

计算机系统基础  
Programming Assignment

# PA 2-3 内建调试器和表达式求值

2022年04月08日

南京大学《计算机系统基础》课程组

# 目录

- nemu启动到进入monitor的流程
- monitor的表达式求值功能
  - 第一步：识别表达式中的各个单元
  - 第二步：表达式求值
- 符号表解析（对符号进行求值）



# NEMU的启动过程（编译完成后）

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```

xxx是测试用例的名字，对应testcase/src/文件夹下的一个测试用例，编译后得到testcase/bin/xxx和testcase/bin/xxx.img两个文件

# NEMU的启动过程

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```

  
`int main(int argc, char *argv[])`

`nemu/src/main.c`

# NEMU的启动过程

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```

```
int main(int argc, char *argv[])
```



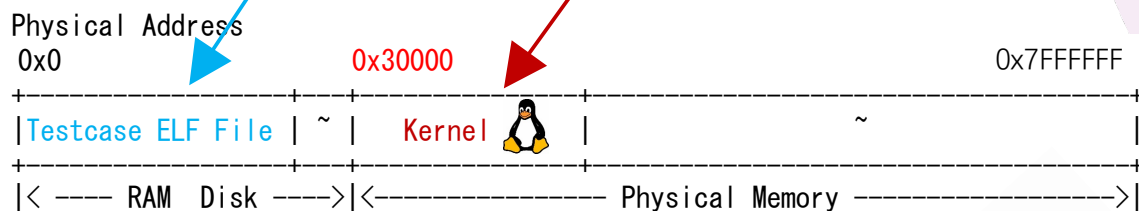
```
static void single_run( const char *img_file_path, const char *elf_file_path)
```

testcase/bin/xxx

kernel/kernel.img

nemu/src/main.c

直接拷贝



带有RAM Disk时的NEMU模拟内存划分方式



# NEMU的启动过程

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```

```
int main(int argc, char *argv[])
```



```
static void single_run( const char *img_file_path, const char *elf_file_path)
```



```
void ui_mainloop(bool autorun)
```

有--autorun为true, 否则为false

[nemu/src/main.c](#)

[nemu/src/monitor/ui.c](#)

# NEMU的启动过程

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```

```
int main(int argc, char *argv[])
```



```
static void single_run( con
```



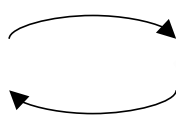
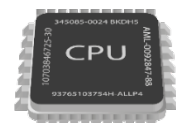
```
void ui_mainloop(bool autorun
```



```
cmd_handler(cmd_c)
```



```
void exec(uint32_t n)
```



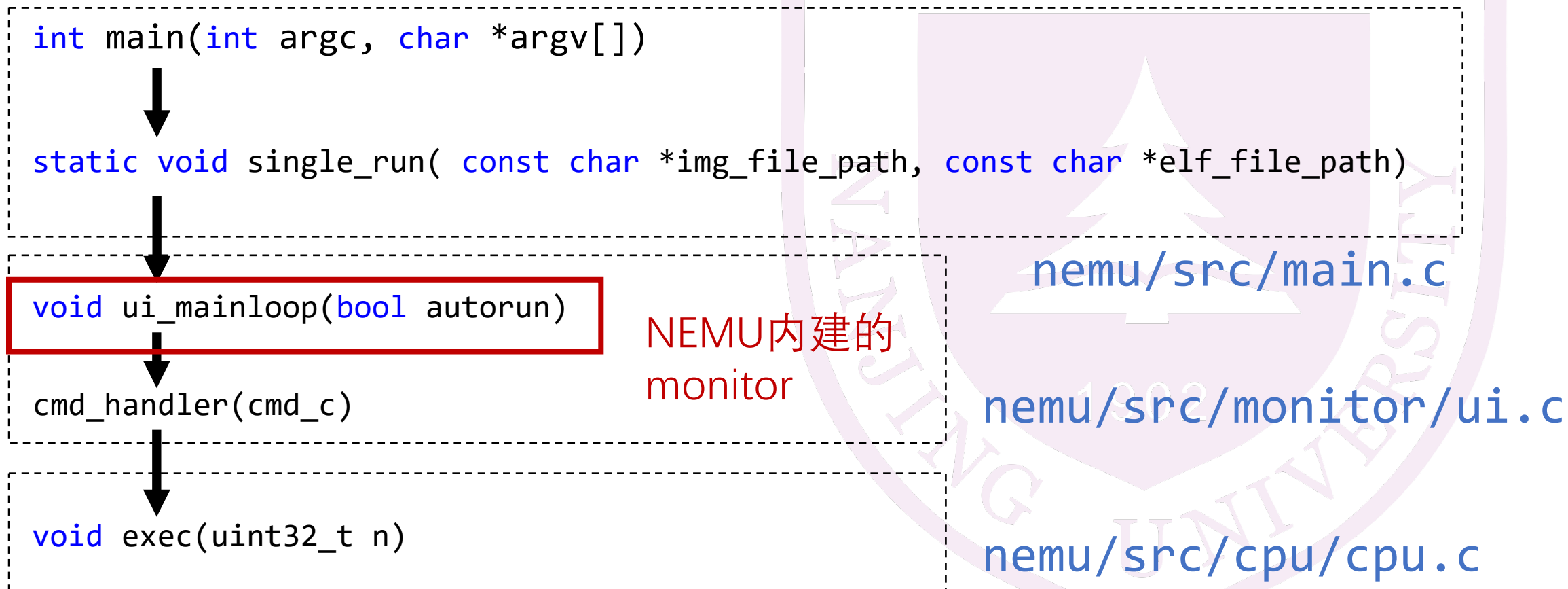
## PA 2-2 以后

1. 先执行内存0x30000处kernel的代码
2. 由kernel完成对testcase ELF文件的装载
3. 跳转到testcase继续执行

[nemu/src/cpu/cpu.c](#)

# NEMU的启动过程

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```





# 内建调试器monitor

- monitor是NEMU内建的基于字符串界面（CLI）的调试器
- 进入CLI调试界面的标志

```
Execute ./kernel/kernel.img ./testcase/bin/mov-c  
hit breakpoint at eip = 0x00030000  
(nemu)
```

在控制台中看到(nemu)这个提示符，光标在后面一闪一闪的，那就进入CLI调试状态了

# 内建调试器monitor

进入CLI调试界面的**两种方法**:

1. 执行参数中不带 `--autorun` 初始化后就等待用户命令

```
$ ./nemu/nemu --autorun --kernel --testcase xxx
```

2. 在kernel或者testcase的代码中加入BREAK\_POINT, 执行到相应位置再进入CLI调试界面

testcase/src/xxx.c

```
int main()  
{  
    1902  
    ...  
    BREAK_POINT;  
    ...  
    return 0;  
}
```

# 内建调试器monitor

Execute `./kernel/kernel.img ./testcase/bin/mov-c`  
hit breakpoint at `eip = 0x00030000`  
(nemu)

命令	格式	使用举例	说明
帮助	help	help	打印帮助信息
继续运行	c	c	继续运行被暂停的程序
退出	q	q	退出当前正在运行的程序
单步执行	si [N]	si 10	单步执行N条指令，N缺省为1
打印程序状态	info <r/w>	info r info w	打印寄存器状态 打印监视点信息
表达式求值 *	p EXPR	p \$eax + 1	求出表达式EXPR的值（EXPR中可以出现数字，0x开头的十六进制数字，\$开头的寄存器，*开头的指针解引用，括号对，和算术运算符）
扫描内存 *	x N EXPR	x 10 0x10000	以表达式EXPR的值为起始地址，以十六进制形式连续输出N个4字节
设置监视点 *	w EXPR	w *0x2000	当表达式EXPR的值发生变化时，暂停程序运行
设置断点 *	b EXPR	b main	在EXPR处设置断点。除此以外，框架代码还提供了宏BREAK_POINT，可以插入到用户程序中，起到断点的作用
删除监视点或断点	d N	d 2	删除第N号监视点或断点

monitor提  
供了好多调  
试的功能

# monitor解析并执行用户命令

```
Execute ./kernel/kernel.img ./testcase/bin/mov-c  
hit breakpoint at eip = 0x00030000  
(nemu) si 10
```

# monitor解析并执行用户命令

Execute ./kernel/kernel.img ./testcase/bin/mov-c

hit breakpoint at eip = 0x00030000

(nemu) si 10

```
void ui_mainloop(bool autorun)
```

```
{
```

```
...
```

```
while (true)
```

```
{
```

