IOWA STATE UNIVERSITY
Department of Electrical and Computer Engineering

# Lecture 41:
# Final Review

# Basic Information

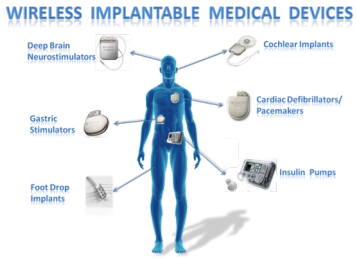
- Time (https://www.registrar.iastate.edu/students/exams/fallexams)
  - 07:30 - 09:30 am, Dec 18 (Wednesday)
- Location
  - Marston 2300
- Format
  - Similar to Midterm exams
  - Closed book/notes
- Scope
  - Lecture 1 to Lecture 40
- 100 points in total
  - about 50 points based on content after Lecture 34

# Review

- OS Introduction
- Process Management
- Memory Management
- Storage Management
- Security
- Virtualization
- Cloud Computing

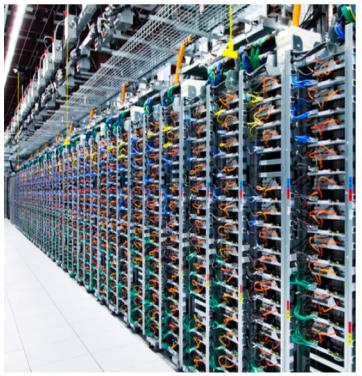
# Review

- Introduction
  - Why OS
    - Fundamental to computer systems
      - Affects correctness, security, performance ... of entire system
    - Fundamental to modern society



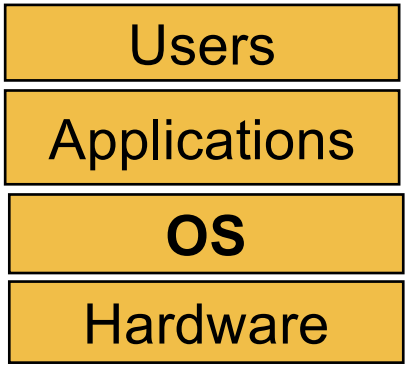


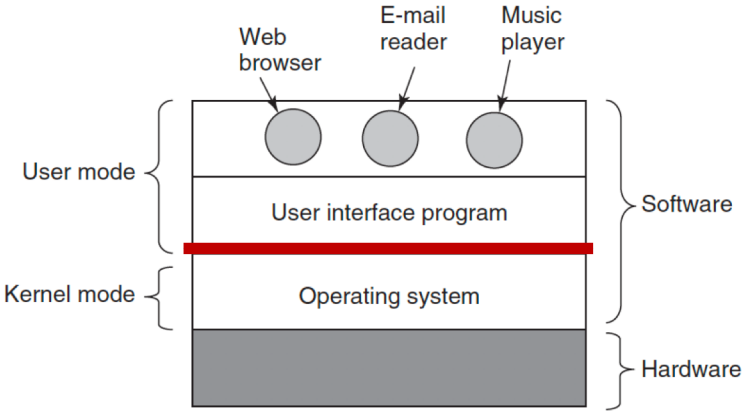
  - What's OS
    - OS is a resource manager
    - OS is a control program
      - an extended/virtualized machine with abstraction



# Review

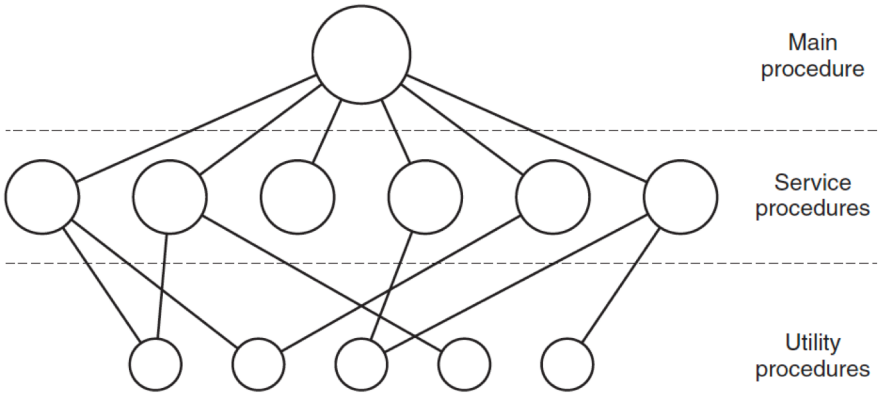
- Introduction
  - OS Abstractions for HW
    - CPU
      - process and/or thread
    - Memory
      - address space
    - Disks
      - Files
  - System Calls
    - Function calls implemented by OS
    - interface to apps/users

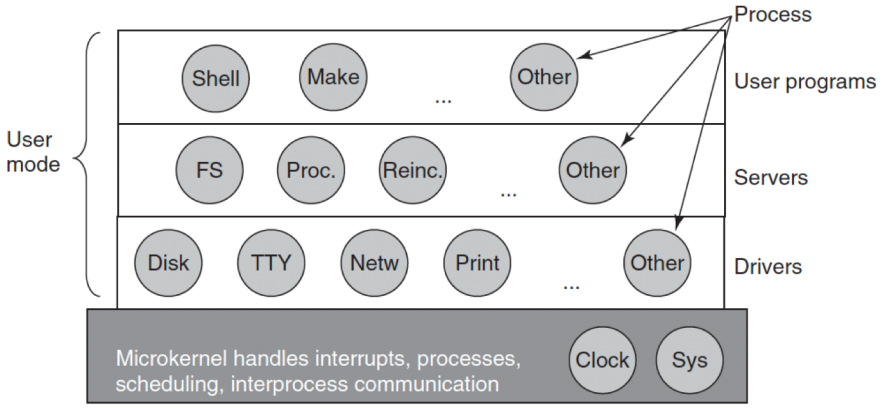


# Review

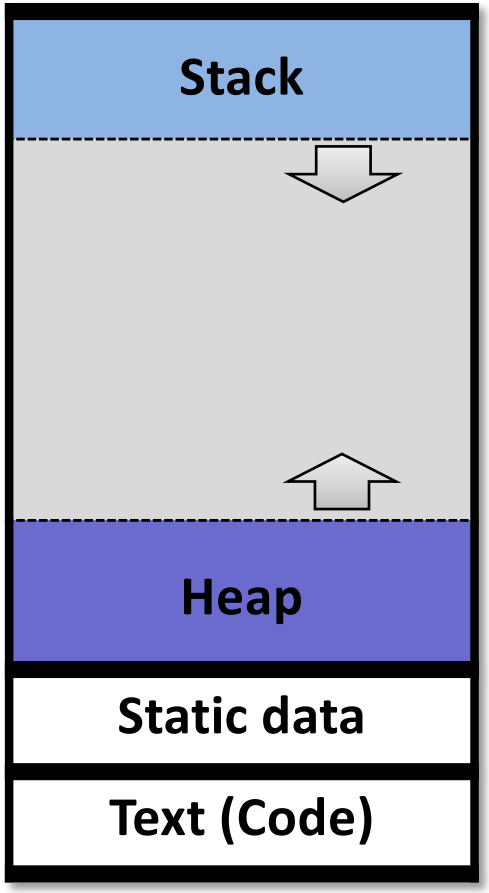
- Introduction
  - OS Structures
    - Monolithic kernel V.S. Microkernel
      - Monolithic: the OS runs as a single program in kernel mode
      - Microkernel: split the OS up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode
      - Tradeoffs



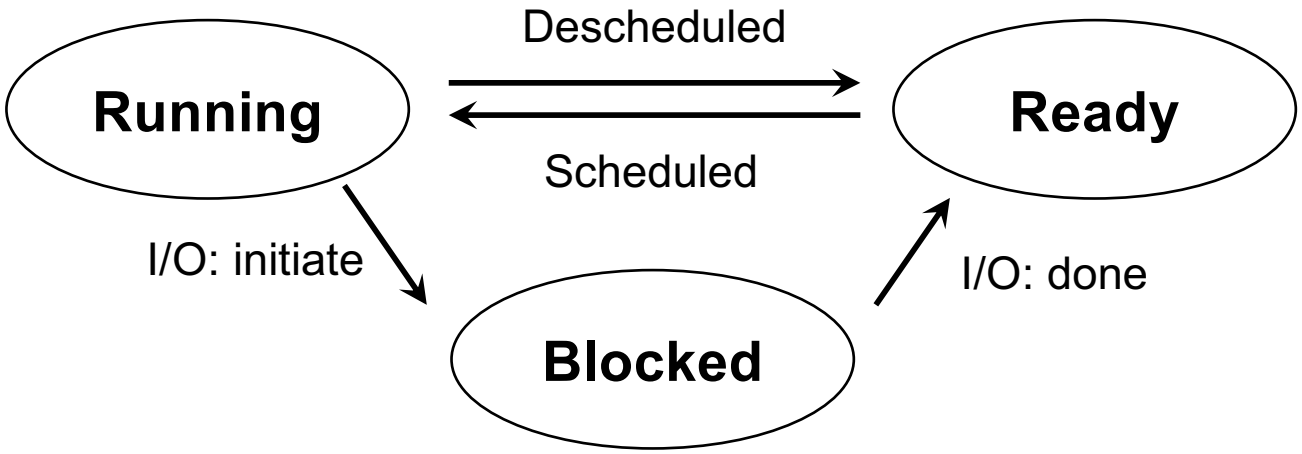
# Review

- Process
  - A program *in execution*
    - holds all the info needed to run a program
      - e.g., register values
  - Address Space
    - Four segments
      - stack
        - local variables, return address, etc.
      - heap
        - dynamically allocated data
      - static data
        - global/static variables
      - code



