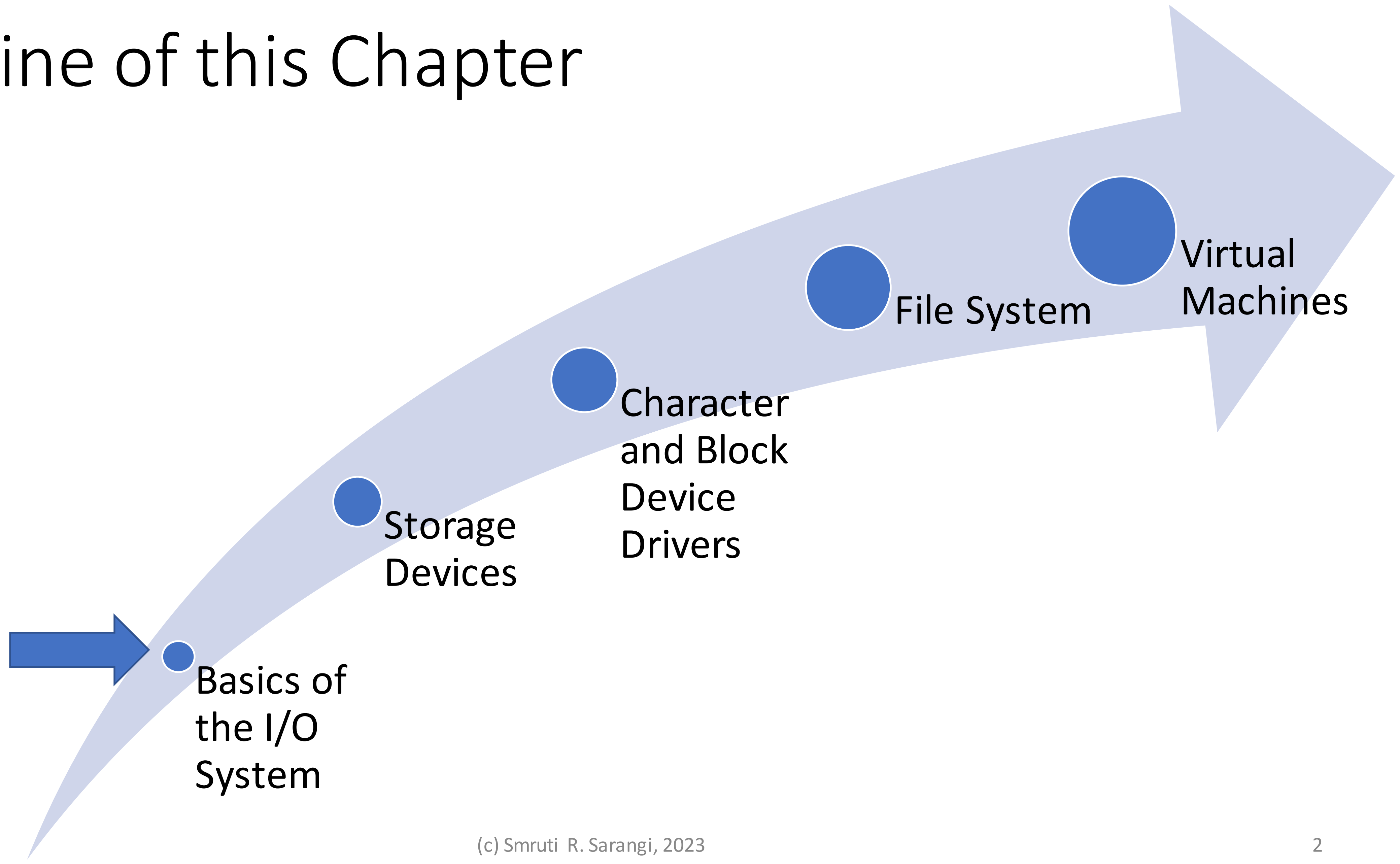
Chapter 7: I/O System, Device Drivers and Virtualization

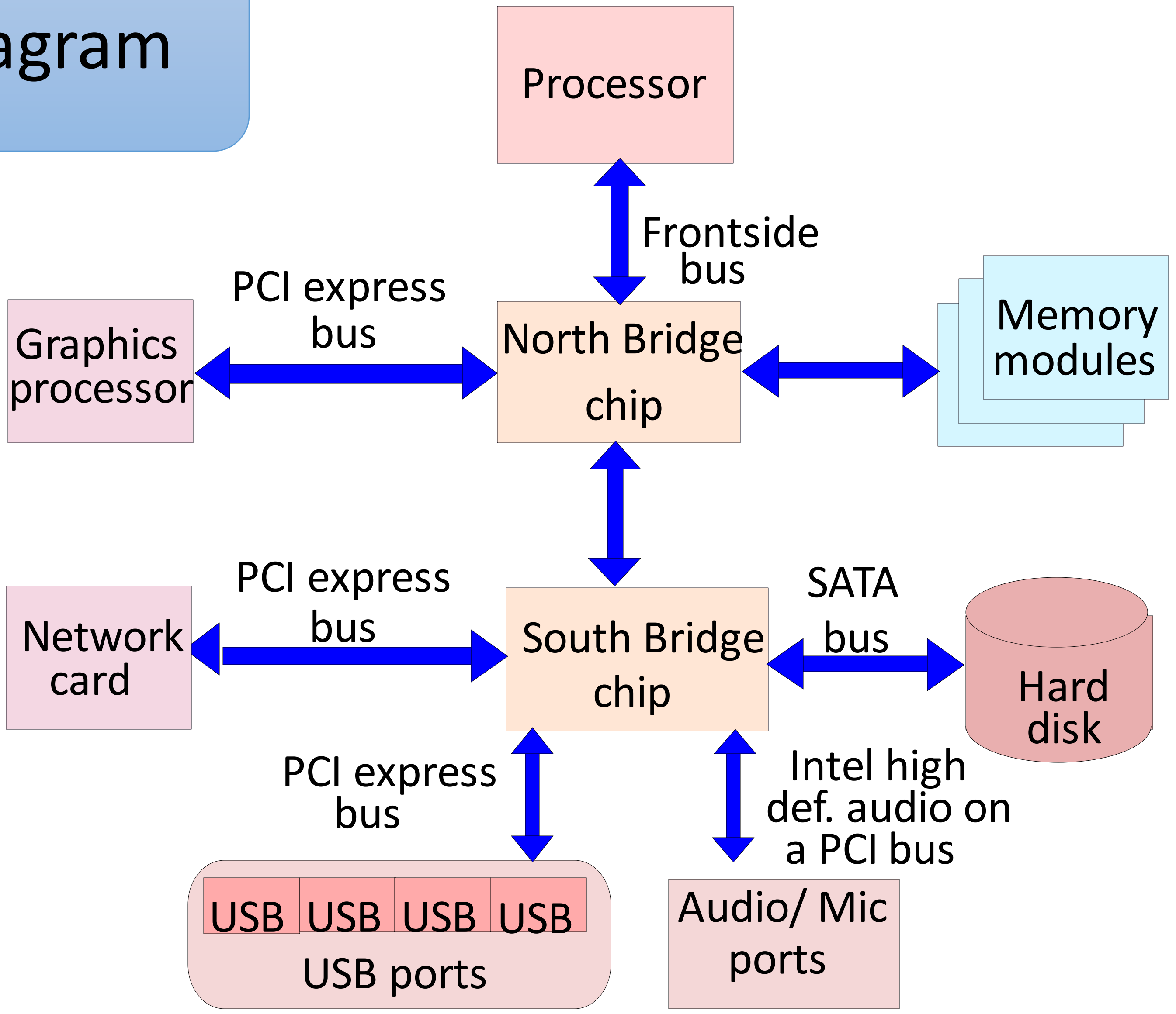
Smruti R Sarangi

IIT Delhi

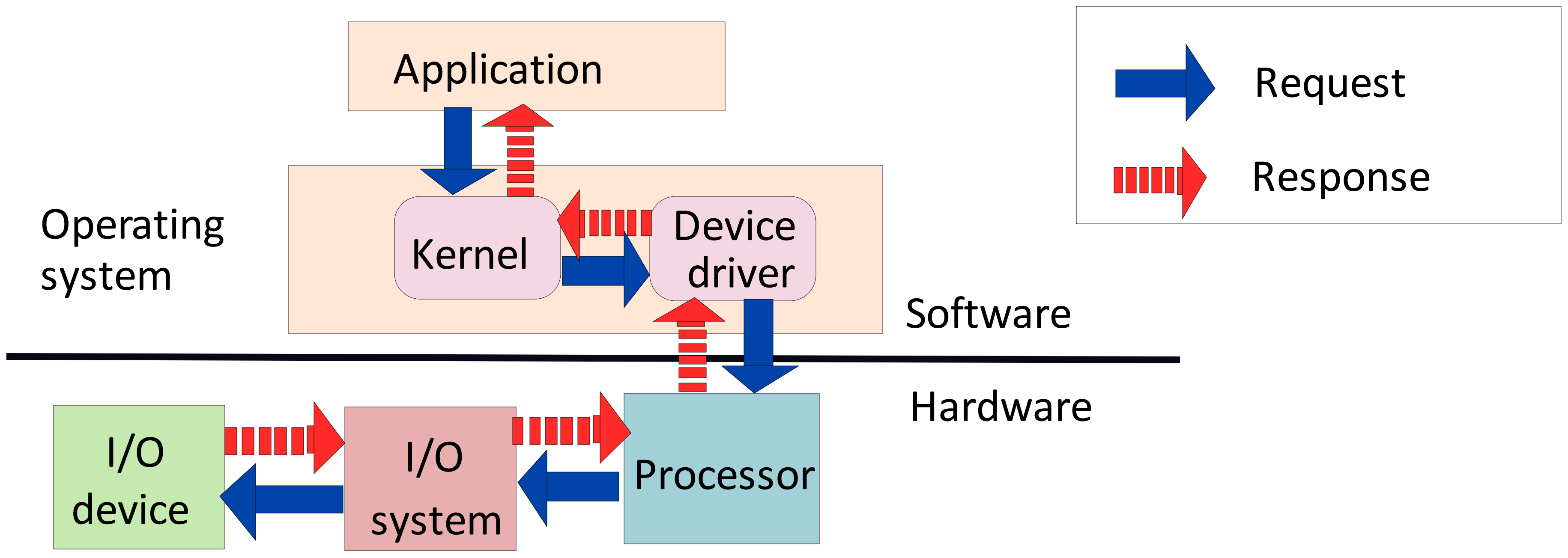
Outline of this Chapter



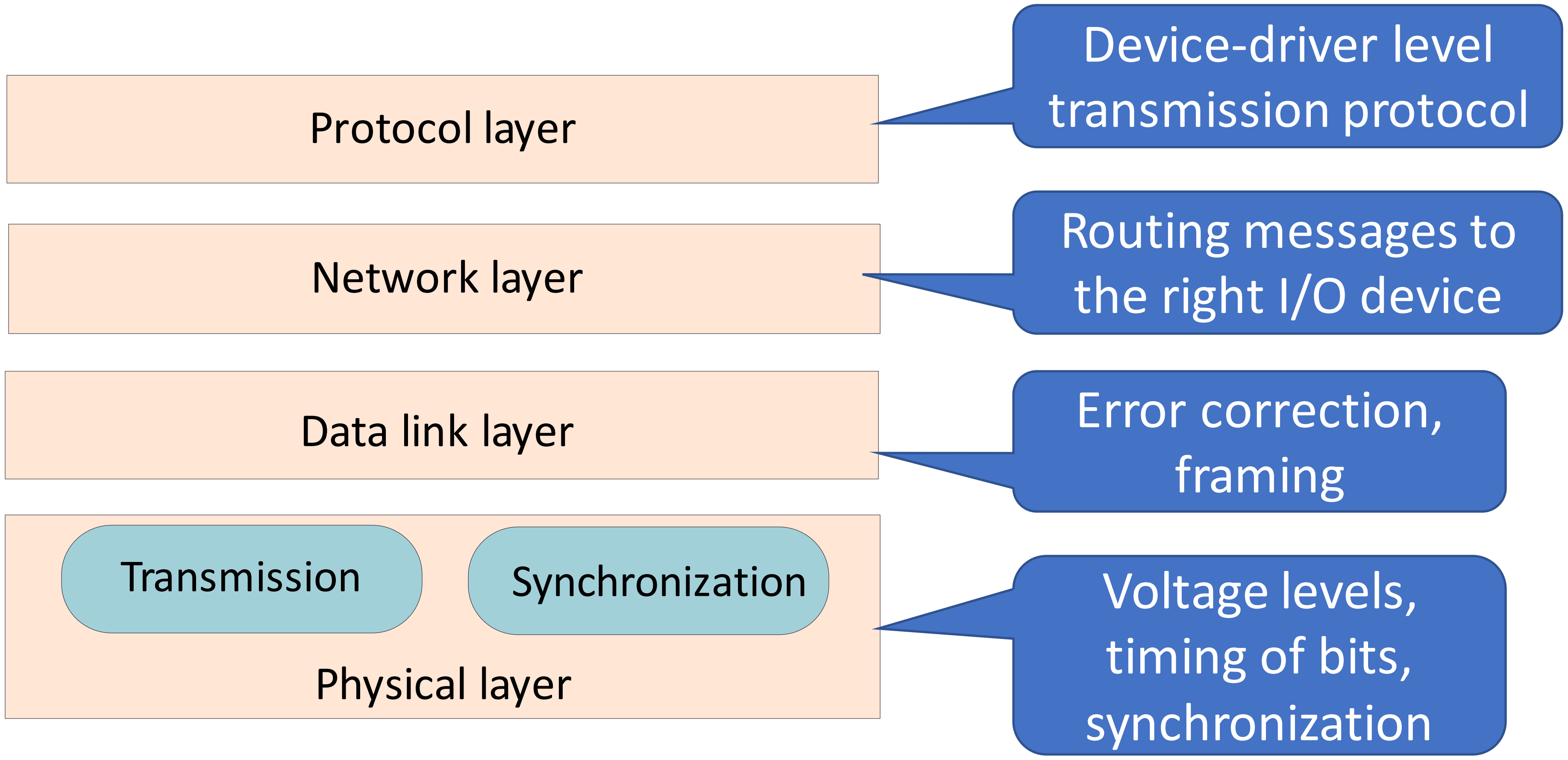
System Diagram



Software Flow



Layers in the I/O Protocol Stack

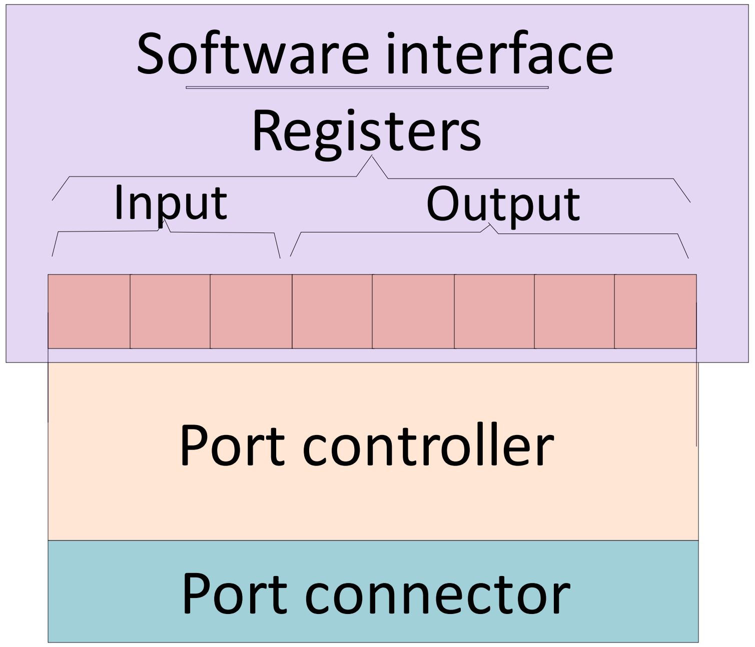

Protocol layer
Network layer
Data link layer
Transmission
Synchronization
Physical layer
Device-driver level transmission protocol
Routing messages to the right I/O device
Error correction, framing
Voltage levels, timing of bits, synchronization

Explanation of the Layers

- Physical Layer
  - Transmission Sublayer → Defines the electrical specifications of the bus, and the methods of encoding data
  - Synchronization Sublayer → The timing of signals
- Data link layer
  - Framing, buffering, error correction, transactions
- Network Layer
  - I/O device addressing, location, routing
- Protocol Layer
  - End-to-end request processing
  - Interrupts, and polling (continue to read an I/O register until its value changes)
  - DMA (Direct Memory Access)

I/O Ports

- Every device on the motherboard exposes a set of I/O ports.
  - An I/O port in this case, is a HW/SW entity → a set of registers
  - Each software I/O port is a wrapper on the actual hardware I/O port
  - A hardware I/O port is a set of connectors used to connect to an external device.



Software Interface

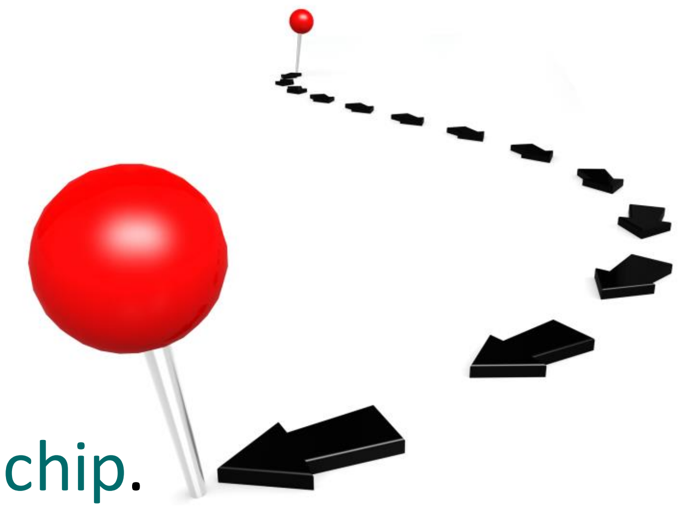
- Each I/O port exposes a set of 8-32 bit registers to software
- Software writes to the registers. The port controller automatically sends the information to the I/O device.
- Similarly, to read data, the processor reads the registers associated with the port controller.
- Example : Intel processors define 64K, 8-bit I/O ports. Each port has a 16-bit port number.
- Reading and writing to the actual device ↔ read and write to the associated software ports (special registers associated with I/O devices)

I/O Address Space

- Set of all I/O ports that are accessible to software
- x86 processors have two instructions to access I/O ports → *in* and *out*
- These are special (privileged) instructions

Instruction	Semantics
in r1,<i/o port>	r1 ← contents of <i/o port>
out r1, <i/o port>	contents of <i/o port> ← r1

I/O Mapped I/O



- A request contains an I/O port address
- The processor sends it to the Northbridge chip.
- The Northbridge chip forwards it to the Southbridge chip (if necessary)
- The Southbridge chip forwards it to the destination. (there might be several more hops)
- Each chip maintains a small routing table.
- The response follows the reverse path.

