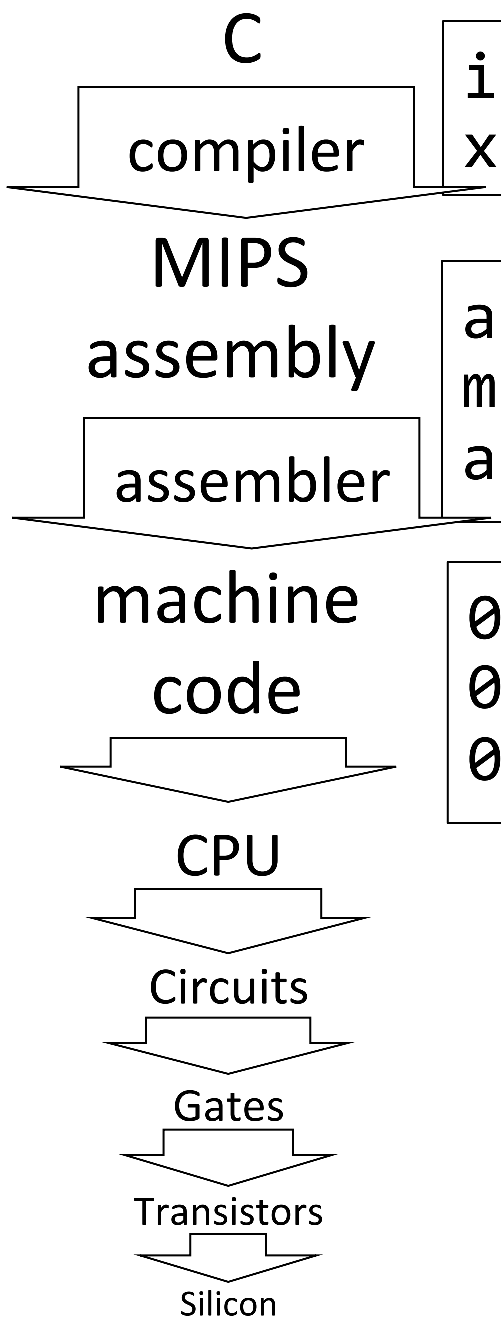
Assemblers, Linkers, and Loaders

Kevin Walsh
CS 3410, Spring 2011
Computer Science
Cornell University

See: P&H Appendix B.3-4



```
int x = 10;
x = 2 * x + 15;
```

On Disk

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

```
00100000000001010000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

In Memory

calc.c

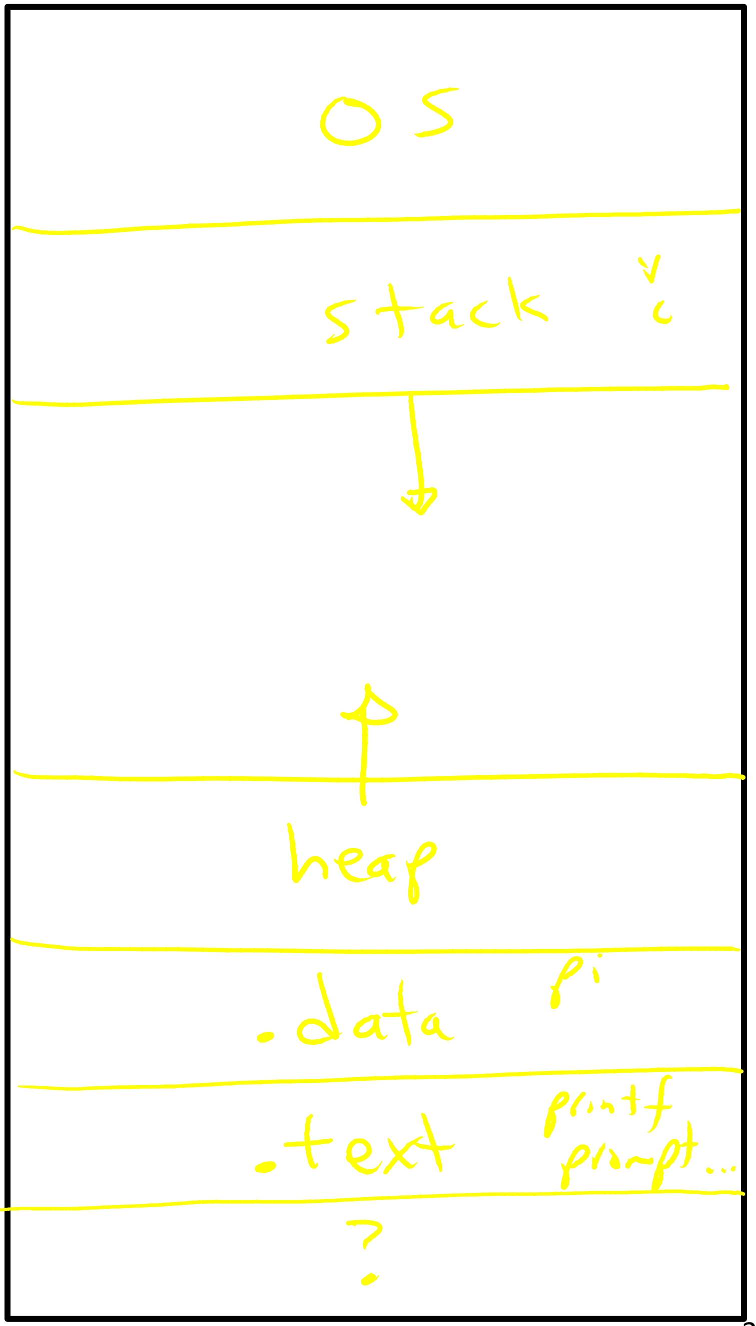
```
vector v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result", c);
```

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
```



```
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);
}
```

```
[csug01] mipsel-linux-gcc –S add1To100.c
```

```
        .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)
```

n

constant string

prolog

```
$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
```

call printf.

