Introduction to System Programming

Introduction to System Programming

- **Why system programming?**
- Basic operation of a computer system
- Summary

Acknowledgement: slides based on the cs:app2e material

Why System Programming?

"I program in Java (JavaScript, Ruby, Perl, Python, PHP, Go, …). I don't need to know this stuff."

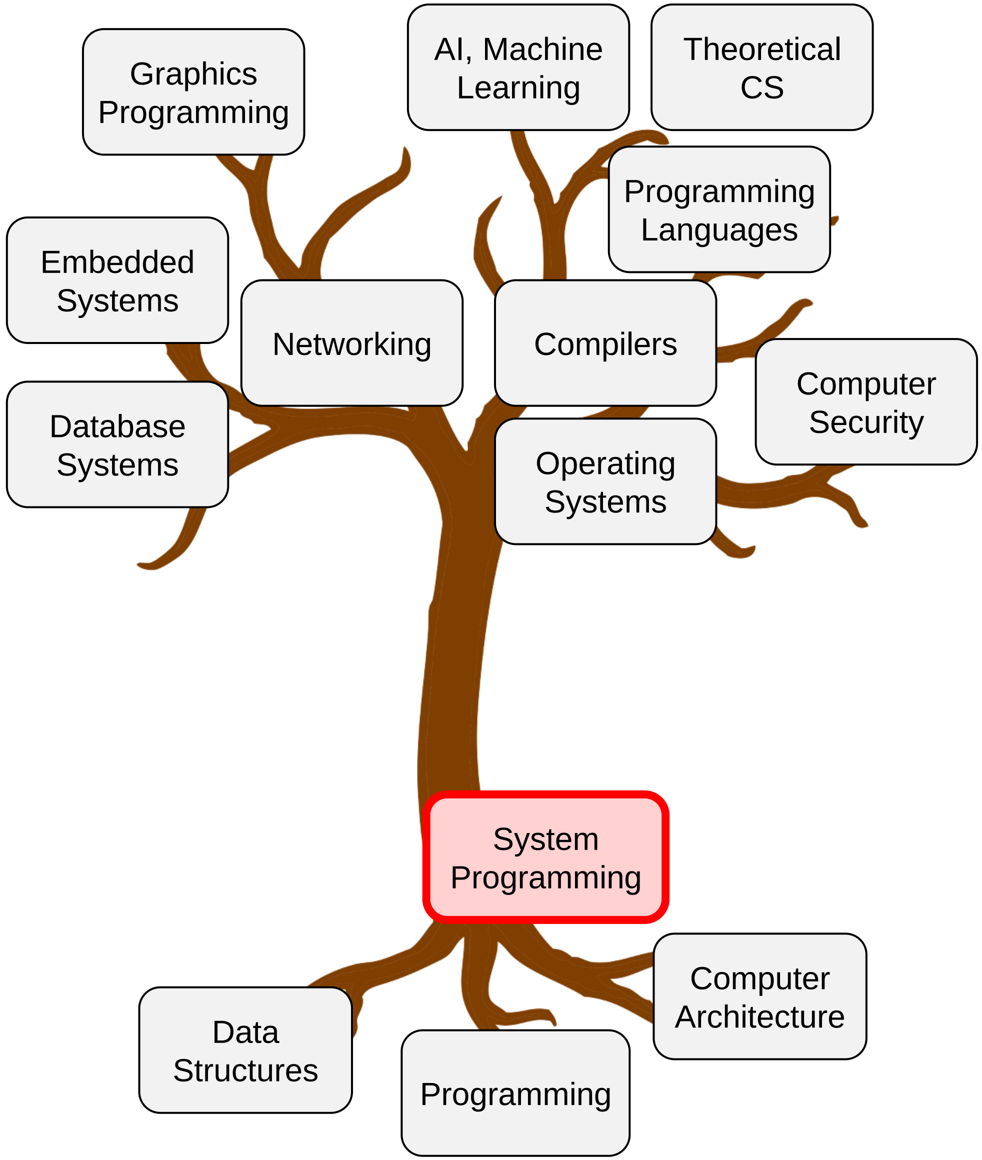
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Why System Programming?

- **Abstractions only partially reflect reality**
  - To be a good computer engineer you have to know that abstractions in computer science are only abstractions; the reality may look very different, especially in special cases
  - To understand the limitations of these abstractions and write safe and efficient code you need to know
    - computer architecture
    - assembly programming
    - system programming
- Programming, data structures, computer architecture, and system programming prepare you for more advanced classes in computer science & engineering
  - operating systems, compilers, embedded systems, networks, data base systems, …

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Why System Programming?



Reality Check #1: Integer and Real Numbers

- Integer: a number with no fractional part

…, -4, -3, -2, -1, 0, 1, 2, 3, 4, …

- Real: rational (integers, fractions) and irrational numbers (√2, π)

…, -4, -3/2, - √2, 0, 1, 1.1, 1.11, π, 100, …

- Abstraction: in a computer program
  - `integer = int` (or `long`)
  - `real = float` (or `double`)

Integer Numbers

- integer.c

```
#include <stdio.h>

void main(void)
{
  int i = 1000000;
  int k;

  for (k = 0; k<20; k++) {
    printf("%2d:  %d\n", k, i);
    i = i * 2;
  }
}
```

- Compile and run

```
$ gcc -o integer integer.c
$ ./integer
```

Integer Numbers

- Result

```
$ gcc -o integer integer.c
$ ./integer
 0:  1000000
 1:  2000000
 2:  4000000
 3:  8000000
 4:  16000000
 5:  32000000
 6:  64000000
 7:  128000000
 8:  256000000
 9:  512000000
10:  1024000000
11:  2048000000
12:  -198967296
13:  -397934592
14:  -795869184
15:  -1591738368
16:  1111490560
17:  -2071986176
18:  150994944
19:  301989888
$
```

```
#include <stdio.h>

void main(void)
{
  int i = 1000000;
  int k;

  for (k = 0; k<20; k++) {
    printf("%2d:  %d\n", k, i);
    i = i * 2;
  }
}
```