**读入用户命令字符串**

分解为命令、参数两个部分

查找cmd\_table并执行相应handler

```
}
```

```
}
```

# monitor解析并执行用户命令

Execute ./kernel/kernel.img ./testcase/bin/mov-c  
hit breakpoint at eip = 0x00030000  
(nemu) si 10

```
void ui_mainloop(bool autorun)
{
```

```
...
```

```
while (true)
```

```
{
```

读入用户命令字符串

分解为命令、参数两个部分

查找cmd\_table并执行相应handler

```
}
```

```
}
```

# monitor解析并执行用户命令

Execute ./kernel/kernel\_image /testcase/bin/monitor  
hit breakpoint at eip 00000000  
(nemu) si 10

```
void ui_mainloop(bool autorun)
{
    ...

    while (true)
    {
        ...
    }
}
```

读入用户命令字符串  
分解为命令、参数两个部分

查找cmd\_table并执行相应handler

```
static struct
{
    char *name;
    char *description;
    int (*handler)(char *);
} cmd_table[] = {
    ...
    {"c", "Continue the execution of the program", cmd_c},
    {"q", "Exit NEMU", cmd_q},
    {"p", "Evaluate an expression", cmd_p},
    {"si", "Single Step Execution", cmd_si},
    ...
};
```

# monitor解析并执行用户命令

Execute ./kernel/kernel  
hit breakpoint at eip  
(nemu) si 10

```
void ui_mainloop(bool autorun)
{
    ...
    while (true)
    {
        ...
    }
}
```

读入用户命令字符串  
分解为命令、参数两个部分  
**查找cmd\_table并执行相应handler**

```
static struct
{
    char *name;
    char *description;
    int (*handler)(char *);
} cmd_table[] = {
    ...
    {"c", "Continue the execution of the program", cmd_c},
    {"q", "Exit NEMU", cmd_q},
    {"p", "Evaluate an expression", cmd_p},
    {"si", "Single Step Execution", cmd_si},
    ...
};

#define cmd_handler(cmd) static int cmd(char *args)
cmd_handler(cmd_si)
{
    ...
}
```



# print asm() 展示单步执行效果

```
make_instr_func(mov_srm82r_v) {  
    int len = 1;  
    OPERAND r, rm;  
    r.data_size = data_size;  
    rm.data_size = 8;  
    len += modrm_r_rm(eip + 1, &r, &rm);  
  
    operand_read(&rm);  
    r.val = sign_ext(rm.val, 8);  
    operand_write(&r);  
    print_asm_2("mov", "", len, &rm, &r);  
    return len;  
}
```

[nemu/include/cpu/instr\\_helper.h](#)

```
void print_asm_0(char *instr, char *suffix, uint8_t len);  
void print_asm_1(char *instr, char *suffix, uint8_t len, OPERAND *opr_1);  
void print_asm_2(char *instr, char *suffix, uint8_t len, OPERAND *opr_1, OPERAND *opr_2);  
void print_asm_3(char *instr, char *suffix, uint8_t len, OPERAND *opr_1, OPERAND *opr_2, OPERAND *opr_3);
```

# monitor解析并执行用户命令

Execute `./kernel/kernel_image /testcase/bin/monitor`

hit breakpoint at eip

(nemu) c

```
void ui_mainloop(bool autorun)
```

```
{
```

```
...
```

```
while (true)
```

```
{
```

读入用户命令字符串

分解为命令、参数两个部分

查找cmd\_table并执行相应handler

```
}
```

```
}
```

```
static struct
```

```
{
```

```
char *name;
```

```
char *description;
```

```
int (*handler)(char *);
```

```
} cmd_table[] = {
```

```
...
```

```
{"c", "Continue the execution of the program", cmd_c},
```

```
{"q", "Exit NEMU", cmd_q},
```

```
{"p", "Evaluate an expression", cmd_p},
```

```
{ #define cmd_handler(cmd) static int cmd(char *args)
```

```
...
```

```
cmd_handler(cmd_c)
```

```
{
```

```
// execute the program
```

```
exec(-1);
```

```
return 0;
```

```
}
```

# 目录

- nemu启动到进入monitor的流程
- monitor的表达式求值功能
  - 第一步：识别表达式中的各个单元
  - 第二步：表达式求值
- 符号表解析（对符号进行求值）



# monitor与表达式求值

```
static struct
{
    char *name;
    char *description;
    int (*handler)(char *);
} cmd_table[] = {
    ...
    {"c", "Continue the execution of the program", cmd_c},
    {"q", "Exit NEMU", cmd_q},
    {"p", "Evaluate an expression", cmd_p},
    {"si", "Single Step Execution", cmd_si},
    ...
};
```

典型

(nemu) p 1+1  
2

# monitor与表达式求值

- 表达式求值用于完善monitor功能

影响到这四个调试命令

命令	格式	使用举例	说明
帮助	help	help	打印帮助信息
继续运行	c	c	继续运行被暂停的程序
退出	q	q	退出当前正在运行的程序
单步执行	si [N]	si 10	单步执行N条指令，N缺省为1
打印程序状态	info <r/w>	info r info w	打印寄存器状态 打印监视点信息
表达式求值 *	p <b>EXPR</b>	p \$eax + 1	求出表达式EXPR的值（EXPR中可以出现数字，0x开头的十六进制数字，\$开头的寄存器，*开头的指针解引用，括号对，和算术运算符）
扫描内存 *	x N <b>EXPR</b>	x 10 0x10000	以表达式EXPR的值为起始地址，以十六进制形式连续输出N个4字节
设置监视点 *	w <b>EXPR</b>	w *0x2000	当表达式EXPR的值发生变化时，暂停程序运行
设置断点 *	b <b>EXPR</b>	b main	在EXPR处设置断点。除此以外，框架代码还提供了宏BREAK_POINT，可以插入到用户程序中，起到断点的作用
删除监视点或断点	d N	d 2	删除第N号监视点或断点

# monitor与表达式求值


- 表达式求值的功用（举两个例子）
  - 例一：查看add测试用例中，test\_data数组的取值
  - 例二：遇到指令`mov 0x40(%edx,%eax,4),%eax`，到底  
0x40(%edx,%eax,4)取值是多少？

# monitor与表达式求值

- 例一：查看add测试用例中，test\_data数组的取值
  - 没有实现表达式求值怎么办？

- 第一步：readelf -s testcase/bin/add, 找到对应的Value值

```
Symbol table '.symtab' contains 23 entries:
  Num:      Value      Size Type      Bind      Vis      Ndx Name
  ...
  22: 00032020      32 OBJECT  GLOBAL DEFAULT      4 test_data
```



- 第二步：(nemu) x 4 0x32020 如果没有实现简单的数字解析，这一步也做不到

```
(nemu) x 4 0x32020
n = 4, expr = 0x32020
0x00032020: 0x00000000 0x00000001 0x00000002 0x7fffffff
```

# monitor与表达式求值

- 例一：查看add测试用例中，test\_data数组的取值
  - 实现了表达式求值怎么办？

- 第一步：(nemu) x 4 test\_data + 4

```
(nemu) x 4 test_data + 4
n = 4, expr = test_data
0x00032024:    0x00000001 0x00000002 0x7fffffff ...
```

完成！很方便！

注意，要完成对test\_data的翻译，不仅仅要实现表达式求值功能，还要有符号表解析功能。



# monitor与表达式求值

- 例二：遇到指令 `mov 0x40(%edx, %eax, 4), %eax`，到底 `0x40(%edx, %eax, 4)` 取值是多少？

- 没有实现表达式求值怎么办？

- 第一步：si单步执行到这一条指令之前
- 第二步：(nemu) info r
- 第三步：掏出纸笔开始算

$$\begin{aligned} & \%edx + \%eax * 4 + 0x40 \\ = & 0x32000 + 0x0 * 4 + 0x40 \\ = & \textcolor{red}{0x32040} \end{aligned}$$

eax	0x00000000
ecx	0x00000001
edx	0x00032000
ebx	0x00000000
esp	0x07ffffd8
ebp	0x07ffffec
esi	0x00000000
edi	0x00000000
eip	0x00030050

# monitor与表达式求值

- 例二：遇到指令mov  
0x40(%edx,%eax,4),%eax, 到底  
0x40(%edx,%eax,4)取值是多少？
  - 实现了表达式求值怎么办？
    - 第一步：si单步执行到这一条指令之前
    - 第二步：(nemu) p \$edx + \$eax \* 4 + 0x40

