Banner number: Name:

Final Exam

CSCI 2132: Software Development

April 18, 2019

Marks	
Question 1 (10)	
Question 2 (10)	
Question 3 (15)	
Question 4 (10)	
Question 5 (10)	
Question 6 (10)	
Question 7 (10)	
Question 8 (10)	
Question 9 (15)	
Question 10 (10)	
Question 11 (10)	
Total (120)	

Instructions:

- **Write your banner number and name at the top of this page *before continuing to read.***
- **This exam has 14 pages, including this title page. Notify me immediately if your copy has fewer than 14 pages.**
- You are allowed to use **two cheat sheets, letter paper, both sides, 10pt font or larger.**
- Understanding the exam questions is part of the exam. Therefore, **questions will *not* be interpreted.** Proctors will only correct errors in question statements and clarify ambiguities, if any.

Question 1 (Programs, processes, and Unix's system design) **10 marks**

(a) Put a checkmark beside each statement that is correct (and none beside each statement that is not).

A process can have a different effective user ID than the real user ID of the user that started the process.	✓
A program is executable code stored in a file.	✓
The relationships between parent and child processes form a tree.	✓
Every process on a Unix system is started from a shell.	
A Unix system can run many processes at the same time.	✓
A Unix process can have only one thread of execution.	
Once a process has been started in the background, it cannot be moved to the foreground.	
Only foreground processes can read keyboard input.	✓

(b) Put a checkmark beside each statement that is correct (and none beside each statement that is not).

In Unix, application programs can interact directly with peripheral devices such as keyboards, cameras or printers.	
Application programs interact with the operating system through system calls.	✓

(c) List two advantages of Unix's onion skin model:

Application programs do not have to deal with the details of different pieces of hardware they want to interact with. The OS provides appropriate device drivers and provides a uniform interface to different pieces of hardware in the form of system calls.

There is a security advantage. Since application programs interact with the OS and hardware through a well-defined set of system calls, the OS can control the operations different application programs are allowed to perform and prevent misbehaving application programs from destabilizing the system.

Question 2 (Files, inodes, and permissions) **10 marks**

(a) Put a checkmark beside each statement that is correct (and none beside each statement that is not).

A file is a stream of bytes.	✓
Reading keyboard input in Unix requires different system calls than reading from a file stored on disk.	
Symbolic links and sockets are types of files in Unix.	✓
A file (as identified by its inode) can exist in multiple directories.	✓
A file and a symbolic link to the file occupy the same inode in Unix.	
A file and a hard link to it may have different permissions.	
The command `echo "Doodledeedoo" >> file.txt` replaces the content of `file.txt` with `Doodledeedoo`.	
The command `prog; echo $?` prints the exit code of `prog`.	✓
Whether a process can access a given file is determined by the owner, group, and permissions of the file and by the effective user ID and the effective group ID of the process.	✓

(b) Choose the correct command line to redirect *both* the `stdout` and `stderr` of the program `prog` to the file `log.txt`.

`prog > log.txt 2> log.txt`	
`prog > log.txt 2>&1`	✓
`prog 2>&1 > log.txt`	

(c) Provide the command you need to use to allow the owner of a file to read, write, and execute the program `prog`. Members of the file's group should be allowed to execute the program but not read or write it. Anybody else is not allowed to access the file in any way.

Use symbolic notation for the permissions:

```
chmod u+rwx,g=x,o= prog
```

Use octal permissions:

```
chmod 710 prog
```

Question 3 (Basic Unix commands and regular expressions) **15 marks**

(a) List 4 standard Unix commands and explain what they do.

