Name: **Student Number:**

Lab Test 2014

Prescription Number: **ENCE360**

Lab Test Course Title: ***Operating Systems***

Time allowed: 90 minutes

Number of pages: 15

Question	Marks
1	
2	
3	
Total:	

- This exam is worth a total of 20 marks
- Contribution to final grade: 20 %
- Length: 3 questions
- Use this *exam paper* for answering *all* questions. If you need more space, use a separate *Answer Booklet* provided.
- *This is an open book test.* Notes, text books and online resources may be used.
- This open book test is supervised as a University of Canterbury exam. Therefore you cannot communicate with anyone other than the supervisors during the test. Anyone using email or other forms of communication with others will be removed and score zero for this test.
- Please answer *all* questions carefully and to the point. Check carefully the number of marks allocated to each question. This suggests the degree of detail required in each answer, and the amount of time you should spend on the question.

1 Instructions

First reboot your PC (to minimise problems during the test)
Now, write your name and student number at the top of the front page of this paper to avoid scoring 0**.**
This test is open book. You are given 90 minutes. To answer each question you need to write the answer on this exam paper. If you need extra space, you may request an answer book.
To help with your written answers, you may edit the files supplied to you. The examiner will not look at the edited files. Your source code files are not marked.
Only this written test paper is marked.
So your source code files are NOT marked.
There are three questions, worth a total of 20 marks.

1.1 Preparation

Log in with your normal user code. At the beginning of the test, the source code files for this test will be available from Learn.
Before the test begins you may download the files from Learn and check that the files are available, you may not view or edit them until the test has begun.
You should now have the following files:

- "2014 lab test one.pdf" – this lab test handout
- `one.c` – source code for question one
- `two.c` – source code for question two
- `threeServer.c, threeClient.c` – source code for question three
- `escape.h`

If you do not have all these files, then call over an exam supervisor promptly.
These source code files are not marked.

1.2 Comments and code layout

Your source code answers (hand-written on this exam paper) should always be neatly laid out and **commented to make it clear to the examiner that you understand the code and concepts**.

This test paper has semi-complete source code for which your task is to fill in all the gaps (empty boxes). Add the minimum code possible – so do not add error checking code. (You are being tested on your understanding of concepts and usage of functions, not error checking skills.)

There are a total of FOUR MARKS across all three questions for commenting in your written answers. So comment almost every line of the code – both the supplied code and your added lines of code.

Separate source code files are also supplied for each question. But any commenting in these separate source code files is not examined.
Only your answers in this written paper are marked.
Your edited source code files are NOT marked.

Do not turn to the next page until instructed.

Question 1: Threads 1 (5 marks)

In the program below we are concurrently using an interface for making withdrawals from a bank account, however we ran into some problems when we try to access the account from many threads. We think that it may be fixable using a semaphore as a lock.

Your task for this question is to identify two problems with the concurrent code below and fix them by completing the code below so the transactions finish correctly as expected (final balance = 0) without adding any delays .
Use as few lines of code as possible (e.g. no error checking) and comment almost every line of code, including both existing and new lines of code (1 of the 5 marks is for commenting).

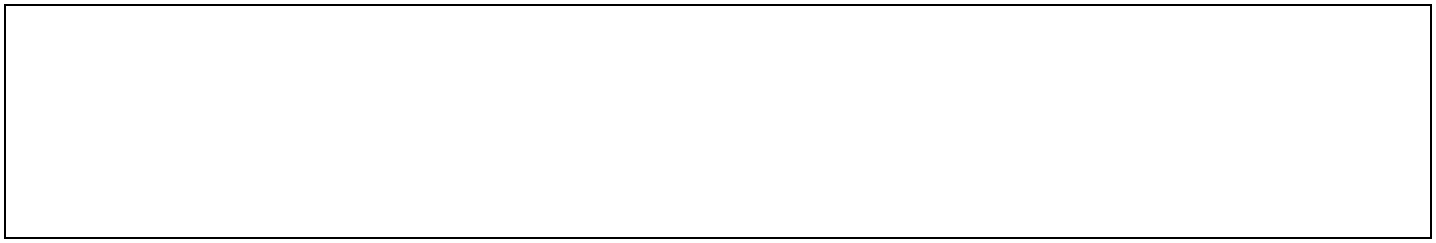
Source code is in **one.c**
Compile one.c using: **`gcc -o one one.c -lpthread`**

