ΕΡΓΑΣΤΗΡΙΟ 1

Εξοικείωση με το Linux

1. Συνδεθείτε στο σύστημα.

   Ανάλογα με το σύστημα είτε η σύνδεση γίνεται αυτόματα (autologin) είτε με το χέρι καταχωρώντας το login και το password. Στο Ubuntu linux η σύνδεση γίνεται αυτόματα και ο χρήστης μεταφέρεται απευθείας στο γραφικό περιβάλλον.

2. Εκτελέστε την εντολή whoami για να εκτυπώσετε το login name.

```
amarg@amarg-vbox:~$ whoami
amarg
```

3. Εκτελέστε την εντολή pwd για να εκτυπώσετε τον τρέχοντα κατάλογο.

```
amarg@amarg-vbox:~$ pwd
/home/amarg
```

4. Εκτελέστε τις εντολές ls, ls –l και ls –la. Τι παρατηρείτε?

   Η εντολή ls εκτυπώνει μόνο τα ονόματα των αρχείων και των υποκαταλόγων του καταλόγου που δέχεται ως όρισμα.

```
amarg@amarg-vbox:~$ ls
acc1            flength1            myCopyc.c   prog.c          sigExample5.c
acc1.o          flength1.c          myCopyc.o   prog.o          sigExample5.o
a.out           flength1.o          myCopy.o    Public          sigExample6
awkscript       flength.c           mydir       results         sigExample6a
check           flength.o           mydir2      rfile           sigExample6a.o
```

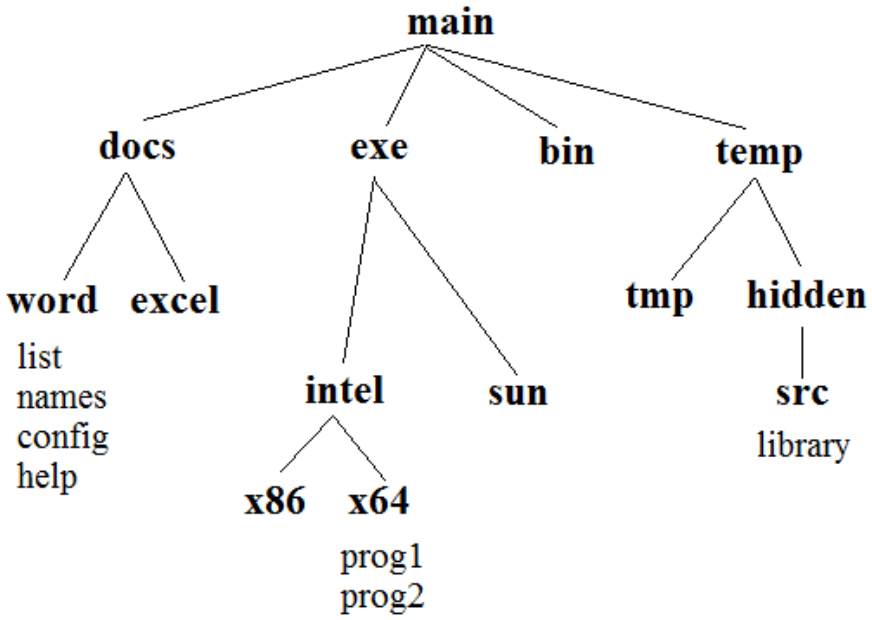
   Η εντολή ls –l εκτυπώνει μία γραμμή για κάθε αρχείο ή υποκατάλογο του τρέχοντος καταλόγου η οποία περιλαμβάνει: τα δικαιώματα πρόσβασης, το πλήθος των συνδέσμων προς το αρχείο ή τον κατάλογο, τον κάτοχο και την ομάδα του κατόχου του αρχείου ή του υποκαταλόγου, το μέγεθός του, την ημερομηνία και την ώρα της τελευταίας προσπέλασης και το όνομά του. Όσες γραμμές ξεκινούν με το γράμμα d αναφέρονται σε καταλόγους, ενώ όσες γραμμές ξεκινούν με μία παύλα (-) αναφέρονται σε κανονικά αρχεία.

```
amarg@amarg-vbox:~$ ls -l
total 49544
-rwxrwxr-x  1 amarg amarg    16872 Σεπ  29 12:13 acc1
-rw-rw-r--  1 amarg amarg     2264 Σεπ  29 12:13 acc1.o
-rwxrwxr-x  1 amarg amarg    16952 Σεπ  29 20:44 a.out
-rw-rw-r--  1 amarg amarg      167 Σεπ  22 10:27 awkscript
-rwxrwxr-x  1 amarg amarg      214 Σεπ  22 09:11 check
-rw-rw-r--  1 amarg amarg      189 Σεπ  29 11:54 chownEx.c
-rw-rw-r--  1 amarg amarg     1680 Σεπ  29 11:55 chownEx.o
```

   Η εντολή ls –l –a εκτυπώνει εκτός από τα συνήθη αρχεία που εκτυπώνει η ls –l και τα κρυφά αρχεία και καταλόγους, το όνομα των οποίων ξεκινά με τελεία (.).

```
amarg@amarg-vbox:~$ ls -l -a
total 49608
drwxr-xr-x 28 amarg amarg     4096 Οκτ  11 18:06 .
drwxr-xr-x  3 root  root      4096 Σεπ  19 19:57 ..
-rwxrwxr-x  1 amarg amarg    16872 Σεπ  29 12:13 acc1
-rw-rw-r--  1 amarg amarg     2264 Σεπ  29 12:13 acc1.o
-rwxrwxr-x  1 amarg amarg    16952 Σεπ  29 20:44 a.out
-rw-rw-r--  1 amarg amarg      167 Σεπ  22 10:27 awkscript
-rw-------  1 amarg amarg     7998 Οκτ   9 12:15 .bash_history
-rw-r--r--  1 amarg amarg      220 Σεπ  19 19:57 .bash_logout
-rw-r--r--  1 amarg amarg     3771 Σεπ  19 19:57 .bashrc
drwx------ 11 amarg amarg     4096 Σεπ  19 20:18 .cache
-rwxrwxr-x  1 amarg amarg      214 Σεπ  22 09:11 check
```

5. Να κατασκευάσετε το επόμενο δέντρο καταλόγων (Υποδείξεις: τα ονόματα με έντονη γραφή [**main, docs, exe, bin, temp, word, excel, intel, x86, x64, sun, tmp, hidden, src**] είναι ονόματα καταλόγων, ενώ τα υπόλοιπα [list, names, config, help, prog1, prog2, library] είναι ονόματα κενών αρχείων κειμένου. Το υποδέντρο με ρίζα το temp θα κατασκευαστεί από τον κατάλογο excel ενώ όλα τα υπόλοιπα τμήματα του δέντρου από τον κατάλογο main.



Μεταφερόμαστε στον προσωπικό μας κατάλογο εκτελώντας την εντολή ***cd*** *και μετά εκτελούμε την εντολή* ***rm –r main*** *για να διαγράψουμε το δέντρο με ρίζα το main εάν υπάρχει γιατί στην περίπτωση αυτή είναι αδύνατο να δημιουργήσουμε κατάλογο με το ίδιο όνομα main στην ίδια θέση. Στη συνέχεια δημιουργούμε τον κατάλογο main εκτελώντας την εντολή* ***mkdir main*** *και μεταβαίνουμε σε αυτόν τον κατάλογο με την εντολή* ***cd main*** *αφού σύμφωνα με την εκφώνηση το μεγαλύτερο τμήμα του δέντρου θα πρέπει να κατασκευαστεί ευρισκόμενοι στον κατάλογο main.*

```
amarg@amarg-vbox:~$ cd
amarg@amarg-vbox:~$ rm -r main
amarg@amarg-vbox:~$ mkdir main
amarg@amarg-vbox:~$ cd main
amarg@amarg-vbox:~/main$
```

Το υπόλοιπο τμήμα του δέντρου μπορεί να κατασκευαστεί με πολλούς τρόπους. Για παράδειγμα, μπορούμε να εργασθούμε οριζόντια, κατασκευάζοντας αρχικά τους καταλόγους docs, exe, bin και temp, στη συνέχεια να μπούμε διαδοχικά (α) στον κατάλογο docs και να κατασκευάσουμε τους καταλόγους word και excel, (β) στον κατάλογο exe και να κατασκευάσουμε τους καταλόγους intel και sun, κ.ο.κ. Η άλλη λύση είναι να κινηθούμε κατακόρυφα δηλαδή να κατασκευάσουμε τον κατάλογο docs και οτιδήποτε υπάρχει κάτω από αυτόν, ύστερα τον κατάλογο exe και οτιδήποτε υπάρχει κάτω από αυτόν, κ.ο.κ. Επειδή η δεύτερη προσέγγιση μας επιτρέπει να εξοικειωθούμε με την έννοια του υποδέντρου, είναι και αυτή που τελικά θα επιλέξουμε. Οι κατάλογοι του δέντρου θα κατασκευαστούν με την mkdir, ενώ τα αρχεία που υπάρχουν σε αυτούς θα κατασκευαστούν με την touch. Το τελευταίο υποδέντρο με ρίζα το temp, θα κατασκευαστεί σε δεύτερο χρόνο, αφού σύμφωνα με την εκφώνηση αυτό θα πρέπει να κατασκευαστεί από τον κατάλογο excel. Σε πρώτη φάση λοιπόν έχουμε:

```
amarg@amarg-vbox:~/main$ mkdir docs
amarg@amarg-vbox:~/main$ mkdir docs/word
amarg@amarg-vbox:~/main$ touch docs/word/list
amarg@amarg-vbox:~/main$ touch docs/word/names
amarg@amarg-vbox:~/main$ touch docs/word/config
amarg@amarg-vbox:~/main$ touch docs/word/help
amarg@amarg-vbox:~/main$ mkdir docs/excel
amarg@amarg-vbox:~/main$ mkdir exe
amarg@amarg-vbox:~/main$ mkdir exe/intel
amarg@amarg-vbox:~/main$ mkdir exe/intel/x86
amarg@amarg-vbox:~/main$ mkdir exe/intel/x64
amarg@amarg-vbox:~/main$ touch exe/intel/x64/prog1
amarg@amarg-vbox:~/main$ touch exe/intel/x64/prog2
amarg@amarg-vbox:~/main$ mkdir bin
amarg@amarg-vbox:~/main$
```

Ο τελευταίος κλάδος του δέντρου που περιλαμβάνει το temp και οτιδήποτε βρίσκεται από κάτω, θα κατασκευαστεί από τον κατάλογο excel. Αρχικά πρέπει να πάμε σε αυτόν τον κατάλογο και επειδή βρισκόμαστε στη main αυτό γίνεται εκτελώντας την εντολή ***cd docs/excel****. Παρατηρούμε τώρα πως ο κατάλογος* ***temp*** *θα πρέπει να κατασκευαστεί κάτω από τον κατάλογο* ***main*** *ο οποίος βρίσκεται κάτω από τον προσωπικό μας κατάλογο* ***~****. Κατά συνέπεια, αυτό το τμήμα του δέντρου θα κατασκευαστεί με την επόμενη αλληλουχία εντολών:*

```
amarg@amarg-vbox:~/main/docs/excel$ mkdir ~/main/temp
amarg@amarg-vbox:~/main/docs/excel$ mkdir ~/main/temp/tmp
amarg@amarg-vbox:~/main/docs/excel$ mkdir ~/main/temp/tmp/hidden
amarg@amarg-vbox:~/main/docs/excel$ mkdir ~/main/temp/tmp/hidden/src
amarg@amarg-vbox:~/main/docs/excel$ touch ~/main/temp/tmp/hidden/src/library
amarg@amarg-vbox:~/main/docs/excel$
```

ΣΧΟΛΙΑ*. Η παραπάνω λύση είναι ενδεικτική αλλά αποτελεί έναν από τους ενδεδειγμένους τρόπους κατασκευής του δέντρου. Ο άλλος τρόπος είναι να μην χρησιμοποιούμε ονόματα διαδρομών αλλά μόνο ονόματα καταλόγων – ωστόσο, επειδή εάν χρησιμοποιήσουμε στην mkdir απλά ονόματα καταλόγων, τότε, επειδή ο νέος κατάλογος δημιουργείται στον κατάλογο στον οποίο βρισκόμαστε, θα πρέπει συνέχεια να χρησιμοποιούμε την εντολή cd για να μεταφερόμαστε στον κατάλογο μέσα στον οποίο θέλουμε να κατασκευάσουμε το νέο κατάλογο, κάτι που είναι αρκετά χρονοβόρο. Για παράδειγμα, εάν είμαστε στον κατάλογο main και θέλουμε να κατασκευάσουμε τον κατάλογο docs και μέσα σε αυτόν τον κατάλογο word, αντί να γράψουμε mkdir docs και μετά mkdir docs/word μπορούμε αρχικά να κατασκευάσουμε τον κατάλογο docs γράφοντας mkdir docs, μετά να μεταβούμε σε αυτόν τον κατάλογο γράφοντας cd docs και αφού μπούμε στον κατάλογο docs να δημιουργήσουμε τον κατάλογο word με την εντολή mkdir word. Μετά θα πρέπει να επιστρέψουμε στον κατάλογο docs γράφοντας cd .. (οι δύο τελείες .. πάντα συμβολίζουν τον γονικό κατάλογο του καταλόγου στον οποίο βρισκόμαστε) και να δημιουργήσουμε τον κατάλογο excel ως mkdir excel. Από τον κατάλογο docs ανεβαίνουμε στον main γράφοντας cd .. και κατασκευάζουμε τον κατάλογο exe ως mkdir exe, κ.ο.κ. Πρόκειται ασφαλώς για μία χρονοβόρα διαδικασία.*

Τα παραπάνω είναι ουσιαστικά και θα πρέπει να κατανοηθούν πλήρως προκειμένου να επιλυθούν οι υπόλοιπες ασκήσεις. Από οποιοδήποτε σημείο του δέντρου Α και αν βρισκόμαστε, η διαδρομή προς ένα οποιοδήποτε άλλο σημείο Β είναι ***μία και μοναδική****. Πώς θα πάμε για παράδειγμα, από τον κατάλογο x64 στον κατάλογο word? Θα πρέπει να ανεβούμε στον κατάλογο intel (γονικός κατάλογος του x64, άρα ..), από τον intel να ανέβουμε πιο πάνω στον exe (γονικός κατάλογος του intel, άρα ..) από τον exe να ανέβουμε ακόμη πιο ψηλά, στον main (γονικός κατάλογος του exe, άρα ..) και μετά να κατεβούμε αρχικά στον docs και μετά στον word. Για να πάμε λοιπόν από τον x64 στον word θα πρέπει να εκτελέσουμε την ακόλουθη αλληλουχία εντολών:*

```
amarg@amarg-vbox:~/main/exe/intel/x64$ pwd
/home/amarg/main/exe/intel/x64 ──────────── εκκίνηση
amarg@amarg-vbox:~/main/exe/intel/x64$ cd ..
amarg@amarg-vbox:~/main/exe/intel$ cd ..
amarg@amarg-vbox:~/main/exe$ cd ..
amarg@amarg-vbox:~/main$ cd docs
amarg@amarg-vbox:~/main/docs$ cd word
amarg@amarg-vbox:~/main/docs/word$ pwd
/home/amarg/main/docs/word ──────────── τερματισμός
amarg@amarg-vbox:~/main/docs/word$
```

Αντί να χρησιμοποιήσουμε πέντε φορές την εντολή cd με ορίσματα (1) ***..*** *(2)* ***..*** *(3)* ***..*** *(4)* ***docs*** *και (5)* ***word*** *μπορούμε να κατασκευάσουμε τη διαδρομή ανάμεσα στους καταλόγους x64 και docs συνδυάζοντας τα παραπάνω ορίσματα με τον χαρακτήρα /. Αυτή η διαδρομή είναι η*

../../../docs/word

Έχοντας κατασκευάσει αυτή τη διαδρομή, μπορούμε να μεταβούμε από τον κατάλογο x64 στον κατάλογο word με μία απλή κλήση της εντολής cd που θα δέχεται ως όρισμα την παραπάνω διαδρομή με το αποτέλεσμα να είναι το ίδιο:

```
amarg@amarg-vbox:~/main/exe/intel/x64$ pwd
/home/amarg/main/exe/intel/x64
amarg@amarg-vbox:~/main/exe/intel/x64$ cd ../../../docs/word
amarg@amarg-vbox:~/main/docs/word$ pwd
/home/amarg/main/docs/word
amarg@amarg-vbox:~/main/docs/word$
```

Με άλλα λόγια, η απλή κλήση της cd με όρισμα μία διαδρομή που αποτελείται από πέντε τμήματα, είναι ισοδύναμη με πέντε κλήσεις της cd με την κάθε cd να παίρνει διαδοχικά τα ορίσματα (1), (2), (3), (4) και (5).

6. Από τον κατάλογο x64 να εμφανίσετε τα περιεχόμενα του καταλόγου word

Η κατάσταση έχει περιγραφεί στο προηγούμενο σχόλιο. Βέβαια, εδώ δεν χρειάζεται να μεταβούμε στον κατάλογο word και κατά συνέπεια, δεν χρειάζεται να καλέσουμε την cd. Θα καλέσουμε απλά από τον κατάλογο x64 την εντολή ls – l περνώντας ως όρισμα στην εντολή τη διαδρομή προς τον κατάλογο word με σημείο εκκίνησης τον κατάλογο x64, έτσι ώστε το λειτουργικό σύστημα να μπορεί να εντοπίσει τον κατάλογο word στο δέντρο καταλόγων:

```
amarg@amarg-vbox:~/main/exe/intel/x64$ ls -l ../../../docs/word
total 0
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 config
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 help
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 list
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 names
amarg@amarg-vbox:~/main/exe/intel/x64$ ──── ΔΕΝ έχουμε φύγει από τον κατάλογο x64 !!!
```

7. Ποια είναι η απόλυτη διαδρομή του αρχείου prog2? Ποιες είναι οι σχετικές διαδρομές του αρχείου prog1 όταν ο τρέχων κατάλογος είναι ο (α) x86, (β) excel και (γ) hidden? Πότε η σχετική διαδρομή προς ένα κατάλογο ταυτίζεται με την απόλυτη διαδρομή?

Η απόλυτη διαδρομή προς ένα αρχείο ή κατάλογο είναι η διαδρομή προς το αρχείο ή τον κατάλογο ξεκινώντας από τη ρίζα του δέντρου ενώ η σχετική διαδρομή είναι διαφορετική ανάλογα με το σημείο εκκίνησης το οποίο μπορεί να είναι οποιοδήποτε. Η σχετική με την απόλυτη διαδρομή ταυτίζονται όταν το σημείο εκκίνησης είναι η ρίζα. Επομένως:

- ***Η απόλυτη διαδρομή προς το αρχείο prog2 είναι η /home/amarg/main/exe/intel/x64/prog2.***
- ***Η σχετική διαδρομή προς το αρχείο prog1 όταν βρισκόμαστε στον κατάλογο x86 είναι η ../x64/prog1.***
- ***Η σχετική διαδρομή προς το αρχείο prog1 όταν βρισκόμαστε στον κατάλογο excel είναι η ../../exe/intel/x64/prog1.***

- ***Η σχετική διαδρομή προς το αρχείο prog1 όταν βρισκόμαστε στον κατάλογο hidden είναι η ../../exe/intel/x64/prog1.***

8. Από τον κατάλογο x86 να διαγράψετε με την εντολή rmdir τον κατάλογο exe. Τι παρατηρείτε?

```
amarg@amarg-vbox:~/main/exe/intel/x86$ rmdir ../..
rmdir: failed to remove '../..': Directory not empty
amarg@amarg-vbox:~/main/exe/intel/x86$
```

Ο κατάλογος δεν διαγράφεται γιατί δεν είναι άδειος και κατά συνέπεια θα πρέπει να χρησιμοποιήσουμε την εντολή rm –r. Για να δούμε όμως, διαγράφεται με αυτή την εντολή rm –r?

```
amarg@amarg-vbox:~/main/exe/intel/x86$ rm -r  ../..
rm: refusing to remove '.' or '..' directory: skipping '../..'
amarg@amarg-vbox:~/main/exe/intel/x86$
```

Ο κατάλογος <u>ΔΕΝ</u> διαγράφεται διότι βρίσκομαι σε υποκατάλογό του !! (ο x86 είναι υποκατάλογος του exe) – είναι σαν να κόβω το κλαδί που κάθομαι: εάν διαγράφω τον κατάλογο exe εγώ τι θα γίνω? Είναι <u>ακριβώς το ίδιο</u> παράδοξο με το να γυρίσει κάποιος πίσω στο χρόνο και να σκοτώσει τον παππού του οπότε εξαφανίζεται και ο ίδιος ☺

9. Από τον κατάλογο main να διαγράψετε τους καταλόγους x86 και hidden.

```
amarg@amarg-vbox:~/main$ rm -r exe/intel/x86
amarg@amarg-vbox:~/main$ rm -r temp/hidden
amarg@amarg-vbox:~/main$
```

10. Από τον κατάλογο bin: (α) να αντιγράψετε το αρχείο names στον κατάλογο x64 με όνομα prog3. (β) Να μετακινήσετε το αρχείο prog1 στον κατάλογο sun. (γ) Να μετακινήσετε το αρχείο list στον κατάλογο tmp.

```
amarg@amarg-vbox:~/main/bin$ cp ../docs/word/names ../exe/intel/x86/prog3
amarg@amarg-vbox:~/main/bin$ mv ../exe/intel/x64 ../exe/sun
amarg@amarg-vbox:~/main/bin$ mv ../docs/word/list ../temp/tmp
amarg@amarg-vbox:~/main/bin$
```

Ας επαληθεύσουμε τις παραπάνω πράξεις: βρίσκονται τα αρχεία στις νέες τους θέσεις? ***Φυσικά και βρίσκονται***!

```
amarg@amarg-vbox:~/main/bin$ ls -l ../exe/intel/x86
total 0
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 20:50 prog3
amarg@amarg-vbox:~/main/bin$ ls -l ../exe/sun
total 0
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:40 prog1
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:40 prog2
amarg@amarg-vbox:~/main/bin$ ls -l ../temp/tmp
total 4
drwxrwxr-x 3 amarg amarg 4096 Οκτ  11 19:47 hidd
-rw-rw-r-- 1 amarg amarg    0 Οκτ  11 19:39 list
amarg@amarg-vbox:~/main/bin$
```

11. Να μεταβείτε στο root directory του συστήματος αρχείων του και να δημιουργήσετε το αρχείο file1. Τι παρατηρείτε?

```
amarg@amarg-vbox:~/main/bin$ cd /
amarg@amarg-vbox:/$ touch file1
touch: cannot touch 'file1': Permission denied
```

Η διαδικασία <u>δεν είναι επιτρεπτή</u> διότι οι απλοί χρήστες μπορούν να δημιουργήσουν αρχεία μόνο στον προσωπικό τους κατάλογο. Μόνο ο διαχειριστής του συστήματος (root) έχει δικαίωμα να δημιουργήσει οπουδήποτε αρχεία.

12. Να καταγράψετε τη μάσκα δικαιωμάτων των αρχείων list, config και prog1 στις τρέχουσες θέσεις τους.

Εάν δεν θυμόμαστε πού βρίσκονται τα παραπάνω αρχεία μπορούμε να τα βρούμε με τη find (ξέρουμε πως βρίσκονται στο δέντρο καταλόγων με ρίζα τη main). Στη συνέχεια εμφανίζουμε τη μάσκα δικαιωμάτων με την ls –l.

```
amarg@amarg-vbox:~$ find main -name list
main/temp/tmp/list
amarg@amarg-vbox:~$ ls -l main/temp/tmp/list
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 main/temp/tmp/list
amarg@amarg-vbox:~$ find main -name config
main/docs/word/config
amarg@amarg-vbox:~$ ls -l main/docs/word/config
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 main/docs/word/config
amarg@amarg-vbox:~$ find main -name prog1
main/exe/sun/prog1
amarg@amarg-vbox:~$ ls -l main/exe/sun/prog1
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:40 main/exe/sun/prog1
amarg@amarg-vbox:~$
```

Παρατηρήστε πως η απόλυτη διαδρομή που επέστρεψε η find διαβιβάστηκε ως όρισμα στην εντολή ls –l έτσι ώστε να εμφανίσει μόνο τη γραμμή που αφορά στο συγκεκριμένο αρχείο. Η μάσκα δικαιωμάτων r w – r w – r - - (664) είναι κοινή για όλα τα αρχεία κάτι που είναι αναμενόμενο. Από την τιμή αυτή της μάσκας οδηγούμαστε στο συμπέρασμα πως η τιμή της umask είναι ίση με 002.

13. Ποιος είναι ο κάτοχος και η ομάδα του αρχείου list? Χρησιμοποιώντας τις εντολές chown και chgrp να αλλάξετε αυτές τις παραμέτρους σε τιμές της αρεσκείας σας.

Από την παραπάνω έξοδο διαπιστώνουμε πως ο κάτοχος και η ομάδα του κατόχου για το αρχείο list είναι ο χρήστης amarg (σε άλλους υπολογιστές φυσικά θα βγάλει διαφορετικά αποτελέσματα). Ας ορίσουμε ως κάτοχο και ως ομάδα κατόχου για το αρχείο list το χρήστη και την ομάδα sys. Η διαδικασία ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ chown sys  main/temp/tmp/list
chown: changing ownership of 'main/temp/tmp/list': Operation not permitted
amarg@amarg-vbox:~$ chgrp sys  main/temp/tmp/list
chgrp: changing group of 'main/temp/tmp/list': Operation not permitted
amarg@amarg-vbox:~$ sudo chown sys  main/temp/tmp/list
amarg@amarg-vbox:~$ sudo chgrp sys  main/temp/tmp/list
amarg@amarg-vbox:~$ ls -l main/temp/tmp/list
-rw-rw-r-- 1 sys sys 0 Οκτ  11 19:39 main/temp/tmp/list
amarg@amarg-vbox:~$
```

Παρατηρήστε πως η απλή κλήση των εντολών δεν είναι επιτρεπτή γιατί οι εντολές chown και chgrp εκτελούνται μόνο από το διαχειριστή του συστήματος (root). Για το λόγο αυτό τις εκτελούμε μέσα από την εντολή sudo δίδοντας τον κωδικό μας αν και όταν ζητηθεί. Στην περίπτωση αυτή οι εντολές εκτελούνται κανονικά όπως φαίνεται από την έξοδο της τελευταίας εντολής ls –l.

14. Η μάσκα δικαιωμάτων του αρχείου list να γίνει 555

```
amarg@amarg-vbox:~$ chmod 555  main/temp/tmp/list
chmod: changing permissions of 'main/temp/tmp/list': Operation not permitted
amarg@amarg-vbox:~$ sudo chown amarg main/temp/tmp/list
amarg@amarg-vbox:~$ sudo chgrp amarg main/temp/tmp/list
amarg@amarg-vbox:~$ chmod 555  main/temp/tmp/list
amarg@amarg-vbox:~$ ls -l main/temp/tmp/list
-r-xr-xr-x 1 amarg amarg 0 Οκτ  11 19:39 main/temp/tmp/list
amarg@amarg-vbox:~$
```

Παρατηρούμε πως αρχικά η εντολή chmod δεν μπορεί να εκτελεστεί και ο λόγος είναι πως δεν είμαστε ο κάτοχος του αρχείου αφού προηγουμένως ως κάτοχο ορίσαμε το χρήστη sys. Εάν όμως χρησιμοποιώντας εκ νέου τις εντολές chown και chgrp επαναφέρουμε τον κάτοχο και την ομάδα του κατόχου στην τιμή amarg, η εντολή εκτελείται χωρίς πρόβλημα. Η εντολή ls –l μας λέει πως η μάσκα του αρχείου list είναι η r – x r – x r – x ή ισοδύναμα 555, όπως άλλωστε ήταν αναμενόμενο.

15. Η μάσκα δικαιωμάτων του καταλόγου word να γίνει 555.

```
amarg@amarg-vbox:~/main$ chmod 555 docs/word
amarg@amarg-vbox:~/main$ ls -l docs/word
total 0
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 config
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 help
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 names
amarg@amarg-vbox:~/main$
```

16. Να διαγράψετε το αρχείο names και να σχολιάσετε το αποτέλεσμα.

