

NAME:

Login name:

**Computer Science 217
Final Exam
May 15, 2009
1:30pm-4:30pm**

This test has eight (8) questions and thirteen (13) pages. Put your name (or login-id) on *every page*, and write out and sign the Honor Code pledge before turning in the test.

``I pledge my honor that I have not violated the Honor Code during this examination."`

Question	Score
1 (8 pts)	
2 (10 pts)	
3 (12 pts)	
4 (20 pts)	
5 (10 pts)	
6 (10 pts)	
7 (10 pts)	
8 (20 pts)	
Total	

QUESTION 1 Building a Program (8 POINTS, 1 point each)

This question concerns the steps involved in creating a binary executable **a.out** for the following program:

```
#include <stdio.h>

/* This is my cool program */
int main(void) {
    int i;

    scanf("%d", &i);
    while (i--)
        printf("i=%d\n", i);
    return 0;
}
```

The four main stages of creating **a.out** are pre-processing, compiling, assembling, and linking. For each operation, circle which component performs the task, where **P** stands for pre-processor, **C** for compiler, **A** for assembler, and **L** for linker.

- P** C A L Inserts the contents of `/usr/include/stdio.h`
- P C A **L** Includes the code that implements `scanf()`
- P **C** A L Ensures that the first argument of `printf()` is of type “char *”
- P** C A L Removes the comment “`This is my cool program`”
- P **C** A L Checks that “`return 0`” returns an integer
- P **C** A L Translates ‘`i--`’ into the “`decl`” instruction
- P C **A** L Determines the argument for the “`jmp`” instruction to determine how far to jump to get the start of the while loop
- P C A **L** Determines the address to call to invoke the `scanf()` function

QUESTION 2: Memory Management (10 POINTS)

2a) Consider a computer system that has virtual memory with **32-bit** addresses and a **16 KB** page size. How many bits are used to identify the **byte offset** in a page? How many **virtual pages** can a process have? (2 points)

16 KB is 2^{14} so the byte offset has 14 bits

With 32-bit addresses, this leaves 18 bits for the page id, so there are 2^{18} pages.

2b) Which component is responsible for the following tasks, the underlying hardware (H) or the operating system (O)? Please circle either “H” or “O” for each item. (4 points)

H O Mapping virtual address into a physical address for pages already in physical memory

H O Deciding which virtual page to “swap out” of physical memory on a “miss”

H O Updating the page tables with new virtual-to-physical page mappings

H O Preventing one user process from accessing the pages of another user process

2c) For caching in a memory hierarchy, what is the motivation for a *larger* cache block size? Please check one answer. (1 point)

Temporal locality

Spatial locality

2d) For caching disk pages in main memory, what overhead is amortized by using *large pages*? Please check one answer. (1 point)