---

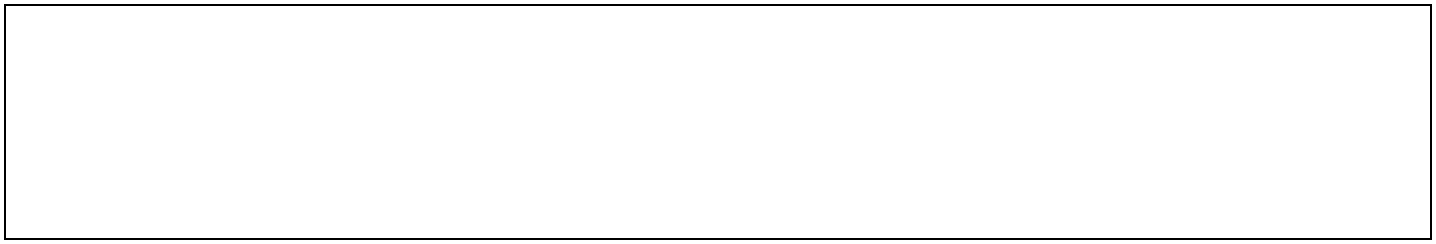
one.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_THREADS 8

pthread_mutex_t mutex;

int balance;

// Bank account interface
int getBalance() {
 return balance;
}

int setBalance(int b) {
 balance = b;
}
```

```
void *make_withdrawal( void *ptr )
{
```



```
  int amount = getBalance();
  printf("Balance before: %d dollars \n", amount);
  amount = amount - 10;
  setBalance(amount);
```



```
  printf("Balance after: %d dollars \n", amount);
}
```

```
int main()
{
  int i = 0;
  pthread_t thread[NUM_THREADS];

  // Start with 100 dollars
  setBalance(100);

  // Initialize mutex
  pthread_mutex_init(&mutex, NULL);

  // Create a bunch of threads to perform some withdrawals
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_create( &thread[i], NULL, make_withdrawal, NULL);
  }

  // Wait for all the threads we created
```

```
  printf("Final balance: %d dollars \n", getBalance());

