

# OSLab 2023-24 — 1η Εργαστηριακή Άσκηση

Κόρδας Νικόλαος - Α.Μ.: 03121032  
Κριθαρίδης Κωνσταντίνος - Α.Μ.: 03121045

29 Μαρτίου 2024

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

Αρχικά, αντιγράψαμε τον φάκελο `char-count` στο `home directory` μας και στην συνέχεια τον μεταφέραμε με χρήση της `scp` στον προσωπικό μας υπολογιστή. Κατόπιν, ξεκινήσαμε να εξετάζουμε τον κώδικα που μας δόθηκε στο αρχείο `a1.1-C.c`. Διαπιστώνουμε πως το εκτελέσιμο που παράγεται από αυτό, ουσιαστικά, ανοίγει το αρχείο που του έχουμε περάσει ως πρώτο όρισμα προς ανάγνωση και το δεύτερο προς εγγραφή, με χρήση της `fopen()`. Το τρίτο όρισμα είναι ο χαρακτήρας που πρόκειται να αναζητηθεί στο αρχείο. Στα παραπάνω ανοίγματα γίνονται και οι απαραίτητοι έλεγχοι αποτυχίας της εντολής. Κατόπιν, το πρόγραμμα διατρέχει ολόκληρο το πρώτο αρχείο, (μέχρι τον χαρακτήρα `EOF`) και με την χρήση της `fgetc()` διαβάει έναν έναν τους χαρακτήρα τους ελέγχοντας αν είναι όμοιοι με τον χαρακτήρα που δόθηκε για ανάγνωση. Εάν είναι αυξάνει έναν μετρητή. Τέλος, εγγράφει το αποτέλεσμα στο δεύτερο αρχείο με την εντολή `fprintf()` και κλείνει τα δύο ανοιχτά αρχεία με την `fclose()`.

Στόχος μας είναι να μιμηθούμε την παραπάνω λειτουργία αντικαθιστώντας την κλήση συναρτήσεων της βιβλιοθήκης της C με `system calls`. Με άλλα λόγια, να κάνουμε το πρόγραμμά μας `linux specific`. Ο κώδικάς μας γράφεται στο αρχείο `a1.1-system_calls.c` ενώ το εκτελέσιμο που παράγεται είναι το `a1.1-system_calls`. Κατά την κλήση του εκτελέσιμου παίρνουμε τα ίδια ορίσματα με παραπάνω. Έτσι, στον κώδικά μας, το πρώτο πράγμα που γίνεται είναι ο έλεγχος των ορισμάτων που δίνονται από τον χρήστη με την συνάρτηση `argument_handling()`. Πιο συγκεκριμένα, αυτή η συνάρτηση ελέγχει εάν ο αριθμός των ορισμάτων είναι σωστός (ακριβώς 3 από τον χρήστη) και αν δόθηκε μόνο ένας χαρακτήρας προς αναζήτηση στο αρχείο. Στην συνέχεια, ορίζοντας τα κατάλληλα `flags` που απαιτούνται για την εγγραφή και την ανάγνωση των αρχείων, χρησιμοποιούμε την κλήση συστήματος `open()` η οποία επιστρέφει `file descriptor` (ακέραιο) για το κάθε αρχείο. Μετά, σύμφωνα και με τις διαφάνειες της εργασίας εκτελούμε την ανάγνωση του αρχείου που δόθηκε χαρακτήρα - χαρακτήρα και κάθε φορά που αυτός είναι όμοιος με αυτόν που δόθηκε προς εύρεση, αυξάνουμε έναν μετρητή κατά 1. Λεπτομερέστερα, η ανάγν-

ωση πραγματοποιείται με τη βοήθεια ενός buffer (πίνακας χαρακτήρων 1024 byte) και της κλήσης συστήματος read(). Η read() σε κάθε επανάληψη του βρόχου διαβάζει μέχρι 1023 byte από το αρχείο, τα αποθηκεύει στον buffer, τον οποίο μετατρέπουμε σε συμβολοσειρά τοποθετώντας στο τέλος του τον χαρακτήρα '\0'. Διατρέχοντάς τον, μετράμε τις εμφανίσεις του c2c και ξανακάνουμε το ίδιο μέχρι να φτάσουμε στο τέλος του αρχείου (rcnt = 0, δηλαδή, η read() δε μπόρεσε να διαβάσει κανένα byte από το αρχείο). Μόλις λήξει η διαδικασία ανάγνωσης, κλείνουμε τον file descriptor του αρχείου αυτού με την κλήση συστήματος close(). Με την snprintf() τυπώνουμε το μήνυμα απόκρισής μας με τον αριθμό φορών που βρέθηκε ο c2c σε έναν buffer. Στην συνέχεια, αξιοποιούμε την κλήση συστήματος write() και τον κώδικα των διαφανειών για να γράψουμε το μήνυμα αυτό στο αρχείο που μας έδωσε ο χρήστης. Τέλος, κλείνουμε και αυτό το αρχείο και τερματίζουμε το προγράμμα μας. Αξίζει να σημειώσουμε πως κάθε κλήση συστήματος συνοδεύεται από τον κατάλληλο έλεγχο για την αποτυχία εκτέλεσής της.

Ο κώδικας για το πρώτο μέρος της εργαστηριακής άσκησης φαίνεται παρακάτω:

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 void argument_handling(int argc, char **argv);
10
11 int main(int argc, char **argv) {
12     argument_handling(argc, argv);
13     int fdr, fdw, oflags, mode;
14     char c2c = 'a';
15     int count = 0;
16     oflags = O_CREAT | O_WRONLY | O_TRUNC;
17     mode = S_IRUSR | S_IWUSR;
18     if((fdr = open(argv[1], O_RDONLY)) == -1) {
19         perror("Problem opening file to read\n");
20         exit(1);
21     }
22     if((fdw = open(argv[2], oflags, mode)) == -1) {
23         perror("Problem opening file to write\n");
24         exit(1);
25     }
26
27     ssize_t rcnt;
28     char buff[1024];
29     c2c = argv[3][0];
30
31     while(1) {
32         rcnt = read(fdr, buff, sizeof(buff) - 1);
33         if(rcnt == 0) break; //end of file
34         if(rcnt == -1) {
35             perror("Failed to read file\n");
36             exit(1);
37         }
```

