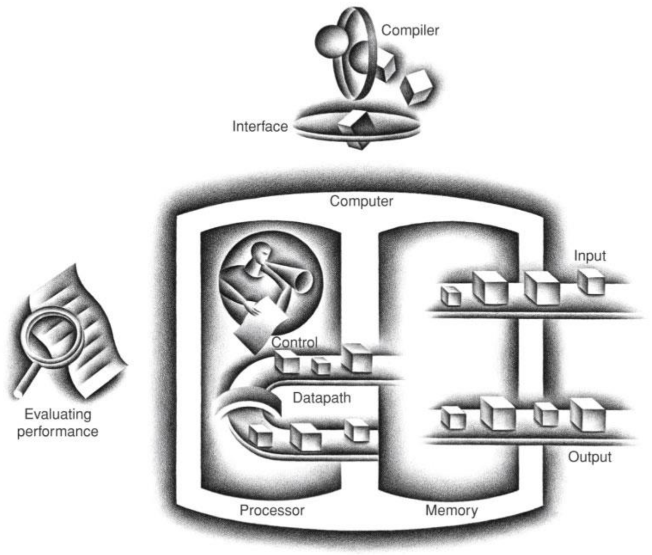
زبان و ساختار کامپیوتر

فصل دوم

کلیات: مروری بر سیستم‌های کامپیوتری



Copyright Notice

Parts (text & figures) of this lecture are adopted from:

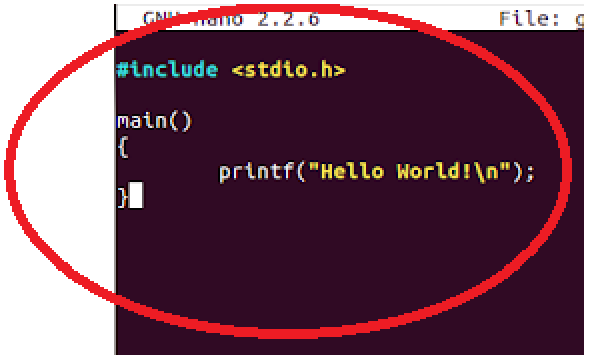
- R E Bryant & D R O'Hallaron, "Computer Systems: A Programmer's Perspective", 3rd Ed, Pearson, 2016
- D Patterson & J Hennessey, "Computer Organization & Design, The Hardware/Software Interface", 5th Ed, MK publishing, 2014
- A Tanenbaum, "Structured Computer Organization", 5th Ed, Pearson, 2006
- M Morris Mano, "Computer System Architecture", Pearson, 1999

Computer Systems

- A computer system consists of hardware & system software that work together to run application programs
  - Specific implementations of systems change over time, but the underlying concepts do not
  - All computer systems have similar hardware and software components that perform similar functions

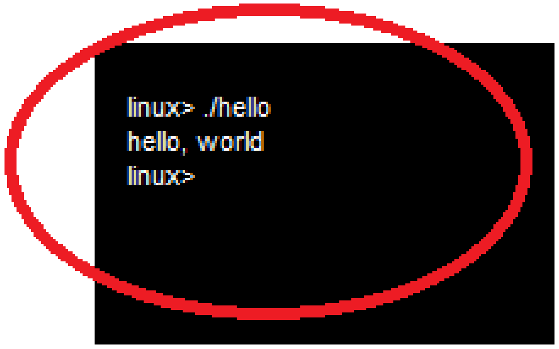
The life-time of a Program

- It is created by a programmer
- Runs on a system
- Prints its simple message
- Terminates



```
GNU nano 2.2.6                File: 
#include <stdio.h>

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

...



```
linux> ./hello
hello, world
linux>
```

Information is Bits + Context

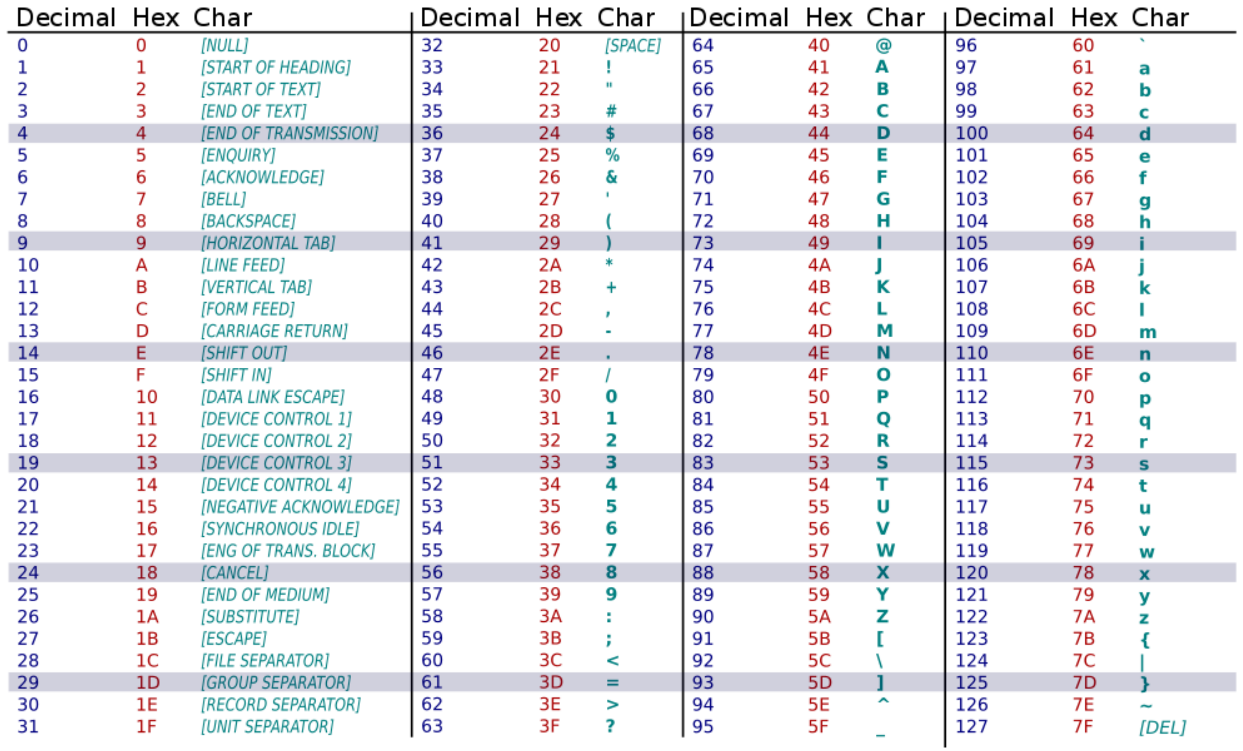
- All information in a system is represented as a bunch of bits
    - disk files
    - programs stored in memory
    - user data stored in memory
    - data transferred across a network
- The only thing that distinguishes data objects is the context
- In different contexts, the same sequence of bytes represent
    - an integer
    - floating-point number
    - character string
    - machine instruction

Context: Source Program

- Our hello program begins life as a source program (source file)
  - created with an editor
  - saved in a text file called hello.c
- The source program is a sequence of bits, organized in 8-bit chunks called bytes
- Each byte has an integer value that corresponds to some character
- Most computer systems represent text characters using the ASCII standard

ASCII Standard

- American Standard Code for Information Interchange
- Represents characters with a unique byte-size integer value



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Hello.C Program

```c
#include <stdio.h>

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

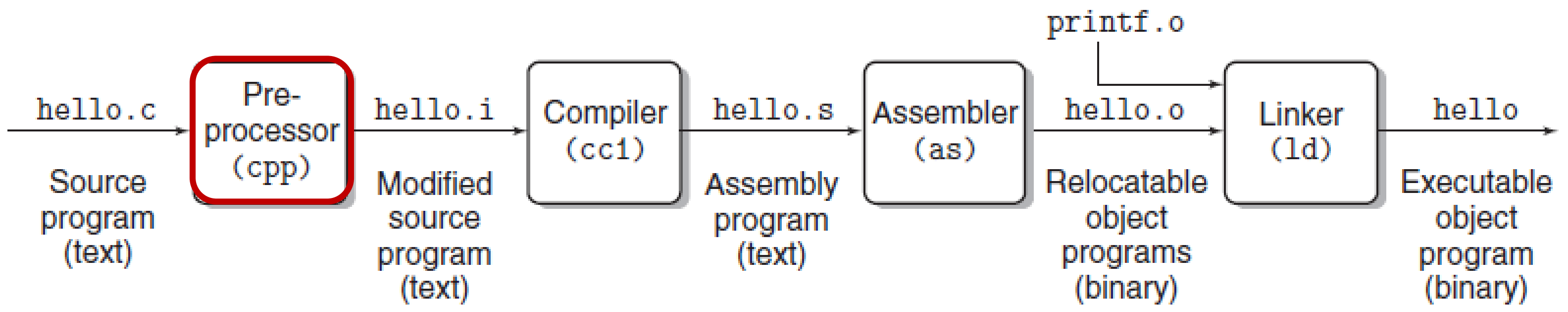
#	i	n	c	l	u	d	e	*SP*	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	*SP*	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	*SP*	*SP*	*SP*	*SP*	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	*SP*	w	o	r	l	d	\	n	"	)	;	\n	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

Different Forms of Programs

- The hello program begins life as a high-level C program
  - it can be read and understood by human beings
- In order to be run on the system
  - It must be translated by other programs into a sequence of low-level machine-language instructions
  - These instructions are then packaged in a form called an executable object program (file) and stored as a binary disk file

The Compilation System

- Preprocessing phase
- Compilation phase
- Assembly phase
- Linking phase

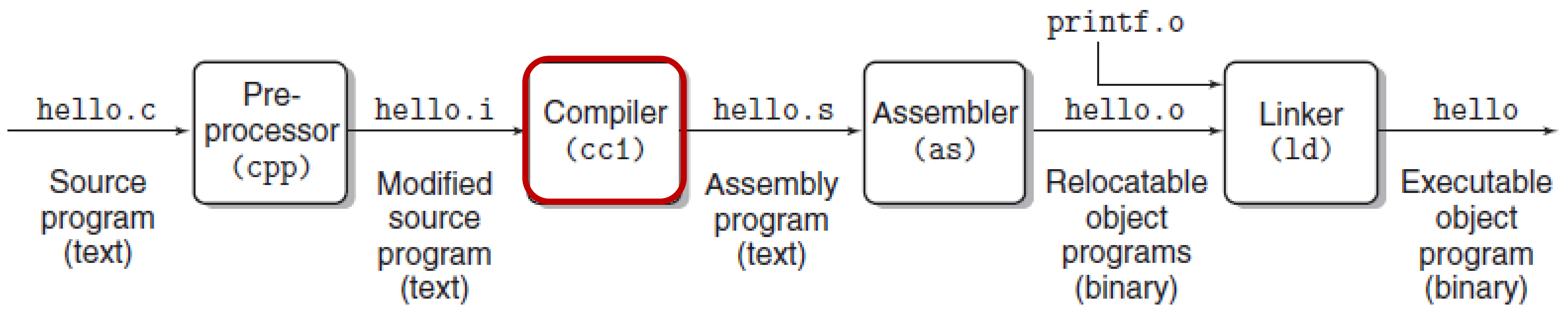


```
#include <stdio.h>

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

```
main:
  subq    $8, %rsp
  movl    $.LC0, %edi
  call    puts
  movl    $0, %eax
  addq    $8, %rsp
  ret
```

The Compilation System

- Preprocessing phase
- Compilation phase
- Assembly phase
- Linking phase

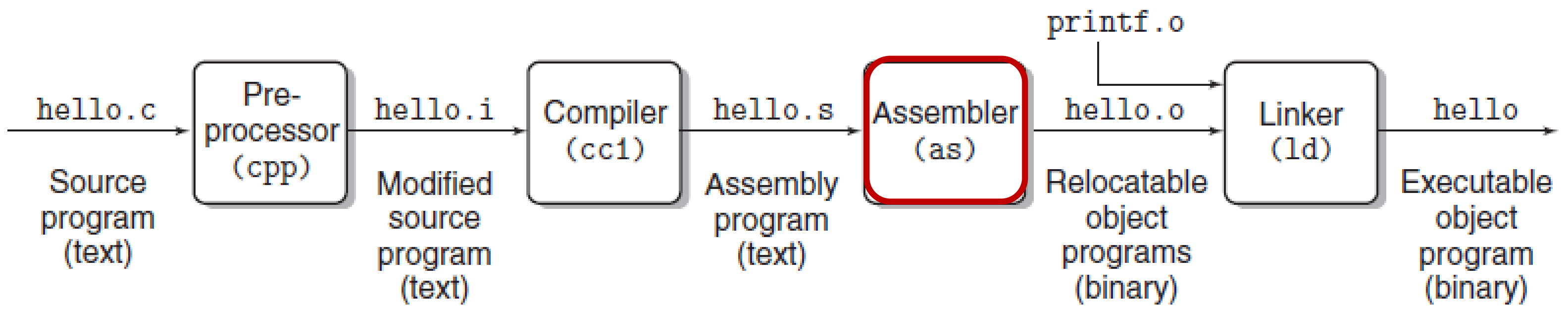


```
#include <stdio.h>

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

```
main:
   subq    $8, %rsp
   movl    $.LC0, %edi
   call    puts
   movl    $0, %eax
   addq    $8, %rsp
   ret
```

