PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

St. George Campus

AUGUST 2015 EXAMINATIONS

CSC 209H1Y
Instructor — Peter McCormick
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*.)

Marking Guide

# 1:	/ 8
# 2:	/20
# 3:	/ 9
# 4:	/14
# 5:	/ 3
# 6:	/12
# 7:	/14
TOTAL:	/80

This final examination consists of 7 questions on 19 pages. A mark of at least 32 out of 80 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. Error checking is *not* necessary unless it is required for correctness.

Good Luck!

Question 1. [8 MARKS]

Part (a) [2 MARKS]

What is the output produced by running the following shell commands?

```
rm -rf out
[[ ! -d out ]] && echo missing
false || echo $?
```

Output:

Part (b) [2 MARKS]

When your shell executes each of the following two lines, what is the difference between each execution in terms of *which* system calls are made?

```
wc -lw < quick.txt

cat quick.txt | wc -lw
```

Part (c) [2 MARKS]

How does a server process handle the listening socket differently than a connected client socket?

Part (d) [2 MARKS]

When using pipes across a `fork` call, describe one thing that can happen if you close too few or too many of your pipe file descriptors.

Question 2. [20 MARKS]

All of the code fragments below compile but many of them have something wrong with them. Check the appropriate box to indicate whether or not there is a problem and provide an explanation. **There are no marks for checking** SOMETHING IS WRONG **without an explanation.**

Part (a) [2 MARKS]

```
char name[] = "Albert";
name[2] = '\0';
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (b) [2 MARKS]

```
char season[] = { 's', 'u', 'm', 'm', 'e', 'r' };
printf("%s\n", season);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (c) [2 MARKS]

```
int i;
int xs[] = { 5, 108, 148, 207, 209 };
for (i = 1; i <= xs[0]; i++) {
    printf("%d\n", xs[i]);
}
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (d) [2 MARKS]

```
char course[9] = "CSC209H1Y";
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (e) [2 MARKS]

```
char **q;
q[0] = malloc(1);
q[0][0] = 209;
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (f) [2 MARKS]

```
char *p = malloc(1);
char *q = (char *) &p;
free(q);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (g) [2 MARKS]

```
int k;
char *r = malloc(1);
char *p;
char *q = p;
free(r);
p = &k;
q = r;
free(q);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (h) [2 MARKS]

```
int i;
char *ps[3] = { malloc(1), malloc(1), malloc(1) };
for (i = 0; i < 3; i++) free(ps[i]);
free(*ps);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (i) [2 MARKS]

```
char src[] = "fall";
char dest[] = "summer";
strncpy(dest, src, strlen(src) + 1);
printf("after summer is %s\n", dest);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Part (j) [2 MARKS]

```
char orig[] = "Hello World";
orig[5] = '\0';
char *dup = strdup(orig);
orig[5] = dup[5] = ' ';
printf("%s =?= %s\n", orig, dup);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Question 3. [9 MARKS]

The specification for the assignment 2 heap memory allocator had an intentional design flaw. If the heap was initially divided up by many small allocations, and then they were all freed, the heap would have all of its space available but large allocation requests could fail because the free list only contained small split up pieces, no one of which could satisfy the request. A Chunk in the free list is considered to be *fragmented* if the sub-region of the heap that it covers is adjacent to and contiguous with the next sub-region (if any) in the free list.

Part (a) [3 MARKS]

Complete the implementation of is_fragment below, returning 1 if the ch argument is fragmented and 0 otherwise. Recall that the free list was maintained in increasing addr order, which you may assume will be the case here.

```
typedef struct chunk {
    void *addr;
    size_t size;
    struct chunk *next;
} Chunk;

int is_fragment(Chunk *ch)
{
















}
```

Continued on next page.

Part (b) [6 MARKS]

Write a function `compact` to clean up a fragmented free list. Use `is_fragment` to determine if a given chunk can be merged with the next chunk (you may assume that it will behave correctly.) Make sure you properly update all relevant fields during merging, and do not forget to free any discarded dynamically allocated memory. After calling `compact`, there should be no fragmented chunks left in the list.

```
void compact(Chunk *list)
{




}
```

Question 4. [14 MARKS]

The *tape archive* or tar file format is built around a repeating, 512 byte fixed size record format, which leads to wasted space since many of those bytes are not necessarily being used. Instead suppose you wanted to use an alternative format which was more space efficient by using a variable sized file header and with no padding of file data. We define a *compact archive* or car file format to contain multiple files within it (just like a tar), with each being represented within the archive according to the following formatted byte sequence:

- name_len (1 byte): the number of bytes in the filename to follow, represented as a machine unsigned byte integer (i.e. between 0-255)
- name (name_len bytes): the actual characters of the filename
- size (4 bytes): the number of bytes of file data to follow, represented as a machine unsigned 32-bit integer
- data (size bytes): the actual file data itself

A car file will consist of this pattern repeated, one for each enclosed file, until the end of the archive file is reached.

Part (a) [9 MARKS]

Implement extract_compact_archive, which takes a file descriptor argument of a currently open car file and repeatedly reads and extracts the enclosing files until the end of the archive file. You may assume that the enclosed filenames do not contain any subdirectory components (i.e. no / separators.) Use the helper function copy_bytes (which is defined elsewhere) to actually read the enclosed file data and write it to a newly opened file.

```
// Function prototype: Reads at most 'num_bytes' from 'in_fd' and writes them to 'out_fd'
int copy_bytes(int in_fd, int out_fd, int num_bytes);

void extract_compact_archive(int fd)
{
```

Continued on next page.

