CS 4120
Introduction to Compilers

Andrew Myers

Cornell University

Lecture 40: Shared libraries
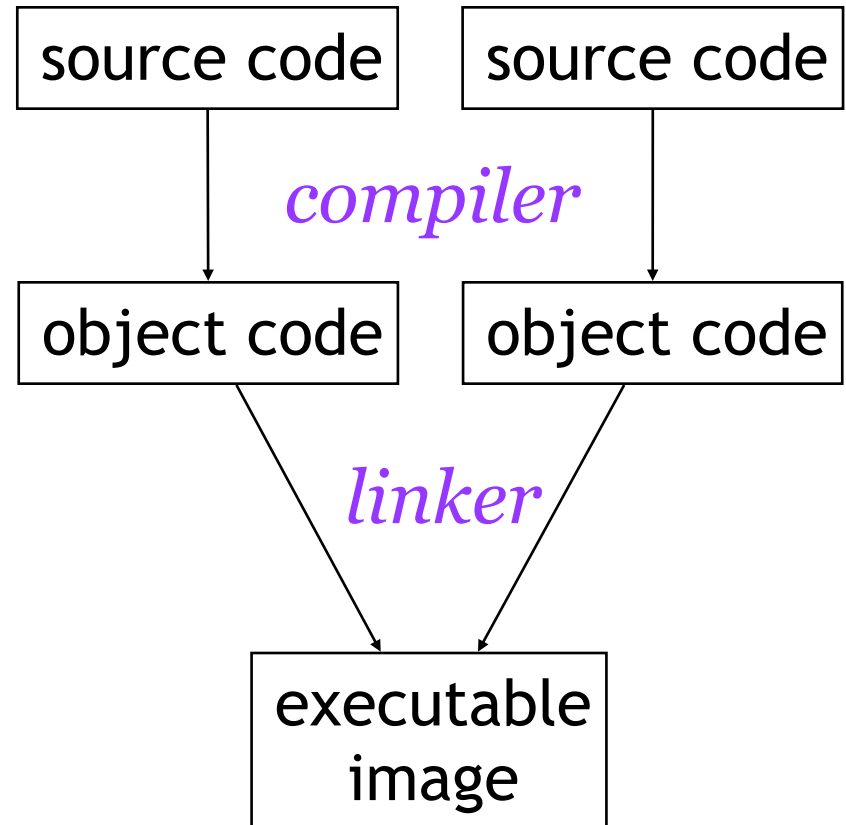and dynamic loading

9 May 2022

Wrap-up on GC

- Last time: memory management and garbage collection

- Takeaway: hidden cost and complexity!
  - 30–50% space overhead from fragmentation
  - scanning/compacting overhead
  - read-barrier or write-barrier overhead on pointers for copying & generational collection.

Object files

- Output of assembler is an **object file**
  - not executable
  - may refer to external symbols (variables, functions, etc.) whose definition is not known.

- Linker joins together object files, resolving external references

```
source code        source code

        compiler

object code        object code

        linker

      executable
        image
```

