Επαναληπτικό εργαστήριο

1) Να εκτελεστούν οι εντολές (α) ps, (β) ps –x, (γ) ps –elf και να σχολιαστεί το αποτέλεσμα. Για πληροφορίες σχετικά με τις επιλογές που χρησιμοποιούνται κατά την κλήση της εντολής, ανατρέξτε στη σελίδα βοήθειας της εντολής ps που εμφανίζεται εκτελώντας την εντολή man ps.

Η εντολή ps εκτυπώνει τα στοιχεία των διεργασιών που έχουν το ίδιο ενεργό αναγνωριστικό χρήστη (effective user id, euid) με αυτό του τρέχοντος χρήστη (δηλαδή του χρήστη που εκτελεί την εντολή και το όνομα του οποίου εκτυπώνεται από την εντολή whoami). Εάν η εντολή κληθεί χωρίς ορίσματα εκτυπώνει τις διεργασίες **που έχουν ξεκινήσει από τον τρέχοντα φλοιό**, συμπεριλαμβανομένου και του ίδιου του φλοιού.

```
amarg@amarg-vbox:~$ ps
    PID TTY          TIME CMD
   2546 pts/0    00:00:00 bash
   2637 pts/0    00:00:00 ps
```

Η εντολή ps με το διακόπτη –x εμφανίζει **όλες τις διεργασίες του συστήματος** με κάτοχο τον τρέχοντα χρήστη.

```
amarg@amarg-vbox:~$ ps -x
    PID TTY      STAT   TIME COMMAND
   1830 ?        Ss     0:00 /lib/systemd/systemd --user
   1831 ?        S      0:00 (sd-pam)
   1836 ?        S<sl   0:00 /usr/bin/pulseaudio --daemonize=no --
   1838 ?        SNsl   0:00 /usr/libexec/tracker-miner-fs
   1841 ?        Sl     0:00 /usr/bin/gnome-keyring-daemon --daemo
```

Η εντολή ps με το διακόπτη –e εμφανίζει όλες τις ενεργές διεργασίες χρησιμοποιώντας το προεπιλεγμένο στυλ εμφάνισης του Linux (generic Linux format).

```
amarg@amarg-vbox:~$ ps -e
    PID TTY          TIME CMD
      1 ?        00:00:01 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 rcu_gp
      4 ?        00:00:00 rcu_par_gp
      6 ?        00:00:00 kworker/0:0H-kblockd
      7 ?        00:00:00 kworker/u2:0-events_unbound
      8 ?        00:00:00 mm_percpu_wq
      9 ?        00:00:00 ksoftirqd/0
     10 ?        00:00:00 rcu_sched
     11 ?        00:00:00 migration/0
     12 ?        00:00:00 idle_inject/0
```

Προκειμένου να εμφανίσουμε το μέγιστο πλήθος πληροφοριών που εκτυπώνει η εντολή, χρησιμοποιούμε μαζί με το διακόπτη –e (που όπως είδαμε εμφανίζει όλες τις ενεργές διεργασίες), μαζί με τους διακόπτες –f (full listing) και –l (long format). Για να το κάνουμε αυτό γράφουμε ps –e –l –f ή πιο απλά ps –elf. Το αποτέλεσμα είναι

```
amarg@amarg-vbox:~$ ps -elf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 25532 -      09:43 ?        00:00:01 /sbin/init splash
1 S root           2       0  0  80   0 -     0 -      09:43 ?        00:00:00 [kthreadd]
1 I root           3       2  0  60 -20 -     0 -      09:43 ?        00:00:00 [rcu_gp]
1 I root           4       2  0  60 -20 -     0 -      09:43 ?        00:00:00 [rcu_par_gp]
1 I root           6       2  0  60 -20 -     0 -      09:43 ?        00:00:00 [kworker/0:0H-kblockd]
1 I root           7       2  0  80   0 -     0 -      09:43 ?        00:00:00 [kworker/u2:0-events_unbound]
1 I root           8       2  0  60 -20 -     0 -      09:43 ?        00:00:00 [mm_percpu_wq]
1 S root           9       2  0  80   0 -     0 -      09:43 ?        00:00:00 [ksoftirqd/0]
1 I root          10       2  0  80   0 -     0 -      09:43 ?        00:00:00 [rcu_sched]
1 S root          11       2  0 -40   - -     0 -      09:43 ?        00:00:00 [migration/0]
5 S root          12       2  0   9   - -     0 -      09:43 ?        00:00:00 [idle_inject/0]
1 S root          14       2  0  80   0 -     0 -      09:43 ?        00:00:00 [cpuhp/0]
5 S root          15       2  0  80   0 -     0 -      09:43 ?        00:00:00 [kdevtmpfs]
1 I root          16       2  0  60 -20 -     0 -      09:43 ?        00:00:00 [netns]
```

Οι πιο σημαντικές από τις πληροφορίες που εμφανίζονται εδώ, είναι η κατάσταση της διεργασίας, ο κωδικός της διεργασίας και της γονικής διεργασίας, η τιμή της προτεραιότητας, ο χρόνος έναρξης και ο συνολικός χρόνος και το όνομα της εντολής που δημιούργησε τη διεργασία. Για περισσότερες πληροφορίες σχετικά με τις στήλες στην έξοδο της εντολής ανατρέξτε στη διεύθυνση

https://man7.org/linux/man-pages/man1/ps.1.html

Για έναν κατάλογο με παραδείγματα χρήσης της εντολής ps ανατρέξτε στη διεύθυνση

https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/

---

2) Να δημιουργήσετε το αρχείο sample.txt με την εντολή touch.txt και τον κατάλογο test με την εντολή mkdir test. Στη συνέχεια για το παραπάνω αρχείο και τον παραπάνω κατάλογο, χρησιμοποιώντας την εντολή chmod να δώσετε τα επόμενα δικαιώματα:

i. **r w – r - - r - x ενεργοποιώντας το setuid bit**
ii. **r – x r w – r - - ενεργοποιώντας το setuid bit και το setgid bit**
iii. **r w x r – x r - - ενεργοποιώντας το setgid bit και το sticky bit**
iv. **r w - - w x r w - ενεργοποιώντας όλα τα ειδικά bits**
v. **r w – r w x – w x ενεργοποιώντας το sticky bit**
vi. **r – x rw - - r w x ενεργοποιώντας το userid bit και το sticky bit**

Κατασκευάζοντας το αρχείο sample.txt και τον κατάλογο test με τις εντολές touch και mkdir, διαπιστώνουμε πως οι μάσκες δικαιωμάτων για αυτά τα δύο αντικείμενα είναι αντίστοιχα r w – r w – r - - (664) και r w x r w x r – x (775) όπως επιβάλλεται από την τιμή της umask η οποία είναι 002. Όσον αφορά τώρα την εκχώρηση των παραπάνω δικαιωμάτων, αυτή πραγματοποιείται με την εντολή

chmod ABCD name

όπου η παράμετρος name είναι είτε το sample.txt είτε το test, ενώ η μάσκα ABCD (που τώρα έχει μήκος 4 ψηφία επειδή θα πρέπει να ορίσουμε και τα τρία ειδικά bits) κατασκευάζεται ως εξής:

Για τα εννέα δικαιώματα πρόσβασης χρησιμοποιούνται τα τρία τελευταία bits BCD, με το γνωστό πλέον τρόπο και η σωστή τιμή για τις παραπάνω περιπτώσεις είναι η εξής (τα bits της δεύτερης τριάδας είναι bold για λόγους ευκολίας στην προεπισκόπηση του κειμένου):

i. r w – r - - r – x ➔ 1 1 0 **1 0 0** 1 0 1 ➔ 6 4 5
ii. r – x r w – r - - ➔ 1 0 1 **1 1 0** 1 0 0 ➔ 5 6 4
iii. r w x r – x r - - ➔ 1 1 1 **1 0 1** 1 0 0 ➔ 7 5 4
iv. r w - - w x r w – ➔ 1 1 0 **0 1 0** 1 1 0 ➔ 6 3 6
v. r w – r w x – w x ➔ 1 1 0 **1 1 1** 0 1 0 ➔ 6 7 3
vi. r – x r w – r w x ➔ 1 0 1 **1 0 0** 1 1 1 ➔ 5 6 7

Η τιμή του A που θα τοποθετηθεί μπροστά υπολογίζεται ως εξής: έστω u το setuid bit, g το setgid bit και t το sticky bit. Το A τότε είναι ένας δυαδικός αριθμός της μορφής ugt όπου το u παίρνει την τιμή 1 εάν ενεργοποιείται το setuid bit και την τιμή 0 στην αντίθετη περίπτωση, το g παίρνει την τιμή 1 εάν ενεργοποιείται το setgid bit και την τιμή 0 στην αντίθετη περίπτωση και το το t παίρνει την τιμή 1 εάν ενεργοποιείται το sticky bit και την τιμή 0 στην αντίθετη περίπτωση. Μετά τον ορισμό των τιμών για τα τρία αυτά bits, η τιμή του A μετατρέπεται στο οκταδικό σύστημα. Κατά συνέπεια, η τιμή του A για τις παραπάνω περιπτώσεις υπολογίζεται ως εξής:

1. Ενεργοποίηση setuid bit ➔ A = 1 0 0 = 4
2. Ενεργοποίηση setuid bit και setgid bit ➔ A =1 1 0 = 6

3. Ενεργοποίηση setgid bit και sticky bit ➔ A = 0 1 1 = 3
4. Ενεργοποίηση όλων των ειδικών bits ➔ A = 1 1 1 = 7
5. Ενεργοποίηση του sticky bit ➔ A = 0 0 1 =1
6. Ενεργοποίηση του setuid bit και του sticky bit ➔ A = 1 0 1 = 5

Συνδυάζοντας τα παραπάνω αποτελέσματα, η εντολή chmod για την καθεμία από τις επόμενες περιπτώσεις θα πρέπει να κληθεί ως εξής (όπου name το sample.txt και το test για το αρχείο και τον κατάλογο αντίστοιχα):

a. r w – r - - r – x ενεργοποιώντας το setuid bit ➔ **chmod 4645 name**
b. r – x r w – r - - ενεργοποιώντας το setuid bit και το setgid bit ➔ **chmod 6564 name**
c. r w x r – x r - - ενεργοποιώντας το setgid bit και το sticky bit ➔ **chmod 3754 name**
d. r w - - w x r w – ενεργοποιώντας όλα τα ειδικά bits ➔ **chmod 7636 name**
e. r w – r w x – w x ενεργοποιώντας το sticky bit ➔ **chmod 1673 name**
f. r – x r w – r w x ενεργοποιώντας το userid bit και το sticky bit ➔ **chmod 5567 name**

Το αποτέλεσμα της διαδικασίας για το αρχείο sample.txt ακολουθεί στη συνέχεια (οι ίδιες εντολές θα εκτελεστούν και για τον κατάλογο test αντικαθιστώντας το όνομα sample.txt με το όνομα test).

```
amarg@amarg-vbox:~$ chmod 4645 sample.txt
amarg@amarg-vbox:~$ ls -l sample.txt
-rwSr--r-x 1 amarg amarg 26 Νοε   2 10:57 sample.txt
amarg@amarg-vbox:~$ chmod 6564 sample.txt
amarg@amarg-vbox:~$ ls -l sample.txt
-r-srwSr-- 1 amarg amarg 26 Νοε   2 10:57 sample.txt
amarg@amarg-vbox:~$ chmod 3754 sample.txt
amarg@amarg-vbox:~$ ls -l sample.txt
-rwxr-sr-T 1 amarg amarg 26 Νοε   2 10:57 sample.txt
amarg@amarg-vbox:~$ chmod 7636 sample.txt
amarg@amarg-vbox:~$ ls -l sample.txt
-rwS-wsrwT 1 amarg amarg 26 Νοε   2 10:57 sample.txt
amarg@amarg-vbox:~$ chmod 1673 sample.txt
amarg@amarg-vbox:~$ ls -l sample.txt
-rw-rwx-wt 1 amarg amarg 26 Νοε   2 10:57 sample.txt
amarg@amarg-vbox:~$ chmod 5567 sample.txt
amarg@amarg-vbox:~$ ls -l sample.txt
-r-srw-rwt 1 amarg amarg 26 Νοε   2 10:57 sample.txt
```

Θυμηθείτε πως ένα μικρό γράμμα (s ή t) για το ειδικό bit σημαίνει πως το bit εκτέλεσης x που βρίσκεται σε εκείνη τη θέση είναι ενεργοποιημένο, ενώ αντίθετα, ένα κεφαλαίο γράμμα (S ή T) για το ειδικό bit σημαίνει πως το bit εκτέλεσης x που βρίσκεται σε εκείνη τη θέση **δεν** είναι ενεργοποιημένο. Με τον τρόπο αυτό είναι δυνατή η κωδικοποίηση δύο πληροφοριών (περί της ενεργοποίησης ή όχι του execute bit και του sticky bit) χρησιμοποιώντας ένα απλό bit.

---

3) Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τη συνδυασμένη χρήση των συναρτήσεων fork, wait και exec.

Παράδειγμα 1. Απλό πρόγραμμα με τη γονική και τη θυγατρική διεργασία να εκτυπώνουν η καθεμία το δικό της μήνυμα (αρχείο forkex1.c).

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

```
void  main(void) {
    pid_t  pid;
```

Η συνάρτηση fork προκαλεί την κλωνοποίηση της γονικής διεργασίας και για το λόγο αυτό παρά το γεγονός πως υπάρχει μία μόνο printf, θα εκτυπωθούν δύο μηνύματα, αφού η printf θα κληθεί και από τις δύο διεργασίες.

```
    pid = fork();
    printf ("Hello !! my pid is %d\n", getpid()); }
```

Η έξοδος της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab4$ ./forkex1
Hello !! my pid is 4005
amarg@amarg-vbox:~/shared/Lab4$ Hello !! my pid is 4006
```

Παρατηρήστε πως το δεύτερο μήνυμα εκτυπώθηκε μετά την εμφάνιση του command prompt και αυτό αποτελεί ένδειξη πως η γονική διεργασία ολοκληρώθηκε πρώτη. Παρατηρήστε επίσης πως οι κωδικοί των διεργασιών έχουν συνεχόμενες τιμές (4005 και 4006) κάτι που είναι εύλογο και συμβαίνει σχεδόν πάντοτε.

Παράδειγμα 2. Σε αυτό το παράδειγμα, η χρήση της συνάρτησης wait διασφαλίζει τον τερματισμό της γονικής διεργασίας μετά τον τερματισμό της θυγατρικής διεργασίες (αρχείο forkex2.c).

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

int main(int argc, char *argv[]) {
```

Σε αυτό το σημείο καλείται η fork για τη δημιουργία της θυγατρικής διεργασίας. Η fork επιστρέφει την τιμή 0 στη θυγατρική διεργασία και τιμή διάφορη του μηδενός (και ίση με το pid της θυγατρικής διεργασίας) στη γονική διεργασία, ενώ σε περίπτωση αποτυχίας επιστρέφει την τιμή -1.

```
   pid_t child_pid = fork();

   if (child_pid < 0) {
      perror("fork() failed");
      exit(EXIT_FAILURE);
   } else if (child_pid == 0) {
```

Η θυγατρική διεργασία εκτυπώνει τις τιμές των pid (process id) και ppid (parent process id) καλώντας τις συναρτήσεις getpid () και getppid (που επιστρέφουν αυτές τις δύο τιμές).

```
      printf("from child: pid=%d, parent_pid=%d\n", (int) getpid(), (int) getppid());
```

Σε αυτό το σημείο και πριν την κλήση της exit για τον τερματισμό της διεργασίας μπορούμε να γράψουμε τον κώδικα που υλοποιεί τη διαδικασία που θα πραγματοποιεί η διεργασία.

```
      exit(33);

   } else if (child_pid > 0) {
```

Η γονική διεργασία εκτυπώνει τις τιμές των pid (process id) και ppid (parent process id) καλώντας τις συναρτήσεις getpid () και getppid (που επιστρέφουν αυτές τις δύο τιμές).

```
printf("from parent: pid=%d child_pid=%d\n", (int) getpid(), (int) child_pid);
```

