Computer Systems Organization

Outline

gcc – The Compiler Driver

cpp – The Preprocessor

The Virtual Address Space

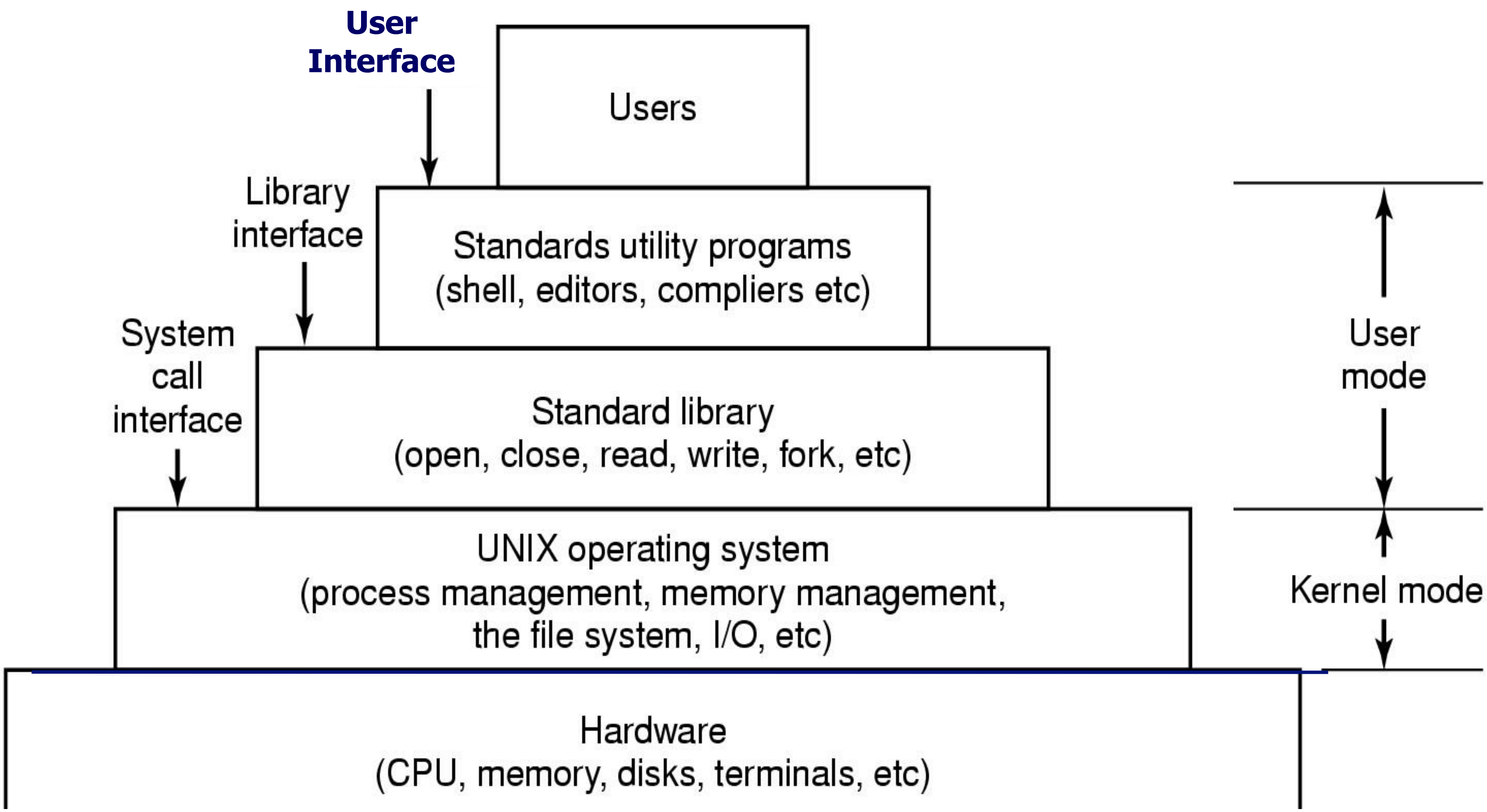
Linking

The Operating System

- The File System
- Processes
- The Memory Hierarchy

Program Execution

A software view



How it works

hello.c program

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

The gcc compilation system

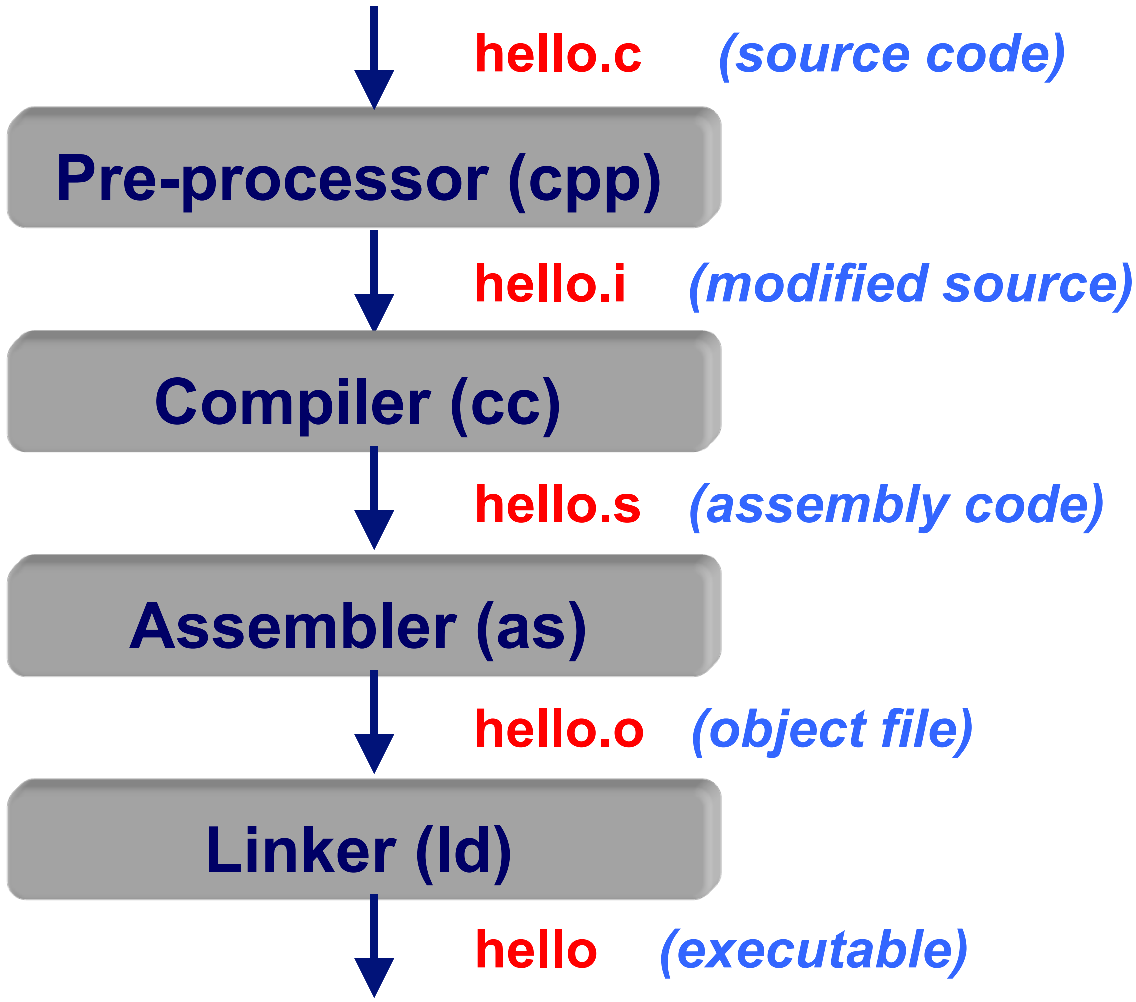
gcc is a ***compiler driver***

- gcc is a script program

gcc invokes the compilation phases

- Preprocessor
- Compiler
- Assembler
- Linker

The gcc compilation system



The Preprocessor: cpp

First step: gcc compiler driver invokes cpp

Output is expanded C source

cpp does text substitution

- **Converts the C source file to another C source file**
- **Expands**

  `#define`

  `#include`

  `#if…`
- **Output is another C source file**

The Preprocessor: cpp

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

Macros

Whitespace required

Defined constants:

```
#define MAXVAL   40000000
```

Whitespace forbidden

Macros:

```
#define MY_MULT(x,y)   ((x)*(y))
```

Input to cpp:

```
a = MY_MULT(b+c,d-foo(17));
```

Input to compiler:

```
a = ((b+c)*(d-foo(17)));
```

Macros - Why the parens?

Whitespace required

Defined constants:

```
#define MAXVAL   40000000
```

Whitespace forbidden

Macros:

```
#define MY_MULT(x,y)   ( x * y )
```

Input to cpp:

```
a = MY_MULT(b+c,d-foo(17));
```

Input to compiler:

```
a = ( b+c * d-foo(17) );
```

Macros – Just Textual Substitution

Macros:

```
#define WACKY_MAC(x,y,z)    x 4 y z ;
```

Input to cpp:

```
a = arr [ WACKY_MAC(b*,]+,c)
```

Perfectly Okay

Substituting:

```
a = arr [ x 4 y z ;
```

Input to compiler:

```
a = arr [ b* 4 ]+ c ;
a = arr[b*4] + c ;
```

No syntax error, after all!