epilog

Variables	Visibility	Lifetime	Location
Function-Local i, m, count, A, argc, argv	within function	function invocation	stack
Global n	whole program	program execution	.data
Dynamic *A	?	between malloc and free	heap

```
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);
}
```

Variables	Visibility	Lifetime	Location
Function-Local i, m, count, A, argc, argv	within function	function invocation	stack
Global n	whole program	program execution	.data
Dynamic *A	?	between malloc and free	heap

C Pointers can be trouble

```
int *trouble()
{ int a; …; return &a; }
```

"address of" something on stack!

```
char *evil()
{ char s[20]; gets(s); return s;
```

also on stack!
Both invalid after return!

```
int *bad()
{ s = malloc(20); … free(s); … return s; }
```

points to free space in heap

(Can't do this in Java, C#, …)

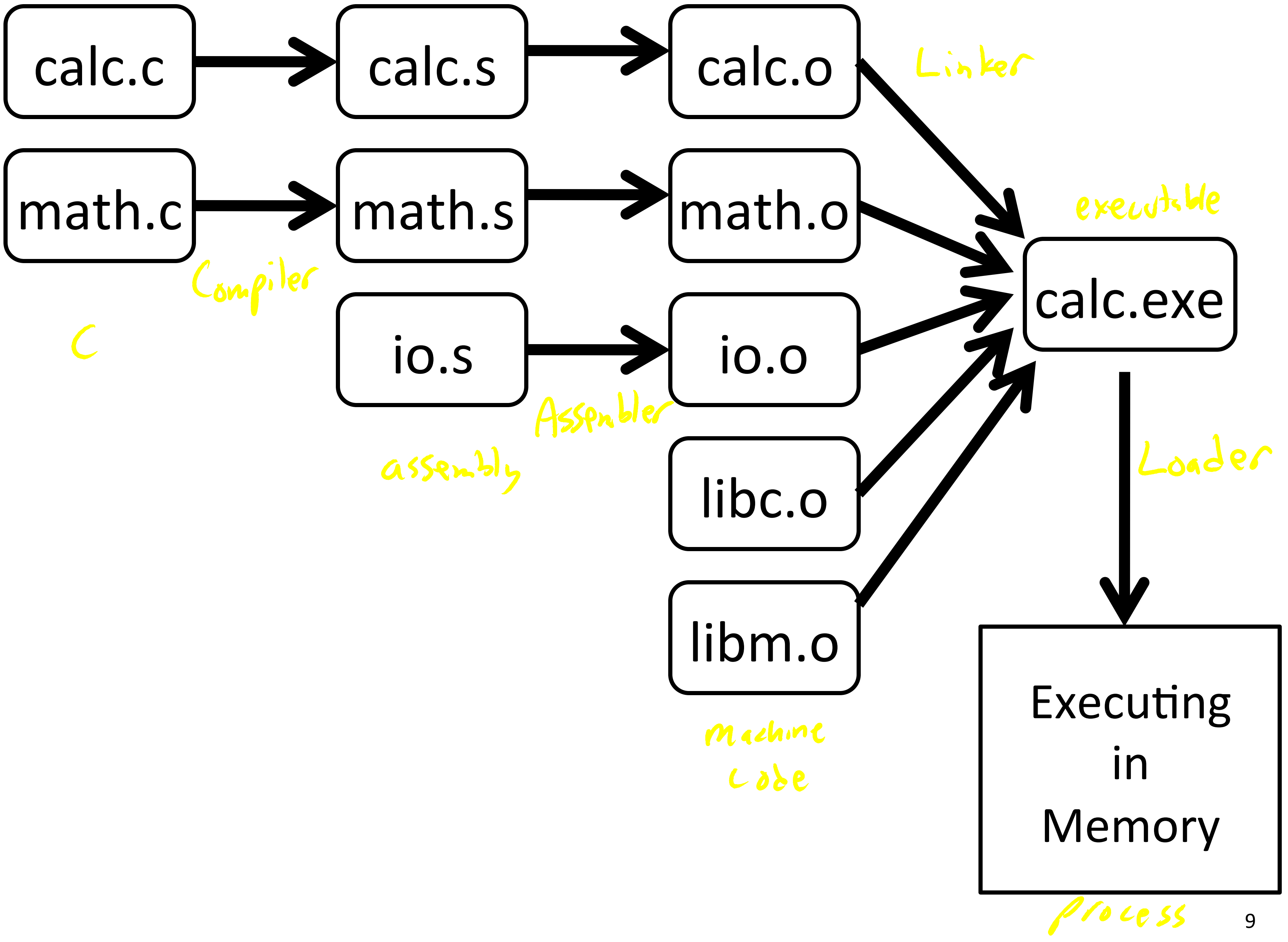
Variables	Visibility	Lifetime	Location
Function-Local i, m, count, A, argc, argv	within function	function invocation	stack
Global n	whole program	program execution	.data
Dynamic *A	?	between malloc and free	heap

