

计算机系统基础
Programming Assignment

PA 2-0 – 汇编基础知识先导课

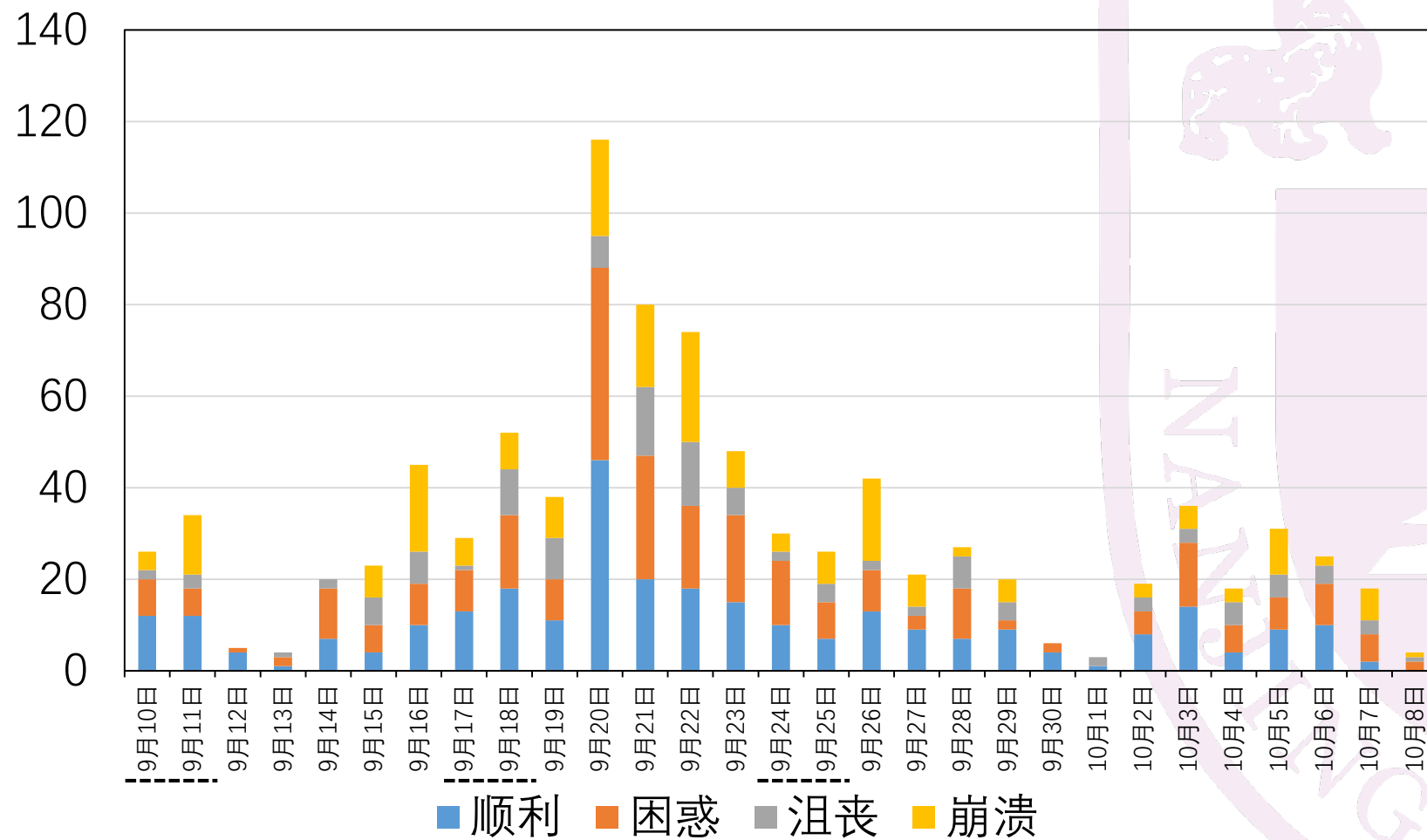
2020年10月9日 / 2020年10月10日
南京大学《计算机系统基础》课程组

习题布置

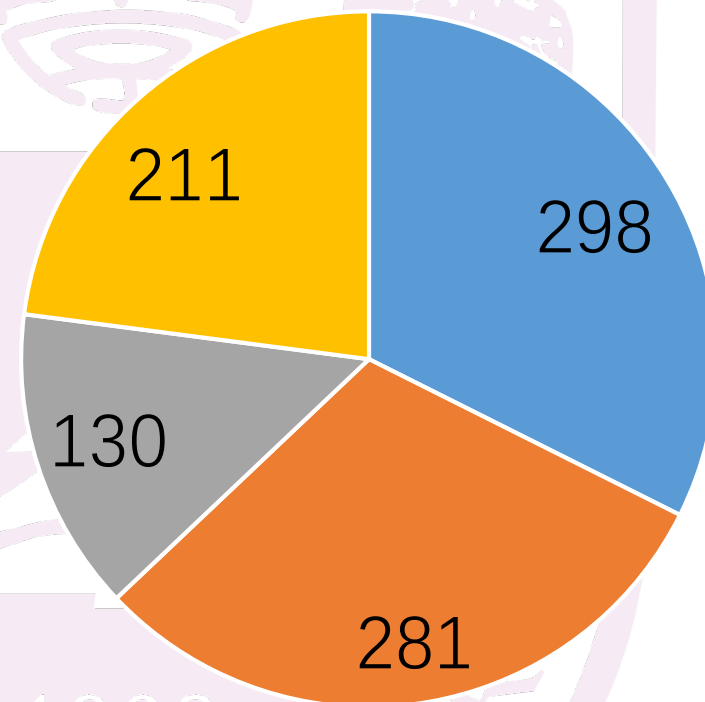
- 第二版课本第2章习题, pg. 79开始, 习题号:
 - 3、9、10、15、21、29、33、40
- 课后作业习题Deadline
 - 2020年10月18日24点
- 作业以电子稿形式上传到cms系统



