CSE251: System Programming

8. ELF & Loading & Memory Layout

Seongil Wi

Department of Computer Science and Engineering

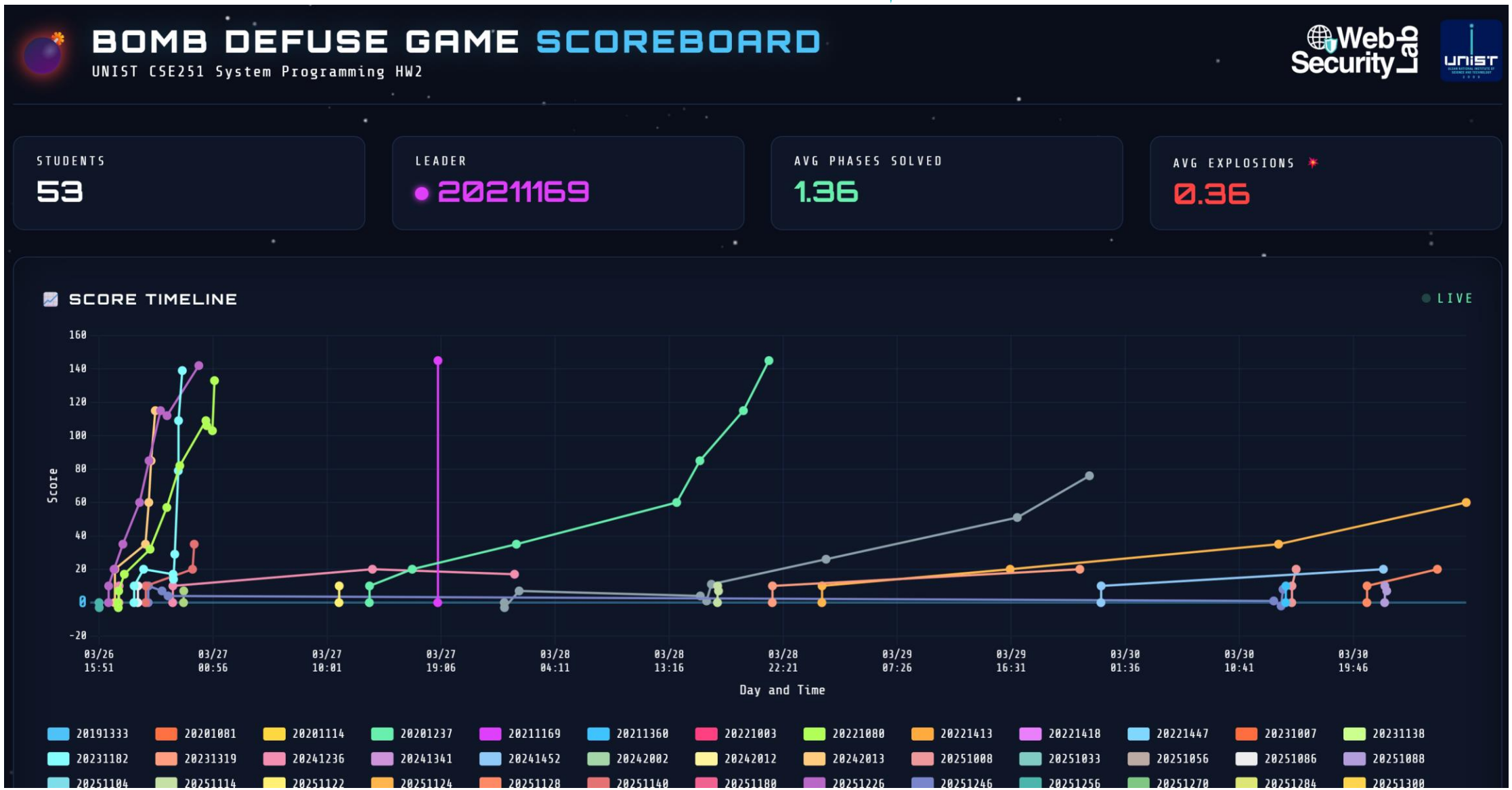
HW2

- Due: Apr. 9, 11:59PM

- Your goal is to defuse your bomb
  - You will earn points for each phase you defuse, with a maximum total of 145 points
  - Each time the bomb explodes, you will lose 3 points (Only to the first 20 explosions)

- You should submit your report via BlackBoard

HW2


BOMB DEFUSE GAME SCOREBOARD
UNIST CSE251 System Programming HW2
Web Security Lab
UNIST
STUDENTS
53
LEADER
20211169
AVG PHASES SOLVED
1.36
AVG EXPLOSIONS
0.36
SCORE TIMELINE
LIVE
Score
160
140
120
100
80
60
40
20
0
-20
03/26 15:51
03/27 00:56
03/27 10:01
03/27 19:06
03/28 04:11
03/28 13:16
03/28 22:21
03/29 07:26
03/29 16:31
03/30 01:36
03/30 10:41
03/30 19:46
Day and Time
20191333
20201081
20201114
20201237
20211169
20211360
20221003
20221080
20221413
20221418
20221447
20231007
20231138
20231182
20231319
20241236
20241341
20241452
20242002
20242012
20242013
20251008
20251033
20251056
20251086
20251088
20251104
20251114
20251122
20251124
20251128
20251140
20251180
20251226
20251246
20251256
20251270
20251284
20251300

Congratulations!

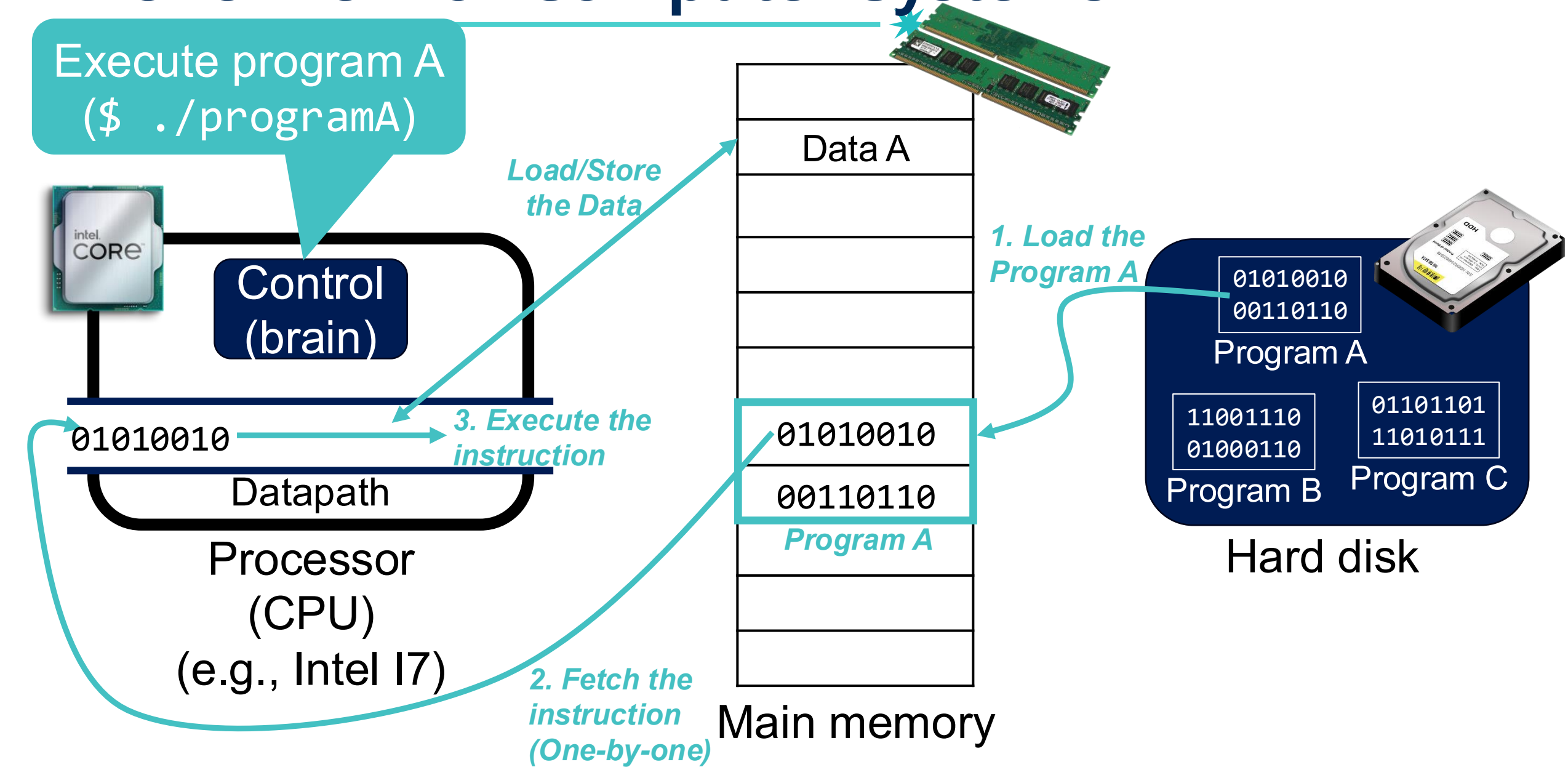
- The three students who quickly defuse all phases
  - 20251226, # of explosions 💥 : 1
  - 20231182, # of explosions 💥 : 2
  - 20221080, # of explosions 💥 : 4

- The three students with the highest final scores
  - 20211169, # of explosions 💥 : 0
  - 20201237, # of explosions 💥 : 0
  - **[It has not been decided yet. Who will take this place?]**

Regarding Cheating

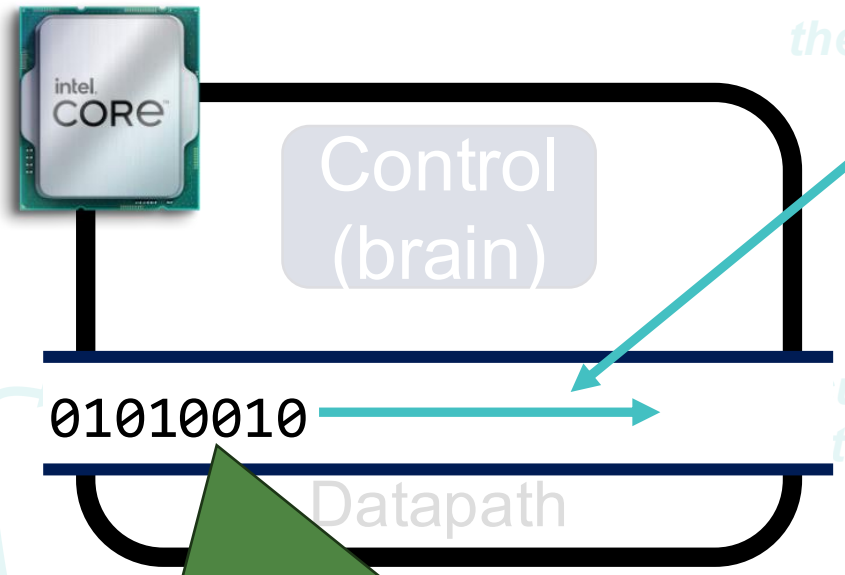
- Our system logs your full activity history, including shell commands and GDB commands.
- You are expected to act with integrity and refrain from any form of cheating.

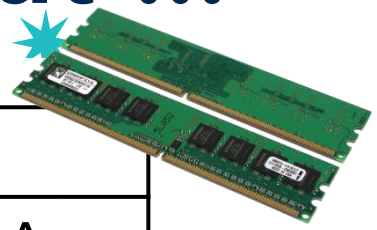
Overview of Computer Systems

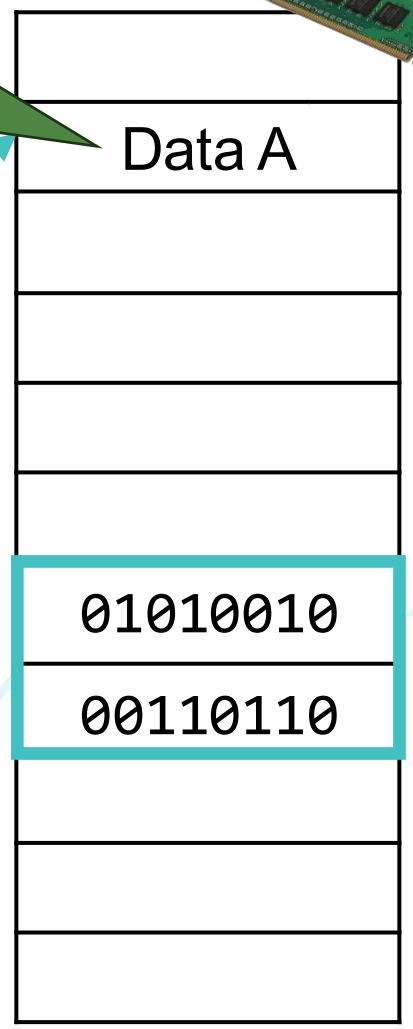


So Far, We Learned About …

How data (bit, byte, int, …) is represented in memory?







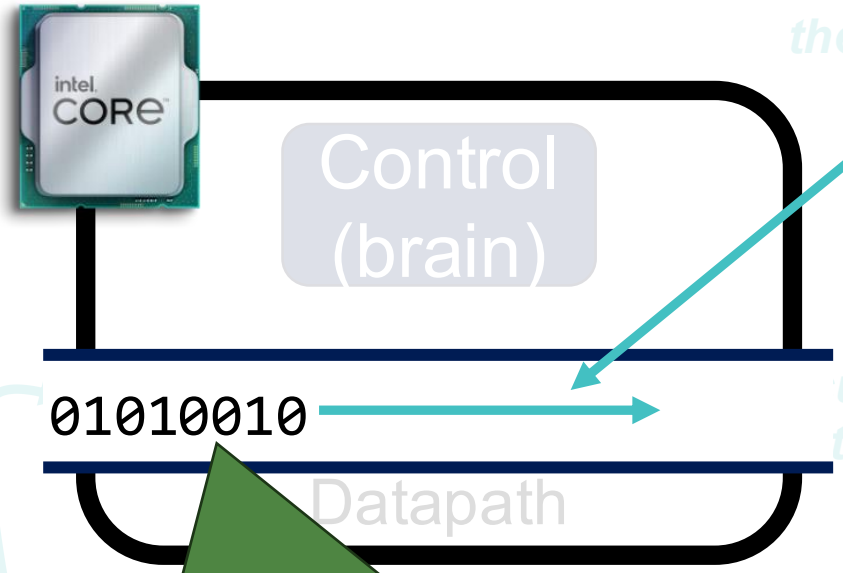
Main memory

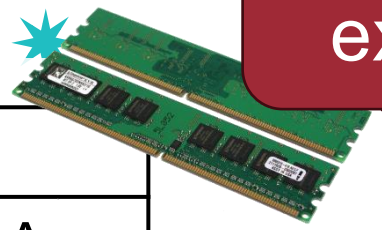
How is assembly code interpreted, and how does it interact with memory?

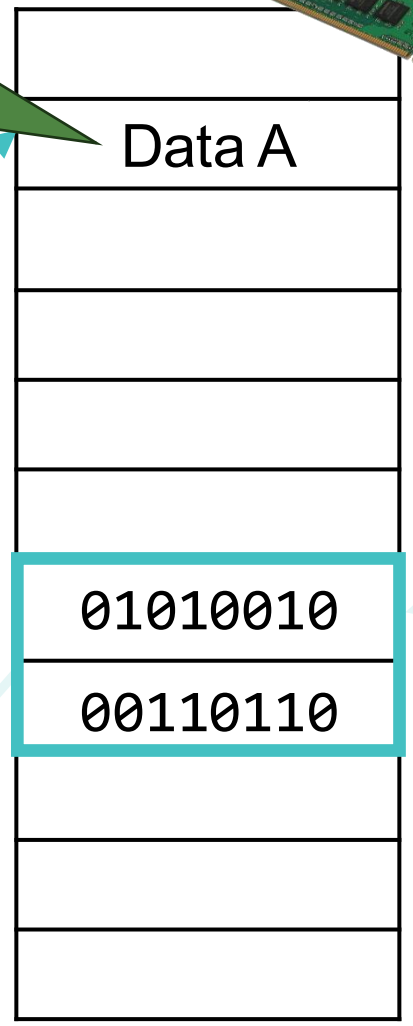
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Structure of an executable program







Main memory

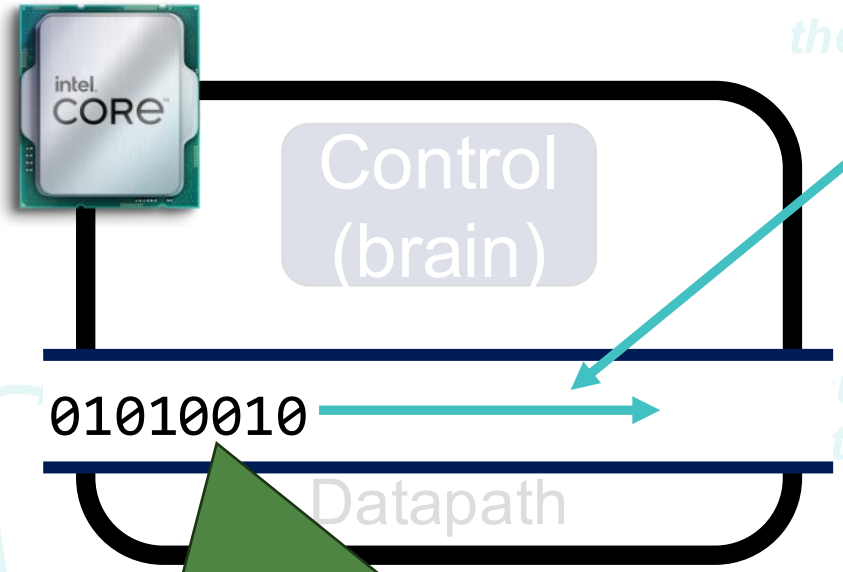
Hard disk

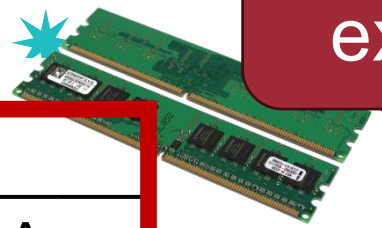
How is assembly code interpreted, and how does it interact with memory?

