

Welcome to Systems Programming!

- ▶ A *Computer System* is the *hardware* and systems software that work in concert to run an application program.
- ▶ In this course, you will learn many practical skills and hopefully obtain a *deep* knowledge of the system that makes you a better programmer.
- ▶ The course has two major components of nearly equal weight:
Lectures/Assignments/Quizzes **C Labs**
56% **44%**
- ▶ You need to work on both to do well in the course. Details in the **Course Outline**.
- ▶ There is ample help available: Lab TAs will be on duty (**TEAMS**) **every** day of the week. Take advantage of it!

What happened in a past term:

Lab 4				Lab 5				Lab 6			
Score	Count	% Sub	% All	Score	Count	% Sub	% All	Score	Count	% Sub	% All
10	51	42.50%	29.48%	10	78	65.00%	45.09%	10	85	68.00%	49.13%
5	29	24.17%	16.76%	6	2	1.67%	1.16%	6	29	23.20%	16.76%
0	40	33.33%	23.12%	4	16	13.33%	9.25%	4	1	0.80%	0.58%
NoSub	53		30.64%	0	24	20.00%	13.87%	3	1	0.80%	0.58%
				NoSub	53		30.64%	0	9	7.20%	5.20%
								NoSub	48		27.75%
TotalSub	120			TotalSub	120			TotalSub	125		
TotalAll	173			TotalAll	173			TotalAll	173		

Administrivia

- ▶ The course text: **Computer Systems: A Programmer's Perspective** 3rd Ed. is the best text I know of, on the subject. I expect you will read the sections we cover **carefully**!
- ▶ The lectures will follow the text closely. We will cover Chapters 1 - 3, 6, 7 & 8 (selected parts).
- ▶ There is a public channel for asking lecture & assignment related questions. Please ask questions in that channel, unless the content of your question is private. Treat the channel as a discussion forum - i.e. if you know how to answer a fellow student, please do so.
- ▶ Assignment submission should be (i) properly organized (ii) legibly written. If a marker cannot read your submission or they need a *scavenger hunt* to put your answer together, the problem will be awarded a zero. After solutions is posted, **a resubmission will not be accepted**. (iii) A Method for re-assessment of assignments will be posted. Please follow that

Administrivia *continued*

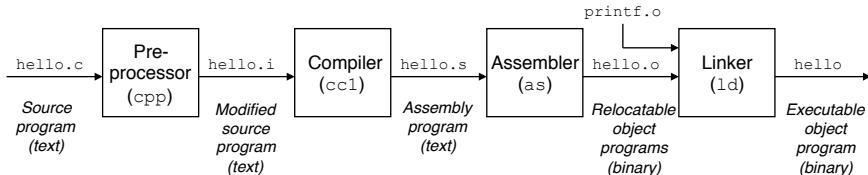
- ▶ Please work on the assignment solutions on your own. Avoid looking for solutions on the web/from friends who have taken the course in past terms *etc etc* .
- ▶ Assignment submission is via Crowdmark
- ▶ **Academic Integrity:**
Past experience shows that there is a great temptation to copy code for the labs and answers for the quizzes. Please understand that we are looking for this. Your submissions will be put through a plagiarism detector. In a past term, 52 students were reported to the AIO, for cheating. 49 of them were found to be **in violation of academic integrity rules**. Last term, in 1120, 43 people were reported to the AIO office.
- ▶ *The Safety Rule:* Discuss the problems with your friends/classmates. **However** the work you turn in **must** be your own. Also remember: Both sides of the copying couplet are considered **equally** guilty.
- ▶ Besides, you will be glad you worked on the course when you get to OS (CS3120)!
- ▶ Now, **ENOUGH OF THIS!** Lets get started

The C compilation Chain

```
#include <stdio.h>

int main(void) {
    printf("Yeah Whatever\n");
    return 0;
}
```

The C compilation chain:



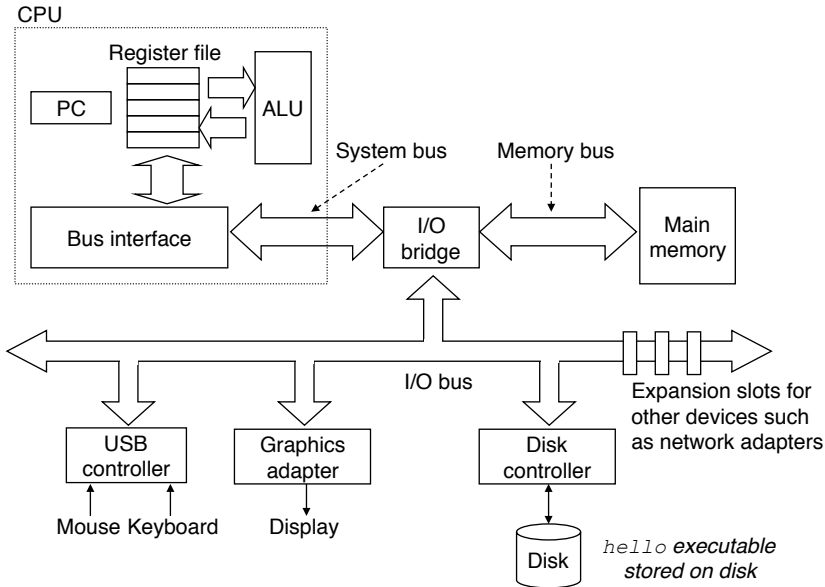
`gcc -Og -S hello.c` generates `hello.s`

