Λειτουργικά Συστήματα Υπολογιστών

1η Εργαστηριακή Άσκηση

oslab064	
Νικόλαος Γιαννόπουλος	03122086
Κωνσταντίνος Καργάκος	03122166

Πίνακας περιεχομένων

1 – Ανάγνωση και εγγραφή αρχείων στη C με κλήσεις συστήματος

```
if (argc != 4) {
    perror("Wrong number of arguments");
    exit(1);
}
```

Αρχικά ελέγχουμε εάν ο χρήστης έχει βάλει το σωστό αριθμό των arguments κατά την εκτέλεση του προγράμματος και με το **perror()** θέτουμε ένα μήνυμα σε περίπτωση σφάλματος.

```
// Open file for reading
int fdr = open(argv[1], O_RDONLY);
if (fdr == -1) {
    perror("Problem opening file to read");
    exit(1);
}

// Open file for writing the result
int oflags = O_WRONLY | O_CREAT | O_TRUNC;
int mode = S_IRUSR | S_IWUSR;
int fdw = open(argv[2], oflags, mode);
if (fdw == -1) {
    perror("Problem opening file to write");
    close(fdr);
    exit(1);
}
```

Με την κλήση συστήματος **read()** ανοίγουμε τα δύο αρχεία. Το ένα για ανάγνωση του κειμένου και το άλλο για να γράψουμε το αποτέλεσμα. Τον τρόπο που θα ανοίξουμε το κάθε αρχείο το ορίζουμε με τα διάφορα **flags** και **modes**. Για το πρώτο αρχείο βάζουμε **O_RDONLY** (άνοιγμα μόνο για ανάγνωση), ενώ για το δεύτερο αρχείο βάζουμε **O_WRONLY** (άνοιγμα για επεξεργασία), **O_CREAT** (δημιουργία αρχείου εάν αυτό δεν υπάρχει), **O_TRUNC** (διαγραφή περιεχομένου του αρχείου κατά το άνοιγμα), καθώς και **S_IRUSR** (δικαίωμα ανάγνωσης), **S_IWUSR** (δικαίωμα εγγραφής).

Η κάθε κλήση συστήματος **open()** επιστρέφει έναν **file descriptor**. Ελέγχουμε αν η τιμή του είναι -1, που σημαίνει ότι έχει γίνει κάποιο σφάλμα κατά την κλήση συστήματος.

```
// Count char
c2c = argv[3][0];
ssize_t rcnt;
for (;;) {
    rcnt = read(fdr, &cc, 1);
    if (rcnt == -1) {
        perror("Problem reading from file");
        close(fdw);
        close(fdr);
        exit(1);
    }
    if (rcnt == 0) break;  // EOF
```

```
    if (cc == c2c) count++;
}
close(fdr);
```

Με την κλήση συστήματος **read()** διαβάζουμε τους χαρακτήρες του αρχείου ανά byte. Έχουμε βάλει ένα loop, στο οποίο σε κάθε επανάληψη καλούμε την **read()**, η οποία διαβάζει έναν χαρακτήρα από τον **file descriptor fdr** και αποθηκεύει τον χαρακτήρα στη διεύθυνση **&cc**. Η **open()** επιστρέφει τον αριθμό των bytes που διάβασε. Εάν επιστρέψει -1 πρόκειται για σφάλμα, ενώ αν επιστρέψει 0 πρόκειται για **end of file**. Με αυτό τον τρόπο ελέγχουμε αν έχει γίνει κάποιο error ή αν έχουμε φτάσει στο τέλος του αρχείου, όπου και κάνουμε break το loop. Επίσης, σε κάθε επανάληψη ελέγχουμε αν ο χαρακτήρας που διάβασε **(cc)** είναι ο χαρακτήρας που ψάχνουμε **(c2c)** και τότε αυξάνουμε κατά 1 τον μετρητή **(count)**.

Τον αριθμό των bytes τον αποθηκεύουμε στη μεταβλητή **rcnt** τύπου **ssize_t**. Ο τύπος αυτός είναι ο τύπος δεδομένων που επιστρέφει η **read()** αλλά και **write()**.

