I/O (2)

Not a Quiz

We have a single-core system with a preemptible kernel. We're concerned about data structure *X*, which is accessed by kernel threads as well as by the interrupt handler for dev.

a) **It's sufficient for threads to mask dev interrupts while accessing *X***
b) **In addition, threads must lock (blocking) mutexes before masking interrupts and accessing *X***
c) **b doesn't work. Instead, threads must lock spinlocks before accessing *X***
d) **In addition to c, the dev interrupt handler must lock a spinlock before accessing *X***
e) **Something else is needed**

Computer Terminal

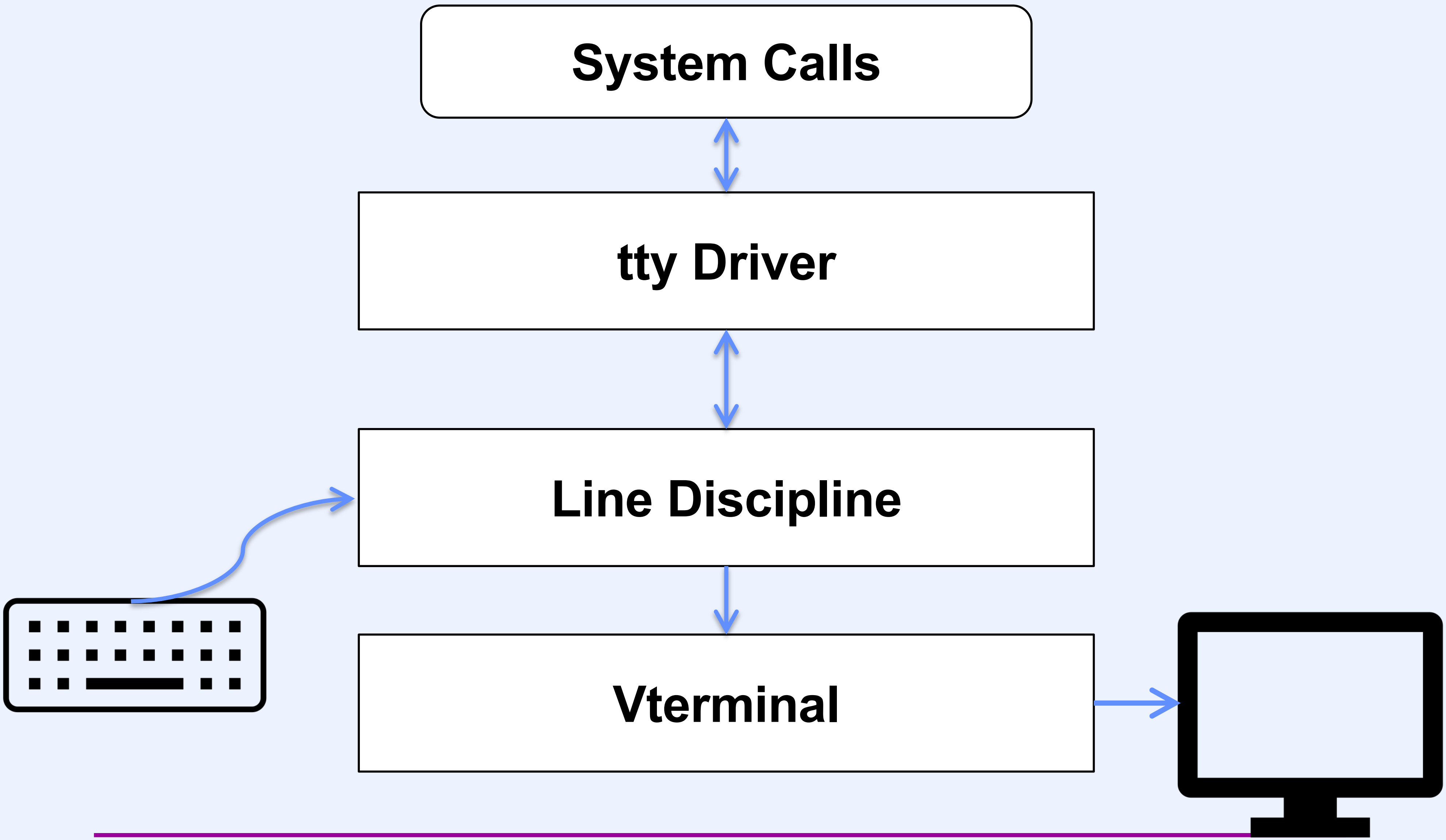
A “tty”

A Typewriter

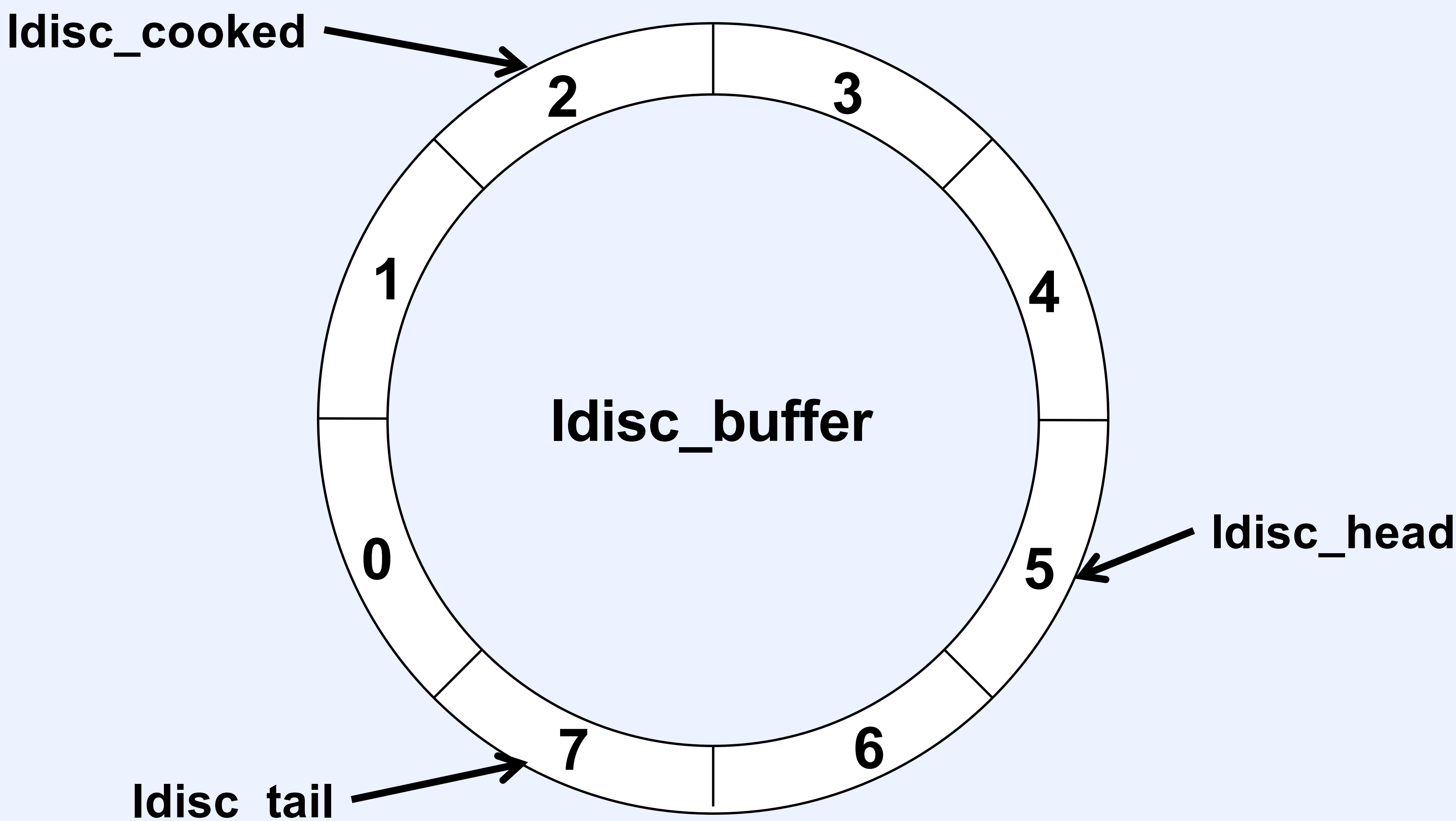
Terminals

- Long obsolete, but still relevant
- Issues
  1) characters are generated by the application faster than they can be sent to the terminal
  2) characters arrive from the keyboard even though there isn't a waiting read request from an application
  3) input characters may need to be processed in some way before they reach the application