`gcc -Og -o hello.o hello.c` generates `hello.o`

`gcc -Og -c hello.c` generates `hello.o`

To run the code `./hello`

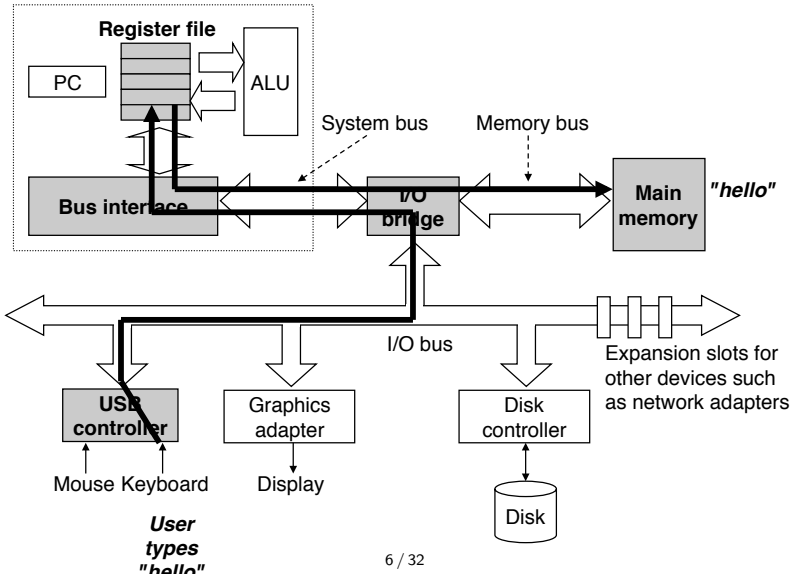
Basic Hardware Organization



Keyboard read - hello.c

Step 1

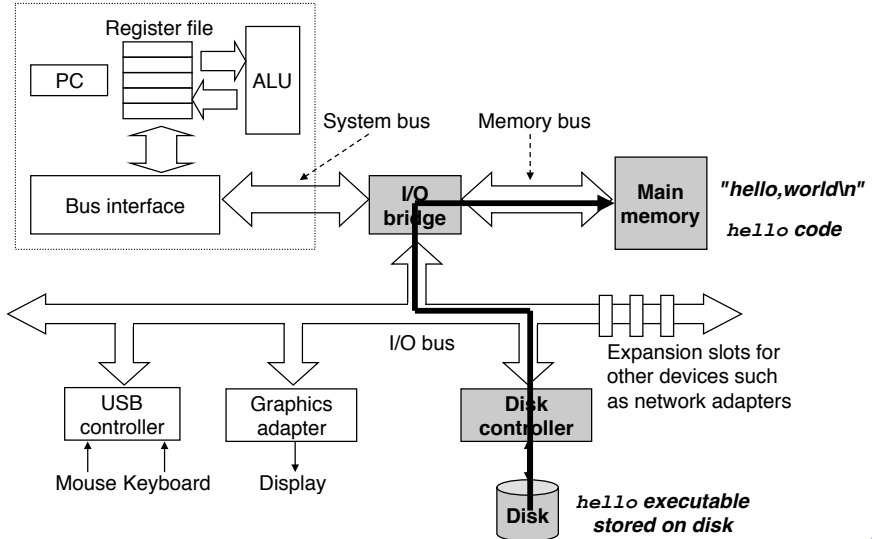
CPU



Load - hello.c

Step 2

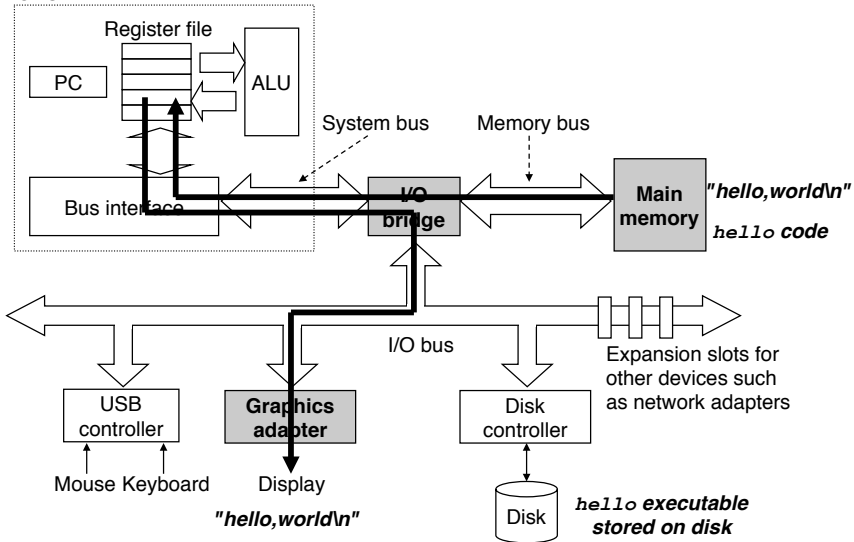
CPU



Execute - hello.c

Step 3

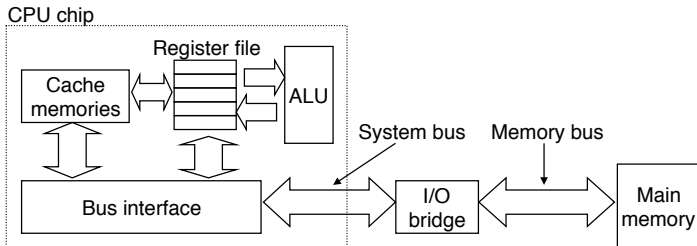
CPU



running hello.c

- ▶ The system spends a lot of time moving information around:
- ▶ `hello.o` stored on the hard disk
→ Loaded into Memory → Instructions copied to processor
Data string ‘‘Yeah...’’ copied → Mem. → display device.
- ▶ Moving stuff around is “**overhead**” to running the program
- ▶ (i) Large storage devices slower than smaller devices
(ii) Faster storage devices more expensive than slower.
- ▶ Typical Hard disk: $\sim 1TB$; RAM: $\sim 1GB$; Reg.File $\sim 0.5KB$
- ▶ Processor reads data from Register File ~ 100 times faster than from RAM.
Processor reads data from RAM $\sim 10^7x$ faster than from Disk.
- ▶ To deal with the Processor-Memory gap: Use smaller faster devices: *Cache Memories* between Processor & Memory.

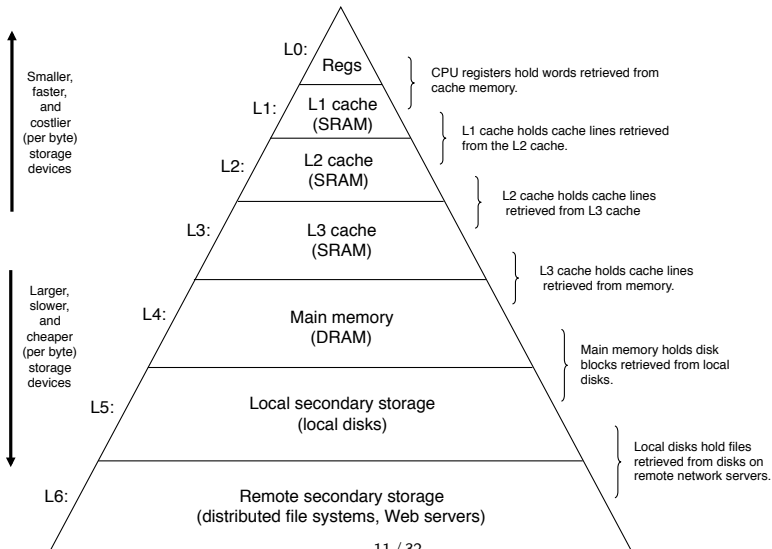
Cache Memories



Instructions that are repeatedly accessed from memory are “cached” closer to the CPU. Think of the instructions in a loop. The speed-up is greater if the code written takes advantage of caches.

