McGill

April 2019
Final Examination

INTRODUCTION TO SOFTWARE SYSTEMS

COMP 206

TUESDAY, APRIL 30, 2019 - 14:00 to 17:00

EXAMINER: Joseph Vybihal ASSOC. EXAMINER: Chad Zammar

STUDENT NAME:		**McGILL ID:**									

INSTRUCTIONS:

EXAM:	CLOSED BOOK ☒ OPEN BOOK ☐
	SINGLE-SIDED ☐ PRINTED ON BOTH SIDES OF THE PAGE ☒
	MULTIPLE CHOICE ANSWER SHEETS ☐ E: The Examination Security Monitor Program detects pairs of students with unusually similar answer patterns on multiple-choice exams. Data generated by this program can be used as admissible evidence, either to initiate or corroborate an investigation or a charge of cheating under Section 16 of the Code of Student Conduct and Disciplinary Procedures. ANSWER IN BOOKLET ☒ EXTRA BOOKLETS PERMITTED: YES ☒ NO ☐ ANSWER ON EXAM ☐
	SHOULD THE EXAM BE: RETURNED ☒ KEPT BY STUDENT ☐
CRIB SHEETS:	NOT PERMITTED ☐ PERMITTED ☒ e.g. one 8 1/2X11 handwritten double-sided sheet Specifications:
DICTIONARIES:	TRANSLATION ONLY ☒ REGULAR ☐ NONE ☐
CALCULATORS:	NOT PERMITTED ☐ PERMITTED (Non-Programmable) ☒
ANY SPECIAL INSTRUCTIONS: e.g. molecular models	

This exam has 10 questions: (assumes 18 minutes per question on average)

Question 1: C language - recursive functions

Write a **recursive** function called counter_recursive having the following signature:
void counter_recursive (int number);

The function will simply print to the screen all the numbers from 1 through *number*. For example, if number is equal to 5, the function will print: 1 2 3 4 5

If number is less than 1, nothing will be printed, and the function should exit without doing anything.

Solution Sample:

```
void counter_recursive (int number)
{
        if (number<1) return;
        counter_recursive(number-1);
        printf("%d ", number);
}
```

Grading:
3 points for checking if number > 1
5 points for making the function recursive
2 points for calling the function before printing [but it depends on the implementation]

Question 2: C language - general programming

Write a C program called `calculate` that performs simple arithmetic operations. The program takes 3 or 5 arguments from the command line, as shown in the following use case examples:

- **calculate 5 * 4** will print 20
- **calculate 4 + 2 * 8** will print 20
    - *Remember that * and / have higher precedence than + and -.*
- **calculate 6 / 3 - 2** will print 0

Calculate supports the following operations: +, -, / and *. Do not check the input for validity, except the number of arguments: 3 or 5. Assume all the input values are integer.

Feel free to use helper functions if it makes your task easier.

The following is a pseudocode solution:

```
Int calc(int x, char op, int y) {
        If (op==’+’) return (x + y);
        If (op==’-‘) return (x – y);
        If (op==’*’) return (x * y);
        If (op==’/’) return (x / y);
}

Int main(int argc, char *argv[]) {
        Int a, b, c, result, doOp;
        Char op1, op2;

        If (argc == 4 || argc == 6) {
                A = atoi(argv[1]);
                B = atoi(arbv[3];
                Op1 = *argv[2];
        }
        if (argc == 6) {
                c = atoi(argv[5]);
                op2 = *argv[4];
        } else exit(1); // wrong number of arguments

        If (op1==’+’ || opt1==’-‘) && (op2==’*’ || op2==’/’) doOp = 2;
        Else doOp = 1;

        If (doOp == 1) result = calc(calc(a,op1,b), op2, c);
        Else result = calc(a,op1,calc(b,op2,c));

        Return 0;
}
```

Question 3: C language - cloning and environment variables

Write a C program that sets an environment variable called **done** to 0. The program will then ask the user to enter 5 numbers in a loop and store the numbers in an array of 5 int elements.

