EXERCISE 1 a) (malla.c)

1. Input Argument Parsing:

- The program receives two arguments (`x` and `y`), which represent the number of rows and columns.
- The function `argCheck()` ensures the correct number of arguments (`./malla -x [value] -y [value]`) and converts the arguments to integers.

2. Tree Creation:

- The program forks a process to create the parent process. Inside this process, it loops `y` times to create columns (processes in the same row).
- Inside each of these columns, another loop is used to create rows (`x` processes), where each new process forks from the last one.

3. Process Forking:

- The `fork()` call is used multiple times to create the parent-child relationship between processes, building the tree structure.
- Each process created is either a parent or a child, based on the PID value returned by the `fork()` function.

4. Final Output:

- Once the tree is created, the program sleeps for 5 seconds to keep processes alive, allowing you to inspect the process tree using `pstree`.

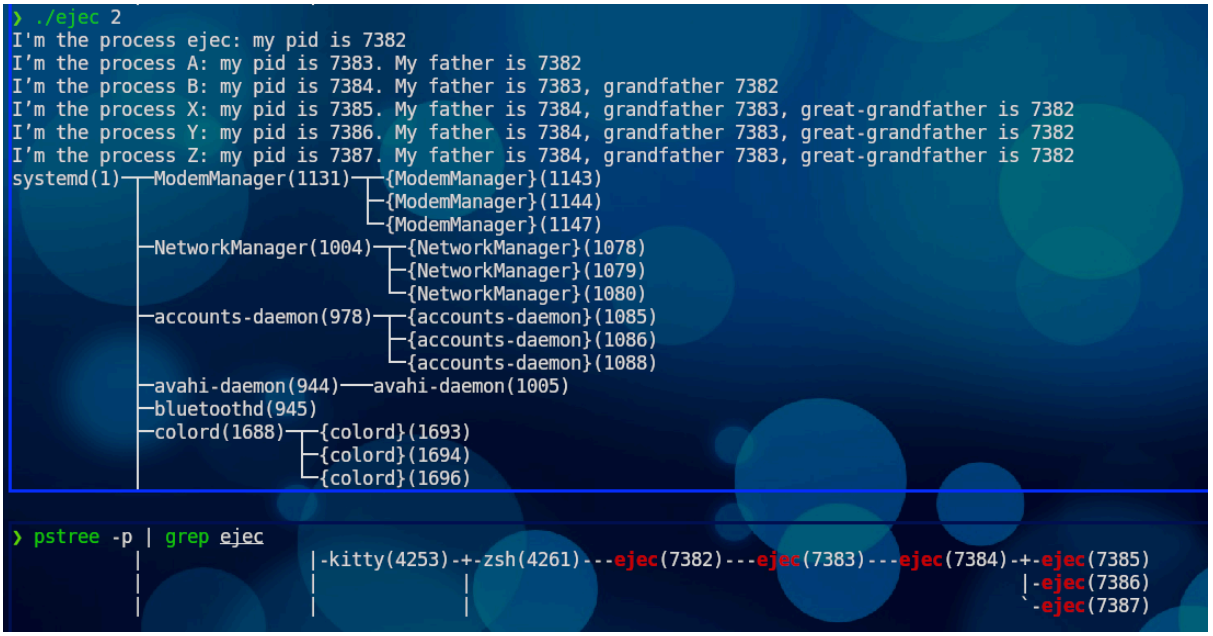
EXERCISE 1 b) (ejec.c)

