PA #1: Multi-Process and IPC

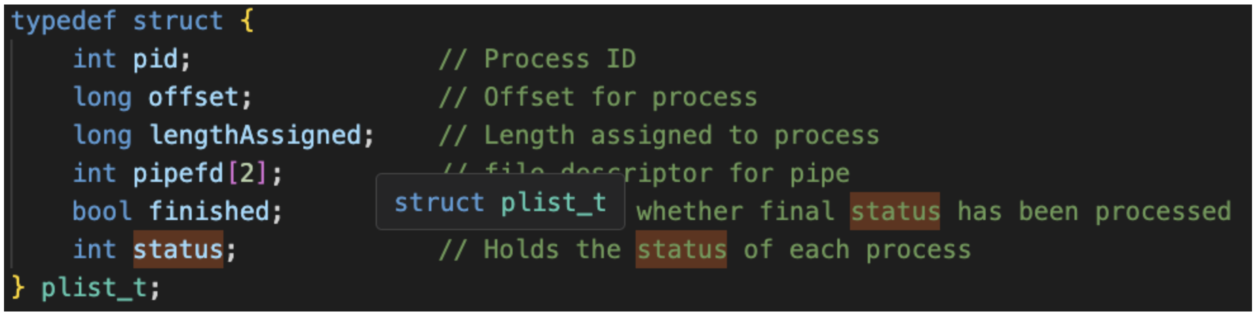
a. Raul Perez-Lopez

b. The design of my multi-processed structure is one parent thread which takes in two inputs from the user:
   a. The number of processes to use
   b. The file to process

With that information, the parent process can calculate the amount of “**workload**” for each thread and the “**offset**” for each thread.

The workload is calculated by getting the size of the file, let it be called **K** and dividing K by the number of processes. So, the workload for each thread is calculated by the formula **K / (# of processes).**

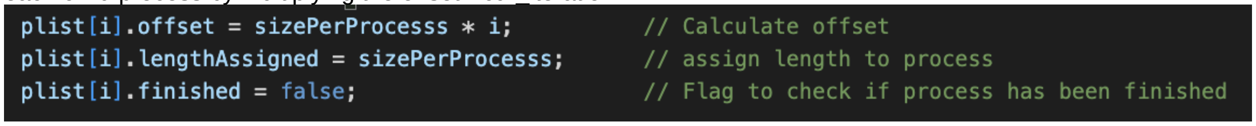
I store the information for each child process in the plist_t struct, which I modified to contain the following information:



```
typedef struct {
    int pid;                    // Process ID
    long offset;                // Offset for process
    long lengthAssigned;        // Length assigned to process
    int pipefd[2];              // file descriptor for pipe
    bool finished;       struct plist_t  whether final status has been processed
    int status;                 // Holds the status of each process
} plist_t;
```

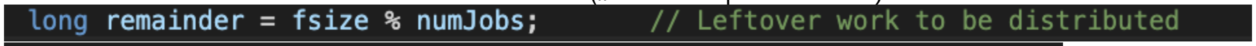
I used an array of this struct where each index “i” represents the ith process.

I use a for loop to launch asynchronously launch all the children processes, and I calculate the **offset** of each child process by multiplying the offset * cur_iteration.

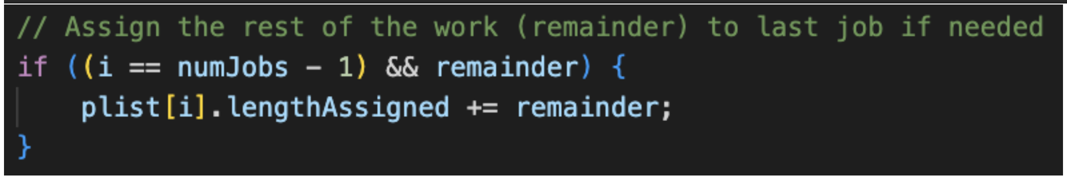


```
plist[i].offset = sizePerProcesss * i;          // Calculate offset
plist[i].lengthAssigned = sizePerProcesss;      // assign length to process
plist[i].finished = false;                      // Flag to check if process has been finished
```

In the case that the file is NOT divisible by the number of processes, I add the “extra work” the last process. Where the most amount of “extra work” that it will do is (#number of processes – 1).

