

---

# **Chapter 2.12: Compilation, Assembling, Linking and Program Execution**

**ITSC 3181 Introduction to Computer Architecture**  
<https://passlab.github.io/ITSC3181/>

Department of Computer Science

Yonghong Yan

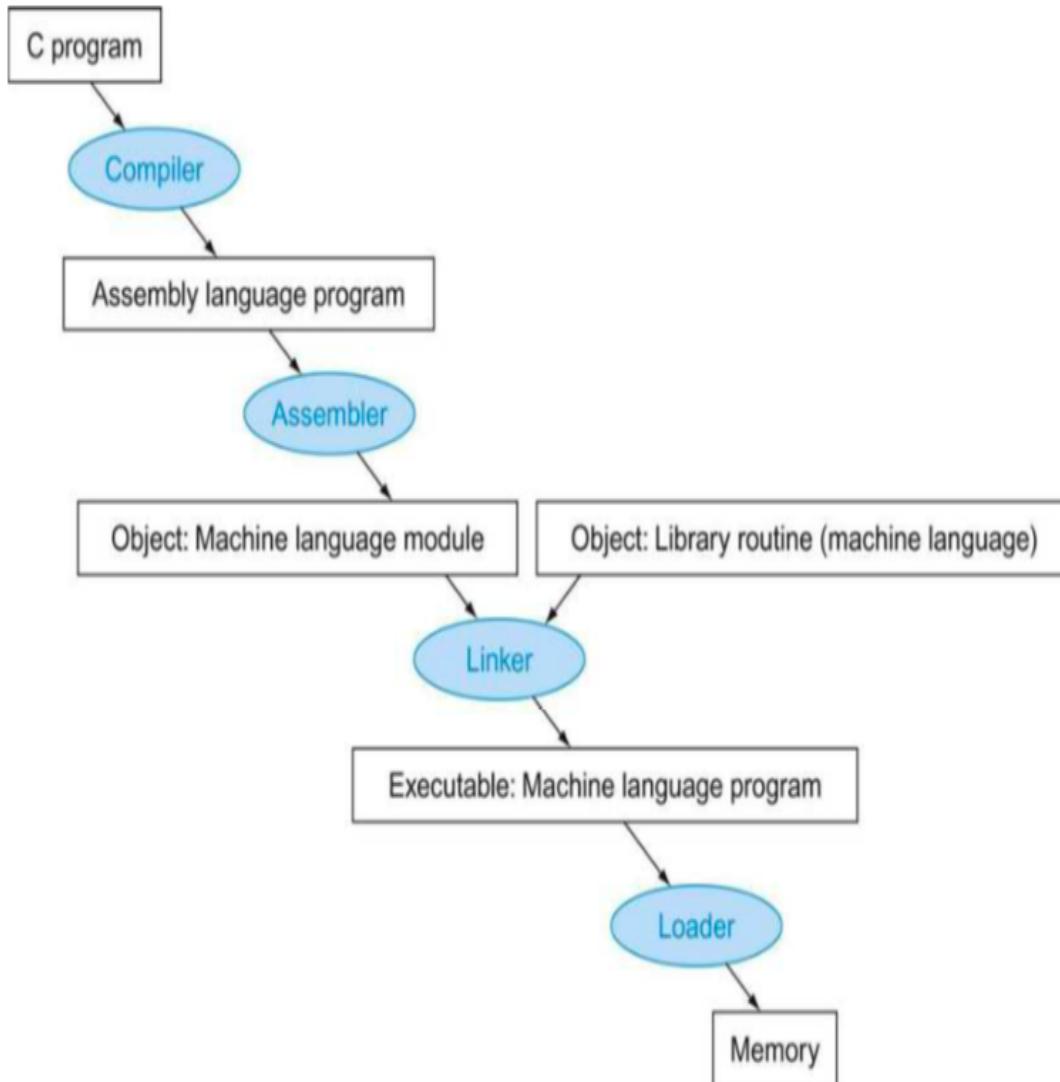
[yyan7@uncc.edu](mailto:yyan7@uncc.edu)

<https://passlab.github.io/yanyh/>

---

# A Translation Hierarchy for C

---



# Compilation Process in C

---

- Compilation process: `gcc hello.c -o hello`
  - Constructing an executable image for an application
  - **Multiple stages**
  - Command:  
`gcc <options> <source_file.c>`
- Compiler Tool
  - `gcc` (GNU Compiler)
    - `man gcc` (on Linux m/c)
  - `icc` (Intel C compiler)

# 4 Stages of Compilation Process

---

## Preprocessing

```
gcc -E hello.c -o hello.i  
hello.c → hello.i
```

## Compilation (after preprocessing)

```
gcc -S hello.i -o hello.s
```

## Assembling (after compilation)

```
gcc -c hello.s -o hello.o
```

## Linking object files

```
gcc hello.o -o hello
```

Output → Executable (a.out)  
Run → ./hello (Loader)

# 4 Stages of Compilation Process

---

## 1. Preprocessing (Those with # ...)

- Expansion of Header files (#include ... )
- Substitute macros and inline functions (#define ...)

## 2. Compilation

- Generates assembly language, .s file
- Verification of functions usage using prototypes
- Header files: Prototypes declaration

## 3. Assembling

- Generates re-locatable object file (contains m/c instructions), .o file
- nm app.o  
0000000000000000 T main  
U puts
- nm or objdump tool used to view object files

# 4 Stages of Compilation Process (contd..)

---

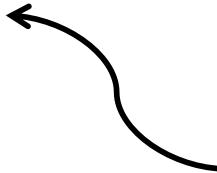
## 4. Linking

- Generates executable file (nm tool used to view exe file)
- Binds appropriate libraries
  - Static Linking
  - Dynamic Linking (default)
- Loading and Execution (of an executable file)
  - Evaluate size of code and data segment
  - Allocates address space in the user mode and transfers them into memory
  - Load dependent libraries needed by program and links them
  - Invokes Process Manager → Program registration

# Compiling a C Program

---

- *gcc <options> program\_name.c*



- Options:

-----  
**-Wall:** Shows all warnings

**-o output\_file\_name:** By default a.out executable file is created when we compile our program with gcc. Instead, we can specify the output file name using "-o" option.

**-g:** Include debugging information in the binary.

**Four stages into one**

- *man gcc*

# Preprocessing

---

- Things with #
  - `#include <stdio.h>`
  - `#define REAL float`
  - Others
- Processes the C source files BEFORE handing it to compiler.
  - `Pre`-process
  - `gcc -E`
  - `cpp`

# File Inclusion

---

- Recall : `#include <filename>`
  - `#include <foo.h>`
    - System directories
  - `#include "foo.h"`
    - Current directories
  - `gcc -I/usr/include` to specify where to search those header files
    - `gcc -I/usr/include sum_full.c -o sum`
- Preprocessing replaces the line “`#include <foo.h>`” with the content of the file `foo.h`

# Macros

---

- Define and replaced by preprocessing
  - Every occurrence of **REAL** will be replaced with **float** before compilation.

```
1 #define REAL float
2
3 REAL sum(int N, REAL X[], REAL a) {
4     int i;
5     REAL result = 0.0;
6     for (i = 0; i < N; ++i)
7         result += a * X[i];
8     return result;
9 }
```

