

Assemblers, Linkers, and Loaders

Prof. Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

See: P&H Appendix A.1-2, A.3-4 and 2.12

Administrivia

Upcoming agenda

- PA2 Work-in-Progress due yesterday, Monday, March 16th
- PA2 due next week, Thursday, March 26th
- HW2 available later today, due before Prelim2 in April
- Spring break: Saturday, March 28th to Sunday, April 5th

Academic Integrity

All submitted work must be your own

- OK to study together, ***but do NOT share soln's***
e.g. CANNOT email soln, look at screen, writ soln for others
- ***Cite your (online) sources***
- “Crowd sourcing” your problem/soln same as copying

Project groups submit joint work

- Same rules apply to projects at the group level
- Cannot use of someone else's soln

Closed-book exams, no calculators

- Stressed? Tempted? Lost?
 - Come see me ***before*** due date!

Plagiarism in any form will not be tolerated

Academic Integrity

“Black Board” Collaboration Policy

- Can discuss approach together on a “black board”
- Leave and write up solution independently
- Do not copy solutions

Plagiarism in any form will not be tolerated

Goal for Today: Putting it all Together

Compiler output is assembly files

Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution

- .

Goal for Today: Putting it all Together

Compiler output is assembly files

Assembler output is obj files

- How does the assembler resolve references/labels?
- How does the assembler resolve external references?

Linker joins object files into one executable

- How does the linker combine separately compiled files?
- How does linker resolve unresolved references?
- How does linker relocate data and code segments

Loader brings it into memory and starts execution

- How does the loader start executing a program?
- How does the loader handle shared libraries?

Big Picture

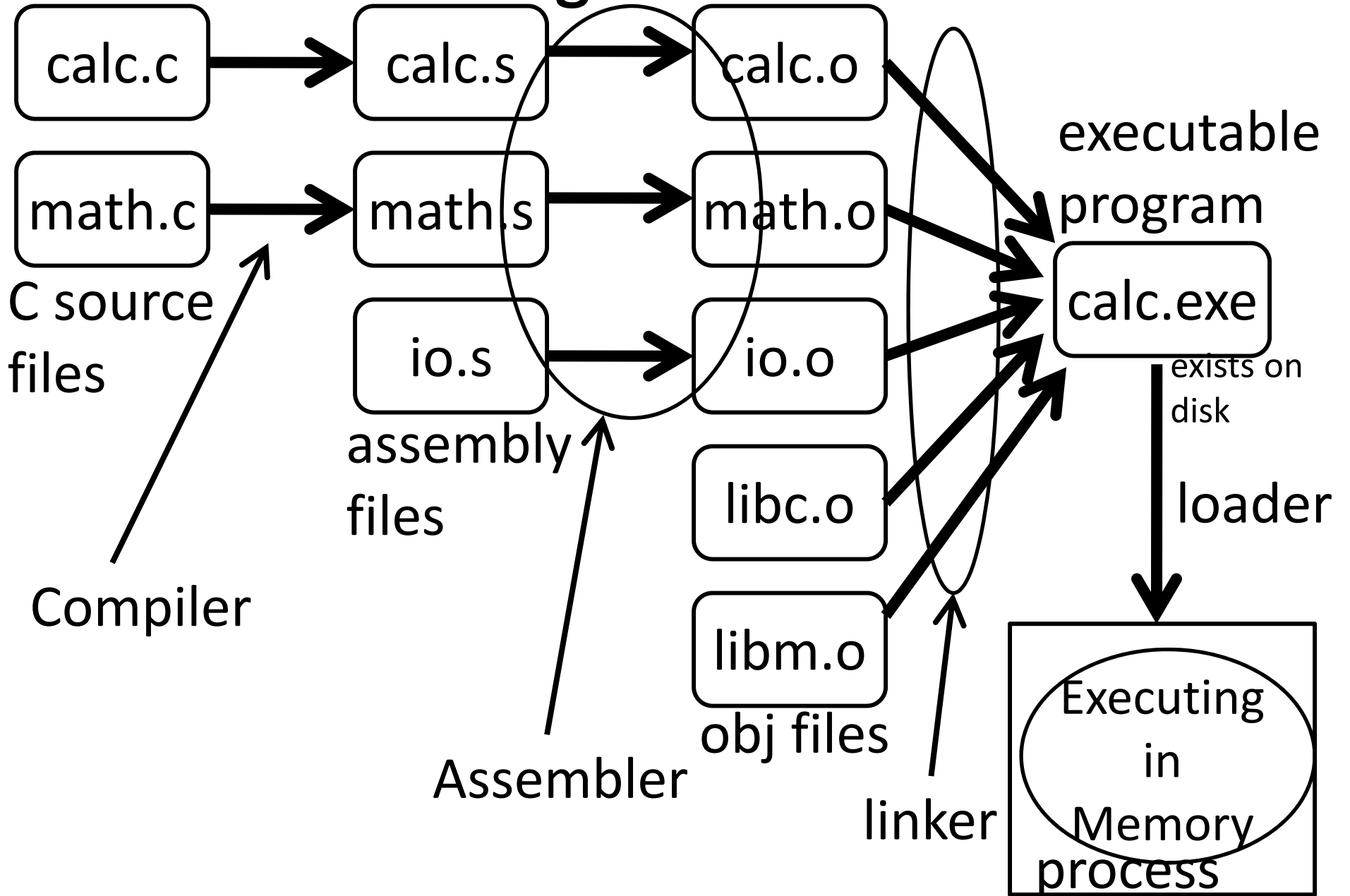
Assembler output is obj files

- Not executable
- May refer to external symbols
- Each object file has its own address space

