

计算机系统基础  
Programming Assignment

# PA 2 程序的执行（第三课） ——PA 2-3.1 表达式求值

2018年10月17日


# monitor与表达式求值

- monitor是NEMU中用于调试的功能组件
  - 通过字符命令界面（command line interface, CLI）来提供调试功能
- 运行nemu/nemu后的基本流程
  - 入口为nemu/src/main.c中的main()函数
  - 完成对nemu的必要初始化后
  - 进入nemu/src/monitor/ui.c中的ui\_mainloop()继续执行
  - ui\_mainloop()根据传入的bool型参数的取值，决定是否启动CLI调式模式
- 进入CLI调试模式的两种方式
  - 一：使用make run运行测试用例，但PA 2.2后就失效了
  - 二：在被执行的代码（kernel或者testcase）中插入BREAK\_POINT语句
    - 当nemu执行到BREAK\_POINT时，暂停指令执行，进入CLI调试界面

# monitor与表达式求值

- 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，那就进入了CLI调试状态了

# monitor与表达式求值

- 表达式求值用于完善monitor功能

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

影响到这四个调试命令

monitor提供了好多调试的功能

# 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？

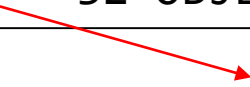
# monitor与表达式求值

- 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

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

注意，要完成对test\_data的翻译，不仅仅要实现本次课所说的表达式求值功能，还要有上次课讲到的符号表解析功能。请结合两次课的内容善自体会。

完成！很方便！

# 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：  $(nemu) \text{ p } \$edx + \$eax * 4 + 0x40$

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

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

# 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

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

# 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）
任意英文单词？
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

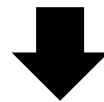
问题	答案
任意十进制数字（不含进制符号）	[0-9]+
任意英文单词？	[a-zA-Z]+
任意十六进制数字（不含进制符号）	[0-9a-fA-F]+
包含11位的十进制数字	[0-9]{11}
以"0x"或"0X"开头的任意十六进制数字	0[xX][0-9a-fA-F]+

# 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
  - 要达成的效果

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	)
"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）

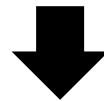
存储在tokens[] 数组中

# 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

```
typedef struct token {  
    int type;  
    char str[32];  
} Token; // 对应数据结构
```

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

# 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	)
"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

这个例子的话添加这几条正则表达式就够了。两次反斜杠啥意思？自己思考一下。实在弄不明白，有大佬知道，群里去求教。

# 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

# 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）

词法分析完了，就是要实现这个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)
```

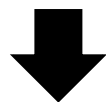
当前待求值表达式在tokens[]数组中的起始位置

勘误：框架代码的eval()函数最后多一个\*success参数？在eval()中的return前根据是否是合法表达式，赋值为true或false。教程等回来以后再做调整。

# 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

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



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？

<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"





# 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？
    - 看教程第52页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。
```