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; }
```

Banned in Java, …

(Can’t do this in Java, C#, …)


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
Linker
executable
Compiler
C
Assembler
assembly
Loader
machine code
process

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

Compilers and Assemblers

$$\begin{tikzcd}\boxed{\text{math.c}} \arrow[r] & \boxed{\text{math.s}} \arrow[r] & \boxed{\text{math.o}}\end{tikzcd}$$

.o = linux
.obj = windows

- call to exported function
- reference to exported var

- call to extern function
- reference extern variable

?

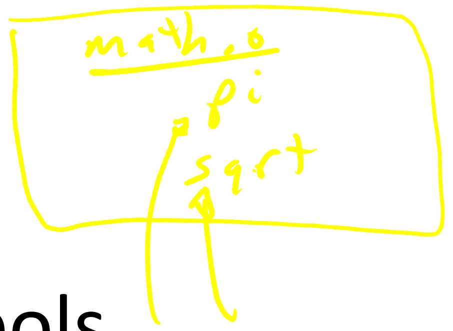
Output is obj files

- 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

- code starts at 0x0
- data starts at 0x0

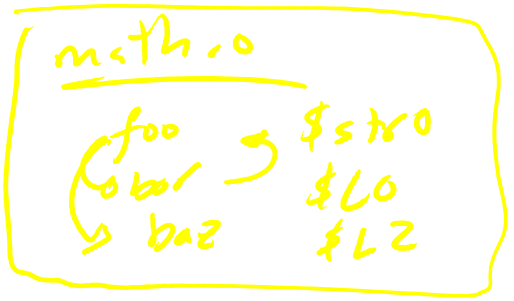
Global labels: Externally visible "exported" symbols

- 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

Header
- Size and position of pieces of file

Text Segment
- instructions

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

Debugging Information
- line number $\rightarrow$ code address map, etc.

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

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;
}
```

global

local only

Defined in some other file

global

local

global

```
gcc -S ... math.c
gcc -c ... math.s
objdump --disassemble math.o
objdump --syms math.o
```

compiler

assembler

reverse assembler

List symbols

```
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)
 ...
```

addresses

Mem[8]

This is wrong address — Need to fix to point to data section!

prolog

Body: $v0 = 0$
$v0 = mem[8 + v0]$
$= mem[8 + 0]$
$= mem[8]$
$= 0x03a0f021$

should be
return random val
$v0 = 7$

(?!)