```
amarg@amarg-vbox:~/main$ cd docs/word
amarg@amarg-vbox:~/main/docs/word$ ls -l
total 0
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 config
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 help
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 names
amarg@amarg-vbox:~/main/docs/word$ rm names
rm: cannot remove 'names': Permission denied
amarg@amarg-vbox:~/main/docs/word$
```

Παρατηρούμε πως η διαγραφή του αρχείου names δεν είναι επιτρεπτή !! Για ποιο λόγο συμβαίνει αυτό? Ο κατάλογος word έχει μάσκα δικαιωμάτων 555 δηλαδή r – x r – x r – x και κατά συνέπεια καμία ομάδα χρηστών δεν έχει δικαίωμα εγγραφής. Τι σημαίνει εγγραφή για ένα κατάλογο? Το ίδιο ακριβώς που σημαίνει και για ένα αρχείο: δυνατότητα τροποποίησης του περιεχομένου του. Με άλλα λόγια δεν έχουμε δικαίωμα να τροποποιήσουμε τα περιεχόμενα του καταλόγου και ως εκ τούτου η διαγραφή αρχείων από αυτόν (που συνιστά τροποποίηση του περιεχομένου του) δεν είναι επιτρεπτή. Το ίδιο πράγμα θα συνέβαινε και εάν προσπαθούσαμε να δημιουργήσουμε ένα νέο αρχείο σε αυτόν τον κατάλογο όπως φαίνεται στη συνέχεια (αφού και η δημιουργία αρχείου συνιστά τροποποίηση των περιεχομένων του καταλόγου).

```
amarg@amarg-vbox:~/main/docs/word$ rm names
rm: cannot remove 'names': Permission denied
amarg@amarg-vbox:~/main/docs/word$ touch newFile
touch: cannot touch 'newFile': Permission denied
amarg@amarg-vbox:~/main/docs/word$
```

Είναι ενδιαφέρον πως αυτός ο περιορισμός αφορά <u>μόνο στα περιεχόμενα του καταλόγου</u> και <u>όχι</u> στα περιεχόμενα των αρχείων του, τα οποία τροποποιούνται κανονικά !! στο επόμενο παράδειγμα, η εντολή ls –l αποθηκεύει χωρίς

κανένα πρόβλημα την έξοδό της στο αρχείο names τα περιεχόμενα του οποίου επομένως τροποποιούνται, χωρίς όμως το ίδιο να μπορεί να διαγραφεί !!!

```
amarg@amarg-vbox:~/main/docs/word$ ls -l > names
amarg@amarg-vbox:~/main/docs/word$ cat names
total 0
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 config
-rw-rw-r-- 1 amarg amarg 0 Οκτ  11 19:39 help
-rw-rw-r-- 1 amarg amarg 0 Οκτ  12 00:05 names
amarg@amarg-vbox:~/main/docs/word$
```

17. Η μάσκα του καταλόγου temp να γίνει 666

```
amarg@amarg-vbox:~$ ls -l main
total 16
drwxrwxr-x 2 amarg amarg 4096 Οκτ  11 19:40 bin
drwxrwxr-x 4 amarg amarg 4096 Οκτ  11 19:39 docs
drwxrwxr-x 4 amarg amarg 4096 Οκτ  11 20:51 exe
drw-rw-rw- 3 amarg amarg 4096 Οκτ  11 20:42 temp
amarg@amarg-vbox:~$
```

18. Να διαγράψετε τον κατάλογο tmp και να σχολιάσετε το αποτέλεσμα

```
amarg@amarg-vbox:~$ rm -r main/temp/tmp
rm: cannot remove 'main/temp/tmp': Permission denied
amarg@amarg-vbox:~$ cd main/temp
bash: cd: main/temp: Permission denied
amarg@amarg-vbox:~$
```

Παρατηρούμε πως η διαγραφή του καταλόγου tmp δεν είναι επιτρεπτή !! Για ποιο λόγο συμβαίνει αυτό? Ο κατάλογος temp έχει μάσκα δικαιωμάτων 666 δηλαδή r w – r w - r w - και κατά συνέπεια καμία ομάδα χρηστών δεν έχει δικαίωμα εκτέλεσης. Τι σημαίνει εκτέλεση για ένα κατάλογο? Σημαίνει το να μπορεί κανείς να μεταβεί σε αυτόν τον κατάλογο εκτελώντας την εντολή cd (change directory). Είναι προφανές πως για να μπορέσει το σύστημα να διαγράψει τον κατάλογο tmp ο οποίος είναι υποκατάλογος του temp, θα πρέπει πρώτα να μπει σε αυτόν τον κατάλογο με την cd, διαδικασία η οποία δεν είναι επιτρεπτή αφού απαγορεύεται από τα δικαιώματα πρόσβασης. Για το λόγο αυτό η διαγραφή του καταλόγου tmp δεν επιτρέπεται. Αυτό είναι κάτι που πιστοποιείται και από την εκτέλεση της εντολής cd tmp η οποία δεν είναι επιτρεπτή: η μάσκα δικαιωμάτων 666 δεν επιτρέπει τη μετάβαση στον κατάλογο temp!!

19. Η μάσκα του καταλόγου x86 να γίνει 333

```
amarg@amarg-vbox:~$ chmod 333 main/exe/intel/x86
amarg@amarg-vbox:~$ ls -l main/exe/intel
total 4
d-wx-wx-wx 2 amarg amarg 4096 Οκτ  11 20:49 x86
amarg@amarg-vbox:~$
```

20. Να εμφανίσετε τα περιεχόμενα του καταλόγου x64 και να σχολιάσετε το αποτέλεσμα.

```
amarg@amarg-vbox:~$ ls  main/exe/intel/x86
ls: cannot open directory 'main/exe/intel/x86': Permission denied
amarg@amarg-vbox:~$ ls -l  main/exe/intel/x86
ls: cannot open directory 'main/exe/intel/x86': Permission denied
amarg@amarg-vbox:~$
```

Παρατηρούμε πως η εμφάνιση των περιεχομένων του καταλόγου x64 δεν είναι επιτρεπτή !! Για ποιο λόγο συμβαίνει αυτό? Ο κατάλογος x64 έχει μάσκα δικαιωμάτων 333 δηλαδή - w x - w x - w x και κατά συνέπεια καμία ομάδα χρηστών δεν έχει δικαίωμα ανάγνωσης. Τι σημαίνει ανάγνωση για ένα κατάλογο? Σημαίνει να μπορεί κάποιος να

εκτελέσει την ls για να εμφανίσει τα περιεχόμενα του καταλόγου. Επειδή όμως η τρέχουσα μάσκα δικαιωμάτων δεν επιτρέπει κάτι τέτοιο, η εντολή ls δεν είναι δυνατόν να εκτελεστεί. Αυτό είναι κάτι που συνηθίζεται στους καταλόγους αρχείων των ftp servers έτσι ώστε τα περιεχόμενα των καταλόγων του συστήματος να μην είναι ορατά σε όλους.

21. Να οριστεί ως default mode για τους νέους καταλόγους το 744. Ποια είναι η τιμή της μάσκας για τα αρχεία?

    Για να ορίσουμε ως default mode για καταλόγους το 744 θα πρέπει να χρησιμοποιήσουμε τιμή umask ίση με 033 αφού είναι 777 - 033 = 744. Με άλλα λόγια, όλοι οι κατάλογοι που θα δημιουργηθούν μετά τον ορισμό της μάσκας στο 033 θα έχουν δικαιώματα πρόσβασης r w x r - - r - - (744) ενώ όλα τα αρχεία που θα δημιουργηθούν μετά τον ορισμό της μάσκας στο 033 θα έχουν δικαιώματα πρόσβασης r w - r - - r - - (644) (δηλαδή αφαιρούμε το x από όλες τις ομάδες χρηστών) όπως πιστοποιείται από το επόμενο παράδειγμα.

```
amarg@amarg-vbox:~$ umask 033
amarg@amarg-vbox:~$ touch f0003
amarg@amarg-vbox:~$ ls -l f0003
-rw-r--r-- 1 amarg amarg 0 Οκτ  12 01:04 f0003
amarg@amarg-vbox:~$ mkdir d0003
amarg@amarg-vbox:~$ ls -l | grep d0003
drwxr--r--  2 amarg amarg     4096 Οκτ  12 01:04 d0003
amarg@amarg-vbox:~$
```

22. Να εμφανίσετε τα περιεχόμενα του αρχείου /etc/passwd.

```
amarg@amarg-vbox:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
```

(στο screenshot δεν φαίνονται όλες οι γραμμές αλλά μόνο οι πρώτες εξ αυτών)

23. Να αναζητήσετε τα αρχεία του καταλόγου /usr με μέγεθος από 100 έως 500 kbytes και για το καθένα από αυτά να εκτυπώσετε το όνομά τους, το μέγεθός τους και τη μάσκα δικαιωμάτων τους.

```
amarg@amarg-vbox:~$ find /usr -size +100 -size -500 -printf "%f\t%s\t%m\n"
fcnal-test.sh   81322   755
ktest.pl        100428  755
btrfs.h 57736   644
ext4.h  68404   644
emu10k1.h       90803   644
ocelot_hsio.h   56321   644
bpmp-abi.h      71087   644
videodev2.h     91601   644
v4l2-controls.h 52087   644
ethtool.h       74798   644
cec-funcs.h     53025   644
bpf.h   138346  644
```

(στο screenshot δεν φαίνονται όλες οι γραμμές αλλά μόνο οι πρώτες εξ αυτών)

24. Να εκτυπωθεί η απόλυτη διαδρομή της εντολής find

```
amarg@amarg-vbox:~$ which find
/usr/bin/find
```

25. Να εκτυπωθούν οι απόλυτες διαδρομές των αρχείων που περιέχουν στο όνομά τους τη συμβολοσειρά find.

```
amarg@amarg-vbox:~$ whereis find
find: /usr/bin/find /usr/share/man/man1/find.1.gz /usr/share/info/find.info-1.gz
      /usr/share/info/find.info.gz /usr/share/info/find.info-2.gz
```

26. Να εκτυπωθεί το πλήρες περιεχόμενο του root directory – δηλαδή το δικό του και όλων των υποκαταλόγων του.

<u>Στην εικόνα φαίνονται οι λίγες πρώτες από τις χιλιάδες γραμμές της εξόδου της εντολής</u>

```
amarg@amarg-vbox:~$ ls -lR /
/:
total 970048
lrwxrwxrwx   1 root root         7 Σεπ  19 19:50 bin -> usr/bin
drwxr-xr-x   3 root root      4096 Σεπ  26 09:47 boot
drwxrwxr-x   2 root root      4096 Σεπ  19 19:55 cdrom
drwxr-xr-x  18 root root      4060 Οκτ  12 09:42 dev
drwxr-xr-x 130 root root     12288 Οκτ  11 18:19 etc
drwxr-xr-x   3 root root      4096 Σεπ  19 19:57 home
```

27. Να εκτυπωσετε τις 5 πρώτες και τις 5 τελευταίες γραμμές του αρχείου /etc/passwd.

Εμφάνιση πέντε πρώτων γραμμών

```
amarg@amarg-vbox:~$ head -n 5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Εμφάνιση πέντε τελευταίων γραμμών

```
amarg@amarg-vbox:~$ tail -n 5 /etc/passwd
gnome-initial-setup:x:124:65534::/run/gnome-initial-setup/:/bin/false
gdm:x:125:130:Gnome Display Manager:/var/lib/gdm3:/bin/false
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
vboxadd:x:998:1::/var/run/vboxadd:/bin/false
```

28. Να εμφανίσετε τη δεντρική δομή του ριζικού καταλόγου του συστήματος.

Η έξοδος της εντολής είναι τεράστια και εκτυπώνεται ταχύτατα. Για να εμφανίσουμε ένα τμήμα της αποθηκεύουμε την έξοδο στο αρχείο treeOut με τον τελεστή ανακατεύθυνσης εξόδου > (το αρχείο που προκύπτει έχει μέγεθος 48 Mbytes και περιέχει περίπου 628 χιλιάδες γραμμές) και στη συνέχεια εμφανίζουμε τις 20 πρώτες γραμμές του με την εντολή head (στον υπολογιστή μου η εντολή tree δεν είχε εγκατασταθεί κατά την εγκατάσταση του λειτουργικού και το σύστημα ζήτησες και την εγκατέστησε επί τόπου).

```
amarg@amarg-vbox:~$ tree / > treeOut
amarg@amarg-vbox:~$ wc -l treeOut
628428 treeOut
amarg@amarg-vbox:~$ ls -l treeOut
-rw-rw-r-- 1 amarg amarg 48360825 Οκτ  12 10:06 treeOut
amarg@amarg-vbox:~$ head -n 20 treeOut
/
├── bin -> usr/bin
├── boot
│   ├── config-5.4.0-47-generic
│   ├── config-5.4.0-48-generic
│   ├── grub
│   │   ├── fonts
│   │   │   └── unicode.pf2
│   │   ├── gfxblacklist.txt
│   │   ├── grub.cfg
│   │   ├── grubenv
│   │   ├── i386-pc
│   │   │   ├── 915resolution.mod
│   │   │   ├── acpi.mod
│   │   │   ├── adler32.mod
│   │   │   ├── affs.mod
│   │   │   ├── afs.mod
│   │   │   ├── ahci.mod
│   │   │   ├── all_video.mod
│   │   │   ├── aout.mod
amarg@amarg-vbox:~$
```

29. Αναφερόμενοι στην άσκηση με το δέντρο καταλόγων, να μετακινήσετε τον κατάλογο exe και όλο το δέντρο που βρίσκεται κάτω από αυτόν στον κατάλογο bin ευρισκόμενοι στον προσωπικό σας κατάλογο.

Αρχικά θα πρέπει να ορίσουμε τα κατάλληλα δικαιώματα πρόσβασης διότι αυτά που ορίσαμε στις προηγούμενες ασκήσεις δεν επιτρέπουν την εκτέλεση της άσκησης. Από τον προσωπικό μας κατάλογο και χρησιμοποιώντας την εντολή chmod με τη μορφή

chmod –R 755 main

ορίζουμε για τον κατάλογο main καθώς και για οτιδήποτε βρίσκεται μέσα σε αυτόν (αυτό γίνεται με το διακόπτη -R) ως μάσκα δικαιωμάτων, την

755 → 111 101 101 → r w x r – x r - x

και πλέον μπορούμε να εκτελέσουμε την άσκηση. Αν και η εντολή cp με το διακόπτη –r επιτρέπει την αντιγραφή δεντρικής δομής, ενώ η εντολή rm με το διακόπτη –r επιτρέπει τη διαγραφή δεντρικής δομής, ωστόσο, η εντολή mv δεν διαθέτει διακόπτη –r που θα επέτρεπε τη μετακίνηση δεντρικής δομής. Για να το κάνουμε αυτό, θα χρησιμοποιήσουμε πρώτα την cp –r για την αντιγραφή στη νέα θέση και στη συνέχεια την εντολή rm –r για τη διαγραφή από την παλιά θέση με τελικό αποτέλεσμα τη μετακίνηση της δεντρικής δομής από την παλιά στη νέα θέση. Οι παραπάνω εντολές μαζί με την εικόνα της αρχικής δεντρικής δομής παρουσιάζονται στην επόμενη οθόνη

```
amarg@amarg-vbox:~$ chmod -R 755 main
amarg@amarg-vbox:~$ tree main
main
├── bin
├── docs
│   ├── excel
│   └── word
│       ├── config
│       ├── help
│       └── names
├── exe
│   ├── intel
│   │   └── x86
│   │       └── prog3
│   └── sun
│       ├── prog1
│       └── prog2
└── temp
    └── tmp
        ├── hidden
        │   └── src
        │       └── library
        └── list

12 directories, 8 files
amarg@amarg-vbox:~$ cp -r main/exe main/bin
amarg@amarg-vbox:~$ rm -r main/exe
amarg@amarg-vbox:~$
```

ενώ η νέα δεντρική δομή που προέκυψε από την παραπάνω διαδικασία ακολουθεί στη συνέχεια. Από τη μελέτη της επόμενης οθόνης και από τη σύγκρισή της με την προηγούμενη, διαπιστώνουμε πως το δέντρο καταλόγων με ρίζα τον κατάλογο exe μετακινήθηκε όντως κάτω από τον κατάλογο bin.

```
amarg@amarg-vbox:~$ tree main
main
├── bin
│   └── exe
│       ├── intel
│       │   └── x86
│       │       └── prog3
│       └── sun
│           ├── prog1
│           └── prog2
├── docs
│   ├── excel
│   └── word
│       ├── config
│       ├── help
│       └── names
└── temp
    └── tmp
        ├── hidden
        │   └── src
        │       └── library
        └── list

12 directories, 8 files
amarg@amarg-vbox:~$
```

30. Να διαβάσετε το αρχείο βοηθείας της εντολής sort και να ταξινομήσετε τα περιεχόμενα του τρέχοντος καταλόγου κατά φθίνουσα διάταξη ως προς το μέγεθος των αρχείων.

    Το αποτέλεσμα αυτής της διαδικασίας παρουσιάζεται στη συνέχεια (εκτυπώνονται οι λίγες πρώτες γραμμές).

```
amarg@amarg-vbox:~$ ls -l | sort -k 5 -n -r
-rw-rw-r--  1 amarg amarg 48360825 Οκτ  12 10:06 treeOut
-rw-r--r--  1 amarg amarg 41492888 Οκτ   9 12:47 full
--wxr--r--  1 amarg amarg  7959821 Σεπ  20 19:58 file2.doc
-rwxrwxr-x  1 amarg amarg    19624 Σεπ  23 15:10 prog
-rw-------  1 amarg amarg    19624 Σεπ  30 12:13 prog2
-rw-------  1 amarg amarg    19624 Σεπ  29 20:47 prog1
-rwxrwxr-x  1 amarg amarg    17480 Οκτ   7 19:28 sigExample8
-rwxrwxr-x  1 amarg amarg    17328 Σεπ  30 15:59 dirFun
-rwxrwxr-x  1 amarg amarg    17272 Οκτ   2 15:23 pipeEx3
-rwxrwxr-x  1 amarg amarg    17208 Οκτ   2 10:44 pipeEx2
-rwxrwxr-x  1 amarg amarg    17184 Οκτ   6 21:14 sigExample6
```

Ο μοναδικός τρόπος για να ανακτήσουμε το μέγεθος των αρχείων είναι η κλήση της εντολής ls –l, η έξοδος της οποίας θα χρησιμοποιηθεί στη συνέχεια από τη sort, ως είσοδος, για να πραγματοποιήσει την ταξινόμηση. Αυτό μπορεί να γίνει με δύο τρόπους: είτε να αποθηκεύσει η ls –l την έξοδό της σε ένα προσωρινό αρχείο, έστω tempFile ως

ls –l > tempFile

το οποίο στη συνέχεια θα χρησιμοποιηθεί ως input file της sort, η οποία επομένως θα κληθεί ως

sort –k 5 –n –r

είτε να στείλει απευθείας η ls την έξοδό της στη sort μέσω μιας τεχνικής που είναι γνωστή ως διασωλήνωση και η οποία θα μελετηθεί με πολύ μεγάλη λεπτομέρεια σε επόμενο μάθημα

ls -l | sort -k 5 -n –r

(το αποτέλεσμα σε αμφότερες τις περιπτώσεις είναι το ίδιο). Όσον αφορά στη χρήση της sort, ο τρόπος με τον οποίο καλείται τεκμηριώνεται ως εξής: (α) η ταξινόμηση θα πρέπει να γίνει ως προς το μέγεθος του αρχείου, που είναι η πέμπτη στήλη στην έξοδο της ls – l. Αυτό είναι κάτι που γνωστοποιείται στη sort με το διακόπτη –k 5 (αντιλαμβάνεστε επομένως πως ένα θέλαμε η ταξινόμηση να γίνει ως προς το όνομα του αρχείου, που είναι η πρώτη στήλη θα γράφαμε –k 1, k.o.k.). (β) η προεπιλεγμένη συμπεριφορά της sort είναι να ταξινομεί την είσοδό της θεωρώντας πως αυτή αποτελείται από λέξεις και όχι από αριθμούς και ως εκ τούτου δεν πραγματοποιεί αριθμητική ταξινόμηση, όπως θέλουμε, αλλά αλφαριθμητική ταξινόμηση, δηλαδή ταξινομεί τις συμβολοσειρές όπως αυτές εμφανίζονται στον τηλεφωνικό κατάλογο. Με άλλα λόγια, μπορεί, ας πούμε, το 1100 ως αριθμός να είναι μεγαλύτερος από το 543, αλλά επειδή λεξικογραφικά το 1 είναι πριν από το 5, πρώτα θα εμφανίσει το 1100 και μετά το 543 που είναι λάθος. Προκειμένου να ενημερώσουμε τη sort πως θα πρέπει να πραγματοποιήσει αριθμητική ταξινόμηση, δηλαδή να χειριστεί τις συμβολοσειρές ως αριθμούς, προσθέτουμε το διακόπτη –n (από τη λέξη numeric), οπότε η είσοδος θα ταξινομηθεί σωστά (για να το δείτε στην πράξη εκτελέστε την παραπάνω εντολή χωρίς το –n και εξετάστε το αποτέλεσμα). (γ) Η προεπιλεγμένη συμπεριφορά της sort είναι αύξουσα ταξινόμηση, δηλαδή από το μικρότερο προς το μεγαλύτερο. Ωστόσο, εμείς θέλουμε την αντίστροφη, φθίνουσα ταξινόμηση από το μεγαλύτερο προς το μικρότερο και αυτό είναι κάτι που το επιτυγχάνουμε προσθέτοντας το διακόπτη –r (από τη λέξη reverse → αντίστροφος) (για να το δείτε στην πράξη εκτελέστε την παραπάνω εντολή χωρίς το –r και εξετάστε το αποτέλεσμα). Εάν λοιπόν η sort κληθεί με τον παραπάνω τρόπο, θα εμφανίσει το αποτέλεσμα της προηγούμενης οθόνης που είναι αυτό που θέλουμε.

ΕΡΓΑΣΤΗΡΙΟ 2

Κανονικές εκφράσεις – grep & awk

Άσκηση 1. Δημιουργήστε ένα νέο directory κάτω από τον προσωπικό σας κατάλογο με όνομα της επιλογής σας. Προηγουμένως ελέγξτε αν το όνομα υπάρχει και στη συνέχεια αν έχει δημιουργηθεί.

```
amarg@amarg-vbox:~$ mkdir testDir
```

Άσκηση 2. Από τον κατάλογο /etc/ αντιγράψτε στο νέο directory το αρχείο passwd, με όνομα passwd1. Πριν από την αντιγραφή ελέγξτε (με το μάτι, όχι με κώδικα – αυτό θα το δούμε στο επόμενο εργαστήριο) ότι το αρχείο υπάρχει. Μετά την αντιγραφή ελέγξτε αν η αντιγραφή έγινε σωστά.

```
amarg@amarg-vbox:~$ cp /etc/passwd ~/testDir/passwd1
```

Άσκηση 3. Να δημιουργήσετε ένα νέο αρχείο passwd2 που θα περιέχει μόνον τις γραμμές του passwd1 που αρχίζουν με τα γράμματα A-M και a-m.

```
amarg@amarg-vbox:~$ grep '^[A-Ma-m]' ~/testDir/passwd1
```

Η παραπάνω κανονική έκφραση επιστρέφει τις γραμμές εκείνες του αρχείου /etc/passwd1 που ξεκινούν από κεφαλαίο γράμμα A έως M ή από μικρό γράμμα a έως m. Αυτό είναι κάτι που το δηλώνουμε ως [A-Ma-m]. Προκειμένου να δείξουμε πως αυτό το γράμμα είναι το πρώτο γράμμα της γραμμής, βάζουμε μπροστά από την παραπάνω έκφραση το χαρακτήρα ^. Το αποτέλεσμα αυτής της διαδικασίας παρουσιάζεται στη συνέχεια.

```
amarg@amarg-vbox:~$ grep '^[A-Ma-m]' ~/testDir/passwd1
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
messagebus:x:103:106::/nonexistent:/usr/sbin/nologin
avahi-autoipd:x:109:116:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
cups-pk-helper:x:113:120:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
avahi:x:115:121:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
hplip:x:119:7:HPLIP system user,,,:/run/hplip:/bin/false
colord:x:121:126:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
geoclue:x:122:127::/var/lib/geoclue:/usr/sbin/nologin
gnome-initial-setup:x:124:65534::/run/gnome-initial-setup/:/bin/false
gdm:x:125:130:Gnome Display Manager:/var/lib/gdm3:/bin/false
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
```

Προκειμένου το παραπάνω αποτέλεσμα να αποθηκευτεί στο αρχείο ~/testDir/passwd2 ανακατευθύνουμε την έξοδο της grep ως

```
amarg@amarg-vbox:~$ grep '^[A-Ma-m]' ~/testDir/passwd1 > ~/testDir/passwd2
```

Για να επαληθεύστε το αποτέλεσμα εκτελέστε την εντολή **cat ~/testDir/passwd2**.

Άσκηση 4. Να δημιουργήσετε ένα νέο αρχείο passwd3 που θα περιέχει όλους τους χρήστες που χρησιμοποιούν φλοιό bash.

```
amarg@amarg-vbox:~$ grep 'bash$' ~/testDir/passwd1
root:x:0:0:root:/root:/bin/bash
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
```

Η κανονική έκφραση που εμφανίζει μόνο τις γραμμές που τελειώνουν σε bash είναι η 'bash$' με το αποτέλεσμα της εφαρμογής της στο αρχείο ~/testDir/passwd1 να απεικονίζεται παραπάνω. Για να αποθηκεύσουμε την έξοδο της grep στο αρχείο passwd χρησιμοποιούμε την εντολή

grep 'bash$' ~/testDir/passwd1 >> ~/testDir/passwd3

και επαληθεύουμε το αποτέλεσμα εκτελώντας την εντολή **cat ~/testDir/passwd3** όπως φαίνεται παρακάτω.

```
amarg@amarg-vbox:~$ grep 'bash$' ~/testDir/passwd1 >~/testDir/passwd3
amarg@amarg-vbox:~$
amarg@amarg-vbox:~$ cat ~/testDir/passwd3
root:x:0:0:root:/root:/bin/bash
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
amarg@amarg-vbox:~$
```

'Ασκηση 5. Χρησιμοποιείστε την εντολή wc για να μετρήσετε πόσες γραμμές αντιστοιχούν στις προηγούμενες δύο περιπτώσεις και να γράψετε το πλήθος τους στο τέλος των αρχείων passwd2.

wc –l ~/testDir/passwd2 >> ~/testDir/passwd2
wc –l ~/testDir/passwd3 >> ~/testDir/passwd2

Οι δύο παραπάνω εντολές είναι και αυτές που υλοποιούν τη ζητούμενη διαδικασία. Η εντολή **wc –l ~/testDir/passwd2** μετράει τις γραμμές του αρχείου ~/testDir/passwd2 και εκτυπώνει το αποτέλεσμα στην οθόνη. Προκειμένου να αποθηκεύσουμε αυτό το αποτέλεσμα στο αρχείο **χωρίς αυτό να επανεγγραφεί** (δηλαδή να διαγραφούν τα υπάρχοντα περιεχόμενά του και να αντικατασταθούν από τα νέα) δεν χρησιμοποιούμε τον τελεστή > αλλά τον τελεστή **>>** ο οποίος ανοίγει το αρχείο, πηγαίνει στο τέλος του και γράφει το νέο περιεχόμενο αμέσως μετά το προηγούμενο **χωρίς** αυτό να διαγράφεται. Τα αποτελέσματα αυτών των εντολών ακολουθούν στη συνέχεια.

```
amarg@amarg-vbox:~$ wc -l  ~/testDir/passwd2 >> ~/testDir/passwd2
amarg@amarg-vbox:~$ cat  ~/testDir/passwd2
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
messagebus:x:103:106::/nonexistent:/usr/sbin/nologin
avahi-autoipd:x:109:116:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
cups-pk-helper:x:113:120:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
avahi:x:115:121:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
hplip:x:119:7:HPLIP system user,,,:/run/hplip:/bin/false
colord:x:121:126:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
geoclue:x:122:127::/var/lib/geoclue:/usr/sbin/nologin
gnome-initial-setup:x:124:65534::/run/gnome-initial-setup/:/bin/false
gdm:x:125:130:Gnome Display Manager:/var/lib/gdm3:/bin/false
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
22 /home/amarg/testDir/passwd2
amarg@amarg-vbox:~$
```

```
amarg@amarg-vbox:~$ wc -l  ~/testDir/passwd3 >> ~/testDir/passwd3
amarg@amarg-vbox:~$ cat  ~/testDir/passwd3
root:x:0:0:root:/root:/bin/bash
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
2 /home/amarg/testDir/passwd3
amarg@amarg-vbox:~$
```

Παρατηρήστε πως σε αμφότερες τις περιπτώσεις, η έξοδος της wc αποθηκεύεται στο τέλος του αρχείου χωρίς τα περιεχόμενα που ήδη υπήρχαν σε αυτό να έχουν διαγραφεί.