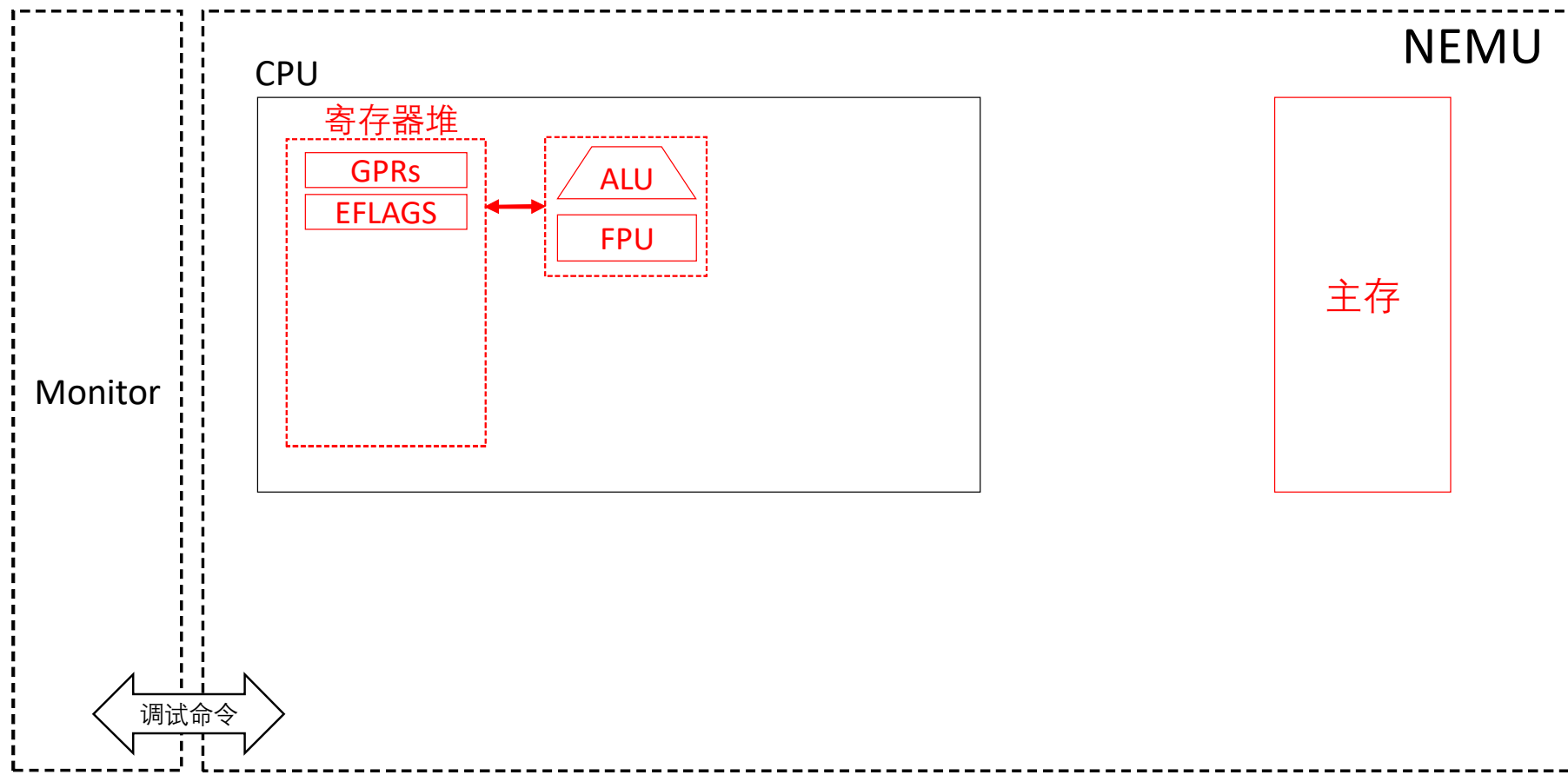
PA-1阶段每日情绪分布



PA-1阶段总体情绪分布百分比



前情提要



目录

- 从高级语言到机器指令 – C语言程序编译的过程
- 直接用汇编语言写程序
- 尝试用gdb调试程序



目录

- 从高级语言到机器指令 - C语言程序编译的过程
- 直接用汇编语言写程序
- 尝试用gdb调试程序



从高级语言到机器指令 — C语言程序编译的过程

hello_world.c

```
#include<stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

```
$ gcc -o hello_world hello_world.c
```

```
$ ./hello_world
```

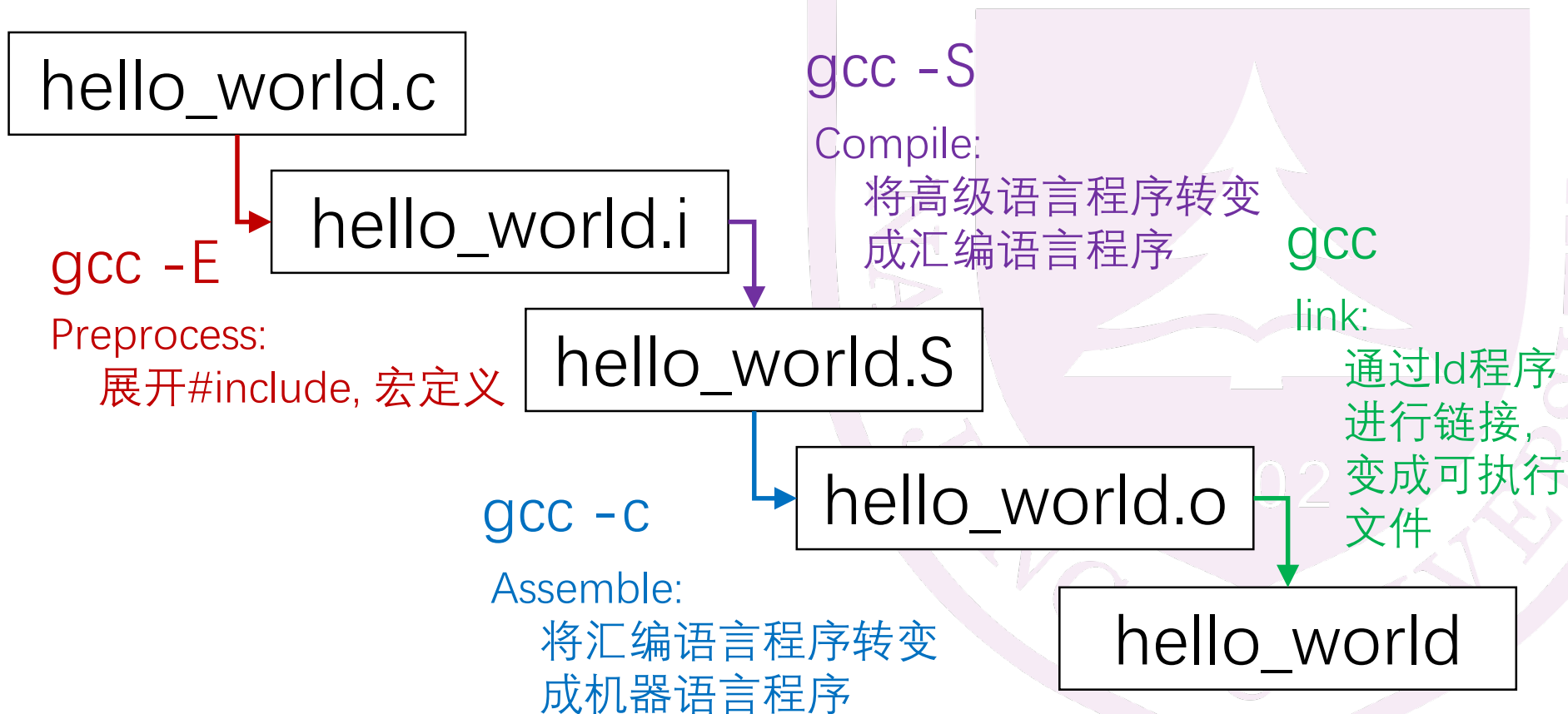
```
Hello World!
```

编译
运行

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.c
```

