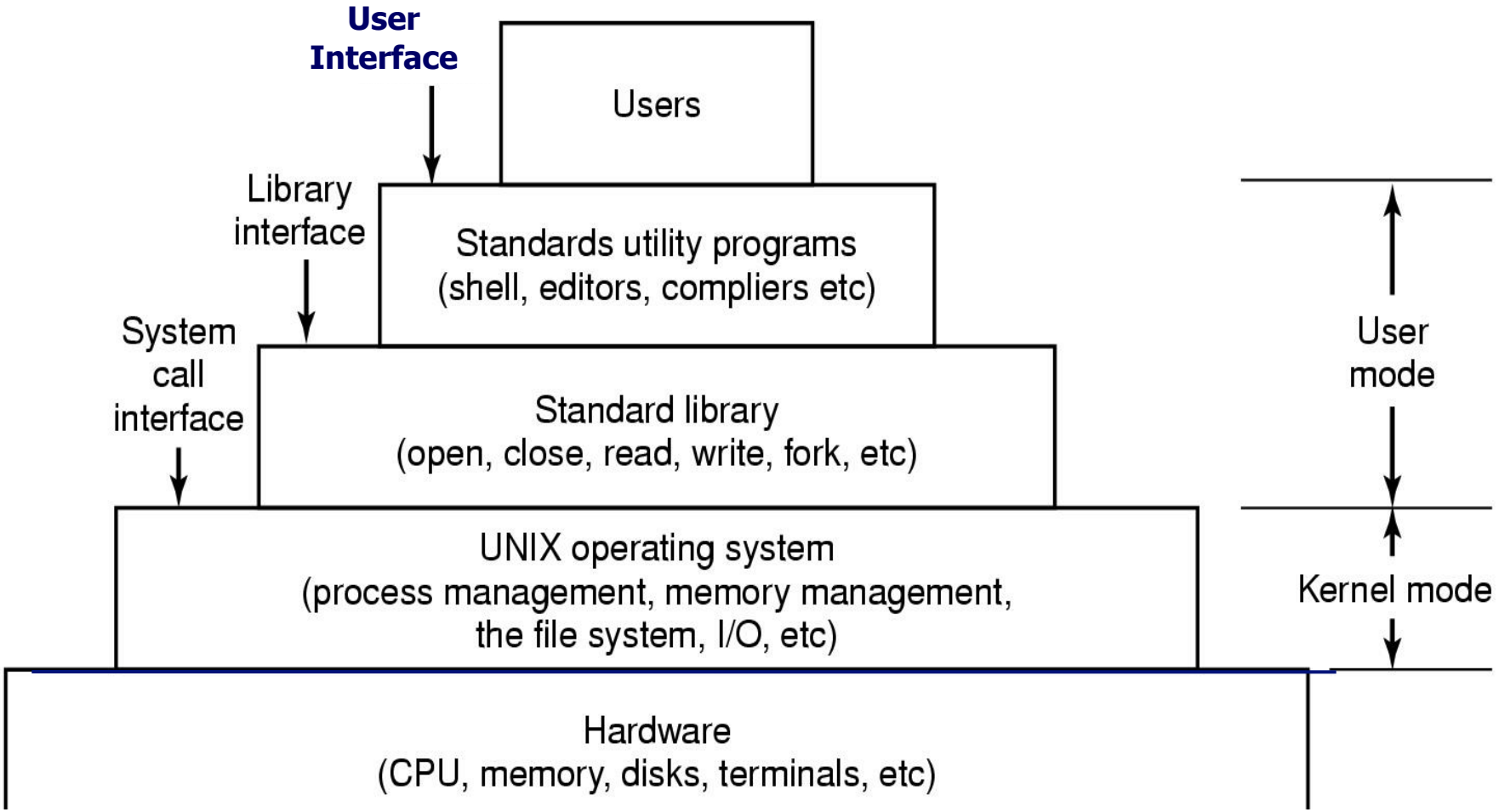
Computer Systems Organization

Today's agenda

Overview of how things work

- Compilation and linking system
- Operating system
- Computer organization

A software view



How it works

hello.c program

```
#include <stdio.h>
int main() {
   printf("hello, world\n");
}
```

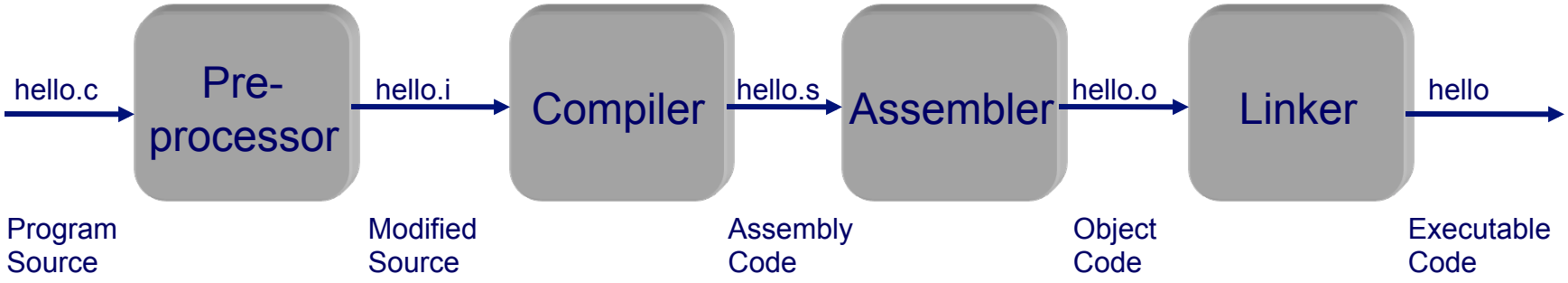
The Compilation system

gcc is the *compiler driver*

gcc invokes several other *compilation phases*

- **Preprocessor**
- **Compiler**
- **Assembler**
- **Linker**

What does each one do? What are their outputs?



Preprocessor

First, gcc compiler driver invokes cpp to generate expanded C source

- cpp just does text substitution
- Converts the C source file to another C source file
- Expands #defines, #includes, etc.
- Output is another C source

Preprocesser

Included files:

```
#include <foo.h>
#include "bar.h"
```

Defined constants:

```
#define MAXVAL   40000000
```

By convention, all capitals tells us it's a constant, not a variable.

Macros:

```
#define MIN(x,y)    ((x)<(y) ? (x):(y))
#define RIDX(i, j, n)   ((i) * (n) + (j))
```

Preprocesser

Conditional compilation:

```
#ifdef …    or  #if defined( … )
#endif
```

- Code you think you may need again (e.g. debug print statements)
  - Include or exclude code based on #define/#ifdef
  - More readable than commenting code out
- Portability
  - Compilers have “built in” constants defined
  - Operating system specific code
    - » #if defined(__i386__) || defined(WIN32) || …
  - Compiler-specific code
    - » #if defined(__INTEL_COMPILER)
  - Processor-specific code
    - » #if defined(__SSE__)

Compiler

Next, gcc compiler driver invokes cc1 to generate assembly code

- **Translates high-level C code into assembly**
  - **Variable abstraction mapped to memory locations and registers**
  - **Logical and arithmetic functions mapped to underlying machine opcodes**

Assembler

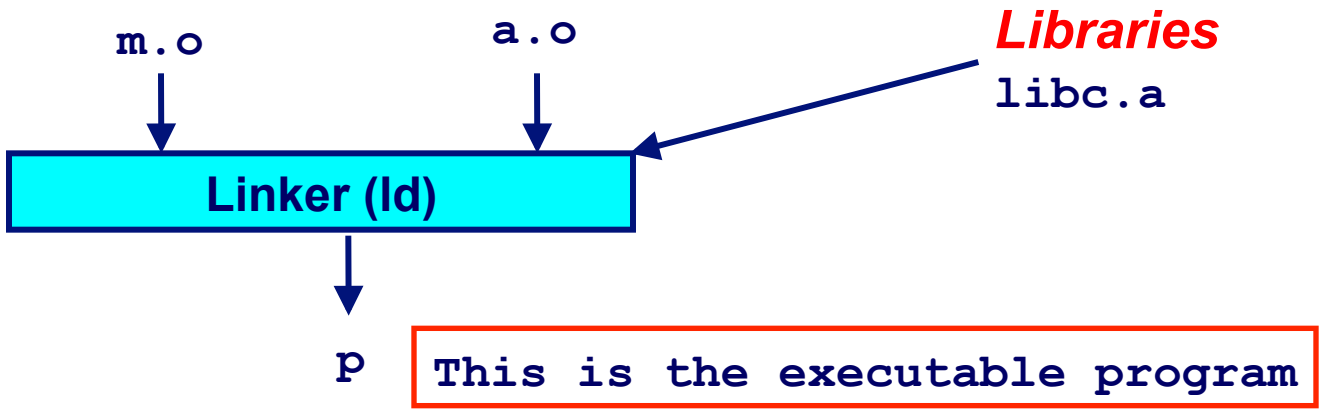
Next, gcc compiler driver invokes as to generate object code

- **Translates assembly code into binary object code that can be directly executed by CPU**

