



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

Σχολή Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών

ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ 1

Μέλη Ομάδας

ΚΑΒΑΔΑΚΗΣ ΣΩΤΗΡΙΟΣ 03122035

ΚΥΖΙΡΙΔΗΣ ΧΡΗΣΤΟΣ 03122431

Περιεχόμενα

1	Ανάγνωση και εγγραφή αρχείων στη C και με τη βοήθεια κλήσεων συστήματος	1
2	Δημιουργία διεργασιών	1
2.1	1
2.2	2
2.3	3
2.4	4
3	Διαδιεργασιακή επικοινωνία	5
4	Εφαρμογή παράλληλης καταμέτρησης χαρακτήρων	8
4.1	Frontend	9
4.2	Dispatcher	10
4.3	Worker	14
4.4	Παρατηρήσεις-Βελτιώσεις	16

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

Το πρόγραμμα που υλοποιήσαμε για το ερώτημα αυτό ικανοποιεί το ζητούμενο της εκφώνησης, δηλαδή να κάνουμε ανάγνωση και εγγραφή αρχείων στη C κάνοντας αποκλειστική χρήση system calls, όπως open, read, write και close, χωρίς τη χρήση της βιβλιοθήκης της C (fopen, fprintf, κ.λπ.). Ο χρήστης δίνει στη γραμμή εντολών τρία ορίσματα, δηλαδή το αρχείο εισόδου, το αρχείο εξόδου και τον χαρακτήρα προς αναζήτηση. Το πρόγραμμα ανοίγει το αρχείο εισόδου για ανάγνωση (O_RDONLY) και το αρχείο εξόδου για εγγραφή (O_WRONLY | O_TRUNC), εξασφαλίζοντας ότι το αρχείο εξόδου θα είναι κενό πριν από την εγγραφή. Έπειτα, διαβάζει επαναληπτικά το αρχείο εισόδου σε έναν buffer μεγέθους 1024 bytes, ελέγχοντας κάθε χαρακτήρα ξεχωριστά και μετρώντας πόσες φορές εμφανίζεται ο target χαρακτήρας. Όταν ολοκληρωθεί η ανάγνωση του αρχείου, δημιουργεί ένα μήνυμα με τη συνολική καταμέτρηση μέσω της snprintf (safe version της sprintf που προλαμβάνει buffer overflow) και το εγγράφει στο αρχείο εξόδου μέσω της write. Τέλος, τα αρχεία κλείνονται κανονικά με close. Επιπλέον, σε κάθε κρίσιμη κλήση συστήματος (open, read, write) γίνεται έλεγχος για σφάλματα και, αν προκύψει κάποιο πρόβλημα, εμφανίζεται κατάλληλο μήνυμα σφάλματος με χρήση της perror, και το πρόγραμμα τερματίζεται με exit(1).

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

int main(int argc, char *argv[]) {
    char buff[1024];
    int count = 0;
    int fdr, fdw;
    size_t len;
    ssize_t rcnt, wcnt;
    fdr = open(argv[1], O_RDONLY); //returns file to be read descriptor
    if (fdr == -1) {
        perror("Problem opening file to read\n");
        exit(1);
    }
    fdw = open(argv[2], O_WRONLY | O_TRUNC);
    if (fdw == -1) {
        perror("Problem opening file to write\n");
        exit(1);
    }

    for (;;) {
        rcnt = read(fdr, buff, sizeof(buff) - 1); //store the data to the buffer
        if (rcnt == 0) //EOF
            break;
        if (rcnt == -1) {
            perror("read");
            exit(1);
        }
        buff[rcnt] = '\0'; //make the last character null terminator
        len = strlen(buff);
        int i = 0;
        while(i < len) {
            if(buff[i++] == argv[3][0]) {
                count++;
            }
        }
    }

    char output[256];
    snprintf(output, sizeof(output), "The character '%c' appears %d times in file %s.\n", argv[3][0], count, argv[1]);
    write(fdw, output, strlen(output));
    close(fdr);
    close(fdw);
    return 0;
}
```

Εικόνα 1.1: Κώδικας για το ερώτημα 1

2 Δημιουργία διεργασιών

2.1

Στο συγκεκριμένο κομμάτι διαχειριζόμαστε το τρόπο με τον οποίο δημιουργούμε διεργασίες παιδιά, καθώς και να πειραματιστούμε με τα αναγνωριστικά της εκάστοτε διεργασίας, γονέα και παιδιού. Πιο συγκεκριμένα μέσω της συνάρτησης fork() της C, την οποία τη καλεί η κύρια διεργασία (γονέας), δημιουργείται ένα αντίγραφο της.

Η `fork()` επιστρέφει στο αντίγραφο της μεταβλητής `p` που ανήκει στη διεργασία παιδί τη τιμή 0, ενώ στη `p` της κύριας διεργασίας αναθέτει το αναγνωριστικό (`pid`) του παιδιού. Η μεταβλητή `mypid` είναι για να κρατάει το αναγνωριστικό της εκάστοτε διεργασίας που τρέχει. Σε περίπτωση αποτυχίας της `fork()` εμφανίζεται αντίστοιχο μήνυμα στην οθόνη και τερματίζεται η διεργασία. Όσον αφορά το κώδικα που εκτελεί η διεργασία παιδί, που βρίσκεται μέσα στο `if (p == 0)` statement, τυπώνει “hello world” και έπειτα το δικό του `id` καθώς και το `id` του γονιού του μέσω της συνάρτησης `getppid()`, και τέλος κάνει `exit(0)` που σημαίνει ότι τερμάτισε με επιτυχία. Ο κώδικας που εκτελεί η διεργασία γονέας, η οποία βρίσκεται μέσα στο `if (p > 0)` statement, τυπώνει το αναγνωριστικό του παιδιού του, που έχει αποθηκευτεί στη μεταβλητή `p`, και έπειτα περιμένει το τερματισμό της διεργασίας παιδιού με τη `wait()` για να μην προκύψει διεργασία zombie.

2.2

Κάνουμε τις εξής αλλαγές πάνω στο 1ο ερώτημα της άσκησης: Δηλώνουμε ως global ακέραια μεταβλητή `x` και την βάζουμε να είναι ίση με 1, την τιμή της οποίας αλλάζουμε σε 10 στη διεργασία παιδί, και έπειτα από αυτό τυπώνουμε τη τιμή της στη διεργασία παιδί και στη κύρια διεργασία. Παρατηρούμε πως τυπώνεται η τιμή `x = 1` για το parent process και η τιμή 10 για το child process, το οποίο είναι αναμενόμενο καθώς με τη κλήση της `fork()` οι δύο διεργασίες δε μοιράζονται τις ίδιες μεταβλητές, αλλά δημιουργούνται αντίγραφα τους στην περίπτωση της εγγραφής (copy on write), ώστε να εξοικονομηθεί αποθηκευτικός χώρος.

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

int x = 1;

int main(int argc, char **argv){
    pid_t p, mypid;
    mypid = -1;
    p = fork();
    if (p < 0){
        perror("fork");
        exit(1);
    }
    else if (p == 0){ //we are inside the child process
        printf("hello world\n");
        mypid = getpid();
        printf("Child id is: %d\n", mypid);
        pid_t parent = getppid();
        printf("Parent id is: %d\n", parent);
        x = 10;
        printf("x for Child is: %d\n", x);
        exit(0);
    }
    else {
        printf("Child id from parent is: %d\n", p);
        printf("x for Parent is: %d\n", x);
        pid_t status;
        wait(&status); //parent waits for the child to finish: otherwise we might end up with a ZOMBIE process
        printf("Child died\n");
    }
    return 0;
}
```