The Compilation System

- Preprocessing phase
- Compilation phase
- Assembly phase
- Linking phase



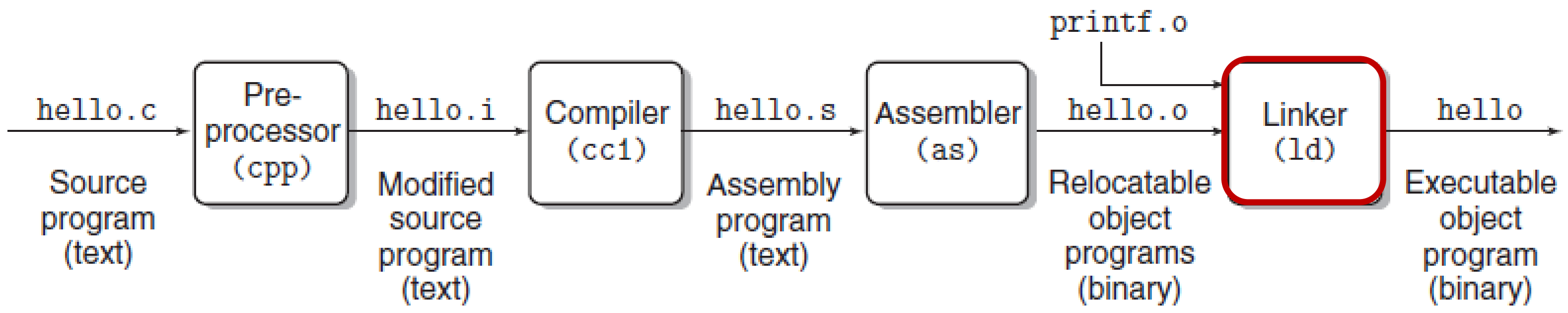
```
#include <stdio.h>

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

```
main:
   subq    $8, %rsp
   movl    $.LC0, %edi
   call    puts
   movl    $0, %eax
   addq    $8, %rsp
   ret
```

ساختار و زبان کامپیوتر

The Compilation System

- Preprocessing phase
- Compilation phase
- Assembly phase
- Linking phase



```
#include <stdio.h>

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

```
main:
  subq    $8, %rsp
  movl    $.LC0, %edi
  call    puts
  movl    $0, %eax
  addq    $8, %rsp
  ret
```

Why Consider?

- Optimizing program performance
- Understanding link-time errors
- Avoiding security holes

Optimizing Program Performance

- Is a switch statement always more efficient than a sequence of if-else statements?
- How much overhead is incurred by a function call?
- Is a while loop more efficient than a for loop?
- Are pointer references more efficient than array indexes?
- Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference?
- How can a function run faster when we simply rearrange the parentheses in an arithmetic expression?

Understanding Link-time Errors

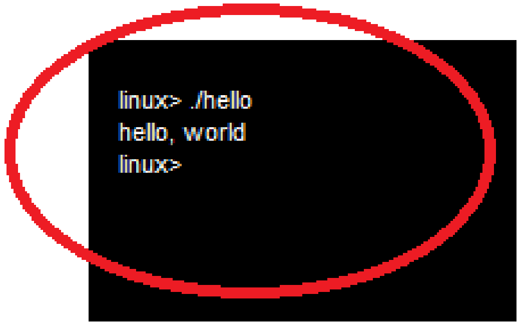
- What does it mean when the linker reports that it cannot resolve a reference?
- What is the difference between a static/global variables?
- What happens if you define two global variables in different C files with the same name?
- What is the difference between a static/dynamic libraries?
- Why does it matter what order we list libraries on the command line?
- Why do some linker-related errors not appear until run time?

Avoiding Security Holes

- First step in learning secure programming is to understand the consequences of the way data and control information are stored on the program stack
  - e.g. avoiding buffer overflows

Program Execution

- To run the executable file on a Unix system, we type its name to an application program known as a shell
- The shell loads and runs the program and waits for it to terminate
- The hello program prints its message to the screen and then terminates
- The shell then prints a prompt and waits for the next input command line



```
linux> ./hello
hello, world
linux>
```

Hardware Organization

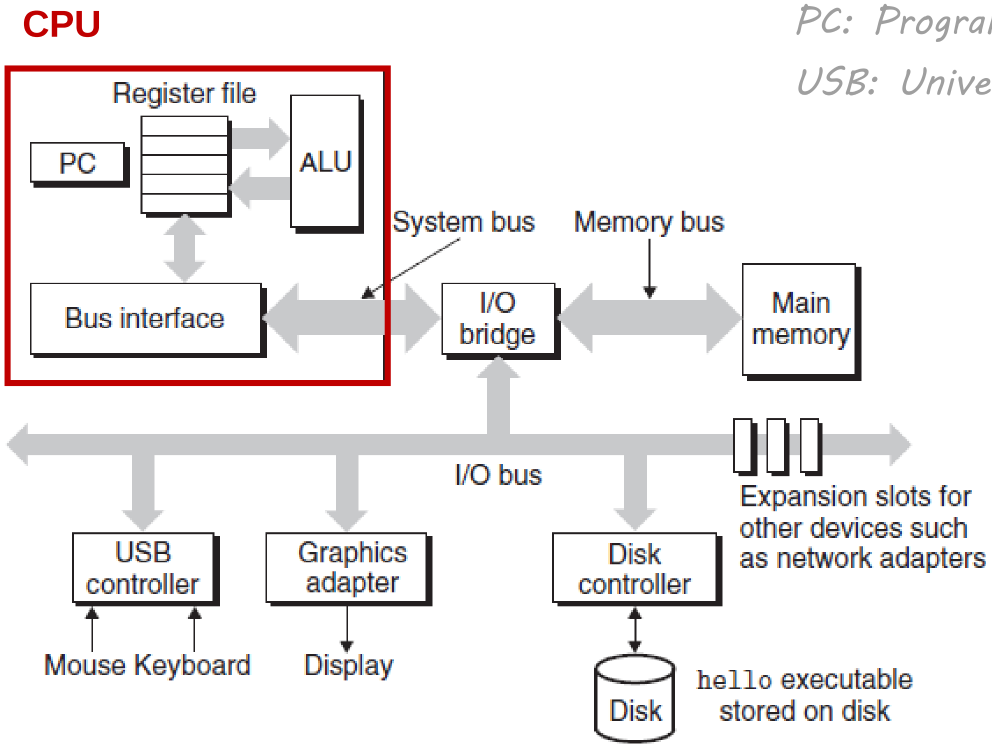
of a typical system

CPU: Central Processing Unit

ALU: Arithmetic/Logic Unit

PC: Program Counter

USB: Universal Serial Bus



Bus

- A collection of electrical conduits
- Carry bytes of information back and forth between the components
- Are typically designed to transfer fixed-size chunks of bytes known as words
  - Word Size: The number of bytes in a word
  - a fundamental system parameter that varies across systems

Input/Output Device

- Connection to the external world, e.g.
  - A keyboard and mouse for user input
  - A display for user output
  - A disk drive for long-term storage of data and programs

I/O Controller/Adapter

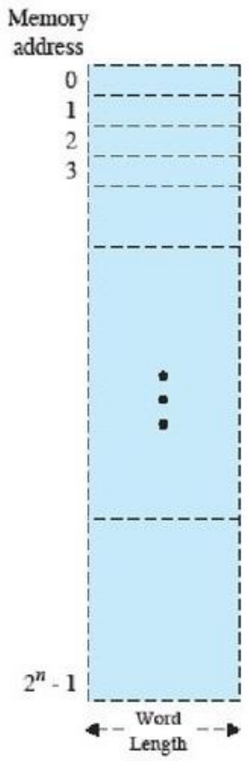
- Each I/O device is connected to the I/O bus by either a controller or an adapter
  - Controllers are chipsets in the device or on the system's motherboard
  - An adapter is a card that plugs into a slot on the motherboard
  - Purpose of each is to transfer information between the I/O bus and an I/O device

Main Memory - 1

- A temporary storage device that holds a program and the data it manipulates
  - Each machine instruction can consist of a variable number of bytes
  - The sizes of data items (program variables) vary according to type

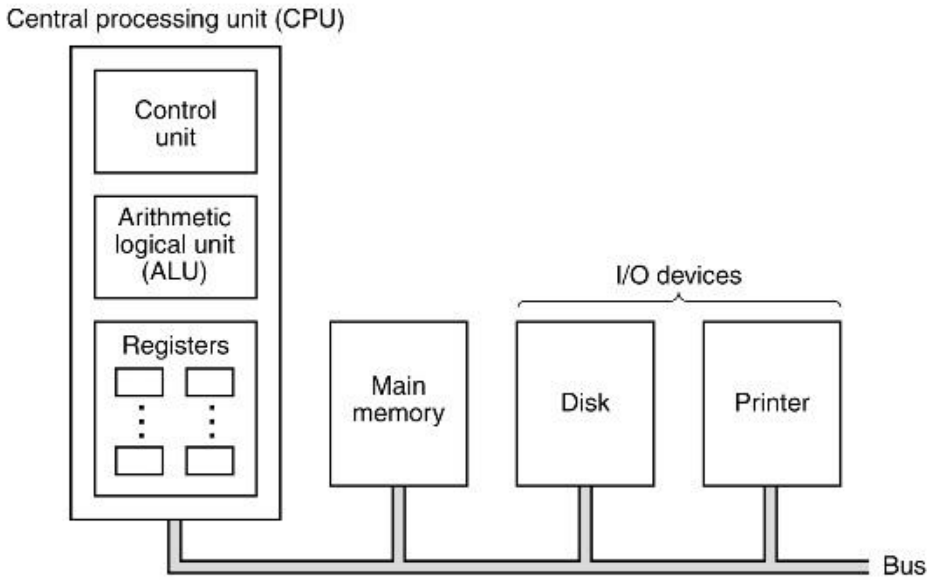
Main Memory - 2

- Physically, main memory is a collection of dynamic random access memory chips
- Logically, memory is organized as a linear array of bytes, each with a unique address

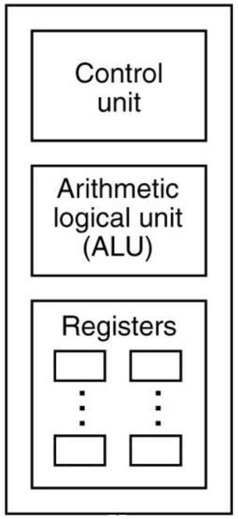


Central Processing Unit (CPU)

- The engine that interprets (executes) instructions stored in main memory



CPU Components



- Control Unit: responsible for
  - fetching instructions from memory
  - determining their type
- Arithmetic Logic Unit (ALU): perform operations
  - Addition, Boolean AND, ...
- Registers: small, high-speed memory used to store
  - temporary results
  - certain control information

Registers

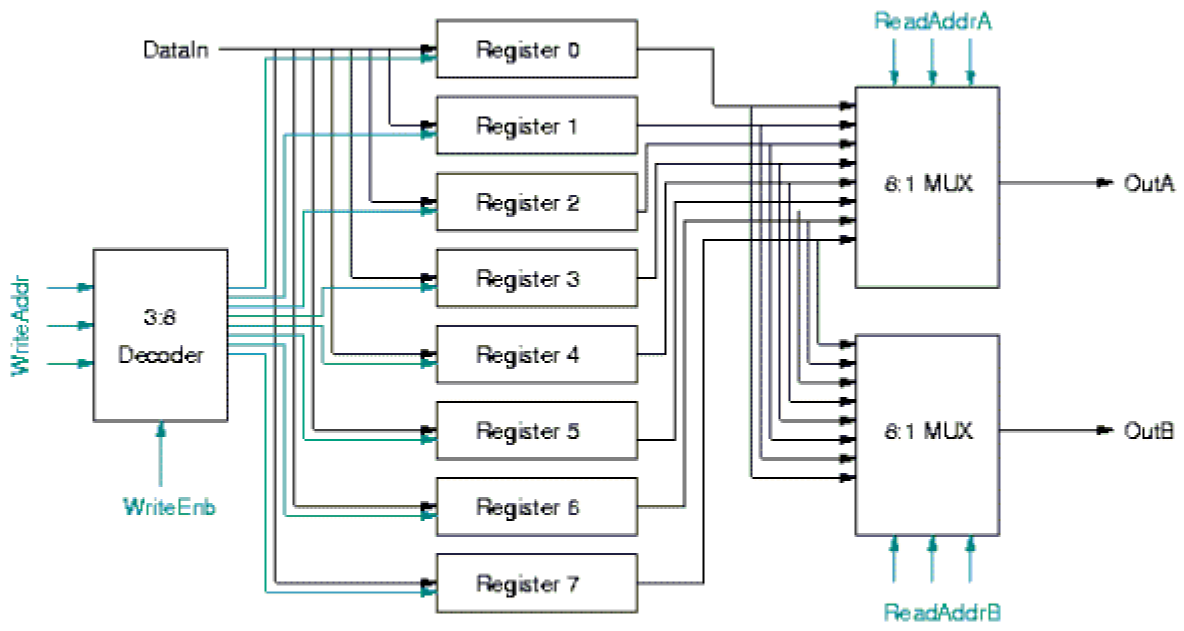
- Small, high-speed memory used to store
  - temporary results
  - certain control information
- Each having a certain size and function
- Can be read and written at high speed since they are internal to the CPU

Registers

- Program Counter (PC): points to the next instruction to be fetched
- Instruction Register (IR): holds the instruction currently being executed
- Other general/special purpose registers
  - Accumulator
  - Index/Base Registers
  - Register files
  - …

Register File

- A small storage device consisting of a collection of word-size registers, each with its own unique name