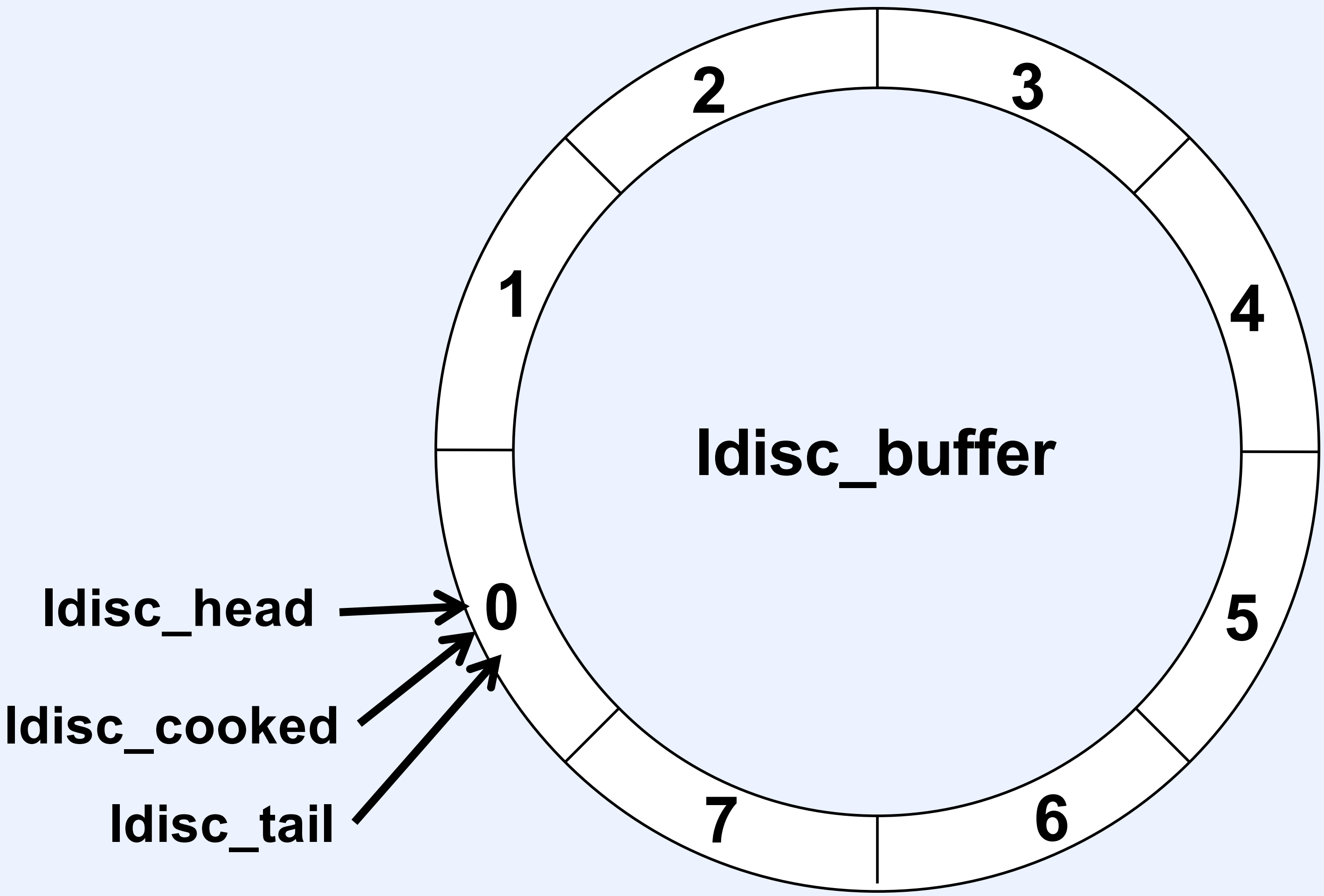
Terminals



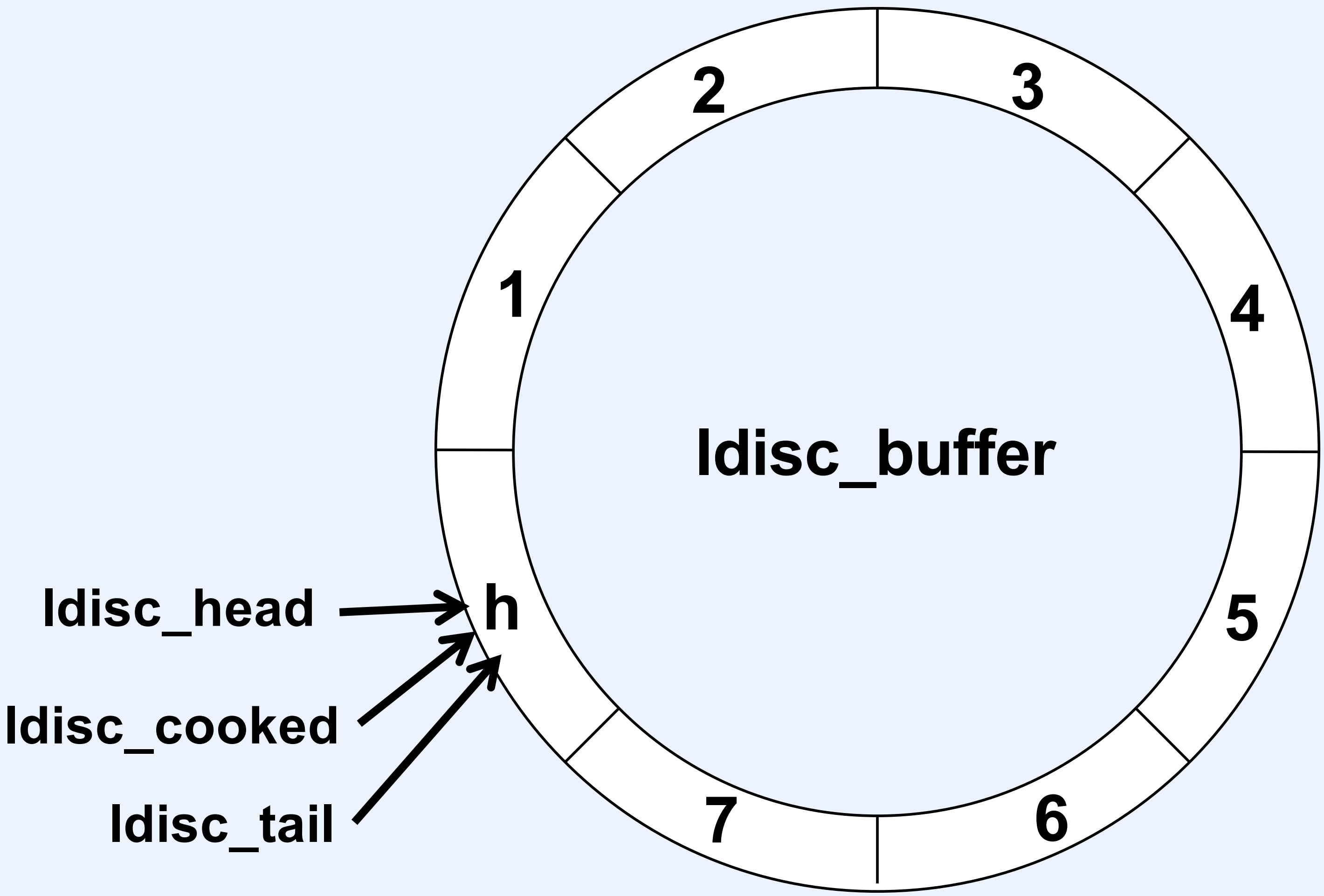
Line Discipline Buffer



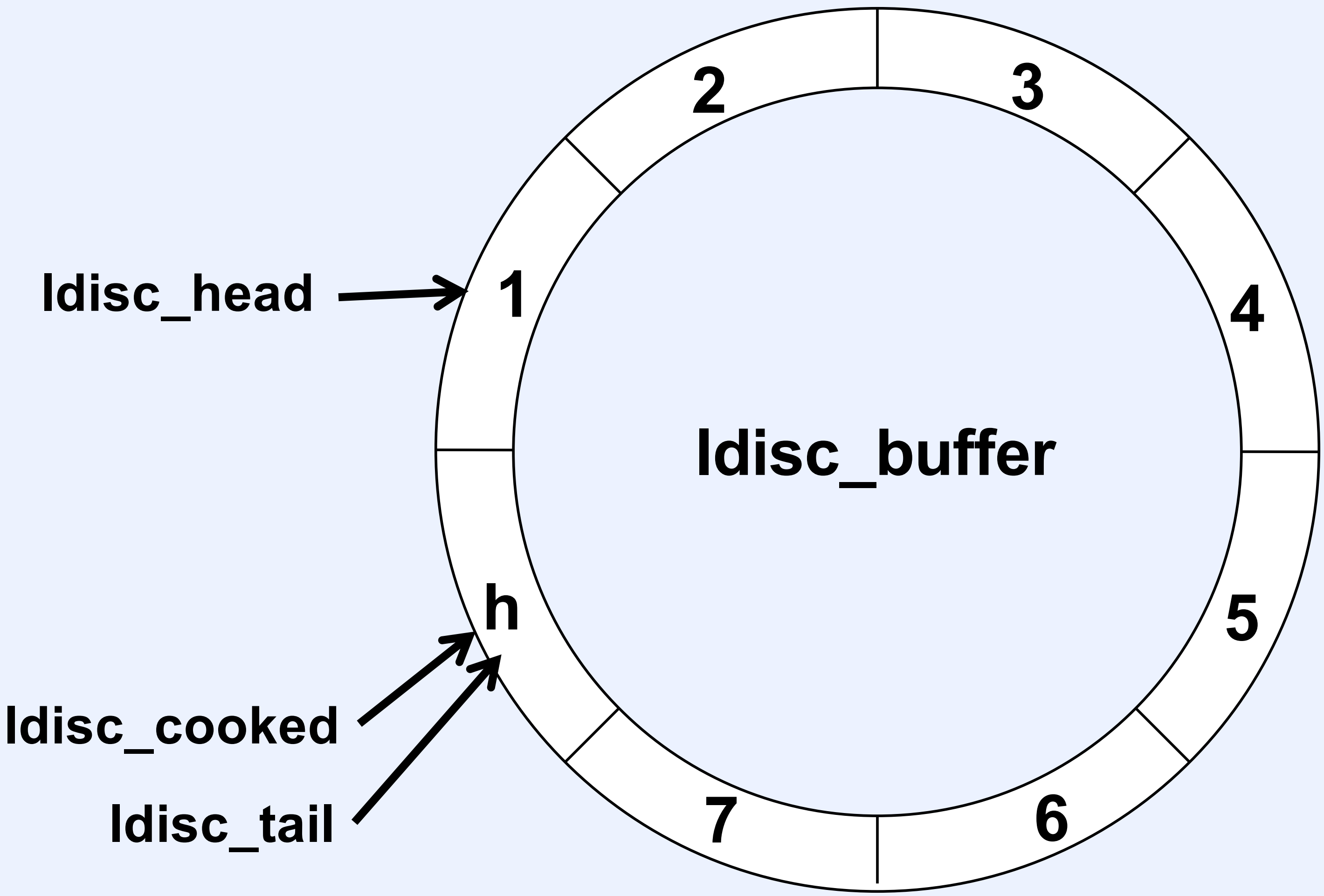
Line Discipline Buffer



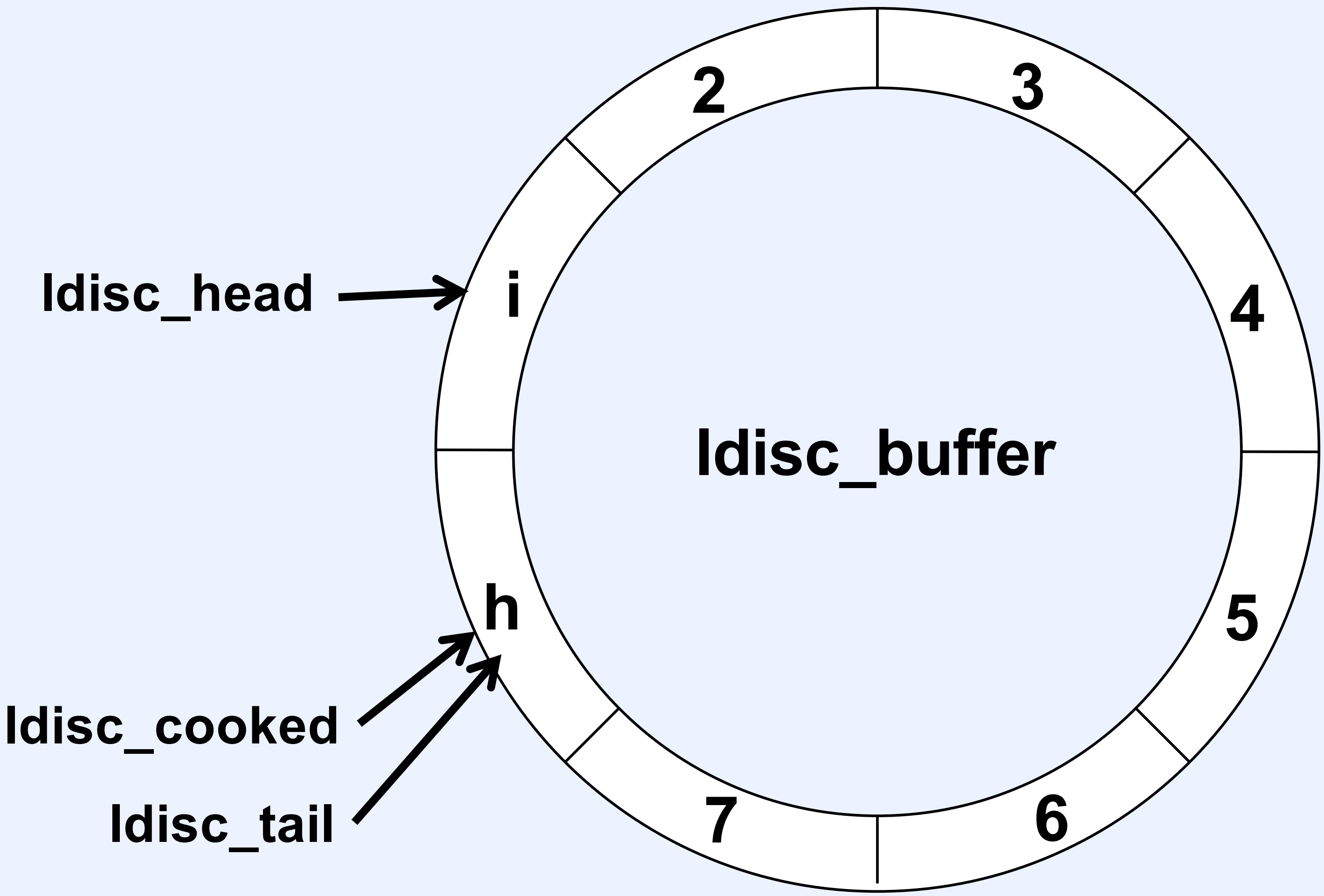
Line Discipline Buffer



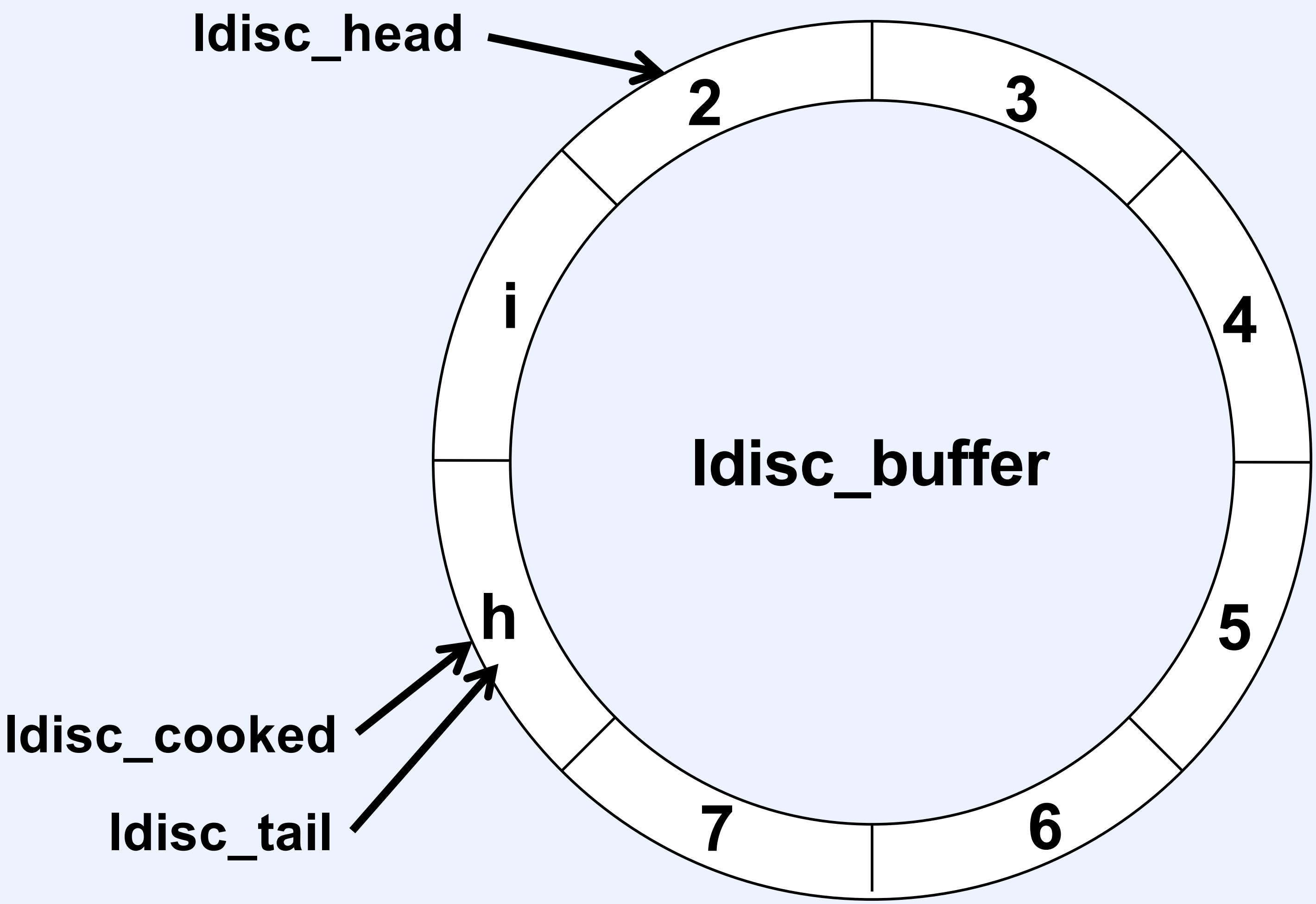
Line Discipline Buffer



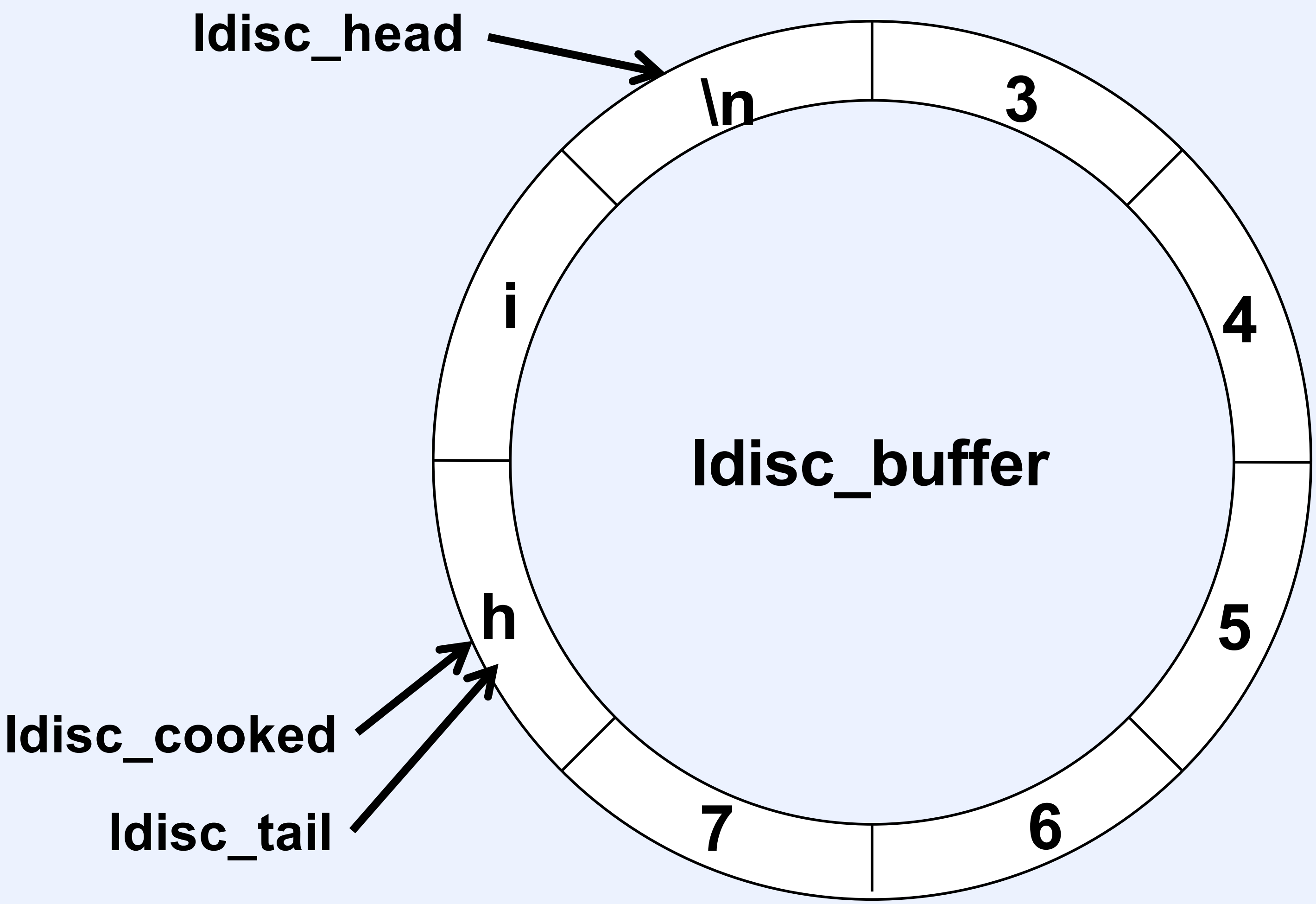
