

目录

一、C 语言文件如下：	1
二、 编译过程.....	3
2.1、预处理.....	3
2.2、编译.....	3
2.3、汇编.....	7
2.4、链接.....	10
三、 函数调用与返回.....	12
四、 课后作业.....	15
图 1 汇编后查看文件头信息.....	8
图 2 编译后查看 Section 信息.....	9
图 3 查看 CPU 指令.....	10
图 4 链接后查看文件头信息.....	10
图 5 init 字段.....	10
图 6 text 字段.....	11
图 7 只读段.....	11
图 8 数据段.....	11
图 9 未初始化段.....	11
图 10 gcc 串行编译.....	15
图 11 使用 make -j2 编译.....	15
图 12 使用 make -j4 编译.....	16

一、C 语言文件如下：

add.h

```
#ifndef __ADD_H__
```

```
#define __ADD_H__
```

```
#define ADD(a,b) ((a)+(b))
```

```
int add(int a, int b);
```

```
#endif
```

add.c

```
#include "add.h"
```

```
int add(int a, int b)
```

```
{
```

```
    return (a+b);
```

```
}
```

```
sub.h
#ifndef __SUB_H__
#define __SUB_H__

#define SUB(a,b) ((a)-(b))

int sub(int a, int b);

#endif
```

```
sub.c
#include "sub.h"
```

```
int sub(int a, int b)
{
    return (a-b);
}
```

```
main.c
#include "sub.h"
#include "add.h"
int g1=10;
int g2;

int main()
{
    static int a=15;
    int c;

    c = sub(a, g1);
    if(c>0) c = SUB(c, g2);
    else c = ADD(c, g2);

    while (c > 0)
    {
        c--;
    }

    return c;
}
```

二、编译过程

2.1、预处理

```
gcc -E main.i main.c
gcc -E add.i add.c
gcc -E sub.i sub.c
进行宏展开
```

2.2、编译

```
gcc -S main.s main.i
gcc -S add.s add.i
gcc -S sub.s sub.i
```

在 **main.s** 中分析参数传递与调用，内含循环语句

main.s 如下

```
.file "main.c"
.text
.globl    g1                                已经初始化的局部变量 g1
.data
.align 4
.type     g1, @object
.size g1, 4

g1:
.long     10
.comm     g2,4,4
.text
.globl     main
.type     main, @function

main:
.LFB0:
.cfi_startproc
endbr64
pushq     %rbp                            压栈
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq      %rsp, %rbp                      栈顶指向栈基址
.cfi_def_cfa_register 6
subq$16, %rsp                            栈顶 rsp-16
movl      g1(%rip), %edx                  传参，将 g1 传入 edx 寄存器中，g1 为第二个参数
movl      a.1920(%rip), %eax              传参，将 a 传入 eax 寄存器中，a 为第一个参数
movl      %edx, %esi                      将 g1 复制到 esi 寄存器中
movl      %eax, %edi                      将 a 复制到 edi 寄存器中
call      sub@PLT                        调用 sub 函数
movl      %eax, -4(%rbp)                  将 a 移入栈 rbp-4 的位置
```

```

    cmpl$0, -4(%rbp)      a 与 0 进行比较
    jle .L2              a<=0 跳转到 L2
    movl    g2(%rip), %eax
    subl %eax, -4(%rbp)    否则执行 sub 操作
    jmp .L4              接下来跳入循环

.L2:
    movl    g2(%rip), %eax
    addl %eax, -4(%rbp)
    jmp .L4

.L5:
    subl $1, -4(%rbp)      执行 c--操作

.L4:
    cmpl$0, -4(%rbp)      循环条件 c>0
    jg .L5                满足循环条件则跳转到 L5
    movl    -4(%rbp), %eax
    leave
    .cfi_def_cfa 7, 8
    ret                返回
    .cfi_endproc

.LFE0:
    .size main, .-main
    .data
    .align 4
    .type    a.1920, @object
    .size a.1920, 4

a.1920:
    .long    15
    .ident   "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    .section .note.GNU-stack,"",@progbits
    .section .note.gnu.property,"a"
    .align 8
    .long    1f - 0f
    .long    4f - 1f
    .long    5                已初始化的 a 分配在 data 段

0:
    .string   "GNU"

1:
    .align 8
    .long    0xc0000002
    .long    3f - 2f

2:
    .long    0x3

3:
    .align 8