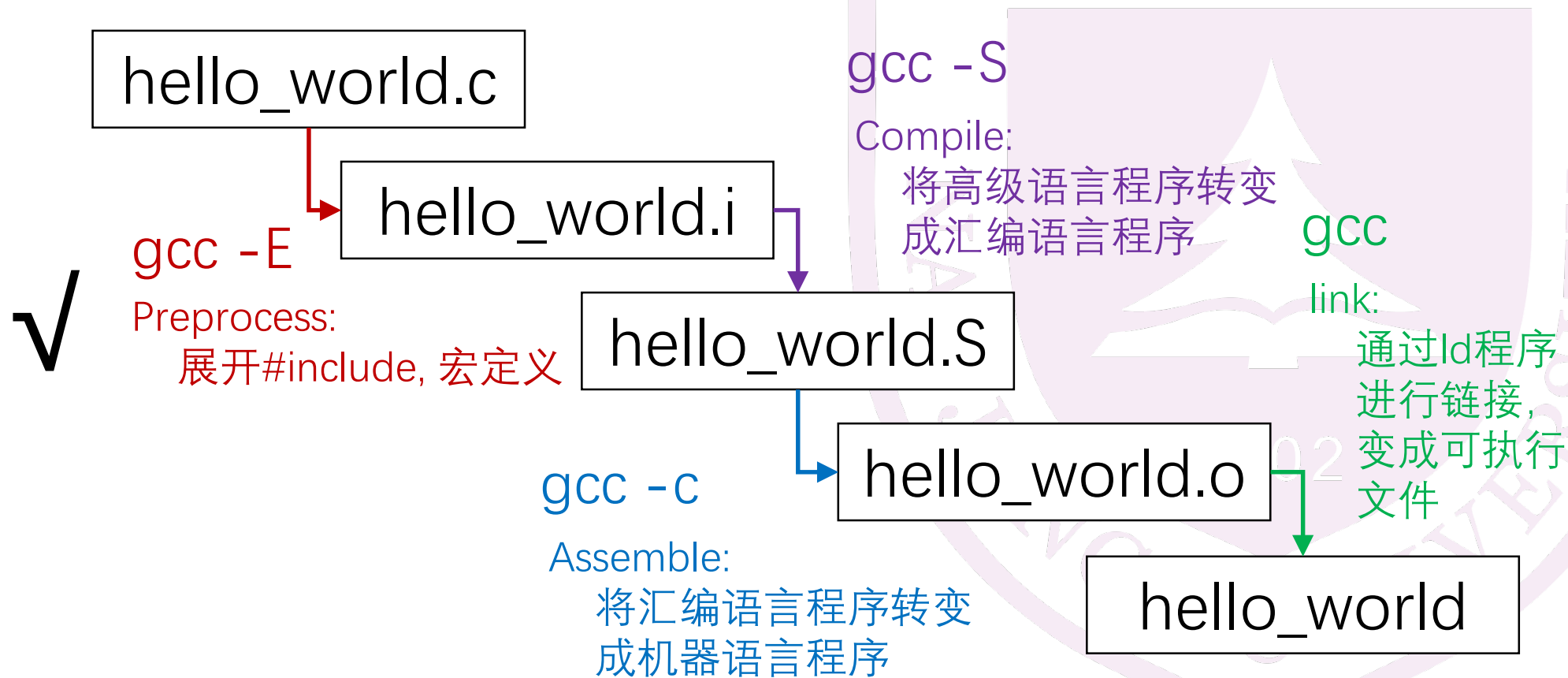
一条命令执行了四个步骤



从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.c
```

一条命令执行了四个步骤



从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -E -o hello_world.i hello_world.c  
$ cat hello_world.i | less
```

hello_world.i 预处理的结果

```
... // 一大堆c语言的代码, 是stdio.h展开的结果  
extern int fprintf (FILE *__restrict __stream,  
    const char *__restrict __format, ...);  
  
extern int printf (const char *__restrict __format, ...); // printf的声明  
  
extern int sprintf (char *__restrict __s,  
    const char *__restrict __format, ...) __attribute__((__nothrow__));  
...  
# 3 "hello_world.c" // 最后, 是我们的程序代码  
  
int main() {  
    printf("Hello World!\n");  
    return 0;  
}
```

2020/10/9

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -E -o hello_world.i hello_world.c
$ cat hello_world.i | less
```

hello_world.i 预处理的结果

... // 一大堆c语言的代码 是stdio.h展开的结果

```
extern int fprintf (F
    const char *__r
```

```
extern int printf (co
```

```
extern int sprintf (c
    const char *__r
```

```
...
# 3 "hello_world.c"
```

```
int main() {
    printf("Hello World!");
    return 0;
}
```

2020/10/9

宏的预处理

