Λειτουργικά Συστήματα Υπολογιστών
1η Εργαστηριακή Άσκηση
Κλήσεις συστήματος, διεργασίες και διεργασιακή επικοινωνία

Συντάκτες: Γεώργιος Πνευματικός 03121058
Νικόλαος Λάππας 03121098

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

Μεταφέρουμε από το /home/oslab/code/char-count τον κώδικα σε C και το Makefile στον προσωπικό μας κατάλογο. Μετατρέπουμε τον κώδικα αυτό σε εκτελέσιμο με την εντολή make a1.1-C και περνάμε τα ζητούμενα ορίσματα που απαιτούνται για την εκτέλεση του κώδικα. Αν εκτελέσουμε λοιπόν $./a1.1 - C\ my\_name.txt\ output.txt\ m$ τότε θα αναζητηθεί πόσες φορές εμφανίζεται ο χαρακτήρας m στο αρχείο my_name.txt και θα εμφανίζουμε τον αριθμό αυτόν στο αρχείο output.txt.

Καλούμαστε να υλοποιήσουμε πρόγραμμα που θα παίρνει τις ίδιες παραμέτρους με το πρόγραμμα που μόλις εξετάσαμε και θα εκτελεί την ίδια λειτουργία. Ωστόσο, πρέπει να αποφύγουμε τα system calls.

Παραθέτουμε τον κώδικα που ζητείται $a1.1 - system\_calls.c$

GNU nano 5.4 a1.1-system_calls.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char **argv){
        int fdr, fdw;//file descriptors
        char c2c = 'a', cc;
        int count = 0;
        ssize_t rcnt, wcnt;//For read and write

        fdr = open(argv[1], O_RDONLY);//We open the file and assign value fdr
                if (fdr == -1){
                perror("open");//If the file doesn't open we receivw error
                exit(1);//we exit with an error
        }

        fdw = open(argv[2], O_WRONLY | O_CREAT, 0644);//Same process as before
                if (fdw == -1){
                perror("open");
                exit(1);
        }

        c2c = argv[3][0];//We assign the search character to c2c

        while ((rcnt = read(fdr, &cc, 1)) > 0)//While read check
                 if (cc == c2c) count++;

                if (rcnt == -1){//If rcnt == -1 we encountered a fatal error
                perror("read");
                exit(1);}

        char output[50];
        int length = snprintf(output, 50, "The character '%c' appears %d times\n", c2c, count);
//We store the above on array output as well as the number of characters which is returned from snprintf to length

        wcnt = write(fdw, output, length);//We write the ouptut array to the output file
                if (wcnt == -1) {
                perror("write");
                exit(1);
        }
        close(fdr);//closing files
        close(fdw);
        return 0;
}
```

Το πρόγραμμα αυτό καλείται ακριβώς με τα ίδια ορίσματα όπως και πριν.

Ωστόσο, αυτήν την φορά ανοίγουμε τα αρχεία με τις εντολές open και σώζουμε τα αρχεία αυτά στους file descriptors (fdr, fdw). Σε περίπτωση όπου κάποιο από τα αρχεία δεν κατάφερε να ανοίξει τότε απαιτούμε τον βίαιο τερματισμό του προγράμματος μέσω της exit(1).

Η εντολή $while\ \big((rcnt\ =\ read(fdr, \&cc, 1)) > 0\big)\ if\ (cc\ ==\ c2c)\ count++;$

Αυτό που κάνει είναι να διαβάζει από το αρχείο (argv[1]) και όσο διαβάζει να εξετάζει την συνθήκη χαρακτήρα. Τονίζουμε πώς η read επιστρέφει τιμή τύπου ssize_t στην μεταβλητή rcnt. Έτσι η rcnt φανερώνει τον αριθμό bytes που διαβάζονται. Όσο λαμβάνει θετική τιμή διαβάζει κανονικά, με την μηδενική τιμή φανερώνεται το τέλος του αρχείου και με την αρνητική τιμή λαμβάνουμε σφάλμα και εκτελούμε βίαιο τερματισμό.

Έπειτα μέσω της εντολής snprintf τοποθετούμε στον πίνακα output[50] το μήνυμα που θα εκτυπώνεται στο output αρχείο καθώς και τον αριθμό εμφάνισης του αναζητούμενου χαρακτήρα στο αρχείο. Η snprintf επιστρέφει τον αριθμό χαρακτήρων που τοποθετούνται στον πίνακα output ενημερώνοντας έτσι την μεταβλητή length.

Με το ίδιο ακριβώς σκεπτικό καλούμε στην συνέχεια

$wcnt\ =\ write(fdw, output, length);$

$if\ (wcnt\ ==\ -1)\ \{$

$perror(\text{"write"});$

$exit(1);$

$\}$

Γράφουμε στο αρχείο argv[2] τα περιεχόμενα του πίνακα output και ελέγχουμε πιθανό λάθος αν wcnt == -1.

Για να λειτουργήσει σωστά το πρόγραμμά μας, όπως κάνει το δοθέν πρόγραμμα $a1.1 - C.c$, έχουμε αντικαταστήσει το '$w+$' με την εντολή '$O\_WRONLY \mid O\_CREAT, 0644$' με την οποία πρακτικά ελέγχουμε αν το αρχείο προς εγγραφή είναι έγκυρο προς εγγραφή, αν ναι και υπάρχει, τότε σβήνουμε το περιεχόμενό του αντικαθιστώντας το με το καινούργιο μήνυμα. Αν δεν υπάρχει σαν αρχείο, τότε το δημιουργεί το πρόγραμμά μας και του εισάγει το ίδιο μήνυμα.

Τέλος κλείνουμε τα αρχεία και τερματίζει το πρόγραμμα αν όλα λειτούργησαν ομαλά.

Τονίζουμε ότι το πρόγραμμα αυτό τρέχει ακριβώς όπως και το προηγούμενο.

Προαιρετικό Ερώτημα: $a1\_checkcalls.c$

Θέλουμε να επεκτείνουμε το πρόγραμμα ώστε να ελέγχουμε αν ο χρήστης έδωσε τα σωστά ορίσματα και σε περίπτωση που δεν συνέβη αυτό να ενημερώνουμε για πιθανά λάθη.

Υλοποιήσαμε τις αλλαγές στην αρχή του σώματος της main συνάρτησης του νέου προγράμματος τις οποίες παραθέτουμε

```
int main (int argc, char **argv){

        if (argc != 4){//Checks if user gave exactly three arguments
        perror("The user gave the wrong amount of calls");
        exit(1);
        }

        if (strlen(argv[3]) != 1){//Checks if third argument is a character
        perror("not character");
        exit(1);
        }

        if (access(argv[1], R_OK) == -1) {//Checks if read is available from first file
        perror("access");
        exit(1);
        }

        if (access(argv[2], F_OK) != -1 && access(argv[2], W_OK) == -1) {
        perror("access");
        exit(1);// This part ends the program if the write file exists but is not readable
    }

        int fdr, fdw;//file descriptors
        char c2c = 'a', cc;
        int count = 0;
        ssize_t rcnt, wcnt;//For read and write

        fdr = open(argv[1], O_RDONLY);//We open the file and assign value fdr
                if (fdr == -1){
                perror("open");//If the file doesn't open we receivw error
                exit(1);//we exit with an error
        }

        fdw = open(argv[2], O_WRONLY | O_CREAT, 0644);
                if (fdw == -1){//With O_CREAT if ths WR file doesn't exist it is created
                perror("open");
                exit(1);
        }
```

- $if\ (argc\ !=\ 4)$

Συγκεκριμένα, η argc φανερώνει το πλήθος των ορισμάτων κατά την εκτέλεση του κώδικα. Έτσι αφού το πρώτο όρισμα argv[0] είναι το ίδιο το εκτελέσιμο πρόγραμμα αν ο χρήστης δεν δώσει τρία επιπρόσθετα ορίσματα κατά την εκτέλεση τότε θα έχει δώσει λανθασμένο αριθμό ορισμάτων με αποτέλεσμα να λαμβάνει το επιθυμητό μήνυμα σφάλματος και να τερματίζει το πρόγραμμα.

- $if\ (strlen(argv[3])\ !=\ 1)$

Αν το τρίτο όρισμα που δίνει ο χρήστης δεν είναι ένας χαρακτήρας, τότε το πρόγραμμα τερματίζει.

- $if\ (access(argv[1], R\_OK)\ ==\ -1)$

Αν το πρώτο αρχείο που δίνουμε ως όρισμα δεν είναι διαθέσιμο για διάβασμα τότε παίρνουμε βίαιο τερματισμό. Δηλαδή ο χρήστης δεν έχει δώσει σωστό πρώτο όρισμα για διάβασμα αρχείου.

- $if\ (access(argv[2], F\_OK)\ !=\ -1\ \&\&\ access(argv[2], W\_OK)\ ==\ -1)$

Αν το δεύτερο αρχείο που δίνεται ως όρισμα υπάρχει (αυτό εξετάζεται από το πρώτο σκέλος) και δεν είναι διαθέσιμο για γράψιμο (αυτό εξετάζεται από το δεύτερο) τότε δόθηκε λανθασμένο δεύτερο αρχείο. Τονίζουμε πως εδώ δεν εξετάζεται η περίπτωση όπου το αρχείο που δόθηκε ως δεύτερο όρισμα δεν υπάρχει. Στην περίπτωση αυτή επιθυμούμε να δημιουργείτε το αρχείο εγγραφής και να αποτελεί την έξοδο, όπως θα αποσαφηνιστεί στην επόμενη εντολή.

Ο τελευταίος έλεγχος, σχετίζεται άμεσα και με την τροποποίηση που κάναμε στην εντολή fdw = open(argv[2], O_WRONLY | O_CREAT, 0644). Πιο συγκεκριμένα μέσω της εντολής fdw = open(argv[2], O_WRONLY | O_CREAT, 0644); Επιχειρούμε να ανοίξουμε το 2^{ο} όρισμα (που αν δεν είναι εγγράψιμο αρχείο θα έχει ήδη τερματιστεί το πρόγραμμα μέσω της access()) το οποίο θα είναι αποκλειστικά αρχείο για γράψιμο. Με το flag O_CREAT καταφέρνουμε να μπορεί να δημιουργηθεί το αρχείο που περάσαμε ως όρισμα σε περίπτωση που το αρχείο δεν υπάρχει.
Το όρισμα 0644 καθορίζει τα δικαιώματα πρόσβασης που έχει ο χρήστης στο UNIX αναφορικά με το αρχείο (να μπορεί ο χρήστης να έχει πρόσβαση σε αυτό, να το διαβάσει και να γράψει).

- Πλέον όταν τρέχουμε τον κώδικα και το τρίτο όρισμα δεν είναι ένας απλός χαρακτήρας λαμβάνεται το μήνυμα: not character
- Όταν τρέχουμε το όρισμα και δίνουμε περισσότερες ή λιγότερες από τρείς παραμέτρους λαμβάνεται το μήνυμα: ‘The user gave the wrong amount of calls’
- Αν δώσουμε σαν πρώτο όρισμα κάποιο αρχείο που δεν υπάρχει λαμβάνουμε πρόβλημα στην access. Τονίζουμε πώς το ίδιο θα συνέβαινε απευθείας και στην open αλλά πλέον ελέγχουμε την προσβασιμότητα στην access πρώτα.
- Τέλος με τα flags αν δώσουμε ένα αρχείο ως δεύτερο όρισμα για εγγραφή που δεν υπάρχει, αυτό θα δημιουργηθεί αυτόματα μέσω της O_CREAT και θα αποτελέσει το αρχείο εξόδου.

Άσκηση 2: Δημιουργία διεργασιών

Παραθέτουμε τον κώδικα για τα ερωτήματα 1 και 2 $a1.2 - fork.c$

