Assemblers, Linkers, and Loaders

Hakim Weatherspoon

CS 3410, Spring 2013

Computer Science

Cornell University

See: P&H Appendix B.3-4 and 2.12

Goal for Today: Putting it all Together

Review Calling Convention

Compiler output is assembly files

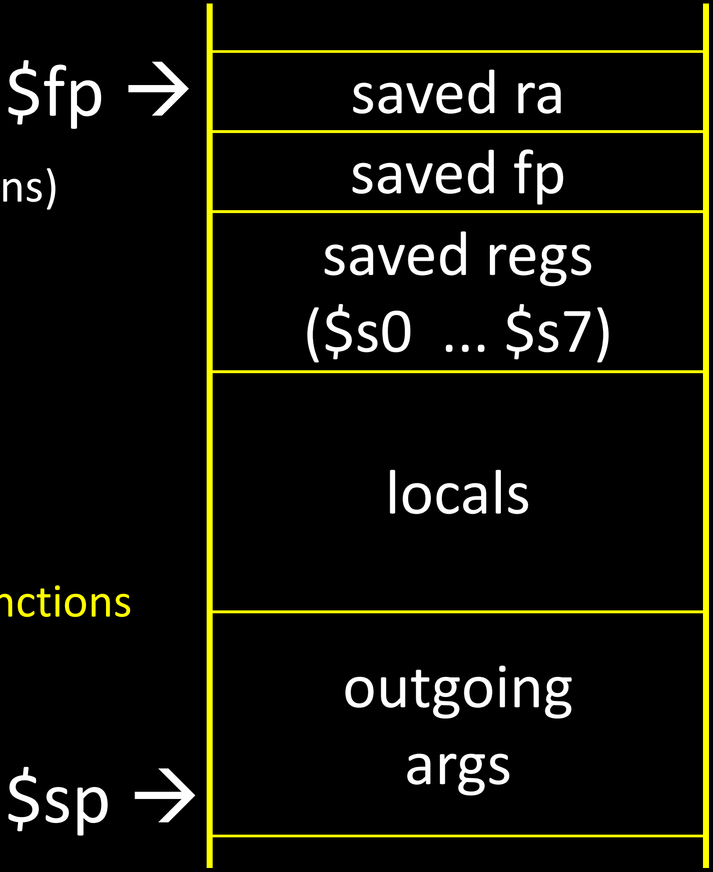
Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution

Recap: Calling Conventions

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent’s stack frame
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains $fp
  - contains local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions (i.e. argument “spilling)
  - contains space for first 4 arguments to sub-functions
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp



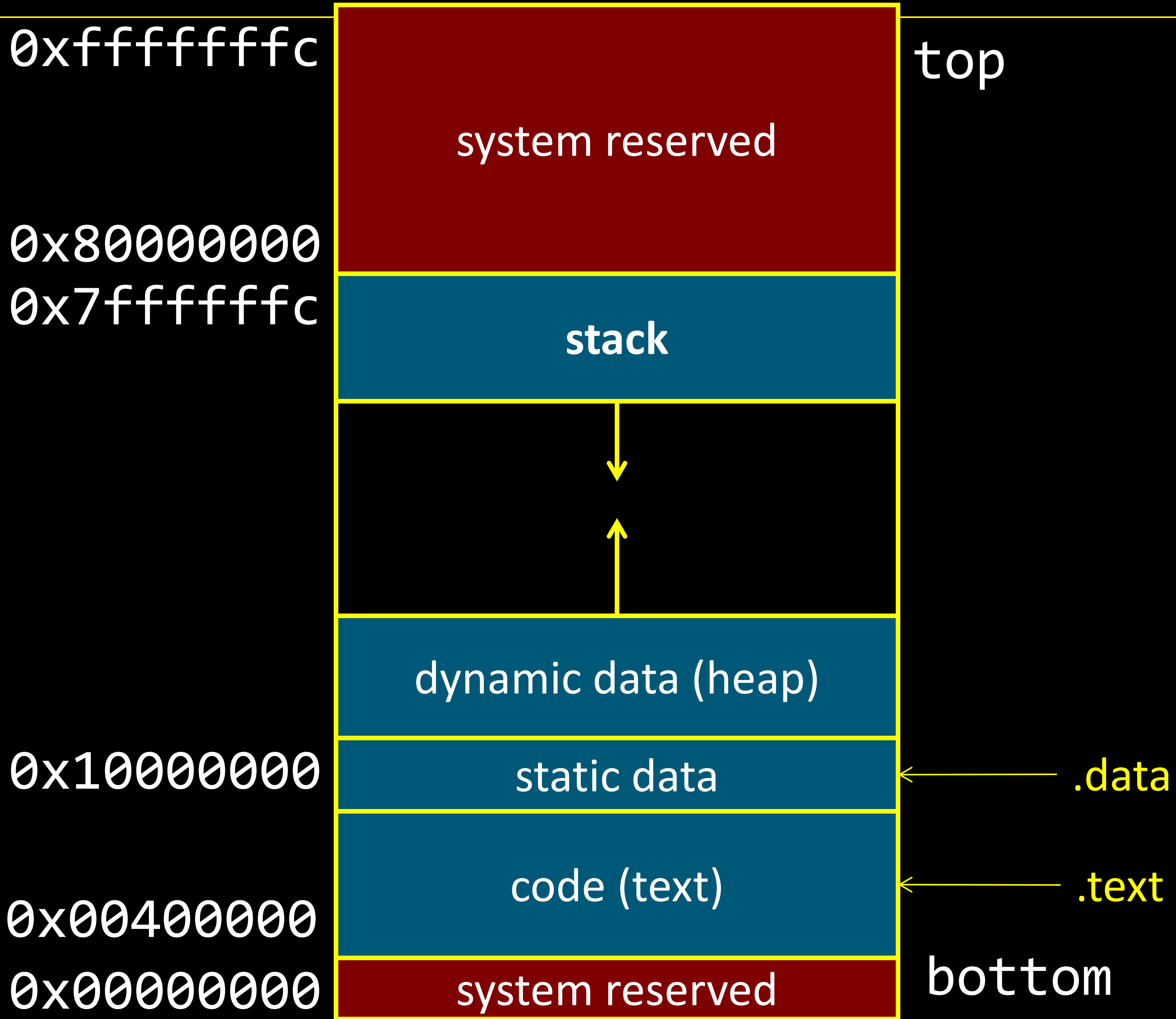
Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web

MIPS Register Conventions

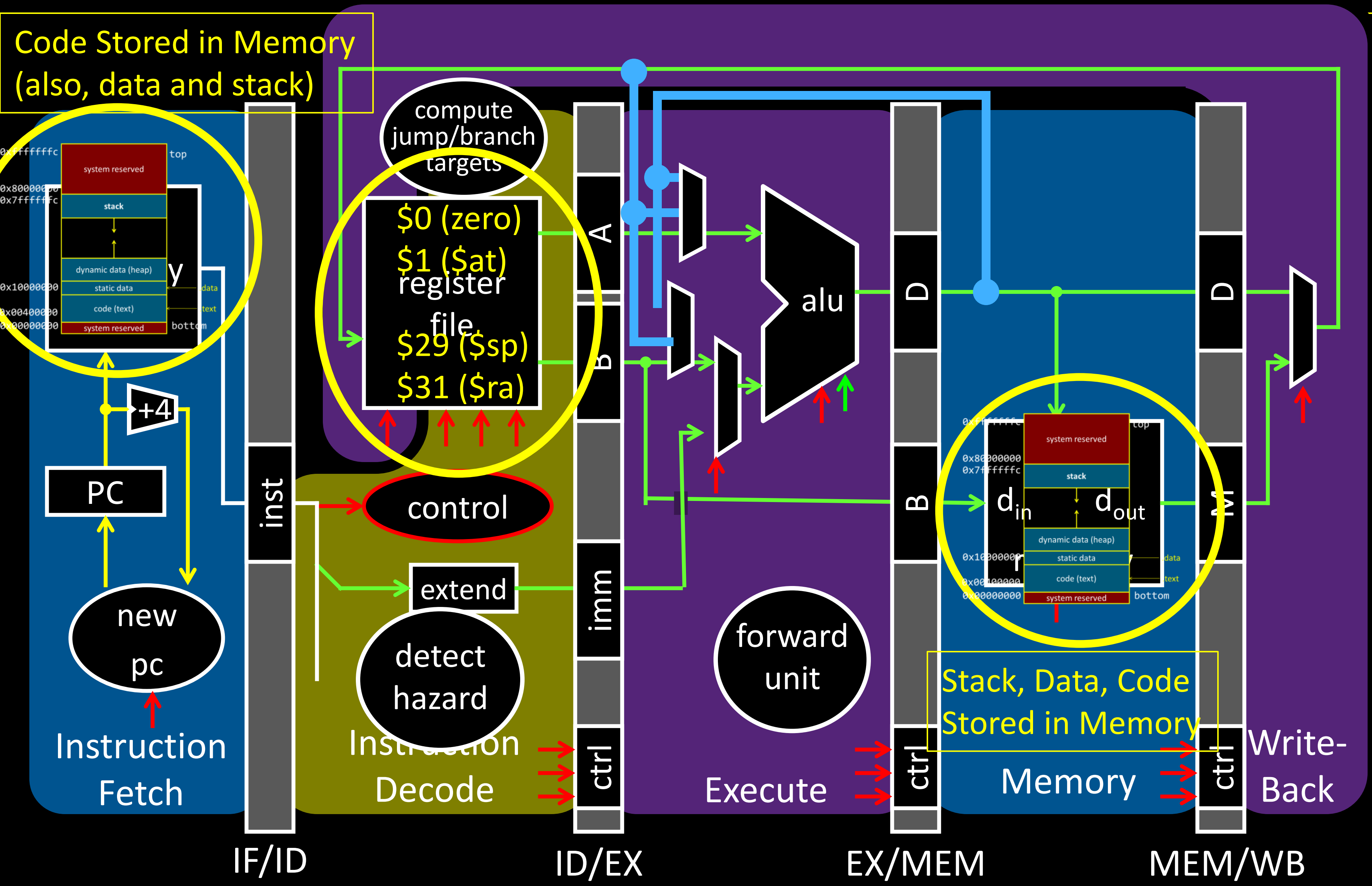
r0	$zero	zero
r1	$at	assembler temp
r2	$v0	function return values
r3	$v1	
r4	$a0	function arguments
r5	$a1	
r6	$a2	
r7	$a3	
r8	$t0	**temps (caller save)**
r9	$t1	
r10	$t2	
r11	$t3	
r12	$t4	
r13	$t5	
r14	$t6	
r15	$t7	