`ls` *can be used to list files in a directory and display information about them.*

`cat` *displays the contents of a file.*

`grep` *can be used to display all lines in a file that match a given pattern.*

`chown` *can be used to change the owner of a file.*

(b) Use a single Unix command line to list all C files in the current directory whose names start with `a` or `b` and do not end with `x`, `y` or `z` (excluding the standard extension `.c` for C files).

```
ls [ab]*[^xyz].c
```

(c) Write a one-line command that allows you to list all files in `/usr/bin` that can be written by someone who is not the owner of the file. (These are a serious security concern.)

```
ls -l /usr/bin | grep -e '....\(w...\|...w\). '
```

(d) Write a one-line command that lists all words in `/usr/share/dict/linux.words` that contain at least 11 vowels.

```
egrep -e '([aeiou].*){11}' /usr/share/dict/linux.words
```

(e) Consider a text file `games.txt` that contains a list of hockey games. Each line has the format "`team1:team2=score1:score2`" (example: `Jets:Sabres=3:2`). If the first team in each game is the home team, and the second team is the visiting team, write a single Unix command line to print out all visiting teams, sorted, without duplicates.

```
cut -d= -f1 games.txt | cut -d: -f2 | sort | uniq
```

Question 4 (Shell scripting) **10 marks**

Write a shell script that takes three directories as argument. Let us call them *src*, *dst*, and *ref* here. The script may assume that there are only regular files in *src*, no directories, no symbolic links nor any other kind of special files. It may also assume that *dst* exists and is empty. *ref* exists and may contain any number of regular files. The script should do the following:

- It inspects every file *f* in *src* and checks whether *ref* contains a file with the same name.
- If no file *f* exists in *ref*, then the file *f* is copied from *src* to *dst*.
- If a file *f* exists in *ref*, then the two files with name *f* in *src* and *ref* are compared.
- If the two files have different contents, the file *f* in *src* is copied to *dst*; otherwise, no action is taken.

You can think about this as a backup script. *src* is the directory you want to back up to *dst*. *ref* is a directory containing an earlier backup. You want to backup only files that are either new or have changed since the backup stored in *ref*.

```
src=$1
dst=$2
ref=$3
for f in `ls $src`; do
    if [ -f $ref/$f ]; then
        diff $src/$f $ref/$f > /dev/null 2>&1
        if [ $? -ne 0 ]; then
            cp $src/$f $dst/$f
        fi
    else
        cp $src/$f $dst/$f
    fi
done
```

Question 5 (Data types and variables) **10 marks**

(a) Assume the C `int` type uses 4 bytes, `char` uses 1 byte, and `unsigned long` uses 8 bytes. What is the size of the following type (as reported by `sizeof`)?

```
struct {
    unsigned long ul;
    int           i;
    char          c;
};
```

☐ 13 bytes ☑ 16 bytes ☐ 8 bytes ☐ 24 bytes

(b) Under the same assumptions as in (a), what is the size of the following type (as reported by `sizeof`)?

```
union {
    unsigned long ul;
    int           i;
    char          c;
};
```

☐ 13 bytes ☐ 16 bytes ☑ 8 bytes ☐ 24 bytes

(c) List the names of the three memory regions where a C program can store data.

For each of these three regions, explain what type of statement in your C code leads to the allocation of space in this region.

↓ **Name**	**Explanation** →
Stack	*Any variable declared in a function is stored in that function's stack frame and the compiler reserves space for this variable on the stack, as part of the stack frame.*
↓ **Name**	**Explanation** →
Heap	*Heap space needs to be allocated explicitly by calling* `malloc`*. (It also needs to be released explicitly by calling* `free`*.)*
↓ **Name**	**Explanation** →
The `DATA` *segment of the program (static address space)*	*Any variable declared outside a function or marked as* `static` *inside a function is stored here.*

Question 6 (Arrays, pointers, and memory management) **10 marks**

(a) In C99, which of the following function definitions are legal?

```
void f(int m, int n, int a[m][n]) { ... }
```
☑

```
void f(int a[m][n], int m, int n) { ... }
```
☐

```
void f(int m, int n, int *a) { ... }
```
☑

```
void f(int m, int n, int a[][n]) { ... }
```
☑

(b) What does the following code print?

```
int main() {
    int a[10] = {1, 1, 1}; int *p, *q;
    for (p = a + 1, q = a + 8; p < q; ++p) {
        *(p + 1) += *p;
        *(p + 2) += *p;
    }
    *(p + 1) += *p; q += 2;
    for (p = a; p < q; ++p) {
        printf("%d ", *p);
    }
    printf("\n");
    return 0;
}
```

```
1 1 2 3 5 8 13 21 34 55   (the first 10 Fibonacci numbers)
```