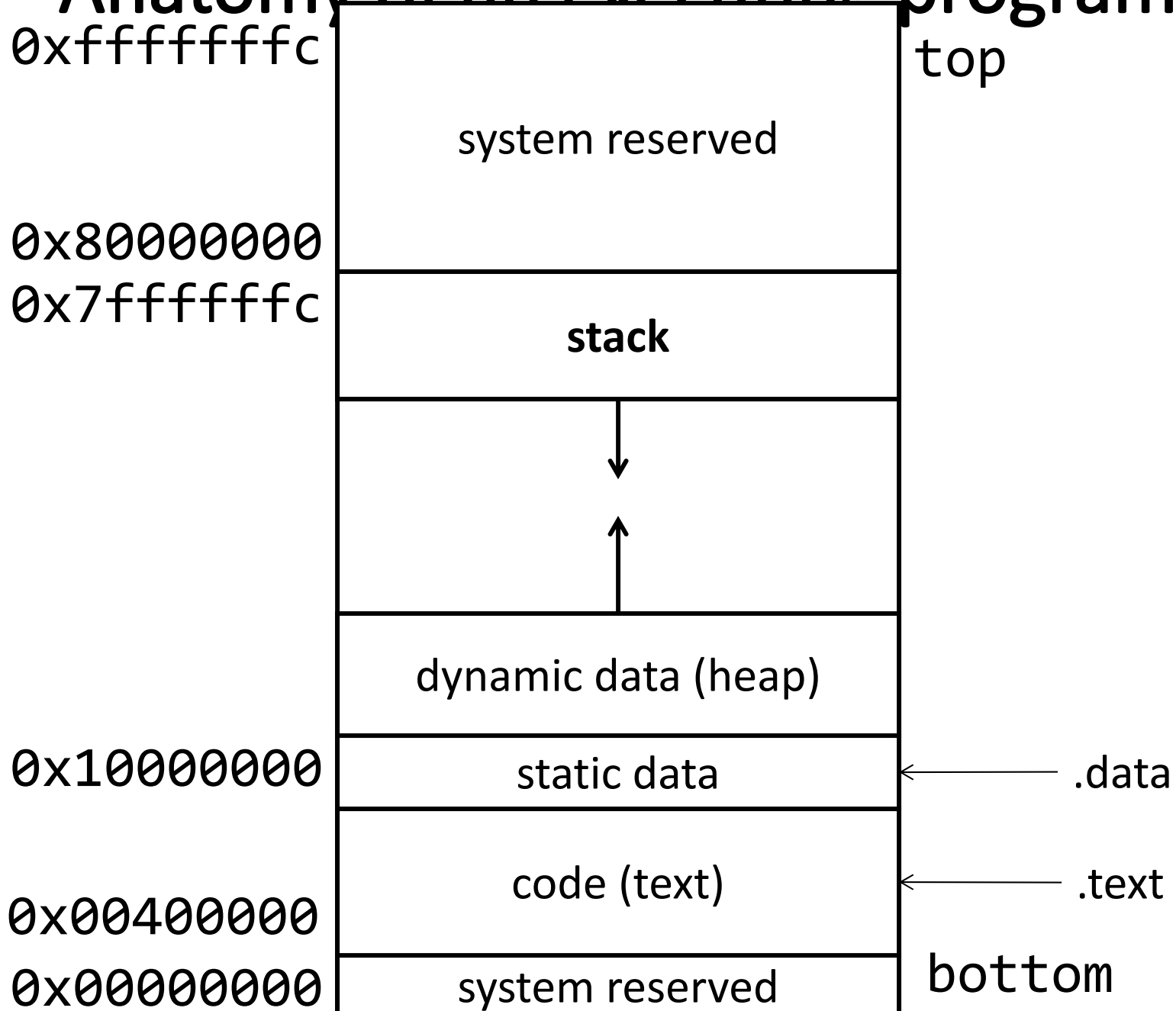
Linker joins these object files into one executable

Loader brings it into memory and executes

Big Picture



Anatomy of an executing program



Example #2: Review of Program Layout

calc.c

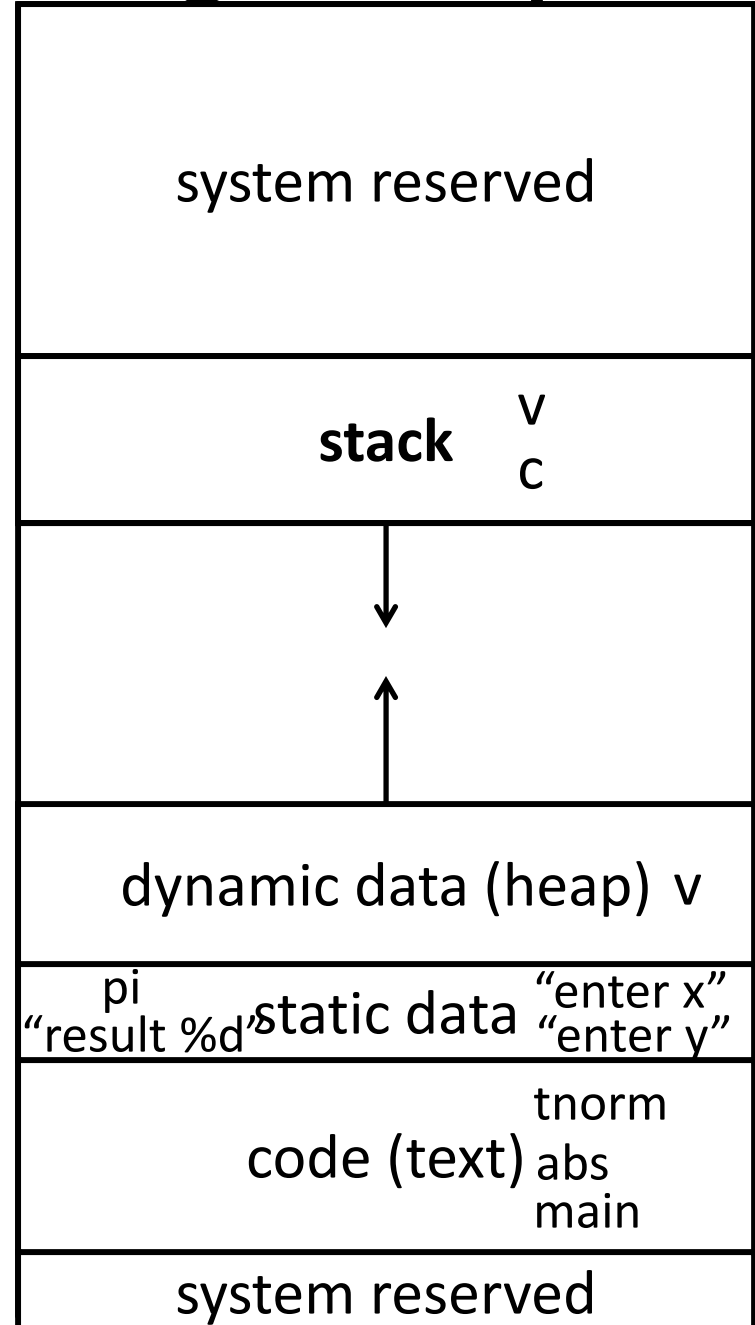
```
vector* (v) = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int (c) = pi + tnorm(v);
print("result %d", c);
```