Real Numbers

- real.c

```
#include <stdio.h>

void main(void)
{
  float a = 1e30;
  float b = 3.14159265358979;

  printf(" 1. %f + %f - %f = %f\n", a,b,a,a+b-a);
  printf(" 2. %f - %f + %f = %f\n", a,a,b,a-a+b);
}
```

- Compile and run

```
$ gcc -o real real.c
$ ./real
```

Real Numbers

- Result

```c
#include <stdio.h>

void main(void)
{
  float a = 1e30;
  float b = 3.14159265358979;

  printf(" 1. %f + %f - %f = %f\n", a,b,a,a+b-a);
  printf(" 2. %f - %f + %f = %f\n", a,a,b,a-a+b);
}
```

```
$ gcc -o real real.c
$ ./real
 1. 1000000015047466219876688855040.000000 + 3.141593 - 1000000015047466219876688855040.000000 = 0.000000
 2. 1000000015047466219876688855040.000000 - 1000000015047466219876688855040.000000 + 3.141593 = 3.141593
$
```

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

/* memcpy definition */
void *memcpy(void *dest, const void *src, size_t n);
```

- Similar to code found in FreeBSD's implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

/* memcpy definition */
void *memcpy(void *dest, const void *src, size_t n);
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

/* memcpy definition */
void *memcpy(void *dest, const void *src, size_t n);
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```

To Understand What Goes Wrong

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

/* memcpy definition */
void *memcpy(void *dest, const void *src, size_t n);
```

```c
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```

Computer Arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties
- Cannot assume all “usual” mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs
- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Reality Check #2: You've Got to Know Assembly

- Chances are, you'll never write programs in assembly
  - Compilers are much better & more patient than you are

- But: Understanding assembly is key to machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Assembly Code Example

- Time Stamp Counter
  - Special 64-bit register in Intel-compatible machines
  - Incremented every clock cycle
  - Read with rdtsc instruction

- Application
  - Measure time (in clock cycles) required by procedure

```
double t;
start_counter();
P();
t = get_counter();
printf("P required %f clock cycles\n", t);
```

Code to Read Counter

- Write small amount of assembly code using GCC's asm facility
- Inserts assembly code into machine code generated by compiler

```
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits
   of the cycle counter.
*/
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
        : "=r" (*hi), "=r" (*lo)
        :
        : "%edx", "%eax");
}
```

Reality Check #3: Random Access Memory

- MatrixAdd.java

```
import java.util.*;

class MatrixAdd
{
  static int N=4096;
  static int R=16;

 static private long MatAdd(int N, int A[][],
                            int B[][], int sum[][])
  {
    long start = System.nanoTime();
    int i,j;

    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        sum[j][i] = A[j][i] + B[j][i];
      }
    }

    long stop  = System.nanoTime();

    return stop - start;
  }

...
```

```
...

  public static void main(String args[])
  {
    System.out.format("Adding two %,d x %,d matrices.%n",
                      N, N);

    int a[][] = new int[N][N];
    int b[][] = new int[N][N];
    int c[][] = new int[N][N];
    int i,j;

    Random rand = new Random();

    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
        a[j][i] = rand.nextInt(65536);
        b[j][i] = rand.nextInt(65536);
      }
    }

    long total = 0;

    for (i=1; i<=R; i++) {
      long t = MatAdd(N, a, b, c);
      total += t;
      System.out.format(" %2d. run: %,d%n", i, t);
    }

    System.out.format("Average runtime: %,d.%n",
                      total / R);
  }
}
```

- Compile and run

```
$ javac MatrixAdd.java
$ java MatrixAdd
```

Reality Check #3: Random Access Memory

- Result

```
$ javac MatrixAdd.java
$ java MatrixAdd
Adding two 4,096 x 4,096 matrices.
  1. run: 1,167,219,175
  2. run: 1,170,854,849
  3. run: 1,157,460,403
  4. run: 1,146,491,235
  5. run: 1,131,016,361
  6. run: 1,127,804,280
  7. run: 1,176,178,946
  8. run: 1,174,429,930
  9. run: 1,123,144,337
 10. run: 1,148,392,584
 11. run: 1,158,618,876
 12. run: 1,143,570,880
 13. run: 1,136,663,775
 14. run: 1,144,929,208
 15. run: 1,130,314,085
 16. run: 1,145,745,940
Average runtime: 1,148,927,179.
```

- Then your friend edits the file for 3 seconds, recompiles and runs it.

Reality Check #3: Random Access Memory

- Result of modified source

```
$ javac MatrixAdd.java
$ java MatrixAdd
Adding two 4,096 x 4,096 matrices.
  1. run: 22,371,258
  2. run: 15,699,727
  3. run: 17,192,265
  4. run: 16,826,659
  5. run: 17,135,681
  6. run: 16,873,783
  7. run: 16,837,033
  8. run: 17,470,509
  9. run: 17,435,977
 10. run: 17,528,881
 11. run: 17,451,922
 12. run: 17,427,301
 13. run: 17,342,101
 14. run: 17,388,893
 15. run: 17,271,563
 16. run: 17,318,061
Average runtime: 17,473,225.
```

- Factor 1,148,927,179 / 17,473,225 = 65 faster ?!

Reality Check #3: Random Access Memory

- Source comparison

Your code:

```
import java.util.*;

class MatrixAdd
{
  static int N=4096;
  static int R=16;

 static private long MatAdd(int N, int A[][],
                      int B[][], int sum[][])
  {
    long start = System.nanoTime();
    int i,j;