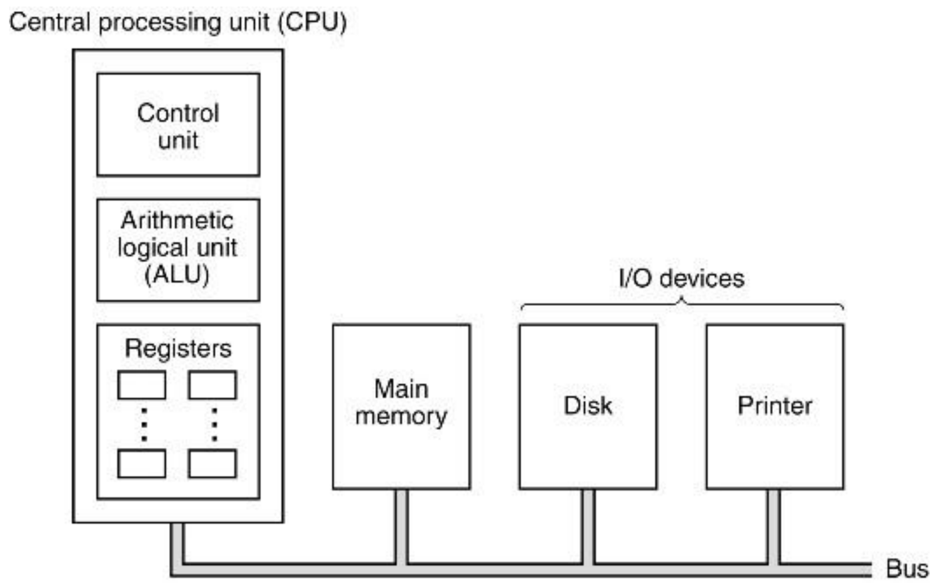
Gray lines are 1-bit signals
Black lines are 10-bit signals

Program Counter (PC)

- Points at (contains the address of) some machine-language instruction in main memory
- Also called IP (Instruction Pointer)
- From the time that power is applied to the system until the time that the power is shut off, a processor repeatedly
  - executes the instruction pointed at by the PC, and
  - updates PC to point to the next instruction

Program Execution by CPU

- CPU executes programs stored in the main memory by:
  - fetching the instructions,
  - examining them,
  - and then executing them one after another

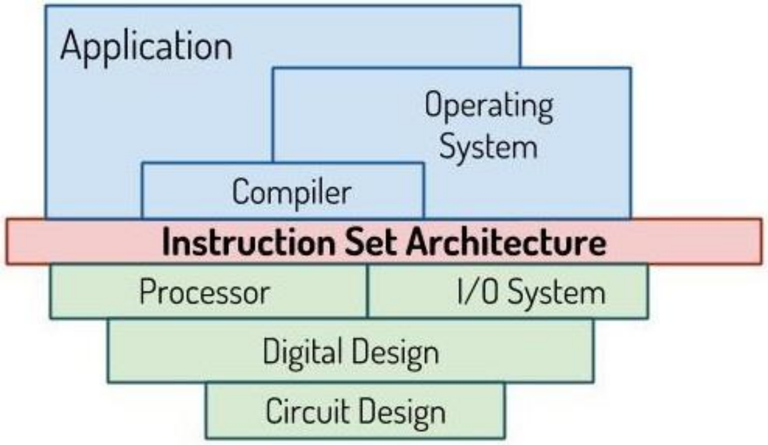


Instruction Execution Steps

- Fetch - Decode - Execute Cycle:
  1. Fetch next instruction from memory into IR
  2. Change PC to point to next instruction
  3. Determine type of instruction just fetched
  4. If instructions uses word in memory, determine where it is
  5. Fetch the word, if needed, into a CPU register
  6. Execute the instruction
  7. Go to step 1 to begin executing the next instruction

Instruction Set Architecture

- A processor appears to operate according to a very simple instruction execution model, defined by its instruction set architecture (ISA)
  - In fact modern processors use far more complex mechanisms to speed up program execution



Simple Operation Examples

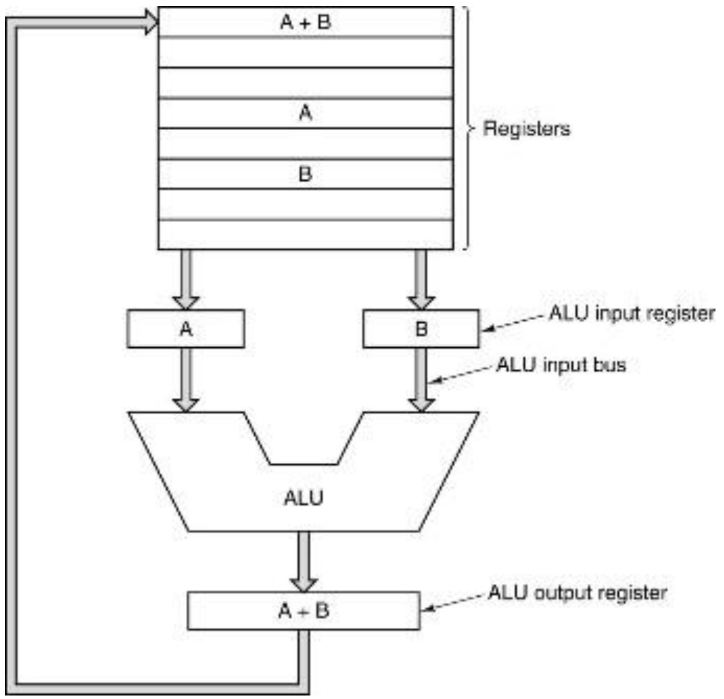
- Load:
  - copy a byte or a word from main memory into a register
- Store:
  - copy a byte or a word from a register to a location in main memory
- Operate:
  - copy the contents of two registers to the ALU
  - perform an arithmetic operation on the two words
  - store the result in a register
- Jump:
  - extract a word from the instruction itself
  - copy that word into the program counter (PC)

Instructions

- Register-memory instructions allow memory words
  - to be fetched into registers
  - be used as ALU inputs in subsequent instructions
- A typical register-register instruction
  - fetches two operands from the registers,
  - brings them to the ALU input registers,
  - performs some operation on them (e.g. addition or Boolean AND),
  - stores the result back in one of the registers

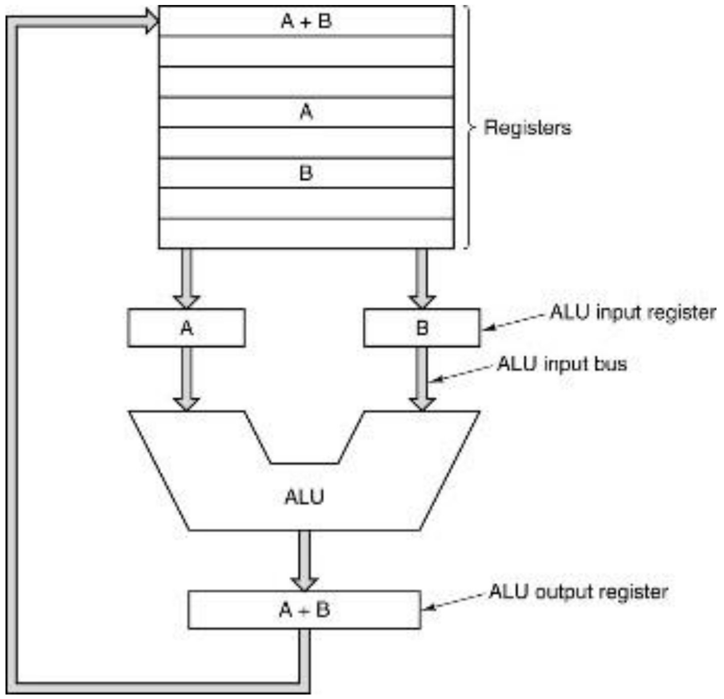
Data Path Cycle

- The process of running two operands through the ALU and storing the result
  - the heart of most CPUs
  - defines what the machine can do



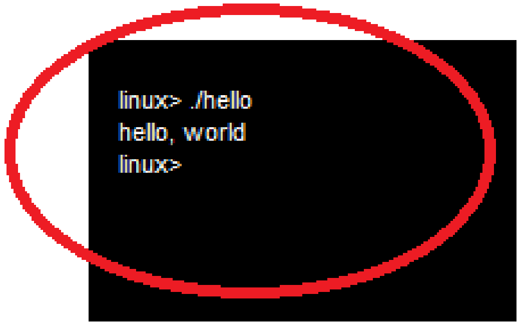
Data Path Cycle

- The faster the data path cycle is, the faster the machine runs
- Modern computers have
  - multiple ALUs operating and/or
  - specialized ALUs for different functions



Back to Hello Program

- What happens when we run our example program?
- The shell loads and runs the program and waits for it to terminate
- The hello program prints its message to the screen and then terminates
- The shell then prints a prompt and waits for the next input command line

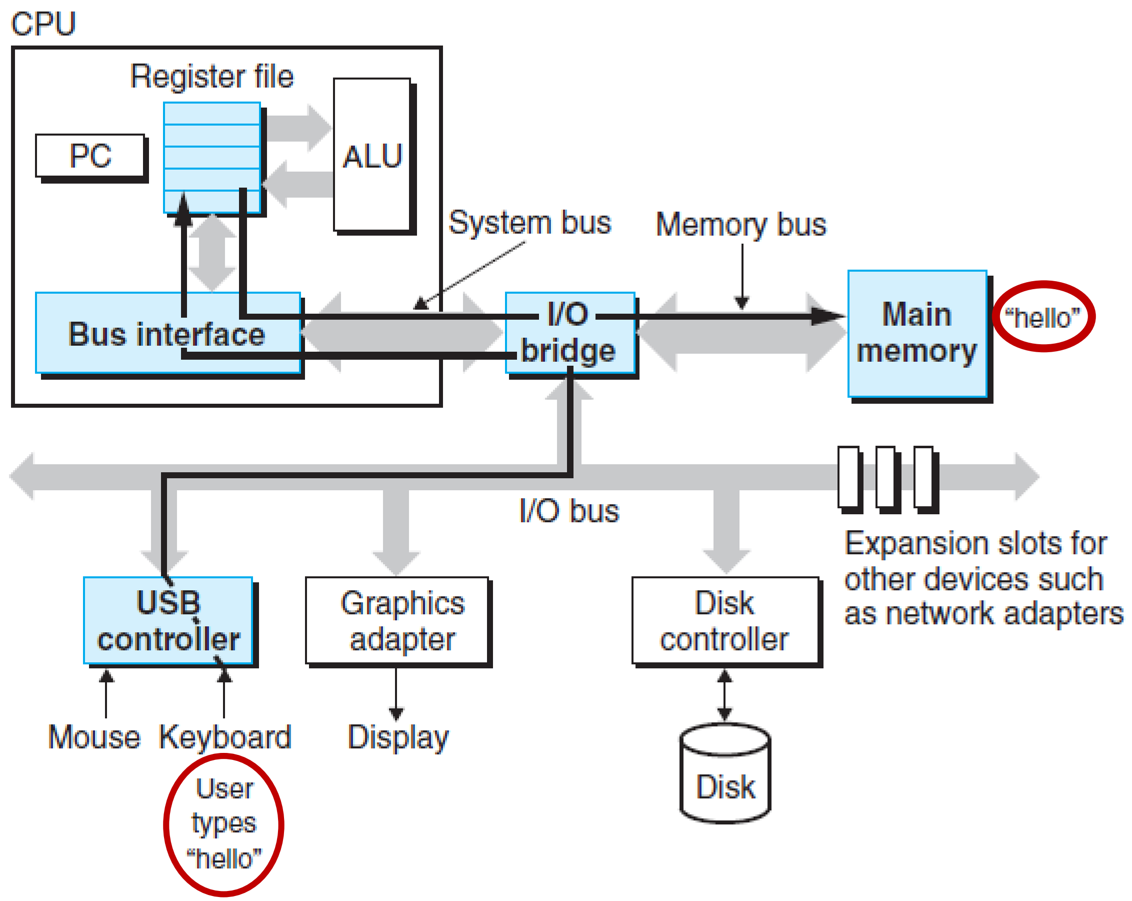


```
linux> ./hello
hello, world
linux>
```

Reading the hello Command

- Initially, the shell program is executing its instructions, waiting for us to type a command
- As we type the characters ./hello at the keyboard, the shell program reads each one into a register and then stores it in memory