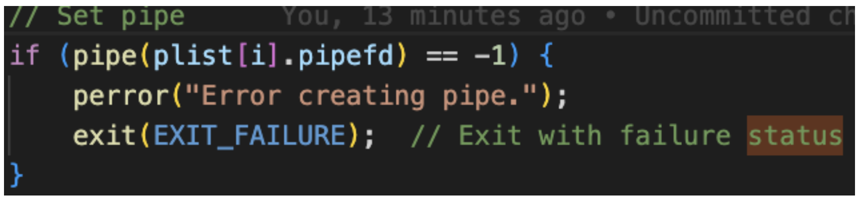


```
long remainder = fsize % numJobs;         // Leftover work to be distributed
```

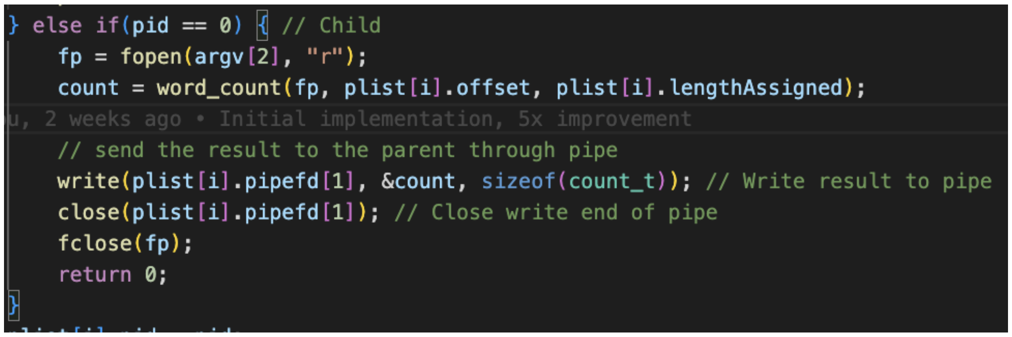


```
// Assign the rest of the work (remainder) to last job if needed
if ((i == numJobs - 1) && remainder) {
    plist[i].lengthAssigned += remainder;
}
```

The **communication** is set up using pipes. I set the pipe before launching the child thread and I write the result to the pipe if the child thread was successful. Later in the processing step I will use the parent thread to read from the pipe.



```
// Set pipe       You, 13 minutes ago • Uncommitted ch
if (pipe(plist[i].pipefd) == -1) {
    perror("Error creating pipe.");
    exit(EXIT_FAILURE);  // Exit with failure status
}
```