    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        sum[j][i] = A[j][i] + B[j][i];
      }
    }

    long stop  = System.nanoTime();

    return stop - start;
  }

...
```

Your friend's code:

```
import java.util.*;

class MatrixAdd
{
  static int N=4096;
  static int R=16;

 static private long MatAdd(int N, int A[][],
                      int B[][], int sum[][])
  {
    long start = System.nanoTime();
    int i,j;

    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
        sum[j][i] = A[j][i] + B[j][i];
      }
    }

    long stop  = System.nanoTime();

    return stop - start;
  }

...
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Random Access Memory Is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting programs to characteristics of a memory system can lead to major speed improvements

Memory System Performance Example

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

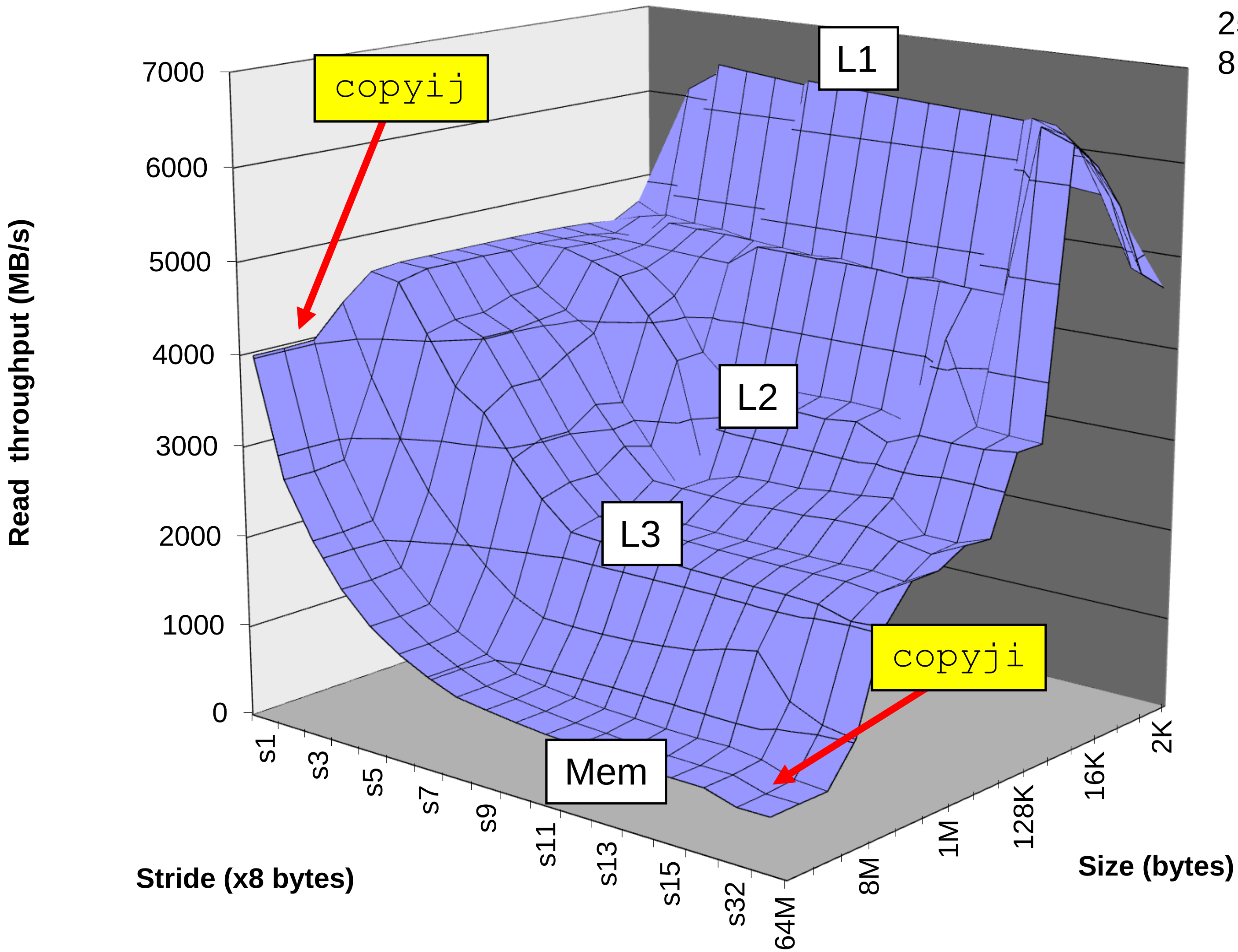
```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

Effect of swapping two lines:
21 times slower (Pentium 4)
6 times slower (Core i7)

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

The Memory Mountain

Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache



Memory Referencing Bug Example

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

- Result is architecture specific

```
fun(0)  →    3.14
fun(1)  →    3.14
fun(2)  →    3.1399998664856
fun(3)  →    2.00000061035156
fun(4)  →    3.14, then segmentation fault
```

Memory Referencing Bug Example

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  →   3.14
fun(1)  →   3.14
fun(2)  →   3.1399998664856
fun(3)  →   2.00000061035156
fun(4)  →   3.14, then segmentation fault
```

Explanation:

Saved State	4
d[0] (high 32bits)	3
d[0] (low 32bit)	2
a[1]	1
a[0]	0

Location accessed by `fun(i)`

Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in a language with runtime bound checks (Java, Ruby, Pascal, ML,…)
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