Unresolved references

source code

```
extern long
abs( int x );
…
y = y + abs(x);
```

assembly code

```
push rcx
call _abs
add rdx, rax
```

object code

51				
E8	00	00	00	00
01	C2			

filled in by linker

Object file structure

file header

text section: unresolved machine code

initialized data

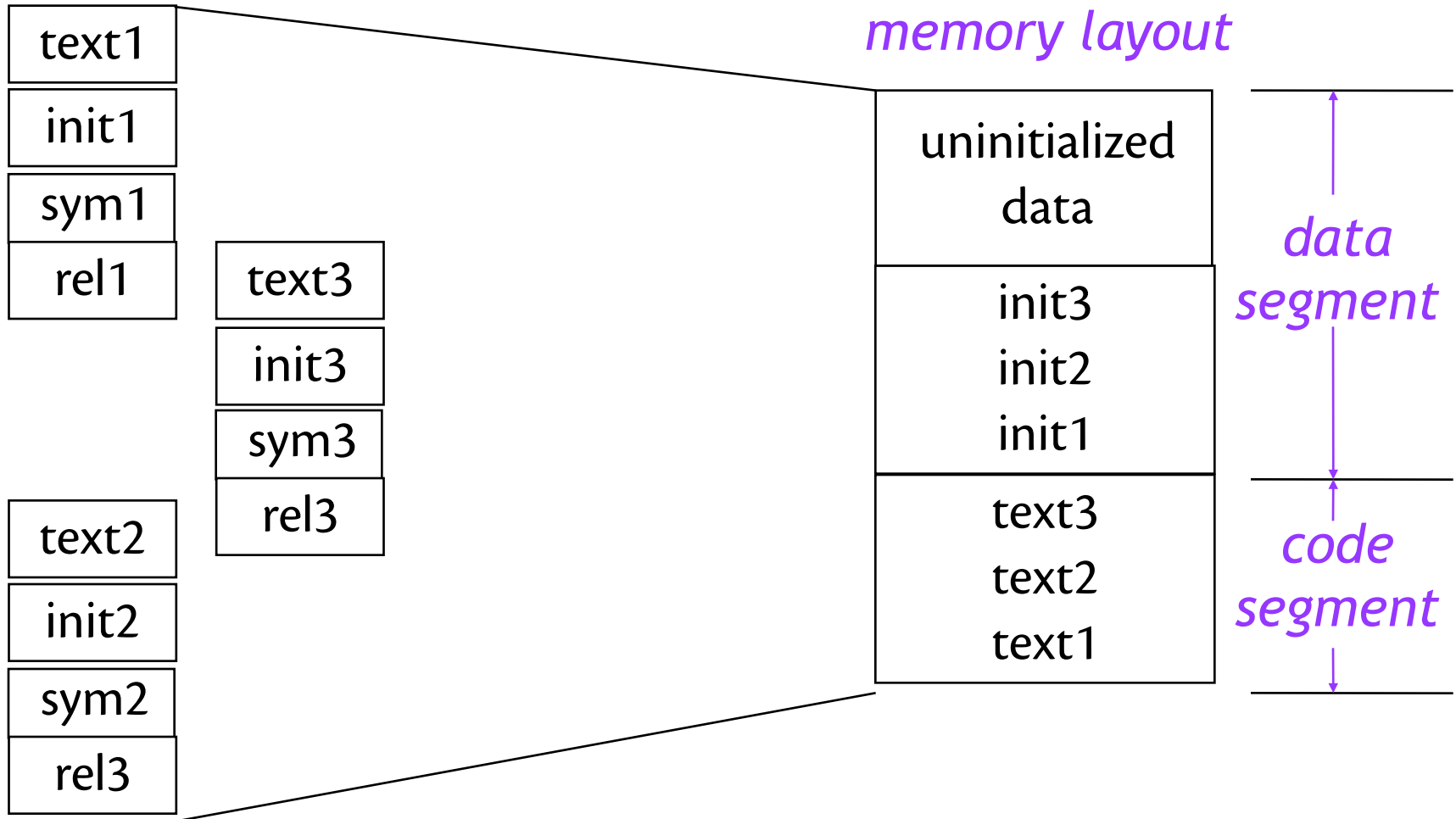| |
|---|
| symbol table (maps identifiers to machine code locations) |