(nemu) p \$edx + \$eax \* 4 + 0x40

204864

等于十六进制 0x32040

完成！很方便！很强大！

# monitor与表达式求值

- 框架代码是如何使用表达式求值功能的？

- 以p命令为例

nemu/src/monitor/ui.c

```
cmd_handler(cmd_p) {  
    ...  
    bool success;           调用expr()函数进行表达式求值  
    uint32_t val = expr(args, &success);  
    if(!success) {  
        printf("invalid expression: '%s'\n", args);  
    }                        表达式求值失败的话输出表达式  
    else {  
        printf("%d\n", val);  
    }                        表达式求值成功的话输出求值结果  
    ...  
}
```

成功案例

```
(nemu) p $edx + $eax * 4 + 0x40  
204864
```

失败案例

```
(nemu) p hahaha  
invalid expression: 'hahaha'
```

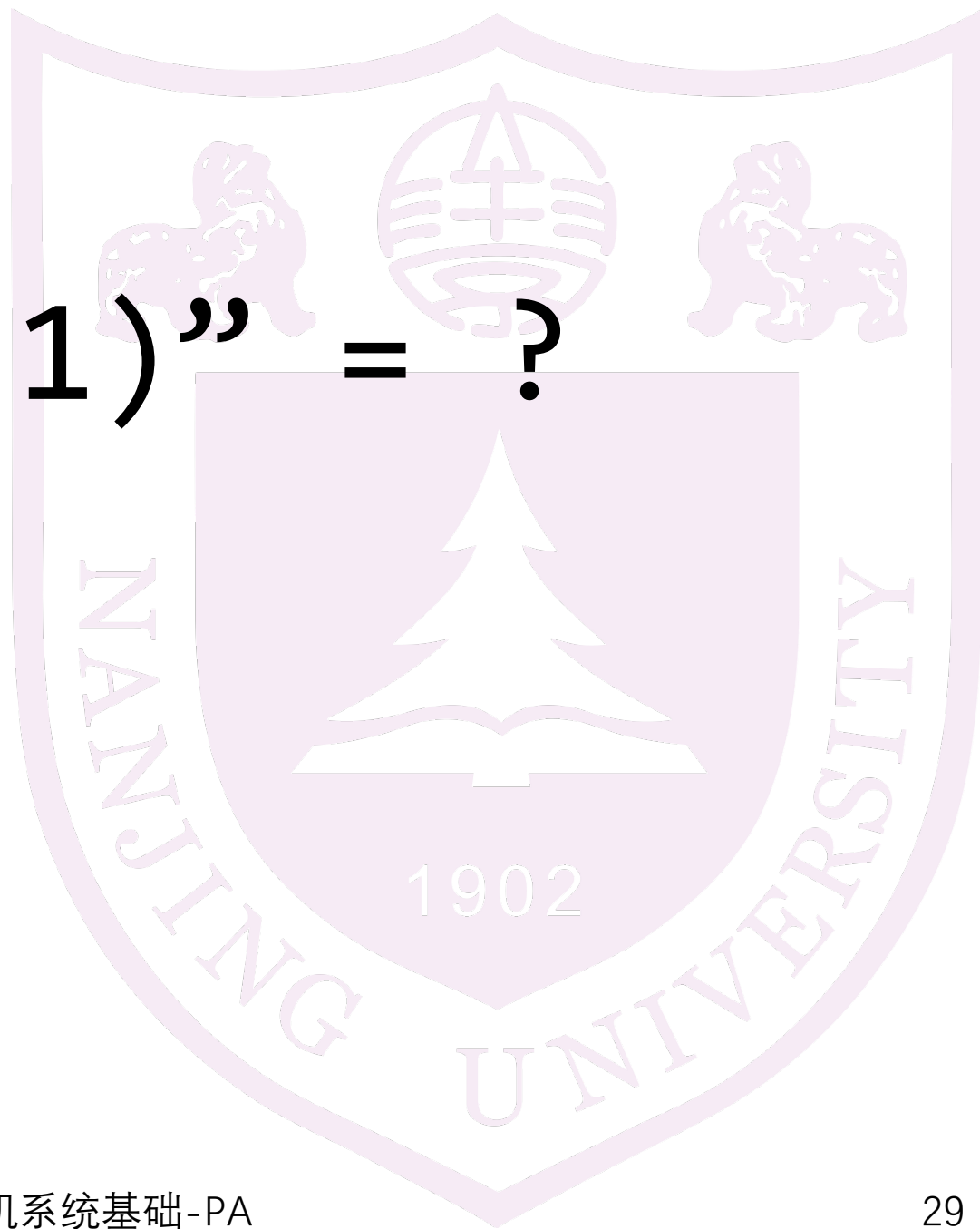
实现表达式求值就是要实现这个函数!

`uint32_t expr(char *e, bool *success)`

- 表达式求值函数原型
- 位于 `nemu/src/monitor/expr.c`
- 两个参数
  - `char *e` 是输入的表达式字符串
  - `bool *success` 用于返回求值是否成功
- `uint32_t` 返回值是求值的结果

# 要解决的问题

$$\text{“}4+3*(2-1)\text{”} = ?$$



# 要解决的问题

$$\text{“}4+3*(2-1)\text{”} = ?$$

第一步：分割字符串，识别  
其中每一个部分的类型

# 要解决的问题

“4+3\*(2-1)” = ?

数字

第一步：分割字符串，识别其中每一个部分的类型

# 要解决的问题

“4+3\*(2-1)” = ?

运算符



第一步：分割字符串，识别其中每一个部分的类型



# 要解决的问题

“ $4+3*(2-1)$ ” = ?

括号

第一步：分割字符串，识别其中每一个部分的类型

# 要解决的问题

“4+3\*(2-1)” = ?

空格

第一步：分割字符串，识别其中每一个部分的类型

# 要解决的问题

$$\text{“}4+3*(2-1)\text{”} = 7$$

第一步：分割字符串，识别  
其中每一个部分的类型

第二步：在识别出类型的基础  
上，运用运算规则，计算结果

# expr()执行的基本流程

- 在expr() 能够被执行之前
  - 在nemu/src/main.c的restart() 函数中
  - 调用了init\_regex(); // 定义在nemu/src/monitor/expr.c
  - 用于初始化正则表达式
- 在p, x, b, w调试命令中使用表达式时, 调用expr() 执行
  - 第一步:

利用初始化好的正则表达式去匹配字符串e, 进行词法分析, 将字符串转换成拥有特定类型的单元序列

```
uint32_t expr(char *e, bool *success) {  
    if(!make_token(e)) {  
        *success = false;  
        return 0;  
    }  
  
    printf("\nPlease implement expr at expr.c\n");  
    assert(0);  
  
    return 0;  
}
```

# expr()执行的基本流程

- 在p, x, b, w调试命令中使用表达式时, 调用expr()执行
  - 第二步:

将这一段替换成对expr.c中eval()函数的调用, 在第一步词法分析结果的基础上进行语法分析和求值, 并return运算结果

```
uint32_t expr(char *e, bool *success) {  
    if(!make_token(e)) {  
        *success = false;  
        return 0;  
    }  
  
    printf("\nPlease implement expr at expr.c\n");  
    assert(0);  
  
    return 0;  
}
```

# expr()执行的基本流程（总结一下）

- 在p, x, b, w调试命令中使用表达式时，调用expr()执行表达式求值的功能，expr()的实现分两步
  - **第一步**：利用初始化好的正则表达式去匹配字符串e，进行**词法分析**，将字符串转换成拥有特定类型的单元序列
  - **第二步**：在第一步词法分析结果的基础上进行**语法分析和求值**，并return运算结果
- 举个例子
  - 输入字符串e，要求它的值

$4+3*(2-1)$

# 目录

- nemu启动到进入monitor的流程
- monitor的表达式求值功能
  - 第一步：识别表达式中的各个单元（使用正则表达式）
  - 第二步：表达式求值
- 符号表解析（对符号进行求值）



# 数学表达式求值（第一步）

- $4+3*(2-1)$ 
  - 第一步：词法分析
    - 要解决的问题（以英文类比）：看懂每一个字母，认出其中的单词
    - 解决方案：利用正则表达式所刻画的字符组合规律，将整个输入字符串切分成一个又一个具有确定类型的单元 (token)
  - 表达式中有那些类型？
    - 数字：十进制，十六进制， .....
    - 运算符：+，-，\*，/，（，）， .....
    - 符号：test\_case， .....
    - 寄存器：\$eax， \$edx， .....