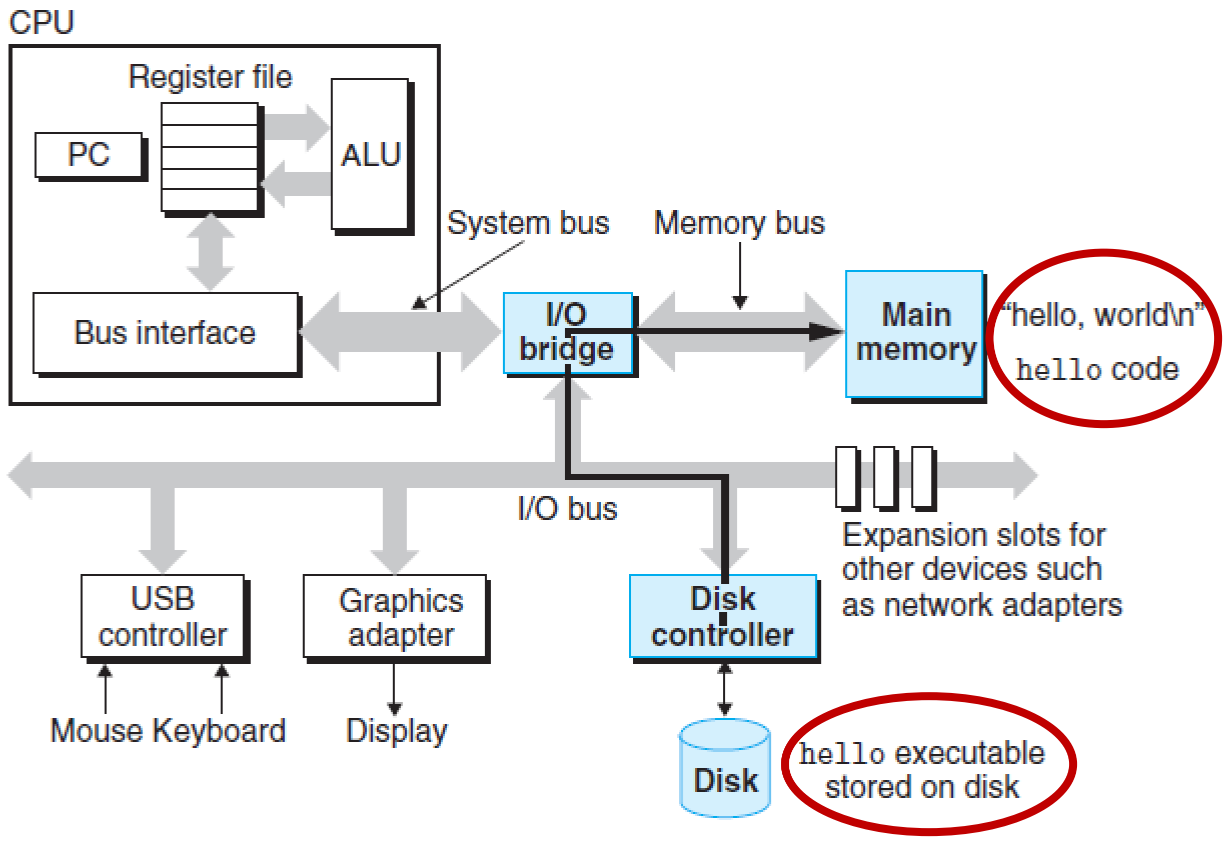
Reading the hello Command



Loading the Executable

- When we hit the enter key on the keyboard, the shell knows that we have finished typing the command
- The shell loads the executable hello file by executing a sequence of instructions that copies the code and data in the hello object file from disk to main memory
  - The data includes the string of characters hello, world\n that will eventually be printed out.
  - Using a technique known as direct memory access (DMA), the data travel directly from disk to main memory, without passing through the processor

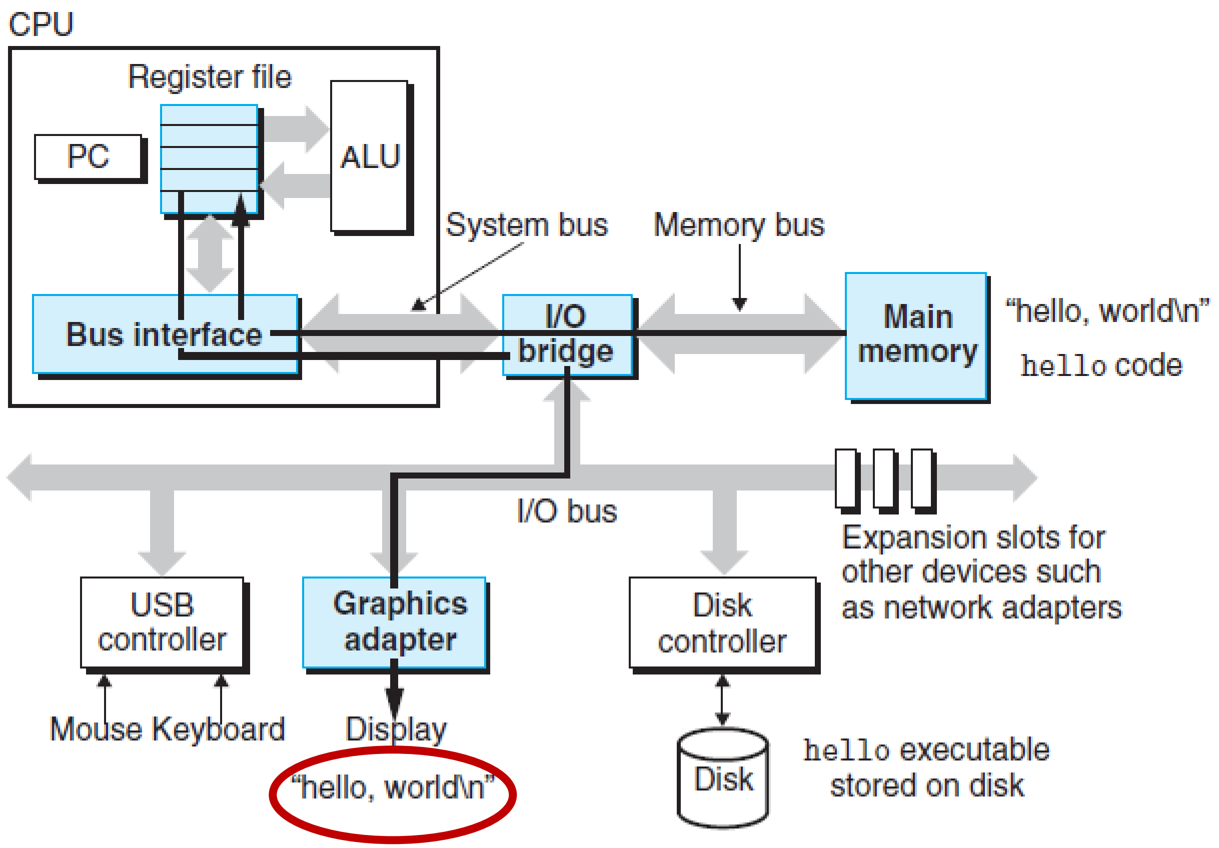
Loading the Executable



Writing the Output String

- Once the code and data in the hello object file are loaded into memory, the processor begins executing the machine-language instructions in the hello program's main routine
- These instructions copy the bytes in the hello, world\n string from memory to the register file, and from there to the display device, where they are displayed on the screen

Writing the Output String



Caches Matter - 1

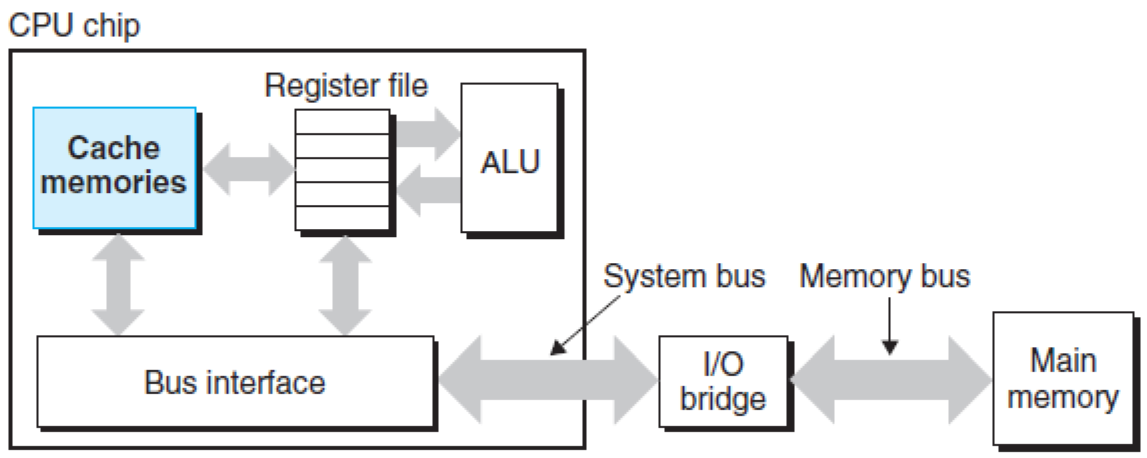
- A system spends a lot of time moving information from one place to another
  - From disk to memory
  - From memory to processor
  - From memory to display device
- Much of this copying is overhead
  - slows down the "real work"

Caches Matter - 2

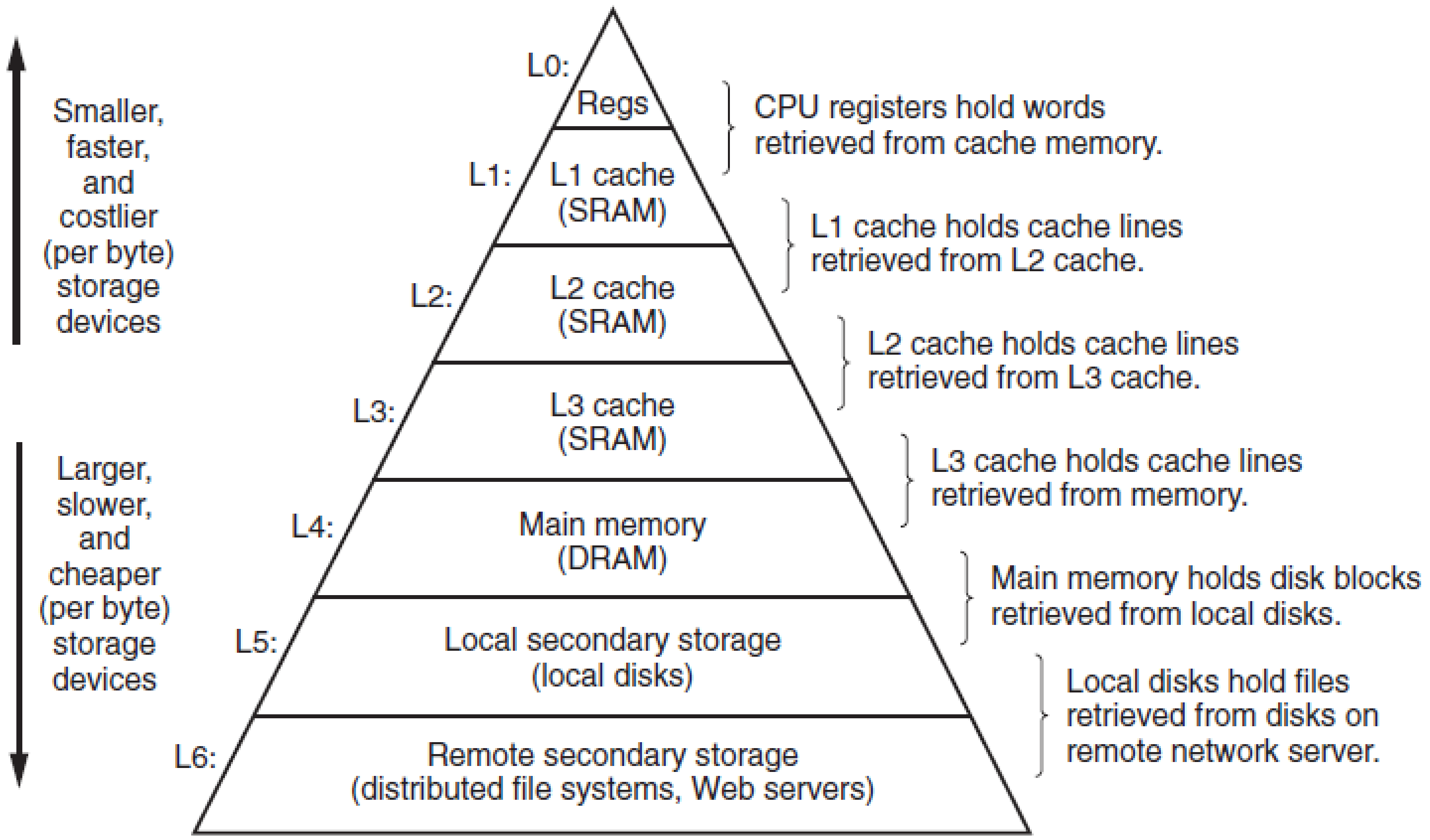
- Large storage devices are slower than smaller ones
- Faster devices are more expensive to build
- Processor-memory gap:
  - It is easier and cheaper to make processors run faster than it is to make main memory run faster
- We can get the effect of both a very large memory and a very fast one by exploiting locality
  - the tendency for programs to access data and code in localized regions

Caches Matter - 3

- Cache memories are smaller, faster storage devices that serve as temporary staging areas for information that the processor is likely to need in the near future

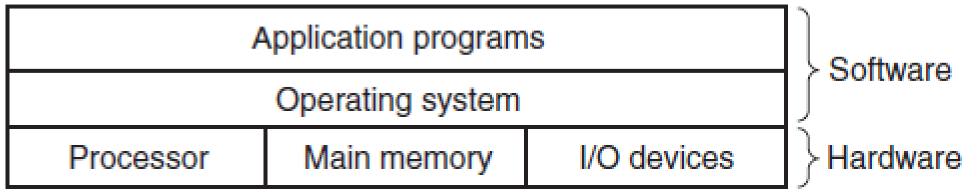


Memory Hierarchy


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: Regs
L1: L1 cache (SRAM)
L2: L2 cache (SRAM)
L3: L3 cache (SRAM)
L4: Main memory (DRAM)
L5: Local secondary storage (local disks)
L6: Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network server.

