Question 1. [8 MARKS]

Circle the correct answer for the statements below.

TRUE **FALSE** Multiple processes on a machine may use the same port number to listen for connections.

TRUE FALSE A `SIGPIPE` signal is sent when a process makes a `write` call to a closed socket or pipe.

TRUE **FALSE** The standard line ending used in network communications is `'\n'|`.

TRUE **FALSE** The function `perror` is used to print error messages for any function.

TRUE FALSE Given the following code, the file `fp` can be closed immediately after the `dup2` call because it is no longer needed.

```
FILE *fp = fopen("myfs", "r");
dup2(fileno(fp), fileno(stdin));
```

TRUE **FALSE** Suppose we have a Makefile with a target `t`. Running `make t` will always create a file called `t` in the current working directory.

TRUE FALSE The code below copies the fields of `b` into `a`.

```
struct node a, b;
//missing code to assign values to the fields in b
a = b;
```

TRUE **FALSE** The code below is an example of a memory leak.

```
struct node {
    int val;
    struct node *next;
};
void free_list(struct node *head) {
    while(head != NULL) {
        free(head);
        head = head->next;
    }
}
```

Question 2. [6 MARKS]

Part (a) [4 MARKS]

Write a shell script that adds every subdirectory of your current working directory to your `PATH` variable. Note that the `PATH` variable requires an absolute path to each subdirectory. Remember that the environment variable `PWD` stores the absolute path to the current working directory. (Do not recurse into subdirectories.)

```
for i in *
do
    if [ -d $i ]; then
        export PATH=$PATH:$(pwd)/$i
    fi
done
```

Part (b) [2 MARKS]

A program `my_prog` in the *parent* directory of the current working directory prints out some plaintext when run.

Write a single shell command that computes the number of words in the output of `my_prog`, and prints this number to the file called `outfile`.

```
../my_prog | wc -w > outfile
```

Question 3. [6 MARKS]

For each pair of code snippets below, select `match` if the pair would output the same thing, `does not match` if the output would be different, or `may not match` if the output might match, but not always.

If you select `does not match` or `may not match`, briefly explain why.

Part (a) [1 MARK]

```
int a = 3;
int *b = &a;
printf("%d", *b);
```

```
int *c = malloc(sizeof(int));
*c = 3;
printf("%d", c);
```

Selection

☐ `match` ☑ `does not match` ☐ `may not match`

Explanation

The printf() on the RHS prints the memory address pointed to by a, rather than the dereferenced value.

Part (b) [1 MARK]

```
int x = 1 << 3;
int y = x | 1;
printf("%d %d\n", x, y);
```

```
int z = 8;
printf("%d %d\n", z, z + 1);
```

Selection

☑ `match` ☐ `does not match` ☐ `may not match`

Explanation

Part (c) [1 MARK]

```
int res = fork();
if (res == 0) {
    execl("/bin/ls", "ls", NULL);
    exit(42);
} else if (res > 0) {
    int status;
    wait(&status);

    if (WIFEXITED(status)) {
        printf("%d", WEXITSTATUS(status));
    }
}
```

```
printf("%d", 42);
```

Selection

☐ match ☑ does not match ☐ may not match

Explanation

The exit() call in the child process comes after execl() and will not be executed. It is also unlikely that "ls" would return 42.

Part (d) [1 MARK]

```
char *s = "hillo world";
s[1] = 'e';
printf("%s", s);
```

```
printf("hello world");
```

Selection

☐ match ☑ does not match ☐ may not match

Explanation

s is a string literal and cannot be mutated.

Part (e) [1 MARK]

```
char *s = malloc(strlen("hello world"));
strcpy(s, "hello world");
printf("%s", s);
```

```
printf("hello world");
```

Selection

☐ match ☐ does not match ☑ may not match

Explanation

It's possible for "hello world" to be printed properly even if no space was explicitly allocated for the null terminator. This depends on what already exists in memory.

Part (f) [1 MARK]

```
char *s = "hello world";
char *t = malloc(sizeof(s));
strcpy(t, s);
printf("%s", t);
```

```
printf("hello world");
```

Selection

☐ match ☐ does not match ☑ may not match

Explanation

It's possible for "hello world" to be printed properly even if insufficient space was allocated for the whole string, depending on factors such as if the extra memory used by t wasn't overwritten.

