

# Assemblers, Linkers, and Loaders

**Prof. Hakim Weatherspoon**

**CS 3410, Spring 2015**

Computer Science

Cornell University

See: P&H Appendix A1-2, A.3-4 and 2.12

# Announcement

## Upcoming agenda

- PA1 due yesterday
- PA2 available and discussed during lab section this week
- PA2 Work-in-Progress due Monday, March 16<sup>th</sup>
- PA2 due Thursday, March 26<sup>th</sup>
- HW2 available next week, due before Prelim2 in April
- **Spring break:** Saturday, March 28<sup>th</sup> to Sunday, April 5<sup>th</sup>

# Goal for Today: Putting it all Together

Brief review of calling conventions

Compiler output is assembly files

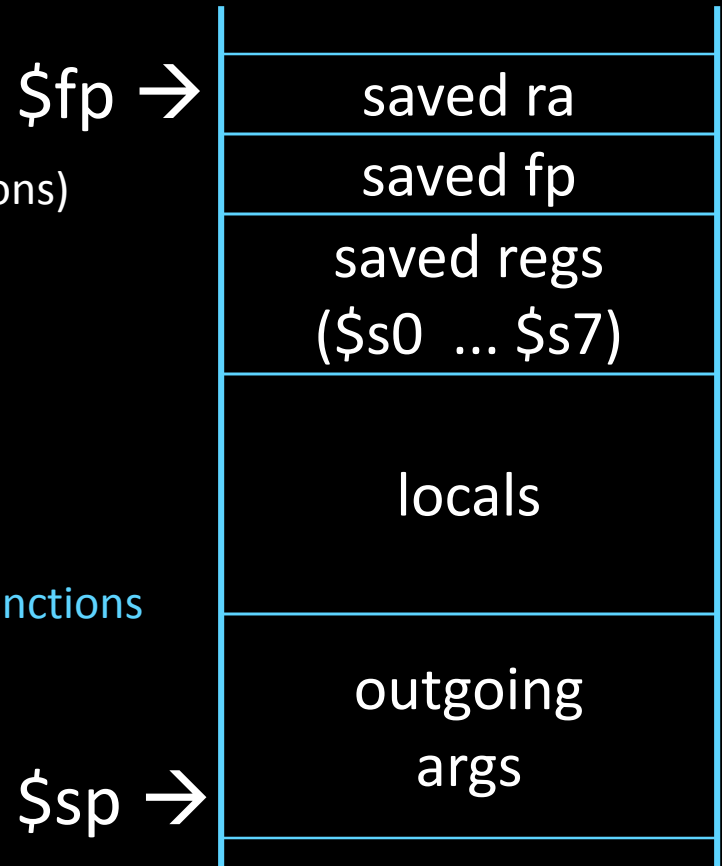
Assembler output is obj files

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 preserved
- Global data accessed via  $\$gp$

