Final Examination

COS 217, Fall 2004

Your name: _____

Your NetID: _____

Your preceptor: _____

There are 8 problems on this examination, worth a total of 120 points. You have 2 hours to complete this examination. (The exam pages are numbered 0-14.)

You are not required to comment your code in this exam unless what you have written is difficult to understand.

This examination is open book and open notes, but no calculators or other electronic devices are permitted.

For some of the problems, partial credit will be given if you can show your work in arriving at solutions. But ***be brief***.

Good luck!

Write out and sign the Honor Code pledge before turning in the test.

"I pledge my honor that I have not violated the Honor Code during this examination."

Problem	Worth	Earned
1	**10**	
2	**5**	
3	**5**	
4	**25**	
5	**10**	
6	**20**	
7	**20**	
8	**25**	
Total	**120**	

1. 10 points

A component of a program (component A) currently accounts for 75% of the program's total running time. Your task is to optimize component A so that the entire program achieves a 2x overall speedup. (Recall that we say we achieve a "2x speedup" when the running time is cut in half.)

(a) How much must you speed up component A to meet this goal?

(b) By optimizing only component A, what is the maximum speedup of the entire program can you possibly hope for?

2. 5 points

Given a large integer array (`int a[N];`), we consider three ways of accessing it:

A. Accessing it sequentially (from `a[0]` to `a[N-1]`);

B. Accessing it in reverse order (from `a[N-1]` to `a[0]`);

C. Accessing all the even-numbered elements first, and then all the odd-numbered elements (`a[0], a[2], . . ., a[N-2], a[1], a[3], . . ., a[N-1]`, assuming `N` is even);

D. Accessing `N` randomly chosen elements.

Give an ordering of the speed of these four methods of access (from the fastest to the slowest). Briefly justify your conclusion. If some of them have roughly equivalent speeds, say so.

3. 5 points

Because Intel-Linux executable files and Intel-Windows executable files both contain machine instructions of the same type (namely, the IA32 instruction set), one might naively think that it should be possible to execute Linux programs on the Windows operating systems and vice versa. Very briefly explain why this is not the case. (Do not use the executable file format difference answer: it is not a critical reason---one could always write a file format translator if it were.)

4. Multiple choice questions. 25 points

For each of the following questions, enumerate ***all*** applicable answers. (No explanation is necessary.)

1)

```
void f(int i) {
    int j;
    if ((j=i%2) || (j=(i+1)%2)) {
        printf("%d\n", j);
    }
}
```

What does this function do?

A. Always prints 0.

B. Always prints 1.

C. Prints nothing.

D. Prints either 0 or 1 depending on the input `i`.

E. Prints 0, 1, or nothing depending on the input `i`.

F. Depends on how the compiler is implemented.

2) Consider the following two programs:

`writer.c`

```
#include <stdio.h>

main() {
    unsigned int x;
    x = 0x10000000;
    fwrite(&x, sizeof(x),
           1, stdout);
}
```

`reader.c`

```
#include <stdio.h>

main() {
    unsigned int x;
    fread(&x, sizeof(x),
          1, stdin);
    printf("0x%x\n", x);
}
```

We execute the following commands on two different machines that share a file system. (Commands on the left machine are typed first.)

`tux` (an Intel machine)

```
% gcc -o writer writer.c
% ./writer > data
```

`bolle` (a Sun machine)

```
% gcc -o reader reader.c
% ./reader < data
```

What's the output produced on `bolle` (the Sun machine)?

A. 0x10000000
B. 0x100000
C. 0x1000
D. 0x10
E. 0x1
F. (Some error message)

3) Consider the following two programs:

`writer.c`

```
#include <stdio.h>

main() {
    unsigned int x;
    x = 0x10000000;
    fprintf(stdout,
            "0x%x", x);
}
```

`reader.c`

```
#include <stdio.h>

main() {
    unsigned int x;
    fscanf(stdin, "%x",
           &x);
    printf("0x%x\n", x);
}
```

We execute the following commands on two different machines that share a file system. (Commands on the left machine are typed first.)

`bolle` (a Sun machine)

```
% gcc -o writer writer.c
% ./writer > data
```

`tux` (an Intel machine)

```
% gcc -o reader reader.c
% ./reader < data
```

What's the output produced on `tux` (the Intel machine)?

