1η Εργαστηριακή Αναφορά
Μέλη Ομάδας:
Σοφία Σάββα **ΑΜ:**03121189
Ιωάννης Πολυχρονόπουλος **ΑΜ:**03121089

1. Ανάγνωση και εγγραφή αρχείων στη C και με τη βοήθεια κλήσεων συστήματος

Κλήσεις συστήματος και βιβλιοθήκες της C

Τα *system calls* είναι διεπαφές (interfaces) που παρέχονται από το λειτουργικό σύστημα στις εφαρμογές για να επικοινωνήσουν με τον πυρήνα του λειτουργικού συστήματος. Αυτές οι διεπαφές επιτρέπουν στις εφαρμογές να εκτελούν διάφορες λειτουργίες όπως ανάγνωση και εγγραφή αρχείων, διαχείριση διεργασιών, επικοινωνία με άλλες διεργασίες και πολλά άλλα.

Η διαφορά μεταξύ της χρήσης *system calls* και της χρήσης μιας *βιβλιοθήκης της C* για ανάγνωση και εγγραφή αρχείων είναι ότι οι system calls είναι πιο κοντά στο hardware και προσφέρουν πιο απευθείας πρόσβαση στις λειτουργίες του λειτουργικού συστήματος. Από την άλλη πλευρά, οι βιβλιοθήκες της C παρέχουν μια πιο αφαιρετική και ευκολότερη στη χρήση διεπαφή για τις ίδιες λειτουργίες.

Ζητούμενο

Μας ζητείται να εκτελέσουμε την καταμέτρηση ενός συγκεκριμένου χαρακτήρα από ένα αρχείο και την αποτύπωση του αποτελέσματος σε ένα αρχείο εξόδου με την χρήση system calls έναντι της βιβλιοθήκης της C.

Πρόγραμμα

Για να γίνει η υλοποίηση του κώδικα που δίνεται με χρήση system calls είναι σημαντικό να κατανοήσουμε τις συναρτήσεις που χρησιμοποιούνται και τα ορίσματά τους.

1. ***open***: Η συνάρτηση *open* χρησιμοποιείται για το άνοιγμα ενός αρχείου και επιστρέφει έναν αριθμό αναγνώρισης του αρχείου (file descriptor) εάν επιτύχει. Εάν αποτύχει, επιστρέφει -1
   Σύνταξη: `open(const char *path, int flags, mode_t mode);`

2. ***read***: Η συνάρτηση *read* χρησιμοποιείται για την ανάγνωση δεδομένων από ένα αρχείο που έχει ήδη ανοίξει με τη συνάρτηση open. Επιστρέφει τον αριθμό των bytes που διαβάστηκαν, εφόσον είναι επιτυχής, ενώ εάν αποτύχει επιστρέφει -1.
   Σύνταξη: `ssize_t read(int fd, void *buf, size_t count);`

3. ***write***: Η συνάρτηση *write* χρησιμοποιείται για την εγγραφή δεδομένων σε ένα αρχείο που έχει ήδη ανοίξει με τη συνάρτηση open. Επιστρέφει τον αριθμό των bytes που εγράφησαν, εφόσον είναι επιτυχής, ενώ εάν αποτύχει, επιστρέφει -1.
   Σύνταξη: `ssize_t write(int fd, const void *buf, size_t count);`

Σημείωση: *Οι παράμετροι των συναρτήσεων αναλύονται ως εξής:*

- *Το **fd** είναι ο αριθμός αναγνώρισης του αρχείου που έχει ήδη ανοίξει με τη συνάρτηση open.*
- *Το **buf** είναι ένας δείκτης σε ένα buffer όπου θα αποθηκευτούν ή ανακτηθούν τα δεδομένα.*
- *Το **count** είναι ο αριθμός των bytes που πρόκειται να διαβαστούν ή γραφτούν.*
- *Οι παραμέτροι **flags** και **mode** της συνάρτησης open χρησιμοποιούνται για τον έλεγχο της λειτουργίας και των δικαιωμάτων πρόσβασης του αρχείου αντίστοιχα.*

Εφόσον, αναλύσαμε και κατανοήσαμε πως λειτουργούν οι παραπάνω συναρτήσεις *system calls* πάμε τώρα να τις εφαρμόσουμε στο κώδικά μας και να αντικαταστήσουμε τις συναρτήσεις που πηγάζουν από την βιβλιοθήκη της C.

- *Άνοιγμα των δυο αρχείων για ανάγνωση και για εγγραφή*

Αρχικά, ανοίγουμε τα δύο αρχεία, το ένα για ανάγνωση και το άλλο για εγγραφή, με την κλήση συστήματος *open*, αντικαθιστώντας την *fopen.*

Στη πρώτη κλήση της *fopen*, ανοίγουμε το αρχείο με κωδικό mode *"r"*, δηλαδή θα ανοίξει για ανάγνωση μονο, ενώ παράλληλα το αρχείο πρέπει να υπάρχει στο σύστημά μας (filesystem). Για την αντίστοιχη υλοποίηση με **system calls** θα κληθεί η *open* με ορίσματα τη διαδρομή (pathname) του αρχείου ανάγνωσης , που θα βρίσκεται ως το δεύτερο όρισμα κατά την εκτέλεση του προγράμματος *argv[1]* (με την προϋπόθεση πως το πρώτο όρισμα είναι το όνομα του εκτελέσιμου) και με flag το O_RDONLY. Σε περίπτωση απρόβλεπτου λάθους η *open* θα επιστρέψει -1 και ενημερώνουμε τον χρήστη με μήνυμα σφάλματος για το αντίστοιχο σφάλμα κατά το άνοιγμα του αρχείου και τερματίζεται άμεσα η διαδικασία με τη κλήση συστήματος *exit* με κατάσταση EXIT_FAILURE (το EXIT_FAILURE είναι ένας προκαθορισμένος κωδικός που υποδηλώνει αποτυχία, όπως αντίστοιχα το -1 στη δήλωση *return -1*).

Στη δεύτερη κλήση της *fopen*, ανοίγουμε το αρχείο με κωδικό mode *"w+"*, δηλαδή το αρχείο θα ανοίξει για εγγραφή (write). Αν το αρχείο υπάρχει, το περιεχόμενό του θα διαγραφεί πριν αρχίσει η εγγραφή νέων δεδομένων, ενώ εάν δεν υπάρχει θα δημιουργηθεί εκ νέου. Επομένως, με **system calls** καλούμε την *open* με διαδρομή το τρίτο όρισμα του πίνακα *argv[]*, flags O_WRONLY|O_CREAT|O_TRUNC και mode S_IRUSR|S_IWUSR. Τα αντίστοιχα modes δίνουν read και write δικαιώματα στον χρήστη για το file που θα δημιουργηθεί εάν το αρχείο δεν υπάρχει στο σύστημα (η *fopen* πετυχαίνει αυτό το κομμάτι με το κωδικό "+"). Και πάλι αν προκύψει λάθος η *open* θα επιστρέψει κωδικό σφάλματος και θα τερματιστεί η διαδικασία με τον τρόπο που αναφέρθηκε προηγουμένως.