(c) Rewrite the following code, which uses variable-length arrays to deal with inputs of different sizes, so that it allocates space for the array on the heap instead and releases the allocated memory when it is done. Fill in the appropriate function calls in the lines marked with ">".

```
int main() {
    int n;
    scanf("%d", &n);
    float nums[n];
    for (int i = 0; i < n; ++i) {
        scanf("%f", nums + i);
    }
    // The stddev function is provided
    float d = stddev(n, nums);
    printf("%f\n", d);
    return 0;
}
```

```
int main() {
    int n;
    scanf("%d", &n);
>   float *nums =
>       malloc(n * sizeof(float));
    for (int i = 0; i < n; ++i) {
        scanf("%f", nums + i);
    }
    float d = stddev(n, nums);
    printf("%f\n", d);
>   free(nums);
>
    return 0;
}
```

Question 7 (Input and output in C) **10 marks**

(a) Provide appropriate `printf` statements to print

The contents of a `char` variable `c`:

```
printf("%c", c);
```

The contents of an `int` variable `i`, padded to 10 characters, with a sign (even if the number is positive).

```
printf("%+10d", i);
```

The contents of a `double` variable `d`:

```
printf("%lf", d);
```

(b) Assume `stream` is a `FILE *` pointing to a file that has already been opened. Provide the correct statement to read an arithmetic expression of the form "*a op b*" from `stream`. You should store the first operand in a `double` variable `a`, the second operand in a `double` variable `b`, and the operator in a character variable `op`. The statement should allow arbitrary spacing around the operands and operator.

```
fscanf(stream, "%lf %c %lf", &a, &op, &b);
```

(c) Assume `buf` is a `char` array of size 128. Provide a single `scanf` statement to read a string of non-whitespace characters into `buf` but make sure that you prevent a buffer overflow in case the user provides more than 128 characters of input.

```
scanf("%127s", buf);
```

(d) Fill in the following code so it reads a 64-bit offset from the beginning of a binary file with name `db.bin`, then reads the 10 characters stored starting at the read offset in the file, closes the file, and prints the read characters to `stdout`.

```
#include <stdio.h>

int main() {
    FILE *db; unsigned long offset; char chars[11] = {};
    db = fopen("db.bin", "r");
    fread(&offset, sizeof(unsigned long), 1, db);
    fseek(db, offset, SEEK_SET);
    fread(chars, sizeof(char), 10, db);
    fclose(db);
    printf("%s\n", chars);
    return 0;
}
```

(Extra space for Question 7 if needed)

Question 8 (Program organization and compilation) **10 marks**

(a) List three benefits of splitting a larger C project (or in fact any large project written in any compiled language) into multiple source files:

Smaller source files are easier to navigate and manipulate in an editor.

Different developers can work on different source files and are less likely to "step on each other's toes", that is, make incompatible changes to the same part of the code.

Using a build tool such a `make`*,* `cargo` *(Rust) or* `cabal` *(Haskell), it is possible to recompile only the part of the code base that has been built since the last time the project was compiled. This can shorten compilation times significantly.*

(b) List and briefly explain the three phases of translation that a C program undergoes in order to turn it into an executable program.

1. *The **preprocessing phase** processes preprocessing directives and* textually transforms *the source code. For example, each* `#include` *statement is replaced with the contents of the file that is included and macro uses are replaced with the macro text introduced by an earlier* `#define` *statement.*

2. *The **compilation phase** translates each preprocessed source file into an object file. This is essentially executable machine code but may include references to symbols defined in other object files.*

3. *The **linking phase** resolves references to symbols defined in other object files and combines the different object files into a final executable file.*

Question 9 (Testing and debugging) **15 marks**

(a) Explain the difference between black-box testing and white-box testing.

Black-box testing treats a program or program component as a black box, without regard for its implementation details. It tests that the program or program component meets its specifications. The goal is to ensure that all requirements are met.