```

37     }
38     buff[rcnt] = '\0';
39     for(size_t i = 0; i < rcnt; i++){
40         if(buff[i] == c2c) count++;
41     }
42 }
43 close(fdr);
44 ssize_t wcnt;
45 snprintf(buff, sizeof(buff), "The character '%c' appears %d times
46         in file %s.\n", c2c, count, argv[1]);
47 size_t len = strlen(buff), idx = 0;
48 do {
49     wcnt = write(fdw, buff+idx, len-idx);
50     if(wcnt == -1){
51         perror("Failed to write to file\n");
52         exit(1);
53     }
54     idx += wcnt;
55 } while(idx < len);
56 close(fdw);
57 return 0;
58 }
59 void argument_handling(int argc, char **argv) {
60     if(argc != 4) {
61         perror("There should be three arguments!! (source file,
62             destination file and character)\n");
63         exit(1);
64     }
65     if(strlen(argv[3]) > 1) {
66         perror("You can search for specific characters, not entire
67             strings!!\n");
68         exit(1);
69     }
69 }

```

## 2 Δημιουργία διεργασιών

Αυτή τη φορά παραθέτουμε πρώτα τον κώδικα της άσκησης διότι αυτός περιέχει απαντήσεις για όλα τα παρακάτω ερωτήματα.

```

1 #include <unistd.h>
2 #include <string.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6 #include <sys/stat.h>
7 #include <sys/types.h>
8 #include <fcntl.h>
9
10 void argument_handling(int argc, char **argv);
11 void print(int fdw, char *buff);

```

```

12 void child(int *x, int fdr, int fdw, char c2c, char *file_to_write)
13     ;
14 void parent(pid_t p, int *x);
15 int main(int argc, char **argv) {
16     argument_handling(argc, argv);
17     int fdr, fdw;
18     int x = 17;
19     int oflags, mode;
20     oflags = O_CREAT | O_WRONLY | O_TRUNC;
21     mode = S_IRUSR | S_IWUSR;
22     if((fdr = open(argv[1], O_RDONLY)) == -1){
23         perror("Problem opening file to read\n");
24         exit(1);
25     }
26     if((fdw = open(argv[2], oflags, mode)) == -1){
27         perror("Problem opening file to write\n");
28         exit(1);
29     }
30
31     pid_t p;
32     p = fork();
33     if(p < 0){
34         perror("fork");
35         exit(1);
36     }
37     else if(p == 0){
38         child(&x, fdr, fdw, argv[3][0], argv[1]);
39         exit(0);
40     }
41     else{
42         parent(p, &x);
43     }
44
45     close(fdr);
46     close(fdw);
47
48     p = fork();
49     if(p < 0){
50         perror("fork");
51         exit(1);
52     }
53     else if(p == 0){
54         char *argv2[] = {"./a1.1-C\0", "", "", "", NULL};
55         for(int i = 1; i <= 3; i++){
56             argv2[i] = (char *)malloc(sizeof(argv[i]));
57             strcpy(argv2[i], argv[i]);
58         }
59         execv(argv2[0], argv2);
60         exit(0);
61     }
62     else{
63         int status;
64         wait(&status);
65         char buff[1024];
66         snprintf(buff, sizeof(buff), "Child process %d exited with
        status %d.\n", p, status);

```

```

67     print(1, buff);
68     exit(0);
69 }
70 }
71
72 void argument_handling(int argc, char **argv) {
73     if(argc != 4) {
74         perror("There should be three arguments!! (source file,
75             destination file and character)\n");
76         exit(1);
77     }
78     if(strlen(argv[3]) > 1) {
79         perror("You can search for specific characters, not entire
80             strings!!\n");
81         exit(1);
82     }
83 }
84
85 void print(int fdw, char *buff) {
86     size_t idx = 0, len = strlen(buff);
87     ssize_t wcnt;
88     do{
89         wcnt = write(fdw, buff+idx, len-idx);
90         if(wcnt == -1){
91             perror("write\n");
92             exit(1);
93         }
94         idx += wcnt;
95     }while(idx < len);
96 }
97
98 void child(int *x, int fdr, int fdw, char c2c, char *file_to_write)
99 {
100     pid_t mypid = getpid(), parpid = getppid();
101     char buff[1024];
102     snprintf(buff, sizeof(buff), "Hello World!\nMy pid is %d.\nMy
103         parent's pid is %d.\n", mypid, parpid);
104     print(1, buff);
105
106     *x = 42;
107     snprintf(buff, sizeof(buff), "Variable x is %d (child).\n", *x);
108     print(1, buff);
109
110     ssize_t rcnt;
111     int count = 0;
112     while(1) {
113         rcnt = read(fdr, buff, sizeof(buff)-1);
114         if(rcnt == 0) break; //end of file
115         if(rcnt == -1){
116             perror("Failed to read file\n");
117             exit(1);
118         }
119         buff[rcnt] = '\0';
120         for(size_t i = 0; i < rcnt; i++){
121             if(buff[i] == c2c) count++;
122         }
123     }

```

