PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL-MAY EXAMINATIONS 2001

CSC 209H1 S
Duration — 3 hours

Examination Aids: ***None***

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

Last Name: SOLUTIONS

First Name: _____

Lecture Section: _____

Lecturer: Reid

---

*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 final examination consists of 9 questions on 16 pages (including this one), *When you receive the signal to start, please make sure that your copy of the examination is complete.* Answer each question directly on the examination paper, in the space provided.

Be aware that concise, well thought-out answers will be rewarded over long rambling ones. Also, unreadable answers will be given zero so write legibly.

You do **not** need to include header files or do error checking in C programs except where specifically mentioned in a question. The last two pages of this exam contain a list of C function prototypes structs, and some Perl, Python and Bourne shell details.

1: _____/10

2: _____/ 6

3: _____/12

4: _____/ 8

5: _____/ 8

6: _____/14

7: _____/10

8: _____/12

9: _____/10

TOTAL: _____/90

Good Luck!

CONT'D...

Question 1. [10 MARKS]

Part (a) [2 MARKS]

In assignment 3, child processes wrote to the parent through pipes. Explain how the parent knew when it could start reading from the pipes.

The parent blocked in a read call until there was something to read.

Part (b) [2 MARKS]

Explain why is it important to close the ends of pipes when they are no longer needed. What problem can occur if the write end of a pipe is not closed?

The process reading from the pipe will not know when the writer has finished putting all the data into the pipe, and will block waiting for more.

Part (c) [2 MARKS]

Give one reason why a programmer would want to block a signal.

To allow a computation such as accessing a file or some other critical region code to complete before handling a signal. (It was not sufficient to say that we don't want to receive the signal.)

Part (d) [2 MARKS]

Give one reason why a programmer would want to alter the behaviour of a signal by installing a signal handling function.

- *Orderly shutdown - to ensure that global resources are properly cleaned up even when the program terminates abnormally.*
- *To print out or return some result before termination*

Part (e) [2 MARKS]

Explain what the zombie state of a process is used for.

To save the termination code of the child process so the parent can collect it when the parent calls wait. (It is not sufficient to give the definition of a zombie process.)

Question 2. [6 MARKS]

Part (a) [3 MARKS]

Scripting languages such as Perl and Python are frequently used in web programming. Explain why scripting languages might be preferred over C for this type of programming.

- *Perl and Python are particularly well-suited to text manipulation which is a major component of web programming.*
- *Many supporting libraries have been written in both languages.*
- *Because they are interpreted languages they are somewhat more portable.*
- *They are higher level languages which makes it easier to write small programs that communicate with each other.*

(It is not the case that Perl or Python code runs faster because you don't have to wait for it to compile.)

Part (b) [3 MARKS]

Given the following C program, circle **all** valid output sequences. Note that 0 or more of the sequences may be valid.

```
int main() {
    int num, status;
    num = 10;
    printf("A: %d\n", num);

    if( fork() == 0 ) {
        num = 24;
        printf("B: %d\n", num);
    } else {
        wait(&status);
    }
    printf("C: %d\n", num);
}
```

A: 10
B: 24
C: 24
C: 24

A: 10
B: 24
C: 24
C: 10

A: 10
B: 24
C: 10
C: 24

A: 10
B: 24
C: 10

A: 10
B: 24
C: 24

A: 10
C: 24
B: 24
C: 24

A: 10
C: 10
B: 24
C: 24

A: 10
C: 10
B: 24

A: 10
C: 24
B: 24

Question 3. [12 MARKS]

Part (a) [1 MARK]

Write a shell command that will concatenate all of the files in the current working directory that end in .h and put them into a file called `headers`.

```
cat *.h > headers
```

Part (b) [2 MARKS]

Consider the following two shell programs:

testsbs

```
#!/bin/sh
if [ -f .sbs ]
then
        echo found
        exit 0
fi
exit 1
```

runtests

```
#!/bin/sh
# ‘ is a backquote

result=‘testsbs‘
```

When `runtests` is executed, the value of `$result` if there *is* a file called .sbs in the current working directory is

found

When `runtests` is executed, the value of `$result` if there *is not* a file called .sbs in the current working directory is

