EE309
Lecture 2: EE209/EE485 Review

INSU YUN (윤인수)

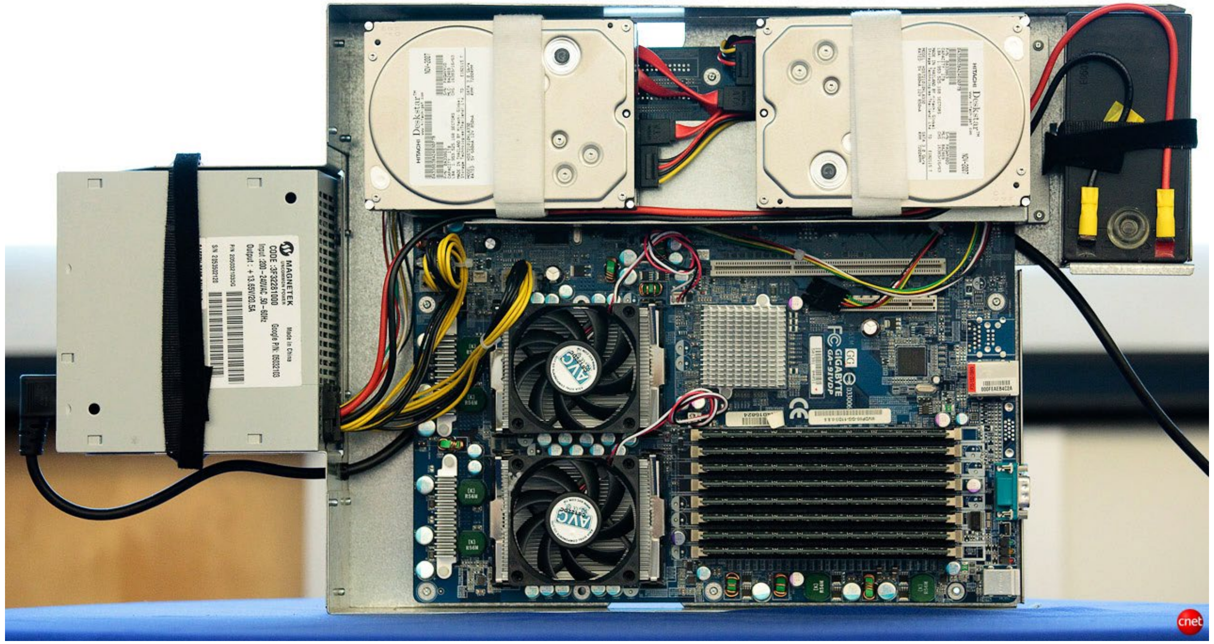
School of Electrical Engineering, KAIST

[Lecture slides based on EE209]

Lecture 1. Introduction

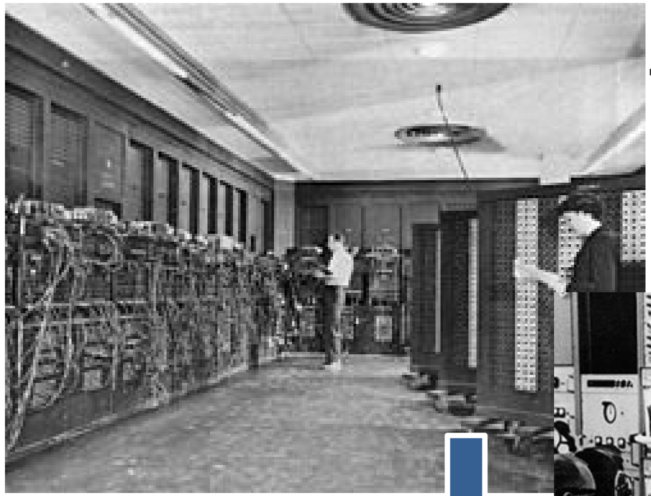
KAIST EE

What does a computer look like today?



- General purpose hardware (x86 architecture)
- Multicore: 4 ~ 60 cores (tens of CPU cores)
- Multiple 10-Gigabit Ethernet (becoming the norm)

Trend#1: Smaller and more powerful



ENIAC (1945)
The first computer

IBM mainframe (1969)
Used by NASA for Apollo 11 to land on the moon

Millions of times faster

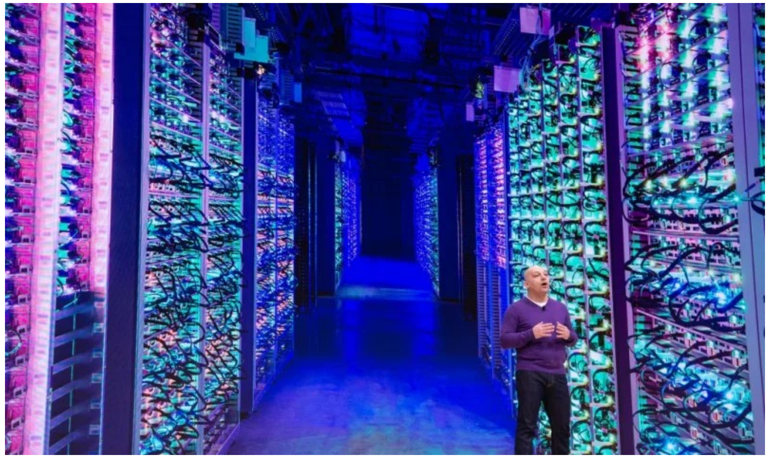
Smartphones today

Trend#2: Ubiquitous, everywhere

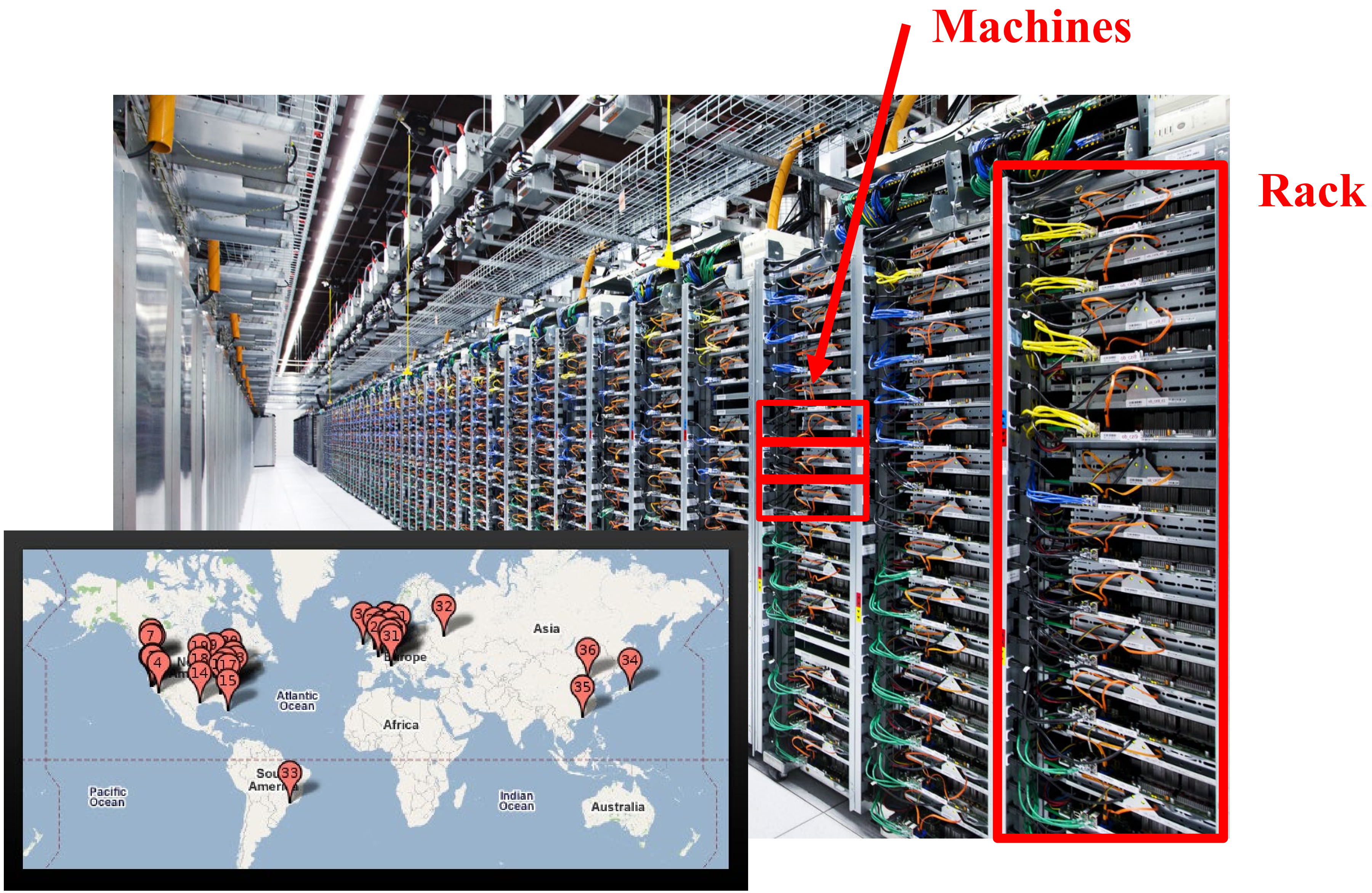
- Computers are dominating our lives!
- More things are becoming computers
  - Cars, watches, speakers, pets, … what’s next?

Trend#3: Growing to a larger scale

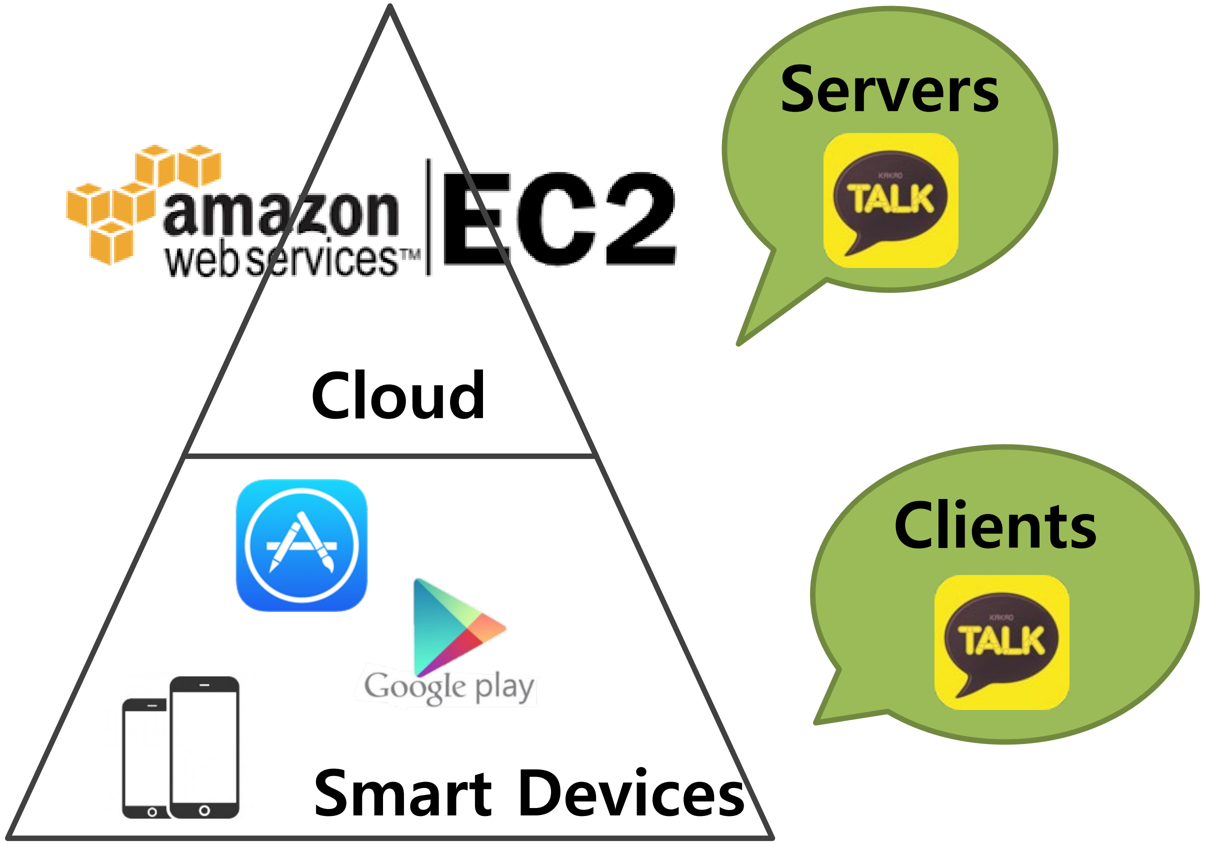
- Scale of the "Cloud"
  - Many machines spread out around the globe
  - Facebook: hundreds of thousands of machines
  - Microsoft: 4 million servers (~2021)
  - Google: 2.5 million servers in 2016
  - Amazon, Google, Facebook, and Microsoft spent $37B in 2020Q3



Google’s Datacenters



Internet-based Services



Hierarchical Structure
of Internet-based Services

Endless applications using cloud

Understanding Computer Systems and Software

- Cloud computing industry
- Software industry

- Not only required in software companies, but just about everywhere
  - Traditional semiconductor industry
    - SoC chip designers. Device manufacturing, …
  - Automobile industry
  - …

Design goals of C

- Support structured programming
- Support development of the Unix OS and tools
    - As Unix became popular, so did C
- Implications for C
    - Good for system-level programming
        - But also used for application-level programming
    - Low-level
        - Close to assembly language; close to machine language; close to hardware
    - Efficiency over portability

Hello World

```
hello.c:

#include <stdio.h>

int main(void) {
   printf("Hello, world\n");
   return 0;
}
```

Include information about standard library

main() function: entry point of execution

Statements of main are enclosed in braces

main() calls printf() function to print the string

$ gcc209 hello.c   (compile)

$ ./a.out   (execution)

C Variable Types

- Variable
  - Name given to a memory area that a program manipulates
  - Each variable has a type
- Character type
  - `char` (8 bit)
- Integral type
  - `short` (16 bit), `int` (32 bit), `long` (64 bit on 64-bit OS)
- Floating point type
  - `float` (32 bit), `double` (64 bit), `long double` (128 bit)
- Generic type
  - `void *` (64 bit on 64-bit OS)

```
char x = 'a';
int x = 10;
float x = 3.14;
```

Constants, Array, Pointer Type

- Constant: identifier whose value doesn't change

```
#define MAX 10
const int MAX = 10;
enum {MAX = 10};
```

