ΕΡΓΑΣΤΗΡΙΟ 6

Αγωγοί

Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τη χρήση των ανώνυμων και επώνυμων αγωγών.

Παράδειγμα 1. Δημιουργία και χρήση αγωγού από την ίδια διεργασία (αρχείο pipeEx1.c).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define MSGSIZE 16
```

Σε αυτό το πρώτο απλό παράδειγμα υπάρχει μία και μοναδική διεργασία, η οποία επικοινωνεί με τον εαυτό της, δηλαδή γράφει δεδομένα στην έξοδό της τα οποία στη συνέχεια ανακατευθύνονται πίσω στην είσοδό της. Αυτό το παράδειγμα προφανώς δεν έχει καμία πρακτική εφαρμογή και ο στόχος του είναι απλά να επιδείξουμε τη χρήση των βασικών συναρτήσεων διαχείρισης αγωγών.

Η διεργασία στέλνει στον εαυτό της τα τρία μηνύματα που ακολουθούν και έχουν τα ονόματα msg1, msg2 και msg3.

```
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main() {
   char inbuf[MSGSIZE];
```

Η συνάρτηση pipe παίρνει ως όρισμα έναν πίνακα p μήκους 2 ακεραίων και εφόσον λειτουργήσει σωστά επιστρέφει σε αυτόν τον πίνακα τους περιγραφείς αρχείων για τα δύο άκρα του αγωγού, με τον περιγραφέα p[0] να σχετίζεται με το άκρο του αναγνώστη και τον περιγραφέα p[1] να σχετίζεται με το άκρο του συγγραφέα. Εάν η συνάρτηση επιστρέψει μηδενική τιμή, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα και η εφαρμογή τερματίζει.

```
   int p[2], i;
   if (pipe(p) < 0) exit(1);
```

Ως άσκηση μπορείτε στην περίπτωση της επιστροφής αρνητικής τιμής να εκτυπώσετε την αριθμητική τιμή του σφάλματος errno και τη λεκτική περιγραφή του, χρησιμοποιώντας τη συνάρτηση perror. Αυτό είναι κάτι που υλοποιείται στην επόμενη άσκηση, οπότε μπορείτε να ανατρέξτε εκεί για να δείτε πώς γίνεται.

Εάν όλα πάνε καλά και δημιουργηθεί ο αγωγός, η διεργασία γράφει στην έξοδό της τα τρία μηνύματα χρησιμοποιώντας τη συνάρτηση write. Πρόκειται για την γνωστή συνάρτηση εγγραφής σε αρχείο η οποία χρησιμοποιείται και στην περίπτωση των αγωγών, κάτι που είναι αναμενόμενο εφόσον και αυτοί περιγράφονται από τους συνήθεις περιγραφείς αρχείων.

```
   write(p[1], msg1, MSGSIZE);
   write(p[1], msg2, MSGSIZE);
   write(p[1], msg3, MSGSIZE);
```

Μετά την εγγραφή των τριών μηνυμάτων στην έξοδό της, η εφαρμογή καλεί μέσα από ένα επαναληπτικό βρόχο τριών επαναλήψεων τη συνάρτηση read, για να διαβάσει αυτά τα μηνύματα. Το μήνυμα που διαβάζεται σε κάθε κύκλο επανάληψης αποθηκεύεται στην περιοχή μνήμης inbuf που έχει μέγεθος MSGSIZE. Στην προκειμένη περίπτωση, η άσκηση είναι πάρα πολύ απλή, διότι γνωρίζουμε και το πόσα μηνύματα αναμένουμε να διαβάσουμε και το μέγεθος του κάθε μηνύματος. Στη γενική περίπτωση τα πράγματα δεν είναι τόσο απλά. Μετά την ανάγνωσή του, το κάθε μήνυμα εκτυπώνεται στην οθόνη.

```
    for (i = 0; i < 3; i++) {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf); }
    return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Προκειμένου να βελτιώσετε την ποιότητα της υλοποίησης, μπορείτε να ελέγξετε την επιστρεφόμενη τιμή των συναρτήσεων read και write και εφόσον αυτή είναι αρνητική, να εκτυπώσετε την τιμή του σφάλματος errno και το κατάλληλο διαγνωστικό μήνυμα καλώντας τη συνάρτηση perror.

```
amarg@amarg-vbox:~/shared/Lab6$ ./pipeEx1
hello, world #1
hello, world #2
hello, world #3
amarg@amarg-vbox:~/shared/Lab6$
```

Παράδειγμα 2. Δημιουργία και χρήση αγωγού από τοπολογία πατέρα – παιδιού (Παράδειγμα Α) (αρχείο pipeEx2.c).

Σε αυτό το παράδειγμα δημιουργούνται δύο διεργασίες οι οποίες σχετίζονται με σχέση πατέρα - παιδιού και οι οποίες επικοινωνούν διά της χρήσεως ενός ανώνυμου αγωγού. Η διεργασία πατέρας αποστέλλει (σε ένα και μόνο βήμα) μία συμβολοσειρά στη διεργασία παιδί, η οποία την παραλαμβάνει μέσω μιας επαναληπτικής διαδικασίας, διαβάζοντας ένα χαρακτήρα κάθε φορά.

