1. Miscellaneous short answers.

(a) Alan has a file named `dec25`. He wants to create a link `xmas` to `dec25`, such that if `dec25` is deleted and then a new `dec25` is created with new content, `xmas` refers to the new version. Advise Alan on whether the link should be a hard link or symbolic link.

> Alan should use a symlink, as a hard link would still point to the original `dec25` even after it has been deleted and replaced.

(b) Anne has a file named `oct31`. She wants to create a link `hlw` to `oct31`, such that if `oct31` is deleted and then a new `oct31` is created with new content, `hlw` refers to the old version. Advise Anne on whether the link should be a hard link or symbolic link.

> Anne should use a hard link, for the same reason as above.

(c) Manfred and Natasha are implementing pipelining in C, equivalent to the shell command

`prog1 | prog2`

Their code fragments are below. What's wrong in each case? Assume that code not shown but outlined in comments (the "ellided") is correct.

Manfred's version

```
// pipe creation ellided
pid_t p1, p2;
int s1, s2;
p1 = fork ();
if (p1 == 0) {
    // close and dup2 ellided
    execlp("prog1", "prog1",
    (char*)NULL);
} else if (p1 > 0) {
    // close ellided waitpid(p1, &s1);
    p2 = fork ():
    // rest of code ellided
```

Natasha's version

```
// pipe creation ellided
pid_t p1, p2;
int s1, s2;
p2 = fork ();
if (p2 == 0) {
    // close and dup2 ellided
    execlp("prog2", "prog2",
    (char*)NULL);
} else if (p2 > 0) {
    // close ellided waitpid(p2, &s2);
    p1 = fork ();
    // rest of code ellided
```

> Manfred's `p2 = fork()` line ends with a `:` which would cause compiler problems, while Natasha execs `prog2` before `prog1`, which is not equivalent to `prog1 | prog2`

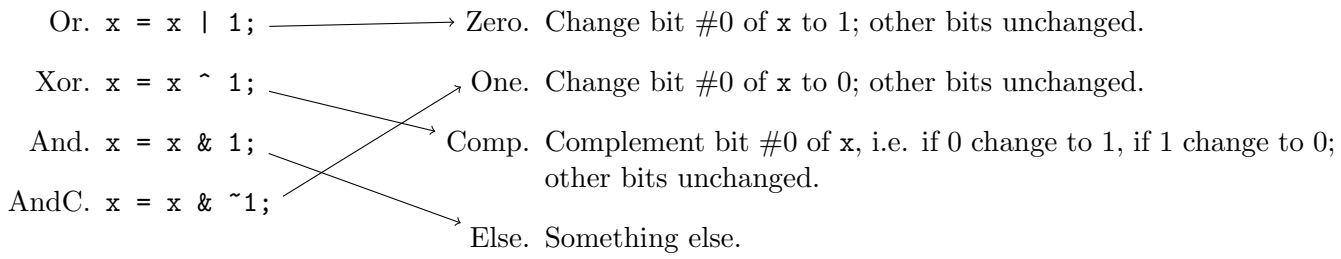
(d) Tom and Jerry both want their programs to, upon the signal `SIGINT`, continue normal running and not respond to the signal. Tom sets the signal action to `SIG_IGN`; Jerry sets the signal action to this no-op handler:

```
void jerry_handler(int sig) { }
```

What happens when Tom's program calls `select` and it blocks, and then `SIGINT` arrives? What happens in Jerry's case?

> In Tom's case, since `SIGINT` is set to `SIG_IGN`, the program ignores the signal, while in Jerry's case, since he uses a handler, `select` will be interrupted and will then return `-1`.

(e) In an 8-bit binary number $b_7b_6b_5b_4b_3b_2b_1b_0$, we say "bit #0" for b_0. E.g., in 00000001, bit #0 is 1; in 11111110, bit #0 is 0. Let `x` be an `unsigned char` variable. Match the code fragments on the left below to the effects on the right (you may just write the labels "Or", "Comp", etc.):



2. The Grande Online-Only Cafe wants to give out prizes to a few of its registered customers by a lucky draw! They already have a customer file like this:

   ```
   743 Blanchett Anne anne.b@gmail.com
   89 Hathaway Cate meow32@hotmail.com
   ```

   etc., there are hundreds of more lines. Each line is a customer record with customer ID, family name (restricted to one word), given name (again one word), and email address, using a single space to separate two fields.

   Assume that there is a program `shuf` (actually it exists) that reads lines from stdin and outputs a random permutation of the lines to stdout. Write a shell pipeline that picks 5 random customer lines and outputs them. The output should be alphabetically ordered by family names; if tie, by given names (if still tie, no further tie-breaking necessary).