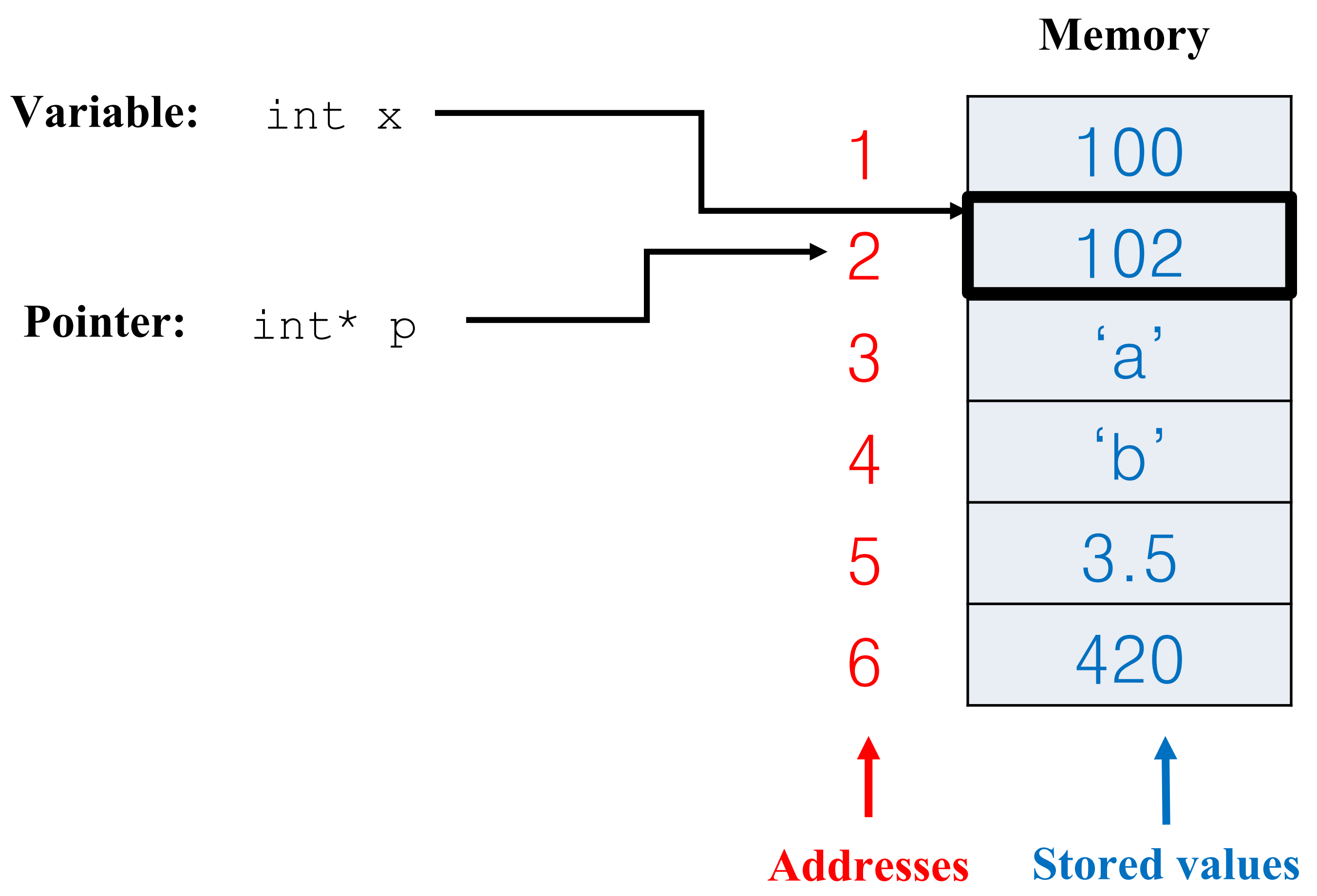
- Array: a collection of elements of the **same** type

```
char c[10];
double pi[5][2];
```

- Pointer: holds a memory address of a variable of some type

```
int *p;
```

Variables and Pointers



Strings and Structures

- String: a collection of characters

```
char *s = “hello world\n”;
char s[12] = “KAIST EE209”;
```

- Structure: a collection of elements whose types can be **different**

```
struct student {
   int id;
   char *name;
};
```

Arithmetic and Logic Operations

- Arithmetic operators
  - `+, -, *, /, %, unary -`
- Logic operators
  - `&&, ||, !`
- Relational operators
  - `==, !=, >, <, >=, <=`
- Bitwise operators
  - `>>, <<, &, |, ^`
- Assignment operators
  - `=, *=, /=, +=, -=, <<=, >>=, =, ^=, |=, %=`

https://www.tutorialspoint.com/cprogramming/c_operators.htm

Statements

- Statement
  - Statements are fragments of the C program that are executed in sequence.
  - Informally: a command that takes a specific action
  - Typically terminated by `;` (a terminator)
- Assignment

```
int i, j;
i = 10;
i = j = 0;
```

- `if` statement

```
if (i < 0)
  statement1;
else
  statement2;
```

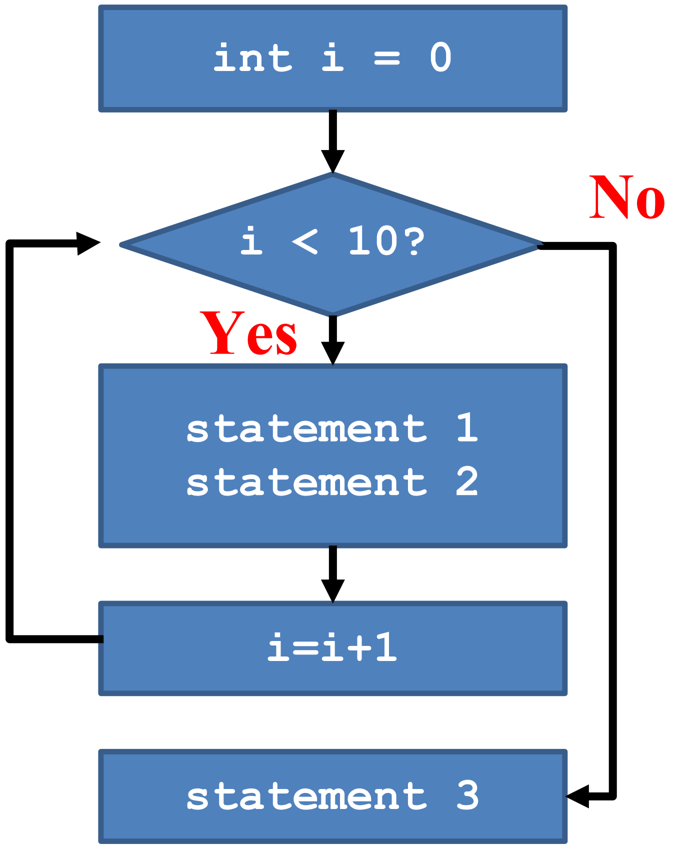
- `switch/case` statement

```
switch (i) {
case 1:
    statement1;
    break;
case 2:
    statement2;
    break;
default:
    statement3;
    break;
}
```

Loop Statements (1)

- **for** statement

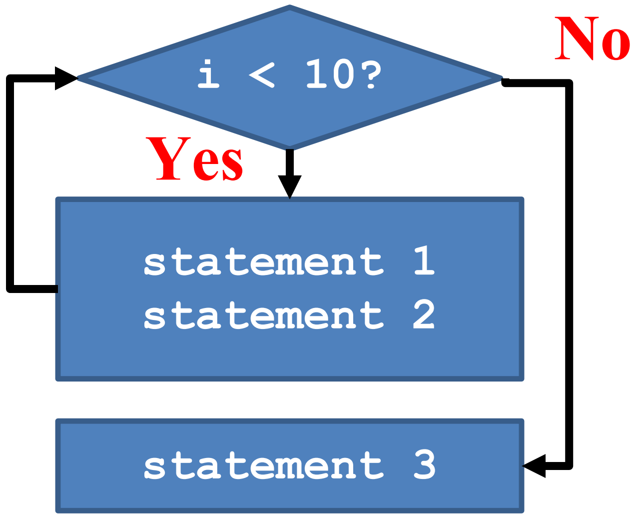
```
for (int i = 0; i < 10; i++){
    statement 1;
    statement 2;
}
statement 3;
```



Loop Statements (2)

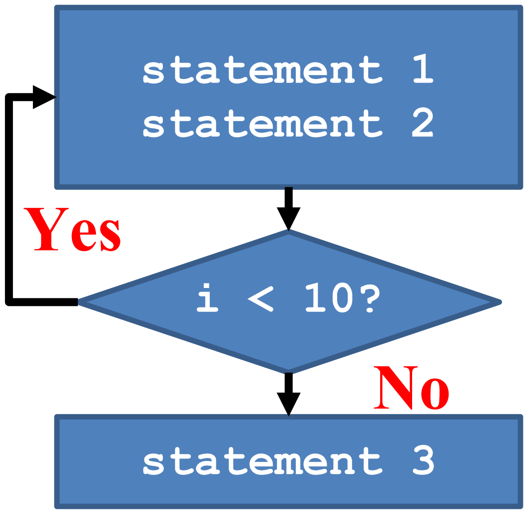
- **while** statement

```
while (i < 10){
    statement 1;
    statement 2;
}
statement 3;
```



- **do while** statement

```
do {
    statement 1;
    statement 2;
} while (i < 10)
statement 3;
```



Loop Statements (3)

- **`break;`** // get out of the current loop/switch

```
while (i < 10) {
    statement1;
    statement2;
    break;
}
statement3;
```

- **`continue;`** // go to the start of the next round

```
while (i < 10) {
    statement1;
    statement2;
    continue;
}
statement3;
```

- **`goto`** `SomeLabel;`

Function Definition and Call

- Function Definition with a Return Statement

```
int add(int x, int y) {
   return x+y;
}
```

- Function Call

```
int sum = add(3,5);
```

Other Statements

- Compound Statements

```
{
    statement1;
    statement2;
}
```

- Comments // for readers, ignored by machines

```
/*
    multiple
    line
    comment
*/
```

```
// single line comment
```

Lecture 4: Compiler

KAIST EE

Building a C Program

- hello.c

```
#include <stdio.h>
int main(void)
{
    /* Write "hello, world\n" to stdout. */
    printf("hello, world\n");
    return 0;
}
```

- Compile and execute hello.c

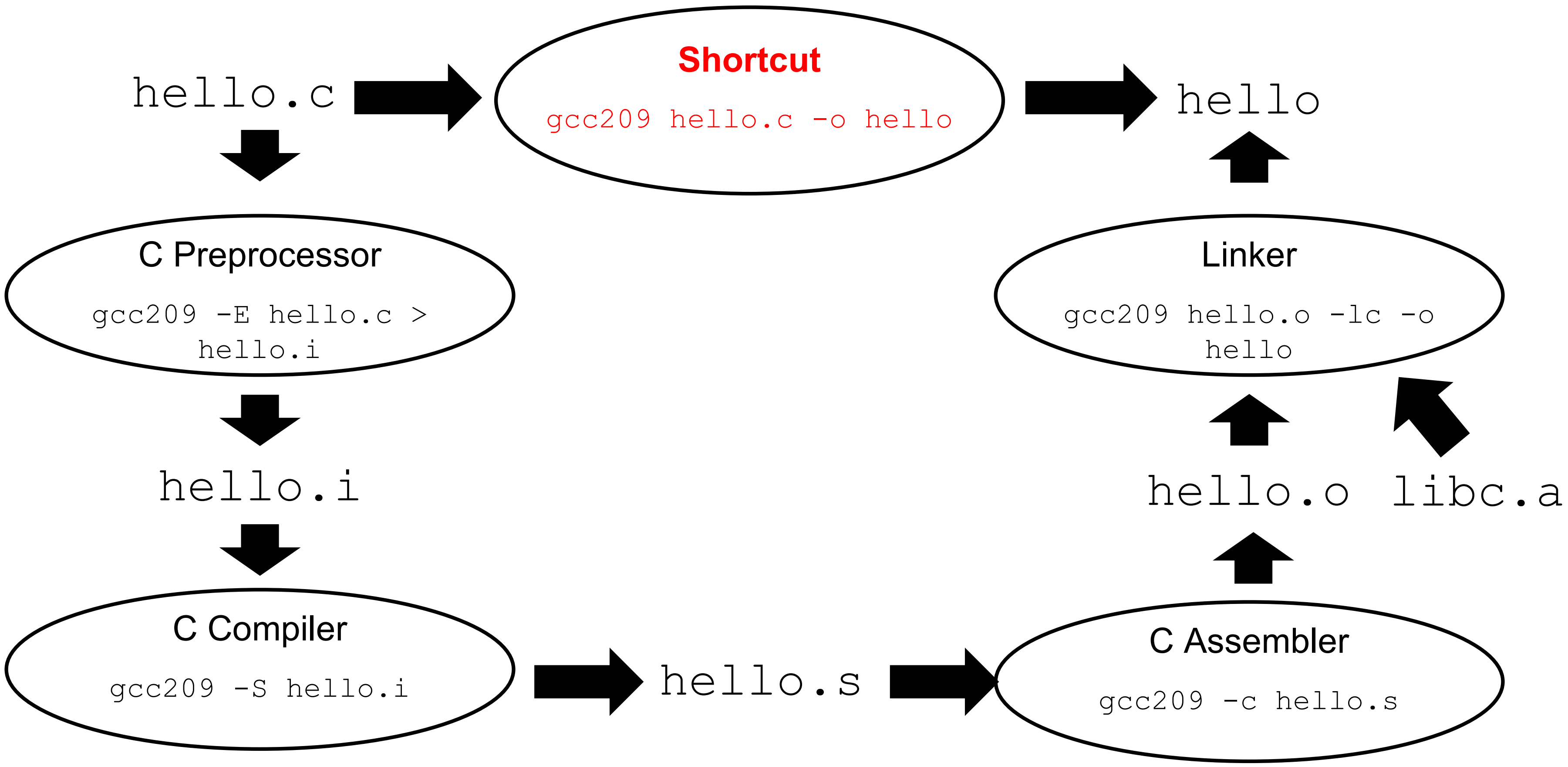
```
ee209@ubuntu:~$ gcc209 hello.c -o hello
ee209@ubuntu:~$ ./hello
hello, world
```

`gcc209` is a script that executes

```
gcc -Wall -Werror -ansi -pedantic -std=c99
```

- -Wall: all warnings
- -Werror: Make all warnings errors
- -ansi: Same as -std=c89
- -pedantic: Follow the ISO C standard specified by -std
- -std=c99: use c99 standard

Shortcut of All Processes



Lecture 5: Debuggers

KAIST EE

(Reference: The ART OF DEBUGGING with GDB, DDD, and ECLIPSE (TAD))

Typical Steps for Debugging with GDB

(a) Build with –g

```
(gdb) gcc -g insertsort.c -o insertsort
```

- Adds extra information to executable file that GDB uses
- Debugging symbols (e.g., line numbers, variable names, etc.)

(b) Run GDB in a different terminal

```
$ gdb insertsort
```

You can run GDB inside Emacs or VIM as well

(c) Set breakpoints, as desired

- the program would stop at each breakpoint when it's executed

```
(gdb) break main
```

- GDB sets a breakpoint at the first executable line of main()

```
(gdb) break process_data
```

- GDB sets a breakpoint at the first executable line of process_data()

Typical Steps for Debugging with GDB (cont.)

(d) Run (or continue) the program

`(gdb) run`

- GDB stops at the breakpoint in main()

`(gdb) continue`

- GDB stops at the breakpoint in process_data()

(e) Step through the program, as desired

`(gdb) step` (repeatedly)

  - GDB executes the next line (repeatedly)

- Note: When next line is a call of one of your functions:
  - `step` command *steps into* the function
  - `next` command *steps over* the function, that is, executes the next line without stepping into the function

Typical Steps for Debugging with GDB (cont.)