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main() {
  int fd[2], errno;  char buf;
  const char* msg = "Hello world .. Have a nice day!!\n";
```

Σε αυτό το σημείο προσπαθούμε να δημιουργήσουμε τον ανώνυμο αγωγό καλώντας τη συνάρτηση pipe η οποία αρχικοποιεί και επιστρέφει έναν πίνακα δύο ακεραίων που περιέχει τους περιγραφείς αρχείων των δύο άκρων του αγωγού. Εάν η συνάρτηση επιστρέψει αρνητική τιμή, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα. Στην περίπτωση αυτή η εφαρμογή εκτυπώνει την αριθμητική τιμή του σφάλματος καθώς και τη λεκτική περιγραφή του καλώντας τη συνάρτηση perror και τερματίζει τη λειτουργία της.

```
  if (pipe(fd) < 0) {
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-1); }
```

Εάν ο αγωγός δημιουργηθεί με επιτυχία, η εφαρμογή (πού νοείται ως γονική διεργασία) καλεί τη συνάρτηση fork για να δημιουργήσει τη θυγατρική διεργασία. Εάν η συνάρτηση fork επιστρέψει αρνητική τιμή, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα. Στην περίπτωση αυτή η εφαρμογή, όπως και προηγουμένως, τερματίζει τη λειτουργία της αφού προηγουμένως εκτυπώσει την αριθμητική και τιμή του σφάλματος και τη λεκτική του περιγραφή.

```
  pid_t cpid = fork();
  if (cpid < 0) {
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-2); }
```

Εάν η fork λειτουργήσει με επιτυχία, δημιουργεί τη θυγατρική διεργασία και επιστρέφει κωδικό και στις δύο διεργασίες. Θυμηθείτε πως ο κωδικός cpid που επιστρέφει η fork στην γονική διεργασία είναι θετικός και ίσος με το PID της θυγατρικής διεργασίας, ενώ ο κωδικός που επιστρέφεται από τη fork στη θυγατρική διεργασία είναι ίσος με το μηδέν. Επομένως ο κώδικας που γράφεται μέσα στη συνθήκη if (cpid==0) { …… } εκτελείται από τη θυγατρική διεργασία, ενώ ο κώδικας που γράφεται μέσα στο else { …. } εκτελείται από τη γονική διεργασία.

// child process

```
if (cpid==0) {
   // child only reads, so close write end
   close(fd[1]);    ←Δείτε το σχόλιο στο τέλος της λύσης
```

Μέσω μιας επαναληπτικής διαδικασίας η οποία διαρκεί για όσο χρόνο η συνάρτηση read επιστρέφει τιμή μεγαλύτερη του μηδενός, η θυγατρική διεργασία διαβάζει έναν έναν τους χαρακτήρες του μηνύματος που της έστειλε η γονική διεργασία. Ο χαρακτήρας που διαβάζεται κατά τη διάρκεια της κάθε επανάληψης αποθηκεύεται στη μεταβλητή buf που είναι τύπου char και στη συνέχεια μέσω της συνάρτησης write εγγράφεται στο αρχείο με περιγραφέα τον *STDOUT_FILENO*. *Αλλά αυτός ο περιγραφέας αρχείου αναφέρεται στην προεπιλεγμένη έξοδο, η οποία είναι η οθόνη. Επομένως ο χαρακτήρας εκτυπώνεται στην οθόνη. Αμφότερες οι διεργασίες μετά την ολοκλήρωση της λειτουργίας τους τερματίζονται καλώντας τη συνάρτηση exit.*

```
   while (read(fd[0], &buf, 1) > 0)
         write(STDOUT_FILENO, &buf, sizeof(buf));
   close(fd[0]);   ←Δείτε το σχόλιο στο τέλος της λύσης
   exit(0); }
```

// parent process

```
else {
   // parent only writes, so close read end
   close(fd[0]);    ←Δείτε το σχόλιο στο τέλος της λύσης
```

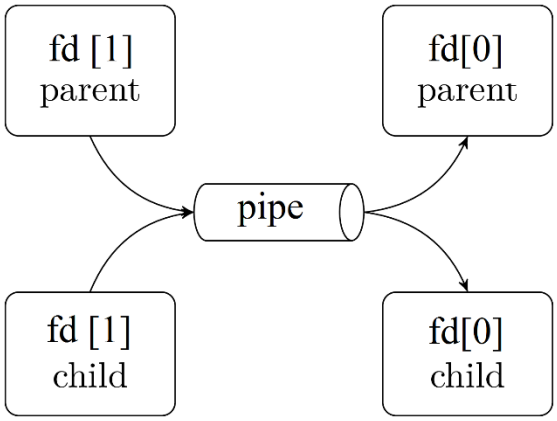
Η γονική διεργασία αποστέλλει την έξοδο της στη θυγατρική διεργασία σε ένα και μόνο βήμα, δηλαδή αποστέλλει όλο το μήνυμα με τη μία, χρησιμοποιώντας ως όρισμα στη συνάρτηση write τη συμβολοσειρά msg.

