[**深入理解计算机系统(第二版) 家庭作业 第二章**](http://blog.csdn.net/zhanyu1990/article/details/24936663)

2.55-2.57

略

2.58

**int** **is\_little\_endian**(){  
    **int** a = 1;  
    return \*((**char**\*)&a);  
}

2.59

(x&0xFF) | (y&~0xFF)

2.60

unsigned **replace\_byte**(unsigned x, unsigned **char** b, **int** i)  
{  
    return (x & ~(0xFF<<(i<<3))) | (b << (i<<3));  
}

2.61

A. !~x

B. !x

C. !~(x>>((sizeof(int)-1)<<3))

D. !(x&0xFF)

注意，英文版中C是最低字节，D是最高字节。中文版恰好反过来了。这里是按中文版来做的。

2.62

这里我感觉应该是英文版对的，int\_shifts\_are\_arithmetic()

**int** **int\_shifts\_are\_arithmetic**(){  
    **int** x = -1;  
    return (x>>1) == -1;  
}

2.63

对于sra，主要的工作是将xrsl的第w-k-1位扩展到前面的高位。

这个可以利用取反加1来实现，不过这里的加1是加1<<(w-k-1)。

如果x的第w-k-1位为0，取反加1后，前面位全为0，如果为1，取反加1后就全是1。

最后再使用相应的掩码得到结果。

对于srl，注意工作就是将前面的高位清0，即xsra & (1<<(w-k) - 1)。额外注意k==0时，不能使用1<<(w-k)，于是改用2<<(w-k-1)。

**int** **sra**(**int** x, **int** k){  
    **int** xsrl = (unsigned) x >> k;  
    **int** w = **sizeof**(**int**) << 3;  
    unsigned z = 1 << (w-k-1);  
    unsigned mask = z - 1;  
    unsigned right = mask & xsrl;  
    unsigned left = ~mask & (~(z&xsrl) + z);  
    return left | right;  
}  
  
**int** **srl**(unsigned x, **int** k){  
    **int** xsra = (**int**) x >> k;  
    **int** w = **sizeof**(**int**)\*8;  
    unsigned z = 2 << (w-k-1);  
    return (z - 1) & xsra;  
}

2.64

**int** **any\_even\_one**(unsigned x){  
    return !!(x & (0x55555555));  
}

2.65

**int** **even\_ones**(unsigned x){  
    x ^= (x >> 16);  
    x ^= (x >> 8);  
    x ^= (x >> 4);  
    x ^= (x >> 2);  
    x ^= (x >> 1);  
    return !(x&1);  
}

x的每个位进行异或，如果为0就说明是偶数个1，如果为1就是奇数个1。

那么可以想到折半缩小规模。最后一句也可以是 return (x^1)&1

2.66

根据提示想到利用或运算，将最高位的1或到比它低的每一位上，忽然想如果x就是10000000..该如何让每一位都为1。于是便想到了二进扩展。先是x右移1位再和原x进行或，变成1100000...，再让结果右移2位和原结果或，变成11110000...，最后到16位，变成11111111...。

**int** **leftmost\_one**(unsigned x){  
    x |= (x >> 1);  
    x |= (x >> 2);  
    x |= (x >> 4);  
    x |= (x >> 8);  
    x |= (x >> 16);  
    return x^(x>>1);  
}

2.67

A.32位机器上没有定义移位32次。

B.beyond\_msb变为 2<<31。

C.定义 a = 1<<15; a<<=15; set\_msb = a<<1; beyond\_msb = a<<2;

2.68

感觉中文版有点问题，注释和函数有点对应不上，于是用英文版的了。

个人猜想应该是让x的最低n位变1。

**int** **lower\_one\_mask**(**int** n){  
    return (2<<(n-1)) - 1;  
}

2.69

unsigned **rotate\_right**(unsigned x, **int** n){  
    **int** w = **sizeof**(unsigned)\*8;  
    return (x>>n) | (x<<(w-n-1)<<1);  
}

2.70

这一题是看x的值是否在 - 2^(n-1) 到 2^(n-1) - 1之间。

如果x满足这个条件，则其第n-1位就是符号位。如果该位为0，则前面的w-n位均为0，如果该位为1，则前面的w-n位均为1。所以本质是判断，x的高w-n+1位是否为0或者为-1。

**int** **fits\_bits**(**int** x, **int** n){  
    x >>= (n-1);  
    return !x || !(~x);  
}

2.71

A.得到的结果是unsigned，而并非扩展为signed的结果。

B.使用int，将待抽取字节左移到最高字节，再右移到最低字节即可。

**int** **xbyte**(unsigned word, **int** bytenum){  
    **int** ret = word << ((3 - bytenum)<<3);  
    return ret >> 24;  
}

2.72

A.size\_t是无符号整数，因此左边都会先转换为无符号整数，它肯定是大于等于0的。

B.判断条件改为if(maxbytes > 0 && maxbytes >= sizeof(val))

2.73

请先参考2.74题。

可知：t = a + b时，如果a，b异号（或者存在0），则肯定不会溢出。

如果a，b均大于等于0，则t<0就是正溢出，如果a，b均小于0，则t>=0就是负溢出。

于是，可以利用三个变量来表示是正溢出，负溢出还是无溢出。

**int** **saturating\_add**(**int** x, **int** y){  
    **int** w = **sizeof**(**int**)<<3;  
    **int** t = x + y;  
    **int** ans = x + y;  
    x>>=(w-1);  
    y>>=(w-1);  
    t>>=(w-1);  
    **int** pos\_ovf = ~x&~y&t;  
    **int** neg\_ovf = x&y&~t;  
    **int** novf = ~(pos\_ovf|neg\_ovf);  
    return (pos\_ovf & INT\_MAX) | (novf & ans) | (neg\_ovf & INT\_MIN);  
}

2.74

对于有符号整数相减，溢出的规则可以总结为：

t = a-b;

如果a, b 同号，则肯定不会溢出。

如果a>=0 && b<0，则只有当t<=0时才算溢出。

如果a<0 && b>=0，则只有当t>=0时才算溢出。

不过，上述t肯定不会等于0，因为当a，b不同号时：

1) a!=b，因此a-b不会等于0。

2) a-b <= abs(a) + abs(b) <= abs(TMax) + abs(TMin)=(2^w - 1)

所以，a，b异号，t，b同号即可判定为溢出。

**int** **tsub\_ovf**(**int** x, **int** y){  
    **int** w = **sizeof**(**int**)<<3;  
    **int** t = x - y;  
    x>>=(w-1);  
    y>>=(w-1);  
    t>>=(w-1);  
    return (x != y) && (y == t);  
}

顺便整理一下汇编中CF，OF的设定规则(个人总结，如有不对之处，欢迎指正)。

t = a + b;

CF: (unsigned t) < (unsigned a) 进位标志

OF: (a<0 == b<0) && (t<0 != a<0)

t = a - b;

CF: (a<0 && b>=0) || ((a<0 == b<0) && t<0) 退位标志

OF: (a<0 != b<0) && (b<0 == t<0)

汇编中，无符号和有符号运算对条件码（标志位）的设定应该是相同的，但是对于无符号比较和有符号比较，其返回值是根据不同的标志位进行的。

详情可以参考第三章3.6.2节。

2.75

根据2-18，不难推导， (x'\*y')\_h = (x\*y)\_h + x(w-1)\*y + y(w-1)\*x。

unsigned **unsigned\_high\_prod**(unsigned x, unsigned y){  
    **int** w = **sizeof**(**int**)<<3;  
    return **signed\_high\_prod**(x, y) + (x>>(w-1))\*y + x\*(y>>(w-1));  
}

当然，这里用了乘法，不属于整数位级编码规则，聪明的办法是使用int进行移位，并使用与运算。即 ((int)x>>(w-1)) & y 和 ((int)y>>(w-1)) & x。

注：不使用long long来实现signed\_high\_prod(int x, int y)是一件比较复杂的工作，而且我不会只使用整数位级编码规则来实现，因为需要使用循环和条件判断。

下面的代码是计算两个整数相乘得到的高位和低位。

**int** **uadd\_ok**(unsigned x, unsigned y){  
    return x + y >= x;  
}

void **signed\_prod\_result**(**int** x, **int** y, **int** &h, **int** &l){  
    **int** w = **sizeof**(**int**)<<3;  
    h = 0;  
    l = (y&1)?x:0;  
    **for**(**int** i=1; i<w; i++){  
        **if**( (y>>i)&1 ) {  
            h += (unsigned)x>>(w-i);  
            **if**(!**uadd\_ok**(l, x<<i)) h++;  
            l += (x<<i);  
        }  
    }  
    h = h + ((x>>(w-1))\*y) + ((y>>(w-1))\*x);  
}

最后一步计算之前的h即为unsigned相乘得到的高位。

sign\_h = unsign\_h - ((x>>(w-1)) & y) - ((y>>(w-1)) & x);

sign\_h = unsign\_h + ((x>>(w-1)) \* y) + ((y>>(w-1)) \* x);

2.76

A. K=5: (x<<2) + x

B. K=9: (x<<3) + x

C. K=30: (x<<5) - (x<<1)

D. K=-56: (x<<3) - (x<<6)

2.77

先计算x>>k，再考虑舍入。

舍入的条件是x<0&&x的最后k位不为0。

**int** **divide\_power2**(**int** x, **int** k){  
    **int** ans = x>>k;  
    **int** w = **sizeof**(**int**)<<3;  
    ans += (x>>(w-1)) && (x&((1<<k)-1));  
    return ans;  
}

2.78

这相当于计算((x<<2) + x) >> 3，当然，需要考虑x为负数时的舍入。

先看上述表达式，假设x的位模式为[b(w-1), b(w-2), ... , b(0)]，那么我们需要计算：

[b(w-1),b(w-2),b(w-3),   ...    ,b(0),  0,    0]

+             [b(w-1),b(w-2),...,b(2),  b(1), b(0)]

最后需要右移3位。因此我们可以忽略下方的b(1),b(0)。

于是就计算(x>>2) + x，再右移一位即是所求答案。

不过考虑到(x>>2) + x可能也会溢出，于是就计算(x>>3) + (x>>1)，这个显然是不会溢出的。再看看b(0)+b(2)会不会产生进位，如果产生进位，则再加一。

最后考虑负数的舍入。负数向0舍入的条件是x<0 && ((x<<2)+x 的后三位不全为0)。满足舍入条件的话，结果再加1。容易证明，加法后三位不全为0可以等价为x后三位不全为0。

**int** **mul5div8**(**int** x){  
    **int** b0 = x&1, b2 = (x>>2)&1;  
    **int** ans = (x>>3) + (x>>1);  
    **int** w = **sizeof**(**int**)<<3;  
    ans += (b0&b2);  
    ans += ((x>>(w-1)) && (x&7));  
    return ans;  
}

2.79

不懂题意，感觉就是2.78。

2.80

A. 1[w-n]0[n]:  ~((1<<n) - 1)

B. 0[w-n-m]1[n]0[m]: ((1<<n) - 1) << m

2.81

A. false，当x=0,y=TMin时，x > y，而-y依然是Tmin，所以-x > -y。

B. true，补码的加减乘和顺序无关（如果是右移，则可能不同）。

C. false，当x=-1, y=1时，~x + ~y = 0xFFFFFFFE，而~(x+y) == 0xFFFFFFFF。

D. true，无符号和有符号数的位级表示是相同的。

E. true，最后一个bit清0，对于偶数是不变的，对于奇数相当于-1，而TMin是偶数，因此该减法不存在溢出情况。所以左边总是<=x。

2.82

A. 令x为无穷序列表示的值，可以得到x\*2^k = Y + x。

所以 x = Y/(2^k - 1)。

B. (a)1/7, (b)9/15 = 3/5, (c)7/63 = 1/9

2.83

浮点数的一个特点就是，如果大于0，则可以按unsigned位表示的大小排序。

如果小于0则相反。注意都为0的情况即可。

所以条件是：

((ux<<1)==0 && (uy<<1)==0) ||

(!sx && sy) ||

(!sx && !sy && ux >= uy) ||

(sx && sy && ux <= uy);

2.84

A. 5.0，5表示为101，因此位数M就是1.01为1.25，小数f为0.01 = 0.25。指数部分应该为E=2，所以其指数部分位表示为e=(2^(k-1)-1) + 2 = 2^(k-1)+1。

位表示三个部分分别是s-e-f，为0-10..01-0100..0。

B.能被准确描述的最大奇数，那么其M=1.111..1，故f部分全为1，E应该为n。当然，这个假设在2^(k-1)>=n的情况下才能成立。这时，s=0,e=n+2^(k-1)-1,f=11...1。值为2^(n+1)-1。

C.最小的规格化数为2^(1-bias)即2^(-2^(k-1)+2)，所以其倒数值V为2^(2^(k-1)-2)，所以M为1.00000，f部分为全0，E=2^(k-1)-2，e部分为2^(k-1)-2 + bias = 2^k - 3，即为11..101。位表示为0-11..101-00..0。

2.85

|  |  |  |
| --- | --- | --- |
| 描述 | 扩展精度 | |
| 值 | 十进制 |
| 最小的正非规格化数 | 2^(-63)\*2^(-2^14+2) | 3.6452e-4951 |
| 最小的正规格化数 | 2^(-2^14+2) | 3.3621e-4932 |
| 最大的规格化数 | (2^64-1)\*2^(2^14-1-63) | 1.1897e+4932 |

2.86

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 描述 | Hex | M | E | V |
| -0 | 0x8000 | 0 | -62 | -- |
| 最小的值>1 | 0x3F01 | 257/256 | 0 | 257\*2^(-8) |
| 256 | 0x4700 | 1 | 8 | -- |
| 最大的非规格化数 | 0x00FF | 255/256 | -62 | 255\*2^(-70) |
| -inf | 0xFF00 | -- | -- | -- |
| Hex为0x3AA0 | 0x3AA0 | 416/256 | -5 | 416\*2^（-13）=13\*2^(-8) |

2.87

|  |  |  |  |
| --- | --- | --- | --- |
| 格式A | | 格式B | |
| 位 | 值 | 位 | 值 |
| 1 01110 001 | -9/16 | 1 0110 0010 | -9/16 |
| 0 10110 101 | 208 | 0 1110 1010 | 208 |
| 1 00111 110 | -7/1024 | 1 0000 0111 | -7/1024 |
| 0 00000 101 | 6/2^17 | 0 0000 0000 | 0 |
| 1 11011 000 | -4096 | 1 1111 0000 | -inf |
| 0 11000 100 | 768 | 0 1111 0000 | inf |

没有特别明白转换成最接近的，然后又说向+inf舍入的含义。

按理说，舍入到+inf就是向上舍入，而并不是找到最接近的。

表格中是按最接近的进行舍入，并且如果超出范围则认为是inf。

如果都按+inf进行舍入，那么第四行格式B将是0 0000 0001。

2.88

A. false，float只能精确表示最高位1和最低位的1的位数之差小于24的整数。所以当x==TMAX时，用float就无法精确表示，但double是可以精确表示所有32位整数的。

B. false，当x+y越界时，左边不会越界，而右边会越界。

C. true，double可以精确表示所有正负2^53以内的所有整数。所以三个数相加可以精确表示。

D. false，double无法精确表示2^64以内所有的数，所以该表达式很有可能不会相等。虽然举例子会比较复杂，但可以考虑比较大的值。

E. false，0/0.0为NaN，(非0)/0.0为正负inf。同号inf相减为NaN，异号inf相减也为被减数的inf。

2.89

float的k=8, n=23。 bias = 2^7 - 1 = 127。

最小的正非规格化数为2^(1-bias-n) = 2^-149。

最小的规格化数为2^(0-bias)\*2 = 2^-126。

最大的规格化数（二的幂）为2^(2^8-2 - bias) = 2^127。

因此按各种情况把区间分为[TMin, -148] [-149, -125] [-126, 127] [128, TMax]。

**float** **fpwr2**(**int** x)  
{  
    /\* Result exponent and fraction \*/  
    unsigned exp, frac;  
    unsigned u;  
    **if** (x < -149) {  
        /\* Too small. Return 0.0 \*/  
        exp = 0;  
        frac = 0;  
    } **else** **if** (x < -126) {  
        /\* Denormalized result \*/  
        exp = 0;  
        frac = 1<<(x+149);  
    } **else** **if** (x < 128) {  
        /\* Normalized result. \*/  
        exp = x + 127;  
        frac = 0;  
    } **else** {  
        /\* Too big. Return +oo \*/  
        exp = 255;  
        frac = 0;  
    }  
    /\* Pack exp and frac into 32 bits \*/  
    u = exp << 23 | frac;  
    /\* Return as float \*/  
    return **u2f**(u);  
}

2.90

A.pi的二进制数表示为：0 10000000 10010010000111111101011，E=128-127=1，

它表示的二进制小数值为：11.0010010000111111101011

B.根据2.82，可知1/7的表示为0.001001[001]...，

所以22/7为11.001001001001001[001]...