# Review

- Process States
  - Three basic states
    - **Running**
    - **Ready**
    - **Blocked**



# Review

- Process Context
  - Stored in a “Process Control Block (PCB)”

| Process management | Memory management | File management |
|---|---|---|
| Registers | Pointer to text segment info | Root directory |
| Program counter | Pointer to data segment info | Working directory |
| Program status word | Pointer to stack segment info | File descriptors |
| Stack pointer | | User ID |
| Process state | | Group ID |
| Priority | | |
| Scheduling parameters | | |
| Process ID | | |
| Parent process | | |
| Process group | | |
| Signals | | |
| Time when process started | | |
| CPU time used | | |
| Children’s CPU time | | |
| Time of next alarm | | |

- Context Switch
  - switching the CPU to another process by
    - saving the context of an old process
    - loading the context of a new process

# Review

- Process APIs
  - Example: fork()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid()); //get process ID
    int rc = fork();       // create a child process
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```

# Review

- Process APIs
  - Example: fork()
    - results (non-deterministic)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

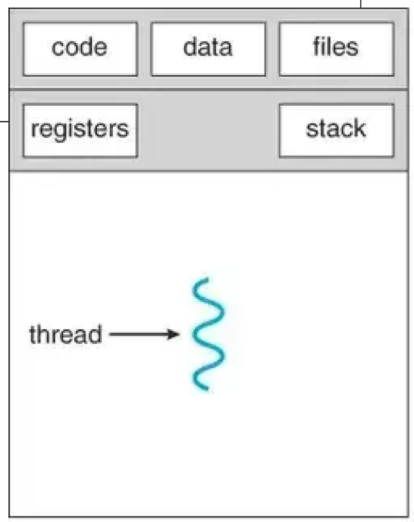
or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

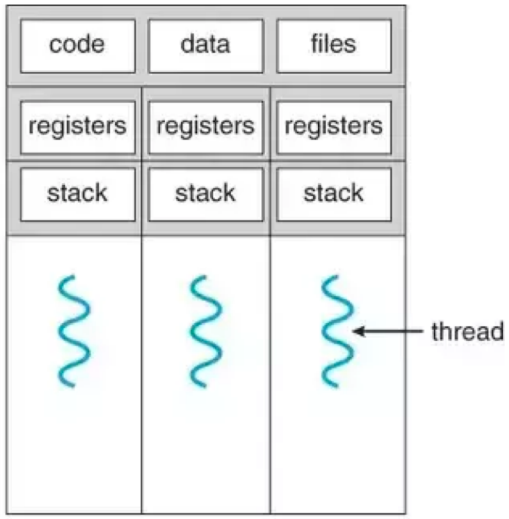
# Review

- Multi-threaded Process
  - Multiple threads of control within a process

| Per-process items | Per-thread items |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |



single-threaded process



multithreaded process

# Review

- Process/Thread Scheduling
  - First-Come, First-Served (FCFS)
  - Shortest-Job-First (SJF)
  - Shortest Remaining Time Next
  - Round Robin (RR)
  - Priority Scheduling

# Review

- Process/Thread Scheduling
  - First-Come, First-Served (FCFS)
    - Scheduling based on the arrival order of processes

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Assume the processes arrive in the order: ***$P_2$ , $P_3$ , $P_1$***
- The Gantt Chart for the schedule is:


- Waiting time: $P_1$ = 6; $P_2$ = 0; $P_3$ = 3
- Average waiting time: (6 + 0 + 3)/3 = 3

# Review

- Inter-Process Communications (IPC)
  - Two basic methods
    - Shared memory
    - Message passing
  - Race Condition (Data Race)
    - two or more processes are reading or writing some **shared data** and the final result depends on who runs precisely when
  - Critical Region
    - A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread
      - Multiple threads executing critical section can result in a race condition.
      - Need to support **mutual exclusion**

# Review

- Mutual Exclusion: Semaphore
  - Semaphore **S** – integer variable
    - Can only be accessed via two indivisible (atomic) operations
      - **down()** and **up()**
        - originally called **P()** and **V()**
        - also called **wait()** and **signal()**
    - meaning of **down()** operation (atomic):

```
down(S) {
    while (S <= 0)
        ; // busy waiting
    S--;
}
```

    - meaning of **up()** operation (atomic):

```
up(S) {
    S++;
}
```

# Review

- Semaphore Example
  - Readers-Writers Problem

READER:

```
While (1) {
        down(protector);
        rc++;
        if (rc == 1)  //first reader
                down(database);
        up(protector);

        read();

        down(protector);
        rc--;
        If (rc == 0) then // last one
                up(database);
        up(protector);
        ….
}
```

WRITER:

```
While (1) {
        generate_data();
        down(database);
        write();
        up(database);
}
```

Two semaphores:

database

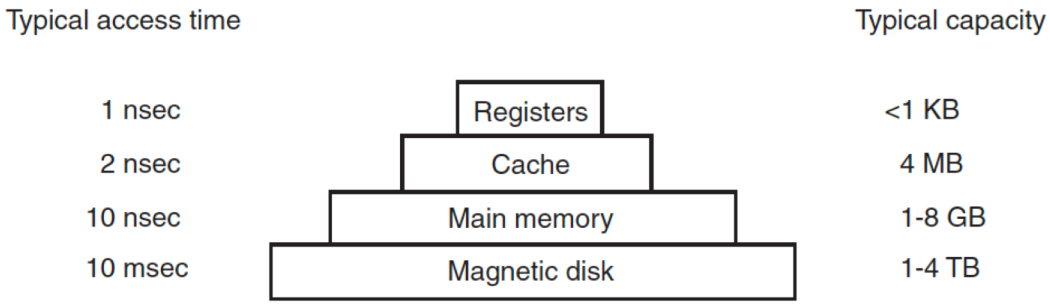
protector

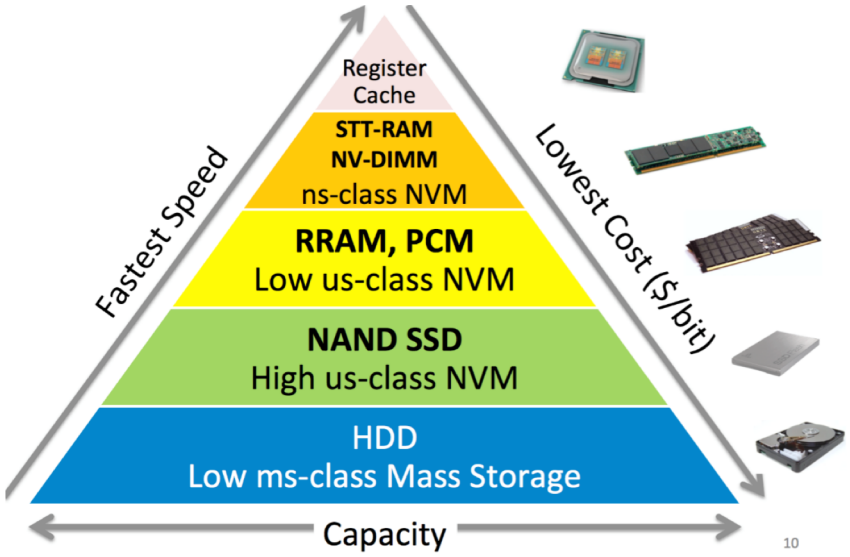
Initial: protector=1, database =1

rc =0

# Review

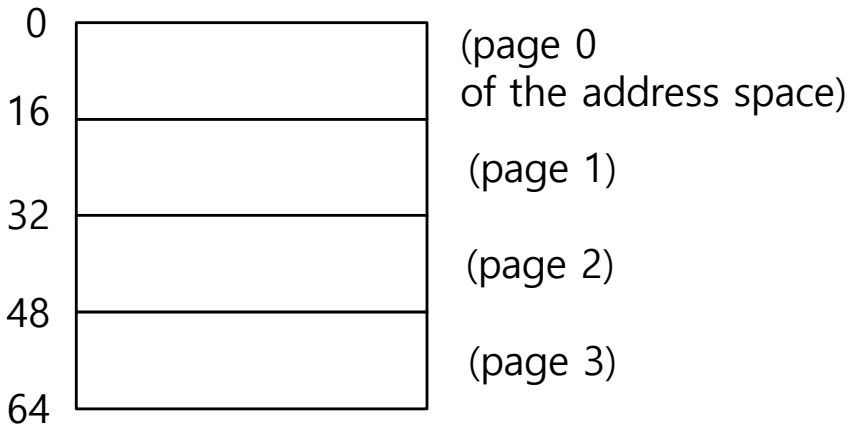
- Memory Hierarchy
  - diverse technologies with tradeoffs
    - latency, capacity, persistency, cost, …
    - non-volatile memories are revolutionizing the market!
      - A “disruptive” technology