Memory Mapped I/O

- Problems with I/O mapped I/O
  - The programmer needs to be aware of the addresses of the I/O ports.
  - The same device might have different port addresses across different motherboards
  - Programs will cease to work on other machines.
  - It is hard to transfer a block of data (need to send it instruction by instruction)
  - **Solution** : memory mapped I/O

Memory Mapped I/O – II

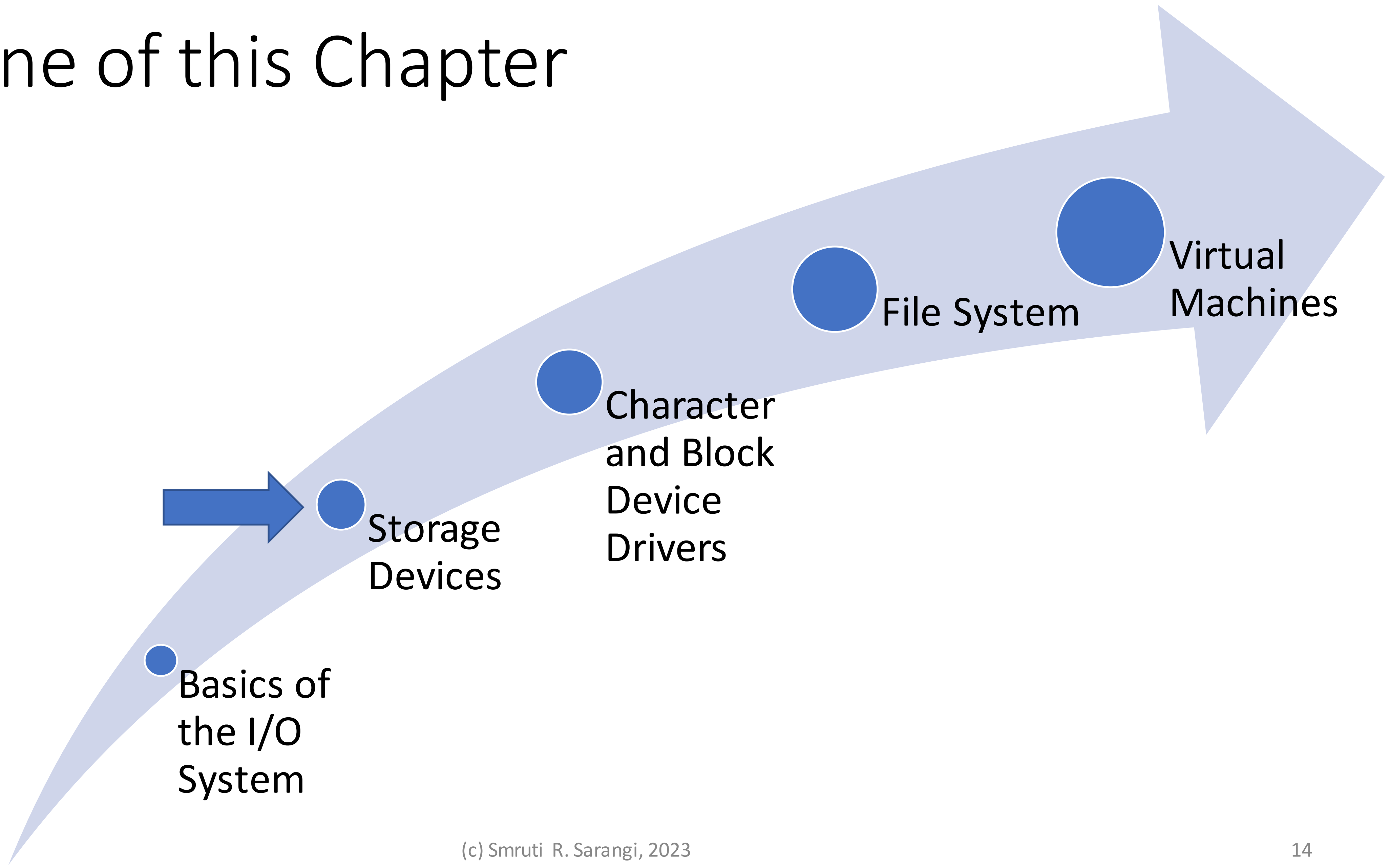
- Define a virtual layer between I/O ports, and the application.
- The operating system can use the paging mechanism to map I/O ports to memory addresses.
- Whenever, we write to a memory address that is mapped to an I/O port, the TLB directs the request to the I/O system.
- Similarly, for reading data, the response comes from the I/O System

The physical address in this case comprises a diversity of devices

Advantages

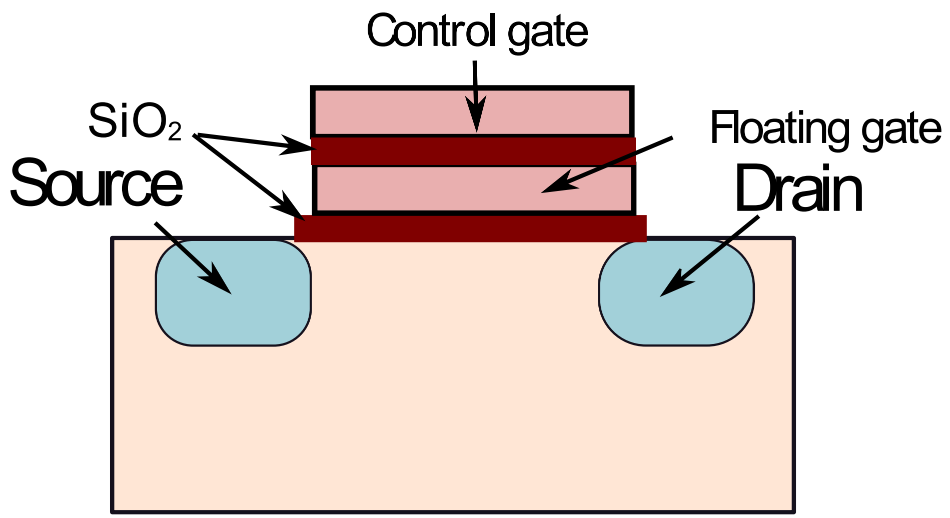
- The operating system creates the mapping between the I/O ports and the memory addresses
- The same I/O program runs on multiple machines
- Reading and writing to I/O devices requires normal load/store instructions
- Easy to read and write a large block of data using block load/store operations available in some architectures (e.g, x86)

Outline of this Chapter

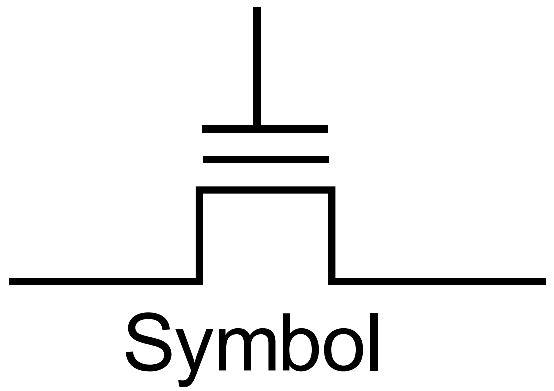


Flash Drives

Floating Gate Transistors



(a)



(b)

- Introduce an additional gate → floating gate
- We can trap electrons in it
- Given that it is separated by SiO_2 layers, the trapped charge can remain there for a long time.

Two states: (0) Charge trapped (1) No charge trapped

Impact on the Threshold Voltage

Threshold voltage when no charge is trapped ➡ V_T 1

Threshold voltage when charge is trapped ➡ V_T^+ 0

Given that electrons are negatively charged, for creating a conducting channel, we need to have:

$$V_T < V_T^+$$

Just set the threshold voltage to a value between V_T and V_T^+
See if the transistor is conducting or not.

Writing to a Floating Gate Transistor

Writing a 0 (programming)

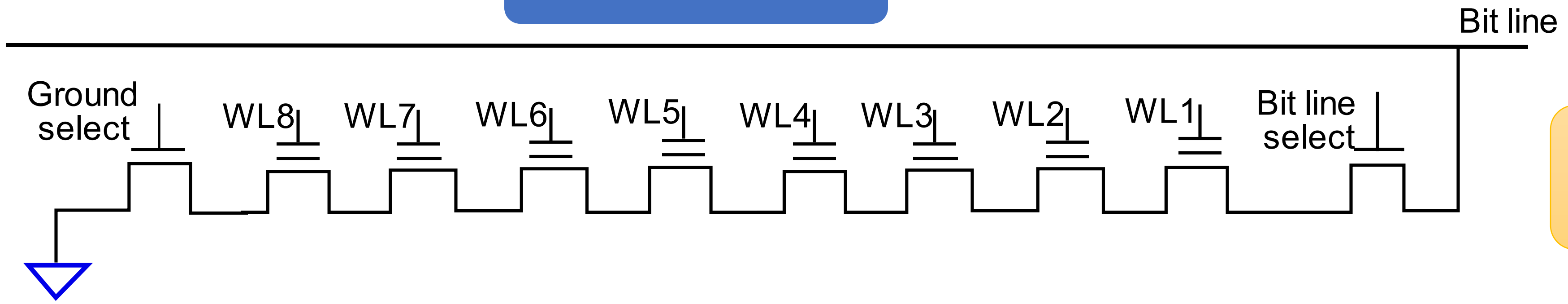
- Apply a strong positive potential at the gate
- The resulting electric field pulls electrons into the floating gate region

Writing a 1 (erasing)

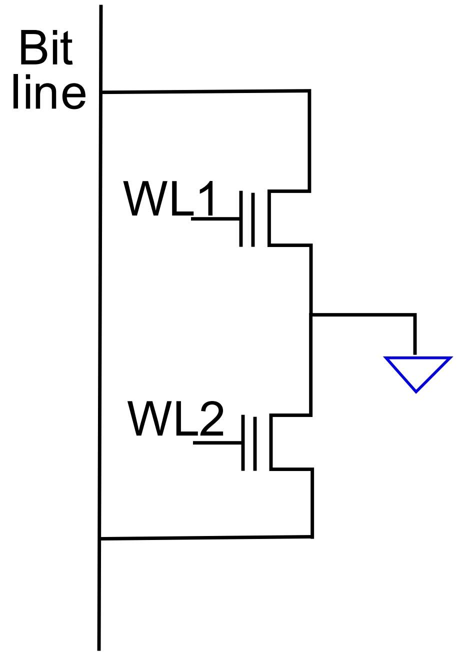
- Apply a strong negative potential at the gate
- Pushes the electrons back into the channel (back to the default state)

Two Configurations of Flash Cells

NAND Flash

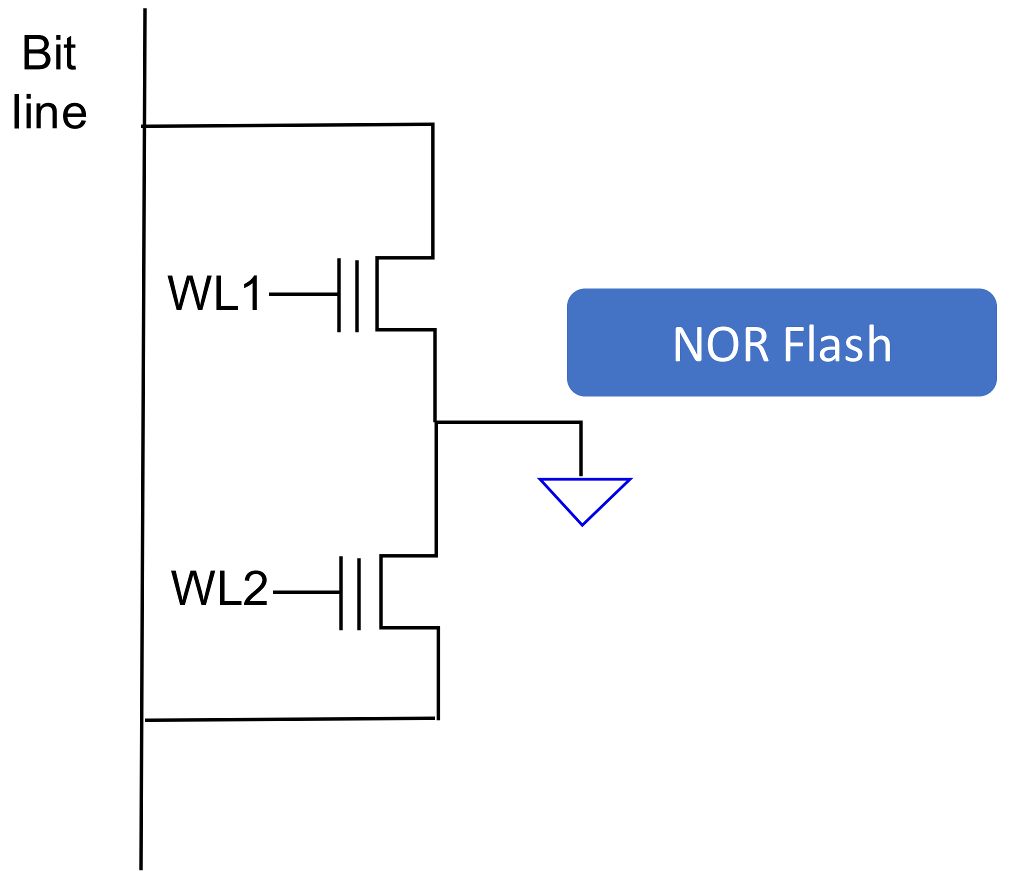


Higher density



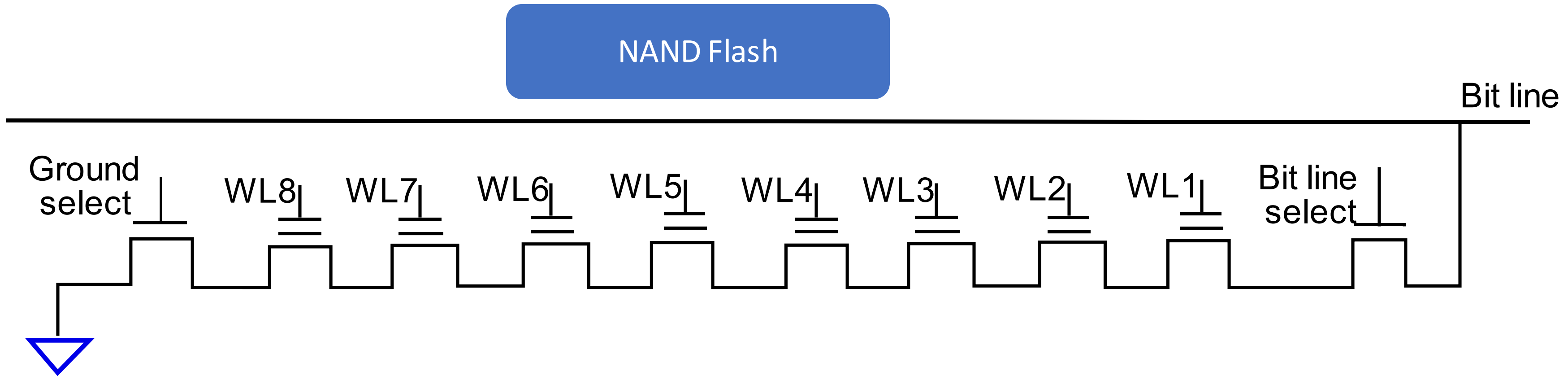
NOR Flash

NOR Flash



- It saves only 2 bits
- Only one of the transistors is enabled (voltage set between V_T and V_T^+)
- The voltage on the bit line is the reverse of the value stored in the cell
- Very similar to a DRAM cell

NAND Flash Cell



- A bunch of transistors connected in series
- Set the voltage on the bit line select and ground select line to 1 (enable them)
- Assume we want to read the value stored in the transistor WL5
- Set the gate voltage of the rest of the transistors to V_T^+ (conducting)
- 40-60% higher density as compared to NOR Flash

Multi-Level Flash

We can store multiple bits per cell.

We can store 1-4 bits per cell. For storing *n* bits, we need to realize 2^n distinct charge levels.

$$\begin{tikzcd}\text{Charge} \arrow[r] & \text{Threshold voltage}\end{tikzcd}$$

Accurately sense based on the current flow

Higher storage density

Lower endurance and higher bit error rate (need more ECC).

Blocks and Pages

These are storage devices

- Flash devices provide page-level access. Page size = 512-4096 bytes
- There is a need to accumulate writes, and then sync to the flash device
- Pages are grouped into **blocks**
- Blocks contain 32-128 pages

Read or write data at the level of pages

A page once written, cannot be written again (unless erased)

There is a need to erase a **block** first, and then write

Program/Erase Cycle

Program phase →

- Consider a fresh page (never written after being erased)
- Programming → write a 0
- By default, a cell contains a 1
- Cannot convert from 0 → 1 without erasing

Erase phase →

- Set all the bits in the block of pages to 1 (not programmed)

? How do we write to a page after writing to it once?

Answer: Copy the full block to a fresh block (non-programmed) other than the page that needs to be modified. Replace its contents with the new contents.

Reliability Issues: Wear Levelling

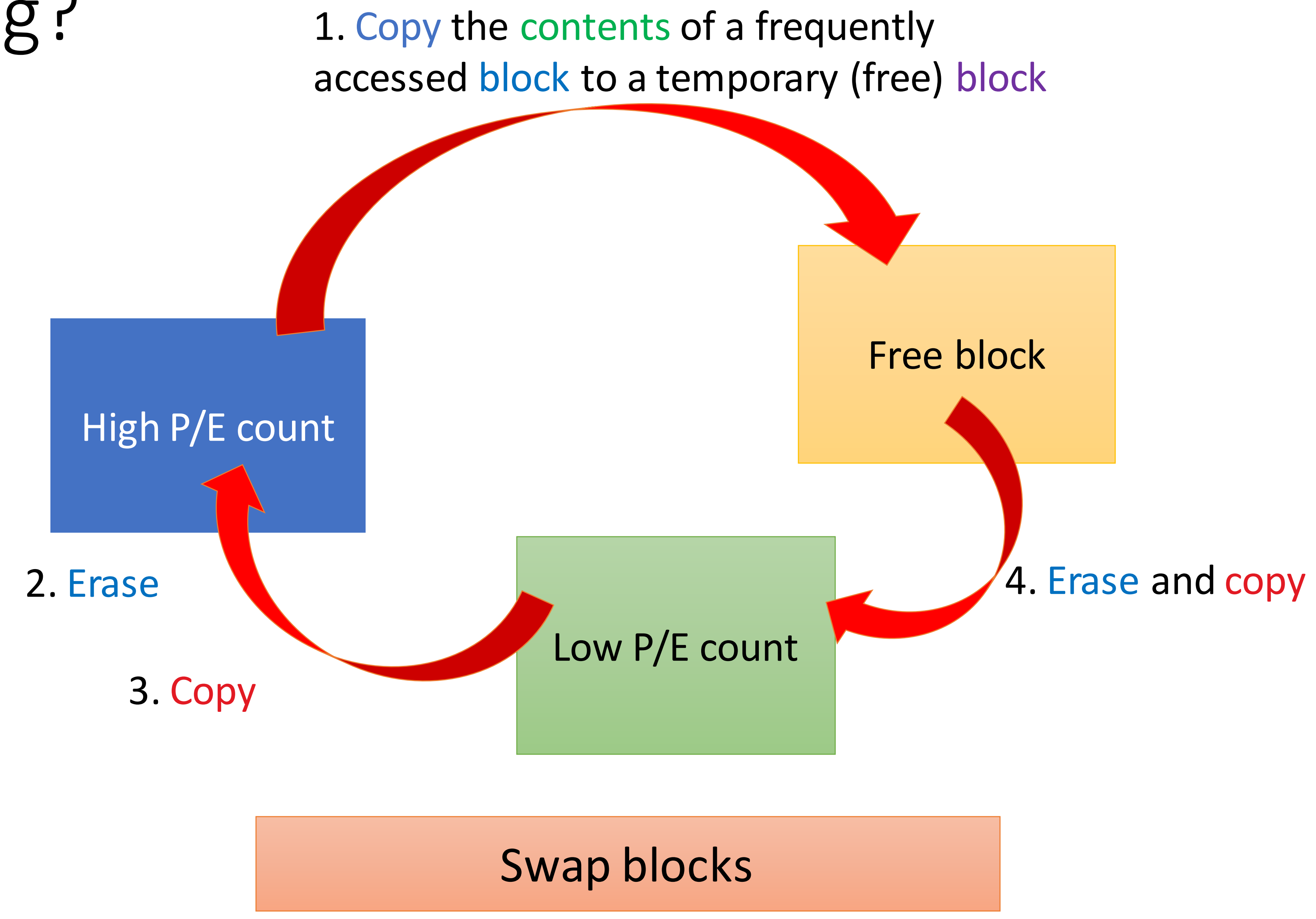
- A flash device can endure a **finite** number of P/E cycles (50-150k). Then, the oxide layer **breaks** down.