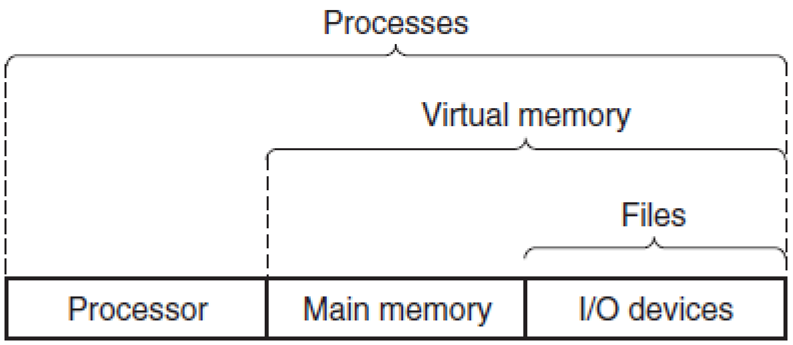
Operating System (OS)

- A layer of software interposed between the application program and the hardware
  - Protects the hardware from misuse by runaway applications
  - Provides applications with simple and uniform mechanisms for manipulating complicated and wildly different low-level hardware devices



Operating System Abstractions

- Files:
  - Abstractions for I/O devices
- Virtual memory:
  - Abstraction for both the main memory and disk I/O devices
- Processes:
  - Abstractions for processor, main memory, and I/O devices

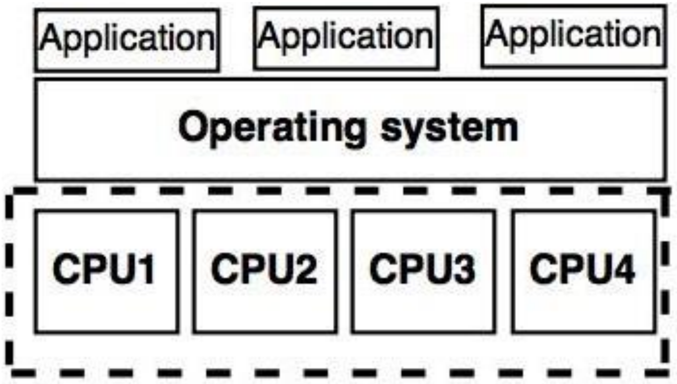


Processes - 1

- The operating system's abstraction for a running program
- Multiple processes can run concurrently on the same system
  - the instructions of one process are interleaved with the instructions of another process
- Each process appears to have exclusive use of the hardware
- In most systems, there are more processes to run than there are CPUs to run them

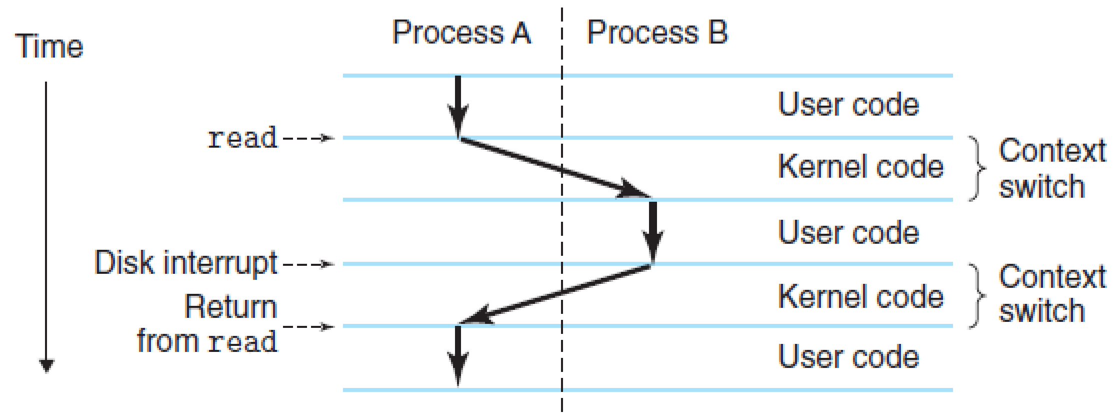
Processes - 2

- Traditional systems execute one program at a time
- Newer multicore processors can execute several programs simultaneously
- A single CPU/core can appear to execute multiple processes concurrently by switching among them



Context Switch

- Saving the context of current process
- Restoring the context of new process
- Passing control to the new process

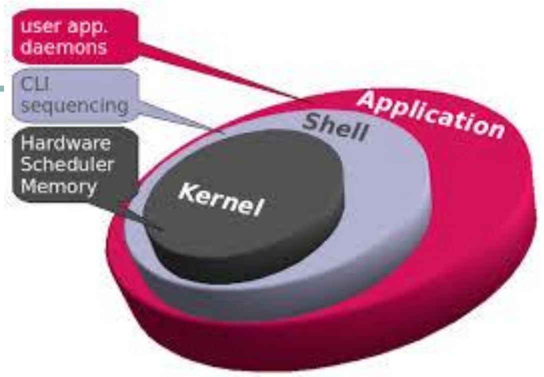


Hello Program Execution

- Two concurrent processes: <u>shell process</u>, <u>hello process</u>
- The shell process is running, waiting for input on the command line
- When asked to run the hello program, the shell invokes a system call that passes control to the operating system
- The operating system saves the shell's context, creates a new hello process and its context, and then passes control to the new hello process
- After hello terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command-line input
- The transition from one process to another is managed by the operating system kernel

Context Switch

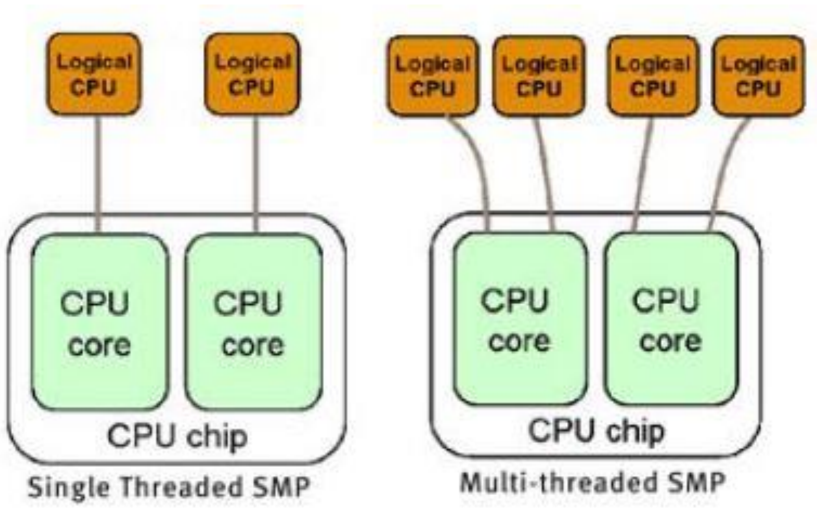
Kernel



- The portion of the operating system code that is always resident in memory
- When an application program requires some action by the operating system (e.g. file read/write) it executes a system call instruction, transferring control to the kernel
- Note that the kernel is not a separate process, but a collection of code and data structures that the system uses to manage all the processes

Threads - 1

- In modern systems a process can consist of multiple execution units, called threads
  - each running in the context of the process
  - sharing the same code and global data



Threads - 2

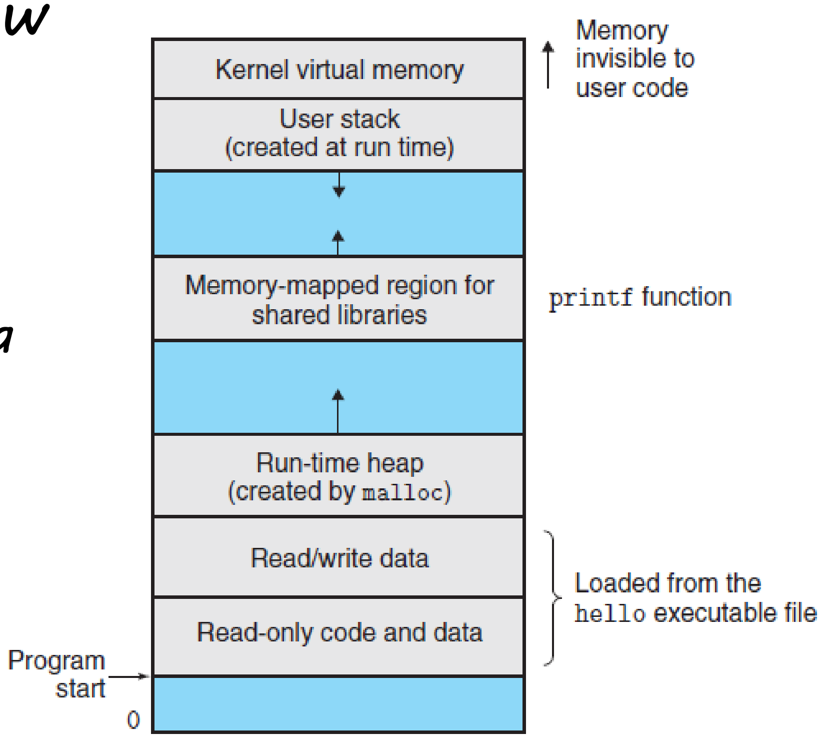
- Why important?
  - the requirement for concurrency in network servers
  - easier to share data between multiple threads than between multiple processes
  - typically more efficient than processes
  - one way to make programs run faster when multiple processors are available

Virtual Memory

- An abstraction that provides each process with the illusion that it has exclusive use of the main memory
- For virtual memory to work, an interaction is required between the hardware and the operating system, including a hardware translation of every address
- Basic idea:
  - Store the contents of a process' virtual memory on disk
  - Use the main memory as a cache for the disk

Virtual Address Space

- The same uniform view of memory, for each process
  - Program code and data
  - Heap
  - Shared libraries
  - Stack
  - Kernel virtual memory


Kernel virtual memory
Memory invisible to user code
User stack (created at run time)
Memory-mapped region for shared libraries
printf function
Run-time heap (created by malloc)
Read/write data
Read-only code and data
Loaded from the hello executable file
Program start
0

Files - 1

- A file is a collection of related information defined by its creator
  - Text Files: consist exclusively of ASCII characters
  - Binary Files: all other files
- Every I/O device is modeled as a file:
  - disks
  - keyboards
  - displays
  - even networks

Files - 2

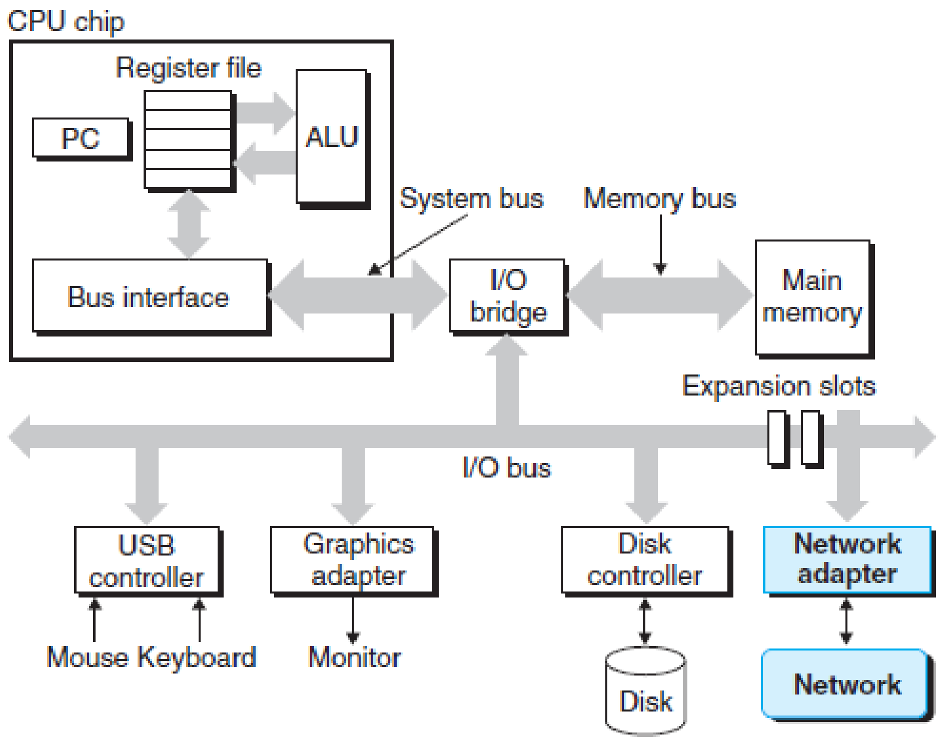
- All input and output is performed by reading and writing files using a small set of system calls
- Files provide a uniform view of all the varied I/O devices, e.g.
  - Application programmers who manipulate the contents of a disk file are unaware of the specific disk technology
  - The same program will run on different systems that use different disk technologies