(f) Examine variables, as desired

```
(gdb) print i
(gdb) print j
(gdb) print temp
```

- GDB prints the value of each variable

(g) Examine the function call stack, if desired

```
(gdb) where
```

- GDB prints the function call stack
- Useful for diagnosing crash in large program

(h) Exit gdb

```
(gdb) quit
```

Other Useful Tips

- How to run with command-line arguments?

```
(gdb) run arg1 arg2
```

- How to handle redirection of stdin, stdout, stderr?

```
(gdb) run < somefile > someotherfile
```

- Print values of expressions (later)
- Break conditionally (later)
- Materials so far are enough for basic usage of GDB

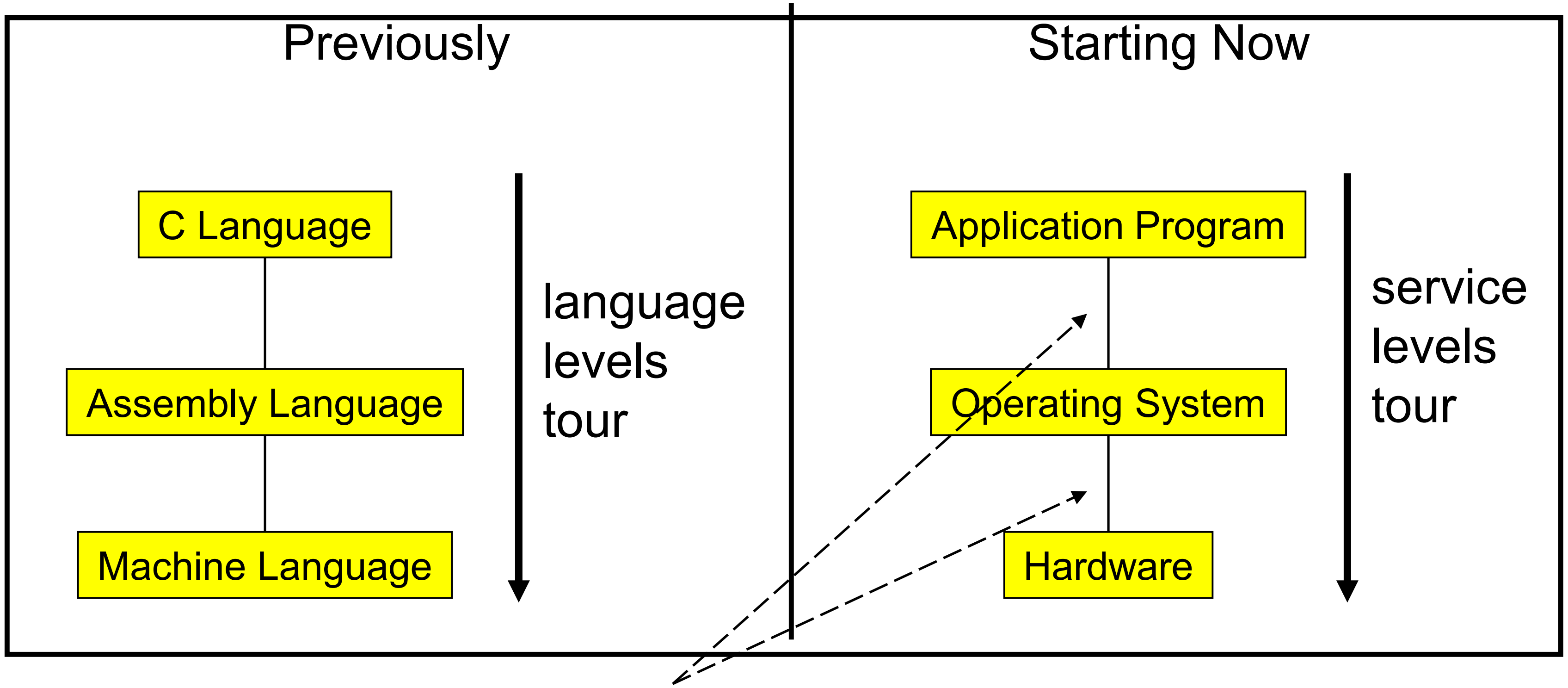
Lecture 16: Exceptions and Processes

KAIST EE

The material for this lecture is drawn from
Computer Systems: A Programmer's Perspective **(Bryant & O'Hallaron) Chapter 8**

Context of this Lecture

Second half of the course



Application programs, OS,
and hardware interact
via exceptions

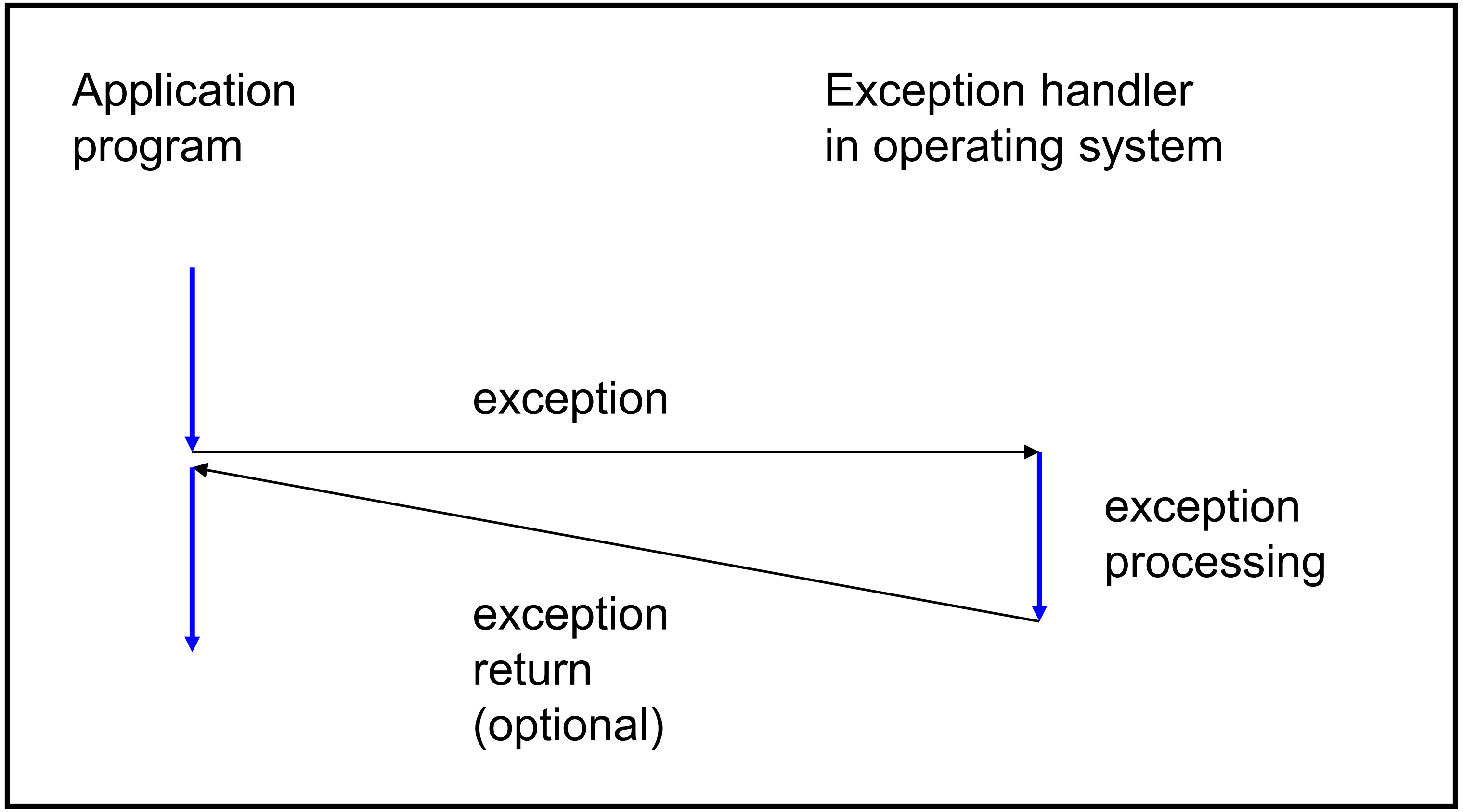
Exceptions

- **Exception**
  - An abrupt change in control flow in response to a change in processor state
  - Transfers control to OS
- Examples:
  - Application program:
    - Requests I/O
    - Requests more heap memory
    - Attempts integer division by 0
    - Attempts to access privileged memory
    - Accesses variable that is not in real memory (see upcoming "Memory Management" lecture)
  - User presses key on keyboard
  - Disk controller finishes reading data

Synchronous (i.e., caused by the execution of the current instruction)

Asynchronous

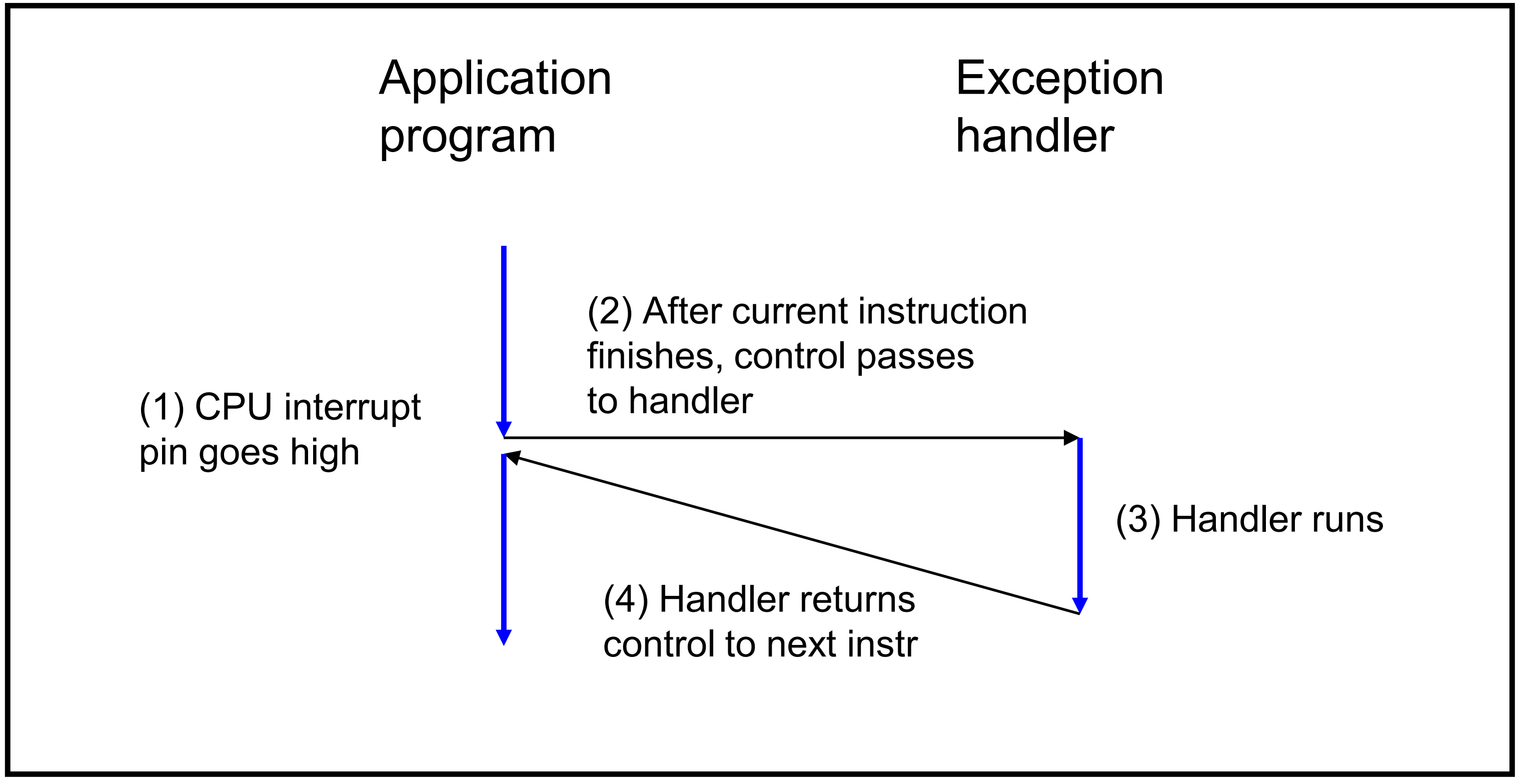
Exceptional Control Flow



Classes of Exceptions

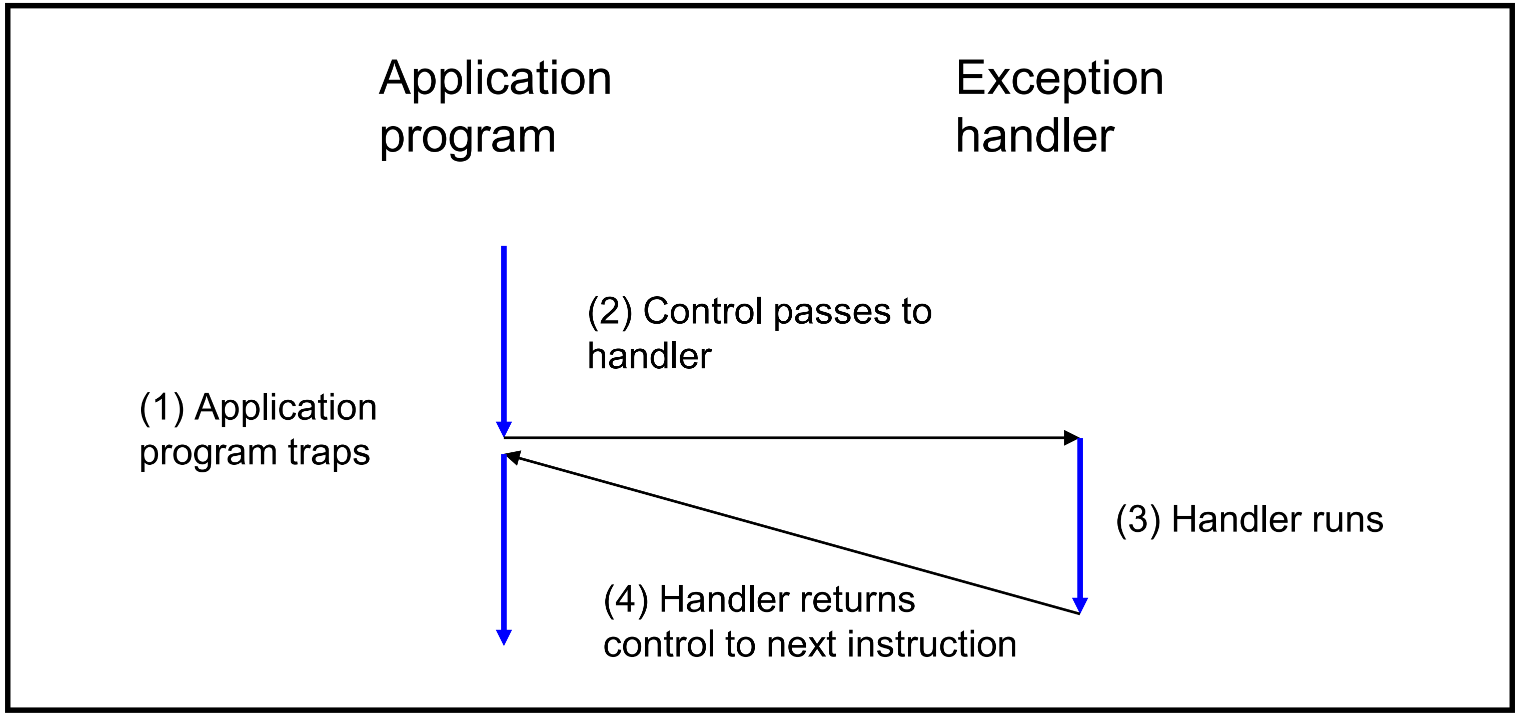
- There are 4 classes of exceptions
  1. Interrupts
  2. Traps
  3. Faults
  4. Aborts

