Assemblers, Linkers, and Loaders

Hakim Weatherspoon
CS 3410, Spring 2012
Computer Science
Cornell University

See: P&H Appendix B.3-4

Administrivia

Upcoming agenda

- HW3 due ***today*** Tuesday, March 13th
- HW4 available by tomorrow, Wednesday March 14th
- PA2 Work-in-Progress circuit due before spring break
- Spring break: Saturday, March 17th to Sunday, March 25th
- HW4 due after spring break, before Prelim2

- Prelim2 Thursday, March 29th, right after spring break

- PA2 due Monday, April 2nd, after Prelim2

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

Example: Add 1 to 100

```
int n = 100;

int main (int argc, char* argv[ ]) {
        int i;
        int m = n;
        int count = 0;

        for (i = 1; i <= m; i++)
                count += i;

        printf ("Sum 1 to %d is %d\n", n, count);
}
# Assemble
[csug01] mipsel-linux-gcc –S add1To100.c
```

Example: Add 1 to 100

```
        .data
        .globl  n
        .align  2
n:      .word   100
        .rdata
        .align  2
$str0:  .asciiz
           "Sum 1 to %d is %d\n"
        .text
        .align  2
        .globl  main
main:   addiu   $sp,$sp,-48
        sw      $31,44($sp)
        sw      $fp,40($sp)
        move    $fp,$sp
        sw      $4,48($fp)
        sw      $5,52($fp)
        la      $2,n
        lw      $2,0($2)
        sw      $2,28($fp)
        sw      $0,32($fp)
        li      $2,1
        sw      $2,24($fp)
```

```
$L2:    lw      $2,24($fp)
        lw      $3,28($fp)
        slt     $2,$3,$2
        bne     $2,$0,$L3
        lw      $3,32($fp)
        lw      $2,24($fp)
        addu    $2,$3,$2
        sw      $2,32($fp)
        lw      $2,24($fp)
        addiu   $2,$2,1
        sw      $2,24($fp)
        b       $L2
$L3:    la      $4,$str0
        lw      $5,28($fp)
        lw      $6,32($fp)
        jal     printf
        move    $sp,$fp
        lw      $31,44($sp)
        lw      $fp,40($sp)
        addiu   $sp,$sp,48
        j       $31
```

Example: Add 1 to 100

```
# Compile
[csug01] mipsel-linux-gcc –c add1To100.o

# Link
[csug01] mipsel-linux-gcc –o add1To100 add1To100.o
  ${LINKFLAGS}
# -nostartfiles –nodefaultlibs
# -static -mno-xgot -mno-embedded-pic
  -mno-abicalls -G 0 -DMIPS -Wall

# Load
[csug01] simulate add1To100
Sum 1 to 100 is 5050
MIPS program exits with status 0 (approx. 2007
  instructions in 143000 nsec at 14.14034 MHz)
```

Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local			
Global			
Dynamic			

```
int n = 100;
int main (int argc, char* argv[ ]) {
        int i, m = n, count = 0, *A = malloc(4 * m);
        for (i = 1; i <= m; i++) { count += i; A[i] = count; }
        printf ("Sum 1 to %d is %d\n", n, count);
}
```

Globals and Locals

Variables Visibility Lifetime Location

Function-Local

Global

Dynamic

C Pointers can be trouble

```
int *trouble()
{ int a; …; return &a; }
char *evil()
{ char s[20]; gets(s); return s;}
int *bad()
{ s = malloc(20); … free(s); … return s; }
```

(Can't do this in Java, C#, …)

