A.) Raul Perez-Lopez (811921958)
B.) The program has two mandatory inputs:
   a. The number of processes
   b. The file being processed

Each unique process is represented using the **plist_t** struct provided

```
typedef struct {
    int pid;
    long offset;
    long lengthAssigned;
    int pipefd[2];
} plist_t;
```

I create a list of (number of processes) to store the processes information. Then, in the initial for loop I do the following:
   c. Compute the offset for each process
   d. Assign the length to that process

The lengthAssigned for each process is calculated by dividing the size of the file being processed by the number of processes we are using.

```
fseek(fp, 0L, SEEK_END);
fsize = ftell(fp);
fclose(fp);
long sizePerProcesss = fsize / numJobs; // Work load per child
```

Then, in the initial for loop I can assign each process their offset and assigned length.

```
for(i = 0; i < numJobs; i++) {
    plist[i].offset = sizePerProcesss * i;          // Calculate offset
    plist[i].lengthAssigned = sizePerProcesss;      // assign length to process
```

If the size of the file is NOT divisible by the number of processes, I assign the last process a little extra work to process the extra bytes.

```
// Assign the rest of the work (remainder) to last job if needed
if ((i == numJobs - 1) && remainder) {
    plist[i].lengthAssigned += remainder;
}
```

Where remainder here is the size of the file modulo with the number of processes.

I then set the pipe, using the **pipefd** variable like so:

```
// Set pipe
if (pipe(plist[i].pipefd) == -1) {
    perror("Error creating pipe.");
    exit(EXIT_FAILURE);  // Exit with failure status
}
```

After assigning this information, I use fork() to launch a child process -> process its “workload” using the provided wordcount function -> if successful, save the results to the pipe (fd[1]).

```
} else if(pid == 0) { // Child
    fp = fopen(argv[2], "r");
    count = word_count(fp, plist[i].offset, plist[i].lengthAssigned);

    // send the result to the parent through pipe
    write(plist[i].pipefd[1], &count, sizeof(count_t)); // Write result to pipe
    close(plist[i].pipefd[1]); // Close write end of pipe
    fclose(fp);
    return 0;
```

This approach assigns equal units of “workload” to each individual process, with the last process having an extra (N – 1) extra units in the worst case.

For IPC, as previously stated I use a pipe. Where the child process writes to the write end of the pipe (fd[1]) and the parent reads from the read end of the pipe (fd[0]).

C.) For crash handling, we make sure that after launching the children processes that the parent waits for each individual process and ensures they terminated normally. I use a for loop, getting the PID for each child process from the **plist_t** list and checking the status using **WIFSIGNALED(status).** While True, then we continue to launch a child process to make sure it correctly processes its “workload”.

```
// While the child process continues to exit irregularly
while (WIFSIGNALED(status)) {

    // Launch new process to replace this
    pid = fork();
    if (pid < 0) printf("Fork failed.\n");
    else if (pid == 0) {
        fp = fopen(argv[2], "r");
        count = word_count(fp, plist[i].offset, plist[i].lengthAssigned);

        write(plist[i].pipefd[1], &count, sizeof(count_t)); // Write result to pipe
        close(plist[i].pipefd[1]);  // Close write end of pipe
        fclose(fp);
        return 0;
    }
    plist[i].pid = pid;
    waitpid(plist[i].pid, &status, 0);
}
```

Once the child terminated successfully, we can read the result from the pipe and add it to our total.

```
read(plist[i].pipefd[0], &result, sizeof(count_t)); // Parent reads result written by child
close(plist[i].pipefd[0]);  // Close read end of pipe

// Add results to total
total.linecount += result.linecount;
total.wordcount += result.wordcount;
total.charcount += result.charcount;
```