Στο σημείο αυτό καλούμε τη συνάρτηση waitpid προκειμένου η γονική διεργασία να **περιμένει** τον ομαλό ή απότομο τερματισμό της θυγατρικής διεργασίας προκειμένου να τερματιστεί και η ίδια. Η waitpid επιστρέφει το process id τη θυγατρικής διεργασίας ενώ επιστρέφει πληροφορίες σχετικά με τον κώδικα επιστροφής και το λόγο του τερματισμού.

```
int status;
pid_t waited_pid = waitpid (child_pid, &status, 0);

if (waited_pid < 0) {
   perror("waitpid() failed");
   exit(EXIT_FAILURE);
} else if (waited_pid == child_pid) {
```

Η μακροεντολή WIFEXITED επιστρέφει true εάν η θυγατρική διεργασία ολοκληρώθηκε κανονικά και false στην αντίθετη περίπτωση. Στην περίπτωση του κανονικού τερματισμού, η WIFEXITED επιστρέφει τον κωδικό επιστροφής της θυγατρικής διεργασίας.

```
   if (WIFEXITED(status)) {
      printf("from parent: child exited with code %d\n", WEXITSTATUS(status)); }}}
```

Η έξοδος της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab4$ ./forkex2
from parent: pid=4057 child_pid=4058
from child: pid=4058, parent_pid=4057
from parent: child exited with code 33
amarg@amarg-vbox:~/shared/Lab4$
```

Παράδειγμα 3. Σε αυτό το παράδειγμα, η γονική και η θυγατρική διεργασία αρχικοποιούν η καθεμία το δικό της αντίγραφο των μεταβλητών local και global σε διαφορετικές τιμές (αρχείο forkex3c).

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

int global = 0;

int main() {
   pid_t child_pid;
   int status;
   int local = 0;
```

Σε αυτό το σημείο καλείται η fork για τη δημιουργία της θυγατρικής διεργασίας. Η fork επιστρέφει την τιμή 0 στη θυγατρική διεργασία και τιμή διάφορη του μηδενός (και ίση με το pid της θυγατρικής διεργασίας) στη γονική διεργασία, ενώ σε περίπτωση αποτυχίας επιστρέφει την τιμή -1.

```
child_pid = fork();

if (child_pid >= 0) /* fork succeeded */ {
```

Κώδικας για τη θυγατρική διεργασία (child_pid = 0)

```
    if (child_pid == 0) /* fork() returns 0 for the child process */ {
      printf("child process!\n");
```

Η αύξηση των τιμών των μεταβλητών κατά μία μονάδα, αφορά **μόνο** τα αντίγραφα των μεταβλητών που ανήκουν στη θυγατρική διεργασία. Αντίθετα οι τιμές των μεταβλητών που ανήκουν στη γονική διεργασία **δεν** επηρεάζονται.

```
      local++;
      global++;
      printf("child PID =  %d, parent pid = %d\n", getpid(), getppid());
      printf("\nchild's local = %d, child's global = %d\n",local,global);

      char * cmd[] = {"whoami",(char*)0};
```

Η θυγατρική διεργασία μέσα από την execv καλεί την εντολή whoami για την εκτύπωση του ονόματος του χρήστη.

```
      return execv("/usr/bin/whoami",cmd);  }
```

Κώδικας για τη γονική διεργασία (child_pid > 0)

```
    else /* parent process */ {

      printf("parent process!\n");
      printf("parent PID =  %d, child pid = %d\n", getpid(), child_pid);
```

Η γονική διεργασία καλεί τη wait για να περιμένει την ολοκλήρωση της εκτέλεσης της θυγατρικής διεργασίας πριν τερματιστεί και η ίδια και στη συνέχεια εκτυπώνει τον κωδικό εξόδου της θυγατρικής διεργασίας που επιστρέφεται από τη μακροεντολή WEXITSTATUS

```
      wait(&status); /* wait for child to exit, and store child's exit status */
      printf("Child exit code: %d\n", WEXITSTATUS(status));
```

Η εκτύπωση των τιμών των μεταβλητών της γονικής διεργασίας αποκαλύπτει ότι πράγματι η αλλαγή που έγινε προηγουμένως αφορά μόνο στα αντίγραφα των μεταβλητών της θυγατρικής διεργασίας. Αντίθετα, οι μεταβλητές της γονικής διεργασίας έχουν τις αρχικές τους μηδενικές τιμές.

```
      printf("\nParent'z local = %d, parent's  global = %d\n",local,global);
      printf("Parent says bye!\n");
      exit(0);  /* parent exits */ }}
```

Εάν η fork επιστρέψει αρνητικό αριθμό, έχει λάβει χώρα κάποιο σφάλμα το οποίο εκτυπώνεται από τη συνάρτηση perror η οποία εκτυπώνει τη λεκτική περιγραφή του σφάλματος errno.

```
   else /* failure */ {
      perror("fork");
      exit(0);  }}
```

Η έξοδος της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab4$ ./forkex3
parent process!
parent PID =  4191, child pid = 4192
child process!
child PID =  4192, parent pid = 4191

child's local = 1, child's global = 1
amarg
Child exit code: 0

Parent'z local = 0, parent's  global = 0
Parent says bye!
```

Παράδειγμα 4. Δημιουργία τοπολογίας διεργασιών δύο επιπέδων με τη γονική διεργασία να δημιουργεί δύο θυγατρικές διεργασίες (αρχείο forkex4.c).

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

Το γεγονός πως στην προκειμένη περίπτωση θα δημιουργηθούν δύο θυγατρικές διεργασίες, σημαίνει πως η συνάρτηση fork θα πρέπει να κληθεί δύο φορές.

```
int main () {
   int pid1, pid2, status1, status2, child1, child2;
```

Η πρώτη κλήση της fork

```
   pid1 = fork();
   if (pid1<0) {
      printf ("Fork operation was uncussesfull\n");
      return -1 ; }
   else if (pid1>0) {
        printf ("Parent process with pid %d and ppid %d \n",getpid(), getppid());
        child1 = wait(&status1);
        printf ("Child with code %d has beem terminated\n", child1);
```

Η δεύτερη κλήση της fork – προκειμένου οι δύο θυγατρικές διεργασίες να ανήκουν στο ίδιο επίπεδο, δηλαδή αμφότερες να αποτελούν άμεσα παιδιά της γονικής διεργασίας, θα πρέπει η δεύτερη fork να τοποθετηθεί στον κώδικα της γονικής διεργασίας.

```
        pid2 = fork();
```

Αυτός ο κώδικας όπως και πριν εκτελείται από τη γονική διεργασία

```
    if (pid2>0)
      {
        printf ("Parent process \n");
        child2 = wait (&status2);
        printf ("Child with code %d has been terminated\n", child2); }
    else {
```

Αυτός ο κώδικας εκτελείται από την πρώτη θυγατρική διεργασία. Αυτή η διεργασία καλεί τη συνάρτηση execl μέσα από την οποία εκτελεί την εντολή ***pwd*** του λειτουργικού συστήματος.

```
        printf ("Child2 with pid %d and ppid %d \n", getpid(), getppid());
        printf ("Calling pwd command...");
        execl ("/usr/bin/pwd","pwd",NULL); }}
  else {
```

Αυτός ο κώδικας εκτελείται από τη δεύτερη θυγατρική διεργασία. Αυτή η διεργασία καλεί τη συνάρτηση execl μέσα από την οποία εκτελεί την εντολή ***mkdir*** του λειτουργικού συστήματος για την κατασκευή του καταλόγου OSLab.

```
      printf ("Child1 with pid %d kai ppid %d \n", getpid(), getppid());
      printf ("Creating a new directory...\n");
      execlp ("mkdir", "mkdir", "OSLab", NULL); }}
```

Η έξοδος της εφαρμογής ακολουθεί στη συνέχεια.

```
Parent process with pid 4269 and ppid 3627
Child1 with pid 4270 kai ppid 4269
Creating a new directory...
Child with code 4270 has beem terminated
Parent process
Child2 with pid 4271 and ppid 4269
/home/amarg/shared/Lab4
Child with code 4271 has been terminated
```

Παρατηρήστε πως οι κωδικοί της γονικής (4269) και των δύο θυγατρικών διεργασιών (4270 και 4271) είναι συνεχόμενοι, κάτι που είναι αναμενόμενο. Από την άλλη πλευρά ο κωδικός του γονέα της γονικής διεργασίας είναι ο 3627. Ποιος είναι ο γονέας της γονικής διεργασίας? Μελετώντας την έξοδο της εντολής ps

```
amarg@amarg-vbox:~/shared/Lab4$ ps
    PID TTY          TIME CMD
   3627 pts/0    00:00:00 bash
```

διαπιστώνουμε πως το 3627 είναι το PID του φλοιού bash ο οποίος επομένως είναι ο γονέας της γονικής διεργασίας. Το γεγονός αυτό είναι αναμενόμενο, αφού για να δημιουργήσουμε τη γονική διεργασία εκτελέσαμε την εφαρμογή από τη γραμμή εντολών του λειτουργικού, δηλαδή ουσιαστικά από το φλοιό bash. Ας αναφέρουμε παρεμπιπτόντως πως η γονική διεργασία του φλοιού bash είναι ο terminal server που χρησιμοποιείται σε κάθε περίπτωση όπως φαίνεται από την εντολή ps –elf

```
0 S amarg     2493    1830  0  80   0 - 40566 poll_s 09:45 ?        00:00:00 /usr/libexec/gvfsd-metadata
0 S amarg     2496    2086  0  80   0 - 105697 poll_s 09:45 ?       00:00:00 update-notifier
0 S amarg     2538    1830  0  80   0 - 204970 poll_s 09:48 ?       00:00:29 /usr/libexec/gnome-terminal-server
0 S amarg     3627    2538  0  80   0 -  2753 do_wai 12:10 pts/0    00:00:00 bash
0 T amarg     3707    3627  0  80   0 -  2343 do_sig 12:30 pts/0    00:00:00 ls --color=auto -lR /
```

γεγονός που και αυτό είναι αναμενόμενο, αφού για να χρησιμοποιήσουμε το φλοιό του λειτουργικού συστήματος θα πρέπει να χρησιμοποιήσουμε ένα τερματικό.

Παράδειγμα 5. Δημιουργία τοπολογίας διεργασιών τριών επιπέδων με τη γονική και τη θυγατρική διεργασία να δημιουργούν η καθεμία θυγατρική διεργασία (αρχείο forkex5.c).

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

Όπως και στην προηγούμενη άσκηση, το γεγονός πως στην προκειμένη περίπτωση θα δημιουργηθούν δύο θυγατρικές διεργασίες, σημαίνει πως η συνάρτηση fork θα πρέπει να κληθεί δύο φορές.

```
int main () {
   int pid1, pid2, status1, status2, child1, child2;
   pid1 = fork();
   if (pid1<0) {
      printf ("Fork operation was uncussesfull\n");
      return -1 ; }
   else if (pid1>0) {
       printf ("Parent process with pid %d and ppid %d \n",getpid(), getppid());
       child1 = wait(&status1);
       printf ("Child with code %d has beem terminated\n", child1);
       }
   else {
```

Ωστόσο, στην προκειμένη περίπτωση, η δεύτερη κλήση της fork – προκειμένου οι δύο θυγατρικές διεργασίες να σχετίζονται μέσω μίας σχέσης πατέρα / παιδιού και κατά συνέπεια η μία διεργασία να αποτελεί παιδί της μίας και γονέας της άλλης - θα πρέπει η δεύτερη fork να τοποθετηθεί στον κώδικα της γονικής διεργασίας.

```
        printf ("Child1 with pid %d kai ppid %d \n", getpid(), getppid());
        pid2 = fork();
        if (pid2>0)
          {
            printf ("Parent process (child1) for child2\n");
            child2 = wait(&status2);
            printf ("Child with code %d has been terminated\n", child2); }
       else {
            printf ("Child2 with pid %d and ppid %d \n", getpid(), getppid());
            printf ("Calling pwd command...");
            execl ("/usr/bin/pwd","pwd",NULL); }

        printf ("Creating a new directory...\n");
        execlp ("mkdir", "mkdir", "OSLab", NULL); }}
```

Η έξοδος της εφαρμογής ακολουθεί στη συνέχεια. Οι δύο νέες διεργασίες λειτουργούν όπως και πριν και κατά συνέπεια, η δημιουργία του καταλόγου OSLab δεν είναι δυνατή, αφού αυτός ο κατάλογος είχε δημιουργηθεί προηγουμένως, με αποτέλεσμα την εμφάνιση του κατάλληλου μηνύματος σφάλματος.

```
amarg@amarg-vbox:~/shared/Lab4$ ./forkex5
Parent process with pid 4892 and ppid 3627
Child1 with pid 4893 kai ppid 4892
Parent process (child1) for child2
Child2 with pid 4894 and ppid 4893
/home/amarg/shared/Lab4
Child with code 4894 has been terminated
Creating a new directory...
mkdir: cannot create directory ‘OSLab’: File exists
Child with code 4893 has beem terminated
amarg@amarg-vbox:~/shared/Lab4$
```

---

4) Υπολογισμός μεγέθους αρχείου με την lseek (αρχείο fsize1.c).

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

int main(int argc, char * argv[]) {
   int fd;
   off_t filelength;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **fsize1 name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc !=2) {

      printf ("Usage: fsize1 pathname\n");
      return (-1); }
```

Προκειμένου η εντολή να λειτουργήσει σωστά, θα πρέπει το αρχείο που όρισε ο χρήστης **να είναι σε θέση να ανοίξει σε κατάσταση μόνο ανάγνωσης** κάτι που γίνεται καλώντας την εντολή **open** με όρισμα **O_RDONLY**. Εάν η εντολή open επιστρέψει τιμή μικρότερη του μηδενός, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα (π.χ. το αρχείο δεν υπάρχει ή δεν είναι επιτρεπτή η πρόσβαση σε αυτό). Η εφαρμογή λοιπόν δεν μπορεί να συνεχίσει και για το λόγο αυτό καλεί την **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος και τερματίζει τη λειτουργία της.

```
   fd = open(argv[1], O_RDONLY );
   if (fd < 0) {
      printf("Error Number : %d\n", errno);
      perror("Error Description");
      return (-2); }
```

Η λογική που κρύβεται πίσω από τη χρήση της **lseek** για τον υπολογισμό του μεγέθους του αρχείου, είναι πως **αυτό το μέγεθος είναι ίσο με την απόσταση σε bytes ανάμεσα στην αρχή και στο τέλος του αρχείου**. Η συνάρτηση lseek μεταφέρει τον δείκτη τρέχουσας θέσης σε οποιαδήποτε θέση στο εσωτερικό του αρχείου (μην ξεχνάτε πως μιλάμε για **αρχεία τυχαίας προσπέλασης** που επιτρέπουν την μετακίνησή μας σε ένα και μόνο βήμα σε οποιαδήποτε θέση του αρχείου) και επιστρέφει **την απόσταση σε bytes ανάμεσα στην αρχή του αρχείου και στην τρέχουσα θέση**. Εάν λοιπόν μεταφερθούμε στο τέλος του αρχείου (δηλαδή αν ορίσουμε ως τρέχουσα θέση το τέλος του αρχείου), τότε η τιμή που επιστρέφεται από την lseek είναι η απόσταση ανάμεσα στην αρχή και στο τέλος του αρχείου, δηλαδή το **μέγεθος** του αρχείου σε bytes.

Η κλήση της συνάρτησης lseek με τη μορφή lseek (fd, A, SEEK_END) ορίζει ως τρέχουσα θέση για το αρχείο που περιγράφεται από τον περιγραφέα αρχείου fd, εκείνη που απέχει A bytes από το τέλος του αρχείου (SEEK_END). Εάν λοιπόν είναι A=0 δηλαδή

εάν καλέσουμε την εντολή με τη μορφή **lseek (fd, 0, SEEK_END)**, τότε ως τρέχουσα θέση ορίζουμε το τέλος του αρχείου. Στην περίπτωση αυτή η τιμή filelength που επιστρέφεται από τη συνάρτηση lseek είναι σύμφωνα με τα όσα αναφέραμε παραπάνω, το μέγεθος του αρχείου.

```
filelength = lseek (fd, 0, SEEK_END);
```

Εάν η επιστρεφόμενη τιμή της lseek είναι αρνητική, έχει λάβει χώρα κάποιο σφάλμα και εκτυπώνεται το κατάλληλο μήνυμα σφάλματος, ενώ στην αντίθετη περίπτωση εκτυπώνεται η επιστρεφόμενη τιμή της lseek που είναι το μέγεθος του αρχείου.

```
if (filelength < 0) {
    printf("Error Number : %d\n", errno);
    perror("Error Description"); }
else printf ("Size of file %s is %d bytes\n", argv[1], (int)filelength);
close (fd);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 ~/sample.c
Size of file /home/amarg/sample.c is 1238 bytes
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 ~/sample.o
Size of file /home/amarg/sample.o is 8888 bytes
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 ~/sample
Size of file /home/amarg/sample is 20032 bytes
```

---

5) Υπολογισμός μεγέθους αρχείου με την read (αρχείο fsize2.c).