Line Discipline Buffer



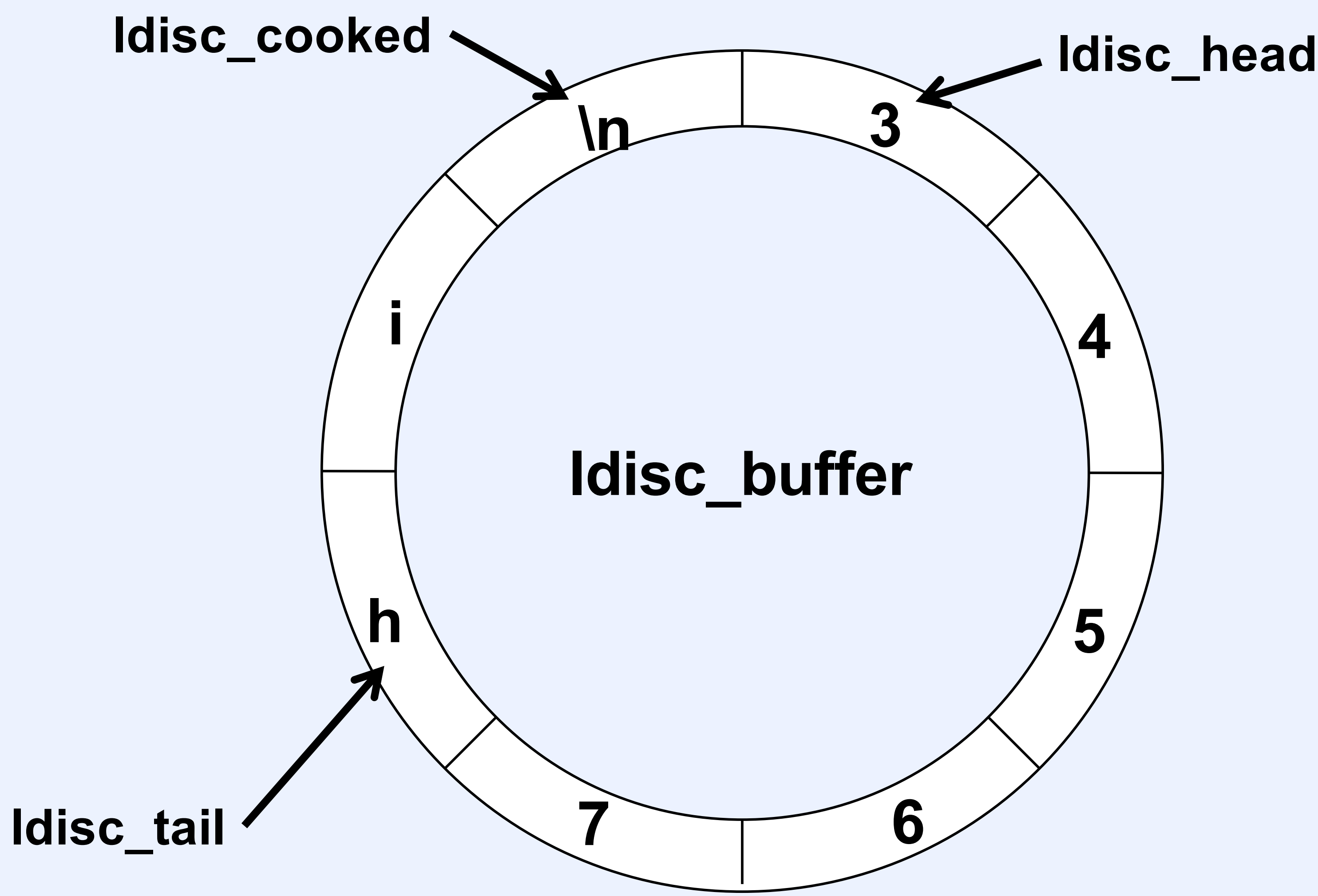
Line Discipline Buffer



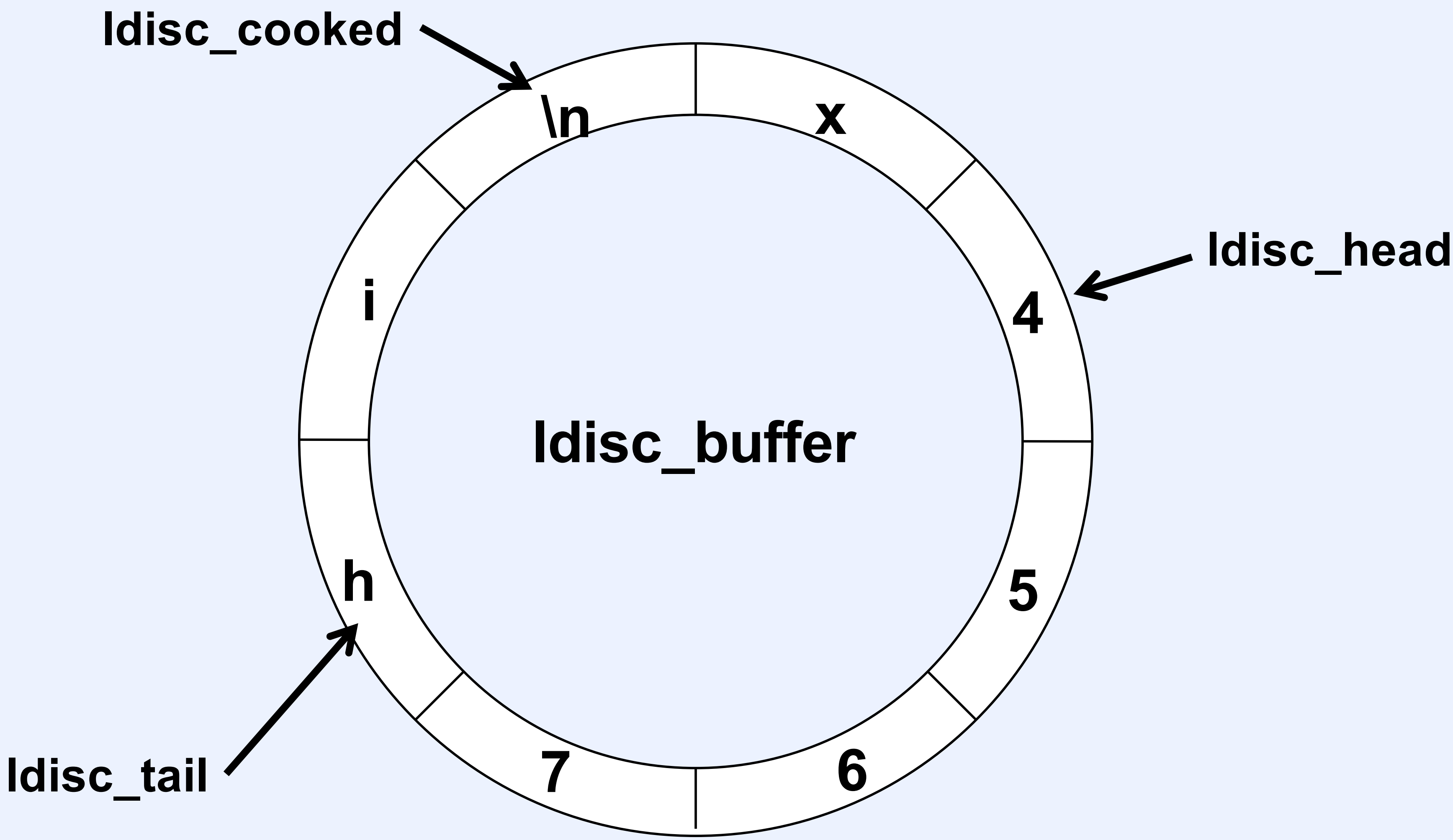
Line Discipline Buffer



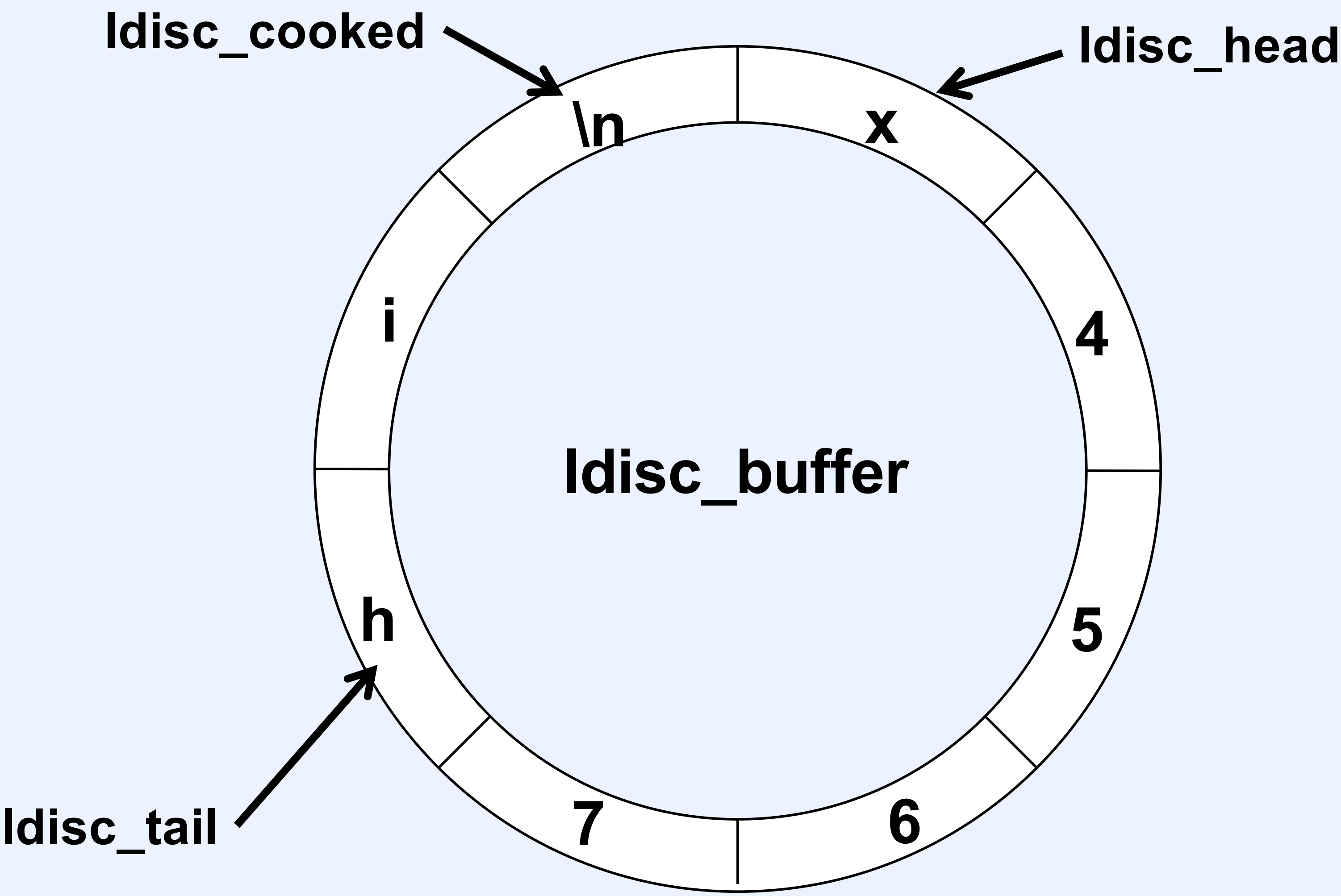
Line Discipline Buffer



Line Discipline Buffer



Line Discipline Buffer



Quiz 1

In which context are characters transformed from raw into cooked?

