

Last Name: \_\_\_\_\_ First Name: \_\_\_\_\_

Student #: \_\_\_\_\_ Signature: \_\_\_\_\_

**UNIVERSITY OF TORONTO MISSISSAUGA**  
**APRIL 2019 FINAL EXAMINATION**  
 CSC209H5S  
 Software Tools and Systems Programming  
 Furkan Alaca & Paul Vrbik  
 Duration - 3 hours  
 Aids: none

*The University of Toronto Mississauga and you, as a student, share a commitment to academic integrity. You are reminded that you may be charged with an academic offence for possessing any unauthorized aids during the writing of an exam. Clear, sealable, plastic bags have been provided for all electronic devices with storage, including but not limited to: cell phones, smart devices, tablets, laptops, calculators, and MP3 players. Please turn off all devices, seal them in the bag provided, and place the bag under your desk for the duration of the examination. You will not be able to touch the bag or its contents until the exam is over.*

*If, during an exam, any of these items are found on your person or in the area of your desk other than in the clear, sealable, plastic bag, you may be charged with an academic offence. A typical penalty for an academic offence may cause you to fail the course.*

*Please note, once this exam has begun, you **CANNOT** re-write it.*

*You must earn 40% or above on the exam to pass the course; else, your final course mark will be set no higher than 47%.*

### MARKING GUIDE

This final examination consists of 6 questions on 24 pages (including this one). When you receive the signal to start, please make sure that your copy of the examination is complete.

# 1: \_\_\_\_\_/28

# 2: \_\_\_\_\_/12

# 3: \_\_\_\_\_/10

If you need more space for one of your solutions, use the last pages of the exam and indicate clearly the part of your work that should be marked.

# 4: \_\_\_\_\_/14

# 5: \_\_\_\_\_/12

# 6: \_\_\_\_\_/14

*Good Luck!* TOTAL: \_\_\_\_\_/90

**Question 1.** [28 MARKS]**Multiple Choice (2 marks each)**

1.1. Select the option below that is **NOT** equivalent to the following statement: `int *p = 20;`

- A. `int* p = 20;`
- B. `int * p = 20;`
- C. `int *p; p = 20;`
- D. `int *p; *p = 20;`
- E. None of the above: All of the above statements are equivalent.

1.2. Consider the program below, which has a function call missing from `main()`. Select the correct function call that would result in the program printing the string "Hello, world!".

```
void increment1(int count) {  
    count++;  
}
```

```
void increment2(int *count_ptr) {  
    (*count_ptr)++;  
}
```

```
int main(int argc, char **argv) {  
    int count = 0;  
    _____; // Correct function call (from the options below) goes here  
    if(count == 1) printf("Hello, world!");  
    return 0;  
}
```

- A. `increment1(&count);`
- B. `increment1(*count);`
- C. `increment2(count);`
- D. `increment2(&count);`
- E. `increment2(*count);`

1.3. Suppose we have the following declarations at the start of a program. Select the statement below that does **NOT** assign the value 4 to the variable `q`.

```
int x = 4;
int *y = &x;
int **z = &y;
```

```
int q;
```

- A. `q = x;`
- B. `q = *(&x);`
- C. `q = *y;`
- D. `q = &(*x);`
- E. `q = **z;`

1.4. Consider the following code fragment, and assume (for the purposes of answering this question) that integers are 4 bytes and pointers are 8 bytes. Select the correct answer for how many bytes are allocated on the stack, and how many on the heap.

```
int *A = malloc(sizeof(int) * 20);
```

- A. Stack: 8 Heap: 160
- B. Stack: 8 Heap: 80
- C. Stack: 0 Heap: 168
- D. Stack: 0 Heap: 88
- E. None of the above.

1.5. Consider the program below, and select the correct output that matches what the program will print.

```
struct StudentNode {
    char *name;
    int num;
};

void updateStudent(struct StudentNode s) {
    s.num = s.num + 2;
    strcat(s.name, " v2");
}

int main() {
    struct StudentNode s1;
    s1.num = 12345;
    s1.name = malloc(256*sizeof(char));
    strcpy(s1.name, "Bob");
    updateStudent(s1);
    printf("%s; ", s1.name);
    printf("%d\n", s1.num);
}
```

- A. Bob; 12345
- B. Bob; 12347
- C. Bob v2; 12345
- D. Bob v2; 12347
- E. This code will cause a segmentation fault, because `strcpy` is unsafe.