      return 0;
}
```

Briefly describe the two things wrong with this program before you had added/modified any code:

Question 2: Pipes (5 Marks)

Here, two processes are communicating both ways using two pipes.
The parent process reads input from a user and sends this to a child process using an unnamed pipe.
The child then sends this text back to the parent using a named pipe.

Your task for this question is to complete the code below (in the empty boxes) using as few lines of code as possible (e.g. no error checking). Comment almost every line of code; including both existing and new lines of code (1 of the 5 marks is for commenting).

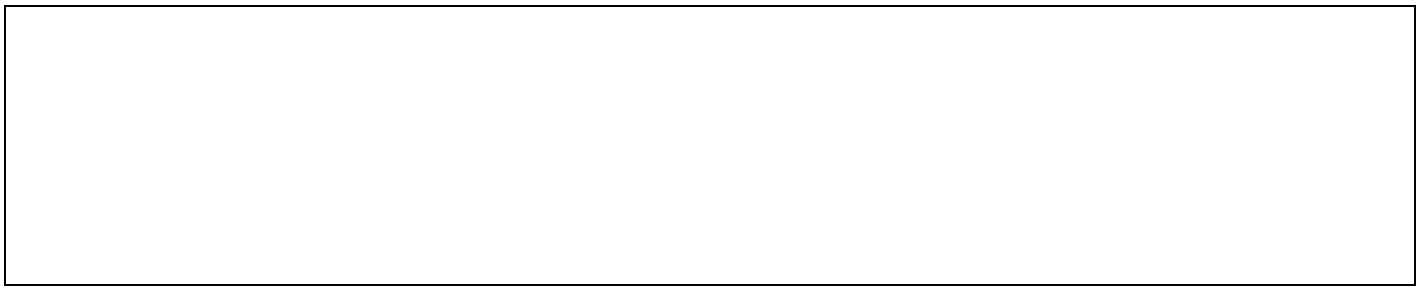
Source code is in **two.c**
Compile two.c using: `gcc two.c -o two`

---

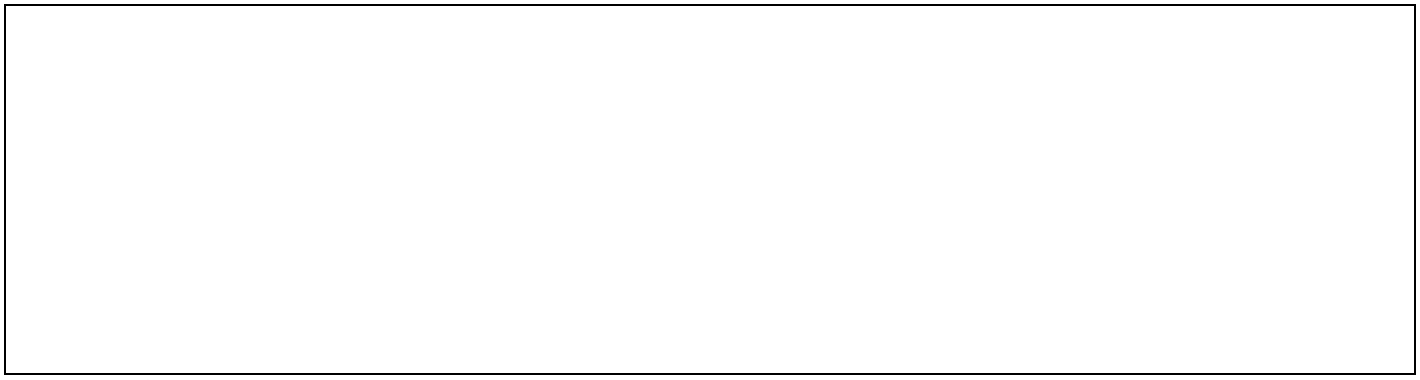
two.c:

```
#include <stdio.h>    /* standard I/O routines */
#include <stdlib.h>   /* standard C funtions   */
#include <string.h>   /* for strlen()          */
#include <pthread.h>  /* thread functions      */
#include <fcntl.h>     /* for S_IFIFO in mknod() and O_RDONLY, O_WRONLY in open() */

#define BUFSIZE 80
const char *named_pipe_filename = "./named_pipe";


int main()
{
  int pid;
  int status; /* status of child process */
  char parent_buffer[BUFSIZE] = {0}; /* unnamed pipe buffer */
  char child_buffer[BUFSIZE] = {0};  /* unnamed pipe buffer */
  char named_pipe_buffer[BUFSIZE] = {0};
  FILE* fp1; /* file pointer for a named pipe (used in the child process)  */
  FILE* fp2; /* file pointer for a named pipe (used in the parent process) */
  int parent_to_child_pipe[2]; /* file descriptor for an unnamed pipe      */

  umask(0);
  mknod(named_pipe_filename, S_IFIFO | 0666, 0); /* create a named pipe */

  pipe(parent_to_child_pipe); /* create an unnamed pipe */
```

```
pid = fork(); /* then fork off a child process */

switch (pid) {

  case 0: /* inside child process  */

    /* GET DATA FROM PARENT USING AN UNNAMED PIPE: */
```

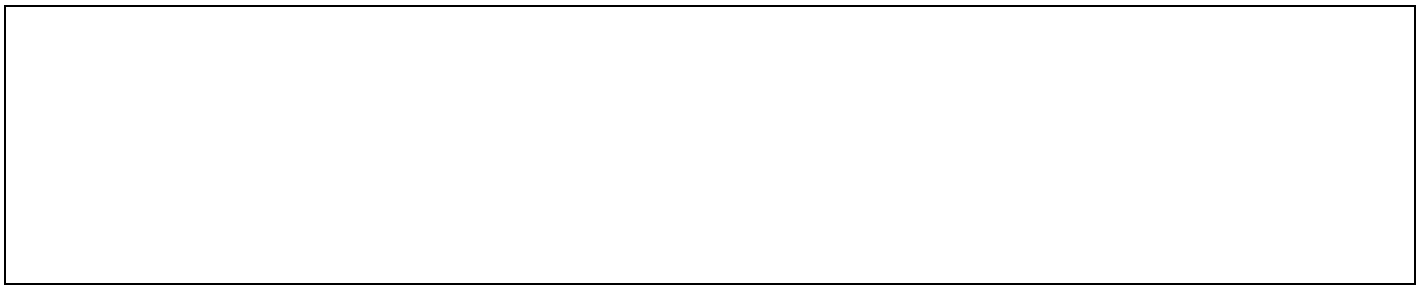