```
  GNU nano 5.4
#include <stdio.h>
#include <sys/types.h> //for pid_t()
#include <unistd.h> //for fork() and getpid()
#include <stdlib.h> //for exit()
#include <sys/wait.h> //for wait()

int main (){
        pid_t p, mypid;
        int x = 11;
        p = fork();
        int status; //argument for wait

        if (p < 0){//This part checks if the pid_t p of fork is negative error
                perror("fork");
                exit(1);
        }
        else if (p == 0){//This is the child section
                x = 10;
                mypid = getpid();
                pid_t father_pid = getppid(); //gives the pid of the child's father
                printf("Hello world! My pid is %d and my father's pid is %d\n", mypid, father_pid);
                printf("The value of x for me, the child, is: %d\n", x);
                exit(0);
        }
        else {//this is the father section
                wait(&status); //wait until child terminates
                x = 3;
                printf("My child's pid is %d\n", p);
                printf("The value of x for father is: %d\n", x);
                exit(0);
        }
}
```

1) Η fork επιστρέφει τύπο pid_t.

Καλούμε την fork μέσω της p = fork();

Δημιουργείται έτσι ένα αντίγραφο της διεργασίας γονέα, η διεργασία παιδί και οι δύο διεργασίες συνεχίζουν την εκτέλεσή τους, αναθέτοντας στην μεταβλητή p την τιμή που τους επιστρέφει η fork. Υπό ομαλή λειτουργία, στην διεργασία γονέα επιστρέφεται το αναγνωριστικό της διεργασίας παιδί κατά την fork ενώ στο παιδί επιστρέφεται η τιμή 0.

Παρακάτω γίνονται οι τρείς έλεγχοι

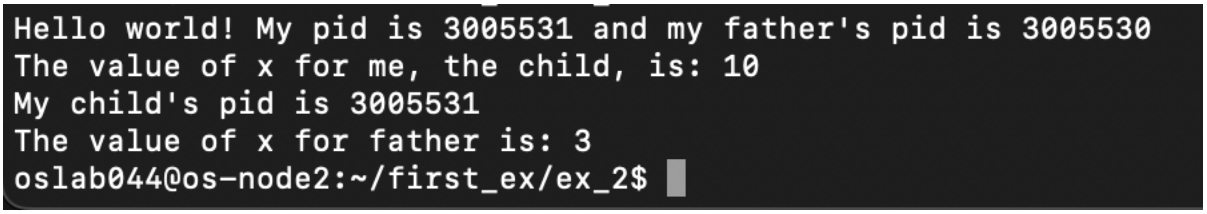
- If (p < 0) έχουμε σφάλμα
- If (p == 0) η διεργασία που δημιουργήθηκε εκτελεί τον κώδικα του παιδιού
- If (p > 0) η διεργασία είναι η γονεϊκή

Η διεργασία παιδί χαιρετάει τον κόσμο και τυπώνει το αναγνωριστικό της και το αναγνωριστικό του πατέρα της.

Η διεργασία γονέα περιμένει τον τερματισμό της διεργασίας παιδί μέσω της wait(&status) και τυπώνει το αναγνωριστικό της διεργασίας παιδί τυπώνοντας το p. Πιο συγκεκριμένα, η διεργασία γονέα (p>0) είναι σε αναμονή μέχρι τον τερματισμό μια διεργασίας παιδί (εδώ περιμένει την μοναδική διεργασία παιδί να τελειώσει) και ύστερα συνεχίζει εκείνος με το δικό του μπλοκ κώδικα. Αν δεν υπάρχει διεργασία παιδί, τότε η εντολή wait(&status) επιστρέφει αμέσως -1. Αν λειτουργήσει σωστά τότε επιστρέφει το pid της διεργασίας παιδιού που μόλις τελείωσε. Τέλος, το status πρόκειται το αναγνωριστικό του πως τερματίστηκε η διεργασία παιδί (π.χ. μέσω σήματος).

2) Ορίζουμε μια μεταβλητή x στον γονέα πριν εκτελεστεί η εντολή p = fork(), δίνοντας της την τιμή 11.
   Μεταβάλλουμε την τιμή της μεταβλητής x τόσο στην διεργασία παιδί (x = 11 μέσα στην συνθήκη if (p == 0)) όσο και στην γονεϊκή διεργασία (x = 3 μέσα στην συνθήκη else)

Παραθέτουμε τι εκτυπώνεται κατά την εκτέλεση του προγράμματος



Παρατηρούμε ότι η διεργασία παιδί τυπώνει x = 10 ενώ η διεργασία γονέα τυπώνει x = 3. Η κάθε διεργασία, λοιπόν, τυπώνει την τιμή που τις ανατέθηκε στο σώμα της. Αυτό συμβαίνει γιατί οποιαδήποτε αλλαγή στη μια διεργασία δεν επηρεάζει καθόλου την άλλη διεργασία. Η διεργασία παιδί είναι (σχεδόν) ένα αντίγραφο της διεργασίας γονέα. Κάθε διεργασία έχει τις δικές της μεταβλητές, οι οποίες έχουν βέβαια ίδια τιμή, διαφορετικό, όμως, program counter, ο οποίος δείχνει στο ίδιο σημείο. Οι μεταβλητές κάθε διεργασίας έχουν ξεχωριστή θέση στην μνήμη και άρα αλλαγές στην μεταβλητή x της διεργασίας γονέα δεν επηρεάζουν την μεταβλητή x στην διεργασία παιδί. Το ίδιο ισχύει και vice versa.

3) Υλοποιήσαμε τον κώδικα κατά τον οποίο η διεργασία παιδί αναζητά το πλήθος των εμφανίσεων του χαρακτήρα στο αρχείο ανάγνωσης.

Για την επίτευξη της αναζήτησης αυτής, στο σώμα της διεργασίας παιδί καλούμε την συνάρτηση counter και ψάχνουμε, έτσι, πόσες φορές εμφανίζεται ο χαρακτήρας στο αρχείο. Αυτό που πρέπει να προσέξουμε είναι ότι το παιδί δεν θέλουμε να χειρίζεται κανένα αρχείο, δηλαδή να μην ελέγχει το ίδιο για εγκυρότητα του αρχείου ανάγνωσης. Αυτό θα γίνεται στον κώδικά μας πριν την εντολή $fork()$. Αυτό είναι εφικτό, διότι έχοντας ανοίξει το αρχείο πριν την κλήση της $fork()$, η διεργασία παιδί θα κληρονομήσει και αυτή τα δικαιώματα ανάγνωσης αυτού του αρχείου, με αποτέλεσμα να μπορεί να το επεξεργαστεί και η ίδια χωρίς να χρειάζεται να το ανοίξει εκ νέου.

Παραθέτουμε την συνάρτηση

```
int counter (int fdr, char c_check){
        char c2c = 'a', cc;
        int count = 0;

        ssize_t rcnt;
        c2c = c_check;

        while (rcnt = read(fdr, &cc, 1) > 0){
                if (rcnt == -1){
                        perror("read");
                        exit(1);
                }
                if (cc == c2c) count++;
        }
        return count;
}
```

Η συνάρτηση αυτή παίρνει ως παραμέτρους το αρχείο για διάβασμα και τον αναζητούμενο χαρακτήρα (*file, char c_check) και επιστρέφει πόσες φορές εμφανίζεται ο χαρακτήρας c_check στο αρχείο.

Παραθέτουμε το υπόλοιπο κομμάτι του κώδικα $fork4.c$

```
        int pipefd[2]; /* pipefd[0] read end of the pipe
                          pipefd[1] write end of the pipe */

        if(pipe(pipefd) == -1){ //creating the pipe (must be created before fork())
                perror("pipe");
                exit(1);
        }

        p = fork();
        int status; //argument for wait

        if (p < 0){
                perror("fork");
                exit(1);
        }

        else if (p == 0){ /*Child's time*/

                x = 10;

                mypid = getpid();
                pid_t father_pid = getppid(); //gives the pid of the child's father
                printf("Hello world! My pid is %d and my father's pid is %d\n", mypid, father_pid);

                printf("The value of x for me, the child, is: %d\n", x);

                count = counter(fdr,argv[2][0]);
                printf("The given character %c is found %d times\n", argv[2][0], count);

                close(pipefd[0]); //close the read end of pipe

                if(write(pipefd[1], &count, sizeof(count)) == -1){ //write the count into the pipe
                        perror("write");
                        exit(1);
                }

                close(pipefd[1]); //close the write end of the pipe in the child

                exit(0);
        }
        else {
                wait(&status); //wait until child terminates
```

```
                x = 3;

                printf("My child's pid is %d\n", p);
                printf("The value of x for father is: %d\n", x);

                close(pipefd[1]); //close the write end of the pipe in the parent

                int received_count;
                if(read(pipefd[0], &received_count, sizeof(received_count)) == -1){
                        perror("read");
                        exit(1);
                }
                printf("The given character %c is found %d times by my child\n", argv[2][0],received_count);

                close(pipefd[0]); //close the read end of the pipe in the parent

                exit(0);
        }
}
```

Όπως φαίνεται στον κώδικα, η διεργασία παιδί καλεί την συνάρτηση counter (δίνοντας ως ορίσματα το αρχείο για διάβασμα και τον αναζητούμενο χαρακτήρα) και επιστρέφει τον συνολικό αριθμό εμφάνισης του χαρακτήρα στην μεταβλητή count.

Παράλληλα εκτελούμε σωλήνωση περνώντας τον αριθμό εμφανίσεων του χαρακτήρα από την διεργασία παιδί στην γονεική διεργασία.

Για την επίτευξη της σωλήνωσης δημιουργούμε έναν πίνακα ακεραίων int pipefd[2] και έπειτα (αναγκαστικά πριν την fork) δημιουργούμε την σωλήνωση και εξετάζουμε αν γίνεται σφάλμα κατά την διαδικασία δημιουργίας της μέσω την εντολής:

$if(pipe(pipefd) == -1){$

$perror("pipe");$

$exit(1);$

}

Στην περίπτωση σφάλματος τερματίζουμε με σφάλμα εξόδου perror(“pipe”). Αν δεν υπάρξει σφάλμα τότε μέσω της pipe αρχικοποιείται ο πίνακας pipefd με τις δύο θέσεις. Η μια θέση αποτελεί την θέση ανάγνωσης (pipefd[0]) ενώ η άλλη την θέση εγγραφής (pipefd[1]).

Έπειτα καλείται η fork με αποτέλεσμα η διεργασία παιδί να παίρνει το δικό της αντίγραφο της pipefd.

Στην διεργασία παιδί κλείνουμε το άκρο ανάγνωσης μέσω της εντολής close(pipefd[0]) και εκτελούμε:

$if(write(pipefd[1], \&count, sizeof(count)) == -1){$

$perror("write");$

$exit(1);$

}

Συγκεκριμένα, περνάμε στην pipe από το άκρο της εγγραφής την τιμή του count, και παράλληλα πάλι ελέγχουμε για πιθανό σφάλμα. Τέλος, κλείνουμε το άκρο εγγραφής στην διεργασία παιδί close(pipefd[1]).

Περνάμε τώρα στην γονεική διεργασία. Κλείνουμε το άκρο εγγραφής γιατί θέλει να διαβάσει τον νέο αριθμό. Έπειτα, διαβάζει:

$if(read(pipefd[0], \&received\_count, sizeof(received\_count)) == -1){$

$perror("read");$

$exit(1);$

}

Όπου ελέγχει για πιθανό σφάλμα κατά την ανάγνωση στην σωλήνωση και κατά την ομαλή λειτουργία, διαβάζει από την σωλήνωση και εισάγει το περιεχόμενο στην μεταβλητή received_count.

4) Στην περίπτωση αυτή, καλούμαστε να αντικαταστήσουμε το τρέχον πρόγραμμα της διεργασίας παιδί με ένα άλλο πρόγραμμα, αυτό του ερωτήματος 1 (αναζήτηση χαρακτήρα) κάνοντας τις απαραίτητες αλλαγές, χρησιμοποιώντας παράλληλα την κατάλληλη εντολή. Αυτό φυσικά θα επιτευχθεί μέσω της εντολής $execv()$.

Παραθέτουμε τον κώδικα $fork4\_2.c$

```c
#include <stdio.h>
#include <sys/types.h> //for pid_t()
#include <unistd.h> //for fork() and getpid()
#include <stdlib.h> //for exit()
#include <sys/wait.h> //for wait()
#include <fcntl.h> //for O_RDONLY


int main(int argc, char *argv[]){

        if(argc != 4){//we call this program providing the same three arguments
        perror("Wrong calls were given");
        exit(1);}

        char cc = argv[3][0];
        int status;

        pid_t p = fork();

        if (p == -1){
        perror("fork");
        exit(1);
        }

        else if (p == 0){
        //we assign the three arguments provided by the user including the executable file in argv2
        char *argv2[] = {"./a1.1-system_calls_copy", argv[1], argv[2], &cc, NULL};

        execv(argv2[0], argv2);//executing ./a1.1-system_calls_copy (original code) with the provided arguments
        perror("execv");
        exit(1);
}