relocation info
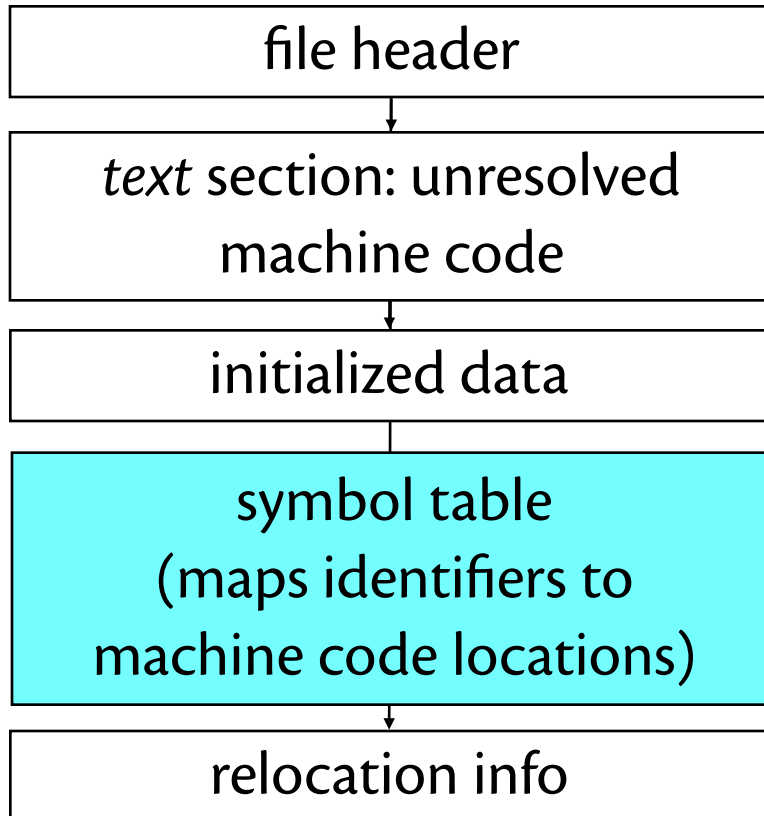
- Object file contains various **sections**
- **text** section contains the compiled code with some patching needed
- For uninitialized data, only need to know total *size* of data segment
- Describes structure of text and data sections
- Points to places in text and data section that need fix-up

Linker output

object files

executable image memory layout

text1
init1
sym1
rel1

text3
init3
sym3
rel3

text2
init2
sym2
rel3

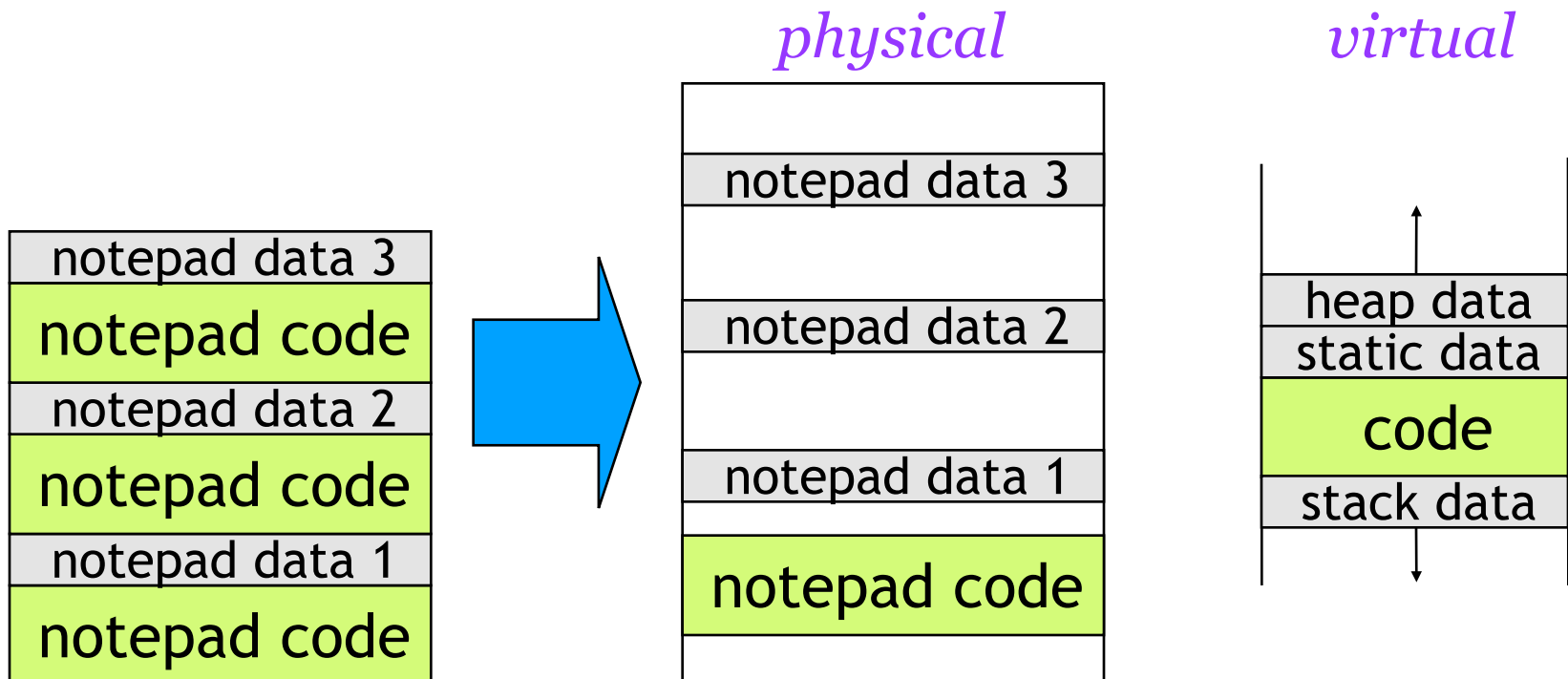| uninitialized data |
|---|
| init3 init2 init1 |
| text3 text2 text1 |