Conditional Compilation

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 cc to generate assembly code**
  - **Translates C source code into assembly code.**
    - **Variables: mapped to memory locations and registers.**
    - **Logical and arithmetic operations: 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

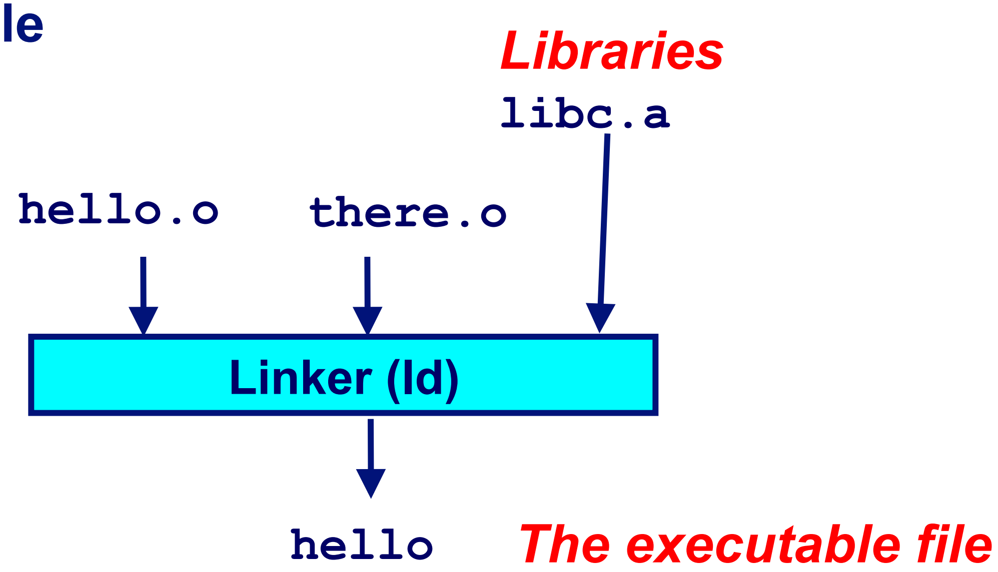
Finally, gcc compiler driver calls linker (ld) to generate executable

Combine:

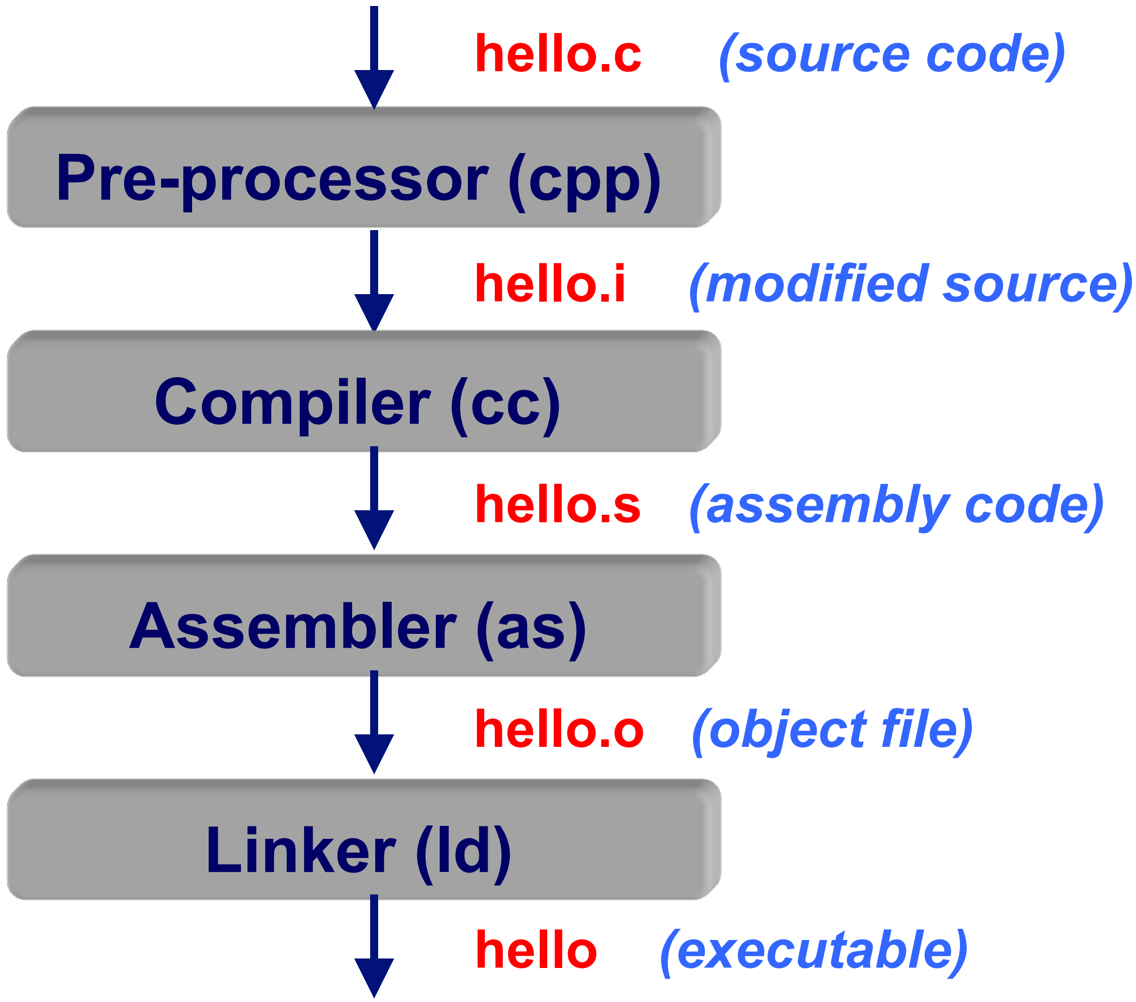
- **One or more object files.**
- **Functions from (static) libraries, as needed.**

Create:

- **The executable file**



The gcc compilation system



GCC variations

Stop after the preprocessor

```
gcc -E hello.c
```

Stop after the C compiler

```
gcc -S hello.c
```

Stop after the assembler

```
gcc -c hello.c
```

Go all the way

```
gcc hello.c -o greeting
```

default is a.out

GCC variations

Stop after the preprocessor

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

default is hello.i

Stop after the C compiler

```
gcc -S hello.c -o greeting.s
```

default is hello.s

Stop after the assembler

```
gcc -c hello.c -o greeting.o
```

default is hello.o

Go all the way

```
gcc hello.c -o greeting
```

default is a.out

GCC variations

Stop after the preprocessor

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

default is hello.i

Stop after the C compiler

```
gcc -S hello.c -o greeting.s
```

default is hello.s

Stop after the assembler

```
gcc -c hello.c -o greeting.o
```

default is hello.o

Go all the way

```
gcc hello.c -o greeting
```

default is a.out

The extension tells where to start

`gcc hello.c`	*Begin with the preprocessor*
`gcc hello.i`	*Begin with the compiler*
`gcc hello.s`	*Begin with the assembler*
`gcc hello.o`	*Begin with the linking*

GCC variations

Print all warnings:

```
gcc -Wall hello.c
```

Produce an assembler listing & stop:

```
gcc -Wa,-alh hello.c -c
```

Optimize the code:

```
gcc -O1 hello.c
```

Include info for gdb and don't optimize too much:

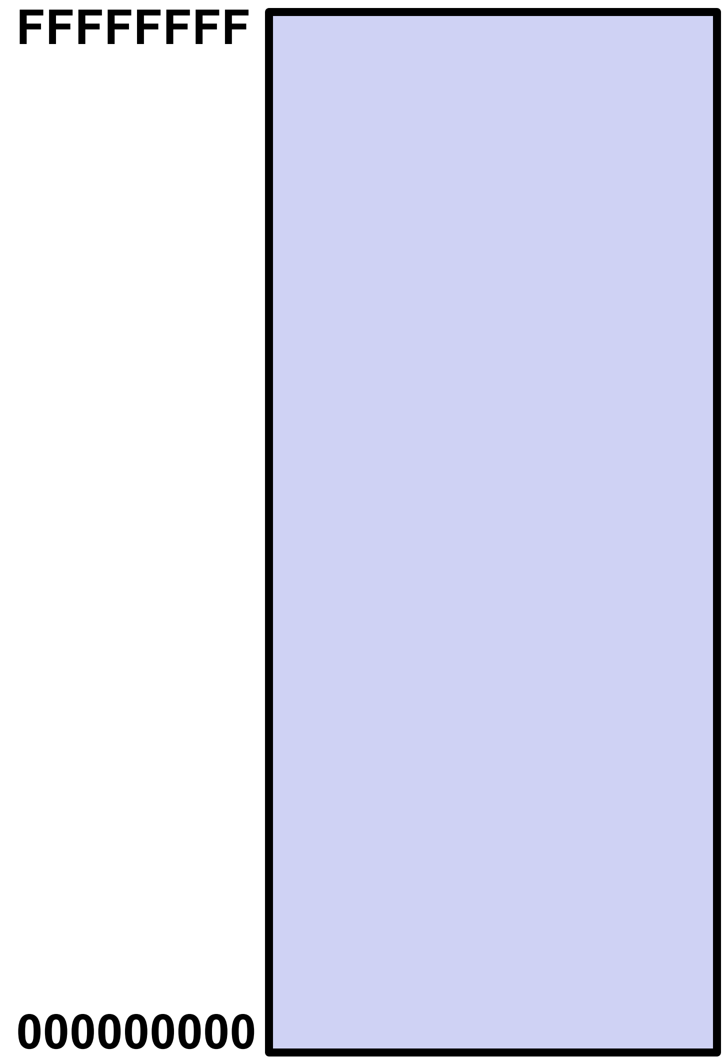
```
gcc -g -Og hello.c
```

Compile for 32-bits or 64-bits:

```
gcc -m32 hello.c
gcc -m64 hello.c
```

The Virtual Address Space

What goes into memory?

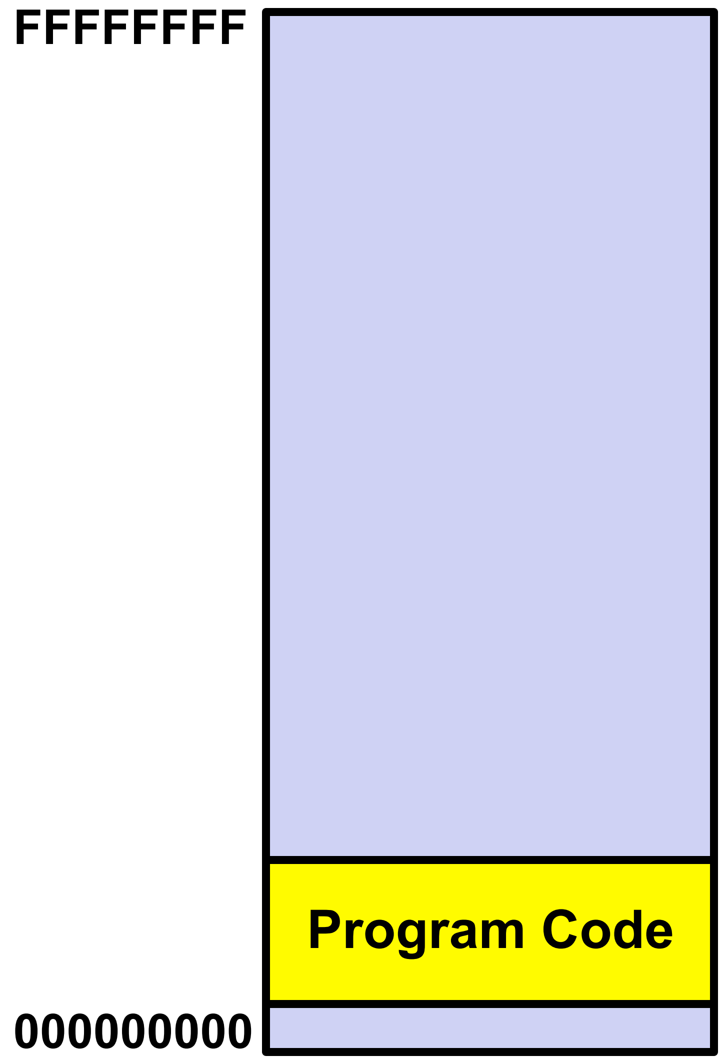


The Virtual Address Space

What goes into memory?