```
// nemu/include/memory/memory.h
#define MEM_SIZE_B 128 * 1024 * 1024
```

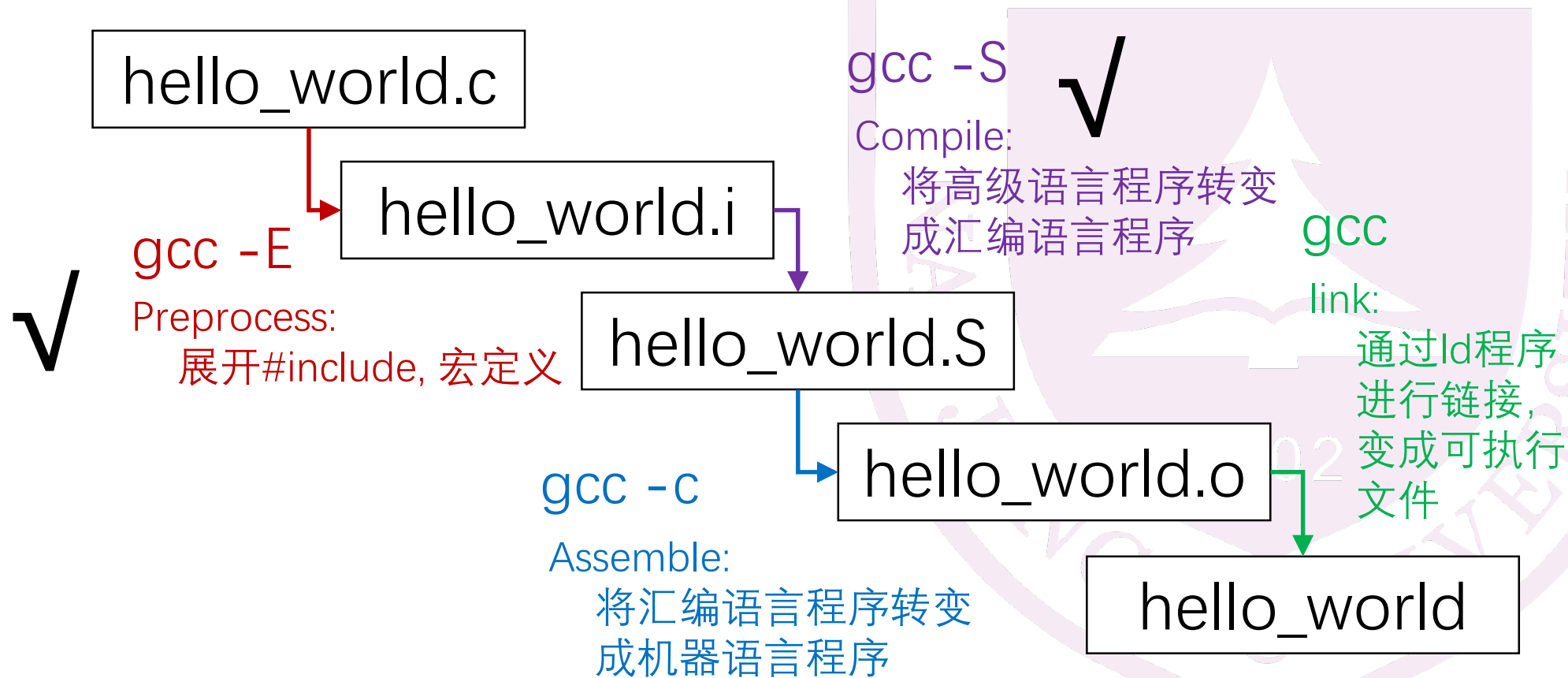
```
// nemu/src/memory/memory.c
uint8_t hw_mem[MEM_SIZE_B];
```

```
// nemu/src/memory/memory.i
uint8_t hw_mem[128 * 1024 * 1024];
```

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.c
```

一条命令执行了四个步骤



从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -S -o hello_world.S hello_world.c
$ cat hello_world.S | less
```

hello_world.S 编译的结果

```
.file "hello_world.c"
.section .rodata // 只读数据
.LC0:
.string "Hello World!"
.text // 代码
.globl main // 全局符号main
.type main, @function // main函数的汇编代码从这里开始
main:
.LFB0:
.cfi_startproc
leal 4(%esp), %ecx
.cfi_def_cfa 1, 0
andl $-16, %esp
pushl -4(%ecx)
```

汇编指令

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -S -o hello_world.S hello_world.c
$ cat hello_world.S | less
```

hello_world.S 编译的结果

```
.file "hello_world.c"
.section .rodata // 只读数据
.LC0:
.string "Hello World!"
.text // 代码
.globl main // 全局符号main
.type main, @function // main函数的汇编
main:
.LFB0:
.cfi_startproc
leal 4(%esp), %ecx
.cfi_def_cfa 1, 0
andl $-16, %esp
pushl -4(%ecx)
```

如何从C语言程序转化为汇编语言程序?