1.6. Select the statement below that is **TRUE** (or *All of the above*, if they are all true), concerning signals.

- A. A process can only send a signal to its parent process or its child processes.
- B. The `kill` system call is used only to terminate processes.
- C. The default action of a `SIGKILL` signal cannot be modified.
- D. A signal handler runs in a separate process from the program receiving the signal.
- E. All of the above are true.

1.7. Recall: (1) The `tee` program, which reads from `stdin` and copies the stream to **both** `stdout` and to the file specified by the command-line argument; (2) The `sort` program, which sorts the lines of text contained in the file specified by the command-line argument, and outputs to `stdout`; and (3) The `head` program, which reads from `stdin` and outputs the first 10 lines to `stdout`. Consider the following shell command, and select the statement below that is **TRUE**.

```
sort input.txt | tee output.txt | head
```

- A. The contents of `input.txt` and `output.txt` are identical.
- B. Only the first 10 lines of `output.txt` will be displayed in the terminal.
- C. The entire contents of `input.txt` followed by the first 10 lines of `output.txt` will be displayed in the terminal.
- D. The entire contents of `output.txt` followed by the first 10 lines of `output.txt` will be displayed in the terminal.
- E. The entire contents of both `input.txt` and `output.txt`, followed by the first 10 lines of `output.txt`, will be displayed in the terminal.

1.8. Select the statement below that is **TRUE**, concerning the invocation of the `read()` or `write()` system calls on a pipe.

- A. A `read()` invocation on a pipe that has completely filled its buffer will return the number of bytes that were read, but only if at least one process still has an open write descriptor to the pipe.
- B. A `read()` invocation on an empty pipe will block indefinitely until data is available, or until all processes close their write descriptors to the pipe.
- C. A `read()` invocation on an empty pipe will immediately generate a `SIGPIPE` signal, if all processes have closed their write descriptors to the pipe.
- D. A `write()` invocation on a full pipe will immediately generate a `SIGPIPE` signal, if another process has an open read descriptor to the pipe but never invokes `read()`.
- E. A `write()` invocation on a pipe will block indefinitely if no process has an open read descriptor to the pipe.

1.9. Select the statement below that is **TRUE**, concerning the `fork()` system call.

- A. After `fork()` returns, the child process will always execute first .
- B. After `fork()` returns, the parent process will always execute first .
- C. If a child process updates the value of a **static** variable, it will be updated in the parent process as well.
- D. The child process will inherit the open file descriptor table of its parent.
- E. The parent process will not terminate until all of its children have terminated.

1.10. Select the statement below that is **TRUE** (or *All of the above*, if they are all true), concerning the `exec()` family of system calls.

- A. The `exec` system calls do not create a new process.
- B. After `exec` is called, the new program inherits the open file descriptor table of the original program.
- C. If a call to `exec` succeeds, it will not return a value.
- D. After `exec`, the new program will retain the PID of the previously-running program.
- E. All of the above are true.

1.11. Select the statement that is **TRUE**, considering the program consisting of the following two source files. Assume that the program is compiled with `gcc -o hello hello.c main.c`.

```
/* Complete contents of main.c */
void hello(void);
int main(void) {
    hello();
    return 0;
}

/* Complete contents of hello.c */
#include <stdio.h>

void hello() {
    printf("Hello, world!\n");
}
```

- A. The program will fail to compile, because `main.c` is missing the line `#include "hello.c"`.
- B. The program will fail to compile, because `main.c` is missing the line `#include <stdio.h>`.
- C. The program will fail to compile, because the `gcc` command is incorrect.
- D. The program will compile successfully, but its behaviour is undefined (e.g., it may trigger a segmentation fault or print out garbage values).
- E. The program will compile and print `Hello, world!` to the terminal.

1.12. Select the statement below that is **TRUE** (or *All of the above*, if they are all true), concerning threads and processes.

- A. Processes do not share the same memory space, but threads belonging to the same process do.
- B. Process creation with `fork` is slow, but thread creation is much faster.
- C. Each thread has its own global `errno` variable.
- D. Threads belonging to the same process share the same heap and global variables, but have separate function call stacks.
- E. All of the above are true.