# About printf in C

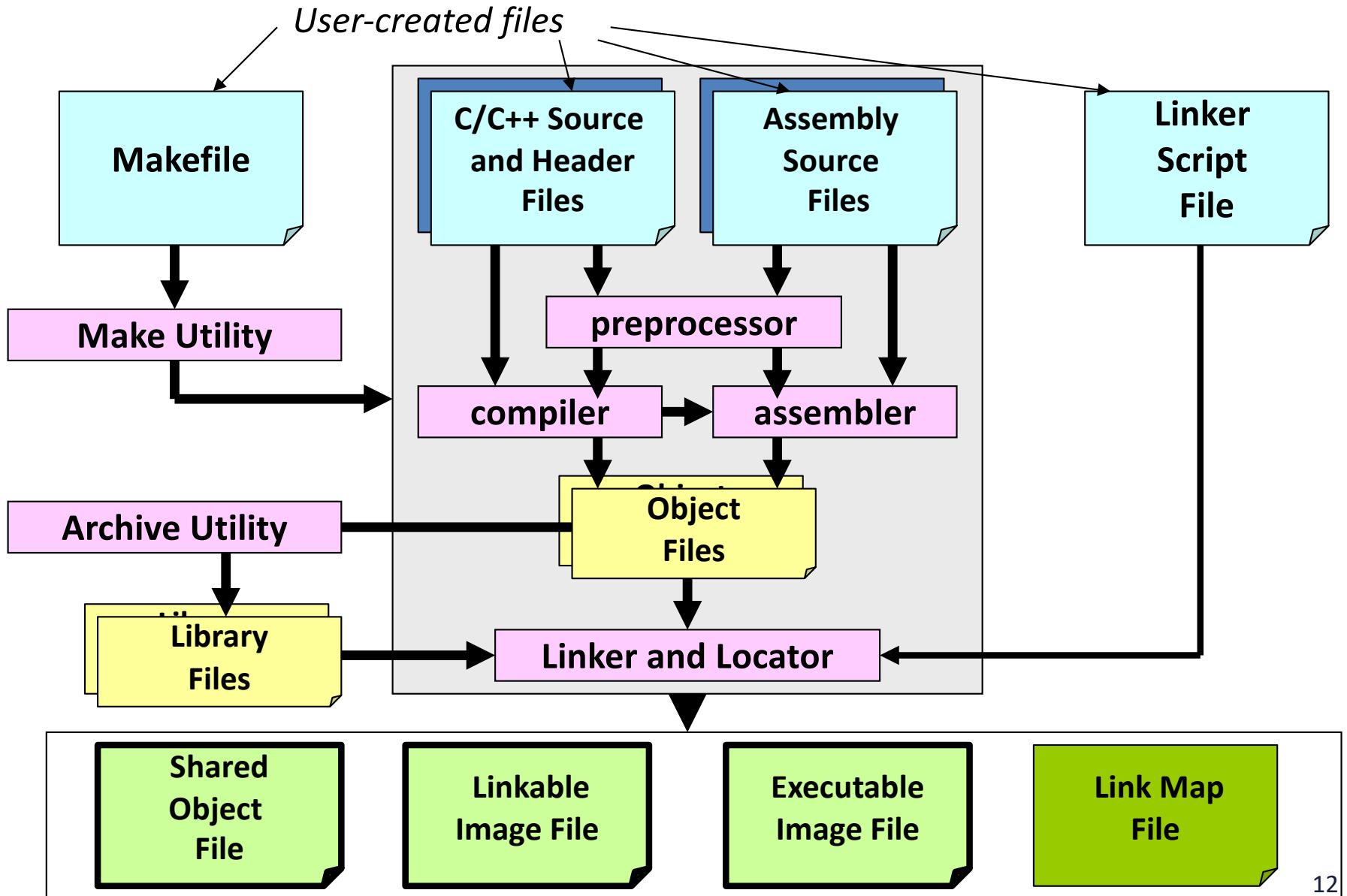
- `printf("format string", vars);`
- Format string?
  - "This year is %d\n"
  - "Your score is %d\n"
- Conversion by %
  - %d : int
  - %f : float, double
  - %c : char
  - %s : char \*, string
  - %e : float, double in scientific form

```
yanyh@vm:~/sum$ ./sum 1000000
=====
Sum 1000000 numbers
Performance: Runtime (ms) MFLOPS
Sum: 3.999949 500.006437
```

The screenshot shows the terminal output of a C program named 'sum' that calculates the sum of 1,000,000 integers. The output is annotated with various colored ovals and dashed arrows to illustrate how the format string is processed:

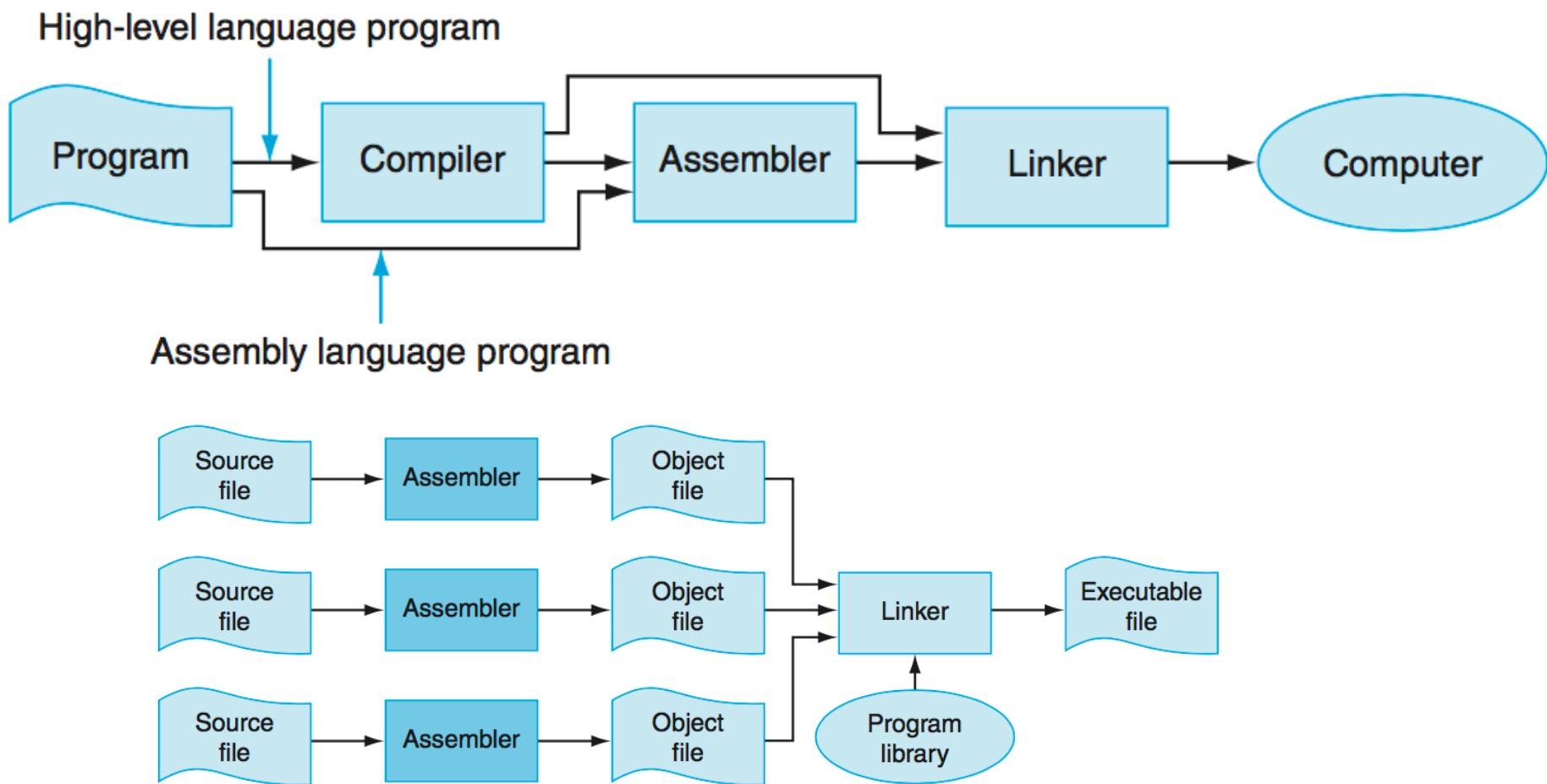
- A yellow oval highlights the placeholder "%d" in the format string, which corresponds to the value "1000000" in the terminal output.
- A yellow oval highlights the placeholder "%d" in the line "Sum %d numbers\n", which corresponds to the value "1000000" in the terminal output.
- A red oval highlights the placeholder "%t" in the line "Performance:\t\tRuntime (ms)\t MFLOPS \n", which corresponds to the values "Runtime (ms)" and "MFLOPS" in the terminal output.
- A green oval highlights the placeholder "%4f" in the line "Sum:\t\t\t%4f\t%4f\n", which corresponds to the value "3.999949" in the terminal output.
- A blue oval highlights the placeholder "%4f" in the line "Sum:\t\t\t%4f\t%4f\n", which corresponds to the value "500.006437" in the terminal output.
- A green oval highlights the placeholder "%t" in the line "Sum:\t\t\t%4f\t%4f\n", which corresponds to the value "Sum:" in the terminal output.
- A green oval highlights the placeholder "%t" in the line "Sum:\t\t\t%4f\t%4f\n", which corresponds to the value "3.999949" in the terminal output.
- A blue oval highlights the placeholder "%t" in the line "Sum:\t\t\t%4f\t%4f\n", which corresponds to the value "500.006437" in the terminal output.

