Machine-Level Programming: Advanced Topics

Prof. Hyuk-Yoon Kwon
https://sites.google.com/view/seoultech-bigdata

Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

OS → managing memory by each process

provide same address range for memory for each process

⇒ virtual memory

Cannot access physical memory directly ⇒ OS manage physical memory &

show conceptual memory layout to the process } ⇒ process can access the address of memory provided by OS & CS

x86-64 Linux Memory Layout

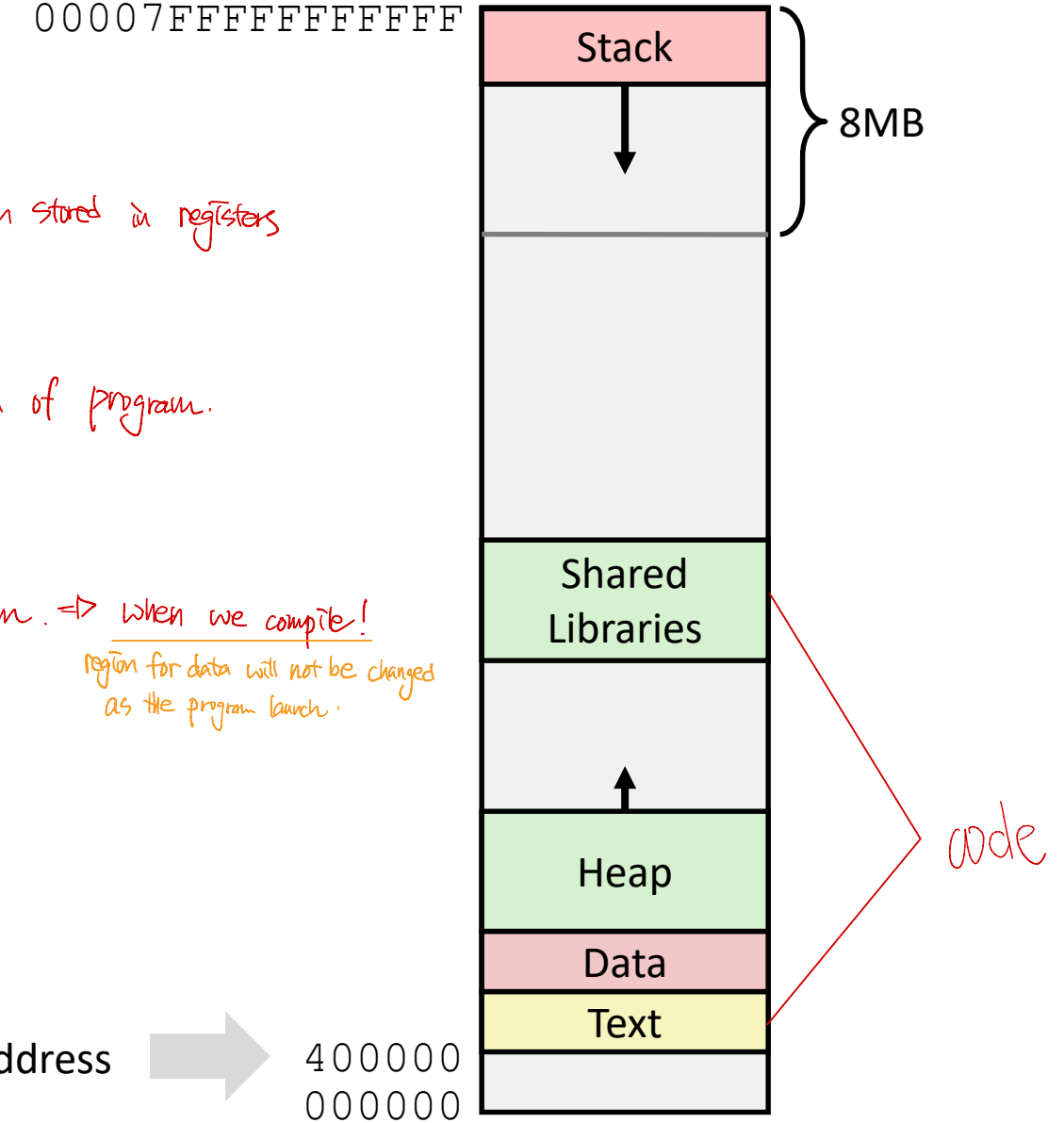
~ supported by OS, all process have the same address range for the memory

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables (return address, values ...) have been stored in registers
- **Heap**
  - Dynamically allocated as needed (not determined in advance) ~ result of execution of program.
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants

Store data

determine the value of data before running the program. => when we compile!
region for data will not be changed as the program launch.

- **Text / Shared Libraries** (source code itself; referenced by program.) ~ related to source codes
  - Executable machine instructions
  - Read-only



Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
  /* Some print statements ... */
}
```

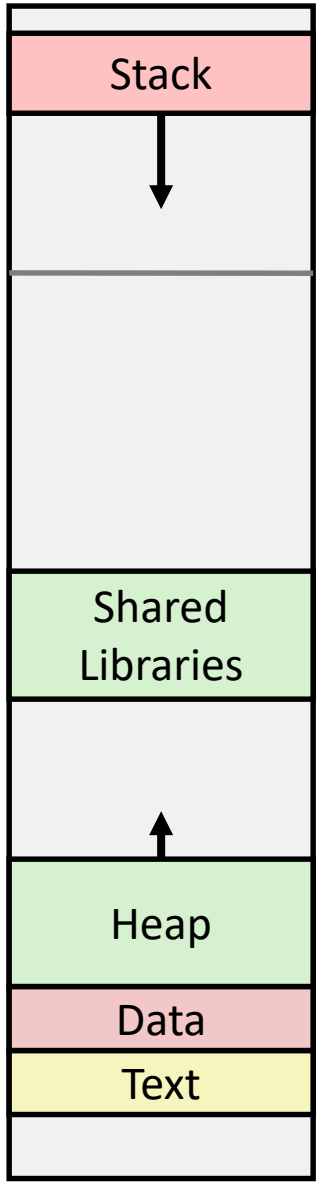
in data

in text

in Stack

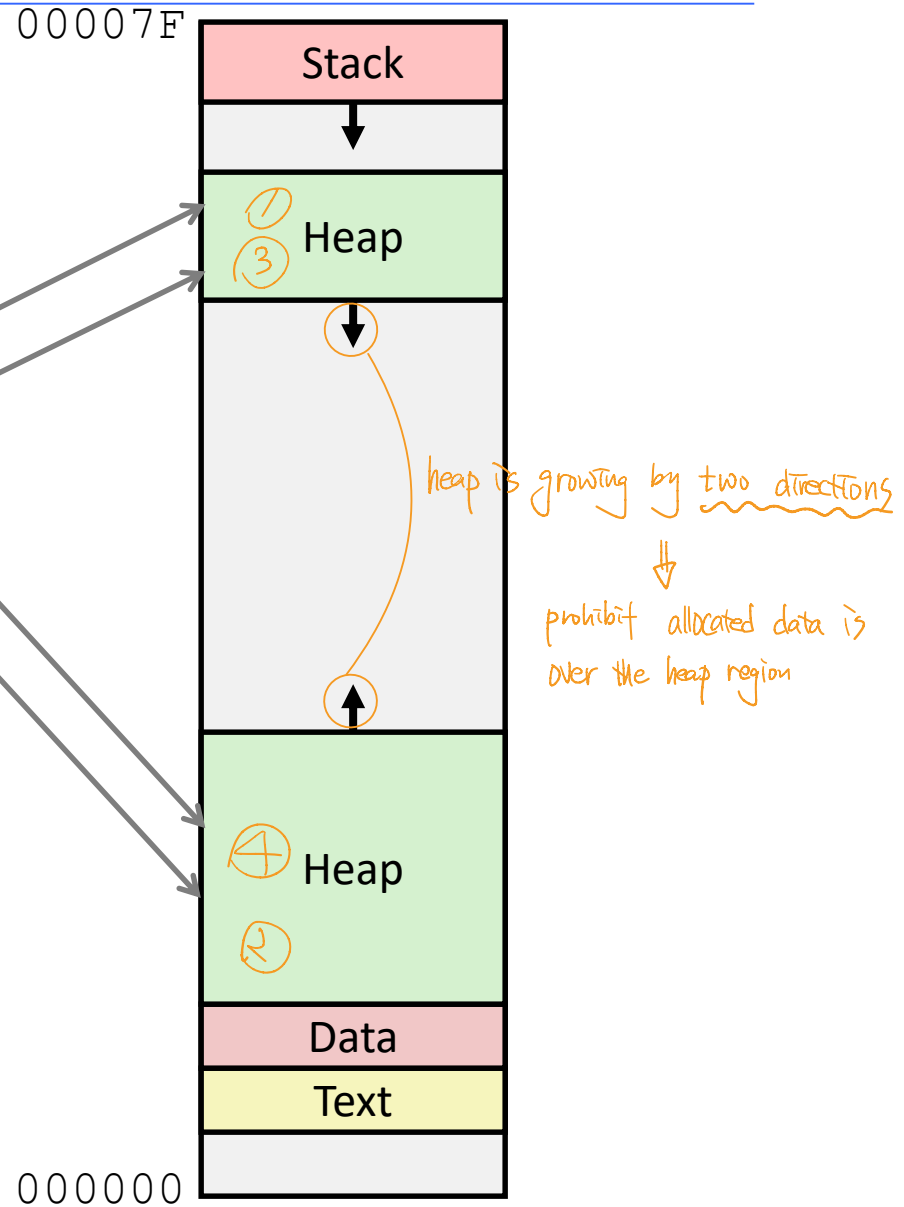
in heap

Where does everything go?



x86-64 Example Addresses

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x0000000008359d120
p2	0x0000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Practice: Memory Allocation Example

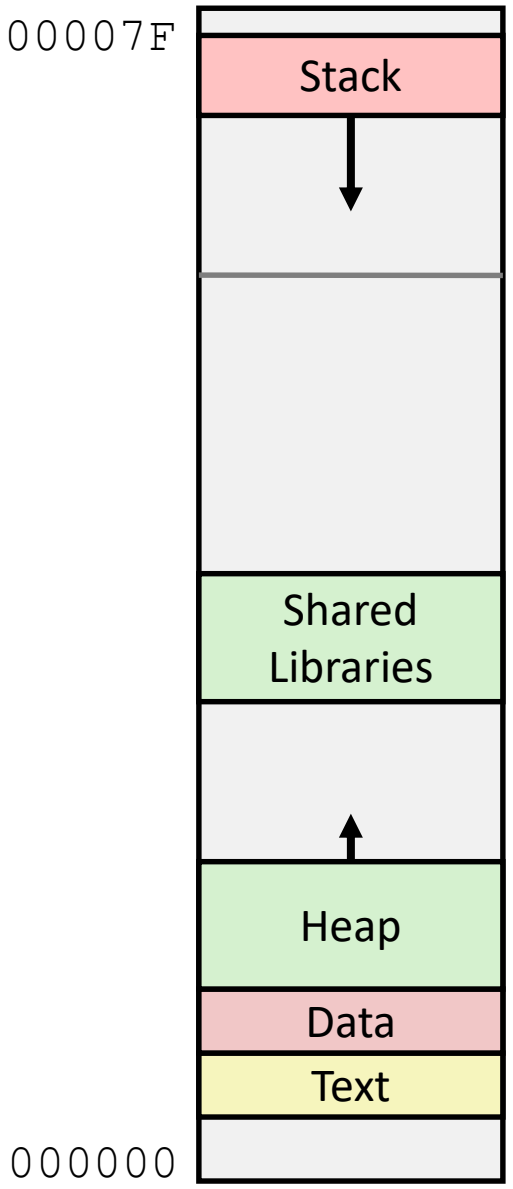
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

Where does everything go?



local	Stack	?
p1	heap	?
p3	heap	?
p4	heap	?
p2	heap	?
big_array	data	?
huge_array	data	?
main()	text	?
useless()	text	?

Today

- Memory Layout
- **Buffer Overflow** when allocated space are overflowed.
  - Vulnerability
  - Protection

Recall: Memory Referencing Bug Example

- Result is system specific

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)   ->    3.14
fun(1)   ->    3.14
fun(2)   ->    3.1399998664856
fun(3)   ->    2.00000061035156
fun(4)   ->    3.14
fun(6)   ->    Segmentation fault
```