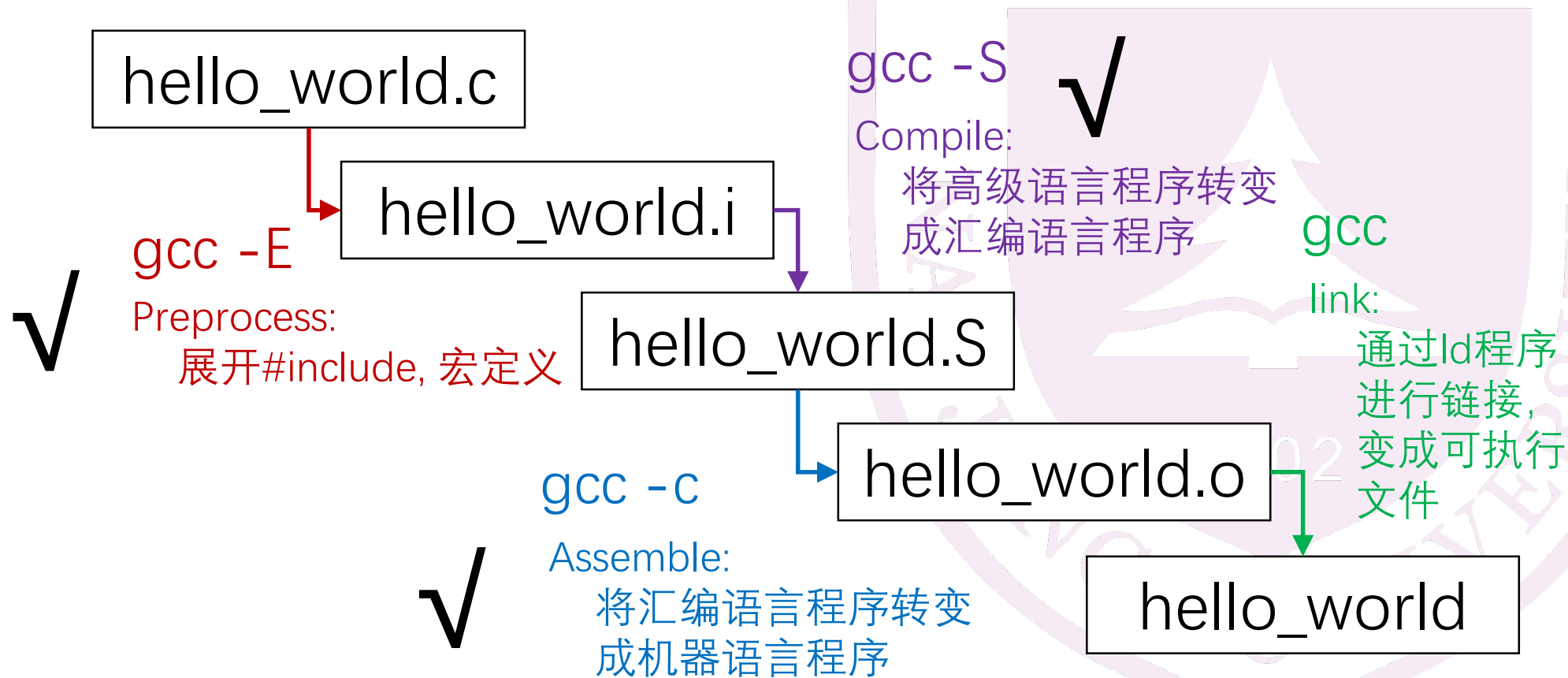
进一步学习: 编译原理

汇编指令

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.c
```

一条命令执行了四个步骤



从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -c -o hello_world.o hello_world.S  
$ hexdump hello_world.o | less
```

[hello_world.o](#)

[查看其内容](#)

00000000	457f	464c	0101	0001	0000	0000	0000	0000
00000010	0001	0003	0001	0000	0000	0000	0000	0000
00000020	02fc	0000	0000	0000	0034	0000	0000	0028
00000030	000f	000e	0001	0000	0007	0000	4c8d	0424
00000040	e483	fff0	fc71	8955	53e5	e851	fffc	ffff
00000050	0105	0000	8300	0cec	908d	0000	0000	8952
00000060	e8c3	fffc	ffff	c483	b810	0000	0000	658d
00000070	59f8	5d5b	618d	c3fc	6548	6c6c	206f	6f57
00000080	6c72	2164	8b00	2404	00c3	4347	3a43	2820
00000090	6544	6962	6e61	3620	332e	302e	312d	2938

机器语言程序!

偏移量

数据，两字节一组，小端

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -c -o hello_world.o hello_world.S  
$ objdump -d hello_world.o | less
```

hello_world.o 反汇编其内容

hello_world.o: file format elf32-i386

Disassembly of section .text:

00000000 <main>:

0: 8d 4c 24 04
4: 83 e4 f0
7: ff 71 fc
a: 55
b: 89 e5
d: 53
e: 51
f: e8 fc ff ff ff

lea 0x4(%esp),%ecx
and \$0xfffffffff0,%esp
pushl -0x4(%ecx)
push %ebp
mov %esp,%ebp
push %ebx
push %ecx
call 10 <main+0x10>

机器读取
并执行的
机器指令

对应的汇
编助记符

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -c -o hello_world.o hello_world.S
$ objdump -d hello_world.o | less
```

hello_world.o 反汇编其内容

hello_world.o: file format elf32-i386

Disassembly of section .text:

00000000 <main>:

0: 8d 4c 24 04
4: 83 e4 f0
7: ff 71 fc
a: 55
b: 89 e5
d: 53
e: 51
f: e8 fc ff ff ff

编码

——对应

真值 (误

lea 0x4(%esp),%ecx
and \$0xfffffffff0,%esp
pushl -0x4(%ecx)
push %ebp
mov %esp,%ebp
push %ebx
push %ecx
call 10 <main+0x10>

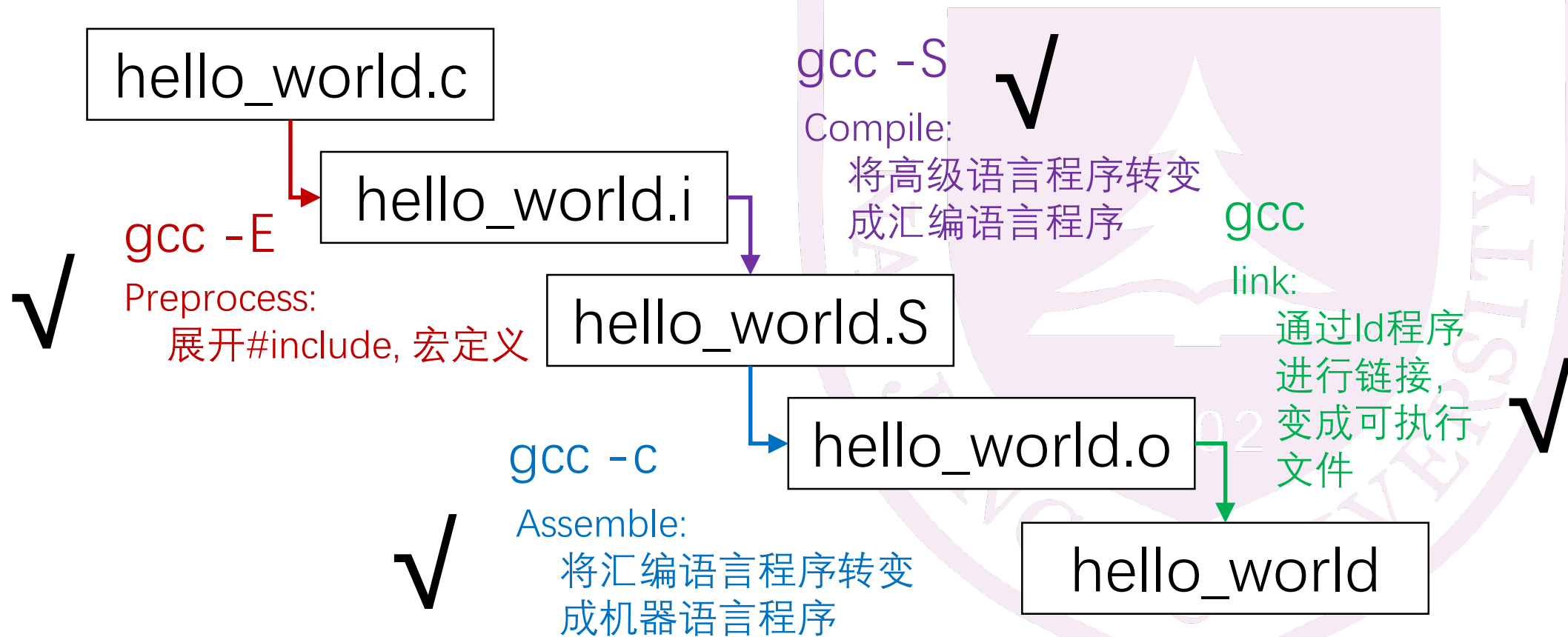
对应的汇编助记符

机器读取并执行的机器指令

从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.c
```

一条命令执行了四个步骤



从高级语言到机器指令 – C语言程序编译的过程

