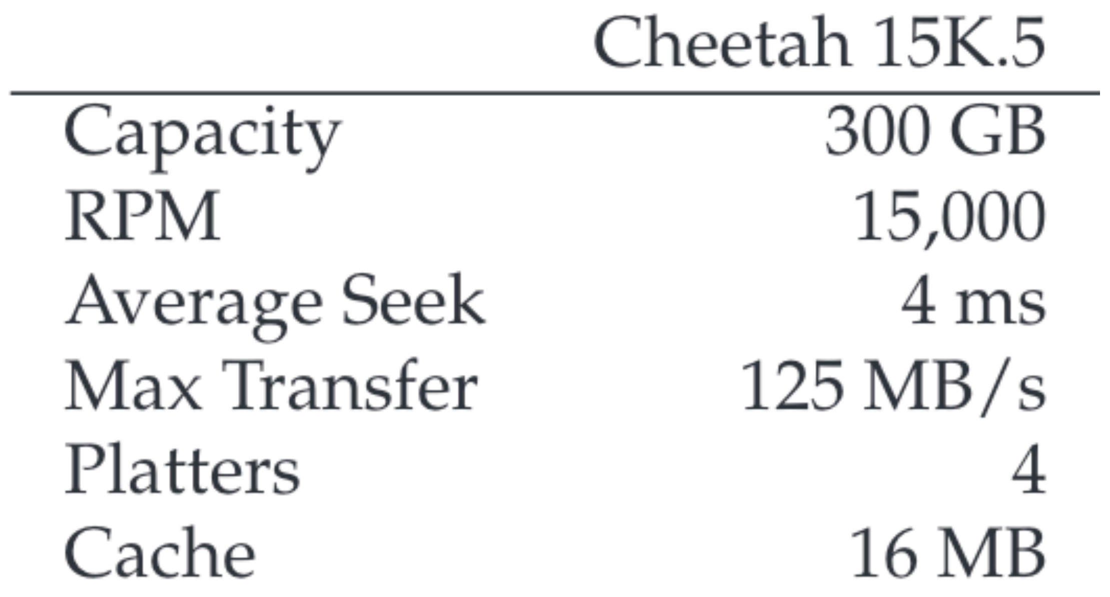


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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