r16	$s0	**saved (callee save)**
r17	$s1	
r18	$s2	
r19	$s3	
r20	$s4	
r21	$s5	
r22	$s6	
r23	$s7	
r24	$t8	**more temps (caller save)**
r25	$t9	
r26	$k0	reserved for kernel
r27	$k1	
r28	$gp	global data pointer
r29	$sp	stack pointer
r30	$fp	frame pointer
r31	$ra	return address

Anatomy of an executing program



Anatomy of an executing program



Code Stored in Memory
(also, data and stack)

Stack, Data, Code
Stored in Memory

Takeaway

We need a calling convention to coordinate use of registers and memory. Registers exist in the Register File. Stack, Code, and Data exist in memory. Both instruction memory and data memory accessed through cache (modified harvard architecture) and a shared bus to memory (Von Neumann).

Next Goal

Given a running program (a process), how do we know what is going on (what function is executing, what arguments were passed to where, where is the stack and current stack frame, where is the code and data, etc)?

Activity #1: Debugging

init():	0x400000
printf(s, …):	0x4002B4
vnorm(a,b):	0x40107C
main(a,b):	0x4010A0
pi:	0x10000000
str1:	0x10000004

CPU:
$pc=0x004003C0
$sp=0x7FFFFFAC
$ra=0x00401090

What func is running?

Who called it?

Has it called anything?

Will it?

Args?

Stack depth?

Call trace?

	0x00000000
	0x0040010c
	0x7FFFFFF4
	0x00000000
	0x00000000
	0x00000000
	0x00000000
	0x004010c4
	0x7FFFFFDC
	0x00000000
	0x00000000
	0x00000015
0x7FFFFFB0	0x10000004
	0x00401090

Compilers and Assemblers

Next Goal

How do we compile a program from source to assembly to machine object code?

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

Example: Add 1 to 100

```
int n = 100;

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

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

          printf ("Sum 1 to %d is %d\n", n, sum);
}
```

```
export PATH=${PATH}:/courses/cs3410/mipsel-linux/bin:/courses/cs3410/mips-sim/bin
or
setenv PATH ${PATH}:/courses/cs3410/mipsel-linux/bin:/courses/cs3410/mips-sim/bin
```

```
# Assemble
[csug03] 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