```
   write(fd[1], msg, strlen(msg));
   close(fd[1]);
   wait(NULL);
   exit(0); }
return 0; }
```

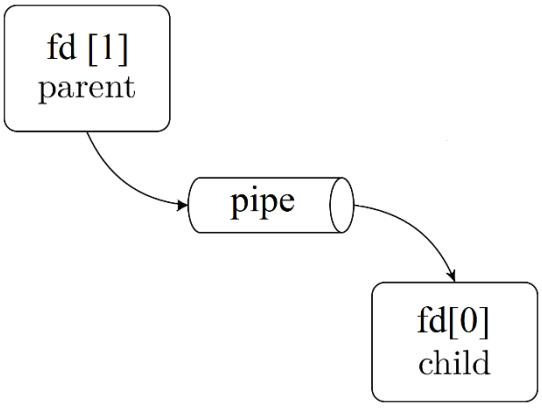
Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab6$ ./pipeEx2
Hello world .. Have a nice day!!
```

Ένα σχόλιο σχετικά με τις δύο συναρτήσεις close που είναι εκτυπωμένες με κόκκινα μεγάλα και έντονα γράμματα. Όπως είναι γνωστό, η συνάρτηση fork πραγματοποιεί κλωνοποίηση της γονικής διεργασίας για να κατασκευάσει τη θυγατρική διεργασία και κατά συνέπεια όπως έχει σχολιαστεί στο οικείο μάθημα, για κάθε μεταβλητή της εφαρμογής υφίστανται δύο αντίγραφα, ένα αντίγραφο στη γονική διεργασία και ένα αντίγραφο στην θυγατρική διεργασία. Αυτό σημαίνει πως ο πίνακας int fd[2] δηλαδή οι περιγραφείς αρχείων fd[0] και fd[1] υπάρχουν και στις δύο διεργασίες και επειδή έχουν τις ίδιες τιμές - αφού μιλάμε για πανομοιότυπα αντίγραφα - δείχνουν στα άκρα του ίδιου αγωγού όπως φαίνεται στο σχήμα που ακολουθεί. Με άλλα λόγια, αμφότερες οι διεργασίες έχουν πρόσβαση σε αμφότερα τα άκρα του αγωγού.



Ωστόσο, η κάθε διεργασία χρειάζεται μόνο τον έναν από αυτούς τους δύο περιγραφείς αρχείων και για το λόγο αυτό καλεί την close για να κλείσει αυτόν που δεν χρειάζεται. Ειδικότερα η γονική διεργασία που θα κάνει εγγραφή χρειάζεται μόνο τον fd[1] οπότε κλείνει τον fd[0], ενώ η θυγατρική διεργασία που θα κάνει μόνο ανάγνωση χρειάζεται μόνο τον fd[0] οπότε κλείνει τον fd[1]. Οι δύο συναρτήσεις close που είναι εκτυπωμένες με κόκκινα μεγάλα και έντονα γράμματα κάνουν ακριβώς αυτό. Η νέα εικόνα που προκύπτει μετά από το κλείσιμο αυτών των δύο περιγραφέων απεικονίζεται στη συνέχεια.



(Σε μία πιο τεχνική περιγραφή, ο αναγνώστης ενημερώνεται πως ο συγγραφέας έχει ολοκληρώσει τη διαδικασία εάν ανιχνεύσει έναν χαρακτήρα EOF (End Of File). Αλλά για να συμβεί αυτό θα πρέπει να μην υπάρχει ανοικτός κανένας άλλος περιγραφέας που να σχετίζεται με άκρο αγωγού που προορίζεται για εγγραφή).

Δείτε και τη διαφάνεια υπ' αριθμόν 19 της σχετικής παρουσίασης.

Παράδειγμα 3. Δημιουργία και χρήση αγωγού από τοπολογία πατέρα – παιδιού (Παράδειγμα Β) (αρχείο pipeEx3.c).

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
```

Σε αυτό το παράδειγμα αρχιτεκτονικής γονικής θυγατρικής διεργασίας η επικοινωνία πραγματοποιείται κατά την αντίστροφη κατεύθυνση σε σχέση με πριν δηλαδή πραγματοποιείται από τη θυγατρική προς τη γονική διεργασία. ειδικότερα η θυγατρική διεργασία ζητά από το χρήστη μέσω ενός μηνύματος να καταχωρήσει κάτι στην οθόνη το οποίο στη συνέχεια μεταβιβάζει στον αγωγό προκειμένου αυτό να διαβαστεί από την γονική διεργασία.

```
int main() {
  int fd[2], errno, count;
  char buffer[1024];
```

Οι διαδικασίες ελέγχου που πραγματοποιούνται (προκειμένου να διαπιστωθεί εάν οι συναρτήσεις pipe και fork έχουν επιστρέψει αρνητική τιμή) είναι ίδιες με πριν και δεν χρειάζεται να επαναληφθούν εκ νέου ενώ η συμπεριφορά της fork και ο τρόπος διάκρισης της γονικής από τη θυγατρική διεργασία θεωρούνται πλέον γνωστά.