Program Code

machine code instructions



The Virtual Address Space

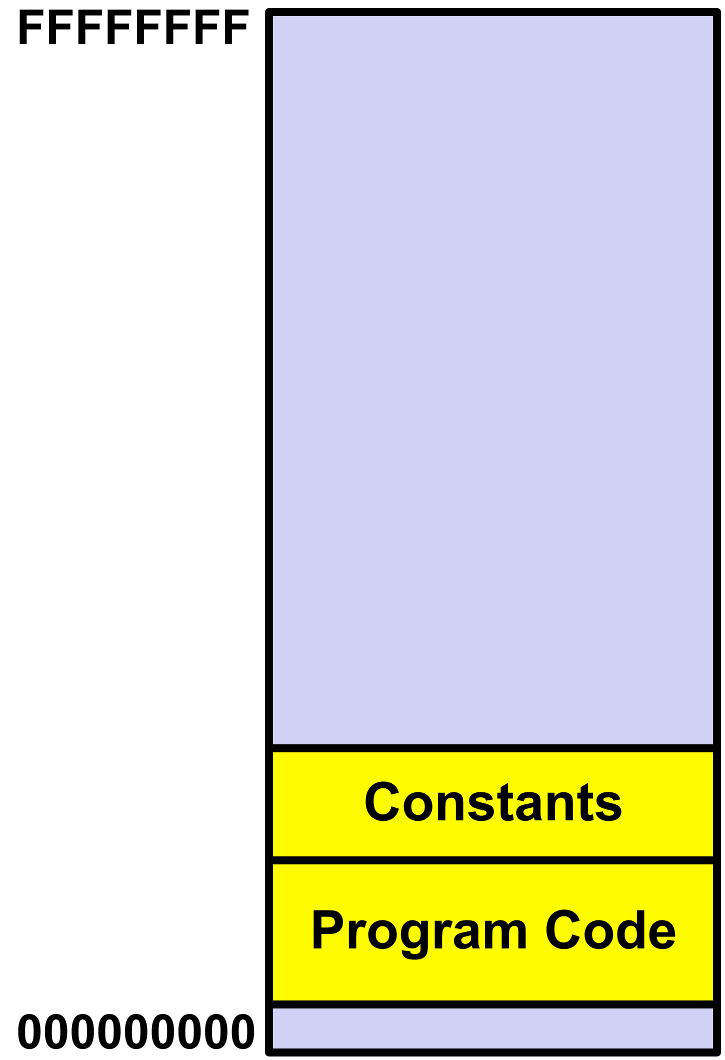
What goes into memory?

Program Code

machine code instructions

Constants

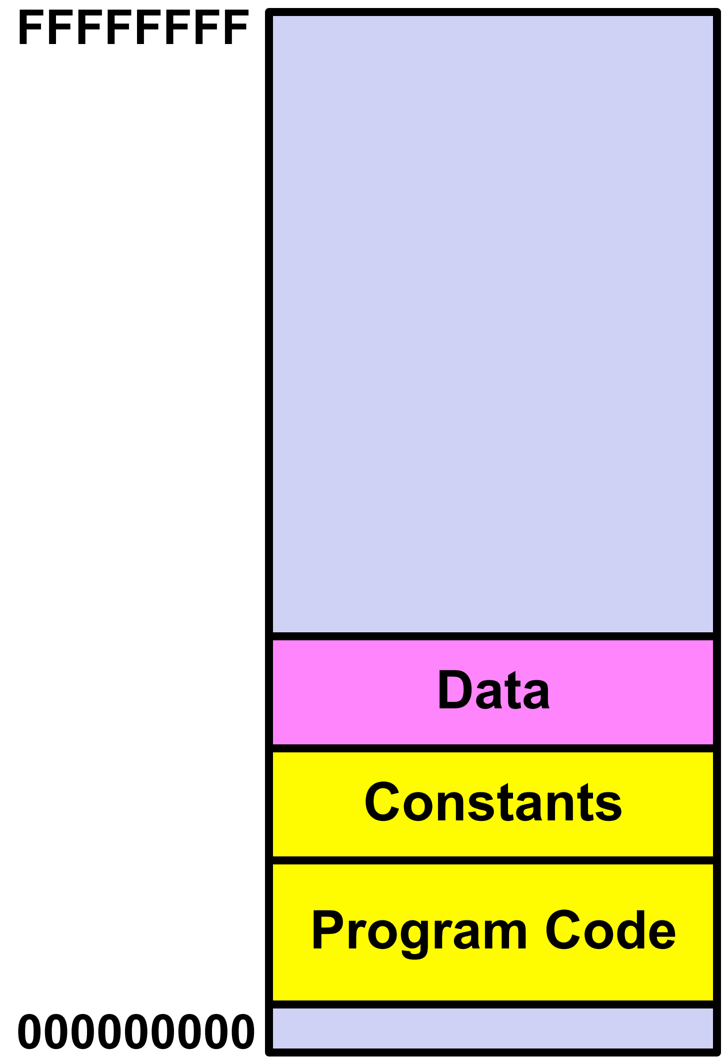
never modified



The Virtual Address Space

What goes into memory?

- **Program Code**
  - machine code instructions
- **Constants**
  - never modified
- **Data**
  - global variables



The Virtual Address Space

What goes into memory?

Program Code

- machine code instructions

Constants

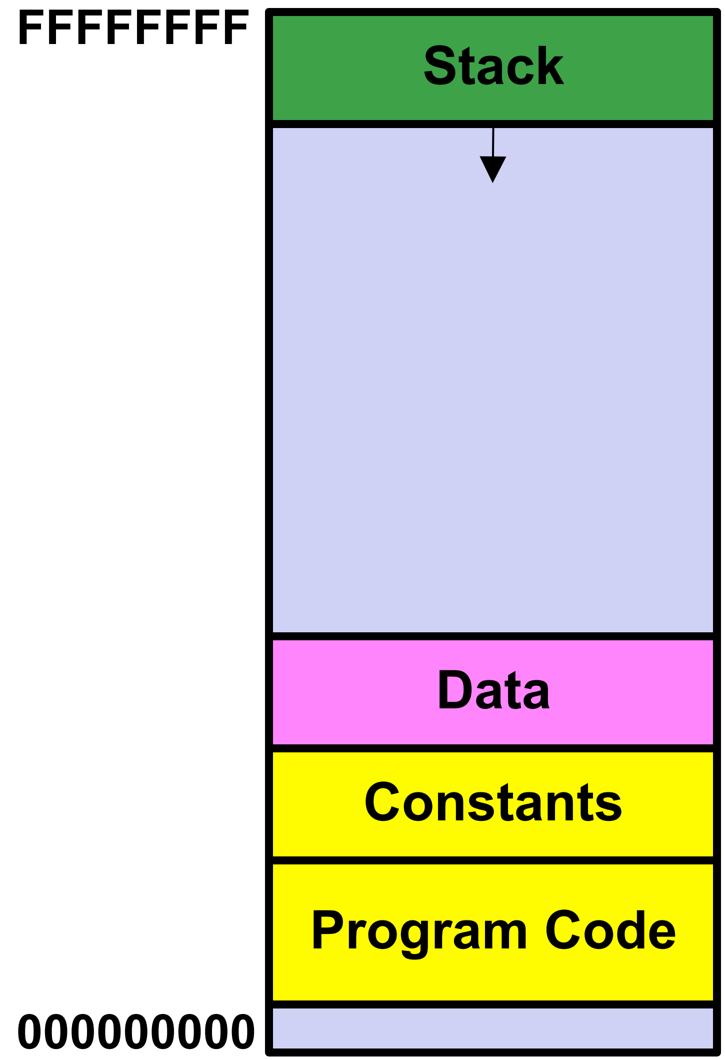
- never modified

Data

- global variables

Stack

- to hold stack frames



The Virtual Address Space

What goes into memory?

Program Code

- machine code instructions

Constants

- never modified

Data

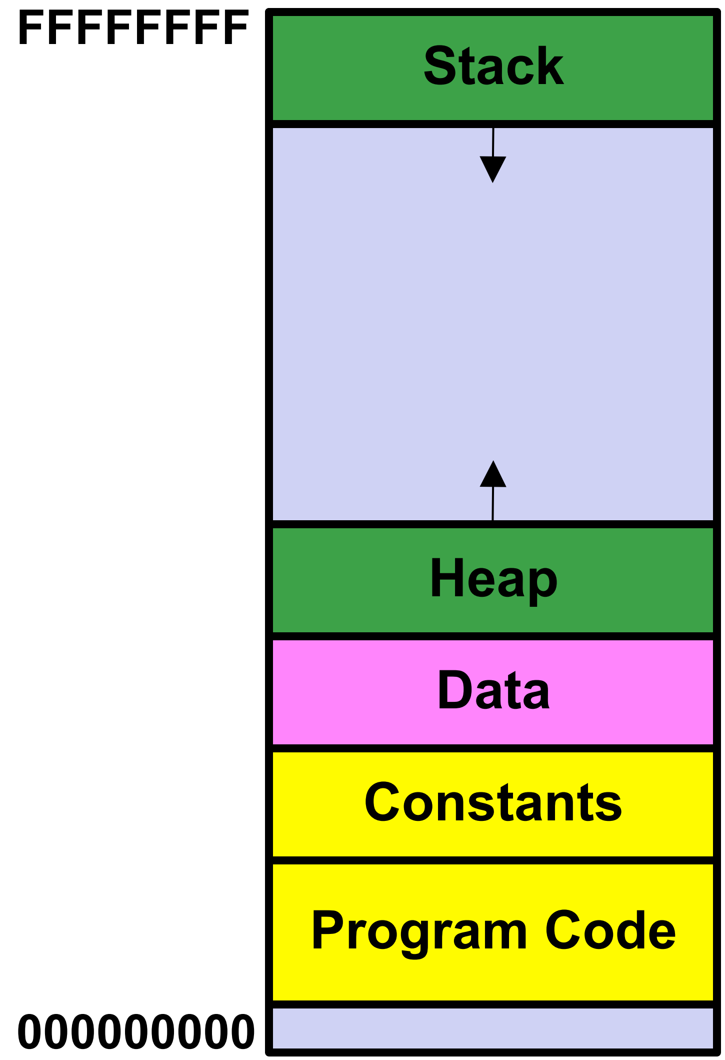
- global variables

Stack

- to hold stack frames

Heap

- memory allocations



The Virtual Address Space

What goes into memory?