$*result is empty*

Part (c) [5 MARKS]

If the contents of the current working directory are

```
a.c a.o bb qq
```

and the following Bourne shell commands are executed, what is printed? (‘ is a backquote and ’ is a single quote.)

```
cmd="ls *"
set ‘$cmd‘
```

`echo $cmd`	ls a.c a.o bb qq
`echo $1`	a.c
`echo ??`	bb qq
`echo '??'`	??
`echo "$cmd"`	ls *

Part (d) [4 MARKS]

Assuming that a Perl or Python program is stored in a file called `getvotes`, describe the two different ways the program could be run. Give any conditions necessary to allow the program to be run, and explain what the user must type.

1. *Run as “perl getvotes” or “python getvotes”*
2. *Put a* `#!` *line at the beginning of the file, make it executable and run the program as “getvotes”.*

Question 4. [8 MARKS]

Given the following Makefile and the contents of the current working directory:

```
CFLAGS = -g -Wall
OBJS = main.o a.o b.o

default: main

main : $(OBJS)
        gcc $(CFLAGS) -o main $(OBJS)

a.o: a.c
        gcc -c a.c
b.o: b.c
        gcc -c b.c
main.o: main.c
        gcc -c main.c
```

```
-rw-r--r--  reid 177 Mar 25 11:24 Makefile
-rw-r--r--  reid 118 Mar 25 11:06 a.c
-rw-r--r--  reid  17 Mar 25 11:06 a.h
-rw-r--r--  reid  72 Mar 25 11:06 b.c
-rw-r--r--  reid  32 Mar 25 11:06 b.h
-rw-r--r--  reid 137 Mar 25 11:08 main.c
```

Part (a) [2 MARKS]

Write the commands that would be executed when the user types `make`.

```
gcc -c main.c
gcc -c a.c
gcc -c b.c
gcc -g -Wall -o main main.o a.o b.o
```

Part (b) [2 MARKS]

Suppose the user then modifies `main.c`, and then types `make`. Write the commands that are executed.

```
gcc -c main.c
gcc -g -Wall -o main main.o a.o b.o
```

Part (c) [2 MARKS]

Write an additional rule with the target name `clean` that removes all `.o` files and the executable `main`.

```
clean:
        rm $(OBJS) main
```

Part (d) [2 MARKS]

If the user modifies `a.h` and then calls `make`, nothing will be recompiled. Change or add a rule so that `a.o` will be re-generated (recompiled) when `a.h` is modified.

```
a.o : a.c a.h
        gcc -c a.c
```

(There are other possibilities.)

Question 5. [8 MARKS]

Write a Perl or Python program that takes as a command-line argument the name of a file containing lists of candidates and votes for each of the candidates. Your program will print the total number of votes cast, the number of votes for each candidate, and the percentage of the total vote each candidate received. (Don't worry about the significant digits.)

An input file and resulting output are given as an example:

Input file:

Jimmy Neutron: Boy Genius	15
Monsters, Inc.	100
Shrek	150
Monsters, Inc.	200
Shrek	102
Jimmy Neutron: Boy Genius	80
Shrek	88

Output:

```
Total votes: 735
Shrek 340 46.25%
Monsters, Inc. 300 40.81%
Jimmy Neutron: Boy Genius 95 12.92%
```

Perl

```
#!/local/bin/perl

$filename = $ARGV[0];

open(VOTES, "<$filename") or die "Couldn't open $filename\n";

my $first = 1;
my %candidates;

while( $line = <VOTES>) {
  chomp($line);
  if($line =~ /([\w\s]+\w)\s+(\d+)$/) {
    $totalvotes += $2;
    $candidates{$1} += $2;
  }
}

print "Total vote = $totalvotes\n";

foreach $c (keys(%candidates)) {
  $percentage = $candidates{$c} / $totalvotes * 100;
  print "$c $candidates{$c} $percentage\n";
}
```


CONT'D...

