PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

St. George Campus

APRIL 2016 EXAMINATIONS

CSC 209H1S
Michelle Craig, David Liu
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*.)

Marking Guide

# 1:	______/15
# 2:	______/13
# 3:	______/ 7
# 4:	______/ 7
# 5:	______/11
# 6:	______/ 6
# 7:	______/11
# 8:	______/ 5
TOTAL:	______/75

This final examination consists of 8 questions on 20 pages. A mark of at least 30 out of 75 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 or specifically requested.

Good Luck!

Question 1. [15 MARKS]

Part (a) [2 MARKS]

Your current working directory contains a C program called `my_prog.c`, as well as some other files. Write the fragment of a bash shell program that checks if the file `my_prog` exists in the current directory, and if it does not, then compiles `my_prog.c` into an executable called `my_prog`, using the c99 standard.

Part (b) [2 MARKS]

The file `classlist` has the format shown in the box on the right. Write a single-line shell command to assign the number of students in section L0101 to the shell variable `NOONSIZE`.

```
990876234     Liu, Sid      L0101
1090876234    Chan, Vic     L5101
1480876134    Huynh, Elaine L0201
...
```

Part (c) [5 MARKS]

Write a shell script to do the following:

- Use a shell **loop** through the numbers 1 through 50, inclusive.
- If the number is less than 25, print it to the console.
- If the number is 25 or greater, run `my_prog` with the number as a command-line argument, and redirect its standard output to a file called "X.out", where X is replaced by the number.

So when the script is run, the numbers 1, 2, 3, ..., 24 should be printed to the screen, and the current directory should contain the files 25.out, 26.out, ..., 50.out.

Part (d) [4 marks]

Here is one line of output from running `ls -1` on the current directory.

```
-rwxr-x--- 1 liudavid instrs   8377 Apr 11 10:53 my_prog
```

Explain what you know about the permissions on the `my_prog` file.

Now show the output if you were to run these two commands on current directory.

```
chmod 641 my_prog; ls -l my_prog
```

Part (e) [2 marks]

Suppose we switch to a new directory which has the three files `cat.c`, `dog.c`, and `pet.h`, as well as a Makefile with the following contents:

```
all: cat snake

cat: cat.o dog.o
    gcc -o $@ $^

snake: dog.o
    echo I am a snake > snake

%.o: %.c pet.h
    gcc -c $<
```

The directory has no other files. Suppose the following commands are run one after the other. Fill in the table to show what files are created or modified as a result of running each command. If no files are created or modified after a particular command, write "NO CHANGE".

Command	Names of files created or changed
make snake	
make	

Question 2. [13 marks]

Some of the code fragments below have a problem. For each fragment indicate whether the code works as intended or whether there is an error (logical error, compile-time error/warning, or runtime error). Assume all programs are compiled using the C99 standard. For this question, we'll assume programs which do not terminate are errors as well. If there is an error in a fragment, explain **briefly** what is wrong in the box. We have intentionally omited the error checking of the system calls to simplify the examples. Do not report this as an error.

Some of the parts will use the following struct definition:

```
struct student {
    int age;
    char *name;
}
```

Part (a)

```
char *s = "Hello";
strcat(s, ", World!");
```

☐ Works as intended ☐ Error

Part (b)

```
int main(int argc, char **argv) {
    char ch;
    char *p = &ch;
    ch = argv[argc-1][0];
    printf("%c\n", p[0]);
    return 0;
}
```

☐ Works as intended ☐ Error

Part (c)

```
struct student hannah;
strcpy(hannah.name, "Hannah");
```

☐ Works as intended ☐ Error

Part (d)

```
struct student hannah = NULL;
// ... missing code ...
if (hannah != NULL) {
    hannah.age = 10;
}
```

☐ Works as intended ☐ Error

Part (e)

```
// Increase the age of a student by amt.
void increase_age(struct student s, int amt) {
    s.age += amt;
}

int main() {
    struct student rob;
    rob.age = 10;
    increase_age(rob, 5);
    printf("%d should be 15\n", rob.age);
}
```