Επιπλέον, κάθε φορά που έχουμε ανοίξει κάποιο αρχείο χρειάζεται να το κάνουμε **close()** πριν το πρόγραμμα τερματίσει. Αυτό ισχύει και στις περιπτώσεις όπου το πρόγραμμα τερματίζει λόγω κάποιου error. Έτσι, πριν το **exit(1)**, το οποίο τερματίζει το πρόγραμμα λόγω σφάλματος κάνουμε **close()** τα ανοικτά αρχεία. Παρόλο που το **exit()** κάνει **close()** από μόνο του τα ανοικτά αρχεία είναι καλή πρακτική να κάνουμε εμείς **close()** πριν το **exit()**.

```
// Write result in the output file
char buf[100];
size_t len, idx = 0;
ssize_t wcnt;
len = sprintf(buf, "The character '%c' appears %d times in file %s.\n",
    c2c, count, argv[1]);
do {
    wcnt = write(fdw, buf + idx, len - idx);
    if (wcnt == -1) {
        perror("Problem writing to file");
        close(fdw);
        exit(1);
    }
    idx += wcnt;
} while (idx < len);
close(fdw);
```

Αφού κάνουμε **close()** το αρχείο ανάγνωσης, γράφουμε το αποτέλεσμα από τη μεταβλητή count στο δεύτερο αρχείο. Χρησιμοποιούμε έναν **buffer**, στον οποίο αποθηκεύουμε όλο το κείμενο που θέλουμε να γράψουμε στο τελικό αρχείο με τη χρήση της **sprint()**. Έπειτα γράφουμε το κείμενο του **buffer** στο αρχείο με τη **write()**. Αυτό γίνεται μέσα σε ένα do-while loop το οποίο εξασφαλίζει ότι θα γραφεί όλο το κείμενο του **buffer** στο αρχείο. Εάν για κάποιο λόγο το **write()** δεν ολοκληρώσει τη διαδικασία, τότε συνεχίζει από το σημείο που σταμάτησε **(idx)** στην επόμενη επανάληψη.

Τέλος, κάνουμε **close()** το τελικό αρχείο και το πρόγραμμα τερματίζει.

*Μία διαφορετική υλοποίηση της ανάγνωσης των χαρακτήρων του αρχείου, που πιθανόν να είναι πιο γρήγορη, είναι η χρήση ενός **buffer** συγκεκριμένου μεγέθους. Σε κάθε επανάληψη, η **read()** θα διαβάζει μαζικά τόσα bytes όσα και το μέγεθος του **buffer** και θα τα αποθηκεύει σε αυτόν. Έπειτα, θα ελέγχουμε τους χαρακτήρες έναν έναν από τον **buffer**.*

2 – Δημιουργία διεργασιών

1 – Χρήση fork

```c
int main() {
    pid_t p, mypid, parentpid;
    int status;

    p = fork();
    if (p < 0) {
        perror("Problem creating child process");
        exit(1);
    }
    if (p == 0) {
        mypid = getpid();
        parentpid = getppid();
        printf("Hello from child process. My pid is %d and my parent's pid is %d\n",
            mypid, parentpid);
        exit(0);
    }

    printf("Child process created with pid %d\n", p);
    wait(&status);
    return 0;
}
```

Η κλήση συστήματος **fork()** δημιουργεί μια διεργασία-παιδί, η οποία κληρονομεί όλες τις μεταβλητές, τα **file descriptors** και τη μνήμη της διεργασίας-πατέρα, δηλαδή δημιουργεί αντίγραφα όλων αυτών των στοιχείων όταν χρειαστεί αλλάξουν από μία διεργασία **(copy-on-write)**. Η διεργασία-παιδί εκτελεί τον κώδικα αμέσως μετά το **fork()**. Το **fork()** επιστρέφει στη διεργασία-πατέρα το αναγνωριστικό **pid** της διεργασίας-παιδί, ενώ στη διεργασία-παιδί το 0. Αν υπάρξει κάποιο σφάλμα στη δημιουργία διεργασίας επιστρέφει το -1. Η τιμή που επιστρέφει το **fork()** αποθηκεύεται στη μεταβλητή **p** τύπου **pid_t**.

Για να εκτελέσει διαφορετικές εντολές η κάθε διεργασία αλλά και για τον έλεγχο σφάλματος ελέγχουμε την τιμή του **p**. Αν **p<0**, τότε έχουμε **error**. Αν **p=0** είναι η διεργασία-παιδί, ενώ αν **p>0** είναι η διεργασία-πατέρα.