Program Code

- machine code instructions

Constants

- never modified

Data

- global variables

Stack

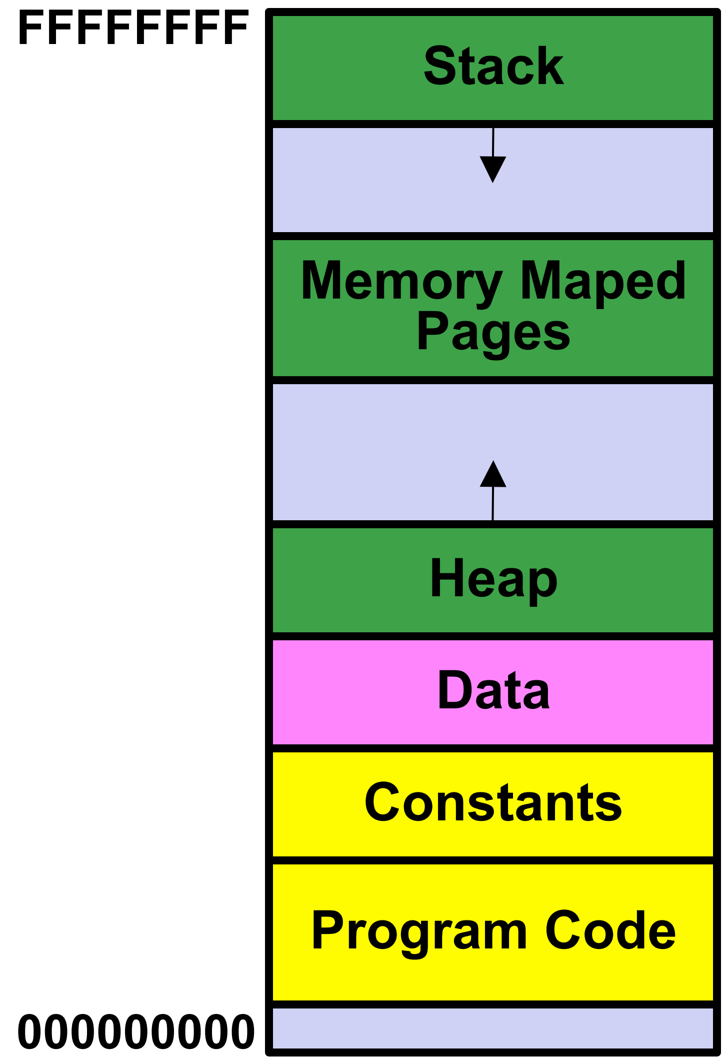
- to hold stack frames

Heap

- memory allocations

Other Stuff

- memory-mapped pages



The Virtual Address Space

What goes into memory?

Program Code

- machine code instructions

Constants

- never modified

Data

- global variables

Stack

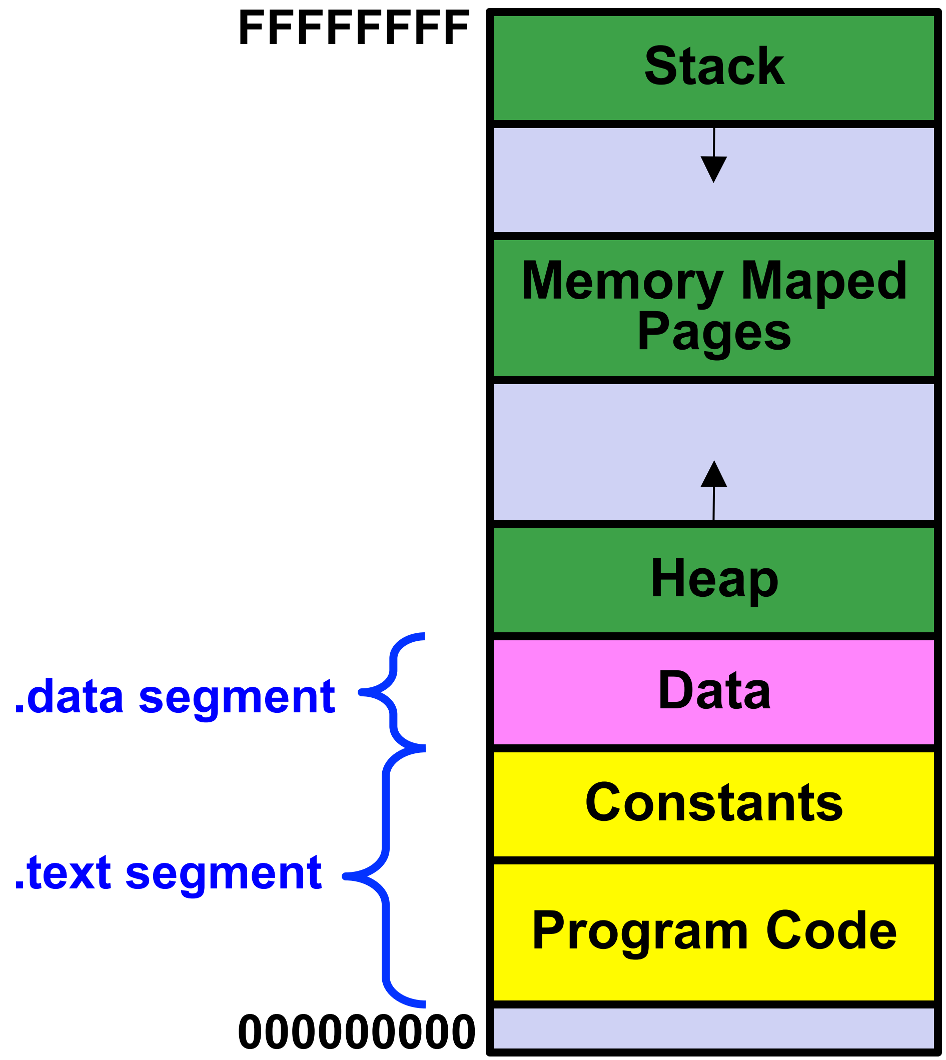
- to hold stack frames

Heap

- memory allocations

Other Stuff

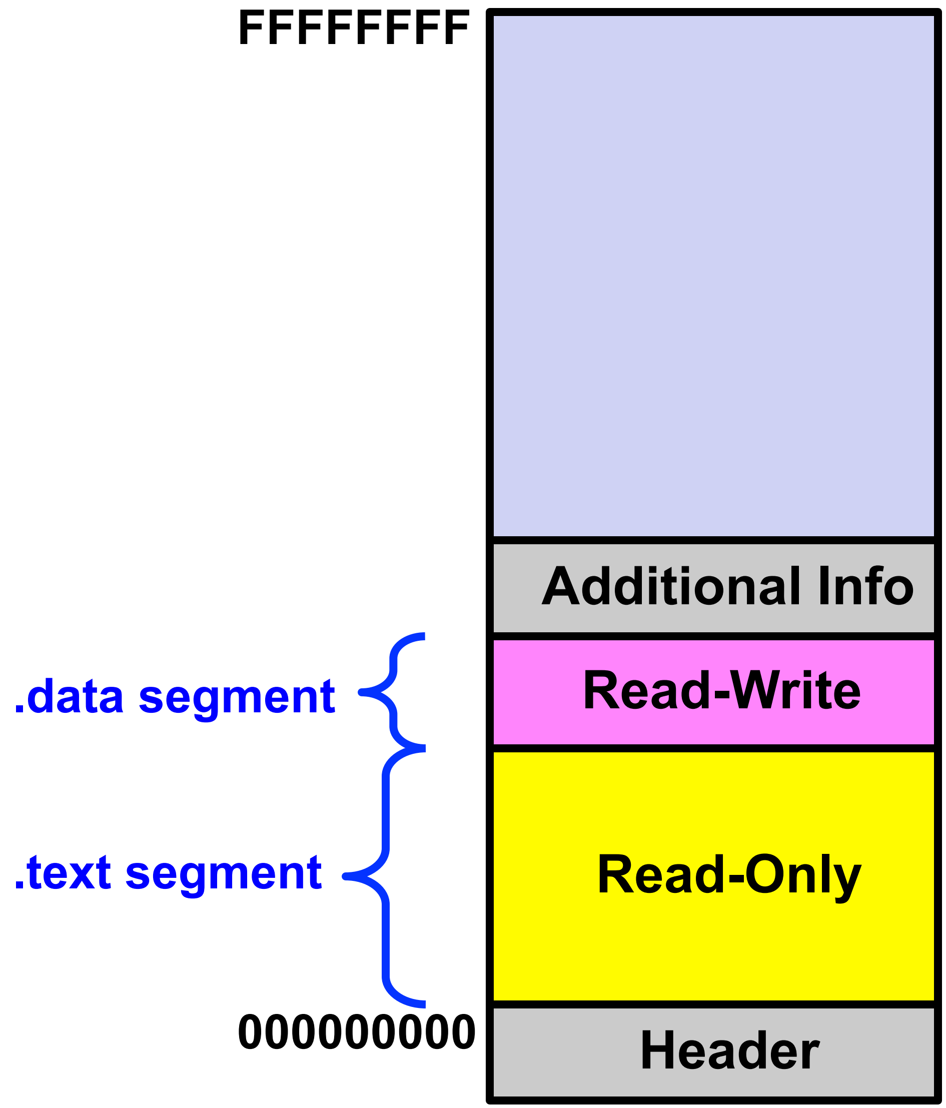
- memory-mapped pages



The Executable File

What is in an ELF file?

- .text segment
  - The read-only bytes
- .data segment
  - The read-write bytes
- Header info
  - Where to put the segments
  - Start address
- Additional Info
  - Info for gdb (optional)



Why Link?

- **Program is composed of smaller source files,**
  rather than one monolithic mass.
- **Build one big library containing all common functions**
  libc.a (Standard C Library); libm.a (Math Library)
- **Quicker Program Build**
  Change one source file, compile, and then relink.
  No need to recompile other source files.
- **Programs contain only the functions they actually use**
  Smaller executable files; less runtime memory usage
- **Many useful functions collectd into a single library file**
  The library is used by all programs

The linking process (ld)