Εικόνα 2.2.1: Κώδικας για τα ερωτήματα 2.1 & 2.2

```
oslab004@os-node1:~/lab1$ ./ex2_1
Child id from parent is: 1853078
x for Parent is: 1
hello world
Children id is: 1853078
Parent id is: 1853077
x for Child is: 10
Child died
```

Εικόνα 2.2.2: Έξοδος του προγράμματος για τα ερωτήματα 2.1 & 2.2

2.3

Στον παρακάτω κώδικα επεκτείνουμε το πρόγραμμα του 1ου ερωτήματος που πραγματοποιεί την ανάγνωση ενός αρχείου κειμένου και μετράει τις εμφανίσεις ενός συγκεκριμένου χαρακτήρα, που δίνεται ως όρισμα στη γραμμή εντολών. Για κάθε μπλοκ δεδομένων (μέγιστου μεγέθους 1024 bytes, δηλαδή 1024 χαρακτήρων) που διαβάζεται από το αρχείο, δημιουργείται μια νέα διεργασία παιδί με τη χρήση της `fork()`, και ο υπολογισμός του πλήθους εμφανίσεων του χαρακτήρα ανατίθεται στη διεργασία αυτή. Η επικοινωνία μεταξύ διεργασίας γονέα και παιδιού επιτυγχάνεται με τη χρήση `pipe`, μέσω του οποίου το παιδί στέλνει το αποτέλεσμα στον γονέα. Πιο αναλυτικά, η κύρια διεργασία αρχικά ανοίγει δύο αρχεία: το πρώτο σε λειτουργία ανάγνωσης (`O_RDONLY`), από όπου διαβάζονται τα δεδομένα, και το δεύτερο σε λειτουργία εγγραφής (`O_WRONLY` — `O_TRUNC`), όπου πρώτα σβήνει οτιδήποτε προϋπήρχε στο αρχείο εγγραφής, και τέλος γράφεται το τελικό αποτέλεσμα. Σε έναν infinite loop (`for(;;)`), το πρόγραμμα διαβάζει μπλοκ δεδομένων από το αρχείο εισόδου, και για κάθε μπλοκ δημιουργεί νέο `pipe` και νέο `fork()`. Η μεταβλητή `current_processes` είναι ένας μετρητής για να εκτυπώνεται στην έξοδο τον αριθμό του ζευγαριού διεργασιών γονέα-παιδιού.

Στη διεργασία παιδί, η οποία τρέχει αν `p == 0`, αρχικά κλείνει το read-end του `pipe` (που δεν χρησιμοποιείται από το παιδί), χαιρετάει τον κόσμο, δίνοντας μαζί το αναγνωριστικό του και του γονιού του, και καλείται η συνάρτηση `child()` η οποία μετράει τις εμφανίσεις του χαρακτήρα στο δεδομένο `buffer`. Το αποτέλεσμα στέλνεται στον γονέα μέσω της εγγραφής στο `pipe`, του οποίου το write end κλείνει σε αυτό το σημείο και το παιδί τερματίζει με `exit(0)`.

Ο γονέας επίσης χαιρετάει τον κόσμο δίνοντας το αναγνωριστικό του και κλείνει το άχρηστο write end του `pipe`, διαβάζει το αποτέλεσμα από το `pipe`, το προσθέτει στο συνολικό μετρητή `count`, και περιμένει την ολοκλήρωση της διεργασίας παιδιού με `wait()`.

Τέλος, το πρόγραμμα κάνει `format` σε string το τελικό αποτέλεσμα μέσω της `snprintf()` (safe version της `sprintf()`) και γράφει το συνολικό πλήθος εμφανίσεων στο αρχείο εξόδου και κλείνει τους file descriptors. Το πρόγραμμα χρησιμοποιεί συστηματικά διαχείριση σφαλμάτων (π.χ. με `perror`) για περιπτώσεις όπου η `read`, `write`, `open`, `pipe` ή `fork` αποτυγχάνουν. Ελέγχοντας το `result.txt`, ορθά γράφεται ο αριθμός εμφανίσεων του target χαρακτήρα, όπως άλλωστε διαπιστώθηκε και στην εργαστηριακή εξέταση.

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

int current_process = 0; //for long text files multiple forks may be needed

int child(char *buffer, char *search){
    int count=0, i=0;
    while (buffer[i] != '\0'){
        if (buffer[i] == *search)
            count++;
        i++;
    }
    return count;
}

int main(int argc, char **argv){
    char buff[1024];
    int fdr, fdw, rcnt, wcnt;
    int count=0;
    fdr=open(argv[1], O_RDONLY);
    if (fdr == -1) {
        perror("open");
        exit(1);
    }
    fdw=open(argv[2], O_WRONLY | O_TRUNC);
    if (fdw == -1) {
        perror("open");
        exit(1);
    }
}
```

Εικόνα 2.3.1: Κώδικας της συνάρτησης `child()` και άνοιγμα αρχείων

```

for (;;) {
    int result=0;
    rcnt=read(fdr, buff, sizeof(buff)-1);
    if (rcnt == 0)
        break;
    if (rcnt == -1) {
        perror("read");
        return 1;
    }
    buff[rcnt] = '\0';
    pid_t p, mypid;
    int pfd[2];
    if (pipe(pfd) < 0) {
        perror("pipe");
        exit(1);
    }
    mypid = -1;
    current_process++;
    p = fork();
    if (p < 0) {
        perror("fork");
        exit(1);
    }
    else if (p == 0) {
        close(pfd[0]);
        mypid = getpid();
        pid_t parent = getppid();
        printf("Hello World from the child process #! My ID is: %d, and my parent's ID is: %d\n", current_process, mypid, parent);
        result=child(buff, &argv[3][0]);
        if (write(pfd[1], &result, sizeof(result)) != sizeof(result)) {
            perror("write failed");
        }
        close(pfd[1]);
        exit(0);
    }
    else {
        close(pfd[1]);
        int child_count;
        if (read(pfd[0], &child_count, sizeof(child_count)) != sizeof(child_count)) {
            perror("read from pipe");
        }
        printf("Hello World from the parent process #! My child's ID is: %d\n", current_process, p);
        count+=child_count;
        pid_t status;
        wait(&status);
    }
}

```

Εικόνα 2.3.2: Κώδικας της infinite loop που αναλαμβάνει τη δημιουργία νέων διεργασιών

