COMP3511 Lab2

C/C++ programming in Linux

Ren, Zhenghang

zrenak@cse.ust.hk

Contents

- Review of Core Concepts of C language with Demo
- Workflow with Suggestions
  - Coding
  - Compiling
  - Debugging
- C Programming in Linux Environment
  (including summary of first two chapters and review of computer organizations, including I/O subsystem. Basic concepts of process and threads)

Why C ?

- Most of the operating systems are written in C[1]
- Precise control on the underlying hardwares

[1] *The benefits and costs of writing a POSIX kernel in a high-level language* (OSDI ‘18)

Basic Data Types

All C data types have fixed size in bits

- Primary data types
- Derived data types (pointers, arrays, functions…)
- User defined data types

Type	Size(in bits)	Format Specifiers	Range
char or signed char	1	%c	-128 to 127
unsigned char	1	%c	0 to 255
int or signed int	4	%d,%i	-2,147,483,648 to 2,147,483,647
short int or signed short int	2	%hd	-32,768 to 32,767
unsigned short int	2	%hu	0 to 65,535
long int or signed long int	8	%ld,%li	-2,147,483,648 to 2,147,483,647
unsigned long int	8	%lu	0 to 4,294,967,295
float	4	%f	
double	8	%lf	
long double	16	%Lf	

Quick Demos

[Demo 1] Hello World

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

[Case study 1] Hello World

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Step 1 Include headers if needed

#include <helper_1>

#include <helper_2>

...

<math.h>	Common mathematics functions
<setjmp.h>	Nonlocal jumps
<signal.h>	Signal handling
<stdalign.h> (since C11)	alignas and alignof convenience macros
<stdarg.h>	Variable arguments
<stdatomic.h> (since C11)	Atomic types
<stdbool.h> (since C99)	Boolean type
<stddef.h>	Common macro definitions
<stdint.h> (since C99)	Fixed-width integer types
<stdio.h>	Input/output
<stdlib.h>	General utilities: memory management, program utilities, string conversions, random numbers
<stdnoreturn.h> (since C11)	noreturn convenience macros
<string.h>	String handling

[2] C header files https://en.cppreference.com/w/c/header

[Case study 1] Hello World

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Step 2 Craft your own innovations

Fundamental tips on syntax:

- Use semicolon ; to end a clause
- Use brackets ( ) to wrap a list of function arguments
- Use curly brackets { } to wrap a code block
- Function signature:
  - type_qualifier of the return value
  - function name
  - a list of arguments

[Demo 2] Swapping Two Intergers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Things you still need to know

1. What **data type**[3] you can use and when to declare a type

- the type `void`
- basic types
  - the type `char`
  - signed integer types
    - standard: `signed char`, `short`, `int`, `long`, `long long` (since C99)
    - extended: implementation defined, e.g. `__int128`
  - unsigned integer types
    - standard: `_Bool` (since C99), `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long` (since C99)
    - extended: implementation-defined, e.g. `__uint128`
  - floating types
    - real floating types: `float`, `double`, `long double`
    - complex types: `float _Complex`, `double _Complex`, `long double _Complex`
    - imaginary types: `float _Imaginary`, `double _Imaginary`, `long double _Imaginary`
- enumerated types
- derived types
  - array types
  - structure types
  - union types
  - function types
  - pointer types
  - atomic types

[3] https://en.cppreference.com/w/c/language/type

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Things you still need to know

1. What **data type** you can use and when to declare a type
2. What **operators** you can have
   - and their precedence and associativity[4]

3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	

[4] https://en.cppreference.com/w/c/language/operator_precedence

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Things you still need to know

1. What **data type** you can use and when to declare a type
2. What **operators** you can have
   - and their precedence and associativity
3. What are the **scope (lifetime) and visibility** of your variables
   - on code block basis; shadow principle

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Things you still need to know

1. What **data type** you can use and when to declare a type
2. What **operators** you can have
   - and their precedence and associativity
3. What are the **scope (lifetime) and visibility** of your variables
   - on code block basis; shadow principle
4. How to **format** a string for printf()[5]

   other examples:

```c
char ch = 'A';
printf("%c\n", ch);
```

```
A
```

```c
float a = 12.67;
printf("%f\n", a);
printf("%e\n", a);
```

```
12.670000
1.267000e+01
```


[5] http://www.cplusplus.com/reference/cstdio/printf/

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Of course, you also need to know about pointers—the spirit of C

- why pointers: arguments are passed by value
- what is a pointer: the address (in virtual memory) of a variable
- related syntax:
  - * (indirection/dereference opeartor):
    - ➢ when used in declaring a variable
      - indicating that the variable is a pointer

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Of course, you also need to know about pointers—the spirit of C

- why pointers: arguments are passed by value
- what is a pointer: the address (in virtual memory) of a variable
- related syntax:
  - * (indirection/dereference opeartor):
    - ➢ when used in declaring a variable
      - indicating that the variable is a pointer
    - ➢ when used as operators in an expression
      - referring to the value pointed by the pointer

[Case study 2] Swapping Two Integers

```c
#include <stdio.h>

int outer_a = 7;
int b = 1;

void swapInt(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = outer_a;
    int b = 100 + 200 / 10 - 3 * 10;
    printf("Before: %d %d\n", a, b);
    swapInt(&a, &b);
    printf("After : %d %d\n", a, b);
    return 0;
}
```

Of course, you also need to know about pointers—the spirit of C

- why pointers: arguments are passed by value
- what is a pointer: the address (in virtual memory) of a variable
- related syntax:
  - * (indirection/dereference opeartor):
    - ➢ when used in declaring a variable
      - indicating that the variable is a pointer
    - ➢ when used as operators in an expression
      - referring to the value pointed by the pointer

  & (address-of opeartor):
  - fetching the address of a variable (resulting in a pointer)

Bridging C++ to C

Utility in C	**Counterpart in C++**

Struct

Utility in C

You can declare a composite data type whose components are physically grouped in memory

```c
struct IntPair {
    int first, second;
};

int IntPair_sum(struct IntPair pair)
{
    return pair.first + pair.second;
}
```

Counterpart in C++

```cpp
struct IntPair {
public:
    IntPair(): first(0), second(0) {}
    int sum() {return first+second; }

private:
    int first, second;
};
```

As opposed to C, C++:

- Do not need separated functions
- Can have access modifiers

Dynamic Memory

Utility in C

You can allocate and release heap memory on demand using malloc[6] / free defined in <stdlib.h>

```
int *intVar = malloc(sizeof(int));
int *intArr = malloc(sizeof(int)*100);
struct IntPair *intPair =
malloc(sizeof(struct IntPair));

free(intVar);
free(intArr);
free(intPair);
```