```
$ gcc -o hello_world hello_world.o
```

```
$ ./hello_world
```

```
Hello World!
```

- 最后将hello_world.o通过ld程序链接其它模块和库文件，得到最终的可执行文件hello_world
- 此系后话（PA 2-2，2-3简要认识），此时先略去不表

和PA的关系

testcase/src/add.c

```
int main()
{
    ...
    return 0;
}
```

gcc

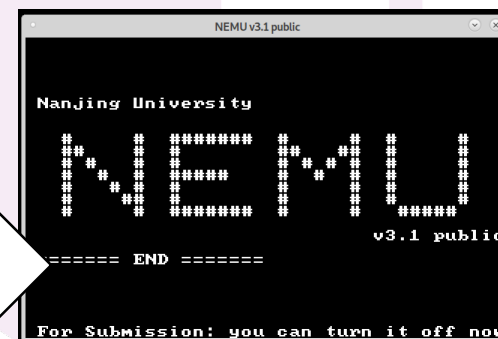
testcase/bin/add

bin/add: file format elf32-i386

Disassembly of section .text:

```
00030000 <start>:
    30000: e9 00 00 00 00
00030005 <main>:
    30005: 55
    30006: 89 e5
    30008: 53
    30009: 83 ec 10
    3000c: e8 8f 00 00 00
```

装载



nemu本身
也被编译成
二进制的可
执行文件

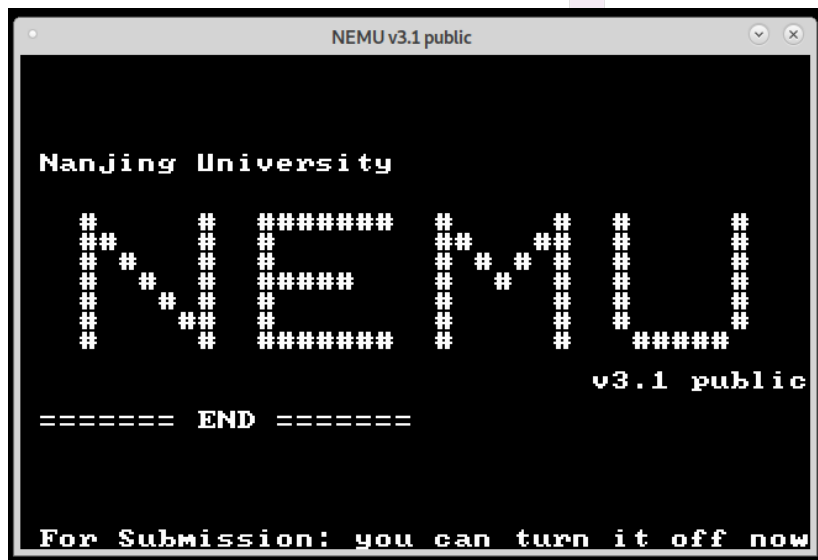
和PA的关系

模拟CPU
解释执行

testcase/bin/add

执行

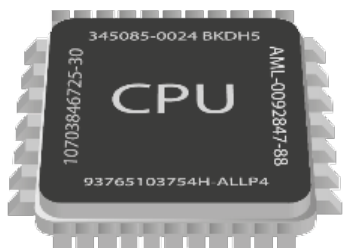
虚拟机 + Linux



都是x86指令的序列

小结

- 不管什么语言写的程序，最后交给CPU执行的，都是机器指令的序列
- 这些指令与对应的汇编助记符一一对应



e9 00 00 00 00 55 89 e5 53 83 ec 10 e

jmp 30005; push %ebp; mov %esp,%ebp; push %ebx; sub \$0x

重要提示!

用objdump命令反汇编目标文件

```
$ objdump -d hello_world.o | less
```



标准的objdump无法识别我们为PA定制的0x82指令，因此我们提供了自己的改造版本，用于反汇编PA的测试用例

objdump4nemu-i386

源码: <https://gitee.com/wlicsnju/binutils4nemu>

目录

- 从高级语言到机器指令 - C语言程序编译的过程
- 直接用汇编语言写程序
- 尝试用gdb调试程序



汇编语言编程

- 如何用汇编语言写一个Hello World程序？

- 第一步：创建一个文本文件，`hello.S`
- 第二步：写入汇编代码
- 第三步：编译运行

```
$ gcc -o hello hello.S  
$ ./hello  
$ Hello World!
```

```
.data  
hello_str: .ascii "Hello World!\n"  
  
.text  
.globl main  
main:  
    movl $4, %eax  
    movl $1, %ebx  
    movl $hello_str, %ecx  
    movl $13, %edx  
    int $0x80
```

汇编语言编程

```
.data
hello_str: .ascii "Hello World!\n"

.text
.globl main
main:
    movl $4, %eax
    movl $1, %ebx
    movl $hello_str, %ecx
    movl $13, %edx
    int $0x80
```

汇编语言编程

数据

```
.data  
hello_str: .ascii "Hello World!\n"
```

代码

```
.text  
.globl main  
main:  
    movl $4, %eax  
    movl $1, %ebx  
    movl $hello_str, %ecx  
    movl $13, %edx  
    int $0x80
```

汇编语言编程

符号
相当于全局变量名

符号的值
符号的值等于其指向区域的地址

```
.data
hello_str: .ascii "Hello World!\n"

.text
.globl main
main:
    movl $4, %eax
    movl $1, %ebx
    movl $hello_str, %ecx
    movl $13, %edx
    int $0x80
```

类型

初始值

其它一些类型

.byte
.short
.long
.string

.zero #字节数

汇编语言编程

全局符号
main

程序的入口，
链接器需要它

```
.data
hello_str: .ascii "Hello World!\n"