epilog

symbol

```
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
```

address

l = local
g = global

section

size

symbol

static/local function @ address 60 code size = 28

global 4 byte variable @ address 4

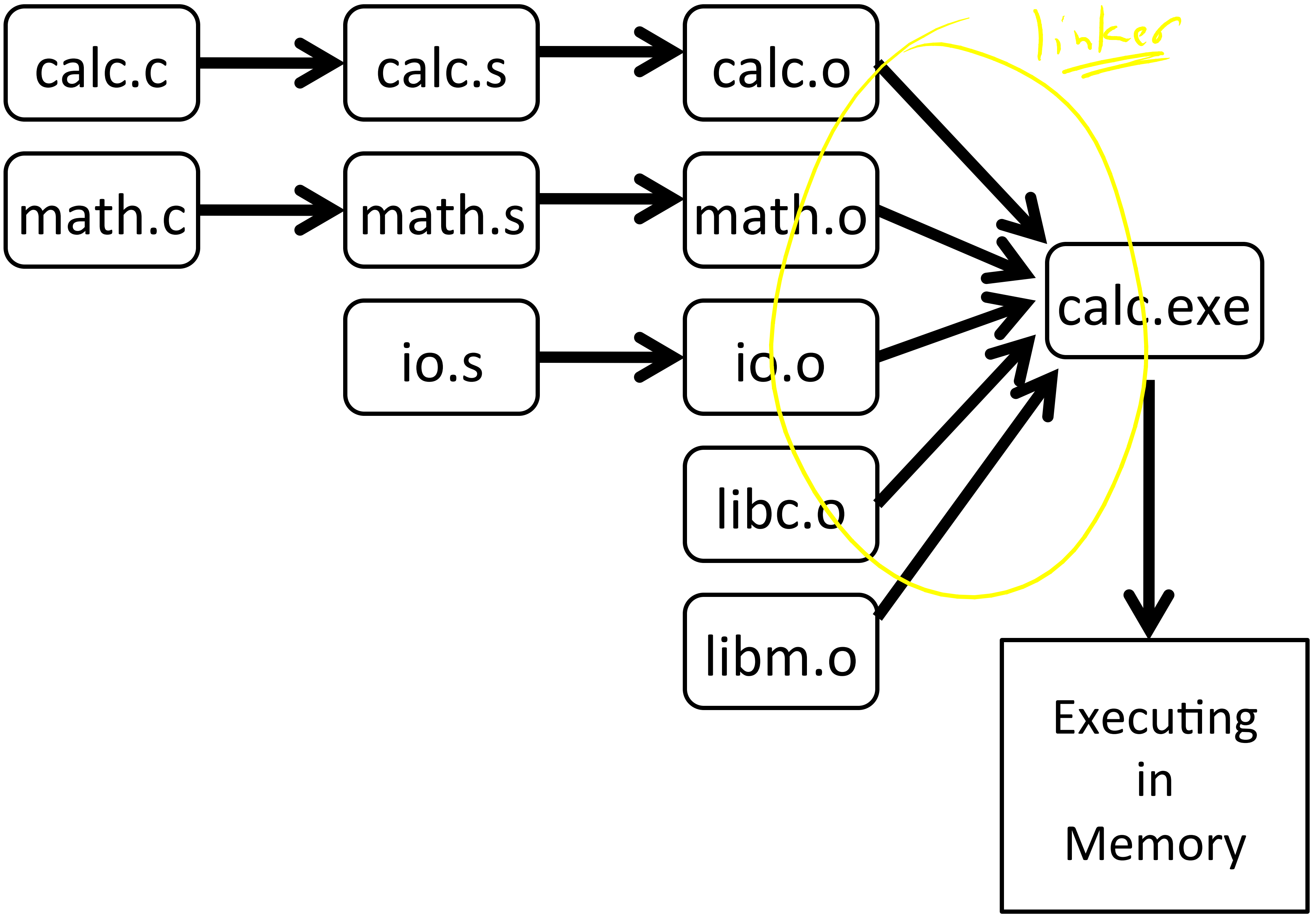
external references

F = function
O = var

Q: Why separate compile/assemble and linking steps?

A: Can recompile one object, then just relink.

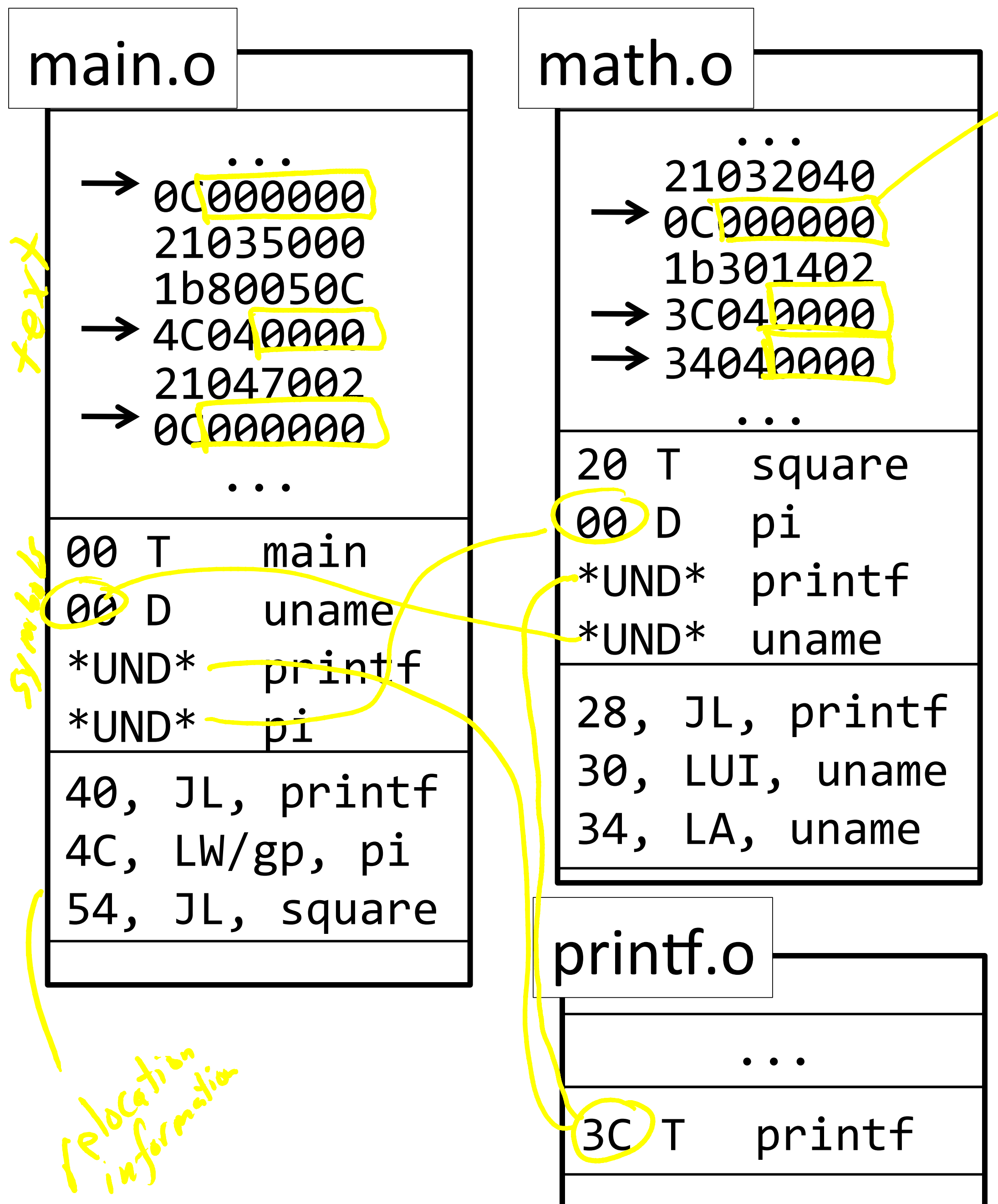
Linkers


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
linker

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



main.o

```
       ...
→ 0C000000
  21035000
  1b80050C
→ 4C040000
  21047002
→ 0C000000
       ...
```

```
00 T    main
00 D    uname
*UND*   printf
*UND*   pi
```

```
40, JL, printf
4C, LW/gp, pi
54, JL, square
```