```

int length = snprintf(buff, sizeof(buff), "%d", count);
wcnt=write(fdw, buff, length);
if (wcnt == -1) {
    perror("write");
    return 1;
}
close(fdr);
close(fdw);
return 0;

```

Εικόνα 2.3.3: Εγγραφή στο αρχείο εξόδου, κλείσιμο αρχείων και έξοδος από τη κύρια διεργασία

```

[oslab004@os-node1:~/lab1$ ./ex2_christos source.txt result.txt s
Hello World from the child process #1! My ID is: 1854334, and my parent's ID is: 1854333
Hello World from the parent process #1! My child's ID is: 1854334
Hello World from the child process #2! My ID is: 1854335, and my parent's ID is: 1854333
Hello World from the parent process #2! My child's ID is: 1854335

```

Εικόνα 2.3.4: Έξοδος του προγράμματος για αρχείο εισόδου μήκους μεταξύ 1024 και 2048 bytes

2.4

Στον συγκεκριμένο κώδικα, χρησιμοποιείται η συνάρτηση `fork()` για τη δημιουργία μίας νέας διεργασίας παιδί, η οποία στη συνέχεια αντικαθιστά τον εαυτό της με την εκτέλεση ενός διαφορετικού εκτελέσιμου αρχείου μέσω της `execv()`. Πιο συγκεκριμένα, η διεργασία γονέας δημιουργεί ένα παιδί, και αν η `fork()` αποτύχει, εμφανίζεται μήνυμα σφάλματος και το πρόγραμμα τερματίζεται. Στη διεργασία παιδί (δηλαδή όταν `pid == 0`), καλείται η `execv()` για να εκτελέσει το αρχείο που εκτελεί τα ζητούμενα της 1ης άσκησης του οποίου το path

είναι `/home/oslab/oslab004/lab1/ex1_kavadas`. Η `execv()` καλείται μαζί με παραμέτρους τα `argv[1]`, `argv[2]`, `argv[3]`, δηλαδή τα ορίσματα που δίνονται στο αρχικό πρόγραμμα κατά την εκκίνηση. Σε περίπτωση αποτυχίας της `execv()`, εκτυπώνεται κατάλληλο μήνυμα σφάλματος. Η διεργασία γονέας καλεί `wait()` για να περιμένει τον τερματισμό του παιδιού προτού τερματίσει και η ίδια. Ουσιαστικά, αυτός ο κώδικας λειτουργεί ως «ενδιάμεσος εκκινητής» για άλλο πρόγραμμα, μεταβιβάζοντας του ορίσματα και εξασφαλίζοντας ότι ο γονέας θα περιμένει την ολοκλήρωσή του.

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

int main(int argc, char **argv){
    pid_t pid = fork();
    if (pid < 0 ) {
        perror("fork");
        exit(1);
    }
    else if (pid == 0){
        char *arg_child[] = {"./home/oslab/oslab004/lab1/ex1_kavadas", argv[1], argv[2], argv[3], NULL};
        if (execv(arg_child[0], arg_child) < 0){
            perror("execv");
            exit(1);
        }
    }
    else {
        int status;
        wait(&status);
    }
    return 0;
}
```

Εικόνα 2.4.1: Κώδικας για το ερώτημα 2.4

3 Διαδιεργασιακή επικοινωνία

Ο παρακάτω κώδικας υλοποιεί ένα παράλληλο πρόγραμμα μέτρησης της συχνότητας εμφάνισης ενός χαρακτήρα μέσα σε ένα αρχείο, αξιοποιώντας 8 διεργασίες (ορίζεται μέσω του `#define p 8`). Πρώτα ορίζουμε globally τη συνάρτηση `sigint_handler` που διαχειρίζεται σήματα που στέλνονται από τον χρήστη ασύγχρονα με τη λειτουργία του προγράμματος, πατώντας `CTRL + C`. Εν γένει το `SIGINT` σήμα τερματίζει τη τρέχουσα διεργασία, όμως εμείς το έχουμε ρυθμίσει ώστε να τυπώνει τη τιμή της `active_processes`, που μετράει το πλήθος των διεργασιών που έχουμε ενεργοποιήσει. Βρίσκουμε το file size σε bytes μέσω της εντολής `lseek(fdr, 0, SEEK_END)`. Έπειτα ξαναχρησιμοποιούμε το `command` αυτό για να μετακινήσουμε το δείκτη στην αρχή του κειμένου. Το αρχείο που δίνεται ως είσοδος διαβάζεται από τον γονέα και τεμαχίζεται σε `p` ίσα chunks (το τελευταίο chunk περιλαμβάνει και το υπόλοιπο). Στη συνέχεια δημιουργούνται `p` ζεύγη από pipes για τη μεταφορά δεδομένων (από γονέα σε παιδί) και αποτελεσμάτων (από παιδί σε γονέα). Ο γονέας δημιουργεί `p` διεργασίες-παιδιά μέσω της `fork()`, με χρονοκαθυστέρηση ενός δευτερολέπτου μέσω της `sleep(1)` για καλύτερη εποπτεία του πως τρέχει ο κώδικας μας μεταξύ κάθε iteration της for loop. Έπειτα κρατάει το `pid` του κάθε παιδιού που δημιουργεί και περιμένει τη κάθε διεργασία να ολοκληρωθεί μέσω της `waitpid`. Όταν κάθε διεργασία-παιδί ολοκληρωθεί και κάνει `exit` με επιτυχία, τότε μαζεύει τα partial results από το pipe κάθε παιδιού και έπειτα τα αθροίζει ώστε να λάβει το τελικό αποτέλεσμα, το οποίο το γράφει και στο output file. Κάθε παιδί κλείνει τα αχρείατα για εκείνο ends των pipes, δημιουργεί το κατάλληλο για εκείνο `chunk.buffer` δυναμικά, λαμβάνει μέσω pipe ένα chunk του αρχείου από τον γονέα μετράει το πλήθος εμφανίσεων του χαρακτήρα που δίνεται ως όρισμα (`argv[3]`) και επιστρέφει το αποτέλεσμα στον γονέα μέσω του αντίστοιχου pipe. Όλες οι διεργασίες-παιδιά αγνοούν το σήμα `SIGINT` (`Ctrl+C`). Η χρήση δυναμικής μνήμης (`malloc`) εξασφαλίζει ότι κάθε διεργασία έχει τον δικό της δυναμικό χώρο για το chunk στο οποίο φάχνει, και στο τέλος γίνεται καθαρισμός (`free & close`). Έτσι, το πρόγραμμα επιτυγχάνει παράλληλη επεξεργασία μεγάλων αρχείων με ασφάλεια και διαχείριση ασύγχρονων σημάτων.


```

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

#define p 8

int active_processes = 0;

void sigint_handler (int sig) { //typically ctrl-c signal
    printf("\nTotal number of active processes: %d\n", active_processes);
    sleep(1);
}

int main (int argc, char *argv[]) {
    int fdr, fdw;
    char buff[1024];
    ssize_t rcnt, wcnt;
    int result=0;

    signal(SIGINT, sigint_handler); //declare the signal handler

    fdr = open(argv[1], O_RDONLY);
    if (fdr == -1) {
        perror("Problem opening file to read\n");
        exit(1);
    }
    fdw = open(argv[2], O_WRONLY | O_TRUNC);
    if (fdw == -1) {
        perror("Problem opening file to write\n");
        exit(1);
    }

    off_t file_size = lseek(fdr, 0, SEEK_END);
    lseek(fdr, 0, SEEK_SET); //set pointer to start of fdr
    off_t chunk_size = file_size / p;
    off_t remainder = file_size % p;

    int data_pfd[p][2]; //for parent to write the file for child to read
    int result_pfd[p][2]; //for child to write result to parent
    pid_t child_pids[p]; //static declaration of pid array for the children