There is a need to ensure that all blocks wear out equally.

- Maintain a P/E counter for each block
- Assume that flash addresses are virtual addresses, which are mapped to physical addresses
- Maintain a mapping table
- This decouples the actual flash address from the address that is visible to software. Allows the hardware to remap obliviously.

$$\begin{tikzcd}\text{Virtual flash address} \arrow[r, "\text{mapping}"] & \text{Physical flash address}\end{tikzcd}$$

What can we achieve with remapping?



Read Disturbance

Definition

If we read a given cell continuously, the neighboring cells start getting programmed.

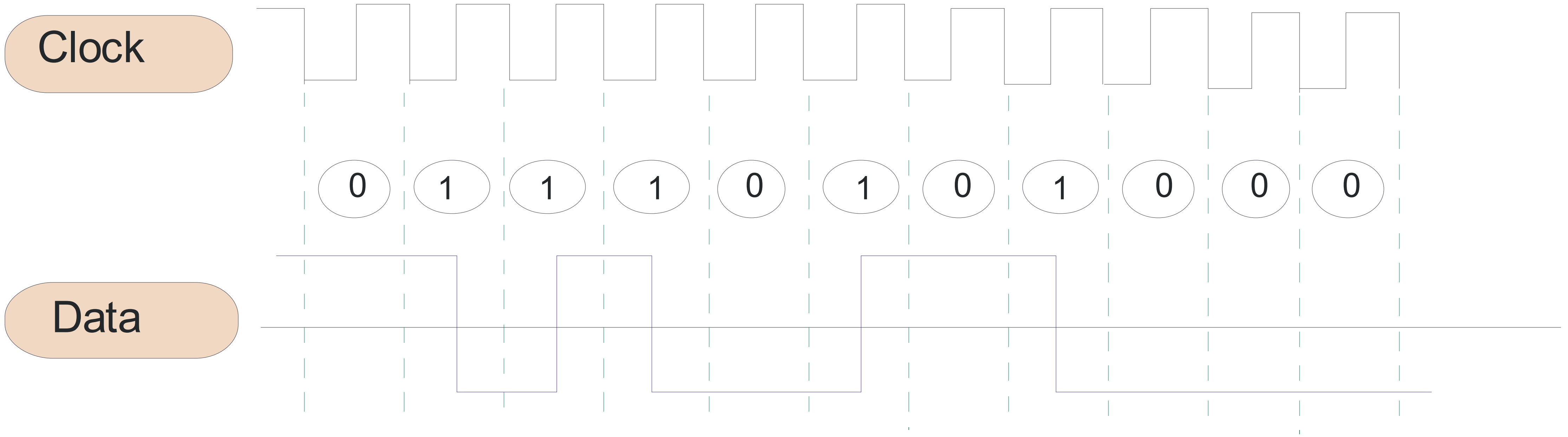
- Reason: In NAND flash, the rest of the cells have to be enabled (voltage at the gate $> V_T^+$)
- A few electrons accumulate in the floating gate.
- Over time, the cell gets programmed.

Solution

- Maintain a read counter for each block
- The moment it exceeds a threshold, copy the block to a new location. It starts from a fresh state.

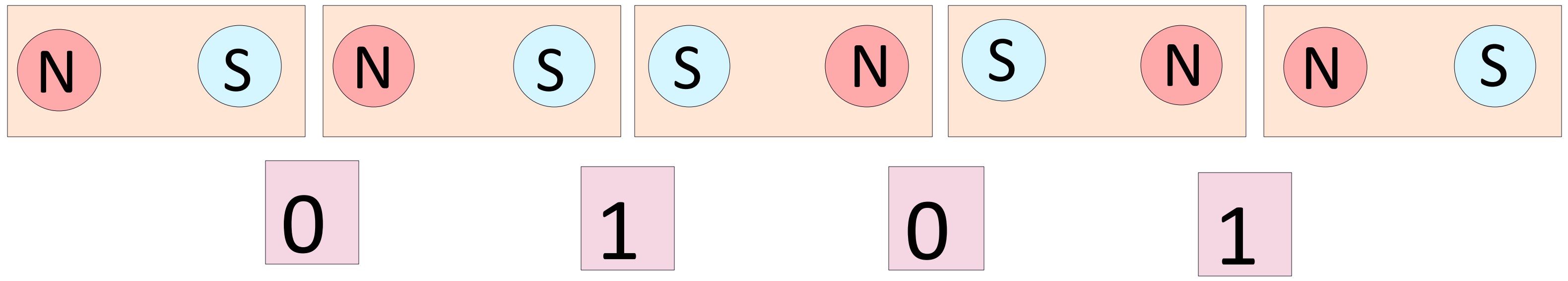
Hard Disks

NRZI (non-return to zero inverted)

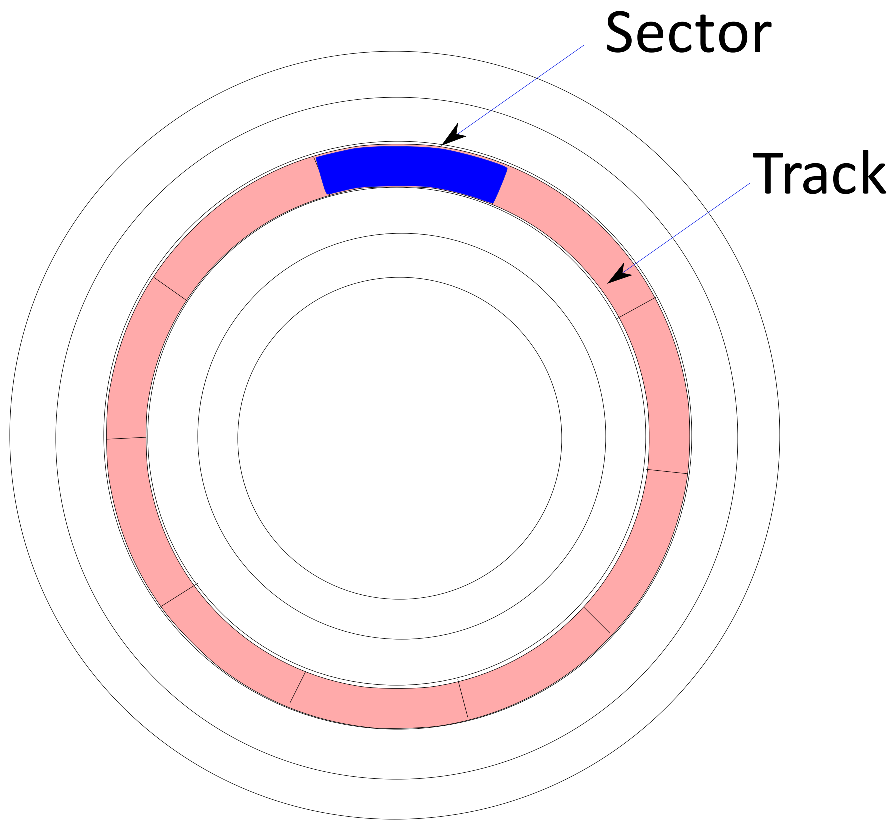


Data Storage in Hard Disks

- Magnetic storage (sequence of tiny magnets)
- **NRZI Encoding**
  - with dummy bits (added to synchronize signals)

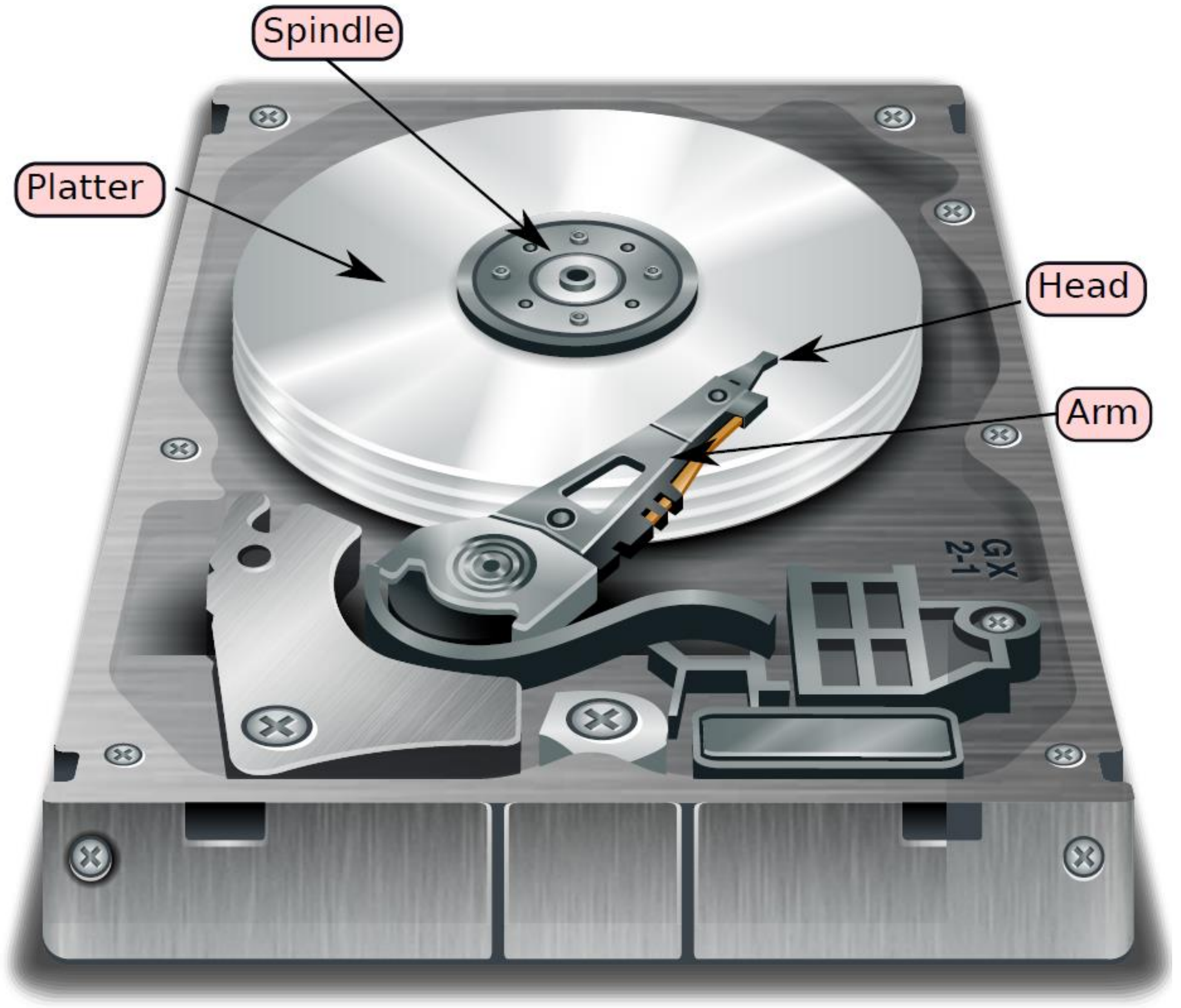


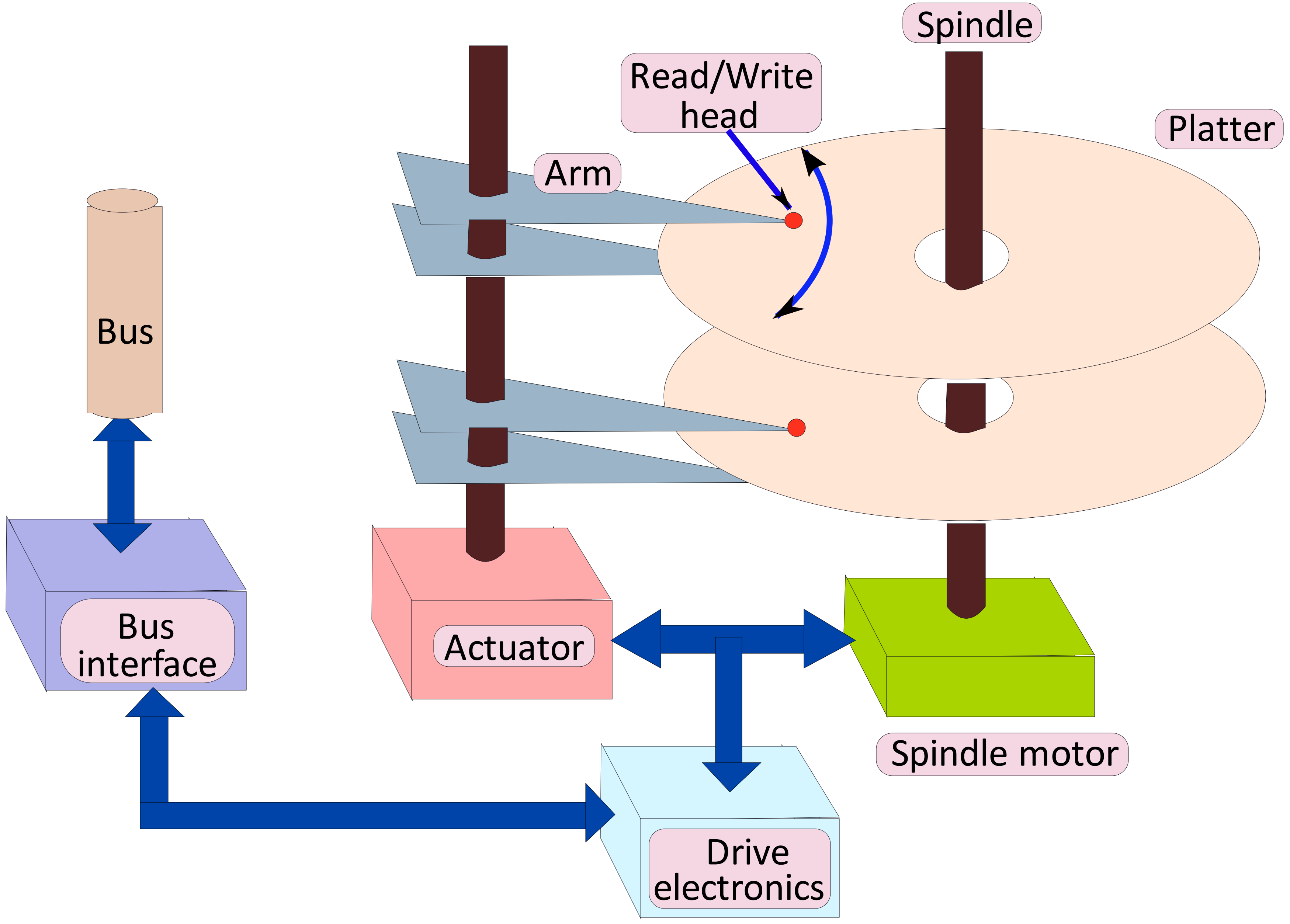
Structure of a Platter



A platter is divided into concentric rings (track)
Each track is divided into fixed size **sectors**

Structure of a hard disk




Spindle
Read/Write head
Platter
Arm
Bus
Bus interface
Actuator
Spindle motor
Drive electronics

Accessing a Given Sector

- Position the head on the correct track
  - seek time
- Wait for the desired sector to come under the head
  - Rotational Latency
  - Assume that the hard disk rotates at a constant angular velocity
- Read or Write data
  - Perform error checking, correction, framing
  - Transfer data to the CPU → (Transfer Time)
  - The transfer starts when the head is on the first bit of the sector

$$T_{disk\ access} = T_{seek} + T_{rot.latency} + T_{transfer}$$

Logical vs Physical Block Address

- Software programs use the logical block address to address a hard disk block (sector)
  - The hard disk internally converts the logical address to a physical address (recording surface, track, sector)
  - It dedicates a recording surface for storing this mapping information.
  - It also has a small cache (DRAM) to store the most recently used mappings
  - The hard disk can also use this mechanism to mark bad sectors and remap logical sectors to healthy physical sectors.

A file maybe split into many chunks and stored all over the disk. A defragmenter rearranges the blocks such that most of the blocks in a file are contiguous.

