Princeton University
COS 217: Introduction to Programming Systems
Spring 2004 Final Exam Answers

Question 1 (a)

A typical instruction consists of an opcode, a source operand, and a destination operand. The execution steps are: fetch instruction, fetch operands, execute the operation, write results, and increment program counter (or instruction pointer).

Question 1 (b)

The operating system creates a process with virtual memory, loads various sections (text, data, bss, etc) of a program into the virtual memory, jumps to the starting instruction of the program, and destroys the process on the exit of the program.

Question 1 (c)

Exceptions are program errors. Examples are faults, and traps. Interrupts are events generated by hardware. Examples are keyboard inputs, and disk I/O operation completion.

Question 1 (d)

A procedure call runs in user mode, whereas a system call looks like a procedure call, but runs in kernel mode.

The typical calling sequence of a procedure call is:

Caller:

- Push arguments onto the stack.
- Call the procedure (including pushing the return address on the stack).
- Get return results (typically from a register).

Callee:

- Setup the stack frame (including local variables).
- Execute the procedure with the arguments and local variables in the stack frame.
- Pop off the stack frame.
- Return to caller (using the saved return address on the stack).

The typical calling sequence of a system call is:

Caller:

- Move arguments to registers or push arguments onto the stack.
- Trap instruction with a system call number.
- Get return results (typically from a register and error code is in a agreed variable).

Callee (in the OS kernel):

The system call mechanism switches the call to the called system call code.

- Setup the stack frame (may need to copy arguments to kernel stack).
- Execute the system call with the arguments.
- Pop off the stack frame.
- Return to caller using a special mode switch instruction (such as IRET).

Question 2

```
#=======================================================================
# sum.s
#=======================================================================

#=======================================================================
   .section ".text"
#=======================================================================
```

```
#-------------------------------------------------------------------
# int sum(int x, int y)
#
# Return the sum of all the numbers between 1 and y that are divisible
# by x.  x and y are positive.
#
# Formal parameter offsets:
   .equ X, 8
   .equ Y, 12
# Local variable offsets:
   .equ I,    -4
   .equ ISUM, -8
#-------------------------------------------------------------------

   .globl sum
   .type sum,@function

sum:

   pushl %ebp
   movl  %esp, %ebp

   # int i = 1;
   pushl $1

   # int iSum = 0;
   pushl $0

loop:

   # if (i > y) goto endloop;
   movl  I(%ebp), %eax
   cmpl  Y(%ebp), %eax
   jg    endloop

   # if ((i % x) != 0) goto endif;
   movl  I(%ebp), %eax
   movl  $0, %edx
   idivl X(%ebp)
   cmpl  $0, %edx
   jne   endif

   # iSum += i;
   movl  I(%ebp), %eax
   addl  %eax, ISUM(%ebp)

endif:

   # i++;
   incl  I(%ebp)

   # goto loop;
   jmp   loop

endloop:

   # return iSum;
   movl  ISUM(%ebp), %eax
   movl  %ebp, %esp
   popl  %ebp
   ret
```