核心：书写各种类型对应的正则表达式，并在expr()函数中第一步的make\_tokens()中用于匹配发现单元



# 数学表达式求值（第一步）

- 正则表达式：Regular Expression
  - 一个正则表达式是一个用来匹配和搜索文本的字符串
  - 正则表达式在操作系统中得到广泛运用（比如grep就是global regular expression print的缩写）
  - 许多编程语言中都提供对正则表达式的支持
- 正则表达式简介
  - 正则表达式最早在1956年提出，并在1968年在计算机中得到广泛应用[1]
  - 一个正则表达式（或叫一个模式，pattern）用于刻画拥有某一个固定模式的字符串的集合

[1] [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)

# 数学表达式求值（第一步）

- 一个正则表达式由一系列普通字符和元字符（metacharacter）组成
  - 普通字符：字母、数字，采用其字面意思
  - 元字符：拥有特殊含义

看颜色识别例子中的普通字符和元字符

举例：[Bb][Aa][Bb][Yy]可以匹配 Baby, baby, bAby, ... 可以不区分大小写的匹配baby这个单词

# 数学表达式求值（第一步）

- 正则表达式简介
  - 普通字符就不用介绍了
  - 元字符的简要说明POSIX basic and extended [1]

元字符	说明	举例
.	匹配任意单个字符，但在括号中时，表示这一个特殊的字符。	a.c 可以匹配"abc"，"a0c"等 [a.c] 只能匹配"a"或"."或"c"
[ ]	匹配位于括号对中的任意单个字符	[abc] 可以匹配"a"，"b"或"c" [a-z] 可以匹配任意一个从"a"到"z"的小写字母
[^ ]	匹配不在括号对中出现过的单个字符	[^abc] 可以匹配除"a"，"b"和"c"以外的任意单个字符
^	匹配目标字符串或行的开头	^abc 可以匹配在字符串或行开头出现的"abc"
\$	匹配目标字符串或行的结尾	[hc]at\$ 可以匹配在字符串或行末尾出现的"hat"或"cat"

[1] [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)

# 数学表达式求值（第一步）

- 正则表达式简介
  - 普通字符就不用介绍了
  - 元字符的简要说明POSIX basic and extended [1]

元字符	说明	举例
( )	子表达式	(abc) 就是一个表达式abc
*	匹配前面的符号零或多次	ab*c 可以匹配"ac", "abc", "abbc", "abbbc"等
{m,n}	匹配前面的符号最少m次 最多n次, 特殊形式{n}, {n,}, {n}	ab{1,2}c 仅可以匹配"abc"或"abbc"
?	匹配前面的表达式零或一次	ab?c 仅可以匹配"ac"或"abc"
+	匹配前面的表达式一或多次	ab+c 可以匹配"abc", "abbc", "abbbc"等
	选择符号, 选择前一个表达式或后一个表达式	more less 可以匹配"more"或者"less"

[1] [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)

# 数学表达式求值（第一步）

- 正则表达式简介
  - 我们来做一些练习

问题
任意十进制数字（不含进制符号）
任意英文字母构成的变量名？
任意十六进制数字（不含进制符号）
包含11位的十进制数字
以"0x"或"0X"开头的任意十六进制数字

答案

# 数学表达式求值（第一步）

- 正则表达式简介
  - 我们来做一些练习

问题
任意十进制数字（不含进制符号）
任意英文字母构成的变量名？
任意十六进制数字（不含进制符号）
包含11位的十进制数字
以"0x"或"0X"开头的任意十六进制数字

答案

`[0-9]+`

`[a-zA-Z]+`

`[0-9a-fA-F]+`

`[0-9]{11}`

`0[xX][0-9a-fA-F]+`

# 数学表达式求值（第一步）

- 回到这个例子：4+3\*(2-1)
  - 第一步：词法分析
  - 要达到的效果

上面一行表示类型，或定义在expr.c的枚举类型enum中（如NUM），或直接用其ASCII编码值（如'+'）。总之，一个类型对应唯一的一个数值。

4+3\*(2-1)



make\_tokens()词法分析

+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+
	NUM		'+'		NUM		'*'		'('		NUM		'-'		NUM		'2')		'1'									
	"4"				"3"						"2"				"1"													
+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+

下面一行是单元对应的字符串内容，有时需要存储下来以便在第二步分析其取值（数字取其数值，符号取其地址等等）

存储在tokens[] 数组中

第一步)

1)

make\_tokens()词法分析

NUM		'-'		NUM		'2'	
"2"				"1"			

存储在tokens[] 数组中

- 第一步)
- 1)
- make\_tokens()词法分析
- |     |  |     |  |     |  |     |  |
|-----|--|-----|--|-----|--|-----|--|
| NUM |  | '-' |  | NUM |  | '2' |  |
| "2" |  |     |  | "1" |  |     |  |
- 存储在tokens[] 数组中

第一步)

1)

make\_tokens()词法分析

NUM		'-'		NUM		'2'	
"2"				"1"			

存储在tokens[] 数组中

第一步)

1)

make\_tokens()词法分析

NUM		'-'		NUM		'2'	
"2"				"1"			

存储在tokens[] 数组中

第一步)

1)

make\_tokens()词法分析

NUM		'-'		NUM		'2'	
"2"				"1"			

存储在tokens[] 数组中

第一步)

1)

make\_tokens()词法分析

NUM		'-'		NUM		'2'	
"2"				"1"			

存储在tokens[] 数组中

第一步)

1)

make\_tokens()词法分析

NUM		'-'		NUM		'2'	
"2"				"1"			

存储在tokens[] 数组中



# 数学表达式求值（第一步）

- 回到这个例子：4+3\*(2-1)
  - 第一步：词法分析
  - 要达成的效果
  - 怎么办？

4 + 3\*(2-1)



make\_tokens()词法分析

+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+
	NUM		'+'		NUM		'*'		'('		NUM		'-'		NUM		'2')															
	"4"				"3"						"2"				"1"																	
+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+

存储在tokens[] 数组中

# 数学表达式求值（第一步）

- 回到这个例子： $4+3*(2-1)$ 
  - 第一步：词法分析
  - 要达到的效果
  - 怎么办？

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE}, // white space
    {"[0-9]{1,10}", NUM}, // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

扩充这个正则表达式集合，把运算符、函数和全局变量名、寄存器等更多的类型都加进来，具体看教程

## 正则表达式:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
```

2022/4/14

# make\_token()执行流程

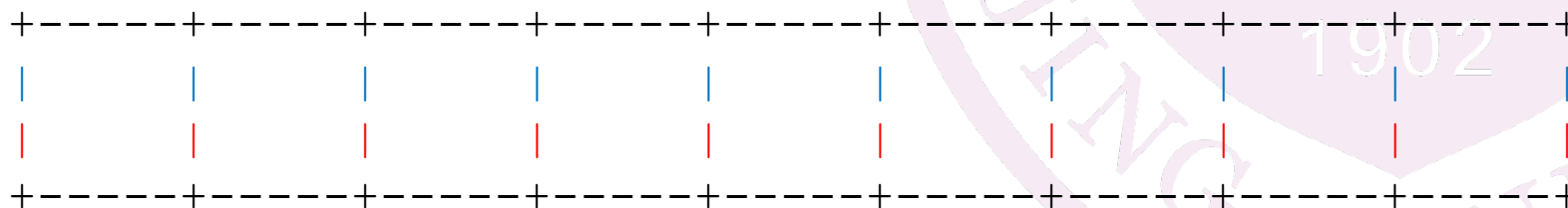
正则表达式:

逐个比对, 看哪个正则表达式正好匹配输入字符串的开头

输入: 4+3\*(2- 1)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},     // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

tokens[]:



# make\_token()执行流程

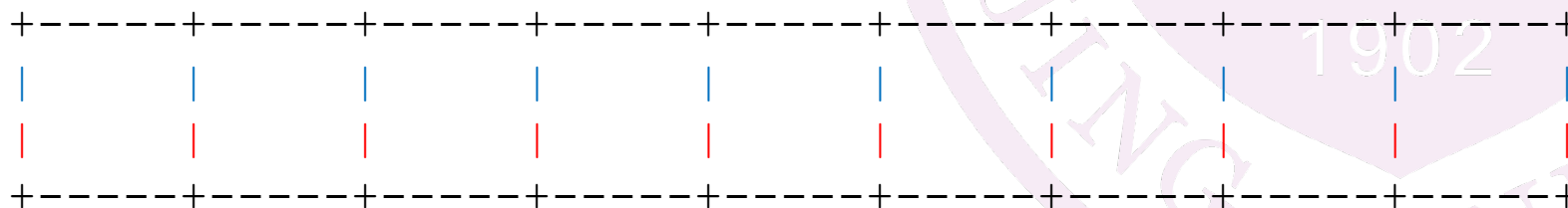
正则表达式:

逐个比对, 看哪个正则表达式正好匹配输入字符串的开头

输入: **4**+3\*(2- 1)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},  // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'},
};
```

tokens[]:



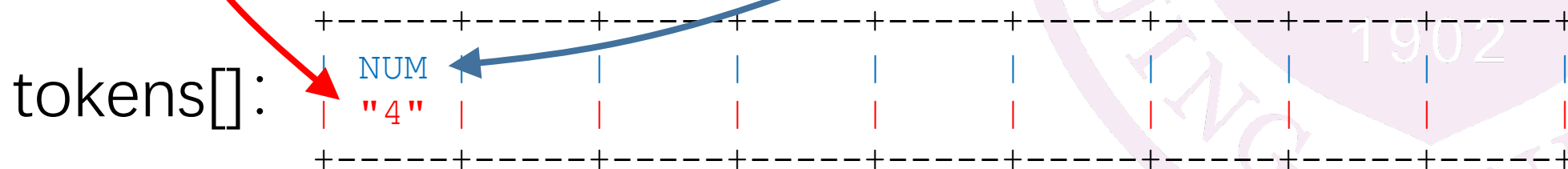
# make\_token()执行流程

正则表达式:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
```

输入: **4**+3\*(2- 1)

记录一个token



## 正则表达式:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

[illegible]

## 正则表达式:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

2022/4/14



# make\_token()执行流程

正则表达式:

输入: **3**\*(2- 1)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'},
};
```

tokens[]:

NUM	'+'	NUM								
"4"		"3"								

# make\_token()执行流程

正则表达式:

输入:  $*(2-1)$

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},     // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

tokens[]:

NUM	'+'	NUM	'*'						
"4"		"3"							

# make\_token()执行流程

正则表达式:

输入: (2- 1)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},     // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

tokens[]:

NUM	'+'	NUM	'*'	'('							
"4"		"3"									

# make\_token()执行流程

正则表达式:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

输入: **2**- 1)

tokens[]:

NUM	'+'	NUM	'*'	'('	NUM	)
"4"		"3"			"2"	

# make\_token()执行流程

## 正则表达式:

输入: - 1)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

tokens[]:	NUM	'+'	NUM	'*'	'('	NUM	'-'	)
	"4"		"3"			"2"		

# make\_token()执行流程

正则表达式:

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE,           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

输入: 1) 空格扔掉

tokens[]:

NUM	'+'	NUM	'*'	'('	NUM	'-'			
"4"		"3"			"2"				

# make\_token()执行流程

正则表达式:

输入: **1**)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

tokens[]:

NUM	'+'	NUM	'*'	'('	NUM	'-'	NUM	)
"4"		"3"			"2"		"1"	

# make\_token()执行流程

## 正则表达式:

输入：)

```
static struct rule {
    char *regex;
    int token_type;
} rules[] = {
    {" +", NOTYPE},           // white space
    {"[0-9]{1,10}", NUM},    // dec
    {"-", '-'},
    {"\\*", '*'},
    {"\\(", '('},
    {"\\)", ')'}
};
```

tokens[]:

NUM	+	NUM	*	(	NUM	-	NUM	)
"4"		"3"			"2"		"1"	



# 数学表达式求值（第一步）

- 回到这个例子：4+3\*(2-1)
  - 第一步：词法分析
  - 要达到的效果

4 + 3 \* ( 2 - 1 )



make\_tokens()词法分析

NUM	'+'	NUM	'*'	'('	NUM	'-'	NUM	)
"4"		"3"			"2"		"1"	

存储在tokens[] 数组中

# expr()函数

NUM	'+'	NUM	'*'	'('	NUM	'-'	NUM	')'
"4"		"3"			"2"		"1"	

```
uint32_t expr(char *e, bool *success) {  
    if(!make_token(e)) {  
        *success = false;  
        return 0;  
    }  
  
    printf("\nPlease implement expr at expr.c\n");  
    assert(0);  
  
    return 0;  
}
```

# expr()函数

NUM	+	NUM	*	(	NUM	-	NUM	)
"4"		"3"			"2"		"1"	

```
uint32_t expr(char *e, bool *success) {
    if(!make_token(e)) {
        *success = false;
        return 0;
    }
}
```

## 调用eval()函数求解tokens数组中对应的表达式值

```
return 结果;
```

}

# 数学表达式求值（第一步plus）

- 有些操作符单凭正则表达式无法准确判断其类型
  - ‘\*’ 可以是乘法，也可以是指针解引用
  - ‘-’ 可以是减法，也可以是取负
- 解决方法：
  - 在`expr()`中调用完`make_tokens()`之后
  - 在`expr()`中调用`eval()`进行求值之前
  - 对`tokens[]`数组再进行一遍扫描
    - 遇到那几个可能有多重含义的操作符
    - 看看前后的`token`类型

举例

NUM - NUM: 左右都是数字，这是减法

啥啥啥 + -NUM: 前面是一个加法符号，后面是个数字，这是负号

# 目录

- nemu启动到进入monitor的流程
- monitor的表达式求值功能
  - 第一步：识别表达式中的各个单元
  - 第二步：表达式求值（在token数组的基础上使用BNF求解）
- 符号表解析（对符号进行求值）

# 数学表达式求值（第二步）

词法分析完了，接下来要实现这个eval()函数来完成求值！

当前表达式求值结果

当前待求值表达式在tokens[]数组中的结束位置

```
uint32_t eval(int s, int e, bool *success)
```

当前待求值表达式在tokens[]数组中的起始位置

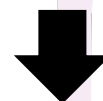
# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - 要解决的问题（以英文类比）：看懂每一个单词，下面要理解整个句子的含义
  - 解决方案：利用BNF所刻画的语法（表达式分解规则），将复杂的表达式先分解到最基本的容易求值的单元，再按照分解的过程，一步步组合回去。

# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - 要达到的效果

$$4 + 3*(2 - 1)$$



第一步

NUM	+	NUM	*	(	NUM	-	NUM	)
"4"		"3"			"2"		"1"	



第二步：eval()给你算出来

$$4 + 3*(2 - 1) = 7$$



# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - 要达成的效果
  - 怎么算？ - 人的话就是按照优先级从高到低一步步算

$$\begin{aligned} & 4+3*(2-1) \\ &= 4+3*1 \\ &= 4+3 \\ &= 7 \end{aligned}$$

当然，在实现这一步时，如果严格用代码来重现纸笔运算的过程，或者采用数据结构课上的中缀转后缀法来计算也没有毛病。这里我们介绍一种更为强大的方法。

# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - 算法怎么写？利用BNF递归求解

<code>&lt;expr&gt; ::= &lt;number&gt;</code>	# 一个数是表达式
<code>  "(" &lt;expr&gt; ")"</code>	# 在表达式两边加个括号也是表达式
<code>  &lt;expr&gt; "+" &lt;expr&gt;</code>	# 两个表达式相加也是表达式
<code>  &lt;expr&gt; "-" &lt;expr&gt;</code>	# 接下来你全懂了
<code>  &lt;expr&gt; "*" &lt;expr&gt;</code>	
<code>  &lt;expr&gt; "/" &lt;expr&gt;</code>	

采用分治法，递归地对表达式进行求值

# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - 算法怎么写？利用BNF递归求解

1

假设已经成功对其中的token进行了识别得到tokens[]数组

先自顶向下利用dominant operator对tokens[]数组进行分解，直至每个<expr>都是单独的token

每一步套用哪条规则进行<expr>的分解？寻找dominant operator，也就是优先级最低的操作。为什么？



# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - 算法怎么写？利用BNF递归求解

1

假设已经成功对其中的token进行了识别得到tokens[]数组

先自顶向下利用dominant operator对tokens[]数组进行分解，直至每个<expr>都是单独的token

每一步套用哪条规则进行<expr>的分解？寻找dominant operator，也就是优先级最低的操作。为什么？

<expr>

"4 + 3\*(2- 1)"

<expr>::= <expr> + <expr>

"4"

+

"3\*(2- 1)"

<expr>::= <expr> \* <expr>

"4"

+

"3"

\*

"(2- 1)"

<expr>::= <expr> - <expr>

"4"

+

"3"

\*

"2"

"1"

2

再自底向上按照分解次序对<expr>求值，利用单独token在第一步词法分析中提取的str域（比如"2"和"1"，或者"test\_data"，或者"\$eax"）来求值很简单吧，结合token类型和str进行合法性检查也简单吧，此基础上往上一层"(2-1)"也就简单了吧……回溯直至完成对原始<expr>的求解

# 数学表达式求值（第二步）

- $4+3*(2-1)$ 
  - **第二步**：语法分析求值，实现eval()函数
  - eval()函数的具体写法？
    - 看教程对应章节的样例代码并进行补完