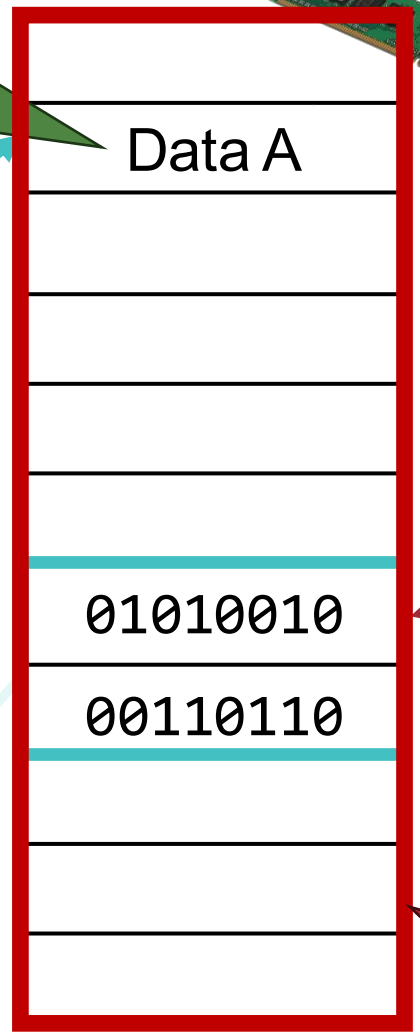
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Structure of an executable program







1. Load the Program A

Program loading process

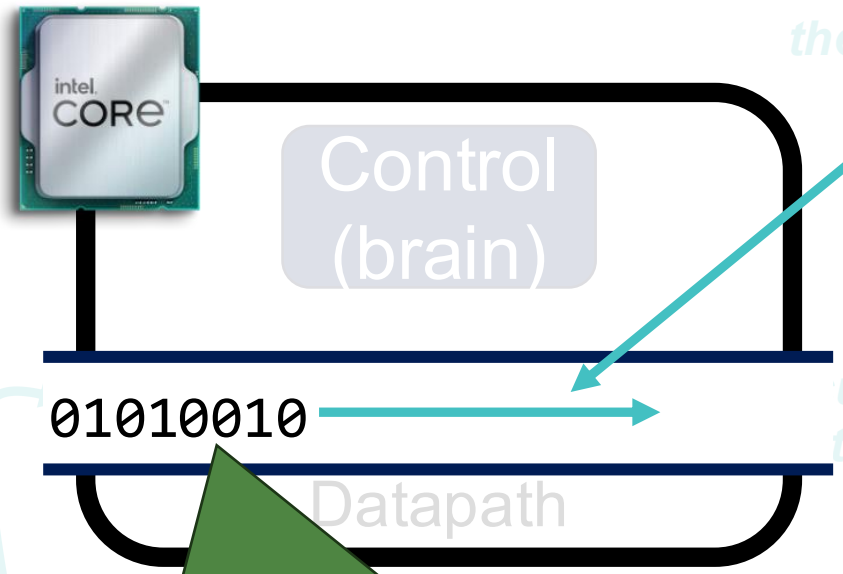
How is assembly code interpreted, and how does it interact with memory?

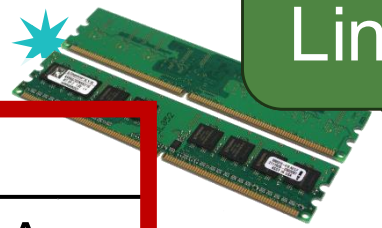
Memory layout

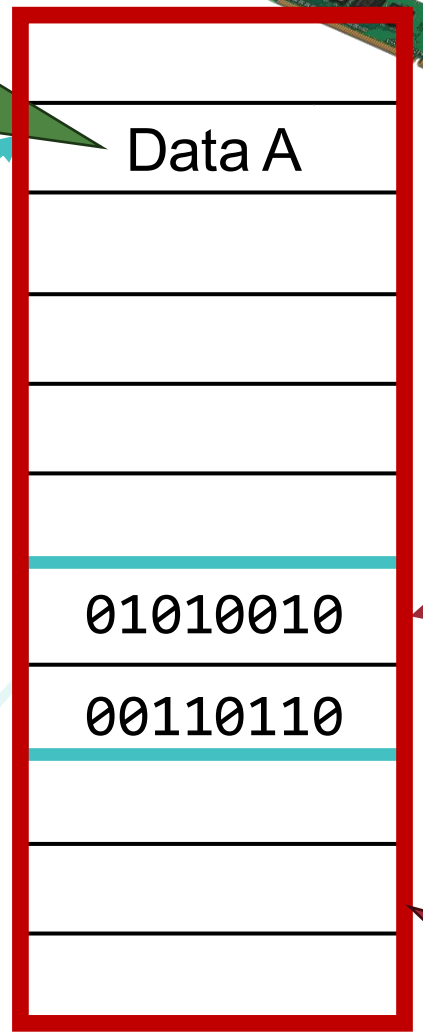
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Executable and Linkable Format (ELF)

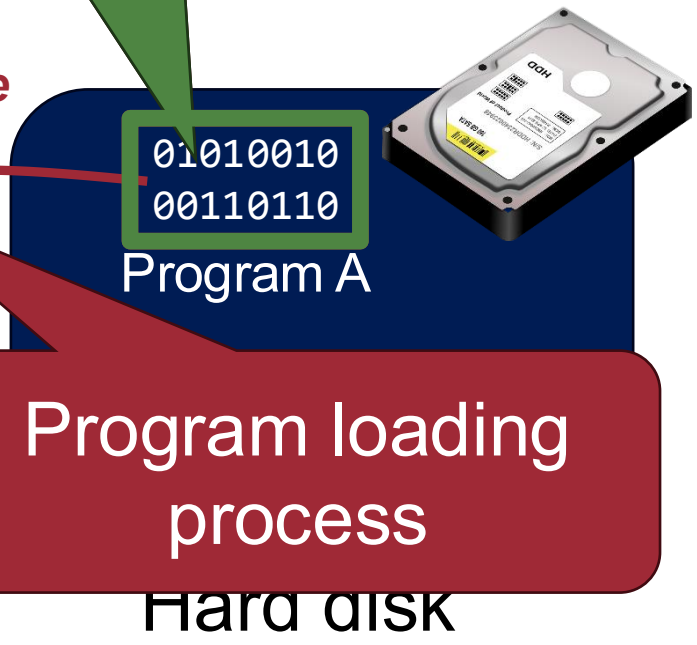






Main memory

1. Load the Program A



Hard disk

Program loading process

How is assembly code interpreted, and how does it interact with memory?

Memory layout

Executable Formats

Executable and Linkable Format (ELF)

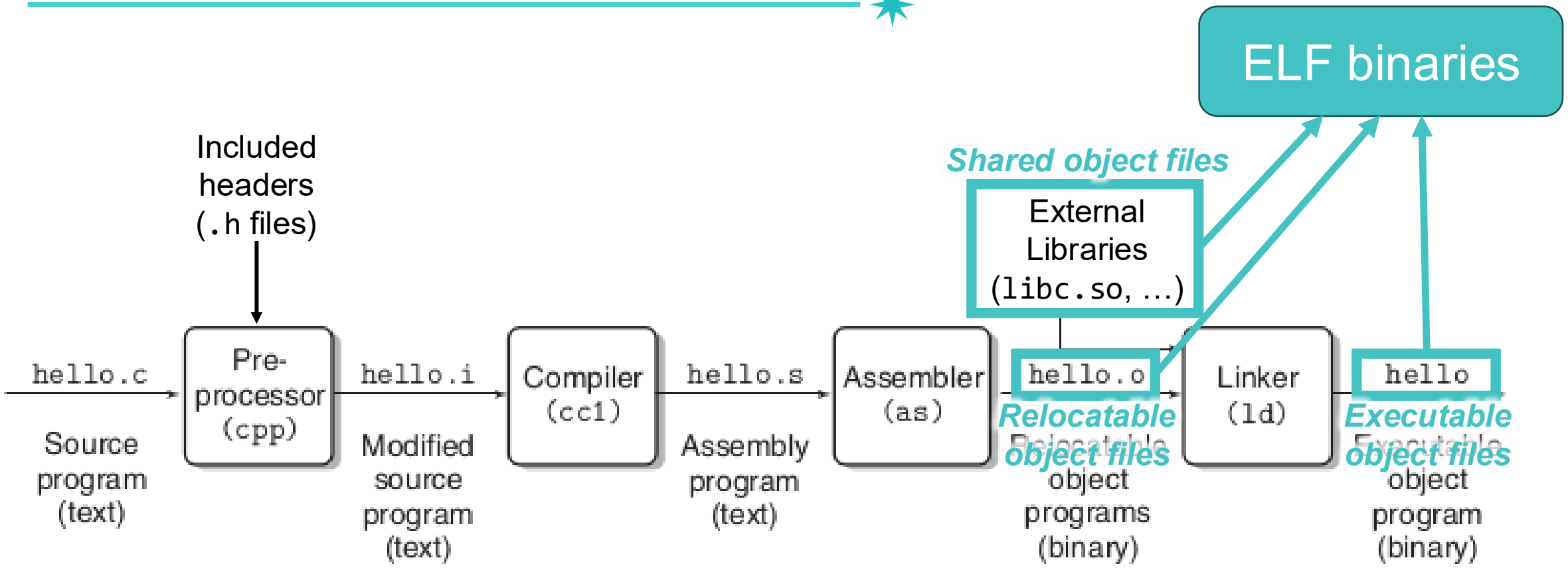
A **standard file format for binary files (object files)** used on Unix and Unix-like systems

- C.f., Windows: Portable Executable (PE)
- C.f., macOS: Mach Object File Format (Mach-O)

- One unified format for
  - Relocatable object files (`.o`)
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- Generic name: ELF binaries

Recap: Summary of the C Toolchain



ELF Object File Format

ELF has two views: the **section view** and the **segment view**:

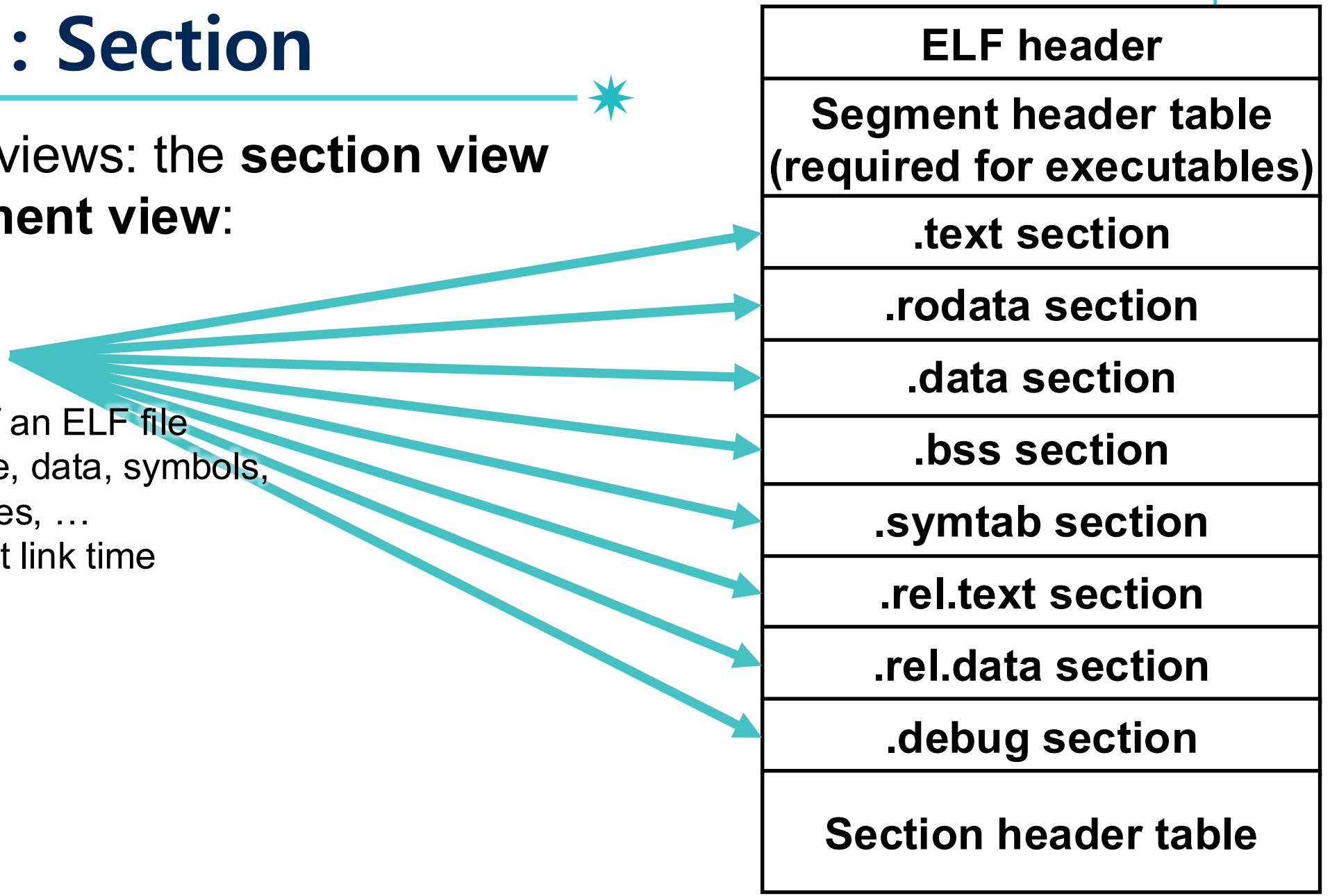
ELF header
Segment header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.text section
.rel.data section
.debug section
Section header table

Term #1: Section

ELF has two views: the **section view** and the **segment view**:

Section

- A logical unit of an ELF file
- Organizes code, data, symbols, relocation entries, …
- Only relevant at link time