RAID – (Redundant Array of Inexpensive Disks)

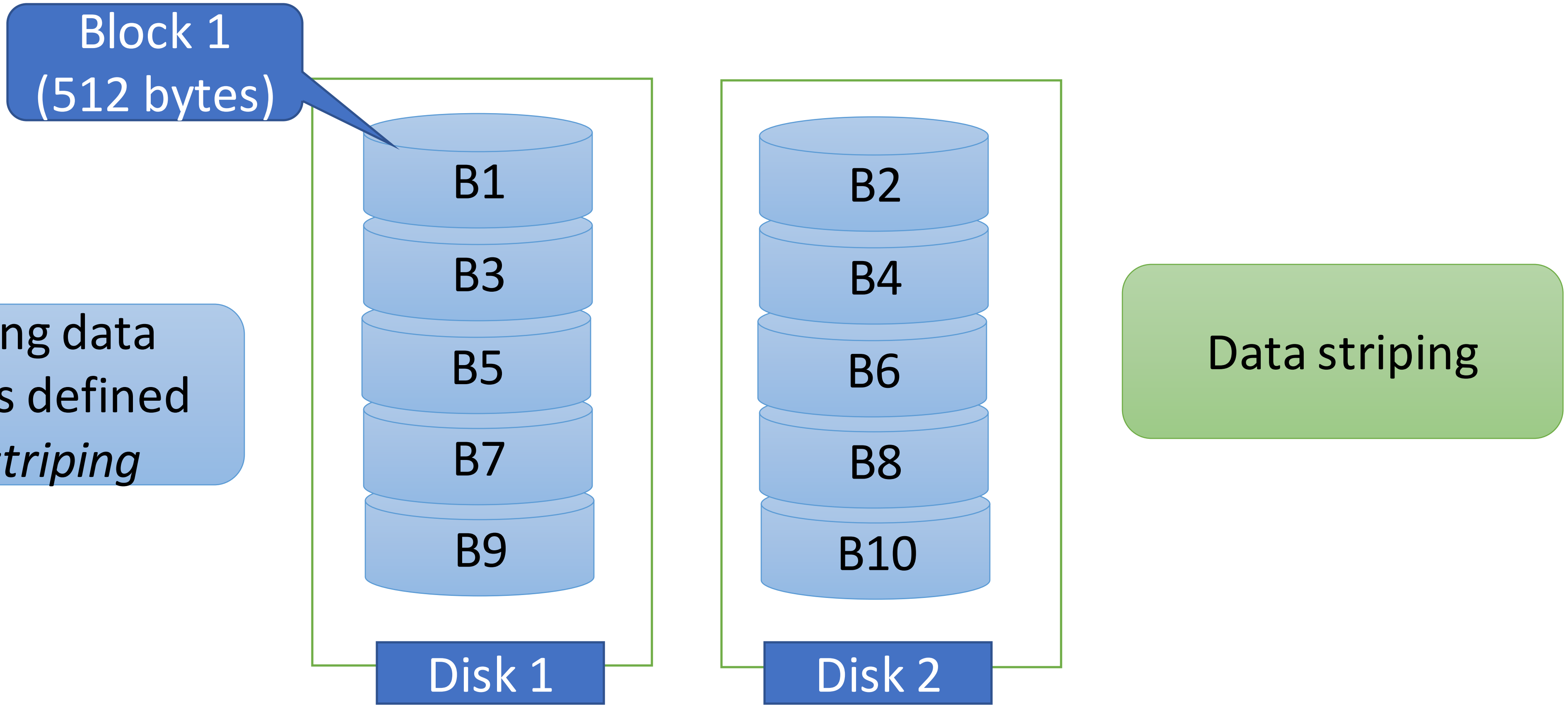
- Hard Disks tend to have high failure rates
  - Too many mechanical components
  - High temperature sensitivity
- What do we do?
  - Do we lose all our data ....
  - NO
- Use RAID
  - Redundant disks
  - To tolerate faults, and recover from failures

RAID – II

- What about bandwidth?
  - Assume we want to read two different blocks from the same disk?
  - We need to read them serially.
  - Low Bandwidth
- How can we increase bandwidth
  - Read from multiple disks in parallel.
  - **Solution**: RAID

Additional resources increase performance and improve reliability

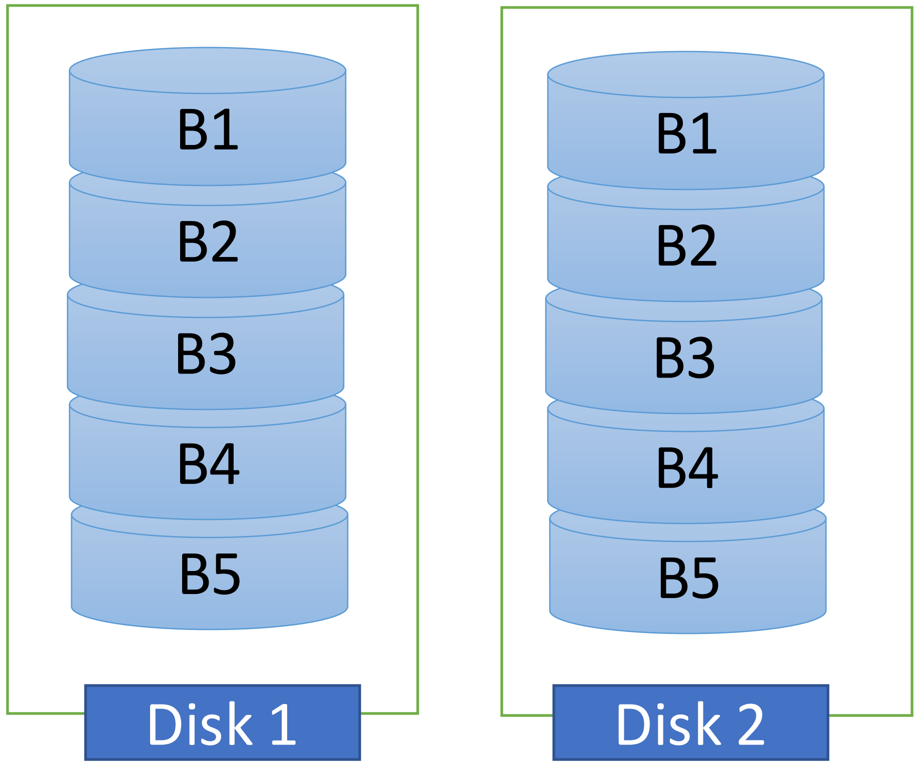
RAID 0



Distributing data across disks defined as *data striping*

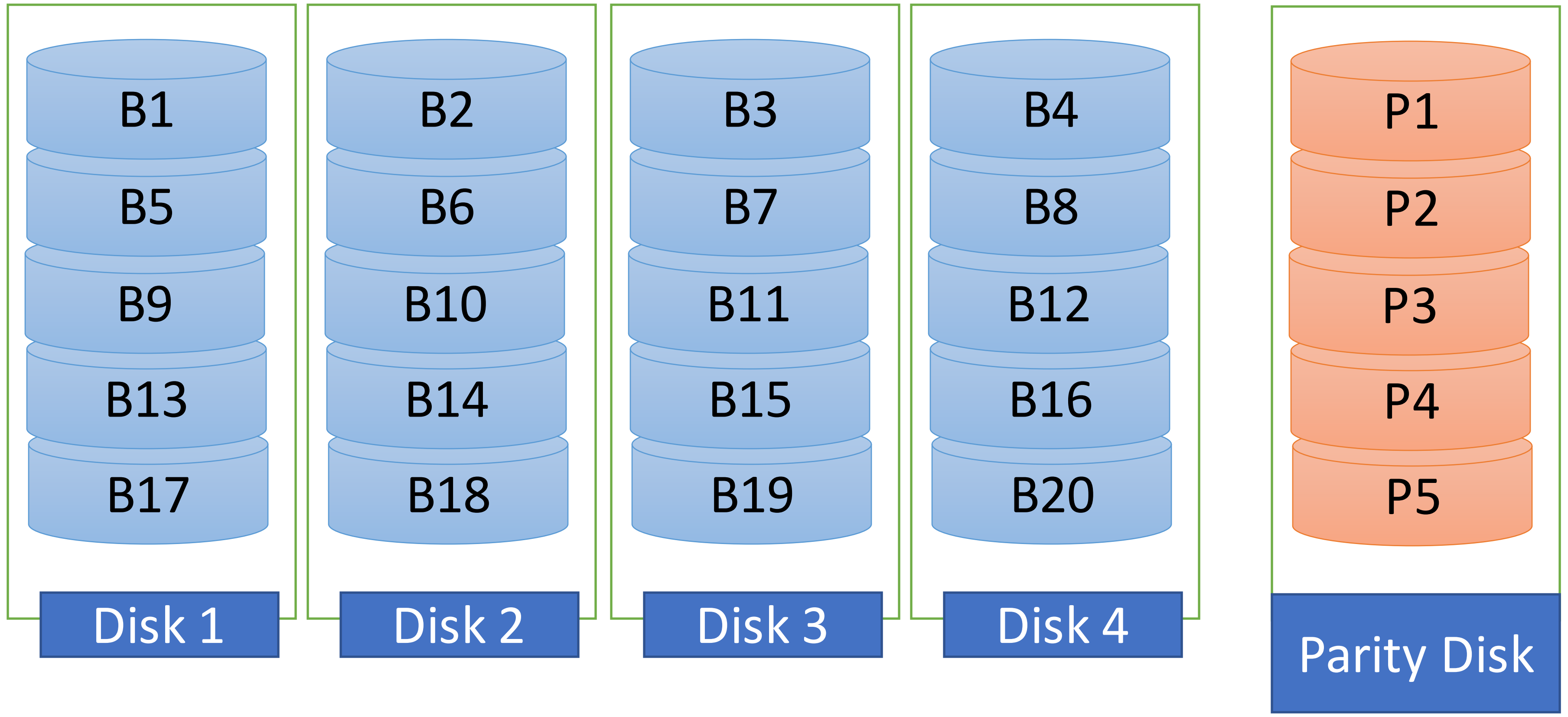
- No Reliability
- Allows us to read even and odd blocks in parallel

RAID 1



- Immune to one disk failure
- Can read blocks B1 and B2 in parallel
- 100% overhead in storage

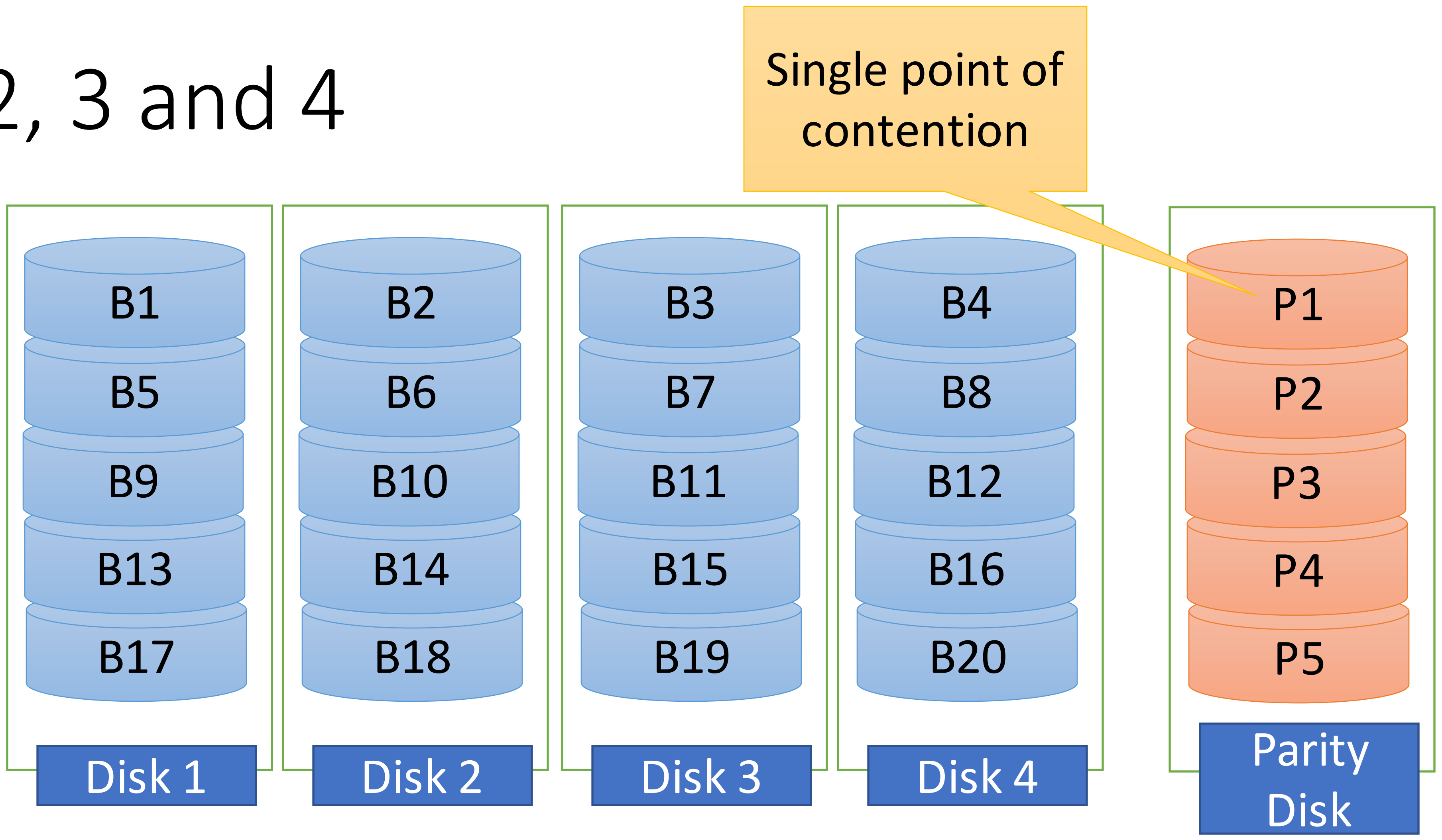
RAID 2, 3, 4



- Immune to one disk failure
- Parity block → P1 = $B1 \oplus B2 \oplus$ B3 $\oplus B4$

- 25% storage overhead
- If a disk fails, add a new disk and reconstruct it from the parity bits

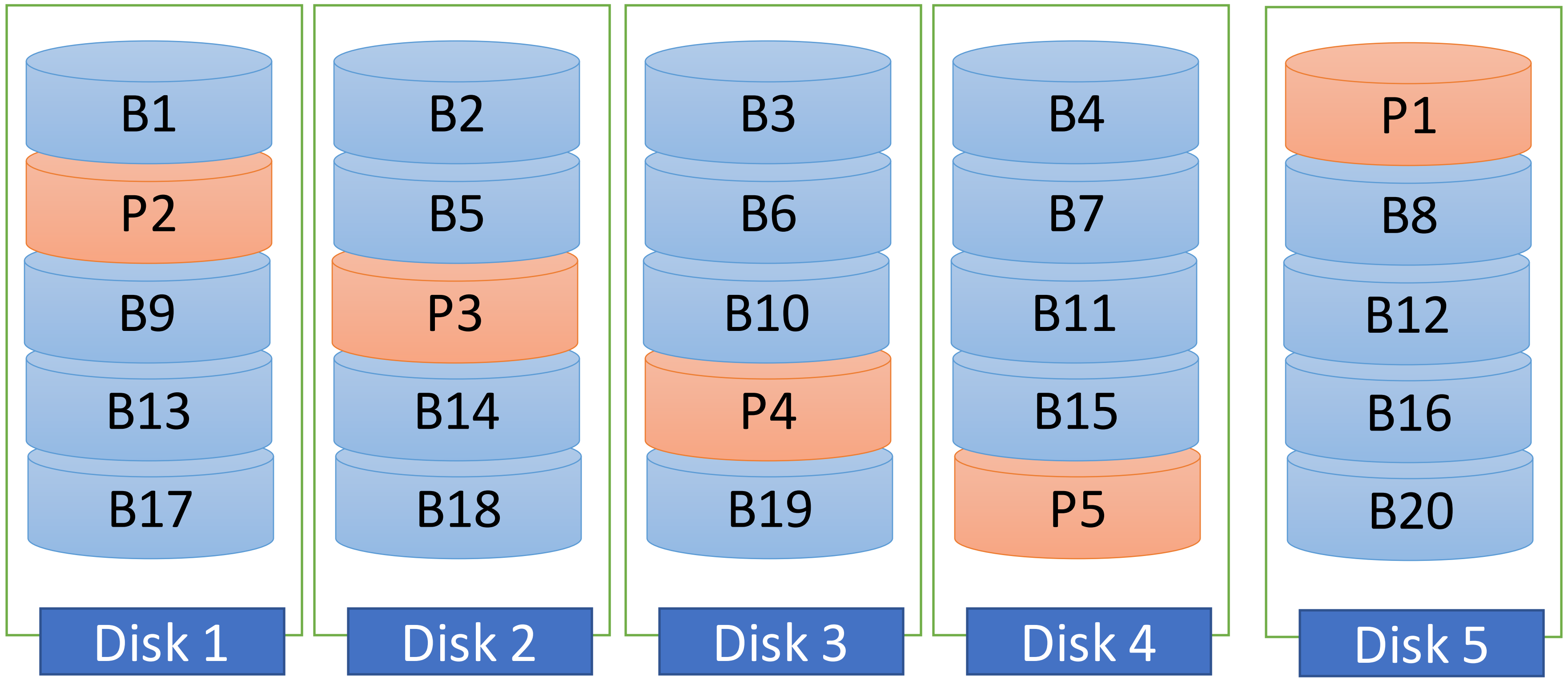
RAID 2, 3 and 4



RAID	Block Size
2	1 bit
3	1 byte
4	1 block (512 bytes)

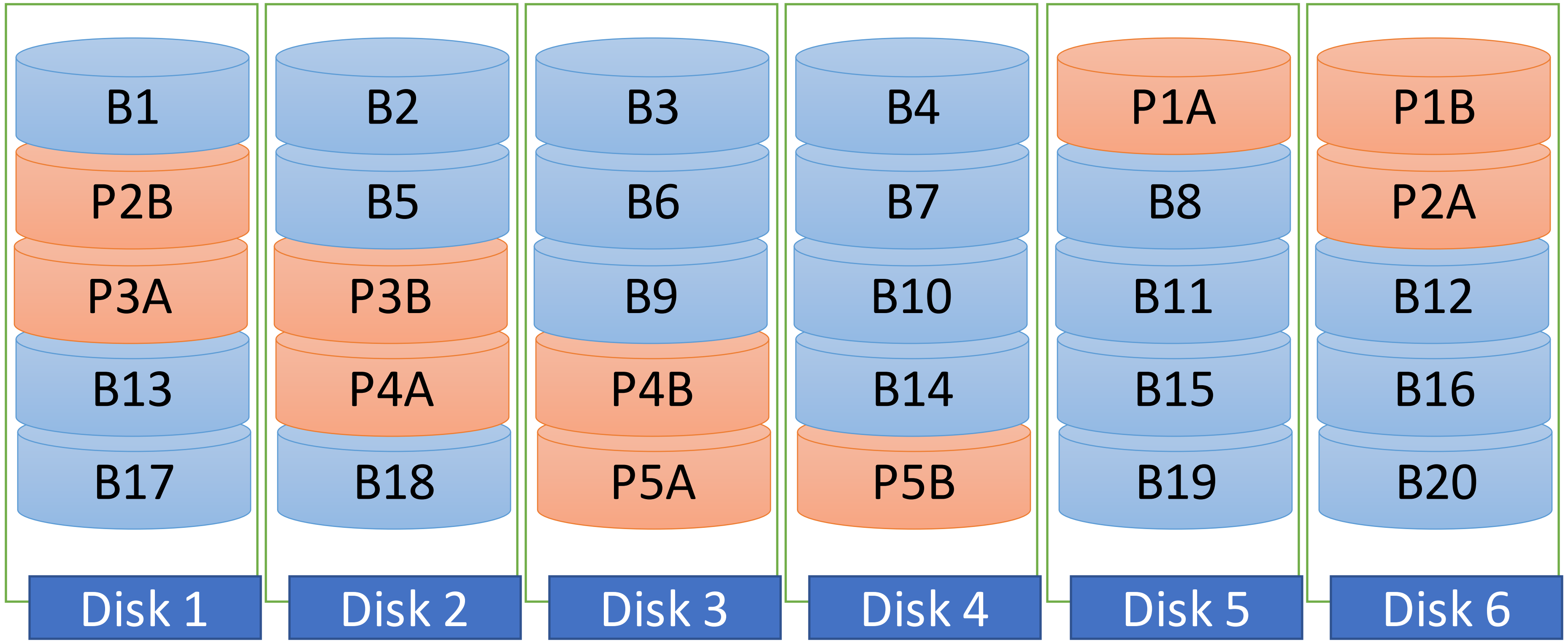
For reading a single block → access all the disks

RAID 5



- To avoid this problem: Distribute the parity blocks across the disks
- Reliable + high bandwidth