```

Εικόνα 3.1: Κώδικας που δηλώνει τον sigint_handler και ανοίγει αρχεία & pipes


```

for (int i = 0; i < p; i++) {
    if (pipe(data_pfd[i]) < 0) {
        perror("data pipe creation error");
        exit(1);
    }

    if (pipe(result_pfd[i]) < 0) {
        perror("result pipe creation error");
        exit(1);
    }
}

for (int i = 0; i < p; i++) { //for all child processes
    sleep(1); //delay 1 sec
    pid_t child_pid = fork(); //create the child process i
    if (child_pid < 0) {
        perror("fork error");
        exit(1);
    }
    else if (child_pid == 0) { //child_process i
        signal(SIGINT, SIG_IGN); //children ignore the signal handler
        for (int j = 0; j < p; j++) { //close all unused pipe ends
            if (j != i) {
                close(data_pfd[j][0]);
                close(data_pfd[j][1]);
                close(result_pfd[j][0]);
                close(result_pfd[j][1]);
            } else {
                close(data_pfd[j][1]);
                close(result_pfd[j][0]);
            }
        }
        off_t current_chunk_size = (i == p - 1) ? (chunk_size + remainder) : chunk_size;
        char *chunk_buffer = malloc(current_chunk_size); //dynamic declaration of current chunk buffer
        if (chunk_buffer == NULL) {
            perror("malloc error");
            exit(1);
        }

        rcnt = read(data_pfd[i][0], chunk_buffer, current_chunk_size); //read from parent
        if (rcnt < 0) {
            perror("read from pipe error");
            exit(1);
        }
        int count = 0; //count occurrences of character
        for (int j = 0; j < rcnt; j++) {
            if (chunk_buffer[j] == argv[3][0]) {
                count++;
            }
        }

        write(result_pfd[i][1], &count, sizeof(count)); //write result back to parent

        free(chunk_buffer); //start cleaning up
        close(data_pfd[i][0]);
        close(result_pfd[i][1]);
        exit(0);
    }
}

```

Εικόνα 3.2: Κώδικας που ελέγχει το άνοιγμα των pipes και εκτελεί το κάθε παιδί

```

    else { //parent process
        child_pids[i] = child_pid;
        active_processes++;
    }
}

for (int i = 0; i < p; i++) { //parent process so close any unneeded ends
    close(data_pfd[i][0]);
    close(result_pfd[i][1]);
}

char *buffer = malloc(chunk_size + remainder);
if (buffer == NULL) {
    perror("malloc error");
    exit(1);
}

for (int i = 0; i < p; i++) { //read data and pass it to data_pipes
    off_t this_chunk_size = (i == p - 1) ? (chunk_size + remainder) : chunk_size;
    rcnt = read(fdr, buffer, this_chunk_size); //parent reads the file
    if (rcnt < 0) {
        perror("read from file error");
        exit(1);
    }

    if (write(data_pfd[i][1], buffer, rcnt) < 0) { //write to data pipe
        perror("write to pipe error");
        free(buffer);
        exit(1);
    }
    close(data_pfd[i][1]); //close pipe after sending data
}
free(buffer);

for (int i = 0; i < p; i++) { //collect results from children
    int partial_result;
    if (read(result_pfd[i][0], &partial_result, sizeof(partial_result)) < 0) {
        perror("read result error");
        exit(1);
    }
    result += partial_result;
    close(result_pfd[i][0]);
}

for (int i = 0; i < p; i++) { //wait for all children to finish
    waitpid(child_pids[i], NULL, 0);
    active_processes--;
}
}

```

Εικόνα 3.3: Κώδικας που εκτελεί η διεργασία γονιός

```

close(fdr); //clean up
close(fdw);

return 0;
}

```

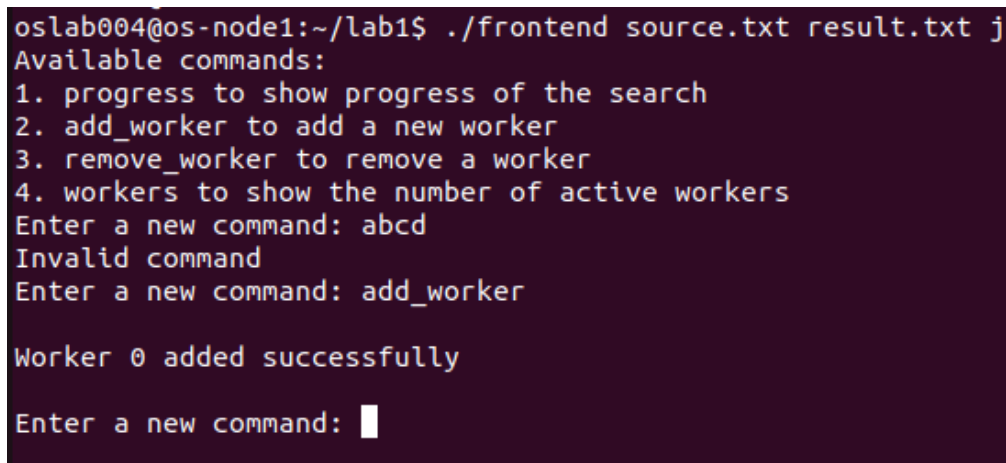
Εικόνα 3.4: Τερματισμός του προγράμματος