Counterpart in C++

```
int *intVar = new int;
int *intArr = new int[100];
IntPair *intPair = new IntPair;

delete intVar;
delete [] intArr;
delete intPair;
```

As opposed to C, C++:

- Use new/delete operators

[6] http://www.cplusplus.com/reference/cstdlib/malloc/

I/O

Utility in C

Use scanf[7] / printf defined in <stdio.h>

```
int age;
char name[100];

printf("Enter your name: ");
scanf("%s", name);
printf("Enter your age: ");
scanf("%d", &age);
```

Counterpart in C++

```
int age;
string name;

cout << "Enter your name: " ;
cin >> name ;
cout << "Enter your age: " ;
cin >> age;
```

As opposed to C, C++:

- Use cin, cout defined in <iostream>

[7] http://www.cplusplus.com/reference/cstdio/scanf/

File I/O

Utility in C

Use fopen/ fscanf[8] / fprintf / fclose defined in

<stdio.h>

```
#include <stdio.h>
int main()
{
    int a, b;
    FILE *fin = fopen("input.txt", "r");
    fscanf(fin, "%d %d", &a, &b);
    fclose(fin);

    int result = a + b;
    FILE *fout = fopen("output.txt","w");
    fprintf(fout, "The sum of %d & %d is
%d\n",
            a, b, result);
    fclose(fout);

    return 0;
}
```

Counterpart in C++

```
#include <fstream>
using namespace std;
int main()
{
    int a, b;
    ifstream fin("input.txt");
    fin >> a >> b ;
    fin.close();
    int result = a + b;
    ofstream fout("output.txt");
    fout << "The sum of " << a << " & "
         << b << " is " << result <<
endl;
    fout.close();
    return 0;
}
```

As opposed to C, C++:

- Use fin, fout defined in <ifstream> <ofstream>

[8] http://www.cplusplus.com/reference/cstdio/fscanf/

String Manipulation

String in C is a NULL-terminated Character array

You can copy a string using strcpy in <string.h>

```
#include <stdio.h>
#include <string.h>

int main()
{
    char str[100];
    strcpy(str, "Hello World!");

    int i = 0;
    while ( str[i] != '\0' && i < 100)
        i++;
        printf("Last char: %c\n", str[i-
1]);

    return 0;
}
```

index	0											12		99
str	H	e	l	l	o		W	o	r	l	d	!	\0	…	

You can manipulate string like files using sscanf defined in <string.h>

String in C++[9]

```
#include <stdio.h>
#include <string.h>
int main()
{
    char line[100], linuxText[20],
machineName[20], kernelText[50];
    int kernelMajor, kernelMinor;

    strcpy(line, "Linux csl2wk19 3.10.0-
327.3.1.el7.x86_64");
    sscanf(line, "%s %s %s", linuxText,
machineName, kernelText);
    sscanf(kernelText, "%d.%d", &kernelMajor,
&kernelMinor);
    printf("Version: %d.%d\n", kernelMajor,
kernelMinor);

    return 0;
}
```


[9] https://www.cplusplus.com/reference/string/string/

Workflow with Suggestions

The Minimal Workflow

Code at remote host:

- vi/vim [source code files]

Minimized Compilation

- gcc [source code files] –o [output file]

Running

- ./[output file]

Code at local desktop:

- First using desktop editors
- Then upload the files via
  - scp, SFTP, Git, etc.

Advanced topics

- Linking external libraries
- Generating debugging information
- Using Make for automation [Demo]

Debugging

- Using GDB [Demo]

Will be discussed in Part 3

code

compile

run/test

[Manual 1] Upload via SFTP

- Filezilla Client can be used to upload and download files from machines which support SFTP

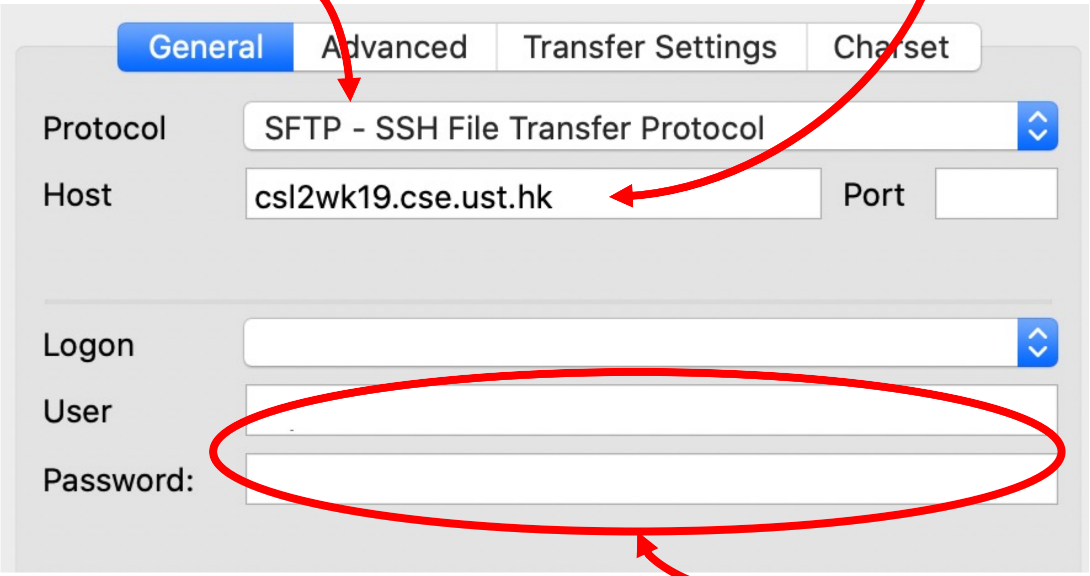
- Please DO NOT install Filezilla Server

[Manual 1] Upload via SFTP (Cont.)

- Setting up a Site Manager
  - File > Site Manager...
  - Press the "New site" button and give an appropriate name (e.g. CS Linux Lab Machine)
  - Press the "Connect" button
- Next time, you just need to select the site you created and then press the "Connect" button