Compilers and Assemblers

Big Picture

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

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”, c);
```

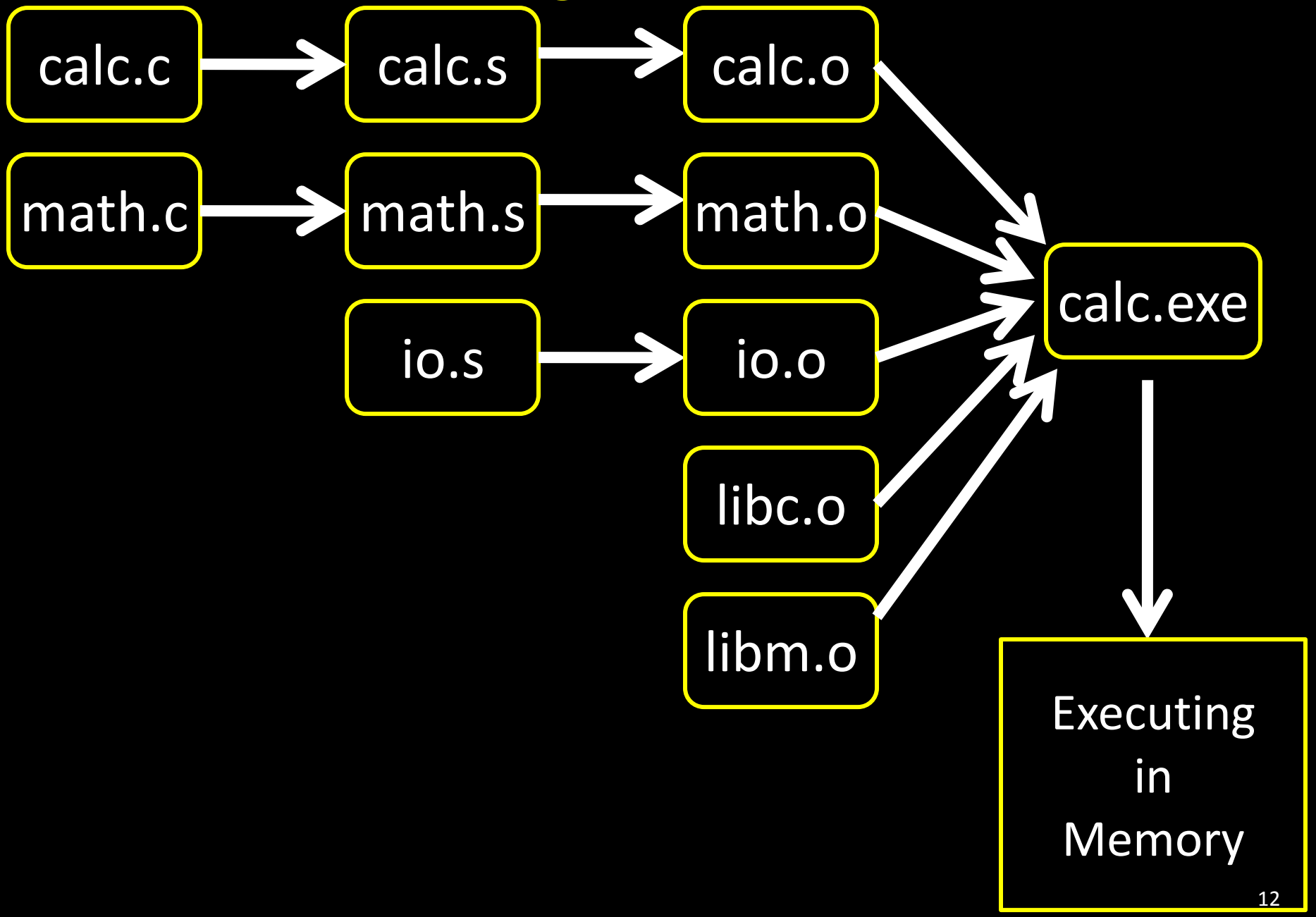
math.c

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

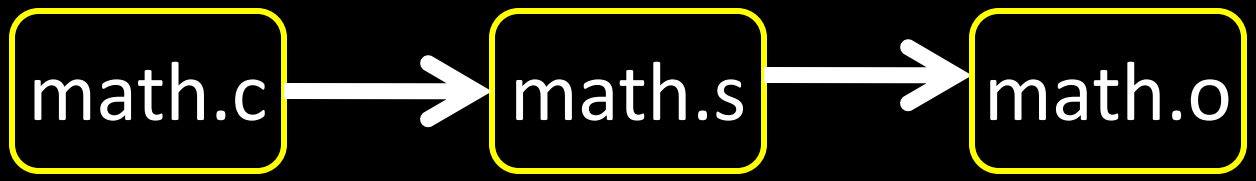
lib3410.o

```
  global variable: pi
  entry point: prompt
  entry point: print
  entry point: malloc
```

Big Picture


calc.c
calc.s
calc.o
math.c
math.s
math.o
io.s
io.o
libc.o
libm.o
calc.exe
Executing in Memory

Big Picture



Output is obj files

- Binary machine code, but not executable
- May refer to external symbols
- Each object file has illusion of its own address space
  - Addresses will need to be fixed later

Symbols and References

Global labels: Externally visible “exported” symbols

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

Local labels: Internal visible only symbols

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

Object file

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

Example

math.c