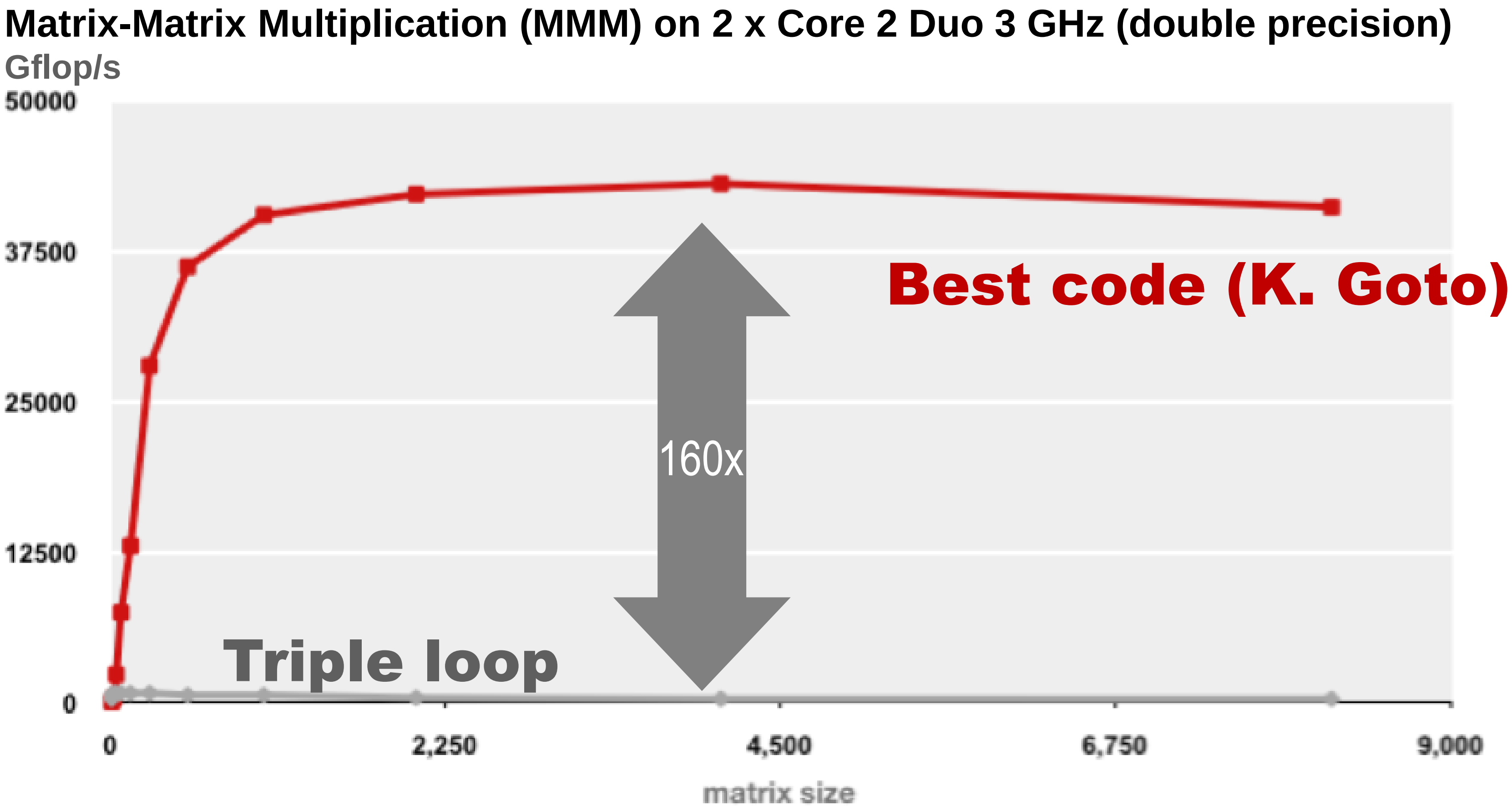
Reality #4: There's more to performance than asymptotic complexity

- Constant factors matter too!

- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops

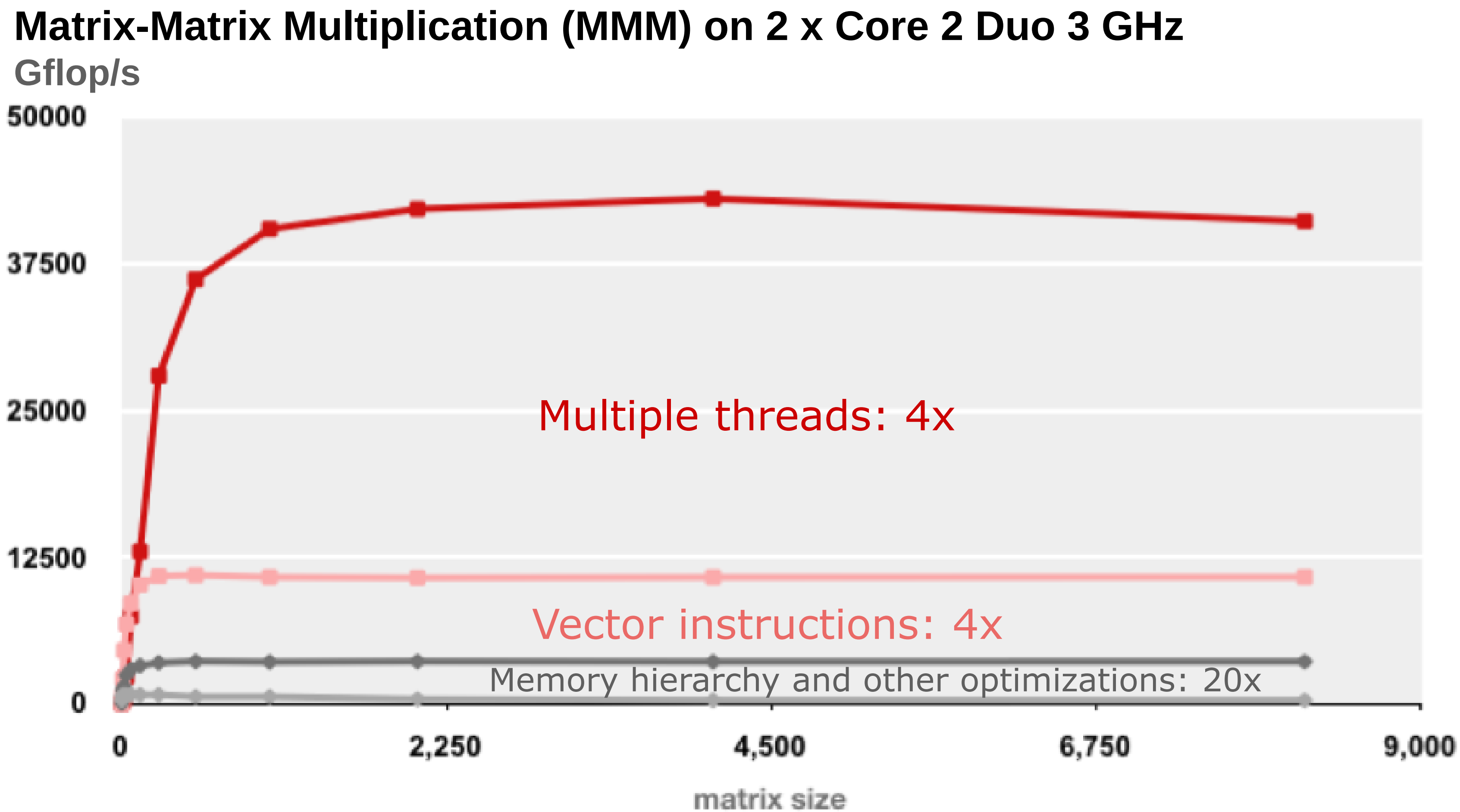
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Example: Matrix Multiplication



- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?

MMM Plot: Analysis



- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- Effect: fewer register spills, L1/L2 cache misses, and TLB misses

Reality #5: Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

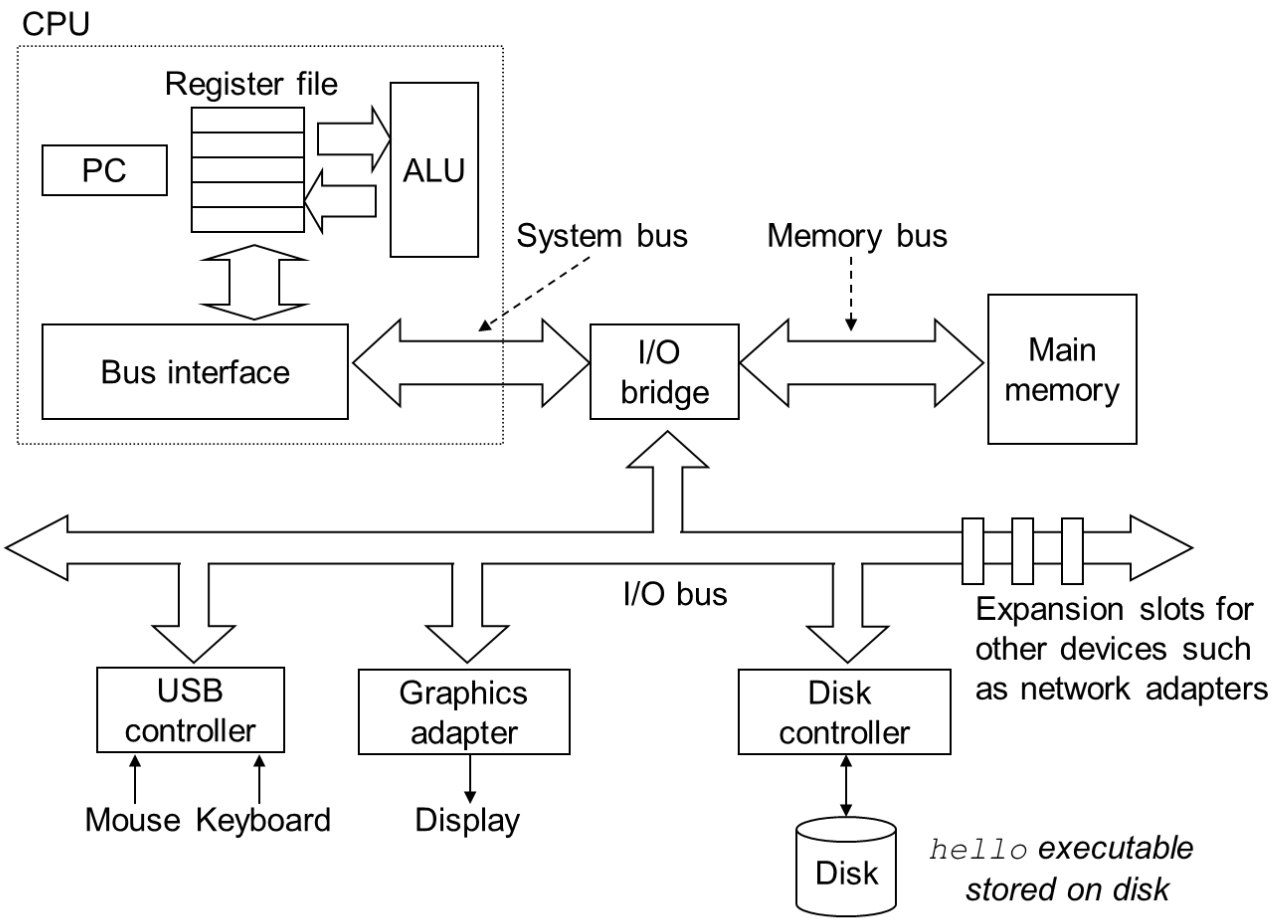
- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Introduction to System Programming