```

4:

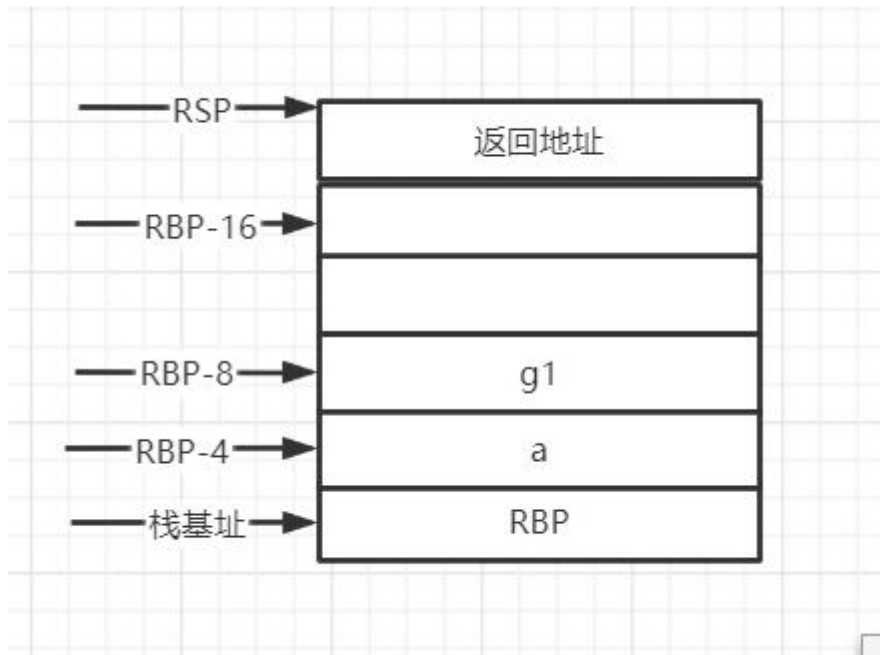
通过被调用函数 **sub** 分析栈内分配过程

sub.s 如下:

```
.file "sub.c"
.text
.globl    sub
.type     sub, @function
sub:
.LFB0:
.cfi_startproc
endbr64
pushq    %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq     %rsp, %rbp
.cfi_def_cfa_register 6
movl     %edi, -4(%rbp)    参数 a 压栈
movl     %esi, -8(%rbp)    参数 g1 压栈
movl     -4(%rbp), %eax    将 a 放入 eax 寄存器中
subl     -8(%rbp), %eax    a-g1 进行 sub 操作保存在 eax 中
popq     %rbp             出栈, 恢复初始状态
.cfi_def_cfa 7, 8
ret                               通过 rip 返回地址
.cfi_endproc
.LFE0:
.size sub, .-sub
.ident   "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04) 9.4.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 8
.long    1f - 0f
.long    4f - 1f
.long    5
0:
.string   "GNU"
1:
.align 8
.long    0xc0000002
.long    3f - 2f
2:
.long    0x3
3:
```

.align 8

4:



使用 `clang -emit-llvm -S -o main.ll main.c` 生成
main.ll 如下:

```
@g1 = dso_local global i32 10, align 4
```

```
@main.a = internal global i32 15, align 4
```

```
@g2 = common dso_local global i32 0, align 4
```

```
; Function Attrs: noline nounwind optnone uwtable
```

```
define dso_local i32 @main() #0 {
```

```
    %1 = alloca i32, align 4
```

//分配变量 c 空间

```
    %2 = alloca i32, align 4
```

//分配变量 a 空间

```
    store i32 0, i32* %1, align 4
```

//置 c 为 0

```
    %3 = load i32, i32* @main.a, align 4
```

//加载 a

```
    %4 = load i32, i32* @g1, align 4
```

//加载 g1

```
    %5 = call i32 @sub(i32 %3, i32 %4)
```

//调用 sub 函数

```
    store i32 %5, i32* %2, align 4
```

//保存 sub 函数结果

```
    %6 = load i32, i32* %2, align 4
```

```
    %7 = icmp sgt i32 %6, 0
```

