# PA1 第三阶段 实验报告

实验进度：

完成 PA1 第三阶段的实验：实现监视点，即 **w expr** 命令

必答题：

## 1 实现监视点池的管理

监视点池通过以下函数管理：

```
// ******** src/monitor/debug/ui.c
// ****************** static int cmd_w( char * args )
WP * new_wp(){
// ****************** static int cmd_d( char * args )
void free_wp( WP * prev ){
bool delete_wp( int num ){
```

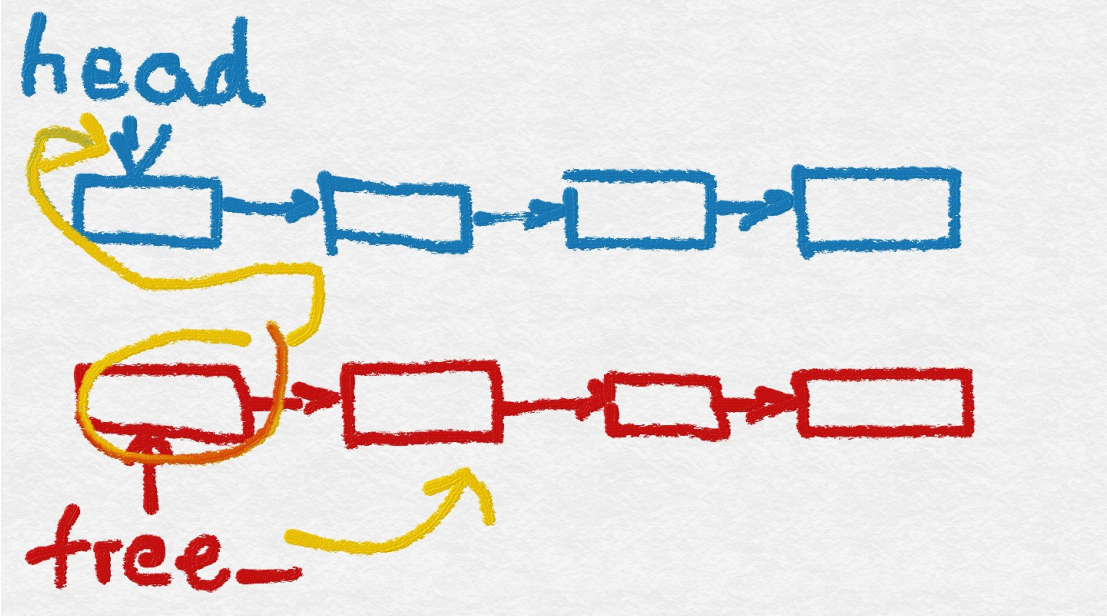
相比于手册中的

**WP* new_wp(); void free_wp(WP *wp);**

增加了函数 **bool delete_wp( int num )**

这个函数可以直接从 ui.c 中调用，避免了 ui.c 中使用 WP * head, WP * free_的不方便。

具体函数的实现就是普通的链表操作：




为了方便，采用了首插入，所以 info w 时可以看到监视点顺序是颠倒的。

## 2 实现监视点

首先说明一下监视点的调用、工作流程：

ui.c 中的 w 命令将监视点添加到链表中，到此为止，不再有任何其他操作

随着程序的进行，不断运行 cpu_exec()，每执行一次，检查一次监视点链表中的监视点有没有变化，如果变化再输出相应内容。

```
WP * new_wp();
static int cmd_w( char * args ){
	bool if_success;
	WP * nwp = new_wp();
	if( strlen( args ) > 32 ){
		printf("String Len Overflow!\n");
		return 0;
	}
	strcpy( nwp->args, args );
	nwp->old_val = expr( args , &if_success);
	return 0;
}
```

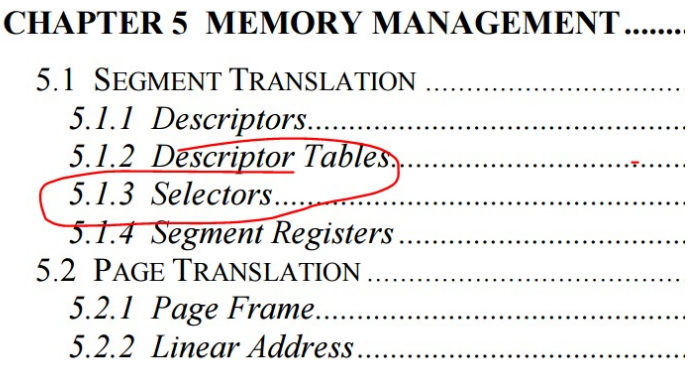
可见 cmd_w 只是单纯地添加监视点。

而在 void cpu_exec(volatile uint32_t n)中，每次执行检查一次监视点即可

```
if ( check_wp() ==  true ){ nemu_state = STOP; }
```

# 3 熟悉 i386 手册

通过目录定位 selector:



CHAPTER 5 MEMORY MANAGEMENT .....