- *Ανάγνωση και εγγραφή στα δυο αρχεία*

Θα αντικαταστήσουμε τώρα την συνάρτηση *fgetc*, όπου διαβάζει ένα χαρακτήρα κάθε φορά από το αρχείο ανάγνωσης, καλώντας την *read*. Για να μην διαβάζουμε έναν χαρακτήρα την φορά και σπαταλάμε χρόνο, ορίζουμε έναν *char buffer* με 4096 (η χρήση ενός buffer μεγέθους $4096 = 2^{12}$ bytes είναι μια συνηθισμένη επιλογή καθώς θεωρείται συχνά μια καλή ισορροπία μεταξύ χρήσης μνήμης και απόδοσης I/O), στη συνέχεια καλούμε την *read* με ορίσματα *fpr* (το file descriptor που επέστρεψε η open) , *buffer* και *sizeof(buffer)*. Αποθηκεύουμε τον αριθμό των byte που διαβάστηκαν στην μεταβλητή *n*, έπειτα συγκρίνουμε όσους χαρακτήρες διαβάσαμε με τον επιθυμητο χαρακτήρα και όπου ταυτίζονται αυξάνουμε και τον *counter*. Την παραπάνω διαδικασία ανάγνωσης την τοποθετούμε σε έναν φαύλο βρόχο (while loop), μέχρι να διαβάσει όλους τους χαρακτήρες του αρχείου. Όσο επιστρέφει η *read* θετικό αριθμό συνεχίζω την ανάγνωση ενώ όταν επιστρέψει 0 σταματάω και αν επιστρέψει -1 δηλώνω πως υπήρξε σφάλμα στην ανάγνωση του αρχείου.

Αφού διαβάσουμε τα περιεχόμενα του φακέλου κλείνουμε το file descriptor *fpr*.

```
// close the file for reading
    if (close(fpr) == -1)
    {
        perror("Problem closing file");
        exit(EXIT_FAILURE);
    }
```

Ύστερα, για την εγγραφή του αποτελέσματος στο αρχείο εξόδου θα αξιοποιήσουμε την συνάρτηση *write* έναντι της *fprintf*. Αρχικά, χρησιμοποιώντας την *snprintf* αποθηκεύουμε στο πίνακα χαρακτήρων *result* (που λειτουργεί ως array buffer) το κείμενο που θέλουμε να τυπώσουμε. Για να γίνει η εγγραφή καλούμε την write με ορίσματα τον file descriptor του αρχείου εξόδου *(fpw)*, τον buffer από τον οποίο θα γράψω *(result)* και ο αριθμός των bytes που θα γραφτούν. Στη περίπτωση που δεν γραφτούν όλα τα bytes, τότε επιστρέφω σφάλμα και τερματίζεται η διαδικασία.

Αφού γίνει η εγγραφή κλείνει το file descriptor *fpw* και τερματίζεται επιτυχώς η διαδικασία με exit με status EXIT_SUCCESS.

```
  // close the output file
    if (close(fpw) == -1)
    {
        perror("Problem closing file");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
```

Ο ολοκληρωμένος κώδικας που χρησιμοποιήθηκε είναι ο παρακάτω:

```
-----------------------------------------------------------------------------------------------
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    int fpr, fpw;
    char cc, c2c = 'a';
    int count = 0;
```

```c
    // Check if the correct number of command-line arguments is provided
    if (argc != 4)
    {
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // open file for reading only
    if ((fpr = open(argv[1], O_RDONLY)) == -1)
    {
        perror("Problem opening file to read");
        exit(EXIT_FAILURE);
    }

    // open file for writing the result. If the file doesn't exist create it
    if ((fpw = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1)
    {
        perror("Problem opening file to write");
        exit(EXIT_FAILURE);
    }

    // character to search for (third parameter in command line)
    c2c = argv[3][0];

    // count the occurrences of the given character
    char buffer[4096];
    while (1)
    {
        ssize_t n = read(fpr, buffer, sizeof(buffer));
        if (n == -1)
        {
            perror("Error reading file");
            exit(EXIT_FAILURE);
        }
        else if (n == 0)
            break;
        else
            for (int i = 0; i < n; i++)
                if (buffer[i] == c2c)
                    count++;
    }

    // close the file for reading
    if (close(fpr) == -1)
    {
        perror("Problem closing file");
        exit(EXIT_FAILURE);
    }

    // write the result in the output file
    char result[150];
    snprintf(result, sizeof(result), "The character '%c' appears %d times in file %s.\n", c2c, count,
argv[1]);
    if (write(fpw, result, strlen(result)) != strlen(result))
    {
        perror("Problem writing to file");
        exit(EXIT_FAILURE);
    }

    // close the output file
    if (close(fpw) == -1)
    {
        perror("Problem closing file");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

Επισήμανση: *Έχουμε εφαρμόσει σε αυτή την άσκηση αλλά και στις επόμενες την τακτική να χειριζόμαστε πιθανά λάθη κατά την κλήση συναρτήσεων με αβέβαιη κατάληξη.*

2. Δημιουργία διεργασιών

Ερώτημα 1

Επεκτείνουμε τον κώδικα της άσκησης 1 με την δημιουργία μιας νέας διεργασίας παιδί αξιοποιώντας την συνάρτηση *fork()*. Όταν η συνάρτηση *fork()* καλείται από μια διεργασία, δημιουργείται ένα ακριβές αντίγραφο της διεργασίας αυτής. Αυτό το νέο αντίγραφο περιέχει όλες τις μεταβλητές, την κατάσταση και τον κώδικα της αρχικής διεργασίας. Κατά την κλήση της, η fork() επιστρέφει στη γονική διεργασία τον αριθμό PID (Process ID) του παιδιού, ενώ στο παιδί επιστρέφεται η τιμή 0. Εάν επιστραφεί στη γονική διεργασία αρνητική τιμή , τότε υπήρξε σφάλμα κατά την κλήση της fork() και δεν δημιουργείται παιδί. Συνεπώς, δημιουργούμαι ένα if statement για να διαχειριστούμε κάθε περίπτωση. Αποθηκεύουμε ότι επιστρέφει η fork() στη μεταβλητή `pid_t pid = fork();` και έπειτα ελέγχουμε αντίστοιχα την τιμή της.

`if (pid < 0):` Επιστρέφουμε πως υπήρξε σφάλμα κατά την δημιουργία του παιδιού και τερματίζουμε το πρόγραμμα με status EXIT_FAILURE.

`else if (pid == 0):` Βρισκόμαστε στην διεργασία παιδί, επομένως χαιρετάμε τον χρήστη και θα αναφέρουμε το αναγνωριστικό του και το αναγνωριστικό του γονέα του με τις συναρτήσεις *getpid()* και *getppid()* αντίστοιχα.

`else:` Βρισκόμαστε στη γονική διεργασία, συνεπώς εκτυπώνουμε το αναγνωριστικό του παιδιού μέσω της μεταβλητής *pid* και περιμένουμε να τερματιστεί η διαδικασία παιδί με την *waitpid()* (χρησιμοποιούμε την *waitpid()* εναντι της *wait()* επειδή παρέχει περισσότερες δυνατότητες και ευελιξία στον έλεγχο των διεργασιών παιδιών). Ο λόγος που περιμένουμε να τερματιστεί η διεργασία παιδί είναι για την αποφυγή διεργασίας zombie, δηλαδή η διεργασία παιδί να παραμείνει στο πίνακα διεργασιών του συστήματος. Καλούμε την waitpid() με τα ακόλουθα ορίσματα → `pid_t wpid = waitpid(pid, &status, 0);`

- ***pid:*** Αυτό είναι το PID της διεργασίας που περιμένουμε να ολοκληρωθεί.
- ***status:*** Ένας δείκτης σε μια μεταβλητή όπου θα αποθηκευτεί ο κωδικός εξόδου της ολοκληρωμένης διεργασίας
- ***0:*** Η waitpid() αναμένει για οποιαδήποτε ολοκληρωμένη διεργασία παιδί.

Ύστερα καλούμε την explain_wait_status, όπως διατυπώνεται και στις διαφάνειες, για να εξηγήσει στον χρήστη τον τρόπο με τον οποίο τερματίστηκε το παιδί. Η συνάρτηση explain_wait_status που παίρνει σαν όρισμα το pid του παιδιού που άλλαξε και το status αυτό. Η συνάρτηση ανάλογα με τον τρόπο που τερματίστηκε το παιδί το pid του οποίου της δίνεται σαν όρισμα εκτυπώνει μήνυμα που εξηγεί τον τρόπο αυτό στον χρήστη. Συγκεκριμένα ανάλογα με το ποιό απο τα macros WIFEXITED, WIFSIGNALED, WIFSTOPPED, WIFCONTINUED είναι αληθές, σύμφωνα με το status του παιδιού, ο χρήστης ενημερώνεται για τον κανονικό τερματισμό του παιδιού, για τον τερματισμό του απο κάποιο σήμα, για την παύση του από ένα σήμα και για το αν συνεχίστηκε η εκτέλεση του. Τέλος, η γονική διεργασία εκτελεί τον υπόλοιπο κώδικα της προηγούμενης άσκησης.

Το αποτέλεσμα στο τερματικό είναι το εξής:

```
└─$ ./2nd_exercise-1 test.txt output.txt a
Parent process: Child ID = 15314
Child process: ID = 15314, Parent ID = 15313
Child process 15314 terminated normally with exit status: 0
```

Ο ολοκληρωμένος κώδικας που χρησιμοποιήθηκε είναι ο παρακάτω:

```
-----------------------------------------------------------------------------------------------------
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
void explain_wait_status(pid_t wpid, int status){
    if (WIFEXITED(status))
        printf("Child process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("Child process %ld terminated by signal: %d\n", (long)wpid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        printf("Child process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));
    else if (WIFCONTINUED(status))
        printf("Child process %d continued\n", wpid);
    else
        printf("Unknown status for child process %d\n", wpid);
}
int main(int argc, char *argv[])
{
    int fpr, fpw;
    char cc, c2c = 'a';
    int count = 0;
    // Check if the correct number of command-line arguments is provided
    if (argc != 4)
    {
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Fork a child process
    pid_t pid = fork();
    if (pid < 0){
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0){ // Child process
        printf("Child process: ID = %d, Parent ID = %d\n", getpid(), getppid());
        exit(EXIT_SUCCESS);
    }
    else{ // Parent process
        printf("Parent process: Child ID = %d\n", pid);
        // Inside the parent process after forking the child
        int status;
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);

        // Proceed with the rest of the code
        int fpr, fpw;
        char cc, c2c = 'a';
        int count = 0;
        // open file for reading only
        if ((fpr = open(argv[1], O_RDONLY)) == -1){
            perror("Error opening input file");
            exit(EXIT_FAILURE); // Terminate the program with a failure status
        }
        // open file for writing the result. If the file doesn't exist create it
        if ((fpw = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1)
        {
            perror("Error opening output file");
            close(fpr); // Close the input file
            exit(EXIT_FAILURE);
        }

        // character to search for (third parameter in command line)
```

```
        c2c = argv[3][0];
        // count the occurrences of the given character
        char buffer[4096];
        while (1)
        {
            ssize_t n = read(fpr, buffer, sizeof(buffer));
            if (n == -1)
            {
                perror("Error reading file");
                exit(EXIT_FAILURE);
            }
            else if (n == 0)
                break;
            else
                for (int i = 0; i < n; i++)
                    if (buffer[i] == c2c)
                        count++;
        }
        // close the file for reading
        if (close(fpr) == -1)
        {
            perror("Problem closing file");
            exit(EXIT_FAILURE);
        }
        // write the result in the output file
        char result[150];
        snprintf(result, sizeof(result), "The character '%c' appears %d times in file %s.\n", c2c,
count, argv[1]);
        if (write(fpw, result, strlen(result)) != strlen(result))
        {
            perror("Problem writing to file");
            exit(EXIT_FAILURE);
        }
        // close the output file
        if (close(fpw) == -1)
        {
            perror("Problem closing file");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS); // Terminate the program with a success status
    }
    return 0;
}
```

Ερώτημα 2

Αρχικά, ορίζουμε την μεταβλητή `int = 10;` στο γονέα πρωτού δημιουργήσουμε την διεργασία παιδί. Στη συνέχεια, αλλάζουμε την τιμή της μεταβλητής μέσα στη διεργασία παιδί και στη γονική διεργασία στα αντίστοιχα if statements. Εκτυπώνουμε την τιμή της μεταβλητής σε κάθε διεργασία και έχουμε το εξής αποτέλεσμα:

```
└─$ ./2nd_exercise-2 test.txt output.txt a
Parent process before creating a child process: x = 10
Parent process: x = 30
Parent process: ID = 2019302000, Child ID = 15628
Child process: x = 20
Child process: ID = 15628, Parent ID = 15624
Child process 15628 terminated normally with exit status: 0
```

Παρατηρούμε πως η μεταβλητή `x` έχει διαφορετική τιμή σε κάθε διεργασία και παράλληλα διαφορετική τιμή με την αρχική (`int x = 10;`). Αυτό συμβαίνει καθώς η διεργασία παιδί ξεκινά με αντίγραφο της μνήμης της γονικής διεργασίας, άρα έχουν ξεχωριστό χώρο μνήμης. Συνεπώς όταν αλλάζουμε την τιμή μιας μεταβλητής σε μία από τις διεργασίες, αυτή η αλλαγή ισχύει μόνο για την συγκεκριμένη διεργασία χωρίς να επηρεάζει την άλλη.

Ο ολοκληρωμένος κώδικας που χρησιμοποιήθηκε είναι ο παρακάτω:

```
-----------------------------------------------------------------------------------------------------
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <sys/wait.h>
void explain_wait_status(pid_t wpid, int status){
    if (WIFEXITED(status))
        printf("Child process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("Child process %ld terminated by signal: %d\n", (long)wpid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        printf("Child process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));
    else if (WIFCONTINUED(status))
        printf("Child process %d continued\n", wpid);
    else
        printf("Unknown status for child process %d\n", wpid);
}

int main(int argc, char *argv[])
{
    int fpr, fpw;
    char cc, c2c = 'a';
    int count = 0;
    // Check if the correct number of command-line arguments is provided
    if (argc != 4){
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Define and set value for variable x in the parent process
    int x = 10;
    printf("Parent process before creating a child process: x = %d\n", x);
    // Fork a child process
    pid_t pid = fork();

    if (pid < 0){
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0){ // Child process
        // Change value for variable x in the child process
        x = 20;
        printf("Child process: x = %d\n", x);
        printf("Child process: ID = %d, Parent ID = %d\n", getpid(), getppid());
        exit(EXIT_SUCCESS);
    }
    else
    { // Parent process
        x = 30;
        printf("Parent process: x = %d\n", x);
        printf("Parent process: ID = %d, Child ID = %d\n", getpid, pid);
        // Wait for the child process to terminate
        int status;
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);
        // Proceed with the rest of the code
        int fpr, fpw;
        char cc, c2c = 'a';
        int count = 0;
        // open file for reading only
        if ((fpr = open(argv[1], O_RDONLY)) == -1){
            perror("Error opening input file");
            exit(EXIT_FAILURE); // Terminate the program with a failure status
        }
        // open file for writing the result. If the file doesn't exist create it
        if ((fpw = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1){
```

