K22 – Operating Systems: Design Principles and Internals

Fall 2025 @dit

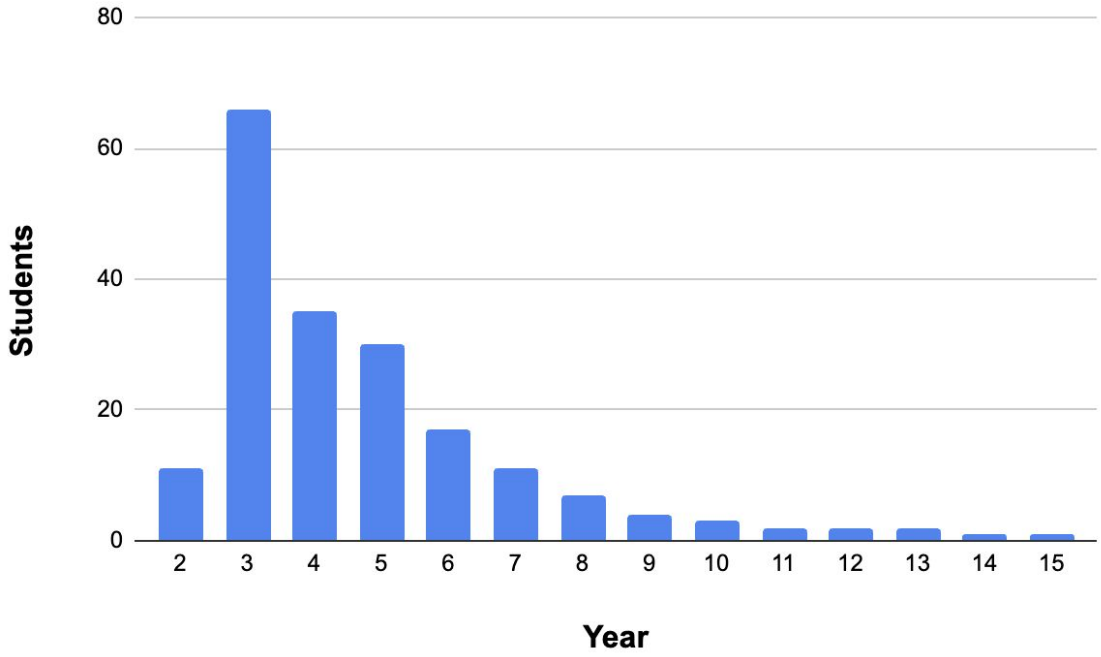
Vaggelis Atlidakis

Lecture 04

References: Similar OS courses @Columbia, @Stanford, @UC San Diego, @Brown, @di (previous years); and textbooks: Operating Systems: Three Easy Pieces, Operating Systems: Principles and Practice, Operating System Concepts, Linux Kernel Development, Understanding the Linux Kernel

Administrivia

- Where we are today? ~200 ppl.



A few clarifications

- Difficult class…does not mean difficult to pass
  - Forget about the grade and make an honest effort
  - Socialize during the semester, and share your "suffering"
  - What do you need a good grade without a support network?
  - Be kind in your perception about others!
- Help me out with the team matching, please
- More support (labs) coming soon…
- So far: We are on-par with any top tier uni offering OSes!

DISCLAIMER

- EMAILs requesting some special "recipe" because for reason X or Y you cannot participate in the programming assignments

  - I respect you reasons, but...
  - It is impossible to do something both convenient for you and also fair for the majority
  - If you wish to pass this class w/o the programming assignments, you should come on September
    - Final exam: **100% credit**
    - Programming assignment requirement: **None**

- I will reply to no more such emails.