Term #2: Segment

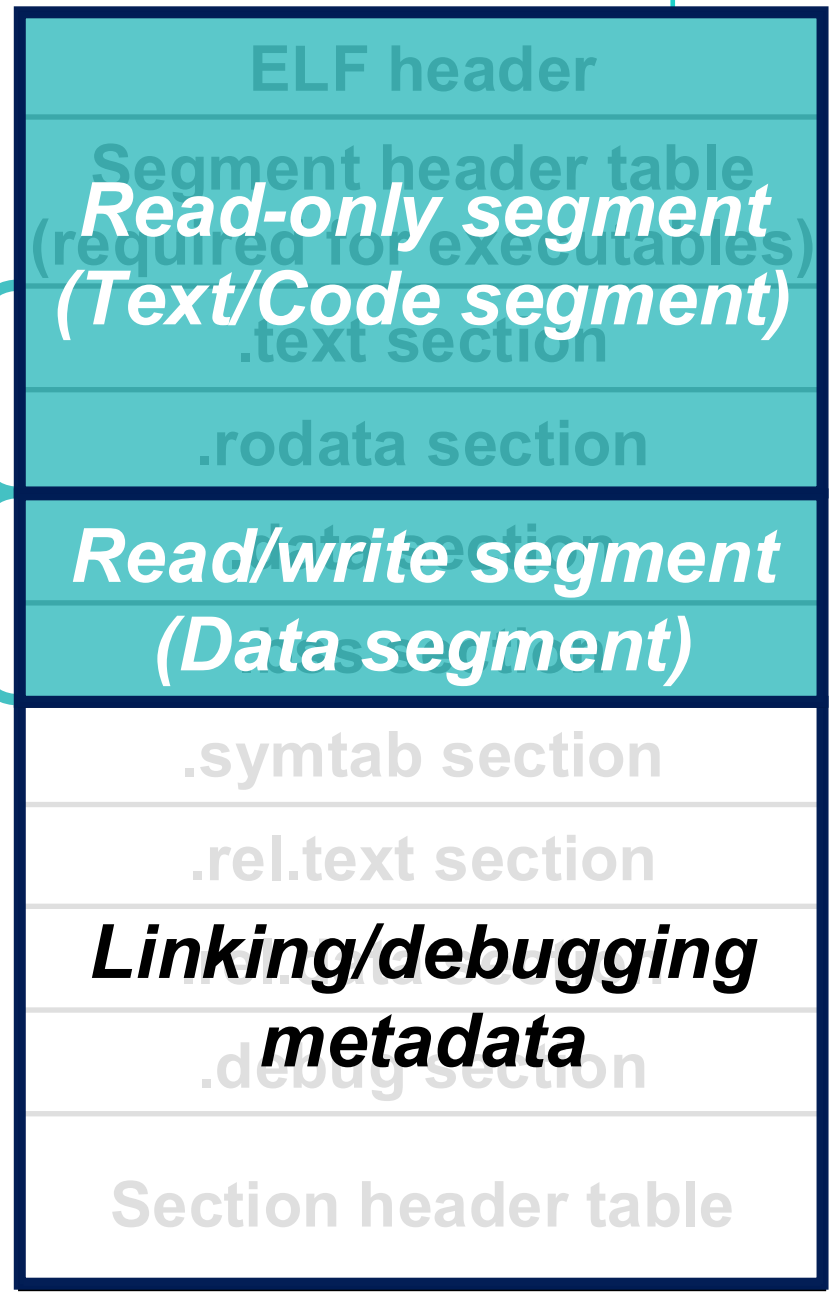
ELF has two views: the **section view** and the **segment view**:

Section

- A logical unit of an ELF file
- Organizes code, data, symbols, relocation entries, …
- Only relevant at link time

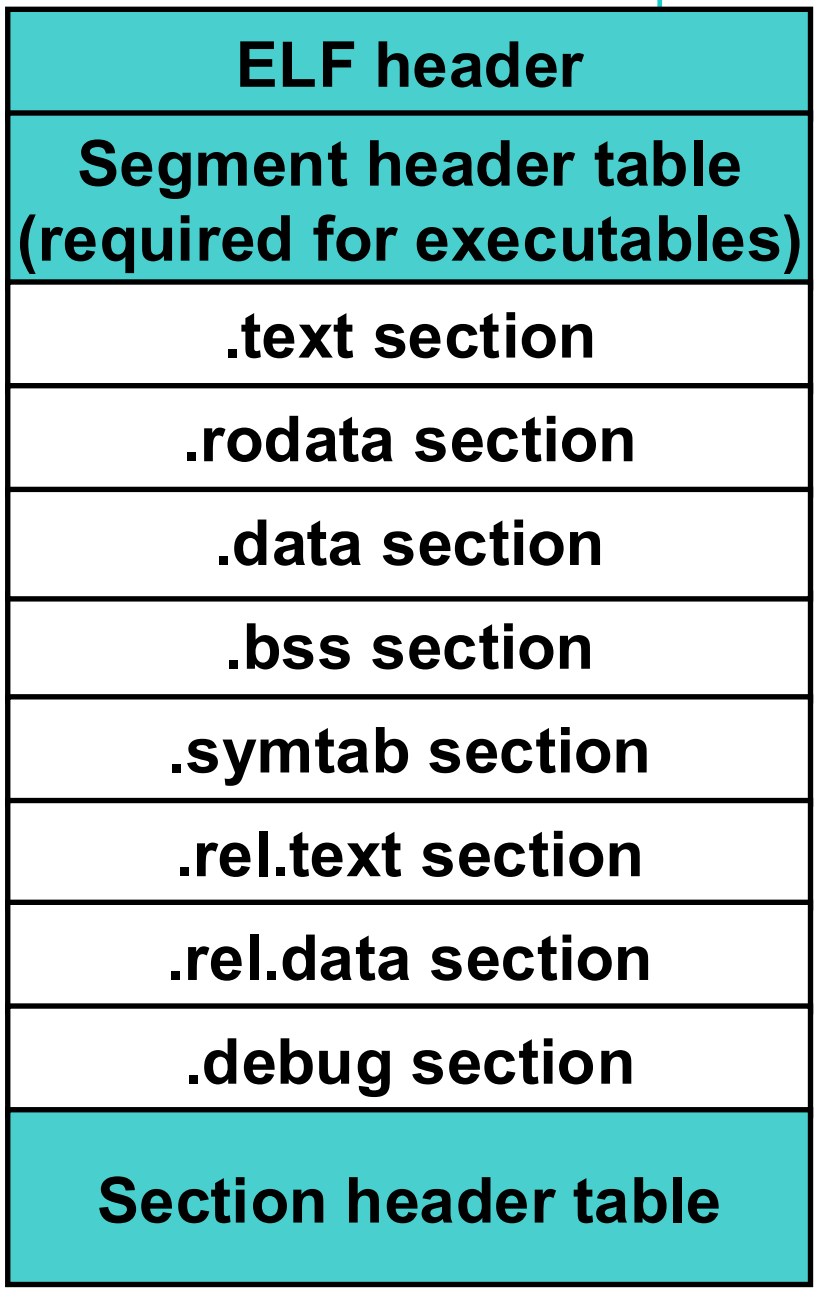
Segment

- Groups one or more sections into single segments
- Eventually be loaded into memory
- Only relevant at runtime



ELF Object File Format

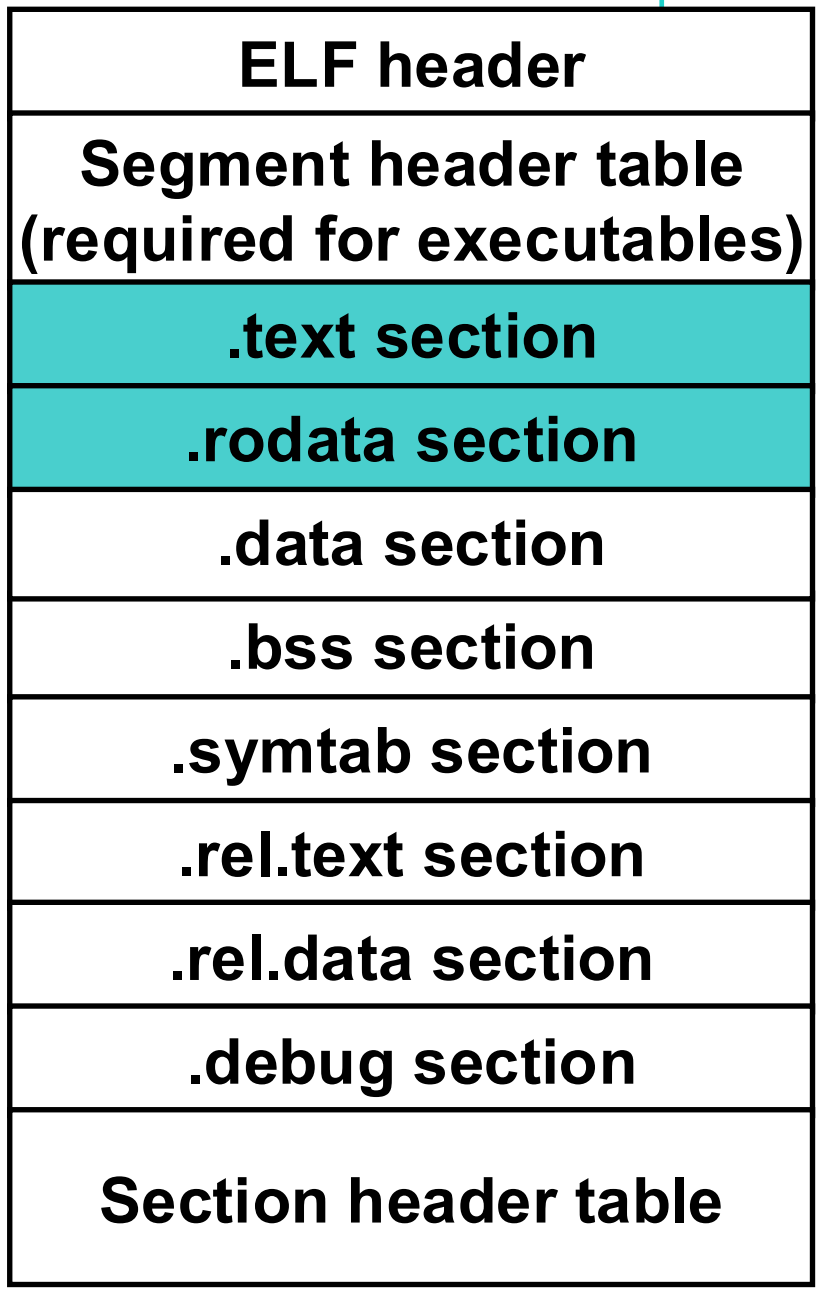
- **ELF header** (`$readelf -h <file>`)
  - Word size, byte ordering, file type (`.o`, `.so`, exec), machine type, etc.
- **Segment header table** (`$readelf -l <file>`)
  - Offsets and sizes of each segment
  - Page size, virtual address memory segments (sections), segment sizes (We will cover these in later.)
  - A.k.a., program header table
- **Section header table** (`$readelf -S <file>`)
  - Offsets and sizes of each section



ELF Object File Format

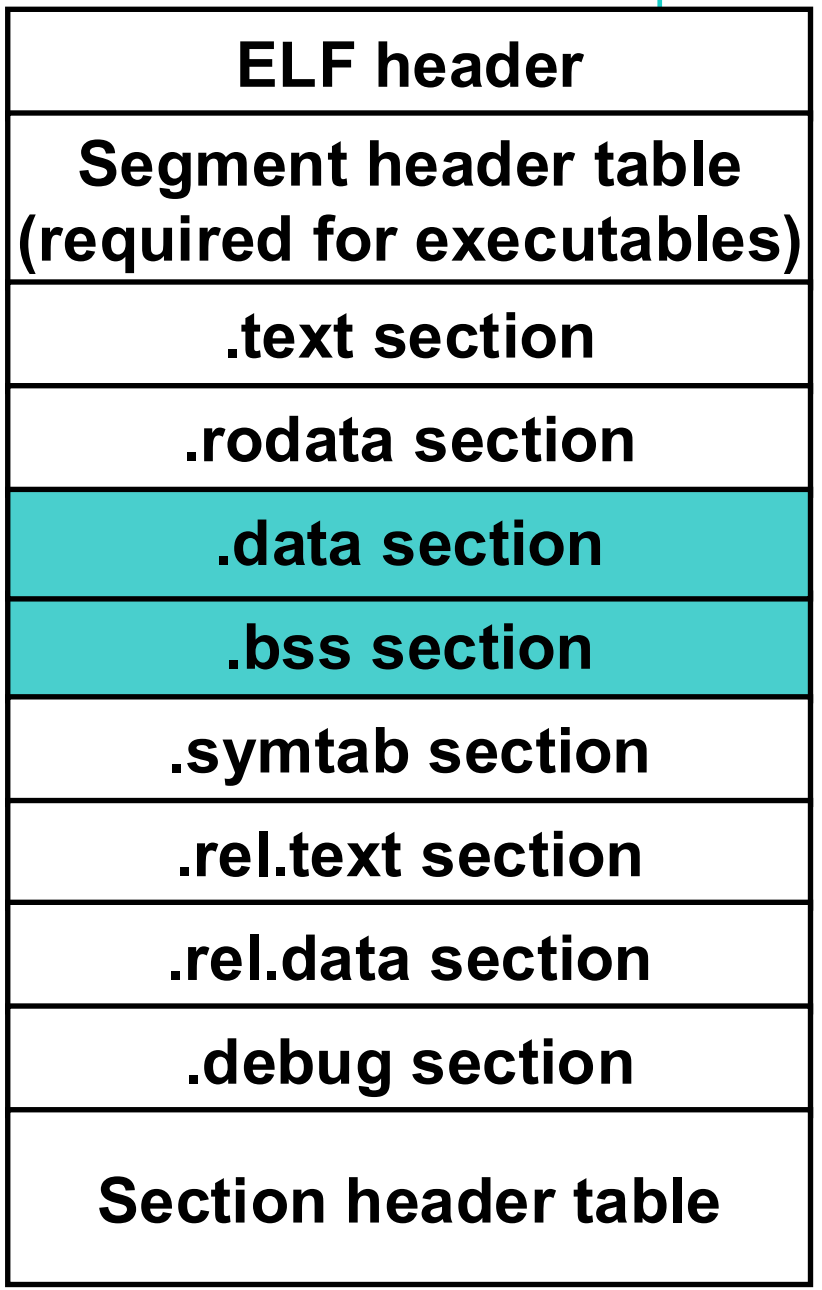
- **`.text` section**
  - Assembly code

- **`.rodata` section** (`$readelf -p .rodata <file>`)
  - **r**ead-**o**nly **data**: string constants, jump tables, …
  - Values that are set once and don't change throughout the execution
  - E.g., `char *ptr = "Ube"; printf("Hello!");`



ELF Object File Format

- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables



(FYI) Global Variables in C

- **Global variables** exist outside of any function, **can be accessed from any function**
- Exist throughout the entire lifespan of a program

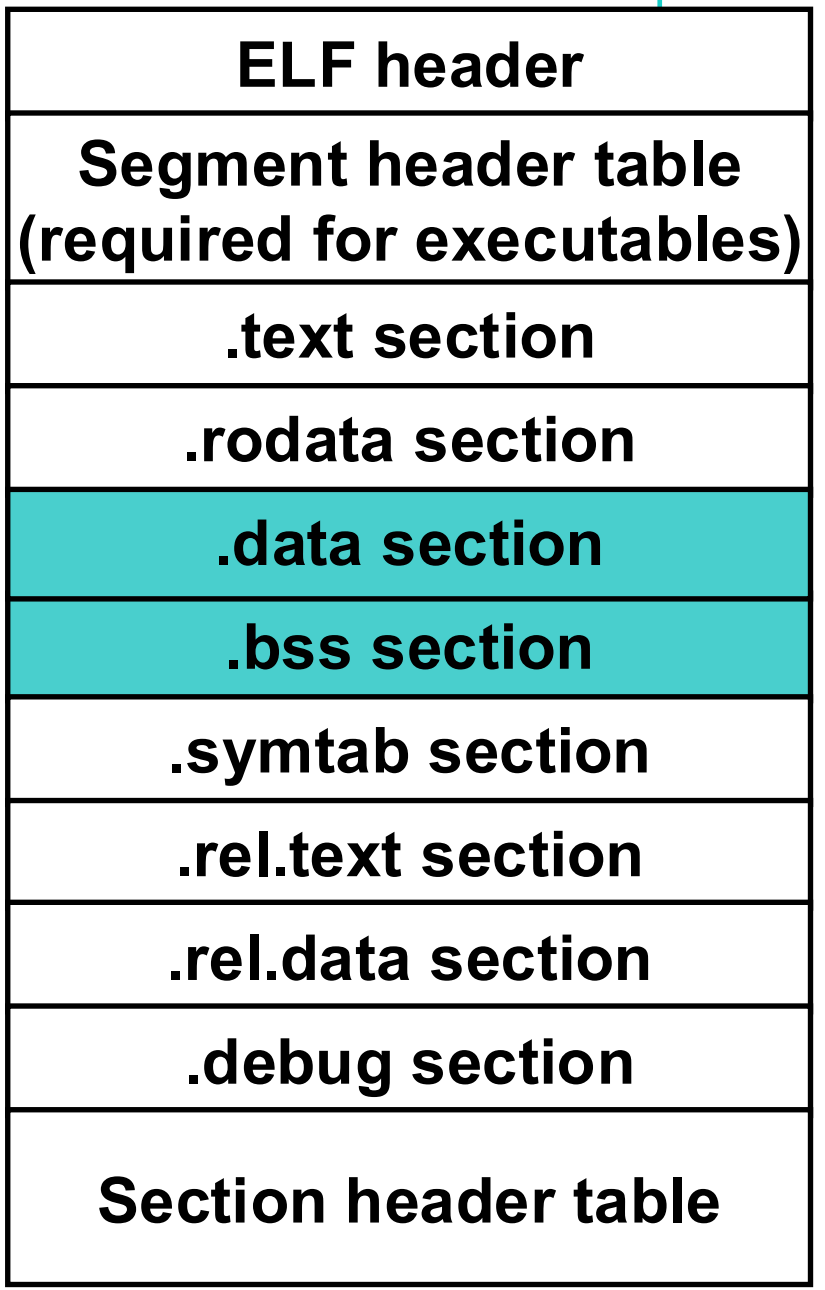
```
int x = 1;

void incr_globals() {
    x ++;
}

int main() {
    printf("x: %d\n", x); // prints 1
    incr_globals();
    printf("x: %d\n", x); // prints 2
    return 0;
}
```