```
if (pipe(fd) < 0) {
    printf("Error Number : %d\n", errno);
    perror("Error Description");
    return (-1); }

pid_t cpid = fork();

if (cpid < 0) {
    printf("Error Number : %d\n", errno);
    perror("Error Description");
    return (-2); }
```

Κώδικας θυγατρικής διεργασίας

```
// child process

if (cpid==0) {
   close(fd[0]);
```

Η θυγατρική διεργασία ζητά από το χρήστη να καταχωρήσει μία συμβολοσειρά από το πληκτρολόγιο την οποία διαβάζει με τη συνάρτηση gets και στη συνέχεια εκτυπώνει στην οθόνη με τη συνάρτηση printf.

```
    printf("Child Process --> Enter an input: ");
    fgets(buffer, sizeof(buffer), stdin);
    printf("Child process: User Inpus is %s", buffer);
```

Στη συνέχεια μέσω της συνάρτησης write εγγράφει τη συμβολοσειρά στο άκρο του συγγραφέα του αγωγού προκειμένου αυτή να διαβαστεί από τη γονική διεργασία.

```
    count = write(fd[1], buffer, strlen(buffer)+1);
    exit(0); }
```

Κώδικας γονικής διεργασίας

```
// parent process

else {
   close(fd[1]);
```

Η γονική διεργασία με τη σειρά της διαβάζει τη συμβολοσειρά από το άκρο του αναγνώστη και την εκτυπώνει στην οθόνη. Η απλή κλήση της wait που υπάρχει στο τέλος, διασφαλίζει πως η γονική διεργασία θα ολοκληρώνεται πάντοτε μετά τον τερματισμό της θυγατρικής διεργασίας.

```
    count = read(fd[0], buffer, sizeof(buffer));
    printf("Parent process: message is %s", buffer);
    wait(NULL);   }
  return 0; }
```

Παρατηρήστε πως σε αυτό το παράδειγμα οι διεργασίες αν και έχουν κλείσει τους περιγραφείς αρχείου που δεν χρειάζονται, ωστόσο στο τέλος δεν κλείνουν και τους περιγραφείς αρχείου που έχουν χρησιμοποιήσει, κάτι που έγινε στο προηγούμενο παράδειγμα. Παρατηρήστε πως αυτή η παράλειψη δεν δημιούργησε κανένα πρόβλημα στην εκτέλεση του κώδικα. Ωστόσο, γενικά, δεν είναι καλό για μία διεργασία να αφήνει ανοιχτά αρχεία κατά τον τερματισμό της και για το

λόγο αυτό τα άκρα του αγωγού που χρησιμοποιήθηκαν θα πρέπει να κλείσουν όπως κάναμε και πριν. Προσθέστε λοιπόν μόνοι σας τον αναγκαίο κώδικα.

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab6$ ./pipeEx3
Child Process --> Enter an input: Hello, my name is Athanasios
Child process: User Inpus is Hello, my name is Athanasios
Parent process: message is Hello, my name is Athanasios
amarg@amarg-vbox:~/shared/Lab6$
```

Παράδειγμα 4. Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών (αρχείο pipeEx4.c).

Αυτό το πολύ ενδιαφέρον παράδειγμα κώδικα προσομοιώνει τον τρόπο με τον οποίο χρησιμοποιούνται οι αγωγοί από το φλοιό του λειτουργικού συστήματος. Ας υποθέσουμε, για παράδειγμα, ότι θέλουμε να εκτελέσουμε την εντολή

ls -l <directory name> | grep <string>

όπου <directory name> ένα όνομα καταλόγου και <string> μία συμβολοσειρά. Όπως είναι γνωστό από τη βασική θεωρία, αυτός ο συνδυασμός των εντολών ls και grep, εκτυπώνει τα ονόματα των αρχείων που βρίσκονται στον κατάλογο που έχει οριστεί και περιέχουν την εν λόγω συμβολοσειρά, όπως απεικονίζεται στο επόμενο παράδειγμα (η τελεία που χρησιμοποιείται στην κλήση της ls –l περιγράφει τον τρέχοντα κατάλογο).

```
amarg@amarg-vbox:~/shared/Lab6$ ls
pipeEx1    pipeEx2    pipeEx3     pipeEx3a.o  pipeEx4    side1    side2
pipeEx1.c  pipeEx2.c  pipeEx3a    pipeEx3.c   pipeEx4.c  side1.c  side2.c
pipeEx1.o  pipeEx2.o  pipeEx3a.c  pipeEx3.o   pipeEx4.o  side1.o  side2.o
amarg@amarg-vbox:~/shared/Lab6$ ls -l . | grep side
-rwxrwxrwx 1 root root 17032 Οκτ   3 19:55 side1
-rwxrwxrwx 1 root root  1100 Οκτ   3 19:55 side1.c
-rwxrwxrwx 1 root root  2440 Οκτ   3 19:55 side1.o
-rwxrwxrwx 1 root root 17032 Οκτ   3 19:55 side2
-rwxrwxrwx 1 root root  1009 Οκτ   3 19:55 side2.c
-rwxrwxrwx 1 root root  2440 Οκτ   3 19:55 side2.o
amarg@amarg-vbox:~/shared/Lab6$
```

Προκειμένου ο φλοιός να υλοποιήσει την παραπάνω αλληλουχία εντολών, δημιουργεί δύο διεργασίες οι οποίες χρησιμοποιούν έναν αγωγό με τον γνωστό πλέον τρόπο και αναθέτει σε αυτές τις δύο διεργασίες την εκτέλεση των συνεργαζόμενων εντολών. Δηλαδή, στην πρώτη θυγατρική διεργασία θα αναθέσει την εκτέλεση της εντολής ls, ενώ στη δεύτερη θυγατρική διεργασία θα αναθέσει την εκτέλεση της εντολής grep. Οι δύο αυτές εντολές θα εκτελεστούν η καθεμία με το δικό της όρισμα και στη συνέχεια μέσω του αγωγού θα επικοινωνήσουν μεταξύ τους οδηγώντας τελικά στο επιθυμητό αποτέλεσμα. Αυτή ακριβώς τη διαδικασία προσομοιώνει το παράδειγμα που ακολουθεί.