   The complete customer file will come from stdin, and your output should go to stdout. Don't worry about duplicate customer records.

   Note that only a shell pipeline is accepted.

```
shuf - | head -5 | sort -k 2,3
```

3. Macy's school computer doesn't have `make` installed for some reason! She needs your help writing a shell script that performs a small but important job of `make`. She has a lot of TeX files (filenames end with ".tex"), and there is a program `pdflatex` for generating corresponding PDF files. Using "A1.tex" as an example, "`pdflatex A1.tex`" generates "A1.pdf". Clearly, she wants to generate if and only if one of:

   - A1.tex exists but A.pdf doesn't
   - both exist, but A1.tex is newer

   "A1.tex" is just an example—this applies to all TeX and PDF files in the current directory.
   Implement a Bourne shell script to do this.
   The command "`basename A1.tex .tex`" outputs A1 to help you.

```
# Assuming an argument will be provided like the actual make function
filename=$(basename "$1" .tex)
# Checks if argument provided exists, then check if either pdf does not exist,
# or if pdf exists and tex file is newer
[ -e "$1" ] && ( [ ! -e "$filename".pdf ] || ( [ -e "$filename".pdf ] &&
    [ "$1" -nt "$filename".pdf ] ) ) && pdflatex "$1"
```

4. If you can look up in RAM quickly, you can look up on disk slowly!
   In this question, you will write code for binary search tree lookup, but the tree is stored in a binary file on disk! The file consists of 0 or more nodes defined by this struct:

```
typedef struct {
    int key;
    long left, right; // file positions (absolute) or -1L
}   node;
```

   `key` is a key as usual. Note that instead of pointers to children, we have file positions (offsets) of children, so their types are long as in `fseek`; accordingly, when there is no left/right child, we use `-1L` (-1 but type `long`).
   The file may be empty, meaning the tree is empty; but if not, we know that the root node is at the beginning–position 0. Other nodes may be anywhere else, we only know that positions of nodes are non-negative multiples of `sizeof(node)`.

   Implement lookup:
   `int is_present(FILE *f, int needle);`
   This looks for `needle` in the tree in `f`. It should return 1 (C true) if `needle` is a key in the tree, 0 (C false) if not.

   You do not know where the current file position is before this function begins. You may assume that `f` allows `fseek`, and `fread` on `f` either hits EOF or succeeds. You may assume that if `left` is non-negative, the left child node exists at that position; similarly for `right`.

```c
int is_present_helper(FILE *f, int needle, long file_pos) {
    if (file_pos == -1L) {
        return 0;
    }
    fseek(f, file_pos, SEEK_SET);
    node root;
    fread(&root, sizeof(root), 1, f);
    if (root.key == needle) {
        return 1;
    }
    if (root.key > needle) {
        return is_present_helper(f, needle, root.left);
    }
    else {
        return is_present_helper(f, needle, root.right);
    }
}

int is_present(FILE *f, int needle) {
    return is_present_helper(f, needle, 0);
}
```

5. The files to submit in this question are `line.h`, `solve.h`, `Makefile`
Helen wishes to modularize the following C file into multiple C files for separate compilation.

```
#include <stdio.h>
typedef struct {
    double m, c; // as in y=mx+c
} line;

double compute_y(const line *L, double x) {
    return L->m * x + L->c;
}

typedef struct {
    double x, y;
} point;

int solve(point *p, const line *L1, const line *L2)
{
    double d = - L1->m + L2->m; if (d == 0) return 0;
    else {
        p->x = L1->c - L2->c;
        p->y = - L1->m * L2->c + L2->m * L1->c; return 1;
    }
}

int main(void)
{
    line L1, L2;
    point P;
    L1.m = 4;
    L2.m = -2; L2.c = 2; printf("%f\n", compute_y(&L1, 0)); solve(&P, &L1, &L2);
    printf("%f %f\n", P.x, P.y);
    return 0;
}
```

```c
#ifndef _LINE_H
#define _LINE_H

typedef struct {
    double m, c; // as in y=mx+c
} line;

double compute_y(const line *L, double x);

#endif
```

line.h

```c
#ifndef _SOLVE_H
#define _SOLVE_H

#include "line.h"

typedef struct {
    double x, y;
} point;

int solve(point *p, const line *L1, const line *L2);

#endif
```

solve.h

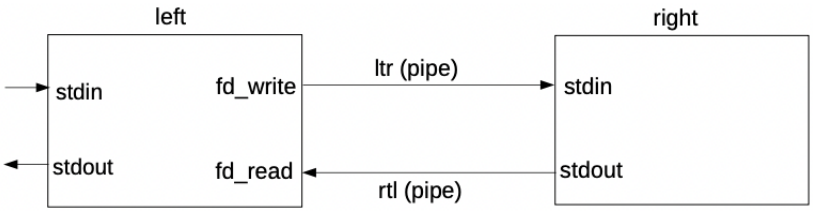
```
myprog : myprog.o line.o solve.o
  gcc -g $^ -o $@

%.o : %.c
  gcc -g -c $<

myprog.o : line.h solve.h
solve.o : solve.h line.h
line.o : line.h
```