```

120 }
121 snprintf(buff, sizeof(buff), "The character '%c' appears %d times
    in file %s.\n", c2c, count, file_to_write);
122 print(fdw, buff);
123 }
124
125 void parent(pid_t p, int *x) {
126     char buff[1024];
127     snprintf(buff, sizeof(buff), "My child's pid is %d.\n", p);
128     print(1, buff);
129     int status;
130     snprintf(buff, sizeof(buff), "Variable x is %d (parent).\n", *x);
131     print(1, buff);
132     wait(&status);
133     snprintf(buff, sizeof(buff), "Child process %d exited with status
        %d.\n", p, status);
134     print(1, buff);
135 }

```

## 2.1 Ερώτημα 1

Για να δημιουργήσουμε μία διεργασία παιδί καλούμε την συνάρτηση `fork()`. Συγκεκριμένα, η συνάρτηση αυτή επιστρέφει 0 στην διεργασία παιδί που δημιουργεί, το `pid` του παιδιού στον γονέα, ενώ την τιμή -1 σε περίπτωση αποτυχίας. Για αυτό μετά την κλήση της διακρίνουμε περιπτώσεις.

- Στην περίπτωση -1 εκτυπώνουμε μήνυμα λάθους.
- Στην περίπτωση 0 εκτελούμε τον κώδικα της διεργασίας παιδιού που περιέχεται στην συνάρτηση `child()`.
- Σε περίπτωση κάποιου αναγνωριστικού `pid` εκτελείται ο κώδικας της πατρικής διαδικασίας, `parent()`.

Για το πρώτο ερώτημα, η `child()` απλά χαιρετάει τον κόσμο με ένα μήνυμα που αποθηκεύεται σε έναν buffer και τυπώνεται με την συνάρτηση `print()` την οποία έχουμε ορίσει εμείς και περιέχει τον κώδικα των διαφανειών για την κλήση συστήματος `write()`. Το μήνυμα που τυπώνει περιέχει το `pid` της και το `pid` της γονεϊκής διαδικασίας της. Προκειμένου να αποκτήσει το `pid` της χρησιμοποιεί την ρουτίνα `getpid()`, ενώ για του γονέα της την `getppid()`. Αντίστοιχα, ο γονέας περιμένει τον τερματισμό κάθε παιδιού του με την `wait()`, από την οποία λαμβάνει και το `status` του τερματισμού. Μόλις το παιδί τερματίσει εκτυπώνει ένα μήνυμα στην οθόνη που περιέχει το `status` αυτό αλλά και τον `pid` του παιδιού (Child process %d exited with status %d. \n).

## 2.2 Ερώτημα 2

Στο ερώτημα αυτό ορίζουμε την μεταβλητή  $x$  και της δίνουμε την τιμή 17 στον γονέα. Κατόπιν δημιουργούμε τα παιδιά. Κατά την δημιουργία τους, λόγω του

copy on write, στιγμιαία έχουμε την ίδια τιμή και σε αυτά (17). Στην συνέχεια, όμως, εφόσον κάθε παιδί έχει το δικό του stack, ουσιαστικά, έχουμε αντιγραφή μόνο του ονόματος της μεταβλητής, αλλά όχι της πραγματικής υπόστασης της στη μνήμη. Για αυτό και όταν αλλάζει η τιμή της σε 42 από το παιδί, αυτό που βλέπουμε στην οθόνη δεν είναι πλέον η γονεϊκή τιμή 17, αλλά η νέα 42. Στην παρακάτω εικόνα φαίνεται το output αυτού του προγράμματος, για αυτό και το προηγούμενο ερώτημα.

```
orion@orionpc ~/Desktop/ntua/semester6/
└─ main ── ./a1.2-fork 1.txt 2.txt a
My child's pid is 14112.
Variable x is 17 (parent).
Hello World!
My pid is 14112.
My parent's pid is 14111.
Variable x is 42 (child).
Child process 14112 exited with status 0.
Child process 14113 exited with status 0.
```

### 2.3 Ερώτημα 3

Η επέκταση που καλούμαστε να κάνουμε σε αυτό το ερώτημα αφορά την ανάθεση αναζήτησης του χαρακτήρα στην διεργασία παιδί. Ουσιαστικά, συμπληρώνουμε τον κώδικα της συνάρτησης `child()` με τον κώδικα `a1.1-system_calls.c`. Το αρχείο ανοίγεται από την γονεϊκή διαδικασία και ο file descriptor περνιέται ως όρισμα στη συνάρτηση - παιδί. Η ανάγνωση γίνεται με τον ίδιο τρόπο, όπως και στην προηγούμενη άσκηση. Καθώς το παιδί και ο γονιός δεν επικοινωνούν (ακόμα) με κάποιο τρόπο μεταξύ τους, το παιδί αναλαμβάνει και το γράψιμο του αποτελέσμάτος του στο αρχείο που έχει επιλέξει ο χρήστης. Ο file descriptor του τελευταίου περνιέται και αυτός ως όρισμα στη συνάρτηση - παιδί. Ο κώδικας του `parent()` δεν αλλάζει και αυτό είναι λογικό, αφού η γονεϊκή διαδικασία δεν επωμίζεται επιπλέον αρμοδιότητες σε αυτό το ερώτημα και άρα συνεχίζει απλά να περιμένει το παιδί να ολοκληρώσει την ανάγνωση του αρχείου και να τερματίσει (`wait()`).

### 2.4 Ερώτημα 4

Στο τελευταίο αυτό ερώτημα, αντί να συμπληρώνουμε τον κώδικα για την ανάγνωση του αρχείου (και την εγγραφή) στην συνάρτηση `child()`, επιλέγουμε να εκτελέσουμε απευθείας το εκτελέσιμο του κώδικα `a1.1-system_calls.c`, του `a1.1-system_calls`. Αυτό επιτυγχάνονται με την ρουτίνα `execv()`. Για να λειτουργήσει σωστά το εκτελέσιμο πρέπει να του περάσουμε τα ίδια ορίσματα με τα οποία κλήθηκε το `a1.2-fork`. Αυτό το επιτυγχάνουμε επίσης με την `execv()`. Η κλήση και

η χρήση του προγράμματος πράγματι γίνεται με επιτυχία, όπως άλλωστε φαίνεται και στην εικόνα που παραθέσαμε παραπάνω.

### 3 Διαδιεργασική Επικοινωνία

Παραθέτουμε τον κώδικα και για αυτή την άσκηση πρώτα.

```
1 #include <unistd.h>
2 #include <math.h>
3 #include <string.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/wait.h>
7 #include <sys/stat.h>
8 #include <sys/types.h>
9 #include <fcntl.h>
10 #include <signal.h>
11
12 #define CPUCORES 8
13 #define STD_ERR 2
14
15 int P = 0;
16
17 void argument_handling(int argc, char **argv);
18 void print(int fdw, char *buff);
19 int min(int a, int b);
20 void child(int fdr, off_t start, size_t bytes_to_read, int *pfd,
21           char c2c);
22 void parent(int fdw, int **pfd, char c2c, char *file_to_write);
23 void sigintHandler(int sig_num);
24
25 int main(int argc, char **argv) {
26     argument_handling(argc, argv);
27     int fdr, fdw;
28     int oflags, mode;
29     oflags = O_CREAT | O_WRONLY | O_TRUNC;
30     mode = S_IRUSR | S_IWUSR;
31     if((fdr = open(argv[1], O_RDONLY)) == -1){
32         perror("Problem opening file to read\n");
33         exit(1);
34     }
35     if((fdw = open(argv[2], oflags, mode)) == -1){
36         perror("Problem opening file to write\n");
37         exit(1);
38     }
39     struct stat st;
40     if(stat(argv[1], &st) < 0){
41         perror("Stat error\n");
42         exit(1);
43     }
44     if(signal(SIGINT, sigintHandler) < 0){
45         perror("Could not establish SIGINT handler");
46         exit(1);
47     }
48     printf("Initiated SIGINT handler\n");
49     sleep(1); //To be able to test the SIGINT handler
```