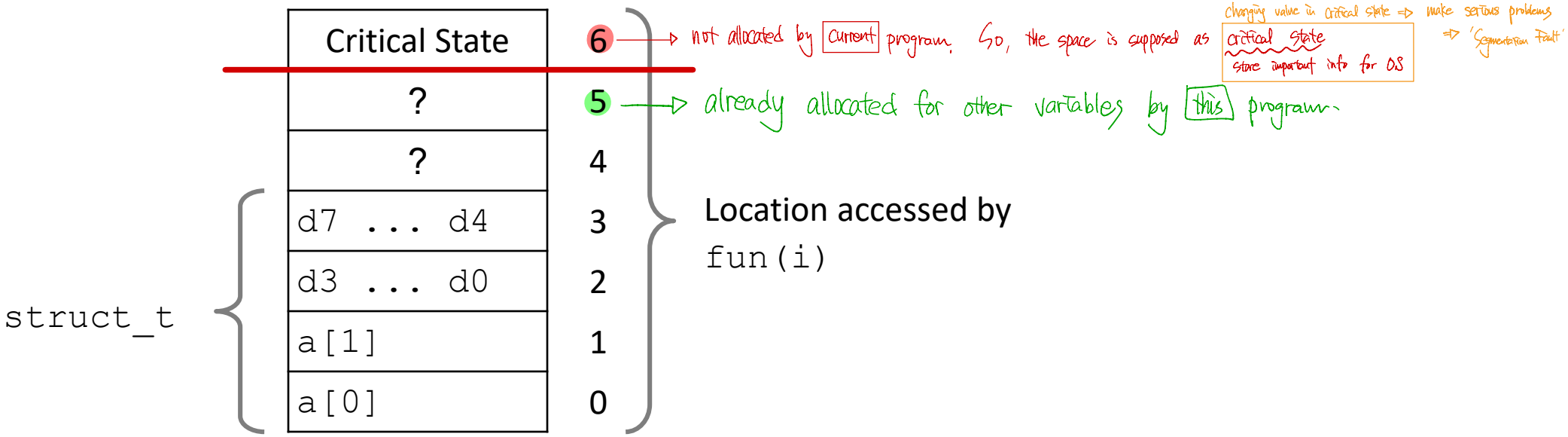
Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

overwrite

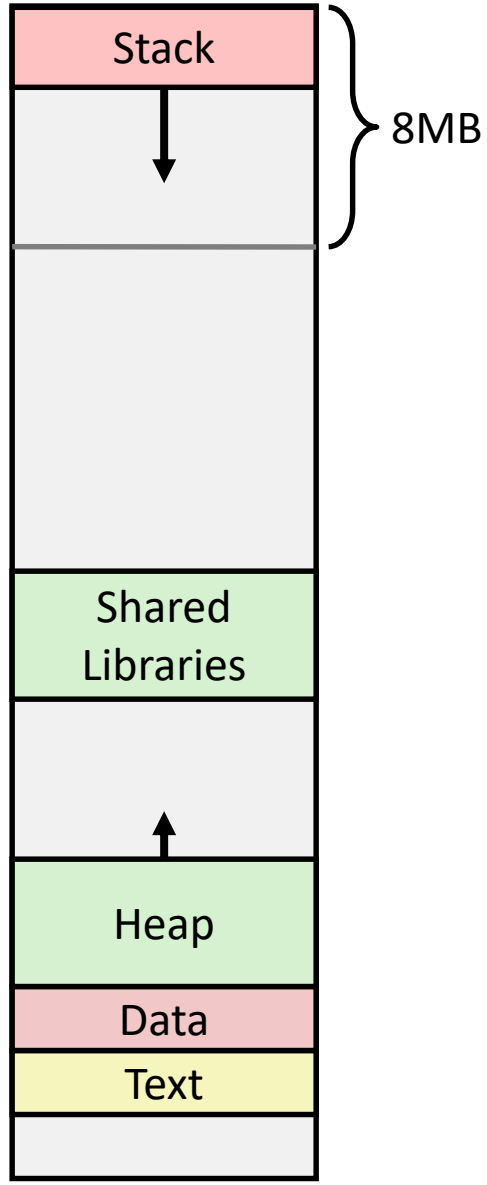
```
fun(0)   ->   3.14
fun(1)   ->   3.14
fun(2)   ->   3.1399998664856
fun(3)   ->   2.00000061035156
fun(4)   ->   3.14
fun(6)   ->   Segmentation fault
```

Explanation:



Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing



String Library Code

Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

can occur buffer overflow!

if input string exceed the size of allocated memory
⇒ Store the string into other region in the memory
OVERWRITE! ⋯ security vulnerability

- No way to specify limit on number of characters to read

Similar problems with other library functions

- **`strcpy`, `strcat`**: Copy strings of arbitrary length
- **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

Buffer Overflow Disassembly

echo:

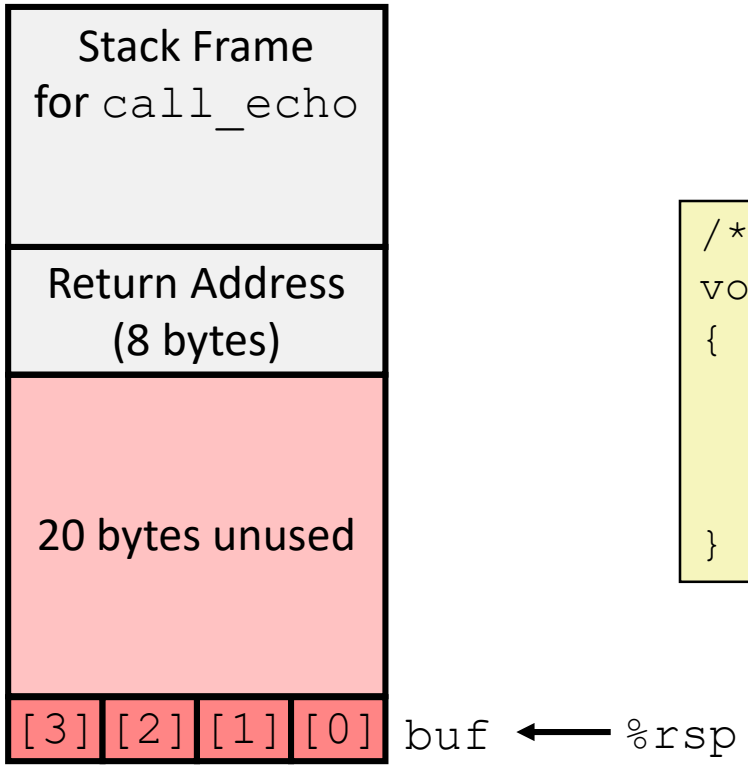
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18           sub    $0x18,%rsp
 4006d3:  48 89 e7              mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
 4006db:  48 89 e7              mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:  48 83 c4 18           add    $0x18,%rsp
 4006e7:  c3                    retq
```

call_echo:

```
 4006e8:  48 83 ec 08           sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00        mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff        callq  4006cf <echo>
 4006f6:  48 83 c4 08           add    $0x8,%rsp
 4006fa:  c3                    retq
```

Buffer Overflow Stack

Before call to gets

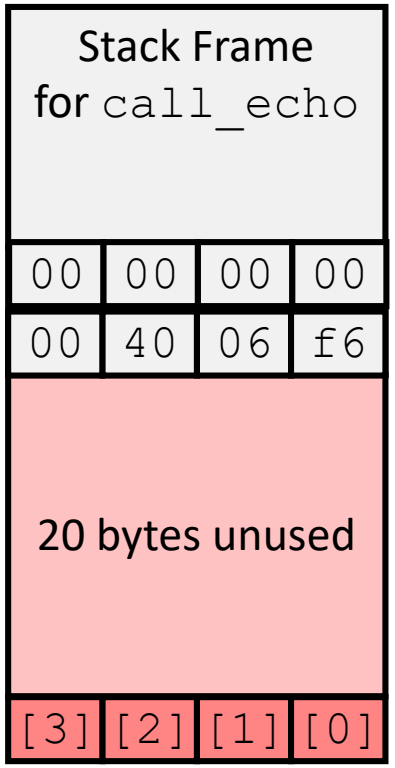



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

Buffer Overflow Stack Example

Before call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

24 bytes including '\0'

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

26 bytes including '\0'

Overflowed buffer and corrupted return pointer

Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
0123456789012345678901234
```

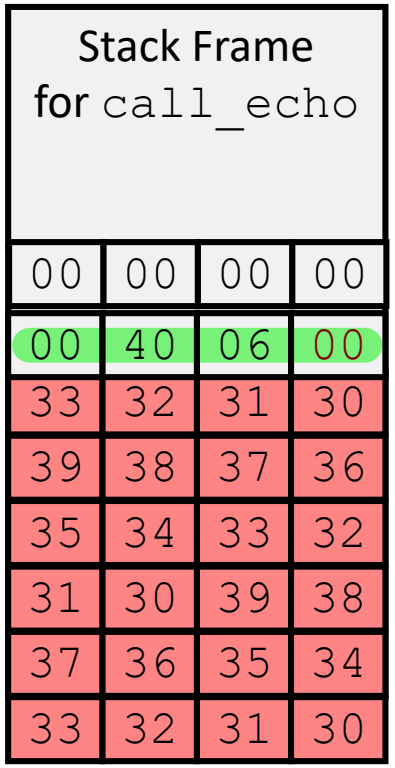
25 bytes including '\0'

→ just luck

Overflowed buffer, corrupted return pointer, but program seems to work!

Buffer Overflow Stack Example #3 Explained

After call to gets



register_tm_clones:

```
  . . .
400600:  mov     %rsp,%rbp
400603:  mov     %rax,%rdx
400606:  shr     $0x3f,%rdx
40060a:  add     %rdx,%rax
40060d:  sar     %rax
400610:  jne     400614
400612:  pop     %rbp
400613:  retq
```

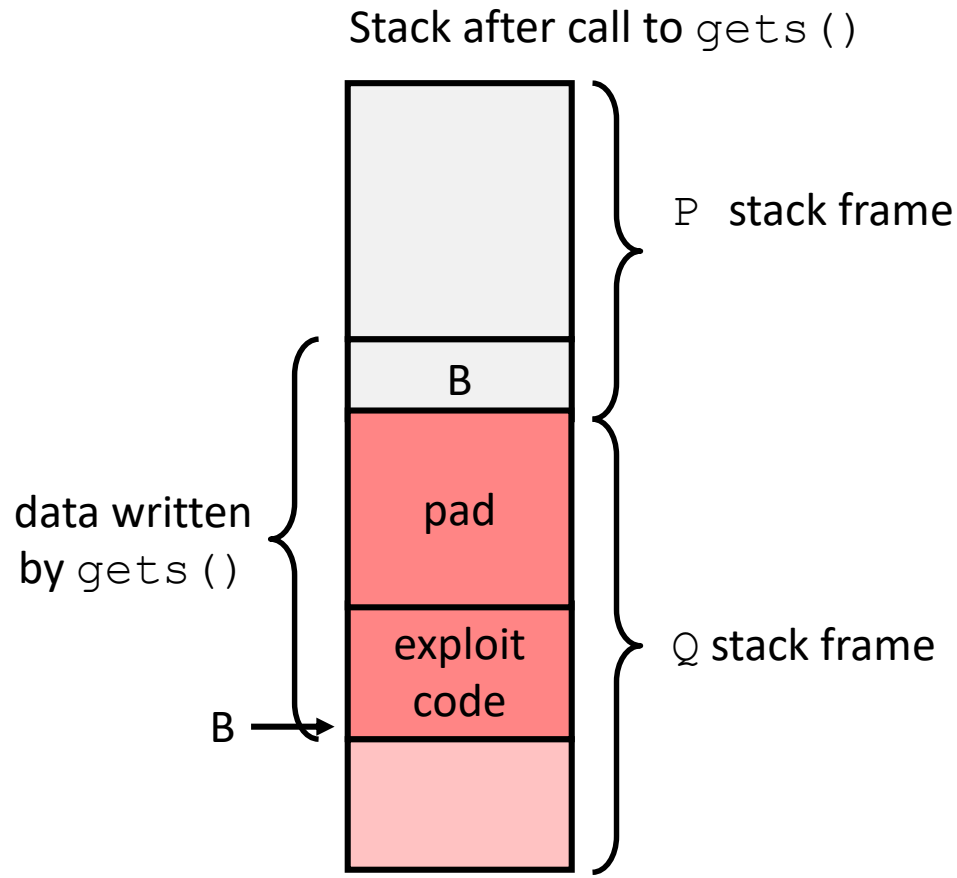
"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