Overview

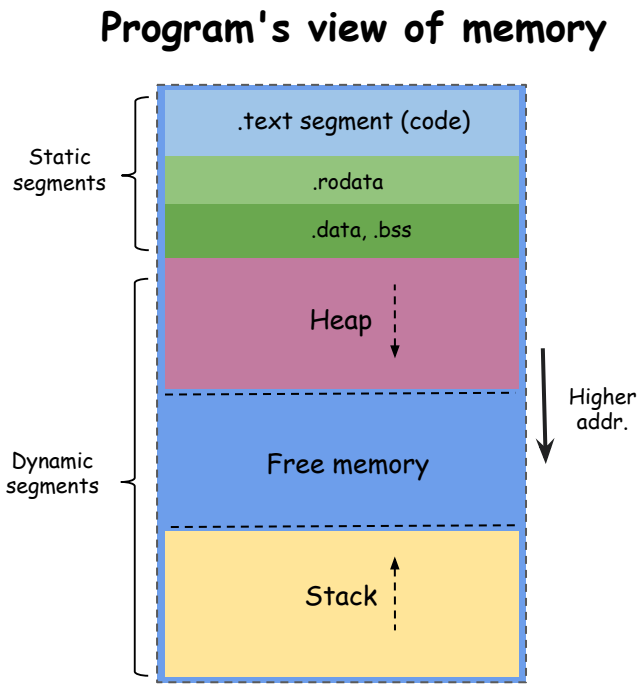
- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - Runtime [Q: How does a program look like in memory?]
  - Processes [Q: What is a process?]
  - IPC [Q: How do processes communicate?]
  - Threads [Q: What is a thread?]
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

Overview

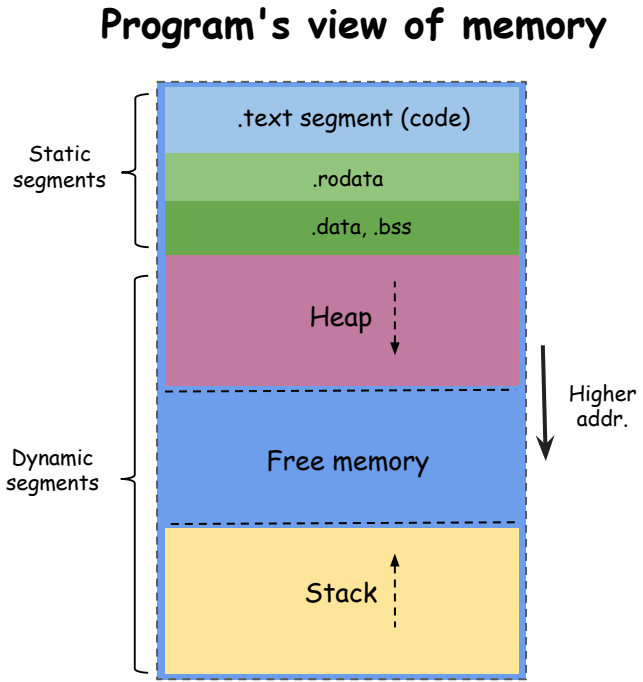
- ~~Intro [Q: What is an OS?]~~
- ~~Events [Q: When does the OS run?]~~
- Runtime [Q: How does a program look like in memory?]
  - Q1: How does a program look like in memory?
  - Q2: Who sets up a program in memory? and how?
  - Q3: Static vs dynamic linking?

...

How does a program look like in memory?

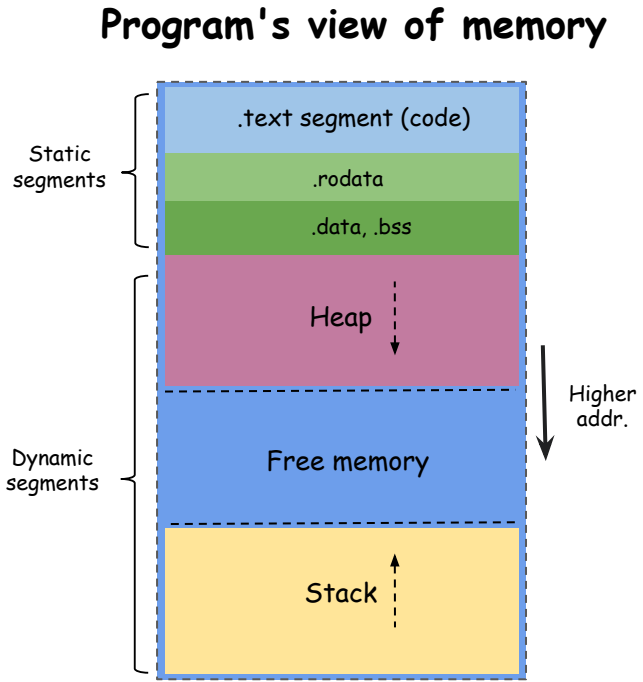


How does a program look like in memory?



