UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2005 EXAMINATIONS

CSC 209 H1 F

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 10 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:	____/ 20
# 2:	____/ 10
# 3:	____/ 10
# 4:	____/ 8
# 5:	____/ 9
# 6:	____/ 7
# 7:	____/ 6
# 8:	____/ 10
# 9:	____/ 10
# 10:	____/ 10
TOTAL:	____/100

Good Luck!

Question 1. [20 MARKS]

Part (a) [5 MARKS]

Circle the correct answer below.

$\boxed{\text{TRUE}}$	FALSE	A file on a UNIX file system is considered to be hidden if its name starts with a period.
$\boxed{\text{TRUE}}$	FALSE	The file `.` always refers to the current directory.
TRUE	$\boxed{\text{FALSE}}$	A file must have *both* read and execute permission to be executed by the shell.
$\boxed{\text{TRUE}}$	FALSE	`PATH` is a shell environment variable listing the directories that should be searched to find a program whose location was not specified on the command line.
TRUE	$\boxed{\text{FALSE}}$	The shell always looks first in the current directory for a program whose location was not specified on the command line.

Part (b) [5 MARKS]

Circle the correct answer below.

TRUE	$\boxed{\text{FALSE}}$	C is a scripting language.
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.
$\boxed{\text{TRUE}}$	FALSE	In C, the value of a pointer variable is interpreted as an address in memory.
TRUE	$\boxed{\text{FALSE}}$	C automatically does error checking for us, so a C programmer does not need to check array bounds or worry about error conditions.
$\boxed{\text{TRUE}}$	FALSE	If `a` is declared as "`int a[10];`", the expression `a[5]` actually performs pointer arithmetic and is equivalent to `*(a+5)`.

Part (c) [5 MARKS]

Circle the correct answer below.

TRUE	$\boxed{\text{FALSE}}$	Pipes are a good choice for transferring data between two processes on different machines.
$\boxed{\text{TRUE}}$	FALSE	Sockets can be used to transfer data between two processes on the same machine.
TRUE	$\boxed{\text{FALSE}}$	Blocking a signal means the signal is thrown away and can never be received.
$\boxed{\text{TRUE}}$	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	$\boxed{\text{FALSE}}$	A memory leak occurs if a parent process exits without calling `wait()` or `waitpid()` for each of its children processes.

Part (d) [5 MARKS]

Suppose we write the following statements.

```
int a[5] = { 2, 4, 6, 8, 10 };
int *p = a;
```

True or false: the following C statements will compile without any warnings or errors.

$\boxed{\text{TRUE}}$	FALSE	`int x = p[4];`
$\boxed{\text{TRUE}}$	FALSE	`double d = a[2];`
$\boxed{\text{TRUE}}$	FALSE	`int y = a[a[0]] + a[0];`
TRUE	$\boxed{\text{FALSE}}$	`double *r = p;`
$\boxed{\text{TRUE}}$	FALSE	`int *s = &a[8];`

Question 2. [10 marks]

Write a C program (let's call it `q2`) that takes one command line argument that consists of a single character c, and echos its standard input to standard output, replacing each character c in the input with a space character.

For example, "`q2 i`" would replace every character "`i`" in its input with a space. Thus, if the input was "`this␣is␣the␣input`", the output would be "`th␣s␣␣s␣the␣␣nput`", where ␣ represents a space character.

You can assume your program is run correctly and you do not need to check for errors.

Sample Solution:

```
#include <stdio.h>

int main(int argc, char **argv)
{
  char r = argv[1][0];
  int ch;

  while ((ch = getchar()) != EOF) {
    if (ch == r)
      putchar(' ');
    else
      putchar(ch);
  }
  return 0;
}
```

Question 3. [10 MARKS]

Write a Bourne shell (`sh`) script that reproduces some of the functionality of the `which` command. `which` takes a list of names and looks through your path for the files which would be executed had these names been given as commands. For example, if we run `which cat dog`, we might get:

```
/bin/cat
dog: Command not found.
```

Recall that the `PATH` variable is a list of directories separated by colons. For example, `PATH` might be `/usr/local/bin:/bin:/usr/bin`. You may assume there are no empty fields in the `PATH`. The real `which` command looks in each directory in the path, in order left-to-right, checking if the specified name is a file with execute permission for the current user.