C.从第9位开始不同。

为了方便测试2.91-2.94，我写了几个公共函数。

typedef unsigned float\_bits;  
  
**float** **u2f**(unsigned x){  
    return \*((**float**\*)&x);  
}  
unsigned **f2u**(**float** f){  
    return \*((unsigned\*)&f);  
}  
  
**bool** **is\_float\_equal**(float\_bits f1, **float** f2){  
    return **f2u**(f2) == f1;  
}  
  
**bool** **is\_nan**(float\_bits fb){  
    unsigned sign = fb>>31;  
    unsigned exp = (fb>>23) & 0xFF;  
    unsigned frac = fb&0x7FFFFF;  
    return exp == 0xFF && frac != 0;  
}  
  
**bool** **is\_inf**(float\_bits fb){  
    unsigned sign = fb>>31;  
    unsigned exp = (fb>>23) & 0xFF;  
    unsigned frac = fb&0x7FFFFF;  
    return exp == 0xFF && frac == 0;  
}  
  
**int** **testFun**( **float\_bits**(\*fun1)(float\_bits),  **float**(\*fun2)(**float**)){  
    unsigned x = 0;  
    **do**{ //test for all 2^32 value  
        float\_bits fb = **fun1**(x);  
        **float** ff = **fun2**(**u2f**(x));  
        **if**(!**is\_float\_equal**(fb, ff)){  
            **printf**("%x error\n", x);  
            return 0;  
        }  
        x++;  
    }**while**(x!=0);

**printf**("Test OK\n");   
    return 1;

}

最后的testFun是用来测试fun1和fun2是否对每种可能的输入都输出相同的值，fun1为题中所要求的函数，fun2为float版本。这个函数大概会运行2到3分钟，也可以写多线程，利用多核处理器求解。

2.91

float\_bits **float\_absval**(float\_bits f){  
    **if**(**is\_nan**(f)) return f;  
    **else** return f & 0x7FFFFFFF;  
}  
**float** **float\_absval\_f**(**float** f){  
    **if**(**is\_nan**(**f2u**(f))) return f;  
    **else** return **fabs**(f);

}

测试即调用testFun(float\_absval, float\_absval\_f);

在测试的时候发现0x7F800001的时候不对了。

后来debug了一下，发现u2f的时候，会篡改原值。

即令x = 0x7F800001。

则f2u(u2f(x))会变成0x7FC00001。奇怪的nan，第22位一定是1。

我将f2u和u2f里用memcpy也同样是不行。

所以，我将testFun中的一个条件变成了：

if(!is\_float\_equal(fb, ff) && !is\_nan(fb))

这个bug实在是不知道怎么回事。想了想，这和高位低位排列是无关的。这个bug还是之后再找吧。也有可能是硬件本身的原因了。

注：C库中也提供了isnan()函数。

2.92

float\_bits **float\_negate**(float\_bits f){  
    **if**(**is\_nan**(f)) return f;  
    **else** return f^0x80000000;  
}  
  
**float** **float\_negate\_f**(**float** f){  
    **if**(**isnan**(f)) return f;  
    return -f;

}

就是将最高位反位。

2.93

float\_bits **float\_half**(float\_bits f){  
    unsigned sign = f>>31;  
    unsigned exp = (f>>23) & 0xFF;  
    unsigned frac = f&0x7FFFFF;  
    **if**(exp == 0) return sign<<31 | ((frac>>1) + ((frac&1)&&((frac>>1)&1)));  
    **else** **if**(exp == 1) return sign<<31 | (( (1<<22) | (frac>>1)) + ((frac&1)&&((frac>>1)&1))) ;  
    **else** **if**(exp != 255) return sign<<31 | (exp-1) << 23 | frac;  
    **else** return f;  
}  
  
**float** **float\_half\_f**(**float** f){  
    **if**(!**isnan**(f)) return (**float**)0.5\*f;  
    **else** return f;

}

需要注意的是，舍入采用的是向偶数舍入。这也是我在测试的过程中发现的。（好吧，书上在浮点数位级编码规则中说过了，眼残没看到）

最后，非规格化的平滑效果让exp==1时的程序变得比较简洁。

2.94

float\_bits **float\_twice**(float\_bits f){  
    unsigned sign = f>>31;  
    unsigned exp = (f>>23) & 0xFF;  
    unsigned frac = f&0x7FFFFF;  
    **if**(exp == 0) return sign<<31 | frac<<1;  
    **else** **if**(exp < 254) return sign<<31 | (exp+1)<<23 | frac;  
    **else** **if**(exp == 254) return sign<<31 | 0xFF<<23;  
    **else** return f;  
}  
  
**float** **float\_twice\_f**(**float** f){  
    **if**(!**isnan**(f)) return (**float**)2\*f;  
    **else** return f;

}

比float\_half简单一些。对于非规格化的平滑，使用移位就可以了，对于规格化，只要exp+1即可，当然，如果exp==254，就要返回inf了。

2.95

float\_bits **float\_i2f**(**int** i)  
{  
    **if**(i == 0) return 0;  
    unsigned x = i>0?i:-i;  
    **int** sign = i>0?0:1;  
    **int** w = **sizeof**(**int**)<<3;  
    **int** j;  
    **for**(j=w-1; j>=0; j--){ //找到最高位  
        **if**( (x>>j) & 1) break;  
    }  
    unsigned bias = 127;  
    unsigned exp, frac;  
    exp = bias + j;  
    **if**(j <= 23) frac = x<<(23-j);  
    **else** {  
        frac = x>>(j-23);  
        unsigned mask = (1<<(j-23)) - 1;  
        **if**( (x&mask) > (1<<(j-24)) ) frac++; //需要舍入到大值  
        **else** **if**( (x&mask) == 1<<(j-24)  &&  (frac&1)) frac++; //舍入到偶数  
        **if**(frac == (1<<24)) exp++; //舍入到偶数超过(1<<24) - 1，指数需要再加1  
    }  
    return sign<<31 | exp<<23 | frac&0x7FFFFF;  
}  
  
void **test**(){  
    **int** x = 0;  
    **do**{  
        float\_bits fb = **float\_i2f**(x);  
        **float** ff = (**float**)x;  
        **if**(!**is\_float\_equal**(fb, ff)){  
            **printf**("error in %d:  %x %x\n", x, fb, **f2u**(ff));  
            return;  
        }  
        x++;  
    }**while**(x!=0);  
    **printf**("Test OK\n");

}

无耻地使用了循环。我也是一点一点测试修改，才通过的。不过好在大方向都知道，所以没有花多少时间，主要纠结点还是在舍入那块。需要特别注意。

2.96

**int** **float\_f2i**(float\_bits f){  
    unsigned sign = f>>31;  
    **int** exp = (f>>23) & 0xFF;  
    **int** frac = (f&0x7FFFFF) | (1<<23);  
    exp -= 127;  
    **if**(exp < 0) return 0;  
    **if**(exp >= 31) return 0x80000000; //绝对值不大于2^31（1<<31）  
    **if**(exp > 23) frac <<= (exp - 23);  
    **else** frac >>= (23 - exp);  
    return sign? -frac : frac;  
}  
  
void **test2**(){  
    **int** x = 0;  
    **do**{  
        **int** m = **float\_f2i**(x);  
        **int** n = (**int**)**u2f**(x);  
        **if**(m != n){  
            **printf**("error in %x:  %d %d\n", x, m, n);  
            return;  
        }  
        x++;  
    }**while**(x!=0);  
    **printf**("Test OK\n");

}

在exp<0和>=31上犯了小错误。开始写成<=0和>=32了。

其实1这个整数就是exp==0的。

而int绝对值不会超过2^31-1，因此1.0000..小数点右移不会到超过30次（否则就越界了），所以exp<=30。而这里刚好用TMin来表示越界，因此不用关心TMin的表示。

[**深入理解计算机系统(第二版) 家庭作业 第三章**](http://blog.csdn.net/zhanyu1990/article/details/9157603)

**3.54**

**int** **decode2**(**int** x, **int** y, **int** z)  
{  
    **int** ret;  
    z -= y; //line 2  
    ret = z; //line 3  
    ret <<= 15;//line 4  
    ret >>= 15;//line 5  
    return ret\*(z^x);  
}

**3.55**

大概算法如下：

x的高32位为xh，低32位为xl。

y的符号位扩展成32位之后为ys（ys为0或者-1）。

dest\_h = (xl\*ys)\_l + (xh\*y)\_l + (xl\*y)\_h

dest\_l = (xl\*y)\_l

注意，所有的乘法都是unsigned\*unsigned。

也就是说对于 1\*(-1)，如果存入两个寄存器中，那么高32位是0，低32位是-1。

相当于 1\*(UNSIGNED\_MAX)。

**3.56**

注意n在这里是一个很小的数，用8位就能表示，也可以用n=n%256表示。

寄存器 变量

esi    x

ebx    n

edi    result

edx    mask

**int** **loop**(**int** x, **int** n)  
{  
    **int** result = 1431655765;  
    **int** mask;  
    **for**(mask = 1<<31; mask != 0; mask = ((**unsigned**)mask)>>n){  
        result ^= (mask & x);  
    }  
    return result;  
}

**3.57**

xp?\*xp:0这个语句是不能被编译成条件传送语句的。因为如果是条件传送语句，那么不论xp为什么，\*xp都会被计算。

我们要写一个和该功能完全一样的能编译成条件传送语句的函数。

于是，我们要想办法不使用\*xp，而使用一个替代的指向0的非空指针。

**int** **cread\_alt**(**int** \*xp)  
{  
    **int** t = 0;  
    **int** \*p = xp?xp:&t;  
    return \*p;  
}

**3.58**

MODE\_A: result = \*p1; \*p1 = \*p2; break;  
MODE\_B: result = \*p1 + \*p2; \*p2 = result; break;  
MODE\_C: result = \*p1; \*p2 = 15; break;  
MODE\_D: \*p2 = \*p1;  
MODE\_E: result = 17; break;  
**default**: result = -1; break;

**3.59**

**int** **switch\_prob**(**int** x, **int** n)  
{  
    **int** result = x;  
    **switch**(n)  
    {  
        **case** 0x28:  
        **case** 0x2a:  
            result <<= 3; break;  
        **case** 0x2b:  
            result >>= 3; break;  
        **case** 0x2c:  
            result <<= 3;   
            result -= x;  
        **case** 0x2d:  
            result \*= result;  
        **case** 0x29: //也可以不要  
        **default**:  
            result += 0x11;  
              
    }  
    return result;  
}

中间有一句话没明白，汇编第12行 lea 0x0(%esi), %esi

**3.60**

对于A[R][S][T]，A[i][j][k] 的位置是 A(,i\*S\*T+j\*T+k,4)。

由汇编代码可知：

S\*T = 63;

T = 9;

R\*S\*T = 2772/4;

所以得 R=11, S=7, T=9。

**3.61**

感觉可以用--j，而不是比较j和n。

**int** **var\_prod\_ele**(**int** n, **int** A[n][n], **int** B[n][n], **int** i, **int** k)  
{  
    **int** j = n-1;  
    **int** result = 0;  
    **for**(; j!=-1; --j)  
        result += A[i][j] \* B[j][k];  
    return result;  
}

但是这样得到的结果仍然会使用到存储器。

按下面的代码，循环里面貌似就没有用到存储器。

但是用到了一个常量4，就是增加a的时候，会add 4。

只需要result，a，e，b，4n这五个变量。

**int** **var\_prod\_ele**(**int** n, **int** A[n][n], **int** B[n][n], **int** i, **int** k)  
{  
    **int** result = 0;  
    **int** \*a = &A[i][0];  
    **int** \*b = &B[0][k];  
    **int** \*e = &A[i][n];  
    **for**(;a!=e;)  
    {  
        result += \*a \* \*b;  
        b+=n;  
        a++;  
    }  
    return result;  
}

下面是其汇编代码的循环部分：

edi是4\*n，ebx和edx分别是b和a，esi是e，eax是result。

ecx是用于存储乘法的寄存器。

L4:

movl (%ebx), %ecx

imull (%edx), %ecx

addl %ecx, %eax

addl %edi, %ebx

addl $4, %edx

cmpl %edx, %esi

jneL4

我怎么感觉前面那个程序，编译器应该也会自动优化。。。

**3.62**

M = 76/4 = 19;

i在edi中，j在ecx中;

**int** **transpose**(**int** M, **int** A[M][M])  
{  
    **int** i,j;     
    **for**(i=0; i<M; ++i)  
    {  
        **int** \*a = &A[i][0];  
        **int** \*b = &A[0][i];  
        **for**(j=0; j<i; ++j)  
        {  
            **int** t = \*a;  
            \*a = \*b;  
            \*b = t;  
            ++a;  
            b += M;  
        }  
    }  
}

**3.63**

E1(n)在esi中，esi = 3n。

E2(n)在ebx中，ebx = 4\*E2(n) = 4\*(2n-1)。

所以E2(n) = 2n-1。

**3.64**

这道题比较考验对知识的拓展应用能力。

根据简单的推测，我们可以知道，imull的两个对象是 ebx和edx，最后edx移动到了(eax)中，所以ebx和edx一个是 \*s1.p，一个是s1.v，并且word\_sum的12行的eax是result的prod的地址，也就是result的地址。而eax只在第5行赋值，所以result的地址是在8(%ebp)中的。也就是说，结构体返回值实际上是利用类似参数的变量进行传入（在8(%ebp)），而传入的是返回结构体变量的地址。

所以：

A.