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

#define BUFSIZE 100

int main (int argc, char *argv[]) {
   int fd, nread, total=0;
   char buf[BUFSIZE];
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **fsize2 name**, όπου name το όνομα ενός αρχείου ή καταλόγου. Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με 2 (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 2, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc !=2) {
   printf ("Usage: flength pathname\n");
   return (-1); }
```

Η λογική που κρύβεται πίσω από τη χρήση της **read** για τον υπολογισμό του μεγέθους του αρχείου, είναι **πως το μέγεθος του αρχείου είναι ίσο με το πλήθος των bytes που είναι αποθηκευμένα σε αυτό**. Εάν λοιπόν διαβάσουμε όλα τα bytes που υπάρχουν αποθηκευμένα στο αρχείο και μετρήσουμε το πλήθος τους, ο αριθμός που θα βρούμε είναι το μέγεθος του αρχείου. Προκειμένου η εντολή να λειτουργήσει σωστά, θα πρέπει το αρχείο που όρισε ο χρήστης **να είναι σε θέση να ανοίξει σε κατάσταση μόνο ανάγνωσης** κάτι που γίνεται καλώντας την εντολή **open** με όρισμα **O_RDONLY**. Εάν η εντολή open επιστρέψει τιμή μικρότερη του μηδενός, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα (π.χ. το αρχείο δεν υπάρχει ή δεν είναι επιτρεπτή

η πρόσβαση σε αυτό). Η εφαρμογή λοιπόν δεν μπορεί να συνεχίσει και για το λόγο αυτό καλεί την **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος και τερματίζει τη λειτουργία της.

```
fd = open(argv[1], O_RDONLY );
if (fd < 0) {
    printf("Error Number : %d\n", errno);
    perror("Error Description");
    return (-2);
```

Αν και μπορούμε να διαβάσουμε **ένα byte κάθε φορά**, ωστόσο για να επιταχύνουμε τη διαδικασία επιλέγουμε να διαβάζουμε **BUFSIZE-1 bytes** κάθε φορά, τα οποία αποθηκεύονται στη μεταβλητή **buf**. Η συνάρτηση **read** επιστρέφει το πλήθος των bytes που διάβασε και καλείται μέσα από ένα βρόχο που εκτελείται επ' άπειρον. Σε κάθε κύκλο αυτής της επαναληπτικής διαδικασίας ανάγνωσης, το πλήθος των bytes που διαβάστηκαν προστίθενται στην τιμή μιας μεταβλητής **total** η οποία αρχικά έχει τεθεί στη μηδενική τιμή και η οποία στο τέλος θα περιέχει το πλήθος των bytes που διαβάστηκαν και που όπως αναφέραμε είναι το μέγεθος του αρχείου. Ο επαναληπτικός βρόχος εκτελείται **για όσο χρονικό διάστημα υπάρχουν bytes για ανάγνωση** δηλαδή για όσο χρόνο η συνάρτηση read επιστρέφει μη μηδενική τιμή. Όταν η εν λόγω συνάρτηση επιστρέψει μηδενική τιμή, αυτό σημαίνει πως δεν βγήκε άλλα bytes για να διαβάσει και κατά συνέπεια **έχει φτάσει στο τέλος του αρχείου**. Στην περίπτωση αυτή εκτελείται η εντολή break που προκαλεί τον τερματισμό του βρόχου, ενώ η τρέχουσα τιμή της μεταβλητής total είναι το μέγεθος του αρχείου η τιμή του οποίου στη συνέχεια εκτυπώνεται στην οθόνη.

```
while (1) {
    nread = read(fd,buf,BUFSIZE-1);
    if (nread == 0){
       break; }
     /* buf[nread] = '\0';*/
    total += nread; }
printf("%d bytes total\n",total);
close(fd);
return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Παρατηρήστε πως τα μεγέθη των αρχείων που υπολογίζονται είναι τα ίδια με αυτά που υπολογίζονται από την προηγούμενη εφαρμογή, κάτι που ασφαλώς είναι αναμενόμενο.

```
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 ~/sample.c
1238 bytes total
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 ~/sample.o
8888 bytes total
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 ~/sample
20032 bytes total
amarg@amarg-vbox:~/shared/Lab5$
```

Τα ίδια αυτά μεγέθη εκτυπώνονται και από την εντολή ls –l.

```
amarg@amarg-vbox:~$ ls -l sample*
-rwxrwxr-x 1 amarg amarg 20032 Οκτ  30 11:05 sample
-rw-rw-r-- 1 sys   sys    1238 Οκτ   8 12:09 sample.c
-rw-rw-r-- 1 amarg amarg  8888 Οκτ  30 11:05 sample.o
```

---

6) Αντιγραφή αρχείου με τις read και write (αρχείο myCopy.c).

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

```
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#define BUFSIZE 512

int main(int argc, char * argv[]) {
   char buffer[BUFSIZE];
   int src, dest, errno;
   int nread, nwritten, n;
   int totalr=0, totalw=0;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **myCopy source target**, όπου **source** το όνομα του αρχείου πηγή και **target** το όνομα του αρχείου προορισμού (πράγματι, για να γίνει η αντιγραφή απαιτείται να καθορίσουμε το όνομα του αρχείου που θα αντιγράψουμε και το όνομα του αρχείου που θα προκύψει από την αντιγραφή). Εάν η εντολή κληθεί με το σωστό αυτό τρόπο, τότε η τιμή της μεταβλητής argc θα είναι ίση με **3** (δείτε τη διαφάνεια 19 της παρουσίασης 03.Shell Programming). Εάν λοιπόν η τιμή της argc δεν είναι ίση με 3, αυτό σημαίνει πως το πρόγραμμα δεν κλήθηκε με το σωστό τρόπο και ως εκ τούτου δεν μπορεί να εκτελεστεί. Για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
   if (argc!=3) {
      printf ("Usage: myCopy source destination\n");
      return (-1); }
```

Το αρχείο – πηγή που βρίσκεται στη συμβολοσειρά **argv[1]** και το περιεχόμενο του οποίου θα αντιγραφεί στο αρχείο προορισμού, ανοίγει υπό συνθήκες μόνο ανάγνωσης **(O_RDONLY**) – εάν η συνάρτηση open που χρησιμοποιείται για αυτή τη διαδικασία επιστρέψει **αρνητική** τιμή, αυτό σημαίνει πως πραγματοποιήθηκε κάποιο σφάλμα (συνηθισμένα σφάλματα είναι **να μην υπάρχει** το αρχείο ή τα δικαιώματα πρόσβασης σε αυτό να είναι τέτοια ώστε **να μην επιτρέπεται** το άνοιγμά του για ανάγνωση). Στην περίπτωση αυτή η εφαρμογή δεν μπορεί να συνεχίσει – καλεί λοιπόν τη συνάρτηση **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος που προέκυψε και τερματίζει τη λειτουργία της.

```
   src = open(argv[1], O_RDONLY);
   if (src < 0) { perror("Source file could not be opened!"); return (-1); }
```

Από την άλλη πλευρά, το αρχείο – προορισμός που βρίσκεται στη συμβολοσειρά **argv[2]** μπορεί να υπάρχει αλλά μπορεί και όχι (οπότε δημιουργείται λόγω του ορίσματος **O_CREATE**). Το όρισμα **O_TRUNC** προκαλεί το **μηδενισμό** του μεγέθους του αρχείου προορισμού εάν αυτό υπάρχει και έχει ανοίξει με επιτυχία υπό καθεστώς O_WRONLY (δηλαδή μόνο για **εγγραφή**) ενώ τα δύο τελευταία ορίσματα **S_IRUSR** και **S_IWUSR** ενεργοποιούν τα bits ανάγνωσης και εγγραφής της μάσκας δικαιωμάτων για τον κάτοχο του αρχείου.

```
   dest = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
```

Εάν η συνάρτηση open που χρησιμοποιείται για αυτή τη διαδικασία επιστρέψει **αρνητική** τιμή, αυτό σημαίνει πως πραγματοποιήθηκε κάποιο σφάλμα Στην περίπτωση αυτή η εφαρμογή δεν μπορεί να συνεχίσει – καλεί λοιπόν τη συνάρτηση **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος που προέκυψε και τερματίζει τη λειτουργία της

```
   if (dest < 0) {
      perror("Target file could not be opened!");
      return (-2); }
```

Με ποιο τρόπο πραγματοποιείται η αντιγραφή του αρχείου? πολύ απλά **διαβάζοντας bytes από το αρχείο –πηγή και εγγράφοντάς τα στο αρχείο προορισμού**, για όσο χρονικό διάστημα **υπάρχουν** bytes για ανάγνωση. Αυτή η διαδικασία πραγματοποιείται μέσα από ένα βρόχο επανάληψης ο οποίος εκτελείται για όσο το πλήθος των bytes που διάβασε η read από το αρχείο πηγή και επιστρέφεται στη μεταβλητή nread είναι **μεγαλύτερος του μηδενός**. Η διαδικασία εγγραφής των bytes που διαβάστηκαν στο αρχείο προορισμού γίνεται με ένα ένθετο βρόχο **do { … } while ( )** οποίος εκτελείται **για όσο χρονικό διάστημα το πλήθος των bytes που έχουν εγγραφεί στο αρχείο προορισμού είναι μικρότερο από το πλήθος των bytes που διαβάστηκαν κατά την τρέχουσα επανάληψη του εξωτερικού βρόχου**. Εάν όλα τα bytes που διαβάστηκαν έχουν εγγραφεί

στο αρχείο προορισμού, πραγματοποιείται η επόμενη επανάληψη του εξωτερικού βρόχου μέχρι τελικά το σύνολο των bytes του αρχείου πηγή να εγγραφεί στο αρχείο προορισμού, συνθήκη που σηματοδοτεί και την ολοκλήρωση της διαδικασίας αντιγραφής του αρχείου.

```
while ((nread = read(src, buffer, BUFSIZE))>0) {
            nwritten=0;
        do {
           n = write (dest, &buffer[nwritten], nread-nwritten);
           nwritten += n;
          } while (nwritten<nread); }

close (src);  close (dest);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Το αρχείο - πηγή hello1 δημιουργείται από την ανακατεύθυνση εξόδου της εντολής **echo** και περιέχει το μήνυμα **Hello world**. Στη συνέχεια καλείται η **myCopy** που αντιγράφει το αρχείο hello1 στο αρχείο **hello2** το οποίο είναι πανομοιότυπο με το αρχείο hello1 όπως διαπιστώνεται στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab5$ echo "Hello world" > hello1
amarg@amarg-vbox:~/shared/Lab5$ cat hello1
Hello world
amarg@amarg-vbox:~/shared/Lab5$ ./myCopy hello1 hello2
amarg@amarg-vbox:~/shared/Lab5$ ls -l hello*
-rwxrwxrwx 1 root root 12 Νοε   5 18:04 hello1
-rwxrwxrwx 1 root root 12 Νοε   5 18:04 hello2
amarg@amarg-vbox:~/shared/Lab5$ cat hello2
Hello world
amarg@amarg-vbox:~/shared/Lab5$
```

---

7) 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
```

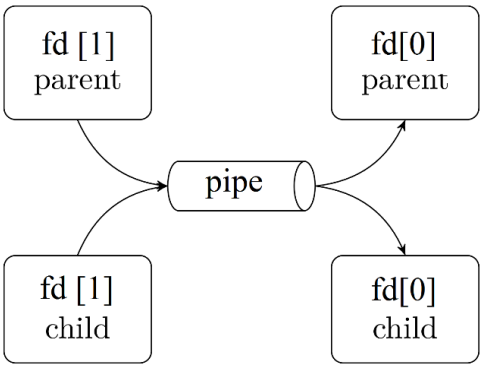
Η γονική διεργασία αποστέλλει την έξοδο της στη θυγατρική διεργασία σε ένα και μόνο βήμα, δηλαδή αποστέλλει όλο το μήνυμα με τη μία, χρησιμοποιώντας ως όρισμα στη συνάρτηση write τη συμβολοσειρά msg.

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

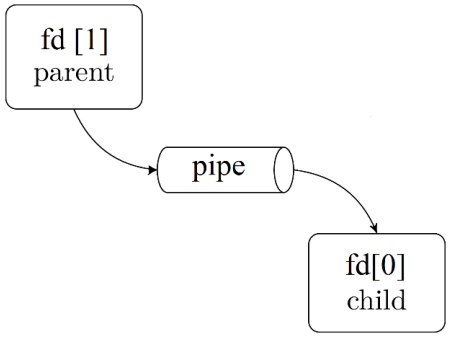
Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
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.

---

8) 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$
```

---

9) 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$
```

---

10) 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$
```

---

11) Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τη χρήση των σημάτων ως μέσο επικοινωνία μεταξύ διεργασιών.

Παράδειγμα 1. Σύλληψη του σήματος SIGINT (αρχείο sigExample1.c).

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include <stdlib.h>
```

Ο χειριστής του σήματος SIGINT το μόνο που κάνει είναι να εκτυπώσει απλά ένα ενημερωτικό μήνυμα ότι το εν λόγω σήμα έχει παραληφθεί. Με άλλα λόγια *τροποποιείται η προεπιλεγμένη συμπεριφορά* σύμφωνα με την οποία η παραλαβή αυτού του σήματος προκαλεί τη διακοπή της διεργασίας.

```
void sig_handler(int signo) {
   if (signo == SIGINT)
      printf("received SIGINT\n"); }

int main(void) {
```

Η συνάρτηση signal συσχετίζει το σήμα SIGINT με το χειριστή handler και σε περίπτωση που αυτό δεν συμβεί επιστρέφει τον κωδικό σφάλματος SIG_ERR.

```
   if (signal(SIGINT, sig_handler) == SIG_ERR)
      printf("\ncan't catch SIGINT\n");
  // A long long wait so that we can easily
  // issue a signal to this process
```

Η ανταλλαγή σημάτων ανάμεσα στις διεργασίες είναι ταχύτατη και πραγματοποιείται σε απειροελάχιστο χρονικό διάστημα για τα δεδομένα του χρήστη. Προκειμένου να είναι δυνατή η αποστολή ενός σήματος από το χρήστη στη διεργασία, εκτελούμε ένα βρόχο με άπειρες επαναλήψεις έτσι ώστε να διασφαλίσουμε ότι η διεργασία θα συλλάβει το σήμα που απέστειλε ο χρήστης. Το σήμα SIGINT δεν προκαλεί τη διακοπή της διεργασίας αφού ο χειριστής handler απλά εκτυπώνει ένα μήνυμα και ο μοναδικός τρόπος για να συμβεί αυτό είναι να ανοίξουμε ένα τερματικό, να προσδιορίσουμε τον κωδικό της διεργασίας με την ps και να της στείλουμε το σήμα SIGKILL (9) (ανοίξτε λοιπόν ένα δεύτερο terminal, εκτελέστε την εντολή ps –x για να βρείτε το PID της διεργασίας και μετά τερματίστε την με την εντολή kill -9 PID – δείτε την έξοδο που ακκολουθεί).

```
  while(1)
     sleep(1);
  return 0; }
```

Αυτή η συμπεριφορά επιδεικνύεται στη συνέχεια.

Παράδειγμα 2. Αδυναμία σύλληψης των σημάτων SIGKILL και SIGSTOP
(αρχείο sigExample2.c).

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include <stdlib.h>
```

Σε αυτό το παράδειγμα επιδεικνύεται ένα ενδιαφέρον χαρακτηριστικό του λειτουργικού συστήματος, σύμφωνα με το οποίο τα σήματα SIGKILL και SIGSTOP δεν μπορούν να υποστούν χειρισμό από το χρήστη (υπό την έννοια πως δεν μπορεί να μεταβληθεί η προεπιλεγμένη συμπεριφορά τους), παρά μόνο από τον πυρήνα. Αυτό γίνεται διότι αυτά τα σήματα επιτρέπουν τον βίαιο τερματισμό μιας διεργασίας η οποία για κάποιο λόγο έχει σταματήσει

να αποκρίνεται. Δεν είναι δύσκολο να γίνει αντιληπτό πως εάν αναθέσουμε στη διεργασία το χειρισμό αυτών των δύο σημάτων και η διεργασία σταματήσει να αποκρίνεται τότε δεν υπάρχει κανένας τρόπος για να τερματιστεί.

