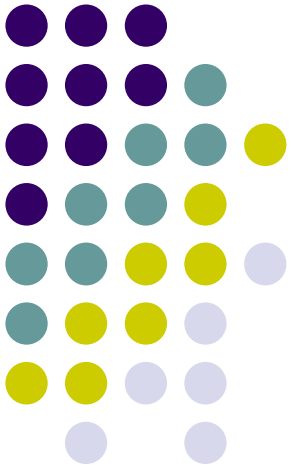
File Structures

Ch03. B Secondary Storage and System Software



2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

Slides are based on the textbook.

datalab

data science laboratory

References

- **Operating Systems: Three Easy Pieces**
  - **http://pages.cs.wisc.edu/~remzi/OSTEP/**
    - **Ch37. Hard Disk Drives**
    - **Ch38. Redundant Disk Arrays (RAID)**
    - **Ch44. Flash-based SSDs**

- **Database Implementation**

Contents

Why Study Disks?

- If files were stored just in memory
  - Data structures are enough
  - No need for file structures
- Secondary Storage
  - Very different from memory
    - Take much more time
    - Not all accesses are equal
- Good file structure design
  - uses knowledge of disk performance to arrange data in ways that minimize access costs

Disks

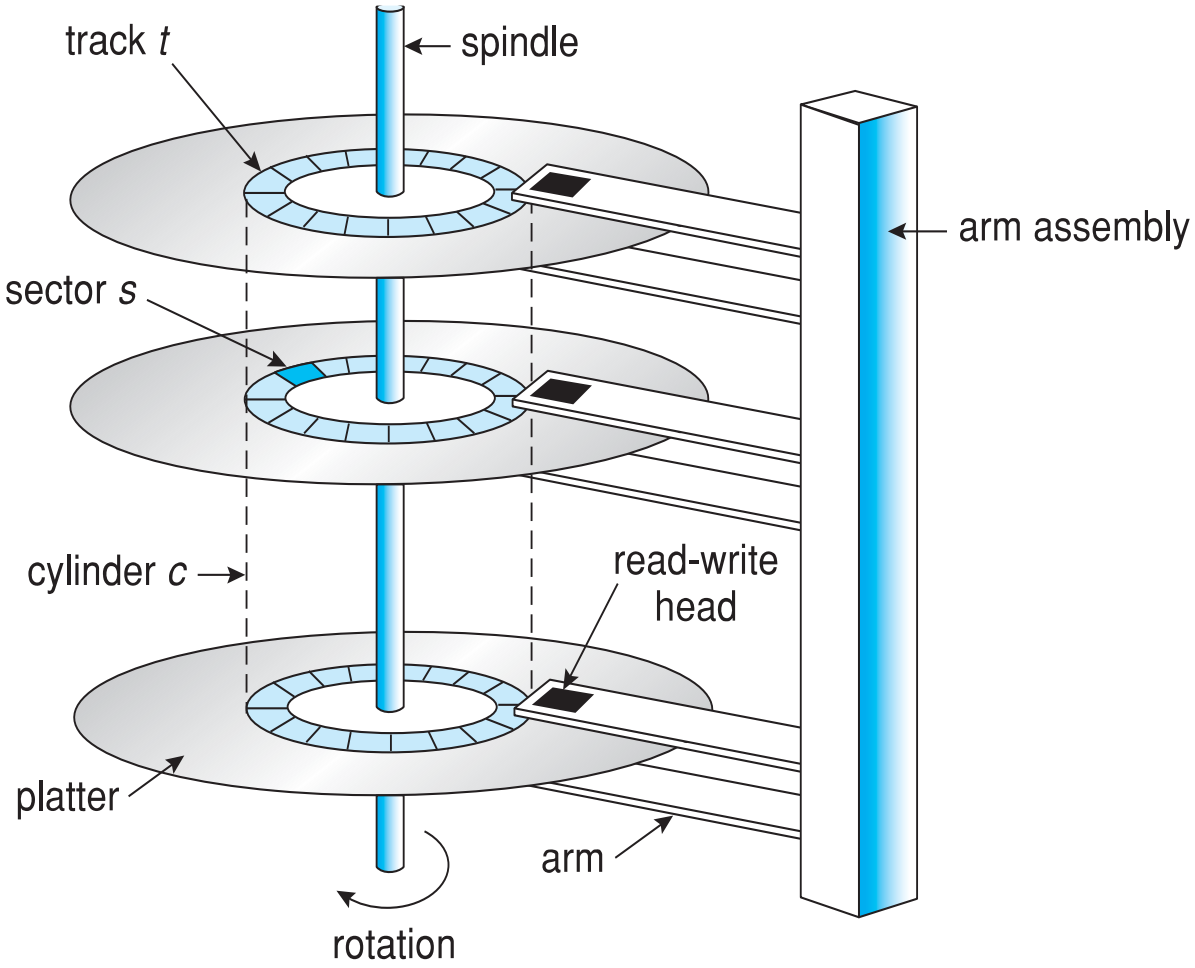
- Random Access Device
- Disk device - DASD(Direct Access Storage Devices)
- Magnetic disk
  - hard disk - most common
  - floppy disk - inexpensive, little capacity
  - magnetic tape - serial access
  - Iomega Zip (100 M byte), Jaz (1 G byte)
- Nonmagnetic disk
  - optical discs - increasingly important for secondary storage
  - e.g. CD, DVD

Organization of Disks (1/2)

- Components of Disk
  - Tracks
  - Sectors
    - the smallest addressable portion of a disk
  - Disk pack
    - a collection of a lot of platters
  - Cylinder (vertical collection of tracks)
    - Arms are moving together
- Seeking
  - r/w arm movement
  - the slowest part of reading data from disk

Organization of Disks (2/2)

- Surface of disk showing tracks and sectors



Organizing Data on disk

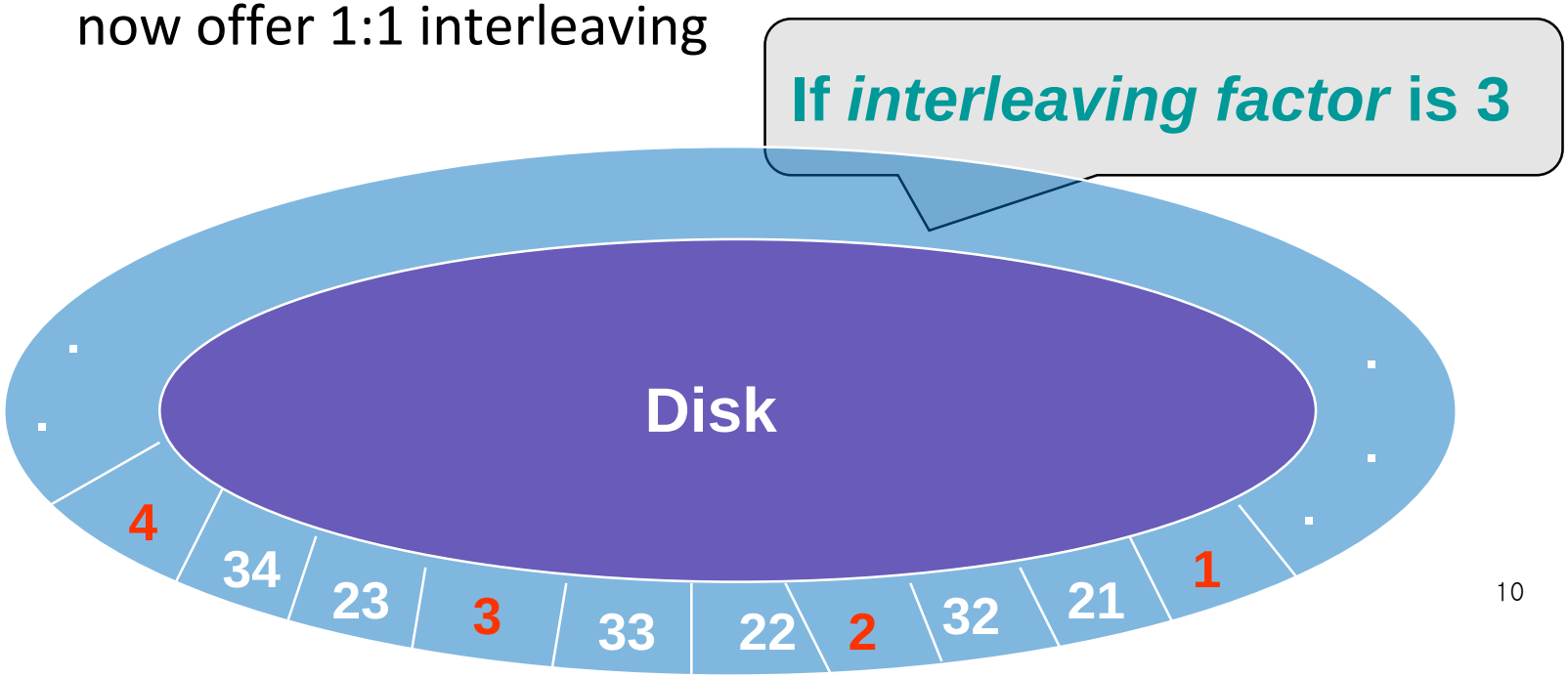
- Two ways to organize data on a disk
  - Organizing tracks by sector
    - (1) Simple View
    - (2) Cluster
    - (3) Extents
  - Organizing tracks by user-defined block