math.c

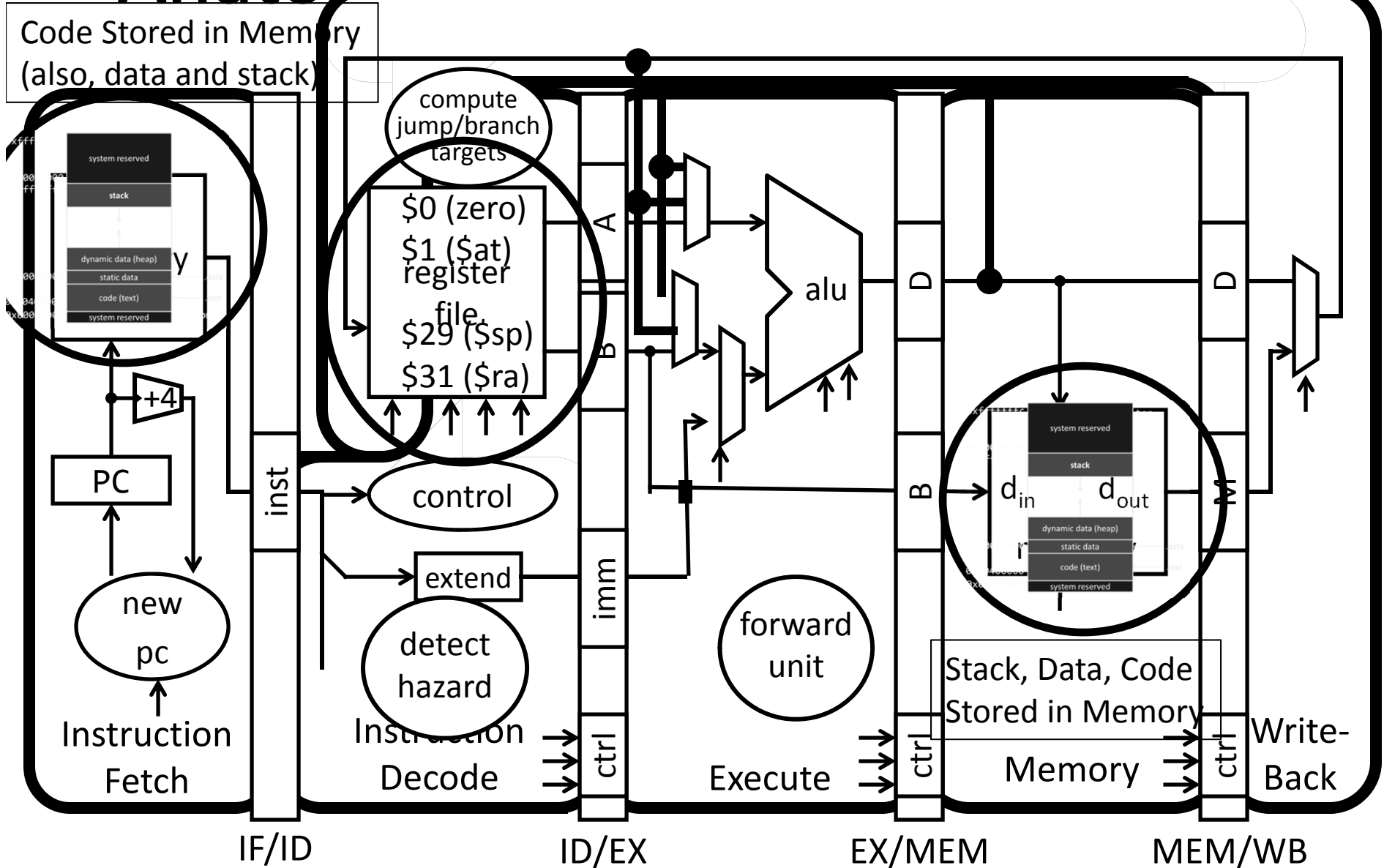
```
int tnorm(vector* v) {
    return abs(v->x)+abs(v->y);
}
```

lib3410.o

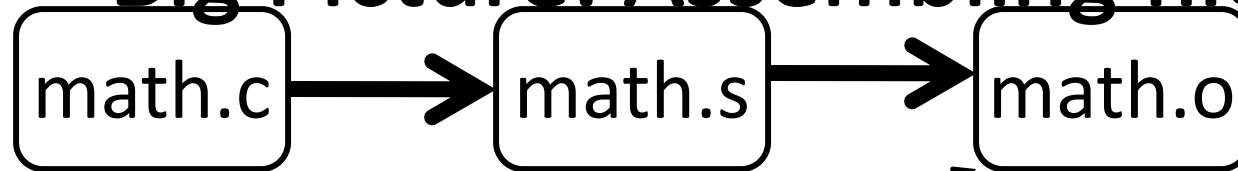
global variable: (pi)
 entry point: prompt
 entry point: print
 entry point: malloc



Anatomy of an executing program



Big Picture: Assembling file separately



.o = Linux
.obj Windows

Output of assembler is a object files

- Binary machine code, but not executable
- How does assembler handle forward references?

Next Goal

How does the assembler handle local references

How does Assembler handle forward references

Two-pass assembly

- Do a pass through the whole program, allocate instructions and lay out data, thus determining addresses
- Do a second pass, emitting instructions and data, with the correct label offsets now determined

One-pass (or backpatch) assembly

- Do a pass through the whole program, emitting instructions, emit a 0 for jumps to labels not yet determined, keep track of where these instructions are
- Backpatch, fill in 0 offsets as labels are defined

How does Assembler handle forward references

Example:

- `bne $1, $2, (L)`
`sll $0, $0, 0`

`(L):addiu $2, $3, 0x2`

The assembler will change this to

- `bne $1, $2, +1`
`sll $0, $0, 0`
`addiu $2, $3, $0x2`

Final machine code

- `0x14220001 # bne`
`0x00000000 # sll`
`0x24620002 # addiu`

How does Assembler handle forward references

Example:

- `bne $1, $2, L`
`sll $0, $0, 0`

`L: addiu $2, $3, 0x2`

The assembler will change this to

- `bne $1, $2, +1`
`sll $0, $0, 0`
`addiu $2, $3, $0x2`

Final machine code

- `0x14220001 # bne`

| | | | |
|--------|--------|--------|------------------------|
| 000101 | 00001 | 00010 | 0000000000000000000001 |
| 000000 | 000000 | 000000 | 0000000000000000000000 |
| 001001 | 00011 | 00010 | 0000000000000000000010 |
- `0x00000000 # sll`
- `0x24620002 # addiu`

How does Assembler handle forward references

Example:

- `bne $1, $2, L`
`sll $0, $0, 0`
`L: addiu $2, $3, 0x2`

The assembler will change this to

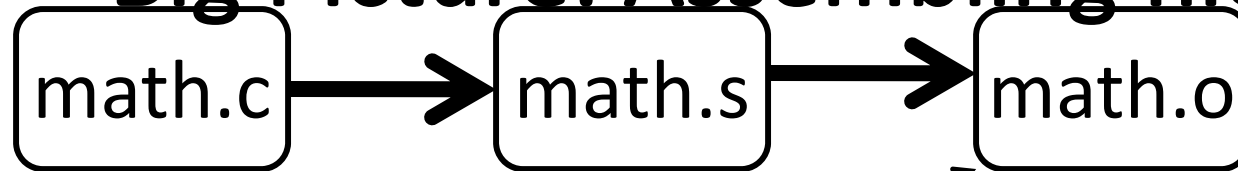
- `bne $1, $2, +1`
`sll $0, $0, 0`
`addiu $2, $3, $0x2`

Final machine code

- `0x14220001 # bne`
`0x00000000 # sll`
`0x24620002 # addiu`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 2 | 2 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 4 | 6 | 2 | 0 | 0 | 0 | 2 |

Big Picture: Assembling file separately



.o = Linux

.obj Windows

Output of assembler is a object files

- Binary machine code, but not executable
- How does assembler handle forward references?
- May refer to external symbols i.e. Need a “symbol table”
- Each object file has illusion of its own address space

– Addresses will need to be fixed later

e.g. .text (code) starts at addr 0x00000000

.data starts @ addr 0x00000000

Next Goal

How does the assembler handle external references

Symbols and References

Global labels: Externally visible “exported” symbols

- Can be referenced from other object files
- Exported functions, global variables

e.g. pi
(from a couple of slides ago)

Local labels: Internal visible only symbols

- Only used within this object file
- static functions, static variables, loop labels, ...

e.g.
static foo
static bar
static baz

e.g.
\$str
\$L0
\$L2

Object file

Header

- Size and position of pieces of file

Text Segment

- instructions

Data Segment

- static data (local/global vars, strings, constants)


Debugging Information

- line number → code address map, etc.