Practice: Memory Referencing Bug Example

- Result is system specific

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)   ->    3.14
fun(1)   ->    3.14
fun(2)   ->    3.1399998664856
fun(3)   ->    2.00000061035156
fun(4)   ->    3.14
fun(6)   ->    Segmentation fault
```

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***

- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more

Example: the original Internet worm (1988)

- **Exploited a few vulnerabilities to spread**
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger "exploit-code padding new-return-address"`**
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- **Once on a machine, scanned for other machines to attack**
  - invaded ~6000 computers in hours (10% of the Internet ☺)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each…

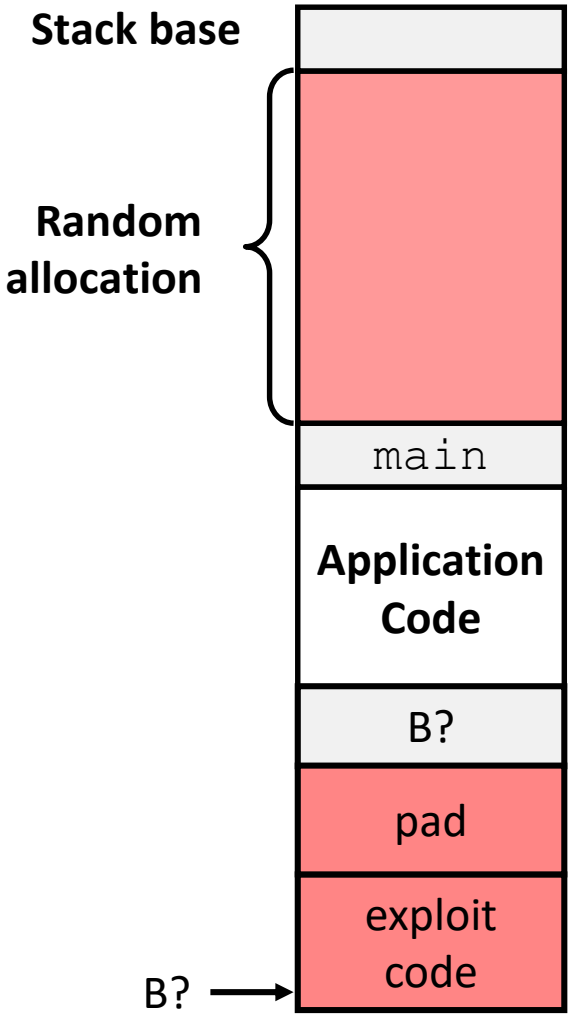
1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**

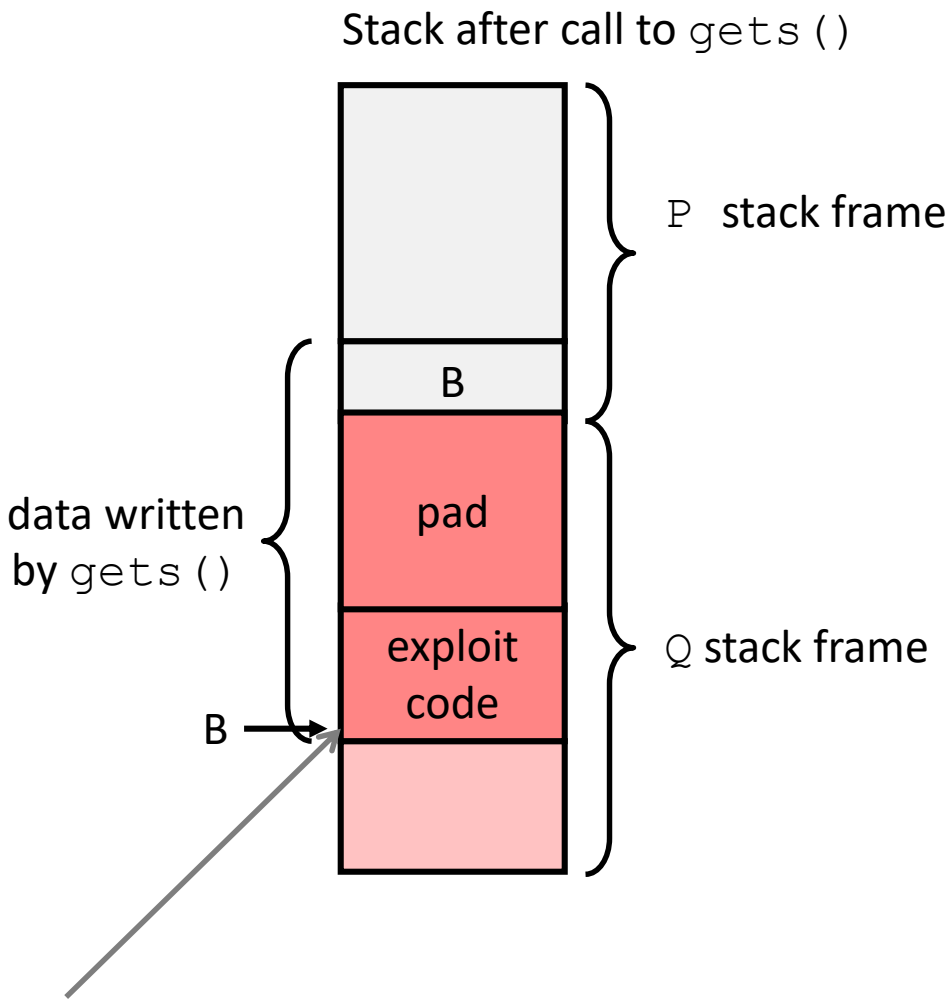
2. System-Level Protections can help

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code



2. System-Level Protections can help

- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory a s either “read-only” or “writeable”
    - Can execute anything readable
  - X86-64 added explicit “execute” permission
  - Stack marked as non-executable



3. Stack Canaries can help

Idea

- Place special value (“canary”) on stack just beyond buffer
- Check for corruption before exiting function

GCC Implementation

- `-fstack-protector`
- Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Protected Buffer Disassembly

using canary

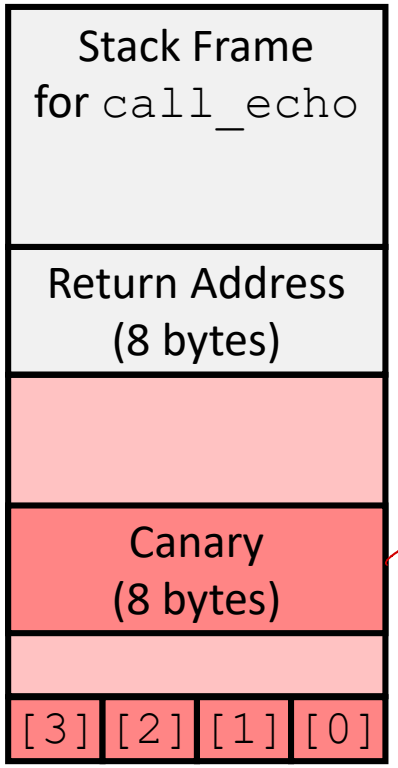
echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

compare canary value before & after executing function.

Setting Up Canary

Before call to gets



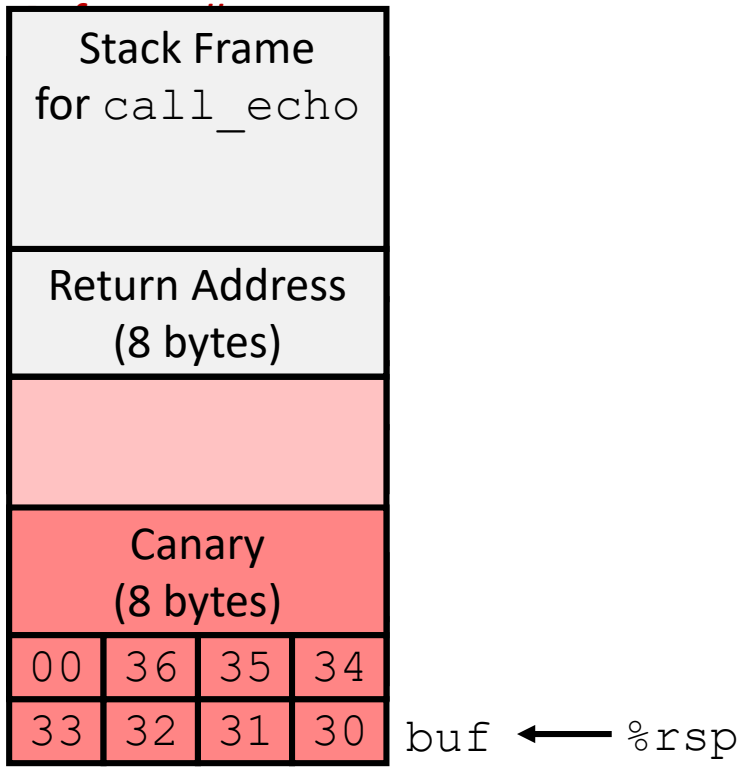

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

middle between return address & actual value in stack
if buffer overflow & overwriting occur, canary must be changed
checking canary before return to origin function!

```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Checking Canary

After call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: *0123456*

```
echo:
    . . .
    movq    8(%rsp), %rax     # Retrieve from
stack
    xorq    %fs:40, %rax      # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
.L6:        . . .
```

Return-Oriented Programming Attacks

Challenge (for hackers)

- Stack randomization makes it hard to predict buffer location
- Marking stack nonexecutable makes it hard to insert binary code

Alternative Strategy

soln

- Use existing code + shared libraries
  - E.g., library code from stdlib

without writing any codes

- String together fragments to achieve overall desired outcome

multiple separate steps to execute codes

- *Does not overcome stack canaries*

Construct program from *gadgets*

by connect multiple gadgets

code blocks

- Sequence of instructions ending in **`ret`**
  - Encoded by single byte **`0xc3`**
- Code positions fixed from run to run
- Code is executable

Gadget Example

```
long ab_plus_c
  (long a, long b, long c)
{
   return a*b + c;
}
```