(1) Interrupts



Cause: Signal from I/O device (asynchronously)
Examples:
  User presses key
  Disk controller finishes reading/writing data

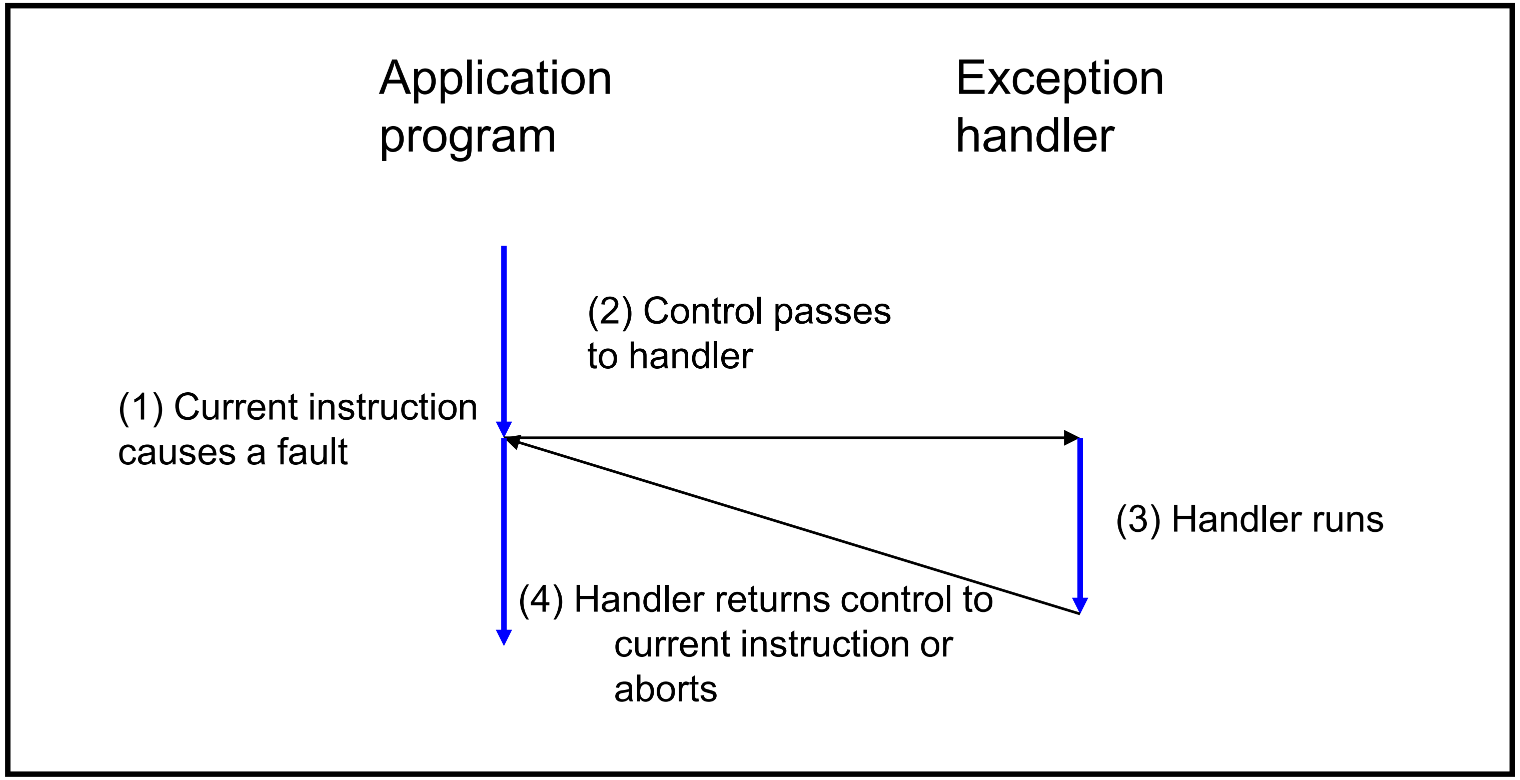
(2) Traps



Cause: Intentional (application program requests OS service)
Examples:
  Application program requests more heap memory
  Application program requests I/O

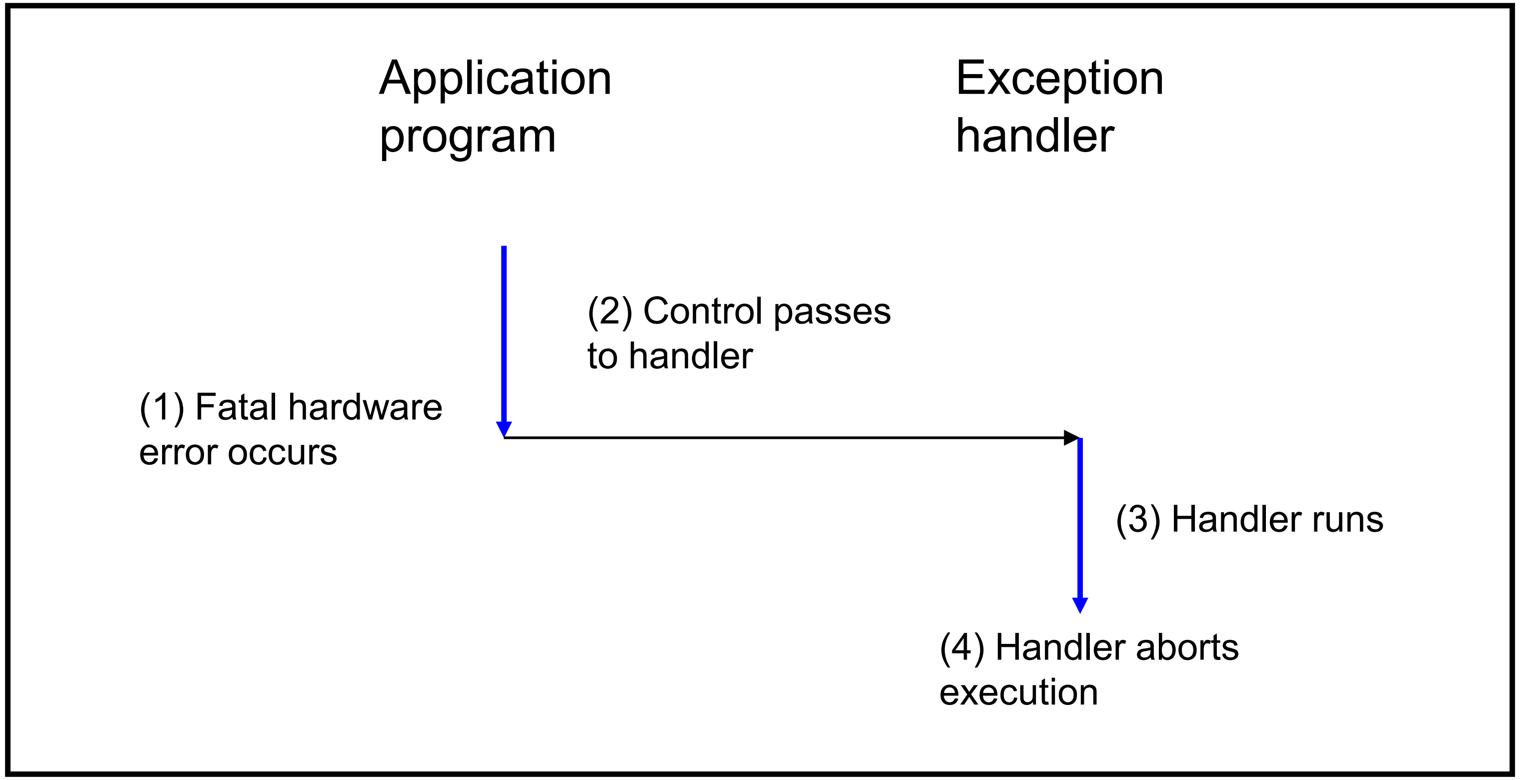
Traps provide a function-call-like interface between application program and OS

(3) Faults



Cause: Application program causes (possibly) recoverable error
Examples:
- Application program accesses privileged memory (seg fault)
- Application program accesses data that is not in real memory (page fault)

(4) Aborts



Cause: Non-recoverable error
Example:
Parity check indicates corruption of memory bit (overheating, cosmic ray!, etc.)

Traps in Intel Processors

- To execute a trap, application program should:
  - Place number in EAX register indicating desired functionality
  - Place parameters in EBX, ECX, EDX registers
  - Execute assembly language instruction “int 128”
- Example: To request more heap memory…

In Linux, 45 indicates request for more heap memory

```
movl    $45, %eax
movl    $1024, %ebx
int     $128
```

Causes trap

Request is for 1024 bytes

System-Level Functions

- For convenience, traps are wrapped in **system-level functions**
- Example: To request more heap memory…

```
/* unistd.h */
void *sbrk(intptr_t increment);
…
```

`sbrk()` is a system-level function

```
/* unistd.s */
Defines sbrk() in assembly lang
Executes int instruction
…
```

```
/* client.c */
…
sbrk(1024);
…
```

A call of a system-level function, that is, a system call

See Appendix for list of some Linux system-level functions

Processes

- **Program**
  - Executable code
- **Process**
  - An instance of a program in execution

Processes

- **Program**
  - Executable code
- **Process**
  - An instance of a program in execution
- Each program runs in the **context** of some process

Processes

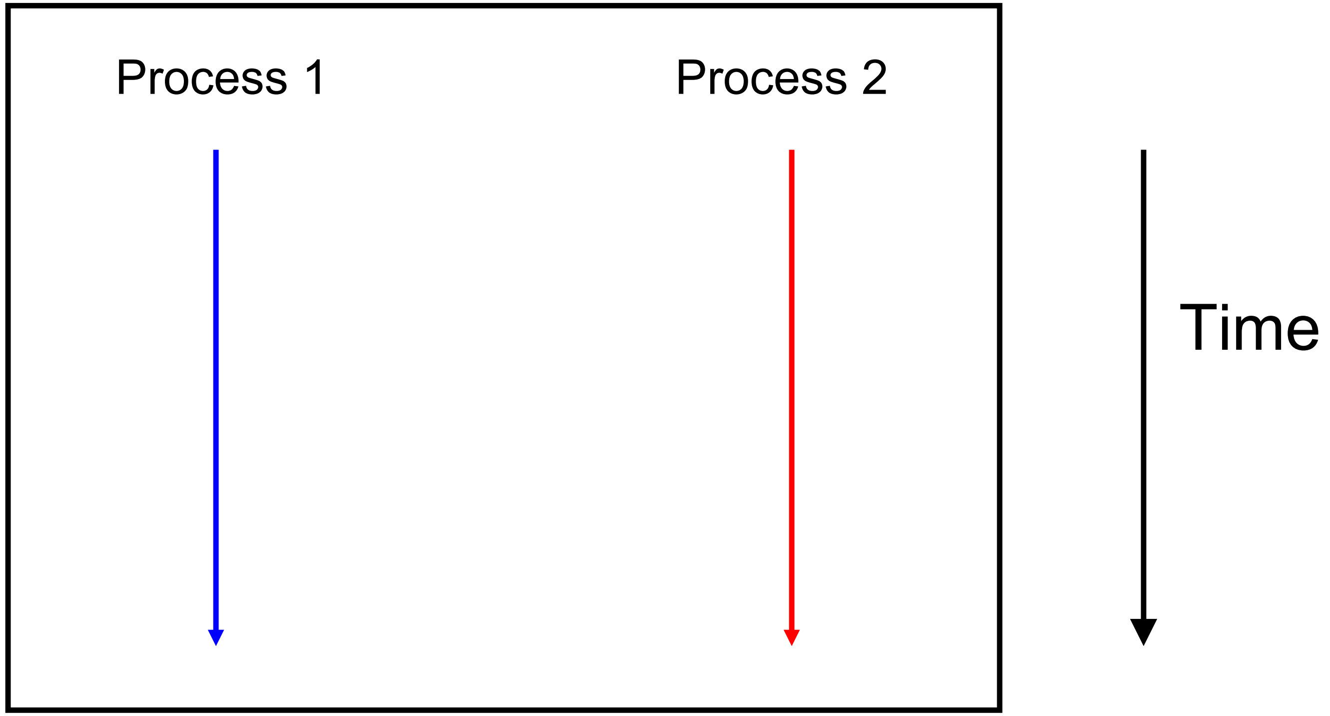
- **Program**
  - Executable code
- **Process**
  - An instance of a program in execution
- Each program runs in the **context** of some process
- **Context** consists of:
  - Process ID
  - Address space
    - TEXT, RODATA, DATA, BSS, HEAP, and STACK
  - Processor state
    - EIP, EFLAGS, EAX, EBX, etc. registers
  - etc.

Significance of Processes

- **Process** is a profound abstraction

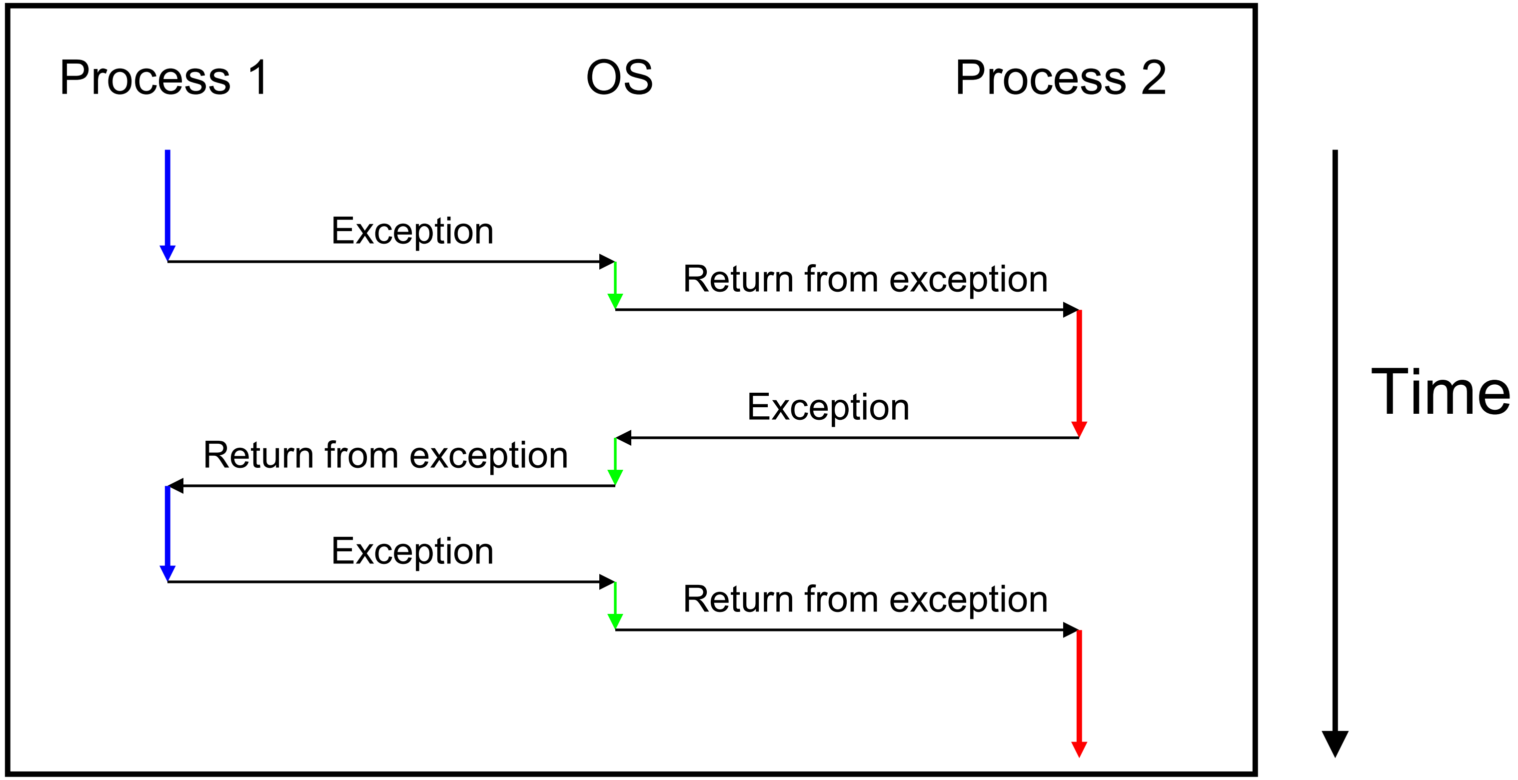
- The process abstraction provides application programs with two key illusions:
  - Private control flow
  - Private address space

