Princeton University
COS 217: Introduction to Programming Systems
Fall 2007 Final Exam Answers

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

Question 1 Part 1

10101010
SEEEEMMM

$N = (-1)^{S} * 1.M * 2^{E-7}$

$= (-1)^{1} * 1.010 * 2^{5-7}$

$= -1 * 1.010 * 2^{-2}$

$= -1 * 101 * 2^{-4}$

$= -5/16$

Question 1 Part 2

10000010
SEEEEMMM

$N = (-1)^{S} * 0.M * 2^{-7+1}$

$= (-1)^{1} * 0.010 * 2^{-6}$

$= -1 * 0.010 * 2^{-6}$

$= -1 * 1 * 2^{-8}$

$= -1/256$

Question 1 Part 3

$N = (-1)^{S} * 0.M * 2^{-(2\text{^}10-1) + 1}$

$= (-1)^{S} * 0.M * 2^{-(1024-1) + 1}$

$= (-1)^{S} * 0.M * 2^{-1023 + 1}$

$= (-1)^{S} * 0.M * 2^{-1022}$

Question 1 Part 4

$N = (-1)^{S} * 1.M * 2^{E-(2\text{^}10-1)}$

$N = (-1)^{S} * 1.M * 2^{E-(1024-1)}$

$N = (-1)^{S} * 1.M * 2^{E-1023}$

Question 2 Part 1

3

Question 2 Part 2

```
        --------
fib(1)  temp   0
        n      1
        --------
fib(2)  temp   0
        n      2
        --------
fib(3)  temp   0
        n      3
        --------
```

Alternate answer:

```
        --------
fib(0)  temp   0
        n      0
        --------
fib(2)  temp   1
        n      2
        --------
fib(3)  temp   0
        n      3
        --------
```

Question 3 Part 1

Given: There are 2^{32} addressable bytes.

Given: There are 2^{12} bytes per page.

So there are 2^{20} addressable pages, and there must be 2^{20} entries in a process's page table.

Given: Each page table entry is 32 bits (alias 4 bytes) long.

So a process's page table consumes 2^{20} entries * 4 bytes per entry = 2^{22} bytes.

Question 3 Part 2

1 GB = 2^{30} bytes. $2^{22}/2^{30} = 1/2^8 = 1/256$. So each page table would consume 1/256 of physical memory.

So as long as the number of processes is substantially below 256, the virtual memory scheme is practical on a laptop with 1 GB of physical memory installed.

Question 3 Part 3

Given: There are 2^{64} addressable bytes.

Given: There are 2^{12} bytes per page.

So there are 2^{52} addressable pages, and there must be 2^{52} entries in a process's page table.

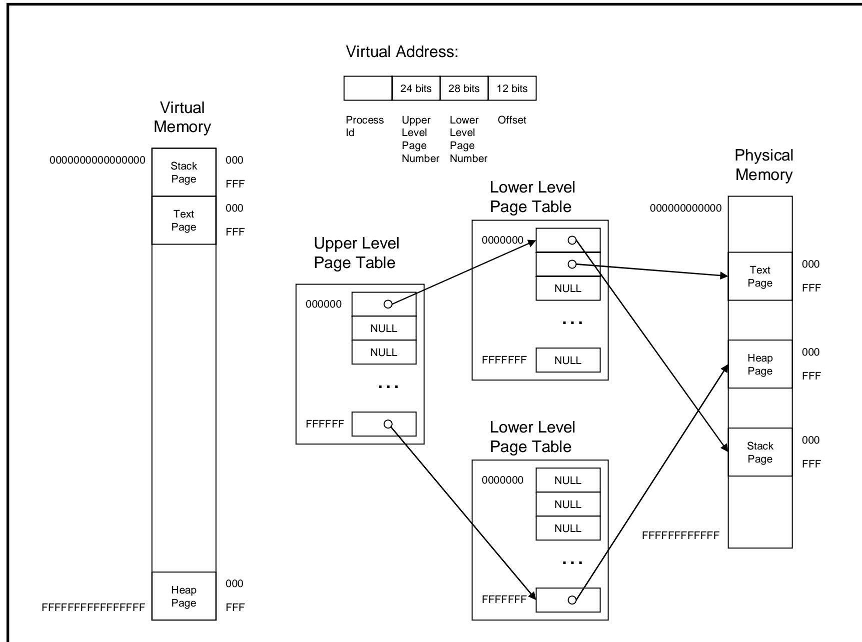
Given: Each page table entry is 48 bits (alias 6 bytes) long.