```
int pi = 3;
int e = 2;
static int randomval = 7;

extern char *username;
extern int printf(char *str, …);

int square(int x) { … }
static int is_prime(int x) { … }
int pick_prime() { … }
int pick_random() {
    return randomval;
}
```

gcc -S … math.c
gcc -c … math.s
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 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     O .data          00000004 randomval
00000060 l     F .text          00000028 is_prime
00000000 l    d  .rodata        00000000 .rodata
00000000 l    d  .comment       00000000 .comment
00000000 g     O .data          00000004 pi
00000004 g     O .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
```

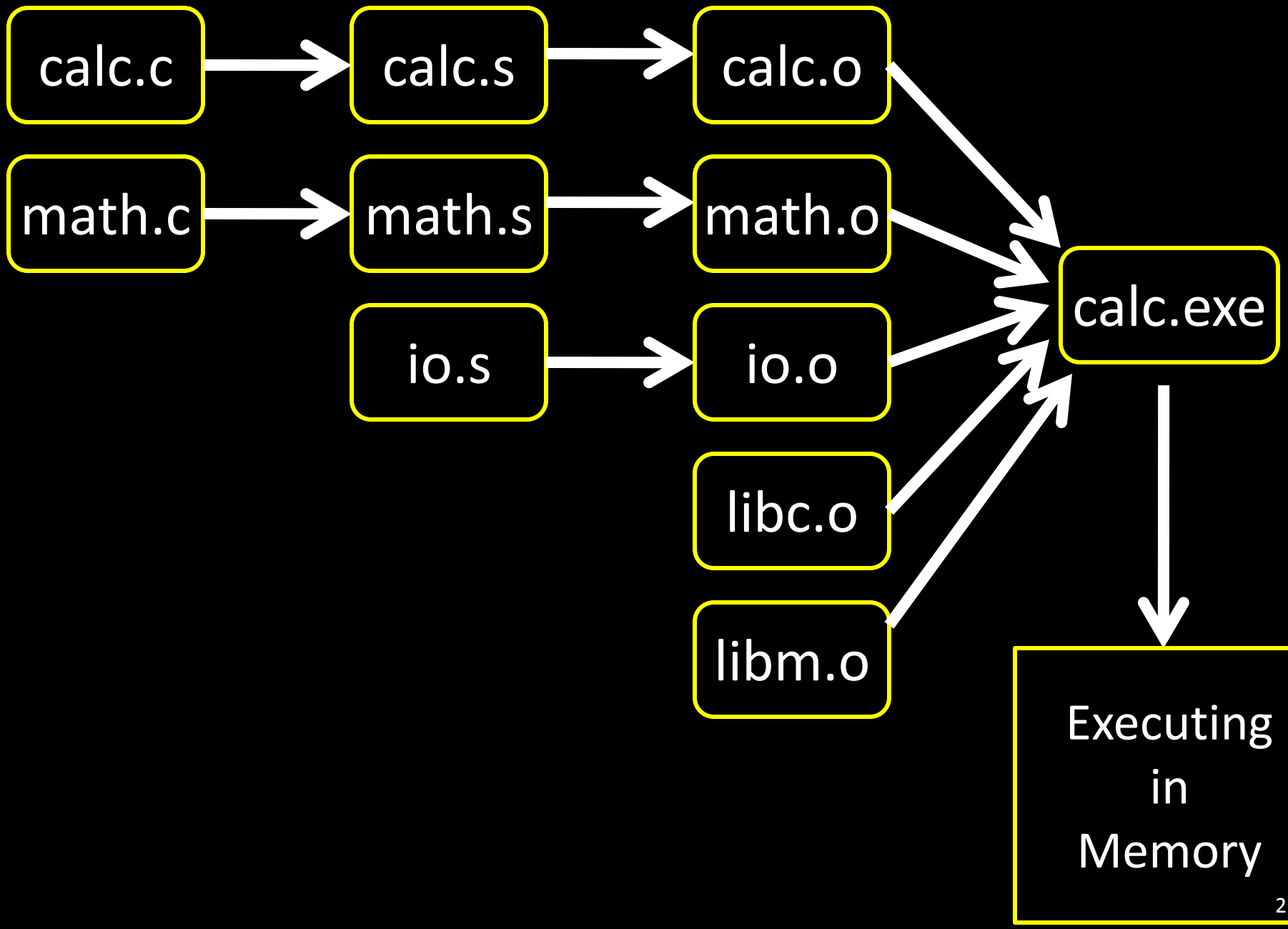
Separate Compilation

Q: Why separate compile/assemble and linking steps?