8(%ebp)为result的地址。

12(%ebp)为s1.p。

16(%ebp)为s1.v。

B.栈中的内容如下表，分配的20个字节用黄底展示（每一行为4个字节）

|  |
| --- |
| y |
| x |
| 返回地址 |
| 保存的ebp（也是当前ebp的指向） |
| s2.sum |
| s2.prod |
| s1.v |
| s1.p |
| &s2 （word\_sum的返回值地址） |
|  |

在汇编中，没懂word\_sum 15: ret $4

以及diff 12: subl $4, %esp的意义何在。

可能是为了清除那个result的返回地址。

C.

传递结构体参数就像正常的传值。结构体的每一个变量可以看做是单独的参数进行传入。

D.

返回结构体的通用策略：将返回变量的地址看做第一个参数传入函数。而不是在函数中分配栈空间给一个临时变量，因为eax确实存不下一个结构体，eax充当返回变量的指针的角色。

**3.65**

B取四的倍数的上整数 = 8。

8+4+ (B\*2)取四的倍数的上整数 = 28。

所以B的可选值为8和7。

2\*A\*B取四的上整数为44，所以A\*B的可选值为21和22。

所以 A=3, B=7。

**3.66**

我们用结构体A表示a\_struct。

首先，根据第11和12行，可以得到 CNT\*size(A) = 196。

根据13行，知道 ecx + 4\*edx + 8为 ap->x[ap->idx]的地址。

ecx存储的是bp（地址）。

ap的地址是 bp + 4 + i\*size(A)

我们知道，ap->x[0] 的地址是 bp + 4 + i\*size(A) + pos(x)，pos(x)为结构体A中x的偏移。

那么ap->x[ap->idx] 的地址是 bp + 4 + i\*size(A) + pos(x) + 4\*(ap->idx)。

所以 4\*edx + 8 = 4 + i\*size(A) + pos(x) + 4\*(ap->idx)。

所以，不难猜测，pos(x)=4，也就是说，在A中，首先是idx，再是x数组。

那么，我们看ap->idx在哪里计算过。

到第10行，edx的结果是 7i + bp[4 + 28\*i]，

bp[4 + 28\*i]是什么呢？它很可能是bp中的a[i]的首地址。

我们先这样猜测，于是size(A) = 28，并且bp[4+28\*i]的值为ap->idx。

另一方面：4\*edx = 28\*i + 4\*bp[4+28\*i] = i\*size(A) + 4\*(ap->idx)。

所以，我们的猜想是正确的。

因此，size(A) = 28，里面包含了一个int idx和一个数组int x[6]。

总共有多少个A呢？CNT = 196/size(A) = 7。

**3.67**

A.

e1.p: 0

e1.x: 4

e2.y: 0

e2.next: 4

B.

总共需要8个字节。

C.

不难知道，赋值前后都应该是整数。

edx就是参数up（一个指针）。

最后结果是用eax - (edx)得到的，说明(edx)是整数，即up->\_\_\_ 为整数，肯定是表示的e2.y。

再看看之前的eax，eax是由(eax)所得，说明到第3行后，eax是个指针。

它是由(ecx)得到的，说明ecx在第二行也是个指针。

而ecx是通过\*(up+4)得到的，所以ecx是一个union指针next，即up->e2.next;

到第三行，eax为\*(ecx)，且是一个指针，所以eax在第三行为int\* p，即up->e2.next->e1.p。

所以，赋值符号后面的表达式就为  \*(up->e2.next->e1.p) - up->e2.y

再看看前面。

最终赋值的地址是 ecx+4，而ecx那时候是一个next指针，而(next+4)必须是一个int，也不难推测它是e1.x。因此前面就为 up->e2.next->e1.x。

结果如下：

void **proc**(union ele \*up)  
{  
    up->e2.next->e1.x = \*(up->e2.next->e1.p) - up->e2.y;  
}

**3.68**

版本一：使用getchar

void **good\_echo**()  
{  
    **char** c;  
    **int** x = 0;  
    **while**( x=**getchar**(), x!='\n' && x!=EOF)  
    {  
        **putchar**(x);  
    }  
}

版本二：使用fgets

void **good\_echo**()  
{  
    const **int** BufferSize = 10;  
    **char** s[BufferSize];  
    **int** i;  
    **while**(**fgets**(s, BufferSize, stdin)!=NULL)  
    {  
        **for**(i=0;s[i];++i)  
           **putchar**(s[i]);  
        **if**(i<BufferSize-1) break;  
    }  
  
    return;  
}

两种方法对于EOF好像没效果，就是输入一定字符后不按回车直接按EOF，没能正确输出。

网上查到的资料说，getchar在输入字符后，如果直接按EOF，并不能退出，只能导致新一轮的输入。

需要在最开始输入的时候按，即按了回车之后按。

而fgets就不知道了，不按回车，就不添加0。

**3.69**

**long** **trace**(tree\_ptr tp)  
{  
    **long** ret = 0;  
    **while**(tp != NULL)  
    {  
        ret = tp->val;  
        tp = tp->left;  
    }  
    return ret;  
}

作用是从根一直遍历左子树，找到第一个没有左子树的节点的值。

**3.70**

**long** **traverse**(tree\_ptr tp)  
{  
    **long** v = MAX\_LONG;  
    **if**(tp != NULL)  
    {  
        v = **min**(**traverse**(tp->left), **traverse**(tp->right));

            //Line16 cmovle: if(r12<rax) rax=r12;

        v = **min**(v, tp->v); //Line20 cmovle: if(rax>rbx) rax=rbx;  
    }  
    return v;  
}

当然，如果要用三目条件表达式的话：

**long** **traverse**(tree\_ptr tp)  
{  
    **long** v = MAX\_LONG, rv, lv;  
    **if**(tp != NULL)  
    {  
        lv = **traverse**(tp->left);  
        rv = **traverse**(tp->right);  
        v = lv < rv ? lv : rv ; //Line16 cmovle: if(r12<rax) rax=r12;  
        v = v > tp->v ? tp->v : v ; //Line20 cmovle: if(rax>rbx) rax=rbx;  
    }  
    return v;  
}

函数的目的是找到树的所有节点的值中最小的一个。

[**深入理解计算机系统(第二版) 家庭作业 第四章**](http://blog.csdn.net/zhanyu1990/article/details/9157663)

**4.43**

没有正确执行pushl %esp，pushl %esp是将esp当前的内容入栈。

如果REG是esp，那么代码是先减去了esp，然后将减了4以后的REG移入了esp。

修改：（我只能想到利用其它的寄存器）

movl REG, %eax

subl $4, %esp

movl %eax, (%esp)

**4.44**

也没有正确执行popl %esp，因为popl %esp是将当前栈顶的值放入esp。

如果REG是esp，那么最后得到esp是栈顶值减4之后的值。

movl (%esp), %eax

addl $4, %esp

movl %eax, REG

**4.45**

我没有按书上给的例子写，而是自己写了一个冒泡。

void **bubble**(**int** \*data, **int** count)  
{  
    **if**(count == 0) return;  
    **int** i, j;  
    **int** \*p, \*q;  
    **for**(i=count-1; i!=0; i--){  
        p = data, q = data + 1;  
        **for**(j=0; j!=i; ++j)  
        {  
            **if**( \*p > \*q )  
            {  
                **int** t = \*p;  
                \*p = \*q;  
                \*q = t;  
            }  
            p++, q++;  
        }  
    }  
}

Y86：(movl就没有区分那么细了，因为太麻烦了。。。)

data:#（从地地址往高地址）

$5, $2, $7, $4, $3, $6, $1, $8

movl $8, %ecx

pushl %ecx  #count = 8

movl data, %ecx

pushl %ecx #push data

call bubble

halt

**bubble:**

pushl %ebp

movl %esp, %ebp

pushl %esi

pushl %ebx

pushl %edx

movl 8(%ebp), %edx   #edx == data

movl 12(%ebp), %esi  #esi == count

andl %esi, %esi

je bubbleEnd #count==0

movl $1, %eax

subl %eax, %esi #count--

je bubbleEnd #count==1

**OuterLoop:**

movl %edx, %ecx # p = data (ecx)

pushl %esi# to save one reg

**InnerLoop:**

movl (%ecx), %eax

movl 4(%ecx), %ebx

subl %eax, %ebx

movl 4(%ecx), %ebx

jg NoSwap

movl %eax, %ebx #swap, so ebx is greater

movl 4(%ecx), %eax

**NoSwap:**

movl %eax, (%ecx)

movl %ebx, 4(%ecx)

movl $4, %eax

addl %eax, %ecx

movl $1, %eax

subl%eax, %esi

jne **InnerLoop**

popl %esi

movl $1, %eax

subl %eax, %esi

jne **OuterLoop**

**bubbleEnd:**

popl %edx

popl %ebx

popl %esi

movl %ebp, %esp

popl %ebp

ret

**4.46**

InnerLoop内改成：（edx是循环利用）

movl (%ecx), %edx

**InnerLoop:**

movl %edx, %eax

movl 4(%ecx), %ebx

subl %ebx, %eax  # compare \*p and \*(p+1)

cmovl %ebx, %edx  #edx is max

movl (%ecx), %eax

cmovg %ebx, %eax    #%eax is min

movl %edx, 4(%ecx)

movl %eax, (%ecx)

movl $4, %eax

addl %eax, %ecx

movl $1, %eax

subl %eax, %esi

jne InnerLoop

**4.47**

我们可以明确的是，这条指令完成的任务为，

ebp <- M4[cur\_ebp]

esp <- cur\_ebp + 4

取指阶段 icode:ifun = D:0

        valP <= PC + 1

译码阶段  valB <= R[%ebp]

执行阶段  valE <= valB + 4

访存阶段  valM <= M4[valB]

写回阶段  R[%esp] <= valE

        R[%ebp] <= valM

**4.48**

取指阶段 icode:ifun = M1[PC] = C:0

        rA:rB <= M1[PC+1]

        valC <= M4[PC+2]

        valP <= PC + 6

译码阶段  valB <= R[rB]

执行阶段  valE <= valB + valC

        SetCC

访存阶段

写回阶段  R[rB] <= valE

**4.49**

**4.50**

**取指**

bool need\_regids =

    icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,

                IIRMOVL, IRMMOVL, IMRMOVL, IADDL };

bool need\_valC =

    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IADDL };

**译码和写回**

int srcA = [

    icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;

    icode in { **IPOPL,** IRET } : RESP;

    1 : RNONE; # Don’t need register

];

int srcB = [

icode in { IOPL, IRMMOVL, IMRMOVL, IADDL } : rB;

icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;

icode in { ILEAVE } : REBP;

1 : RNONE; # Don’t need register

];

int dstE = [

icode in { IRRMOVL} && Cnd : rB;

icode in { IIRMOVL, IOPL, IADDL } : rB;

icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE  } : RESP;

1 : RNONE; # Don’t write any register

];

int dstM = [

    icode in { IMRMOVL, IPOPL}:rA;

    icode in { ILEAVE }: REBP;

    1 : RNONE; # Don’t write any register

];

**执行**

int aluA = [

icode in { IRRMOVL, IOPL } : valA;

icode in { IIRMOVL, IRMMOVL, IMRMOVL, IADDL } : valC;

icode in { ICALL, IPUSHL}:-4;

icode in { IRET, IPOPL, ILEAVE}:4;

# Other instructions don’t need ALU

];

int aluB = [

icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,

IPUSHL, IRET, IPOPL, ILEAVE, IADDL } : valB;

icode in { IRRMOVL, IIRMOVL}:0;

# Other instructions don’t need ALU

];

bool set\_cc = icode in { IOPL, IADDL };

**访存**

int mem\_addr = [

icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;

icode in { IPOPL, IRET } : valA;

icode in { ILEAVE } : valB;

# Other instructions don’t need address

];

bool mem\_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };

**4.51 -- 4.56**

感觉蛮麻烦的，不想写啊。。。

**4.57**

A.

发现加载/使用冒险的逻辑公式：

( E\_icode in {IMRMOVL, IPOPL} && E\_dstM in {d\_srcA, d\_srcB})

&&

!(E\_icode == IMRMOVL && D\_icode == IPUSHL);

B.

e\_valA = [ (E\_icode==IPUSH) && (M\_dstM==E\_srcA) : m\_valM;

            1 : E\_valA;

         ];

**4.58**

版本1，在预测正确的情况下执行7条指令，预测错误时执行9条指令并插入一个bubble。

版本2，执行8条指令，但是在外部循环需要多执行3条指令,否则就需要多用一个寄存器。

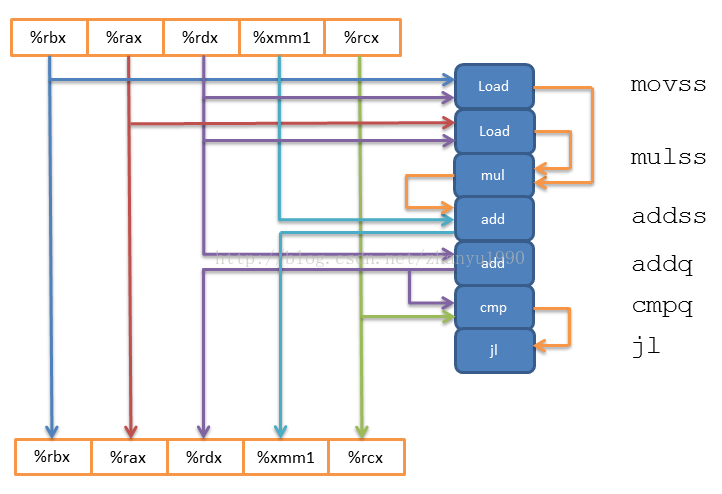
暂时没有想到好的办法。

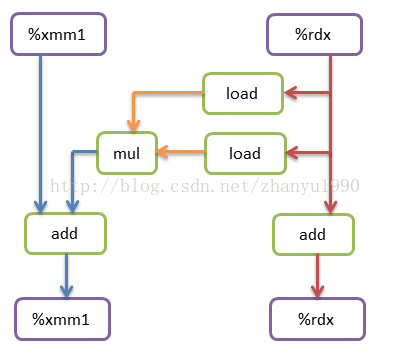
光从内循环看来，版本2平均执行次数比版本1要少，因为可以假设预测错误的概率是50%。

[**深入理解计算机系统(第二版) 家庭作业 第五章**](http://blog.csdn.net/zhanyu1990/article/details/9205385)

**5.15**

A.