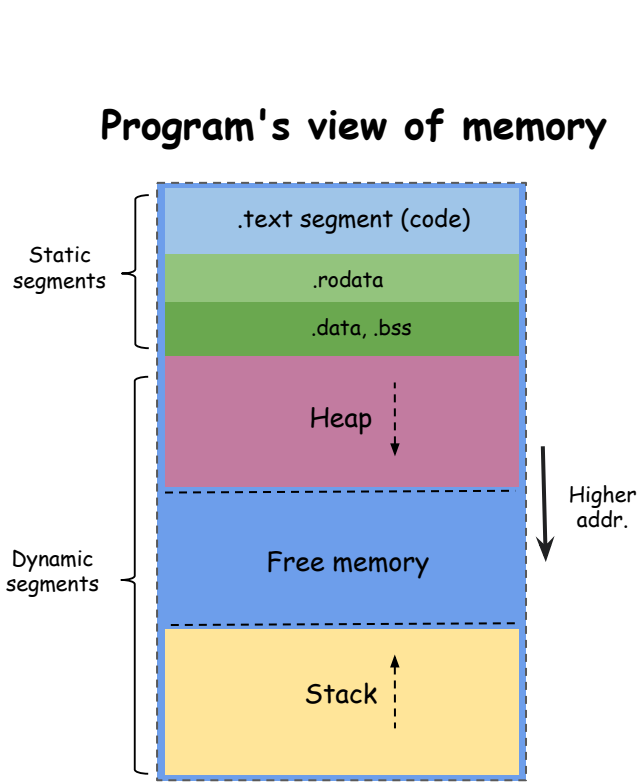
- ***Static Segments***: *Their size is static during runtime*
  - **.text**: Executable instr. of the program—read-execute perms
  - **.rodata**: Constant values—read-only perms
  - **.data**: Initialized global and static variables—read-write perms
  - **.bss**: Uninitialized global and static variables—read-write perms

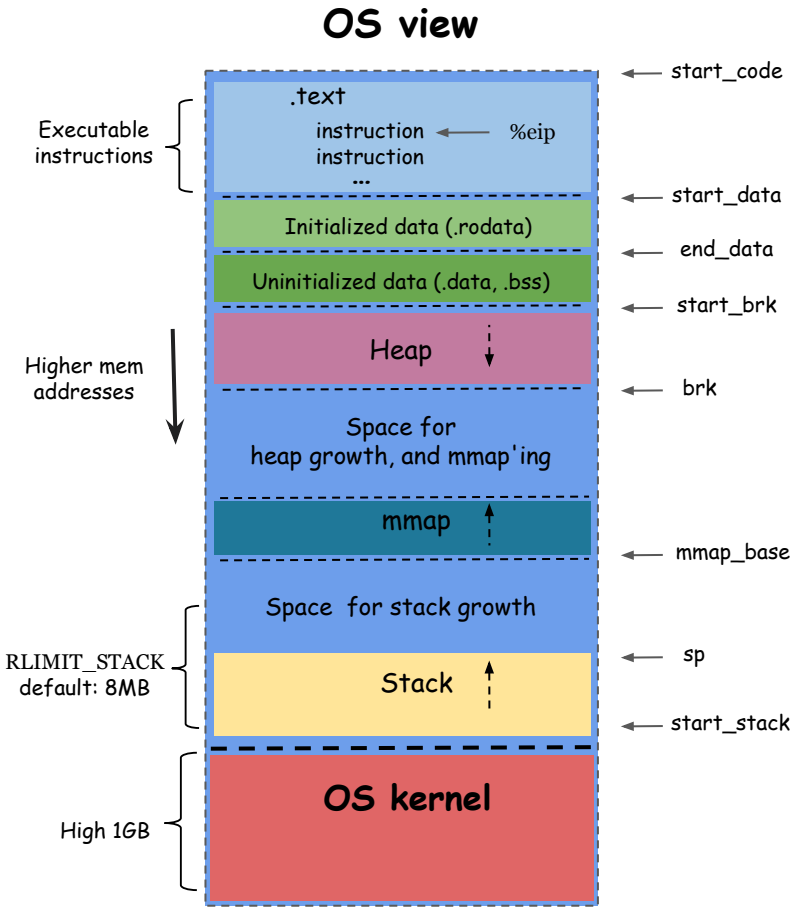
How does a program look like in memory?