(1) Physical placement of sectors (1/2)

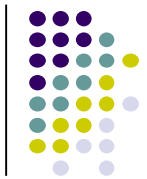
- Simple view
  - Sectors are adjacent, fixed-sized segments of a track that happen to hold a file
  - adequate way to view a file *logically*, but always not good way to store sectors *physically*
  - Interleaving technique can be used

(1) Physical placement of sectors (2/2)

- Simple view (continued)
  - interleaving
    - problems occur due to the gap between disk revolution speed and disk controller speed
    - high performance disks (with high speed disk controller) now offer 1:1 interleaving



(2) Clusters (1/4)



- file manager (part of an operating system)
  - Another view of sector organization
  - to map the logical parts of the file to their corresponding physical locations
  - viewing the file as a series of clusters of sectors

- Clusters (for improving file access speed)
  - a fixed number of contiguous sectors
  - can access all sectors in a cluster without requiring an additional seek
    - ***No*** additional seek (arm movement) for one cluster

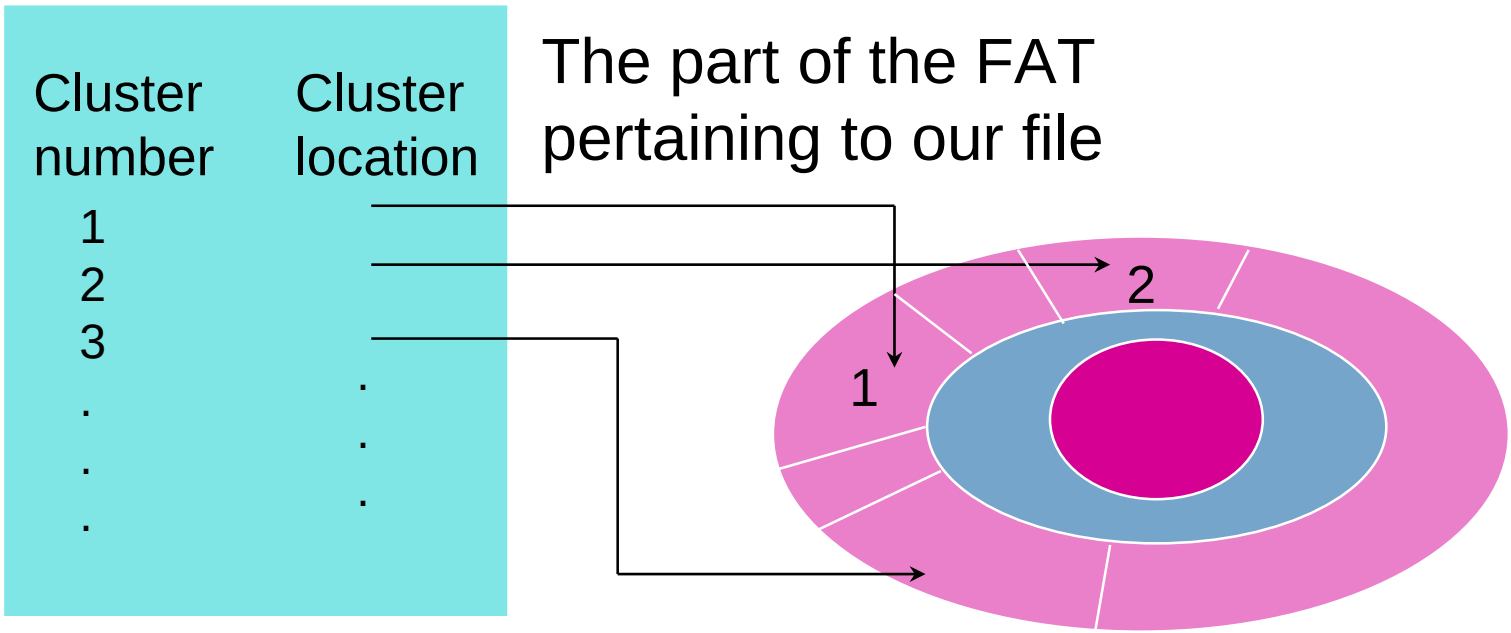
(2) Clusters (2/4)

- By using file allocation table(FAT)
  - the file manager ties logical sectors (like logical records) to the physical clusters
  - the FAT contains a list of all the clusters in a file
    - clusters are ordered according to the logical order of the sectors
    - each cluster entry in the FAT is an entry giving the physical location of the cluster
  - 1977 FAT for use on floppy disks by Bill Gates (DOS)

(2) Clusters (3/4)

- FAT A table containing mappings to the physical locations of all the clusters in all files on disk storage

File allocation table(FAT)

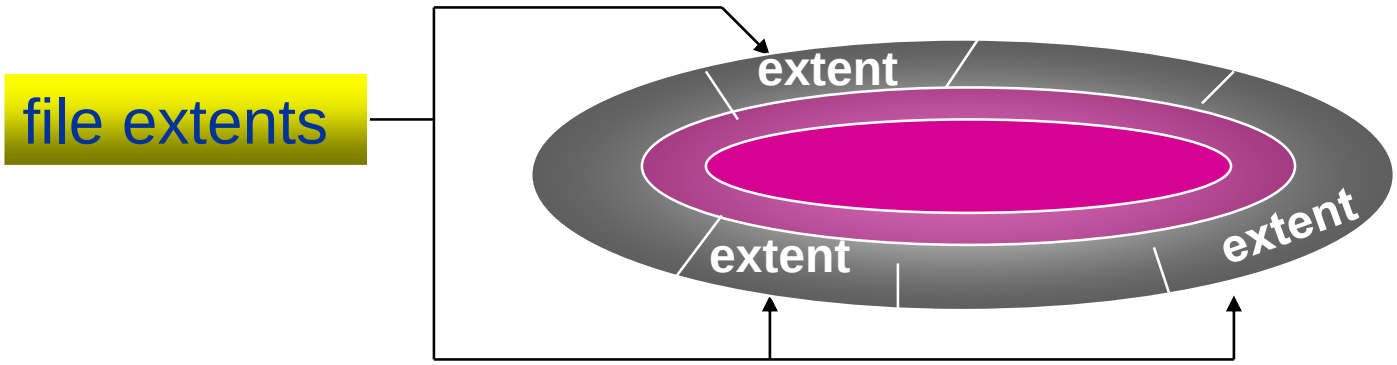


(2) Clusters (4/4)

- Cluster size
  - set the cluster size to be used on a disk when the disk is initialized
  - the default size is three 512-byte sectors per cluster
  - larger clusters guarantee the ability to read more sectors without seeking (when a file is processed sequentially)

(3) Extents

- **Extents (further attempt for file access speed)**
  - Contiguous clusters per extent and Mapping table
  - File may have several extents
  - Very useful when accessing the whole file sequentially
  - Many extents of a file increase amount of seeking



Fragmentation

- **Fragmentation**
  - the space that goes unused within a cluster, block, track, or other unit of physical storage
  - internal fragmentation: loss of space within a sector or cluster
  - trade-offs in use of large cluster sizes

- Example: the size of a sector is 512 bytes and the size of all records is 300 bytes
  - store only one record per sector
    - internal fragmentation
    - the loss of space within a sector
  - allow records to span sectors
    - some records may be retrieved only by accessing two sectors

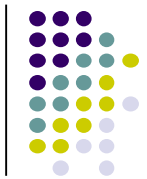
Organizing Data on disk

- Two ways to organize data on a disk
  - Organizing tracks by sector
    - (1) Simple View
    - (2) Cluster
    - (3) Extents
  - Organizing tracks by user-defined block