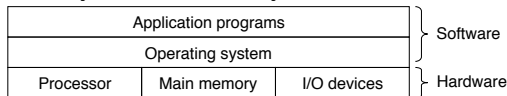
Memories Hierarchy

Inserting smaller, faster storage between Processor and a larger, slower storage is a general idea.

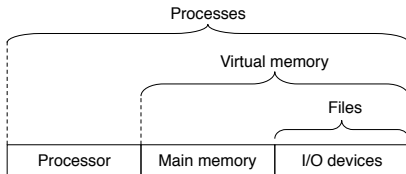


Role of the Operating System in the Execution

The OS manages the hardware. Applications programs do not directly access memory.



OS: (i) Protects hardware from runaway applications
(ii) Provides applications a uniform mechanism to access device that wildly differ in speed.

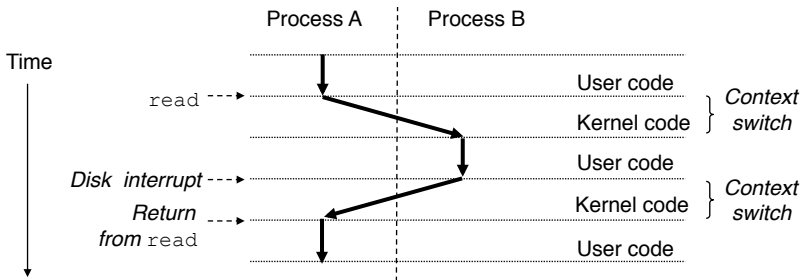


The OS handles this by providing Applications program with abstractions: *Processes*; *Virtual Memory*; & *Files*

Fundamental abstractions provided by the OS

★ **Process:** OS's abstraction of a running program.

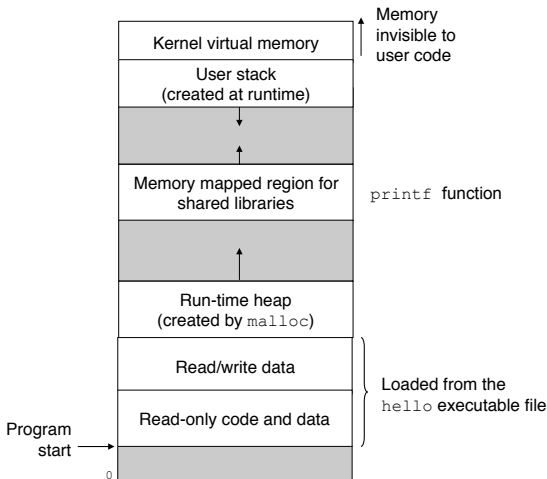
- ▶ Processes can be run concurrently.
- ▶ Traditional system sequential & more processes than processors.
- ▶ Instructions of P_A interleaved with those of P_B .
- ▶ Context Switching



Fundamental abstractions provided by the OS

★ **Virtual Memory** Abstraction that allows each process the illusion that it has exclusive use of memory.

The virtual address space seen by each process has several well defined areas with specific purposes:



Fundamental abstractions provided by the OS

★ **Files:** A *file* is a sequence of bytes. Every I/O device including displays, keyboards and even the network is modelled as a file. Files provide applications a uniform view across many types of I/O devices.

Quantifying performance of the 'system'

Amdahl's law

- α is the fraction of time an enhancement can be used
- 'k' is the speedup due to the enhancement

• So,

- αt_{old} is the original time
- $(\alpha t_{old})/k$ is the new time

• Or

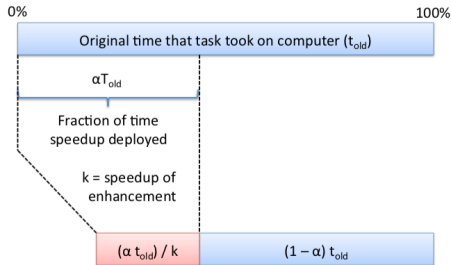
$$t_{new} = (1 - \alpha)t_{old} + (\alpha t_{old})/k$$

- Let 'speedup' (S) = t_{old}/t_{new}

• So,

$$S = \frac{1}{(1 - \alpha) + \alpha/k}$$

This holds for any 'system'.