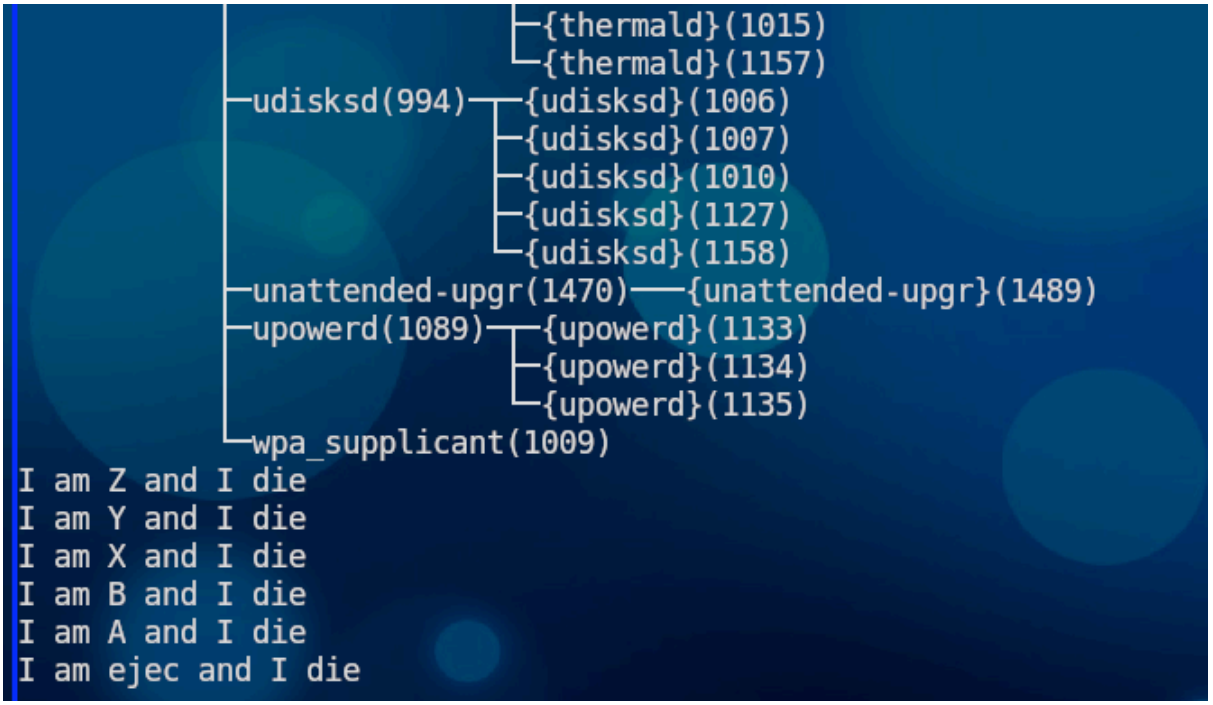
1. Input Parsing:
   - The program accepts a single argument representing the number of seconds before executing the alarm, verified by `argCheck()` and `secondChecker()`.

2. Process Tree Creation:
   - The program uses `fork()` calls to create processes `A`, `B`, `X`, `Y`, and `Z`, establishing their parent-child relationships.
   - PIDs are printed for each process to show their hierarchy.

3. **Signal Handling and Alarm:**
   - Process `Z` sets up a signal handler for `SIGALRM` with the `handle_alarm()` function.
   - The `alarm()` system call is used to schedule the `SIGALRM` signal after the specified number of seconds.
   - When the alarm goes off, `handle_alarm()` is triggered, forking a child process to execute the `pstree` command and display the tree.

4. Termination:
   - After the alarm, processes `Z`, `Y`, `X`, `B`, `A`, and `ejec` are terminated sequentially, ensuring that parents do not die before their children.




├─{thermald}(1015)
└─{thermald}(1157)
├─udisksd(994)─┬─{udisksd}(1006)
├─{udisksd}(1007)
├─{udisksd}(1010)
├─{udisksd}(1127)
└─{udisksd}(1158)
├─unattended-upgr(1470)──{unattended-upgr}(1489)
├─upowerd(1089)─┬─{upowerd}(1133)
├─{upowerd}(1134)
└─{upowerd}(1135)
└─wpa_supplicant(1009)
I am Z and I die
I am Y and I die
I am X and I die
I am B and I die
I am A and I die
I am ejec and I die

EXERCISE 3 (copiar.c)

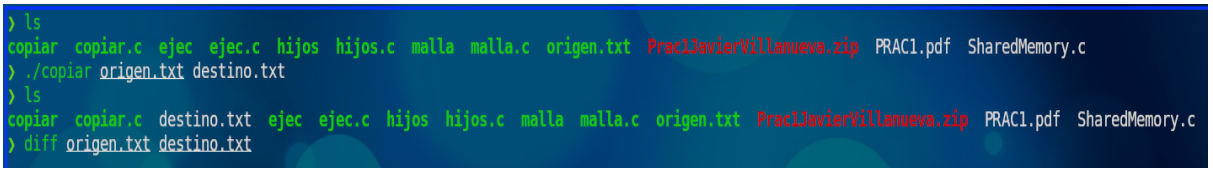
1. Input Validation:
   - The program expects two arguments: the source file and the destination file. This is verified by `argCheck()`.