关键路径是%xmm1更新路径上的加法。

B. CPE下界是浮点加法的延迟。

C. 两个load操作的吞吐量界限。（我觉得是2.00）

D. 因为乘法不在关键路径上，乘法也是流水线执行的，其限制因素为吞吐量界限。整个程序的限制因素为最后的浮点数加法的延迟，这个延迟对float和double都是3.00。

书上之前说关键路径，现在其实可以再仔细分析一下(以下属于个人分析)：

把执行指令写出了就明了了。

以整数为例：

相同底色表示这些指令在一个循环内执行，以及同一个循环内的初始值：

所以我觉得整数的时候CPE是2.00，可是为什么是3.00呢？

浮点数的话，延迟是3.00没问题。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 时间线 | xmm1\_add单元 | xmm1 | mul单元发射 | load单元发射 | rdx\_add单元 | rdx的值 |
| 1 |  | 0 |  | load1 | add | +0 |
| 2 |  | 0 |  | load2 |  | +1 |
| 3 |  |  |  | load1 | add | +1 |
| 4 |  |  |  | load2 |  | +2 |
| 5 |  |  |  | load1 | add | +2 |
| 6 |  |  | mul (load延迟4) | load2 | ... |  |
| 7 |  |  |  | ... |  |  |
| 8 |  | 0 | mul |  |  |  |
| 9 | add 整数mul 延迟为3 | 0 |  |  |  |  |
| 10 |  | added 整数加法 延迟为1 | mul |  |  |  |
| 11 | add |  |  |  |  |  |
| 12 |  | added |  |  |  |  |

**5.16**

void **inner4**(vec\_ptr u, vec\_ptr v, data\_t \*dest)  
{  
    **long** **int** i;  
    **int** length = **vec\_length**(u);  
    data\_t \*udata = **get\_vec\_start**(u);  
    data\_t \*vdata = **get\_vec\_start**(v);  
    data\_t sum = (data\_t) 0;  
    **int** limit = length - 2;  
    **for** (i = 0; i < limit; i++) {  
        sum = sum + udata[i] \* vdata[i];  
        sum = sum + udata[i+1] \* vdata[i+1];  
        sum = sum + udata[i+2] \* vdata[i+2];  
    }  
      
    **for**(; i<length; ++i)  
        sum = sum + udata[i] \* vdata[i];  
    \*dest = sum;  
}

A. 因为load吞吐量为1.00，每计算一个值需要两次load，所以CPE不可能低于2.00。

B. 关键路径上仍然有N个浮点加法，所以循环展开并没有改变

**5.17**

A. load执行单元的吞吐量

B. IA32可用寄存器实际只有6个，而三路展开需要i, limit, udata, vdata，以及存储udata[i], vdata[i]的寄存器，所以肯定有些循环变量会溢出到寄存器，这会影响效率。（至于为什么是2.67以及为什么四路展开的整数会是2.33，还不是很清楚）。

**5.18**

void **inner4**(vec\_ptr u, vec\_ptr v, data\_t \*dest)  
{  
    **long** **int** i;  
    **int** length = **vec\_length**(u);  
    data\_t \*udata = **get\_vec\_start**(u);  
    data\_t \*vdata = **get\_vec\_start**(v);  
    data\_t sum = (data\_t) 0;  
    **int** limit = length - 2;  
    **for** (i = 0; i < limit; i++) {  
        **int** x1 = udata[i] \* vdata[i];  
        **int** x2 = udata[i+1] \* vdata[i+1];  
        **int** x3 = udata[i+2] \* vdata[i+2];  
        sum = sum + (x1 + x2 + x3);  
    }  
    **for**(; i<length; ++i)  
        sum = sum + udata[i] \* vdata[i];  
    \*dest = sum;  
}

**5.19**

void \***optimized\_memset**(void \*s, **int** c, size\_t n)  
{  
    unsigned **int** K = **sizeof**(unsigned **long**);  
    unsigned **char** \*schar = (unsigned **char**\*)s;  
    unsigned **long** \*lchar;  
    unsigned **long** fill = 0;  
    **int** i = 0;  
    **for**(i = 0; i < K; i++)  
        fill += (c&0xff) << (i<<3);  
  
    // n如果是个负数，会变成一个很大的正数，这应该不需要处理吧？  
    // size\_t应该是unsigned int，n应该不可能是  
    //一般K都是2的整数次幂，也可以用schar&(K-1)来求schar%K  
    **while**((unsigned)schar%K && n)  
    {  
        \*schar++ = (unsigned **char**)c;  
        n--;  
    }  
  
    lchar = (unsigned **long**\*) schar;  
    **while** ( n >= K ) {  
        \*lchar++ = fill;  
        n -= K; //不知道这里如果用++和--会不会影响整体的效率  
    }  
  
    schar = (unsigned **char**\*) lchar;  
    **while**(n) //剩余的n  
    {  
        \*schar++ = (unsigned **char**)c;  
        --n;  
    }  
    **return**s;  
}

**5.20**

**double** **poly\_optimized**(**double** a[], **double** x, **int** degree)  
{  
    **long** **int** i;  
    **double** result = 0;  
    **double** s = 0, powx4 = 1;  
    **double** x2 = x\*x;  
    **double** x4 = x2\*x2;  
    **long** **int** limit = degree-3;  
    **for**(i = 0; i <= limit; i += 4)  
    {  
        **double** v1 = a[i] + a[i+1]\*x;  
        **double** v2 = a[i+2] + a[i+3]\*x;  
        v1 = v1 + v2 \* x2;  
        s = s + v1 \* powx4;  
        powx4 \*= x4;  
    }  
  
    **for**(; i <= degree; ++i)  
    {  
        s += a[i]\*powx4;  
        powx4 \*= x;  
    }  
    return s;  
}

关键路径就是一个浮点数乘法，因此CPE是浮点乘法延迟的1/4，然而每次计算都需要load 4个值，所以CPE还是1.00。

**5.21**

void **psum**(**float** a[], **float** p[], **long** **int** n)  
{  
    **long** **int** i;  
    **int** v = 0;  
    **for**(i=0; i<n-1; i+=2)  
    {  
        **int** v1 = a[i];  
        **int** v2 = a[i+1];  
        v2 = v1 + v2;

        p[i] = v + v1;  
        p[i+1] = v + v2;

        v = v + v2;   
    }  
      
    **for**(; i<n; i++)  
    {  
        v = v + a[i];  
        p[i] = v;  
    }  
}

**5.22**

假设最开始需要100T的时间，那么A需要20T，B需要30T，C需要50T。

将B提到3倍，也就是B需要10T，那么总时间为80T。

将C提到1.5倍，也就是C需要33.3T，那么总时间为83.3T。

所以提高B会使得性能更优。

[**深入理解计算机系统(第二版) 家庭作业 第六章**](http://blog.csdn.net/zhanyu1990/article/details/10262511)

**6.23**

我们可以认为，磁道沿半径方向是均匀分布的。假设半径为r的磁盘总的磁道是K，那么除掉内部的x\*r（磁道数为x\*K），剩下的磁道数为 (1-x)\*K。

那么总容量为  2\*pi\*x\*r\*(1-x)\*K，其中pi，r和K都是常数，那么只剩下x\*(1-x)。

这个函数在x = 0.5的时候取最大。

**6.24**

T\_seek = 3 ms

T\_maxrotate = 60\*1000/12000 ms = 5 ms

T\_avgrotate = 0.5\*T\_maxrotate = 2.5 ms

T\_transfer = T\_maxrotate/500 = 0.01 ms

T = T\_seek + T\_avgrotate + T\_transfer = 5.51 ms

**6.25**

3MB文件，我们假设1MB = 1000KB，而1KB = 1024B（这个好算一些）。

那么3MB文件就有3000个逻辑块（扇区），需要读6个磁道。

T\_maxrotate = 5 ms

T\_transfer = 0.01 ms

A.最好情况是6个磁道都在一个柱面上，只需要寻一次道，而且文件是顺序存储。

T = T\_seek + 0.5\*T\_maxrotate + 6\*T\_maxrotate =  35.5ms

B. 最差的情况 3000\*(T\_seek + 0.5\*T\_maxroate + T\_transfer) = 16530ms

**6.26**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 高速 缓存 | m | C | B | E | S | t | s | b |
| 1 | 32 | 2048 | 4 | 4 | 128 | 23 | 7 | 2 |
| 2 | 32 | 2048 | 4 | 512 | 1 | 30 | 0 | 2 |
| 3 | 32 | 2048 | 8 | 1 | 256 | 21 | 8 | 3 |
| 4 | 32 | 2048 | 8 | 128 | 2 | 28 | 1 | 3 |
| 5 | 32 | 2048 | 32 | 1 | 64 | 21 | 6 | 5 |
| 6 | 32 | 2048 | 32 | 4 | 16 | 23 | 4 | 5 |

**6.27**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 高速 缓存 | m | C | B | E | S | t | s | b |
| 1 | 32 | 8192 | 16 | 1 | 512 | 19 | 9 | 4 |
| 2 | 32 | 4096 | 4 | 4 | 256 | 22 | 8 | 2 |
| 3 | 32 | 4096 | 4 | 8 | 128 | 23 | 7 | 2 |
| 4 | 32 | 2048 | 32 | 4 | 16 | 22 | 4 | 5 |

感觉最后一行答案有错误，C应该是4096，或者t是23，或者E为2。

**6.28\_6.29**

Address\_start = (tag<<5) | (set<<2);

Address\_end = (tag<<5) | (set<<2) | 3;

|  |  |  |
| --- | --- | --- |
|  | 行1 | 行2 |
| 组0 | 0x0120-0x0123 | - |
| 组1 | 0x08A4-0x08A7 | 0x0704-0x0707 |
| 组2 | - | - |
| 组3 | - | 0x064C-0x064F |
| 组4 | 0x18F0-0x18F3 | 0x00B0-0x00B3 |
| 组5 | 0x0E34-0x0E37 | - |
| 组6 | 0x1238-0x123B | - |
| 组7 | - | 0x1BDC-0x1BDF |

**6.30**

b = 2, s = 2, t = 12 - b - s = 8;

A.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CT | CT | CT | CT | CT | CT | CT | CT | CI | CI | CO | CO |

B.

|  |  |  |  |
| --- | --- | --- | --- |
| 操作 | 地址 | 命中？ | 读出的值 |
| 读 | 0x409 | No | - |
| 写 | 0x40A | Yes | - |
| 读 | 0x833 | Yes | 0xD0 |

**6.31**

A. C = E\*B\*S = 128 字节

B. b = 2, s = 3, t = 13-b-s = 8;

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CT | CT | CT | CT | CT | CT | CT | CT | CI | CI | CI | CO | CO |

**6.32**

A.0x0718

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

B.

|  |  |
| --- | --- |
| 参数 | 值 |
| 块偏移CO | 0x00 |
| 索引CI | 0x6 |
| 标记CT | 0x38 |
| 命中？ | Yes |
| 返回值 | 0xFA |

**6.33**

A.0x16EC

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

B.

|  |  |
| --- | --- |
| 参数 | 值 |
| 块偏移CO | 0x00 |
| 索引CI | 0x3 |
| 标记CT | 0xB7 |
| 命中？ | No |
| 返回值 | - |

**6.34**

0x1314-0x1317

0x1794-0x1797

**6.35**

对于写分配的高速缓存，每次写不命中时，需要读取数据到高速缓存中。

该高速缓存只有2个组，对于相同的i,j，src[i][j]和dst[i][j]对应相同的组。

src[0] src[2] 对应组0；

src[1] src[3] 对于组1。

dst同src。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| dst数组 | | | | |
|  | 列0 | 列1 | 列2 | 列3 |
| 行0 | m | h | m | h |
| 行1 | m | m | h | m |
| 行2 | m | h | m | h |
| 行3 | m | m | h | m |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| src数组 | | | | |
|  | 列0 | 列1 | 列2 | 列3 |
| 行0 | m | m | m | m |
| 行1 | m | m | m | m |
| 行2 | m | m | m | m |
| 行3 | m | m | m | m |

**6.36**

缓存能完全容得下两个数组，所以只会出现冷不命中。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| dst数组 | | | | |
|  | 列0 | 列1 | 列2 | 列3 |
| 行0 | m | h | h | h |
| 行1 | m | h | h | h |
| 行2 | m | h | h | h |
| 行3 | m | h | h | h |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| src数组 | | | | |
|  | 列0 | 列1 | 列2 | 列3 |
| 行0 | m | h | h | h |
| 行1 | m | h | h | h |
| 行2 | m | h | h | h |
| 行3 | m | h | h | h |

**6.37**

A.缓存为1024字节，数组大小为2\*256\*4=2048，所以x[0]和x[1]的每一个元素（x[0][i]和x[1][i]）对应的高速缓存是同一个块。

因此，每次都在加载，驱逐。不命中率为100%。

B. 缓存足够大，只有冷不命中，不命中率为1/8 = 12.5%。

C. 这种情况相当于是只有冷不命中，不命中率为12.5%。

D. 不会降低，因为块大小不变时，冷不命中的概率不可能被减小。

E. 会降低，因为一个块的大小增加，冷不命中的频率就降低。

**6.38**

写了个程序进行测试的。

结果如下：

|  |  |  |
| --- | --- | --- |
| 函数 | N=64 | N=60 |
| sumA | 0.25 | 0.25 |
| sumB | 1 | 0.25 |
| sumC | 0.5 | 0.25 |