.text
.globl main
main:

    movl $4, %eax
    movl $1, %ebx
    movl $hello_str, %ecx
    movl $13, %edx
    int $0x80
```

汇编语言编程

```
.data
hello_str: .ascii "Hello World!\n"

.text
.globl main
main:
    movl $4, %eax
    movl $1, %ebx
    movl $hello_str, %ecx
    movl $13, %edx
    int $0x80
```

从第一条指令开始顺序往下执行，遇到jmp、call等指令则跳转



汇编语言编程（指令的格式）

(gcc接受的格式，也是我们写程序采用的格式)

AT&T格式： 指令 长度后缀 源操作数, 目的操作数

`movl $0x7, %eax`

`MOV EAX, 0x7`

INTEL格式： 指令 目的操作数, 源操作数

(i386手册上采用的格式)

汇编语言编程（指令的格式）

(gcc接受的格式，也是我们写程序采用的格式)

AT&T格式：指令 长度后缀 源操作数, 目的操作数

movl \$0x7, %eax

MOV EAX, 0x7

INTEL格式：指令 目的操作数, 源操作数

(i386手册上采用的格式)

汇编语言编程

- i386指令集：有哪些指令我们可以用？

类型	举例 (长度后缀省略)
传送指令	mov, xchg, push, lea, ..., in, out, ...
定点算术运算指令	add, sub, mul, div, inc, dec, cmp, ...
按位运算指令	and, not, or, xor, test, shr, shl, ...
控制转移指令	call, ret, jmp, jne, jb, ..., int, iret, ...

详见i386手册，以及《实验指导 (Guide)》第一页上列举的参考资料和网站，注意Intel和At&t格式的区别

汇编语言编程（指令的格式）

(gcc接受的格式，也是我们写程序采用的格式)

AT&T格式：指令长度后缀 源操作数，目的操作数

`movl $0x7, %eax`

`MOV EAX, 0x7`

INTEL格式：指令 目的操作数，源操作数

(i386手册上采用的格式)

汇编语言编程（长度后缀）

名称	长度（比特）	长度后缀
字节 Byte	8	b
字 Word	16	w
双字 Double Word	32	l

`movb $0x7, %al`

`movw $0x7, %ax`

`movl $0x7, %eax`

和alu实现中的data_size相关

汇编语言编程（指令的格式）

(gcc接受的格式，也是我们写程序采用的格式)

AT&T格式：指令长度后缀源操作数，目的操作数

`movl $0x7, %eax`

`MOV EAX, 0x7`

INTEL格式：指令目的操作数，源操作数

(i386手册上采用的格式)

汇编语言编程（操作数寻址）

课本pg. 93, 图3.4

操作数硬编
码在指令

立即数寻址

寄存器寻址

操作数在
寄存器

movl \$0x7, %eax

汇编语言编程（操作数寻址）

课本pg. 93, 图3.4

操作数硬编
码在指令

立即数寻址

寄存器寻址

操作数在
寄存器

`movl $0x7, %eax`

可概括除了立即数和寄存器寻址以外的各种寻址方式:

`movl 0x1100(%ebx, %eax, 4), %edx`

操作数在
内存

基址加比例变址加位移

内存地址 = 基址 + 变址 * 比例 + 位移

汇编语言编程

- 如何用汇编语言写一个Hello World程序？

- 第一步：创建一个文本文件，`hello.S`
- 第二步：写入汇编代码
- 第三步：编译运行

```
$ gcc -o hello hello.S  
$ ./hello  
$ Hello World!
```

```
.data  
hello_str: .ascii "Hello World!\n"  
  
.text  
.globl main  
main:  
    movl $4, %eax  
    movl $1, %ebx  
    movl $hello_str, %ecx  
    movl $13, %edx  
    int $0x80
```


熟练掌握汇编语言的重要性

- 能够看懂并修改底层代码
 - Linux kernel
 - BIOS
- 逆向工程，掌握程序行为的细节信息
- 对程序的高度优化
- 显得特别专业特别厉害 ✧ (◡ ◡ ◡)

小练习1

```
.globl main
main:
    movl $1, %eax
    movl $1, %ebx
loop:
    movl %ebx, %ecx
    addl %eax, %ecx
    movl %ebx, %eax
    movl %ecx, %ebx
    jmp loop
```

每一步中，eax, ecx, ebx中保存的是几？
这是一个求解什么的程序？

小练习2

.fill的含义:

.fill repeat , size , value

含义是反复拷贝size个字节, 重复repeat次, 其中size和value是可选的, 默认值分别为1和0

array是什么数据结构?
array中的数据如何变化?

```
.data
array: .fill 12
.text
.globl main
main:
    movl $0, %ebx
    movl $1, %eax
    movl $1, array(%ebx, %eax, 4)
    incl %eax
    movl $1, array(%ebx, %eax, 4)
    movl $array, %ebx
loop:
    movl $1, %eax
    movl (%ebx, %eax, 4), %ecx
    movl %ecx, array
    incl %eax
    movl (%ebx, %eax, 4), %edx
    movl %edx, 0x4(%ebx)
    addl %ecx, %edx
    movl %edx, (%ebx, %eax, 4)
    jmp loop
```

array
↓
内存中连续12个字节的0

movl array, %ebx

把这句换成 会在 这句出段错误, 为什么?

目录

- 从高级语言到机器指令 - C语言程序编译的过程
- 直接用汇编语言写程序
- 尝试用gdb调试程序



使用gdb调试程序

- 安装gdb调试器: `sudo apt-get install gdb`
- 调试某程序: `gdb <程序名>`
- 常用命令: 见后页



gdb常用命令

<https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>

命令	说明
r	运行程序
c	(触发断点后) 继续运行
si [N]	单步执行N条指令，默认为1
x /nfu <地址>	查看<地址>处内存，打印n个单位，每个单位长度为u，打印值的类型为f，如，x /12ub <地址> f的取值: c d f o s u x ... u的取值: b: Byte h: Half-word (two bytes) w: Word (four bytes) g: Giant word (eight bytes)
info	打印信息，如，info r打印寄存器信息
b <地址>	设置断点 breakpoint
w <地址>	设置监视点 watchpoint
bt	打印栈帧链

用gdb来调试小练习2

保存为fib.S \$ gcc -o fib fib.S 或 gcc -g -o fib fib.S