Question 3 (a)

```
#include <stdio.h>

#define ESCAPE_CHAR 0377

enum State {START_STATE, ONE_IN_A_ROW_STATE, TWO_IN_A_ROW_STATE,
            COMPRESS_STATE};
```

```
int main(void)
{
   int iChar;
   int iPrevChar;
   int iCount;
   enum State iState = START_STATE;

   for (;;)
   {
      iChar = getchar();

      switch (iState)
      {
         case START_STATE:
            if (iChar == EOF)
            {
               return 0;
            }
            else if (iChar == ESCAPE_CHAR)
            {
               iPrevChar = iChar;
               iCount = 1;
               iState = COMPRESS_STATE;
            }
            else
            {
               iPrevChar = iChar;
               iState = ONE_IN_A_ROW_STATE;
            }
            break;

         case ONE_IN_A_ROW_STATE:
            if (iChar == EOF)
            {
               putchar(iPrevChar);
               return 0;
            }
            else if (iChar != iPrevChar)
            {
               putchar(iPrevChar);
               iPrevChar = iChar;
               if (iChar == ESCAPE_CHAR)
               {
                  iCount = 1;
                  iState = COMPRESS_STATE;
               }
               else
                  iState = ONE_IN_A_ROW_STATE;
            }
            else
            {
               iState = TWO_IN_A_ROW_STATE;
            }
            break;

         case TWO_IN_A_ROW_STATE:
            if (iChar == EOF)
            {
               putchar(iPrevChar);
               putchar(iPrevChar);
               return 0;
            }
            else if (iChar != iPrevChar)
            {
               putchar(iPrevChar);
               putchar(iPrevChar);
               iPrevChar = iChar;
               if (iChar == ESCAPE_CHAR)
               {
                  iCount = 1;
                  iState = COMPRESS_STATE;
               }
               else
```

```
                  iState = ONE_IN_A_ROW_STATE;
            }
            else
            {
               iCount = 3;
               iState = COMPRESS_STATE;
            }
            break;

         case COMPRESS_STATE:
            if (iChar == EOF)
            {
               putchar(ESCAPE_CHAR);
               putchar(iPrevChar);
               putchar(iCount);
               return 0;
            }
            else if (iChar != iPrevChar)
            {
               putchar(ESCAPE_CHAR);
               putchar(iPrevChar);
               putchar(iCount);
               iPrevChar = iChar;
               if (iChar == ESCAPE_CHAR)
               {
                  iCount = 1;
                  iState = COMPRESS_STATE;
               }
               else
                  iState = ONE_IN_A_ROW_STATE;
            }
            else
            {
               iCount++;
               if (iCount == 255)
               {
                  putchar(ESCAPE_CHAR);
                  putchar(iPrevChar);
                  putchar(iCount);
                  iState = START_STATE;
               }
               else
                  iState = COMPRESS_STATE;
            }
            break;
      }
   }

   return 0;
}
```

Question 3 (b)

```
#include <stdio.h>

#define ESCAPE_CHAR 0377

int main(void)
{
   int i;
   int iChar;
   int iCount;

   while ((iChar = getchar()) != EOF)
   {
      if (iChar == ESCAPE_CHAR)
      {
         iChar = getchar();
         iCount = getchar();
         for (i = 0; i < iCount; i++)
            putchar(iChar);
      }
```

```
      else
         putchar(iChar);
   }

   return 0;
}
```

Question 3 (c)

Produce output that is known to be right or wrong:

- Compress file X to produce file Y. Manually examine Y. Decompress file Y to produce file Z. Use diff to compare X and Z; they should be identical. Repeat for various X...

Boundary condition tests:

- X contains 0, 1, 2, 3, 4 characters, with no consecutive characters the same.
- X contains 1, 2, 3, 4, 254, 255, 256 characters, all of which are 0377.
- X contains 1, 2, 3, 4, 254, 255, 256 characters, all of which are the same.
- X contains only pairs of characters.
- X contains only triplets of characters.

Stress tests:

- X contains a large number of random characters.
- X contains a large number of characters, all of which are 0377.
- X contains a large number of characters, all of which are the same.
- X contains a large number of characters, with no consecutive characters the same.
- X contains a large number of characters, all of which occur in pairs.
- X contains a large number of characters, all of which occur in triplets.

Logical path tests:

- X contains a random number of random characters.