        else{
        wait(&status);
}

return 0;
}
```

Στον συγκεκριμένο κώδικα καλείται η fork όπως και στις προηγούμενες περιπτώσεις που εξετάσαμε. Όμως, η διεργασία παιδί δημιουργεί τον πίνακα argv2 στον οποίο τοποθετούνται το αρχικό εκτελέσιμο πρόγραμμα και οι υπόλοιπες παράμετροι που έδωσε ο χρήστης κατά την κλήση του τρέχον προγράμματος. Το τελευταίο όρισμα είναι το NULL που φανερώνει το τέλος του πίνακα.

Έπειτα, η διεργασία παιδί καλεί execv η οποία δέχεται ως ορίσματα το εκτελέσιμο αρχείο στο οποίο θα μεταβεί η εκτέλεση του προγράμματος καθώς και τις παραμέτρους (δύο αρχεία και τον χαρακτήρα) με τις οποίες θα κληθεί.

Αν η execv αποτύχει, τότε επιστρέφει και παίρνουμε σφάλμα.

Ωστόσο, αν πετύχει τότε δεν επιστρέφει και εκτελείται το πρώτο πρόγραμμα αναζήτησης χαρακτήρων που δημιουργήσαμε (a1.1-system_calls_copy) παίρνοντας σαν ορίσματα τις κατάλληλες παραμέτρους από τον argv2.

Άσκηση 3: Διαδιεργασιακή επικοινωνία

Ζητούμενο: Επεκτείνετε το πρόγραμμα της Άσκησης 2 ώστε να δημιουργεί P διεργασίες παιδιά (το P μπορεί να είναι ορισμένο σαν σταθερά στο πρόγραμμά σας) οι οποίες θα αναζητούν παράλληλα το χαρακτήρα στο αρχείο και η γονεϊκή διεργασία θα συλλέγει και θα τυπώνει το συνολικό αποτέλεσμα. Όταν το πρόγραμμά σας θα δέχεται Control+C από το πληκτρολόγιο (δηλαδή το σήμα SIGINT) θα πρέπει αντί να τερματίζει, να τυπώνει το συνολικό αριθμό διεργασιών που αναζητούν το αρχείο.

Η μορφή της εντολής για την εκτέλεση του προγράμματος:

$$./a.out\ 'file\_to\_search\_from'\ 'character\_to\_count'$$

Όπου

a.out το όνομα του εκτελέσιμου αρχείου της άσκησης $ask3.c$

$file\_to\_search\_from$ το αρχείο-στόχος, γνωστό ως $argv[1]$

$character\_to\_count$ ο χαρακτήρας προς αναζήτηση, γνωστό ως $argv[2][0]$

Η διαδικασία που ακολουθήσαμε για την υλοποίηση του ζητούμενου είναι η εξής:

Συνάρτηση $void\ sigint\_handler(){}$

Την συγκεκριμένη συνάρτηση την υλοποιήσαμε προκειμένου να διαχειριζόμαστε την εμφάνιση του σήματος SIGINT, το οποίο προκύπτει όταν ο χρήστης πατήσει Ctrl+C από το πληκτρολόγιο. Αυτό που κάνει η συνάρτηση αυτή, $void\ sigint\_handler(int\ sig\_number){}$, είναι να τυπώνει τον αριθμό τον διεργασιών που έχουν ξεκινήσει να τρέχουν προτού εμφανιστεί το σήμα SIGINT. Για να το κάνει αυτό έχουμε αρχικοποιήσει μια global μεταβλητή εκτός της συνάρτησης αυτής, την $int\ num\_processes\ =\ 0$, η οποία θα αποτελέσει και τον μετρητή διεργασιών παιδιών. Προκειμένου να κληθεί η συνάρτηση, έχουμε γράψει εντός της συνάρτησης $main(){}$ την εντολή $signal(SIGINT, \&sigint\_handler)$; στην οποία σαν πρώτο όρισμα δηλώνουμε το σήμα προς ανίχνευση, SIGINT, και σαν δεύτερο την συνάρτηση η οποία θα διαχειριστεί το προς ανίχνευση σήμα, την συνάρτηση $void\ sigint\_handler(){}$ δηλαδή. Μια καλύτερη υλοποίηση για την ανίχνευση του σήματος αποτελεί το system call $sigaction()$, το οποίο σαν υλοποίηση διαφέρει λιγάκι από την $signal()$, αλλά εκτελεί το ίδιο ζητούμενο, την ανίχνευση δηλαδή του εκάστοτε σήματος και την διαχείρισή του. Όπως βλέπουμε και από την εντολή man signal σε linux παράθυρο, η συνάρτηση $sigaction()$ είναι περισσότερο συμβατή με κάθε διαφορετικό distro linux. Δεν αντιμετωπίσαμε κάποιο πρόβλημα συμβατότητας εμείς, για αυτό κρατήσαμε την ‘απλή’ υλοποίηση. Κάτι ακόμα που χρειάζεται να αναφέρουμε είναι η ύπαρξη της εντολής $signal(SIGINT, SIG\_IGN)$ μέσα στην εκτέλεση του κώδικα των διεργασιών παιδιών. Αυτή είδαμε ότι χρειάζεται καθώς χωρίς αυτή βλέπαμε να καλείται η συνάρτηση $void\ sigint\_handler(){}$ περισσότερες από μια φορές, ενώ εμείς θέλουμε μόλις ανιχνευτεί το σήμα SIGINT το πρόγραμμα να διακόπτεται και να τυπώνει μόνο μια φορά των αριθμό των

διεργασιών που αναζητούσαν το αρχείο μέχρι τότε, και μετά να συνεχίζει από το σημείο που σταμάτησε.

Ακολουθούνε οι σχετικοί κώδικες όπως τους έχουμε υπολογίσει:

```
void sigint_handler(int sig_number){
	printf("Signal SIGINT found! Total number of processes searching the file: %d\n", num_processes);
}
```

Δεν υπάρχει $exit(0)$ στην συνάρτηση αυτή, γιατί μετά το Ctrl+C το πρόγραμμά μας θέλουμε να συνεχίζει από εκεί που σταμάτησε πριν την κλήση της συνάρτησης $sigint\_handler()$.

```
	signal(SIGINT, &sigint_handler);

/*	struct sigaction sa;
	sa.sa_handler = &sigint_handler;
	sigaction(SIGINT, &sa, NULL); //for portability use it is better
*/
```

Συνάρτηση $void\ end\_of\_child(){}$

Η χρησιμότητα της συνάρτησης αυτής έγκειται στην απαίτηση να γνωρίζουμε τον αριθμό των διεργασιών που μένουν να ολοκληρώσουν την αναζήτηση του χαρακτήρα-στόχου. Η αρχικοποίηση αυτής της συνάρτησης γίνεται μέσω της εντολής $signal(SIGUSR1, end\_of\_child)$. Πρόκειται, δηλαδή, για ένα σήμα το οποίο ενεργοποιείται μέσα στην συνάρτηση $main()$ και καλείται από την εκάστοτε διεργασία παιδί όταν τελειώσει την δουλειά της. Μόλις ολοκληρωθεί η αναζήτηση χαρακτήρα-στόχου από μια διεργασία παιδί, τότε καλείται η συνάρτηση $void\ end\_of\_child()$ μέσω του σήματος $kill(getppid(), SIGUSR1)$, σήμα το οποίο δηλώνει στην διεργασία πατέρα (εξ' ου και το $getppid()$) ότι τελείωσε (terminate/ kill) μια διεργασία παιδί, μείωσε κατά μια σε αριθμό τις εναπομείναντες διεργασίες.

```
void end_of_child(){
	num_processes--;
}
```

Συνάρτηση $int\ char\_counter(){}$

Αρχικά αντιγράψαμε την συνάρτηση $int\ counter(char\ *\ file, char\ c\_check){}$ από την άσκηση 2 που είχαμε υλοποιήσει με σκοπό να αναθέτουμε και εδώ στην διεργασία παιδί τον έλεγχο εμφάνισης ενός δοθέντος από τον χρήστη χαρακτήρα σε κάποιο πάλι δοθέν από τον χρήστη αρχείο. Η διαφορά, όμως, με την προηγούμενη άσκηση είναι ότι στην παρούσα ζητείται η υλοποίηση P διεργασιών παιδιών (στον δικό μας κώδικα έχουμε κάνει $\#define\ P\ 5$ και άρα θα έχουμε 5 διεργασίες παιδιά). Αυτές οι διεργασίες πρέπει να είναι υπεύθυνες για την παράλληλη αναζήτηση των εμφανίσεων του χαρακτήρα που αναζητάμε. Αυτό μας φέρνει αντιμέτωπους με το ερώτημα: «Πώς θα υλοποιήσουμε το ζητούμενο, πώς δηλαδή θα τρέχουν οι διεργασίες παιδιά 'παράλληλα' ;». Η απάντηση (μια τουλάχιστον απάντηση) στο ερώτημα αυτό είναι ο χωρισμός

του αρχείου-στόχου σε κομμάτια (segments) και η ανάθεση σε κάθε διεργασία παιδί της αναζήτησης του χαρακτήρα-στόχου μόνο σε ένα κομμάτι κώδικα. Εμείς επιλέξαμε αυτά τα κομμάτια κώδικα να χωριστούνε ισόποσα, όπως άλλωστε φαίνεται από την γραμμή κώδικα $segment\_size = file\_size / P$. Αργότερα στην αναφορά μας θα αναλύσουμε πως βρίσκουμε τις μεταβλητές $file\_size$ και $segment\_size$.
Η ανάγκη αυτή να αναθέσουμε σε κάθε διεργασία παιδί ένα συγκεκριμένο κομμάτι κώδικα συνέβαλε στην επιπλέον προσθήκη ορισμάτων (argument calls) στην προαναφερθείσα συνάρτηση $int\ char\_counter$ και συγκεκριμένα την προσθήκη των $off\_t\ start\_pos$ και $off\_t\ end\_pos$ με τα οποία θα ορίζουμε σε κάθε διεργασία το χωρίο που θέλουμε αυτή να αναζητά τον χαρακτήρα-στόχο.

Τέλος, για να κλείσουμε με την παρούσα συνάρτηση είναι σημαντικό να αναφέρουμε και να αιτιολογήσουμε ορισμένες ακόμα προσθήκες μέσα στην υλοποίησή της που την διαφοροποιεί από την αντίστοιχη συνάρτηση που χρησιμοποιήσαμε στην άσκηση 2. Συγκεκριμένα, αναφερόμαστε σε δυο ελέγχους που έχουμε προσθέσει:

$if(lseek(fdr, start\_pos, SEEK\_SET) == -1)\{$
$perror("lseek");$
$exit(1);$
$\}$
Και $if(lseek(fdr, 0, SEEK\_CUR) >= end\_pos)\ break;$

Ξεκινώντας με τον πρώτο έλεγχο, αυτός αρχικοποιεί την μεταβλητή $start\_pos$ μέσω του system call $lseek()$ και ταυτόχρονα ελέγχει για τυχόν αστοχία κατά την κλήση. Αποτυχία σημαίνει ότι η συνάρτηση επιστρέφει τιμή -1 και πηγαίνουμε στην κατάσταση $perror(“lseek”)$ και σταματάει η υλοποίηση του κώδικα από εκεί και ύστερα. Επιτυχία σημαίνει ότι η συνάρτηση θα επιστρέψει το ζητούμενο offset. Ακολουθεί η υλοποίηση της συνάρτησης και η επεξήγηση του κάθε ορίσματος:

$off\_t\ lseek(int\ fdr, off\_t\ offset, int\ whence)$
‘fdr’: το αρχείο στο οποίο θα γίνει η αναζήτηση και του οποίου μετακινούμε τον δείκτη ανάλογα του σημείο που θέλουμε να ξεκινήσει η αναζήτηση του χαρακτήρα-στόχου
‘offset’: η απόσταση που θέλουμε να μετακινήσουμε τον δείκτη
‘whence’: η θέση από την οποία θα γίνει η μετακίνηση και η οποία μπορεί να πάρει μια από τις ακόλουθες τιμές¨
‘SEEK_SET’: η απόσταση αφορά την αρχή του αρχείου
‘SEEK_CUR’: η απόσταση αφορά την τρέχουσα θέση στην οποία βρίσκεται ο δείκτης του αρχείου
‘SEEK_END’: η απόσταση αφορά το τέλος του αρχείου

Πρακτικά αυτό που γίνεται είναι η νέα αρχή αναζήτησης ορίζεται ως η απόσταση που δείχνει το whence + το offset.
Επειδή εμείς θέλουμε κάθε διεργασία να τρέχει μέσα σε ένα προκαθορισμένο κομμάτι, έρχεται και κολλάει ο δεύτερος έξτρα έλεγχος που προαναφέραμε, και συγκεκριμένα αυτό που κάνει είναι να ελέγξει αν έχουμε φτάσει στο επιθυμητό τέλος αναζήτησης για την εκάστοτε διεργασία παιδί. Αν ναι, τότε βγαίνουμε από το $while(rcnt = read(fdr, \&cc, sizeof(cc)) > 0)\{\}$ και άρα διακόπτουμε την καταμέτρηση της εκάστοτε διεργασίας παιδί και προχωράει η επόμενη την αναζήτηση από το σημείο που σταμάτησε η προηγούμενή της.

Ακολουθούνε οι σχετικοί κώδικες για τους οποίους έγινε αναφορά στο παρόν κομμάτι της αναφοράς μας:

```
off_t end_pos; //end of reading for each child process
off_t start_pos = 0; //initial starting position for the first child
off_t file_size, segment_size;
```

```
file_size = lseek(fd, 0, SEEK_END);

/* calculating the segment size for each child process */
segment_size = file_size / P;

/* if the file size is not evenly divisible we adjust the segment for the last process */
if(file_size % P != 0){ segment_size++; }
```

```
end_pos = start_pos + segment_size; //the end position for the current child
```

Παραπάνω φαίνεται πώς αρχικοποιούνται η αρχική θέση ($start\_pos$) αναζήτησης στο αρχείο, το μέγεθος κάθε segment το οποίο θα αναζητά η κάθε διεργασία παιδί, πώς υπολογίζεται, επίσης, η τελική θέση (end_pos) της αναζήτησης κάθε διεργασίας παιδί καθώς και η ισότιμη μοιρασιά αν τυχόν δεν είναι τέλεια η διαίρεση μεταξύ $file\_size$ και του αριθμού διεργασιών P.

```c
int char_counter(char *file, char c_check, off_t start_pos, off_t end_pos){ //for character count
        char c2c = 'a', cc;
        int count = 0;

        int fdr = open(file, O_RDONLY);
        if(fdr == -1){
                perror("open");
                exit(1);
        }

        if(lseek(fdr, start_pos, SEEK_SET) == -1){
                perror("lseek");
                exit(1);
        }

        ssize_t rcnt;
        c2c = c_check;

        while(rcnt = read(fdr, &cc, sizeof(cc)) > 0){
                if(cc == c2c) count++;
                if(lseek(fdr, 0, SEEK_CUR) >= end_pos) break; //stop reading if reached the end position
        }

        if(rcnt == -1){
                perror("read");
                exit(1);
        }
        close(fdr);
        return count;
}
```

Κρίνεται σημαντικό να αναφέρουμε τρεις ελέγχους που φαίνονται στην προηγούμενη φωτογραφία. Συγκεκριμένα :

I. Έλεγχος εγκυρότητας του αρχείου-στόχου προς ανάγνωση κατά το άνοιγμα

Ελέγχουμε αν έχουμε έγκυρη είσοδο για το αρχείο προς ανάγνωση, αν δηλαδή μπορούμε να το ανοίξουμε και αν ναι, τότε το ανοίγουμε σε ‘read-only’ μορφή. Ειδάλλως πετάμε error, πρόβλημα κατά το άνοιγμα του αρχείου.

Ακολουθεί ο κώδικας του συγκεκριμένου ελέγχου:

$$int\ fdr\ =\ open(file, O\_RDONLY);$$
$$if(fdr\ ==\ -1){$$
$$perror("open");$$
$$exit(1);$$
$$}$$

II. Έλεγχος εγκυρότητας του αρχείου-στόχου ανάγνωση αφού ανοιχτεί

Ελέγχουμε αν το αρχείο που ανοίξαμε μπορούμε να το διαβάσουμε. Αν ναι, τότε συνεχίζουμε με τον έλεγχο των εμφανίσεων του χαρακτήρα στόχου.

III. Έλεγχος σωστής λειτουργίας της $lseek()$ σε δυο σημεία για διαφορετικούς λόγους

Ο πρώτος έλεγχος για την $lseek()$ αφορά την έγκυρη εισαγωγή των ορισμάτων στην συνάρτηση, αν αυτή δηλαδή αντιστοιχούν ένα προς ένα στα επιθυμητά όπως έχουμε αναφέρει παραπάνω.