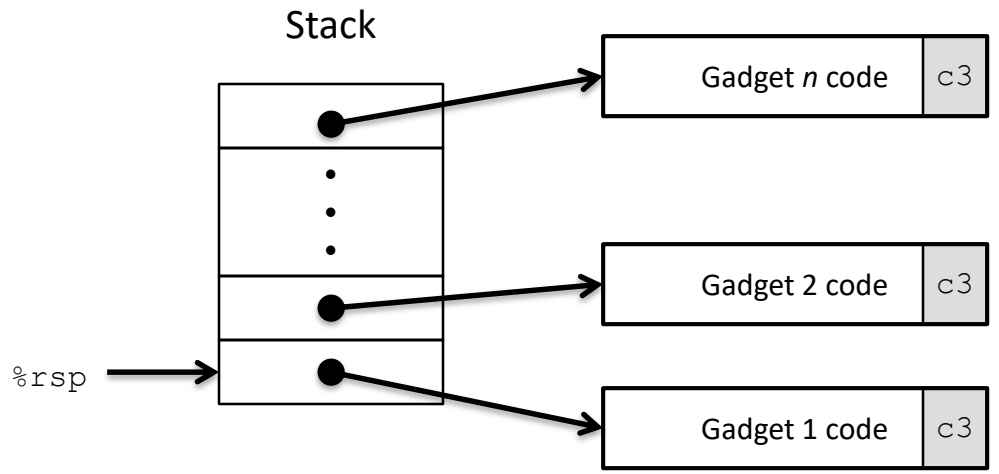
```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

rax ← rdi + rdx

Gadget address = `0x4004d4`

- **Use tail end of existing functions**

ROP Execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

Today

- **Storage technologies and trends**
- Locality of reference
- Caching in the memory hierarchy

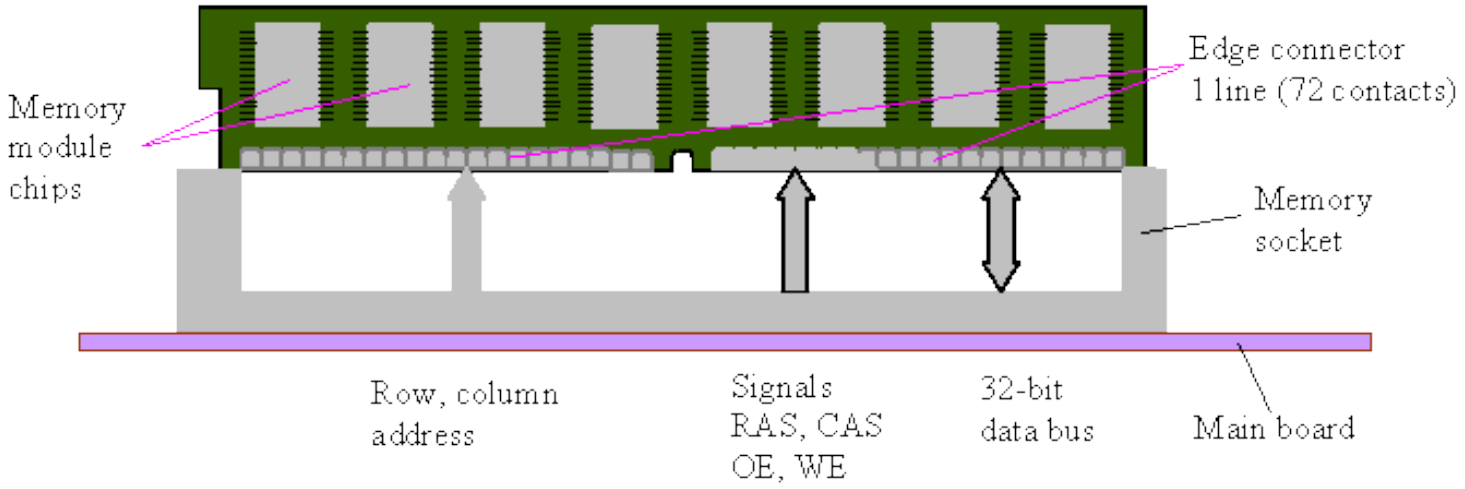
Random-Access Memory (RAM)

Key features

- RAM is traditionally packaged as a chip.
- Basic storage unit is normally a cell (one bit per cell).
- Multiple RAM chips form a memory.

RAM comes in two varieties:

- SRAM (Static RAM)
- DRAM (Dynamic RAM)



SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Cost	Applications
SRAM	4 or 6	1X	No	100x	Cache memories
DRAM	1	10X	Yes	1X	Main memories, frame buffers

Nonvolatile Memories

- **DRAM and SRAM are volatile memories**
  - Lose information if powered off.

like disk!

- **Nonvolatile memories retain value even if powered off**
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase capability

entire erasing

  - Flash memory: EEPROMs. with partial (block-level) erase capability
    - Wears out after about 100,000 erasings

못쓰게 되다.

- **Uses for Nonvolatile Memories**
  - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)

what is it?

PC boot ~ OS boot : check status of HW, change configuration

even if OS has trouble, BIOS/UEFI is always working because they are stored in non-volatile memories!

UEFI

  - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)

SSD

  - Disk caches

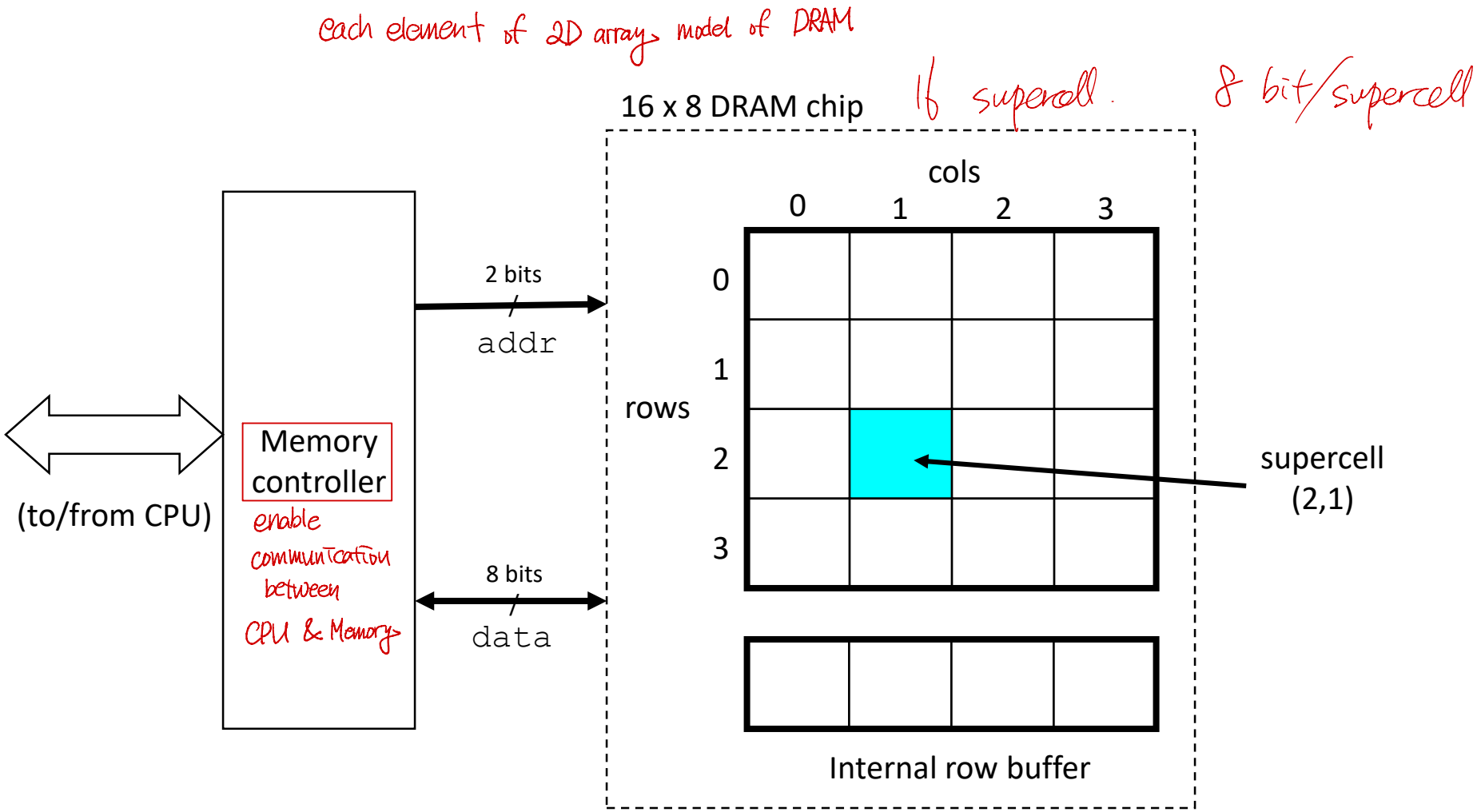
minimizing gap between memory & disk