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

int main(int argc, char *argv[]) {
   int pid, pid_ls, pid_grep, fd[2];
```

Αρχικά πραγματοποιούμε ως συνήθως το γνωστό έλεγχο σωστής χρήσης της εφαρμογής, η οποία για να λειτουργήσει σωστά θα πρέπει να ορίσουμε ένα όνομα καταλόγου και μία συμβολοσειρά. Με άλλα λόγια θα πρέπει να καταχωρήσουμε στη γραμμή εντολών τρεις συμβολοσειρές: το όνομα της εφαρμογής και τα δύο ορίσματα της. Εάν λοιπόν η τιμή του ορίσματος argc δεν είναι ίση με 3, αυτό σημαίνει όπως το πρόγραμμα δεν καλείται με το σωστό τρόπο, οπότε εκτυπώνει ένα

ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του. Από την άλλη πλευρά, εάν η εφαρμογή κληθεί σωστά, τότε το όνομα του καταλόγου θα βρίσκεται αποθηκευμένο στη συμβολοσειρά argv[1], ενώ η δεύτερη συμβολοσειρά θα βρίσκεται αποθηκευμένη στη συμβολοσειρά argv[2]. Αυτή η πληροφορία χρειάζεται προκειμένου να γίνει κατανοητός ο κώδικας που ακολουθεί.

```
if (argc != 3) {
    printf("Usage: pipeEx4 dirname string\n");
    return (-1); }

printf("Parent process: Grepping %s for %s\n", argv[1], argv[2]);
```

Στο σημείο αυτό καλείται η συνάρτηση pipe για να δημιουργήσει τους περιγραφείς αρχείων για το άκρο ανάγνωσης και το άκρο εγγραφής του αγωγού. Εάν η συνάρτηση επιστρέψει την τιμή -1, αυτό σημαίνει πως η δημιουργία του αγωγού δεν κατέστη δυνατή για κάποιο λόγο και το πρόγραμμα τερματίζεται, αφού δεν είναι δυνατόν να εκτελεστεί χωρίς τον αγωγό.

```
if (pipe(fd)==-1) {
    printf("Parent process: Failed to create pipe\n");
    return (-2); }
```

Στο σημείο αυτό καλείται η πρώτη fork για να δημιουργήσει την πρώτη θυγατρική διεργασία στην οποία θα ανατεθεί η εκτέλεση της εντολής grep. Εάν η fork επιστρέψει αρνητική τιμή, αυτό σημαίνει πως η διεργασία δεν δημιουργήθηκε και το πρόγραμμα ολοκληρώνεται χωρίς επιτυχία, ενώ στην αντίθετη περίπτωση η διεργασία δημιουργείται και η εφαρμογή συνεχίζει τη λειτουργία της.

```
// Fork a process to run grep

pid_grep = fork();
if (pid_grep == -1) {
    printf("Parent process: Could not fork process to run grep\n");
    return (-3); }
else if (pid_grep==0) {

     printf("Child process 1: grep child will now run\n");
```

Στο σημείο αυτό αντί να δουλέψουμε με το συνήθη τρόπο, καλούμε τη συνάρτηση dup2 (έτσι ώστε να έρθουμε σε επαφή με όσο το δυνατό περισσότερες αναρτήσεις). Η συνάρτηση dup2 (από τη λέξη duplicate → αντιγράφω, αναπαράγω) δέχεται ως όρισμα έναν περιγραφέα αρχείου και μία τιμή και δημιουργεί ένα νέο αντίγραφο αυτού του περιγραφέα στο οποίο εκχωρεί την καθοριζόμενη τιμή. Μετά τη δημιουργία αυτού του δεύτερου περιγραφέα αρχείου, οι δύο περιγραφείς είναι εντελώς ισοδύναμοι και μπορούν να χρησιμοποιηθούν ο ένας τη θέση του άλλου. Στον κώδικα του παραδείγματος, ως τιμή για τον νέο περιγραφέα, ο οποίος αποτελεί αντίγραφο του fd[0], ορίζεται η τιμή *STDIN_FILENO η οποία περιγράφει την προεπιλεγμένη είσοδο, που είναι το πληκτρολόγιο. Από τη στιγμή που έχουμε δημιουργήσει αντίγραφα του περιγραφέα fd[0], μπορούμε να τον κλείσουμε, αφού πλέον δεν χρειαζόμαστε άλλο, ενώ μαζί με αυτόν κλείνουμε και τον περιγραφέα fd[1], για το λόγο που εξηγήσαμε προηγουμένως. Αυτές οι διαδικασίες πραγματοποιούνται από τις δύο συναρτήσεις close που ακολουθούν την κλήση της dup2.*