- Abstractions are good but don’t forget reality
- **Basic operation of a computer system**
- Summary

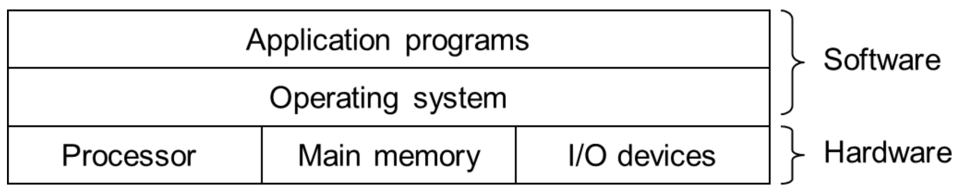
Hardware Organization

- Typical hardware organization

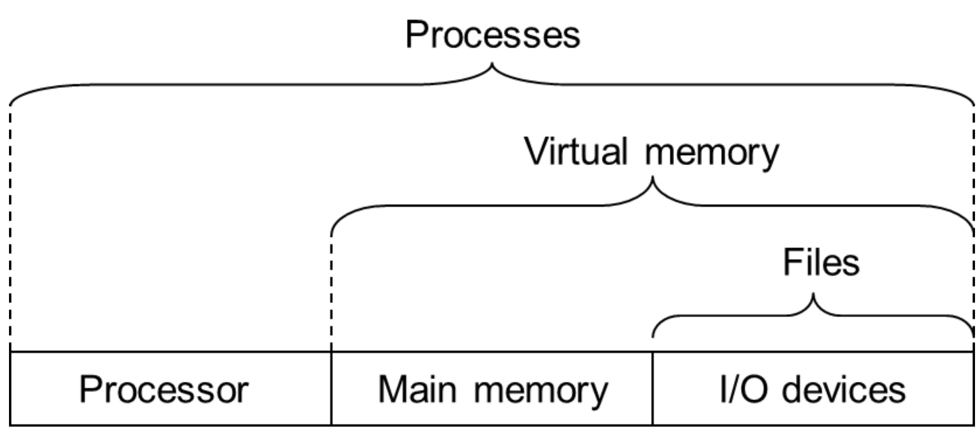


Operating System Basics

- The operating system manages the hardware
  - protect H/W from misuse by buggy/malicious programs
  - provide simple and uniform mechanisms for manipulating hardware devices



- Fundamental abstractions provided by the OS
  - processes
  - virtual memory
  - files



Processes and Threads

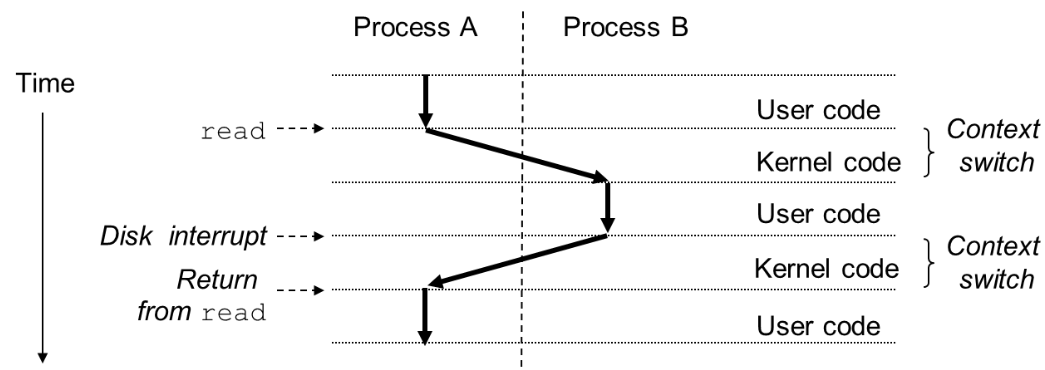
- A process is the operating systems abstraction for a running program

- multiple processes can run concurrently
  - multi-core processors: true parallelism
  - single-cores: apparent parallelism through context-switching

- OS provides the illusion that the process has exclusive access to the H/W
  - full memory address space
  - exclusive access to the I/O devices

Processes and Threads

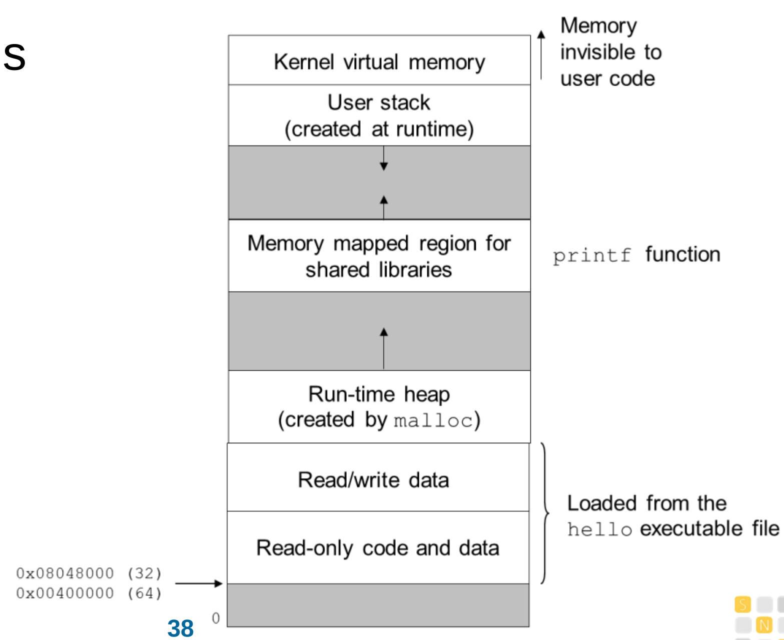
- Context-Switching
  - OS keeps track of all state (context) that a process needs in order to run
    - current PC
    - register file
    - memory contents
    - open files
  - switch to a new process
    - periodically
    - when a process has to wait for an event