Amdahl's Law Example:

- ▶ The design goal for a new release of a 'system' is to provide a 2X performance improvement. Adding additional processors, can only enhance 70% of the system. How much would you need to improve this part for the application to reach the design goal? (Find k)

- ▶ If we could speed-up the processing of the 70% of the system, so that the time spent by it is negligible, what will be the overall speed-up?

Bits and Bytes: Review

- ▶ A Computer Stores *everything* as numbers.
- ▶ The only symbols (modern) computers use are '0' and '1'.
A single '0' or '1' is called a *bit* (for *binary digit*).
- ▶ A group of 8 contiguous bits, treated as a unit is called a *Byte*.
A lesser know unit: a group of 4 bits is called a *Nybble*.
- ▶ When the bits are grouped, there are different methods for interpreting them.
- ▶ Each method of interpretation is called a *bit model*.
The bit model used to represent strictly non-negative integers is *unsigned binary*.
- ▶ A study of the bit-models of the different C data types allows a deeper understanding based on the actual memory the bits occupy.

Placeholder representation

We write numbers in a *positional representation* system: $(357)_{10}$ is

$$3 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

A number in base (or *radix*) r : $(d_{n-1}d_{n-2} \dots d_1d_0)_r$ is

$$d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + \dots + d_1 \times r^1 + d_0 \times r^0$$

In base 2 or binary: the *value* of: $(b_{n-1}b_{n-2} \dots b_1b_0)_2$ is:

$$b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_1 \times 2^1 + b_0 \times 2^0$$

So, in (unsigned) binary 101010_2 is:

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

which is equal to 42_{10} .

Conversion Decimal \rightarrow Binary

A simple algorithm allows is to convert decimal \rightarrow binary:

1. Start with two columns: *value* and *bit*, next to each other
2. For the given value, *bit* = 0 if value *even* and *bit* = 1 if *odd*.
3. Update: $\text{value} \leftarrow \lfloor \text{number}/2 \rfloor$
4. If *value* = 1 stop, else goto 1.
5. The binary representation of the given value is now read bottom(MSB) to top (LSB)in the *bit* column.

The process carried out for 123_{10}

$\lfloor v/2 \rfloor$	value (v)	BIT
—	123	1
$\lfloor 123/2 \rfloor$	= 61	1
$\lfloor 61/2 \rfloor$	= 30	0
$\lfloor 30/2 \rfloor$	= 15	1
$\lfloor 15/2 \rfloor$	= 7	1
$\lfloor 7/2 \rfloor$	= 3	1
$\lfloor 3/2 \rfloor$	= 1	1

Reading the *bit* column bottom→top gives: 1111011_2

Let $\vec{x} = [x_{w-1}, \dots, x_0]$ be a vector of w -bits.

$$\text{Ntn: } B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i$$

Q: Is the algorithm *correct*?

Q: Why does it work?

Q: Can you think of another algorithm to do the same?

Note: Practice: $63_{10}, 127_{10}, 255_{10}, 254_{10}, 511_{10}, 1024_{10}$ **Q.** What gen. can you make about how decimal numbers are related to their bit-patterns in binary? Can you use this knowledge to convert 1033_{10} mentally?

A few Examples:

Hexadecimal Representation:

Modern computers have 32 or 64-bit units words. It is difficult to examine groups of 32 or 64 bits. To overcome this, we use base 16.

Representation in base 16 or HEXADECIMAL (commonly: HEX).
Need 16 symbols, identifying "higits" from 0 to 15.

Problem: In decimal, we represent values larger than nine using two or more symbols.

In HEX the symbols 0 - 9 are the same as in decimal.

After that: $10 \rightarrow A$; $11 \rightarrow B$; $12 \rightarrow C$; $13 \rightarrow D$; $14 \rightarrow E$; $15 \rightarrow F$.

HEX values are identified with the prefix 0x. e.g. *0xF00D*

Q: How many bits does it take to represent all possible values between 0 & 15?