5.1 SEGMENT TRANSLATION .....

*5.1.1 Descriptors*.....

*5.1.2 Descriptor Tables*.....

*5.1.3 Selectors*.....

*5.1.4 Segment Registers* .....

5.2 PAGE TRANSLATION .....

*5.2.1 Page Frame*.....

*5.2.2 Linear Address*.....

- 查阅 i386 手册 理解了科学查阅手册的方法之后, 请你尝试在 i386 手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
  - EFLAGS 寄存器中的 CF 位是什么意思?

**2.3.4.1 Status Flags**

The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction. Refer to Appendix C for definition of each status flag.

  - ModR/M 字节是什么?

**17.2.1 ModR/M and SIB Bytes**

The ModR/M and SIB bytes follow the opcode byte(s) in many of the 80386 instructions. They contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

The ModR/M byte contains three fields of information:

- The mod field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes

  - mov 指令的具体格式是怎么样的?

| Opcode | Instruction | Clocks | Description |
|---|---|---|---|
| 88 /r | MOV r/m8,r8 | 2/2 | Move byte register to r/m byte |
| 89 /r | MOV r/m16,r16 | 2/2 | Move word register to r/m word |
| 89 /r | MOV r/m32,r32 | 2/2 | Move dword register to r/m dword |
| 8A /r | MOV r8,r/m8 | 2/4 | Move r/m byte to byte register |
| 8B /r | MOV r16,r/m16 | 2/4 | Move r/m word to word register |
| 8B /r | MOV r32,r/m32 | 2/4 | Move r/m dword to dword register |
| 8C /r | MOV r/m16,Sreg | 2/2 | Move segment register to r/m word |
| 8D /r | MOV Sreg,r/m16 | 2/5,pm=18/19 | Move r/m word to segment register |
| A0 | MOV AL,moffs8 | 4 | Move byte at (seg:offset) to AL |
| A1 | MOV AX,moffs16 | 4 | Move word at (seg:offset) to AX |
| A1 | MOV EAX,moffs32 | 4 | Move dword at (seg:offset) to EAX |
| A2 | MOV moffs8,AL | 2 | Move AL to (seg:offset) |
| A3 | MOV moffs16,AX | 2 | Move AX to (seg:offset) |
| A3 | MOV moffs32,EAX | 2 | Move EAX to (seg:offset) |
| B0 + rb | MOV reg8,imm8 | 2 | Move immediate byte to register |
| B8 + rw | MOV reg16,imm16 | 2 | Move immediate word to register |
| B8 + rd | MOV reg32,imm32 | 2 | Move immediate dword to register |
| Ciiiiii | MOV r/m8,imm8 | 2/2 | Move immediate byte to r/m byte |
| C7 | MOV r/m16,imm16 | 2/2 | Move immediate word to r/m word |
| C7 | MOV r/m32,imm32 | 2/2 | Move immediate dword to r/m dword |

- shell 命令 完成 PA1 的内容之后, nemu 目录下的所有.c 和.h 和文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在 PA1 中编写了多少行代码? (Hint: 使用 git checkout 可以回到"过去", 具体使用方法请查阅 man git-

checkout ) 你可以把这条命令写入 Makefile 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 make count 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, nemu 目录下的所有.c 和.h 文件总共有多少行代码?

**My project, without blanks:**

```
yangmin@yangmin-ThinkPad-T540p:~/ics2015$ find ./nemu -name "*.c" |xargs cat|grep -v ^$|wc -l
2169
yangmin@yangmin-ThinkPad-T540p:~/ics2015$ find ./nemu -name "*.h" |xargs cat|grep -v ^$|wc -l
1107
```

**Origin project, without blanks:**

```
yangmin@yangmin-ThinkPad-T540p:~/下载/ics2015-master$ find ./nemu -name "*.c" |xargs cat|grep -v ^$|wc -l
1851
yangmin@yangmin-ThinkPad-T540p:~/下载/ics2015-master$ find ./nemu -name "*.h" |xargs cat|grep -v ^$|wc -l
1100
```

- 使用 man 打开工程目录下的 Makefile 文件, 你会在 CFLAGS 变量中看到 gcc 的一些编译选项. 请解释 gcc 中的 -Wall 和 -Werror 有什么作用? 为什么要使用 -Wall 和 -Werror ?

-Wall
结合所有上述的`-W'选项.通常我们建议避免这些被警告的用法，我们相信,恰当结合宏的使用能够轻易避免这些用法。
剩下的`-W...'选项不包括在`-Wall'中,因为我们认为在必要情况下,这些被编译器警告的程序结构,可以合理的用在"干净的"程序中.
-Werror
视警告为错误;出现任何警告即放弃编译.
To interrupt the program when some warnings appears so that the program can run well, -Wall and -Werror are used for this purpose.