Declaring a variable outside of a function makes it "global"

ELF Object File Format

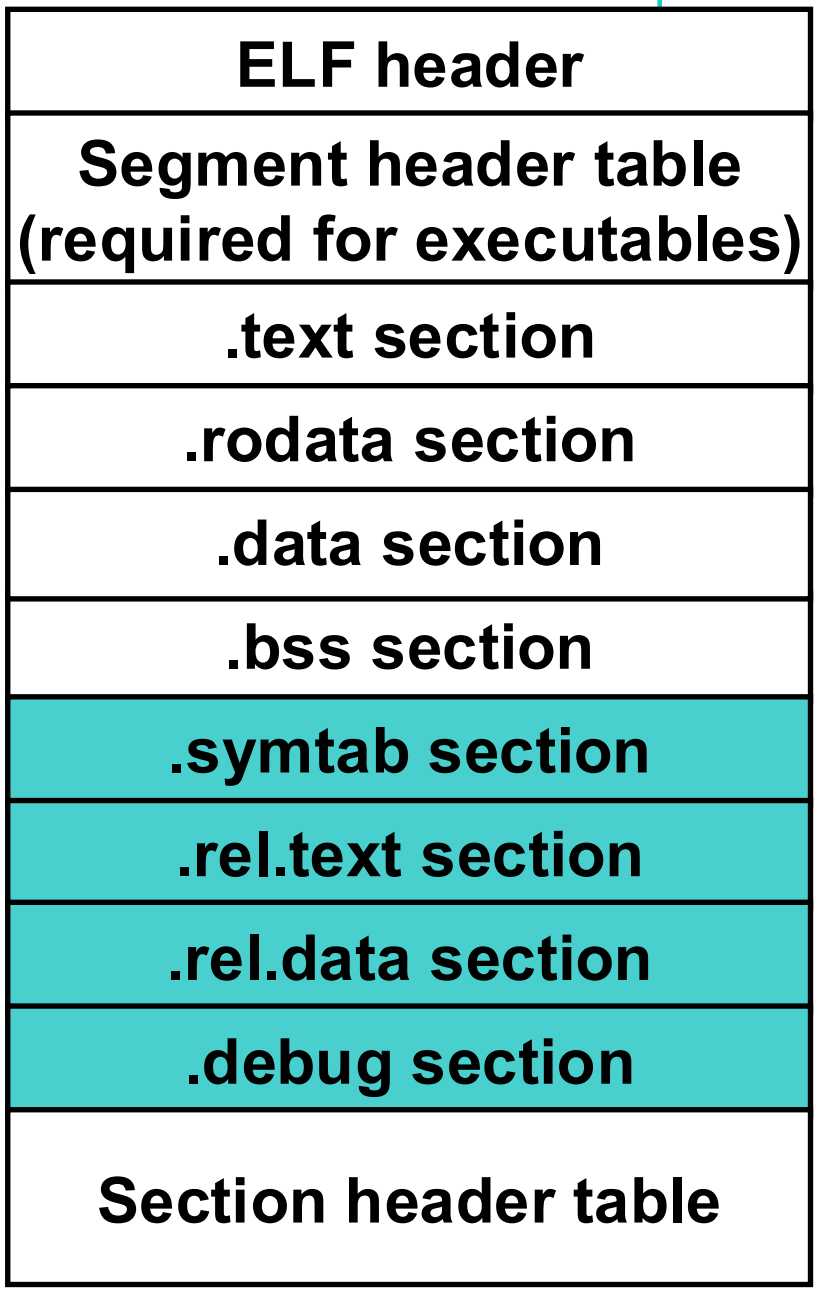
- **`.data` section**
  - Initialized global variables
- **`.bss` section**
  - Uninitialized global variables
  - Occupies no space in the executable file (save space)
  - When the program is loaded, memory for the `.bss` region is allocated and initialized to zero

```
int x = 1; // initialized global var
int y;     // uninitialized global var

int main() {
    ...
}
```



ELF Object File Format

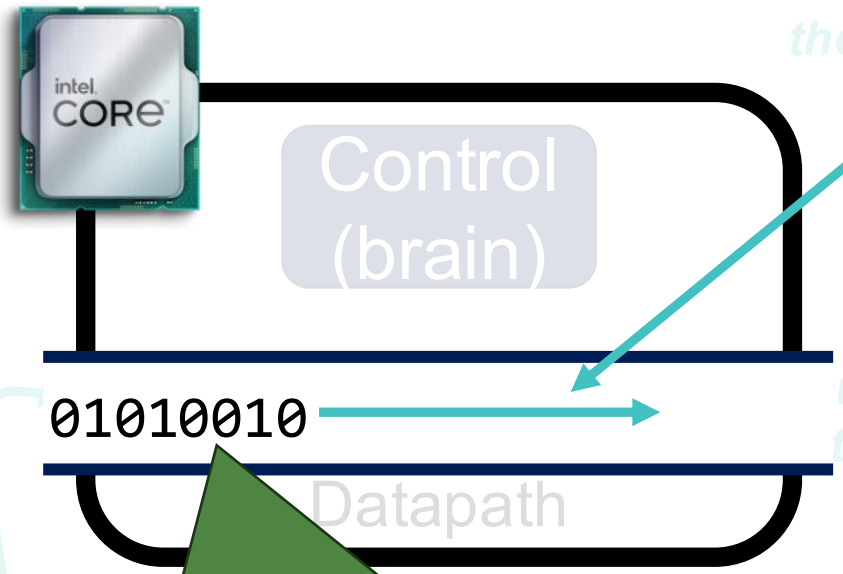
- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying
- **`.rel.data` section**
  - Relocation info for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug` section**
  - Info for symbolic debugging (`$gcc -g`)

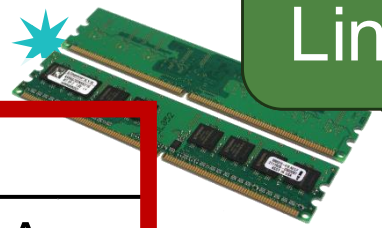


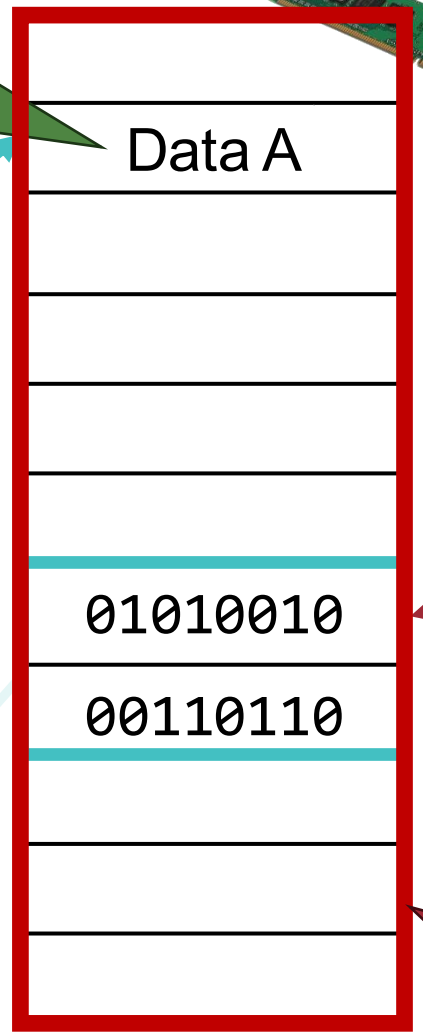
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Executable and Linkable Format (ELF)

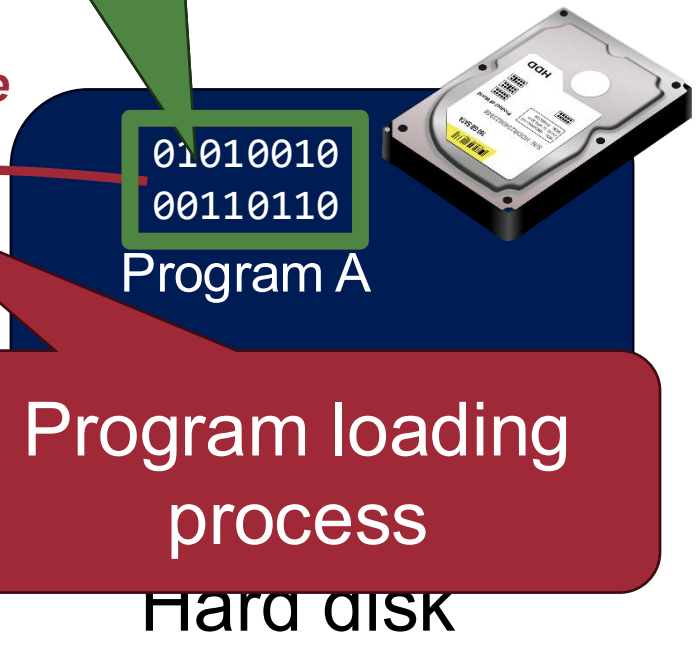






Main memory

1. Load the Program A



Hard disk

Program loading process

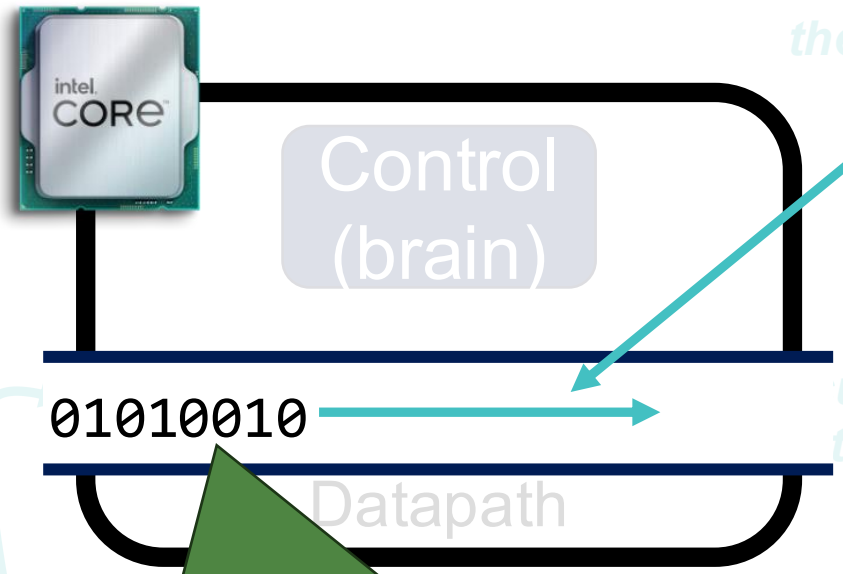
How is assembly code interpreted, and how does it interact with memory?

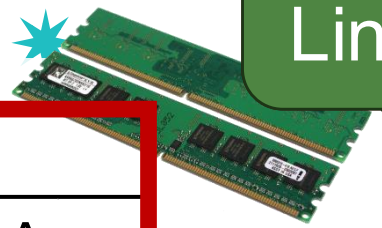
Memory layout

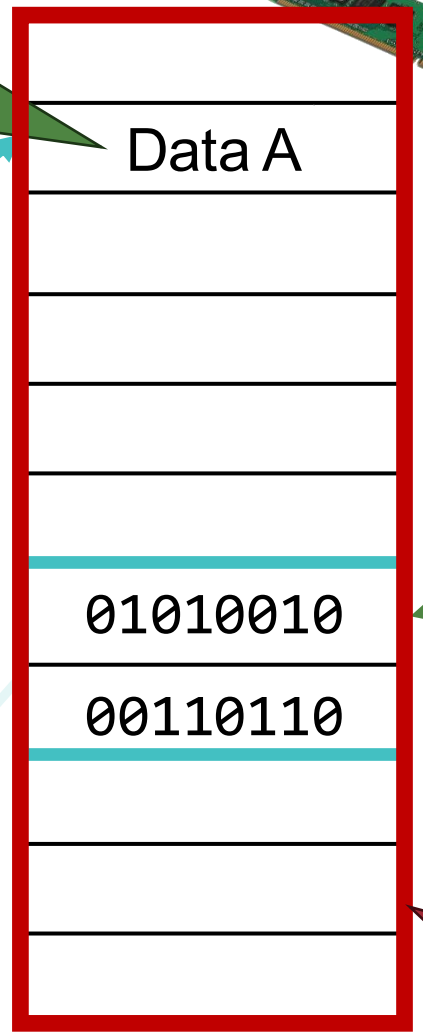
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Executable and Linkable Format (ELF)

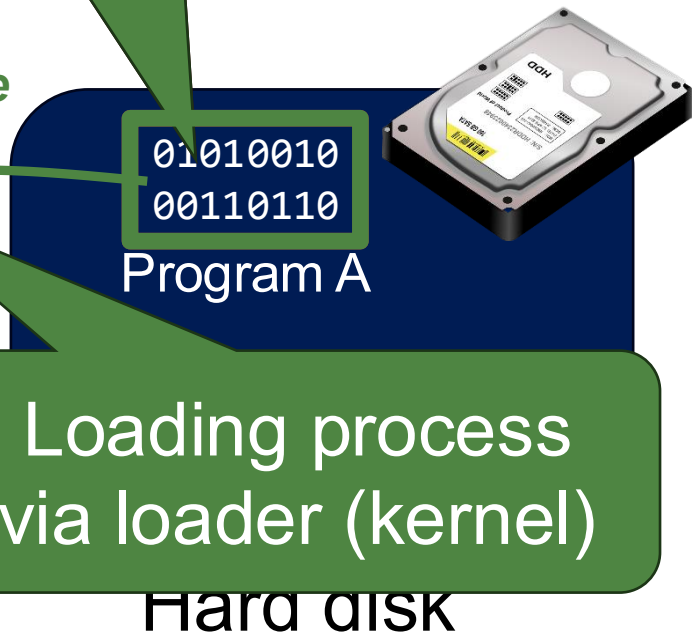






Main memory

1. Load the Program A



Hard disk

Loading process via loader (kernel)

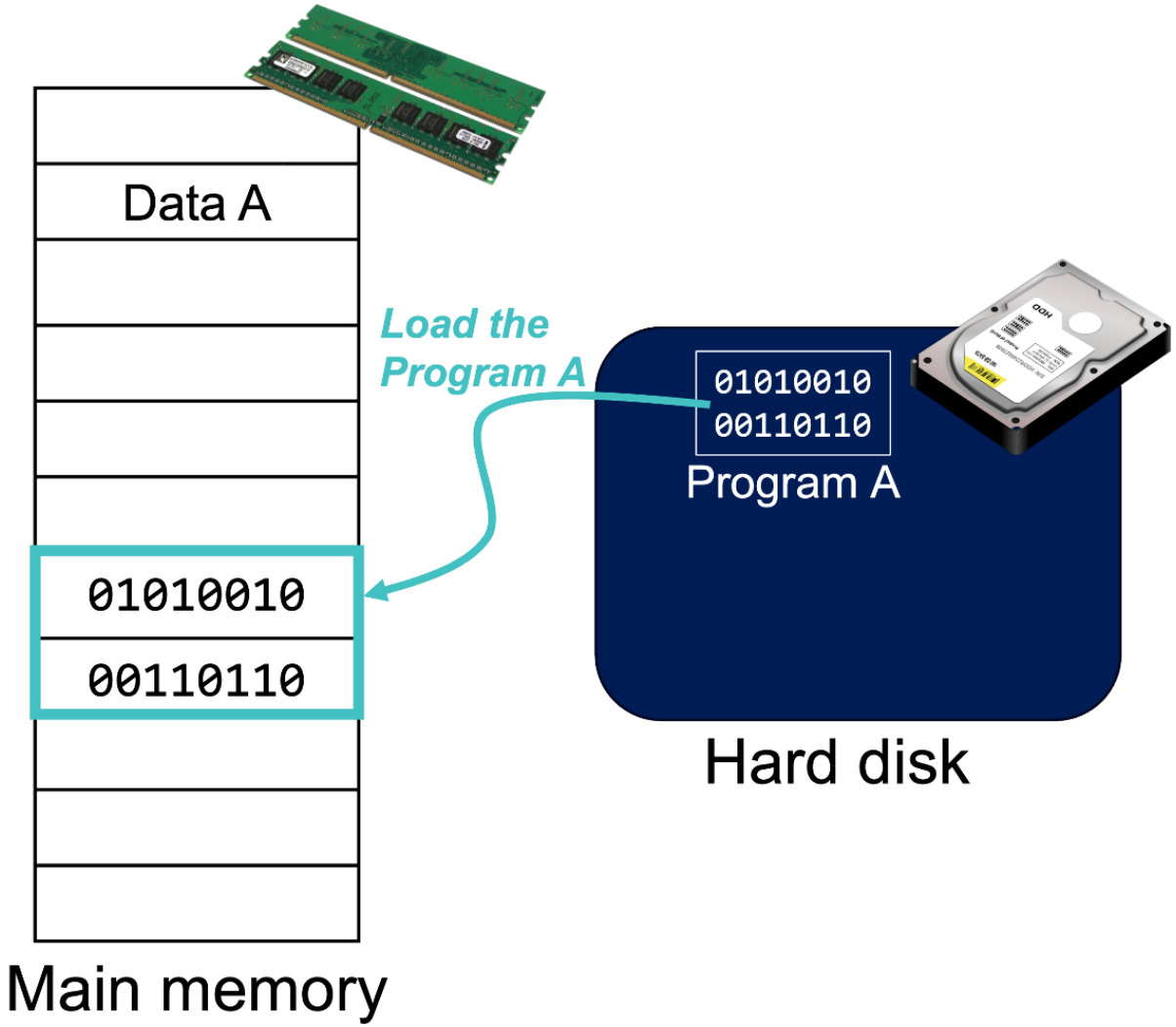
How is assembly code interpreted, and how does it interact with memory?