Networks - 1

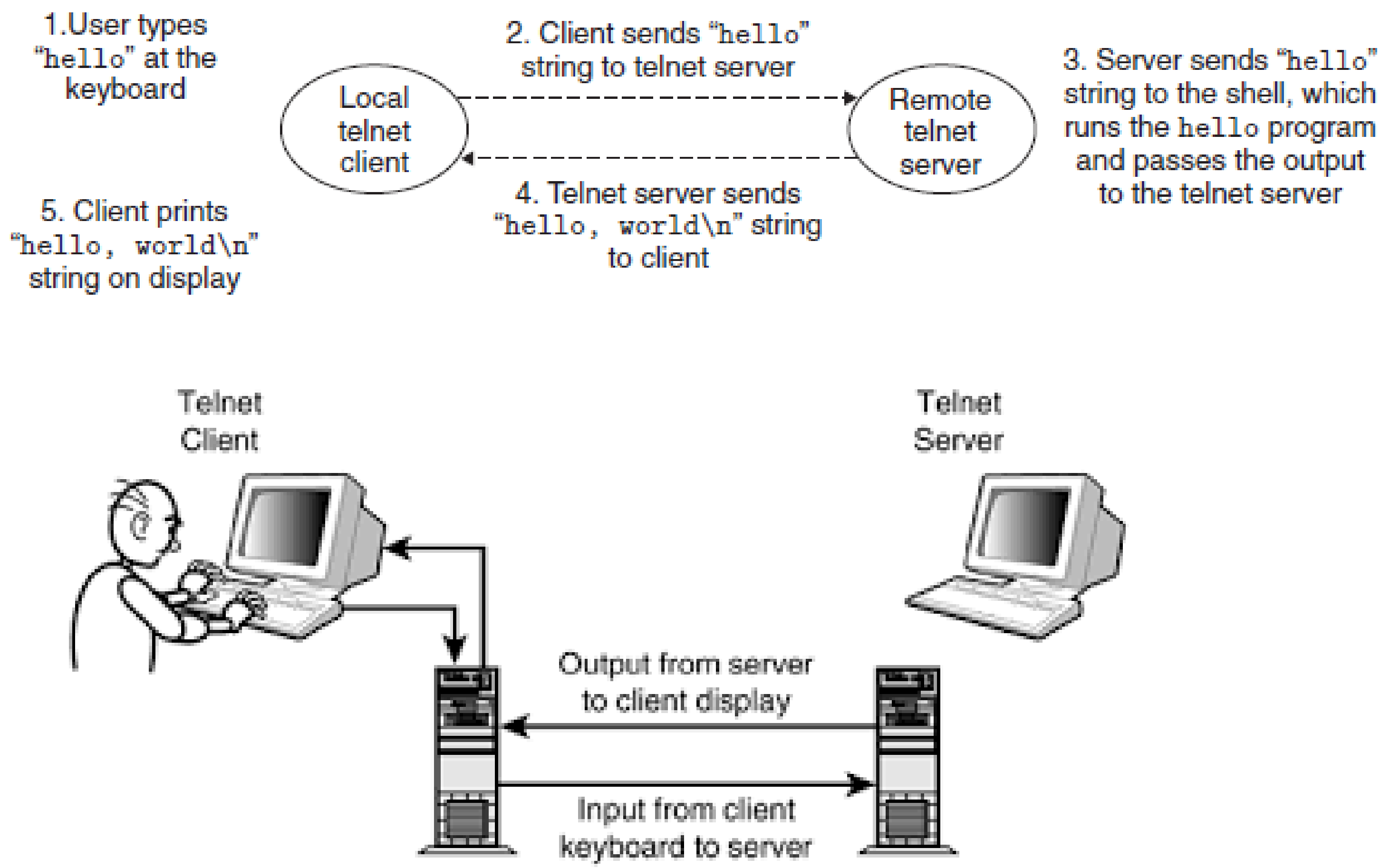
- Modern systems are linked to other systems by networks
- Network applications:
  - email
  - instant messaging
  - the WorldWideWeb
  - FTP
  - telnet
- All based on the ability to copy information over a network

Networks - 2

- Can be viewed as another I/O device



Run hello on a Remote Machine


1.User types "hello" at the keyboard
2. Client sends "hello" string to telnet server
3. Server sends "hello" string to the shell, which runs the hello program and passes the output to the telnet server
4. Telnet server sends "hello, world\n" string to client
5. Client prints "hello, world\n" string on display
Local telnet client
Remote telnet server
Telnet Client
Telnet Server
Output from server to client display
Input from client keyboard to server

Stored Program Concept

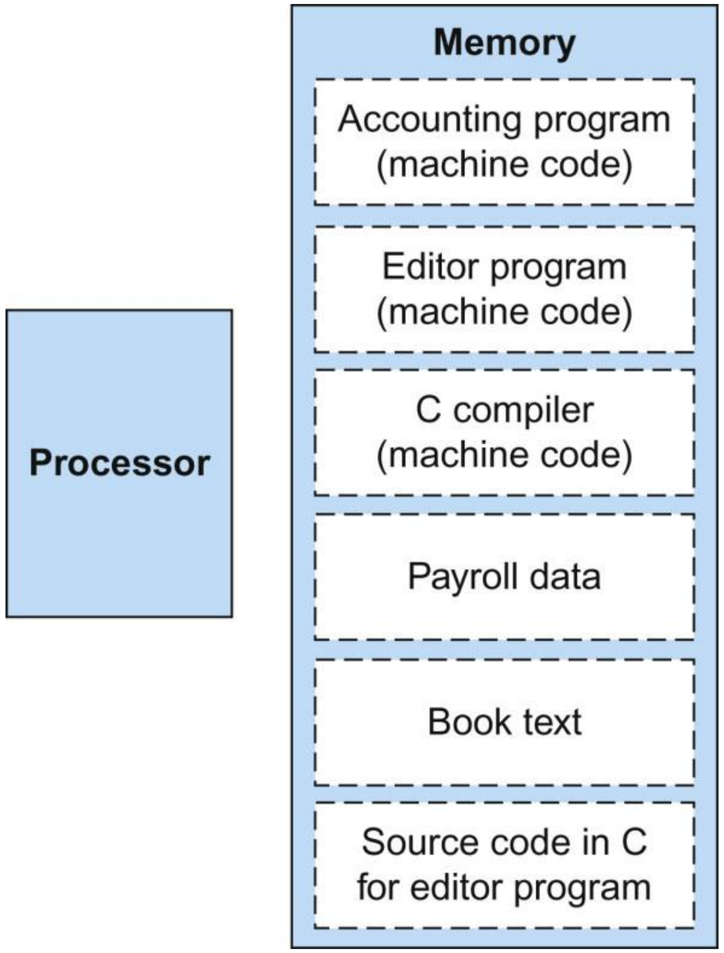
- Programs consist of
  - instructions and data
- Both instructions and data are
  - represented as numbers
    - a series of 0's and 1's
  - stored in a linear memory array

Stored Program Concept (cont.)

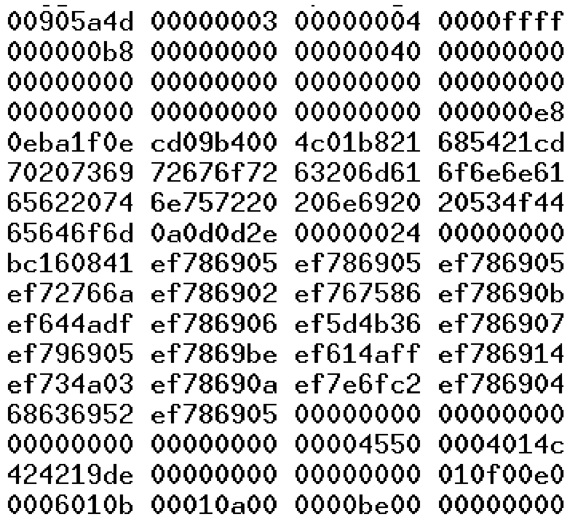
- Before a program runs, it is loaded in memory
- Entire program can be read or written just like data
- Everything (instructions & data) has a memory address
- These apply to all kinds of programs
    - operating systems, applications, games, …
- This concept simplifies SW/HW of computer systems
    - Memory used for both data and code

Stored Program Concept (cont.)

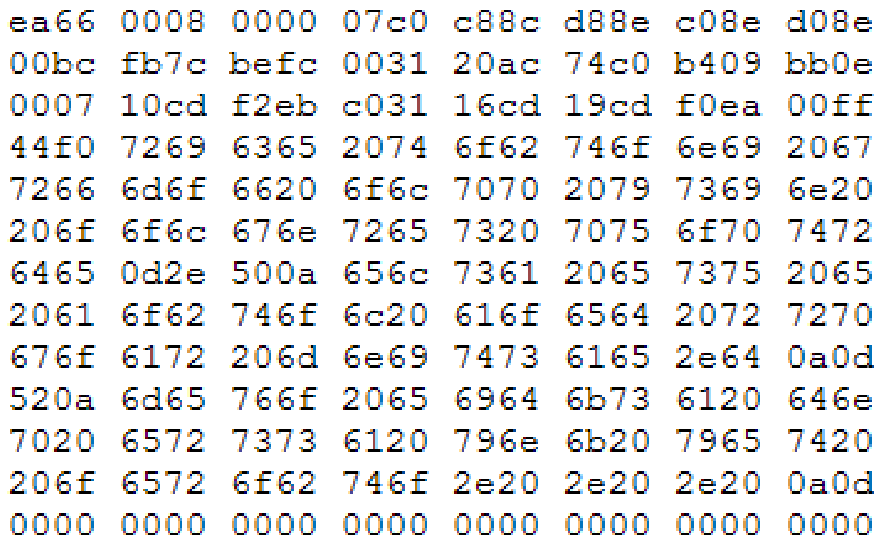
FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

Stored Program Concept (examples)



```
00905a4d 00000003 00000004 0000ffff
000000b8 00000000 00000040 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 000000e8
0eba1f0e cd09b400 4c01b821 685421cd
70207369 72676f72 63206d61 6f6e6e61
65622074 6e757220 206e6920 20534f44
65646f6d 0a0d0d2e 00000024 00000000
bc160841 ef786905 ef786905 ef786905
ef72766a ef786902 ef767586 ef78690b
ef644adf ef786906 ef5d4b36 ef786907
ef796905 ef7869be ef614aff ef786914
ef734a03 ef78690a ef7e6fc2 ef786904
68636952 ef786905 00000000 00000000
00000000 00000000 00004550 0004014c
424219de 00000000 00000000 010f00e0
0006010b 00010a00 0000be00 00000000
```



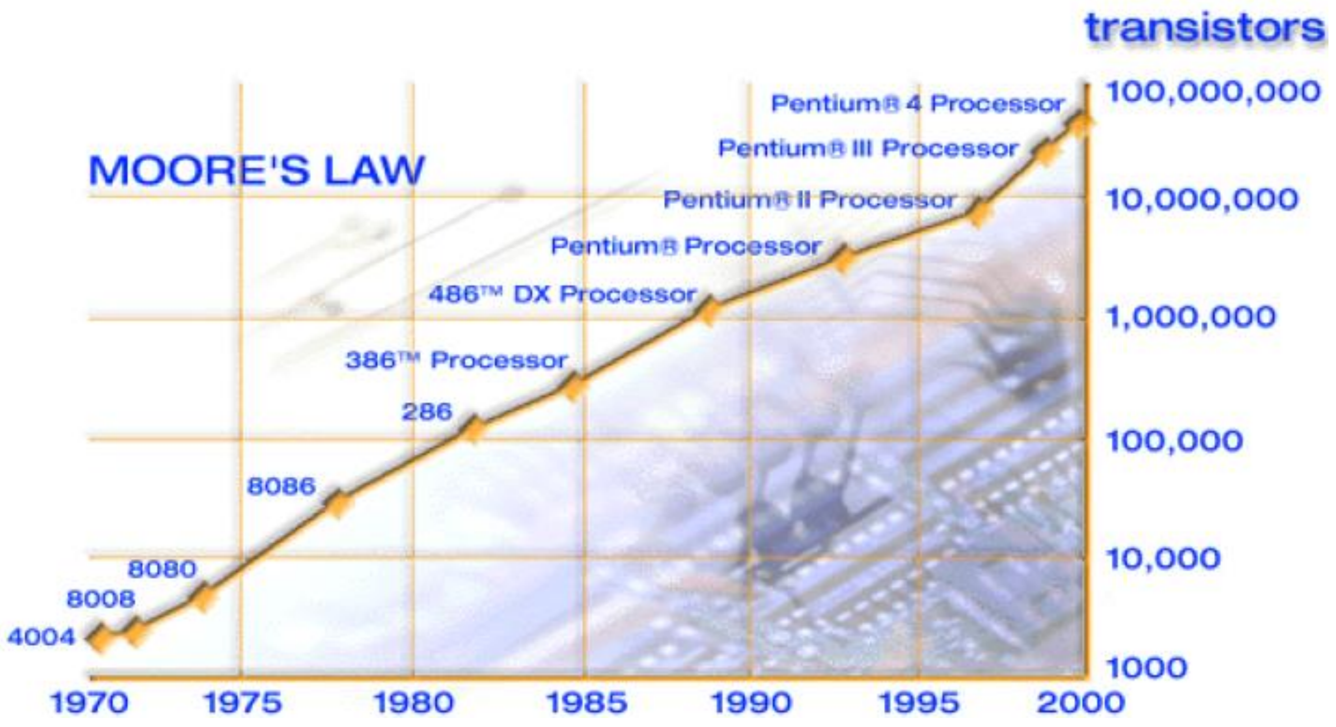
```
ea66 0008 0000 07c0 c88c d88e c08e d08e
00bc fb7c befc 0031 20ac 74c0 b409 bb0e
0007 10cd f2eb c031 16cd 19cd f0ea 00ff
44f0 7269 6365 2074 6f62 746f 6e69 2067
7266 6d6f 6620 6f6c 7070 2079 7369 6e20
206f 6f6c 676e 7265 7320 7075 6f70 7472
6465 0d2e 500a 656c 7361 2065 7375 2065
2061 6f62 746f 6c20 616f 6564 2072 7270
676f 6172 206d 6e69 7473 6165 2e64 0a0d
520a 6d65 766f 2065 6964 6b73 6120 646e
7020 6572 7373 6120 796e 6b20 7965 7420
206f 6572 6f62 746f 2e20 2e20 2e20 0a0d
0000 0000 0000 0000 0000 0000 0000 0000
```

