UNIVERSITY OF TORONTO
Faculty of Arts and Science

AUGUST 2006 EXAMINATIONS

CSC 209 H1 Y

Duration — 3 hours

Examination Aids: One letter sized (8.5" × 11") sheet of paper
No electronic aids allowed: no cell phones, calculators, computers, etc.

Student Number:	
Last (Family) Name(s):	**SOLUTIONS**
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 term test consists of 8 questions on 14 pages (including this one). The last page includes C function prototypes and `sh` shell usage notes that might be useful: you may detach and keep this sheet. *When you receive the signal to start, please make sure that your copy of the test is complete.*

Answer each question directly on the test paper, in the space provided. If you need more space for one of your solutions, *indicate clearly the part of your work that should be marked and which question it answers.*

You do not need to include the "`#!`" line in Bourne shell scripts you are asked to write. In C programs, you do not need to add the "`#include`" lines, nor do error checking unless the question requires it or the program would not function correctly given valid input without error checking.

Though you should attempt to write programs using correct syntax, *minor* syntactical errors will not be penalized.

Marking Guide

# 1:	______/10
# 2:	______/16
# 3:	______/ 8
# 4:	______/ 8
# 5:	______/ 6
# 6:	______/10
# 7:	______/ 8
# 8:	______/22
TOTAL:	______/88

Good Luck!

Question 1. True or false. [10 MARKS]

Part (a) [5 MARKS]

Circle the correct answer for the questions below (1 mark each).

$\boxed{\text{TRUE}}$	FALSE	A *shell* is a user's interface to the Unix system.
$\boxed{\text{TRUE}}$	FALSE	The Unix operating system (kernel) manages access to the computer's hardware.
$\boxed{\text{TRUE}}$	FALSE	*System function calls* are a C program's interface to the Unix kernel and the computer's hardware.
TRUE	$\boxed{\text{FALSE}}$	Finding the length of a string in C is a good example of a *system function call*.
TRUE	$\boxed{\text{FALSE}}$	The term "Standard C Library" commonly refers to a set of 5 essential reference books that explain how to use the C programming language.

Part (b) [5 MARKS]

Circle the correct answer for the questions below (1 mark each).

$\boxed{\text{TRUE}}$	FALSE	In C, with the declaration `int a[10]`, the type of `a` is `int *`.
$\boxed{\text{TRUE}}$	FALSE	The `fork()` function (or a function that calls `fork()`) is the only way a new process can be created by a C program.
TRUE	$\boxed{\text{FALSE}}$	The `exec` family of functions allow us to run another program temporarily and return back to running our original program.
$\boxed{\text{TRUE}}$	FALSE	Unrelated processes can communicate with each other using signals.
$\boxed{\text{TRUE}}$	FALSE	Sockets allow programs on different machines to communicate with each other.

Question 2. Short answer. [16 MARKS]

Part (a) [2 MARKS]

Write a Bourne shell (`sh`) command that outputs the product of 209 and 369. Use as few processes as possible.

```
expr 209 \* 369
```

- −1 marks for correct answer using extra processes (eg., backquotes, echo)

Part (b) [2 MARKS]

Write a Bourne shell (`sh`) command that outputs the number of different settings of default shells being used on CDF. HINT: the default shell is the 7th "`:`"-delimited field in `/etc/passwd`. The programs `sort` and `uniq` may also be useful.

```
cut -d : -f 7 /etc/passwd | sort | uniq | wc -l
```

Part (c) [2 MARKS]

Write C statement(s) to allocate space for an array of 5 signed integers such that the space is automatically deallocated when it goes out of scope.

```
int a[5];
```

Part (d) [2 MARKS]

Write C statement(s) to allocate space for an array of n signed integers, where n is a variable of type `int`.

```
int *p = malloc(n*sizeof(int));
```

Part (e) [2 MARKS]

Write an equivalent expression in C for the expression "`*(a + 3) = 10;`" where `a` is declared as `int a[5];`.

```
a[3] = 10;
```

Question 2. (CONTINUED)

Part (f) [3 MARKS]

Suppose I have two processes communicating over a pair of pipes (one pipe for each direction). Process A sends a character to process B using `fprintf(outfd, "%c", c)` which process B reads with `fscanf(infd, "%c", &c)`. Process B should then reply with a character sent to process A using similar code. The pipes and file descriptors are correctly set up.