```
# Assemble
[csug01] mipsel-linux-gcc –c add1To100.s

# 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, sum = 0, *A = malloc(4 * m);
        for (i = 1; i <= m; i++) { sum += i; A[i] = sum; }
        printf ("Sum 1 to %d is %d\n", n, sum);
}
```

Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local i, m, sum	w/in func	func invocation	stack
Global n, str	whole prgm	prgm execution	.data
Dynamic A **C Pointers can be trouble**	Anywhere that has a ptr	b/w malloc and free	heap

Example #2: Review of Program Layout

calc.c

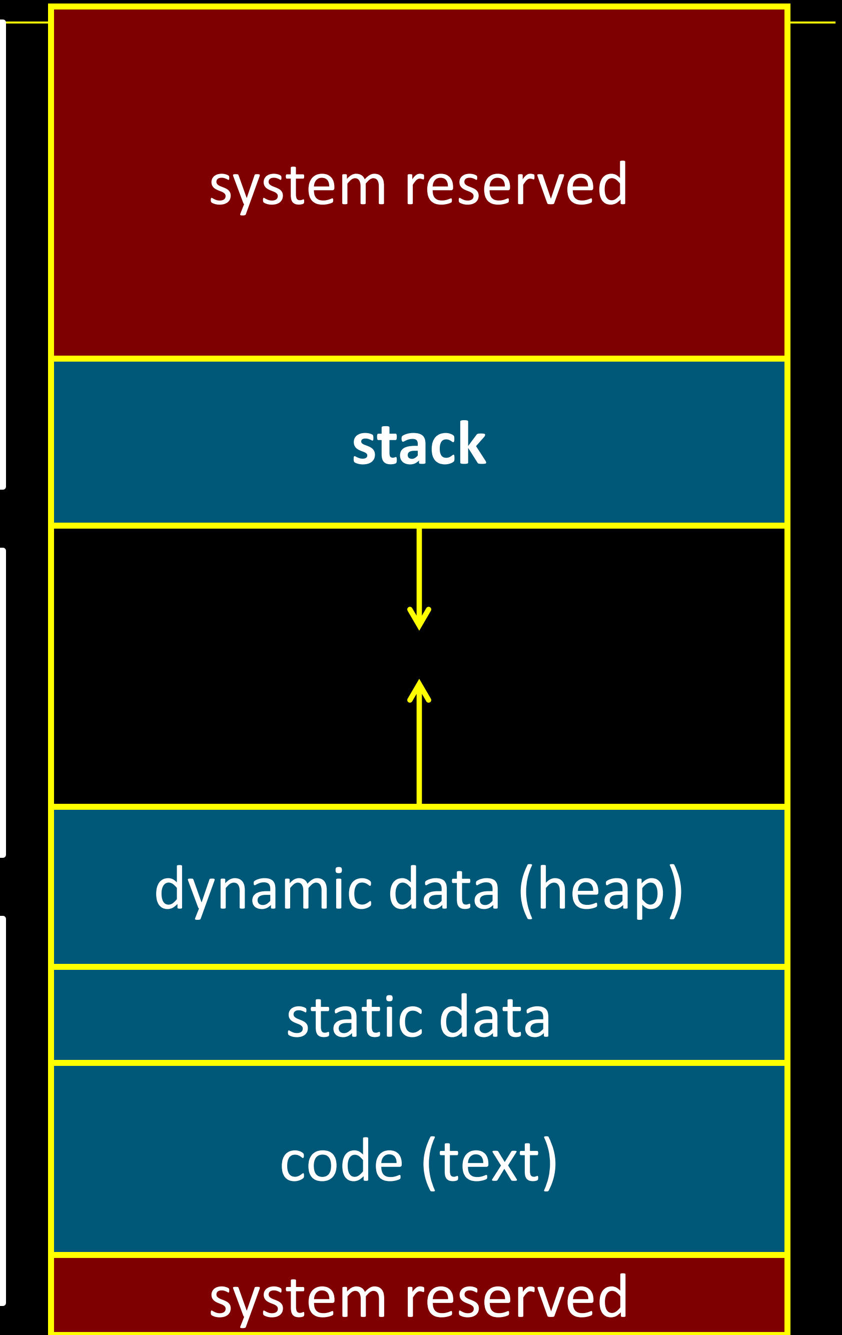
```
vector* v = malloc(8);
v->x = prompt(“enter x”);
v->y = prompt(“enter y”);
int c = pi + tnorm(v);
print(“result %d”, c);
```

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
```



