作业参考答案

2.60:

Byte extraction and insertion code is useful in many contexts. Being able to write this sort of code is an important skill to foster.

code/data/rbyte-ans.c

```
unsigned replace_byte (unsigned x, int i, unsigned char b) {
    int itimes8 = i << 3;
    unsigned mask = 0xFF << itimes8;

    return (x & ~mask) | (b << itimes8);
}
```

code/data/rbyte-ans.c

2.64

This problem is very simple, but it reinforces the idea of using different bit patterns as masks.

code/data/bits.c

```
/* Return 1 when any odd bit of x equals 1; 0 otherwise.  Assume w=32 */
int any_odd_one(unsigned x) {
    /* Use mask to select odd bits */
    return (x&0xAAAAAAAA) != 0;
}
```

code/data/bits.c

2.68

Here is the code:

code/data/bits.c

```
/*
 * Mask with least signficant n bits set to 1
 * Examples: n == 6 --> 0x3F, n == 17 --> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n) {
    /*
     * 2^n-1 has bit pattern 0...01..1 (n 1's)
     * But, we must avoid a shift by 32

     */
    return (2<<(n-1)) - 1;
}
```

code/data/bits.c

The code makes use of the trick that `(1<<n)-1` creates a mask of `n` ones. The only challenge is to avoid shifting by w when `n` $= w$. Instead of writing `1<<n`, we write `2<<(n-1)`. This code will not work for `n` $= 0$, but that's not a very useful case, anyhow.

2.72

This code illustrates the hidden dangers of data type `size_t`, which is defined to be `unsigned` on most machines.

A. Since this one data value has type `unsigned`, the entire expression is evaluated according to the rules of unsigned arithmetic. As a result, the conditional expression will always succeed, since every value is greater or equal to 0.

B. The code can be corrected by rewriting the conditional test:

```
/* Must do signed comparison in event maxbytes < 0 */
if (maxbytes >= sizeof(val))
```

2.76

The solution to this problem can easily be adapted from the solution to Problem 2.30. The main point is to make sure students see that connection.

code/data/calloc.c

```
1  void *calloc(size_t nmemb, size_t size) {
2      size_t asize =  nmemb * size;
3      /* Check for overflow */
4      if (nmemb == 0 || size == 0 || asize / nmemb != size)
5          /* Error */
6          return NULL;
7      void *result = malloc(asize);
8      if (result != NULL) {
9          memset(result, 0, asize);
10         return result;
11     }
12     return NULL;
13 }
```

code/data/calloc.c

2.80

The requirement that the function must not overflow makes this problem more challenging than Problem 2.79. The idea in our solution is to compute the lower 2 bits, including the bias separately, to derive a value `incr` that will be either 0, 1, or 2, that can be added to the remaining bits of `3*x`.

code/data/bits.c

```
1  /* Compute 3/4*x with no overflow */
2  int threefourths(int x) {
3      int xl2 = x & 0x3;
4      int xl1 = (x&1) << 1;
5      int x_mask = x >> ((sizeof(int)<<3)-1);
6      int bias = x_mask & 3;
7      int incr = (xl2+xl1+bias) >> 2;
8      int s2 = x >> 2;
9      int s1 = x >> 1;
10     return s1 + s2 + incr;
11 }
```

code/data/bits.c

2.84

This problem helps students appreciate the property of IEEE floating point that the relative magnitude of two numbers can be determined by viewing the combination of exponent and fraction as an unsigned integer. Only the signs and the handling of ± 0 requires special consideration.

code/data/floatcomp-ans.c

```
int float_le(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    return
        (ux<<1 == 0 && uy<<1 == 0) ||  /* Both are zero */
        (sx && !sy) ||                 /* x < 0, y >=  0 */
        (!sx && !sy && ux <= uy) ||    /* x >= 0, y >= 0 */
        (sx && sy && ux >= uy);        /* x <  0, y <  0 */
}
```

code/data/floatcomp-ans.c

2.86

This exercise is of practical value, since Intel-compatible processors perform all of their arithmetic in extended precision. It is interesting to see how adding a few more bits to the exponent greatly increases the range of values that can be represented.

Description	Extended precision	
	Value	Decimal
Smallest pos. denorm.	$2^{-63} \times 2^{-16382}$	3.64×10^{-4951}
Smallest pos. norm.	2^{-16382}	3.36×10^{-4932}
Largest norm.	$(2 - \epsilon) \times 2^{16383}$	1.19×10^{4932}

3.60

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces students to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops.

A. We can see that `result` must be in register `%rax`, since this value gets returned as the final value. Parameter `x` is passed in `%rdi`. Parameter `n` is passed in `%esi` and then copied into `%ecx`. Register `%edx` is initialized to 1. We can infer that `mask` must be `%rdx`.