```
    printf("\nChild received %i unnamed-piped characters as: %s\n",
    (int)strlen(child_buffer),child_buffer);

    /* SEND DATA BACK TO PARENT USING A NAMED PIPE: */
```

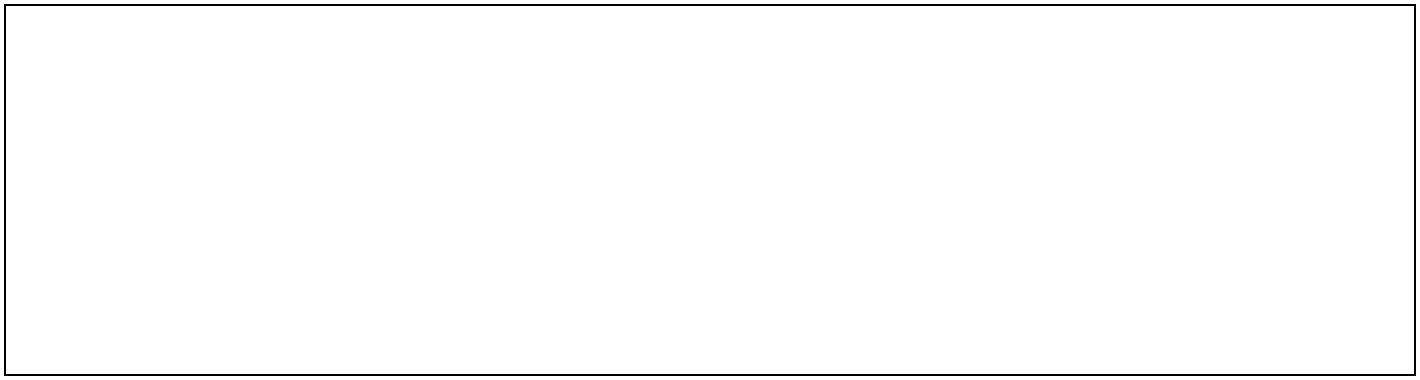


```
    exit(0);
```

```
    default: /* inside parent process */

      printf("\nEnter text to pipe: ");
      scanf("%s", parent_buffer);/* enter text (up to a space) */

      /* SEND DATA TO CHILD USING AN UNNAMED PIPE: */
```



```
      /* GET DATA BACK FROM CHILD USING A NAMED PIPE: */
```



```
      printf("\nParent received %i named-piped characters as: %s\n\n",
      (int)strlen(named_pipe_buffer),named_pipe_buffer);
      wait(&status); /* wait for child process to end */
  }

