PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL EXAMINATIONS 2007

CSC 209H1 S
St. George Campus

Duration — 3 hours

Examination aids: One 8.5 x 11 sheet of paper (double-sided)

Student Number: |_|_|_|_|_|_|_|_|_|_|_|

Last Name: SOLUTIONS

First Name: _____

Instructor: _____

---

*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.)

This examination consists of 9 questions on 18 pages (including this one). *When you receive the signal to start, please make sure that your copy of the examination is complete and fill in your student number on every page.* If you need more space for one of your solutions, use the last pages of the exam or the back of this page and *indicate clearly the part of your work that should be marked.*

1: _____/ 9

2: _____/ 6

3: _____/ 6

4: _____/ 7

5: _____/24

6: _____/ 6

7: _____/ 9

8: _____/11

9: _____/10

TOTAL: _____/88

Good Luck!

Question 1. [9 MARKS]

Part (a) [5 MARKS]

Circle the correct answer below.

TRUE	[FALSE]	A process whose parent has not called wait is an orphaned process.
[TRUE]	FALSE	A read on a pipe will block when there is nothing to ready to read on the pipe.
TRUE	[FALSE]	If a client's request for a socket connection arrives between the times that the server has called `listen` and `accept`, the connection will be denied.
TRUE	[FALSE]	If an struct is passed in as an argument in a C function, any modifications made to the contents of the struct will be visible when the function returns.
[TRUE]	FALSE	Read and execute permissions are needed on a shell program to execute it.

Part (b) [2 MARKS]

Fill in the third argument to `strncat`. Assume that `str1` and `str2` both contain valid strings.

```
char str1[SIZE1];
char str2[SIZE2];

strncat(str1, str2,_____SIZE1 - strlen(str1)_________);
// This could also be SIZE1 - strlen(str1) - 1
```

Part (c) [2 MARKS] Write the output of the following C snippet.

```
char a[10];
char *b = malloc(10 * sizeof(char));
printf("A = %d\n", sizeof(a));
printf("B = %d\n", sizeof(b));
```

10

4

Question 2. [6 MARKS]

The current working directory contains 3 files: `hop`, `skip`, and `jump`. The contents of each file are shown below:

`hop`

```
#!/bin/sh
arg=$1
echo "Running hop"
if [ -x $1 ]; then
      $1
fi
```

`skip`

```
#!/bin/sh
echo "Running skip"
```

`jump`

```
#!/bin/sh
echo "Running jump"
```

For each of the following commands, give the output that would appear when the command is run. (' is a single quote and ‘ is a back quote.)

`run="hop"` `$run`	`Running hop`
`echo "$run hop"`	`hop hop`
`echo '*'`	`*`
`echo ‘*‘`	`Running hop Running jump`
`[sh]*`	`Running hop Running skip`
`set ???p` `echo $2`	`skip`

Question 3. [6 MARKS]

Part (a) [2 MARKS]

I told my friend who has an account on CDF that I have a program she can run in `~reid/bin/prog`. When I run `ls -l ~reid/bin/prog` I see `-rwxr-xr-x` for the permissions on the file. When she runs `~reid/bin/prog` she gets "Permission denied." What could be the cause?

She doesn't have execute permissions on one of the directories on the path.

Part (b) [2 MARKS]

What are `htonl()` and `ntohl()` for? Why do we need to use them on the port number when creating a socket but not on character arrays that are read and written on the socket?

They convert data from the host byte order to network byte order and back again. We don't need to use them on character arrays because a character is one byte, so byte ordering doesn't change anything.

Part (c) [1 MARK]

What could go wrong if the reader of a pipe forgets to close `fd[1]`?

It won't know that the writer has closed the pipe and a read will block forever after the writer closes its end of the pipe.