Linker

Finally, gcc compiler driver calls linker (ld) to generate executable

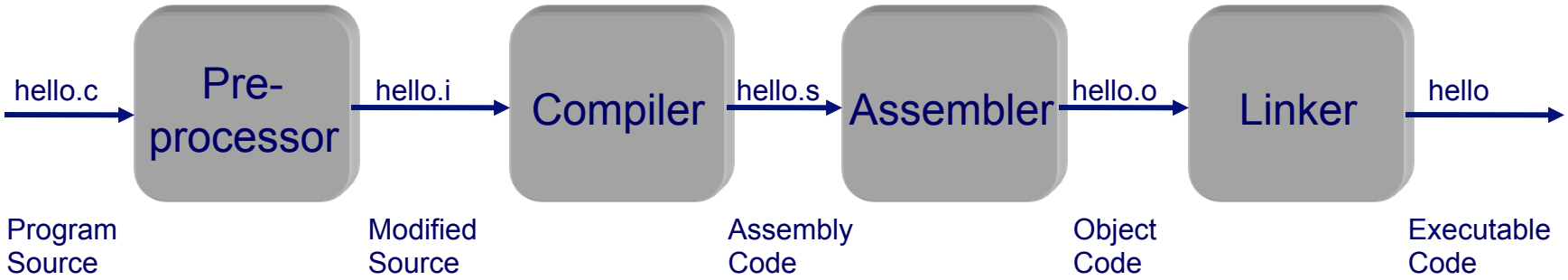
- **Links together object code and static libraries to form final executable**



Summary of compilation process

Compiler driver **(cc or gcc) coordinates all steps**

- **Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).**
- **Passes command line arguments to appropriate phases**



http://thefengs.com/wuchang/courses/cs201/class/03

GCC variations

Stop after compiler stage

What form is the input?

Run just the preprocessor (cpp)

- `gcc -m32 -E hello.c -o hello.i`

Run just the C compiler

- `gcc -m32 -S -x cpp-output hello.i -o hello.s`

Run just the assembler (as)

- `gcc -m32 -c -x assembler hello.s -o hello.o`

Run just the linker (ld)

- `gcc -m32 hello.o -o hello`

Compile for 32-bit architecture
`-m64`

The linking process (ld)

Merges object files

- Merges multiple relocatable (.o) object files into a single executable program.

Resolves external references

- *External reference*: reference to a symbol defined in another object file.
- Resolves multiply defined symbols with some restrictions
- Strong symbols = initialized global variables, functions
- Weak symbols = uninitialized global variables used to allow overrides of function implementations
- Rules
    - Multiple strong symbols not allowed
    - Choose strong symbols over weak symbols
    - Choose any weak symbol if multiple ones exist

Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Efficiency

- Compilation time
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file used by all programs
  - Executable files and running memory images contain only code for the functions they actually use.

Libraries and linking

Two types of libraries

- Static libraries
  - Library of code that linker copies into the executable at compile time
- Dynamic shared object libraries
  - Code loaded at run-time by system loader upon program execution

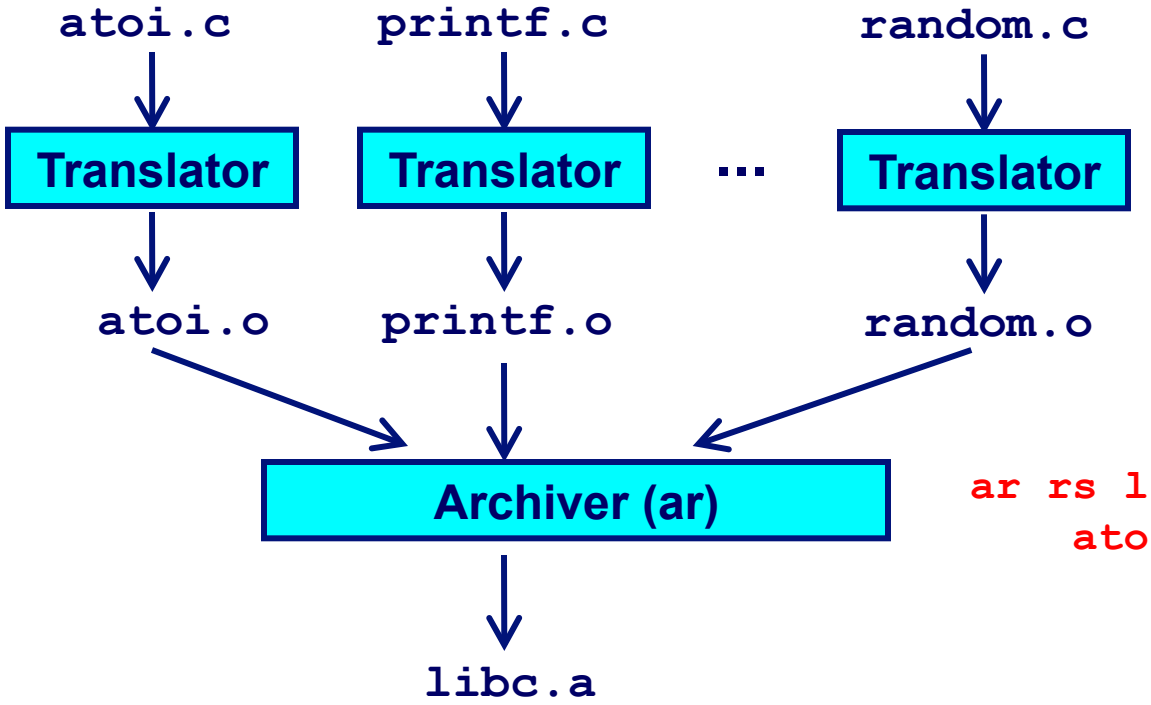
Static linking

Linking process

- Relocates symbols from their relative locations in the `.o` files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
  - References in both code and data

```
a();          /* reference to symbol a */
int *xp=&x;   /* reference to symbol x */
```

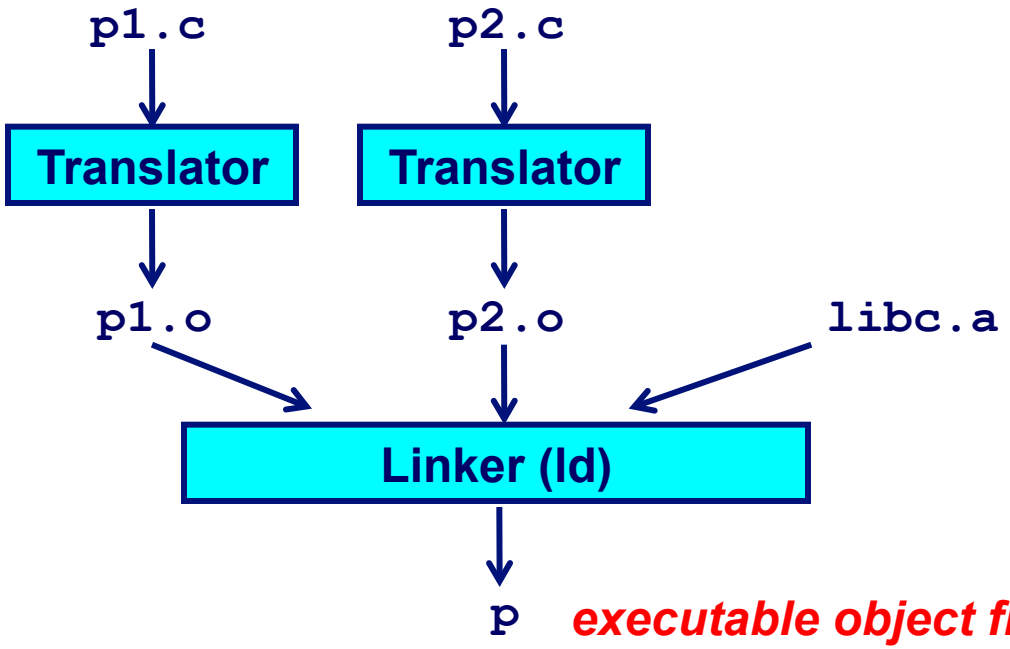
Static library example




```
ar rs libc.a \
    atoi.o printf.o … random.o
```

The C standard library

Static library example



static library (archive) of relocatable object files concatenated into one file.

executable object file (with code and data for `libc` functions needed by `p1.c` and `p2.c` copied in)

Creating static libraries

Suppose you have utility code in x.c, y.c, and z.c that all of your programs use

- **Option 1: Copy and link together individual .o files**

```
gcc -o hello hello.o x.o y.o z.o
```

- **Option 2: Create a library *libmyutil.a* using ar and ranlib and link library in statically**

```
libmyutil.a : x.o y.o z.o
    ar rvu libmyutil.a x.o y.o z.o
    ranlib libmyutil.a
    gcc -o hello hello.c -L. -lmyutil
```

  - **Note: Only the library code "hello" needs from *libmyutil* is copied directly into binary**

```
nm libmyutil.a
```

  - **Lists functions in library**

http://thefengs.com/wuchang/courses/cs201/class/03/libexample

Commonly Used Libraries

`libc.a` (the C standard library)

- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 1 MB archive of 226 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Problems with static libraries