  return 0;
}
```

Question 3: Sockets (10 Marks)

Below is the code for socket server and client programs for checking remote directory listings. You are provided with a client program for connecting and making queries to this server program. The client and server code are incomplete. They should be able to accept connections, before handing off each request to a **new process** in order that it can process many queries **concurrently**. You should use the system **ls -l** executable in order to process requests.

Your task for this question is to complete the code below (in the empty boxes) using as few lines of code as possible (e.g. no error checking). Comment almost every line of code, including both existing and new lines of code (2 of the 10 marks are for commenting).

Source code files are in **threeClient.c** and **threeServer.c**
Compile threeServer.c using: `gcc threeServer.c -o threeServer`
Compile threeClient.c using: `gcc threeClient.c -o threeClient`
To run:

- open two terminals
- in one terminal type: **threeServer 1234**
- in the other terminal type: **threeClient localhost 1234 /**

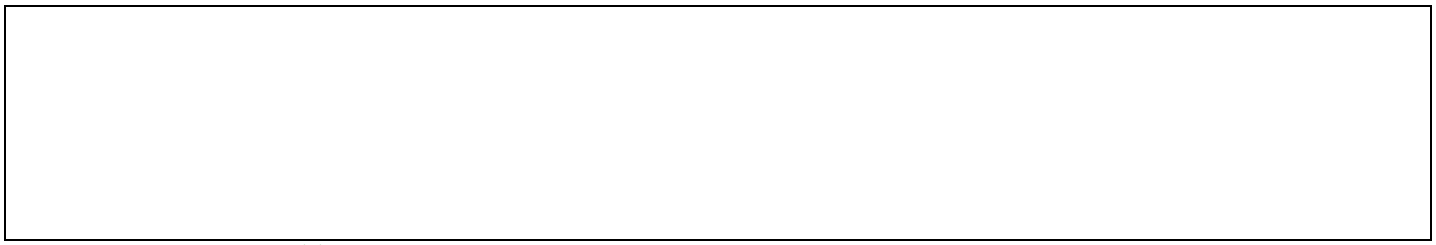
(Note: always run threeServer before threeClient)

---

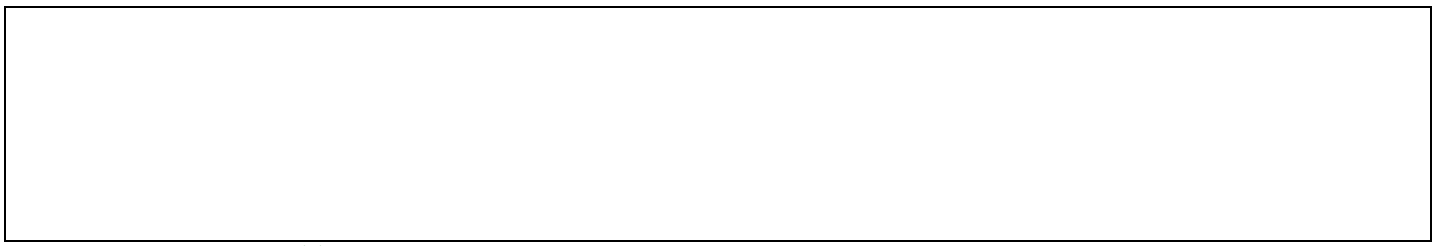
threeServer.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <errno.h>
#include <unistd.h>

#include "escape.h"

#define MAXDATASIZE 1024 // max buffer size
```

```
int listen_on(char *port)
{
  int s = socket(AF_INET,SOCK_STREAM,0);

  struct sockaddr_in sa;
  sa.sin_family = AF_INET;          /* communicate using internet address */
  sa.sin_addr.s_addr = INADDR_ANY; /* accept all calls                    */
  sa.sin_port = htons(atoi(port)); /* this is port number                 */
```

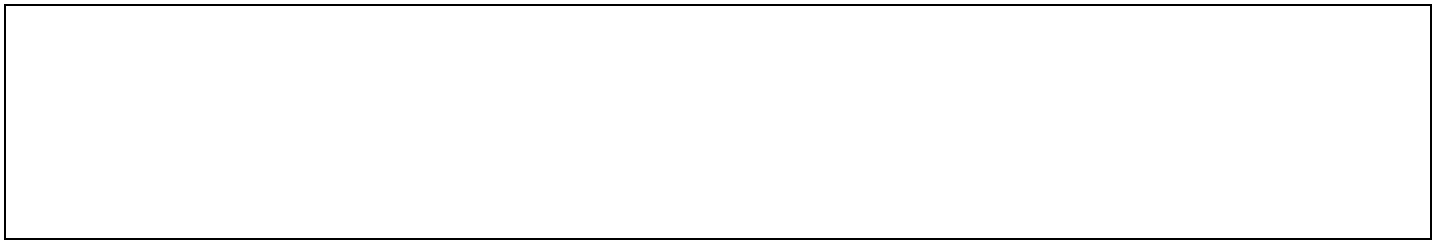


```
  if(rc == -1) { // Check for errors
    perror("bind");
    exit(1);
  }
```

