PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

St. George Campus

APRIL 2017 EXAMINATIONS

CSC 209H1S
Instructor:
Karen Reid
Duration: 3 hours

Examination Aids: One double-sided 8.5x11 sheet of paper. No electronic 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 9 questions on 20 pages. A mark of at least 31 out of 79 on this exam is required to pass this course. *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. For shell programs, you do not need to include the `#!/bin/sh`. Error checking is not necessary unless it is required for correctness.
Answers that contain a mixture of correct and incorrect or irrelevant statements will not receive full marks.

Good Luck!

MARKING GUIDE

# 1:	_____/12
# 2:	_____/ 6
# 3:	_____/ 6
# 4:	_____/ 8
# 5:	_____/ 8
# 6:	_____/ 5
# 7:	_____/15
# 8:	_____/ 9
# 9:	_____/10
TOTAL:	_____/79

Question 1. [12 MARKS]

Part (a) [8 MARKS] Circle the correct answer for the statements below.

TRUE FALSE You can send a signal to an unrelated process. (An unrelated process is neither an ancestor nor descendant.)

TRUE FALSE Given the following call to `accept`, the socket `sockfd` should be closed as soon as a new client has been accepted because it is no longer needed.

```
int accept(int sockfd, struct sockaddr *client, socklen_t *addrlen);
```

TRUE FALSE The code below has a memory leak.

```
void helper(int *arr, int size) {
    arr = malloc(sizeof(int) * size);
}
```

TRUE FALSE All character arrays must be null-terminated.

TRUE FALSE If a process tries to read from an open pipe before there is any data in the pipe, it could get an error.

TRUE FALSE Open file descriptors are inherited across a `fork` call but *not* across an `exec` call.

TRUE FALSE The reason we don't see the shell prompt appear when running a program in the foreground is that the program we are running in the foreground is a child of the shell, and the shell has called wait.

TRUE FALSE If we have two files that are hard links for the same file, they cannot have different permissions or a diferent owner.

Part (b) [2 MARKS]

Consider the following code fragment.

```
int age[5] = {4, 70, 52, 18, 16};
int *p = &age[2];
```

Check the boxes of the statements below that are true after the code fragment has run.

☐ The following is an illegal statement: `age[0] = p[0];`

☐ The values in the array after the execution of the statement `age[0] = p[0];` are `{52, 70, 52, 18, 16}`

☐ The expression `*(a+3)`could lead to a segmentation fault

☑ The expression `*(p+3)` could lead to a segmentation fault

Part (c) [2 MARKS]

Given the following code snippet, check the box for line or lines that would **not** lead to any problems with the code regardless of the value of `argv[1]`.

```
if(argc < 2) {
    return 1;
}
char name[30];
// missing code that modifies name such that it is a valid string
```

- ☐ `strncat(name, argv[1], 30);`
- ☐ `strncat(name, argv[1], strlen(argv[1]) + 1);`
- ☐ `strncat(name, argv[1], 30-strlen(name) + 1);`
- ☑ `strncat(name, argv[1], 30-strlen(name) - 1);`
- ☐ `strncat(name, argv[1], strlen(name) + 1);`

Question 2. [6 MARKS]

Part (a) [2 MARKS]

In the two's complement representation of a negative number, the highest bit has the value 1 for negative numbers and 0 for positive numbers. For example, in an 8-bit number **1**111 1001 is the two's complement representation for -7, while **0**000 0111 is the two's complement representation for 7.

`status` is an int that holds a two's complement number as the last 8 bits. Fill in the conditional expression below so that the if statement executes correctly. Note that you only need the information above to solve the problem.

```
unsigned int status;
// missing code to assigns to status a two's complement value in the lowest 8 bits


if(_____________________________________________) {
    printf("The value in num is negative\n");
} else {
    printf("The value in num is positive\n");
}
```

Part (b) [4 MARKS]

Write a shell program that takes a filename as an argument and prints the full path to each location in the `$PATH` variable where the file is found. You may not use any programs in your solution (such as where, ls, find etc) other than `tr` which is explained below.

Suppose the program is called `mywhere`, then here is an example.

```
reid@wolf$ ./mywhere python
/usr/local/bin/python
/usr/bin/python
```

Recall that `$PATH` is a colon-separated list of absolute paths. To turn it into a space-separated list, we use a program called `tr` that reads from standard input and translates all characters of its first argument to its second argument.

For example the following line will output "/usr/bin /sbin /bin"

```
echo /usr/bin:/sbin:/bin | tr : " "
```

Step 1: Write one line that will assign to a shell variable `dirs` the result of using `tr` on `$PATH`.

Step 2: Using the list in `dirs` complete the program described above.

Question 3. [6 MARKS]

Below is a simple C program. In the space below the program, draw a complete memory diagram showing all of the memory allocated immediately before `get_ext` returns. Clearly distinguish between 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 *get_ext(char *filename){
    char *ptr = strchr(filename, '.');
    char *ext = malloc(strlen(ptr+1) + 1);
    strncpy(ext, ptr+1, strlen(ptr+1)+1);
    // Draw memory at this point in the program.
    return ext;
}

int main()  {
    char name[8] = "prog.py";
    char *e = get_ext(name);
    printf("%s %s\n", name, e);
    return 0;
}
```

Question 4. [8 MARKS]

Each of the code fragments below has a problem. Explain what is wrong, and then fix the code by printing neatly on the code itself. Note that failing to check for errors on system calls is **not** the problem.

Part (a) [1 MARK]

```
if(argc > 2) {
    char *name = malloc(strlen(argv[2]) + 1);
    name = argv[2];
}
```

Part (b) [1 MARK]

```
if(argc > 2) {
    char s[30] = argv[1];
    s[0] = 'A';
}
```

Part (c) [1 MARK]

```
int *firstfive(int *A) {
    int vals[5];
    for(int i = 0; i < 5; i++) {
        vals[i] = A[i];
    }
    return vals;
}
```

Part (d) [2 MARKS]

```
struct point{
    int x;
    int y;
};
//Return the sum of the coordinates of p and negate their values
int sum_and_flip(struct point p) {
    int sum = p.x + p.y;
    p.x = p.x * -1;
    p.y = p.y * -1;
    return sum;
}
```

Part (e) [1 MARK]

```
int status;
if(wait(&status) != -1) {
    printf("status is %d", WEXITSTATUS(status));
}
```

Part (f) [2 MARKS] Note that failing to check for errors is *not* the problem with the code below.

```
int main() {
    char buf[8];
    int fd[2];
    pipe(fd);
    int r = fork();

    if(r == 0) {
        close(fd[1]);
        while(read(fd[0], buf, 8) == 8) {
            printf("buf=%s\n", buf);
        }
        exit(0);
    } else {
        close(fd[0]);
        strcpy(buf, "Hi ");
        write(fd[1], buf, 8);

        wait(NULL);
        exit(0);
    }
    return 1;
}
```

Question 5. [8 MARKS]

Part (a) [6 MARKS]

The program below creates a directory from the first argument to `main`. Using **only** the lines of code provided in the table on the right, fill in the code in the table on the left so that when the program receives the `SIGINT` signal, the permissions of the directory are changed to `0700` and prints a message notifying the user that the permissions have changed before the program terminates.

label	code line
—	`int main(int argc, char **argv) {`
—	`    mkdir(argv[1], 0400);`
z	`    printf("Directory created\n");`
—	`// The rest of the program is omitted.`
—	`// No code will be added below this comment.`

Use the labels in the table below to clearly identify the lines you have chosen. Note that some incorrect lines have been added as options, and that extra slots have been added to the table on the left.
An exmple has been given for you, using line "z".

label	code line
a	`chmod(path, 0700);`
b	`exit(0);`
c	`exit(1);`
d	`printf("Changed %s\n", path);`
e	`char *path;`
f	`path = NULL;`
g	`path = argv[1];`
h	`path = malloc(strlen(argv[1])+1);`
i	`sa.sa_flags = 0;`
k	`sa.sa_handler = &fixperm;`
m	`sa.sa_handler = fixperm;`
n	`sigaction(SIGINT, &sa, NULL);`
o	`sigaction(SIGINT, sa, NULL);`
p	`sigemptyset(&sa.sa_mask);`
s	`struct sigaction sa;`
t	`int fixperm(char *path) {`
u	`int fixperm(int code) {`
w	`void fixperm(char *path) {`
x	`void fixperm(int code) {`
y	`}`
z	`printf("Directory created\n");`

Part (b) [2 MARKS]

Add the error checking for the call to `chmod(path, 0700)` so that it prints an error message and terminates the program with a value of 1.

Question 6. [5 MARKS]

Consider the following program that compiles and runs to completion without error.

```
int main() {
    int r = fork();
    if(r == 0) {
        printf("First\n");
        r = fork();
        if(r == 0) {
            printf("Second\n");
            exit(0);
        } else {
            printf("Third\n");
        }
    }
    printf("Fourth\n");
    if(wait(NULL) != -1) {
        printf("Fifth\n");
    }
    exit(0);
}
```

Part (a) [1 MARK] How many processes are run including the process that calls `main`?

Part (b) [4 MARKS] Check the boxes for the statements below that are true.

- ☐ "First" must be displayed first
- ☐ "Second" must be displayed after "First" (not necessarily immediately after)
- ☐ "Fourth" is displayed three times
- ☐ "Second" could be displayed after "Fourth" (not necessarily immediately after)
- ☐ "Second" and "Third" could be displayed in either order
- ☐ The process that prints "Second" also prints "First"
- ☐ The second last line displayed must be "Fifth"
- ☐ The last line displayed must be "Fifth"

Question 7. [15 MARKS]

The goal of this question is to write a program called `redir` that allows standard input and/or standard output to be redirected from or to a file.

Command line arguments are specified below. Square brackets mean that the argument is optional.

```
redir [-i infile] [-o outfile] prog [arg1 ... ]
```

Examples:

`redir ls`	No redirection. The program `ls` is run.
`redir -i names sort`	The program `sort` is run and input is redirected from the file called `names`
`redir -o listing.txt ls -l`	The program `ls` is run with the argument `-l`, and the output is redirected to the file `listing.txt`
`redir -i data.csv -o names cut -f 1 -d ","`	The program `cut` is run with the arguments `-f 1 -d ","`. Input is redirected from `data.csv`, and output is redirected to the file `names`
`redir -o names -i data.csv cut -f 1 -d ","`	(Same as previous.)

Part (a) [2 MARKS]

Rewrite the first four examples as shell commands. In other words, how would you achieve the same results without using the program `redir`?

`redir ls`	
`redir -i names sort`	
`redir -o listing.txt ls -l`	
`redir -i data.csv -o names cut -f 1 -d ","`	

Part (b) [7 MARKS]

Complete the function below that parses the command line arguments. It returns the index into `argv` that holds the name of program to run, and sets the input and output file names as the last two parameters (where appropriate) so that they may be used in the main program when `read_options` returns. Do not use `getopt`.

Assume all arguments are passed correctly, so you don't need to consider too few arguments or other incorrect options.

```
int read_options(int argc, char **argv, _________________________________,

                                        _________________________________) {
```

```
/* Fill in the initialization for infile and outfile, and the arguments to read_options
 *
 * infile is set to the name of the input file to redirect from if the
 *        -i option was used, and is set to NULL if input is not redirected.
 * outfile is set to the name of the output file to redirect to if the
 *        -o option was used, and is set to NULL if output is not redirected.
 * index is set to the index into the argv array that holds the name of the
 *         program to run.
*/

int main(int argc, char **argv) {


    char *infile = ____________________________________________________________;


    char *outfile = ____________________________________________________________;


    int index  = read_options(argc, argv, ___________________, ___________________);
```

Part (c) [6 MARKS]

Given that the command line arguments have been parsed correctly, so that the variables from the the previous question have their correct values, complete the program as specified.

Question 8. [9 MARKS]

Suppose every machine in our labs is running a load monitoring server that is listening for connections on port 33333. When a client connects to the server, the server will send an integer (in network byte order) representing the load on the machine, and will then close the socket.

Your task is to complete the code below for a client that will connect to each of the lab machines in turn and store the load values in an array. The `host_ip` addresses are stored in a file with one address per line with unix line endings(`'\n'`). Example:

```
128.100.32.2
128.100.32.23
128.100.32.37
```

Assume there will never be more than MAXHOSTS IP addresses in the file.

```
#define MAXNAME 64
#define MAXHOSTS 30

struct machine {
    char *host_ip[MAXNAME];  // The ip address read from the file
    int load;
};

int main(int argc, char* argv[]) {


    struct machine lab[MAXHOSTS];
```

Part (a) [4 MARKS]

Populate the `host_ip` fields in the array `lab` with the IP addresses from a file called "hosts". Any unused array elements should have the `host_ip` field set to the empty string. Remember to remove newline character from the input.

Part (b) [5 MARKS]

Now, for each host IP address in the `lab` array, create a socket connection to the server, store the integer (in the correct format) received from the server in the `load` field, and close the socket.

Some of the setup is done for you below. Recall that `inet_pton` can be used to convert the string form of the host IP address into the form needed by the `sin_addr` field of the `struct sockaddr_in`.

```
int soc;
struct sockaddr_in peer;

peer.sin_family = PF_INET;
peer.sin_port = htons(PORT);
printf("PORT = %d\n", PORT);
```

Question 9. [10 MARKS]

Consider the following parts of a server that will manage multiple clients. Note that error checking has been removed for brevity and some parts of the program are missing. The questions below assume that the missing components are correctly implemented.

```
struct client {
    int fd;
    char *name;
    struct client *next;
};

static struct client *addclient(struct client *top, int fd, char *name) {
    struct client *p = malloc(sizeof(struct client));
    p->fd = fd;
    p->name = malloc(strlen(name)+1);
    strncpy(p->name, name, strlen(name)+1);
    p->next = top;
    top = p;
    return top;
}

int main(int argc, char **argv) {

    // Assume all variables are correctly declared and initialized
    // The appropriate calls have been made to set up the socket.

    FD_ZERO(&readset);
    FD_SET(listenfd, &readset);

    maxfd = listenfd;

    while (1) {
        n = select(maxfd + 1, &readset, NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &readset)){
            clientfd = accept(listenfd, (struct sockaddr *)&q, &len);

            char buffer[MAXNAME];
            read(clientfd, buffer, MAXNAME);
            printf("connection from %s\n", buffer);

            head = addclient(head, clientfd, buffer);
        }
        // code to check each client that is ready to read follow
```

Part (a) [4 MARKS]

When the server is run we notice that the program runs, but blocks forever in `select` even though new connections are coming in and there is always more data ready to read from all connected clients. This is the result of two problems in the code.

Explain the problem and fix it in the code above. (Lack of error checking is not the problem.)

Part (b) [3 MARKS]

Now that the problem in the previous question is fixed, the server correctly handles multiple connections as long as the clients don't experience any delays. But sometimes, when one client is slow, it seems to slow everything down. Explain the problem and how to fix it. (You don't need to write the code.)

Part (c) [3 MARKS]

Complete the function below to free the client list:

```
void free_list(struct client *top) {
```

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

Total Marks = 79

C function prototypes and structs:

```
int accept(int sock, struct sockaddr *addr, int *addrlen)
int bind(int sock, struct sockaddr *addr, int addrlen)
int chmod(const char *path, mode_t mode)
int close(int fd)
int closedir(DIR *dir)
int connect(int sock, struct sockaddr *addr, int addrlen)
int dup2(int oldfd, int newfd)
int execlp(const char *file, char *argv0, ..., (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)
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);
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 inet_pton(int af, const char *src, void *dst);
      /*af=PF\_INET; pass in address of sin\_addr field as dst */
int kill(int pid, int signo)
int listen(int sock, int n)
int mkdir(const char *pathname, mode_t mode)
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)
       /* actions include SIG_DFL and SIG_IGN */
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)
char *strrchr(const char *s, int c)
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 hostent {
    char *h_name; // name of host
    char **h_aliases; // alias list
    int h_addrtype; // host address type
    int h_length; // length of address
    char *h_addr; // address
}
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 */
};
```

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 shell commands:
cat, cut, echo, ls, read, sort, uniq, wc
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
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
$?	Exit value of previously executed command