Question 4. [10 MARKS]

Consider the program below.

Part (a) [2 MARKS]

(i) What is the value of `strlen(course)` just before `stem` returns? 9

(ii) What is the value of `strlen(s)` just before `stem` returns? 6

(iii) What is the value of `strlen(ptr)` just before `stem` returns? 0

(iv) What does the print statement in `main` print? CSC209H1F CSC209

Part (b) [8 MARKS]

Draw a complete memory diagram showing all of the memory allocated immediately before `stem` returns. Clearly label the different sections of memory (stack, heap, global), as well as different stack frames. You must show where each variable is stored, but make sure it's clear in your diagram what is a variable *name* vs. the *value* stored for that variable. You may show a pointer value by drawing an arrow to the location in memory with that address, or may make up address values.

```
char *stem(char *course) {
    char *s = malloc(strlen(course) + 1);
    strncpy(s, course, strlen(course) + 1);
    char *ptr = s;
    while(*ptr != 'H') {
        ptr++;
    }
    *ptr = '\0';
    // Draw memory diagram at this point
    return s;
}

int main() {
    char *c1 = "CSC209H1F";
    char *c2 = stem(c1);
    printf("%s %s\n", c1, c2);
    return 0;
}
```

Question 5. [8 MARKS]

Implement the following function according to its documentation.

```
struct pair {
    char *name;
    char *value;
}

/*
 * You are given a null-terminated string, s, in the following form:
 * - It contains between 1 and 10 lines, where each line except the last ends with '\n'
 * - Each line is of the form "<name>: <value>", where <name> and <value> are non-empty strings.
 *   Note that there is exactly one space between the colon and value string.
 * - Each name and value string do *not* contain a colon (but may contain spaces).
 *
 * For example:
 *   - "a: bc"
 *   - "name1: a\nname2: b"
 *   - "Content-Type: text/plain\nHost: teach.cs.toronto.edu"
 *
 * Your job is to parse the string, storing each name and value in an array of
 * struct pairs. Any unfilled pairs in the array have their name and value
 * fields set to NULL (we've initialized this for you).
 */
```

```
struct pair *parse_pairs(char *s) {
    // Initialize an array (assume there's at most 10 pairs in s).
    struct pair *data = malloc(10 * sizeof(struct pair));
    for (int i = 0; i < 10; i++) {
        data[i].name = NULL;
        data[i].value = NULL;
    }

    int count = 0;  // The number of pairs read so far.
    char *curr = s;
    while (count < 10) {
        // Read a new name.
        char *colon_pos = strchr(curr, ':');
        int name_len = colon_pos - curr;
        data[count].name = malloc(name_len + 1);
        strncpy(data[count].name, curr, name_len);
        data[count].name[name_len] = '\0';

        // Read a new value.
        char *value_start = colon_pos + 2;  // +2 to skip the ": ".
        char *value_end = strchr(value_start, '\n');
        if (value_end == NULL) {
            // This is the last key-value pair.
            // Note thave value_start is null-terminated, and contains exactly the last value string.
            data[count].value = malloc(strlen(value_start) + 1);
            strcpy(data[count].value, value_start);
            break;
        } else {
            // There's more key-value pairs.
            int value_len = value_end - value_start;
            data[count].value = malloc(value_len + 1);
            strncpy(data[count].value, value_start, value_len);
            data[count].value[value_len] = '\0';

            curr = value_end + 1;  // +1 to skip the '\n'
            count++;
        }
    }

    return data;
}
```

Question 6. [8 MARKS]

Write a program that takes at least one command-line argument, where each argument is the name of an executable to run.

The program should fork a new child for each command-line argument, and the child should call `execl` to run the executable named by the argument. (The executable itself takes no arguments.) The child should exit with a non-zero value if the `execl` fails.

The parent process should wait for all of its child processes to complete. If *all* of its child processes exit successfully with an exit status of 0, the parent prints `"Success\n"`. Otherwise, the parent prints `"Failure\n"`.