Slow disk seek time

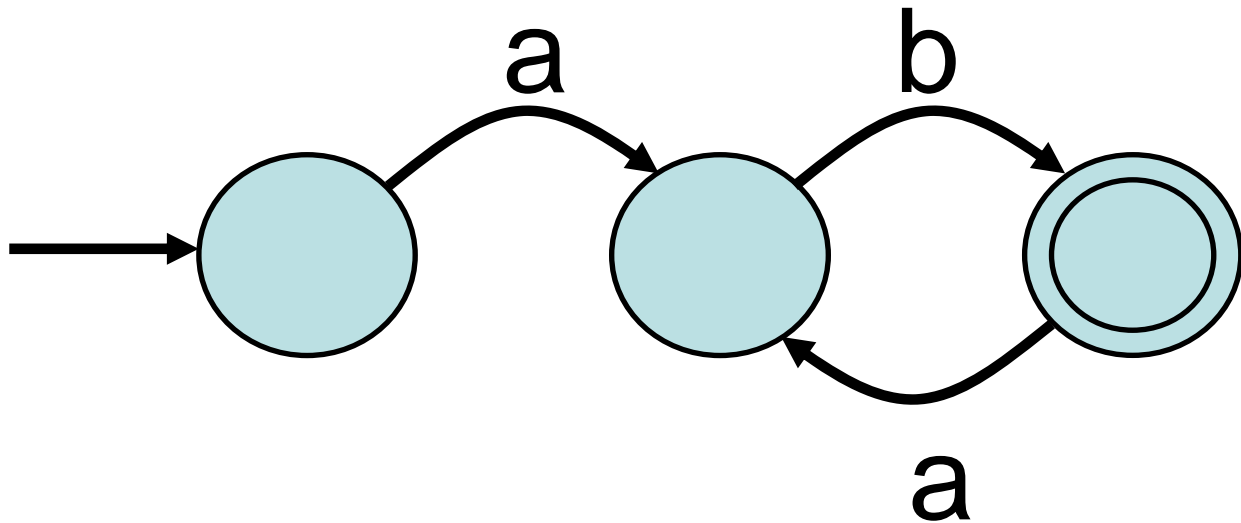
Low disk throughput

2e) Why does implementing **malloc()** and **free()** with a single free list (with free blocks of different sizes) lead to a lot of virtual-memory *page faults*? (2 points)

Finding a suitable free block (large enough, and perhaps also a “good” fit, depending on the policy) may require sequencing through many free blocks. These memory references could easily span many pages, forcing swapped-out pages to be brought in from disk.

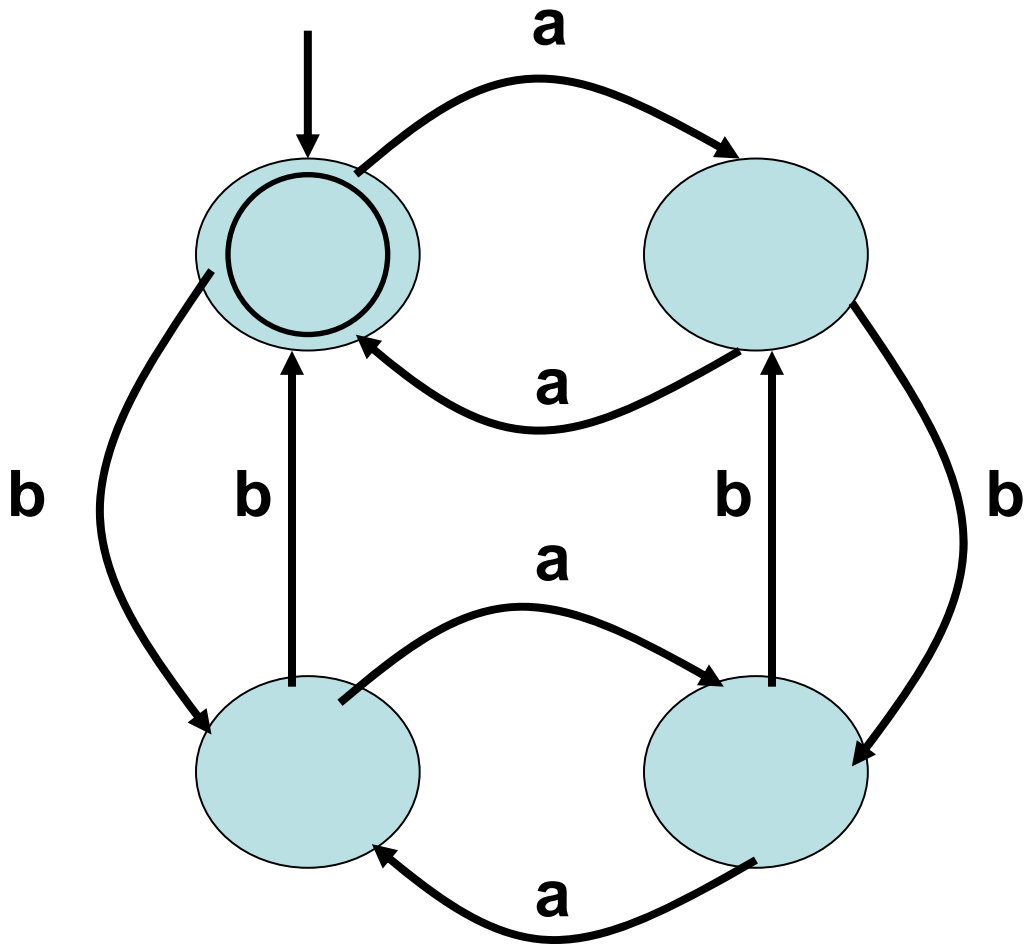
QUESTION 3: Deterministic Finite Automata (12 POINTS)