Multiple copies of common code on disk

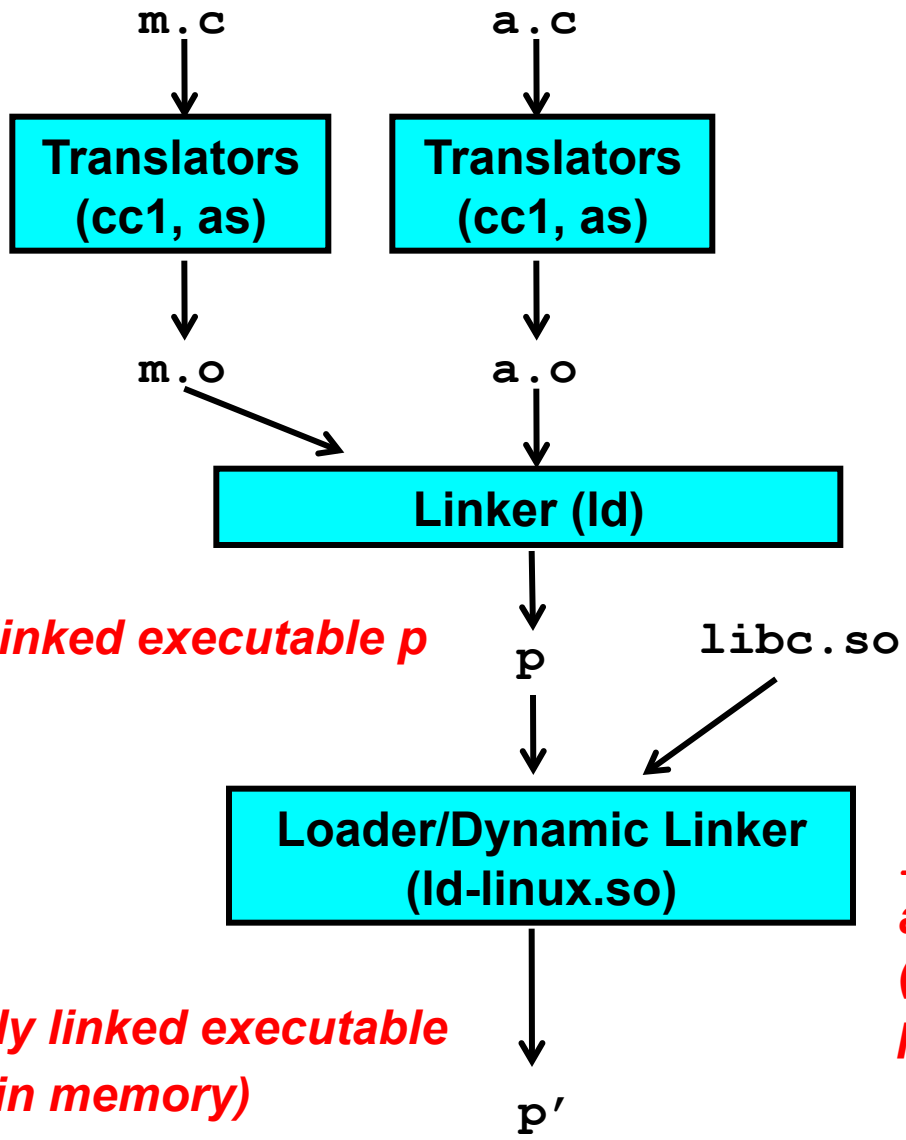
- "gcc program.c –lc" creates an a.out with libc object code copied into it (libc.a)
- Almost all programs use libc!
- Large amounts of disk space with the same code in it

Dynamic libraries

Have binaries compiled with a reference to a library of shared objects versus an entire copy of the library

- Libraries loaded at run-time from file system
- "ldd <binary>" to see dependencies
  - gcc flags "–shared" and "-soname" to create dynamic shared object files (.so)
- Caveat
  - How does one ensure dynamic libraries are present across all run-time environments?
  - Static linking (via gcc's –static flag)
    - Self-contained binaries to avoid problem with DLL versions

Dynamically Linked Shared Libraries



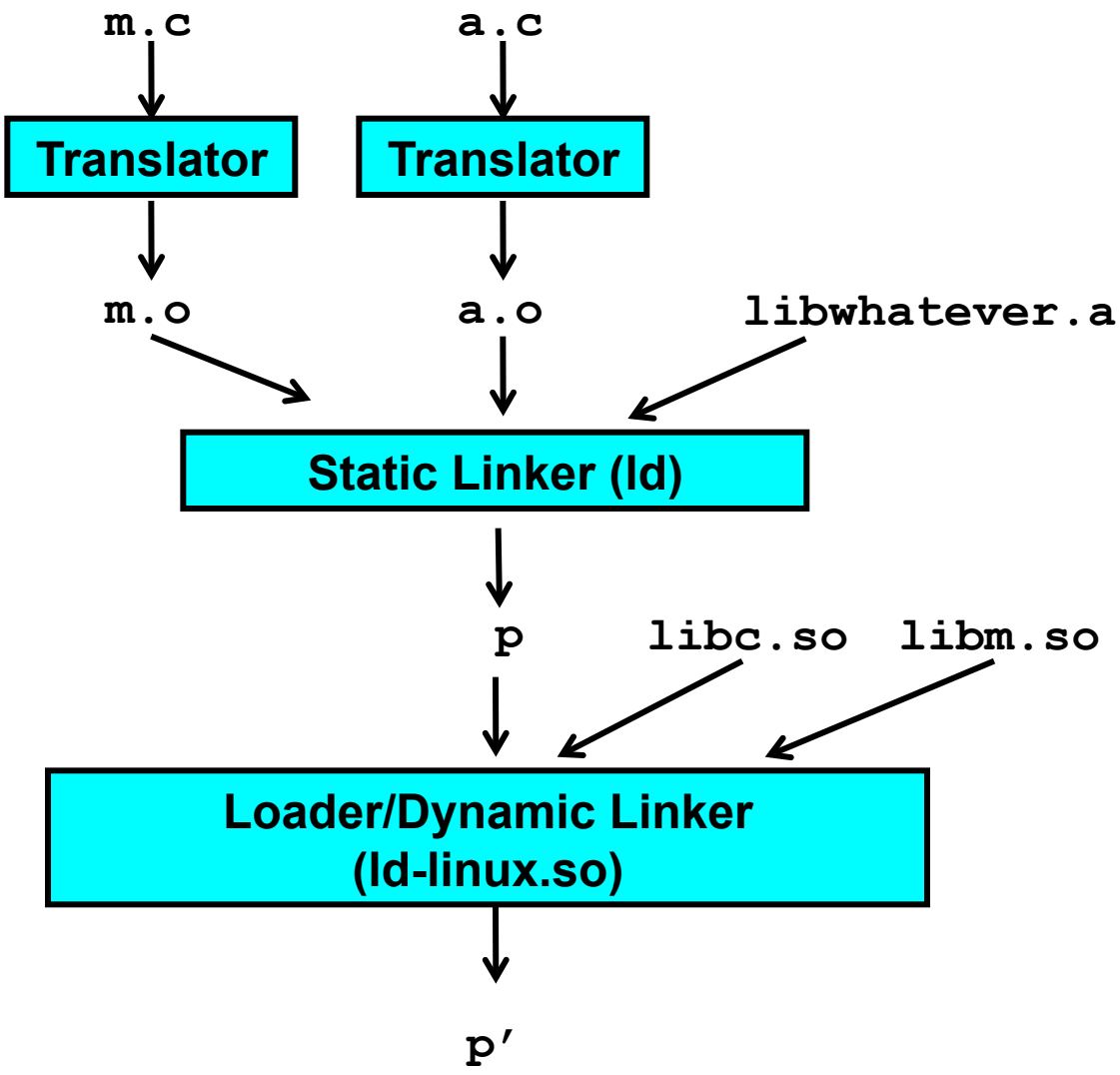
Partially linked executable p (on disk)

Shared library of dynamically relocatable object files

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and (potentially) shared among processes.

Fully linked executable p' (in memory)

The Complete Picture



Program execution

gcc/cc output an executable in the ELF format (Linux)

- Executable and Linkable Format

Standard unified binary format for

- Relocatable object files (`.o`),
- Shared object files (`.so`)
- Executable object files

Equivalent to Windows Portable Executable (PE) format

ELF Object File Format

ELF header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Program header table

- Page size, addresses of memory segments (sections), segment sizes.