Python

```
#!/local/bin/python
import sys
import re

f = open(sys.argv[1], "r");

pat="(.*)\s+(\d+)$"
candidates = {}
totalvotes = 0

for line in f.readlines() :
    m = re.search(pat, line)
    if m :
        name, votes = m.group(1), m.group(2)
        if candidates.has_key(name) :
            candidates[name] += int(votes)
        else:
            candidates[name] = int(votes)

        totalvotes += int(votes)

print "Total votes:", totalvotes
for c in candidates.keys() :
    percentage = candidates[c] * 100 / float(totalvotes)
    print c, " ", candidates[c], " ",  percentage
```

Question 6. [14 MARKS]

Part (a) [2 MARKS]

In assignment 3, the child processes all accessed the same file. Explain why we did not need to use a semaphore to protect access to the file.

The processes only read from the file so they would not corrupt the file. As long as each process opened the file itself, it did not share the pointer to the current location in the file.

Part (b) [4 MARKS]

The Unix pipe system call sets up a pipe data structure in the kernel and synchronizes access to the pipe. There are three aspects of pipes that require some form of synchronization. One need for synchronization is to ensure that only one process at a time may access the pipe data structure. Describe the other two situations where synchronization is needed to ensure the correct operation of the pipe.

1. *Synchronization is needed to ensure that no one reads from an empty pipe.*
2. *Synchronization is needed to ensure that no one writes to a full pipe*

Part (c) [8 MARKS]

Complete the following pseudo-code functions to show how semaphores may be used to implement the synchronization necessary for the correct operation of pipes. You may assume the existence of the following functions:

`InitSem(int S, int value)`	Initialize the semaphore variable `S` to `value`
`acquire(int S)`	Perform the acquire or wait operation on the semaphore variable `S`.
`release(int S)`	Perform the release or signal operation on the semaphore variable `S`.
`read(buf)`	Read from some shared data structure into the buffer `buf`
`write(buf)`	Write to some shared data structure from the buffer `buf`

```
CreatePipe(int pipesize) {
    int mutex    /* semaphore variable */
    int full, empty /* Solution */
    InitSem(full, pipesize) /*Solution */
    InitSem(empty, 0) /*Solution */
    InitSem(mutex, 1)
}


ReadFromPipe(buf) {
    acquire(empty) /*Solution */
    acquire(mutex)
    read(buf)
    release(mutex)
    release(full) /*Solution */
}

WriteToPipe(buf) {
    acquire(full) /*Solution */
    acquire(mutex)
    write(buf)
    release(mutex);
    release(empty); /*Solution */
}
```

Question 7. [10 MARKS]

The following code is part of a program where a maximum of 3 clients connect to a server. The server prompts each client to enter a sequence of names and numbers. Clients execute a loop in which they read a prompt, write a name, read a prompt, and write a number. One problem with the program is the the server does not detect when a client closes a socket. However, there is a more serious problem with the implementation of this program. Assume that all of the set up and initialization is correct.

```
for ( ; ; ) {
    rset = allset;      /* structure assignment */
    nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

    if (FD_ISSET(listenfd, &rset)) {    /* new client connection */
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
        printf("accepted a new client\n");

        Writen(connfd, "Enter a name", 13);
        for (i = 0; i < 3; i++)
            if (client[i] < 0) {
                client[i] = connfd; /* save descriptor */
                break;
            }
        if (i == 3) printf("too many clients");

        FD_SET(connfd, &allset);            /* add new descriptor to set */
        if (connfd > maxfd) maxfd = connfd; /* for select */
    }

    for (i = 0; i < 3; i++) {   /* check all clients for data */
        if (FD_ISSET(client[i], &rset)) {
            n = Readline(client[i], line, MAXLINE);
            printf ("Client name = %s\n", line);
            Writen(client[i], "Enter a number", 15);
            n = Readline(client[i], line, MAXLINE);
           printf ("Number = %d\n", atoi(line));
        }
    }
}
```

Part (a) [2 MARKS]

The major flaw of this program could cause it to hang (block) indefinitely. Explain what the problem is and under what circumstances it could cause the program to hang.