```
if(lseek(fdr, start_pos, SEEK_SET) == -1){
        perror("lseek");
        exit(1);
    }
```

Ο δεύτερος έλεγχος για την $lseek()$ πρόκειται για το πότε θα σταματήσει η κάθε διεργασία παιδί να αναζητά τον χαρακτήρα-στόχο, ανάλογα με το κομμάτι κώδικα που έχει οριστεί στην καθεμία μέσω $start\_pos$ και end_pos.

```
if(lseek(fdr, 0, SEEK_CUR) >= end_pos) break;
```

Συνάρτηση $main(){}$

Εδώ ξεκινάμε με την ανάλυση της κύριας συνάρτησης του προγράμματός μας, την $int\ main(int\ argc, char\ **\ argv){}$ συνάρτηση, στην οποία έχουμε αυστηρά ορίσει να δέχεται 2 ορίσματα, σαν πρώτο το όνομα του αρχείου-στόχου και σαν δεύτερο τον χαρακτήρα-στόχο. Αυτόματα το $argc$ θα γίνει ίσο με τρία με αυτόν τον τρόπο. Προκειμένου να μην υπάρξει λανθασμένη είσοδος και δεν τρέξει σωστά ο κώδικάς μας έχουμε βάλει τον εξής έλεγχο:

I. Έλεγχος σωστού αριθμού εισόδων

Αυτό υλοποιείται χάρη στην μεταβλητή $int\ argc$ που προαναφέραμε, και συγκεκριμένα στο γεγονός ότι αυτόματα γίνεται ίση με 3. Γίνεται 3 by default γιατί εμείς του εισάγουμε 2 argument calls και από μόνο του προσθέτει και το όνομα του εκτελέσιμου που τρέχουμε.
Αν τυχόν δεν εισάγουμε τον σωστό αριθμό εισόδων τότε πιάνεται το λάθος, μπαίνουμε δηλαδή στο $if(){}$ κομμάτι του κώδικα στο οποίο τυπώνουμε ένα μήνυμα στον χρήστη λέγοντάς του ότι έχουμε λάθος αριθμό inputs.

Ακολουθεί ο κώδικας του συγκεκριμένου ελέγχου:

```
if(argc != 3){
        perror("User gave us the wrong call arguments.\n");
        exit(1);
}
```

Συνεχίζοντας, συμπληρωματικά με την αναφορά που έχουμε κάνει στο κομμάτι της εργασίας μας για την υλοποίηση της συνάρτησης $int\ char\_counter(){}$, και συγκεκριμένα αναφερόμαστε στην εύρεση του συνολικού μεγέθους (σε bytes) του αρχείου-στόχου, χρειάστηκε να καλέσουμε την συνάρτηση $open()$ με την εντολή $int\ fd\ =\ open(argv[1], O\_RDONLY)$ προκειμένου να ανακτήσουμε το ζητούμενο μέγεθος όπως προαναφέραμε κάνοντας χρήση της $lseek()$. Η συνάρτηση $open()$ παίρνει σαν πρώτο όρισμα το $argv[1]$ που αντιστοιχεί στο αρχείο-στόχο (σε εκείνο θα γίνει η αναζήτηση του χαρακτήρα-στόχου και καθορίζεται από τον χρήστη κατά την εκτέλεση) και σαν δεύτερο όρισμα το O_RDONLY με το οποίο δηλώνουμε ότι θα γίνει μόνο ανάγνωση του αρχείου αυτού.

Πλέον είμαστε έτοιμοι να ξεκινήσουμε με την υλοποίηση των P διεργασιών παιδιών και την ανάθεση σε καθένα από αυτά την εύρεση του χαρακτήρα-στόχου σε ένα προκαθορισμένο κομμάτι του αρχείου-στόχου. Για τον σκοπό αυτό ξεκινάμε με την επαναληπτική loop, $for(int\ i\ =\ 0;\ i\ <\ P;\ i++){}$, με την οποία πρακτικά κατασκευάζουμε P διεργασίες αφού μέσα σε αυτή υπάρχει η εντολή $p = fork()$. Μπαίνοντας στην λούπα, θα αυξάνουμε τον μετρητή $num\_processes$++ προκειμένου να δείχνουμε ότι αυξήθηκε κατά ένα ο αριθμός των διεργασιών παιδιών. Εν συνεχεία, ελέγχουμε για πιθανό error κατά την κλήση της $fork()$, αν η κλήση δεν πετάξει $perror()$ πηγαίνουμε στον επόμενο έλεγχο $else\ if(p\ ==\ 0){}$, ο οποίος, αν ικανοποιείται, μας εντάσσει στην διεργασία παιδί. Μέσα σε αυτή την συνθήκη, υπάρχει ο κώδικας ο οποίος ανανεώνει την μεταβλητή που υποδεικνύει το τέλος ελέγχου για χαρακτήρα-στόχο σε κάθε παιδί ($end\_pos\ =\ start\_pos\ +\ segment\_size$) και ο κώδικάς ο οποίος καλεί την συνάρτηση $char\_counter(){}$ με τα επιθυμητά και ανανεωμένα ορίσματα κάθε φορά. Πριν κληθεί αυτή η συνάρτηση, βλέπουμε την ύπαρξη μια εντολής $if(i == P - 1){}$ η οποία απλά μας γλιτώνει υπολογιστικό κόστος από την άποψη ότι και χωρίς αυτή το πρόγραμμά μας θα έτρεχε κανονικά, αλλά για εξοικονόμηση πράξεων λέμε ότι όταν φτάσουμε στην τελευταία διεργασία παιδί τότε το $end\_pos\ =\ file\_size$ όπως είναι λογικό επόμενο.

Τώρα θα παρατηρήσουμε, ακόμα, την ύπαρξη μιας σειράς εντολών σχετικά με το pipeline. Αυτό μας είναι χρήσιμο για την διεργασιακή επικοινωνία των διεργασιών παιδιών με την γονεϊκή, καθώς τα δεδομένα που συλλέγει κάθε παιδί (δεδομένα: αριθμός εμφάνισης του χαρακτήρα-στόχου σε κάθε χωρίο μέσω της μεταβλητής $int\ count$) θα πρέπει μετά να τα συλλέξει ο πατέρας και να τυπώσει τον συνολικό επιθυμητό αριθμό. Για να γίνει αυτό, γυρνάμε λίγο πίσω στον κώδικά μας πριν την λούπα, στο σημείο όπου ορίζουμε το pipeline μέσω της εντολής $int\ pipesfd[P][2]$. Πρόκειται για έναν 2D πίνακα από ακεραίους, με την πρώτη διάσταση P να αναπαριστά και να αφορά κάθε διεργασία παιδί ξεχωριστά και με την δεύτερη διάσταση

αναφερόμαστε στα file descriptors του Unix system, ένα για το άκρο ανάγνωσης $pipesfd[P][0]$ και ένα για το άκρο εγγραφής $pipesfd[P][1]$. Επιστρέφοντας στην λούπα και συγκεκριμένα στο σημείο που το αφήσαμε, μένει να εγγράψουμε στο pipeline του κάθε παιδιού τον αριθμό που επιστρέφει η κλήση της συνάρτησης $char\_counter(){}$ για κάθε διαδικασία παιδί. Ακολουθεί ο κώδικας υλοποίησης εγγραφής:

$$
\begin{aligned}
&close(pipesfd[i][0]);\\
&if(write(pipesfd[i][1], \&count, sizeof(count)) == -1){\\
&\quad perror("write");\\
&\quad exit(1);\\
&\}\\
&close(pipesfd[i][1]);
\end{aligned}
$$

Πρώτα κλείνουμε το άκρο ανάγνωσης προκειμένου να εκμεταλλευτούμε και να αξιοποιήσουμε καλύτερα τους πόρους του συστήματος όταν δεν χρειάζεται η ανάγνωση από pipeline και να δείξουμε ότι αναμένεται εγγραφή προτού γίνει κάποια ανάγνωση και προκληθεί σφάλμα από τυχόν άδειο για παράδειγμα pipeline. Στην συνέχεια, με την συνάρτηση $write(pipesfd[i][1], \&count, sizeof(count))$ γίνεται η εγγραφή του count στην θέση $pipesfd[i][1]$, όπου i η εκάστοτε διεργασία παιδί (ξεκινώντας από το 0). Ταυτόχρονα γίνεται ο έλεγχος για σωστή κλήση της συνάρτησης $write()$. Μόλις η διαδικασία αυτή ολοκληρωθεί κλείνουμε το άκρο εγγραφής.
Τέλος, προτού βγούμε από το κομμάτι κώδικα των διεργασιών παιδιών, δηλώνουμε μέσω σήματος ότι μια διεργασία παιδί ολοκλήρωσε την αναζήτηση, καλείται η συνάρτηση $void\ end\_of\_child()$, η οποία με την σειρά της εκτελεί $num\_processes --$ δείχνοντας ότι μια διεργασία παιδί τελείωσε. Πριν βγούμε από την λούπα δεν ξεχνάμε να ανανεώσουμε την μεταβλητή που καθορίζει, εν μέρει, την αρχική θέση εύρεσης χαρακτήρα-στόχου για κάθε παιδί ($start\_pos\ +=\ segment\_size$).

Τελειώνοντας, πλέον, με τον κώδικα για τις διεργασίες παιδιά, περνάμε στην υλοποίηση του κώδικα για την διεργασία γονιού. Αρχικά, υπάρχει ένα while loop με το οποίο αδρανοποιούμε την διεργασία γονιού μέχρι να τελειώσει κάθε διεργασία παιδί χάρη στην εντολή $wait(\&status)$. Ακολουθεί ο κώδικας:

$$
\begin{aligned}
&int\ terminated\_processes\ =\ 0;\\
&while\ (terminated\_processes\ <\ P)\ \{\\
&\quad wait(\&status);\ //\ Decrement\ for\ each\ child\ process\ that\ finishes\\
&\quad terminated\_processes\ ++;\\
&\}
\end{aligned}
$$

Σκοπός του γονέα, όπως έχουμε ήδη αναφέρει, είναι η συλλογή των δεδομένων που μαζεύει κάθε διεργασία παιδί, η συλλογή, δηλαδή, του αριθμού εμφάνισης του χαρακτήρα-στόχου από κάθε διεργασία παιδί και ύστερα η εκτύπωση στην οθόνη του χρήστη (terminal) τον συνολικό αυτό αριθμό. Η διεργασία γονέα επικοινωνεί με τις διεργασίες παιδιά μέσω του pipeline που ήδη έχουμε δημιουργήσει. Αυτό που μένει είναι ο γονέας να διαβάσει από το άκρο ανάγνωσης τα δεδομένα που έχει εγγράψει κάθε παιδί.

Ακολουθεί ο κώδικας ανάγνωσης:

$$
\begin{aligned}
&for(int\ i\ =\ 0;\ i\ <\ P;\ i++)\{\\
&\quad close(pipesfd[i][1]);\\
&\quad if(read(pipesfd[i][0], \&received\_char\_count, sizeof(received\_char\_count))\ ==\\
&\qquad\quad =\ -1)\{\\
&\qquad perror("read");\\
&\qquad exit(1);\\
&\quad \}\\
&\quad total\_count\ +=\ received\_char\_count;\\
&\quad close(pipesfd[i][0]);\\
&\}
\end{aligned}
$$

Συγκεκριμένα, αυτό που κάνουμε είναι να ορίζουμε μια μεταβλητή $int\ received\_char\_count$, στην οποία θα αποθηκεύουμε κάθε φορά τον ακέραιο που έχει αποθηκεύσει κάθε παιδί στο pipeline και θα τον προσθέτουμε στην μεταβλητή $total\_count$ την οποία έχουμε αρχικοποιήσει σε τιμή μηδέν ($int\ total\_count\ =\ 0$). Μέσα στον λούπα που φαίνεται παραπάνω, η οποία τρέχει τόσες φορές όσες ο αριθμός P των διεργασιών παιδιών, κλείνουμε, αρχικά, το άκρο εγγραφής του σωλήνα (pipeline) για αντίστοιχους λόγους με αυτούς που αναφέραμε για το κλείσιμο του άκρου ανάγνωσης όταν αναμένεται εγγραφή, και ύστερα καλούμε την συνάρτηση $read(\{\})$. Με την συνάρτηση αυτή αποθηκεύουμε τον αριθμό που υπάρχει στο $pipesfd[i][0]$ και τον αποθηκεύουμε στην μεταβλητή $received\_char\_count$. Έτσι, καταλήγουμε να έχουμε το επιθυμητό συνολικό αποτέλεσμα στην μεταβλητή $total\_count$. Δεν ξεχνάμε και εδώ να κλείσουμε το άκρο ανάγνωσης του pipeline όταν τελειώνουμε με την ανάγνωση για κάθε παιδί. Να επισημάνουμε το εξής: καίριας σημασίας το $exit(0)$ που έχουμε βάλει προτού βγούμε από το $else\ i(p\ ==\ 0)\{\}$. Αυτό είναι υπεύθυνο για την ομαλή λειτουργία το fork() μέσα στην for loop. Αυτό γιατί? Γιατί ειδάλλως μετά την πρώτη επανάληψη, μαζί με την γονεϊκή διεργασία και το κάθε παιδί θα καλούσε την $fork()$ φτιάχνοντας έξτρα διεργασίες, με αποτέλεσμα να έχουμε αρχικά 1 διεργασία παιδί, μετά δυο, μετά τέσσερεις, μετά οκτώ ... με αποτέλεσμα να μην τρέχει σωστά το πρόγραμμά μας.

Βγαίνοντας από την λούπα, τυπώνουμε το αποτέλεσμα που θέλουμε ως εξής:

$$printf("The\ total\ number\ of\ times\ character\ \%c\ occurred: \%d\backslash n", argv[2][0], total\_count)$$

Να επισημάνουμε ότι τα $sleep()$ που υπάρχουν στον κώδικά μας είναι για τον έλεγχο του σήματος SIGINT, καθυστερούν την εκτέλεση του κώδικα τόσα δευτερόλεπτα όσα αναγράφονται στην παρένθεση ώστε ο προγραμματιστής να προλαβαίνει να πατήσει Ctrl+C. Το πρώτο, μάλιστα, sleep(10) που υπάρχει εντός της διεργασία παιδί συμβάλει στην παράλληλη εκτέλεση των διεργασιών παιδιών.

Ακολουθεί ο συνολικός κώδικας για την $main(){}$:

```
int main(int argc, char **argv){

	if(argc != 3){
		perror("User gave us the wrong call arguments.\n");
		exit(1);
	}

	signal(SIGINT, &sigint_handler);
	signal(SIGUSR1, end_of_child);

/*	struct sigaction sa;
	sa.sa_handler = &sigint_handler;
	sigaction(SIGINT, &sa, NULL); //for portability use it is better
*/
	int pipesfd[P][2]; //array from pipelines for each child process
	for(int i = 0; i < P; i++){
		if(pipe(pipesfd[i]) == -1){
			perror("pipe");
			exit(1);
		}
	}

	pid_t p;

	int total_count = 0;
	int status;

	off_t end_pos; //end of reading for each child process
	off_t start_pos = 0; //initial starting position for the first child
	off_t file_size, segment_size;

	/* opening the file to get its size */
	int fd = open(argv[1], O_RDONLY);
	if(fd == -1){
		perror("open");
```

```
        /* opening the file to get its size */
        int fd = open(argv[1], O_RDONLY);
        if(fd == -1){
                perror("open");
                exit(1);
        }
        file_size = lseek(fd, 0, SEEK_END);

        /* calculating the segment size for each child process */
        segment_size = file_size / P;

        /* if the file size is not evenly divisible we adjust the segment for the last process */
        if(file_size % P != 0){ segment_size++; }

        for(int i = 0; i < P; i++){

//              num_processes++;

                p = fork();

                num_processes++;

                if(p == -1){
                        perror("fork");
                        exit(1);
                }
                else if(p == 0){
                        signal(SIGINT, SIG_IGN);

                        sleep(10); //testing

                        end_pos = start_pos + segment_size; //the end position for the current child

                        if (i == P-1)end_pos = file_size;
                        printf("Child process %d searching the file ...\n", getpid());
```

```
                        printf("Child process %d searching the file ...\n", getpid());
                        printf("Child process %d starting from position %ld to %ld\n", getpid(), start_pos, end_pos);

                        int count = char_counter(argv[1], argv[2][0], start_pos, end_pos);

//                      sleep(2); //for 2 seconds pause, testing SIGINT

                        printf("Child process %d found character %c %d time(s)\n", getpid(), argv[2][0], count);

                        close(pipesfd[i][0]);
                        if(write(pipesfd[i][1], &count, sizeof(count)) == -1){
                                perror("write");
                                exit(1);
                        }
                        close(pipesfd[i][1]);
                        kill(getppid(), SIGUSR1);

                        exit(0);
                }
                start_pos += segment_size; /*parent process updates the starting
                                             position for the next child*/
                sleep(2);
        }
        //parent now
        int received_char_count;

//      signal(SIGINT, &sigint_handler);
        int terminated_processes = 0;
        while (terminated_processes < P) {
        wait(&status); // Decrement for each child process that finishes
        terminated_processes ++;
        }

        for(int i = 0; i < P; i++){
                close(pipesfd[i][1]);
                if(read(pipesfd[i][0], &received_char_count, sizeof(received_char_count)) == -1){
```

```
        for(int i = 0; i < P; i++){
                close(pipesfd[i][1]);
                if(read(pipesfd[i][0], &received_char_count, sizeof(received_char_count)) == -1){
                        perror("read");
                        exit(1);
                }
                total_count += received_char_count;

                close(pipesfd[i][0]);
        }
        printf("The total number of times character %c occurred: %d\n", argv[2][0], total_count);
        printf("The total number of processes was %d\n", P);

        exit(0);
}
```

Άσκηση 4: Εφαρμογή παράλληλης καταμέτρησης χαρακτήρων

Ζητούμενο: Υλοποιήστε μια ολοκληρωμένη εφαρμογή παράλληλης καταμέτρησης χαρακτήρων σε ένα αρχείο. Η εφαρμογή θα δέχεται από το χρήστη το όνομα του αρχείου και το χαρακτήρα προς αναζήτηση και κατά τη διάρκεια της εκτέλεσής της θα μπορεί να δέχεται από το χρήστη εντολές για: α) πρόσθεση/αφαίρεση εργατών (διεργασιών) αναζήτησης, β) παρουσίαση πληροφορίας σε σχέση με τους εργάτες που συμμετέχουν στην αναζήτηση και γ) ενημέρωση σχετικά με την πρόοδο της αναζήτησης (ποσοστό ολοκλήρωσης εργασίας και αριθμό χαρακτήρων που έχουν βρεθεί μέχρι στιγμής).

Εισαγωγή:

Στην συγκεκριμένη άσκηση ζητείται η υλοποίηση ενός προγράμματος για παράλληλη καταμέτρηση χαρακτήρων. Αυτό που την διαφοροποιεί από την άσκηση 3 είναι ότι τώρα πρέπει να δομήσουμε κατάλληλα τον κώδικά μας στα τρία ακόλουθα μέρη:

1. Πρόγραμμα για το front end, διεργασία η οποία περιλαμβάνει την επικοινωνία με τον χρήστη, υπεύθυνη να διευθύνει κατάλληλα την κάθε εντολή εισόδου, είτε εξυπηρετώντας την είτε προωθώντας την στο δεύτερο πρόγραμμα αυτό για τον dispatcher.
2. Πρόγραμμα για το dispatcher, διεργασία η οποία είναι υπεύθυνη για τον κατάλληλο διαμοιρασμό εργασίας στους εργάτες (πρόγραμμα για τους workers πιο κάτω). Ο dispatcher δέχεται όσες εντολές προωθεί το front end σε αυτόν, συγκεκριμένα εντολές για πρόσθεση ('increase x') ή αφαίρεση ('decrease x') x εργατών, εντολή για τερματισμό ('terminate') ενός εργάτη, εντολή για την ως τώρα πορεία του προγράμματος ('progress'), εντολή για την παρούσα κατάσταση των εργατών ('info') και τέλος την εντολή 'exit' με την οποία εκτελείται ο κώδικάς μας χωρίς να δίνεται άλλη ευκαιρία στον χρήστη για περεταίρω εντολές. Πέρα από την επικοινωνία μεταξύ dispatcher και front end, ο dispatcher θα πρέπει να κατανέμει ορθά την δουλειά στους διαθέσιμους εργάτες, να συλλέγει την πληροφορία από αυτούς και να επιστρέφει το συνολικό αποτέλεσμα πίσω στο front end.
3. Πρόγραμμα για τους workers, διεργασίες δυναμικά μεταβαλλόμενου αριθμού (ο αριθμός των εργατών αρχικοποιείται από εμάς τους προγραμματιστές και αυξομειώνεται από τον χρήστη με προαναφερθείσες εντολές), καθεμία διεργασία υπεύθυνη για την αναζήτηση του χαρακτήρα-στόχου που έχει καθοριστεί από τον χρήστη στο προκαθορισμένο (από τον dispatcher) χωρίο του αρχείου-στόχου. Το αποτέλεσμα που έχει συλλέξει κάθε διεργασία εργάτη στέλνεται πίσω στον dispatcher.

Για την υλοποίηση της ζητούμενης εφαρμογής - καταμέτρηση ενός χαρακτήρα-στόχου μέσα σε ένα αρχείο-στόχο - το πρόγραμμα του front end κάνει $execv$ μέσα στην $fork()$ του παιδιού και αρχίζει το πρόγραμμα του dispatcher, ομοίως και το πρόγραμμα του dispatcher κάνει $execv$ μέσα στην $fork()$ του παιδιού και αρχίζει το πρόγραμμα του worker. Αυτά θα αναλυθούν παρακάτω.

Πρόγραμμα για το front end:

Κατά κύριο λόγο το συγκεκριμένο πρόγραμμα αφορά στην επικοινωνία μεταξύ του χρήστη και της εφαρμογής που υλοποιήσαμε, το οποίο συνεπάγεται την προώθηση ή την υλοποίηση αυτής της πληροφορίας εισόδου στην διεργασία παιδί (πρόγραμμα dispatcher).

Ακολουθούν οι global δηλωμένες μεταβλητές:

```
#define most_workers 20

volatile sig_atomic_t responseReady = 0;

int pipefdwrite[2];
int pipefdread[2];
```

$\#define\ most\_workers\ 20$ αφορά το άνω όριο για τον αριθμό των εργατών που μπορούμε να έχουμε

$volatile\ sig\_atomic\_t\ responseReady\ =\ 0$ αφορά στην επικοινωνία με σήματα (ενεργοποιείται σε τιμή 1 όταν έχουμε απάντηση από τον dispatcher)

$int\ pipefdwrite[2], int\ pipefdread[2]$ αφορούν στην επικοινωνία μεταξύ των διεργασιών front end και dispatcher, με το $pipewrite$ θα στέλνει ο front end το input του χρήστη στον dispatcher, ενώ με το $piperead$ θα διαβάζει το front end την απάντηση που θα του στέλνει ο dispatcher

Ακολουθεί η ανάλυση των συναρτήσεων που υλοποιήσαμε:

$$void\ signalHandler(int\ sig)\{\}$$

Είναι η συνάρτηση που είναι υπεύθυνη να ανανεώνει την μεταβλητή $responseReady = 1$ προκειμένου να δεχόμαστε μετά σήμα από τον dispatcher ότι έχει έτοιμη την απάντηση προς εκτύπωση. Αυτή ενεργοποιείται με το σήμα SIGUSR2.

```
void signalHandler (int sig){
    if (sig == SIGUSR2)
        responseReady = 1;
    printf("\nsignal received back\n");
}
```

$void\ callend(){}$

Είναι η συνάρτηση που καλείται μέσα στην $void\ signalHandler2(int\ sig){}$ (συνάρτηση η οποία θα αναλυθεί στη συνέχεια) και είναι υπεύθυνη για την εκτύπωση του συνολικού αριθμού εμφάνισης του χαρακτήρα-στόχου μετά τις αναζητήσεις που έγιναν στο αρχείο-στόχο. Αυτή συλλέγει τα δεδομένα από το pipeline που έχει εγκατασταθεί για την διεργασιακή επικοινωνία μεταξύ front_end και dispatcher.

```
void callend(){
    wait(NULL);
    int front_end_count;
        if (read(pipefdread[0], &front_end_count, sizeof(front_end_count)) == -1)
    {
        perror("read");
        exit(1);
    }

    printf("Total_count is %d\n", front_end_count);
    exit(0);
}
```

Είναι σημαντικό να αναφέρουμε την σημασία της ύπαρξης του $wait(NULL)$ εντός αυτής της συνάρτησης. Αυτό θα αναφερθεί στο τέλος της ανάλυσης του παρόντος προγράμματος για τον front end (βλ. σελίδα 29).

$void\ signalHandler2(int\ sig){}$

Είναι η συνάρτηση που ενεργοποιείται με το σήμα SIGUSR1 και είναι υπεύθυνη να μας ενημερώσει για την ολοκλήρωση της διεργασίας dispatcher. Μόλις τελειώσει ο dispatcher, αυτός στέλνει ένα σήμα στο front_end ότι τελείωσα την εκτέλεσή μου έχω απάντηση, $kill(getppid(), SIGUSR1)$ (διεργασία πατέρα του, για αυτό βλέπουμε $getppid()$ στην εντολή). Αυτή η συνάρτηση καλεί την $callend()$και πρακτικά τελειώνει η εφαρμογή μας με την εκτύπωση του μηνύματος.

```
void signalHandler2 (int sig){
    printf("\nThe dispatcher has ended\n");

    callend();

}
```

$int\ main(int\ argc, char\ **\ argv)\{\}$

Μπαίνουμε τώρα στην κύρια υλοποίηση του κώδικα για το front_end στην συνάρτηση $main()$ η οποία δέχεται τρία ορίσματα, το εκτελέσιμο αρχείο ‘./front_end’, το αρχείο-στόχο στο οποίο θα γίνει η αναζήτηση του χαρακτήρα-στόχου και τον χαρακτήρα-στόχο. Γίνεται, λοιπόν, στην αρχή ο έλεγχος για έγκυρο input.

