UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Lab Test 2023

Prescription Number: **ENCE360**

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

Time allowed: 90 minutes

- This exam is worth a total of 20 marks
- Contribution to final grade: 20 %
- Length: 3 questions
- For answering *all* questions, write your answers in the source code files - and upload to Learn.
- *This is an open book test (but not online).* Printed notes, text books and files from your COSC Linux Home directory, Learn and Quiz Server may be used.
- This open book test is supervised as a University of Canterbury exam. So you cannot communicate with anyone other than the supervisors during the test. Anyone using any form 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.

Instructions

For submitting your answers to this lab test, write your answers in the source code files and upload them to Learn during this test.

Preparation

Login with your normal student usercode to the computer and CodeBox will automatically start.

You have access to Learn and access to files on the Quiz Server (e.g. solutions to all labs) - but no other online sites.

To access files in your COSC Linux Home directory, open a terminal window and type ***coschomedir*** and follow the prompts.

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 "2023 lab test.zip" from Learn and check that the following files are available – but you may not read or edit them until the test has begun (other than these first three pages). You should now have the following files:

- "2023 lab test.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

If you do not have all these files, then call over an exam supervisor promptly.

Comments and code layout

Your source code answers should always be **commented to make it clear to the examiner that you understand the code and concepts**.

This test has semi-complete source code for which your task is to fill in all the gaps, looking like: ////////////// 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 answers. So comment almost every line of the code – both the supplied code and your added lines of code.

Just a reminder that for submitting your answers to this lab test, write your answers in the source code files and upload them to Learn during this test.

Do not turn to the next page until instructed.

Question 1: Threads 1 (4 marks)

If the main parent thread creates a child thread and then this main parent thread finishes before its child thread ends, does this automatically kill the child thread – or does that child thread continue regardless of when the main parent thread ends?
This is the question that one.c seeks to answer by initially having a one second delay in the main parent thread, during which time you **print "Hello World"** inside the child thread. Then you introduce a **five second delay** in the child thread, before printing in the child thread and observe the difference.
(Hint: there is a big difference between waiting 5 seconds versus waiting 1 second for a result.)

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 4 marks is for commenting.

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

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

void *print_message_function( void *ptr );

int main()
{
```

```

```

```
        sleep(1);
        return(0);
}

void *print_message_function( void *ptr )
{
        printf("child thread\n");
```

```

```

```
        pthread_exit(NULL);
}
```

Run the program, with and without a 5 second delay in the child thread - write down the results displayed and then answer the question:
"If the main parent thread finishes before a child thread, does that child thread still continue?"
//… Note: write down the results displayed at the end of your code file (in commented out space) …

```

```

Question 2: Pipes (6 Marks)

Here, two processes are communicating both ways by using two pipes. One process reads input from the user and handles it. The other process performs some translation of the input and hands it back to the first process for printing.

Your task for this question is to complete the code below 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 6 marks is for commenting.

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

```c
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>


void translator(int input_pipe[], int output_pipe[])
{
    int c;
    char ch;
    int rc;

    /* first, close unnecessary file descriptors */
```

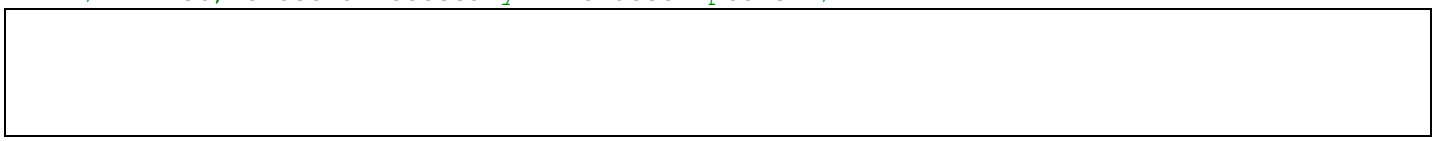
```
```

```c
    // enter a loop of reading from the user_handler's pipe one character at a time
    //                 translate this character and send it back to the user handler
    while (read(input_pipe[0], &ch, 1) > 0) {
        c = ch;
        if (isascii(c) && isupper(c))
            c = tolower(c);

        ch = c;
        /* send translated character back to user_handler via a pipe */
```