GoogleEarth.exe **Linux Kernel**

Eight Design Ideas

- Design for Moore's Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy

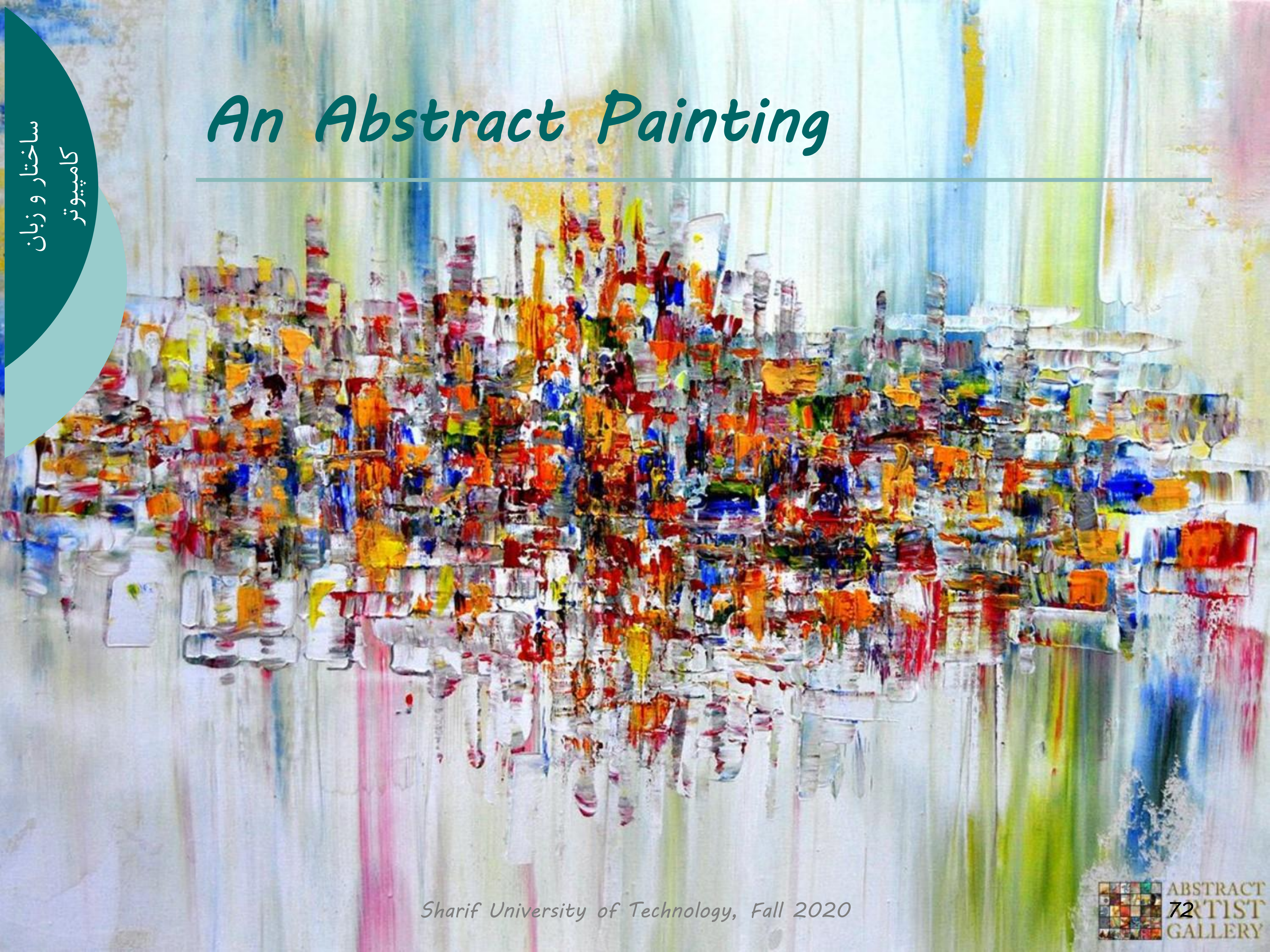
Moore's Law


MOORE'S LAW
transistors
100,000,000
10,000,000
1,000,000
100,000
10,000
1000
1970
1975
1980
1985
1990
1995
2000
4004
8008
8080
8086
286
386™ Processor
486™ DX Processor
Pentium® Processor
Pentium® II Processor
Pentium® III Processor
Pentium® 4 Processor

Abstraction

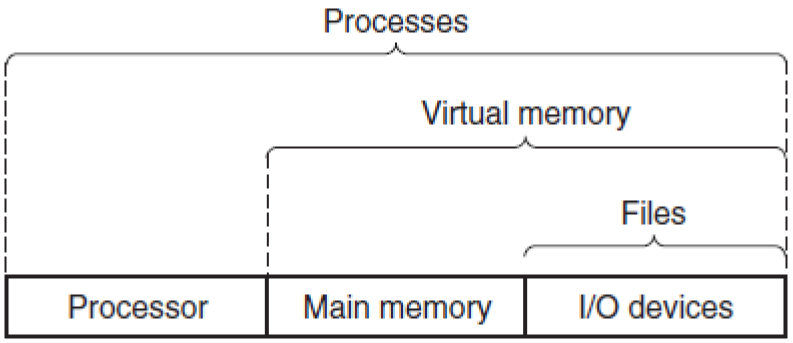
- The process of removing physical, spatial, or temporal details or attributes in the study of objects or systems in order to more closely attend to other details of interest
- Very similar in nature to the process of generalization
- Abstraction enables us to
  - suppress irrelevant details
  - thus reduce a complex subject to something easier to understand

An Abstract Painting



OS Abstractions (Reminder)

- Files:
  - Abstractions for I/O devices
- Virtual memory:
  - Abstraction for both the main memory and disk I/O devices
- Processes:
  - Abstractions for processor, main memory, and I/O devices



Make Common Case Fast

- The common case is often
  - Simpler than the rare case
  - Easier to enhance
- You know the common case by
  - Experimentation
  - Measurement

Amdahl's Law - 1

$$T_{\text{improved}} = T_{\text{affected}}/\text{improvement factor} + T_{\text{unaffected}}$$

- Example 1:
  - Multiply accounts for 80s/100s
  - Reduce multiply execution time to 25%
  - Overall time improvement?
    - $T_{improved}$ = 80/4 + 20 = 40s
    - Overall time reduction to 40%
- Corollary: make the common case fast

Amdahl’s Law - 2

$$T_{\text{improved}} = T_{\text{affected}}/\text{improvement factor} + T_{\text{unaffected}}$$

- Example 2:
  - multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?
    - 20 = 80/n + 20
    - Cannot be done!

Amdahl's Law - 3

- When we speed up one part of a system, the effect on the overall performance depends on
  - how significant this part was and
  - how much it sped up
- To significantly speed up the entire system, improve the speed of a very large fraction of the overall system
- Make common case fast!

Performance via Parallelism

- Parallelism:
  - doing two or more things at once
- Instruction-level parallelism:
  - parallelism is exploited within individual instructions
- Processor-level parallelism:
  - CPUs work together on the same problem

Instruction-level Parallelism

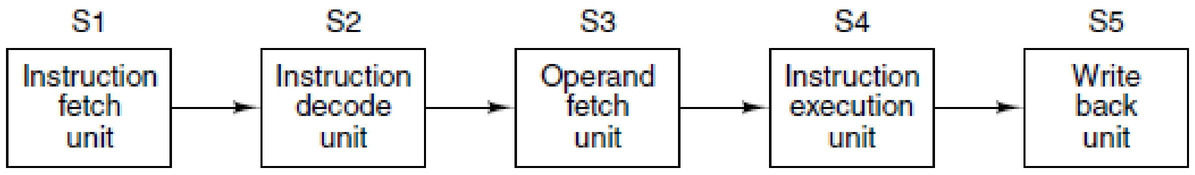
- Parallelism is exploited within individual instructions
  - Pipelining: instruction execution is divided into many parts, run in parallel
  - Superscalar Architectures: two or more instructions executed in parallel

Prefetching

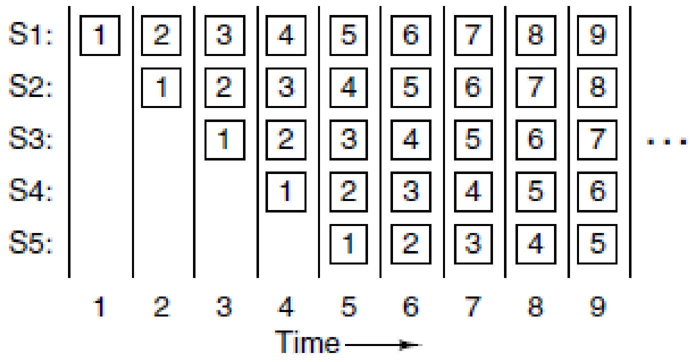
- A concept used in early designs
- Instruction execution divided into independent phases:
  - Fetch
  - Actual execution
- Phases of subsequent instructions executed in parallel

Performance via Pipelining

Pipelining carries prefetching much further

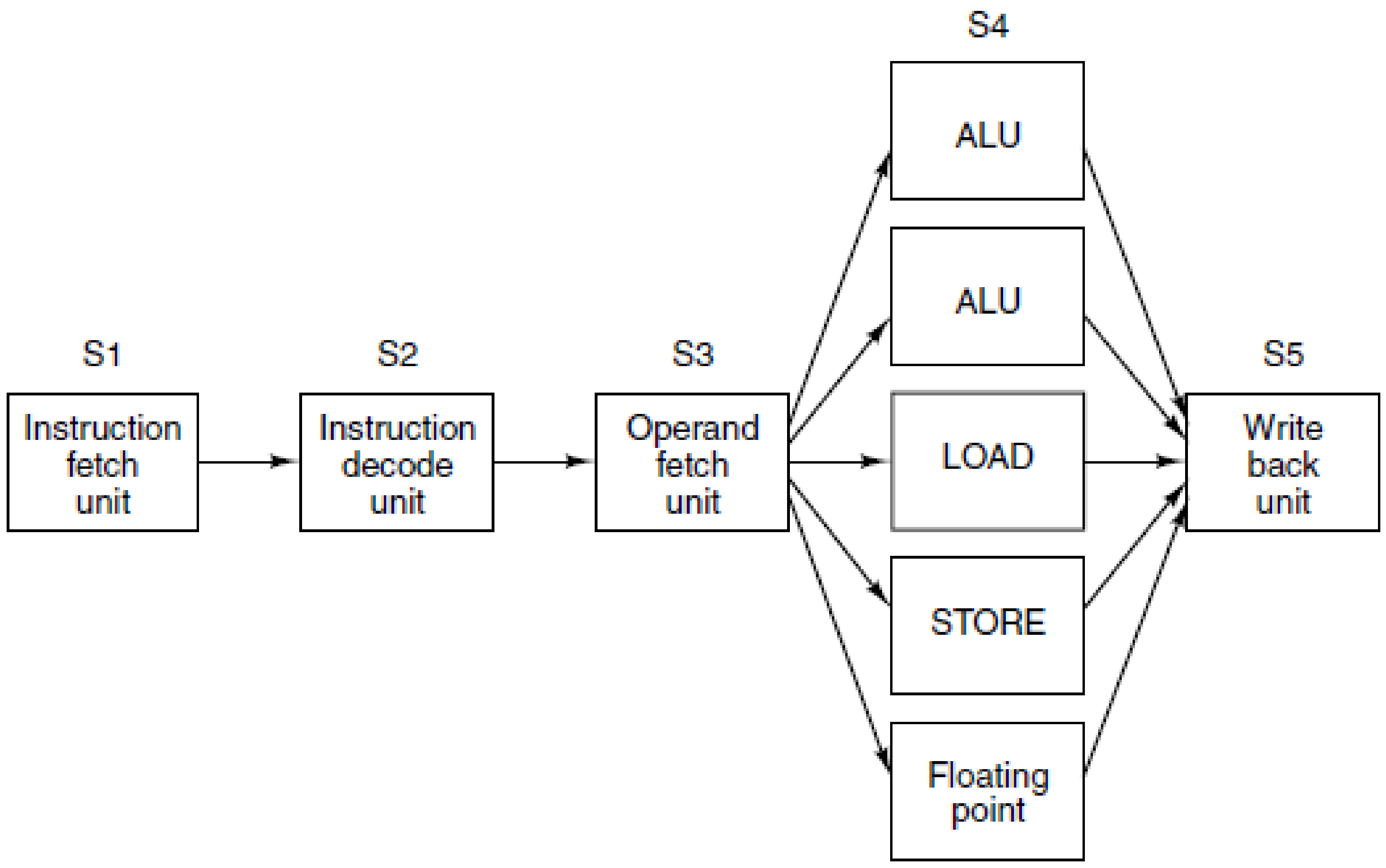


(a)



(b)