Για **p=0** η διεργασία-παιδί εκτυπώνει το μήνυμα με το **pid** της και το **pid** της διεργασίας-πατέρα. Αυτές τις τιμές τις παίρνει από τα **getpid()** και **getppid()** αντίστοιχα.

Για **p>0** η διεργασία-πατέρας εκτυπώνει το μήνυμα με το **pid** της διεργασίας-παιδί που δημιούργησε. Το **pid** αυτό το έχει στη μεταβλητή **p** κατά την εκτέλεση του **fork()**. Στη συνέχεια η διεργασία πατέρας περιμένει τη διεργασία-παιδί να ολοκληρώσει (να κάνει **exit()**) με την εντολή **wait()**. Επίσης, ορίζουμε τη μεταβλητή **status** ή οποία περιλαμβάνει πληροφορίες για τον τερματισμό της διεργασίας-παιδί, τις οποίες δεν αξιοποιούμε σε αυτή την άσκηση.

```
Child process created with pid 20953
Hello from child process. My pid is 20953 and my parent's pid is 20952
```

2 – Μεταβλητή με fork

Πριν τη δημιουργία νέας διεργασίας ορίζουμε τη μεταβλητή x=0. Στη διεργασία-παιδί θέτουμε x=1, ενώ στη διεργασία-πατέρας θέτουμε x=2. Η έξοδος κατά την εκτέλεση είναι:

```
Child process created with pid 21128
Parent: x = 2
Hello from child process. My pid is 21128 and my parent's pid is 21127
Child: x = 1
```

Βλέπουμε πως κατά την εκτύπωση της μεταβλητής από τις δύο διεργασίες, η τιμή της είναι αυτή που ορίζουμε μέσα στην κάθε διεργασία, παρόλο που η μεταβλητή ορίζεται αρχικά πριν τη δημιουργία της νέας διεργασίας. Αυτό συμβαίνει γιατί, κατά τη δημιουργία της διεργασίας, δημιουργείται ένα αντίγραφο της μεταβλητής x με την τιμή που έχει, οπότε μετά είναι ανεξάρτητη από αυτή της διεργασίας-πατέρα και η κάθε διεργασία μπορεί να θέσει τη δίκη της τιμή στη μεταβλητή.

3 – Αναζήτηση από τη διεργασία-παιδί

```c
// Create pipe
int pfd[2];
if (pipe(pfd) == -1) {
    perror("Problem creating pipe");
    close(fdr);
    close(fdw);
    exit(1);
}
```

Πριν τη δημιουργία της διεργασίας παιδί ανοίγουμε τα δύο αρχεία με τον ίδιο τρόπο όπως και στην άσκηση 1, καθώς και δημιουργούμε ένα **pipe** (**pfd**) για την επικοινωνία των διεργασιών. Ελέγχουμε, επίσης, αν υπήρξε κάποιο σφάλμα κατά τη δημιουργία του pipe. Στη συνέχεια δημιουργείται η διεργασία-παιδί, στην οποία αναθέτουμε μόνο την αναζήτηση του

χαρακτήρα στο αρχείο. Η διεργασία-παιδί αρχικά κάνει **close()** το άκρο ανάγνωσης του **pipe** (**pfd[0]**) μιας και χρειάζεται μόνο να γράψει στο **pipe** το αποτέλεσμα. Αντίθετα, η διεργασία-πατέρα κάνει **close()** το άκρο εγγραφής του **pipe** (**pfd[1]**), αφού περιμένει να τελειώσει η διεργασία-παιδί και έπειτα διαβάζει το αποτέλεσμα από το **pipe**.

