EE309
Lecture 2: EE209/EE485 Review

INSU YUN (윤인수)

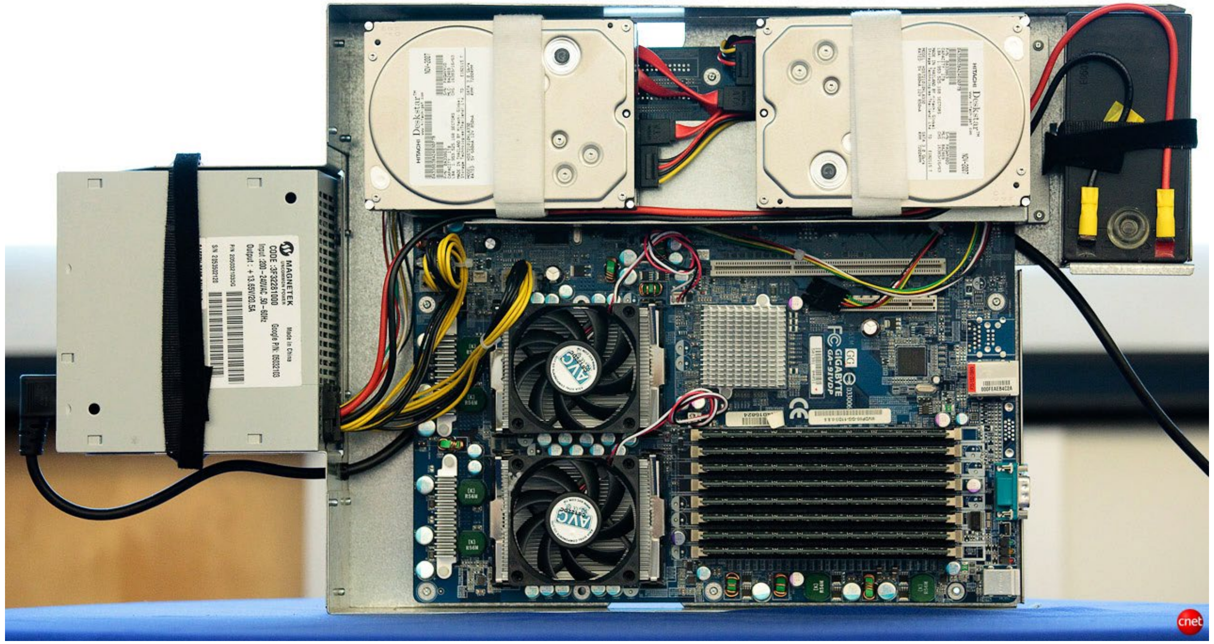
School of Electrical Engineering, KAIST

[Lecture slides based on EE209]

Lecture 1. Introduction

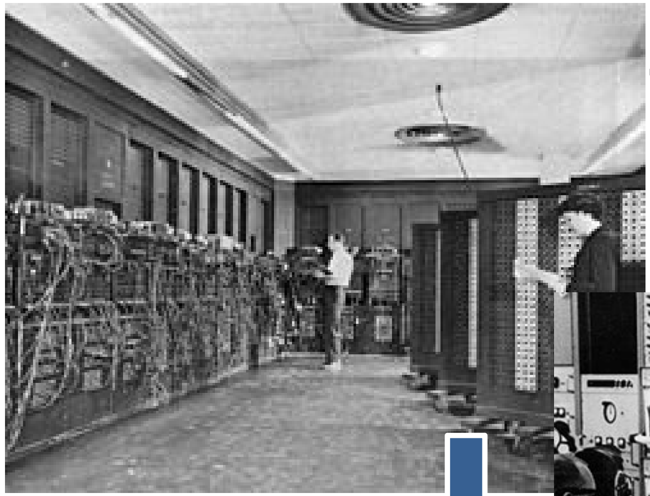
KAIST EE

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

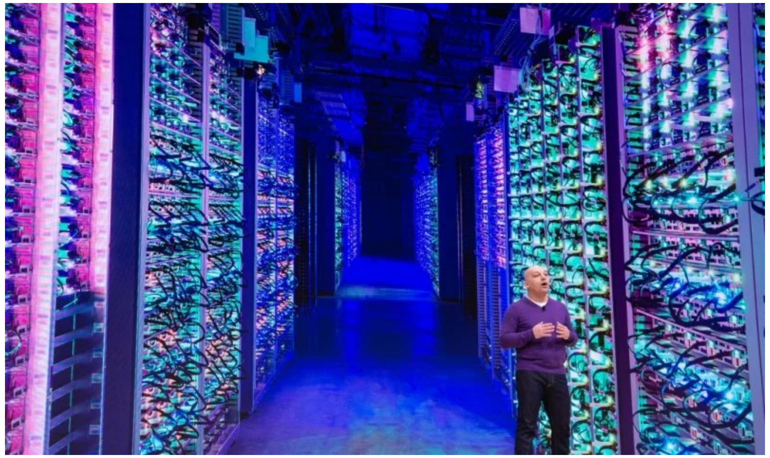
Smartphones today

Trend#2: Ubiquitous, everywhere

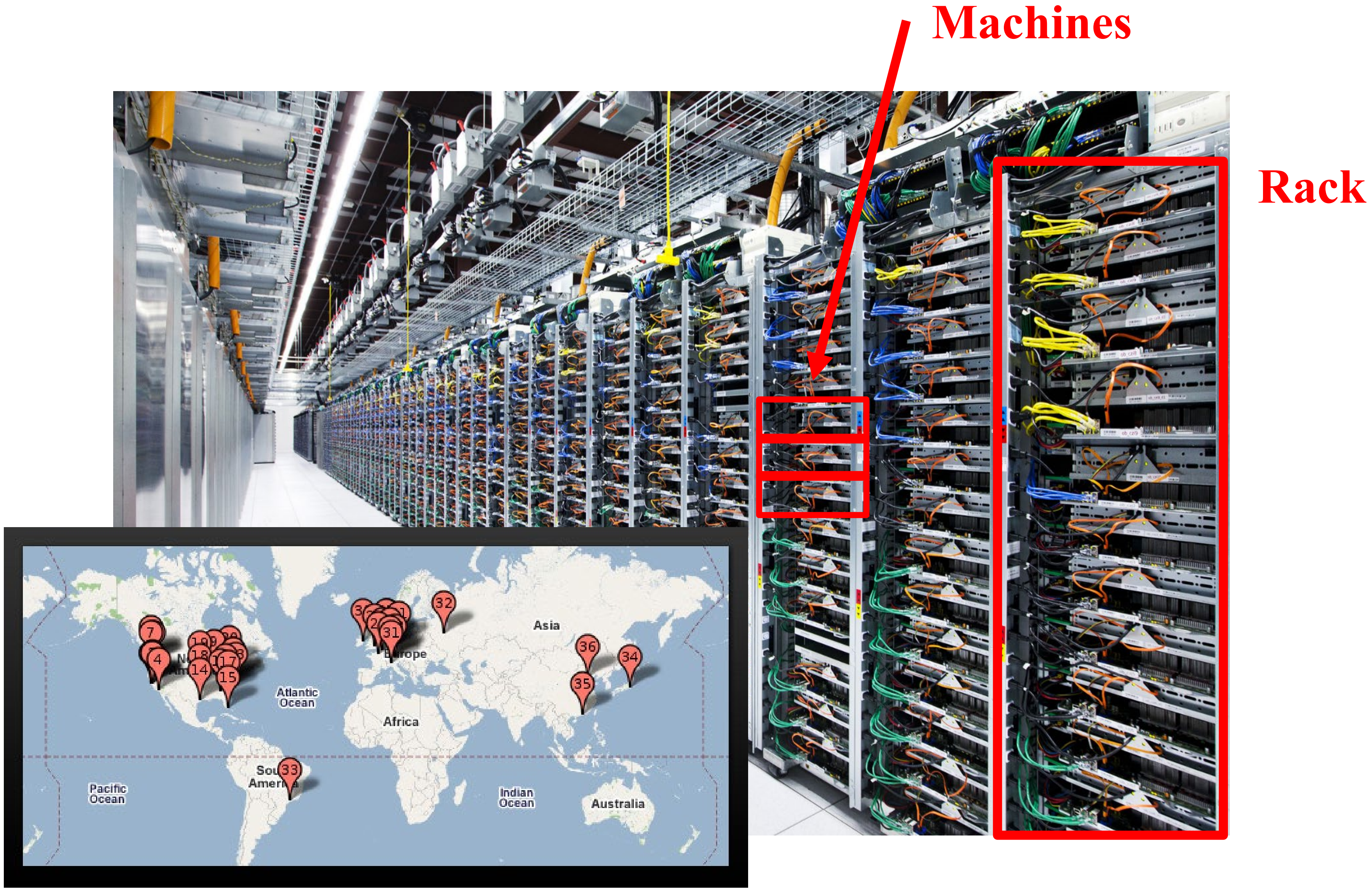
- Computers are dominating our lives!
- More things are becoming computers
  - Cars, watches, speakers, pets, … what's next?