`.text` section

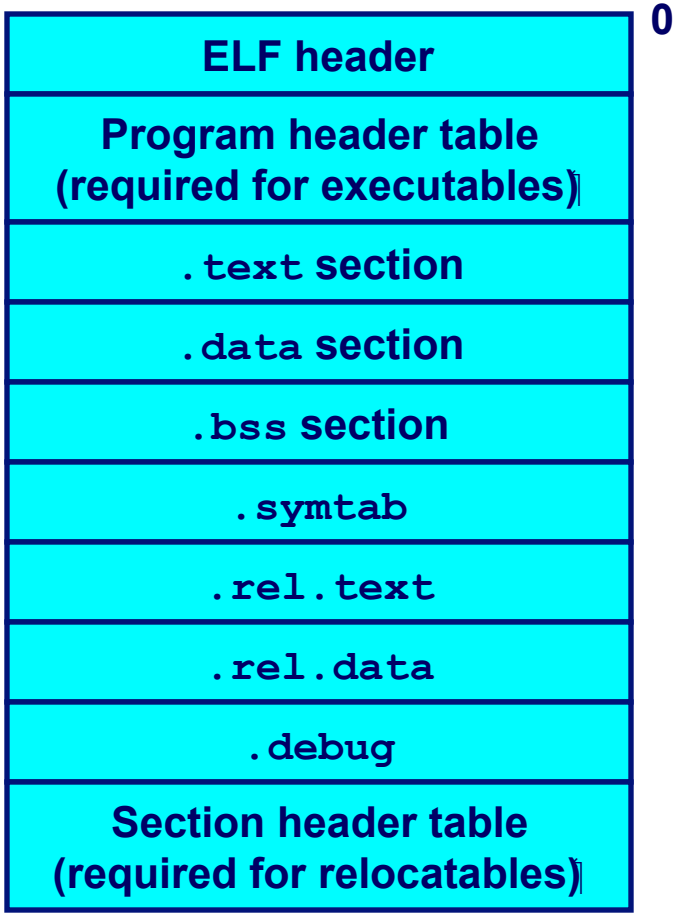
- Code

`.data` section

- Initialized (static) data

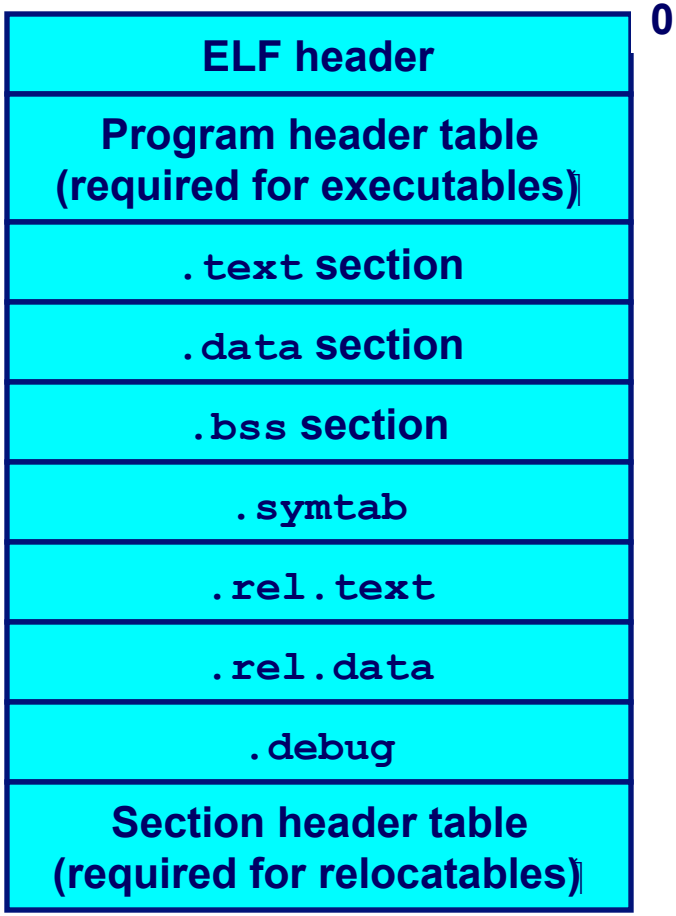
`.bss` section

- Uninitialized (static) data
- "Block Started by Symbol"



ELF Object File Format (cont)

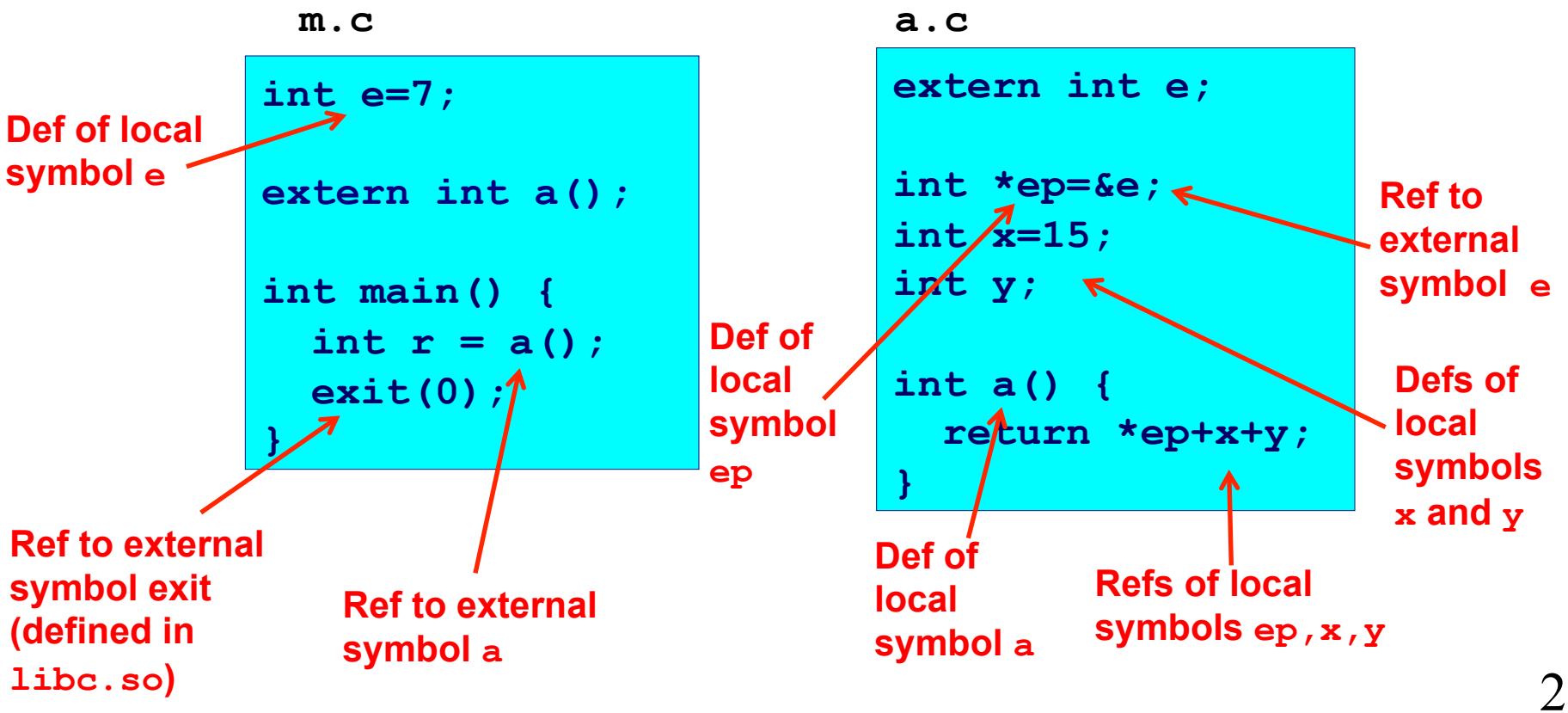
- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for `.text` section
- **`.rel.data` section**
  - Relocation info for `.data` section
- **`.debug` section**
  - Info for symbolic debugging (`gcc -g`)



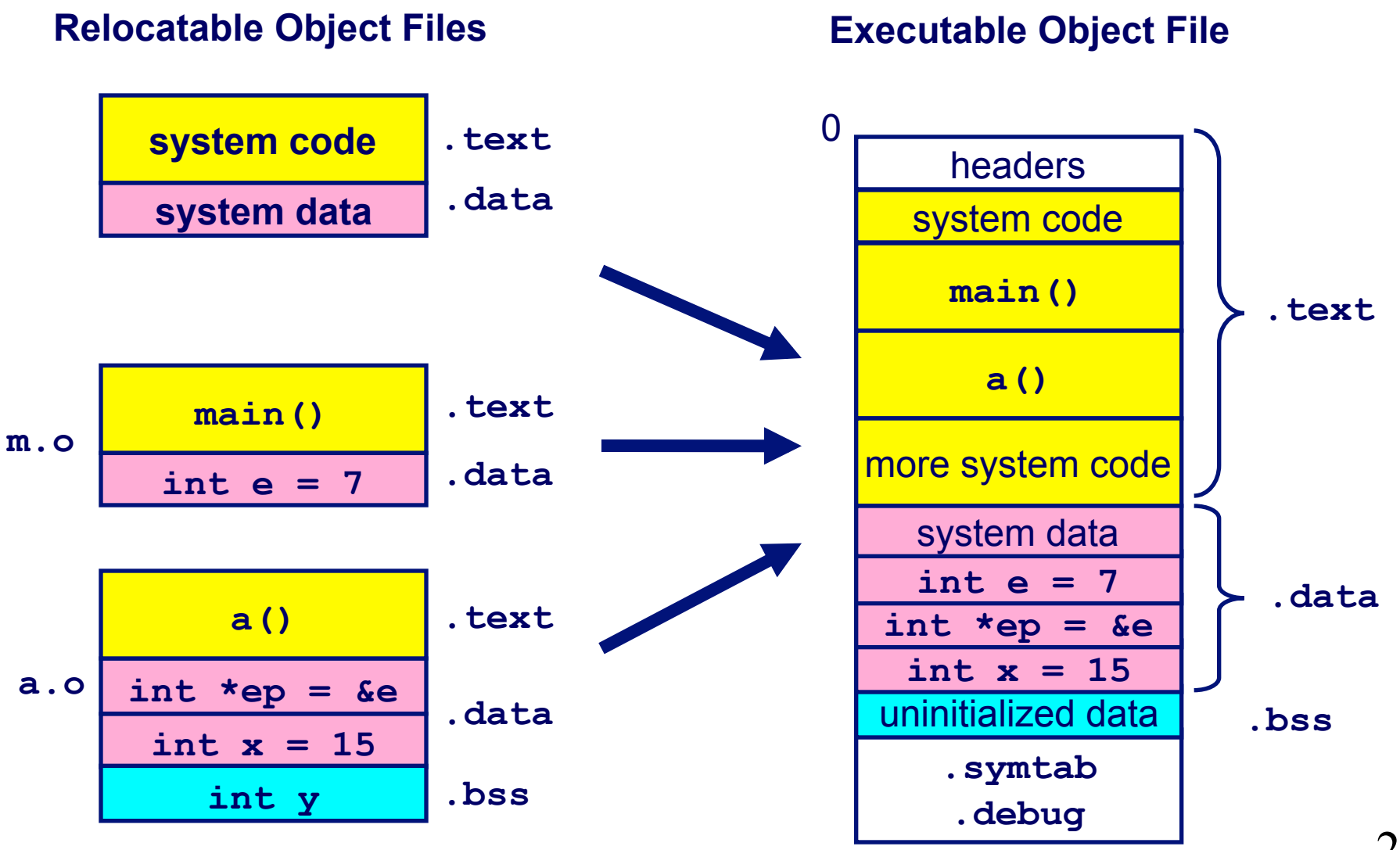

```
readelf -a
```

Example

- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.
- Addresses of variables/functions must be resolved when loaded
  - More on this later