Ο χειριστής των σημάτων SIGKILL και SIGSTOP το μόνο που κάνει για το καθένα από αυτά, είναι να εκτυπώσει απλά ένα ενημερωτικό μήνυμα ότι το εν λόγω σήμα έχει παραληφθεί. Με άλλα λόγια *τροποποιείται η προεπιλεγμένη συμπεριφορά* σύμφωνα με την οποία η παραλαβή αυτού του σήματος προκαλεί τον απότομο τερματισμό της διεργασίας.

```
void sig_handler(int signo) {
  if (signo == SIGKILL)
    printf("received SIGKILL\n");
  if (signo == SIGSTOP)
    printf("received SIGSTOP\n"); }

int main(void) {
```

Το γεγονός πως το λειτουργικό σύστημα δεν επιτρέπει το χειρισμό των δύο αυτών σημάτων από τη διεργασία, παρουσιάζεται στην πράξη ελέγχοντας τον κωδικό επιστροφή της συνάρτησης signal. Εάν αυτός ο κωδικός είναι ίσος με SIG_ERR, αυτό σημαίνει πως δεν κατέστη δυνατή η συσχέτιση του καθενός από αυτά τα σήματα με την αντίστοιχη συνάρτηση χειρισμού, επειδή πολύ απλά το λειτουργικό σύστημα απαγόρευσε την πραγματοποίηση αυτής της συσχέτισης. Η εκτέλεση του κώδικα αποκαλύπτει πώς τα πράγματα είναι όντως έτσι. Τα δύο λοιπόν αυτά σήματα, δεν μπορούν να εκχωρηθούν για χειρισμό στις διεργασίες του χρήστη, αφού αυτό αποτελεί καθήκον του πυρήνα και μόνο του πυρήνα.

```
  if (signal(SIGKILL, sig_handler) == SIG_ERR)
    printf("\ncan't catch SIGKILL\n");
  if (signal(SIGSTOP, sig_handler) == SIG_ERR)
    printf("\ncan't catch SIGSTOP\n");
 // A long long wait so that we can easily
 // issue a signal to this process
```

Όπως έχει ήδη αναφερθεί, η ανταλλαγή σημάτων ανάμεσα στις διεργασίες είναι ταχύτατη και πραγματοποιείται σε απειροελάχιστο χρονικό διάστημα και για το λόγο αυτό, όπως και πριν προκειμένου να είναι δυνατή η αποστολή ενός σήματος από το χρήστη στη διεργασία, εκτελούμε ένα βρόχο με άπειρες επαναλήψεις έτσι ώστε να διασφαλίσουμε ότι η διεργασία θα συλλάβει το σήμα που απέστειλε ο χρήστης.

```
 while(1)
   sleep(1);
 return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Η εφαρμογή όπως και πριν τερματίζεται με την εντολή kill -9 PID όπου PID ο κωδικός της διεργασίας που μπορεί να βρεθεί με την εντολή ps –x. Ωστόσο, τώρα μπορούμε να χρησιμοποιήσουμε και το συνδυασμό Ctrl-C αφού δεν τροποποιήσαμε τη συμπεριφορά της διεργασίας όταν δέχεται το σήμα SIGINT.

```
  59216 pts/0    Ss+    0:00 bash
  59697 ?        Sl     0:08 /snap/snap-store/481/usr
  60537 pts/1    Ss     0:00 bash
  61270 pts/1    S+     0:00 ./sigExample2
  61278 pts/2    Ss     0:00 bash
  61289 pts/2    R+     0:00 ps -x
amarg@amarg-vbox:~/shared/Lab7$ kill -9 61270
amarg@amarg-vbox:~/shared/Lab7$
```

```
amarg@amarg-vbox:~/shared/Lab7$ ./sigExample2

can't catch SIGKILL

can't catch SIGSTOP
Killed
amarg@amarg-vbox:~/shared/Lab7$
```

Παράδειγμα 4. Ανταλλαγή σήματος μεταξύ γονικής και θυγατρικής διεργασίας – Παράδειγμα Α (αρχείο sigExample4.c).

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

Αυτό το απλό παράδειγμα περιλαμβάνει τη χρήση μιας γονικής και μία θυγατρικής διεργασίας. Η γονική διεργασία δημιουργεί τη θυγατρική διεργασία καλώντας τη συνάρτηση fork και στη συνέχεια της αποστέλλει διαδοχικά τα σήματα SIGHUP, SIGINT και SIGQUIT. Στην προκειμένη περίπτωση υπάρχουν τρεις διεργασίες χειρισμού, μία διεργασία για κάθε σήμα, οι οποίες απλά εκτυπώνουν ένα ενημερωτικό μήνυμα σύμφωνα με το οποίο παρέλαβαν το σήμα. Η αποστολή των σημάτων γίνεται ξανά με την kill, όχι όμως με την εντολή που καλείται από το τερματικό αλλά δια της κλήσεως της ομώνυμης συνάρτησης της γλώσσας C, η οποία δέχεται ως όρισμα τον κωδικό της διαδικασίας στην οποία θα αποτελεί ένα σήμα και τον κωδικό αυτού του σήματος. Τα υπόλοιπα είναι ακριβώς ίδια με πριν.

Ο χειριστής του σήματος SIGHUP

```
void sighup() { // handler for SIGHUP
    signal(SIGHUP, sighup);
    printf("CHILD: I have received a SIGHUP\n"); }
```

Ο χειριστής του σήματος SIGINT

```
void sigint() { // handler for SIGINT
    signal(SIGINT, sigint); /* reset signal */
    printf("CHILD: I have received a SIGINT\n"); }
```

Ο χειριστής του σήματος SIGQUIT

```
void sigquit() { // handler for SIGQUIT
   printf("My DADDY has Killed me!!!\n");
   exit(0); }

void main() {
    int pid;
```

Η συσχέτιση των συναρτήσεων χειρισμού με τα κατάλληλα σήματα

```
    signal(SIGHUP, sighup);
    signal(SIGINT, sigint);
    signal(SIGQUIT, sigquit);
```

Δημιουργία της θυγατρικής διεργασίας με τη fork

```
    pid = fork ();
    if (pid < 0) {
        perror("fork");
        exit(1); }
```

Η θυγατρική διεργασία εκτελεί απλά έναν ατέρμονα βρόχο χωρίς να κάνει απολύτως τίποτε, απλά και μόνο για να παραλάβει τα σήματα από τη γονική διεργασία. Ο βρόχος for (;;); είναι ατέρμων όπως και ο while (1) { … } χρησιμοποιήσαμε στα προηγούμενα παραδείγματα.

```
    if (pid == 0) { for (;;);  }
```

Η γονική διεργασία χρησιμοποιώντας τη συνάρτηση kill της C, αποστέλλει στη θυγατρική διαδικασία τα σήματα SIGHUP, SIGINT και SIGQUIT το ένα μετά το άλλο και με καθυστέρηση τριών δευτερολέπτων ανάμεσα στις διαδοχικές αποστολές - αυτό το κάνει καλώντας τη συνάρτηση sleep(3) - προκειμένου ο χρήστης να προλάβει να αντιληφθεί τη διαδικασία παραλαβής και λήψης των σημάτων . Το τελευταίο σήμα SIGQUIT, προκαλεί και τον τερματισμό της θυγατρικής διεργασίας.

```
else {
  printf("\nPARENT: sending SIGHUP\n\n");
  kill(pid, SIGHUP);
  sleep(3); // pause for 3 secs
  printf("\nPARENT: sending SIGINT\n\n");
  kill(pid, SIGINT);
  sleep(3); // pause for 3 secs
  printf("\nPARENT: sending SIGQUIT\n\n");
  kill(pid, SIGQUIT);
  sleep(3); }}
```

Παράδειγμα εξόδου της εφαρμογής ακόλουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab7$ ./sigExample4

PARENT: sending SIGHUP

CHILD: I have received a SIGHUP

PARENT: sending SIGINT

CHILD: I have received a SIGINT

PARENT: sending SIGQUIT

My DADDY has Killed me!!!
```

Παράδειγμα 5. Ανταλλαγή σήματος μεταξύ γονικής και θυγατρικής διεργασίας – Παράδειγμα Β

(αρχείο sigExample5.c).

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

// SIGALARM = 14
// SIGCHLD =  17
```

Αυτό το παράδειγμα επιδεικνύει το γεγονός πως όταν τερματίζεται η λειτουργία μιας θυγατρικής διεργασίας η γονική διεργασία ενημερώνεται για τον τερματισμό της θυγατρικής διεργασίας λαμβάνοντας το ειδικό σήμα SIGCHILD. Πράγματι, στην εφαρμογή του παραδείγματος η θυγατρική διεργασία αποστέλλει στη γονική διεργασία μόνο το σήμα SIGALARM - ωστόσο η γονική διεργασία δεν λαμβάνει μόνο ένα, αλλά δύο σήματα, με το δεύτερο σήμα SIGCHLID που αποστέλλεται σε αυτή όταν η θυγατρική διεργασία τερματιστεί.

Ο χειριστής των σημάτων SIGALARM, SIGUSR1 και SIGCHLD (το SIGUSR δεν χρησιμοποιείται εδώ αλλά επειδή είναι σήμα γενικής χρήσεως θα μπορούσαμε εάν το επιθυμούσαμε να εκχωρήσουμε σε αυτό κάποια διαδικασία).

```
void signalHandler(int signal) {
    printf("Cought signal %d!\n",signal);
    if (signal==SIGCHLD) {
            printf("Child ended\n");
            wait(NULL); }}
```

```
int main() {
```

Η συσχέτιση της συνάρτησης χειρισμού με τα τρία σήματα.

```
    signal(SIGALRM,signalHandler);
    signal(SIGUSR1,signalHandler);
    signal(SIGCHLD,signalHandler);
```

Ο κώδικας της θυγατρικής διεργασίας. Παρατηρήστε πως η θυγατρική διεργασία στέλνει στη γονική διεργασία μόνο το σήμα SIGALARM και τίποτε άλλο.

```
    if (!fork()) {
       printf("Child running...\n");
       sleep(2);
       printf("Child sending SIGALRM...\n");
       kill (getppid(),SIGALRM);
       sleep(5);
       printf("Child exitting...\n");
       return 0; }
```

Ωστόσο, όταν η θυγατρική διεργασία ολοκληρωθεί, ο πυρήνας του λειτουργικού συστήματος στέλνει στη γονική διεργασία το σήμα SIGCHLD, προκειμένου να της γνωστοποιήσει τον τερματισμό της θυγατρικής διεργασίας. Αυτό δεν φαίνεται πουθενά στον κώδικα, διότι αποτελεί λειτουργία του πυρήνα και θα πραγματοποιηθεί σε κάθε περίπτωση, είτε το ζητήσει ο χρήστης είτε όχι. Το μόνο που έχουμε να κάνουμε (και μπορούμε να κάνουμε) είναι ένα προσδιορίσουμε τη συμπεριφορά της διεργασίας όταν λάβει αυτό το σήμα. Το μόνο που κάνουμε είναι απλά να εκτυπώσουμε στην οθόνη μας το μήνυμα Child ended.

```
    printf("Parent running, PID=%d. Press ENTER to exit.\n",getpid());
    getchar();
    printf("Parent exitting...\n");
    return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./sigExample5
Parent running, PID=1962. Press ENTER to exit.
Child running...
Child sending SIGALRM...
Cought signal 14!
Child exitting...
Cought signal 17!
Child ended

Parent exitting...
```

---

12) Σε αυτό το παράδειγμα αρχιτεκτονικής client - server με UNIX sockets, ο server λειτουργεί επαναληπτικά και επιτρέπει τη σύνδεση ενός πελάτη κάθε φορά. Ο πελάτης συνδέεται στο socket του server και αντιγράψει απλά την είσοδό του στον server ο οποίος και την εκτυπώνει.

(αρχεία un-client.c και un-server.c).

Κώδικας διακομιστή

Αρχείο un-server.c

```
// Source: https://www.danlj.org/mkj/lad/src/userver.c.html
// (Johnson & Troan, pp.298-299)

#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>
```

Σε αυτή την απλή εφαρμογή client – server, ο client χρησιμοποιώντας ένα UNIX Socket συνδέεται στον server και του αποστέλλει μηνύματα τα οποία εκτυπώνονται στην οθόνη του server. Η μεταφορά των δεδομένων γίνεται με τη συνάρτηση CopyData.

Η συνάρτηση CopyData αντιγράφει αρχεία από το socket from στο socket to μέσω μίας επαναληπτικής διαδικασίας κατά τη διάρκεια της οποίας διαβάζει ομάδες 1024 bytes κάθε φορά με τη συνάρτηση read από το άκρο from και τα γράφει με τη συνάρτηση write στο άκρο to. Ο επαναληπτικός βρόχος εκτελείται για όσο χρονικό διάστημα υπάρχουν δεδομένα για ανάγνωση και ως εκ τούτου η συνάντηση τερματίζεται όταν δεν υπάρχει τίποτε άλλο για να μεταφερθεί.

```
void copyData(int from, int to) {
   char buf[1024];
   int amount;
   while ((amount = read(from, buf, sizeof(buf))) > 0) {
     if (write(to, buf, amount) != amount) {
         perror("Message from write:");
         exit(1); }}
   if (amount < 0) {
      perror("Message from read:");
      exit(1); } }
```

Η διεργασία του διακομιστή αναμένει αιτήματα εισερχομένων συνδέσεων στο αρχείο ./sample-socket που υλοποιεί ένα Unix Socket (θυμηθείτε πως σε αυτόν τον τύπο δικτυακών υποδοχέων, ως διεύθυνση νοείται η διαδρομή στο δέντρο καταλόγων του εν λόγω αρχείου). Από τη στιγμή που λάβει χώρα η αποκατάσταση μιας σύνδεσης με ένα πελάτη, ο διακομιστής αντιγράφει δεδομένα από το socket στην προεπιλεγμένη έξοδο (ο περιγραφέας αρχείου της οποίας είναι ίσος με 1) μέχρι να λάβει χώρα τερματισμός της σύνδεσης από τον πελάτη. Από τη στιγμή που αυτή η σύνδεση τερματιστεί, ο διακομιστής δεν τερματίζει αλλά περιμένει τη σύνδεση ενός νέου πελάτη. Ωστόσο, σε κάθε περίπτωση μόνο ένας πελάτης μπορεί να συνδεθεί κάθε φορά.

```
int main(void) {
   struct sockaddr_un address;
   int sock, conn;
   socklen_t addrLength;
```

Σε αυτό το σημείο δημιουργείται το passive socket τύπου UNIX (*PF_UNIX*) για επικοινωνία με σύνδεση (*SOCK_STREAM*) στο οποίο ο διακομιστής αναμένει συνδέσεις πελατών.

```
   if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
      perror("Message from socket:");
      exit(1); }
```

Εάν το αρχείο του socket υπάρχει ήδη στο δίσκο αυτό διαγράφεται.

```
   unlink("./sample-socket");
```

Αρχικοποιούνται τα πεδία της δομής sockaddr_un (για Unix Sockets) και στη συνέχεια αυτή η δομή διαβιβάζεται ως όρισμα στην bind η οποία συσχετίζει την τοπική διεύθυνση addr (που έχει αρχικοποιηθεί προηγουμένως) με την ακέραια μεταβλητή sock που έχει επιστραφεί από την κλήση συστήματος socket.

```
address.sun_family = AF_UNIX;          /* Unix domain socket */
strcpy(address.sun_path, "./sample-socket");
addrLength = sizeof(address.sun_family) + strlen(address.sun_path);
```

Καλείται η συνάρτηση bind

```
if (bind(sock, (struct sockaddr *) &address, addrLength)){
   perror("Message from bind:");
   exit(1); }
```

Καλείται η συνάρτηση listen μέσω της οποίας ο διακομιστής αναμένει αιτήματα σύνδεσης πελάτη. Το μήκος της ουράς στην οποία τοποθετούνται τα εκκρεμή αιτήματα σύνδεσης ορίζεται ίσο με 5.