data segment

code segment

Executable file structure

- Same as object file, but ready to be executed as-is

- Pages of code and data brought in lazily from text and data section
  $\Rightarrow$ rapid start-up

- Text section shared across processes

- Symbols for debugging (global, stack frame layouts, line numbers, etc.)

file header

text section: execution-ready machine code

initialized data

optional: symbol table
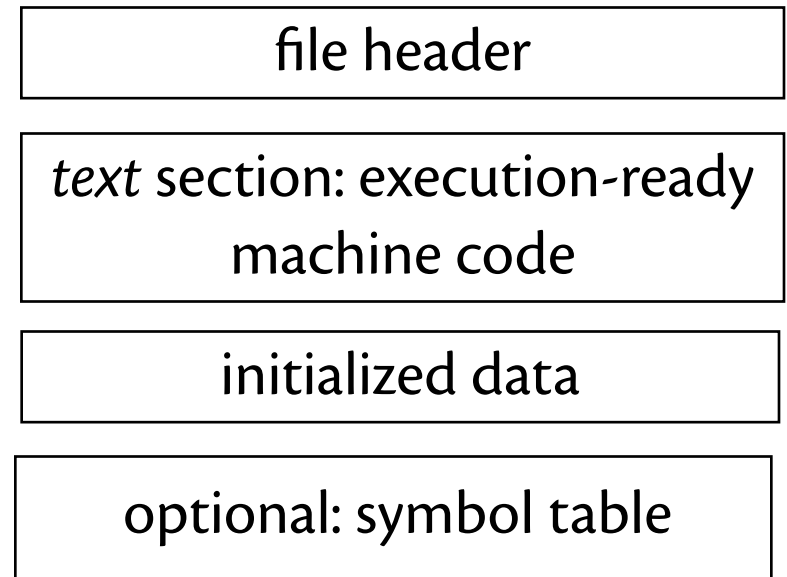
Executing programs

- Multiple copies of program share code (text), have own data

- Data appears at same virtual address in every process

physical　　　　　　*virtual*

| notepad data 3 |
| notepad code |
| notepad data 2 |
| notepad code |
| notepad data 1 |
| notepad code |

| |
| notepad data 3 |
| |
| notepad data 2 |
| |
| notepad data 1 |
| notepad code |
| |

| heap data |
| static data |
| code |
| stack data |

Libraries

- *Library* : collection of object files

- Linker adds all object files necessary to resolve undefined references in explicitly named files