# Tools and Steps for Program Execution



# Code Can be in Assembly Language

- Assembly language either is written by a programmer or is the output of a compiler.



**FIGURE A.1.1 The process that produces an executable file.** An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

# High-Level Program, Assembly Code and Binary

```
1 #include <stdio.h>
2 int main (int argc, char * argv[])
3 {
4     int i;
5     int sum = 0;
6     for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
7     printf ("The sum from 0 .. 100 is %d\n", sum);
8 }
```

```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4, 32($29)
sw       $5, 36($29)
sw       $0, 24($29)
sw       $0, 28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu   $14, $14
addiu   $8, $14, 1
slti    $1, $8, 101
sw $8, 28($29)
mflo    $15
addu    $25, $24, $15
bne    $1, $0, -9
sw       $25, 24($29)
lui    $4, 4096
lw       $5, 24($29)
jal    1048812
addiu   $4, $4, 1072
lw       $31, 20($29)
addiu   $29, $29, 32
jr     $31
move    $2, $0
```

```
.text
.align 2
.globl main
main:
    subu    $sp, $sp, 32
    sw      $ra, 20($sp)
    sd      $a0, 32($sp)
    sw      $0, 24($sp)
    sw      $0, 28($sp)

loop:
    lw      $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw      $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw      $t9, 24($sp)
    addu  $t0, $t6, 1
    sw      $t0, 28($sp)
    ble   $t0, 100, loop
    la      $a0, str
    lw      $a1, 24($sp)
    jal   printf
    move  $v0, $0
    lw      $ra, 20($sp)
    addu  $sp, $sp, 32
    jr     $ra

.str:
    .data
    .align 0
    .asciiz "The sum from 0 .. 100 is %d\n"
```

```
0010011101111011111111111100000
101011110111110000000000000010100
10101111010010000000000000100000
1010111101001010000000000000100100
101011110100000000000000000011000
101011110100000000000000000011100
100011110101110000000000000011100
100011110111000000000000000011000
000000011100111000000000000011001
001001011100100000000000000000001
001010010000000100000000001100101
101011110101000000000000000011100
00000000000000000000111100000010010
00000011000011111100100000100001
000101000010000011111111110111
101011110111001000000000000011000
00111100000001000001000000000000
100011110100101000000000000011000
0000110000010000000000000000110100
0010010010000100000000000000110000
100011110111110000000000000010100
0010011101111010000000000000100000
000000111100000000000000000000001000
00000000000000000000000000000000100001
```

**FIGURE A.1.2 MIPS machine language code for the squares of integers between 0 and 100.**

# Hand-On, sum x86\_64

[https://passlab.github.io/ITSC3181/exercises/sum/sum\\_full.c](https://passlab.github.io/ITSC3181/exercises/sum/sum_full.c)

[https://passlab.github.io/ITSC3181/exercises/sum/sum\\_full\\_x86.s](https://passlab.github.io/ITSC3181/exercises/sum/sum_full_x86.s)

```
35 REAL sum(int N, REAL X[], REAL a) {  
36     int i;  
37     REAL result = 0.0;  
38     for (i = 0; i < N; ++i)  
39         result += a * X[i];  
40     return result;  
41 }
```

- A method in assembly

- **.globl: a global symbol**
- **.type**
- **.cfi\_startproc**
- **.cfi\_endproc**
- **ret: return**

- for loop

- check  $i < N$ , if true continue, else  
 goto end;  
 loop body  
  $i++$   
 end

```
.LFE4:  
.size  init, .-init  
.globl  sum  
.type   sum, @function  
  
sum:  
.LFB5:  
.cfi_startproc  
pushq  %rbp  
.cfi_def_cfa_offset 16  
.cfi_offset 6, -16  
movq  %rsp, %rbp  

```

# Sum, RISC-V and MIPS

- Mainly different instructions

```
REAL sum(int N, REAL X[], REAL a)
    int i;
    REAL result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}
```

- for loop

+ check  $i < N$ , if true,  
continue, else goto end;  
— loop body  
—  $i++$   
— end

|       | RISC-V Version  | MIPS Version   |
|-------|---|--|
| sum:  | .globl sum<br>.type sum, @function  | sw \$6,32(\$fp)<br>sw \$fp,12(\$fp)  |
|       | addi sp,sp,-48<br>sd s0,40(sp)<br>addi s0,sp,48<br>mv a5,a0<br>sd a1,-48(s0)<br>fsw fa0,-40(s0)<br>sw a5,-36(s0)<br>sw zero,-24(s0)<br>sw zero,-20(s0)  | sw \$0,8(\$fp)<br>nop  |
| .L10: | j .L9   | \$L10:   |
|       | lw a5,-20(s0)<br>slli a5,a5,2<br>ld a4,-48(s0)<br>add a5,a4,a5<br>flw fa4,0(a5)<br>flw fa5,-40(s0)<br>fmul.s fa5,fa4,fa5<br>flw fa4,-24(s0)<br>fadd.s fa5,fa4,fa5<br>fsw fa5,-24(s0)<br>lw a5,-20(s0)<br>addiw a5,a5,1<br>sw a5,-20(s0) | lw \$2,8(\$fp)<br>nop<br>sll \$2,\$2,2<br>lw \$3,28(\$fp)<br>nop<br>addu \$2,\$3,\$2<br>lwcl \$f2,0(\$2)<br>lwcl \$f0,32(\$fp)<br>nop<br>mul.s \$f0,\$f2,\$f0<br>lwcl \$f2,12(\$fp)<br>nop<br>add.s \$f0,\$f2,\$f0<br>swcl \$f0,12(\$fp)<br>lw \$2,8(\$fp)<br>nop<br>addiu \$2,\$2,1<br>sw \$2,8(\$fp) |
| .L9:  | lw a4,-20(s0)<br>lw a5,-36(s0)<br>sext.w a4,a4<br>sext.w a5,a5<br>blt a4,a5,.L10<br>flw fa5,-24(s0)<br>fmv.s fa0,fa5<br>ld s0,40(sp)<br>addi sp,sp,48<br>jr ra  | \$L9:<br>lw \$3,8(\$fp)<br>lw \$2,24(\$fp)<br>nop<br>slt \$2,\$3,\$2<br>bne \$2,\$0,\$L10<br>nop<br>lwcl \$f0,12(\$fp)<br>move \$sp,\$fp<br>lw \$fp,20(\$sp)<br>addiu \$sp,\$sp,24<br>j \$31<br>nop  |
|       | .size sum, .-sum<br>.section .rodata<br>.align 3  |  |