Merging Object Files into an Executable Object File



A word about relocation

Compiler does not know where code will be loaded into memory upon execution

- Instructions and data that depend on location must be "fixed" to actual addresses
- i.e., variables, pointers, jump instructions

`.rel.text` section

- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying
- (e.g., a() in m.c)

`.rel.data` section

- Addresses of pointer data that will need to be modified in the merged executable
- (e.g., ep in a.c)

Want to know more about linking?

cs.pdx.edu/~harry/Blitz/BlitzDoc/ObjectFileFormat.pdf

- Describes the format of object and executable files
- Not ELF, but similar in spirit
- Simplified
- Also discusses the linking process

The run-time system

Program runs on top of operating system that implements

- File system
- Memory management
- Processes
- Device management
- Network support
- etc.

Operating system functions

Protection

- Protects the hardware/itself from user programs
- Protects user programs from each other
- Protects files from unauthorized access

Resource allocation

- Memory, I/O devices, CPU time, space on disks

Operating system functions

Abstract view of resources

- Files as an abstraction of storage devices
- System calls an abstraction for OS services
- Virtual memory a uniform memory space abstraction for each process
  - Gives the illusion that each process has entire memory space
- A process (in conjunction with the OS) provides an abstraction for a virtual computer
  - Slices of CPU time to run in

Unix file system

Key concepts

- **Everything is a file.**
  - **Keyboards, mice, CD-ROMS, disks, modems, networks, pipes, sockets**
  - **One abstraction for accessing most external things**
- **A file is a stream of bytes with no other structure.**
  - **On the hard disk or from an I/O device**
  - **Higher levels of structure are an application concept, not an operating system concept**
    - » No “records” (contrast with Windows/VMS)

Unix file systems

Managed by OS on disk

- Dynamically allocates space for files
- Implements a name space so we can find files
- Hides where the file lives and its physical layout on disk
- Provides an illusion of sequential storage

All we have to know to find a file is its name

Process abstraction

A fundamental concept of operating systems.

A process is an instance of a program when it is running.

- A program is a file on the disk containing instructions to execute
- A process is an instance of that program loaded in memory and running
  - Like you baking the cookies, following the instructions

A process includes

- Code and data in memory, CPU state, open files, thread of execution

How does a program get executed?

The operating system creates a process.

- Including among other things, a virtual memory space

System loader reads program from file system and loads its code into memory

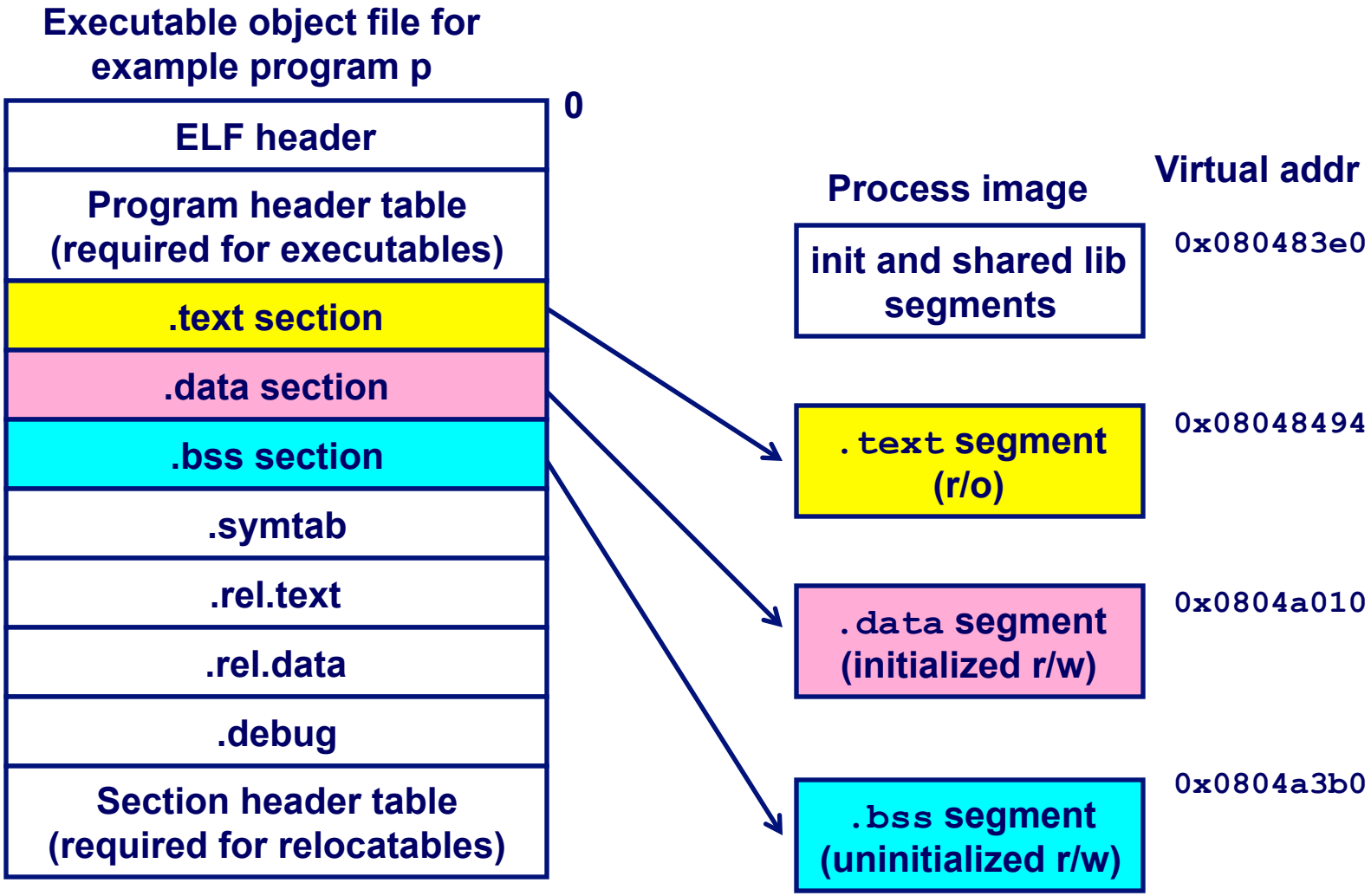
- Program includes any statically linked libraries
- Done via DMA (direct memory access)

System loader loads dynamic shared objects/libraries into memory

Then it starts the thread of execution running

- Note: the program binary in file system remains and can be executed again

Loading Executable Binaries



Where are programs loaded in memory?

To start with, imagine a primitive operating system.

- Single tasking.
- Physical memory addresses go from zero to N.

The problem of loading is simple

- Load the program starting at address zero
- Use as much memory as it takes.
- Linker binds the program to absolute addresses
- Code starts at zero
- Data concatenated after that
- etc.

Where are programs loaded, cont'd

Next imagine a multi-tasking operating system on a primitive computer.

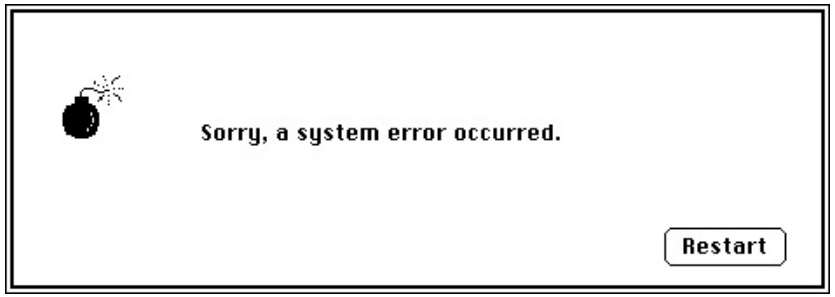
- Physical memory space, from zero to N.
- Applications share space
- Memory allocated at load time in unused space
- Linker does not know where the program will be loaded
- Binds together all the modules, but keeps them relocatable

How does the operating system load this program?

- Not a pretty solution, must find contiguous unused blocks

How does the operating system provide protection?

- Not pretty either



Where are programs loaded, cont'd

Next, imagine a multi-tasking operating system on a modern computer, with hardware-assisted virtual memory

The OS creates a virtual memory space for each user's program.