Part (d) [1 MARK] Explain why the following code is incorrect (aside from the lack of error checking).

```
char line[10]
int fd[2];
int newfd;
pipe(fd);
newfd = fd[0];
read(newfd, line, 10);
```

Need to use dup2 to copy the file descriptor

Question 4. [7 MARKS]

Write a C program that takes a string and a directory name as arguments and prints the names of the files in that directory whose names end with the string. For example, if the program is called with `.html foo`, then it will print the names of all of the files in `foo` that end with `.html`. (Do not create another process.)

```
int
numdirs(char *dirname) {

  DIR *dir;
  struct dirent *dp;
  struct stat sbuf;
  int dircount = 0;
  int filecount = 0;
  char path[128];

  if((dir = opendir(dirname)) == NULL) {
      return -1;
  }

  while((dp = readdir(dir)) != NULL) {
     /* compare strings
     printf("Number of directories = %d\n", dircount);
     printf("Number of files = %d\n", filecount);
     return 0;
  }
}
```

Marking:

- 1 for argument handling
- 1 for opendir
- 1 for loop with readdir
- 1 for constructing the path for stat
- 2 for correctly comparing strings.
- 1 for printing

Question 5. [24 MARKS]

The code below is the start of a very simple implementation of a shell. The only functionality in this shell is the ability to read lines from standard input and execute a program in the background or in the foreground. Recall that when a program is executing in the foreground, the shell user cannot start another command until the program has terminated. If a program is started in the background by appending '&' to the command string, then the shell user can execute another command before the first one terminates.

You do not need to provide error handling for the given code.

Part (a) [9 MARKS] Complete the loop below according to the comments.

```
int main() {

    char line[30];
    char **args;
    char *ptr;
    int background;

    while(1) {
        printf (">");
        fgets(line, 30, stdin);
        ptr= strchr(line, '\n');
        *ptr = '\0';

        /* If the user typed return, then there is no command to execute */
        if(strlen(line) == 0) {
            continue;
        }

        /*mkarray returns an array of the words in line using space as a delimiter*/
        args = mkarray(line);

        /* If the user just typed return go back to the top of the loop */
        if(args[0] == NULL) {
            continue;
        }

        /*checkbackground returns 1 if the program should be run in the background
        and 0 otherwise */
        background = checkbackground(args);

        /* Run the program specified by args.  If the program is to be run
        in the foreground, then block until the process terminates and
        print a message if the program terminated normally, otherwise return
        to the top of the loop to wait for the user to enter another command.
        Check if any background programs have terminated, and print a message
        if they terminated normally. */
```

Part (a) (CONTINUED)

```
            if((pid = fork()) == -1) {
                perror("fork");
            } else if(pid == 0) { /* child */
                execvp(args[0], args);
                perror(args[0]);
                exit(1);
            } else {
                numkids++;

                if(background) {
                    for(i = 0 ; i < numkids; i++) {
                        int rpid;
                        if((rpid = waitpid(-1, &status, WNOHANG)) == -1) {
                            perror("waitpid");
                        } else if(rpid > 0) {
                            numkids--;
                            if(WIFEXITED(status)) {
                                printf("Process %d exited normally\n", rpid);
                            }
                        }
                    }
                } else {
                    if((pid = wait(&status)) == -1) {
                        perror("wait");
                    } else {
                        numkids--;
                        if(WIFEXITED(status)) {
                            printf("Process %d exited normally\n", pid);
                        }
                    }
                }
            }
        }
    }
```

Part (b) [8 MARKS]

Implement the function `char **mkarray(char *line)` that returns an array of the words in the string `line`, using a space as the delimiter.

```
char **mkarray(char *line) {
    char **strs = malloc(10 * sizeof(char *));
    char *start = line;
    char *ptr;
    int i = 0;

    while((ptr = strchr(start, ' ')) != NULL) {
        *ptr = '\0';
        strs[i] = start;
        start = ptr + 1;
        i++;
    }

    if(*start != '\0') {
        strs[i] = start;
        i++;
    }

    strs[i] = NULL;
    return strs;
}
```

Part (c) [3 MARKS]

Based on your implementation of `mkarray()`, is there any memory that should be freed explicitly outside of `mkarray()`? If so, show the code below that the calling process would run to free the correct memory. State where in the loop in part (a) this code to free memory should go.

```
free(args);
```

After fork is called, the parent should free args before the next loop iteration.

Part (d) [4 MARKS]

Write the function `int checkbackground(char **args)` that returns 1 if the program should be run in the background, and 0 otherwise. (Hint: the function should modify `args` so that the array contains only the function name and the actual program arguments.)

```
int checkbackground(char **args) {
    int size = 0;

    while(args[size] != NULL)
        size++;

    if(args[size - 1][0] == '&') {
        args[size - 1] = NULL;
        return 1;
    }
    return 0;
}
```

- 1 for finding the last element of args
- 1 for checking to see if it is '&'
- 1 for setting it to NULL
- 1 for returning the correct value.