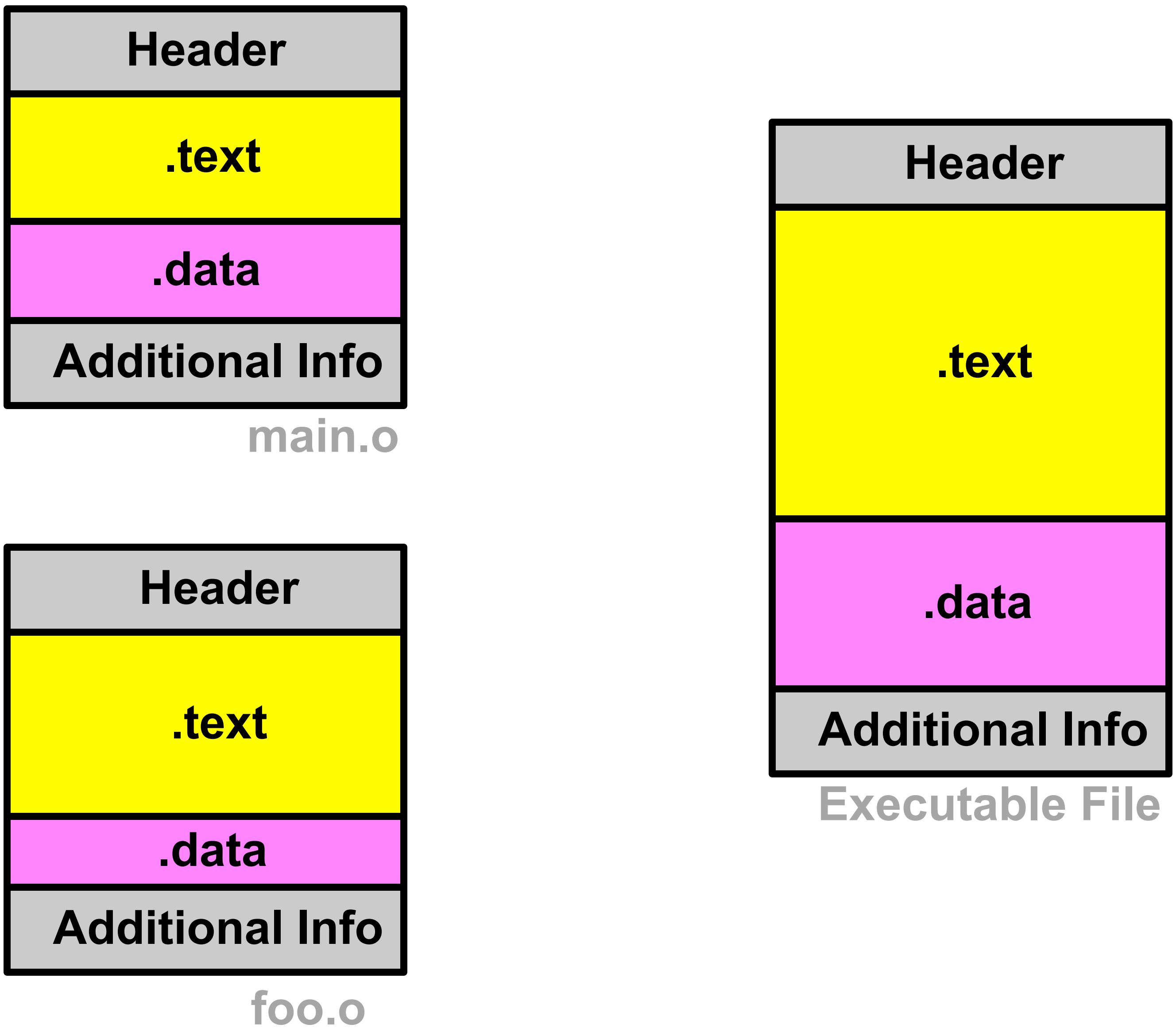
Merges multiple relocatable (.o) object files into a single executable program.

Resolves external references

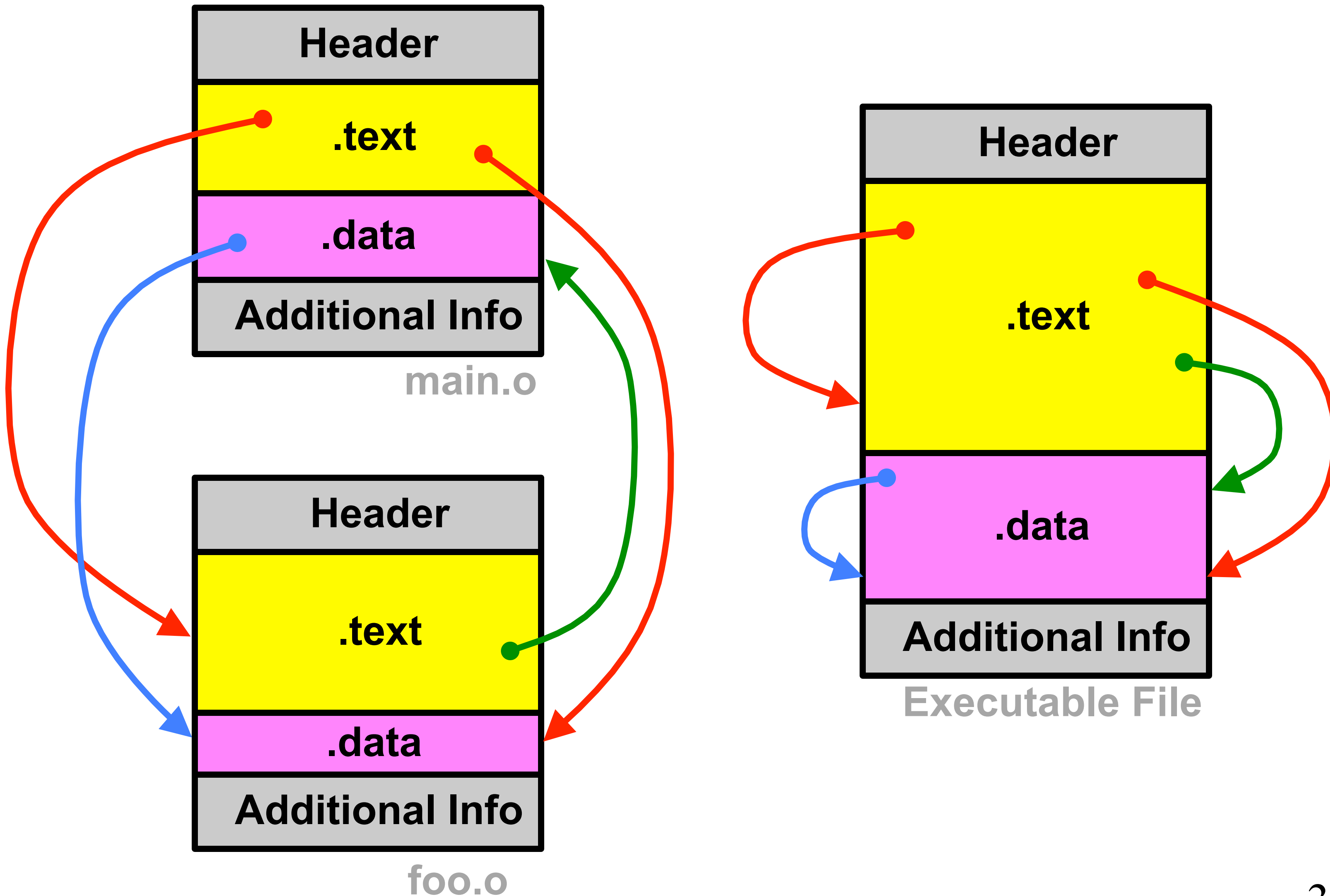
External reference: reference to a symbol defined in another object file.

Ensures each symbol is uniquely defined

The Linking Process

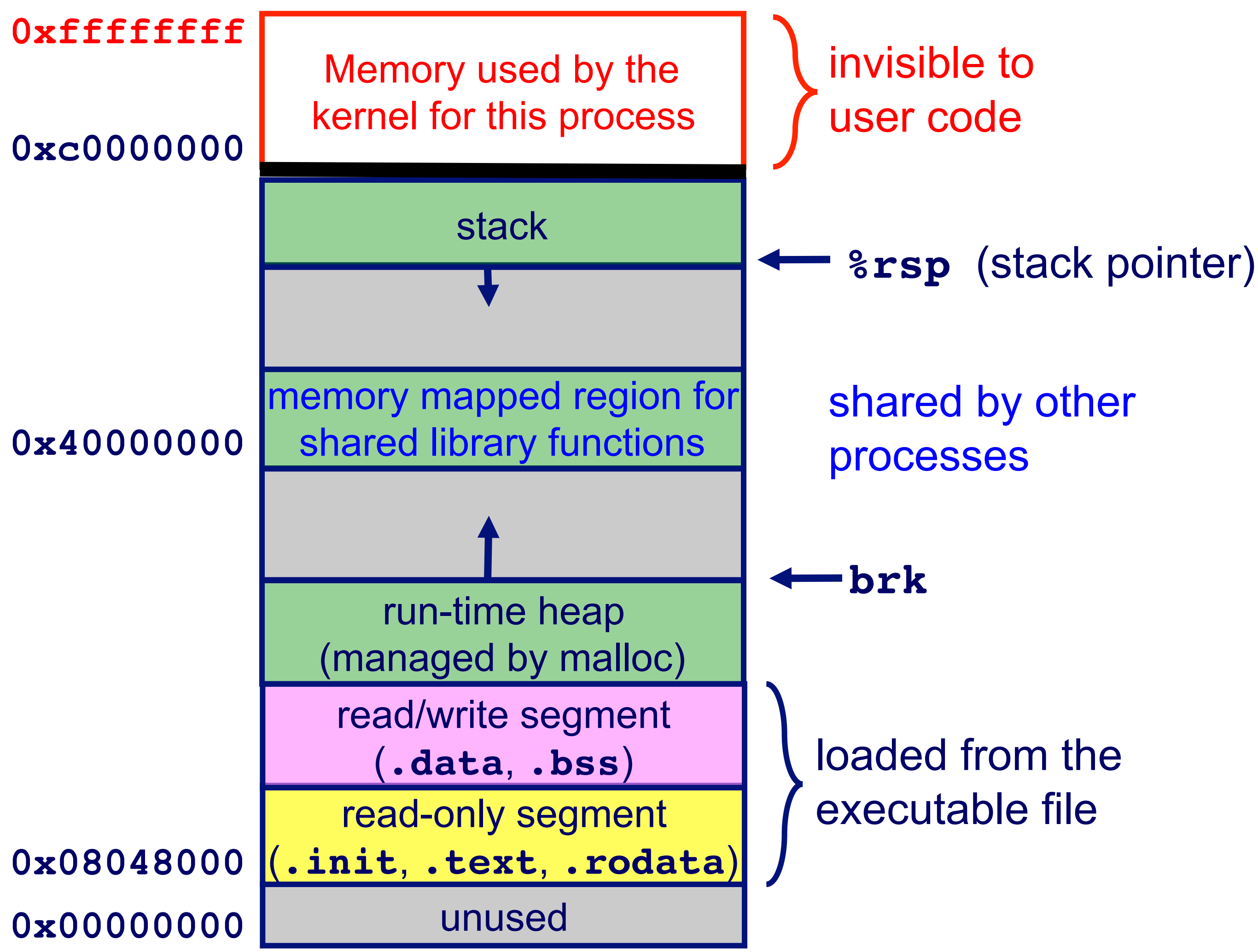


Resolving External References



Example Virtual Address Space

This is what the program "sees"



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
  - The function is loaded at run-time by system loader upon execution

Three Kinds of Object Files (Modules)

Relocatable object file (`.o` file)

- Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each `.o` file is produced from exactly one source (`.c`) file