math.o

```
       ...
  21032040
→ 0C000000
  1b301402
→ 3C040000
→ 34040000
       ...
```

```
20 T    square
00 D    pi
*UND*   printf
*UND*   uname
```

```
28, JL, printf
30, LUI, uname
34, LA, uname
```

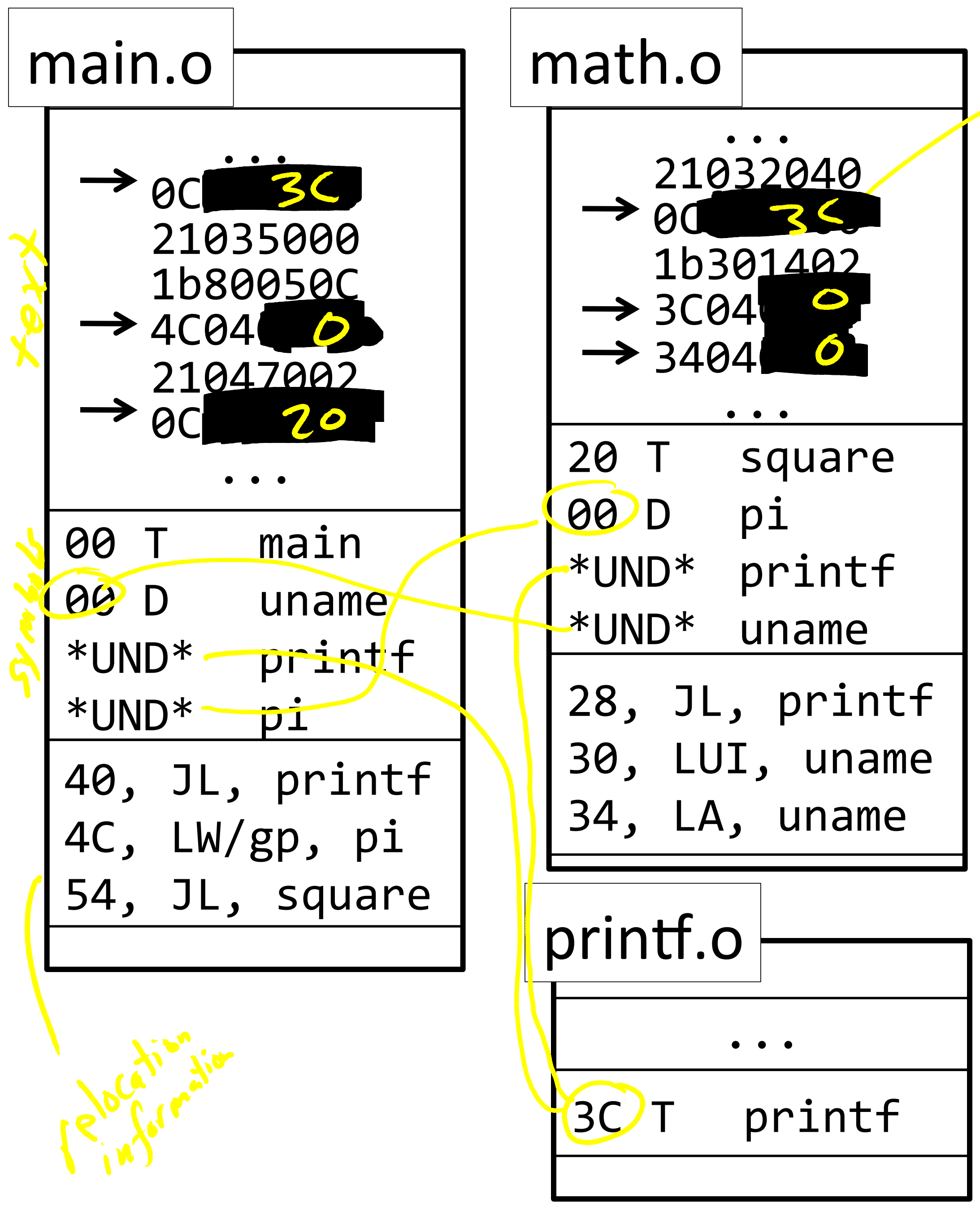
printf.o

```
    ...
```

```
3C T    printf
```

lots of ∅ bytes - these are external references, need to be fixed

① Find UND symbols in other tables



main.o

```
        ...
→ 0C[redacted]3C
  21035000
  1b80050C
→ 4C04[redacted]0
  21047002
→ 0C[redacted]20
        ...
```

```
00    T      main
00    D      uname
*UND* printf
*UND* pi
```

```
40, JL, printf
4C, LW/gp, pi
54, JL, square
```

math.o

```
        ...
  21032040
→ 0C[redacted]3C
  1b301402
→ 3C04[redacted]0
→ 3404[redacted]0
        ...
```

```
20    T    square
00    D    pi
*UND* printf
*UND* uname
```

```
28, JL, printf
30, LUI, uname
34, LA, uname
```

printf.o

```
   ...
3C T   printf
```

lots of ∅ bytes - these are external references, need to be fixed

① Find UND symbols in other tables

② Patch Code

→ Addresses collide

uname	@	0
pi	@	0
main	@	0
square	@	20

...