So a process's page table consumes 2^{52} entries * 6 bytes per entry = 2^{52} * 2 * 3 bytes = 3 * 2^{53} bytes.

Question 3 Part 4

8 GB = 8 * 2^{30} bytes = 2^{33} bytes. 2^{33} bytes is much less than 3 * 2^{53} bytes. So the page table for a single process would not fit in all of physical memory. So the virtual memory scheme is impractical on a server with 8 GB of physical memory installed.

Question 3 Part 5



Question 4

This is the DFA in textual form:

```
START_STATE (S)
  H:  H_STATE
  D:  D_STATE
  other:  START_STATE

H_STATE
  u:  Hu_STATE
  H:  H_STATE
  D:  D_STATE
  other:  START_STATE

Hu_STATE
  m:  Hum_STATE
  H:  H_STATE
  D:  D_STATE
  other:  START_STATE

Hum_STATE
  p:  Hump_STATE
  H:  H_STATE
  D:  D_STATE
  other:  START_STATE

Hump_STATE
  t:  Humpt_STATE
  H:  H_STATE
  D:  D_STATE
```

```
  other:  START_STATE

Humpt_STATE
  y:  Humpty_STATE
  H:  H_STATE
  D:  D_STATE
  other:  START_STATE

Humpty_STATE
  D:  HumptyD_STATE
  other:  Humpty_STATE

HumptyD_STATE
  u:  HumptyDu_STATE
  D:  HumptyD_STATE
  other:  Humpty_STATE

HumptyDu_STATE
  m:  HumptyDum_STATE
  D:  HumptyD_STATE
  other:  Humpty_STATE

HumptyDum_STATE
  p:  HumptyDump_STATE
  D:  HumptyD_STATE
  other:  Humpty_STATE

HumptyDump_STATE
  t:  HumptyDumpt_STATE
  D:  HumptyD_STATE
  other:  Humpty_STATE

HumptyDumpt_STATE
  y:  SUCCESS_STATE
  D:  HumptyD_STATE
  other:  Humpty_STATE

D_STATE
  u:  Du_STATE
  D:  D_STATE
  H:  H_STATE
  other:  START_STATE

Du_STATE
  m:  Dum_STATE
  D:  D_STATE
  H:  H_STATE
  other:  START_STATE

Dum_STATE
  p:  Dump_STATE
  D:  D_STATE
  H:  H_STATE
  other:  START_STATE

Dump_STATE
  t:  Dumpt_STATE
  D:  D_STATE
  H:  H_STATE
  other:  START_STATE

Dumpt_STATE
  y:  Dumpty_STATE
  D:  D_STATE
  H:  H_STATE
  other:  START_STATE

Dumpty_STATE
  H:  DumptyH_STATE
  other:  Dumpty_STATE
```

```
DumptyH_STATE
  u:  DumptyHu_STATE
  H:  DumptyH_STATE
  other:  Dumpty_STATE

DumptyHu_STATE
  m:  DumptyHum_STATE
  H:  DumptyH_STATE
  other:  Dumpty_STATE

DumptyHum_STATE
  p:  DumptyHump_STATE
  H:  DumptyH_STATE
  other:  Dumpty_STATE

DumptyHump_STATE
  t:  DumptyHumpt_STATE
  H:  DumptyH_STATE
  other:  Dumpty_STATE

DumptyHumpt_STATE
  y:  SUCCESS_STATE
  H:  DumptyH_STATE
  other:  Dumpty_STATE

SUCCESS_STATE (F)
  all:  SUCCESS_STATE
```

Question 5 Part 1

In line 1, ".roadata" should be ".rodata".

Question 5 Part 2

The loop within the convert_byte() function prints the least significant binary digit, followed by the next least significant binary digit, etc. It should print the most significant binary digit, followed by the next most significant binary digit, etc.