Question 4 (a)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

int main(int argc, char *argv[])
{
   int piPipeFd[2];
   int iProducerPid;
   int iConsumerPid;
   int iRet;

   iRet = pipe(piPipeFd);
   if (iRet == -1) {perror(argv[0]); return 1; }

   fflush(NULL);
   iProducerPid = fork();
   if (iProducerPid == -1) {perror(argv[0]); return 1; }

   if (iProducerPid == 0)
   {
      char *ppcArgv[3] = {"ls", "-l", NULL};

      iRet = close(piPipeFd[0]);
      if (iRet == -1) {perror(argv[0]); exit(1); }
      iRet = close(1);
      if (iRet == -1) {perror(argv[0]); exit(1); }
      iRet = dup(piPipeFd[1]);
      if (iRet == -1) {perror(argv[0]); exit(1); }
      iRet = close(piPipeFd[1]);
      if (iRet == -1) {perror(argv[0]); exit(1); }
```

```
      execvp(ppcArgv[0], ppcArgv);
      perror(argv[0]);
      exit(1);
   }

   fflush(NULL);
   iConsumerPid = fork();
   if (iConsumerPid == -1) {wait(NULL); perror(argv[0]); return 1; }

   if (iConsumerPid == 0)
   {
      char *ppcArgv[2] = {"more", NULL};

      iRet = close(piPipeFd[1]);
      if (iRet == -1) {perror(argv[0]); exit(1); }
      iRet = close(0);
      if (iRet == -1) {perror(argv[0]); exit(1); }
      iRet = dup(piPipeFd[0]);
      if (iRet == -1) {perror(argv[0]); exit(1); }
      iRet = close(piPipeFd[0]);
      if (iRet == -1) {perror(argv[0]); exit(1); }

      execvp(ppcArgv[0], ppcArgv);
      perror(argv[0]);
      exit(1);
   }

   iRet = close(piPipeFd[0]);
   if (iRet == -1) {wait(NULL); wait(NULL); perror(argv[0]); return 1; }
   iRet = close(piPipeFd[1]);
   if (iRet == -1) {wait(NULL); wait(NULL); perror(argv[0]); return 1; }

   wait(NULL);
   wait(NULL);

   return 0;
}
```

Question 4 (b)

Add the statement

```
alarm(360);
```

immediately prior to the call to execvp that executes the “more” program.

Explanation:

- The program must measure wall-clock time, not CPU time. So the program must create an alarm, not an interval timer.
- Alarms are not preserved across a fork. So the alarm must be created within the child process, not within the parent process.
- Alarms are preserved across an exec. So an alarm created in the second child process will be inherited by the “more” program.
- Registration of a user-defined signal handler is not preserved across an exec. So there is no reason to register a user-defined signal handler in the second child process.
- The default handler for the SIGALRM signal causes process exit. Since that is the desired behavior, there is no need to register a user-defined signal handler.

Question 5 (a)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#define N 10000

int MakeData( int x[], float y[], int size )
{
   int c;
   int i;
```

```
   for ( i = 0; i < N && ((c = getchar()) != EOF); i++)
      x[i] = c;
   size = i;
   i = 0;
   while ( i < size )
   {
      y[i] = (float) x[i] + i;
      i++;
   }
   return size;
}

int main(void)
{
   int size;
   int *iBuf;
   float *fBuf;
   iBuf = (int *) malloc(sizeof(int) * N);
   fBuf = (float *) malloc(sizeof(float) * N);
   size = MakeData( iBuf, fBuf, N );
   fwrite( fBuf, sizeof(float), size, stdout );
   return 0;
}
```

Question 5 (b)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#define N 10000

int MakeData( int x[], float y[], int size )
{
   int c;
   int i;
   assert( x != NULL );
   assert( y != NULL );
   for ( i = 0; i < N && ((c = getchar()) != EOF); i++)
      x[i] = c;
   size = i;
   i = 0;
   while ( i < size )
   {
      y[i] = (float) x[i] + i;
      i++;
   }
   return size;
}

int main(void)
{
   int size;
   int *iBuf;
   float *fBuf;
   size_t uiRet;
   iBuf = (int *) malloc(sizeof(int) * N);
   assert( iBuf != NULL );
   fBuf = (float *) malloc(sizeof(float) * N);
   assert( fBuf != NULL );
   size = MakeData( iBuf, fBuf, N );
   uiRet = fwrite( fBuf, sizeof(float), size, stdout );
   assert( uiRet == size );
   return 0;
}
```