Choose SFTP as the transfer protocol

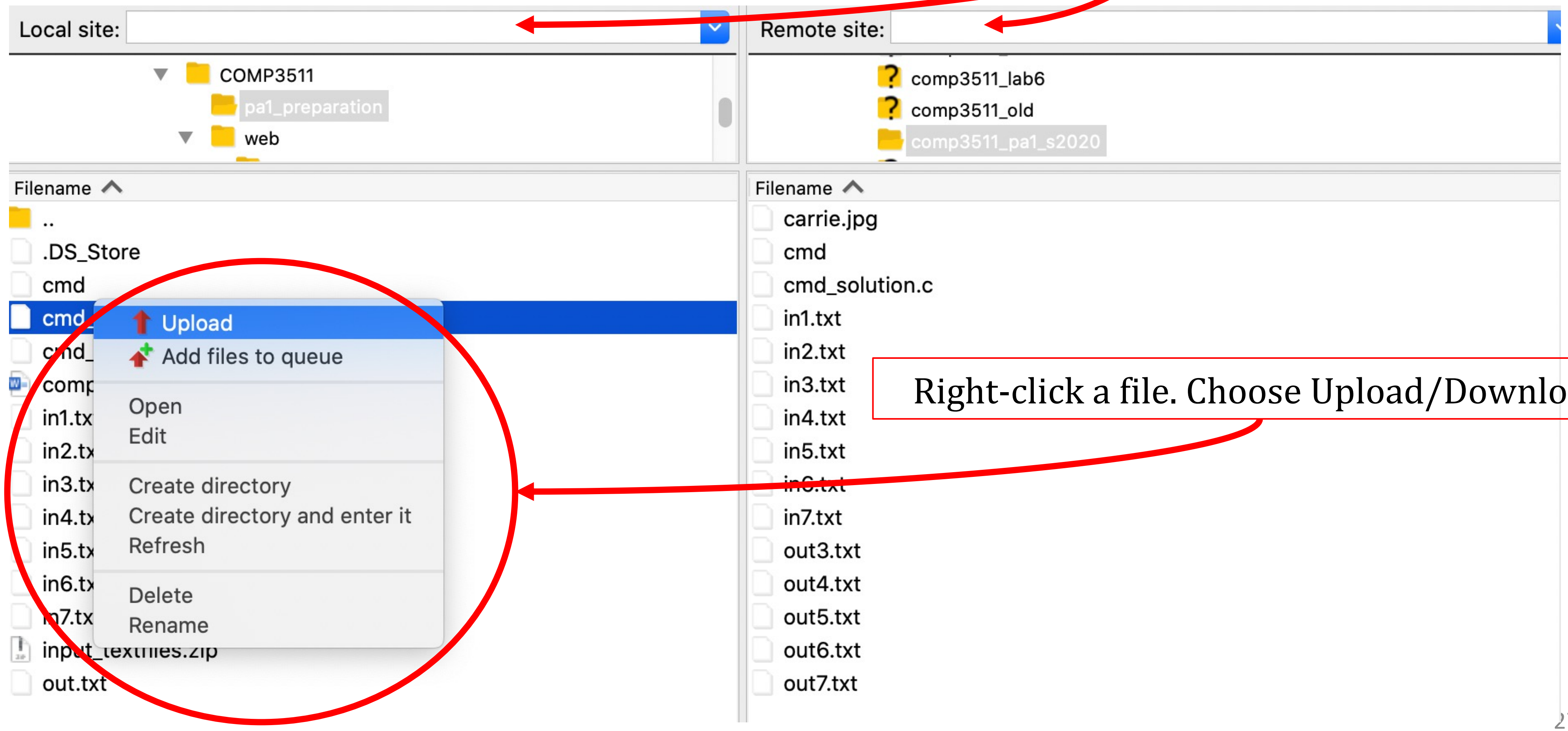
Enter a machine name (csl2wk**XX**.cse.ust.hk, where **XX**=01-40)



Enter your username and password

[Manual 1] Upload via SFTP (Cont.)

Choose a directory in a local machine and a remote machine (Linux)



Right-click a file. Choose Upload/Download

The Complete Pipeline

Code at remote host:

- vi/vim [source code files]

Minimized Compilation

- gcc [source code files] –o [output file]

Running

- ./[output file]

Code at local desktop:

- First using desktop editors
- Then upload the files via
  - scp, SFTP, Git, etc.

Advanced topics

- Linking external libraries
- Generating debugging information
- Using Make for automation [Demo]

Debugging

- Using GDB [Demo]

code

compile

run/test

Part 3

Advanced Compilation Options

- Specify external libraries (-l) to link

  e.g. gcc foo.c –o foo –lmath

  “math” refers to the C math library, which gcc does not link to automatically

  it is used when we our code needs to includes <math.h>

- Generating information for debugging (-g)

  e.g. gcc buggy.c –o buggy –g

  (so that later you can execute “gdb swap”)

The Complete Pipeline

Code at remote host:

- vi/vim [source code files]

Code at local desktop:

- First using desktop editors
- Then upload the files via
    - scp, SFTP, Git, etc.

Minimized Compilation

- gcc [source code files] –o [output file]

Advanced topics

- Linking external libraries
- Generating debugging information
- Using Make for automation [Demo]

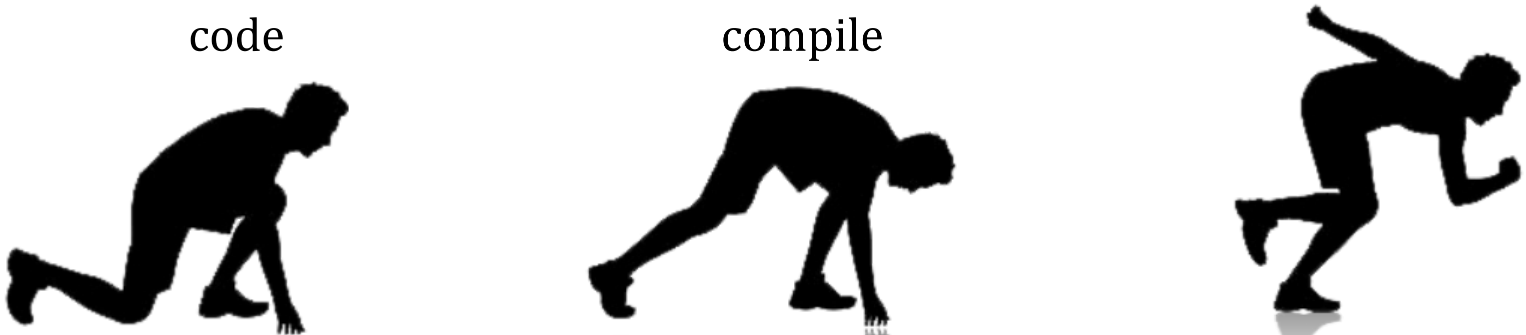
Running

- ./[output file]

Debugging

- Using GDB [Demo]

code compile run/test



Part 3

Using Make for Automation

- The `make` utility is a program to automate the compilation and execution of programs
  - Advanced usage[9] of make is tricky
  - In this course, we only need to learn the basics
- `Makefile` is a configuration file to instruct the make utility how to compile and run the programs
- `Makefile` consists of multiple rules. The syntax is as follows:

```
Target: pre-requisites
<TAB> command 1 to build the target
<TAB> command 2 to build the target
…
```


[9] https://www.gnu.org/software/make/manual/

Using Make for Automation (Cont.)

