计算机系统设计实验
PA1 - 开天辟地的篇章: 最简单的计算机

安祺 1913630
南开大学计算机学院

日期: 2022 年 3 月 22 日

目录

1 概述

1.1 实验目的

1. 熟悉 GNU/Linux 平台
2. 初步探究“程序在计算机上运行”的相关原理
3. 初步学习 GDB 并在 PA 上实现简易调试器

1.2 实验内容

阶段一 模拟寄存器结构，实现调试器基本功能。
阶段二 实现调试功能的表达式求值，并完善阶段一中的扫描内存函数。
阶段三 实现调试功能中的监视点，学习断点相关知识与 i386。

2 阶段一

2.1 实现正确的寄存器结构体

首先查看CPU_state的实现要求,如代码1所示。在reg.c文件中,要求对CPU_state结构体按成员名称访问寄存器。这里有两种解决办法:

1. **宏**。类似于 linux 系统<sys/socket.h>文件中s_addr的定义方式,在CPU_state中逐寄存器定义#define eax gpr[0]._32。
2. **union**。借助 struct 中成员的名称对数组成员进行访问，需要使用 union 在内存中将 struct 和数组重叠排列。

Listing 1: CPU_state断言要求

```
assert(sample[R_EAX] == cpu.eax);
assert(sample[R_ECX] == cpu.ecx);
assert(sample[R_EDX] == cpu.edx);
assert(sample[R_EBX] == cpu.ebx);
assert(sample[R_ESP] == cpu.esp);
assert(sample[R_EBP] == cpu.ebp);
assert(sample[R_ESI] == cpu.esi);
assert(sample[R_EDI] == cpu.edi);
```

为避免使用宏,这里声明一个 8 个成员的 struct 与 gpr 数组在内存中借助 union 重叠放置，如代码2所示。在做实验的时候发现，原有代码其实不能在大端字节序机器上正常工作，但现有实验环境均为小端字节序，暂不做处理，标记为 TODO。

Listing 2: CPU_state定义

```
typedef struct {
  union{
    /* TODO: How does it work on big-endian machine? */
    union {
      uint32_t _32;
      uint16_t _16;
      uint8_t _8[2];
    } gpr[8];

    struct{
      rtlreg_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
  };

  vaddr_t eip;
} CPU_state;
```

2.2 cmd_c执行指令数

实验指导书 [1] 中提到了下面问题:

> 调用*cpu_exec()*的时候传入了参数*-1*，你知道这是什么意思吗?

只需查看cpu_exec()参数被如何使用即可，相关代码如代码3所示。这里输入的参数就是继续执行单条指令的次数，由于参数类型为uint64_t，因此输入 −1 后 n 的值为(uint64_t)(-1)即 $2^{64}-1$，这个数非常大，可以看作不停地执行指令。

Listing 3: cpu_exec定义

```
void cpu_exec(uint64_t n) {
  // other code

  bool print_flag = n < MAX_INSTR_TO_PRINT;
  for (; n > 0; n --) {
    /* Execute one instruction, including instruction fetch,
     * instruction decode, and the actual execution. */
    exec_wrapper(print_flag);

  // other code
}
```

2.3 调试器

2.3.1 单步执行

调试器中实现新的指令只需要在cmd_table中添加对应项实现单步调试的函数及函数即可。在ui_mainloop函数中会在cmd_table数组中搜索对应项去调用执行，详见代码4。代码5一次性给出本实验新添加的所有cmd_table项，其中cmd_si对应单步执行的实现函数。

单步执行的实现相对简单，只需要调用cpu_exec即可实现，cpu_exec会执行指定数量的指令后返回。由于实现相对简单，不列出代码。

Listing 4: ui_mainloop函数定义主要代码

```
void ui_mainloop(int is_batch_mode) {
  //other code
  while (1) {
    //other code
    int i;
    for (i = 0; i < NR_CMD; i ++) {
      if (strcmp(cmd, cmd_table[i].name) == 0) {
        if (cmd_table[i].handler(args) < 0) { return; }
        break;
}}}}
```

Listing 5: cmd_table新项

```
  { "si", "'si [N=1]',Step through N instructions.", cmd_si},
  { "info", "'info <r|w>', Print information.", cmd_info },
  { "p", "'p <EXPR>', Print the value of expression <EXPR>.", cmd_p },
  { "x", "'x <N> <EXPR>', Dump N 4-bytes.", cmd_x },
  { "w", "'w <EXPR>', Watch the expression <EXPR>.", cmd_w },
  { "d", "'d <N>', Delete the watchpoint.", cmd_d },
```