```
if (listen(sock, 5)){
   perror("Message from listen:");
   exit(1); }
```

Μέσω μίας επαναληπτικής διαδικασίας η οποία τερματίζεται μόνο όταν σταματήσει η λειτουργία του ίδιου του διακομιστή, καλείται συνεχώς η συνάρτηση accept για να διαβάσει δεδομένα από τον πελάτη. Η accept δημιουργεί ένα νέο active socket με ονομα conn που συνομιλεί με το active socket του πελάτη (το socket που δημιουργεί η accept είναι διαφορετικό από το αρχικό passive socket που χρησιμοποιήθηκε ως όρισμα στη listen) και μέσω του οποίου ο πελάτης αποστέλλει τα δεδομένα του στον διακομιστή που τον εξυπηρετεί. Η αποστολή αυτών των δεδομένων και η παραλαβή τους από το διακομιστή, γίνεται καλώντας τη συνάρτηση CopyData (conn, 1) η οποία μεταφέρει δεδομένα από το socket conn στην προεπιλεγμένη έξοδο του διακομιστή που έχει file descriptor με τιμή ίση με 1. Μετά τον τερματισμό της λειτουργίας του πελάτη, ο διακομιστής αναμένει ενεργός αναμένοντας αιτήματα σύνδεσης ενός νέου πελάτη.

```
while ((conn = accept(sock, (struct sockaddr *) &address, &addrLength)) >= 0) {
   printf("---- getting data\n");
   copyData(conn, 1);
   printf("---- done\n");
   close(conn); }
```

Εάν η κλήση της accept δεν είναι επιτυχής, η εφαρμοφή τερματίζει τη λειτουργία της με μήνυμα λάθους.

```
if (conn < 0) {
   perror("Message from accept:"); exit(1); }
close(sock);
```

Κώδικας πελάτη

Αρχείο un-client.c

```
// Source: https://www.danlj.org/mkj/lad/src/userver.c.html
// (Johnson & Troan, pp.298-299)

#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>

// Copies data from file descriptor 'from' to file descriptor 'to until nothing is left to be
// copied. Exits if an error occurs.

void copyData(int from, int to) {
   char buf[1024];
```

```
    int amount;
    while ((amount = read(from, buf, sizeof(buf))) > 0) {
      if (write(to, buf, amount) != amount) {
          perror("Message from write:"); exit(1); }}

    if (amount < 0) {
       perror("Message from read:");
       exit(1); } }
```

Ο κώδικας του πελάτη μοιάζει πάρα πολύ με τον κώδικα του διακομιστή και αυτό φυσικά είναι αναμενόμενο διότι αυτές οι δύο διεργασίες θα αρχικοποιηθούν με τον ίδιο ακριβώς τρόπο. Η συνάρτηση CopyData προφανώς θα πρέπει να αντιγραφεί και στον πηγαίο κώδικα του πελάτη (αφού ο διακομιστής και ο πελάτης αποτελούν δύο ανεξάρτητες εφαρμογές με το δικό της κώδικα η καθεμία και ως εκ τούτου η εν λόγω συνάντηση πρέπει να δηλωθεί σε αμφότερες τις εφαρμογές), ενώ προκειμένου να είναι δυνατή η επικοινωνία ανάμεσα στις δύο εφαρμογές, η συνάρτηση socket θα πρεπε σε αμφότερες να κληθεί με τον ίδιο ακριβώς τρόπο. Η δομή *sockaddr_un αρχικοποιείται και αυτή με τον ίδιο ακριβώς τρόπο, με τη διαφορά πως στην εφαρμογή του διακομιστή διαβιβάζεται ως όρισμα στη συνάρτηση bind, ενώ στην εφαρμογή του πελάτη διαβιβάζεται ως όρισμα στη συνάρτηση connect. Μετά την αποκατάσταση της σύνδεσης, καλείται η συνάρτηση CopyData προκειμένου να λάβει χώρα η ανταλλαγή των δεδομένων ανάμεσα στις διεργασίας του πελάτη και του διακομιστή. Παρατηρήστε πως ενώ στον πελάτη η CopyData καλείται μία και μοναδική φορά, στο διακομιστή καλείται μέσα από έναν επαναληπτικό βρόχο κάτι που είναι αναμενόμενο, διότι ο διακομιστής αναμένει συνεχώς αιτήματα σύνδεσης από πολλούς και διαφορετικούς πελάτες (αλλά μόνο από ένα πελάτη κάθε φορά).*

```
int main(void) {
   struct sockaddr_un address;
   int sock;
   socklen_t addrLength;
   if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0){
       perror("Message from socket:");
       exit(1); }
   address.sun_family = AF_UNIX;    /* Unix domain socket */
   strcpy(address.sun_path, "./sample-socket");
   addrLength = sizeof(address.sun_family) + strlen(address.sun_path);
   if (connect(sock, (struct sockaddr *) &address, addrLength)){
       perror("Message from connect:");
       exit(1); }
   copyData(0, sock);
   close(sock);
   return 0; }
```

Τυπικό παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Παρατηρήστε πως ενώ ο διακομιστής στο αριστερό τερματικό έχει ξεκινήσει μία και μοναδική φορά αναμένοντας συνδέσεις, στο δεξί τερματικό εμφανίζονται περισσότεροι από ένας πελάτες οι οποίοι αποστέλλουν ένα ενδεικτικό μήνυμα στο διακομιστή και στη συνέχεια τερματίζουν τη λειτουργία τους με το συνδυασμό πλήκτρων Ctrl-C. Όταν τερματιστεί η σύνδεση με ένα πελάτη δια της χρήσεως του εν λόγω συνδυασμό πλήκτρων, ο διακομιστής αποκρίνεται εκτυπώνοντας το μήνυμα done στη συνέχεια εξακολουθεί να παραμένει ενεργός αναμένοντας αιτήματα άλλων πελατών. Ο διακομιστής τερματίζεται και αυτός με τον ίδιο τρόπο, δηλαδή με την το συνδυασμό πλήκτρων Ctrl-C.

```
amarg@amarg-vbox:~$ ./un-server
---- getting data
Hello from Client1
---- done
---- getting data
Hello from Client2
---- done
---- getting data
Hello from Client3
---- done
---- getting data
Hello from Client4
---- done
^C
amarg@amarg-vbox:~$
```

```
amarg@amarg-vbox:~$ ./un-client
Hello from Client1
^C
amarg@amarg-vbox:~$ ./un-client
Hello from Client2
^C
amarg@amarg-vbox:~$ ./un-client
Hello from Client3
^C
amarg@amarg-vbox:~$ ./un-client
Hello from Client4
^C
amarg@amarg-vbox:~$
```

13) Σε αυτό το παράδειγμα αρχιτεκτονικής client – server με Unix sockets, ο server και ο client σχετίζονται με μία σχέση πατέρα (server) – παιδιού (client) που δημιουργείται μέσω της fork και επικοινωνούν στέλνοντας ένα μήνυμα ο ένας στον άλλο. (αρχείο fork-cl-srv.c).

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#define SOCKETNAME "MySocket"
```

Αυτό το δεύτερο παράδειγμα επικοινωνίας πελάτη με διακομιστή χρησιμοποιώντας Unix Sockets είναι ακριβώς το ίδιο με πριν, με τη διαφορά ότι ενώ προηγουμένως οι δύο διεργασίες ήταν εντελώς ανεξάρτητες μεταξύ τους σε αυτό το παράδειγμα σχετίζονται μεταξύ τους με σχέση πατέρα - παιδιού. Κατά συνέπεια εκτελείται ο ίδιος κώδικας.

```
int main(void) {

   struct sockaddr_un sa;
   int fd_skt, fd_client; char buf[100];
   int written; ssize_t readb;
```

Όπως και πριν, εάν το αρχείο MySocket υπάρχει από προηγούμενη εκτέλεση του κώδικα, διαγράφεται έτσι ώστε να δημιουργηθεί από την εφαρμογή

```
   (void) unlink (SOCKETNAME);
```

ενώ στη συνέχεια αρχικοποιείται η δομή *sockaddr_un sa η οποία θα χρησιμοποιηθεί στις συναρτήσεις connect του client και bind του server.*

```
   strcpy (sa.sun_path, SOCKETNAME);
   sa.sun_family = AF_UNIX;
```

Ο κώδικας της θυγατρικής διεργασίας (πελάτης)

Στο σημείο αυτό καλείται η συνάρτηση fork για τη δημιουργία της θυγατρικής διεργασίας. Εάν η επιστρεφόμενη τιμή αυτής της συνάρτησης είναι ίση με το μηδέν, τότε όπως έχουμε ήδη αναφέρει, ο κώδικας που αντιστοιχεί σε αυτή τη συνθήκη, είναι ο κώδικας της θυγατρικής διεργασίας. Στο πρώτο βήμα της διαδικασίας, η θυγατρική διεργασία καλεί τη συνάρτηση socket για να δημιουργήσει ένα socket για επικοινωνία με το διακομιστή και εφόσον όλα λειτουργήσουν χωρίς πρόβλημα, εκτυπώνει ένα ενημερωτικό μήνυμα.

```
 if (fork() == 0) { /* client */
   if ((fd_skt = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("Message from socket [client] : ");
       exit(-1); }
   printf ("[Client] ==> Socket %d has been created\n", fd_skt);
```

Επειδή όπως είναι γνωστό, η γονική και η θυγατρική διεργασία εκτελούνται ταυτόχρονα, υπάρχει περίπτωση η θυγατρική διεργασία να καλέσει τη συνάρτηση connect πριν τη δημιουργία του socket από την γονική διεργασία. Εάν συμβεί αυτό, προφανώς η επικοινωνία μεταξύ των διεργασιών δεν είναι δυνατή. Η θυγατρική διεργασία διαπιστώνει την εμφάνιση αυτής της προβληματικής κατάστασης, ελέγχοντας την επιστρεφόμενη τιμή της συνάρτησης connect και εξετάζοντας εάν αυτή είναι ίση με ENOENT. Στον κώδικα που ακολουθεί, η θυγατρική διεργασία καλεί επαναληπτικά τη συνάρτηση connect. Εάν διαπιστώσει πως η εν λόγω συνάρτηση επιστρέφει την τιμή ENOENT, αναστέλλει τη λειτουργία της για ένα δευτερόλεπτο καλώντας τη συνάρτηση sleep και προσπαθεί να συνδεθεί εκ νέου, κατά την επόμενη

επανάληψη. Όταν η γονική διεργασία δημιουργήσει το δικό της socket, τότε η συνάρτηση connect θα επιστρέψει κωδικό επιτυχίας, γεγονός που σημαίνει πως η σύνδεση ανάμεσα στις δύο διεργασίες έχει αποκατασταθεί.

```
while (connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) == -1) {
   if (errno == ENOENT) {
      sleep(1); continue; }
   else {
    perror("Message from connect [client]"); exit(-2); }}
```

Μετά την αποκατάσταση της επικοινωνίας, ο πελάτης αρχικά γράφει στο server καλώντας τη συνάρτηση write

```
printf ("[Client] ==> Connection has been established .. let us write to server\n");
written = write(fd_skt, "Hello!", 7);
if (written==-1) {
    perror("Message from write [client]");  exit(-3); }
else printf ("[Client] ==> %d bytes written to server\n", written);
```

και στη συνέχεια διαβάζει από το server καλώντας τη συνάρτηση read.

```
printf ("[Client] ==> Now let us read from server\n");
readb = read(fd_skt, buf, sizeof(buf));
if (readb==-1) {
    perror("Message from read [client]");
    exit(-4); }
else printf ("[Client] ==> %zd bytes read from server\n", readb);
printf("Client got %s\n", buf);
```

Οι διαδικασίες ανάγνωσης και εγγραφής πραγματοποιούνται χρησιμοποιώντας το socket fd που δημιούργησε η connect το οποίο μετά τον τερματισμό της επικοινωνίας κλείνει με τη συνάρτηση close.

```
close(fd_skt);
exit(0); }
```

Ο κώδικας της γονικής διεργασίας (διακομιστής)

```
else { /* server */
```

Όπως και στο προηγούμενο παράδειγμα, έτσι και στην προκειμένη περίπτωση, ο διακομιστής καλεί τη συνάρτηση socket για να δημιουργήσει το δικτυακό υποδοχέα στον οποίο θα αναμένει συνδέσεις από τους πελάτες, τη συνάρτηση bind για να συσχετίσει τη διεύθυνση sa (που έχει αρχικοποιηθεί με τον τρόπο που το κάναμε προηγουμένως) με το εν λόγω socket και τη συνάρτηση listen η οποία του επιτρέπει να μπει σε κατάσταση αναμονής αιτημάτων σύνδεσης από τους πελάτες.

```
if ((fd_skt = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
    perror("Message from socket [server]");
    exit(-1); }
printf ("[Server] ==> Socket %d has been created\n", fd_skt);
if (bind(fd_skt, (struct sockaddr *) &sa, sizeof(sa))){
   perror("Message from bind [server]");
   exit(-2); }
printf ("[Server] ==> Socket %d has been bound to address\n", fd_skt);
if (listen(fd_skt, 5)){
   perror("Message from listen [server]");
   exit(-3); }
```

```
printf ("[Server] ==> Listening for incoming messages\n");
```

Στη συνέχεια καλεί τη συνάρτηση accept για να δημιουργήσει μία σύνδεση εξυπηρέτησης με τον πελάτη.

```
if ((fd_client = accept(fd_skt, NULL, 0))<0) {
   perror("Message from accept [server]");
   exit(-4); }
```

Μετά την αποκατάσταση της επικοινωνίας, ο διακομιστής αρχικά διαβάζει από τον client καλώντας τη συνάρτηση read

```
printf ("[Server] ==> let us read from client via socket %d\n", fd_client);
  readb = read(fd_client, buf, sizeof(buf));
  if (readb == -1) {
      perror("Message from read [server] : ");
      exit(-5); }
  printf("Server got %s ==> %zd bytes read from client\n", buf, readb);
```

και στη συνέχεια γράφει στον client καλώντας τη συνάρτηση write.

```
printf ("[Server] ==> let us write to client\n");
if (write(fd_client, "Goodbye!", 9)==-1) {
    perror("Message from write [server]");
    exit(-6); }
```

Στο τέλος της διαδικασίας κλείνει τα sockets που χρησιμοποιήθηκαν.

```
close(fd_skt);
close(fd_client);
exit(0); }}
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./fork-cl-srv
[Server] ==> Socket 3 has been created
[Server] ==> Socket 3 has been bound to address
[Server] ==> Listening for incoming messages
[Client] ==> Socket 3 has been created
[Server] ==> let us read from client via socket 4
[Client] ==> Connection has been established .. let us write to server
Server got Hello! ==> 7 bytes read from client
[Server] ==> let us write to client
amarg@amarg-vbox:~$ [Client] ==> 7 bytes written to server
[Client] ==> Now let us read from server
[Client] ==> 9 bytes read from server
Client got Goodbye!
```

14) Σε αυτό το παράδειγμα αρχιτεκτονικής client – server με TCP / IP sockets, ο client αποστέλλει στον server ένα μήνυμα που παραλαμβάνεται και εκτυπώνεται από αυτόν. (αρχεία server1-tcp.c και client1-tcp.c).

Σε αυτό το παράδειγμα όπως και στο προηγούμενο ο ο πελάτης αποστέλλει ένα μήνυμα στον διακομιστή ο οποίος το παραλαμβάνει και το εκτυπώνει η διαφορά είναι που στην προκειμένη περίπτωση δεν χρησιμοποιούνται unix αλλά tcp IP sockets ωστόσο η λογική είναι ακριβώς η ίδια. Το socket που χρησιμοποιείται είναι SOCK_STREAM και κατά συνέπεια χρησιμοποιείται επικοινωνία με σύνδεση.