# Sum, x86\_64

- Number of instructions per loop iteration
  - Count it

```
35 REAL sum(int N, REAL X[], REAL a) {  
36     int i;  
37     REAL result = 0.0;  
38     for (i = 0; i < N; ++i)  
39         result += a * X[i];  
40     return result;  
41 }
```

$$\text{CPU Time}(s) = \frac{\# \text{ Instructions}}{\text{Program}} \times \frac{\# \text{ Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

```
.LFE4:  
.size    init, .-init  
.globl   sum  
.type    sum, @function  
  
sum:  
.LFB5:  
.cfi_startproc  
pushq   %rbp  
.cfi_def_cfa_offset 16  
.cfi_offset 6, -16  
movq   %rsp, %rbp  
.cfi_def_cfa_register 6  
movl   %edi, -20(%rbp)  
movq   %rsi, -32(%rbp)  
movss  %xmm0, -24(%rbp)  
pxor   %xmm0, %xmm0  
movss  %xmm0, -4(%rbp)  
movl   $0, -8(%rbp)  
jmp    .L11  
  
.L12:  
movl   -8(%rbp), %eax  
cltq  
leaq   0(%rax,4), %rdx  
movq   -32(%rbp), %rax  
addq   %rdx, %rax  
movss  (%rax), %xmm0  
mulss  -24(%rbp), %xmm0  
movss  -4(%rbp), %xmm1  
addss  %xmm1, %xmm0  
movss  %xmm0, -4(%rbp)  
addl   $1, -8(%rbp)  
  
.L11:  
movl   -8(%rbp), %eax  
cmpq   -20(%rbp), %eax  
jl    .L12  
movss  -4(%rbp), %xmm0  
popq   %rbp  
.cfi_def_cfa 7, 8  
ret  
.cfi_endproc  
  
.LFE5:  
.size    sum, .-sum  
.section .rodata  
  
.LC2:  
.string "Usage: sum <n> (default %d)\n"  
.align 8
```

# When to Use Assembly Language

---

- Advantage: Speed, size and predictable
  - No compiler middle-man
  - Fit for mission-critical, embedded domain, e.g. space shuttle or car control
- Hybrid approach
  - Non-critical part in high-level language
  - Critical part in assembly language
- Explore special instructions
  - E.g. those special-purpose instructions that can do more than one thing

# Drawbacks of Assembly Language

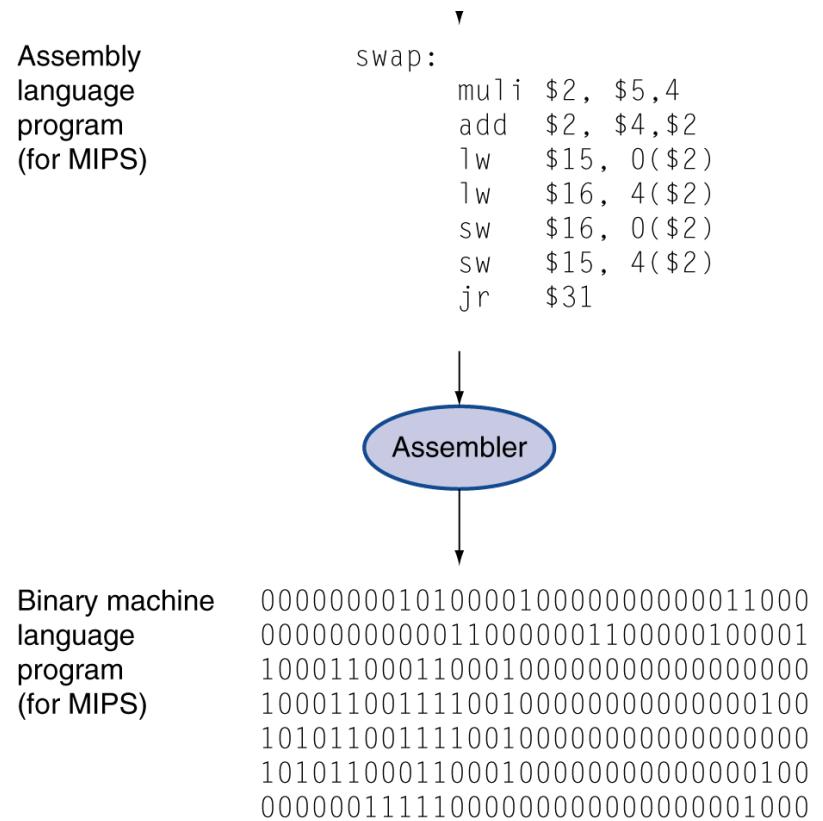
---

Assembly language has many (and more) disadvantages that strongly argue against its wide-spread use.

- Machine-specific code, i.e. assembly code are not portable
  - Rewrite for new or different architectures
- Harder than high level language to write large code or software
  - Harder to keep a high-level software structure
  - Harder to read and debug
- Most compilers are good enough to convince that you do not need to write assembly code for general-purpose applications
  - Except embedded or IoT domain

# Assembler

- Translates file of assembly language statements into a file of binary machine instructions and binary data.
  - Two main steps:
    - Find memory address for symbols (e.g. functions).
    - Translate each assembly statement by combining the numeric equivalents of opcodes, register specifiers, and labels into a legal instruction
      - Binary
  - Produce object files



# Object File

ELF Format: [https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

|                    |              |              |                        |              |                       |
|--------------------|--------------|--------------|------------------------|--------------|-----------------------|
| Object file header | Text segment | Data segment | Relocation information | Symbol table | Debugging information |
|--------------------|--------------|--------------|------------------------|--------------|-----------------------|

**FIGURE A.2.1 Object file.** A UNIX assembler produces an object file with six distinct sections.

```
#include <stdio.h>

int a[10]={0,1,2,3,4,5,6,7,8,9};
int b[10];

int main(int argc, char* argv[]) {
    int i;
    static int k = 3;

    for(i = 0; i < 10; i++) {
        printf("%d\n",a[i]);
        b[i] = k*a[i];
    }
}
```

