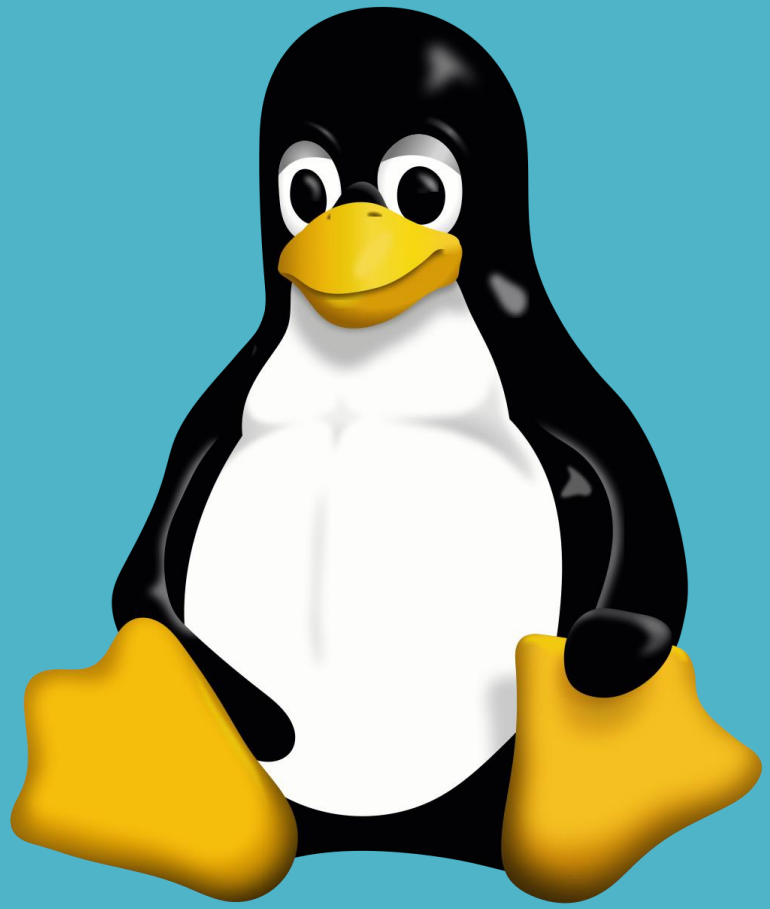
Εισαγωγή στο Linux



Ιστορική αναδρομή

1965, Bell, **General Electric, MULTICS (Multiplexed Information and Computing Service – πολυπλεγμένες υπηρεσίες πληροφοριών και υπολογισμού MIT**) **GE-645**.

1969 Ken Thomson, **UNICS (Uniplexed Information and Computing Service) DEC PDP–7 Assembly**

- **Σύστημα διαχείρισης αρχείων (file system)**
- **Μηχανισμός ελέγχου διεργασιών (process control mechanism)**
- **Εντολές χρήσης αρχείων (file utilities)**
- **Διερμηνευτής εντολών (command interpreter)**.

Το **1970** ο **Brian Kernigham** καθιέρωσε τη λέξη **UNIX** ως το όνομα του νέου λειτουργικού συστήματος (αντικαθιστώντας την αρχική ονομασία **UNICS**).

Ιστορική αναδρομή

1973, Ο Dennis Ritchie, ξαναγράφει από την αρχή το **UNIX**, σε γλώσσα **C**.

Το Unix γίνεται **Portable**

Το λειτουργικό μοιράζεται από την **AT&T** μαζί με τον πηγαίο κώδικα σε γλώσσα **C** ➔ error correction και νέες δυνατότητες.

Μειονέκτημα ➔ πολλές διαφορετικές εκδόσεις του **UNIX** (**HP UNIX**, **DEC UNIX**, **Solaris**, **BSD UNIX**, **UNIX SYSTEM V**, **CSO UNIX**), οι οποίες εν γένει παρουσιάζουν μεταξύ τους αρκετές ασυμβατότητες.

IEEE (Institute of Electrical and Electronic Engineering) ➔ POSIX

1980, η **Microsoft**, **XENIX**, εμπορική έκδοσης για **16-bit μικροεπεξεργαστές**

Ιστορική αναδρομή

1980, Berkeley Univ. **4.1 BSD**

- **Εφαρμογές καταμερισμού χρόνου (time sharing applications).**
- **Μεγαλύτερος χώρος διευθύνσεων (address space),**
- **Σύστημα εικονικής μνήμης (virtual memory)**
- **Ενισχυμένο σύστημα διαχείρισης αρχείων**
- **Υποστήριξη τοπικών δικτύων**
- **Εξελιγμένος διερμηνευτής εντολών.**

1982, **UNIX SYSTEM III (AT&T)**
Απομακρυσμένη επεξεργασία (remote job entry),
Σύστημα ελέγχου πηγαίου κώδικα (Source Code Control System)

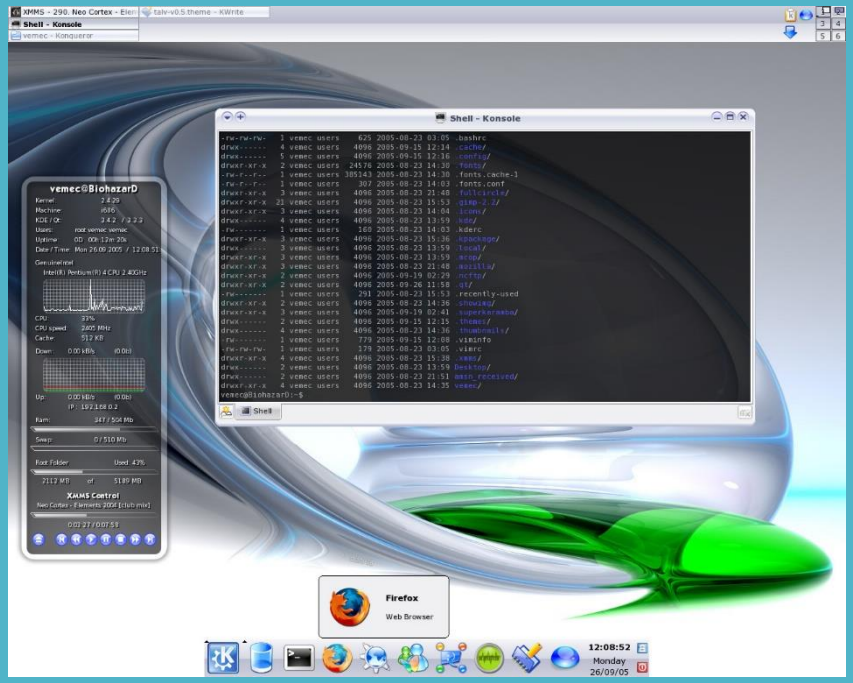
UNIX SYSTEM V (Interprocess communication).

Sun Microsystems ➔ SunOS

1991 Linux, Linus Torvalds, πυρήνας UNIX (UNIX kernel)
1994 Πρώτη έκδοση

Πρόσβαση σε Linux

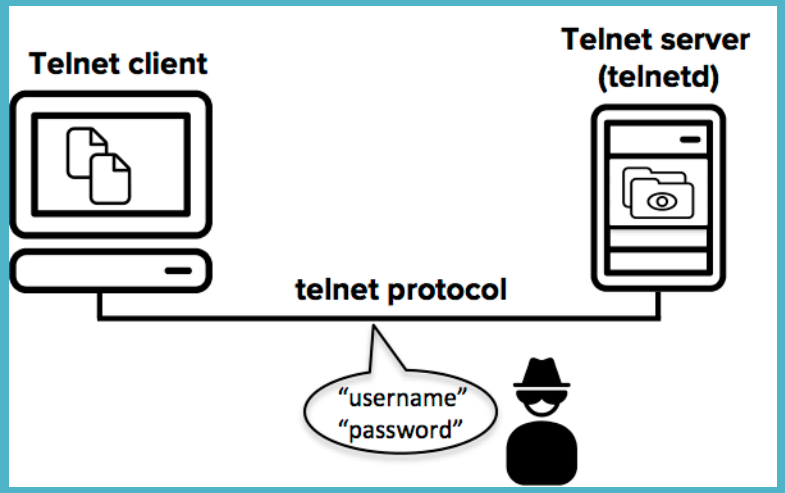
Linux Box (υπολογιστής με εγκατεστημένο μόνο Linux)

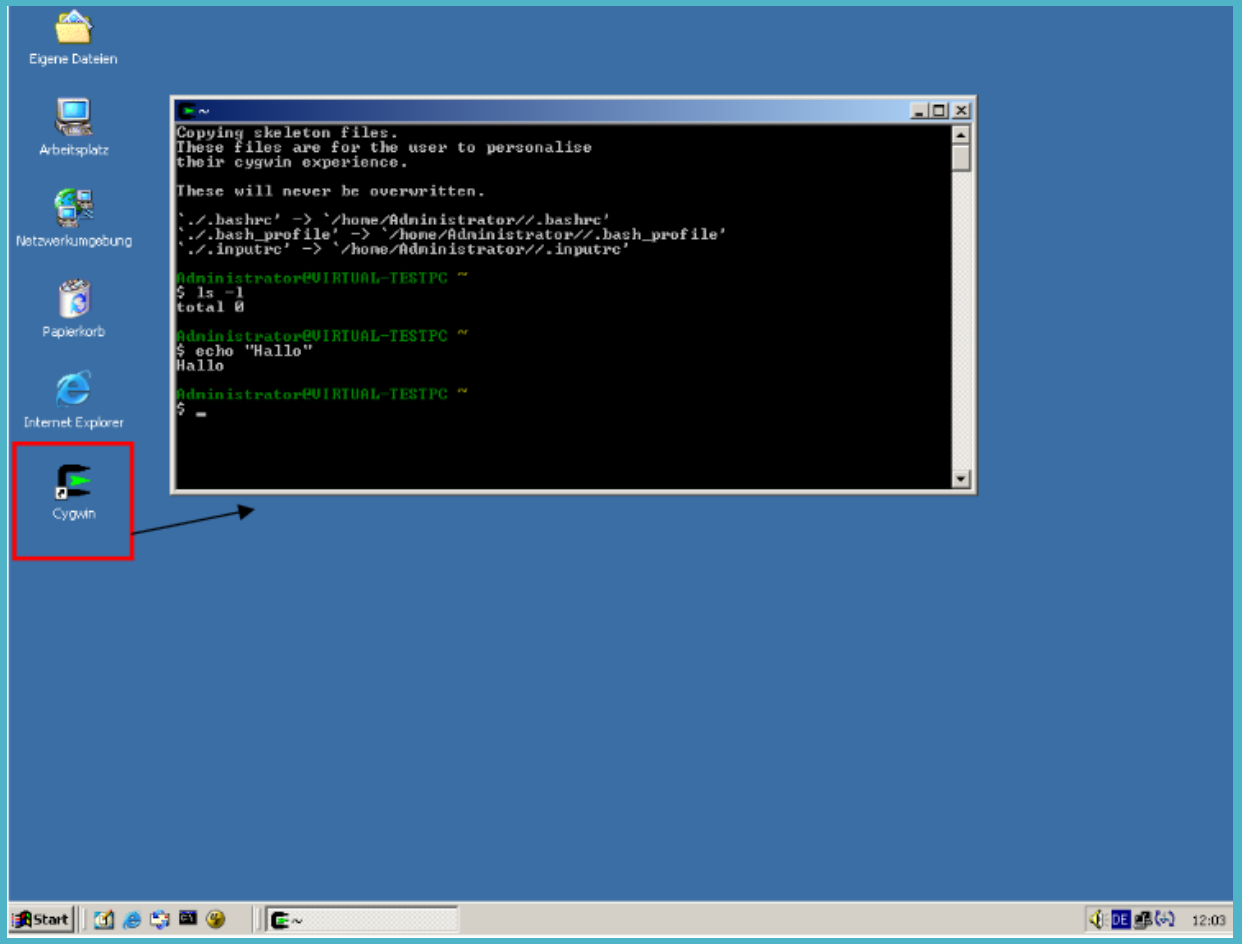


Windows και Linux στον ίδιο υπολογιστή με Boot Manager

Πρόσβαση σε Linux

Σύνδεση σε απομακρυσμένο υπολογιστή με Linux

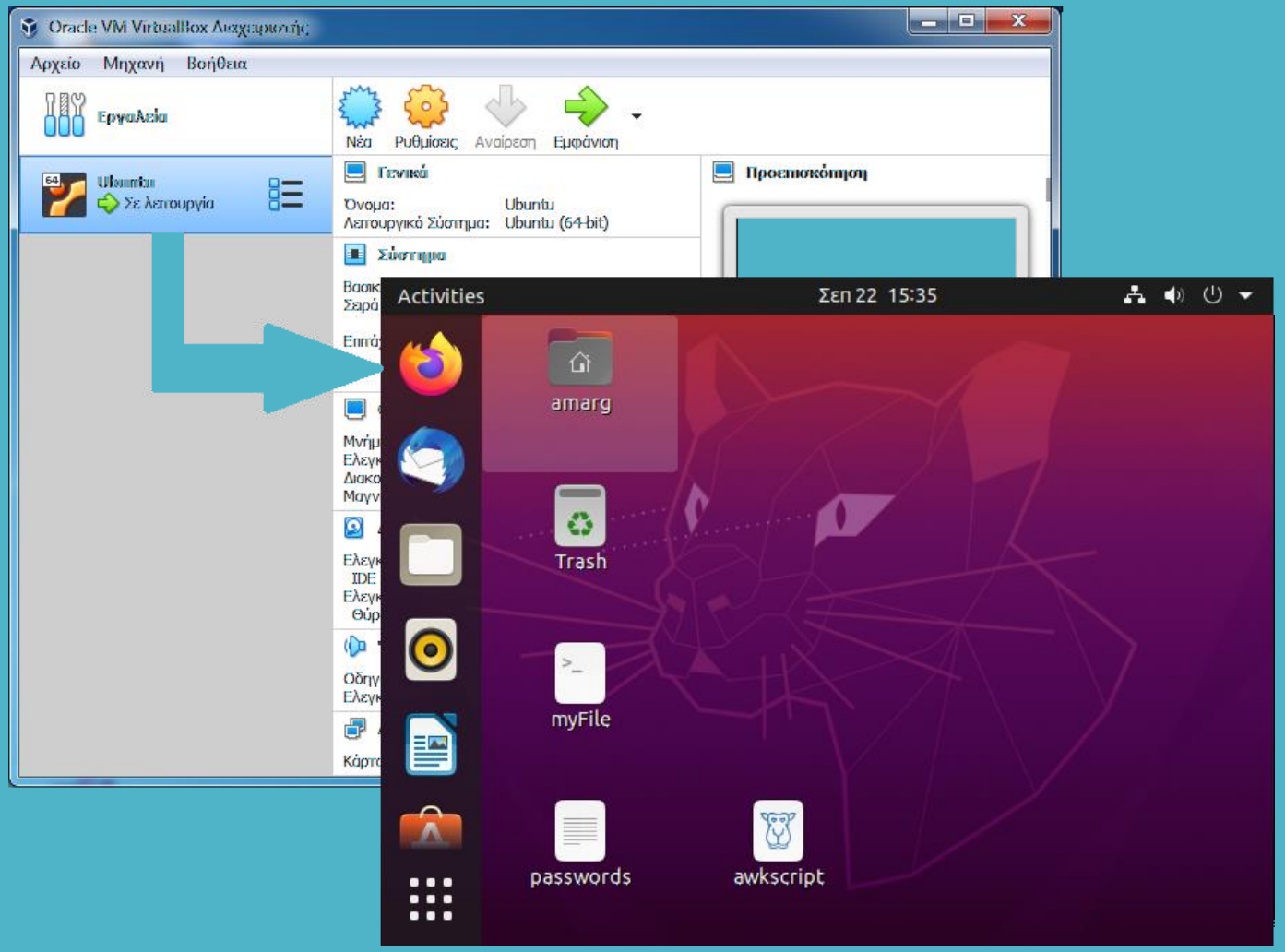




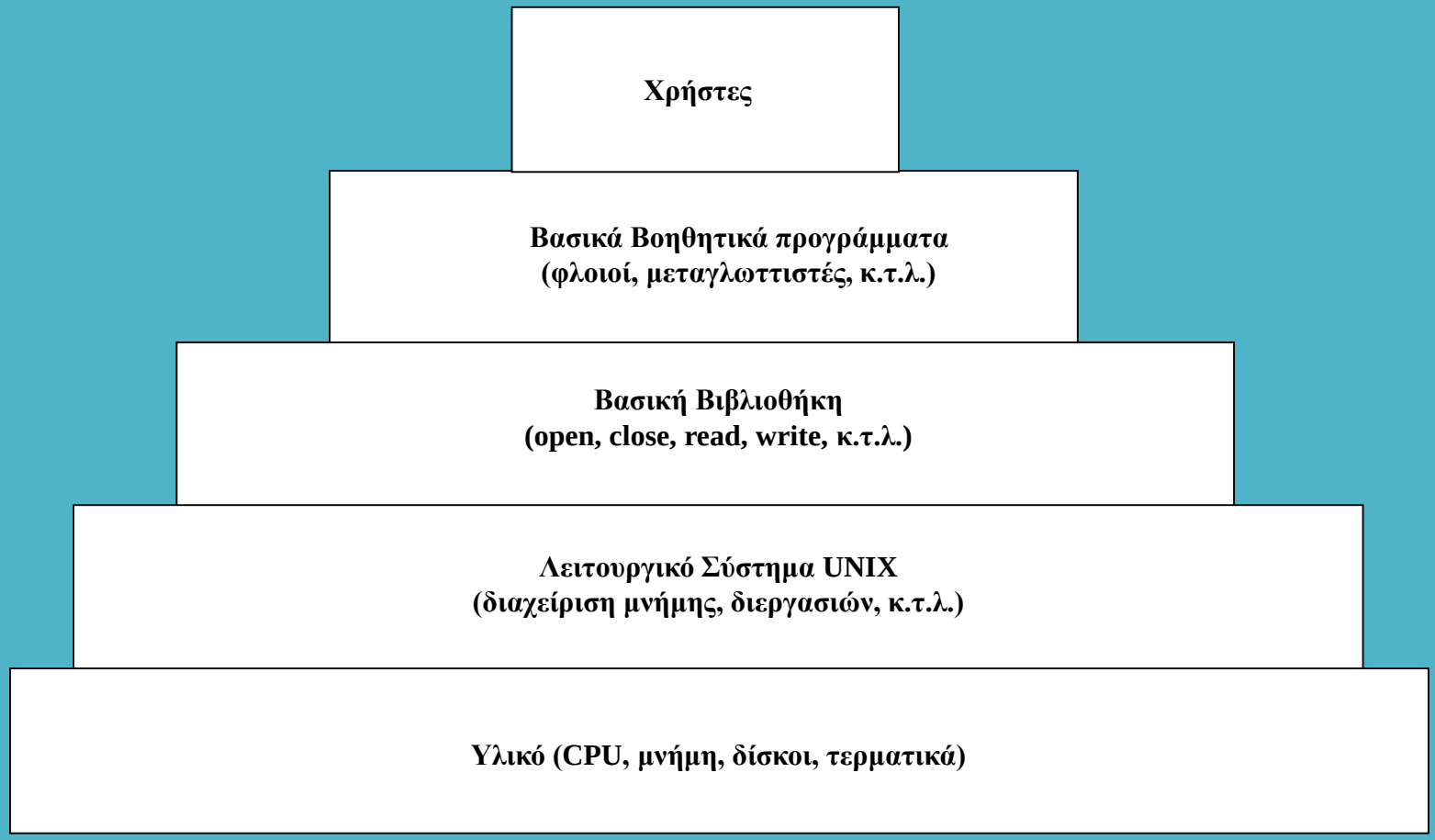
Cygwin

Πρόσβαση σε Linux

Εικονική μηχανή με Linux σε Virtual Box / VMWARE



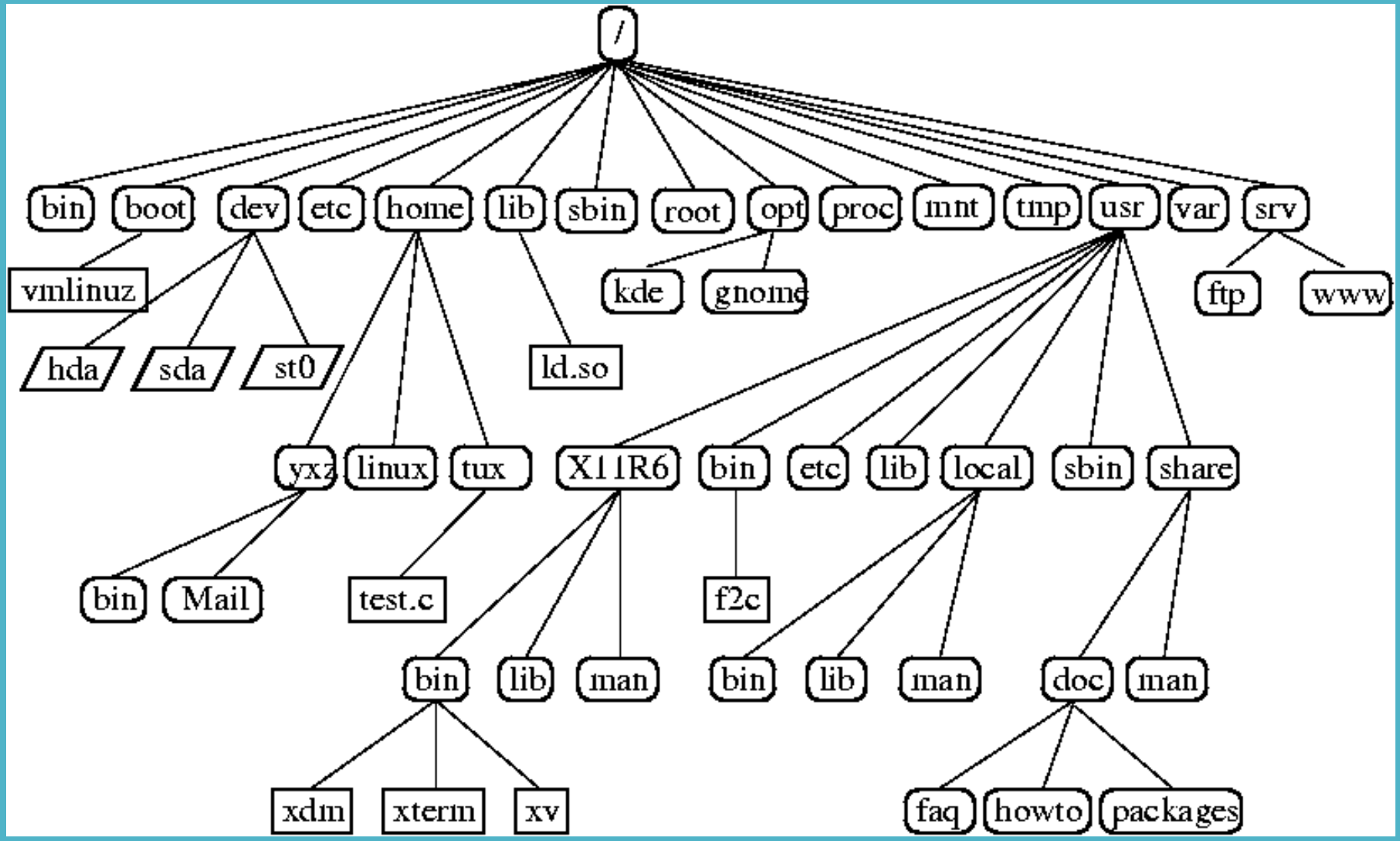
Δομή πυραμίδας



Τύποι διαχείρισης

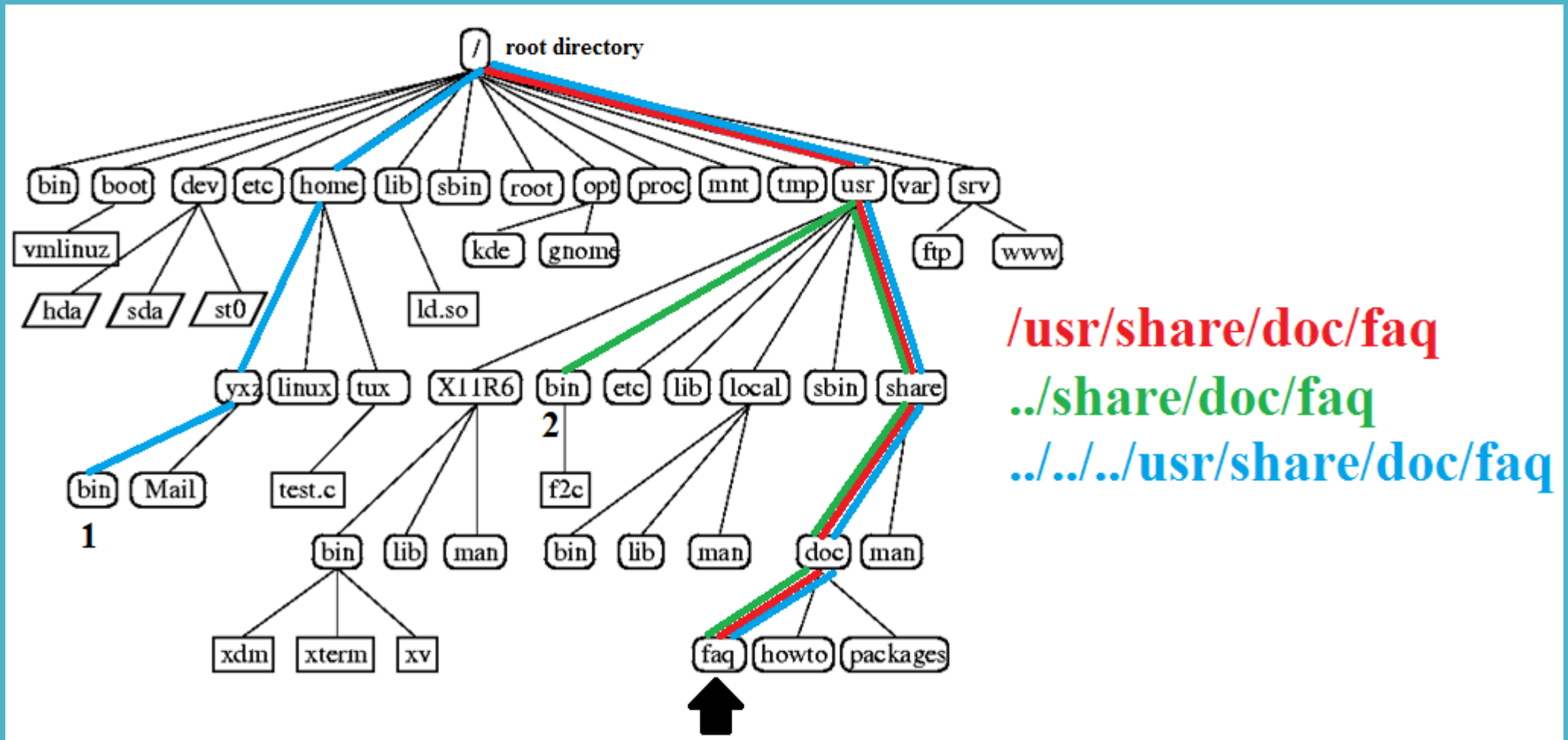
- Διαχείριση επεξεργαστή (διεργασίες, νήματα, χρονοπρογραμματισμός)
- Διαχείριση μνήμης (κεντρική, ενδιάμεση και ιδεατή μνήμη)
- Διαχείριση αρχείων και καταλόγων (δημιουργία, διαγραφή, δικαιώματα)
- Διαχείριση περιφερειακών (προσάρτηση και διαμόρφωση διατάξεων)
- Διαχείριση εκτυπώσεων (ουρές εκτύπωσης)
- Διαχείριση χρηστών (προσθήκη και διαγραφή χρήστη, δικαιώματα πρόσβασης)
- Διαχείριση δικτύου (απομακρυσμένη πρόσβαση, ασφάλεια)

Ιεραρχική δομή καταλόγων



Σχετική και απόλυτη διαδρομή

Σχετική διαδρομή → εξαρτάται από το σημείο εκκίνησης
Απόλυτη διαδρομή → ξεκινά πάντοτε από το root directory

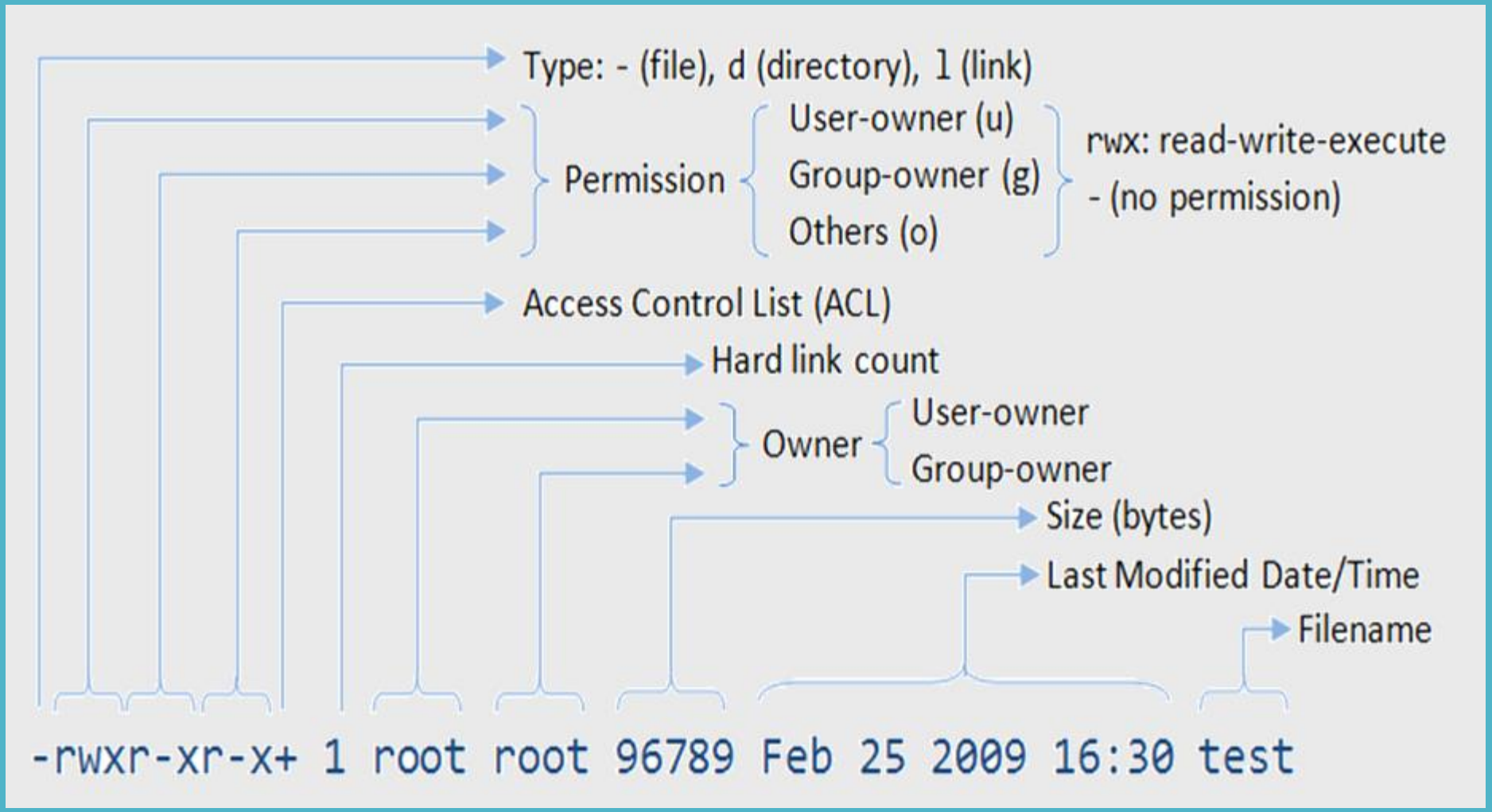


Δικαιώματα πρόσβασης

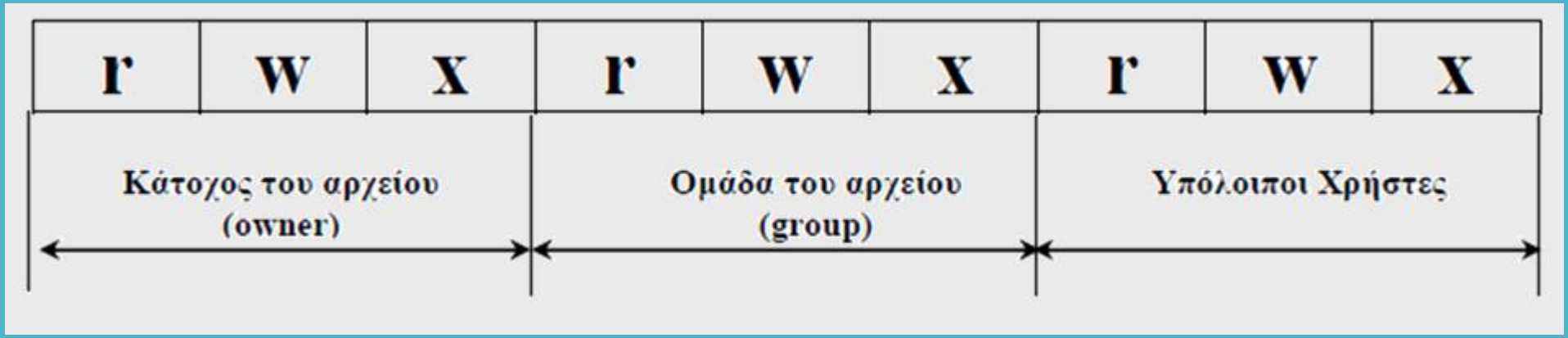
```
drwxr-xr-x 2 amarg amarg     4096 Σεπ  19 20:09 Downloads
-rw-rw-r-- 1 amarg amarg     1521 Σεπ  20 19:58 file1.doc
-rw-rw-r-- 1 amarg amarg       44 Σεπ  20 19:59 file1.txt
-rw-rw-r-- 1 amarg amarg      178 Σεπ  20 19:59 file1.zip
-rw-rw-r-- 1 amarg amarg  7959821 Σεπ  20 19:58 file2.doc
-rw-rw-r-- 1 amarg amarg       90 Σεπ  20 19:59 file2.txt
-rw-rw-r-- 1 amarg amarg      129 Σεπ  20 20:00 file2.zip
-rw-rw-r-- 1 amarg amarg     2797 Σεπ  20 19:59 file3.doc
-rw-rw-r-- 1 amarg amarg    13600 Σεπ  20 20:00 file3.zip
-rw-rw-r-- 1 amarg amarg      550 Σεπ  20 19:59 file4.doc
-rwxrwxr-x 1 amarg amarg      369 Σεπ  22 09:43 isEmpty
-rwxrwxr-x 1 amarg amarg      200 Σεπ  22 09:10 lex
drwxr-xr-x 2 amarg amarg     4096 Σεπ  19 20:09 Music
drwxr-xr-x 2 amarg amarg     4096 Σεπ  22 15:33 Pictures
drwxr-xr-x 2 amarg amarg     4096 Σεπ  19 20:09 Public
-rw-rw-r-- 1 amarg amarg      262 Σεπ  22 10:08 results
-rwxrwxr-x 1 amarg amarg       80 Σεπ  22 09:12 rfile
drwxrwxr-x 2 amarg amarg     4096 Σεπ  22 09:23 S
drwxrwx--- 1 root  vboxsf       0 Σεπ  21 10:30 shared
drwxrwxr-x 2 amarg amarg     4096 Σεπ  19 23:18 Shared
drwxr-xr-x 2 amarg amarg     4096 Σεπ  19 20:09 Templates
-rwxrwxr-x 1 amarg amarg       65 Σεπ  20 17:02 tlatin
drwxr-xr-x 2 amarg amarg     4096 Σεπ  19 20:09 Videos
```

Δικαιώματα πρόσβασης | κάτοχος | ομάδα | μέγεθος | ημερομηνία / ώρα | όνομα

Δικαιώματα πρόσβασης



Δικαιώματα πρόσβασης



Παραδείγματα

r w x - - - - - -

r w – r w – r - -

r w – r w – r w –

r – - r - - r - -

Δικαιώματα πρόσβασης

Μετατροπή μάσκας δικαιωμάτων στο οκταδικό σύστημα

Σε κάθε τριάδα δικαιωμάτων αντικαθιστούμε κάθε γράμμα (r,w,x) με 1 και κάθε παύλα με 0. Με τον τρόπο αυτό προκύπτει ένας δυαδικός αριθμός των τριών bits ο οποίος μετατρέπεται στο **οκταδικό** σύστημα

r	-	x	r	-	-	r	-	x
1	0	1	1	0	0	1	0	1
101 = 5			100 = 4			101 = 5		

Επομένως r-xr—r-x → 544

Με τον ίδιο τρόπο

rwxr-xr-x → 111101101 → 755
rw-rw-rw- → 110110110 → 666
rwxrwxrwx → 111111111 → 777

Δικαιώματα πρόσβασης

Αντίστροφα

756 → 111 101 110 → r w x r – x r w –
644 → 110 100 100 → r w x r - - r - -
653 → 110 101 011 → r w – r – x - w x

Σε κάθε τριάδα

XYZ

X → READ (R) Y → WRITE (W) Z → EXECUTE (X)

Default permissions

Για αρχεία 666 → rw-rw-rw-
Για καταλόγους 777 → rwxrwxrwx

Umask = 022

Για αρχεία 666 – 022 = 644 → 110 100 100 → r w – r - - r- -
Για καταλόγους 777 – 022 → 755 → 111 101 101 → r w x r - x r - x

Δικαιώματα πρόσβασης

Μεταβολή δικαιωμάτων αρχείου

chmod 753 myFile

Ως μάσκα δικαιωμάτων του myFile ορίζεται η

753 → 111 101 011 → r w x r - x - w x

Μεταβολή κατόχου αρχείου

chown terry myFile

Μεταβολή ομάδας αρχείου

chgrp users myFile

Διαχείριση καταλόγων

Δημιουργία καταλόγου **mkdir myDir**

Διαγραφή κενού καταλόγου **rmdir myDir**

Εμφάνιση ονόματος τρέχοντος καταλόγου **pwd**

Μετάβαση σε κατάλογο **cd myDir**

Εμφάνιση δεντρικής δομής καταλόγου **tree myDir**

Εμφάνιση περιεχομένων καταλόγου **ls myDir –l –a -R**

Διαχείριση αρχείων

Αντιγραφή αρχείου **cp file1 file 2 (-f, -i)**

Εμφάνιση περιεχομένων αρχείου **cat myFile**

Συνένωση περιεχομένων αρχείου **cat file1 file2**

Μετακίνηση αρχείου **mv file1 file2**

Διαγραφή αρχείου **rm myFile (-f, -i, -r)**

Εύρεση αρχείου **find searchDir**

ΠΑΡΑΔΕΙΓΜΑΤΑ ΕΥΡΕΣΗΣ

find /usr –name readme
find /home – perm 652
find /usr –size 1024
find / -user amarg

find /usr –size +100 –size -500 –name student –perm 755 –printf “%f\t%s\t%m\n”

Διαχείριση αρχείων

Αναζήτηση αρχείου **which startx**

Εμφάνιση απόλυτης διαδρομής **whereis startx**

Αναζήτηση κειμένου σε αρχείο **grep –e Linux readme**

Εμφάνιση γραμμών από την αρχή **head –n 20 readme**

Εμφάνιση γραμμών από το τέλος **tail –n 50 readme**

Εμφάνιση περιεχομένων ανά σελίδα (page down) **more -25 readme**

Εμφάνιση περιεχομένων ανά σελίδα (page down & page up) **less -25 readme**

Καταμέτρηση χαρακτήρων, λέξεων και γραμμών **wc –c –w –l readme**

Δημιουργία κενού αρχείου κειμένου **touch myFile**

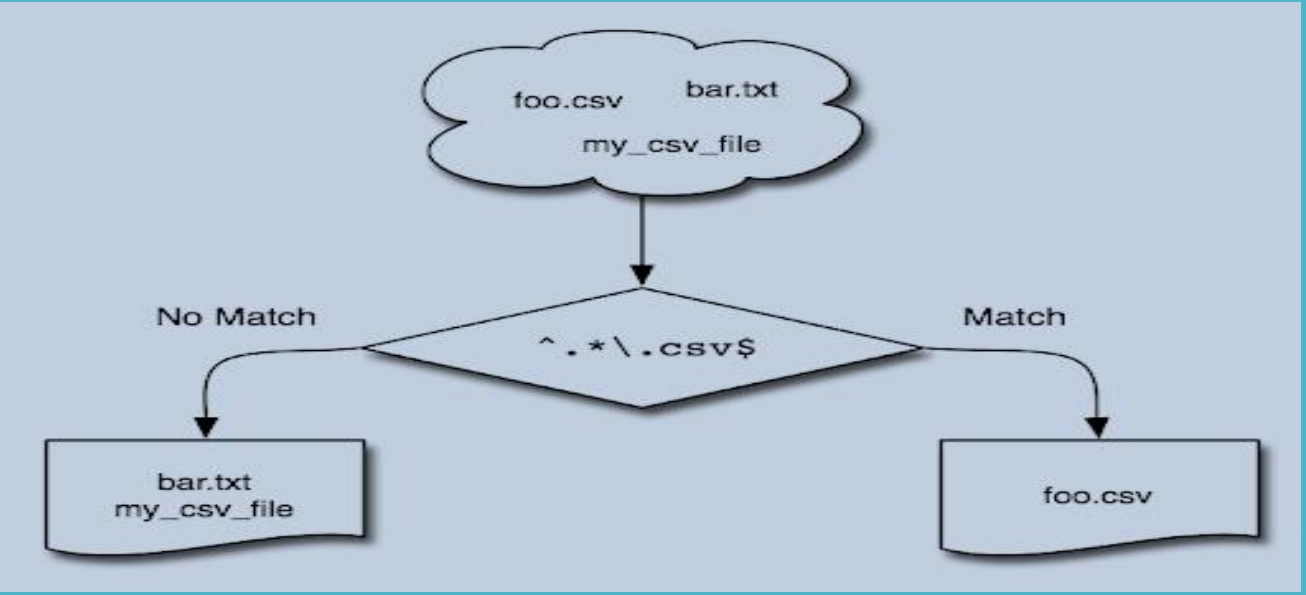
Κανονικές Εκφράσεις

Κανονικές Εκφράσεις

- Επιτρέπουν τον καθορισμό οποιουδήποτε συνδυασμού χαρακτήρων με συμβολικό και συνοπτικό τρόπο. Αποτελούνται από συνδυασμούς χαρακτήρων και μεταχαρακτήρων.

- Η βασική τους χρησιμότητα είναι η αναζήτηση αλφαριθμητικών σε αρχεία κειμένου. Η κάθε λέξη του αρχείου ελέγχεται εάν μπορεί να δημιουργηθεί από την κανονική έκφραση και εάν τα πράγματα είναι όντως έτσι, τότε επιστρέφεται στην έξοδο της εντολής.

- Αποτελούν αναπόσπαστο συστατικό των μεταγλωττιστών και των διερμηνευτών και σχετίζονται με ειδικές δομές που ονομάζονται μηχανές πεπερασμένων καταστάσεων.



Κανονικές Εκφράσεις

- Οι χαρακτήρες που χρησιμοποιούνται στις κανονικές εκφράσεις, περιλαμβάνουν το σύνολο των πεζών και κεφαλαίων γραμμάτων, τα ψηφία, και άλλους συχνά χρησιμοποιούμενους χαρακτήρες όπως είναι οι

~ ‘ ! @ # _ - = : ; /

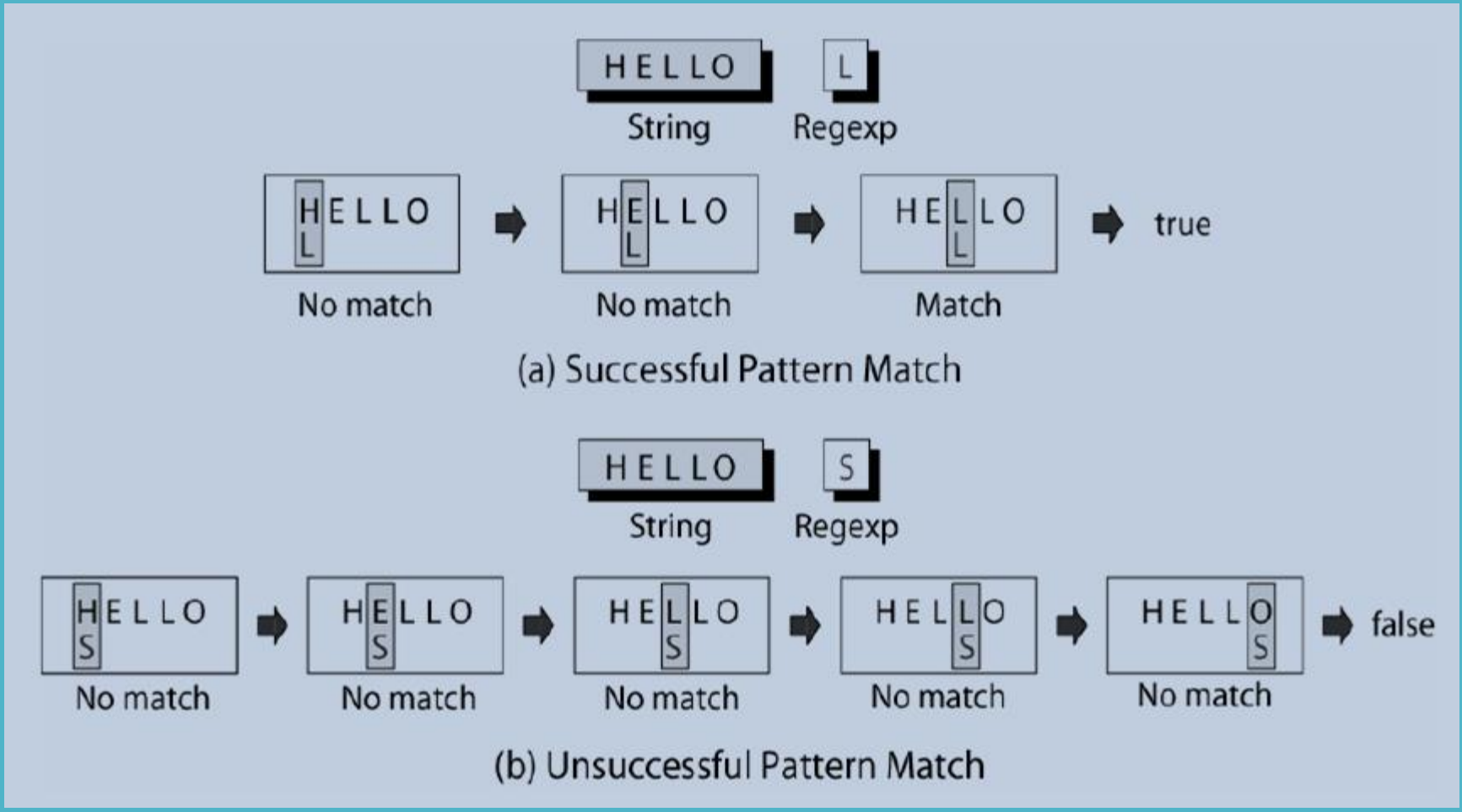
- Οι μεταχαρακτήρες που χρησιμοποιούνται στις κανονικές εκφράσεις είναι οι

\ . * [^ $]

Μία κανονική έκφραση αποτελείται από άτομα (απλούς χαρακτήρες, τελείες, anchors κλπ) που συνδυάζονται μεταξύ τους με τελεστές, με τον ίδιο ακριβώς τρόπο με τον οποίο στις αλγεβρικές εκφράσεις, τα μαθηματικά σύμβολα συνδυάζονται με τους αριθμητικούς τελεστές.

Κανονικές Εκφράσεις

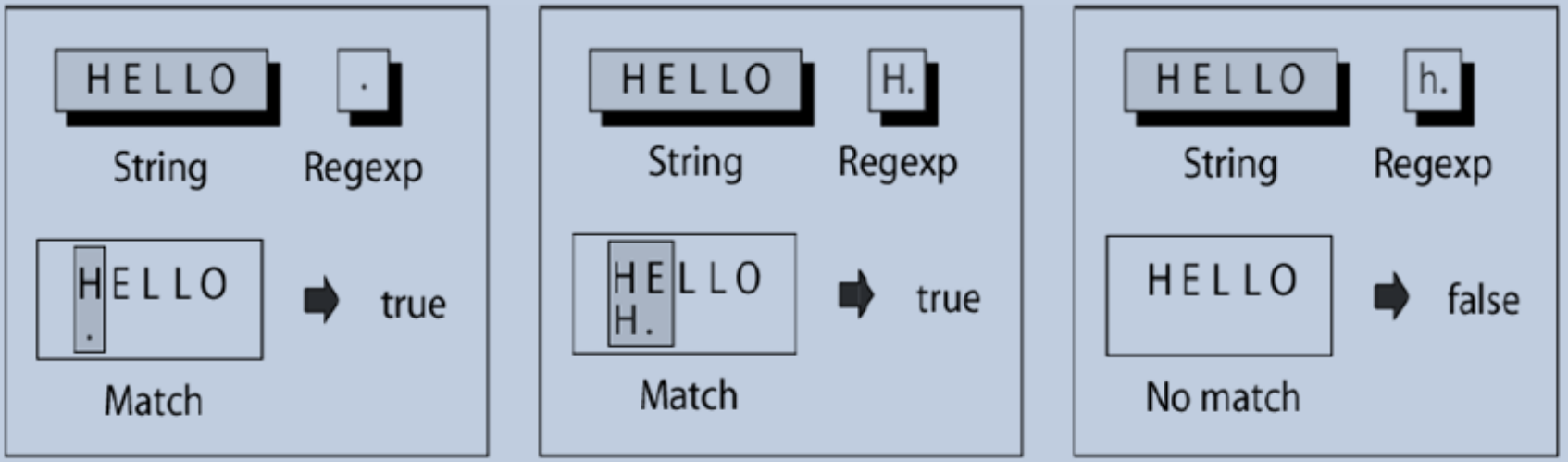
Τρόπος λειτουργίας κανονικών εκφράσεων



Κανονικές Εκφράσεις

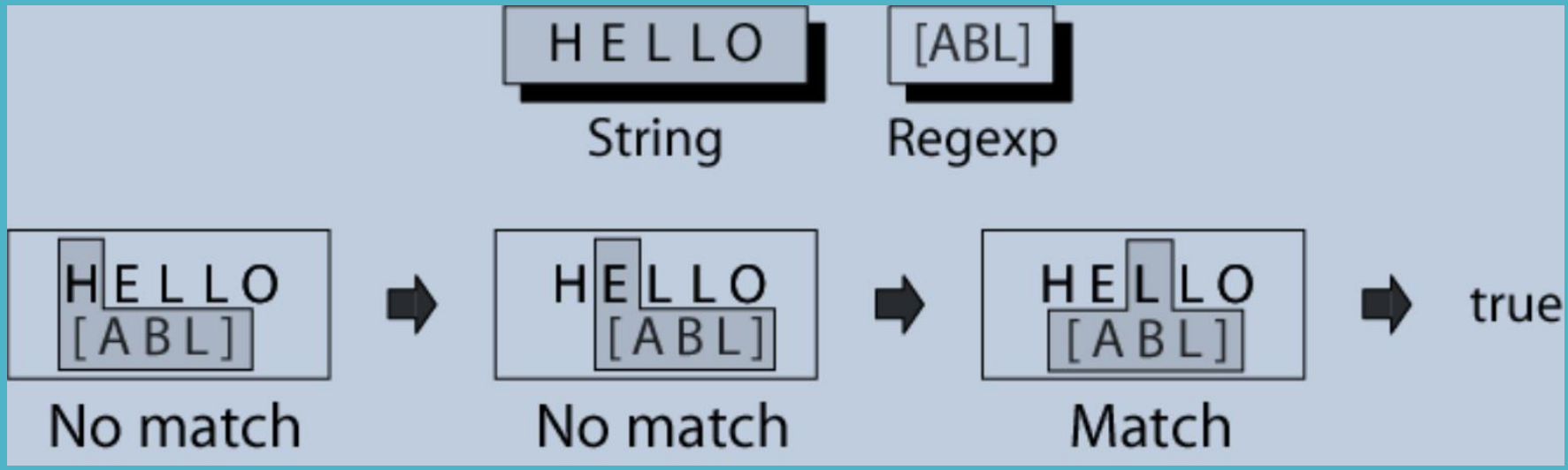
Τρόπος λειτουργίας κανονικών εκφράσεων

Μια τελεία (dot) αντιστοιχεί με οποιοδήποτε απλό χαρακτήρα εκτός από τον new line character (\n).