4 Εφαρμογή παράλληλης καταμέτρησης χαρακτήρων

Η εφαρμογή καταμέτρησης αποτελείται από τρία ανεξάρτητα αλλά συνεργαζόμενα προγράμματα, το frontend, τον dispatcher και τον worker. Το πρώτο πρόγραμμα που εκτελείται είναι το frontend, που καλείται από τον χρήστη και βασική του λειτουργία είναι η επικοινωνία χρήστη-εφαρμογής μέσω του terminal, αλλά και η κατάλληλη διαβίβαση των αιτημάτων στον dispatcher. Ο dispatcher είναι το κύριο πρόγραμμα της εφαρμογής, καθώς αναλαμβάνει την κύρια διαχείριση της εφαρμογής, την διεκπεραίωση των αιτημάτων και την διαχείριση των workers. Τέλος οι workers αναλαμβάνουν την αναζήτηση των εμφανίσεων του επιθυμητού χαρακτήρα στο κομμάτι που τους έχει ανατεθεί.

4.1 Frontend

Κατά την εκκίνηση του frontend παρουσιάζεται το μενού με τις διαθέσιμες εντολές προς τον χρήστη και στην συνέχεια αναμένει για την εισαγωγή εντολής (Εικόνα 4.1.1). Μετά την εισαγωγή της εντολής πραγματοποιείται ένας έλεγχος ορθότητας και αν είναι ορθή κωδικοποιείται με έναν μονοψήφιο αριθμό (εικόνα 4.1.2). Πιο συγκεκριμένα, η εντολή προσθήκης εργάτη `add_worker` κωδικοποιείται με 1, η εντολή ενημέρωσης προόδου `progress` με 2, η εντολή αφαίρεσης εργάτη `remove_worker` με 3 και η εντολή ενημέρωσης ενεργών εργατών `workers` με 4.

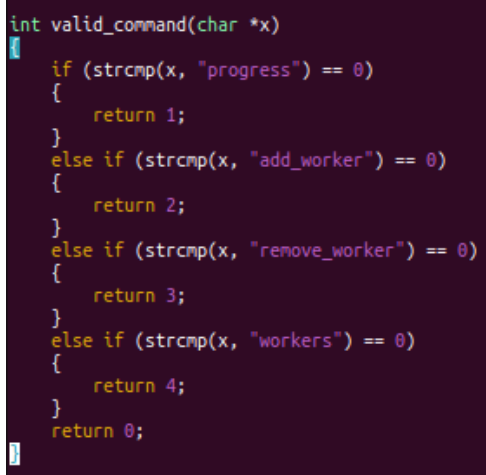


```
oslab004@os-node1:~/lab1$ ./frontend source.txt result.txt j
Available commands:
1. progress to show progress of the search
2. add_worker to add a new worker
3. remove_worker to remove a worker
4. workers to show the number of active workers
Enter a new command: abcd
Invalid command
Enter a new command: add_worker

Worker 0 added successfully

Enter a new command: █
```

Εικόνα 4.1.1: Μενού εφαρμογής και έλεγχος ορθότητας



```
int valid_command(char *x)
{
    if (strcmp(x, "progress") == 0)
    {
        return 1;
    }
    else if (strcmp(x, "add_worker") == 0)
    {
        return 2;
    }
    else if (strcmp(x, "remove_worker") == 0)
    {
        return 3;
    }
    else if (strcmp(x, "workers") == 0)
    {
        return 4;
    }
    return 0;
}
```

Εικόνα 4.1.2: Κωδικοποίηση εντολής

Η επικοινωνία frontend-dispatcher γίνεται μέσω ενός συνδυασμού pipes και signals. Πιο συγκεκριμένα, όταν το frontend λάβει κάποια εντολή στέλνει τον κωδικό από την κωδικοποίηση μέσω ενός pipe στον dispatcher και τον ενημερώνει με ένα σήμα. Ομοίως, όταν ο dispatcher ολοκληρώσει μια εντολή αποστέλλει μέσω pipe στο frontend το μήνυμα προς εκτύπωση και τον ενημερώνει μέσω ενός σήματος. Τέλος υπάρχει ένα ειδικό σήμα, το οποίο στέλνει ο dispatcher στο frontend, όταν ολοκληρωθεί η διαδικασία, αφού προηγουμένως έχει στείλει από ένα διαφορετικό pipe (δεν θέλουμε να μπλεχτεί με τα pipes των εντολών) το μήνυμα τερματισμού. Όταν το frontend λάβει το παραπάνω σήμα ο signal handler πραγματοποιεί την εκτύπωση του τελικού μηνύματος με τα αποτελέσματα και τερματίζει με ασφάλεια την εφαρμογή, τερματίζοντας τον dispatcher και κλείνοντας όλα τα ανοιχτά pipes (εικόνα 4.1.3).

```

void signal_handler(int sig)
{
    if (sig != SIGUSR2)
    {
        return;
    }
    char message[256];
    ssize_t bytes_read = read(fd_final_message[0], message, sizeof(message) - 1);
    if (bytes_read < 0)
    {
        perror("Failed to read final message from dispatcher");
        exit(1);
    }
    message[bytes_read] = '\0';
    if (write(fdw, message, strlen(message)) < 0){
        perror("failed to write result");
    }
    printf("%s\n", message);
    printf("All work has been completed. Exiting program.\n");
    kill(dispatcher_pid, SIGTERM);
    close(fd_frontend_to_dispatcher[1]);
    close(fd_dispatcher_to_frontend[0]);
    close(fd_final_message[0]);
    close(fdw);
    exit(0);
}

```

Εικόνα 4.1.3: Signal handler εξόδου

4.2 Dispatcher

Ο dispatcher δημιουργείται από το frontend με ένα fork, που ακολουθείται αμέσως από ένα `execv`. Για ορίσματα ο dispatcher έχει τα ορίσματα του frontend και επιπλέον τους απαραίτητους descriptors των pipes του frontend, διότι η άμεση πρόσβαση σε αυτά μέσω αντιγράφων, χάνεται μετά την εκτέλεση της `execv`.