[Demo] Compile multiple programs

- Suppose you want to compile 2 executable programs (`hello` and `io_console`)
- You can create a minimal Makefile with the following 4 rules:

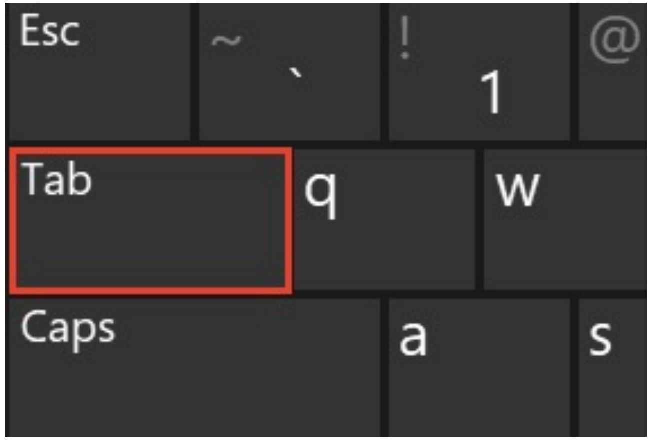
Tabs MUST be used,
Spaces are not allowed

```
all: hello io_console

hello: hello.c
	gcc -o hello hello.c

io_console: io_console.c
	gcc -o io_console io_console.c

clean:
	rm -f hello io_console
```



Using Make for Automation (Cont.)

Expected results

```
$> ls hello.c io_console.c Makefile
Makefile     hello.c             io_console.c

$> make clean
rm -f hello io_console

$> make
gcc -o hello hello.c
gcc -o io_console io_console.c

$> ./hello
Hello World!

$> ./io_console
Enter your name:
```

Learning makefile

https://makefiletutorial.com

The Complete Pipeline

Code at remote host:

- vi/vim [source code files]

Code at local desktop:

- First using desktop editors
- Then upload the files via
  - scp, SFTP, Git, etc.

Minimized Compilation

- gcc [source code files] –o [output file]

Advanced topics

- Linking external libraries
- Generating debugging information
- Using Make for automation [Demo]

Running

- ./[output file]

Debugging

- Using GDB [Demo]

code

compile

run/test

Using GDB

- GDB stands for **GNU Debugger**
- Make sure that the file is compiled with "**-g**"
- Usage
  - Enter: gdb [program file]
  - Quit: quit or Ctrl+D
  - List program: list
  - Set breakpoint: break
  - Run until stop: run [args]
  - Print variable: print [variable name]

```
(gdb) l
1 #include <stdio.h>
2
3 int main()
4 {
5     int arr[5];
6     int i;
7     for (i=0;i<5;i++)
8         arr[i] = i;
9     for (i=0;i<=5;i++)

(gdb) break 9
Breakpoint 1 at 0x400544: file buggy.c,
line 9.

(gdb) run
Starting program:
/homes/cspeter/test/buggy

Breakpoint 1, main () at buggy.c:9
9     for (i=0;i<=5;i++)

(gdb) p arr
$1 = {0, 1, 2, 3, 4}
(gdb) p i
$2 = 5
```

Using GDB

- If no breakpoint is set, “run” will execute to the end
- Otherwise, it will stop at the first breakpoint it meets.
  - To continue, you have 3 options:
    - continue: GDB will continue executing until the next break point
    - next: GDB will execute the next line as a single instruction (even if it is a function call)
    - step: GDB will execute the next line, if it is a function call, it will step into it

Using GDB

[Demo] Debugging buggy

```c
#include <stdio.h>

int main() {
    int arr[5];
    int i;
    for (i=0;i<5;i++)
        arr[i] = i;
    for (i=0;i<=5;i++)
        printf("%d\n", arr[i]);
    return 0;
}
```

Learning GDB

https://www.cs.cmu.edu/~gilpin/tutorial/

C Programming in Linux OS

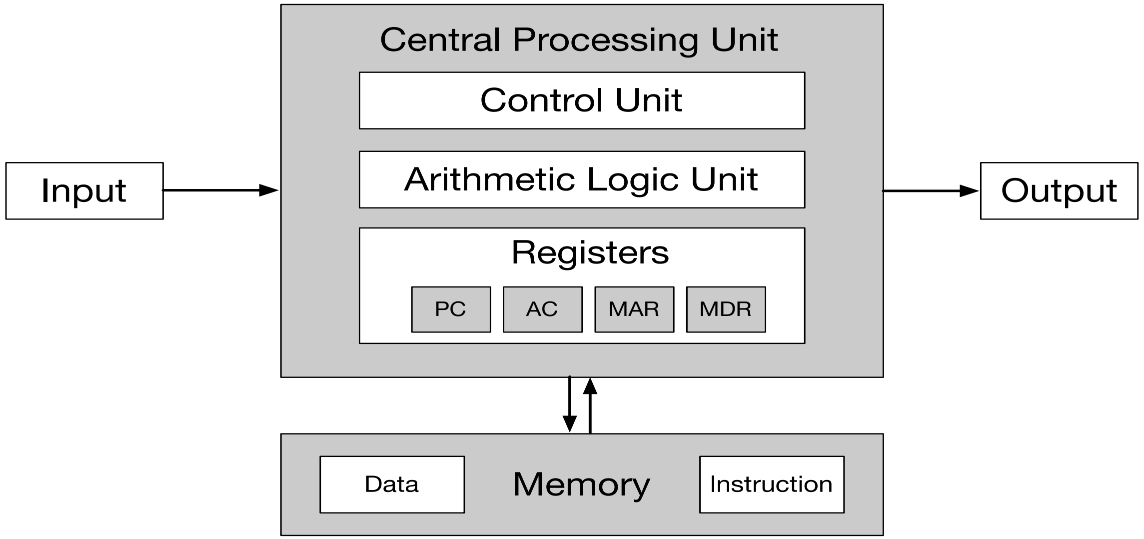
- Review of first two chapters
- More Details in I/O subsystem
- Process, Threads and Address Space

Chapter 1 Summary

- To best utilize the fast CPU, modern OSes employ multiprogramming, which allows many jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute
- Multitasking can be considered an extension of multiprogramming wherein CPU scheduling (to be discussed in Chapter 5) can rapidly switch CPU between executing different processes, providing users with a fast response time, which is important for interactive types of jobs
- Operating systems provide mechanisms for protecting. Protection measures the control of the resource (hardware and software) access available in computer systems.
- Virtualization is a technology that allows an OS to run as an application within another OS. It involves abstracting hardware into several different execution environments – each referred as a virtual machine.
- The virtual machine or VM creates an illusion for multiple processes in that each process "thinks" that it runs on a dedicate CPU with its own memory.