Κανονικές Εκφράσεις

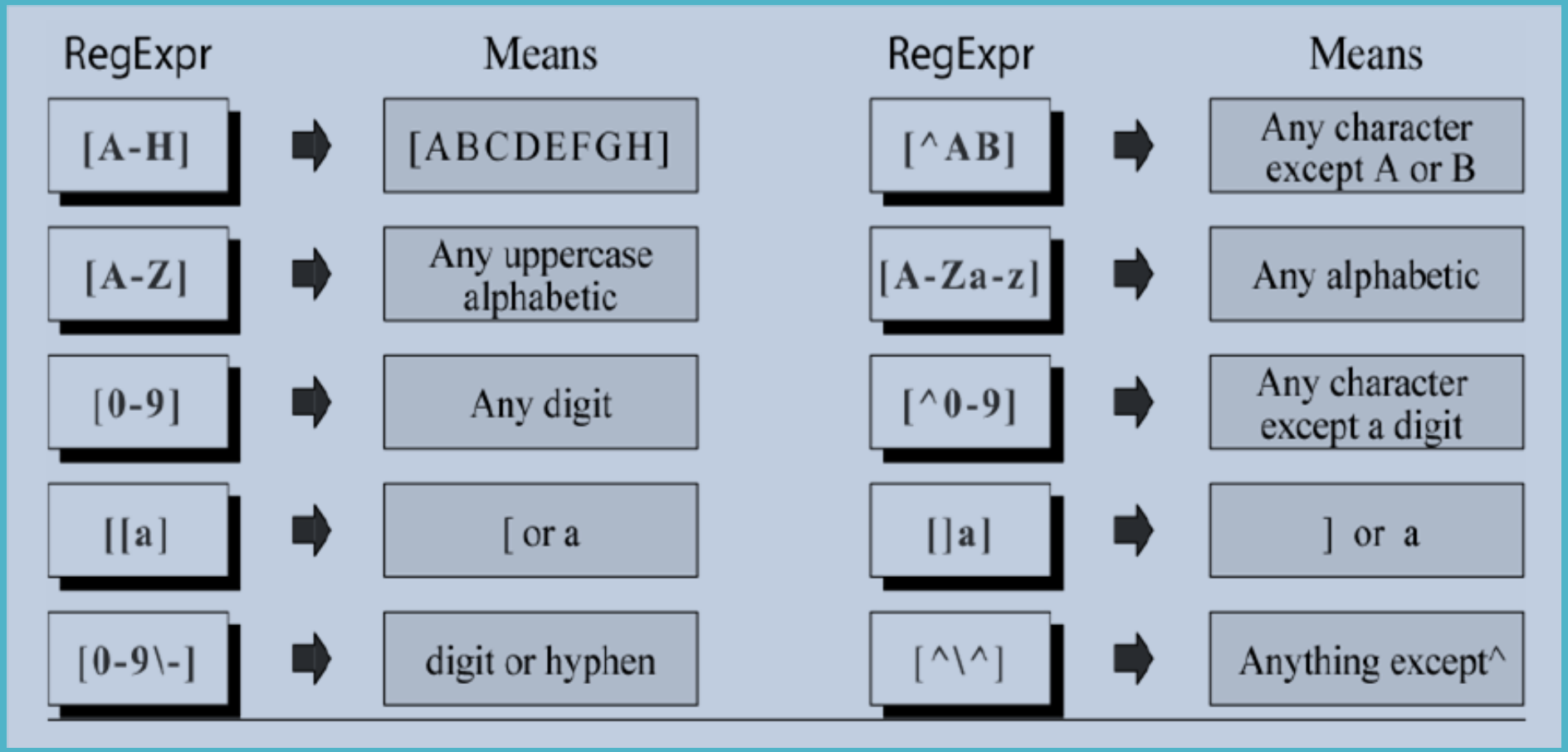
Τρόπος λειτουργίας κανονικών εκφράσεων



Οποιοσδήποτε χαρακτήρας από A έως E → [A-E]
Εναλλακτική γραφή → [ABCDE]
Οποιοσδήποτε χαρακτήρας από A έως Z και από a έως z → [A-Za-z]

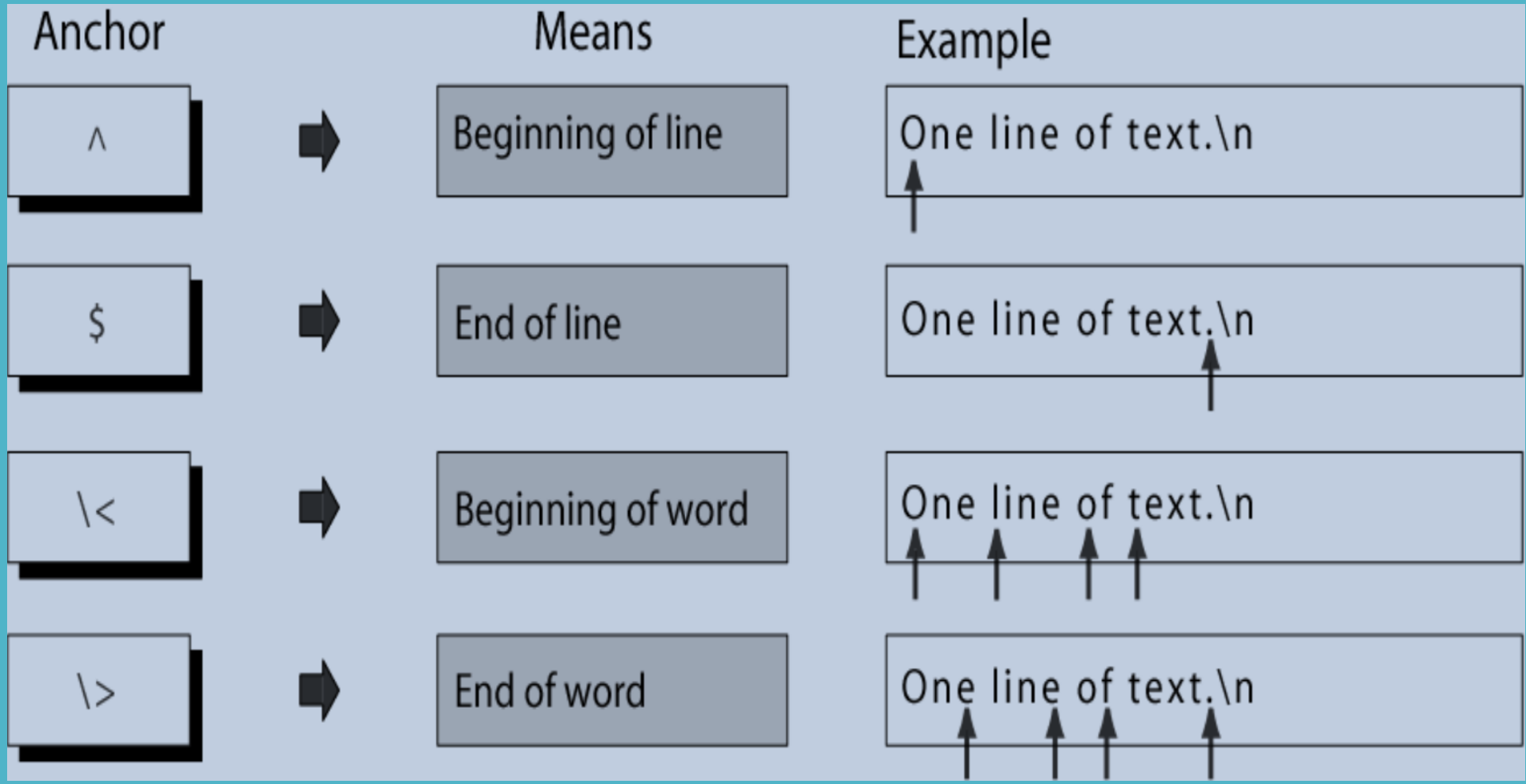
Κανονικές Εκφράσεις

Τρόπος λειτουργίας κανονικών εκφράσεων



Κανονικές Εκφράσεις

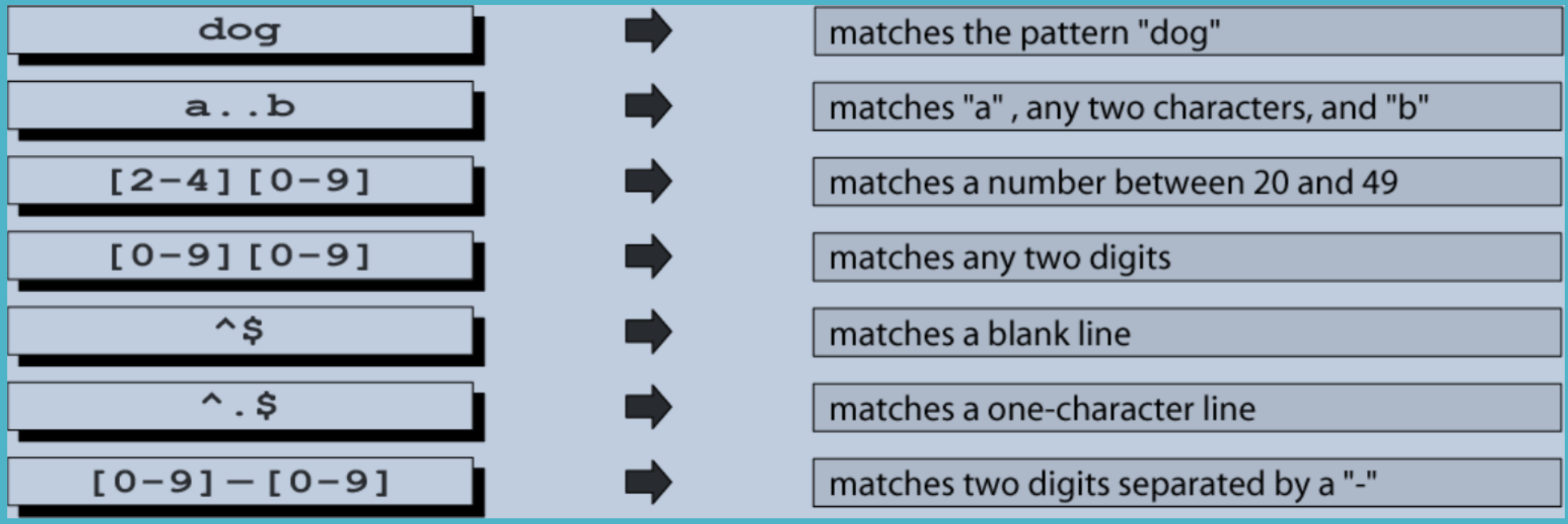
Τρόπος λειτουργίας κανονικών εκφράσεων



Κανονικές Εκφράσεις

Τελεστής ακολουθίας

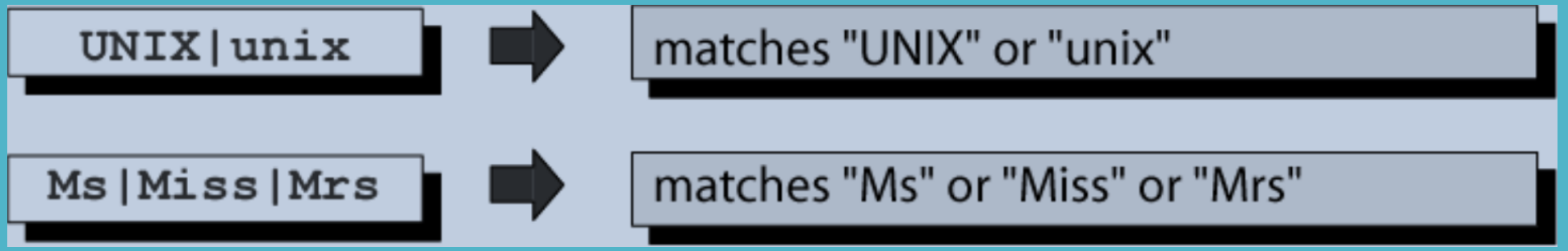
Ο **τελεστής ακολουθίας (sequence operator)** δεν υπάρχει ως σύμβολο. Αυτό σημαίνει ότι αν μια σειρά από atoms φαίνονται σε μια κανονική έκφραση, υποδηλώνεται η παρουσία ενός αόρατου sequence operator ανάμεσά τους.



Κανονικές Εκφράσεις

Τελεστής εναλλαγής

Ο **τελεστής εναλλαγής (alternation operator)** χρησιμοποιείται για να ορίσει μια ή περισσότερες εναλλακτικές περιπτώσεις.

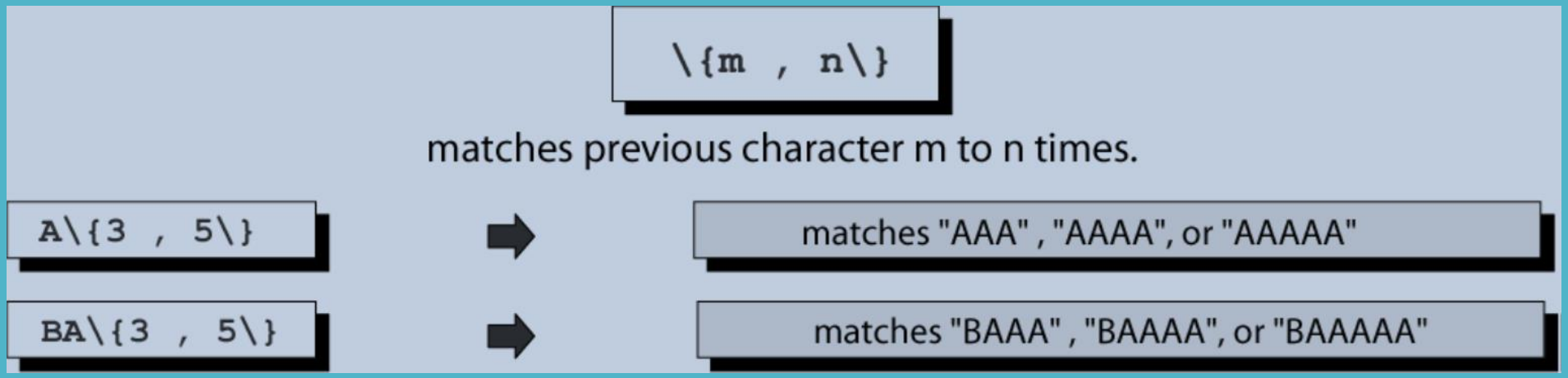


Κανονικές Εκφράσεις

Τελεστής επανάληψης

Ο **τελεστής επανάληψης (repetition operator)** καθορίζει ότι το atom ή η έκφραση που υπάρχει ακριβώς πριν από την επανάληψη μπορεί να επαναληφθεί.

m είναι ο ελάχιστος αριθμός επαναλήψεων.
n είναι ο μέγιστος αριθμός επαναλήψεων.



Κανονικές Εκφράσεις

Τελεστής επανάληψης

Formats

`\{m\}`	➡	matches previous atom exactly m times
`\{m, \}`	➡	matches previous atom m times or more
`\{, n\}`	➡	matches previous atom n times or less

Examples

`CA\{5\}`	➡	`CAAAAA`
`CA\{3,\}`	➡	`CAAA, CAAAA, CAAAAA, …`
`CA\{,2\}`	➡	`C, CA, CAA`

Κανονικές Εκφράσεις

Τελεστής επανάληψης

Formats

*	➡	special case: matches previous atom zero or more times
+	➡	special case: matches previous atom one or more times
?	➡	special case: matches previous atom 0 or one time only

Examples

BA*	➡	B, BA, BAA, BAAA, BAAAA, . . .
B.*	➡	B, BA . . . BZ, BAA . . . BZZ, BAAA . . . BZZZ, . . .
.*	➡	zero or more characters
.+	➡	one or more characters
[0-9]?	➡	zero or one digit

Κανονικές Εκφράσεις

Τελεστής ομαδοποίησης

Ο **τελεστής ομαδοποίησης (group operator)** είναι ένα ζεύγος παρενθέσεων που ανοίγουν και κλείνουν. Όταν μια ομάδα χαρακτήρων περικλείεται σε παρενθέσεις ο επόμενος τελεστής εφαρμόζεται σε όλη την ομάδα.

Η εντολή grep

(General Regular Expression Parser)

grep options RegEX [input file]

grep –e pattern ή –regexp patterns → εκτυπώνει τις γραμμές που περιέχουν το pattern
grep –f file → διαβάζει τα patterns από αρχείο
grep –i → αγνοεί τη διαφορά κεφαλαίων και μικρών γραμμάτων (ignore case)
grep –v → invert match (εκτυπώνει αυτά που δεν ταιριάζουν με το πρότυπο)
grep –w → εκτυπώνει τις γραμμές στις οποίες το ταίριασμα αφορά σε ολόκληρες λέξεις.
grep -n → μπροστά από κάθε γραμμή στην έξοδο εκτυπώνει τον αριθμό γραμμής
grep –r → διαβάζει όλα τα αρχεία ενός καταλόγου (recursively).

Η εντολή grep

ΠΑΡΑΔΕΙΓΜΑΤΑ

Εκτυπώνει τις γραμμές του αρχείου /etc/passwd που περιέχουν οπουδήποτε τη λέξη bash

```
grep bash /etc/passwd
```

Έξοδος

```
root:x:0:0:root:/root:/bin/bash
amarg:x:1000:1000:amarg:/home/amarg:/bin/bash
```

Εκτυπώνει τις γραμμές του αρχείου file.txt που ξεκινούν με τη λέξη linux

```
grep '^linux' file.txt
```

Εκτυπώνει τις γραμμές του αρχείου file.txt που τελειώνουν με τη λέξη linux

```
grep 'linux$' file.txt
```

Εκτυπώνει τις γραμμές του αρχείου file.txt που περιέχουν ΜΟΝΟ τη λέξη linux

```
grep '^linux$' file.txt
```

Η εντολή grep

ΠΑΡΑΔΕΙΓΜΑΤΑ

Εκτυπώνει τις λέξεις του αρχείου words που ξεκινούν με co περιέχουν τέσσερις οποιουσδήποτε χαρακτήρες και τελειώνουν με er

grep 'co....er' words

Εκτυπώνει τις λέξεις του αρχείου words που ξεκινούν με co περιέχουν δύο οποιουσδήποτε χαρακτήρες, ο τρίτος χαρακτήρας είναι r ή p ή q και τελειώνουν με es

grep 'co..[rpq]es' words

Εκτυπώνει τις λέξεις του αρχείου words που ξεκινούν με br, ο τρίτος χαρακτήρας μπορεί να είναι οποιοσδήποτε, ο τέταρτος χαρακτήρας ΔΕΝ μπορεί να είναι j, ακολουθεί αυθαίρετο πλήθος οποιωνδήποτε χαρακτήρων και τελειώνουν με x.

grep '^[A-Z] ' words

Εκτυπώνει τις λέξεις του αρχείου words που ξεκινούν με κεφαλαίο γράμμα

Η εντολή grep

ΠΑΡΑΔΕΙΓΜΑΤΑ

- Όλες οι γραμμές που δεν ξεκινούν από κεφαλαίο αγγλικό χαρακτήρα :
  - **grep '^[^A-Z]' file**
- Όλες οι γραμμές που περιέχουν !,& ,* :
  - **grep '([\!*\&])' file**
- Όλες οι γραμμές που περιέχουν την τιμή $1.99 :
  - **grep '\$1\.99' file**
- Όλες οι γραμμές με μήκος 2 χαρακτήρες :
  - **grep '^..$'file ή ^.\{2\}$**

Η εντολή grep

ΠΑΡΑΔΕΙΓΜΑΤΑ

- Όλες οι γραμμές που έχουν μήκος ακριβώς 17 χαρακτήρες:
  - **grep '^.\{17\}$' file**
- Όλες οι γραμμές που έχουν μήκος τουλάχιστον 25 χαρακτήρες:
  - **grep '^.\{25,\}$'**
- Όλες οι γραμμές που δεν έχουν μήκος 3 χαρακτήρες:
  - **grep -v '^...$'file**

Η εντολή grep

ΠΑΡΑΔΕΙΓΜΑΤΑ

- Όλες οι γραμμές που ξεκινούν με * :
  - **^***
- Όλες οι γραμμές που δεν περιέχουν αριθμούς :
  - **^[^0-9]*$**
- Όλες οι γραμμές που περιέχουν τα έτη 1991 έως 1995 :
  - **199[1-5]**
- Οποιαδήποτε ακολουθία χαρακτήρων δεν περιέχει ψηφία:
  - **[A-Za-z][A-Za-z]***

Η εντολή grep

ΠΑΡΑΔΕΙΓΜΑΤΑ

- Οποιοσδήποτε προσημασμένος ακέραιος :
  - **[+\-][0-9][0-9]***
- Οποιαδήποτε ακολουθία χαρακτήρων :
  - **.* (ιδιωματισμός!)**
- Οποιοδήποτε αναγνωριστικό (identifier) :
  - **[a-zA-Z_][a-zA-Z_0-9]***
- Οποιοσδήποτε πραγματικός αριθμός χωρίς πρόσημο:
  - **[0-9]+[.][0-9]+**
  - **[0-9]*[.][0-9]***

Η εντολή AWK

(Aho, Weinberger, Kerningham)

Γλώσσα ταυτοποίησης προτύπων που αναλύει αρχεία κειμένου που δέχεται στην είσοδό της και επιστρέφει τις εγγραφές που ταιριάζουν σε ένα πρότυπο.

Χρησιμοποιεί το συντακτικό της γλώσσας C και περιέχει δεσμευμένες λέξεις, σταθερές, μεταβλητές, εντολές, συναρτήσεις και πίνακες.

Κάθε γραμμή του αρχείου εισόδου μπορεί να διαχωριστεί σε πεδία που μπορούν να διαχειριστούν ανεξάρτητα ορίζοντας τον κατάλληλο διαχωριστή. Ο default separator είναι το κενό (space) ενώ για να οριστεί άλλος χαρακτήρας χρησιμοποιείται ο διακόπτης –F.

Η αναζήτηση προτύπων στο αρχείο εισόδου γίνεται με τη βοήθεια κανονικών εκφράσεων.

$ awk –F [field separator] ‘ { awk program } ’ [input_file]

$ awk –F [field separator] -f [awk script] [input_file]

Η εντολή AWK

(Aho, Weinberger, Kerningham)

Μεταβλητές

Μεταβλητές γενικού τύπου π.χ.

title="Number of students" , no=100, weight=77.9

ΕΙΔΙΚΕΣ (δεσμευμένες) ΜΕΤΑΒΛΗΤΕΣ

$n	n-οστό πεδίο στη γραμμή, $0 – ολόκληρη η γραμμή	
FS	διαχωριστής πεδίων (εξ ορισμού κενό και tab)	
OFS	διαχωριστής πεδίων αρχείου εξόδου (εξ ορισμού κενό)	
NR	αριθμός εγγραφής (γραμμής)	} Αρχικοποιούνται για κάθε γραμμή
NF	πλήθος πεδίων της γραμμής	
FILENAME	όνομα αρχείου εισόδου	
RS	διαχωριστής εγγραφών αρχείου εισόδου (εξ ορισμού new line)	
ORS	διαχωριστής εγγραφών αρχείου εξόδου (εξ ορισμού new line)	

Η εντολή AWK

Τα προγράμματα awk διαιρούνται σε τρία κύρια blocks:

BEGIN block, block επεξεργασίας, END block

Εκτός και αν ορίζεται ρητά, όλες οι εντολές εμφανίζονται στο block επεξεργασίας.

Οποιοδήποτε από τα 3 τμήματα μπορεί να παραλείπεται.

Οι εντολές διαιρούνται σε δύο τμήματα:

- Ένα κριτήριο επιλογής, που αναφέρει στην awk τι πρέπει να ταιριάξει, και
- Μια αντίστοιχη ενέργεια που αναφέρει στην awk τι θα κάνει όταν βρεθεί μια γραμμή που ταιριάζει με το συγκεκριμένο κριτήριο επιλογής.

Το τμήμα ενεργειών της εντολής βρίσκεται σε { } και μπορεί να περιέχει πολλές εντολές.

Οι εντολές που διαθέτουν κριτήριο επιλογής εφαρμόζονται σε κάθε γραμμή που αντιστοιχεί ή καθιστά αληθές το κριτήριο, ανάλογα αν αυτό είναι μια κανονική έκφραση ή μια λογική έκφραση.

Οι εντολές που δεν έχουν κριτήρια επιλογής εφαρμόζονται σε κάθε γραμμή του αρχείου εισόδου.

Η εντολή AWK

Αντιστοίχιση προτύπων

- Κάθε γραμμή πριν επεξεργαστεί μπορεί να αντιστοιχηθεί (να ταιριάξει με ένα πρότυπο). Το πρότυπο περικλείεται σε / /.
- Format :

  /pattern/ { action } εκτελείται αν η γραμμή περιέχει το πρότυπο

  !/pattern/ { action } εκτελείται αν η γραμμή ΔΕΝ περιέχει το πρότυπο
- παραδείγματα:

```
/^$/              { print "This line is blank " }
/text/            { print "This line includes text" }
/[0-9]+$/         { print "Integer:", $0 }
/[a-z]+/          { print "String:", $0 }
/^[A-Z]/          { print "start with an uppercase letter" }
```

Η εντολή AWK

```
$cat names.txt
Per Wisung 021-336699
Jan Medin 021-332211
Hans Persson 021 112233
Göran Persson 021-336666
```

```
$awk ´{print "Name: ", $1,$2, "Telephone:", $3}
´ names.txt
```

```
Name: Per Wisung        Telephone: 021-336699
Name: Jan Medin         Telephone: 021-332211
Name: Hans Persson      Telephone: 021
Name: Göran Persson     Telephone: 021-336666
```

Η εντολή AWK

```
$ cat sales
John Anderson,12,23,7,42
Joe Turner,10,25,15,50
Susan Greco,15,13,18,46
Bob Burmeister,8,21,17,46

$ awk -F, '{print $1,$5}' sales

John Anderson 42
Joe Turner 50
Susan Greco 46
Bob Burmeister 46
```

Η εντολή AWK

```
$ cat emp.data
John Anderson:sales:1980
Joe Turner:marketing:1982
Susan Greco:sales:1985
Ike Turner:pr:1988
Bob Burmeister:accounting:1991

$ awk -F: '$3 == 1980,$3 == 1985 {print $1, $3}' emp.data
John Anderson 1980
Joe Turner 1982
Susan Greco 1985
```

Η εντολή AWK

Συντακτικό των εντολών - παραδείγματα

```
if ( found == true )                    #   if (expr)
      print "Found"                     #       {action1}
else                                    #   else
      print "Not found"                 #       {action2}

while ( i <= 100)                       #   while (cond.)
      { i=i+1; print i }                #       {action}

do                                      #   do
      { i=i+1; print i }                #       {action}
while ( i<100)                          #   while (cond.)

for (i=1; i<10; i++ ) {                 #    for (set; test;increment)
      i2= i*i                           #       {action}
      printf(" %d*%d = %d\n", i, i, i2)
}
```

Η εντολή AWK

Περιεχόμενα του αρχείου /etc/passwd

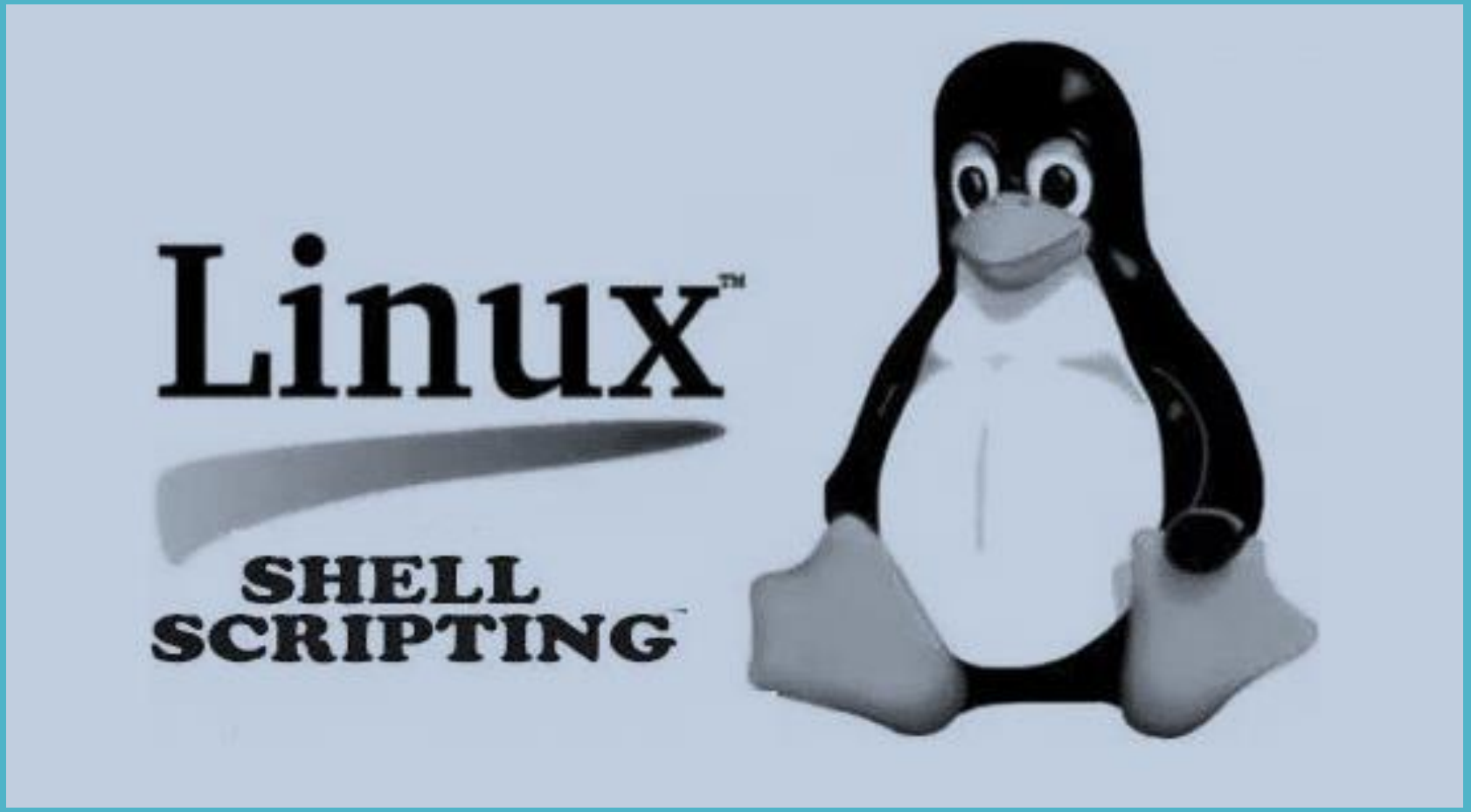
```
root:x:0:0:Super-User:/:/bin/csh
sysadm:x:0:0:System V Administration:/usr/admin:/bin/sh
cmwlogin:x:0:994:CMW Login UserID:/usr/CMW:/sbin/csh
diag:x:0:996:Hardware Diagnostics:/usr/diags:/bin/csh
daemon:x:1:1:daemons:/:/dev/null
bin:x:2:2:System Tools Owner:/bin:/dev/null
uucp:x:3:5:UUCP Owner:/usr/lib/uucp:/bin/csh
sys:x:4:0:System Activity Owner:/var/adm:/bin/sh
adm:x:5:3:Accounting Files Owner:/var/adm:/bin/sh
lp:x:9:9:Print Spooler Owner:/var/spool/lp:/bin/sh
auditor:x:11:0:Audit Activity Owner:/auditor:/bin/sh
dbadmin:x:12:0:Security Database Owner:/dbadmin:/bin/sh
guest:x:998:998:Guest Account:/usr/people/guest:/bin/csh
```

Η εντολή AWK

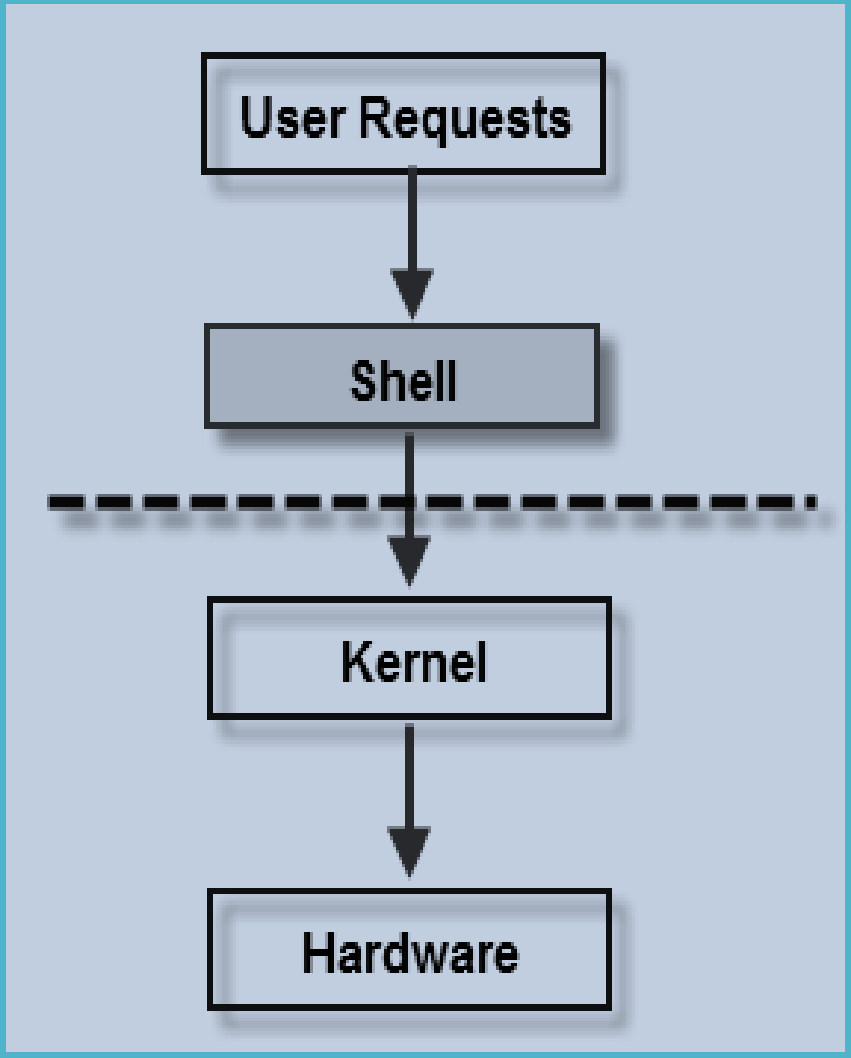
- Δημιουργία λίστας με τους χρήστες του συστήματος με μήκος login name έως 4 χαρακτήρες

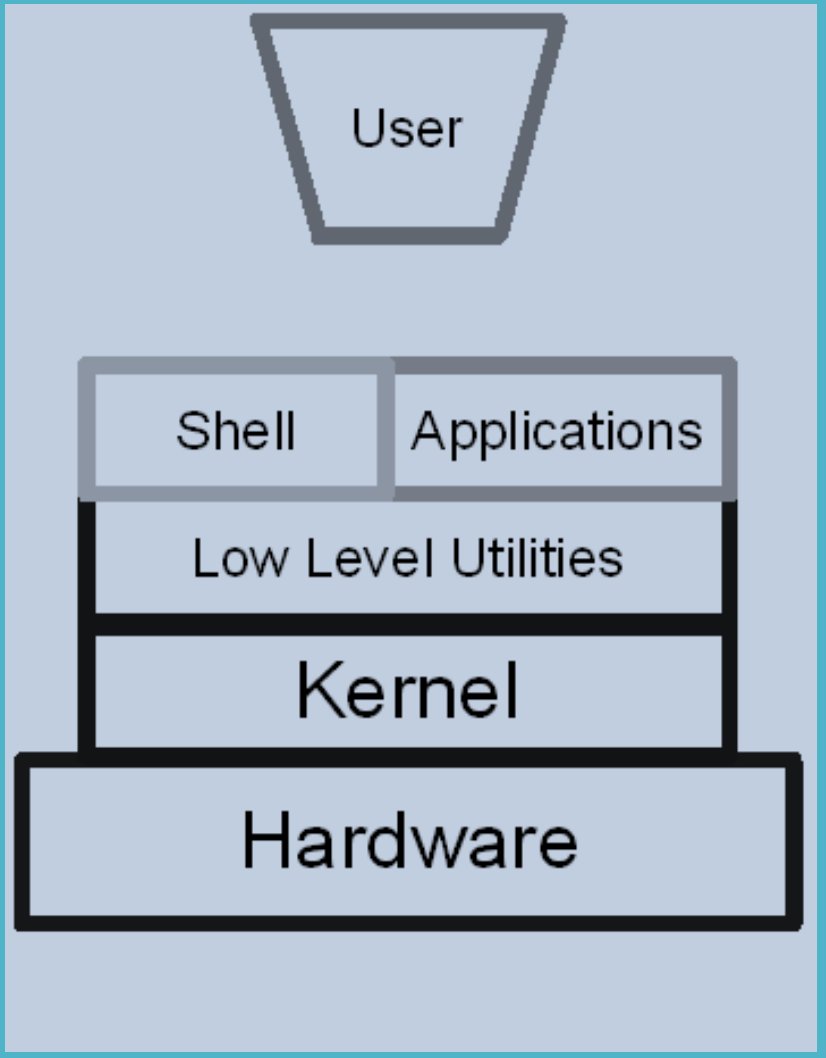
```
$ grep  "^[^:]\{1,4\}:" /etc/passwd | awk -F: '{print $5}'
Super-User
Hardware Diagnostics
System Tools Owner
UUCP Owner
System Activity Owner
Accounting Files Owner
Print Spooler Owner
```

Shell Programming

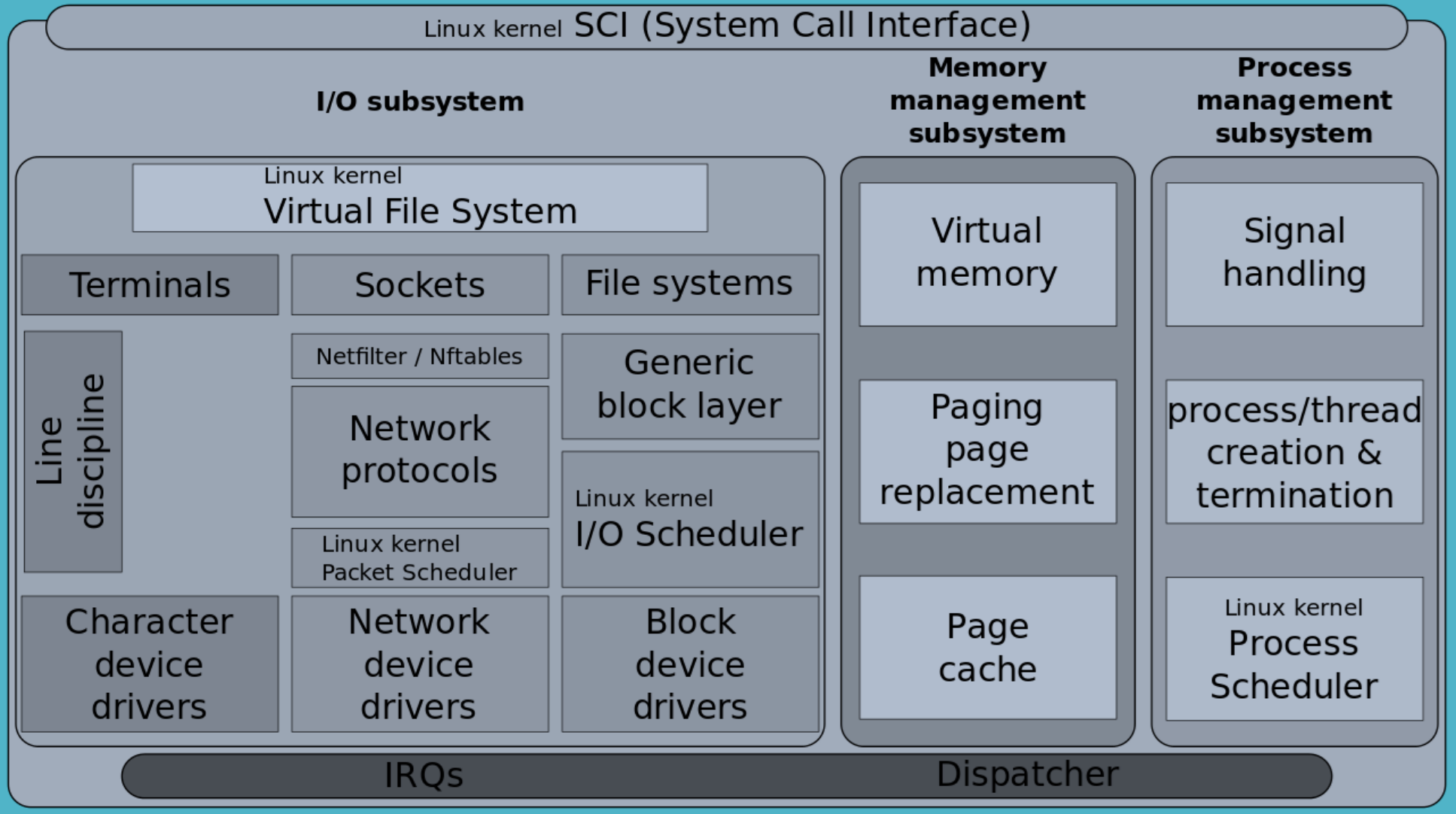


Φλοιός και πυρήνας





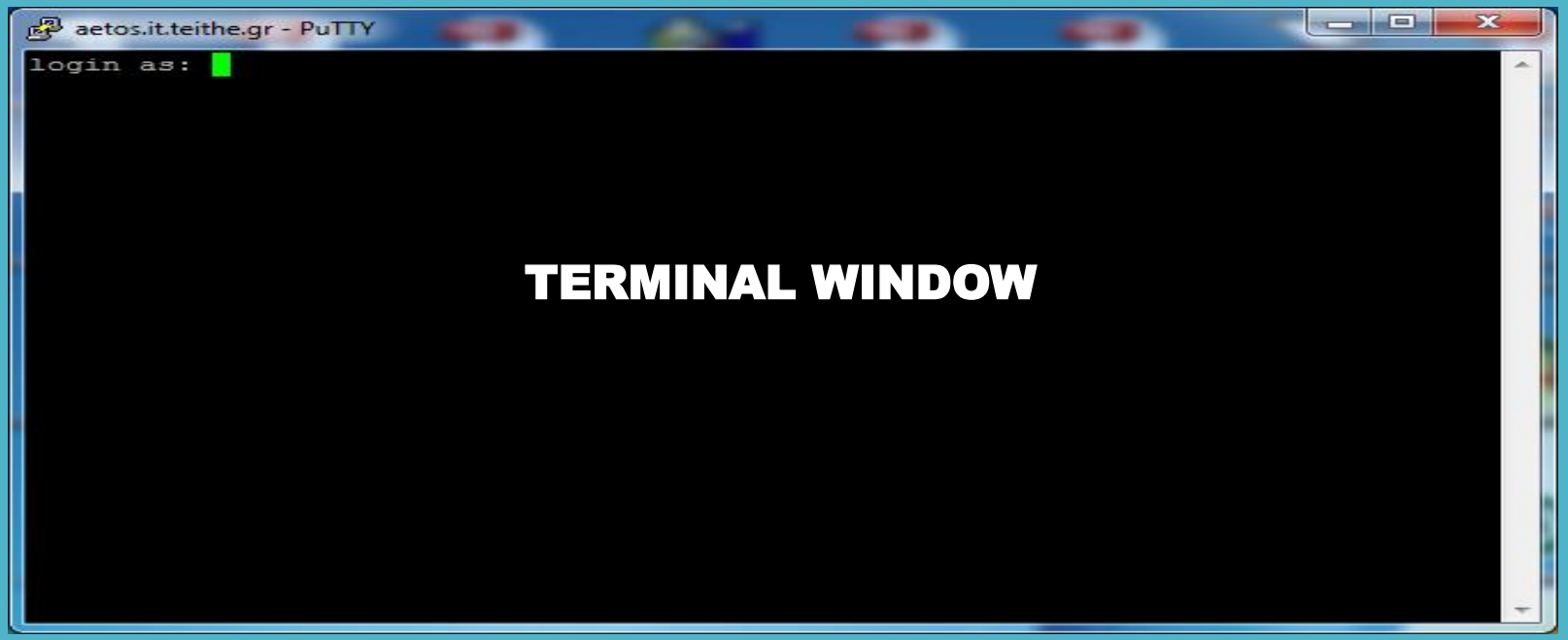
Φλοιός και πυρήνας



Ταυτοποίηση χρήστη

Σε ένα λειτουργικό σύστημα Linux για κάθε χρήστη ορίζονται οι επόμενες ιδιότητες

Login name ή username → συνήθως μέχρι 8 χαρακτήρες.
Password → συνθηματικό εισόδου.
User Id → μοναδικός αριθμός που ταυτοποιεί τον κάθε χρήστη.
Group Id → μοναδικός αριθμός που ταυτοποιεί την πρωτεύουσα ομάδα του χρήστη.
Home Directory → ο προσωπικός κατάλογος του κάθε χρήστη
Φλοιός (Shell) → το περιβάλλον αλληλεπίδρασης του χρήστη με τον πυρήνα



Σύνδεση στο σύστημα

Εάν ο χρήστης καταχωρήσει το σωστό login name και το σωστό password το σύστημα τον μεταφέρει αυτόματα στον προσωπικό του κατάλογο.
Εκτελούνται οι εντολές των αρχείων /etc/profile (για όλους τους χρήστες).

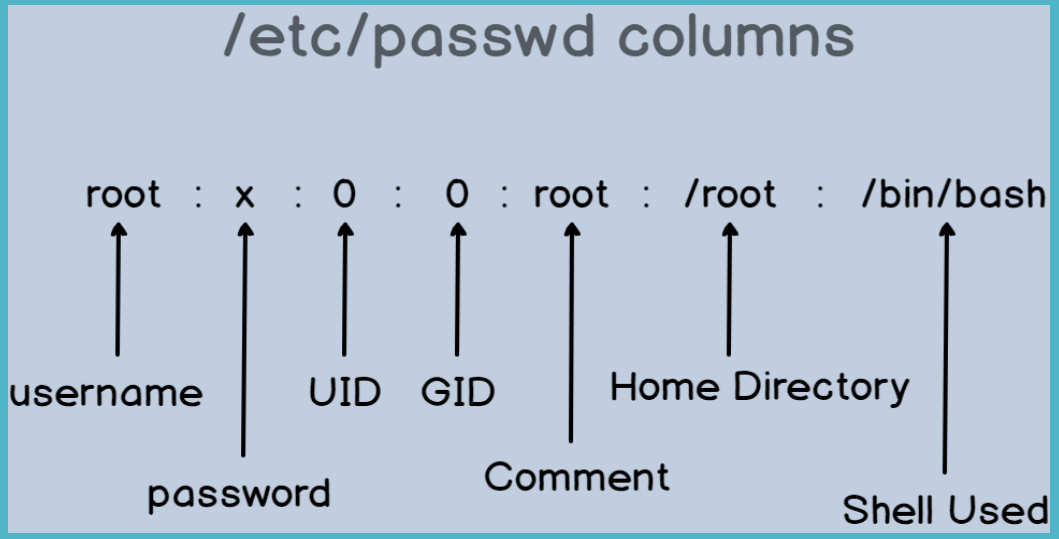
Το home directory κάθε χρήστη μπορεί περιέχει ένα ή περισσότερα από τα ακόλουθα bash startup files, τα οποία περιέχουν εντολές που εφαρμόζονται μόνον για την τρέχουσα χρήση του συστήματος:

~/.bash_profile,
~/.bash_login,
~/.profile,
~/.bashrc, και
~/.bash_logout

```
[root@tecmint ~]# cat /etc/profile
# /etc/profile

# System wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, as this
# will prevent the need for merging in future updates.

pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *)
            if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}


if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
        # ksh workaround
        EUID=`id -u`
        UID=`id -ru`
    fi
```

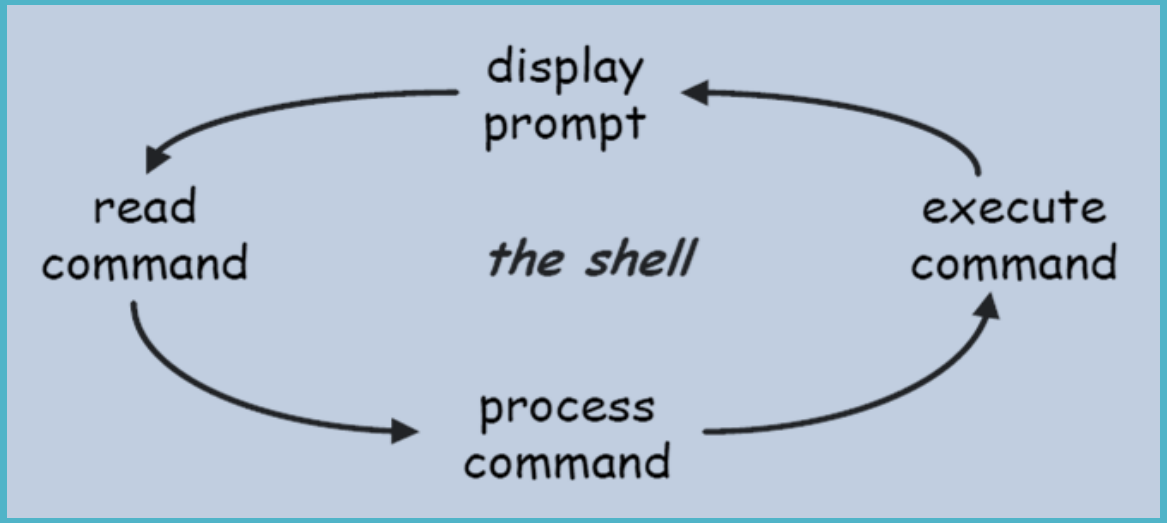
Το αρχείο /etc/passwd

```
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
```



Η λειτουργία του φλοιού

Ο φλοιός εκτελεί επαναληπτικά τις παρακάτω τέσσερις εργασίες



1. **Εμφανίζει την προτροπή (command prompt)**
2. **Διαβάζει την εντολή του χρήστη**
3. **Αποκωδικοποιεί και επεξεργάζεται την εντολή του χρήστη**
4. **Εκτελεί την εντολή του χρήστη**

Login shell : δημιουργείται κάθε φορά που γίνεται login σε ένα account.

Nonlogin shell : δημιουργείται όταν αρχίζει ένα πρόσθετο bash shell , κατά τη διάρκεια της σύνδεσης του χρήστη, όπως π.χ. όταν ανοίγει ένα terminal window.

Διαθέσιμοι φλοιοί

```
vivek@nixcraft-asus:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
/bin/ksh93
/bin/rksh93
/bin/tcsh
/usr/bin/tcsh
vivek@nixcraft-asus:~$
```

Comparison between shells

- sh (bourne shell)
  - small
  - good scripting capability
  - popular with system administrators
- csh (c-shell)
  - extends sh
  - uses a c-like syntax
  - created by Bill Joy
  - popular with UNIX users
- bash
  - "bourne-again" shell
  - extends sh with some features form csh
  - The linux default
  - We will use bash
- tcsh
  - extension of csh

Ανάκτηση τρέχοντος φλοιού: echo $SHELL

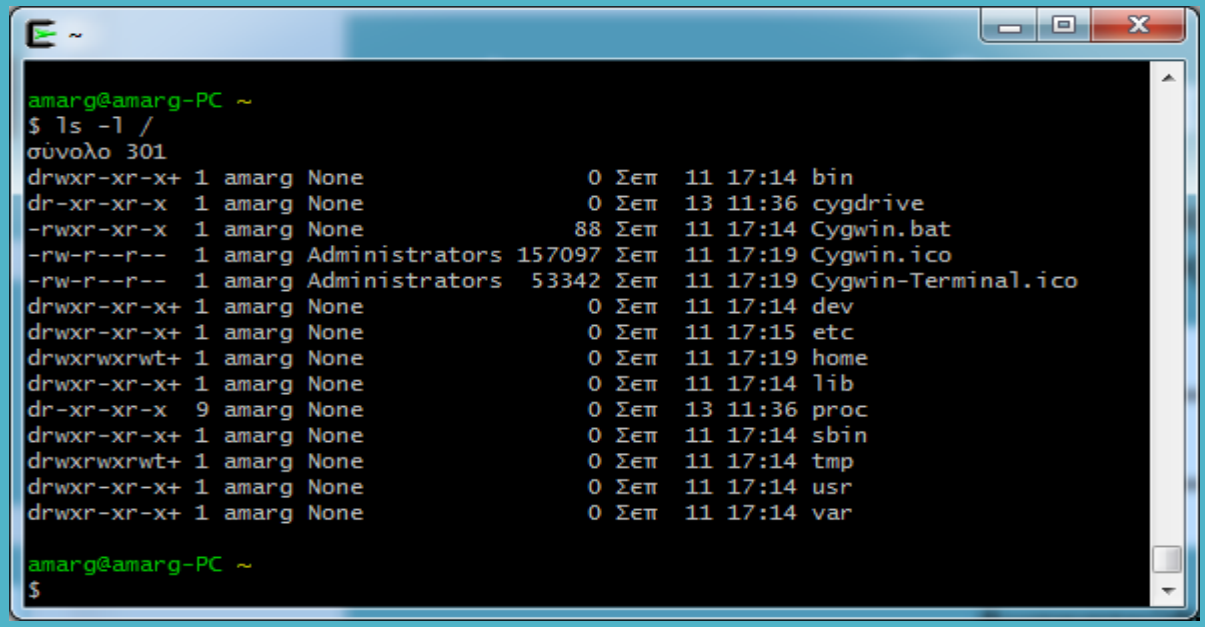
Αλλαγή φλοιού: chsh –s [shell name] π.χ.

chsh –s /bin/tcsh

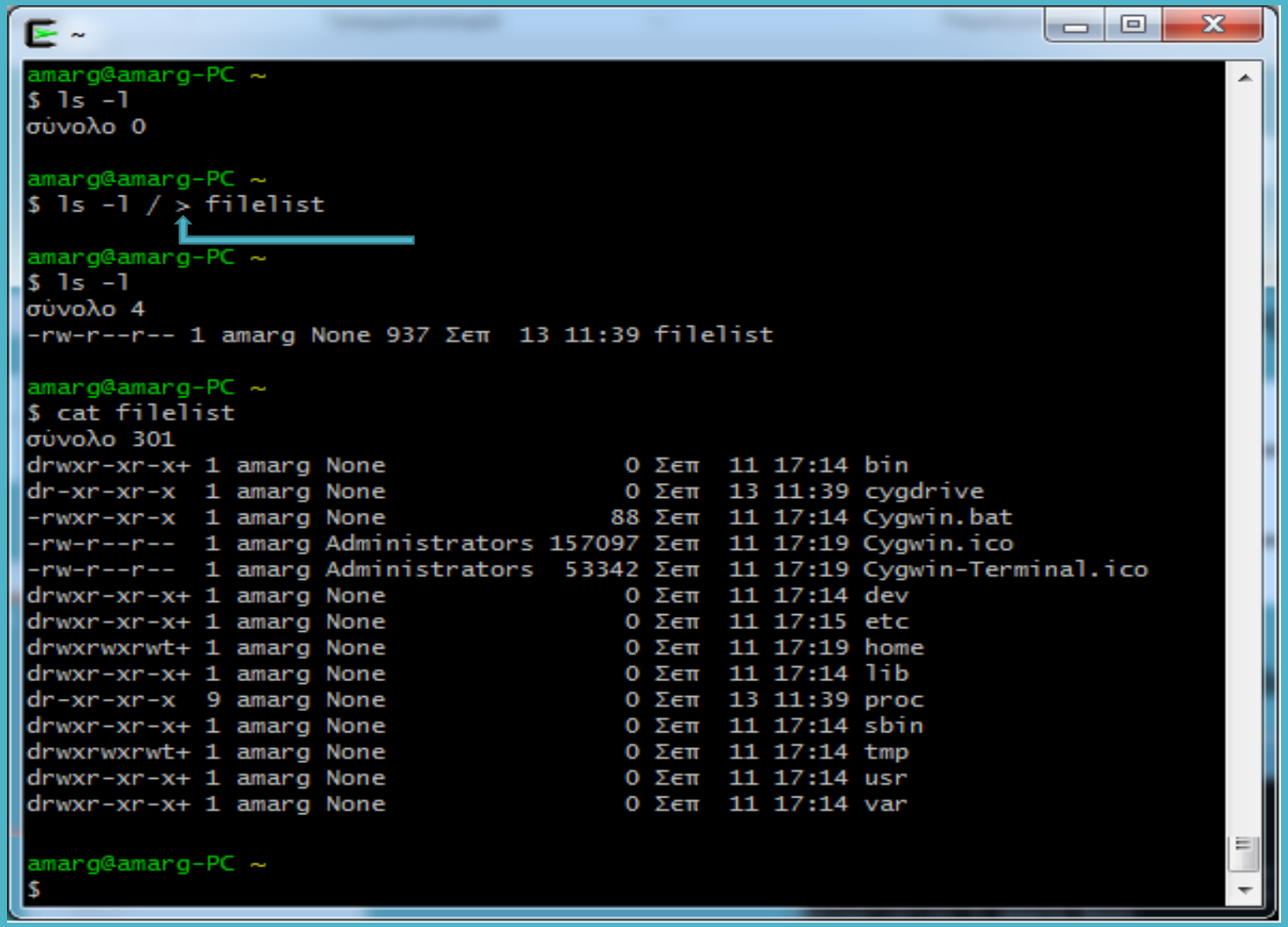
Ανακατεύθυνση και διασωλήνωση

Η προεπιλεγμένη είσοδος των εντολών είναι το πληκτρολόγιο
Η προεπιλεγμένη έξοδος των εντολών είναι η οθόνη

Ανακατεύθυνση εισόδου < **Ανακατεύθυνση εξόδου >**



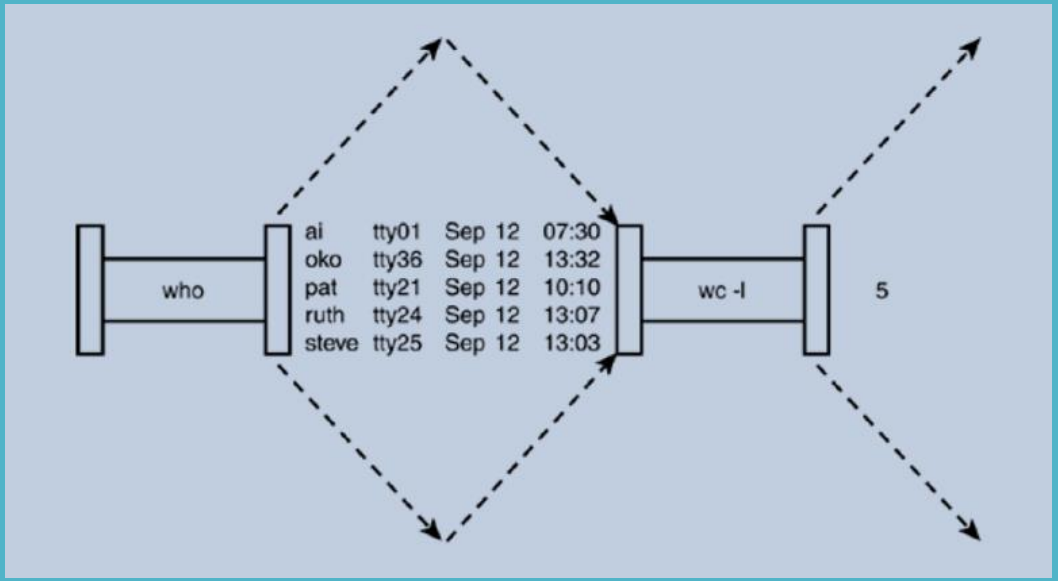
Ανακατεύθυνση και διασωλήνωση



who > connectedUsers
wc –l < connectedUsers

Ανακατεύθυνση και διασωλήνωση

Διασωλήνωση → η έξοδος μιας εντολής δεν εκτυπώνεται στην οθόνη αλλά γίνεται είσοδος μιας άλλης εντολής.