# 数学表达式求值（大总结）

- 在monitor提供的几个命令中被使用
- 代码实现在nemu/src/monitor/expr.c，对外提供的接口是expr()函数
- 实现方案基本分两步走
  - 第一步：利用初始化好的正则表达式去匹配字符串e，进行词法分析，将字符串转换成拥有特定类型的单元序列
    - 第一步plus：对于可能存在歧义的运算符进行特殊处理
  - 第二步：在第一步词法分析结果的基础上进行语法分析和求值，并return运算结果

# 数学表达式求值（大总结）

```
uint32_t expr(char *e, bool *success) {  
    if(!make_token(e)) {  
        *success = false;  
        return 0;  
    }  
  
    // 第一步plus: 对可能产生多义的运算符进行进一步确认类型。  
    // 实现要点: 扫描tokens[]数组, 根据嫌疑运算符前后的符号  
    // 类型进一步明确其含义。  
  
    return eval(?, ?, success);  
}
```

// 第一步: 词法分析, 得到tokens[]数组。  
// 实现要点: 写一堆正则表达式。

// 第二步: 语法分析并求值, 得到运算结果。  
// 实现要点: 自己想好一堆BNF (教程基本都给了), 先自顶向下  
// 利用dominant operator对整个tokens[]数组所代  
// 表的表达式进行分解。再自底向上求解整个表达式的值。

# 目录

- nemu启动到进入monitor的流程
- monitor的表达式求值功能
  - 第一步：识别表达式中的各个单元
  - 第二步：表达式求值
- 符号表解析（对符号进行求值）





testcase/src/add.c

```
int test_data[] = {0, 1, 2, 0x7fffffff, 0x80000000,
0x80000001, 0xfffffffffe, 0xffffffffff};
```

readelf -s testcase/bin/add

Symbol table '.symtab' contains 23 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
22:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	test_data

在x调试命令中使用符号的名称

```
(nemu) x 4 test_data + 4
```

```
0x00032024: 0x00000001 0x00000002 0x7fffffff ...
```

# 符号表的解析

- 要解决的问题：

给定一个符号（如，全局变量）的名字，返回其值

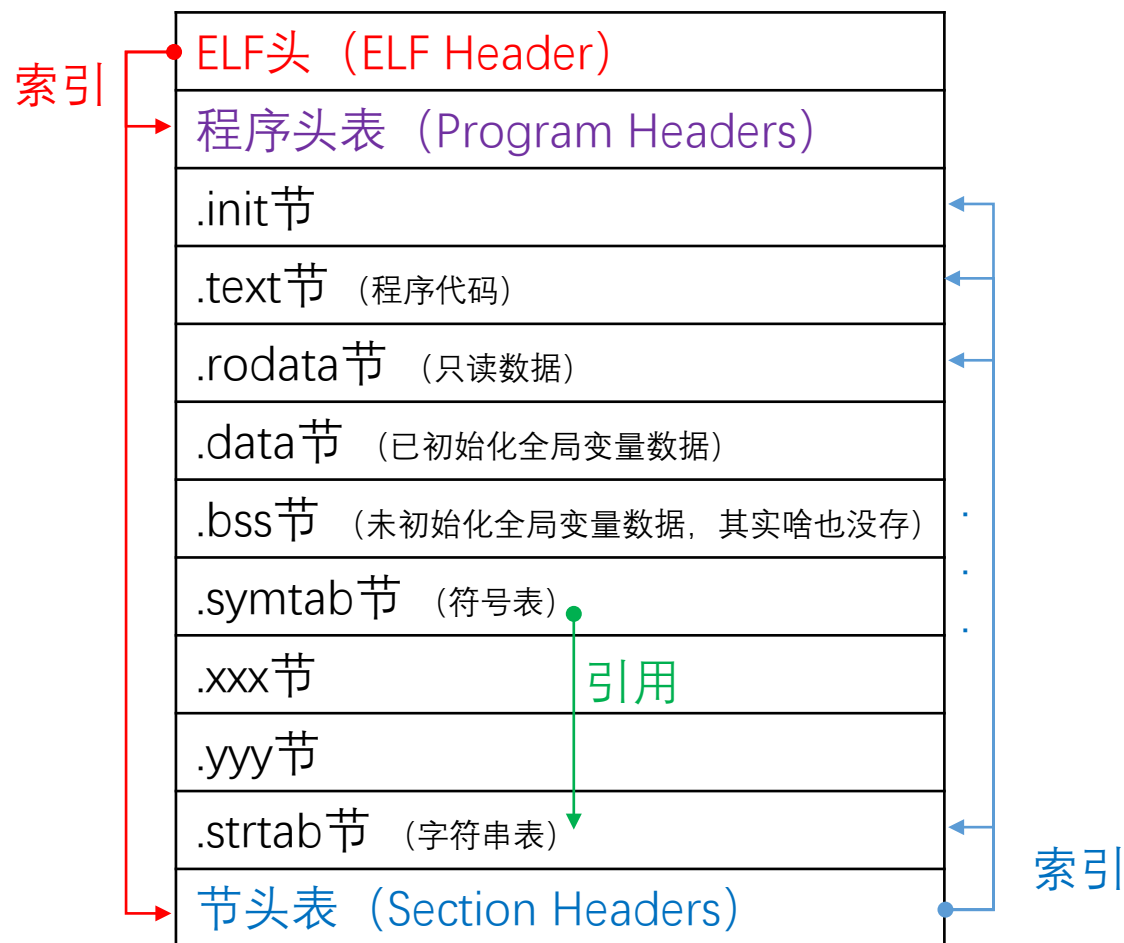
[testcase/src/add.c](#)

```
int test_data[] = {0, 1, 2, 0x7fffffff, 0x80000000, 0x80000001,
0xfffffffffe, 0xffffffffff};
```

NEMU中的交互式调试界面

```
(nemu) x 4 test_data
n = 4, expr = test_data
0x00032020:      0x00000000 0x00000001 0x00000002 0x7fffffff
```

# 符号表的解析

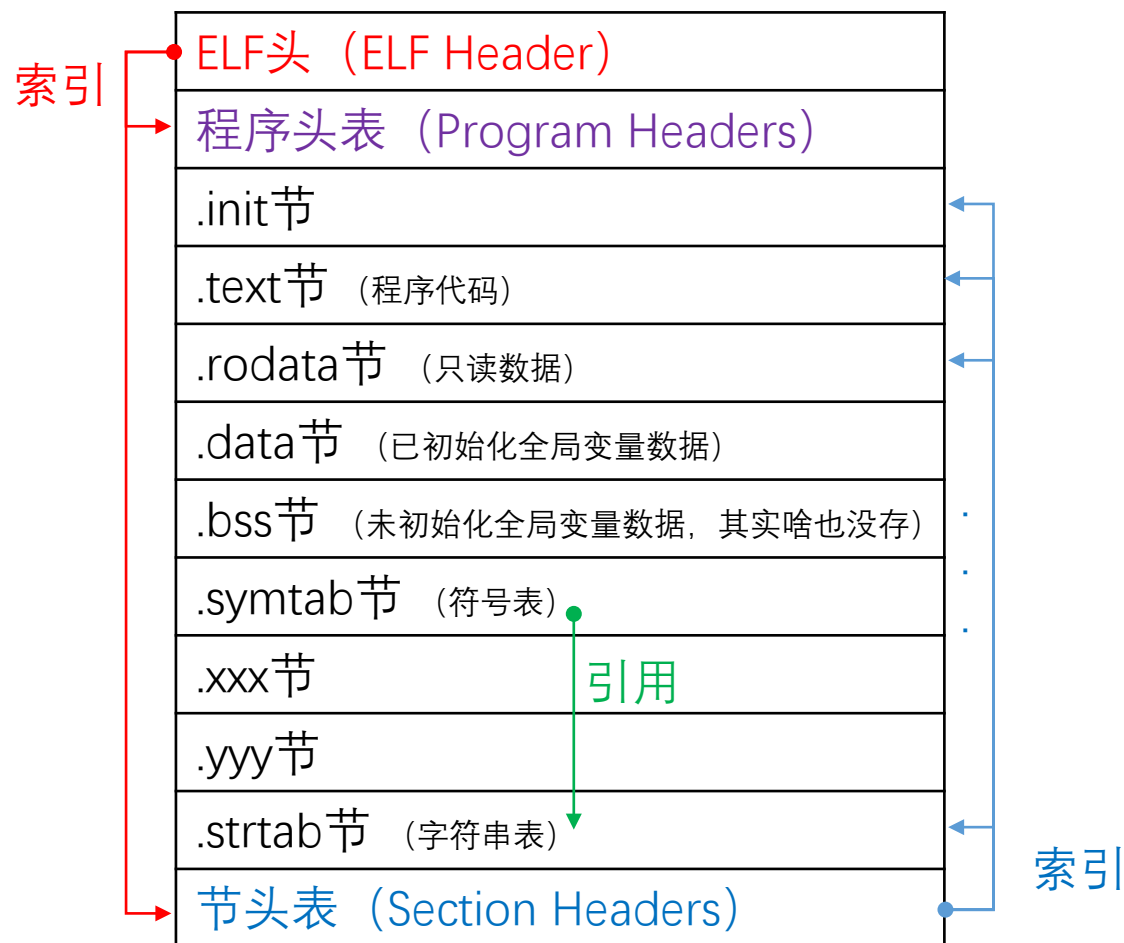


## 符号表解析程序

输入: ELF文件, 带查询符号名  
输出: 符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. for 符号表中的每一项
4.     if 该项名 == 带查询符号名
5.         return 查找成功, 符号的内存地址
6.     end if
7. end for
8. return 查找失败