In this question, you will draw a deterministic finite automata (DFA) that accepts particular strings. Please draw your DFA diagrams with a start state (marked with an incoming arrow) and success states (circled twice). As an example, consider a DFA that reports “success” for an input that repeats the characters “ab” one or more times and reports “failure” otherwise. The inputs “ab”, “abab”, and “ababab” would lead to success, whereas “a”, “aba”, “abc”, “789”, or “cabab” would not. That DFA would be drawn as:



Draw a DFA that accepts only strings of a’s and b’s that have an even number of each character. For example, the DFA would accept the empty string, “aa”, “bb”, “aabb”, “abab”, and “ababbb”, but not “aab” or “aaaba”. Ensure that your DFA has the *minimum* possible number of states.

Answer on the next page.



There are four states, depending on whether the input stream has, thus far, included even or odd numbers of a's and b's, respectively. In the picture above, the two rightmost states correspond to having an odd number of a's, and the bottom two states correspond to having an odd number of b's. The upper left (accept) state corresponds to having an even number of both a's and b's. Any input that is not an 'a' or a 'b' implicitly leads to a permanent failure state.

A relatively common mistake was to have an extra "start" state in addition to the single accept state above. This answer, while correct, does not have the minimum number of states. Some answers also had additional redundant states.

QUESTION 4: Reduced Instructions for Silly C (20 POINTS)

The instruction set of a computer defines the basic operations it can perform, and all other operations must be performed by combining these operations together. This question explores the challenges of “making do” with an instruction set with less functionality. Rather than writing assembly-language programs, we envision a reduced version of C with limited functionality, and build functions that implement the missing operations.

4a) Suppose C does not support the multiplication operation (e.g., “4 * 5”). Write a function `multiply()` that performs multiplication of two unsigned integers, and returns the unsigned integer result. Do not use multiplication, division, bit-wise operations, or any functions from libraries. Make your code as *short* as possible. (6 points)

Integer multiplication can be implemented through repeated addition, as follows

```
unsigned multiply(unsigned a, unsigned b) {
    unsigned c = 0;

    while (a-- > 0)
        c += b;

    return c;
}
```

Another similar answer is:

```
unsigned multiply(unsigned a, unsigned b) {
    unsigned i, sum=0;

    for (i = 0; i < b; i++)
        sum += a;
    return sum;
}
```

An even shorter answer, using recursion, is

```
unsigned multiply(unsigned a, unsigned b) {
    return a ? b + multiply(a-1,b) : 0;
}
```

4b) Suppose C does not have a left shift operator (e.g., “k << 3”). Write a function `lefty()` that performs left shift on an unsigned integer, shifting an unsigned integer number of times. Do not use multiplication, division, or bit-wise operations, or any functions from libraries. Do *not* use the `multiply()` function from question **4a**. Make your code as *short* as possible. (6 points)

Each bit of shifting multiplies a number by two. Or, stated in terms of addition, each bit of shifting corresponds to adding the number to itself. So, this is a valid implementation:

```
unsigned lefty(unsigned k, unsigned shift) {
    while (shift--)
        k += k;

    return k;
}
```