Άσκηση 6. Να δημιουργήσετε ένα αρχείο κειμένου με όνομα test. Στο αρχείο αυτό θα πρέπει να περιέχονται γραμμές με το παρακάτω περιεχόμενο (προφανώς θα περιέχονται και γραμμές με κανονικό κείμενο):

1. **Κενές γραμμές**
2. **Γραμμές μήκους ενός χαρακτήρα**
3. **Γραμμές που περιλαμβάνουν αριθμούς που αντιστοιχούν στους αριθμούς μηνών (π.χ. 02, 2, 11, 9, 07 κλπ)**
4. **Γραμμές που περιλαμβάνουν δεκαδικούς αριθμούς με 1,2 ή 3 το πολύ δεκαδικά ψηφία και 5 το πολύ ακέραια ψηφία (π.χ. 27.59, 1.9, 4589.772)**
5. **Αναγνωριστικά (identifiers) της γλώσσας Java**
6. **Ώρα σε 12ωρη βάση με την ένδειξη am ή pm**
7. **Γραμμές που τελειώνουν με τελεία (.)**

Στη συνέχεια να γράψετε τις κατάλληλες κανονικές εκφράσεις σε εντολές grep ώστε να εμφανίσετε στην οθόνη όλες τις γραμμές που αντιστοιχούν στα παραπάνω.

Για την επίλυση της άσκησης δημιουργούμε το αρχείο test με τα επόμενα ενδεικτικά περιεχόμενα (μπορείτε να κατεβάσετε αυτό το αρχείο από το eclass από το φάκελο Έγγραφα → Εργαστήρια → Λύσεις εργαστηρίων → Εργαστήριο 2 με όνομα test.txt)

```
 1 Hello world !!
 2 Have a nice day !!
 3
 4 r
 5 1
 6 $
 7
 8 02
 9 2
10 11
11 9
12 07
13 Department of Digital Systems
14 University of Thessaly
15 Larissa
16
17
18
19 e
20 0
21 1
22 11
23 456.34
24 12.3434535
25 12132425.01
26 class
27 Mathematics
28 private
29 C++
30 protected
31
32
33
34 12:54
35 03:59
36 45:45
37 Hi.
38 Yes.
39
40
41 11
42
```

Τα ζητούμενα αποτελέσματα προκύπτουν ως εξής:

1. Κενές Γραμμές

```
amarg@amarg-vbox:~$ gedit test
amarg@amarg-vbox:~$ grep '^$' test



amarg@amarg-vbox:~$
```

2. Γραμμές μήκους ενός χαρακτήρα

```
amarg@amarg-vbox:~$ grep '^.$' test
r
1
$
2
9
e
0
1
amarg@amarg-vbox:~$
```

3. Γραμμές που περιλαμβάνουν αριθμούς που αντιστοιχούν στους αριθμούς μηνών (π.χ. 02, 2, 11, 9, 07 κ.τ.λ)

```
amarg@amarg-vbox:~$ egrep '^(0?[1-9]|1[012])$' test
1
2
11
9
07
1
11
11
amarg@amarg-vbox:~$
```

4. Γραμμές που περιλαμβάνουν δεκαδικούς αριθμούς με 1,2 ή 3 το πολύ δεκαδικά ψηφία και 5 το πολύ ακέραια ψηφία (π.χ. 27.59, 1.9, 4589.772)

```
amarg@amarg-vbox:~$ grep '^[0-9][0-9]\{,4\}\.[0-9][0-9]\{,2\}$' test
456.34
amarg@amarg-vbox:~$
```

5. Αναγνωριστικά (identifiers) της γλώσσας Java

```
amarg@amarg-vbox:~$ egrep 'class|private|protected' test
class
private
protected
amarg@amarg-vbox:~$
```

6. Ώρα σε 12ωρη βάση

```
amarg@amarg-vbox:~$ egrep '^(0[0-9]|1[01]):[0-5][0-9]$' test
03:59
amarg@amarg-vbox:~$
```

7. Που τελειώνουν με τελεία

```
amarg@amarg-vbox:~$ grep '^.*\.$' test
Hi.
Yes.
amarg@amarg-vbox:~$
```

Άσκηση 7. Να κατασκευαστεί κανονική έκφραση που να αναγνωρίζει διευθύνσεις email του Πανεπιστημίου Θεσσαλίας με login name το οποίο να αποτελείται από τουλάχιστον οκτώ γράμματα (κεφαλαία ή/και μικρά) ή/και αριθμούς.

Η ζητούμενη κανονική έκφραση έχει την ακόλουθη μορφή σύμφωνα με την οποία οι λέξεις που παράγονται ή γίνονται δεκτές από την κανονική έκφραση έχουν τουλάχιστον 8 *({8,})* χαρακτήρες που μπορεί να είναι κεφαλαίοι ([A-Z]) ή μικροί ([a-z]) αγγλικοί χαρακτήρες ή ψηφία ([0-9]) ενώ αμέσως μετά του οκτώ χαρακτήρες ακολουθεί η σταθερή υποσυμβολοσειρά @uth.gr η οποία πρέπει να εμφανίζεται αυτούσια ως έχει (αφού όλα τα emails του Πανεπιστημίου Θεσσαλίας τελειώνουν με αυτή).

[A-Za-z0-9]{8,}@uth.gr

Για τον έλεγχο της ορθότητας της κανονικής έκφρασης κατασκευάσαμε το αρχείο emails περιεχόμενα (μπορείτε να κατεβάσετε αυτό το αρχείο από το eclass από το φάκελο Έγγραφα → Εργαστήρια → Λύσεις εργαστηρίων → Εργαστήριο 2 με όνομα test.txt) με περιεχόμενο

```
amarg@amarg-vbox:~/shared$ cat emails.txt
amarg@uom.gr
atmargaris@uth.gr
isavvas@uth.gr
amarg200x@yahoo.gr
amarg@teilar.gr
karetsos@uth.gr
amarg@amarg-vbox:~/shared$
```

Στην περίπτωση αυτή, η χρήση της egrep με αυτή την κανονική έκφραση και το παραπάνω αρχείο θα μας δώσει

```
amarg@amarg-vbox:~/shared$ egrep '[A-Za-z0-9]{8,}@uth.gr' emails.txt
atmargaris@uth.gr
karetsos@uth.gr
amarg@amarg-vbox:~/shared$
```

και δεν είναι δύσκολο να διαπιστώσει κανείς, πως τα αποτελέσματα στα οποία οδηγεί είναι πράγματι σωστά.

Άσκηση 8. Δίδεται το επόμενο αρχείο κειμένου με όνομα results το οποίο περιέχει τις επιδόσεις ενός φοιτητή κατά τη διάρκεια ενός χρονικού διαστήματος.

```
PROGRAMMING EAR  2010   7
DATABASES   XEIM 2011   6
UNIX        EAR  2010   4
NETWORKS    EAR  2010  10
JAVA        XEIM 2011   8
MATH1       XEIM 2010   2
PHYSICS     XEIM 2010   5
C++         XEIM 2011   7
ENGLISH     XEIM 2010   9
MATH2       EAR  2011   5
```

Χρησιμοποιώντας τη γλώσσα awk να εκτυπώσετε τα μαθήματα που πέρασε ο φοιτητής κατά την εξεταστική περίοδο XEIM 2010 και να υπολογίσετε το μέσο όρο των βαθμών για αυτά τα μαθήματα.

Το πρόγραμμα awk που επιλύει το παραπάνω πρόβλημα μπορεί να κληθεί μέσα από τη γραμμή εντολών ως

```
awk 'BEGIN { count=0; sum=0; mean } /XEIM 2010/ { sum+=$4; count++; print $0 } END { mean=sum/count; print "Mean="mean}' results
```

ή εναλλακτικά, να αποθηκευτεί σε αρχείο το οποίο θα περαστεί ως όρισμα στην awk ως

awk -f awkscript results

όπου το αρχείο awkscript έχει το περιεχόμενο που ακολουθεί στη συνέχεια

```
BEGIN {
  count=0;
  sum=0;
  mean=0
 }
```

Το BEGIN block περιέχει εντολές που εκτελούνται μία και μοναδική φορά στην αρχή. Εδώ συνήθως δηλώνονται και αρχικοποιούνται μεταβλητές. Ο ζητούμενος μέσος όρος θα προκύψει ως mean = sum / count όπου count το πλήθος των μαθημάτων που πληρούν το κριτήριο και sum το άθροισμα *των βαθμών για αυτά τα μαθήματα.*

```
 /XEIM 2010/ {
  sum+=$4;
  count++;
  print $0
 }
```

Ο κώδικας που ακολουθεί θα εκτελεστεί ΜΟΝΟ για εκείνες τις γραμμές που περιέχουν οπουδήποτε τη συμβολοσειρά XEIM 2010. Για κάθε τέτοια γραμμή το count αυξάνεται κατά ένα η τιμή του $4 που περιέχει το βαθμό του μαθή-*ματος προστίθεται στο sum ενώ ολόκληρη η γραμμή ($0) εκτυπώνεται στην* οθόνη του χρήστη.

```
END {
  mean=sum/count;
  print "Mean="mean
 }
```

Το END block περιέχει εντολές που εκτελούνται μία και μοναδική φορά στο *τέλος. Οι μεταβλητές sum και count έχουν τις σωστές τιμές οπότε υπολογίζουμε το μέσο όρο τους ως mean = sum / count και εκτυπώνουμε το αποτέλεσμα.*

(μπορείτε να κατεβάσετε το αρχείο results από το eclass από το φάκελο Έγγραφα → Εργαστήρια → Λύσεις εργαστηρίων → Εργαστήριο 2 με όνομα test.txt) .

Το αποτέλεσμα είναι προκύπτει ως εξής:

```
amarg@amarg-vbox:~$ awk -f awkscript results
MATH1       XEIM 2010   2
PHYSICS     XEIM 2010   5
ENGLISH     XEIM 2010   9
Mean=5,33333
```

ΕΡΓΑΣΤΗΡΙΟ 3

Pipelines & Shell programming

1. **Να προσδιορίσετε το όνομα του φλοιού με τον οποίο συνδέεστε στο σύστημα.**

   Το όνομα του φλοιού σύνδεσης στο σύστημα περιέχεται στη μεταβλητή περιβάλλοντος SHELL η τιμή της οποίας εμφανίζεται με τον ακόλουθο τρόπο

```
amarg@amarg-vbox:~$ echo $SHELL
/bin/bash
```

   Επομένως ο φλοιός σύνδεσης είναι ο /bin/bash.

2. **Να εμφανίσετε τα κρυφά αρχεία του προσωπικού σας καταλόγου και να εκτυπώσετε στην οθόνη τα περιεχόμενά τους. Ποια από αυτά σχετίζονται με τη διαδικασία σύνδεσης στο σύστημα?**

   Στο λειτουργικό σύστημα Linux (και γενικά στο Unix) τα κρυφά αρχεία είναι εκείνα το όνομα των οποίων ξεκινά με τελεία. Προκειμένου να εμφανίσουμε αυτά τα αρχεία καλούμε την εντολή ls με το διακόπτη –a (all) ενώ για να εμφανίσουμε μόνο αυτά τα αρχεία και όχι όλα στέλνουμε την έξοδο της ls στην grep με κανονική έκφραση την '^\.' Το αποτέλεσμα είναι:

```
amarg@amarg-vbox:~$ ls -a | grep '^\.'
.
..
.bash_history
.bash_logout
.bashrc
.cache
.config
.ddd
.gnupg
.local
.profile
.sudo_as_admin_successful
.vboxclient-clipboard.pid
.vboxclient-display-svga-x11.pid
.vboxclient-draganddrop.pid
.vboxclient-seamless.pid
amarg@amarg-vbox:~$
```

   Τα αρχεία που σχετίζονται με τη διαδικασία σύνδεσης που περιέχουν στο όνομά τους τη λέξη bash (επειδή ο φλοιός σύνδεσης είναι το bash shell). Αυτά τα αρχεία είναι τα

```
.bash_history
.bash_logout
.bashrc
```

3. **Να εκτυπώσετε τον κατάλογο των διαθέσιμων φλοιών του υπολογιστικού σας συστήματος.**

   Τα ονόματα των διαθέσιμων φλοιών του συστήματος περιλαμβάνονται στο αρχείο /etc/shells τα περιεχόμενα του οποίου εκτυπώνονται με την εντολή cat. Αυτά είναι

```
amarg@amarg-vbox:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
```

4. **Να μετρήσετε το πλήθος των χρηστών του συστήματος που συνδέονται στο σύστημα με το φλοιό bash.**

   Για κάθε χρήστη του συστήματος, ο φλοιός σύνδεσης είναι το τελευταίο πεδίο της γραμμής για αυτόν το χρήση στο αρχείο /etc/passwd. Για να εμφανίσουμε αυτές τις γραμμές καλούμε την grep με κανονική έκφραση την 'bash$'.

```
amarg@amarg-vbox:~$ grep 'bash$' /etc/passwd
root:x:0:0:root:/root:/bin/bash
amarg:x:1000:1000:Athanasios Margaris,,,:/home/amarg:/bin/bash
```

   Προκειμένου να μετρήσουμε το πλήθος αυτών των χρηστών θα πρέπει να περάσουμε την παραπάνω έξοδο στην είσοδο της wc με διακόπτη –l για να μετρήσει γραμμές και φυσικά το αποτέλεσμα αυτής της εντολής είναι ίσο με 2, αφού υπάρχουν δύο τέτοιες γραμμές.

```
amarg@amarg-vbox:~$ grep 'bash$' /etc/passwd | wc -l
2
```

5. Να εκτυπώσετε τις τιμές των μεταβλητών περιβάλλοντος USER, HOME και SHELL. Τι παρατηρείτε?

   Οι τιμές αυτών των μεταβλητών περιβάλλοντος εκτυπώνονται με την εντολή echo – το αποτέλεσμα είναι

```
amarg@amarg-vbox:~$ echo $USER
amarg
amarg@amarg-vbox:~$ echo $HOME
/home/amarg
amarg@amarg-vbox:~$ echo $SHELL
/bin/bash
```

   Παρατηρούμε πως αυτές οι μεταβλητές περιβάλλοντος περιέχουν το όνομα του χρήστη (login name), τον προσωπικό κατάλογο (home directory) και το φλοιό σύνδεσης (login shell).

6. Να βρεθεί το αποτέλεσμα της εκτέλεσης των παρακάτω εντολών :

   ls | wc -l
   who | wc -l
   ls *.c | wc -l
   who | sort

- Η πρώτη εντολή μετρά το πλήθος των περιεχομένων του τρέχοντος καταλόγου.
- Η δεύτερη εντολή μετρά το πλήθος των συνδεδεμένων χρηστών οι οποίοι εκτυπώνονται για την εντολή who.
- Η τρίτη εντολή μετρά το πλήθος των αρχείων του τρέχοντος καταλόγου με κατάληξη .c.
- Η τέταρτη εντολή ταξινομεί τους συνδεδεμένους χρήστες του συστήματος με βάση το όνομά τους.

Εκτελέστε στους υπολογιστές σας τις παραπάνω εντολές και καταγράψτε το αποτέλεσμα.

PIPELINES

Άσκηση 1. Να μετρηθεί το πλήθος των γραμμών του αρχείου /etc/passwd

```
amarg@amarg-vbox:~$ cat /etc/passwd | wc -l
48
amarg@amarg-vbox:~$
```

Η wc με το διακόπτη –l μετρά τις γραμμές της εισόδου της που την λαμβάνει από την cat μέσω διασωλήνωσης. Λαμβάνοντας υπόψη πως σε αυτό το αρχείο υπάρχει μία γραμμή για κάθε χρήστη, στον υπολογιστή που εκτελέστηκε η εντολή υπάρχου συνολικά δηλωμένοι 48 χρήστες.

Άσκηση 2. Χρησιμοποιώντας διασωλήνωση να βρείτε το πλήθος των καταλόγων που υπάρχουν στον προσωπικό σας κατάλογο (πριν την εκτέλεση της εντολής εκτελέστε την cd <ENTER>).

```
amarg@amarg-vbox:~$ ls -l | grep '^d' | wc -l
30
amarg@amarg-vbox:~$
```

Οι γραμμές στην έξοδο της ls –l που αναφέρονται σε καταλόγους ξεκινούν με το γράμμα d. Προκειμένου να κρατήσουμε μόνο αυτές τις γραμμές διαβιβάζουμε την έξοδο της ls –l στην εντολή grep με κανονική έκφραση την ‘^d’ η οποία περιγράφει τις γραμμές που ξεκινούν με το γράμμα d. Στη συνέχεια η grep στέλνει την έξοδό της στην εντολή wc η οποία με το διακόπτη –l μετρά τις γραμμές που εκτύπωσε η grep, δηλαδή τις γραμμές που ξεκινούν από d και κατά συνέπεια περιγράφουν καταλόγους. Στον υπολογιστή στον οποίο εκτελέστηκε αυτό το pipeline βρέθηκαν 30 κατάλογοι ενώ σε άλλους υπολογιστές το αποτέλεσμα θα είναι προφανώς διαφορετικό. Για να κατανοήσετε τον τρόπο με τον οποίο κατασκευάζεται σταδιακά το pipeline, αρχικά εκτελέστε την εντολή **ls –l**, ύστερα την εντολή **ls –l | grep ‘^d’** και τέλος την εντολή **ls – l | grep ‘^d’ | wc –l** και παρατηρήστε τα ενδιάμεσα αποτελέσματα.

Άσκηση 3. Να εκτυπωθεί το μήκος του username από το αρχείο /etc/passwd με τη μέγιστη τιμή.

```
amarg@amarg-vbox:~$ cat /etc/passwd | awk -F: '{print $1}' | wc -L
19
amarg@amarg-vbox:~$
```

Η εντολή awk δέχεται ως είσοδο τα περιεχόμενα του αρχείου /etc/passwd που της αποστέλλει η cat. Επειδή σε αυτό το αρχείο η κάθε γραμμή περιέχει πεδία που διαχωρίζονται μεταξύ τους με το χαρακτήρα : θα πρέπει να γνωστοποιήσουμε στην awk πως ο διαχωριστής πεδίων σε κάθε γραμμή είναι ο : κάτι που γίνεται με το διακόπτη –F:. Στην περίπτωση αυτή η awk εκτελεί για κάθε γραμμή την εντολή print $1 η οποία εκτυπώνει στην έξοδό της μόνο το πρώτο πεδίο ($1) της κάθε γραμμής, που είναι το username του εκάστοτε χρήστη. Με άλλα λόγια, η έξοδος της awk είναι μία στήλη με τα usernames των χρηστών. Αυτή η έξοδος αποστέλλεται στην εντολή wc η οποία με το διακόπτη –L προσδιορίζει τη λέξη με το μεγαλύτερο μήκος και εκτυπώνει την τιμή αυτού του μήκους. Στον υπολογιστή που εκτελέστηκε η εντολή, το μέγιστο μήκος username είναι το 19. Για να κατανοήσετε τον τρόπο με τον οποίο κατασκευάζεται σταδιακά το pipeline αρχικά εκτελέστε την εντολή **cat /etc/passwd**, στη συνέχεια την εντολή **cat /etc/passwd | awk –F: ‘{ print $1 }’** και τέλος την εντολή **cat /etc/passwd | awk –F: ‘{ print $1 }’ | wc –L** και παρατηρήστε τα ενδιάμεσα αποτελέσματα.

Άσκηση 4. Όπως στην Άσκηση 3, αλλά επιπλέον να εκτυπωθεί το username που αντιστοιχεί σε αυτό το μήκος καθώς και η γραμμή του αρχείου που το περιέχει.

cat /etc/passwd | awk -F: 'BEGIN {line=0;maxlength=0} {if (length($1)>maxlength) {maxlength=length($1);line=$0}} END {print line;print maxlength}'

Η εκτέλεσης της παραπάνω εντολής στον υπολογιστή που γράφτηκε το πρόγραμμα μας δίνει

```
amarg@amarg-vbox:~$ cat /etc/passwd | awk -F: 'BEGIN {line=0;maxlength=0} {if (length($1)>maxlength)
{maxlength=length($1);line=$0}} END {print line;print maxlength}'
gnome-initial-setup:x:124:65534::/run/gnome-initial-setup/:/bin/false
19
amarg@amarg-vbox:~$
```

Με άλλα λόγια, το username με το μέγιστο μήκος είναι το **gnome-initial-setup** με μήκος 19 χαρακτήρες.

ΣΧΟΛΙΟ. Επειδή στην προκειμένη περίπτωση το πρόγραμμα είναι αρκετά εκτεταμένο και δυσανάγνωστο, είναι προτιμότερο, όπως έχουμε σχολιάσει και στο μάθημα να το καταχωρήσουμε σε ένα script file το οποίο στη συνέχεια θα περάσουμε ως όρισμα στην awk με το διακόπτη –f. Δημιουργούμε λοιπόν το αρχείο awkScript με περιεχόμενο

```
BEGIN {
   line=0;
   maxlength=0
   }

{
   if (length($1)>maxlength) {
      maxlength=length($1);
      line=$0}
}
END {
   print line;
   print maxlength
}
```

και στη συνέχεια, πολύ απλά γράφουμε (λαμβάνοντας φυσικά το ίδιο αποτέλεσμα με πριν)

cat /etc/passwd | awk -F: -f awkscript

ή ακόμη πιο απλά

awk -F: -f awkscript /etc/passwd

Πώς λειτουργεί το πρόγραμμα: *θυμηθείτε πως στο BEGIN block γράφουμε εντολές που εκτελούνται μία και μοναδική φορά στην αρχή, στο END block γράφουμε εντολές που εκτελούνται μία και μοναδική φορά στο τέλος, ενώ ενδιάμεσο block (MAIN block) γράφουμε εντολές που εκτελούνται για κάθε γραμμή του αρχείου εισόδου, δηλαδή για κάθε γραμμή του αρχείου /etc/passwd. Για να προσδιορίσουμε το login name με το μεγαλύτερο μήκος αλλά και την τιμή αυτού του μήκους, στο BEGIN block ορίζουμε και αρχικοποιούμε δύο βοηθητικές μεταβλητές, την line που θα περιέχει σε κάθε επανάληψη τη γραμμή με το login name που έχει το τρέχον μέγιστο μήκος (δηλαδή το μέγιστο μήκος που έχει βρεθεί μέχρι αυτή την επανάληψη) και τη μεταβλητή maxlength που περιέχει το τρέχον μέγιστο μήκος. Το login name για την κάθε γραμμή – θυμηθείτε πως ολόκληρη η τρέχουσα γραμμή αποθηκεύεται στο $0 – περιέχεται στο $1, ενώ το μήκος συμβολοσειράς στην awk υπολογίζεται από τη συνάρτηση length – δηλαδή η συνάρτηση length($1) υπολογίζει το μήκος της συμβολοσειράς $1. Τα υπόλοιπα είναι όπως στη C: εάν κατά την τρέχουσα επανάληψη το length($1) είναι μεγαλύτερο από το τρέχον maxlength, θέτουμε maxlength=length($1) και line=$0. Στο τέλος της επεξεργασίας το maxlength θα περιέχει το μέγιστο μήκος ενώ η μεταβλητή line το login name για αυτό το μέγιστο μήκος, οπότε στο END block απλά εκτυπώνουμε τις τιμές τους.*

Άσκηση 5. Να εκτυπωθεί η γραμμή της εξόδου της εντολής ls –l εφαρμοζόμενη στον τρέχοντα κατάλογο για το αρχείο με το μεγαλύτερο μέγεθος.

Με χρήση της awk (η λογική είναι **ακριβώς η ίδια** με πριν, απλά τώρα η είσοδος στην awk έρχεται από το pipeline **ls –l | grep '^-'** που εκτυπώνει τις γραμμές της ls –l που ξεκινούν με – και κατά συνέπεια αναφέρονται σε **αρχεία**)

```
amarg@amarg-vbox:~$ ls -l | grep '^-' | awk 'BEGIN {size=0}{ if ($5>size) {size=$5;line=$0}}
END{ print line; print size }'
-rw-rw-r--  1 amarg amarg 48360825 Οκτ  12 10:06 treeOut
48360825
amarg@amarg-vbox:~$
```

Με άλλα λόγια, στον υπολογιστή που εκτελέστηκε το πρόγραμμα, το αρχείο με το μεγαλύτερο μέγεθος είναι το treeOut ενώ το μέγεθός του είναι **48360825** bytes. Μπορείτε ως άσκηση να αποθηκεύσετε το παραπάνω πρόγραμμα σε ένα αρχείο το οποίο θα περάσετε στην awk με το διακόπτη –f.

Με χρήση της sort

```
amarg@amarg-vbox:~$ ls -l | sort -k 5 -n -r | head -n 1
-rw-rw-r--  1 amarg amarg 48360825 Οκτ  12 10:06 treeOut
amarg@amarg-vbox:~$
```

Η ls –l στέλνει την έξοδό της στη sort η οποία κάνει φθίνουσα (-r), αριθμητική (-n) ταξινόμηση ως προς το μέγεθος του αρχείου που είναι η πέμπτη στήλη στην έξοδο της ls –l (-k 5). Στη συνέχεια η έξοδος της sort διαβιβάζεται στην εντολή head η οποία με το διακόπτη –n εκτυπώνει μόνο την πρώτη γραμμή της εξόδου της sort, η οποία (αφού η sort έκανε φθίνουσα ταξινόμηση) αναφέρεται στο αρχείο με το μεγαλύτερο μέγεθος. Για να κατανοήσετε τον τρόπο με τον οποίο κατασκευάζεται σταδιακά το pipeline αρχικά εκτελέστε την εντολή **ls -l**, στη συνέχεια την **εντολή ls – l | sort –k 5 –n –r** και τέλος την **εντολή ls – l | sort –k 5 –n –r | head –n 1** και παρατηρήστε τα ενδιάμεσα αποτελέσματα.

Για πιο λεπτομερείς οδηγίες σχετικά με τη χρήση της sort δείτε τη **λύση της Άσκησης 30** του Εργαστηρίου 1.

‘Ασκηση 6. Για κάθε αρχείο του καταλόγου /usr/bin το όνομα του οποίου ξεκινά από a να εκτυπώσετε το πλήρες όνομα της διαδρομής του, τα δικαιώματα πρόσβασης και το μέγεθός του. Στη συνέχεια χρησιμοποιώντας τη γλώσσα AWK να προσδιορίσετε το άθροισμα των μεγεθών αυτών των αρχείων.