# 符号表的解析



## 符号表解析程序

输入：ELF文件，带查询符号名  
输出：符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. for 符号表中的每一项
4.     if 该项名 == 带查询符号名
5.         return 查找成功，符号的内存地址
6.     end if
7. end for
8. return 查找失败

# ELF头的编程解析

## ELF Header:

```
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x30000
Start of program headers: 52 (bytes into file)
Start of section headers: 18276 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 3
Size of section headers: 40 (bytes)
Number of section headers: 15
Section header string table index: 14
```

节头表位置

节头表对应字符串表的索引

```
#define EI_NIDENT 16
```

```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    ElfN_Addr e_entry;
    ElfN_Off e_phoff;
    ElfN_Off e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} ElfN_Ehdr;
```

ELF头，它位于整个ELF文件最开始的地方。在32位Linux系统中，<elf.h>头文件中的Elf32\_Ehdr数据结构与之对应。

# 解析节头表，找到符号表和字符串表

找到名为'.symtab'的符号表和名为'.strtab'的（对应节头表的）字符串表

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	00000000	00000000	00	AX	0	0	0
[ 1]	.text	PROGBITS	00030000	001000	0000d0	00	AX	0	0	1
[ 2]	.eh_frame	PROGBITS	000300d0	0010d0	000084	00	A	0	0	4
[ 3]	.got.plt	PROGBITS	00032000	002000	00000c	04	WA	0	0	4
[ 4]	.data	PROGBITS	00032020	002020	000120	00	WA	0	0	32
[ 5]	.comment	PROGBITS	00000000	002140	000026	01	MS	0	0	1
[ 6]	.debug_aranges	PROGBITS	00000000	002168	000040	00		0	0	8
[ 7]	.debug_info	PROGBITS	00000000	0021a8	000142	00		0	0	1
[ 8]	.debug_abbrev	PROGBITS	00000000	0022ea	0000d6	00		0	0	1
[ 9]	.debug_line	PROGBITS	00000000	0023c0	0000dc	00		0	0	1
[10]	.debug_str	PROGBITS	00000000	00249c	001a37	01	MS	0	0	1
[11]	.shstrtab	PROGBITS	00000000	0044d3	00005f9	00		0	0	1
[12]	.symtab	SYMTAB	00000000	0044cc	0000190	10		13	15	4
[13]	.strtab	STRTAB	00000000	00465c	0000078	00		0	0	1
[14]	.shstndx	STRTAB	00000000	0046d4	0000090	00		0	0	1

Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
L (link order), O (extra OS processing required), G (group), T (TLS),  
C (compressed), x (unknown), o (OS specific), E (exclude),  
p (processor specific)

There are no section groups in this file.

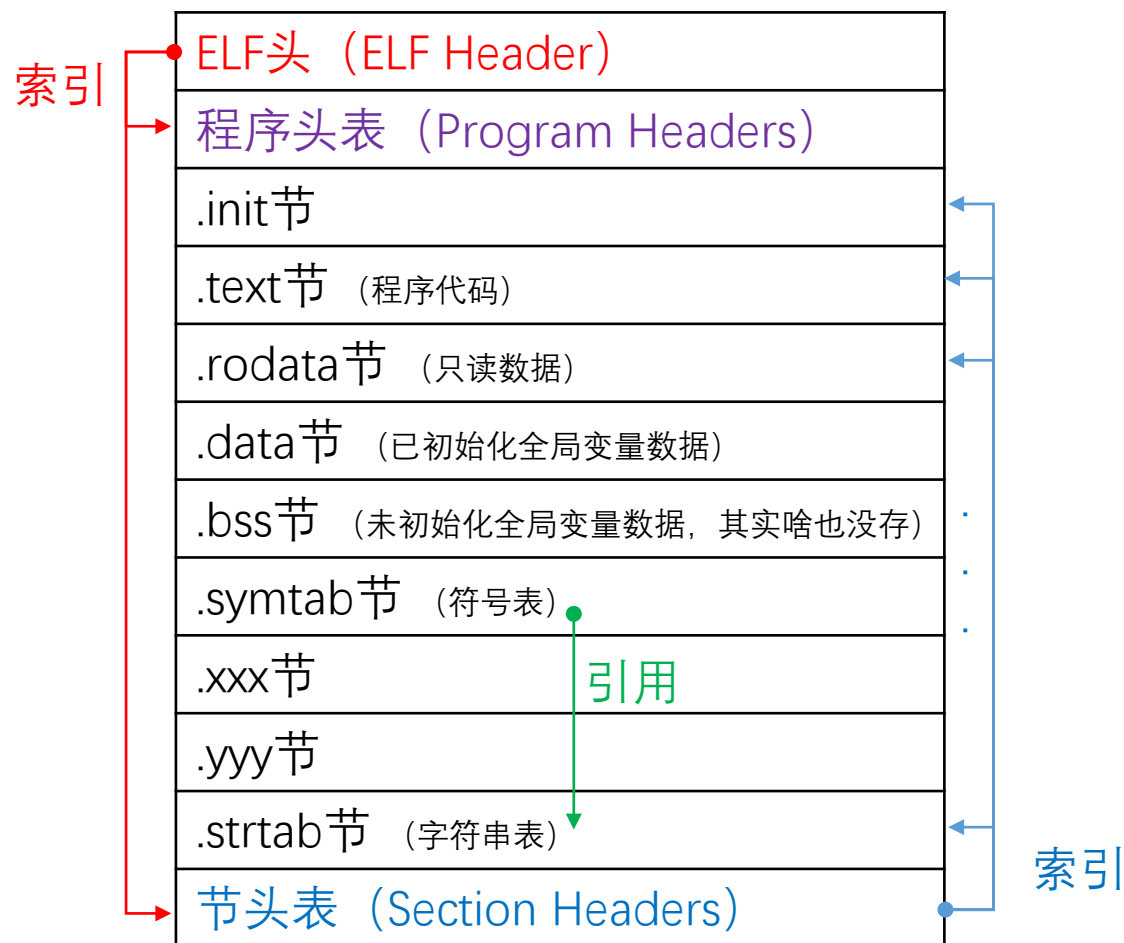
<elf.h>

```
typedef struct {  
    uint32_t    sh_name;  
    uint32_t    sh_type;  
    uint32_t    sh_flags;  
    Elf32_Addr  sh_addr;  
    Elf32_Off   sh_offset;  
    uint32_t    sh_size;  
    uint32_t    sh_link;  
    uint32_t    sh_info;  
    uint32_t    sh_addralign;  
    uint32_t    sh_entsize;  
} Elf32_Shdr;
```

这里的sh\_name只是一个索引值，只有用该索引值去查了.shstrtab之后才能得到.text, .data这样的字符串，而.shstrtab在哪里呢？ELF Header中的e\_shstrndx变量告诉我们，它在Section Headers数组中的第14项

具体技术和符号表+字符串表解析方式一样

# 符号表的解析



## 符号表解析程序

输入：ELF文件，带查询符号名  
输出：符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. **for 符号表中的每一项**
4.     **if 该项名 == 带查询符号名**
5.         **return 查找成功，符号的内存地址**
6.     **end if**
7. **end for**
8. **return 查找失败**

# 符号表的解析

- 通过节头表定位'.symtab'节在ELF文件中的位置
- 符号表也是个数组，其类型为Elf32\_Sym

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[12]	.symtab	SYMTAB	00000000	0044cc	000190	10		13	15	4