Question 5 Part 3

The main() function is missing its epilog. That is, the main() function fails to restore the values of the EBP and ESP registers, and fails to pop the old value of EBP from the stack. Thus, when the ret instruction executes, the item at the top of the stack is not the return address. So flow of control returns to an improper place in memory.

Question 5 Part 4

```
.section ".rodata"
cFormat1:
        .asciz "%d"
        .align 4
cFormat2:
        .string "Enter a number between 0 and 255: "

        .section ".text"
## Formal parameter offset
        .equ BYTE, 8

## Local variable offset
        .equ COUNT, -4

## Constants
        .equ INITIAL_COUNT, 8
        .equ MASK, 0x80
        .globl convert_byte
        .type convert_byte, @function
convert_byte:
        pushl %ebp
        movl %esp, %ebp
```

```
        pushl %ebx
        pushl %esi
        movl $INITIAL_COUNT, %esi
        movl BYTE(%ebp), %eax
        movl %eax, %ebx
        jmp .L2

.L3:
        movsx %bl, %eax
        andl $MASK, %eax
        shrl $7, %eax
        movl %eax, 4(%esp)
        movl $cFormat1, (%esp)
        call printf
        movsx %bl, %eax
        sall %eax
        movl %eax, %ebx

.L2:
        subl $1, %esi
        cmpl $-1, %esi
        jne .L3
        movl $10, (%esp)
        call putchar
        popl %esi
        popl %ebx
        movl %ebp, %esp
        popl %ebp
        ret

## Local variable offset
        .equ MAINBYTE, -5
        .equ NUM, -8

## Constants
        .equ INITIAL_COUNT, 8

        .globl main
        .type main, @function
main:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        movl $cFormat2, (%esp)
        call printf
        leal NUM(%ebp), %eax
        movl %eax, 4(%esp)
        movl $cFormat1, (%esp)
        call scanf
        movl NUM(%ebp), %eax
        movb %al, MAINBYTE(%ebp)
        movsx MAINBYTE(%ebp),%eax
        movl %eax, (%esp)
        call convert_byte
        movl $0, %eax
        addl $8, %esp
        movl %ebp, %esp
        popl %ebp
        ret
```

Question 5 Part 5

The movsx (move with sign extension) instruction copies the contents of the source operand to the destination operand, and sign extends the value throughout the destination operand.

Question 5 Part 6

```
.section ".rodata"
cFormat1:
        .asciz "%d"
        .align 4
cFormat2:
        .string "Enter a number between 0 and 4294967295: "

        .section ".text"
## Formal parameter offset
        .equ BYTE, 8

## Local variable offset
        .equ COUNT, -4

## Constants
        .equ INITIAL_COUNT, 32
        .equ MASK, 0x80000000
        .globl convert_byte
        .type convert_byte, @function
convert_byte:
        pushl %ebp
        movl %esp, %ebp
        pushl %ebx
        pushl %esi
        movl $INITIAL_COUNT, %esi
        movl BYTE(%ebp), %eax
        movl %eax, %ebx
        jmp .L2

.L3:
        movl %ebx, %eax
        andl $MASK, %eax
        shrl $31, %eax
        movl %eax, 4(%esp)
        movl $cFormat1, (%esp)
        call printf
        movl %ebx, %eax
        sall %eax
        movl %eax, %ebx

.L2:
        subl $1, %esi
        cmpl $-1, %esi
        jne .L3
        movl $10, (%esp)
        call putchar
        popl %esi
        popl %ebx
        movl %ebp, %esp
        popl %ebp
        ret

## Local variable offset
        .equ MAINBYTE, -8
        .equ NUM, -8

## Constants
        .equ INITIAL_COUNT, 8

        .globl main
        .type main, @function
main:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        movl $cFormat2, (%esp)
        call printf
```

```
        leal NUM(%ebp), %eax
        movl %eax, 4(%esp)
        movl $cFormat1, (%esp)
        call scanf
        movl NUM(%ebp), %eax
        # Deleted two lines from this place.
        movl %eax, (%esp)
        call convert_byte
        movl $0, %eax
        addl $8, %esp
        movl %ebp, %esp
        popl %ebp
        ret
```

Question 6

The answers to Question 6 Part1 and Part 2 were provided by Lindsey Poole.