☐ Works as intended ☐ Error

Part (f)

```
// Compute the sum of an array of integers
int compute_sum(int numbers[]) {
    int sum = 0;
    for (int i = 0; i < sizeof(numbers); i++) {
        sum += numbers[i];
    }
    return sum;
}
```

☐ Works as intended ☐ Error

Part (g)

```
int fd[2];
int result = fork();
pipe(fd);
if (result == 0) {
    close(fd[0]);
    write(fd[1], "csc209", 7);
} else {
    close(fd[1]);
    char buf[7];
    read(fd[0], buf, 7);
    printf("%s\n", buf);
}
exit(0);
```

☐ Works as intended ☐ Error

Part (h)

```
// Read all bytes from the file descriptors in ‘fds’ as characters,
// and print them. ‘num_fds’ is the number of file descriptors, and
// ‘max_fd’ is the value of the largest one.
void read_ints(int *fds, int num_fds, int max_fd) {
    char data;
    fd_set set;
    FD_ZERO(&set);
    for (int i = 0; i < num_fds; i++) {
        FD_SET(fds[i], &set);
    }
    while (select(max_fd + 1, &set, NULL, NULL, NULL) > 0) {
        for (int i = 0; i < num_fds; i++) {
            if (FD_ISSET(fds[i], &set)) {
                if (read(sum, &data, 1) > 0) {
                    printf("%c\n", data);
                }
            }
        }
    }
}
```

☐ Works as intended ☐ Error

Part (i)

```
struct node {
    int item;
    struct node *next;
}

// Compute the sum of the items in a linked list with the given head,
// but do not modify the list.
int sum(struct node *head) {
    int s = 0;
    while (head != NULL) {
        s += head->item;
        *head = *(head->next);
    }
    return s;
}
```

☐ Works as intended ☐ Error

Part (j)

```
// Remove the dots from word
char *word = "Ex.ampl.e";
char *result = malloc(strlen(word) + 1); // upper-limit if word has no dots
for (int i = 0; i < strlen(word); i++) {
    if (word[i] != '.') {
        strncat(result, word[i], 1);
    }
}
```

☐ Works as intended ☐ Error

Question 3. [7 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 `fun` 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.

```
int *fun(char *ptr) {
    int *int_ptr = malloc(sizeof(int));
    if (ptr[0] == 'Z') {
        *int_ptr = 1;
    } else {
        *int_ptr = 5;
        *ptr = 'Z';
    }
    // Draw the memory at this point in the execution
    return int_ptr;
}

int main() {
    char letters[4] = "abc";
    int *x;
    x = fun(letters);
    return *x;
}
```

Question 4. [7 MARKS]

Write a program which takes a single command-line argument, which is the name of a text file. The program should open the file, and replace every occurrence of the character `'a'` (but *not* `'A'`) with the character `'Z'`, leaving all other characters unchanged.

Example run, if the program is called `replacer`:

```
> echo "All sunshine and rainbows :)" > data.txt
> replacer data.txt
> cat data.txt
All sunshine Znd rZinbows :)
```

You may *not* read in more than one character from the file at a time, and in particular do not try to read in the whole file as a single string, as it may be extremely large. You may not use any other files. We are looking for how you use multiple reads, writes, and seeks with the file to accomplish this task by changing the existing file.

You *must* use binary I/O commands in your program.

Do not write any include statements or perform error-checking; you may assume that the name of a readable and writable text file is always given as the argument.

In the previous program, you were instructed that you must use binary I/O. Could the program have been written using character I/O statements? (check one) ☐ Yes ☐ No

If you answered yes, in the box below write *one* binary I/O statement that you used in your code and below it, write the character I/O statement that would need to replace it if you were to change your solution to use character I/O. (You may feel that there are multiple statements that need to change. Just show us one.)

If you answered no, explain why the problem could not be solved using character I/O.