<elf.h>

Symbol table '.symtab' contains 25 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00030000	0	SECTION	LOCAL	DEFAULT	1	
2:	000300d0	0	SECTION	LOCAL	DEFAULT	2	
3:	00032000	0	SECTION	LOCAL	DEFAULT	3	
4:	00032020	0	SECTION	LOCAL	DEFAULT	4	
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000	0	SECTION	LOCAL	DEFAULT	6	
7:	00000000	0	SECTION	LOCAL	DEFAULT	7	
8:	00000000	0	SECTION	LOCAL	DEFAULT	8	
9:	00000000	0	SECTION	LOCAL	DEFAULT	9	
10:	00000000	0	SECTION	LOCAL	DEFAULT	10	
11:	00000000	0	SECTION	LOCAL	DEFAULT	11	
12:	00000000	0	FILE	LOCAL	DEFAULT	ABS	add.c
13:	00000000	0	FILE	LOCAL	DEFAULT	ABS	
14:	00032000	0	OBJECT	LOCAL	DEFAULT	3	__GLOBAL_OFFSET_TABLE__
15:	000300c8	0	FUNC	GLOBAL	HIDDEN	1	__x86.get_pc_thunk.ax
16:	00030005	32	FUNC	GLOBAL	DEFAULT	1	add
17:	000300cc	0	FUNC	GLOBAL	HIDDEN	1	__x86.get_pc_thunk.bx
18:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	__bss_start
19:	00030025	163	FUNC	GLOBAL	DEFAULT	1	main
20:	00032040	256	OBJECT	GLOBAL	DEFAULT	4	ans
21:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	_edata
22:	00032140	0	NOTYPE	GLOBAL	DEFAULT	4	_end
23:	00030000	0	NOTYPE	GLOBAL	DEFAULT	1	start
24:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	test_data

```
typedef struct {
    uint32_t    st_name;
    Elf32_Addr  st_value;
    uint32_t    st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t    st_shndx;
} Elf32_Sym;
```

st\_name – 符号名称，对应strtab中的偏移量  
st\_value – 符号的地址  
st\_size – 符号所占用的字节数  
st\_info – 包含了Type信息，man elf查看说明

testcase/src/add.c

```
int test_data[] = {0, 1, 2,
0x7fffffff, 0x80000000,
0x80000001, 0xffffffff,
0xffffffff};
```

readelf -s add



# 符号表的解析

- 符号表(.symtab)与字符串表(.strtab)结合, 获取符号的字符串形式的名称

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[13]	.strtab	STRTAB	00000000	00465c	000078	00		0	0	1

hexdump -C add

00004650	20 20 03 00 20 00 00 00	11 00 04 00 00 61 64 64	.. .....add
00004660	2e 63 00 5f 47 4c 4f 42	41 4c 5f 4f 46 46 53 45	.c._GLOBAL_OFFSE
00004670	54 5f 54 41 42 4c 45 5f	00 5f 5f 78 38 36 2e 67	T_TABLE.__x86.g
00004680	65 74 5f 70 63 5f 74 68	75 6e 6b 2e 61 78 00 61	et_pc_thunk.ax.a
00004690	64 64 00 5f 5f 78 38 36	2e 67 65 74 5f 70 63 5f	dd.__x86.get_pc_
000046a0	74 68 75 6e 6b 2e 62 78	00 5f 5f 62 73 73 5f 73	thunk.bx.__bss_s
000046b0	74 61 72 74 00 6d 61 69	6e 00 61 6e 73 00 5f 65	tart.main.ans._e
000046c0	64 61 74 61 00 5f 65 6e	64 00 74 65 73 74 5f 64	data._end.test_d
000046d0	61 74 61 00 2e 73 79	6d 74 61 62 00 2e 73 74	ata...symtab..st

'\0'

Symbol table '.symtab' contains 25 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
24:	00032020	32	OBJECT	GLOBAL	DEFAULT	4	6d

```
typedef struct {
    uint32_t  st_name;
    Elf32_Addr st_value;
    uint32_t  st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t   st_shndx;
} Elf32_Sym;
```

# 符号表的解析

.strtab在ELF文件中起始位置 + 符号表项.st\_name => 字符串

## 符号表解析程序

输入：ELF文件，带查询符号名  
输出：符号在内存中的地址

1. 读入ELF头
2. 定位符号表
3. for 符号表中的每一项
4. **if 该项名 == 带查询符号名**
5.       return 查找成功，符号的内存地址
6.   end if
7. end for
8. return 查找失败

# NEMU相关代码

[nemu/src/monitor/elf.c](#)

```
uint32_t look_up_symtab(char *sym, bool *success) {
    int i;
    for(i = 0; i < nr_symtab_entry; i++) {
        uint8_t type = ELF32_ST_TYPE(symtab[i].st_info);
        if((type == STT_FUNC || type == STT_OBJECT) &&
            strcmp(strtab + symtab[i].st_name, sym) == 0) {
            *success = true;
            return symtab[i].st_value;
        }
    }

    *success = false;
    return 0;
}
```

# NEMU相关代码

nemu/src/monitor/elf.c

找到名为.symtab的符号表和  
名为.strtab的字符串表

```
/* Load section header table 读取节头表 */
uint32_t sh_size = elf->e_shentsize * elf->e_shnum;
Elf32_Shdr *sh = malloc(sh_size);
fseek(fp, elf->e_shoff, SEEK_SET);
fread(sh, sh_size, 1, fp);

/* Load section header string table 读取节头表对应的字符串表 */
char *shstrtab = malloc(sh[elf->e_shstrndx].sh_size);
fseek(fp, sh[elf->e_shstrndx].sh_offset, SEEK_SET);
fread(shstrtab, sh[elf->e_shstrndx].sh_size, 1, fp);

int i;
for(i = 0; i < elf->e_shnum; i++) { /* 扫描节头表 */
    if(sh[i].sh_type == SHT_SYMTAB && /* 这一步和解析符号名称时的操作一样，等一下细讲
                                     strcmp(shstrtab + sh[i].sh_name, ".symtab") == 0) {
        /* Load symbol table from exec_file 得到符号表 */
        symtab = malloc(sh[i].sh_size);
        fseek(fp, sh[i].sh_offset, SEEK_SET);
        fread(symtab, sh[i].sh_size, 1, fp);
        nr_symtab_entry = sh[i].sh_size / sizeof(symtab[0]);
    }
    else if(sh[i].sh_type == SHT_STRTAB &&
            strcmp(shstrtab + sh[i].sh_name, ".strtab") == 0) {
        /* Load string table from exec_file 得到符号表对应的字符串表 */
        strtab = malloc(sh[i].sh_size);
        fseek(fp, sh[i].sh_offset, SEEK_SET);
        fread(strtab, sh[i].sh_size, 1, fp);
    }
}
```

# 符号表的解析

- 符号表解析了有啥用？
- 如果你想写一个链接器
  - 可以将处于不同.o文件中的全局变量或函数的调用和定义通过内存地址联系到一起
  - `readelf -s nemu/src/cpu/decode/opcode.o`
  - `readelf -s nemu/src/cpu/instr/mov.o`

Symbol table '.symtab' contains 165 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
149:	000002a0	176	FUNC	GLOBAL	DEFAULT	39	mov_i2rm_b
151:	00000350	176	FUNC	GLOBAL	DEFAULT	39	mov_i2rm_v
...							

mov.o

opcode.o

如果发现符号表中有多个Type为FUNC或OBJECT，Bind类型为GLOBAL，其Ndx都显示在某一个section中被定义了的符号具有同样的名字：

multiple definition of xxx

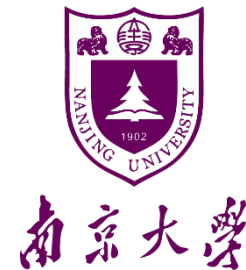
去掉static void instr\_execute\_2op()前面的static就能够触发（比如尝试mov.c和sar.c，把static去掉），观察一下对应的符号表，是不是有什么变化？

Symbol table '.symtab' contains 249 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
223:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	mov_i2r_b
224:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	mov_i2r_v
...							

# 符号表的解析

- 符号表解析了有啥用？
- 对于NEMU来说
  - 你可以使用 `x test_data` 来查看 `test_data` 的起始地址
    - 再使用 `x 起始地址+offset` 来查看 `test_data` 的内容
    - 也可以使用 `x *(test_data + offset)` 来查看 `test_data` 的内容
  - 你也可以使用 `b main` 来在 `main` 函数开始处设置断点
- 在NEMU中使用上述功能涉及对表达式求值功能的实现
  - 相应教程：看教程PA 2-3部分
  - 代码：nemu/src/monitor/expr.c



# PA 2-3 结束

整个PA2授课完成