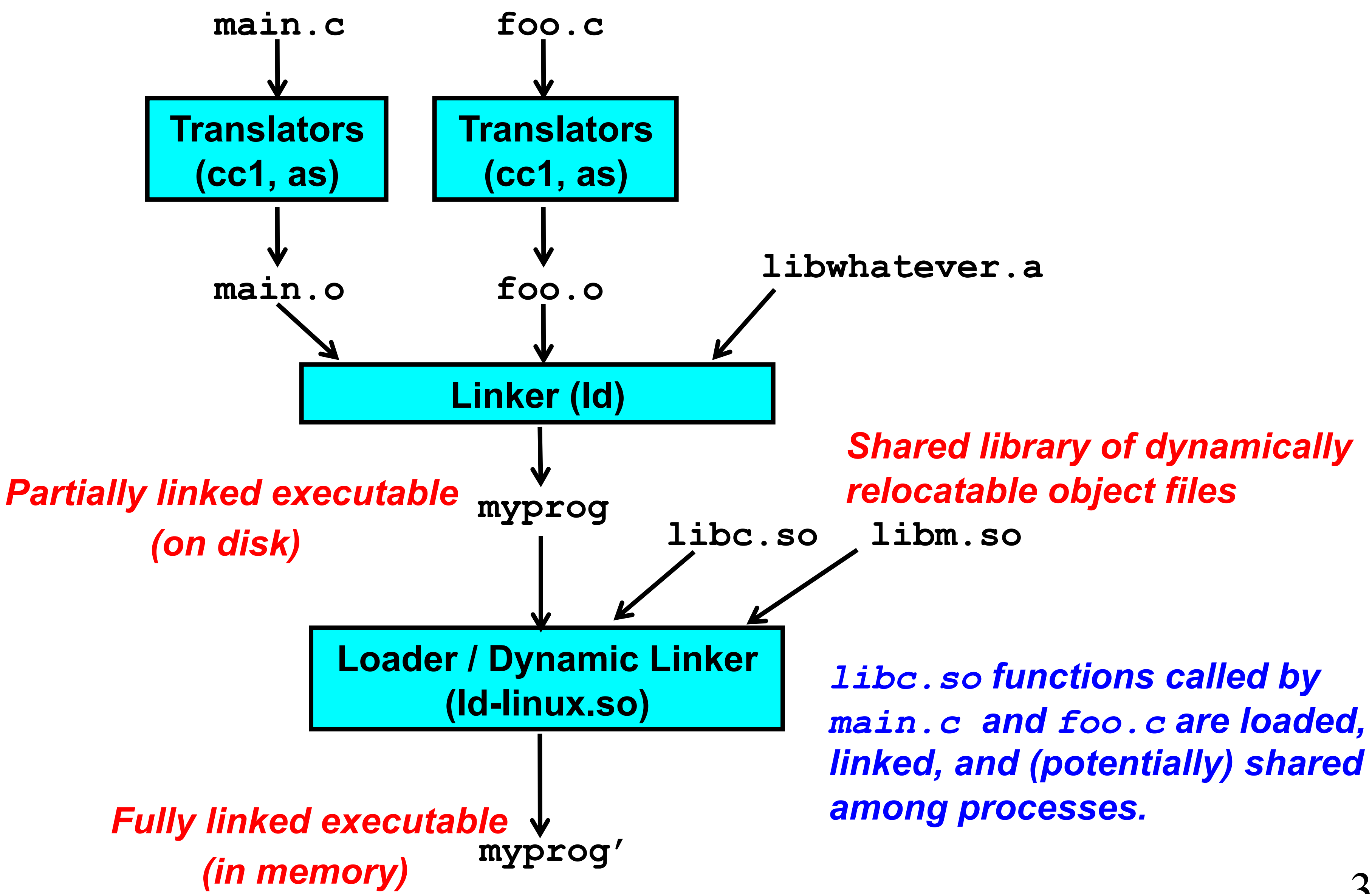
Executable object file (`a.out` file)

- Contains code and data in a form that can be copied directly into memory and then executed.

Shared object file (`.so` file)

- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called ***Dynamic Link Libraries*** **(DLLs)** by Windows

The Complete Picture



The Operating System

Programs run on top of operating system

OS 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 → an abstraction of storage devices
- System Calls → an abstraction for OS services
- Virtual memory→ a uniform memory space for each process
    - Gives the illusion that each process has entire memory space
- A process → an abstraction for a virtual computer
    - "Timeslicing" – Dividing CPU time into pieces
    - Each program gets a slice of time
    - All programs make progress, but only when they "have" the CPU
    - Each program must wait when other programs are executing

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.

Higher levels of structure are an application concept, not an operating system concept

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 recipe for cookies
- A **process** is an instance of that program loaded in memory and running
  - The act of baking a particular batch of cookies

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 a virtual address space

System loader reads executable from file system and loads into memory

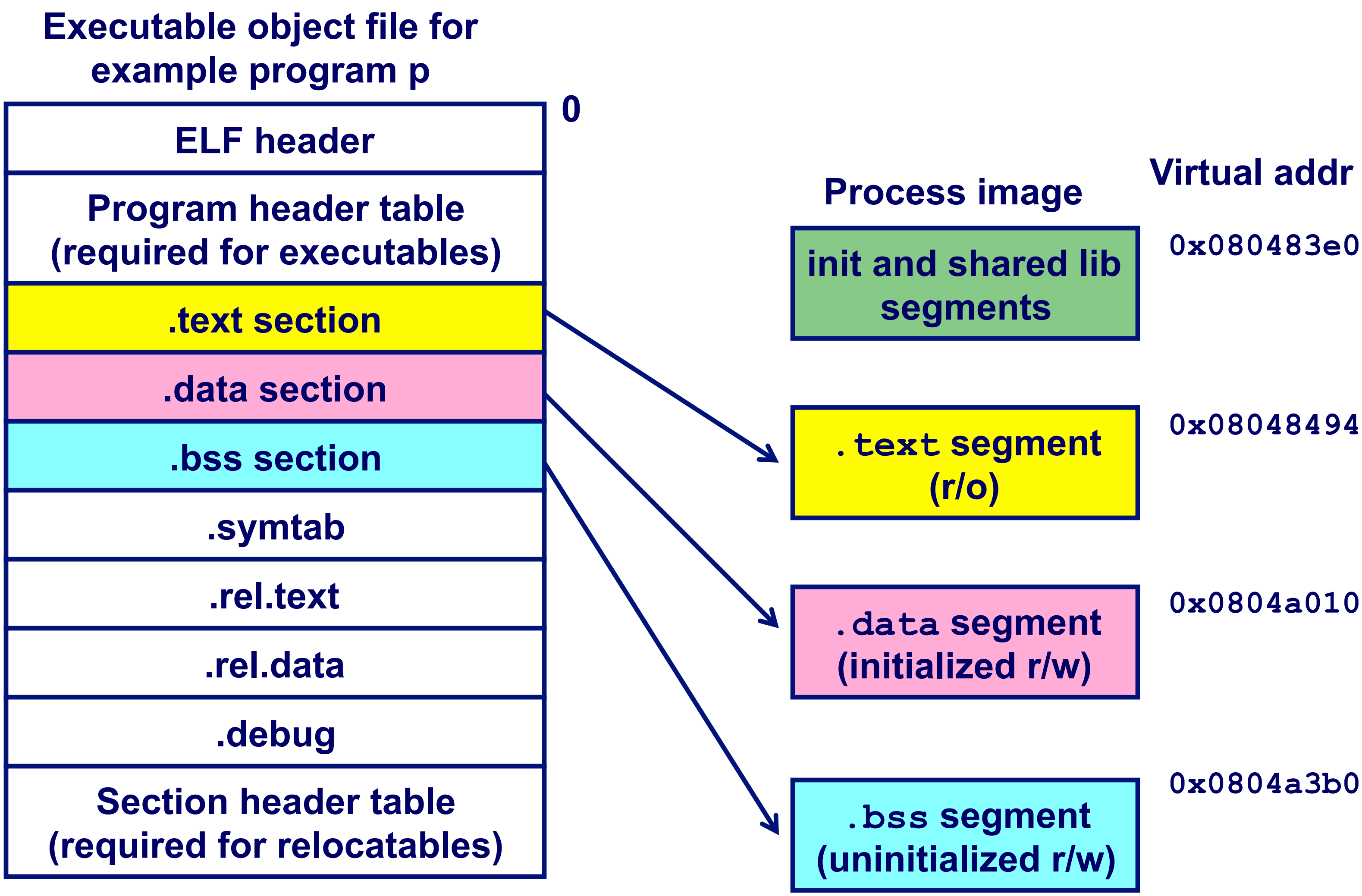
- Already includes statically linked library functions

System loader loads dynamic shared objects/libraries into memory

Then it starts the thread of execution running

- Registers & Stack are initialized
- The thread is scheduled (Jump to the starting addresses)

Loading Executable Binaries



Where are programs loaded in memory?

To start with, imagine a primitive operating system.

- Only one process at a time
- Physical memory addresses go from zero to N.

The problem of loading is simple

- Load the program 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

Where are programs loaded, cont'd

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

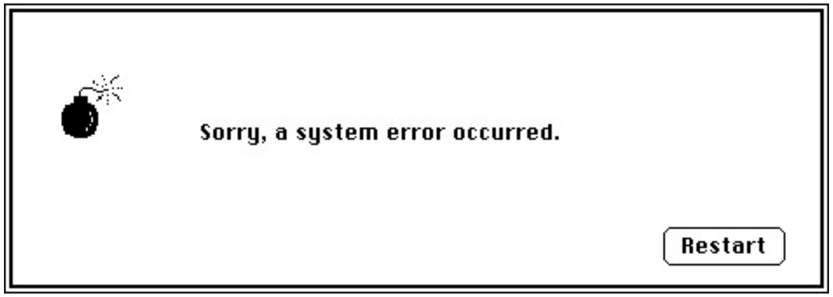
- Physical memory space, from zero to N.
- Applications share space
- Memory allocated at load time in unused space
- Linker does not know where the program will be loaded
- Binds together all the modules, but keeps them relocatable

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.