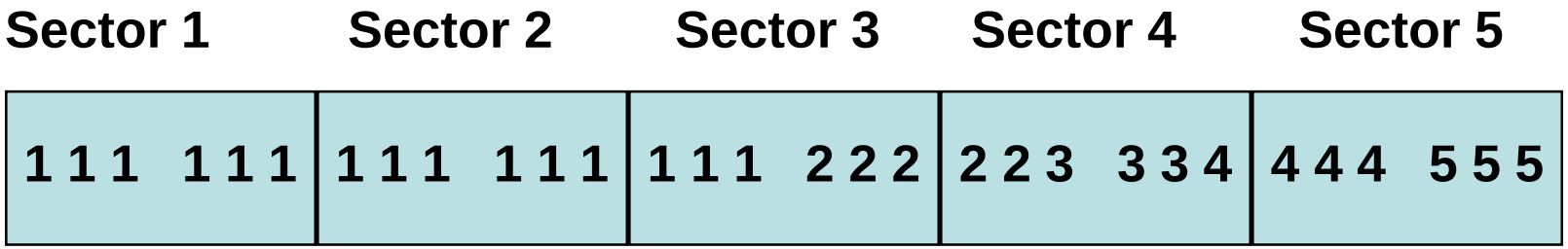
Organizing Tracks by Block (1/3)

- User-defined size of blocks can be fixed or variable
- *No* sector spanning ==> *No* internal fragmentation
- Blocking factor
  - the number of records that are to be stored in each block
- In block-addressing, each block of data is accompanied by one or more subblocks containing extra information about the data block
- Note: the "block" has a different meaning in the context of the UNIX I/O system

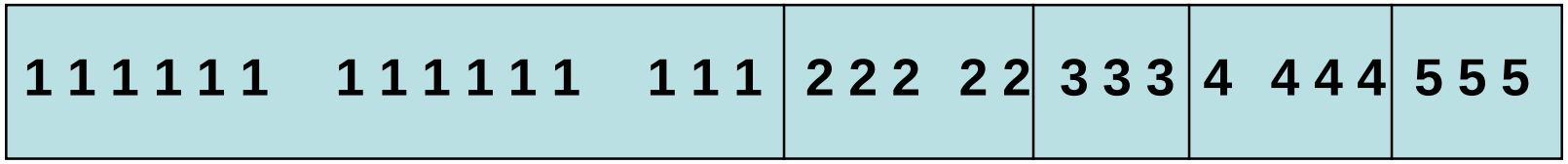
Organizing Tracks by Block (2/3)



- Sector organization vs. Block organization



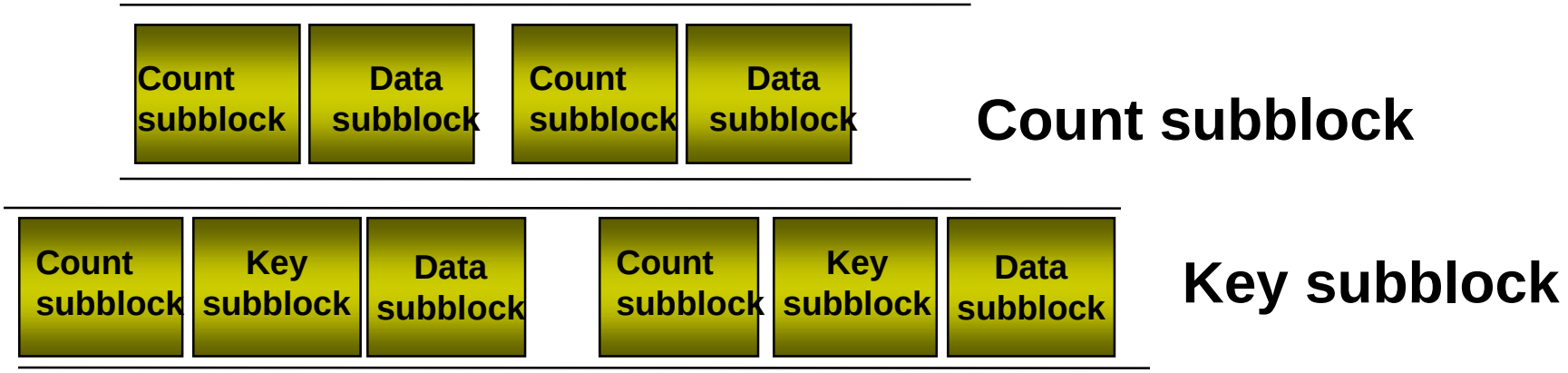
(a) sector organization



(b) block organization

Organizing Tracks by Block (3/3)

- Subblocks
  - count subblocks
    - contain the num of bytes in the accompanying data block
  - key subblocks
    - contain key for last record in data block



Nondata Overhead (1/3)

- Stored during preformatting
- Sector-addressable disks (physical)
  - sector/track address, condition, gaps, and synchronization marks
  - of no concern to programmer
- Block-organized disks (user-defined)
  - some of nondata for programmer
  - more nondata provided with blocks than with sectors

Nondata Overhead (2/3)

- Ex) block-addressable disks
  - 20,000 bytes/track
  - subblock, interblock gap equivalent to 300 bytes/block
  - If we want to store a file containing 100-byte records, how many records can be stored per track?
    - if blocking factor is 10, then 20000 / ((10*100)+300) = 15 blocks ==> 150 records
    - if blocking factor is 60, 180 records(=3 blocks)

Nondata Overhead (3/3)

- Block-organized scheme
  - larger blocking factor (many records in a block)
    - less amount of nondata block
    - more efficient use of storage
  - pros.
    - flexibility -> savings in time & efficiency
  - cons.
    - internal fragmentation problem
    - necessary for programmer and OS to do extra data organization
    - sector interleaving cannot be used to improve performance

Disk as Bottleneck

- Disk performance is increasing, but still slow!
- Even high-performance network is faster than disk

  (5 M/ sec HD is slower than 100M bps LAN)
- Disk bound jobs
  - CPU and network must wait
- Solution techniques
  - multiprogramming
  - striping - parallel I/Os
  - avoid accessing disk
    - RAM disk, disk caches - locality

Contents

- 3.1 Disks
- 3.8 A journey of a Byte
- 3.9 Buffer Management
- 3.10 I/O in Unix

Write()

- What happens when a program writes a byte to a file on a disk?

User's program

```
main(){
    ...
    write(textfile, ch, 1)
    ...
    ...
}
```

OS's file I/O subsystem

Get one byte from variable ch in user program's data area

Write it to current location in text file

User's data area:

ch: P

The File Manager

- the file manager
  - is a part of OS
  - deals with file-related matters and I/O devices
  - Can be thought of as several layers of procedures

- layers of procedures involved in transmitting a byte from a program's data area to a file
  - the upper layers deal with symbolic, or logical aspects of file management
  - the lower layers deal with the physical aspects

A journey of a byte (1/2)

Logical layer

1. The program asks the OS to write the contents of the variable *c* to the next available position in TEXT.

step1: write (textfile, ch, 1)

2. The OS passes the job on to the file manager

step2: OS to the FM

// File Manager //

3. The file manager looks up TEXT in a table containing information about it, such as whether the file is open and available for use, what types of access are allowed, if any, and what physical file the logical name TEXT corresponds to.

step3: look up textfile in a file descriptor table

4. The file manager searches a FAT for the physical location of the sector that is to contain the byte.

step4: search a FAT for the physical location of the sector

5. The file manager makes sure that the last sector in the file has been stored in a system I/O buffer in RAM, then deposits the 'P' into its proper position in the buffer.

step5: I/O buffering in RAM

6. The file manager gives instructions to the I/O processor about where the byte is stored in RAM and where it needs to be sent on the disk.

step6: give instructions to the I/O processor

A journey of a byte (2/2)

// I/O Subsystem //

7. The I/O processor finds a time when the drive is available to receive the data and puts the data in proper format for the disk. It may also buffer the data to send it out in chunks of the proper size for the disk.

step7: I/O processor

8. The I/O processor sends the data to the disk controller.

step7: I/O processor to Disk Controller

// Disk Controller //

9. The controller instructs the drive to move the r/w head to the proper track, waits for the desired sector to come under the r/w head, then sends the byte to the drive to be deposited, bit-by-bit, on the surface of the disk.