```
int main (int argc, char **argv){


    if (argc != 3){
        perror("wrong argumenrs");
        exit(1);
    }
```

Στην συνέχεια ενεργοποιούμε τα σήματα SIGUSR1 και SIGUSR2 με τις ακόλουθες εντολές προκειμένου να είναι διαθέσιμα προς χρήση είτε για να δηλωθεί input από τον χρήστη ($SIGUSR1 -> kill(dispatcher, SIGUSR1)$) είτε για να δηλωθεί έτοιμη απάντηση από τον dispatcher ($SIGUSR2 -> kill(getppid(), SIGUSR1)$)

```
signal(SIGUSR1, signalHandler2);
signal(SIGUSR2, signalHandler);
```

Πριν καλέσουμε την συνάρτηση $fork()$ αυτό που κάνουμε είναι να ελέγξουμε αν οι σωληνώσεις μας είναι έγκυρες. Αυτό το κάνουμε με τον ακόλουθο τρόπο ταυτόχρονα και για τις δυο:

```
 if (pipe(pipefdwrite) == -1 || pipe(pipefdread) == -1)
{
    perror("pipe");
    exit(1);
}
```

Ήρθε η ώρα που κάνουμε $fork()$ και δημιουργούμε την διεργασία παιδί, τον dispatcher. Σε αυτόν εισερχόμαστε αν ικανοποιείται η συνθήκη $else\ if\ (dispatcher\ ==\ 0)$ και τότε κάνουμε τα εξής:

Κλείνουμε το άκρο εγγραφής στην σωλήνωση $pipefdwrite$ που αφορά εγγραφή του σήματος input από τον front end

Κλείνουμε το άκρο ανάγνωσης στην σωλήνωση $pipefdread$ που αφορά ανάγνωση απάντησης από τον dispatcher

Και ακολουθεί το σημαντικότερο κομμάτι της σωλήνωσης:

Φτιάχνουμε ‘αντίγραφα’ των σωληνώσεων με την εντολή $dup2$, και συγκεκριμένα το μεν άκρο ανάγνωσης της σωλήνωσης $pipefdwrite$ θα αντικαταστήσει το STDIN_FILENO, το δε άκρο εγγραφής της σωλήνωσης $pipefdread$ με το STDOUT_FILENO

Κατασκευάζουμε στην συνέχεια τον πίνακα $argv2[]$, ο οποίος περιέχει σαν input σε μορφή συμβολοσειράς και περιέχει τα ορίσματα της κλήσης του προγράμματος του dispatcher. Για να μεταβούμε στην εκτέλεση του κώδικα του dispatcher, καλούμε $execv(“./dispatcher”, argv2)$.

```
else if (dispatcher == 0){

    close (pipefdwrite[1]);
    close (pipefdread[0]);

    dup2(pipefdwrite[0], STDIN_FILENO);
    close(pipefdwrite[0]);

    dup2(pipefdread[1], STDOUT_FILENO);
    close(pipefdread[1]);

    char *argv2[] = {"./dispatcher", argv[1], argv[2], NULL};
    execv(argv2[0], argv2);//executing ./dispatcher (original code) with the provided arguments
    perror("execv");
    exit(1);
}
```

Στο κομμάτι του front end τώρα, κλείνουμε το άκρο ανάγνωσης της μιας σωλήνωσης και εισερχόμαστε σε ένα while loop. Σε αυτό το loop τυπώνουμε τις διαθέσιμες εντολές για να γνωρίζει ο εκάστοτε χρήσης τις υπάρχουσες εντολές για να χρησιμοποιήσει σωστά την εφαρμογή μας.

```
else {//This is the front_end part now

      close(pipefdwrite[0]);  // Close the read-end of the pipe

  while (1) {
        printf("Enter a command \n");
        printf("\nThe accessible commands are: \n");
        printf("1) 'increase x (enter)': adding x workers\n");
        printf("2) 'decrease x (enter)': killing x workers\n");
        printf("3) 'terminate (enter)': killing the first active worker\n");
        printf("4) 'progress (enter)': for file percentage and count of character till now\n");
        printf("5) 'info (enter)': for info\n");
        printf("6) 'exit (enter)': to proceed\n");
```

Ύστερα, αυτό που κάνουμε είναι να περιμένουμε αρχικά μέχρι η διεργασία παιδί (dispatcher) να αλλάξει κατάσταση. Αυτό επιτυγχάνεται μέσω της εντολής $pid\_t\ result =$

$waitpid(dispatcher, \&status, WNOHANG)$, με το $WHOHANG$ να επιβάλει άμεση επιστροφή αν η διεργασία παιδί δεν έχει αλλάξει κατάσταση. Αν ο dispatcher δεν έχει ολοκληρώσει την εκτέλεσή του αλλά τρέχει ακόμα, τότε η μεταβλητή result παίρνει την τιμή 0.
Από αυτό φαίνεται ότι με τον επόμενο έλεγχο, αυτό που καταφέρνουμε είναι να κάνουμε break, δηλαδή να βγούμε από την loop που τυπώνονται οι διαθέσιμες εντολές όταν πλέον δεν έχει νόημα έξτρα input από τον χρήστη, αφού η διαδικασία εύρεσης του χαρακτήρα-στόχου έχει ολοκληρωθεί (ο dispatcher έχει ολοκληρώσει).

```
char input[100];
int status;
pid_t result = waitpid(dispatcher, &status, WNOHANG);
if (result != 0) {
    break;
}
```

Ύστερα δεχόμαστε από τον χρήστη το input command και ελέγχουμε αν αυτό είναι έγκυρο και αν αυτό πρόκειται για την εντολή 'exit'. Αν συμβαίνει κάτι από τα δυο τότε κάνουμε πάλι break και βγαίνουμε από το while loop.

Στην συνέχεια, με την εντολή $input[strcspn(input, "\n")] = 0$ αυτό που κάνει η συνάρτηση $strcspn()$ είναι να αναζητάει στο input που του δώσαμε τον χαρακτήρα newline ('\n) και παράλληλα να τον αντικαθιστούμε με το null terminator ('\0').

Περνάμε μετά μέσω σωλήνωσης το input του χρήστη ώστε να είναι διαθέσιμο άκρο ανάγνωσης του dispatcher, προκειμένου αυτός να το διαχειριστεί κατάλληλα και τον ενημερώνουμε ότι έχουμε έτοιμο input.

```
// Remove newline
input[strcspn(input, "\n")] = 0;
printf("sending signal %s", input);

if (write(pipefdwrite[1], input, strlen(input) + 1) == -1) {
    perror("write");
    break;
 }

// Notify the dispatcher that there's new data
kill(dispatcher, SIGUSR1);
```

Όσο δεν έχουμε απάντηση από τον dispatcher περιμένουμε. Όταν πλέον έχουμε απάντηση έτοιμη από τον dispatcher, αρχικοποιούμε ξανά την μεταβλητή $responseReady = 0$ και διαβάζουμε την απάντηση από τον dispatcher Μέσω του άκρου ανάγνωσης της κατάλληλης

σωλήνωσης. Κάνουμε $sizeof(response) - 1$ για να επιβεβαιώσουμε ότι θα υπάρχει χώρος για null terminator στο τέλος της απάντησης μας (πρακτικά να χωράει ένα '\n' στο τέλος). Αφότου διαβάσουμε την απάντηση, προσθέτουμε τον null terminator στο τέλος, και μπορούμε να την τυπώσουμε.

```
        while (!responseReady) {
        sleep(1);
    }
        responseReady = 0;
        char response[1024];

      ssize_t nbytes = read(pipefdread[0], response, sizeof(response) - 1); // Leave space for null terminator
            if (nbytes == -1) {
             perror("read");
                exit(1);
            }
        else {
    response[nbytes] = '\0'; // Null-terminate the string
}
        printf("%s\n", response);
        printf("\n");
```

Αυτό που μένει τώρα είναι έξω από την while loop, αν έχει προκύψει κάποιο break από αυτά που αναφέραμε, τυπώνουμε το μήνυμα

```
printf("User chose to exit. Waiting for the dispatcher to end\n");
```

και περιμένουμε μέχρι τον τερματισμό της διεργασίας παιδιού (dispatcher) ώστε να γίνει ομαλή αποδέσμευση μνήμης.

Εδώ ερχόμαστε να αιτιολογήσουμε το wait(NULL) που είδαμε στην συνάρτηση $void\ callend()$. Στον κώδικά μας, αν δεν γίνει break στο while(1) που αναλύσαμε με την εκτύπωση των εντολών, τότε δεν θα αποδεσμευτεί η μνήμη που καταλαμβάνει η διεργασία παιδί όπως θα έπρεπε να γίνει μόλις ολοκληρώσει την εκτέλεσή της, και αυτό επειδή δεν θα μεταβεί ποτέ στις τελευταίες γραμμές κώδικα χωρίς να υπάρξει κάποιο break. Αυτό επιλύεται προσθέτοντας σε εκείνη την συνάρτηση αυτό το wait, συνάρτηση την οποία θα καλέσει σίγουρα το πρόγραμμά μας αν δεν υπάρξει κάποιο πρόβλημα.

```
 GNU nano 5.4                                                front_end.c
int main (int argc, char **argv){


    if (argc != 3){
        perror("wrong argumenrs");
        exit(1);
    }

    signal(SIGUSR1, signalHandler2);
    signal(SIGUSR2, signalHandler);

     if (pipe(pipefdwrite) == -1 || pipe(pipefdread) == -1)
    {
        perror("pipe");
        exit(1);
    }

    pid_t dispatcher;

    dispatcher = fork();

    if (dispatcher < 0){
        perror("fork");
        exit(1);
    }

    else if (dispatcher == 0){

        close (pipefdwrite[1]);
        close (pipefdread[0]);

        dup2(pipefdwrite[0], STDIN_FILENO);
        close(pipefdwrite[0]);

        dup2(pipefdread[1], STDOUT_FILENO);
        close(pipefdread[1]);
```

```
       char *argv2[] = {"./dispatcher", argv[1], argv[2], NULL};
       execv(argv2[0], argv2);//executing ./dispatcher (original code) with the provided arguments
       perror("execv");
       exit(1);
   }

   else {//This is the front_end part now

        close(pipefdwrite[0]);  // Close the read-end of the pipe

     while (1) {
           printf("Enter a command \n");
           printf("\nThe accessible commands are: \n");
           printf("1) 'increase x (enter)': adding x workers\n");
           printf("2) 'decrease x (enter)': killing x workers\n");
           printf("3) 'terminate (enter)': killing the first active worker\n");
           printf("4) 'progress (enter)': for file percentage and count of character till now\n");
           printf("5) 'info (enter)': for info\n");
           printf("6) 'exit (enter)': to proceed\n");
           char input[100];
           int status;
           pid_t result = waitpid(dispatcher, &status, WNOHANG);
           if (result != 0) {
               break;
           }

           if (fgets(input, sizeof(input), stdin) == NULL || strncmp(input, "exit", 4) == 0) {
               break;  // Exit loop if user types 'exit' or on error
           }

           // Remove newline
           input[strcspn(input, "\n")] = 0;
```

```
            printf("sending signal %s", input);

            if (write(pipefdwrite[1], input, strlen(input) + 1) == -1) {
                perror("write");
                break;
             }

            // Notify the dispatcher that there's new data
            kill(dispatcher, SIGUSR1);

        while (!responseReady) {
        sleep(1);
    }
        responseReady = 0;
        char response[1024];

      ssize_t nbytes = read(pipefdread[0], response, sizeof(response) - 1); // Leave space for null terminator
            if (nbytes == -1) {
             perror("read");
                exit(1);
            }
        else {
    response[nbytes] = '\0'; // Null-terminate the string
}
        printf("%s\n", response);
        printf("\n");

        }

        printf("User chose to exit. Waiting for the dispatcher to end\n");

        close(pipefdwrite[1]);  // Close the write-end of the pipe
        wait(NULL);  // Wait for the dispatcher to ensure it has finished
    }

    }
```

Πρόγραμμα για τον dispatcher:

Περνάμε πλέον στο κομμάτι του Dispatcher.

Όπως αναφέραμε προηγουμένως, ο dispacther βρίσκεται σε διαφορετικό αρχείο από το front end, και ουσιαστικά μέσω της execv, η ροή εκτέλεσης της διεργασίας dispatcher μεταβαίνει στο εκτελέσιμο αρχείο του dispatcher.

Αρχικά αναφέρουμε πώς στον κώδικα του dispatcher έχουμε αναθέσει πολλές global μεταβλητές ώστε αυτές να είναι ορατές σε όλες τις συναρτήσεις που έχουμε υλοποιήσει αλλά και στα σήματα.

Παραθέτουμε τις global μεταβλητές

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <errno.h>

#define most_workers 20//This is the limit for most workers in the program


   typedef struct // This struct contains information about all workers
{              // Typedef because we will store this information in an array
    pid_t pid;
    off_t workerstart;
    off_t workerend;
    int active;
} Task;

volatile sig_atomic_t signalReceived = 0;

char sending[100];

Task WORKERS[most_workers];//This is the list that contains the information about the Workers

int current_worker;//This will be useful in the loop of the dispatcher that
volatile sig_atomic_t command_available = 0;//This receive positive value whenver siganl is received from front_end
volatile off_t segment_size;
off_t file_size, start, end;
volatile int remaining_workers = 10;
volatile int total_current_workers = 10;//How many workers we will have in total
int terminated_children = 0;//How many children have exited correcrtly
int count = 0;//How many times the character has been found

     int pipefdread[most_workers][2];
```

Ορισμένες μεταβλητές χαρακτηρίζονται και volatile. Αυτό επί της ουσίας υποδεικνύει ότι οι τιμές των συγκεκριμένων μεταβλητών, ενδεχομένως, να επηρεάζονται από εξωτερικά σήματα και όχι από την φυσική ροή του προγράμματος.

```
void signalHandler(int sig)//When the front end sends command it sends it with SIGUSR1 signal
    if (sig == SIGUSR1)
    command_available = 1;

}
```

Όποτε το front end αναζητά πληροφορία αποστέλλει στον dispatcher το σήμα SIGUSR1 το οποίο επεξεργάζεται ο signal handler που δέχεται το σήμα SIGUSR1 και αλλάζει την μεταβλητή command_available σε 1. Όπως είδαμε προηγουμένως, η τιμή 1 στην μεταβλητή command_available υποδηλώνει πώς έχει ζητήσει κάποια πληροφορία το front end.

```
int main (int argc, char **argv){


   signal(SIGUSR1, signalHandler);
```

Εντός της κύριας συνάρτησης του dispatcher (στην αρχή κιόλας της ροής εκτέλεσης) δηλώνουμε ότι η επεξεργασία του σήματος SIGUSR1 θα γίνεται στην συνάρτηση signalHandler που είδαμε ακριβώς παραπάνω.