```

49 off_t size = st.st_size;
50 P = CPUCORES;
51 off_t batch_size = size / P;
52 //printf("File size = %ld, batch size = %ld, P = %d\n", size,
    batch_size, P);
53 off_t idx = 0, extra_left = size - P * batch_size;
54 int **pfd = (int**)malloc(P * sizeof(int *));
55 for(int i = 0; i < P; ++i) {
56     pfd[i] = (int *)malloc(2 * sizeof(int));
57 }
58 pid_t p;
59 for(int i = 0; i < P; i++) {
60     size_t bytes_to_read = batch_size + (i < extra_left);
61     if(pipe(pfd[i]) < 0){
62         perror("pipe\n");
63         exit(1);
64     }
65     p = fork();
66     if(p < 0) {
67         perror("fork\n");
68         exit(1);
69     }
70     else if(p == 0) {
71         child(fdr, idx, bytes_to_read, pfd[i], argv[3][0]);
72         exit(0);
73     }
74     idx += bytes_to_read;
75 }
76 int status = 0;
77 for(int i = 0; i < P; i++) wait(&status);
78 if(p > 0) {
79     parent(fdw, pfd, argv[3][0], argv[1]);
80 }
81 close(fdr);
82 close(fdw);
83 for(int i = 0; i < P; ++i) {
84     free(*(pfd + i));
85 }
86 free(pfd);
87 }
88
89 void argument_handling(int argc, char **argv) {
90     if(argc != 4) {
91         perror("There should be three arguments!! (source file,
            destination file and character)\n");
92         exit(1);
93     }
94
95     if(strlen(argv[3]) > 1) {
96         perror("You can search for specific characters, not entire
            strings!!\n");
97         exit(1);
98     }
99 }
100
101 void print(int fdw, char *buff) {
102     size_t idx = 0, len = strlen(buff);

```

```

103 ssize_t wcnt;
104 do{
105     wcnt = write(fdw, buff+idx, len-idx);
106     if(wcnt == -1){
107         perror("write\n");
108         exit(1);
109     }
110     idx += wcnt;
111 }while(idx < len);
112 }
113
114 int min(int a, int b) {
115     return (a < b ? a : b);
116 }
117
118 void child(int fdr, off_t start, size_t bytes_to_read, int *pfd,
119     char c2c) { //[start, start+bytes_to_read-1]
120     ssize_t rcnt = 0;
121     int count = 0;
122     char buff[1024];
123     close(pfd[0]);
124     while(bytes_to_read) {
125         //start += rcnt;
126         //lseek(fdr, start, SEEK_SET);
127         //usleep(5000); //!THIS CAUSES AN ERROR DUE TO SHARED FILE
128         //POINTER BETWEEN THE CHILDREN
129         rcnt = read(fdr, buff, min(bytes_to_read, sizeof(buff) - 1));
130         if(rcnt == 0) break; //end of file
131         if(rcnt == -1){
132             print(STD_ERR, "Failed to read file\n");
133             exit(1);
134         }
135         buff[rcnt] = '\0';
136         bytes_to_read -= rcnt;
137         for(size_t i = 0; i < rcnt; i++){
138             if(buff[i] == c2c) count++;
139         }
140     }
141     //printf("Child starting at %ld read %d '%d's\n", start, count,
142     //c2c);
143     write(pfd[1], &count, sizeof(count));
144     close(pfd[1]);
145 }
146
147 void parent(int fdw, int **pfd, char c2c, char *file_to_write) {
148     int count = 0, cnt = 0;
149     for(int i = 0; i < P; i++){
150         close(pfd[i][1]);
151         read(pfd[i][0], &cnt, sizeof(cnt));
152         count += cnt;
153         close(pfd[i][0]);
154     }
155     char buff[1024];
156     snprintf(buff, sizeof(buff), "The character '%c' appears %d times
157         in file %s.\n", c2c, count, file_to_write);
158     print(fdw, buff);
159 }

```