Κώδικας διακομιστή

Αρχείο server1-tcp.c

```
#include <stdio.h>
#include <sys/socket.h> // AF_INET and SOCK_STREAM
#include <stdlib.h>     // EXIT_SUCCESS, EXIT_FAILURE
#include <netinet/in.h> // INADDR_ANY
#include <string.h>     // bzero
#include <unistd.h>     // read and write
```

#define PORT 8080 ← αριθμός θύρας για σύνδεση = 8080 (συνήθως για proxy service)

```
int main(int argc, char const *argv[]) {
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
   char buffer[1024] = {0};
   char *hello = "Hello from server";
```

Δημιουργία του socket στο server AF_INET → IPv4, SOCK_STREAM → TCP

```
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
      perror("socket failed");
      exit(EXIT_FAILURE); }
```

Εδώ μπορούμε να ορίσουμε κάποιες παραμέτρους λειτουργίας του socket. Είναι αρκετά τεχνικό και εξειδικευμένο κομμάτι και δεν έχει νόημα να ασχοληθούμε περισσότερο σε αυτό το εισαγωγικό επίπεδο (εκείνο που γίνεται εδώ είναι να επιτραπεί η εκκίνηση πολλαπλών στιγμιότυπων διακομιστής στην ίδια θύρα υπό την προϋπόθεση που σχετίζονται με διαφορετική τοπική διεύθυνση IP).

```
   if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,   &opt, sizeof(opt))) {
      perror("setsockopt");
      exit(EXIT_FAILURE); }
```

Όπως και πριν αρχικοποιούμε με τον κατάλληλο τρόπο τα διάφορα πεδία της δομής address που τώρα είναι τύπου *sockaddr_in* . Η τιμή AF_INET υποδηλώνει πως θα χρησιμοποιηθεί το πρωτόκολλο IPv4, η τιμή *INADDR_ANY υποδηλώνει πως το socket δεν θα δέχεται αιτήσεις σύνδεσης από μία συγκεκριμένη διεύθυνση αλλά από κάθε διεύθυνση, ενώ το πεδίο port λαμβάνει την τιμή της θύρας στην οποία θα αναμένονται αιτήσεις σύνδεσης, που είναι η θύρα 8080.*

```
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = INADDR_ANY;
   address.sin_port = htons (PORT);
```

Τα υπόλοιπα είναι ακριβώς ίδια με τα προηγούμενα παραδείγματα αφού τα Unix sockets και τα TCP / IP sockets δουλεύουν με τον ίδιο ακριβώς τρόπο. Αρκετά καλούμε bind για να συσχετίσουμε το socket που δημιουργήθηκε πιο πάνω με την παραπάνω διεύθυνση. Στη συνέχεια καλούμε τη listen έτσι ώστε ο διακομιστής να είναι σε θέση να τεθεί σε κατάσταση αναμονής αιτημάτων σύνδεσης από πελάτες (παρατηρήστε πως στην προκειμένη περίπτωση το δεύτερο όρισμα είναι ισο με 3 και κατά συνέπεια στην ουρά αναμονής εκκρεμών αιτημάτων σύνδεσης μπορούν να εκχωρηθούν μέχρι τρία τέτοια αιτήματα), ενώ τέλος καλείται η accept που αποκαθιστά την επικοινωνία με τον πελάτη και επιστρέφει το socket μέσω του οποίου θα πραγματοποιηθεί η διαδικασία της επικοινωνίας (σημειώστε πως όλοι οι διακομιστές δουλεύουν παντού και πάντα με τον ίδιο ακριβώς τρόπο δηλαδή καλούνε με τη σειρά τις συναρτήσεις socket, bind, listen και accept (ενώ κάτι αντίστοιχο ισχύει και για τους πελάτες, οι οποίοι καλούν παντού και πάντα τις συναρτήσεις socket και connect) και αυτό σημαίνει πως εάν εάν κατανοηθεί πλήρως ένα παράδειγμα επικοινωνίας πελάτη διακομιστή θα κατανοηθούν και όλα τα

υπόλοιπα παρόμοια παραδείγματα αφού τα πάντα λειτουργούν ακριβώς με τον ίδιο τρόπο). Μετά την αποκατάσταση της σύνδεσης καλείται η συνάρτηση read προκειμένου ο διακομιστής να διαβάσει το μήνυμα που του αποστέλλει ο πελάτης χρησιμοποιώντας ως περιγραφέα αρχείου το socket που επέστρεψε η access. Μετά την παραλαβή του μηνύματος, αυτό εκτυπώνεται στην οθόνη.

```
if  (bind (server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    perror("bind failed");
    exit(EXIT_FAILURE); }

if (listen(server_fd, 3) < 0) {
    perror("listen");
    exit(EXIT_FAILURE); }

if ((new_socket = accept (server_fd, (struct sockaddr *)&address,
                                    (socklen_t*)&addrlen)) < 0) {
    perror("accept");
    exit(EXIT_FAILURE); }

valread = read (new_socket, buffer, 1024);

printf("%s\n",buffer );
return 0; }
```

Στην εφαρμογή του πελάτη πραγματοποιείται ακριβώς η ίδια διαδικασία και η ίδια αρχικοποίηση μεταβλητών. Με άλλα λόγια, χρησιμοποιείται και εδώ η θύρα 8080 αφού προφανώς για να επικοινωνήσουν δύο διεργασίες θα πρέπει να χρησιμοποιήσουν την ίδια θύρα. Η συνάρτηση socket καλείται ακριβώς με τις ίδιες παραμέτρους με την αντίστοιχη συνάρτηση του διακομιστή, δηλαδή με ορίσματα τις τιμές AF_INET (που υποδηλώνει τη χρήση του πρωτοκόλλου IPv4) και SOCK_STREAM που υποδηλώνει τη χρήση TCP socket και υπηρεσία με σύνδεση. Μετά την κατάλληλη αρχικοποίηση και τη δημιουργία του socket, καλείται η συνάρτηση connect για τη σύνδεση στο διακομιστή και λαμβάνει χώρα η αποστολή του μηνύματος στο διακομιστή χρησιμοποιώντας τη συνάρτηση send. Η μοναδική νέα συνάρτηση που βλέπετε εδώ είναι η *inet_pton η οποία μετατρεπει μία διεύθυνση από μορφή κειμένου (για παράδειγμα 127.0.0.1) σε δυαδική μορφή (σε έναν δυαδικό αριθμό μήκους 32 bits) προκειμένου να χρησιμοποιηθεί στη συνάρτηση connect (που απαιτεί δυαδικό όρισμα).*

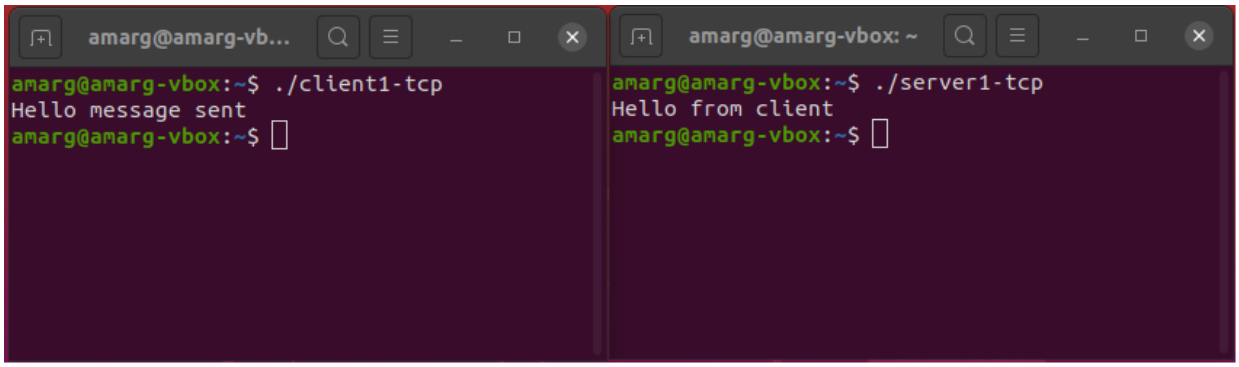
Κώδικας πελάτη

Αρχείο client1-tcp.c

```
#include <stdio.h>
#include <sys/socket.h> // AF_INET and SOCK_STREAM
#include <arpa/inet.h>  // storage size of serv_addr
#include <unistd.h>     // getpid, getppid, fork
#include <string.h>     // bzero

#define PORT 8080

int main(int argc, char const *argv[]) {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
    char buffer[1024] = {0};

    if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1; }

    serv_addr.sin_family = AF_INET;
```

```
serv_addr.sin_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary form

if (inet_pton (AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0) {
   printf("\nInvalid address/ Address not supported \n");
   return -1; }

if (connect (sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
   printf("\nConnection Failed \n");
   return -1; }

send (sock , hello , strlen(hello) , 0 );
printf("Hello message sent\n");
return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Λαμβάνοντας υπόψη πως ο πελάτης και ο διακομιστής αποτελούν δύο ανεξάρτητες μεταξύ τους διεργασίες για την καθεμία εκ των οποίων υπάρχει ξεχωριστός πηγαίος κώδικας και απαιτείται ξεχωριστή διαδικασία μεταγλώττισης και δημιουργίας του εκτελέσιμου αρχείου, είναι αυτονόητο πως οι δύο αυτές διεργασίες θα εκτελεστούν η καθεμία στο δικό της τερματικό όπως συμβαίνει συνήθως σε αυτές τις περιπτώσεις (μόνο όταν ο πελάτης και ο διακομιστής συνδέονται με σχέση γονικής - θυγατρικής διεργασίας εκτελείται ο ίδιος κώδικας και ως εκ τούτου χρειαζόμαστε μόνο ένα τερματικό - σε όλες τις άλλες περιπτώσεις απαιτείται η χρήση δύο τερματικών).



Παράδειγμα 6. Ανταλλαγή ενός μηνύματος μεταξύ σταθμών με αυτοδύναμα πακέτα. (αρχεία server2-udp.c και client2-udp.c).

Στο εν λόγω παράδειγμα, επιδεικνύεται η αποκατάσταση μιας επικοινωνίας χωρίς σύνδεση (ή ασυνδεσμικής επικοινωνίας) στην οποία χρησιμοποιούνται αυτοδύναμα πακέτα (datagrams). Με άλλα λόγια, δεν απαιτείται η προγενέστερη αποκατάσταση μιας σύνδεσης ανάμεσα στα δύο άκρα και για το λόγο αυτό δεν χρησιμοποιούνται οι συναρτήσεις listen και accept. Στις συνδέσεις αυτού του είδους, ο πελάτης ορίζει απλά τη διεύθυνση στην οποία θέλει να αποστείλει το πακέτο και το αποστέλλει χωρίς να υπάρχει καμία εγγύηση και χωρίς να ελέγχει καν εάν το πακέτο έφτασε με επιτυχία. Κατά την παραλαβή, ο πελάτης καθορίζει από που θέλει να παραλάβει το πακέτο και ενημερώνεται για την κατάσταση του αποστολέα. Σε αυτού του τύπου την επικοινωνία δεν υφίσταται αρχιτεκτονική client-server αλλά όλοι οι κόμβοι είναι ομότιμοι δηλαδή λειτουργούν με τον ίδιο τρόπο (ωστόσο, χρησιμοποιούμε αυτούς τους όρους στα ονόματα των αρχείων απλά και μόνο για να τα ξεχωρίσουμε μεταξύ τους – αλλά σε εισαγωγικά για να τονίσουμε την ομοτιμία των κόμβων).

Κώδικας «διακομιστή»

Αρχείο server2-udp.c

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

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT    8080
#define MAXLINE 1024

int main() {
   int sockfd, len, n;
   char buffer[MAXLINE];
   char *hello = "Hello from server";
   struct sockaddr_in servaddr, cliaddr;
```

Παρά τη διαφορετική φύση της επικοινωνίας, τα πάντα λειτουργούν όπως πριν. Οι δύο διεργασίες καλούν τη συνάρτηση socket για να δημιουργήσουν την κατάλληλη υποδομή για τη μεταξύ τους επικοινωνία, μόνο που αυτή τη φορά χρησιμοποιείται ως όρισμα στη συνάρτηση η σταθερά SOCK_DGRAM για να δείξουμε πως στην προκειμένη περίπτωση η επικοινωνία που αποκαθίσταται είναι επικοινωνία χωρίς σύνδεση.

```
   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket creation failed");
      exit(EXIT_FAILURE); }
```

Στο επόμενο βήμα αρχικοποιείται με τον τρόπο που είδαμε και στα προηγούμενα παραδείγματαη η δομή servaddr προκειμένου να διαβιβαστεί ως όρισμα στη συνάρτηση bind η οποία λειτουργεί και αυτή με τον τρόπο που έχουμε περιγράψει στα προηγούμενα παραδείγματα.

```
   memset(&servaddr, 0, sizeof(servaddr));
   memset(&cliaddr, 0, sizeof(cliaddr));
   servaddr.sin_family    = AF_INET; // IPv4
   servaddr.sin_addr.s_addr = INADDR_ANY;
   servaddr.sin_port = htons(PORT);

  if (bind (sockfd, (const struct sockaddr *)&servaddr,  sizeof(servaddr)) < 0) {
      perror("bind failed");
      exit(EXIT_FAILURE); }
```

Επειδή πρόκειται για επικοινωνία χωρίς σύνδεση δεν καλούνται (όπως αναφέραμε πιο πάνω) οι συναρτήσεις listen και accept οπότε οι δύο διεργασίες προχωρούν απευθείας στην αποστολή και λήψη δεδομένων, χρησιμοποιώντας τις συναρτήσεις sendto και recvfrom. Η διεργασία «διακομιστής» πρώτα διαβάζει το μήνυμα του «πελάτη» και στη συνέχεια αποστέλλει το δικό της μήνυμα σε αυτόν, ενώ η διεργασία «πελάτης» ο πηγαίος κώδικας της οποίας ακολουθεί στη συνέχεια, πραγματοποιεί πρώτα τη διαδικασία αποστολής του μηνύματος στο διακομιστή και στη συνέχεια παραλαμβάνει το μήνυμα που στάλθηκε από αυτόν.

```
   n = recvfrom (sockfd, (char *)buffer, MAXLINE, MSG_WAITALL,  (struct sockaddr *) &cliaddr, &len);
   buffer[n] = '\0';
   printf("Client : %s\n", buffer);

   sendto (sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM,  (const struct sockaddr *) &cliaddr, len);

   printf("Hello message sent.\n");
   return 0; }
```

Κώδικας «πελάτη»

Αρχείο client2-udp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

int main() {
   int sockfd, len, n;
   char buffer[MAXLINE];
   char *hello = "Hello from client";
   struct sockaddr_in     servaddr;
```

Αρχικά καλείται η socket με όρισμα SOCK_DGRAM για τη δημιουργία του socket επικοινωνίας.

```
   if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket creation failed");
      exit(EXIT_FAILURE); }
```

Αμέσως μετά αρχικοποιείται η δομή servaddr προκειμένου να διαβιβαστεί ως όρισμα στη συνάρτηση sendto.

```
   memset(&servaddr, 0, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(PORT);
   servaddr.sin_addr.s_addr = INADDR_ANY;
```

Ακολουθούν οι διαδικασίες αποστολής και παραλαβής δεδομένων από την ομότιμη διεργασία.

```
   sendto (sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM, (const struct sockaddr *) &servaddr, sizeof(servaddr));
   printf("Hello message sent.\n");

   n = recvfrom (sockfd, (char *)buffer, MAXLINE, MSG_WAITALL,
                  (struct sockaddr *) &servaddr, &len);
   buffer[n] = '\0';
   printf("Server : %s\n", buffer);
   close(sockfd);
   return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./client2-udp
Hello message sent.
Server : Hello from server
```

```
amarg@amarg-vbox:~$ ./server2-udp
Client : Hello from client
Hello message sent.
```

15) Σε αυτό το παράδειγμα αρχιτεκτονικής client – server με Unix sockets, η συνάρτηση run_server μέσα από ένα infinite loop καλεί συνεχώς την accept για να συλλάβει αιτήματα σύνδεσης από τέσσερις διεργασίες – πελάτες (που υλοποιούνται από τη συνάρτηση run_client), χρησιμοποιώντας τη select. Στη συνέχεια επικοινωνεί με τον καθέναν από τους πελάτες ανταλλάσσοντας με αυτόν ένα μήνυμα. (αρχείο one-serv-n-cls-unix.c).

```
#include <stdio.h>
#include <sys/socket.h> // AF_INET and SOCK_STREAM
#include <arpa/inet.h>  // storage size of serv_addr
#include <unistd.h>     // getpid, getppid, fork
#include <string.h>     // bzero
#include <stdlib.h>
#include <errno.h>
#include <sys/un.h>

#define SOCKETNAME "MySocket"
```

Σε αυτό το παράδειγμα επιδεικνύεται η χρήση της select η οποία αποτελεί τη βασική συνάρτηση επικοινωνίας με πολλές τερματικές μονάδες ταυτόχρονα. Στην προκειμένη περίπτωση η εφαρμογή συνίσταται στη χρήση μίας συνάρτηση διακομιστή με όνομα run_server η οποία δέχεται αιτήματα εξυπηρέτησης από πελάτες χρησιμοποιώντας την accept με την επιλογή του πελάτη που θα συνδεθεί να στηρίζεται στη χρήση της select. Στη συνέχεια ο διακομιστής επικοινωνεί με τον καθέναν από αυτούς τους πελάτες ανταλλάσσοντας με αυτόν ένα μήνυμα. Όσον αφορά στους πελάτες αυτοί αποτελούν θυγατρικές διεργασίες της διεργασίας του διακομιστή και ως εκ τούτου δημιουργούνται με την κλήση της συνάρτησης fork. Ο καθένας από αυτούς τους πελάτες εκτελεί τη συνάρτηση run_client. Η επικοινωνία ανάμεσα στις διεργασίες στηρίζεται στη χρήση Unix Sockets.

Κώδικας διακομιστή

// Ο κώδικας του server

```
static int run_server(struct sockaddr_un *sap) {

   int fd_skt, fd_client, fd_hwm = 0, fd;
   char buf[100];
   fd_set set, read_set;
   ssize_t nread;
```

Ο διακομιστής αρχικά εκτυπώνει στην οθόνη το PID του.