2.3.2 打印寄存器

打印寄存器的实现也比较简单，访问全局变量cpu的值即可，代码6给出了打印寄存器和打印监视点的代码。其中regsl数组保存了所有寄存器的名称。

Listing 6: 打印寄存器信息

```
static int cmd_info(char *args){
  // other code
  case 'r':
    for(int i=0;i<sizeof(cpu.gpr)/sizeof(*cpu.gpr);i++){
        printf("%s\t\t0x%08X\t%d\n",
          regsl[i], cpu.gpr[i]._32,cpu.gpr[i]._32);
    }
    printf("eip\t\t0x%08X\n",cpu.eip);
    return 0;
// other code
}
```

2.3.3 扫描内存

扫描内存可借助vaddr_read实现，在cmd_table注册的函数为cmd_x。代码7给出了扫描内存的实现，其中expr在阶段二实现。

Listing 7: 扫描内存实现

```
static int cmd_x(char *args){
  char *n_arg=strtok(args," ");
  char *expr_arg=strtok(NULL," ");

  int N=atoi(n_arg);
  bool success=success;
  uint32_t addr=expr(expr_arg,&success);
  if(N==0||!success){ /* handle errror */ }

  for(int i=0;i<4*N;i+=4,addr+=4){
    printf("%#X <+%d>:\t\t0x%08X\n", addr, i, vaddr_read(addr,4));
  }
  return 0;
}
```

2.3.4 监视点

监视点的添加借助new_wp实现，监视点的删除借助free_wp实现。这两个函数在阶段三实现，输入参数的解析不再赘述。

3 阶段二

本着一劳永逸的原则，对所有 C 语言支持的运算符都进行了实现。

3.1 算术表达式的词法分析

3.1.1 token 定义

根据现有代码框架，我们需要`rules`数组中添加 token 及对应正则表达式。首先，运算符和寄存器的正则表达式容易表示，简单枚举即可。数字的表达式相对复杂，需要单独考虑不同进制下数字的表示，这里实现了十进制、八进制、十六进制无符号整数的正则表达式，所有定义参见代码8。这里需注意 C 语言对字符串进行了一次转移，正则表达式也进行了一次转义，一共两次。

Listing 8: token 定义

```
{" +", TK_NOTYPE},
{"([1-9][0-9]*)|(0[0-7]+)|(0x[0-9A-Fa-f]+)|0", TK_NUM},
{"\\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip)", TK_REG},
{"\\(", '('},
{"\\)", ')'},
{"\\+", '+'},
{"-", '-'},
{"\\*", '*'},
{"/",   '/'},
{"%",   '%'},
{"==", TK_EQ},
{"!=", TK_NE},
{"<=", TK_LE},
{">=", TK_GE},
{">",  '>'},
{"<",  '<'},
{"&&",     TK_AND},
{"\\|\\|", TK_OR},
{"!",      '!'},
{"&",   '&'},
{"\\|", '|'},
{"\\^", '^'},
{"~",   '~'},
{"<<",  TK_LS},
{">>",  TK_RS},
```

3.1.2 词法分析

词法分析时将字符流转换成 token 流，所有识别出的 token 保存在tokens数组中。框架代码中使用遍历的方式对正则表达式进行匹配，这种方式在项目较少时性能尚可，使用状态机进行分析是更好但更复杂的选择。

实现词法分析的主要代码见9。这里统一起见，数字也像寄存器一样将字符串记录下来。

Listing 9: 实现词法分析

```
static bool make_token(const char *e) {
  // for each token matched with rules[i]
  switch (rules[i].token_type) {
    case TK_NOTYPE: break;
    case TK_REG:
    case TK_NUM:
      if(substr_len >= sizeof(tokens->str)-1){ /* handle error */}
      strncpy(tokens[nr_token].str,substr_start,substr_len);
      tokens[nr_token].str[substr_len]='\0';
    default:
      tokens[nr_token].type=rules[i].token_type;
      nr_token++;
      break;
    }
  // end for
}
```

3.2 token 的后处理

Listing 10: token 的后处理

```
//for tokens[i] in tokens
  case '-':
    if(i==0||(tokens[i-1].type!=')'&& tokens[i-1].type!=TK_NUM)){
      tokens[i].type=TK_MINUS;  // 替换所有负号
    }
    break;
//end for
}
```