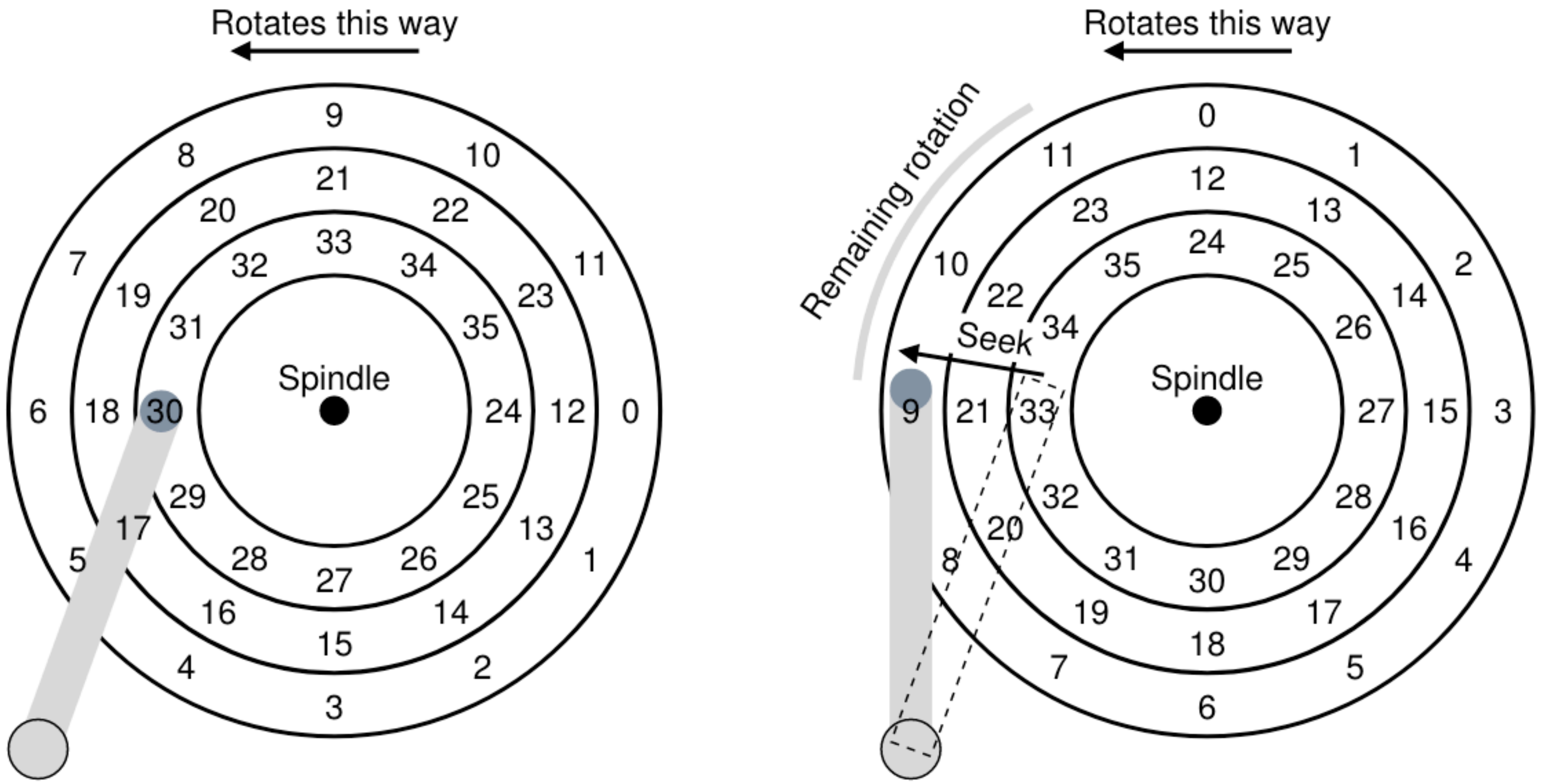


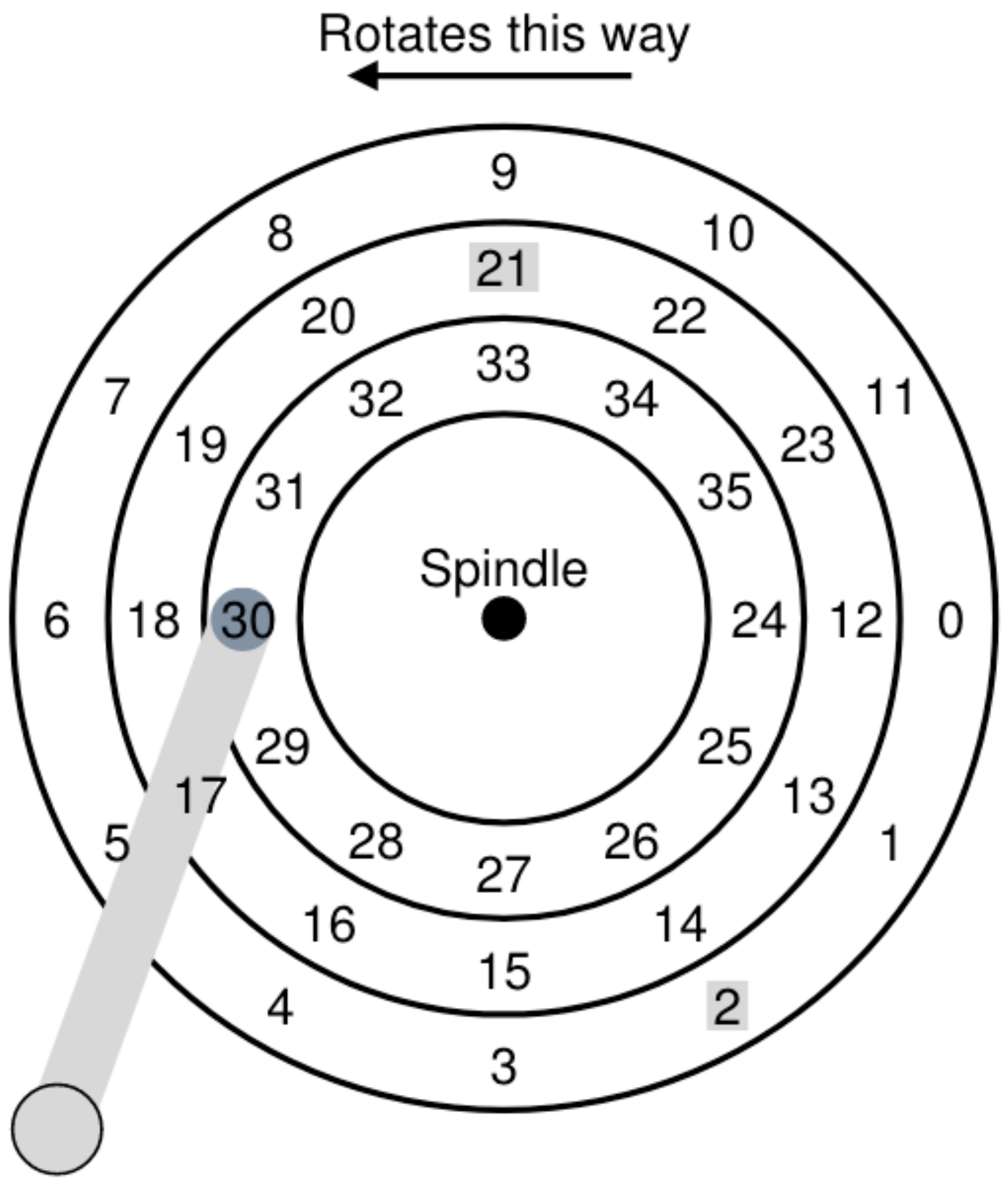
Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

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)

Disk scheduling problem

- python3 disk.py -a 10,15,32,11,33,16 -G
- Given a sequence of requests, reorder requests to service them quicker


Rotates this way
Spindle
9
8
10
21
20
22
33
7
32
34
11
19
23
31
35
6
18
30
24
12
0
29
25
17
13
5
28
26
1
27
16
14
15
4
2
3