OSLab 2023-24 — 1η Εργαστηριακή Άσκηση

Κόρδας Νικόλαος - A.M.: 03121032
Κριθαρίδης Κωνσταντίνος - A.M.: 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 αναζητήσει 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() διαβάζει 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 προγραμμά 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:

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

void argument_handling(int argc, char **argv);

int main(int argc, char **argv) {
  argument_handling(argc, argv);
  int fdr, fdw, oflags, mode;
  char c2c = 'a';
  int count = 0;
  oflags = O_CREAT | O_WRONLY | O_TRUNC;
  mode = S_IRUSR | S_IWUSR;
  if((fdr = open(argv[1], O_RDONLY)) == -1) {
    perror("Problem opening file to read\n");
    exit(1);
  }
  if((fdw = open(argv[2], oflags, mode)) == -1) {
    perror("Problem opening file to write\n");
    exit(1);
  }

  ssize_t rcnt;
  char buff[1024];
    c2c = argv[3][0];

  while(1) {
    rcnt = read(fdr, buff, sizeof(buff) - 1);
      if(rcnt == 0) break; //end of file
      if(rcnt == -1) {
      perror("Failed to read file\n");
      exit(1);
```

```
    }
    buff[rcnt] = '\0';
    for(size_t i = 0; i < rcnt; i++){
      if(buff[i] == c2c) count++;
    }
  }
  close(fdr);
  ssize_t wcnt;
  snprintf(buff, sizeof(buff), "The character '%c' appears %d times
     in file %s.\n", c2c, count, argv[1]);
  size_t len = strlen(buff), idx = 0;
  do {
    wcnt = write(fdw, buff+idx, len-idx);
    if(wcnt == -1){
      perror("Failed to write to file\n");
      exit(1);
    }
    idx += wcnt;
  } while(idx < len);
  close(fdw);
  return 0;
}

void argument_handling(int argc, char **argv) {
  if(argc != 4) {
    perror("There should be three arguments!! (source file,
    destination file and character)\n");
    exit(1);
  }