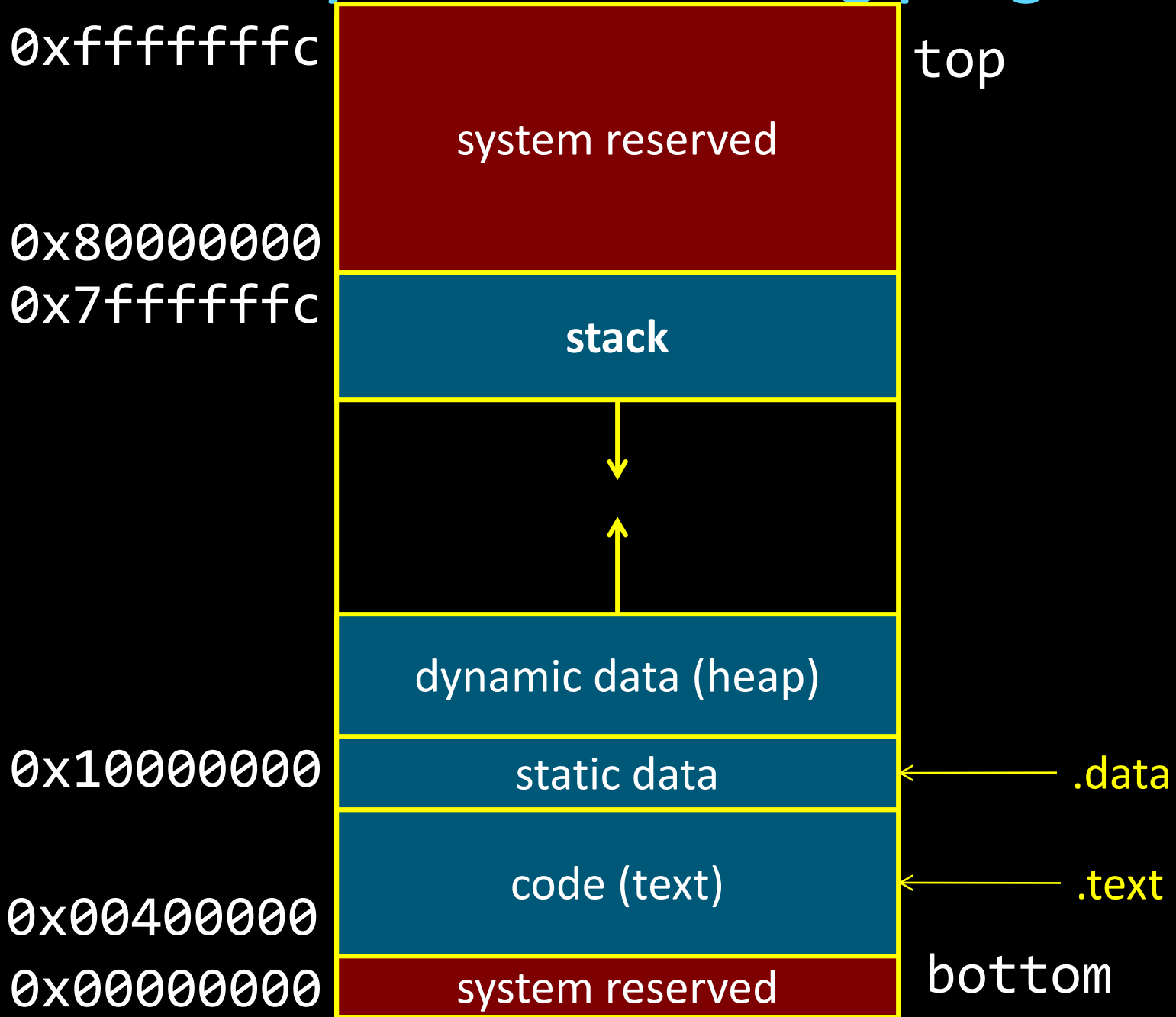


**Warning:** There is no one true MIPS calling convention.  
lecture != book != gcc != spim != web

# MIPS Register Conventions

r0	\$zero	zero	r16	\$s0	<b>saved (callee save)</b>
r1	\$at	assembler temp	r17	\$s1	
r2	\$v0	function return values	r18	\$s2	
r3	\$v1		r19	\$s3	
r4	\$a0	function arguments	r20	\$s4	
r5	\$a1		r21	\$s5	
r6	\$a2		r22	\$s6	
r7	\$a3		r23	\$s7	
r8	\$t0	<b>temps (caller save)</b>	r24	\$t8	<b>more temps (caller save)</b>
r9	\$t1		r25	\$t9	reserved for kernel
r10	\$t2		r26	\$k0	
r11	\$t3		r27	\$k1	
r12	\$t4		r28	\$gp	global data pointer
r13	\$t5		r29	\$sp	stack pointer
r14	\$t6		r30	\$fp	frame pointer
r15	\$t7		r31	\$ra	return address