程序清单：

unsigned calc\_address(unsigned head, int i, int j, int N)  
{  
    return head + i\*N\*4 + j\*4;  
}  
  
unsigned calc\_line(unsigned add, int block\_size, const int line)  
{  
    return (add / block\_size) % line;  
}  
  
int isHit(unsigned cache[], unsigned r, unsigned a, int block\_size)  
{  
    a = a / block\_size \* block\_size;  
    unsigned tmp = cache[r];  
    cache[r] = a;  
    return tmp == a;  
}  
  
void testA(int block\_size, const int line, unsigned add, const int N)  
{  
    unsigned cache[line];  
    memset(cache, -1, sizeof(cache));  
    int hit = 0, miss = 0;  
  
    for(int i=0; i<N; ++i)  
    for(int j=0; j<N; ++j)  
    {  
        unsigned address = calc\_address(add, i, j, N);  
        unsigned r = calc\_line(address, block\_size, line);  
        hit += isHit(cache, r, address, block\_size);  
    }  
    miss = N\*N - hit;  
    cout << "sumA " << hit << " " << miss << endl;  
    cout << (double)miss/(N\*N) << endl;  
}  
  
void testB(int block\_size, const int line, unsigned add, const int N)  
{  
    unsigned cache[line];  
    memset(cache, -1, sizeof(cache));  
    int hit = 0, miss = 0;  
  
    for(int j=0; j<N; ++j)  
    for(int i=0; i<N; ++i)  
    {  
        unsigned address = calc\_address(add, i, j, N);  
        unsigned r = calc\_line(address, block\_size, line);  
        hit += isHit(cache, r, address, block\_size);  
    }  
    miss = N\*N - hit;  
    cout << "sumB " << hit << " " << miss << endl;  
    cout << (double)miss/(N\*N) << endl;  
}  
  
  
void testC(int block\_size, const int line, unsigned add, const int N)  
{  
    unsigned cache[line];  
    memset(cache, -1, sizeof(cache));  
    int hit = 0, miss = 0;  
  
    for(int j=0; j<N; j+=2)  
    for(int i=0; i<N; i+=2)  
    {  
        unsigned a, r;  
        a = calc\_address(add, i, j, N);  
        r = calc\_line(a, block\_size, line);  
        hit += isHit(cache, r, a, block\_size);  
  
        a = calc\_address(add, i+1, j, N);  
        r = calc\_line(a, block\_size, line);  
        hit += isHit(cache, r, a, block\_size);  
  
        a = calc\_address(add, i, j+1, N);  
        r = calc\_line(a, block\_size, line);  
        hit += isHit(cache, r, a, block\_size);  
  
        a = calc\_address(add, i+1, j+1, N);  
        r = calc\_line(a, block\_size, line);  
        hit += isHit(cache, r, a, block\_size);  
    }  
    miss = N\*N - hit;  
    cout << "sumC " << hit << " " << miss << endl;  
    cout << (double)miss/(N\*N) << endl;  
}  
  
void testHit()  
{  
    const unsigned start\_address = 0x08000000;  
    const int cache\_size = 4\*1024;  
    const int block\_size = 16;  
  
    int line = cache\_size / block\_size;  
    int N = 64;  
    cout << N << endl;  
    testA(block\_size, line, start\_address, N);  
    testB(block\_size, line, start\_address, N);  
    testC(block\_size, line, start\_address, N);  
    cout << "-----" << endl;  
    N = 60;  
    cout << N << endl;  
    testA(block\_size, line, start\_address, N);  
    testB(block\_size, line, start\_address, N);  
    testC(block\_size, line, start\_address, N);  
}

**6.39**

A. 写总数为 16\*16\*4 = 1024

B C. 这种情况只有冷不命中，一个block能存下16个int，不命中率为1/16。不命中总数为64。

**6.40**

A. 写总数为 16\*16\*4 = 1024

B C.

E = 1024/64 = 16，每4行才能容得下数组的一行，每次j在变化时，都会不断驱逐掉之前的缓存。所以，只有循环内是可以命中的。不命中率为1/4，不命中总数为256。

**6.41**

A. 写总数为 16\*16\*4 = 1024

B.

一行能保持4个square结构体的值，所以第一个双循环不命中率为1/4。

第二个双循环，不命中率为1/12。

总的不命中次数为 16\*16/4 + 16\*16\*3/12 = 16\*16/2 = 128。

C. 不命中率为 1/8。

前的缓存。所以，只有循环内是可以命中的。不命中率为1/4，不命中总数为256。

**6.42**

每行只有4个字节，那只可能是循环中后三个命中，第一个不命中，不命中率为0.25。

感觉这个题目设定应该不是每行4个字节，64个字节也好啊。

**6.43**

和上题一样，每次要写一个char，不命中率仍然为0.25。

**6.44**

每次写一个int，每次都不会命中。不命中率为100%。

**6.45**

固定stride，看吞吐量随着工作集大小改变的情况。

那个mountain的程序不会用，心想还是先看完书，所以也没花功夫看。

**6.46 6.47暂时都没想到**

可能的想法是，将两层循环都进行二展开，这样至少可以：

dst[j\*dim+i] = src[i\*dim+j];

dst[(j+1)\*dim+i] = src[i\*dim+j+1];

dst[j\*dim+i+1] = src[(i+1)\*dim+j];

dst[(j+1)\*dim+i+1] = src[(i+1)\*dim+j+1];

只要不是遇到缓存抖动，不命中率最多就是50%，还是会比较低的。

如果加入运算，就可以想办法同时更新 dst[i][j], dst[j][i]。

但是这样读不命中就可能高了。

等待高人回答吧！

[**深入理解计算机系统(第二版) 家庭作业 第七章**](http://blog.csdn.net/zhanyu1990/article/details/10262725)

**7.6**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 符号 | swap.o 条目 | 符号类型 | 定义符号的模块 | 所属的节 |
| buf | Yes | extern | main.o | .data |
| bufp0 | Yes | global | swap.o | .data |
| bufp1 | Yes | local | swap.o | .bss |
| swap | Yes | global | swap.o | .text |
| temp | No | - | - | - |
| incr | Yes | local | swap.o | .text |
| count | Yes | local | swap.o | .data |

**7.7**

可以将定义double x;移入f()内部。

也可以在double x前加上static。

**7.8**

A.

a)REF(main.1)-->DEF(main.1)

b)REF(main.2)-->DEF(main.2)

B. UNKNOWN

C. ERROR

**7.9**

p2中的main是弱符号，链接后的引用是foo6.c中的函数main。

main的第一个字节就是0x55（pushl %ebp），所以能打印出0x55。

**7.10**

A. gcc p.o libx.a p.o

B. gcc p.o libx.a liby.a libx.a

C. gcc p.o liby.a libx.a liby.a libz.a

**7.11**

未初始化的全局变量在目标文件中是没有分配存储空间，而在加载之后，会需要一些空间。

所以这些空间可能是为.bss中的符号准备的。但具体为什么是0x104和0xe8就不清楚了。

**7.12**

|  |  |  |
| --- | --- | --- |
| 行号 | 运行时存储器地址 | 值 |
| 15(bufp0) | 0x080483CB | 0x0804945C |
| 16(buf[1]) | 0x080483D0 | 0x08049458 |
| 18(bufp1) | 0x080483D8 | 0x08049548 |
| 18(buf[1]) | 0x080483DC | 0x08049458 |
| 23(bufp1) | 0x080483E7 | 0x08049548 |

**7.13**

这道题可以推测出来哪些代码需要重定位。

当然，我们假设所有的符号都已经有了运行时地址。

.text中有3个地方有重定位。

第12行是调用p3，返回值被放在了edx中。

14行要将xp的值放入eax。这里的0x0其实应该重定位为xp的地址，所以这应该是一个绝对引用。

接着add (%eax),%edx，也就是完成了\*xp + p3()。

然后再push %edx，作为参数压入栈。

第17行是调用p2()。最后返回。

所以.text中有三个地方需要重定位。

是相对或者绝对的话，我不太清楚（猜测的）。

|  |  |  |  |
| --- | --- | --- | --- |
| 行号 | 节偏移 | 重定位类型 | 符号名字 |
| 12 | 0x12 | 相对 | p3 |
| 14 | 0x19 | 绝对 | xp |
| 17 | 0x21 | 相对 | p2 |

在.data节中，第一个x是不需要重定位的，第二个xp的值需要重定位为x的地址。

|  |  |  |  |
| --- | --- | --- | --- |
| 行号 | 节偏移 | 重定位类型 | 符号名字 |
| 12 | 0x04 | 绝对 | x |

**7.14**

汇编代码其实比较好理解。eax是val，最开始val-100和5比较，如果大于5，就直接到16行去+6。

否则根据跳转表跳到相应的地址（edx\*4+0）。

所以，重定位时，需要将所有的ja和jmp指令的目的地修改为对应的地址。

（我不太明白这里该怎么重定位，因为代码中不是四个字节的地址（第7,10,12,15行），所以可能是相对的，如果是相对的就不需要重定位了）

需要注意的是第8行，jmp \*0x0(,%edx,4)这一句。

这里的0x0是指.rodata的地址，这是肯定需要重定位的。

|  |  |  |  |
| --- | --- | --- | --- |
| 行号 | 节偏移 | 重定位类型 | 符号名字 |
| 8 | 0x11 | 绝对 | .rodata |

在.rodata中，所有的跳转表都需要重定位到相应代码的位置。

符号名字就不知道了。

|  |  |  |  |
| --- | --- | --- | --- |
| 行号 | 节偏移 | 重定位类型 | 符号名字 |
| .rodata1 | 0x0,0x4,0x8,0xc,0x10,0x14 | 绝对 | ? |

**7.15**

用的是windows系统，不太清楚这些怎么用。

好像连LDD都没有。

而且在mingw中也没找到libc.a，倒是有libstdc++。

[**深入理解计算机系统(第二版) 家庭作业 第八章**](http://blog.csdn.net/zhanyu1990/article/details/10896987)

**8.9**

|  |  |
| --- | --- |
| 进程对 | 是否并发 |
| AB | No |
| AC | Yes |
| AD | Yes |
| BC | Yes |
| BC | Yes |
| CD | Yes |

**8.10**

A. 调用一次，返回两次： fork

B. 调用一次，从不返回： execve, longjmp

C. 调用一次，返回一次或者多次： setjmp

**8.11**

4行

**8.12**

8行

**8.13**

->x=2

->x=4->x=3

满足这种拓扑即可。

**8.14**

主进程只打印一行。

主进程的直接子进程会打印一行，子进程的子进程又打印一行。

所以是3行。

**8.15**

这里的子进程不是exit，而是return，说明两个子进程都要到回到main函数去打印那里的hello。所以是5行。

**8.16**

输出counter = 2，因为全局变量也是复制的，而不是共享的。

**8.17**

满足

Hello--->1->Bye

      \        \

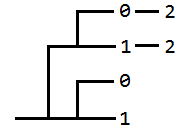
       \--->0---->2-->Bye

这种拓扑即可。

|  |  |  |
| --- | --- | --- |
| Hello  1  Bye  0  2  Bye | Hello  1  0  Bye  2  Bye | Hello 0 1 Bye 2 Bye |

**8.18**

画一下进程图就可以知道。



所以ACE是可能的。

**8.19**

总共会输出2^n行。

**8.20**

其实这道题不需要你在环境变量里抓取name，只要用execve。

**int** **main**(**int** argc, **char**\* args[])  
{  
    **execve**("/bin/ls", args, environ); //没有错误处理，注意环境变量  
    return 0;  
}

**8.21**

abc或者bac。c肯定在a和b之后。

**8.22**

不太清楚/bin/sh -c command中，-c以及command的意义。

所以把-c和command都当做/bin/sh的参数吧。

但是如果command有空格怎么办？

**int** **mysystem**(**char** \*command)  
{  
    **int** status;  
    **char** \*argv[4];  
    **char** \*a0 = "sh";  
    **char** \*a1 = "-c";  
    **if**( **fork**()==0 ) /\*子进程\*/  
    {   
        argv[0] = a0;  
        argv[1] = a1;  
        argv[2] = command;  
        argv[3] = NULL;  
        **execve**("/bin/sh", args, environ);  
        return -1; //执行异常  
    }  
    **else**{ /\*父进程\*/  
        **if**( **wait**(&status) > 0)  
        {  
            **if**(**WIFEXITED**(status) != 0)  
                return **WEXITSTATUS**(status);  
            **else** return status;  
        }  
        **else** return -1; //wait异常  
    }  
}

**8.23**

一个可能的原因是，在第一个信号发给父进程之后，父进程进入handler，并且阻塞了SIGUSR2，第二个信号依然可以发送，然而，之后的3个信号便会被抛弃了。因为是连续发送，所以很可能是没等上下文切换，这5个信号就同时发送了。所以只有2个信号被接收。

**8.24**

#include "csapp.h"  
#**define** N 2  
  
**int** **main**()  
{  
    **int** status, i;  
    pid\_t pid;  
    **char** errorInfo[128];  
    /\* Parent creates N children \*/  
    **for**(i=0;i<N;i++)  
        **if** ((pid = **Fork**()) == 0) /\* Child \*/  
            **exit**(100+i);  
  
    /\* Parent reaps N children in no particular order \*/  
    **while** ((pid = **waitpid**(-1, &status, 0)) > 0) {  
        **if** (**WIFEXITED**(status))  
            **printf**("child %d terminated normally with exit status=%d\n",  
                    pid, **WEXITSTATUS**(status));  
        **else** **if**(**WIFSIGNALED**(status))  
        {  
            //为什么只在写只读文本时才报错？下面的程序对所有异常退出都会提示  
            **printf**("child %d terminated by signal %d: ",   
                pid, **WTERMSIG**(status) );  
            **psignal**(**WTERMSIG**(status), errorInfo); //psignal会打印sig的信息  
        }  
    }  
    /\* The only normal termination is if there are no more children \*/  
    **if** (errno != ECHILD)  
        **unix\_error**("waitpid error");  
    **exit**(0);  
}

**8.25**

fgets的定义如下：

char \*fgets(char \*buf, int bufsize, FILE \*stream);

参数：

\*buf: 字符型指针，指向用来存储所得数据的地址。

bufsize: 整型数据，指明buf指向的字符数组的大小。

\*stream: 文件结构体指针，将要读取的文件流。

这个应该是用alarm发送信号给自己，然后在信号处理程序里面做文章。

显然，在tfgets里一开始需要调用fgets。然而，因为五秒时间到了，fgets还没有返回，所以我们必须在处理程序里直接跳转到某个地方进行tfgets的NULL返回。这就需要用到非本地跳转。

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <signal.h>  
#include <setjmp.h>

sigjmp\_buf env;  
void **tfgets\_handler**(**int** sig)  
{  
    **signal**(SIGALRM, SIG\_DFL);  
    **siglongjmp**(env, 1);  
}  
  
**char** \***tfgets**(**char** \*buf, **int** bufsize, FILE \*stream)  
{  
    static const **int** TimeLimitSecs = 5;  
    **signal**(SIGALRM, tfgets\_handler)  
    **alarm**(TimeLimitSecs);  
    **int** rc = **sigsetjmp**(env, 1);  
    **if**(rc == 0) return **fgets**(buf, bufsize, stream);  
    **else** return NULL; //alarm，time out  
}

注意这里如果用非sig版本，程序将不会正确运行。（尝试了只会time out一次）。

sig版本是可以供信号处理程序使用的版本。

**8.26**

暂时没有做。感觉加上JID的话，得自己写程序来管理每个job。

1）需要定义一个struct结构体数组，JID是1，那么就是数组的第一个。

结构体中有pid和状态。状态分空，运行，停止等等。

难点是JID和pid的互相查找，有点像hash。

如果不用hash，就是枚举数组寻找pid。所以，结构体数组大小要先设定（最多运行的jobs数目），不能太大，否则会增加复杂度。

2）可以写一个addjob和deletejob函数，以pid为参数。

当然，对于后台运行的进程，要注意竞争。

3）需要写handler函数，当子进程运行完毕或终止，需要用handler函数来回收。

还需要写一些handler来实现发送消息的功能，比如ctrl-z和ctrl-c。

4）jobs就是列出数组中状态不为空的进程。

可以写一个searchJID(pid\_t pid)函数，根据某个pid来查询JID，当然，这个是可以优化的，当前可以只写枚举。

5）fg和bg倒是比较简单，给子进程发送那些消息，让那些消息重新execve。这个好像也可以用非本地跳转来实现。

[**深入理解计算机系统(第二版) 家庭作业 第九章**](http://blog.csdn.net/zhanyu1990/article/details/18502257)