  if(strlen(argv[3]) > 1) {
    perror("You can search for specific characters, not entire
    strings!!\n");
    exit(1);
  }
}
```

2 Δημιουργία διεργασιών

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

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

void argument_handling(int argc, char **argv);
void print(int fdw, char *buff);
```

```c
void child(int *x, int fdr, int fdw, char c2c, char *file_to_write)
    ;
void parent(pid_t p, int *x);

int main(int argc, char **argv) {
  argument_handling(argc, argv);
  int fdr, fdw;
  int x = 17;
  int oflags, mode;
  oflags = O_CREAT | O_WRONLY | O_TRUNC;
  mode = S_IRUSR | S_IWUSR;
  if((fdr = open(argv[1], O_RDONLY)) == -1){
    perror("Problem opening file to read\n");
    exit(1);
  }
  if((fdw = open(argv[2], oflags, mode)) == -1){
    perror("Problem opening file to write\n");
    exit(1);
  }

  pid_t p;
  p = fork();
  if(p < 0){
    perror("fork");
    exit(1);
  }
  else if(p == 0){
    child(&x, fdr, fdw, argv[3][0], argv[1]);
    exit(0);
  }
  else{
    parent(p, &x);
  }

  close(fdr);
  close(fdw);

  p = fork();
  if(p < 0){
    perror("fork");
    exit(1);
  }
  else if(p == 0){
    char *argv2[] = {"./a1.1-C\0", "", "", "", NULL};
    for(int i = 1; i <= 3; i++){
      argv2[i] = (char *)malloc(sizeof(argv[i]));
      strcpy(argv2[i], argv[i]);
    }
    execv(argv2[0], argv2);
    exit(0);
  }
  else{
    int status;
    wait(&status);
    char buff[1024];
    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,
        destination file and character)\n");
75          exit(1);
76      }
77
78      if(strlen(argv[3]) > 1) {
79          perror("You can search for specific characters, not entire
        strings!!\n");
80          exit(1);
81      }
82  }
83
84  void print(int fdw, char *buff) {
85    size_t idx = 0, len = strlen(buff);
86    ssize_t wcnt;
87    do{
88      wcnt = write(fdw, buff+idx, len-idx);
89      if(wcnt == -1){
90        perror("write\n");
91        exit(1);
92      }
93      idx += wcnt;
94    }while(idx < len);
95  }
96
97  void child(int *x, int fdr, int fdw, char c2c, char *file_to_write)
        {
98    pid_t mypid = getpid(), parpid = getppid();
99    char buff[1024];
100   snprintf(buff, sizeof(buff), "Hello World!\nMy pid is %d.\nMy
        parent's pid is %d.\n", mypid, parpid);
101   print(1, buff);
102
103   *x = 42;
104   snprintf(buff, sizeof(buff), "Variable x is %d (child).\n", *x);
105   print(1, buff);
106
107   ssize_t rcnt;
108   int count = 0;
109   while(1) {
110     rcnt = read(fdr, buff, sizeof(buff)-1);
111       if(rcnt == 0) break; //end of file
112       if(rcnt == -1){
113       perror("Failed to read file\n");
114       exit(1);
115     }
116     buff[rcnt] = '\0';
117     for(size_t i = 0; i < rcnt; i++){
118       if(buff[i] == c2c) count++;
119     }
```

```
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 παιδού 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 αποτελέσματός 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 επιτυγνάνουμε 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.

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

#define CPUCORES 8
#define STD_ERR 2

int P = 0;

void argument_handling(int argc, char **argv);
void print(int fdw, char *buff);
int min(int a, int b);
void child(int fdr, off_t start, size_t bytes_to_read, int *pfd,
    char c2c);
void parent(int fdw, int **pfd, char c2c, char *file_to_write);
void sigintHandler(int sig_num);

int main(int argc, char **argv) {
  argument_handling(argc, argv);
  int fdr, fdw;
  int oflags, mode;
  oflags = O_CREAT | O_WRONLY | O_TRUNC;
  mode = S_IRUSR | S_IWUSR;
  if((fdr = open(argv[1], O_RDONLY)) == -1){
    perror("Problem opening file to read\n");
    exit(1);
  }
  if((fdw = open(argv[2], oflags, mode)) == -1){
    perror("Problem opening file to write\n");
    exit(1);
  }
  struct stat st;
  if(stat(argv[1], &st) < 0){
    perror("Stat error\n");
    exit(1);
  }
  if(signal(SIGINT, sigintHandler) < 0){
    perror("Could not establish SIGINT handler");
    exit(1);
  }
  printf("Initiated SIGINT handler\n");
  sleep(1); //To be able to test the SIGINT handler
```

```
  off_t size = st.st_size;
  P = CPUCORES;
  off_t batch_size = size / P;
  //printf("File size = %ld, batch size = %ld, P = %d\n", size,
    batch_size, P);
  off_t idx = 0, extra_left = size - P * batch_size;
  int **pfd = (int**)malloc(P * sizeof(int *));
  for(int i = 0; i < P; ++i) {
    pfd[i] = (int *)malloc(2 * sizeof(int));
  }
  pid_t p;
  for(int i = 0; i < P; i++) {
    size_t bytes_to_read = batch_size + (i < extra_left);
    if(pipe(pfd[i]) < 0){
      perror("pipe\n");
      exit(1);
    }
    p = fork();
    if(p < 0) {
      perror("fork\n");
      exit(1);
    }
    else if(p == 0) {
      child(fdr, idx, bytes_to_read, pfd[i], argv[3][0]);
      exit(0);
    }
    idx += bytes_to_read;
  }
  int status = 0;
  for(int i = 0; i < P; i++) wait(&status);
  if(p > 0) {
    parent(fdw, pfd, argv[3][0], argv[1]);
  }
  close(fdr);
  close(fdw);
  for(int i = 0; i < P; ++i) {
    free(*(pfd + i));
  }
  free(pfd);
}

void argument_handling(int argc, char **argv) {
    if(argc != 4) {
        perror("There should be three arguments!! (source file,
    destination file and character)\n");
        exit(1);
    }

    if(strlen(argv[3]) > 1) {
        perror("You can search for specific characters, not entire
    strings!!\n");
        exit(1);
    }
}

void print(int fdw, char *buff) {
  size_t idx = 0, len = strlen(buff);
```

```
  ssize_t wcnt;
  do{
    wcnt = write(fdw, buff+idx, len-idx);
    if(wcnt == -1){
      perror("write\n");
      exit(1);
    }
    idx += wcnt;
  }while(idx < len);
}

int min(int a, int b) {
  return (a < b ? a : b);
}

void child(int fdr, off_t start, size_t bytes_to_read, int *pfd,
    char c2c) { //[start, start+bytes_to_read-1]
  ssize_t rcnt = 0;
  int count = 0;
  char buff[1024];
  close(pfd[0]);
  while(bytes_to_read) {
    //start += rcnt;
    //lseek(fdr, start, SEEK_SET);
    //usleep(5000); //!THIS CAUSES AN ERROR DUE TO SHARED FILE
    POINTER BETWEEN THE CHILDREN
    rcnt = read(fdr, buff, min(bytes_to_read, sizeof(buff) - 1));
      if(rcnt == 0) break; //end of file
      if(rcnt == -1){
      print(STD_ERR, "Failed to read file\n");
      exit(1);
    }
    buff[rcnt] = '\0';
    bytes_to_read -= rcnt;
    for(size_t i = 0; i < rcnt; i++){
      if(buff[i] == c2c) count++;
    }
  }
  //printf("Child starting at %ld read %d '%d's\n", start, count,
    c2c);
  write(pfd[1], &count, sizeof(count));
  close(pfd[1]);
}

void parent(int fdw, int **pfd, char c2c, char *file_to_write) {
  int count = 0, cnt = 0;
  for(int i = 0; i < P; i++){
    close(pfd[i][1]);
    read(pfd[i][0], &cnt, sizeof(cnt));
    count += cnt;
    close(pfd[i][0]);
  }
  char buff[1024];
  snprintf(buff, sizeof(buff), "The character '%c' appears %d times
     in file %s.\n", c2c, count, file_to_write);
  print(fdw, buff);
}
```

```
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 παρουσίαζοντας 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
```

```c
#include "config.h"

void startup(); //show startup welcome message
void handle_frontend_input(int argc, char **argv); //assert that input is of correct format
void parse(char *buff, int *command_id, int *workers); //parse the user instruction
void open_dispatcher(int argc, char **argv, int *disp_pid, int *pipe_to_disp, int *pipe_from_disp); //create dispatcher process
void sighandler(int signum); //signal handler for SIGUSR1 and SIGINT

int disp_pid, pipe_to_disp, pipe_from_disp;

/*
argc = 3;
argv = {
    0: a1.4-frontend,
    1: char* file_to_read,
    2: char c2c
}
*/

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

    startup();

    open_dispatcher(argc, argv, &disp_pid, &pipe_to_disp, &pipe_from_disp);

  if(signal(SIGUSR1, sighandler) < 0) {
        perror("Could not establish SIGUSR1 handler.\n");
    }
    while(1){
        char buff[1024];
        if(fgets(buff, sizeof(buff), stdin) == NULL){
            printf("Input terminated.\n");
            break;
        }
        int command_id;
        int workers;
        parse(buff, &command_id, &workers);
        char valid = 1; //is actually bool
        switch (command_id){
        case 0:
            write(pipe_to_disp, &command_id, sizeof(command_id));
            printf("Adding %d workers...\n", workers);
            write(pipe_to_disp, &workers, sizeof(workers));
            break;
        case 1:
            write(pipe_to_disp, &command_id, sizeof(command_id));
            printf("Removing %d workers...\n", workers);
            write(pipe_to_disp, &workers, sizeof(workers));
            break;
        case 2:
            write(pipe_to_disp, &command_id, sizeof(command_id));
```

```
            printf("Displaying info...\n");
            break;
        case 3:
            write(pipe_to_disp, &command_id, sizeof(command_id));
            printf("Displaying progress...\n");
            break;
        default:
            valid = 0;
            printf("Please enter valid instruction.\n");
            break;
        }
        if(valid && (kill(disp_pid, SIGUSR1) < 0)) {
            print(STD_ERR, "Error while sending the signal to the dispatcher\n");
        }
    }
}
```