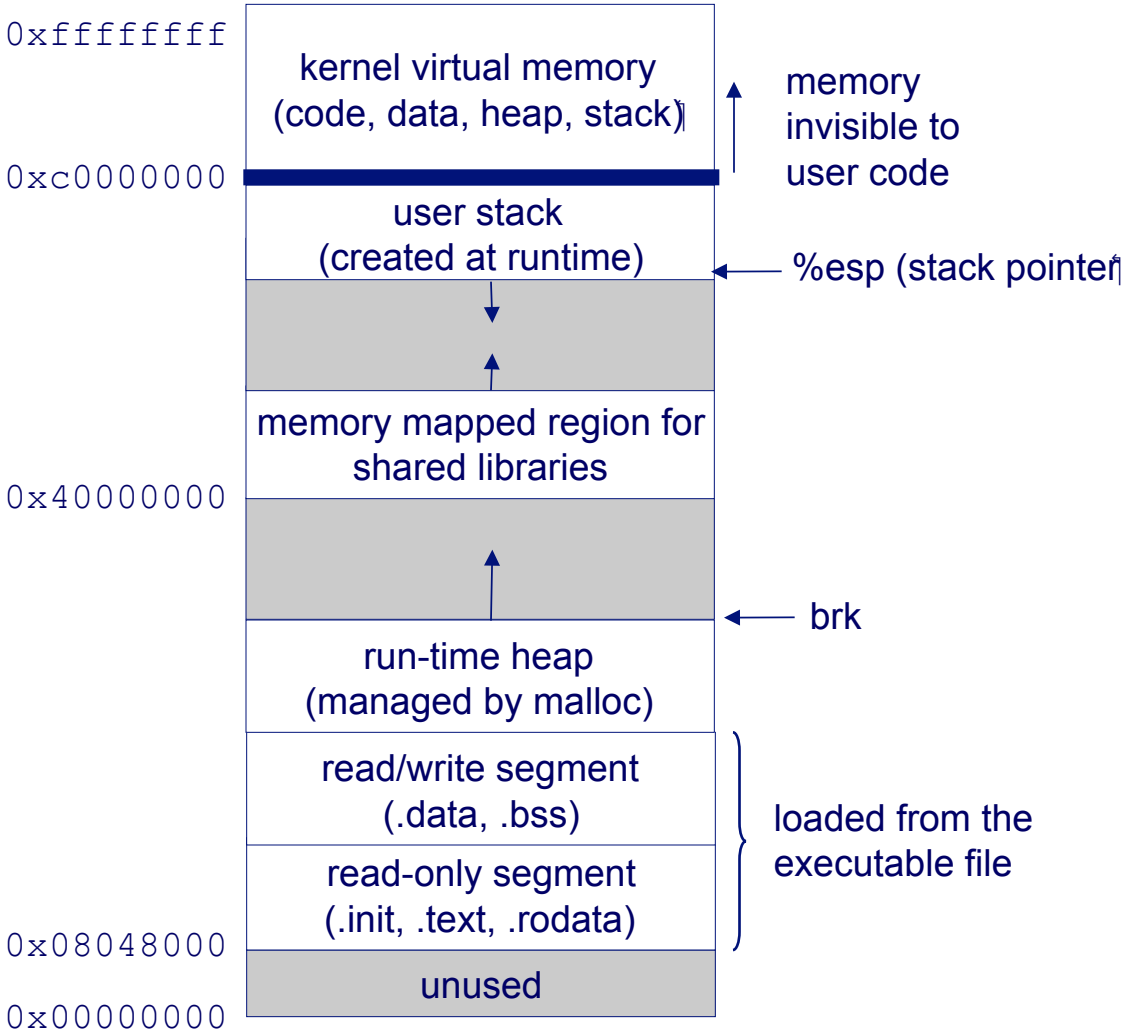
- As though there is a single user with the whole memory all to itself.

Now we're back to the simple model

- The linker statically binds the program to virtual addresses
- At load time, the operating system allocates memory, creates a virtual address space, and loads the code and data.
- Binaries are simply virtual memory snapshots of programs (Windows .com format)

Example memory map

Nothing is left relocatable, no relocation at load time



Modern linking and loading

Dynamic linking and loading

- Single, uniform, “flat” VM address space
- But, code must be relocatable again
  - Many dynamic libraries, no fixed/reserved addresses to map them into
  - As a security feature to prevent predictability in exploits (Address-Space Layout Randomization)

The memory hierarchy

- Operating system and CPU memory management unit gives each process the “illusion” of a uniform, dedicated memory space
  - i.e. 0x0 – 0xFFFFFFFF for IA32
  - Allows multitasking
  - Hides underlying non-uniform memory hierarchy

Memory heirarchy motivation

In 1980

- CPUs ran at around 1 mHz.
- A memory access took about as long as a CPU instruction
- Memory was not a bottleneck to performance

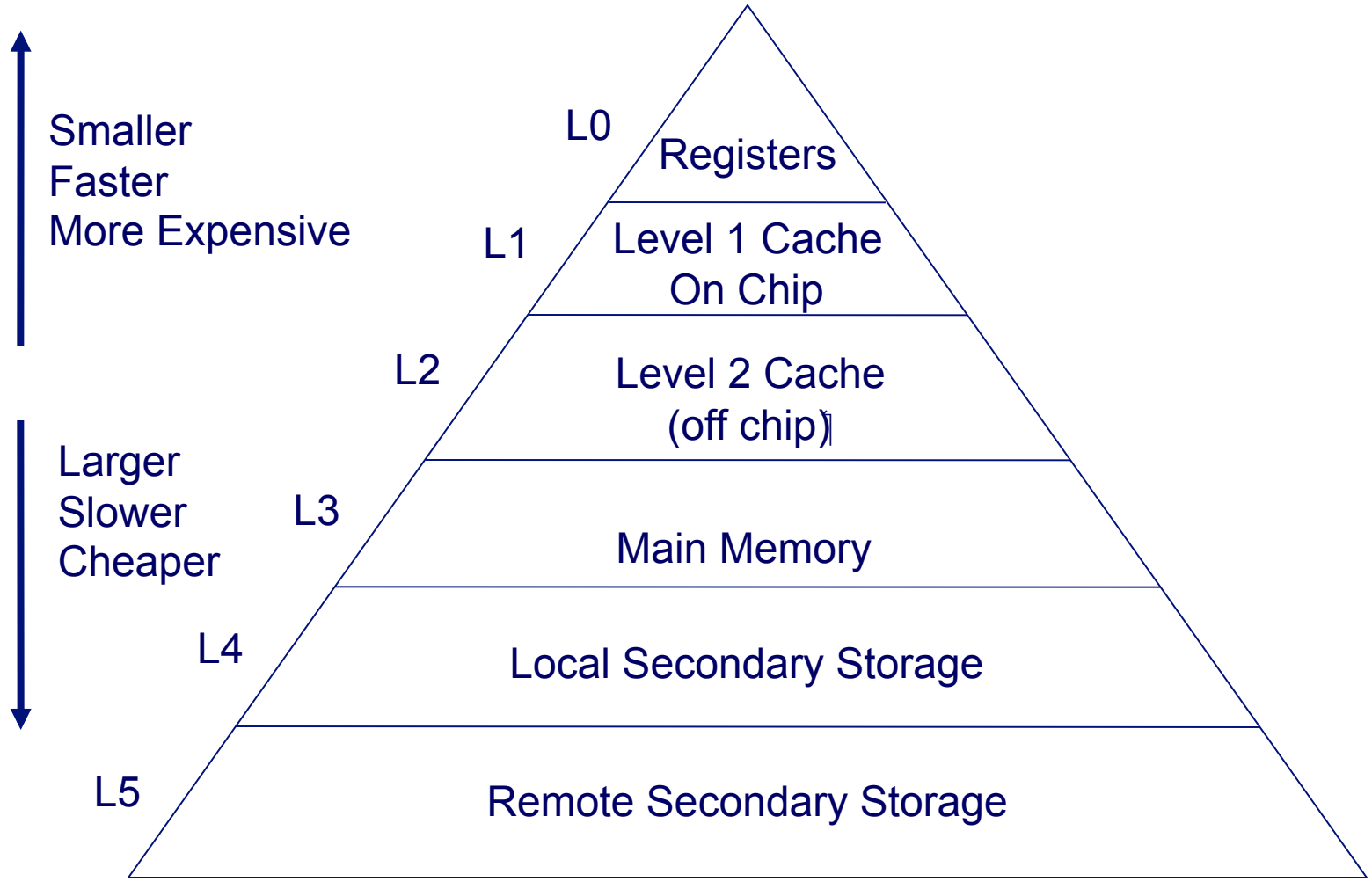
Today

- CPUs are about 3000 times faster than in 1980
- DRAM Memory is about 10 times faster than in 1980

We need a small amount of faster, more expensive memory for stuff we'll need in the near future