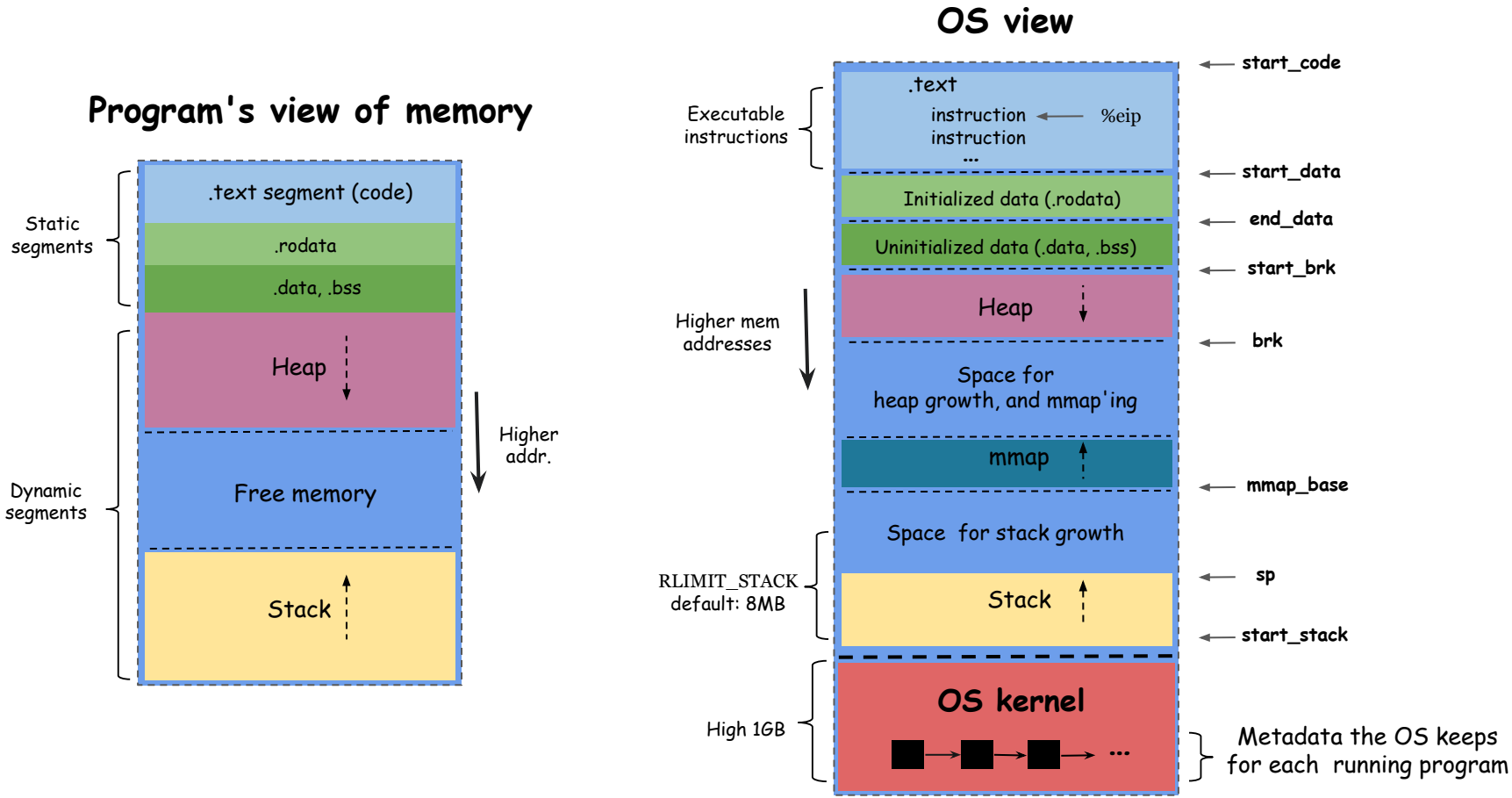
- ***Static Segments***: *Their size is static during runtime*
  - .text: Executable instr. of the program—read-execute perms
  - .rodata: Constant values—read-only perms
  - .data: Initialized global and static variables—read-write perms
  - .bss: Uninitialized global and static variables—read-write perms
- ***Dynamic segments***: *Their size can grow during runtime*
  - Heap: Grows (commonly) towards higher addresses and contains variables dynamically allocated—read-write perms
  - Stack: Grows (commonly) towards lower addresses and is used for bookkeeping during function calls—read-write perms

How does a program look like in memory?