1.13. Select the statement that is **TRUE**, considering the program below.

```
struct my_struct {
    char *name;
};

void array_chief(struct my_struct *s) {
    char new[4] = {'a', 'b', 'c', '\0'};
    s->name = new;
}

int main(void) {
    struct my_struct s1;
    s1.name = "Bob";
    array_chief(&s1);
    printf("%s\n", s1.name);
    /* Do other things, call some other functions... */
    return 0;
}
```

- A. The program will fail to compile, because `array_chief()` assigns an array to a pointer variable.
  - B. The program will fail to compile, because `s1.name` in `main()` is a read-only string literal, which `array_chief()` attempts to overwrite.
  - C. The program will compile without errors, but a segmentation fault will be triggered when `main()` assigns the return value of `array_chief()` to `s1.name`, since the latter is a read-only string literal.
  - D. The program will compile without errors, but its behaviour is undefined, e.g., it may print `abc` or it may print other garbage values or result in other unpredictable behaviour.
  - E. The program will compile without errors, and will always print out `abc`.
- 1.14. Select the file type that is most appropriate for opening with `fopen` using the `"rb"` flag.
- A. A C program's header file(s).
  - B. A C program's object file(s).
  - C. A C program's source file(s).
  - D. A Makefile.
  - E. None of the above are appropriate to open using the `"rb"` flag.

**Question 2.** [12 MARKS]**Structs and Dynamically Allocated Arrays**

The program below defines a struct to manage a *dynamically-allocated array*. Your job is to write two helper functions, `initialize` and `add`, to make the program work correctly. The requirements are as follows:

- `initialize` initializes an `array_list` struct, which is passed in as the single input parameter. By default, the array should have a capacity of 5.
- `add` appends one or more integers contained in the array passed in the first parameter (which may be either on the stack or the heap). The second parameter specifies the number of integers contained in the array being passed in the first parameter. The third parameter specifies the `array_list` to which the new integer(s) should be added. If there is not enough space in the `array_list`, a new array should be allocated that is big enough to hold **double** the new elements plus the existing elements (i.e. double the total of the two), and the contents of the old array should be moved into the new one (for full marks, do this without writing a loop) before appending the new integers to the list.

We have not provided you with the function signatures for `initialize` and `add`: You need to determine these yourself, in a way that satisfies both the requirements given above and the correctness of the program below.

Both `initialize` and `add` should perform any necessary error checking: They should return 0 on success, and -1 on failure.

```
struct array_list {
    int *contents;
    size_t capacity;      // Current capacity of the array
    size_t curr_elements; // Number of elements currently occupied in the array
};

int main(void) {
    struct array_list list;
    initialize(&list);
    int a[11] = {2, 0, 9, 4, 5, 6, 7, 8, 9, 10, 11};
    add(a, 11, &list);

    // The loop below should print "2 0 9 4 5 6 7 8 9 10 11 "
    for(int i = 0; i < list.curr_elements; i++)
        printf("%d ", list.contents[i]);
    /* The program does some fancy stuff with the array here,
     * generates some output, and performs any remaining
     * cleanup before terminating.
     */
    return 0;
}
```

You may write your helper functions on the next page (but two blank pages have been provided, in case you need the extra space).





**Question 3.** [10 MARKS]**Signals**

Study the following program that installs a signal handler.

```
int turn = 0;

void handler(int code) {
    if(turn == 0) {
        fprintf(stderr, "First\n");
        turn = 1;
        /* D */
    }
    else {
        fprintf(stderr, "Second\n");
        kill(getpid(), SIGQUIT);
    }
    fprintf(stderr, "Here\n");
}

int main(void) {
    struct sigaction sa;
    sa.sa_handler = handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGINT);

    /* A */

    sigaction(SIGTERM, &sa, NULL);

    /* B */

    fprintf(stderr, "Done\n");

    /* C */
    return 0;
}
```

On the next page, provide the output of the above program when the events described in each subquestion occurs, assuming that the code runs correctly, i.e., no undefined behaviour or other unspecified events occur. Treat each subquestion as if the program were restarted. Each event is described as a signal that is delivered to the program just before the program executes the line of code following the specified comment line (i.e., A, B, C, or D). Give the **TOTAL** output of the program in each case.