B. They are initialized to 0 and 1, respectively.

C. The condition for continuing the loop is that `mask` is nonzero.

D. The `salq` instruction updates `mask` to be `mask << n`.

E. Variable `result` is updated to be `result | (x&mask)`.

F. Here is the original code:

```
long loop(long x, int n)
{
    long result = 0;
    long mask;
    for (mask = 0x1; mask != 0; mask = mask << n) {
        result |= (x & mask);
    }
    return result;
}
```

3.64

This problem demonstrates that the same principles of nested array access extend beyond two levels.

A. Array element `A[i][j][k]` is located at address $x_A + 8(T(S \cdot i + j) + k)$.

B. Consider the following annotated version of the assembly code:

```
   long store_ele(long i, long j, long k, long *dest)
   i in %rdi, j in %rsi, k in %rdx, dest in %rcx
 1 store_ele:
 2   leaq    (%rsi,%rsi,2), %rax     3 j
 3   leaq    (%rsi,%rax,4), %rax     13 j
 4   movq    %rdi, %rsi              Copy i
 5   salq    $6, %rsi                64 i
 6   addq    %rsi, %rdi              65 i
 7   addq    %rax, %rdi              13 j + 65 i = 13(5 i + j)
 8   addq    %rdi, %rdx              13(5 i + j) + k
 9   movq    A(,%rdx,8), %rax        M[x_A + 8 (13(5 i + j) + k)]
10   movq    %rax, (%rcx)
11   movl    $3640, %eax
12   ret
```

We can see that memory reference on line 9 indicate that $T = 13$ and $S = 5$. We can see on line 11 that the total array size is 3640 bytes. From this, we get $R = 3640/(8 \cdot 13 \cdot 5) = 7$.

3.68

This problem is like a puzzle, where a number of clues must be assembled to get the answer. It tests students' understanding of structure layout, including the need to insert padding to satisfy alignment. The right way to solve the problem is to write out formulas for the offsets of the different fields in terms of A and B and then determine the possible solutions.

We can see from the assembly code that fields `t` and `u` of structure `str2` are at offsets 8 and 12, respectively. We can see that field `y` of structure `str1` is at offset 184. We can write the following equations for these offsets:

$$\begin{aligned} B + e_1 &= 8 \\ B + e_1 + 4 + 2A + e_2 &= 32 \\ 4A \cdot B + e_3 &= 184 \end{aligned}$$

where e_1, e_2, and e_3 represent amounts of padding we need to insert in the structures to satisfy alignment. We can also see that $e_1 \in \{0, 1, 2, 3\}$, $e_2 \in \{0, 2, 4, 6\}$, and $e_3 \in \{0, 4\}$.

From the first equation, we can see that $B \in \{5, 6, 7, 8\}$. We can substitute the first equation into the second to get

$$2A + e_2 = 20$$

which implies $A \in \{7, 9, 8, 10\}$. Finally, the third equation implies $A \cdot B \in \{45, 46\}$.

The only combination satisfying all three constraints is $A = 9$ and $B = 5$.

5.14

This problem gives practice applying loop unrolling.

```
1  /* Inner Product.  6 X 1 unrolling */
2  void inner_u6x1(vec_ptr u, vec_ptr v, data_t *dest)
3  {
4      long i;
5      long length = vec_length(u);
6      long limit = length-5;
7      data_t *udata = get_vec_start(u);
8      data_t *vdata = get_vec_start(v);
9      data_t sum = (data_t) 0;
10     /* Do 6 elements at a time */
11     for (i = 0; i < limit; i+=6) {
12         sum = sum + udata[i] * vdata[i]
13             + udata[i+1] * vdata[i+1]
14             + udata[i+2] * vdata[i+2]
15             + udata[i+3] * vdata[i+3]
16             + udata[i+4] * vdata[i+4]
17             + udata[i+5] * vdata[i+5];
18     }
19     /* Finish off any remaining elements */
20     for (; i < length; i++) {
21         sum = sum + udata[i] * vdata[i];
22     }
23     *dest = sum;
24 }
```

A. We must perform two loads per element to read values for `udata` and `vdata`. Since the machine has two functional units that can do this operation, the CPE cannot be less than 1.0.

B. The performance for floating point is still limited by the 3 cycle latency of the floating-point adder.

6.24

This is a good check of the student's understanding of the factors that affect disk performance. It's a nice model problem that can be easily changed from term to term. First we need to determine a few basic properties of the file and the disk. The file consists of 4,000 512-byte logical blocks. For the disk, $T_{avg\,seek}$ = 4 ms, $T_{max\,rotation}$ = 4 ms, and $T_{avg\,rotation}$ = 2 ms.

A. *Best case:* In the optimal case, the blocks are mapped to contiguous sectors, on the same cylinder, that can be read one after the other without moving the head. Once the head is positioned over the first sector it takes 4 full rotations (1,000 sectors per rotation) of the disk to read all 4,000 blocks. So the total time to read the file is $T_{avg\,seek} + T_{avg\,rotation} + 4 * T_{max\,rotation} = 4 + 2 + 16 = 22$ ms.