Here is an example of how the program could be called:

```
$ ./run_all ls date ps
```

```
int main(int argc, char **argv) {
    for (int i = 1; i < argc; i++) {
        if (fork() == 0) {
            // Child process: run an executable.
            // Note: we use execlp to search for executables on $PATH,
            // but we don't care about this (vs. execl) for grading purposes.
            execlp(argv[i], argv[i], NULL);
            perror("execlp");
            exit(1);
        }
    }

    // Parent here.
    int status;
    int all_success = 1;
    for (int i = 1; i < argc; i++) {
        wait(&status);
        if (! (WIFEXITED(status) && WEXITSTATUS(status) == 0) ) {
            all_success = 0;
        }
    }

    if (all_success) {
        printf("Success\n");
    } else {
        printf("Failure\n");
    }

    return 0;
}
```

Question 7. [6 MARKS]

Suppose we want to write a program that does the following:

- The parent creates a child process.
- The parent sends the child process a single integer.
- The child multiplies the integer by 2, then sends the result back to the parent.
- The parent prints the result to standard output.

Here is an incorrect implementation of this program. We have omitted error-checking for brevity. Note that lack of error checking is not the problem.

```
int main() {
    int val, result;
    int fd[2];
    pipe(fd);

    int pid = fork();

    if (pid > 0) { // Parent process
        val = 10;
        write(fd[1], &val, sizeof(int));
        close(fd[1]);
    } else { // Child process
        read(fd[0], &val, sizeof(int));
        close(fd[0]);
        result = val * 2;
        write(fd[1], &result, sizeof(int));
        close(fd[1]);
        return 0;
    }

    // Parent process here
    read(fd[0], &result, sizeof(int));
    close(fd[0]);
    printf("result: %d\n", result);
    return 0;
}
```

Part (a) [2 MARKS] Explain what could go wrong with the *parent process* when we run this program.

The parent might call `read` *from the pipe before the child does. In this case, the parent would read the number it wrote (10) and print it out, rather than 20.*

Part (b) [2 MARKS] Explain what could go wrong with the *child process* when we run this program.

The parent might call `read` *from the pipe before the child does. In this case, the child process would block when it calls* `read`*, as it still has the pipe's write end open.*

Part (c) [2 MARKS] Briefly explain how to fix the program.

We need to use two pipes: one for the parent to send data to the child, and another for the child to send data to the parent.

Question 8. [10 MARKS]

For each of the statements below, explain one example of what would cause the outcomes or results described in the comments in each box. Answers must be precise.

`result = read(fd, buf, SIZE);` `// process blocks`	fd is for a socket or a pipe, and the other end has not written any bytes
`result = read(fd, buf, SIZE);` `// result == 0`	file, pipe or socket is closed (Not error)
`result = read(fd, buf, SIZE);` `// result < SIZE`	reading the last bytes of a file OR only result bytes were available on the pipe or socket
`result = read(fd, buf, SIZE);` `// process exits with a segmentation fault`	buf is not a valid memory pointer
`result = write(fd, buf, SIZE);` `// process blocks`	fd is a pipe and the pipe is full
`result = wait(&status);` `// process blocks`	A child is still running
`result = wait(&status);` `// returns immediately and result > 0`	A childs has terminated
`result = select(maxfd, &rset, NULL, NULL, NULL);` `// process blocks`	None of the file descriptors in rset have something ready to read
`result = accept(fd, &addr, sizeof(addr));` `// process blocks`	No client has called connect
`result = accept(fd, &addr, sizeof(addr));` `// result > 0`	A client has called connect

Question 9. [9 MARKS]

Below is an implementation of an interactive client that reads a hostname from standard input, makes a connection to a server running on `PORT`, and then reads two pieces of information from the server: the number of users (as a 4-digit null-terminated string), and the average load (as a 4-digit null-terminated string).