Παραδείγματα

1) **Να μετρηθεί το πλήθος των συνδεδεμένων χρηστών**

who | wc – l

2) **Να μετρηθεί το πλήθος των καταλόγων του συστήματος**

ls -l / | grep -e '^d' | wc -l

3) **Ανάκτηση των ονομάτων χρηστών και αποθήκευσή τους σε αρχείο**

cat /etc/passwd | awk -F : '{print $1}' > alluser.txt

Bash shell → εσωτερικές εντολές

alias
bg
bind
break
builtin
case
cd
command
compgen
complete
continue
declare
dirs
disown
echo
enable
eval
exec
exit
export
fc
fg
getopts
hash
help
history
if
jobs
kill
let
local
logout
popd
printf
pushd
pwd
read
readonly
return
set
shift
shopt
source
suspend
test
times
trap
type
typeset
ulimit
umask
unalias
unset
until
wait
while

Μεταβλητές περιβάλλοντος tsch

- Περιέχουν τις τιμές παραμέτρων που είναι αναγκαίες για τη λειτουργία του συστήματος.
- Αρχικοποιούνται για κάθε χρήστη κατά την είσοδό του στο σύστημα.
- Οι τιμές τους μεταβιβάζονται σε κάθε θυγατρική διεργασία που ξεκινά από το φλοιό.
- Γράφονται με κεφαλαία γράμματα.

- COLUMNS → το πλήθος των στηλών στην οθόνη του τερματικού σύνδεσης
- DISPLAY → χρησιμοποιείται από το γραφικό περιβάλλον
- EDITOR → η διαδρομή προς τον προεπιλεγμένο επεξεργαστή κειμένου
- GROUP → η πρωτεύουσα ομάδα του χρήστη
- HOST → το όνομα του συστήματος στο οποίο χρησιμοποιείται το κέλυφος
- HOSTTYPE → ο τύπος του συστήματος στον οποίο χρησιμοποιείται το κέλυφος
- MACHTYPE → ο τύπος του επεξεργαστή του συστήματος
- OSTYPE → ο τύπος του λειτουργικού συστήματος UNIX
- PATH → λίστα καταλόγων αναζήτησης εκτελέσιμων αρχείων
- REMOTEHOST → το όνομα του συστήματος από το οποίο συνδέθηκε ο χρήστης
- USER → το όνομα του χρήστη που έχει συνδεθεί στο σύστημα.

Μεταβλητές περιβάλλοντος (printenv)

```
PAGER=less
HOSTNAME=icon
MAILCHECK=60
PS1=$
USER=username
MACHTYPE=i486-pc-linux-gnu
EDITOR=emacs
DISPLAY=:0.0
LOGNAME=username
SHELL=/bin/bash
HOSTTYPE=i486
OSTYPE=linux-gnu
HISTSIZE=150
HOME=/home/username
TERM=xterm-debian
TEXEDIT=jed
PATH=/usr/sbin:/usr/sbin:/usr/local/bin:
/usr/bin:/bin:/usr/bin/X11:/usr/games
_=/usr/bin/printenv
```

Τοπικές μεταβλητές κελύφους tsch

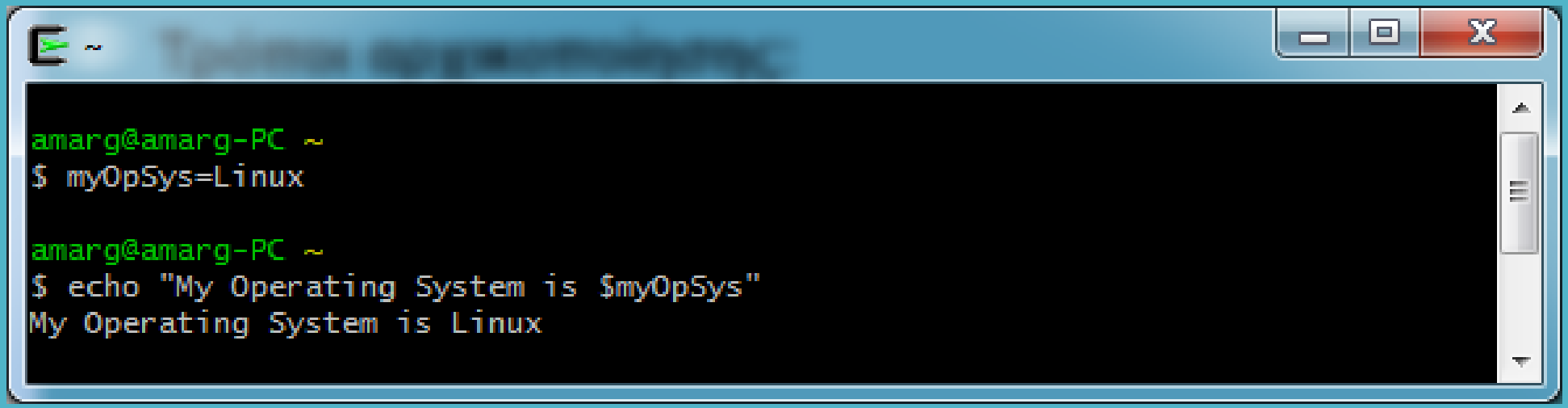
Οι τοπικές μεταβλητές αρχικοποιούνται από το φλοιό
Δεν μεταβιβάζονται σε άλλα προγράμματα που εκτελούνται στο σύστημα.

cdpath → λίστα καταλόγων αναζήτησης υποκαταλόγων για την εντολή cd.
cwd → το πλήρες όνομα της διαδρομής του τρέχοντος καταλόγου.
gid o → ο κωδικός της πρωτεύουσας ομάδας του χρήστη.
group → το όνομα της πρωτεύουσας ομάδας του χρήστη.
home → η απόλυτη διαδρομή προς τον προσωπικό κατάλογο του χρήστη.
inputmode → παίρνει μία από τις τιμές insert και overwrite**.**
path → λίστα καταλόγων αναζήτησης εκτελέσιμων αρχείων.
prompt → ορισμός του command prompt.
uid → ο κωδικός του χρήστη.
user → το όνομα του χρήστη.
shell → η απόλυτη διαδρομή προς το φλοιό που χρησιμοποιείται.

Τοπικές μεταβλητές κελύφους

Τρόποι αρχικοποίησης

1) Απευθείας αρχικοποίηση

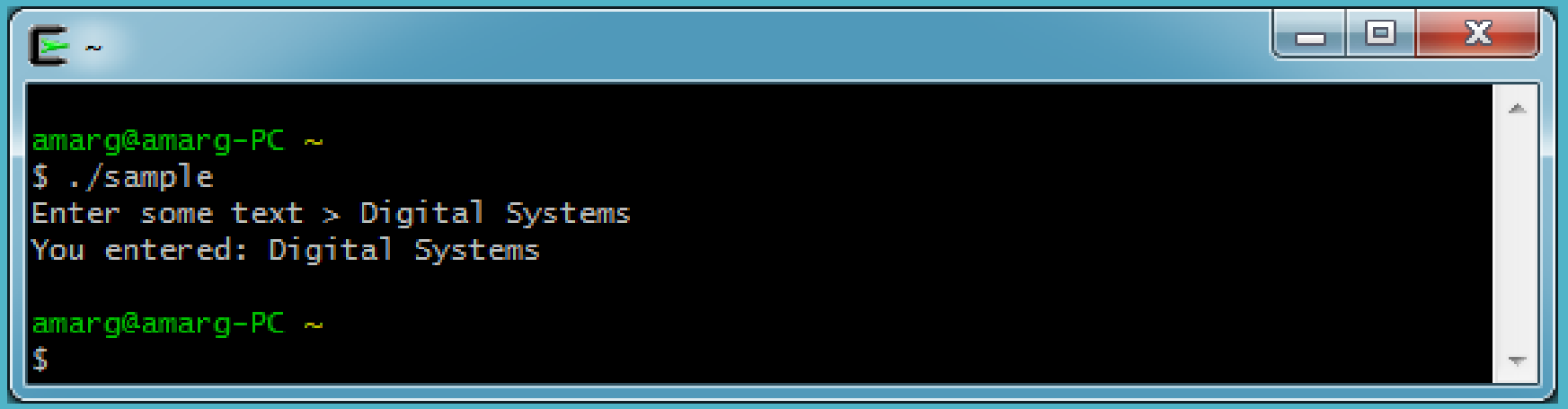


Τοπικές μεταβλητές κελύφους

2) Αρχικοποίηση με την εντολή read

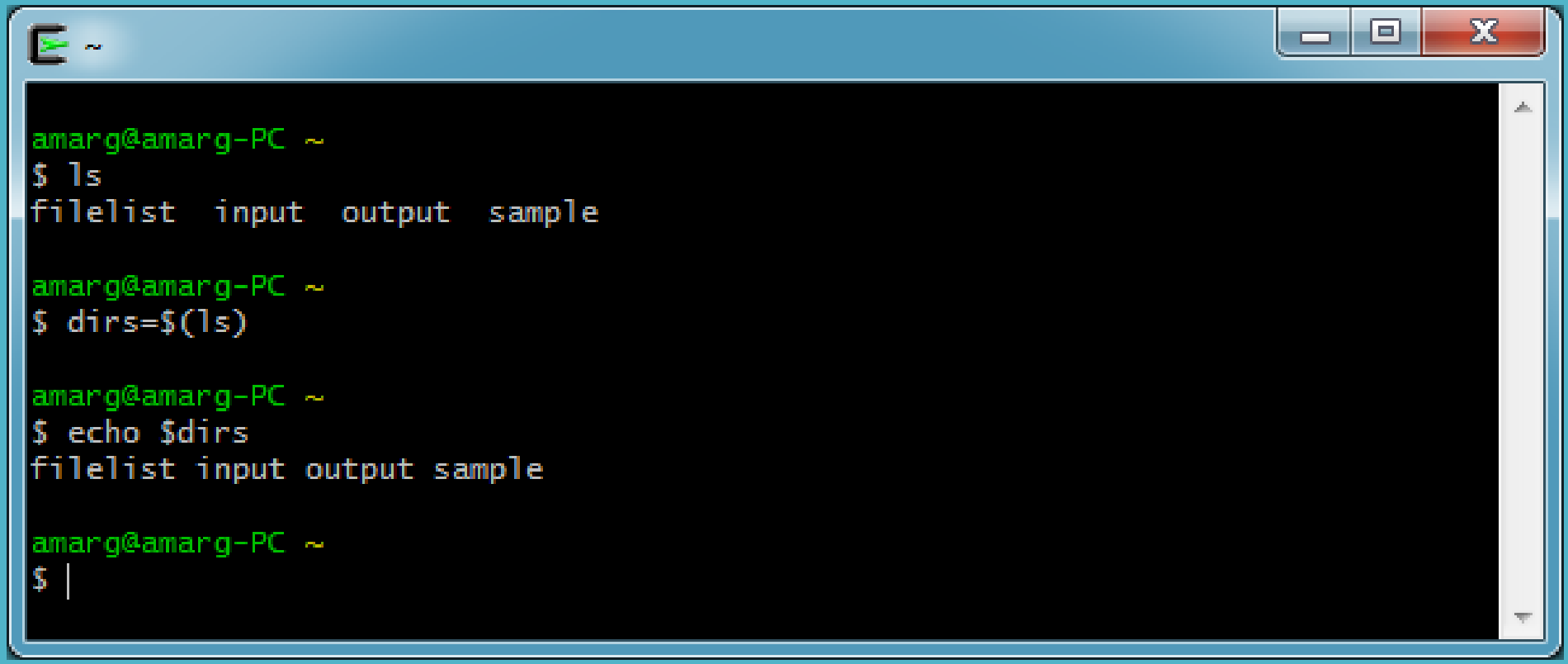
A Sample Script

```
#!/bin/bash

echo -n "Enter some text > "
read text
echo "You entered: $text"
```



Τοπικές μεταβλητές κελύφους

3) Εκχώρηση σε μεταβλητή της εξόδου εντολής



ΓΕΝΙΚΑ ➔ $VARIABLE=$(COMMAND)

Τοπικές μεταβλητές κελύφους

4) Αρχικοποίηση από τη γραμμή εντολών

Η κατάσταση είναι παρόμοια με τη C όπου τα ορίσματα της γραμμής εντολών αποθηκεύονται στα ορίσματα της συνάρτησης main

int main (int argc, char ** argv) ή
int main (int argc, char * argv [])

Έστω πως η main αποτελεί την κύρια συνάρτηση του κώδικα **fileMerge.c**

Εάν στη γραμμή εντολών γράψουμε **fileMerge file1 file2 targetFile**

τότε τα ορίσματα της main αρχικοποιούνται ως

argv [0] = “fileMerge”
argv [1] = “file1”
argv [2] = “file2”
argv [3] = “targeFile”

και **argc = 4**

Τοπικές μεταβλητές κελύφους

Έστω μία εντολή της μορφής

command arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9 arg10 arg11 ….

Το κέλυφος περιέχει δέκα μεταβλητές **$0, $1, $2, $3, $4, $5, $6, $7, $8, $9** οι οποίες αρχικοποιούνται ως εξής:

$0 → command ← argv [0]
$1 → arg1 ← argv [1]
$2 → arg2 ← argv [2]
$3 → arg3 ← argv [3]
$4 → arg4 ← argv [4]
$5 → arg5 ← argv [5]
$6 → arg6 ← argv [6]
$7 → arg7 ← argv [7]
$8 → arg8 ← argv [8]
$9 → arg9 ← argv [9]

Παράδειγμα → **find / -name linux –perm 755 –size 100**

$0 $1 $2 $3 $4 $5 $6 $7

Τοπικές μεταβλητές κελύφους

Η προσπέλαση του δέκατου ορίσματος και όσων υπάρχουν από εκει και πέρα γίνεται με επαναληπτική χρήση της **shift**

$0 → command		$0 → arg1		$0 → arg2		$0 → arg3
$1 → arg1		$1 → arg2		$1 → arg3		$1 → arg4
$2 → arg2		$2 → arg3		$2 → arg4		$2 → arg5
$3 → arg3		$3 → arg4		$3 → arg5		$3 → arg6
$4 → arg4	**shift**	$4 → arg5	**shift**	$4 → arg6	**shift**	$4 → arg7
$5 → arg5		$5 → arg6		$5 → arg7		$5 → arg8
$6 → arg6		$6 → arg7		$6 → arg8		$6 → arg9
$7 → arg7		$7 → arg8		$7 → arg9		$7 → arg10
$8 → arg8		$8 → arg9		$8 → arg10		$8 → arg11
$9 → arg9		$9 → arg10		$9 → arg11		$9 → arg12

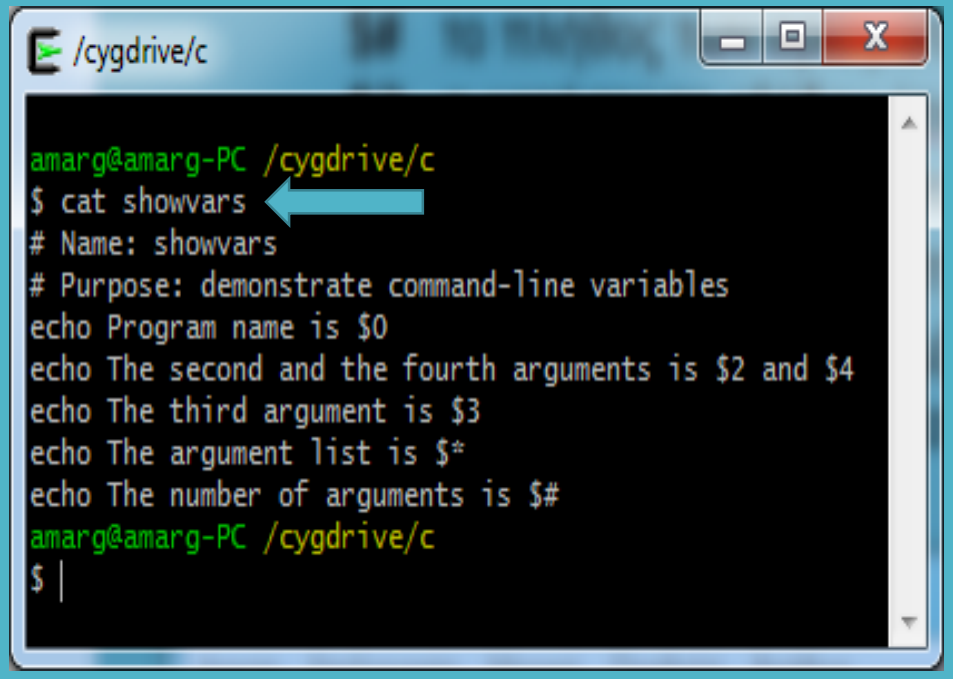
Εναλλακτικά μπορούμε να χρησιμοποιήσουμε άγκιστρα

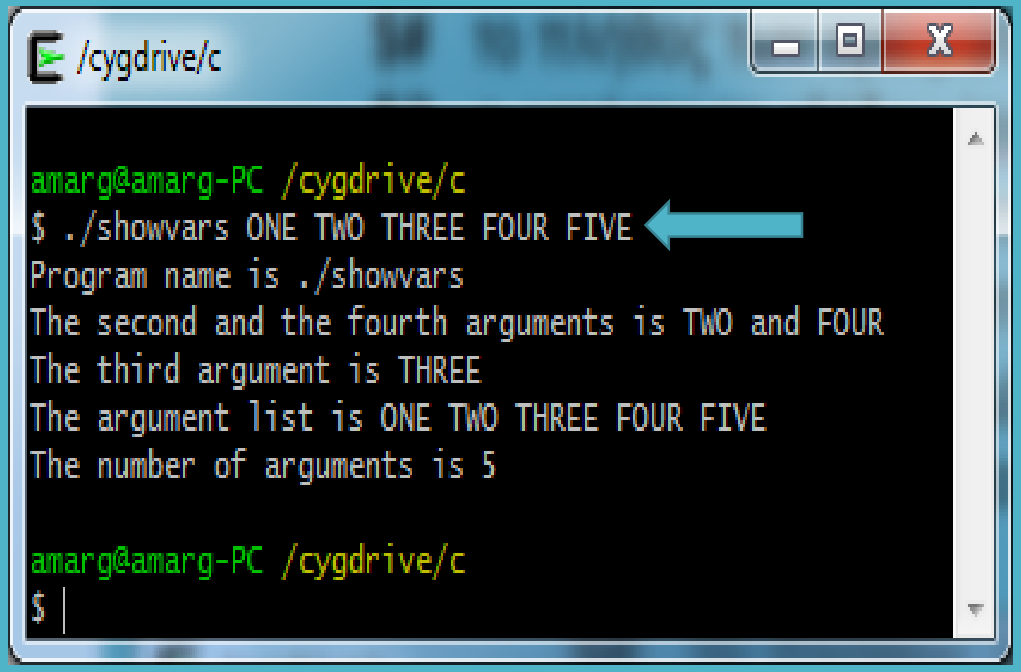
${10}, ${11}, ${12}

Τοπικές μεταβλητές κελύφους

Το σύνολο των μεταβλητών του κελύφους

$# το πλήθος των ορισμάτων θέσης
$? η κατάσταση εξόδου (exit status) της εντολής που εκτελέστηκε τελευταία
$$ ο αριθμός διεργασίας του φλοιού
$! ο αριθμός διεργασίας της διεργασίας που εκτελείται στο παρασκήνιο
$* ένα string που περιλαμβάνει όλα τα ορίσματα
$@ το ίδιο με το $*, εκτός αν χρησιμοποιούνται εισαγωγικά





Shell programming

Τα shell scripts είναι αρχεία κειμένου που περιέχουν κώδικα γραμμένο στη γλώσσα του φλοιού

```
#!/bin/bash
number=0
while [ $number -lt 10 ]; do
      echo "Number = $number"
      number=$((number+1))
done
~
~
~
~
~
~
~
~
"whileExample" 6 lines, 112 characters
```

Για να εκτελεστεί το shell script αρχικά το κάνουμε εκτελέσιμο ως

chmod +x whileExample

και μετά το εκτελούμε ως

./whileExample

Shell programming

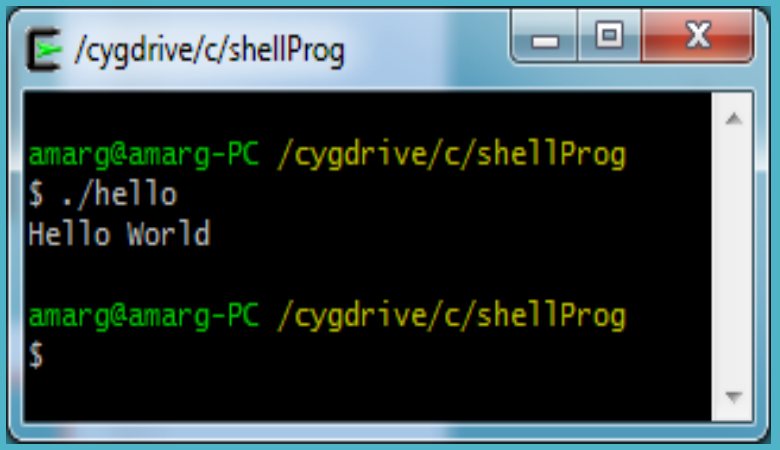
Μερικά απλά shell scripts

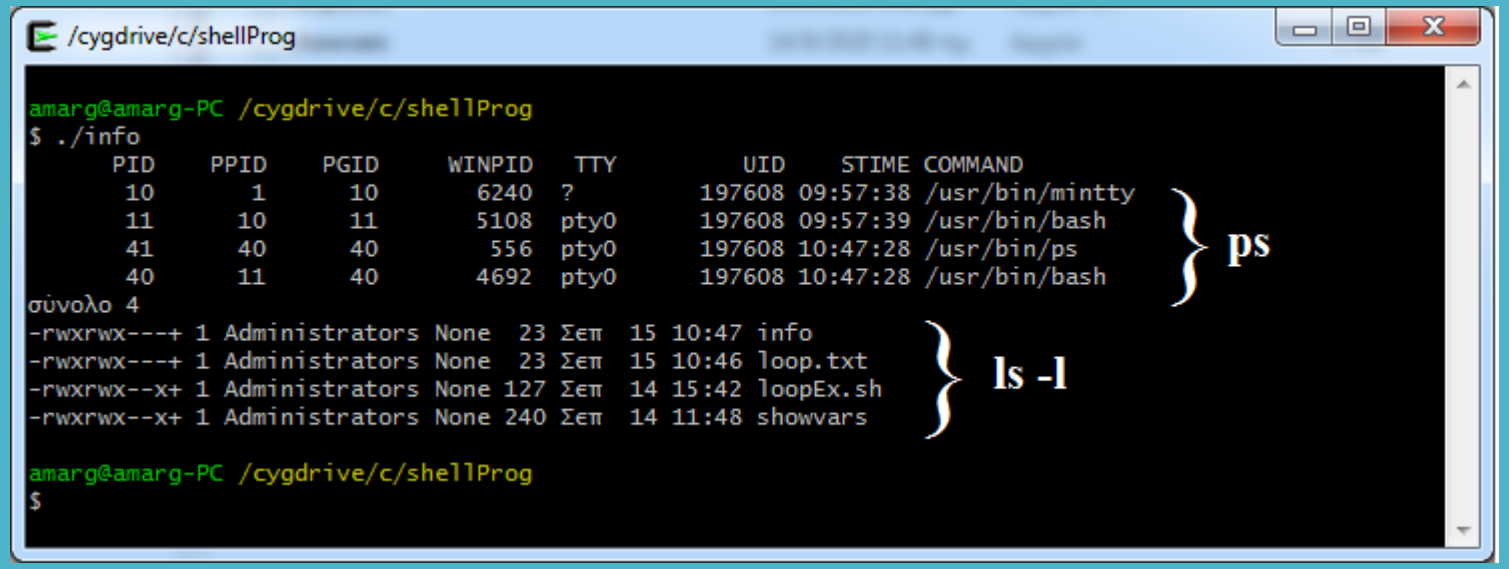
Εκτυπώνει το μήνυμα
Hello world

```
#!/bin/bash
echo "Hello World"
```

Εκτυπώνει την έξοδο των εντολών ps και ls -l

```
#!/bin/bash
ps;  ls -l
```





Shell programming

Τα προγράμματα φλοιού περιέχουν μία ή περισσότερες από τις παρακάτω δομές

- Μεταβλητές
- Λογικές δομές (if, case ...)
- Δομές επανάληψης (while, for, until)
- Συναρτήσεις (functions)
- Σχόλια

Shell programming

Αριθμητικοί και λογικοί τελεστές

+	Ο τελεστής της πρόσθεσης
-	Ο τελεστής της αφαίρεσης
*	Ο τελεστής του πολλαπλασιασμού
/	Ο τελεστής της διαίρεσης
%	Ο τελεστής του υπολοίπου (module) της διαίρεσης
<<	Ο τελεστής της αριστεράς μετατόπισης (left shift)
>>	Ο τελεστής της δεξιάς μετατόπισης (right shift)
<=	Ο τελεστής σύγκρισης «μικρότερο ή ίσο»
>=	Ο τελεστής σύγκρισης «μεγαλύτερο ή ίσο»
<	Ο τελεστής σύγκρισης «μικρότερο»
>	Ο τελεστής σύγκρισης «μεγαλύτερο»
==	Ο τελεστής ισότητας που χρησιμοποιείται στις συγκρίσεις
!=	Ο τελεστής ανισότητας που χρησιμοποιείται στις συγκρίσεις
&	Ο λογικός τελεστής & για πράξεις ανάμεσα σε bits (bitwise AND)
!	Ο λογικός τελεστής OR για πράξεις ανάμεσα σε bits (bitwise OR)
^	Ο λογικός τελεστής XOR για πράξεις ανάμεσα σε bits (bitwise XOR)
&&	Ο λογικός τελεστής AND
!!	Ο λογικός τελεστής OR

Shell programming

Τελεστές καταχώρησης

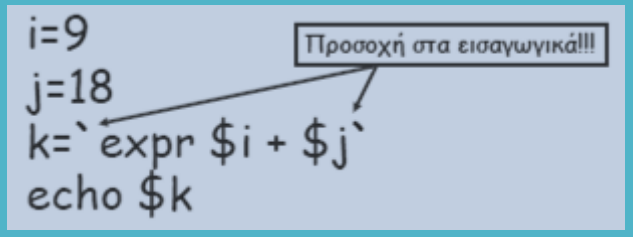
=	Καταχωρεί τιμή σε μεταβλητή του προγράμματος. Για παράδειγμα η πρόταση a=b καταχωρεί στην μεταβλητή a την τιμή της μεταβλητής b
+=	Πραγματοποιεί την πράξη της πρόσθεσης ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το αποτέλεσμα στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a+=b είναι ισοδύναμη με την πράξη a=a+b
-=	Πραγματοποιεί την πράξη της αφαίρεσης ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το αποτέλεσμα στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a-=b είναι ισοδύναμη με την πράξη a=a-b
*=	Πραγματοποιεί την πράξη του πολλαπλασιασμού ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το αποτέλεσμα στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a*=b είναι ισοδύναμη με την πράξη a=a*b
/=	Πραγματοποιεί την πράξη της διαίρεσης ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το αποτέλεσμα στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a/=b είναι ισοδύναμη με την πράξη a=a/b
%=	Πραγματοποιεί την πράξη της διαίρεσης ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το υπόλοιπο της διαίρεσης στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a%=b είναι ισοδύναμη με την πράξη a=a%b
<<=	Πραγματοποιεί την πράξη της αριστεράς μετατόπισης ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το αποτέλεσμα στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a<<=b είναι ισοδύναμη με την πράξη a=a<<b
>>=	Πραγματοποιεί την πράξη της δεξιάς μετατόπισης ανάμεσα στις τιμές δύο μεταβλητών και στη συνέχεια καταχωρεί το αποτέλεσμα στην μεταβλητή που βρίσκεται στο αριστερό μέλος της πρότασης. Έτσι η πράξη a>>=b είναι ισοδύναμη με την πράξη a=a>>b

Shell programming

Αριθμητικές πράξεις

```
#!/bin/bash

first_num=0
second_num=0

echo -n "Enter the first number --> "
read first_num
echo -n "Enter the second number -> "
read second_num

echo "first number + second number = $((first_num + second_num))"
echo "first number - second number = $((first_num - second_num))"
echo "first number * second number = $((first_num * second_num))"
echo "first number / second number = $((first_num / second_num))"
echo "first number % second number = $((first_num % second_num))"
echo "first number raised to the"
echo "power of the second number   = $((first_num ** second_num))"
```

Χρησιμοποιώντας την expr



Shell programming

Αριθμητικές πράξεις

```
~$ ./calc
Enter the first number --> 100
Enter the second number -> 5
first number + second number = 105
first number - second number = 95
first number * second number = 500
first number / second number = 20
first number % second number = 0
first number raised to the
power of the second number = 10000000000
~$
```

Εντολές διακλάδωσης

Δομή if – elif - fi

```
if expression;
   then
     commands;
   elif expression;
     then
     commands;
   elif expression;
     then
     commands  ...
   else
     commands;
fi
```

Numeric Comparison	Returns true (0) if:
[$num1 -eq $num2]	num1 equals num2
[$num1 -ne $num2]	num1 does not equal num2
[$num1 -lt $num2]	num1 is less than num2
[$num1 -gt $num2]	num1 is greater than num2
[$num1 -le $num2]	num1 is less than or equal to num2
[$num1 -ge $num2]	num1 is greater than or equal to num2

String Comparison	Returns true (0) if:
[str1 = str2]	str1 equals str2
[str1 != str2]	str1 does not equal str2
[str1 < str2]	str1 precedes str2 in lexical order
[str1 > str2]	str1 follows str2 in lexical order
[-z str1]	str1 has length zero (holds null value)
[-nstr1]	str1has nonzero length (contains one or more characters)

Εντολές διακλάδωσης

Παραδείγματα

```
~$ cat testIf
#!/bin/bash
echo -n "Enter a number --> "
read number
if [ $number -eq 100 ]
then
 echo "Number is equal to 100"
elif [ $number -lt 100 ]
then
 echo "Number is less than 100"
else
 echo "Number is greater than 100"
fi
~$
```

```
~$ ./testIf
Enter a number --> 200
Number is greater than 100
~$ ./testIf
Enter a number --> 100
Number is equal to 100
~$ ./testIf
Enter a number --> 35
Number is less than 100
~$
```

```
~$ cat testIf
#!/bin/bash
echo -n "Enter a number --> "
read number
if [ $number -gt 100 ] && [ $number -lt 200 ]
then
 echo "The number lies between 100 and 200"
else
 echo "Number lies outside the interval [100, 200]"
fi
~$
```

```
~$ ./testIf
Enter a number --> 150
The number lies between 100 and 200
~$ ./testIf
Enter a number --> 250
Number lies outside the interval [100, 200]
~$
```

Σύνθετες εντολές με τους λογικούς τελεστές && και ||

Εντολές διακλάδωσης

Παραδείγματα

Έλεγχος άρτιας ή περιττής τιμής

```
#!/bin/bash
number=0

echo -n "Enter a number > "
read number

echo "Number is $number"
if [ $((number % 2)) -eq 0 ]; then
    echo "Number is even"
else
    echo "Number is odd"
fi
```

Η εντολή case

Βασική σύνταξη

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    ...
esac
```

Πρόγραμμα

```
~$ cat testCase
#!/bin/sh

# take a number from user
echo "Enter number:"
read num
case $num in
  1)
    echo "It's one!"
    ;;
  2)
    echo "It's two!"
    ;;
  3)
    echo "It's three!"
    ;;
  *)
    echo "It's something else!"
    ;;
esac
echo "End of script."
~$
```

Έξοδος

```
~$ ./testCase
Enter number:
1
It's one!
End of script.
~$ ./testCase
Enter number:
2
It's two!
End of script.
~$ ./testCase
Enter number:
3
It's three!
End of script.
~$ ./testCase
Enter number:
4
It's something else!
End of script.
~$
```

Βρόχοι επανάληψης (for)

Βρόχος for

for i in λίστα λέξεων;
do
εντολές
done

```
#!/bin/bash
for i in January February March April May June; do
echo $i
done
~
```

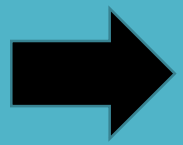
(το $i παίρνει διαδοχικά τις τιμές της λίστας λέξεων)

```
~$ ./months
January
February
March
April
May
June
~$
```

Βρόχοι επανάληψης (for)

Η λίστα των λέξεων στο βρόχο for μπορεί να προκύπτει από την εκτέλεση εντολής

```
~$ cat ./userNames
#!/bin/bash
for i in $(cat /etc/passwd | awk -F : '{print $1}'); do
echo $i
done
```



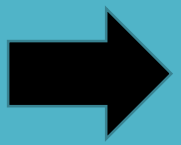
```
~$ ./userNames
root
daemon
bin
sys
sync
games
man
lp
mail
news
uucp
proxy
www-data
backup
list
irc
gnats
nobody
_apt
systemd-timesync
systemd-network
systemd-resolve
messagebus
salvus
ntp
sshd
rtkit
cups-pk-helper
usbmux
dnsmasq
avahi
geoclue
pulse
saned
colord
gdm
epmd
```

Υπολογισμός μεγέθους καταλόγου

```
~$ ls -l
total 5
-rw-r--r-- 1 user user   0 Sep 15 08:41 'Welcome to CoCalc.term'
-rwxr-xr-x 1 user user 573 Sep 15 08:25  calc
-rwxr-xr-x 1 user user  76 Sep 15 08:32  months
-rwxr-xr-x 1 user user 119 Sep 15 09:09  totalSize
-rwxr-xr-x 1 user user  81 Sep 15 08:46  userNames
~$
```

Υπολογισμός μεγέθους καταλόγου

```
~$ cat ./totalSize
#!/bin/bash
size=0
for i in $(ls -l | awk '{print $5}'); do
        echo $i
        size=$((size+i))
done
echo "Total size is $size"
~$
```

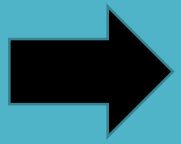


```
~$ ./totalSize
0
573
76
119
81
Total size is 849
~$
```

Βρόχοι επανάληψης (while)

```
while [ condition ]
do
        command1
        command2
        ..
        ....
        commandN
done
```

```
~$ cat testWhile
#!/bin/bash
number=0
while [ $number -lt 10 ]; do
echo "Number = $number"
number=$((number + 1))
done
~$
```



```
~$ ./testWhile
Number = 0
Number = 1
Number = 2
Number = 3
Number = 4
Number = 5
Number = 6
Number = 7
Number = 8
Number = 9
~$
```

Βρόχοι επανάληψης (while)

Internal Field Separator (IFS) ➔ :
Read (word segmentation of strings)

```
~$ 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
```

```
~$ cat readPasswd
#!/bin/bash
file=/etc/passwd
# set field delimiter to :
# read all 7 fields into 7 vars
while IFS=: read -r user enpass uid gid desc home shell
do
    # only display if UID >= 500
        [ $uid -ge 500 ] && echo "User $user ($uid) assigned \"$home\" home directory with $shell shell."
done < "$file"
~$
```

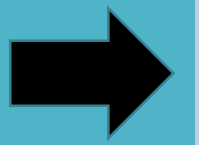
while there are data to read (return code is zero)

```
~$ ./readPasswd
User nobody (65534) assigned "/nonexistent" home directory with /usr/sbin/nologin shell.
User salvus (1000) assigned "/home/salvus" home directory with /bin/bash shell.
User user (2001) assigned "/home/user" home directory with /bin/bash shell.
User sbt (999) assigned "/home/sbt" home directory with /bin/false shell.
User rstudio-server (998) assigned "/home/rstudio-server" home directory with /bin/sh shell.
~$
```

Βρόχοι επανάληψης (until)

```
until [ condition ]
do
    command1
    command2
    ...
    ....
    commandN
done
```

```
~$ cat ./testUntil
#!/bin/bash
number=0
until [ $number -ge 10 ]; do
echo "Number = $number"
number=$((number + 1))
done
~$
```



```
~$ ./testUntil
Number = 0
Number = 1
Number = 2
Number = 3
Number = 4
Number = 5
Number = 6
Number = 7
Number = 8
Number = 9
~$
```

Η εντολή test

Expression	*Description*
-d *file*	True if *file* is a directory.
-e *file*	True if *file* exists.
-f *file*	True if *file* exists and is a regular file.
-L *file*	True if *file* is a symbolic link.
-r *file*	True if *file* is a file readable by you.
-w *file*	True if *file* is a file writable by you.
-x *file*	True if *file* is a file executable by you.
file1 -nt *file2*	True if *file1* is newer than (according to modification time) *file2*
file1 -ot *file2*	True if *file1* is older than *file2*
-z *string*	True if *string* is empty.
-n *string*	True if *string* is not empty.
string1 = *string2*	True if *string1* equals *string2.*
string1 != *string2*	True if *string1* does not equal *string2.*

Η εντολή test

```
#!/bin/bash
for filename in $@;
do
    if [ -f $filename ]; then
       result="$filename is a regular file"
    else
       if [ -d $filename ]; then
          result="$filename is a directory"
       fi
    fi
    if [ -w $filename ]; then
       result="$result and it is writable"
    else
       result="$result and it is not writable"
    fi
    echo "$result"
done
```

Παραδείγματα

Εκτύπωση του μήκους της κάθε λέξης ενός αρχείου κειμένου

```
#!/bin/bash
count=0
for i in $(cat testfile);
do
    count=$((count + 1))
    echo "Word $count ($i) contains $(echo -
  n $i | wc -c) characters"
done
```

Παραδείγματα

Εύρεση του μέσου όρου μιας ακολουθίας αριθμών που καταχωρούνται από το πληκτρολόγιο

```
#!/bin/bash
sum=0
num=0
while true; do
echo "-n enter a number [0-100] (0 for quit) :";read score
if [ $score -lt 0 ] || [ $score -gt 100 ]; then
    echo "try again!!"
elif [ $score -eq 0 ]; then
    echo "average = $average"
    exit 0
else
    sum=$((sum+score))
    num=$((num+1))
    average=$((sum/num))
fi
done
echo "exit - end"
```

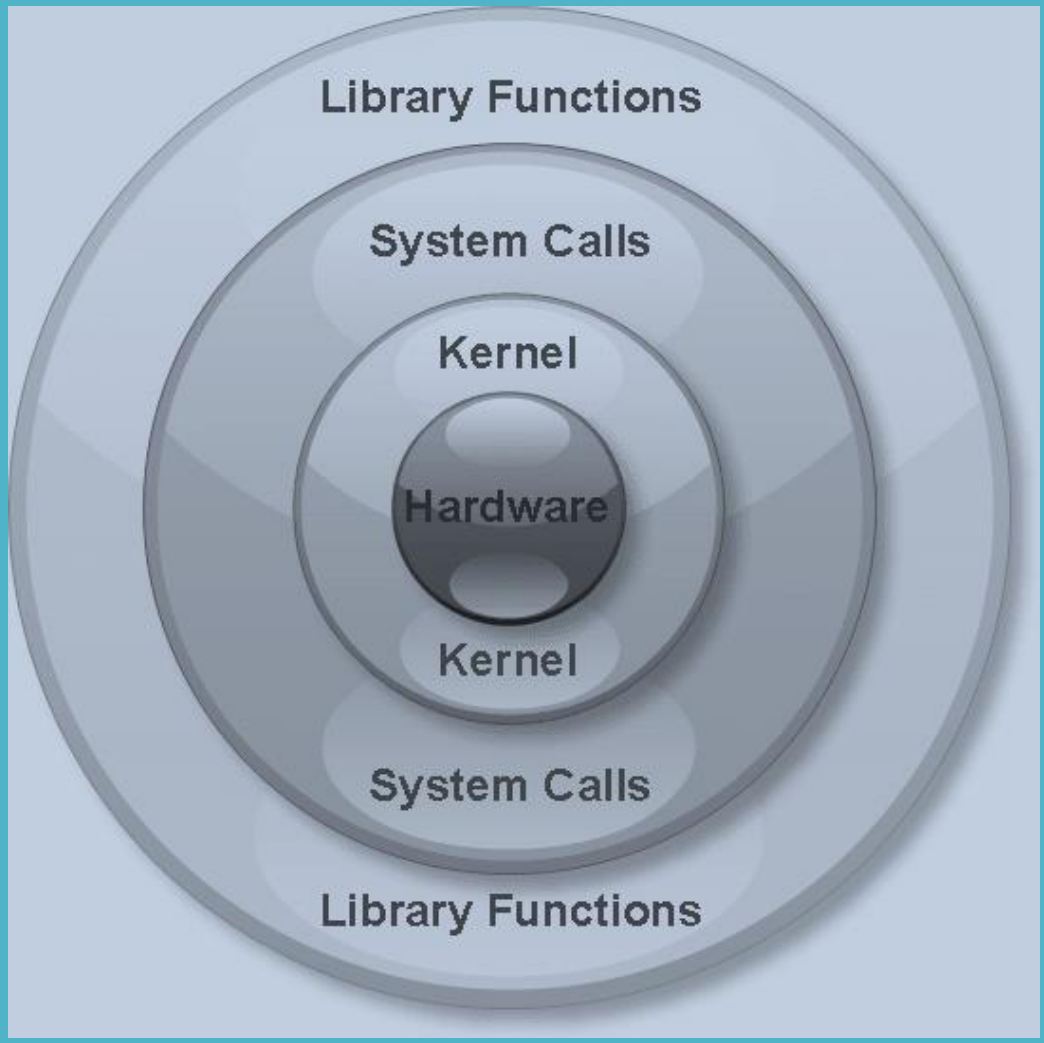
Παραδείγματα

Να γραφεί ένα shell script που θα δέχεται το login name ενός χρήστη και θα εμφανίζει πόσες φορές έχει κάνει logged on (χρήση των εντολών who, grep, wc)

ΛΥΣΗ

```
#!/bin/bash
times=`who | grep $1 | wc -l`
echo "$1 is logged on $times times"
```

Κλήσεις συστήματος



Κλήσεις συστήματος

Στο λειτουργικό σύστημα Linux οι πάσης φύσεως εφαρμογές εκτελούνται σε δύο διαφορετικές καταστάσεις λειτουργίας:

- Κατάσταση λειτουργίας χρήστη (User Mode) → σε αυτή την κατάσταση λειτουργίας εκτελούνται οι εφαρμογές των χρηστών.

- Κατάσταση λειτουργίας πυρήνα (Kernel Mode) → σε αυτή την κατάσταση λειτουργίας εκτελούνται οι οδηγοί συσκευών και άλλα κρίσιμα προγράμματα υπό καθεστώς αυξημένης προστασίας έτσι ώστε να μην προκαλείται κατάρρευση του συστήματος. Ο κώδικας που εκτελείται σε kernel mode έχει πρόσβαση σε όλο το σύστημα.

Ο στόχος των εφαρμογών που εκτελούνται σε kernel mode είναι η εξυπηρέτηση των εφαρμογών που εκτελούνται σε user mode.

Μία κλήση συστήματος είναι ο τρόπος με τον οποίο ένα πρόγραμμα που εκτελείται σε user mode ζητά εξυπηρέτηση από ένα πρόγραμμα που εκτελείται σε kernel mode.

Κλήσεις συστήματος

Με τις κλήσεις συστήματος μπορούμε να πραγματοποιήσουμε πολλές και διαφορετικές λειτουργίες, όπως είναι για παράδειγμα, οι ακόλουθες:

- Διαχείριση αρχείων → open, create, delete, κ.τ.λ.
- Έλεγχος διεργασιών → kill, wait, κ.τ.λ.
- Διαχείριση συσκευών → request, release, κ.τ.λ.
- Ανάκτηση και ορισμός τιμών παραμέτρων → set time, get time, κ.τ.λ.
- Διαχείριση επικοινωνιών και δικτύων → send, receive, κ.τ.λ

Κλήσεις συστήματος

Λόγω του καθεστώτος αυξημένης προστασίας στο kernel mode ο τρόπος χρήσης των κλήσεων συστήματος από τις συναρτήσεις του χρήστη είναι πολύ περιοριστικός:

- Επιτρέπεται μόνο η μεταβίβαση παραμέτρων σταθερού μήκους που είναι ίσο με το μήκος που χρησιμοποιείται για τους δείκτες (pointers).

- Ως κωδικοί επιστροφής για τις κλήσεις συστήματος χρησιμοποιούνται μόνο προσημασμένοι ακέραιοι. Ένας κωδικός επιστροφής ίσος με το μηδέν, υποδηλώνει επιτυχή ολοκλήρωση, ενώ ένας αρνητικός κωδικός παραπέμπει στην εκδήλωση σφάλματος.

Ο κωδικός επιστροφής της τελευταίας κλήσης συστήματος αποθηκεύεται στη μεταβλητή errno.

Υπάρχουν τρεις συναρτήσεις ανάγνωσης του σφάλματος:

perror () → εκτυπώνει ένα μήνυμα σφάλματος.
strerror () → επιστρέφει φράση σταθερού μήκους που περιγράφει το σφάλμα.
sys_errlist () → χρησιμοποιεί δείκτες σε μηνύματα συστήματος αλλά δεν είναι standard συνάρτηση στο Linux.

Κλήσεις συστήματος

Οι κωδικοί επιστροφής (συνήθως) είναι αποθηκευμένοι στο αρχείο /usr/include/asm/errno.h

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Arg list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD           10     /* No child processes */
#define EAGAIN           11     /* Try again */
#define ENOMEM           12     /* Out of memory */
#define EACCES           13     /* Permission denied */
#define EFAULT           14     /* Bad address */
#define ENOTBLK          15     /* Block device required */
#define EBUSY            16     /* Device or resource busy */
#define EEXIST           17     /* File exists */
#define EXDEV            18     /* Cross-device link */
#define ENODEV           19     /* No such device */
#define ENOTDIR          20     /* Not a directory */
#define EISDIR           21     /* Is a directory */
#define EINVAL           22     /* Invalid argument */
#define ENFILE           23     /* File table overflow */
#define EMFILE           24     /* Too many open files */
#define ENOTTY           25     /* Not a typewriter */
#define ETXTBSY          26     /* Text file busy */
#define EFBIG            27     /* File too large */
#define ENOSPC           28     /* No space left on device */
#define ESPIPE           29     /* Illegal seek */
#define EROFS            30     /* Read-only file system */
#define EMLINK           31     /* Too many links */
#define EPIPE            32     /* Broken pipe */
#define EDOM             33     /* Math argument out of domain of func */
#define ERANGE           34     /* Math result not representable */
#define EDEADLK          35     /* Resource deadlock would occur */
#define ENAMETOOLONG     36     /* File name too long */
#define ENOLCK           37     /* No record locks available */
#define ENOSYS           38     /* Function not implemented */
#define ENOTEMPTY        39     /* Directory not empty */
#define ELOOP            40     /* Too many symbolic links encountered */
#define EWOULDBLOCK      EAGAIN /* Operation would block */
#define ENOMSG           42     /* No message of desired type */
#define EIDRM            43     /* Identifier removed */
#define ECHRNG           44     /* Channel number out of range */
#define EL2NSYNC         45     /* Level 2 not synchronized */
#define EL3HLT           46     /* Level 3 halted */
#define EL3RST           47     /* Level 3 reset */
#define ELNRNG           48     /* Link number out of range */
#define EUNATCH          49     /* Protocol driver not attached */
#define ENOCSI           50     /* No CSI structure available */
#define EL2HLT           51     /* Level 2 halted */
#define EBADE            52     /* Invalid exchange */
#define EBADR            53     /* Invalid request descriptor */
#define EXFULL           54     /* Exchange full */
#define ENOANO           55     /* No anode */
#define EBADRQC          56     /* Invalid request code */
#define EBADSLT          57     /* Invalid slot */

#define EDEADLOCK        EDEADLK

#define EBFONT           59     /* Bad font file format */
#define ENOSTR           60     /* Device not a stream */
#define ENODATA          61     /* No data available */
#define ETIME            62     /* Timer expired */
#define ENOSR            63     /* Out of streams resources */
#define ENONET           64     /* Machine is not on the network */
#define ENOPKG           65     /* Package not installed */
#define EREMOTE          66     /* Object is remote */
#define ENOLINK          67     /* Link has been severed */
#define EADV             68     /* Advertise error */
#define ESRMNT           69     /* Srmount error */
#define ECOMM            70     /* Communication error on send */
#define EPROTO           71     /* Protocol error */
#define EMULTIHOP        72     /* Multihop attempted */
#define EDOTDOT          73     /* RFS specific error */
#define EBADMSG          74     /* Not a data message */
#define EOVERFLOW        75     /* Value too large for defined data type */
#define ENOTUNIQ         76     /* Name not unique on network */
#define EBADFD           77     /* File descriptor in bad state */
#define EREMCHG          78     /* Remote address changed */
#define ELIBACC          79     /* Can not access a needed shared library */
#define ELIBBAD          80     /* Accessing a corrupted shared library */
#define ELIBSCN          81     /* .lib section in a.out corrupted */
#define ELIBMAX          82     /* Attempting to link in too many shared libraries */
#define ELIBEXEC         83     /* Cannot exec a shared library directly */
#define EILSEQ           84     /* Illegal byte sequence */
#define ERESTART         85     /* Interrupted system call should be restarted */
#define ESTRPIPE         86     /* Streams pipe error */
#define EUSERS           87     /* Too many users */
#define ENOTSOCK         88     /* Socket operation on non-socket */
#define EDESTADDRREQ     89     /* Destination address required */
#define EMSGSIZE         90     /* Message too long */
#define EPROTOTYPE       91     /* Protocol wrong type for socket */
#define ENOPROTOOPT      92     /* Protocol not available */
#define EPROTONOSUPPORT  93     /* Protocol not supported */
#define ESOCKTNOSUPPORT  94     /* Socket type not supported */
#define EOPNOTSUPP       95     /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT     96     /* Protocol family not supported */
#define EAFNOSUPPORT     97     /* Address family not supported by protocol */
#define EADDRINUSE       98     /* Address already in use */
#define EADDRNOTAVAIL    99     /* Cannot assign requested address */
#define ENETDOWN         100    /* Network is down */
#define ENETUNREACH      101    /* Network is unreachable */
#define ENETRESET        102    /* Network dropped connection because of reset */
#define ECONNABORTED     103    /* Software caused connection abort */
#define ECONNRESET       104    /* Connection reset by peer */
#define ENOBUFS          105    /* No buffer space available */
#define EISCONN          106    /* Transport endpoint is already connected */
#define ENOTCONN         107    /* Transport endpoint is not connected */
#define ESHUTDOWN        108    /* Cannot send after transport endpoint shutdown */
#define ETOOMANYREFS     109    /* Too many references: cannot splice */
#define ETIMEDOUT        110    /* Connection timed out */
#define ECONNREFUSED     111    /* Connection refused */
#define EHOSTDOWN        112    /* Host is down */
#define EHOSTUNREACH     113    /* No route to host */
#define EALREADY         114    /* Operation already in progress */
#define EINPROGRESS      115    /* Operation now in progress */
#define ESTALE           116    /* Stale NFS file handle */
#define EUCLEAN          117    /* Structure needs cleaning */
#define ENOTNAM          118    /* Not a XENIX named type file */
#define ENAVAIL          119    /* No XENIX semaphores available */
#define EISNAM           120    /* Is a named type file */
#define EREMOTEIO        121    /* Remote I/O error */
#define EDQUOT           122    /* Quota exceeded */

#define ENOMEDIUM        123    /* No medium found */
#define EMEDIUMTYPE      124    /* Wrong medium type */
```

Κλήσεις συστήματος

Οι κλήσεις συστήματος καλούνται από τις συναρτήσεις του χρήστη

Παράδειγμα → άνοιγμα αρχείου

Χρησιμοποιώντας τη C → FILE * fopen (const char * path, const char * mode)

FILE * fp = fopen (“myfile.txt”, “r+”)

Χρησιμοποιώντας την κλήση συστήματος open

int open (const char * path, int flags)

int fd = open (“myfile.txt”, O_RDWR)

Η fopen είναι portable, η open γενικά όχι

LINUX System Call Quick Reference

Jialong He
Jialong_he@bigfoot.com
http://www.bigfoot.com/~jialong_he

Introduction

System call is the services provided by Linux kernel. In C programming, it often uses functions defined in **libc** which provides a wrapper for many system calls. Manual page section 2 provides more information about system calls. To get an overview, use “man 2 intro” in a command shell.

It is also possible to invoke **syscall()** function directly. Each system call has a function number defined in **<syscall.h>** or **<unistd.h>**. Internally, system call is invokded by software interrupt 0x80 to transfer control to the kernel. System call table is defined in Linux kernel source file “**arch/i386/kernel/entry.S**”.