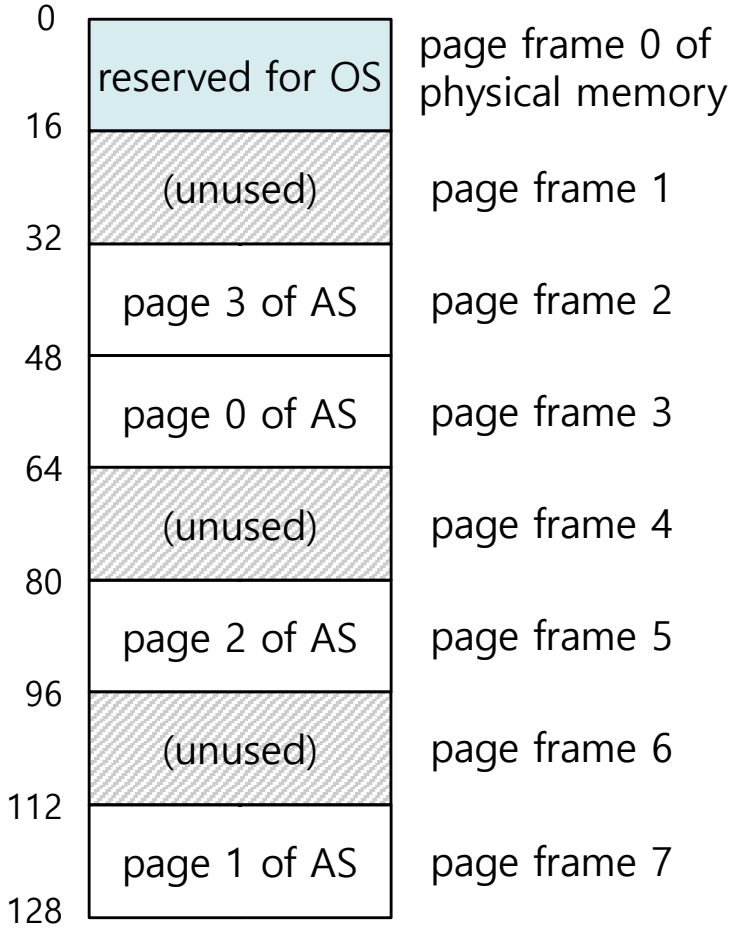



# Review

- Paging
  - split up address space into fixed-size units called ***pages***
    - physical memory is also split into fixed-size units called ***page frames***
    - Flexibility & simplicity



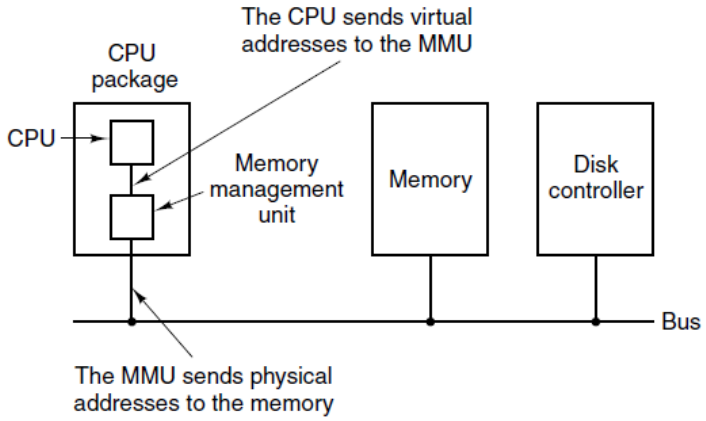

**64-Byte Address Space**

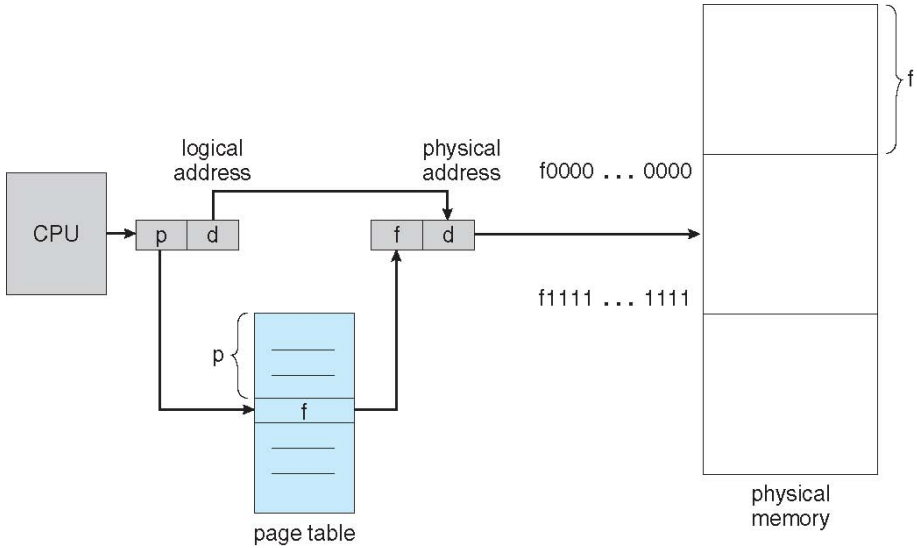



**128-Byte Physical Memory**

# Review

- Address Translation via **Page Table**
  - Each virtual address is divided into two parts:
    - VPN: virtual page number (p)
      - used as an index into the **page table**
    - Offset: offset within the page (d)
  - Hardware involved: MMU



# Review

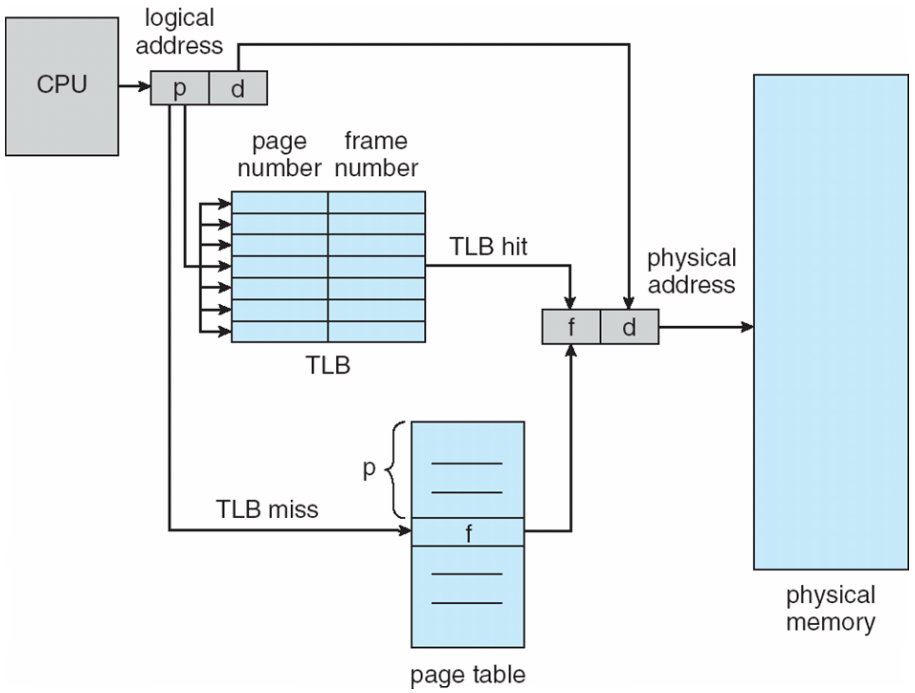
- Page Fault
  - Requested page not in the physical memory
    - MMU reports a page fault to CPU
    - CPU gives control to the OS (page fault handler routine)
    - OS fetches a page from the disk
      - May needs to evict an existing page from memory
    - Instruction is restarted
- Potential Issues of Paging
  - Time
    - for every memory access, one additional access is needed
  - Space
    - A page table can get awfully large
    - Each process needs to have a page table