The program will then creates a clone process (using fork()). The clone should run a function called **sum()** that calculates the total sum of all the numbers in the array. After that the clone will set the **done** environment variable to 1. It will also create a second environment variable called **result** and will set it to the return value of the **sum()** function.

The parent process wait on **done** variable until it is set to 1. If it is 1, the parent will print to the screen : The sum of elements is: *result.* Note that *result* is the content of the environment variable called **result**.

Please note that you need to choose a suitable signature for your **sum()** function, meaning that it is up to you to decide about the type and the number of arguments as well as the type of the return value.

Example pseudo code solution:

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

int sum(int a[], int size) {
        int index, result=0;

        for(index=0; index<size; index++) result += a[index];
        return result;
}

Int main() {
        int i, array[5], result;
        long pid;
        char string[20];

        for(i=0;i<5;i++) scanf("%d",&array[i]);

        setenv("done","0",1);

        pid = fork();

        If (pid == 0) { //child
                Result = sum(array,5);
                Sprintf(string,"%d", result);
                Setenv("result", string, 1);
                Setenv("done","1", 1);
        } else { //parent
                While(strcmp(getenv("done"),"0")==0); // busy wait
                Result = atoi(getenv("done"));
                Printf("The sum of elements is: %d\n", result);
        }

        Return 0;
}
```

Question 4: BASH scripting

Write a bash script that achieves the following:

a. Takes 2 arguments that represent a file name and a word. ***// 1 point***
b. Check if arguments are exactly 2, if not exit with error code 1. ***// 1 point***
c. Check if the file exists in the local directory, if not exit with error code 2. ***// 2 points***
d. If the file exists, search if the file has the word provided previously (the second argument provided to the script). This means that you need to search the content of the file for the occurence of the word. ***// 2 points***
e. If the word exists in the file, print the message: the file *fileName* contains the word *word.* ***// 2 points***
f. If not print a message *word* not found in file *fileName*". ***// 1 point***
g. Exit with error code zero. ***// 1 point***

Question 5: Theory - programming vs scripting

How is C programming different from BASH scripting? When would you use one over the other and why?

bash ------------VS--------- C
interpreted ------------ compiled
slower ------------------ faster
Unix only -------------- more portable
limited ------------------ more powerful

Bash is used to automate session activities and customize the user's session. The C language is used to interact with the computer system in a fast and direct way.

Question 6: Debugging

You are given the following name.c file:

```
#include <stdio.h>

char* get_name()
{
    char name[100];
    printf("Please enter a name: ");
    scanf("%s", name);
```

```
        return name;
    }

    int main()
    {
      char* name = get_name();
      printf("Your name is %s\n", name);
      return 0;
    }
```

You compile it using this gcc command: **gcc name.c** and you run the generated executable. The program will ask you to input a name, which you do. Surprisingly, the result that is printed out to the screen is not what you were expecting. You suspect that there might be a bug in the program, so you decide to debug it using gdb. Please answer the following:

a. Can you run your program with gdb right away, or do you need to recompile and why?
   3 points. [No, we need to recompile by adding -g gdb flags to include source code and symbol table]
b. What are the necessary gdb commands that you will be using in order to find the bug? Please describe the full process in detail.
   4 points for describing the process of debugging [like using backtrace, placing a breakpoint, stepping with next etc.]
c. What do you think the bug is?
   3 points. [get_name() is returning a local variable that is destroyed by the time it is being used by main]

Question 7: Profiling

You run a C program and you are not satisfied with the execution time. You run your program again using gprof in order to better understand which part of your code is taking longer to execute. The following is the results that gprof printed out to the screen. First the flat profile:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  0.00      0.00     0.00        1     0.00     0.00  main
 96.53      0.44     0.44       10    44.40    44.40  evaluate
  3.47      0.46     0.02        1     2.18    46.22  fibonacci
  0.00      0.46     0.00  1000000     0.00     0.00  resolve
```