step9: Disk controller

Physical layer

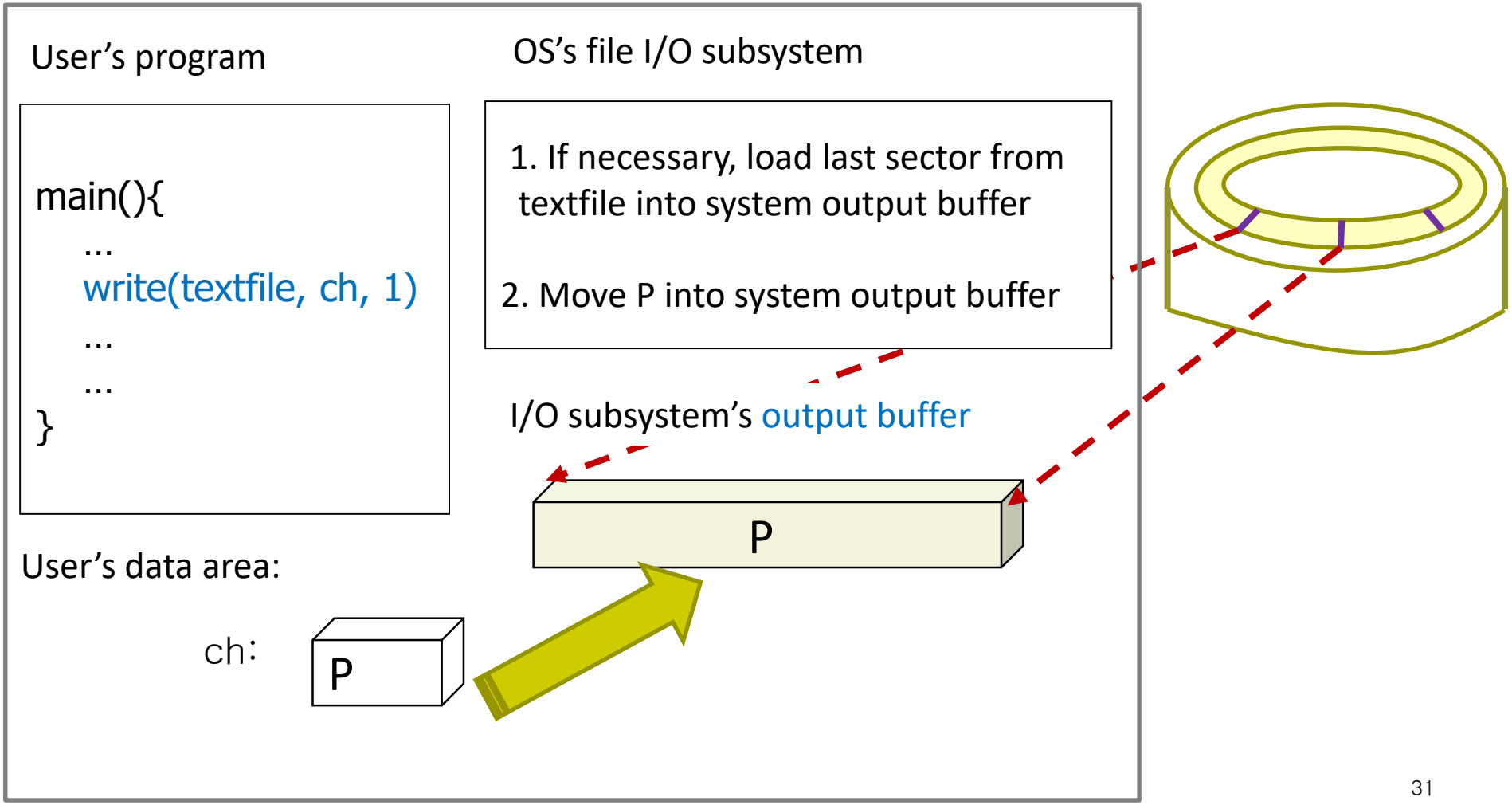
The I/O Buffer

5. The file manager makes sure that the last sector in the file has been stored in a system I/O buffer in RAM, then deposits the 'P' into its proper position in the buffer.

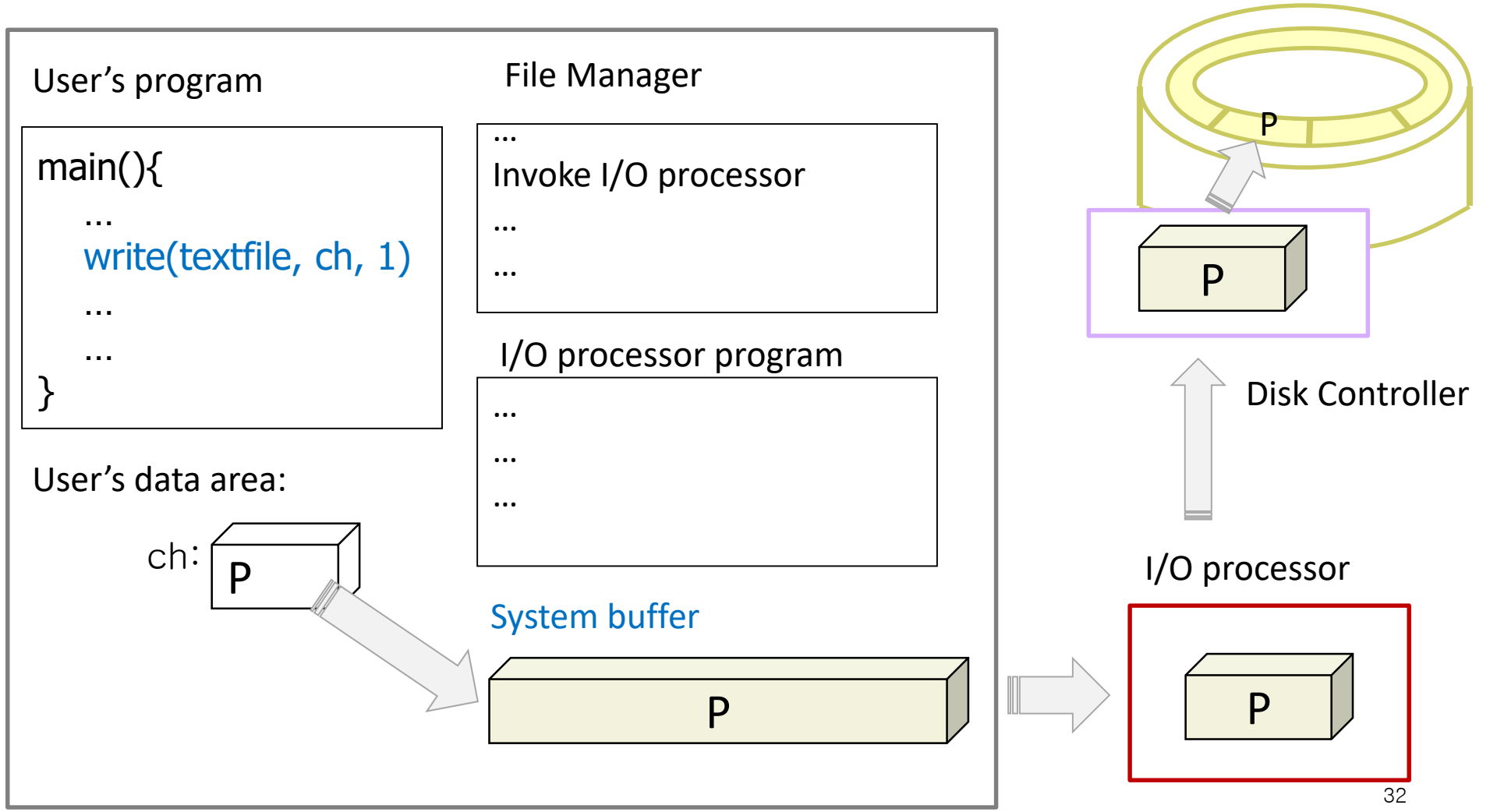
step5: I/O buffering in RAM

- At step 5
  - FM determines whether the sector to contain P is already in memory or needs to be loaded into memory
    - find an available system I/O buffer space
  - the system I/O buffer
    - allows the file manager to read and write data in sector-sized or block-sized units
  - FM usually waits until the buffer is flushed, instead of sending the sector immediately to the disk

FM and I/O buffer



A byte leaves memory



Contents

- 3.1 Disks
- 3.7 Storage as Hierarchy
- 3.8 A journey of a Byte
- 3.9 Buffer Management
- 3.10 I/O in Unix

What is buffer?