Trend#3: Growing to a larger scale

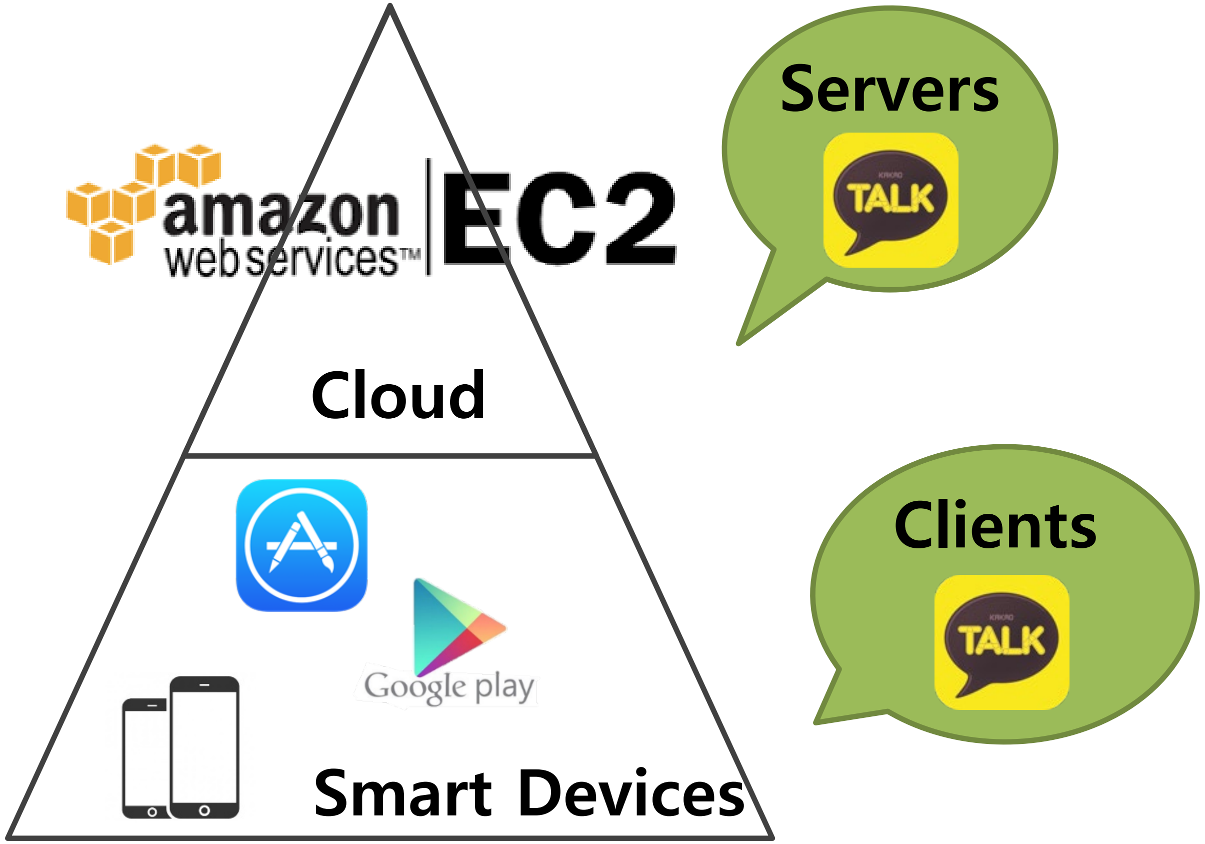
- Scale of the "Cloud"
  - Many machines spread out around the globe
  - Facebook: hundreds of thousands of machines
  - Microsoft: 4 million servers (~2021)
  - Google: 2.5 million servers in 2016
  - Amazon, Google, Facebook, and Microsoft spent $37B in 2020Q3



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};
```

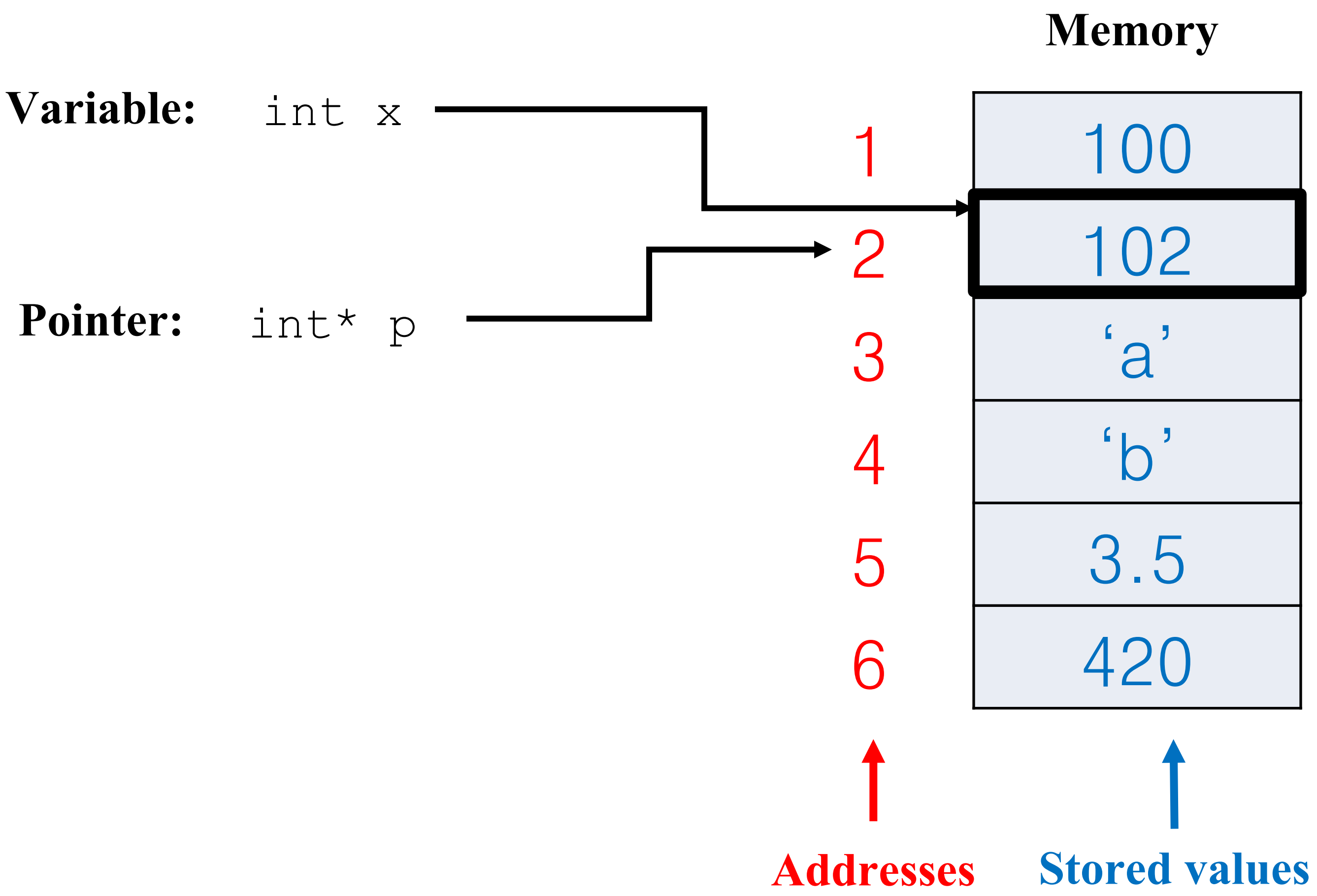
- Array: a collection of elements of the **same** type

