

CSC 209H5 S 2017 Midterm Test

Duration — 50 minutes

Aids allowed: none

Student Number:

Last Name: First Name:

Instructors: Chaturvedi, Petersen

*Do **not** turn this page until you have received the signal to start.*

(Please fill out the identification section above, **write your name on the back of the test**, and read the instructions below.)

Good Luck!

This midterm consists of 4 questions on 8 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.*

Comments are not required, although they may help us mark your answers.

No error checking is required **except in question 4**.

You do not need to provide include statements for your programs.

If you use any space for rough work, indicate clearly what you want marked.

1: / 3

2: / 5

3: / 4

4: / 3

TOTAL: / 15

Question 1. [3 MARKS]

Makefiles and the Shell Assume you have a shell window open, and the current working directory contains a Makefile, the C source files `foo.c` and `bar.c`, and the header file `proto.h`. Here is the contents of the Makefile:

```

FLAGS = -Wall -std=c99
DEPENDENCIES = proto.h

all: prog

test: prog
    ./prog foobar > prog_output

prog: foo.o bar.o
    gcc ${FLAGS} -o $@ $^

%.o: %.c ${DEPENDENCIES}
    gcc ${FLAGS} -c $<

clean:
    rm -f *.o prog prog_output

```

Part (a) [1 MARK]

If the command “`make`” were executed in the shell, what files would you expect to be created?

Part (b) [1 MARK]

Assuming that the executable “`prog`” had been successfully created in the past, if you were to edit the file “`foo.c`” and then run the command “`make prog`”, what files would be created or updated?

Part (c) [1 MARK]

You’ve tested your code and are ready to submit. Assuming that the current directory is part of a *git* repository, what commands would you run on the shell to submit the source files for marking? (Please use the same process as for A1 and A2 and assume that you have not previously submitted source files.)

Question 2. [5 MARKS]

Files and I/O

Part (a) [1 MARK]

Write code to read an integer and a string of length no more than 50, in that order, from standard input.

Part (b) [2 MARKS]

Write code to open the file named “`data.txt`” from the current working directory. Then, read an integer from offset 512 in the file.

Part (c) [2 MARKS]

The current working directory contains a directory “`dir_a`”. That directory contains a file named “`readme.txt`”. Write a snippet of code that places that file’s permissions into an integer variable.

Question 3. [4 MARKS]

Structs and Pointers Each of the subquestions below asks for a short snippet of code. They all have the following `Employee` struct available:

```
struct Employee {
    char *name;
    int eid;                // Employee ID

    struct Employee *reports; // A linked list of employees who report to this person
    struct Employee *next;    // To implement the linked list of reports
};
```

This structure should be familiar: like the `TreeNode` from A2, it creates a tree of employees.

Part (a) [1 MARK]

Create and initialize a `struct Employee` for the person “Andrew Petersen” with employee ID 13. This person has no reports and does not report to anyone. Use a string literal for the name.

Part (b) [1 MARK]

Dynamically allocate an `Employee` and assign it to `emp_ptr`. Then, initialize it by setting the name to a **copy** of `name` and the employee ID to 42. This person has no reports and does not report to anyone.

```
struct Employee *emp_ptr;
char *name = ...           // Initialized to a mystery value
```

Part (c) [1 MARK]

Assume that you initialized the **Employee** struct correctly. **free** all of the memory that was allocated, and remove the dangling pointer.

```
struct Employee *emp_ptr;  
.....                // emp_ptr initialized correctly
```

Part (d) [1 MARK]

Write a recursive function that implements a post-order traversal of a tree of **Employees**. The function should print the name of each employee on its own line. Unlike A2, there is no need to indent the names. The linked list of **reports** is implemented in a straightforward manner, with no dummy head or tail nodes.

```
void print_employees(struct Employee *emp_root) {
```

Question 4. [3 MARKS]

Processes and Memory Write a program that prints the total number of odd-length arguments provided on the command line. (The executable name should not be included in this count.) The program should **fork** one child for each argument. The child should exit with a 0 if its command line argument has even length and a 1 if its command line argument has an odd length. The parent process should print the number of odd-length arguments using the return codes of its children.

Please make sure to include all necessary error checks in this code and to report errors appropriately.

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.] /

C function prototypes:

```

pid_t fork(void);
int fclose(FILE *stream)
FILE *fopen(const char *file, const char *mode)
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
void free(void *ptr)
int fscanf(FILE *restrict stream, const char *restrict format, ...);
int fseek(FILE *stream, long offset, int whence)
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int lstat(const char *restrict path, struct stat *restrict buf);
void *malloc(size_t size);
void perror(const char *s)
int scanf(const char *restrict format, ...);
size_t strlen(const char *s)
int strncmp(const char *s1, const char *s2, size_t n)
char *strncpy(char *dest, const char *src, size_t n)
char *strstr(const char *haystack, const char *needle)
pid_t wait(int *stat_loc);

```

Excerpt from the fseek man page:

If whence is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

Excerpt from lstat man page:

All of these system calls return a stat structure, which contains the following fields:

```

struct stat {
    dev_t    st_dev;    /* ID of device containing file */
    ino_t     st_ino;    /* inode number */
    mode_t    st_mode;   /* protection */
    nlink_t   st_nlink;  /* number of hard links */
    uid_t     st_uid;    /* user ID of owner */
    gid_t     st_gid;    /* group ID of owner */
    dev_t     st_rdev;   /* device ID (if special file) */
    off_t     st_size;   /* total size, in bytes */
    ...

```

Excerpt from the wait man page:

WIFEXITED(status)

True if the process terminated normally by a call to _exit(2) or exit(3).

WEXITSTATUS(status)

If WIFEXITED(status) is true, evaluates to the low-order 8 bits of the argument passed to _exit(2) or exit(3) by the child.

Print your name in this box.