A: Can recompile one object, then just relink.

Linkers

Big Picture


calc.c
calc.s
calc.o
math.c
math.s
math.o
io.s
io.o
calc.exe
libc.o
libm.o
Executing in Memory

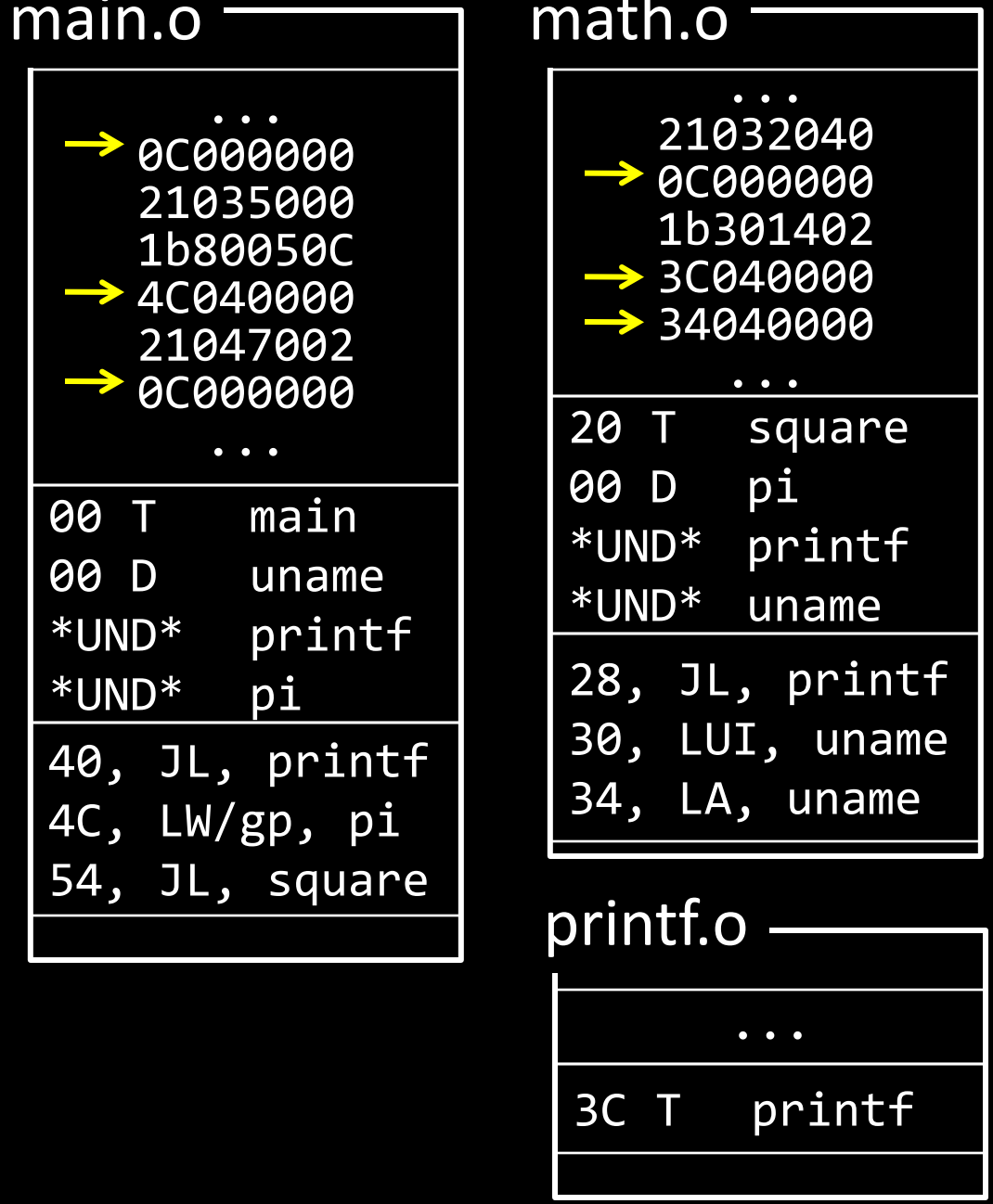
Linkers