```
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;
```

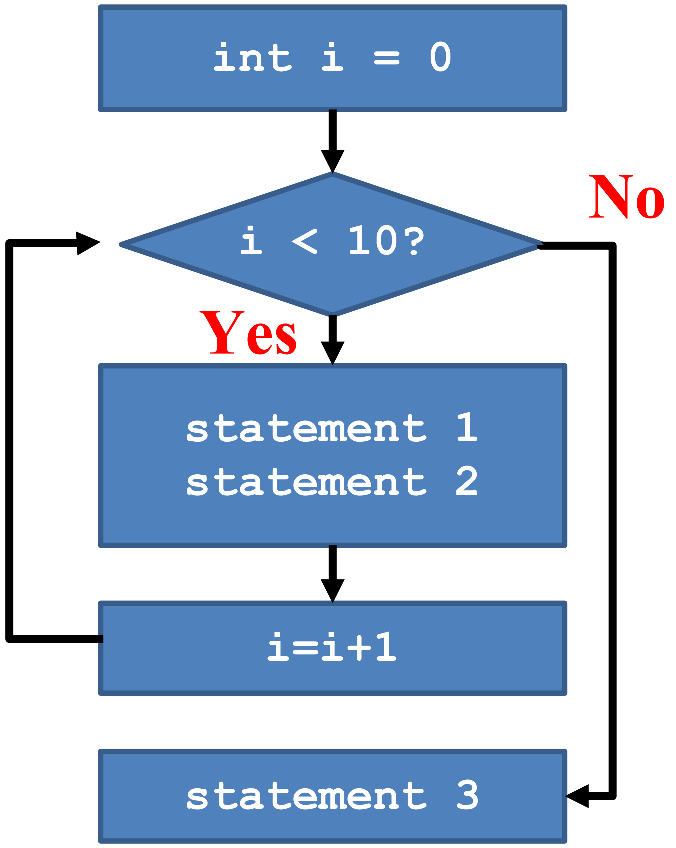
- switch/case statement

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

Loop Statements (1)

- **for** statement

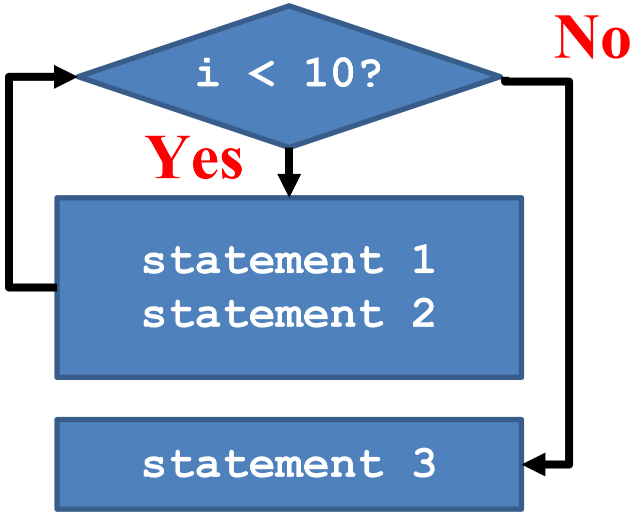
```
for (int i = 0; i < 10; i++){
    statement 1;
    statement 2;
}
statement 3;
```



Loop Statements (2)

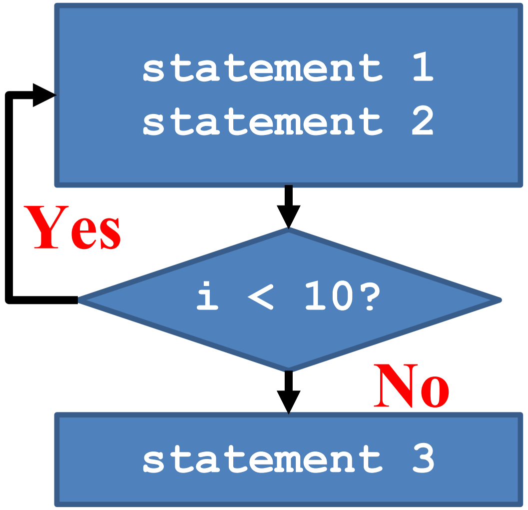
- **while** statement