难以在递归求解过程中区分负号和减法，因此需要提前进行处理，将 token 流中所有的负号替换为新的 token 类别。对于解引用和乘法也使用相同的方法进行处理。实现代码参见代码10。

3.3 递归求值

遵循 KISS 法则，这里使用递归的方式对表达式进行求值，性能更好的方法是进行自底向上的语法分析。

3.3.1 递归下降

递归求值的总体框架如代码11所示，分三种情况考虑：

1. **单个 token**：直接返回 token 的值。
2. **括号**：对括号进行去除。
3. **多 token**：对优先级最高的算法进行运算，并递归求解其余 token。

Listing 11: 递归求值

```
uint32_t eval(int lidx, int ridx, bool *success) {
  if (lidx > ridx) {/* handle error */}
  else if (lidx == ridx) {
    /* Single token.
     * For now this token should be a number.
     * Return the value of the number.
     */
    return token_value(lidx,success);
  }
  else if (check_parentheses(lidx, ridx, success) == true) {
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(lidx + 1, ridx - 1, success);
  }
  else {
    /* get the position of dominant operator in the token expression
       */
    int op=get_op_pos(lidx,ridx,success);
    /* return the value */
    return cal_expr(op,lidx,ridx,success);
  }
}
```

3.3.2 单 token 求值

非运算符的单个 token 为数字和寄存器两种，其中数字使用sscanf进行解析，寄存器通过搜索寄存器名字实现，参见代码12。

Listing 12: 求解单个 token 的值

```
static int token_value(int pos,bool* success){
  uint32_t val=0;
  switch(tokens[pos].type){
    default: return *success=false;
    case TK_NUM:
      if(EOF==sscanf(tokens[pos].str,"%i",&val)){
        printf("unknow num:%s\n",tokens[pos].str);
        return *success=false;
      }
      return val;
    case TK_REG:
      if(0==strcmp(tokens[pos].str+1,"eip")){
        return cpu.eip;
      }
      for(int i=0;i<8;i++){
        if(0==strcmp(tokens[pos].str+1,regsl[i])){
          return cpu.gpr[i]._32;
        }
      }
      Assert(0,"Unimplemented token: %d\n",tokens[pos].type);
  }
  return 0;
}
```

3.3.3 去除括号

在代码10中，检测到括号后，对边界进行收缩，实现去括号。代码中13实现了括号的检查，思想是对左右括号进行计数。

3.3.4 运算符优先级

求解多 token 表达式的值，首先需要确定优先级最低的操作符位置，这里通过定义操作数优先级等级的方式比较运算符的优先级，参见代码15。这里的优先级与常规的运算符优先级正好相反，越晚计算的运算符优先级越大。同时不同的运算符有不同

Listing 13: 检查括号

```
bool check_parentheses(int lidx,int ridx,bool *success){
  int lb=0;
  for(int i=lidx+1;i<ridx;i++){
    if(tokens[i].type=='('){
      lb++;
    }else if(tokens[i].type==')'){
      if(lb==0) return false;
      lb--;
    }
  }
  if(lb!=0){ /* handle error */ }
  return (tokens[lidx].type=='(') && (tokens[ridx].type==')');
}
```

的操作数，也对其进行了定义。这里使用switch-case语法实现对不同的操作符返回其优先级和操作数数量，这是一种简便但是低效的方法，考虑效率应使用数据表进行实现。

查找一串 token 中最晚进行计算的运算符的位置，需要对其进行遍历，考虑所有等级最大的运算符。同时考虑到所有实现的运算符均为左结合，应选择其中最靠右出现的。此外，其中的括号及括号中的内容直接进行跳过，因为其总是最先运算。相关实现参见代码14。

Listing 14: 获得最晚进行计算的运算符位置

```
static int get_op_pos(int lidx,int ridx,bool* success)
{
  int op=lidx, priority=-1;
  bool inbracket=false;
  for(int i=lidx;i<=ridx;i++){
    switch (tokens[i].type){
      case '(': inbracket=true; break;
      case ')': inbracket=false; break;
      default:break;
    }
    if(inbracket) continue;

    int curr_prio=priority_of(tokens[i].type);
    if(curr_prio>=priority){
      op=i;
      priority=curr_prio;
```

```
    }
  }
  // some code
  return op;
}
```