=> region called disk cache in the disk

store frequently used data => improving performance of accessing disk

Conventional DRAM Organization

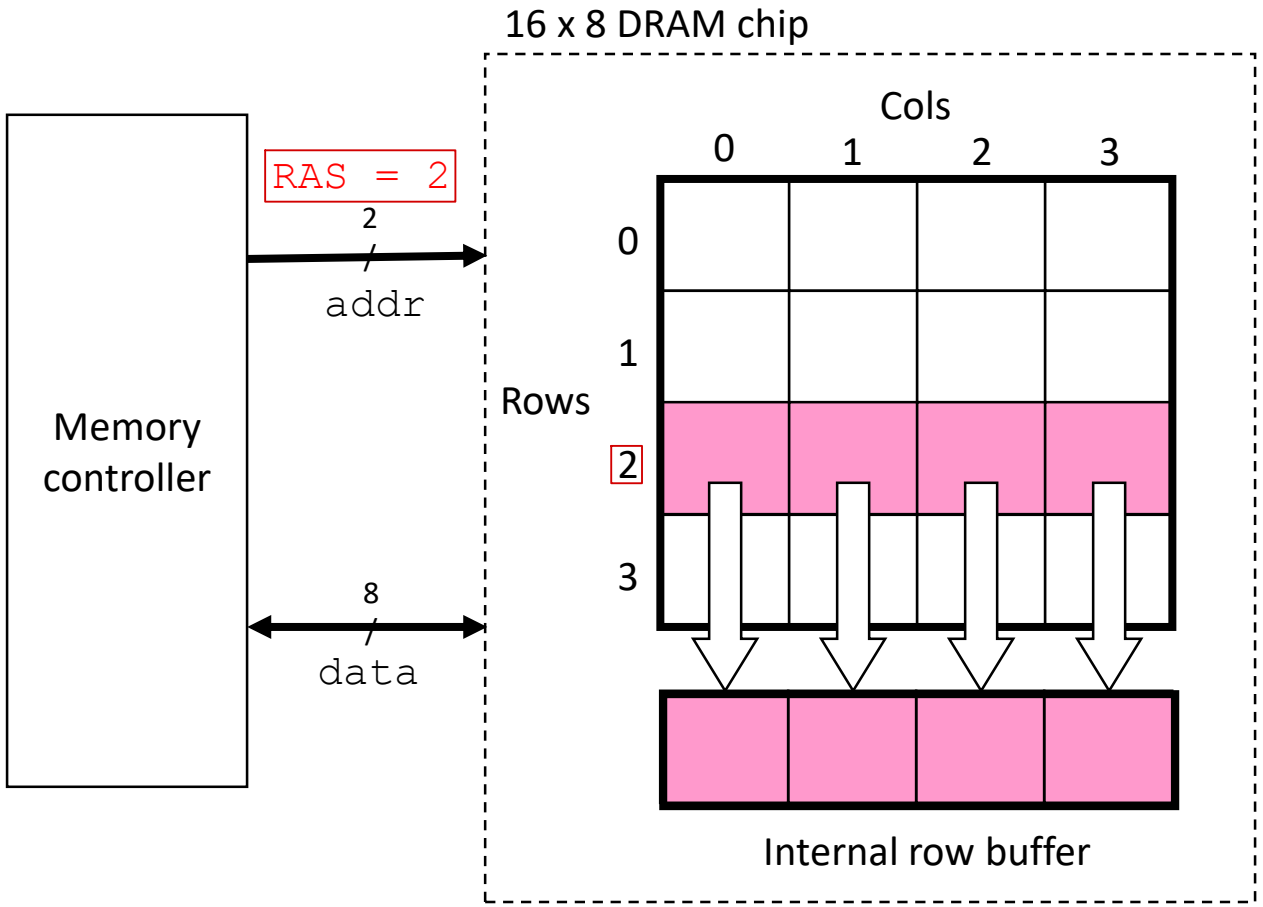
- **d x w DRAM:**
  - dw total bits organized as d supercells of size w bits



Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2.

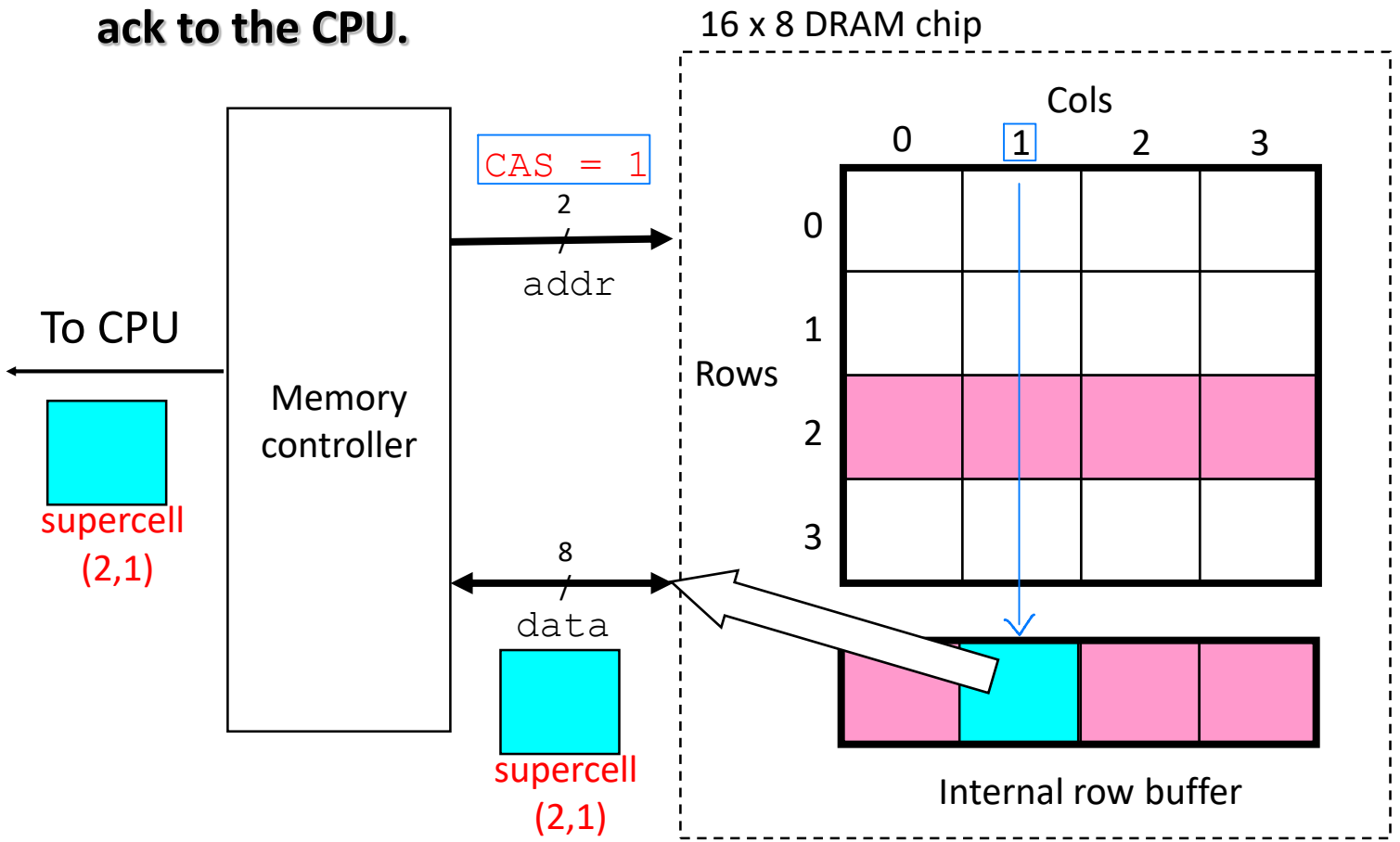
Step 1(b): Row 2 copied from DRAM array to row buffer.



Reading DRAM Supercell (2,1)

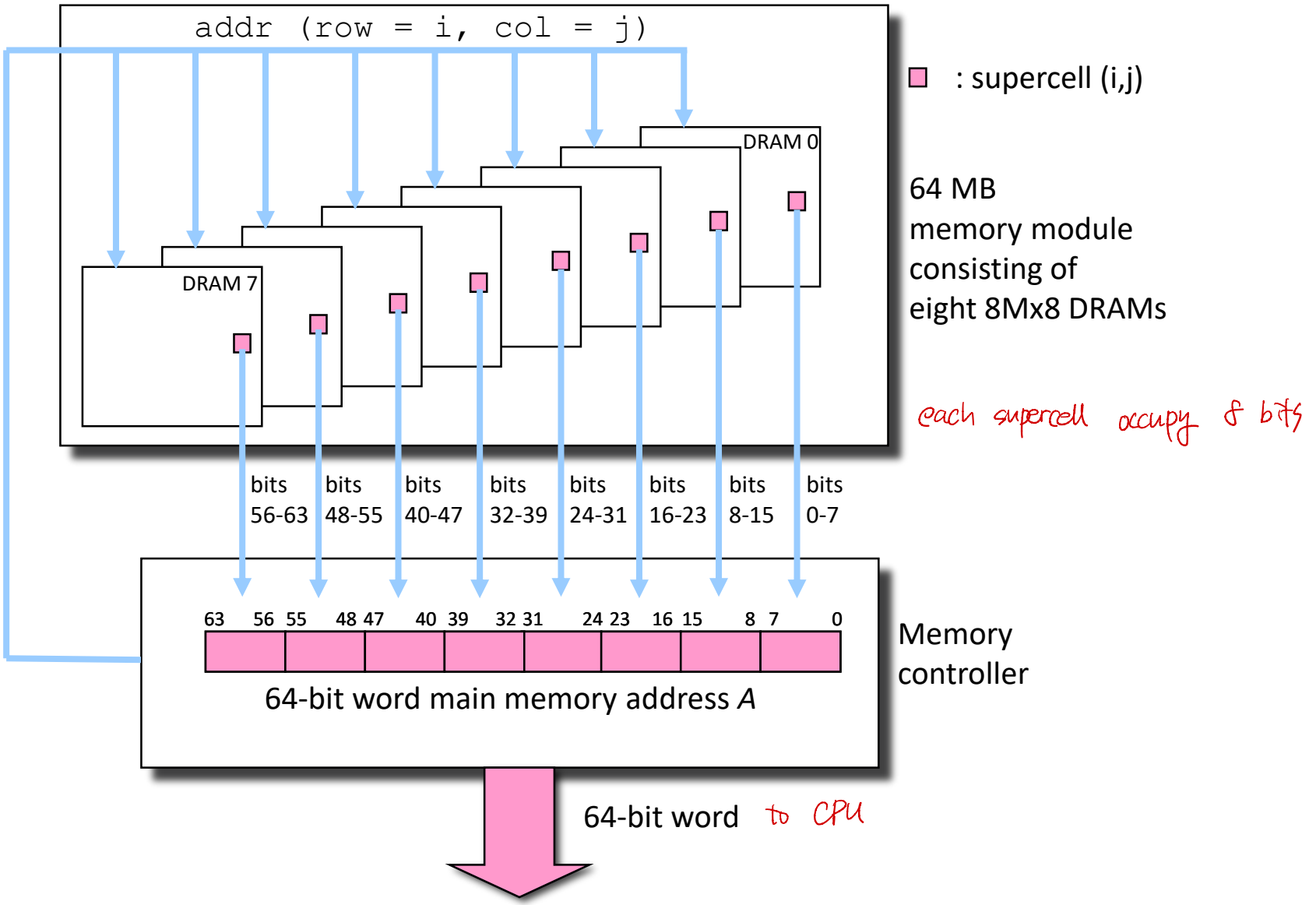
Step 2(a): Column access strobe (CAS) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually b ack to the CPU.



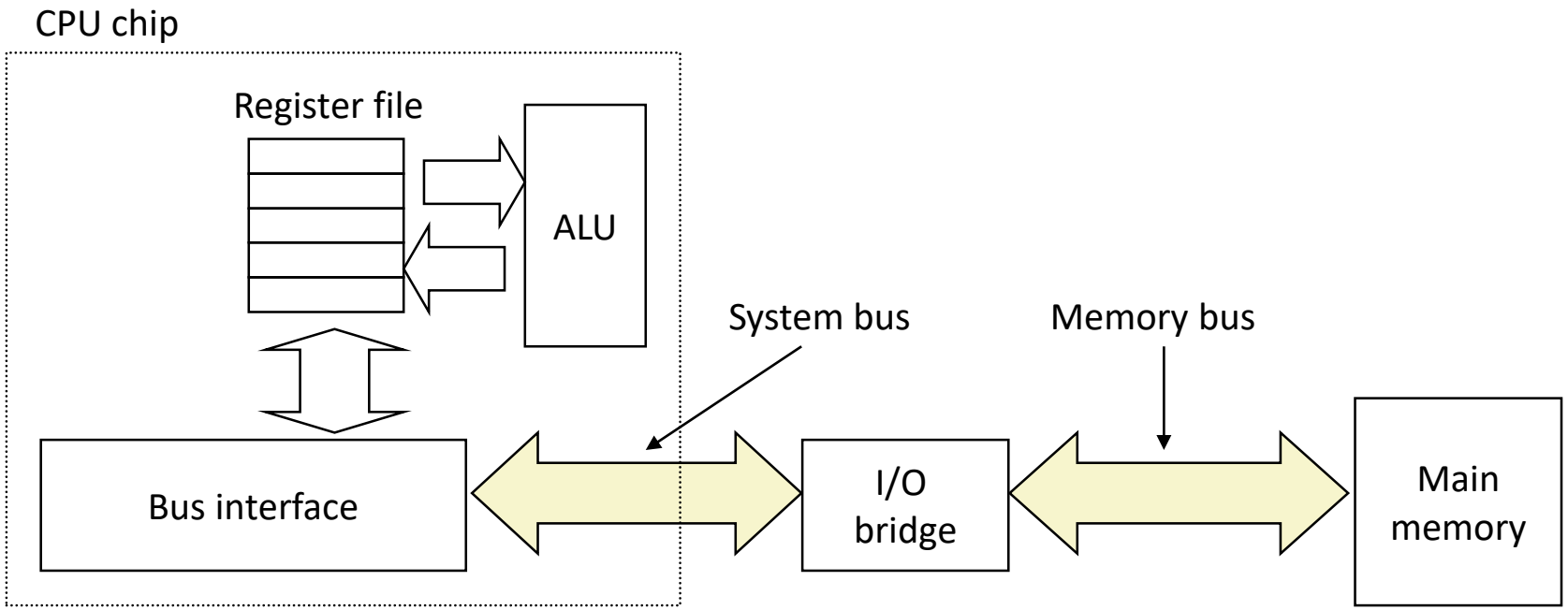
Memory Modules

each module consists of multiple DRAM => access multiple data in parallel time



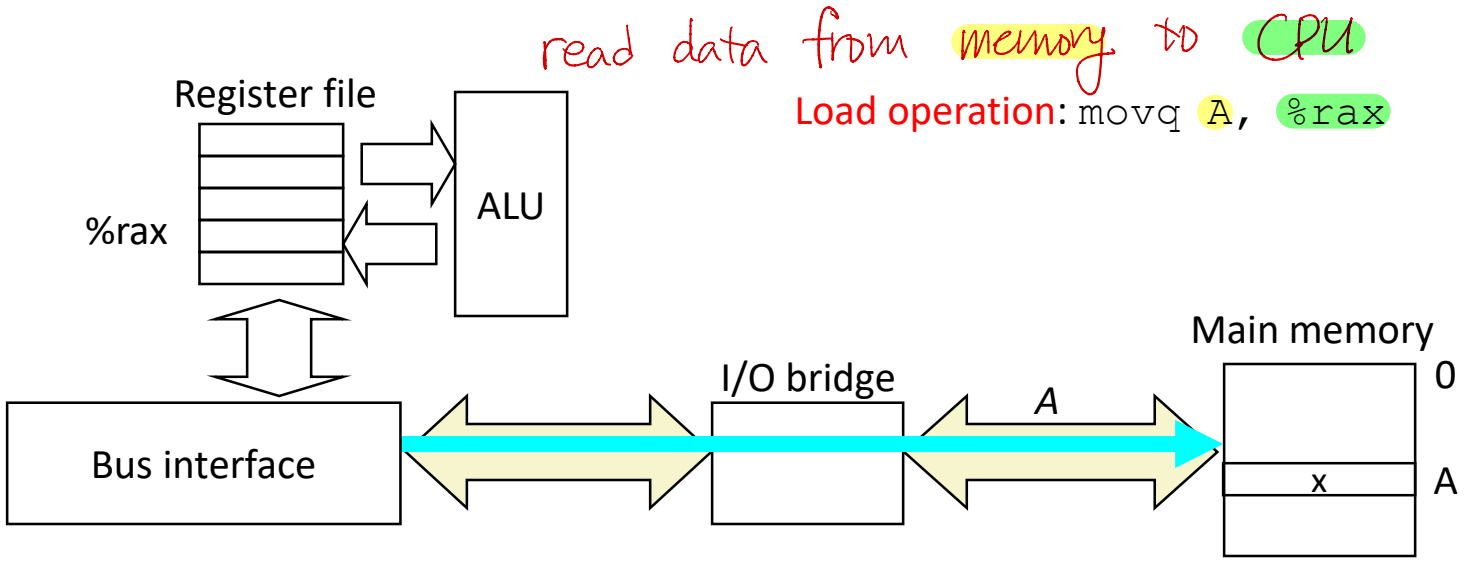
Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
  CPU ↔ Memory
- Buses are typically shared by multiple devices.
  such as IO devices
  keyboard, mouse, ...



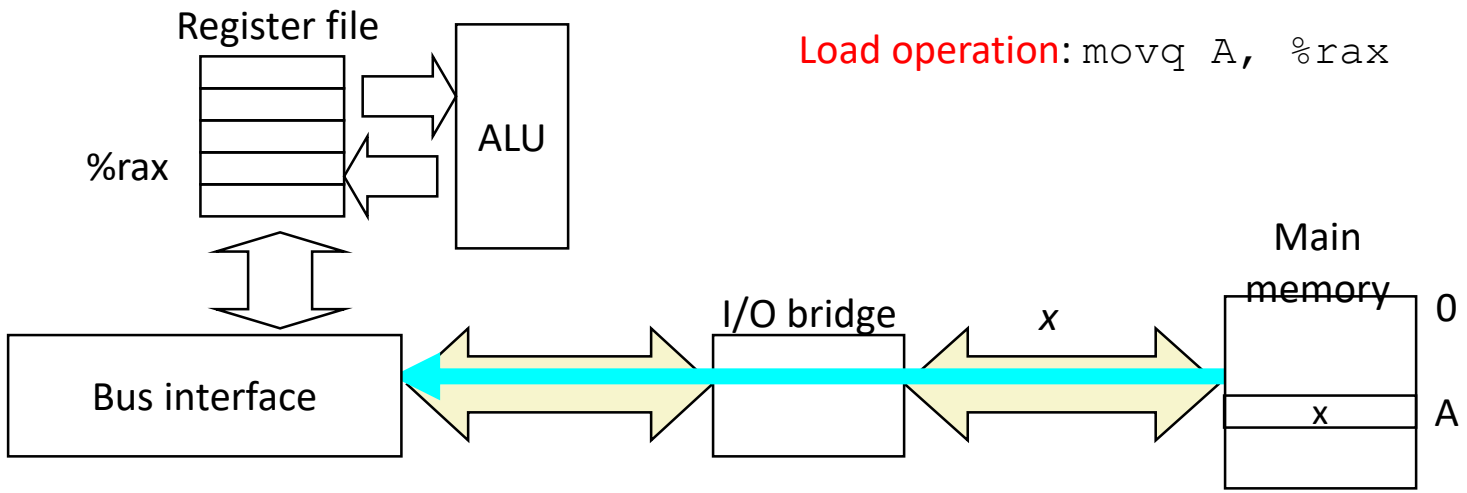
Memory Read Transaction (1)

- CPU places address A on the memory bus.



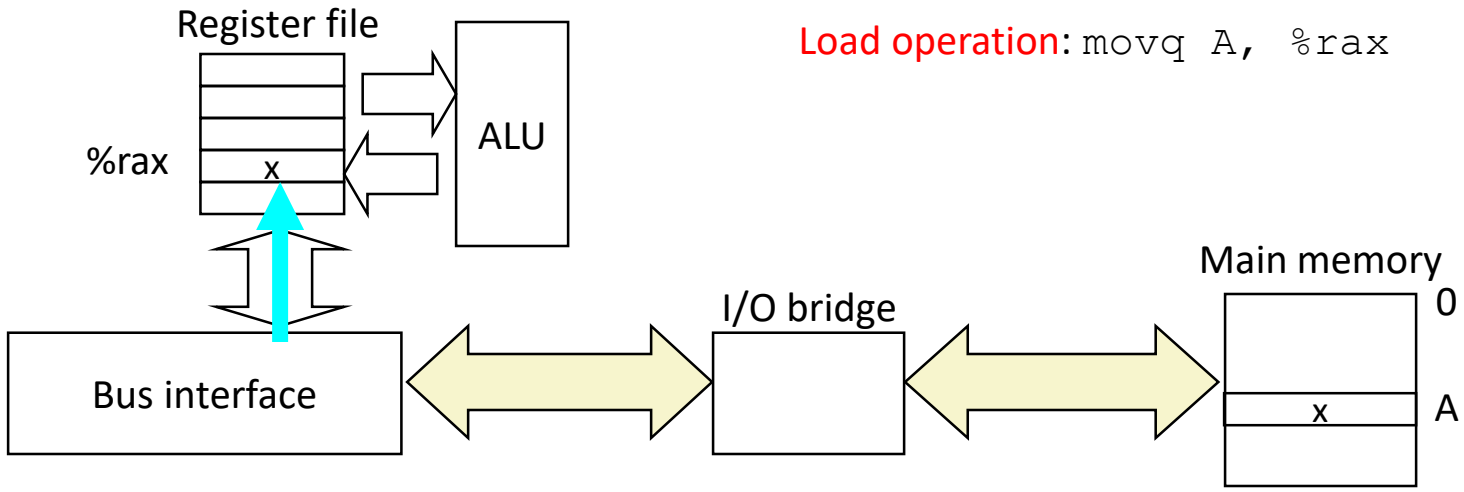
Memory Read Transaction (2)

- **Main memory reads A from the memory bus, retrieves word x, and places it on the bus.**



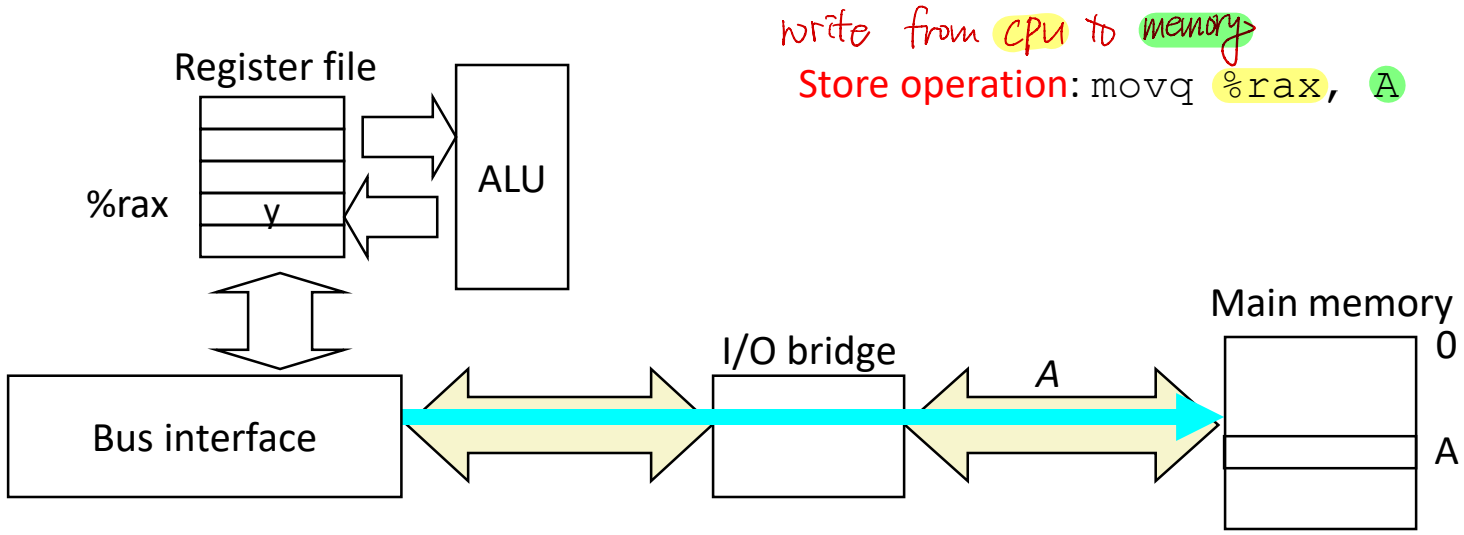
Memory Read Transaction (3)

- CPU read word x from the bus and copies it into register %rax.



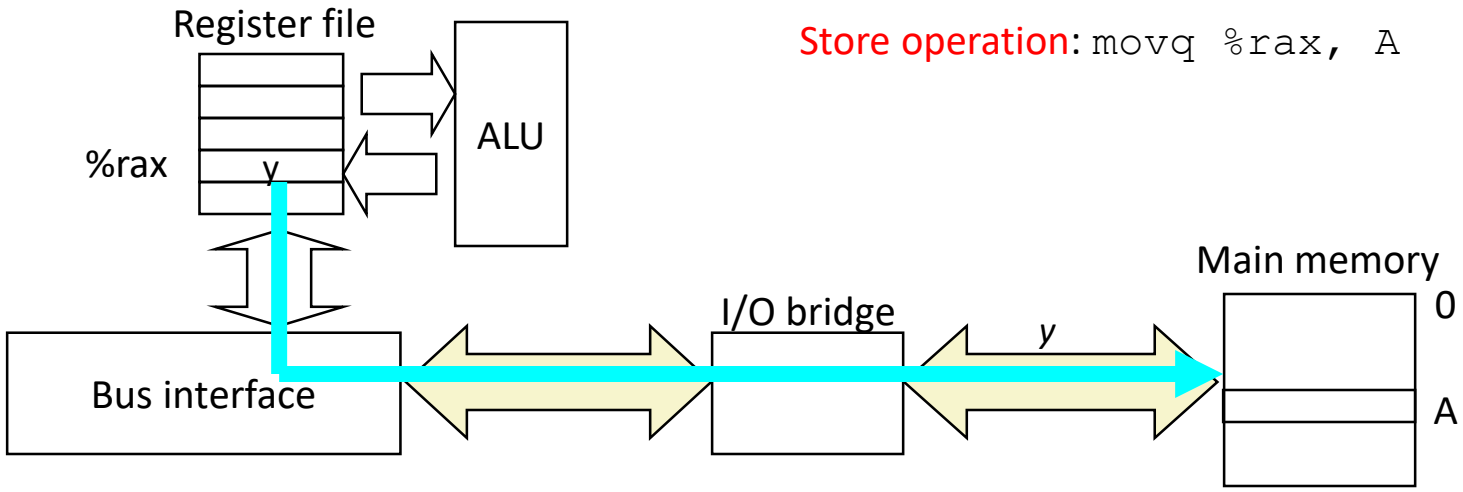
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arri ve.



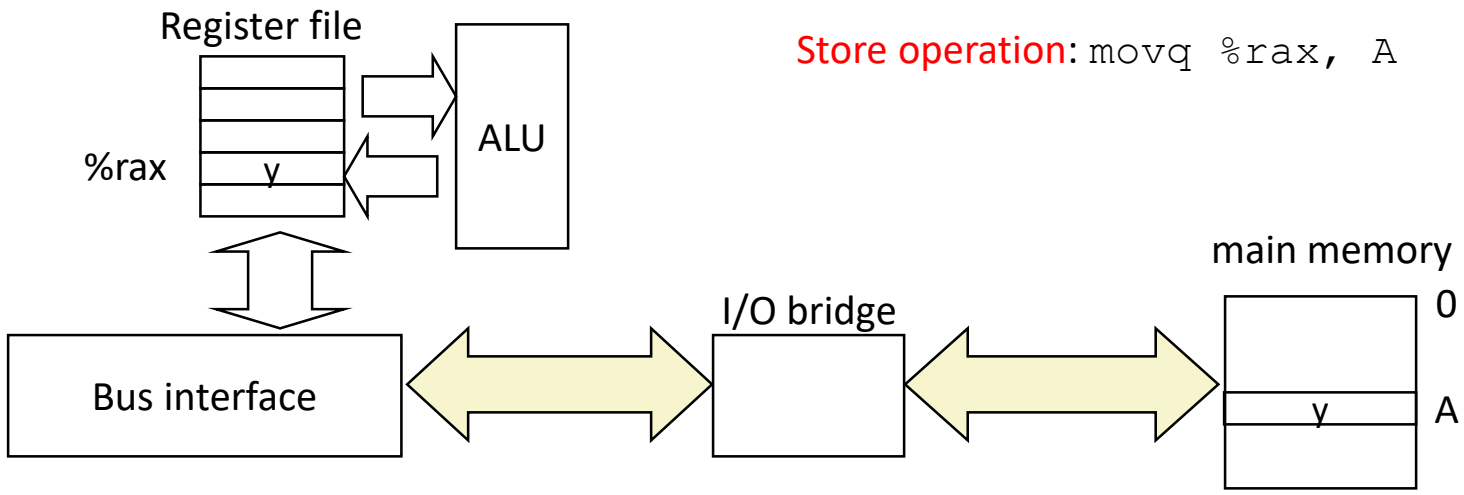
Memory Write Transaction (2)

- CPU places data word y on the bus.

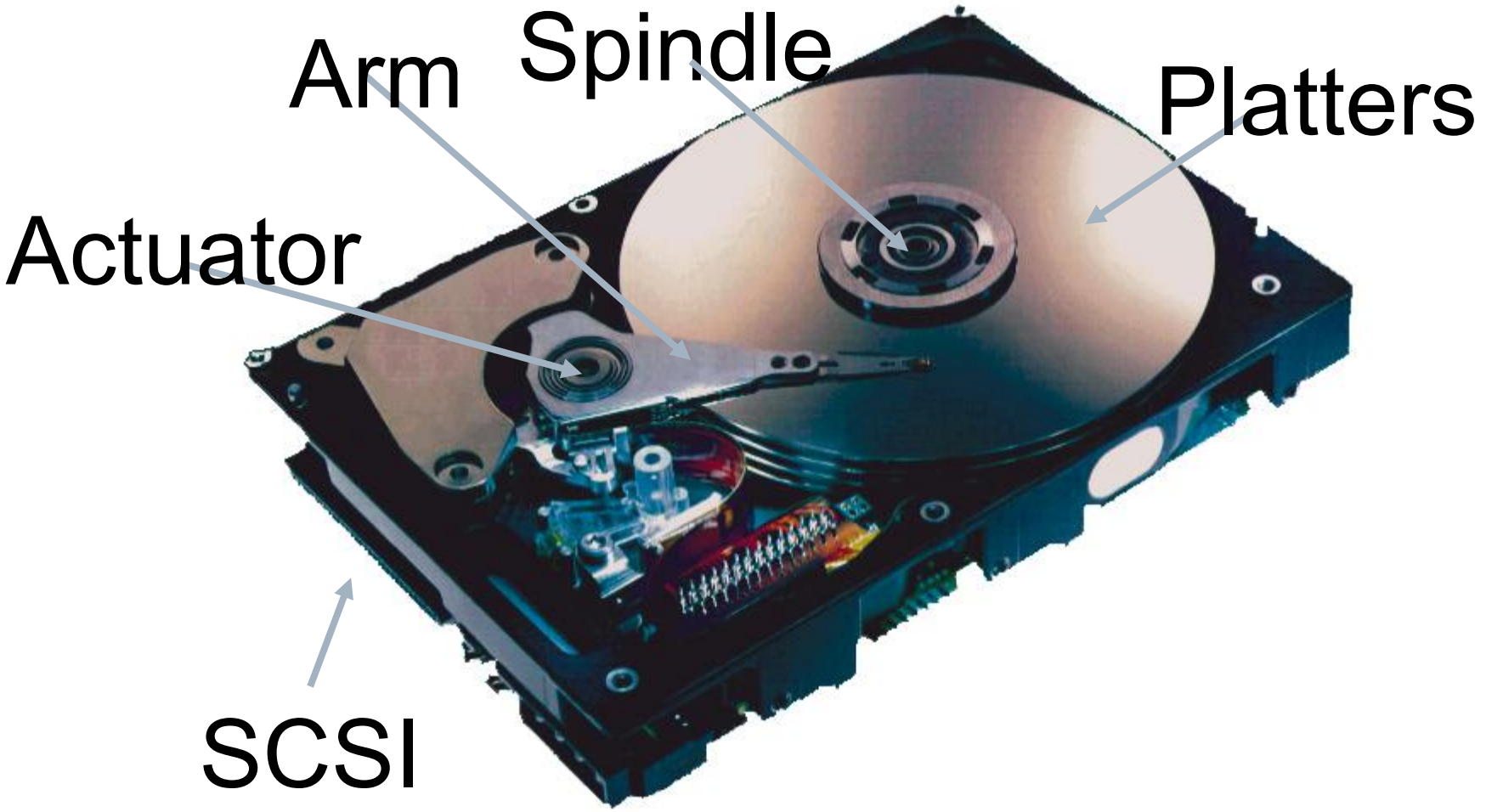