System Call Example

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

int main(void) {

        long ID1, ID2;
        /*----------------------------*/
        /* direct system call         */
        /* SYS_getpid (func no. is 20) */
        /*----------------------------*/
        ID1 = syscall(SYS_getpid);
        printf ("syscall(SYS_getpid)=%ld\n", ID1);

        /*----------------------------*/
        /* "libc" wrapped system call  */
        /* SYS_getpid (Func No. is 20) */
        /*----------------------------*/
        ID2 = getpid();
        printf ("getpid()=%ld\n", ID2);

        return(0);
}
```

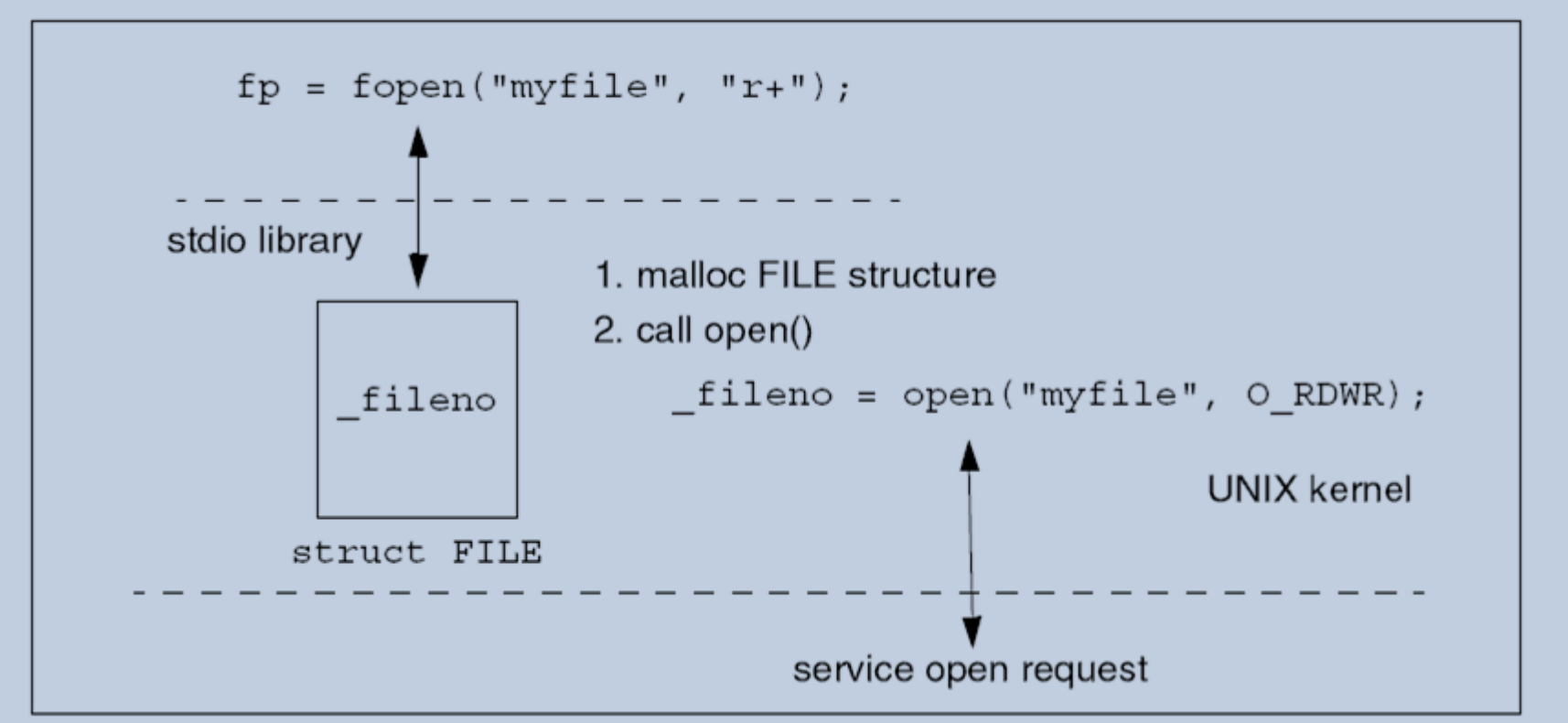
System Call Quick Reference

No	Func Name	Description	Source
1	exit	terminate the current process	*kernel/exit.c*
2	fork	create a child process	*arch/i386/kernel/process.c*
3	read	read from a file descriptor	*fs/read_write.c*
4	write	write to a file descriptor	*fs/read_write.c*
5	open	open a file or device	*fs/open.c*
6	close	close a file descriptor	*fs/open.c*
7	waitpid	wait for process termination	*kernel/exit.c*
8	creat	create a file or device ("man 2 open" for information)	*fs/open.c*
9	link	make a new name for a file	*fs/namei.c*
10	unlink	delete a name and possibly the file it refers to	*fs/namei.c*
11	execve	execute program	*arch/i386/kernel/process.c*
12	chdir	change working directory	*fs/open.c*
13	time	get time in seconds	*kernel/time.c*
14	mknod	create a special or ordinary file	*fs/namei.c*
15	chmod	change permissions of a file	*fs/open.c*
16	lchown	change ownership of a file	*fs/open.c*
18	stat	get file status	*fs/stat.c*
19	lseek	reposition read/write file offset	*fs/read_write.c*
20	getpid	get process identification	*kernel/sched.c*
21	mount	mount filesystems	*fs/super.c*
22	umount	unmount filesystems	*fs/super.c*
23	setuid	set real user ID	*kernel/sys.c*
24	getuid	get real user ID	*kernel/sched.c*
25	stime	set system time and date	*kernel/time.c*
26	ptrace	allows a parent process to control the execution of a child process	*arch/i386/kernel/ptrace.c*
27	alarm	set an alarm clock for delivery of a signal	*kernel/sched.c*
28	fstat	get file status	*fs/stat.c*
29	pause	suspend process until signal	*arch/i386/kernel/sys_i386.c*
30	utime	set file access and modification times	*fs/open.c*
33	access	check user's permissions for a file	*fs/open.c*
34	nice	change process priority	*kernel/sched.c*
36	sync	update the super block	*fs/buffer.c*
37	kill	send signal to a process	*kernel/signal.c*
38	rename	change the name or location of a file	*fs/namei.c*
39	mkdir	create a directory	*fs/namei.c*
40	rmdir	remove a directory	*fs/namei.c*
41	dup	duplicate an open file descriptor	*fs/fcntl.c*
42	pipe	create an interprocess channel	*arch/i386/kernel/sys_i386.c*
43	times	get process times	*kernel/sys.c*
45	brk	change the amount of space allocated for the calling process's data segment	*mm/mmap.c*
46	setgid	set real group ID	*kernel/sys.c*
47	getgid	get real group ID	*kernel/sched.c*
48	sys_signal	ANSI C signal handling	*kernel/signal.c*
49	geteuid	get effective user ID	*kernel/sched.c*
50	getegid	get effective group ID	*kernel/sched.c*

51	acct	enable or disable process accounting	*kernel/acct.c*
52	umount2	unmount a file system	*fs/super.c*
54	ioctl	control device	*fs/ioctl.c*
55	fcntl	file control	*fs/fcntl.c*
56	mpx	(unimplemented)	
57	setpgid	set process group ID	*kernel/sys.c*
58	ulimit	(unimplemented)	
59	olduname	obsolete uname system call	*arch/i386/kernel/sys_i386.c*
60	umask	set file creation mask	*kernel/sys.c*
61	chroot	change root directory	*fs/open.c*
62	ustat	get file system statistics	*fs/super.c*
63	dup2	duplicate a file descriptor	*fs/fcntl.c*
64	getppid	get parent process ID	*kernel/sched.c*
65	getpgrp	get the process group ID	*kernel/sys.c*
66	setsid	creates a session and sets the process group ID	*kernel/sys.c*
67	sigaction	POSIX signal handling functions	*arch/i386/kernel/signal.c*
68	sgetmask	ANSI C signal handling	*kernel/signal.c*
69	ssetmask	ANSI C signal handling	*kernel/signal.c*
70	setreuid	set real and effective user IDs	*kernel/sys.c*
71	setregid	set real and effective group IDs	*kernel/sys.c*
72	sigsuspend	install a signal mask and suspend caller until signal	*arch/i386/kernel/signal.c*
73	sigpending	examine signals that are blocked and pending	*kernel/signal.c*
74	sethostname	set hostname	*kernel/sys.c*
75	setrlimit	set maximum system resource con sumption	*kernel/sys.c*
76	getrlimit	get maximum system resource con sumption	*kernel/sys.c*
77	getrusage	get maximum system resource con sumption	*kernel/sys.c*
78	gettimeofday	get the date and time	*kernel/time.c*
79	settimeofday	set the date and time	*kernel/time.c*
80	getgroups	get list of supplementary group IDs	*kernel/sys.c*
81	setgroups	set list of supplementary group IDs	*kernel/sys.c*
82	old_select	sync. I/O multiplexing	*arch/i386/kernel/sys_i386.c*
83	symlink	make a symbolic link to a file	*fs/namei.c*
84	lstat	get file status	*fs/stat.c*
85	readlink	read the contents of a symbolic link	*fs/stat.c*
86	uselib	select shared library	*fs/exec.c*
87	swapon	start swapping to file/device	*mm/swapfile.c*
88	reboot	reboot or enable/disable Ctrl-Alt-Del	*kernel/sys.c*
89	old_readdir	read directory entry	*fs/readdir.c*
90	old_mmap	map pages of memory	*arch/i386/kernel/sys_i386.c*
91	munmap	unmap pages of memory	*mm/mmap.c*
92	truncate	set a file to a specified length	*fs/open.c*
93	ftruncate	set a file to a specified length	*fs/open.c*
94	fchmod	change access permission mode of file	*fs/open.c*
95	fchown	change owner and group of a file	*fs/open.c*
96	getpriority	get program scheduling priority	*kernel/sys.c*
97	setpriority	set program scheduling priority	*kernel/sys.c*
98	profil	execut ion time profile	
99	statfs	get file system statistics	*fs/open.c*
100	fstatfs	get file system statistics	*fs/open.c*
101	ioperm	set port input/output permissions	*arch/i386/kernel/ioport.c*
102	socketcall	socket system calls	*net/so cket.c*
103	syslog	read and/or clear kernel message ring buffer	*kernel/printk.c*
104	setitimer	set value of interval timer	*kernel/itimer.c*
105	getitimer	get value of interval timer	*kernel/itimer.c*
106	sys_newstat	get file status	*fs/stat.c*
107	sys_newlstat	get file status	*fs/stat.c*
108	sys_newfstat	get file status	*fs/stat.c*
109	olduname	get name and information about current kernel	*arch/i386/kernel/sys_i386.c*
110	iopl	change I/O privilege level	*arch/i386/kernel/ioport.c*
111	vhangup	virtually hangup the current tty	*fs/open.c*
112	idle	make process 0 idle	*arch/i386/kernel/process.c*
113	vm86old	enter virtual 8086 mode	*arch/i386/kernel/vm86.c*
114	wait4	wait for process termination, BSD style	*kernel/exit.c*
115	swapoff	stop swapping to file/device	*mm/swapfile.c*
116	sysinfo	returns information on overall system statistics	*kernel/info.c*
117	ipc	System V IPC system calls	*arch/i386/kernel/sys_i386.c*
118	fsync	synchronize a file's complete in-core state with that on disk	*fs/buffer.c*
119	sigreturn	return from signal handler and cleanup stack frame	*arch/i386/kernel/signal.c*
120	clone	create a child process	*arch/i386/kernel/process.c*
121	setdomainname	set domain name	*kernel/sys.c*
122	uname	get name and information about current kernel	*kernel/sys.c*
123	modify_ldt	get or set ldt	*arch/i386/kernel/ldt.c*
124	adjtimex	tune kernel clock	*kernel/time.c*
125	mprotect	set protection of memory mapping	*mm/mprotect.c*
126	sigprocmask	POSIX signal handling functions	*kernel/signal.c*
127	create_module	create a loadable module entry	*kernel/module.c*
128	init_module	initialize a loadable module entry	*kernel/module.c*
129	delete_module	delete a loadable module entry	*kernel/module.c*

130	get_kernel_syms	retrieve exported kernel and module symbols	*kernel/module.c*
131	quotactl	manipulate disk quotas	*fs/dquot.c*
132	getpgid	get process group ID	*kernel/sys.c*
133	fchdir	change working directory	*fs/open.c*
134	bdflush	start, flush, or tune buffer-dirty-flush daemon	*fs/buffer.c*
135	sysfs	get file system type information	*fs/super.c*
136	personality	set the process execution domain	*kernel/exec_domain.c*
137	afs_syscall	(unimplemented)	
138	setfsuid	set user identity used for file system checks	*kernel/sys.c*
139	setfsgid	set group identity used for file system checks	*kernel/sys.c*
140	sys_llseek	move extended read/write file pointer	*fs/read_write.c*
141	getdents	read directory entries	*fs/readdir.c*
142	select	sync. I/O multiplexing	*fs/select.c*
143	flock	apply or remove an advisory lock on an open file	*fs/locks.c*
144	msync	synchronize a file with a memory map	*mm/filemap.c*
145	readv	read data into multiple buffers	*fs/read_write.c*
146	writev	write data into multiple buffers	*fs/read_write.c*
147	sys_getsid	get process group ID of session leader	*kernel/sys.c*
148	fdatasync	synchronize a file's in-core data with that on disk	*fs/buffer.c*
149	sysctl	read/write system parameters	*kernel/sysctl.c*
150	mlock	lock pages in memory	*mm/mlock.c*
151	munlock	unlock pages in memory	*mm/mlock.c*
152	mlockall	disable paging for calling process	*mm/mlock.c*
153	munlockall	reenable paging for calling process	*mm/mlock.c*
154	sched_setparam	set scheduling parameters	*kernel/sched.c*
155	sched_getparam	get scheduling parameters	*kernel/sched.c*
156	sched_setscheduler	set scheduling algorithm parameters	*kernel/sched.c*
157	sched_getscheduler	get scheduling algorithm parameters	*kernel/sched.c*
158	sched_yield	yield the processor	*kernel/sched.c*
159	sched_get_priority_max	get max static priority range	*kernel/sched.c*
160	sched_get_priority_min	get min static priority range	*kernel/sched.c*
161	sched_rr_get_interval	get the SCHED_RR interval for the named process	*kernel/sched.c*
162	nanosleep	pause execution for a specified time (nano seconds)	*kernel/sched.c*
163	mremap	re-map a virtual memory address	*mm/mremap.c*
164	setresuid	set real, effective and saved user or group ID	*kernel/sys.c*
165	getresuid	get real, effective and saved user or group ID	*kernel/sys.c*
166	vm86	enter virtual 8086 mode	*arch/i386/kernel/vm86.c*
167	query_module	query the kernel for various bits pertain ing to modules	*kernel/module.c*
168	poll	wait for some event on a file descriptor	*fs/select.c*
169	nfsservctl	syscall interface to kernel nfs daemon	*fs/filesystems.c*
170	setresgid	set real, effective and saved user or group ID	*kernel/sys.c*
171	getresgid	get real, effective and saved user or group ID	*kernel/sys.c*
172	prctl	operations on a process	*kernel/sys.c*
173	rt_sigreturn		*arch/i386/kernel/signal.c*
174	rt_sigaction		*kernel/signal.c*
175	rt_sigprocmask		*kernel/signal.c*
176	rt_sigpending		*kernel/signal.c*
177	rt_sigtimedwait		*kernel/signal.c*
178	rt_sigqueueinfo		*kernel/signal.c*
179	rt_sigsuspend		*arch/i386/kernel/signal.c*
180	pread	read from a file descriptor at a given offset	*fs/read_write.c*
181	sys_pwrite	write to a file descriptor at a given offset	*fs/read_write.c*
182	chown	change ownership of a file	*fs/open.c*
183	getcwd	Get current working directory	*fs/dcache.c*
184	capget	get process capabilities	*kernel/capability.c*
185	capset	set process capabilities	*kernel/capability.c*
186	sigaltstack	set/get signal stack context	*arch/i386/kernel/signal.c*
187	sendfile	transfer data between file descriptors	*mm/filemap.c*
188	getpmsg	(unimplemented)	
189	putpmsg	(unimplemented)	
190	vfork	create a child process and block parent	*arch/i386/kernel/process.c*

Κλήσεις συστήματος



Η fopen (user mode) καλεί την open (kernel mode)

Κλήσεις συστήματος

FILE * fp = fopen (“myfile.txt”, “r+”)

```
typedef struct
{
    int level;
    unsigned flags;
    char fd;
    unsigned char hold;
    int bsize;
    unsigned char_FAR* buffer;
    unsigned char_FAR* curp;
    unsigned istemp;
    short token;
}FILE;
```

int fd = open (“myfile.txt”, O_RDWR)

Το πεδίο fd (file descriptor) που επιστρέφει η open δεν είναι παρά ένα από τα πεδία της δομής FILE που επιστρέφει η fopen η οποία επομένως προσφέρει περισσότερες δυνατότητες διαχείρισης του αρχείου.

Κλήσεις συστήματος

- Όταν λοιπόν ένα πρόγραμμα πραγματοποιεί μία κλήση συστήματος, η λειτουργία του διακόπτεται και το σύστημα μεταφέρεται σε kernel mode.
- Η τρέχουσα κατάσταση της διεργασίας αποθηκεύεται έτσι ώστε αργότερα να συνεχίσει από εκεί που σταμάτησε.
- Ο πυρήνας προσδιορίζει τι ακριβώς είναι αυτό που έχει ζητηθεί και εάν το αίτημα της διεργασίας είναι έγκυρο και εάν η διεργασία που ζήτησε την εξυπηρέτηση δικαιούται να το κάνει.
- Εάν πληρούνται οι παραπάνω προϋποθέσεις, ο πυρήνας προσπελαύνει το κατάλληλο σε κάθε περίπτωση hardware μέσω των οδηγών συσκευών για την εξυπηρέτηση του αιτήματος.
- Όταν ολοκληρωθεί η εξυπηρέτηση του αιτήματος, ο πυρήνας αποκαθιστά την τρέχουσα κατάσταση της διεργασίας έτσι ώστε αυτή να συνεχίσει από εκεί που σταμάτησε και επιστρέφει τον έλεγχο στη διεργασία – το σύστημα επιστρέφει ξανά σε user mode.

Δομές και δείκτες στη C

Στη γλώσσα προγραμματισμού C μια δομή επιτρέπει τον ορισμό ενός σύνθετου τύπου δεδομένων.

ΠΑΡΑΔΕΙΓΜΑ → Ένα σημείο στις τρεις διαστάσεις περιγράφεται από συντεταγμένες x, y, z.

Για την περιγραφή ενός σημείου έχουμε δύο επιλογές: (α) να ορίσουμε τρεις ανεξάρτητες μεταξύ τους μεταβλητές του κατάλληλου τύπου π.χ.

```
double x, y, z;
```

και να προχωρήσουμε στην αρχικοποίησή τους π.χ. γράφοντας

```
x=10;   y=20;   z=5;
```

ή να ορίσουμε μία δομή με όνομα point ως

```
struct point { double x; double y; double z;  };
```

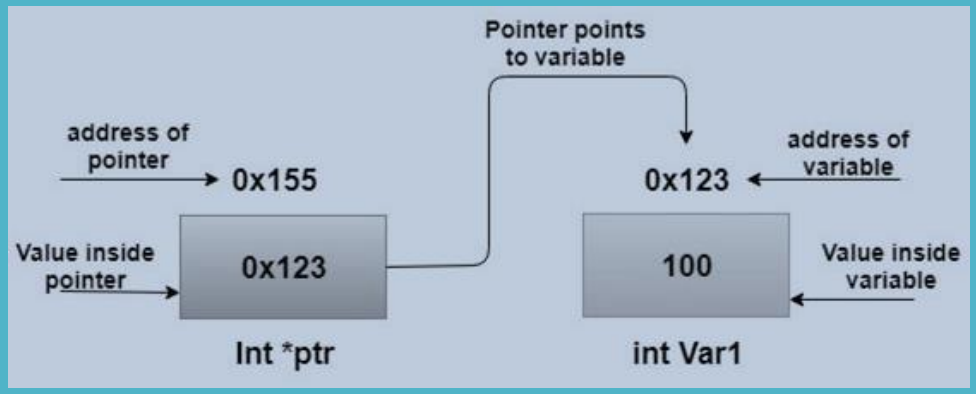
και αφού δηλώσουμε ένα point ως struct point p; να γράψουμε

```
p.x=10;      p.y=20;      p.z=5
```

όπου τώρα είναι εμφανές πως οι συντεταγμένες x, y και z αφορούν το ίδιο σημείο p.

Δομές και δείκτες στη C

Στη γλώσσα προγραμματισμού C ένας δείκτης (pointer) είναι μία μεταβλητή το περιεχόμενο της οποίας είναι η διεύθυνση μιας άλλης μεταβλητής.



Στη γλώσσα C ένας δείκτης προς μία μεταβλητή δηλώνεται χρησιμοποιώντας το σύμβολο * ενώ η διεύθυνση της μεταβλητής δηλώνεται χρησιμοποιώντας το σύμβολο & - θα είναι για παράδειγμα,

```
int Var1 = 100;
int *ptr = &Var1;
```

Θεωρώντας έναν πίνακα μεταβλητών κάποιου τύπου, π.χ έναν πίνακα ακεραίων, αυτός είναι ισοδύναμος με έναν δείκτη που δείχνει στο πρώτο κελί του πίνακα. Για παράδειγμα, εάν δηλώσουμε έναν πίνακα δέκα ακεραίων της μορφής int A [10], τα σύμβολα A και &A[0] εκφράζουν το ίδιο πράγμα.

Δομές και δείκτες στη C

Οι δείκτες αποτελούν σημαντικότατα στοιχεία της γλώσσας C αφού επιτρέπουν την πραγματοποίηση εξαιρετικά σημαντικών διαδικασιών που μεταξύ άλλων περιλαμβάνουν:

- Άμεση πρόσβαση στη μνήμη.
- Τη δυνατότητα επιστροφής περισσότερων από μία τιμές
- Τη μεταβίβαση πληροφοριών από την καλούσα συνάρτηση προς τη συνάρτηση που καλείται και αντίστροφα.
- Τη δυναμική δέσμευση και αποδέσμευση μνήμης.

Pass by value & Pass by reference

Θεωρώντας μία μεταβλητή π.χ. int n=5, μία μεταβλητή αναφοράς (reference variable) δεν είναι παρά ένα ψευδώνυμο αυτής της μεταβλητής που δηλώνεται ως int & r = n.

Αυτές οι μεταβλητές επιτρέπουν τη μεταβίβαση πληροφοριών ανάμεσα σε δύο συναρτήσεις func1 και func2 οι οποίες αλληλεπιδρούν καλώντας η μία την άλλη.

Οι παραπάνω αρχές ισχύουν στη γλώσσα C++ ενώ στην απλή C η αναφορά στις μεταβλητές γίνεται χρησιμοποιώντας δείκτες προς αυτές.

Δομές και δείκτες στη C

Παράδειγμα → η main καλεί τη συνάρτηση Twice που διπλασιάζει τις τιμές των ορισμάτων της

```c
#include <stdio.h>

void Twice (int a, int b)  {
      a *= 2;
      b *= 2; }

int main() {
     int x = 5, y = 8;
     printf ("Old values ==> x=%d, y=%d\n", x,y);
     Twice(x,y);
     printf ("New values ==> x=%d, y=%d\n", x,y);
     return (0); }
```

```
amarg@amarg-vbox:~$ ./twice
Old values ==> x=5, y=8
New values ==> x=5, y=8
amarg@amarg-vbox:~$
```

Παρατηρήστε πως η τιμή των μεταβλητών x και y δεν άλλαξαν τιμή! Σε αυτόν τον τύπο κλήσης (pass by value), οι παράμετροι a και b της συνάρτησης λαμβάνουν αντίγραφα των μεταβλητών x και y. Ωστόσο, επειδή είναι τοπικές μεταβλητές που παύουν να υφίστανται όταν τερματιστεί η συνάρτηση, δεν έχουν καμία επίδραση στη συνάρτηση main που κάλεσε την Twice.

```
amarg@amarg-vbox:~$ ./twice
Old values ==> x=5, y=8
Old values ==> a=5, b=8
Old values ==> a=10, b=16
New values ==> x=5, y=8
amarg@amarg-vbox:~$
```

Δομές και δείκτες στη C

Παράδειγμα → η main καλεί τη συνάρτηση Twice που διπλασιάζει τις τιμές των ορισμάτων της

```c
#include <stdio.h>

void Twice (int * a, int * b)  {
      printf ("Old values ==> a=%d, b=%d\n", *a,*b);
      (*a) *= 2;
      (*b) *= 2;
      printf ("Old values ==> a=%d, b=%d\n", *a,*b); }

int main() {
     int x = 5, y = 8;
     printf ("Old values ==> x=%d, y=%d\n", x,y);
     Twice(&x,&y);
     printf ("New values ==> x=%d, y=%d\n", x,y);
     return (0); }
```

```
amarg@amarg-vbox:~$ ./twice
Old values ==> x=5, y=8
Old values ==> a=5, b=8
Old values ==> a=10, b=16
New values ==> x=10, y=16
amarg@amarg-vbox:~$
```

Στην προκειμένη περίπτωση (pass by reference) εκείνο που αντιγράφεται στις μεταβλητές a και b δεν είναι οι τιμές των x και y αλλά οι διευθύνσεις αυτών των τιμών, οι οποίες αποτελούν αναφορές προς τις θέσεις αποθήκευσης των πραγματικών μεταβλητών. Με άλλα λόγια, δεν χρησιμοποιείται αντίγραφο των x και y αλλά τα πραγματικά x και y των οποίων επομένως οι τιμές μεταβάλλονται.

Δομές και δείκτες στη C

Εναλλακτικά μπορούμε να επιστρέψουμε το αποτέλεσμα στην καλούσα συνάρτηση χρησιμοποιώντας τη δεσμευμένη λέξη return όπως φαίνεται στη συνέχεια.

```c
#include <stdio.h>

int sum (int a, int b)  {
     int c;
     c = a + b;
     return c; }

int main() {
     int x = 5, y = 8, z = 0;
     z = sum (x, y);
     printf ("z=%d\n", z);
     return (0); }
```

```
amarg@amarg-vbox:~$ ./pbyval
z=13
amarg@amarg-vbox:~$ _
```

Γιατί τώρα το αποτέλεσμα είναι σωστό παρά το γεγονός πως χρησιμοποιούμε τοπικές μεταβλητές? Διότι, πολύ απλά, αυτό υπολογίστηκε πριν την επιστροφή της συνάρτησης sum και εκείνο που επιστράφηκε στην καλούσα συνάρτηση είναι ένα αντίγραφο της υπολογιζόμενης τιμής.

Αυτός ο μηχανισμός επιστροφής αποτελέσματος (return by value) είναι ο πλέον κατάλληλος για την επιστροφή των τιμών μεταβλητών που έχουν δηλωθεί εντός της συνάρτησης ή για την επιστροφή των τιμών ορισμάτων που έχουν διαβιβαστεί με τη μέθοδο pass by value. Ωστόσο δεν χρησιμοποιείται για μεγάλες δομές επειδή είναι σχετικά αργός (εκεί προτιμάται η επιστροφή της διεύθυνσης της τιμής [return by address] όπου δεν επιστρέφεται η μεταβλητή αλλά η διεύθυνσή της στη μνήμη).

Διαχείριση σφαλμάτων

Η συνάρτηση perror (message) εκτυπώνει την τιμή του ορίσματος message μαζί με μία περιγραφή του τελευταίου σφάλματος που προέκυψε και το οποίο έχει αποθηκευτεί στη μεταβλητή errno.

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main () {
   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL) {
      errnum = errno;
      fprintf (stderr, "Value of errno: %d\n", errno);
      perror ("Error printed by perror");
      fprintf (stderr, "Error opening file: %s\n",
               strerror( errnum )); }
   else fclose (pf);
   return 0; }
```

```
amarg@amarg-vbox:~$ ./errDem
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
amarg@amarg-vbox:~$
```

Στο συγκεκριμένο παράδειγμα προέκυψε σφάλμα εισόδου – εξόδου επειδή ο χρήστης προσπάθησε να ανοίξει για ανάγνωση ένα αρχείο που δεν υπάρχει.

stderr → είναι ο προεπιλεγμένος προορισμός για την αποθήκευση των σφαλμάτων της εφαρμογής. Αντιστοιχεί στην τιμή fd = 2 και για τις εντολές που εκτελούνται από την κονσόλα είναι η οθόνη του τερματικού.

EXIT_SUCCESS → 0 παραπέμπει σε επιτυχή λειτουργία
EXIT_FAILURE→ 1 παραπέμπει σε αποτυχία

Οι κωδικοί errno ΔΕΝ ΤΑΥΤΙΖΟΝΤΑΙ με τους κωδικούς επιστροφής της main !!

Διαχείριση διεργασιών

Linux Process

Management

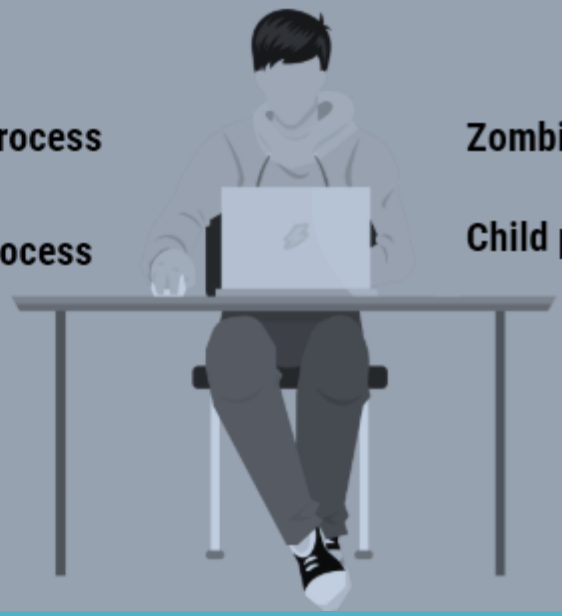
Daemon process

Orphan process

Zombie process

Parent process

Child process



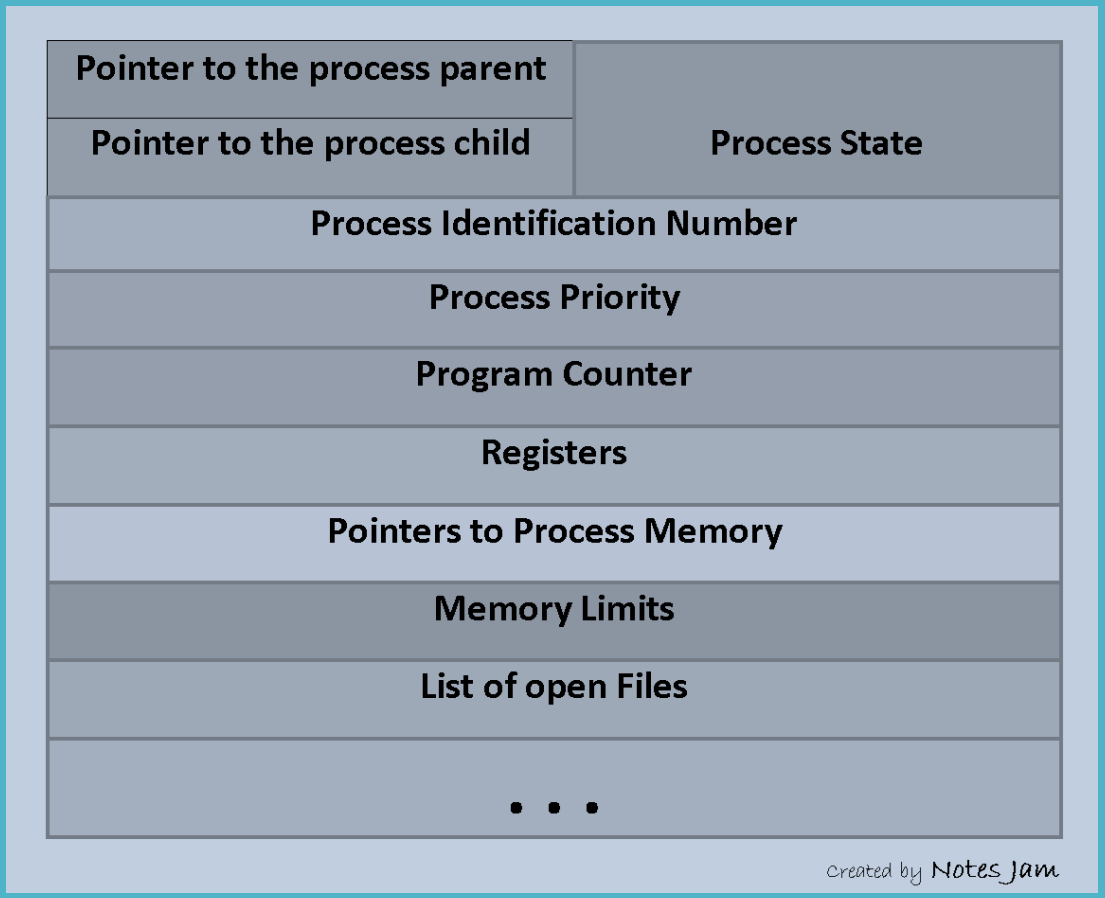
Διαχείριση διεργασιών

Μιλώντας γενικά, οποιοδήποτε πρόγραμμα εκτελείται σε ένα υπολογιστικό σύστημα, είναι γνωστό ως διεργασία (process). Η κάθε διεργασία εκτελείται ανεξάρτητα από τα άλλα προγράμματα και χρησιμοποιεί τους δικούς της πόρους.

Για κάθε διεργασία που εκτελείται σε ένα υπολογιστικό σύστημα, το λειτουργικό διατηρεί και ενημερώνει συνεχώς μια ειδική δομή δεδομένων που είναι γνωστή ως

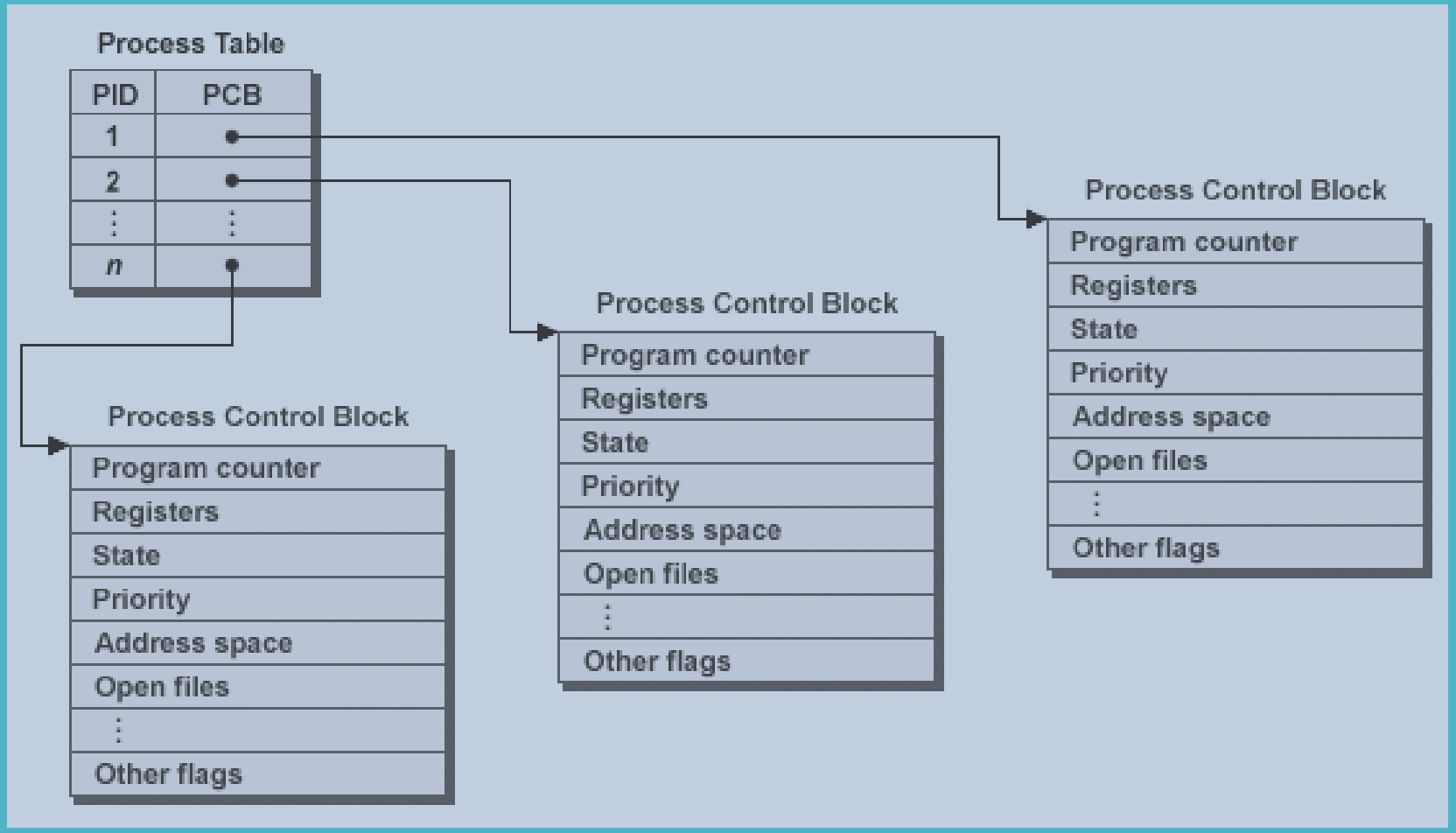
Ομάδα Ελέγχου Διεργασίας

(Process Control Block, PCB)



Διαχείριση διεργασιών

Το κάθε λειτουργικό σύστημα διατηρεί και ενημερώνει συνεχώς έναν πίνακα διεργασιών (process table) ο οποίος για κάθε διεργασία που εκτελείται στο σύστημα περιέχει το Process Control Block της.



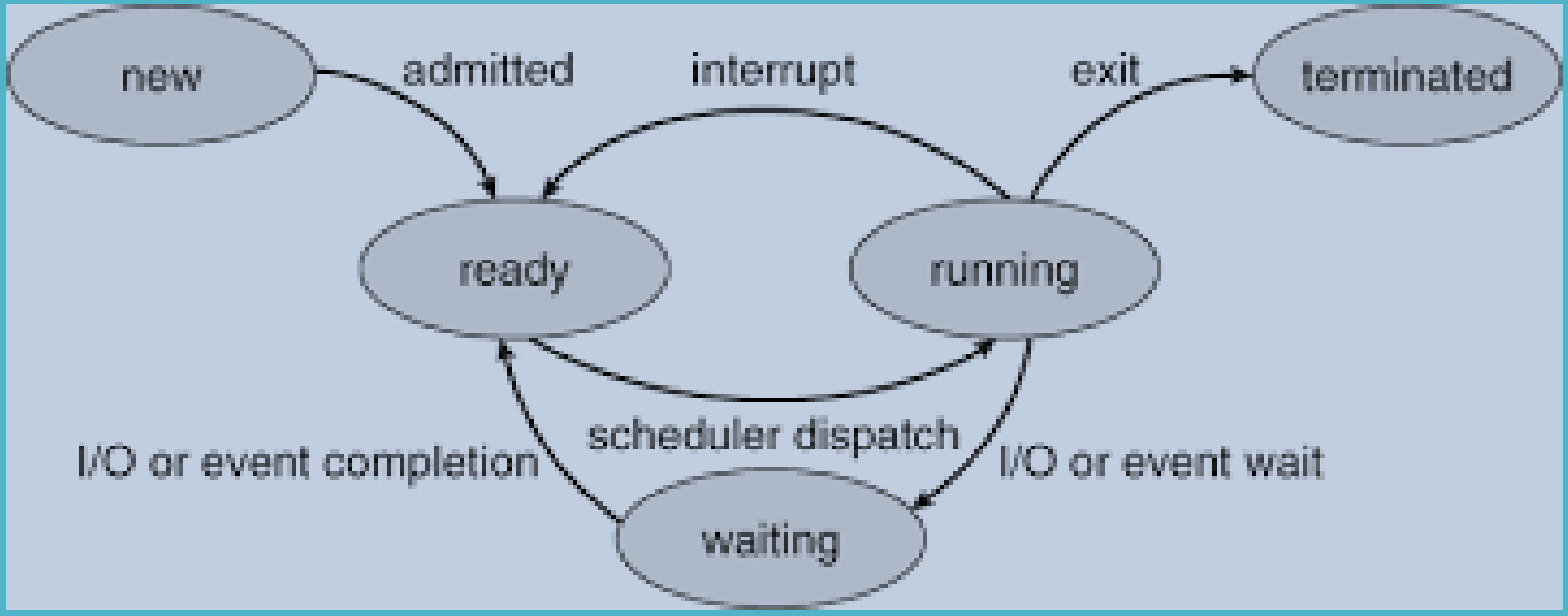
Διαχείριση διεργασιών

Οι πιο σημαντικές μορφές διαχείρισης διεργασιών από ένα λειτουργικό σύστημα, περιλαμβάνουν:

- Δημιουργία διεργασίας (creation, new process) από το σύστημα ή τους χρήστες.
- Τερματισμός διεργασίας λόγω ολοκλήρωσης (completion) ή κρίσιμου σφάλματος (termination) με ταυτόχρονη αποδέσμευση πόρων.
- Αναστολή διεργασίας (susppend) στα πλαίσια εφαρμογής της εφαρμοζόμενης πολιτικής χρονοπρογραμματισμού.
- Επαναφορά διεργασίας (resume) που έχει ανασταλεί.
- Μεταβολή της τιμής της προτεραιότητας διεργασίας (change process priority).
- Κλείδωμα διεργασίας (block) σε αναμονή κάποιου συμβάντος.
- Ενεργοποίηση κλειδωμένης διεργασίας (wakeup) που την επαναφέρει στην κατάσταση ετοιμότητας.
- Διανομή διεργασίας (dispatch) που τη μεταφέρει από την κατάσταση ετοιμότητας στην κατάσταση εκτέλεσης (δηλαδή της εκχωρεί τον επεξεργαστή για να εκτελεστεί).
- Επικοινωνία με άλλες διεργασίες (inter-process communication).

Διαχείριση διεργασιών

Κύκλος ζωής διεργασίας σε λειτουργικό σύστημα (process state diagram)



Μεταγωγή περιβάλλοντος (Context Switching)

Διαχείριση διεργασιών

Νήματα (Threads)

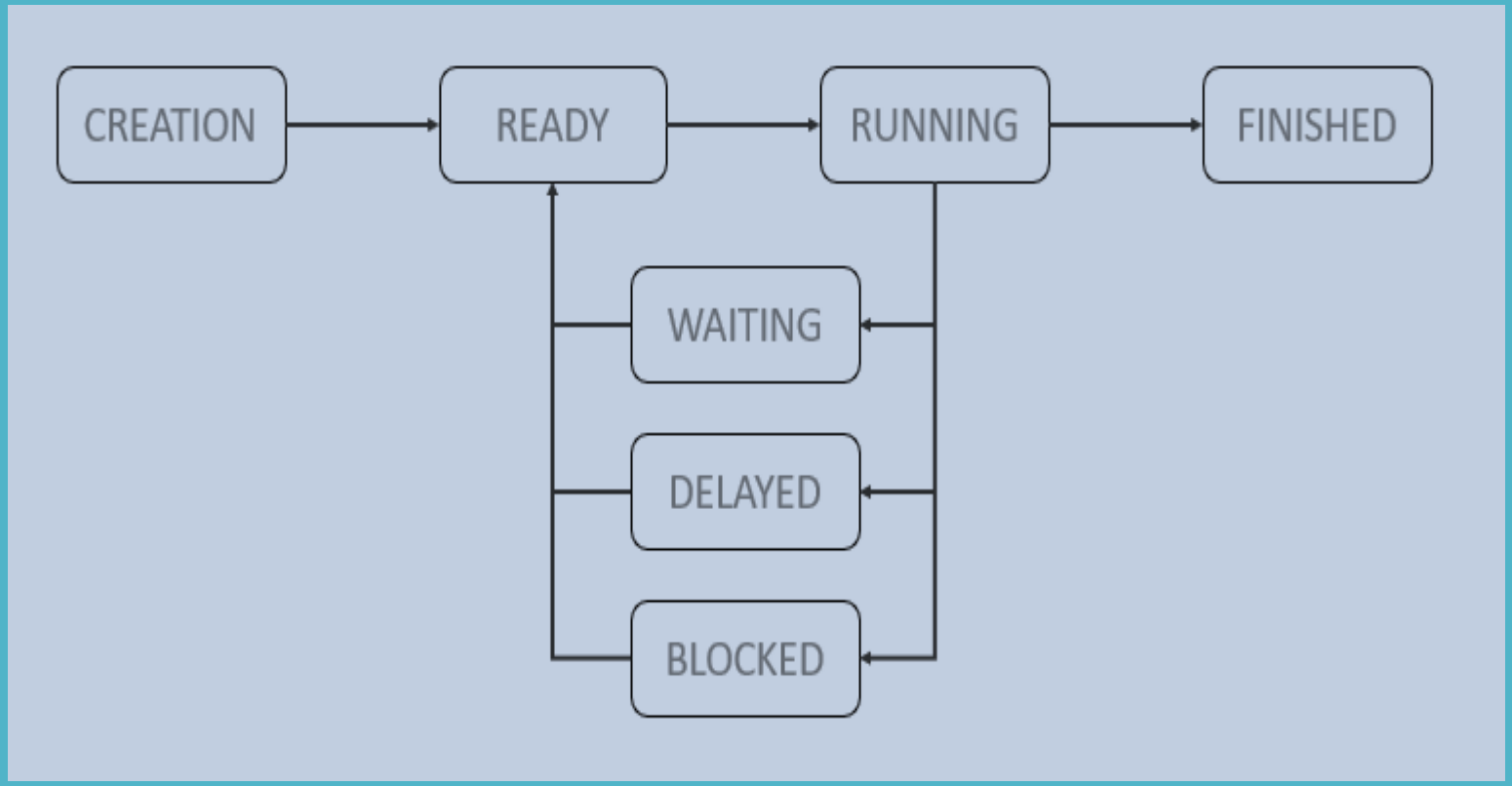
- Τα Threads ξεκινούν από το ίδιο πρόγραμμα, έχουν τα ίδια χαρακτηριστικά που περιγράφουν τις διεργασίες (αρχεία, κάτοχος, ομάδα κατόχου, τρέχων κατάλογος εργασίας, κ.τ.λ.)
- Αποτελούν τη στοιχειώδη μονάδα χρονοπρογραμματισμού στο Linux.
- Σε αυτή την προσέγγιση η διεργασία ορίζεται ως ένα σύνολο από threads που μοιράζονται τους ίδιους πόρους του συστήματος.
- Η μεταγωγή περιβάλλοντος (context switching) είναι γίνεται ανάμεσα σε threads και όχι ανάμεσα σε διεργασίες με αποτέλεσμα να γίνεται πολύ πιο γρήγορα.

Η βιβλιοθήκη pthreads (POSIX threads) <pthread.h>

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
void pthread_exit (void *retval)
int pthread_join (pthread_t thread, void **retval)
int pthread_detach (pthread_t thread)
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
int pthread_mutex_lock (pthread_mutex_t *mutex)
int pthread_mutex_unlock (pthread_mutex_t *mutex)
int pthread_mutex_destroy (pthread_mutex_t *mutex)

Διαχείριση διεργασιών

Κύκλος ζωής thread σε λειτουργικό σύστημα (process state diagram)



Διαχείριση διεργασιών

Στο λειτουργικό σύστημα Linux το Process Control Block υλοποιείται μέσω της δομής task_struct

<linux/sched.h> /home/amarg/focal/kernel/sched/sched.h

task_struct Struct Reference

`#include <sched.h>`

Data Fields

volatile long	state
void *	stack
atomic_t	usage
unsigned int	flags
unsigned int	ptrace
int	on_rq
int	prio
int	static_prio
int	normal_prio
unsigned int	rt_priority
struct sched_class *	sched_class
struct sched_entity	se
struct sched_rt_entity	rt
unsigned char	fpu_counter
unsigned int	policy
int	nr_cpus_all
cpumask_t	cpus_allowe
struct list_head	tasks
struct mm_struct *	mm
struct mm_struct *	active_mm
int	exit_state
int	exit_code
int	exit_signal
int	pdeath_signal
unsigned int	jobctl
unsigned int	personality
unsigned	did_exec:1
unsigned	in_execve:1
unsigned	in_iowait:1
unsigned	no_new_privs:1
unsigned	sched_reset_on_fork:1
unsigned	sched_contributes_to_load:1
pid_t	pid
pid_t	tgid
struct task_struct __rcu *	real_parent
struct task_struct __rcu *	parent
struct list_head	children
struct list_head	sibling
struct task_struct *	group_leader
struct list_head	ptraced
struct list_head	ptrace_entry
struct pid_link	pids [PIDTYPE_MAX]
struct list_head	thread_group
struct completion *	vfork_done
int __user *	set_child_tid
int __user *	clear_child_tid
cputime_t	utime
cputime_t	stime
cputime_t	utimescaled
cputime_t	stimescaled
cputime_t	gtime
cputime_t	prev_utime
cputime_t	prev_stime
unsigned long	nvcsw
unsigned long	nivcsw
struct timespec	start_time
struct timespec	real_start_time
unsigned long	min_flt
unsigned long	maj_flt
struct task_cputime	cputime_expires
struct list_head	cpu_timers [3]
struct cred __rcu *	real_cred
struct cred __rcu *	cred
char	comm [TASK_COMM_LEN]
int	link_count
int	total_link_count
struct thread_struct	thread
struct fs_struct *	fs
struct files_struct *	files
struct nsproxy *	nsproxy
struct signal_struct *	signal
struct sighand_struct *	sighand
sigset_t	blocked
sigset_t	real_blocked
sigset_t	saved_sigmask
struct sigpending	pending
unsigned long	sas_ss_sp
size_t	sas_ss_size
int(*	notifier)(void *pri
void *	notifier_data
sigset_t *	notifier_mask
struct callback_head *	task_works
struct audit_context *	audit_context
struct seccomp	seccomp
u32	parent_exec_id
u32	self_exec_id
spinlock_t	alloc_lock
raw_spinlock_t	pi_lock
void *	journal_info
struct bio_list *	bio_list
struct reclaim_state *	reclaim_state
struct backing_dev_info *	backing_dev_info
struct io_context *	io_context
unsigned long	ptrace_message
unsigned long	ptrace_message
siginfo_t *	last_siginfo
struct task_io_accounting	ioac
struct rcu_head	rcu
struct pipe_inode_info *	splice_pipe
struct page_frag	task_frag
int	nr_dirtied
int	nr_dirtied_pause
unsigned long	dirty_paused_when
unsigned long	timer_slack_ns
unsigned long	default_timer_slack_ns

Διαχείριση διεργασιών

Η δομή task_struct

```
volatile long state
```

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_ZOMBIE           4
#define TASK_STOPPED          8
```

(οι κυριότερες καταστάσεις)

```
int prio
```
dynamic priority considered by the scheduling mechanism

```
int static_prio
```
static priority (assigned during process startup - changed with nice)

```
int normal_prio
```
dynamic priority computed by static priority and scheduling policy

```
unsigned int rt_priority
```
priority for real time processes

```
pid_t pid
```
process ID

```
pid_t tgid
```
thread group ID

```
struct task_struct __rcu * real_parent
```
real parent process

```
struct task_struct __rcu * parent
```
recipient of SIGCHLD

```
struct list_head children
```
list of children processes

```
unsigned int policy
```
scheduling policy

```
struct task_struct *next_task, *prev_task;
struct task_struct *next_run,  *prev_run;
```

```
int exit_state
int exit_code
int exit_signal
```

```
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
```

now in cred struct

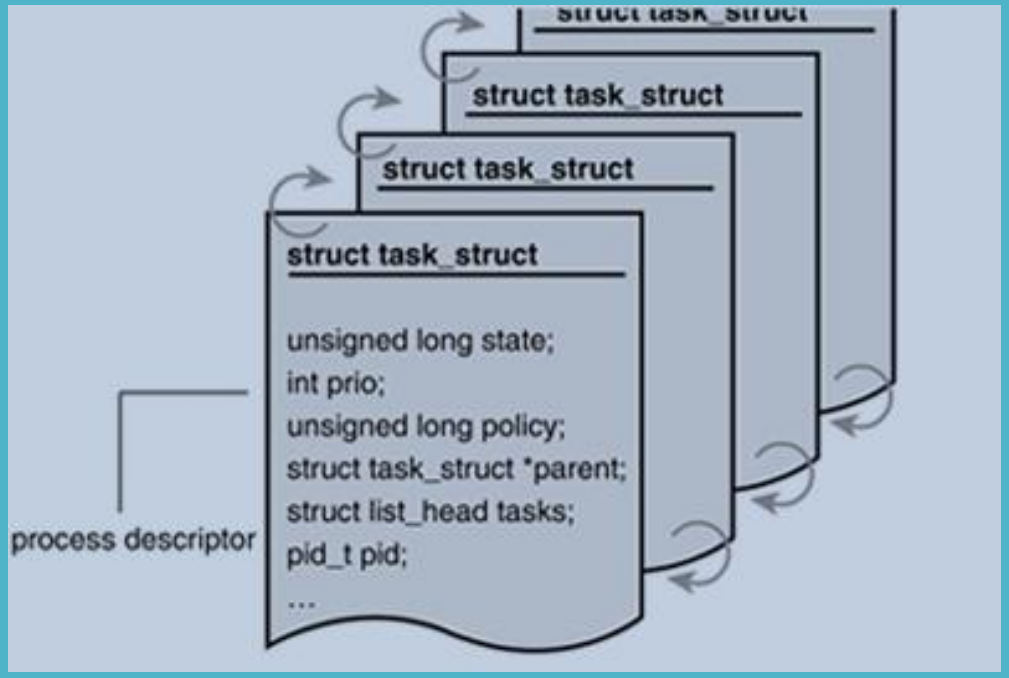
```
struct cred {
        kuid_t          uid;
        kgid_t          gid;
        kuid_t          suid;
        kgid_t          sgid;
        kuid_t          euid;
        kgid_t          egid;
        kuid_t          fsuid;
        kgid_t          fsgid;
```

Διαχείριση διεργασιών

Στο λειτουργικό σύστημα Linux ο ορισμός της δομής task_struct βρίσκεται στο αρχείο

<linux/sched.h> /home/amarg/focal/kernel/sched/sched.h

Ο πίνακας διεργασιών στο Linux αποτελείται από μία δομή task_struct για κάθε διεργασία που εκτελείται στο σύστημα, με αυτές τις δομές να αποθηκεύονται στους κόμβους μιας κυκλικής διπλής συνδεδεμένης λίστας.



Διαχείριση διεργασιών

Εκτύπωση πίνακα διεργασιών στο Linux → η εντολή ps

```
amarg@amarg-vbox:~$ ps -elF
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN    RSS PSR STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 25481 -      11336   0 09:42 ?        00:00:02 /sbin/init splash
1 S root           2       0  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kthreadd]
1 I root           3       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [rcu_gp]
1 I root           4       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [rcu_par_gp]
1 I root           6       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kworker/0:0H-kblockd]
1 I root           8       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [mm_percpu_wq]
1 S root           9       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [ksoftirqd/0]
1 I root          10       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [rcu_sched]
1 S root          11       2  0 -40   - -     0 -          0   0 09:42 ?        00:00:00 [migration/0]
5 S root          12       2  0   9   - -     0 -          0   0 09:42 ?        00:00:00 [idle_inject/0]
1 S root          14       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [cpuhp/0]
5 S root          15       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kdevtmpfs]
1 I root          16       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [netns]
1 S root          17       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [rcu_tasks_kthre]
1 S root          18       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kauditd]
1 S root          19       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [khungtaskd]
1 S root          20       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [oom_reaper]
1 I root          21       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [writeback]
1 S root          22       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kcompactd0]
1 S root          23       2  0  85   5 -     0 -          0   0 09:42 ?        00:00:00 [ksmd]
1 S root          24       2  0  99  19 -     0 -          0   0 09:42 ?        00:00:00 [khugepaged]
1 I root          70       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kintegrityd]
1 I root          71       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kblockd]
1 I root          72       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [blkcg_punt_bio]
1 I root          73       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [tpm_dev_wq]
1 I root          74       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [ata_sff]
1 I root          75       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [md]
1 I root          76       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [edac-poller]
1 I root          77       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [devfreq_wq]
1 S root          78       2  0 -40   - -     0 -          0   0 09:42 ?        00:00:00 [watchdogd]
1 S root          81       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kswapd0]
1 S root          82       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [ecryptfs-kthrea]
1 I root          84       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kthrotld]
```

Διαχείριση διεργασιών

Χαρακτηριστικά διεργασιών (pid & ppid)

Process Id (pid) → Θετικός ακέραιος που προσδιορίζει την κάθε διεργασία με μοναδικό τρόπο μέσα στο σύστημα (τύπου pid_t → int).

Parent Process Id (ppid) → Θετικός ακέραιος που προσδιορίζει την γονική της κάθε διεργασίας με μοναδικό τρόπο μέσα στο σύστημα (τύπου pid_t →int).

Ο τύπος pid_t είναι int που ορίζεται στο αρχείο types.h ως **typedef int pid_t;**

Για την ανάκτηση του pid και του ppid για την κάθε διεργασία χρησιμοποιούνται οι συναρτήσεις

```
pid_t getpid ();
pid_t getppid ();
```

που είναι δηλωμένες στο αρχείο επικεφαλίδας unistd.h.

Διαχείριση διεργασιών

Χαρακτηριστικά διεργασιών (uid & gid)

User Id (uid) (θετικός ακέραιος) → Ο κωδικός του χρήστη που δημιουργεί τη διεργασία. Εάν είναι uid = 0 τότε ο εν λόγω χρήστης είναι ο διαχειριστής του συστήματος (root) ο οποίος δεν υπόκειται σε έλεγχο ασφάλειας από τον πυρήνα του λειτουργικού. Οι κωδικοί των χρηστών είναι αποθηκευμένοι στο αρχείο συστήματος /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
```

Group Id (gid) (θετικός ακέραιος) → Ο κωδικός της πρωτεύουας ομάδας του παραπάνω χρήστη. Οι κωδικοί των ομάδων χρηστών είναι αποθηκευμένοι στο αρχείο /etc/groups.

```
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,amarg
tty:x:5:syslog
disk:x:6:
lp:x:7:
mail:x:8:
news:x:9:
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
dialout:x:20:
fax:x:21:
voice:x:22:
cdrom:x:24:amarg
floppy:x:25:
tape:x:26:
sudo:x:27:amarg
audio:x:29:pulse
dip:x:30:amarg
www-data:x:33:
```

Διαχείριση διεργασιών

Χαρακτηριστικά διεργασιών (uid & gid)

Η χρήση ενός και μοναδικού gid για τις διεργασίες με το ίδιο uid δημιουργούσε προβλήματα, εάν ο ίδιος χρήστης έπρεπε να προσπελάσει αρχεία με διαφορετικό gid.