Listing 15: 运算符优先级

```
int priority_of(int type){
  switch(type){
    default: return -1;
    case TK_MINUS:
    case TK_DEREF:
    case '!':
    case '~':return 2;
    case '*':
    case '/':
    case '%': return 3;
    case '+':
    case '-': return 4;
    case TK_LS:
    case TK_RS: return 5;
    case TK_LE:
    case TK_GE:
    case '>':
    case '<': return 6;
    case TK_EQ:
    case TK_NE:return 7;
    case '&': return 8;
    case '^': return 9;
    case '|': return 10;
    case TK_AND: return 11;
    case TK_OR: return 12;
  }
}
```

3.3.5 运算求值

得到最先进行运算的运算符 OP，就可以对一串 token 进行递归求解了。首先递归求解 OP 的所有操作数，然后返回 OP 的计算结果，实现参见代码16。

Listing 16: 递归计算表达式

```
static int cal_expr(int op,int lidx,int ridx,bool* success)
{
#define EVAL_VAL1 (eval(lidx, op - 1, success))
#define EVAL_VAL2 (eval(op + 1, ridx, success))
  switch (tokens[op].type) {
    default: return *success=false;
    case TK_MINUS: return - EVAL_VAL2;
    case TK_DEREF: return vaddr_read(EVAL_VAL2,4);
    case '!': return ! EVAL_VAL2;
    case '~': return ~ EVAL_VAL2;
    case '*': return EVAL_VAL1 * EVAL_VAL2;
    case '/': return EVAL_VAL1 / EVAL_VAL2;
    case '%': return EVAL_VAL1 % EVAL_VAL2;
    case '+': return EVAL_VAL1 + EVAL_VAL2;
    case '-': return EVAL_VAL1 - EVAL_VAL2;
    case TK_LS: return EVAL_VAL1 << EVAL_VAL2;
    case TK_RS: return EVAL_VAL1 >> EVAL_VAL2;
    case TK_LE: return EVAL_VAL1 <= EVAL_VAL2;
    case TK_GE: return EVAL_VAL1 >= EVAL_VAL2;
    case '>': return EVAL_VAL1 > EVAL_VAL2;
    case '<': return EVAL_VAL1 < EVAL_VAL2;
    case TK_EQ: return EVAL_VAL1 == EVAL_VAL2;
    case TK_NE: return EVAL_VAL1 != EVAL_VAL2;
    case '&': return EVAL_VAL1 & EVAL_VAL2;
    case '^': return EVAL_VAL1 ^ EVAL_VAL2;
    case '|': return EVAL_VAL1 | EVAL_VAL2;
    case TK_AND: return EVAL_VAL1 && EVAL_VAL2;
    case TK_OR: return EVAL_VAL1 || EVAL_VAL2;
  }
#undef EVAL_VAL1
#undef EVAL_VAL2
}
```

4 阶段三

4.1 监视点池

代码段17给出了监视点的数据结构,其中expr_str用于表示监视的表达式、old_value用于记录上一次的监视值 (用于检查表达式的值是否发生变化)。

Listing 17: 监视点的数据结构

```
typedef struct watchpoint {
  int NO;
  struct watchpoint *next;

  char expr_str[128];
  int old_value;
} WP;
```

代码段18给出了创建一个监视点的实现。首先在链表中插入一个值，然后在为其计算表达式的值并保存表达式。链表操作不再赘述。

Listing 18: 分配新的监视点

```
WP* new_wp(const char* expr_str){
  if(free_ == NULL || strlen(expr_str)>=sizeof(head->expr_str)){
    return NULL;
  }

  bool success=true;
  int value=expr(expr_str,&success);
  if(!success) return NULL;

  WP* temp=head;
  head=free_;
  head->next=temp;
  free_=free_->next;

  head->old_value=value;
  strcpy(head->expr_str,expr_str);
  return head;
}
```

代码段21给出了删除一个监视点的实现。遍历整个链表，将目标编号的监视点从

链表中删除。链表操作不再赘述。

Listing 19: 释放特定监视点

```
void free_wp(int no){
  WP *curr=head, *last=NULL;
  while(curr!=NULL && curr->NO!=no ) {
    last=curr;
    curr=curr->next;
  }
  if(curr==NULL) {
    return;
  }else if(curr == head){
    head=curr->next;
  }else{
    last->next=curr->next;
  }

  curr->next=free_;
  free_=curr;
}
```