Linker combines object files into an executable file

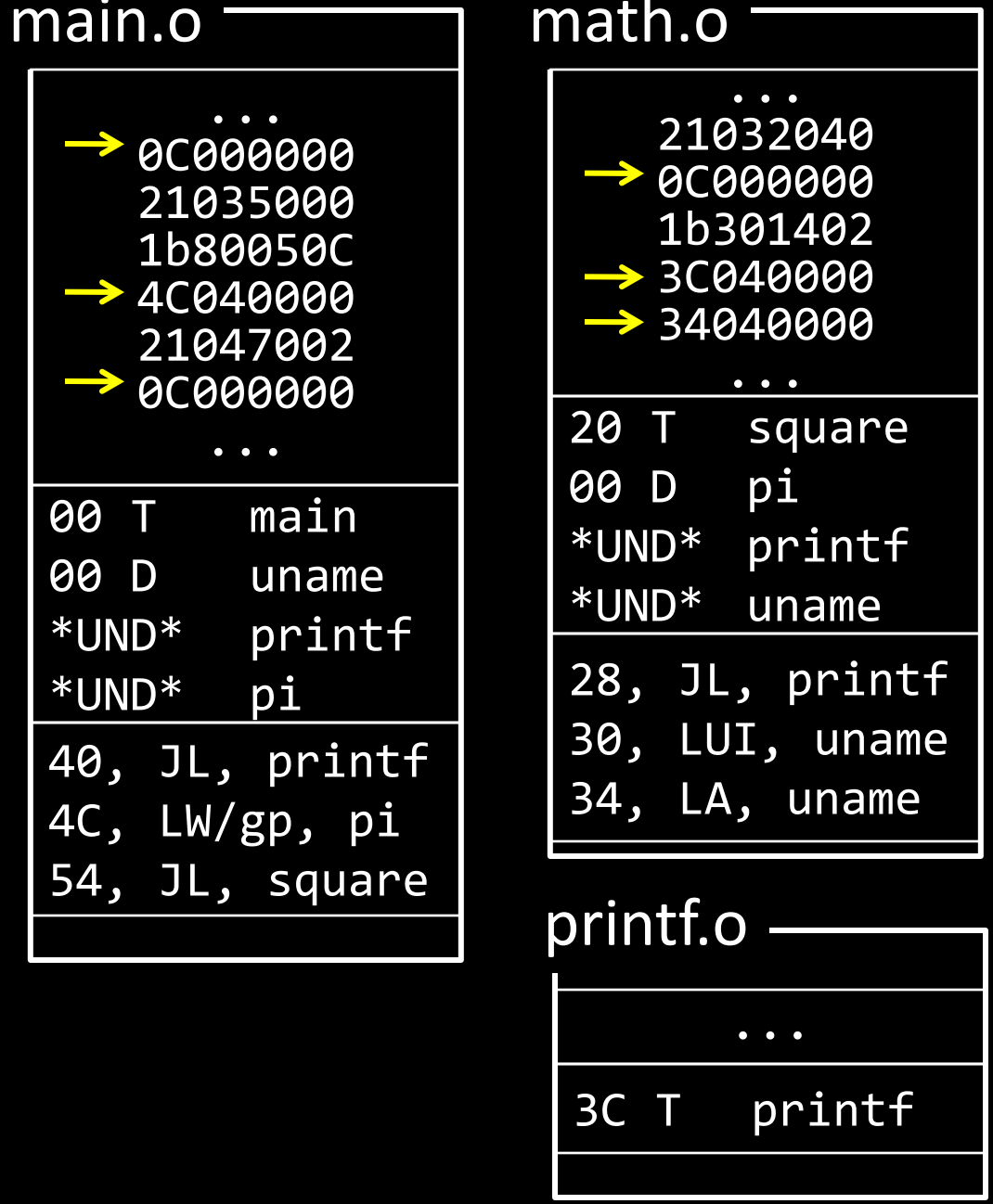
- Relocate each object's text and data segments
- Resolve as-yet-unresolved symbols
- Record top-level entry point in executable file