Private Control Flow: Illusion



Hardware and OS give each application process the illusion that it is the only process running on the CPU

Private Control Flow: Reality



All application processes -- and the OS process -- share the same CPU(s) (i.e., multitasking, time slicing)

Context Switches

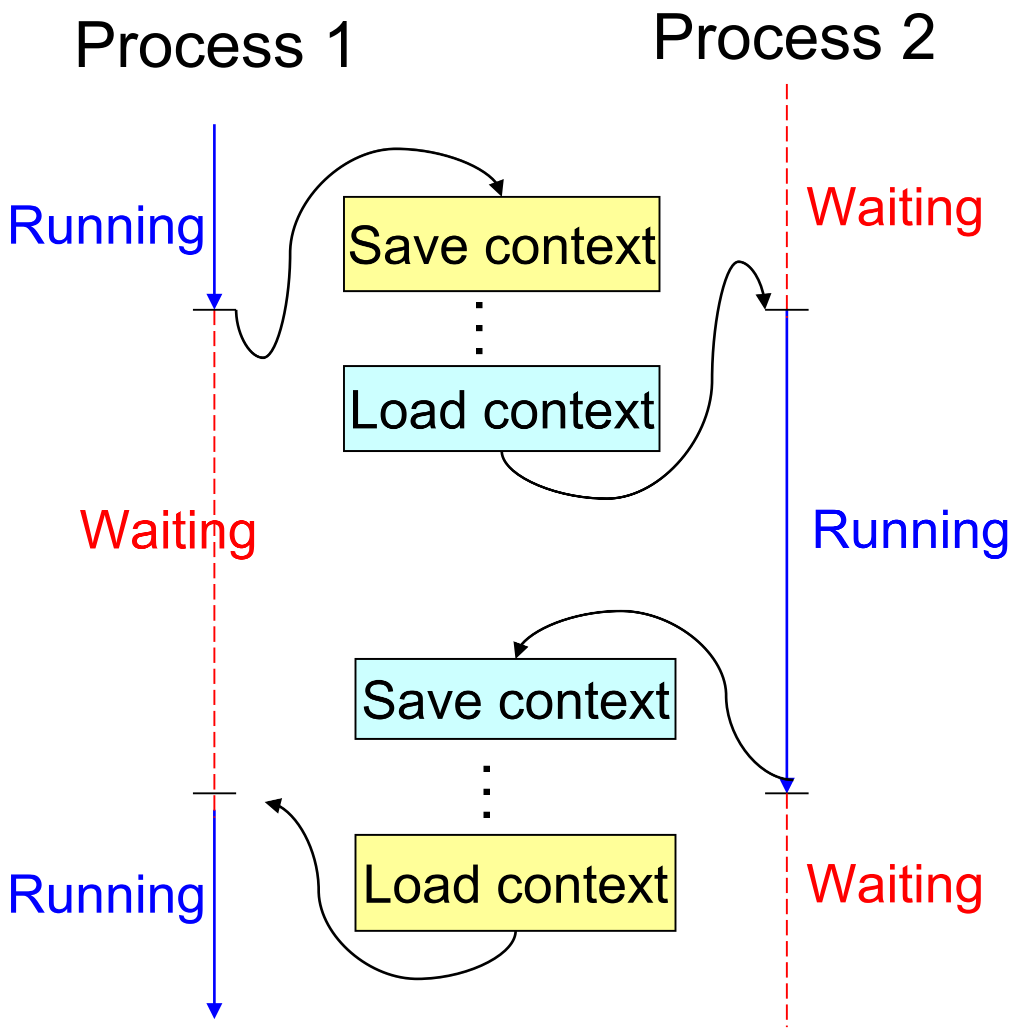
- **Context switch**
  - The activity whereby the OS assigns the CPU to a different process
  - Occurs during exception handling, at the discretion of OS
- Exceptions can be caused:
  - Synchronously, by application program (trap, fault, abort)
  - Asynchronously, by external event (interrupt)
  - **Asynchronously, by hardware timer**
    - So no process can dominate the CPUs
- Exceptions are the mechanism that enables the illusion of private control flow

Context Details

- What does the OS need to save/restore during a context switch?
  - Process state
    - New, ready, waiting, terminated
  - CPU registers
    - EIP, EFLAGS, EAX, EBX, …
  - I/O status information
    - Open files, I/O requests, …
  - Memory management information
    - Page tables (see “Memory Management” lecture)
  - Accounting information
    - Time limits, group ID, ...
  - CPU scheduling information
    - Priority, queues

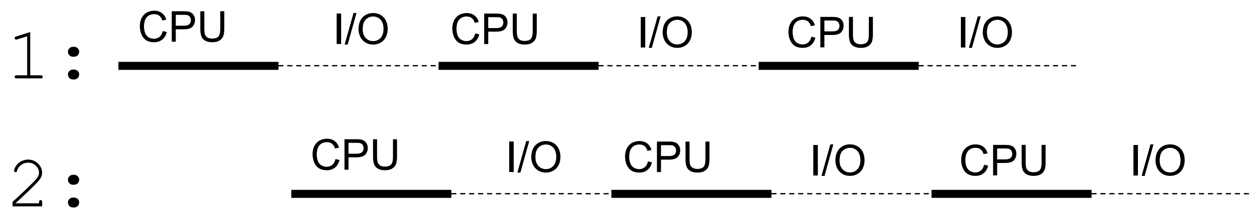
Context Switch Details

- Context
  - State that the OS needs to restart a preempted process
- Context switch
  - Save the context of current process
  - Restore the saved context of some previously preempted process
  - Pass control to this newly restored process



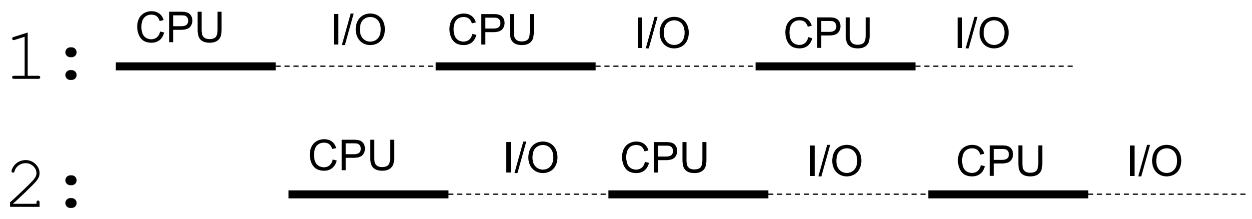
When Should OS Do Context Switch?

- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk access



When Should OS Do Context Switch?

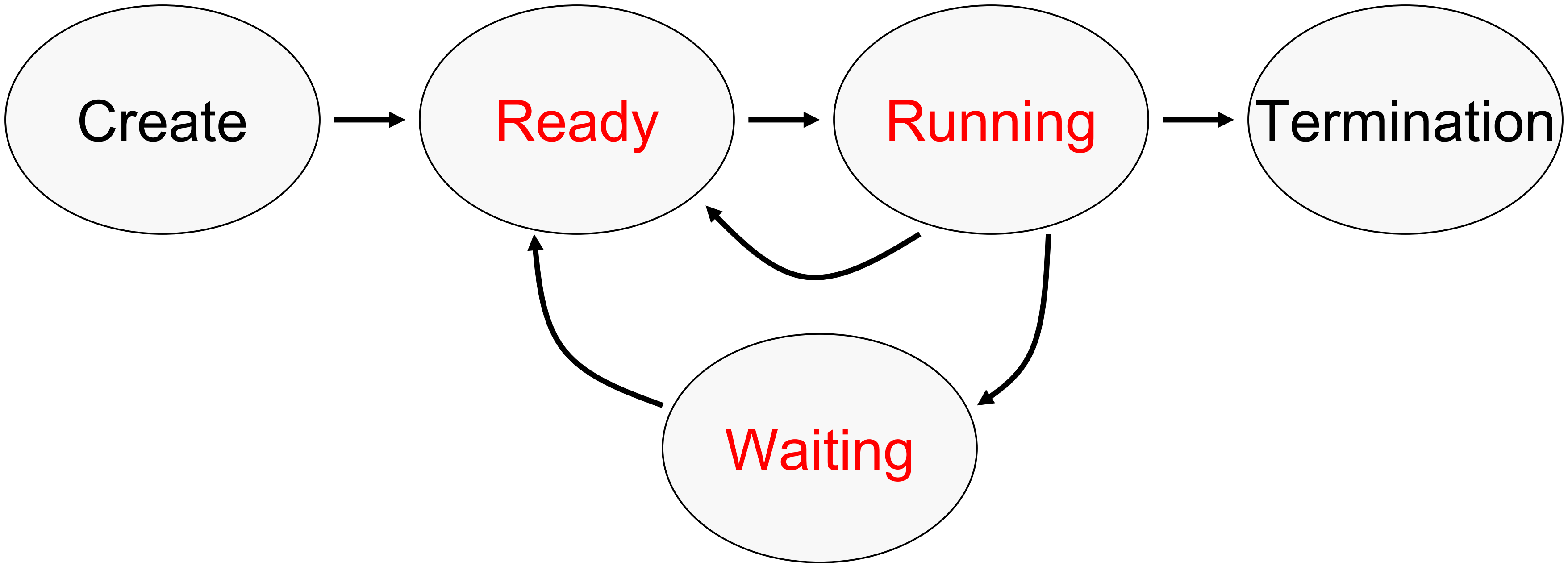
- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk access



- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine
  - Trade-off efficiency for a finer granularity of fairness

Life Cycle of a Process

- **Running**: instructions are being executed
- **Waiting**: waiting for some event (e.g., I/O finish)
- **Ready**: ready to be assigned to a processor



Lecture 17:
Memory Management

KAIST EE

Motivation for Memory Hierarchy

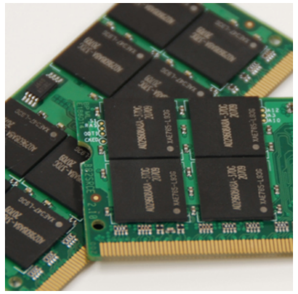
- ***Faster*** storage technologies are ***more expensive***
  - Cost more money per byte
  - Have lower storage capacity
  - Require more power and generate more heat
- The gap between processing and memory is widening
  - Processors have been getting faster and faster
  - Memory speed is not improving as dramatically
- Well-written programs tend to exhibit ***good locality***
  - Across time: repeatedly referencing the same variables
  - Across space: often accessing other variables located nearby

Want the ***speed*** of fast storage with the ***cost*** and ***capacity*** of slow storage

Key idea: ***memory hierarchy***!

Simple Three-Level Hierarchy

- **Registers**
  - Usually reside directly on the processor chip
  - Essentially no latency, referenced directly in instructions
  - Low capacity (e.g., 32-512 bytes)
- **Main memory**
  - Around 100 times slower than a clock cycle
  - Constant access time for any memory location
  - Modest capacity (e.g., 1 GB-512GB)

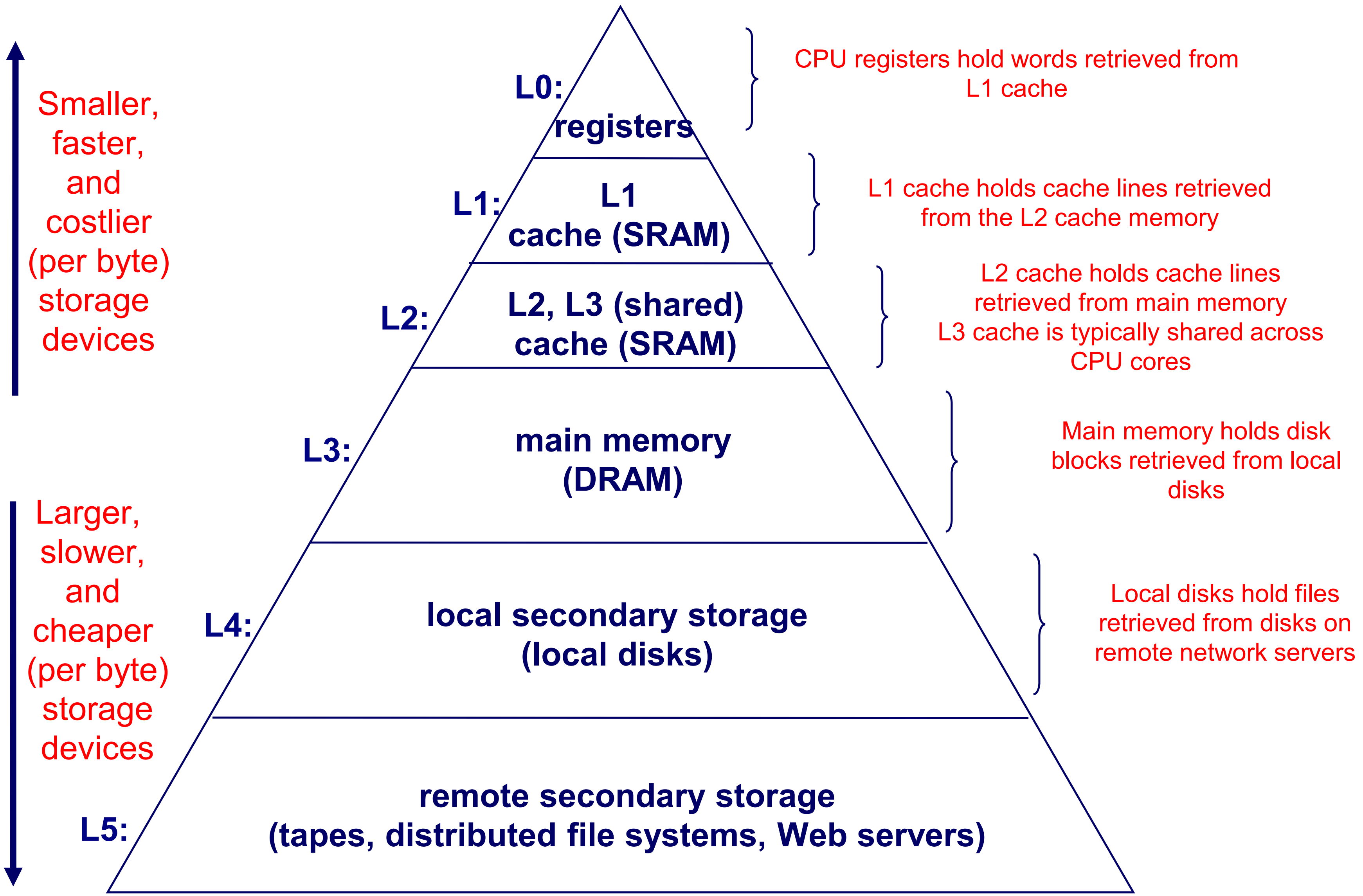


- **Disk**
  - Around 100,000 times slower than main memory
  - Faster when accessing many bytes in a row
  - High capacity (e.g., 1-10s of TB)