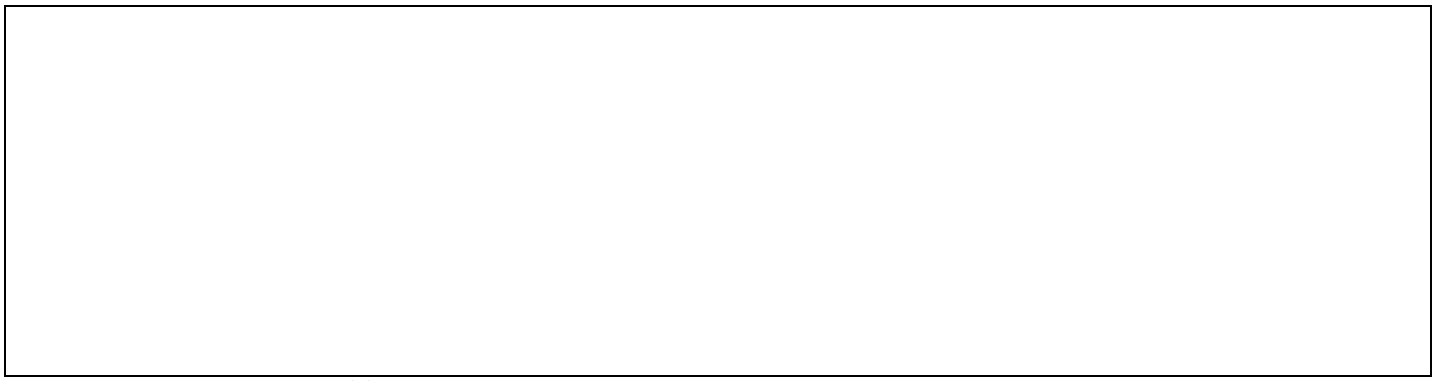


```
  if(rc == -1) { // Check for errors
    perror("listen");
    exit(1);
  }

  return s;
}
```

```
// wait for call to socket, and then return new socket
int accept_connection(int s) {

  struct sockaddr_in caller;
  int length = sizeof(caller);
```



```
  return msgsock;
}



// handle request by forking a new process
int handle_request(int msgsock, char *dir) {

  if(fork() == 0) {
    // in child process: redirect stdout to the socket and ls to the socket
```

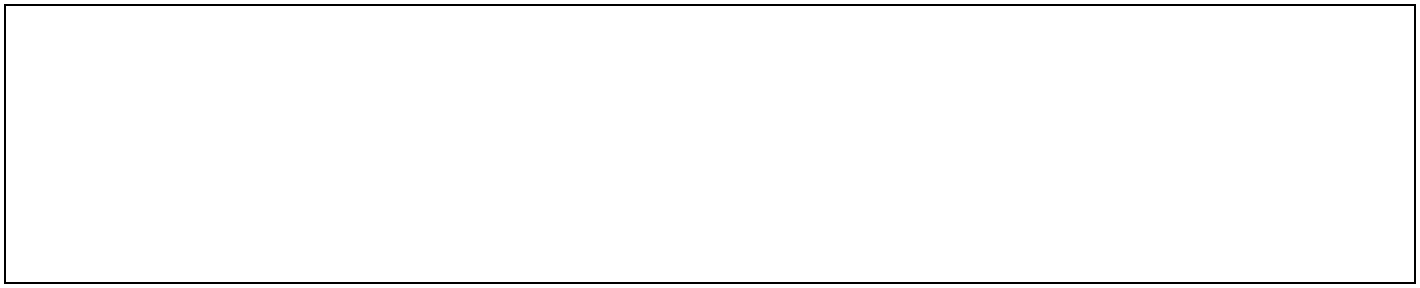


```
    close(msgsock); // close the socket (done)
    exit(0);  // exit child process
  }

    else {
    // in parent process
    // close socket, otherwise the client doesn't finish
    close(msgsock);
  }
}
```