Προχωράμε στην κύρια συνάρτηση του προγράμματος η οποία χρησιμοποιεί πολλές από τις λειτουργίες που υλοποιήσαμε και στην άσκηση 3. Πιο συγκεκριμένα, πάλι έχουμε αρχικά total_current_workers και έτσι χωρίζουμε το off_t segment_size αρχικά σε total_current_workers κομμάτια. Αυτό το επιτυγχάνουμε φυσικά μέσω της εντολής

$segment\_size = file\_size / total\_current\_workers;$

Τονίζουμε ότι οι total_current_workers είναι οι συνολικοί workers που είτε έχουν χρησιμοποιηθεί, είτε θα χρησιμοποιηθούν.

Μετά εισερχόμαστε στην κύρια λούπα του dispatcher που μοιράζει την δουλεία στους workers.

Η κύρια διαφορά σε σχέση με την Άσκηση 3, είναι πώς πλέον η συνθήκη ελέγχου στην λούπα γίνεται κατ' αυτόν τον τρόπο

$\textbf{\textit{while}}\ (start < file\_size)$

Αυτό το χρειαζόμαστε γιατί δεν έχουμε σταθερό αριθμό εργατών πλέον. Πρέπει δηλαδή το πρόγραμμά μας να ανταποκρίνεται στην δυναμική αλλαγή των workers.

Εντός της λούπας αυτής, καλείται το fork μέσω της εντολής: $worker = fork();$

Έτσι, η διεργασία του κάθε εργάτη εισέρχεται στο παρακάτω τμήμα κώδικα λόγω της συνθήκης worker == 0 (διεργασία παιδί)

```
else if (worker == 0){//This is the code for worker
        close (pipefdread[current_worker][0]);

        dup2(pipefdread[current_worker][1], STDOUT_FILENO);//This redirecets the output of the worker to the dispatcher
        close(pipefdread[current_worker][1]);

        char start_str[20];
        char end_str[20];

        snprintf(start_str, sizeof(start_str), "%lld", (long long)start);
        snprintf(end_str, sizeof(end_str), "%lld", (long long)end);
        char cc = argv[2][0];
        if (a == -1){
        char *argv2[] = {"./worker", argv[1], &cc, start_str, end_str, NULL};
        execv("./worker", argv2);
         exit(1);
        }

        else {
            char start_str2[20];
            char end_str2[20];
             snprintf(start_str2, sizeof(start_str2), "%lld", (long long)start_remaining);
            snprintf(end_str2, sizeof(end_str), "%lld", (long long)end_remaining);
            char *argv2[] = {"./worker", argv[1], &cc, start_str, end_str, start_str2, end_str2, NULL};
            execv("./worker", argv2);
         exit(1);
        }

}//end of worker body here
```

Ο κάθε εργάτης καλεί execv (προκειμένου να τρέξει σε ένα ξεχωριστό αρχείο) και παίρνει ως ορίσματα το εκτελέσιμο αρχείο (./worker) καθώς και τον char *argv2[]. Στον πίνακα char *argv2 εισάγουμε το αρχείο καθώς και τον προς αναζήτηση χαρακτήρα αλλά και τα off_t start, end που δηλώνουν το τμήμα του αρχείου στο οποίο θα ψάξει ο εργάτης. Συνήθως, λοιπόν, η execv καλείται με 5 ορίσματα, ωστόσο, σε περίπτωση που έχει τερματιστεί βίαια ένας εργάτης πιο πριν τότε ο κώδικας εισέρχεται στο else όπου θα περάσει στον εργάτη και την αρχή και το τέλος του εκάστοτε worker προκειμένου να μελετηθεί και εκείνο το κομμάτι αρχείου. Έτσι, το πρόγραμμα μας είναι τελείως ανθεκτικό στον βίαιο τερματισμό κάποιου εργάτη.

Αξίζει να τονίσουμε ότι ο *argv2 με τον οποίο θα κληθεί το εκτελέσιμο ./worker δέχεται ορίσματα μόνο συμβολοσειρών. Για τον λόγο αυτό με τις εντολές

$$snprintf(start\_str, sizeof(start\_str), "\%lld", (long\ long)start);$$
$$snprintf(end\_str, sizeof(end\_str), "\%lld", (long\ long)end);$$

μετατρέπουμε τα off_t start, end (που ουσιαστικά αποτελούν long long μεταβλητές) σε συμβολοσειρές, τις οποίες μέσω της snprintf τοποθετούμε στα start_str, end_str αντιστοίχως. Τις συμβολοσειρές αυτές θα δούμε αργότερα πως θα τις επεξεργαστούμε στον worker.

Παράλληλα αξίζει να σημειωθεί ότι η εντολή:

$$dup2(pipefdread[current\_worker][1], \boldsymbol{STDOUT\_FILENO});$$

μεταφέρει την έξοδο του worker στον file_descriptor pipefdread[current_worker] ώστε να μπορεί αργότερα ο dispatcher να λάβει την συγκεκριμένη πληροφορία.
Κλείνοντας, τονίζουμε πάλι ότι υπό φυσιολογική ροή ο worker δεν θα επιστρέψει μετά την κλήση της execv.

```
        }//end of worker body here

        start = end;
        end += segment_size;
        remaining_workers--;
        current_worker++;

        sleep(2);

         if(command_available == 1){
            getcommand();
            command_available = 0;
        }

    }//This is were the while loop ends
```

Στο σημείο αυτό, βρισκόμαστε πάλι στην ροή εκτέλεσης του dispatcher που μεταβάλλει τις παραπάνω τιμές προκειμένου να καλέσει τον επόμενο εργάτη και να του ανατεθούν τα σωστά χωρία προς αναζήτηση.

Η ακόλουθη εντολή θα εξηγηθεί στην συνέχεια:

$$if(command\_available == 1)\{$$

$$getcommand();$$

$$command\_available = 0;$$

$$\}$$

```
    while(terminated_children < total_current_workers)
        sleep(2);

        if (write(STDOUT_FILENO, &count, sizeof(count)) == -1)
    {
        perror("write");
        exit(1);
    }
    kill(getppid(), SIGUSR1);
}
```

Όταν ο dispatcher τελειώσει με την δημιουργία εργατών μεταβαίνει σε νέα λούπα στην οποία περιμένει ώσπου να τερματίσουν όλοι οι εργάτες και έπειτα στέλνει την συνολική τιμή

μετρήματος πίσω στο front_end και αποστέλλει το σήμα SIGUSR1 στο front end δηλώνοντας ότι τερμάτισε κανονικά.

Αυτό γίνεται μέσω της εντολής: $kill\ (getppid(), SIGUSR1);$

Περνάμε πλέον σε όλα τα σήματα που έχουμε υλοποιήσει προκειμένου να λειτουργεί με σωστό τρόπο η παράλληλη επεξεργασία.

Ξεκινώντας με την $handle\_sigchild$ η οποία ενεργοποιείται κάθε φορά που στέλνεται το σήμα SIGCHILD (τερματίζει ένας worker)

```
void handle_sigchild(int sig) // This signal informs the dispatcher that a child finished it's work correctly
{
    int status;
    pid_t pid;
    int counter, i;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    {
        if (WIFEXITED(status))
        {
            // Find which worker has exited to read the correct pipe
    for (i = 0; i < most_workers; i++)
            {
    if (WORKERS[i].pid == pid)
{ //We use the array of workers to find which worker terminated

    close(pipefdread[i][1]);
    if (read(pipefdread[i][0], &counter, sizeof(counter)) == -1)
    {
     perror("read");
    exit(1);
    }
    count += counter;

    close(pipefdread[i][0]); // Close the read end after reading

    WORKERS[i].active = 0;// Mark the worker as inactive
    WORKERS[i].workerstart = -1;
    WORKERS[i].workerend = -1;
    WORKERS[i].pid = -1;

                    // WORKERS[i].pid = -1;
    terminated_children++;//This is for the last loop of the dispatcher
    break;
                }
            }
        }
    }
}
```

Πιο συγκεκριμένα, όποτε τερματιστεί μια διεργασία μέσω της εντολής

$pid\ =\ waitpid(-1, \&status, \boldsymbol{WNOHANG}))\ >\ 0$

ανιχνεύεται η διεργασία αυτή που τερμάτισε. Ο προσδιορισμός WHOHANG χρησιμεύει ώστε η διαδικασία αυτή να είναι non-blocking, να επιστρέφει, δηλαδή, αμέσως αν δεν υπάρχουν τερματισμένα παιδιά.

Η εντολή: $\boldsymbol{if}\ (\boldsymbol{WIFEXITED}(status))$

ελέγχει αν η διεργασία αυτή τερματίστηκε σωστά

Τονίζουμε επίσης, ότι σε αυτό το σήμα γίνονται όλες οι επιθυμητές ενέργειες. Δηλαδή, για την άμεση απόκτηση πληροφοριών κάθε φορά εντός της handle_sigchild αυξάνουμε την global μεταβλητή count (που εν τέλει αναπαριστά τις συνολικές φορές που βρέθηκε ο χαρακτήρας προς αναζήτηση στο συνολικό αρχείο) κατά counter (που αναπαριστά τις φορές που βρέθηκε ο χαρακτήρας στο segment του συγκεκριμένου εργάτη που τερματίστηκε). Παράλληλα, θέτουμε τον εργάτη σε inactive.

Εντός της κύριας συνάρτησης του dispatcher και προτού κληθεί το fork() έχουμε θέσει και τον χειρισμό του σήματος SIGCHILD

```
struct sigaction sa;

// Setting up SIGCHLD handler
memset(&sa, 0, sizeof(sa));
sa.sa_handler = handle_sigchild;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1)
{
    perror("sigaction SIGCHLD");
    exit(EXIT_FAILURE);
}
```

Προχωράμε στα commands που στέλνει το front end στον dispatcher για την αναζήτηση πληροφορίας. Όπως είδαμε προηγουμένως, όταν το front end στείλει σήμα SIGUSR1 στον dispatcher, τότε η volatile sig_atomic_t command_available λαμβάνει τιμή ίση με 1 από τον handler που αρχικά είδαμε.

Εντός του while loop όπου ο dispatcher μοιράζει το αρχείο στους workers εξετάζει (σε δύο σημεία) την ακόλουθη συνθήκη.

```
if(command_available == 1){
    getcommand();
    command_available = 0;
}
```

if(command_available == 1) τότε καλείται η get_command() η οποία διαβάζει την πληροφορία που ζήτησε το front end και καλεί την processCommand(). Στο τέλος της, η get_command(), στέλνει το σήμα SIGUSR2 στο front end, δηλώνοντας ότι έχει δώσει απάντηση στον πατέρα.

```c
void getcommand() {
    // Make stdin non-blocking
    int flags = fcntl(STDIN_FILENO, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl GETFL");
        exit(EXIT_FAILURE);
    }
    if (fcntl(STDIN_FILENO, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl SETFL");
        exit(EXIT_FAILURE);
    }

    char buffer[1024];
    int bytesRead = 0;
    while ((bytesRead = read(STDIN_FILENO, buffer, sizeof(buffer) - 1)) > 0) {
        buffer[bytesRead] = '\0'; // Null-terminate the string
        // Process the buffer content here. This simple example expects complete commands.
        processCommand(buffer);
    }
    if (bytesRead == -1 && errno != EAGAIN && errno != EWOULDBLOCK) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    // Signal to the parent process that processing is done.
    kill(getppid(), SIGUSR2);
}
```

Η processCommand(), έπειτα, διαβάζει την εντολή που έστειλε το front end και εκτελεί μια από τις ακόλουθες συναρτήσεις ανάλογα με το ζητούμενο που έλαβε το front end από τον χρήστη.

```c
void processCommand(char *signalInfo) {
    char command[50];
    int value = 0;

    sscanf(signalInfo, "%s %d", command, &value);

    if (strcmp(command, "increase") == 0) {
        increase_workers(value);
    } else if (strcmp(command, "decrease") == 0) {
        decrease_workers(value);
    } else if (strcmp(command, "terminate") == 0) {
        terminate_first_active_worker();
    } else if (strcmp(command, "progress") == 0) {
        give_file_percentage_and_count_till_now();
    } else if (strcmp(command, "info") == 0) {
        get_info();
    }
}
```

Ουσιαστικά στην get_command() λαμβάνεται η εντολή και αυτή στέλνεται στην process_Command() στην οποία γίνεται η αποκωδικοποίηση της εντολής αυτής μέσω της sscanf().