//c 与 0 比较

```
    br i1 %7, label %8, label %12
```

//大于 0 跳转到 label 8, 否则跳转到 label 12

8:

; preds = %0

```
    %9 = load i32, i32* %2, align 4
```

```

%10 = load i32, i32* @g2, align 4
%11 = sub nsw i32 %9, %10      //进行 SUB 操作
store i32 %11, i32* %2, align 4
br label %16                  //跳转到 label 16

12:                             ; preds = %0
    %13 = load i32, i32* %2, align 4
    %14 = load i32, i32* @g2, align 4
    %15 = add nsw i32 %13, %14  //进行 ADD 操作
    store i32 %15, i32* %2, align 4
    br label %16

16:                             ; preds = %12, %8
    br label %17

17:                             ; preds = %20, %16
    %18 = load i32, i32* %2, align 4
    %19 = icmp sgt i32 %18, 0
    br i1 %19, label %20, label %23  //若 c>0 则跳转到 label 20, 否则跳出循环

20:                             ; preds = %17
    %21 = load i32, i32* %2, align 4
    %22 = add nsw i32 %21, -1
    store i32 %22, i32* %2, align 4
    br label %17              //返回 label 17 再次进入循环

23:                             ; preds = %17
    %24 = load i32, i32* %2, align 4
    ret i32 %24
}

```

```
declare dso_local i32 @sub(i32, i32) #1
```

2.3、汇编

```

gcc -c -o main.o main.s
gcc -c -o add.o add.s
gcc -c -o sub.o sub.s

```

```
objdump -f main.o
```

地址未知需要重定位

```
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$ objdump -f main.o  
  
main.o:      file format elf64-x86-64  
architecture: i386:x86-64, flags 0x00000011:  
HAS_RELOC, HAS_SYMS  
start address 0x0000000000000000  
  
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$
```

图 1 汇编后查看文件头信息

objdump -x main.o Section 信息


```

Sections:
Idx Name          Size      VMA              LMA              File off  Algn
  0 .text          0000004f  0000000000000000  0000000000000000  00000040  2**0
                        CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data          00000008  0000000000000000  0000000000000000  00000090  2**2
                        CONTENTS, ALLOC, LOAD, DATA
  2 .bss           00000000  0000000000000000  0000000000000000  00000098  2**0
                        ALLOC
  3 .comment       0000002b  0000000000000000  0000000000000000  00000098  2**0
                        CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000c3  2**0
                        CONTENTS, READONLY
  5 .note.gnu.property 00000020  0000000000000000  0000000000000000  000000c8  2**3
                        CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .eh_frame      00000038  0000000000000000  0000000000000000  000000e8  2**3
                        CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

SYMBOL TABLE:
0000000000000000 l   df *ABS*  0000000000000000 main.c
0000000000000000 l   d  .text  0000000000000000 .text
0000000000000000 l   d  .data  0000000000000000 .data
0000000000000000 l   d  .bss   0000000000000000 .bss
0000000000000004 l   0  .data  0000000000000004 a.1920
0000000000000000 l   d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l   d  .note.gnu.property 0000000000000000 .note.gnu.property
0000000000000000 l   d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l   d  .comment 0000000000000000 .comment
0000000000000000 g   0  .data  0000000000000004 g1
0000000000000004 0  *COM*  0000000000000004 g2
0000000000000000 g   F  .text  000000000000004f main
0000000000000000 *UND* 0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000 *UND* 0000000000000000 sub

RELOCATION RECORDS FOR [.text]:
OFFSET          TYPE          VALUE
000000000000000e R_X86_64_PC32  g1-0x0000000000000004
0000000000000014 R_X86_64_PC32  .data
000000000000001d R_X86_64_PLT32  sub-0x0000000000000004
000000000000002c R_X86_64_PC32  g2-0x0000000000000004
0000000000000037 R_X86_64_PC32  g2-0x0000000000000004

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET          TYPE          VALUE
0000000000000020 R_X86_64_PC32  .text