Question 6. [6 MARKS]

Write a Bourne shell program called `makepath` that takes a pathname as a command line argument and creates all the components of that path if they don't already exist. For example,

```
makepath foo/bar/baz
```

should create the directories `foo`, `foo/bar`, and `foo/bar/baz`. You may not use `mkdir -p` in your script. You may find the following commands useful:

```
basename NAME - Print NAME with any leading directory components removed.
                (basename / produces '/')
dirname NAME - Print NAME with its trailing /component removed;
               if NAME contains no /'s print '.'  (dirname / produces '/')
```

The easy way to do this is to write a recursive function:

```
#!/bin/sh -x

function mkdir_rec() {
    base=`basename $1`

    if [ $base != "." -a $base != "/" ]
    then
        mkdir_rec `dirname $1`
        mkdir $1
    fi
}

mkdir_rec $1
```

Of course you can do it iteratively. Here is one solution; there are probably simpler ones.

```
#!/bin/sh -x

path=$1

dir=‘dirname $path‘
base=‘basename $path‘
list=" "

# create a list
while [ $base != "." -a $base != "/" ]
do
    list="$base $list"
    base=‘basename $dir‘
    dir=‘dirname $dir‘

    if [ $dir == "." ]
    then
        newpath="."
    else
        newpath=""
    fi
done

for d in $list
do
    mkdir $newpath/$d
    newpath=$newpath/$d
done
```

Question 7. [9 MARKS]

Part (a) [6 MARKS]

A CSC207 student needs some help automating tests. The student has written a program called `parsewiki` that takes a file name as a command line argument and writes to standard output. The files in the directory `tests` are to be used as test input to `parsewiki`. There is a directory called `expected` that contains identically named files with the expected output.

Write a Bourne shell program that runs `parsewiki` on each of the files in `tests`, places the output in file of the same name as the input file in a directory called `actual`, and then uses `diff` to compare the actual output file to the expected output file, and will print the message "Test failed on file X" (where X is replaced with the name of the file that failed) if the expected and actual output differ. No other output should be printed.

Note that `diff` returns 0 if the files are the same and 1 if they differ.

```
#!/bin/sh

for f in tests/*
do
    file=‘basename $f‘
    parsewiki $file > actual/$file
    diff actual/$file expected/$file > /dev/null
    result=$?

    if [ $result eq 1 ]
    then
        print Test on $file failed
    fi
done
```

There is some ambiguity in where the directories are.

Marking

- 1 for iterating over files in tests
- 1 for getting the name of the file
- 1 for running the program and redirecting the output
- 1 for running diff
- 1 for sending the output of diff to /dev/null
- 1 for checking the return value of diff and printing message

Part (b) [3 MARKS]

Write a Bourne shell program that prints the name of each file in `tests` that does not have a matching file in `expected`.

```
for f in tests/*
do
    file=‘basename $f‘
    if [ ! -f expected/$file]
    then
        echo $file
    fi
done
```

Marking

- 1 for iterating over files in tests
- 1 for getting the file name
- 1 for testing the existence of the file in expected

Question 8. [11 MARKS]

Consider the following program. Assume the program runs without error. The error handling has been removed to make the code shorter.

```
void sig_int(int signo) {
    write(2, "caught SIGINT\n", sizeof("caught SIGINT\n"));
    /*pause();*/ /* Uncomment this line for part (c) */
    /*F*/
    return;
}

int main(void) {
    sigset_t newmask, oldmask, pendmask;
    struct sigaction newact;

    sigemptyset(&newact.sa_mask);
    sigaddset(&newact.sa_mask, SIGTERM);
    sigaddset(&newact.sa_mask, SIGINT);
    newact.sa_handler= sig_int;
    newact.sa_flags = 0;
    /*A*/
    sigaction(SIGINT, &newact, NULL) ;

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGTERM);
    sigaddset(&newmask, SIGINT);

    /*B*/
    printf("Blocking signals\n");
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    /*C*/
    sigpending(&pendmask);
    if(sigismember(&pendmask, SIGTERM))
     printf("SIGTERM pending\n");
    if(sigismember(&pendmask, SIGINT))
     printf("SIGINT pending\n");

    /*D*/
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    printf("Unblocking signals\n");

    /*E*/
    printf("All Done\n");
    exit(0);
}
```

Part (a) [5 MARKS]

Describe what happens if the signal SIGINT arrives at the process at each of the letter markers A, B, D, E, and F. If any output is produced write it down, otherwise write "None." Assume the program is restarted each time.