```
            perror("Error opening output file");
            close(fpr); // Close the input file
            exit(EXIT_FAILURE);
        }
        // character to search for (third parameter in command line)
        c2c = argv[3][0];
        // count the occurrences of the given character
        char buffer[4096];
        while (1){
            ssize_t n = read(fpr, buffer, sizeof(buffer));
            if (n == -1){
                perror("Error reading file");
                exit(EXIT_FAILURE);
            }
            else if (n == 0)
                break;
            else
                for (int i = 0; i < n; i++)
                    if (buffer[i] == c2c)
                        count++;
        }
        // close the file for reading
        if (close(fpr) == -1){
            perror("Problem closing file");
            exit(EXIT_FAILURE);
        }
        // write the result in the output file
        char result[150];
        snprintf(result, sizeof(result), "The character '%c' appears %d times in file %s.\n", c2c,
count, argv[1]);
        if (write(fpw, result, strlen(result)) != strlen(result)){
            perror("Problem writing to file");
            exit(EXIT_FAILURE);
        }
        // close the output file
        if (close(fpw) == -1){
            perror("Problem closing file");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS); // Terminate the program with a success status
    }
    return 0;
}
```

Ερώτημα 3

Επεκτείνοντας τον προηγούμενο κώδικα, θα αναθέσουμε την αναζήτηση του χαρακτήρα στη διεργασία παιδί. Αρχικά, ανοίγουμε τα αρχεία στη γονική διεργασία προτού καλέσουμε την *fork()*, ώστε να έχει πρόσβαση το παιδί στα αρχεία. Έπειτα, τοποθετούμε τον κώδικα για την αναζήτηση του χαρακτήρα και την εγγραφή του αποτελέσματος στο αρχείο εξόδου μέσα στο if statement του παιδιού. Τέλος κλείνουμε τα αρχεία στη γονική διεργασία.

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <sys/wait.h>

void explain_wait_status(pid_t wpid, int status)
{
    if (WIFEXITED(status))
        printf("Child process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));

    else if (WIFSIGNALED(status))
        printf("Child process %ld terminated by signal: %d\n", (long)wpid, WTERMSIG(status));

    else if (WIFSTOPPED(status))
```

```c
        printf("Child process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));

    else if (WIFCONTINUED(status))
        printf("Child process %d continued\n", wpid);

    else
        printf("Unknown status for child process %d\n", wpid);
}

int main(int argc, char *argv[])
{
    // Check if the correct number of command-line arguments is provided
    if (argc != 4)
    {
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Proceed with the rest of the code
    int fpr, fpw;
    char cc, c2c = 'a';
    int count = 0;

    // open file for reading only
    if ((fpr = open(argv[1], O_RDONLY)) == -1)
    {
        perror("Error opening input file");
        exit(EXIT_FAILURE);
    }

    // open file for writing the result. If the file doesn't exist create it
    if ((fpw = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1)
    {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // character to search for (third parameter in command line)
    c2c = argv[3][0];



    // Fork a child process
    pid_t pid = fork();

    if (pid < 0)
    {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    { // Child process

        // count the occurrences of the given character */
        char buffer[4096];
        while (1)
        {
            ssize_t n = read(fpr, buffer, sizeof(buffer));
            if (n == -1)
            {
                perror("Error reading file");
                exit(EXIT_FAILURE);
            }
            else if (n == 0)
                break;
            else
                for (int i = 0; i < n; i++)
                    if (buffer[i] == c2c)
                        count++;
        }

        // write the result in the output file
```

```
        char result[150];
        snprintf(result, sizeof(result), "The character '%c' appears %d times in file %s.\n", c2c,
count, argv[1]);
        if (write(fpw, result, strlen(result)) != strlen(result))
        {
            perror("Problem writing to file");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
    }
    else
    { // Parent process

        // Wait for the child process to terminate
        int status;
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);

        // close the file for reading
        if (close(fpr) == -1)
        {
            perror("Problem closing file");
            exit(EXIT_FAILURE);
        }

        // close the output file
        if (close(fpw) == -1)
        {
            perror("Problem closing file");
            exit(EXIT_FAILURE);
        }

        exit(EXIT_SUCCESS);
    }

    return 0;
}
```

Ερώτημα 4

Θα δημιουργήσουμε μια διεργασία παιδί που θα εκτελέσει το εκτελέσιμο αρχείο “a1.1-C”, με τα κατάλληλα ορίσματα, όπου μας δόθηκε από την πρώτη άσκηση. Για τον σκοπό θα αξιοποιήσουμε την συνάρτηση *execv*. Η συνάρτηση αυτή χρησιμοποιείται για να αντικαταστήσει τον τρέχοντα κώδικα ενός προγράμματος με ένα νέο πρόγραμμα, διατηρώντας το ίδιο PID και τον ίδιο χώρο μνήμης, αλλά το περιεχόμενο αλλάζει. Τα ορίσματα που δέχεται είναι δύο:

```
execv(path, argv);
```

1. ***path:*** Το μονοπάτι προς το εκτελέσιμο αρχείο που θα φορτωθεί. Αυτό πρέπει να είναι το πλήρες μονοπάτι ή ένα μονοπάτι σχετικό με τον τρέχοντα κατάλογο.
2. ***argv:*** Ένας πίνακας από συμβολοσειρές που περιέχει τα ορίσματα που θα περαστούν στο πρόγραμμα. Το πρώτο στοιχείο του πίνακα είναι τυπικά το όνομα του προγράμματος. Ο πίνακας πρέπει να τερματίζεται με μια τιμή NULL για να δηλώνει το τέλος των ορισμάτων.

Εάν η *execv* αποτύχει, επιστρέφει και δηλώνουμε την αιτία σφάλματος με την *perror* και τερματίζουμε με status EXIT_FAILURE.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
void explain_wait_status(pid_t wpid, int status){
    if (WIFEXITED(status))
        printf("Child process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("Child process %ld terminated by signal: %d\n", (long)wpid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        printf("Child process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));
    else if (WIFCONTINUED(status))
        printf("Child process %d continued\n", wpid);
    else
        printf("Unknown status for child process %d\n", wpid);
}
int main(int argc, char *argv[]){
    if (argc != 4){
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);}
    pid_t pid = fork();
    if (pid < 0){
        perror("Fork failed");
        exit(EXIT_FAILURE);}
    else if (pid == 0){ // Child process
        // Create argument list for execv
        char *args[] = {"a1.1-C", argv[1], argv[2], argv[3], NULL};
        // Execute char count child program
        execv("a1.1-C1", args);
        // execv returns only if an error occurs
        perror("Execv failed");
        exit(EXIT_FAILURE);
    }
    else{ // Parent process
        // Wait for the child process to terminate
        int status;
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);
        printf("Parent process: Child process has terminated.\n");}
    return 0;}
```

3. Διαδιεργασιακή Επικοινωνία

Ξεκινάμε την επέκταση του κώδικα του Ερωτήματος 2 ορίζοντας των αριθμό των διεργασιών παιδιών που θα δημιουργήσουμε για την παράλληλη αναζήτηση του επιθυμητού χαρακτήρα.

```c
#define P 10 // Number of child processes
```

Signal Handler

Στη συνέχεια εγκαθιστούμε έναν διαχειριστή σήματος μέσω της χρήσης της *signal* της βιβλιοθήκης <***signal.h***>, ώστε να διαχειριστούμε όπως εμείς επιθυμούμε όταν το πρόγραμμά μας λάβει ένα SIGINT σήμα, δηλαδή όταν ο χρήστης πατήσει Ctrl+C.

```c
signal(SIGINT, sigint_handler); // Install signal handler for SIGINT
```

Η συνάρτηση *signal* δέχεται δύο ορίσματα. Το πρώτο όρισμα είναι το σήμα που θα διαχειριστεί και το δεύτερο είναι ο δείκτης προς τη συνάρτηση που θα κληθεί όταν ληφθεί το σήμα. Η

συνάρτηση που θα κληθεί ονομάζεται sigint_handler και εκτυπώνει το συνολικό αριθμό διεργασιών που αναζητούν το αρχείο.