Hex	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Conversion Binary \longleftrightarrow Hex

- ▶ To convert *Binary* \rightarrow *Hex*:
Starting from the LSB (right end), create groups of 4 bits.
Pad the front with zeros if necessary:
e.g. 0111 1111 1111 1000
- ▶ Replace each group of 4 with the “higit” from table on the prev. slide: e.g. 0x7 F F 8
- ▶ To go *Hex* \rightarrow *Binary*:
Replace each “higit” with the group of 4 bits from table:
- ▶ e.g. 0xB A D
e.g. 1011 1010 1101
- ▶ **Q:** How do we go *Decimal* \longleftrightarrow *Hex*? **Q:** Written in hex, what is the range of values of a single byte?

Practice

Practice Problem (pp) 2.2 - text, p. 38

- ▶ For $n = 19$ find 2^n in decimal & hex
- ▶ For $2^n = 32$, Find n in decimal & 2^n in Hex
item For $2^n = 0x80$, Find n in decimal & 2^n in Hex
- ▶ For $2^n = 0x10000$, Find n in decimal & 2^n in decimal
- ▶ For $2^n = 16,384$, Find n in decimal & 2^n in Hex
- ▶ **Q(pp2.4):** Without converting, what is: (i) $0x503c + 0x8$ (ii) $0x503c + 0x40$ (iii) $0x50ea - 0x503c$?

Practice

Basic C data types

Every computer has a *word size*. Technically, it is the largest number of contiguous bits that the ALU handles. Practically, it manifests itself as the nominal **size of pointer data** i.e. the the number of bits required for a **virtual address**.

A 32-bit *word* limits virtual address space to *4GB* and A 64-bit *word* to $2^{64} = 2^4 \cdot 2^{60} = 16$ Exabytes!

Signed	Unsigned	Bytes
[signed] char	unsigned char	1
short	unsigned short	2
int	unsigned	4
long	unsigned long	4, 8
int32_t	uint_32	4
int64_t	uint_64	8
char *		8
float		4
double		8

- ▶ These are typically allocated sizes. Not guaranteed.
- ▶ The C standard only guarantees *minimum* sizes. Actual size is implementation dependant.
- ▶ To avoid vagaries of “typical” sizes **ISO C99** defines `int32_t`, `int64_t`. These are guaranteed 4 & 8 bytes. In fact `intN_t` for $N = 8, 16, 32, 64$.
- ▶ **All** pointers are 8 bytes (4 bytes for 32-bit systems).
- ▶ Compilers typically treat `char` as signed. Not guaranteed by C std. Use prefix `signed`.
- ▶ C std. only guarantees lower bounds. Not upper bounds (except `intN_t` types).

printf, scanf

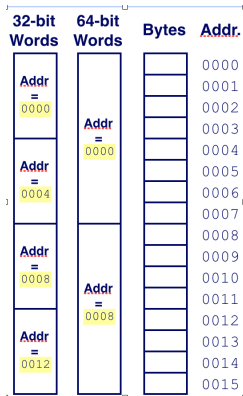
`printf()`: A general purpose output formatting function.

`scanf()`: Reads data from `stdin` and stores them according to the `format` into the location(s) pointed to by additional arguments.

- ▶ Not defined in C. These are part of the standard library of functions (`stdio.h`).
- ▶ 1st arg: string of chars to be printed, each % indicating where one of the arguments substituted.
e.g. `printf("%2d\t %4.2f", dayNum, temperature, cost)`
The *format string* above specifies print:
dayNum is an `int` at least 2 digits wide `%2d`;
temperature as floating point;
cost print as floating point, at least 4 characters wide, 2 after the decimal point.
- ▶ Other format specifiers: `u` unsigned; `x`, `X` (int) hex; `c` single char
`f`, `g`, `G` floating point; `p` pointer; `(void *)`.
- ▶ More on this next class.

Multi-byte Data and Addressing

- ▶ Programs refer to data by address:
Conceptually, a very large array of bytes (not in reality).
- ▶ An address is like an index into that array and, a pointer variable stores an address.
- ▶ System provides private address spaces for each process.
- ▶ Addresses specify byte locations: Address of first byte in word.
Successive word addresses differ by 8 bytes (for 64-bit) and

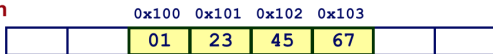


Multi-byte Data and Addressing

Variable x has a 4-byte value: 0x01234567

Address given by &x is 0x100

Big Endian



Little Endian