# Review

- Space Overhead of Page Tables
  - A (linear) page table can be large
    - Example (revisit):
      - 32-bit address space (4GB) with 4KB pages
      - 12 bits for offset within a page (4K=2^12)
      - 20 bits for VPN (32 – 12 = 20)
      - $4MB$= 2^20 $entries * 4\ Bytes\ per\ page\ table\ entry$
  - Each process needs to have a page table
    - 100 processes needs 4MB * 100 = 400MB

# Review

- Translation Lookaside Buffer (TLB)
  - Hardware cache for speeding up paging
  - Address translation with TLB & page table

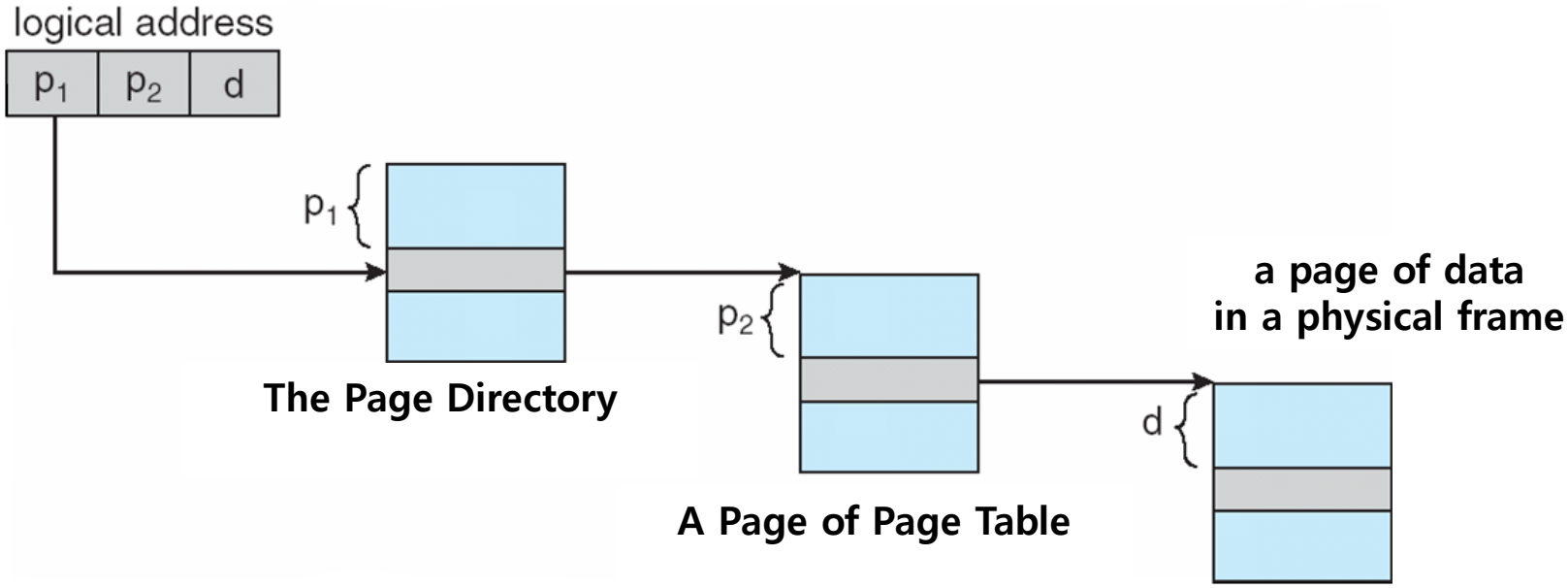


# Review

- TLB may improve performance greatly
  - Effective Access Time (EAT)
    - effective time for accessing memory with TLB
    - TLB Hit ratio = $\alpha$
      - percentage of times that a page number is found in the TLB
      - TLB hit: one memory access
      - TLB miss: two memory accesses
    - Consider $\alpha$ = 80%, 100ns for each memory access
      - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
    - Consider a more realistic hit ratio $\alpha$ = 99%; still 100ns for each memory access
      - EAT = 0.99 x 100 + 0.01 x 200 = 101ns
  - Temporal & spatial locality

# Review

- Multi-Level Page Tables
  - Paging the page table itself
    - e.g., a two-level page table
      - the page directory index (p1) is used to identify a page directory entry (PDE) in the page directory
      - the page table index (p2) is used to identify a page table entry



# Review

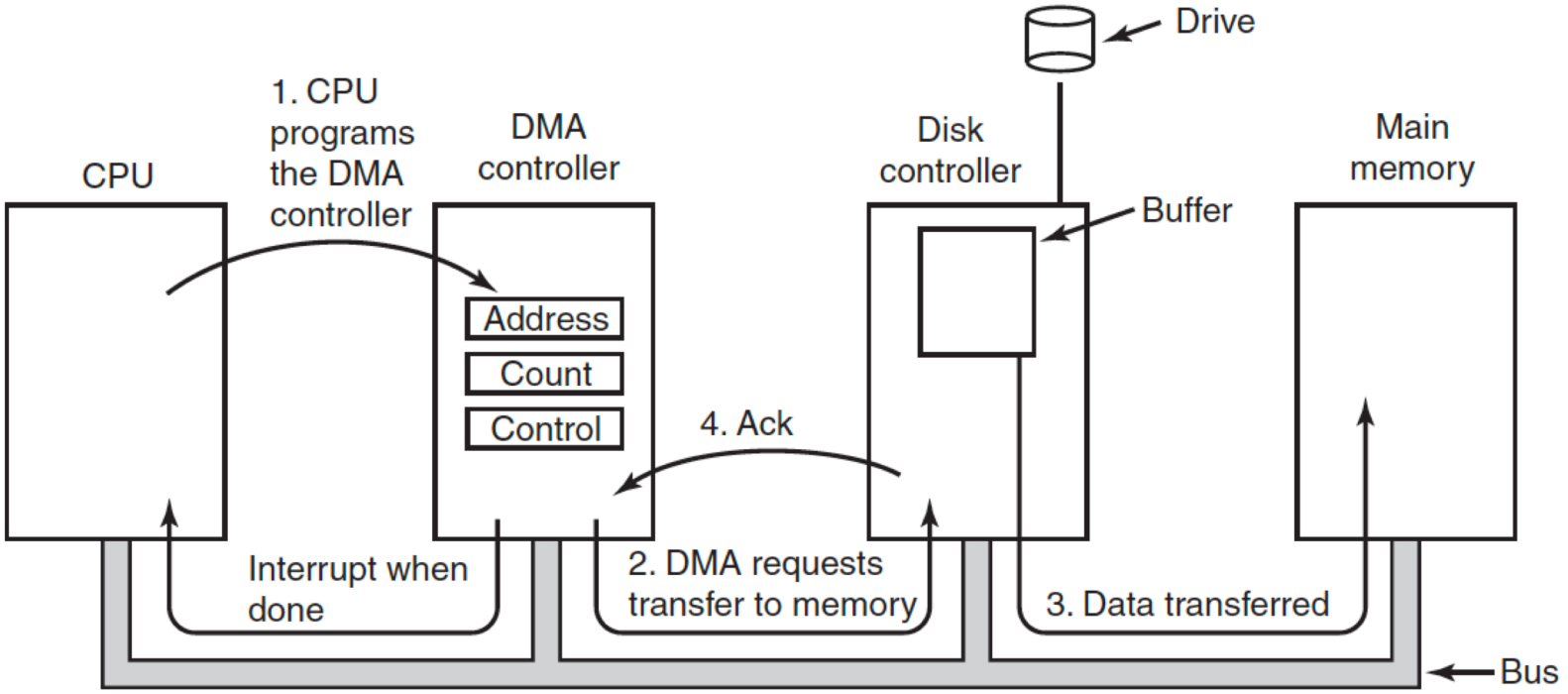
- Page Replacement Algorithms
  - When free physical memory is low, page replacement algorithms decide which page to evict
    - the physical memory serves as a cache of the **swap space**
      - a cache miss leads to a page fault
    - the goal is to minimize the number of page faults
  - The Optimal Algorithm
  - FIFO Algorithm
  - LRU Algorithm
  - Clock Algorithm