```
void sigint_handler(int signum){
    printf("\nNumber of child processes searching and counting: %d\n", P);
}
```

Επικοινωνία παιδιών-γονέα

Για να επιτευχθεί η σωστή επικοινωνία μεταξύ παιδιών και γονέα και για να γίνεται επιστροφή του σωστού αριθμού εμφανίσεων πρέπει να δημιουργηθεί ένα σύστημα από pipes που θα επιτρέπει στο κάθε παιδί να γράψει το αποτέλεσμα της αναζήτησης του και στον γονέα να διαβάσει και να αθροίσει τις εμφανίσεις. Αυτό γίνεται ορίζοντας ένα δισδιάστατο πίνακα pipefd[P][2]. Η δημιουργία των pipes γίνεται με *for loop* όπου ως όρισμα στην συνάρτηση *pipe* μπαίνει δείκτης στην κάθε γραμμή του πίνακα pipefd έτσι δημιουργούνται P pipes με read pipefd[i][0] και write pipefd[i][1]. Προφανώς ελέγχουμε για πιθανά λάθη και τερματίζουμε με αποτυχία αν εντοπιστεί κάποιο.

```
int pipefd[P][2];
// Create P pipes
for (int i = 0; i < P; i++)
        if (pipe(pipefd[i]) == -1)
        {
                perror("Pipe creation failed");
                exit(EXIT_FAILURE);
        }
```

Διαχωρισμός Αρχείου

Για να αντλήσουμε πληροφορίες για το αρχείο θα χρησιμοποιήσουμε το system call stat της βιβλιοθήκης <sys/stat.h>. Ορίζουμε επομένως ένα struct τύπου stat με όνομα st όπου και θα αποθηκευτούν οι πληροφορίες που θα αντλήσουμε. Χρησιμοποιούμε το system call (ελέγχοντας για ύπαρξη σφαλμάτων) και παίρνουμε την integer type off_t μεταβλητή file_size που περιέχει το μέγεθος του φακέλου σε bytes. Ύστερα υπολογίζουμε το μέγεθος του κάθε τμήματος που θα χειριστεί το παιδί και κάνουμε initialize τις μεταβλητές start και end που θα έχουν διαφορετική τιμή για κάθε παιδί. Κάθε παιδί θα διαχειρίζεται $\frac{file\_size}{P}$ bytes του φακέλου, εκτός από το τελευταίο παιδί που θα διαχειρίζεται και το υπόλοιπο της διαίρεσης $\left(\frac{filesize}{P} + filesize\%P\right)$ bytes. Ένα πρόβλημα που παρατηρείται με αυτή την υλοποίηση είναι όταν έχουμε παραπάνω διεργασίες παιδιά από ότι χαρακτήρες καθώς κάθε παιδί διαχειρίζεται 0 bytes του αρχείου (ακέραια διαίρεση) εκτός από το τελευταίο παιδί που διαχειρίζεται όλα τα bytes του αρχείου. Για να το διορθώσουμε αυτό μπορούμε να ορίσουμε πως όταν $P > filesize$ τότε $P = filesize$. Ωστόσο αυτό δεν ζητείται στην άσκηση και όταν ρωτήσαμε βοηθό εάν χρειάζεται η διαχείριση αυτής της περίπτωσης, αναφέρθηκε πως επαρκεί απλά να γραφτεί στην αναφορά (η τακτική αυτή χρησιμοποιείται στην 4η άσκηση).

Δημιουργία διεργασιών παιδιά και χρήση pipe

Εκτελούμε ένα for loop και ξεκινάμε τη δημιουργία παιδιών με κλήση της *fork()*. Η κάθε διεργασία παιδί κλείνει το read του pipe που της αντιστοιχεί καθώς δεν θα το χρησιμοποιήσει.

```
// Close read end of the pipe
if (close(pipefd[i][0]) == -1){
        perror("Close read end of pipe failed");
        exit(EXIT_FAILURE);
}
```

Ύστερα, υπολογίζει τις μεταβλητές start και end όπου η end εάν η συνθήκη $(i == P - 1)$ είναι true και βρισκόμαστε στο τελευταίο παιδί θα παίρνει την τιμή file_size-1 ενώ αν είναι false θα υπολογίζεται από το άθροισμα του start και του portion_size. Για την καταμέτρηση γίνεται κλήση της char_count που δημιουργήσαμε με τα κατάλληλα ορίσματα (αρχείο που θα διαβαστεί, χαρακτήρας, start, end) και για την εγγραφή στο pipe χρησιμοποιείται η write που θα γράφει στο write end του pipe που αντιστοιχεί στο παιδί την τιμή που είναι αποθηκευμένη στον char_count. Τέλος κλείνει το write end του pipe και τερματίζει επιτυχώς.
Με τη συνάρτηση συνάρτηση count_char η οποία ανοίγει τον φάκελο που περιέχει το κείμενο και μετράει τις εμφανίσεις του χαρακτήρα σε ένα διάστημα του φακέλου που καθορίζεται από τα ορίσματά της. Έτσι, δίνεται η δυνατότητα σε κάθε παιδί να αναζητήσει τον χαρακτήρα στο 1/P του φακέλου. Η υλοποίηση της συνάρτησης γίνεται με την lseek. Αρχικά χρησιμοποιώντας την μετακινούμε το file offset στην αρχική μας θέση και ξεκινάει η ανάγνωση. Με χρήση while loop όσο το offset βρίσκεται εντός του διαστήματος start-end και δεν έχουμε φτάσει στο τέλος του φακέλου κάνουμε καταμέτρηση των εμφανίσεων του target_char.

Parent process

Η διεργασία γονέας περιμένει τον τερματισμό των παιδιών της και κλείνει το write end του πρώτου pipe.

```
// Close write end of the first pipe
if (close(pipefd[0][1]) == -1)
{
      perror("Close write end of pipe failed");
      exit(EXIT_FAILURE);
}
```

Ύστερα διαβάζει τα αποτελέσματα από όλα τα pipes και τα αθροίζει. Κατά την διάρκεια της ανάγνωσης εκτυπώνει το αποτέλεσμα κάθε pipe και κλείνει το read end του. Τέλος ανοίγει τον φάκελο στον οποίο θέλουμε να αποθηκεύσουμε το αποτέλεσμα , τον γράφει , τον κλείνει και το πρόγραμμα τερματίζει επιτυχώς.

```c
---------------------------------------------------------------------------------------------------
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>
#define P 10 // Number of child processes
void explain_wait_status(pid_t wpid, int status){
    if (WIFEXITED(status))
        printf("Child process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("Child process %ld terminated by signal: %d\n", (long)wpid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        printf("Child process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));
    else if (WIFCONTINUED(status))
        printf("Child process %d continued\n", wpid);
    else
        printf("Unknown status for child process %d\n", wpid);
}

void sigint_handler(int signum){
    printf("\nNumber of child processes searching and counting: %d\n", P);}

int count_char(char *filename, char target_char, off_t start, off_t end, int child){
    int fpr, count = 0;
    char cc;
    //  open file for reading only
    if ((fpr = open(filename, O_RDONLY)) == -1)    {
        printf("Problem opening file to read\n");
        exit(EXIT_FAILURE);
    }
    // Move to the starting position
    lseek(fpr, start, SEEK_SET);
    // count the occurrences of the given character in the assigned portion of the file
    while (read(fpr, &cc, 1) > 0 && lseek(fpr, 0, SEEK_CUR) <= end)
        if (cc == target_char)
            count++;
    // close the file for reading
    if (close(fpr) == -1){
        perror("Problem closing file");
        exit(EXIT_FAILURE);
    }
    return count;
}
int main(int argc, char *argv[]){
    if (argc != 4){
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // Install signal handler for SIGINT
    signal(SIGINT, sigint_handler);
    int pipefd[P][2];
    pid_t pid;
    // Create P pipes
    for (int i = 0; i < P; i++)
        if (pipe(pipefd[i]) == -1){
            perror("Pipe creation failed");
            exit(EXIT_FAILURE);
        }
    // Get the size of the file
    struct stat st;
    if (stat(argv[1], &st) == -1){
        perror("Unable to get file size");
        exit(EXIT_FAILURE);
    }
    off_t file_size = st.st_size;
    // Calculate the portion of the file that each child process will handle
    off_t portion_size = file_size / P;
```

