Question 1. [8 MARKS]

All of the code fragments below compile but many of them have something wrong. Check the appropriate box to indicate whether or not there is a problem and provide an explanation. **There are no marks for checking SOMETHING IS WRONG without an explanation.**

Part (a) [1 MARK]

```
char city[7] = "Toronto";
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* assigning to string without enough space for null terminator */

Part (b) [1 MARK]

```
int * p;
*p = 12;
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* derefercing a pointer before pointing to anything */

Part (c) [1 MARK]

```
char * hobby = "running";
hobby[0] = "s";
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* assigning to a string in read-only memory */

Part (d) [1 MARK]

```
char fullname[30] = "Frederick";
fullname[4] = '\0';
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* nothing wrong - can assign a null-terminator to shorten a string */

Part (e) [1 MARK]

```
char postal[6] = "M4A";
strncat(postal, "3K5", 4);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* strncat without enough space */

Part (f) [1 MARK]

```
int j;
int x[5] = {4,2,7,8,9};
for (j = 0; j <= 5; j++) {
    fprintf("%d\n", x[j]);
}
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: accessing array elements beyond the end */

Part (g) [1 MARK]

```
char class[7] = "CSC209";
class[6] = "H";
int i = strlen(class);
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* strlen depends on null terminator */

Part (h) [1 MARK]

```
double average(int * a) {
    int i;
    double sum;
    for (i = 0; i < sizeof(a); i++ ) {
        sum += a[i];
    }
    return sum / sizeof(a);
}
```

☐ SOMETHING IS WRONG ☐ IT IS FINE

Explanation: SOLUTION: /* using sizeof in a function to try to get length of array */

Question 2. [10 MARKS]

All the subquestions below concern this `Makefile`.

Makefile

```
all: compile test

compile: prog1 prog2

prog1 : prog1.o tree.o
        gcc ${FLAGS} -o prog1 prog1.o tree.o

prog2 : prog2.o tree.o
        gcc ${FLAGS} -o prog2 prog2.o tree.o

test: prog1 prog2
        prog1 inputFile > prog1.out
        prog2 inputFile > prog2.out

%.o : %.c tree.h
        gcc ${FLAGS} -c $<
```

The subquestions must be evaluated in order. The state of the files for one question sometimes depends on the previous question. You may assume that all compilations are successful.

Part (a) [1 MARK]

In the Makefile the lines for compiling files work correctly even though the variable `FLAGS` is not defined inside `Makefile`. Give one reason why.

SOLUTIONS: An undefined variable is just an empty string so there are simply no flags in the gcc statement. Or it is possible t

Part (b) [1 MARK]

We compiled `prog1` using this Makefile and then tried to use gdb on the executable. It did not work. What needs to be changed in the Makefile to fix the problem?

SOLUTION: FLAGS= -g -Wall

Part (c) [1 MARK]

Suppose you have only the files `prog1.c, prog2.c, tree.c, tree.h and inputFile` in the same directory with `Makefile`. Which files, if any, are created, deleted, or modified when you call `make prog1`?

SOLUTION: tree.o prog1.o and prog1 are created.

Part (d) [1 MARK]

Which files, if any, are created, deleted, or modified if you call `make prog1` immediately again? Why?

SOLUTION: Nothing happens because the targets are up to date.

Part (e) [1 MARK]

Which files, if any, are created, deleted, or modified if you next call `make compile`?

SOLUTION: PROG2.O AND PROG2 ARE CREATED.

Part (f) [1 MARK]

In the box below, write a rule for a `clean` target that will remove all `.o` files, all executables and any output files created by using this Makefile.

SOLUTION
```
clean:
        rm -f *.o
        rm -f prog?
```

Part (g) [1 MARK]

Suppose you now run `make clean` followed by `make all`. What files do you now have in the directory? You do not need to list the original .c or .h files

SOLUTION: tree.o prog1.o prog2.o prog1 prog2 prog1.out prog2.out

Part (h) [3 MARKS]

In the Makefile above, the `test` rule only creates the two output files. In the box below, write additional actions to append to this rule so that it compares the two output files. If they are identical, it should print the message "test passes" to stdout. If they are not identical, it should print the message "Failed test: prog1.out does not match prog2.out" to standard error.

SOLUTION
```
test: prog1 prog2
    prog1 inputFile > prog1.out
    prog2 inputFile > prog2.out
    -diff prog1.out prog2.out > /dev/null
    if [ $? == 0 ] ; then
      echo test passes
    else
      echo Failed test: prog1.out does not match prog2.out > /dev/stderr
    fi
```

Question 3. [10 MARKS]

You have a program `foo` that prints to standard output and standard error and also has a meaningful return value.

Part (a) [2 MARKS]

Give shell command(s) to call `foo`, ignoring its return value and saving its standard output into the variable `Y`, and leaving standard error printing to the screen.