```
   printf ("Server pid is %d\n", getpid());
```

Ο διακομιστής καλεί ως συνήθως τη συνάρτηση socket για να δημιουργήσει το socket με ορίσματα UNIX_SOCK (αφού θα χρησιμοποιηθεί Unix Socket και SOCK_STREAM αφού η εφαρμογή χρησιμοποιεί επικοινωνία με σύνδεση).

```
   if ((fd_skt = socket (AF_UNIX, SOCK_STREAM, 0)) < 0) {
      printf("\n Socket creation error \n"); return -1; }
```

Παρατηρήστε πως στην προκειμένη περίπτωση η διεύθυνση sockaddr_un *sap που περνά ως όρισμα στη συνάρτηση bind για να συσχετιστεί με το socket που δημιουργήθηκε από τη συνάρτηση socket διαβιβάζεται ως όρισμα στη συνάρτηση run_server αφού πρώτα αρχικοποιηθεί μέσα στη συνάρτηση main.

```
   if (bind (fd_skt, (struct sockaddr *) sap, sizeof(*sap)) < 0) {
      perror("Bind"); exit(EXIT_FAILURE); }
```

Ο διακομιστής τίθεται σε καθεστώς αναμονής εισερχομένων αιτημάτων σύνδεσης καλώντας τη συνάρτηση listen. Η σταθερά SOMAXCONN ορίζεται στο αρχείο socket.h στην τιμή 128 αλλά γενικά τίθεται σε μικρότερη τιμή.

```
   if (listen (fd_skt, SOMAXCONN)<0) {
      perror("Listen"); exit(EXIT_FAILURE); }
```

Οι μακροεντολές FD_ZERO και FD_SET λειτουργούν με παρόμοιο τρόπο με τον οποίο λειτουργούν οι συναρτήσεις sigemptyset και sigaddset που είδαμε στην περίπτωση των σημάτων. Ειδικότερα, η μακροεντολή FD_ZERO αρχικοποιεί το

σύνολο των file descriptors εκχωρώντας σε όλα τα bits την τιμή μηδέν, ενώ η FD_SET ενεργοποιεί το bit του συνόλου fd_set που αντιστοιχεί στον περιγραφέα αρχείου που δέχεται ως όρισμα.

```
if (fd_skt > fd_hwm) fd_hwm = fd_skt;
FD_ZERO (&set);
FD_SET (fd_skt, &set);
```

Η διαδικασία που πραγματοποιείται στον κεντρικό βρόχο της συνάρτησης του διακομιστή, έχει ως εξής. Καταρχήν παρατηρήστε πως ο βρόχος είναι ατέρμων, δηλαδή εκτελείται επ' άπειρον και αυτό διότι ένας διακομιστής υποτίθεται πως είναι συνεχώς ενεργός, αναμένοντας αιτήματα σύνδεσης. Με άλλα λόγια, για να ολοκληρωθεί η διαδικασία θα πρέπει ο χρήστης να τερματίσει τη λειτουργία της συνάρτησης του διακομιστή πατώντας τον συνδυασμό πλήκτρων Ctrl-C.

```
while (1) {
   read_set = set;
```

Παρατηρήστε πως η select στην προκειμένη περίπτωση, δέχεται μόνο ένα όρισμα τύπου fd_set το οποίο αντιστοιχεί σε διαδικασία ανάγνωσης (διότι ενδιαφέρεται μόνο να διαβάσει δεδομένα από τους πελάτες και όχι να στείλει δεδομένα σε αυτούς, ή να αναφέρει μηνύματα σφάλματος). Το πρώτο όρισμα της select σύμφωνα με τη θεωρία είναι ίσο με fd_skt+1.

```
   if (select(fd_hwm+1, &read_set, NULL, NULL, NULL)==-1) {
      perror("Select"); exit(EXIT_FAILURE); }
```

Μέσα από έναν δεύτερο ένθετο βρόχο for, η συνάρτηση εξετάζει έναν προς έναν τους περιγραφείς αρχείων που έχουν τιμές από 0 έως fd_hwm.

```
   for (fd = 0; fd <= fd_hwm; fd++)
```

Εάν συναντήσει περιγραφέα fd που ανήκει στο σύνολο read_set (οπότε η FD_ISSET επιστρέφει true)

```
      if (FD_ISSET(fd, &read_set)) {
```

ΚΑΙ αυτός ο περιγραφέας είναι ο fd_skt που επέστρεψε η socket και χρησιμοποιείται στην bind και στη listen ΤΟΤΕ

```
         if (fd == fd_skt) {
```

καλεί τη συνάρτηση accept με πρώτο όρισμα αυτόν τον περιγραφέα η οποία επιστρέφει τον περιγραφέα fd_client για το active socket μέσω του οποίου θα πραγματοποιηθεί η επικοινωνία με τον πελάτη (εάν η accept δεν λειτουργήσει με επιτυχία η εφαρμογή τερματίζεται με μήνυμα σφάλματος)

```
            if ((fd_client = accept(fd_skt, NULL, 0))<0) {
               perror("Select"); exit(EXIT_FAILURE); }
```

Ενεργοποιεί το bit του συνόλου fd_set που αντιστοιχεί στον περιγραφέα fd_client

```
            FD_SET (fd_client, &set);

            if (fd_client > fd_hwm) fd_hwm = fd_client; }

         else {
```

Εάν η read από το αρχείο με περιγραφέα fd αποτύχει και ως εκ τούτου η συνάρτηση επιστρέψει αρνητικό αριθμό, η διαδικασία τερματίζεται με μήνυμα σφάλματος.

```
            if ((nread = read(fd, buf, sizeof(buf)))<0) {
                perror("Read"); exit(EXIT_FAILURE); }
```

Εάν δεν υπάρχουν δεδομένα για ανάγνωση τότε η παραπάνω διαδικασία αναιρείται δηλαδή το bit που αντιστοιχεί σε αυτόν τον περιγραφέα τίθεται από την FD_CLR στη μηδενική τιμή, δηλαδή απενεργοποιείται ενώ η τιμή του fd_hwm μειώνεται κατά μία μονάδα.

```
        if (nread == 0) {
            FD_CLR (fd, &set);
            if (fd == fd_hwm) fd_hwm--;
            close(fd); }
```

Διαφορετικά, ο περιγραφέας διαβάζει το μήνυμα του πελάτη, το εκτυπώνει στην οθόνη και του απσοτέλει το μήνυμα GoodBye !!. Εάν η διαδικασία αποστολής (δηλαδή εγγραφής στο fd) αποτύχει, η διαδικασία τερματίζεται με μήνυμα σφάλματος.

```
          else {
            printf("Server got \"%s\"\n", buf);
            if (write(fd, "Goodbye!", 9)==-1) {
               perror("Write"); exit(EXIT_FAILURE); }}}}}

  close(fd_skt);
  close(fd_client);
  printf ("Server terminates\n");
  return 1; }
```

Κώδικας πελάτη

// Ο κώδικας του client

Η συνάρτηση του πελάτη run_client καλεί τη fork για να δημιουργήσει μία διεργασία και περιορίζεται στον κώδικα της θυγατρικής διεργασίας η οποία αντιστοιχεί στη συνθήκη pid = 0.

```
static int run_client(struct sockaddr_un *sap) {

  if (fork() == 0) {
     int fd_skt;
     char buf[100];
```

Αρχικά καλείται η συνάρτηση socket με τα ίδια ορίσματα με αυτά που κλήθηκε στη συνάρτηση run_server έτσι ώστε οι δύο διεργασίες να μπορέσουν να επικοινωνήσουν μεταξύ τους

```
     if ((fd_skt = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n"); return -1; }
```

και στη συνέχεια καλείται επαναληπτικά η connect μέχρι τελικά η διεργασία πελάτης να καταφέρει να συνδεθεί στο διακομιστή (αυτό γίνεται διότι όπως σχολιάσαμε και σε άλλο παράδειγμα, υπάρχει περίπτωση όταν η θυγατρική διεργασία αιτηθεί σύνδεση, να μην έχει δημιουργηθεί ακόμη το άλλο άκρο της σύνδεσης στη διεργασία του διακομιστή και ως εκ τούτου η σύνδεση να μην είναι δυνατή).

```
     while (connect(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) == -1) {
       if (errno == ENOENT) {
          sleep(1);
          continue; }
       else {
        perror("Message from connect [client]");
        exit(-2); }}
```

Μετά την αποκατάσταση της σύνδεσης, η θυγατρική διεργασία δημιουργεί και διαμορφώνει κατάλληλα ένα μήνυμα (που περιέχει το PID της) το οποίο στη συνέχεια στέλνει με τη συνάρτηση write στο διακομιστή χρησιμοποιώντας το fd_skt που επέστρεψε η socket.

```
snprintf (buf, sizeof(buf), "Hello from %ld!", (long)getpid());
if (write (fd_skt, buf, strlen(buf)+1) < 0) {
   perror("Write"); exit(EXIT_FAILURE); }
```

Στη συνέχεια διαβάζει με τη συνάρτηση read το μήνυμα που της έστειλε η συνάρτηση του διακομιστή το οποίο εκτυπώνει στην οθόνη της και κλείνει το socket της επικοινωνίας.

```
if ((read(fd_skt, buf, sizeof(buf))) < 0) {
   perror("Read"); exit(EXIT_FAILURE); }
printf("Client %d got %s\n", getpid(), buf);
close(fd_skt);

printf ("Client %d terminates\n", getpid());
exit(EXIT_SUCCESS); }
return 1; }
```

Η συνάρτηση main αρχικοποιεί τη διεύθυνση sa τύπου sockaddr_un στις τιμές SOCKETNAME και AF_UNIX. Στη συνέχεια μέσα από ένα βρόχο τεσσάρων επαναλήψεων καλεί τη συνάρτηση run_client για να δημιουργήσει τις τέσσερις θυγατρικές διεργασίες – πελάτες και στη συνέχεια μία φορά τη συνάρτηση run_server για να δημιουργήσει τη διεργασία του διακομιστή. Αμφότερες οι συναρτήσεις παίρνουν ως όρισμα ένα δείκτη στη δομή sa.

```
int main(void) {
   struct sockaddr_un sa;
   int nclient;
   (void)unlink(SOCKETNAME);
   strcpy(sa.sun_path, SOCKETNAME);
   sa.sun_family = AF_UNIX;
   for (nclient = 1; nclient <= 4; nclient++)
      run_client(&sa);
   run_server(&sa);
   exit(EXIT_SUCCESS); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
Server pid is 3082
Server got "Hello from 3086!"
Server got "Hello from 3085!"
Server got "Hello from 3087!"
Client 3086 got Goodbye!
Client 3086 terminates
Client 3087 got Goodbye!
Client 3085 got Goodbye!
Server got "Hello from 3084!"
Client 3087 terminates
Client 3085 terminates
Client 3084 got Goodbye!
Client 3084 terminates
```

---

16) Επικοινωνία διεργασιών με κλείδωμα κοινόχρηστου αρχείου (αρχεία producer.c και consumer.c).

Αρχείο producer.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
```

Το κοινόχρηστο αρχείο που θα χρησιμοποιηθεί έχει όνομα data.dat ενώ το μήνυμα που θα αποθηκευτεί στο αρχείο ορίζεται ως η σταθερή συμβολοσειρά DataString.

```
#define FileName "data.dat"
#define DataString "Now is the winter of our discontent\n"
```

Σε αυτή την εφαρμογή, οι δύο διεργασίες εκ των οποίων η πρώτη αποτελεί μία διεργασία συγγραφέα ενώ η δεύτερη, μία διεργασία αναγνώστη, επικοινωνούν μεταξύ τους, δηλαδή ανταλλάσσουν ένα μήνυμα χρησιμοποιώντας ένα κοινόχρηστο αρχείο. Με άλλα λόγια, η διεργασία συγγραφέας ανοίγει το αρχείο, αποθηκεύει σε αυτό το μήνυμα και κλείνει το αρχείο, ενώ στη συνέχεια η διεργασία αναγνώστης, ανοίγει το αρχείο, διαβάζει το μήνυμα που της έστειλε η διεργασία συγγραφέας και κλείνει το αρχείο. Προκειμένου να αποτρέψουμε την ταυτόχρονη προσπέλαση του αρχείου από τις δύο διεργασίες χρησιμοποιείται ένας μηχανισμός κλειδώματος έτσι ώστε να διασφαλίσουμε ότι μόνο μία διεργασία μπορεί να ανοίξει το αρχείο σε κάθε χρονική στιγμή.

```
int main() {
  struct flock lock;
```

Το πιο σημαντικό κομμάτι του κώδικα σε αυτή την περίπτωση είναι η αρχικοποίηση της δομής lock. Ορίζοντας ως l_type την τιμή F_WRLCK κλειδώνουμε το αρχείο για αποκλειστική χρήση (exclusive lock) προκειμένου να πραγματοποιηθεί διαδικασία εγγραφής, ενώ τα υπόλοιπα ορίσματα κλειδώνουν ολόκληρο το αρχείο, από το πρώτο έως το τελευταίο byte (EOF). Το τελευταίο πεδίο είναι το PID της διεργασίας που κλειδώνει το αρχείο.

```
  lock.l_type = F_WRLCK;    /* read/write (exclusive versus shared) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */

  int fd; /* file descriptor to identify a file within a process */
```

Η διεργασία – συγγραφέας ανοίγει το αρχείο data.dat για ανάγνωση / εγγραφή (O_RDWR), το δημιουργεί εάν δεν υπάρχει (O_CREAT) και με default mode 0666 (rw-rw-rw-)

```
  if ((fd = open(FileName, O_RDWR | O_CREAT, 0666)) < 0)  {
      perror("open failed..."); exit(-1); }
```

Το αρχείο τίθεται σε κατάσταση exclusive lock περνώντας ως όρισμα στην fcntl τη δομή lock.

```
  if (fcntl(fd, F_SETLK, &lock) < 0) {
      perror("fcntl failed to get lock..."); exit(-2); }
  else {
```

Η διεργασία συγγραφέας αποθηκεύει στο αρχείο το περιεχόμενο της συμβολοσειράς DataString

```
    write(fd, DataString, strlen(DataString)); /* populate data file */
    fprintf(stderr, "Process %d has written to data file...\n", lock.l_pid); }
```

Στο τέλος της διαδικασίας το αρχείο ξεκλειδώνει έτσι ώστε να είναι προσπελάσιμο από άλλες διεργασίες και τελικά κλείνει αφού δεν χρειάζεται πλέον.

```
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0) {
      perror("explicit unlocking failed..."); exit(-1); }
```

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

Αρχείο consumer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define FileName "data.dat"
```

Όπως αναφέραμε προηγουμένως, η διεργασία αναγνώστης, ανοίγει το αρχείο, διαβάζει το μήνυμα που της έστειλε η διεργασία συγγραφέας και κλείνει το αρχείο.

```
int main() {
 struct flock lock;
```

Η αρχικοποίηση της δομής lock γίνεται ακριβώς όπως και πριν

```
 lock.l_type = F_WRLCK;    /* read/write (exclusive) lock */
 lock.l_whence = SEEK_SET; /* base for seek offsets */
 lock.l_start = 0;         /* 1st byte in file */
 lock.l_len = 0;           /* 0 here means 'until EOF' */
 lock.l_pid = getpid();    /* process id */

 int fd; /* file descriptor to identify a file within a process */
```

Εδώ το αρχείο ανοίγει μόνο για ανάγνωση.