system reserved

stack

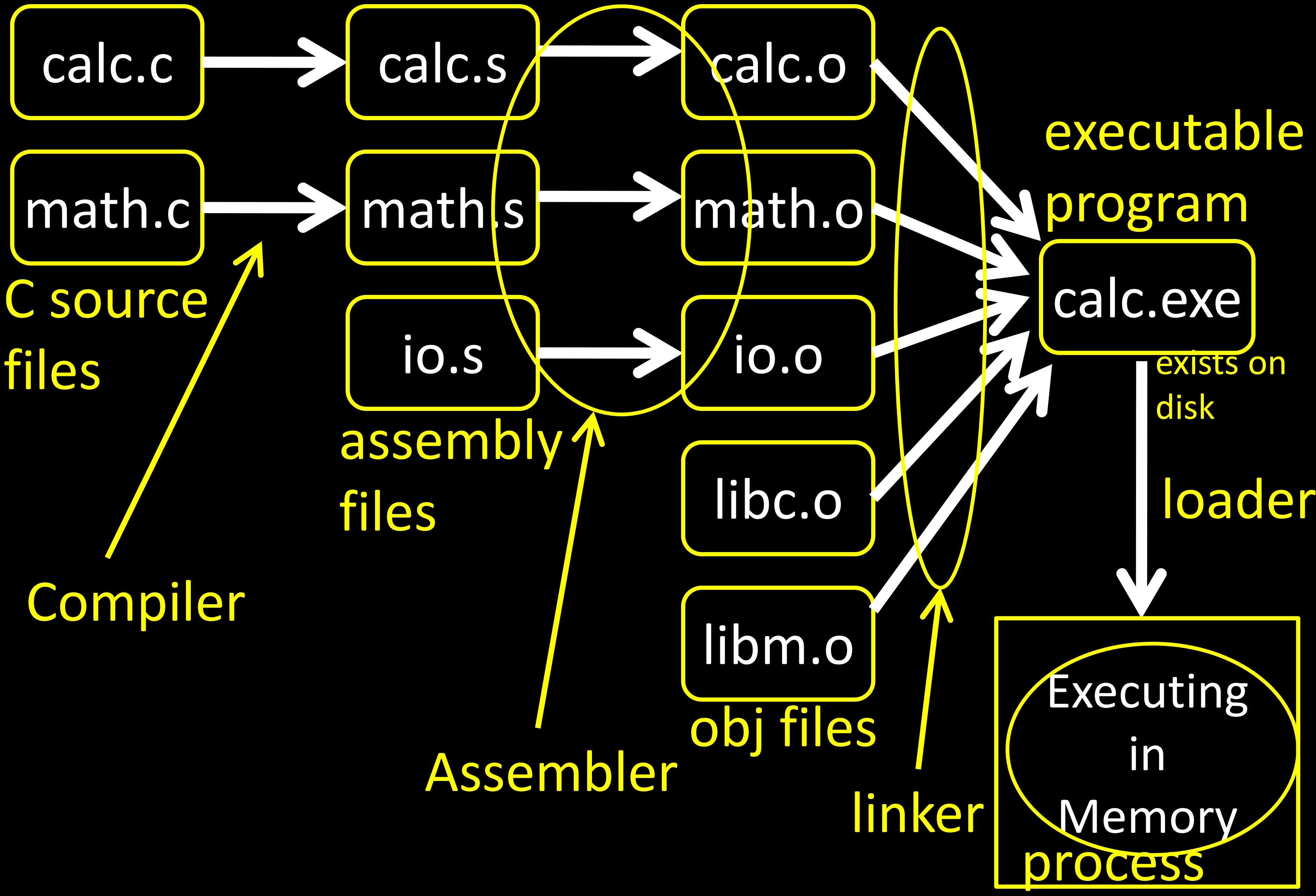
dynamic data (heap)

static data

code (text)

system reserved

Assembler



calc.c

math.c

C source files

Compiler

calc.s

math.s

io.s

assembly files

Assembler

calc.o

math.o

io.o

libc.o

libm.o

obj files

linker

executable program

calc.exe

exists on disk

loader

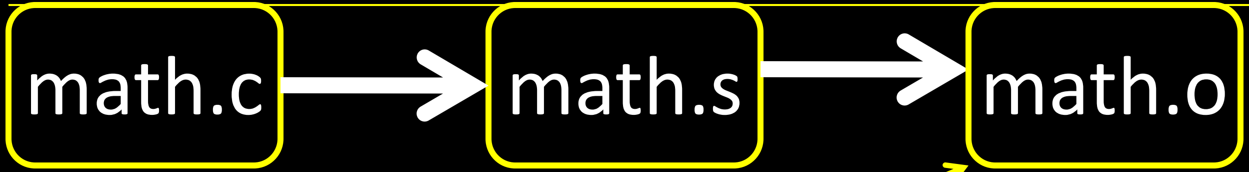
Executing in Memory

process

Next Goal

How do we understand the machine object code that an assembler creates?

