Princeton University
COS 217: Introduction to Programming Systems
Fall 2006 Final Exam Answers

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

Question 1

```
(1)(a) Q
(1)(b) 81
(1)(c) 81
(2)(a) 240
(2)(b) -16 (assuming two's complement notation)
(3)(a) A
(3)(b) 65
(3)(c) 65
(4) The ASCII character set covers the range 0x00 through 0x7F only.
```

Question 2

This answer is courtesy of Chang Kim.

(1) 15

```
(2) ------------------
    n   5
    m   0   foo(5, 0)
    ------------------
    n   5
    m   1   foo (5, 1)
    ------------------
    n   5
    m   2   foo(5, 2)
    ------------------
    n   5
    m   3   foo(5, 3)
    ------------------
```

(3) foo(n, m) computes the value of n*m when m is non-negative. When m is negative foo(n, m) returns zero regardless of the value of n. This function has two limitations:
-- The value of n*m can be incorrect when n*m is larger than 2^31-1 (integer overflow).
-- When m is a very large integer, a stack overflow can happen.

Question 3

Some of parts of this answer are courtesy of David August, Ph.D.

(1) A race condition is a flaw in a program whereby the output of the program is unexpectedly and critically dependent on the sequence or timing of other events.

(2) Registers and memory are both examples of hardware that can store data.

(3) An "exception" is a sudden change in control flow in response to some change in the processor's state. An "interrupt" is a type of exception. An interrupt occurs asynchronously when the CPU receives a signal from an external device.

(4) Together the C "if" and "goto" statements are sufficient to implement any arbitrary control flow construct.

(5) Overflow has occurred during the addition of unsigned integers u and v if and only if the sum is less than u (or v).

(6) Yes. x86 is more CISC than RISC. Quoting the lecture notes, the x86 architecture "interprets instructions of various lengths, and is designed to economize on memory (size of instructions)."

(7) No. Fixed length instructions are more RISC than CISC. Quoting the lecture notes, with RISC "instructions all the same size and all the same format, designed to economize on decoding complexity (and time, and power drain)."

(8) Hardware that implements the cosine function would be considered more CISC than RISC. Implementing such an infrequently-used special-purpose instruction in hardware would imply decoding complexity, contrary to the goals of the RISC approach.

(9) By Amdahl's Law, the maximum speedup is

$$\lim_{s\to\infty} 1/(1-f+f/s) = 1/(1-f) = 1/(1-0.95) = 1/0.05 = 20 \text{ times}$$

Question 4

This is a text-only version of the DFA:

```
State CLOSED (the start state)
   F:  go to OPEN
   R:  go to CLOSED
   B:  go to CLOSED
   N:  go to CLOSED

State OPEN
   F:  go to OPEN
   R:  go to OPEN
   B:  go to OPEN
   N:  go to CLOSED
```

Question 4 Extra Credit

This is a text-only version of the DFA:

```
State CLOSED (the start state)
   F:  go to OPENING
   R:  go to CLOSED
   B:  go to CLOSED
   N:  go to CLOSED
   O:  go to ERROR
   C:  go to CLOSED

State OPENING
   F:  go to OPENING
   R:  go to STOPPED_MIDWAY
   B:  go to STOPPED_MIDWAY
   N:  go to CLOSING
   O:  go to OPEN
   C:  go to ERROR

State OPEN
   F:  go to OPEN
   R:  go to OPEN
   B:  go to OPEN
   N:  go to CLOSING
   O:  go to OPEN
   C:  go to ERROR

State CLOSING
   F:  go to OPENING
   R:  go to STOPPED_MIDWAY
   B:  go to STOPPED_MIDWAY
   N:  go to CLOSING
   O:  go to ERROR
   C:  go to CLOSED
```

```
State STOPPED_MIDWAY
   F:  go to OPENING
   R:  go to STOPPED_MIDWAY
   B:  go to STOPPED_MIDWAY
   N:  go to CLOSING
   O:  go to ERROR
   C:  go to ERROR

State ERROR
   (No outgoing transitions)
```

Question 5

This answer is courtesy of Chang Kim.

(1) Line 20 should be ".equ IBUF, -256".

(2) Lines 42 and 46 should be "leal IBUF(%ebp), %eax" followed by "pushl %eax".

(3) Line 69 should be "movb IBUF(%ebp,%eax,1), %ecx"

(4) and (5) Any two of the following three solutions are correct.

(a) Use of gets() at line 43 is vulnerable to buffer overrun when a user inputs a long string. In this code's case, a string which is longer than 256 bytes will overwrite the return address of the main function, resulting in possible segmentation fault at run time.

(b) At line 67-68, the elements of uiFreq are modified via an index variable tc. The value of tc, however, can be negative or larger than the size of uiFreq when a non-alphabetical character is given in the input string. This results in a write attempt on a memory segment that is protected by the system such as TEXT or RODATA section.

(c) When gets() returns NULL and does not change the contents of the buffer, then it can also cause a segmentation fault because the subsequent instructions access the contents of buf[] without verifying the return value of gets(). Specifically, this case can happen when a user inputs an EOF without any leading characters before it (e.g., by typing Ctrl-D).

Question 6

Many answers are acceptable. These are examples:

(1)

```
/* Write the byte 0x00 followed by the byte 0x01 to stream psFile. */
FILE *psFile;
unsigned short usData = 0x0001;
...
fwrite(&usData, 2, 1, psFile);
```

(2)

```
/* Allocate enough memory to store an integer. */
int *pi;
pi = (int)malloc(2);
```

(3)

```
/* Request 1024 more bytes of heap memory. */
sbrk(1024);
```

(4)

```
/* Print the character 'A' */
putchar(97);
```

Question 7

(1)

This answer is courtesy of Guilherme Ottoni.

```
#define _GNU_SOURCE
/* Use the "new style" implementation of signal handling.
   Not really necessary in this program. */
#include <stdio.h>
#include <signal.h>
#include <assert.h>
#include <unistd.h>

static int count;

static void print_message(void)
{
  printf("\nIterated %d times.\n", count);
  printf("Press CTRL-C to restart timer...\n");
}

static void ctrl_C_handler(int iSignal)
{
  void (*pfRet)(int);
  pfRet = signal(SIGINT, SIG_IGN);
  assert(pfRet != SIG_ERR);

  count = 0;

  alarm(1);
}

static void alarm_handler(int iSignal)
{
  void (*pfRet)(int);

  print_message();

  pfRet = signal(SIGINT, ctrl_C_handler);
  assert(pfRet != SIG_ERR);
}

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

  /* Register the signal handlers. */
  pfRet = signal(SIGALRM, alarm_handler);
  assert(pfRet != SIG_ERR);
  pfRet = signal(SIGINT, ctrl_C_handler);
  assert(pfRet != SIG_ERR);

  printf("Press CTRL-C to start timer...\n");

  while (1) {
    count++;
  }
  return 0;
}
```

(2) One way is to type crtl-\ to send the process a SIGQUIT signal. Another way is to type ctrl-Z to send the process a SIGTSTP signal, and thereby place the process in the background. Then use the ps command to determine the process's id, and issue the "kill -SIGKILL *processid*" command to kill the process.

(3) SIGPROF would be more appropriate because we want to measure the processor's speed, without including the time consumed by other processes, I/O operations, etc.