→ Need to relocate first

main.o

```
      ...
→ 0C000000
  21035000
  1b80050C      (2)
→ 4C040000
  21047002
→ 0C000000
      ...
00 T     main
00 D     uname  (B)
*UND*    printf
*UND*    pi
40, JL, printf
4C, LW/gp, pi
54, JL, square
```

math.o

```
      ...
  21032040
→ 0C000000
  1b301402      (1)
→ 3C040000
→ 34040000
      ...
20 T    square
00 D    pi   (A)
*UND*   printf
*UND*   uname
28, JL, printf
30, LUI, uname
34, LA, uname
```

printf.o

```
      ...      (3)
3C T    printf
```

calc.exe

```
        ...
   21032040
   0C40023C
   1b301402     (1)
   3C041000
   34040004
        ...
   0C40023C
   21035000
   1b80050c     (2)
   4C048004
   21047002
   0C400020
        ...
   10201000
   21040330     (3)
   22500102
        ...
pi     00000003  (A)
uname  0077616B  (B)
entry:400100
text: 400000
data:1000000
```

printf = 400200 + 3C

uname = 1000004

address 400000

400100

400200

1000000

1000004

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

Unix

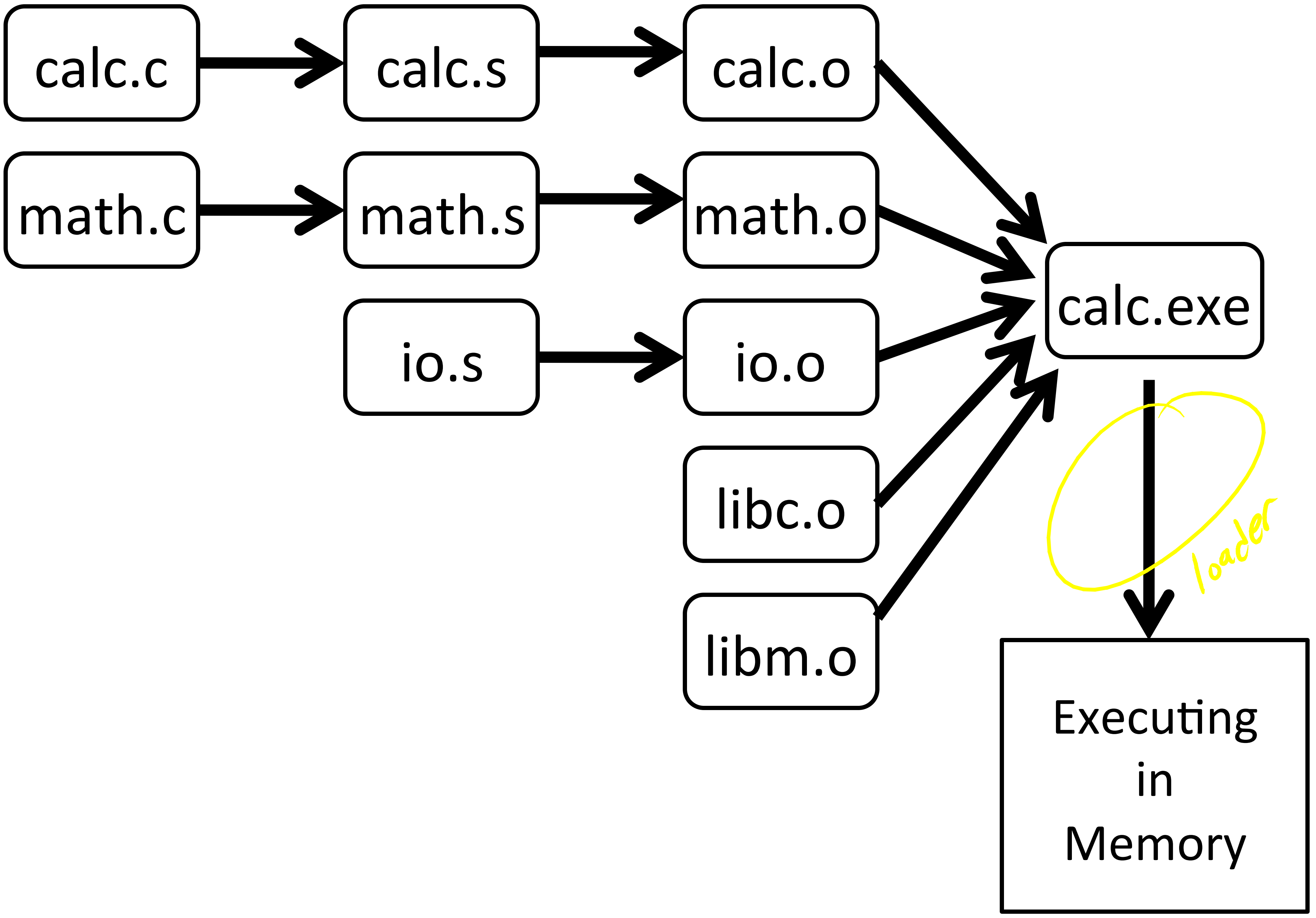
- a.out — older
- COFF: Common Object File Format
- ELF: Executable and Linking Format
- …