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

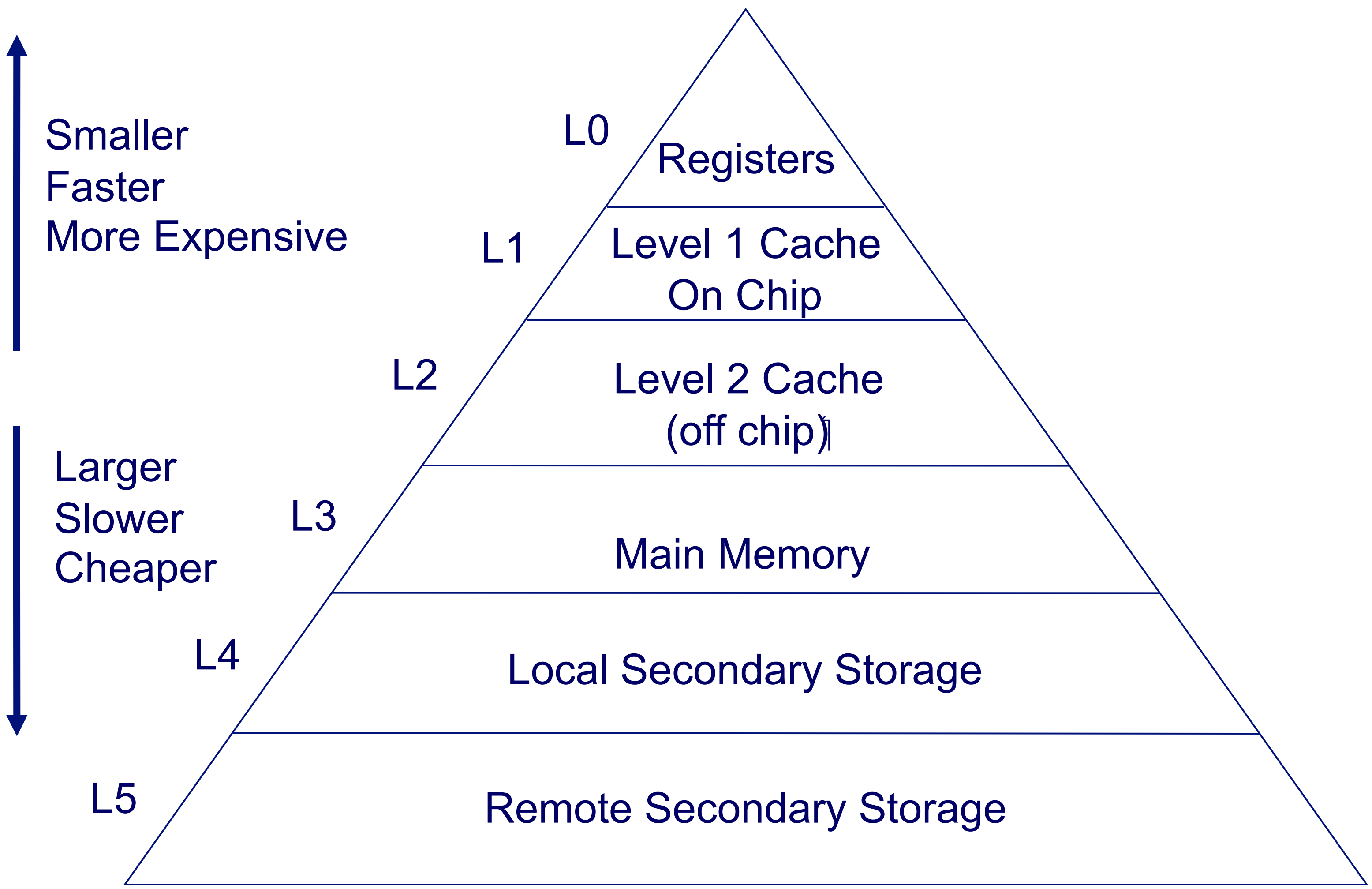
Today

- 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

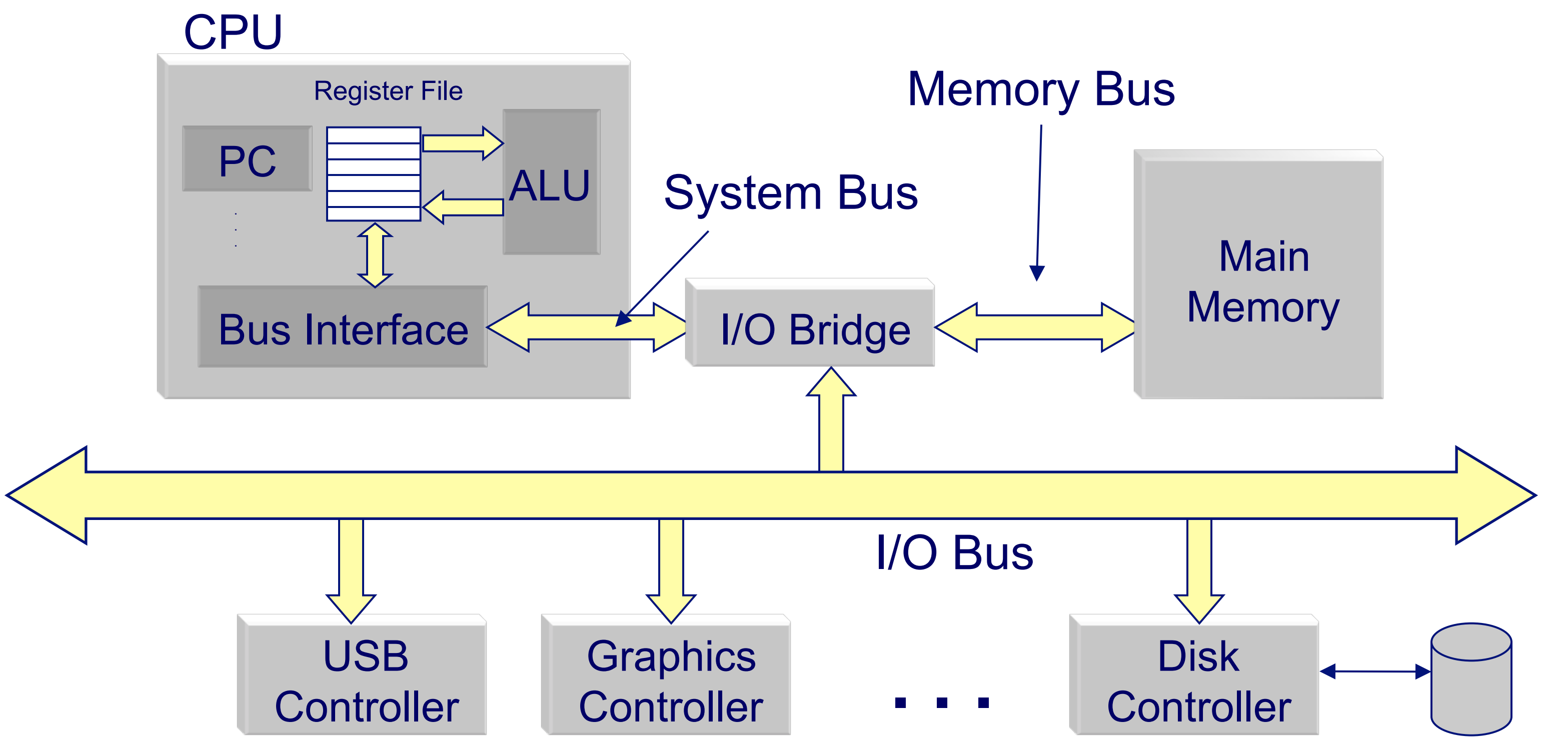
- How do you know what you'll need in the future?
- Locality
- L1, L2, L3 caches