```
Y=`foo`
```

Part (b) [3 MARKS]

Give shell command(s) to call `foo`, ignoring its standard output and standard error (so that it doesn't appear on the screen) and saving the return value in the variable X.

```
foo  >/dev/null 2>&1  # or foo >& /dev/null
X=$?
```

Part (c) [2 MARKS]

Show how to call `foo`, sending its standard output to the program `wc`, sending the standard error to the file `foo.err` and ignoring the return value.

```
foo 2>foo.err | wc
```

Part (d) [3 MARKS]

A program called `myclient` uses the USER environment variable to identify the user, and takes an arbitrary number of arguments. You would like to test `myclient` with different values for the USER environment variable.

Write a shell program, `test_myclient` that takes a user name to store in USER as the first argument. The remaining arguments are arguments for `myclient`. `test_myclient` will set the USER environment variable and then call `myclient` with the correct set of arguments.

```
#!/bin/bash
    export USER=$1
    shift
    myclient "$@"
```

Question 4. [8 MARKS]

Below are four versions of a function `set_to_default` which is intended to be called as follows:

```
char ** names;
set_to_default(names, "Unknown", 4);     /* create an array of 4 names all set to "Unknown" */
/* OTHER CODE */
```

All of the implementations compile. Some of them may work, some may not. Some may work but the design choices restrict how the calling code can use `names`. For each subquestion, check one box indicating how the implementation will work in the context of the whole program. Then explain your answer paying particular attention to the restrictions that would be placed on the actions in the `OTHER CODE` statements in the calling function.

Part (a) [2 MARKS]

```
void set_to_default(char **names, char *default, int size) {
    int i;
    names = malloc(sizeof(char *) * size);
    int needed_size = strlen(default) + 1;
    for (i = 0; i < size; i++ ) {
        names[i] = malloc(sizeof(needed_size));
        strncpy(names[i], default, needed_size);
    }
}
```

Actually none of the implementations will work properly since the local variable names is a copy of the argument to the function and it is immediately reset inside all 4 implementations. We should have passed the address of names and had the first parameter be `char ***names_ptr` which we used to get a local `char **names = *names_ptr`.

This was a mistake on our part. The few students who noticed this, got full marks but we also marked the answers below as correct as well.

Part (b) [2 MARKS]

```
void set_to_default(char **names, char *default, int size) {
    int i;
    names = malloc(sizeof(char *) * size);
    int needed_size = strlen(default) + 1;
    char * default_name = malloc(sizeof(needed_size));
    strncpy(default_name, default, needed_size);
    for (i = 0; i < size; i++ ) {
        names[i] = default_name;
    }
}
```

memory allocated but all names point to same copy of default name So if we change one, they all change

Part (c) [2 MARKS]

```
void set_to_default(char **names, char *default, int size) {
    int i;
    names = malloc(sizeof(char *) * size);
    int needed_size = strlen(default) + 1;
    char default_name[needed_size];
    strncpy(default_name, default, needed_size);
    for (i = 0; i < size; i++ ) {
        names[i] = default_name;
    }
}
```

SOLUTION: Doesn't work because all pointers point to local char array on stack

Part (d) [2 MARKS]

```
void set_to_default(char **names, char *default, int size) {
    int i;
    names = malloc(sizeof(char *) * size);
    for (i = 0; i < size; i++ ) {
        names[i] = default;
    }
}
```

SOLUTION: All names in array will be set to default but are read-only memory

Question 5. [17 MARKS]

Part (a) [7 MARKS]

You have a set of input files that are encrypted with a form of run-length encoding and have the following format. The first 4 bytes of the file are an integer in binary representation that indicate the number of header bytes that come next. After the header, the file alternates between one integer followed by one character. You will write a function that given a filename, decrypts that file and sends the decrypted result to a file descriptor.

To do this you only care about the pairs of integers and their corresponding characters so first you can skip the header. The integer indicates how many of that character should come next in the file. So, for example, if the data consists of a binary representation of 1 followed by a 'b' and then a binary 5 followed by an 'o', the output sent to the file-descriptor should be 'booooo'. The last thing the function should do is close the file descriptor.

Complete the function `decode`.