```

156
157 void sigintHandler(int sig_num) {
158     if(signal(SIGINT, sigintHandler) < 0){
159         perror("Could not establish SIGINT handler");
160         exit(1);
161     }
162     sleep(1); //To be able to test the SIGINT handler
163     char buff[1024];
164     snprintf(buff, sizeof(buff), "There are %d processes reading the
        input file in parallel\n", P);
165     print(1, buff);
166 }

```

Ο κώδικας αποτελεί επέκταση αυτού της 2ης άσκησης, όμως εδώ αντί ένα παιδί να διαβάζει το αρχείο, πολλά (P) παιδιά διαβάζουν το αρχείο παράλληλα. Τα παιδιά επικοινωνούν το πλήθος εμφανίσεων του χαρακτήρα που μέτρησαν στον γονέα τους, ο οποίος είναι υπεύθυνος να αθροίσει τις εμφανίσεις του χαρακτήρα στα τμήματα του αρχείου που διάβασε το κάθε παιδί και να εγγράψει στο αρχείο εξόδου το συνολικό πλήθος εμφανίσεων του χαρακτήρα αναζήτησης στο αρχείο εισόδου.

Προσθέτουμε χειριστή του σήματος SIGINT που να τυπώνουν (όλες οι ενεργές διεργασίες) το πόσα παιδιά διαβάζουν το αρχείο. Βάζουμε sleep(1) μετά από κάθε εκτέλεση του χειριστή σήματος ώστε να μας δίνει το χρονικό περιθώριο να πατήσουμε το Ctrl+C πριν ολοκληρωθεί η εκτέλεση των διεργασιών. Εφόσον θέλουμε ο χειριστής να εκτελείται κάθε φορά που γίνεται Ctrl+C, πρέπει μέσα στη ρουτίνα εξυπηρέτησης της διακοπής να ξαναθέσουμε τον sigintHandler ως τον χειριστή του σήματος SIGINT.

Ανοίγουμε το αρχείο εισόδου από τον γονέα πριν δημιουργήσει το παιδί, και άρα τα ανοιχτά αρχεία του παραμένουν ανοιχτά και για το παιδί. Μέσω του struct stat λαμβάνουμε το μέγεθος N του αρχείου σε bytes και χωρίζουμε βέλτιστα το αρχείο σε batches στα P παιδιά. Το P εδώ επιλέγεται να ισούται με τη σταθερά του προγράμματος CPUCORES = 8 για να διευκολύνεται η παραλληλη επεξεργασία. Το μέγεθος του batch είναι  $\lfloor \frac{N}{P} \rfloor + 1$  για  $0 \leq i < N - (P - 1)\lfloor \frac{N}{P} \rfloor$  και  $\lfloor \frac{N}{P} \rfloor$  για  $N - (P - 1)\lfloor \frac{N}{P} \rfloor \leq i < P$ . Με αυτόν τον τρόπο, ελαχιστοποιούμε το μέγιστο μέγεθος batch για δεδομένο P, οπότε και τον μέγιστο χρόνο εκτέλεσης ενός παιδιού, με αποτέλεσμα να ελαχιστοποιείται και ο συνολικός χρόνος εκτέλεσης του αλγορίθμου.

Για να γλιτώσουμε τον χρόνο ανοίγματος του αρχείου από το κάθε παιδί, ανοίγουμε το αρχείο μία φορά στον γονέα πριν δημιουργηθούν τα παιδιά, όπως εξηγήσαμε και πριν. Εκτός από τα ανοιχτά αρχεία, τα παιδιά μοιράζονται έτσι και τον δείκτη του πού βρίσκονται στο κάθε ανοιχτό αρχείο. Επιπλέον, παρότι τα παιδιά μπορούν να εκτελούνται παράλληλα, ο διάδρομος δεδομένων από και προς τον δίσκο είναι κοινός για όλες τις διεργασίες, δηλαδή τα read εκτελούνται στην πραγματικότητα σειριακά. Ακόμα, επειδή η κλήση συστήματος read είναι blocking, δεν υπάρχουν προβλήματα συγχρονισμού. Συμπερασματικά, μπορούμε να αναθέσουμε σε κάθε

διεργασία να διαβάσει συγκεκριμένο πλήθος bytes ανάλογα με το batch size που της αναλογεί, χωρίς απαραίτητα αυτά να είναι ένα συνεχόμενο τμήμα του αρχείου που να έχουμε χωρίσει εξαρχής, καλώντας απλά τη read από το κάθε παιδί.

Τέλος, η επικοινωνία από τα παιδιά προς τον γονέα για την μεταφορά της πληροφορίας του πλήθους εμφανίσεων του χαρακτήρα αναζήτησης στο αρχείο εισόδου γίνεται μέσω σωληνώσεων (pipes). Ο γονέας διατηρεί ένα πίνακα P θέσεων που στην κάθε θέση έχει έναν πίνακα 2 θέσεων που αντιστοιχεί στο pipe του με το αντίστοιχο παιδί. Αφού κάνουμε pipe τον κάθε πίνακα 2 θέσεων με την κλήση συστήματος pipe (και ελέγχοντας για σωστή εκτέλεση αφού δεν επιστρέφεται αρνητικός αριθμός), μετά το fork() κλείνουμε το άκρο του που δεν χρησιμοποιείται από την κάθε διεργασία. Για κάθε pipe, στη θέση 1 (άκρο εγγραφής) γράφει το παιδί και από τη θέση 0 (άκρο ανάγνωσης) διαβάζει ο γονέας. Αφού ολοκληρωθεί η επικοινωνία του γονέα με ένα παιδί, και οι 2 διεργασίες κλείνουν τα εναπομείναντα ανοιχτά άκρα του pipe τους. Το τελικό αποτέλεσμα το συγκεντρώνει ο γονέας και το γράφει στο αρχείο εξόδου.

## 4 Εφαρμογή παράλληλης καταμέτρησης χαρακτήρων

Ξεκινάμε πάλι παρουσιάζοντας τον κώδικα της άσκησης. Παρόλα αυτά, επειδή χωρίζεται σε πολλά αρχεία και έχει μεγάλο μέγεθος, παραθέτουμε αρχικά μόνο τις main και τις δηλώσεις συναρτήσεων κάθε αρχείου. Ο πλήρης κώδικας μπορεί να βρεθεί στο παράρτημα στο τέλος της αναφοράς.

```
1 #include <unistd.h>
2 #include <string.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6 #include <sys/stat.h>
7 #include <sys/types.h>
8 #include <fcntl.h>
9 #include <signal.h>
10 #include <math.h>
11 #include <stdint.h>
12
13 #define STD_IN 0
14 #define STD_OUT 1
15 #define STD_ERR 2
16
17 #define MAX_WORKERS 64
18
19 int min(int a, int b); //returns the minimum of a and b
20 int max(int a, int b); //returns the maximum of a and b
21 void print(int fdw, char *buff); //prints the content of buff to
    the file descriptor fdw
22 void show_pstree(pid_t p); //show process tree
23 void scan(int fd, char *buff, unsigned int bytes_to_read); //scan
    bytes_to_read bytes from file descriptor fd to buff
24 void itoa(int num, char *str); //convert int num to char array str
```

```

1 #include "config.h"
2
3 void startup(); //show startup welcome message
4 void handle_frontend_input(int argc, char **argv); //assert that
   input is of correct format
5 void parse(char *buff, int *command_id, int *workers); //parse the
   user instruction
6 void open_dispatcher(int argc, char **argv, int *disp_pid, int *
   pipe_to_disp, int *pipe_from_disp); //create dispatcher process
7 void sighandler(int signum); //signal handler for SIGUSR1 and
   SIGINT
8
9 int disp_pid, pipe_to_disp, pipe_from_disp;
10
11 /*
12  argc = 3;
13  argv = {
14      0: a1.4-frontend,
15      1: char* file_to_read,
16      2: char c2c
17  }
18  */
19
20 int main(int argc, char **argv) {
21     handle_frontend_input(argc, argv);
22
23     startup();
24
25     open_dispatcher(argc, argv, &disp_pid, &pipe_to_disp, &
   pipe_from_disp);
26
27     if(signal(SIGUSR1, sighandler) < 0) {
28         perror("Could not establish SIGUSR1 handler.\n");
29     }
30     while(1){
31         char buff[1024];
32         if(fgets(buff, sizeof(buff), stdin) == NULL){
33             printf("Input terminated.\n");
34             break;
35         }
36         int command_id;
37         int workers;
38         parse(buff, &command_id, &workers);
39         char valid = 1; //is actually bool
40         switch (command_id){
41             case 0:
42                 write(pipe_to_disp, &command_id, sizeof(command_id));
43                 printf("Adding %d workers...\n", workers);
44                 write(pipe_to_disp, &workers, sizeof(workers));
45                 break;
46             case 1:
47                 write(pipe_to_disp, &command_id, sizeof(command_id));
48                 printf("Removing %d workers...\n", workers);
49                 write(pipe_to_disp, &workers, sizeof(workers));
50                 break;
51             case 2:
52                 write(pipe_to_disp, &command_id, sizeof(command_id));

```