- A process can consist of multiple *threads*
  - not as heavy as full processes
  - easier sharing of data
  - typically more efficient scheduling

Virtual Memory

- An abstraction of the physical memory
- Provides each process with the illusion that it has exclusive use of the main memory
- Managed by the OS with the help of a hardware translation unit, the MMU (memory management unit)
- Typical layout on Linux systems



Files

- a file is a sequence of bytes

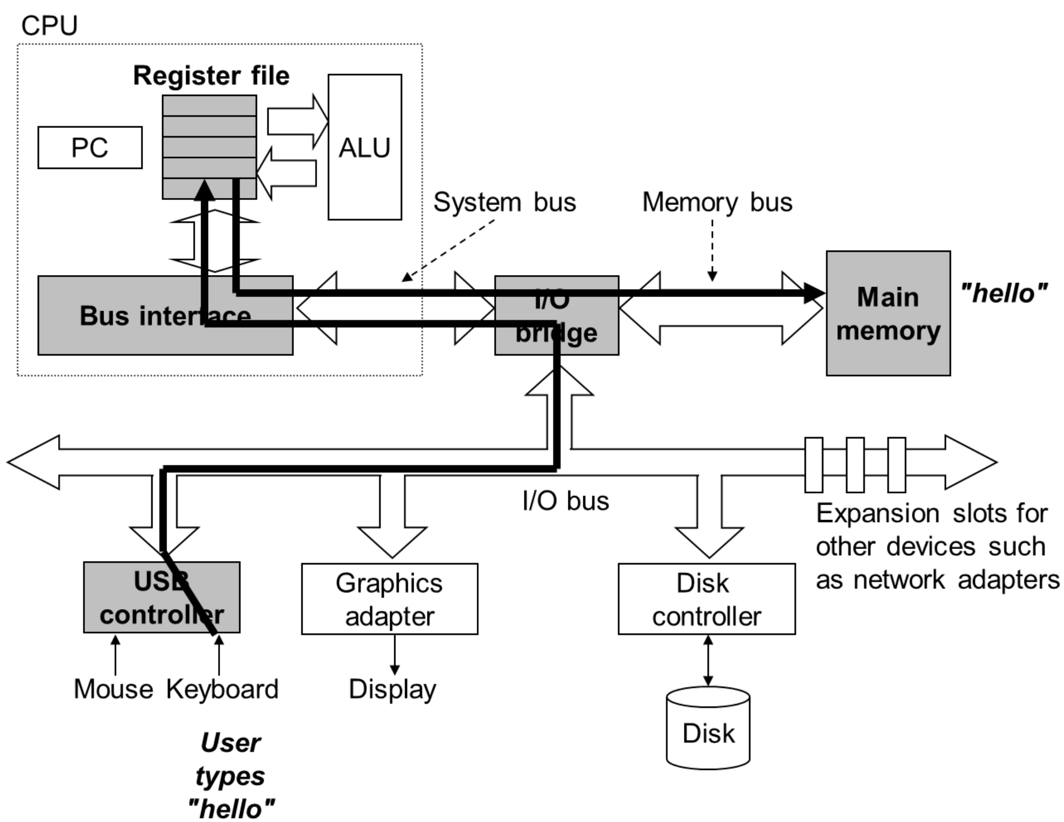
- in *nix systems, “everything” is modeled as a file
  - disk
  - keyboard
  - mouse
  - display
  - network
  - shared memory