# Anatomy of an executing program





# 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).



# Compilers and Assemblers

# Next Goal

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

# Big Picture

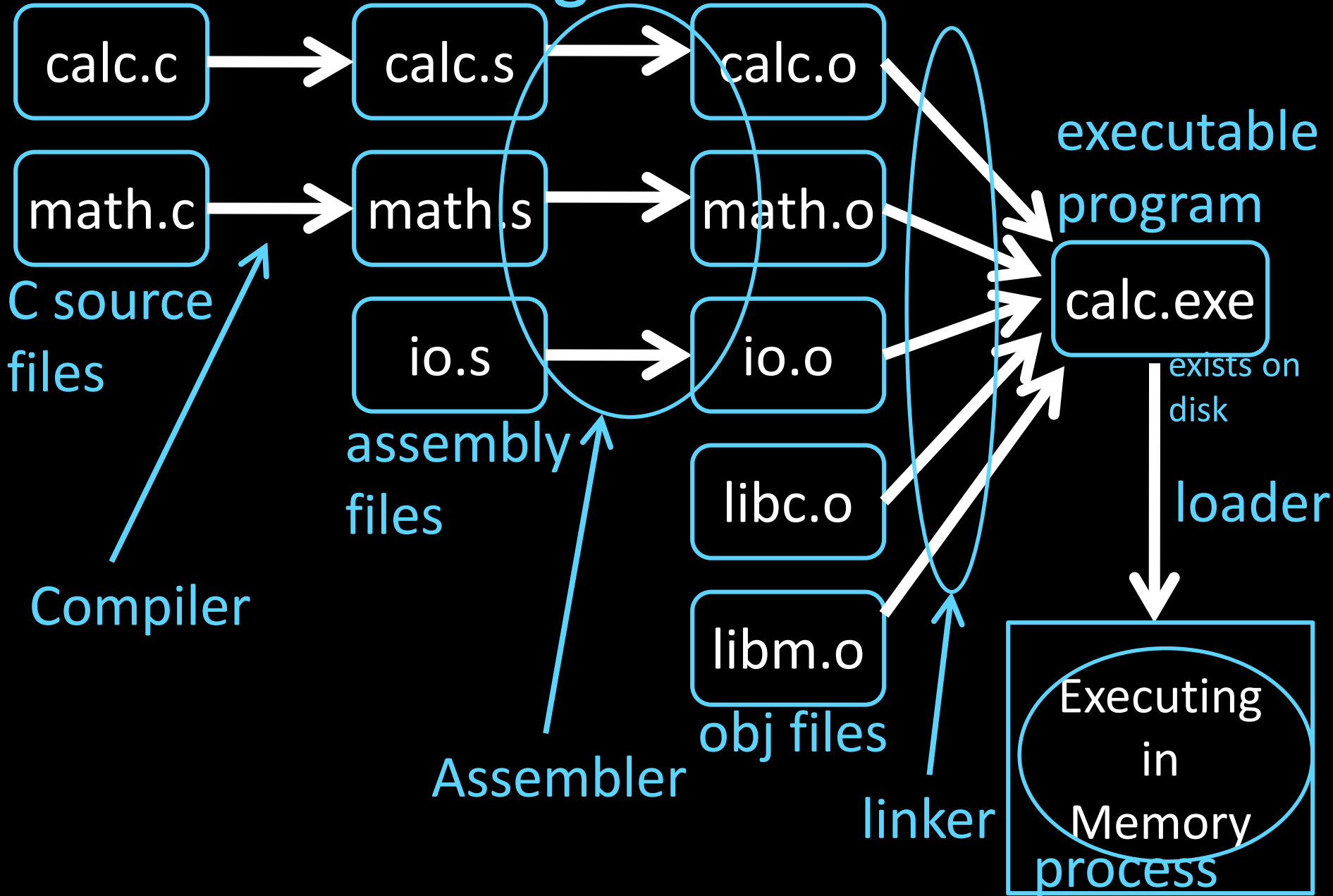
Compiler output is assembly files

Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution

# Big Picture



# Next Goal

How do we (as humans or compiler) program on top of a given ISA?