The memory heirarchy



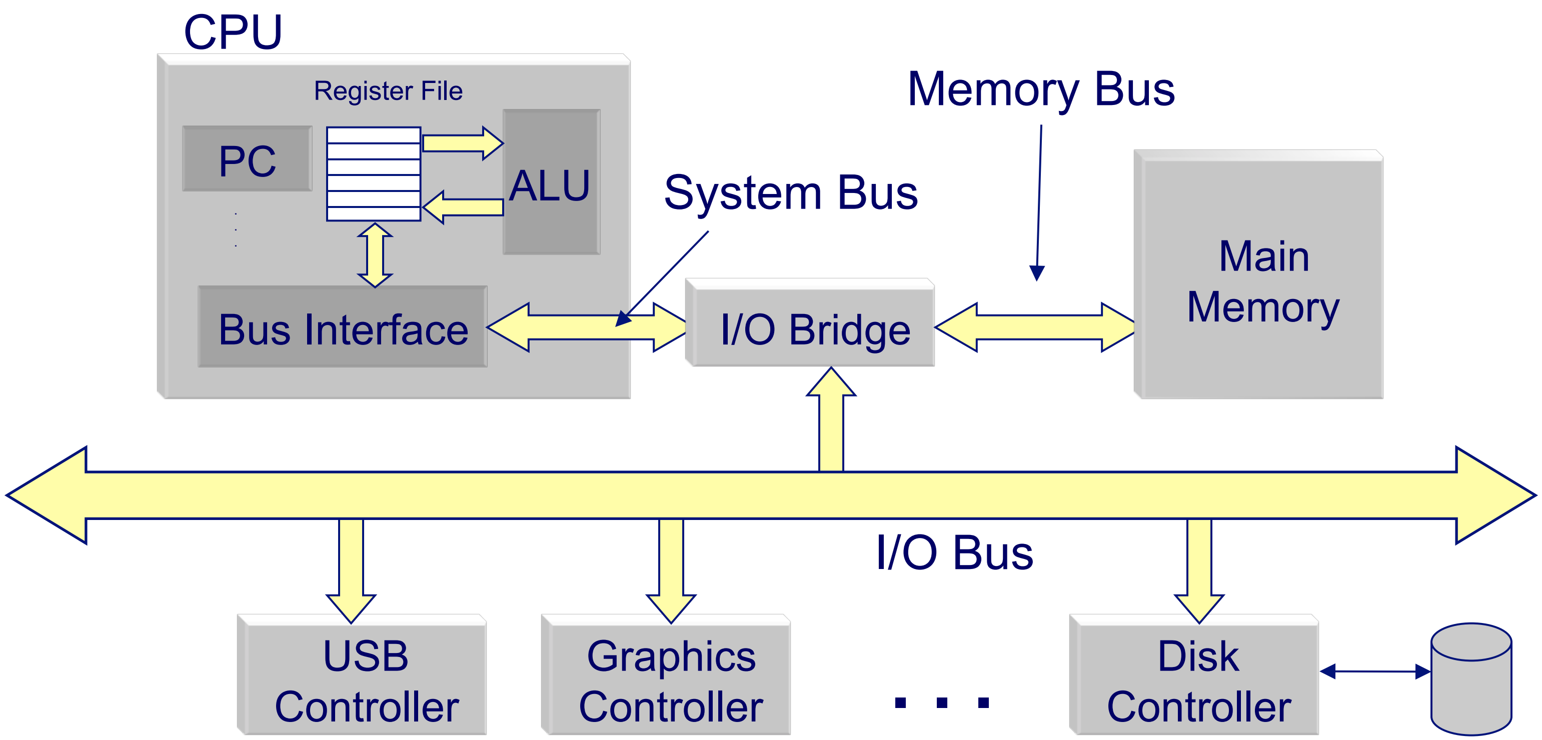
Hardware organization

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



Summary using hello.c

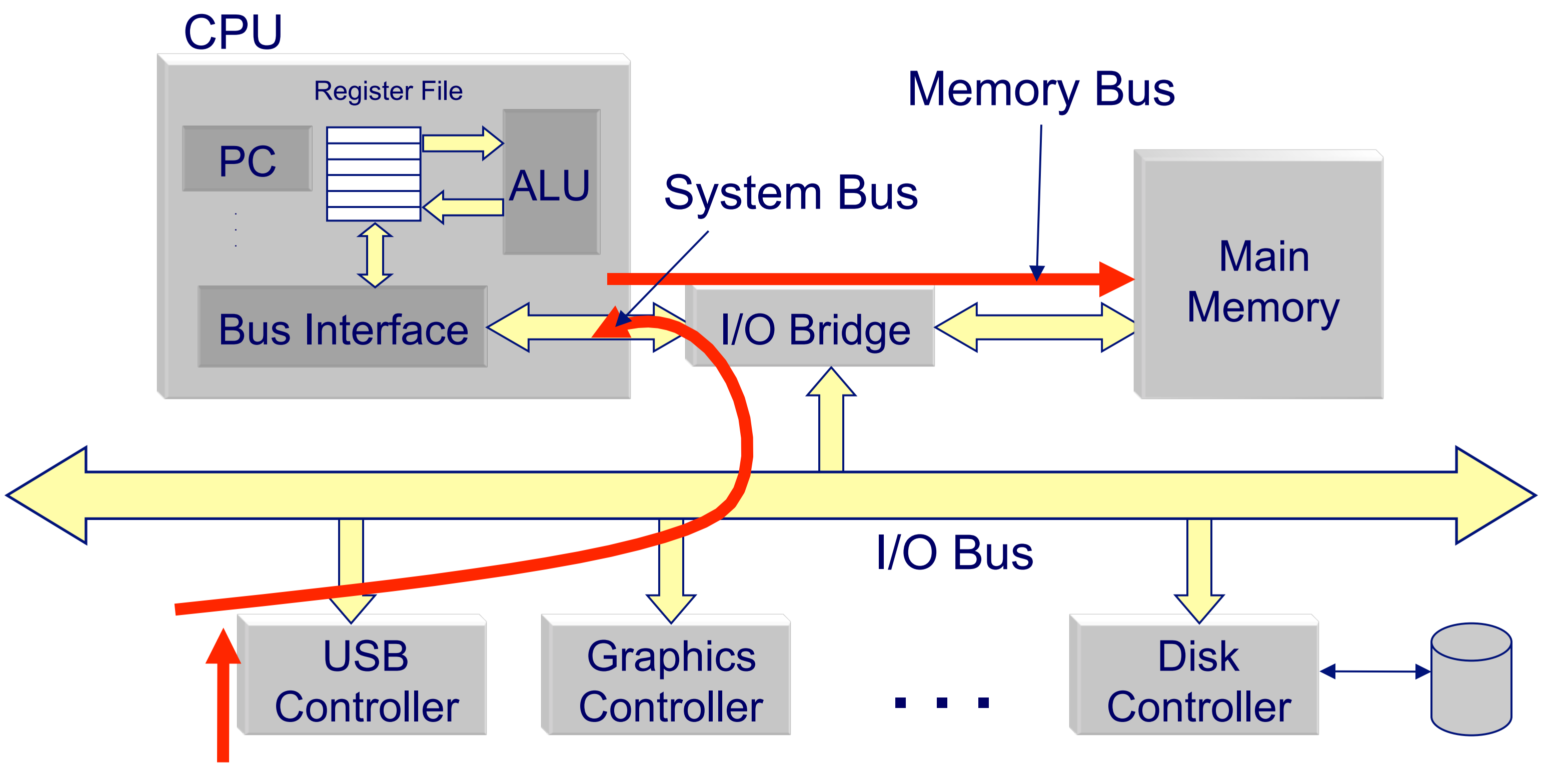
1. Shell process running, waiting for input



Summary using hello.c

3. Command read into registers

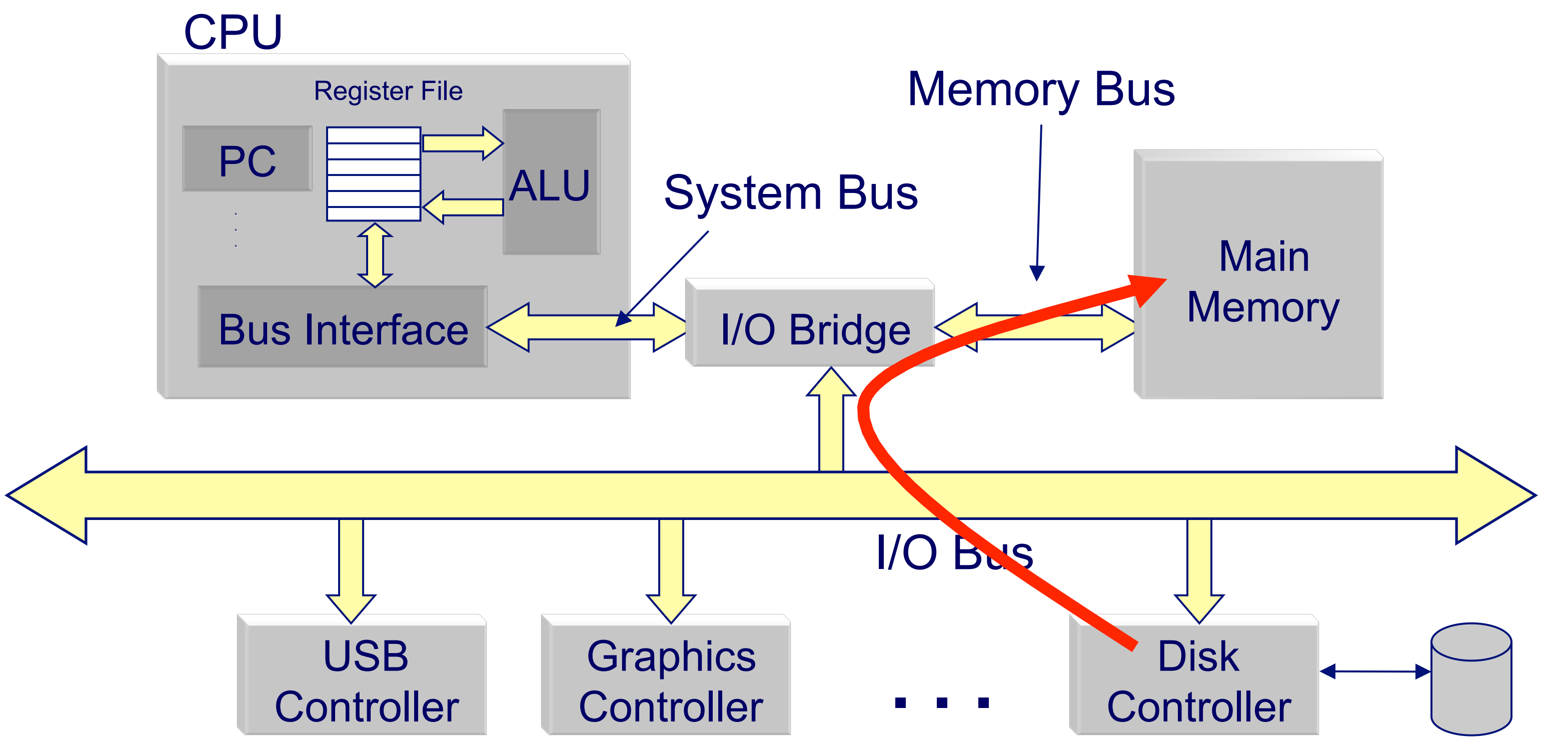
4. Before sent to main memory before being read by shell process



2. User types ./hello

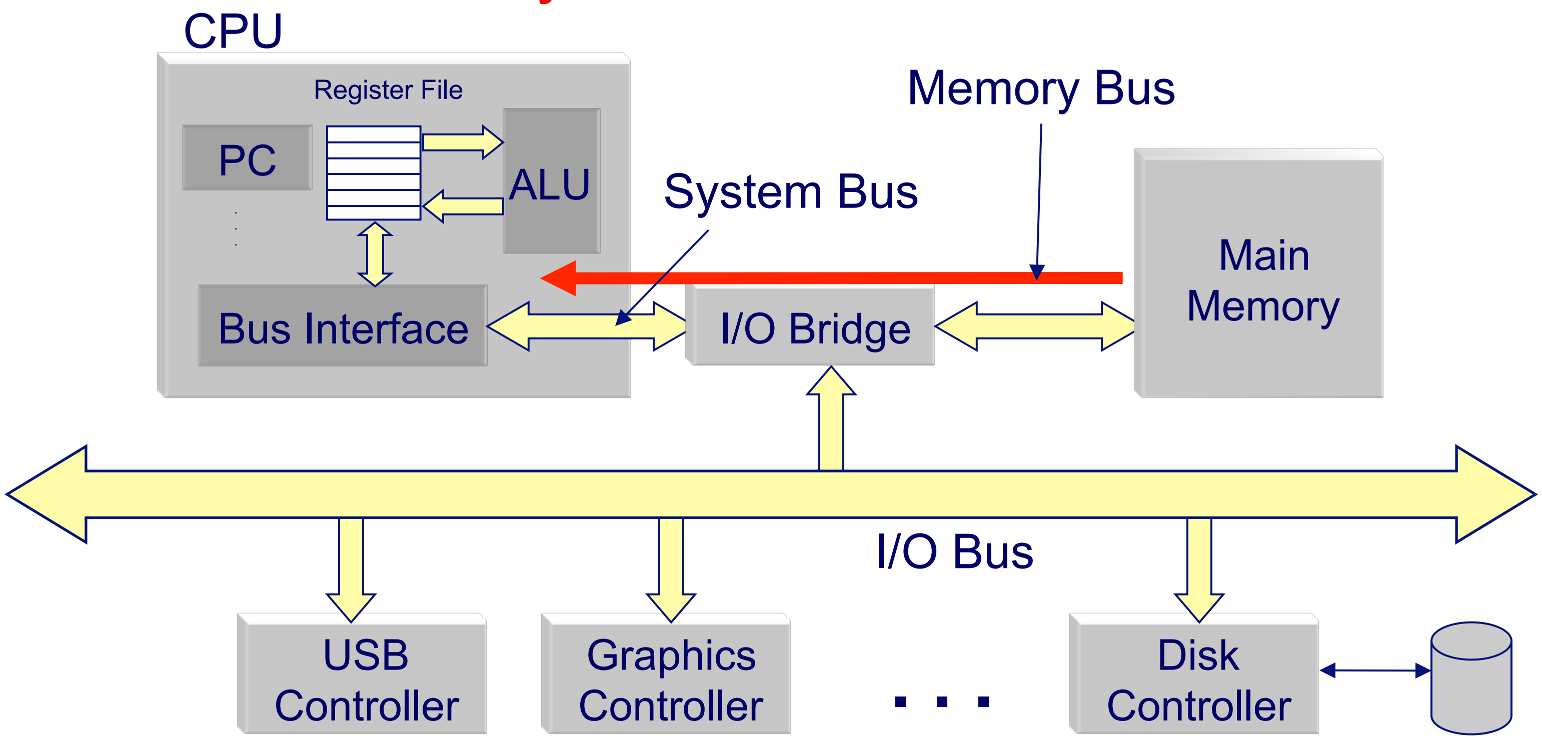
Summary using hello.c

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