9.11

A.虚拟地址0x027c

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

B.地址翻译

|  |  |
| --- | --- |
| 参数 | 值 |
| VPN | 0x09 |
| TLB索引 | 0x01 |
| TLB标记 | 0x02 |
| TLB命中 | No |
| 缺页 | No |
| PPN | 0x17 |

C.物理地址格式

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

D.物理地址引用

|  |  |
| --- | --- |
| 参数 | 值 |
| 字节偏移 | 0x0 |
| 缓存索引 | 0xF |
| 缓存标记 | 0x17 |
| 缓存命中 | No |
| 返回缓存字节 | - |

9.12

A.虚拟地址0x03a9

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

B.地址翻译

|  |  |
| --- | --- |
| 参数 | 值 |
| VPN | 0x0E |
| TLB索引 | 0x02 |
| TLB标记 | 0x03 |
| TLB命中 | No |
| 缺页 | No |
| PPN | 0x11 |

C.物理地址格式

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

D.物理地址引用

|  |  |
| --- | --- |
| 参数 | 值 |
| 字节偏移 | 0x1 |
| 缓存索引 | 0xA |
| 缓存标记 | 0x11 |
| 缓存命中 | No |
| 返回缓存字节 | - |

9.13

A.虚拟地址0x0040

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

B.地址翻译

|  |  |
| --- | --- |
| 参数 | 值 |
| VPN | 0x01 |
| TLB索引 | 0x01 |
| TLB标记 | 0x00 |
| TLB命中 | No |
| 缺页 | Yes |
| PPN | - |

9.14

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zhanyu1990/article/details/18502257)[在CODE上查看代码片](https://code.csdn.net/snippets/162861)



**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zhanyu1990/article/details/18502257)[在CODE上查看代码片](https://code.csdn.net/snippets/162861)

1. #include <sys/types.h>
2. #include <sys/stat.h>
3. #include <fcntl.h>
4. #include <unistd.h>
5. #include <sys/mman.h>
6. **int** main()
7. {
8. **int** fd;
9. **char** \*start;
10. fd = open("hello.txt", O\_RDWR, 0); //打开文件
11. start = mmap(NULL, 1, PROT\_WRITE, MAP\_SHARED, fd, 0);
12. close(fd);
13. if(start == MAP\_FAILED) return -1;//判断是否映射成功
14. (\*start) = 'J';
15. munmap(start, 1);
16. return 0;
17. }

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

int main()

{

int fd;

char \*start;

fd = open("hello.txt", O\_RDWR, 0); //打开文件

start = mmap(NULL, 1, PROT\_WRITE, MAP\_SHARED, fd, 0);

close(fd);

if(start == MAP\_FAILED) return -1;//判断是否映射成功

(\*start) = 'J';

munmap(start, 1);

return 0;

}

注：代码经过测试，必须使用MAP\_SHARED，不能使用MAP\_PRIVATE，文件异常并未检测。

9.15

|  |  |  |
| --- | --- | --- |
| 请求 | 块大小 | 块头部 |
| malloc(3) | 8 | 0x9 |
| malloc(11) | 16 | 0x11 |
| malloc(20) | 24 | 0x19 |
| malloc(21) | 32 | 0x21 |

9.16

怎么感觉答案都是16呢。。。

空闲块至少是16字节。已分配块不需要pred和succ，所以这八个字节可以装数据，加上头和脚，就是16字节，而且满足单字双字对齐。

求鉴定。

9.17

显然，我们需要修改find\_fit函数（之前的版本在practise习题中写过，书后有答案）。

代码可以参考9.18。

为此需要先定义一个全局的cur\_point指针，表示上次搜索之后指向哪个块（有效载荷首地址）。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zhanyu1990/article/details/18502257)[在CODE上查看代码片](https://code.csdn.net/snippets/162861)

1. static void \*find\_fit(**size\_t** asize)
2. {
3. void \*bp = cur\_point;
4. do{
5. if( !GET\_ALLOC(HDRP(bp)) && (asize <= GET\_SIZE(HDRP(bp))) )
6. return cur\_point = bp;
7. bp = (GET\_SIZE(HDRP(bp)) == 0) ? heap\_listp : NEXT\_BLKP(bp);
8. }while(bp != cur\_point);
10. return NULL;
11. }

static void \*find\_fit(size\_t asize)

{

void \*bp = cur\_point;

do{

if( !GET\_ALLOC(HDRP(bp)) && (asize <= GET\_SIZE(HDRP(bp))) )

return cur\_point = bp;

bp = (GET\_SIZE(HDRP(bp)) == 0) ? heap\_listp : NEXT\_BLKP(bp);

}while(bp != cur\_point);

return NULL;

}

另外，需要释放cur\_point指向的指针时，它可能和前面的空闲块合并，我们应该将cur\_point指向前一个空闲块首地址。在coalesce()函数中需要做如下修改：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zhanyu1990/article/details/18502257)[在CODE上查看代码片](https://code.csdn.net/snippets/162861)

1. else if (!prev\_alloc && next\_alloc) { /\* Case 3 \*/
2. size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));
3. PUT(FTRP(bp), PACK(size, 0));
4. PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));
5. if(cur\_point == bp) cur\_point = PREV\_BLKP(bp);
6. bp = PREV\_BLKP(bp);
7. }
8. else { /\* Case 4 \*/
9. size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) +
10. GET\_SIZE(FTRP(NEXT\_BLKP(bp)));
11. PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));
12. PUT(FTRP(NEXT\_BLKP(bp)), PACK(size, 0));
13. if(cur\_point == bp) cur\_point = PREV\_BLKP(bp);
14. bp = PREV\_BLKP(bp);
15. }

else if (!prev\_alloc && next\_alloc) { /\* Case 3 \*/

size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));

PUT(FTRP(bp), PACK(size, 0));

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));

if(cur\_point == bp) cur\_point = PREV\_BLKP(bp);

bp = PREV\_BLKP(bp);

}

else { /\* Case 4 \*/

size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) +

GET\_SIZE(FTRP(NEXT\_BLKP(bp)));

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));

PUT(FTRP(NEXT\_BLKP(bp)), PACK(size, 0));

if(cur\_point == bp) cur\_point = PREV\_BLKP(bp);

bp = PREV\_BLKP(bp);

}

9.18

需要考虑四个问题：

1. 初始化的时候，序言块和结尾块是怎么样的。

    序言块八字节8|0x11。

    结尾块四字节0|0x11。

2. 为堆申请更多空间的时候（sbrk），如何更改结尾块

    记录最后一个块的alloc。

    结尾块向后延伸申请的空间，并将刚多出的空间作为一个空闲块。设置为size|(alloc<<1)。再合并该空闲块。这里如何合并呢？需要判断最后一块是否已分配，可通过epilogue来判断。

3. 某个空闲块匹配的时候，如何设置头和下一块的头。

    我们基于以下假设：某个空闲块匹配，上一个和下一个一定不是空闲块（否则可以合并）。

    所以头部就设置为（asize|0x011)。

    如果要分割，则下一块的头部设置为（size-asize|0x010），不用合并，因为再下一块肯定是已经分配。

4. 释放某个块时，如何合并。

    检查头部，alloc\_prev = 上一块是否分配

    检查下一个块的头部，alloc\_next = 下一个块是否分配。

    再根据那四种情况分别设置。

    最后，如果下一块已分配，则需要将下一块头部设置为(原头部&(~0x010))。

另外，在mm\_malloc中，分配多大的块也要发生相应的变化，因为现在最小块大小可以是DSIZE，而不是2\*DSIZE。

以下是我的完整代码：代码并没有进行严格测试，如果有问题可以一起讨论。

memlib.c

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zhanyu1990/article/details/18502257)[在CODE上查看代码片](https://code.csdn.net/snippets/162861)

1. #include <stdlib.h>
2. #include <stdio.h>
3. #include <errno.h>
5. #define MAX\_HEAP (1<<24)
6. /\* Private global variables \*/
7. static **char** \*mem\_heap; /\* Points to first byte of heap \*/
8. static **char** \*mem\_brk; /\* Points to last byte of heap plus 1 \*/
9. static **char** \*mem\_max\_addr; /\* Max legal heap addr plus 1\*/
11. /\*
12. \* mem\_init - Initialize the memory system model
13. \*/
14. void mem\_init(void)
15. {
16. mem\_heap = (**char** \*)malloc(MAX\_HEAP);
17. mem\_brk = (**char** \*)mem\_heap;
18. mem\_max\_addr = (**char** \*)(mem\_heap + MAX\_HEAP);
19. }
21. /\*
22. \* mem\_sbrk - Simple model of the sbrk function. Extends the heap
23. \* by incr bytes and returns the start address of the new area. In
24. \* this model, the heap cannot be shrunk.
25. \*/
26. void \*mem\_sbrk(**int** incr)
27. {
28. **char** \*old\_brk = mem\_brk;
30. if ( (incr < 0) || ((mem\_brk + incr) > mem\_max\_addr)) {
31. errno = ENOMEM;
32. fprintf(stderr, "ERROR: mem\_sbrk failed. Ran out of memory...\n");
33. return (void \*)-1;
34. }
35. mem\_brk += incr;
36. return (void \*)old\_brk;
37. }
39. void mem\_end(void)/\*added by zhanyu\*/
40. {
41. free(mem\_heap);
42. }

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#define MAX\_HEAP (1<<24)

/\* Private global variables \*/

static char \*mem\_heap; /\* Points to first byte of heap \*/

static char \*mem\_brk; /\* Points to last byte of heap plus 1 \*/

static char \*mem\_max\_addr; /\* Max legal heap addr plus 1\*/

/\*

\* mem\_init - Initialize the memory system model

\*/

void mem\_init(void)

{

mem\_heap = (char \*)malloc(MAX\_HEAP);

mem\_brk = (char \*)mem\_heap;

mem\_max\_addr = (char \*)(mem\_heap + MAX\_HEAP);

}

/\*

\* mem\_sbrk - Simple model of the sbrk function. Extends the heap

\* by incr bytes and returns the start address of the new area. In

\* this model, the heap cannot be shrunk.

\*/

void \*mem\_sbrk(int incr)

{

char \*old\_brk = mem\_brk;

if ( (incr < 0) || ((mem\_brk + incr) > mem\_max\_addr)) {

errno = ENOMEM;

fprintf(stderr, "ERROR: mem\_sbrk failed. Ran out of memory...\n");

return (void \*)-1;

}

mem\_brk += incr;

return (void \*)old\_brk;

}

void mem\_end(void)/\*added by zhanyu\*/

{

free(mem\_heap);

}

mm.c

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zhanyu1990/article/details/18502257)[在CODE上查看代码片](https://code.csdn.net/snippets/162861)

1. #include <stdio.h>
3. /\* Basic constants and macros \*/
4. #define WSIZE 4 /\* Word and header/footer size (bytes) \*/
5. #define DSIZE 8 /\* Double word size (bytes) \*/
6. #define CHUNKSIZE (1<<12) /\* Extend heap by this amount (bytes) \*/
8. #define MAX(x, y) ((x) > (y)? (x) : (y))
10. /\* Pack a size and allocated bit into a word \*/
11. //#define PACK(size, alloc) ((size) | (alloc))
12. #define PACK(size, prev\_alloc, alloc) ((size)|(prev\_alloc<<1)|(alloc))
14. /\* Read and write a word at address p \*/
15. #define GET(p) (\*(unsigned int \*)(p))
16. #define PUT(p, val) (\*(unsigned int \*)(p) = (val))
18. /\* Read the size and allocated fields from address p \*/
19. #define GET\_SIZE(p) (GET(p) & ~0x7)
20. #define GET\_ALLOC(p) (GET(p) & 0x1)
21. #define GET\_PREVALLOC(p) ((GET(p) & 0x2)>>1)
23. /\* Given block ptr bp, compute address of its header and footer \*/
24. #define HDRP(bp) ((char \*)(bp) - WSIZE)
25. #define FTRP(bp) ((char \*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE)
27. /\* Given block ptr bp, compute address of next and previous blocks \*/
28. #define NEXT\_BLKP(bp) ((char \*)(bp) + GET\_SIZE(((char \*)(bp) - WSIZE)))
29. #define PREV\_BLKP(bp) ((char \*)(bp) - GET\_SIZE(((char \*)(bp) - DSIZE)))