Symbol Table

- External (exported) references
- Unresolved (imported) references

Object File



Example

math.c

```
int pi = 3; } global
int e = 2; }
static int randomval = 7;
    ↙ local (to current file)

extern char *username;
extern int printf(char *str, ...);
    ↙ external (defined in another file)

int square(int x) { ... }

static int is_prime(int x) { ... }
    ↙ local
int pick_prime() { ... } } global
int pick_random() {
    return randomval;
}
```

gcc -S .. math.c

Compiler

gcc -c .. math.s

Assembler

objdump --disassemble math.o

objdump --syms math.o

Objdump disassembly

```
csug01 ~$ mipsel-linux-objdump --disassemble math.o  
math.o:      file format elf32-tradlittlemips  
Disassembly of section .text:
```

00000000 <pick_random>:

| | | | |
|-----|----------|-------|----------|
| 0: | 27bdfff8 | addiu | sp,sp,-8 |
| 4: | afbe0000 | sw | s8,0(sp) |
| 8: | 03a0f021 | move | s8,sp |
| c: | 3c020000 | lui | v0,0x0 |
| 10: | 8c420008 | lw | v0,8(v0) |
| 14: | 03c0e821 | move | sp,s8 |
| 18: | 8fbe0000 | lw | s8,0(sp) |
| 1c: | 27bd0008 | addiu | sp,sp,8 |
| 20: | 03e00008 | jr | ra |
| 24: | 00000000 | nop | |

00000028 <square>:

| | | | |
|-----|----------|-------|----------|
| 28: | 27bdfff8 | addiu | sp,sp,-8 |
| 2c: | afbe0000 | sw | s8,0(sp) |
| 30: | 03a0f021 | move | s8,sp |
| 34: | afc40008 | sw | a0,8(s8) |

Objdump disassembly

```
csug01 ~$ mipsel-linux-objdump --disassemble math.o
```

```
math.o:      file format elf32-tradlittlemips
```

```
Disassembly of section .text:
```

```
Address      instruction Mem[8] = instruction 0x03a0f021 (move s8,sp)
```

```
00000000 <pick random>:
```

| | | | | | |
|-----|----------|-------|----------|------------|------------------------|
| 0: | 27bdfff8 | addiu | sp,sp,-8 | } prologue | resolved (fixed) later |
| 4: | afbe0000 | sw | s8,0(sp) | | |
| 8: | 03a0f021 | move | s8,sp | | |
| c: | 3c020000 | lui | v0,0x0 | } body | |
| 10: | 8c420008 | lw | v0,8(v0) | | |
| 14: | 03c0e821 | move | sp,s8 | } epilogue | |
| 18: | 8fbe0000 | lw | s8,0(sp) | | |
| 1c: | 27bd0008 | addiu | sp,sp,8 | | |
| 20: | 03e00008 | jr | ra | | |
| 24: | 00000000 | nop | | | |

symbol

```
00000028 <square>:
```

| | | | |
|-----|----------|-------|----------|
| 28: | 27bdfff8 | addiu | sp,sp,-8 |
| 2c: | afbe0000 | sw | s8,0(sp) |
| 30: | 03a0f021 | move | s8,sp |
| 34: | afc40008 | sw | a0,8(s8) |

Objdump symbols

```
csug01 ~$ mipsel-linux-objdump --syms math.o  
math.o:      file format elf32-tradlittlemips
```

SYMBOL TABLE:

| | | | | | |
|----------|---|----|---------------|----------|---------------|
| 00000000 | l | df | *ABS* | 00000000 | math.c |
| 00000000 | l | d | .text | 00000000 | .text |
| 00000000 | l | d | .data | 00000000 | .data |
| 00000000 | l | d | .bss | 00000000 | .bss |
| 00000000 | l | d | .mdebug.abi32 | 00000000 | .mdebug.abi32 |
| 00000008 | l | 0 | .data | 00000004 | randomval |
| 00000060 | l | F | .text | 00000028 | is_prime |
| 00000000 | l | d | .rodata | 00000000 | .rodata |
| 00000000 | l | d | .comment | 00000000 | .comment |
| 00000000 | g | 0 | .data | 00000004 | pi |
| 00000004 | g | 0 | .data | 00000004 | e |
| 00000000 | g | F | .text | 00000028 | pick_random |
| 00000028 | g | F | .text | 00000038 | square |
| 00000088 | g | F | .text | 0000004c | pick_prime |
| 00000000 | | | *UND* | 00000000 | username |
| 00000000 | | | *UND* | 00000000 | printf |

Objdump symbols

csug01 ~\$ mipsel-linux-objdump --syms math.o

math.o: file format elf32-tradlittlemips

| Address | l: local | segment | segment | size | |
|---------------|----------|-----------------|-----------|----------|--------------------|
| SYMBOL TABLE: | | | | | |
| 00000000 | 1 | g: global | | 00000000 | math.c |
| 00000000 | 1 | df *ABS* | | 00000000 | |
| 00000000 | 1 | d .text | | 00000000 | .text |
| 00000000 | 1 | d .data | | 00000000 | .data |
| 00000000 | 1 | d .bss | | 00000000 | .bss |
| 00000000 | 1 | d .mdebug.abi32 | | 00000000 | .mdebug.abi32 |
| 00000008 | 1 | 0 .data | | 00000004 | randomval |
| 00000060 | 1 | F .text | | 00000028 | is_prime |
| 00000000 | 1 | d .rodata | | 00000000 | .rodata func @ |
| 00000000 | 1 | d .comment | | 00000000 | .comment addr=0x60 |
| 00000000 | g | 0 .data | | 00000004 | pi size=0x28 byte |
| 00000004 | g | 0 .data | | 00000004 | e |
| 00000000 | g | F .text | | 00000028 | pick_random |
| 00000028 | g | F .text | | 00000038 | square |
| 00000088 | g | F .text | | 0000004c | pick_prime |
| 00000000 | f: func | *UND* | external | 00000000 | username |
| 00000000 | O: obj | *UND* | reference | 00000000 | printf |

Static local

func @

addr=0x60

size=0x28 byte

UND external
UND reference

Separate Compilation

Q: Why separate compile/assemble and linking steps?

- a) Removes the need to recompile the whole program
- b) Need to just recompile a small module
- c) Separation of concern: Linker coalesces object files
- d) All the above
- e) None of the above

Separate Compilation

Q: Why separate compile/assemble and linking steps?