Widening Processor/Memory Gap

- Gap in speed increasing from 1986 to 2000
  - CPU speed improved ~55% per year
  - Main memory speed improved only ~10% per year
- Main memory as major performance bottleneck
  - Many programs stall waiting for reads and writes to finish
- Changes in the memory hierarchy
  - Increasing the number of registers
    - 8 integer registers in the x86 vs 16 in x86_64
  - Adding caches between registers and main memory
    - Level-1, -2, -3 cache on chip

An Example Memory Hierarchy



Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

- Locality example

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Locality example
  - Program data
    - Temporal: the variable **sum**
    - Spatial: variable `a[i+1]` accessed soon after `a[i]`

Locality of Reference

- Two kinds of locality
  - **Temporal locality**: recently referenced items are likely to be referenced in near future
  - **Spatial locality**: items with nearby addresses tend to be referenced close together in time

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

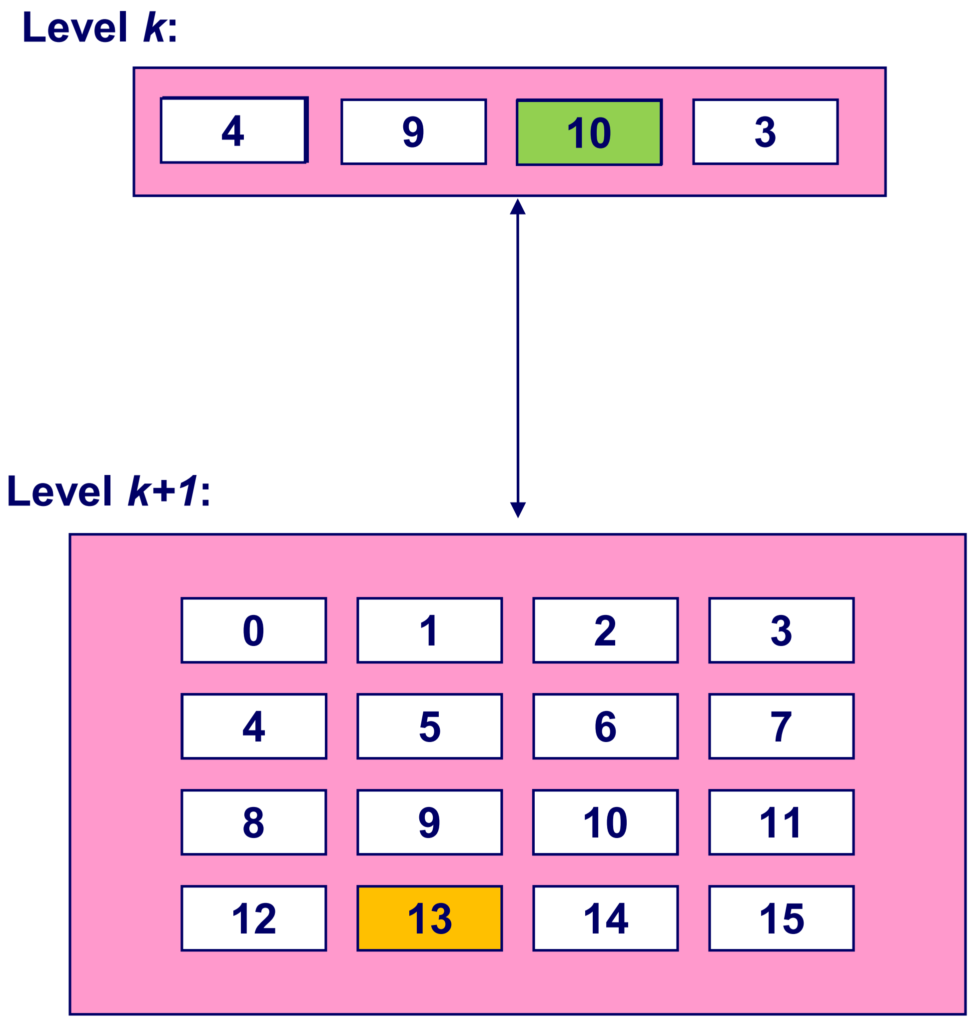
- Locality example
  - Program data
    - Temporal: the variable **sum**
    - Spatial: variable `a[i+1]` accessed soon after `a[i]`
  - Instructions
    - Temporal: cycle through the for-loop repeatedly
    - Spatial: reference instructions in sequence

Locality Makes Caching Effective

- Cache
  - Smaller and faster storage device that acts as a staging area
  - … for a *subset* of the data in a larger, slower device
- Caching and the memory hierarchy
  - Storage device at level *k* is a cache for level *k+1*
  - Registers as cache of L1/L2 cache and main memory
  - Main memory as a cache for the disk
  - Disk as a cache of files from remote storage
- ***Locality*** of access is the key
  - Most accesses satisfied by first few (faster) levels
  - Very few accesses go to the last few (slower) levels

Cache Hit and Miss

- Cache hit
  - Program accesses a block available in the cache
  - Satisfy directly from cache
  - e.g., request for “10”
- Cache miss
  - Program accesses a block not available in the cache
  - Bring item into the cache
  - e.g., request for “13”
- Where to place the item?
- Which item to evict?

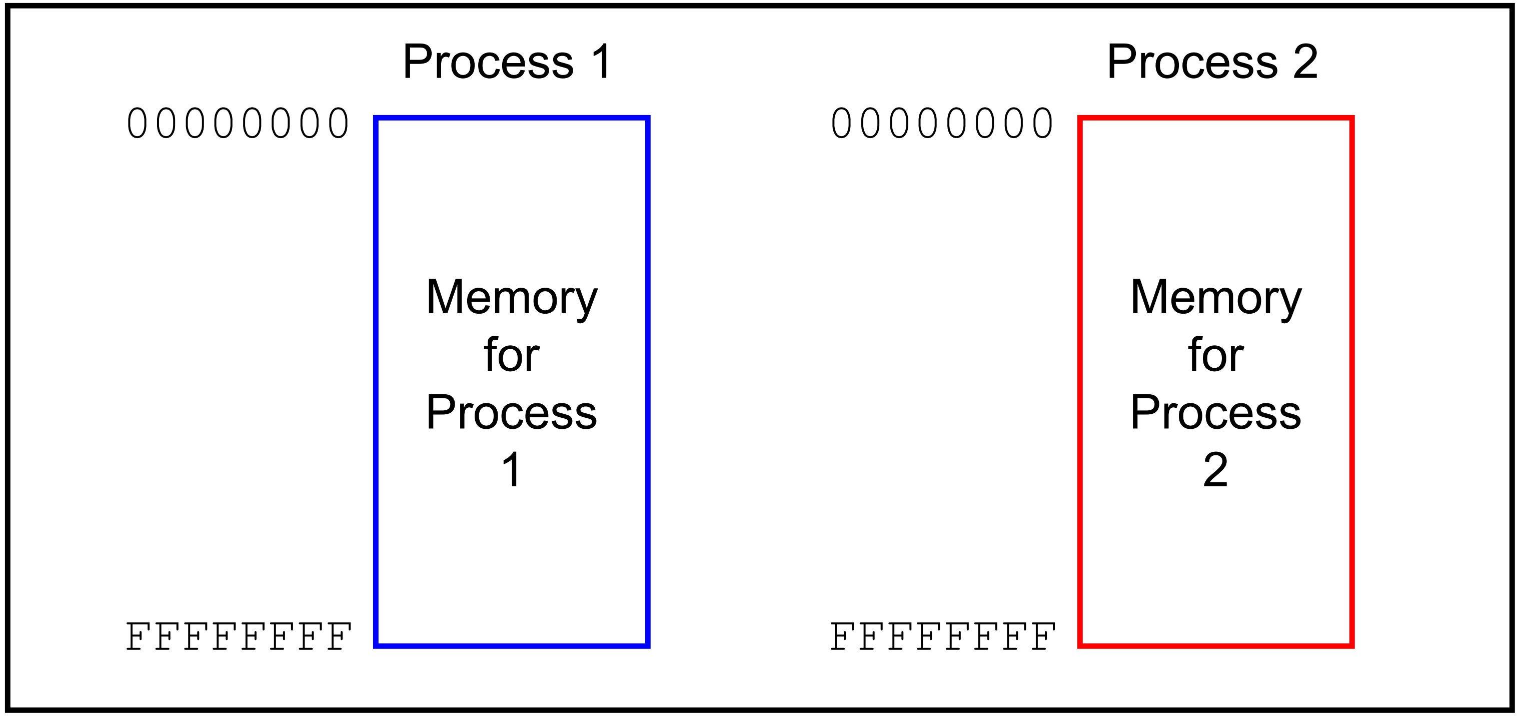


Automatic Allocation: Virtual Memory

- Give programmer the illusion of a very large memory
  - Large: 4 GB of memory with 32-bit addresses
  - Uniform: contiguous memory locations, from 0 to 2^{32}-1
- Independent of
  - the actual size of the main memory
  - the presence of any other processes sharing the computer
- **Key idea #1**: separate “address” from “physical location”
  - Virtual addresses: generated by the program
  - Memory locations: determined by the hardware and OS
- **Key idea #2**: caching
  - Swap virtual pages between main memory and the disk

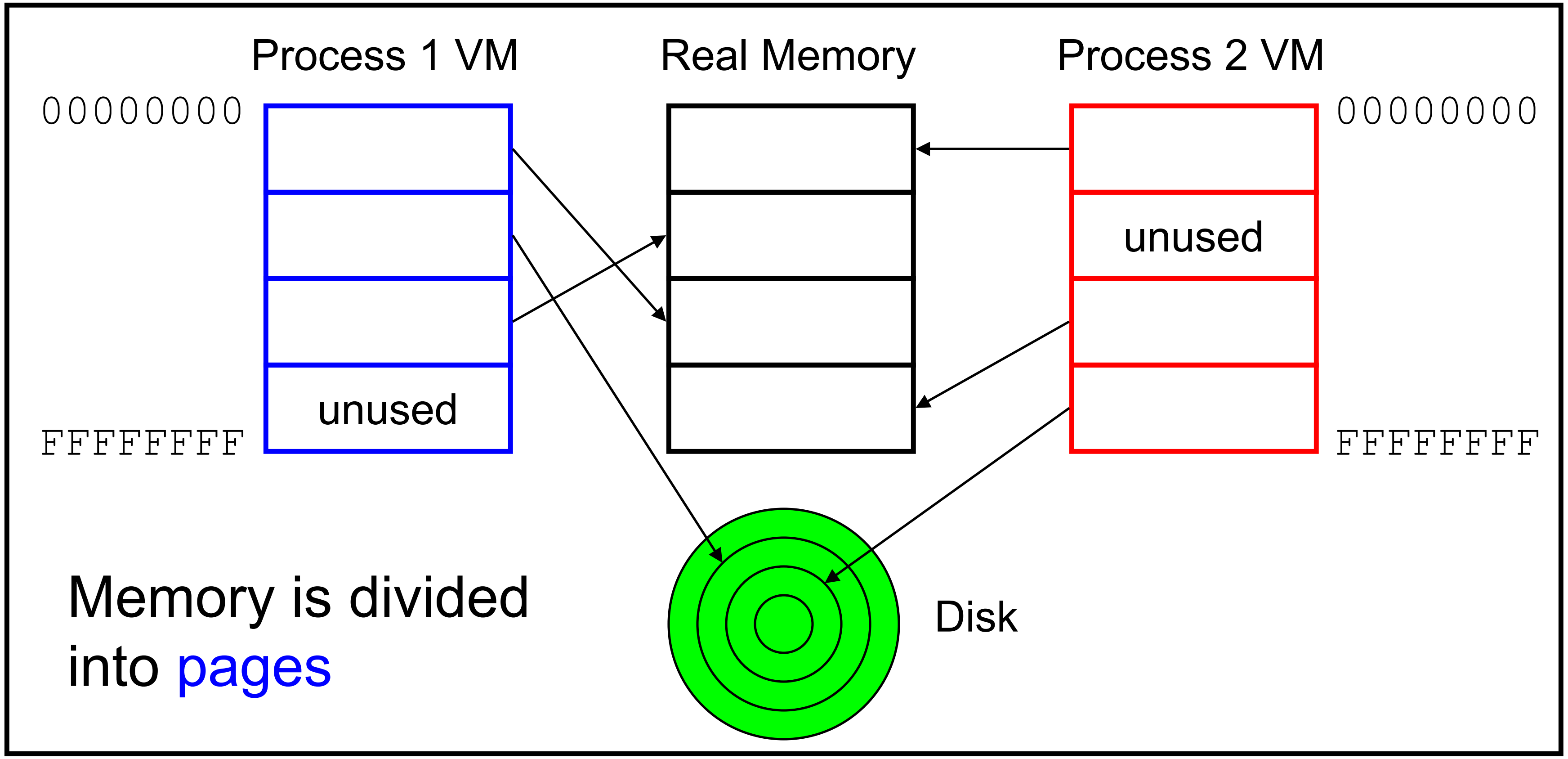
One of the best ideas in computer systems!