a) **In the context of the thread performing the *read* system call**

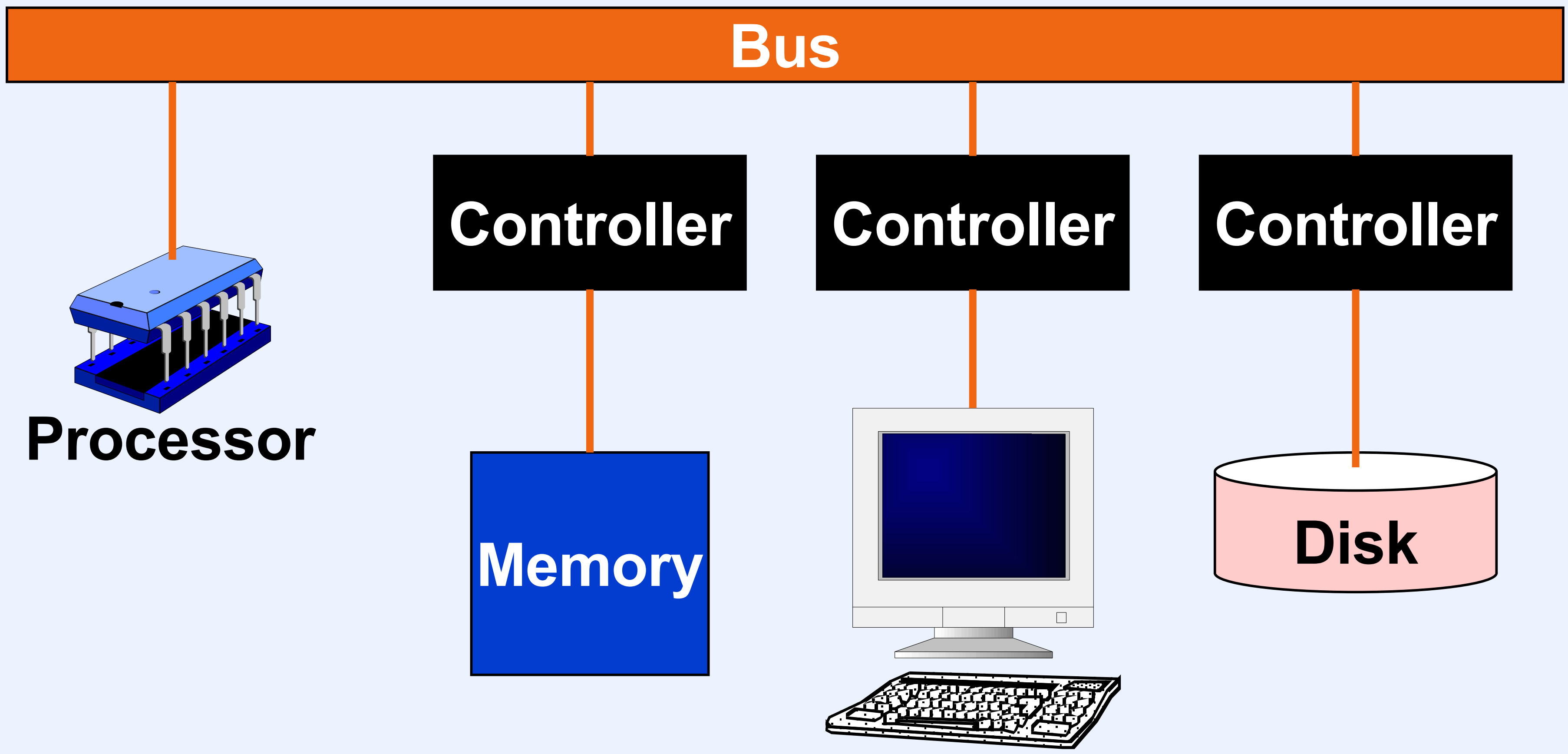
b) **In the interrupt context (i.e., on a “borrowed” stack)**

c) **Some other context**

Input/Output

- **Architectural concerns**
  - memory-mapped I/O
    - programmed I/O (PIO)
    - direct memory access (DMA)