# Contents of Object File for the Sample C program

---

| Offset   | Contents | Comment  |
|--|----------|--|
| <b>Header section</b>                            |          |  |
| 0  | 124      | number of bytes of Machine code section  |
| 4  | 44       | number of bytes of initialized data section  |
| 8  | 40       | number of bytes of Uninitialized data section (array <code>b[]</code> )<br><i>(not part of this object module)</i> |
| 12   | 60       | number of bytes of Symbol table section  |
| 16   | 44       | number of bytes of Relocation information section  |
| <b>Machine code section</b> (124 bytes)          |          |  |
| 20   | X        | code for the top of the <code>for</code> loop (36 bytes)   |
| 56   | X        | code for call to <code>printf()</code> (22 bytes)  |
| 68   | X        | code for the assignment statement (10 bytes)   |
| 88   | X        | code for the bottom of the <code>for</code> loop (4 bytes)   |
| 92   | X        | code for exiting <code>main()</code> (52 bytes)  |
| <b>Initialized data section</b> (44 bytes)       |          |  |
| 144  | 0        | beginning of array <code>a[]</code>  |
| 148  | 1        |  |
| :  |          |  |
| 176  | 8        |  |
| 180  | 9        | end of array <code>a[]</code> (40 bytes)   |
| 184  | 3        | variable <code>k</code> (4 bytes)  |
| <b>Symbol table section</b> (60 bytes)           |          |  |
| 188  | X        | array <code>a[]</code> : offset 0 in Initialized data section (12 bytes)   |
| 200  | X        | variable <code>k</code> : offset 40 in Initialized data section (10 bytes)   |
| 210  | X        | array <code>b[]</code> : offset 0 in Uninitialized data section (12 bytes)   |
| 222  | X        | <code>main</code> : offset 0 in Machine code section (12 bytes)  |
| 234  | X        | <code>printf</code> : external, used at offset 56 of Machine code section (14 bytes)                               |
| <b>Relocation information section</b> (44 bytes) |          |  |
| 248  | X        | relocation information   |

# Some Terms

---

- Object file vs Executable
  - Object file is the file for binary format of machine instructions, not linked with others, nor positioned (in memory) for execution
  - Executable is binary format of object files that are linked and positioned ready for execution.
- Symbol
  - Names, e.g. global function name, variable name
- Library
  - Archive or package of multiple object files

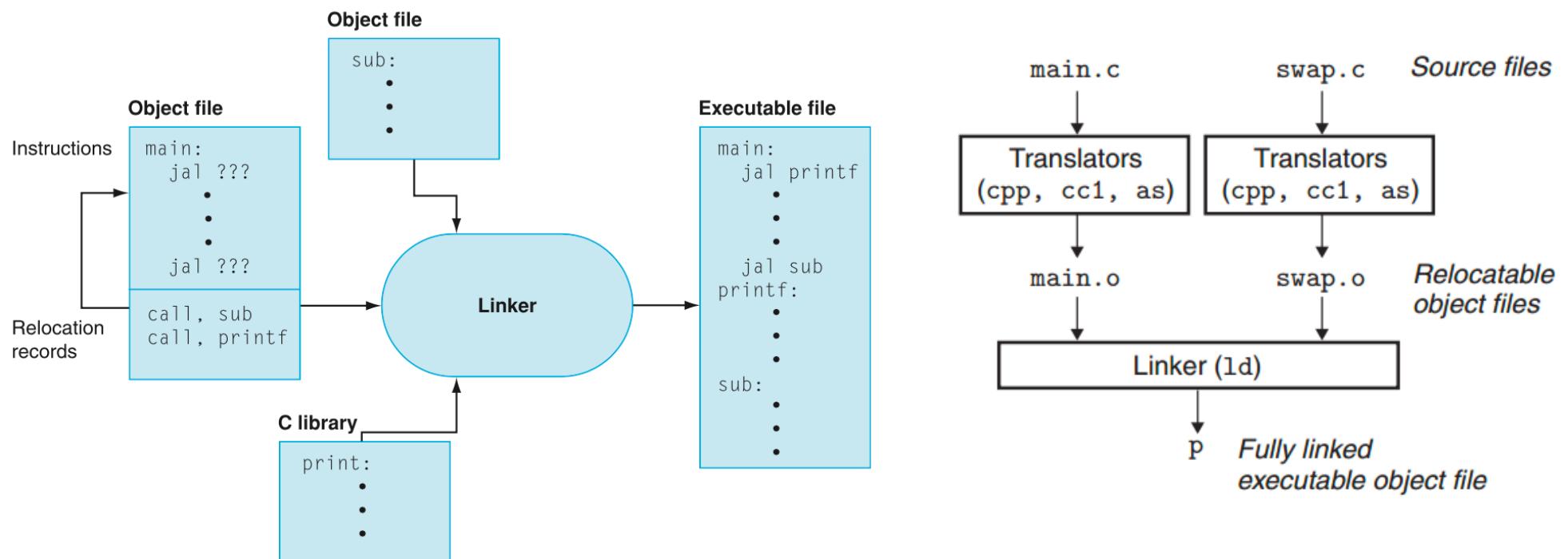
# Inspect an ELF Object File or Executable

---

- Executable and Linkable Format (ELF)
  - [https://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)
- readelf and objdump command in Linux to inspect object/executable file or disassembly
  - Only objdump can do disassembly
- nm command to display symbol information
- Try sum\_full.o and sum example
  - sum\_full.o is an object file
  - sum is an executable

# Linking

- Linker (`ld` command) searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.



# Linking Multiple files to make executable file

---

- Two programs, prog1.c and prog2.c for one single task
  - To make single executable file using following instructions

First, compile these two files with option "-c"

```
gcc -c prog1.c  
gcc -c prog2.c
```

-c: Tells gcc to compile and assemble the code, but not link.

We get two files as output, prog1.o and prog2.o

Then, we can link these object files into single executable file using below instruction.

```
gcc -o prog prog1.o prog2.o
```

Now, the output is prog executable file.

We can run our program using  
**./prog**

# Linking with other libraries

---

- Normally, compiler will read/link libraries from /usr/lib directory to our program during compilation process.
  - Library are precompiled object files
- To link our programs with libraries like pthreads and realtime libraries (rt library).
  - `gcc <options> program_name.c -Ipthread -Irt`
    - Ipthread: Link with pthread library → **libpthread.so** file
    - Irt: Link with rt library → **librt.so** file
    - Option here is "**-I<library>**"

Another option "**-L<dir>**" used to tell gcc compiler search for library file in given **<dir>** directory.

# Compile Multiple Files and Link to One Executable

---

- Split the sum\_full.c into two files
  - sum.c that only contains the definition of sum method
    - Also the “#define REAL float” line on top
  - Remove the sum definition from sum\_full.c, but still keep sum method declaration (referred to as function signature)
  - Compile both together and generate sum executable
- **Compile in one step:** `gcc sum_full.c sum.c -o sum`
  - The command compiles each \*.c file one by one into object files and then link the two object files into one executable
- **Compile in multiple steps:** compile each .c file one by one and link together

# Compile in One Step

---

```
yanyh@vm:~/sum$ ls  
sum.c sum_full.c  
yanyh@vm:~/sum$ gcc sum.c sum_full.c -o sum  
yanyh@vm:~/sum$ ls  
sum sum.c sum_full.c  
yanyh@vm:~/sum$ ./sum 1000000
```

=====  
Sum 1000000 numbers

---

| Performance: | Runtime (ms) | MFLOPS     |
|--------------|--------------|------------|
| Sum:         | 3.999949     | 500.006437 |

# Compile in Multiple Steps

```
yanyh@vm:~/sum$ gcc -c sum.c
yanyh@vm:~/sum$ ls
sum  sum.c  sum_full.c  sum.o
yanyh@vm:~/sum$ gcc -c sum_full.c
yanyh@vm:~/sum$ ls
sum  sum.c  sum_full.c  sum_full.o  sum.o
yanyh@vm:~/sum$ gcc sum.o sum_full.o -o sum
yanyh@vm:~/sum$ ls
sum  sum.c  sum_full.c  sum_full.o  sum.o
yanyh@vm:~/sum$ ./sum 1000000
```