An even shorter version, using recursion, is

```
unsigned lefty(unsigned k, unsigned shift) {
    return shift ? lefty(k+k, shift-1) : k;
}
```

4c) Suppose C only supports one size of unsigned integers (**unsigned**) but you want to perform arithmetic on larger integers. Define a **big_unsigned** data structure that consists of two unsigned integers corresponding to the upper and lower bits of a **big_unsigned** number. Implement a function that performs addition on two **big_unsigned** numbers and returns the **big_unsigned** result. Use only comparison operations and unsigned arithmetic. (8 points)

The main challenge is to correctly handle the carry from the low-order bits to the upper. Here is one implementation:

```
typedef struct {
    unsigned upper;
    unsigned lower;
} big_unsigned;

big_unsigned big_add(big_unsigned a, big_unsigned b) {
    big_unsigned c;

    c.upper = a.upper + b.upper;
    c.lower = a.lower + b.lower;

    if (c.lower < a.lower)
        c.upper++;

    return c;
}
```

As an aside, quite a few students checked for carry as

“if (c.lower < a.lower) || (c.lower < b.lower)”

but comparing to sum to either of the two arguments is sufficient; comparing both is not needed.

Also, several students reported an error message if computing the sum of the “upper” bits (plus carry) led to an overflow. However, this is not necessary, as unsigned arithmetic is always assumed to do modulo arithmetic. As such, it is okay to ignore the carry out of the upper bits.

Also, several students had more complex logic for checking for overflow in adding the lower bits, including answers that mistakenly assumed a particular size for unsigned ints.

Some students implemented the function assuming the function parameters and the return value were pointers. This is fine (as the question did not specify which approach to take), though it tended to lead to more complex code. Also, some students created separate unsigned variables for the upper and lower parts of the sum, and only assigned the fields in the struct at the end; while correct, this also led to more complex code.

Finally, one student came up with the following very concise answer (even though this question did not specifically ask for a concise answer) by (i) computing the sum directly in one of the two function parameters and (ii) incrementing for the carry based on a Boolean expression. In particular:

```
big_unsigned big_add(big_unsigned a, big_unsigned b) {  
    a.lower += b.lower;  
    a.upper += b.upper + (a.lower < b.lower);  
    return a;  
}
```

QUESTION 5: Process Control (10 POINTS)

5a) Consider the similarities and differences between *calling a function* and *performing a context switch*. Circle whether the following statements are true for a function call (F), a context switch (C), or both (B). (3 points)

F C **B** Values stored in registers are saved so they can be restored later

F **C** B Control of the computer transfers to the operating system

F C **B** The instruction pointer (EIP) changes to execute instructions in a new location

5b) A call to `fork()` creates a child process that inherits a copy of the parent's virtual address space. The virtual address space is quite large, so copying every byte would be quite time consuming. How is this overhead avoided? (3 points)

While `fork()` does create a separate address space for the child process, the pages in memory is not necessarily physically copied unless either the child or the parent modifies the page. This is implemented by having both the parent and child initially point to a single shared copy of each page. When either process modifies a page, a separate copy is made at that time. The “copy on write” semantics significantly reduces the overhead of cloning the parent process.

5c) Give *two* examples of why a process might leave the “running” state. (2 points)

The operating system may force a process to leave the running state when its time quantum expires, to allow another process to run. A process may also leave the running state when it is stalled waiting for I/O to complete.

5d) Suppose a user types

```
echo foobar | wc -l
```

at the UNIX prompt. What are *stdin* and *stdout* for the `echo` and `wc` processes? (2 points)

*For `echo`, *stdin* is the keyboard and *stdout* is directed to the pipe, which forms the *stdin* for `wc`. For `wc`, *stdin* comes from the pipe and *stdout* is directed to the screen.*

*Alternatively, focusing on the *contents* of the I/O rather than where they come from.... For `echo`, *stdin* does not matter – note that `foobar` is input in `argv[]` not from *stdin* – and *stdout* is “`foobar`”. For `wc`, *stdin* is “`foobar`” and *stdout* is “`1`”.*

QUESTION 6: Review (10 POINTS)

6a) If the character `'\0'` can be used to signify the end of a *string*, why can't it be used as a way to signify the end of a *file* (instead of using the integer EOF)? (2 points)

A string consists only of characters, where `'\0'` has a special 'end of string' meaning and `'\0'` cannot appear in the body of the string. A file, though, may have arbitrary binary data, including bytes with the value 0 (the same as the ASCII code of `'\0'`); hence, a separate mechanism is necessary to indicate the end of a file.

6b) Consider the following code, where `i` and `j` are unsigned integers:

```
i = 7;
printf("%d\n", i=j ? 0 : j);
```

where `j` has already been assigned a value. Give a concise description of what the code prints to standard output. (2 points)

The statement `"i=j"` assigns the value of `j` to `i`, and has the associated value of `j`. So, if `j` is non-zero, the statement is true (and `"0"` is printed), and if `j` is zero, the statement is false (and the value of `j`, which is 0 in this case, is printed). Either way, then, `"0\n"` is printed.

6c) Consider the following operations in C where `i` and `k` are unsigned integers:

```
i = (((k/4) * 4) >> 2) << 2);
```

Rewrite to compute the same result as succinctly as possible. (2 points)

*The `"(k/4) * 4"` essentially zeroes out the lower two bits of `k`. Then, the `">> 2"` and `"<< 2"` essentially do the same thing again, having no effect. This could be done much more concisely by simply by ANDing `k` with `111...1100`. That is, by doing `"i = k & ~3;"`.*

6d) Give an example of a programming error that is caught by the *preprocessor*, and another that is caught by the *compiler*. (2 points)

The preprocessor would detect errors like misspelling the name of an including `.h` file (e.g., `"#include <stdiio.h>"`) or omitting the `"/"` at the end of a comment. The compiler would detect errors like misspelling a keyword (e.g., `"retun 0"`) or calling a function with parameters that have the wrong type (assuming the compiler sees the function declaration prior to the call).*

6e) What does

```
printf("%x %x\n", 0xfad - 0xcab, 0xfad & 0xdff);
```

print to standard output? (2 points)

The "0xfad - 0xcab" has hex value 302. The "0xfad & 0xdff" applies a bitmask "d" (1101) to the "f" in "fad", resulting in "d", so the result is "dad" in hex.

So, the output of printf() is "302 dad\n".

QUESTION 7: Reading Fork Code (10 POINTS)

This question concerns the following program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void cos217(void) {
    pid_t pid;

    if (!(pid = fork())) {
        fork();
        fork();
        fork();
        printf("cos217\n");
    }
}

int main(void) {
    cos217();
    printf("cos217\n");
    return 0;
}
```

7a) How many times does this program print “cos217”? (4 points)

The original parent process prints “cos217\n” twice, once in main() and once in the “if” clause.

The first child, created by the fork() in the “if” statement, prints “cos217\n” once in main().

The second child, created by the first fork() in the body of the “if”, prints “cos217\n” twice.

The parent and this second child both invoke the second fork() in the body of the “if”, leading to the creation of two more children who each print “cos217\n” twice.

These four processes invoke the final call to fork() in the body of the “if”, leading to the creation of four more children who each print “cos217\n” twice.

*In total, then, we have $2 + 1 + 2 + 2*2 + 4*2$, for a grand total of **17**.*

7b) When running the program, sometimes parts of the output look like this:

```
cos217cos217
```

```
cos217
```

Why does this happen, and how can it be prevented? (6 points)

The operating system can swap out a process any time, e.g., when its time quantum expires. This could happen in the middle of the running of printf(). To prevent this from happening, the parent process could wait for the child to complete before performing its tasks. Note that calling fflush() may not be sufficient to prevent this problem, because the printing of characters may still be interrupted if a process is swapped out to run another process.

QUESTION 8: Code Reading, and Writing (20 POINTS)

8a) Consider the following code:

```
...
double* x;

x = (double *) malloc(sizeof(double*));
&x = 3.145;
...
```

Identify *two* bugs in the code, and rewrite the code to be correct. (2 points)

The argument to malloc() should be the size of a double, not the size of a pointer to a double. The “&x” is getting the address of x (the address of where the pointer is stored), rather than dereferencing the pointer to assign a value to the memory location where x points. The correct code is:

```
x = (double *) malloc(sizeof(double));
*x = 3.145;
```

8b) Consider the following code:

```
char a[10][20][30];
int i, j, k, sum = 0;

...
for (j=0; j<20; j++)
  for (i=0; i<10; i++)
    for (k=0; k < strlen(a[i][j]); k++)
      sum += (a[i][j][k] == 'J');
```

Give a high-level description of what this code computes (i.e., what does “**sum**” store at the end)? What are *two* reasons why this code runs slowly? Rewrite the code to run faster. (4 points)

*This code counts the number of occurrences of the letter ‘J’ in a two-dimensional array of strings. Some students mistakenly said that the code counts the number of ‘J’ characters in a three-dimensional array of characters, but this is not correct because the code does **not** count any ‘J’ characters that may occur **after** the ‘\0’ at the end of each string a[i][j].*

The code would have better spatial locality if the order of the i and j loops were reversed, and the call to strlen() were pulled out of the loop. The revised code is:

```

char a[10][20][30];
int i, j, k, sum=0, len;
...
for (i=0; i<10; i++)
  for (j=0; j<20; j++) {
    len = strlen(a[i][j]);
    for (k=0; k<len; k++)
      sum += (a[i][j][k] == 'J');
  }

```

One student had an alternative answer that was clever, replacing the `strlen()` check with a direct comparison to the `'\0'` character:

```

for (i=0; i<10; i++)
  for (j=0; j<20; j++)
    for (k=0; a[i][j][k] != '\0'; k++)
      sum += (a[i][j][k] == 'J');

```

Another student made the code more efficient by performing loop unrolling instead.

8c) Consider the following code for swapping two numbers

```

void swap(int i, int j) {
    int t;

    t = i;
    i = j;
    j = t;
}

int main(void) {
    int a = 5, b = 10;
    printf("a=%d, b=%d\n", a, b);
    swap(a,b);
    printf("a=%d, b=%d\n", a, b);
}

```

What does the program produce as output? What is the bug? Rewrite the code with the bug fixed. Show the modification of `main()` to call your new version of `swap()`. (3 points)

The program prints "a=5, b=10" twice, because `swap()` is only swapping the local copies of the arguments `i` and `j`, because C has call-by-value semantics. The code should be rewritten as:

```

void swap(int* i, int* j) {
    int t;

    t = *i;
    *i = *j;
    *j = t;
}

```

```

int main(void) {
    int a = 5, b = 10;
    printf("a = %d, b=%d\n", a, b);
    swap(&a, &b);
    printf("a = %d, b=%d\n", a, b);
}

```

8d) This question follows up on question **8c**. Rewrite the swap code to be “generic” (i.e., to work on any type of input data), and show the necessary code to call your new version of `swap()` to swap two integers. (6 points)

*Making swap() generic is challenging because the swap() function does not know the type of the data stored at *i and *j. So a truly generic swap() function must rely on its client to supply (1) a function for creating a new temporary object of the proper type, (2) a function for assigning one object of the proper type to another, and (3) a function for destroying the temporary object. The complete program thus would be this:*

```

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void *intAllocate(void) {
    void *item;
    item = malloc(sizeof(int));
    assert(item != NULL);
    return item;
}

void intDeallocate(void *item) {
    free(item);
}

void intAssign(void *item1, const void *item2) {
    int *int1 = (int *) item1;
    int *int2 = (int *) item2;
    *int1 = *int2;
}

void swap(void *i, void *j,
          void *(*allocate)(void),
          void (*deallocate)(void *item),
          void (*assign)(void *item1, const void *item2)) {
    void *t;

    t = (*allocate)();
    (*assign)(t, i);
    (*assign)(i, j);
    (*assign)(j, t);
    (*deallocate)(t);
}

```

```

}

int main(void) {
    int a = 5, b = 10;
    printf("a = %d, b=%d\n", a, b);
    swap(&a, &b, intAllocate, intDeallocate, intAssign);
    printf("a = %d, b=%d\n", a, b);
    return 0;
}

```

If we can assume that the temporary object is in the heap and was allocated by a single call of malloc(), calloc(), or realloc(), then the client need not supply a "deallocate" function:

```

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void *intAllocate(void) {
    void *item;
    item = malloc(sizeof(int));
    assert(item != NULL);
    return item;
}

void intAssign(void *item1, const void *item2) {
    int *int1 = (int *) item1;
    int *int2 = (int *) item2;
    *int1 = *int2;
}

void swap(void *i, void* j,
          void *(*allocate)(void),
          void (*assign)(void *item1, const void *item2)) {
    void *t;

    t = (*allocate)();
    (*assign)(t, i);
    (*assign)(i, j);
    (*assign)(j, t);
    free(t);
}

int main(void) {
    int a = 5, b = 10;
    printf("a = %d, b=%d\n", a, b);
    swap(&a, &b, intAllocate, intAssign);
    printf("a = %d, b=%d\n", a, b);
    return 0;
}

```

But a truly generic swap() function could not make those assumptions.

A much simpler alternative is to have the client provide the temporary variable. Then the client would not need to supply functions to create or destroy the temporary object. For example, main() could have a third variable “t” and pass “&t” to swap():

```
#include <stdio.h>
#include <assert.h>

void intAssign(void *item1, const void *item2) {
    int *int1 = (int *) item1;
    int *int2 = (int *) item2;
    *int1 = *int2;
}

void swap(void *i, void *j, void *t,
          void (*assign)(void *item1, const void *item2)) {
    (*assign)(t, i);
    (*assign)(i, j);
    (*assign)(j, t);
}

int main(void) {
    int a = 5, b = 10, t;
    printf("a = %d, b=%d\n", a, b);
    swap(&a, &b, &t, intAssign);
    printf("a = %d, b=%d\n", a, b);
    return 0;
}
```

Common mistake: Many student wrote the swap() function like this:

```
void swap (void *i, void *j) {
    void *t;
    t = j;
    j = i;
    i = t;
}
```

That function swaps the values of the function's formal parameters, but does not affect the values of the client's corresponding actual parameters. From the client's point of view, calling that function has no effect at all.

8e) Consider the following function `foo()`:

```
int foo(unsigned num) {
    int i;

    for (i = 0; num; i++)
        num &= (num-1);

    return i;
}
```

State concisely what function `foo()` returns. Do *not* describe how the function works, just what it computes – your answer should be no more than 10 words long. (5 points)

The function `foo()` returns the number of “1” bits in `num`.

[An explanation of why: The integer `i` stores the count. The loop counts one “1” bit (by incrementing `i`) on each iteration, stopping when `num` becomes 0. The manipulation “`num &= (num-1)`” zeroes out the least-significant “1” bit in the number 1. To see how, imagine `num` is “1111”. Then, `num-1` is “1110”, and the ANDing of the two numbers is 1110 (zeroing out the last bit). Now, imagine `num` is “111100”. Then, `num-1` is “111011” and the ANDing of the two numbers is 111000 (zeroing out the third-to-last bit). Essentially, any trailing 0 bits in `num` get turned into “1” bits when doing “`num-1`” (because of the need to “borrow a 1” from the previous column), and the least-significant “1” becomes a “0” (because the “1” was “borrowed” to perform the subtraction).]