# Assembler

Translates text *assembly language* to **binary machine code**

**Input:** a text file containing MIPS instructions in human readable form

```
addi r5, r0, 10
mul1 r5, r5, 2
addi r5, r5, 15
```

**Output:** an **object file** (.o file in Unix, .obj in Windows) containing MIPS instructions in executable form

```
001000000000010100000000000001010
000000000000001010010100001000000
001000001010010100000000000001111
```

# Assembly Language

Assembly language is used to specify programs at a low-level

Will I program in assembly?

# Assembly Language

Assembly language is used to specify programs at a low-level

What does a program consist of?

- MIPS instructions
- Program data (strings, variables, etc)



# Assembler

Assembler:

Input:

- assembly instructions
- + psuedo-instructions
- + data and layout directives

Output:

Object file

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

# Assembler

Assembler:

Input:

assembly instructions

+ psuedo-instructions

+ data and layout directives

Output:

Object File

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

# MIPS Assembly Language Instructions

## Arithmetic/Logical

- ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
- ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
- MULT, DIV, MFLO, MTLO, MFHI, MTHI

## Memory Access

- LW, LH, LB, LHU, LBU, LWL, LWR
- SW, SH, SB, SWL, SWR

## Control flow

- BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
- J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

## Special

- LL, SC, SYSCALL, BREAK, SYNC, COPROC

# Assembler

Assembler:

Input:

assembly instructions

+ psuedo-instructions

+ data and layout directives

Output:

Object file

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

# Pseudo-Instructions

## Pseudo-Instructions

### NOP # do nothing

- SLL r0, r0, 0

### MOVE reg, reg # copy between regs

- ADD R2, R0, R1 # copies contents of R1 to R2

### LI reg, imm # load immediate (up to 32 bits)

### LA reg, label # load address (32 bits)

### B label # unconditional branch

### BLT reg, reg, label # branch less than

- SLT r1, rA, rB # r1 = 1 if  $R[rA] < R[rB]$ ; o.w. r1 = 0
- BNE r1, r0, label # go to address label if  $r1 \neq r0$ ; i.t.  $rA < rB$

# Assembler

Assembler:

Input:

assembly instructions

+ pseudo-instructions

+ data and layout directives

Output:

Object file

Slightly higher level than plain assembly

e.g: takes care of delay slots

(will reorder instructions or insert nops)

# Program Layout

Programs consist of **segments** used for different purposes

- **Text**: holds instructions
- **Data**: holds statically allocated program data such as variables, strings, etc.

data

```
"cornell cs"  
13  
25
```

text

```
add r1,r2,r3  
ori r2, r4, 3  
...
```

# Assembling Programs

Assembly files consist of a mix of

+ instructions

+ pseudo-instructions

+ assembler (data/layout) directives  
(Assembler lays out binary values  
in memory based on directives)

Assembled to an Object File

- Header
- Text Segment
- Data Segment
- Relocation Information
- Symbol Table
- Debugging Information