Then the call graph:

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]             0.00    0.46        1        main [1]
                0.02    0.44       1/1           fibonacci [2]
-----------------------------------------------
                0.00    0.46       1/1           main [1]
[2]    100.0    0.02    0.44        1        fibonacci [2]
                0.44    0.00      10/10          evaluate [3]
                0.00    0.00 1000000/1000000     resolve [4]
-----------------------------------------------
                0.02    0.44      1/1            fibonacci [1]
[3]     95.7    0.44    0.00      10         evaluate [3]
-----------------------------------------------
                0.02    0.44       1/1           fibonacci [1]
[4]      0.0    0.00    0.00 1000000         resolve [4]
-----------------------------------------------
```

Please answer the following questions and justify:

a. Which function does main() call?

2.5 points. [fibonacci is being called directly by main, see index [2] nested under [1]]

b. Which function does resolve() call?

2.5 points. [None, ther are no index number nested under [4]]

c. Which function is taking the longest time to execute?

2.5 points. [evaluate, .44/10 = 0.04, which is larger than fibonacci's 0.02]

d. How many times is the fibonacci() function called?

2.5 points. [one time, Fibonacci is marked with 1/1, one call from total of one call]

Question 8: Makefile

Assume a program composed of 3 modules: **mod1.c, mod2.c** and **mod3.c**. You need to create a makefile to help compile these modules in different ways. The makefile must support the following:

a. when the user types **make**, all by itself, all 3 modules are compiled into an executable called **exe_release** is generated, with no special options.It is just the regular module execution file.

2.5 points

b. when the user types **make debug**, all 3 modules are compiled in debug mode and an executable called **exe_debug** is generated

2.5 points

c. when the user types **make profile**, all 3 modules are compiled in profiler mode (meaning that gprof flags are enabled) and an executable called **exe_profile** is generated.
2.5 points

d. when the user types **make clean**, all previously generated executables and all object files will be removed.
2.5 points

Sample Makefile

```
all: mod1.o mod2.o mod3.o
gcc -o exe_release mod1.o mod2.o mod3.o

debug: mod1.o mod2.o mod3.o
gcc -d -o exe_debug mod1.o mod2.o mod3.o

profile: mod1.o mod2.o mod3.o
gcc -pg -o exe_profile mod1.o mod2.o mod3.o

clean:
rm -rf *.o; rm -rf exe_*

mod1.o: mod1.c
gcc -c mod1.c

mod2.o: mod2.c
gcc -c mod2.c

mod3.o: mod3.c
gcc -c mod3.c
```

Question 9: Interprocess communication

List and explain all the different methods that will enable two different running programs to communicate. When would you use one over the other and why?

- **using files ………………………………. when processes are not in same computer**
- **using environment variables ………. when they share the same shell**
- **shared memory ………………………. When need to modify the same structures**
- **socket ………………………………….. when not on the same network**

Question 10: Version control system

A software company decides to use git as a versioning control system. Please answer the following:

1. How is git different than creating manual backups and why is it better?

2.5 points for explaining the advantages of using git, like:

- **easier access to change history (backup only keeps a single backup instance)**
- **better suited for team work (everyone backs up to the same place)**
- **easier to maintain (permissions, moveable)**
- **easier to know who modified what and when (tracks programmers)**

2. What are the git commands that you need to use to do:

   1.5 points each for providing the appropriate commands:

   a. create a new repository called MyAwesomeProject

      mkdir MyAwesomeProject
      cd MyAwesomeProject
      git init

   b. stage 2 c files to the repo: file1.c and file2.c

      git add file1.c file2.c

   c. adding a useful message describing what changes we made in the file

      git commit -m "initial project files"

   d. create a new branch called *feature1*

      git branch feature1

   e. navigate to the *feature1* branch

      git checkout feature1