B. *Random case:* In this case, where blocks are mapped randomly to sectors, reading each of the 4,000 blocks requires $T_{avg\,seek} + T_{avg\,rotation}$ ms, so the total time to read the file is $(T_{avg\,seek} + T_{avg\,rotation}) * 4,000 = 24,000$ ms (24 seconds).

6.28

Another inverse cache indexing problem, using the same cache as the previous problem.

A. Set 2 contains no valid lines, so no addresses will hit.

B. Set 4 contains two valid lines: Line 0 and Line 1. Line 0 has a tag of `0xC7`. There are four bytes in each block, and thus four addresses will hit in Line 0. These addresses have the binary form `1 1000 1111 00xx`. Thus, the following four hex addresses will hit in Line 0 of Set 4: `0x18f0`, `0x18f1`, `0x18f2`, and `0x18f3`.

   Similarly, the following four addresses will hit in Line 1 of Set 4: `0x00b0`, `0x00b1`, `0x00b2`, `0x00b3`.

C. Set 5 contains one valid line, Line 0, with a tag of `0x71`. Addresses that hit in this line have the binary form `0 1110 0011 01xx`. Thus, the following four hex addresses will hit in Line 0 of Set 5: `0x0e34`, `0x0e35`, `0x0e36`, `0x0e37`.

D. Set 7 contains one valid line, Line 1, with a tag of `0xde`. Addresses that hit in this line have the binary form `1 1011 1101 11xx`. Thus, the following four hex addresses will hit in Line 1 of Set 7: `0x1bdc`, `0x1bdd`, `0x1bde`, `0x1bdf`.

6.32

Address `0x16E8`

A. Address format (one bit per box):

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

B. Memory reference:

Parameter	Value
Block Offset (CO)	0x0
Index (CI)	0x2
Cache Tag (CT)	0xB7
Cache Hit? (Y/N)	N
Cache Byte returned	–

6.40

Both loops access the array in row-major order. The first loop performs 256 writes. Since each cache line holds two structures, half of these references hit and half miss. The second loop performs a total of 768 writes. For each pair of structures, there is an initial cold miss, followed by 5 hits. So this loop experiences a total of 128 misses. Combined, there are $256 + 768 = 1024$ writes, and $128 + 128 = 256$ misses.

A. What is the total number of writes? 1024 writes.

B. What is the total number of writes that miss in the cache? 256 writes.

C. What is the miss rate? $256/1024 = 25\%$.

7.10

These are more drills, in the spirit of Problem 7.3, that help the students understand how linkers use static libraries when they resolve symbol references.

A. `gcc p.o libx.a`

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

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

7.12

The solution outline is identical to Problem 7.5:

A. $0x4004f8 - 4 - 0x4004ea = 0xa$

B. $0x400500 - 4 - 0x4004da = 0x22$

8.10

A. Called once, returns twice: `fork`

B. Called once, never returns: `execve` and `longjmp`.

C. Called once, returns one or more times: `setjmp`.

8.12

This program has a similar process graph as the program in Figure 8.17. There are four processes, each of which prints one "hello" line in `doit` and one "hello" line in `main` after it returns from `doit`. Thus, the program prints a total of eight "hello" lines.

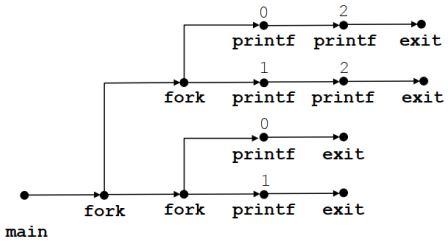
8.17

Inspection of the process graph in the solution to Problem 8.4 shows that there are only three possible outcomes (each column is an outcome):

```
Hello   Hello   Hello
1       1       0
Bye     0       1
0       Bye     Bye
2       2       2
Bye     Bye     Bye
```

8.18

This problem really tests the students' understanding of concurrent process execution. The key is to draw the process graph:



A. 112002 (possible)

B. 211020 (not possible)

C. 102120 (possible)

D. 122001 (not possible)

E. 100212 (possible)

8.20

This is an easy problem for students who understand the `execve` function and the structure of the `argv` and `envp` arrays. Notice that a correct solution must pass a pointer to the `envp` array (the global `environ` pointer on our system) to correctly mimic the behavior of `/bin/ls`.

code/ecf/myls-ans.c

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv) {
4   Execve("/bin/ls", argv, environ);
5   exit(0);
6 }
```

code/ecf/myls-ans.c

9.11

The following series of address translation problems give the students more practice with translation process. These kinds of problems make excellent exam questions because they require deep understanding, and they can be endlessly recycled in slightly different forms.

```
A. 00 0010 0111 1100

B.      VPN:            0x9
        TLBI:           0x1
        TLBT:           0x2
        TLB hit?        N
        page fault?     N
        PPN:            0x17

C.      0101 1111 1100

D.      CO:             0x0
        CI:             0xf
        CT:             0x17
        cache hit?      N
        cache byte?     -
```