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

Question 1 Part a

3x

Explanation:

By Amdahl's Law:

```
overallspeedup = 1 / ((1 - f) + f / s)
```

where:

```
f is the fraction of the program enhanced
s is the speedup of the enhanced portion
```

We must compute the value of s that will yield an overallspeedup of 2. So:

```
2 = 1 / ((1 - .75) + .75 / s)
2 = 1 / (.25 + .75 / s)
2(.25 + .75 / s) = 1
.5 + 1.5 / s = 1
1.5 / s = .5
1.5 = .5s
s = 3
```

Question 1 Part b

4x

Explanation:

If component A took no time at all, then the program would consume only 25% of its former execution time. That is, the program would be 4 times faster.

Question 2

A = B < C < D

C incurs twice as many cache misses as A and B. D incurs a cache miss for almost every single access.

Question 3

Binary files prepared for different operating systems contain different system calls.

Question 4 Part 1

B

Explanation:

The key to tracing the program is to understand these two facts:

(1) The C programming language uses "short circuit boolean evaluation":

- When evaluating an expression of the form A || B, if A is true, then B will not be evaluated. If A is false, then B will be evaluated.
- And, incidentally, when evaluating an expression of the form A && B, if A is true, then B will be evaluated. If A is false, then B will not be evaluated.

(2) The "if" statement considers 0 to mean FALSE, and non-0 to mean TRUE.

Question 4 Part 2

D

Explanation:

writer.c assigns the mathematical value 0x10000000 to x. Since Intel is a little-endian architecture, the number will appear in memory as 00000010. So the fwrite function writes the four bytes 00000010 to the data file.

reader.c's call of fread reads the four bytes 00000010 into x. Since Sun is a big-endian architecture, the mathematical value of x will be precisely that: 00000010. So the call to printf writes 0x followed by the ASCII codes of the hexadecimal digits that comprise that mathematical value: 10.

Question 4 Part 3

A

Explanation:

writer.c assigns the mathematical value 0x10000000 to x. The fprintf function writes 0x followed by the ASCII codes of the hexadecimal digits that comprise that mathematical value: 10000000.

reader.c's call of fscanf reads ASCII codes of hexadecimal digits, uses them to compose the mathematical value 0x10000000, and assigns that mathematical value to x. The call of printf writes 0x followed by the ASCII codes of the hexadecimal digits that comprise that mathematical value: 10000000.

Question 4 Part 4

A, B

Explanation:

Concerning A: If the destination of a call instruction is in the same file and section as the call instruction, then the assembler computes the displacement and places it in the machine language call instruction. If the destination of a call instruction is in a different file or section from the call instruction, then the assembler generates a relocation record; the linker then uses that relocation record to compute the displacement and patch the call instruction.

Concerning B: If the destination of a jump instruction is in the same file and section as the jump, then the assembler computes the displacement and places it in the machine language jump instruction. If the destination of a jump instruction is in a different file or section from the jump instruction, then the assembler generates a relocation record; the linker then uses that relocation record to compute the displacement and patch the jump instruction.

Concerning C: Function return addresses are known only at runtime. Neither the compiler nor the linker computes them.

Concerning D: Through the trap instruction, the assembler and linker are insulated from knowledge of the addresses of operating system code.

Question 4 Part 5

A, B, C

Explanation:

Concerning A: For example, if the name of a library function changes, then you would need to edit, recompile, and relink.

Concerning B: For example, if the type of a library function's formal parameter changes from int to double, then you would not need to edit your code. (The compiler will allow

an int actual parameter to match with a double formal parameter, and will do the type conversion automatically.) But you would need to recompile and relink.

Concerning C: For example, if only the implementation of a library function changes, then you would not need to edit or recompile. But you would need to relink.

Concerning D: If you did nothing, then the program's behavior would not be altered. So you cannot "do nothing."

Question 5

```
handler: 1, 0
handler: 0, 0
handler: 0, 0
handler: 0, 0
...
```

Explanation:

The answer is tricky. It relies on a point that was covered specifically during the Fall 2004 semester.

At program startup, i is initialized to 0. main() assigns NULL (alias 0) to p. main()'s call to signal() registers handler() as the signal handler for SIGSEGV (segmentation fault) signals. main()'s assignment of 1 to *p causes a segmentation fault, so the operating system sends a SIGSEGV signal to the program. When the SIGSEGV signal arrives, handler() executes, printing "handler 1, 0". handler() assigns &i to p, and reregisters handler() as the handler for SIGSEGV signals.

handler() returns, and execution continues with the statement that generated the signal, that is, with the statement "*p = 1;".

Now, here's the tricky part... Actually it's imprecise to say that execution continues with the *statement* that generated the signal. It's more precise to say that execution continues with the *machine language instruction* that generated the signal.

The statement "*p = 1" consists of multiple machine language instructions, probably of this form:

```
   movl $p, %eax
   movl $1, (%eax)
```

It was the second of those instructions that generated the segmentation fault. So execution continues with the second of those instructions.