A. 0x10000000
B. 0x100000
C. 0x1000
D. 0x10
E. 0x1
F. (Some error message)

4) Which of the following tasks are performed by the assembler and/or the linker?

A. Patching up the destination addresses of jump instructions associated with function calls;
B. Patching up the destination addresses of jump instructions associated with if-then-else statements in C source codes;
C. Patching up the return addresses associated with function call return instructions;
D. Patching up the destination addresses associated with system calls (addresses of operating system code that begin to implement system call services).

5) A C library (a `.a` file) is changed in some way. You have a program that was produced by (statically) linking to this library. The changes that have occurred in the library are supposed to alter the behavior of your program. In order to reflect the latest changes that have occurred in the library, which of the following are possible actions that you may need to take?

A. Edit your source code, recompile, and re-link;
B. Just recompile and re-link;
C. Just re-link;
D. Do nothing.

5. 10 points

What does the following program print?

```
#include <stdio.h>
#include <signal.h>

int i;
int *p;

void handler(int sig_num) {
    printf("handler: %d, %d\n", p==NULL, i);
    p = &i;
    signal(SIGSEGV, handler);
}

main() {
    p = NULL;
    signal(SIGSEGV, handler);
    *p = 1;
    printf("main: %d, %d\n", p==NULL, i);
}
```

6. 20 points

In this problem, we implement the Unix command:

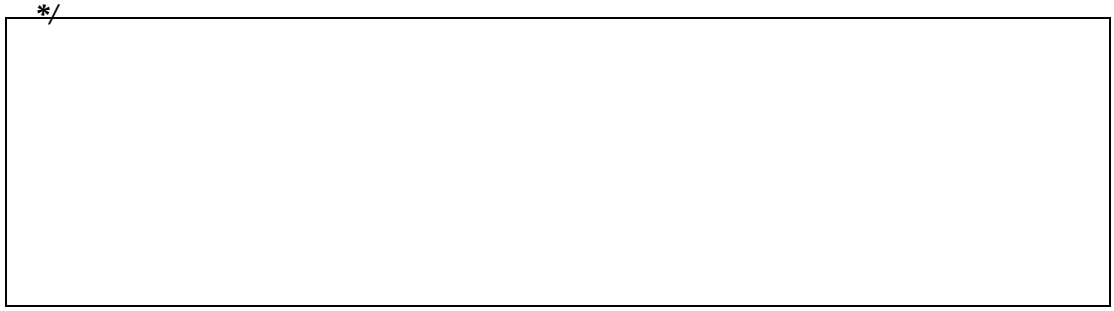
```
/bin/sort < input | /usr/bin/uniq | /usr/bin/wc > output
```

This command counts the number of unique lines in the input file and sends the result to an output file. You're given the following partial code. Your task is to fill the four smaller boxes with code to complete the program. Note that the lines of code that you need to write contain ***only*** `fork()`, `open()`, `close()`, and `runCommand()` calls---nothing else is needed. Omit all error checking.

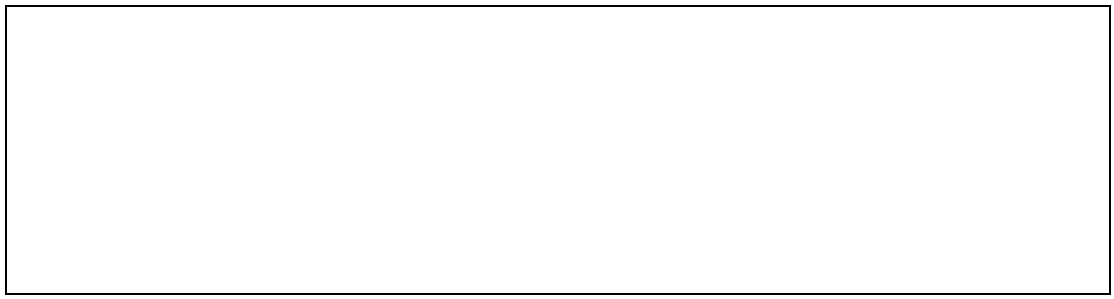
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

void runCommand(char *path, char *commandName,
                int input, int output) {
    dup2(input, 0);
    dup2(output, 1);
    close(input);
    close(output);
    execl(path, commandName, NULL);
}

int main() {
    int p[2], q[2], status, input, output;

    /* create the first pipe. */
    pipe(p);