The problem is that they both stop running in the middle of this code and never proceed to or complete the "reply" phase. What could be an explanation for this problem?

SAMPLE SOLUTION

Buffering: perhaps the `fprintf()` output is being line or block buffered and is never being sent to the kernel for output to the pipe.

Describe briefly in English one way to fix this problem (you do not have to write the code).

There are many possible solutions:

- Use low-level I/O functions like `write()` that don't use buffering
- Flush the `outfd` buffer using `fflush()` following the `fprintf()` call
- Turn off buffering on the pipe file pointers using `setbuf()`

MARKING SCHEME:

- 2 marks for identifying a reasonable/correct possible problem
- 1 mark for identifying a correct way to correct the mentioned problem (even if the problem doesn't apply here)

Part (g) [3 MARKS]

Write the body of a C function that takes a string `s` and a character `c` as arguments and returns true if `c` appears in the string `s`, and returns false otherwise. Do not use any other functions (no library functions).

```
int strhas(const char *s, char c) {
```

SAMPLE SOLUTION

```
int strhas(const char *s, char c) {
  while(*s)
    if (*s == c)
      return 1;
  return 0;
}
```

MARKING SCHEME:

- 1 mark: correctly processing string
- 1 mark: correctly checking for `c` in string
- 1 mark: correctly returning true (nonzero) and false (zero)
- minus half for using library functions

Question 3. [8 MARKS]

Some programs change their behaviour based on the program name with which they were called. For example, if the `grep` program is executed using the program name `egrep`, it allows extended regular expressions instead of the traditional regular expressions (`egrep` may be a hard or soft link to `grep`). Another example is `bash`, to which `sh` is typically linked. If `bash` is executed using the name `sh` it operates in a stripped-down mode (like the real old `sh` program).

Write a Bourne shell (`sh`) script called `chameleon` that changes its behaviour based on how it's called. If your script is invoked with the name `foo` (*i.e.*, the user used the command line "`foo`" which ran your script), you should call the shell function `do_foo`. If your script is invoked with the name `bar`, you should call the shell function `do_bar`. Assume these functions exist (do not write them!). Otherwise (when invoked with any other name), attempt to run the program of the same name in either the `/bin` or `/usr/bin` directory (you may assume `chameleon` is not located in these directories). In all cases, pass *all* command line arguments on to the new program or shell function.

HINTS: Set your path appropriately. Recall that `basename` strips the directory from filenames.

SAMPLE SOLUTION

```
#!/bin/sh

prog=`basename $0`

if [ "$prog" = "foo" ]; then
    do_foo "$@"
elif [ "$prog" = "bar" ]; then
    do_bar "$@"
else
    PATH=/bin:/usr/bin
    "$prog" "$@"
fi
```

MARKING SCHEME:

- 1 mark: calling basename appropriately
- 2 marks: if checks to foo and bar
- 1 mark: running `do_foo` and `do_bar`
- 2 marks: executing default case
- 2 marks: sending arguments to called function/program
- deductions for failure to properly quote

Question 4. [8 MARKS]

Write a Bourne shell (`sh`) script called `termall` that attempts to terminate all processes with process ID between 1 and 10000. (Of course, it will only succeed for processes owned by the current user.) The `SIGTERM` signal should be used.

HINT: Don't kill yourself too early!

SAMPLE SOLUTION

```
#!/bin/sh

pid=1

while [ $pid != 10000 ]; do
    if [ $pid != $$ ]; then
        kill -s TERM $pid
    fi
    pid=`expr $pid + 1`
done

if [ $$ -le 10000 ]; then
    kill -s TERM $$
fi
```

(Note that kill defaults to sending SIGTERM if no signal specified.)