```
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

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

- Compile and execute hello.c

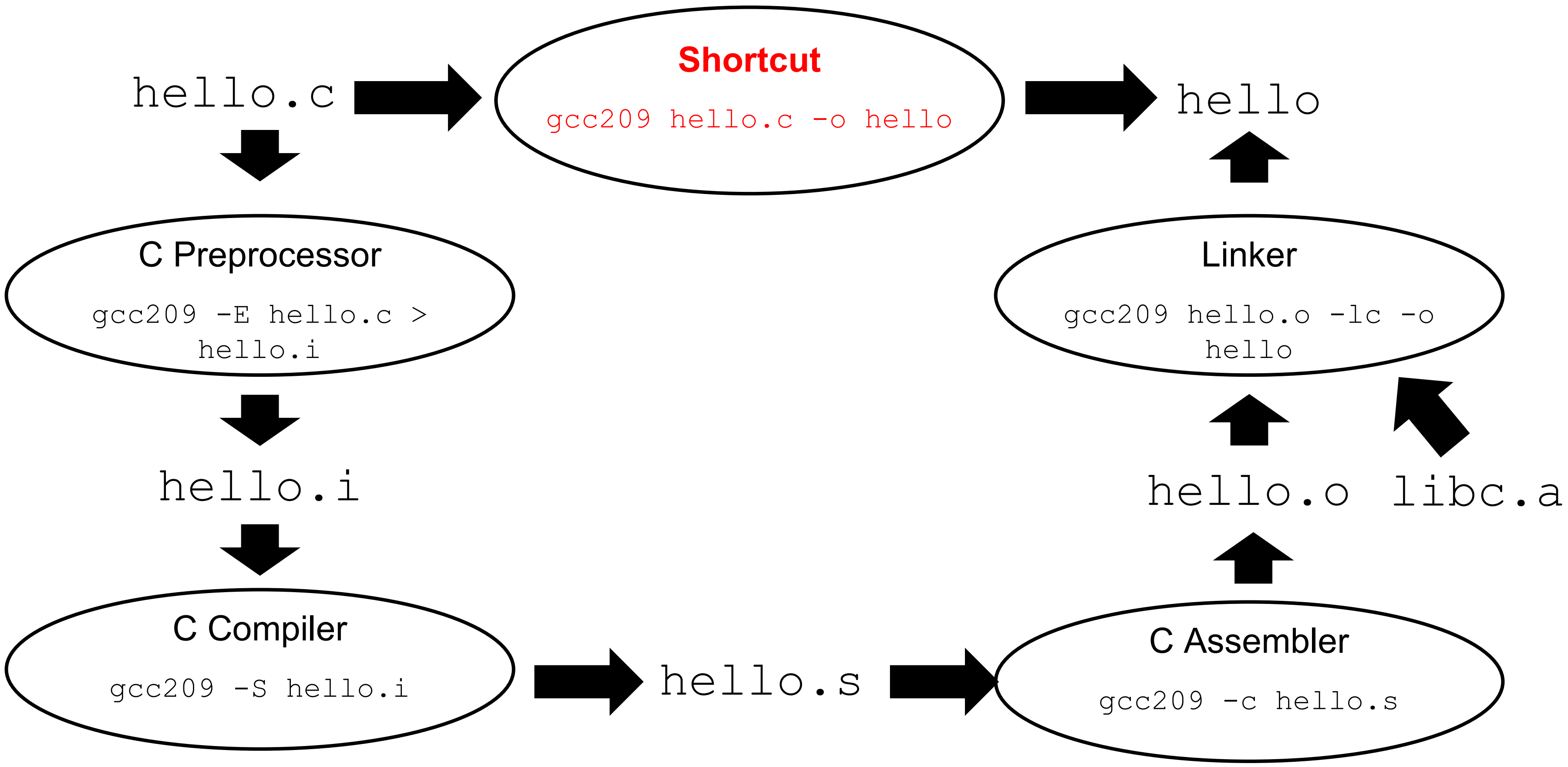
```
ee209@ubuntu:~$ gcc209 hello.c -o hello
ee209@ubuntu:~$ ./hello
hello, world
```

`gcc209` is a script that executes

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

all warnings | **Make all warnings errors** | **Same as -std=c89** | **Follow the ISO C standard specified by -std** | **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