Memory layout

Loading Process

What is Loading?

- The process of moving an executable (ELF file) into memory **for execution**



What happens when you run a program?

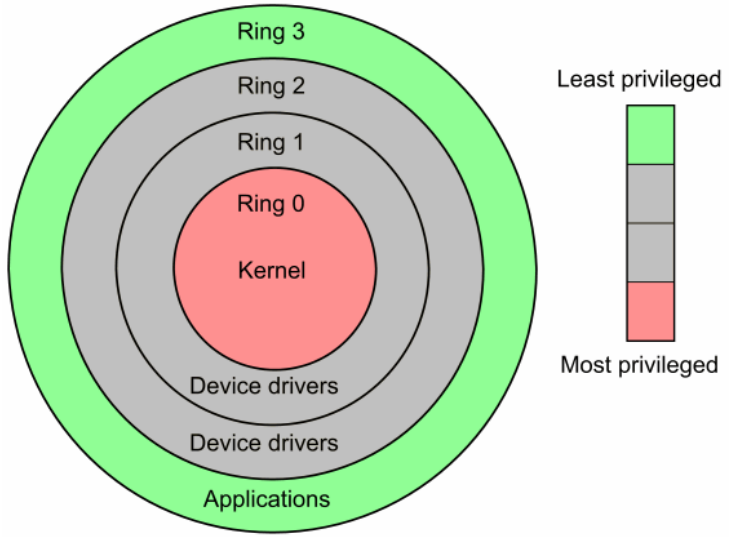
`($ ./a.out)`

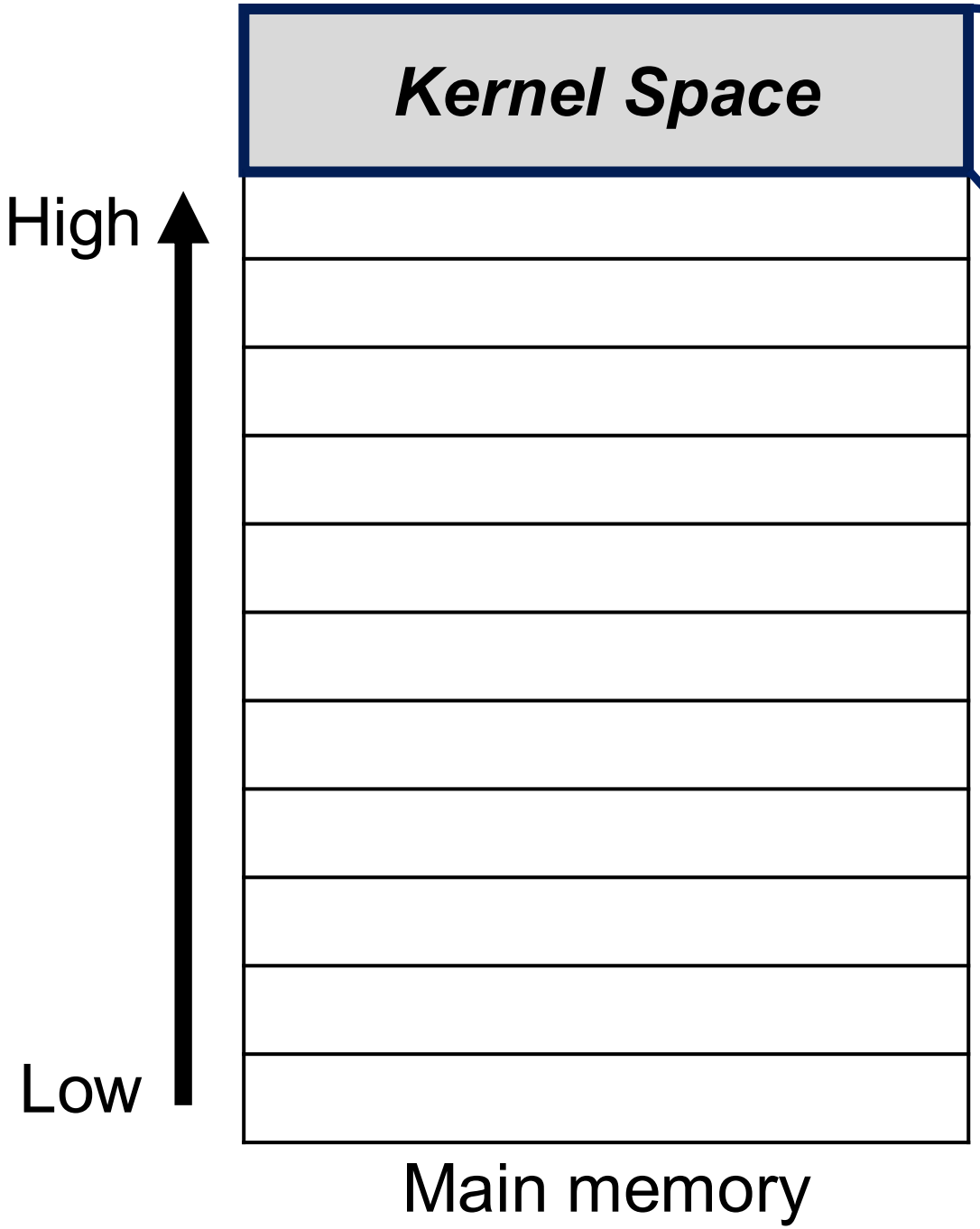
What Happens when You Run a Program?

1. The kernel reads the ELF executable and checks its headers

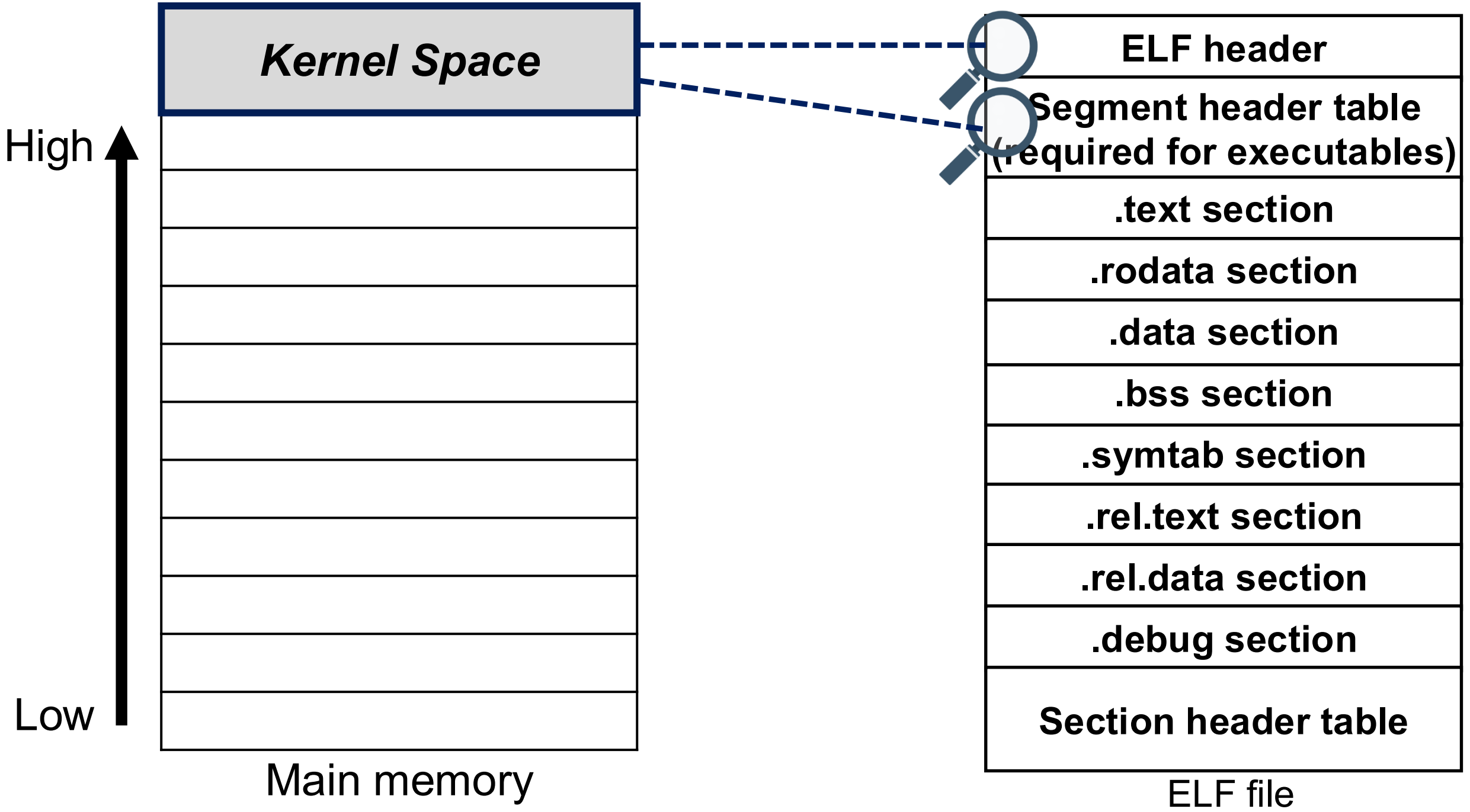
FYI: Kernel

- A computer **program** at the core of the OS that has complete control over everything in the system (software & hardware resources)
- It is the portion of the OS code that is always resident in memory
- It will be covered in more detail in CSE311 (Operating Systems)





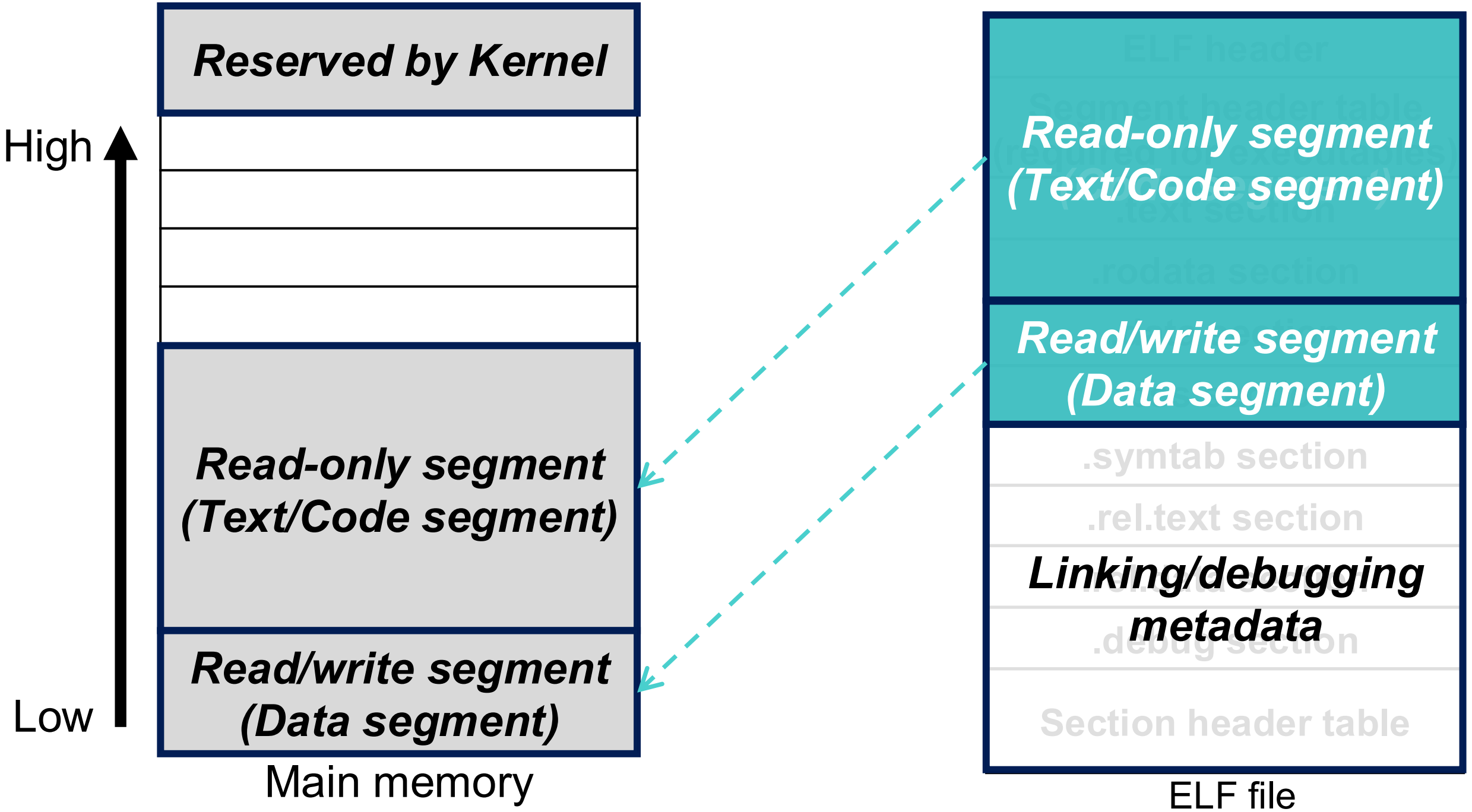
- Used by the OS to store kernel code and privileged process-management data
- User code cannot read from nor write to these addresses


Kernel Space
High
Low
Main memory
ELF header
Segment header table
(required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.text section
.rel.data section
.debug section
Section header table
ELF file

What Happens when You Run a Program?

1. The kernel reads the ELF executable and checks its headers
2. The kernel maps (loadable) segments into the program's memory address part

The `.bss` region is allocated as zero-initialized memory at load time.


Reserved by Kernel
High
Low
Read-only segment
(Text/Code segment)
Read/write segment
(Data segment)
Main memory
Read-only segment
(Text/Code segment)
Read/write segment
(Data segment)
.symtab section
.rel.text section
Linking/debugging
metadata
.debug section
Section header table
ELF file

What Happens when You Run a Program?

1. The kernel reads the ELF executable and checks its headers
2. The kernel maps (loadable) segments into the program's memory address part
3. The kernel also sets up stack (`args`, env vars) and registers (`rip`, `rsp`, ...)
4. If the executable is dynamically linked, the kernel also loads the dynamic linker (`ld-linux.so`) specified in the ELF file
5. The kernel (or dynamic linker) transfers control the entry point of the binary: `_start`

We will cover about it later!

Who Invokes `main`?

