PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

St. George Campus

DECEMBER 2017 EXAMINATIONS

CSC 209H1F
Liu, Reid
3 hours

No Examination Aids

Student Number: |_|_|_|_|_|_|_|_|_|_|_|_|_|

Last (Family) Name(s): _____

First (Given) Name(s): _____

---

*Do **not** turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below *carefully*.)

This final examination consists of 10 questions on 18 pages. A mark of at least 31 out of 77 on this exam is required to pass this course, otherwise your final course grade will be no higher than 47%. *When you receive the signal to start, please make sure that your copy of the examination is complete.*
You are not required to add any `#include` lines, and unless otherwise specified, you may assume a reasonable maximum for character arrays or other structures. Error checking is not necessary unless it is required for correctness or specifically requested.

Good Luck!

Marking Guide

# 1:	_____/ 8
# 2:	_____/ 6
# 3:	_____/ 6
# 4:	_____/10
# 5:	_____/ 8
# 6:	_____/ 8
# 7:	_____/ 6
# 8:	_____/10
# 9:	_____/ 9
# 10:	_____/ 6
TOTAL:	_____/77

Question 1. [8 MARKS]

Circle the correct answer for the statements below.

TRUE FALSE Multiple processes on a machine may use the same port number to listen for connections.

TRUE FALSE A `SIGPIPE` signal is sent when a process makes a `write` call to a closed socket or pipe.

TRUE FALSE The standard line ending used in network communications is `'\n'`.

TRUE FALSE The function `perror` is used to print error messages for any function.

TRUE FALSE Given the following code, the file `fp` can be closed immediately after the `dup2` call because it is no longer needed.

```
FILE *fp = fopen("myfs", "r");
dup2(fileno(fp), fileno(stdin));
```

TRUE FALSE Suppose we have a Makefile with a target `t`. Running `make t` will always create a file called `t` in the current working directory.

TRUE FALSE The code below copies the fields of `b` into `a`.

```
struct node a, b;
//missing code to assign values to the fields in b
a = b;
```

TRUE FALSE The code below is an example of a memory leak.

```
struct node {
    int val;
    struct node *next;
};
void free_list(struct node *head) {
    while(head != NULL) {
        free(head);
        head = head->next;
    }
}
```

Question 2. [6 MARKS]

Part (a) [4 MARKS]

Write a shell script that adds every subdirectory of your current working directory to your `PATH` variable. Note that the `PATH` variable requires an absolute path to each subdirectory. Remember that the environment variable `PWD` stores the absolute path to the current working directory. (Do not recurse into subdirectories.)

Part (b) [2 MARKS]

A program `my_prog` in the *parent* directory of the current working directory prints out some plaintext when run.

Write a single shell command that computes the number of words in the output of `my_prog`, and prints this number to the file called `outfile`.

Question 3. [6 MARKS]

For each pair of code snippets below, select `match` if the pair would output the same thing, `does not match` if the output would be different, or `may not match` if the output might match, but not always.

If you select `does not match` or `may not match`, briefly explain why.

Part (a) [1 MARK]

```
int a = 3;
int *b = &a;
printf("%d", *b);
```

```
int *c = malloc(sizeof(int));
*c = 3;
printf("%d", c);
```

Selection

☐ `match` ☐ `does not match` ☐ `may not match`

Explanation

Part (b) [1 MARK]

```
int x = 1 << 3;
int y = x | 1;
printf("%d %d\n", x, y);
```

```
int z = 8;
printf("%d %d\n", z, z + 1);
```

Selection

☐ `match` ☐ `does not match` ☐ `may not match`

Explanation

Part (c) [1 MARK]

```
int res = fork();
if (res == 0) {
    execl("/bin/ls", "ls", NULL);
    exit(42);
} else if (res > 0) {
    int status;
    wait(&status);

    if (WIFEXITED(status)) {
        printf("%d", WEXITSTATUS(status));
    }
}
```

```
printf("%d", 42);
```

Selection

☐ match ☐ does not match ☐ may not match

Explanation

Part (d) [1 MARK]

```
char *s = "hillo world";
s[1] = 'e';
printf("%s", s);
```

```
printf("hello world");
```

Selection

☐ match ☐ does not match ☐ may not match

Explanation

Part (e) [1 MARK]

```
char *s = malloc(strlen("hello world"));          printf("hello world");
strcpy(s, "hello world");
printf("%s", s);
```

Selection

☐ match ☐ does not match ☐ may not match

Explanation

Part (f) [1 MARK]

```
char *s = "hello world";                          printf("hello world");
char *t = malloc(sizeof(s));
strcpy(t, s);
printf("%s", t);
```

Selection

☐ match ☐ does not match ☐ may not match

Explanation

Question 4. [10 MARKS]

Consider the program below.

Part (a) [2 MARKS]

(i) What is the value of `strlen(course)` just before `stem` returns? _____

(ii) What is the value of `strlen(s)` just before `stem` returns? _____

(iii) What is the value of `strlen(ptr)` just before `stem` returns? _____

(iv) What does the print statement in `main` print? _____

Part (b) [8 MARKS]

Draw a complete memory diagram showing all of the memory allocated immediately before `stem` returns. Clearly label the different sections of memory (stack, heap, global), as well as different stack frames. You must show where each variable is stored, but make sure it's clear in your diagram what is a variable *name* vs. the *value* stored for that variable. You may show a pointer value by drawing an arrow to the location in memory with that address, or may make up address values.

```
char *stem(char *course) {
    char *s = malloc(strlen(course) + 1);
    strncpy(s, course, strlen(course) + 1);
    char *ptr = s;
    while(*ptr != 'H') {
        ptr++;
    }
    *ptr = '\0';
    // Draw memory diagram at this point
    return s;
}

int main() {
    char *c1 = "CSC209H1F";
    char *c2 = stem(c1);
    printf("%s %s\n", c1, c2);
    return 0;
}
```

Question 5. [8 MARKS]

Implement the following function according to its documentation.

```
struct pair {
    char *name;
    char *value;
}

/*
 * You are given a null-terminated string, s, in the following form:
 * - It contains between 1 and 10 lines, where each line except the last ends with '\n'
 * - Each line is of the form "<name>: <value>", where <name> and <value> are non-empty strings.
 *   Note that there is exactly one space between the colon and value string.
 * - Each name and value string do *not* contain a colon (but may contain spaces).
 *
 * For example:
 *   - "a: bc"
 *   - "name1: a\nname2: b"
 *   - "Content-Type: text/plain\nHost: teach.cs.toronto.edu"
 *
 * Your job is to parse the string, storing each name and value in an array of
 * struct pairs. Any unfilled pairs in the array have their name and value
 * fields set to NULL (we've initialized this for you).
 */
struct pair *parse_pairs(char *s) {
    // Initialize an array (assume there's at most 10 pairs in s).
    struct pair *data = malloc(10 * sizeof(struct pair));
    for (int i = 0; i < 10; i++) {
        data[i].name = NULL;
        data[i].value = NULL;
    }









    return data;
}
```

Question 6. [8 MARKS]

Write a program that takes at least one command-line argument, where each argument is the name of an executable to run.

The program should fork a new child for each command-line argument, and the child should call `execl` to run the executable named by the argument. (The executable itself takes no arguments.) The child should exit with a non-zero value if the `execl` fails.

The parent process should wait for all of its child processes to complete. If *all* of its child processes exit successfully with an exit status of 0, the parent prints `"Success\n"`. Otherwise, the parent prints `"Failure\n"`.

Here is an example of how the program could be called:

```
$ ./run_all ls date ps
```

Question 7. [6 MARKS]

Suppose we want to write a program that does the following:

- The parent creates a child process.
- The parent sends the child process a single integer.
- The child multiplies the integer by 2, then sends the result back to the parent.
- The parent prints the result to standard output.

Here is an incorrect implementation of this program. We have omitted error-checking for brevity. Note that lack of error checking is not the problem.

```
int main() {
    int val, result;
    int fd[2];
    pipe(fd);

    int pid = fork();

    if (pid > 0) { // Parent process
        val = 10;
        write(fd[1], &val, sizeof(int));
        close(fd[1]);
    } else { // Child process
        read(fd[0], &val, sizeof(int));
        close(fd[0]);
        result = val * 2;
        write(fd[1], &result, sizeof(int));
        close(fd[1]);
        return 0;
    }

    // Parent process here
    read(fd[0], &result, sizeof(int));
    close(fd[0]);
    printf("result: %d\n", result);
    return 0;
}
```

Part (a) [2 MARKS] Explain what could go wrong with the *parent process* when we run this program.

Part (b) [2 MARKS] Explain what could go wrong with the *child process* when we run this program.

Part (c) [2 MARKS] Briefly explain how to fix the program.

Question 8. [10 MARKS]

For each of the statements below, explain one example of what would cause the outcomes or results described in the comments in each box. Answers must be precise.

`result = read(fd, buf, SIZE);` `// process blocks`	
`result = read(fd, buf, SIZE);` `// result == 0`	
`result = read(fd, buf, SIZE);` `// result < SIZE`	
`result = read(fd, buf, SIZE);` `// process exits with a segmentation fault`	
`result = write(fd, buf, SIZE);` `// process blocks`	
`result = wait(&status);` `// process blocks`	
`result = wait(&status);` `// returns immediately and result > 0`	
`result = select(maxfd, &rset, NULL, NULL, NULL);` `// process blocks`	
`result = accept(fd, &addr, sizeof(addr));` `// process blocks`	
`result = accept(fd, &addr, sizeof(addr));` `// result > 0`	

Question 9. [9 MARKS]

Below is an implementation of an interactive client that reads a hostname from standard input, makes a connection to a server running on `PORT`, and then reads two pieces of information from the server: the number of users (as a 4-digit null-terminated string), and the average load (as a 4-digit null-terminated string).

Assume `connect_to_server` and `add_host` are implemented correctly, and that `connect_to_server` will not block. Error checking is omitted for brevity.

```
#define MAXHOSTS 10
struct server {
    int soc;
    char host[64];
    char number[5];
    char load[5];
};
// Add hostname and soc to an empty slot in servers array.
void add_host(struct server servers, char *hostname, int soc);

// Create a socket and connect to the server at port and hostname. Assume connection succeeds.
int connect_to_server(int port, const char *hostname);

int main() {
    fd_set rset, allset;
    int maxfd;
    char buf[MAXLINE];
    struct server servers[10];

    // initialize all strings in servers to empty strings, and the soc field to -1
    init_server(servers);

    FD_ZERO(&allset);
    maxfd = 0;

    while (1) {
        // Get hostname from stdin and make connection
        read(fileno(stdin), buf, MAXLINE);
        int soc = connect_to_server(PORT, buf);

        add_host(servers, buf, soc);
        FD_SET(soc, &allset);
        if (soc > maxfd)  maxfd = soc;

        rset = allset;
        select(maxfd+1, &rset, NULL, NULL, NULL);

        // Check to see which hosts have data ready
        for (int i = 0; i < MAXHOSTS; i++) {
            if (FD_ISSET(servers[i].soc, &rset)) {
                read(servers[i].soc, servers[i].number, 5);
                read(servers[i].soc, servers[i].load, 5);
                printf("%s: %s %s\n", servers[i].host, servers[i].number, servers[i].load);
            }
        }
    }
}
```

Part (a) [3 MARKS]

When we run the program all the output is correct, but we notice that we cannot type in another host name to check until we have received all of the output from the previous host. There are three problems with the program that need to be fixed to address this issue. Explain each problem. Failing to check for errors is **not** a problem with the code, and all helper functions are correctly implemented.

Part (b) [6 MARKS]

Fix the code by rewriting it below. You do not need to re-write any code above the call to `FD_ZERO`.

Question 9. (CONTINUED)

Question 10. [6 MARKS]

You are given a binary file that contains an array of structs of the type shown below. Complete the following function according to its documentation.

```
#define MAXNAME 32

typedef struct {
    char name[MAXNAME];
    int offset;
    int length;
} Inode;

/* Write an Inode containing the fields "name", "offset", and "length" to the
 * Inode at index "ind" in the file named "filename". Include appropriate error
 * checking to handle the case where "ind" is not a valid index.
 * Return 0 if the write was successful, and -1 in the case of error.
 * Do not use a loop.
 */
int write_inode(char *filename, int ind, char *name, int offset, int length) {
```

This page can be used if you need additional space for your answers.

Total Marks = 77

C function prototypes and structs:

```
int accept(int sock, struct sockaddr *addr, int *addrlen)
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)
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)
      /* mode can be "r", "w", "r+", "w+", "a", "a+" */
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);
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 pipe(int filedes[2])
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)
       /* actions include SIG_DFL and SIG_IGN */
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=PF_INET, type=SOCK_STREAM, protocol=0 */
int sprintf(char *s, const char *format, ...)
int stat(const char *file_name, 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);
```

```
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);

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*/
}
```

Shell comparison operators

Shell	Description
-d filename	Exists as a directory
-f filename	Exists as a regular file.
-r filename	Exists as a readable file
-w filename	Exists as a writable file.
-x filename	Exists as an executable file.
-z string	True if empty string
str1 = str2	True if str1 equals str2
str1 != str2	True if str1 not equal to str2
int1 -eq int2	True if int1 equals int2
-ne, -gt, -lt, -le	For numbers
!=, >, >=, <, <=	For strings
-a, -o	And, or.

Useful Makefile variables:

$@	target
$^	list of prerequisites
$<	first prerequisite
$?	return code of last program executed

Important environment variables: PATH, PWD, HOME

Useful shell commands:

cat, cut, echo, ls, read, set, sort, test, uniq
ps aux - prints the list of currently running processes
grep (returns 0 if match is found, 1 if no match was found, and 2 if there was an error)
grep -v displays lines that do not match
wc (-clw options return the number of characters, lines, and words respectively)
diff (returns 0 if the files are the same, and 1 if the files differ)

$0	Script name
$#	Number of positional parameters
$*	List of all positional parameters
"$@"	List of all positional parameters where each parameter is quoted
$?	Exit value of previously executed command