Sum 1000000 numbers

| Performance: | Runtime (ms) | MFL0PS |
|--------------|--------------|--------|
|--------------|--------------|--------|

|      |          |            |
|------|----------|------------|
| Sum: | 9.999990 | 200.000191 |
|------|----------|------------|

# Try readelf

---

```
yanyh@vm:~/sum$ readelf -a sum
```

ELF Header:

|              |   |
|--------------|---|
| Magic:       | 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |
| Class:       | ELF64   |
| Data:        | 2's complement, little endian                   |
| Version:     | 1 (current)                                     |
| OS/ABI:      | UNIX – System V                                 |
| ABI Version: | 0   |
| Type:        | EXEC (Executable file)                          |

```
yanyh@vm:~/sum$ readelf -a sum.o
```

ELF Header:

|              |   |
|--------------|---|
| Magic:       | 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |
| Class:       | ELF64   |
| Data:        | 2's complement, little endian                   |
| Version:     | 1 (current)                                     |
| OS/ABI:      | UNIX – System V                                 |
| ABI Version: | 0   |
| Type:        | REL (Relocatable file)                          |
| Machine:     | Advanced Micro Devices X86-64                   |
| Version:     | 0x1   |

# Try objdump for both object file and executable

---

```
yanyh@vm:~/sum$ objdump -x sum.o
```

```
sum.o:      file format elf64-x86-64
sum.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
```

## Sections:

| Idx | Name  | Size     | VMA                                   | LMA              |
|-----|-------|----------|---------------------------------------|------------------|
| 0   | .text | 00000060 | 0000000000000000                      | 0000000000000000 |
|     |       |          | CONTENTS, ALLOC, LOAD, READONLY, CODE |                  |
| 1   | .data | 00000000 | 0000000000000000                      | 0000000000000000 |
|     |       |          | CONTENTS, ALLOC, LOAD, DATA           |                  |
| 2   | .bss  | 00000000 | 0000000000000000                      | 0000000000000000 |

# “objdump -D” to disassembly: convert binary object code back to symbolic assembly code

---

```
yanyh@vm:~/sum$ objdump -D sum  
sum:      file format elf64-x86-64
```

Disassembly of section .interp:

|                                 |                                      |
|---------------------------------|--------------------------------------|
| 000000000400238 <.interp>:      |                                      |
| 400238:    2f                   | (bad)                                |
| 400239:    6c                   | insb    (%dx),%es:(%rdi)             |
| 40023a:    69 62 36 34 2f 6c 64 | imul    \$0x646c2f34,0x36(%rdx),%esp |
| 400241:    2d 6c 69 6e 75       | sub     \$0x756e696c,%eax            |
| 400246:    78 2d                | js      400275 <_init-0x37b>         |
| 400248:    78 38                | js      400282 <_init-0x36e>         |
| 40024a:    36 2d 36 34 2e 73    | ss sub   \$0x732e3436,%eax           |
| 400250:    6f                   | outsl   %ds:(%rsi),(%dx)             |
| 400251:    2e 32 00             | xor     %cs:(%rax),%al               |

# nm: list symbols from object files

- T: define a symbol
- U: undefined symbol
  - Linker to link
- Address are relative

```
yanyh@vm:~/sum$ nm sum.o
0000000000000000T sum
yanyh@vm:~/sum$ nm sum_full.o
U atoi
U drand48
U fprintf
U ftime
000000000000ca T init
000000000000119 T main
U malloc
U printf
U puts
0000000000000000 T read_timer
0000000000000065 T read_timer_ms
U srand48
U __stack_chk_fail
U stderr
U sum

yanyh@vm:~/sum$ nm sum
U atoi@@GLIBC_2.2.5
000000000602078 B __bss_start
000000000602088 b completed.7588
000000000602068 D __data_start
000000000602068 W data_start
000000000400700 t deregister_tm_clones
000000000400780 t __do_global_dtors_aux
000000000601e18 t __do_global_dtors_aux
U drand48@@GLIBC_2.2.5
```

# Static Linking

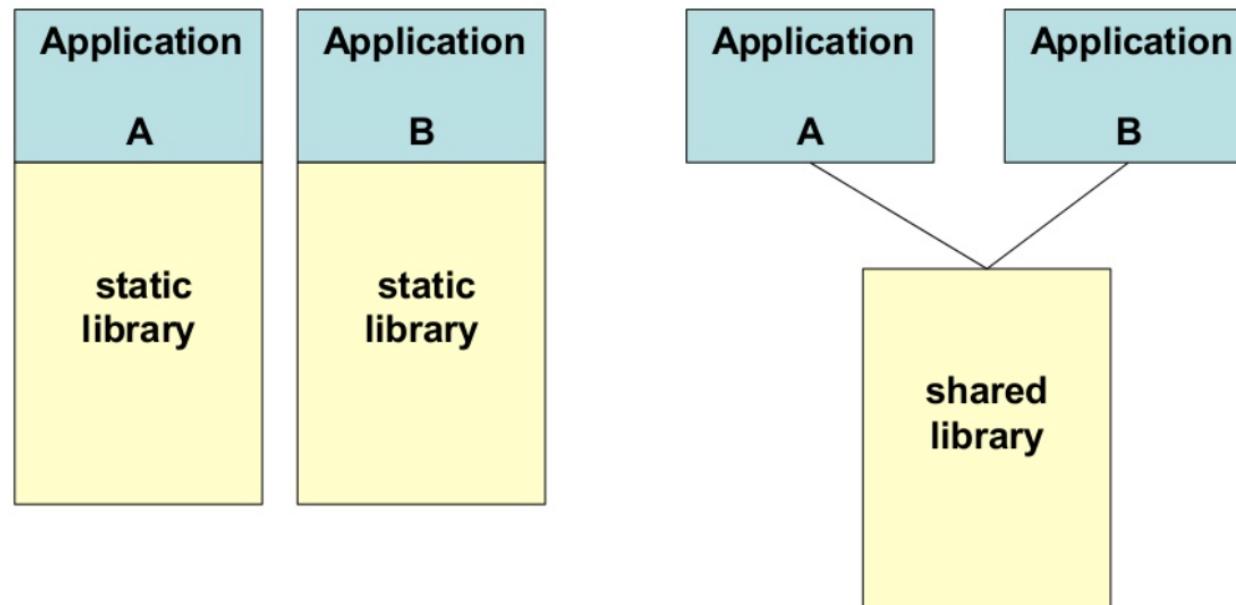
---

- If multiple programs want to use `read_timer` functions
  - They all include the full definition in their source code
    - Duplicate: If the function changes, we need to change each file
  - Separate `reader_timer` in a new file, compile and statically linked with other object files to create executables
    - Duplicate the same object in multiple executables.
- Dynamic linking at the runtime
  - Create a dynamic library that provides `reader_timer` implementation
  - Tell ld to link the library at the runtime
  - Runtime load and link them on the fly and execute

# Static Library vs Shared (Dynamic) Library

---

- Static library needs to be duplicated in every executable
  - Bigger code size, better optimized
- Shared library are loaded on the fly during the execution
  - Smaller code size, performance hits of loading shared memory