How does a program look like in memory?



How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
  unsigned long sp;
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
  unsigned long sp;
  printf("@.rodata variable: %p\n", message);
  printf("@.bss variable: %p\n", &uninitialized_global);
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  char *heap = (char *)malloc(50 * sizeof(char));
  printf("@heap variable: %p\n", heap);
  printf("@main / Current sp: %lx\n", sp);

  foo();
}
```

How does a program look like in memory?

```c
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
  unsigned long sp;
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
  unsigned long sp;
  printf("@.rodata variable: %p\n", message);
  printf("@.bss variable: %p\n", &uninitialized_global);
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  char *heap = (char *)malloc(50 * sizeof(char));
  printf("@heap variable: %p\n", heap);
  printf("@main / Current sp: %lx\n", sp);

  foo();
}
```

```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p   …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p   …  [heap]
ffffbe610000-ffffbe798000   r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000   ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000   r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000   rw-p     … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000   rw-p     …
fffff6461000-fffff6482000   rw-p     … [stack]

➜  ./sample
@ .rodata variable: 0xaaaadcdb0100
```

How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
  unsigned long sp;
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
  unsigned long sp;
  printf("@.rodata variable: %p\n", message);
  printf("@.bss variable: %p\n", &uninitialized_global);
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  char *heap = (char *)malloc(50 * sizeof(char));
  printf("@heap variable: %p\n", heap);
  printf("@main / Current sp: %lx\n", sp);

  foo();
}
```

```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000   r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000   ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000   r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000   rw-p      … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000   rw-p      …
fffff6461000-fffff6482000   rw-p     … [stack]

➜  ./sample
@ .rodata variable: 0xaaaadcdb0100
@ .bss  variable: 0xaaaadcdb1020
```

How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
   unsigned long sp;
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
   unsigned long sp;
   printf("@.rodata variable: %p\n", message);
   printf("@.bss variable: %p\n", &uninitialized_global);
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   char *heap = (char *)malloc(50 * sizeof(char));
   printf("@heap variable: %p\n", heap);
   printf("@main / Current sp: %lx\n", sp);

   foo();
}
```

```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000   r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000   ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000   r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000   rw-p      … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000   rw-p      …
fffff6461000-fffff6482000   rw-p     … [stack]
```

```
➜  ./sample
@ .rodata variable: 0xaaaadcdb0100
@ .bss  variable: 0xaaaadcdb1020
@ heap variable: 0xaaaaed20d6b0
```

How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
   unsigned long sp;
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
   unsigned long sp;
   printf("@.rodata variable: %p\n", message);
   printf("@.bss variable: %p\n", &uninitialized_global);
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   char *heap = (char *)malloc(50 * sizeof(char));
   printf("@heap variable: %p\n", heap);
   printf("@main / Current sp: %lx\n", sp);

   foo();
}
```

```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000   r-xp      … /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000   ---p      … /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000   r--p      … /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000   rw-p      … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000   rw-p      …
ffffff6461000-ffffff6482000   rw-p    … [stack]

➜  ./sample
@ .rodata variable: 0xaaaadcdb0100
@ .bss  variable: 0xaaaadcdb1020
@ heap variable: 0xaaaaed20d6b0
@ main – Current sp: ffffff64810e0
```

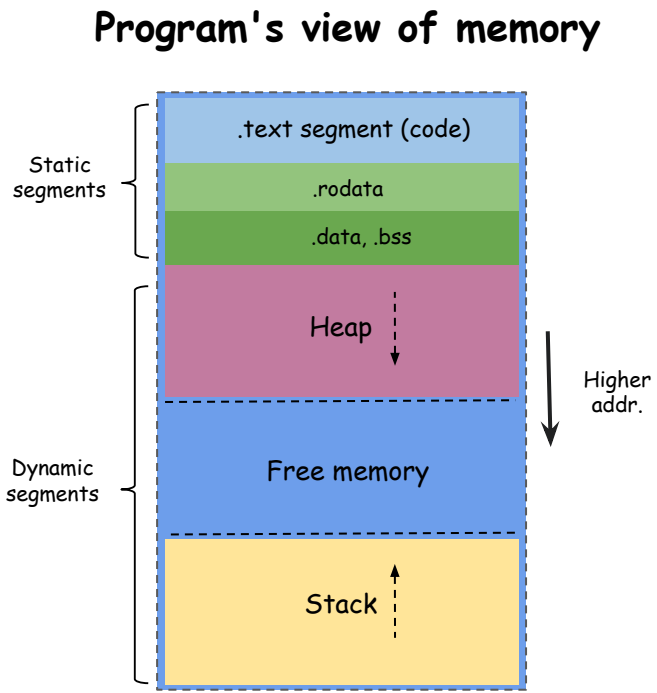
How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
   unsigned long sp;
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
   unsigned long sp;
   printf("@.rodata variable: %p\n", message);
   printf("@.bss variable: %p\n", &uninitialized_global);
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   char *heap = (char *)malloc(50 * sizeof(char));
   printf("@heap variable: %p\n", heap);
   printf("@main / Current sp: %lx\n", sp);