33. typedef unsigned **size\_t**;
35. **int** mm\_init(void);
36. static void \*extend\_heap(**size\_t** words);
37. void \*mm\_malloc(**size\_t** size);
38. void mm\_free(void \*bp);
39. static void \*coalesce(void \*bp);
40. static void \*find\_fit(**size\_t** asize);
41. static void place(void \*bp, **size\_t** asize);
42. void mm\_end(); /\*added by zhanyu\*/
44. extern void \*mem\_sbrk(**int** incr); //memlib.c
45. extern void mem\_init(void); //memlib.c
46. extern void mem\_end(void); //memlib.c
48. void \*heap\_listp;
50. **int** mm\_init(void)
51. {
52. mem\_init(); /\* added by zhanyu \*/
53. /\* Create the initial empty heap \*/
54. if ((heap\_listp = mem\_sbrk(4\*WSIZE)) == (void \*)-1) return -1;
55. PUT(heap\_listp, 0); /\* Alignment padding \*/
56. PUT(heap\_listp + (1\*WSIZE), PACK(DSIZE, 1, 1)); /\* Prologue header \*/
57. PUT(heap\_listp + (2\*WSIZE), PACK(DSIZE, 1, 1)); /\* Prologue footer \*/
58. PUT(heap\_listp + (3\*WSIZE), PACK(0, 1, 1)); /\* Epilogue header \*/
59. heap\_listp += (2\*WSIZE);
61. /\* Extend the empty heap with a free block of CHUNKSIZE bytes \*/
62. if (extend\_heap(CHUNKSIZE/WSIZE) == NULL) return -1;
63. return 0;
64. }
66. static void \*extend\_heap(**size\_t** words)
67. {
68. **char** \*bp;
69. **size\_t** size;
70. **int** prev\_alloc;
72. /\* Allocate an even number of words to maintain alignment \*/
73. size = (words % 2) ? (words+1) \* WSIZE : words \* WSIZE;
74. if ((**long**)(bp = mem\_sbrk(size)) == -1) return NULL;
75. prev\_alloc = GET\_PREVALLOC(HDRP(bp));
76. /\* Initialize free block header/footer and the epilogue header \*/
77. PUT(HDRP(bp), PACK(size, prev\_alloc, 0)); /\* Free block header \*/
78. PUT(FTRP(bp), PACK(size, prev\_alloc, 0)); /\* Free block footer \*/
79. PUT(HDRP(NEXT\_BLKP(bp)), PACK(0, 0, 1)); /\* New epilogue header, the prev\_alloc is 0\*/
81. /\* Coalesce if the previous block was free \*/
82. return coalesce(bp);
83. }
85. void mm\_free(void \*bp)
86. {
87. **size\_t** size = GET\_SIZE(HDRP(bp));
88. **size\_t** nsize = GET\_SIZE(HDRP(NEXT\_BLKP(bp)));
89. **int** prev\_alloc = GET\_PREVALLOC(HDRP(bp));
90. **int** next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));
91. PUT(HDRP(bp), PACK(size, prev\_alloc, 0));
92. PUT(FTRP(bp), PACK(size, prev\_alloc, 0));
93. PUT(HDRP(NEXT\_BLKP(bp)), PACK(nsize, 0, next\_alloc));
94. if(!next\_alloc) PUT(FTRP(NEXT\_BLKP(bp)), PACK(nsize, 0, next\_alloc));
95. coalesce(bp);
96. }
98. static void \*coalesce(void \*bp)
99. {
100. **size\_t** prev\_alloc = GET\_PREVALLOC(HDRP(bp));
101. **size\_t** next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));
102. **size\_t** size = GET\_SIZE(HDRP(bp));
104. if (prev\_alloc && next\_alloc) { /\* Case 1 \*/
105. return bp;
106. }
108. else if (prev\_alloc && !next\_alloc) { /\* Case 2 \*/
109. size += GET\_SIZE(HDRP(NEXT\_BLKP(bp)));
110. PUT(HDRP(bp), PACK(size, 1, 0));
111. PUT(FTRP(bp), PACK(size, 1, 0));
112. /\*\*then the next block is surely nonempty and it's prev\_alloc is 0\*/
113. }
115. else if (!prev\_alloc && next\_alloc) { /\* Case 3 \*/
116. /\*the prev prev block is surely nonempty\*/
117. size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));
118. PUT(FTRP(bp), PACK(size, 1, 0));
119. PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 1, 0));
120. bp = PREV\_BLKP(bp);
121. }
123. else { /\* Case 4 \*/
124. /\*the prev prev block is surely nonempty\*/
125. size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) +
126. GET\_SIZE(FTRP(NEXT\_BLKP(bp)));
127. PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 1, 0));
128. PUT(FTRP(NEXT\_BLKP(bp)), PACK(size, 1, 0));
129. bp = PREV\_BLKP(bp);
130. /\*\*then the next block is surely nonempty and it's prev\_alloc is 0\*/
131. }
132. return bp;
133. }


137. static void \*find\_fit(**size\_t** asize)
138. {
139. // First fit search
140. void \*bp;
142. for (bp = heap\_listp; GET\_SIZE(HDRP(bp)) > 0; bp = NEXT\_BLKP(bp)) {
143. if (!GET\_ALLOC(HDRP(bp)) && (asize <= GET\_SIZE(HDRP(bp))))
144. return bp;
145. }
146. return NULL; // No fit
147. }

150. void \*mm\_malloc(**size\_t** size)
151. {
152. **size\_t** asize; /\* Adjusted block size \*/
153. **size\_t** extendsize; /\* Amount to extend heap if no fit \*/
154. **char** \*bp;
156. /\* Ignore spurious requests \*/
157. if (size == 0) return NULL;
159. /\* Adjust block size to include overhead and alignment reqs. \*/
160. if (size <= DSIZE) asize = DSIZE;
161. else asize = DSIZE \* ((size + (WSIZE) + (DSIZE-1)) / DSIZE);
163. /\* Search the free list for a fit \*/
164. if ((bp = find\_fit(asize)) != NULL) {
165. place(bp, asize);
166. return bp;
167. }
169. /\* No fit found. Get more memory and place the block \*/
170. extendsize = MAX(asize,CHUNKSIZE);
171. if ((bp = extend\_heap(extendsize/WSIZE)) == NULL) return NULL;
172. place(bp, asize);
173. return bp;
174. }
176. static void place(void \*bp, **size\_t** asize)
177. {
178. **size\_t** csize = GET\_SIZE(HDRP(bp));
179. **size\_t** nsize = GET\_SIZE(HDRP(NEXT\_BLKP(bp)));
180. **int** prev\_alloc = GET\_PREVALLOC(HDRP(bp));
181. **int** next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));
182. if ((csize - asize) >= (2\*DSIZE)) {
183. PUT(HDRP(bp), PACK(asize, prev\_alloc, 1));
184. PUT(FTRP(bp), PACK(asize, prev\_alloc, 1));
185. bp = NEXT\_BLKP(bp);
186. PUT(HDRP(bp), PACK(csize-asize, 1, 0));
187. PUT(FTRP(bp), PACK(csize-asize, 1, 0));
188. }
189. else {
190. PUT(HDRP(bp), PACK(csize, prev\_alloc, 1));
191. PUT(HDRP(NEXT\_BLKP(bp)), PACK(nsize, 1, next\_alloc));
192. }
193. }

196. void mm\_end() /\*added by zhanyu\*/
197. {
198. mem\_end();
199. }

#include <stdio.h>

/\* Basic constants and macros \*/

#define WSIZE 4 /\* Word and header/footer size (bytes) \*/

#define DSIZE 8 /\* Double word size (bytes) \*/

#define CHUNKSIZE (1<<12) /\* Extend heap by this amount (bytes) \*/

#define MAX(x, y) ((x) > (y)? (x) : (y))

/\* Pack a size and allocated bit into a word \*/

//#define PACK(size, alloc) ((size) | (alloc))

#define PACK(size, prev\_alloc, alloc) ((size)|(prev\_alloc<<1)|(alloc))

/\* Read and write a word at address p \*/

#define GET(p) (\*(unsigned int \*)(p))

#define PUT(p, val) (\*(unsigned int \*)(p) = (val))

/\* Read the size and allocated fields from address p \*/

#define GET\_SIZE(p) (GET(p) & ~0x7)

#define GET\_ALLOC(p) (GET(p) & 0x1)

#define GET\_PREVALLOC(p) ((GET(p) & 0x2)>>1)

/\* Given block ptr bp, compute address of its header and footer \*/

#define HDRP(bp) ((char \*)(bp) - WSIZE)

#define FTRP(bp) ((char \*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE)

/\* Given block ptr bp, compute address of next and previous blocks \*/

#define NEXT\_BLKP(bp) ((char \*)(bp) + GET\_SIZE(((char \*)(bp) - WSIZE)))

#define PREV\_BLKP(bp) ((char \*)(bp) - GET\_SIZE(((char \*)(bp) - DSIZE)))

typedef unsigned size\_t;

int mm\_init(void);

static void \*extend\_heap(size\_t words);

void \*mm\_malloc(size\_t size);

void mm\_free(void \*bp);

static void \*coalesce(void \*bp);

static void \*find\_fit(size\_t asize);

static void place(void \*bp, size\_t asize);

void mm\_end(); /\*added by zhanyu\*/

extern void \*mem\_sbrk(int incr); //memlib.c

extern void mem\_init(void); //memlib.c

extern void mem\_end(void); //memlib.c

void \*heap\_listp;

int mm\_init(void)

{

mem\_init(); /\* added by zhanyu \*/

/\* Create the initial empty heap \*/

if ((heap\_listp = mem\_sbrk(4\*WSIZE)) == (void \*)-1) return -1;

PUT(heap\_listp, 0); /\* Alignment padding \*/

PUT(heap\_listp + (1\*WSIZE), PACK(DSIZE, 1, 1)); /\* Prologue header \*/

PUT(heap\_listp + (2\*WSIZE), PACK(DSIZE, 1, 1)); /\* Prologue footer \*/

PUT(heap\_listp + (3\*WSIZE), PACK(0, 1, 1)); /\* Epilogue header \*/

heap\_listp += (2\*WSIZE);

/\* Extend the empty heap with a free block of CHUNKSIZE bytes \*/

if (extend\_heap(CHUNKSIZE/WSIZE) == NULL) return -1;

return 0;

}

static void \*extend\_heap(size\_t words)

{

char \*bp;

size\_t size;

int prev\_alloc;

/\* Allocate an even number of words to maintain alignment \*/

size = (words % 2) ? (words+1) \* WSIZE : words \* WSIZE;

if ((long)(bp = mem\_sbrk(size)) == -1) return NULL;

prev\_alloc = GET\_PREVALLOC(HDRP(bp));

/\* Initialize free block header/footer and the epilogue header \*/

PUT(HDRP(bp), PACK(size, prev\_alloc, 0)); /\* Free block header \*/

PUT(FTRP(bp), PACK(size, prev\_alloc, 0)); /\* Free block footer \*/

PUT(HDRP(NEXT\_BLKP(bp)), PACK(0, 0, 1)); /\* New epilogue header, the prev\_alloc is 0\*/

/\* Coalesce if the previous block was free \*/

return coalesce(bp);

}

void mm\_free(void \*bp)

{

size\_t size = GET\_SIZE(HDRP(bp));

size\_t nsize = GET\_SIZE(HDRP(NEXT\_BLKP(bp)));

int prev\_alloc = GET\_PREVALLOC(HDRP(bp));

int next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));

PUT(HDRP(bp), PACK(size, prev\_alloc, 0));

PUT(FTRP(bp), PACK(size, prev\_alloc, 0));

PUT(HDRP(NEXT\_BLKP(bp)), PACK(nsize, 0, next\_alloc));

if(!next\_alloc) PUT(FTRP(NEXT\_BLKP(bp)), PACK(nsize, 0, next\_alloc));

coalesce(bp);

}

static void \*coalesce(void \*bp)

{

size\_t prev\_alloc = GET\_PREVALLOC(HDRP(bp));

size\_t next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));

size\_t size = GET\_SIZE(HDRP(bp));

if (prev\_alloc && next\_alloc) { /\* Case 1 \*/

return bp;

}

else if (prev\_alloc && !next\_alloc) { /\* Case 2 \*/

size += GET\_SIZE(HDRP(NEXT\_BLKP(bp)));

PUT(HDRP(bp), PACK(size, 1, 0));

PUT(FTRP(bp), PACK(size, 1, 0));

/\*\*then the next block is surely nonempty and it's prev\_alloc is 0\*/

}

else if (!prev\_alloc && next\_alloc) { /\* Case 3 \*/

/\*the prev prev block is surely nonempty\*/

size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));

PUT(FTRP(bp), PACK(size, 1, 0));

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 1, 0));

bp = PREV\_BLKP(bp);

}

else { /\* Case 4 \*/

/\*the prev prev block is surely nonempty\*/

size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) +

GET\_SIZE(FTRP(NEXT\_BLKP(bp)));

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 1, 0));

PUT(FTRP(NEXT\_BLKP(bp)), PACK(size, 1, 0));

bp = PREV\_BLKP(bp);

/\*\*then the next block is surely nonempty and it's prev\_alloc is 0\*/

}

return bp;

}

static void \*find\_fit(size\_t asize)

{

// First fit search

void \*bp;

for (bp = heap\_listp; GET\_SIZE(HDRP(bp)) > 0; bp = NEXT\_BLKP(bp)) {

if (!GET\_ALLOC(HDRP(bp)) && (asize <= GET\_SIZE(HDRP(bp))))

return bp;

}

return NULL; // No fit

}

void \*mm\_malloc(size\_t size)

{

size\_t asize; /\* Adjusted block size \*/

size\_t extendsize; /\* Amount to extend heap if no fit \*/

char \*bp;

/\* Ignore spurious requests \*/

if (size == 0) return NULL;

/\* Adjust block size to include overhead and alignment reqs. \*/

if (size <= DSIZE) asize = DSIZE;

else asize = DSIZE \* ((size + (WSIZE) + (DSIZE-1)) / DSIZE);

/\* Search the free list for a fit \*/

if ((bp = find\_fit(asize)) != NULL) {

place(bp, asize);

return bp;

}

/\* No fit found. Get more memory and place the block \*/

extendsize = MAX(asize,CHUNKSIZE);

if ((bp = extend\_heap(extendsize/WSIZE)) == NULL) return NULL;

place(bp, asize);

return bp;

}

static void place(void \*bp, size\_t asize)

{

size\_t csize = GET\_SIZE(HDRP(bp));

size\_t nsize = GET\_SIZE(HDRP(NEXT\_BLKP(bp)));

int prev\_alloc = GET\_PREVALLOC(HDRP(bp));

int next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));

if ((csize - asize) >= (2\*DSIZE)) {

PUT(HDRP(bp), PACK(asize, prev\_alloc, 1));

PUT(FTRP(bp), PACK(asize, prev\_alloc, 1));

bp = NEXT\_BLKP(bp);

PUT(HDRP(bp), PACK(csize-asize, 1, 0));

PUT(FTRP(bp), PACK(csize-asize, 1, 0));

}

else {

PUT(HDRP(bp), PACK(csize, prev\_alloc, 1));

PUT(HDRP(NEXT\_BLKP(bp)), PACK(nsize, 1, next\_alloc));

}

}

void mm\_end() /\*added by zhanyu\*/

{

mem\_end();

}

9.19

1) a; 对于伙伴系统，如果要申请大小为33的空间，那么需要分配64个空间。如果申请大小为65的空间，那么块大小就需要128，所以最多可能有约50%的空间被浪费。b中，最佳适配要搜索所有空间，所以肯定比首次适配要慢一些。c，边界标记主要功能是释放一个块时，能立即和前后空闲块合并。如果空闲块不按顺序排列的话，其实也能够和前一个或者后一个空闲块进行合并，但如果要和前后一起合并，可能会有些困难，那需要搜索前后块在空闲链表中的位置，并且删除一个再进行合并。可以参考P576，LIFO方法。d，其实任何分配器都可能有外部碎片，只要剩余的空闲块大小和足够但是单个都不够，就会产生外部碎片。

2) d; 块大小递增，那么最佳适配法找到的块和首次适配找到的块是同一个，因为最佳适配总是想找一个刚好大于请求块大小的空闲块。a，块大小递减，首次适配很容易找到，所以分配性能会很高。b，最佳适配方法无论怎样，都要搜索所有的链表（除非维护成块大小递增的链表）。c，是匹配的最小的。

3) c; 保守的意思就是所有可能被引用的堆都会被标记，int像指针，所以可能认为它表示的地址是正在被引用的（实际上它只是个int）。

9.20

首先，没有什么头绪~除了写好分配器，还需要考虑存储器映射吧。

真的写起来不是两三个小时就能搞定的，还可能要测试各种bug，最后性能测试也挺麻烦的。

于是乎……写不出来了。

[**深入理解计算机系统(第二版) 家庭作业 第十章**](http://blog.csdn.net/zhanyu1990/article/details/18602557)