- Definition
  - the part of main memory available for storage of copies of disk blocks
- Program buffers vs. System I/O buffers

- Buffer manager
  - subsystem responsible for the allocation for blocks
  - goal:
    - minimize the number of disk access
    - utilize the memory space effectively

Buffer Management

- Buffer bottlenecks
  - What if program performs both input and output on one character at a time, and only one I/O buffer is available?
  - At least two buffer - for input and output
- I/O bound jobs: wait for I/O completion

- Buffering Strategies for Performance
  - use more than one buffer
  - I/O system processes next block when CPU is processing current one

Buffering Strategies

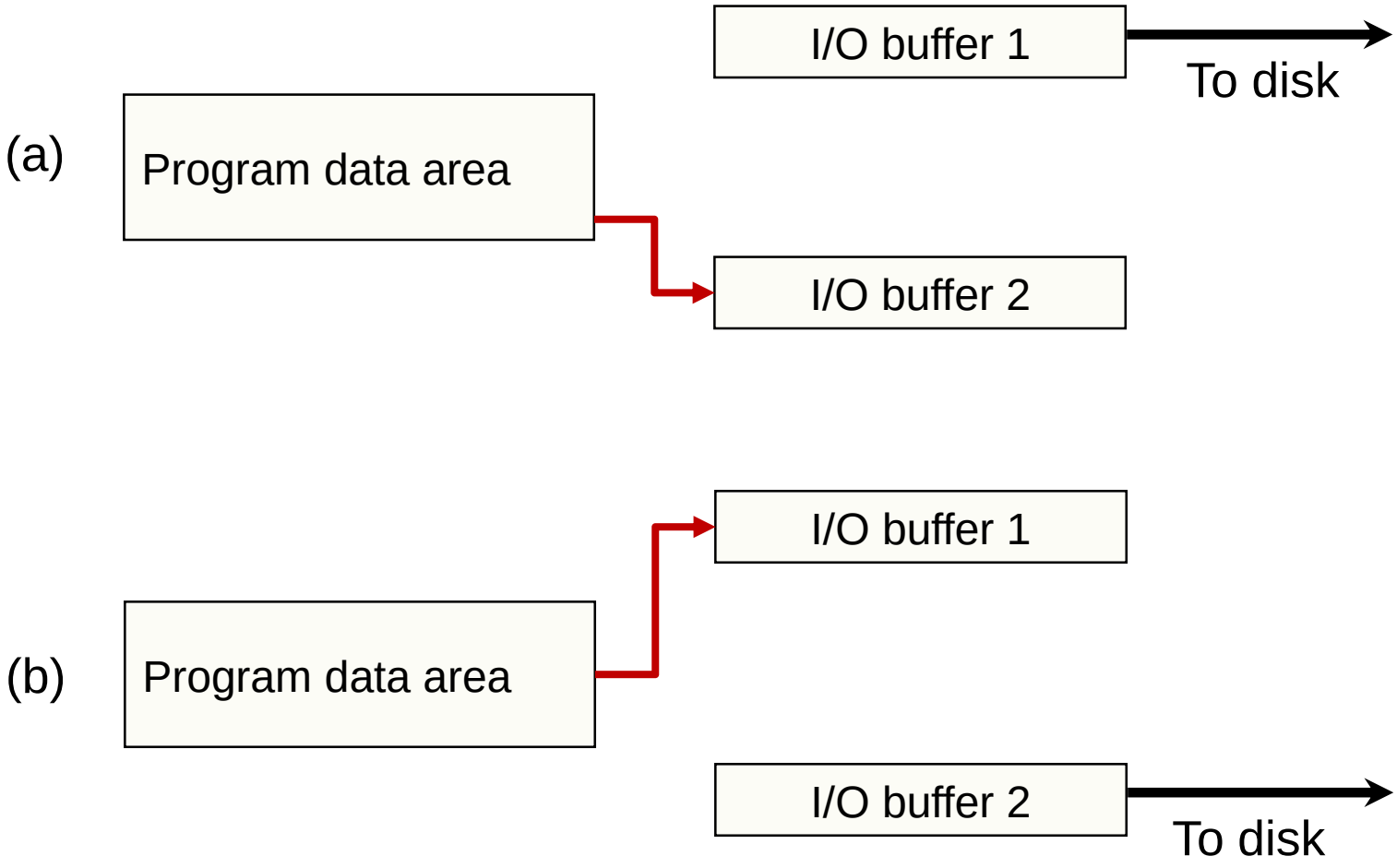
- Multiple buffering
- Move mode and locate mode
- Scatter/Gather I/O

Multiple buffering (1/2)

- Double buffering
  - swapping roles of two buffers after I/O finished
  - allows O/S operating on one buffer while the other buffer is being loaded or emptied
- any number of buffers can be used
- Buffer pooling
  - takes from a pool of available buffers
  - decides which buffer to take from a buffer pool
    - take buffer by LRU

Multiple buffering (2/2)

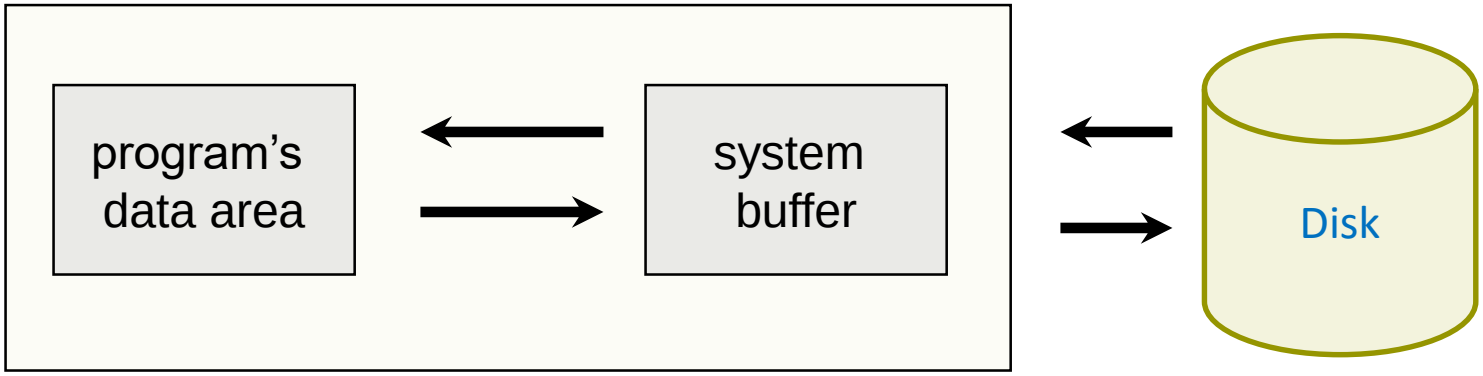
- Example of double buffering



Move & Locate mode (1/2)

- Move mode
  - using both system buffer & program buffer
  - moving data from one place in RAM to another before they can be accessed
  - sometimes, unnecessary data moves

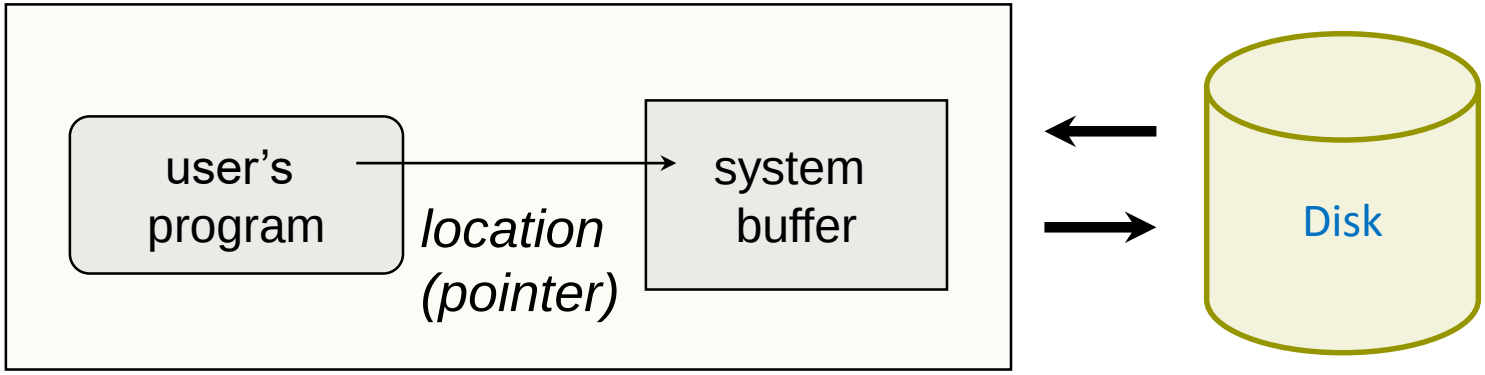
Move mode



Move & Locate mode (2/2)