```
 if ((fd = open(FileName, O_RDONLY)) < 0) {
    perror("open to read failed..."); exit(-1); }
```

Στην προκειμένη περίπτωση πριν επιχειρήσουμε να διαβάσουμε από το αρχείο θα πρέπει να ελέγξουμε εάν αυτό είναι κλειδωμένο καλώντας την συνάρτηση fcntl με όρισμα *F_GETLK*. *Εάν η επιστρεφόμενη τιμή της συνάρτησης δεν είναι η F_UNLCK αυτό σημαίνει πως κάποια άλλη διεργασία έχει κλειδώσει το αρχείο προκειμένου να το χρησιμοποιήσει με αποκλειστικό τρόπο. Επομένως δεν είναι δυνατή η προσπέλαση του από τη διεργασία αναγνώστη, η οποία για το λόγο αυτό εκτυπώνει ένα μήνυμα λάθους και τερματίζει τη λειτουργία της.*

```
 fcntl(fd, F_GETLK, &lock); /* sets lock.l_type to F_UNLCK if no write lock */
 if (lock.l_type != F_UNLCK) {
    perror("file is still write locked..."); exit(-1); }
```

Στη συνέχεια, η διεργασία αναγνώστης κλειδώνει με τη σειρά της το αρχείο προκειμένου κατά τη διάρκεια της διαδικασίας ανάγνωσης του περιεχομένου του να μην μπορεί να γράψει σε αυτό καμία άλλη διεργασία.

```
 lock.l_type = F_RDLCK; /* prevents any writing during the reading */
 if (fcntl(fd, F_SETLK, &lock) < 0) {
    perror("can't get a read-only lock..."); exit(-1); }
```

Στη συνέχεια διαβάζει το μήνυμα που είναι αποθηκευμένο στο αρχείο ένα χαρακτήρα κάθε φορά και το εκτυπώνει στην προεπιλεγμένη της έξοδο που είναι η οθόνη.

```
 int c; /* buffer for read bytes */
 while (read(fd, &c, 1) > 0)    /* 0 signals EOF */
        write(STDOUT_FILENO, &c, 1); /* write one byte to the standard output */
```

Στο τέλος της διαδικασίας το αρχείο ξεκλειδώνει έτσι ώστε να είναι προσπελάσιμο από άλλες διεργασίες και τελικά κλείνει αφού δεν χρειάζεται πλέον.

```
lock.l_type = F_UNLCK;
if (fcntl(fd, F_SETLK, &lock) < 0) {
    perror("explicit unlocking failed..."); exit(-1); }

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

Παράδειγμα εξόδου της διεργασίας, ακολουθεί στη συνέχεια. Η κάθε διεργασία εκτελείται στο δικό της τερματικό, αν και στην προκειμένη περίπτωση επειδή οι διεργασίες δεν επικοινωνούν μεταξύ τους μέσω της ανταλλαγής μηνυμάτων, μπορούν να εκτελεστούν η μία μετά την άλλη στο ίδιο τερματικό. Στην περίπτωση αυτή, η πρώτη διεργασία θα ανοίξει το αρχείο, θα γράψει το μήνυμα της και θα το κλείσει ολοκληρώνοντας τη λειτουργία της, ενώ στη συνέχεια εκτελείται η δεύτερη διεργασία, η οποία ανοίγει το αρχείο, διαβάζει το μήνυμα, το εκτυπώνει στην οθόνη της και τέλος κλείνει το αρχείο. Με τον τρόπο αυτό οι δύο διεργασίες επικοινωνούν μεταξύ τους.

```
amarg@amarg-vbox:~$ ./producer
Process 3249 has written to data file...
amarg@amarg-vbox:~$ ./consumer
Now is the winter of our discontent
```

17) Επικοινωνία διεργασιών με χρήση κοινόχρηστης μνήμης (αρχεία memwriter.c και memreader.c).

Η εφαρμογή memwriter γράφει ένα ενδεικτικό μήνυμα (*This is the way the world ends*) στην κοινόχρηστη μνήμη χρησιμοποιώντας βοηθητικό αρχείο και αναμένει για 12 δευτερόλεπτα έτσι ώστε να δώσει τη δυνατότητα στην εφαρμογή memreader να διαβάσει τα περιεχόμενα από την κοινόχρηστη μνήμη.

Αρχείο memwriter.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include "shmem.h"

#define ByteSize 512
#define MemContents "This is the way the world ends...\n"

int main() {
```

Σε αυτή την εφαρμογή, η επικοινωνία ανάμεσα στις δύο διεργασίες στηρίζεται στη χρήση ενός βοηθητικού αρχείου το οποίο δημιουργείται από την εφαρμογή memwriter. Η εντολή shm_open δέχεται παρόμοια ορίσματα με την open και στην προκειμένη περίπτωση ανοίγει το αρχείο *shMemEx* το οποίο δημιουργεί *(O_CREAT)* για διαδικασίες ανάγνωσης / εγγραφής *(O_RDWR)* με δικαιώματα *0644 (rw-r—r--)*. Είναι σημαντικό να γίνει κατανοητό πως το εν λόγω αρχείο υφίσταται μόνο για το χρονικό διάστημα εκτέλεσης της εν λόγω διεργασίας, αφού με τον τερματισμό της, το αρχείο διαγράφεται από το δίσκο. Αυτό το αρχείο μπορεί να βρεθεί στον κατάλογο /dev/shm του συστήματος αρχείων.

```
int fd = shm_open ("/shMemEx", O_RDWR | O_CREAT, 0644);
```

Εάν για κάποιο λόγο το αρχείο δεν μπορεί να δημιουργηθεί η εφαρμογή δεν γίνεται να εκτελεστεί και ως εκ τούτου τερματίζει εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

```
if (fd < 0) { perror ("Can't open shared mem segment..."); exit (-1); }
```

Εάν το αρχείο δημιουργηθεί με επιτυχία καλείται η συνάρτηση ftruncate για να ορίσει το μέγεθος του αρχείου το οποίο τίθεται στην τιμή ByteSize (το μέγιστο μήκος του μηνύματος που μπορεί να ανταλλαγεί ανάμεσα στις διεργασίες).

ftruncate(fd, ByteSize);

Σε αυτό το σημείο καλείται η συνάρτηση mmap για να δημιουργήσει μία νέα απεικόνιση στον εικονικό χώρο διευθύνσεων της διεργασίας. Αν και μπορούμε να ορίσουμε ως πρώτο όρισμα σε αυτή τη συνάρτηση κάποια διεύθυνση, ωστόσο, αφήνουμε αυτό το όρισμα στην τιμή NULL προκειμένου να αποφασίσει ο πυρήνας σχετικά με την περιοχή μνήμης στην οποία θα τοποθετηθεί η απεικόνιση. Ως μήκος για αυτή την απεικόνιση ορίζουμε την τιμή ByteSize που είναι το μέγιστο μήκος του μηνύματος που θα ανταλλαγεί ανάμεσα στις δύο διεργασίες. Στη συνέχεια χρησιμοποιούμε τα flags PROT_READ και PROT_WRITE ορίζοντας για αυτή την περιοχή δικαιώματα ανάγνωσης και εγγραφής καθώς και το flag MAP_SHARED που καθιστά κοινόχρηστη αυτή την απεικόνιση (κάτι που είναι αναγκαίο προκειμένου στη διαδικασία να χρησιμοποιηθεί το βοηθητικό αρχείο μέσω του οποίου οι δύο διεργασίες θα χρησιμοποιήσουν από κοινού αυτή την περιοχή μνήμης). Το επόμενο όρισμα είναι ο περιγραφέας αρχείου fd για αυτό το βοηθητικό αρχείο, ο οποίος έχει επιστραφεί από τη συνάρτηση shm_open ενώ το τελευταίο όρισμα που εκφράζει την απόσταση μέσα στο αρχείο στο οποίο θα τοποθετηθούν τα κοινόχρηστα δεδομένα τίθεται στη μηδενική τιμή, έτσι ώστε αυτά να τοποθετηθούν στην αρχή του αρχείου. Εάν η απεικόνιση δεν μπορεί να δημιουργηθεί για κάποιο λόγο, η εκτέλεση της εφαρμογής δεν είνα δυνατή, οπότε τερματίζει εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

```
caddr_t memptr = mmap(NULL, ByteSize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if ((caddr_t)-1==memptr) { perror ("Can't get segment..."); exit (-2); }
```

Στο επόμενο βήμα ορίζουμε έναν σημαφόρο χρησιμοποιώντας τη συνάρτηση sem_open. Επειδή ο σημαφόρος δημιουργείται από τη συνάρτηση αφού χρησιμοποιούμε το flag O_CREAT θα πρέπει να ορίσουμε τα permissions (που τα θέτουμε στην τιμή 0644) και την αρχική τιμή του σημαφόρου που είναι η τιμή 0. Εάν ο σημαφόρος δεν δημιουργηθεί, η εκτέλεση της εφαρμογής δεν είνα δυνατή, οπότε και πάλι τερματίζει εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

```
sem_t* semptr = sem_open("sem", O_CREAT, 0644, 0);
if (semptr==(void*) -1) { perror ("sem_open"); exit (-2); }
```

Εάν όλες οι παραπάνω διαδικασίες πραγματοποιηθούν χωρίς πρόβλημα οι δύο διαδικασίες μπορούν πλέον να επικοινωνήσουν μεταξύ τους. Σε αυτό λοιπόν το σημείο καλούμε τη συνάρτηση strcpy προκειμένου να αντιγράψουμε το περιεχόμενο της συμβολοσειράς που περιέχει το μήνυμα στην κοινόχρηστη περιοχή μνήμης.

```
strcpy(memptr, MemContents);
```

Στη συνέχεια αυξάνουμε την τιμή του σημαιοφόρου κατά μία μονάδα (unlock) προκειμένου να δώσουμε τη δυνατότητα στην άλλη διεργασία να διαβάσει το μήνυμα (εάν αυτή η συνάρτηση δεν εκτελεστεί σωστά, ξανά η διεργασία τερματίζεται εκτυπώνοντας μήνυμα σφάλματος) και αμέσως μετά η διεργασία αναστέλλεται για 12 δευτερόλεπτα (ο αριθμός είναι ενδεικτικός : μπορείτε να βάλετε οποίο νούμερο θέλετε) έτσι ώστε η άλλη διεργασία να προλάβει να εκτελεστεί και να διαβάσει το μήνυμα από το κοινόχρηστο αρχείο.

```
if (sem_post (semptr) < 0) { perror ("sem_post"); exit (-3); }
sleep(12); /* give reader a chance */
```

Μετά την παρέλευση των 12 δευτερολέπτων κατά τη διάρκεια των οποίων (ευελπιστούμε πως) η άλλη διεργασία προσπέλασε το αρχείο και διάβασε το κοινόχρηστο μήνυμα, η διαδικασία ολοκληρώνεται καταργώντας την απεικόνιση μνήμης και κλείνοντας τα εμπλεκόμενα αρχεία και τις δομές που χρησιμοποιήθηκαν (σημαφόροι).

```
munmap(memptr, ByteSize); /* unmap the storage */
close(fd);
sem_close(semptr);
shm_unlink("/shMemEx");

return 0; }
```

Η εφαρμογή memreader εντός 12 δευτερολέπτων από την έναρξη της memwriter, διαβάζει από την κοινόχρηστη μνήμη το περιεχόμενό της και το εκτυπώνει στην οθόνη.

Αρχείο memreader.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include "shmem.h"

#define ByteSize 512
#define MemContents "This is the way the world ends...\n"
```

Η διεργασία memreader σχεδόν σε όλα λειτουργεί με τον ίδιο ακριβώς τρόπο και ως εκ τούτου ισχύουν όλα όσα αναφέραμε στην διεργασία memwriter. Η μόνη διαφορά είναι ότι στη συνάρτηση shm_open δεν χρησιμοποιούμε το flag O_CREAT αλλά μόνο το O_RDWR αφού το αρχείο έχει ήδη δημιουργηθεί από τη διεργασία memwriter. Τα υπόλοιπα ισχύουν ως έχουν.

```
int main() {

  int fd = shm_open("/shMemEx", O_RDWR, 0644);

  if (fd < 0) { perror ("Can't get file descriptor..."); exit (-1); }

  caddr_t memptr = mmap(NULL, ByteSize, PROT_READ | PROT_WRITE,   MAP_SHARED, fd, 0);
  if ((caddr_t)-1==memptr) { perror ("Can't access segment..."); exit (-1);  }
  sem_t* semptr = sem_open("sem", O_CREAT, 0644, 0);
  if (semptr == (void*)-1) { perror ("sem open"); exit (-1); }
```

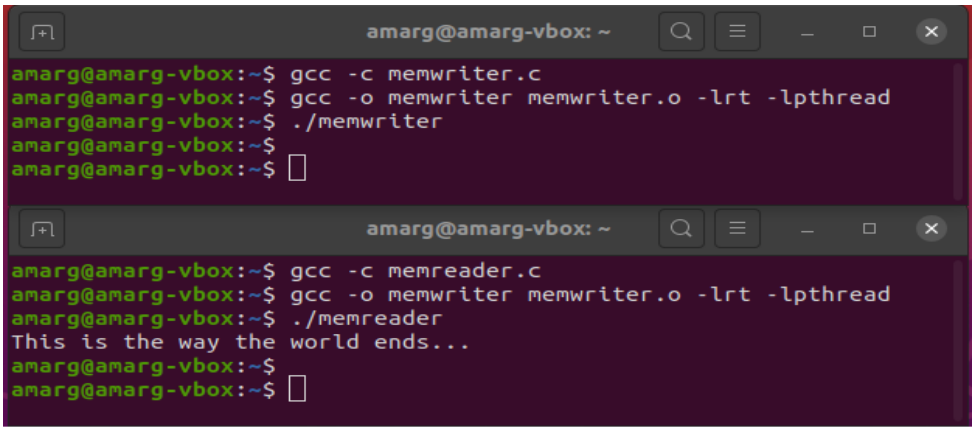
Ωστόσο, υπάρχει διαφοροποίηση στον τρόπο χρήσης του σημαφόρου. Θυμηθείτε πως για να είναι δυνατή η ανάγνωση του αρχείου από τη διεργασία memreader, η διεργασία memwriter θα πρέπει να αυξήσει την τιμή του σημαφόρου κατά μία μονάδα καλώντας την συνάρτηση sem_post έτσι ώστε να ξεκλειδώσει το αρχείο για να χρησιμοποιηθεί από τη διεργασία memreader. Εάν η τιμή του σημαφόρου δεν είναι διαφορετική του μηδενός (και ειδικότερα, μεγαλύτερη του μηδενός) η διεργασία memreader δεν μπορεί να προσπελάσει το αρχείο. Εκείνο που κάνει είναι να καλεί τη συνάρτηση sem_wait η οποία αναμένει να λάβει ο σημαφόρος θετική τιμή και επιστρέφει μηδενική τιμή όταν αυτό συμβεί. Όταν λοιπόν ο σημαφόρος αυξηθεί κατά μία μονάδα και το αρχείο έχει ξεκλειδώσει, τότε η διεργασία memreader διαβάζει ένα προς ένα τα bytes από την περιοχή μνήμης που υποδηλώνεται από το δείκτη memprt που επέστρεψε η συνάρτηση mmap και μέσω μίας επαναληπτικής διαδικασίας που εκτελείται τόσες φορές όση είναι και η τιμή του ByteSize εκτυπώνει τους χαρακτήρες στην προεπιλεγμένη έξοδο που είναι η οθόνη.

```
  if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
     int i;
     for (i = 0; i < strlen (MemContents); i++)
          write(STDOUT_FILENO, memptr + i, 1);
     sem_post(semptr); }
```

Η διαδικασία ολοκληρώνεται καταργώντας την απεικόνιση μνήμης και κλείνοντας τα εμπλεκόμενα αρχεία και τις δομές που χρησιμοποιήθηκαν (σημαφόροι).

```
  munmap(memptr, ByteSize);
  close(fd);
  sem_close(semptr);
  unlink(BackingFile);
  return 0; }
```

Η εκτέλεση των εφαρμογών η καθεμία από το δικό της ξεχωριστό τερματικό, ακολουθεί στη συνέχεια.



Εάν κατά τη διάρκεια της εκτέλεσης του memwriter ελέγξουμε τα περιεχόμενα του καταλόγου /dev/shm θα διαπιστώσουμε πως σε αυτόν τον κατάλογο υπάρχει το βοηθητικό αρχείο που περιέχει το μήνυμα που ανταλλάσσεται ανάμεσα στις δύο διεργασίες και το οποίο διαγράφεται όταν ολοκληρωθεί η λειτουργία της καθώς και το αρχείο του σεμαφόρου sem.sem.