Big Picture



.o = Linux
.obj Windows

Output is obj files

- Binary machine code, but not executable
- 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

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

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;
}
```

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.

Takeaway

We need a calling convention to coordinate use of registers and memory. Registers exist in the Register File. Stack, Code, and Data exist in memory. Both instruction memory and data memory accessed through cache (modified harvard architecture) and a shared bus to memory (Von Neumann).

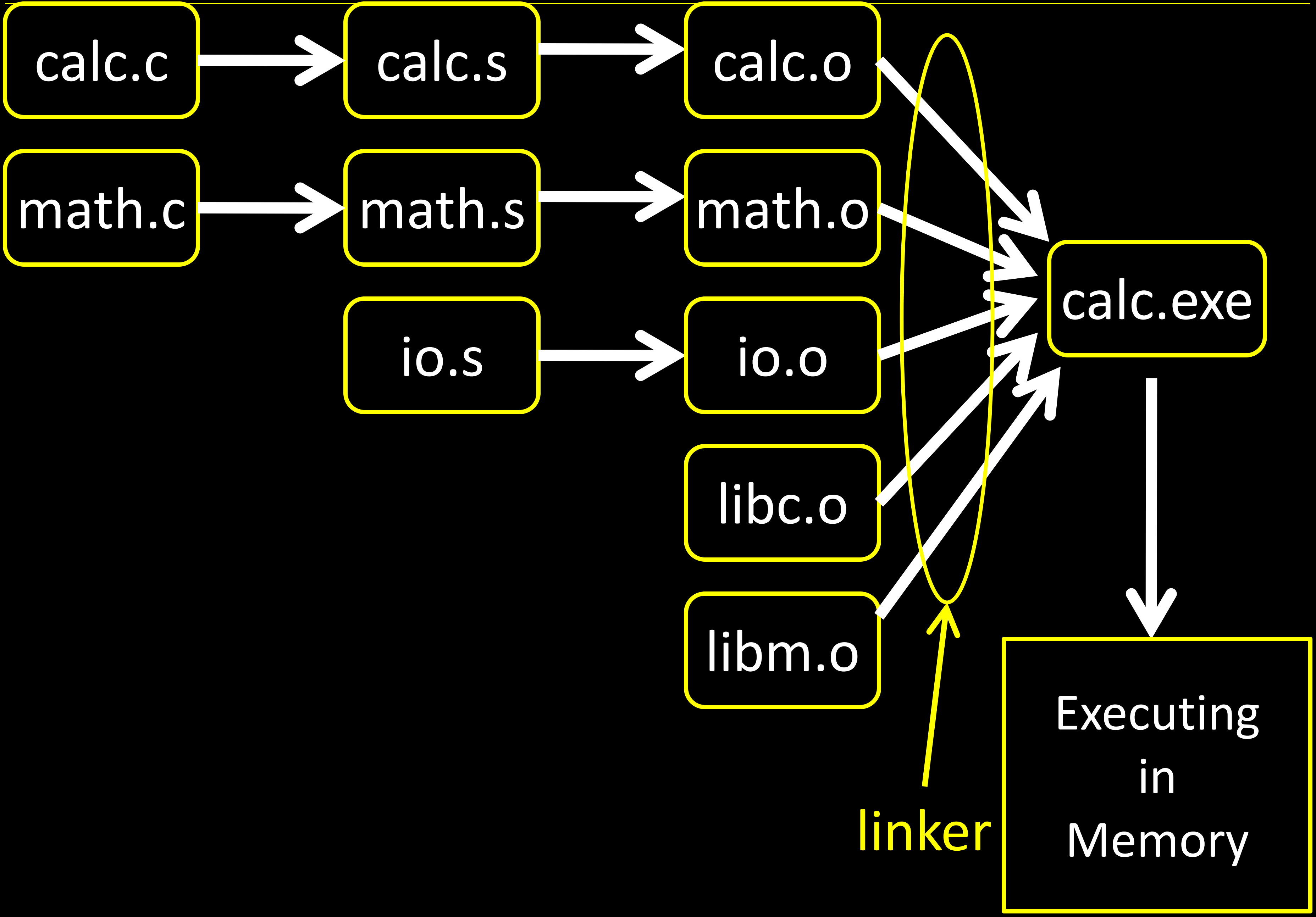
Need to **compile** from a high level source language to **assembly**, then **assemble** to machine object code. The Objdump command can help