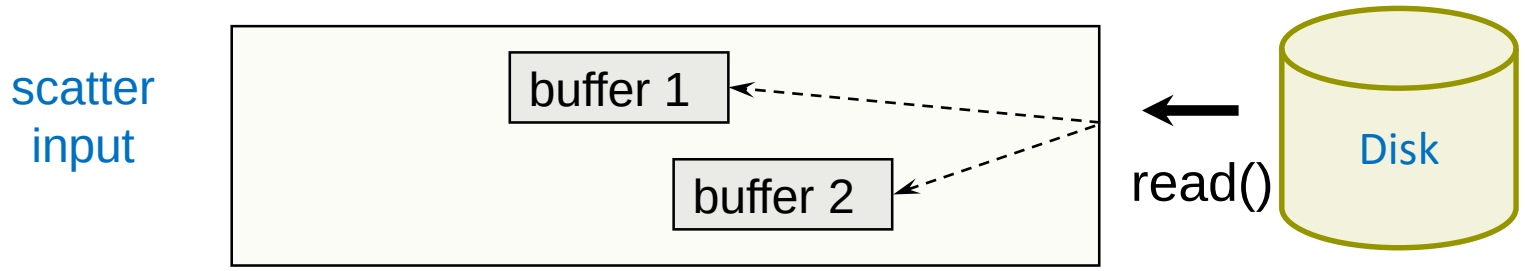
- Locate mode
  - using system buffer only or program buffer only
  - perform I/O directly between secondary storage and program buffer (program's data area)
  - system buffers handle all I/Os, but program uses locations through pointer variable

Locate mode

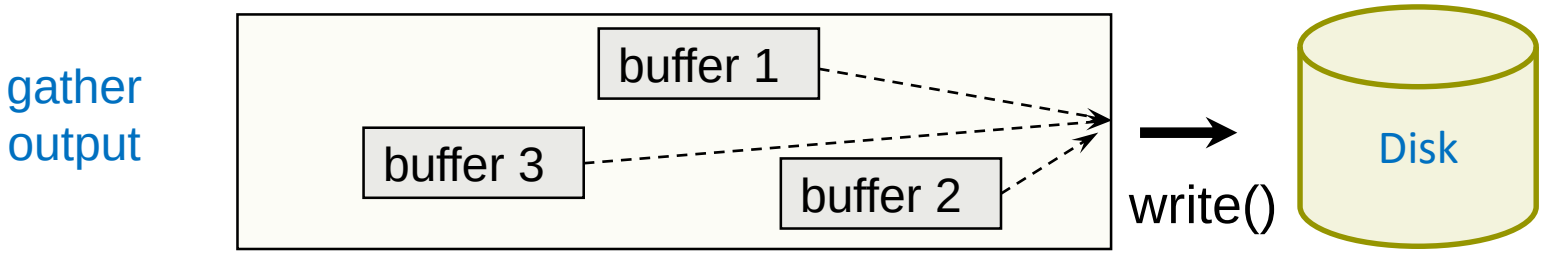


Scatter/Gather IO

- A file with many blocks and each block with header and data
- Need to put the headers in one buffer and the data in a different buffer ==> may occur complication
- Scatter-input mode
  - a single **READ** can scatter data into a collection of buffers



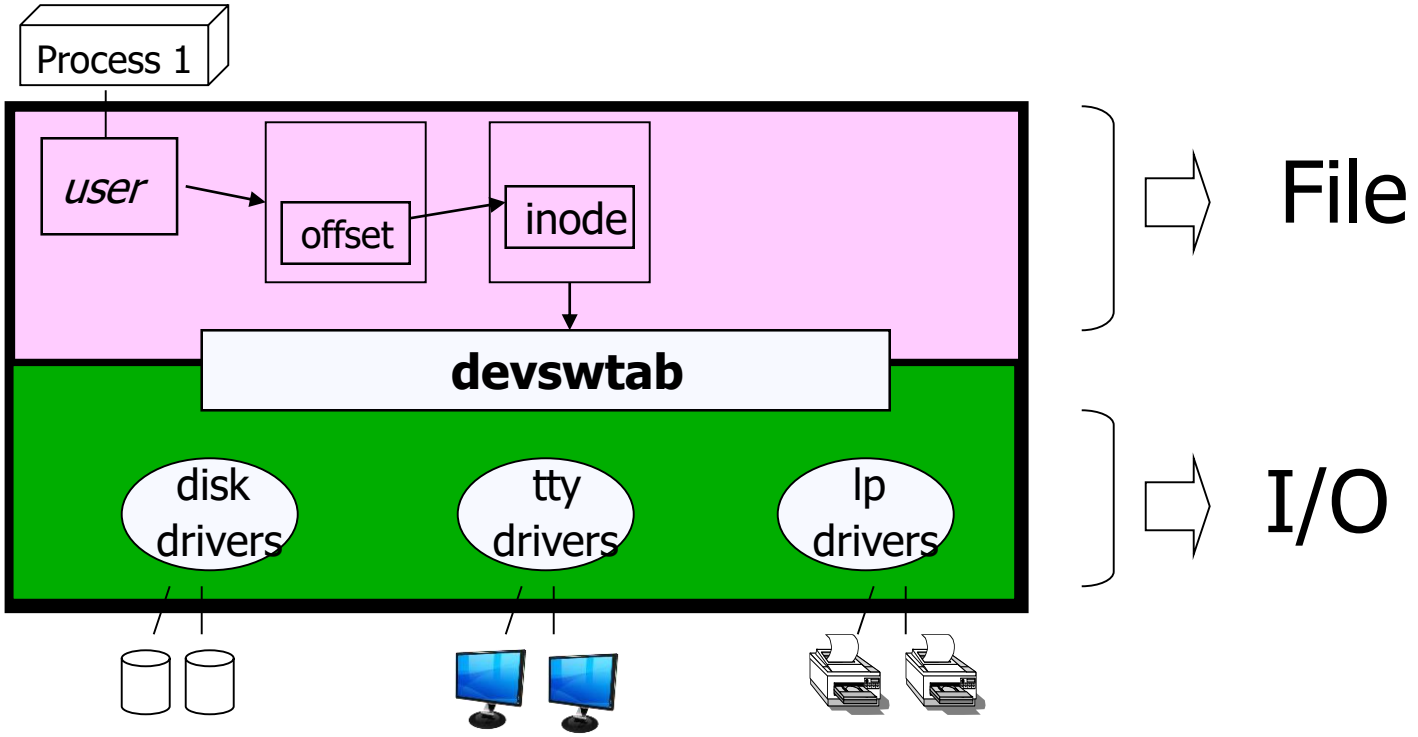
- Gather-output mode
  - a single **WRITE** can gather several buffers and output