Memory Write Transaction (3)

- Main memory reads data word y from the bus and stores it at address A.



What's Inside A Disk Drive?



connect into
main board

Image courtesy of Seagate Technology

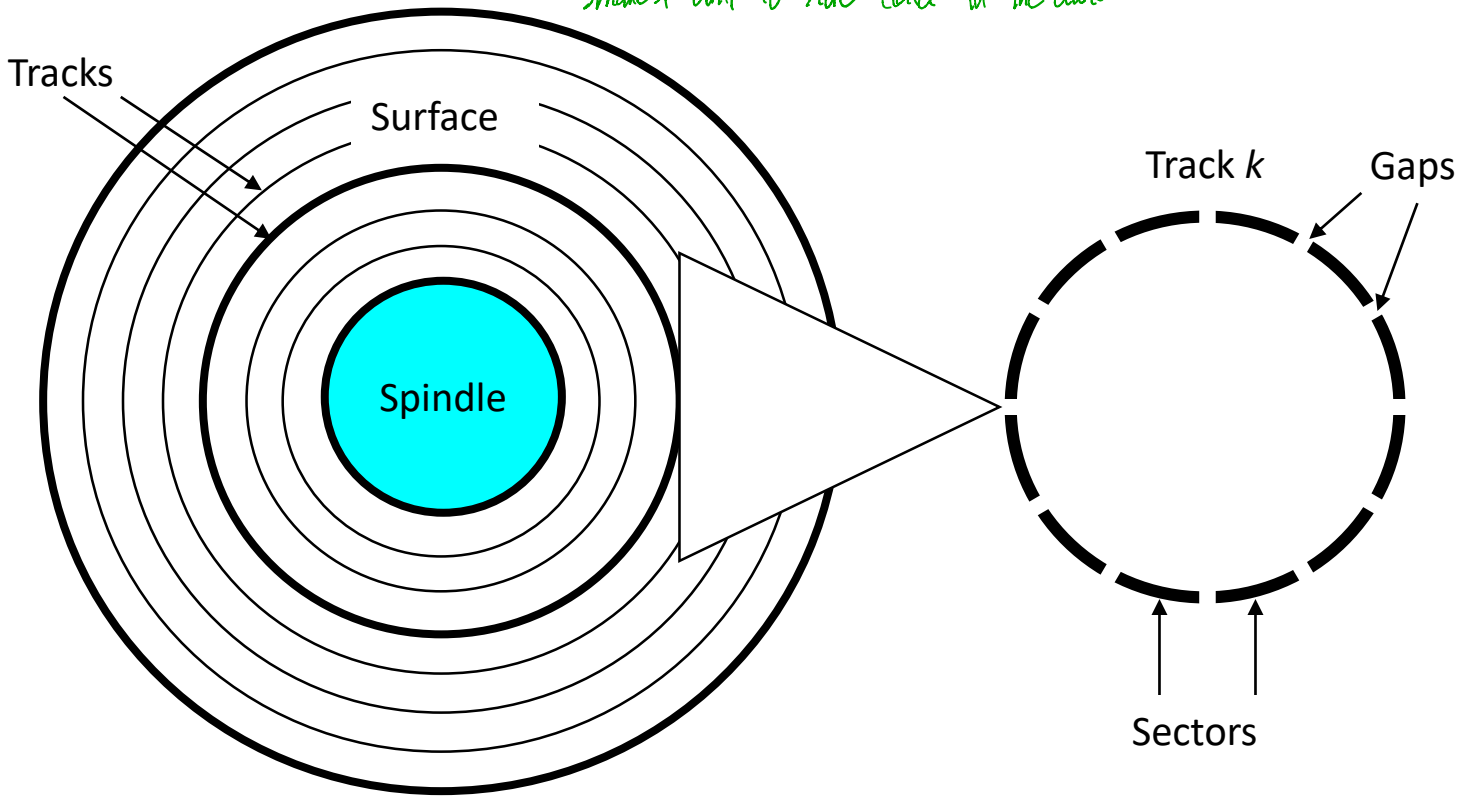
Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.

front
back

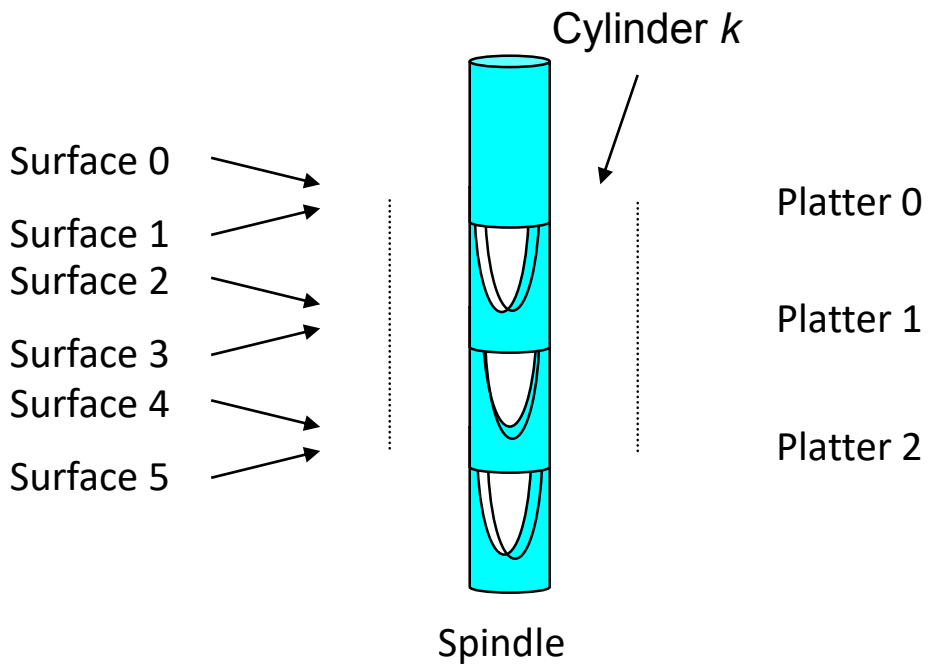
platter
surface
track
sector

smallest unit to store data in the disk



Disk Geometry (Muliple-Platter View)

- **Aligned tracks form a cylinder.**



Disk Capacity

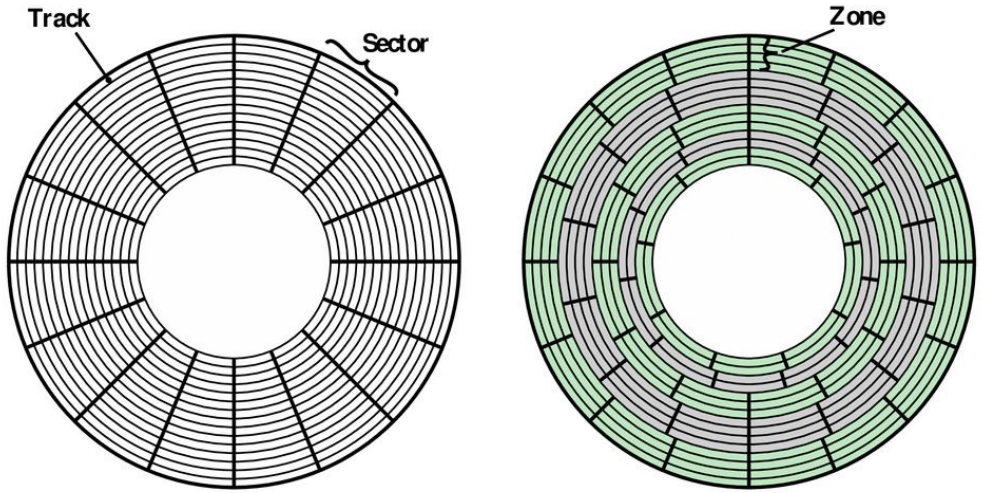
- **Capacity: maximum number of bits that can be stored.**
  - Vendors express capacity in units of gigabytes (GB), where
    1 GB = 10^9 Bytes.

- **Capacity is determined by these technology factors:**
  - Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track. how much data in a track
  - Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment. how many tracks in one surface
  - Areal density (bits/in2): product of recording and track density.

Recording zones

- **Modern disks partition tracks into dis joint subsets called recording zones**
  - Each track in a zone has the same nu mber of sectors, determined by the ci rcumference of innermost track.
  - Each zone has a different number of s ectors/track, outer zones have more s ectors/track than inner zones.
  - So we use **average** number of sectors /track when computing capacity.

Figure 7.3
Comparison of Disk Layout Methods



Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x

(# tracks/surface) x (# surfaces/platter) x

(# platters/disk)

Example:

- 512 bytes/sector