```
    // Set fd[0] (stdin) to the read end of the pipe

    if (dup2 (fd[0], STDIN_FILENO) == -1) {
       printf("Child process 1: grep dup2 failed\n");
       return (-4); }

    // Close the pipe now that we've duplicated it
    close(fd[0]);
    close(fd[1]);
```

Το μόνο που έμεινε στο σημείο αυτό, είναι να αναθέσουμε στην θυγατρική διεργασία που δημιουργήσαμε, να εκτελέσει την εντολή grep. Για να το κάνουμε αυτό καλούμε κάποια από τις συναρτήσεις της οικογένειας exec. Στο παράδειγμα του κώδικα καλείται η execv , στην οποία τα ορίσματα της εντολής προς εκτέλεση δεν διαβιβάζονται το ένα μετά το άλλο υπό τη μορφή λίστας, όπως συμβαίνει με την execl, αλλά καταχωρούνται πρώτα σε ένα πίνακα συμβολοσειρών (στο παράδειγμά μας τον πίνακα *new_argv*) και περνάμε ως όρισμα στη συνάρτηση, αυτόν τον πίνακα. Στην πραγματικότητα, η συνάρτηση που καλείται σε αυτό το σημείο, δεν είναι η απλή execv αλλά η execve, η οποία παίρνει ένα επιπλέον όρισμα που περιέχει μεταβλητές περιβάλλοντος. Αυτές οι μεταβλητές καταχωρούνται σε ένα άλλο πίνακα συμβολοσειρών με όνομα *envp*. *Είναι προφανές, πως από τη στιγμή που μπορούμε να εκτελέσουμε μία εντολή του λειτουργικού συστήματος ορίζοντας για το σκοπό αυτό τις κατάλληλες σε κάθε περίπτωση μεταβλητές περιβάλλοντος, θα πρέπει αυτή η δυνατότητα να προσφέρεται και από τις συναρτήσεις exec και εδώ επιδεικνύεται ένας τρόπος με τον οποίο γίνεται αυτό. Θυμηθείτε πως όταν καλείται η συνάρτηση exec η (θυγατρική στην προκειμένη περίπτωση) διεργασία που την κάλεσε, παύει πλέον να υφίσταται, αφού αντικαθίσταται στη μνήμη από τη διεργασία που δημιουργεί η εντολή που καλείται από την exec και η οποία στην προκειμένη περίπτωση, είναι η εντολή grep.*

```
// Setup the arguments/environment to call
char * new_argv[] = { "/bin/grep", argv[2], 0 };
char * envp[] = { "HOME=/", "PATH=/bin:/usr/bin", "USER=brandon", 0 };
// Call execve(2) which will replace the executable image of this process
execve(new_argv[0], &new_argv[0], envp);
// Execution will never continue in this process unless execve returns
// because of an error
printf("Child process 1: Oops, grep failed!\n");
return (-5); }
```

Στο σημείο αυτό καλείται η δεύτερη fork για να δημιουργήσει τη δεύτερη θυγατρική διεργασία στην οποία θα ανατεθεί η εκτέλεση της εντολής ls -l. Ο κώδικας είναι ακριβώς ίδιος με πριν και οτιδήποτε έχει αναφερθεί προηγουμένως, ισχύει αυτούσιο ως έχει.

```
// Fork a process to run ls
pid_ls = fork();
if (pid_ls == -1) {
    printf("Parent Process: Could not fork process to run ls\n");
    return (-6); }
else if (pid_ls == 0) {
    printf("Child process 2: ls child will now run\n");
    printf("----------------------\n");
    // Set fd[1] (stdout) to the write end of the pipe
    if (dup2(fd[1], STDOUT_FILENO) == -1) {
       fprintf(stderr, "ls dup2 failed\n");
       return (-7); }
    // Close the pipe now that we've duplicated it
    close(fd[0]);
    close(fd[1]);
    // Setup the arguments/environment to call
    char *new_argv[] = { "/bin/ls", "-la", argv[1], 0 };
    char *envp[] = { "HOME=/", "PATH=/bin:/usr/bin", "USER=brandon", 0 };
    // Call execve(2) which will replace the executable image of this process
    execve(new_argv[0], &new_argv[0], envp);
    // Execution will never continue in this process unless execve returns
    // because of an error
    fprintf(stderr, "child: Oops, ls failed!\n");
    return (-8); }
```

Από το σημείο αυτό και κάτω, εκτελείται ο κώδικας της γονικής διεργασίας. Η γονική διεργασία δεν συμμετέχει στη λειτουργία του αγωγού - το μόνο που κάνει είναι να δημιουργήσει τις θυγατρικές διεργασίες οι οποίες και θα χρησιμοποιήσουν τον αγωγό - και για το λόγο αυτό κλείνει τους δύο περιγραφείς αρχείων, αφού δεν τους χρειάζεται.

Ουσιαστικά, το μόνο που κάνει η γονική διεργασία, είναι να καλέσει τη συνάρτηση wait για να αναμένει την ολοκλήρωση των θυγατρικών διεργασιών και να ολοκληρωθεί μετά από αυτές, ενώ εκτυπώνει και κάποια ενημερωτικά μηνύματα.