   foo();
}
```

```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000   r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000   ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000   r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000   rw-p      … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000   rw-p      …
ffffff6461000-ffffff6482000   rw-p    … [stack]
```

```
➜  ./sample
@ .rodata variable: 0xaaaadcdb0100
@ .bss  variable: 0xaaaadcdb1020
@ heap variable: 0xaaaaed20d6b0
@ main – Current sp: ffffff64810e0
@ foo   – Current sp: ffffff64810c0
```

How does a program look like in memory?




```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000  r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000  ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000  r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000  rw-p     … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000  rw-p     …
fffff6461000-fffff6482000  rw-p     … [stack]

➜  ./sample
@ .rodata variable: 0xaaaadcdb0010
@ .bss  variable: 0xaaaadcdb1020 > 0xaaaadcdb0010
@ heap variable: 0xaaaaed20d6b0
@ main – Current sp: fffff64810e0
@ foo   – Current sp: fffff64810c0 < fffff64810e0 (main's sp)
```

Who sets up the program in memory?

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
    - We need a specification to serve as the contract (interface) between executables and the loader
    - **ELF**: The Executable and Linkable Format (ELF)

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
    - The static segments are initialized by copying from the ELF
    - The default dynamic segments are laid out by the loader, and the OS intervenes to manage them

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELF's entry point (e.g., _start)

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELF's entry point (e.g., _start)
    - If the executable is statically linked, the loader's job is complete

Static linking

hello.c

```
extern void foo(int);

int bar(int a){
   a += 1;
   return a;
}

void main(void){
   foo(1);
   bar(2);
}
```

➜ gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜ readelf -s hello.o

```
# Table of all global, exported symbols with offset and  section number
Num:    Value              Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0    SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000     0    SECTION LOCAL  DEFAULT    3 .data
  4: 0000000000000000     0    SECTION LOCAL  DEFAULT    4 .bss
  …
 10: 0000000000000000    32    FUNC    GLOBAL DEFAULT    1 bar
 11: 0000000000000020    36    FUNC    GLOBAL DEFAULT    1 main
 12: 0000000000000000     0    NOTYPE  GLOBAL DEFAULT  UND foo
```

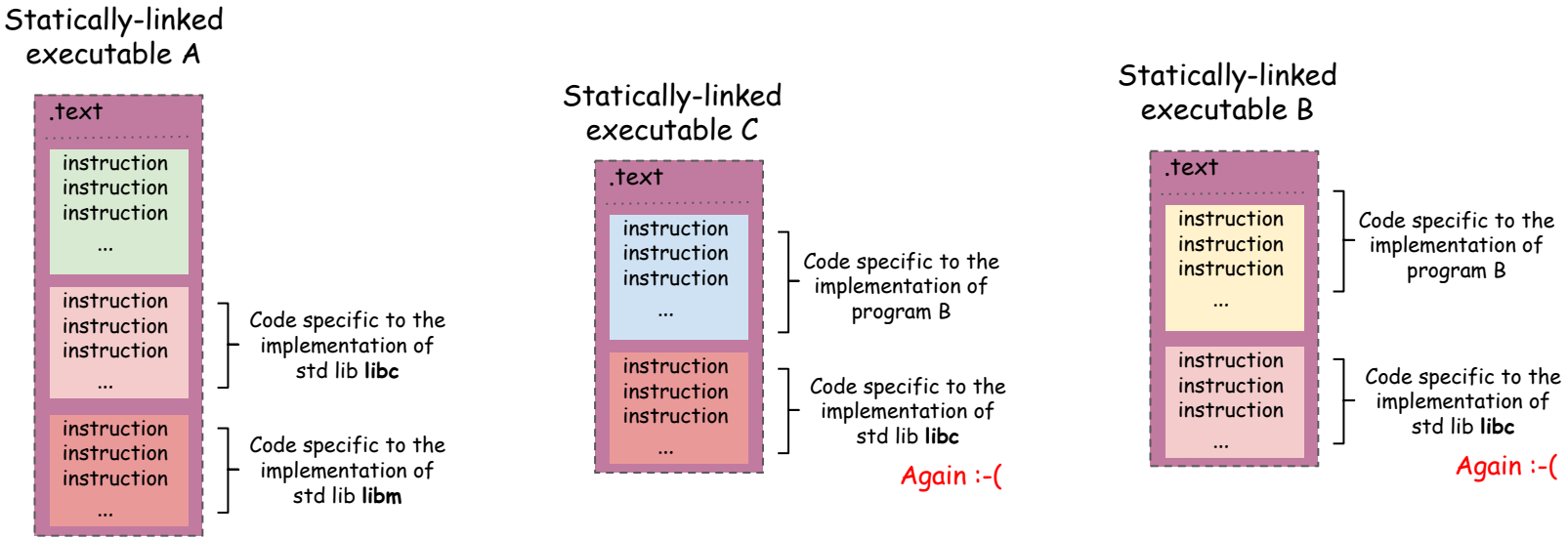
➜ nm hello.o