```

图 2 编译后查看 Section 信息

.text 代码段， CPU 指令

.data 数据段， 初始化全局变量或者静态变量等

.bss 未初始化段， 包含未初始化的全局变量或者静态变量等

symbol table 符号表， 主要包含 section 符号、函数符号等

重定位记录， 地址还不是最终的地址

objdump -d main.o CPU 指令

```

main.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0:  f3 0f 1e fa                endbr64
 4:  55                          push   %rbp
 5:  48 89 e5                    mov     %rsp,%rbp
 8:  48 83 ec 10                 sub     $0x10,%rsp
 c:  8b 15 00 00 00 00          mov     0x0(%rip),%edx        # 12 <main+0x12>
12:  8b 05 00 00 00 00          mov     0x0(%rip),%eax        # 18 <main+0x18>
18:  89 d6                       mov     %edx,%esi
1a:  89 c7                       mov     %eax,%edi
1c:  e8 00 00 00 00            callq   21 <main+0x21>
21:  89 45 fc                    mov     %eax,-0x4(%rbp)
24:  83 7d fc 00                cmpl    $0x0,-0x4(%rbp)
28:  7e 0b                       jle     35 <main+0x35>
2a:  8b 05 00 00 00 00          mov     0x0(%rip),%eax        # 30 <main+0x30>
30:  29 45 fc                    sub     %eax,-0x4(%rbp)
33:  eb 0f                       jmp     44 <main+0x44>
35:  8b 05 00 00 00 00          mov     0x0(%rip),%eax        # 3b <main+0x3b>
3b:  01 45 fc                    add     %eax,-0x4(%rbp)
3e:  eb 04                       jmp     44 <main+0x44>
40:  83 6d fc 01                subl    $0x1,-0x4(%rbp)
44:  83 7d fc 00                cmpl    $0x0,-0x4(%rbp)
48:  7f f6                       jg      40 <main+0x40>
4a:  8b 45 fc                    mov     -0x4(%rbp),%eax
4d:  c9                          leaveq  %eax
4e:  c3                          retq

```

图 3 查看 CPU 指令

后面的#12、#18、#30、#3b 都需要重定位，此时的地址都是 0x0000

2.4、链接

```
gcc -o main main.o add.o sub.o
```

```
objdump -f main 文件头信息
```

EXEC_P 表示取消了可重定位标志

```

zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$ gcc -static -o main main.o add
.o sub.o
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$ objdump -f main

main:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000401bc0

```

图 4 链接后查看文件头信息

```
objdump -x main Section 节选信息
```

```

4 .init      0000001b 0000000000401000 0000000000401000 00001000 2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE

```

图 5 init 字段

.text 代码段

6	.text	CONTENTS, ALLOC, LOAD, READONLY, CODE	00091900	00000000004011a0	00000000004011a0	000011a0	2**4
		CONTENTS, ALLOC, LOAD, READONLY, CODE					

图 6 text 字段

.rodata 只读段

9	.rodata	CONTENTS, ALLOC, LOAD, READONLY, CODE	0001bfec	0000000000495000	0000000000495000	00095000	2**5
		CONTENTS, ALLOC, LOAD, READONLY, DATA					

图 7 只读段

.data 数据段包含 g1 全局变量

20	.data	CONTENTS, ALLOC, LOAD, DATA	00001a50	00000000004c00e0	00000000004c00e0	000bf0e0	2**5
		CONTENTS, ALLOC, LOAD, DATA					