For this question, you may not use the `which` command, but you may use your program `q2` from question 2, if you wish. You may assume that the command line arguments do not contain any slashes.

SAMPLE SOLUTION:

```
#!/bin/sh

pathlist=`echo $PATH | ./q2 :`

# for each argument
for name
do
	found=
	for dir in $pathlist
	do
		if [ -z "$found" -a -x "$dir/$name" ]; then
			echo "$dir/$name"
			found=1
		fi
	done

	if [ -z "$found" ]; then
		echo $name: Command not found.
	fi
done
```

Question 4. [8 MARKS]

Part (a) [4 MARKS]

The `fgets()` function reads a line of input from a file, and stores the string *and* the newline character (if it is read) in the buffer. This can be annoying if we don't need the newline character, as we must remember to throw away before using the input.

Write the body of the `chomp()` function below. This function should remove the `'\n'` from the end of a string, if it exists.

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

SAMPLE SOLUTION:

```
int len = strlen(s);
if (s[len-1] == '\n') {
  s[len-1] = '\0';
}
```

```
}
```

Part (b) [2 MARKS]

Describe how executing the following lines of code could result in a crash. Recall that the `%s` format specifier for `scanf()` reads a string of non-white-space characters into an array of characters.

```
char s[10];
scanf("%s", s);
```

scanf can't know how big the array s is, so if it reads a long input word, it might overflow s and cause a segmentation fault

Part (c) [2 MARKS]

Rewrite the lines of code in part (b) to correct this error. Your code must still store input into string `s`.

SAMPLE SOLUTION:

```
char s[10];
scanf("%9s", s);
```

An answer using fgets() is also acceptable, but the code would have a different effect.

Question 5. [9 marks]

The following code was written by a professional C programmer. It appears in an important C program that was developed for a major company.

```
void* Realocate(void*buf, int os, int ns)
{

      void*temp;
      temp = malloc(os);
      memcpy((void*)temp, (void*)buf, os);
      free(buf);
      buf = malloc(ns);
      memset(buf, 0, ns);
      memcpy((void*)buf, (void*)temp, ns);
      return buf;

}
```

Part (a) [4 marks]

There are several serious errors or memory leaks in the above code. Identify 2 of them and explain why they are errors.

major errors (full points):

- temp is malloc'ed but not free'd — memory leak
- ns bytes are copied from temp, which only has os bytes... if ns > os, we overrun the array and may cause a segmentation fault
- the return values from the malloc calls are not checked before they are used.. if malloc failed for any reason, we'll crash

half point problems (not errors, but bad design):

- the first malloc/memcpy is unnecessary... we could just assign buf directly to temp
- os is unnecessary... malloc/free knows how much memory was originally allocated to the pointer

bad style, but doesn't count as an error:

- "Realocate" is spelled incorrectly

Part (b) [2 marks]

Is the `memset()` function call necessary? (The `memset(p,v,n)` function sets `n` bytes pointed to by `p` to value `v`.) Explain why or why not.

No. The next statement is a memcpy, overwriting all the memory we just set to zero. It is a waste of time.

Part (c) [2 marks]

Are the type casts in the `memcpy()` function calls necessary? Explain.

No. The memcpy function takes a void *, to which any pointer can be automatically converted.

Part (d) [1 MARK]

What C library function(s) should the programmer have used instead of writing this function full of errors?

```
realloc(buf, ns)
```

Question 6. [7 MARKS]

Consider the following body of a C program.

```
int main() {
  int i, nchildren = 0;
  pid_t pid;

  for (i = 1; i <= 2; i++) {
    pid = fork();
    if (pid > 0)
      nchildren++;
  }

  printf("I had %d children\n", nchildren);
  return nchildren;
}
```

Part (a) [2 MARKS]

How many *new* processes (in addition to the initial process) are created by this code?

Three

Part (b) [2 MARKS]

I executed this program once and generated the following output.

```
I had 1 children
I had 0 children
I had 2 children
I had 1 children
```

Adding the numbers suggests that 4 child processes were created. Explain why this differs from your answer for part (a).