Αρχικά εκτελούμε την εντολή find με τον τρόπο που φαίνεται στη συνέχεια όπου εκτυπώνονται οι πρώτες γραμμές της εξόδου της εντολής (τα αρχεία που επιστρέφονται είναι πολύ περισσότερα)

```
amarg@amarg-vbox:~$ find /usr/bin -name 'a*' -printf "%f\t%m\t%s\n"
appres  755     14648
apt-cache       755     88536
apt-sortpkgs    755     47504
apt-get 755     47576
alsabat 755     47552
alsaucm 755     31528
apport-unpack   755     2068
apropos 777     6
avahi-publish-address   777     13
avahi-publish-service   777     13
avahi-resolve   755     22768
```

Προκειμένου να προσδιορίσουμε το άθροισμα των μεγεθών αυτών των αρχείων θα περάσουμε την παραπάνω έξοδο ως είσοδο στην awk παρατηρώντας πως σε κάθε γραμμή το μέγεθος εκτυπώνεται στην τρίτη στήλη και κατά συνέπεια θα καταχωρηθεί στη μεταβλητή $3 (εδώ ως διαχωριστής πεδίων χρησιμοποιείται το tab που μαζί με το space αποτελούν τους προεπιλεγμένους διαχωριστές και ως εκ τούτου δεν χρειάζεται να κάνουμε κάτι – όπως κάναμε με το αρχείο /etc/passwd όπου ως διαχωριστής χρησιμοποιείται ο χαρακτήρας : και για το λόγο αυτό χρησιμοποιήσαμε στην awk το διακόπτη –F:). Για να υπολογίσουμε το άθροισμα των μεγεθών των αρχείων που επιστρέφονται από τη find – προσθέτοντας μεταξύ τους τις τιμές της μεταβλητής $3 για την κάθε γραμμή, αρχικοποιούμε στο BEGIN block τη μεταβλητή sum=0, η οποία στο main block

προσαυξάνεται κατά την τρέχουσα τιμή της μεταβλητής $3, με μία εντολή της μορφής sum += $3. Μετά την ολοκλήρωση της επεξεργασίας της εισόδου, η μεταβλητή sum περιέχει τη ζητούμενη τιμή, η οποία εκτυπώνεται στο END block. Η παραπάνω διαδικασία και το αποτέλεσμά της παρουσιάζονται στη συνέχεια.

```
amarg@amarg-vbox:~$ find /usr/bin -name 'a*' -printf "%f\t%m\t%s\n"
| awk 'BEGIN {sum=0}{ sum+=$3 } END {print sum}'
1912769
amarg@amarg-vbox:~$
```

SHELL PROGRAMMING

Για την κατανόηση των παραδειγμάτων και των ασκήσεων που ακολουθούν, θα είναι καλό να κάνετε μία καλή επανάληψη τη σχετική θεωρία από το σχετικό αρχείο παρουσίασης στο eClass ***(Έγγραφα → Θεωρία → 03. Προγραμματίζοντας στο κέλυφος)****. Για κάθε παράδειγμα ή λυμένη άσκηση (εντός παρενθέσεων και με μπλε χρώμα) αναφέρεται το όνομα του αρχείου που περιέχει τον κώδικα έτσι ώστε να μην χρειαστεί να τον πληκτρολογήσετε από την αρχή (οπότε εστιάστε την προσοχή σας στο να κατανοήσετε τι ακριβώς κάνει). Αυτά τα αρχεία υπάρχουν επίσης eClass* ***(Έγγραφα → Εργαστήρια → Λύσεις Εργαστηρίων → Εργαστήριο 3)****.*

Παραδείγματα

1. ***Παράδειγμα While Loop που εκτελείται πέντε φορές*** **(αρχείο lab3ex1).** *Το while loop εκτελείται επ' άπειρον αφού η τιμή του valid είναι πάντα true (δεν αλλάζει μέσα στο loop). Η αρχική τιμή του count είναι 1 και σε κάθε κύκλο επανάληψης αυξάνεται κατά μία μονάδα. Όταν η τιμή του count γίνει ίση με 5, η συνθήκη στο εσωτερικό if γίνεται αληθής και εκτελείται η εντολή break η οποία τερματίζει το loop. Επομένως, αυτό το loop εκτελείται πέντε φορές.*

```
#!/bin/bash
valid=true
count=1
while [ $valid ]
   do
        echo $count
        if [ $count -eq 5 ];
        then
           break
        fi
   ((count++))
   done
```

Η έξοδος του προγράμματος παρουσιάζεται στη συνέχεια

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex1
1
2
3
4
5
amarg@amarg-vbox:~/shared/Lab3$
```

2. Σύγκριση αλφαριθμητικών και λογικοί τελεστές **(αρχείο lab3ex2).** *Το πρόγραμμα επιδεικνύει τη διαδικασία σύγκρισης αλφαριθμητικών και τη χρήση λογικών τελεστών. Ο χρήστης καλείται να δώσει ένα username και ένα password. Εάν ο χρήστης καταχωρήσει ως login το admin* ***KAI*** *ως password το secret, το πρόγραμμα εκτυπώνει το μήνυμα valid user, ενώ στην αντίθετη περίπτωση εκτυπώνεται το μήνυμα invalid user.*

```
#!/bin/bash
echo "Enter username"
read username
echo "Enter password"
read password
if [[ ( $username == "admin" && $password == "secret" ) ]]; then echo "valid user"
else echo "invalid user"
fi
```

Η έξοδος του προγράμματος παρουσιάζεται στη συνέχεια

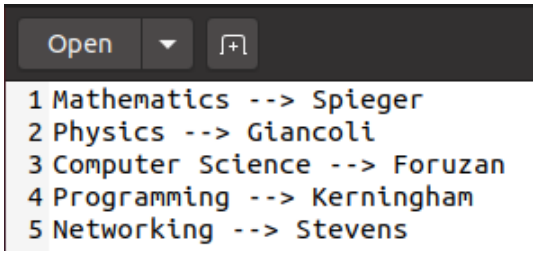
```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex2
Enter username
amarg
Enter password
digital
invalid user
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex2
Enter username
amarg
Enter password
secret
invalid user
```

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex2
Enter username
admin
Enter password
digital
invalid user
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex2
Enter username
admin
Enter password
secret
valid user
```

3. Ανάγνωση αρχείου (αρχείο lab3ex3). *Το πρόγραμμα επιδεικνύει την είσοδο από αρχείο με τον τελεστή **<** που υπάρχει στο τέλος του κώδικα. Το πρόγραμμα μέσα από ένα βρόχο επανάληψης διαβάζει μία προς μία τις γραμμές του αρχείου ενώ όταν φτάσει στο τέλος του αρχείου και δεν υπάρχει τίποτε άλλο να διαβάσει, η διαδικασία ολοκληρώνεται. Για την εκτέλεση του κώδικα κατασκευάστε ένα αρχείο με όνομα book.txt που να περιέχει κάποιες γραμμές κειμένου – γράψτε ό,τι θέλετε – και αποθηκεύστε το στον ίδιο κατάλογο με το αρχείο του κώδικα. Εναλλακτικά μπορείτε για συντομία να κατασκευάσετε το αρχείο book.txt αποθηκεύοντας σε αυτό την έξοδο κάποιας εντολής, π.χ. ως ls –l > book.txt.*

```
#!/bin/bash
file='book.txt'
while read line; do
    echo $line
done < $file
```

Η έξοδος του προγράμματος → αρχικά κατασκευάζουμε ένα κατάλογο book.txt με ενδεικτικό περιεχόμενο με κάποιον editor (π.χ. με το gedit). Το πρόγραμμα αποθηκεύεται στο ίδιο αρχείο με το shell script.



Το αρχείο εισόδου δεν περνά ως όρισμά αλλά υπάρχει μέσα στον κώδικα οπότε απλά καλούμε το αρχείο κελύφους. Το αποτέλεσμα ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex3
Mathematics --> Spieger
Physics --> Giancoli
Computer Science --> Foruzan
Programming --> Kerningham
Networking --> Stevens
amarg@amarg-vbox:~/shared/Lab3$
```

4. Σύγκριση των περιεχομένων δύο καταλόγων. (αρχείο lab3ex4)

```
#!/bin/bash
# cmp_dir - program to compare two directories
```

Για να εκτελεστεί το πρόγραμμα θα πρέπει ο χρήστης να δώσει δύο ορίσματα που να περιγράφουν ονόματα καταλόγων. Εάν λοιπόν η τιμή της μεταβλητής $# που περιέχει το πλήθος των ορισμάτων της

γραμμής εντολών δεν είναι ίσος με 2, το πρόγραμμα δεν μπορεί να εκτελεστεί – για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
# Check for required arguments

if [ $# -ne 2 ]; then
    echo "usage: $0 directory_1 directory_2"
    exit 1
fi
```

Για να εκτελεστεί το πρόγραμμα θα πρέπει τα δύο ορίσματα που έδωσε ο χρήστης να αντιστοιχούν αμφότερα σε καταλόγους και όχι π.χ. σε αρχεία. Εάν αυτό δεν συμβαίνει, το πρόγραμμα δεν μπορεί να **εκτελεστεί – για το λόγο αυτό εκτυπώνει ένα ενημερωτικό μήνυμα όπου απαιτείται και τερματίζει τη λειτουργία του.**

```
# Make sure both arguments are directories

if [ ! -d $1 ]; then
    echo "$1 is not a directory!"
    exit 1
fi
if [ ! -d $2 ]; then
    echo "$2 is not a directory!"
    exit 1
fi
```

Εάν ο κώδικας φτάσει σε αυτό το σημείο και δεν έχει τερματιστεί προηγουμένως, αυτό σημαίνει πως ο χρήστης έχει δώσει ακριβώς δύο ονόματα που περιγράφουν καταλόγους και κατά συνέπεια πληρούνται οι προϋποθέσεις εκτέλεσης του κώδικα. Η σύγκριση των περιεχομένων των δύο καταλόγων πραγματοποιείται ελέγχοντας εάν το κάθε όνομα αντικειμένου του πρώτου καταλόγου είναι αρχείο και αν ναι, εάν ανήκει στο δεύτερο κατάλογο. Εάν αυτό δεν συμβαίνει, τότε το εν λόγω αρχείο απουσιάζει από τον δεύτερο κατάλογο, οπότε η μεταβλητή missing που αρχικά είχε τεθεί στο μηδέν αυξάνεται κατά μία μονάδα. Στο τέλος της σύγκρισης το πρόγραμμα εκτυπώνει το πλήθος των αρχείων που υπάρχουν στον πρώτο κατάλογο αλλά όχι στο δεύτερο.

```
# Process each file in directory_1, comparing it to directory_2
missing=0

for filename in $1/*; do

    fn=$(basename $filename)
    if [ -f $filename ]; then
        if [ ! -f $2/$fn ]; then
            echo "$fn is missing from $2"
            missing=$((missing + 1))
        fi
    fi
done
echo "$missing files missing"
```

Ένας εύκολος τρόπος για να ελέγξουμε το πρόγραμμα είναι να χρησιμοποιήσουμε και στα δύο ορίσματα το ίδιο όνομα καταλόγου οπότε δεν θα παρουσιαστεί καμία απολύτως απόκλιση ανάμεσα στα περιεχόμενά τους, όπως απεικονίζεται στην επόμενη έξοδο.

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3ex4 /home/amarg /home/amarg
0 files missing
amarg@amarg-vbox:~/shared/Lab3$
```

Για μία πιο λεπτομερή επίδειξη, έστω ο κατάλογος Lab3 με τα επόμενα περιεχόμενα.

```
amarg@amarg-vbox:~/shared$ ls -l Lab3
total 7819
-rwxrwxrwx 1 root root     124 Οκτ  28 17:30 book.txt
-rwxrwxrwx 1 root root    1521 Σεπ  25 12:14 file1.doc
-rwxrwxrwx 1 root root      44 Σεπ  25 12:14 file1.txt
-rwxrwxrwx 1 root root     178 Σεπ  25 12:14 file1.zip
-rwxrwxrwx 1 root root 7959821 Σεπ  25 12:14 file2.doc
-rwxrwxrwx 1 root root      90 Σεπ  25 12:14 file2.txt
-rwxrwxrwx 1 root root     129 Σεπ  25 12:14 file2.zip
-rwxrwxrwx 1 root root    2797 Σεπ  25 12:14 file3.doc
-rwxrwxrwx 1 root root   13600 Σεπ  25 12:14 file3.zip
-rwxrwxrwx 1 root root     550 Σεπ  25 12:14 file4.doc
-rwxrwxrwx 1 root root     201 Σεπ  25 12:06 lab3ex1
-rwxrwxrwx 1 root root     214 Σεπ  25 12:07 lab3ex2
-rwxrwxrwx 1 root root      80 Σεπ  25 12:08 lab3ex3
-rwxrwxrwx 1 root root     707 Σεπ  22 09:14 lab3ex4
-rwxrwxrwx 1 root root    1147 Σεπ  25 12:11 lab3script1.txt
-rwxrwxrwx 1 root root     267 Σεπ  25 12:13 lab3script2.txt
-rwxrwxrwx 1 root root     370 Σεπ  25 12:15 lab3script3.txt
-rwxrwxrwx 1 root root     301 Σεπ  25 12:21 lab3script5.txt
-rwxrwxrwx 1 root root      87 Σεπ  25 12:14 months.txt
-rwxrwxrwx 1 root root       0 Οκτ  20 12:36 newf
```

Τον αντιγράφουμε μαζί με τα περιεχόμενά του στη θέση Lab3a και στη συνέχεια από τον κατάλογο Lab3a διαγράφουμε τα αρχεία lab3ex1, lab3ex2, lab3ex3 και lab3ex4.

```
amarg@amarg-vbox:~/shared$ cp -r Lab3 Lab3a
amarg@amarg-vbox:~/shared$ rm Lab3a/lab3ex1
amarg@amarg-vbox:~/shared$ rm Lab3a/lab3ex2
amarg@amarg-vbox:~/shared$ rm Lab3a/lab3ex3
amarg@amarg-vbox:~/shared$ rm Lab3a/lab3ex4
```

Εάν καλέσουμε τώρα το αρχείο κελύφους με ορίσματα τους καταλόγους Lab3 και Lab3a, το πρόγραμμα θα αναφέρει τα τέσσερα αρχεία που απουσιάζουν από το δεύτερο κατάλογο καθώς και το πλήθος τους.

```
amarg@amarg-vbox:~/shared$ Lab3/lab3ex4 Lab3 Lab3a
lab3ex1 is missing from Lab3a
lab3ex2 is missing from Lab3a
lab3ex3 is missing from Lab3a
lab3ex4 is missing from Lab3a
4 files missing
```

Μπορείτε να επαναλάβετε την παραπάνω διαδικασία με δικά σας αρχεία και καταλόγους.

Ασκήσεις

Άσκηση 1. Να γράψετε ένα σενάριο φλοιού (shell script) που : (αρχείο labscript1)

1) **Θα δέχεται ως όρισμα εισόδου το όνομα ενός καταλόγου. Το πρόγραμμα θα ελέγχει εάν ο κατάλογος υπάρχει και εάν όχι θα τον δημιουργεί.**
2) **Θα δημιουργεί δύο νέους καταλόγους μέσα στον αρχικό κατάλογο (απαιτείται έλεγχος για το αν υπάρχουν ήδη τα ονόματα των 2 καταλόγων).**
3) **Θα μετακινεί όλα τα αρχεία του καταλόγου στους δύο νέους καταλόγους ως εξής : Ο ένας κατάλογος θα περιλαμβάνει όλα τα αρχεία με όνομα που ξεκινά από τα γράμματα A-L (κεφαλαία και μικρά) ενώ ο άλλος κατάλογος τα υπόλοιπα αρχεία.**
4) **Θα εμφανίζει στην οθόνη το πλήθος των αρχείων σε καθένα από τους δύο νέους καταλόγους**

Για την επίδειξη της άσκησης χρησιμοποιήστε την εντολή touch για να κατασκευάσετε στον προσωπικό σας κατάλογο αρχεία με τα κατάλληλα ονόματα, π.χ. 5-6 αρχεία με όνομα που ξεκινά από τα γράμματα A-L (κεφαλαία και μικρά) και άλλα τόσα αρχεία με όνομα που να ξεκινά με τους υπόλοιπους χαρακτήρες έτσι ώστε να δείτε το αποτέλεσμα στην πράξη.

```
#!/bin/bash
```

Για να εκτελεστεί το πρόγραμμα θα πρέπει ο χρήστης να δώσει <u>ένα</u> όρισμα που να περιγράφει κατάλογο. Εάν λοιπόν η τιμή της μεταβλητής $# δεν είναι ίση με 1, το πρόγραμμα εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
if [ $# -ne 1 ]; then
  echo "Usage myScript dirName"
  exit 0
fi
```

Εάν ο κατάλογος δεν υπάρχει, το πρόγραμμα ρωτάει από το χρήστη εάν επιθυμεί να τον δημιουργήσει ή όχι και ανάλογα με την απάντησή του είτε δημιουργεί τον κατάλογο και συνεχίζει, είτε τερματίζει.

```
if [ ! -d "$1" ]; then
  echo -n "Directory does not exist. Do you want to create it? (y/n) --> "
  read answer
  if [ $answer = "N" -o $answer = "n" ]; then
    exit 0
   else mkdir $1
  fi
fi
```

Σε αυτό το σημείο ο χρήστης δίνει τα ονόματα των δύο υποκαταλόγων. Το πρόγραμμα ελέγχει εάν οι δύο υποκατάλογοι υπάρχουν και εάν αυτό δεν συμβαίνει τους δημιουργεί με την mkdir.

```
echo "Directory exists. Give the names of two subdirectories"
echo -n "Give the name of Subdir 1 --> "
read subdir1
echo -n "Give the name of Subdir 2 --> "
read subdir2
echo "Trying to create subdirectories in the directory $1..."
if [ -d "$1/$subdir1" ]; then
  echo "$subdir1 exists and there is no need to create it"
else mkdir $1/$subdir1
```

```
fi
if [ -d "$1/$subdir2" ]; then
  echo "$subdir2 exists and there is no need to create it"
else mkdir $1/$subdir2
fi
```

Η εντολή ls | grep '\<[A-La-l] επιστρέφει τα ονόματα των αρχείων και καταλόγων το όνομα των οποίων ξεκινά από A έως L ή από a έως l. Στην περίπτωση αυτή, η μεταβλητή i του βρόχου for παίρνει διαδοχικά ένα προς ένα αυτά τα ονόματα. Με άλλα λόγια σε κάθε κύκλο επανάληψης του βρόχου το $i περιέχει το όνομα του επόμενου αρχείου ή καταλόγου. Εάν το $i αναφέρεται σε αρχείο οπότε η συνθήκη if [-f $i] επιστρέφει true, το αρχείο αντιγράφεται με την εντολή cp στον πρώτο υποκατάλογο (cp $i $1/$subdir1) ενώ εάν αναφέρεται σε κατάλογο απλά αγνοείται (μας ενδιαφέρουν μόνο τα αρχεία).

```
for i in $(ls | grep '\<[A-La-l]')
do
  if [ -f $i ]; then
    echo "Copying $i to $1/$subdir1"
    cp $i $1/$subdir1
  fi
done
```

Εδώ γίνεται ακριβώς η ίδια διαδικασία με πριν, αλλά αυτή τη φορά ενδιαφέρομαστε για όλα τα υπόλοιπα αρχεία, οπότε στη συνθήκη της grep βάζουμε τον τελεστή ^ για να πάρουμε τα ονόματα που ΔΕΝ ξεκινούν από A έως L ή από a έως l. Αυτά τα αρχεία αντιγράφονται στο δεύτερο υποκατάλογο (cp $i $1/$subdir2).

```
for i in $(ls | grep '\<[^A-La-l]')
do
  if [ -f $i ]; then
    echo "Copying $i to $1/$subdir2"
    cp $i $1/$subdir2
  fi
done
```

Στο τέλος της διαδικασίας εμφανίζουμε το πλήθος των αρχείων που έχουν αντιγραφεί στους δύο υποκαταλόγους.

```
echo "$1/$subdir1 now has $(ls $1/$subdir1 | wc -w) file(s)"
echo "$1/$subdir2 now has $(ls $1/$subdir2 | wc -w) file(s)"
```

Η έξοδος της εφαρμογής ακολουθεί στη συνέχεια.

Εάν ο χρήστης δεν καλέσει το αρχείο με το σωστό πλήθος ορισμάτων εκτυπώνεται το κατάλληλο ενημερωτικό μήνυμα σύμφωνα με αυτά που αναφέραμε παραπάνω.

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script1
Usage myScript dirName
```

Εάν το αρχείο κληθεί σωστά, ρωτά το χρήστη εάν ο κατάλογος που όρισε θέλει να δημιουργηθεί (εάν δεν υπάρχει) και στη συνέχεια συμπεριφέρεται σύμφωνα με την εκφώνηση οδηγώντας σε έξοδο σαν και αυτή που παρουσιάζεται στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script1
Usage myScript dirName
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script1 myNewDir
Directory does not exist. Do you want to create it? (y/n) --> y
Directory exists. Give the names of two subdirectories
Give the name of Subdir 1 --> newSubdir1
Give the name of Subdir 2 --> newSubdir2
Trying to create subdirectories in the directory myNewDir...
Copying book.txt to myNewDir/newSubdir1
Copying file1.doc to myNewDir/newSubdir1
Copying file1.txt to myNewDir/newSubdir1
Copying file1.zip to myNewDir/newSubdir1
Copying file2.doc to myNewDir/newSubdir1
Copying file2.txt to myNewDir/newSubdir1
Copying file2.zip to myNewDir/newSubdir1
Copying file3.doc to myNewDir/newSubdir1
Copying file3.zip to myNewDir/newSubdir1
Copying file4.doc to myNewDir/newSubdir1
Copying lab3ex1 to myNewDir/newSubdir1
Copying lab3ex2 to myNewDir/newSubdir1
Copying lab3ex3 to myNewDir/newSubdir1
Copying lab3ex4 to myNewDir/newSubdir1
Copying lab3script1 to myNewDir/newSubdir1
Copying lab3script2 to myNewDir/newSubdir1
Copying lab3script3 to myNewDir/newSubdir1
Copying lab3script5 to myNewDir/newSubdir1
Copying book.txt to myNewDir/newSubdir2
Copying file1.txt to myNewDir/newSubdir2
Copying file1.zip to myNewDir/newSubdir2
Copying file2.txt to myNewDir/newSubdir2
Copying file2.zip to myNewDir/newSubdir2
Copying file3.zip to myNewDir/newSubdir2
Copying months.txt to myNewDir/newSubdir2
Copying newf to myNewDir/newSubdir2
myNewDir/newSubdir1 now has 18 file(s)
myNewDir/newSubdir2 now has 8 file(s)
```

Άσκηση 2. Να γράψετε ένα σενάριο φλοιού (shell script) το οποίο θα δέχεται ως ορίσματα εισόδου το όνομα ενός καταλόγου (η ύπαρξη του οποίου θα ελέγχεται) και μια επέκταση ονόματος αρχείων. Το σενάριο φλοιού θα ελέγχει το πλήθος των ορισμάτων εισόδου και στη συνέχεια θα βρίσκει και θα εμφανίζει όλα τα ονόματα των αρχείων του καταλόγου που έχουν την ίδια επέκταση με το δεύτερο όρισμα εισόδου. **(αρχείο labscript2)**

#!/bin/bash

Για να εκτελεστεί το πρόγραμμα θα πρέπει ο χρήστης να δώσει <u>δύο ορίσματα</u> που να ορίζουν ένα όνομα καταλόγου και μία επέκταση. Εάν λοιπόν η τιμή της μεταβλητής $# δεν είναι ίση με 2, το πρόγραμμα εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
if [ $# -ne 2 ]; then
  echo "Usage myScript dirName extension"
  exit 0
fi
```

Εάν το όρισμα που έδωσε ο χρήστης είναι ένας κατάλογος που δεν υπάρχει, το πρόγραμμα δεν μπορεί να συνεχίσει, οπότε εκτυπώνει ένα ενημερωτικό μήνυμα και ξανά τερματίζει τη λειτουργία του.

```
if [ ! -d "$1" ]; then
  echo "Directory $1 does not exist. Aborting ... "
  exit 0
fi
```

Εάν ο κώδικας φτάσει σε αυτό το σημείο και δεν έχει τερματιστεί προηγουμένως, αυτό σημαίνει πως ο χρήστης έχει δώσει τα σωστά ορίσματα και κατά συνέπεια πληρούνται οι προϋποθέσεις εκτέλεσης του κώδικα. Στην περίπτωση αυτή, για κάθε αρχείο που υπάρχει στον κατάλογο $1 (που έχει δοθεί ως πρώτο όρισμα) το πρόγραμμα ελέγχει εάν το όνομα του αρχείου (που είναι αποθηκευμένο στη μεταβλητή $file) τελειώνει σε $2 (δηλαδή στην επέκταση που έχει ορίσει ο χρήστης) και εάν ισχύει κάτι τέτοιο τότε το όνομα του αρχείου εκτυπώνεται στην οθόνη με την εντολή echo. Με τον τρόπο αυτό, με την ολοκλήρωση του προγράμματος, θα έχουν εκτυπωθεί στην οθόνη τα ονόματα των αρχείων που τελειώνουν με την κατάληξη που έχει ορίσει ο χρήστης.

```
for file in "$1"/*; do
  if [[ $file == *.$2 ]]; then
     echo $file
  fi
done
```

Ακολουθεί παράδειγμα εξόδου του προγράμματος. Στην παραπάνω έξοδο η τελεία αναφέρεται στον τρέχοντα κατάλογο. Για να ελέγξετε το πρόγραμμα μπορείτε να κατασκευάσετε όλα τα παρακάτω αρχεία με την touch.

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script2 . doc
./file1.doc
./file2.doc
./file3.doc
./file4.doc
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script2 . zip
./file1.zip
./file2.zip
./file3.zip
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script2 . txt
./book.txt
./file1.txt
./file2.txt
./months.txt
amarg@amarg-vbox:~/shared/Lab3$
```

Άσκηση 3. Να αναπτύξετε ένα πρόγραμμα σεναρίου κελύφους με όνομα isEmpty το οποίο θα δέχεται ως όρισμα ένα όνομα καταλόγου η ύπαρξη του οποίου θα ελέγχεται (επίσης θα ελέγχεται και το σωστό πλήθος ορισμάτων). Το πρόγραμμα θα ελέγχει εάν ο κατάλογος είναι κενός ή όχι εκτυπώνοντας το κατάλληλο μήνυμα ενώ εάν ο κατάλογος δεν είναι κενός, θα εκτυπώνονται τα περιεχόμενά του καθώς και το πλήθος τους. **(αρχείο labscript3)**

```
#!/bin/bash
```

Για να εκτελεστεί το πρόγραμμα θα πρέπει ο χρήστης να δώσει ένα όρισμα που να περιγράφει κατάλογο. Εάν λοιπόν η τιμή της μεταβλητής $# δεν είναι ίση με 1, το πρόγραμμα εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
if [ $# -ne 1 ]; then
  echo "Usage: isEmpty dirName"
```

```
  exit 0
fi
```

Εάν το όρισμα που έδωσε ο χρήστης είναι ένας κατάλογος που δεν υπάρχει, το πρόγραμμα δεν μπορεί να συνεχίσει, οπότε εκτυπώνει ένα ενημερωτικό μήνυμα και ξανά τερματίζει τη λειτουργία του.

```
if [ ! -d "$1" ]; then
  echo "Directory $1 does not exist. Aborting ... "
  exit 0
fi
```

Το πρόγραμμα αρχικοποιεί έναν μετρητή στη μηδενική τιμή. Στη συνέχεια καλεί την εντολή ls με όρισμα το όνομα του καταλόγου που έχει ορίσει ο χρήστης (που είναι αποθηκευμένο στη μεταβλητή $1) και αποστέλλει το αποτέλεσμά της στο for loop μέσα από το οποίο η μεταβλητή i παίρνει ένα προς ένα όλα τα ονόματα που επέστρεψε η ls. Για κάθε τέτοιο όνομα η μεταβλητή count αυξάνεται κατά μία μονάδα.

```
count=0
for i in $(ls $1); do
   echo $i
   count=$((count+1))
done
```

Εάν με την ολοκλήρωση της διαδικασίας η τιμή του count είναι ίση με το μηδέν, αυτό σημαίνει πως ο κατάλογος είναι κενός και εκτυπώνεται το κατάλληλο μήνυμα, ενώ στην αντίθετη περίπτωση ο κατάλογος δεν είναι κενός και το πρόγραμμα εκτυπώνει το πλήθος των περιεχομένων του.

```
if [ $count -ne 0 ]; then
   echo "Directory $1 is not empty and contains $count items"
else
   echo "Directory $1 is empty"
fi
```

Παράδειγμα εξόδου ακολουθεί στη συνέχεια αρχικά με έναν κενό κατάλογο και στη συνέχεια με ένα κατάλογο που περιέχει αρχεία και καταλόγους.

```
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script3 testDir
Directory testDir is empty
amarg@amarg-vbox:~/shared/Lab3$ ./lab3script3 myNewDir
newSubdir1
newSubdir2
Directory myNewDir is not empty and contains 2 items
amarg@amarg-vbox:~/shared/Lab3$
```

Άσκηση 4 (Προς επίλυση). Να αναπτύξετε ένα πρόγραμμα σεναρίου κελύφους με όνομα mycopy το οποίο χρησιμοποιεί δύο παραμέτρους που αντιστοιχούν σε δύο ονόματα αρχείων. Το πρόγραμμα να αντιγράφει το πρώτο αρχείο στο δεύτερο, λειτουργώντας με τον ακόλουθο τρόπο:

1. Να ελέγχει αν τα ορίσματα είναι δύο. Αν δεν είναι, να τερματίζει εμφανίζοντας το κατάλληλο ενημερωτικό μήνυμα.
2. Να ελέγχει αν το αρχείο πηγή υπάρχει. Αν δεν υπάρχει να τερματίζει εμφανίζοντας το κατάλληλο ενημερωτικό μήνυμα.

3. Να ελέγχει αν υπάρχει το αρχείο - στόχος. Αν δεν υπάρχει να κάνει την αντιγραφή. Αν υπάρχει να ρωτάει το χρήστη εάν επιθυμεί να συνεχίσει την αντιγραφή πάνω στο παλιό αρχείο, να διαβάζει την απάντησή μας και να πράττει ανάλογα.
4. Να ελέγχει την περίπτωση το δεύτερο όρισμα να μην είναι αρχείο αλλά κατάλογος και να λειτουργεί όπως και πριν.

Άσκηση 5. Να γράψετε ένα σενάριο φλοιού (shell script) το οποίο θα δέχεται ως όρισμα εισόδου το όνομα ενός καταλόγου. Το πρόγραμμα θα ελέγχει εάν καλείται με το σωστό πλήθος ορισμάτων και εάν ο κατάλογος που έχει οριστεί υπάρχει και εάν πληρούνται αυτές οι δύο προϋποθέσεις θα υπολογίζει και θα εκτυπώνει το συνολικό μέγεθος των αρχείων που υπάρχουν σε αυτόν τον κατάλογο. **(αρχείο labscript5)**

```
#!/bin/bash
```

Για να εκτελεστεί το πρόγραμμα θα πρέπει ο χρήστης να δώσει ένα όρισμα που να περιγράφει κατάλογο. Εάν λοιπόν η τιμή της μεταβλητής $# δεν είναι ίση με 1, το πρόγραμμα εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία του.