- Object files, libraries searched in user-specified order for external references

  ld main.o foo.o /usr/lib/X11.a /usr/lib/libc.a

- Library has index over all object files for rapid searching

Shared libraries

- Problem: libraries take up a lot of memory when linked into many running applications

- Solution: *shared libraries (e.g. DLLs)*

Physical memory
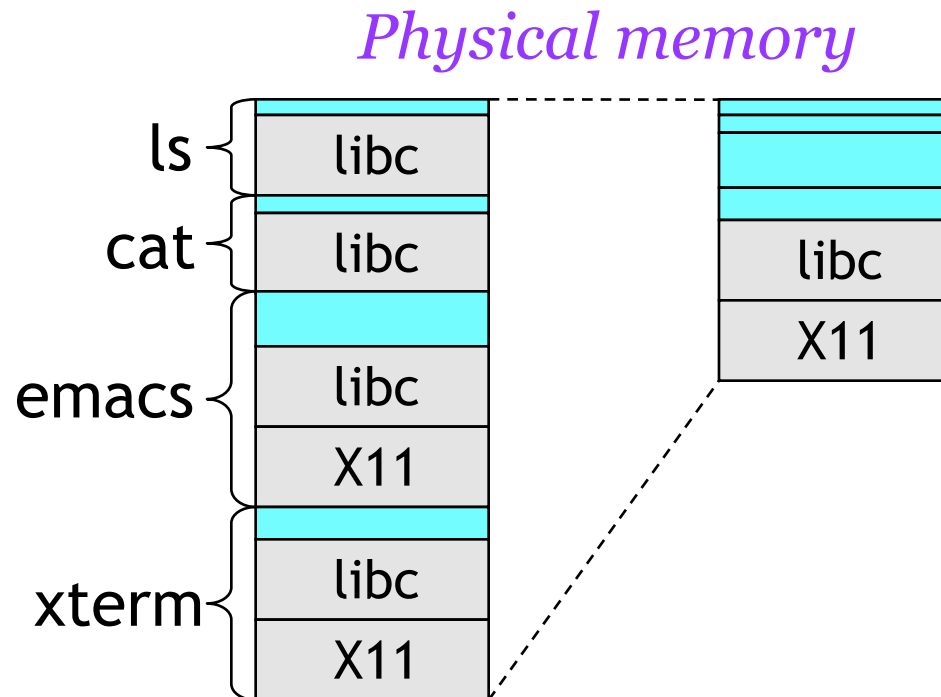
Step 1: Jump tables

- Executable file does not contain library code; library code loaded dynamically.

- Library code found in separate shared library file (similar to DLL); linking done against **import library** that does not contain code.

- Library compiled at fixed address, starts with **jump table** to allow new versions; application code jumps to jump table (indirection).