Assume `connect_to_server` and `add_host` are implemented correctly, and that `connect_to_server` will not block. Error checking is omitted for brevity.

```
#define MAXHOSTS 10
struct server {
    int soc;
    char host[64];
    char number[5];
    char load[5];
};
// Add hostname and soc to an empty slot in servers array.
void add_host(struct server servers, char *hostname, int soc);

// Create a socket and connect to the server at port and hostname. Assume connection succeeds.
int connect_to_server(int port, const char *hostname);

int main() {
    fd_set rset, allset;
    int maxfd;
    char buf[MAXLINE];
    struct server servers[10];

    // initialize all strings in servers to empty strings, and the soc field to -1
    init_server(servers);

    FD_ZERO(&allset);
    maxfd = 0;

    while (1) {
        // Get hostname from stdin and make connection
        read(fileno(stdin), buf, MAXLINE);
        int soc = connect_to_server(PORT, buf);

        add_host(servers, buf, soc);
        FD_SET(soc, &allset);
        if (soc > maxfd)  maxfd = soc;

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

        // Check to see which hosts have data ready
        for (int i = 0; i < MAXHOSTS; i++) {
            if (FD_ISSET(servers[i].soc, &rset)) {
                read(servers[i].soc, servers[i].number, 5);
                read(servers[i].soc, servers[i].load, 5);
                printf("%s: %s %s\n", servers[i].host, servers[i].number, servers[i].load);
            }
        }
    }
}
```

Part (a) [3 MARKS]

When we run the program all the output is correct, but we notice that we cannot type in another host name to check until we have received all of the output from the previous host. There are three problems with the program that need to be fixed to address this issue. Explain each problem. Failing to check for errors is **not** a problem with the code, and all helper functions are correctly implemented.

- stdin is not added to the set that select is monitoring
- it will block in the first read call of the while loop if the user doesn't type something.
- a server might take a long time in between sending the number and sending the load. We should be calling select in between these two reads.

Part (b) [6 MARKS]

Fix the code by rewriting it below. You do not need to re-write any code above the call to `FD_ZERO`.

```
#define MAXHOSTS 10
struct server {
    int soc;
    char host[64];
    char number[5];
    char load[5];
};
// Add hostname and soc to an empty slot in servers array.
void add_host(struct server servers, char *hostname, int soc);

// Create a socket and connect to the server at port and hostname. Assume connection succeeds.
int connect_to_server(int port, const char *hostname);

int main() {
    fd_set rset, allset;
    int maxfd;
    char buf[MAXLINE];
    struct server servers[10];

    // initialize all strings in servers to empty strings, and the soc field to -1
    init_server(servers);

    FD_ZERO(&allset);
```

```
    FD_SET(fileno(stdin), &allset); //OK if they just use stdin
    maxfd = 0;

    while (1) {
        // Get hostname from stdin and make connection
        rset = allset;
        select(maxfd+1, &rset, NULL, NULL, NULL);
        if(FD_ISSET(fileno(stdin), &rset)) {
            read(fileno(stdin), buf, MAXLINE);
            int soc = connect_to_server(PORT, buf);

            add_host(servers, buf, soc);
            FD_SET(soc, &allset);
            if (soc > maxfd)  maxfd = soc;
        }

        // Check to see which hosts have data ready
        for (int i = 0; i < MAXHOSTS; i++) {
            if (FD_ISSET(servers[i].soc, &rset)) {
                if(servers[i].number[0] == '\0') {
                    read(servers[i].soc, servers[i].number, 5);
                } else {
                    read(servers[i].soc, servers[i].load, 5);
                    printf("%s: %s %s\n", servers[i].host, servers[i].number, servers[i].load);
                }
            }
        }
    }
}
```

Question 10. [6 MARKS]

You are given a binary file that contains an array of structs of the type shown below. Complete the following function according to its documentation.

```
#define MAXNAME 32

typedef struct {
    char name[MAXNAME];
    int offset;
    int length;
} Inode;

/* Write an Inode containing the fields "name", "offset", and "length" to the
 * Inode at index "ind" in the file named "filename". Include appropriate error
 * checking to handle the case where "ind" is not a valid index.
 * Return 0 if the write was successful, and -1 in the case of error.
 * Do not use a loop.
 */
int write_inode(char *filename, int ind, char *name, int offset, int length) {


int write_inode(char *filename int index, char *name, int offset, int length) {
    Inode inode;
    FILE *fs =  fopen(filenname, "r+");
    strncpy(inode.name, name, MAXNAME);
    inode.offset = offset;
    inode.length = length;
    if(fseek(fs, index * sizeof(Inode), SEEK_SET) == -1) {
        perror("fseek");
        return -1;
    };
    fwrite(&inode, sizeof(Inode), 1, fs);
    return 0;
}
```