MARKING SCHEME:

- 3 marks: doing loop from 1 to 10000 correctly (test, expr, etc.)
- 2 marks: killing processes inside loop
- 1 mark: clear attept to use TERM signal (using TERM or SIGTERM is fine)
- 2 marks: killing self at appropriate time (0 marks here if could be killed inside loop)

Question 5. [6 MARKS]

Write the body of the `strunixtonet()` C function below. This function should write the contents of the string `s` to the socket `fd`, translating the string from Unix line-ending format to network line-ending format, *i.e.*, converting `'\n'` characters to the `'\r\n'` combination. This function does not modify `s` and returns the number of bytes written to `fd`.

```
int strunixtonet(int fd, const char *s) {
```

Sample Solution

```
int strunixtonet(int fd, const char *s)
{
  int b = 0;

  for (; *s; s++) {
    if (*s == '\n') {
      write(fd, "\r\n", 2); /* assume success */
      b+=2;
    } else {
      write(fd, s, 1); /* assume success */
      b++;
    }
  }
  return b;
}
```

Marking Scheme:

- 2 marks: walking down s to string end, not changing contents of s
- 1 mark: writing to fd correctly
- 2 marks: replacing `\n` with `\r\n` correctly
- 1 mark: returning correct number of bytes

Question 6. [10 MARKS]

The "message of the day" (motd) is printed to a user after a successful login. CDF uses this to display lab information and remind users not to eat or drink in the labs. It is typically also used to display information about outages or changes to the systems.

Write a C program that will display the `/etc/motd` file without having a user log in. When a client connects to TCP port 17 (the "Quote of the Day" port), your program should send the contents of `/etc/motd` to the client (in network line-ending format as described in Question 5), then disconnect.

You may assume that the `/etc/motd` exists and is readable by all users. You may use the `strunixtonet` function from the previous question in your solution.

SAMPLE SOLUTION

```
#define MAXLEN 1024

int main()
{
  struct sockaddr_in my_addr;    /* my address information */
  char buf[MAXLEN];              /* read buffer for file to send */
  int listenfd, clientfd;
  FILE *motd;

  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(17);  /* set to TCP port 17 */
  my_addr.sin_addr.s_addr = INADDR_ANY;
  memset(&(my_addr.sin_zero), 0, 8);

  /* set up TCP server; error checking not required */
  listenfd = socket(PF_INET, SOCK_STREAM, 0);
  bind(listenfd, (struct sockaddr *)&my_addr,
       sizeof(struct sockaddr));
  listen(listenfd, 1);  /* 1 backlog is plenty here */

  while (1) {
    clientfd = accept(listenfd, NULL, NULL); /* don't care about client address */
    /* client is connected, send the file */
    /* (reopens each time in case file changes) */
    motd = fopen("/etc/motd", "r");
    while (fgets(buf, MAXLEN, motd)) {
      strunixtonet(clientfd, buf);
    }
    fclose(motd);
    close(clientfd);
  }

  return 0;
}
```

Marking Scheme:

- 3 marks: setting up server correctly (socket, bind, listen, accept; small errors or omissions in setting up `my_addr` should not be penalized)
- 1 mark: using `htons(17)` for port setting
- 2 marks: correctly accepting and handling multiple connections (in serial)
- 3 marks: correctly reading and echoing `/etc/motd` file using network line-endings (using `strunixtonet` or not)
- 1 mark: closing any file descriptors that if left open would use up all available fds (i.e., client fds, any files opened in the loop)

Question 7. [8 MARKS]

Suppose we have a program `x` that only takes input from standard input. If we want the input to come from a file instead, we could easily use shell I/O redirection. The problem comes when we want `x` to run on several files: we would need to write a loop to run `x` with input redirected from each file.

Write a Bourne shell (`sh`) script called `wrap` that takes a program, any switches to that program (each starting with a `-`), and a list of files (none start with a `-`) to iteratively run on. For example,

```
wrap x -z a b c
```

would run `x -z` three times, with input reading from `a`, `b` and `c`.

HINT: The command "`expr -z : -`" prints `1`.

SAMPLE SOLUTION

```
#!/bin/sh

exec="$1"
shift
while [ `expr "$1" : -` = "1" ]; do
    exec="$exec $1"
    shift
done

while [ "$1" ]; do
    $exec < "$1"
    shift
done
```

MARKING SCHEME:

- 2 marks: appropriate use of quoting
- 1 mark: `$1` is name of program to run
- 2 marks: grabbing program options and correctly composing executing command line
- 1 mark: processing filenames correctly
- 2 marks: correctly executing program for each filename

Question 8. [22 MARKS]

You will write a program in C named `cat2` that takes two command lines as arguments, executes both commands in parallel, and combines the output of each to standard output (in a particular way, described now). Whenever a byte of output is available from one program but not the other, it is printed to stdout. Whenever bytes are available from both programs, a byte from the first program is printed followed by a byte from the second program.