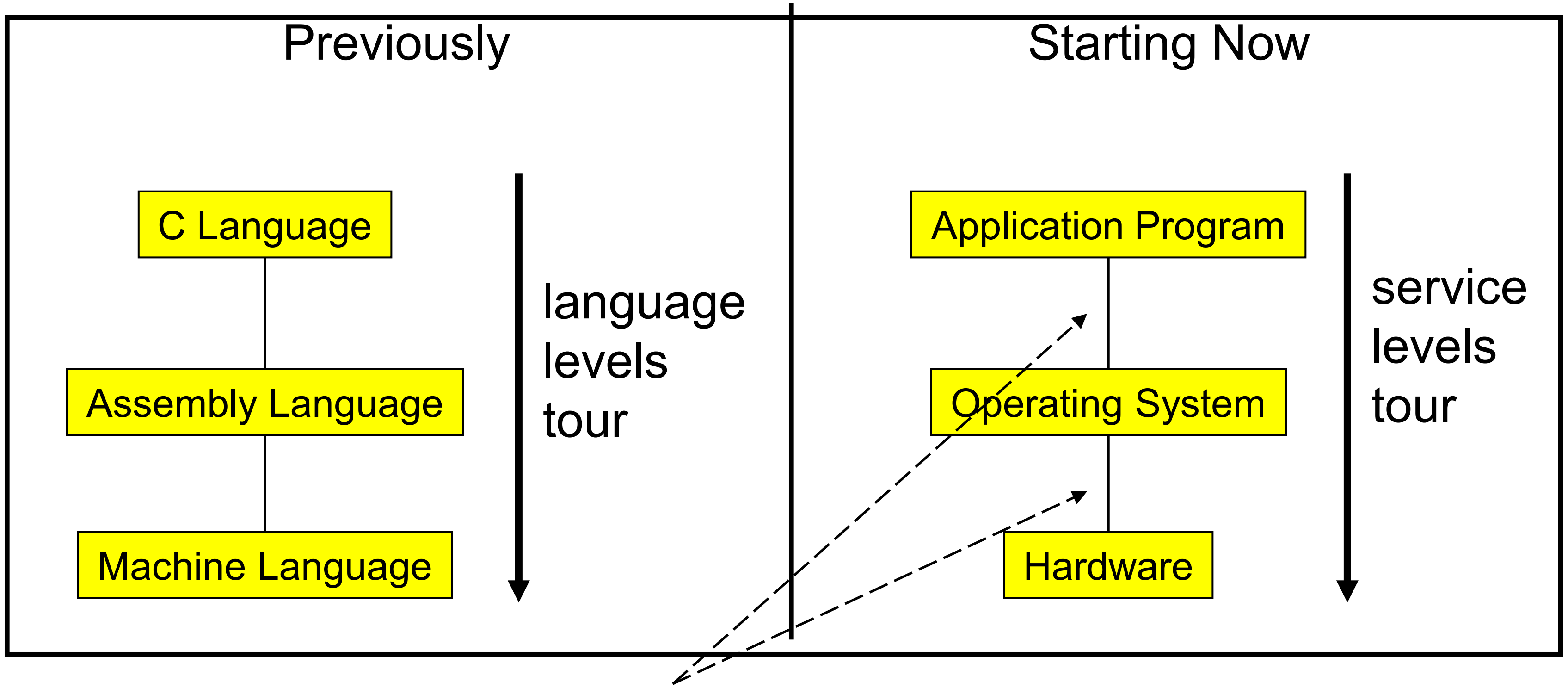
Lecture 16: Exceptions and Processes

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

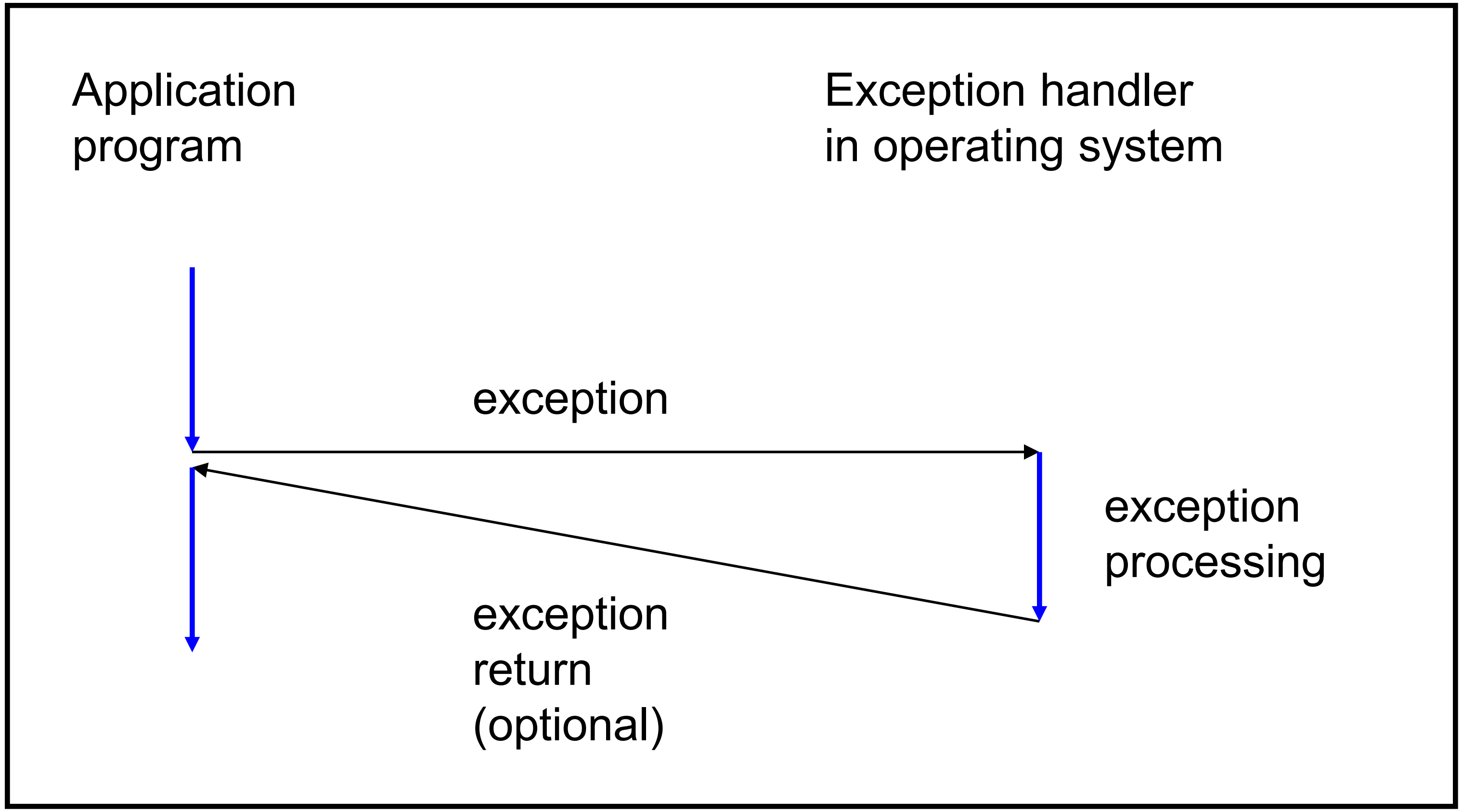
Exceptions

- **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