	What happens	Output
A	Program terminates	None
B	call sig_int	caught SIGINT Blocking signals Unblocking signals All Done
D	continue executing until signals unblocked call sig_int and continue	Blocking signals SIGINT pending caught SIGINT Unblocking signals All done
E	call sig_int	Blocking signals Unblocking signals caught SIGINT All Done
F	sig_int will be called twice but will complete before being called again because SIGINT is blocked in sig_int	caught SIGINT caught SIGINT

Part (b) [2 MARKS]

Describe what happens if the signal SIGTERM arrives at the process a markers B and D. If any output is produced write it down, otherwise write "None." Assume the program is restarted each time.

	What happens	Output
B	Program terminates	None
D	Program terminates after sigprocmask is called Unblocking signals	Blocking signals

Part (c) [4 MARKS]

Suppose that the line with `pause()` is uncommented in the `sig_int()` function, and that SIGINT arrives such that `sig_int` is called. What happens if SIGTERM arrives after the program calls `pause()`?

The program actually terminates because the SIGTERM interrupts the pause. (DOUBLE CHECK. It may behave differently on CDF)

I would also accept the following answer (because that's what I thought it would do): The program will block. SIGTERM is blocked during the signal handling function, so it doesn't cause pause to return.

What if SIGQUIT arrives after the program calls `pause()`? (Assume the program is restarted and SIGINT arrives such that `sig_int()` is called.)

The pause will return and the program will terminate.

Question 9. [10 MARKS]

Complete the function below that takes an array of two open socket descriptors as an argument. The function implements a subtraction game where the players are given a number to start with. They each take turns subtracting 1, 2 or 3 from `number`. The person that causes the total to reach 0 wins.

In this implementation, the players write each move on the file descriptor as an `int`. You can assume that no other type of data is sent, and you do not need to verify that the number is in the correct range. You must use `select` to avoid blocking.

When a player wins, write a message to both players indicating that the game has been won.

When a player sends a message out of turn, write a message on the socket descriptor telling the player, "It's not your turn." If either player closes their socket, then write a message to the other player, close the sockets, and return.

```
void play(int fd[]) {
    int number = 21;
```

```
void play(int fd[]) {
int turn = 0;
int total = 21;
int value = 0;

// add to select set
fdset_t allset, read;
FD_CLR(&allset);
FD_SET(fd[0], &allset);
FD_SET(fd[1], &allset);
if(fd[0] > fd[1]) {
    maxfd = fd[0];
} else {
    maxfd = fd[1];
}

while(1) {
    read = allset;
    nready = select(maxfd, &read, NULL, NULL, timeout);

    for(i = 0; i < 2; i++) {
        if(FD_ISSET(i)) {
            int me = i;
            int you = (i + 1) % 2;
            if(turn = me) {
                if(read(&value, sizeof(int), fd[me]) > 0){
                    total -= value;
                    if(total <= 0) {
                        write("I win", 6 fd[you]);
```

```
                        write("I win", 6 fd[me]);
                        close(fd[you]); close(fd[me]);
                        exit(0);
                    }
                } else {
                    write("Game over", 10, fd[you]);
                    close(fd[you]);
                    exit(0);
                }
            } else {
                if(read(&value, sizeof(int), fd[me]) > 0) {
                    write("It's not your turn", 18, fd[me]);
                } else {
                    write("I quit", 7, fd[you]);
                    close(fd[you]);
                    exit(0);
                }
            }
        }
    }
}
```

Warning: this code was not run, so there are probably still errors in it.

Marking:

- set up set of fds
- set up of max fd
- calling select
- reading an int
- using FD_ISSET correctly for both fds
- checking for winning and writing message
- checking for closed socket (return value of read) and writing message
- checking whose turn it is
- checking when to exit the program (solution doesn't currently do this)
-

Total Marks = 88