**Note:** When a process exits due to a SIGTERM, SIGQUIT, or SIGKILL, the shell process prints “Terminated”, “Quit”, or “Killed”, respectively, after the program terminates. Include these messages in your answers where applicable.

3.1. Two SIGTERM signals arrive one after the other at A.

3.2. SIGTERM arrives at B and SIGTERM arrives again at C.

3.3. SIGTERM arrives at B and SIGINT arrives at D.

3.4. SIGTERM arrives at B and SIGKILL at D

3.5. True or False: `fprintf` is async-signal-safe, assuming it is used in a single-threaded program.

**Question 4.** [14 MARKS]**Forking**

4.1 Consider the program below, and enter the correct numbers in the table on the right-hand side.

```
int main(void)
{
    int i = 0;
    printf("Broccoli\n");
    int r = fork();
    printf("Cucumbers\n");
    if (r == 0) {
        printf("Kale\n");
        int k = fork();
        if (k >= 0) {
            printf("Peppers\n");
        }
    } else if (r > 0) {
        wait(NULL);
        printf("Cabbage\n");
        while(fork() == 0) {
            printf("Carrots\n");
            i++;
            if(i == 3) break;
        }
        i = 0;
        while(fork() > 0) {
            printf("Spinach\n");
            i++;
            if(i == 2) break;
        }
    }
    return 0;
}
```

Fruit Name	Times Printed
Broccoli	
Cucumbers	
Kale	
Peppers	
Cabbage	
Carrots	
Spinach	

You may use the blank space above for any sketches or rough work (it will not be marked). Then, please also answer the questions on the next page.

Assuming that the above program runs without errors (e.g., `fork` always returns successfully, and the program is not terminated by a signal such as `SIGKILL`):

4.2. How many distinct processes print “Spinach”?

4.3. How many distinct processes print “Carrots”?

4.4. True or False: The second line of output should **ALWAYS** be “Cucumbers”.

4.5. True or False: The last line of output should **ALWAYS** be “Spinach”.

4.6. True or False: “Peppers” is **ALWAYS** printed before “Cabbage”.

4.7. List all the vegetable(s) that will **NEVER** be printed after the bash prompt re-appears.

4.8. List all the vegetable(s) that **MIGHT** be printed after the bash prompt re-appears.

**Question 5.** [12 MARKS]**Pipes**

Write a program that forks two children. We refer to the first child (i.e. the child that is created first) as Child A, and the second as Child B.

Child A must be able to send a stream of bytes to Child B over a pipe. All processes must close file descriptors at the earliest appropriate point in the program, and perform any necessary error checking. Right after this is done, Child A must invoke `foo_a()` and Child B must invoke `foo_b()` (you may assume that neither of these functions will return back to `main`). You do not need to worry about writing any data to the pipe—simply ensure that the pipe is correctly set up to allow Child A to send data over it to Child B.

```
int main(void)
{
```



**Question 6.** [14 MARKS]**Sockets and Select**

Consider the chat server program below (simplified from Tutorial 11) which listens for incoming messages from any connected client, and echos the received messages to all other clients. Fill in the blanks with the correct C statements to make the program work correctly, as stated.

```
/* #include statements cut out to save space */
#define PORT 30000
#define MAX_CONNECTIONS 12

struct sockname {
    int sock_fd;
    char *username;
};

/* Accept a connection and return the new client's file descriptor,
 * or -1 on error. The client's file descriptor and user name will
 * be saved in the struct sockname array pointed to by *users.
 */
int accept_connection(int fd, struct sockname *users);

/* Read a message from client_index in the struct sockname array
 * pointed to by *users, and send the message to all other
 * clients that are currently connected to the server.
 * Return client_index's file descriptor if the connection has
 * been closed, or 0 otherwise.
 */
int read_from(int client_index, struct sockname *users);

/* Create a new socket, set it to listen for incoming connections
 * on the specified port, and return the socket's file descriptor.
 */
int setup_socket(int port);

int main(void) {
    // Create and initialize users array
    struct sockname users[MAX_CONNECTIONS];
    for (int i = 0; i < MAX_CONNECTIONS; i++) {
        users[i].sock_fd = -1;
        users[i].username = NULL;
    }

    // Create the socket FD.
    int sock_fd = setup_socket(PORT);

    int max_fd = -----;
```

```

// Initialize file descriptor set, to listen to multiple file descriptors.
fd_set all_fds, listen_fds;

-----;
-----;

while (1) {
    listen_fds = all_fds;

    if (-----) {
        perror(NULL);
        exit(1);
    }
    // If there is a pending connection, accept the connection from the new client.

    if (-----) {
        int client_fd = accept_connection(sock_fd, users);

        if (-----) {
            -----;
        }

        -----;
        printf("Accepted connection\n");
    }
    // Next, process any disconnections or messages received from clients
    for (int i = 0; i < MAX_CONNECTIONS; i++) {
        if (users[i].sock_fd > -1 && // complete this if statement on next line

            -----) {
            // Don't worry about reducing max_fd

            int client_closed = -----;

            if (-----) {
                -----;

                -----;
                printf("Client %d disconnected\n", client_closed);
            }
        }
    }
}
return 1; // Should never get here.
}