# Review

- I/O Devices
  - Two basic types
    - Block devices
      - stores information in fixed-size blocks, each one with its own address
      - All transfers are in units of one or more entire (consecutive) blocks
        - can read or write each block independently of all the other ones
      - E.g., Hard disks, CDROM, USB
    - Character devices
      - delivers or accepts a stream of characters (bytes), without any block structure
      - not addressable and does not have any seek operation
      - E.g., printer, network interface card (NIC)

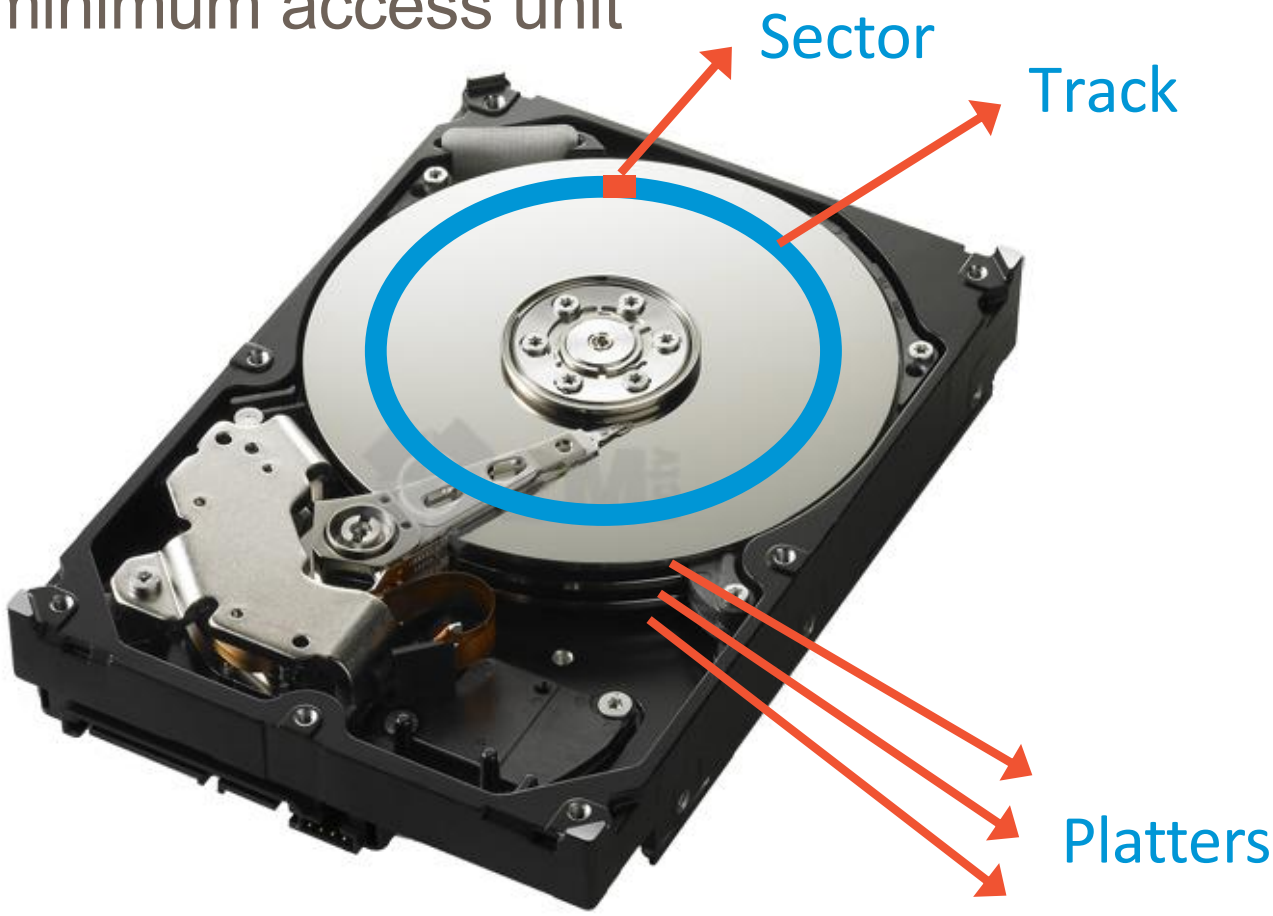
# Review

- Direct Memory Access (DMA)
  - transfer data between memory & I/O device without involving CPU



# Review

- Hard Disk Drive (HDD)
  - a sector is the minimum access unit
    - e.g., 512B



# Review

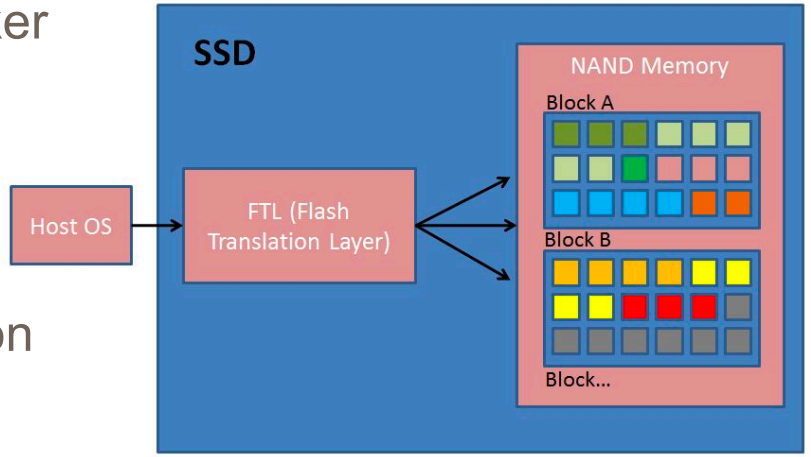
- HDD I/O Time & I/O Rate
  - I/O time ($T_{I/O}$) includes three parts
    - **Seek**
    - **rotational delay**
    - **Transfer**

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

  - I/O rate ($R_{I/O}$): $$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$
  - Favor sequential workloads

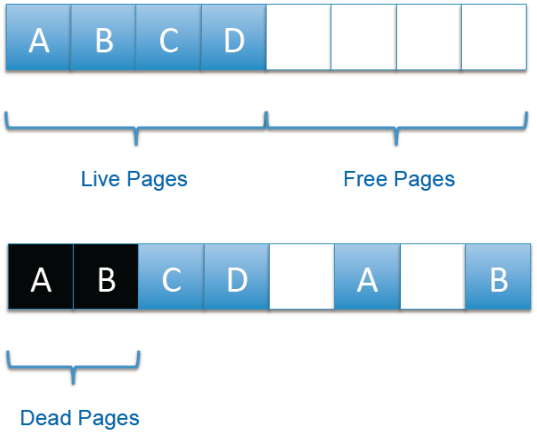
# Review

- Solid State Drives (SSDs)
  - Flash Memory
    - **SLC vs MLC**
      - MLC is used in consumer marker
    - **NOR** vs **NAND**
      - NAND is used in SSDs
    - **Block**
      - minimum unit of **erase** operation
      - contain multiple pages
    - **Page**
      - minimum unit of **program** operation
    - each cell can only stand a limited number of program/erasure cycles (**P/E cycles**)



# Review

- Solid State Drives (SSDs)
  - Flash Translation Layer (FTL)
    - Logical block mapping
      - maps logical addresses to physical addresses
      - maintains a mapping table
      - out-of-space update (append-only)
    - Garbage Collection
      - re-cycle invalid pages
      - source of I/O instability
    - Wear leveling
      - let the flash cells be erased/programmed about the same number of times