Windows

- PE: Portable Executable

All support both executable and object files

Loaders and Libraries


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

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

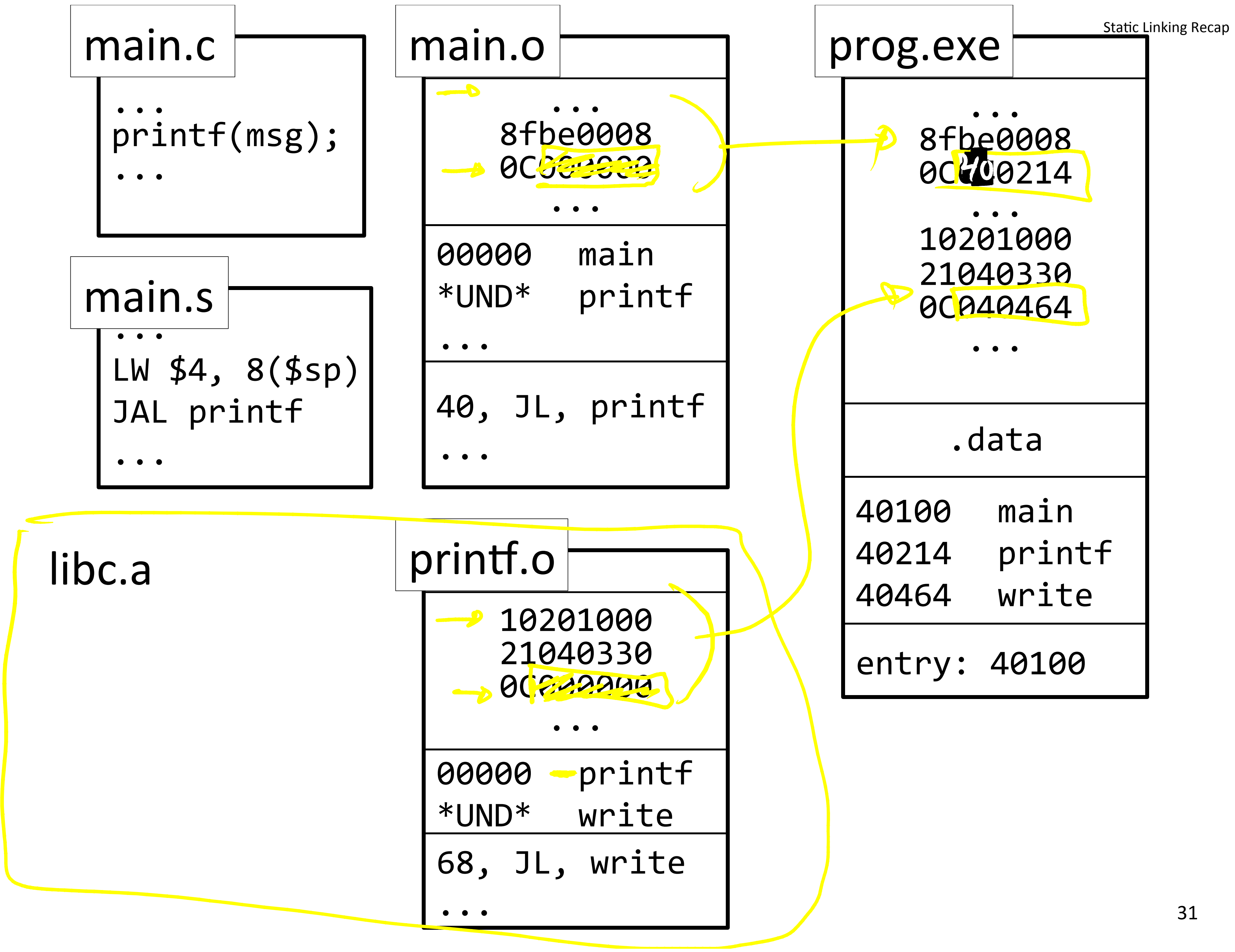
.a = linux
.lib = windows

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, ….



main.c

```
...
printf(msg);
...
```

main.s

```
...
LW $4, 8($sp)
JAL printf
...
```

main.o

```
    ...
    8fbe0008
    0C~~000000~~
    ...
00000   main
*UND*   printf
...
40, JL, printf
...
```

libc.a

printf.o

```
    10201000
    21040330
    0C~~000000~~
    ...
00000   printf
*UND*   write
68, JL, write
...
```

prog.exe

```
    ...
    8fbe0008
    0C040214
    ...
    10201000
    21040330
    0C040464
    ...
    .data
40100   main
40214   printf
40464   write
entry: 40100
```

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 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

```
00400010 <main>:
  LW temp, GOT[0]
  ~~jal 0x00400330~~
  JALR temp
  ...
  jal 0x00400620
  ...
  jal 0x00400330
  ...
00400330 <printf>:
  ...
00400620 <gets>:
  ...
```