Private Address Space: Illusion



Hardware and OS give each application process
the illusion that it is the only process using memory

Private Address Space: Reality



All processes use the same real memory
Hardware and OS provide application programs with a virtual view of memory, i.e. virtual memory (VM)

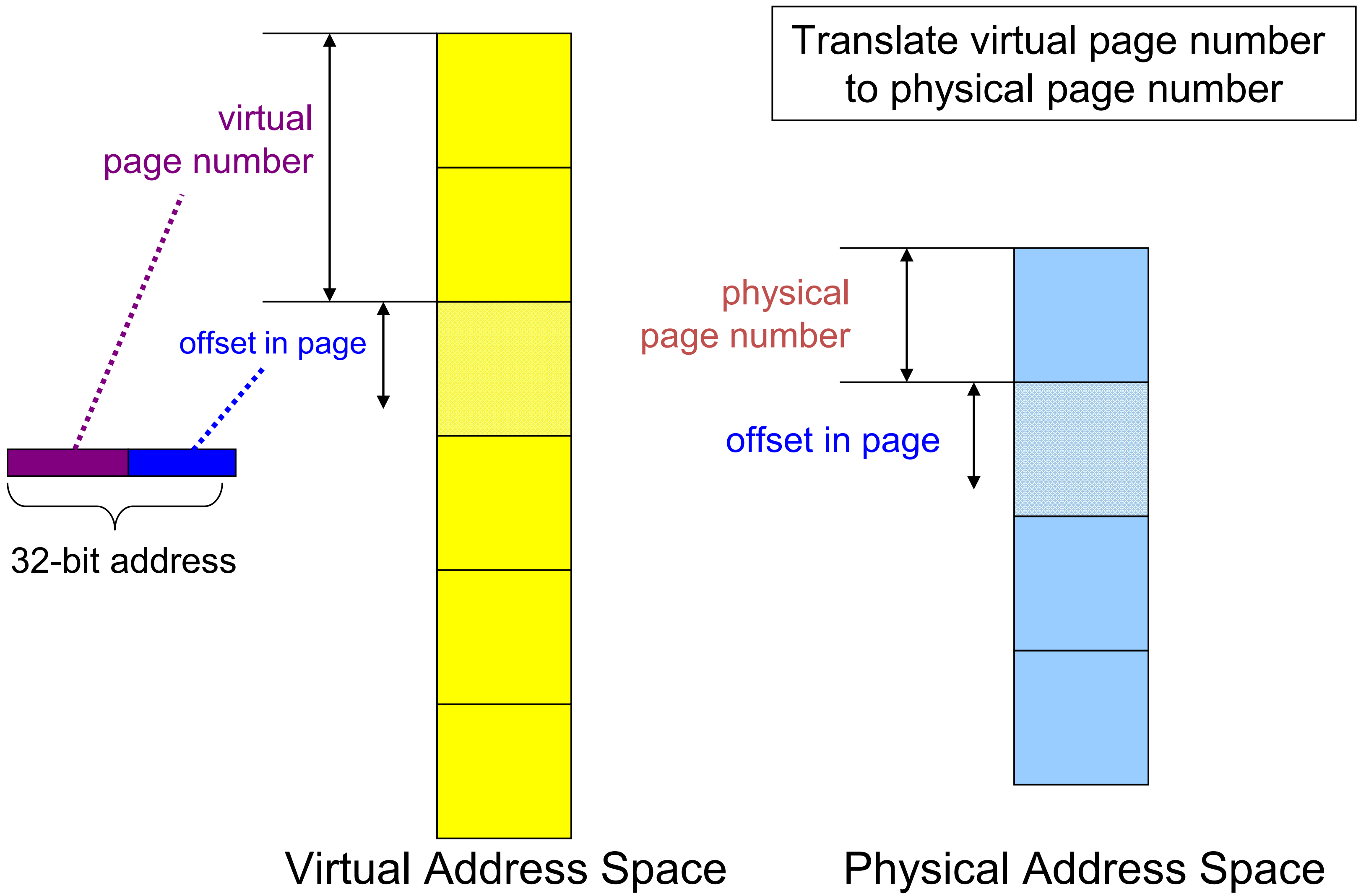
Making Good Use of Memory and Disk

- Good use of the disk
  - Read and write data in large “pages”
  - … to amortize the cost of “seeking” on the disk
  - e.g., page size of 4 KB
- Good use of main memory
  - Although the address space is large
  - … programs usually access only small portions at a time
  - Keep the “working set” in main memory
    - Demand paging: only bring in a page when needed
    - Page replacement: selecting good page to swap out
- Goal: avoid thrashing
  - Continually swapping between memory and disk

Virtual Address for a Process

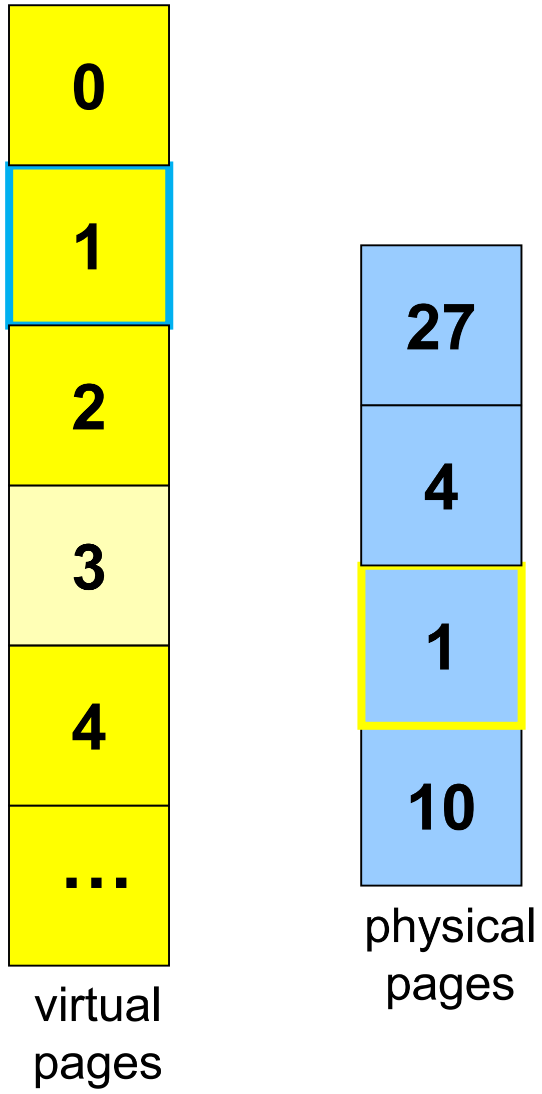
- Virtual page number
  - Number of the page in the virtual address space
  - Extracted from the upper bits of the (virtual) address
  - … and then mapped to a physical page number
- Offset in a page
  - Number of the byte within the page
  - Extracted from the lower bits of the (virtual) address
  - … and then used as offset from start of physical page
- Example: 4 KB pages
  - 20-bit page number: 2^{20} virtual pages
  - 12-bit offset: bytes 0 to 2^{12}-1

Virtual Address for a Process



Page Table to Manage the Cache

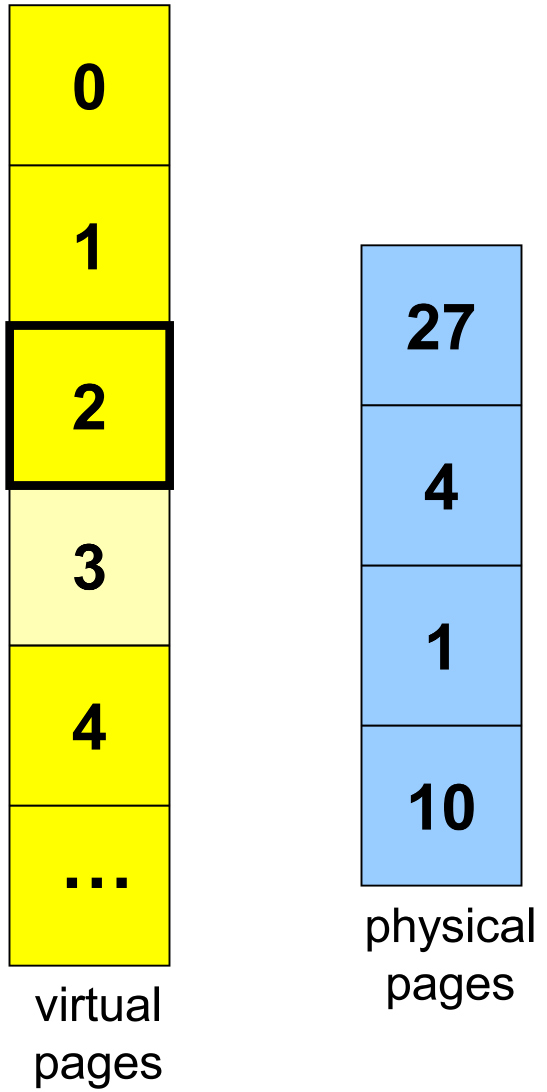
- Current location of each virtual page
  - Physical page number, or
  - Disk address (or null if unallocated)
- Example
  - Page 0: at location xx on disk
  - Page 1: at physical page 2
  - Page 3: not yet allocated
- Page “hit” handled by hardware
  - Compute the physical address
    - Map virtual page # to physical page #
    - Concatenate with offset in page
  - Read or write from main memory
    - Using the physical address
- Page “miss” triggers an exception…



"Miss" Triggers Page Fault

- Accepting the page not in main memory

	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…



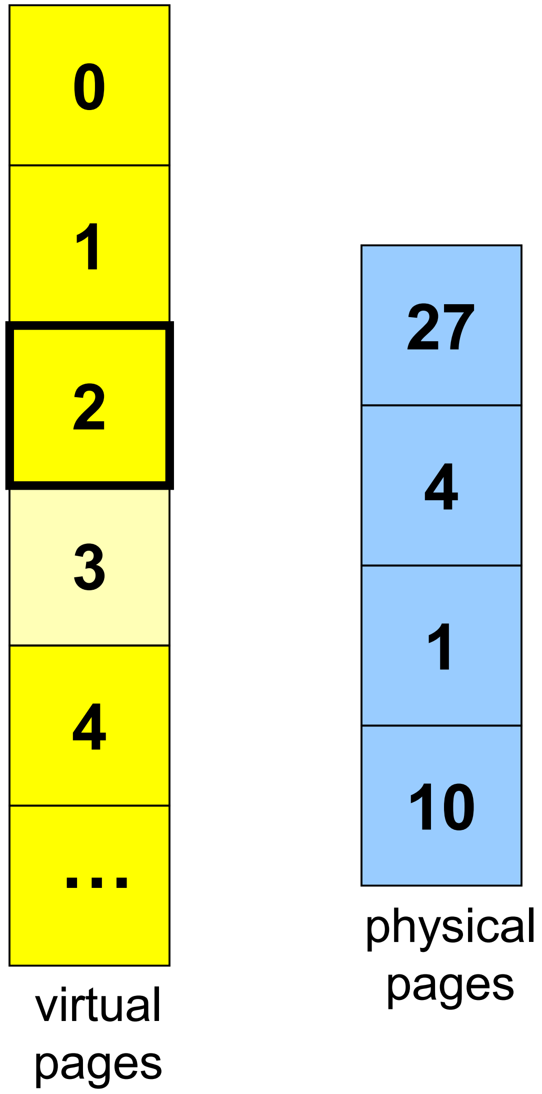
"Miss" Triggers Page Fault

- Accessing the page not in main memory

	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…

```
movl 00002104, %eax
```

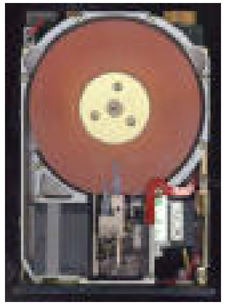
Virtual page #2 at location yy on disk!