Για το λόγο αυτό σε μεταγενέστερες υλοποιήσεις μία διεργασία μπορούσε να συσχετιστεί το πολύ με NGROUP διαφορετικά gid όπου η σταθερά NGROUP = 32 δηλώνεται στο <param.h>.

Ορισμός συνόλου gids για την κάθε διεργασία

int setgroups (size_t num, const gid_t * groupList)

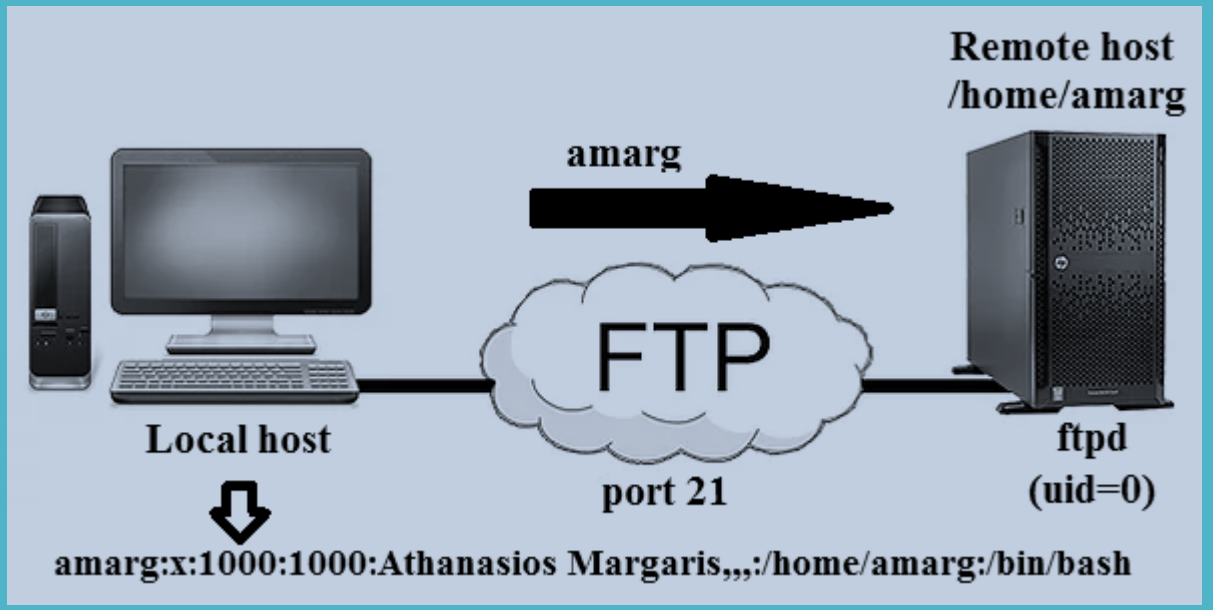
Ανάκτηση συνόλου gids για την κάθε διεργασία

int getgroups (size_t num, gid_t * groupList)

Διαχείριση διεργασιών

Αλλαγή του uid και οι εφαρμογές setuid

Σε ορισμένες περιπτώσεις η επιτυχής εκτέλεση μιας διαδικασίας απαιτεί τη χρήση της τιμής uid ενός διαφορετικού χρήστη – παράδειγμα η εφαρμογή ftpd.



Το uid του ftpd τίθεται στην τιμή 1000 (συνήθως ξεκινά νέο αντίγραφο της διεργασίας)
Ωστόσο εάν απαιτηθεί ξανά να είναι uid = 0 (π.χ για να ανοίξει μία σύνδεση) αυτό είναι αδύνατο αφού μία διεργασία δεν μπορεί να χορηγήσει στον εαυτό της δικαιώματα διαχειριστή.

Διαχείριση διεργασιών

Αλλαγή του uid και οι εφαρμογές setuid

ΛΥΣΗ → Χρησιμοποιούνται τρία διαφορετικά uid !

- Real uid (ruid) → πραγματικό uid
- Saved uid (suid) → αποθηκευμένο uid
- Effective uid (euid) → ενεργό uid

Για την περίπτωση του ftpd
Αρχικά ruid = suid = euid = 0 (root access)
Όταν συνδεθεί ο χρήστης με uid = ID τότε
euid = ID ενώ ruid = suid = 0
Όταν απαιτείται root access γίνεται euid = 0
και μετά την ολοκλήρωση ξανά euid = ID

```
cred {
kuid_t          uid;
kgid_t          gid;
kuid_t          suid;
kgid_t          sgid;
kuid_t          euid;
kgid_t          egid;
kuid_t          fsuid;
kgid_t          fsgid;
```

Στη γενική περίπτωση κατά το login του χρήστη θέτουμε ruid = suid = euid = ID < /etc/passwd
Όταν απαιτείται αλλαγή του uid τα ruid και euid καθορίζουν εάν αυτό που ζητά ο χρήστης είναι επιτρεπτό και εάν ναι, θέτουμε euid = ruid ή euid = suid. Εάν είναι euid = 0 (root access) το euid μπορεί να λάβει οποιαδήποτε τιμή.

Διαχείριση διεργασιών

Αλλαγή του uid και οι εφαρμογές setuid

Τα στοιχεία των χρηστών είναι αποθηκευμένα στο αρχείο συστήματος /etc/passwd

-rw-r--r-- 1 root root 2797 Σεπ 19 20:17 /etc/passwd

Παρατηρούμε πως ο κάτοχος του αρχείου είναι ο διαχειριστής (root) ο οποίος έχει δικαίωμα ανάγνωσης και εγγραφής (r w -) ενώ οι άλλοι χρήστες έχουν δικαίωμα μόνο ανάγνωσης (r - -).

Ωστόσο, ο κάθε χρήστης χρησιμοποιώντας την εντολή passwd μπορεί να αλλαξει τον κωδικό πρόσβασης στο σύστημα, με το νέο κωδικό να αποθηκεύεται στο αρχείο /etc/passwd αντικαθιστώντας τον παλαιό!

Πως είναι δυνατή η τροποποίηση του περιεχομένου ενός αρχείου, στο οποίο ο χρήστης δεν έχει δικαίωμα εγγραφής?

Διαχείριση διεργασιών

Αλλαγή του uid και οι εφαρμογές setuid

Η απάντηση βρίσκεται στη μάσκα δικαιωμάτων της εφαρμογής /usr/bin/passwd

```
amarg@amarg-vbox:~$ which passwd
/usr/bin/passwd
amarg@amarg-vbox:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 Μαϊ  28 09:37 /usr/bin/passwd
```

Το bit s (setuid bit) επιτρέπει την εκτέλεση του προγράμματος passwd από οποιονδήποτε χρήστη με τα δικαιώματα που έχει ο κάτοχος του προγράμματος. Αυτός ο κάτοχος είναι ο root με δικαιώματα διαχειριστή και για το λόγο αυτό είναι δυνατή η τροποποίηση του αρχείου /etc/passwd.

Βέβαια θα πρέπει ταυτόχρονα το ruid να ταυτίζεται με το uid της γραμμής του /etc/passwd που πρόκειται να μεταβληθεί, αλλιώς ο κάθε χρήστης θα μπορούσε να αλλάξει το password οποιουδήποτε άλλου χρήστη !!

Οι εφαρμογές αυτού του τύπου που όταν εκτελούνται έχουν το ίδιο euid (ή egid) με τον κάτοχο του προγράμματος ανεξάρτητα από το ποιος χρήστης τις εκτελεί λέγονται εφαρμογές τύπου setuid ή setgid.

Διαχείριση διεργασιών

Αλλαγή του uid και οι εφαρμογές setuid

Εάν ID ειναι το uid του χρήστη που χρησιμοποιεί μία εφαρμογή setuid αυτό αποθηκεύεται στο euid η παλαιά τιμή του οποίου αποθηκεύεται στο suid έτσι ώστε να μπορεί να ανακτηθεί αργότερα.

Άρα για την εκτέλεση αυτών των προγραμμάτων γίνονται οι εκχωρήσεις τιμών

suid = euid και euid = ID

ενώ στο τέλος το euid ανακτά την αρχική του τιμή ως euid = suid. Επίσης μπορούμε να θέσουμε ruid = ID

Οι σχετικές κλήσεις συστήματος

setuid	set real user ID
getuid	get real user ID
geteuid	get effective user ID
getegid	get effective group ID
setreuid	set real and effective user IDs
setregid	set real and effective group IDs
setfsuid	set user identity used for file system checks
setfsgid	set group identity used for file system checks
setresuid	set real, effective and saved user or group ID
getresuid	get real, effective and saved user or group ID

```
int   setuid   (uid_t);
int   setgid   (gid_t);
int   seteuid  (uid_t);
int   setegid  (gid_t);
int   setreuid (uid_t, uid_t);
int   setregid (gid_t, gid_t);
uid_t getuid   (void);
gid_t getgid   (void);
uid_t geteuid  (void);
gid_t getegid  (void);
```

Διαχείριση διεργασιών

setuid, setgid & sticky bits

setuid bit → εάν αυτό το bit είναι on η εφαρμογή δεν εκτελείται με τα δικαιώματα του χρήστη που την εκτελεί, αλλά με τα δικαιώματα του κατόχου της. Δεν επηρεάζει τα δικαιώματα των καταλόγων και απαιτεί την ενεργοποίηση του bit x στην τριάδα bits του κατόχου.

setgid bit → εάν αυτό το bit είναι on η εφαρμογή δεν εκτελείται με τα δικαιώματα της ομάδας του χρήστη που την εκτελεί, αλλά με τα δικαιώματα της ομάδας του κατόχου της. Εάν χρησιμοποιηθεί με καταλόγους, όλα τα νέα αρχεία που δημιουργούνται μέσα σε αυτούς λαμβάνουν τα δικαιώματα του γονικού καταλόγου. Απαιτεί την ενεργοποίηση του bit x στην τριάδα bits της ομάδας του κατόχου.

sticky bit → επιδρά μόνο σε καταλόγους και εάν ενεργοποιηθεί, τότε όλα τα αρχεία που υπάρχουν μέσα σε αυτούς μπορούν να τροποποιηθούν μόνο από τους κατόχους τους (π.χ. ο κατάλογος /tmp).

Διαχείριση διεργασιών

setuid, setgid & sticky bits

Με ποιο τρόπο γίνεται η ενεργοποίηση των τριών ειδικών bits

Ενεργοποίηση setuid bit → 4
Ενεργοποίηση setguid bit → 2
Ενεργοποίηση sticky bit → 1

Αυτός ο αριθμός γράφεται πριν από την οκταδική μάσκα δικαιωμάτων της chmod η οποία πλέον έχει τέσσερα ψηφία

Διαχείριση διεργασιών

setuid, setgid & sticky bits

Το γράμμα που εκτυπώνεται (s ή t) είναι κεφαλαίο ή μικρό ανάλογα με το εάν το bit εκτέλεσης x έχει ενεργοποιηθεί ή όχι, αντίστοιχα

Παραδείγματα ορισμού setuid bit

chmod **4**544 file1 (r – x r - - r - -) → – r – s r - - r - - 1 amarg amarg 0 Σεπ 24 23:02 file1
chmod **4**644 file1 (r w – r - - r - -) → - r w S r - - r - - 1 amarg amarg 0 Σεπ 24 23:02 file1

Παραδείγματα ορισμού setgid bit

chmod **2**656 file1 (r w – r – x r w -) → - r w – r – s r w - 1 amarg amarg 0 Σεπ 24 23:02 file1
chmod **2**665 file1 (r w – r w – r - -) → - r w – r w S r - x 1 amarg amarg 0 Σεπ 24 23:02 file1

Παραδείγματα ορισμού sticky bit

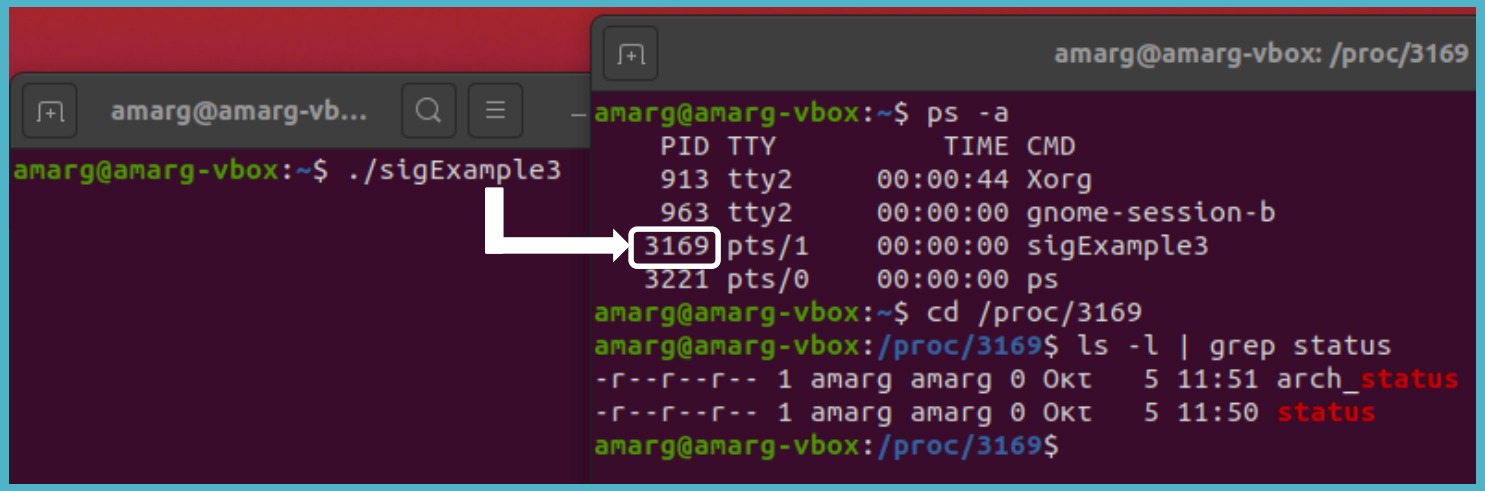
chmod **1**755 file1 (r w x r – x r – x) → - r w x r – x r - t 1 amarg amarg 0 Σεπ 24 23:02 file1
chmod **1**756 file1 (r w x r – x r w -) → -r w x r – x r w T 1 amarg amarg 0 Σεπ 24 23:02 file1

Απενεργοποίηση των setuid, setgid και sticky bits chmod **0**544 file1 **ή πιο απλά** chmod 544 file1

Διαχείριση διεργασιών

Ανάκτηση των πληροφοριών που σχετίζονται με την κατάσταση της διεργασίας

Για κάθε διεργασία με pid xyzt, το αρχείο /proc/xyzt/status περιέχει πληροφορίες για την κατάσταση εκτέλεσης της διεργασίας.



Αυτό το αρχείο περιέχει όλες τις πληροφορίες που εκτυπώνονται από την εντολή ps (στην πραγματικότητα η εντολή ps ανατρέχει στο proc file system για να ανακτήσει τις πληροφορίες που εκτυπώνει) αλλά και άλλες πληροφορίες που δεν εκτυπώνει η ps

Διαχείριση διεργασιών

Παράδειγμα περιεχομένων status file

```
Name:   rsyslogd
Umask:  0022
State:  S (sleeping)
Tgid:   373
Ngid:   0
Pid:    373
PPid:   1
TracerPid:      0
Uid:    104     104     104     104
Gid:    110     110     110     110
FDSize:         64
Groups:         4 5 110
NStgid:         373
NSpid:  373
NSpgid:         373
NSsid:  373
VmPeak:           224332 kB
VmSize:           224332 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      4792 kB
VmRSS:      4792 kB
RssAnon:            1040 kB
RssFile:            3752 kB
RssShmem:              0 kB
VmData:            18304 kB
VmStk:       132 kB
VmExe:       468 kB
VmLib:      3568 kB
VmPTE:        80 kB
VmSwap:                0 kB
HugetlbPages:          0 kB
CoreDumping:    0
THP_enabled:    1
Threads:        4
```

```
SigQ:   0/7730
SigPnd:         0000000000000000
ShdPnd:         0000000000000000
SigBlk:         0000000000000000
SigIgn:         0000000001001206
SigCgt:         0000000180314001
CapInh:         0000000000000000
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         0000003fffffffff
CapAmb:         0000000000000000
NoNewPrivs:     0
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   1
Cpus_allowed_list:      0
Mems_allowed:
00000000,00000000,00000000,00000000,00000000
Mems_allowed_list:      0
voluntary_ctxt_switches:        51
nonvoluntary_ctxt_switches:     13
```

Διαχείριση διεργασιών

Οι πληροφορίες που αποθηκεύονται στο status file

Field	Content
Name	filename of the executable
Umask	file mode creation mask
State	state (R is running, S is sleeping, D is sleeping in an uninterruptible wait, Z is zombie, T is traced or stopped)
Tgid	thread group ID
Ngid	NUMA group ID (0 if none)
Pid	process id
PPid	process id of the parent process
TracerPid	PID of process tracing this process (0 if not)
Uid	Real, effective, saved set, and file system UIDs
Gid	Real, effective, saved set, and file system GIDs
FDSize	number of file descriptor slots currently allocated
Groups	supplementary group list
NStgid	descendant namespace thread group ID hierarchy
NSpid	descendant namespace process ID hierarchy
NSpgid	descendant namespace process group ID hierarchy
NSsid	descendant namespace session ID hierarchy
VmPeak	peak virtual memory size
VmSize	total program size
VmLck	locked memory size
VmPin	pinned memory size
VmHWM	peak resident set size (“high water mark”)

Διαχείριση διεργασιών

Οι πληροφορίες που αποθηκεύονται στο status file

VmRSS	size of memory portions. It contains the three following parts (VmRSS = RssAnon + RssFile + RssShmem)
RssAnon	size of resident anonymous memory
RssFile	size of resident file mappings
RssShmem	size of resident shmem memory (includes SysV shm, mapping of tmpfs and shared anonymous mappings)
VmData	size of private data segments
VmStk	size of stack segments
VmExe	size of text segment
VmLib	size of shared library code
VmPTE	size of page table entries
VmSwap	amount of swap used by anonymous private data (shmem swap usage is not included)
HugetlbPages	size of hugetlb memory portions
CoreDumping	process's memory is currently being dumped (killing the process may lead to a corrupted core)
THP_enabled	process is allowed to use THP (returns 0 when PR_SET_THP_DISABLE is set on the process
Threads	number of threads
SigQ	number of signals queued/max. number for queue
SigPnd	bitmap of pending signals for the thread ←
ShdPnd	bitmap of shared pending signals for the process ←
SigBlk	bitmap of blocked signals ←
SigIgn	bitmap of ignored signals ←
SigCgt	bitmap of caught signals ←
CapInh	bitmap of inheritable capabilities

Διαχείριση διεργασιών

Οι πληροφορίες που αποθηκεύονται στο status file

CapPrm	bitmap of permitted capabilities
CapEff	bitmap of effective capabilities
CapBnd	bitmap of capabilities bounding set
CapAmb	bitmap of ambient capabilities
NoNewPrivs	no_new_privs, like prctl(PR_GET_NO_NEW_PRIV, …)
Seccomp	seccomp mode, like prctl(PR_GET_SECCOMP, …)
Speculation_Store_Bypass	speculative store bypass mitigation status
Cpus_allowed	mask of CPUs on which this process may run
Cpus_allowed_list	Same as previous, but in “list format”
Mems_allowed	mask of memory nodes allowed to this process
Mems_allowed_list	Same as previous, but in “list format”
voluntary_ctxt_switches	number of voluntary context switches
nonvoluntary_ctxt_switches	number of non voluntary context switches

Διαχείριση διεργασιών

Δημιουργία διεργασιών – η εντολή fork ()

Η δημιουργία διεργασιών στο λειτουργικό σύστημα Linux πραγματοποιείται με την εντολή

pid_t fork (void);

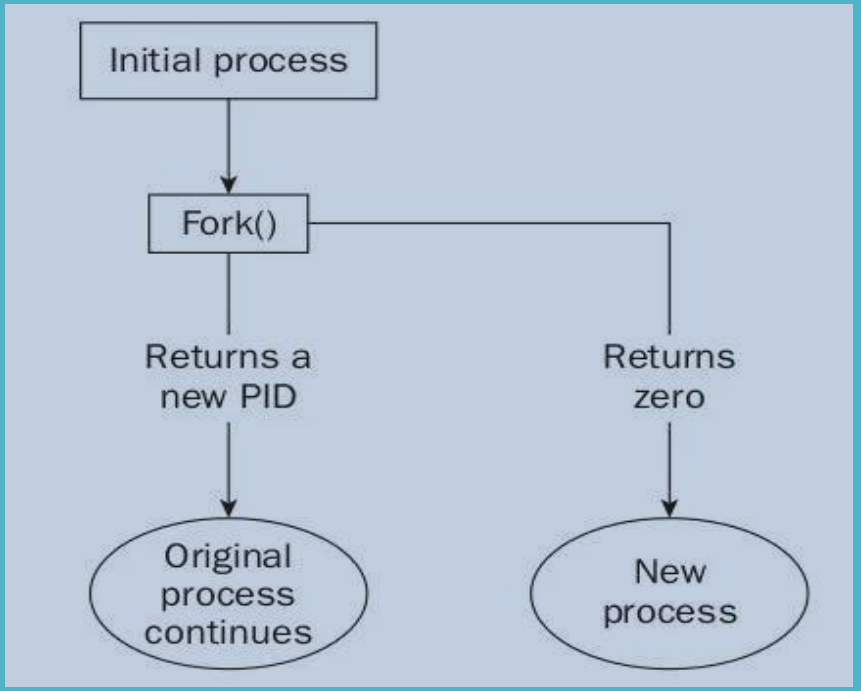
Η νέα διεργασία που δημιουργείται με τον τρόπο αυτό αποτελεί θυγατρική διεργασία της διεργασίας που κάλεσε τη fork και ως εκ τούτου οι δύο διεργασίες σχετίζονται με σχέση γονέα – παιδιού.

Η fork επιστρέφει την τιμή της τόσο στη γονική όσο και στη θυγατρική διεργασία !!

- Η τιμή που επιστρέφεται στη γονική διεργασία είναι το pid της θυγατρικής διεργασίας.
- Η τιμή που επιστρέφεται στη θυγατρική διεργασία είναι η τιμή 0.

Με τον τρόπο αυτό είναι δυνατή η διάκριση ανάμεσα στις δύο διεργασίες.

Μία επιστρεφόμενη τιμή ίση με -1 υποδηλώνει την εμφάνιση σφάλματος δημιουργίας της διεργασίας.



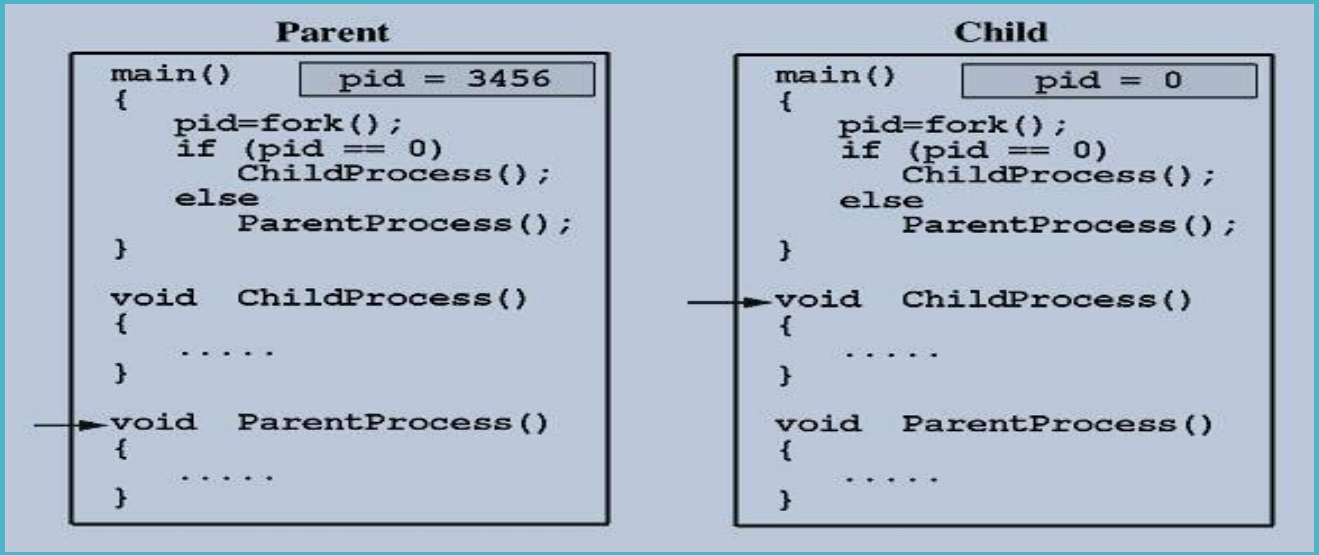
Διαχείριση διεργασιών

Δημιουργία διεργασιών – η εντολή fork ()

Η γονική και η θυγατρική διεργασία έχουν διαφορετικό χώρο διευθύνσεων μνήμης αλλά με το ίδιο περιεχόμενο αφού εκτελείται ο ίδιος κώδικας.

Μετά την κλήση της fork () οι δύο διεργασίες αρχίζουν να εκτελούνται και υφίστανται ταυτόχρονα.

Οι μεταβλητές που έχουν αρχικοποιηθεί πριν τη κλήση της fork () έχουν τις ίδιες τιμές στις δύο διεργασίες ενώ από εκεί και πέρα οι τροποποιήσεις που πραγματοποιούνται στη μεταβλητή της μίας διεργασίας δεν επηρεάζουν την αντίστοιχη μεταβλητή της άλλης διεργασίας.



Διαχείριση διεργασιών

Τερματισμός διεργασιών – η εντολή exit ()

Μία διεργασία μπορεί να τερματιστεί με δύο διαφορετικούς τρόπους και πιο συγκεκριμένα:

- Φυσιολογικά, όταν ολοκληρωθεί ή καλώντας την εντολή exit ή την εντολή _exit ()
- Απροσδόκητα, όταν δεχθεί σήμα τερματισμού (kill) ή όταν καταρρεύσει το σύστημα.

Υπάρχουν δύο συναρτήσεις για κανονικό τερματισμό, η **void _exit (int exitCode)** η οποία είναι κληση συστήματος και η **void exit (int exitCode)** η οποία είναι συνάρτηση της C. Στη δεύτερη περίπτωση, η συνάρτηση exit καλεί τις συναρτήσεις που έχουν οριστεί ως

int atexit (void (*function) (void))

και ύστερα καλεί τη συνάρτηση _exit(). Ο απροσδόκητος τερματισμός μιας διεργασίας γίνεται από την

int kill (pid_t pid, int sig)

όπου pid ο κωδικός της διεργασίας προς τερματισμό και sig κατάλληλο σήμα τερματισμού. Εάν είναι pid > 0 το σήμα στέλνεται στη διεργασία με αυτόν τον κωδικό, ενώ εάν είναι pid = -1 το σήμα στέλνεται σε όλες τις διεργασίες εκτός από την init κατά το shutdown του συστήματος.

Το λειτουργικό διαθέτει την εντολή **kill pid** όπου pid ο κωδικός της διεργασίας προς τερματισμό

Διαχείριση διεργασιών

Αναστολή διεργασιών – η εντολή wait4

Στενά συνδεδεμένη με τις συναρτήσεις fork και exit είναι η συνάρτηση

pid_t wait4 (pid_t *pid*, int * *wstatus*, int *options*, struct rusage * *rusage*);

η οποία ανακτά και επιστρέφει τον κωδικό επιστροφής της εξεταζόμενης διεργασίας, όπου:

pid → το process id της διεργασίας της οποίας αναμένεται ο κωδικός τερματισμού με τιμές

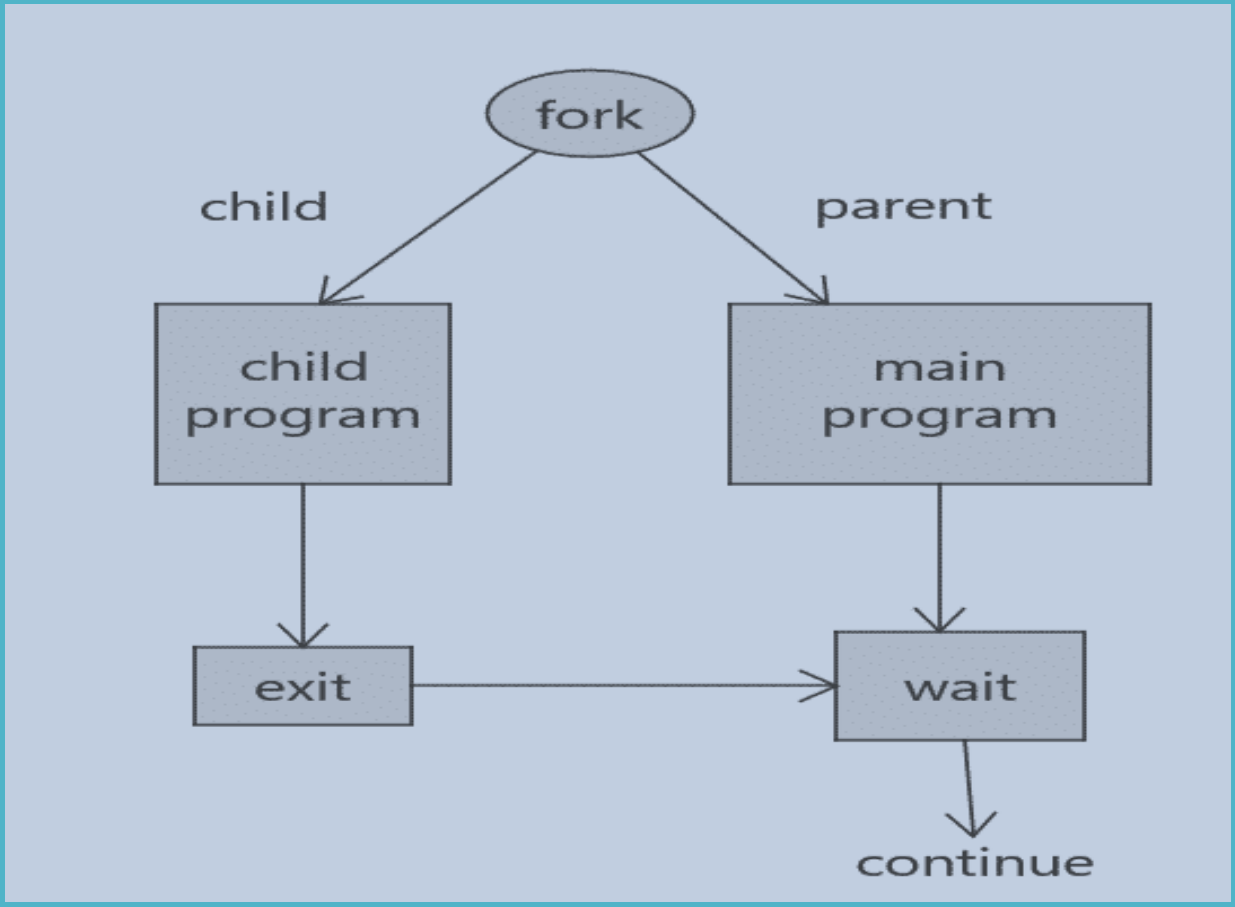
- < -1 → αναμένει τον τερματισμό οποιασδήποτε θυγατρικής διεργασίας της οποίας το pgid είναι το ίδιο με από απόλυτη τιμή της παραμέτρου pid.
- = -1 → αναμένει τον τερματισμό οποιασδήποτε θυγατρικής διεργασίας.
- = 0 → αναμένει τον τερματισμό διεργασιών που ανήκουν στην ίδια ομάδα διεργασιών με την τρέχουσα διεργασία.
- > 0 αναμένει τον τερματισμό της διεργασίας με process id ίσο με pid.

wstatus → δείκτης στην περιοχή αποθήκευσης του κωδικού τερματισμού της διεργασίας.

Κατά τη διάρκεια αυτής της διαδικασίας αναμονής του τερματισμού της θυγατρικής διεργασίας, η εκτέλεση της διεργασίας που κάλεσε τη συνάρτηση wait4, αναστέλλεται.

Διαχείριση διεργασιών

Συνδυασμένη χρήση των fork, wait & exit



Διαχείριση διεργασιών

Εκτέλεση εντολών και προγραμμάτων μέσα από διεργασίες

Οι επόμενες οκτώ συναρτήσεις επιτρέπουν την εκτέλεση ενός προγράμματος μέσα από ένα άλλο και χαρακτηρίζονται από την ολική αντικατάσταση του προγράμματος που καλεί αυτές τις συναρτήσεις από το πρόγραμμα που καλείται.

Εάν επιθυμούμε τη διατήρηση του αρχικού προγράμματος θα πρέπει να δημιουργήσουμε μία νέα διεργασία καλώντας τη fork και στη συνέχεια να καλέσουμε την κατάλληλη από τις παραπάνω συναρτήσεις μέσα από τη νέα διεργασία.

```
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);
int execlpe(const char *file, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Αυτές οι συναρτήσεις διαφέρουν απλά στον τρόπο με τον οποίο οι παράμετροι της εντολής κλήσης τους περνούν στο νέο πρόγραμμα.

Διαχείριση διεργασιών

Εκτέλεση εντολών και προγραμμάτων μέσα από διεργασίες

Οι παραπάνω συναρτήσεις διαχωρίζονται σε δύο κατηγορίες με κριτήριο το εάν το πέμπτο γράμμα του ονόματός τους είναι το **l** ή το **v**

execl execle execlp execlpe execv execve execvp execvpe

- Οι συναρτήσεις με το γράμμα l (list) δέχονται μία λίστα ορισμάτων (καταχωρημένα το ένα μετά το άλλο και χωρισμένα με κόμμα) η οποία τερματίζεται με NULL.
- **execl ("/bin/bash", "ls", "-l", "-R", "-a", NULL)**

- Οι συναρτήσεις με το γράμμα v (vector) δέχονται ως όρισμα έναν δείκτη προς ένα διάνυσμα δεικτών που περιέχει τα ορίσματα της συνάρτησης.

```
char * array [] = {"ls", "-l", "-R", "-a", NULL}
execl ("/bin/bash", array)
```

- Οι συναρτήσεις με το γράμμα p (path) αναζητούν το πρόγραμμα προς εκτέλεση στη λίστα των καταλόγων που βρίσκονται καταχωρημένοι στη μεταβλητή PATH.

- Οι συναρτήσεις με το γράμμα e (environment) δέχονται ένα δείκτη προς ένα νέο περιβάλλον προς το πρόγραμμα που φορτώνεται για εκτέλεση.

Διαχείριση διεργασιών

Οι συναρτήσεις system και popen

Εάν μία διεργασία καλέσει τη συνάρτηση system που ορίζεται ως

int system (const char * cmd)

περιμένει την ολοκλήρωση του προγράμματος cmd πριν συνεχίσει με την εκτέλεση του δικού της κώδικα. Η system επομένως δημιουργεί μία νέα θυγατρική διεργασία και εκτελεί το πρόγραμμα του φλοιού καλώντας την exec, το οποίο με τη σειρά του εκτελεί την εντολή cmd. Ο κωδικός επιστροφής της θυγατρικής διεργασίας διαβιβάζεται στη γονική διεργασία με τη wait.

FILE * popen (const char * cmd, const char * mode)

Η popen εκτελεί την εντολή cmd και δημιουργεί μία διασωλήνωση ανάμεσα στην εφαρμογή που κάλεσε την popen και στην εφαρμογή cmd επιστρέφοντας ένα δείκτη προς το ρεύμα δεδομένων που άνοιξε για διαδικασία ανάγνωσης ή εγγραφής.

Η cmd εκτελείται από το κέλυφος όπως και στη system ενώ το mode είναι “r” εάν η γονική διεργασία θέλει να διαβάσει την έξοδο της εντολής cmd και “w” εάν η εντολή cmd δέχεται είσοδο από τη διεργασία. Στο τέλος καλείται η

int pclose (FILE * stream)

που επιστρέφει τον κωδικό τερματισμού της θυγατρικής διεργασίας μέσω της wait.

Διαχείριση διεργασιών

Διαχείριση περιβάλλοντος

Η κάθε διεργασία που ξεκινά σε ένα σύστημα Linux με κάποιον από τους παραπάνω τρόπους έχει πρόσβαση στο σύνολο των μεταβλητών περιβάλλοντος του συστήματος.

Η ανάκτηση της τιμής μιας μεταβλητής περιβάλλοντος γίνεται με τη συνάρτηση

char * getenv (const char * variable)

Η προσθήκη περιεχομένου στο περιβάλλον γίνεται με τη συνάρτηση

int putenv (char * string)

όπου string συμβολοσειρά της μορφής variable = value ενώ η τροποποίηση γίνεται με τη συνάρτηση

int setenv (const char * variable, const char * value, int overwrite)

Τέλος η διαγραφή μιας μεταβλητής περιβάλλοντος γίνεται με τη συνάρτηση

int unsetenv (const char * variable)

Οι παραπάνω συναρτήσεις είναι δηλωμένες στο αρχείο επικεφαλίδας **stdlib.h**

Διαχείριση διεργασιών

Διαχείριση Πόρων

Για κάθε συνηθισμένη διεργασία που εκτελείται σε ένα σύστημα Linux υπάρχει άνω όριο σχετικά με τους πόρους (επεξεργαστή, μνήμη, δίσκο) που της διατίθενται για την εκτέλεσή της. Για κάθε πόρο ορίζονται δύο τιμές ορίων, μία για το μέγιστο όριο (hard) και μία για το ελάχιστο όριο (soft).

Η ανάκτηση των πόρων που έχουν ανατεθεί σε μία διεργασία γίνεται από τη συνάρτηση

int getrlimit (int resource, struct rlimit * rlp)

ενώ ο καθορισμός των πόρων γίνεται με τη βοήθεια της συνάρτησης

int setrlimit (int resource, const struct rlimit * rlp)

όπου rlimit κατάλληλα ορισμένη δομή δεδομένων που ορίζεται ως (<sys/resource.h>)

struct rlimit { rlim_t rlim_cur; rlim_t rlim_max };

Οι καθιερωμένοι πόροι είναι οι εξής:

- RLIMIT_CORE → μέγιστο μέγεθος αρχείου
- RLIMIT_CPU → μέγιστος χρόνος χρήσης της CPU (seconds)
- RLIMIT_DATA → μέγιστο μέγεθος τμήματος δεδομένων (bytes)
- RLIMIT_FSIZE → μέγιστο μέγεθος αρχείων (bytes)
- RLIMIT_NOFILE → μέγιστος αριθμός ταυτόχρονα ανοικτών αρχείων
- RLIMIT_STACK → μέγιστο μέγεθος στοίβας (bytes)

Διαχείριση διεργασιών

Η διεργασία init

Αποτελεί daemon process που ξεκινά κατά την εκκίνηση του υπολογιστή και εκτελείται συνεχώς μέχρι τον τερματισμό του που γίνεται με την εντολή

shutdown –h [-r] now

Είναι η γονική διεργασία όλων των διεργασιών που τρέχουν στο σύστημα και αναλαμβάνει τη διαχείριση των ορφανών διεργασιών. Η αποτυχημένη της εκκίνηση οδηγεί σε kernel panic.

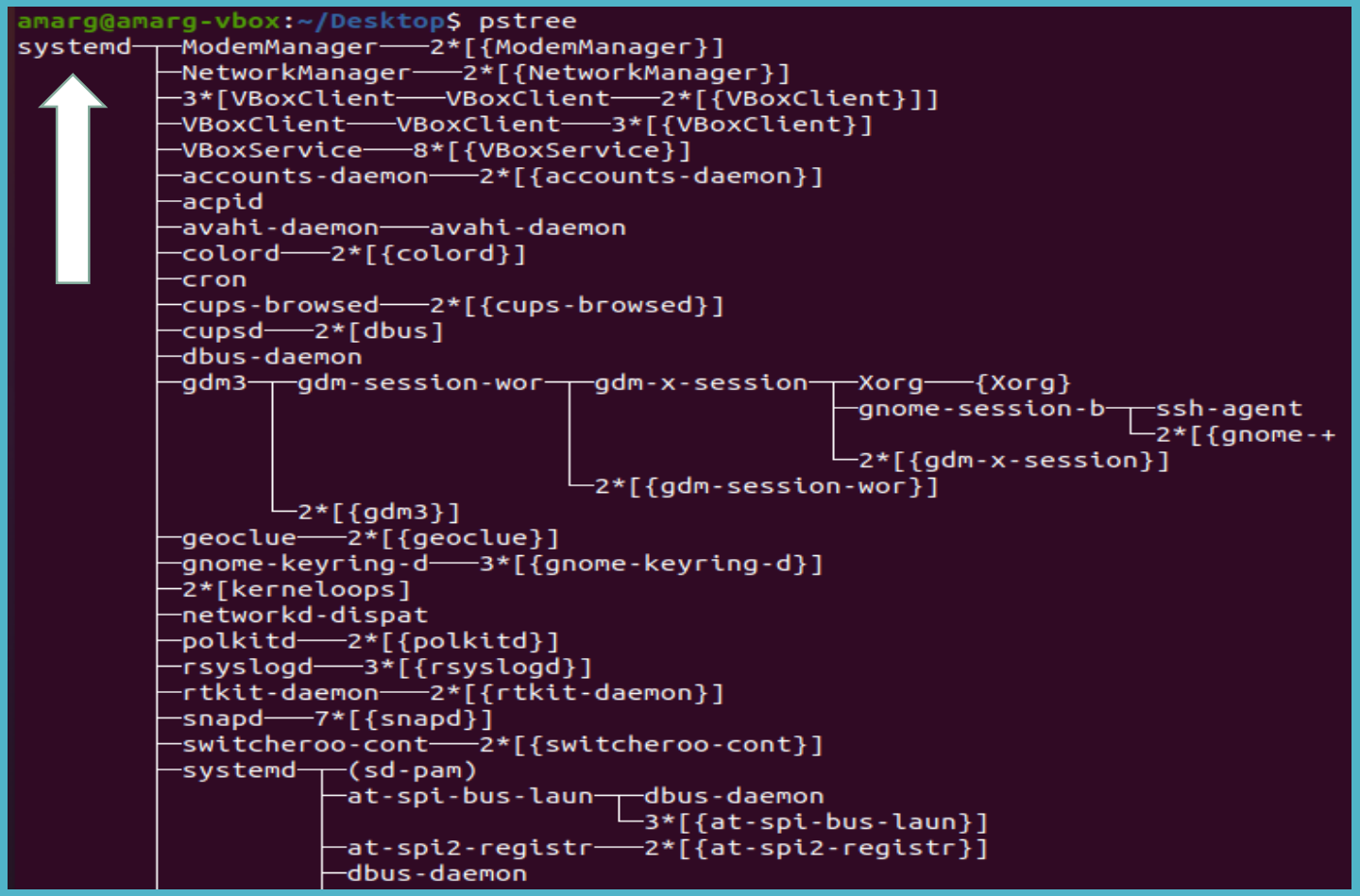
Η συμπεριφορά της καθορίζεται από τα περιεχόμενα του αρχείου /etc/inittab και μία από τις πιο σημαντικές παραμέτρους που ορίζονται εδώ είναι ο καθορισμός του runlevel εκκίνησης του συστήματος → **id:3:initdefault**

runlevel 0	προκαλεί τον τερματισμό της λειτουργίας του συστήματος
runlevel 1	εκκινεί το λειτουργικό σύστημα σε κατάσταση απλού χρήστη **(single user mode)**
runlevel 2	εκκινεί το λειτουργικό σύστημα σε κατάσταση πολλών χρηστών **(multi user mode)** αλλά χωρίς υποστήριξη δικτύου
runlevel 3	εκκινεί το λειτουργικό σύστημα σε κατάσταση πολλών χρηστών **(multi user mode)** με ενεργοποιημένη την υποστήριξη δικτύου
runlevel 4	είναι δεσμευμένο αλλά δεν χρησιμοποιείται
runlevel 5	εκκινεί το σύστημα με ταυτόχρονη ενεργοποίηση του γραφικού περιβάλλοντος των **X Windows**
runlevel 6	προκαλεί την επανεκκίνηση του συστήματος.

Διαχείριση διεργασιών

Η διεργασία systemd

Στις τελευταίες εκδόσεις του λειτουργικού συστήματος η διεργασία init έχει αντικατασταθεί από τη διεργασία systemd η οποία οδηγεί σε καλύτερη διαχείριση του συστήματος.



Διαχείριση διεργασιών

Μεταβολή προτεραιότητας διεργασίας – η εντολή nice

Η εντολή nice επιτρέπει τη μεταβολή της τιμής προτεραιότητας των διεργασιών. Η προεπιλεγμένη τιμή προτεραιότητας είναι η τιμή 0, η μέγιστη τιμή είναι -20 και η ελάχιστη είναι η 19.

Όσο μεγαλύτερη είναι η τιμή της προτεραιότητας μιας διεργασίας τόσο πιο μεγάλος είναι ο χρόνος που απασχολεί τη CPU και τόσο μεγαλύτερη είναι η εκχώρηση πόρων.

Η κλήση της nice χωρίς ορίσματα εκτυπώνει την τρέχουσα τιμή προτεραιότητας (συνήθως 0) ενώ η κλήση της εντολής με τη μορφή

nice –nN cmd

προκαλεί αύξηση της τιμής της προτεραιότητας της διεργασίας που προκύπτει από την εκτέλεση της εντολής cmd κατά N. Για παράδειγμα η εντολή

nice –n10 gedit sample.txt

ξεκινά την εφαρμογή gedit με τιμή προτεραιότητας 0 (default) + 10 = 10 καθιστώντας τη διεργασία που δημιουργείται λιγότερο επείγουσα σε σχέση με τις υπόλοιπες.

Σε επίπεδο κώδικα C η συνάρτηση nice ορίζεται ως **int nice (int increment)**.

Διαχείριση διεργασιών

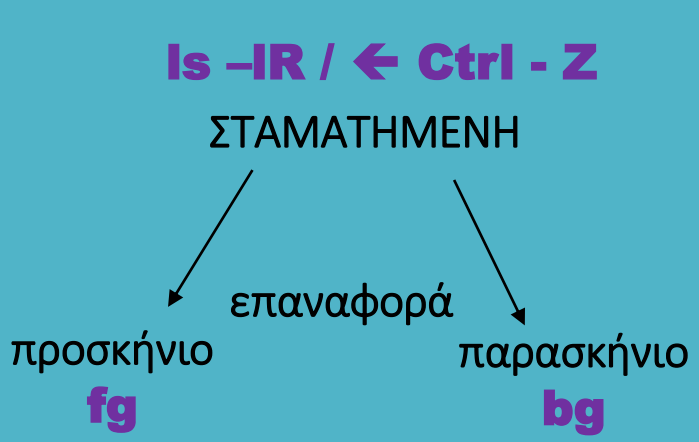
Προσκήνιο και παρασκήνιο

Μία διεργασία εκτελείται στο προσκήνιο (foreground) όταν εκτελείται από τη γραμμή εντολών και κατά συνέπεια απασχολεί το φλοιό.

Μία διεργασία εκτελείται στο παρασκήνιο (background) όταν δεν δεσμεύει το φλοιό και ως εκ τούτου παράλληλα με την εκτέλεσή της μπορούμε μέσω του φλοιού να αλληλεπιδράσουμε με το σύστημα. Μια διεργασία μπορεί να ξεκινήσει απευθείας στο παρασκήνιο με τη βοήθεια του τελεστή & που καταχωρείται ως τελευταίος χαρακτήρας κατά την κλήση της δηλαδή ως

cmd [arguments] &

Εάν επιθυμούμε την αναστολή της διεργασίας που εκτελείται χρησιμοποιούμε το συνδυασμό Ctrl-Z.




```
amarg@amarg-vbox:~/Desktop$ ps -x | sort -k1 -r -n
  13839 pts/0    S+     0:00 sort -k1 -r -n
  13838 pts/0    R+     0:00 ps -x
  13831 pts/0    T      0:00 ls --color=auto -lR /
  13825 pts/0    Ss     0:00 bash
```

```
amarg@amarg-vbox:~/Desktop$ jobs
[1]+  Stopped                 ls --color=auto -lR /
```

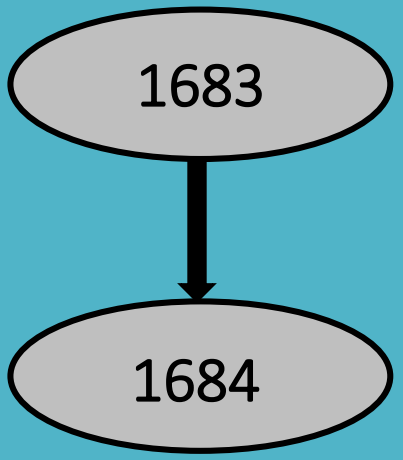
Το job ως έννοια σχετίζεται με το φλοιό

Διαχείριση διεργασιών

Παράδειγμα 1

Δημιουργία θυγατρικής διεργασίας με κλήση της fork().

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



Η printf εκτελείται και από τη γονική και από τη θυγατρική διεργασία (για αυτό και εκτυπώνονται δύο μηνύματα ενώ υπάρχει μία και μοναδική printf) και εκτυπώνει το pid της καθεμίας από αυτές.

```
amarg@amarg-vbox:~$ ./shared/Lab4/forkex1
Hello !! my pid is 1683
amarg@amarg-vbox:~$ Hello !! my pid is 1684
```

Η γονική διεργασία ολοκληρώθηκε πρώτη (αν και θα μπορούσε πρώτη να ολοκληρωθεί η θυγατρική) και για το λόγο αυτό το shell prompt εμφανίστηκε πριν την ολοκλήρωση της θυγατρικής διεργασίας.

Διαχείριση διεργασιών

Παράδειγμα 2

```c
#include <unistd.h>     /* for fork(), getpid() */
#include <sys/types.h>  /* for waitpid() */
#include <sys/wait.h>   /* for waitpid() */
#include <stdlib.h>     /* for exit() */
#include <stdio.h>      /* for printf(), perror() */

int main(int argc, char *argv[]) {
    pid_t child_pid = fork();
    if (child_pid < 0) {
        perror("fork() failed");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        printf("from child: pid=%d, parent_pid=%d\n", (int) getpid(), (int) getppid());
        exit(33);
    } else if (child_pid > 0) {
        printf("from parent: pid=%d child_pid=%d\n", (int) getpid(), (int) child_pid);
        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) {
            if (WIFEXITED(status)) {
                printf("from parent: child exited with code %d\n", WEXITSTATUS(status)); }}}
    exit(EXIT_SUCCESS); }
```

OUTPUT

```
amarg@amarg-vbox:~$ ./shared/Lab4/forkex2
from parent: pid=1804 child_pid=1805
from child: pid=1805, parent_pid=1804
from parent: child exited with code 33
```

Ο γονέας περιμένει τον τερματισμό του παιδιού

Διαχείριση διεργασιών

Παράδειγμα 3

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

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

        if (child_pid == 0) /* fork() returns 0 for the child process */ {
            printf("child process!\n");
            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};
            return execv("/usr/bin/whoami",cmd); /* call whoami command */ }

         else /* parent process */ {
             printf("parent process!\n");
             printf("parent PID =  %d, child pid = %d\n", getpid(), child_pid);
             wait(&status); /* wait for child to exit, and store child's exit status */
             printf("Child exit code: %d\n", WEXITSTATUS(status));
             /* The change in local and global variable in child process */
             /* should not reflect here in parent process. */
             printf("\nParent'z local = %d, parent's  global = %d\n",local,global);
             printf("Parent says bye!\n");
             exit(0);  /* parent exits */ }}

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

```
parent process!
parent PID =  1843, child pid = 1844
child process!
child PID =  1844, parent pid = 1843

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

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

OUTPUT

Οι μεταβλητές local και global έχουν διαφορετικές τιμές στις δύο διεργασίες !