  - I/O processors (channels)
- **Software concerns**
  - device drivers
  - concurrency of I/O and computation

Simple I/O Architecture



PIO Registers

GoR	**GoW**	**IER**	**IEW**					**Control register**
RdyR	**RdyW**							**Status register**
								Read register
								Write register

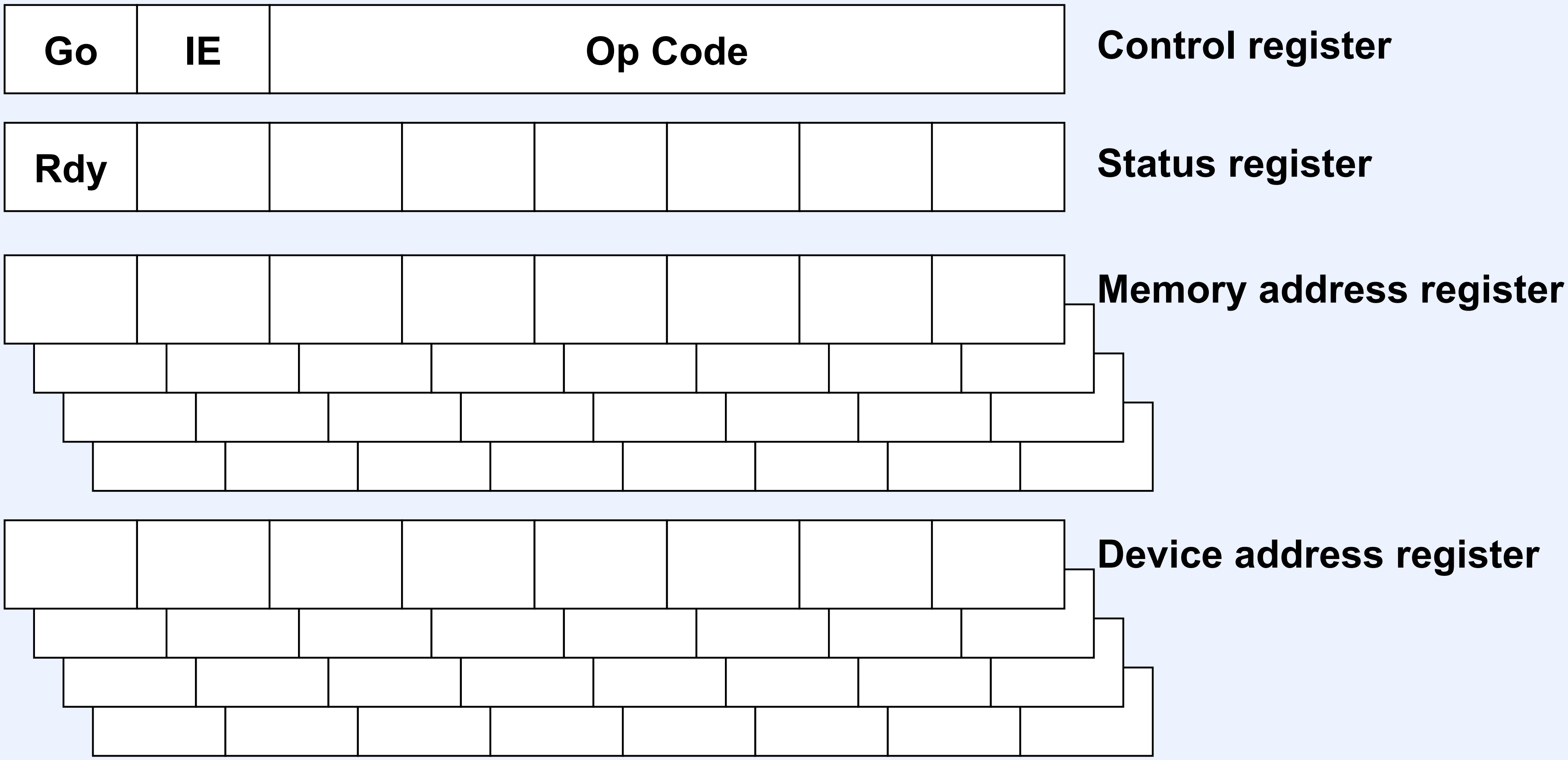
Legend:

GoR	Go read (start a read operation)
GoW	Go write (start a write operation)
IER	Enable read-completion interrupts
IEW	Enable write-completion interrupts
RdyR	Ready to read
RdyW	Ready to write

Programmed I/O

- E.g.: Terminal controller
- Procedure (write)
  - write a byte into the *write register*
  - set the WGO bit in the *control register*
  - wait for WREADY bit (in *status register*) to be set (if interrupts have been enabled, an interrupt occurs when this happens)

DMA Registers

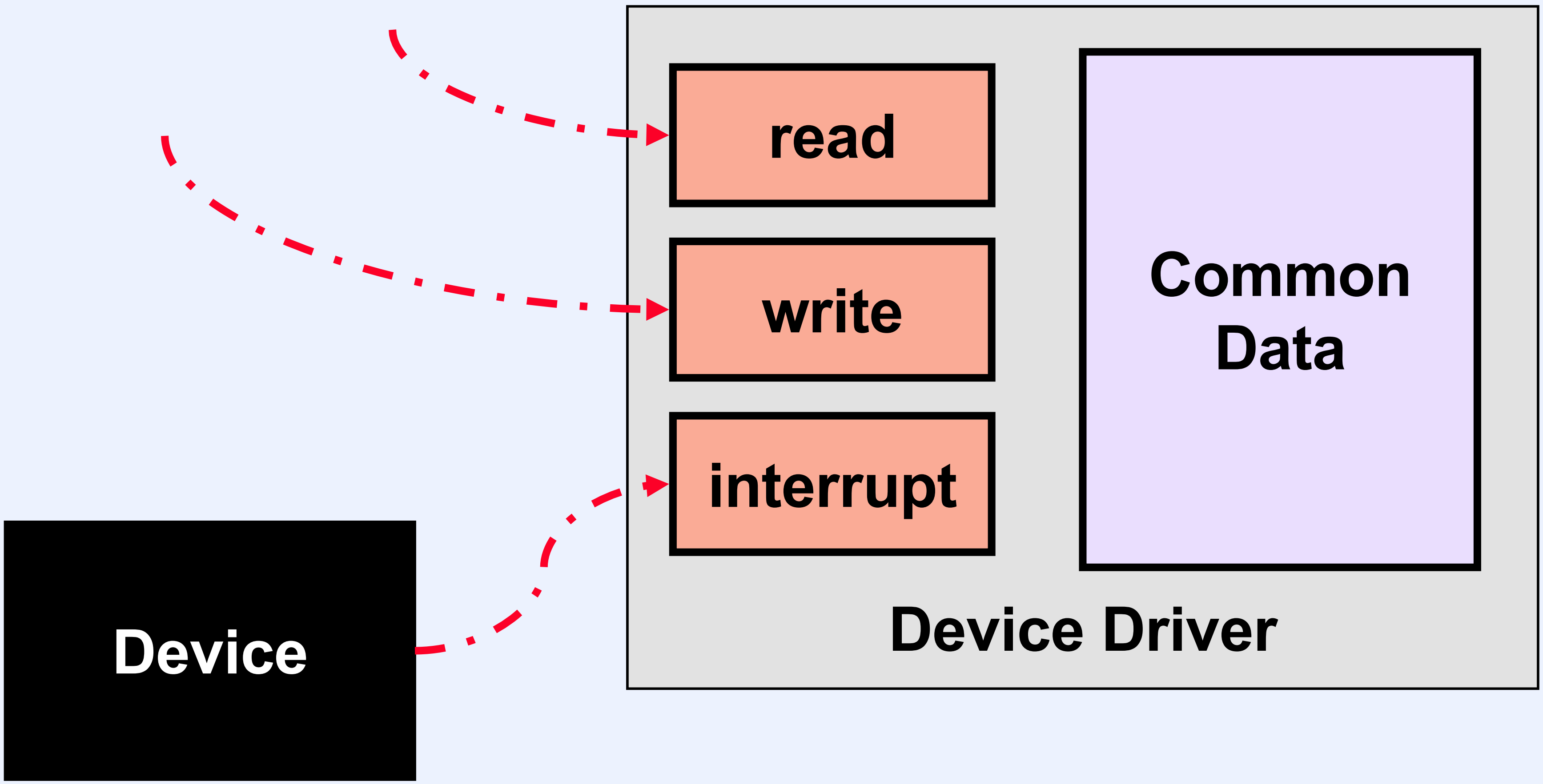


Legend:	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready

Direct Memory Access

- E.g.: Disk controller
- Procedure
  - set the *disk address* in the *device address register* (only relevant for a seek request)
  - set the *buffer address* in the *memory address register*
  - set the *op code* (SEEK, READ or WRITE), the GO bit and, if desired, the interrupt ENABLE bit in the *control register*
  - wait for interrupt or for READY bit to be set

Device Drivers



PDP-8


digital PDP-8/I
DIGITAL EQUIPMENT CORPORATION/MAYNARD MASSACHUSETTS
Rim Loader
7756 6032
7757 6031
7760 5357
7761 6036
7762 7106
7763 7006
7764 7510
7765 5357
7766 7006
7767 6031
7770 5367
7771 6034
7772 7420
7773 3776
7774 3376
7775 5356
Data Field
Inst Field
Program Counter
Memory Address
Memory Buffer
Link
Accumulator
Step Counter
Multiplier Quotient
And
Tad
Isz
Dca
Jms
Jmp
Iot
Opr
Fetch
Execute
Defer
Word Count
Current Address
Break
Ion
Pause
Run
Start
Load Add
Dep
Exam
Cont
Stop
Sing Step
Sing Inst
POWER
PANEL LOCK

PDP-8 Boot Code

```
07756 6032 KCC
07757 6031 KSF
07760 5357 JMP .-1
07761 6036 KRB
07762 7106 CLL RTL
07763 7006 RTL
07764 7510 SPA
07765 5357 JMP 7757
07766 7006 RTL
07767 6031 KSF
07770 5367 JMP .-1
07771 6034 KRS
07772 7420 SNL
07773 3776 DCA I 7776
07774 3376 DCA 7776
07775 5356 JMP 7756
07776 0000 AND 0
07777 5301 JMP 7701
```


VAX-11/780

VAX-11/780 Boot

- Separate “console computer”
  - LSI-11
  - read boot code from floppy disk
  - load OS from root directory of first file system on primary disk

Configuring the OS (1)

- Early Unix
  – OS statically linked to contain all needed device drivers
    - all device-specific info included with drivers
  – disk drivers contained partitioning description

Configuring the OS (2)

- Later Unix
  - OS statically linked to contain all needed device drivers
    - at boot time, OS would probe to see which devices were present and discover device-specific info
  - partition table in first sector of each disk

IBM PC

Issues

- Open architecture
  – large market for peripherals, most requiring special drivers
  – how to access boot device?
  – how does OS get drivers for new devices?