When handler() assigned &i to p, it affected the value of p in memory. But that assignment did not affect the contents of the EAX register. So EAX still contains NULL (alias 0). So the second of those instructions again generates a segmentation fault.

So the operating system sends a SIGSEGV signal to the program. When the SIGSEGV signal arrives, handler() executes, printing "handler 0, 0". handler() assigns &i to p, and reregisters handler() as the handler for SIGSEGV signals.

handler() returns, and execution continues with the machine language instruction that generated the signal.

Etc., infinitely.

Question 6

First box:

```
        if (fork() == 0) {
           close(p[0]);
           input = open("input", O_RDONLY);
           runCommand("/bin/sort", "sort", input, p[1]);
        }
```

Second box:

```
        if (fork() == 0) {
           close(p[1]);
           close(q[0]);
           runCommand("/usr/bin/uniq", "uniq", p[0], q[1]);
        }
```

Third box:

```
        if (fork() == 0) {
           close(p[0]);
           close(p[1]);
           close(q[1]);
           output = open("output", O_WRONLY|O_CREAT|O_TRUNC, 00600);
           runCommand("/usr/bin/wc", "wc", q[0], output);
        }
```

Fourth box:

```
        close(p[0]);
        close(p[1]);
        close(q[0]);
        close(q[1]);
```

Question 7

```
# some constants that might be of use to you
        .equ INT_SIZE, 4                # size of an integer
        .equ N, 256
        .equ M, 32
        .equ S_SIZE, INT_SIZE*3         # size of struct S
        .equ DATA_OFF, INT_SIZE         # offset of s.data[0] within s

.section .bss
     a:
        .skip INT_SIZE * N              # space for the array a[N]
     s:
        .skip S_SIZE                    # space for struct S s
     ss:
        .skip S_SIZE * M                # space for struct S ss[M]

.section .text
        .globl _start
     _start:

        .equ P, -4

        # struct s *p;
        subl  $4, %esp

        # p = &ss[13];
        movl  $13, %eax
        movl  $S_SIZE, %ecx
        imull %ecx
        addl  $ss, %eax
        movl  %eax, P(%ebp)

        # f(a, s, ss, p->data, &(p->id));

           # Push &(p->id)
           pushl P(%ebp)

           # Push p->data
           movl  P(%ebp), %eax
           addl  $DATA_OFF, %eax
           pushl %eax

           # Push ss
           pushl $ss

           # Push s
           movl  $DATA_OFF, %eax
           movl  $1, %ecx
           pushl s(%eax ,%ecx, INT_SIZE)   # push s.data[1]
           movl  $0, %ecx
           pushl s(%eax ,%ecx, INT_SIZE)   # push s.data[0]
           pushl s                         # push s.id

           # Push a
           pushl $a

           # call the function
           call  f

           # Pop the stack
           addl $28, %esp

        # exit(0);
        movl  $1, %eax
        movl  $0, %ebx
        int   $0x80
```

Question 8 Part a

```
int SymTable_same(SymTable_T oSymtable1,
                  SymTable_T oSymTable2,
                  int (*pfCompare)(void *pvValue1, void *pvValue2));

/* Perform a "deep compare" of oSymTable1 and oSymTable2.  More precisely...
   Return 1 (TRUE) if oSymTable1 and oSymTable2 contain the same bindings, that is,
   the same key/value pairs.  Otherwise return 0 (FALSE).
   The *pfCompare function should compare two given values; it should return 0 if
   the values are equal, and non-0 otherwise. */
```

Question 8 Part b

```
struct CompareInfo
{
    SymTable_T oSymTable2;
    int (*pfCompare)(void *pvValue1, void *pvValue2);
    int iSameSoFar;
};

static void bindingFoundInTable2(char *pcKey, void *pvValue, void *pvExtra)
{
    struct CompareInfo *psCompareInfo = (struct CompareInfo*)pvExtra;
    void *pvValue2;

    if (! SymTable_contains(psCompareInfo->oSymTable2, pcKey))
        pcCompareInfo->iSameSoFar = 0;
    else
    {
        pvValue2 = SymTable_get(psCompareInfo->oSymTable2, pcKey);
        if ((*(psCompareInfo->pfCompare))(pvValue, pvValue2) != 0)
            pcCompareInfo->iSameSoFar = 0;
    }
}

int SymTable_same(SymTable_T oSymTable1,
                  SymTable_T oSymTable2,
                  int (*pfCompare)(void *pvValue1, void *pvValue2))
{
    struct CompareInfo sCompareInfo;

    if (SymTable_getLength(oSymTable1) != SymTable_getLength(oSymtable2))
        return 0;

    sCompareInfo.oSymTable2 = oSymTable2;
    sCompareInfo.pfCompare = pfCompare;
    sCompareInfo.iSameSoFar = 1;

    SymTable_map(oSymTable1, bindingFoundInTable2, (void*)&sCompareInfo);
    return sCompareInfo.iSameSoFar;
}
```