4.2 检查监视点

这里通过一个函数同时实现更新检查点和打印检查点信息，参见代码20。如果输入的参数为真，则更新监视点并打印变化值，返回打印个数 (用于更新)；如果输入参数为假，则不跟新监视点数值，对所有监视点进行打印，返回打印个数 (用于打印)。

在cpu_exec函数中插入对监视点的更新，以实现在每条指令执行后观察监视点是否发生变化。如果有至少一个监视点发生变化，则暂停 cpu 的执行。

5 i386 手册

通过在手册目录中搜索"selector" 我们得到三个结果，及要查询的范围:

1. "5.1.3 Selector"
2. "17.2.2.11 Instruction Set Detail " 下的"ARPL Adjust RPL Field of Selector"
3. "5-6 Format of a Selector"

Listing 20: 检测监视点变化

```
int print_wps(bool check_change){
  int print_num=0;
  for(WP* curr=head;curr!=NULL;curr=curr->next){
    if(!check_change){
      printf("%d\t%s\t\t\t= %d\n",curr->NO,curr->expr_str,curr->
          old_value);
      print_num++;
    }else{
      bool success=true;
      int new_value=expr(curr->expr_str, &success);
      assert(success);
      if(new_value!=curr->old_value){
        printf("%d\t%s\t\t\t= %d -> %d\n",curr->NO,curr->expr_str,
          curr->old_value,new_value);
        curr->old_value=new_value;
        print_num++;
      }
    }
  }
  return print_num;
}
```

Listing 21: 每步执行都检查所有监视点

```
/* Simulate how the CPU works. */
void cpu_exec(uint64_t n) {
  //other code
  for (; n > 0; n --) {
    /* Execute one instruction, including instruction fetch,
     * instruction decode, and the actual execution. */
    exec_wrapper(print_flag);

#ifdef DEBUG
    /* DONE: check watchpoints here. */
    if(print_wps(true)>0){
      nemu_state=NEMU_STOP;
    }
#endif
  //other code
}
```

6 Bug 总结

6.1 高版本 gcc 的 Werror 问题

由于高版本编译器中会比低版本编译器多出一些警，不改变代码的情况下，Werror 会让代码不能通过。这里希望继续开启 Werror 并在高版本编译器编译通过，于是忽略该警告，见代码23。

Listing 22: 忽略警告

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpragmas"
#pragma GCC diagnostic ignored "-Wrestrict"
strcat(decoding.asm_buf, decoding.assembly);
#pragma GCC diagnostic pop
```

6.2 括号检查

在检测括号时，对于(1)+(1)的情况没有进行考虑，导致出错。

7 手册必答题

7.1 简易调试器节省的时间

$$time = 500 \times 0.9 \times 20 \times (30 - 10)s = 50h$$

简易调试器可以节约 50 个小时。

7.2 i386 手册范围

"EFLAGS" 的范围为如下，图表对应正文中有对 CF 标志位的说明：

1. "2-8 EFLAGS Register"
2. "4-1 Systems Flags of EFLAGS Register"
3. "2.3.4 Flags Register"
4. "APPENDIX C STATUS FLAG SUMMA"

"ModR/M 字节" 的范围如下，

1. "17.2.1 ModR/M and SIB B"
2. "17-2 ModR/M and SIB Byte Formats"

"mov 指令的具体格式" 的范围如下,

1. "MOV Move Data"
2. "MOV Move to/from Special"

7.3 shell 统计代码行数

利用管道对所有文件的文本按行计数，xargs 捕获一个命令的输出传递给另外一个命令，wc 用于统计词数,"-l" 参数指定按行统计。grep 用于输出匹配行，ind 用于查找文件。

Listing 23: 统计代码行数

```
find -regex ".*\.h\|.*\.c" | xargs cat |wc -l
find -regex ".*\.h\|.*\.c" | xargs cat |grep -v ^$|wc -l //去空行
```

要查看仓库初始代码行数只需`git checkout pa0`切换到原理的分支即可。

7.4 Wall 和 Werror

Wall 参数会打开 gcc 的所有警告，许多默认不打开的警告都会被打开。Werror 会将所有的警告视为错误，编译时检查更严格。

8 感悟与体会

做实验的时候没有一起写报告，前后相隔近一个月，导致很多东西遗忘了。下次实验和报告一起做。

参考文献

[1] YU Z. Ics2017 programming assignment[EB/OL]. [March 22, 2022]. https://nju-ics.gitbooks.io/ics2017-programming-assignment/.