- 300 sectors/track (on average)
- 20,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

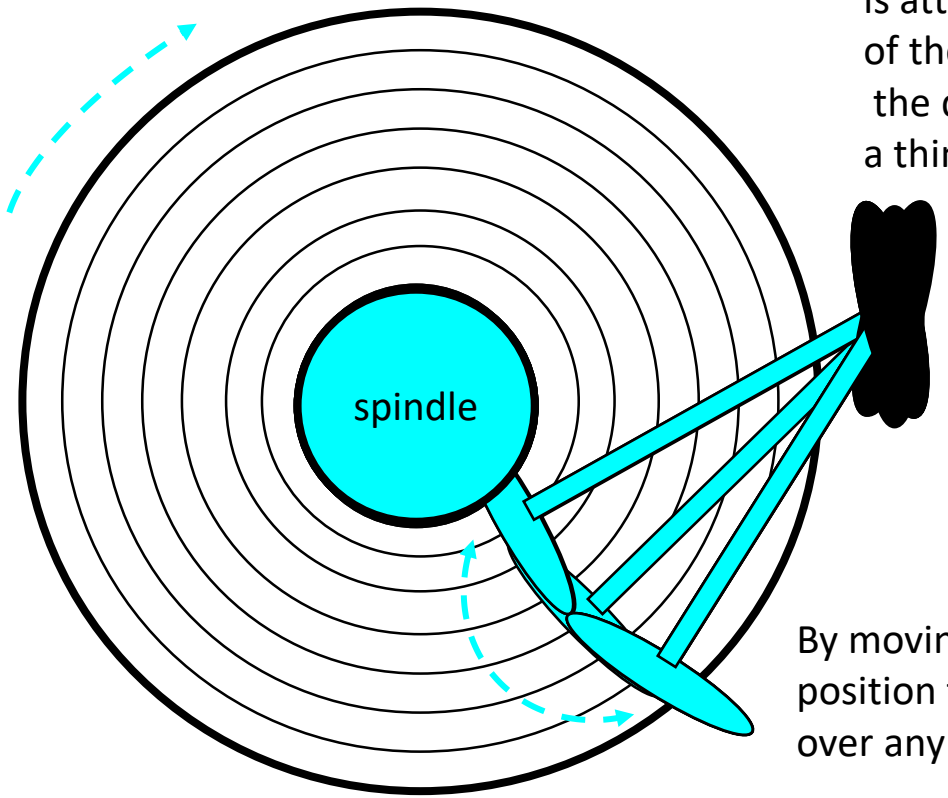
=> Capacity = 512 x 300 x 20000 x 2 x 5 = 30,720,000,000 bytes = 30.72 GB

cf) $1GB = 10^9$ byte

Disk Operation (Single-Platter View)

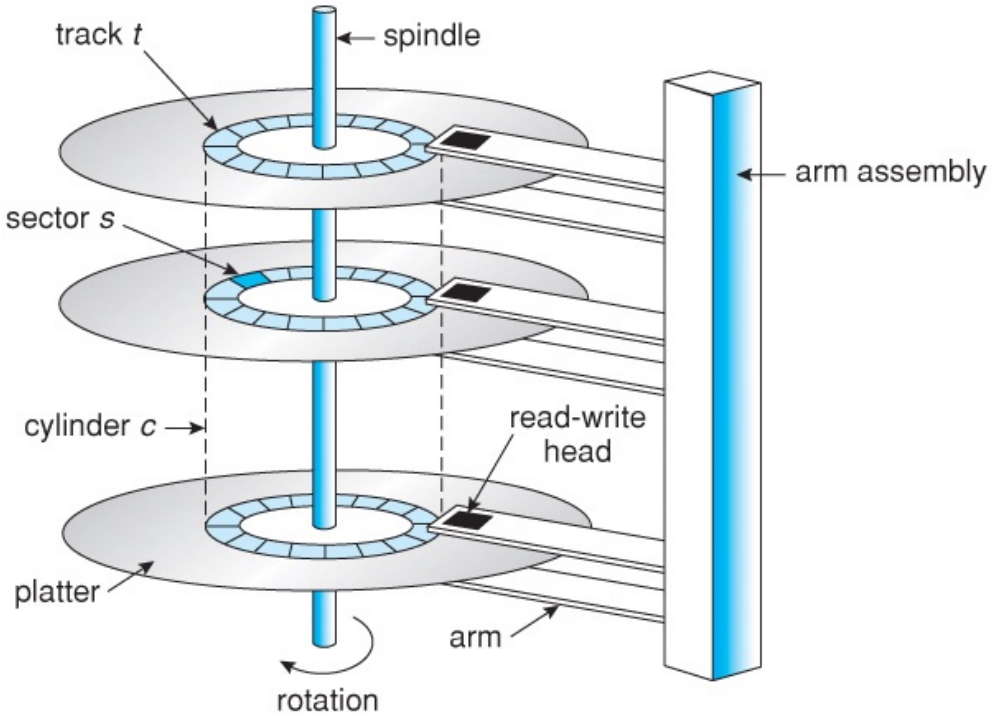
The disk surface spins at a fixed rotational rate

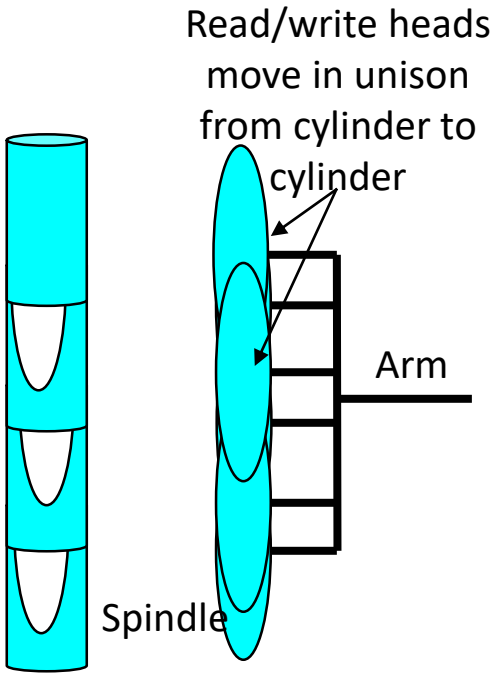
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.



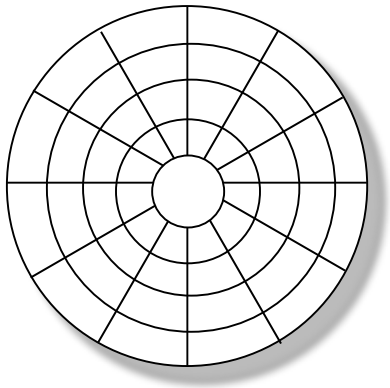
By moving radially, the arm can position the read/write head over any track.

Disk Operation (Multi-Platter View)





Disk Structure - top view of single platter

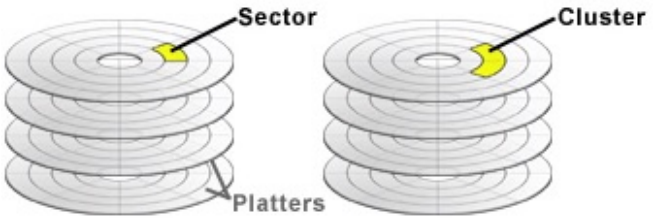


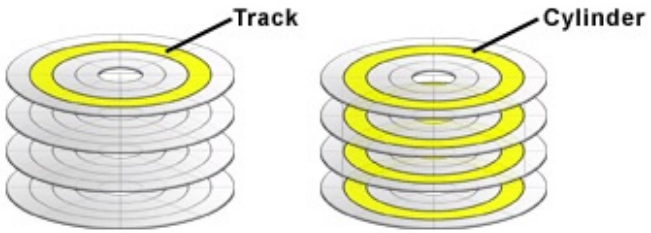
arm can access multiple tracks in the same cylinder at a time!

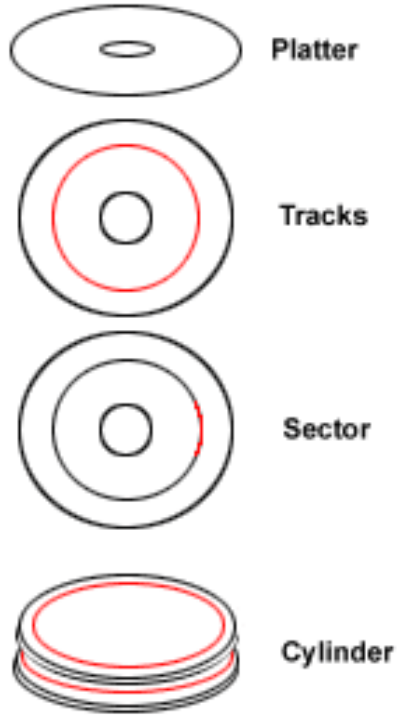
Surface organized into tracks

Tracks divided into sectors

Cylinder ⇒ set of multiple tracks for all the platters.

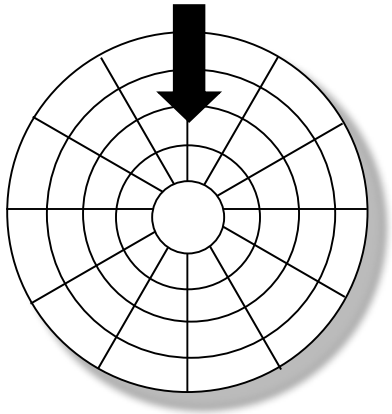






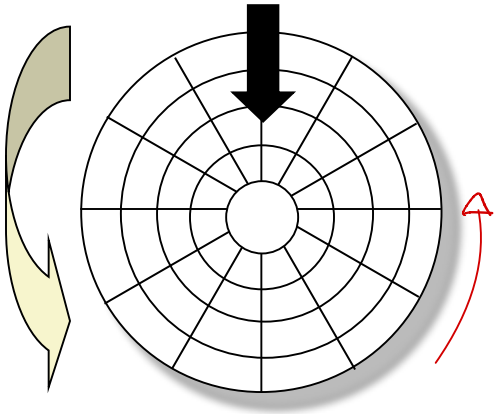
ComputerHope.com

Disk Access



Head in position above a track

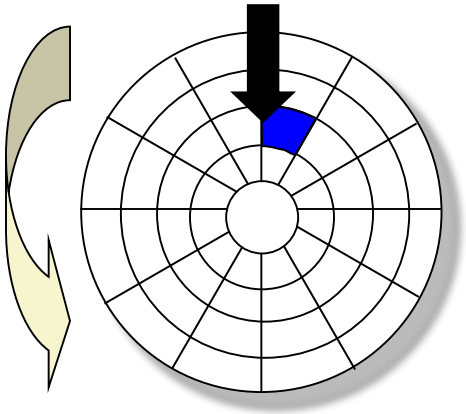
Disk Access



Rotation is counter-clockwise

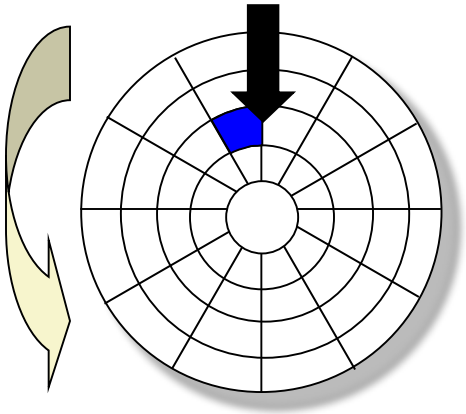
always!

Disk Access – Read



About to read blue sector

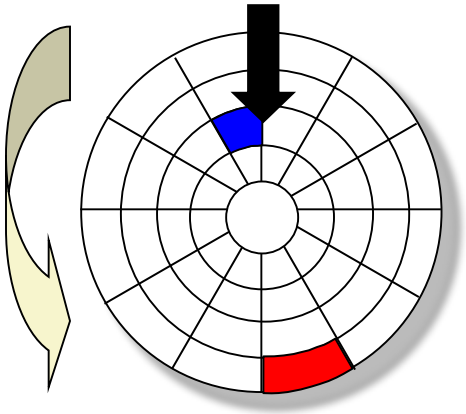
Disk Access – Read



After BLUE read

After reading blue sector

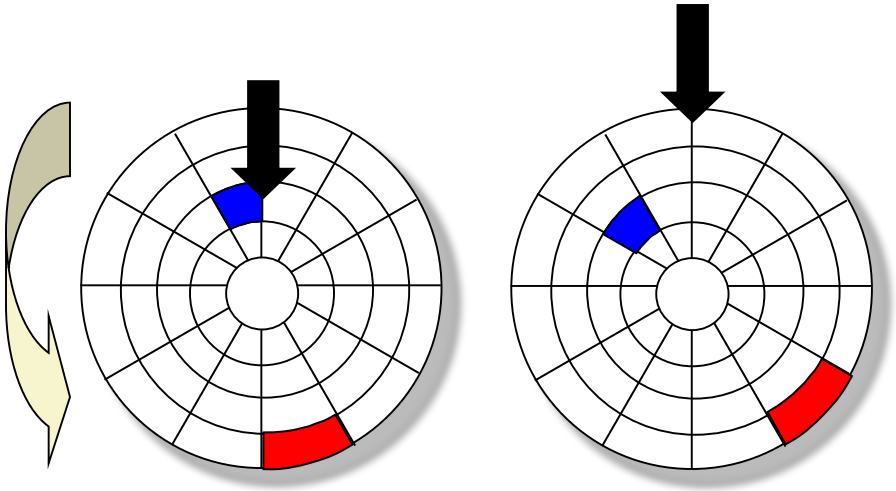
Disk Access – Read



After BLUE read

Red request scheduled next

Disk Access – Seek

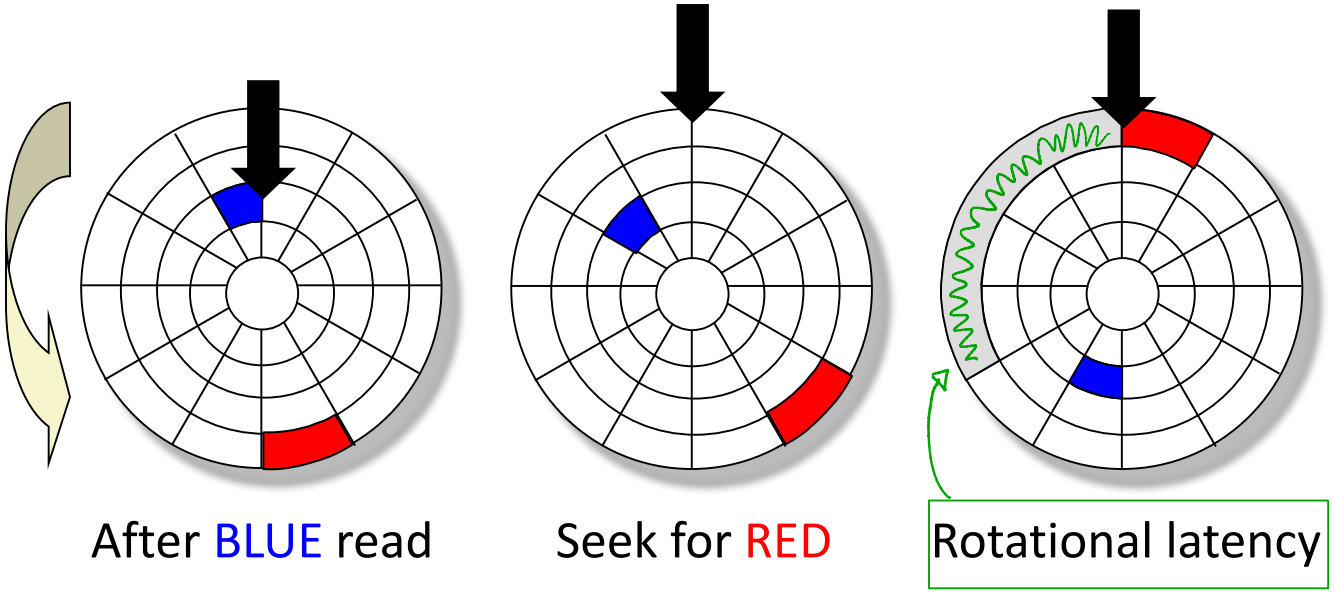


After BLUE read