```
    off_t start, end;
    //  Create P child processes
    for (int i = 0; i < P; i++){
        pid = fork();
        if (pid == -1){
            perror("Fork failed");
            exit(EXIT_FAILURE);
        }
        else if (pid == 0){
            // Close read end of the pipe
            if (close(pipefd[i][0]) == -1){
                perror("Close read end of pipe failed");
                exit(EXIT_FAILURE);
            }
            // Calculate the portion of the file for this child process
            start = i * portion_size;
            end = (i == P - 1) ? file_size - 1 : start + portion_size;
            int char_count = count_char(argv[1], argv[3][0], start, end, i + 1);
            // Write char_count to the corresponding pipe
            if (write(pipefd[i][1], &char_count, sizeof(char_count)) == -1){
                perror("Write to pipe failed");
                exit(EXIT_FAILURE);
            }
            // Close write end of the pipe
            if (close(pipefd[i][1]) == -1){
                perror("Close write end of pipe failed");
                exit(EXIT_FAILURE);
            }
            exit(EXIT_SUCCESS);
        }
    }
    // Parent process
    for (int i = 0; i < P; i++){
        int status; // Wait for all the workers to exit
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);
    }
    // Close write end of the first pipe
    if (close(pipefd[0][1]) == -1){
        perror("Close write end of pipe failed");
        exit(EXIT_FAILURE);
    }
    // Read char_count from each pipe
    int total_count = 0;
    for (int i = 0; i < P; i++){
        int count;
        if (read(pipefd[i][0], &count, sizeof(count)) == -1){
            perror("Read from pipe failed");
            exit(EXIT_FAILURE);
        }
        total_count += count;
        printf("Parent Process: Counted characters from pipe %d is %d\n", i, count);
        close(pipefd[i][0]); // Close read end of the pipe
    }
    // Write total_count to output file
    int fpw;
    if ((fpw = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)) == -1){
        printf("Problem opening file to write\n");
        exit(EXIT_FAILURE);
    }
    // write the result in the output file
    char result[150];
    snprintf(result, sizeof(result), "The character '%c' appears %d times in file %s.\n", argv[3][0],
total_count, argv[1]);
    if (write(fpw, result, strlen(result)) != strlen(result)){
        perror("Problem writing to file");
        exit(EXIT_FAILURE);
    }
    // close the output file
    if (close(fpw) == -1){
        perror("Problem closing file");
        exit(EXIT_FAILURE);}
    return 0;}
```

---

4. Εφαρμογή Παράλληλης Καταμέτρησης Χαρακτήρων

Χρησιμοποιώντας τους κώδικες των προηγούμενων ασκήσεων ως εφαλτήριο θα υλοποιήσουμε τα τρία προγράμματα που ζητούνται ***(frontend, dispatcher, worker).***

Front-End

- *Δήλωση εργατών από τον χρήστη*

Ξεκινάμε με το να ζητάμε από τον χρήστη να μας δώσει τον επιθυμητό αριθμό εργατών, που πρέπει να παράγουμε δυναμικά για να πραγματοποιήσουν την παράλληλη καταμέτρηση χαρακτήρων.

```
// Declare the starting number of workers
int num_workers;
printf("> Enter number of Workers: ");
scanf("%d", &num_workers);
fflush(stdin);
```

- *Signal Handlers*

Στη συνέχεια, εγκαθιστούμε δυο διαχειριστές σημάτων για το SIGINT (ή Ctrl+C) και SIGKILL σήμα, όπου αντίστοιχα θα διαχειριστούμε με τις συναρτήσεις `handler_SIGINT` και `handler_SIGKILL`.

```
// Install signal handler for SIGINT and SIGKILL
signal(SIGINT, handler_SIGINT);
signal(SIGKILL, handler_SIGKILL);
```

- *Γεφύρωση επικοινωνίας μεταξύ Front-End και Dispatcher*

Δημιουργούμαι μία σωλήνωση για την επικοινωνία μεταξύ του Front-End και Dispatcher.

```
// Create pipe for communication between frontend and dispatcher
if (pipe(fd) == -1){
        perror("Error creating pipe");
        exit(EXIT_FAILURE);
}
```

Έπειτα, δημιουργούμαι μια εργασία παιδί με την συνάρτηση *fork(),* η οποία θα εκτελέσει τον dispatcher. Προτού τρέξουμε το εκτελέσιμο χρησιμοποιούμε την συνάρτηση *dup2* για να αντικαταστήσουμε το standard output του dispatcher με το write end του pipe αφενός για πιο ομαλή επικοινωνία και αφετέρου διότι μόνο το Front-End πρόγραμμα μπορεί να έχει άμεση επικοινωνία με τον χρήστη. Η συνάρτηση αυτή δέχεται δύο παραμέτρους. Η πρώτη είναι το αναγνωριστικό που θέλουμε να αντιγράψουμε, στη δικιά μας περίπτωση το write end of pipe (fd[1]) και η δεύτερη παράμετρος είναι το αναγνωριστικό αρχείου που θέλουμε να αντικαταστήσουμε, το standard output στη συγκεκριμένη περίπτωση.

```
// Replace standard output with write end of pipe
dup2(fd[1], 1); // File descriptor 1 refers to the stdout
```

Ύστερα, ορίζουμε τον πίνακα συμβολοσειρών με τα ορίσματά μας, όπου προσθέτουμε ως επιπλέον όρισμα τον αριθμό εργατών που πρέπει δυναμικά να δημιουργήσουμε. Ωστόσο, ο αριθμός εργατών πρέπει να μετατραπεί σε συμβολοσειρά για να δηλωθεί ως όρισμα. Για αυτό το σκοπό αξιοποιούμε την sprintf.

```
sprintf(workers, "%d", num_workers);
```

Η προσθήκη αυτή μας επιτρέπει να δηλώσουμε τον αριθμό των εργατών στον dispatcher, χωρίς να χρειαστεί να χρησιμοποιήσουμε την σωλήνωση χωρίς λόγο. Τέλος, τρέχουμε το πρόγραμμα dispatcher με την βοήθεια της συνάρτησης *execv*.

```
// Execute dispatcher
char *args[] = {"a1.4-dispatcher", argv[1], argv[2], argv[3], workers, NULL};
execv("a1.4-dispatcher", args);
```

- *Ανάγνωση δεδομένων από τον Dispatcher*

Αφού ο dispatcher έχει δημιουργήσει όλους τους εργάτες που ζητήθηκαν για την καταμέτρηση των χαρακτήρων και απολάβει το τελικό αποτέλεσμα, το στέλνει στο Front-End (δείτε Dispatcher). Ο Front-End διαβάζει το αποτέλεσμα μέσω της *read* και αποθηκεύει το αποτέλεσμα στη μεταβλητή counter.

```
int counter;
read(fd[0], &counter, sizeof(int));
```

Τέλος, εκτυπώνουμε το αποτέλεσμα στον χρήστη και περιμένουμε να τερματιστεί ο dispatcher.