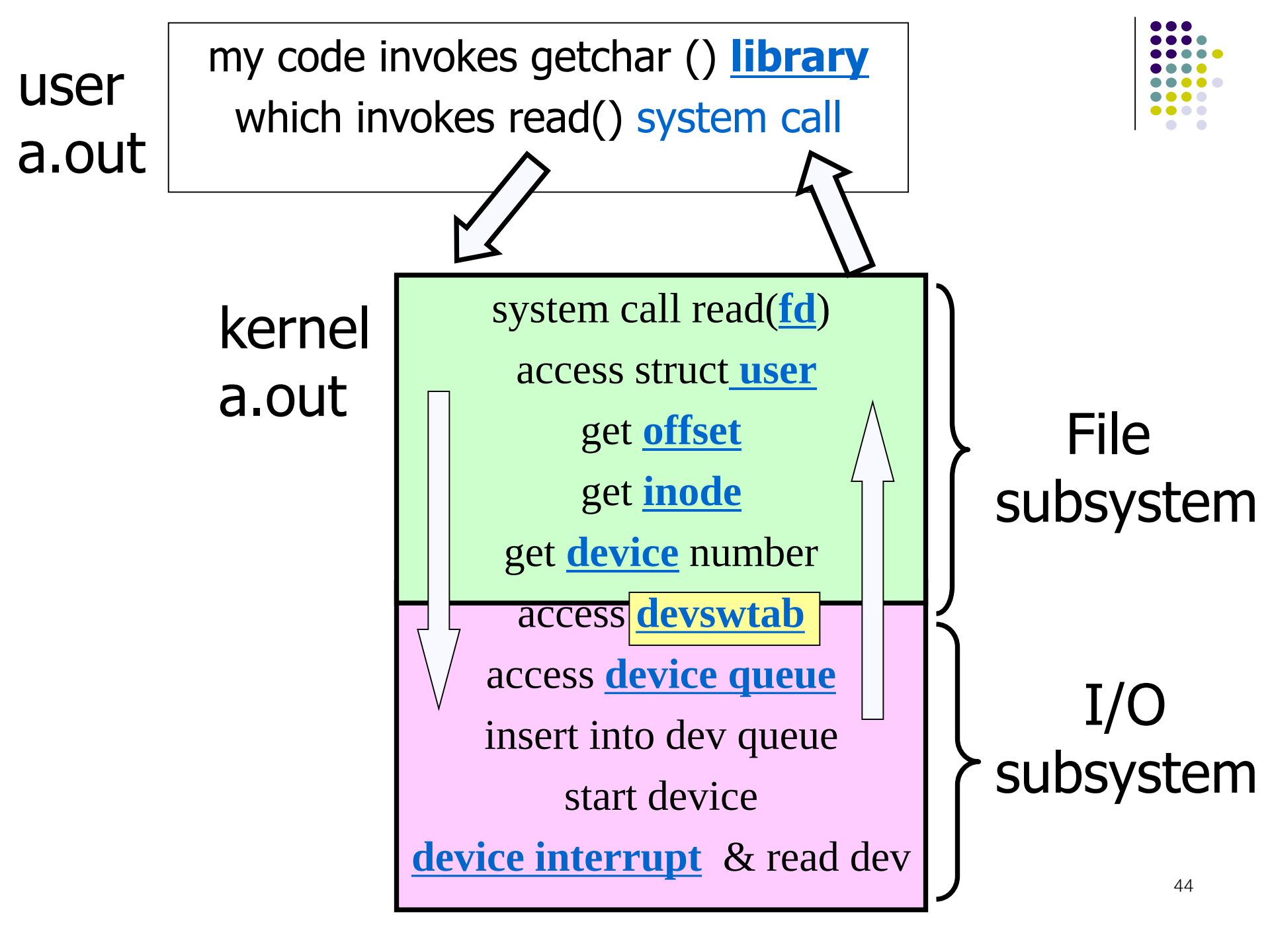


Contents

- 3.1 Disks
- 3.7 Storage as Hierarchy
- 3.8 A journey of a Byte
- 3.9 Buffer Management
- 3.10 I/O in Unix

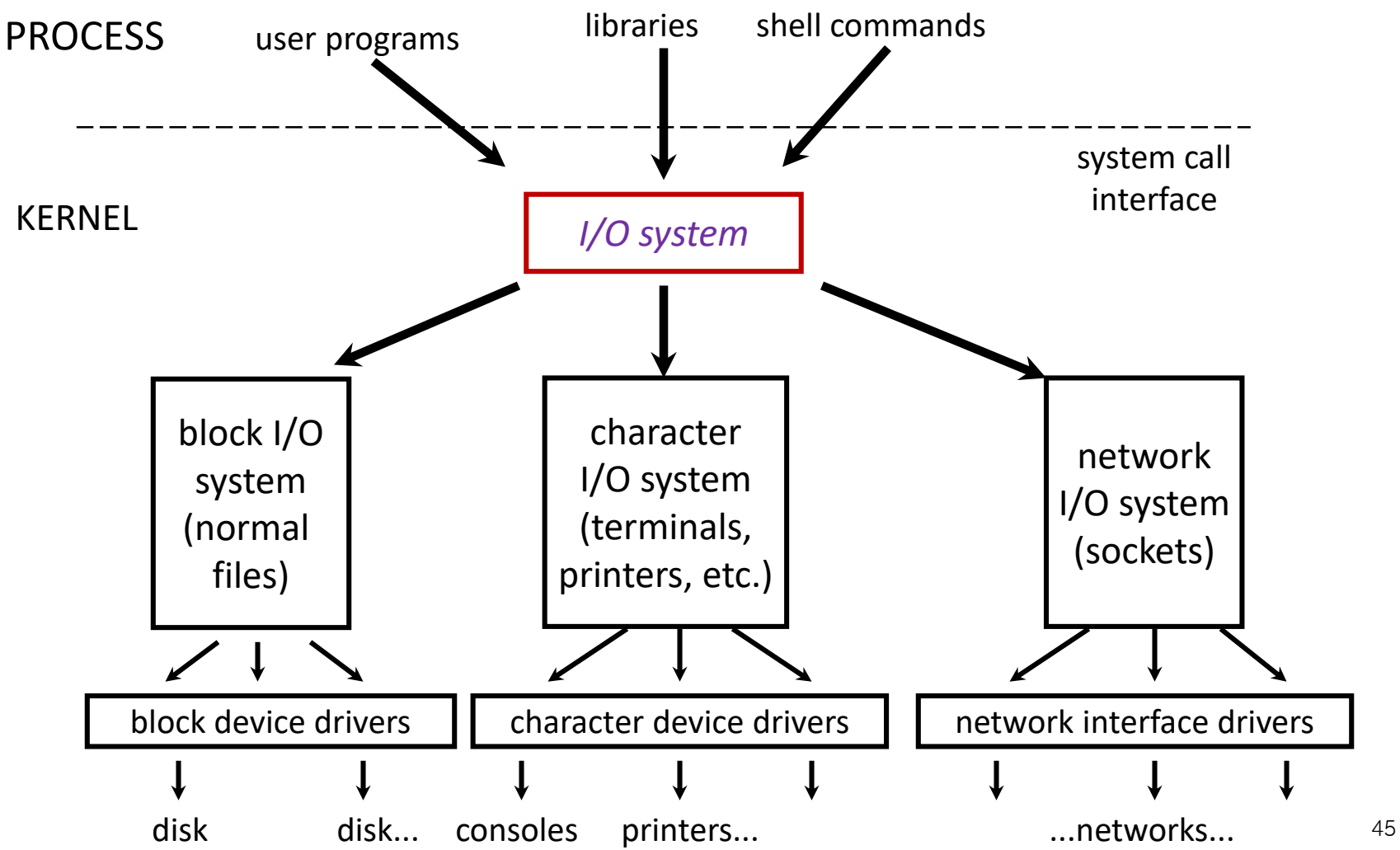
Detour: File vs I/O




user
a.out
my code invokes getchar () library
which invokes read() system call
kernel
a.out
system call read(fd)
access struct user
get offset
get inode
get device number
access devswtab
access device queue
insert into dev queue
start device
device interrupt & read dev
File
subsystem
I/O
subsystem

I/O in UNIX

<Kernel I/O structure>

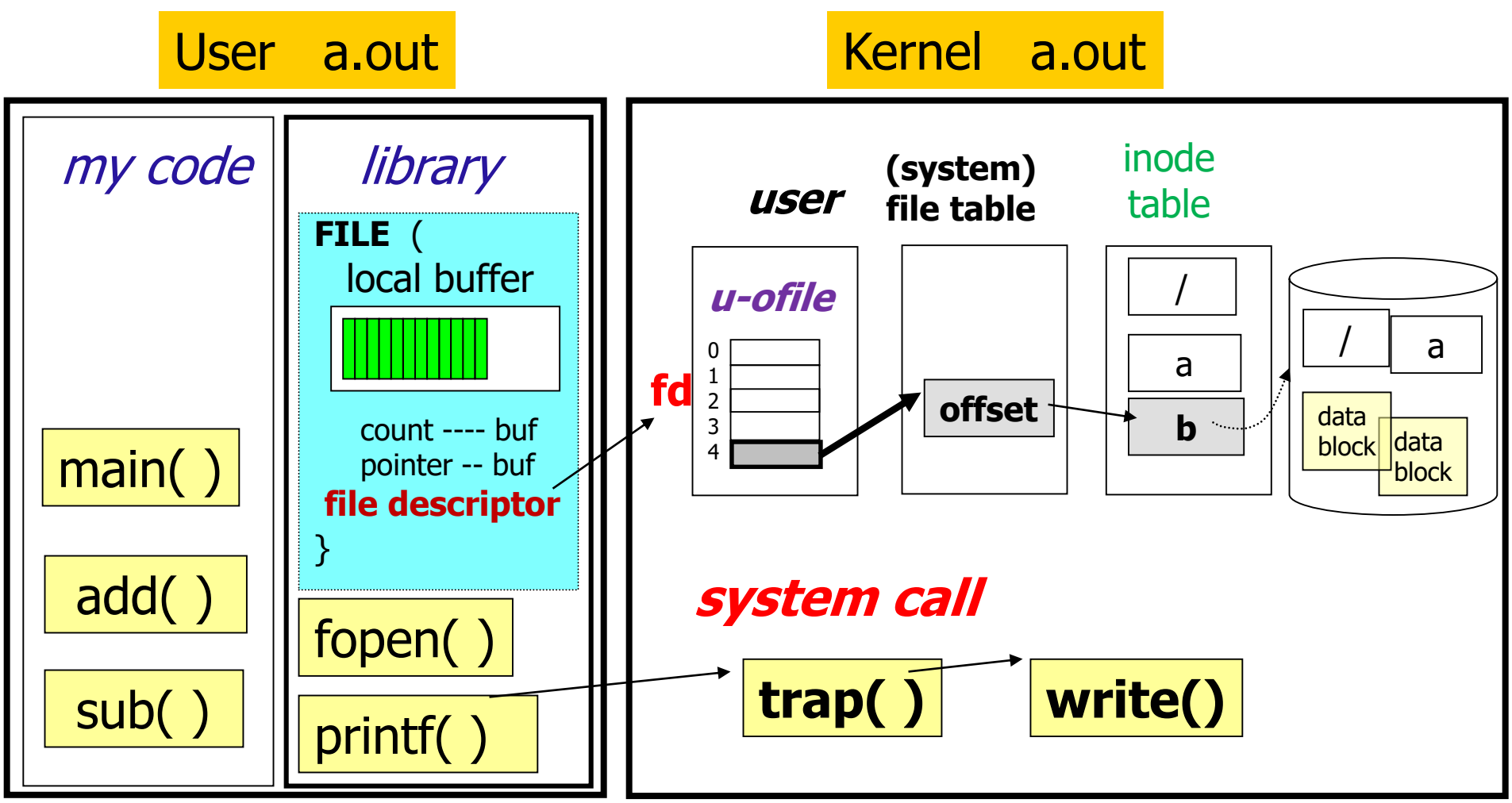


UNIX Kernel

- The central part of UNIX operating system
- View all I/O as operating on a sequence of bytes
- Make all operations below top layer independent of application's logical view of file