A: Separately compiling modules and linking them together obviates the need to recompile the whole program every time something changes

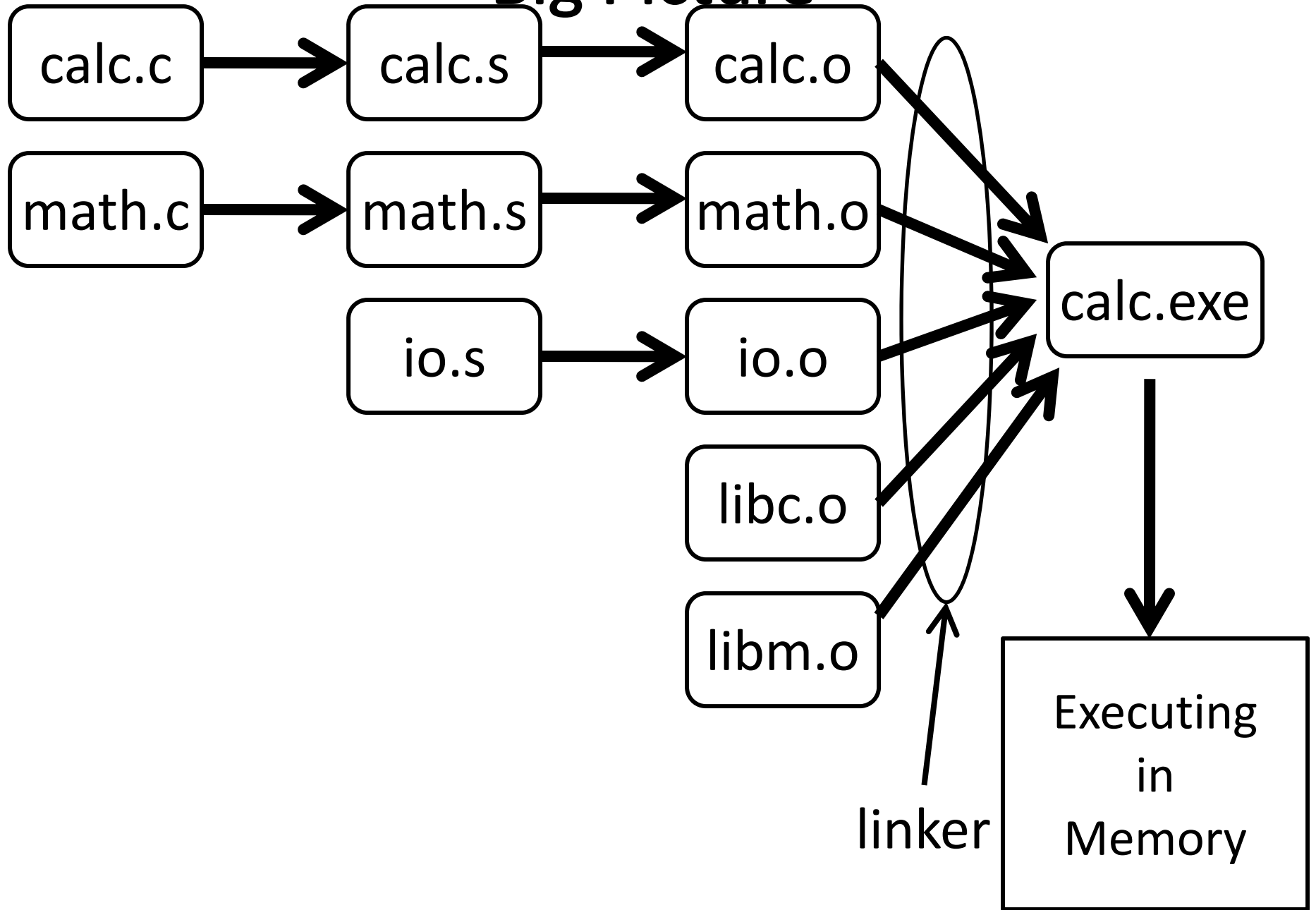
- Need to just recompile a small module
- A linker coalesces object files together to create a complete program

Linkers

Next Goal

How do we link together separately compiled and assembled machine object files?

Big Picture



Linkers

Linker combines object files into an executable file

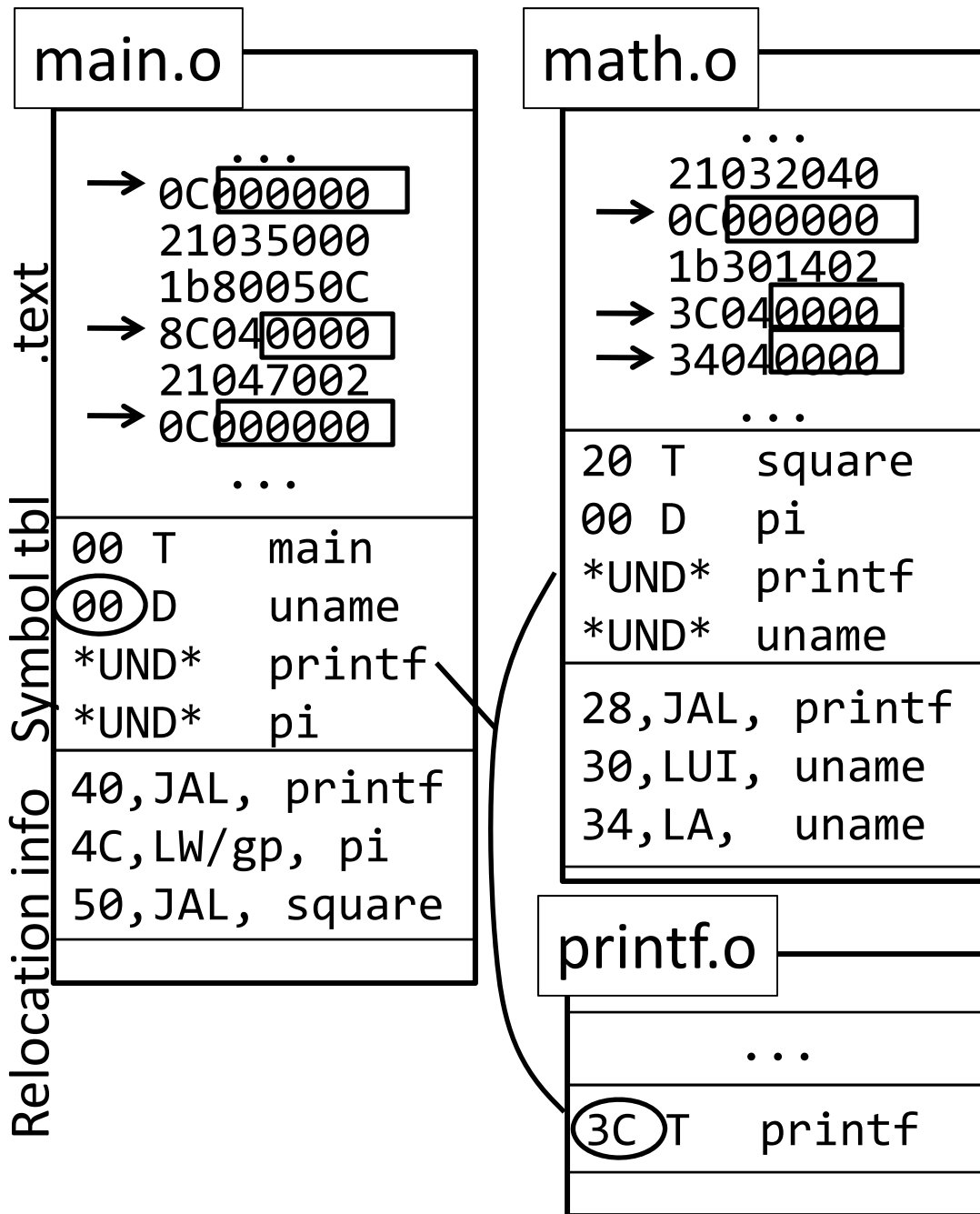
- Relocate each object's text and data segments
- Resolve as-yet-unresolved symbols
- Record top-level entry point in executable file

End result: a program on disk, ready to execute

- E.g. ./calc Linux
 ./calc.exe Windows
 simulate calc Class MIPS simulator

.