The second Readline will block if the client does not send the second message.

Part (b) [8 MARKS]

Fix the program so that it will not block unnecessarily. You do not need to rewrite all of the code, just indicate where the portions of the code you write belong in the provided code and cross out any code that should be deleted.

```
\* initialize a flag array to keep track of whether we are reading a name
or a number */
int readname[3];
for(i = 0; i < 3; i++) {
   readname[i] = 1;
}


/* Replace the last for loop */
for(i = 0; i < 3; i++) {
    if (FD_ISSET(client[i], &rset)) {
        n = Readline(client[i], line, MAXLINE);
        printf("Client name = %s\n", line);
        Writen(client[i], "Enter a number", 15);
        readname[i] = 0;
    } else {
        n = Readline(client[i], line, MAXLINE);
        printf("Number = %d\n", atoi(line);
        Writen(client[i], "Enter a name", 13);
        readname[i] = 1;
    }
}
```

Question 8. [12 MARKS]

Part (a) [2 MARKS]

Name the two TCP socket functions that are used to initiate and complete the 3-way handshake.

connect and accept

Part (b) [2 MARKS]

What is the purpose of the 3-way handshake?

To establish the connection between the client and the server, so either side is able to tell if all packets have been received.

Part (c) [8 MARKS]

Complete the following socket **client** program that establishes a connection to a remote server given by `hostname` on the port `SERVER_PORT`. The server sends the client the address of another server to connect to and a message to write to that server and then closes the connection.

The client reads two messages from the first server. The first message is the address of the second server in network byte order. The length of the second message is between 1 and `BLOCK_SIZE` bytes. The client makes a connection to the second server using the address sent from the first server and the same port, and writes to the second server the bytes it received from the original server and closes the connection.

You must ensure that the client closes the socket, even when an error occurs. Recall that `htons()` must be called to convert a port into network byte order, and that the socket type will be `AF_INET`.

```
#define SERVER_PORT  30000
#define BLOCK_SIZE 1024

char *hostname = "penguin";

int main(void) {
    struct hostent *hp = gethostbyname(hostname);
    struct in_addr address;
```

```
    int i, n;    FILE *fp;    int soc;    char buf[BLOCK_SIZE];
    struct sockaddr_in peer;

    peer.sin_family = AF_INET;
    peer.sin_port = htons(SERVER_PORT);
    peer.sin_addr = *((struct in_addr *)hp->h_addr);


    soc = socket(AF_INET, SOCK_STREAM, 0);
    if (connect(soc, (struct sockaddr *)&peer, sizeof(peer)) == -1) {
        perror("client:connect"); close(soc);
        exit(1);
    }
```


CONT'D...

```
    Readn(soc, &address, sizeof(struct in_addr));
    n = Readn(soc, buf, BLOCK_SIZE);
    close(soc);

    peer.sin_addr = address;
    soc = socket(AF_INET, SOCK_STREAM, 0);
    if (connect(soc, (struct sockaddr *)&peer, sizeof(peer)) == -1) {
        perror("client:connect"); close(soc);
        exit(1);
    }

    Writen(soc, buf, n);
}
```

(An extra page for the previous question.)

Question 9. [10 MARKS]

When creating programs to automatically test student assignments, we want to detect if a student's program gets into an infinite loop. Complete the C program below that takes the name of a second program to run as a command line argument. Your program will create a process to execute the second program (which itself takes no arguments). If the second program does not terminate after 10 seconds, the original process will terminate it. Your program will print out an error message if it must terminate the second process.

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

    int pid;
    int status;

    if((pid = fork()) == 0 ) {
        execl(argv[1], argv[1], 0);

    } else {
        sleep(10);
        if(waitpid(pid, &status, WNOHANG) == 0) {
            kill(pid, SIGKILL);
            printf("Probable infinite loop\n");
        }
    }
}
```