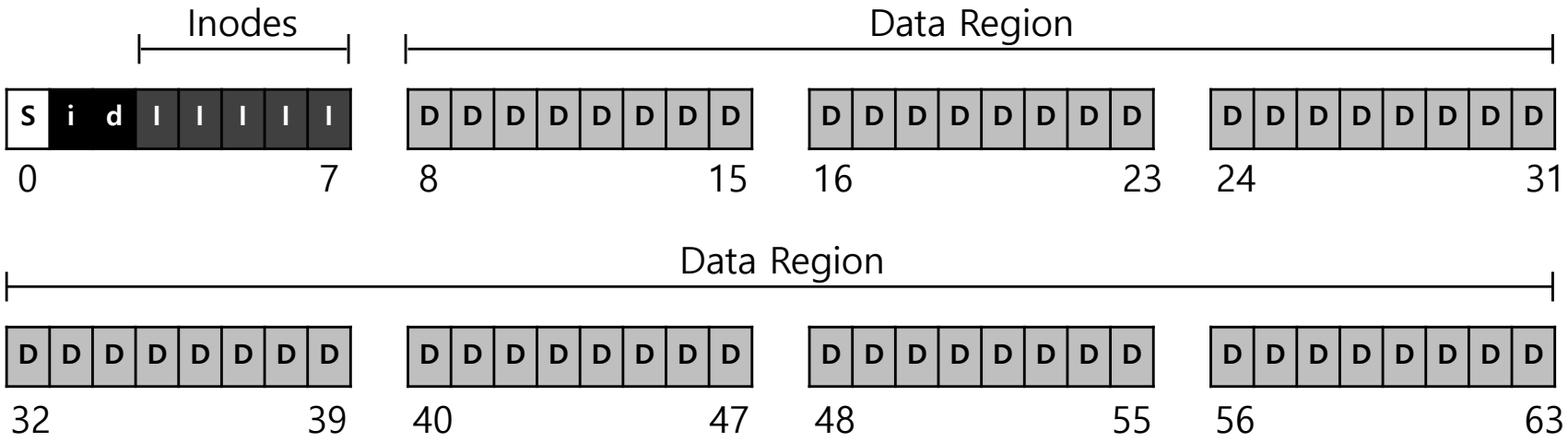


# Review

- File Systems
  - File: contains user data
    - A **file system** (FS) is responsible for managing and storing files persistently on disk
      - data structures
      - implementations of file operations
    - Each file has a unique, low-level name called **inode number** in the file system
      - each file has a corresponding inode data stucture storing the metadata
      - inode number is used to find the inode in the inode table
  - Directory: contains a list of (user-readable name, low-level name) pairs.
    - each entry refers to either *files* or other *directories*

# Review

- File Systems
  - Basic layout
    - data region: user data
    - metadata region: inodes, bitmaps, superblock



# Review

- File Systems
  - Timeline of reading a file (/foo/bar) from disk

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[2] |
|---|---|---|---|---|---|---|---|---|---|---|
| **open(bar)** | | | read | | | | | | | |
| | | | | read | | read | | | | |
| | | | | | | | read | | | |
| | | | | | read | | | | | |
| **read()** | | | | | read | | | | | |
| | | | | | write | | | read | | |
| **read()** | | | | | read | | | | | |
| | | | | | write | | | | read | |
| **read()** | | | | | read | | | | | |
| | | | | | write | | | | | read |

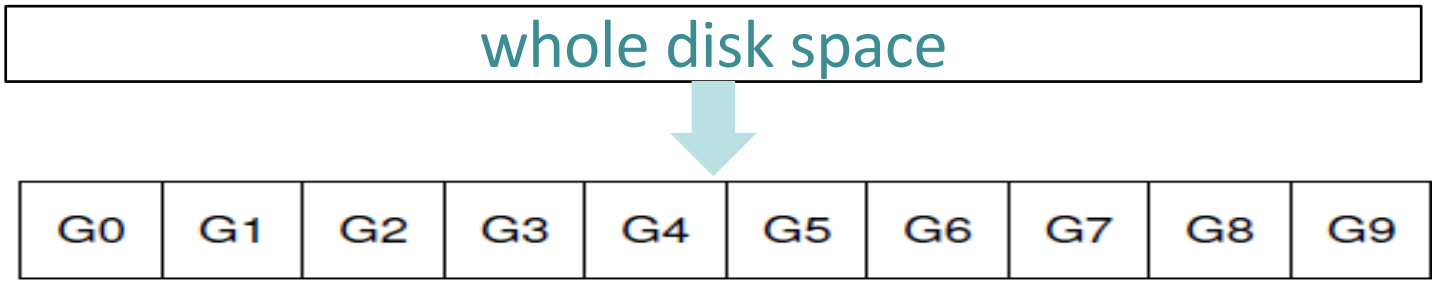
# Review

- File Systems
  - Caching & Buffering
    - Reading and writing files are expensive, incurring many I/Os
    - FSes use system memory (DRAM) to cache reads and buffer writes
      - **page cache** in Linux
      - FS can optimize the writes in memory, e.g.:
        - batch some updates into a smaller set of I/Os
        - avoiding unnecessary I/O (e.g., overwritten in memory)
  - Applications may force flushing dirty data to disk by calling `fsync()`

# Review

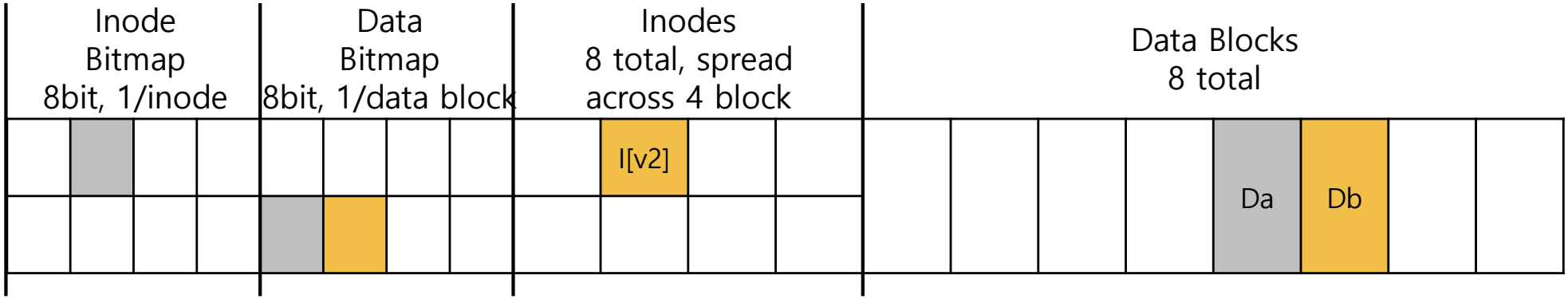
- File Systems
  - The 1st Unix File System (~1974)


  - The Fast File System (FFS, ~1984)
    - Key insight: disk awareness
      - data structures and allocation polices match the internals of disks
    - Divide the disk into cylinder groups (block groups)
      - place related stuff in the same group, avoid long seek
      - each group has a layout similar to the 1st Unix FS



# Review

- File Systems
  - Crash Consistency Problem
    - An user operation may generate multiple low-level writes that need to be committed atomically
    - Failure events may interrupt the writes and lead to inconsistency or corruption of FS



# Review

- File Systems
  - Two common techniques for data protection
    - Journaling
      - Also called Write-Ahead-Logging (WAL)
      - Basic Idea
        - Do not write to the main FS data structures directly
        - Write to a “journal” data structure first
        - Update the main FS data structures only after all relevant writes are safely stored in the journal
    - FSCK
      - file system checker
      - scan FS metadata, identify and fix inconsistencies between metadata structures
      - cannot fix all issues

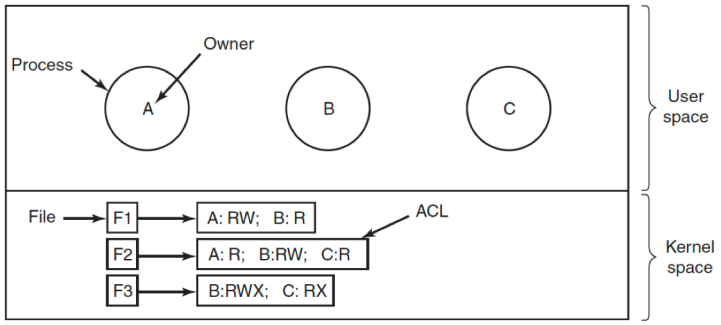
# Review

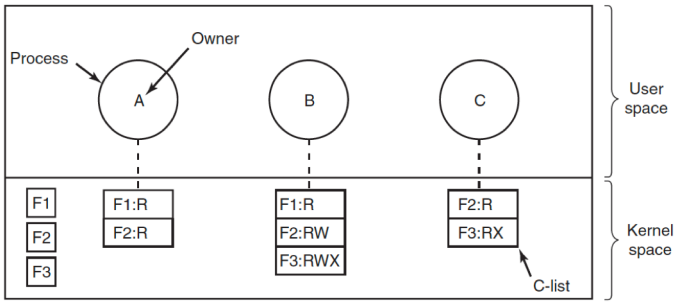
- Security
  - Classic Goals & Threats
    - Confidentiality
      - Having secrete data remain secret
    - Integrity
      - Unauthorized users should not be able to modify data
    - Availability
      - Nobody can disturb the system to make it unusable
  - Trusted Computing Base (TCB)
    - the (minimal) set of hardware and software necessary for enforcing all security rules

| Goal | Threat |
|---|---|
| Confidentiality | Exposure of data |
| Integrity | Tampering with data |
| Availability | Denial of service |

# Review

- Security
  - Access Control List (ACL)
    - associate with each object an (ordered) list containing all the domains that may access the object, and how
  - Capability List (C-list)
    - associated with each user/process a list of objects that may be accessed
      - individual items on a C-list are called **capabilities**