Seek for RED

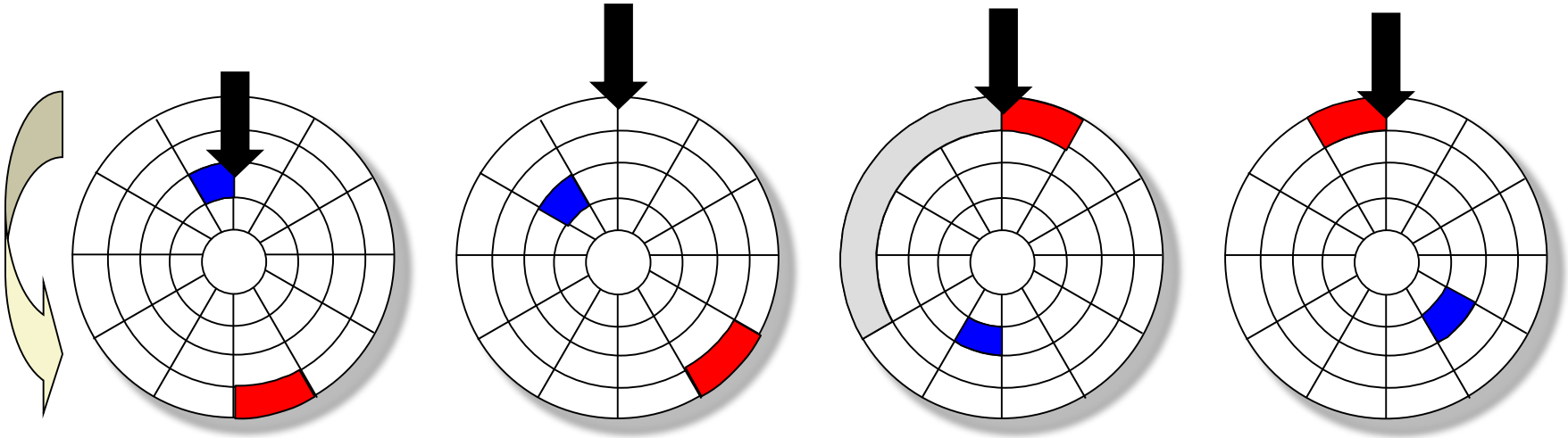
Seek to red's track

Disk Access – Rotational Latency



Wait for red sector to rotate around

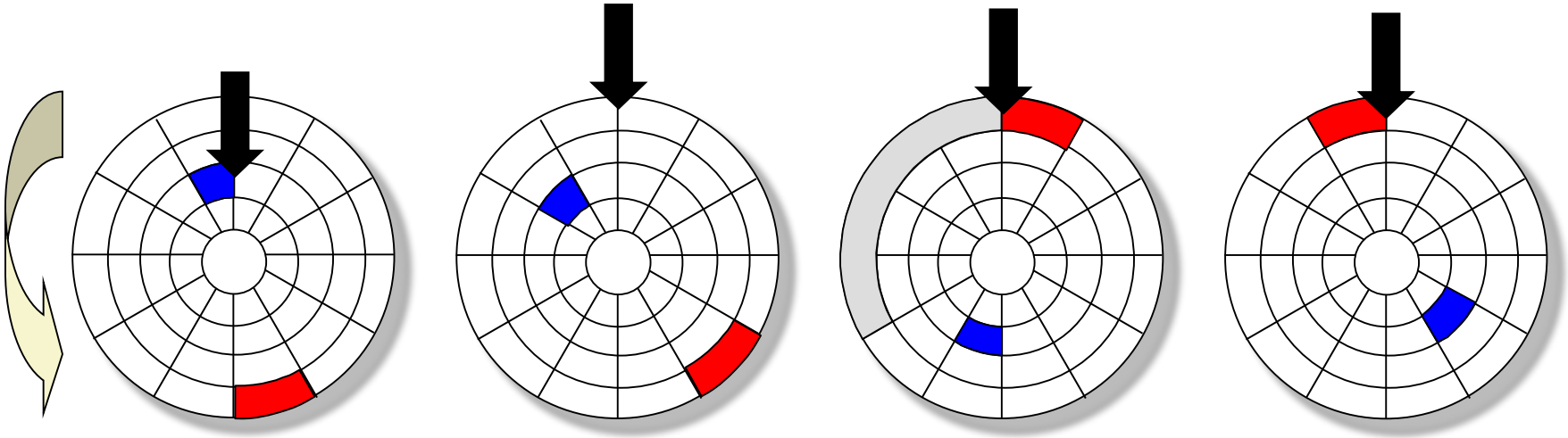
Disk Access – Read



After BLUE read | Seek for RED | Rotational latency | After RED read

Complete read of red

Disk Access – Service Time Components



Data transfer

Seek

Seek time => moving arm into other sector

Rotational latency

Data transfer

☆ => 3 components for accessing the data

Disk Access Time

- **Average time to access some target sector approximated by :**
  - $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$
- **Seek time ($T_{avg\ seek}$)**
  - Time to position heads over cylinder containing target sector.
  - Typical $T_{avg\ seek}$ is 3—9 ms
- **Rotational latency ($T_{avg\ rotation}$)**
  - Time waiting for first bit of target sector to pass under r/w head.
  - $T_{avg\ rotation} = 1/2 \times 1/RPMs \times 60\ sec/1\ min$
  - Typical $T_{avg\ rotation}$ = 7200 RPMs
- **Transfer time ($T_{avg\ transfer}$)**
  - Time to read the bits in the target sector.
  - $T_{avg\ transfer} = 1/RPM \times 1/(avg\ \#\ sectors/track) \times 60\ secs/1\ min.$

Disk Access Time Example

- **Given:**
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.

Of 1sec = 1000 ms

- **Derived:**
  - $T_{avg\ rotation}$ = $(1/2 \times 1/7200 \times 60) \times 1000\ ms/sec = 4ms$
  - $T_{avg\ transfer}$ = $(1/7200 \times 1/400 \times 60) \times 1000\ ms/sec = 0.02\ ms$
  - T_{access} = $9ms + 4ms + 0.02ms = 13.02\ ms$

- **Important points:**
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower then DRAM.