- Combine both

# Hands-On for dynamic linking

---

- Sum example for static and dynamic linking: from sum.c and sum\_full.c created in the last exercise,
  - Create a new file read\_timer.c that includes the read\_timer and read\_timer\_ms definition in the file
  - Leave only the read\_timer and read\_timer\_ms declaration in the sum\_full.c
    - They are the interface of the two methods.
  - Compile read\_timer.c into a dynamic library
    - The library name is my\_read\_timer, and the library file is libmy\_read\_timer.so. You can choose any name.
  - Compile sum.c and sum\_full.c and link with lib my\_read\_timer
    - `gcc sum_full.c sum.c -o sum -L. -lmy_read_timer`
  - Use ldd command to list dependent libraries

# Build Steps with Dynamic Library

```
yanyh@vm:~/sum$ ls
```

```
read_timer.c sum.c sum_full.c
```

```
yanyh@vm:~/sum$ gcc -shared -fPIC -o libmy_read_timer.so read_timer.c
```

```
yanyh@vm:~/sum$ ls
```

```
libmy_read_timer.so read_timer.c sum.c sum_full.c
```

```
yanyh@vm:~/sum$ gcc sum.c sum_full.c -o sum
```

```
/tmp/ccqjRuag.o: In function `main':
```

```
sum_full.c:(.text+0x108): undefined reference to `read_timer'
```

```
sum_full.c:(.text+0x13e): undefined reference to `read_timer'
```

```
collect2: error: ld returned 1 exit status
```

Linking error: cannot find  
reader\_timer  
implementation when linking  
from sum\_full.o

```
yanyh@vm:~/sum$
```

```
yanyh@vm:~/sum$ gcc sum.c sum_full.c -o sum -lmy_read_timer
```

```
/usr/bin/ld: cannot find -lmy_read_timer
```

```
collect2: error: ld returned 1 exit status
```

Linking error: do not know where  
to find the libread\_timer.so file.

```
yanyh@vm:~/sum$
```

```
yanyh@vm:~/sum$ gcc sum.c sum_full.c -o sum -L. -lmy_read_timer
```

```
yanyh@vm:~/sum$ ls
```

```
libmy_read_timer.so read_timer.c sum sum.c sum_full.c
```

```
yanyh@vm:~/sum$ ./sum 1000000
```

-L<...>: to tell where to find the library  
file, in this case, the current folder (.)  
-l<...>: to tell the library file name, which  
will be expanded to lib<...>.so file

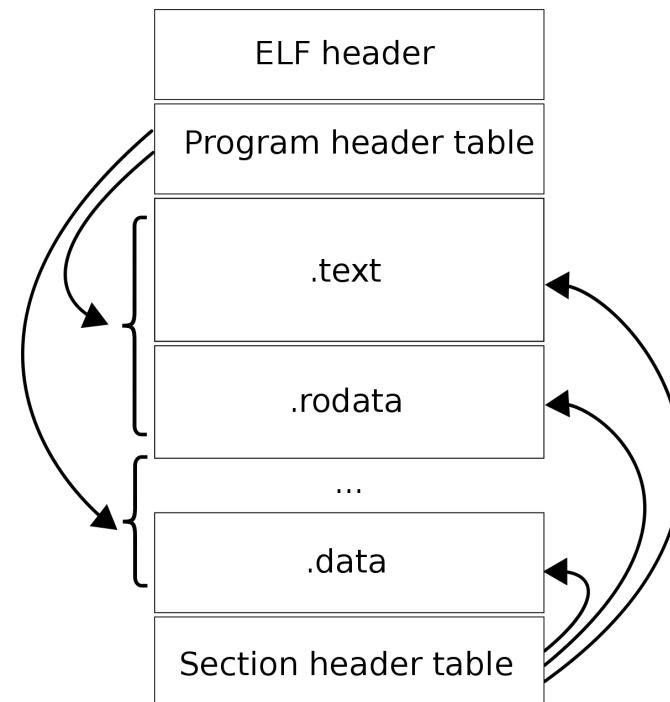
# ldd command to list the dependent libraries

---

```
yanyh@vm:~/sum$ ldd sum
 linux-vdso.so.1 => (0x00007fff8b382000)
 libmy_read_timer.so (0x00007f437c0ae000)
 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f437bce4000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f437c2b0000)
```

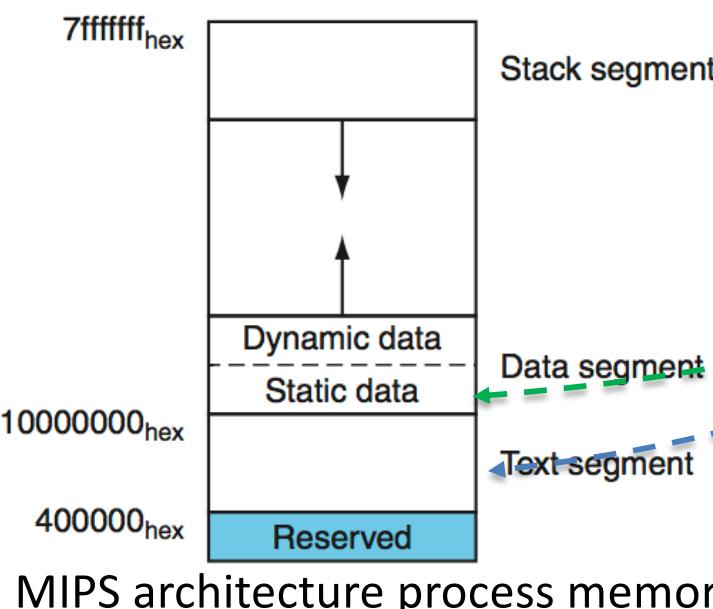
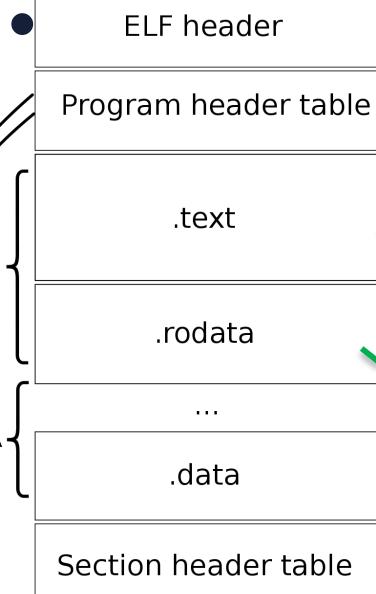
# Loading a File for Execution

- Steps:
  - It reads the executable's header to determine the size of the text and data segments.
  - It creates a new address space for the program. This address space is large enough to hold the text and data segments, along with a stack segment (see Section A.5).
  - It copies instructions and data from the executable into the new address space.
  - It copies arguments passed to the program onto the stack.
  - It initializes the machine registers. In general, most registers are cleared, but the stack pointer must be assigned the address of the first free stack location (see Section A.5).
  - It jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's **main** routine. If the **main** routine returns, the start-up routine terminates the program with the exit system call.