*Θα μπορούσαμε αντί να χρησιμοποιήσουμε **pipe**, η διεργασία-παιδί να γράφει το αποτέλεσμα στο αρχείο εξόδου. Όμως, ή εκφώνηση της άσκησης λέει να αναθέσουμε στη διεργασία-παιδί μόνο την αναζήτηση του χαρακτήρα και όχι τον υπόλοιπο χειρισμό των αρχείων.*

```
// Write count to pipe
if (write(pfd[1], &count, sizeof(count)) != sizeof(count)) {
    perror("Problem writing to pipe");
    close(fdr);
    close(fdw);
    exit(1);
}

close(pfd[1]);
```

Η διεργασία-παιδί διαβάζει και ελέγχει τους χαρακτήρες του αρχείου έναν έναν, όπως στην άσκηση 1, και αφού βρει το αποτέλεσμα το γράφει στο **pipe**. Κατά την εγγραφή στο pipe ελέγχει για πιθανό σφάλμα και μετά κλείνει το άκρο εγγραφής του **pipe** (**pfd[1]**), αφού πλέον δεν το χρειάζεται.

```
wait(&status);

// Read count from pipe
int count;
if (read(pfd[0], &count, sizeof(count)) != sizeof(count)) {
    perror("Problem reading from pipe");
    close(fdw);
    exit(1);
}
close(pfd[0]);
```

Η διεργασία-πατέρας περιμένει με **wait()** τη διεργασία-παιδί να τελειώσει, διαβάζει το αποτέλεσμα από το **pipe** και το αποθηκεύει στη μεταβλητή **count**. Κατά την ανάγνωση από το **pipe** ελέγχει για πιθανό σφάλμα και μετά κλείνει το άκρο ανάγνωσης του **pipe** (**pfd[0]**), αφού πλέον δεν το χρειάζεται. Τέλος, γράφει το αποτέλεσμα στο αρχείο εξόδου, όπως στην άσκηση 1.

*Μια διαφορετική υλοποίηση της διεργασίας-πατέρα θα ήταν αντί να περιμένει με **wait()** τη διεργασία-παιδί να τελειώσει και μετά να διαβάζει από το **pipe**, να χρησιμοποιήσουμε την blocking ιδιότητα της **read()**, με την οποία η **read()** θα σταματούσε τη ροή της διεργασίας-πατέρα έως ότου λάβει δεδομένα στο pipe από τη διεργασία-παιδί. Μόνο τότε η διεργασία-πατέρα θα συνέχιζε και θα έγγραφε το αποτέλεσμα στο αρχείο εξόδου. Με αυτή την*

*υλοποίηση και για να εξασφαλίσουμε ότι η διεργασία-παιδί θα τελειώσει πριν τελειώσει η διεργασία-πατέρα, θα πρέπει να βάλουμε ένα **wait()** πριν τελειώσει η διεργασία-πατέρα.*

Γενικότερα αυτή η υλοποίηση μπορεί να κάνει το πρόγραμμα πιο γρήγορο μιας και η διεργασία-πατέρα δεν περιμένει τη διεργασία-παιδί να τελειώσει, αλλά αρκεί να γράψει το αποτέλεσμα στο pipe για να συνεχίσει.

4 – Αντικατάσταση της διεργασίας-παιδί με execv

```
if (p == 0) {
    mypid = getpid();
    parentpid = getppid();
    printf("Hello from child process. My pid is %d and my parent's pid is %d\n",
        mypid, parentpid);

    char *args[] = {"./main-source-code", argv[1], argv[2], argv[3], NULL};
    execv(args[0], args);

    // If execv returns, there was an error
    perror("Problem executing child process");
    exit(1);
}
```

Σκοπός είναι η διεργασία-παιδί που δημιουργούμε να εκτελεί τον κώδικα που μας δόθηκε **(main-source-code.c).** Γι' αυτό, χρησιμοποιούμε το **execv()**, το οποίο αντικαταστεί τη διεργασία με τον κώδικα που ορίζουμε. Το **execv()** παίρνει δύο ορίσματα: το **path** του εκτελέσιμου προγράμματος που θέλουμε να εκτελέσει και έναν πίνακα με **strings** που αντιπροσωπεύει τα **arguments** αυτού του προγράμματος.

Στη δική μας περίπτωση θέλουμε να εκτελέσει το compiled κώδικα που μας δόθηκε, δηλαδή το αρχείο **[./main-source-code]**. Στον πίνακα **args** το 1ο στοιχείο είναι επίσης το **path** του εκτελέσιμου προγράμματος, το 2ο στοιχείο είναι το **input file**, το 3ο στοιχείο είναι το **output file**, το 4ο στοιχείο είναι το **search character** και το 5ο στοιχείο πρέπει να είναι **NULL**.