Listing 1: Main body of frontend

```
#include "config.h"

struct worker_node {
    pid_t pid;
    int pipe_from_worker, pipe_to_worker;
    int start, bytes_to_read;
    char removed; //is actually bool

    struct worker_node *next, *prev;
};

typedef struct worker_node worker;

struct workpool_node {
    off_t start;
    int size;
    int left;
    struct workpool_node *next;
};

typedef struct workpool_node work;

void handle_dispatcher_input(int argc, char **argv); //assert that input is of correct format
void sighandler(int signum); //signal handler for SIGUSR1
void adding_workers(int workers); //execute add workers command and add them to worker_list
void removing_workers(int workers); //execute remove workers command and mark them as removed
void info(); //execute info command
void progress(); //execute progress command
void create_worker(char *file_to_read, char c2c, worker* w); //create worker process
void segment_workpool(worker *w, int bytes_read); //create new work item in case of abnormal termination of worker
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;
```

```
    workpool_tail = workpool;
    off_t bytes_left = size;
    count = 0;
    while(bytes_left > 0){
        for(work *wp = workpool; wp != NULL; wp = wp->next) {
            while(wp->left > 0){
                for(worker *w = worker_list; w != NULL; w = w->next
){
                    int status;
                    if(w->pid != -2 && w->bytes_to_read != 0){ //
the worker was running before
                        waitpid(w->pid, &status, WUNTRACED);
                        size_t bytes_to_read_of_w = 0;
                        int cnt;
                        read(w->pipe_from_worker, &
bytes_to_read_of_w, sizeof(bytes_to_read_of_w));
                        read(w->pipe_from_worker, &cnt, sizeof(cnt)
);
                        size_t bytes_read = w->bytes_to_read -
bytes_to_read_of_w;
                        bytes_left -= bytes_read;
                        count += cnt;
                        wp->left -= w->bytes_to_read;
                        if(bytes_to_read_of_w != 0){//Worker exited
 abnormally while potentially having read some bytes
                            P--;
                            delete_worker(w, bytes_read);
                            printf("Worker deleted! Bytes read = %
ld. New P = %d\n", bytes_read, P);
                            continue;
                        }
                        close(w->pipe_from_worker);
                        close(w->pipe_to_worker);
                    }
                    w->bytes_to_read = 0;
                    if(w->removed == 1){
                        remove_from_worklist(w);
                    }
                    else if(wp->size > 0){
                        w->start = wp->start;
                        w->bytes_to_read = min(wp->size, batch_size
);
                        if(bytes_left != 0) {
                            create_worker(file_to_read, c2c, w);
                            wp->start += w->bytes_to_read;
                            wp->size -= w->bytes_to_read;
                        }
                        else{
                            break;
                        }
                    }
                }
            }
        }
    }
  for(worker *w = worker_list; w != NULL; w = w->next){
    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;
```

```
    char buff[1024];
    while(bytes_to_read) {
        start += rcnt;
        lseek(fdr, start, SEEK_SET);
    rcnt = read(fdr, buff, min(bytes_to_read, sizeof(buff) - 1));
      if(rcnt == 0) break; //end of file
      if(rcnt == -1) {
      print(STD_ERR, "Failed to read file\n");
      exit(1);
    }
        buff[rcnt] = '\0';
    bytes_to_read -= rcnt;
    for(size_t i = 0; i < rcnt; i++){
      if(buff[i] == c2c) count++;
    }
  }
    write(pipe_to_disp, &bytes_to_read, sizeof(bytes_to_read));
  write(pipe_to_disp, &count, sizeof(count));
    close(pipe_from_disp);
    close(pipe_to_disp);
    exit(0);
}
```