图 8 数据段

.bss 未初始化段包含为初始化变量 g2

24	.bss	CONTENTS, ALLOC, LOAD, DATA	00001738	00000000004c2240	00000000004c2240	000c1230	2**5
		ALLOC					

图 9 未初始化段

objdump -d main

0000000000001129 <main>:

```
1129: f3 0f 1e fa      endbr64
112d: 55              push    %rbp
112e: 48 89 e5        mov     %rsp,%rbp
1131: 48 83 ec 10     sub     $0x10,%rsp
1135: 8b 15 d5 2e 00 00 mov     0x2ed5(%rip),%edx      # 4010 <g1>
113b: 8b 05 d3 2e 00 00 mov     0x2ed3(%rip),%eax      # 4014 <a.1922>
1141: 89 d6          mov     %edx,%esi
1143: 89 c7          mov     %eax,%edi
1145: e8 46 00 00 00 callq   1190 <sub> //与 sub 初始地址对应
114a: 89 45 fc        mov     %eax,-0x4(%rbp)
114d: 83 7d fc 00     cmpl    $0x0,-0x4(%rbp)
1151: 7e 0b          jle     115e <main+0x35>
1153: 8b 05 c3 2e 00 00 mov     0x2ec3(%rip),%eax      # 401c <g2>
1159: 29 45 fc        sub     %eax,-0x4(%rbp)
115c: eb 0f          jmp     116d <main+0x44>
115e: 8b 05 b8 2e 00 00 mov     0x2eb8(%rip),%eax      # 401c <g2>
1164: 01 45 fc        add     %eax,-0x4(%rbp)
1167: eb 04          jmp     116d <main+0x44>
1169: 83 6d fc 01     subl    $0x1,-0x4(%rbp)
```

```

116d: 83 7d fc 00      cmpl    $0x0,-0x4(%rbp)
1171: 7f f6            jg      1169 <main+0x40>
1173: 8b 45 fc          mov     -0x4(%rbp),%eax
1176: c9               leaveq
1177: c3               retq

```

0000000000001178 <add>:

```

1178: f3 0f 1e fa      endbr64
117c: 55               push    %rbp
117d: 48 89 e5          mov     %rsp,%rbp
1180: 89 7d fc          mov     %edi,-0x4(%rbp)
1183: 89 75 f8          mov     %esi,-0x8(%rbp)
1186: 8b 55 fc          mov     -0x4(%rbp),%edx
1189: 8b 45 f8          mov     -0x8(%rbp),%eax
118c: 01 d0            add     %edx,%eax
118e: 5d               pop     %rbp
118f: c3               retq

```

0000000000001190 <sub>:

```

1190: f3 0f 1e fa      endbr64
1194: 55               push    %rbp
1195: 48 89 e5          mov     %rsp,%rbp
1198: 89 7d fc          mov     %edi,-0x4(%rbp)
119b: 89 75 f8          mov     %esi,-0x8(%rbp)
119e: 8b 45 fc          mov     -0x4(%rbp),%eax
11a1: 2b 45 f8          sub     -0x8(%rbp),%eax
11a4: 5d               pop     %rbp
11a5: c3               retq
11a6: 66 2e 0f 1f 84 00 00  nopw    %cs:0x0(%rax,%rax,1)
11ad: 00 00 00

```

三、函数调用与返回

编写 test.c 如下:

```

int ADD(int a, int b, int c, int d, int e, int f)
{
    return (a+b+c+e+d+f);
}

```

```

int main()
{
    int g;

    g = ADD(1,2,3,4,5,6);
}

```

```
    return 0;
}
```

分析汇编语言如下：

```
.file "test.c"
.text
.globl  ADD
.type   ADD, @function
ADD:
.LFB0:
.cfi_startproc
endbr64
pushq   %rbp                                压栈
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq     %rsp, %rbp
.cfi_def_cfa_register 6
movl     %edi, -4(%rbp)                    变量 a
movl     %esi, -8(%rbp)                    变量 b
movl     %edx, -12(%rbp)                   变量 c
movl     %ecx, -16(%rbp)                   变量 d
movl     %r8d, -20(%rbp)                   变量 e
movl     %r9d, -24(%rbp)                   变量 f
movl     -4(%rbp), %edx
movl     -8(%rbp), %eax
addl %eax, %edx                            a+b
movl     -12(%rbp), %eax
addl %eax, %edx                            +c
movl     -20(%rbp), %eax
addl %eax, %edx                            +d
movl     -16(%rbp), %eax
addl %eax, %edx                            +e
movl     -24(%rbp), %eax
addl %edx, %eax                            +f 通过 eax 寄存器返回
popq     %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size ADD, .-ADD
.globl  main
.type   main, @function
main:
```

```

.LFB1:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    subq$16, %rsp
    movl     $6, %r9d
    movl     $5, %r8d
    movl     $4, %ecx
    movl     $3, %edx
    movl     $2, %esi
    movl     $1, %edi
    call     ADD
    movl     %eax, -4(%rbp)
    movl     $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

.LFE1:
    .size main, .-main
    .ident    "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04) 9.4.0"
    .section  .note.GNU-stack,"",@progbits
    .section  .note.gnu.property,"a"
    .align 8
    .long     1f - 0f
    .long     4f - 1f
    .long     5
0:
    .string    "GNU"
1:
    .align 8
    .long     0xc0000002
    .long     3f - 2f
2:
    .long     0x3
3:
    .align 8
4:

```


四、课后作业

(1) 由上编写的函数调用文件的汇编语言分析

```
pushq    %rbp                                压栈
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq     %rsp, %rbp
.cfi_def_cfa_register 6
movl     %edi, -4(%rbp)                      变量 a
movl     %esi, -8(%rbp)                      变量 b
movl     %edx, -12(%rbp)                     变量 c
movl     %ecx, -16(%rbp)                     变量 d
movl     %r8d, -20(%rbp)                     变量 e
movl     %r9d, -24(%rbp)                     变量 f
```

可知 函数的形式参数和局部变量保存在栈内
编译器通过堆来管理动态内存

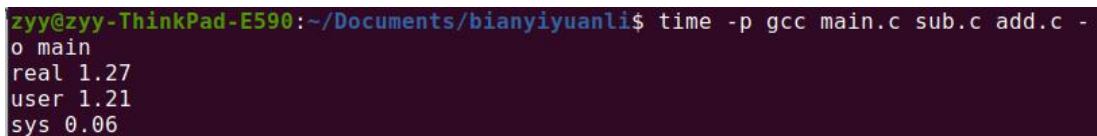
(2) cmake 文件如下:

```
cmake_minimum_required(VERSION 2.8)
project( Main )
```

```
add_executable( Main main.c sub.c add.c)
```

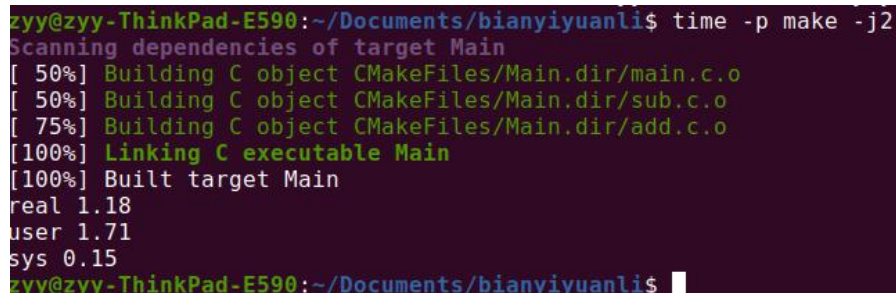
适当增加代码行数，使用 `time -p` 查看编译时间

如图所示使用 `gcc` 串行编译 `main.c add.c sub.c`, 而 `make -j2` 以及 `make -j4` 使用并行编译+串行链接



```
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$ time -p gcc main.c sub.c add.c -
o main
real 1.27
user 1.21
sys 0.06
```

图 10 gcc 串行编译



```
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$ time -p make -j2
Scanning dependencies of target Main
[ 50%] Building C object CMakeFiles/Main.dir/main.c.o
[ 50%] Building C object CMakeFiles/Main.dir/sub.c.o
[ 75%] Building C object CMakeFiles/Main.dir/add.c.o
[100%] Linking C executable Main
[100%] Built target Main
real 1.18
user 1.71
sys 0.15
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$
```

图 11 使用 make -j2 编译

```
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$ time -p make -j4
Scanning dependencies of target Main
[ 75%] Building C object CMakeFiles/Main.dir/add.c.o
[ 75%] Building C object CMakeFiles/Main.dir/sub.c.o
[ 75%] Building C object CMakeFiles/Main.dir/main.c.o
[100%] Linking C executable Main
[100%] Built target Main
real 0.88
user 2.03
sys 0.13
zyy@zyy-ThinkPad-E590:~/Documents/bianyiyuanli$
```

图 12 使用 make -j4 编译

实验多次后得到如下结果

次数	gcc	make -j2	make -j4
1	1.27s	1.18s	0.88s
2	1.39s	1.16s	1.07s
3	1.38s	1.09s	0.95s
4	1.32s	1.06s	0.87s
5	1.36s	1.01s	0.97s
平均值	1.344s	1.1s	0.948s

从结果看, make -jn 的结果要比 gcc 串行编译更好, 但是效果并不显著, 应该是串行链接时花费了较多时间且此处只并行编译了三个文件, 如果文件更多效果将会更加明显。