```
// Print the number of times the character appears in the file
printf("Front End: The character '%c' appears %d times in file %s.\n", argv[3][0], counter, argv[1]);
// Wait for the dispatcher process to terminate
int status;
pid_t wpid = waitpid(pid, &status, 0);
printf("Front-End is exiting normal\n");
return 0;
```

Σημείωση: *Στο τέλος του αρχείου υπάρχει ένα σχόλιο για ένα «σχέδιο» ενός ολοκληρωμένου Front-End, που θα διαχειριζόταν επιπλέον λειτουργίες.*

Ολοκληρωμένος κώδικας:

```
-----------------------------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include "config.h"
#define BUFFER_SIZE 1024

void explain_wait_status(pid_t wpid, int status)
{
    if (WIFEXITED(status))
        printf("Dispatcher process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("Dispatcher process %ld terminated by signal: %d\n", (long)wpid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        printf("Dispatcher process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));
    else if (WIFCONTINUED(status))
        printf("Dispatcher process %d continued\n", wpid);
    else
        printf("Unknown status for Dispatcher process %d\n", wpid);}
```

```c
void handler_SIGINT(int signal)
{    // Code to handle SIGINT signal
}

void handler_SIGKILL(int signal)
{    // Code to handle SIGTSTP signal
}

int main(int argc, char *argv[])
{
    if (argc != 4)
    {
        fprintf(stderr, "Incorrect Syntax\n");
        fprintf(stderr, "Usage: %s <input_file> <output_file> <character_to_search>\n", argv[0]);
        fprintf(stderr, "Example: %s input.txt output.txt a\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Declare the starting number of workers
    int num_workers;
    printf("> Enter number of Workers: ");
    scanf("%d", &num_workers);
    fflush(stdin);

    int fd[2];
    char buffer[BUFFER_SIZE];

    // Install signal handler for SIGINT and SIGKILL
    signal(SIGINT, handler_SIGINT);
    signal(SIGKILL, handler_SIGKILL);

    // Create pipe for communication between frontend and dispatcher
    if (pipe(fd) == -1)
    {
        perror("Error creating pipe");
        exit(EXIT_FAILURE);
    }

    pid_t pid = fork();

    if (pid < 0)
    {
        perror("Error forking process");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {   // Replace standard output with write end of pipe
        dup2(fd[1], 1);

        // Send the number of workers to the dispatcher through an argument
        char workers[5];
        sprintf(workers, "%d", num_workers);

        // Execute dispatcher
        char *args[] = {"a1.4-dispatcher", argv[1], argv[2], argv[3], workers, NULL};
        execv("a1.4-dispatcher", args);

        // If execv fails, print error message
        perror("Error executing dispatcher");
        exit(EXIT_FAILURE);
    }
    int counter;
    read(fd[0], &counter, sizeof(int));
    // Print the number of times the character appears in the file
    printf("Front End: The character '%c' appears %d times in file %s.\n", argv[3][0], counter,
argv[1]);

    // Wait for the dispatcher process to terminate
    int status;
    pid_t wpid = waitpid(pid, &status, 0);
    printf("Front-End is exiting normal\n");
    return 0;
}
```

Dispatcher

- *Δημιουργία εργατών δυναμικά*

Αποθηκεύουμε τον αριθμό εργατών από την επιπλέον παράμετρο που προσθέσαμε ως ακέραιο με την βοήθεια της συνάρτησης *atoi (ASCII to Integer)*.

```
int num_workers = atoi(argv[4]);
```

Στη συνέχεια, δημιουργούμε έναν πίνακα δυναμικά, με την συνάρτηση *malloc*, όπου θα αποθηκεύσουμε τα Process ID των εργατών.

```
int *worker_pids = (int *)malloc(num_workers * sizeof(int));
```

Προφανώς, πρέπει να υπολογίσουμε το μέγεθος του αρχείου ανάγνωσης για να γνωρίζουμε το ποσοστό που θα διαβάσει κάθε εργάτης. Αυτό το πετυχαίνουμε ακριβώς όπως και στην 3η Άσκηση με την προσθήκη πως επιτρέπονται μέχρι 80 εργάτες και πως εάν οι εργάτες είναι περισσότεροι από τον αριθμό των χαρακτήρων που υπάρχουν στο αρχείο (δηλαδή το μέγεθος του αρχείου είναι μικρότερο από τον αριθμό των εργατών) τότε ορίζουμε των αριθμό των εργατών ίσο με το μέγεθος του αρχείου. Έτσι κάθε εργάτης θα διαβάσει έναν χαρακτήρα και θα επιστρέψει την καταμέτρηση του στον Dispatcher.

Παράλληλα, δημιουργούμε σωληνώσεις για να επικοινωνούμε από τον dispatcher με κάθε worker που έχουμε. Αυτό το πετυχαίνουμε ορίζοντας έναν δισδιάστατο πίνακα, με δυναμικό τρόπο, που θα αποθηκεύει τα file descriptor της κάθε σωλήνωσης.

```
// Create array for the pipes connecting to the workers
int **fd_array = (int **)malloc(num_workers * sizeof(int *));
for (int i = 0; i < num_workers; i++)
      fd_array[i] = (int *)malloc(2 * sizeof(int));
```

Με την χρήση ενός for loop θα δημιουργήσουμε όσα παιδιά θέλουμε να γίνουνε εργάτες. Αρχικά, εγκαθιδρύουμε επικοινωνία μεταξύ του dispatcher και του κάθε worker δημιουργώντας τις αντίστοιχες σωληνώσεις από τον πίνακα που δημιουργήσαμε παραπάνω. Στη συνέχεια, με την συνάρτηση fork δημιουργούμε τα παιδιά που θα γίνουν workers και αποθηκεύουμε τα αντίστοιχα PID's στον πίνακα `worker_pids`. Προτού κάθε παιδί εκτελέσει το "a1.4-worker" και γίνει worker, θα περάσουμε σε κάθε παιδί το διάστημα από ΄που πρέπει να κάνει καταμέτρηση χαρακτήρων ως ορίσματα (Start_Byte, End_Byte). Ο υπολογισμός για το start και end byte γίνεται στο τέλος του for loop με τον ίδιο τρόπου όπως και στην 3η Άσκηση. Τώρα, όπως και στον Front-End, έτσι κι εδώ χρησιμοποιούμε την *dup2* για τους ίδιους σκοπούς και λόγους. Τέλος, κάθε παιδί εκτελεί το `"./a1.4-worker"` και γίνεται worker.

Σημείωση: το worker_id[5] έχει προστεθεί ως παράμετρος για debugging σκοπούς

- *Ανάγνωση αποτελεσμάτων κάθε εργάτη*

Αρχικά, περιμένουμε από κάθε worker να τελειώσει την καταμέτρηση και να επιστρέψει το αποτέλεσμά του στον dispatcher.

```
for (int i = 0; i < num_workers; i++){// Wait for all the workers to finish
        int status;
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);
}
```

Έπειτα, μέσω των αντίστοιχων σωληνώσεων διαβάζουμε τον αριθμό χαρακτήρων που επέστρεψε κάθε worker και τους προσθέτουμε για να πάρουμε την συνολική καταμέτρηση.

```
    int total = 0;
    for (int i = 0; i < num_workers; i++)
    {
        int local;
        read(fd_array[i][0], &local, sizeof(int));
        total += local;
    }
```

Τέλος, στέλνουμε το τελικό αποτέλεσμα στον Front-End μέσω του standard output και αποδεσμεύουμε την μνήμη των πινάκων που δημιουργήθηκα δυναμικά.