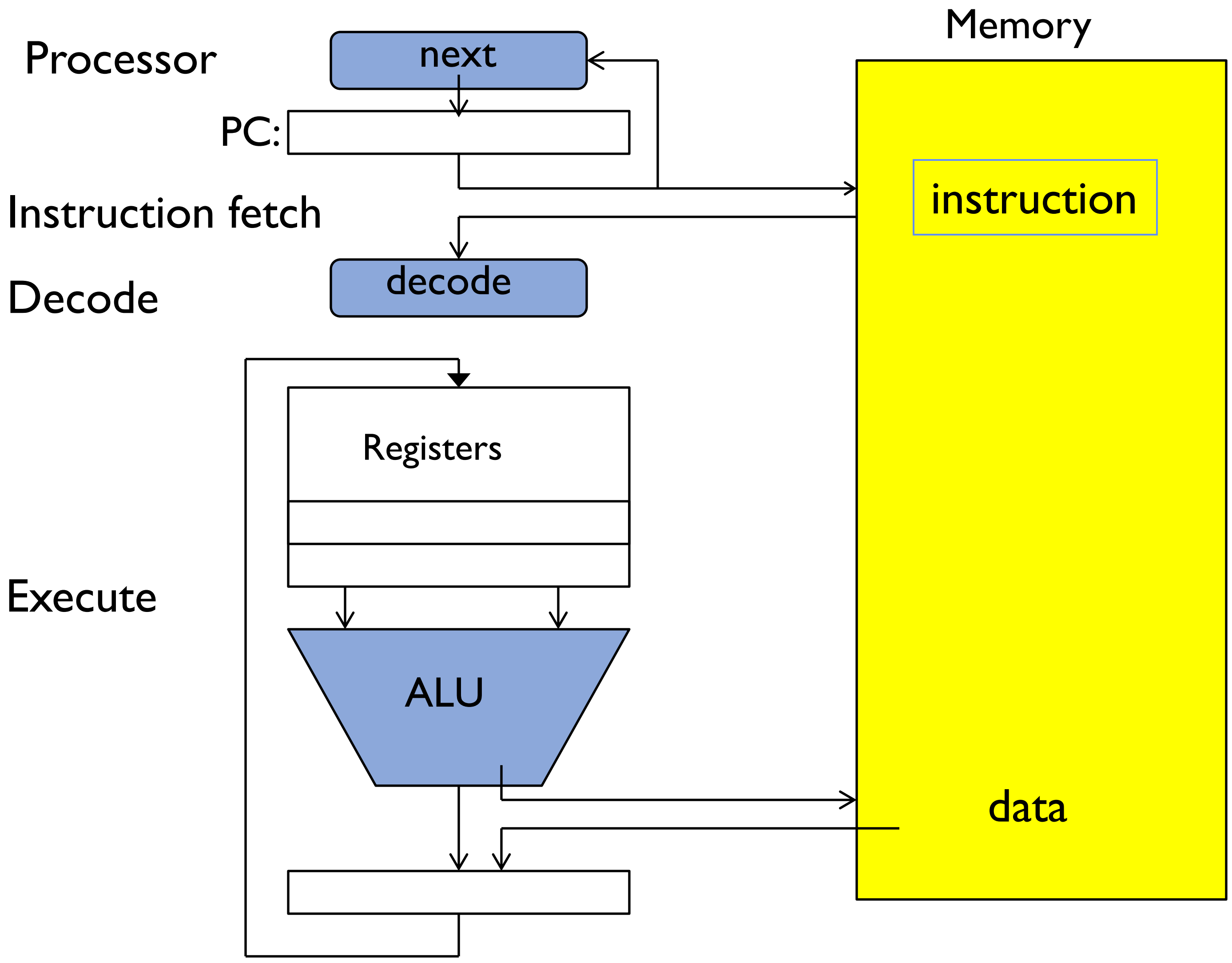
A Von Neumann Architecture

- A processing unit that contains an arithmetic logic unit (ALU) and processor registers
- A control unit that contains an instruction register (IR) and program counter (PC)
- Memory that stores data and instructions – along with caches
- External mass storage (not shown in the figure)
- Input and output mechanisms



Instruction Fetch/Decode/Execute

The instruction cycle



Chapter 2 Summary

- The three primary approaches for interacting with an operating system are (1) command interpreters (CLI or Shell), (2) graphical user interfaces, and (3) touchscreen interfaces
- **System calls** provide services made available by an operating system, where programmers use a system call's application programming interface (API) for accessing system-call services. The standard C library provides the system-call interface for UNIX and Linux
- Operating systems include a collection of **system programs** that provide utilities to users – so users can use the operating system services
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's **policies**. An operating system implements these policies through specific **mechanisms**

I/O Subsystems

- There are two main jobs of a computer: (1) computing on CPU and (2) I/O operation on a variety of I/O devices - in many cases, the main job is the I/O operation, e.g., word processor, web surfing, database queries
- I/O devices vary greatly in their function and speed, e.g., a mouse, a hard disk, a flash drive, and a robot – need different ways to manage them
- The basic I/O hardware elements include ports, buses, and device controllers, which accommodate a wide variety of I/O devices
- **Device drivers** – the OS software encapsulates device details - presents a uniform device-access interface – abstraction, to I/O subsystem, similar to system calls that provide a standard interface between an application and the operating system

I/O Subsystem Functions

- **Buffering** - storing data temporarily while it is being transferred
- **Caching** - faster device holding copy of data
  - A cache here refers to a region of fast memory that holds copies of data
  - The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, only holds a copy on faster storage of an item that resides elsewhere
- **Spooling** - hold output for a device which does not allow interleaved data, e.g., printers
  - It coordinates concurrent output, usually for slow peripheral devices
  - The term "spool" originates with the Simultaneous Peripheral Operations On-Line (IBM)