Ο dispatcher αμέσως μετά την `execv` ορίζει μερικές global μεταβλητές για την χρήση τους εκτός της main, αλλά και μερικά structs που χρησιμοποιούνται για την υλοποίηση των συναρτήσεων. Ειδικότερα ορίζεται ένα struct worker, που περιέχει πληροφορίες για έναν worker, όπως ο αριθμός του worker, η περιοχή κειμένου που δουλεύει, οι descriptors για τα pipes dispatcher-worker, το `chunk_id` που το έχει ανατεθεί (αν δεν έχει ανατεθεί είναι -1) αλλά και αν ο συγκεκριμένος worker είναι active.

Για την διαχείριση των chunks του κειμένου έχει οριστεί ένα struct chunk. Το συγκεκριμένο struct περιέχει το πεδίο ορισμού το συγκεκριμένου chunk, τον worker που του έχει ανατεθεί (αν δεν του έχει ανατεθεί είναι -1) αλλά και boolean μεταβλητές που δηλώνουν αν το chunk είναι ανατεθειμένο και αν έχει επεξεργαστεί πλήρως. Οι δύο τελευταίες μεταβλητές είναι ιδιαίτερα χρήσιμες για το distribution της εργασίας αλλά και την εξασφάλιση της ορθής και αποτελεσματικής λειτουργίας σε περίπτωση που ο ανατεθειμένος worker πεθάνει απρόβλεπτα, όπως θα φανεί στην συνέχεια.

Στην συνέχεια ορίζεται ένα WorkMessage struct, το οποίο χρησιμοποιείται για την αποστολή της περιοχής αναζήτησης στο worker (αρχή και μέγεθος chunk). Τέλος αρχικοποιείται ένας πίνακας από workers struct, στον οποίο θα περιέχονται όλες οι πληροφορίες για όλους τους workers. Να σημειωθεί πως ο αριθμός των μέγιστων workers, δηλώνεται με μία macro. Όλα τα παραπάνω, παρουσιάζονται στο παρακάτω στιγμιότυπο.

```

#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
#include <stdbool.h>
#include <errno.h>
#define MAX_WORKERS 36

int frontend_to_dispatcher_pfd[2];
int dispatcher_to_frontend_pfd[2];
int active_workers = 0, completed_chunks = 0, total_chunks = 0, chunk_size = 10;
int total_chars_found = 0;
int fd_read, fd_write, fd_final_message;

typedef struct {
    pid_t pid;
    int worker_to_dispatcher_pfd[2];
    int dispatcher_to_worker_pfd[2];
    off_t start_pos, offset;
    int chunk_id; //which chunk each worker is working on: -1 if unassigned
    bool active;
} Worker;

typedef struct {
    off_t start_pos;
    off_t chunk_size;
    bool assigned;
    bool completed;
    int worker_id; //id of the worker that has been assigned the chunk
} WorkChunk;

typedef struct {
    off_t start_pos;
    off_t chunk_size;
} WorkMessage;

Worker workers[MAX_WORKERS]; //declare array of workers statically

```

Εικόνα 4.2.1: Βασικά structs και data για τον dispatcher

Οι δύο πρώτες συναρτήσεις που εκτελούνται είναι η `inititalize_workers` και η `createworkchunks`. Η `inititalize_workers` θέτει όλους τους workers inactive και ότι δεν έχουν ανατεθειμένο chunk. Με τον όρο `inactive` εννοείται ότι ο συγκεκριμένος worker δεν έχει δημιουργηθεί ακόμη. Η `create_work_chunk` με την χρήση της `lseek` προσδιορίζει το μέγεθος του κειμένου και υπολογίζει το πλήθος στον chunk. Στην συνέχεια με `dynamic memory allocation`, δημιουργείται ο πίνακας των chunk, όπου κάθε chunk είναι `uncompleted` και `unassigned`.

Όπως αναφέρθηκε η εντολές από το frontend, λαμβάνονται με τον συνδυασμό σημάτων και pipe. Ειδικότερα, αν σταλεί από το frontend το σήμα `SIGUSR1`, ο signal handler του dispatcher διαβάζει από το pipe την κωδικοποιημένη εντολή, την αποκωδικοποιεί και καλεί την αντίστοιχη συνάρτηση. Τα παραπάνω παρουσιάζονται στο στιγμιότυπο, που ακολουθεί.

```

void signal_handler(int sig) {
    char code_sent[1];
    read(frontend_to_dispatcher_pfd[0], &code_sent, sizeof(code_sent));
    int code = atoi(code_sent);
    printf("\n");
    switch (code) {
        case 1:
            progress_info();
            break;
        case 2:
            add_worker();
            break;
        case 3:
            remove_worker();
            break;
        case 4:
            workers_info();
            break;
        default:
            break;
    }
}

```

Εικόνα 4.2.2: Ο signal handler του dispatcher για SIGUSR1

Στη main του προγράμματος όσο υπάρχουν ανοιχτά chunks εκτελείτε ένα loop, όπου αναμένεται μια εντολή από το frontend, η οποία αποκωδικοποιείται και στην συνέχεια εκτελείτε. Η ανάγνωση των εντολών από το pipe γίνεται με non blocking τρόπο, ώστε να μην διακόπτεται η ροή του προγράμματος. Στο loop τρέχει και η συνάρτηση distribute_work, με σκοπό την κατανομή των μη ανατεθειμένων chunks στους μη εργαζόμενους workers, στέλνοντας τους μέσω pipe ένα work_message struct.

```

while (completed_chunks < total_chunks) {
    for (int i = 0; i < MAX_WORKERS; i++) {
        if (workers[i].active && workers[i].chunk_id != -1) {
            int flags = fcntl(workers[i].worker_to_dispatcher_pfd[0], F_GETFL, 0); //retrieve current file status flags
            fcntl(workers[i].worker_to_dispatcher_pfd[0], F_SETFL, flags | O_NONBLOCK); //set the file descriptor to non-blocking & any previous flags
            int count;
            int read_result = read(workers[i].worker_to_dispatcher_pfd[0], &count, sizeof(count));

            if (read_result > 0) {
                total_chars_found += count;
                workchunk(workers[i].chunk_id).completed = true;
                completed_chunks++;
                workers[i].chunk_id = -1;
            }
            else if (read_result < 0 && errno != EAGAIN && errno != EWOULDBLOCK) { //checks if the error is not due to the non-blocking nature of the pipe
                perror("Error reading from worker");
            }
            fcntl(workers[i].worker_to_dispatcher_pfd[0], F_SETFL, flags); //restore original flags
        }
    }
    if (active_workers > 0) {
        distribute_work();
    }
    usleep(50000);
}

```

Εικόνα 4.2.3: Loop της main του dispatcher

Για την εντολή προσθήκης εργάτη εκτελείται η συνάρτηση add_worker από τον dispatcher. Εφόσον ο αριθμός των εργατών δεν ξεπερνάει τον μέγιστο επιτρεπτό αριθμό, ο dispatcher δημιουργεί τα pipes για την επικοινωνία του νέου worker-dispatcher και στην συνέχεια πραγματοποιεί ένα fork. Αν το fork επιτύχει, πραγματοποιείται ένα execv με το εκτελέσιμο worker και διαβιβάζονται ως ορίσματα χρήσιμες πληροφορίες για τον worker, όπως το file name του αρχείου κειμένου, οι descriptors των pipes αλλά και ο χαρακτήρας προς αναζήτησης. Στην περίπτωση, που οι workers υπερβαίνουν το μέγιστο αριθμό, επιστρέφεται στο frontend και στην συνέχεια στον χρήστη το μήνυμα "No available workers".