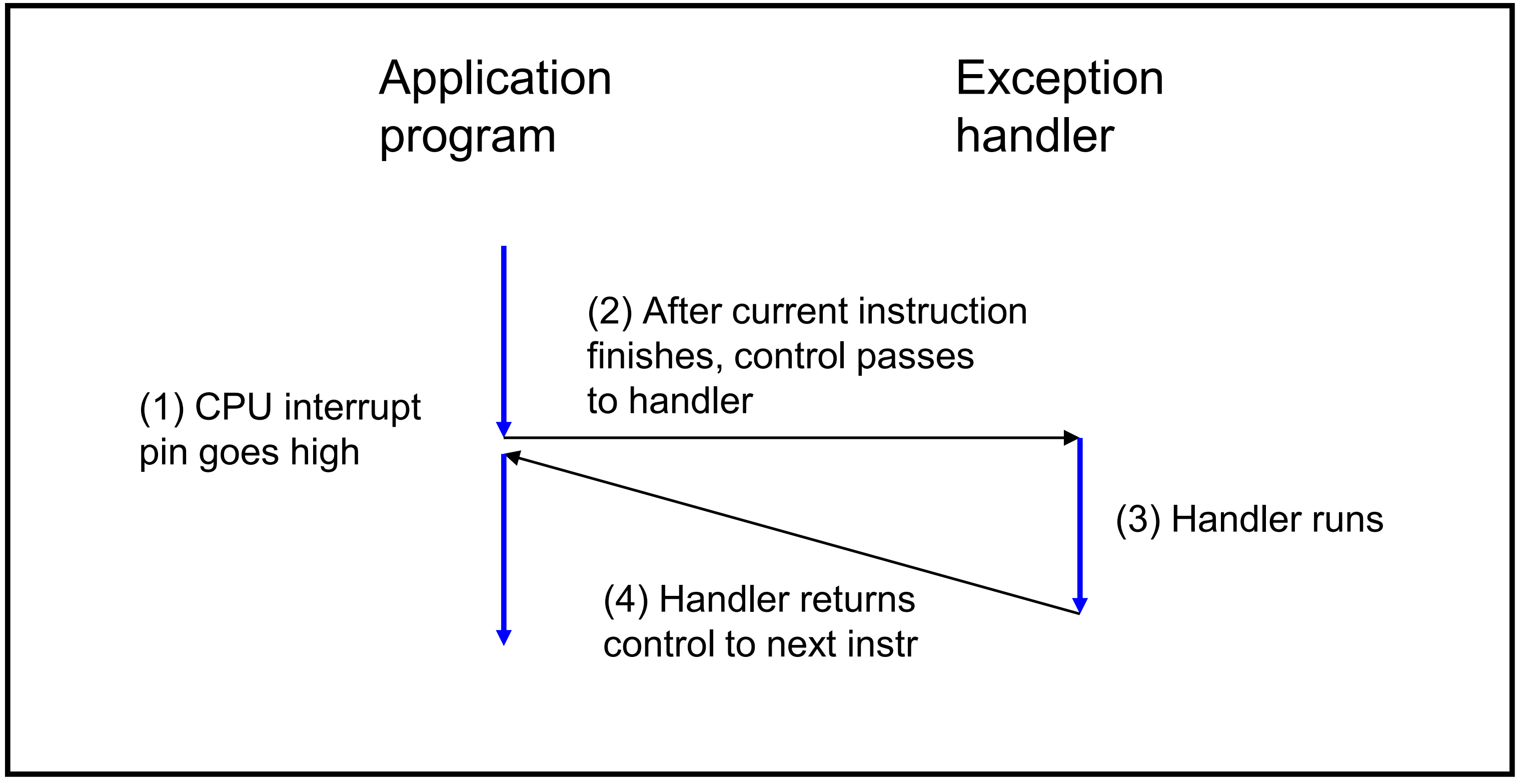
Exceptional Control Flow



Classes of Exceptions

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

(1) Interrupts

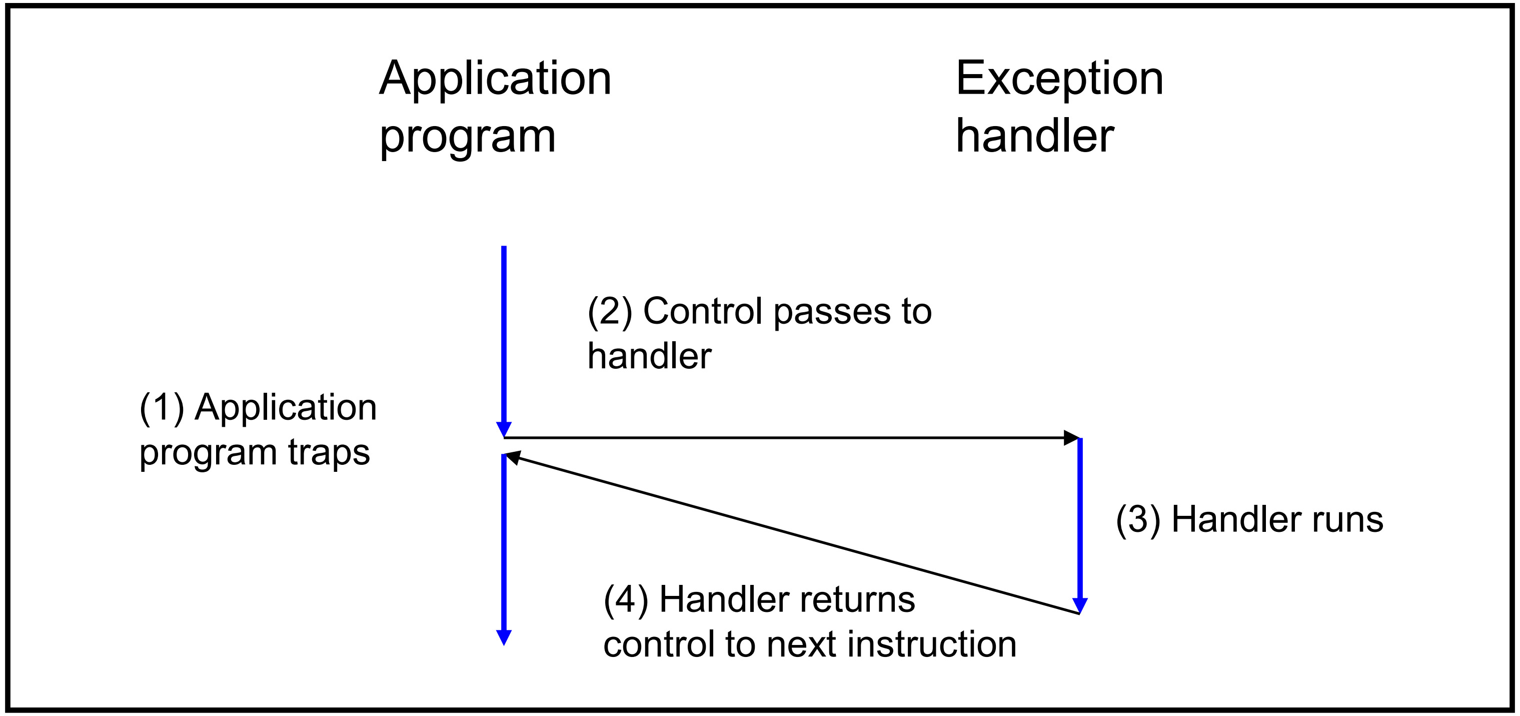


Cause: Signal from I/O device (asynchronously)

Examples:

- User presses key
- Disk controller finishes reading/writing data

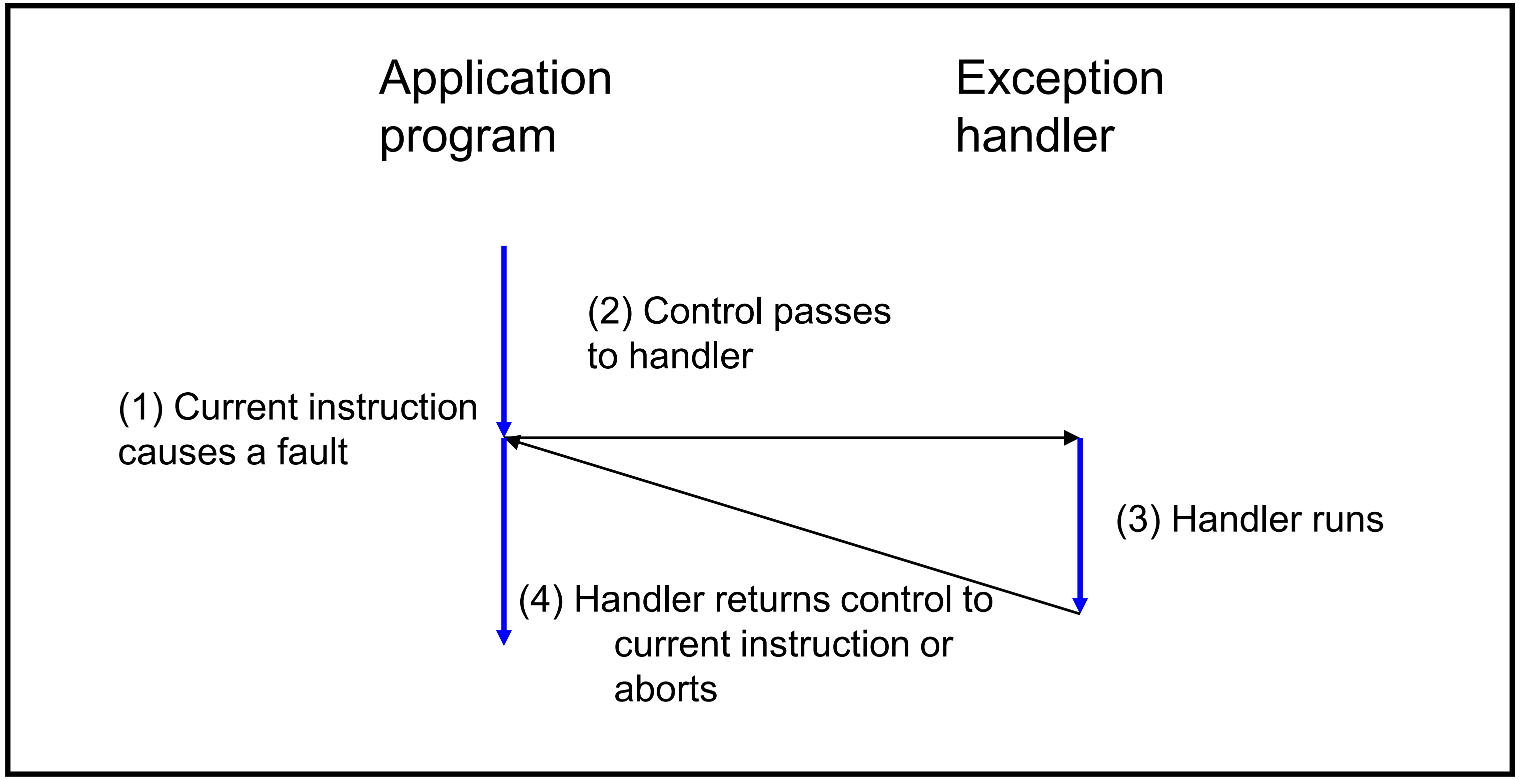
(2) Traps



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

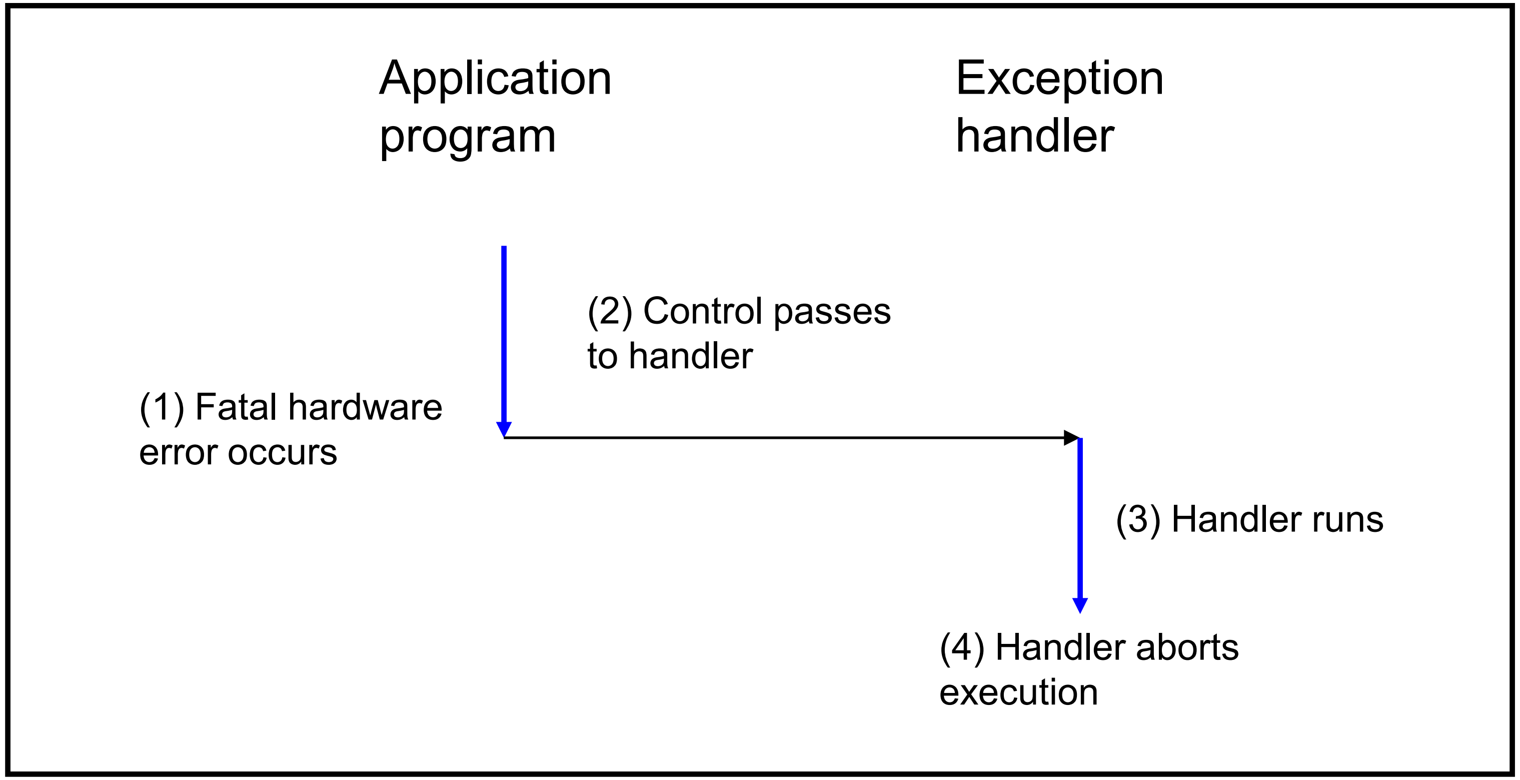


Cause: Application program causes (possibly) recoverable error

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

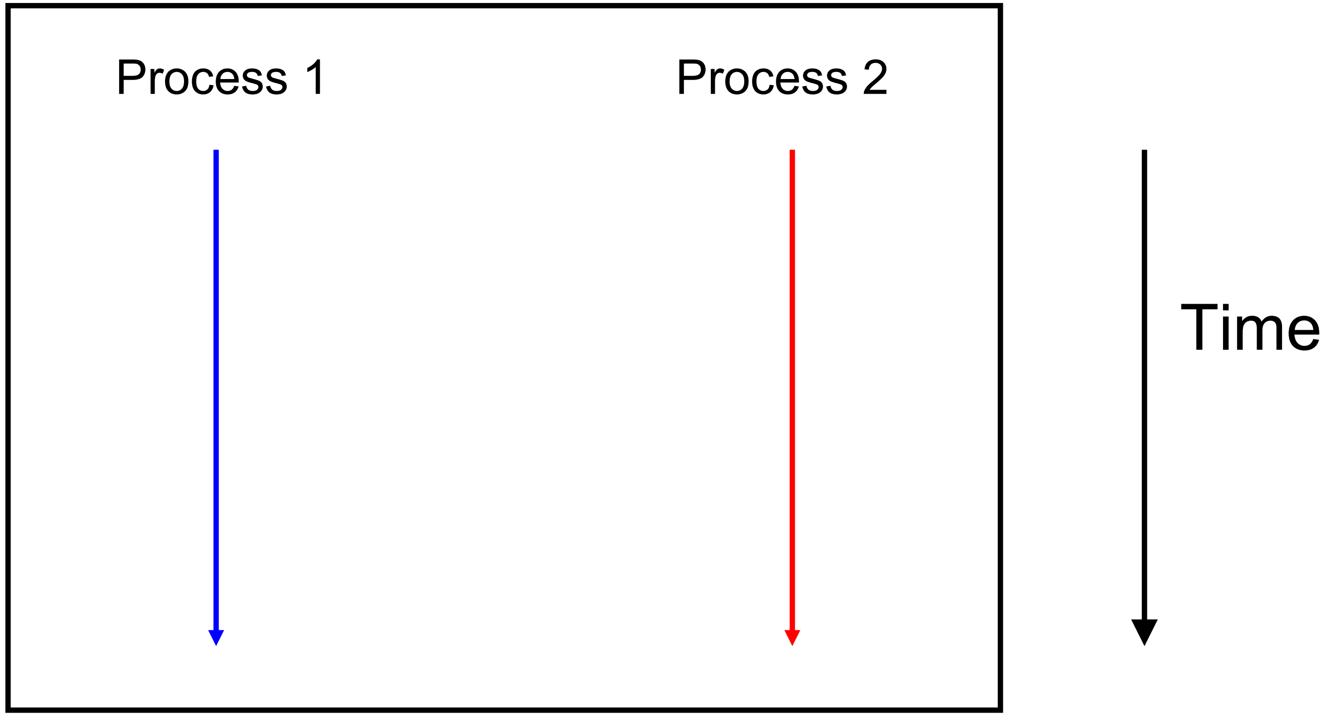
- **Program**
  - Executable code
- **Process**
  - An instance of a program in execution
- Each program runs in the **context** of some process
- **Context** consists of:
  - Process ID
  - Address space
    - TEXT, RODATA, DATA, BSS, HEAP, and STACK
  - Processor state
    - EIP, EFLAGS, EAX, EBX,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