White-box testing is driven by knowledge of the implementation details of a program. The goal is to design tests that exercise every line of code and test that it behaves as intended.

(b) Explain the relationship between testing and debugging.

Testing is used to test whether the program meets all its requirements, that is, it is a first step to discover bugs. Debugging is the process of pinpointing the piece of code that causes the erroneous behaviour and then fix this piece of code in order to eliminate the bug.

(c) List 4 `gdb` commands and explain what they do.

`run`*, possibly with command line arguments, allows us to run a program that has been loaded into* `gdb`.

`break`*, with a line number or function name as argument, sets a breakpoints.*

`continue` *continues the execution after the program was stopped at a breakpoint.*

`display` *marks a variable or more complicated expression to be displayed automatically every time the program stops at a breakpoint.*

Question 10 (Makefiles) **10 marks**

Consider a program `studentdb` to manage a database of students. The main source file is `main.c`. Basic functions for manipulating student records are implemented in `student.c` and declared in `student.h` to make them available to other source files. To store the student records, the program uses a linked list implemented in `linked_list.c`. The linked list type and functions to manipulate linked lists are declared in `linked_list.h` to make them available to other source files. Finally, the linked list uses a memory manager declared in `memmgr.h` and implemented in `memmgr.c` to manage the memory occupied by its list nodes more efficiently than allocating and deallocating them using individual calls to `malloc` and `free`. This gives the following file skeletons:

`main.c`

```
#include "student.h"
#include "linked_list.h"

...
```

`student.c`

```
#include "student.h"

...
```

`linked_list.c`

```
#include "linked_list.h"
#include "memmgr.h"

...
```

`memmgr.c`

```
#include "memmgr.h"

...
```

`student.h`

```
...
```

`linked_list.h`

```
...
```

`memmgr.h`

```
...
```

Write a `Makefile` that ensures that

- `make studentdb` builds an up-to-date version of `studentdb` that reflects the current state of all source files and
- `make studentdb` only compiles and links files that need to be recompiled or linked based on the files that have changed since the last time `studentdb` was built.

```
studentdb: main.o student.o linked_list.o memmgr.o
	gcc -o $@ $^

main.o: main.c student.h linked_list.h
	gcc -c $<

student.o: student.c student.h
	gcc -c $<

linked_list.o: linked_list.c linked_list.h memmgr.h
	gcc -c $<

memmgr.o: memmgr.c memmgr.h
	gcc -c $<
```

Question 11 (C programming) **10 marks**

Two strings are anagrams of each other if they contain the same letters; each character has to occur the same number of times but they do not have to occur in the same order. For example, "smartest" and "mattress" are anagrams because they both contain one letter "a", one letter "e", one letter "m", one letter "r", two letters "s", and two letters "t".

Write a C program that reads two lines of input from `stdin`. If these two lines are anagrams, the program should print "`... and ... are anagrams`" to `stdout`, where the `...` are to be replaced with the two input lines. Otherwise, it should print "`... and ... are not anagrams`" to `stdout`. Again, the `...` are to be replaced with the two input lines. Your program must be able to handle any input size and must not leak memory. You do not need to perform any error checks.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int charcmp(const void *a, const void *b) {
    return *((char *) a) - *((char *) b);
}

int main() {
    char *a = NULL; char *b = NULL;
    size_t ca = 0;  size_t cb = 0;
    int na, nb;
    na = getline(&a, &ca, stdin);
    nb = getline(&b, &cb, stdin);
    a[--na] = 0;
    b[--nb] = 0;
    char tmpa[na + 1], tmpb[nb + 1];
    strcpy(tmpa, a);
    strcpy(tmpb, b);
    qsort(tmpa, na, 1, charcmp);
    qsort(tmpb, nb, 1, charcmp);
    if (strcmp(tmpa, tmpb) == 0) {
      printf("%s and %s are anagrams\n", a, b);
    } else {
      printf("%s and %s are not anagrams\n", a, b);
    }
    free(b);
    free(a);
    return 0;
}
```

(Extra space for Question 11 if needed)