The following are example solutions. Your solution will differ.

For full marks, your solution should:

- Not rely on operational precedence.
- Not assume ASCII.
- Not assume data type sizes.
- Remove magic numbers.
- Not use gets().
- Use assert() for function parameters and memory allocations.
- Free allocated memory

We accepted other solutions that ameliorated these problems.

Question 6 Part 1

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

void invertCase(int *buf, int *numChar)
{
   int i;

   /* Don't rely on operational precedence. */
   for(i = 0; i < 1024; i++) {
      /* don't assume ascii */
      if (isupper(buf[i])) {
         buf[i] = tolower(buf[i]);
      } else if (islower(buf[i])) {
         buf[i] = toupper(buf[i]);
      }
   }

   *numChar = i;
}

void getData(int *buf)
{
   /* Don't use gets().  Need to use
    * getc() since buf is int now. */
   int i;
   for (i=0; i < 1024; i++) {
      buf[i] = getc(stdin);
      if (buf[i] == EOF || buf[i] == '\n')
         break;
   }
   buf[i + 1] = 0;
}

void writeData(int *buf)
{
```

```
   int i;
   /* Must use putc() since buf is int now. */
   for (i=0; i < 1024; i++) {
      putc(buf[i], stdout);
      if (buf[i] == 0)
         break;
   }
}

int main(int argc, char *argv[])
{
   int *buf;
   int *numChar;

   /* Don't assume data type sizes.
    * buf is int for internationalization. */
   buf = calloc(1024, sizeof(int));
   numChar = calloc(1, sizeof(int));

   getData(buf);
   invertCase(buf, numChar);
   writeData(buf);

   return 0;
}
```

Question 6 Part 2

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

/* Define MAX_LEN. */
enum {MAX_LEN = 1024};

void invertCase(int *buf, int *numChar)
{
   int i;
   /* Assert buf != NULL. */
   /* Assert numChar != NULL. */
   assert(buf != NULL);
   assert(numChar != NULL);

   /* Use MAX_LEN and null terminator*/
   for(i = 0; i < MAX_LEN && buf[i] != 0; i++) {
      /* don't assume ascii */
      if (isupper(buf[i])) {
         buf[i] = tolower(buf[i]);
         /* Make numchar useful. */
         (*numChar)++;
      } else if (islower(buf[i])) {
         buf[i] = toupper(buf[i]);
         /* Make numchar useful. */
         (*numChar)++;
      }
   }
}

void getData(int *buf)
{
   int i;
   /* Assert buf != NULL. */
   assert(buf != NULL);

   /* Use MAX_LEN. */
   for (i=0; i < MAX_LEN-1; i++) {
      buf[i] = getc(stdin);
      if (buf[i] == EOF || buf[i] == '\n')
```

```
         break;
   }
   buf[i + 1] = 0;
}

void writeData(int *buf, int *numChar)
{
   int i;
   /* Assert buf != NULL. */
   /* Assert numChar != NULL. */
   assert(buf != NULL);
   assert(numChar != NULL);

   /* Use MAX_LEN. */
   for (i=0; i < MAX_LEN; i++) {
      putc(buf[i], stdout);
      if (buf[i] == 0)
         break;
   }

   /* Make numChar do something useful. */
   printf("%d characters converted\n", *numChar);
}

int main(int argc, char *argv[])
{
   int *buf;
   int *numChar;

   /* Use MAX_LEN. */
   buf = calloc(MAX_LEN, sizeof(int));
   numChar = calloc(1, sizeof(int));

   /* Assert buf != NULL. */
   /* Assert numChar != NULL. */
   assert(buf != NULL);
   assert(numChar != NULL);

   getData(buf);
   invertCase(buf, numChar);
   writeData(buf, numChar);

   /* Free buf. */
   /* Free numChar. */
   free(buf);
   free(numChar);

   return 0;
}
```

Copyright © 2008 by Robert M. Dondero, Jr.