```
int decode(char * filename, int fd) {

#define MAXLINE 256   /* needs to go earlier if you want to use it */

    FILE *fp = fopen(filename, "r");
    int i, header_lines;
    char junk[MAXLINE];
    char c;
    int count;
    fread(&header_lines, 1, sizeof(int), fp);
    for (i = 0; i < header_lines; i++) {
        fread(junk, 1, MAXLINE, fp);
    }

    while (fread(&count, 1, sizeof(int), fp) != 0) {
        fread(&c, sizeof(char), 1, fp);
        printf("count: %d\n",count);
        for (i = 0; i < count; i++) {
                write(out, &c, sizeof(char));
        }
    }
    fclose(fp);
    return 0;
}
```

Part (b) [8 MARKS]

Assume that you have a `decode` function that works according to the specification. It doesn't matter for this question if you didn't quite manage to get it correct in Part a, we will assume it works.

Assume also that you have a corresponding `encode` function with the signature `encode(char * filename, int fd)` that reads a series of characters from file descriptor `fd`, does the run-length encoding and creates and writes the encrypted file to `filename`.

For this question, you must complete the C program below that creates two child processes and communicates with them using pipes. One of the children will call the `decode` to read an encrypted input file and pass the unencrypted contents to the parent through the pipe. Although it is a bit unrealistic, the parent process you write won't do anything with the unencrypted data except pass it along (through a different pipe) to the other child. The other child will call the `encode` function to read the raw data from the parent, encrypt it and write it to a new file. You could imagine that in a real program, the parent would be doing something interesting with the unencrypted data before sending it back to be re-enrypted but that would be too much for this question! For full marks, pipes and files must be closed at the appropriate times.

Although comments are not required for full marks, they are probably helpful to keep yourself on track and to allow the markers to see what you are trying to do. You do not need to show the error checking on your system calls.

```
int main() {
    char * original = "original_encrypted_file.xxx";
    char * created = "resulting_encrypted_file.xxx";



    int fd_in[2];
    int fd_out[2];
    int pid;

    pipe(fd_in);
    pid = fork();
    if (pid == 0) {
      /* I am the child who will do the reading and write to the parent */
      close(fd_in[0]);
      decode(original, fd_in[1]);
      exit(0);
    } else {
      /* I am the parent who needs to fork to make other child */
      pipe(fd_out);
      pid = fork();
      if (pid == 0) {
        /* I am the child who will read from the parent */
        close(fd_in[0]);  /* still open in parent */
        close(fd_in[1]);  /* still open in parent */
        close(fd_out[1]);  /* still open in parent */
        encode(created, fd_out[0]);
        exit(0);
      } else {
      /* I am still the parent */
      close(fd_out[0]);
      close(fd_in[1]);
      char c;
      while (read(fd_in[0], &c, sizeof(char)) != 0) {
         /* just pass along the character */
         write(fd_out[1], &c, sizeof(char));
      }
    }
    return 0;
  }
}
```

Part (c) [2 MARKS]

In this course, we have emphasized the importance of error checking, but because of space and time constraints, we have told you not to do any error checking in your answers.

For this question choose one of the system calls that you used in part (b) and write a snippet of C code that would replace the code fragment containing the call with code that properly tests for an error.

Code from Part B without any error checking:

Replacement code that checks for error:

SOLUTION:Any one of close or pipe or read or write checking the return code and calling perror and exit correctly.

Question 6. [8 MARKS]

Study the following program that installs a signal handler.

```
int turn = 0;

void handler(int code) {
    if(turn == 0) {
        fprintf(stderr, "First\n");
        turn = 1;
        /* D */

    } else {
        fprintf(stderr, "Second\n");
        kill(getpid(), SIGQUIT);
    }
    fprintf(stderr,"Here\n");
}

int main() {

    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGQUIT);

    /* A*/

    sigaction(SIGTERM, &sa, NULL);

    /* B */

    fprintf(stderr, "Done\n");