Listing 3: Main body of worker

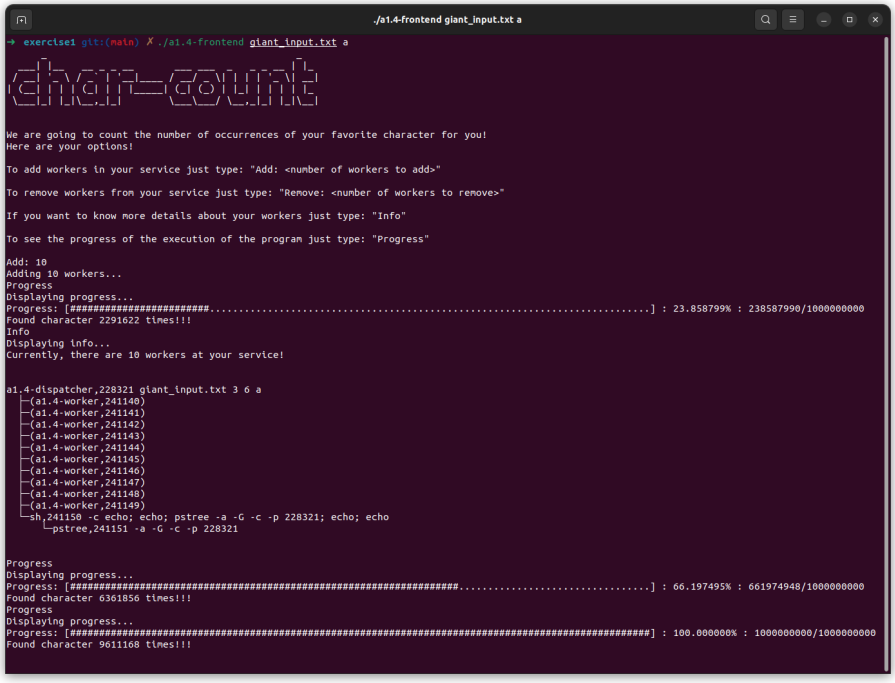
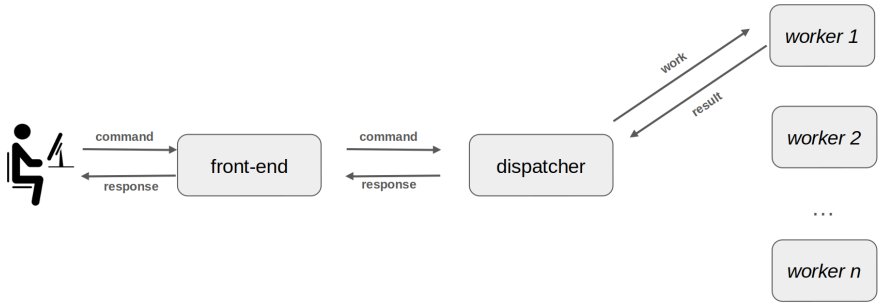