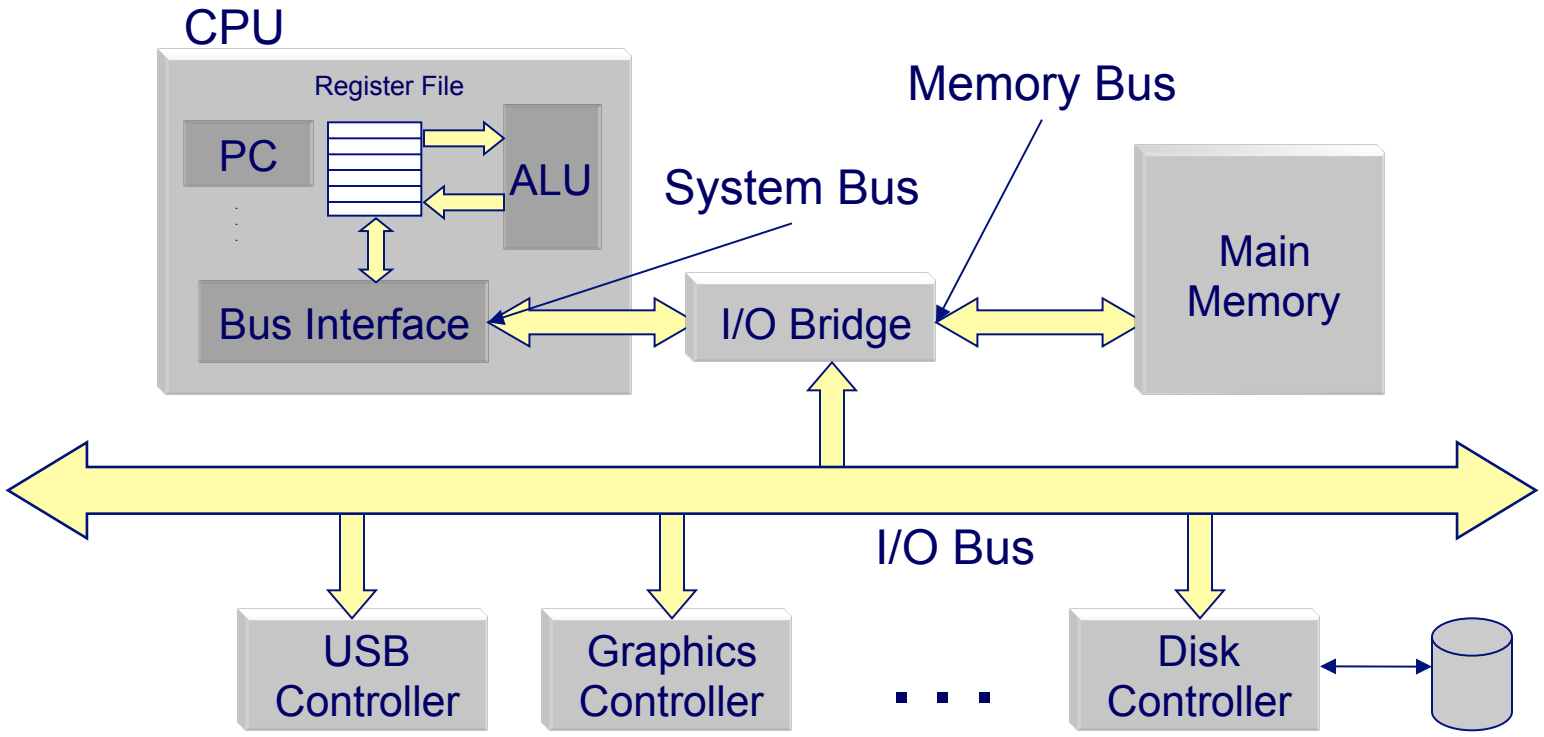
- How do you know what you'll need in the future?
- Locality
- L1, L2, L3 caches

The memory heirarchy



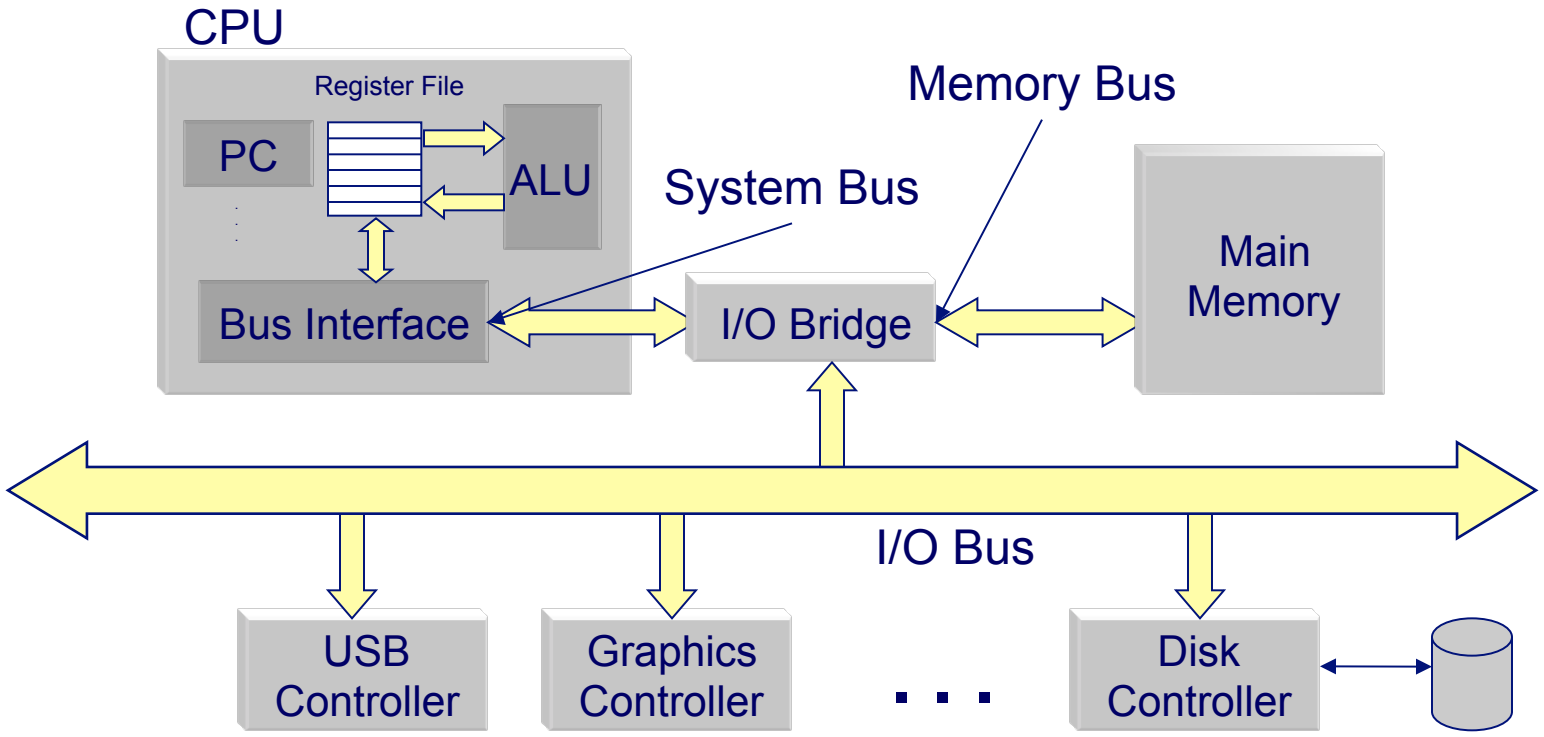
Hardware organization

The last piece…how does it all run on hardware?



Summary using hello.c

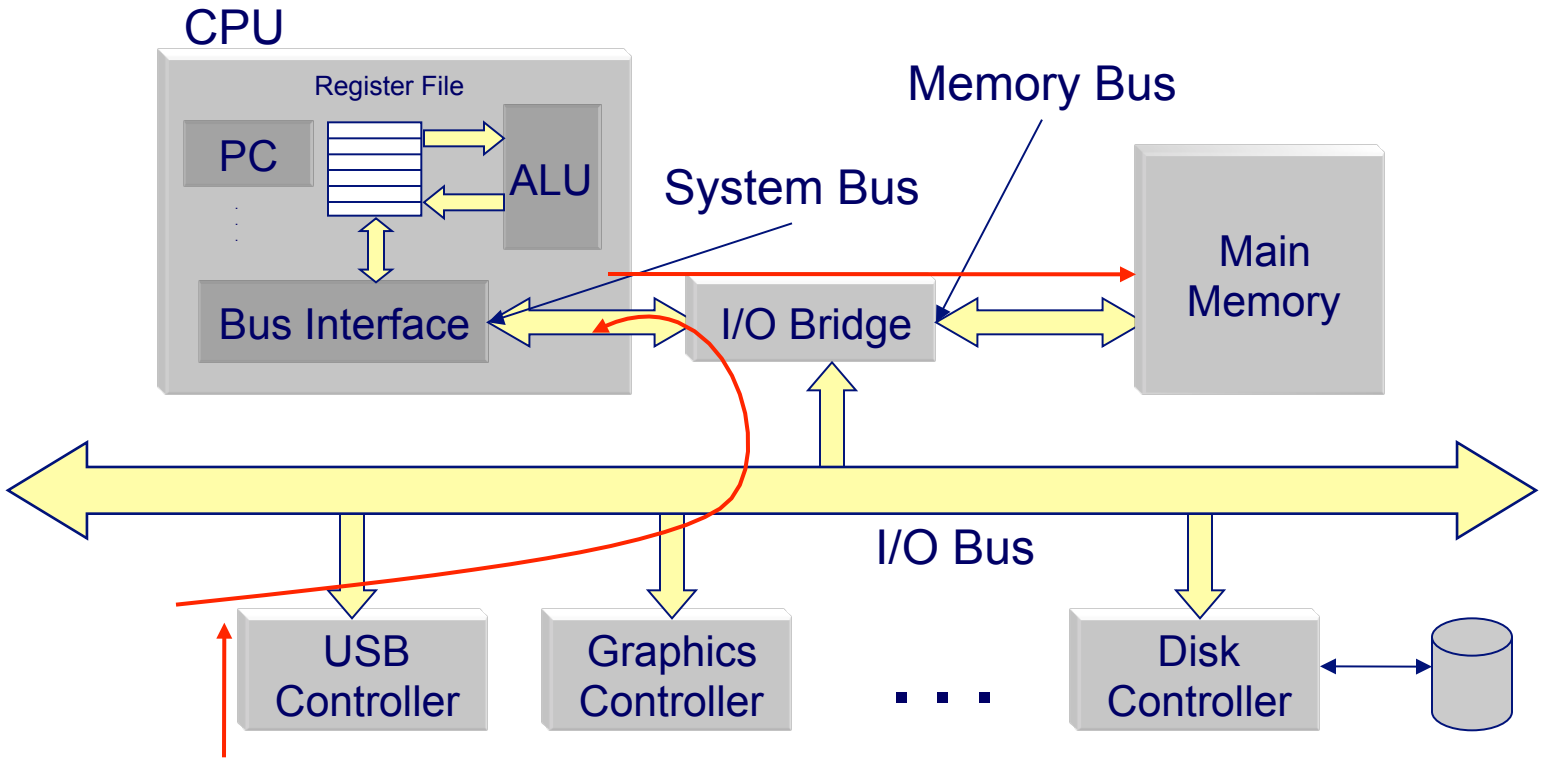
1. **Shell process running, waiting for input**