- single interface to interact with files

CSE 컴퓨터공학부 Department of Computer Science & Engineering

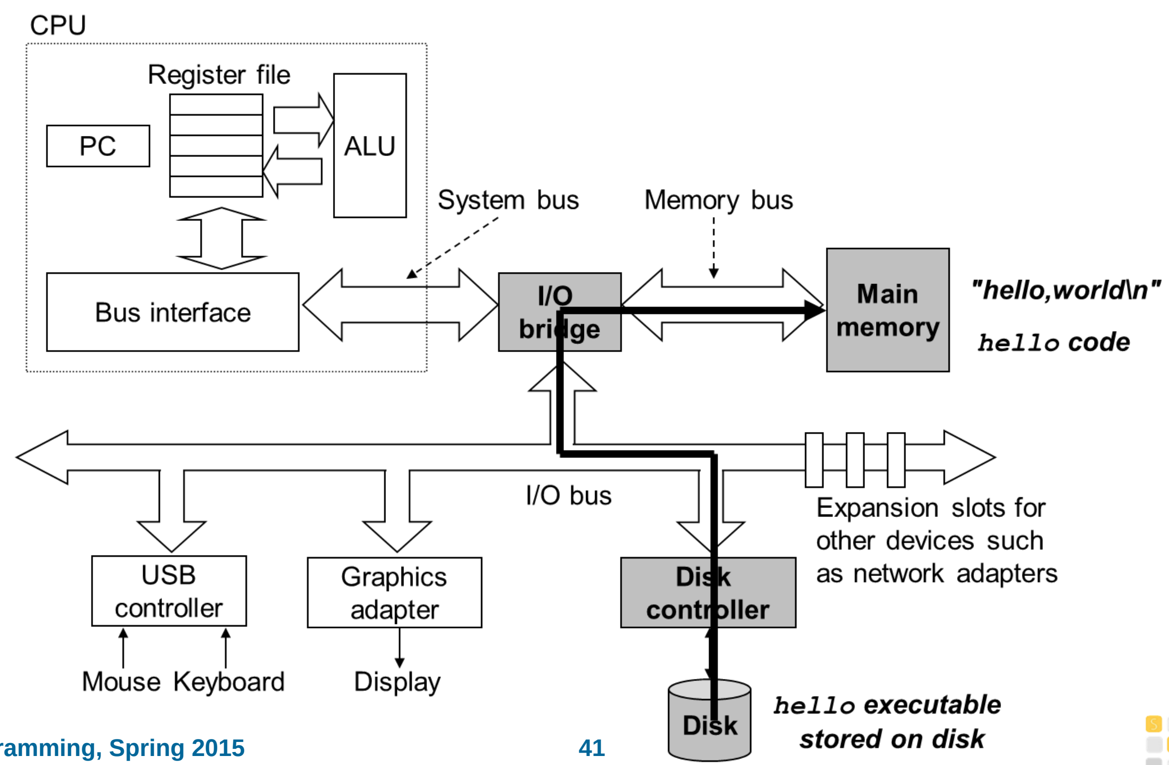
Running a Program on a System

- Running `hello`
  - shell reads input from keyboard, one character at a time into a register and from there to the main memory



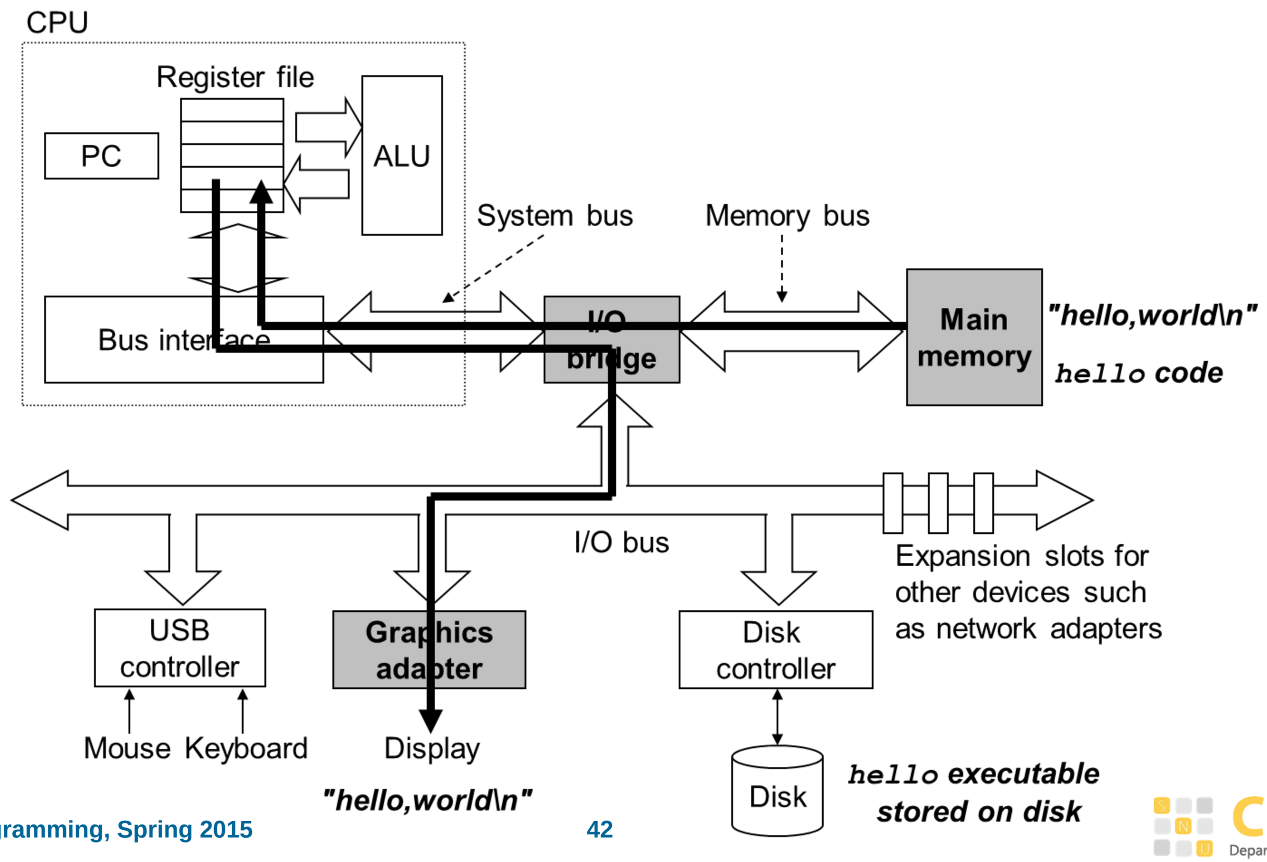
Running a Program on a System

- Running `hello`
  - <enter> signals the end of the command to the shell
  - the shell searches for a file named 'hello' in the file system
  - loads the program from disk into memory
  - starts running the program



Running a Program on a System

- Running `hello`
  - the processor begins executing the machine-instructions
  - copies the string ‘hello, world\n’ to registers, and from there to the display adapter via *system calls*



Introduction to System Programming

- Abstractions are good but don't forget reality
- Basic operation of a computer system
- **Summary**

System Programming

- System software (or systems software) is computer software or an operating system designed to operate and control the computer hardware and to provide a platform for running application software.
  (source: Wikipedia)

- System Programming
  - concerns itself with writing system software
  - from relatively high-level (web servers) down to very low-level (boot loaders)

Wrapping it up

- In this class you will learn
  - learn how to interact with the operating system on a low level
  - get an idea of what the operating system is doing
  - how to write safer code
  - how to write faster code

- Prerequisites
  - some programming experience
  - computer architecture
  - x86 assembly
  - SP will require a substantial effort
    - read the book
    - do the homework
    - start with labs early!

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Reading Assignment

- For Thursday, March 4
  - chapter 1 of the text book