Who Invokes main?


```c
#include<stdio.h>
int main(void) {
    puts("hello world");
    return 0;
}
```

Compile `hello.c` with the following command:

```
$ gcc hello.c -o hello
```

And disassemble the resulting binary file to figure out the followings

```
401040 <_start>:
 401044: xor  ebp,ebp
 401046: mov  r9,rdx
 401049: pop  rsi
 ...
 401058: mov  rdi,0x401126 # address of main
 40105f: call QWORD PTR [rip+0x2f73]
          # 403fd8 <__libc_start_main@GLIBC>
```

Who Invokes main?


```c
#include<stdio.h>
int main(void) {
    puts("hello world");
    return 0;
}
```

Compile `hello.c` with the following command:

```
$ gcc hello.c -o hello
```

And disassemble the resulting binary file to figure out the followings

```
401040 <_start>:
 401044: xor   ebp,ebp
 401046: mov   r9,rdx
 401049: pop   rsi
 ...
 401058: mov   rdi,0x401126 # address of main
 40105f: call QWORD PTR [rip+0x2f73]
          # 403fd8 <__libc_start_main@GLIBC>
```

The entry point of the binary is `_start`, not `main`

The main() is never invoked in the binary

Who Invokes main?


```c
#include<stdio.h>
int main(void) {
    puts("hello world");
    return 0;
}
```

Compile `hello.c` with the following command:

```
$ gcc hello.c -o hello
```

And disassemble the resulting binary file to figure out the followings

```
401040 <_start>:
 401044: xor   ebp,ebp
 401046: mov   r9,rdx
 401049: pop   rsi
 ...
 401058: mov   rdi,0x401126 # addres    main
 40105f: call QWORD PTR [rip+0x2f73]
           # 403fd8 <__libc_start_main@GLIBC>
```

The entry point of the binary is `_start`, not `main`

The main() is never invoked in the binary

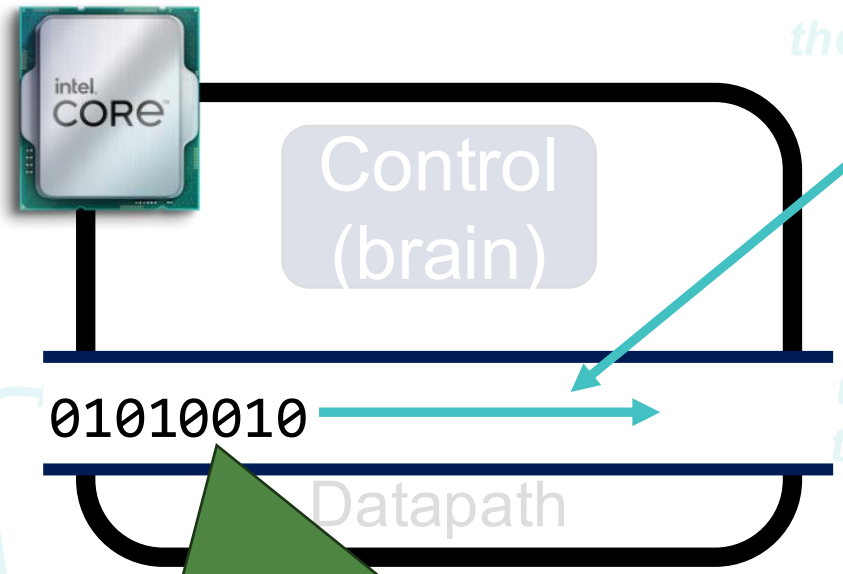
It will simply invoke `__libc_start_main`, but nothing else

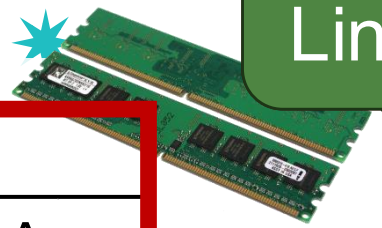
`__libc_start_main` *performs C runtime initialization and invokes* `main`

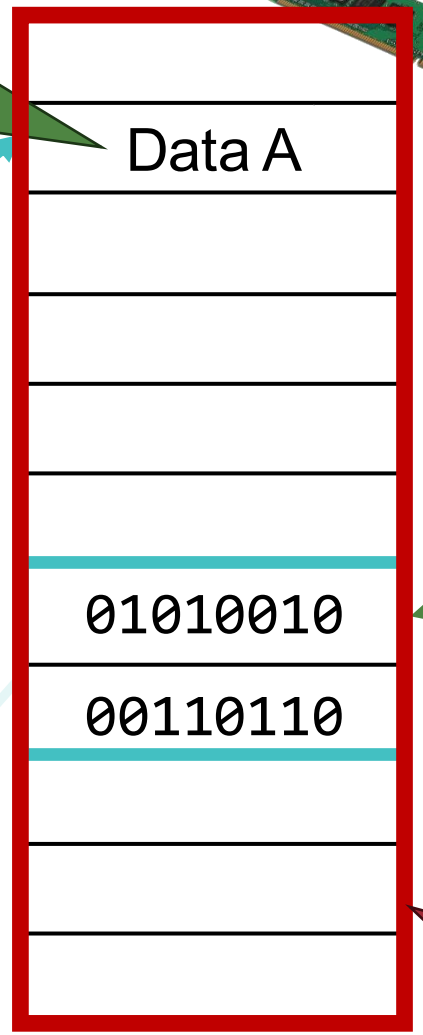
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Executable and Linkable Format (ELF)

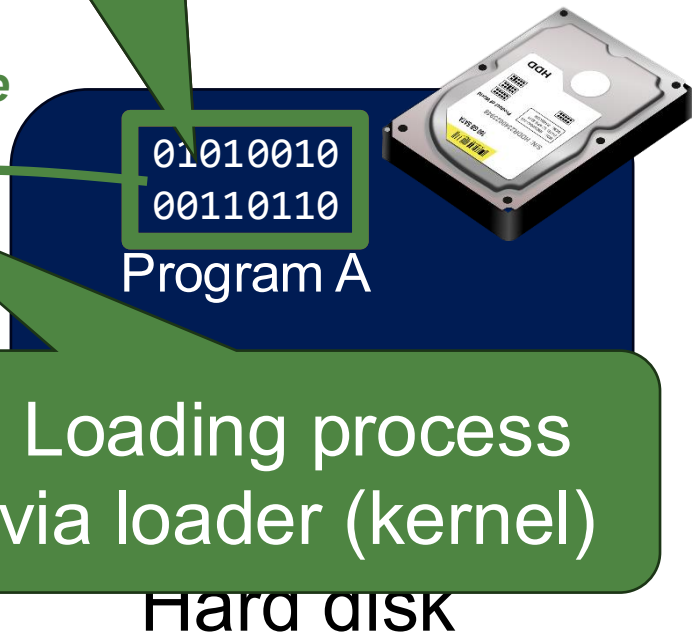






Main memory

1. Load the Program A



Hard disk

Loading process via loader (kernel)

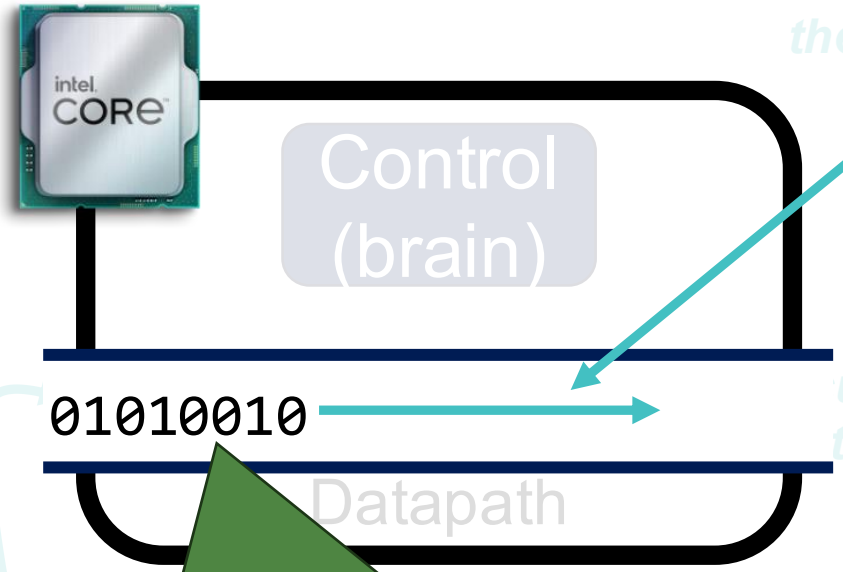
How is assembly code interpreted, and how does it interact with memory?

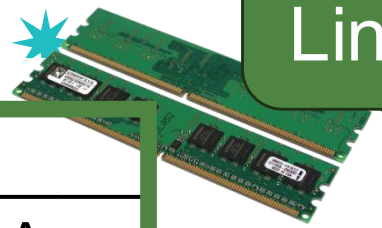
Memory layout

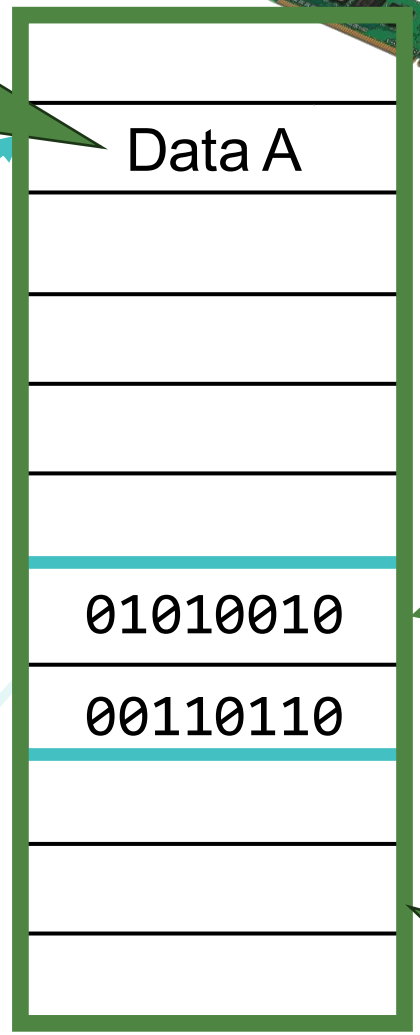
Today's Topic

How data (bit, byte, int, …) is represented in memory?

Executable and Linkable Format (ELF)







Main memory

1. Load the Program A

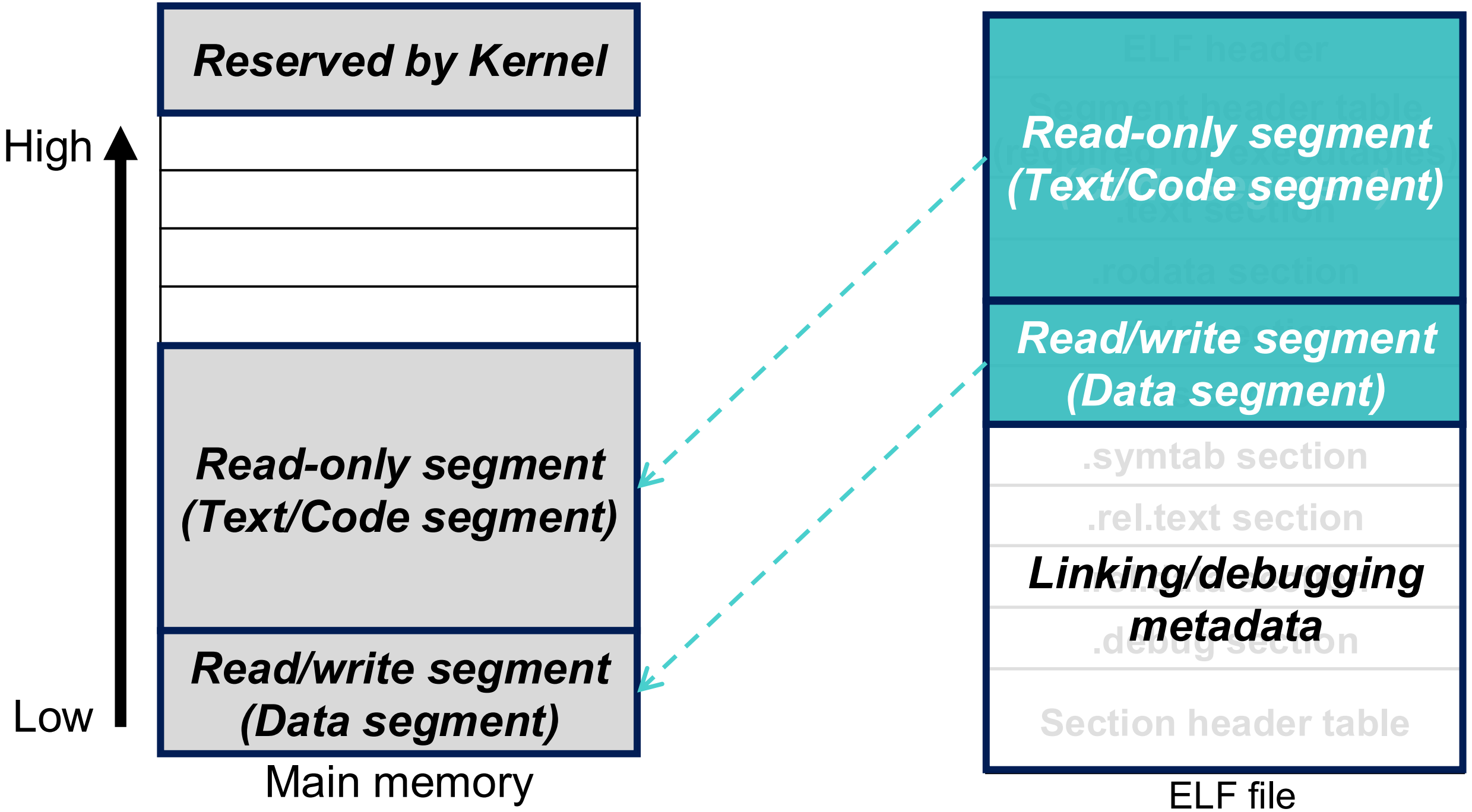
Hard disk

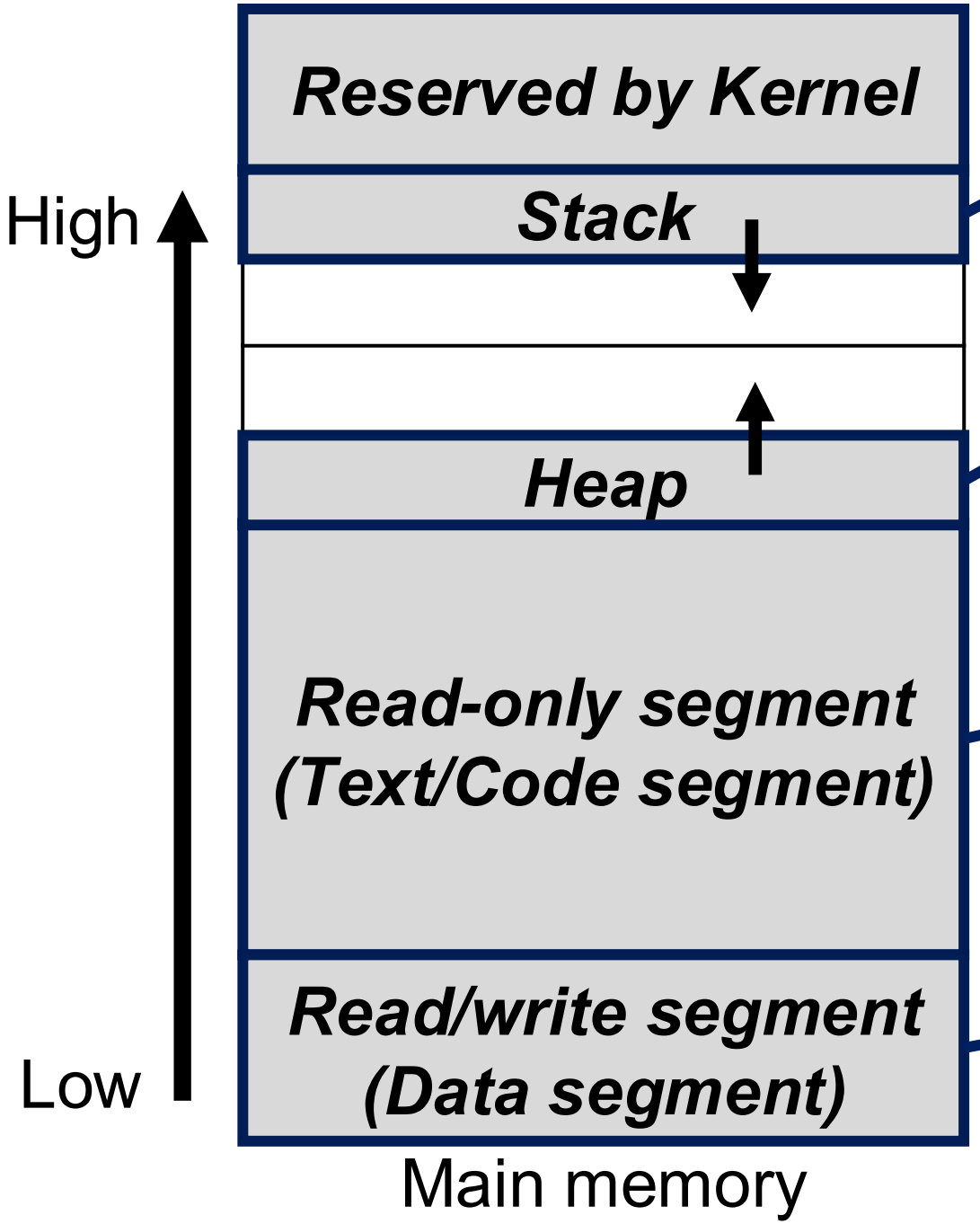
Loading process via loader (kernel)

How is assembly code interpreted, and how does it interact with memory?