I/O Hardware

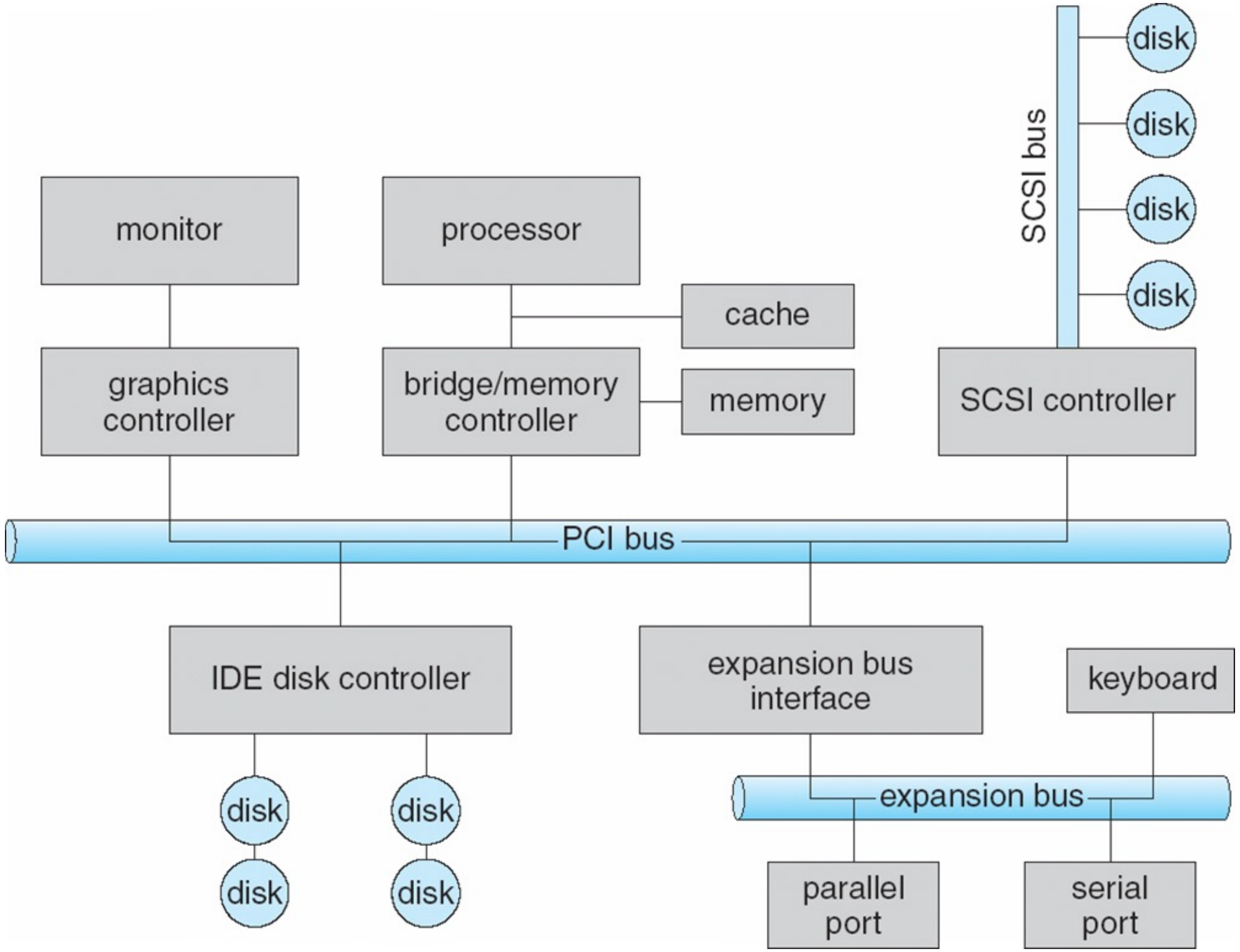
- A variety of I/O devices – storage, transmission, and human-interface
- Common concepts – signals from I/O devices interface with computer
  - **Port** – connection point for device, and **bus** for shared access
    - **PCI** bus common in PCs and servers, PCI Express (**PCIe**)
    - **Expansion bus** connects relatively slow devices
- **Controller** (**host adapter**) – electronics that operate port, bus, device
  - It contains processor, microcode, private memory, bus controller

I/O Hardware (Cont.)

- Devices usually have **registers** where device driver places commands, addresses, and data to write, or read data from registers after execution
  - Data-in register, data-out register, status register, control register
- Devices have addresses, used by I/O instructions

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

A Typical PC Bus Structure



Interrupts

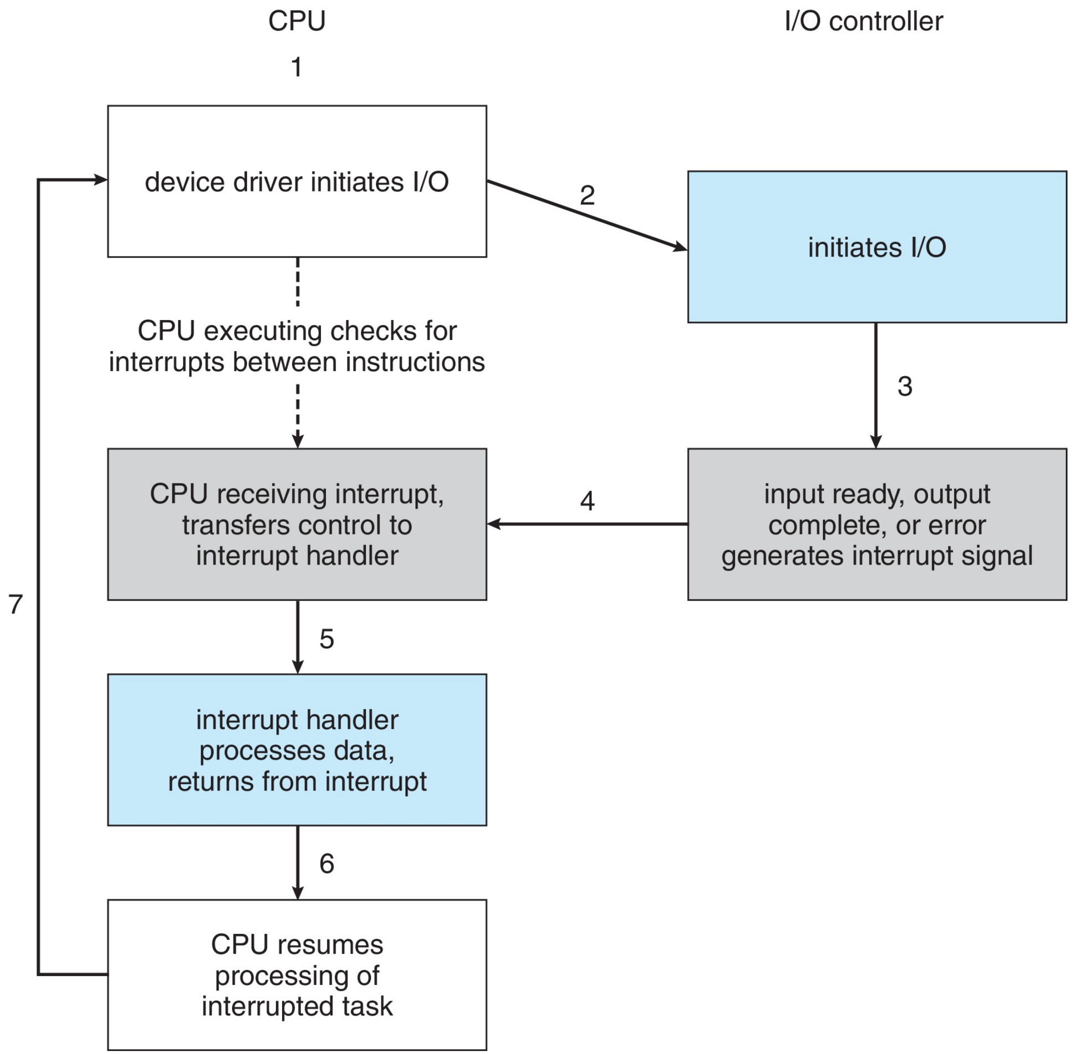
Interrupt enables the CPU to respond to asynchronous events, e.g., a device controller becomes ready for service. It has a few desirable features:

1. The ability to defer interrupt handling during critical processing – maskable vs. nonmaskable
2. To dispatch to the proper interrupt handler (OS kernel routine) for a specific device
3. Multilevel interrupts needed to distinguish between high- and low-priority interrupts, esp. when there are multiple concurrent interrupts
4. Get the OS attention directly (separately from I/O requests), for activities such as page faults and errors such as division by zero – traps or software generated interrupt

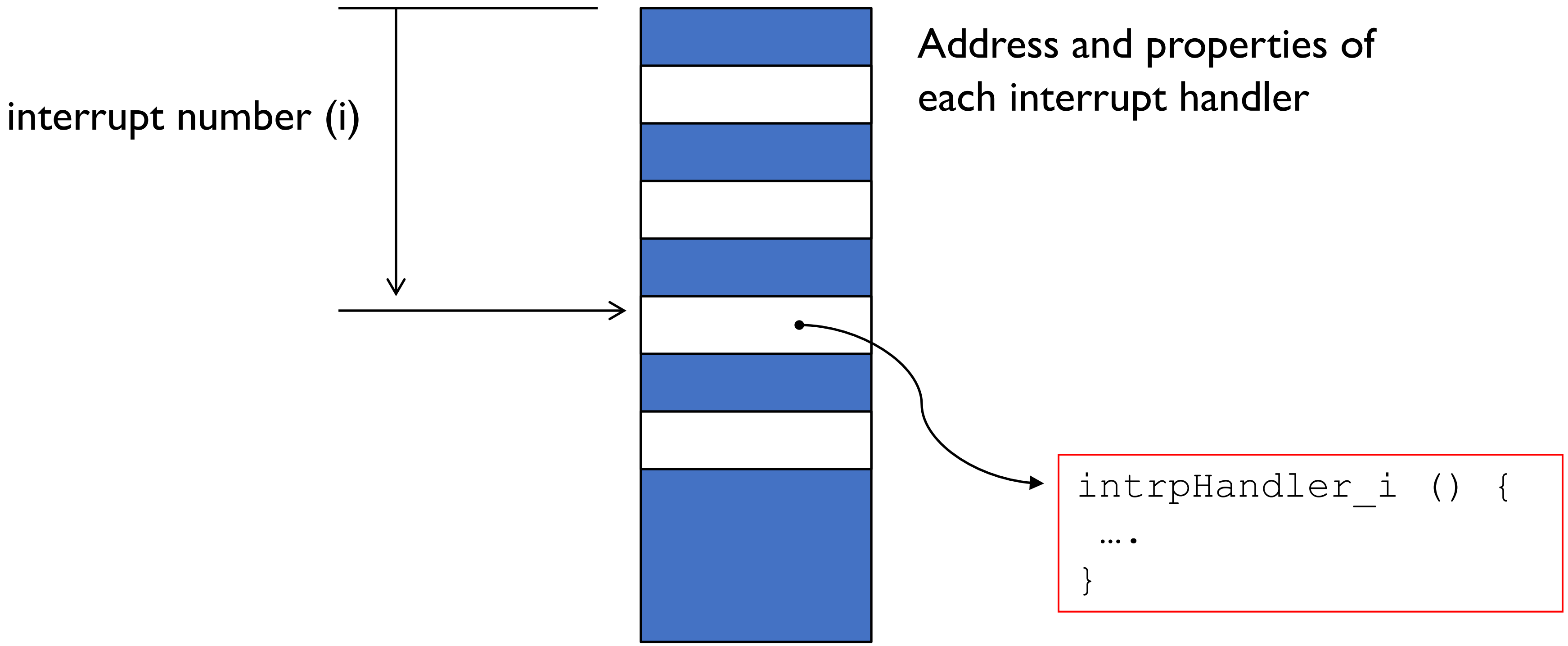
Interrupts (Cont.)

- Most CPUs have two interrupt request lines
  - One is nonmaskable interrupt, for events such as memory errors, power failure
  - The other is maskable interrupt, for it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. This is used by device controllers to request service
- Interrupt vector a table containing memory addresses of interrupt handlers
- Interrupt mechanism is also used for a variety of exceptions
  - This includes arithmetic errors such as dividing by zero, illegal memory access, or attempting to execute a privileged instruction from user mode
- System call executes via trap to trigger kernel to execute request
  - A trap or called a software interrupt, is given a relatively low interrupt priority compared with those assigned to device interrupts
- It implements a system of interrupt priority levels
  - This enables the CPU to defer the handling of low-priority interrupts while allowing a high-priority interrupt to preempt execution of a low-priority interrupt.
- Multi-CPU systems can process interrupts concurrently