us understand structure of machine code which is broken into hdr,   txt and data segments, debugging information, and symbol table

Linkers

Next Goal

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

Big Picture


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

Linkers

Linker combines object files into an executable file

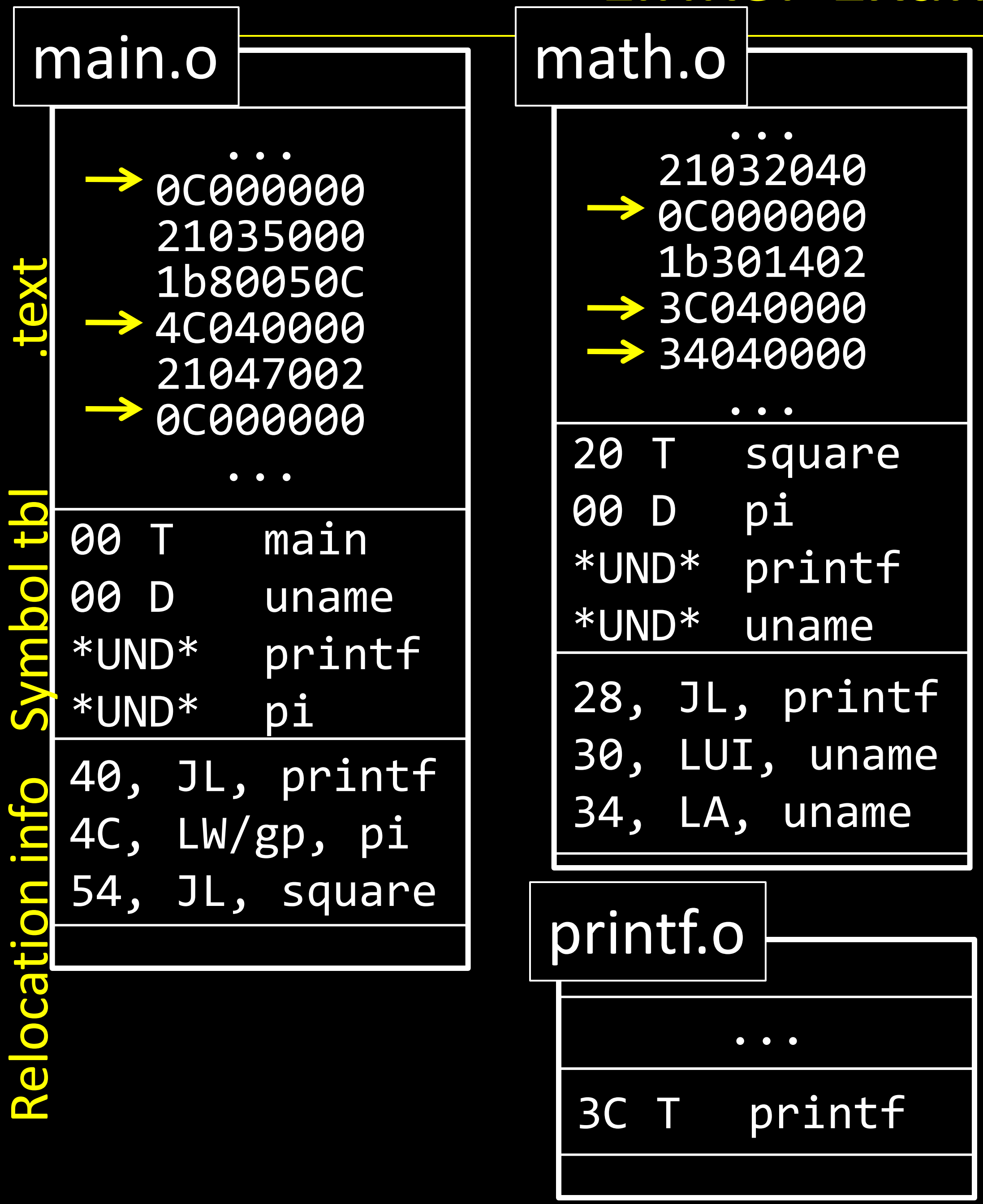
- Relocate each object's text and data segments
- Resolve as-yet-unresolved symbols
- Record top-level entry point in executable file

End result: a program on disk, ready to execute

- E.g.

./calc	Linux
./calc.exe	Windows
simulate calc	Class MIPS simulator

Linker Example



main.o

.text

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

Symbol tbl

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

Relocation info