Linker Example



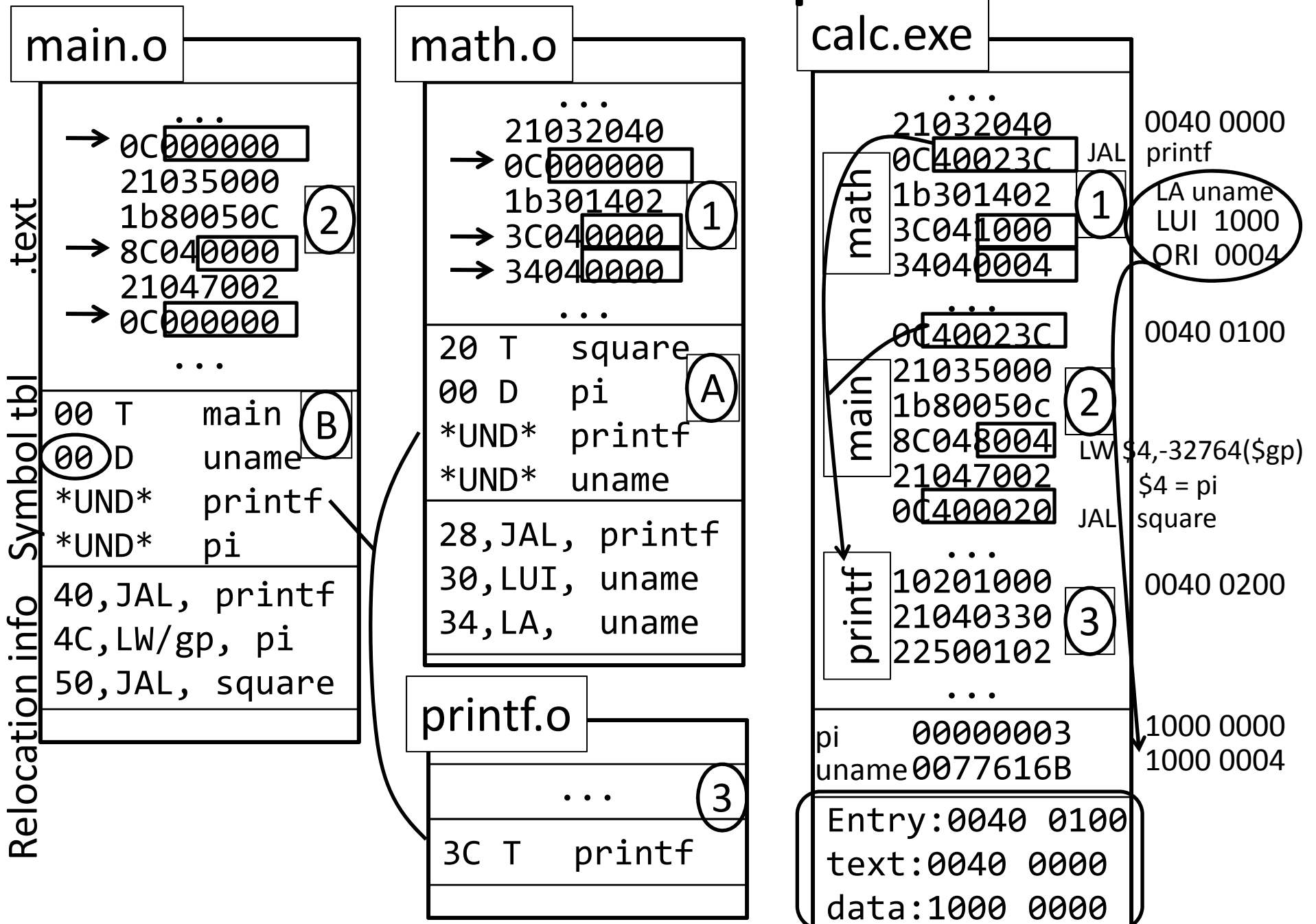
External references need to be resolved (fixed)

Steps

- 1) Find UND symbols in symbol table
- 2) Relocate segments that collide

e.g. uname @0x00
 pi @ 0x00
 square @ 0x00
 main @ 0x00

Linker Example



Object file

Header

- location of main entry point (if any)

Text Segment

- instructions

Data Segment

- static data (local/global vars, strings, constants)

Relocation Information


- Instructions and data that depend on actual addresses
- Linker patches these bits after relocating segments

Symbol Table

- Exported and imported references

Debugging Information

Object File



Object File Formats

Unix

- a.out
- COFF: Common Object File Format
- ELF: Executable and Linking Format
- ...