```
```

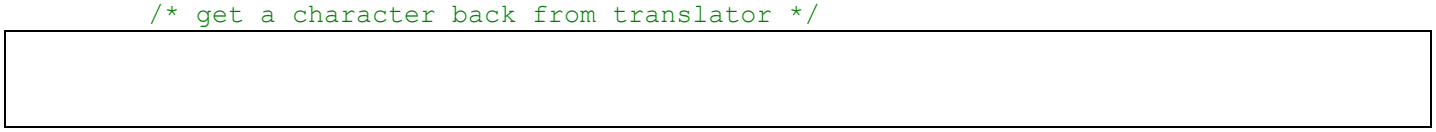
```c
        if (rc == -1)
        {
            perror("translator: write");
            close(input_pipe[0]);
            close(output_pipe[1]);
            exit(1);
        }
    }

    close(input_pipe[0]);
    close(output_pipe[1]);
    exit(0);
}
```

```
void user_handler(int input_pipe[], int output_pipe[])
{
    int c;
    char ch;
    int rc;

    /* first, close unnecessary file descriptors */
```



```
    printf("Enter text to translate:\n");

    // loop: get input from user one character at a time
    //       send via one pipe to the translator one character at a time
    //       read via other pipe what the translator returned, one character at a time
    //       send to stdout one character at a time
    //       exit on EOF from user
    while ((c = getchar()) > 0) {
        ch = (char)c;

        /* send a character to translator via a pipe */
```

```
        if (rc == -1) {
            perror("user_handler: write");
            close(input_pipe[0]);
            close(output_pipe[1]);
            exit(1);
        }

        /* get a character back from translator */
```



```
        c = (int)ch;
        if (rc <= 0) {
            perror("user_handler: read");
            close(input_pipe[0]);
            close(output_pipe[1]);
            exit(1);
        }

        putchar(c);
        if (c=='\n' || c==EOF) break;
    }

    close(input_pipe[0]);
    close(output_pipe[1]);
    exit(0);
}
```

```
int main(int argc, char* argv[])
{
    int user_to_translator[2];
    int translator_to_user[2];
    int pid;
    int rc;

    rc = pipe(user_to_translator);
    if (rc == -1) {
        perror("main: pipe user_to_translator");
        exit(1);
    }

    rc = pipe(translator_to_user);
    if (rc == -1) {
        perror("main: pipe translator_to_user");
        exit(1);
    }


    pid = fork();

    switch (pid) {

        case -1:
            perror("main: fork");
            exit(1);

        case 0:
            translator(user_to_translator, translator_to_user);
            exit(0);

        default:
            user_handler(translator_to_user, user_to_translator);
    }

    return 0;
}
```

// Show below the output for the following input: I wish YOU a HAPPY new YEAR

//… write down the results displayed at the end of your code file (in commented out space) …

Question 3: Sockets (10 Marks)

Below is the code for a socket server program and a socket client program.

Your task for this question is to complete the code below 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 **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**

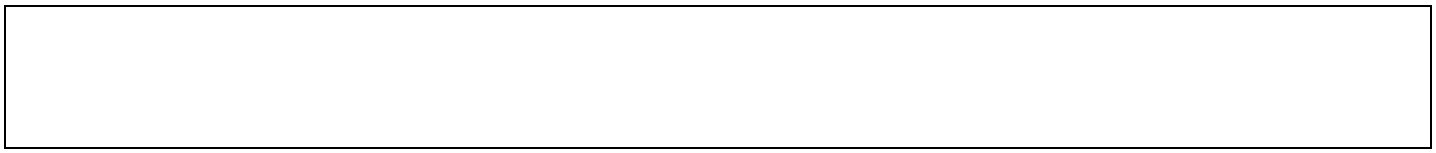
(Note: always run threeServer before threeClient)

---

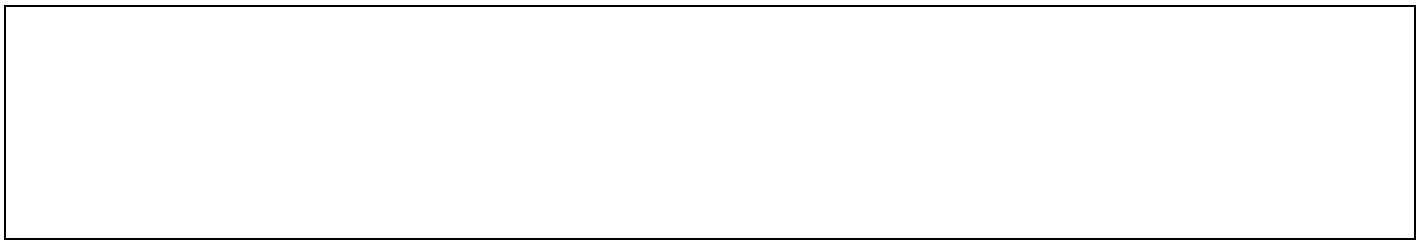
threeServer.c:

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

#define MAXDATASIZE 1024

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

    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]);
```