A Superscalar Processor

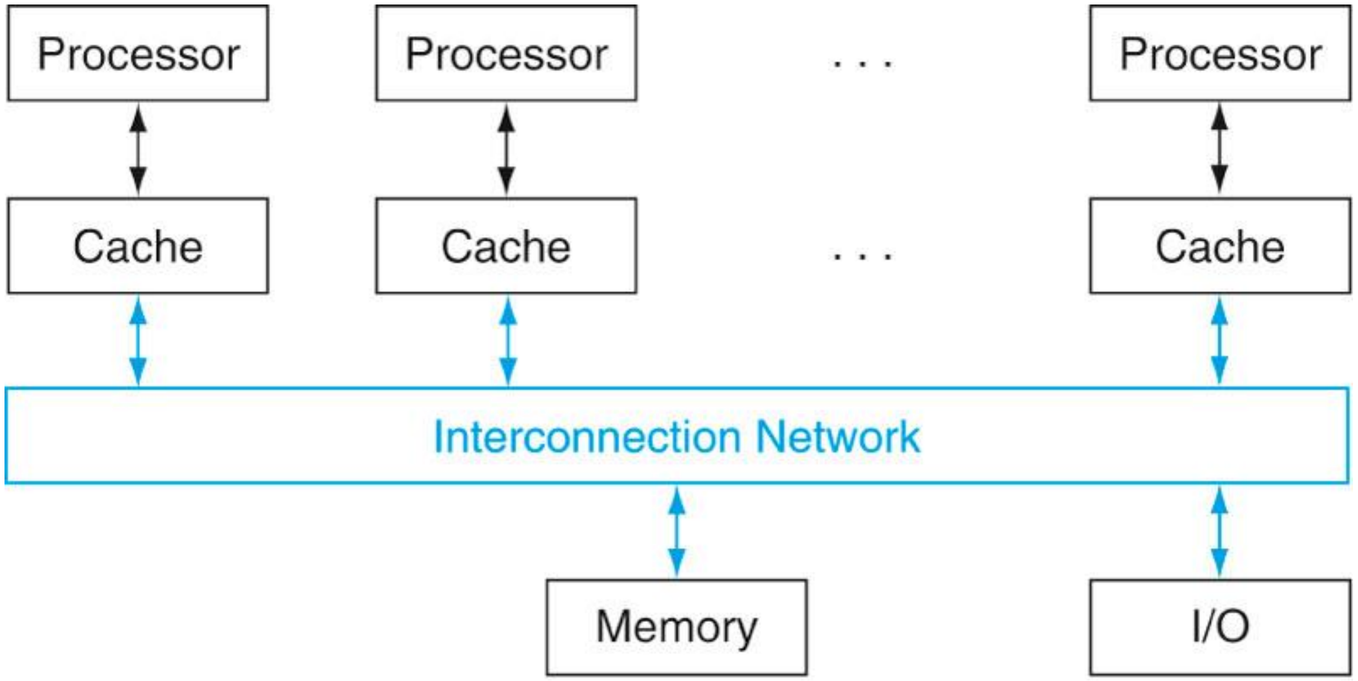


Processor-level Parallelism

- CPUs work together on the same problem
  - Data Parallel Computers:
    - same calculations performed on different sets of data
  - Multiprocessors: (tightly coupled)
    - a system with more than one CPU sharing a common memory
  - Multicomputers: (loosely coupled)
    - large numbers of interconnected computers, each having its own private memory

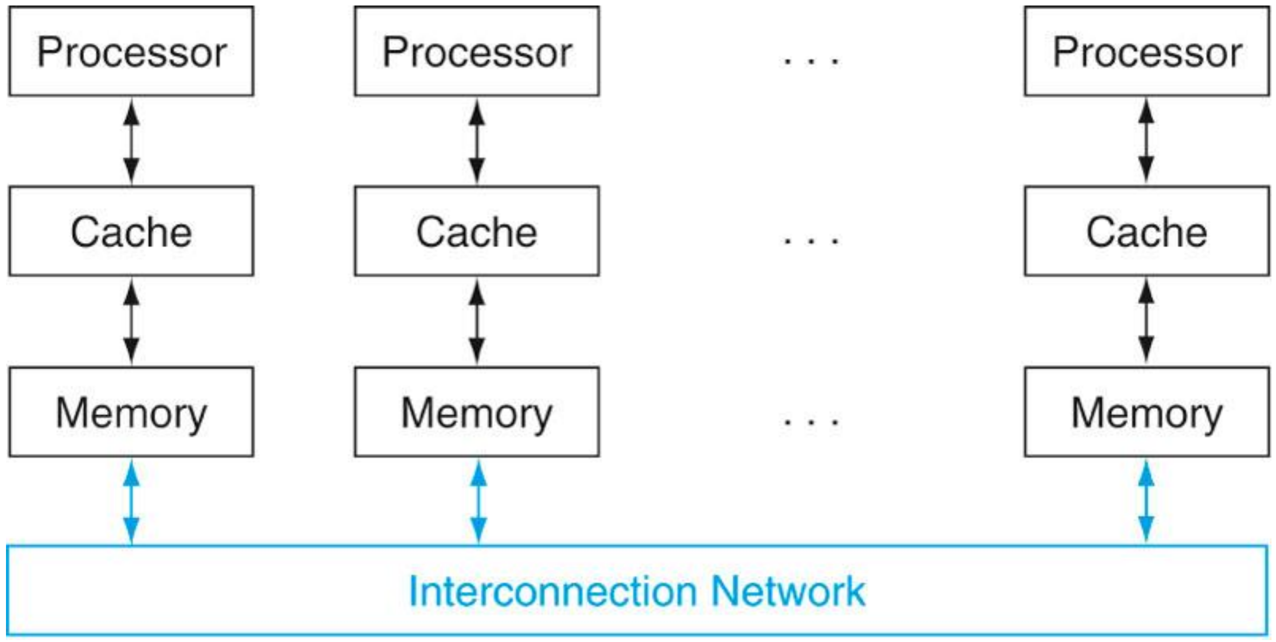
Multiprocessors

More than one CPU sharing a common memory

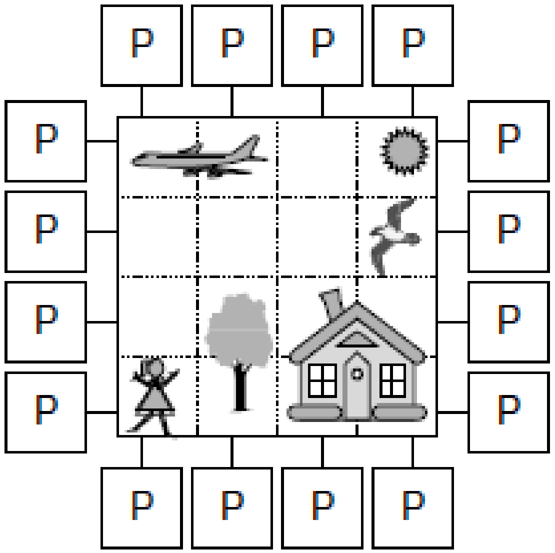


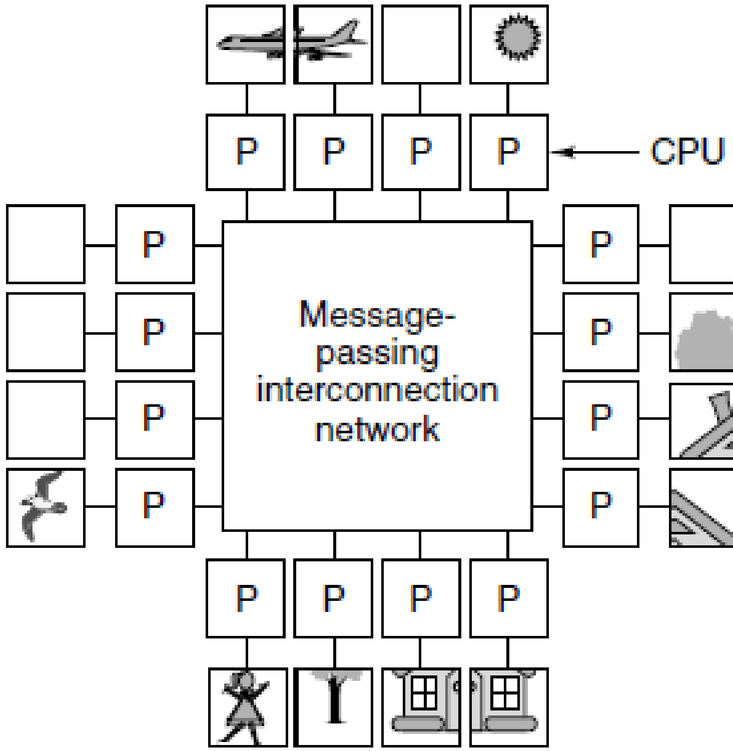
Multicomputers

large numbers of interconnected computers each having its own private memory



Multiprocessing vs. Multicomputing





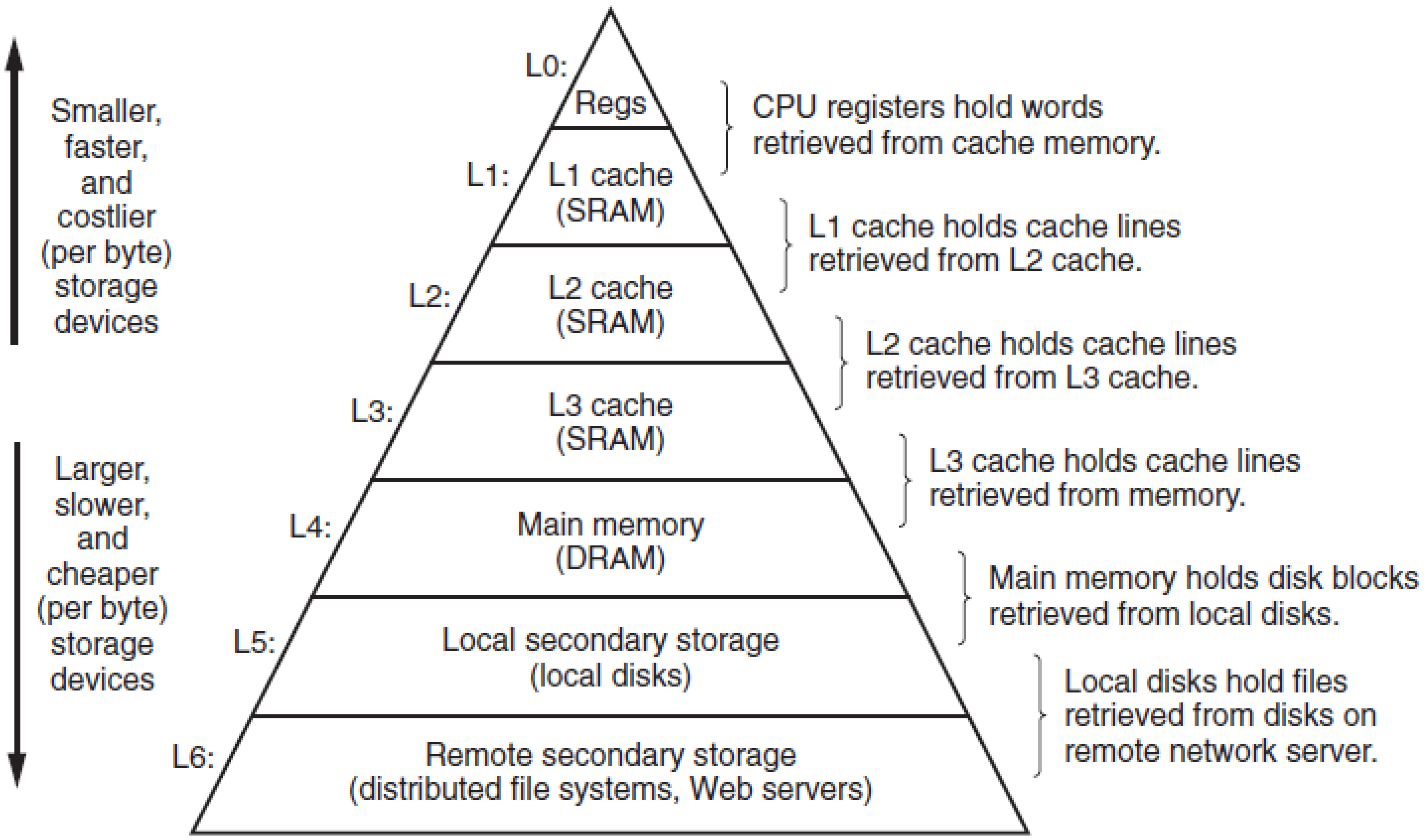
Parallelism (Summary)

- Parallelism: doing two or more things at once
- Instruction-level parallelism: parallelism is exploited within individual instructions
  - Pipelining: instruction execution is divided into many parts, run in parallel
  - Superscalar Architectures: two or more instructions executed in parallel
- Processor-level parallelism: CPUs work together on the same problem
  - Data Parallel Computers:
    - same calculations performed on different sets of data
  - Multiprocessors: (tightly coupled)
    - a system with more than one CPU sharing a common memory
  - Multicomputers: (loosely coupled)
    - large numbers of interconnected computers, each having its own private memory

Performance via Prediction

- It can be better to ask for forgiveness than to ask for permission!
- It can be faster on avg. to guess & start working rather than wait until you know for sure, if:
  - the mechanism to recover from a misprediction is not too expensive
  - your prediction is relatively accurate

Memory Hierarchy (Reminder)


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: Regs
L1: L1 cache (SRAM)
L2: L2 cache (SRAM)
L3: L3 cache (SRAM)
L4: Main memory (DRAM)
L5: Local secondary storage (local disks)
L6: Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network server.

Dependability via Redundancy

- Computers need to be dependable
- Any physical device can fail
- We make systems dependable by redundant components that:
  - take over when a failure occurs
  - help detect failures

Outlines

- The life-time of a program
- Hardware organization of a typical system
- Operating System
- Stored Program Concept
- Computer design ideas
- Next Topic:

  Instruction Set Architecture (ISA)