```
# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main
```

➜ gcc -o exe foo.o hello.o
➜ nm exe

```
# All relocations from statically-linked objects are resolved
…
0000000000000758  T foo
0000000000000714  T bar
0000000000000734  T main
```

Static linking: The problem



Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELFs entry point (e.g., _start)
    - If the executable is statically linked, the loader's job is complete
    - If the executable is dynamically linked, we need more…

Dynamic linking

```
➜ gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜ readelf -s hello.o
```

```
# Table of all global, exported symbols with offset and section number
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	.text
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	.data
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	.bss
…							
10:	0000000000000000	32	FUNC	GLOBAL	DEFAULT	1	bar
11:	0000000000000020	36	FUNC	GLOBAL	DEFAULT	1	main
12:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	foo

```
➜ nm hello.o
```

```
# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main
```

```
➜ gcc -o exe foo.o hello.o
➜ nm exe
```

```
# All relocations from statically-linked objects are resolved
…
0000000000000758  T foo
0000000000000714  T bar
0000000000000734  T main
```

Dynamic linking

- What if we know how to find this at runtime?

```
➜ gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜ readelf -s hello.o

# Table of all global, exported symbols with offset and section number
Num:    Value              Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0    SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000     0    SECTION LOCAL  DEFAULT    3 .data
  4: 0000000000000000     0    SECTION LOCAL  DEFAULT    4 .bss
  …
 10: 0000000000000000    32    FUNC    GLOBAL DEFAULT    1 bar
 11: 0000000000000020    36    FUNC    GLOBAL DEFAULT    1 main
 12: 0000000000000000     0    NOTYPE  GLOBAL DEFAULT  UND foo

➜ nm hello.o

# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main

➜ gcc -o exe foo.o hello.o
➜ nm exe

# All relocations from statically-linked objects are resolved
…
0000000000000758 T foo
0000000000000714 T bar
0000000000000734 T main
```

Dynamic linking

- What if we know how to find this at runtime?
- Well…another drama starts…

```
➜  gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜  readelf -s hello.o

# Table of all global, exported symbols with offset and  section number
Num:    Value          Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .data
  4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .bss
  …
 10: 0000000000000000    32 FUNC    GLOBAL DEFAULT    1 bar
 11: 0000000000000020    36 FUNC    GLOBAL DEFAULT    1 main
 12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo

➜  nm  hello.o

# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main
```

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELFs entry point (e.g., _start)
    - If the executable is statically linked, the loader's job is complete
    - If the executable is dynamically linked we need more…
  - **The dynamic linker resolves symbols at runtime**
    - Specified in the .interp section of the ELF (e.g., ld-linux.so)
    - Yet another time we deferred stuff for later

Dynamic Linking / Loading