Η εντολή αφαίρεσης worker υλοποιείται από την συνάρτηση remove_worker και αφαιρεί πάντα τον πιο πρόσφατο εργάτη, που έχει δημιουργηθεί. Η ασφαλής αφαίρεση πραγματοποιείται αποστέλλοντας ένα SIGTERM σήμα στον worker και κλείνοντας όλα τα ανοιχτά άκρα από pipes μεταξύ dispatcher και worker. Για την ορθή συνέχεια λειτουργίας της εφαρμογής αν ο worker εργαζόταν σε ένα chunk αυτό σημειώνεται ως unassigned, ώστε να το αναλάβει ένας άλλος worker σε μια μελλοντική κλήση της distribute_work. Τέλος στην περίπτωση, όπου κανένας worker δεν είναι active επιστρέφεται στο frontend και στον χρήστη το μήνυμα "There are no active workers to remove". Η υλοποίηση της remove_worker παρουσιάζεται παρακάτω.

```

void remove_worker() {
    char message[50];
    if (active_workers == 0) {
        snprintf(message, sizeof(message), "There are no active workers to remove\n");
        if (write(dispatcher_to_frontend_pfd[1], message, strlen(message)) < 0) {
            perror("Error writing in to pipe from dispatcher to frontend");
        }
        return;
    }
    int worker_id = -1;
    for (int i = active_workers-1; i >= 0; i--) { //Last Added First Removed
        if (workers[i].active) {
            worker_id = i;
            break;
        }
    }

    if (worker_id != -1) {
        int chunk_id = workers[worker_id].chunk_id;
        if (kill(workers[worker_id].pid, SIGTERM) < 0) {
            perror("Process kill failed");
            snprintf(message, sizeof(message), "Failed to terminate worker %d", worker_id);
            if (write(dispatcher_to_frontend_pfd[1], message, strlen(message)) < 0) {
                perror("Error writing to frontend pipe");
            }
            return;
        }
        close(workers[worker_id].dispatcher_to_worker_pfd[0]);
        close(workers[worker_id].dispatcher_to_worker_pfd[1]);
        close(workers[worker_id].worker_to_dispatcher_pfd[0]);
        close(workers[worker_id].worker_to_dispatcher_pfd[1]);
        workers[worker_id].active = false;
        workers[worker_id].pid = 0;
        workers[worker_id].chunk_id = -1;
        active_workers--;
        if (chunk_id != -1 && workchunk[chunk_id].assigned) {
            workchunk[chunk_id].assigned = false;
            workchunk[chunk_id].worker_id = -1;
        }
    }
    snprintf(message, sizeof(message), "Worker %d removed successfully\n", worker_id);
    if (write(dispatcher_to_frontend_pfd[1], message, strlen(message)) < 0) {
        perror("Error writing in to pipe from dispatcher to frontend");
    }
    return;
}

```

Εικόνα 4.2.4: Συνάρτηση remove_worker

Η συνάρτηση progress_info επιστρέφει το ποσοστό των ολοκληρωμένων chunks και τον αριθμό των εμφανίσεων, που έχουν βρεθεί, ώστε να ενημερωθεί ο χρήστης για την πρόοδο της αναζήτησης, αφού έχει εισαγάγει την εντολή progress.

Η εντολή workers, υλοποιείται από την συνάρτηση workers και επιστρέφει τον αριθμό των workers που είναι active, ώστε να ενημερωθεί ο χρήστης για τον ενεργό αριθμό χρηστών.

Οι υλοποιήσεις των δύο παραπάνω συναρτήσεων παρουσιάζονται στην συνέχεια.

```

void progress_info() {
    int completed = 0;
    for (int i = 0; i < total_chunks; i++) {
        if (workchunk[i].completed) {
            completed++;
        }
    }
    float progress_percentage = (total_chunks > 0) ? ((float)completed / total_chunks) * 100 : 0;

    char progress_message[256]; //message to be sent to frontend
    snprintf(progress_message, sizeof(progress_message),
        "Progress: %.2f%% completed (%d/%d chunks), Total Characters Found: %d\n",
        progress_percentage, completed, total_chunks, total_chars_found);

    if (write(dispatcher_to_frontend_pfd[1], progress_message, strlen(progress_message)) < 0) { //send progress info to frontend
        perror("Failed to write progress info to pipe");
    }
}

void workers_info() {
    char workers_message[256];
    snprintf(workers_message, sizeof(workers_message), "Active Workers: %d\n", active_workers);
    if (write(dispatcher_to_frontend_pfd[1], workers_message, strlen(workers_message)) < 0) {
        perror("Failed to write workers info to pipe");
    }
}

```

Εικόνα 4.2.5: Συναρτήσεις progress_info και workers_info

Όταν όλα τα chunks έχουν μετρηθεί η main του dispatcher καλεί την συνάρτηση collect_results. Αυτή η συνάρτηση στέλνει μέσω ειδικού pipe στο dispatcher το τελικό μήνυμα με τον αριθμό των εμφανίσεων και τον ενημερώνει με ένα σήμα SIGUSR2, το οποίο διαχειρίζεται ο signal handler, που αναφέρθηκε στην ανάλυση του frontend. Μετά την κλήση της collect_results ο dispatcher στέλνει SIGTERM στους active_workers, κλείνει τα ανοιχτά pipes και απελευθερώνει την δεσμευμένη μνήμη του πίνακα των chunks (Εικόνα 4.2.6)


```

collect_results();
for (int i = 0; i < MAX_WORKERS; i++) {
    if (workers[i].active) {
        kill(workers[i].pid, SIGTERM);
        close(workers[i].dispatcher_to_worker_pfd[1]);
        close(workers[i].worker_to_dispatcher_pfd[0]);
    }
}
close(frontend_to_dispatcher_pfd[0]);
close(dispatcher_to_frontend_pfd[1]);
free(workchunk);
return 0;