Το execv αντικαταστεί τη διεργασία-παιδί τη στιγμή που το καλούμε. Η μόνη περίπτωση που επιστρέφει κάτι είναι αν έχει υπάρξει κάποιο σφάλμα και σε αυτή την περίπτωση εμφανίζουμε ένα μήνυμα σφάλματος.

3 – Διαδιεργασιακή επικοινωνία

```c
#define P 3  // Number of child processes

int active_children = 0;

// Signal handler for SIGINT
void sighandler(int signum) {
    printf("\nActive search processes: %d\n", active_children);
}
```

Αρχικά, ορίζουμε τον αριθμό των διεργασιών-παιδία που θέλουμε να δημιουργήσουμε, μια global μεταβλητή που θα αποθηκεύουμε των αριθμών των ενεργών διεργασιών κατά την εκτέλεση και μια συνάρτηση **sighandler()**, την οποία θα χρησιμοποιήσουμε για να εμφανίζουμε το συνολικό αριθμό ενεργών διεργασιών που αναζητούν το αρχείο σε περίπτωση που το πρόγραμμα δεχτεί κάποιο σήμα.

```c
// Get file size
off_t filesize = lseek(fdr, 0, SEEK_END);
if (filesize == -1) {
    perror("Problem getting file size");
    close(fdr);
    exit(1);
}
close(fdr);
```

Στο main και αφού ανοίξουμε το αρχείο ανάγνωσης μετράμε το μέγεθος της συμβολοσειράς που περιέχει με τη χρήση της **lseek()**. Η **lseek** μετακινεί τον δείκτη αρχείου στο τέλος του αρχείου **fdr** και έτσι επιστρέφει τον αριθμό των **bytes** που περιέχει. Αυτό το αποθηκεύουμε στη μεταβλητή **filesize**, η οποία είναι τύπου **off_t**. Αν η **lseek** επιστρέψει -1 πρόκειται για σφάλμα.

```c
int pipes[P][2];
for (int i = 0; i < P; i++) {
    if (pipe(pipes[i]) == -1) {
        perror("Problem creating pipe");
        close(fdr);
        exit(1);
    }
}
```

Δημιουργούμε #P **pipes** για την επικοινωνία κάθε διεργασίας-παιδί με τη διεργασία-πατέρα.

```c
off_t sliced_size = filesize / P;
```

Χωρίζουμε το μέγεθος του αρχείου σε P κομμάτια, ώστε κάθε διεργασία-παιδί να κάνει αναζήτηση σε διαφορετικό τμήμα του κειμένου.

Με τη χρήση του for loop δημιουργούμε P διεργασίες-παιδιά. Σε κάθε επανάληψη αυξάνουμε τη μεταβλητή **active_children** κατά 1 και κάνουμε **fork()** για τη δημιουργία της διεργασίας.

```c
if (pid == 0) {
    signal(SIGINT, SIG_IGN);  // Ignore SIGINT in child processes
    close(pipes[i][0]);
```

Στη διεργασία-παιδί θέτουμε τη **signal()**, ώστε να αγνοεί το σήμα **Control+C (SIGINT)**, μιας και η διαχείριση αυτού του σήματος θέλουμε να γίνεται μόνο από τη διεργασία-πατέρα.

Έπειτα, κάθε διεργασία-παιδί ανοίγει το δικό του αρχείο ανάγνωσης. Ο λόγος, για τον οποίο γίνεται αυτό, είναι γιατί οι διεργασίες μοιράζονται κοινό δείκτη αρχείου και αυτός μπορεί να βρίσκεται σε μια συγκεκριμένη θέση κάθε φορά. Αντίθετα, εμείς θέλουμε κάθε διεργασία να διαβάζει διαφορετικό τμήμα του αρχείου ταυτόχρονα με τις άλλες.

```c
off_t start_index = i * sliced_size;
off_t end_index = (i == P - 1) ? filesize : (i + 1) * sliced_size;

// Set file offset to start_index
if (lseek(fdr, start_index, SEEK_SET) == -1) {
    perror("Problem setting file offset");
    close(fdr);
    exit(1);
}
```