# Review

- Information Leakage Channels
  - Covert Channels
  - Steganography
- Cryptography
  - Secret-Key Cryptography
  - Public-Key Cryptography
    - Everyone picks a (public key, private key) pair and publishes the public key
    - User X sends a secret message to User Y
      - X encrypts the message using Y's public key
      - Y decrypts the message using Y's secret key
- Digital Signature

# Review

- Authentication
  - Prove the identity of a user
  - Based on three general principles:
    - Something the user knows.
    - Something the user has.
    - Something the user is
  - Common methods
    - Password
    - Hardware token
    - Software token
    - Biometrics
    - Multi-factor authentication

# Review

## parse.c

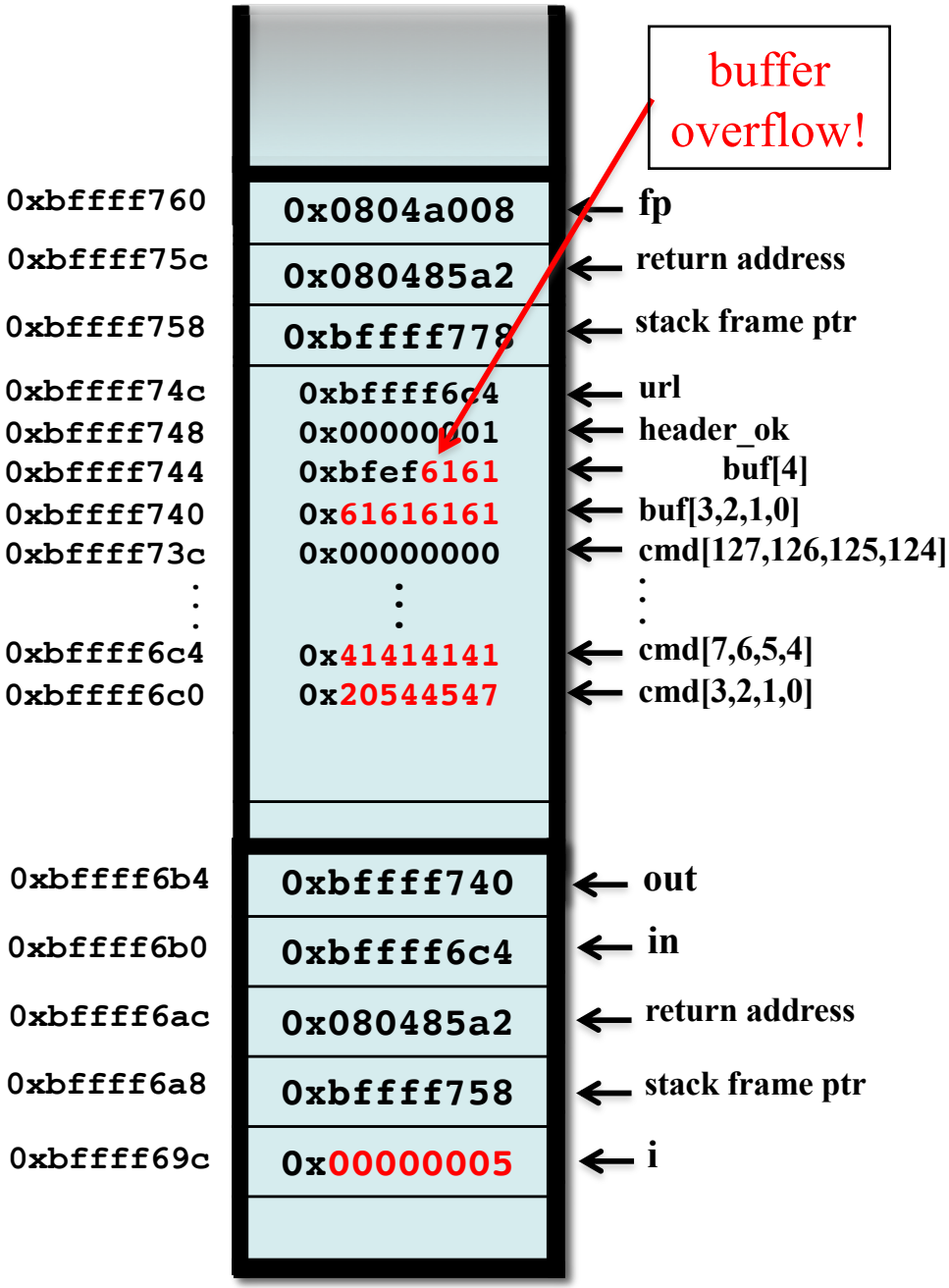
```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   out[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 128, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

## input file (read to cmd[128])

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



# Review

## parse.c

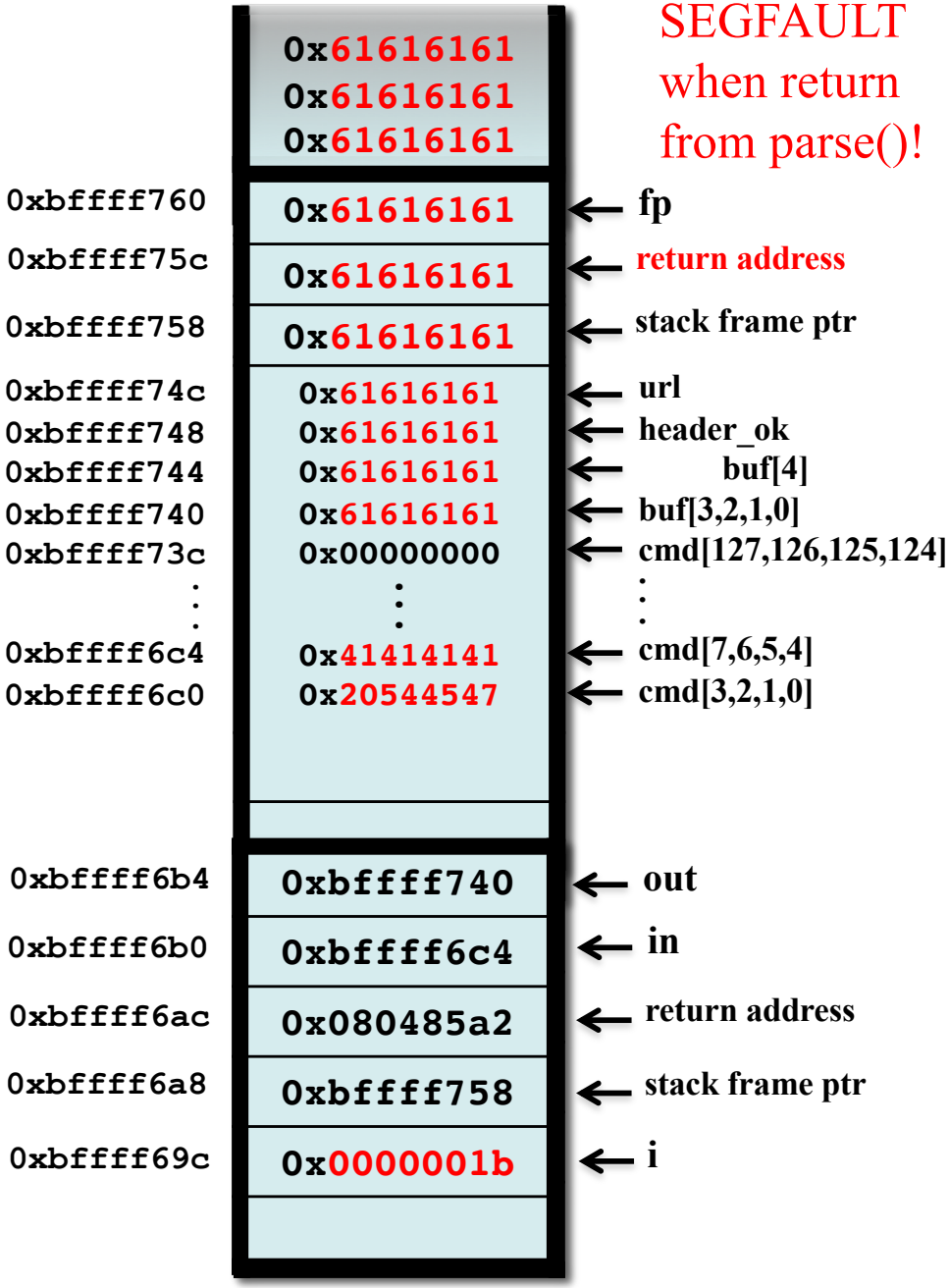
```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   out[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 128, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