Διαχείριση διεργασιών

Η δομή του κώδικα χρήσης της fork

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

int main() {
    pid_t pid, pid1;

A   Κώδικας πριν την κλήση της fork.
    Υπάρχει μία και μοναδική διεργασία
    η οποία είναι η γονική διεργασία

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;}

    else if (pid == 0) { /* child process */

         Γ  Κώδικας που εκτελείται από
            τη θυγατρική διεργασία
    }
    else { /* parent process */

         B  Κώδικας που εκτελείται
            από τη γονική διεργασία
    }

    return 0;}
```

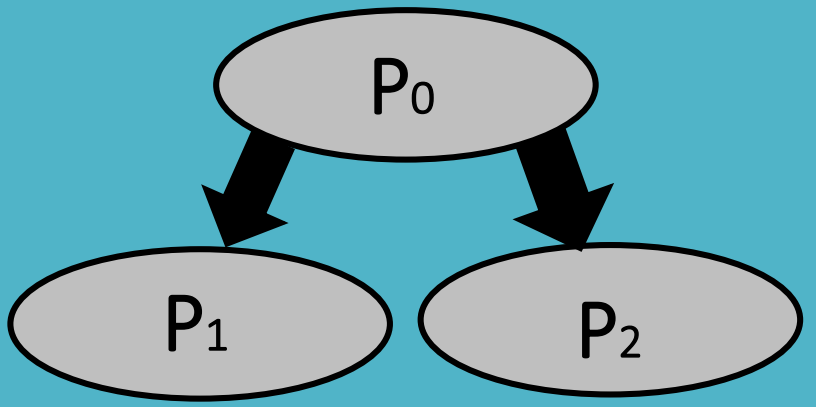
Εάν λοιπόν θέλουμε να γράψουμε κώδικα που να εκτελείται από τη γονική διεργασία, αυτός μπορεί να γραφεί είτε στην Ενότητα Α είτε στην Ενότητα Β, οι οποίες είναι ισοδύναμες.

Αντίθετα ο κώδικας για τη θυγατρική διεργασία γράφεται ΜΟΝΟ στην Ενότητα Γ

Διαχείριση διεργασιών

Τοπολογίες διεργασιών

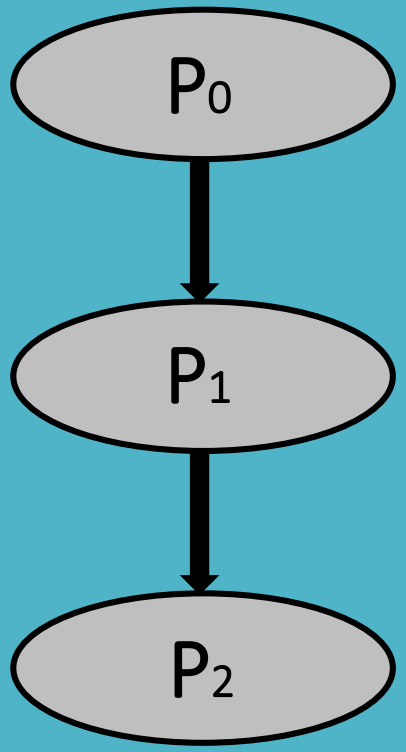
```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main () {
    int pid1, pid2;
    pid1 = fork();
    if (pid1>0) {
                        Κώδικας για τη
                        διεργασία P0

        pid2 = fork();
        if (pid2>0)
            {
                        Κώδικας για τη
                        διεργασία P0
            }
         else {
                        Κώδικας για τη
                        διεργασία P2
              }
     }
    else {
                        Κώδικας για τη
                        διεργασία P1
          }}
```



Διαχείριση διεργασιών

Τοπολογίες διεργασιών

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

int main () {
    int pid1, pid2;
    pid1 = fork();
    if (pid1>0) {
                        Κώδικας για τη
                        διεργασία P0
                }
    else {
                                    Κώδικας για τη
                                    διεργασία P1
            pid2 = fork();
            if (pid2>0)
              {
                                    Κώδικας για τη
                                    διεργασία P1
              }
            else {
                                    Κώδικας για τη
                                    διεργασία P2
                }

                    Κώδικας για τη
                    διεργασία P1
        }}
```



Διαχείριση διεργασιών

Παράδειγμα 4

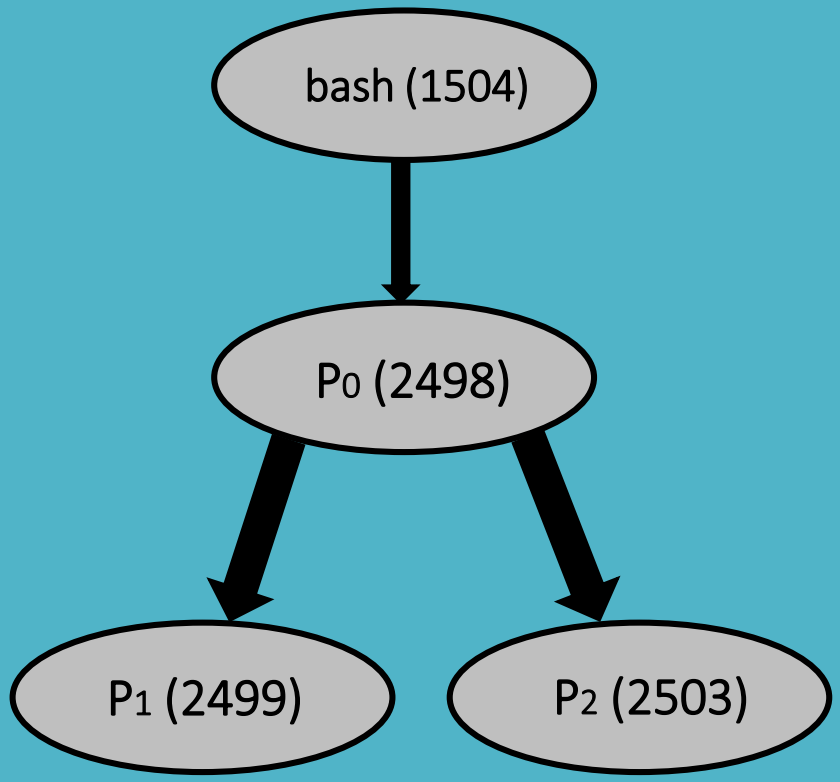
```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

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);
          pid2 = fork();
          if (pid2>0)
              {
                printf ("Parent process \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); }}
    else {
            printf ("Child1 with pid %d kai ppid %d \n", getpid(), getppid());
            printf ("Creating a new directory...\n");
            execlp ("mkdir", "mkdir", "OSLab", NULL); }}
```

Διαχείριση διεργασιών

Παράδειγμα 4

```
amarg@amarg-vbox:~$ shared/Lab4/forkex4
Parent process with pid 2498 and ppid 1504
Child1 with pid 2499 kai ppid 2498
Creating a new directory...
Child with code 2499 has beem terminated
Parent process
Child2 with pid 2503 and ppid 2498
/home/amarg
Child with code 2503 has been terminated
```

```
amarg@amarg-vbox:~$ ps
    PID TTY          TIME CMD
   1504 pts/0    00:00:00 bash
   2504 pts/0    00:00:00 ps
```



Διαχείριση διεργασιών

Παράδειγμα 5

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

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 {
            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); }}
```

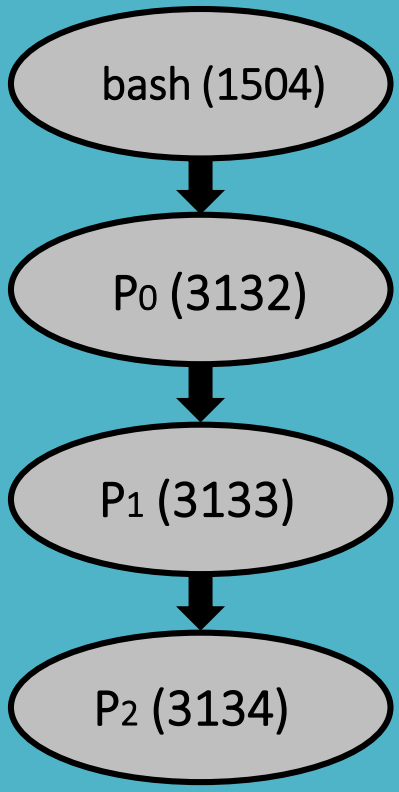
Διαχείριση διεργασιών

Παράδειγμα 5

```
amarg@amarg-vbox:~$ ./shared/lab4/forkex5
Parent process with pid 3132 and ppid 1504
Child1 with pid 3133 kai ppid 3132
Parent process (child1) for child2
Child2 with pid 3134 and ppid 3133
/home/amarg
Child with code 3134 has been terminated
Creating a new directory...
mkdir: cannot create directory ‘OSLab’: File exists
Child with code 3133 has beem terminated
```

```
 1496 ?        Ssl    0:14 /usr/libexec/gnome-terminal
 1504 pts/0    Ss     0:00 bash
 1541 ?        Ssl    0:00 /usr/libexec/gvfsd-metadata
 1544 ?        Sl     0:01 update-notifier
 2627 ?        Sl     0:00 /usr/libexec/gvfsd-network
 2646 ?        Sl     0:00 /usr/libexec/gvfsd-dnssd --
 2759 pts/1    Ss     0:00 bash
 2958 pts/2    Ss+    0:00 bash
 3144 ?        Ssl    0:00 /usr/libexec/tracker-store
 3151 pts/0    S+     0:00 ./forkex6
 3152 pts/0    S+     0:00 ./forkex6
 3153 pts/0    S+     0:00 ./forkex6
```

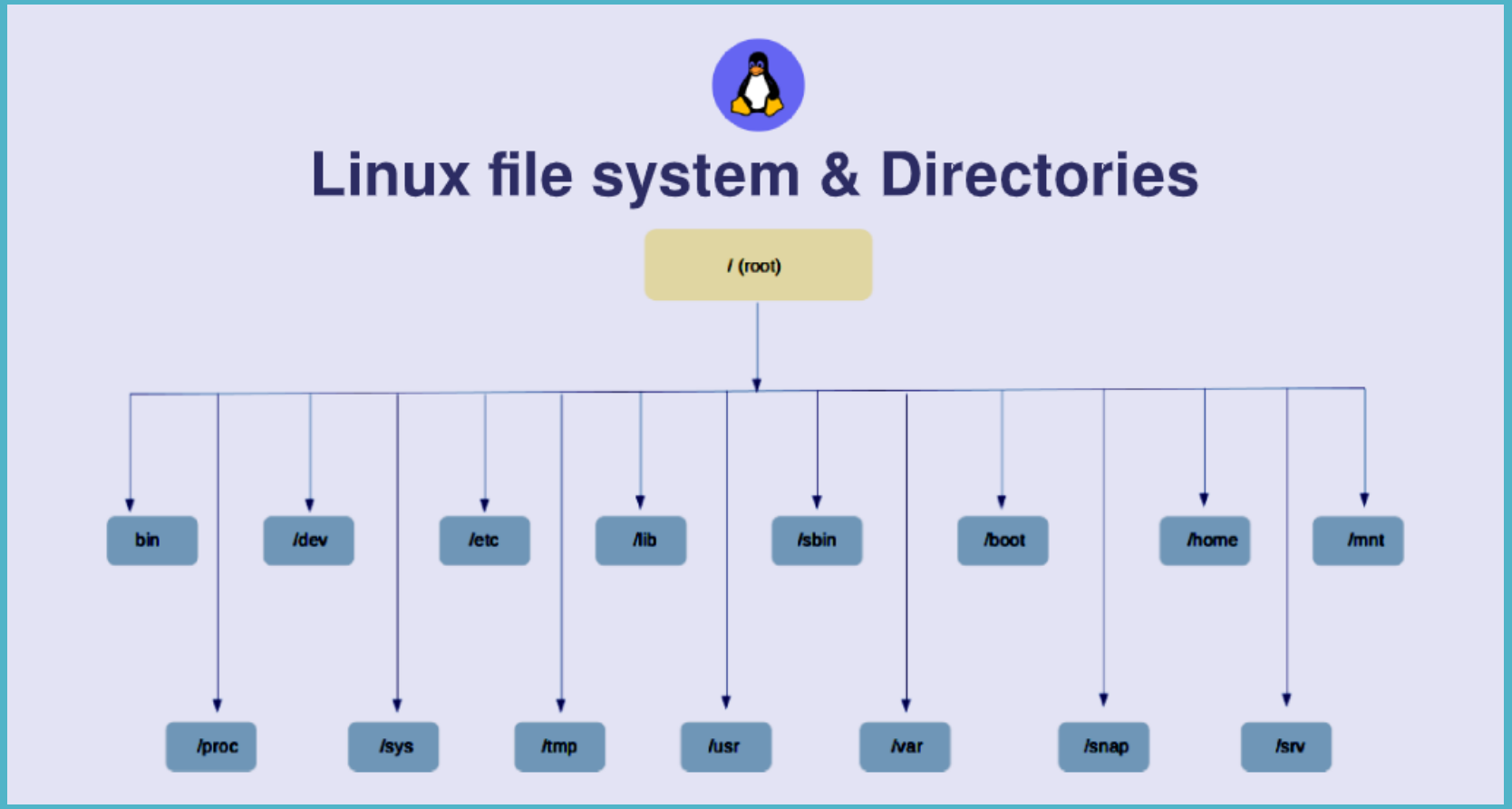
Διαφορετικά pids για διαφορετικές εκτελέσεις



Έξοδος της ps -xelf

```
0  1000    1496     861  20   0 826352 54472 -      Ssl  ?        0:14 \_ /usr/libexec/gnome-terminal-server HOME=/h
0  1000    1504    1496  20   0  19644  5388 do_wai Ss   pts/0    0:00 |   \_ bash GJS_DEBUG_TOPICS=JS ERROR;JS LOG
0  1000    3151    1504  20   0   2488   580 do_wai S+   pts/0    0:00 |   |   \_ ./forkex6 SHELL=/bin/bash SESSION_
1  1000    3152    3151  20   0   2488    76 do_wai S+   pts/0    0:00 |   |       \_ ./forkex6 SHELL=/bin/bash SESS
1  1000    3153    3152  20   0   2488    84 hrtime S+   pts/0    0:00 |   |           \_ ./forkex6 SHELL=/bin/bash
```

Αρχεία και κατάλογοι

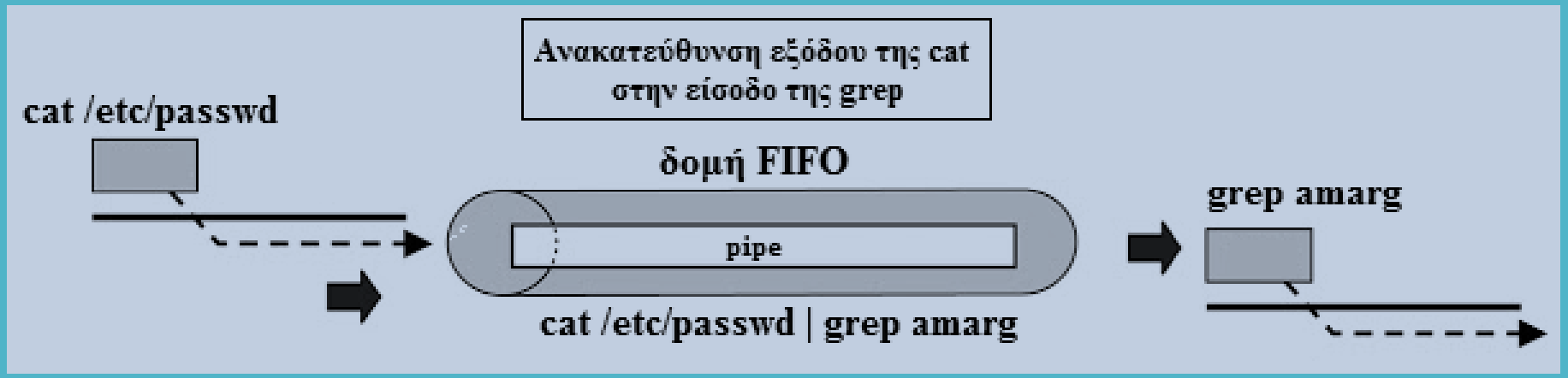


Αρχεία και κατάλογοι

Τα αρχεία αποτελούν μία από τις θεμελιώδεις δομικές μονάδες του λειτουργικού συστήματος Linux, αφού σχεδόν τα πάντα σε αυτό το λειτουργικό σύστημα, περιγράφονται από κατάλληλα αρχεία. Ειδικότερα, στο Linux υφίστανται οι επόμενοι τύποι αρχείων:

1) Συνήθη αρχεία (regular files)→ πρόκειται για τα γνωστά αρχεία των χρηστών που αποτελούν ομάδες bytes αποθηκευμένες στο δίσκο.

2) Αγωγοί ή σωληνώσεις (pipes) → πρόκειται για ειδικά αρχεία τα οποία μπορεί να είναι αποθηκευμένα στο δίσκο του συστήματος αλλά μπορεί και όχι, τα οποία επιτρέπουν την ενδοεπικοινωνία ανάμεσα στις διεργασίες του συστήματος (Interprocess Communication, IPC).



Αρχεία και κατάλογοι

3) Κατάλογοι (directories) → Πρόκειται για αρχεία που αποτελούνται από τη λίστα των αρχείων που περιέχονται σε αυτά.

4) Αρχεία συσκευών (device files) → πρόκειται για ειδικά αρχεία που χρησιμοποιούνται για την αναπαράσταση και προσπέλαση των συσκευών του συστήματος.

```
amarg@amarg-vbox:~$ ls /dev
autofs          ecryptfs  log            net       sg0       tty12  tty25  tty38  tty50  tty63      ttyS17  ttyS3   vboxguest  vcsa5  zero
block           fb0       loop0          null      sg1       tty13  tty26  tty39  tty51  tty7       ttyS18  ttyS30  vboxuser   vcsa6  zfs
bsg             fd        loop1          nvram     shm       tty14  tty27  tty4   tty52  tty8       ttyS19  ttyS31  vcs        vcsu
btrfs-control   full      loop2          port      snapshot  tty15  tty28  tty40  tty53  tty9       ttyS2   ttyS4   vcs1       vcsu1
bus             fuse      loop3          ppp       snd       tty16  tty29  tty41  tty54  ttyprintk  ttyS20  ttyS5   vcs2       vcsu2
cdrom           hidraw0   loop4          psaux     sr0       tty17  tty3   tty42  tty55  ttyS0      ttyS21  ttyS6   vcs3       vcsu3
char            hpet      loop5          ptmx      stderr    tty18  tty30  tty43  tty56  ttyS1      ttyS22  ttyS7   vcs4       vcsu4
console         hugepages loop6          pts       stdin     tty19  tty31  tty44  tty57  ttyS10     ttyS23  ttyS8   vcs5       vcsu5
core            hwrng     loop7          random    stdout    tty2   tty32  tty45  tty58  ttyS11     ttyS24  ttyS9   vcs6       vcsu6
cpu_dma_latency i2c-0     loop-control   rfkill    tty       tty20  tty33  tty46  tty59  ttyS12     ttyS25  udmabuf vcsa       vfio
cuse            initctl   mapper         rtc       tty0      tty21  tty34  tty47  tty6   ttyS13     ttyS26  uhid    vcsa1      vga_arbiter
disk            input     mcelog         rtc0      tty1      tty22  tty35  tty48  tty60  ttyS14     ttyS27  uinput  vcsa2      vhci
dri             kmsg      mem            sda       tty10     tty23  tty36  tty49  tty61  ttyS15     ttyS28  urandom vcsa3      vhost-net
dvd             lightnvm  mqueue         sda1      tty11     tty24  tty37  tty5   tty62  ttyS16     ttyS29  userio  vcsa4      vhost-vsock
```

5) Συμβολικοί σύνδεσμοι (symbolic links) → πρόκειται για αρχεία που περιέχουν τη διαδρομή προς ένα άλλο αρχείο του συστήματος (αντιστοιχούν στα shortcuts των Microsoft Windows).

```
amarg@amarg-vbox:/$ ls -lR / | grep '^l'
lrwxrwxrwx   1 root root          7 Σεπ  19 19:50 bin -> usr/bin
lrwxrwxrwx   1 root root          7 Σεπ  19 19:50 lib -> usr/lib
```

6) Υποδοχείς (sockets) → πρόκειται για αρχεία που επιτρέπουν την ενδοεπικοινωνία των διεργασιών με πιο αποδοτικό και ευέλικτο τρόπο σε σχέση με τους αγωγούς.

Αρχεία και κατάλογοι

- Το κάθε αρχείο του συστήματος προσδιορίζεται με μοναδικό τρόπο από μία ειδική δομή δεδομένων που λέγεται i-node (index node ή information node) και το οποίο περιέχει όλες τις πληροφορίες του αρχείου (δικαιώματα πρόσβασης, μέγεθος, όνομα, κ.τ.λ.).

- Υπάρχουν δύο τύποι i-node, τα i-node μνήμης που διατηρούνται για κάθε ανοιχτό αρχείο και τα i-nodes δίσκου που διατηρούνται για κάθε αρχείο του δίσκου.

- Αυτά τα δύο είδη i-nodes δεν περιέχουν τις ίδιες πληροφορίες.

  - Όταν ανοίγουμε ένα αρχείο το i-node δίσκου αντιγράφεται σε ένα inode-μνήμης.
  - Όταν αποθηκεύουμε ένα αρχείο το i-μνήμης αντιγράφεται σε ένα inode-δίσκου.

```
amarg@amarg-vbox:/$ ls -li
total 970048
     13 lrwxrwxrwx   1 root root         7 Σεπ  19 19:50 bin -> usr/bin
 786433 drwxr-xr-x   3 root root      4096 Σεπ  26 09:47 boot
 802791 drwxrwxr-x   2 root root      4096 Σεπ  19 19:55 cdrom
      2 drwxr-xr-x  18 root root      4000 Σεπ  28 12:37 dev
 262145 drwxr-xr-x 130 root root     12288 Σεπ  26 09:55 etc
 131073 drwxr-xr-x   3 root root      4096 Σεπ  19 19:57 home
     14 lrwxrwxrwx   1 root root         7 Σεπ  19 19:50 lib -> usr/lib
     15 lrwxrwxrwx   1 root root         9 Σεπ  19 19:50 lib32 -> usr/lib32
     16 lrwxrwxrwx   1 root root         9 Σεπ  19 19:50 lib64 -> usr/lib64
     17 lrwxrwxrwx   1 root root        10 Σεπ  19 19:50 libx32 -> usr/libx32
```

Αρχεία και κατάλογοι

Οι πληροφορίες που είναι αποθηκευμένες σε ένα i-node επιστρέφονται ως τα πεδία της επόμενης δομής.

```
struct stat {
        dev_t st_dev;          ID of device containing the file
        ino_t st_ino;          Serial number for the file.
        mode_t st_mode;        Access mode and file type for the file (see Flags).
        nlink_t st_nlink;      Number of links to the file.
        uid_t st_uid;          User ID of file owner.
        gid_t st_gid;          Group ID of group owner.
        dev_t st_rdev;         Device ID (if the file is a character or block special device).
        off_t st_size;         File size in bytes (if the file is a regular file).
        time_t st_atime;       Time of last access.
        time_t st_mtime;       Time of last data modification.
        time_t st_ctime;       Time of last file status change.
        blksize_t st_blksize;  A file system-specific preferred I/O block size for this object.
        blkcnt_t st_blocks;    Number of blocks allocated for this file.
        mode_t st_attr;        The DOS-style attributes for this file (see Flags).
};
```

Αρχεία και κατάλογοι

mode_t st_mode values for file type

Flag	Meaning	
S_IFMT	Type of file.	Η τιμή στο οκταδικό σύστημα
S_IFBLK	File is a block-device special file.	0060000
S_IFCHR	File is a character-device special file.	0020000
S_IFIFO	File is a FIFO special file.	0010000
S_IFREG	File is a regular file.	0100000
S_IFDIR	File is a directory.	0040000
S_IFSOCK	File is a socket.	0140000

Αρχεία και κατάλογοι

Μακροεντολές για τον έλεγχο της κατάστασης του αρχείου (επιστρέφουν TRUE ή FALSE)

Macro	Meaning
S_ISBLK (m)	File is a block-device special file.
S_ISCHR (m)	File is a character-device special file.
S_ISFIFO (m)	File is a FIFO special file.
S_ISREG (m)	File is a regular file.
S_ISDIR (m)	File is a directory.
S_ISSOCK (m)	File is a socket.

Αρχεία και κατάλογοι

Flag	Meaning	
S_IRWXU	Read, write, execute/search permission for owner.	**USER**
S_IRUSR	Read permission for owner.	r
S_IWUSR	Write permission for owner.	w
S_IXUSR	Execute/search permission for owner.	x
S_IRWXG	Read, write, execute/search for group.	**GROUP**
S_IRGRP	Read permission for group.	r
S_IWGRP	Write permission for group.	w
S_IXGRP	Execute/search permission for group.	x
S_IRWXO	Read, write, execute/search for others.	**OTHERS**
S_IROTH	Read permission for others.	r
S_IWOTH	Write permission for others.	w
S_IXOTH	Execute/search permission for others.	x

mode_t st_mode values for permission

Flag	Meaning	
S_IDPOU	Delete, change permission, and take ownership for owner.	**USER**
S_IDUSR	Delete permission for owner.	
S_IPUSR	Change permission permission for owner.	
S_IOUSR	Take ownership permission for owner.	
S_IDPOG	Delete, change permission, and take ownership for group.	**GROUP**
S_IDGRP	Delete permission for group.	
S_IPGRP	Change permission permission for group.	
S_IOGRP	Take ownership permission for group.	
S_IDPOO	Delete, change permission, and take ownership for others.	**OTHERS**
S_IDOTH	Delete permission for others.	
S_IPOTH	Change permission permission for others.	
S_IOOTH	Take ownership permission for others.	
S_ISUID	Set-user-ID on execution	**setuid bit**
S_ISGID	Set-group-ID on execution	**setgid bit**
S_ISVTX	Restricted-deletion flag for directories	**sticky bit**

Αρχεία και κατάλογοι

Παράδειγμα εμφάνισης πληροφοριών αρχείου

```
#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;
   if (argc != 2) {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        exit(EXIT_FAILURE); }
   if (stat(argv[1], &sb) == -1) {
        perror("stat");
        exit(EXIT_FAILURE); }

   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; }

   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); }
```

Αρχεία και κατάλογοι

Παράδειγμα εμφάνισης πληροφοριών αρχείου

```
amarg@amarg-vbox:~$ ./filestat forkex5
File type:                regular file
I-node number:            274381
Mode:                     100775 (octal)
Link count:               1
Ownership:                UID=1000   GID=1000
Preferred I/O block size: 4096 bytes
File size:                17048 bytes
Blocks allocated:         40
Last status change:       Sat Sep 26 20:49:56 2020
Last file access:         Sat Sep 26 20:49:59 2020
Last file modification:   Sat Sep 26 20:49:56 2020
```

```
amarg@amarg-vbox:~$ ./filestat Music
File type:                directory
I-node number:            935608
Mode:                     40755 (octal)
Link count:               2
Ownership:                UID=1000   GID=1000
Preferred I/O block size: 4096 bytes
File size:                4096 bytes
Blocks allocated:         8
Last status change:       Sat Sep 19 20:09:47 2020
Last file access:         Mon Sep 28 12:37:51 2020
Last file modification:   Sat Sep 19 20:09:47 2020
```

```
amarg@amarg-vbox:~$ ls -l forkex5
-rwxrwxr-x 1 amarg amarg 17048 Σεπ  26 20:49 forkex5
```

```
amarg@amarg-vbox:~$ ls -l | grep Music
drwxr-xr-x  2 amarg amarg     4096 Σεπ  19 20:09 Music
```

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

ΔΙΑΠΙΣΤΩΝΟΥΜΕ ΟΤΙ

1) Η κατάσταση (mode) αποτελείται από: (α) τύπο αρχείου, (β) τροποποιητής, (γ) δικαιώματα πρόσβασης
2) Το όνομα του αρχείου ΔΕΝ περιλαμβάνεται στις πληροφορίες του i-node (struct stat).

Mode → 10 0 775
10 → regular file
0 → τροποποιητής (sgt)
775 → r w x r w x r – x

Mode → 04 0 755
04 → directory
0 → τροποποιητής (sgt)
755 → r w x r – x r – x

Επομένως η εντολή ls –l στην πραγματικότητα εκτυπώνει τις τιμές των πεδίων της δομής stat.

Αρχεία και κατάλογοι

Περιγραφέας αρχείου (File Descriptor, fd)

Οι περιγραφείς αρχείου είναι θετικοί ακέραιοι αριθμοί με μικρές τιμές οι οποίοι για κάθε ανοικτό αρχείο υποδεικνύουν τη θέση του στον πίνακα των ανοικτών αρχείων που διατηρεί η κάθε διεργασία.

Κατά την εκκίνηση κάθε διεργασίας αρχικοποιούνται οι επόμενοι περιγραφείς αρχείου

0 (STDIN_FILENO) → standard input (stdin)
1 (STDOUT_FILENO) → standard output (stdout)
2 (STDERR_FILENO) → standard error (stderr)

Για αρκετές από τις συναρτήσεις διαχείρισης αρχείων υφίστανται δύο εκδοχές στην πρώτη εκ των οποίων το αρχείο καθορίζεται από το όνομά του (char * pathname), ενώ στη δεύτερη από τον περιγραφέα αρχείου του (int fd). Για παράδειγμα, η συνάρτηση stat που επιστρέφει μία δομή stat με τις πληροφορίες του i-node ενός αρχείου εμφανίζεται στις επόμενες δύο εκδοχές.

```
int stat (const char * pathname, struct stat * statbuf);
int stat (int fd, struct stat * statbuf);
```

Αρχεία και κατάλογοι

Δικαιώματα πρόσβασης, κάτοχος και ομάδα αρχείου, time stamps

Αν και στη δομή stat υπάρχουν όλες οι πληροφορίες σχετικά με τα δικαιώματα πρόσβασης για το υπό θεώρηση αρχείο, ωστόσο, η ανάκτησή τους διευκολύνεται σημαντικά από τη συνάρτηση

int access (const char * pathname, int mode) ← <unistd.h>

όπου mode μία ή περισσότερες από τις παρακάτω σταθερές συνδυασμένες με τον τελεστή της λογικής διάζευξης (logical OR)

F_OK → το αρχείο υπάρχει?
R_OK → υφίσταται δικαίωμα ανάγνωσης?
W_OK → υφίσταται δικαίωμα εγγραφής?
X_OK → υφίσταται δικαίωμα εκτέλεσης?

Η συνάρτηση επιστρέφει 0 εάν το δικαίωμα πρόσβασης υφίσταται και τη σταθερά EACCESS στην αντίθετη περίπτωση.

Ορισμός δικαιωμάτων πρόσβασης από τον κάτοχο (owner) του αρχείου ή το διαχειριστή (root)

int chmod (const char * pathname, mode_t mode);
int fchmod (int fd, mode_t mode); **← <sys/stat.h>**

Αρχεία και κατάλογοι

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

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

 /* Check file existence. */
 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;  }

 /* Check read access. */
 rval = access (path, R_OK);
 if (rval == 0)
  printf ("%s is readable\n", path);
 else
  printf ("%s is not readable (access denied)\n", path);

 /* Check write access. */
 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; }
```

```
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
--wxr--r-- 1 amarg amarg 7959821 Σεπ  20 19:58 file2.doc
-r-xr--r-- 1 amarg amarg    2797 Σεπ  20 19:59 file3.doc
-rwxr--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 χρήσης της access

Αρχεία και κατάλογοι

```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;
  if (argc!=2) {
      printf ("Usage testChmod pathname\n");
      return (-1); }
  fd = access(argv[1], F_OK);
  if(fd == -1){
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-2); }
  else {
    stat(argv[1], &info);
    printf("Original file permissions were: %08o\n",
           info.st_mode);
    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); }
```

```
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
```

Αρχικά γίνεται έλεγχος ύπαρξης του αρχείου με την access και στη συνέχεια καλείται η chmod για να δώσει δικαιώματα r w x στον κάτοχο (S_IRWXU) και στην ομάδα (S_IRWXG). Παρατηρήστε πως αλλάξει όλη η μάσκα δικαιωμάτων και επειδή δεν έχουν οριστεί δικαιώματα για τους υπόλοιπους χρήστες, αυτά έχουν απενεργοποιηθεί.

Παράδειγμα χρήσης της chmod

Αρχεία και κατάλογοι

Δικαιώματα πρόσβασης, κάτοχος και ομάδα αρχείου, time stamps

Τροποποίηση κατόχου και ομάδας κατόχου αρχείου (<unistd.h>)

int chown (const char * pathname, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);

Τροποποίηση χρονικών σφραγίδων (st_mtime & st_atime)

Α τρόπος (System V → POSIX)
<utime.h>

```
struct utimbuf {
        time_t actime;
        time_t modtime; };
```

int utime (const char * pathname,
struct utimebuf * buf)

Β Τρόπος (BSD Unix)
<systime/h>

```
struct timeval {
        long tv_sec;
        long tv_usec};
```

int utime (const char * pathname,
struct timeval * tvf);

Προεπιλεγμένη μάσκα δικαιωμάτων (sys/stat.h)

int umask (int newmask);

Αρχεία και κατάλογοι

```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;
    if (argc !=2) {
        printf ("Usage: testChown pathname\n");
        return (-1); }
    retVal = chown (argv[1], (uid_t)3, (gid_t)3);
    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); }
```

```
amarg@amarg-vbox:~$ ls -l | grep months.txt
-rw-rw-r--  1 amarg amarg     87 Σεπ  23 14:44 months.txt
amarg@amarg-vbox:~$ ./testChown months.txt
Error Number : 1
Error Description: Operation not permitted
amarg@amarg-vbox:~$ sudo ./testChown months.txt
Assignment to months.txt of user SYS and group SYS
                    has been performed succesfully
amarg@amarg-vbox:~$ ls -l | grep months.txt
-rw-rw-r--  1 sys   sys       87 Σεπ  23 14:44 months.txt
amarg@amarg-vbox:~$
```

Η επιτυχής κλήση της συνάρτησης απαιτεί δικαιώματα διαχειριστή και για το λόγο αυτό εκτελείται μέσα από την εντολή sudo (switch user, do) η οποίο by default προσφέρει root access. Στο παράδειγμα ο νέος owner είναι ο user sys με user id ίσο με 3.

```
amarg@amarg-vbox:~$ cat /etc/passwd | grep sys
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Παράδειγμα χρήσης της chown

Αρχεία και κατάλογοι

Οι βασικές μορφές διαχείρισης αρχείων

Άνοιγμα / δημιουργία αρχείου – οι εντολές open και creat ← <unistd.h> & <fcntl.h>

int open (char * pathname, int flags, mode_t mode);

- pathname → το όνομα της διαδρομής προς το αρχείο
- flags → το είδος της πρόσβασης [απαιτείται κάποιο από τα O_RDONLY (read only), O_WRONLY (write only) ή O_RDWR (read/write) σε συνδυασμό με κάποια άλλα προαιρετικά flags όπως π.χ, το O_APPEND (τα δεδομένα γράφονται στο τέλος του αρχείου)]
- mode → εάν στα flags συμπεριληφθούν τα O_CREAT ή O_TMPFILE τότε ορίζεται το mode που μπορεί να πάρει κάποιες από τις τιμές S_IRWXU , S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH).

int creat (char * pathname, mode_t mode);

Είναι ισοδύναμη με την κλήση της open με flags O_CREAT | O_WRONLY | O_TRUNC .

Οι συναρτήσεις επιστρέφουν τον file descriptor fd προς το αρχείο που άνοιξη ή δημιουργήθηκε. Άλλες συναρτήσεις που επιστρέφουν file descriptors είναι η pipe (που δημιουργεί αγωγούς) και οι socket, accept και connect (network programming).

Κλείσιμο αρχείου → **int close (int fd)** ← <unistd.h>

Αρχεία και κατάλογοι

Οι βασικές μορφές διαχείρισης αρχείων

Ανάγνωση και εγγραφή σε αρχείο – οι συναρτήσεις read και write ← <unistd.h>

Οι εντολές read και write που επιτρέπουν την ανάγνωση δεδομένων από αρχείο και την εγγραφή δεδομένων σε αυτό (όχι όμως για low level διαδικασίες όπως τα message queues), έχουν τη μορφή

int read (int fd, void * buf, size_t length);
int write (int fd, const void * buf, size_t length)

Αμφότερες οι συναρτήσεις επιστρέφουν το πλήθος των bytes που διαβάστηκαν ή εγγράφηκαν (αντίστοιχα) ή την τιμή -1 για την περίπτωση σφάλματος.

Αναζήτηση σε αρχείο τυχαίας προσπέλασης – η συνάρτηση lseek ← <unistd.h>

Η συνάρτηση fseek επιτρέπει τον ορισμό της τρέχουσας θέσης σε αρχείο τυχαίας προσπέλασης και επιστρέφει τη νέα τρέχουσα θέση – η συνάρτηση ορίζεται ως

int lseek (int fd, off_t offset, int whence)

Η παράμετρος whence παίρνει μία από τις τιμές SEEK_SET (αρχή αρχείου), SEEK_CUR (τρέχουσα θέση), SEEK_END (τέλος αρχείου) και δηλώνει από πιο σημείο θα μετακινηθούμε κατά offset bytes με την τιμή του offset να μπορεί να είναι και αρνητική.

Αρχεία και κατάλογοι

Παράδειγμα 1 → Υπολογισμός μεγέθους αρχείου με την lseek

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

```
amarg@amarg-vbox:~/shared/Lab5$ ls -l *.o
-rwxrwxrwx 1 root root 4224 Σεπ  29 11:43 filestat.o
-rwxrwxrwx 1 root root 2528 Σεπ  29 16:30 fsize1.o
-rwxrwxrwx 1 root root 2200 Σεπ  29 17:19 fsize2.o
-rwxrwxrwx 1 root root 2632 Σεπ  29 20:50 myCopy.o
-rwxrwxrwx 1 root root 2800 Σεπ  29 11:43 testAccess.o
-rwxrwxrwx 1 root root 2768 Σεπ  29 15:28 testChmod.o
-rwxrwxrwx 1 root root 2272 Σεπ  29 15:43 testChown.o
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 filestat.o
Size of file filestat.o is 4224 bytes
amarg@amarg-vbox:~/shared/Lab5$ ./fsize1 myCopy.o
Size of file myCopy.o is 2632 bytes
amarg@amarg-vbox:~/shared/Lab5$
```

Η lseek επιστρέφει την απόσταση σε bytes της τρέχουσας θέσης από την αρχή του αρχείου. Εάν ως τρέχουσα θέση οριστεί το τέλος του αρχείου, αυτή η απόσταση είναι προφανώς ίση με το τέλος του αρχείου.

Αρχεία και κατάλογοι

Παράδειγμα 2 → Υπολογισμός μεγέθους αρχείου με τη read

```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];
    if (argc !=2) {
    printf ("Usage: flength pathname\n");
    return (-1); }
    fd = open(argv[1], O_RDONLY);
    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; }
```

```
amarg@amarg-vbox:~/shared/Lab5$ ls -l *.o
-rwxrwxrwx 1 root root 4224 Σεπ  29 11:43 filestat.o
-rwxrwxrwx 1 root root 2528 Σεπ  29 16:30 fsize1.o
-rwxrwxrwx 1 root root 2200 Σεπ  29 17:19 fsize2.o
-rwxrwxrwx 1 root root 2632 Σεπ  29 20:50 myCopy.o
-rwxrwxrwx 1 root root 2800 Σεπ  29 11:43 testAccess.o
-rwxrwxrwx 1 root root 2768 Σεπ  29 15:28 testChmod.o
-rwxrwxrwx 1 root root 2272 Σεπ  29 15:43 testChown.o
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 filestat.o
4224 bytes total
amarg@amarg-vbox:~/shared/Lab5$ ./fsize2 myCopy.o
2632 bytes total
amarg@amarg-vbox:~/shared/Lab5$
```

Σε κάθε επανάληψη το πρόγραμμα διαβάζει BUSIZE bytes από το αρχείο και προσθέτει το πλήθος των bytes που διάβασε στη μεταβλητή total.

Η read επιστρέφει το πλήθος των bytes που διάβασε και επομένως όταν επιστρέψει 0 σημαίνει πως έχει φτάσει στο τέλος του αρχείου. Στην περίπτωση αυτή η total περιέχει το μέγεθος του αρχείου.

Αρχεία και κατάλογοι

Παράδειγμα 3 → Αντιγραφή αρχείου με τις read και write

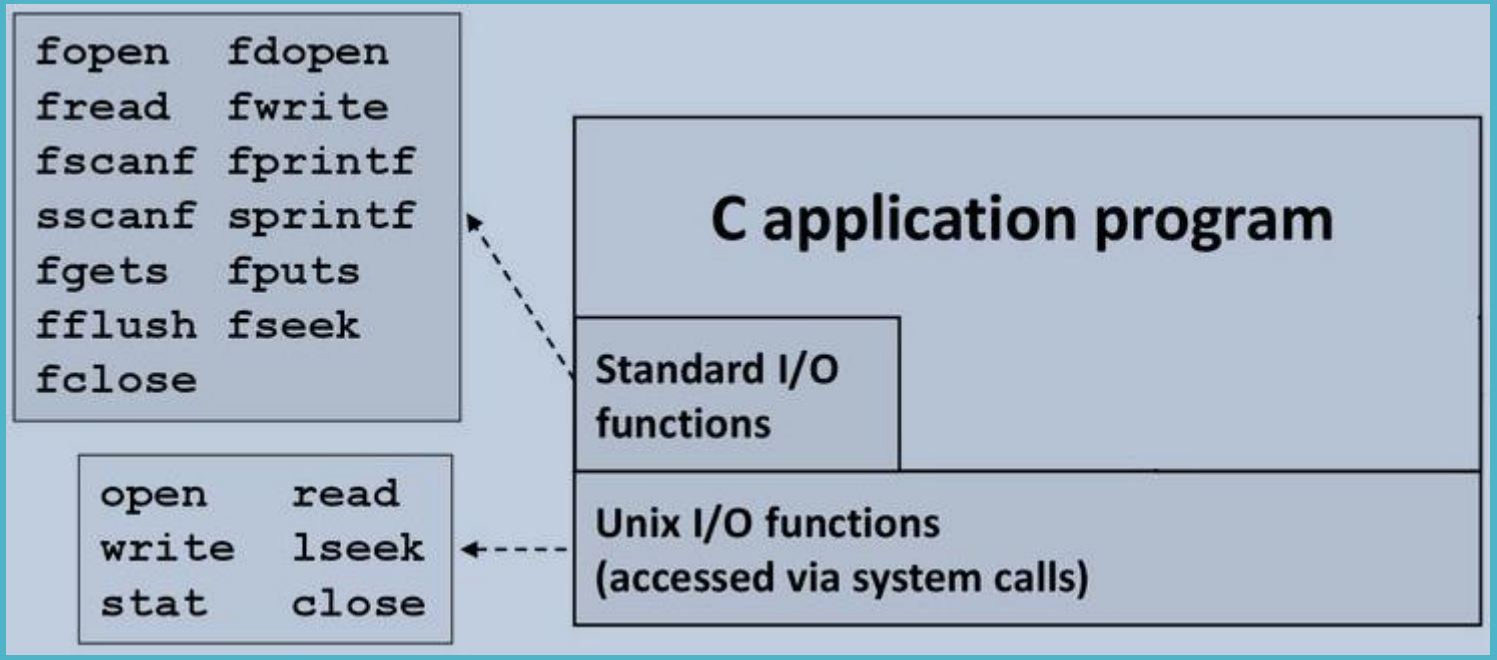
```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#define BUFSIZE 512
```

```c
int main(int argc, char * argv[]) {
    char buffer[BUFSIZE];
    int src, dest, errno;
    int nread, nwritten, n;
    int totalr=0, totalw=0;
    if (argc!=3) {
        printf ("Usage: flength source destination\n");
        return (-1); }
    src = open(argv[1], O_RDONLY);
    if (src < 0) {
        perror("Source file could not be opened!");
        return (-1); }
    dest = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
    if (dest < 0) {
        perror("Target file could not be opened!");
        return (-2); }
    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); }
```

Για όσο υπάρχουν bytes για ανάγνωση, το πρόγραμμα διαβάζει bytes από το αρχείο εισόδου και τα αποθηκεύει στο αρχείο εξόδου

Αρχεία και κατάλογοι

System I/O vs C Language I/O



SYSTEM I/O → Κλήσεις συστήματος, δεν προχωρά σε ενδιάμεση αποθήκευση και μορφοποίηση της εισόδου

C LANGUAGE I/O → Μέρος της βιβλιοθήκης της C με όνομα libc, πραγματοποιεί μορφοποίηση εισόδου και ενδιάμεση αποθήκευση.

Αρχεία και κατάλογοι

File I/O functions

- `#include <stdio.h>`

function	description
`FILE* fopen(char* filename, char* mode)`	mode is "r", "w", "a"; returns pointer to file or NULL on failure
`int fgetc(FILE* file)` `int fgets(char* buf, int size, FILE* file)`	read a char from a file; read a line from a file
`int fputc(char c, FILE* file)` `int fputs(char* s, FILE* file)`	write a char to a file; write a string to a file
`int feof(FILE* file)`	returns non-zero if at EOF
`int fclose(FILE* file)`	returns 0 on success

`FILE* stdin`
`FILE* stdout`
`FILE* stderr`
	streams representing console input, output, and error

- most return EOF on any error (which is -1, but don't rely on that)

4

```
typedef struct

{

    int level;

    unsigned flags;

    char fd;

    unsigned char hold;

    int bsize;

    unsigned char_FAR* buffer;

    unsigned char_FAR* curp;

    unsigned istemp;

    short token;

}FILE;
```

Αρχεία και κατάλογοι

Παράδειγμα 4 → Αντιγραφή αρχείου με συναρτήσεις της C

```c
int main(int argc, char * argv[]) {
    char cTemp;
    FILE * src, * dest;
    if (argc!=3) {
        printf ("Usage: flength 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); }
```

```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>
```

Συναρτήσεις

fopen
fclose
fread
fwrite

Αρχεία και κατάλογοι

Εντολές διαχείρισης καταλόγων

Ανάκτηση καταλόγου εργασίας (unistd.h) → `char * getcwd (char * buf, size_t size)`

Αλλαγή τρέχοντος καταλόγου (unistd.h) →

`int chdir (const char * pathname) & int chdir (int df)`

Αλλαγή ριζικού καταλόγου (unistd.h) → `int chroot (const char * pathname)`

Δημιουργία καταλόγου (unistd.h, fcntl) →

`int mkdir (const char * pathname, mode_t mode)`

Διαγραφή κενού καταλόγου (unistd.h) → `int rmdir (const char * pathname)`

Ανάγνωση περιεχομένων καταλόγου (δομή DIR ← dirent.h)

```
DIR * opendir (const char * pathname)
int closedir (DIR * dir)
struct dirent * readdir (DIR * dir)
```
← **επιστροφή του επόμενου αρχείου στον κατάλογο**

Αρχεία και κατάλογοι

Παράδειγμα 1 → Δημιουργία και διαγραφή καταλόγου

```c
int main(int argc, char * argv[]) {
    long size; int retVal; DIR * dir;
    char * buf, * ptr, arg [20]="ls -l | grep ";
    if (argc!=2) {
        printf ("Usage: dirFun dirname\n");
        return (-1); }
    dir = opendir(argv[1]);
    if (dir) {
        printf ("Directory exists. Aborting...\n");
    closedir(dir);
    return (-2);  }
    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);
    retVal = mkdir (argv[1],0755);
    if (retVal==-1) {
        printf("Error Number : %d\n", errno);
        perror("Error Description");
        return (-2); }
    strcat (arg, argv[1]);
    system (arg);
    printf ("Removing directory %s\n", argv[1]);
    rmdir (argv[1]);
    return (0); }
```

```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>
```

```
amarg@amarg-vbox:~$ ./dirFun mydir
Current Working Directory is /home/amarg
Creating directory mydir inside directory /home/amarg
drwxr-xr-x  2 amarg amarg    4096 Σεπ  30 16:03 mydir
Removing directory mydir
```

Επίδειξη των συναρτήσεων mkdir και rmdir

Αρχεία και κατάλογοι

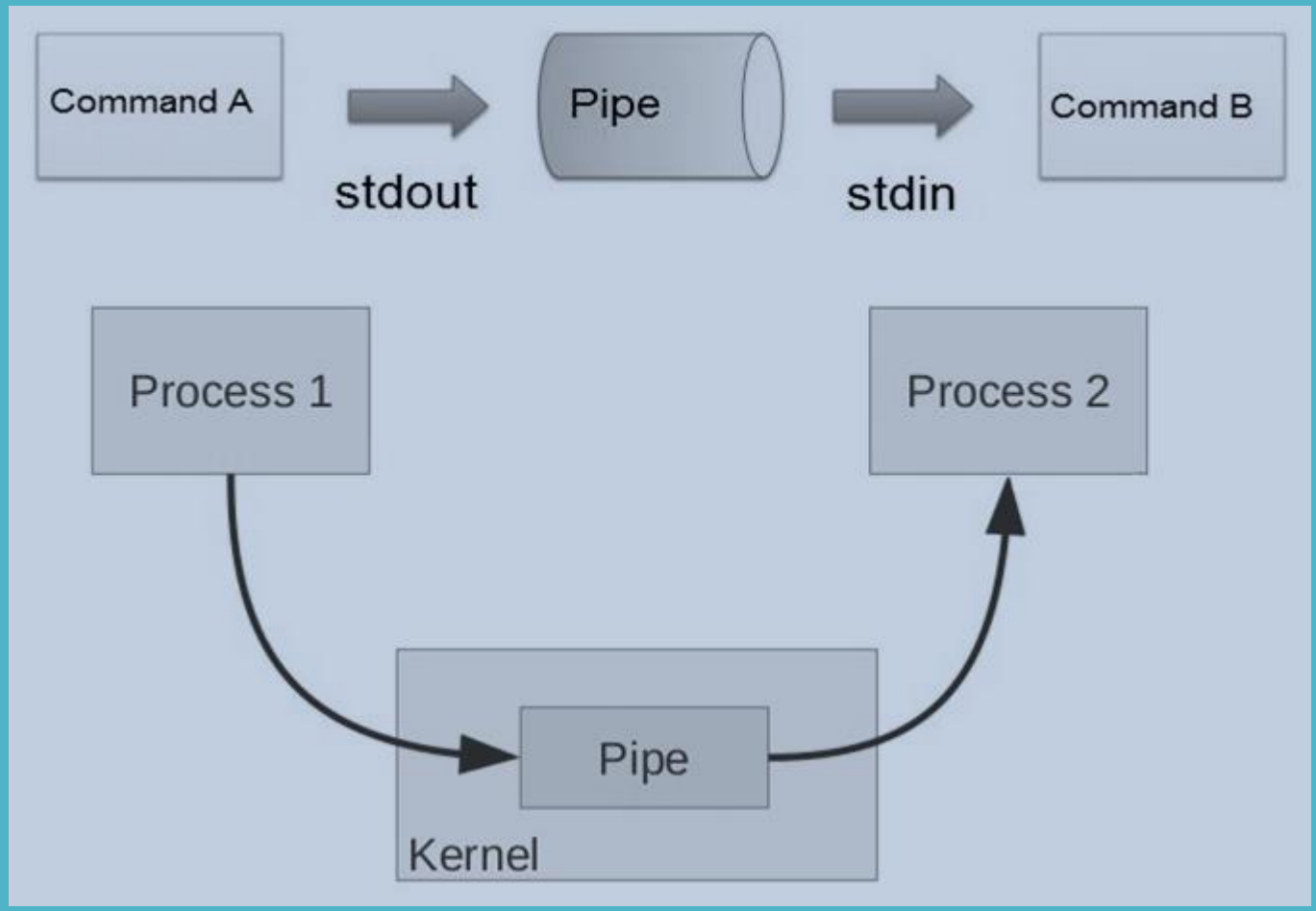
Παράδειγμα 2 → Εκτύπωση περιεχομένων καταλόγου

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

```
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
```

Το πρόγραμμα εκτυπώνει τα ονόματα των περιεχομένων του καταλόγου (αρχεία και κατάλογοι), ένα όνομα σε κάθε γραμμή συμπεριλαμβανομένων και των κρυφών αντικειμένων.

Αγωγοί


Command A
stdout
Pipe
stdin
Command B
Process 1
Process 2
Pipe
Kernel

Αγωγοί

Οι αγωγοί (pipes) αποτελούν τον έναν από τους αρκετούς τρόπους επικοινωνίας μεταξύ διεργασιών (οι άλλες λύσεις είναι η χρήση κοινόχρηστης μνήμης, οι ουρές μηνυμάτων και οι υποδοχείς).

Αποτελούν κανάλι επικοινωνίας που συνδέει την έξοδο της μιας διεργασίας με την είσοδο της άλλης.

Η επικοινωνία είναι απλής κατεύθυνσης (unnamed pipes) ή (σπάνια) διπλής κατεύθυνσης (named pipes) – υποστηρίζεται μόνο από λίγα συστήματα, σχεδόν όλα προσφέρουν απλής κατεύθυνσης.

Στο ένα άκρο υπάρχει η διεργασία συγγραφέας που στέλνει την έξοδό της στον αγωγό.

Στο άλλο άκρο υπάρχει η διεργασία αναγνώστης που διαβάζει την είσοδό της από τον αγωγό.

Οι αγωγοί διαθέτουν το δικό τους σύστημα αρχείων (pipefs) που δεν προσαρτάται κάτω από τον ριζικό κατάλογο αλλά παράπλευρα με αυτό.

Δεν είναι δυνατή η άμεση εξέτασή τους από το χρήστη όπως συμβαίνει με τα συστήματα αρχείων.

Το σύστημα pipefs αποθηκεύεται ως ένα IMFS (In-Memory File System) το οποίο ως δανείζεται τους πόρους του για την αποθήκευση αρχείων και καταλόγων όχι από το δίσκο αλλά από τη μνήμη.

Η χωρητικότητα ενός αγωγού είναι περιορισμένη (65536 bytes από τον πυρήνα 2.6.11 και μετά).

Αγωγοί

Απαιτείται συγχρονισμός διεργασιών → αν ο αναγνώστης προσπεράσει το συγγραφέα, αναστέλλει τη λειτουργία του μέχρι να υπάρξουν περισσότερα δεδομένα, ενώ το ίδιο ισχύει και όταν ο συγγραφέας προσπεράσει τον αναγνώστη.

Ο αναγνώστης αναστέλλει τη λειτουργία του όταν ο αγωγός είναι άδειος, αναμένοντας νέα δεδομένα.

Ο συγγραφέας αναστέλλει τη λειτουργία του όταν ο αγωγός είναι πλήρης, αναμένοντας την ανάγνωση δεδομένων από τον αναγνώστη.

Όταν κλείσουν όλοι οι περιγραφείς αρχείων που σχετίζονται με το άκρο του συγγραφέα, η ανάγνωση δεδομένων από τον αγωγό επιστρέφει τον κωδικό EOF.

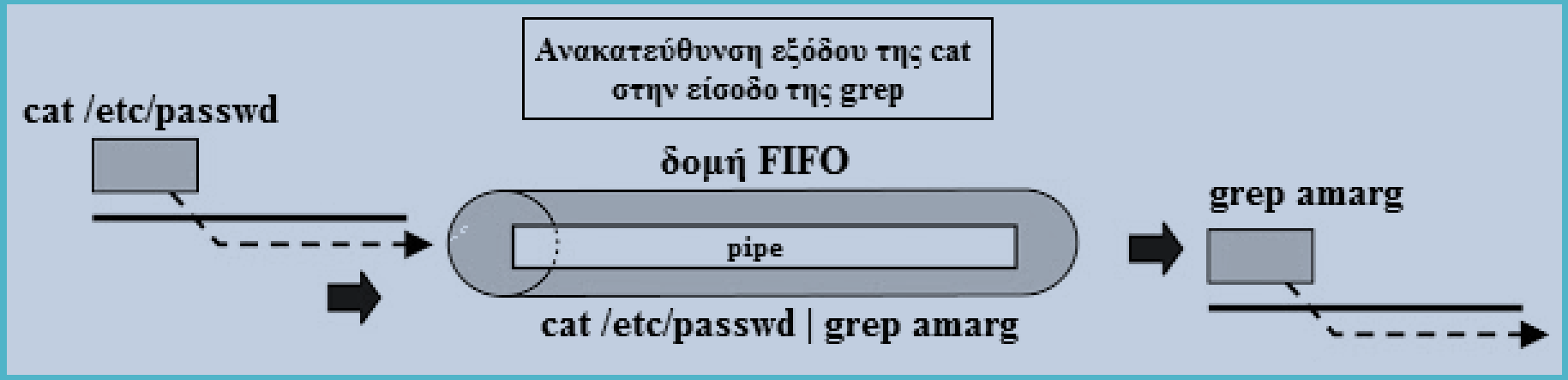
Η βέλτιστη απόδοση επιτυγχάνεται όταν ο αναγνώστης διαβάζει δεδομένα με την ίδια ταχύτητα με την οποία ο συγγραφέας γράφει δεδομένα.

Οι αγωγοί υλοποιούνται τόσο σε επίπεδο φλοιού όσο και σε επίπεδο πυρήνα αν και οι λειτουργίες που προσφέρει ο πυρήνας είναι πολύ πιο γενικές σε σχέση με αυτές που προσφέρει ο φλοιός.

Οι αγωγοί δεν είναι κανονικά αρχεία (δεν εμφανίζονται στο σύστημα αρχείων αν και διαθέτουν i-node) και δεν υπάρχει σύνδεσμος προς αυτούς (είναι εσωτερικά αντικείμενα του πυρήνα).

Αγωγοί

Παράδειγμα χρήσης αγωγού ανάμεσα στις εντολές cat και grep



Η ανακατεύθυνση της εξόδου της cat στην είσοδο της grep γίνεται με δύο τρόπους:

Χρησιμοποιώντας ενδιάμεσο αρχείο στο οποίο η cat αποθηκεύει την έξοδό της και από το οποίο η grep διαβάζει την είσοδό της, δηλαδή ως

```
cat /etc/passwd > tempFile
grep amarg < tempFile
```

ή χρησιμοποιώντας αγωγό ως

```
cat /etc/passwd | grep amarg < tempFile
```

Στη δεύτερη περίπτωση δεν απαιτείται η χρήση του ενδιάμεσου αρχείου tempFile.

Αγωγοί

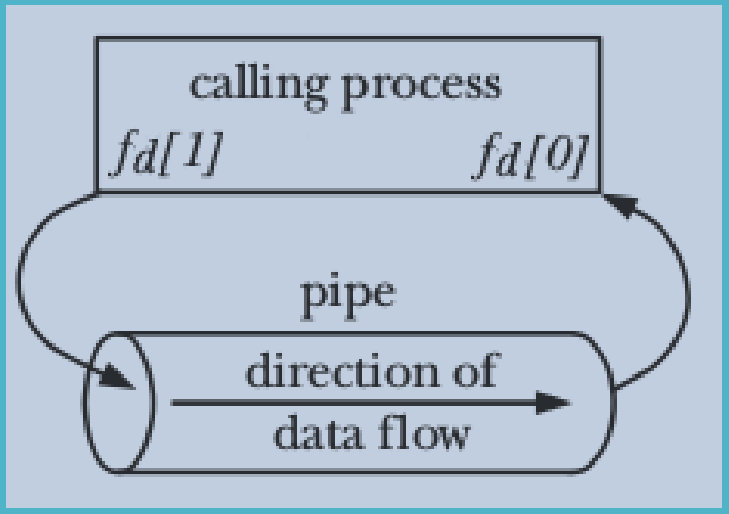
Η δημιουργία ενός αγωγού στηρίζεται στη χρήση της συνάρτησης pipe (unistd.h)

int pipe (int fd [2]);

Αυτή η συνάρτηση δημιουργεί έναν αγωγό και επιστρέφει στο όρισμα fd (που είναι πίνακας δύο ακεραίων) τους δύο περιγραφείς αρχείων που σχετίζονται με τα δύο άκρα του αγωγού.