    – jump table doesn't move ⇒ library can evolve.

```
program:            library:
                    scanf: jmp real_scanf

call printf         printf: jmp real_printf

                    putc: jmp real_putc
```
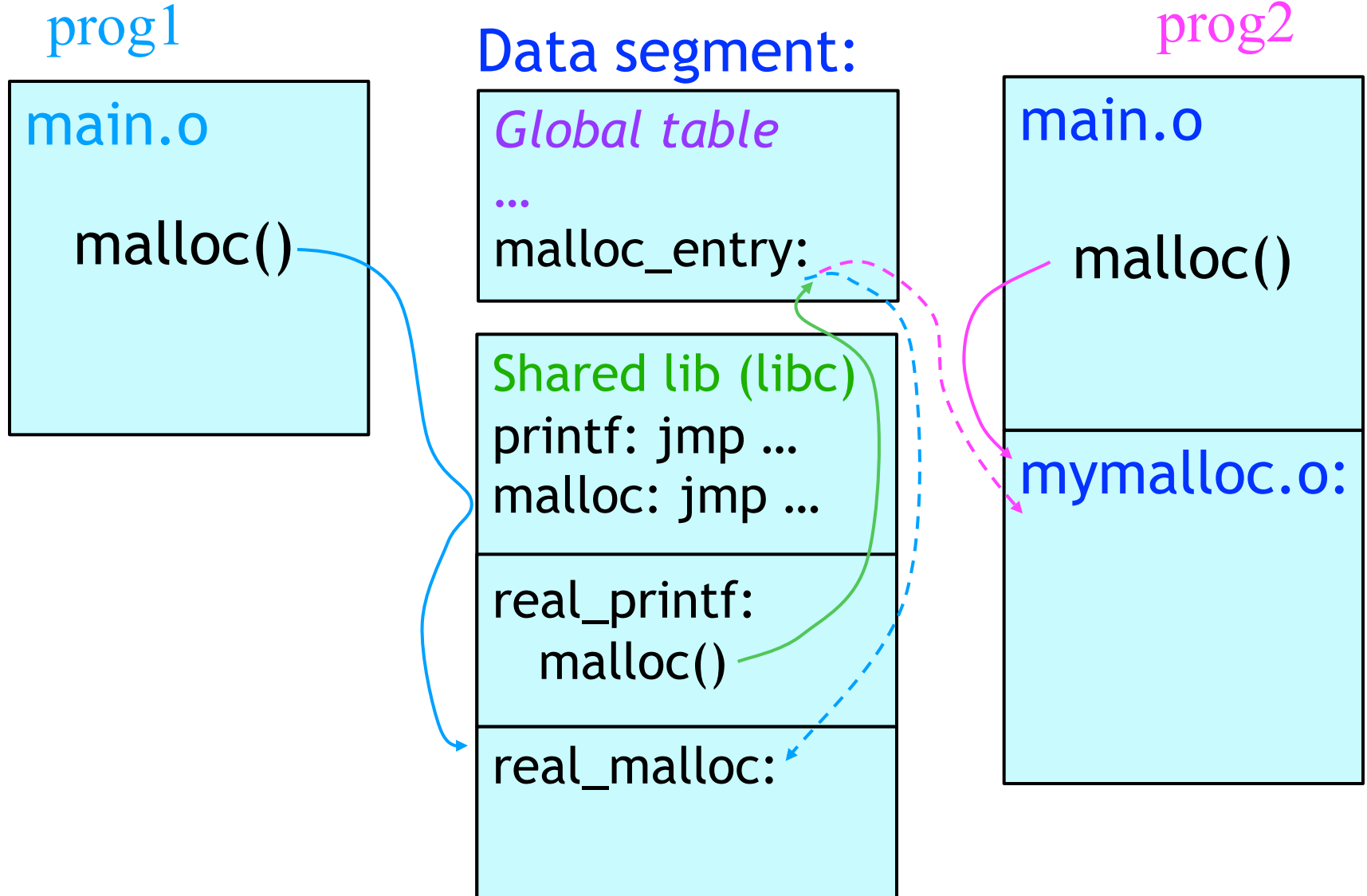
Global tables

- Problem: shared libraries may depend on external symbols (even symbols within the shared library); different applications may have different *linkage*:

```
gcc -o prog1 main.o /usr/lib/libc.a
gcc -o prog2 main.o mymalloc.o /usr/lib/libc.a
```

- If routine in `libc.a` calls `malloc()`, `prog1` should get standard version; for `prog2`, version in `mymalloc.o`

- Solution: Calls to external symbols made through **global offset tables** unique to each program, generated at dynamic load time.