Βρίσκουμε έτσι το **start index** και **end index**, δηλαδή το τμήμα αναζήτησης κάθε διεργασίας. Το **end index** της τελευταίας διεργασίας ισούται με το **filesize**. Με τη χρήση πάλι της **lseek()** μεταφέρουμε το δείκτη στο **start index**, ώστε να αρχίσει να διαβάζει από εκείνο το σημείο.

```c
// Read from start_index to end_index
for (off_t index = start_index; index < end_index; index++) {
    if (read(fdr, &cc, 1) == -1) {
        perror("Problem reading from file");
        close(fdr);
        exit(1);
    }
    if (cc == c2c) child_count++;
}

// Write child_count to pipe
if (write(pipes[i][1], &child_count, sizeof(child_count)) != sizeof(child_count)) {
    perror("Problem writing to pipe");
    close(fdr);
    exit(1);
}

close(fdr);
close(pipes[i][1]);
exit(0);
```

Στη συνέχεια με το **for loop** διαβάζει το τμήμα του αρχείου που του αντιστοιχεί και αφού τελειώσει γράφει το αποτέλεσμα στο **pipe**. Η κάθε διεργασία-παιδί συνδέεται με το δικό της pipe αλλά και με το δικό της τμήμα του αρχείου που της αντιστοιχεί μέσω του αριθμού **i** που προκύπτει από το **for loop**. Τέλος, κλείνει το αρχείο ανάγνωσης και το άκρο εγγραφής του **pipe** και κάνει **exit()**.

```c
signal(SIGINT, sighandler);  // Set signal handler for SIGINT
int total_count = 0;

// Wait for all children to finish
for (int i = 0; i < P; i++) {
    wait(NULL);
    active_children--;
}
```

Στη διεργασία-πατέρα, αντίστοιχα, θέτουμε τη **signal()**, ώστε να εκτελεί τη συνάρτηση **sighandler()**, που ορίσαμε νωρίτερα, όταν δέχεται **Control+C**. Επίσης, με ένα **for loop** κάνει **wait()**, ώστε να τελειώσουν όλες οι διεργασίες-παιδιά. Κάθε φορά που τελειώνει μια διεργασία-παιδί μειώνουμε τη μεταβλητή **active_children** κατά 1.

```c
// Read child_count from pipes and calculate total_count
for (int i = 0; i < P; i++) {
    int child_count;
    if (read(pipes[i][0], &child_count, sizeof(child_count)) != sizeof(child_count)) {
        perror("Problem reading from pipe");
        exit(1);
    }
    close(pipes[i][0]);
    total_count += child_count;
}
```

Αφού τελειώσουν όλες οι διεργασίες-παιδιά, η διεργασία-πατέρα διαβάζει το αποτέλεσμα από κάθε **pipe** και το προσθέτει στο συνολικό αποτέλεσμα.

Τέλος, ανοίγει το αρχείο εξόδου και γράφει το αποτέλεσμα σε αυτό, όπως στην άσκηση 1.

Σχετικά με το signal(): εάν βάζαμε μόνο το `signal(SIGINT, sighandler)` πριν το fork(), τότε η κάθε διεργασία-παιδί θα κληρονομούσε αυτή τη διαχείριση του σήματος. Έτσι, όταν το πρόγραμμα δεχόταν το **Control+C** θα εκτελούσαν τη συνάρτηση **sighandler()** όλες οι διεργασίες, δηλαδή παραπάνω από μια φορά. Γι' αυτό, βάζουμε τη διαχείριση αυτή του σήματος μόνο στη διεργασία-πατέρα, ώστε μόνο αυτή να εκτελέσει τη συνάρτηση **sighandler().** Έτσι, όμως, σε κάθε διεργασία-παιδί δεν ορίζουμε τι να κάνει εάν δεχτεί το σήμα **Control+C** και by default αυτό το σήμα τερματίζει τη διεργασία. Γι' αυτό και σε κάθε διεργασία-παιδί βάζουμε το `signal(SIGINT, SIG_IGN)`, ώστε να αγνοεί το σήμα.

*Έχουμε φτιάξει ένα **Makefile**, όπου κάνει compile κάθε αρχείο **.c** σε αντίστοιχο εκτελέσιμο, καθώς και φροντίζει να υπάρχει το εκτελέσιμο **main-source-code**, το οποίο εκτελούμε από το πρόγραμμα ex02-4 μέσω του execv.*