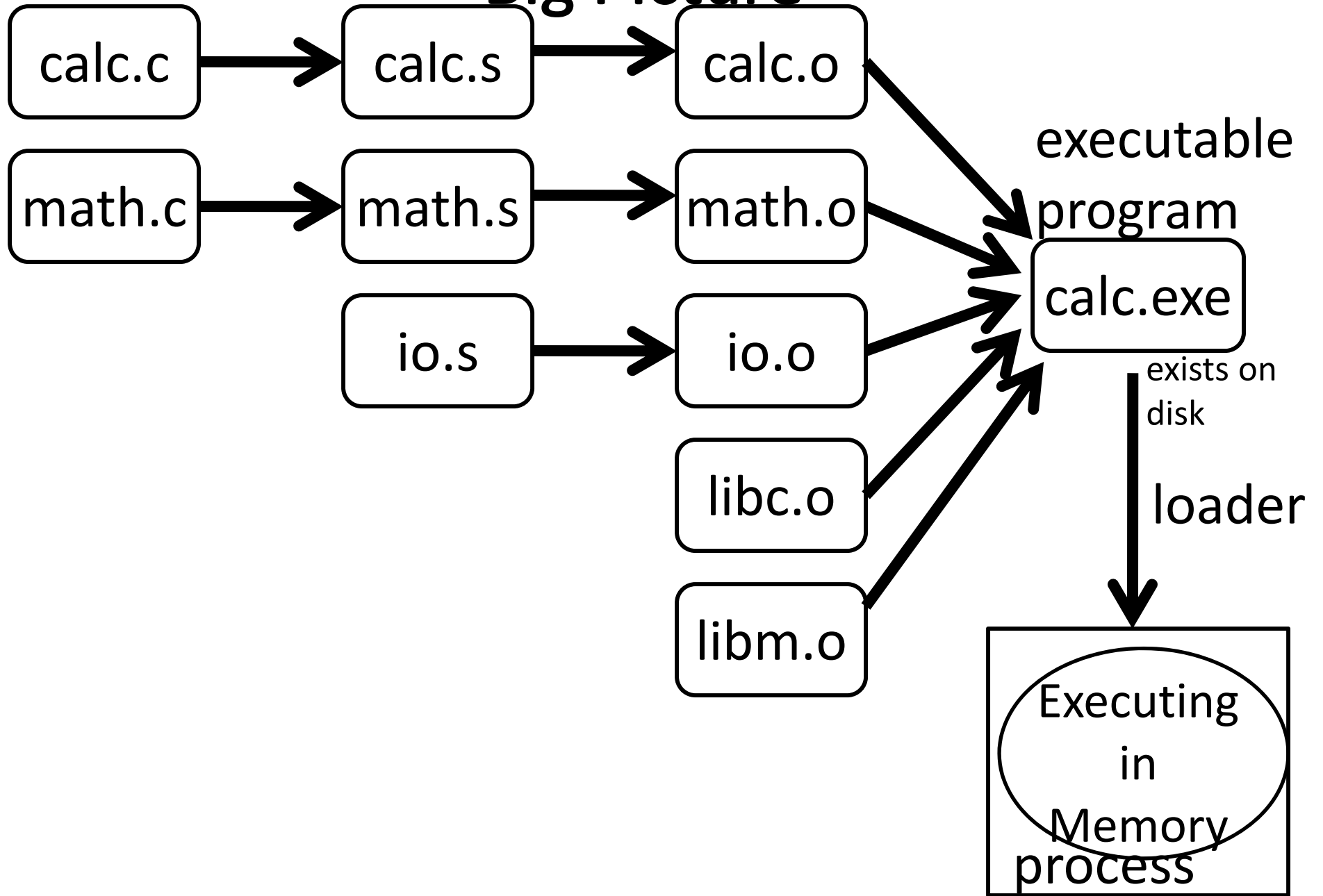
Windows

- PE: Portable Executable

All support both executable and object files

Loaders and Libraries

Big Picture



Loaders

Loader reads executable from disk into memory

- Initializes registers, stack, arguments to first function
- Jumps to entry-point

Part of the Operating System (OS)

Static Libraries

Static Library: Collection of object files
(think: like a zip archive)

Q: But every program contains entire library!

A: Linker picks only object files needed to resolve undefined references at link time

e.g. libc.a contains many objects:

- printf.o, fprintf.o, vprintf.o, sprintf.o, snprintf.o, ...
- read.o, write.o, open.o, close.o, mkdir.o, readdir.o, ...
- rand.o, exit.o, sleep.o, time.o,

Shared Libraries

Q: But every program still contains part of library!

A: shared libraries

- executable files all point to single *shared library* on disk
- final linking (and relocations) done by the loader

Optimizations:

- Library compiled at fixed non-zero address
- Jump table in each program instead of relocations
- Can even patch jumps on-the-fly

Direct Function Calls

Direct call:

```
00400010 <main>:  
    ...  
    jal 0x00400330  
    ...  
    jal 0x00400620  
    ...  
    jal 0x00400330  
    ...  
00400330 <printf>:  
    ...  
00400620 <gets>:  
    ...
```

Drawbacks:

Linker or loader must edit every use of a symbol (call site, global var use, ...)

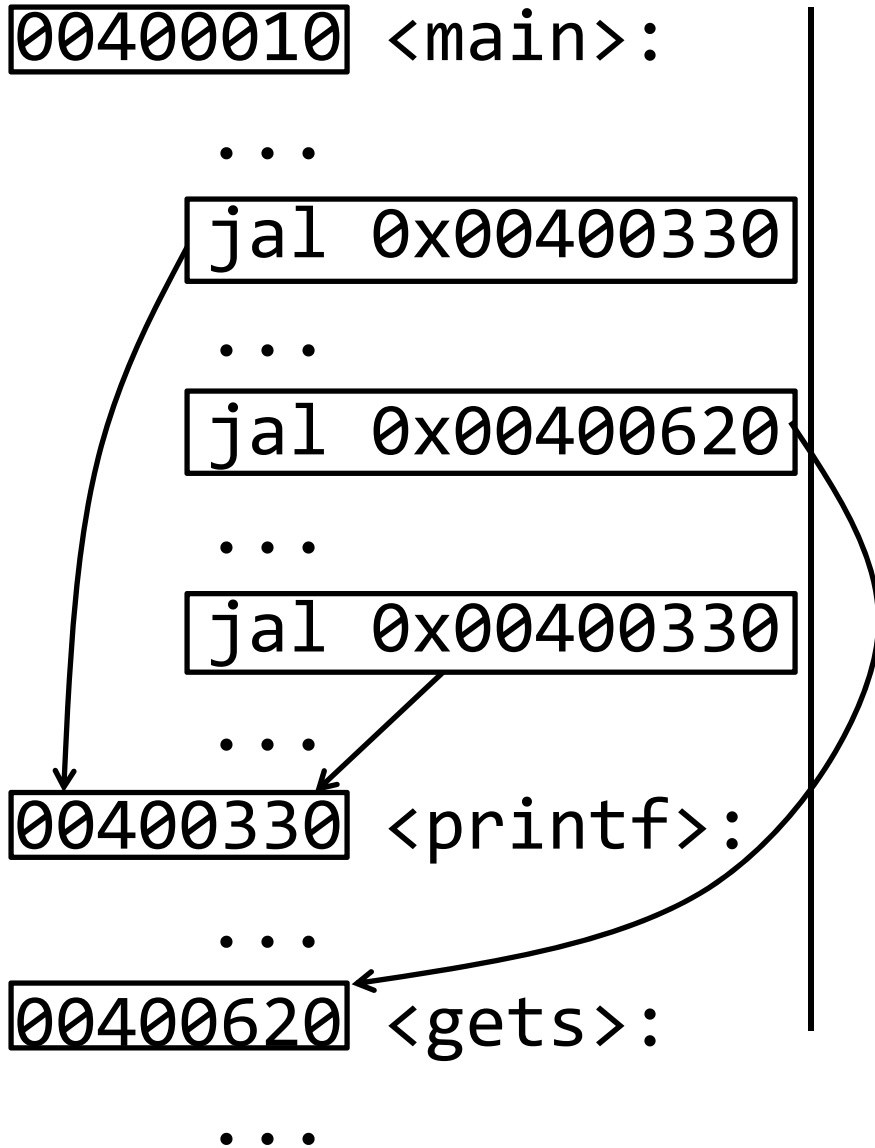
Idea:

Put all symbols in a single “global offset table”

Code does lookup as needed

Indirect Function Calls

Indirect call:



GOT: global offset table

| |
|---------------------|
| 0x00400010 # main |
| 0x00400330 # printf |
| 0x00400620 # gets |
| |
| |