RAID 6

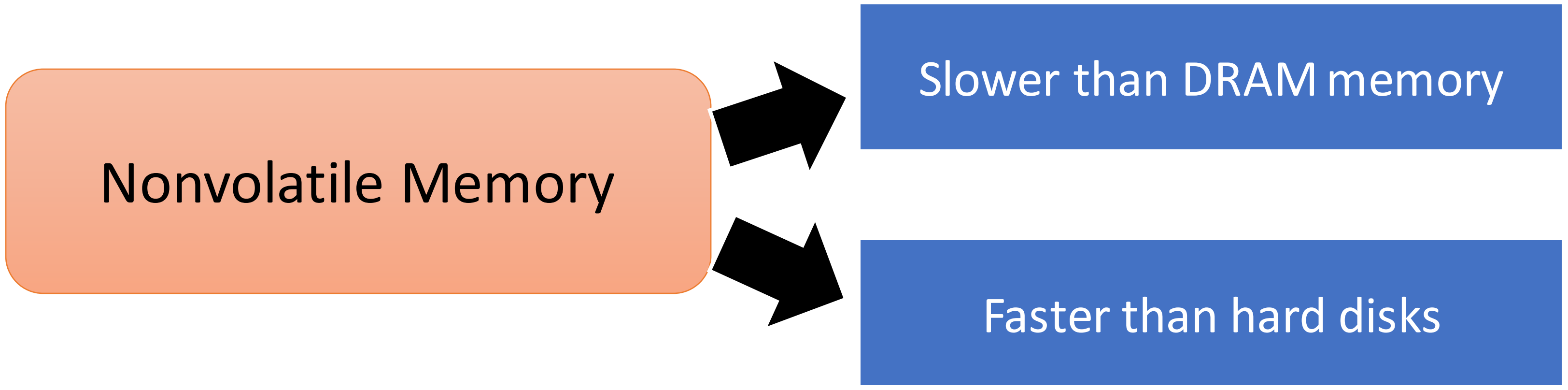


- Immune to two disk failures
- Have two parity blocks (parity is computed differently)
- High reliability at the cost of increased storage overhead

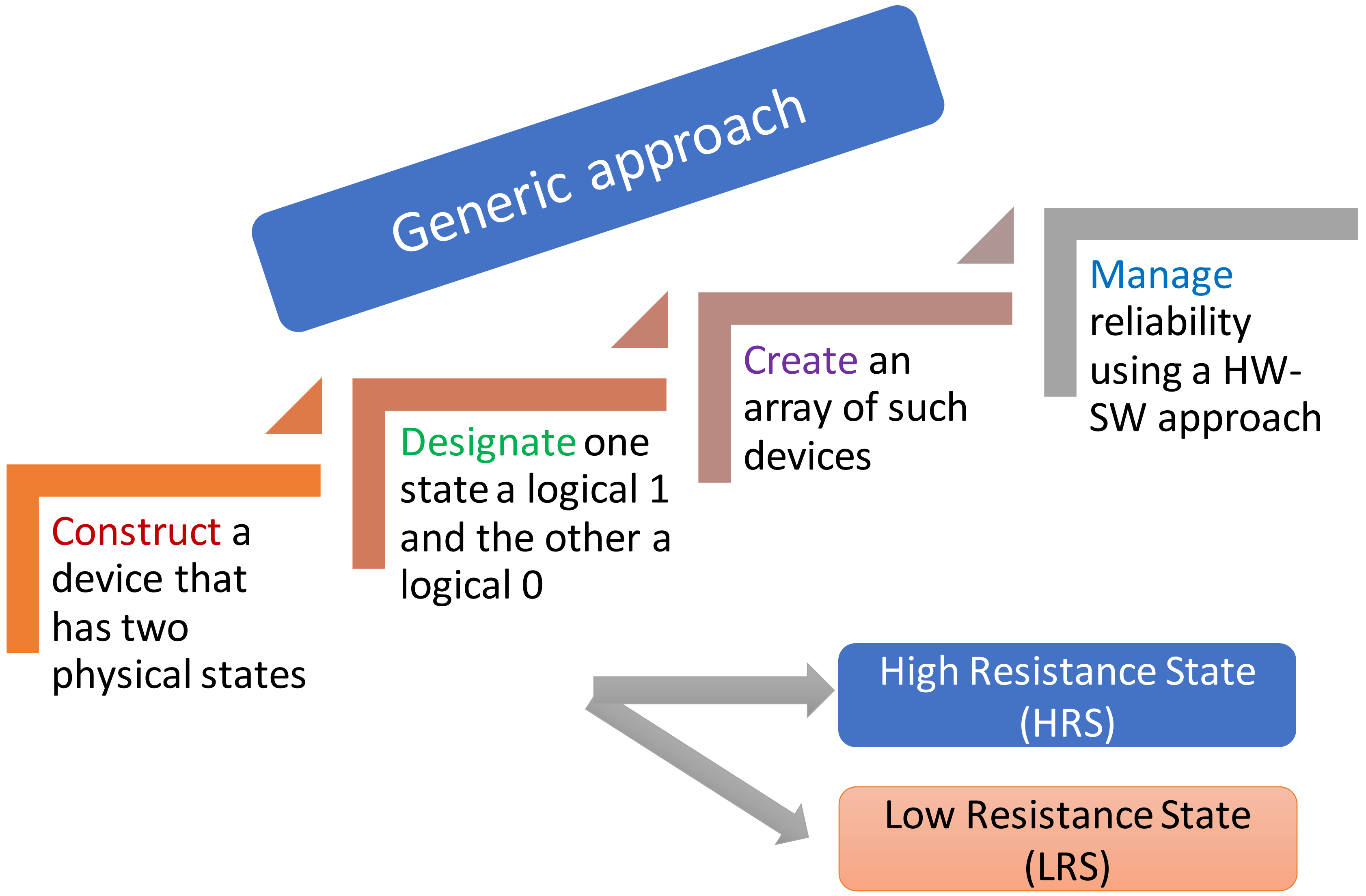
Memories as Storage Devices

The main problem with DRAM **memory** is that when the power is turned off, the data disappears. Can this be solved?

Answer: Use some kind of nonvolatile memory (NVM) whose data is not lost upon a loss of power. Can also act as a storage device.



Basic Idea of NVMs (Nonvolatile memories)



Generic approach

Construct a device that has two physical states

Designate one state a logical 1 and the other a logical 0

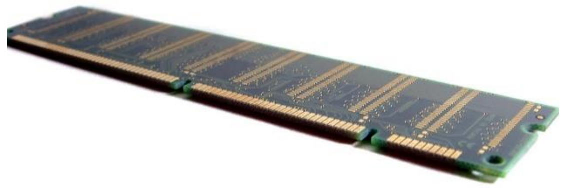
Create an array of such devices

Manage reliability using a HW-SW approach

High Resistance State (HRS)

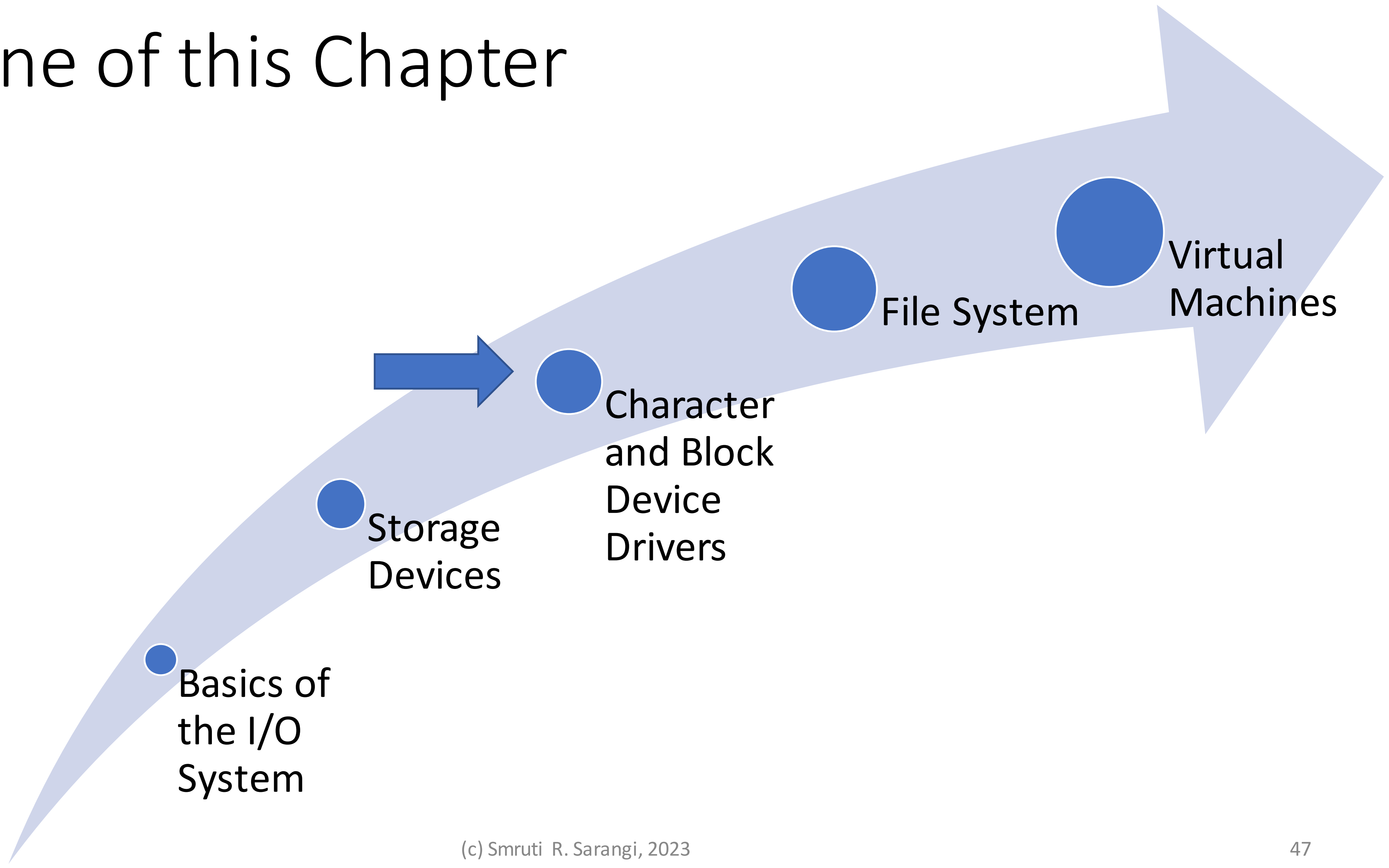
Low Resistance State (LRS)

Types of NVMs



Type of NVM	Basic Principle
Flash memory	A floating gate transistor that has an additional gate that can be made to store electrons (based on the magnitude of the voltage)
Ferroelectric RAM (FeRAM)	The degree of polarization of the ferroelectric material is a function of the voltage applied to it in the past.
Magnetoresistive RAM (MRAM)	The direction of magnetization of a ferromagnetic material is a function of the direction of current flow through it (in the past)
Phase change memory (PCM)	We use a small heater to change the state from amorphous to crystalline.
ReRAM	Based on the history of the voltage applied, a filament forms based between the anode and cathode. The resistance is determined by the width of this filament.

Outline of this Chapter



Basic Concepts

Represent each device as a file (devfs is in /dev). Use standard file-access primitives.

Block device

- The device has a set of addressable locations
- The minimum transfer size is 1 block (tens of bytes at least)

A device driver provides a convenient interface to HW

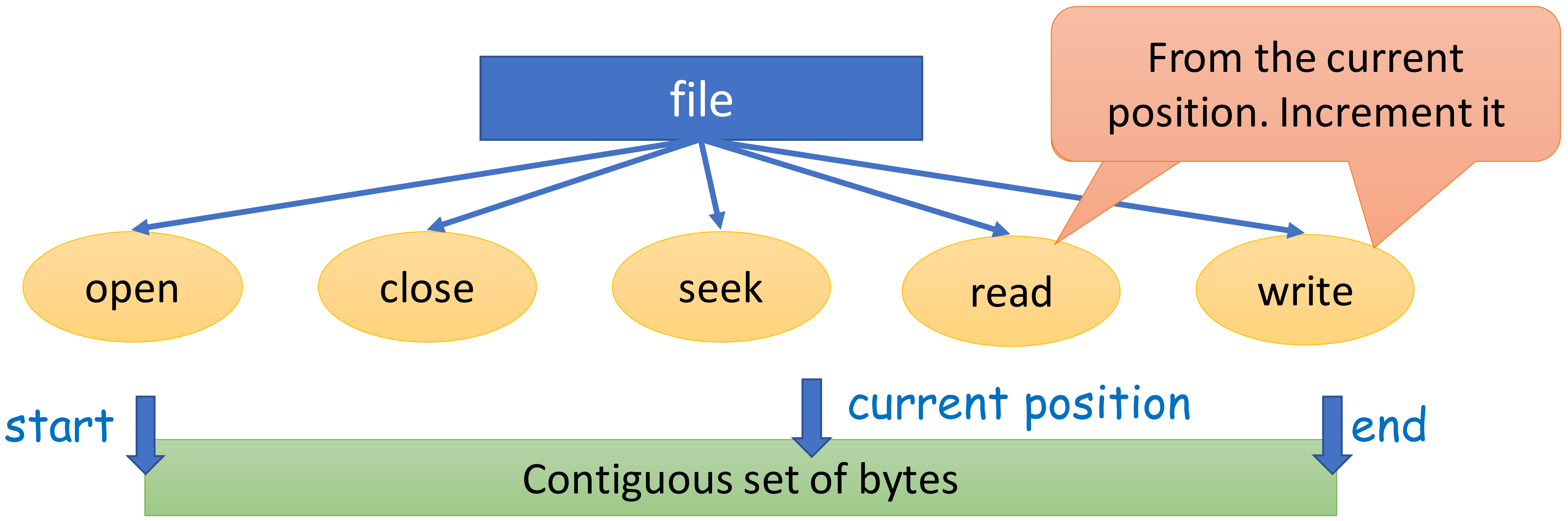
Character device

Read and write groups of characters at a time. Typically, does not have addressable locations. May have the notion of a *position* in a stream of characters.

Linux's File Interface: Used by Character Device Drivers, Regular Files, etc.

Standard interface

- Represent everything including block and character devices as files.



open

close

seek – go to an address

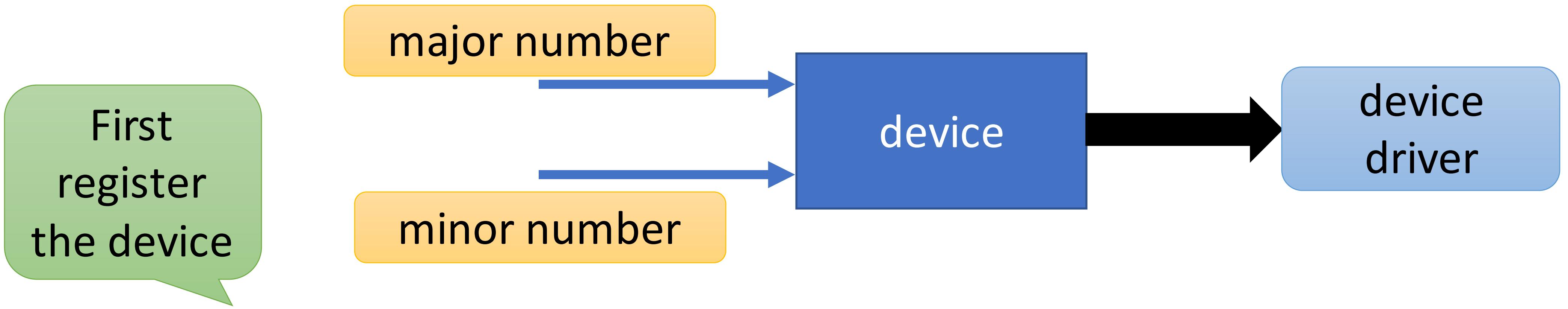
read

write

Block Devices

Registering a Block Device

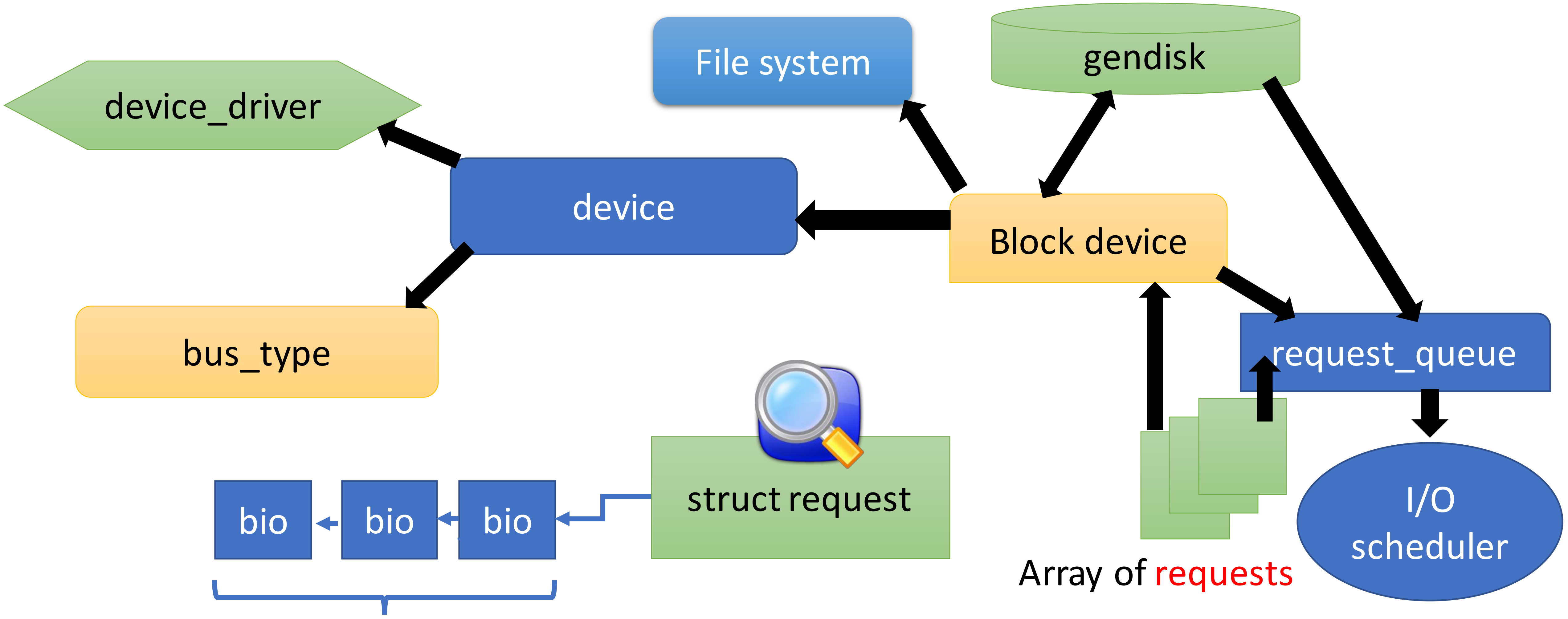
/block/genhd.c



- *register_blkdev (unsigned int major, const char *name, void (*probe)(dev_t devt))*
  - Register the device with the kernel (listed in /proc/devices)
  - The kernel will subsequently use the major number to identify the device
  - *devt* is a u32 that identifies the device: 12 bits for the major number and 20 bits for the minor number
  - The minor number is used only by the device driver