```

53         printf("Displaying info...\n");
54         break;
55     case 3:
56         write(pipe_to_disp, &command_id, sizeof(command_id));
57         printf("Displaying progress...\n");
58         break;
59     default:
60         valid = 0;
61         printf("Please enter valid instruction.\n");
62         break;
63     }
64     if(valid && (kill(disp_pid, SIGUSR1) < 0)) {
65         print(STD_ERR, "Error while sending the signal to the
dispatcher\n");
66     }
67 }
68 }

```

Listing 1: Main body of frontend

```

1  #include "config.h"
2
3  struct worker_node {
4      pid_t pid;
5      int pipe_from_worker, pipe_to_worker;
6      int start, bytes_to_read;
7      char removed; //is actually bool
8
9      struct worker_node *next, *prev;
10 };
11
12 typedef struct worker_node worker;
13
14 struct workpool_node {
15     off_t start;
16     int size;
17     int left;
18     struct workpool_node *next;
19 };
20
21 typedef struct workpool_node work;
22
23 void handle_dispatcher_input(int argc, char **argv); //assert that
input is of correct format
24 void sighandler(int signum); //signal handler for SIGUSR1
25 void adding_workers(int workers); //execute add workers command and
add them to worker_list
26 void removing_workers(int workers); //execute remove workers
command and mark them as removed
27 void info(); //execute info command
28 void progress(); //execute progress command
29 void create_worker(char *file_to_read, char c2c, worker* w); //
create worker process
30 void segment_workpool(worker *w, int bytes_read); //create new work
item in case of abnormal termination of worker
31 void remove_from_worklist(worker *w); //remove worker from
worker_list

```

```

32 void delete_worker(worker *w, int bytes_read); //abnormal death of
    worker: segment workpool and remove from worklist
33
34 int P = 0;
35 int fdr;
36 int pipe_from_front, pipe_to_front;
37 char c2c;
38 pid_t front_pid;
39 off_t size;
40 worker *worker_list;
41 worker *worker_tail;
42 work *workpool;
43 work *workpool_tail;
44 int count = 0;
45
46 /*
47 argc = 5;
48 argv = {
49     0: "a1.4-dispatcher.c",
50     1: char* file_to_read,
51     2: int pipe_from_front,
52     3: int pipe_to_front,
53     4: char c2c
54 };
55 */
56
57 int main(int argc, char **argv) {
58     handle_dispatcher_input(argc, argv);
59     char *file_to_read = (char *) malloc(sizeof(argv[1]));
60     strcpy(file_to_read, argv[1]);
61     pipe_from_front = atoi(argv[2]);
62     pipe_to_front = atoi(argv[3]);
63     c2c = argv[4][0];
64
65     front_pid = getppid();
66
67     if(signal(SIGUSR1, sighandler) < 0) {
68         perror("Could not establish SIGUSR1 handler.\n");
69     }
70
71     int fdr;
72     if((fdr = open(file_to_read, O_RDONLY)) == -1){
73         print(STD_ERR, "Problem opening file to read\n");
74         exit(1);
75     }
76     struct stat st;
77     if(fstat(fdr, &st) < 0){
78         perror("Stat error\n");
79         exit(1);
80     }
81     size = st.st_size;
82     off_t batch_size = max(1, sqrt(size));
83     workpool = (work *) malloc(sizeof(work));
84     workpool->start = 0;
85     workpool->size = size;
86     workpool->left = size;
87     workpool->next = NULL;

```

```

88     workpool_tail = workpool;
89     off_t bytes_left = size;
90     count = 0;
91     while(bytes_left > 0){
92         for(work *wp = workpool; wp != NULL; wp = wp->next) {
93             while(wp->left > 0){
94                 for(worker *w = worker_list; w != NULL; w = w->next
95             ){
96                 int status;
97                 if(w->pid != -2 && w->bytes_to_read != 0){ //
98                 the worker was running before
99                     waitpid(w->pid, &status, WUNTRACED);
100                     size_t bytes_to_read_of_w = 0;
101                     int cnt;
102                     read(w->pipe_from_worker, &
103                     bytes_to_read_of_w, sizeof(bytes_to_read_of_w));
104                     read(w->pipe_from_worker, &cnt, sizeof(cnt)
105                 );
106                     size_t bytes_read = w->bytes_to_read -
107                     bytes_to_read_of_w;
108                     bytes_left -= bytes_read;
109                     count += cnt;
110                     wp->left -= w->bytes_to_read;
111                     if(bytes_to_read_of_w != 0){//Worker exited
112                     abnormally while potentially having read some bytes
113                         P--;
114                         delete_worker(w, bytes_read);
115                         printf("Worker deleted! Bytes read = %
116                     ld. New P = %d\n", bytes_read, P);
117                         continue;
118                     }
119                     close(w->pipe_from_worker);
120                     close(w->pipe_to_worker);
121                 }
122                 w->bytes_to_read = 0;
123                 if(w->removed == 1){
124                     remove_from_worklist(w);
125                 }
126                 else if(wp->size > 0){
127                     w->start = wp->start;
128                     w->bytes_to_read = min(wp->size, batch_size
129                 );
130
131                 if(bytes_left != 0) {
132                     create_worker(file_to_read, c2c, w);
133                     wp->start += w->bytes_to_read;
134                     wp->size -= w->bytes_to_read;
135                 }
136                 else{
137                     break;
138                 }
139             }
140         }
141     }
142 }
143 }
144 }
145 for(worker *w = worker_list; w != NULL; w = w->next){
146     int status;

```



```

137     if(w->pid != -2) {
138         waitpid(w->pid, &status, WUNTRACED);
139     }
140     remove_from_worklist(w);
141     P--;
142 }
143 while(1){
144     sleep(5);
145 };
146 }

```

Listing 2: Main body of dispatcher