One of the processes created in the second "round" was the child of the parent from the first "round." This process inherited the `nchildren` variable from the parent, with value 1, even though it has no children, explaining the "phantom" child.

Part (c) [3 MARKS]

Suppose the following line was added immediately before the `printf()` call.

```
  wait(NULL);
```

Could the output in part (b) be generated by the resulting program? Why or why not?

Yes. Suppose process A is the original process, process B is its first child (from the first "round"), process C is its second child (from the second "round"), and process D is the child of B (from the second "round").

Since C has no children, its wait call returns immediately with error (errno == ECHILD), and prints 1. D also has no child, so it also may return immediately, possibly scheduled after C and printing 0. Though A had 2 children, it only waits for one to terminate, and since C exited, A could now print 2. Finally, B's child already exited, so it can now print 1 and finish.

Question 7. [6 marks]

Often we want to find the definition of a symbol that is included in a C header file. The symbol might not be in the original file we included, but might be in some file it includes. For example, if we include `<signal.h>`, we can use the symbol `SIGCHLD`, but it's actually defined in `<bits/signum.h>`. If we were to look in `signal.h` for the definition of `SIGCHLD` we wouldn't find it; instead we must look in one of the header files that it includes.

Fortunately we wrote the system tool `printdepends` in Assignment 1 that can help us. Write a Bourne shell (`sh`) script that uses `printdepends` and `grep` to print all lines in all recursively included files that contain the given string. Your script will be executed with two command line arguments: the first is the string we want to find, the second is the file name of the C source file or header file to start with.

For example, if your script was named `q6`, we might execute it as:

`q6 SIGCHLD myprog.c`

You do not have to do any error checking, and you may assume that the file exists. Recall that `printdepends` prints on standard output the name of the C source file followed by a colon (i.e., "`myprog.c:`"), followed by a list of files that are (recursively) included by our C source file, one per line. Assume that `printdepends` prints any "file not found" messages on standard error. Recall also that the `-v` switch for `grep` inverts the sense of matching, causing all lines that do *not* match the specified regular expression to be printed.

Sample Solution:

```
#!/bin/sh
files=`printdepends $2 2>/dev/null | grep -v :$`
grep $1 $files
```

Question 8. [10 MARKS]

Part (a) [8 MARKS]

Complete the code for the following C function. This function takes two file descriptors that are associated with sockets (assume they are connected). This function should block until one of these file descriptors is ready for reading, then return the file descriptor that is ready (if both are ready for reading, either can be returned).

```
int readytoread (int fd1, int fd2)
{
```

SAMPLE SOLUTION:

```
  int max = fd1 > fd2 ? fd1 : fd2;
  fd_set rset;
  FD_ZERO(&rset);
  FD_SET(fd1, &rset);
  FD_SET(fd2, &rset);
  select(max+1, &rset, NULL, NULL, NULL);
  if (FD_ISSET(fd1, &rset))
    return fd1;
  else
    return fd2;

}
```

Part (b) [2 MARKS]

Describe how your code above needs to be changed (or whether it needs to be changed) if the two file descriptors are associated with the read ends of two pipes instead.

Our code does not need to be changed because `select` works on any type of file. This is the power of using a single interface (file descriptors) to access all sorts of byte streams.

Question 9. [10 marks]

Part (a) [7 marks]

Help! I've written a web server program that forks a new process to handle each client connection, but several pieces of my code have gotten mixed up! Put these pieces of code back into a correct order.

1. `while(1) {`

2. `pid = fork();`

3. `if (pid == 0) {`

4. `} /* end if */`

5. `} /* end while */`

6. `bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));`

7. `clientfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);`

8. `sockfd = socket(PF_INET, SOCK_STREAM, 0);`

9. `listen(sockfd, 5);`

10.
```
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(80);
my_addr.sin_addr.s_addr = INADDR_ANY;
```

11. `sin_size = sizeof(struct sockaddr_in);`

12.
```
readAndWriteToClient();
exit(0);
```

13. `close(sockfd);`

14. `close(clientfd);`

Correct order:

There are two correct orders:
10, 8, 6, 9, 1, 11, 7, 2, 3, 13, 12, 4, 14, 5
or
8, 10, 6, 9, 1, 11, 7, 2, 3, 13, 12, 4, 14, 5

Common errors:

- almost everyone put 11 too early. The `addrlen` argument to `accept` is "a value-result parameter: it should initially contain the size of the structure pointed to by addr; on return it will contain the actual length (in bytes) of the address returned." Thus it must be inside the loop (with accept).