2. Pipe Setup:
   - A pipe is created using the `pipe()` function, which gives two file descriptors: one for reading and one for writing.
   - A `fork()` call creates a child process.

3. Parent Process (Reading):
   - The parent process opens the source file for reading and reads the file contents into a buffer.
   - It then writes the buffer contents to the pipe (writing end).

4. Child Process (Writing):
   - The child process closes the writing end of the pipe and opens the destination file for writing.
   - It reads the data from the pipe (reading end) and writes it to the destination file.

5. Synchronization:
   - The parent waits for the child to finish writing using `wait()` to ensure that both processes complete successfully.



EXERCISE 3 (hijos.c)

1. Argument Parsing:
   - The program accepts two arguments: `x` (number of child processes) and `y` (number of sub-child processes). These are validated using `argCheck()`.

2. Shared Memory Setup:
   - The program uses `shmget()` to allocate a shared memory segment to store the PIDs of all processes. The size of the segment is proportional to the total number of processes (`x + y + 1`).
   - The `shmat()` function attaches this shared memory to the processes.

3. Tree Creation:
   - A loop creates `x` child processes, each storing its PID in the shared memory.
   - The last child process creates `y` sub-child processes, each also storing their PID in the shared memory.

```
// Create x child processes (level 1 of the tree)
for (int i = 1; i <= x; i++) {
    pid = fork();  // Create child process
    switch (pid) {
        case -1:    // Error case
            perror("ERROR: Couldn't create process\n");
            exit(1);  // Exit if fork fails
            break;

        case 0:     // Inside child process
            pidList[i] = getpid();  // Store the current child's PID
            if (i == x) {  // Last child has to have more child processes
                for (int j = 1; j <= y; j++) {
                    pid = fork();
                    switch (pid) {
                        case -1:    // Error case
                            perror("ERROR: Couldn't create subchild process\n");
                            exit(1);  // Exit if fork fails
                            break;

                        case 0:     // Inside subchild process
                            pidList[x + j] = getpid();  // Store the subchild's PID
                            printf("I am the subchild %d, my parents are: ", getpid());
                            for (int k = 0; k <= x; k++) {
                                printf("%d", pidList[k]);
                                if (k < x && pidList[k+1]!=0 && pidList[k+1]!=getpid())
                                    printf(", ");
                                else
                                    break;
                            }
                            printf("\n");
                            exit(0);  // Subchild process exits after printing its message
                            break;

                        default:    // Inside parent process
                            wait(NULL);  // Parent waits for each subchild to finish
                            break;
                    }
                }
            }
            exit(0);  // Child process exits after creating subchildren
            break;

        default:    // Inside parent process
            wait(NULL);  // Parent waits for each child to finish
            break;
    }
}
```

4. Process Information:

- Each sub-child process prints its PID and the PIDs of all its ancestors (stored in shared memory).

5. Shared Memory Detachment and Cleanup:

- After the process tree is created, the super-parent (first process) detaches the shared memory with `shmdt()` and removes it using `shmctl()` to avoid memory leaks.

```
if (getpid() == pidList[0]) {   // In the superparent process, print information about the final children
    printf("I am the superfather (%d), my final children are: ", getpid());
    for (int i = x + 1; i <= x + y; i++) {
        printf("%d", pidList[i]);
        if (i < x + y)
            printf(", ");
    }
    printf("\n");
}

// Only one process can destroy the shared memory (otherwise an error appears), and it will be the parent
if (getpid() == pidList[0]) {
    // Detach shared memory from the superfather process
    if (shmdt(pidList) == -1) {
        perror("ERROR: Couldn't detach shared memory\n");
        exit(1);
    }

    // Destroy the shared memory segment
    if (shmctl(shmid, IPC_RMID, NULL) == -1) {
        perror("ERROR: Couldn't destroy shared memory\n");
        exit(1);
    }
}
```