This page can be used as additional space for `extract_compact_archive`.

```
}
```

Part (b) [5 MARKS]

Provide an implementation of the copy_bytes helper routine. Return the total number of bytes which have actually been read from in_fd. Be defensive and do not assume that in_fd necessarily has num_bytes left to read before the end of that file (hence the return value.) Conversely, do not accidentally read *more* than the requested number of bytes from in_fd. Do not dynamically allocate any memory, and use a fixed constant sized buffer.

```
int copy_bytes(int in_fd, int out_fd, int num_bytes)
{

}
```

Question 5. [3 MARKS]

Consider the following program:

```
void handler(int signum)
{
    switch (signum) {
    case SIGTERM: printf("terminate\n"); break;
    case SIGINT: printf("interrupt\n"); break;
    case SIGKILL: printf("kill\n"); break;
    default: printf("signal\n");
    }
}

int main()
{
    struct sigaction act = {};
    act.sa_flags = 0;
    act.sa_handler = SIG_IGN;
    sigaction(SIGTERM, &act, NULL);

    act.sa_handler = handler;
    sigaction(SIGINT, &act, NULL);
    sigaction(SIGKILL, &act, NULL);

    printf("pid %d\n", getpid());
    while (!feof(stdin)) {
        fgetc(stdin);
    }
    printf("parent done\n");
    return 0;
}
```

Part (a) [1 MARK]

Suppose you ran this program in one terminal window, and it happened to print out pid 1000. Then, in a second shell window, suppose you entered the shell command `kill -INT 1000`. What output or effect, if any, would you observe in the first window where the program is running?

Part (b) [1 MARK]

Next, suppose you entered the command `kill -TERM 1000` in the second window. What output or effect, if any, would you observe in the first window?

Part (c) [1 MARK]

Finally, suppose you entered the command `kill -KILL 1000` in the second window. What output or effect, if any, would you observe in the first window?

Question 6. [12 MARKS]

We have studied the system call primitives that a shell command interpreter like Bash uses to implement the pipe mechanism (i.e. the | operator), which connects the standard output of one program to the standard input of another. The piper program implements this same pipeline operation. It takes as arguments the specification of *two* program invocations, separated by a special triple-dash --- marker, and executes the two programs as though the first was being piped to the second. For example, consider the following two invocations of piper:

```
./piper grep CFLAGS Makefile --- wc -l

./piper progA A1 A2 A3 ... --- progB B1 B2 B3 ...
```

These invocations will have the same effect as directly running the following commands using the builtin shell pipe operator:

```
grep CFLAGS Makefile | wc -l

progA A1 A2 A3 ... | progB B1 B2 B3 ...
```

Starting from the code below, implement the piper program. Finding the triple-dash marker in the argument list has been handled for you. Make sure each sub-program is only executed with the arguments intended for that program.

```
int main(int argc, char *argv[])
{
    char **sep = argv;
    while (*sep != NULL) {
        if (strcmp(*sep, "---") == 0) break;
        sep++;
    }
    if (*sep == NULL) { fprintf(stderr, "Missing '---' separator in arguments\n"); return 1; }

    int pfds[2]; // 0 read end, 1 write end
```

This page can be used as additional space for `piper`.

```
    return 0;
}
```

Question 7. [14 MARKS]

Part (a) [7 MARKS]

Provide an implementation of the helper function `select_one` which simplifies the use of the `select` system call. This function takes an array of file descriptors (`fds`) and its length (`n`). Each array entry will contain either a valid non-negative integer representing a file descriptor, or `-1` to indicate an empty slot. This function should setup and call `select` to check for the read readiness of each non-empty descriptor slot in the array, and it should return the first descriptor in the array that `select` indicates is ready for reading, or -1 otherwise if an error occurred.

```
int select_one(int fds[], int n)
{
```

```
}
```

Part (b) [7 MARKS]

Complete the code below to implement a *broadcasting* echo server. Your server must handle multiple concurrent client connections, and each time that a client sends data to the server, it must retransmit it back to all currently connected clients (including the originating one.) Make use of the select_one helper from the first part of this question (you can assume that it will work as intended.) Make sure you gracefully handle disconnecting clients.

```
int main(int argc, char *argv[])
{
    struct sockaddr_in claddr;
    socklen_t addrlen = sizeof (claddr);
    const int N = FD_SETSIZE;
    int conns[N];
    int i;
    for (i = 0; i < N; i++) conns[i] = -1;

    int listenfd;
    // Assume that all setup calls to socket(), bind() and listen() are done; listenfd is listening
    conns[listenfd] = listenfd;
```

Continued on next page.

This page can be used as additional space for the broadcast echo server.

```
    return 0;
}
```

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

Total Marks = 80

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 connect(int sock, struct sockaddr *addr, int addrlen)
int creat(const char *pathname, int mode); 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);
char *index(const char *s, int c)
int kill(int pid, int signo)
int listen(int sock, int n)
unsigned long int ntohl(unsigned long int netlong)
unsigned short int ntohs(unsigned short int netshort)
int open(const char *path, int oflag, int mode)
       /* oflag is O_WRONLY | O_CREAT | O_TRUNC for write and O_RDONLY for read */
int pipe(int filedes[2])
ssize_t read(int d, void *buf, size_t nbytes);
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=AF_INET, type=SOCK_STREAM, protocol=0 */
int sprintf(char *s, const char *format, ...)
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_list; // address
}
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 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
$*	List of all positional parameters
"$@"	Quote preserving list of all positional parameters
$?	Exit value of previously executed command

Useful Makefile variables:

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