```
// Parent doesn't need the pipes
close(fd[0]);
close(fd[1]);
printf("Parent: Parent will now wait for children to finish execution\n");
// Wait for all children to finish
while (wait(NULL) > 0);
printf("---------------------\n");
printf("parent: Children has finished execution, parent is done\n");
return 0; }
```

Παράδειγμα εκτέλεσης της εφαρμογής, ακολουθεί στη συνέχεια. Η εφαρμογή δέχεται ως όρισμα τον προσωπικό κατάλογο του χρήστη και τη συμβολοσειρά sample και επιστρέφει όσα αρχεία περιέχουν στο όνομά τους αυτή τη συμβολοσειρά.

```
amarg@amarg-vbox:~/shared/LAb6$ ./pipeex4 ~ sample
Parent process: Grepping /home/amarg for sample
Parent: Parent will now wait for children to finish execution
Child process 2: ls child will now run
---------------------
Child process 1: grep child will now run
-rwxrwxr-x  1 amarg amarg    20032 Oct 30 11:05 sample
srwxrwxr-x  1 amarg amarg        0 Oct 17 12:20 sample-socket
-rw-rw-r--  1 sys   sys       1238 Oct  8 12:09 sample.c
-rw-rw-r--  1 amarg amarg     8888 Oct 30 11:05 sample.o
-r-srw-rwt  1 amarg amarg       26 Nov  2 10:57 sample.txt
---------------------
parent: Children has finished execution, parent is done
amarg@amarg-vbox:~/shared/LAb6$
```

Παράδειγμα 5. Παράδειγμα δημιουργίας επώνυμου αγωγού

Το τελευταίο παράδειγμα του σημερινού εργαστηρίου, αποτελεί παράδειγμα χρήσης επωνύμων αγωγών, οι οποίοι περιγράφονται από πραγματικά αρχεία του συστήματος, αποθηκεύονται στο σκληρό δίσκο και επιτρέπουν την επικοινωνία εντελώς ανεξάρτητων μεταξύ τους διεργασιών και όχι μόνο διεργασιών που σχετίζονται με σχέση γονέα - παιδιού. Οι δύο αυτές διεργασίες χαρακτηρίζονται η καθεμία από το δικό της πηγαίο κώδικα και κατά συνέπεια η καθεμία διαθέτει το δικό της εκτελέσιμο αρχείο, ενώ επικοινωνούν μεταξύ τους μέσω του επώνυμου αγωγού, εκτελούμενη η καθεμία από το δικό της ξεχωριστό τερματικό. Αυτή η κατάσταση επιδεικνύεται στο επόμενο παράδειγμα.

Στο επόμενο παράδειγμα οι δύο διεργασίες χρησιμοποιούν ως επώνυμο αγωγό το ίδιο αρχείο του συστήματος (το αρχείο /tmp/myfifo) και ως εκ τούτου θα πρέπει αμφότερες να γνωρίζουν πού βρίσκεται, έτσι ώστε να δηλώσουν τη διαδρομή του. Παρατηρήστε πως το αρχείο αυτό δημιουργείται στον κατάλογο /tmp και ως εκ τούτου αποτελεί προσωρινό βοηθητικό αρχείο, το οποίο μπορεί να διαγραφεί μετά την ολοκλήρωση της εκτέλεσης των δύο διεργασιών, αφού δεν χρειάζεται πλέον.

Εφαρμογή 1 (αρχείο side1.c)

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

```
int main() {
   int fd;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";
```

Η εφαρμογή side1 δημιουργεί το αρχείο FIFO στη θέση /tmp/myfifo με δικαιώματα πρόσβασης 666.

```
   // Creating the named file(FIFO) → mkfifo (<pathname>, <permission>)
   mkfifo (myfifo, 0666);

   char arr1[80], arr2[80];
```

Αυτός ο επαναληπτικός βρόχος λειτουργεί συνεχώς μέχρι η λειτουργία του να τερματιστεί από το χρήστη

```
   while (1) {
```

Σε κάθε κύκλο επανάληψης

ΑΡΧΙΚΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΕΓΓΡΑΦΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο εγγραφής (O_WRONLY)

```
      // Open FIFO for write only
      fd = open (myfifo, O_WRONLY);
```

Διαβάζουμε από το πληκτρολόγιο μία συμβολοσειρά με μέγιστο μήκος 80 χαρακτήρες που καταχωρείται από το χρήστη.

```
      // Take an input arr2 from user.
      // 80 is maximum length
      fgets (arr2, 80, stdin);
```

Εγγράφουμε αυτή τη συμβολοσειρά στο αρχείο το επώνυμου αγωγού και στη συνέχεια το κλείνουμε.

```
      // Write the input arr2 on FIFO and close it
      write (fd, arr2, strlen(arr2)+1);
      close (fd);
```

ΣΤΗ ΣΥΝΕΧΕΙΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΑΝΑΓΝΩΣΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο ανάγνωσης (O_RDONLY)

```
      // Open FIFO for Read only
      fd = open (myfifo, O_RDONLY);
```

Διαβάζουμε από το αρχείο του επώνυμου αγωγού τη συμβολοσειρά που έχει εγγράψει η άλλη διεργασία, εκτυπώνουμε τη συμβολοσειρά στην οθόνη του τερματικού μας και στη συνέχεια κλείνουμε τον αγωγό.