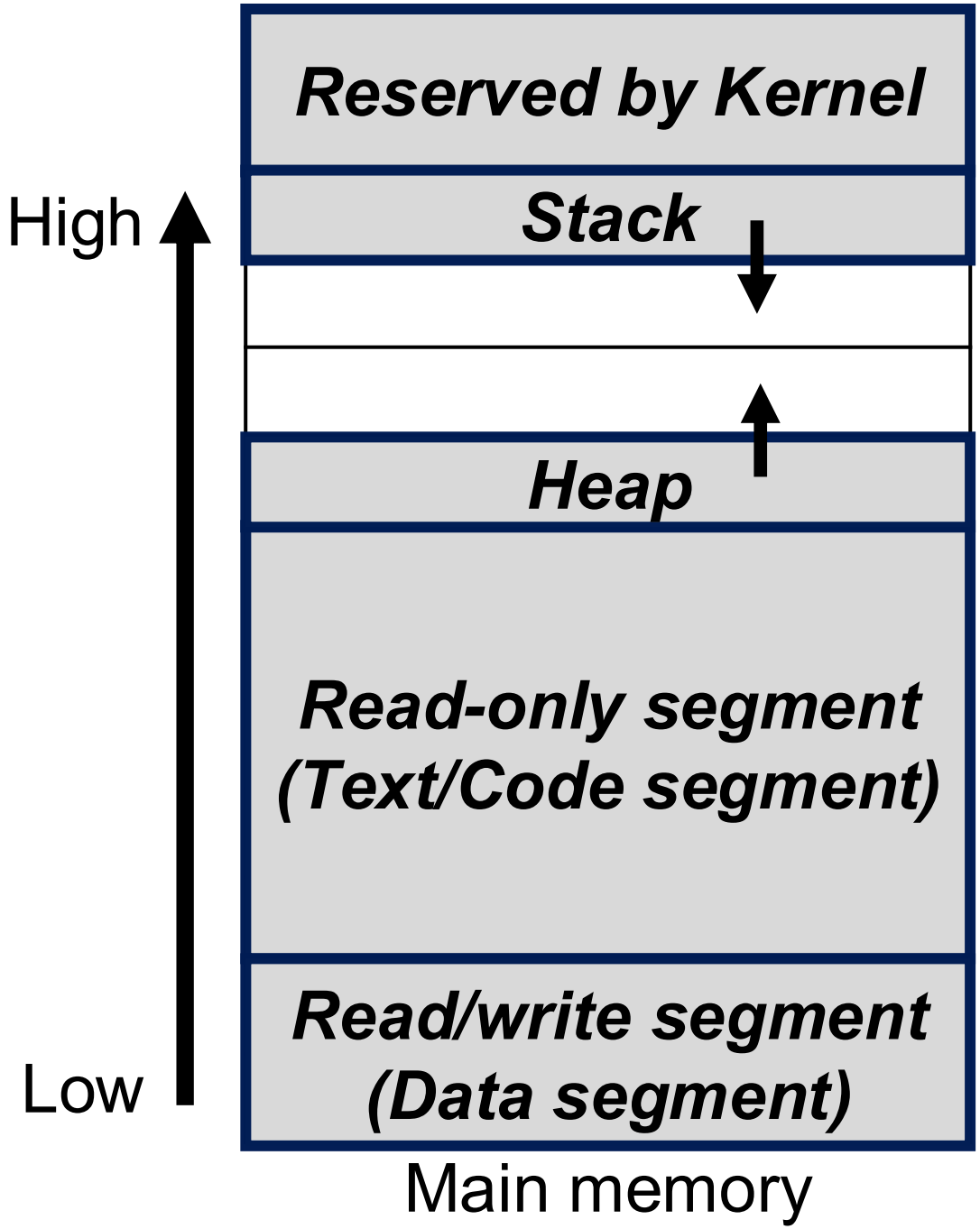
Stack, heap, code, data regions

Memory Layout

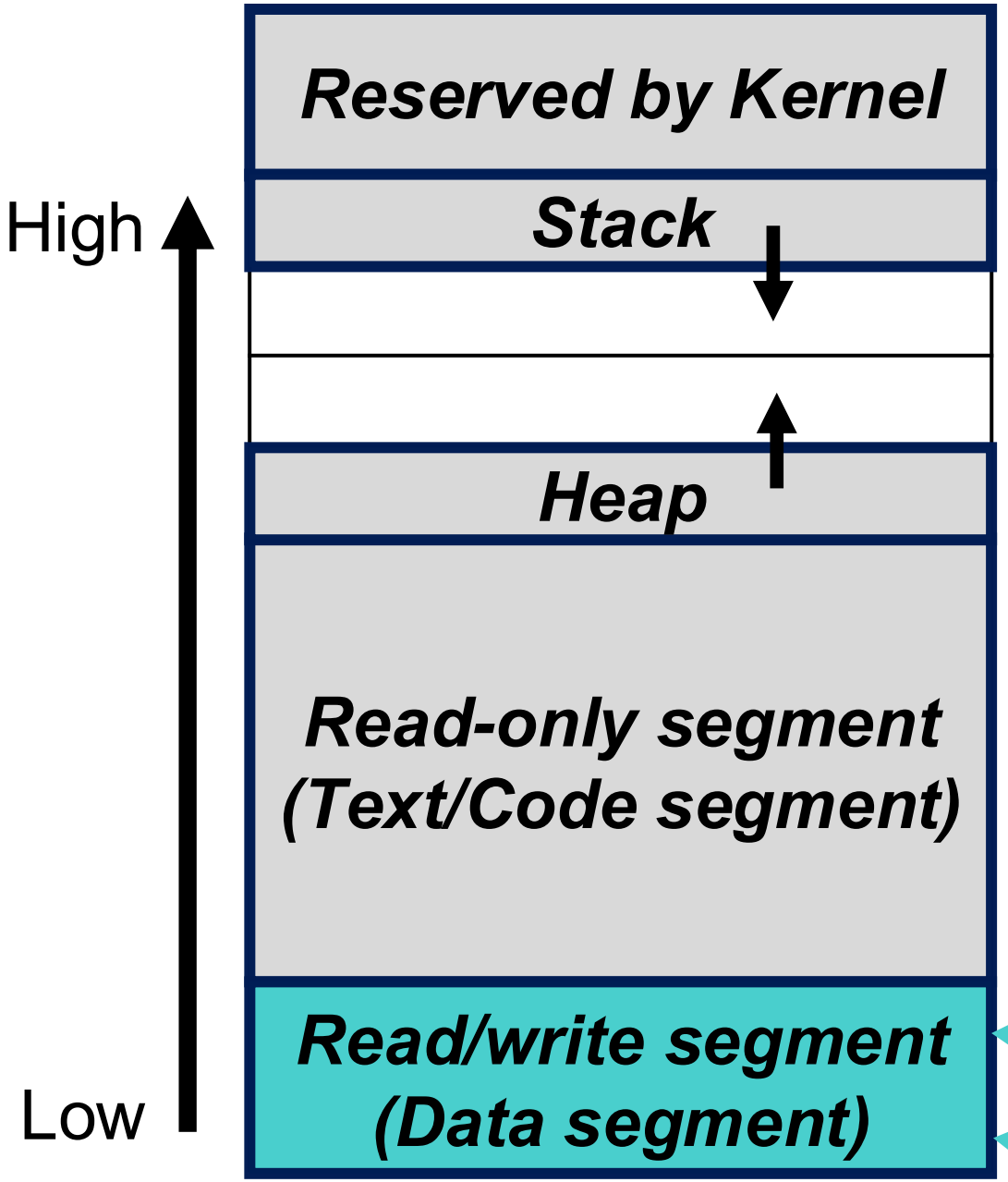

Reserved by Kernel
High
Low
Read-only segment
(Text/Code segment)
Read/write segment
(Data segment)
Main memory
Read-only segment
(Text/Code segment)
Read/write segment
(Data segment)
.symtab section
.rel.text section
Linking/debugging
metadata
.debug section
Section header table
ELF file



- Stores local variables, function call information
- Dynamic memory for programmer to allocate (e.g., via `malloc()` or `realloc()`)
- Stores instructions being executed
- Stores (initialized and uninitialized) global variables

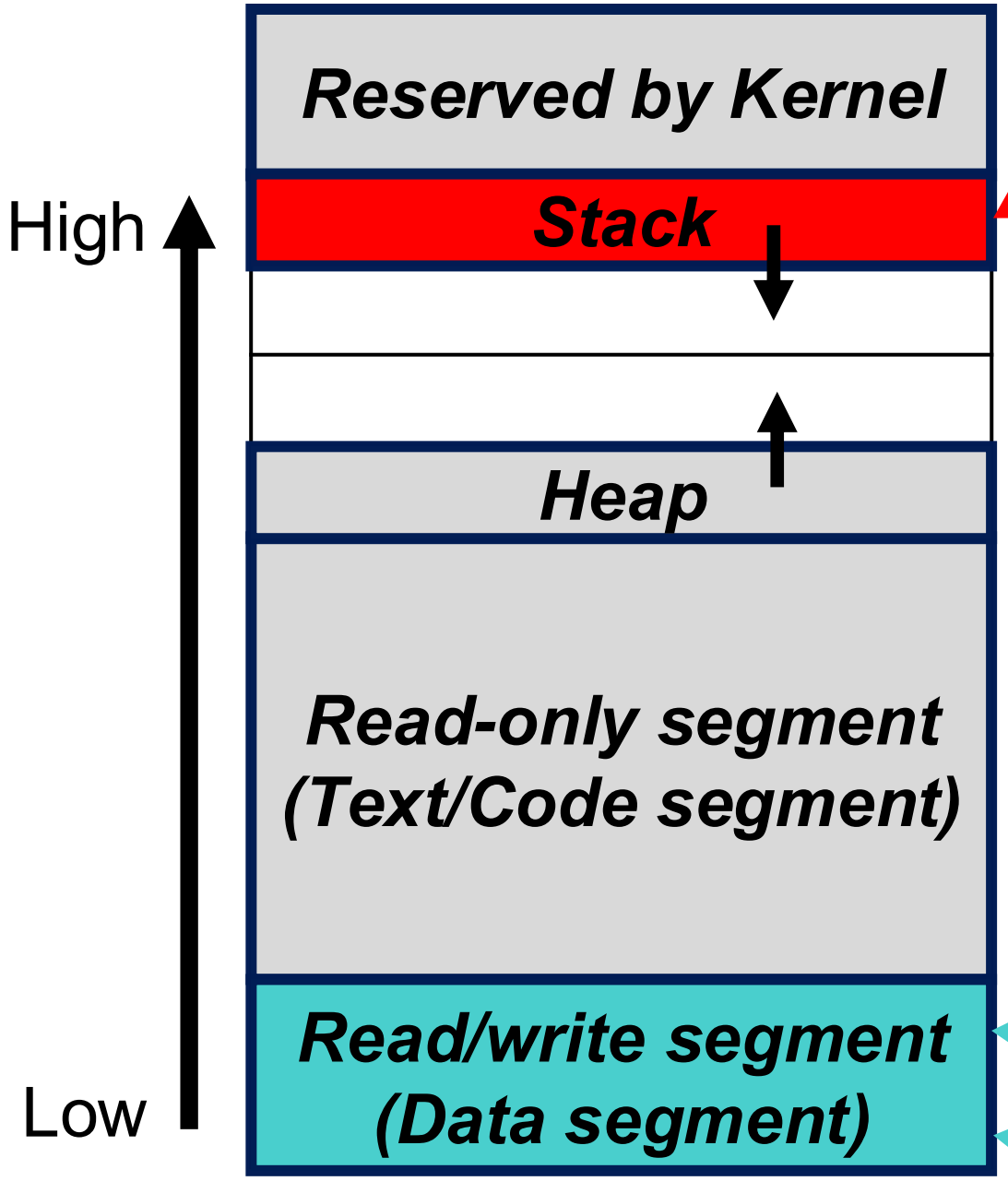



```
int A = 10;
int B;

int main () {
  int local;
  int *p;
  char *ptr = “Ube”
  char str[5] = “Miso”
  p = (int*) malloc(40);
}
```



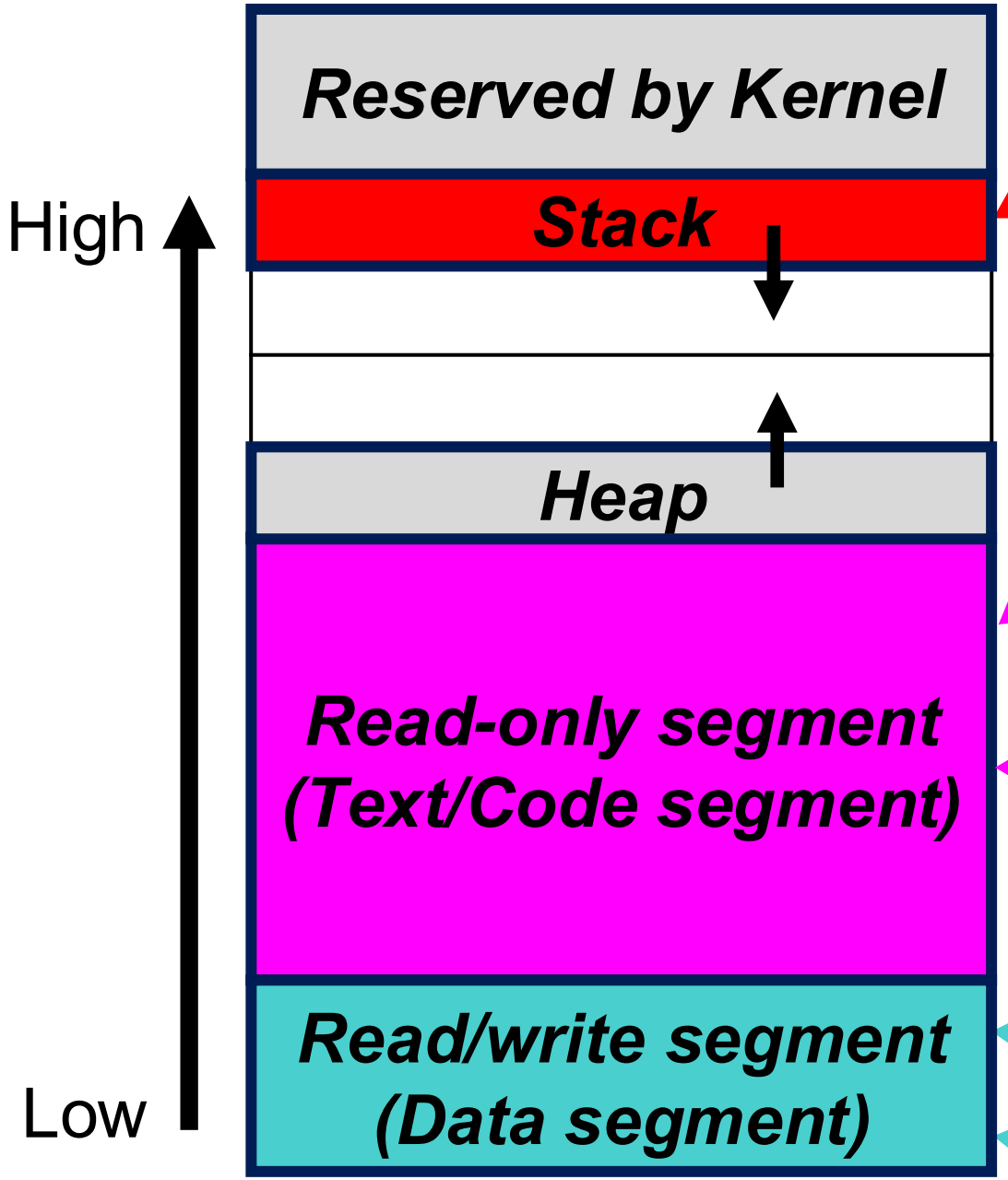

```
int A = 10;
int B;

int main () {
   int local;
   int *p;
   char *ptr = "Ube"
   char str[5] = "Miso"
   p = (int*) malloc(40);
}
```

Initialized & uninitialized global variables




```
int A = 10;
int B;

int main () {
   int local;
   int *p;
   char *ptr = “Ube”
   char str[5] = “Miso”
   p = (int*) malloc(40);
}
```