```
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
```

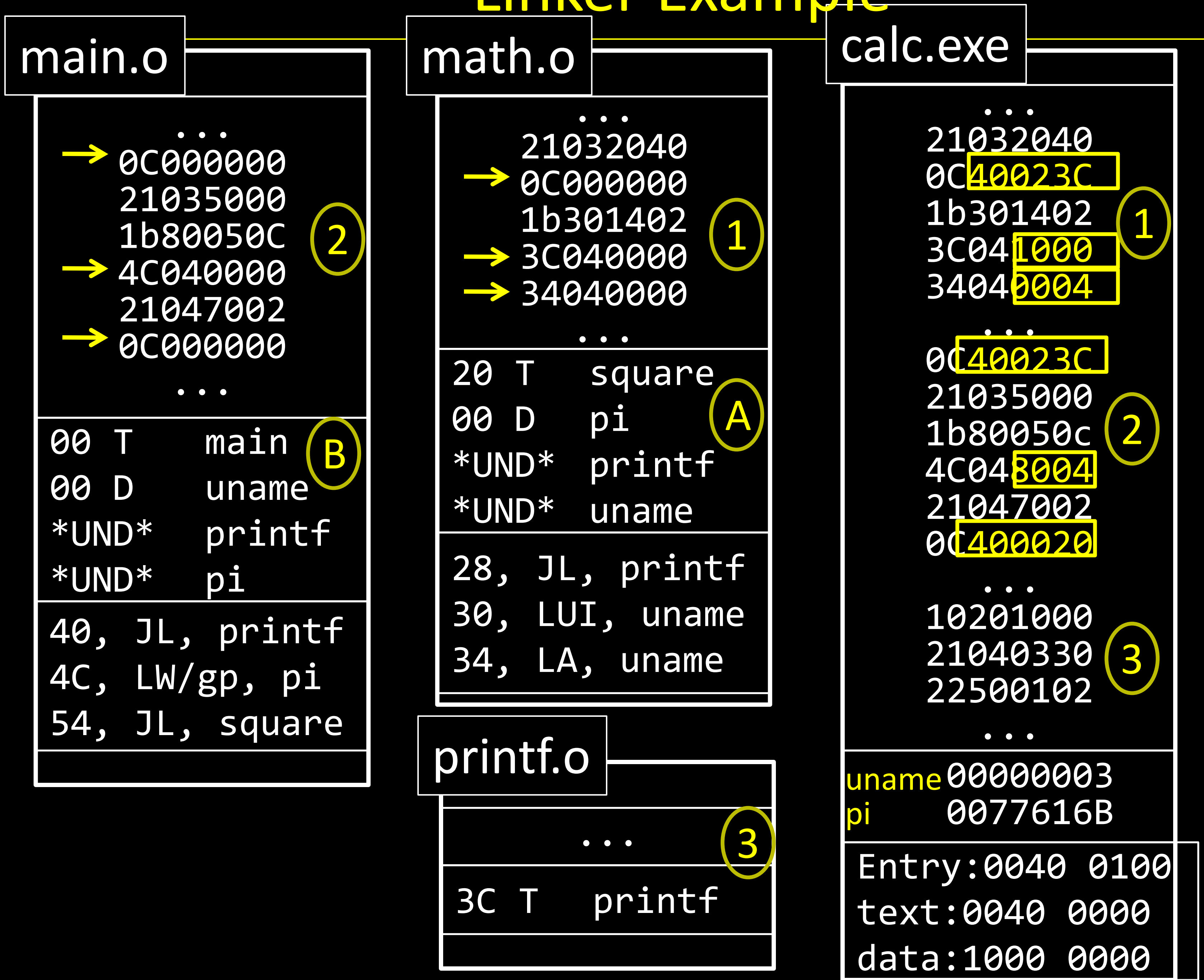
```
28, JL, printf
30, LUI, uname
34, LA, uname
```

printf.o

```
...
```

```
3C T   printf
```

Linker Example



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
```

calc.exe

```
...
21032040
0C40023C
1b301402
3C041000
34040004
...
0C40023C
21035000
1b80050c
4C048004
21047002
0C400020
...
10201000
21040330
22500102
...
```

```
uname 00000003
pi    0077616B
```

```
Entry:0040 0100
text:0040 0000
data:1000 0000
```

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

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

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

Administrivia

Upcoming agenda

- Schedule PA2 Design Doc Mtg for ***next*** Monday, Mar 11th
- HW3 due next Wednesday, March 13th
- PA2 Work-in-Progress circuit due ***before*** spring break

- Spring break: Saturday, March 16th to Sunday, March 24th

- Prelim2 Thursday, March 28th, right after spring break
- PA2 due Thursday, April 4th