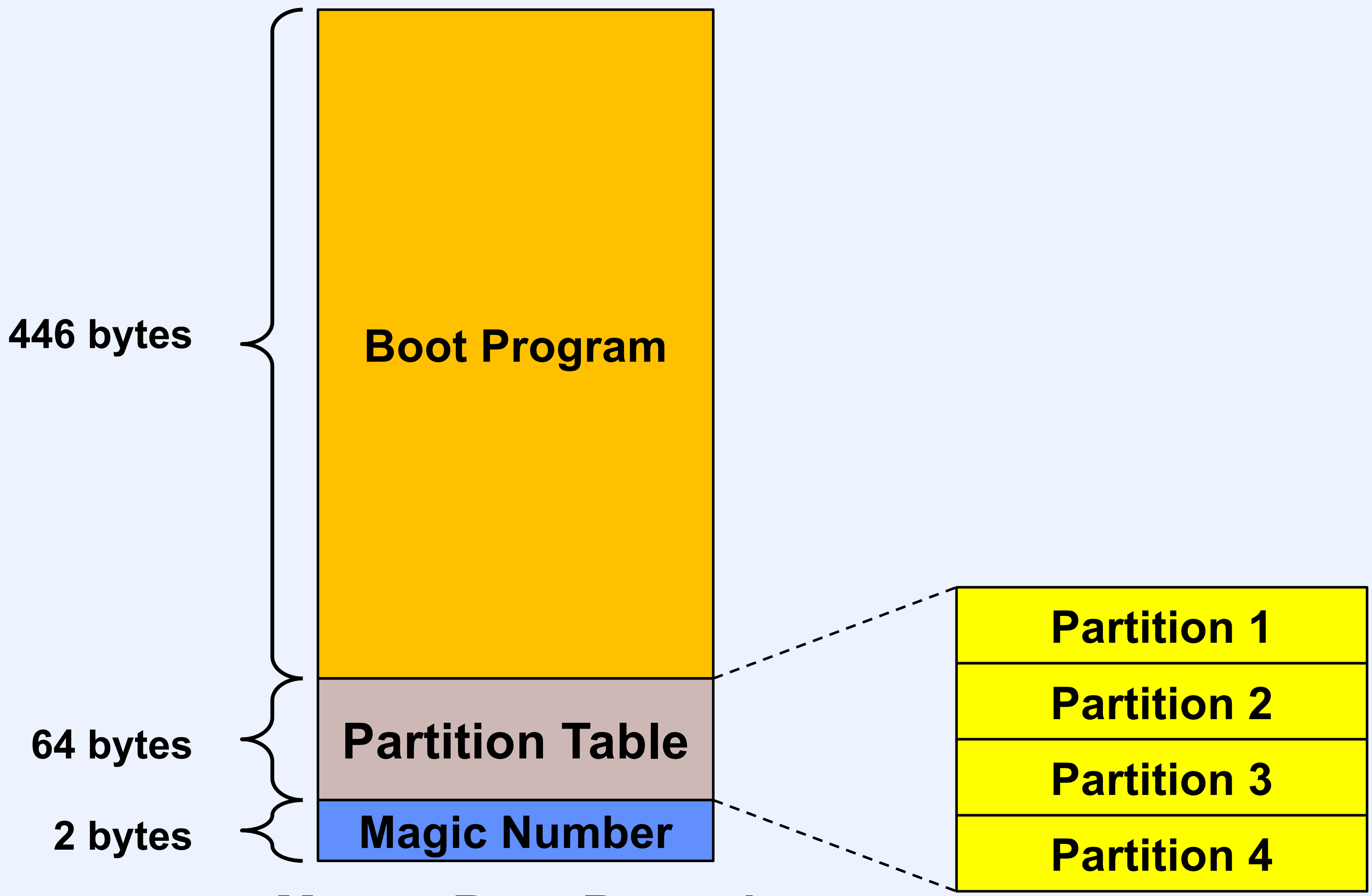
The Answer: BIOS

- Basic Input-Output System
  - code stored in *non-volatile RAM*
    - CMOS, flash, whatever …
  - configuration data also in NV RAM
    - including set of boot-device names
  - three primary functions
    - power-on self test (POST)
    - load and transfer control to boot program
    - provide drivers for all devices
  - main BIOS on motherboard
  - additional BIOSes on other boards

POST

- On power-on, CPU executes BIOS code
  - located in last 64k of first megabyte of address space
  - initializes hardware
  - counts memory locations

Getting the Boot Program



Master Boot Record

Linux Booting (1)

- Two stages of booting provided by one of:
  - lilo (Linux Loader)
    - uses sector numbers of kernel image
  - grub (Grand Unified Boot Manager)
    - understands various file systems
  - both allow dual (or greater) booting
    - select which system to boot from menu
      - perhaps choice of Linux or Windows

Linux Booting (2)

assembler code (*startup_32*)

- **Kernel image is compressed**
  - step 1: set up stack, clear BSS, uncompress kernel, then transfer control to it

assembler code (different *startup_32*)

- **Process 0 is created**
  - step 2: set up initial page tables, turn on address translation

C code (*start_kernel*)

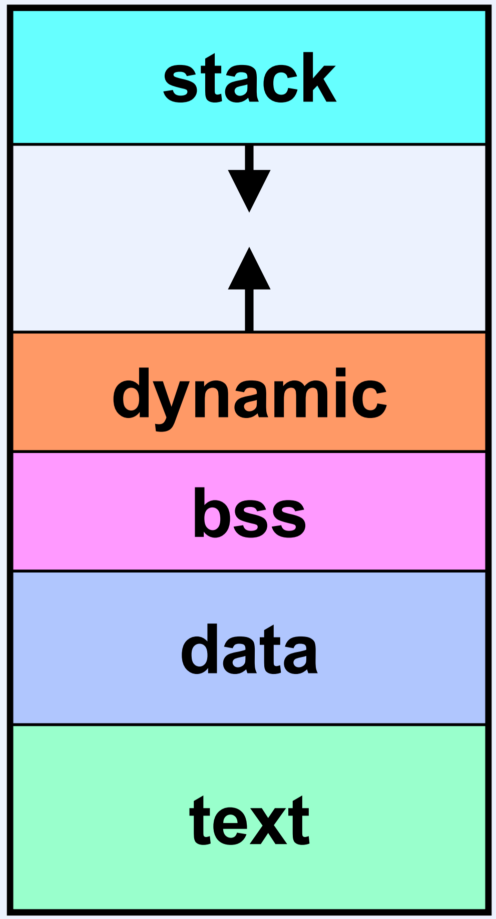
- **Do further initialization**
  - step 3: initialize rest of kernel, create init process (#1)
  - invoke scheduler

Beyond BIOS

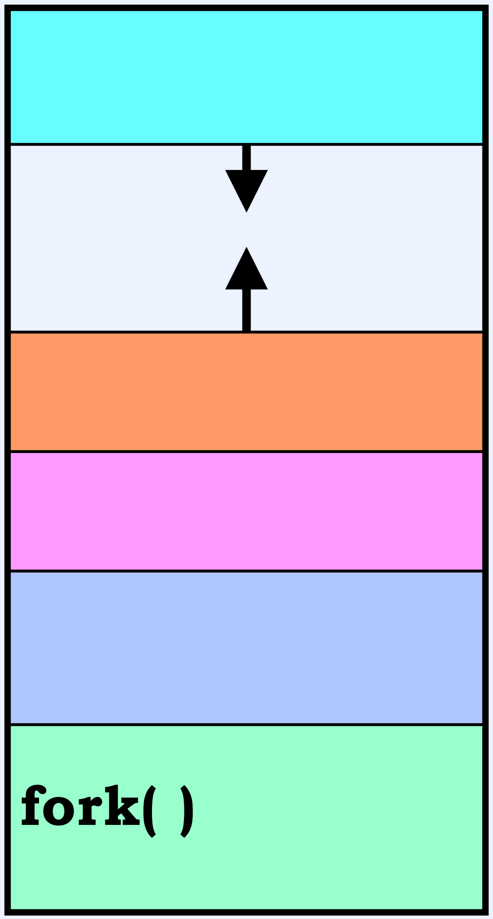
- BIOS
  - designed for 16-bit x86 of mid 1980s
- Open Firmware
  - designed by Sun in the 1990s
  - portable
    - drivers, boot code in Forth
      - compiled into bytecode
  - used on non-Intel systems
- UEFI (Unified Extensible Firmware Interface)
  - improved BIOS originally from Intel
  - also uses bytecode