Interrupt-Driven I/O Cycle



Interrupt Vector

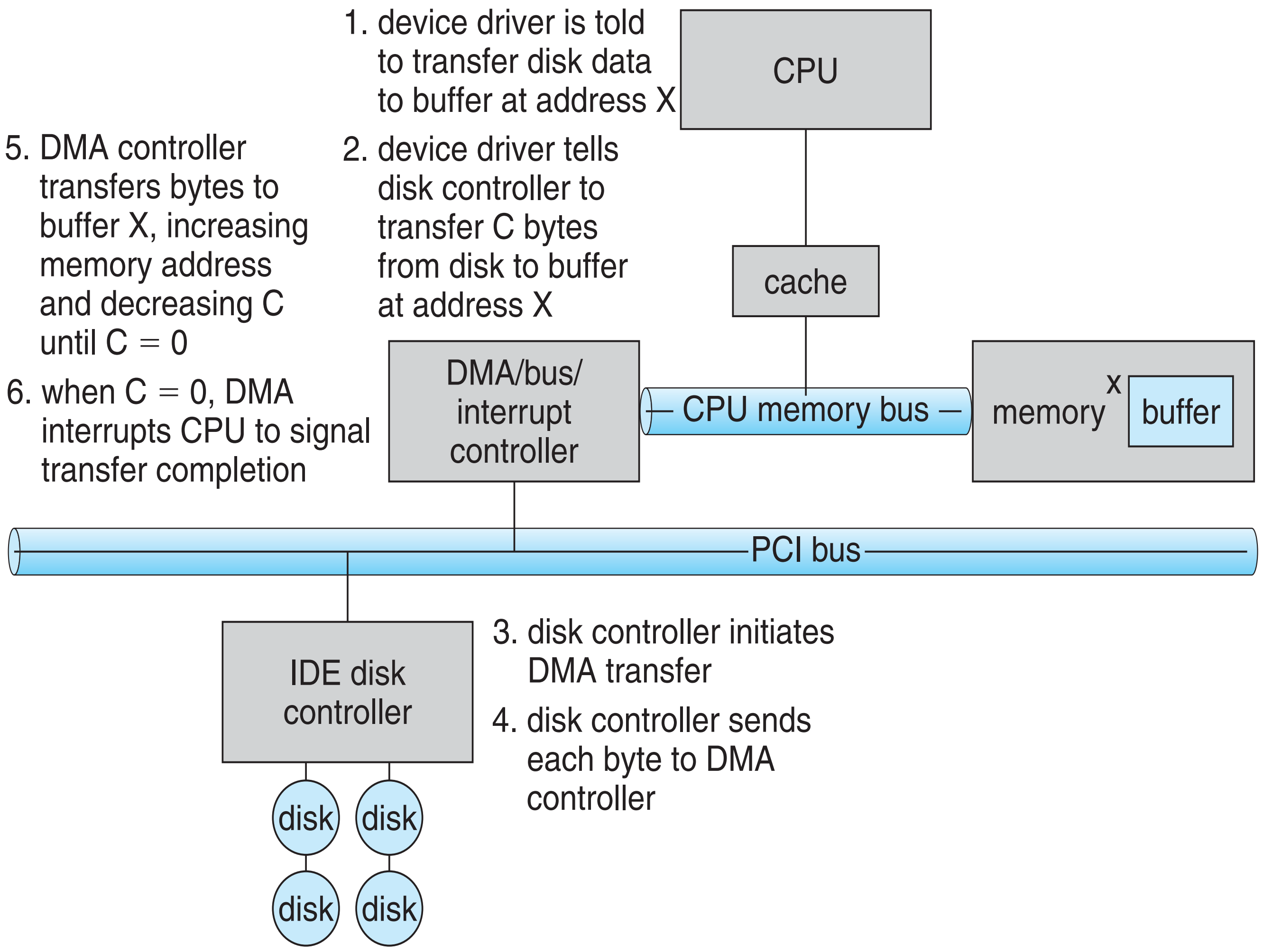


- Interrupt vector is a table containing memory addresses of interrupt handlers

Direct Memory Access

- **Programmed I/O** - use an CPU to monitor device status and to feed data into a controller register one byte at a time (for handling slow devices)
- DMA controller is used to avoid programmed I/O (one byte at a time) for large data movement (e.g., a disk drive) - bypasses CPU to transfer data directly between I/O device and the memory
- OS writes DMA command block into memory
  - Source and destination of the transfer
  - Read or write mode, a count of the number of bytes to be transferred
  - Writes location of command block to DMA controller
  - The DMA controller proceeds to operate the memory bus directly - momentarily prevent CPU from accessing main memory
  - When done, interrupts to signal completion
- DMA is standard in all modern computers, from smartphones to mainframes

Six Step Process to Perform DMA Transfer



System Call and API

- System Calls are programming interfaces to the services provided by the OS – not directly accessed by application programs
- Application Program Interface (API) specifies a set of functions that are available to application programmers, including the parameters passed to the function and return values it may expect
- There is a need for separating API and underlying system call:
  - Program portability by using API, in which API can remain the same across different OSes or different platforms
  - To hide the complex details in system calls from users

System Call Types

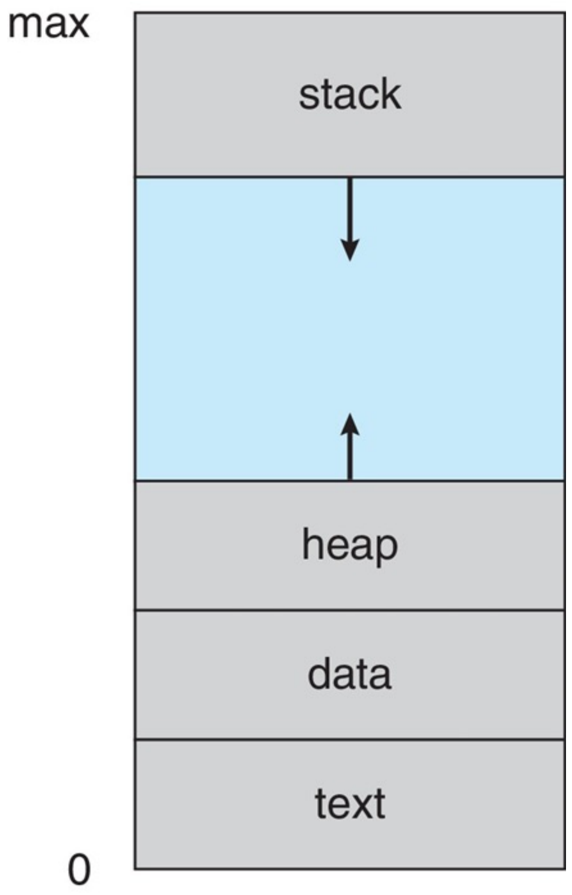
- **Process control:** end, abort, create, terminate, allocate and free memory.
- **File management:** create, open, close, delete, read file etc.
- Device management
- Information maintenance
- Communication

Process

- Process – An OS abstraction of a running program. A process can be described by its state: the contents of memory in its address space, the contents of CPU registers (including the program counter and stack pointer, among others), and information about I/O (such as open files to be read or written).
- The OS provides system calls for operations on processes, typically, including creation, termination, and other useful calls
- A process list – an kernel data structure OS manages to contain information about all processes currently in the system.

Address Space

- Address Space – the address space of a process contains all of the memory state of the running program
- This is the **abstraction** that the OS provides to the running program – this allows CPU to deal with this space address instead of the actual physical memory (to be discussed later)
- In a multiprogram OS, multiple programs reside inside memory, each program can be loaded at some arbitrary physical address(es)
- Another abstraction called virtual memory with virtual address will be introduced
- In a multi-threaded process, threads share code (program), data and I/O (files to be accessed)



Threads within a Process

- If a process has a single thread of execution - a single unique execution context fully describes the state, i.e., the current activity of the thread
- A thread is executing on a processor (CPU) when it is resident in processor (CPU) registers
- Most modern operating systems allow a process to have multiple threads of execution and thus to perform more than one task at a time.
  - A web browser might have one thread display images or text while another thread retrieves data from the network
- Threads within a process share code, data, I/O and files
  - On multicore systems, multiple threads of a process can run in parallel on different CPU cores