## input file (read to cmd[128])

```
GET AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



# Review

## parse.c

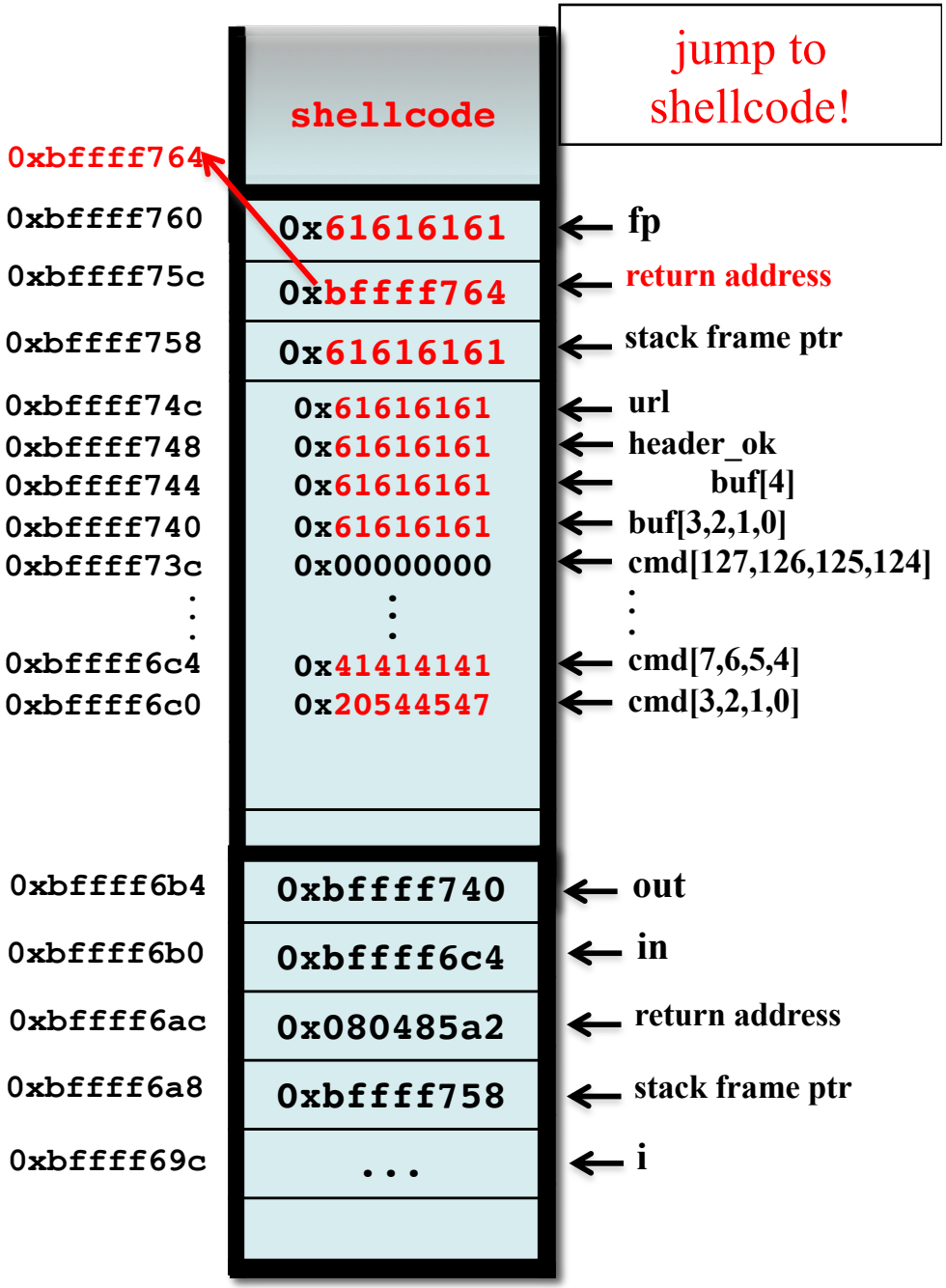
```
 1:void copy_lower (char* in, char* out) {
 2:   int i = 0;
 3:   while (in[i]!='\0' && in[i]!='\n') {
 4:     out[i] = tolower(in[i]);
 5:     i++;
 6:   }
 7:   out[i] = '\0';
 8:}

 9:int parse(FILE *fp) {
10:   char buf[5], *url, cmd[128];
11:   fread(cmd, 1, 128, fp);
12:   int header_ok = 0;
.
.
.
19:   url = cmd + 4;
20:   copy_lower(url, buf);
21:   printf("Location is %s\n", buf);
22:   return 0; }

23: /** main to load a file and run parse */
```

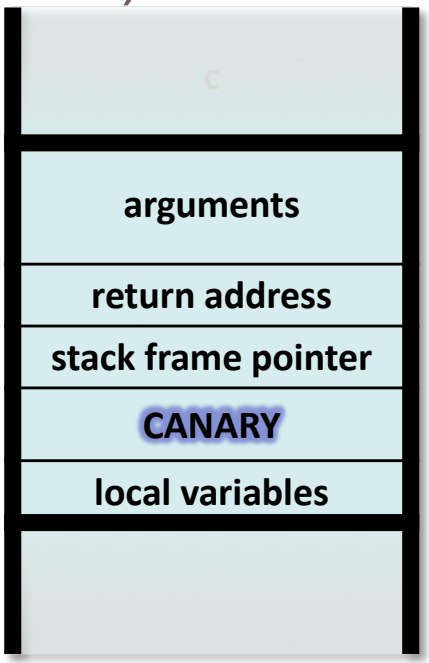
## input file (read to cmd[128])

```
GET AAAAAAAAAAAAAAAAAAAA\x64\xf7\xff\xbfAAAA
\xeb\x1f \x5e\x89\x76\x08\x31\xc0\x88\.....\xff\xff/bin/sh
```



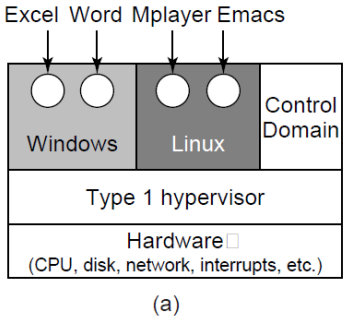
# Review

- Defense against buffer overflow attack
  - Non-execute
    - Prevent attack code execution by marking stack and heap as **non-executable**
  - Address Space Layout Randomization (ASLR)
    - Start stack/heap at a random location
    - Map shared libraries to random location
      - Attacker cannot jump directly to exec function
  - StackGuard
    - Embed “canary” in stack frames and verify ttheir integrity prior to function return

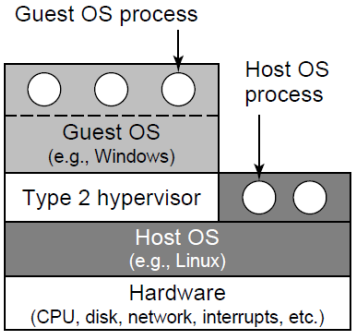


# Review

- Virtual Machine Monitor (VMM)
  - "Hypervisor"
    - Type 1: run on bare metal
    - Type 2: run on a host OS
  - Key responsibilities
    - Time-share CPU among guests
    - Space-share memory among guests
    - Simulate disk, network, and other devices



(a)



(b)

# Review

- Virtualization Techniques
  - Simulation
  - Trap-and-Emulate
  - Binary Translation
  - Hardware Support
- Paravirtualization
  - Do not aim to present a VM that looks just like the actual underlying hardware
    - expose a set of hypercalls to the guest
  - “Xen and the Art of Virtualization” [SOSP’03]

# Review

- Cloud Computing
  - Infrastructure as a Service (IaaS)
  - Platform as a Service (PaaS)
  - Software as a Service (SaaS)
  - Public vs. private clouds:
    - Shared across arbitrary organizations/customers vs. Internal to one organization
- The Google File System [SOSP'03]
  - distributed: master + chunkservers
  - fault tolerance: replication

# 563X Advanced Data Storage Systems!

- Similar Style to CPR E 588 in Spring 2019
  - “I just wanted to thank you for this semester in CPR E 588, and I thought it was a very good course. It really pushed me to learn more advanced topics in embedded systems that were not covered in any of my undergraduate courses like 488...”
  - “ ... At first I really struggled getting through them because I have never had to read academic papers before, but as the semester went on, I definitely got more comfortable with them and I have even started reading some other papers on my own.”
  - “Coming into the class, I had reservations about its structure. This probably was due to me never taking a "seminar" class before. However, by the end of the semester, this came to be one of the classes I looked forward to attending.”
  - “I was skeptical at first as this was a paper-based class, but I have learned a lot and by the end, I really enjoyed this class.”
  - “This was a great class and was in the top 5 I have taken at ISU”
  - Overall, the instructor has been an effective teacher: 4.75/5
  - Overall, this course has been effective in advancing my learning: 4.69/5

# Good Luck!