```

*[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.]*

*[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.]*

*[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:**

```

int accept(int sock, struct sockaddr *addr, int *addrlen)
char *asctime(const struct tm *timeptr)
int bind(int sock, struct sockaddr *addr, int addrlen)
int close(int fd)
int closedir(DIR *dir)
int connect(int sock, struct sockaddr *addr, int addrlen)
char *ctime(const time_t *clock);
int dup2(int oldfd, int newfd)
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *argv[])
int fclose(FILE *stream)
int FD_ISSET(int fd, fd_set *fds)
void FD_SET(int fd, fd_set *fds)
void FD_CLR(int fd, fd_set *fds)
void FD_ZERO(fd_set *fds)
char *fgets(char *s, int n, FILE *stream)
int fileno(FILE *stream)
pid_t fork(void)
FILE *fopen(const char *file, const char *mode)
int fprintf(FILE * restrict stream, const char * restrict format, ...);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
int fseek(FILE *stream, long offset, int whence);
    /* SEEK_SET, SEEK_CUR, or SEEK_END*/
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
pid_t getpid(void);
pid_t getppid(void);
unsigned long int htonl(unsigned long int hostlong) /* 4 bytes */
unsigned short int htons(unsigned short int hostshort) /* 2 bytes */
char *index(const char *s, int c)
int kill(int pid, int signo)
int listen(int sock, int n)
void *malloc(size_t size);
void *memmove(void *dest, const void *src, size_t n);
unsigned long int ntohl(unsigned long int netlong)
unsigned short int ntohs(unsigned short int netshort)
int open(const char *path, int oflag)
    /* oflag is O_WRONLY | O_CREAT for write and O_RDONLY for read */
DIR *opendir(const char *name)
int pclose(FILE *stream)
int pipe(int filedes[2])
FILE *popen(char *cmdstr, char *mode)
ssize_t read(int d, void *buf, size_t nbytes);
struct dirent *readdir(DIR *dir)
int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
int sigaddset(sigset_t *set, int signum)
int sigemptyset(sigset_t *set)
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
    /* how has the value SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK */
unsigned int sleep(unsigned int seconds)
int socket(int family, int type, int protocol) /* family=AF_INET, type=SOCK_STREAM, protocol=0 */
int sprintf(char *s, const char *format, ...)
int stat(const char *filename, struct stat *buf)

```

```

char *strchr(const char *s, int c)
size_t strlen(const char *s)
char *strncat(char *dest, const char *src, size_t n)
int strncmp(const char *s1, const char *s2, size_t n)
char *strncpy(char *dest, const char *src, size_t n)
long strtol(const char *restrict str, char **restrict endptr, int base);
char *strrchr(const char *s, int c)
char *strstr(const char *haystack, const char *needle)
int wait(int *status)
int waitpid(int pid, int *stat, int options) /* options = 0 or WNOHANG*/
ssize_t write(int d, const void *buf, size_t nbytes);

```

**Useful macros:**

```

WIFEXITED(status)      WEXITSTATUS(status)
WIFSIGNALED(status)   WTERMSIG(status)
WIFSTOPPED(status)    WSTOPSIG(status)

```

**Useful structs:**

```

struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
struct sockaddr_in {
    sa_family_t sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char pad[8]; /*Unused*/
}
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 */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t st_blocks; /* number of 512B blocks allocated */
    time_t st_atime; /* time of last access */
    time_t st_mtime; /* time of last modification */
    time_t st_ctime; /* time of last status change */
};

```