Details

Overall Structure of the Block I/O System



List of memory regions

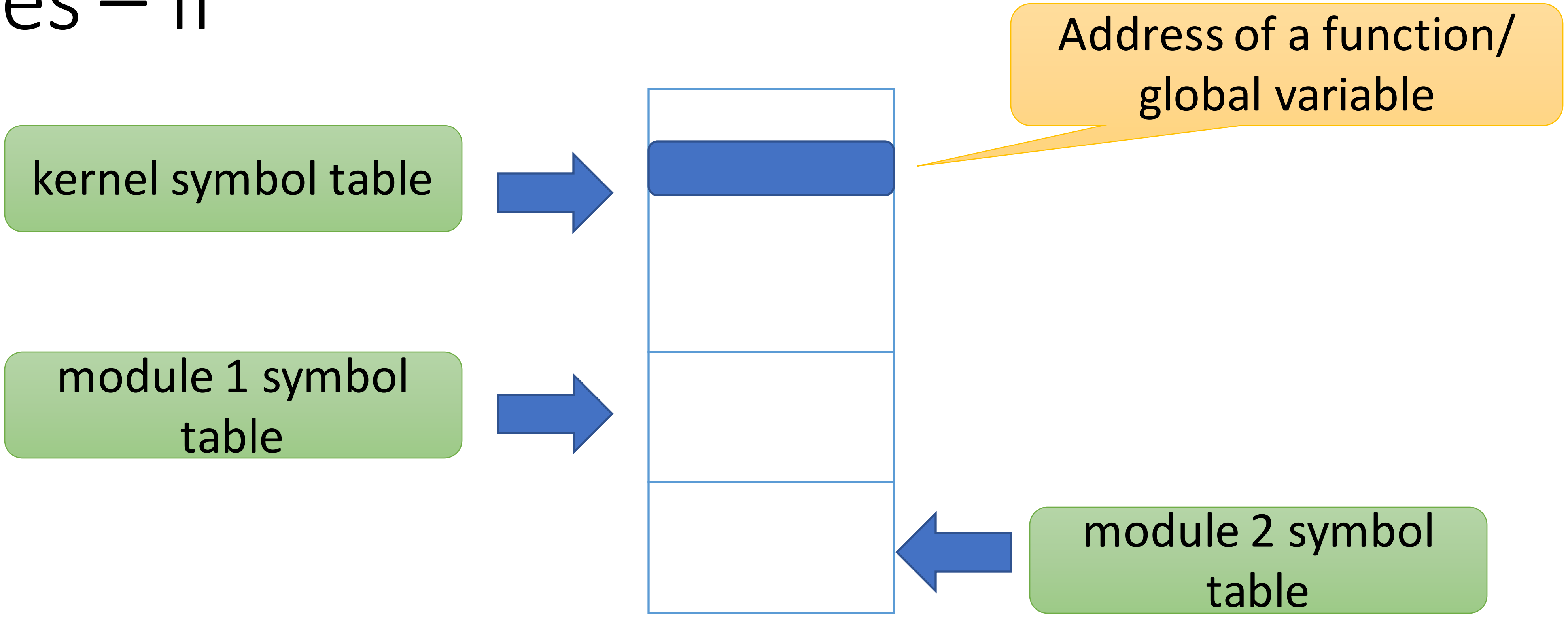
Notion of a Module and Device Driver

- The Linux kernel is a large monolithic piece of code
  - We should not be forced to recompile the kernel for every single driver that we add → Regular users should never do this
  - ✖ We should also not be forced to statically link all possible drivers
  - A module is like a dll or shared object in a kernel
  - Load the device drivers as a module: make the module a part of the kernel code (not necessarily the binary) or load at runtime
- *Advantage*: size and memory footprint arguments (as dlls and shared objs)
- Load a module:
  1. *insmod* command (invoked by the superuser)
  2. The kernel does it automatically based on the action (such as mounting a filesystem)

Loading and Unloading a Module

- The role of *insmod* or the function called by the daemon *kerneld* is to do the following:
  - Locate the code of the module
  - Map it to the kernel space
  - Increase the kernel's runtime code size and memory footprint
  - Like a linker, change the addresses of all the symbols that the module plans to use from the **global symbol table** (kernel and other modules)
  - Add the symbols it exports to the global symbol table
- Unloading a module
  - Maintain a reference count with each module
  - Once it reaches zero and the user wishes to unload the module, follow the reverse sequence of steps.

Modules – II



- The modules can see (and use) the kernel's symbol table
- A module can also see (and use) the symbol tables of other modules
  - We can make one module load after another to enforce a relationship

/include/linux/device/driver.h

struct device_driver

```
struct device_driver {

        const char *name;
        struct bus_type *bus;
        struct module *owner;

        struct of_device_id ...;
```

Name of this device, type of the bus, name of the module that corresponds to the driver.

Name and type of the driver (used to match a device)

function pointers: probe, sync_state, remove, shutdown, suspend, resume

Called to bind the driver with a device

Device specific information

```
}
```

Add the Disk to the System

device_add_disk(...)

Define a few data structures

/include/linux/device.h

struct device

```
dev_t devt;
u32 id;

struct device *parent;
struct bus_type *bus;
struct device_driver *driver;

struct bus_dma_region *dma_range_map;
struct list_head dma_pools;
struct device_physical_location
*physical_location;
```

Device identifier

High-level information

DMA regions

Location of the device: dock, lid, position on the motherboard

A Note about *bus_type*

Devices are connected to the processor via a bus (represented by bus_type). *struct bus_type* contains a list of function pointers and generic attributes like the name of the bus.

- Every bus has a single device that is considered the parent (this could be virtual also). Along with the parent, it contains multiple devices and device drivers.
- Common function pointers:
  - probe → Check if a device matches a driver.
  - sync_state → Sync the device's state with its software state
  - remove/shutdown/suspend/resume/online/offline → Self-explanatory
  - dma_configure → Manage the DMA space associated with the bus

/include/linux/blkdev.h

Generic Disk (*struct gendisk*)

```
int major;
char disk_name[];

struct xarray part_tbl;     /* partition table */
struct block_device *part0;
struct block_device_operations *fops;

struct request_queue *queue;
```

major number and disk name

Pointer to the partition table and block device (for the first partition)

List of pending requests

- This is a wrapper on a block device
- It also contains a pointer to a struct of type *block_device_operations* → contains a list of function pointers for opening, releasing and operating a block device

/include/linux/blk_types.h

struct block_device

A disk-like device that supports a file system and a request queue

```
sector_t bd_start_sect;
sector_t bd_nr_sectors;

dev_t bd_dev;
struct device bd_device;

struct super_block *bd_super;
struct inode *bd_inode;

struct gendisk *bd_disk;
struct request_queue* bd_queue;
```

size in sectors

pointer to the device

Details of the file system on this device

Pointer to a generic disk device

Queue of requests

struct request_queue

/include/linux/blkdev.h

one per device

```
struct request_queue {
        struct request *last_merge;
        struct elevator_queue *elevator;

        per CPU software request queue
        Hardware request queues
}
```

pointer to the last request

I/O request queue that interfaces with the I/O scheduler

per CPU queues that store requests

Software front-end of HW request queues

- A request queue for a certain device contains both software request queues and hardware queues (with a software counterpart that periodically syncs with it)

Software and Hardware Queues

- The per-CPU software queues are lockless queues used to hold requests from a single CPU
- They can be merged and reordered.
- They are then sent to the hardware dispatch queues (part of the device driver)
- Here, there are requests from all CPUs: more merging and reordering
- The driver sends the requests to the device at the right points of time (after appropriately delaying)
- Every request and its response (from the device) is identified with a tag
  - Needed for concurrent requests

struct request

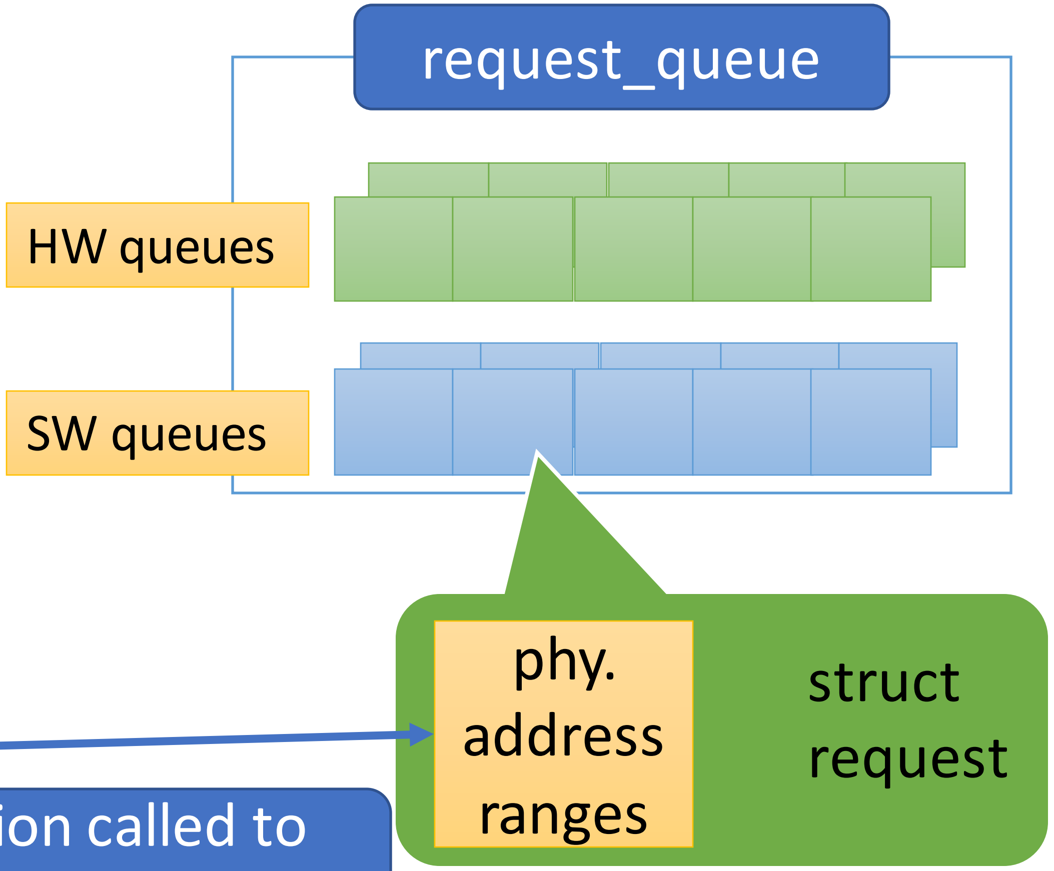
/include/linux/blk-mq.h

```
struct request {
        struct request_queue *q;
        struct blk_mq_ctx *mq_ctx;
        struct blk_mq_hw_ctx *mq_hctx;
        struct block_device *part;

        unsigned int __data_len;
        sector_t sector;
        unsigned int deadline, timeout;

        struct bio *bio;
        rq_end_io_fn *end_io;
}
```

Back pointers

Params

Function called to complete the request



struct bio (block operations currently active)

https://www.kernel.org/doc/Documentation/block/biodoc.txt

/include/linux/blk_types.h

```
struct bio {
        struct block_device     *bi_bdev;
        struct bio_vec          *bi_io_vec;
}
```

Back pointer to the block device

Array of I/O ranges

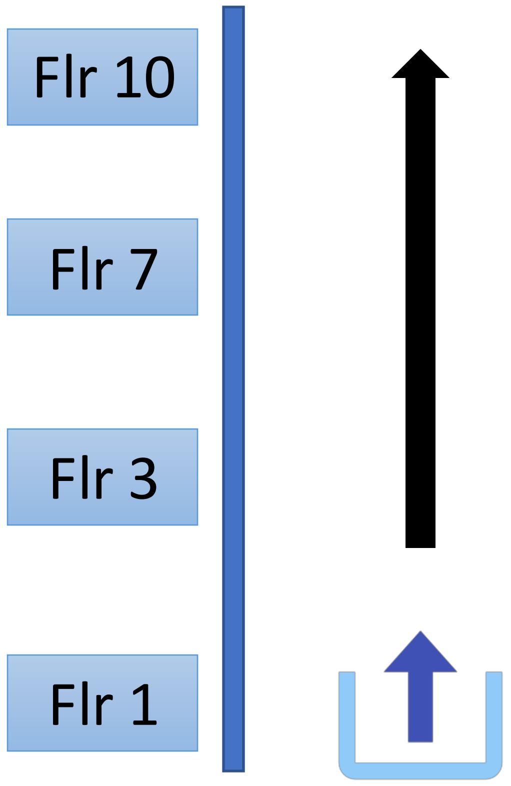
phy page, length, offset

- It is possible to merge multiple bio structs or bio-vectors (<page,len,offset> tuples) to create a larger I/O request
- Storage devices like disks, SSDs and NVMs prefer sequential read/writes from large contiguous regions.

I/O Scheduling Algorithms (Elevators)

- Classic elevator scheduling algorithm
  - An elevator keeps going up until there are requests and then keeps going down until there are requests (regardless of the FIFO order and priority)
  - Early elevator-based disk scheduling algorithms did the same: move from the innermost track to the outermost track and then back (allowed us to leverage spatial locality). This minimizes back and forth movement of the disk head.
- Linux uses three kinds of I/O scheduling algorithms

Deadline BFQ Kyber



I/O Scheduling Algorithms

Deadline scheduler

- Store requests in queues (in sector order) and serve them in that order (prefer reads). Also, store requests as per their deadline. When a deadline expires, schedule all the requests that have timed out (from the deadline queues). After finishing this, start serving from the default queues.

BFQ (Budget Fair Queueing)

- Similar to CFS Scheduling, BFQ gives processes a certain number of sectors in a large "sector slice".

KYBER

More about I/O Schedulers

They do three things

Delay

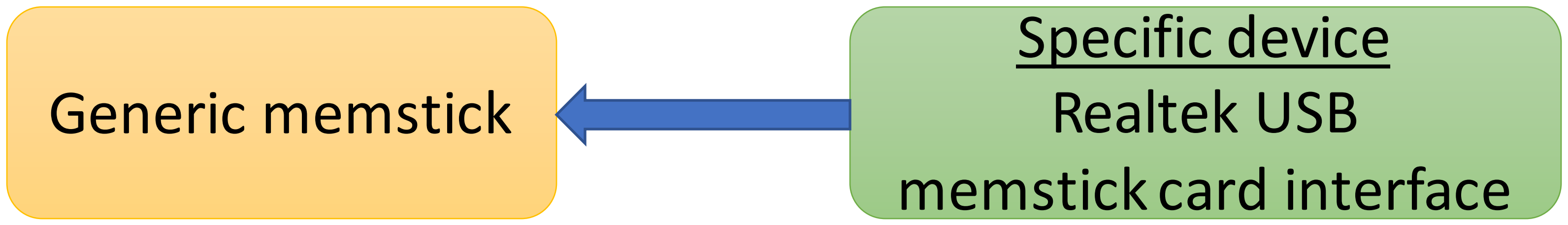
Basic Device Driver (Sony's Memory Stick Driver)

/drivers/memstick/core/mspro_block.c

- The module starts by calling *mspro_block_init*
  - Registers the driver name and block device "mspro_block"
  - The driver is represented by a structure (*mspro_block_driver*) that contains multiple function pointers to probe/initialize, remove, suspend and resume
- The initialization function (*mspro_block_probe*)
  - Initialize the card
  - Create a sysfs entry: special file system to expose specific attributes of kernel objects like devices.
  - Initialize the disk structures: *gendisk, block_device, request_queue, tags*
  - Add the disk to the device

Transferring Data to/from the Memory Card

/drivers/memstick/host/rtsx_usb_ms.c



- *rtsx_usb_ms_handle_req*
  - Keep a queue of requests – get the next request (part of memstick's interface)
  - Based on the type: read, write, or do a bulk transfer
  - For reading and writing: create a set of USB read/write commands and pass them on to the low-level USB driver
  - For a bulk transfer →
    - Setup a one-way pipe: a virtual file in which one process can write and another can read
    - Initiate the transfer by writing USB commands to the corresponding registers

open

close

read

write

Character Devices

USB Keyboard

/drivers/hid/usbhid/usbkbd.c

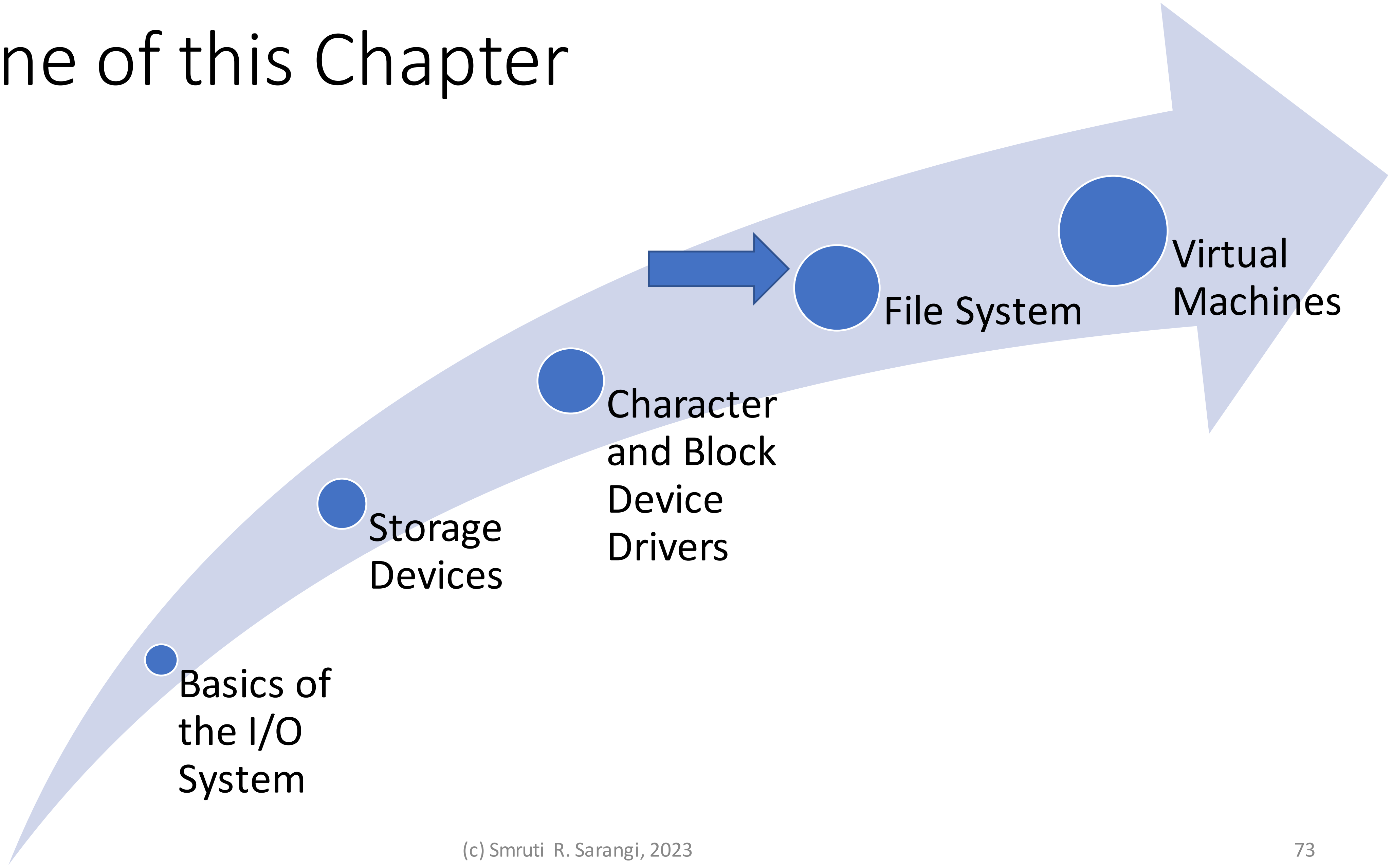
- *usb_kbd_probe* is the entry point for adding a module
- Allocate a generic input device. Associate it with the keyboard device.
- Initialize all the function pointers that will be called when the module will be opened, closed, and a key press event will be processed
- Two crucial functions for registering and deregistering character devices
  - *register_chrdev*: Register the major and minor numbers of the device. Register the driver structure (list of function pointers).
  - unregister_chrdev: Do the reverse

Register interrupt handler: *usb_kbd_irq*

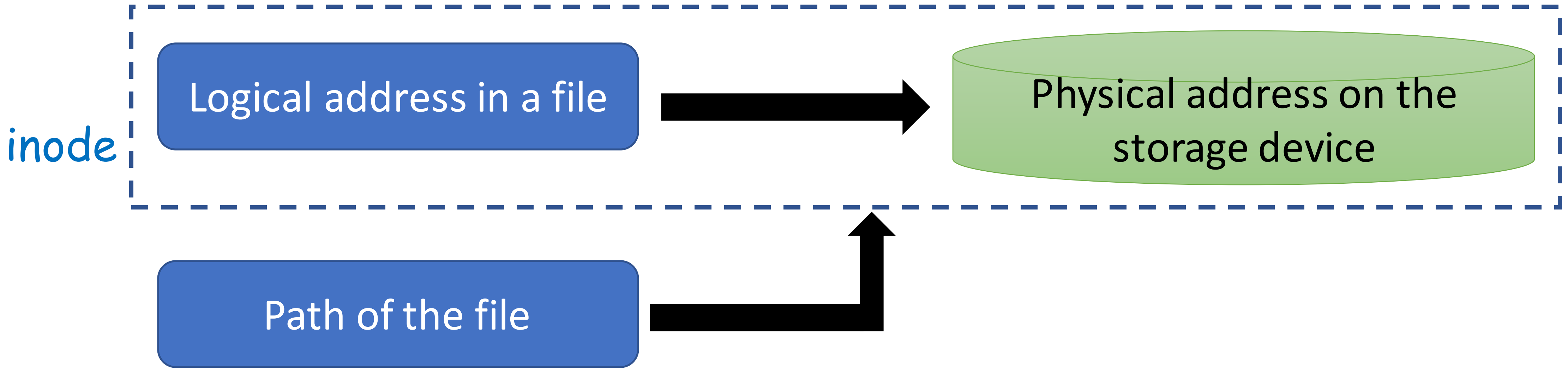
```
void usb_kbd_irq(struct urb *urb)
```

- *struct urb*
  - USB request block: generic data structure to hold information pertaining to USB device requests and responses
- Check the status of all special keys: Ctrl, Alt, Scroll Lock, …
- Check if the same key has been pressed twice (continued to be pressed down)
- Access a table to read the Ascii code from a table that is indexed with the code sent by the USB keyboard
  - Report the key pressed
  - Take special action for keys such as Num Lock, Caps Lock, etc.