10.6

输出 fd2 = 4

已经有0 1 2被打开，fd1是3，fd2是4，关闭fd2之后再打开，还是4。

10.7

**int** **main**(**int** argc, **char** \*\*argv)  
{  
    **int** n;  
    rio\_t rio;  
    **char** buf[MAXBUF];  
  
    **Rio\_readinitb**(&rio, STDIN\_FILENO);  
    **while**((n = **Rio\_readnb**(&rio, buf, MAXBUF)) != 0)  
        **Rio\_writen**(STDOUT\_FILENO, buf, n);  
}

10.8

只需要将stat那句话改为： fstat(atoi(argv[1]), &stat);

当然，如果需要加其他处理的话（比如判断参数对错，fd是否存在等等），还需要添加一些语句。

10.9

这里应该是表明，输入重定向到了foo.txt，然而3这个描述符是不存在的。

说明foo.txt并没有单独的描述符3。

所以Shell执行的代码应该是这样的：

**if** (**Fork**() == 0) {/\* Child \*/  
    **int** fd = **open**("foo.txt", O\_RDONLY, 0);  
    **dup2**(fd, 1);  
    **close**(fd);  
    **Execve**("fstatcheck", argv, envp);  
}

10.10

这里使用一个重定向的技术即可。如果参数个数为2，那么就将标准输入重定向到文件。

程序并没有检测各种错误。

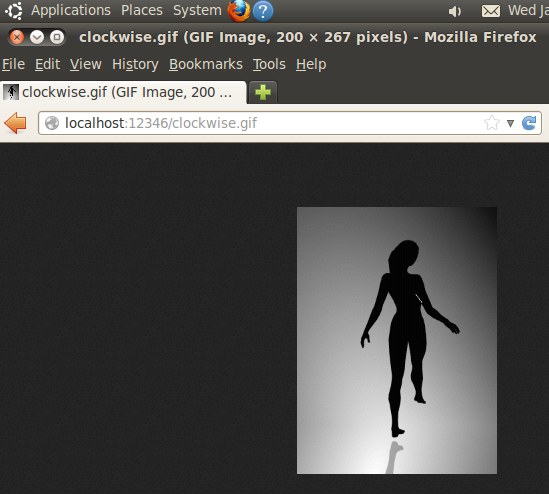
**int** **main**(**int** argc, **char** \*\*argv)  
{  
    **int** n;  
    rio\_t rio;  
    **char** buf[MAXLINE];  
      
    **if**(argc == 2){  
        **int** fd = **open**(argv[2], O\_RDONLY, 0);  
        **dup2**(fd, STDIN\_FILENO);  
        **close**(fd);  
    }  
    **Rio\_readinitb**(&rio, STDIN\_FILENO);  
    **while**((n = **Rio\_readlineb**(&rio, buf, MAXLINE)) != 0)  
        **Rio\_writen**(STDOUT\_FILENO, buf, n);  
}

[**深入理解计算机系统(第二版) 家庭作业 第十一章**](http://blog.csdn.net/zhanyu1990/article/details/18739919)

11.6

A. 因为**read\_requesthdrs**中已经打印出了请求报头，所以只要打印请求行即可。在doit函数中第一个sscanf语句之后添加下面的语句即可：**printf**("%s %s %s\n", method, uri, version);

B. 用火狐浏览器输出结果：



另外，如果要存成文件的话，可能需要另存为？

C. A的结果可以表明，浏览器使用HTTP/1.1

D. 请求行和报头如下：

GET /clockwise.gif HTTP/1.1

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:20.0) Gecko/20100101 Firefox/20.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

Connection: keep-alive

User-Agent: 系统以及浏览器情况

Accept：可以接受的媒体；

Accept-Encoding:可以接受的编码方案；

Accept-Language:能够接受的语言；

其实上述内容是百度得到的大笑

11.7

在get\_filetype函数里面添加：

**else** **if**(**strstr**(filename, ".mpg") || **strstr**(filename, ".mp4"))  
     **strcpy**(filetype, "video/mpg");

11.8

并没有想到很好的方法。

因为结束子进程之前，我们不可以关闭已连接描述符。这就导致我们还是得让serve\_dynamic或者是doit函数或者main函数中要等待子进程结束。迭代服务器的设计让这个问题变得比较无聊。

在main函数之前加入代码：

int chdEnded ;

#include <signal.h>  
void **child\_signal**(**int** sig)  
{  
    pid\_t pid;

**while**((pid = **waitpid**(-1, NULL, WNOHANG)) > 0)  
    ;

    chdEnded = 1;   
}

在main函数中添加语句 signal(SIGCHILD, child\_handle);

每次accept之前，让chdEnded = 0;

并且在doit()中的serve\_dynamic之后添加:

while(!chdEnded) pause();//or do sth

删掉serve\_dynamic里的wait(NULL);

11.9

serve\_static中的存储器映射语句改为：

srcfd = **open**(filename, O\_RDONLY, 0);  
srcp = (**char**\*)**malloc**(**sizeof**(**char**)\*filesize);  
**rio\_readn**(srcfd, srcp, filesize);  
**close**(srcfd);  
**rio\_writen**(fd, srcp, filesize);  
**free**(srcp);

11.10

HTML文件：

<html>

<body>

<form name="input" action="cgi-bin/adder" method="get">

Num1: <input type="text" name="num1"/>  <br/>

Num2: <input type="text" name="num2"/>  <br/>

<input type="submit" value="Submit"/>

</form>

</body>

</html>

因为提交的表单里面有参数名字（num1=x&num2=y），所以要修改相应的adder.c：

**int** **parseNum**(**char**\*s)  
{  
    **int** i = **strlen**(s) - 1;  
    **while**(i>0 && s[i-1]>='0'&&s[i-1]<='9' )   
        i--;  
    return **atoi**(s+i);  
}  
  
**int** **main**(void) {  
    **char** \*buf, \*p;  
    **char** arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];  
    **int** n1=0, n2=0;  
  
    /\* Extract the two arguments \*/  
    **if** ((buf = **getenv**("QUERY\_STRING")) != NULL) {  
        p = **strchr**(buf, '&');  
        \*p = 0;  
        **strcpy**(arg1, buf);  
        **strcpy**(arg2, p+1);  
        n1 = **parseNum**(arg1);  
        n2 = **parseNum**(arg2);  
    }  
  
    /\* Make the response body \*/  
    **sprintf**(content, "Welcome to add.com: ");  
    **sprintf**(content, "%sTHE Internet addition portal.\r\n<p>", content);  
    **sprintf**(content, "%sThe answer is: %d + %d = %d\r\n<p>",  
            content, n1, n2, n1 + n2);  
    **sprintf**(content, "%sThanks for visiting!\r\n", content);  
  
    /\* Generate the HTTP response \*/  
    **printf**("Content-length: %d\r\n", (**int**)**strlen**(content));  
    **printf**("Content-type: text/html\r\n\r\n");  
    **printf**("%s", content);  
    **fflush**(stdout);  
    **exit**(0);  
}

更好的做法是在本页显示，目前暂时不会。

11.11

想到的办法就是定义一个变量rmtd，表示请求的方法。

在client\_error，serve\_static和serve\_dynamic中添加一个参数mtd（改的地方也比较多），表示方法。如果mtd为HEAD，就只打印头部。

结果如下：

bnuzhanyu@ubuntu:~/CSAPP11/cgi-bin$ telnet localhost 12345

Trying ::1...

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

HEAD / HTTP/1.1

HTTP/1.0 200 OK

Server: Tiny Web Server

Content-length: 2722

Content-type: text/html

Connection closed by foreign host.

bnuzhanyu@ubuntu:~/CSAPP11/cgi-bin$ telnet localhost 12345

Trying ::1...

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

HEAD /clockwise.gif HTTP/1.1

HTTP/1.0 200 OK

Server: Tiny Web Server

Content-length: 126150

Content-type: image/gif

Connection closed by foreign host.

11.12

还是使用rmtd表示方法。

这里需要改的还有读取报头。

用POST方法，表单的参数是在报头之后。

报头中有一项是Content-Length，读取出来，在报头读完之后，接着读取Content-Length个字节（注意最后添0），这些字节就是form用post方法提交的数据。

主要修改的就是doit方法和read\_request方法。

下面的程序只能针对参数为文本的情况，且参数总长度最大不超过MAXLINE。

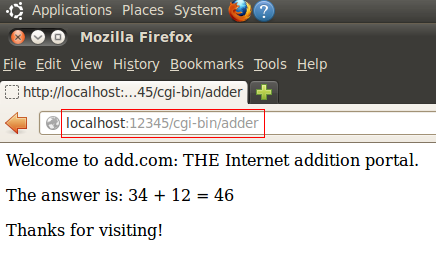
#**define** M\_GET 0  
#**define** M\_POST 1  
#**define** M\_HEAD 2  
#**define** M\_NONE (-1)

void **doit**(**int** fd)  
{  
    **int** is\_static;  
    **int** rmtd = 0;  
    **struct** stat sbuf;  
    **char** buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];  
    **char** filename[MAXLINE], cgiargs[MAXLINE];  
    rio\_t rio;  
  
    /\*for post\*/  
    **int** contentLen;  
    **char** post\_content[MAXLINE];  
  
    /\* Read request line and headers \*/  
    **rio\_readinitb**(&rio, fd);  
    **rio\_readlineb**(&rio, buf, MAXLINE);  
    **sscanf**(buf, "%s %s %s", method, uri, version);  
    **printf**("%s %s %s\n", method, uri, version);  
  
    **if**(**strcmp**(method, "GET") == 0) rmtd = M\_GET;  
    **else** **if**(**strcmp**(method, "POST") == 0) rmtd = M\_POST;  
    **else** **if**(**strcmp**(method, "HEAD") == 0) rmtd = M\_HEAD;  
    **else** rmtd = M\_NONE;  
  
    **if** (rmtd == M\_NONE) {  
        **clienterror**(fd, method, "501", "Not Implemented",  
            "Tiny does not implement this method", rmtd);  
        return;  
    }  
  
    contentLen = **read\_requesthdrs**(&rio, post\_content, rmtd);  
  
    /\* Parse URI from GET request \*/  
    is\_static = **parse\_uri**(uri, filename, cgiargs);  
    **if** (**stat**(filename, &sbuf) < 0) {  
        **clienterror**(fd, filename, "404", "Not found",  
                "Tiny couldn't find this file", rmtd);  
        return;  
    }  
  
    **if** (is\_static) {/\* Serve static content \*/  
        **if** (!(**S\_ISREG**(sbuf.st\_mode)) || !(S\_IRUSR & sbuf.st\_mode)) {  
            **clienterror**(fd, filename, "403", "Forbidden",  
                    "Tiny couldn't read the file", rmtd);  
            return;  
        }  
        **serve\_static**(fd, filename, sbuf.st\_size, rmtd);  
    }  
  
    **else** {/\* Serve dynamic content \*/  
        **if** (!(**S\_ISREG**(sbuf.st\_mode)) || !(S\_IXUSR & sbuf.st\_mode)) {  
            **clienterror**(fd, filename, "403", "Forbidden",  
                    "Tiny couldn't run the CGI program", rmtd);  
            return;  
        }  
        **if**(rmtd == M\_POST) **strcpy**(cgiargs, post\_content);  
        **serve\_dynamic**(fd, filename, cgiargs, rmtd);  
    }  
}

**int** **read\_requesthdrs**(rio\_t \*rp, **char**\* content, **int** rmtd)  
{  
    **char** buf[MAXLINE];  
    **int** contentLength = 0;  
    **char** \*begin;  
    **rio\_readlineb**(rp, buf, MAXLINE);  
  
    **while**(**strcmp**(buf, "\r\n")) {  
        **rio\_readlineb**(rp, buf, MAXLINE);  
        **printf**("%s", buf);  
        **if**(rmtd == M\_POST && **strstr**(buf, "Content-Length: ")==buf)  
            contentLength = **atoi**(buf+**strlen**("Content-Length: "));  
    }  
    **if**(rmtd == M\_POST){  
        contentLength = **rio\_readnb**(rp, content, contentLength);  
        content[contentLength] = 0;  
        **printf**("POST\_CONTENT: %s\n", content);  
    }  
    return contentLength;  
}

strstr那句也可以用strncmp代替（效率应该高一些）。

表单改成Post之后，运行效果：



11.13

为了测试EPIPE错误，我在read\_requesthdrs里面添加了sleep(5)。

于是，在浏览器里请求之后，立即断开。进程出现错误：

GET /add.html HTTP/1.1

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:20.0) Gecko/20100101 Firefox/20.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

Connection: keep-alive

Segmentation fault

bnuzhanyu@ubuntu:~/CSAPP11$

为了解决这个问题，我用了setjmp和longjmp。

当进程捕捉到SIGPIPE时，进入一个信号处理函数：

jmp\_buf buf;  
void **epipe\_signal**(**int** sig)  
{  
    **longjmp**(buf, 1);  
}

而在main函数中，doit部分需要这样改：

rc = **setjmp**(buf);  
**if**(rc == 0) {  
    **doit**(connfd);  
    **close**(connfd);  
}

这样做虽然能解决这个问题，然而，如果是在子进程中longjmp（也就是adder里sleep(5)时，浏览器关闭了），那么会不会有问题呢？

为了测试，我又在adder中加入了sleep(5)。

并且setjmp后面的语句改为：

**if**(rc == 0) {  
    **printf**("rc=%d   pid=%d\n", rc, **getpid**());  
    **doit**(connfd);  
    **close**(connfd);  
}  
**else** **printf**("rc=%d   pid=%d\n", rc, **getpid**());

结果是，当我在adder的sleep期间关闭浏览器，没有输出rc的消息。

后来想起来，输出已经被重定向了，而且是子进程和父进程的输出都被重定向了（没懂为什么都重定向了）。即，当浏览器再次请求时，rc=0也不会输出。

后来在setjmp后添加了dup2(2,1)，即将标准输出重定向到标准错误，就可以看见输出了。

奇怪的是，pid都是main的pid。这是不是说明，在子进程中调用longjmp仍然可以回到父进程。那么子进程去哪儿了呢？这个问题还是需要研究一下。

今天测试了一下，子进程的longjmp还是在子进程中。

所以，现在不知道之前那个程序为什么一直输出main的pid，这太奇怪了。

这也就证明我上面的写法是不正确的。

但是，想一想，觉得如果是在子进程中遇到关闭描述符的问题，那么可以直接退出这个进程啊，因为没必要再让这个进程存活下去了。所以，只要简单地判断当前进程是不是子进程，如果是，则exit，如果不是，则longjmp。

先用一个变量mainpid在主函数开始时赋值为getpid();

if(getpid() == mainpid) longjmp(buf,1); else exit(0);