```
.data
array: .fill 12
.text
.globl main
main:
    movl $0, %ebx
    movl $1, %eax
    movl $1, array(%ebx, %eax, 4)
    incl %eax
    movl $1, array(%ebx, %eax, 4)
    movl $array, %ebx
loop:
    movl $1, %eax
    movl (%ebx, %eax, 4), %ecx
    movl %ecx, array
    incl %eax
    movl (%ebx, %eax, 4), %edx
    movl %edx, 0x4(%ebx)
    addl %ecx, %edx
    movl %edx, (%ebx, %eax, 4)
    jmp loop
```

\$ gdb fib

(gdb) b main # 在main处设置断点

(gdb) b loop # 在loop处设置断点

(gdb) r # run 并触发断点

(gdb) x /3uw &array # 打印内存首地址&array

3个无符号整型 (u)

每个单元长度为32位 (w)

(gdb) c # continue 并触发下一个断点

(gdb) x /3uw &array

需要加取地址符是因为汇编代码中没有声明array的类型，C语言数组类型无需&

熟练掌握gdb对做PA的好处

- 用gdb调试nemu
 - 当遇到assertion fail或者segmentation fault的时候

```
$ cd pa2020_spring/  
$ gdb ./nemu/nemu  
(gdb) run --test-alu adc  
执行和出错信息  
(gdb) bt  
打印栈帧链帮助定位bug和相应的  
函数调用参数  
(gdb) q
```

nemu执行参数,
参见Makefile

在run前使用命令可以在
执行前打断点:

(gdb) b main

熟练掌握gdb对做PA的好处

- 用gdb调试nemu
 - 当遇到assertion fail或者segmentation fault的时候

```
$ cd pa2020_spring/  
$ gdb ./nemu/nemu  
(gdb) run --test-alu adc
```

等同，参见Makefile中的debug目标

执行和出错信息

```
(gdb) bt
```

打印栈帧链帮助定位bug和相应的函数调用参数

```
(gdb) q
```

```
$ gdb -ex=run --args ./nemu/nemu --test-alu adc
```

修改参数后，使用make debug进入gdb调试

nemu内建的调试器 (PA 2-3完善)

- 命令和gdb类似，由nemu内部的monitor实现，可以实现对测试用例（testcase）的单步执行、打印机器状态等功能
- 到了PA 2-1阶段
 - 修改根目录Makefile中run目标的对应规则后使用 `make run` 命令
 - 或根目录执行 `./nemu/nemu --testcase <测试用例名>` 来进入调试界面

gdb与nemu内建调试器monitor区别和联系

模拟CPU
解释执行

testcase/bin/add

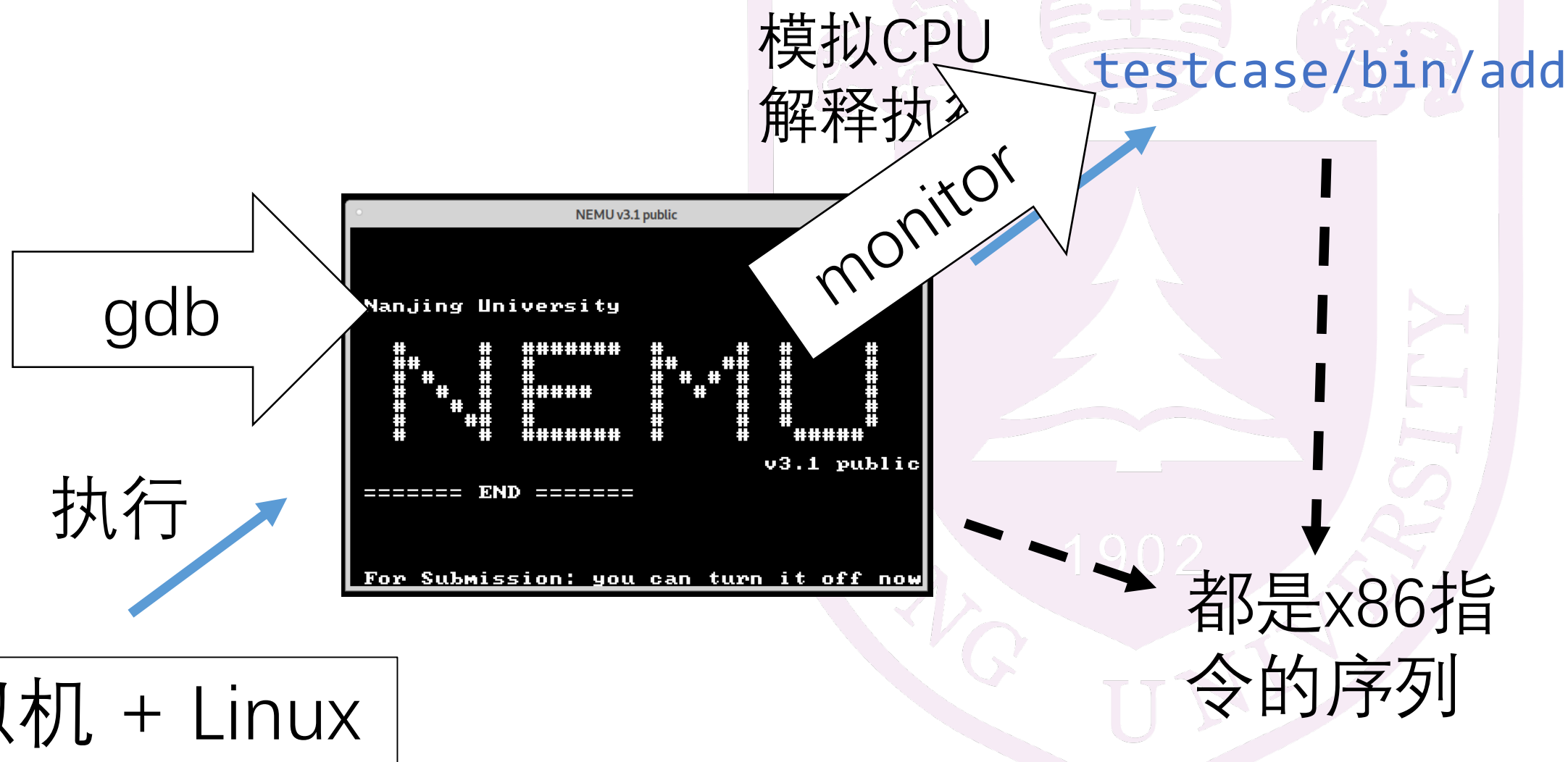
执行



虚拟机 + Linux

都是x86指令的序列

gdb与nemu内建调试器monitor区别和联系





PA 2-0 结束