Question 5. [11 MARKS]

Part (a) [3 MARKS] Consider the following simple program.

```
int main() {
    if (fork() == 0) {
        printf("[CHILD] I'm exiting!\n");
    } else {
        wait(NULL);
        printf("[PARENT] Child exited\n");
    }
    return 0;
}
```

David says, "Because the parent code comes after the child code, the entire child code will execute before any of the parent code. But then the child will exit before the parent calls wait, which means wait will block indefinitely, causing the program to never terminate." There are multiple false claims in what David said. Identify all of them and explain why they are wrong, and then explain what actually happens when this program is run.

Part (b) [2 MARKS]

Here is an incorrect attempt to communicate between a child and parent process.

```
int main() {
    int data;
    if (fork() == 0) {
        data = 10;
    } else {
        wait(NULL);
        printf("%d should equal 10\n", data); // Child has set data and exited.
    }
    exit(0);
}
```

Explain what the problem is, and what happens when the program runs.

Part (c) [3 MARKS]

We have seen in lecture how to use `execl` to change the behaviour of a running process. Which of the following properties of a process **change** when execl is called? (Select all that apply.)

☐ the PID
☐ the open file-descriptors
☐ the set of instructions to execute
☐ the parent's PID
☐ the instruction pointer indicating which instruction to execute next
☐ the values of the variables already in memory

Part (d) [3 MARKS]

We have also seen how to use `fork` to create a new process. Which of the following properties are **different** between the parent and child processes right after fork returns, but before its return value is assigned to a variable? (Select all that apply.)

☐ the PID
☐ the open file-descriptors
☐ the set of instructions to execute
☐ the parent's PID
☐ the instruction pointer indicating which instruction to execute next
☐ the values of the variables already in memory

Question 6. [6 marks]

Write a funtion `convert` whose first parameter is an array of strings and second parameter is the number of elements in this array. Your function should return a string that is the reverse of a longest string in the array (any longest will do in the case of a tie) or NULL if the array was empty. You must not change the strings in the array. You may assume all the strings in the array are null-terminated.

Question 7. [11 MARKS]

In this question you will write a program that forks processes that communicate with signals.

Your program must fork two children. One child sends a SIGUSR1 signal to the other child approximately every second (use `sleep(1)` between sending the signals). The other child does nothing except print numbers to standard out from 1 to 10000. But every time a SIGUSR1 signal arrives, it prints `"quit poking me"` to standard error. When its counting is finished, this child exits with the number of times it was poked as the exit code.

Once the child receiving the pokes has exited, the parent must report the number of pokes (by printing a message to standard output) and then kill the child doing the poking.

Part (a) [1 MARK]

Select the true statement below and then explain your reasoning.

☐ The child that does the poking must be forked first.
☐ The child that does the poking must be forked second.
☐ It doesn't matter in which order the children are forked; the program could be written to work in either way.

Explanation:

Part (b) [10 MARKS]

Write your program here and on the next page. Do not write the include statements or include error checking on your system calls. You **must write comments** so that the marker can clearly see what you are trying to do.

Question 8. [5 MARKS]

These questions concern the **client-server model** we discussed in lecture.

Part (a) [1 MARK]

Explain the role of the **server** in the client-server model.

Part (b) [1 MARK]

Explain the role of the **client** in the client-server model.

Part (c) [1 MARK]

What is the advantage of using the client-server model over simply forking processes and using pipes?

Part (d) [1 MARK]

Why do we need to specify a port number in the server process?

Part (e) [1 MARK]

The server uses the `accept` system call to establish a connection with a client. The prototype of the system call is

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

It both takes in a file descriptor and returns a file descriptor. What is the purpose of the file descriptor it returns; i.e., how is it used by the server process?

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

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

Total Marks = 75

C function prototypes and structs:

```
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);
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 */
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 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 Makefile variables:

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

Useful shell commands:

```
cat, cut, echo, ls, read, sort, uniq, set
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
$?	Exit value of previously executed command