Princeton University
COS 217: Introduction to Programming Systems
Spring 2009 Final Exam Answers

The exam was a three-hour, open-book, open-notes exam.

Question 1

The *preprocessor* inserts the contents of /usr/include/stdio.h.

The *linker* includes the code that implements scanf().

The *compiler* ensures that the first argument of printf() is of type "char *".

The *preprocessor* removes the comment "This is my cool program".

The *compiler* checks that "return 0" returns an integer.

The *compiler* translates 'i--' into the "decl" instruction.

The *assembler* determines the argument for the "jmp" instruction to determine how far to jump to get the start of the while loop.

The *linker* determines the address to call to invoke the scanf() function.

Question 2 Part A

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.

Question 2 Part B

The *hardware* is responsible for mapping virtual address into a physical address for pages already in physical memory

The *operating system* is responsible for deciding which virtual page to "swap out" of physical memory on a "miss"

The *operating system* is responsible for updating the page tables with new virtual-to-physical page mappings

The *hardware* is responsible for preventing one user process from accessing the pages of another user process

Question 2 Part C

Spatial locality

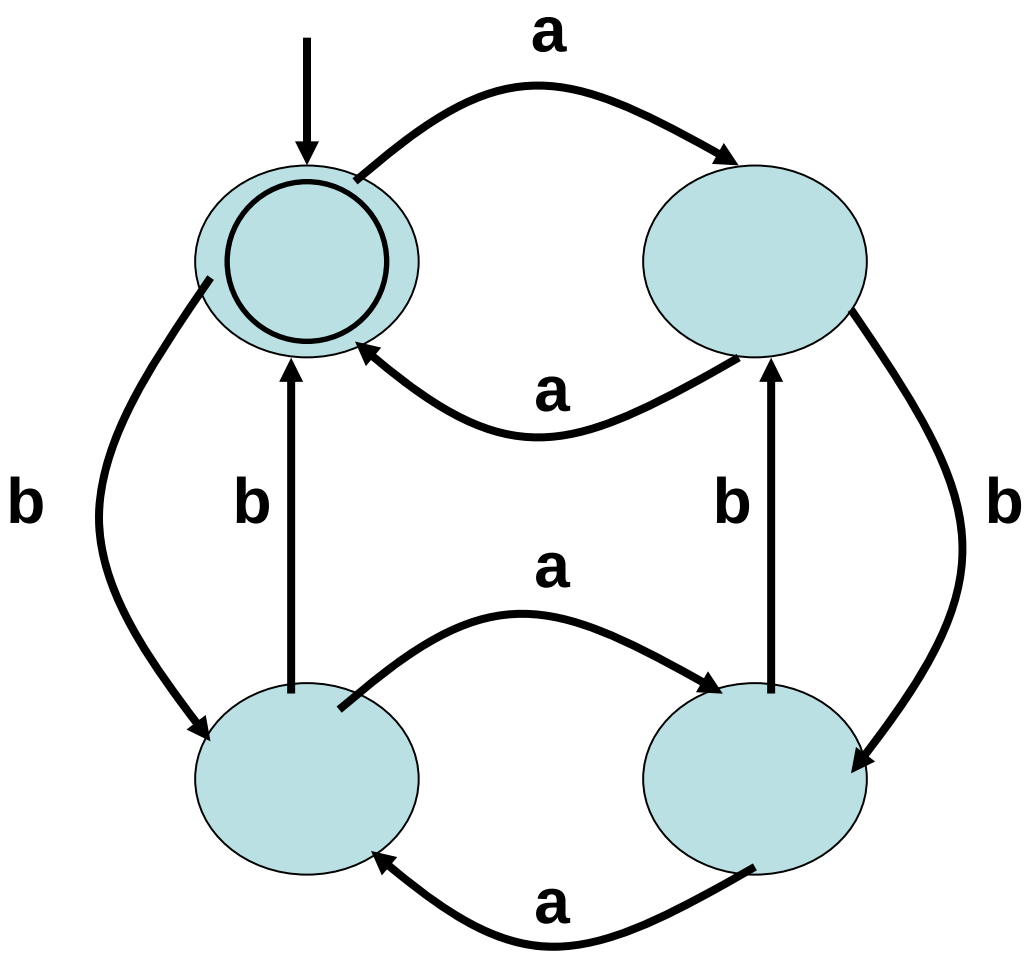
Question 2 Part D

Slow disk seek time.

Question 2 Part E

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



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 Part A

Integer multiplication can be implemented through repeated addition, as follows

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

Question 4 Part B

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;
}
```

Question 4 Part C

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 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 Part A

"Values stored in registers are saved so they can be restored later" is true for *both* a function call and a context switch.

"Control of the computer transfers to the operating system" is true for a *context switch*.

"The instruction pointer (EIP) changes to execute instructions in a new location" is true for *both* a function call and a context switch.

Question 5 Part B

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.

Question 5 Part C

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.

Question 5 Part D

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 Part A

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.

Question 6 Part B

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

Question 6 Part C

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;".

Question 6 Part D

The preprocessor would detect errors like misspelling the name of an including .h file (e.g., "#include <stdioo.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 or definition prior to the call).

Question 6 Part E

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 Part A

The original parent process prints "cos217\n" once in main(). The first child, created by the fork() in the "if" statement, prints "cos217\n" twice, once in main() and once in the "if" clause. 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 1 + 2 + 2 + 2*2 + 4*2, for a grand total of 17.

Dondero Supplement 1:

This statement:

```
if (! (pid = fork())) ...
```

forks a child process. Then:

-- In the parent process, fork() returns the process id of the child process, which is non-zero. The assignment operator assigns that non-zero value to pid, and itself evaluates to that non-zero value. The "!" operator interprets the non-zero value as meaning TRUE. The "!" operator, when applied to that non-zero value, evaluates to 0 (alias FALSE). So the condition of the "if" statement is FALSE in the parent process. And so the parent process does not execute the body

of the "if" statement.

-- In the child process, fork() returns 0. The assignment operator assigns 0 to pid, and itself evaluates to 0. The "!" operator interprets 0 as meaning FALSE. The "!" operator, when applied to 0, evaluates to 1 (alias TRUE). So the condition of the "if" statement is TRUE in the child process. So the child process executes the body of the "if" statement.

This code is equivalent:

```
   pid_t iPid;
   ...
   iPid = fork();
   if (iPid == 0) ...
```

Dondero Supplement 2:

One way to trace the program by drawing a "process graph" for it, as shown on page 604 in Chapter 8 of the Bryant & O'Hallaron book.

Let's number the four calls of fork() 1, 2, 3, and 4, like this:

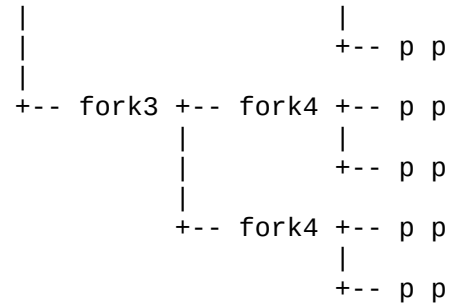
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void cos217(void) {
   pid_t pid;
   if (!(pid = fork())) {   <-- 1
      fork();               <-- 2
      fork();               <-- 3
      fork();               <-- 4
      printf("cos217\n");
   }
}

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

Let's use "p" to indicate a call of printf(). Then here's a process graph for that program:

```
-- fork1 +-- p
         |
         +-- fork2 +-- fork3 +-- fork4 +-- p p
                   |         |         |
                   |         |         +-- p p
                   |         |
                   |         +-- fork4 +-- p p
```



Note that there are 17 "p"s in the graph.

Question 7 Part B

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 Part A

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;
```

Question 8 Part B

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.

Question 8 Part C

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);
}
```

Question 8 Part D

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.

Question 8 Part E

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).