End result: a program on disk, ready to execute

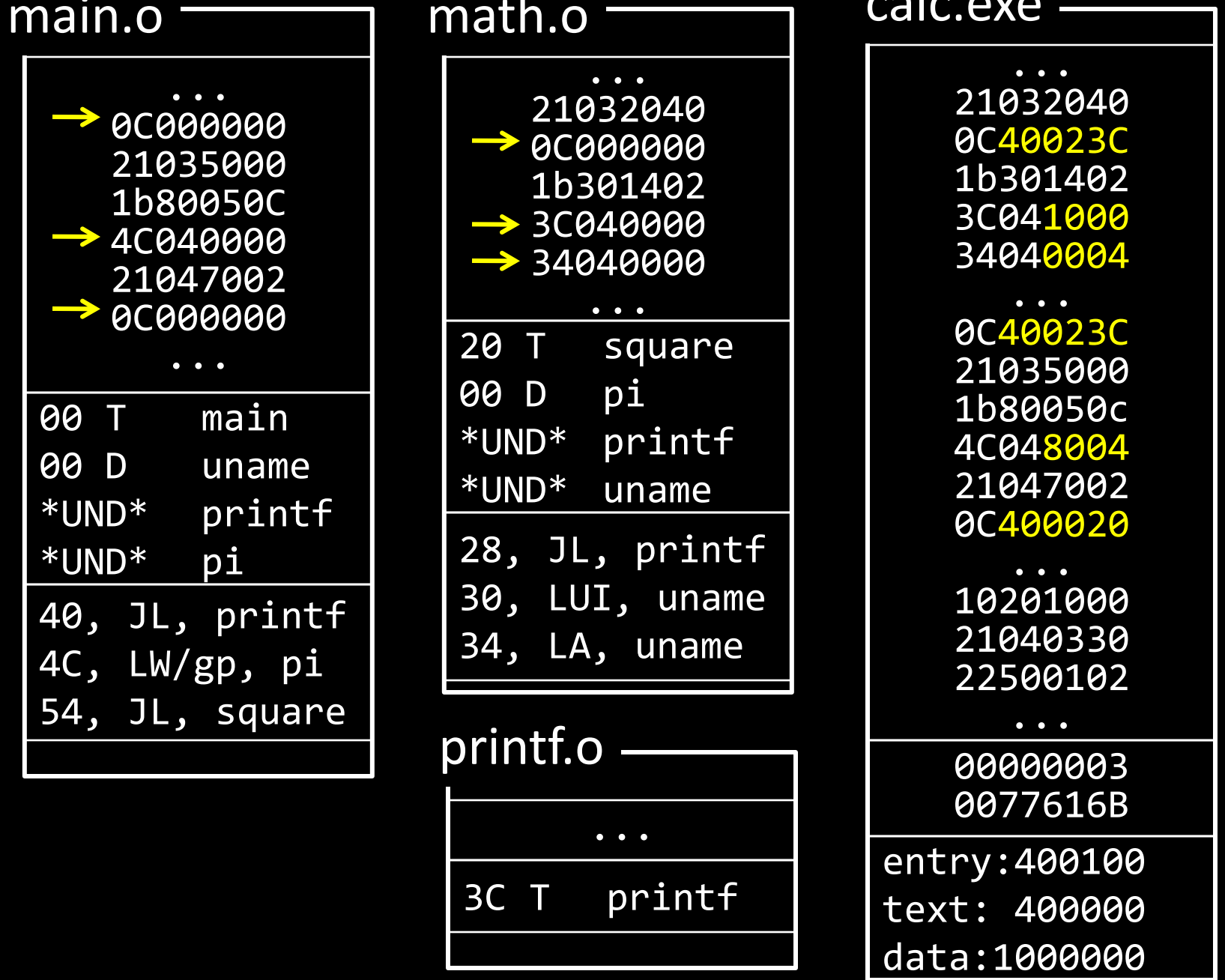
Linker Example



Linker Example



Linker Example



Object file

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

File Formats

Unix

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

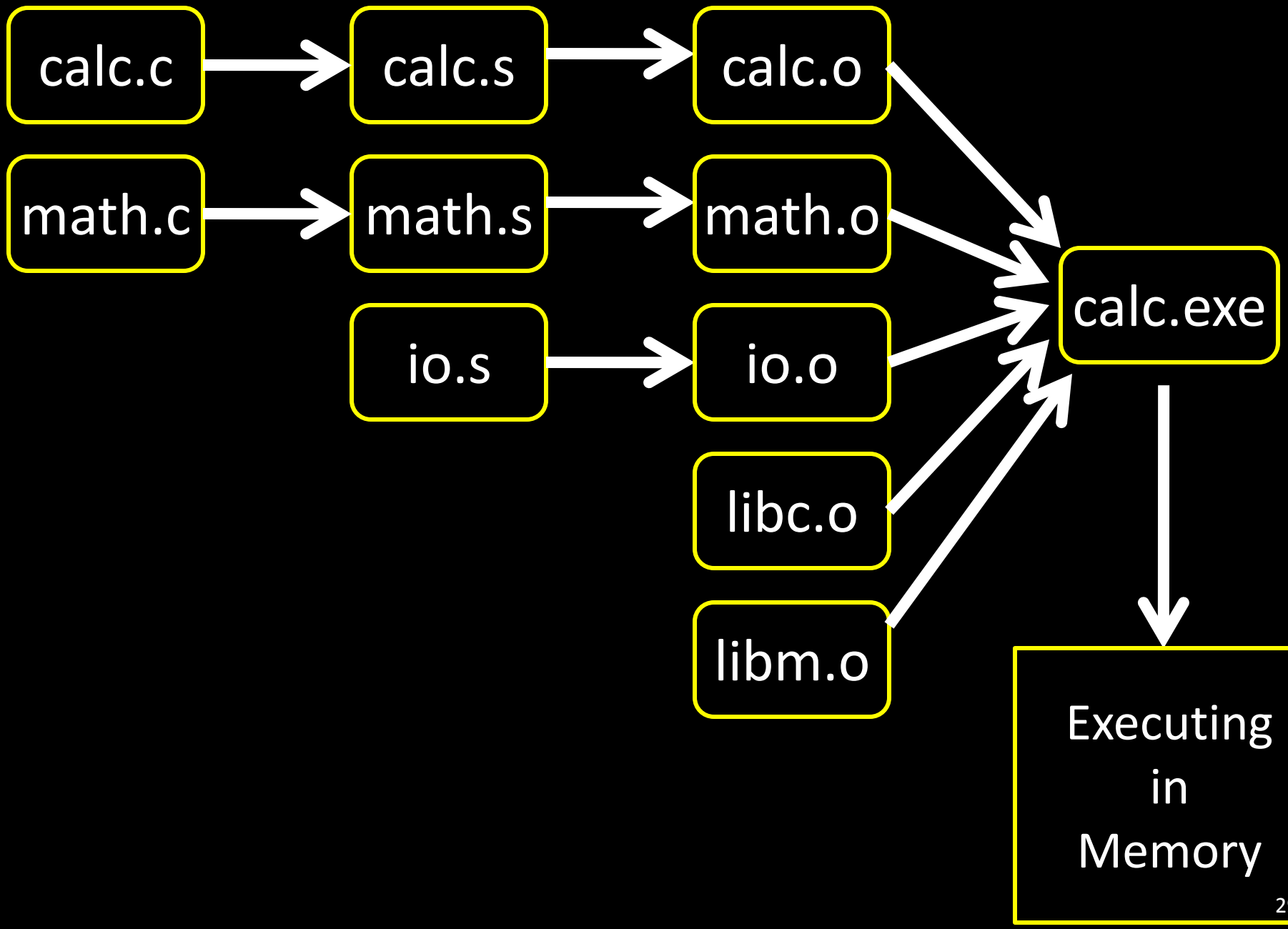
Windows

- PE: Portable Executable

All support both executable and object files

Loaders and Libraries

Big Picture


calc.c
calc.s
calc.o
math.c
math.s
math.o
io.s
io.o
libc.o
libm.o
calc.exe
Executing in Memory

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

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

GOT: global offset table

Indirect Function Calls

Indirect call:

```
00400010 <main>:
  ...
  lw t9, ? # printf
  jalr t9
  ...
  lw t9, ? # gets
  jalr t9
  ...
00400330 <printf>:
  ...
00400620 <gets>:
  ...
```

```
# data segment
  ...
  ...
# global offset table
# to be loaded
# at -32712(gp)
.got
.word 00400010 # main
.word 00400330 # printf
.word 00400620 # gets
  ...
```