Ο περιγραφέας αρχείου fd [0] σχετίζεται με το άκρο του αναγνώστη (read).

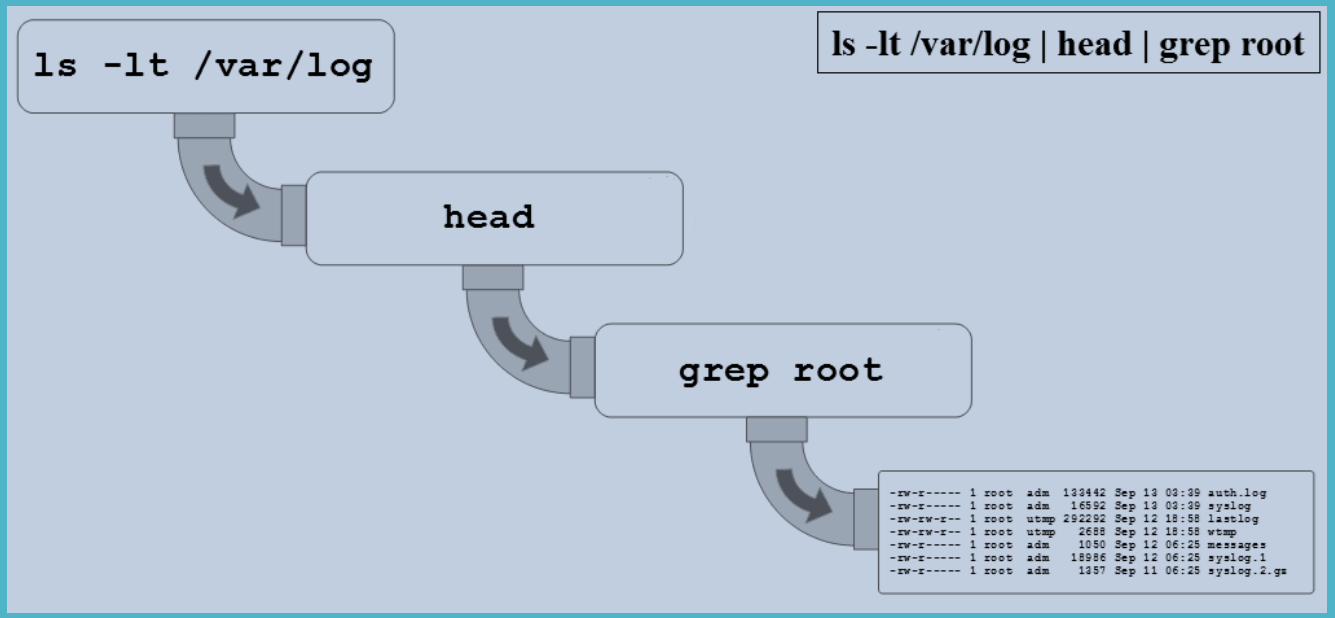
Ο περιγραφέας αρχείου fd [1] σχετίζεται με το άκρο του συγγραφέα (write).



Οι περιγραφείς αρχείου που δεν χρησιμοποιούνται μπορούν να κλείσουν με τη συνάρτηση close.

Αγωγοί

ΠΛΕΟΝΕΚΤΗΜΑ → οι διεργασίες εκτελούνται ταυτόχρονα και όχι ακολουθιακά η μία μετά την άλλη όπως στην περίπτωση χρήσης προσωρινού αρχείου. Όλα τα δεδομένα διακινούνται μέσω του πυρήνα.

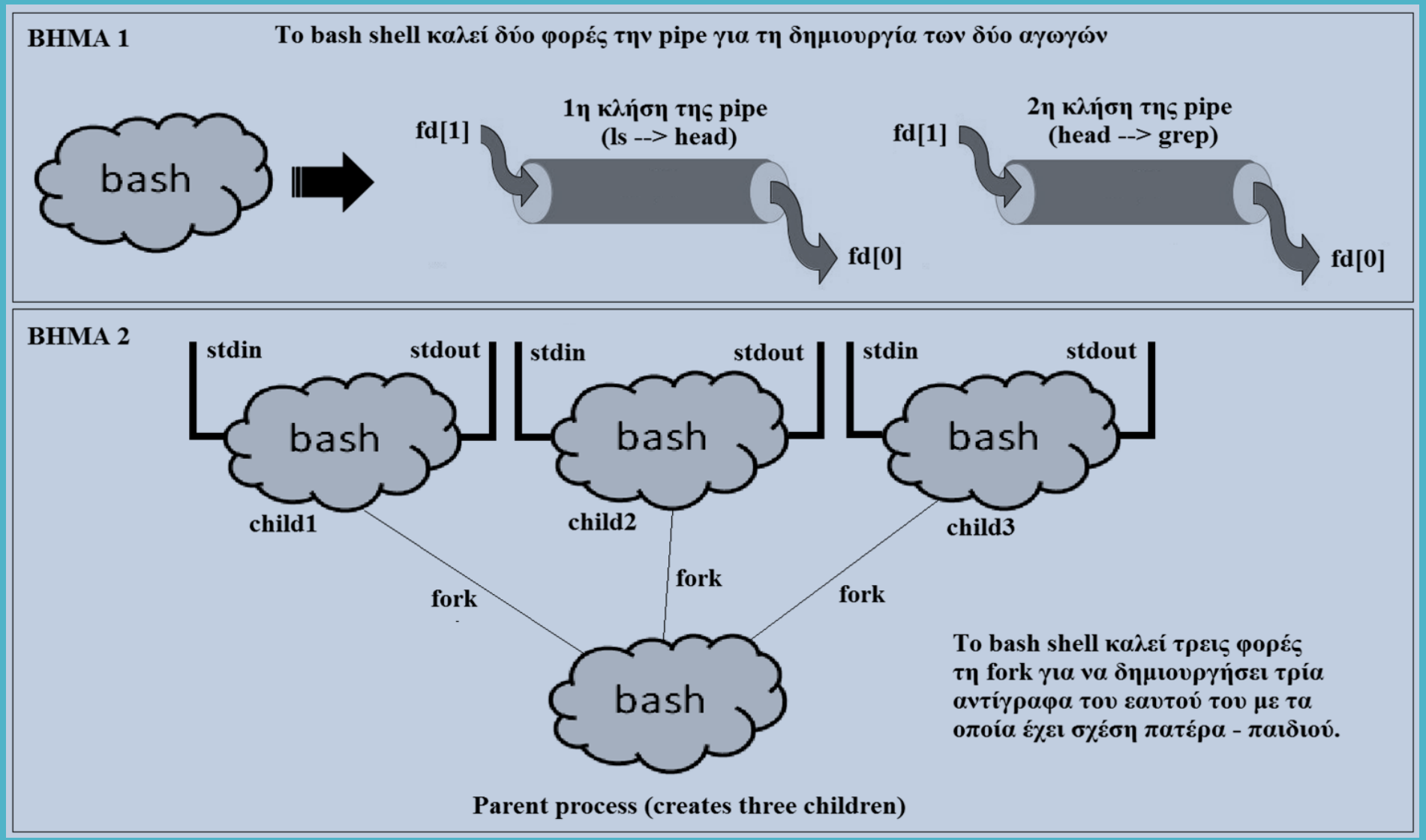


ΜΕΙΟΝΕΚΤΗΜΑ → οι διεργασίες πρέπει να σχετίζονται με σχέση γονέα – παιδιού ή να αποτελούν αμφότερες παιδιά του ιδίου γονέα. Αυτό είναι περιοριστικό για εφαρμογές client / server.

ΜΕΙΟΝΕΚΤΗΜΑ → Σε περιπτώσεις εγγραφών πολύ μεγάλου μεγέθους, δεν διασφαλίζεται η ατομικότητα και εάν υπάρχουν πολλοί συγγραφείς ενδέχεται να επέλθει σύγχυση.

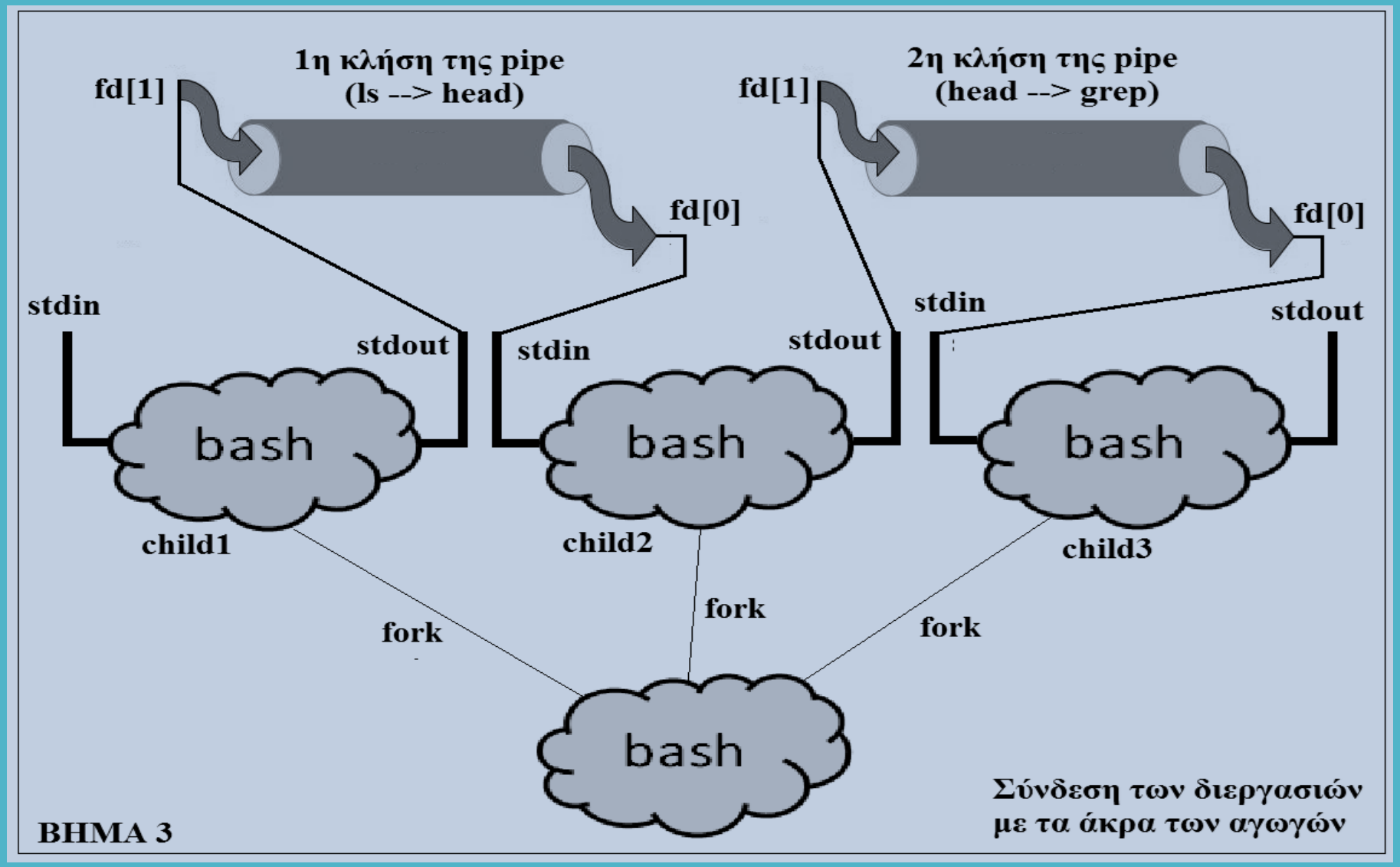
Αγωγοί

Η διαδικασία δημιουργίας και χρήσης αγωγών από το φλοιό, αποτελείται από τέσσερα βήματα:



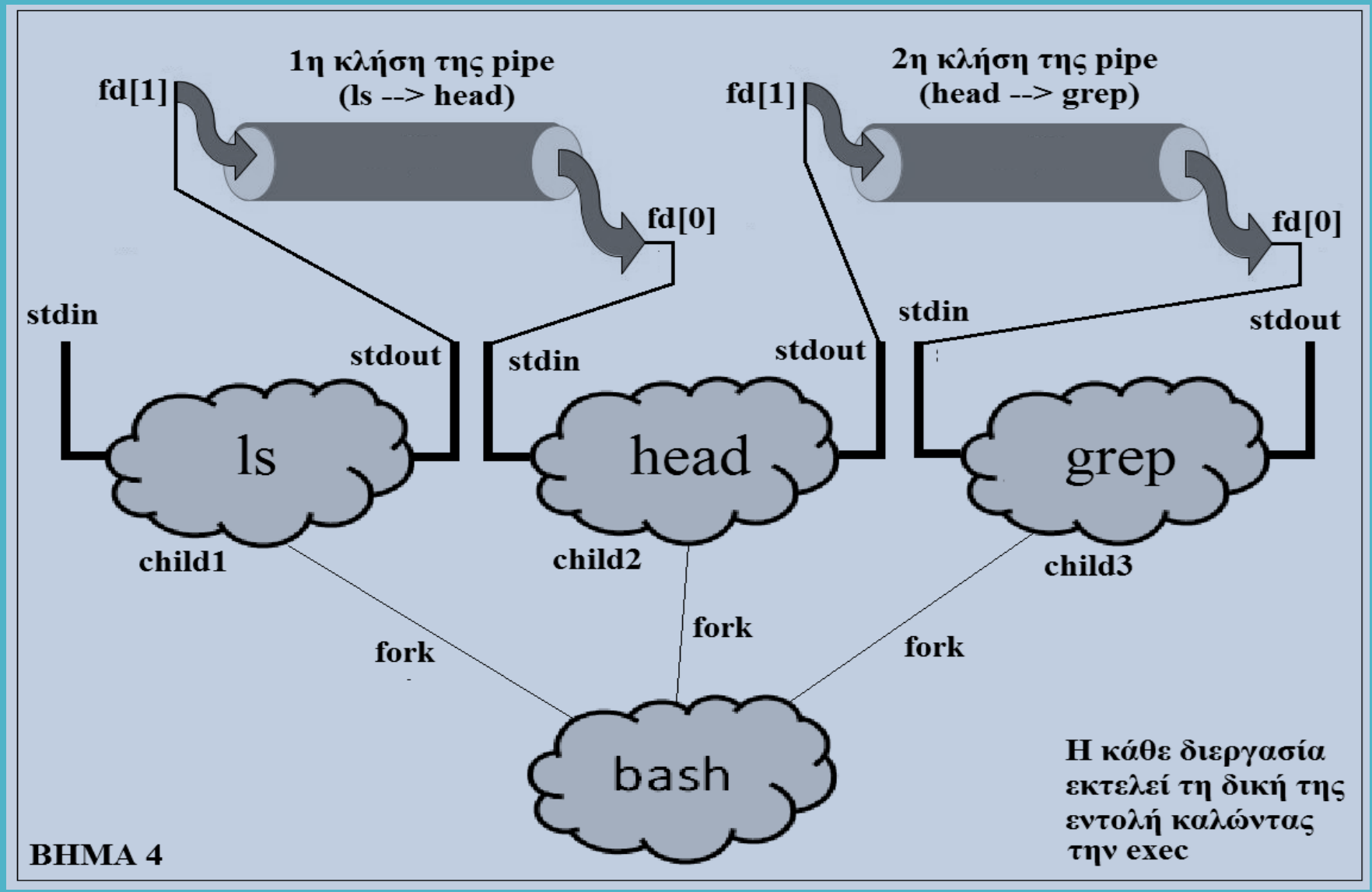
Αγωγοί

Η διαδικασία δημιουργίας και χρήσης αγωγών από το φλοιό, αποτελείται από τέσσερα βήματα:



Αγωγοί

Η διαδικασία δημιουργίας και χρήσης αγωγών από το φλοιό, αποτελείται από τέσσερα βήματα:

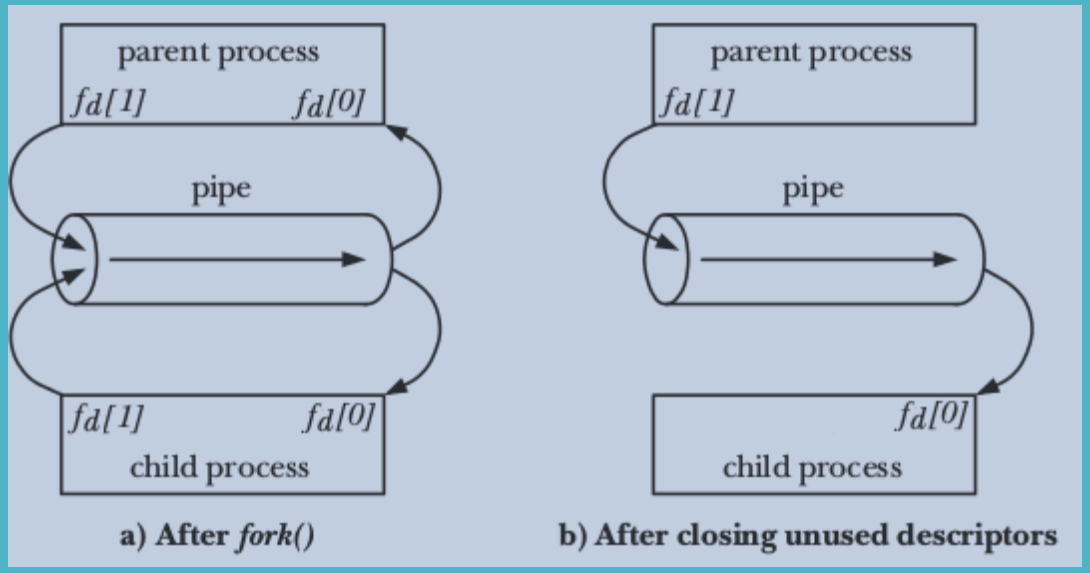


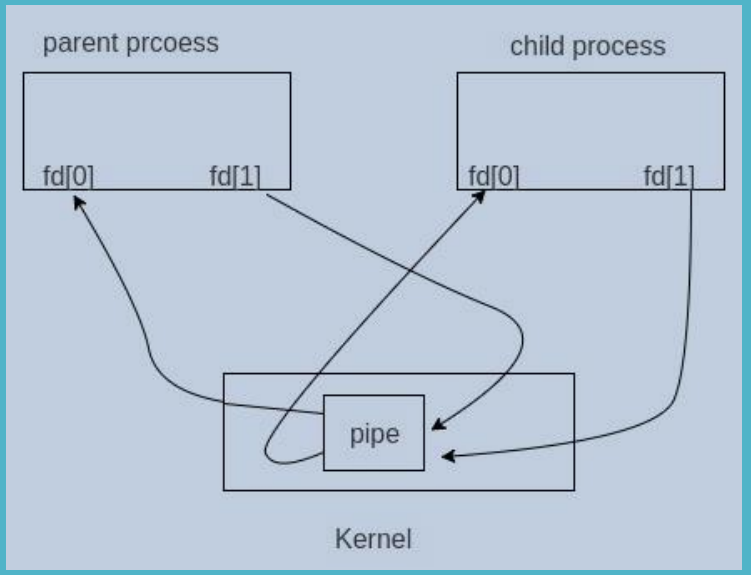
Αγωγοί

Στο παραπάνω παράδειγμα η επικοινωνία πραγματοποιείται ανάμεσα στα παιδιά της γονικής διεργασίας που βέβαια είναι ο φλοιός bash.

Η εντολή fork καλείται αυτόματα από το φλοιό χωρίς την παρέμβαση του χρήστη.

Ωστόσο είναι δυνατή η κλίση της fork μέρα από το πρόγραμμα και η δημιουργία μιας τοπολογίας πατέρα – παιδιού έτσι ώστε η επικοινωνία να μην πραγματοποιηθεί ανάμεσα στα παιδιά της γονικής διεργασίας αλλά ανάμεσα στη γονική και στη θυγατρική διεργασία.





Οι file descriptors κληροδοτούνται στη θυγατρική διεργασία και κατά συνέπεια δείχνουν στα ίδια άκρα.

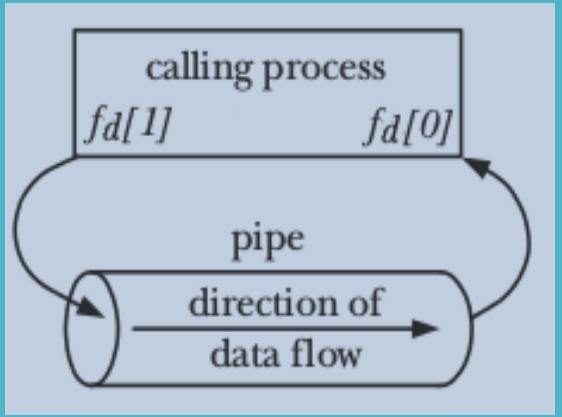
Αγωγοί

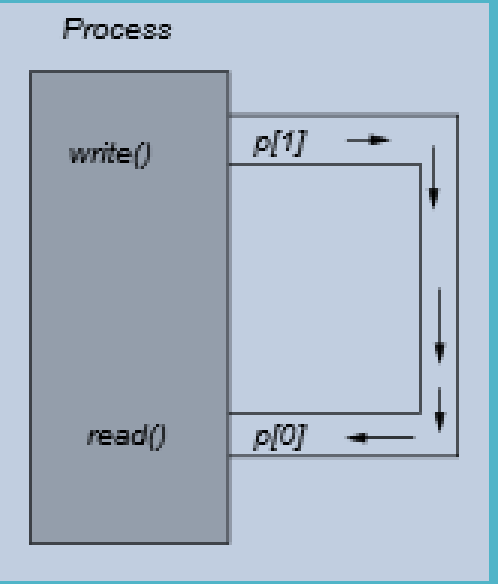
Παράδειγμα 1 → Δημιουργία και χρήση αγωγού από μία διεργασία

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

#define MSGSIZE 16


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

int main() {
    char inbuf[MSGSIZE];
    int p[2], i;

    if (pipe(p) < 0)
        exit(1);

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

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

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

Σε αυτό το πολύ απλό παράδειγμα υπάρχει μία και μοναδική διεργασία η οποία μιλά στον εαυτό της, δηλαδή γράφει δεδομένα στο σωλήνα τα οποία στη συνέχεια διαβάζει στην είσοδό της. Δεν υπάρχει επικοινωνία μεταξύ διαφορετικών διεργασιών.



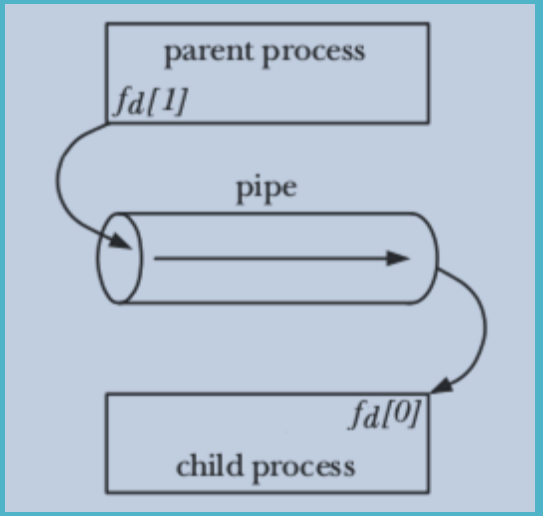


Αγωγοί

Παράδειγμα 2 → Δημιουργία και χρήση αγωγού από τοπολογία πατέρα - παιδιού

```c
int main() {
  int fd[2], errno;
  char buf;
  const char* msg = "Hello world .. Have a nice day!!\n";
  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); }
  // child process
  if (cpid==0) {
    // child only reads, so close write end
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0)
           write(STDOUT_FILENO, &buf, sizeof(buf));
    close(fd[0]);
    _exit(0); }
  // parent process
  else {
    // parent only writes, so close read end
    close(fd[0]);
    write(fd[1], msg, strlen(msg));
    close(fd[1]);
    wait(NULL);
    exit(0); }
  return 0; }
```

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




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

Η διεργασία πατέρας γράφει στον αγωγό ολόκληρο το μήνυμα σε ένα και μόνο βήμα, ενώ η διεργασία παιδί μέσω μίας επαναληπτικής διαδικασίας διαβάζει έναν έναν τους χαρακτήρες του μηνύματος και τους εκτυπώνει στην οθόνη.

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού από τοπολογία πατέρα - παιδιού

```c
int main() {
  int fd[2], errno, count;
  char buffer[1024];
  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); }
  // child process
  if (cpid==0) {
    close(fd[0]);
    printf("Child Process --> Enter an input: ");
    fgets(buffer, sizeof(buffer), stdin);
    printf("Child process: User Inpus is %s", buffer);
    count = write(fd[1], buffer, strlen(buffer)+1);
    exit(0); }
  // parent process
  else {
    close(fd[1]);
    count = read(fd[0], buffer, sizeof(buffer));
    printf("Parent process: message is %s", buffer);
    wait(NULL);   }
  return 0; }
```

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

```
amarg@amarg-vbox:~$ ./pipeEx3
Child Process --> Enter an input: Hello World !!
Child process: User Inpus is Hello World !!
Parent process: message is Hello World !!
```

Η διεργασία παιδί ζητά από το χρήστη να καταχωρήσει μία συμβολοσειρά την οποία στη συνέχεια γράφει στον αγωγό. Αυτή η συμβολοσειρά διαβάζεται στο άλλο άκρο από τη διεργασία παιδί και στη συνέχεια εκτυπώνεται στην οθόνη.

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

Ο κώδικας που ακολουθεί δέχεται ως όρισμα ένα όνομα καταλόγου dirname και μία συμβολοσειρά προς αναζήτηση string και στη συνέχεια εκτελεί τη διασωλήνωση

ls –l dirname | grep string

για να εκτυπώσει τα περιεχόμενα του καταλόγου dirname που περιέχουν στο όνομά τους τη συμβολοσειρά string.

Αυτό το παράδειγμα λειτουργεί με τον ίδιο τρόπο με τον οποίο το bash shell υλοποιεί διασωληνώσεις σαν την παραπάνω τις οποίες εκτελεί ο χρήστης από τη γραμμή εντολών.

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

```
int main(int argc, char *argv[]) {
    int pid, pid_ls, pid_grep, fd[2];
    if (argc != 3) {
        printf("Usage: pipeEx4 dirname string\n");
        return (-1); }
    printf("Parent process: Grepping %s for %s\n", argv[1], argv[2]);
    if (pipe(fd)==-1) {
        printf("Parent process: Failed to create pipe\n");
        return (-2); }
    // 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");
         // 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]);
         // 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); }
```

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

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

```
// 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); }
// 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; }
```

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

Το αποτέλεσμα της εκτέλεσης του παραπάνω κώδικα που ακολουθεί στη συνέχεια

```
amarg@amarg-vbox:~$ ./pipeEx4 shared prog
Parent process: Grepping shared for prog
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
-rwxrwxrwx  1 root  root   19624 Sep 23 17:21 prog
-rwxrwxrwx  1 root  root     407 Sep 23 17:21 prog.c
-rwxrwxrwx  1 root  root    2432 Sep 23 17:21 prog.o
--------------------
parent: Children has finished execution, parent is done
```

είναι ακριβώς το ίδιο με εκείνο που παίρνουμε εάν εκτελέσουμε στη γραμμή εντολών του λειτουργικού συστήματος την εντολή

ls –l shared | grep prog

```
amarg@amarg-vbox:~$ ls -l shared | grep prog
-rwxrwxrwx 1 root root  19624 Σεπ  23 17:21 prog
-rwxrwxrwx 1 root root    407 Σεπ  23 17:21 prog.c
-rwxrwxrwx 1 root root   2432 Σεπ  23 17:21 prog.o
```

Αυτός λοιπόν είναι και ο τρόπος με τον οποίο υλοποιούνται οι διασωληνώσεις από το φλοιό του λειτουργικού συστήματος.

Αγωγοί

Αναπαραγωγή περιγραφέων αρχείου

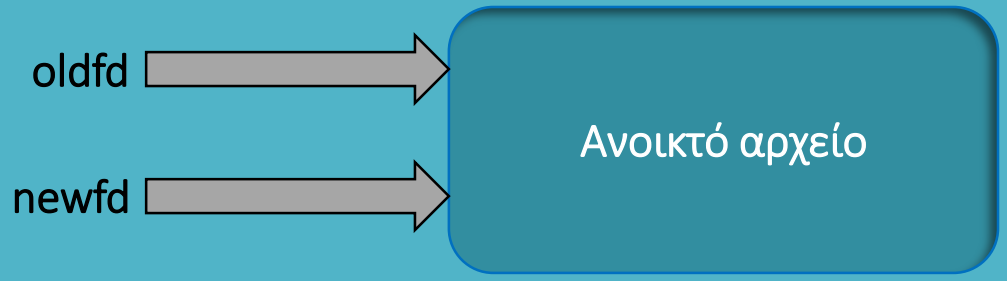
Πραγματοποιείται από τις συναρτήσεις του αρχείου unistd.h

```
int dup (int oldfd);
Int dup2 (int oldfd, int newfd);
```

οι οποίες δημιουργούν νέους περιγραφείς αρχείου που δείχνουν σε ένα ανοιχτό αρχείο. Είναι ιδιαίτερα χρήσιμες όταν ο χρήστης θέλει να ανακατευθύνει τα stdin, stdout και stderr.

Η συνάρτηση dup επιστρέφει έναν περιγραφέα αρχείου που έχει όσο το δυνατό μικρότερη τιμή, ενώ εάν θέλουμε να δημιουργήσουμε έναν περιγραφέα αρχείου με συγκεκριμένη τιμή, χρησιμοποιούμε την dup2. Σε αμφότερες τις περιπτώσεις οι περιγραφείς oldfd και newfd δείχνουν στο ίδιο αρχείο.

Εάν ο περιγραφέας newfd δείχνει σε ανοικτό αρχείο, αυτό κλείνει προκειμένου ο νέος περιγραφέας να δείξει στο ίδιο αρχείο με τον περιγραφέα oldfd.



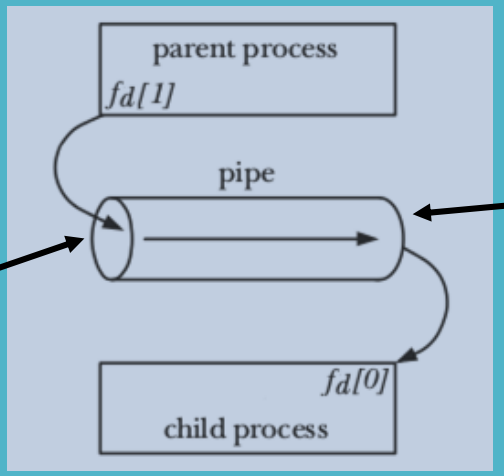
Η αναπαραγωγή ενός περιγραφέα αρχείου ΔΕΝ ΣΥΜΠΙΠΤΕΙ με το άνοιγμα του αρχείου δύο φορές.

Αγωγοί

Το βασικό χαρακτηριστικό των αγωγών που παρουσιάστηκαν μέχρι τώρα είναι πως δεν σχετίζονται με αρχεία που αποθηκεύονται στο σκληρό δίσκο. Περιγράφονται από ειδικές δομές που αποθηκεύονται προσωρινά στη μνήμη και διαγράφονται με την ολοκλήρωση της λειτουργίας.

Το μειονέκτημά τους είναι πως χρησιμοποιούνται μόνο στην περίπτωση διεργασιών που σχετίζονται μεταξύ τους με σχέση γονέα - παιδιού. Η διαδικασία έχει ως εξής:

- Η γονική διεργασία δημιουργεί τον αγωγό με την pipe.
- Η γονική διεργασία (συγγραφέας) δημιουργεί τη θυγατρική διεργασία (αναγνώστης) με τη fork.
- Η θυγατρική διεργασία κλείνει το άκρο εγγραφής του αγωγού αφού θα πραγματοποιήσει ανάγνωση.
- Η θυγατρική διεργασία εκτελεί το θυγατρικό πρόγραμμα με κάποια έκδοση της exec.
- Η γονική διεργασία κλείνει το άκρο ανάγνωσης του αγωγού αφού θα πραγματοποιήσει εγγραφή.
- Η γονική διεργασία γράφει στον αγωγό.
- Η θυγατρική διεργασία διαβάζει από τον αγωγό.



Η θυγατρική διεργασία κλείνει το άκρο εγγραφής

Η γονική διεργασία κλείνει το άκρο ανάγνωσης.

Αγωγοί

Η επικοινωνία διεργασιών που δεν σχετίζονται με σχέση γονέα – παιδιού αλλά μπορεί να είναι οποιεσδήποτε, γίνεται με τη βοήθεια των επώνυμων αγωγών που είναι γνωστοί ως FIFO.

FIFO (First In First Out)

Υφίστανται ως αρχεία στο σκληρό δίσκο και επιτρέπουν την πολλαπλή εγγραφή δεδομένων από πολλούς συγγραφείς ταυτόχρονα τα οποία διαβάζονται από έναν αναγνώστη και μέσω μιας αρχιτεκτονικής που μοιάζει με την αρχιτεκτονική client / server.

Με εξαίρεση τον διαφορετικό τρόπο δημιουργίας τους, η λειτουργία τους είναι παρόμοια με αυτή των απλών (ανώνυμων) αγωγών.

Οι επώνυμοι αγωγοί ανοίγουν με την open η οποία χρησιμοποιεί στο άκρο του αναγνώστη το flag O_RDONLY και στο άκρο του συγγραφέα το flag O_WRONLY.

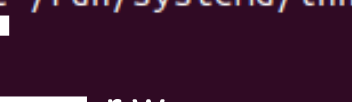
Τα αρχεία του σκληρού δίσκου που αναφέρονται σε επώνυμους αγωγούς χαρακτηρίζονται από την εμφάνιση του γράμματος p ως πρώτο γράμμα της μάσκας δικαιωμάτων που εκτυπώνει η ls –l.

```
prw------- 1 root root    0 Οκτ    3 10:59 1.ref
prw------- 1 root root    0 Οκτ    3 10:59 2.ref
```

Αγωγοί

Η εκτέλεση του προγράμματος filestat με όρισμα ένα αρχείο επώνυμου αγωγού μας δίνει

```
amarg@amarg-vbox:~/shared/Lab5$ ./filestat /run/systemd/inhibit/1.ref
File type:                FIFO/pipe
I-node number:            412
Mode:                     10600 (octal)
Link count:               1
Ownership:                UID=0   GID=0
Preferred I/O block size: 4096 bytes
File size:                0 bytes
Blocks allocated:         0
Last status change:       Sat Oct  3 10:59:23 2020
Last file access:         Sat Oct  3 10:59:23 2020
Last file modification:   Sat Oct  3 10:59:23 2020
amarg@amarg-vbox:~/shared/Lab5$
```



Η δημιουργία επώνυμων αγωγών γίνεται με τις συναρτήσεις mkfifo και mknod – από αυτές χρησιμοποιείται συνήθως η mkfifo που χαρακτηρίζεται από φορητότητα και όχι η mknod η οποία παρεμπιπτόντως εκτός από αρχεία FIFO μπορεί να κατασκευάσει και άλλους τύπους αρχείων όπως block & character device files ενώ πριν την υλοποίηση της εντολής mkdir χρησιμοποιούνταν και για τη δημιουργία καταλόγων. Η συνάρτηση mkfifo ορίζεται ως

int mkfifo (const char *pathname, mode_t mode);

Αγωγοί

Παράδειγμα δημιουργίας επώνυμου αγωγού με τη συνάρτηση mkfifo

```c
// C program to implement one side of FIFO ...
// this side writes first, then reads

#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";
    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);
    char arr1[80], arr2[80];
    while (1) {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);
        // Take an input arr2ing from user.
        // 80 is maximum length
        fgets(arr2, 80, stdin);
        // Write the input arr2ing on FIFO and close it
        write(fd, arr2, strlen(arr2)+1);
        close(fd);
        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);
        // Read from FIFO
        read(fd, arr1, sizeof(arr1));
        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd); }
    return 0; }
```

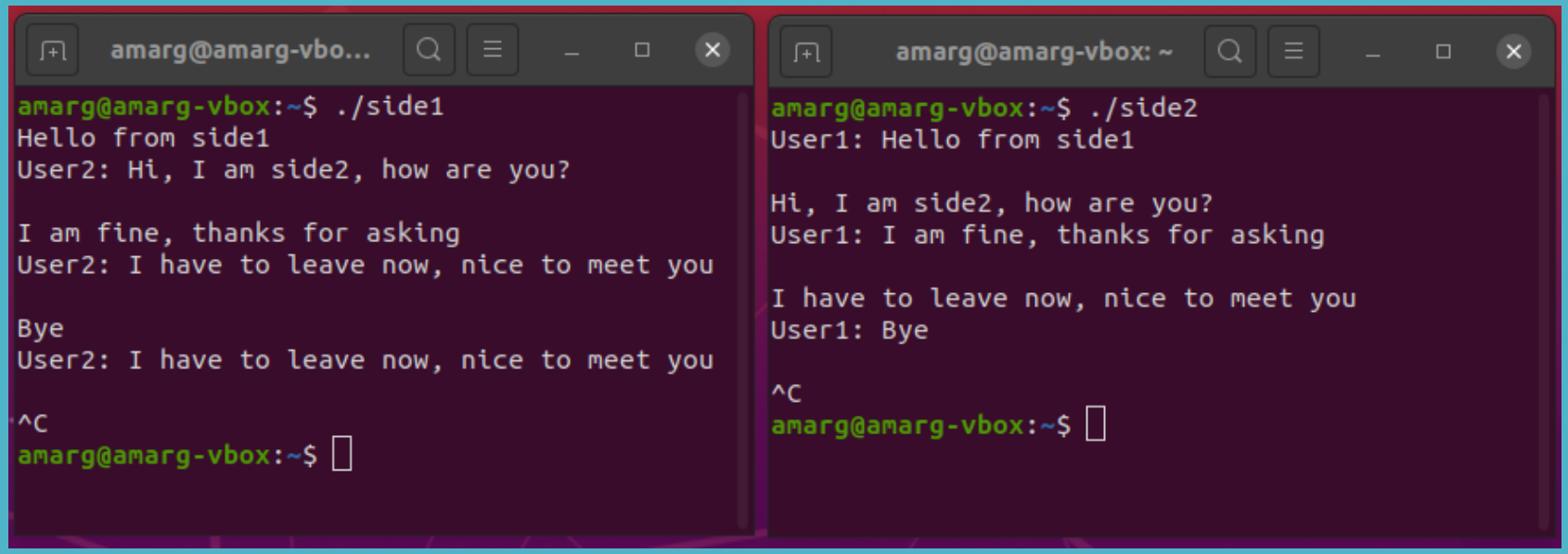
```c
// C program to implement one side of FIFO ...
// this side reads first, then writes

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

int main() {
    int fd1;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";
    // Creating the named file(FIFO)
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1) {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);

        // Now open in write mode and write
        // string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1); }
    return 0; }
```

Αγωγοί

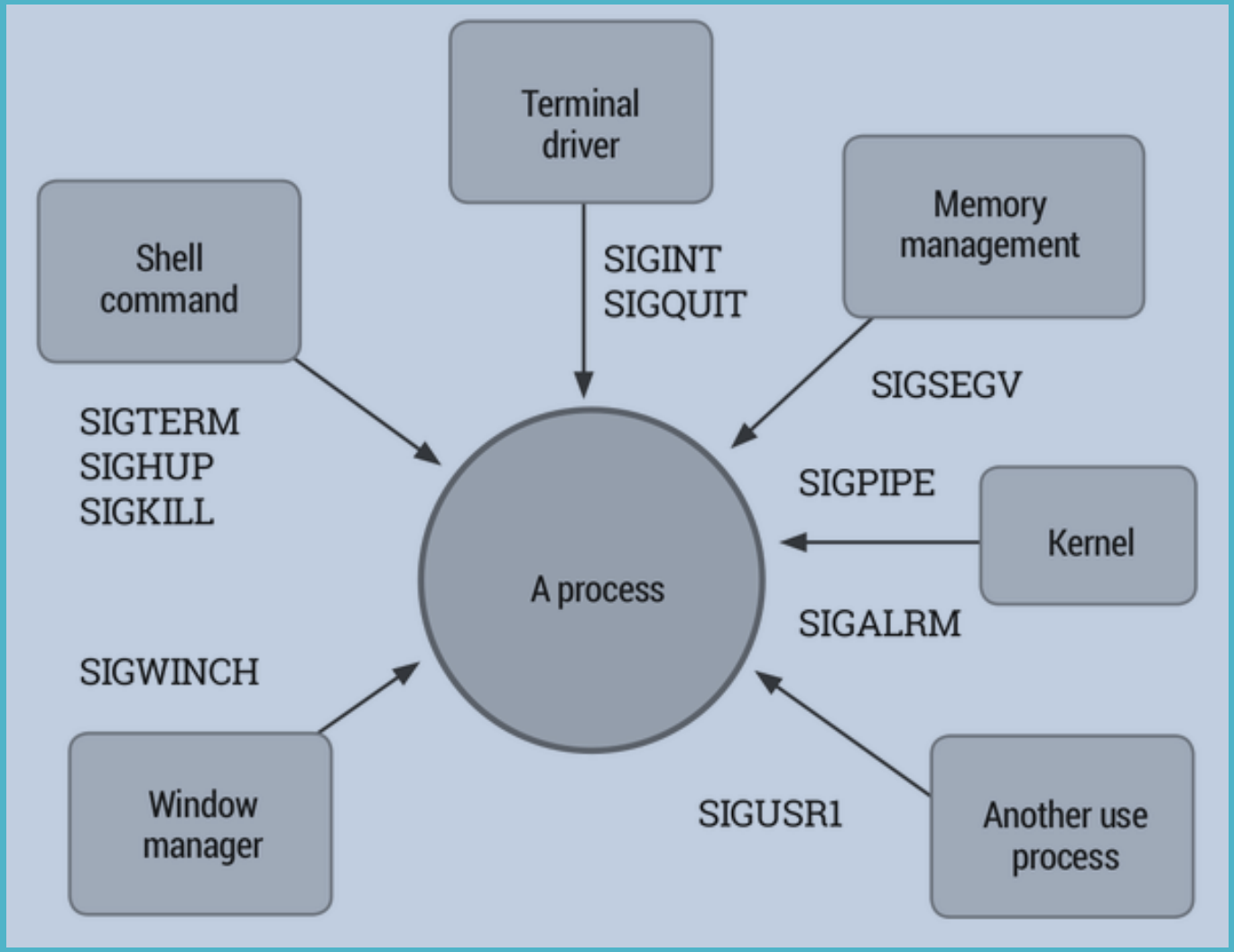
Οι επώνυμοι αγωγοί περιγράφονται από πραγματικά αρχεία του συστήματος

```
amarg@amarg-vbox:~$ ls -l /tmp
total 40
-rw------- 1 amarg amarg    0 Οκτ   3 10:59 config-err-ljzuZQ
prw-rw-r-- 1 amarg amarg    0 Οκτ   3 15:38 myfifo
drwx------ 2 amarg amarg 4096 Οκτ   3 10:59 ssh-KFr1WVeZZu5b
```

Οι δύο εφαρμογές είναι ανεξάρτητες η μία από την άλλη και εκτελούνται παράλληλα σε δύο διαφορετικά τερματικά ανταλλάσσοντας μηνύματα τα οποία διακινούνται μέσω του αγωγού.



Σήματα


Terminal driver
SIGINT
SIGQUIT
Shell command
SIGTERM
SIGHUP
SIGKILL
Memory management
SIGSEGV
SIGPIPE
Kernel
A process
SIGALRM
SIGWINCH
Window manager
SIGUSR1
Another use process

Σήματα

Τα σήματα (signals) αποτελούν την απλούστερη μορφή επικοινωνίας ανάμεσα στις διεργασίες.

Επιτρέπουν τη διακοπή μιας διεργασίας από τον πυρήνα ή μία άλλη διεργασία με στόχο τη διαχείριση κάποιου συμβάντος. Μετά το χειρισμό του συμβάντος η διεργασία συνεχίζει τη λειτουργία της.

Τα σήματα είναι ασύγχρονα. Μπορούν να εμφανιστούν από τον οποιοδήποτε και ανά πάσα στιγμή.

Όταν μία διεργασία παραλάβει κάποιο σήμα, πραγματοποιεί κάποιο από τα ακόλουθα:

- Το αγνοεί.
- Ζητά από τον πυρήνα να συλλάβει (catch) το σήμα πριν επιτρέψει στη διεργασία να συνεχίσει.
- Αφήνει τον πυρήνα να εκτελέσει την προεπιλεγμένη απόκριση για το εν λόγω σήμα.

Ορισμός σήματος (signal.h)

Αρχικά ένα σήμα ορίζονταν από τη συνάρτηση signal (δεν χρησιμοποιείται σήμερα) που δηλώνεται ως

void * signal (int signum, void * handler);

όπου signum ένας αριθμός που ταυτοποιεί το σήμα και handler ένας δείκτης σε μία συνάρτηση χωρίς ορίσματα και επιστρεφόμενη τιμή η οποία εκτελείται από τον πυρήνα και η ολοκλήρωση της οποίας σηματοδοτεί τη συνέχιση της εκτέλεσης της διεργασίας από το σημείο που είχε σταματήσει.

Σήματα

Το σύνολο των διαθέσιμων σημάτων στο Linux εμφανίζεται με την εντολή kill –l η οποία επίσης χρησιμοποιείται και για την αποστολή τους.

```
amarg@amarg-vbox:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Τα σήματα με αριθμούς από 1 έως 31 είναι τα συνήθη σήματα του λειτουργικού συστήματος Linux.

Τα σήματα με αριθμούς από 32 έως 64 αποτελούν σήματα πραγματικού χρόνου και χρησιμοποιούνται από τα ομώνυμα συστήματα. Παρατηρήστε πως τα σήματα 32 και 33 ΔΕΝ εκτυπώνονται από την εντολή kill –l και ο λόγος για αυτό είναι πως χρησιμοποιούνται εσωτερικά από τη βιβλιοθήκη NPTL (Native POSIX Threads Library). Τα σήματα αυτά ορίζονται ως

SIGWAITING	32	Ignore All LWPs blocked
SIGLWP	33	Ignore Virtual Interprocessor Interrupt for Threads Library

και οι διαδικασίες που υλοποιούν ΔΕΝ υποστηρίζονται από το Linux.

Σήματα

Οι τιμές των σημάτων είναι δηλωμένες στο αρχείο signal.h ως ακέραιοι αριθμοί που ορίζονται με define
(οι παραπάνω τιμές αφορούν στο BSD UNIX ενώ για το Linux είναι αυτές που δίνει η kill -l)

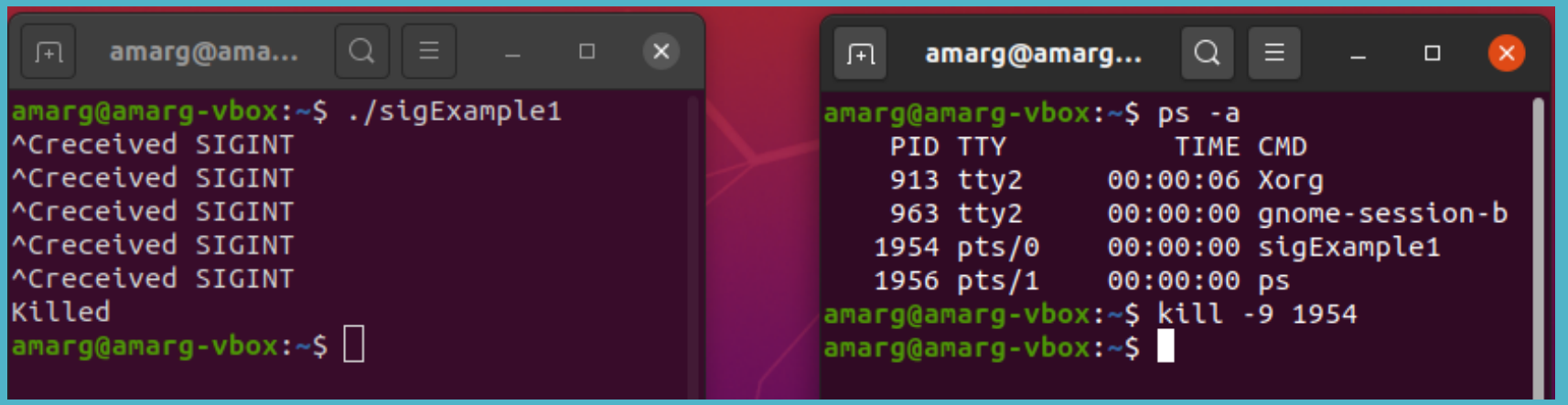
```
#define     SIGHUP      1     /* hangup */
#define     SIGINT      2     /* interrupt */
#define     SIGQUIT     3     /* quit */
#define     SIGILL      4     /* illegal instruction (not reset when caught) */
#define     SIGTRAP     5     /* trace trap (not reset when caught) */
#define     SIGABRT     6     /* abort() */
#define     SIGEMT      7     /* EMT instruction */
#define     SIGFPE      8     /* floating point exception */
#define     SIGKILL     9     /* kill (cannot be caught or ignored) */
#define     SIGBUS      10    /* bus error */
#define     SIGSEGV     11    /* segmentation violation */
#define     SIGSYS      12    /* bad argument to system call */
#define     SIGPIPE     13    /* write on a pipe with no one to read it */
#define     SIGALRM     14    /* alarm clock */
#define     SIGTERM     15    /* software termination signal from kill */
#define     SIGURG      16    /* urgent condition on IO channel */
#define     SIGSTOP     17    /* sendable stop signal not from tty */
#define     SIGTSTP     18    /* stop signal from tty */
#define     SIGCONT     19    /* continue a stopped process */
#define     SIGCHLD     20    /* to parent on child stop or exit */
#define     SIGTTIN     21    /* to readers pgrp upon background tty read */
#define     SIGTTOU     22    /* like TTIN for output if (tp->t_local&LTOSTOP) */
#define     SIGIO       23    /* input/output possible signal */
#define     SIGXCPU     24    /* exceeded CPU time limit */
#define     SIGXFSZ     25    /* exceeded file size limit */
#define     SIGVTALRM   26    /* virtual time alarm */
#define     SIGPROF     27    /* profiling time alarm */
#define     SIGWINCH    28    /* window size changes */
#define     SIGINFO     29    /* information request */
#define     SIGUSR1     30    /* user defined signal 1 */
#define     SIGUSR2     31    /* user defined signal 2 */
```

Σήματα

Παράδειγμα χρήσης της συνάρτησης signal – η διεργασία συλλαμβάνει το σήμα SIGINT

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

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

int main(void) {
  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
  while(1)
    sleep(1);
  return 0; }
```

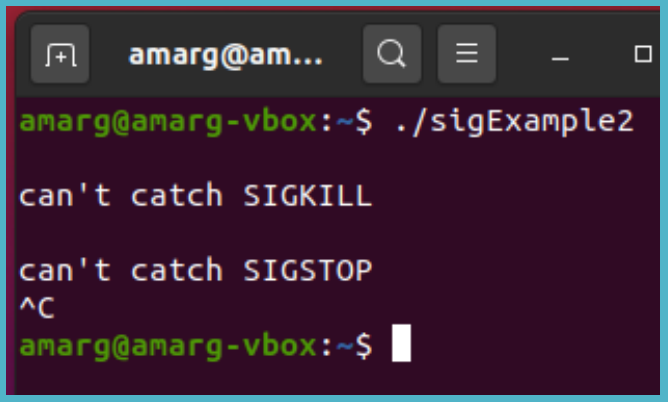
Η διεργασία ΔΕΝ τερματίζει με Ctrl – C αφού ο handler του σήματος SIGINT δεν καλεί την exit ή τη return. Για τον τερματισμό της διεργασίας ανοίγουμε ένα άλλο terminal, εκτελούμε την εντολή ps –a για να προσδιορίσουμε το pid της και την τερματίζουμε με την εντολή kill.



Σήματα

Τα σήματα SIGKILL και SIGSTOP δεν μπορούν να υποστούν χειρισμό από το χρήστη !!

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

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

int main(void) {
  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
  while(1)
    sleep(1);
  return 0; }
```



Ο χειρισμός του SIGINT αυτή τη φορά έγινε από τον πυρήνα και ως εκ τούτου η χρήση του συνδυασμού Ctrl-C οδήγησε στον τερματισμό της διεργασίας.

#define SIG_KERNEL_ONLY_MASK (rt_sigmask(SIGKILL) | rt_sigmask(SIGSTOP))

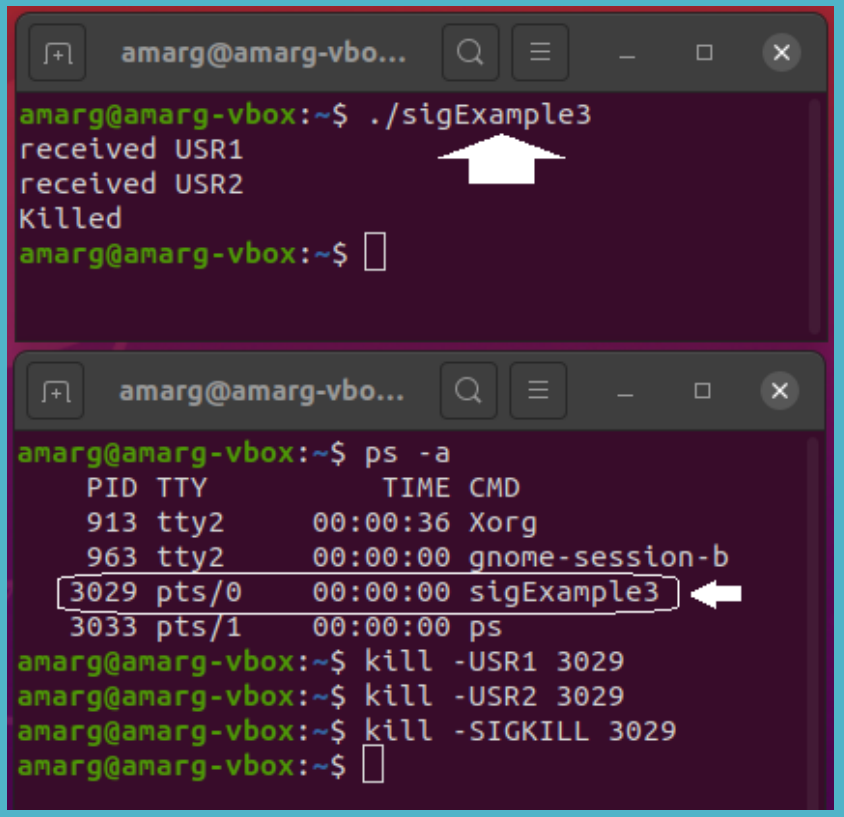
Ο λόγος για αυτό είναι τα εν λόγω σήματα επιτρέπουν τον δια της βίας τερματισμό (kill) μας διεργασίας η οποία έχει σταματήσει να αποκρίνεται. Εάν αναθέσουμε το χειρισμό αυτών των σημάτων στη διεργασία και η διεργασία σταματήσει να αποκρίνεται, δεν υπάρχει κανένας τρόπος για να τερματιστεί. Για το λόγο αυτό ο χειρισμός αυτών των σημάτων γίνεται από τον πυρήνα.

Σήματα

Τα σήματα γενικής χρήσεως SIGUSR1 και SIGUSR2

Το σύστημα προσφέρει τα σήματα SIGUSR1 και SIGUSR2 τα οποία είναι γενικής χρήσεως και μπορούν να χρησιμοποιηθούν σύμφωνα με τις ανάγκες του χρήστη, σε αντίθεση με όλα τα υπόλοιπα τα οποία είναι εντελώς συγκεκριμένα ως προς τον τρόπο λειτουργίας τους.

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

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

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



Αυτά τα σήματα – όπως και όλα τα υπόλοιπα – στέλνονται χρησιμοποιώντας την εντολή kill.

Σήματα

Πώς συμπεριφέρεται ο πυρήνας όταν παραλάβει ένα σήμα:

- Το αγνοεί.
- Αναστέλλει την εκτέλεση της διεργασίας.
- Τερματίζει απότομα τη διεργασία
- Τερματίζει απότομα τη διεργασία, αποτυπώνοντας τον πυρήνα (core dump).

Τι συμβαίνει όταν μία διεργασία λάβει ένα σήμα του οποίου εκτελεί τον χειριστή?

Σύμφωνα με τα όσα αναφέραμε, αναστέλλει την εκτέλεσή της και καλεί εκ νέου το χειριστή. Αυτό είναι κάτι που απαιτεί προσεκτικό προγραμματισμό και τη χρήση συναρτήσεων επανεισόδου.

Ωστόσο, δεν λειτουργεί όταν χρησιμοποιούνται τεχνικές κλειδώματος, π.χ. κλείδωμα αρχείου.

ΛΥΣΗ → Δεν στέλνουμε στη διεργασία σήματα των οποίων ήδη εκτελεί το χειριστή. Ωστόσο, αυτό οδηγεί σε δυσκολία στο χειρισμό εάν στέλνονται πολλά σήματα με μεγάλη ταχύτητα και ενδέχεται να οδηγήσει σε υπερβολική αύξηση του μεγέθους της στοίβας !!!

POSIX → Στα POSIX λειτουργικά, η διεργασία αναμένει για την μεταβίβαση του δεύτερου σήματος μέχρι να ολοκληρωθεί ο χειρισμός του πρώτου σήματος. Τα σήματα που αναμένουν τη μεταβίβασή τους χαρακτηρίζονται ως εκκρεμή σήματα (pending signal).

Εάν σταλεί ένα σήμα ενώ ήδη υπάρχει ένα εκκρεμές στιγμιότυπό του, τα δύο σήματα συγχωνεύονται.

Σήματα

Ανάκτηση των πληροφοριών της διεργασίας όσον αφορά στο χειρισμό των σημάτων της

Οι πληροφορίες αυτού του τύπου μπορούν να βρεθούν στα πεδία SigBlk, SigIgn και SigCgt οι τιμές των οποίων αποθηκεύονται στο αρχείο /proc/[pid]/status όπου pid το process id της διεργασίας.