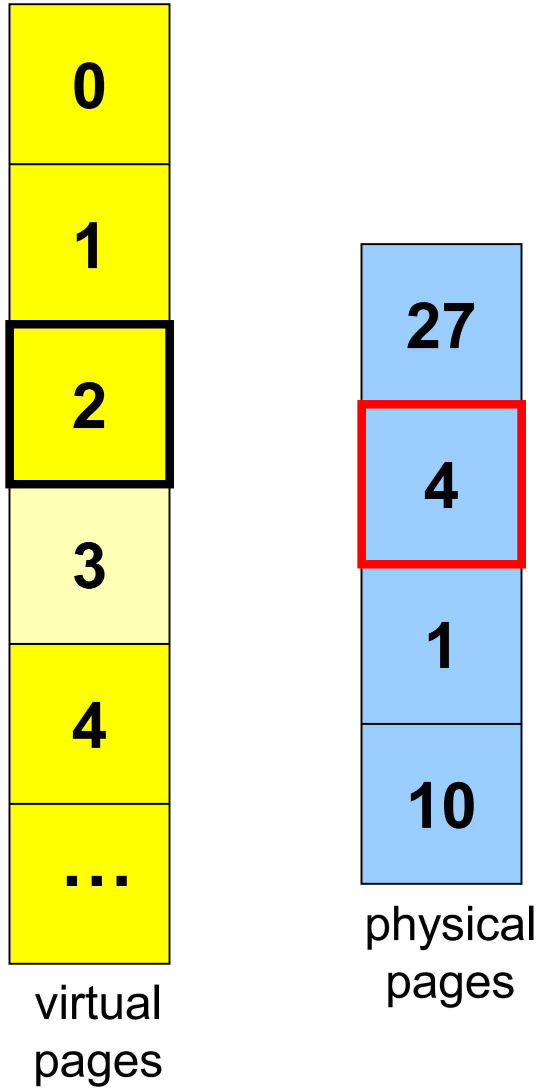


OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…

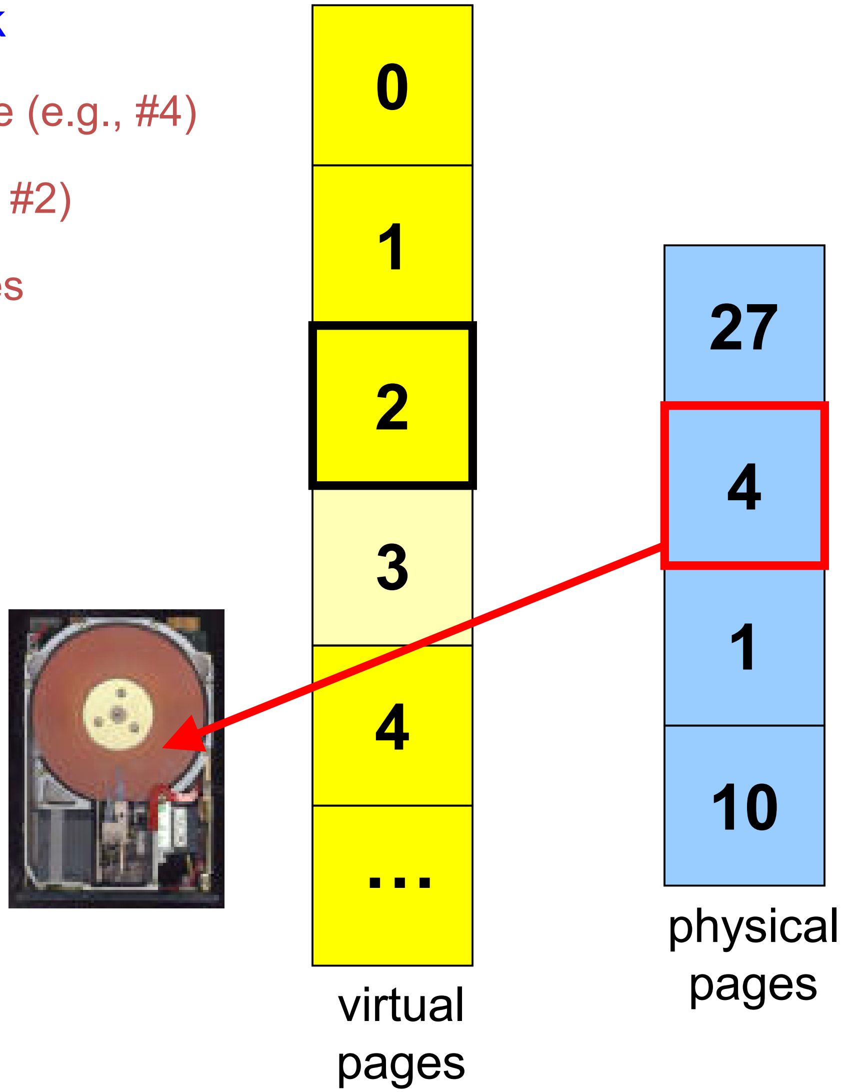




OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

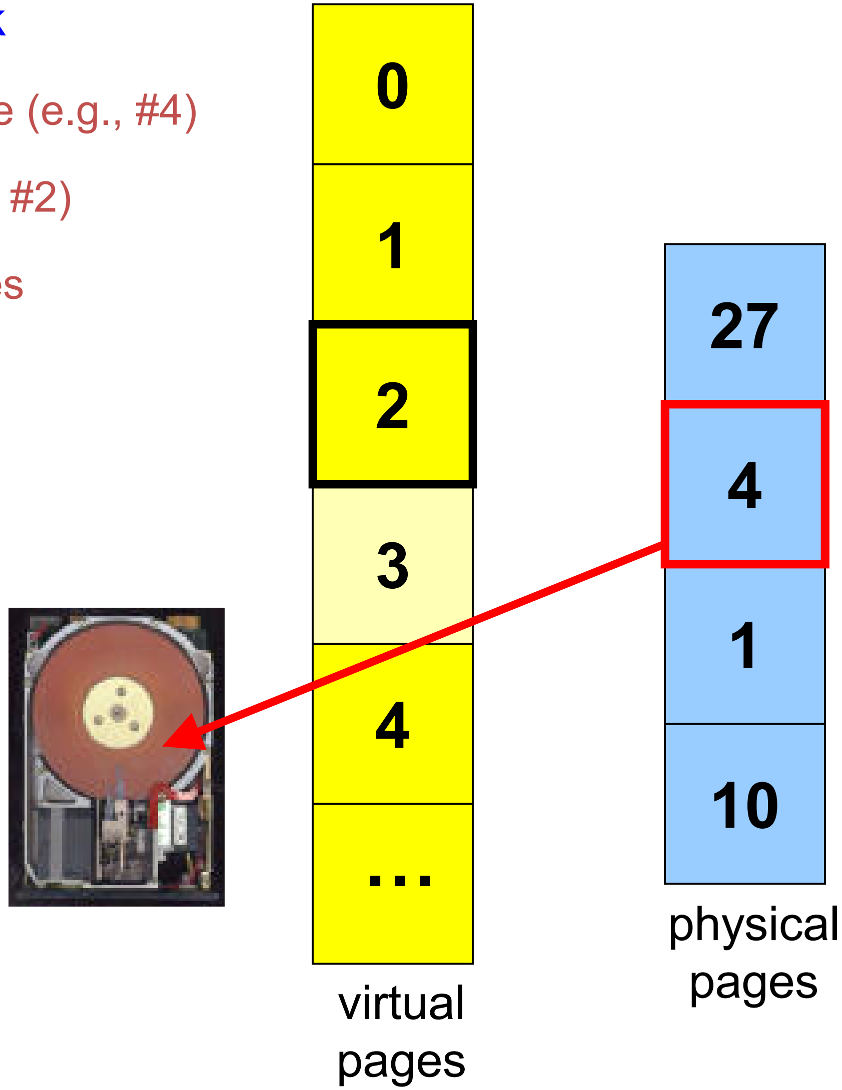
	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	1	1
		…



OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

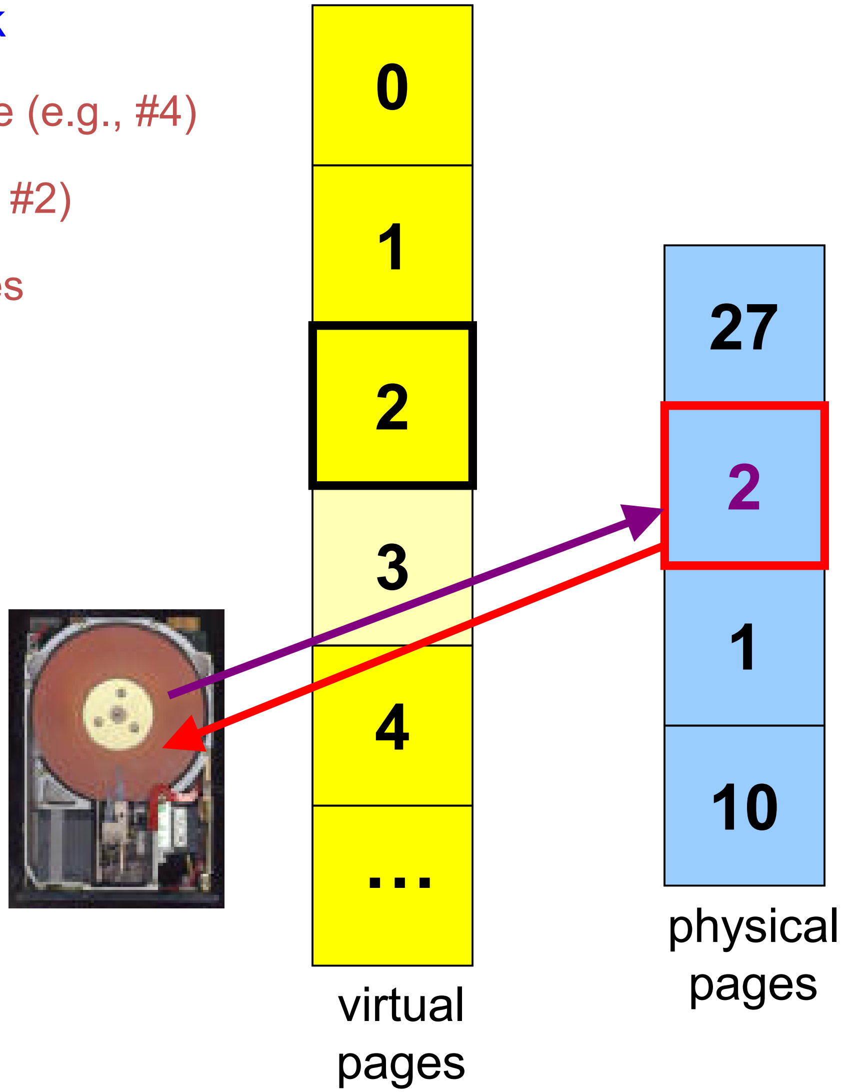
	V	Physical or disk address
0	0	xx
1	1	2
2	**0**	**yy**
3	0	null
4	~~1~~ 0	~~1~~ zz
		…



OS Handles the Page Fault

- Bringing page in from the disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries

	V	Physical or disk address
0	0	xx
1	1	2
2	~~0~~ 1	~~yy~~ 1
3	0	null
4	~~1~~ 0	~~1~~ zz
		…



VM as a Tool for Memory Protection

- Memory protection
  - Prevent processes from unauthorized reading or writing of memory

- User process should not be able to
  - Modify the read-only text section in its own address space
  - Read or write operating-system code and data structures
  - Read or write the private memory of other processes

- Hardware support
  - Permission bits in page-table entries (e.g., read-only)
  - Separate identifier for each process (i.e., process-ID)
  - Switching between *unprivileged* mode (for user processes) and *privileged* mode (for the operating system)

EE209 #19:
I/O Management

KAIST EE

Example: Opening a File

- `FILE *fopen("myfile.txt", "r")`
  - Opens the named file and return a stream
  - Includes a mode, such as “r” for read or “w” for write
- Creates a FILE data structure for the file
  - Mode, status, buffer, …
  - Assigns fields and returns a pointer
- Opens or creates the file, based on the mode
  - Write (‘w’): create the file with default permissions
  - Read (‘r’): open the file as read-only
  - Append (‘a’): open or create file, and seek to the end

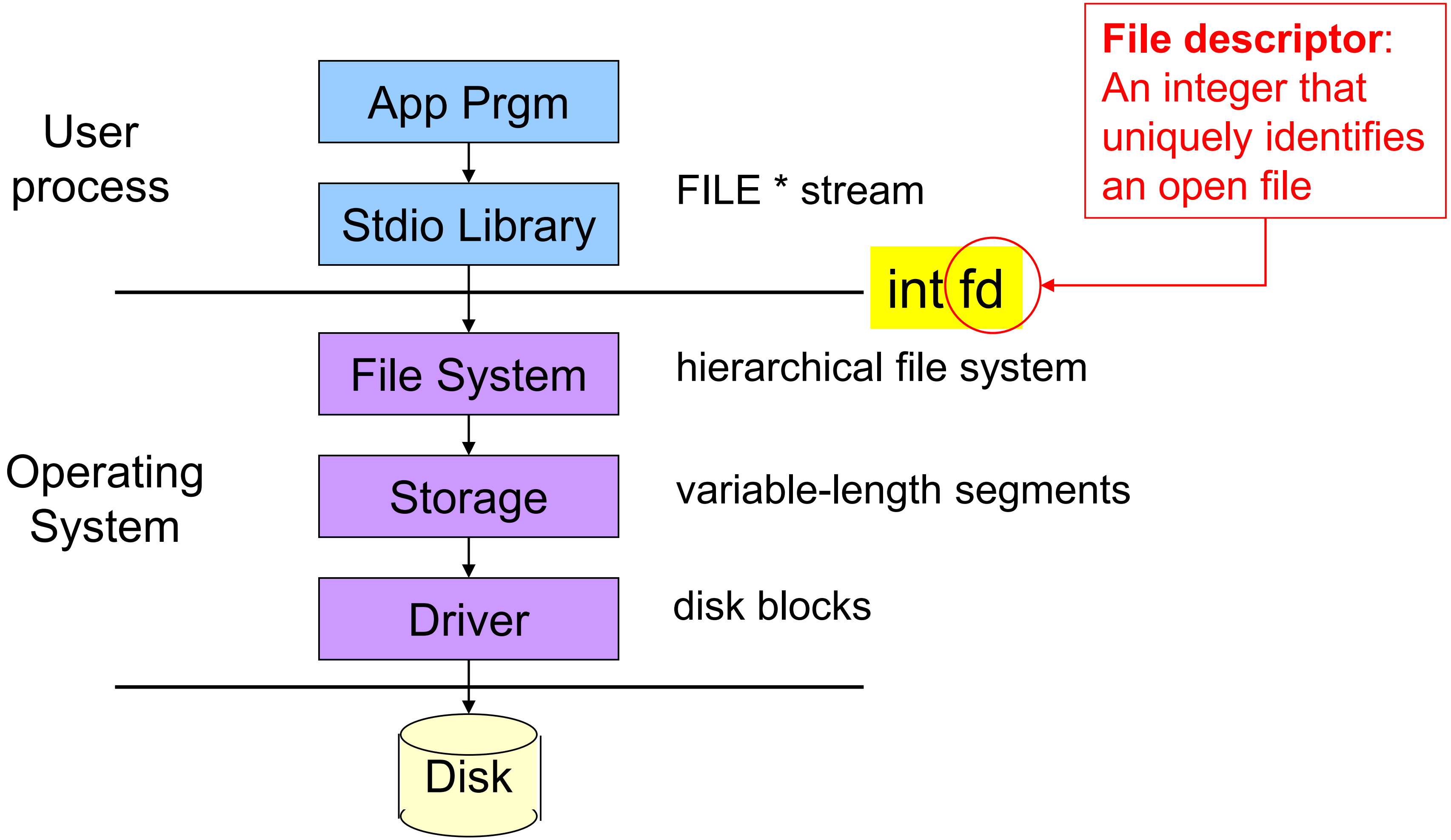
Example: Formatted I/O

- `int fprintf(fp1, "Number: %d\n", i)`
  - Convert and write output to stream in specified format
- `int fscanf(fp1, "FooBar: %d", &i)`
  - Read from stream in format and assign converted values

- Specialized versions
  - `printf(…)` is just `fprintf(stdout, …)`
  - `scanf(…)` is just `fscanf(stdin, …)`

  - `<stdio.h> has a variable FILE* stdin;`

Layers of Abstraction



System-Level Functions for I/O

```
int creat(char *pathname, mode_t mode);
```

- Creates a new file named `pathname`, and returns a file descriptor

```
int open(char *pathname, int flags, mode_t mode);
```

- Opens the file `pathname` and returns a file descriptor

```
int close(int fd);
```

- Closes `fd`

```
int read(int fd, void *buf, int count);
```

- Reads up to `count` bytes from `fd` into the buffer at `buf`

```
int write(int fd, void *buf, int count);
```

- Writes up to `count` bytes into `fd` from the buffer at `buf`

```
int lseek(int fd, int offset, int whence);
```

- Assigns the file pointer of `fd` to a new value by applying an `offset`

Example: `open()`

- Converts a path name into a file descriptor
  - `int open(const char *pathname, int flags, mode_t mode);`
- Arguments
  - `pathname`: name of the file
  - `flags`: bit flags for `O_RDONLY, O_WRONLY, O_RDWR`
  - `mode`: permissions to set if file must be created
- Returns
  - File descriptor (or -1 if error)
- Performs a variety of checks
  - e.g., whether the process is entitled to access the file
- Underlies `fopen()`

Example: `read()`

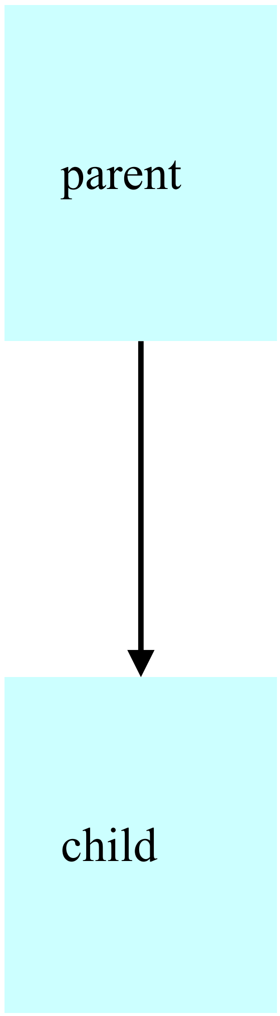
- Reads bytes from a file descriptor
  - `int read(int fd, void *buf, int count);`
- Arguments
  - File descriptor: integer descriptor returned by `open()`
  - Buffer: pointer to memory to store the bytes it reads
  - Count: maximum number of bytes to read
- Returns
  - Number of bytes read
    - Value of 0 if nothing more to read
    - Value of -1 if an error
- Performs a variety of checks
  - Whether file has been opened, whether reading is okay
- Underlies `getchar()`, `fgets()`, `scanf()`, etc.

EE209 #20:
Process Management

KAIST EE

Creating a New Process

- Cloning an existing process
  - Parent process creates a new child process
  - The two processes then run concurrently
- Child process inherits state from parent
  - Identical (but separate) copy of virtual address space
  - Copy of the parent's open file descriptors
  - Parent and child share access to open files
- Child then runs independently
  - Executing independently, including invoking a new program
  - Reading and writing its own address space



Fork System-Level Function

- `fork()` is called once
  - but returns twice, once in each process
  - because a new process is created, as a result of fork()
  - 1+1 = 2

- Telling which process is which
  - Parent: `fork()` returns the child's process ID
  - Child: `fork()` returns 0

```
pid = fork();
if (pid != 0) {
    /* in parent */
   …
} else {
    /* in child */
    …
}
```

Executing a New Program

- `fork()` copies the state of the parent process
  - Child continues running the parent program
  - … with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example

program

NULL-terminated array
Contains command-line arguments
(to become “argv[]” of ls)

```
execvp("ls", argv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

Waiting for the Child to Finish

- Parent should wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for a child to terminate: `wait()`
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for specific child to terminate: `waitpid()`
  - Blocks till a child with particular process ID terminates

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```