Outline of this Chapter

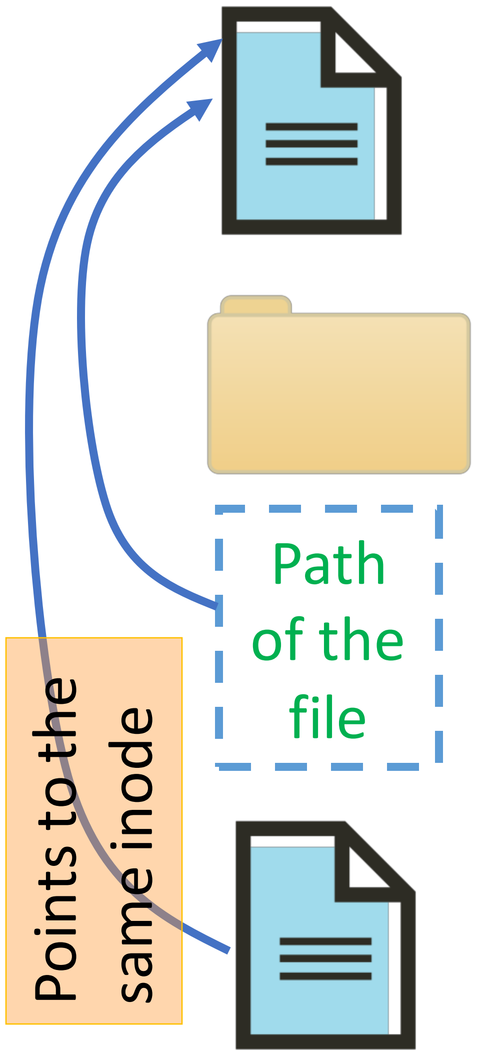


File System


inode
Logical address in a file
Physical address on the storage device
Path of the file

- Every disk can be divided into partitions: logical disks
- Each partition may have a boot block (used to boot the OS) (typically starts from the 0th storage location of the storage device)
- Every file system then has a superblock
- Location of the root inode ("/" in Linux)
- Metadata: max. file size, max. length of a filename, …

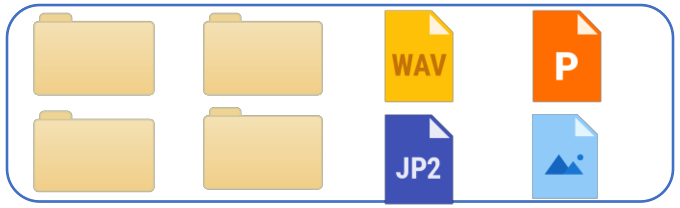
File, Directory and File System



File

Contiguous array of bytes.

Directory

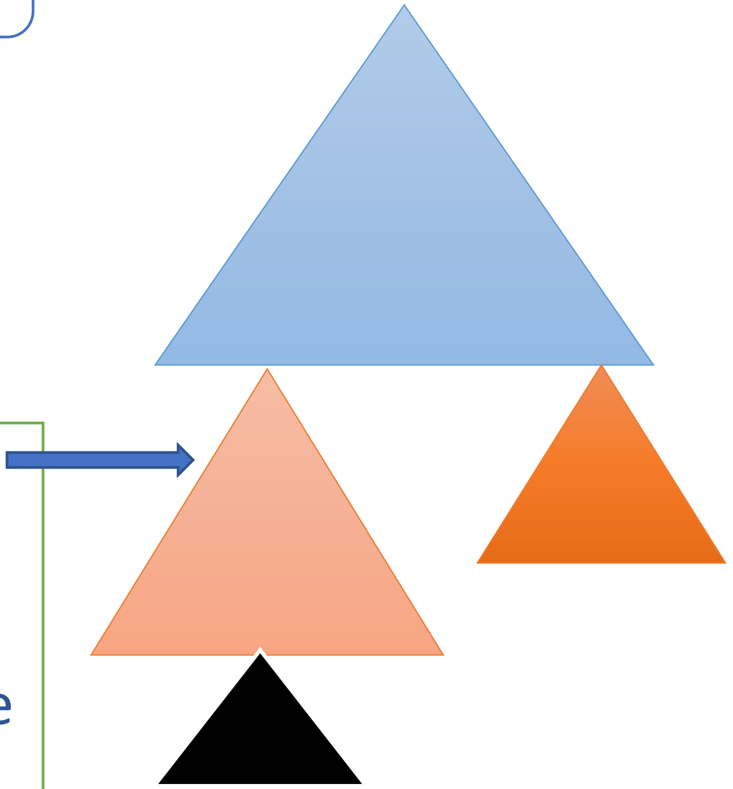


Symbolic link

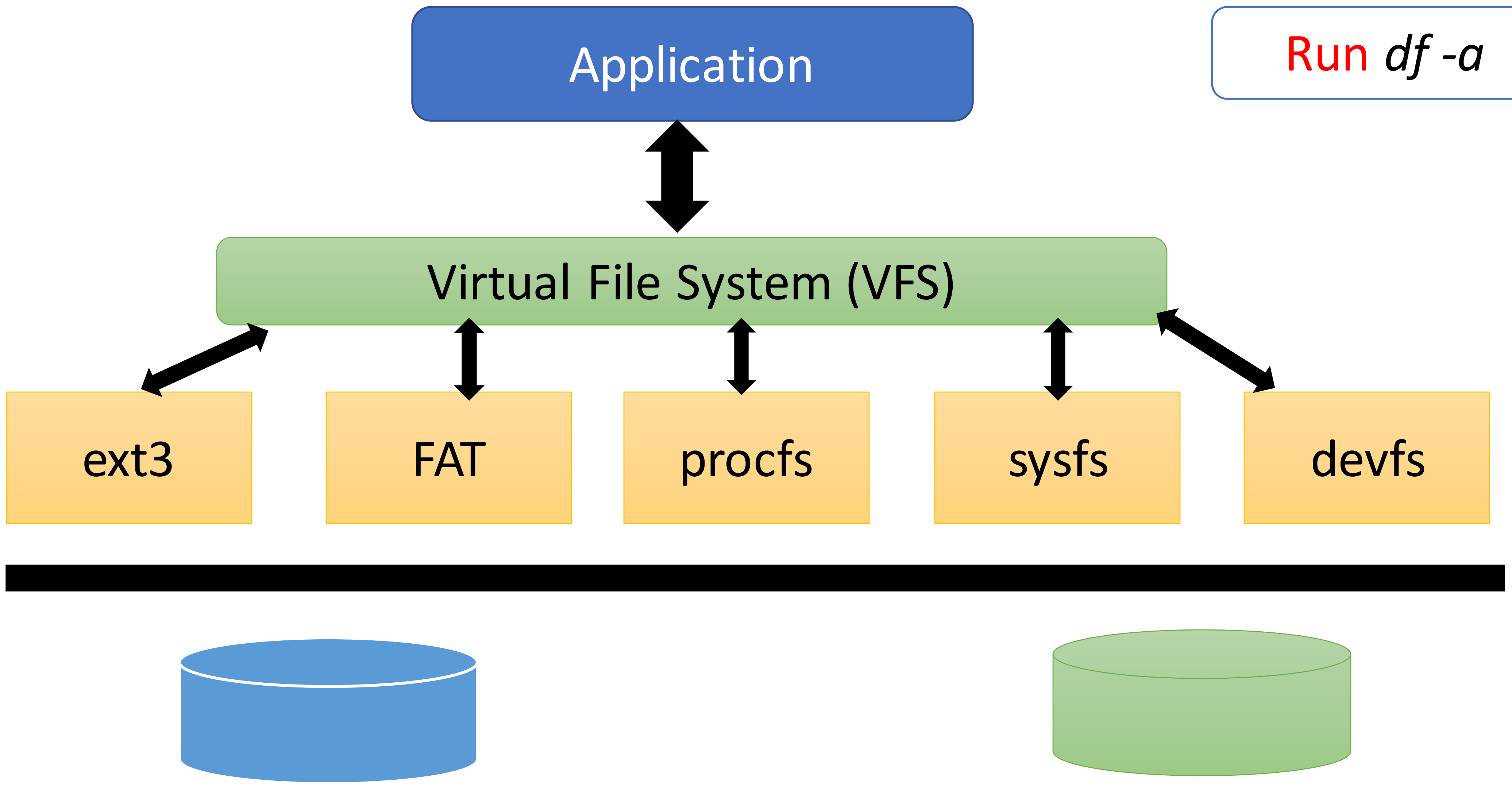
Hard link

Mounted file system. **Map** the root directory of the mounted file system to a directory in the parent file system. *mount* command

File System



Linux VFS (Virtual File System)



Run *df -a*

struct inode

VFS inode

/include/linux/fs.h

```
struct inode {
        struct super_block       *i_sb;
        unsigned long            i_ino;
        struct address_space    *i_mapping;
        dev_t                    i_rdev;
        blkcnt_t                 i_blocks;
        loff_t                   i_size;
        umode_t   i_mode; kuid_t   i_uid; kgid_t   i_gid;
}
```

unique inode number

pointer to the page cache

device hosting the file system

size of the file

regular file, dir, character device, block device, FIFO pipe, sym link, socket

user id and group id

ext4 (/fs/ext4)

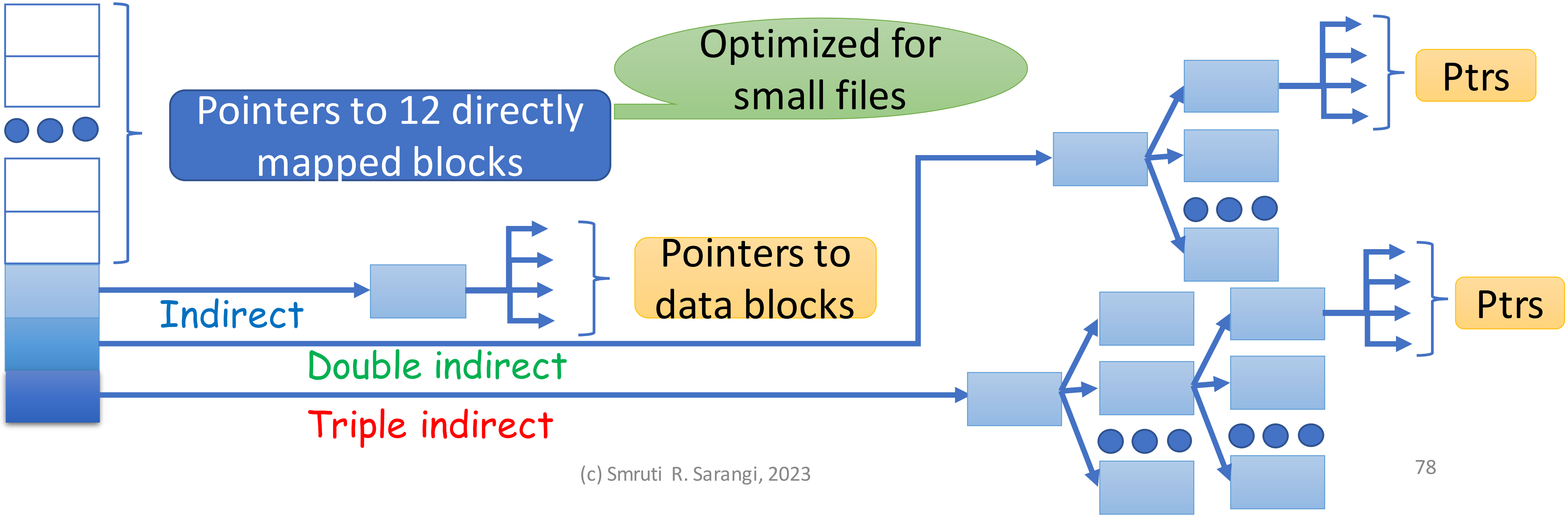
Linux and Android's default filesystem

Filesystems typically declare their own inode structures. Most important structure:

struct ext4_inode

Backward compatibility mode

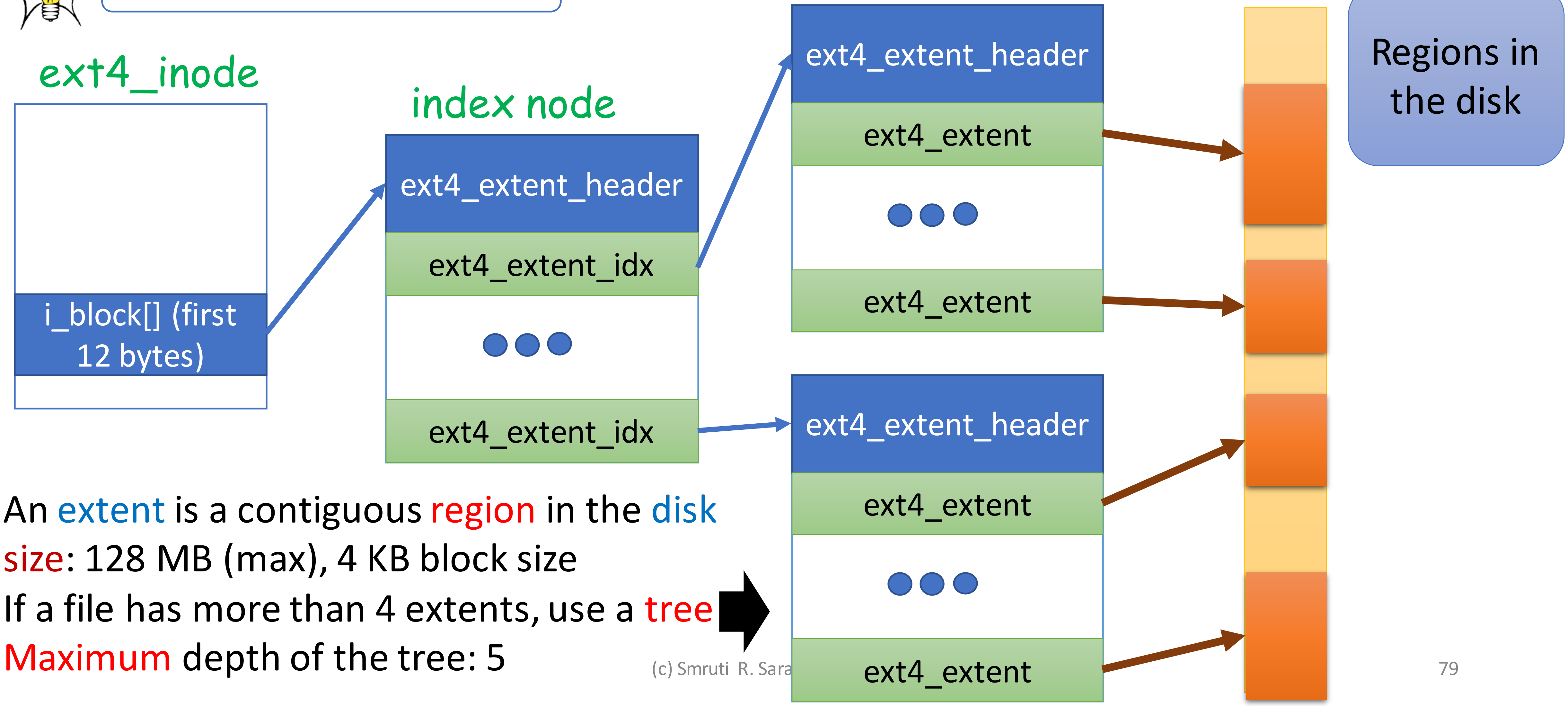
uint32 i_block [EXT4_N_BLOCKS] (list of logical block pointers)



ext4 inode based on *extents* (scalable mode)

/fs/ext4/extents.c

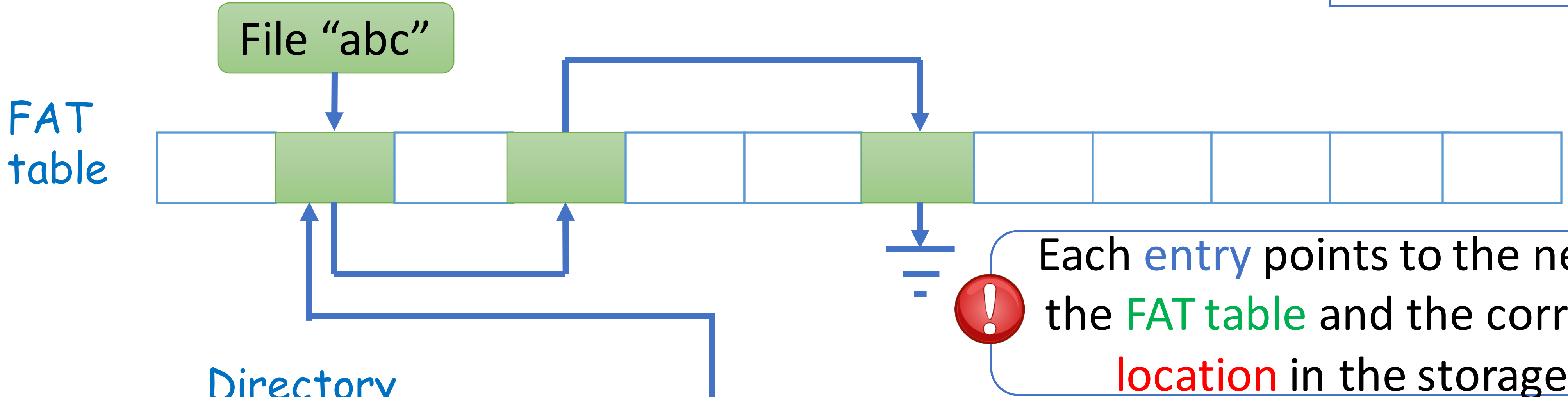
Use the idea of folios



- An extent is a contiguous region in the disk size: 128 MB (max), 4 KB block size
- If a file has more than 4 extents, use a tree
- Maximum depth of the tree: 5

exFAT (Extended File Allocation Table)

/fs/fat/fat.h



Each entry points to the next entry in the FAT table and the corresponding location in the storage device

Name	attr	FAT table entry
abc	…	…

It can contain sub-directories also. Each sub-directory is represented as a sequence of data blocks. Each block contains a linear list of rows.

exfat_dentry

exFAT: Support for much larger file and filesystem sizes. Hash-based file name lookup, pre-allocation.

dentry

VFS structure

/include/linux/dcache.h

```
struct dentry {

        struct hlist_bl_node      d_hash;
        struct dentry             *d_parent;
        struct qstr               d_name;
        struct inode              *d_inode;

        struct list_head          d_child;
        struct list_head          d_subdirs;

}
```

hash collision chain

map a filename to an inode

list of siblings and children

Embed a B-tree in an array: create dummy internal nodes and also keep some unoccupied space

ext_4_direntry: arrange as a linear list in each dir. block

inode num.

length of the file name

file type

file name

struct dx_root and *struct dx_node:* 3-level hashtable

Put dummy entries in the data blocks (corresp. to directory entries). Use them to realize a 3-level B-tree that is indexed by the hash of the file name. The last level is organized as a linear list.