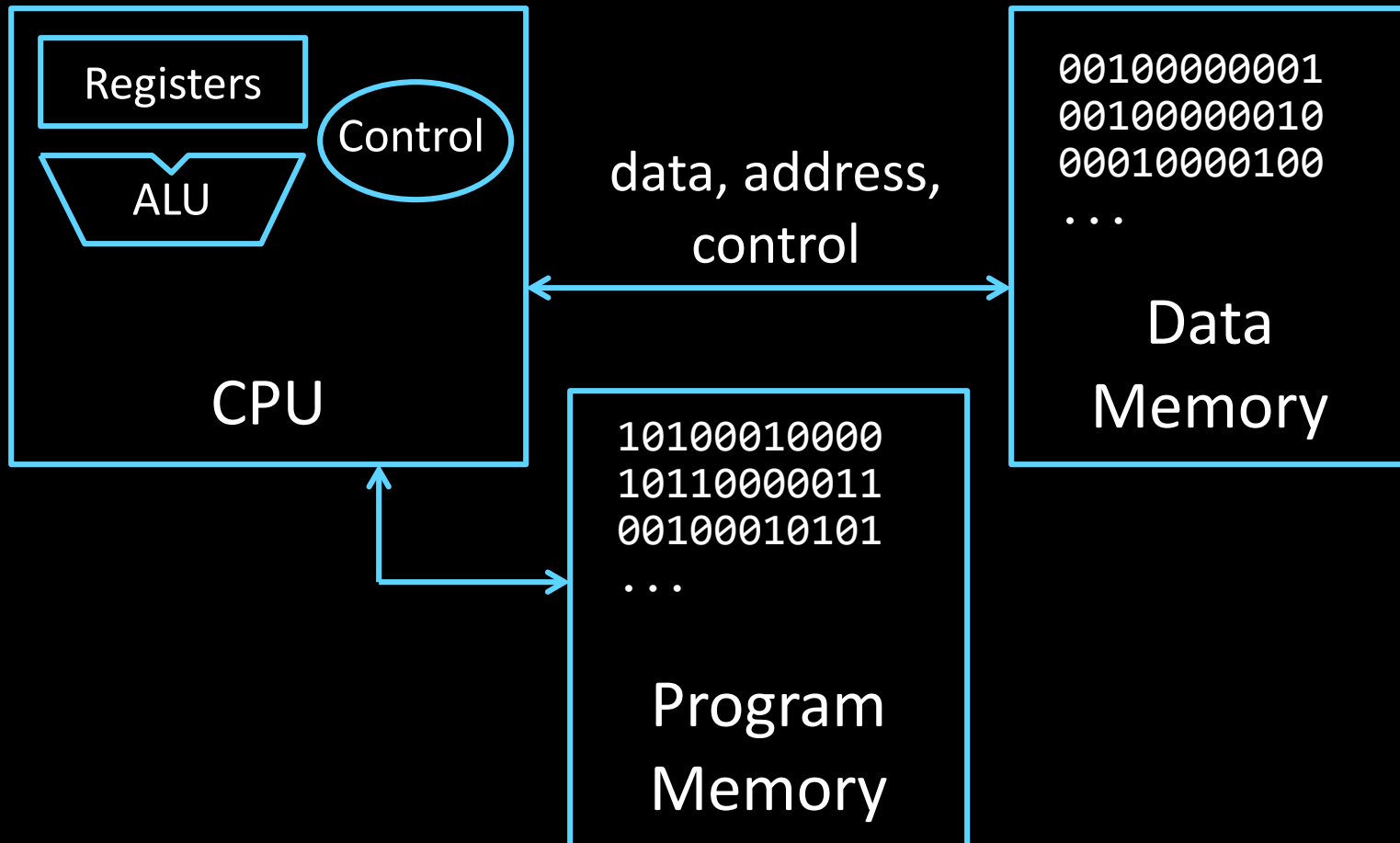
```
.text  
.ent main  
main: la $4, Larray  
li $5, 15  
...  
li $4, 0  
jal exit  
.end main  
.data  
Larray:  
.long 51, 491, 3991
```