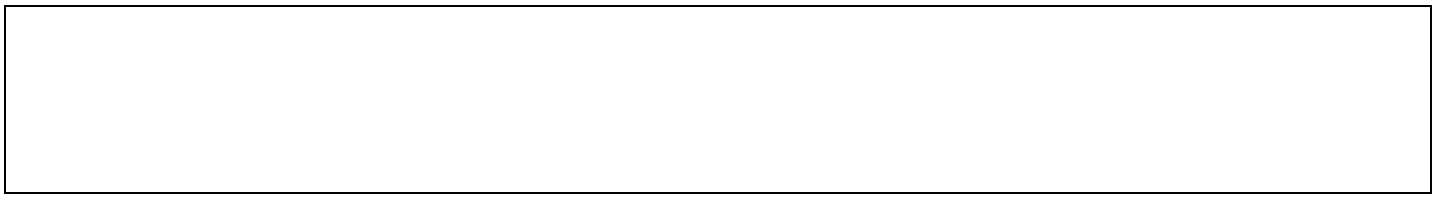
```
struct sockaddr_in sa, caller;
sa.sin_family = AF_INET;
sa.sin_addr.s_addr = INADDR_ANY;
sa.sin_port = htons(atoi(argv[1]));
```



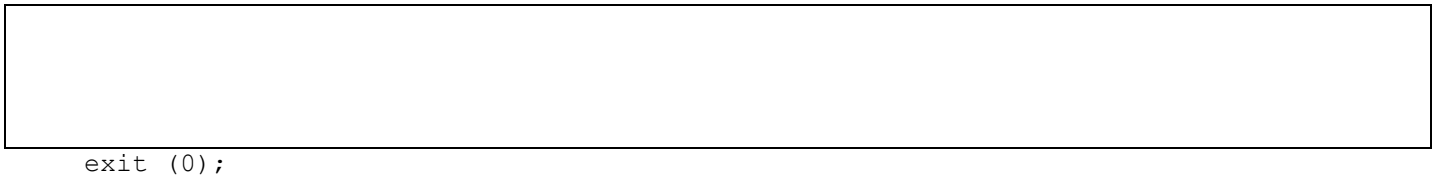
```
socklen_t length = sizeof(caller);
```



```
char message[MAXDATASIZE] = "congrats you successfully connected to the server!";
while (strlen(message) > 0)
{
    int numbytes; // number of bytes of data read from socket
    // send data to the client and then get data back from the client:
```



```
    message[numbytes - 1] = '\0';
}
```



```
exit (0);
}
```

threeClient.c:

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

#define MAXDATASIZE 1024

int main(int argc, char *argv[])
{
    if (argc != 3) {
            fprintf(stderr,"usage: threeClient hostname port-number\n");
            exit(1);
    }
```

```
    struct addrinfo their_addrinfo;
    struct addrinfo *their_addr = NULL;
    memset(&their_addrinfo, 0, sizeof(struct addrinfo));
    their_addrinfo.ai_family = AF_INET;
    their_addrinfo.ai_socktype = SOCK_STREAM;
    getaddrinfo(argv[1], argv[2], &their_addrinfo, &their_addr);
```

```
    char buffer[MAXDATASIZE]; //buffer contains data from/to server
    int numbytes; // number of bytes of data read from socket
    // get data from the server:
```

```
    while (numbytes > 0)
    {
        buffer[numbytes-1] = '\0';
        printf("%s\n", buffer);
        // send data to the server and then get data back from the server:
```

```
    }
```

```
    return 0;
}
```

// The expected output is listed below - very briefly describe why the output appears like this
//… at the end of your code file (in commented out space) …

```
congrats you successfully connected to the server
congrats you successfully connected to the serv
congrats you successfully connected to the se
congrats you successfully connected to the
congrats you successfully connected to th
congrats you successfully connected to
congrats you successfully connected t
congrats you successfully connected
congrats you successfully connect
congrats you successfully conne
congrats you successfully con
congrats you successfully c
congrats you successfully
congrats you successful
congrats you successf
congrats you succes
congrats you succ
congrats you su
congrats you
congrats yo
congrats
congrat
congr
con
c
```

END OF PAPER