Figure 1: Παράδειγμα εκτέλεσης της εφαρμογής με πολύ μεγάλο αρχείο εισόδου

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.

Ύστερα, 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 που αναλαμβάνει ο κάθε worker 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. Σχετικά με το workers list, αυτό είναι αρχικά κενό, και προστίθενται ή αφαιρούνται workers ανάλογα με τις εντολές του χρήστη. Όταν το workers list γίνει μη κενό, ξεκινάει η αναζήτηση, μέχρι να ολοκληρωθεί το workpool.

Το κάθε work item χωρίζεται σε τμήματα μεγέθους batch size (ή λιγότερο αν είναι το τελευταίο batch στο work item) που ανατίθενται κυκλικά στους workers που βρίσκονται στο workers list. Στην αρχή, οι workers δημιουργούνται με execv μόλις έρθει η σειρά τους, περνώντας ως παραμέτρους το σημείο του αρχείου από όπου πρέπει να ξεκινήσουν την αναζήτηση, το πλήθος bytes προς ανάγνωση, τα pipes για επικοινωνία με τον dispatcher, τον χαρακτήρα προς αναζήτηση και το όνομα του αρχείου εισόδου. Όταν δημιουργούνται, αρχίζουν να διαβάζουν το αρχείο μέχρι είτε να διαβάσουν τα bytes που τους ανατέθηκαν, είτε να τερματίσουν βίαια.

Όταν ξαναέρθει η σειρά τους κυκλικά, ο dispatcher περιμένει τον καθένα (σε περίπτωση που δεν είχε τερματίσει ή είχε μείνει σε zombie state), διαβάζει (από το pipe) πόσα bytes του έμειναν από όσα του είχαν ανατεθεί και πόσες φορές βρήκε τον χαρακτήρα προς αναζήτηση και προσθέτει το πλήθος αυτό στον συνολικό αριθμό εμφανίσεων του χαρακτήρα. Αν έμειναν μη μηδενικά bytes για ανάγνωση από όσα είχαν ανατεθεί στον worker, φτιάχνουμε νέο work item με αρχή το σημείο όπου σταμάτησε (βίαια) την ανάγνωση ο worker και μέγεθος ίσο με το πλήθος bytes που του έμειναν. Αν μένουν και άλλοι χαρακτήρες προς ανάγνωση στο αρχείο, ο dispatcher τον ξαναδημιουργεί για να διαβάσει επόμενο τμήμα του αρχείου.

Η προσθήκη workers γίνεται με την προσθήκη στοιχείων στο workers list σημειωμένα ως previously not running (σημειώνοντας στο πεδίο pid την τιμή -2), που όταν έρθει η σειρά τους κυκλικά, ο dispatcher θα το ανιχνεύσει και αντί να περιμένει να τερματίσουν και να διαβάσει από το (πριν ανύπαρκτο) pipe τους, απλά τους δημιουργεί εκείνη τη στιγμή και τους αναθέτει τμήμα του αρχείου από το τωρινό work item. Η αφαίρεση workers γίνεται παρόμοια, σημειώνοντας τα ως removed (θέτοντας το πεδίο removed = 1) και όταν φτάσει η σειρά τους κυκλικά, ο dispatcher θα διαβάσει το αποτέλεσμα της εκτέλεσής τους και στη συνέχεια απλώς δεν θα τους ξαναδημιουργήσει.

4.3 Worker

Ο κώδικας του worker μοιάζει πολύ με εκείνο της άσκησης 3 (διαδιεργασιακή 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 ο worker στέλνει μήνυμα στον dispatcher την πρόοδό του πριν τερματίσει βίαια (βλ. ΣΗΜΕΙΩΣΗ), ενώ αν είναι SIGUSR1 καταλαβαίνει 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.