Η process_Command() καλεί οποιαδήποτε συνάρτηση με βάση το τι ζήτησε το front end και η συνάρτηση αυτή εκτελεί την λειτουργία της και στέλνει την απάντηση της στο front_end.

```
void get_info(){

    char message[256];
        int length = snprintf(message, sizeof(message), "All the available workers who have already completed their part or not are %d\n", total_current_
    if (length > 0 && length < sizeof(message)) {
        sendMessageWithWrite(message);
    }
    else {
        perror("snprintf");
        exit(1);
    }
    char message2[256];
    length = snprintf(message2, sizeof(message), "The remaining workers are %d\n", remaining_workers);
    if (length > 0 && length < sizeof(message)) {
        sendMessageWithWrite(message2);
    }
        else {
        perror("snprintf");
        exit(1);
    }
}

void give_file_percentage_and_count_till_now(){
    double file_percentage;
    long long examined;

    for (int i = 0; i <= current_worker; i++)
        if (WORKERS[i].active == 0)
        examined += (WORKERS[i].workerend -WORKERS[i].workerstart);

    file_percentage = examined * 100 / file_size;


    char message[100];
    int message_length = snprintf(message, sizeof(message), "the percentage of the file examined up till now is %f\n", file_percentage);
    if (message_length < 0 || message_length >= sizeof(message)) {
        perror("snprintf");
        exit(1);
    }


    ssize_t bytes_written = write(STDOUT_FILENO, message, message_length);
    if (bytes_written == -1) {
        perror("write");
        exit(1); // Exit or handle the error accordingly
    }

    // Prepare the message for total_count_till_now
    message_length = snprintf(message, sizeof(message), "The total_count_till now is %d\n", count);
    if (message_length < 0 || message_length >= sizeof(message)) {
        perror("snprintf");
        exit(1); // Handle or exit on error
    }

    // Write the message to STDOUT_FILENO
    bytes_written = write(STDOUT_FILENO, message, message_length);
    if (bytes_written == -1) {
        perror("write");
        exit(1); // Exit or handle the error accordingly
```

```c
void increase_workers(int value){
    char message[256];
    if (total_current_workers + value < most_workers){
        total_current_workers += value;
        remaining_workers += value;
        segment_size = (file_size - end)/remaining_workers;
        int length = snprintf(message, sizeof(message), "Workers increased by %d\n", value);
    if (length > 0 && length < sizeof(message)) {
        sendMessageWithWrite(message);
    }
    else {
        perror("snprintf");
        exit(1);
    }
    }
    else {
         int length = snprintf(message, sizeof(message),"Couldn't increase because I cant have so many wokrers on my project\n");
    if (length > 0 && length < sizeof(message)) {
        sendMessageWithWrite(message);
    }
    else {
        perror("snprintf");
        exit(1);
    }
}
}

void decrease_workers(int value){
    char message[256];
    if (remaining_workers - value >= 1){
        total_current_workers -= value;
        remaining_workers -= value;
        segment_size = (file_size - end)/remaining_workers;
        int length = snprintf(message, sizeof(message), "Workers decreased by %d\n", value);
    if (length > 0 && length < sizeof(message)) {
        sendMessageWithWrite(message);
    }
    else {
        perror("snprintf");
        exit(1);
    }
    }
    else{
        int length = snprintf(message, sizeof(message),"Couldn't decrease because no one will be left on my project\n");
    if (length > 0 && length < sizeof(message)) {
        sendMessageWithWrite(message);
    }
    else {
        perror("snprintf");
        exit(1);
    }
    }
}
```

```c
void terminate_first_active_worker()
{

        if (remaining_workers < 1){
            char message[] = "You should have told me to kill the worker earlier\n";
             ssize_t bytes_written = write(STDOUT_FILENO, message, sizeof(message) - 1);
            if (bytes_written == -1) {
                perror("write");
                exit(1); // Exit or handle the error accordingly
            }
            return;
        }
    for (int i = 0; i <= current_worker; i++)
    {
        if (WORKERS[i].active == 1)
        { // If active worker found, we kill him and save its data

            kill(WORKERS[i].pid, SIGKILL);

            char message[] = "The first worker I found was killed\n";

            // Write the message to STDOUT_FILENO
            ssize_t bytes_written = write(STDOUT_FILENO, message, sizeof(message) - 1);
            if (bytes_written == -1) {
                perror("write");
                exit(1); // Exit or handle the error accordingly
            }


            // We set the variables of the deactivated worker to 'zero'
            WORKERS[i].pid = -1;
            WORKERS[i].active = -2;//The worker died by command and it's start and end must be restored
            break;
        }
    }
    total_current_workers--;
    terminated_children++;
}
```

Πρόγραμμα για τους workers:

Η υλοποίηση του συγκεκριμένου κώδικα ξεκινάει όταν ο dispatcher καλέσει $execv$ μέσα στην $fork()$ για το παιδί. Τότε, η εκτέλεση περνάει στην υλοποίηση του νέου εκτελέσιμου προγράμματος, με παραμέτρους που καθορίζονται από τις παραμέτρους της συνάρτησης $execv$ που το κάλεσε. Συγκεκριμένα, κάνουμε $execv("./worker", argv2)$ με τον πίνακα argv2 να περιέχει τις κατάλληλες παραμέτρους για την έγκυρη εκτέλεση του αρχείου worker. Όντας, πλέον, εντός του προγράμματος για τους εργάτες, βλέπουμε την συνάρτηση $int\ char\_counter(){}$ υπεύθυνη για την σωστή καταμέτρηση του χαρακτήρα-στόχου για την οποία έχουμε ήδη αναφερθεί στην άσκηση 3 της παρούσας αναφοράς. Συνεχίζοντας, μπαίνουμε στην συνάρτηση $int\ main(){}$, η οποία δέχεται είτε πέντε είτε επτά ορίσματα ανάλογα του πως έχει κληθεί, το οποίο επηρεάζεται από το τι έχει επιστρέψει η συνάρτηση $check\_remaining\_file\_search()$, αν δηλαδή έχουμε χρωστούμενο χωρίο προς αναζήτηση από κάποιον εργάτη που του επιβλήθηκε terminate λόγω της command '$terminate$'.

Ανάλογα, λοιπόν, με το πως έχει γίνει η κλήση της $execv$ χρησιμοποιούμε τις κατάλληλες παραμέτρους ώστε να καλέσουμε την συνάρτηση $int\ char\_counter(){}$ και να συλλέξουμε την πληροφορία του εκάστοτε εργάτη που αφορά τον αριθμό εμφάνισης του χαρακτήρα-στόχου στο συγκεκριμένο χωρίο που του έχει ανατεθεί. Αν η συνάρτηση $check\_remaining\_file\_search()$ δεν έχει επιστρέψει την τιμή -1, σημαίνει ότι του εργάτη j (με $j \neq -1\ και\ j\ \epsilon\ (0, total\_current\_workers \pm x), x\ increased\ or\ decreased\ by\ user$ του επιβλήθηκε τερματισμός πριν προλάβει να αναζητήσει όλο του το χωρίο, και άρα πρέπει να αναζητηθεί και το χρωστούμενο αυτό χωρίο. Για τον λόγο αυτό γίνεται ξανά κλήση της συνάρτησης $int\ char\_counter(){}$ βάζοντας το χρωστούμενο χωρίο για αναζήτηση. Τα δυο αποτελέσματα που επέστρεψαν οι δυο αυτές κλήσεις της συνάρτησης $int\ char\_counter(){}$ προστίθενται ώστε να πάρουμε τον συνολικό αριθμό εμφάνισης του χαρακτήρα-στόχου χάρη στον εργάτη που δούλεψε τώρα.

Αυτό που μένει να κάνουμε είναι να στείλουμε πίσω στον dispatcher (διεργασία γονέα του worker) το αποτέλεσμα που αποθηκεύσαμε στην μεταβλητή $counter$. Αυτό επιτυγχάνεται χάρη στην σωλήνωση που έχει δημιουργηθεί στο πρόγραμμα του dispatcher και αφορά την επικοινωνία μεταξύ dispatcher και του εκάστοτε worker (βλέπε $dup2()$ στον dispatcher). Άρα, αυτό που μένει να κάνει ο worker είναι το ακόλουθο:

```
if (write(STDOUT_FILENO, &counter, sizeof(counter)) == -1)
{
    perror("write");
    exit(1);
}
exit(0);
```

Με αυτό στέλνει την τιμή της μεταβλητής counter πίσω στον dispatcher ελέγχοντας παράλληλα για πιθανό error.

Τέλος, κρίνουμε σημαντικό να αναφερθεί η χρησιμότητα της εντολής $atoll$. Αυτό που κάνει είναι να μετατρέπει ένα $string$, στην περίπτωσή μας τα $argv[i]$ $με$ $i = 3, 4, 5, 6$, σε $long\ long$ αφού τα θέλουμε να είναι off_t. Ειδάλλως θα παίρναμε το ακόλουθο warning:

$$initialization\ of\ 'off\_t'\ \{aka\ 'long\ int'\}\ from\ 'char\ *'\ makes\ integer\ from\ pointer\ without\ a\ cast$$

Σχόλιο:

Οφείλουμε να αναφέρουμε ότι η διαδικασία που έχουμε υλοποιήσει για την Εφαρμογή της Παράλληλης Καταμέτρησης Χαρακτήρων δεν ανταποκρίνεται με τον πιο ακριβή τρόπο στο ζητούμενο. Πιο συγκεκριμένα, θα έπρεπε ιδανικά το κομμάτι αναζήτησης (segment_size) που αναθέτουμε σε κάθε εργάτη να ανατίθεται με δυναμικό τρόπο και κυκλικά. Όποτε ένας εργάτης, δηλαδή, τελείωνε την αναζήτηση του χαρακτήρα-στόχου στο κομμάτι του, τότε να μπορεί να του ανατεθεί εκ νέου ένα ακόμα κομμάτι αρχείου προς αναζήτηση αφότου φυσικά έχουν ανατεθεί σε κάθε διαθέσιμο εργάτη τα κομμάτια αρχείου που μεσολαβούν. Αν είχαμε, δηλαδή, 10 εργάτες αρχικά, τότε αφότου ανατεθούν κομμάτια και στους 10 και λήξει κτην αναζήτησή του κάποιος, τότε να μην τελείωνε, να μην γινόταν μη-διαθέσιμος (όπως υλοποιήσαμε εμείς), αλλά να του ανέθετε ο dispatcher το επόμενο διαθέσιμο προς αναζήτηση κομμάτι. Ο χρήστης θα μπορεί και σε αυτό το σενάριο να δίνει τις εντολές που εξετάσαμε παραπάνω.

```
 GNU nano 5.4                                                    worker.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>


int char_counter(char *file, char c_check, off_t start_pos, off_t end_pos)
{ // for character count
    char c2c = 'a', cc;
    int count = 0;

    int fdr = open(file, O_RDONLY);
    if (fdr == -1)
    {
        perror("open");
        exit(1);
    }

    if (lseek(fdr, start_pos, SEEK_SET) == -1)
    {
        perror("lseek from worker");
        exit(1);
    }

    ssize_t rcnt;
    c2c = c_check;

    while ((rcnt = read(fdr, &cc, sizeof(cc))) > 0)
    {
        if (cc == c2c)
            count++;
        if (lseek(fdr, 0, SEEK_CUR) >= end_pos)
```

```
        if (lseek(fdr, 0, SEEK_CUR) >= end_pos)
            break; // stop reading if reached the end position
    }

    if (rcnt == -1)
    {
        perror("read");
        exit(1);
    }
    close(fdr);
    return count;
}


int main (int argc, char **argv){

int counter2 = 0;
char cc = argv[2][0];
off_t start = atoll(argv[3]);
off_t end = atoll(argv[4]);
int counter = char_counter(argv[1], cc, start, end);
if (argc > 5){
    off_t start_remaining = atoll(argv[5]);
    off_t end_remaining = atoll(argv[6]);
    counter2 = char_counter(argv[1], cc, start_remaining, end_remaining);
}
//printf("%d", counter);
counter += counter2;
    sleep(1);

 if (write(STDOUT_FILENO, &counter, sizeof(counter)) == -1)
    {
        perror("write");
        exit(1);
    }
```

```
  //printf("%d", counter);
  counter += counter2;
      sleep(1);

   if (write(STDOUT_FILENO, &counter, sizeof(counter)) == -1)
      {
          perror("write");
          exit(1);
      }
          sleep(2);
  exit(0);
  }
```

Τέλος, παραθέτουμε ενδεικτικά ορισμένες εκτελέσεις του κώδικά μας.

Εκτελούμε $./front\_end\ my\_name.txt\ e$

i. ‘increase 1’ και μετά ‘info’

```
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed
increase 1
sending signal increase 1
signal received back
Workers increased by 1


Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed
info
sending signal info
signal received back
All the available workers who have already completed their part or not are 11
The remaining workers are 7


Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed

The dispatcher has ended
Total_count is 8
```

ii. ‘terminate’ και μετά ‘info’

```
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed
terminate
sending signal terminate
signal received back
The first worker I found was killed


Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed
info
sending signal info
signal received back
All the available workers who have already completed their part or not are 9
The remaining workers are 6


Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed

The dispatcher has ended
Total_count is 8
```

iii. ‘progress’

```
oslab044@os-node1:~/first_ex/ex_4$ ./front_end my_name.txt e
Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed
progress
sending signal progress
signal received back
the percentage of the file examined up till now is 9.000000
The total_count_till now is 1


Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed
progress
sending signal progress
signal received back
the percentage of the file examined up till now is 36.000000
The total_count_till now is 4
```

```
Enter a command

The accessible commands are:
1) 'increase x (enter)': adding x workers
2) 'decrease x (enter)': killing x workers
3) 'terminate (enter)': killing the first active worker
4) 'progress (enter)': for file percentage and count of character till now
5) 'info (enter)': for info
6) 'exit (enter)': to proceed

The dispatcher has ended
Total_count is 8
```