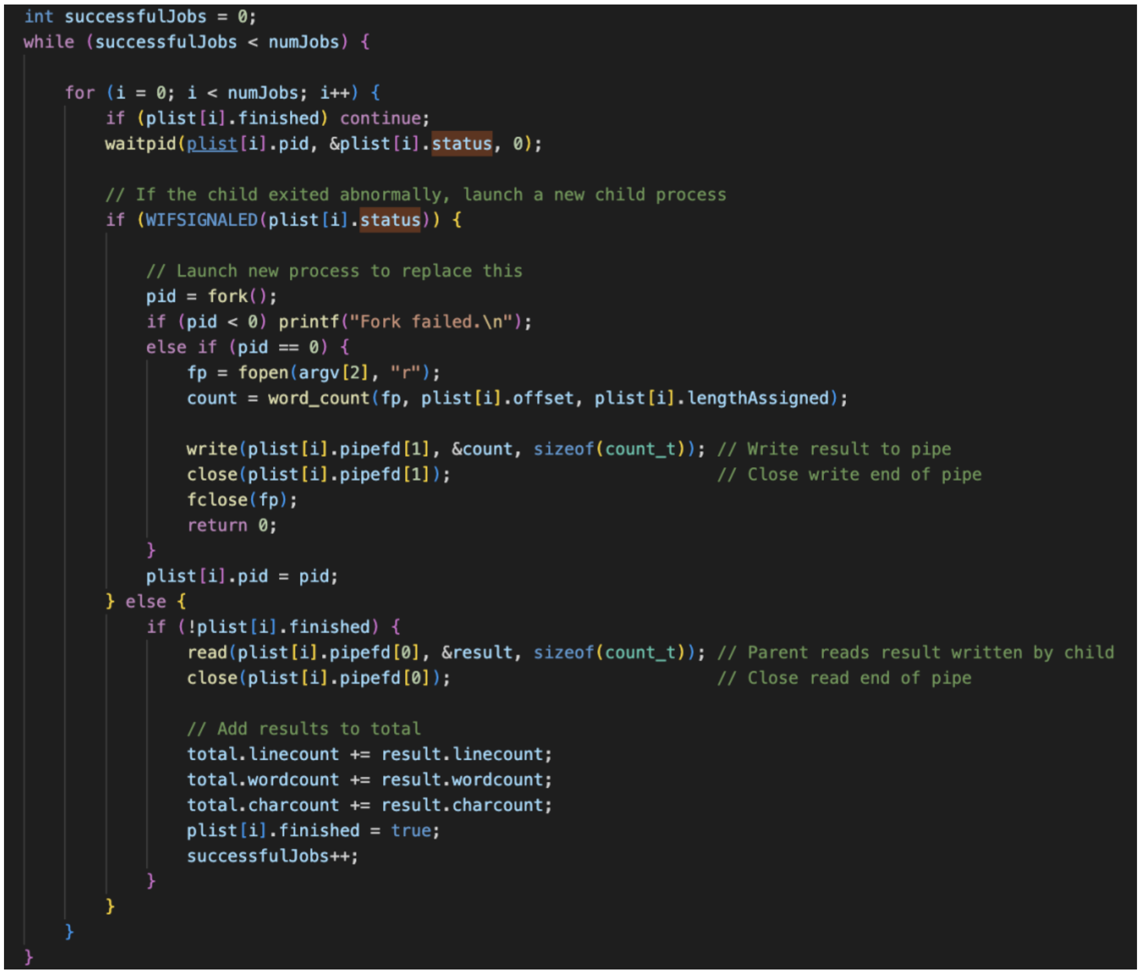
```
} else if(pid == 0) { // Child
    fp = fopen(argv[2], "r");
    count = word_count(fp, plist[i].offset, plist[i].lengthAssigned);
u, 2 weeks ago • Initial implementation, 5x improvement
    // send the result to the parent through pipe
    write(plist[i].pipefd[1], &count, sizeof(count_t)); // Write result to pipe
    close(plist[i].pipefd[1]); // Close write end of pipe
    fclose(fp);
    return 0;
}
```

This approach allows each thread to process their fair share of the file and sends the result to the parent using pipes. In the next section I will show how the results are processed and how the parent reads the result from the children processes.

C. There is a 3rd optional input from the program which sets the crash rate of each child process. This means that on the initial launch of the child process the result may not have been attained and therefore that section of the file needs to be re-processed.

After launching the initial **numJobs** processes, I check each process and their status. If any of them failed, i must launch another thread to process the portion of the file not processed. To do this asynchronously, i maintain a variable called **successfulJobs** which maintains the count of unique jobs that successfully finished. While successful Jobs < numJobs, i loop through our array of processes and see check their status. In a perfect world, i only loop through this array once but if at least one child processed failed then the number of successful jobs will be < numJobs after the first iteration.

When we find a process that has failed, we fork a new process to handle the portion which failed and continue. Else we read the results from their respective file descriptor and add it to our final result.



```
int successfulJobs = 0;
while (successfulJobs < numJobs) {

    for (i = 0; i < numJobs; i++) {
        if (plist[i].finished) continue;
        waitpid(plist[i].pid, &plist[i].status, 0);

        // If the child exited abnormally, launch a new child process
        if (WIFSIGNALED(plist[i].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;
        } else {
            if (!plist[i].finished) {
                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;
                plist[i].finished = true;
                successfulJobs++;
            }
        }
    }
}
```