    /*
     * run "/bin/sort" in the first child process.
     * close the un-used end of the pipe.
     * redirect stdin to the input file called "input".
     * redirect stdout to the output-end of the pipe p.
     */
```

(Code continues on the next page.)

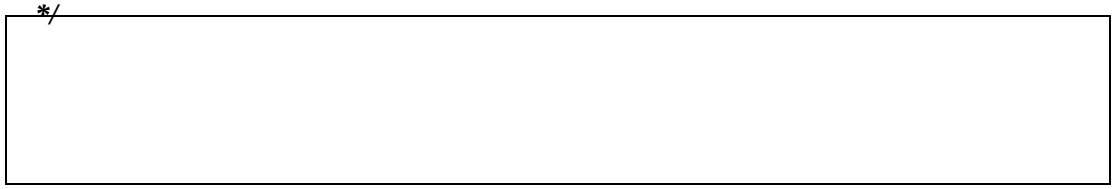
```
    /* create the second pipe. */
    pipe(q);

    /*
     * run "/usr/bin/uniq" in the second child process.
     * close un-used ends of pipes.
     * redirect stdin to the input-end of the pipe  p.
     * redirect stdout to the output-end of the pipe  q.
     */
```



```
    /*
     * run "/usr/bin/wc"  in the third child process.
     * close un-used ends of pipes.
     * redirect stdin to the input-end of the pipe  q.
     * redirect stdout to the output file called  "output".
     */
```



```
    /*
     * the parent process.
     * close all unused ends of pipes.
     * wait for all three children to terminate.
     */
```



```
    wait(&status);
    wait(&status);
    wait(&status);
}
```

7. 20 points

In this problem, we translate the following C fragment into an IA 32 assembly language code fragment:

```
include <stdio.h>

#define N 256
#define M 32

struct S {
    int id;
    int data[2];
};

int a[N];
struct S s;
struct S ss[M];

_start() {
    struct S *p;
    p = &ss[13];
    f(a, s, ss, p->data, &(p->id));
    exit(0);
}
```

We assume the procedure `f()` is defined somewhere else and we do not concern ourselves with its implementation in this problem. On the next page, complete the assembly language version of the above code. Be sure to use the standard C calling sequence. You only need to handle the calling of function `f()` and the subsequent cleanup. You do NOT need to worry about the calling or cleaning up of function `_start()`. Comment your assembly code to describe at least which procedure argument you're dealing with. In this problem, we assume that a C `int` (integer) data item takes 4 bytes, and there is never extraneous space between two consecutive integer variables stored in memory.

```
# 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:









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

8. 25 points

You're given the following interface of a symbol table ADT in a **symtable.h** file. (This is the same interface as the one discussed in lectures.)

```
typedef struct SymTable *SymTable_T;

/* create a new, empty table. */
SymTable_T SymTable_new(void);

/* enter (key,value) binding in the table; else return 0 if already there */
int SymTable_put(SymTable_T table, char *key, void *value);

/* look up key in the table, return value (if present) or else NULL */
void *SymTable_get(SymTable_T table, char *key);

/* apply f() to every binding in the table ... */
void SymTable_map(SymTable_T table,
   void (*f)(char *key, void *value, void *extra),
   void *extra);

/* Return the number of bindings in table.
It is a checked runtime error for table to be NULL. */
int SymTable_getLength(SymTable_T table);

/* Remove from table the binding whose key is key. Return 1 if successful, 0 otherwise.
  It is a checked runtime error for table or key to be NULL. */
int SymTable_remove(SymTable_T table, char *key);
```

You are to add a new function to the symbol table interface:

```
int SymTable_same(SymTable_T t1, SymTable_T t2, ...)
```

This function takes two symbol tables as inputs and returns `1` if the two tables are identical, or else `0`. The two symbol tables are considered identical iff they contain identical bindings. Two bindings are considered identical if they have the same keys and the same values. Two values do not necessarily need to share the same memory storage location to be considered "same."

(a) Give the complete prototype (interface) of the `SymTable_same` function. (Hint: you may need extra arguments in addition to the two symbol tables.)

(b) Give an implementation of the `SymTable_same` function. Note that you may ***not*** assume knowledge of how the existing symbol table ADT is implemented (in terms of, for example, whether it is implemented as a linked list or a hash table). (A correct answer could take about 30 lines of code, although we won't penalize you if you use a few more.)