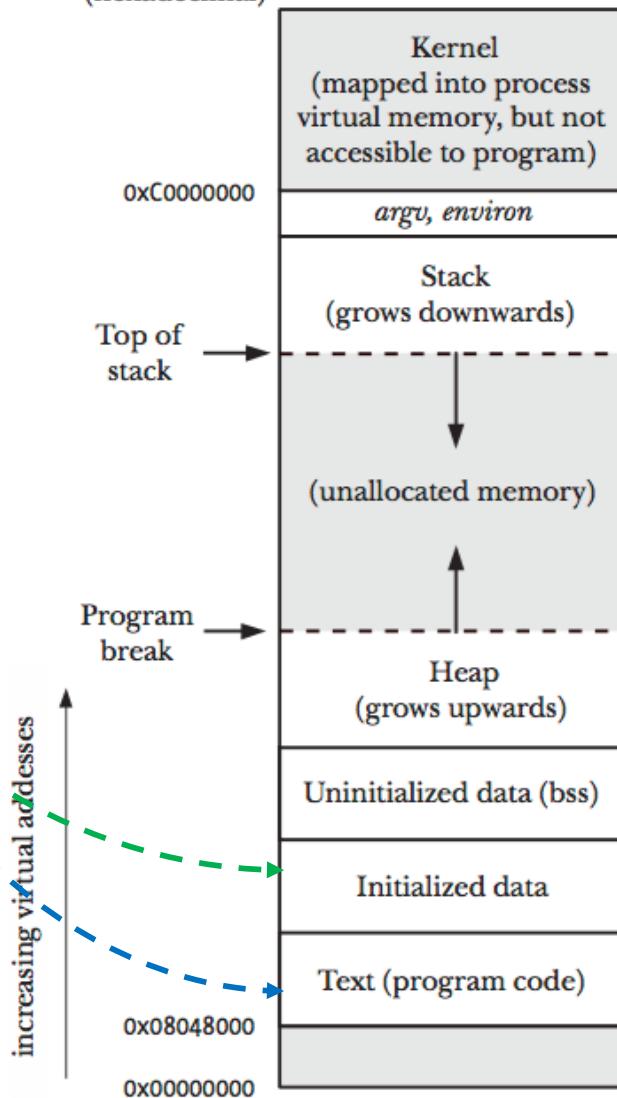


# Memory Layout of A Process

## ELF format of an executable



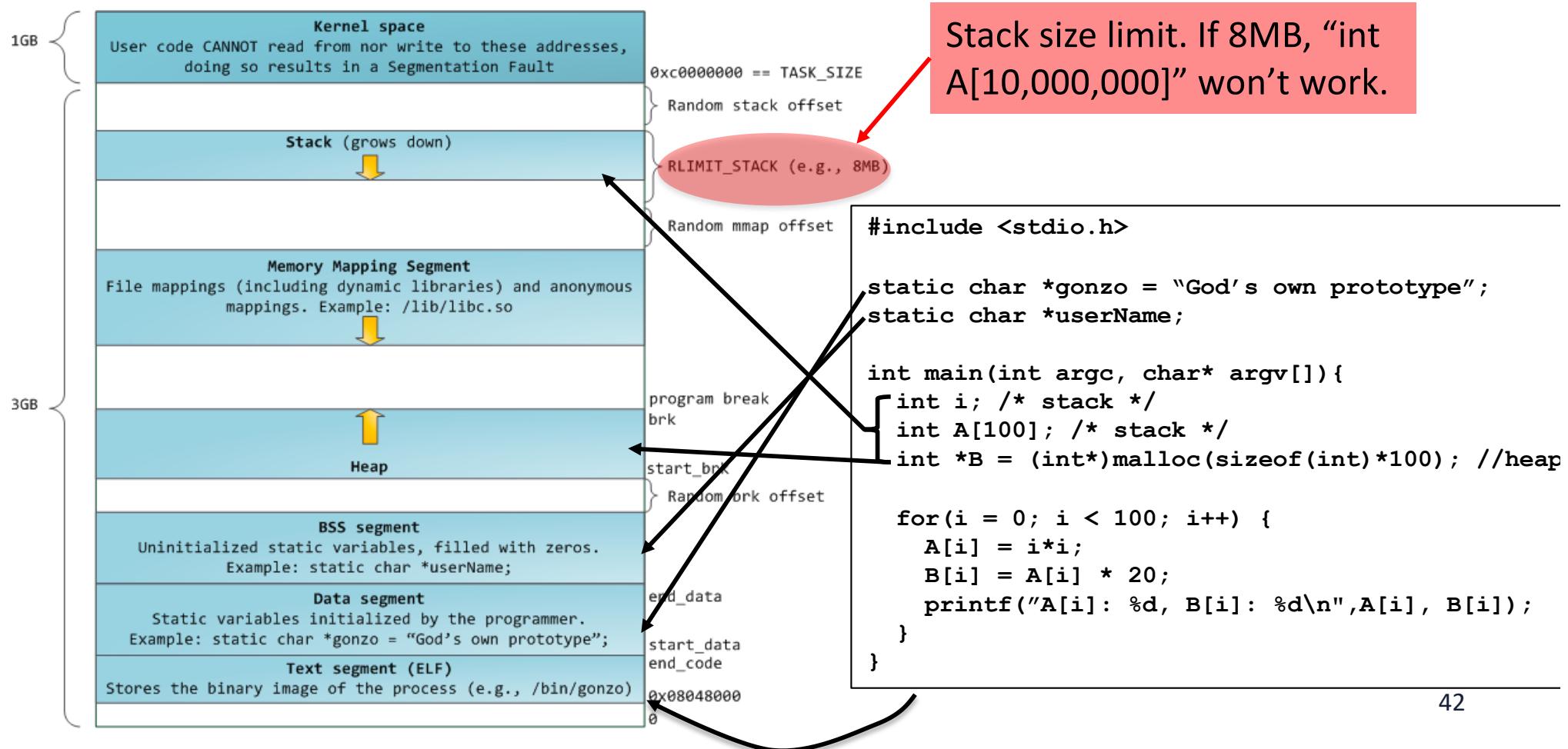
Virtual memory address  
(hexadecimal)



`/proc/kallsyms`  
provides addresses of  
kernel symbols in this  
region (`/proc/ksyms` in  
kernel 2.4 and earlier)

# Linux Process Memory in 32-bit System (4G space)

- Code (machine instructions) → Text segment
- Static variables → Data or BSS segment
- Function variables → stack (i, A[100] and B)
  - A is a variable that stores memory address, the memory for A's 100 int elements is in the stack
  - B is a memory address, it is stored in stack, but the memory B points to is in heap (100 int elements)
- Dynamic allocated memory using malloc or C++ “new” → heap (B[100])



# Check the Memory Map of a Process

- Given a process ID:
  - pmap <pid>
  - cat /proc/<pid>/maps

```
yanyh@vm:~$ pmap 7153
7153: -bash
0000000000400000      976K r-x-- bash
00000000006f3000      4K r----- bash
00000000006f4000      36K rw---- bash
00000000006fd000      24K rw---  [ anon ]
0000000001a31000      3224K rw---  [ anon ]
00007f03653f3000      44K r-x-- libnss_file
00007f03653fe000      2044K ----- libnss_file
00007f03655fd000      4K r----- libnss_file
00007f03655fe000      4K rw--- libnss_file
00007f03655ff000      24K rw---  [ anon ]
```

```
yanyh@vm:~$ cat /proc/7153/maps
00400000-004f4000 r-xp 00000000 08:02 794409
006f3000-006f4000 r--p 000f3000 08:02 794409
006f4000-006fd000 rw-p 000f4000 08:02 794409
006fd000-00703000 rw-p 00000000 00:00 0
01a31000-01d57000 rw-p 00000000 00:00 0
7f03653f3000-7f03653fe000 r-xp 00000000 08:02 917692
-gnu/libnss_files-2.23.so
7f03653fe000-7f03655fd000 ---p 0000b000 08:02 917692
-gnu/libnss_files-2.23.so
7f03655fd000-7f03655fe000 r--p 0000a000 08:02 917692
-gnu/libnss_files-2.23.so
```