Two Basic Security Concepts

Access list for a file

All the users and groups that can access the file and the particular rights that they will have: read, write, execute, etc.

Capability list of a user or a group of users

The rights on files enjoyed by all users and groups. Very basic support in Linux. Read the documentation: man 7 capabilities

Page Cache

/include/linux/fs.h

```
struct address_space {
    struct inode          *host;
    struct xarray         i_pages;


    struct rb_root_cached   i_mmap;
    unsigned long           nrpages;

    const struct address_space_operations *a_ops;

    struct list_head      private_list;
    void                  *private_data;
}
```

host inode

Pointers to cached pages arranged as a radix tree

red-black tree that stores all mapped vmas

#pages

functions to bring in and evict folios

private data to be used by the owner

- Most file operations happen on the page that is stored in-memory
- i_mmap stores vmas of all the processes that share the address_space (rmap pages of the file)

Named Pipes (special file system)

Create a named pipe

Note the 'p'

Wait till the pipe is written to

```
DELL@Desktop-home2 ~
$ mkfifo mypipe

DELL@Desktop-home2 ~
$ file mypipe
mypipe: fifo (named pipe)

DELL@Desktop-home2 ~
$ ls -al mypipe
prw-rw-rw- 1 DELL None 0 Apr 23 09:38
mypipe

DELL@Desktop-home2 ~
$ tail -f mypipe
I love my OS course
```

```
DELL@Desktop-home2 ~
$ echo "I love my OS
course" > mypipe
```

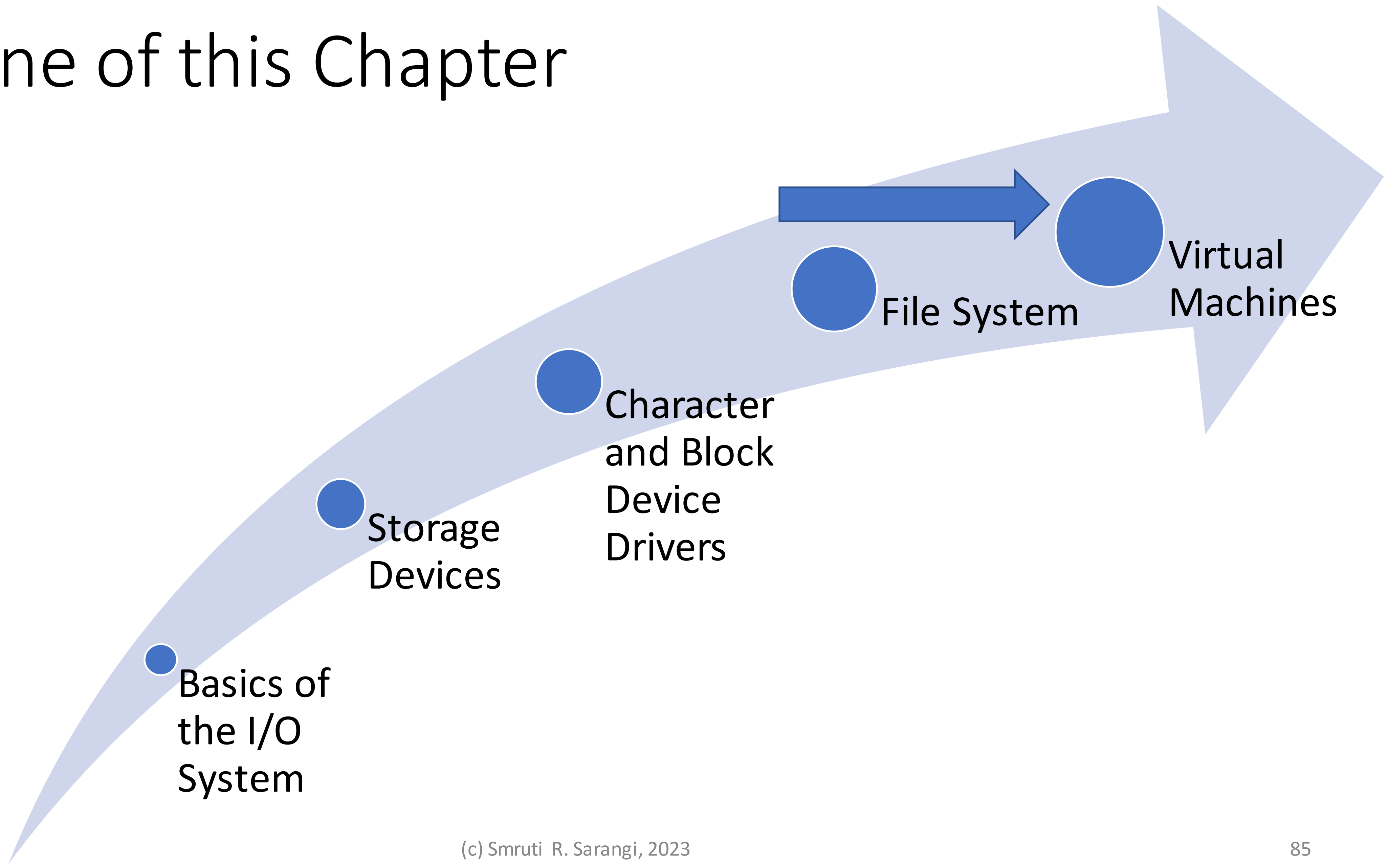
Write to the pipe

Unnamed pipes

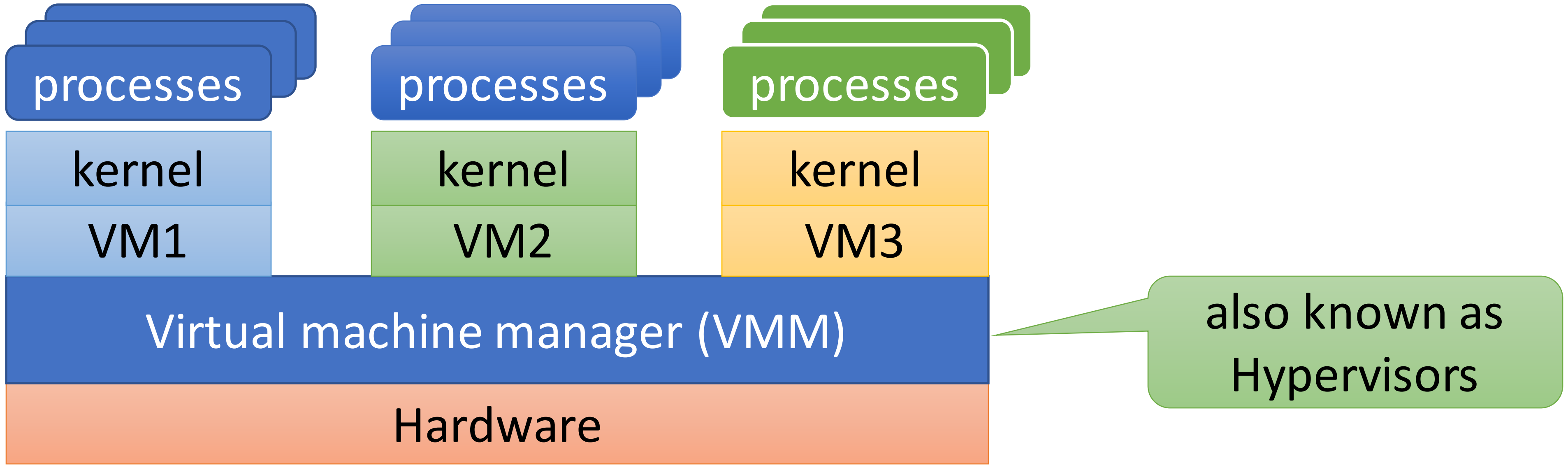
int pipe (int fd[2]);

Write to fd[1], read from fd[0]: they are *int* file descriptors

Outline of this Chapter



Run an OS as an Application



Types of Hypervisors

Type	Explanation	Examples
Type 0	Hardware or Firmware-based VMM. Minimal software support	IBM LPAR, Oracle LDOM
Type 1	An OS that allows us to run other guest OSes. Provides native support and maps guest addresses to physical memory directly.	Linux KVM, VMWare ESX, Citrix XenServer
Type 2	Run a guest OS as a regular application	VMPlayer, VirtualBox
Para-virtualization	The code of the guest OS is changed (unlike the previous cases) and made VM aware	Xen

Advantages of VMs in Cloud Environments

Create custom environments

- Create custom **environments** (OS + libraries + software) and run them on any machine on a cloud.

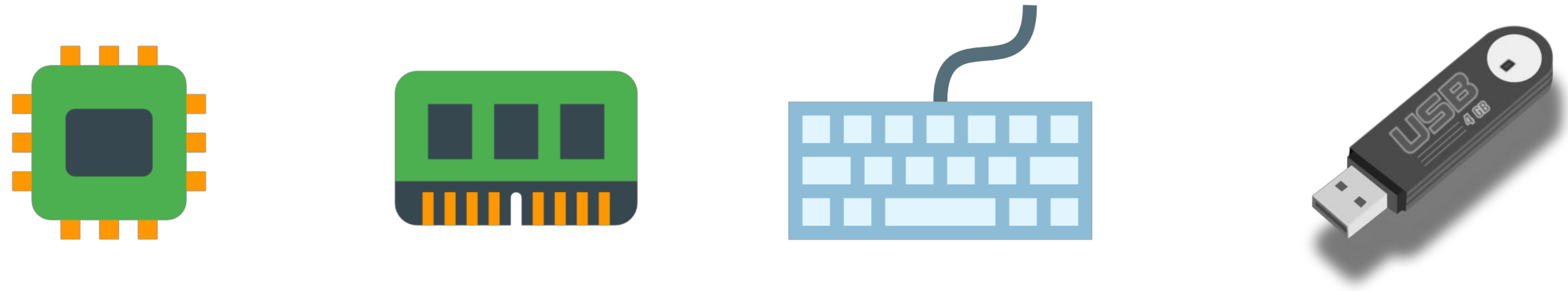
Virtualize the CPU, network, file system, memory, and hardware

- Transparently migrate the VM across different types of machines

Server consolidation

- Live migration of VMs, efficiently heterogeneous server resources

Virtualization



CPU

Memory

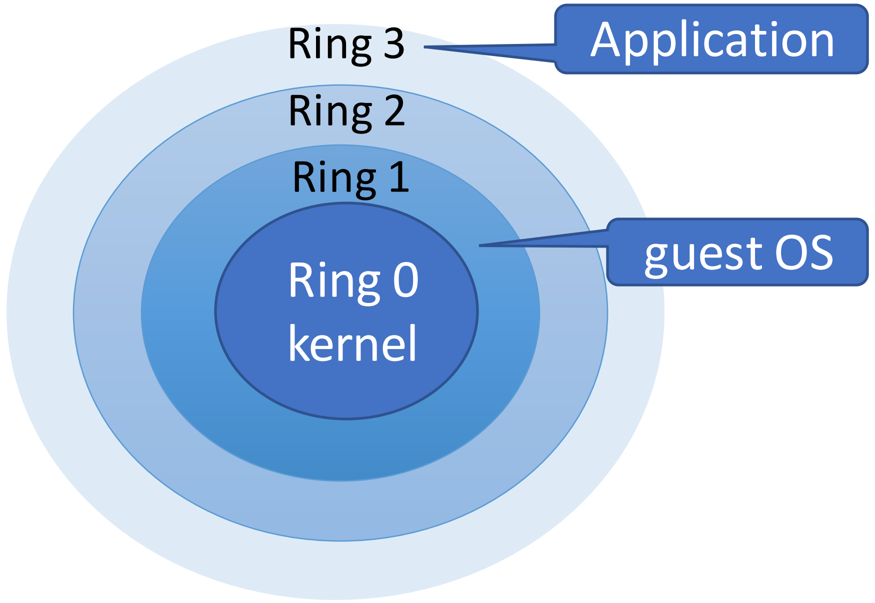
i/o

storage

The guest OS has no idea that it is running on a hypervisor.

The OS needs to think that it exclusively owns these devices

Virtualize the CPU



Regular instructions

Privileged instructions

Trap and emulate: The guest OS runs with user privileges. Invoking a privileged inst. leads to a trap. The hypervisor catches it and successfully emulates the instruction.

Instructions that execute differently based on the privilege level

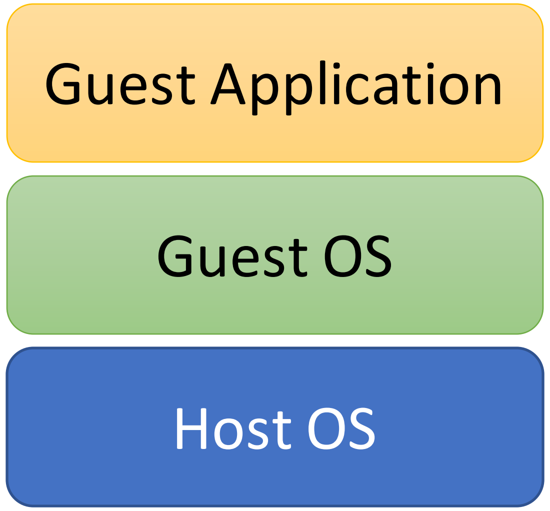
Binary translation

Binary Translation

- The hypervisor follows a read-ahead scheme.
  - Whenever a new code page is accessed, a fault is generated (either the TLB entry is missing or the page is marked non-accessible)
  - The hypervisor reads the page and a few more subsequent pages
  - It scans all the instructions.
  - If there is a sensitive instruction (changes its behavior based on the privilege level), replace it with an alternative sequence
  - Make it generate a trap or make the behavior the same (guest & kernel)

What about memory?

Type 1 Hypervisors



Type	Explanation
GVA	Guest virtual addr.
GPA	Guest physical addr.
HVA	Host virtual addr.
HPA	Host physical addr.

TLB entry

GVA → HPA

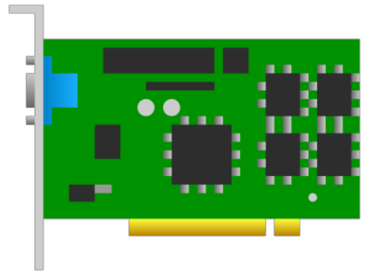
Nested paging

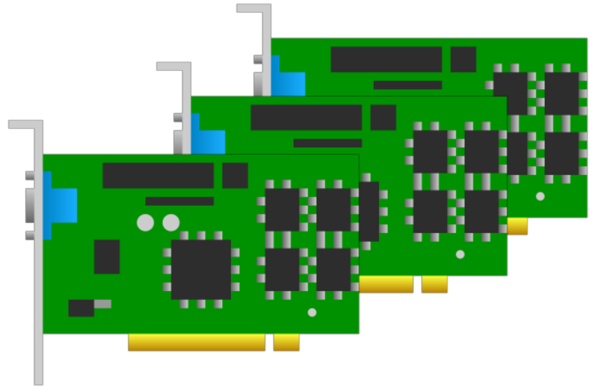
1. TLB miss → walk the guest and host page tables. (slow)
2. No need to track guest page table modifications.
3. VMM maintains only 1 page table.

Shadow paging

1. The VMM builds a GVA → HPA page table (one per guest process).
2. Keeps the shadow page table consistent with both page tables. TLB Miss: Access just the shadow page table (*fast*)
3. Track every guest page table update

I/O Virtualization





One network card appears as several virtual network cards (one for each VM)

Emulation: The hypervisor traps every I/O instruction and memory-mapped ld/st instruction (make pages non-accessible). Follows the trap-and-emulate method. Every guest OS uses emulated devices. [slow]

With H/W support (PCI-X devices): Use the SR-IOV and MR-IOV protocols. A single device exposes separate buffers (request queues), interrupts and DMA streams to each VM. MR-IOV allows sharing the resources with multiple computers.

Paravirtualization

Why not modify the guest operating system to make it VM aware?

- Xen [1] was the first popular para-virtualized hypervisor. Guest OSes communicate via system calls (also known as hypercalls).
- Avoid instructions that change their behavior based on the privilege level
- Ensure that either all privileged instructions either trap in VM mode or are replaced with non-privileged variants that do the same job (may invoke VMM routines)
- Use "fast exception handlers" that are directly registered in the CPU's exception and system call table. Page fault handlers need to be there with Xen (HW reasons 🤔)
- Guest OSes can directly write into the CR3 register and change the current page table address (mapping: GVA → HPA). Updates to the page table need to go through the hypervisor. Xen maintains a shadow page table for each process and it updates it when the guest OS lets it know.
- Interrupts are sent as lightweight events to the guest OS (similar to signals)
- I/O: Create a bounded queue (request buffer) between the guest OS and Xen (aka I/O ring because it is a circular queue).

HW-Assisted Virtualization

Intel VT-x

AMD-V

Event	Action
Privileged instruction execution	Most of the time it is allowed like *syscall enter* and *exit* because in this case, VMs run in Ring 0 (albeit in non-root mode or VM mode). Some I/O instructions necessitate a VM exit. The device needs to be emulated.
Memory accesses	Use shadow page tables that are directly stored in HW (fast path). The CPU walks both the guest and host page tables on a TLB miss and updates the shadow page table (if there has been an update to either page table, slow path).
I/O	Use the SR-IOV and MR-IOV protocols
Storage	The hypervisor maintains a guest-to-host block table.

Bibliography

1. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

srsarangi@cse.iitd.ac.in