GOT: global offset table

printf	400330
gets	400620

Indirect call:

```
00400010 <main>:
   ...
   lw t9, [redacted] -32708(gp)
   jalr t9
   ...
   lw t9, ? # gets
   jalr t9
   ...
00400330 <printf>:
   ...
00400620 <gets>:
   ...
```

```
# ~~data~~ segment
   ...
   ...
# global offset table
# to be loaded
# at -32712(gp)
.got
-712 .word 00400010 # main
-708 .word 00400330 # printf
-704 .word 00400620 # gets
   ...
```

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
RA 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
 if (!loaded) {
    load printf and fix table
 →  call printf
    return
```

Indirect call with on-demand dynamic linking:

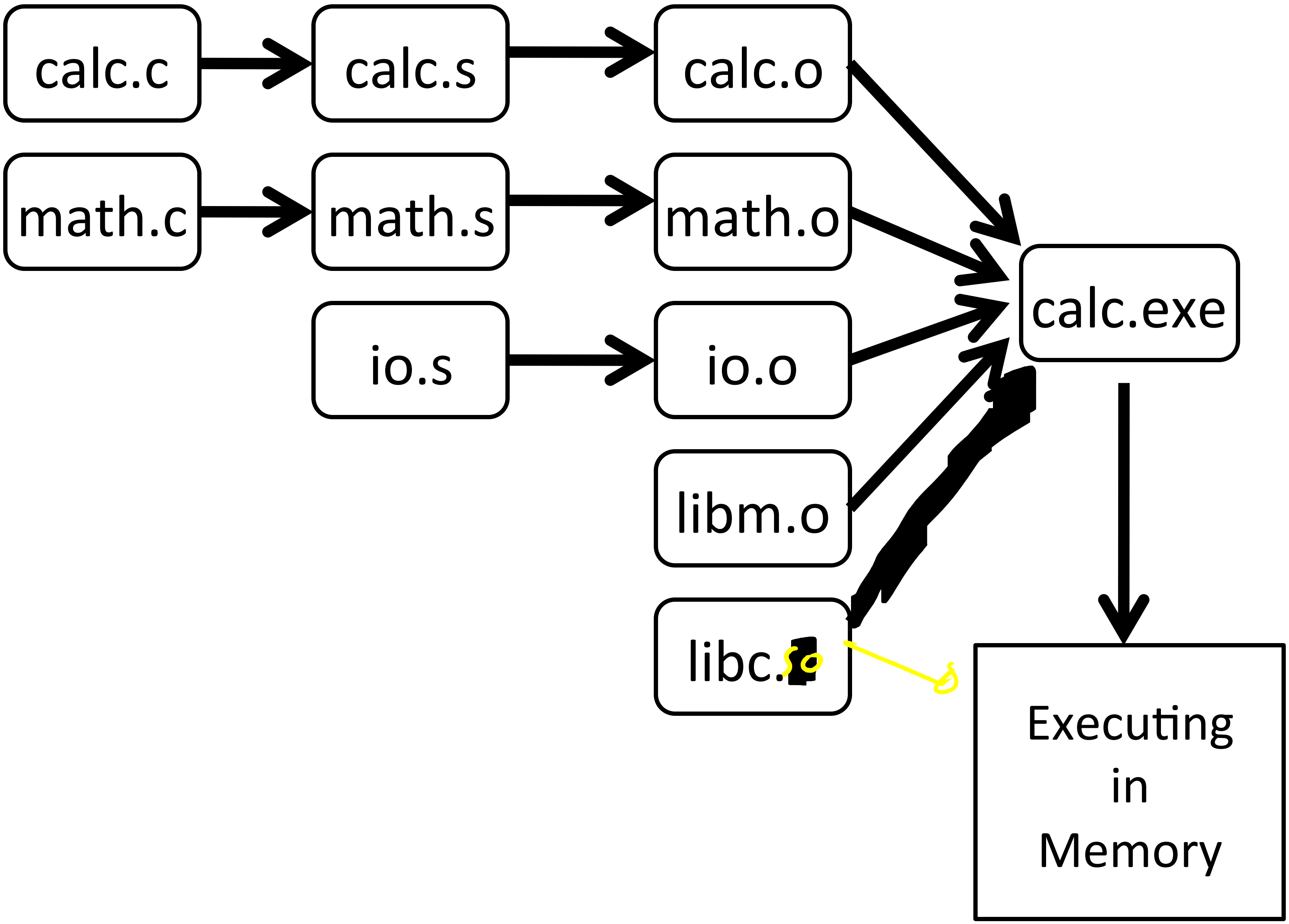
```
00400010 <main>:
   ...
   # load address of prints
   # from .got[1]
   lw t9, -32708(gp)
   # also load the index 1
   li t8, 1
   # now call it
   jalr t9
   ...
.got
   .word 00400888 # open
   .word 00400888 # prints
   .word 00400888 # gets
   .word 00400888 # foo
```

```
   ...
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
```


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

Windows: dynamically loaded library (DLL)

- PE format

Unix: dynamic shared object (DSO) .so

- 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 linking

-
-
-

Dynamic linking

-
-
-
- dll code is probably already in memory
- And can do the linking incrementally, on-demand