```
write(1, &total, sizeof(int));

for (int i = 0; i < num_workers; i++)
        free(fd_array[i]);
free(fd_array);
free(worker_pids);
```

Ολοκληρωμένος κώδικας:

```
-----------------------------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <signal.h>
#include "config.h"

#define BUFFER_SIZE 1024
#define MAX_WORKERS 100

void explain_wait_status(pid_t wpid, int status){
    if (WIFEXITED(status))
        fprintf(stderr, "Worker process %ld terminated normally with exit status: %d\n", (long)wpid,
WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        fprintf(stderr, "Worker process %ld terminated by signal: %d\n", (long)wpid,
WTERMSIG(status));
    else if (WIFSTOPPED(status))
        fprintf(stderr, "Worker process %ld stopped by signal: %d\n", (long)wpid, WSTOPSIG(status));
    else if (WIFCONTINUED(status))
        fprintf(stderr, "Worker process %d continued\n", wpid);
    else
        fprintf(stderr, "Unknown status for Worker process %d\n", wpid);
}

int main(int argc, char *argv[]){
    char buffer[BUFFER_SIZE];
    int num_workers = atoi(argv[4]);

    if (num_workers > MAX_WORKERS)
        num_workers = MAX_WORKERS;
```

```c
    int *worker_pids = (int *)malloc(num_workers * sizeof(int));
    // Get the size of the file
    struct stat st;
    if (stat(argv[1], &st) == -1){
        perror("Unable to get file size");
        exit(EXIT_FAILURE);
    }
    off_t file_size = st.st_size;
    if (num_workers > file_size)
        num_workers = file_size;
    fprintf(stderr, "Dispatcher: The filesize is %d\n", file_size);
    // Calculate the portion of the file that each child process will handle
    off_t portion_size = file_size / num_workers;
    off_t start = 0, end = portion_size;
    // Create array for the pipes connecting to the workers
    int **fd_array = (int **)malloc(num_workers * sizeof(int *));
    for (int i = 0; i < num_workers; i++)
        fd_array[i] = (int *)malloc(2 * sizeof(int));

    // Fork worker processes
    for (int i = 0; i < num_workers; i++){
        // Create the pipe connection
        if (pipe(fd_array[i]) == -1){
            perror("Error creating pipe");
            exit(EXIT_FAILURE);
        }
        // Create the child worker
        worker_pids[i] = fork();

        if (worker_pids[i] < 0){
            perror("Error forking worker process");
            exit(EXIT_FAILURE);
        }
        else if (worker_pids[i] == 0){
            // Execute worker
            // fprintf(stderr, "Dispatcher: Child %d with start-> %d and end-> %d\n", i, start, end);
            char start_byte[10], end_byte[10], worker_id[5];
            sprintf(start_byte, "%d", start);
            sprintf(end_byte, "%d", end);
            sprintf(worker_id, "%d", i);
            char *args[] = {"./a1.4-worker", argv[1], argv[2], argv[3], start_byte, end_byte,
worker_id, NULL};
            dup2(fd_array[i][1], 1);
            execvp("./a1.4-worker", args);
            perror("Error executing Worker");
            exit(EXIT_FAILURE);
        }
        close(fd_array[i][1]); // Close write for dispatcher
        // Calculate the portion of the file for the next worker
        start = end;
        end = (i == num_workers - 2) ? file_size - 1 : start + portion_size;
    }
    pid_t pid;
    // Wait for all the workers to finish
    for (int i = 0; i < num_workers; i++){
        int status;
        pid_t wpid = waitpid(pid, &status, 0);
        explain_wait_status(wpid, status);
    }
    int total = 0;
    for (int i = 0; i < num_workers; i++){
        int local;
        read(fd_array[i][0], &local, sizeof(int));
        total += local;
    }
    fprintf(stderr, "Dispatcher: Parent total: %d\n", total);
    write(1, &total, sizeof(int));
    for (int i = 0; i < num_workers; i++)
        free(fd_array[i]);
    free(fd_array);
    free(worker_pids);
}
```

Worker

Ο κάθε worker έχει σχεδιαστεί με ακριβώς τον ίδιο τρόπο όπως στην 3η Άσκηση.

Αρχικά, αποθηκεύουμε τα απαραίτητα ορίσματα που χρειαζόμαστε σε μεταβλητές και μετατρέπουμε συμβολοσειρές σε ακέραιους αριθμούς (όπως και στον Dispatcher). Έπειτα, ανοίγουμε το αρχείο ανάγνωσης και καλούμε την συνάρτηση `count_char` που καταμετράει τους επιθυμητούς χαρακτήρες σε ένα συγκεκριμένο διάστημα (αναλύσαμε τη συνάρτηση αυτή στη 3η Άσκηση).

Τέλος, στέλνουμε το αποτέλεσμα στον Dispatcher μέσω του standard output και κλείνουμε το αρχείο ανάγνωσης που είχαμε ανοίξει.

Ολοκληρωμένος κώδικας:

```
-----------------------------------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <signal.h>
#include "config.h"
int count_char(char *filename, char target_char, off_t start, off_t end, int worker_id)
{
    int fpr, count = 0;
    char cc;
    //  open file for reading only
    if ((fpr = open(filename, O_RDONLY)) == -1){
        fprintf(stderr, "Problem opening file to read\n");
        exit(EXIT_FAILURE);
    }
    // Move to the starting position
    lseek(fpr, start, SEEK_SET);
    // count the occurrences of the given character in the assigned portion of the file
    while (read(fpr, &cc, 1) > 0 && lseek(fpr, 0, SEEK_CUR) <= end){
        if (cc == target_char)
            count++;
    }
    close(fpr);
    return count;
}
int main(int argc, char *argv[])
{
    char *filename = argv[1];
    char target_char = argv[3][0];
    off_t start = atoi(argv[4]);
    off_t end = atoi(argv[5]);
    int worker_id = atoi(argv[6]);
    int fpr;
    char cc;
    //  open file for reading only
    if ((fpr = open(filename, O_RDONLY)) == -1){
        fprintf(stderr, "Problem opening file to read\n");
        exit(EXIT_FAILURE);
    }
    //  Count the occurrences of the given character in the assigned portion of the file
    int count = count_char(filename, target_char, start, end, worker_id);
    write(1, &count, sizeof(int));
    // close the file for reading
    close(fpr);
    return 0;
}
-----------------------------------------------------------------------------------------------------
```

Τελικές Σημειώσεις

Η τελική άσκηση προφανώς δεν είναι ολοκληρωμένη λόγω έλλειψης χρόνου, ωστόσο θα αναφέρουμε μερικές ιδέες που θα μπορούσαμε να υλοποιήσουμε εάν προλαβαίναμε.

- *Χρήση των config.h και util.c*

Θα μπορούσαμε να δημιουργήσουμε το αρχείο config.h για να ορίσουμε σταθερές (π.χ. MAX_WORKERS, BUFFER_SIZE, κλπ.) και να δηλώσουμε κοινόχρηστες συναρτήσεις από τον dispatcher και front-end (π.χ. explain_wait_status). Ενώ στο util.c αρχείο μπορούμε να υλοποιήσουμε τις συναρτήσεις που δηλώσαμε στη config.h.

- *Διαχείριση Σημάτων*

Όταν τερματίζουμε έναν worker μέσω ενός σήματος, ας πούμε SIGKILL, (από ένα άλλο τερματικό) τότε πρέπει να επιστρέφουμε μέχρι πιο byte έχει διαβάσει και να αναθέσουμε σε έναν άλλο worker να ολοκληρώσει την καταμέτρηση. Γι αυτό τον σκοπό θα χρειαζόμασταν να ορίσουμε έναν πίνακα με τα start και end bytes του κάθε worker, έναν διαχειριστή σήματος, όπου θα ανέθετε σε κάποιον άλλον worker να καταμετρήσει τα υπόλοιπα bytes του worker που «σκοτώθηκε» και σωστή επικοινωνία μεταξύ του dispatcher και των workers μέσω των σωληνώσεων.