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
```

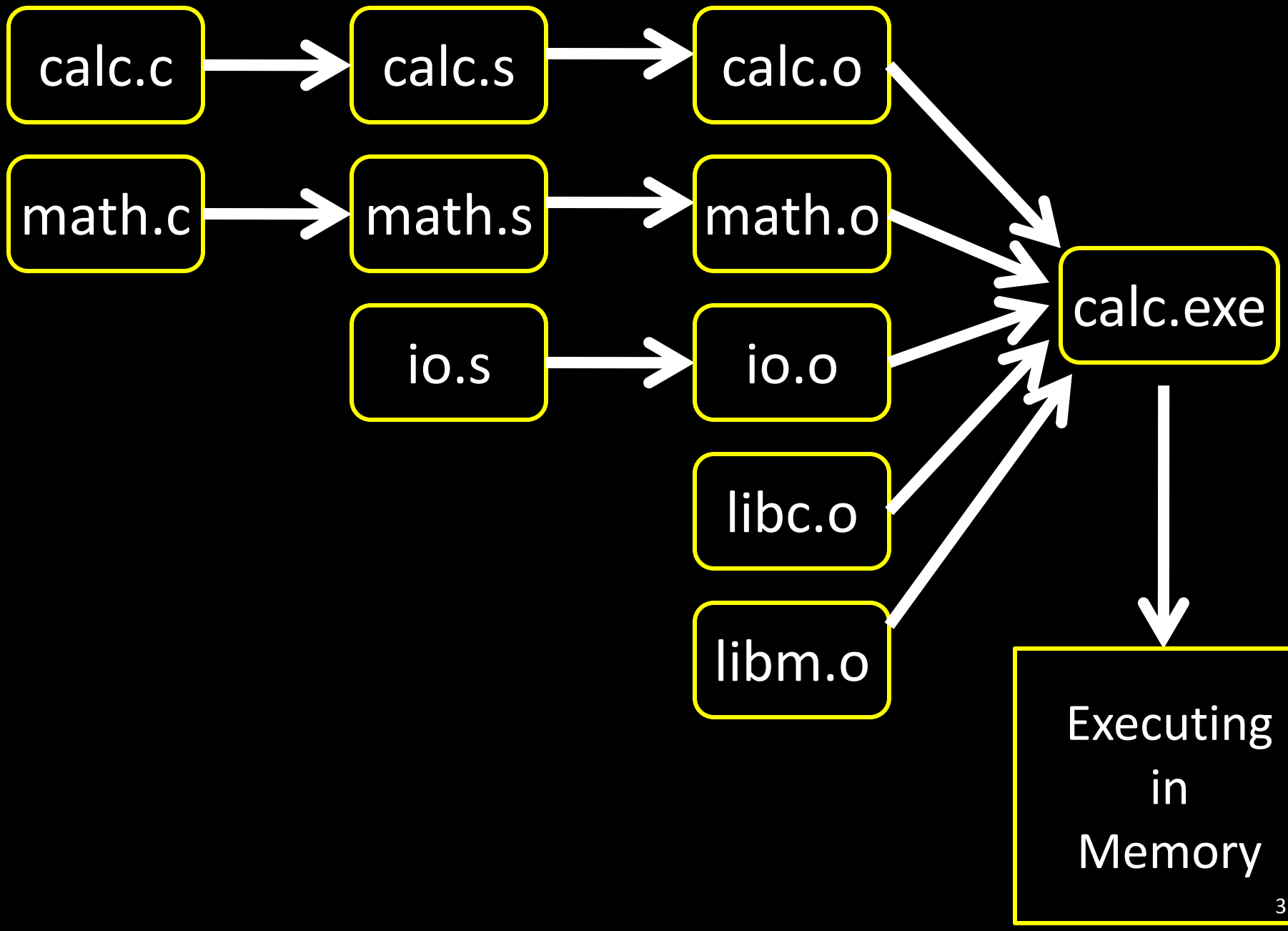
```
   ...
00400888 <dlresolve>:
   # t9 = 0x400888
   # t8 = index of func that
   #    needs to be loaded

   # 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


calc.c
calc.s
calc.o
math.c
math.s
math.o
io.s
io.o
calc.exe
libc.o
libm.o
Executing in Memory

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