- line 9 cannot be inside the loop. We only call `listen` once to set up the incomming connections queue.
- 13 should close the listening fd for the child process, so it should be in the if before line 12.
- 14 should close the client fd in the parent, who isn't ever going to talk to the client (that's the child's job), so it should be outside the if inside the loop.

Marking: -1 per error

Part (b) [3 MARKS]

Circle the above line numbers where I should add some error checking.

Lines 2 (or 3), 6, 7, 8, and 9 must be circled. 13, 14 and 12 may be circled.
Marking: -1 per error

Question 10. [10 marks]

Now that we've put the lines of my web server program in the correct order, I've compiled it and installed it at `/usr/local/sbin/mywebserver`. Unfortunately, it's a little buggy and keeps crashing with segmentation faults (`SIGSEGV`).

Write a C program that will invoke `/usr/local/sbin/mywebserver` (with no arguments) and restart it whenever it exits abnormally. After `/usr/local/sbin/mywebserver` exits abnormally, you should sleep for five seconds and then execute it again, and loop. If the server exits normally, your program should terminate.

Sample Solution:

```
int main()
{
  int status;
  while (1) {
    if (fork() == 0) {
      execl("/usr/local/sbin/mywebserver", "mywebserver", NULL);
    }
    wait(&status);
    if (WIFSIGNALED(status)) {
      sleep(5);
    } else {
      exit(0);
    }
  }
  return 1;  /* never executes */
}
```

There is also a fairly simple solution using signal handlers, but you need to use `fork` before you `exec` and you must catch the `SIGCHLD` signal (not the `SIGSEGV` signal!).

Some students restarted the webserver if `WIFEXITED(status) && WEXITSTATUS(status)` was true. An abnormal exit is usually understood to mean exit via signal, distinguished from a normal exit with error status (which is tested for by this expression). We accepted this on the exam though.

```
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 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);
int fgetc(FILE *stream);
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 *stream, const char *format, ...);
int getchar(void);
struct hostent *gethostbyname(const char *name);
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
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); /* O_RDONLY, O_WRONLY or O_RDWR */
DIR *opendir(const char *name);
int pclose(FILE *stream);
int pipe(int filedes[2]);
FILE *popen(char *cmdstr, char *mode);
int putchar(int c);
ssize_t read(int fd, void *buf, size_t count);
struct dirent *readdir(DIR *dir);
int select(int n, 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 is PF_UNIX, PF_INET; type is SOCK_STREAM, SOCK_DGRAM */
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);
char *strstr(const char *haystack, const char *needle);
int wait(int *status);
int waitpid(int pid, int *stat, int options); /* options = 0 or WNOHANG*/
ssize_t write(int fd, const void *buf, size_t count);

WIFEXITED(status)       WIFSIGNALED(status)     WIFSTOPPED(status)
WEXITSTATUS(status)     WTERMSIG(status)        WSTOPSIG(status)

struct hostent {
   char *h_name;     /* official 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 sigaction {
   void (*sa_handler)(int);   /* function or SIG_DFL or SIG_IGN */
   sigset_t sa_mask;
   int sa_flags;     /* SA_NOCLDSTOP, SA_RESTART, SA_NOMASK, etc. */
};

struct sockaddr_in {
   sa_family_t    sin_family; /* AF_INET */
   u_int16_t      sin_port;
   struct in_addr sin_addr;
   unsigned char  sin_zero[8]; /*Unused*/
};
```

Shell variables:

`$$` shell process ID
`$?` last program exit status
`$#` number of arguments
`$*` all arguments as string
`"$@"` all arguments as quoted list

Shell test 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, -ge, -lt, -le	Comparisons for numbers
!=, >, >=, <, <=	Comparisons for strings
-a, -o	And, or