Significance of Processes

- **Process** is a profound abstraction

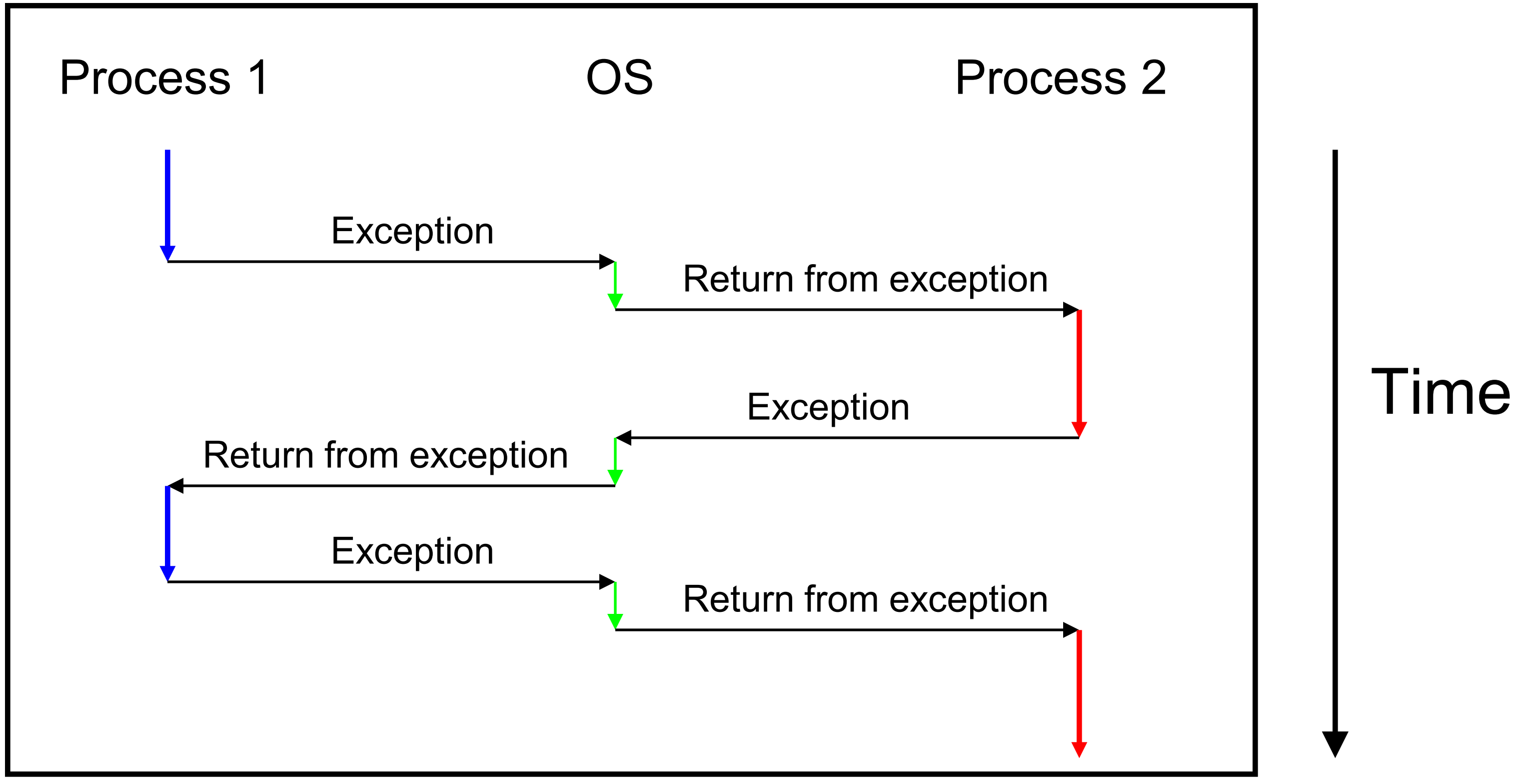
- The process abstraction provides application programs with two key illusions:
  - Private control flow
  - Private address space

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

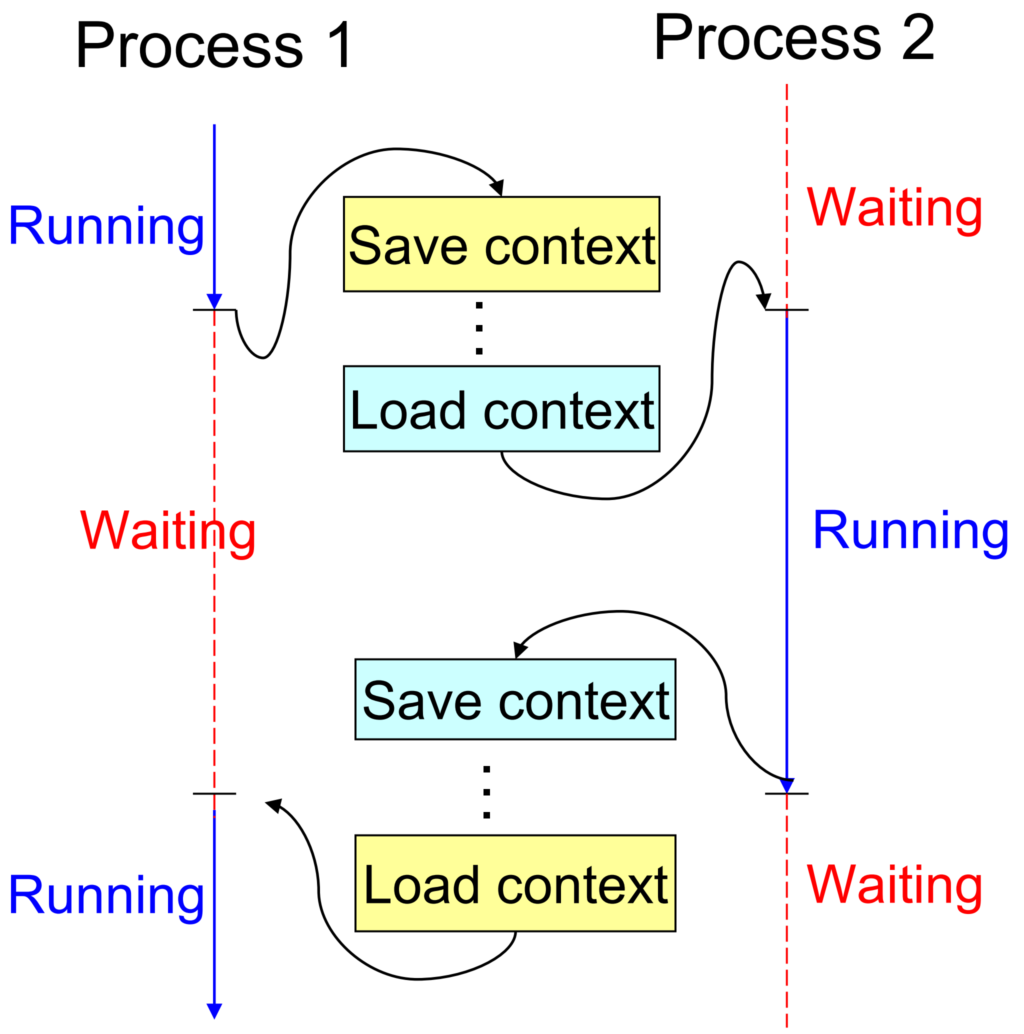
- **Context switch**
  - The activity whereby the OS assigns the CPU to a different process
  - Occurs during exception handling, at the discretion of OS
- Exceptions can be caused:
  - Synchronously, by application program (trap, fault, abort)
  - Asynchronously, by external event (interrupt)
  - **Asynchronously, 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

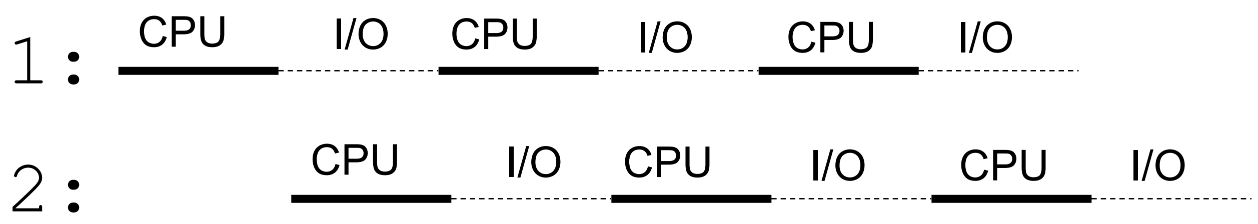
Context Switch Details

- 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



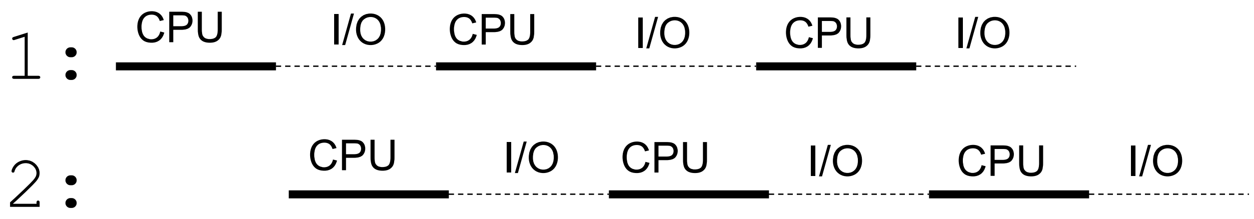
When Should OS Do Context Switch?

- 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