    /* C */
    return 0;
}
```

Question 6. (CONTINUED)

Show the output of the program when events described in each row occur. Treat each row as if the program were restarted. Each event is described as a signal that arrives *just before* the process executes the line of code following the specified comment line. Give the **total** output of the program in each case.

Default behaviour:

- SIGQUIT - prints "Quit" before terminating the process.
- SIGTERM - prints "Terminated" before terminating the process.

Events	Output
Two SIGTERM signals arrive one after the other at A	Terminated
SIGTERM arrives at B and SIGTERM arrives at C	
SIGTERM arrives at B and SIGQUIT arrives at D	
SIGTERM arrives at B and SIGTERM arrives at D	

Question 7. [5 MARKS]

Part (a) [1 MARK]

How do you tell when there is nothing more to read on a pipe or a socket?

The other end has been closed, and read returns 0.

Part (b) [3 MARKS]

Explain what each of the following system calls do to set up a socket for a server:

`int bind(int sock, struct sockaddr *addr, int addrlen)`

$\boxed{\text{Associates the socket with the IP address and port}}$

`int listen(int sock, int n)`

$\boxed{\text{Sets up the queue in the kernel for partial connections.}}$

`int accept(int sock, struct sockaddr *addr, int *addrlen)`

$\boxed{\text{Accepts a connection from the network and fills in the addr of that connection and returns the new socked on which this conne}}$

Part (c) [1 MARK]

Which (if any) of the system calls in the previous question might block?

Only accept can block.

Question 8. [11 MARKS]

Recall from assignment 4, the dropbox client performed a synchronization operation every N seconds that compared last modified times of files on the client to the last modified times of the same files stored on the server. If a file on the server was newer, it was downloaded to the client. If a file on the client was newer, it was uploaded to the server.

For this question, we will be implementing a simpler client. Instead of uploading and downloading files, the client will print to standard output the names of the files that are newer on the server. This means that the only messages being sent to and received from the server are `sync_messages`.

The other modification we will make to the client is that instead of synchronizing with the server every N seconds, it synchronizes only when the user prompts it to by entering a command on the keyboard. In other words, the client will read from standard input. If it receives the character "s" it will initiate a synchronization operation as long as no synchronization operation is in progress. If the client is in the middle of a synchronization operation when the user types "s", the client will display a message to the user: "Sync in progress, try again later" and will not initiate a synchronization operation. If the user types "q", the client will close the socket and terminate.

Complete the code below that uses `select` to implement this modification. Assume that all variables and constants have been correctly initialized and the socket connection to the server has already been established. Assume the user will only type valid input. No error handling is required.

Part (a) [1 MARK]

Explain the purpose of using `select` in the case.

SOLUTION: We need select so the client can respond to either the keyboard or the server and not block on one when the other will be the next to communicate.

Here are the types, variables and constants needed for the code below. Note the function prototype for `getnextfile`. You do not need to implement it, but are welcome to use it.

```
// data structure for sending and receiving file information
// from the server
struct sync_message {
    char filename[MAXNAME];
    time_t mtime;
    int size;
};

// constants used to keep track of the current state of input
#define IDLE 0  // No synchronization operation in progress.
#define SYNC 1  // Synchronization in progress.

int sock;  // the file descriptor that connects to the server
DIR *dir;  // the directory pointer

// Gets the next file entry from the directory, and fills in
// correct values in r. Assume this is implemented.
int getnextfile(DIR *dir, struct sync_message *r);
```

Part (b) [2 MARKS]

Finish setting up the variables needed to run select correctly as the code continues in the next subquestion.

```
fd_set rset, allset;

maxfd = sock;
FD_ZERO(&allset);
FD_SET(sock, &allset);
FD_SET(fileno(stdin), &allset);
```

Part (c) [8 MARKS]

Complete the loop below to handle input coming from the user.

```
struct sync_message request, response;
state = IDLE;

while(1) {
    rset = allset;
    select(maxfd + 1, &rset, NULL, NULL, NULL);

    if(FD_ISSET(sock, &rset)) {
         if(state == SYNC) {
             read(sock, &response, sizeof(struct sync_message));
             if(response.mtime > request.mtime) {  // server has newer version
                 printf("%s %d", response.filename, response.mtime);
             }
             if(getnextfile(dir, &request) == 0) {
                 state = IDLE;
             } else {
                 write(sock, request, sizeof(struct sync_message));
             }

         } else {
             fprintf(stderr, "error state");
         }
     }


if(FD_ISSET(fileno(stdin), rset)) {
    char buf[20];
    read(fileno(stdin, buf, 20));
    if(buf[0] == 's') {
        if( state == IDLE) {
            // start the sync procedure
            dir = opendir(argv[2]);
            if(getnextfile(dir, &request) == 0) {
                state = IDLE;
            } else {
                write(sock, request, sizeof(struct sync_message));
                state = SYNC;
            }
        } else {
            fprintf(stderr, "Sync in progress. Try again later.\n");
        }

    } else if(buf[0] = 'q') {
        close(sock);
        closedir(dir);
        exit(0);
    }
}
```

Question 8. (continued)

This page can be used if you need additional space for your answers.