```

1  #include "config.h"
2
3  void handle_worker_input(int argc, char **argv); //assert that
   input is of correct format
4  void sighandler(int signum); //signal handler for SIGUSR1 and
   SIGKILL
5
6  int pipe_from_disp, pipe_to_disp;
7
8  /*
9  argc = 7;
10 argv =
11 { 0: "a1.4worker",
12   1: off_t start,
13   2: size_t bytes_to_read,
14   3: int pipe_from_disp,
15   4: int pipe_to_disp,
16   5: char c2c
17   6: char* file_to_read,
18 };
19 */
20
21 size_t bytes_to_read;
22 int pipe_to_disp;
23 int count;
24
25 int main(int argc, char **argv) {
26     handle_worker_input(argc, argv);
27     if(signal(SIGTERM, sighandler) < 0){
28         perror("Could not establish SIGTERM handler.\n");
29     }
30     char *file_to_read = (char *) malloc(sizeof(argv[1]));
31     off_t start = (off_t) atoi(argv[1]);
32     bytes_to_read = (size_t) atoi(argv[2]);
33     int pipe_from_disp = atoi(argv[3]);
34     pipe_to_disp = atoi(argv[4]);
35     char c2c = argv[5][0];
36     strcpy(file_to_read, argv[6]);
37     int fdr;
38     if((fdr = open(file_to_read, O_RDONLY)) == -1){
39         print(STD_ERR, "Problem opening file to read\n");
40         exit(1);
41     }
42     ssize_t rcnt = 0;
43     count = 0;

```

```

44     char buff[1024];
45     while(bytes_to_read) {
46         start += rcnt;
47         lseek(fdr, start, SEEK_SET);
48         rcnt = read(fdr, buff, min(bytes_to_read, sizeof(buff) - 1));
49         if(rcnt == 0) break; //end of file
50         if(rcnt == -1) {
51             print(STD_ERR, "Failed to read file\n");
52             exit(1);
53         }
54         buff[rcnt] = '\0';
55         bytes_to_read -= rcnt;
56         for(size_t i = 0; i < rcnt; i++){
57             if(buff[i] == c2c) count++;
58         }
59     }
60     write(pipe_to_disp, &bytes_to_read, sizeof(bytes_to_read));
61     write(pipe_to_disp, &count, sizeof(count));
62     close(pipe_from_disp);
63     close(pipe_to_disp);
64     exit(0);
65 }

```

Listing 3: Main body of worker

```

./a1.4-frontend giant_input.txt a
+ exercise1 gtl:(main) X ./a1.4-frontend giant_input.txt a

graph LR
    main[main] --> worker1[worker]
    main --> worker2[worker]
    main --> worker3[worker]
    main --> worker4[worker]
    main --> worker5[worker]
    main --> worker6[worker]
    main --> worker7[worker]
    main --> worker8[worker]
    main --> worker9[worker]
    main --> worker10[worker]

We are going to count the number of occurrences of your favorite character for you!
Here are your options!

To add workers in your service just type: "Add: <number of workers to add>"
To remove workers from your service just type: "Remove: <number of workers to remove>"
If you want to know more details about your workers just type: "Info"
To see the progress of the execution of the program just type: "Progress"

Add: 10
Adding 10 workers...
Progress
Displaying progress...
Progress: [#####] : 23.858799% : 238587990/1000000000
Found character 2291622 times!!!
Info
Displaying info...
Currently, there are 10 workers at your service!

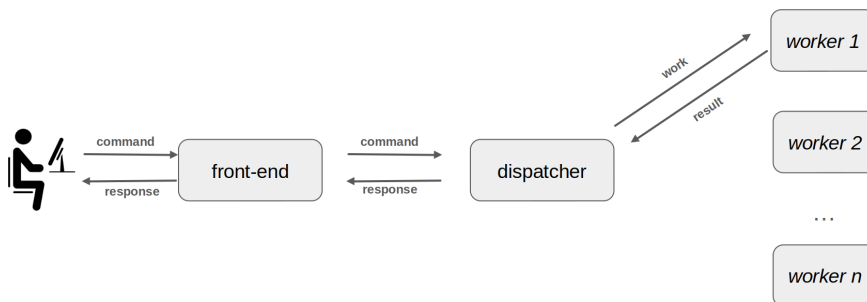
a1.4-dispatcher,228321 giant_input.txt 3 6 a
├─(a1.4-worker,241140)
├─(a1.4-worker,241141)
├─(a1.4-worker,241142)
├─(a1.4-worker,241143)
├─(a1.4-worker,241144)
├─(a1.4-worker,241145)
├─(a1.4-worker,241146)
├─(a1.4-worker,241147)
├─(a1.4-worker,241148)
├─(a1.4-worker,241149)
└─sh,241150 -c echo; echo; pstree -a -G -c -p 228321; echo; echo
    └─pstree,241151 -a -G -c -p 228321