```
if [ $# -ne 1 ]; then
  echo "Usage myScript dirName"
  exit 0
fi
```

Εάν το όρισμα που έδωσε ο χρήστης είναι ένας κατάλογος που δεν υπάρχει, το πρόγραμμα δεν μπορεί να συνεχίσει, οπότε εκτυπώνει ένα ενημερωτικό μήνυμα και ξανά τερματίζει τη λειτουργία του.

```
if [ ! -d "$1" ]; then
  echo "Directory $1 does not exist. Aborting ... "
  exit 0
fi
```

Η στήλη με τις τιμές των μεγεθών των αρχείων του καταλόγου που καταχώρησε ο χρήστης (η πέμπτη στήλη στην έξοδο της ls –l $1) και βρίσκεται στη μεταβλητή $1 υπολογίζεται από το pipeline ls -l $`| grep '^-' | awk '{ print $5}' **(μετά από τόσα παραδείγματα που έγιναν ελπίζω να είναι κατανοητό τι ακριβώς γίνεται εδώ). Αυτά τα μεγέθη μέσα από το for loop στο οποίο αποστέλλεται το αποτέλεσμα της παραπάνω διαδικασίας εκχωρούνται ένα προς ένα στη μεταβλητή i και προστίθενται μεταξύ τους με το γνωστό τρόπο. Στο τέλος το πρόγραμμα εκτυπώνει το ζητούμενο αποτέλεσμα.**

```
size=0
for i in $(ls -l $`| grep '^-' | awk '{ print $5}')
do
  size=$((size+$i))
done
echo "Total size of files in $1 is $size"
```

Παράδειγμα της εξόδου ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./lab3script5 shared
Total size of files in shared is 1298499
amarg@amarg-vbox:~$
```

ΕΡΓΑΣΤΗΡΙΟ 4

Διαχείριση διεργασιών

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 εμφανίσετε το δέντρο των διεργασιών του συστήματος εκτελώντας στη γραμμή εντολών την εντολή pstree.

3. 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.

4. Να εκτελέσετε την εντολή ls –l με αυξημένη τιμή προτεραιότητας κατά 12 και την εντολή pwd με αυξημένη τιμή προτεραιότητας κατά 5.

   Η προεπιλεγμένη τιμή προτεραιότητας στο λειτουργικό σύστημα Linux είναι η τιμή 0 όπως φαίνεται αν εκτελέσουμε την εντολή nice χωρίς ορίσματα.

```
amarg@amarg-vbox:~/Desktop$ nice
0
amarg@amarg-vbox:~/Desktop$
```

   Στη γενική περίπτωση η τιμή της προτεραιότητας που ορίζεται με την nice έχει τιμές από -20 έως 19 με τη μέγιστη προτεραιότητα να είναι η -20 και η ελάχιστη προτεραιότητα να είναι η 19.

   Όσον αφορά στην επίλυση της άσκησης, οι εντολές που δίδονται εάν εκτελεστούν ως έχουν θα εκτελεστούν με την προεπιλεγμένη τιμή προτεραιότητας που είναι η τιμή 0. Για να εκτελέσουμε την εντολή ls –l με αυξημένη τιμή προτεραιότητας κατά 12 θα πρέπει να γράψουμε

```
amarg@amarg-vbox:~$ nice -n 12 ls -l
total 97440
-rwxrwxr-x  1 amarg amarg    16872 Σεπ  29 12:13 acc1
-rw-rw-r--  1 amarg amarg     2264 Σεπ  29 12:13 acc1.o
-rwxrwxr-x  1 amarg amarg    17064 Οκτ  18 11:41 addrInfo
-rwxrwxrwx  1 amarg amarg     1064 Οκτ  18 11:44 addrInfo.c
-rw-rw-r--  1 amarg amarg     2704 Οκτ  18 11:40 addrInfo.o
-rw-rw-r--  1 amarg amarg      386 Οκτ  20 11:53 allusers
-rw-rw-r--  1 amarg amarg      167 Σεπ  22 10:27 awkscript
drwxrwxr-x  3 amarg amarg     4096 Οκτ  15 14:36 bin
-rw-rw-r--  1 amarg amarg      124 Οκτ  28 17:30 book.txt
```

   ενώ για να εκτελέσουμε την εντολή pwd με αυξημένη τιμή προτεραιότητας κατά 5 θα πρέπει να γράψουμε

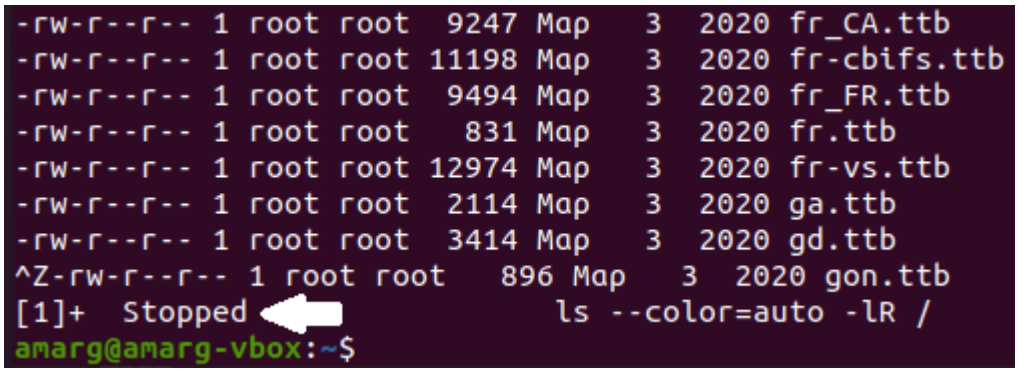
```
amarg@amarg-vbox:~$ nice -n 5 pwd
/home/amarg
amarg@amarg-vbox:~$
```

   Σημειώστε πως η προτεραιότητα **NI** που ορίζεται με τη nice σχετίζεται με το χώρο του χρήστη και **ΔΕΝ** είναι η πραγματική προτεραιότητα **PR** της διεργασίας η οποία σχετίζεται με το χώρο του πυρήνα. Οι δύο αυτές τιμές προτεραιότητα σχετίζονται με την έκφραση **PR = 20 + NI** και για τιμή PR = 0 η τιμή του NI είναι ίση με -20. Δυστυχώς είναι αδύνατο να δοθούν εδώ περισσότερες πληροφορίες και ευελπιστώ όλα αυτά να τα δείτε αναλυτικά στο μάθημα των λειτουργικών συστημάτων.

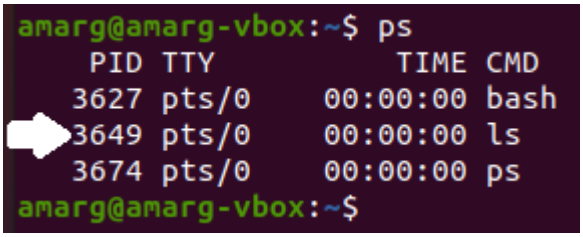
5. (α) Να εκτελέσετε την εντολή ls –lR / (που εμφανίζει την απόλυτη διαδρομή όλων των αρχείων του συστήματος) από 2-3 δευτερόλεπτα λειτουργίας σταματήστε τη πατώντας το συνδυασμό πλήκτρων Ctrl-Z. (β) Εκτελέστε την εντολή ps και αναζητήστε την εντολή στη λίστα των ενεργών διεργασιών που εκτυπώνεται. (γ) Καταγράψτε το pid της. (δ) Τερματίστε την εντολή γράφοντας kill -9 pid (για να δείτε το ρόλο της επιλογής -9 εμφανίστε τη σελίδα βοηθείας της εντολής γράφοντας man kill). (ε) επαληθεύστε τον τερματισμό της εντολής εκτελώντας ξανά την εντολή ps. (στ) Να επαναλάβετε το βήμα (α) αλλά αντί για το βήμα (β) να εκτελέσετε την εντολή fg για να επαναφέρετε τη διεργασία στο προσκήνιο. Στη συνέχεια σταματήστε τη εκ νέου με Ctrl-Z και αυτή τη φορά

χρησιμοποιήστε την εντολή bg για να εκτελέσετε τη διεργασία στο παρασκήνιο. Είναι δυνατή τώρα η αναστολή της εκτέλεση της εντολής και αν όχι, γιατί?

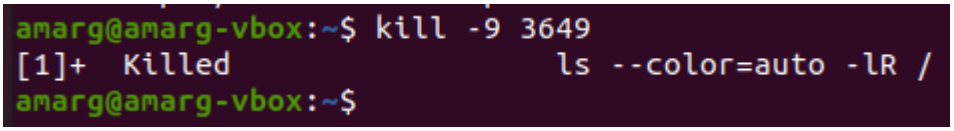
Αρχικά εκτελούμε την εντολή ls –l R και μετά από λίγο πατάμε Ctrl-Z αναστέλλοντας τη λειτουργία της.



Εκτελώντας την εντολή ps διαπιστώνουμε πως ο κωδικός της διεργασίας (process id, PID) που δημιουργήθηκε από την εκτέλεση της εντολής είναι 3649.



Στη συνέχεια τερματίζουμε τη διεργασία καλώντας την εντολή kill -9 PID όπου PID = 3649 ο κωδικός διεργασίας.



και επαληθεύουμε πως η διεργασία έχει τερματιστεί εκτελώντας ξανά την ps, η οποία πλέον ΔΕΝ την εμφανίζει.

Ας εκτελέσουμε ξανά την εντολή και ας τη σταματήσουμε και πάλι με το συνδυασμό Ctrl-Z. Εάν τώρα στη γραμμή εντολών πληκτρολογήσουμε fg (foreground) και πατήσουμε το Enter προκαλούμε τη συνέχιση της λειτουργίας της σταματημένης εντολής στο προσκήνιο, ενώ εάν αντίθετα πληκτρολογήσουμε στη γραμμή εντολών bg (background) και πατήσουμε το Enter, προκαλούμε τη συνέχιση της λειτουργίας της σταματημένης εντολής στο παρασκήνιο. Η διαφορά ανάμεσα στο προσκήνιο και στο παρασκήνιο είναι η εξής: Όταν μία διεργασία εκτελείται στο προσκήνιο δεσμεύει το τερματικό μας αφού η έξοδός της εκτυπώνεται στην οθόνη και κατά συνέπεια για να μπορέσουμε να χρησιμοποιήσουμε ξανά το τερματικό θα πρέπει η διεργασία να ολοκληρώσει τη λειτουργία της. Ωστόσο, δεν χάνουμε την πρόσβαση στο πληκτρολόγιο και κατά συνέπεια μπορούμε να αλληλοεπιδράσουμε με μία διεργασία η οποία εκτελείται στο προσκήνιο (για παράδειγμα, να αναστείλουμε ή να τερματίσουμε τη λειτουργία της με τους συνδυασμούς πλήκτρων Ctrl-Z και Ctrl-C). Από την άλλη πλευρά, όταν μία διεργασία εκτελείται στο παρασκήνιο δεν συνδέεται με το πληκτρολόγιο και ως εκ τούτου δεν μπορούμε να αλληλοεπιδράσουμε μαζί της. Για παράδειγμα, εάν επαναφέρουμε τη σταματημένη ls –lR / στο παρασκήνιο εκτελώντας την εντολή bg θα διαπιστώσουμε πως τώρα οι συνδυασμοί πλήκτρων Ctrl-C και Ctrl-Z δεν λειτουργούν διότι έχει χαθεί η σύνδεση με το πληκτρολόγιο και κατά συνέπεια θα πρέπει αναγκαστικά να περιμένουμε να ολοκληρωθεί. Εναλλακτικά μπορούμε να ανοίξουμε ένα άλλο τερματικό, να ανακτήσουμε το PID της διεργασίας με την εντολή ps και να τερματίσουμε τη λειτουργία της με την εντολή kill όπως κάναμε προηγουμένως.

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.

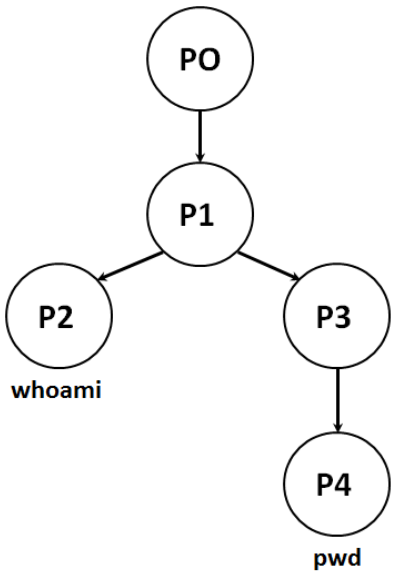
```
       printf ("Child1 with pid %d kai ppid %d \n", getpid(), getppid());
       pid2 = fork();
       if (pid2>0)
        {
          printf ("Parent process (child1) for child2\n");
          child2 = wait(&status2);
          printf ("Child with code %d has been terminated\n", child2); }
      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$
```

Άσκηση. Συνδυάζοντας τα Παραδείγματα 4 και 5 να κατασκευάσετε την τοπολογία διεργασιών του επόμενου σχήματος. Η κάθε διεργασία θα εκτυπώνει τις τιμές των παραμέτρων pid και ppid και στη συνέχεια θα καλεί την exec για να εκτελέσει την εντολή που τη συνοδεύει στο επόμενο σχήμα (για όσες έχουν συνοδευτική εντολή). Η κάθε γονική διεργασία θα αναμένει την ολοκλήρωση των θυγατρικών διεργασιών της και μετά θα τερματίζει και η ίδια.



ΕΡΓΑΣΤΗΡΙΟ 5

Αρχεία και κατάλογοι

Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τη χρήση των συναρτήσεων διαχείρισης αρχείων και καταλόγων.

Παράδειγμα 1. Χρήση της stat για την εμφάνιση των πληροφοριών του i-node και για αρχείο που διαβιβάζεται ως όρισμα στη γραμμή εντολών (αρχείο filestat.c).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  struct stat sb;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **filestat 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) {
     fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
     exit(EXIT_FAILURE); }
```

Η λήψη των ζητούμενων πληροφοριών γίνεται με την κλήση της συνάρτησης stat η οποία εάν κληθεί με επιτυχία επιστρέφει στη δομή **struct stat sb** τις τιμές των ιδιοτήτων του αρχείου. Εάν η επιστρεφόμενη τιμή είναι -1, έχει ανακύψει κάποιο σφάλμα και η συνάρτηση τερματίζει τη λειτουργία της.

```
  if (stat(argv[1], &sb) == -1) {
     perror("stat");
     exit(EXIT_FAILURE); }
```

Αρχικά εξετάζουμε την τιμή του πεδίου **st_mode** η οποία ορίζει τον τύπο του αρχείου με τον ακόλουθο τρόπο: εάν είναι ίση με **S_IFBKL** το αρχείο είναι αρχείο παράλληλης συσκευής (block device file), εάν είναι ίση με **S_IFCHR** το αρχείο είναι αρχείο σειριακής συσκευής (character device file), εάν είναι **S_IFDIR** το αρχείο κατάλογος (directory), εάν είναι **S_IFIFO** το αρχείο είναι αρχείο επώνυμου αγωγού (FIFO/pipe), εάν είναι **S_IFLNK** το αρχείο είναι συμβολικός σύνδεσμος (symbolic link), εάν είναι **S_IFREG** το αρχείο είναι σύνηθες αρχείο (regular file), ενώ εάν είναι **S_IFSOCK** το αρχείο είναι αρχείο δικτυακού υποδοχέα (socket). Ο έλεγχος της τιμής του πεδίου st_mode και η εκτύπωση του κατάλληλου μηνύματος, γίνεται με την case / switch.

```
printf("File type:                ");
  switch (sb.st_mode & S_IFMT) {
   case S_IFBLK:  printf("block device\n");          break;
   case S_IFCHR:  printf("character device\n");      break;
   case S_IFDIR:  printf("directory\n");             break;
   case S_IFIFO:  printf("FIFO/pipe\n");             break;
   case S_IFLNK:  printf("symlink\n");               break;
   case S_IFREG:  printf("regular file\n");          break;
   case S_IFSOCK: printf("socket\n");                break;
   default:       printf("unknown?\n");              break; }
```

Στη συνέχεια εκτυπώνουμε τις τιμές των υπόλοιπων πεδίων της δομής που περιλαμβάνουν: την τιμή του i-node (το πεδίο **st_ino**), την τιμή της κατάστασης (το πεδίο **st_mode**) στην οποία τα δώδεκα τελευταία bits περιέχουν τη μάσκα δικαιωμάτων

ενώ τα υπόλοιπα τον τύπο του αρχείου (σύνηθες αρχείο ή κατάλογος) και τον τροποποιητή (τα τρία ειδικά bits), το πλήθος των συνδέσμων προς το αρχείο (το πεδίο **st_link**), το UID και το GID του αρχείου (δηλαδή τον κωδικό του κατόχου και της ομάδας του κατόχου του αρχείου) (τα πεδία **st_uid** και **st_gid**), το (προτιμώμενο) μέγεθος του block δεδομένων που μεταφέρεται από ή προς το δίσκο σε ένα βήμα κατά την πραγματοποίηση μιας διαδικασίας ανάγνωσης ή εγγραφής (το πεδίο **st_blksize**), το μέγεθος του αρχείου (**st_size**), το πλήθος των blocks στο δίσκο που δεσμεύονται από το αρχείο (το πεδίο **st_blocks**) , τη χρονική στιγμή της τελευταίας μεταβολής της κατάστασης του αρχείου (το πεδίο **st_ctime**), τη χρονική στιγμή της τελευταίας προσπέλασης του αρχείου (το πεδίο **st_atime**) και τη χρονική στιγμή της τελευταίας τροποποίησης του αρχείου (το πεδίο **st_mtime**).

```
printf("I-node number:          %ld\n", (long) sb.st_ino);
  printf("Mode:                 %lo (octal)\n", (unsigned long) sb.st_mode);
  printf("Link count:           %ld\n", (long) sb.st_nlink);
  printf("Ownership:            UID=%ld   GID=%ld\n", (long) sb.st_uid, (long) sb.st_gid);
  printf("Preferred I/O block size: %ld bytes\n", (long) sb.st_blksize);
  printf("File size:            %lld bytes\n", (long long) sb.st_size);
  printf("Blocks allocated:       %lld\n", (long long) sb.st_blocks);
  printf("Last status change:     %s", ctime(&sb.st_ctime));
  printf("Last file access:       %s", ctime(&sb.st_atime));
  printf("Last file modification:   %s", ctime(&sb.st_mtime));
  exit(EXIT_SUCCESS); }
```

Ένα παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια, για ένα σύνηθες αρχείο (πρώτη εικόνα) και για έναν κατάλογο (δεύτερη εικόνα). Παρατηρήστε τη διαφορά στην πρώτη γραμμή (File type).

```
amarg@amarg-vbox:~/shared/Lab5$ ./filestat ~/sample.c
File type:                regular file
I-node number:            279500
Mode:                     100664 (octal)
Link count:               1
Ownership:                UID=1000   GID=1000
Preferred I/O block size: 4096 bytes
File size:                1238 bytes
Blocks allocated:         8
Last status change:       Thu Oct  8 12:09:40 2020
Last file access:         Tue Nov  3 12:21:45 2020
Last file modification:   Thu Oct  8 12:09:40 2020
amarg@amarg-vbox:~/shared/Lab5$
```

```
amarg@amarg-vbox:~/shared/Lab5$ ./filestat /bin
File type:                directory
I-node number:            917523
Mode:                     40755 (octal)
Link count:               2
Ownership:                UID=0   GID=0
Preferred I/O block size: 4096 bytes
File size:                40960 bytes
Blocks allocated:         80
Last status change:       Wed Nov  4 11:05:01 2020
Last file access:         Wed Nov  4 11:05:09 2020
Last file modification:   Wed Nov  4 11:05:01 2020
```

Παράδειγμα 2. Έλεγχος της ύπαρξης και των δικαιωμάτων πρόσβασης του αρχείου με τη χρήση της access (αρχείο testAccess.c).

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[]) {
 char* path = argv[1];
 int rval;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **testAccess 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 **(αυτός ο έλεγχος ΔΕΝ ΓΙΝΕΤΑΙ στο αρχείο του eClass, προσθέστε το μόνοι σας)**.

```
  if (argc != 2) {
     fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
     exit(EXIT_FAILURE); }
```

Καταρχήν θα πρέπει να ελέγξουμε εάν το αρχείο υπάρχει, αλλιώς δεν έχει νόημα να συνεχίσουμε. Αυτό γίνεται καλώντας τη συνάρτηση access με το όρισμα **F_OK**. Εάν η access επιστρέψει την τιμή 0 αυτό σημαίνει πως το αρχείο υπάρχει και μπορούμε να συνεχίσουμε, ενώ στην αντίθετη περίπτωση εξετάζουμε την τιμή της μεταβλητής errno. Εάν αυτή τιμή είναι ίση με **ENOENT** αυτό σημαίνει πως το αρχείο δεν υπάρχει, ενώ εάν είναι ίση με **EACCESS**, αυτό σημαίνει πως το αρχείο υπάρχει αλλά δεν είναι προσβάσιμο –σε κάθε περίπτωση η λειτουργία της εφαρμογής τερματίζεται.

```
 rval = access (path, F_OK);
 if (rval == 0)
    printf ("%s exists\n", path);
 else {
    if (errno == ENOENT)
        printf ("%s does not exist\n", path);
    else if (errno == EACCES)
         printf ("%s is not accessible\n", path);
  return 0;  }
```

Σε αυτό το σημείο εξετάζουμε εάν έχουμε δικαίωμα ανάγνωσης κάτι που γίνεται καλώντας τη συνάρτηση access με το όρισμα **R_OK**. Εάν η access επιστρέψει την τιμή 0 αυτό σημαίνει πως έχουμε δικαίωμα ανάγνωσης του αρχείου, κάτι που δε ισχύει στην αντίθετη περίπτωση. Η εφαρμογή εκτυπώνει το κατάλληλο μήνυμα.

```
 rval = access (path, R_OK);
 if (rval == 0) printf ("%s is readable\n", path);
 else printf ("%s is not readable (access denied)\n", path);
```

Σε αυτό το σημείο εξετάζουμε εάν έχουμε δικαίωμα εγγραφής κάτι που γίνεται καλώντας τη συνάρτηση access με το όρισμα **W_OK**. Εάν η access επιστρέψει την τιμή 0 αυτό σημαίνει πως έχουμε δικαίωμα εγγραφής του αρχείου, κάτι που δε ισχύει στην αντίθετη περίπτωση, όπου εξετάζουμε τι ακριβώς συμβαίνει (είτε το συγκεκριμένο αρχείο δεν είναι εγγράψιμο (**EACCESS**) είτε όλο το σύστημα αρχείων είναι μόνο για ανάγνωση (**EROFS**)). Η εφαρμογή εκτυπώνει το κατάλληλο μήνυμα.

```
 rval = access (path, W_OK);
 if (rval == 0) printf ("%s is writable\n", path);
 else if (errno == EACCES) printf ("%s is not writable (access denied)\n", path);
 else if (errno == EROFS) printf ("%s is not writable (read-only filesystem)\n", path);
 return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Για την επίδειξη της εφαρμογής δημιουργούμε τέσσερα αρχεία στα οποία ορίζουμε όλους τους δυνατούς συνδυασμούς δικαιωμάτων ανάγνωσης και εγγραφής για τον κάτοχο (δηλαδή - -, - w, r - , r w) και στη συνέχεια καλούμε την εφαρμογή acc1 για να ανακτήσουμε τα παραπάνω δικαιώματα πρόσβασης, με τα αποτελέσματα φυσικά να βρίσκονται σε πλήρη συμφωνία με αυτά που προκύπτουν από τη μελέτη της μάσκας δικαιωμάτων.

```
amarg@amarg-vbox:~$ chmod 144 file1.doc
amarg@amarg-vbox:~$ chmod 344 file2.doc
amarg@amarg-vbox:~$ chmod 544 file3.doc
amarg@amarg-vbox:~$ chmod 744 file4.doc
amarg@amarg-vbox:~$ ls -l *.doc
- -- xr--r-- 1 amarg amarg    1521 Σεπ  20 19:58 file1.doc
- -w xr--r-- 1 amarg amarg 7959821 Σεπ  20 19:58 file2.doc
- r- xr--r-- 1 amarg amarg    2797 Σεπ  20 19:59 file3.doc
- rw xr--r-- 1 amarg amarg     550 Σεπ  20 19:59 file4.doc
amarg@amarg-vbox:~$ ./acc1 file1.doc
file1.doc exists
file1.doc is not readable (access denied)
file1.doc is not writable (access denied)
amarg@amarg-vbox:~$ ./acc1 file2.doc
file2.doc exists
file2.doc is not readable (access denied)
file2.doc is writable
amarg@amarg-vbox:~$ ./acc1 file3.doc
file3.doc exists
file3.doc is readable
file3.doc is not writable (access denied)
amarg@amarg-vbox:~$ ./acc1 file4.doc
file4.doc exists
file4.doc is readable
file4.doc is writable
amarg@amarg-vbox:~$
```

Παράδειγμα 3. Αλλαγή δικαιωμάτων πρόσβασης του αρχείου με την εντολή chmod (αρχείο testChmod.c).

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

int main (int argc, char * argv[]) {
  int fd;
  struct stat info;
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **testChmod 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 testChmod pathname\n");
     return (-1); }
```

Προκειμένου να γίνει κατανοητός ο τρόπος λειτουργίας της εφαρμογής, το πρόγραμμα ανακτά την παλαιά μάσκα δικαιωμάτων έτσι ώστε αυτή να εκτυπωθεί μαζί με τη νέα μάσκα δικαιωμάτων δίδοντας με τον τρόπο αυτό την ευκαιρία στο χρήστη να εντοπίσει τις αλλαγές που πραγματοποιήθηκαν. Αρχικά το πρόγραμμα καλεί τη συνάρτηση **access** που χρησιμοποιήσαμε και προηγουμένως για να ελέγξει εάν το αρχείο ή ο κατάλογος υπάρχει. Εάν η συνάρτηση επιστρέψει την τιμή -1, το εξεταζόμενο αντικείμενο δεν υπάρχει και η εφαρμογή εκτυπώνει το κατάλληλο μήνυμα σφάλματος και τερματίζει.