- Data structures related to unix files
  - file-descriptor table: owned by user process
  - open-file table: owned by kernel
  - index-node: a kind of FAT (one inode for each file in use)
  - index-node table: owned by kernel

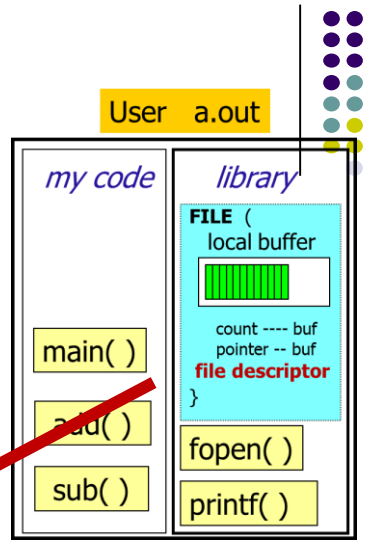
Detour: FILE vs fd



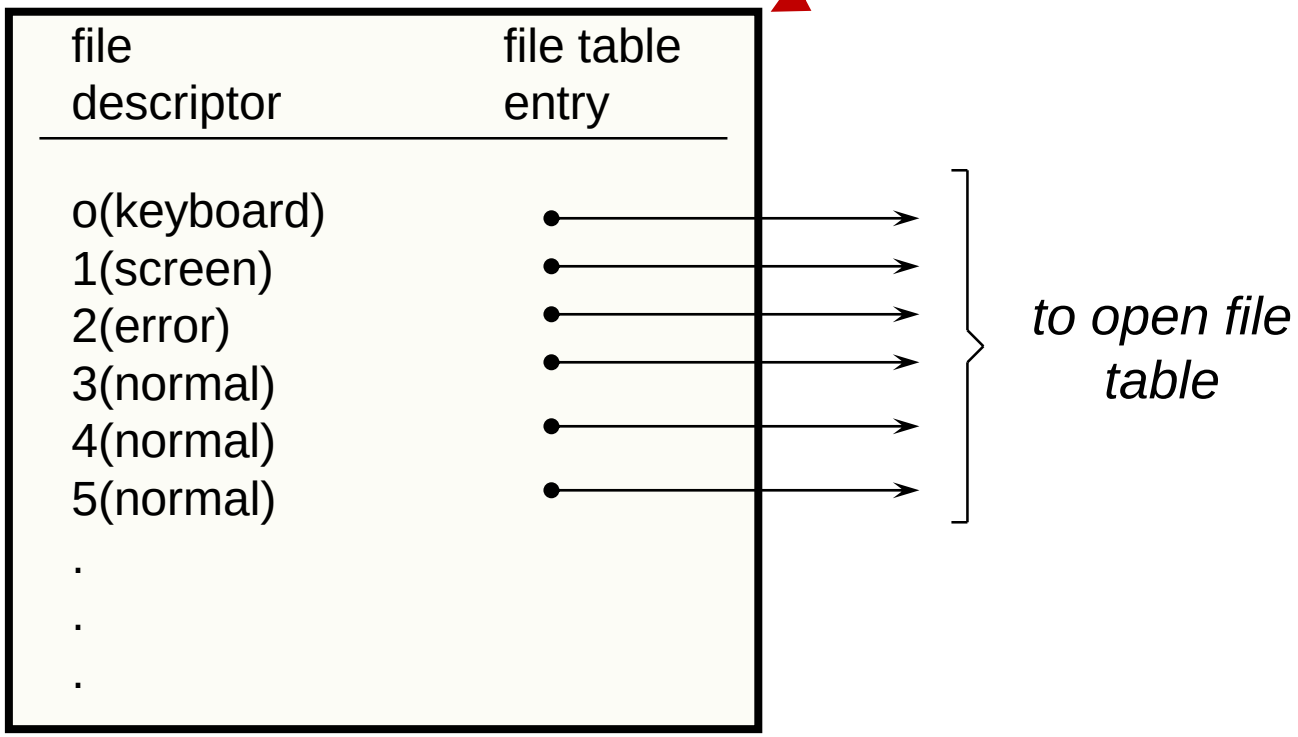
When the local buffer (in FILE) becomes empty, Read() system call fills this buffer again

File descriptor table

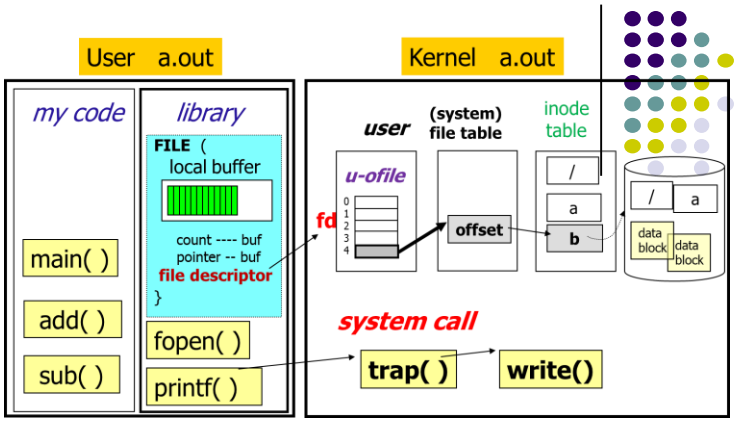
- File descriptor table
  - associates each file descriptor to open file table
  - every process has its own file descriptor table



fd table



Open file table



- Open file table
  - entries for every open file
  - file structures: r/w mode, offset, reference count

open file table

R/W mode	# of processes using it	Offset of next access	ptr to write routine	.....	inode table entry
write	1	100		.....	

to *inode* table

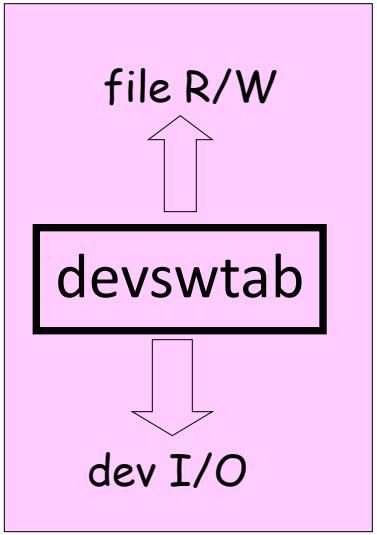
write() routine for this type of file

Inode Structure

- Inode (Index node)
  - data structure used to describe a file
  - when a file is opened, a copy of inode is loaded into RAM for rapid access
  - has a list of disk blocks of the file
  - this list is UNIX counterpart to FAT

Device driver

- I/O processor program invoked by kernel performing I/O for devices
- Size of device drivers
  - very big (majority) -- so many devices
  - still growing fast
  - strength of Linux -- supports many dev



- devswtab[dev, op] -- (2-dimensional array)
  - offers hardware independence
  - Two type of device drivers
    - block dev sw table dwrite, reada (asynchronously)
    - char dev sw table synchronous read/write
      eg tape-to-disk copy, tty, ...

Caution: terminology

- "block dev" does not mean "transfer in units of block"
- "char dev" does not mean "transfer in units of char"

- "char" means "not use buf cache"
  - hence "no dwrite, no reada", no asynchronous r/w

- "block" means "use buf cache"
  - hence "dwrite, reada", and asynchronous r/w

- Same disk can be operated either as block dev of char dev
  - eg file system partition -- block device
    swap partition -------- char device

Q&A