# Assembling Programs

Assembly with a but using (modified) Harvard architecture

- Need segments since data and program stored together in memory



# Takeaway

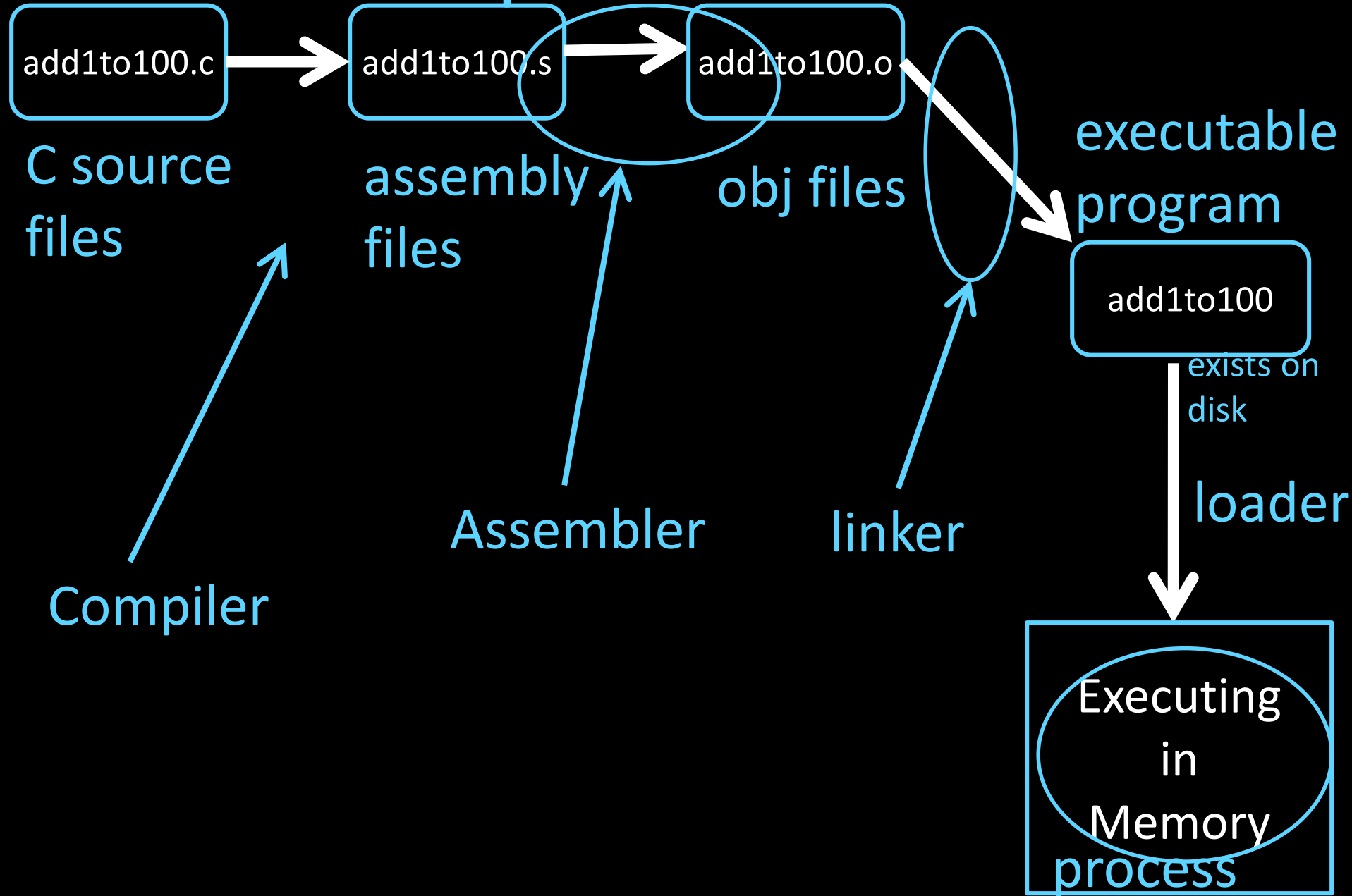
## Assembly is a low-level task

- Need to assemble assembly language into machine code binary. Requires
  - Assembly language instructions
  - *pseudo-instructions*
  - And Specify layout and data using *assembler directives*
- Today, we use a modified Harvard Architecture (Von Neumann architecture) that mixes data and instructions in memory
  - ... but kept in separate *segments*
  - ... and has separate caches

# Next Goal

Put it all together: An example of compiling a program from source to assembly to machine object code.