Indirect Function Calls

Indirect call:

`00400010` <main>:

...
`lw $t9, -32708($gp)`
`jalr $t9`

...
`lw $t9, -32704($gp)`
`jalr $t9`

...
`lw $t9, -32708($gp)`
`jalr $t9`

`00400330` <printf>:

...
`00400620` <gets>:

...

data segment

GOT: global offset table

| | |
|---|---------------------|
| 0 | 0x00400010 # main |
| 4 | 0x00400330 # printf |
| 8 | 0x00400620 # gets |
| | |
| | |

global offset table

to be loaded

at `-32712($gp)`

printf = $4 + (-32712) + \$gp$

gets = $8 + (-32712) + \$gp$

Indirect Function Calls

Indirect call:

00400010 <main>:

...
lw \$t9, -32708(\$gp)
jalr \$t9

...
lw \$t9, -32704(\$gp)
jalr \$t9

...
lw \$t9, -32708(\$gp)
jalr \$t9

00400330 <printf>:

...
00400620 <gets>:
...

data segment

.got

.word 0x00400010 # main

.word 0x00400330 # printf

.word 0x00400620 # gets

global offset table

to be loaded

at **-32712(\$gp)**

printf = 4+(-32712)+\$gp

gets = 8+(-32712)+\$gp

Dynamic Linking

Indirect call with on-demand dynamic linking:

```
00400010 <main>:
    ...
    # load address of prints
    # from .got[1]
    lw t9, -32708(gp)

    # now call it
    jalr t9
    ...
.got
    .word 00400888 # open
    .word 00400888 # prints
    .word 00400888 # gets
    .word 00400888 # foo
```

Dynamic Linking

Indirect call with on-demand dynamic linking:

00400010 <main>:

```
...  
# load address of prints  
# from .got[1]  
lw t9, -32708(gp)  
# also load the index 1  
li t8, 1  
# now call it  
jalr t9  
...
```

.got

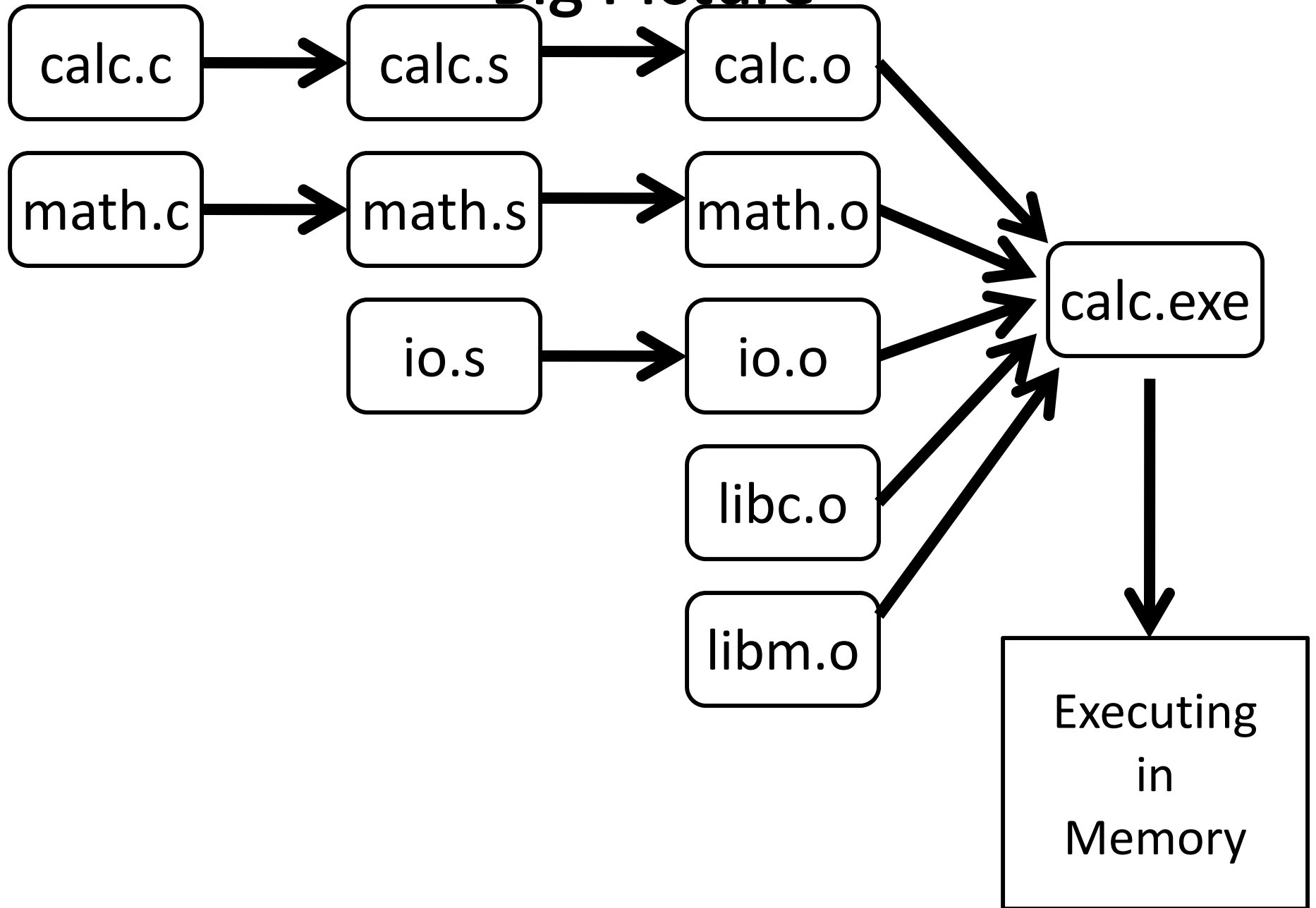
```
.word 00400888 # open  
# .word 00400888 # prints  
.word 00400888 # gets  
.word 00400888 # foo
```

```
...  
00400888 <dlresolve>:  
# t9 = 0x400888  
# t8 = index of func that  
# needs to be loaded  
  
# load that func  
... # t7 = loadfromdisk(t8)
```

```
# save func's address so  
# so next call goes direct  
... # got[t8] = t7
```

```
# also jump to func  
jr t7  
# it will return directly  
# to main, not here
```

Big Picture



Dynamic Shared Objects

Windows: dynamically loaded library (DLL)

- PE format

Unix: dynamic shared object (DSO)

- ELF format

Unix also supports Position Independent Code (PIC)

- Program determines its current address whenever needed (no absolute jumps!)
- Local data: access via offset from current PC, etc.
- External data: indirection through Global Offset Table (GOT)
- ... which in turn is accessed via offset from current PC

Static and Dynamic Linking

Static linking

- Big executable files (all/most of needed libraries inside)
- Don't benefit from updates to library
- No load-time linking

Dynamic linking

- Small executable files (just point to shared library)
- Library update benefits all programs that use it
- Load-time cost to do final linking
 - But dll code is probably already in memory
 - And can do the linking incrementally, on-demand

Recap

Compiler output is assembly files

Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution