Princeton University
COS 217: Introduction to Programming Systems
Spring 2008 Final Exam Answers

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

Question 1 Part A

The "i = j" assigns i the value of j, rather than comparing the two integers. The correct code would do "i == j". The original code incorrectly outputs "Equal" when j!=0 and i!=j, and incorrectly outputs "Not equal" when i==j==0.

Question 1 Part B

Casting a float to an int converts floats greater than -1 and less than 1 to 0, leading the function to return the wrong value when *fp1 and *fp2 are different, but differ by less than 1. The correct code would use if statements to return (say) 1, 0, of -1 if *fp1 is greater than, equal to, or less than *fp2, respectively. Also, depending on the size of floats and ints, the code might produce the wrong answer when the difference between the two floats is larger than the range of int; the correction to compare the floats directly would also address this problem.

Question 1 Part C

The for loop goes through the number two bits at a time, with the bit operations extracting the last bit ("x & 1") and second-to-last bit ("x & 2"), respectively, and XORing them to produce a 1 if they are different. However, the "x & 2" leaves the second-to-last bit in the second-to-last bit position, so the two bits are not XORed with each other. Instead, "x & 2" should be changed to "((x & 2) >> 1)".

Question 1 Part D

When str[] is an empty string (i.e., str[0] is ‘\0’), the code erroneously modifies str[-1]. The code should specify how the empty string is handled. If the specification says that an empty string should be left unmodified, then the last line of code should be "if (i > 0) str[--i] – ‘\0’;" instead.

Question 2 Part A

Even if backwards compatibility were not an issue, assigning a code to EOF is problematic, as this value might reasonably appear in binary data inputs.

Question 2 Part B

A large page amortizes the cost of accessing the disk, but also consumes a larger amount of physical memory, perhaps forcing the eviction of other data that may be useful.

Question 2 Part C

The page tables are kept in the OS’s virtual address space to prevent user processes from accessing or modifying them; this is important for protecting processes from each other. The pages are pinned in physical memory so the hardware knows where to access them in translating virtual addresses to physical addresses, and to avoid a circular dependency where the page tables are needed to determine where the page tables have been placed in physical memory.

Question 2 Part D

Most jumps move the instruction pointer a small distance in the code (e.g., to the beginning of an "else" clause or the start/end of a loop. Representing the jump as a displacement allows a more concise encoding of the jump instructions, e.g., with a 1-byte displacement rather than a 4-byte address. Also, using displacements allows the assembler to completely translate some jump instructions that it otherwise could not.

Question 2 Part E

Similarity: Both involve executing code at a new location, requiring registers to be saved and the instruction pointer loaded with the address of the new code to run. (They also look similar to the programmer, though this relates more to how they are invoked by the programmer than it does to how they are actually executed.)

Difference: The function call is executed within the user process (in non-privileged mode), whereas a system call involves a context switch so the operating system can execute the code (in privileged mode) on behalf of the user process.

Question 2 Part F

References to global variables in other .o files.
References to function calls in other .o files.

Question 2 Part G

A "good fit" strategy avoids the fragmentation of memory that a "first fit" strategy often causes, while being much faster than a "best fit" strategy that may have to traverse the entire free list to locate the best free block.

Question 3 Part A

In the function cos217(), the first fork creates one child process. The parent fails the "if" test, but the child passes it (because the child has a pid of 0). The child then forks, creating a third process. The first and second child each print within the if clause. Then, all three processes return and print in main(). This results if "cos217" printing five (5) times.

Question 3 Part B

The program cos217.c is 19 lines long. Hence, the concatenation of the file with itself is 38 lines long. Piping the 38 lines to "wc –l" produces an output of "38".

Question 3 Part C

The output of the first call to "wc –l" is one (1) line long, so the output is "1".

Question 4 Part A

```
int timediff(struct timeval t1, struct timeval t2) {
 /* Returns time differences in seconds, rounded up */
   float sec = t2.tv_sec - t1.tv_sec + 1.0E-6F * (t2.tv_usec - t1.tv_usec);

   if ((sec - (int) sec) > 0)
     return (int) (sec+1);
   else
     return (int) sec;
}
```

Question 4 Part B

```
#define _GNU_SOURCE

#include <stdio.h>
#include <sys/time.h>
#include <signal.h>
#include <assert.h>
#include <stdlib.h>

struct timeval t1, t2;

int timediff(struct timeval t1, struct timeval t2) {
  /* Returns time differences in seconds, rounded up */
  float sec = t2.tv_sec - t1.tv_sec + 1.0E-6F * (t2.tv_usec - t1.tv_usec);

  if ((sec - (int) sec) > 0)
    return (int) (sec+1);
  else
    return (int) sec;
}

void handle(int iSignal) {
  int difference;

  gettimeofday(&t2, NULL);
  difference = timediff(t1, t2);

  if (difference >= 80)
    exit(1);

  while (difference--)
    putchar('*');
  printf("\n");

  t1.tv_sec = t2.tv_sec;
  t1.tv_usec = t2.tv_usec;
}

int main() {
  void (*pfRet)(int);

  /* Get initial time (before installing signal handler) */
  gettimeofday(&t1, NULL);

  /* Install signal handler */
  pfRet = signal(SIGINT, handle);
  assert(pfRet != SIG_ERR);

  while (1) {
  }
}
```

Question 5 Part A

Answer #1, where NANDing a number with "11111111" inverts the bits

```
   LDI 0xFF
   STORE (0)
   IN
   NANDM (0)
   OUT
```

Answer #2, where NANDing a number with itself inverts the bits

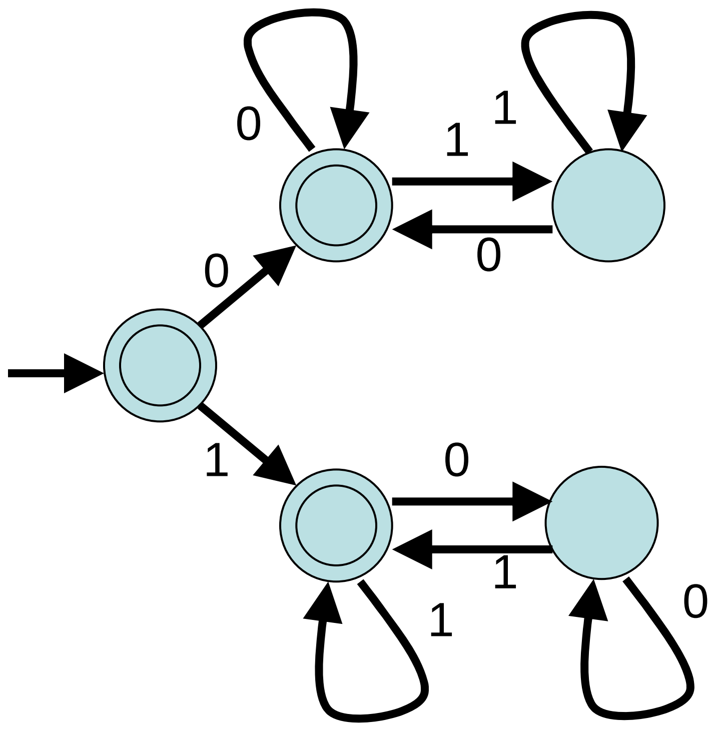
```
   IN
   STORE (0)
   NAND (0)
   OUT
```

Question 5 Part B

The code loops through the bits from right to left by shifting right to move the next bit into the rightmost bit position. Within each iteration of the loop, extract that bit by masking out the others (by NANDing with 00000001, and inverting the result as above), and display the result.

```
      LDI 0x01        # Create and store bitmask 00000001
      STORE (0)
      LDI 0xFF        # Create and store bitmask 11111111
      STORE (1)
      IN              # Read and store the input
      STORE (2)
LOOP: NANDM (0)       # Extract the bit, and output it
      NANDM (1)
      OUT
      READ (2)        # Prepare for the next bit
      SHR
      STORE (2)
      JNZ LOOP
```

Question 6


0
1
1
0
0
1
0
1
1
0