```
      // Read from FIFO
      read (fd, arr1, sizeof(arr1));
      // Print the read message
      Printf ("User2: %s\n", arr1);

      close(fd); }
   return 0; }
```

Εφαρμογή 2 (αρχείο side2.c)

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

int main() {
   int fd1;
```

Η εφαρμογή side2 δημιουργεί το αρχείο FIFO στη θέση /tmp/myfifo με δικαιώματα πρόσβασης 666.

```
   // FIFO file path
   char * myfifo = "/tmp/myfifo";

   // Creating the named file(FIFO) → mkfifo (<pathname>,<permission>)

   mkfifo(myfifo, 0666);

   char str1[80], str2[80];
```

Αυτός ο επαναληπτικός βρόχος λειτουργεί συνεχώς μέχρι η λειτουργία του να τερματιστεί από το χρήστη

```
   while (1) {
```

ΑΡΧΙΚΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΑΝΑΓΝΩΣΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο ανάγνωσης (O_RDONLY)

```
      // First open in read only and read
      fd1 = open (myfifo,O_RDONLY);
```

Διαβάζουμε από το αρχείο του επώνυμου αγωγού τη συμβολοσειρά που έχει εγγράψει η άλλη διεργασία, εκτυπώνουμε τη συμβολοσειρά στην οθόνη του τερματικού μας και στη συνέχεια κλείνουμε τον αγωγό.

```
      read (fd1, str1, 80);

       // Print the read string and close
      printf("User1: %s\n", str1);

      close(fd1);
```

ΣΤΗ ΣΥΝΕΧΕΙΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΕΓΓΡΑΦΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο εγγραφής (O_WRONLY)

```
      // Now open in write mode and write string taken from user.
      fd1 = open(myfifo,O_WRONLY);
```

Διαβάζουμε από το πληκτρολόγιο μία συμβολοσειρά με μέγιστο μήκος 80 χαρακτήρες που καταχωρείται από το χρήστη.

```
      fgets(str2, 80, stdin);
```

Εγγράφουμε αυτή τη συμβολοσειρά στο αρχείο το επώνυμου αγωγού και στη συνέχεια το κλείνουμε.

```
    write (fd1, str2, strlen(str2)+1);
    close(fd1); }
return 0; }
```

Παρατηρήστε πως οι δύο διεργασίες είναι παρόμοιες, αλλά η μία πραγματοποιεί την αντίστροφη λειτουργία της άλλης: ενώ η πρώτη διεργασία πραγματοποιεί πρώτα εγγραφή και μετά ανάγνωση, η δεύτερη διεργασία πραγματοποιεί πρώτα ανάγνωση και μετά εγγραφή. Για την εκτέλεση της εφαρμογής θα πρέπει να μεταγλωττιστούν οι δύο διεργασίες ξεχωριστά, η μία μετά την άλλη, να δημιουργηθούν τα δύο εκτελέσιμα αρχεία και στη συνέχεια να εκτελέσουν τον δικό του κώδικα το καθένα και από το δικό του τερματικό. Η λειτουργία της εφαρμογής απαιτεί την καταχώρηση ενός μηνύματος πρώτα από τη διεργασία 1. Αυτό το μήνυμα θα εμφανιστεί στο τερματικό της διεργασίας 2, η οποία στη συνέχεια μπορεί να καταχωρήσει το δικό της μήνυμα, το οποίο θα εμφανιστεί με τη σειρά του στο τερματικό της διεργασίας 1. Με τον τρόπο αυτό, οι δύο διεργασίες ανταλλάσσουν μηνύματα μεταξύ τους, προσομοιώνοντας τη λειτουργία ενός chat και για όσο χρονικό διάστημα ο χρήστης επιθυμεί κάτι τέτοιο, διότι ο επαναληπτικός βρόχος εκτελείται συνεχώς και επ' άπειρον. Για να ολοκληρωθεί η εφαρμογή, θα πρέπει ο χρήστης να πατήσει σε αμφότερα τα τερματικά τον συνδυασμό πλήκτρων Ctrl-C. Παράδειγμα εξόδου αυτής της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab6$ ./side1
Hey !! Do you want to play math?
User2: Sure, why not ...

1+1 = ?
User2: 2

2+4 = ?
User2: 6

5 + 5 10?
User2: You answered !! 10

no !!! 55 !! looser ...
User2: bye bye :-(

bye !! lol
^C
amarg@amarg-vbox:~/shared/Lab6$
```

```
amarg@amarg-vbox:~/shared/LAb6$ ./side2
User1: Hey !! Do you want to play math?

Sure, why not ...
User1: 1+1 = ?

2
User1: 2+4 = ?

6
User1: 5 + 5 10?

You answered !! 10
User1: no !!! 55 !! looser ...

bye bye :-(
User1: bye !! lol

^C
amarg@amarg-vbox:~/shared/LAb6$
```