```

Εικόνα 4.2.6: Τελευταία βήματα πριν την ολοκλήρωση εκτέλεσης του dispatcher

4.3 Worker

Ο παρακάτω κώδικας υλοποιεί μία διεργασία worker που συνεργάζεται με τον dispatcher μέσω pipes για την παράλληλη επεξεργασία αρχείων. Ο worker λαμβάνει μέσω pipe μια δομή WorkMessage από τον dispatcher, η οποία περιέχει τη θέση εκκίνησης (start_pos) και το μέγεθος chunk (chunk_size) που του ανατίθεται να επεξεργαστεί. Επιπρόσθετα έχουμε βάλει τον worker να έχει μια global μεταβλητή που να λέει αν η δουλειά αναζήτησης στο αρχείο έχει ολοκληρωθεί με επιτυχία.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <stdbool.h>
#include <string.h>

int dispatcher_to_worker_pfd[2];
int worker_to_dispatcher_pfd[2];
volatile sig_atomic_t should_exit = 0; //variable can change at any time
bool task_completed;

typedef struct {
    off_t start_pos;
    off_t chunk_size;
} WorkMessage;

```

Εικόνα 4.3.1: Δήλωση των global variables και του WorkMessage struct

Έπειτα ορίζουμε την συνάρτηση του προγράμματος, την Work, στην οποία ουσιαστικά γίνεται η αναζήτηση των χαρακτήρων που παίρνει ως ορίσματα το read file, το WorkMessage που έχει λάβει από τον dispatcher καθώς και τα pipes επικοινωνίας προς και από τον dispatcher. Εφόσον και αν ανοίξει με επιτυχία το read file, μετακινήσει με επιτυχία τον δείκτη στη σωστή θέση με lseek, τον buffer στον οποίο θα φορτώσει τους χαρακτήρες του κειμένου και μετρήσει τους χαρακτήρες στο chunk που του αναλογεί, τότε μπαίνει ένα delay της τάξης του 1 δευτερολέπτου για καλύτερο visualisation, και τέλος αποδεσμεύουμε τη μνήμη που έπαιρνε ο buffer και κλείνουμε το αρχείο με descriptor fdr. Άπαξ και το πρόγραμμα μπει στον term_handler, η μεταβλητή should_exit γίνεται με 1 (που ισοδυναμεί με integer true).

```

void work(const char* file, char target, WorkMessage work_msg, int dispatcher_to_worker_pfd[2], int worker_to_dispatcher_pfd[2]) {
    off_t start_pos = work_msg.start_pos;
    off_t offset = work_msg.chunk_size;

    int fdr = open(file, O_RDONLY);
    if (fdr < 0) {
        perror("Worker failed to open file");
        exit(1);
    }

    if (lseek(fdr, start_pos, SEEK_SET) < 0) { //move to start position
        perror("Worker failed to seek in file");
        close(fdr);
        exit(1);
    }

    char* buffer = malloc(offset);
    if (buffer == NULL) {
        perror("Worker: Memory allocation error");
        close(fdr);
        exit(1);
    }

    int bytes_read = read(fdr, buffer, offset);
    if (bytes_read < 0) {
        perror("Worker failed to read file");
        free(buffer);
        close(fdr);
        exit(1);
    }

    int count = 0; //count occurrences of target character
    for (int i = 0; i < bytes_read; i++) {
        if (buffer[i] == target) {
            count++;
        }
        if (should_exit) {
            printf("Worker: Termination requested during processing\n");
            break;
        }
    }

    sleep(1); // puts a small delay after work is done
    if (write(worker_to_dispatcher_pfd[1], &count, sizeof(count)) < 0) {
        perror("Worker failed to write to dispatcher");
    }
    free(buffer);
    close(fdr);
}

void term_handler(int sig) {
    should_exit = 1;
}

```

Εικόνα 4.3.2: Συνάρτηση Work και term_handler

Όσον αφορά τη main() συνάρτηση του προγράμματος αρχικά λαμβάνει τα ορίσματα που δέχεται μέσω execv() από τον dispatcher και κλείνει τα άχρηστα για εκείνον pipes. Έπειτα μπαίνει σε μία while loop που τερματίζει αν και μόνο αν η boolean μεταβλητή που έχει αντιγράψει από τον dispatcher γίνει TRUE, μέσα στην οποία ουσιαστικά διαβάζει τα work messages που λαμβάνει από τον dispatcher. Αν ο dispatcher έδωσε στον worker να αναλύσει chunk του οποίου το size είναι μικρότερο από 10 bytes, αυτό σημαίνει ότι βρισκόμαστε στο τελευταίο chunk, άρα αφήνουμε τον συγκεκριμένο worker να κάνει αυτό που του ανέθεσε ο dispatcher, και να βγει μετά από το loop. Τέλος απλά κλείνει τα pipes που απέμεναν ανοιχτά και τερματίζει.

```

int main(int argc, char* argv[]) {
    signal(SIGTERM, term_handler);
    off_t start_pos = atoll(argv[1]); //to long long if file is very long
    off_t offset = atoll(argv[2]);
    const char* file = argv[3];
    char target = argv[4][0];
    dispatcher_to_worker_pfd[0] = atoi(argv[5]);
    worker_to_dispatcher_pfd[1] = atoi(argv[6]);
    task_completed = (argv[7][0] == '1');
    close(dispatcher_to_worker_pfd[1]); //worker does not write to this pipe
    close(worker_to_dispatcher_pfd[0]); //worker does not read from this pipe

    WorkMessage work_msg;
    while (!task_completed) {
        if (read(dispatcher_to_worker_pfd[0], &work_msg, sizeof(work_msg)) <= 0) {
            exit(1);
        }
        if (work_msg.chunk_size < 10){
            task_completed = true;
        }
        work(file, target, work_msg, dispatcher_to_worker_pfd, worker_to_dispatcher_pfd);
    }
    close(dispatcher_to_worker_pfd[0]);
    close(worker_to_dispatcher_pfd[1]);
    return 0;
}

```

Εικόνα 4.3.3: Main συνάρτηση του Worker

4.4 Παρατηρήσεις-Βελτιώσεις

Κατά την εξέταση επισημάνθηκε από τον εξεταστή πως το `chunk_size` για λόγους αποδοτικότητας θα έπρεπε να είναι μεγαλύτερο από 10 bytes, το οποίο δεν αποτελεί πρόβλημα, καθώς το `chunk_size` μπορεί να απλά να μεταβληθεί χωρίς να γίνει κάποια αλλαγή στον υπόλοιπο κώδικα. Τέλος, προτάθηκε από τον εξεταστή ως βελτίωση, η δυνατότητα προσθήκης και αφαίρεσης πολλών εργατών με μία εντολή από τον χρήστη, ώστε να μην χρειάζεται να πληκτρολογεί επανειλημμένα την ίδια εντολή.