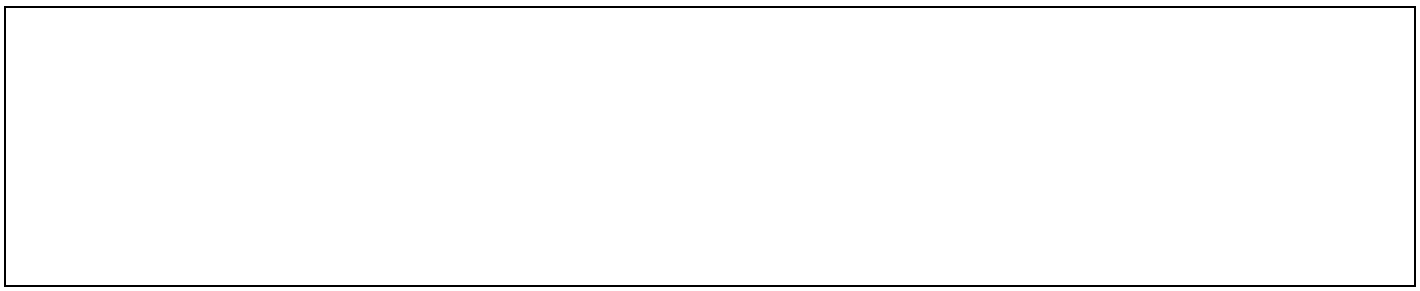
```
int main(int argc, char *argv[])  {
  char buffer[MAXDATASIZE];
  char dir[MAXDATASIZE];

  if (argc != 2) {
    fprintf(stderr,"usage: threeServer port-number\n");
    exit(1);
  }

  printf("\nThis is the server with pid %d listening on port %s\n", getpid(), argv[1]);

  // setup the server to bind and listen on a port
  int s = listen_on(argv[1]);

  while(1) { // forever

    int msgsock = accept_connection(s); // wait for a client to connect
    read (msgsock, buffer, MAXDATASIZE - 1);  //read the directory listing request

    printf("\nListing directory request: \"%s\"\n", buffer);

    // escape dangerous strings
    escape(dir, MAXDATASIZE, buffer);

    // handle the request with a new process
    handle_request(msgsock, dir);

  }

  close(s);
  exit (0);
}
```

threeClient.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>

#define MAXDATASIZE 1024 // max buffer size

int client_socket(char *hostname, char *port)
{
```

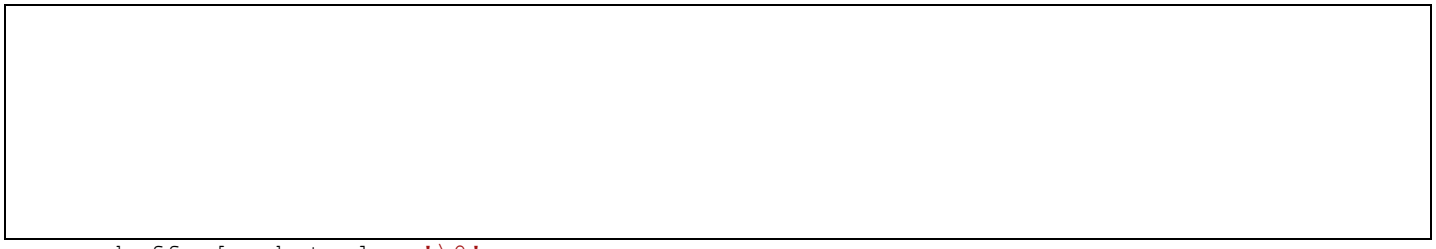


```
  struct addrinfo their_addrinfo; // server address info
  struct addrinfo *their_addr = NULL; // connector's address information
  memset(&their_addrinfo, 0, sizeof(struct addrinfo));
  their_addrinfo.ai_family = AF_INET;         /* use an internet address */
  their_addrinfo.ai_socktype = SOCK_STREAM;  /* use TCP rather than datagram */
  getaddrinfo( hostname, port, &their_addrinfo, &their_addr);/* get IP info */
```



```
  if(rc == -1) {
    perror("connect");
    exit(1);
  }

  return sockfd;
}
```

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

    if (argc != 4) {
      fprintf(stderr,"usage: threeClient hostname port-number directory\n");
      exit(1);
    }

    int sockfd = client_socket(argv[1], argv[2]);

    write(sockfd, argv[3], strlen(argv[3]) + 1);


    int numbytes = 0;
    do {
      // read from the socket:
```



```
      buffer[numbytes] = '\0';

      printf("%s", buffer);

    } while(numbytes > 0);

    printf("\n");
    close(sockfd);

    return 0;
}
```

END OF PAPER