```
fd = access(argv[1], F_OK);
If (fd == -1){

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

Στην αντίθετη περίπτωση, κατά την οποία το αντικείμενο που όρισε ο χρήστης είναι ένα υπάρχον αντικείμενο, το πρόγραμμα καλεί τη συνάρτηση **stat** που είδαμε στην πρώτη άσκηση, η οποία επιστρέφει στη δομή info τα περιεχόμενα του i-node για το εν λόγω αντικείμενο (αρχείο ή κατάλογος). Η μάσκα δικαιωμάτων περιλαμβάνεται στο πεδίο **st_mode**, η τιμή του οποίου εκτυπώνεται στο **οκταδικό** σύστημα με **8** ψηφία όπως υπαγορεύεται από τον προσδιοριστή **%08o** που χρησιμοποιούμε στην printf (το %o (ο από τη λέξη octal → οκταδικό) εκτυπώνει την τιμή στο οκταδικό σύστημα ενώ το 08 από μπροστά προκαλεί την εκτύπωση με 8 ψηφία).

```
else {
   stat(argv[1], &info);
   printf("Original file permissions were: %08o\n", info.st_mode);
```

Τέλος, καλούμε τη συνάρτηση **chmod** για την αλλαγή της μάσκας δικαιωμάτων. Η τιμή S_IRWXU εκχωρεί στον κάτοχο του αρχείου (USER, S_IRWX**U**) δικαιώματα ανάγνωσης, εγγραφής και εκτέλεσης (S_I**RWX**U), ενώ η τιμή S_IRWXG εκχωρεί στην ομάδα του κατόχου του αρχείου (GROUP, S_IRWX**G**) δικαιώματα ανάγνωσης, εγγραφής και εκτέλεσης (S_I**RWX**G). Ο συνδυασμός αυτών των δύο τελεστών με τη δυαδική λογική διάζευξη (**|**), εκχωρεί στο αντικείμενο δικαιώματα ανάγνωσης, εγγραφής και εκτέλεσης, τόσο για τον κάτοχο όσο και για την ομάδα του κατόχου. Εάν η συνάρτηση chmod επιστρέψει τιμή διαφορετική του μηδενός αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα το οποίο εκτυπώνεται από τη συνάρτηση perror, ενώ στην αντίθετη περίπτωση καλείται ξανά η stat με τον τρόπο με τον οποίο αυτό έγινε προηγουμένως, προκειμένου να ανακτήσει και να εκτυπώσει τη νέα τιμή της μάσκας δικαιωμάτων του αρχείου.

```
   if (chmod(argv[1], S_IRWXU|S_IRWXG) != 0) perror("chmod() error");
   else {
     stat(argv[1], &info);
     printf("After chmod(), file permissions are: %08o\n", info.st_mode); }}
 return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ls -l | grep results
-rw-rw-r--  1 amarg amarg     262 Σεπ  22 10:08 results
amarg@amarg-vbox:~$ ./testChmod results
Original file permissions were: 00100664
After chmod(), file permissions are: 00100770
amarg@amarg-vbox:~$ ls -l | grep results
-rwxrwx---  1 amarg amarg     262 Σεπ  22 10:08 results
```

Η αρχική μάσκα δικαιωμάτων είναι η **00100664** που διαχωρίζεται ως **0010 0 664**. Το **0010** σημαίνει πως το αντικείμενο είναι κανονικό αρχείο, η τιμή **0** του τροποποιητή σημαίνει πως **δεν έχουν τεθεί** τα ειδικά bits (setuid, setgid και sticky bit) ενώ η μάσκα **664** που στο δυαδικό είναι η 110 110 100 ορίζει τα δικαιώματα **r w – r w – r – –** . Μετά την εκτέλεση της εντολής η νέα μάσκα είναι η **00100770**. Τα bits **0010** και **0** είναι τα ίδια με πριν ενώ τα νέα δικαιώματα πρόσβασης είναι τα **770** δηλαδή **111 111 000** ή **r w x r w x – – –**. Παρατηρήστε πως πλέον **ο κάτοχος και η ομάδα έχουν δικαίωμα ανάγνωσης, εγγραφής και εκτέλεσης** όπως άλλωστε αναμέναμε και πως η αλλαγή που έγινε άλλαξε **ολόκληρη** τη μάσκα. Πράγματι, ενώ στην αρχή οι άλλοι χρήστε είχαν δικαίωμα ανάγνωσης, τώρα δεν έχουν **τίποτε**, αφού η συνάρτηση chmod **δεν όρισε** δικαιώματα για τους άλλους χρήστες και ως εκ τούτου αυτοί έχασαν και αυτά που είχαν προηγουμένως.

Παράδειγμα 4. Αλλαγή κατόχου και ομάδας κατόχου αρχείου με την εντολή chown (αρχείο testChown.c).

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

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

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **testChmod 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 testChown pathname\n");
      return (-1); }
```

Εάν η εφαρμογή κλήθηκε με το σωστό τρόπο, καλεί τη συνάρτηση **chown** για την αλλαγή του κατόχου και της ομάδας του κατόχου για το αρχείο ή τον κατάλογο που όρισε ο χρήστης (και το οποίο βρίσκεται καταχωρημένο στη συμβολοσειρά argv[1]). Σε αντίθεση με τη γραμμή εντολών του λειτουργικού όπου υπάρχουν **δύο** εντολές, η **chown** για την αλλαγή του κατόχου και η **chgrp** για την αλλαγή της ομάδας του κατόχου, η γλώσσα C διαθέτει **μία και μοναδική συνάρτηση** chown που μεταβάλλει και τις δύο αυτές παραμέτρους (δείτε τη διαφάνεια 15 της παρουσίασης 06.Files and Directories). Στο παράδειγμα της άσκησης ως κάτοχο και ομάδα κατόχου του αρχείου ορίζουμε τον χρήστη **SYS** και την ομάδα **SYS** οι κωδικοί των οποίων έχουν αμφότεροι την τιμή 3, όπως προκύπτει από τα περιεχόμενα των αρχείων **/etc/passwd** (που περιέχει τα στοιχεία των χρηστών) και **/etc/group** (που περιέχει τα στοιχεία των ομάδων χρηστών).

```
amarg@amarg-vbox:~/shared/Lab4$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

```
amarg@amarg-vbox:~/shared/Lab4$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,amarg
```

```
  retVal = chown (argv[1], (uid_t)3, (gid_t)3);
```

Εάν η συνάρτηση επιστρέψει την τιμή -1, σημαίνει πως έλαβε χώρα κάποιο σφάλμα. Στην περίπτωση αυτή η συνάρτηση καλεί την **perror** για να εκτυπώσει μία λεκτική περιγραφή του σφάλματος που προέκυψε και τερματίζει τη λειτουργία της ενώ στην αντίθετη περίπτωση εκτυπώνει ένα ενημερωτικό μήνυμα που ενημερώνει το χρήστη πως η εφαρμογή λειτούργησε σωστά και πραγματοποίησε τη μεταβολή του κατόχου και της ομάδας του κατόχου του αρχείου.

```
  if(retVal == -1){
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-2); }
```

```
  printf ("Assignment to %s of user SYS and group SYS
                                    has been performed succesfully\n", argv[1]);
  return (0); }
```

Παράδειγμα χρήσης της εφαρμογή ακολουθεί στη συνέχεια. Παρατηρήστε πως εάν η εφαρμογή κληθεί με το συνήθη τρόπο **δεν λειτουργεί** αλλά εκτυπώνει το μήνυμα **Operation not permitted** (μη επιτρεπτή λειτουργία) – ο λόγος για αυτό είναι πως οι εντολές αυτές με τον τρόπο που καλούνται μπορούν να εκτελεστούν μόνο από το διαχειριστή του συστήματος.

```
amarg@amarg-vbox:~/shared/Lab5$ ./testChown ~/sample.c
Error Number : 1
Error Description: Operation not permitted
amarg@amarg-vbox:~/shared/Lab5$
```

Η εκτέλεση της εφαρμογής ως διαχειριστής πραγματοποιείται με την εντολή **sudo** η οποία απαιτεί την καταχώρηση του κωδικού του τρέχοντα χρήστη. Στην περίπτωση αυτή, η εφαρμογή εκτελείται χωρίς πρόβλημα.

```
amarg@amarg-vbox:~/shared/Lab5$ sudo ./testChown ~/sample.c
[sudo] password for amarg:
Assignment to /home/amarg/sample.c of user SYS and group SYS has been performed succesfully
amarg@amarg-vbox:~/shared/Lab5$
```

Η εκτέλεση της εντολής ls –l πιστοποιεί πως η εφαρμογή εκτελέστηκε με επιτυχία, αφού πλέον ο κάτοχος και η ομάδα του κατόχου έχουν την τιμή SYS.

```
-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
```

Παράδειγμα 5. 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
```

Παράδειγμα 6. 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 ίσος με 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
```

Παράδειγμα 7. 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$
```

Παράδειγμα 8. Αντιγραφή αρχείου με τις συναρτήσεις της C (αρχείο myCopyc.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
```

Σε αυτό το παράδειγμα υλοποιούμε την εφαρμογή αντιγραφής αρχείου χρησιμοποιώντας **τις συναρτήσεις της γλώσσας C** προκειμένου να γίνει αντιληπτή η διαφορά στη σύνταξη. Αντί για τις συναρτήσεις **open**, **close**, **read** και **write**, χρησιμοποιούνται οι **fopen**, **fclose**, **fread** και **frite**. Σε αντίθεση με τις κλήσεις συστήματος χαμηλού επιπέδου που προσδιορίζουν το αρχείο χρησιμοποιώντας **έναν απλό περιγραφέα αρχείου** που είναι μία ακέραια τιμή, οι συναρτήσεις υψηλού επιπέδου της γλώσσας C χρησιμοποιούν μία ολόκληρη δομή τύπου **FILE**, με τον περιγραφέα αρχείου να αποτελεί ένα από τα πεδία αυτής της δομής. Επειδή η ανάγνωση και η εγγραφή γίνονται διαβάζοντας και γράφοντας **ένα byte κάθε φορά**, το αρχείο –πηγή ανοίγει σε κατάσταση **rb (read binary)** ενώ το αρχείο προορισμού ανοίγει σε κατάσταση **wb (write binary)**. Η διαδικασία της αντιγραφής πραγματοποιείται όπως και πριν μέσω μιας επαναληπτικής διαδικασίας και σε κάθε κύκλο επανάληψης μεταφέρεται ένα απλό byte από το αρχείο πηγή στο αρχείο προορισμού. Η δομή του κώδικα και ο τρόπος λειτουργίας του είναι ακριβώς ίδια με πριν και δεν απαιτείται να γίνει κάποιος πρόσθετος σχολιασμός.

```
int main(int argc, char * argv[]) {
   char cTemp;
   FILE * src, * dest;

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

   src = fopen(argv[1], "rb");
   if (!src) {
      printf ("Source File could not be opened! Aborting...");
      return (-2); }

   dest = fopen(argv[2], "wb");
   if (!src) {
      printf ("Destination file could not be opened! Aborting...");
      return (-3); }

   while (fread(&cTemp, 1, 1, src) == 1) {
         fwrite(&cTemp, 1, 1, dest); }

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

Παράδειγμα 9. Δημιουργία και διαγραφή καταλόγου (αρχείο dirFun.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>
#include <dirent.h>

int main(int argc, char * argv[]) {
   long size; int retVal; DIR * dir;
   char * buf, * ptr, arg [20]="ls -l | grep ";
```

Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **dirFun directory-name** όπου **directory-name** το όνομα του καταλόγου που θέλουμε να χρησιμοποιήσουμε. 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: dirFun dirname\n");
      return (-1); }
```

Ο στόχος της εφαρμογής είναι η **δημιουργία** του καταλόγου το όνομα του οποίου ορίζεται από το χρήστη. Για το λόγο αυτό καλούμε τη συνάρτηση **opendir** η οποία ανοίγει έναν κατάλογο και ελέγχουμε την επιστρεφόμενη τιμή της. Εάν αυτή η τιμή είναι διαφορετική του μηδενός, η εφαρμογή εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία της, γιατί προφανώς **δεν μπορούμε** να κατασκευάσουμε έναν κατάλογο που να έχει το ίδιο όνομα με έναν υπάρχοντα κατάλογο.

```
   dir = opendir(argv[1]);
   if (dir) {
      printf ("Directory exists. Aborting...\n");
   closedir(dir);
   return (-2);  }
```

Στο σημείο αυτό η εφαρμογή ανακτά και εκτυπώνει το όνομα του τρέχοντος καταλόγου καλώντας τη συνάρτηση **getcwd** η οποία επιστρέφει την απόλυτη διαδρομή αυτού του καταλόγου. Επειδή **δεν γνωρίζουμε** το μήκος σε χαρακτήρες αυτής της απόλυτης διαδρομής καλούμε τη συνάρτηση **pathconf** με ορίσματα τον τρέχοντα κατάλογο (**.**) και τη σταθερά **_PC_MAX_PATH** η οποία επιστρέφει την τιμή αυτού του μήκους. Στη συνέχεια καλούμε τη συνάρτηση **malloc** για να δεσμεύσουμε την περιοχή μνήμης **buf** με αυτό το μέγεθος η οποία στη συνέχεια χρησιμοποιείται ως όρισμα στην **getcwd** που επιστρέφει την απόλυτη διαδρομή του τρέχοντος καταλόγου.

```
   size = pathconf(".", _PC_PATH_MAX);
   if ((buf = (char *)malloc((size_t)size)) != NULL)
      ptr = getcwd(buf, (size_t)size);
   printf ("Current Working Directory is %s\n", buf);
   printf ("Creating directory %s inside directory %s\n", argv[1], buf);
   free(buf);
```

Μετά την εκτύπωση των κατάλληλων ενημερωτικών μηνυμάτων, η εφαρμογή καλεί τη συνάρτηση **mkdir** για να δημιουργήσει τον κατάλογο που όρισε ο χρήστης και το όνομα του οποίου είναι αποθηκευμένο στη συμβολοσειρά argv[1]. Ο χρήστης πρέπει επίσης να ορίσει και τα **δικαιώματα πρόσβασης** για το νέο κατάλογο που εδώ είναι τα **755 (r w x r - x r – x).** Εάν η συνάρτηση επιστρέψει την τιμή -1 αυτό σημαίνει πως έχει λάβει χώρα κάποιο σφάλμα, οπότε η εφαρμογή καλεί τη συνάρτηση **perror** για να εκτυπώσει μία λεκτική περιγραφή αυτού του σφάλματος και τερματίζει τη λειτουργία της.

```
   retVal = mkdir (argv[1],0755);
   if (retVal==-1) {
      printf("Error Number : %d\n", errno);
      perror("Error Description");
      return (-2); }
```

Προκειμένου να επαληθεύσουμε πως ο κατάλογος δημιουργήθηκε με επιτυχία, καλείται η εντολή **ls –l | grep argv[1]** (με τη συμβολοσειρά **argv[1]** να περιέχει το όνομα του καταλόγου που δημιουργήθηκε) για να εκτυπωθεί η γραμμή για τον τρέχοντα

κατάλογο. Η συμβολοσειρά ls –l | grep argv[1] προκύπτει από τη **συνένωση** της συμβολοσειράς ls –l | grep που βρίσκεται αποθηκευμένη στη μεταβλητή arg και της συμβολοσειράς argv[1] (διαδικασία η οποία πραγματοποιείται από τη συνάρτηση **strcat** του αρχείου string.h) και διαβιβάζεται ως όρισμα στη συνάρτηση **system** η οποία και εκτελεί την εντολή. Στο τέλος της διαδικασίας καλείται η συνάρτηση rmdir για τη διαγραφή του καταλόγου που δημιουργήσαμε παραπάνω και η εφαρμογή τερματίζει.

```
strcat (arg, argv[1]);
system (arg);
printf ("Removing directory %s\n", argv[1]);
rmdir (argv[1]);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~/shared/Lab5$ ./dirFun myDir
Current Working Directory is /home/amarg/shared/Lab5
Creating directory myDir inside directory /home/amarg/shared/Lab5
drwxrwxrwx 1 root root     0 Νοε   6  2020 myDir
Removing directory myDir
amarg@amarg-vbox:~/shared/Lab5$
```

Παράδειγμα 10. Εκτύπωση περιεχομένων καταλόγου (αρχείο myLS.c).

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

int main(int argc, char * argv[]) {
   DIR * dip;
   struct dirent * dit;
   int files = 0;
```

Η κατάσταση είναι ακριβώς ίδια με πριν. Η σωστή εκτέλεση της εφαρμογής απαιτεί την κλήση της με τη μορφή **myLs directory-name** όπου **directory-name** το όνομα του καταλόγου τα περιεχόμενα του οποίου θέλουμε να εμφανίσουμε. 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: myLs dirname\n");
   return (-1); }
```

Ο στόχος της εφαρμογής είναι η **εμφάνιση των περιεχομένων** του καταλόγου το όνομα του οποίου ορίζεται από το χρήστη και ο οποίος φυσικά θα πρέπει να υπάρχει. Για το λόγο αυτό καλούμε τη συνάρτηση **opendir** η οποία ανοίγει έναν κατάλογο και ελέγχουμε την επιστρεφόμενη τιμή της. Εάν αυτή η τιμή είναι η τιμή **NULL** αυτό σημαίνει πως η κλήση της opendir δεν είχε επιτυχία. Για το λόγο αυτό, η εφαρμογή εκτυπώνει ένα ενημερωτικό μήνυμα και τερματίζει τη λειτουργία της.

```
if ((dip = opendir (argv[1]))==NULL) {
   printf("Error Number : %d\n", errno);
   perror("Error Description");
   return (-2); }
```

Η εμφάνιση των περιεχομένων του καταλόγου πραγματοποιείται από την **επαναληπτική** κλήση της συνάρτησης **readdir** η οποία επιστρέφει έναν **δείκτη** προς την επόμενη καταχώρηση του καταλόγου ή τιμή **NULL** εάν έχει φτάσει στο τέλος του καταλόγου ή έχει συμβεί κάποιο σφάλμα. Μέσα λοιπόν από έναν βρόχο επανάληψης που **εκτελείται για όσο χρονικό**

διάστημα η συνάρτηση readdir δεν επιστρέφει τιμή NULL, η εφαρμογή ανακτά το όνομα της **επόμενης καταχώρησης** του καταλόγου που είναι αποθηκευμένο στη μεταβλητή **d_name** (θυμηθείτε από τη C πως ο τελεστής → χρησιμοποιείται για την προσπέλαση πεδίων δομής η οποία προσπελαύνεται μέσω **δείκτη**) και το εκτυπώνει στην οθόνη. Με τον τρόπο αυτό, στο τέλος της διαδικασίας, θα έχουν εκτυπωθεί στην οθόνη τα περιεχόμενα του καταλόγου που έχει ορίσει ο .

```
printf ("List of contents of directory %s\n", argv[1]);
while ((dit = readdir(dip))!=NULL) {
      files++;
      printf("%s\n", dit->d_name); }
printf ("Directory %s contains %d file(s)\n", argv[1], files);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Το πρόγραμμα εκτυπώνει τα ονόματα των περιεχομένων του καταλόγου (αρχεία και κατάλογοι), ένα όνομα σε κάθε γραμμή συμπεριλαμβανομένων και των κρυφών αντικειμένων.

```
amarg@amarg-vbox:~$ ./myLs .
List of contents of directory .
.bashrc
months.txt
filestat.o
file2.zip
Templates
forkex5
lex
myCopyc.o
forkex4
pid.c
Downloads
file1.txt
forkex5.c
testChmod
flength
file3.zip
.bash_logout
.vboxclient-draganddrop.pid
.sudo_as_admin_successful
.bash_history
rfile
forkex7
```

ΕΡΓΑΣΤΗΡΙΟ 6

Αγωγοί

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// child process

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

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

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

// parent process

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

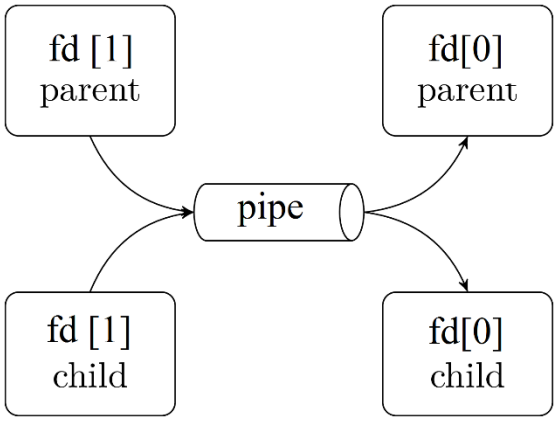
Η γονική διεργασία αποστέλλει την έξοδο της στη θυγατρική διεργασία σε ένα και μόνο βήμα, δηλαδή αποστέλλει όλο το μήνυμα με τη μία, χρησιμοποιώντας ως όρισμα στη συνάρτηση write τη συμβολοσειρά msg.

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

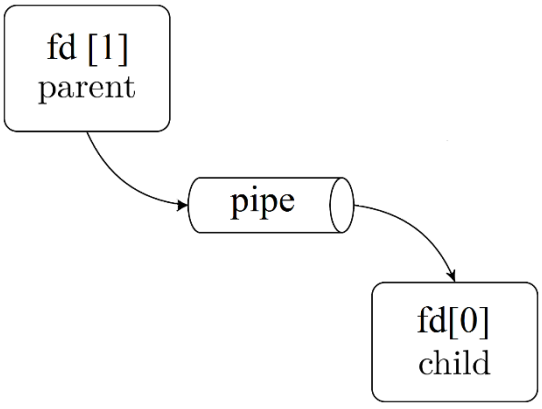
Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

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

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



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



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

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

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

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

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

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

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

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

pid_t cpid = fork();

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

Κώδικας θυγατρικής διεργασίας

```
// child process

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

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

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

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

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

Κώδικας γονικής διεργασίας

```
// parent process

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

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

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

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

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

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
// Fork a process to run grep

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Παράδειγμα 5. Παράδειγμα δημιουργίας επώνυμου αγωγού

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

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

Εφαρμογή 1 (αρχείο side1.c)

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

```
int main() {
   int fd;

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

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

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

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

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

```
   while (1) {
```

Σε κάθε κύκλο επανάληψης

ΑΡΧΙΚΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΕΓΓΡΑΦΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο εγγραφής (O_WRONLY)

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

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

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

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

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

ΣΤΗ ΣΥΝΕΧΕΙΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΑΝΑΓΝΩΣΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο ανάγνωσης (O_RDONLY)

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

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

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

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

Εφαρμογή 2 (αρχείο side2.c)

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

int main() {
   int fd1;
```

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

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

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

   mkfifo(myfifo, 0666);

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

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

```
   while (1) {
```

ΑΡΧΙΚΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΑΝΑΓΝΩΣΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο ανάγνωσης (O_RDONLY)

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

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

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

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

      close(fd1);
```

ΣΤΗ ΣΥΝΕΧΕΙΑ ΠΡΑΓΜΑΤΟΠΟΙΟΥΜΕ ΔΙΑΔΙΚΑΣΙΑ ΕΓΓΡΑΦΗΣ

Ανοίγουμε τον επώνυμο αγωγό σε κατάσταση μόνο εγγραφής (O_WRONLY)

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

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

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

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

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

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

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

1+1 = ?
User2: 2

2+4 = ?
User2: 6

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

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

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

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

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

2
User1: 2+4 = ?

6
User1: 5 + 5 10?

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

bye bye :-(
User1: bye !! lol

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

ΕΡΓΑΣΤΗΡΙΟ 7

Σήματα

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 3. Τα σήματα γενικής χρήσεως SIGUSR1 και SIGUSR2
(αρχείο sigExample3.c).

Αυτό το παράδειγμα επιδεικνύει τη χρήση των σημάτων SIGUSR1 και SIGUSR2 τα οποία είναι γενικής χρήσεως και μπορούν να χρησιμοποιηθούν για να καλύψουν τις υφιστάμενες σε κάθε περίπτωση ανάγκης του χρήστη. Στο απλό παράδειγμα που ακολουθεί. ο χρήστης εκτελεί την εφαρμογή sigExample3 με το γνωστό τρόπο. Στη συνέχεια ανοίγει ένα δεύτερο τερματικό, βρίσκει το process ID της διεργασίας με την εντολή ps -x και κατόπιν χρησιμοποιώντας την εντολή kill, αποστέλει στη διεργασία τα δύο παραπάνω σήματα. Αυτά τα δύο σήματα έχουν συσχετιστεί με την ίδια συνάρτηση χειρισμού, η οποία όταν καλείται κατά την λήψη του σήματος, απλά εκτυπώνει ένα ενημερωτικό μήνυμα σύμφωνα με το οποίο τα σήματα παρελήφθησαν.

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

Ο χειριστής των σημάτων SIGUSR1 και SIGUSR2

```
void sig_handler(int signo) {
   if (signo == SIGUSR1)
      printf("received USR1\n");
   if (signo == SIGUSR2)
      printf("received USR2\n"); }
```

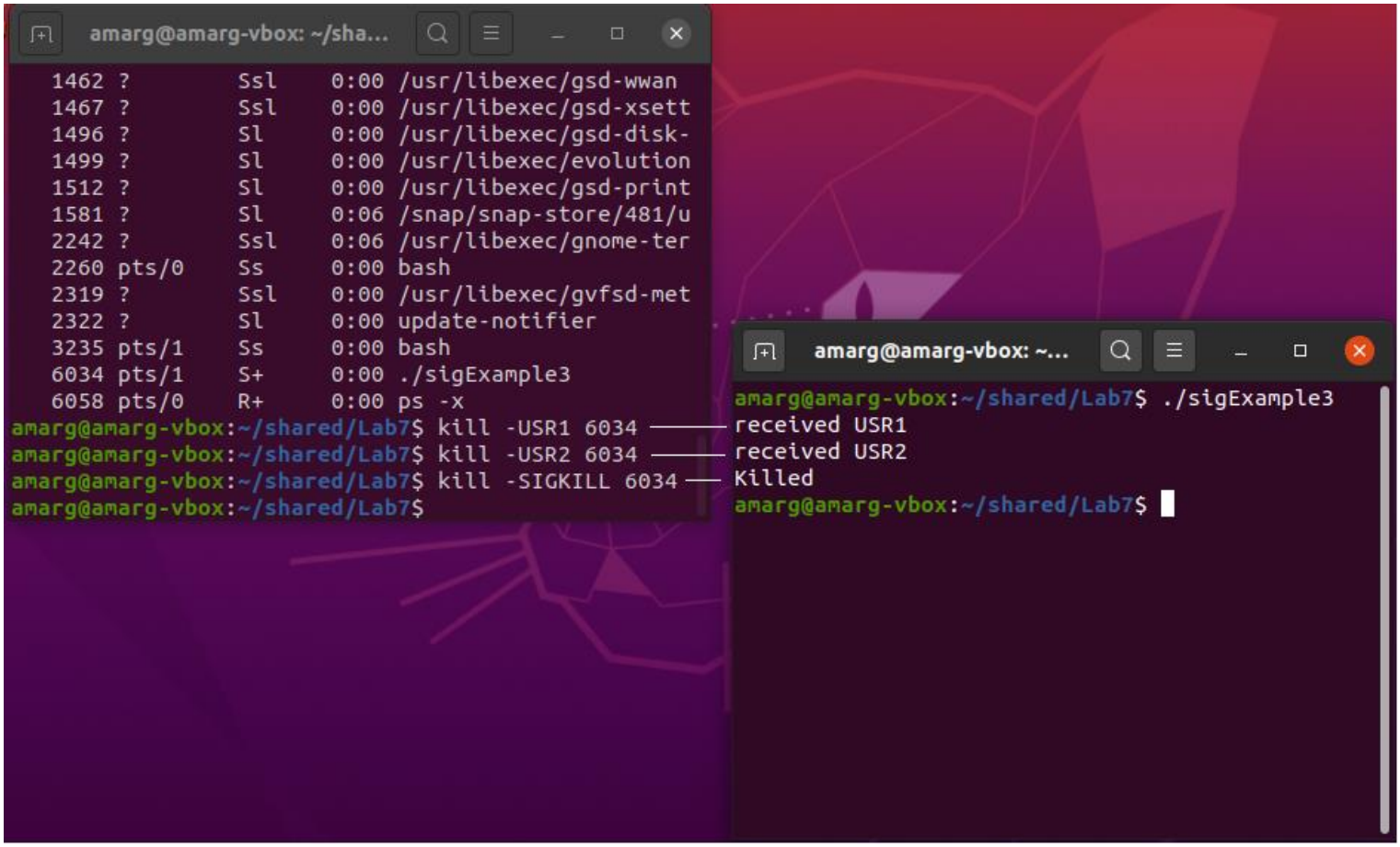
Η συνάρτηση signal συσχετίζει τα σήματα SIGUSR1 και SIGUSR2 με το χειριστή handler και σε περίπτωση που αυτό δεν συμβεί επιστρέφει τον κωδικό σφάλματος SIG_ERR.

```
int main(void) {
   if (signal(SIGUSR1, sig_handler) == SIG_ERR)
       printf("\ncan't catch USR1\n");
   if (signal(SIGUSR2, sig_handler) == SIG_ERR)
       printf("\ncan't catch USR2\n");

  // A long long wait so that we can easily
  // issue a signal to this process
```

Όπως και προηγουμένως μετά τα παραπάνω 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.



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 θυγατρικηή 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...
```

Παράδειγμα 6. Ανταλλαγή σήματος μεταξύ γονικής και πολλών θυγατρικών διεργασιών (αρχείο sigExample6.c).