Global tables

prog1

Data segment:

prog2

main.o

Global table

...

malloc_entry:

malloc()

main.o

malloc()

Shared lib (libc)

printf: jmp ...

malloc: jmp ...

mymalloc.o:

real_printf:

malloc()

real_malloc:

Using global tables

- Global table contains entries for all external references

```
malloc(n) ⇒ push [rbp + n]
             mov rax, [malloc_entry]
             call rax                      ; indirect jump
```

- Non-shared application code unaffected

- Same-object references can still be used directly

- Global table entries (`malloc_entry`) placed in non-shared memory locations so each program has different linkage

- Initialized by dynamic loader when program begins: reads symbol tables, relocation info.

- Code above may be dynamically generated as trampoline at load time

Relocation

- Before widespread support for virtual memory, code had to be **position-independent** (could not contain fixed memory addresses)

- With virtual memory, all programs could start at same address, *could* contain fixed addresses

- Problem with shared libraries (*e.g.*, DLLs): if allocated at fixed addresses, can collide in virtual memory (code, data, global tables, …)

  - Collision $\Rightarrow$ code copied and explicitly relocated

- Back to position-independent code!

Dynamic shared objects

- Unix systems: code typically compiled as a **dynamic shared object** (DSO): fully relocatable

- Shared libraries can be mapped to any address in virtual memory—no copying!

- *Questions:*

  – how to make code completely relocatable?

  – what is the performance impact?

Relocation difficulties

- No **absolute addresses** (directly named memory locations) anywhere:

  - Not in **calls** to external functions

  - Not for global variables in data segment

  - Not even for global table entries

  ```
  push [rbp + n]
  mov rax, [malloc_entry]     ;   Oops!
  call rax
  ```

- Not a problem: branch instructions, local calls: **relative addressing**

Global offset tables

- Can put address of all globals into global table
- But…can't put the global table at a fixed address: not relocatable!

- Three solutions:
  1. (Usual approach) Use address arithmetic on current program counter (rip register) to find global table. Link-time constant offset between rip and global table.
  2. Pass global table address as an extra argument: affects first-class functions (callee global table address stored in current GT)
  3. Stick global table entries into the current object's dispatch table : DT *is* the global table (ideal, but only works for OO code)

Cost of DSOs

- Call to external function f:

  call f_stub

  ...

f_stub: jmp [rip + f_offset]
- Global variable accesses:

  lea rax, [rip + v_offset]

  mov rbx, [rax]
- Calling global functions ≈ calling single-inheritance methods

- Global variables: *more* expensive than local variables

- Most benchmarks run w/o DSOs!

Prelinking/prebinding

- Idea: precompute relocation of dynamic libraries to virtual addresses to speed up load times

- Conflicting libraries assigned disjoint virtual memory regions

  $\Rightarrow$ whole-system optimization, usually done every few weeks

Dynamic linking

- Shared libraries (DLLs) and DSOs can be linked dynamically into a running program

- Normal case: implicit linking. When setting up global tables, shared libraries are automatically loaded if necessary (even *lazily*), symbols looked up & global tables created.

- Explicit dynamic linking: application can choose how to extend its own functionality

  - *Unix:* h = dlopen(filename) loads an object file into some free memory (if necessary), allows query of globals: p = dlsym(h, name)

  - *Windows:* h = LoadLibrary(filename), p = GetProcAddress(h, name)

Conclusions

- Shared libraries and DSOs allow efficient memory use on a machine running many different programs that share code.

- Improves cache, TLB performance overall.

- But hurts individual program performance: indirections through global tables, code bloat

- Globals are more expensive than they look!

- Important new functionality: dynamic extension of program.

What did we miss?

- attribute grammars
- instruction scheduling
- cache-layout optimizations
- targeting modern HW: FPGA, GPU, TPU
- matrix optimizations (tiling, etc.)
- stream optimizations (fusion)
- reflection/metaobjects
- coroutines and algebraic effects
- interpreters

Going forward

- Courses of interest:
  - CS 6120 (spring) : advanced compilers
  - CS 4110 (fall biyearly): programming languages
  - CS 6110 (spring) : advanced programming languages
- Research opportunities
- Final project due: Monday at **noon**