Summary using hello.c

3. Command read into registers

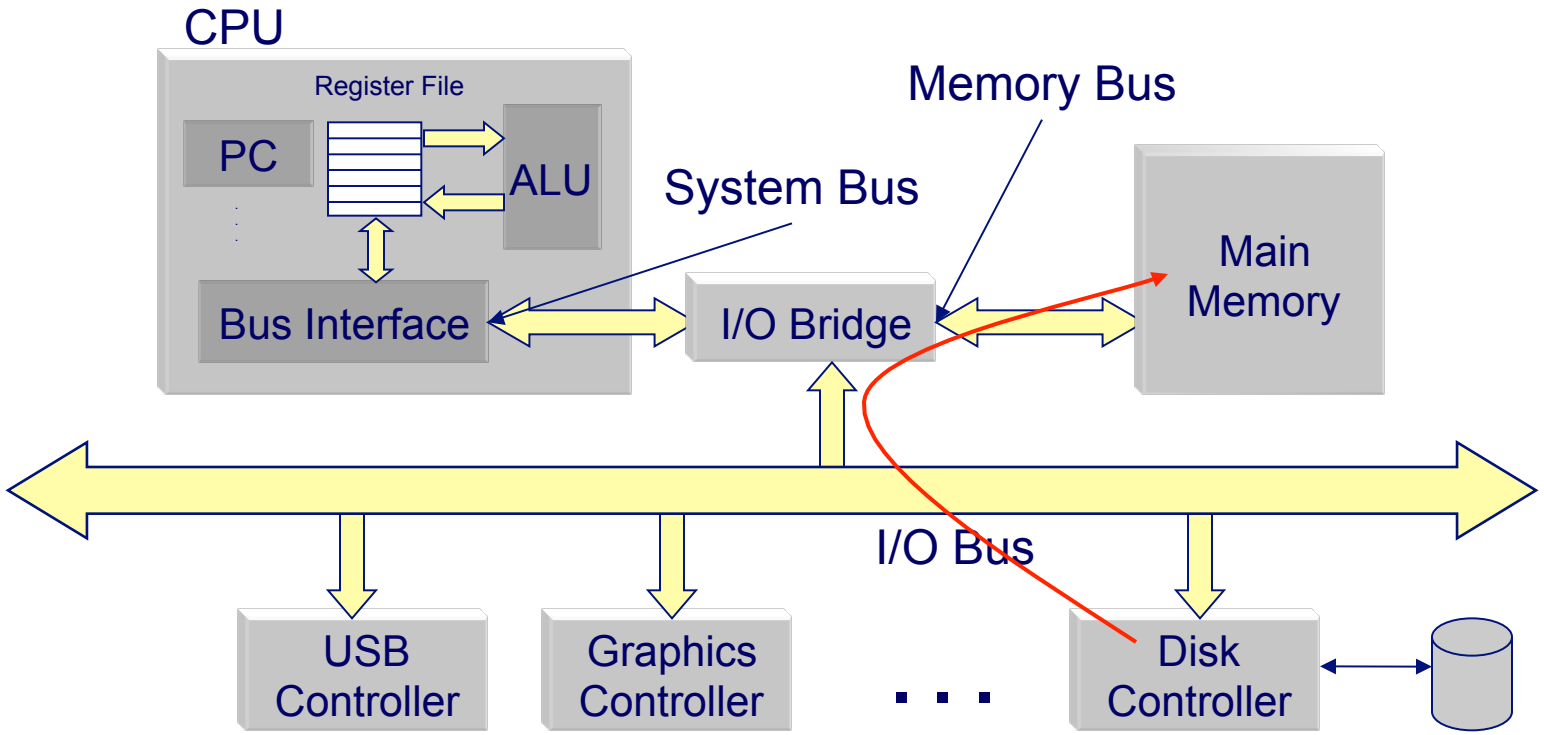
4. Before sent to main memory before being read by shell process



2. User types ./hello

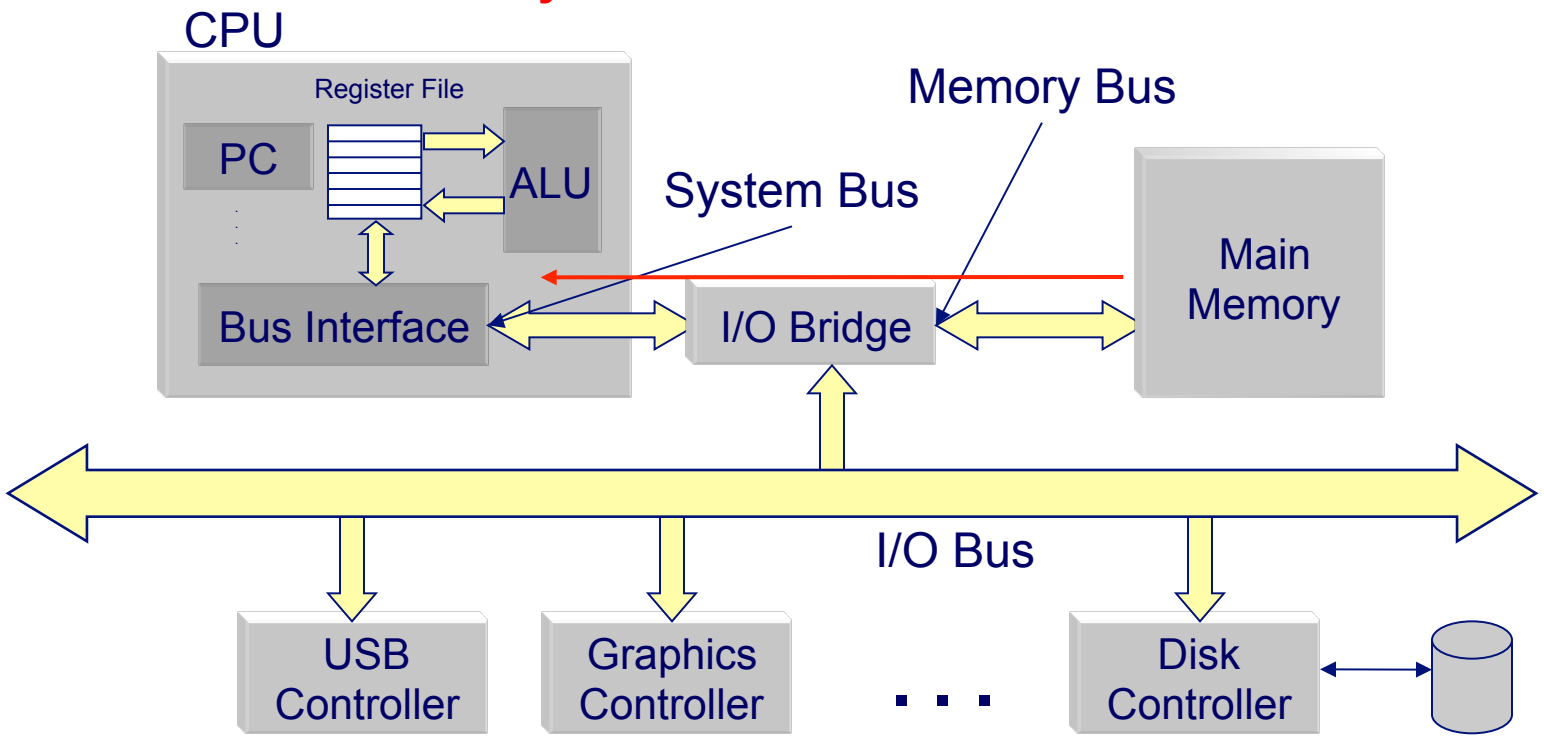
Summary using hello.c

5. Shell process creates new process through OS and initiates DMA of hello executable from disk to main memory



Summary using hello.c

6. CPU executes hello code from main memory



Summary using hello.c

7. CPU copies string “hello, world\n” from main memory to display