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

Σε αυτό το παράδειγμα η γονική διεργασία δημιουργεί τέσσερις θυγατρικές διεργασίες και ανιχνεύει τον τερματισμό τους δια της χρήσεως του σήματος SIGCHILD.

```
#define NUMPROCS 4   /* number of processes to fork */
int nprocs;          /* number of child processes */
```

Η κάθε θυγατρική διεργασία που δημιουργείται, εκτελεί τη συνάρτηση chlild (n) όπου n ένας ακέραιος αριθμός που παίρνει μικρές θετικές τιμές και κάνει κάτι πάρα πολύ απλό (για λόγους επίδειξης και μόνο): εκτυπώνει το PID της καθώ και το PID του γονέα της, αναστέλλει τη λειτουργία της για n δευτερόλεπτα (καλώντας τη sleep) και στη συνέχεια καλεί τη συνάρτηση exit για να τερματιστεί (εκτυπώνοντας το κατάλληλο μήνυμα), με κωδικό επιστροφής 100+n (μπορείτε αν θέλετε να αλλάξετε αυτή τη συμπεριφορά και να καθορίσετε να κάνει κάτι διαφορετικό).

```
void child (int n) {
    printf("\tChild[%d]: child pid=%d, sleeping for %d seconds\n", n, getpid(), n);
    sleep(n);
    printf("\tchild[%d]: I'm exiting\n", n);
    exit(100+n); }
```

Η συνάρτηση catch αποτελεί τη συνάρτηση χειρισμού του σήματος SIGCHLD – αυτό δηλώνεται μέρα στη main. Αν και δέχεται ένα όρισμα snum, ωστόσο δεν το χρησιμοποιεί πουθενά (!). Η συνάρτηση catch καλεί τη wait προκειμένου μέσω της δομής status και δια της χρήσεως της μακροεντολής WEXITSTATUS να ανακτήσει τον κωδικό επιστροφής της θυγατρικής διεργασίας που ολοκληρώθηκε τελευταία, ενώ μειώνει το πλήθος των διεργασιών κατά μία μονάδα.

```
void catch (int snum) {
    int pid, status;
    pid = wait(&status);
    printf("Parent process: child process pid=%d exited with value %d\n",
            pid, WEXITSTATUS(status));
    nprocs--; }
```

```
int main (int argc, char **argv) {
    int pid, i;
```

Το πρώτο πράγμα που κάνουμε στη συνάρτηση main είναι να συσχετίσουμε την συνάρτηση catch με το σήμα SIGCHLD έτσι ώστε η εν λόγω συνάρτηση να χρησιμοποιηθεί ως η συνάρτηση χειρισμού αυτού του σήματος.

```
    signal(SIGCHLD, catch);
```

Η γονική διεργασία μέσα από ένα βρόχο που εκτελείται NUMPROCS φορές καλεί τη fork για να δημιουργήσει ισάριθμες θυγατρικές διεργασίες.

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

Η κάθε θυγατρική διεργασία καλεί τη συνάρτηση child (i) όπου i η τρέχουσα τιμή του μετρητή του βρόχου.

```
        if (pid==0) child(i);
```

Η γονική διεργασία σε κάθε κύκλο επανάληψης αυξάνει την τιμή του nprocs κατά μία μονάδα (αφού σε κάθε κύκλο επανάληψης δημιουργείται και μία νέα διεργασία).

```
        else nprocs++; }
```

Για όσο χρονικό διάστημα υπάρχουν διεργασίες, η γονική διεργασία αναστέλλει τη λειτουργία της καλώντας τη συνάρτηση sleep. Καθώς οι θυγατρικές διεργασίες ολοκληρώνονται η μία μετά την άλλη και στέλνουν το σήμα SIGCHLD στη γονική διεργασία, αυτό διαβάζεται από τη συνάρτηση catch η οποία εκτυπώνει το μήνυμα τερματισμού της κάθε διεργασίας και μειώνει το πλήθος των διεργασιών κατά μία μονάδα. Όταν τερματιστούν όλες οι θυγατρικές διεργασίες (οπότε είναι nprocs = 0) ενεργοποιείται η γονική διεργασία η οποία τερματίζεται και αυτή.

```
    printf("Parent process: going to sleep\n");
    while (nprocs != 0) {
            printf("parent: sleeping\n");
            sleep(60); }
    printf("Parent process: exiting\n");
   exit(0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./sigExample6
Parent process: going to sleep
parent: sleeping
        Child[3]: child pid=1702, sleeping for 3 seconds
        Child[2]: child pid=1701, sleeping for 2 seconds
        Child[1]: child pid=1700, sleeping for 1 seconds
        Child[0]: child pid=1699, sleeping for 0 seconds
        child[0]: I'm exiting
Parent process: child process pid=1699 exited with value 100
parent: sleeping
        child[1]: I'm exiting
Parent process: child process pid=1700 exited with value 101
parent: sleeping
        child[2]: I'm exiting
Parent process: child process pid=1701 exited with value 102
parent: sleeping
        child[3]: I'm exiting
Parent process: child process pid=1702 exited with value 103
Parent process: exiting
```

Παράδειγμα 7. Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος – Παράδειγμα Α (αρχείο sigExample7.c).

Σε αυτό το παράδειγμα, η διεργασία μπλοκάρει το σήμα SIGTERM για δύο δευτερόλεπτα χρησιμοποιώντας τη συνάρτηση sigprocmask. Μετά την παρέλευση των δύο δευτερολέπτων το σήμα ξεμπλοκάρεται.

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

Η μεταβλητή got_signal έχει τιμή 0 όταν δεν έχει παραληφθεί ένα σήμα και τιμή 1 στην αντίθετη περίπτωση.

```
static int got_signal = 0;
```

Η συνάρτηση χειρισμού σήματος απλά θέτει την καθολική μεταβλητή got_signal στην τιμή 1.

```
static void hdl (int sig) {
    got_signal = 1; }

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

Οι επόμενες δηλώσεις μεταβλητών και αρχικοποιήσεις σχετίζονται με την δομή act που περνάμε ως όρισμα στην sigaction.

```
    sigset_t mask;
    sigset_t orig_mask;
    struct sigaction act;
```

Η δομή act αρχικοποιείται στη μηδενική τιμή και στη συνέχεια το πεδίο handler τίθεται στη συνάρτηση hdl που απλά αρχικοποιεί η μεταβλητή got_signal στην τιμή 1 για να υποδηλώσει πως παρελήφθει ένα σήμα.

```
    memset (&act, 0, sizeof(act));
    act.sa_handler = hdl;
```

Η συνάρτηση sigaction που έχει αντικαταστήσει την παρωχημένη συνάρτηση signal συσχετίζει το σήμα SIGTERM με τη δομή act και έμμεσα με τη συνάρτηση χειρισμού hdl.

```
    if (sigaction(SIGTERM, &act, 0)) {
       perror ("sigaction");
      return 1; }
```

Προκειμένου να αρχικοποιήσουμε το σύνολο σημάτων mask έτσι ώστε να περιέχει μόνο το σήμα SIGTERM που θεωρούμε σε αυτό το παράδειγμα, χρησιμοποιούμε τον πρώτο τρόπο αρχικοποίησης, δηλαδή, εκκενώνουμε τη μάσκα από όλα τα σήματα καλώντας τη συνάρτηση sigemptyset και μετά προσθέτουμε σε αυτή το σήμα SIGTERM με τη συνάρτηση sigaddset.

```
    printf ("Emptying signal set...\n");
    sigemptyset (&mask);
    printf ("Adding SIGTERM to signal set...\n");
    sigaddset (&mask, SIGTERM);

   // SIGTERM signal is added to orig_mask
   printf ("Adding signal set to the original mask...\n");
```

Έχοντας αρχικοποιήσει το σύνολο σημάτων mask έτσι ώτε να περιέχει μόνο το σήμα SIGTERM, καλούμε τη συνάρτηση sigpropmask έτσι ώστε να προσθέσουμε το σύνολο σημάτων της mask (δηλαδή ουσιαστικά το σήμα SIGTERM) στην αρχική μάσκα orig_mask.

Θυμηθείτε πως ο σκοπός της άσκησης είναι να μπλοκάρουμε (δείτε το όρισμα SIG_BLOCK) το σήμα SIGTERM. Η νέα μάσκα που προκύπτει από την κλήση της sigprocmask προκύπτει από την ένωση της αρχικής μάσκας και του συνόλου mask που περέχει τα σήματα (στην προκειμένη περίπτωση το σήμα SIGTERM) που επιθυμούμε να μπλοκάρουμε.

```
if  (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0) {
     perror ("sigprocmask");
     return 1; }
```

Στη συνέχεια καλούμε ξανά την sigpropmask αλλά αυτή τη φορά με όρισμα SIG_SETMASK έτσι ώστε να ορίσουμε τη νέα μάσκα ως την τρέχουσα μάσκα, διαδικασία που θα οδηγήσει και στο μπλοκάρισμα του σήματος.

```
// SIGTERM signal is blocked
printf ("Blocking SIGTERM signal...\n");
if  (sigprocmask(SIG_SETMASK, &orig_mask, NULL) < 0) {
     perror ("sigprocmask");
     return 1; }
```

Μετά την παρέλευση δύο δευτερολέπτων (η αναστολή γίνεται με τη sleep) το σήμα ξεμπλοκάρεται.

```
printf ("Sleeping for 2 seconds\n");
sleep (2);
if (got_signal) puts ("Got signal");
return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./sigExample7
Emptying signal set...
Adding SIGTERM to signal set...
Adding signal set to the original mask...
Blocking SIGTERM signal...
Sleeping for 2 seconds
amarg@amarg-vbox:~$ gedit sigExample7.c
```

Παράδειγμα 8. Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος – Παράδειγμα Β (αρχείο sigExample8.c).

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

Εδώ χρησιμοποιούνται τα σήματα SIGUSR1 (σήμα γενικής χρήσεως), SIGHUP (στη γενική περίπτωση ενημερώνει τη διεργασία πως το τερματικό που την παρακολουθεί έχει αποσυνδεθεί) και SIGINT (που προκαλεί τη διακοπή της)

void handle_signal(int signal); **Η συνάρτηση χειρισμού των τριών σημάτων**

void handle_sigalrm (int signal); **Ενημερώνει το χρήστη ότι παρέλαβε το σήμα SIGALARM**

void do_sleep(int seconds); **Η βασική συνάρτηση της εφαρμογής**

Στο κυρίως πρόγραμμα αρχικοποιούμε τη δομή sa την οποία σχετίζουμε με τα τρία σήματα ενδιαφέροντος και στη συνέχεια μέσα από έναν ατερόνα βρόχο καλούμε τη συνάρτηση do_sleep.

```
int main() {
   struct sigaction sa;
   printf("My pid is: %d\n", getpid());  // we need the pid to use kill from another terminal
   sa.sa_handler = &handle_signal;   // here we assign the signal handler
```

Η σημαία SA_RESTART ελέγχει το τι ακριβώς συμβαίνει όταν ένα σήμα αποσταλεί και η συνάρτηση χειρισμού του ολοκληρωθεί χωρίς πρόβλημα. Υπάρχουν δύο εναλλακτικές που μπορούν να εμφανιστούν: είτε η βιβλιοθήκη που κάλεσε τη συνάρτηση να συνεχίσει να λειτουργεί, είτε να επιστρέψει κωδικό σφάλματος. Εάν χρησιμοποιηθεί αυτή η σημαία τότε η επιστροφή από το χειριστή του σήματος επιτρέπει τη συνέχιση της διαδικασίας.

```
sa.sa_flags = SA_RESTART;
```

Κατά τη διάρκεια της εκτέλεσης της συνάρτησης χειρισμού μπλοκάρουμε όλα τα σήματα,

```
// Block every signal during the handler
sigfillset(&sa.sa_mask);
```

Εδώ συσχετίζουμε τα τρία σήματα με τη δομή sa που αρχικοποιήθηκε προηγουμένως.

```
// Signals SIGHUP, SIGUSR1 and SIGINT are associated with struct sa

if (sigaction(SIGHUP, &sa, NULL) == -1) { perror("Error: cannot handle SIGHUP"); }
if (sigaction(SIGUSR1, &sa, NULL) == -1) { perror("Error: cannot handle SIGUSR1"); }
if (sigaction(SIGINT, &sa, NULL) == -1) { perror("Error: cannot handle SIGINT"); }
```

Το σήμα SIGKILL δεν μπορεί να δηλωθεί εδώ αφού τερματίζει τη διεργασία.

```
// Will always fail, SIGKILL is intended to force kill your process
if (sigaction(SIGKILL, &sa, NULL) == -1) { perror("Cannot handle SIGKILL"); }
```

Το πρόγραμμα μπαίνει σε ένα infinite loop καλώντας επαναληπτικά τη συνάρτηση do_sleep.

```
for (;;) {
    printf("\nSleeping for ~3 seconds\n");
    do_sleep(3); }}
```

Η συνάρτηση χειρισμού των τριών σημάτων αρχικοποιεί το όνομα του σήματος που παραλαμβάνεται κάθε φορά. Εάν το σήμα αυτό είναι το SIGINT εκτυπώνει ένα ενημερωτικό μήνυμα πως η διεργασία θα τερματιστεί ενώ αν παραλάβει κάτι μη αναμενόμενο εκτυπώνει μήνυμα σφάλματος στο αρχείο stderr.

```
void handle_signal (int signal) {
  const char *signal_name;
  sigset_t pending;
  // Find out which signal we're handling
  switch (signal) {
    case SIGHUP:
      signal_name = "SIGHUP";
      break;
    case SIGUSR1:
      signal_name = "SIGUSR1";
      break;
    case SIGINT:
      printf("Caught SIGINT, exiting now\n");
      exit(0);
    default:
      fprintf(stderr, "Caught wrong signal: %d\n", signal);
      return; }
```

Μετά την παραλαβή του σήματος η συνάρτηση ζητά από το χρήστη να στείλει ένα άλλο σήμα και τον ενημερώνει πως θα αναστείλει τη λειτουργία της για τρία δευτερόλεπτα.

```
// However, printf in signal handlers IS NOT reccommented !
printf("Caught %s, sleeping for ~3 seconds\n"
    "Try sending another SIGHUP / SIGINT / SIGALRM "
    "(or more) meanwhile\n", signal_name);
do_sleep(3);
printf("Done sleeping for %s\n", signal_name);
```

Εάν σε αυτό το χρονικό διάστημα των τριών δευτερολέπτων ο χρήστης στείλει νέα σήματα στην εργασία αυτά θα μπούνε στη λίστα των εκκρεμών σημάτων προκειμένου η εργασία να τα παραλάβει και να ανταποκριθεί ανάλογα όταν επανέλθει σε λειτουργία. Η ανάκτηση της λίστας αυτών των εκκρεμών σημάτων γίνεται με τη συνάρτηση sigpending η οποία επιστρέφει την εν λόγω πληροφορία στη μεταβλητή pending που είναι τύπου sigset_t.

```
// So what did you send me while I was asleep?
sigpending(&pending);
```

Εάν σε αυτή τη λίστα υπάρχουν τα σήματα SIGHUP και SIGUSR1 η εφαρμογή εκτυπώνει το κατάλληλο ενημερωτικό μήνυμα.

```
if (sigismember(&pending, SIGHUP)) { printf("A SIGHUP is waiting\n"); }
if (sigismember(&pending, SIGUSR1)) { printf("A SIGUSR1 is waiting\n"); }
printf("Done handling %s\n\n", signal_name); }
```

Η συνάρτηση handle_sigalarm εκτυπώνει απλά ένα ενημερωτικό μήνυμα σχετικά με το σήμα που παρέλαβε. Το σήμα που αναμένει να λάβει είναι το SIGALARM που δημιουργείται από την alarm, διαφορετικά εκτυπώνει μήνυμα σφάλματος.

```
void handle_sigalrm(int signal) {
    if (signal != SIGALRM) { fprintf(stderr, "Caught wrong signal: %d\n", signal); }
    printf("Got sigalrm, do_sleep() will end\n"); }
```

```
void do_sleep(int seconds) {
```

Η συνάρτηση do_sleep συσχετίζει το SIGALARM με τη δομή sa που αρχικοποιείται κατάλληλα.

```
    struct sigaction sa;
    sigset_t mask;

    sa.sa_handler = &handle_sigalrm;
```

***SA_RESETHAND* == > Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler.**

```
    sa.sa_flags = SA_RESETHAND;
    sigfillset(&sa.sa_mask);

    sigaction(SIGALRM, &sa, NULL);
```

Ανακτάται η τρέχουσα μάσκα,

```
    // Get the current signal mask
    sigprocmask(0, NULL, &mask);
```

αφαιρείται από αυτή το SIGALARM

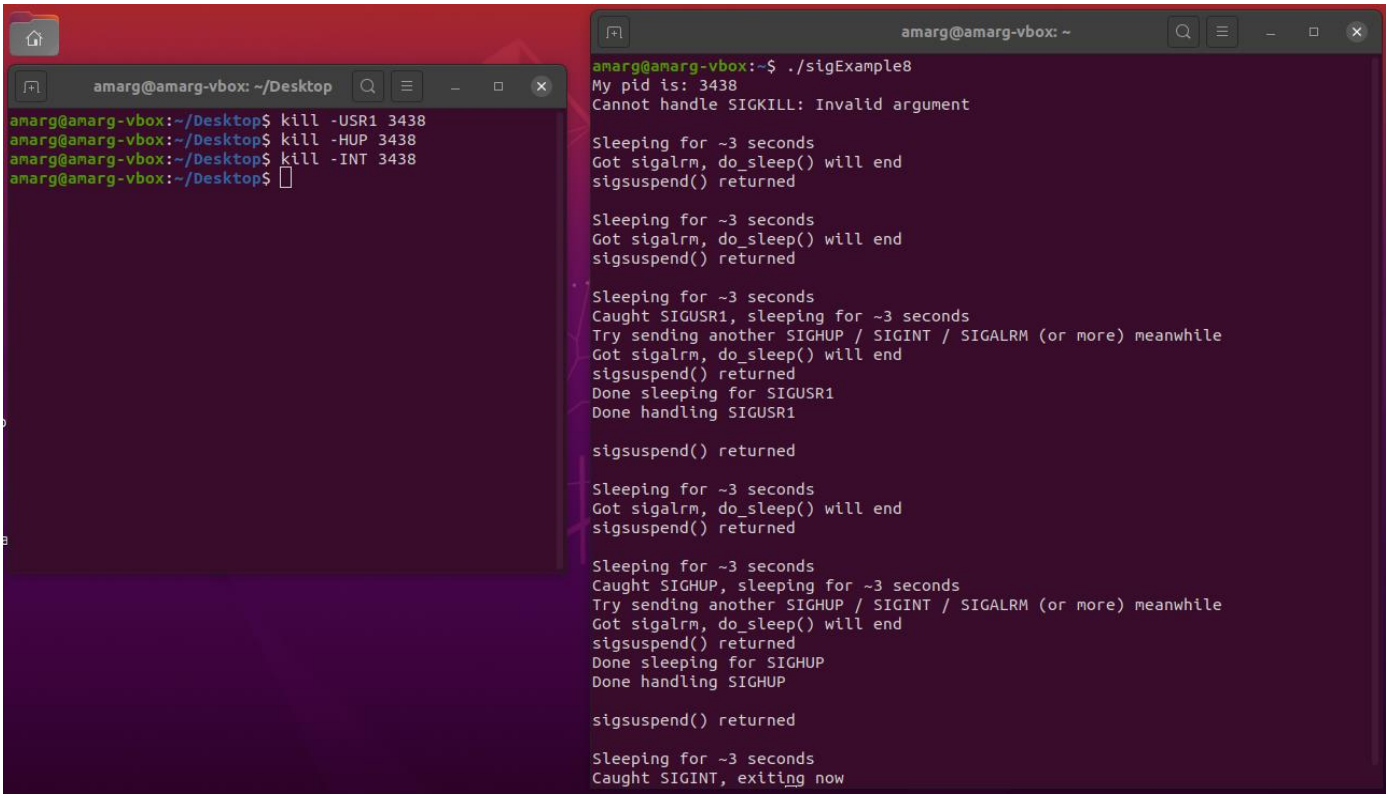
```
    // Unblock SIGALRM
    sigdelset(&mask, SIGALRM);
```

και στη συνέχεια στέλνεται αυτό σήμα ύστερα από seconds δευτερόλεπτα. Αυτό γίνεται με τη συνάρτηση alarm.

Η συνάρτηση alarm() δέχεται ως όρισμα έναν ακέραιο αριθμό που εκφράζει ένα χρονικό διάστημα εκφρασμένο σε δευτερόλεπτα και μετά την παρέλευση αυτού του χρονικού διαστήματος δημιουργεί ένα σήμα SIGALARM η προεπιλεγμένη συμπεριφορά του οποίου είναι ο τερματισμός της διεργασίας (ωστόσο, μπορούμε χρησιμοποιώντας την κατάλληλη συνάρτηση χειρισμού, να τροποποιήσουμε αυτή τη συμπεριφορά ανάλογα με τις απαιτήσεις μας).

```
alarm(seconds);
sigsuspend(&mask);
printf("sigsuspend() returned\n"); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.



ΕΡΓΑΣΤΗΡΙΟ 8

Υποδοχείς

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

Παράδειγμα 1. 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:~$
```

Παράδειγμα 2. 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!
```

Παράδειγμα 3. 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

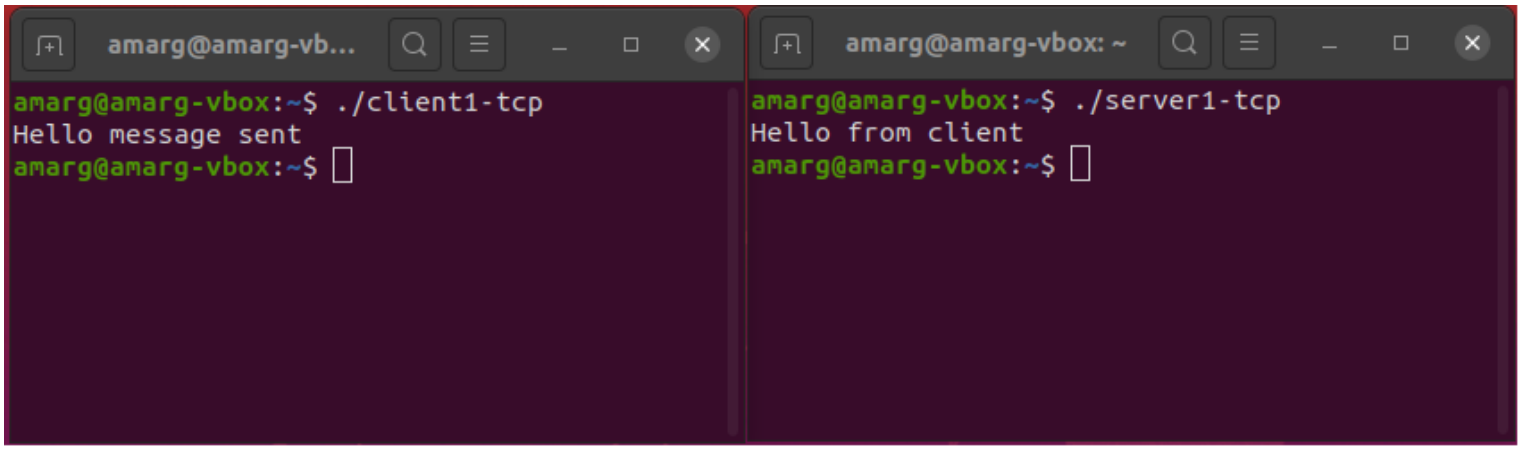
μηνύματος στο διακομιστή χρησιμοποιώντας τη συνάρτηση send. Η μοναδική νέα συνάρτηση που βλέπετε εδώ είναι η *inet_pton η οποία μετατρεπει μία διεύθυνση από μορφή κειμένου (για παράδειγμα 127.0.0.1) σε δυαδική μορφή (σε έναν δυαδικό αριθμό μήκους 32 bits) προκειμένου να χρησιμοποιηθεί στη συνάρτηση connect (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 4. Η εφαρμογή πελάτης συνδέεται σε έναν απομακρυσμένο Web server (port 80) η IP διεύθυνση του οποίου δίδεται από το χρήστη και διαβάζει τις 1000 πρώτες γραμμές του κώδικα HTML της κεντρικής σελίδας τις οποίες και εκτυπώνει. (αρχείο inetExample.c).

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

Όπως είναι γνωστό, η επικοινωνία με μία ιστοσελίδα του παγκόσμιου διαδικτύου, στηρίζεται στη χρήση του πρωτοκόλλου HTTP (Hhypertext Transfer Protocol) το οποίο μεταξύ άλλων προσφέρει τις δικές του εντολές για την ανταλλαγή πληροφοριών ανάμεσα στον Web Client και στο Web Server. Σε αυτό το παράδειγμα, θα χρησιμοποιήσουμε την εντολή GET του πρωτοκόλλου HTTP 1.0 για να διαβάσουμε τμήμα του κώδικα HTML της κεντρικής σελίδας του απομακρυσμένου δικτυακού τόπου. Χωρίς να είμαστε σε θέση σε αυτό το σημείο να δώσουμε πολλές πληροφορίες σχετικά με την ακριβή σύνταξη των εντολών του εν λόγω πρωτοκόλλου, αναφέρουμε απλά πώς το αίτημα (REQUEST) που θα πρέπει να στείλουμε στο διακομιστή για να πραγματοποιήσουμε αυτή τη διαδικασία θα πρέπει να έχει τη μορφή *"GET / HTTP/1.0\r\n\r\n" (για όποιον ενδιαφέρεται να ενημερωθεί σχετικά με τον τρόπο λειτουργίας και τα χαρακτηριστικά αυτού του πρωτοκόλλου υπάρχουν αναρίθμητες διευθύνσεις στο διαδίκτυο με τις σχετικές πληροφορίες όπως είναι για παράδειγμα η διεύθυνση https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol).

```
#define REQUEST "GET / HTTP/1.0\r\n\r\n"
```

Λαμβάνοντας υπόψη πως αυτή τη φορά θα συνδεθούμε σε έναν απομακρυσμένο διακομιστή ο κώδικας του οποίου προφανώς έχει γραφεί από κάποιον άλλο, το μόνο το οποίο θα κάνουμε είναι να γράψουμε τον κώδικα του πελάτη ο οποίος θα συνδεθεί σε αυτόν τον διακομιστή. Ο διακομιστής υπάρχει κάπου και απλά συνδεόμαστε σε αυτόν για να ζητήσουμε κάποια εξυπηρέτηση. Για να το κάνουμε βέβαια αυτό, θα πρέπει να γνωρίζουμε το πρωτόκολλο στο οποίο στηρίζεται η λειτουργία του (για παράδειγμα αν είναι Web server, FTP server, Email server, κ.τ.λ), έτσι ώστε να ξέρουμε με ποιον τρόπο θα επικοινωνήσουμε μαζί του. Στο εν λόγω παράδειγμα, ο απομακρυσμένος διακομιστής είναι ένας web server ο οποίος χρησιμοποιεί το πρωτόκολλο HTTP. Εάν λοιπόν θέλουμε να κατασκευάσουμε μία πραγματική επαγγελματική εφαρμογή και όχι ένα μικρό πρόγραμμα επίδειξης σαν και αυτό που παρουσιάζεται εδώ, θα πρέπει να γνωρίζουμε πλήρως τον τρόπο λειτουργίας και τις εντολές αυτού του πρωτοκόλλου.

```
int main(int argc, char * argv[]) {
   struct sockaddr_in sa;
   int fd_skt;
   char buf[1000];
   ssize_t nread;
```

Στην προκειμένη περίπτωση για να εκτελεστεί ο κώδικας θα πρέπει να δώσουμε από τη γραμμή εντολών την διεύθυνση του απομακρυσμένου διακομιστή στον οποίο θα συνδεθεί η εφαρμογή πελάτης μας. Αυτό σημαίνει πως η σωστή χρήση της εφαρμογής απαιτεί την καταχώρηση ακριβώς ενός ορίσματος, δηλαδή την εκχώρηση από το σύστημα στη μεταβλητή

argc της τιμής 2. Εάν δεν ισχύει αυτό, τότε το πρόγραμμα τερματίζεται εκτυπώνονται στο κατάλληλο ενημερωτικό μήνυμα χρήσης (πρόκειται για έναν έλεγχο που τον έχουμε κάνει αρκετές φορές στο παρελθόν και ως εκ τούτου θεωρείται γνωστός).

```
if (argc!=2) {
  printf ("Usage: inetExample xxx.xxx.xxx.xxx\n");
  exit (-1); }
```

Η αρχικοποίηση της δομής *sockaddr_in* γίνεται όπως και πριν, χρησιμοποιώντας ως όρισμα το *AF_INET* (αφού θα χρησιμοποιήσουμε το δικτυακό πρωτόκολλο *IPv4*) ενώ ως αριθμός θύρας χρησιμοποιούμε τον αριθμό *80* αφού είναι γνωστό πως αυτός είναι ο αριθμός θύρας του πρωτοκόλλου HTTP. Λαμβάνοντας υπόψη πως η διεύθυνση στην οποία θα συνδεθούμε θα δοθεί από το χρήστη μέσω του πληκτρολογίου, είναι προφανές πως αυτή θα ανακτηθεί από το όρισμα *argv[1]* το οποίο χρησιμοποιείται για αυτόν ακριβώς τον λόγο.

```
sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr =  inet_addr(argv[1]);
```

Οι εφαρμογές διακομιστή, είτε υλοποιούνται από εμάς είτε έχουν υλοποιηθεί από άλλους, είναι απομακρυσμένοι και απλά συνδεόμαστε σε αυτούς, δουλεύουν παντού και πάντα με τον ίδιο τρόπο, δηλαδή καλώντας τις συναρτήσεις socket, bind, listen και accept με τον τρόπο που περιγράψαμε στα προηγούμενα παραδείγματα. Ο πελάτης λοιπόν σε αυτό το παράδειγμα, θα κάνει ό,τι έκανε και πριν, δηλαδή θα καλέσει τη συνάρτηση socket με τις ίδιες ρυθμίσεις με αυτές του διακομιστή, έτσι ώστε τελικά να μπορέσει να επικοινωνήσει μαζί του και θα συνδεθεί σε αυτόν χρησιμοποιώντας τη συνάρτηση connect επίσης με τον τρόπο που είδαμε (γενικά είναι καλό να γνωρίζετε, πως όλες αυτές οι εφαρμογές πελάτη διακομιστή είναι εύκολες στον προγραμματισμό τους, όσον αφορά το κομμάτι της διασύνδεσης, διότι κάνουν όλες ακριβώς τα ίδια πράγματα).

```
if ((fd_skt = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Message from socket [client] : ");
    exit(-1); }

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

Μετά τη σύνδεση στο διακομιστή, ο πελάτης του αποστέλλει το αίτημα *REQUEST* με τιμή *"GET / HTTP/1.0\r\n\r\n"* γνωστοποιώντας του πως θέλει να διαβάσει δεδομένα από αυτόν. Όπως και στις προηγούμενες εφαρμογές, η αποστολή του μηνύματος πραγματοποιείται με τη συνάρτηση *write* και χρησιμοποιώντας τον περιγραφέα αρχείου *fd_skt* που επέστρεψε η συνάρτηση *connect*.