Makefile

6. In this question, you will implement in Q6.c (starter provided) setting up the processes and pipes in this diagram (there is also a parent process lurking, not shown):



The right process will eventually exec a program that isn't aware of this setup except to read from stdin and write to stdout. Therefore before exec, you will set up stdin to be the read end of the ltr pipe, stdout to be the write end of the rtl pipe.

The left process won't exec, instead call

`void do_left(int fd_read, int fd_write);`

Therefore you will call `do_left` so that `fd_read` is the read end of the rtl pipe, `fd_write` is the write end of the ltr pipe. Note that the left process inherits stdin and stdout from the parent, and `do_left` can perform I/O on all 4 distinct FDs.
Both left and right are child processes of a common parent. The code is structured as follows:

```
void leftright(void)
{
    // parent begins
    int ltr[2];
    int rtl[2];

    pipe(ltr);
    pipe(rtl);

    pid_t left = fork();
    if (left == 0) {
        // left child code , part (a)
    } else {
        pid_t right = fork();
        if (right == 0) {
            // right child code , part (b)
            execlp("right", "right", (char*)NULL);
            exit(1);
        } else {
            // parent code after both forks, part (c)
            wait(NULL);
            wait(NULL);
        }
    }
}
```

(a) Complete the code for the left child.

```
// left child code , part (a)
close(rtl[1]);
int fd_read = rtl[0];   // Assign reading end to fd_read
close(ltr[0]);
int fd_write = ltr[1];  // Assign writing end to fd_write
do_left(fd_read, fd_write);
```

(b) Complete the code for the right child.

```
// right child code , part (b)
close(ltr[1]);  // Close unnecessary writing end
dup2(ltr[0], STDIN_FILENO);
close(ltr[0]);  // Close duplicate fd
close(rtl[0]);  // Close unnecessary reading end
dup2(rtl[1], STDOUT_FILENO);
close(rtl[1])   // Close duplicate fd
execlp("right", "right", (char*)NULL);
exit(1);
```

(c) Complete the code for the parent after forking and before waiting.

```
// parent code after both forks, part (c)
close(ltr[0]);  // Closing pipes because parent doesn't use them
close(ltr[1]);
close(rtl[0]);
close(rtl[1]);
wait(NULL);
wait(NULL);
```

7. The `do_left` from the last question
   `void do_left(int fd_read, int fd_write);`

   actually does this job:

   - Copy data from `fd_read` to stdout, verbatim.
   - Copy data from stdin to `fd_write` **but** change all lower case letters to upper case (in the sense of `toupper`).

   Since there are two data sources with unknown arrival times, `select` is a simple way to wait for data, given that the two input FDs won't change.

   Implement `do_left`. Use buffer sizes of 512 bytes. You may assume that `write` is successful and doesn't block. You may assume `fd_read` $\neq 0$. When any of the input sources gives EOF, call `exit(0)`.

```c
#include <stdio.h>
#include <sys/select.h>
#include <unistd.h>
#include <ctype.h>

void do_left(int fd_read, int fd_write)
{
    #define BUF_SIZE 512
    char buf[BUF_SIZE];
    int read_val;
    fd_set read_fds;
    for(;;) {
        FD_ZERO(&read_fds);
        FD_SET(STDIN_FILENO, &read_fds);
        FD_SET(fd_read, &read_fds);
        if (select(fd_read + 1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(1);
        }
        if (FD_ISSET(fd_read, &read_fds)) {     // fd_read ready for reading
            // EOF or Error
            if (((read_val = read(fd_read, buf, BUF_SIZE)) == 0) || (read_val == -1)) {
                exit(0);
            }
            // No error checking because assuming write is successful
            write(STDOUT_FILENO, buf, read_val);
        }
        else if (FD_ISSET(STDIN_FILENO, &read_fds)) {
            // EOF or Error
            if (((read_val = read(STDIN_FILENO, buf, BUF_SIZE)) == 0) || (read_val == -1))
    {
                exit(0);
            }
            char charbuf;
            for (int i = 0; i < read_val; i++) {
                charbuf = (char) toupper(buf[i]);
                write(fd_write, &charbuf, 1);
            }
        }
    }
}
```