Progress
Displaying progress...
Progress: [#####] : 66.197495% : 661974948/1000000000
Found character 6361856 times!!!
Progress
Displaying progress...
Progress: [#####] : 100.000000% : 1000000000/1000000000
Found character 9611168 times!!!

```

Figure 1: Παράδειγμα εκτέλεσης της εφαρμογής με πολύ μεγάλο αρχείο εισόδου

Η εφαρμογή χωρίζεται σε 3 δομικά στοιχεία: το frontend, τον dispatcher και τους workers.



Σε κάθε στοιχείο, ελέγχουμε ότι όλα τα ορίσματα είναι σωστά και πως σε όλες τις εντολές που εκτελούμε υπάρχει έλεγχος για μη κανονική εκτέλεση με τα κατάλληλα error messages να εμφανίζονται στον χρήστη. Η επικοινωνία μεταξύ δομικών στοιχείων διαφορετικών επιπέδων γίνεται μέσω pipes (2 μονόδρομα για αμφίδρομη επικοινωνία) που γίνονται cloned από τον parent στο παιδί/στα παιδιά του. Φροντίζουμε παντού να κλείνουμε τα άκρα των pipes που δεν χρησιμοποιούνται ή όταν σταματήσουν να χρησιμοποιούνται.

## 4.1 Frontend

Το frontend λειτουργεί ως διεπαφή του χρήστη με την εφαρμογή. Αρχικά τυπώνει το welcome message με οδηγίες χρήσης της εφαρμογής. Δέχεται 4 εντολές:

- Add: <number of workers to add>
- Remove: <number of workers to remove>
- Info
- Progress

Η εντολή Add: x προσθέτει x workers στην αναζήτηση (αν δεν υπερβαίνουν τον μέγιστο αριθμό workers), ενώ η Remove: x αφαιρεί x workers από την αναζήτηση (αν υπάρχουν τουλάχιστον x workers στην αναζήτηση). Η Info εκτελεί την εντολή pstree όπως υπήρχε στις διαφάνειες για να δείξει στον χρήστη τα PID's του dispatcher και των workers, όπως επίσης και το πλήθος των workers. Τέλος, η Progress εμφανίζει στον χρήστη ένα progress bar που δείνει πόσο ποσοστό του αρχείου έχει αναγνωστεί, ενώ εμφανίζει και σε νούμερα το ποσοστό και τον απόλυτο αριθμό των bytes που έχουν διαβαστεί και των συνολικών, όπως και το πλήθος των εμφανίσεων του χαρακτήρα προς αναζήτηση μέχρι στιγμής.

Αφού ενεργοποιηθεί ο sighandler μας για το σήμα SIGUSR1 στην αρχή της

εκτέλεσης του frontend για να επιτρέπεται η επικοινωνία με τον dispatcher, το frontend δημιουργεί τον dispatcher περνώντας του ως ορίσματα το όνομα του αρχείου προς ανάγνωση, τον χαρακτήρα προς αναζήτηση και τα pipes για επικοινωνία μεταξύ τους.

Έστερα, διαβάζει επαναληπτικά τις εντολές (σε γραμμές) από τον χρήστη. Το frontend αφού κάνει parse την εντολή του χρήστη και ελέγξει ότι έχει το σωστό format, τη στέλνει στο dispatcher (μέσω των pipes και την ασύγχρονη αποστολή σήματος SIGUSR1), ο οποίος αναλαμβάνει να την εκτελέσει, αν αυτό είναι επιτρεπτό. Στη συνέχεια, λαμβάνει πληροφορία από τον dispatcher σχετικά με την εκτέλεση της εντολής (διακοπή από σήμα SIGUSR1 και η πληροφορία σε pipe) και επιστρέφει το αποτέλεσμα της εκτέλεσης πίσω στον χρήστη.

## 4.2 Dispatcher

Ο dispatcher λειτουργεί ως συντονιστής του έργου της αναζήτησης. Επικοινωνεί με το frontend για να εκτελέσει τις εντολές του χρήστη, δημιουργεί διεργασίες παιδιά workers που θα εκτελούν την αναζήτηση, ενώ φροντίζει ώστε να διατηρείται η ακεραιότητα της εφαρμογής ανεξαρτήτως εξωτερικών παρεμβολών.

Αρχικά, ανοίγει το αρχείο προς αναζήτηση για να λάβει το μέγεθός του, size, σε bytes. Αυτό χρησιμοποιείται για τον υπολογισμό του batch size ως  $\max(1, \sqrt{\text{size}})$ , μια σχεδιαστική επιλογή που έγινε για να είναι σχετικά μικρό το batch size σε σχέση με το μέγεθος του αρχείου. Οι εντολές του χρήστη λαμβάνονται ασύγχρονα με σήματα SIGUSR1 από το frontend. Ο dispatcher διατηρεί δύο διπλά συνδεδεμένες λίστες, 1 για το workers list και 1 για το workpool.

Αρχικά, το workpool έχει ένα work item, το συνολικό αρχείο, με bytes to read ίσα με το μέγεθος του αρχείου, και start = 0. Νέα work items φτιάχνονται όταν ένας worker πεθαίνει βίαια, οπότε τότε λαμβάνουμε την πληροφορία του μέχρι πού είχε διαβάσει και φτιάχνουμε νέο work item για το υπόλοιπο τμήμα που του είχε ανατεθεί.

## 4.3 Worker

Ο κώδικας του worker μοιάζει πολύ με εκείνο της άσκησης 3 (διαδικασιακή επικοινωνία). Ουσιαστικά, όπως όλα τα προηγούμενα τμήματα της εφαρμογής, ελέγχει τα ορίσματα που του δίνει ο dispatcher, ενώ κατόπιν ακολουθεί κομμάτι κώδικα με την συνάρτηση signal() το οποίο είναι υπεύθυνο για την διαχείριση των σημάτων. Συγκεκριμένα, μόλις λάβει ένα σήμα παραδίδει την ροή της εκτέλεσης στην συνάρτηση sighandler(), η οποία κάνει το εξής: Εφόσον το σήμα είναι SIGTERM εκτυπώνει μήνυμα στο STD\_ERR, ενώ αν είναι SIGUSR1 καταλαμβάνει πως πρόκειται για μήνυμα επικοινωνίας με τα άλλα κομμάτια της εφαρμογής. Έτσι, διαβάζει το pipe της και ανάλογα με το περιεχόμενό του εκτελεί την κατάλληλη πράξη. Στην συνέχεια, κάθε εργάτης ανοίγει το αρχείο που επιθυμεί ο χρήστης (και του έχει γνωστοποιηθεί από τον dispatcher) και παίρνει

έναν file descriptor σε αυτό. Στη συνέχεια, αρχίζει να διαβάζει το αρχείο από την θέση την οποία του έχει γνωστοποιήσει ο dispatcher (start) και για όσα byte του έχει αναθέσει επίσης ο dispatcher (bytes\_to\_read). Η έναρξη από το σημείο υπόδειξης του dispatcher επιτυγχάνεται με την βοήθεια της συνάρτησης lseek. Ο τρόπος ανάγνωσης του αρχείου είναι ίδιος με εκείνον σε όλες τις προηγούμενες ασκήσεις της εργασίας και περιγράφεται στις συνοδευτικές διαφάνειες. Μόλις ο εργάτης τελειώσει την ανάγνωση που του έχει ανατεθεί, γράφει πόσες φορές έχει βρει τον ζητούμενο χαρακτήρα στο δικό του άκρο του pipe μεταξύ αυτού και του dispatcher. Τέλος, κλείνει το pipe του και τερματίζεται.

**ΣΗΜΕΙΩΣΗ:** Αν ένας εργάτης "πεθάνει βίαια", ο sighandler κάνει το εξής: Πριν τερματίσει την εκτέλεσή του στέλνει στον dispatcher πόσες φορές βρήκε τον χαρακτήρα που του ζητήθηκε για όσο κομμάτι της δουλειάς του μπόρεσε να εκτελέσει, καθώς και πόσα byte πρόλαβε να διαβάσει, έτσι ώστε ο dispatcher να μπορέσει αφενός να καταγράψει τον αριθμό των φορών που βρέθηκε ο χαρακτήρας και αφετέρου να ανακατανείμει την δουλειά που δεν μπόρεσε να ολοκληρωθεί (εναπομείναντα αδιάβαστα byte) στους υπόλοιπους εργάτες.