```
write (fd_skt, REQUEST, strlen(REQUEST));
```

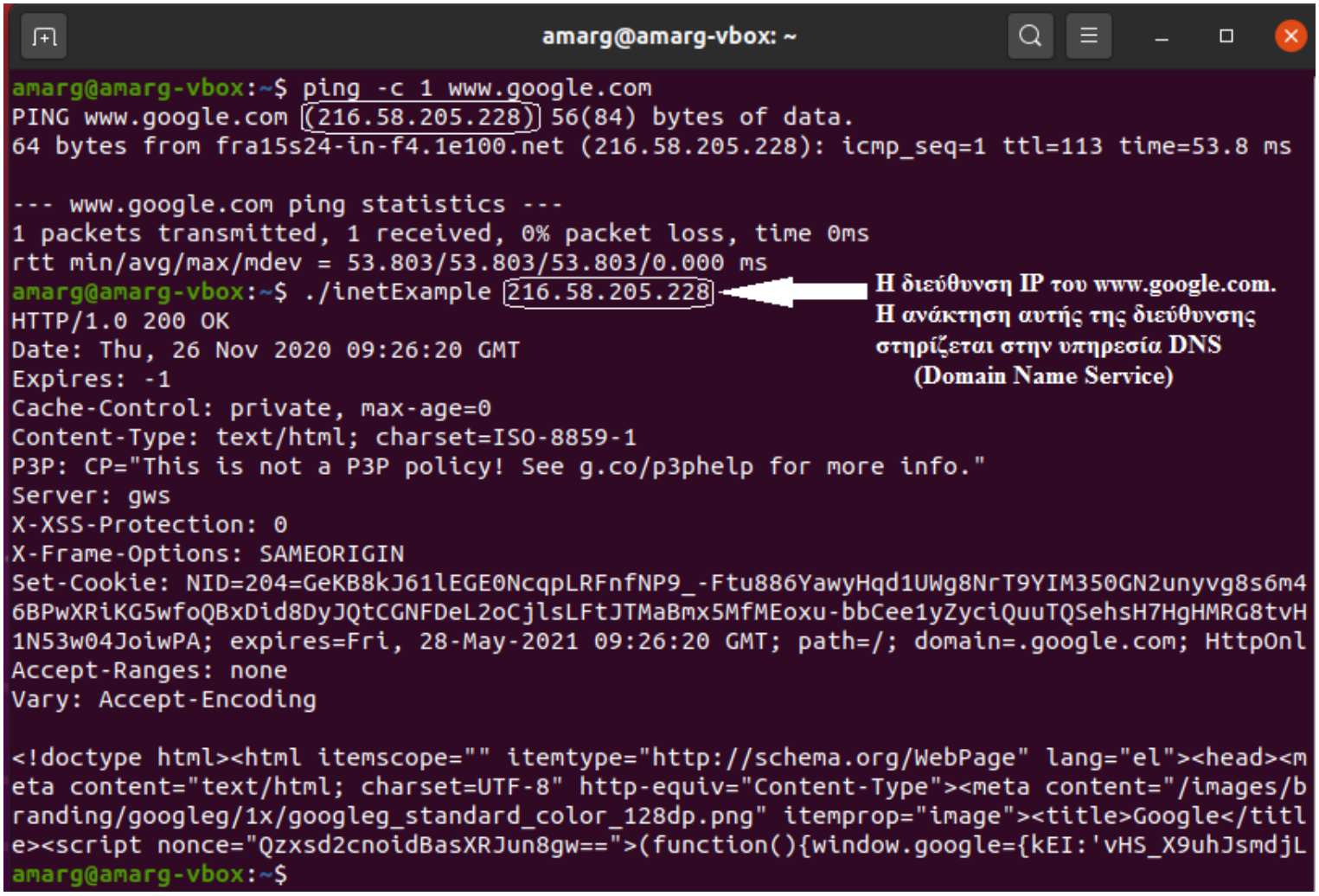
Ο διακομιστής αποκρίνεται στο αίτημα του πελάτη επιτρέποντάς του να διαβάσει δεδομένα κάτι που γίνεται από τον πελάτη καλώντας τη συνάρτηση read. Το πλήθος των δεδομένων που θα διαβάσει ο πελάτης υπαγορεύεται από το μέγεθος της περιοχής μνήμης στην οποία θα αποθηκευτούν τα δεδομένα που διαβάζονται και το οποίο στην προκειμένη περίπτωση είναι ίσο με 1000. Εάν θέλετε να διαβάσετε περισσότερα ή λιγότερα δεδομένα τροποποιήστε την τιμή αυτού του αριθμού.

```
nread = read (fd_skt, buf, sizeof(buf));
```

Η εφαρμογή ολοκληρώνεται με την εκτύπωση των δεδομένων από τον πελάτη στην οθόνη του τερματικού του, κάτι που δίνετε καλώντας τη συνάρτηση write με περιγραφέα αρχείου τον *STDOUT_FILENO που παραπέμπει στην προεπιλεγμένη έξοδο. Κατά τον τερματισμό της διαδικασίας κλείνει και το socket επικοινωνίας με το διακομιστή.*

```
(void) write (STDOUT_FILENO, buf, nread);
close(fd_skt);
exit(0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Επειδή αυτή η εφαρμογή δέχεται ως όρισμα αριθμητική και όχι συμβολική διεύθυνση IP, αρχικά καλούμε την εντολή ping με όρισμα τη συμβολική διεύθυνση την οποία θέλουμε να χρησιμοποιήσουμε, έτσι ώστε να ανακτήσουμε την αριθμητική διεύθυνση IP του απομακρυσμένου διακομιστή.



Παράδειγμα 5. Ο χρήστης καλεί την ping με μία συμβολική διεύθυνση για να ανακτήσει την πραγματική IP διεύθυνση την οποία στη συνέχεια καταχωρεί ως όρισμα στην εφαρμογή addrInfo. (αρχείο addrInfo.c).

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

Αυτό το παράδειγμα επιδεικνύει τον τρόπο χρήσης της εντολής getaddrInfo. Η εντολή δέχεται ως όρισμα (1) μία διεύθυνση IP (στη μορφή xxx.xxx.xxx.xxx), (2) μία υπηρεσία (η οποία περιγράφεται από έναν αριθμό θύρας) και (3) έναν δείκτη σε μία δομή *addrinfo* (στο παράδειγμά μας αυτό το όρισμα είναι το hint) η οποία καθορίζει τα κριτήρια επιλογής των πληροφοριών που επιστρέφονται. Η συνάρτηση επιστρέφει μέσω του τέταρτου ορίσματος που δέχεται (αυτό το όρισμα στο παράδειγμά μας είναι το όρισμα infop) μία ή περισσότερες δομές τύπου *addrinfo* η καθεμία των οποίων περιέχει μία διεύθυνση IP που να μπορεί να χρησιμοποιηθεί σε μία κλήση της bind ή της connect.

Επειδή η συνάρτηση δέχεται ως όρισμα αριθμητική διεύθυνση IP της μορφής xxx.xxx.xxx.xxx ενώ οι χρήστες συνήθως χρησιμοποιούν για λόγους ευκολίας συμβολικές διευθύνσεις (για παράδειγμα, www.uth.gr), αρχικά όπως θα δούμε κατά την εκτέλεση του προγράμματος θα καλέσουμε την εντολή ping όπως κάναμε και στο προηγούμενο παράδειγμα, η οποία για την συμβολική διεύθυνση που θέλουμε να χρησιμοποιήσουμε, θα μας επιστρέψει την πραγματική αριθμητική διεύθυνση η οποία θα διαβιβαστεί στην εφαρμογή ως όρισμα από τη γραμμή εντολών. Αντιλαμβάνεστε πως πλέον έχουμε

φύγει από τις εφαρμογές client-server (αν και θα επιστρέψουμε στη συνέχεια), αφού αυτό το παράδειγμα επιδεικνύει απλά τον τρόπο χρήσης μιας συνάρτησης ανάκτησης πληροφοριών. Το TCP/IP προσφέρει πάρα πολλές τέτοιες συναρτήσεις ανάκτησης δικτυακών πληροφοριών και η getaddressinfo είναι απλά μία από αυτές.

```
int main (int argc, char * argv[]) {
   int retVal; char err[50];
   struct addrinfo *infop = NULL, hint;
```

Σε αυτό το σημείο αρχικοποιούμε τη δομή hint η οποία θα αποτελέσει το τρίτο όρισμα της εντολής. Αρχικά θέτουμε σε όλα τα πεδία της την τιμή 0 και στη συνέχεια εκχωρούμε στα κατάλληλα πεδία τις τιμές AF_INET και SOCK_STREAM για να ορίσουμε πως ενδιαφέρομαστε για διευθύνσεις που χρησιμοποιούν το πρωτόκολλο IPV4 και TCP Sockets και κατά συνέπεια προσφέρουν επικοινωνία με σύνδεση.

```
   memset(&hint, 0, sizeof(hint));
   hint.ai_family = AF_INET;
   hint.ai_socktype = SOCK_STREAM;
```

Εδώ πραγματοποιούμε το συνήθη έλεγχο ορθής χρήσης αφού για να εκτελεστεί ο κώδικας θα πρέπει να δώσουμε από τη γραμμή εντολών την διεύθυνση του απομακρυσμένου δικτυακού τόπου. Εάν λοιπόν δεν ισχύει η συνθήκη argc = 2, τότε το πρόγραμμα τερματίζεται εκτυπώνονται στο κατάλληλο ενημερωτικό μήνυμα χρήσης

```
 if (argc!=2) {
      printf ("Usage: inetExample xxx.xxx.xxx.xxx\n");
      exit (-1); }
```

Στο σημείο αυτό καλείται η συνάρτηση με πρώτο όρισμα την αριθμητική διεύθυνση IP που καταχώρησε ο χρήστης, δεύτερο όρισμα τον αριθμό θύρας 80 (που παραπέμπει σε πρωτόκολλο HTTP) – προσέξτε πως ο αριθμός μπαίνει σε εισαγωγικά γιατί το δεύτερο όρισμα της συνάρτησης δηλώνεται ως const char *service και κατά συνέπεια δεν είναι αριθμός αλλά συμβολοσειρά και τρίτο όρισμα τη δομή hint που αρχικοποιήσαμε παραπάνω. Η συνάρτηση επιστρέφει τη ζητούμενη πληροφορία στο τέταρτο όρισμα infop.

```
retVal = getaddrinfo(argv[1], "80", &hint, &infop);
```

Εάν η επιστρεφόμενη τιμή της συνάρτησης δεν είναι μηδέν, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα και ως εκ τούτου το πρόγραμμα τερματίζεται εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

```
if (retVal!=0) {
      strcpy (err,gai_strerror(retVal));
      printf ("Error found ==> %s\n", err);
      exit (-1); }
```

Εάν η εντολή επιστρέψει με επιτυχία, εκτυπώνει τις τιμές των πληροφοριών που ανέκτησε. Επειδή στη γενική περίπτωση η εντολή μπορεί να επιστρέψει περισσότερες από μία δομές addrinfo, η εκτύπωση των πληροφοριών γίνεται μέσα από έναν επαναληπτικό βρόχο στον οποίο σε κάθε κύκλο επανάληψης εκτυπώνεται το περιεχόμενο της καθεμιάς από αυτές τις δομές. Αυτές οι δομές αποτελούν τους κόμβους μιας λίστας με τον επόμενο κόμβο που προσπελαύνεται να προσδιορίζεται από τον δείκτη next και τον επαναληπτικό βρόχο να εκτελείται για όσο χρονικό διάστημα ο δείκτης next δεν έχει τιμή NULL και κατά συνέπεια υπάρχει επόμενη δομή για εκτύπωση.

```
for ( ; infop != NULL; infop = infop->ai_next) {
      struct sockaddr_in *sa = (struct sockaddr_in *)infop->ai_addr;
      printf("%s port: %d protocol: %d\n", inet_ntoa(sa->sin_addr),
           ntohs(sa->sin_port), infop->ai_protocol); }
   exit(0);}
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Επειδή αυτή η εφαρμογή δέχεται ως όρισμα αριθμητική και όχι συμβολική διεύθυνση IP, αρχικά όπως και πριν καλούμε την εντολή ping με όρισμα τη συμβολική διεύθυνση την οποία θέλουμε να χρησιμοποιήσουμε, έτσι ώστε να ανακτήσουμε την αριθμητική διεύθυνση IP του απομακρυσμένου υπολογιστή.

```
amarg@amarg-vbox:~$ ping www.uth.gr
PING zeus.uth.gr (194.177.200.17) 56(84) bytes of data.
64 bytes from zeus.uth.gr (194.177.200.17): icmp_seq=1 ttl=53 time=39.0 ms
amarg@amarg-vbox:~$ ./addrInfo 194.177.200.17
194.177.200.17 port: 80 protocol: 6
amarg@amarg-vbox:~$
```

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.
```

Παράδειγμα 7. 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
```

ΕΡΓΑΣΤΗΡΙΟ 9

Διαδιεργασιακή Επικοινωνία

Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τους διάφορους τρόπους επικοινωνίας μεταξύ διεργασιών.

Παράδειγμα 1. 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
```

Παράδειγμα 2. 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.

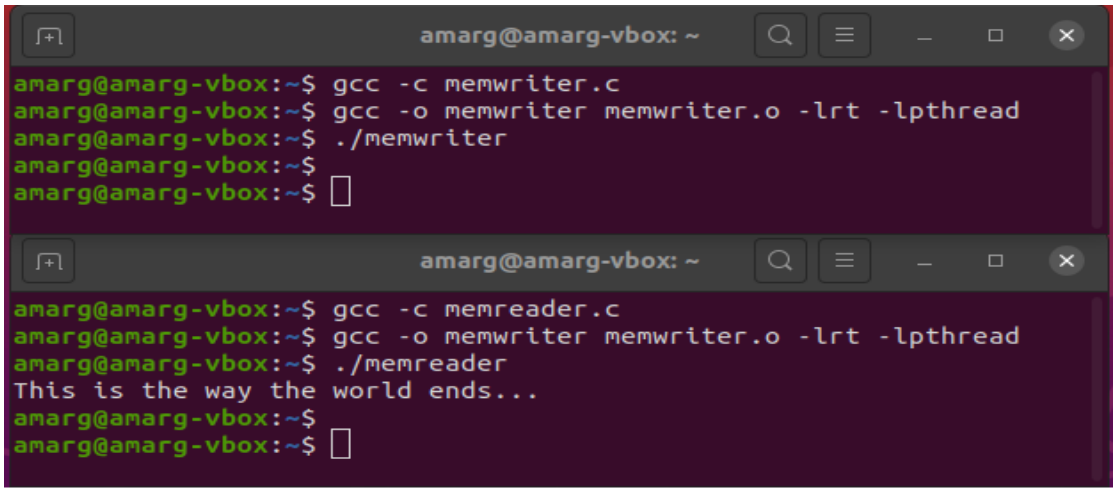
```
  if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
     int i;
     for (i = 0; i < strlen (MemContents); i++)
```

```
        write(STDOUT_FILENO, memptr + i, 1);
    sem_post(semptr); }
```

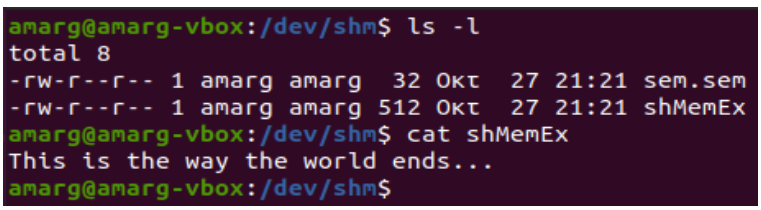
Η διαδικασία ολοκληρώνεται καταργώντας την απεικόνιση μνήμης και κλείνοντας τα εμπλεκόμενα αρχεία και τις δομές που χρησιμοποιήθηκαν (σημαφόροι).

```
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.



Παράδειγμα 3. Επικοινωνία διεργασιών με χρήση ουράς μηνυμάτων (αρχείο mqExample.c).

```
#include <mqueue.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <string.h>

#define MSG_SIZE 16384
```

Σε αυτή την εφαρμογή, επιδεικνύεται η χρήση μιας ουράς μηνυμάτων (message queue) η οποία υλοποιείται ως ένα αρχείο του σκληρού δίσκου και η οποία έχει τη δυνατότητα να φιλοξενήσει ένα συγκεκριμένο αριθμό μηνυμάτων. Για λόγους απλότητας, δεν χρησιμοποιούμε δύο διεργασίες, αλλά μόνο μία διεργασία η οποία εκτελείται δύο φορές: την πρώτη φορά αποθηκεύει κάποια μηνύματα στην ουρά μηνυμάτων, ενώ τη δεύτερη φορά διαβάζει τα μηνύματα που είχαν αποθηκευτεί κατά την πρώτη εκτέλεση και γράφει άλλα 10 μηνύματα. Βέβαια, σε μία πραγματική εφαρμογή, αυτή η διαδικασία θα μπορούσε να υλοποιηθεί χρησιμοποιώντας δύο διεργασίες, με τη δεύτερη διεργασία να ανοίγει το αρχείο της ουράς μηνυμάτων που έχει δημιουργήσει η πρώτη διεργασία και να διαβάζει τα μηνύματα που έχουν αποθηκευτεί από αυτή και

γράφοντας με τη σειρά της τα δικά της μηνύματα, τα οποία στη συνέχεια θα μπορούσαν να διαβαστούν από την πρώτη διεργασία, κ.ο.κ. Ωστόσο εδώ, για λόγους απλότητας, όπως σχολιάσαμε, αυτή η διαδικασία πραγματοποιείται από μία και μοναδική διεργασία η οποία εκτελείται δύο φορές.

```
void handler (int sig_num) { printf ("Received sig %d.\n", sig_num); }
```
Ο χειριστής του σήματος SIGUSR1

```
int main (int argc, char * argv []) {
  struct mq_attr attr, old_attr;
  struct sigevent sigevent;
  mqd_t mqdes, mqdes2;
  char * message = "Hello world";
  char buf [MSG_SIZE];
  unsigned int prio=0, i;
```

Αρχικά δημιουργούμε μία νέα ουρά μηνυμάτων καλώντας τη συνάρτηση mq_open. Τα μηνύματα αυτής της ουράς αποθηκεύονται στο αρχείο του δίσκου mqueue1 το οποίο δημιουργείται στον κατάλογο /dev/mqueue. Το αρχείο δημιουργείται για ανάγνωση και εγγραφή (O_RDWR | O_CREAT) με δικαιώματα 0664 ενώ επειδή το τελευταίο όρισμα της συνάρτησης είναι NULL χρησιμοποιούνται για την ουρά προεπιλεγμένες ιδιότητες.

```
  mqdes = mq_open ("/mqueue1", O_RDWR | O_CREAT, 0664, NULL);
```

Μετά τη δημιουργία της ουράς καλούμε τη συνάρτηση mq_getattr για να εκτυπώσουμε τις τιμές των (προεπιλεγμένων) ιδιοτήτων της και πιο συγκεκριμένα το μέγιστο πλήθος των μηνυμάτων που μπορεί να φιλοξενήσει, το μήκος του κάθε μηνύματος καθώς και το πλήθος των μηνυμάτων που βρίσκονται στην ώρα αυτή τη χρονική στιγμή.

```
  mq_getattr (mqdes, &attr);
  printf ("Max number of messages on the queue ==> %ld messages.\n", attr.mq_maxmsg);
  printf ("Size of messages on the queue ==> %ld bytes.\n", attr.mq_msgsize);
  printf ("%ld messages are currently on the queue.\n", attr.mq_curmsgs);
```

Εάν στην ουρά μηνυμάτων υπάρχουν μηνύματα για παραλαβή, αυτά παραλαμβάνονται μέσω μίας επαναληπτικής διαδικασίας, η οποία καλεί τη συνάρτηση mq_receive για να παραλάβει ένα μήνυμα σε κάθε κύκλο επανάληψης. Το όρισμα *O_NONBLOCK χρησιμοποιείται έτσι ώστε η συνάρτηση να μην μπλοκάρει εάν δεν βρει μηνύματα προς παραλαβή αλλά αντίθετα, να επιστρέψει αμέσως, με κωδικό σφάλματος EAGAIN. Αυτός είναι ο μοναδικός κωδικός σφάλματος ο οποίος επιτρέπεται να παραληφθεί, ενώ αν επιστραφεί οποιοσδήποτε άλλος κωδικός σφάλματος, αυτό σημαίνει πως έλαβε χώρα κάποιο άλλο σφάλμα και η εφαρμογή τερματίζει τη λειτουργία της. Λαμβάνοντας υπόψη πως η εφαρμογή θα εκτελεστεί δύο φορές και έτσι ώστε την πρώτη φορά μόνο να αποστείλει μηνύματα στην ουρά μηνυμάτων, ενώ τη δεύτερη φορά να διαβάσει τα μηνύματα που αποθήκευσε την πρώτη φορά, είναι προφανές πως ο κώδικας που βρίσκεται στο block if (attr.mq_curmsgs != 0) θα εκτελεστεί μόνο τη δεύτερη , διότι μόνο τότε θα βρει μηνύματα για να παραλάβει.*

```
  if (attr.mq_curmsgs != 0) {
    attr.mq_flags = O_NONBLOCK;
    mq_setattr (mqdes, &attr, &old_attr);
    while (mq_receive (mqdes, &buf[0], MSG_SIZE, &prio) != -1)
        printf ("Received a message with priority %d.\n", prio);
    if (errno != EAGAIN) { perror ("mq_receive()"); exit (EXIT_FAILURE); }
    mq_setattr (mqdes, &old_attr, 0); }
```

Μετά την τοποθέτηση στην κενή ουρά του πρώτου μηνύματος, στάλθηκε στη διεργασία το σήμα SIGUSR1 με τιμή 10 επειδή η διεργασία ζήτησε να ειδοποιηθεί όταν συμβεί κάτι τέτοιο.

```
  signal (SIGUSR1, handler);
  sigevent.sigev_signo = SIGUSR1;
  if (mq_notify (mqdes, &sigevent) == -1) {
    if (errno == EBUSY)
      printf ("Another process has registered for notification.\n");
      exit (EXIT_FAILURE); }
```

Ανεξάρτητα από το εάν θα λάβει χώρα η διαδικασία ανάγνωσης των μηνυμάτων της ουράς ή όχι (αυτό εξαρτάται από το εάν η εφαρμογή εκτελείται για πρώτη ή για δεύτερη φορά), σε αυτό το σημείο η εφαρμογή μέσω μιας επαναληπτικής διαδικασίας και χρησιμοποιώντας τη συνάρτηση mq_send αποστέλλει στην ουρά μηνυμάτων δέκα νέα μηνύματα. Προκειμένου να επιδείξουμε το γεγονός πως οι ουρές μηνυμάτων παρά το γεγονός πως αποτελούν δομές FIFO υποστηρίζουν τιμές προτεραιότητας και κατά συνέπεια το μήνυμα που διαβάζεται πρώτο είναι το παλαιότερο μήνυμα (δηλαδή αυτό που στάλθηκε πρώτο) αλλά με τη μέγιστη τιμή προτεραιότητας, τα δέκα μηνύματα που αποθηκεύονται στο αρχείο της ουράς, έχουν τιμές προτεραιότητας οι οποίες αυξάνονται κατά 5.

```
for (i= 0; i < attr.mq_maxmsg; i++) {
    printf ("Writing a message with priority %d.\n", prio);
    if (mq_send (mqdes, message, strlen(message)+1, prio) == -1) perror ("mq_send()");
    prio += 5;}
```

Μετά την ολοκλήρωση της διαδικασίας αποστολής, η διεργασία τερματίζεται κλείνοντας τον περιγραφέα του αρχείου της ουράς μηνυμάτων.

```
mq_close (mqdes);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής, ακολουθεί στη συνέχεια. Παρατηρήστε πως τα μηνύματα έχουν διαταχθεί ανάλογα με την τιμή της προτεραιότητάς τους και κατά συνέπεια τα μηνύματα παρελήφθησαν έτσι ώστε αυτά που χαρακτηρίζονται από μεγαλύτερη τιμή προτεραιότητας να παραληφθούν πρώτα.

```
amarg@amarg-vbox:~$ ./mqExample1
Max number of messages on the queue ==> 10 messages.
Size of messages on the queue ==> 8192 bytes.
10 messages are currently on the queue.
Received a message with priority 45.
Received a message with priority 40.
Received a message with priority 35.
Received a message with priority 30.
Received a message with priority 25.
Received a message with priority 20.
Received a message with priority 15.
Received a message with priority 10.
Received a message with priority 5.
Received a message with priority 0.
Writing a message with priority 0.
Received sig 10.
Writing a message with priority 5.
Writing a message with priority 10.
Writing a message with priority 15.
Writing a message with priority 20.
Writing a message with priority 25.
Writing a message with priority 30.
Writing a message with priority 35.
Writing a message with priority 40.
Writing a message with priority 45.
amarg@amarg-vbox:~$
```

Αυτή η διαδικασία δημιουργεί στον κατάλογο /dev/mqueue το αρχείο mqueue1 τα περιεχόμενα του οποίου ακολουθούν στη συνέχεια. Το πεδίο QSIZE : 120 σε αυτό το αρχείο υποδηλώνει πως στην ουρά mqueue1 υπάρχουν 120 bytes προς παραλαβή [10 messages x strlen ("Hello world")]= 10 x 12 = 120 τα οποία έχουν παραληφθεί από τον πυρήνα και αναμένουν να διαβαστούν.

```
amarg@amarg-vbox:/dev/mqueue$ ls -l
total 0
-rw-rw-r-- 1 amarg amarg 80 Νοε   1 14:33 mqueue1
amarg@amarg-vbox:/dev/mqueue$ cat mqueue1
QSIZE:120        NOTIFY:0     SIGNO:0     NOTIFY_PID:0

amarg@amarg-vbox:/dev/mqueue$
```