# Example: Add 1 to 100



# 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++)
```

```
        sum += 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
```

```
# Compile
```

```
[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 + 4);
```

```
    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			
Global			
Dynamic			

**C Pointers can be trouble**

```
int *trouble()
```

```
{ int a; ...; return &a; }
```

```
char *evil()
```

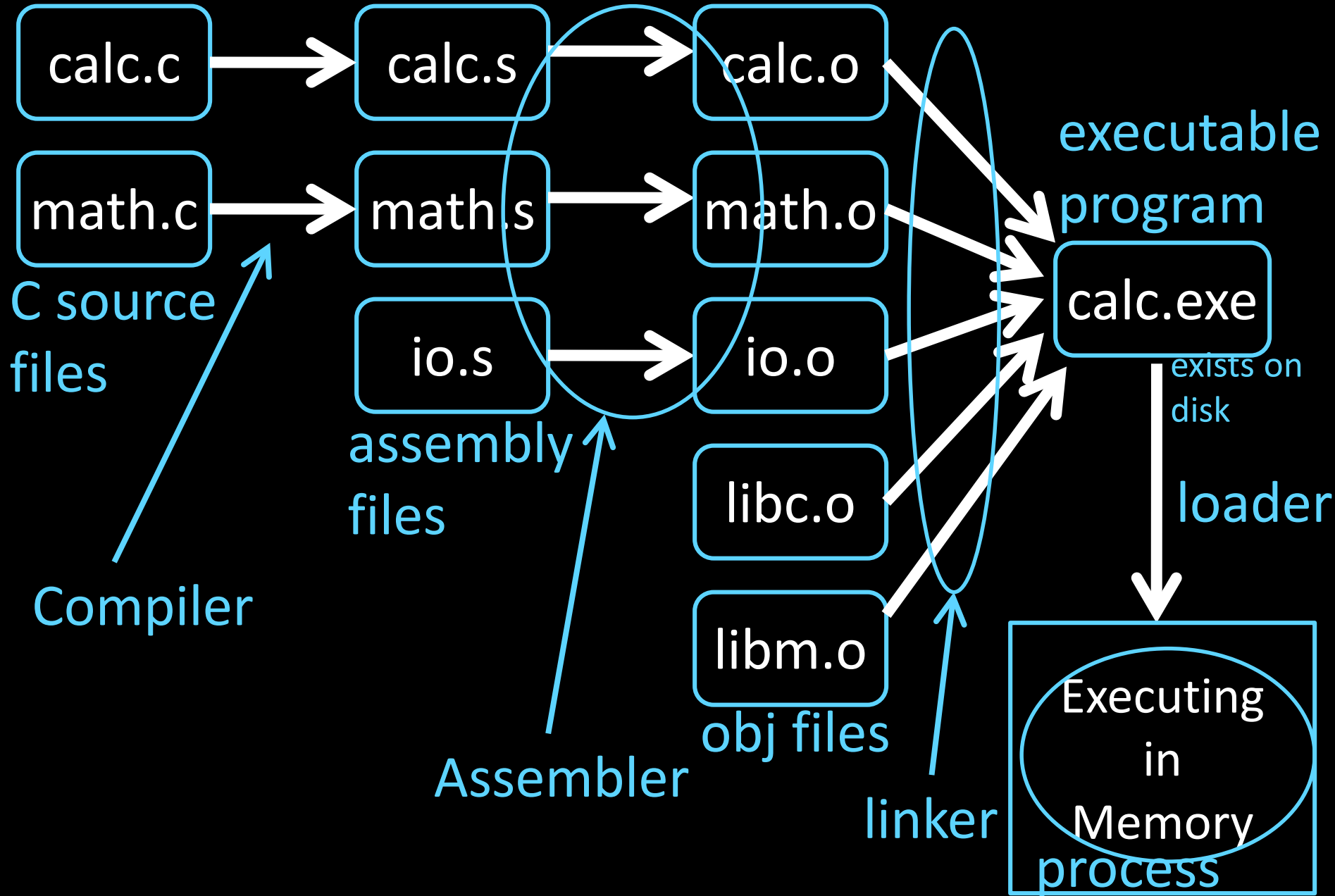
```
{ char s[20]; gets(s); return s; }
```

```
int *bad()
```

```
{ s = malloc(20); ... free(s); ... return s; }
```

(Can't do this in Java, C#, ...)