Initialized & uninitialized global variables

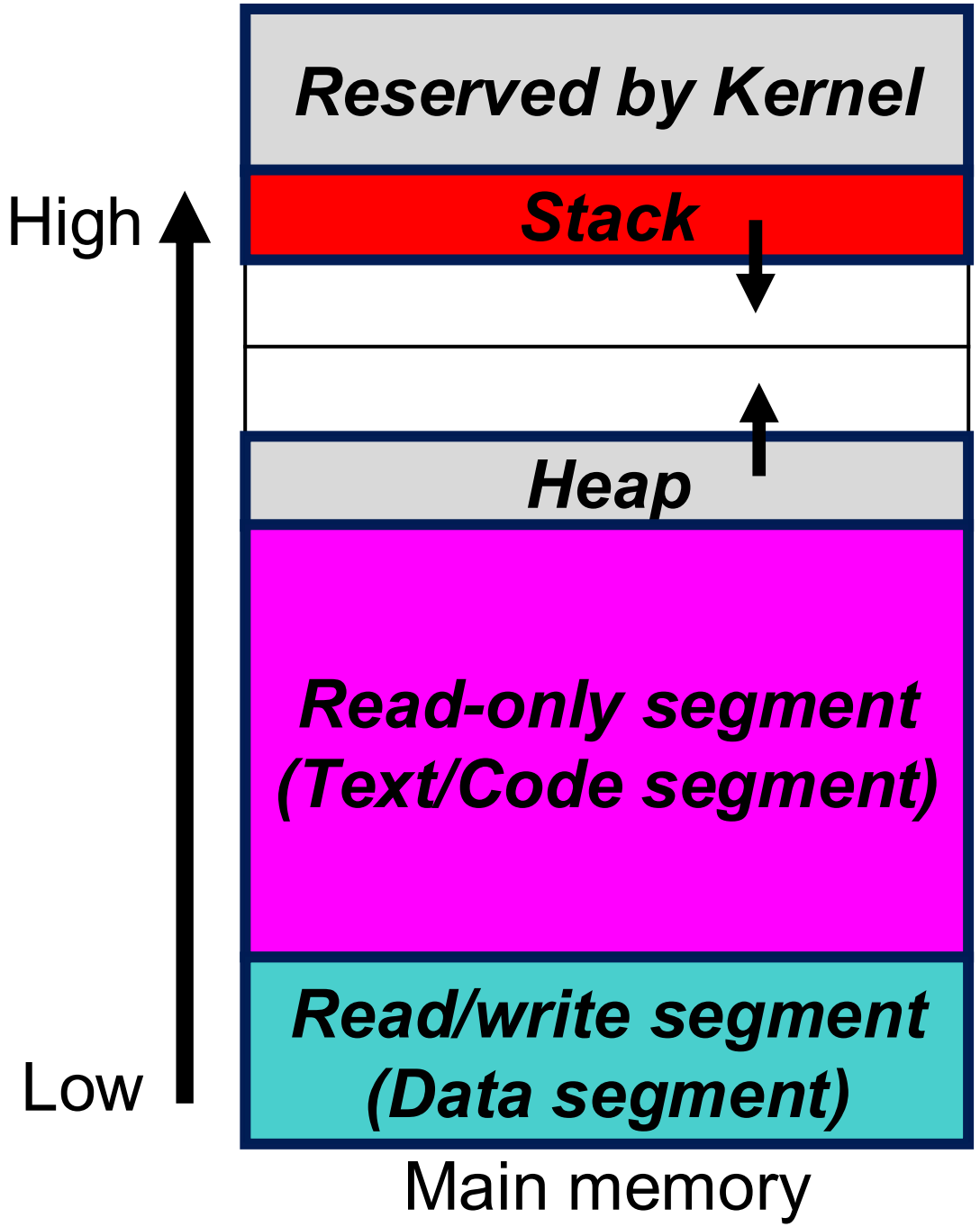



```
int A = 10;
int B;

int main () {
  int local;
  int *p;
  char *ptr = "Ube"
  char str[5] = "Miso"
  p = (int*) malloc(40);
}
```

Initialized & uninitialized global variables

.rodata

Assembly

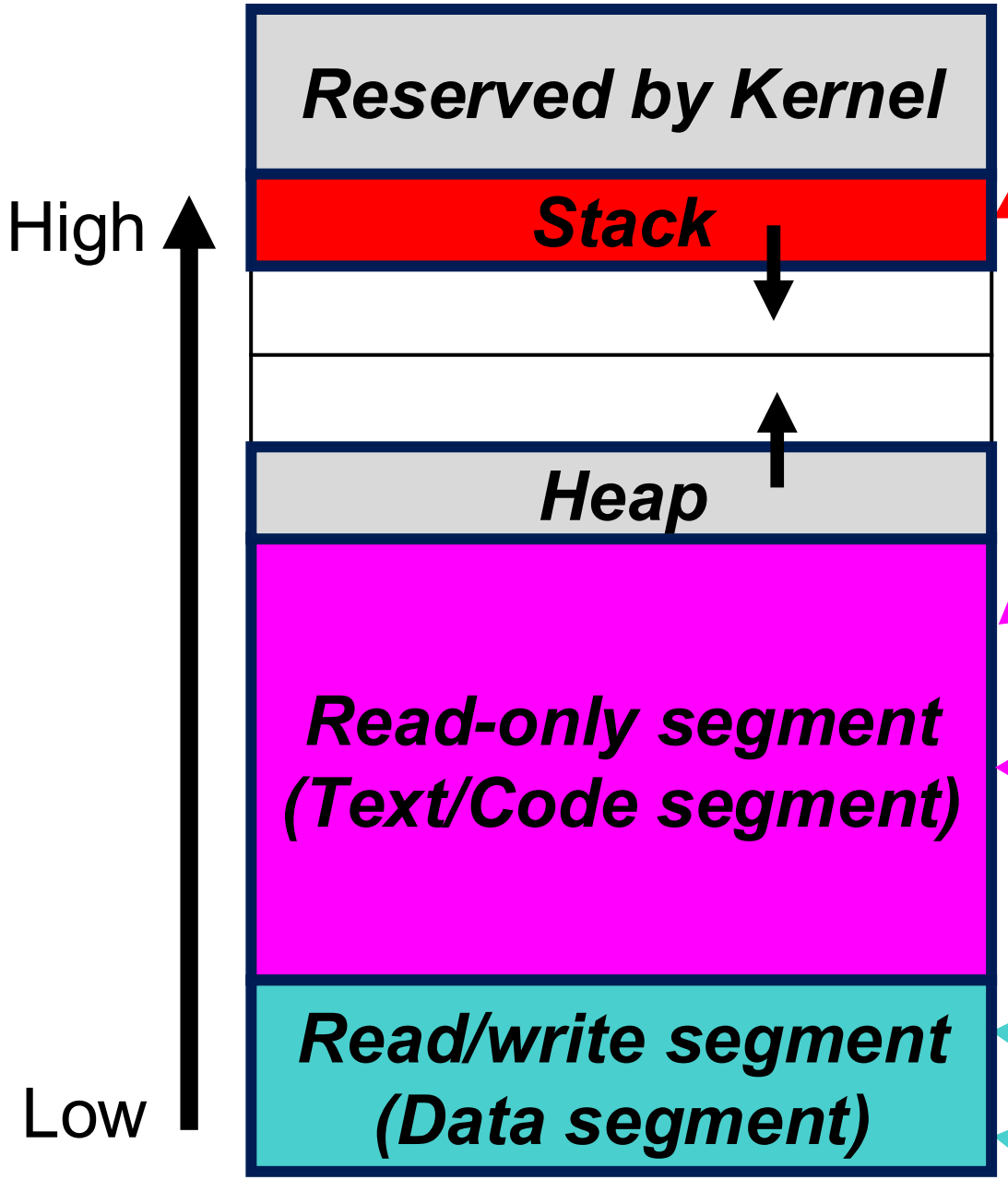



```
int A = 10;
int B;
```

char * is an 8-byte pointer that stores an address of a character

```
int *p;
char *ptr = "Ube"
char str[5] = "Miso"
int*) malloc(40);
```

char[] is an array of a characters that stores the actual characters in a string

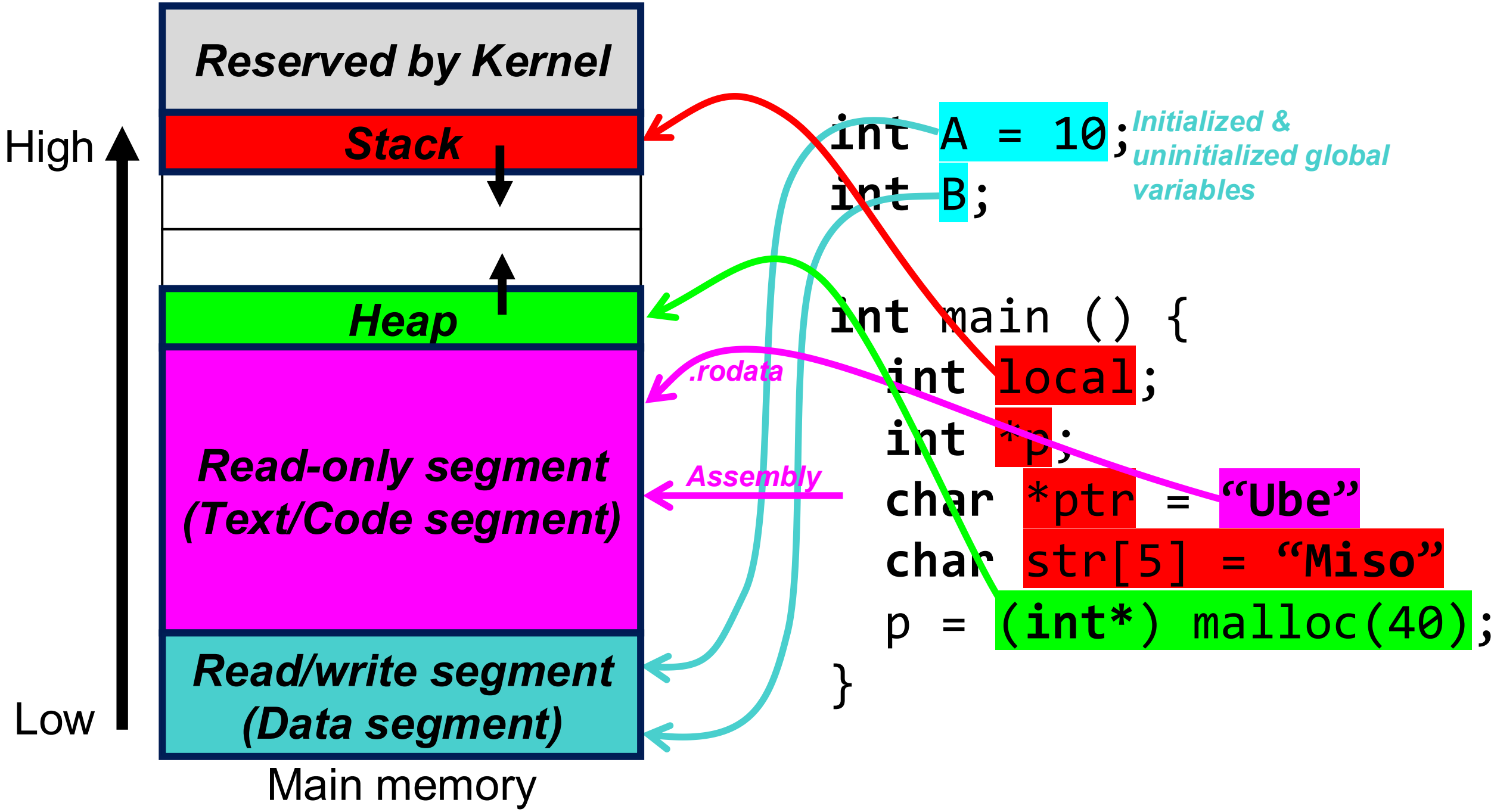



```
int A = 10;
int B;

int main () {
  int local;
  int *p;
  char *ptr = "Ube"
  char str[5] = "Miso"
  p = (int*) malloc(40);
}
```

Initialized & uninitialized global variables

.rodata

Assembly


Reserved by Kernel
Stack
Heap
Read-only segment
(Text/Code segment)
Read/write segment
(Data segment)
High
Low
Main memory
.rodata
Assembly
int A = 10;
int B;
int main () {
int local;
int *p;
char *ptr = “Ube”
char str[5] = “Miso”
p = (int*) malloc(40);
}
Initialized &
uninitialized global
variables

Layout Caveats

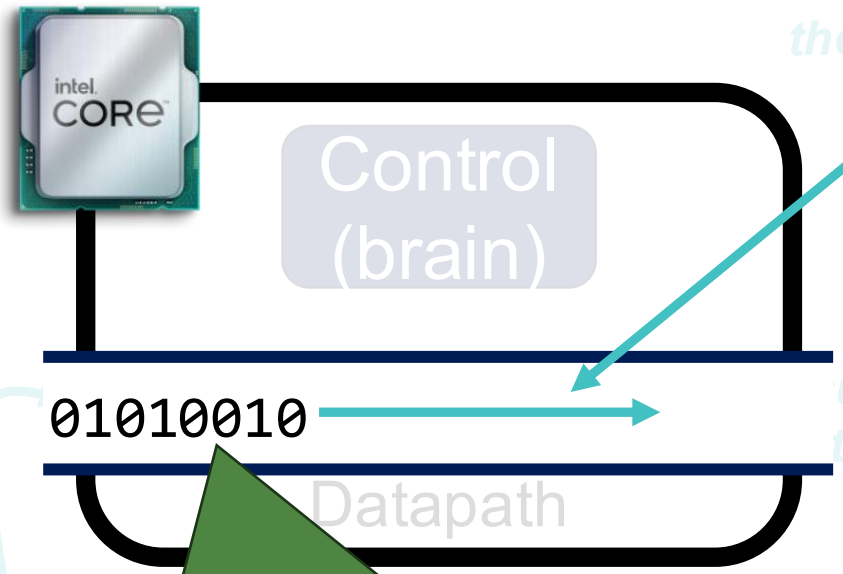
- Modern systems also:
  - Map shared libraries, which are code used by a program that do not appear in its ELF file
  - Randomize the location of the mapped sections

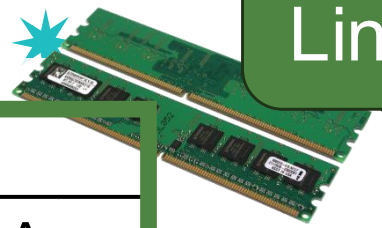
- However, the logical layout remains the same

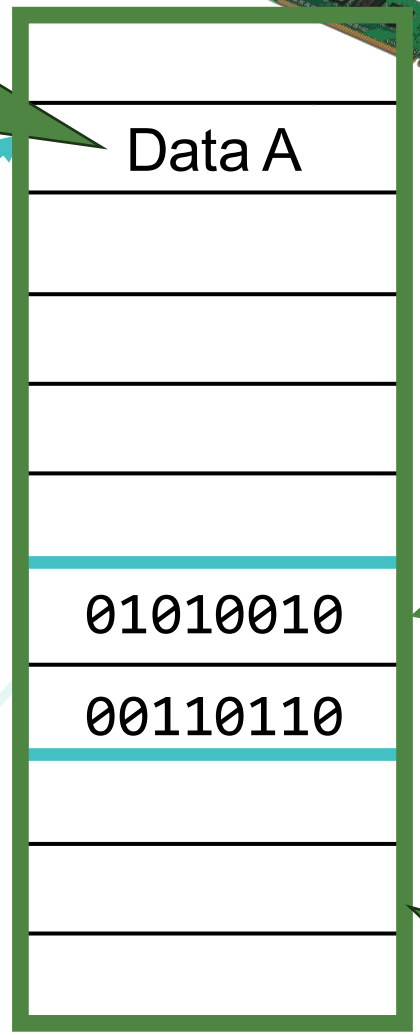
Conclusion

How data (bit, byte, int, …) is represented in memory?

Executable and Linkable Format (ELF)

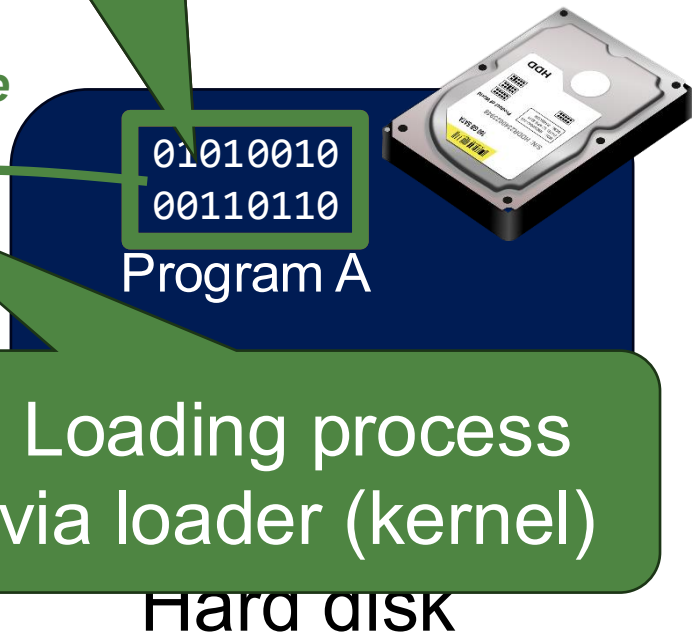






Main memory

1. Load the Program A



Hard disk

Loading process via loader (kernel)

How is assembly code interpreted, and how does it interact with memory?

Stack, heap, code, data regions

Question?