UNIX Structure

The Unix Address Space

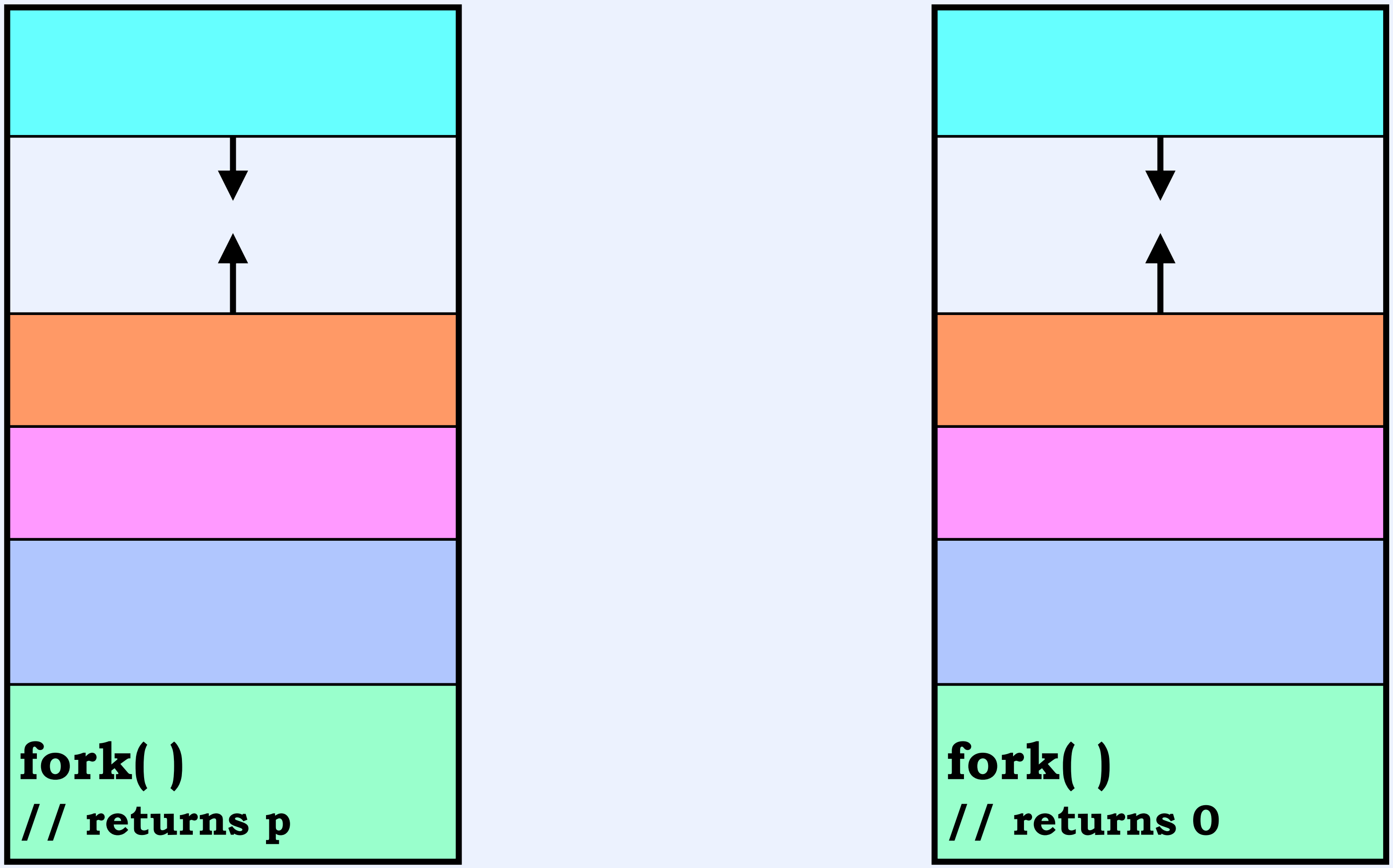


Creating a Process: Before



parent
process

Creating a Process: After



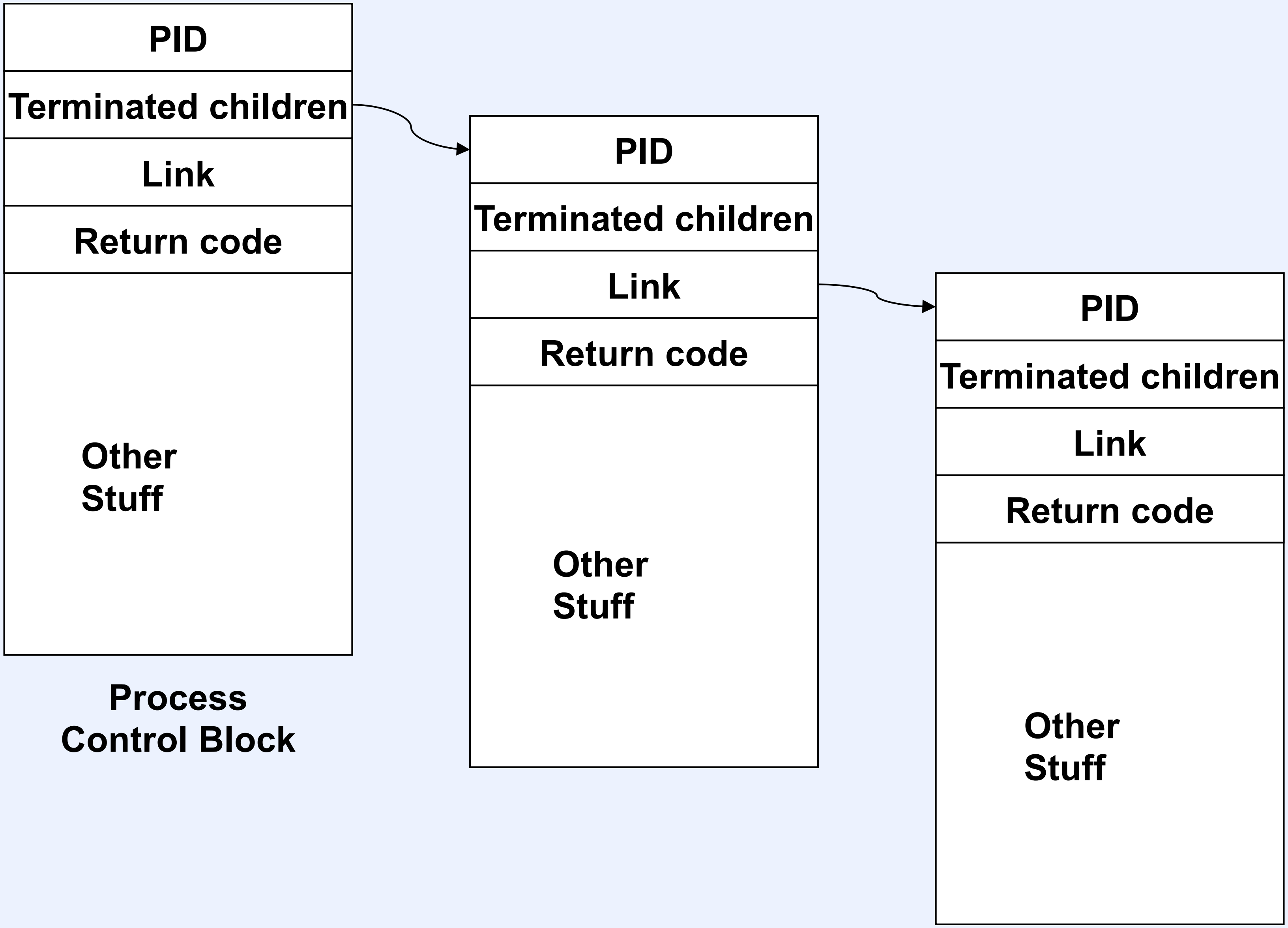
parent process

child process (pid = p)

Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
   /* some code is here for the child to execute */
   exit(n);
} else {
   int ReturnCode;
   while(pid != wait(&ReturnCode))
      ;
   /* the child has terminated with ReturnCode as its
      return code */
}
```

Process Control Blocks



Process
Control Block

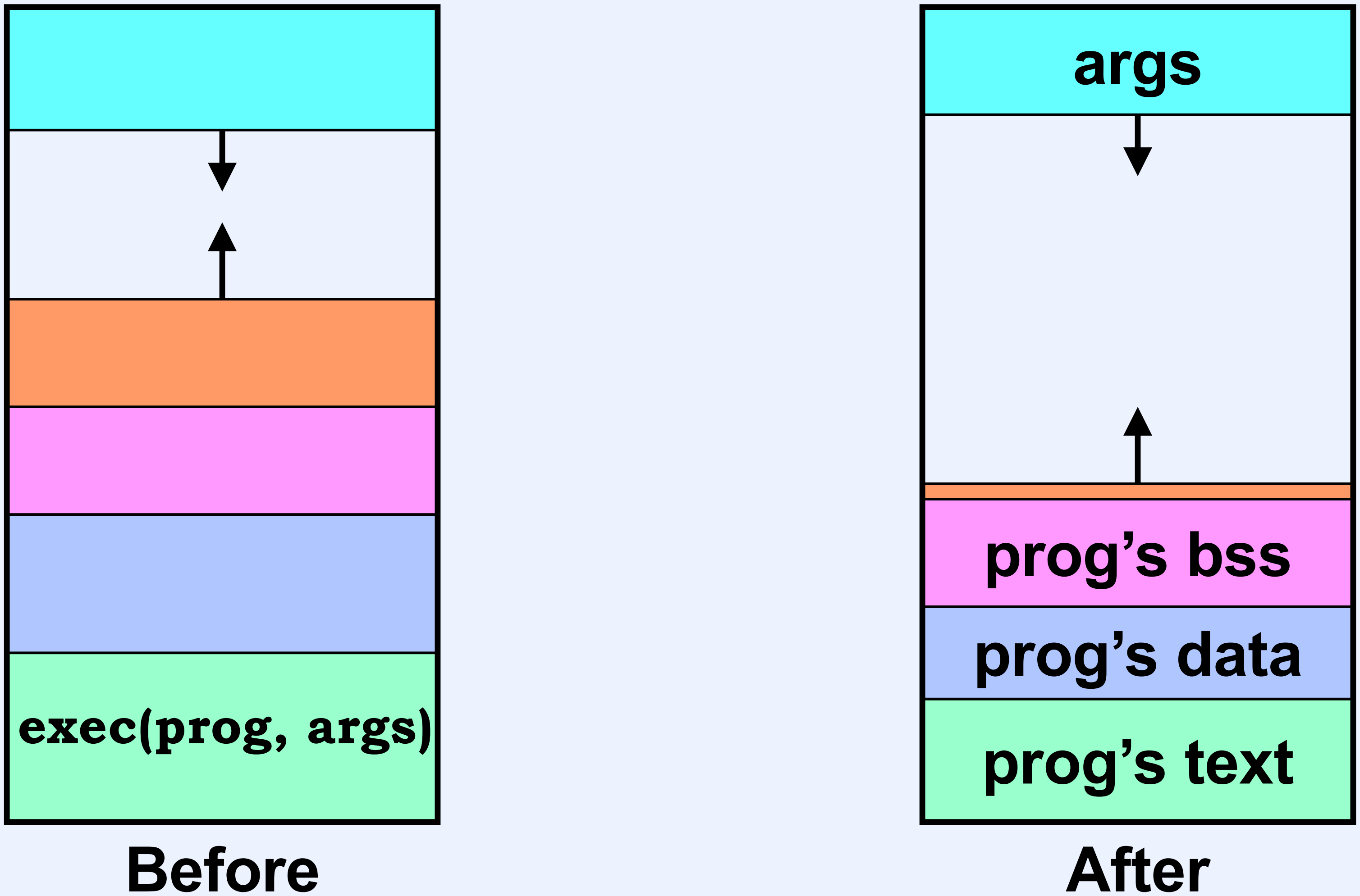
Exec

```
int pid;
if ((pid = fork()) == 0) {
   /* we’ll soon discuss what might take place before exec
      is called */
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))      /* ignore the return code */
   ;
```

Loading a New Image



Quiz 2

```
int A=0, B=0, C=0, D=0;
A=1;
if (fork() > 0) {
  B=1;
  A=111;
} else {
  C=2;
  if (fork() > 0) {
    D=222;
  } else {
    D=A+B+C;
    // what value is now
    // in D for this process?
  }
}
exit(0);
```

Answer:

a) 0
b) 3
c) 113
d) indeterminate