# Example #2: Review of Program Layout



# Example #2: Review of Program Layout

calc.c

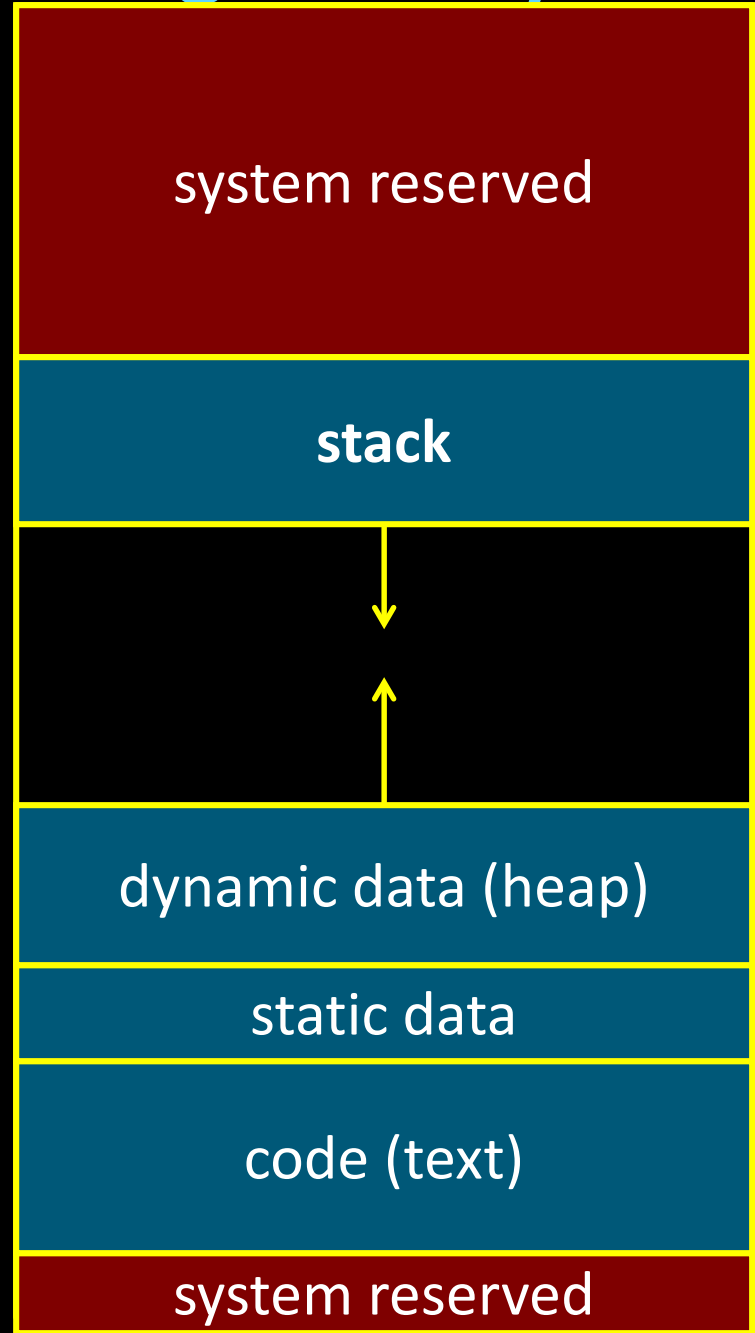
```
vector* v = malloc(8);  
v->x = prompt("enter x");  
v->y = prompt("enter y");  
int c = pi + tnorm(v);  
print("result %d", c);
```

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



# Takeaway

Compiler produces assembly files

- (contain MIPS assembly, pseudo-instructions, directives, etc.)

Assembler produces object files

- (contain MIPS machine code, missing symbols, some layout information, etc.)

Linker produces executable file

- (contains MIPS machine code, no missing symbols, some layout information)

Loader puts program into memory and jumps to first instruction

- (machine code)

# Recap

Compiler output is assembly files

Assembler output is obj files

Next Time

Linker joins object files into one executable

Loader brings it into memory and starts execution