For example, if program `a` prints 5 a's and program `b` prints 5 b's, `cat2 "a" "b"` might output `aaabababbb`.

You may assume you are given two command lines as `argv[1]` and `argv[2]`, but you may not assume that a command line can be executed successfully. If one command line cannot be executed, the behaviour of `cat2` should be identical to the `cat` command.

Part (a) [4 MARKS]

Write a high-level overview of your algorithm. Mention which system calls you use in each step. For example, your algorithm might begin with the following line:

```
    create two pipes
```

Part (b) [3 MARKS]

Assume you are given a function `char **break_into_args(char *s)` that takes a command line string and returns a pointer to an `argv` array representing this command line.

Write the C code the child process should execute to run the command line passed in as `s`. Assume all file descriptors are already set up for you. Remember that the `execvp` call is not guaranteed to succeed!

```
void do_child(char *s) {
```

SAMPLE SOLUTION

```
    char **args = break_into_args(s);
    execvp(args[0], args);
    perror("exec");
    exit(1);
}
```

Question 8. (CONTINUED)

Part (c) [7 MARKS]

Write the C code for the first part of the main function. You should not do any reading or writing, but should do everything up to that point. The child processes should execute the do_child function with an appropriate argument at the appropriate time. Make sure you close any unused file descriptors.

You may use the comment "/* now do likewise for second child */" to replace the second fork call and reduce how much you must write.

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

SAMPLE SOLUTION

```
int pa[2], pb[2];

pipe(pa);
pipe(pb);

if (!fork()) {
  /* child a */
  close(pb[0]);
  close(pb[1]);
  close(pa[0]);
  dup2(pa[1], STDOUT_FILENO);
  do_child(argv[1]);
}
if (!fork()) {
  /* child b */
  close(pa[0]);
  close(pa[1]);
  close(pb[0]);
  dup2(pb[1], STDOUT_FILENO);
  do_child(argv[2]);
}
close(pa[1]);
close(pb[1]);
```

MARKING SCHEME:

- 1 mark: calling pipe appropriately
- 1 mark: declaring variables correctly
- 1 mark: forking appropriately
- 1 mark: child closing unused pipe fds
- 1 mark: child calling dup2
- 1 mark: child calling do_child
- 1 mark: parent closing unused ends of pipes

```
/* postconditions: Child processes have been started, all unused file descriptors
 *   have been closed. Ready to start reading and echoing! */
```

Question 8. (continued)

Part (d) [8 marks]

Write the C code the parent will execute to read output from the child processes and output to stdout such that whenever a byte of output is available from one program but not the other, it is printed to stdout, and whenever bytes are available from both programs, a byte from the first program is printed followed by a byte from the second program. Hint: Use `read` to read one byte at a time.

```
/* preconditions: Child processes have been started, file descriptors are set up
 *   properly. Ready to read from fd's ________ and ________ to echo to stdout. */
```

Sample Solution

```
char c;
int numopen = 2;
int maxfdp1 = (pa[0] > pb[0] ? pa[0] : pb[0]) + 1;
fd_set allset, rset;
FD_ZERO(&allset);
FD_SET(pa[0], &allset);
FD_SET(pb[0], &allset);
while (numopen > 0) {
  rset = allset;
  select(maxfdp1, &rset, NULL, NULL, NULL);
  if (FD_ISSET(pa[0], &rset)) {
    if (read(pa[0], &c, 1))
      putchar(c);
    else {
      close(pa[0]);
      FD_CLR(pa[0], &allset);
      numopen--;
    }
  }
  if (FD_ISSET(pb[0], &rset)) {
    if (read(pb[0], &c, 1))
      putchar(c);
    else {
      close(pb[0]);
      FD_CLR(pb[0], &allset);
      numopen--;
    }
  }
}

return 0;
```

Marking Scheme:

- 1 mark: initializing `fd_set` appropriately
- 1 mark: setting maxfdp1 to larger fd + 1
- 1 mark: select call appropriate
- 1 mark: `FD_ISSET` test to read from only ready fds
- 1 mark: reading from child process
- 1 mark: removing fd from `fd_set` on EOF
- 2 mark: general correctness / reasonable approach / exiting from loop