```
# cat /proc/1/status
...
SigBlk: 0000000000000000
SigIgn: fffffffe57f0d8fc
SigCgt: 0000000280b2603
...
```

SigBlk → ομάδα μπλοκαρισμένων σημάτων
SigIgn → ομάδα σημάτων που αγνοούνται
SigCgt → ομάδα σημάτων που έχουν συλληφθεί

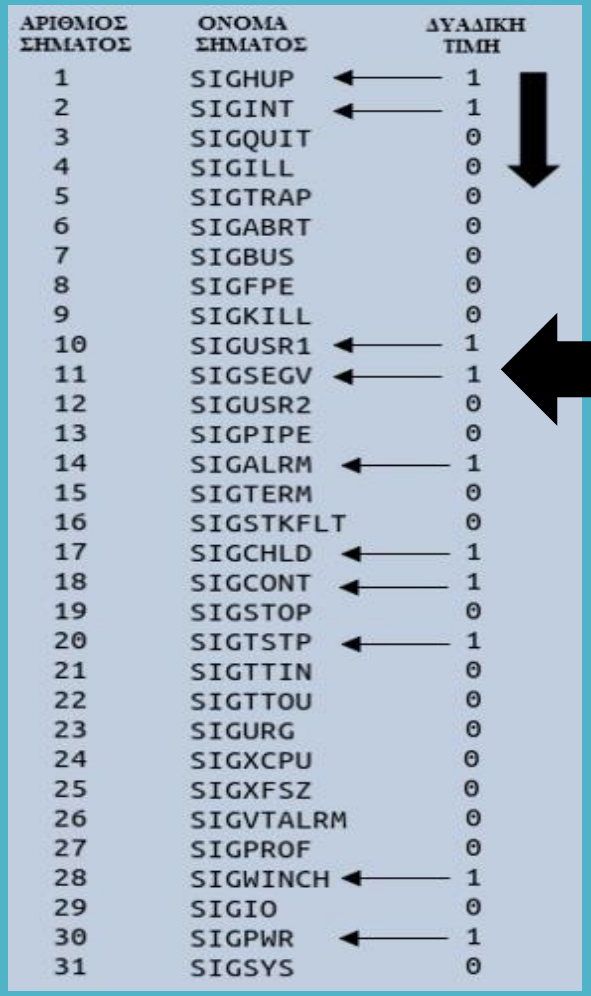
Ο δεκαεξαδικός αριθμος που βρίσκετα στα δεξιά της κάθε γραμμής αποτελεί μία μάσκα bit που εάν μετατραπεί στο δυαδικό σύστημα, δηλαδή σε ακολουθία από 0 και 1 περιέχει την πληροφορία σχετικά με το εάν το σήμα που αντιστοιχεί στο κάθε bit ικανοποιεί ή όχι την αντίστοιχη ιδιότητα.

00000000280b2603 ==> 0101000000010110010011000000011

Η μάσκα περιέχει 16 δεκαεξαδικούς αριθμούς και κατά συνέπεια αντιστοιχεί σε έναν δυαδικό αριθμό μήκους 64 bits. Από αυτά τα bits τα πρώτα 32 αντιστοιχούν στα standard signals ενώ τα υπόλοιπα 32 αντιστοιχούν στα real time signals του προτύπου POSIX.

Σήματα

Ανάκτηση των πληροφοριών της διεργασίας όσον αφορά στο χειρισμό των σημάτων της



ΑΡΙΘΜΟΣ ΣΗΜΑΤΟΣ	ΟΝΟΜΑ ΣΗΜΑΤΟΣ	ΔΥΑΔΙΚΗ ΤΙΜΗ
1	SIGHUP	1
2	SIGINT	1
3	SIGQUIT	0
4	SIGILL	0
5	SIGTRAP	0
6	SIGABRT	0
7	SIGBUS	0
8	SIGFPE	0
9	SIGKILL	0
10	SIGUSR1	1
11	SIGSEGV	1
12	SIGUSR2	0
13	SIGPIPE	0
14	SIGALRM	1
15	SIGTERM	0
16	SIGSTKFLT	0
17	SIGCHLD	1
18	SIGCONT	1
19	SIGSTOP	0
20	SIGTSTP	1
21	SIGTTIN	0
22	SIGTTOU	0
23	SIGURG	0
24	SIGXCPU	0
25	SIGXFSZ	0
26	SIGVTALRM	0
27	SIGPROF	0
28	SIGWINCH	1
29	SIGIO	0
30	SIGPWR	1
31	SIGSYS	0

Τα σήματα που αντιστοιχούν σε bit με τιμή 1 ικανοποιούν την ιδιότητα που σχετίζονται με τη μάσκα.

Στην προκειμένη περίπτωση η μάσκα είναι η

SigCgt → ομάδα σημάτων που έχουν συλληφθεί

και κατά συνέπεια τη χρονική στιγμή εξέτασης της μάσκας έχουν συλληφθεί τα σήματα

SIGHUP	SIGCHLD
SIGINT	SIGCONT
SIGUSR1	SIGSTP
SIGSEGV	SIGWINCH
SIGALARM	SIGTPWR

SigBlk → ομάδα μπλοκαρισμένων σημάτων
(όλα τα στοιχεία μηδέν → κανένα μπλοκαρισμένο σήμα)

Σήματα

Ανάκτηση των πληροφοριών της διεργασίας όσον αφορά στο χειρισμό των σημάτων της

```
SigIgn: fffffffe57f0d8fc
```

SigIgn → ομάδα σημάτων που αγνοούνται

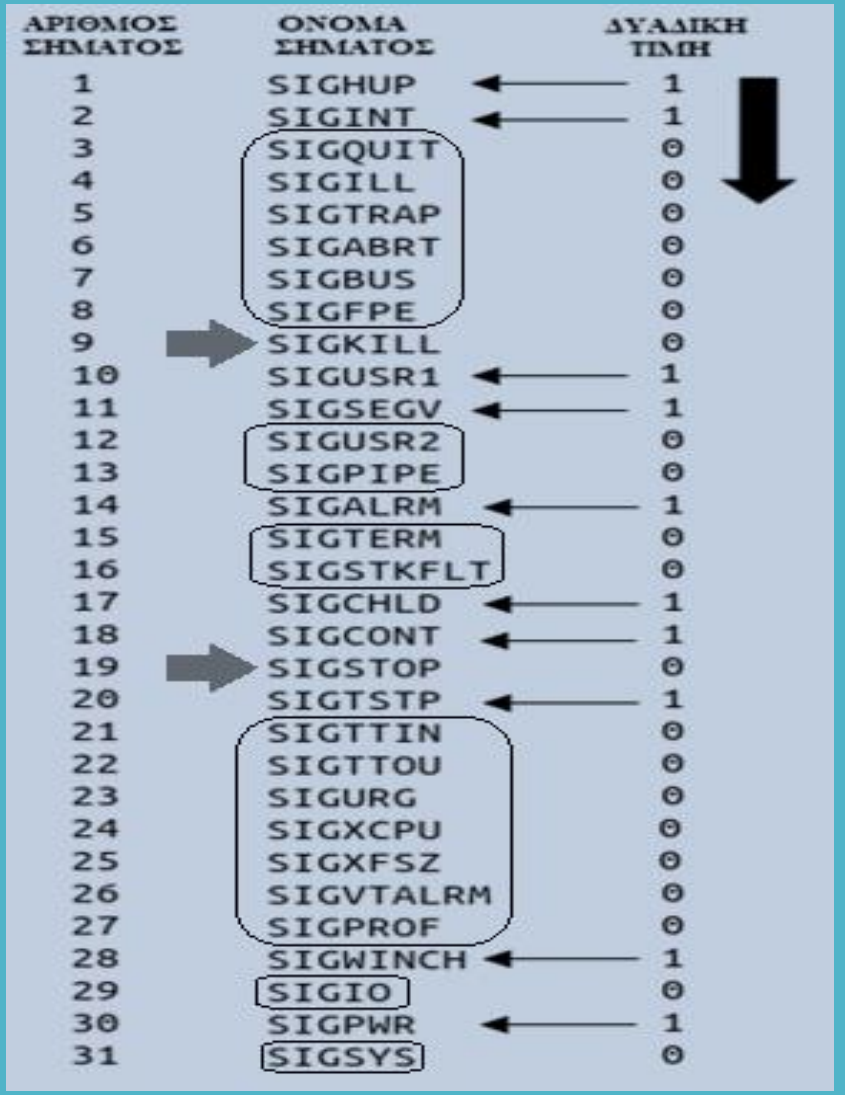
11111111111111111111111111111111

00101011111110000110110001111110 0

Τα πρώτα τριανταδύο 1 αντιστοιχούν στα σήματα πραγματικού χρόνου που όλα αγνοούνται

10101111111000011011000111111000

Το σύνολο των σημάτων που αγνοούνται είναι (προφανώς) ξένο του συνόλου των σημάτων που συλλαμβάνονται. Εάν πάρουμε την ένωση αυτών των δύο συνόλων και προσθέσουμε και τα δύο σήματα SIGKILL και SIGSTOP που υφίστανται χειρισμό MONO από τον πυρήνα καταλήγουμε στο σύνολο των 32 διαθέσιμων standard signals.



Σήματα

Παράδειγμα ανταλλαγής σήματος μεταξύ γονικής και θυγατρικής διεργασίας (Παράδειγμα Α)

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

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

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

void main() {
     int pid;
     signal(SIGHUP, sighup);
     signal(SIGINT, sigint);
     signal(SIGQUIT, sigquit);
     /* get child process */
     pid = fork ();
     if (pid < 0) {
          perror("fork");
          exit(1); }
     if (pid == 0) { for (;;);  }
     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); }}
```

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

```
amarg@amarg-vbox:~$ ./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!!!
```

Εδώ η διεργασία – γονέας στέλνει διαδοχικά στη θυγατρική διεργασία τα σήματα

SIGHUP
SIGINT
SIGQUIT

στα οποία αποκρίνεται καλώντας τον κατάλληλο χειριστή (sighup, sigint & sigquit)

Σήματα

Παράδειγμα ανταλλαγής σήματος μεταξύ γονικής και θυγατρικής διεργασίας (Παράδειγμα Β)

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

// SIGALARM = 14
// SIGCHLD =  17

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

int main() {
        signal(SIGALRM,signalHandler);
        signal(SIGUSR1,signalHandler);
        signal(SIGCHLD,signalHandler);
        if (!fork()) {
            printf("Child running...\n");
            sleep(2);
            printf("Child sending SIGALRM...\n");
            kill(getppid(),SIGALRM);
            sleep(5);
            printf("Child exitting...\n");
            return 0; }
        printf("Parent running, PID=%d. Press ENTER to exit.\n",getpid());
        getchar();
        printf("Parent exitting...\n");
        return 0; }
```

```
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...
```

Εδώ η θυγατρική διεργασία στέλνει στη γονική διεργασία μόνο το σήμα SIGALARM.

Ωστόσο, το μήνυμα Caught signal 17! Σημαίνει πως η διεργασία έλαβε και το σήμα SIGCHLD το οποίο στάλθηκε αυτόματα από το σύστημα όταν ολοκληρώθηκε η θυγατρική διεργασία.

Σήματα

Παράδειγμα ανταλλαγής σήματος μεταξύ γονικής και πολλών θυγατρικών διεργασιών

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

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); }

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

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

        int pid, i;
        signal(SIGCHLD, catch);
        for (i=0;i<NUMPROCS;i++) {
                pid=fork();
                if (pid<0) { perror("fork"); exit(1); }
                if (pid==0) child(i);
                else nprocs++; }
        printf("Parent process: going to sleep\n");
        while (nprocs != 0) {
                printf("parent: sleeping\n");
                sleep(60); }
        printf("Parent process: exiting\n");
        exit(0); }
```

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

Η γονική διεργασία δημιουργεί τέσσερις θυγατρικές διεργασίες και ανιχνεύει τον τερματισμό τους μέσω του σήματος SIGCHLD.

```
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
```

Σήματα

Σύνολα σημάτων. Ένας πολύ χρήσιμος τύπος δεδομένων είναι ο τύπος sigset_t (signal.h) ο οποίος χρησιμοποιείται για την περιγραφή συνόλου σημάτων σαν και αυτά που ορίζονται μέσω της μάσκας.

Το πρότυπο POSIX προσφέρει πέντε συναρτήσεις χειρισμού συνόλων σημάτων.

int sigemptyset (sigset_t * set) → αδειάζει το σύνολο σημάτων το οποίο εκκενώνεται.

int sigfillset (sigset_t * set) → προσθέτει στο σύνολο σημάτων όλα τα διαθέσιμα σήματα.

int sigaddset (sigset_t * set, int signum) → προσθέτει στο σύνολο σημάτων set το σήμα με κωδικό signum.

int sigdelset (sigset_t * set, int signum) → αφαιρεί από το σύνολο σημάτων set το σήμα με κωδικό signum.

int sigismember (const sigset_t * set, int signum) → επιστρέφει τιμή true (μη μηδενική) ή false (μηδενική) ανάλογα με το εάν το σήμα signum ανήκει ή όχι στην ομάδα σημάτων set.

Οι δύο τρόποι αρχικοποίησης του sigset_t είναι να το εκκενώσουμε με την sigemptyset και μετά να προσθέσουμε τα σήματα που θέλουμε με την sigaddset ή να προσθέσουμε σε αυτό όλα τα διαθέσιμα σήματα με την sigfillset και μετά να απομακρύνουμε αυτά που δεν χρειάζονται με την sigdelset.

Σήματα

Αντί για την παρωχημένη συνάρτηση signal χρησιμοποιείται η sigaction που δηλώνεται ως

int sigaction (int signum, struct sigaction * act, struct sigaction * oact);

όπου signum ο αριθμός του σήματος και act ο χειριστής του σήματος.

Εάν είναι oact != NULL αυτή περιγράφει τις ρυθμίσεις του σήματος πριν την κλήση της sigaction.
Εάν είναι act = NULL η τρέχουσα ρύθμιση του σήματος παραμένει αμετάβλητη.

Η επιστρεφόμενη τιμή είναι μηδενική για επιτυχή εκτέλεση και αρνητική όταν εμφανίζεται σφάλμα.

Η δομή sigaction ορίζεται ως

```
struct sigaction {
        sighandler_t sa_handler;
        sigset_t sa_mask;
        unsigned long sa_flags;
        void (* sa_restorer) (void); }
```

Ο sa_handler είναι δείκτης σε μία συνάρτηση της μορφής

void handler (int signum);

ή εναλλακτικά να έχει μία από τις τιμές SIG_IGN (ignore) και SIG_DFL (do something).

Σήματα

Το πεδίο sa_flags περιέχει flags που επιτρέπουν στη διεργασία να τροποποιεί διαφορετικές συμπεριφορές σημάτων.

Το πεδίο sa_mask περιέχει τα σήματα που θα πρέπει να διακόπτονται όταν καλείται ο χειριστής του σήματος. Αυτά τα σήματα ΔΕΝ απορρίπτονται αλλά η μεταβίβασή τους καθυστερεί έως ότου η διεργασία είναι σε θέση να προχωρήσει στο χειρισμό τους.

Χρήσιμες συναρτήσεις διαχείρισης μάσκας

int sigpropmask (int what, const sigset_t * set, sigset_t * oldest);

Το όρισμα what καθορίζει τον τρόπο χειρισμού της μάσκας (εάν είναι set = NULL, αγνοείται) με τιμές:

SIG_BLOCK → τα σήματα του set προτίθενται στην τρέχουσα μάσκα σήματος.
SIG_UNBLOCK → τα σήματα του set διαγράφονται από την τρέχουσα μάσκα σήματος.
SIG_SETMASK → τα σήματα του set διακόπτωνται ενώ τα άλλα παραμένουν μη διακοπτώμενα.

Το oldest είναι η αρχική μάσκα και αγνοείται εάν έχει τιμή NULL. Η τρέχουσα μάσκα βρίσκεται ως

sigpropmask (SIG_BLOCK, NULL, &currentSet);

Σήματα

Η ανάκτηση της λίστας των εκκρεμών σημάτων γίνεται με τη συνάρτηση

int sigpending (sigset_t * set);

Η διαδικασία της αναμονής ενός σήματος πραγματοποιείται από τη συνάρτηση

int pause (void);

η οποία δεν επιστρέφει παρά μόνο όταν ένα σήμα μεταβιβαστεί στη διεργασία. Εναλλακτικά μπορούμε να καταφύγουμε στη χρήση της συνάρτησης

int sigsuspend (const sigset_t * mask)

Σήματα

Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος

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

static int got_signal = 0;
static void hdl (int sig) { got_signal = 1; }

int main (int argc, char *argv[]) {
        sigset_t mask;
        sigset_t orig_mask;
        struct sigaction act;
        memset (&act, 0, sizeof(act));
        act.sa_handler = hdl;
        if (sigaction(SIGTERM, &act, 0)) {
                perror ("sigaction");
                return 1; }
        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");
        if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0) {
                perror ("sigprocmask");
                return 1; }
        // SIGTERM signal is blocked
        printf ("Blocking SIGTERM signal...\n");
        if (sigprocmask(SIG_SETMASK, &orig_mask, NULL) < 0) {
                perror ("sigprocmask");
                return 1; }
        printf ("Sleeping for 2 seconds\n");
        sleep (2);
        if (got_signal) puts ("Got signal");
        return 0; }
```

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.

Χρησιμοποιούνται οι συναρτήσεις

- sigaction
- sigsmptyset
- sigaddset
- sigprocmask

```
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
```

Σήματα

Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος

Η εφαρμογή δέχεται τα σήματα SIGHUP, SIGUSR1 και SIGINT που δίνει ο χρήστης από άλλο terminal

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

void handle_signal(int signal);
void handle_sigalrm(int signal);
void do_sleep(int seconds);

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
    sa.sa_flags = SA_RESTART;
    // Block every signal during the handler
    sigfillset(&sa.sa_mask);
    // 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"); }
    // Will always fail, SIGKILL is intended to force kill your process
    if (sigaction(SIGKILL, &sa, NULL) == -1) { perror("Cannot handle SIGKILL"); }
    for (;;) {
        printf("\nSleeping for ~3 seconds\n");
        do_sleep(3); }}
```

Το κυρίως πρόγραμμα: τα τρία σήματα ενδιαφέροντος συσχετίζονται με τον signal handler και στη συνέχεια το πρόγραμμα μπαίνει σε ένα infinite loop καλώντας επαναληπτικά τη συνάρτηση do_sleep.

Σήματα

Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος

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

Ο signal handler εκτυπώνει ενημερωτικές πληροφορίες, καλεί την do_sleep και εκτυπώνει τα ονόματα των σημάτων που εκκρεμούν

Σήματα

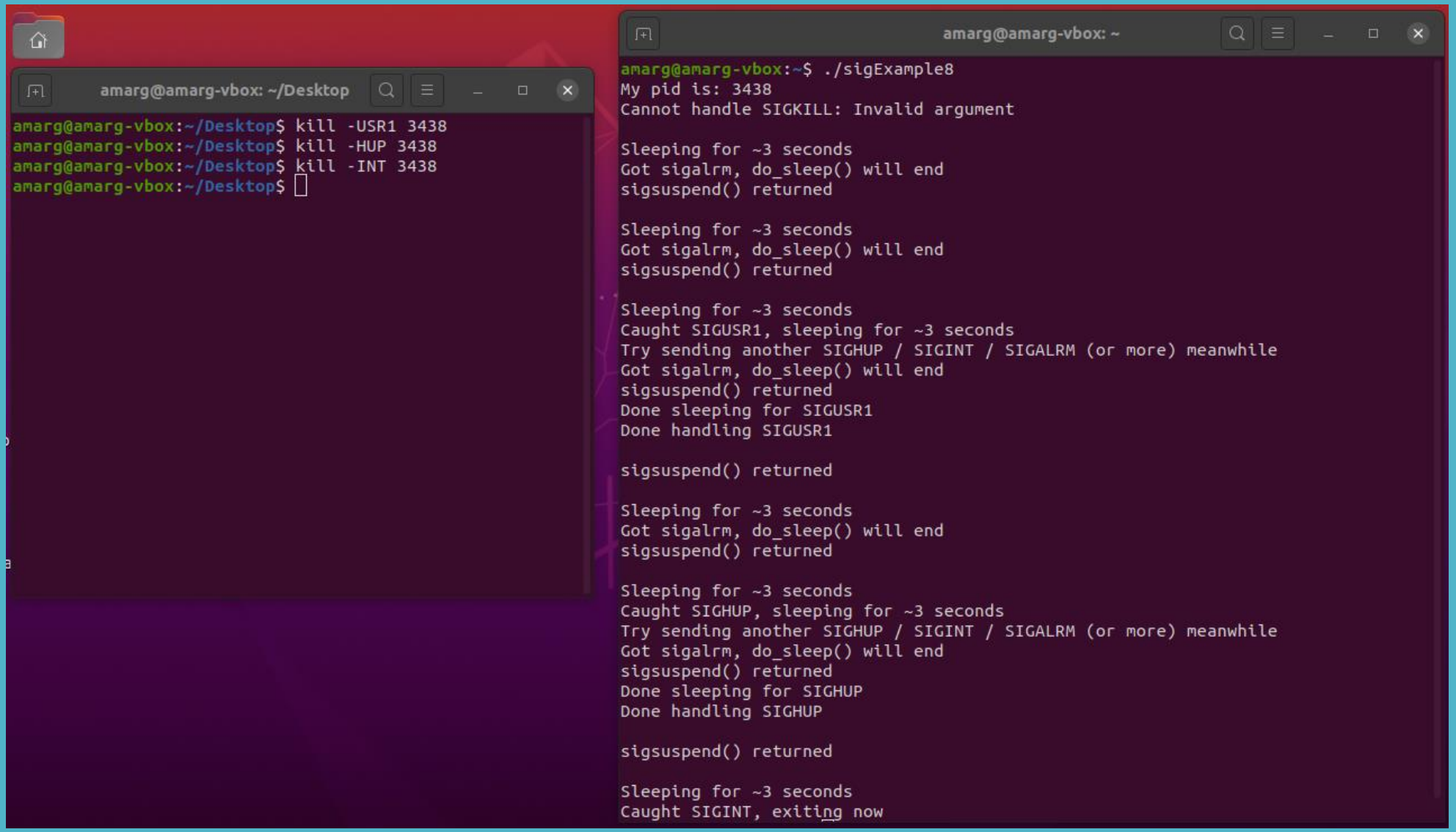
Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος

```
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) {
    struct sigaction sa;
    sigset_t mask;
    sa.sa_handler = &handle_sigalrm; // Intercept and ignore SIGALRM
    sa.sa_flags = SA_RESETHAND; // Remove the handler after first signal
    sigfillset(&sa.sa_mask);
    sigaction(SIGALRM, &sa, NULL);
    // Get the current signal mask
    sigprocmask(0, NULL, &mask);
    // Unblock SIGALRM
    sigdelset(&mask, SIGALRM);
    // Wait with this mask
    alarm(seconds);
    sigsuspend(&mask);
    printf("sigsuspend() returned\n"); }
```

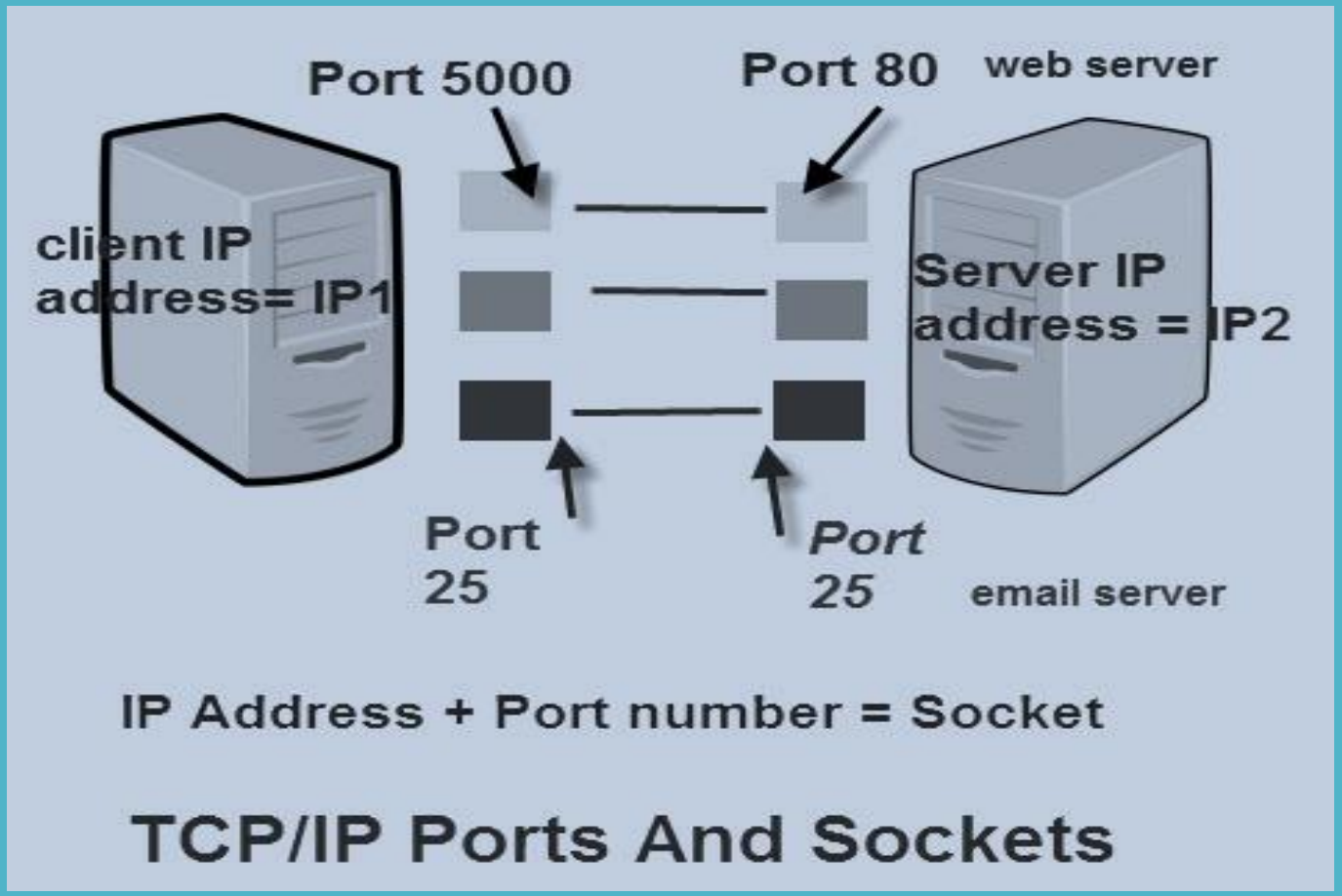
Η handle_sigalarm εκτυπώνει απλά ένα ενημερωτικό μήνυμα σχετικά με το σήμα που παρέλαβε. Η do_sleep συσχετίζει το SIGALARM με τη δομή sa που αρχικοποιείται κατάλληλα. Ανακτάται η τρέχουσα μάσκα, αφαιρείται από αυτή το SIGALARM και στη συνέχεια στέλνεται αυτό σήμα ύστερα από seconds δευτερόλεπτα.

Σήματα

Παράδειγμα χρήσης των συναρτήσεων διαχείρισης σήματος



Υποδοχείς


Port 5000
Port 80
web server
client IP address= IP1
Server IP address = IP2
Port 25
Port 25
email server
IP Address + Port number = Socket
TCP/IP Ports And Sockets

Υποδοχείς

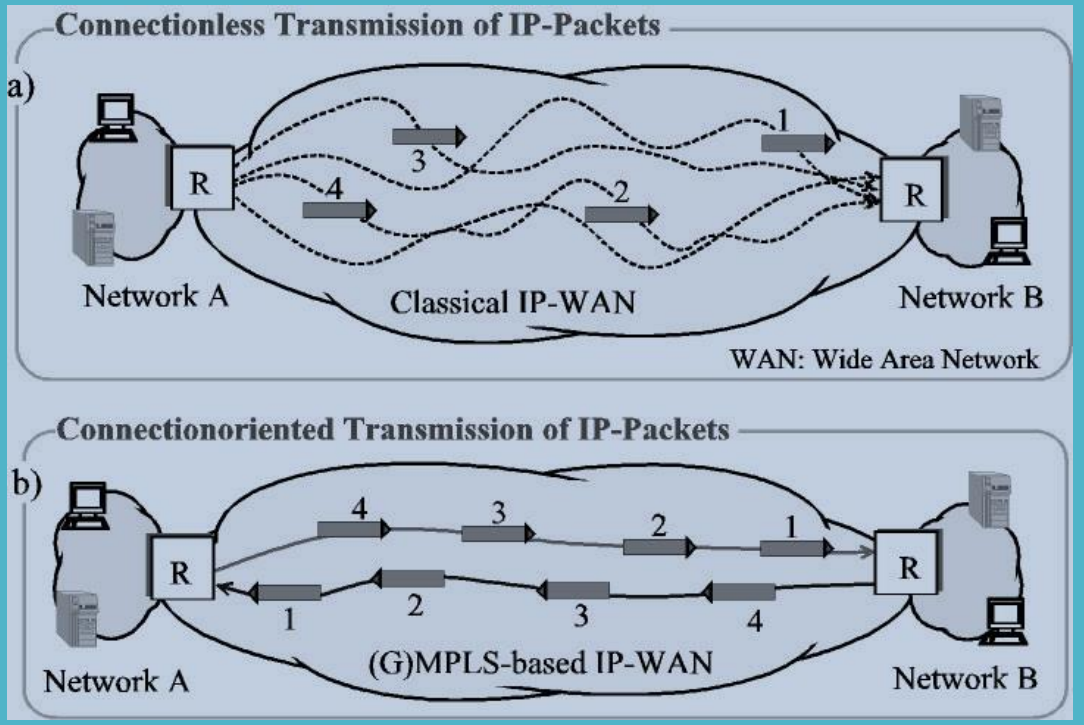
Οι υποδοχείς (sockets) είναι ειδικές δομές που επιτρέπουν την επικοινωνία ανάμεσα σε διεργασίες που εκτελούνται στον ίδιο ή σε διαφορετικούς υπολογιστές.

Η επικοινωνία μεταξύ υπολογιστών απαιτεί τη χρήση των κατάλληλων πρωτοκόλλων τα οποία είναι γνωστά ως δικτυακά πρωτόκολλα.

Η επικοινωνία μεταξύ υπολογιστών μπορεί να είναι τόσο με σύνδεση (connection oriented) όσο και χωρίς σύνδεση (connectionless).

Επικοινωνία με σύνδεση: η αποστολή των δεδομένων απαιτεί την αποκατάσταση της επικοινωνίας ανάμεσα στα δύο άκρα (τηλεφωνικό σύστημα).

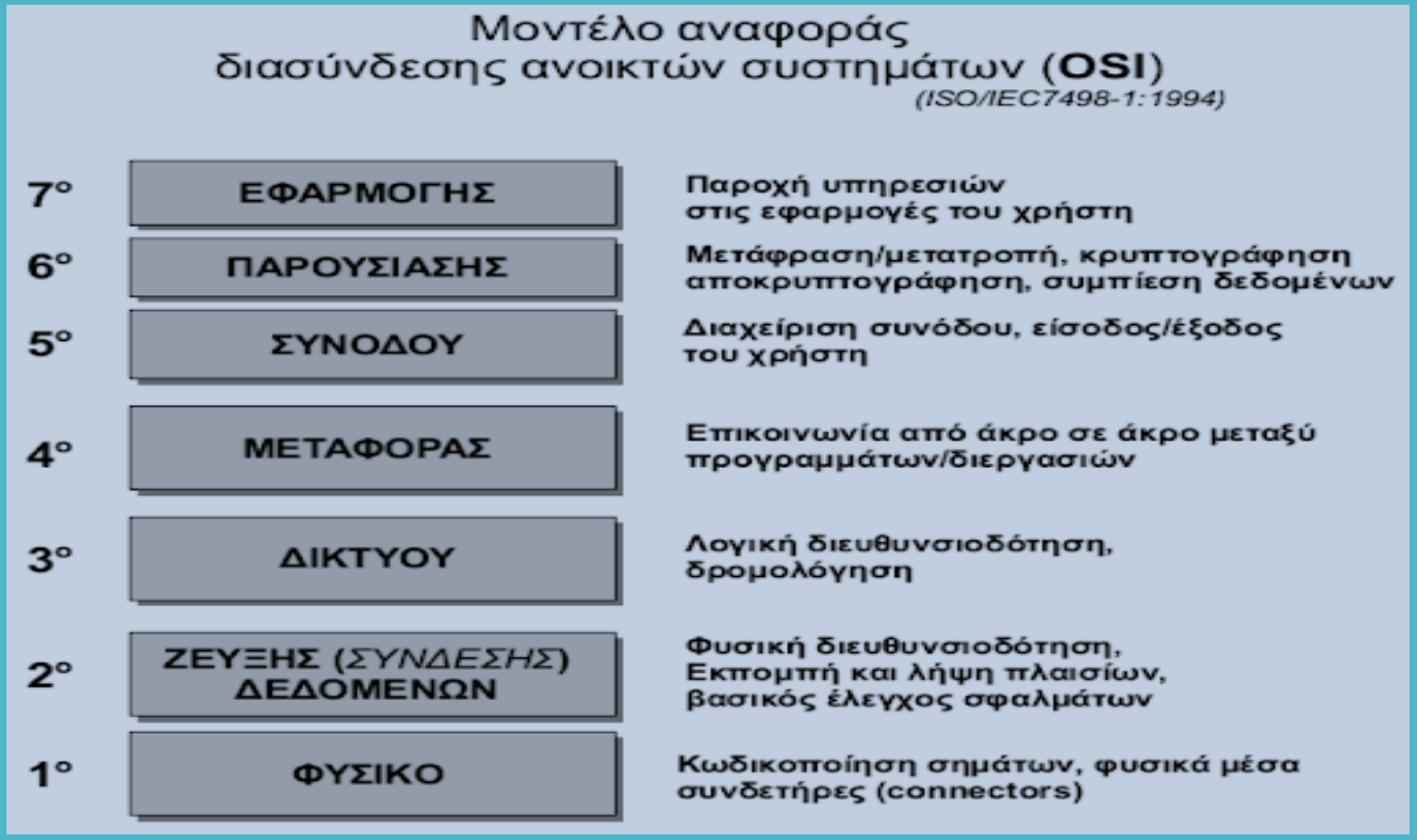
Επικοινωνία χωρίς σύνδεση: το κάθε πακέτο περιέχει τη διεύθυνση του παραλήπτη και δρομολογείται χωριστά και ανεξάρτητα από τα υπόλοιπα (ταχυδρομικό σύστημα.)



Υποδοχείς

Το μοντέλο αναφοράς OSI

Μοντέλο επτά επιπέδων – δεν χρησιμοποιείται ποτέ σε αυτή τη μορφή αλλά αποτελεί τη βάση και το πρότυπο πάνω στο οποίο στηρίζονται όλα τα δικτυακά μοντέλα αναφοράς

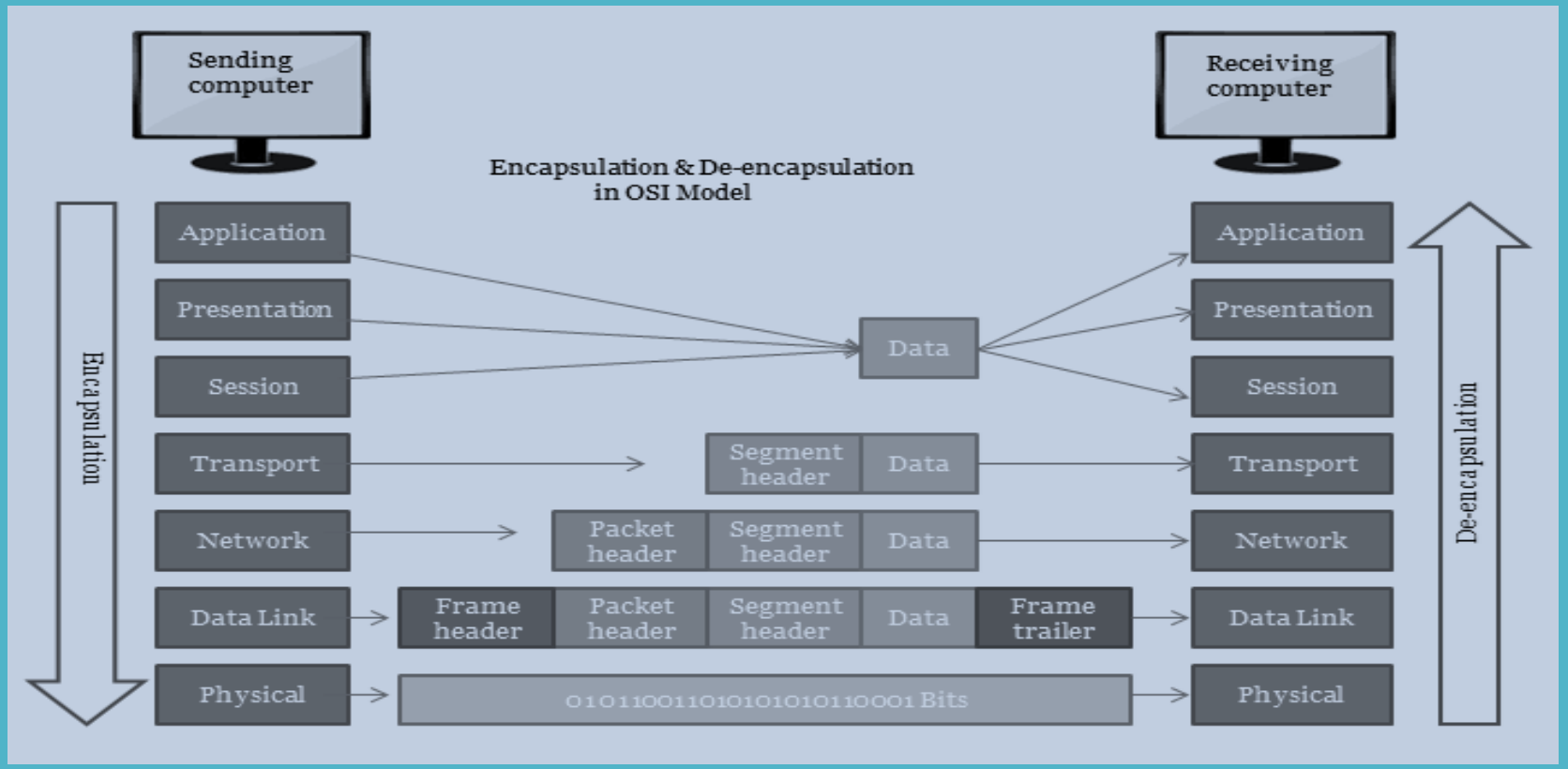


Υποδοχείς

Το μοντέλο αναφοράς OSI

Επικοινωνία υπολογιστών μέσω του μοντέλου αναφοράς OSI

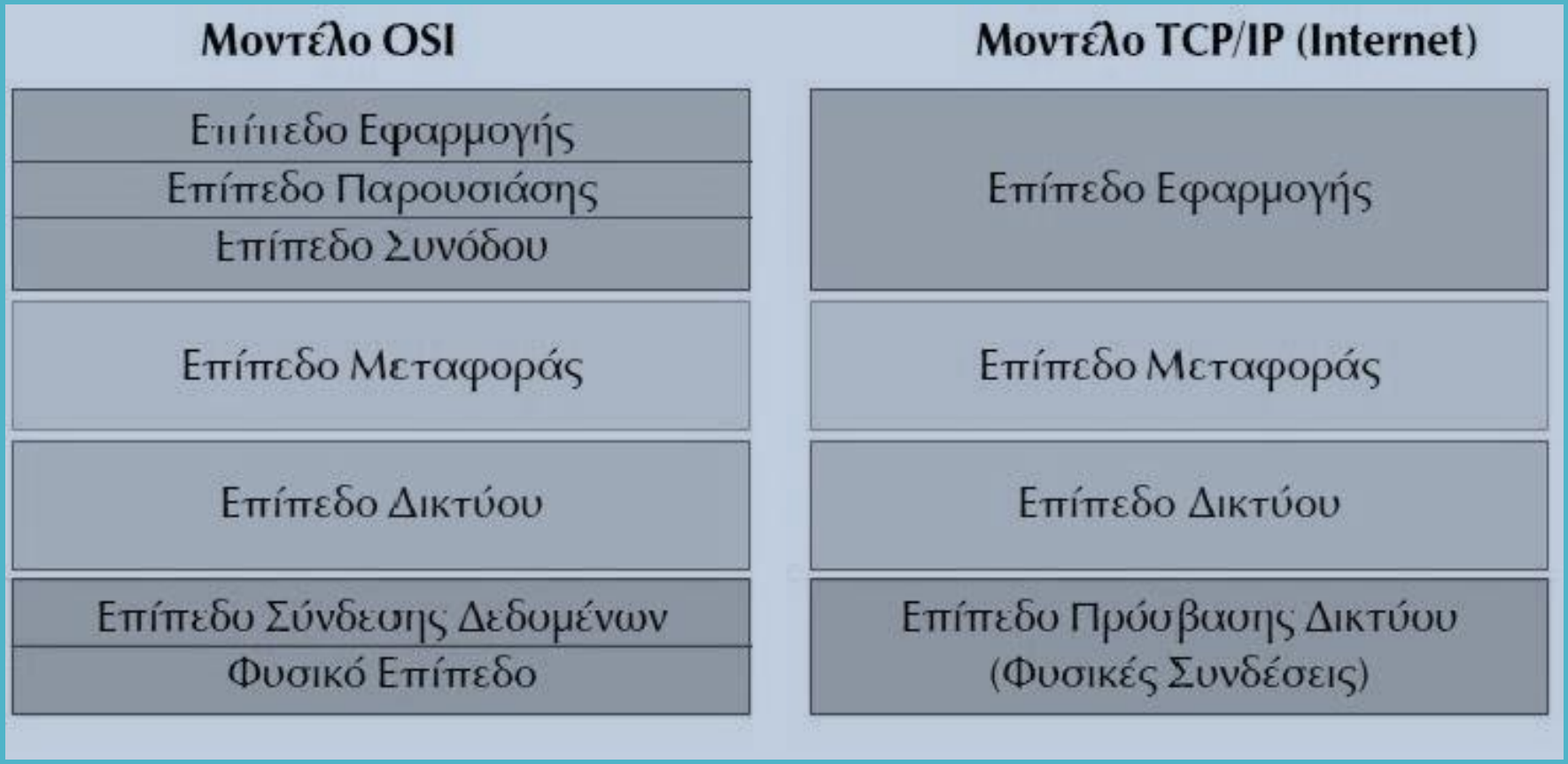
Στον υπολογιστή Α τα πακέτα κατεβαίνουν από το επίπεδο εφαρμογής προς το φυσικό επίπεδο, ενώ στον υπολογιστή Β κινούνται αντίστροφα. Κατά την κάθοδο, σε κάθε επίπεδο προστίθενται δεδομένα τα οποία αφαιρούνται κατά την άνοδο στον υπολογιστή – παραλήπτη (ενθυλάκωση δεδομένων).



Υποδοχείς

Το μοντέλο αναφοράς TCP / IP

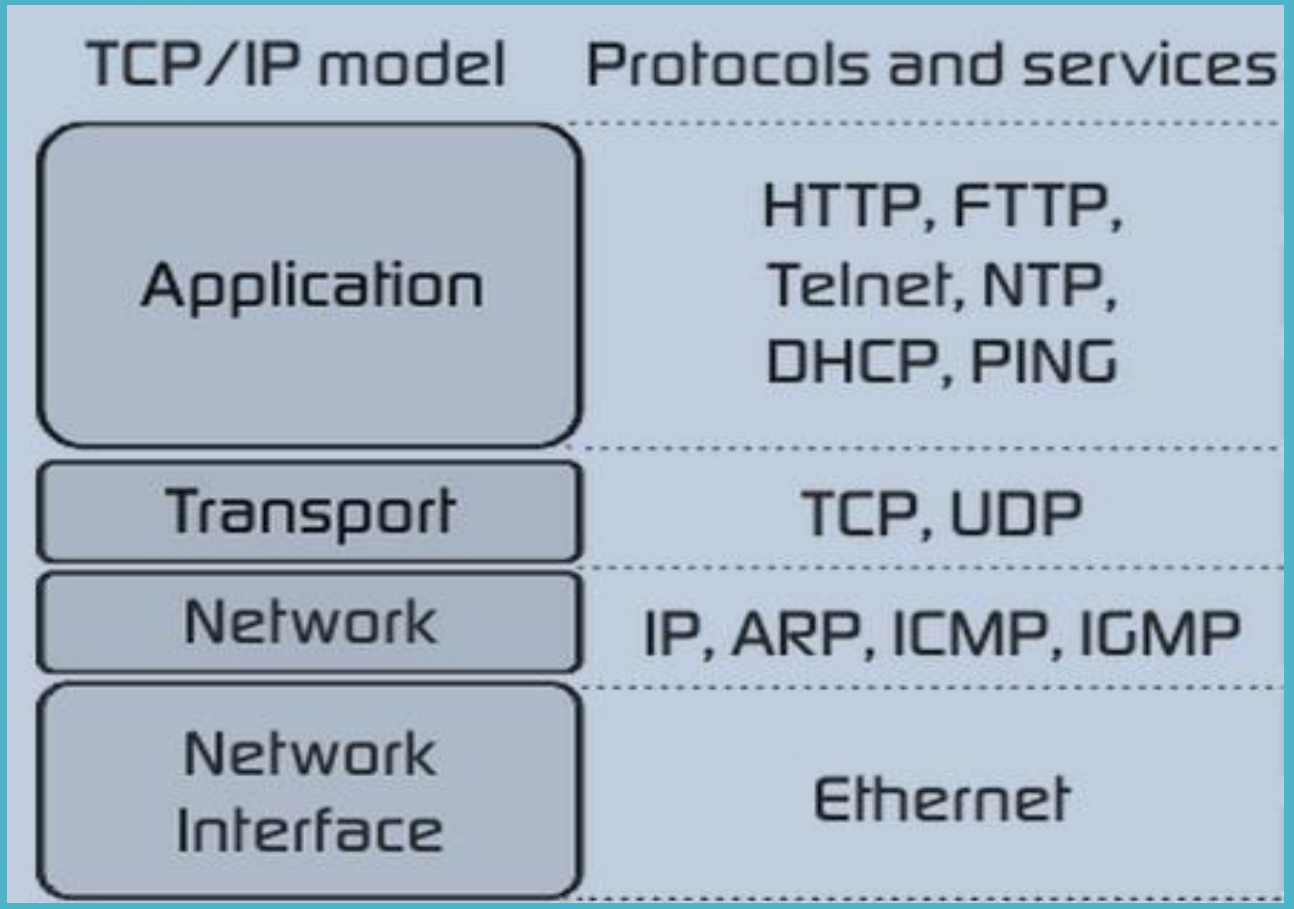
Αποτελεί το πιο διαδεδομένο δικτυακό πρωτόκολλο και συνιστά απαραίτητη προϋπόθεση για τη σύνδεση ενός υπολογιστή στο παγκόσμιο διαδίκτυο. Περιέχει τέσσερα επίπεδα.



Υποδοχείς

Το μοντέλο αναφοράς TCP / IP

Υπάρχουν πολλά πρωτόκολλα που μπορούν να ενταχθούν στα τέσσερα επίπεδα του TCP / IP και τα πιο σημαντικά από αυτά παρουσιάζονται στη συνέχεια.

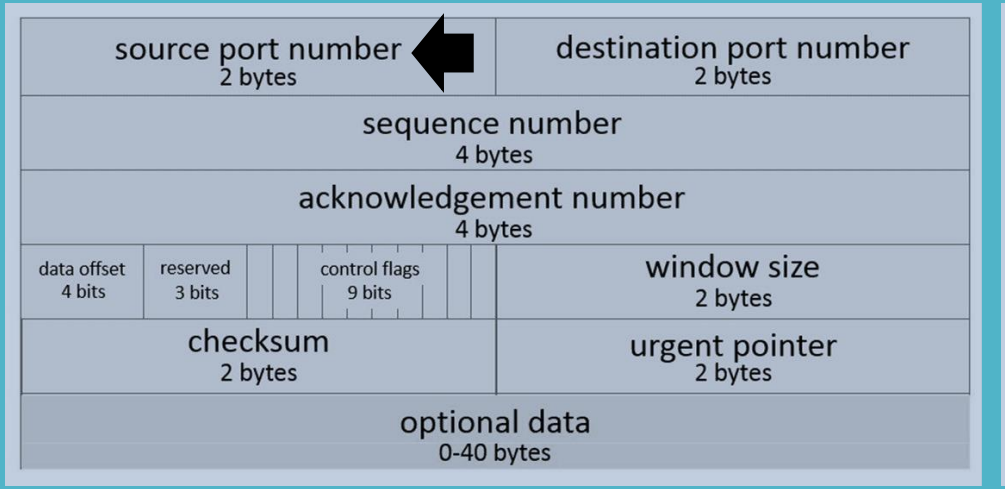


Υποδοχείς

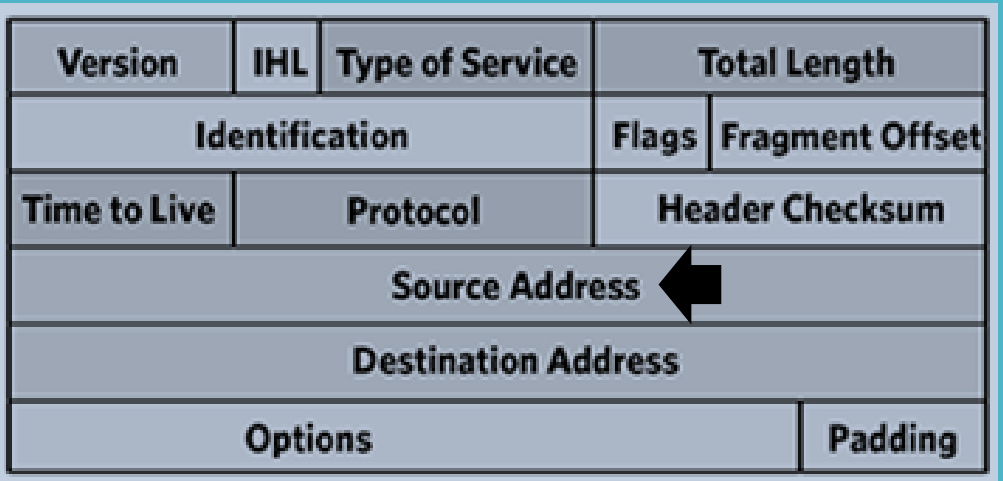
Το μοντέλο αναφοράς TCP / IP

Οι επικεφαλίδες που προστίθενται στα πακέτα δεδομένων στα επίπεδα μεταφοράς, δικτύου και πρόσβασης δικτύου (πρωτόκολλα TCP, IP και Ethernet).

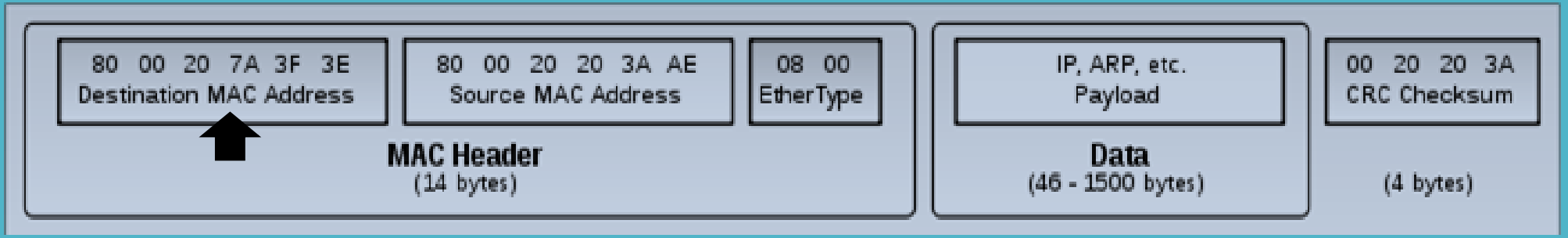
TCP Header



IP Header



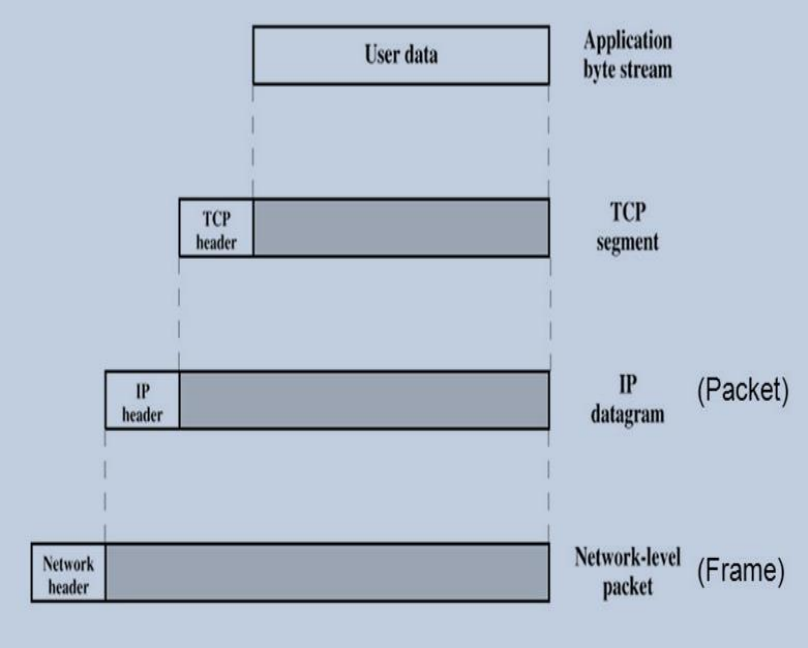
Ethernet header

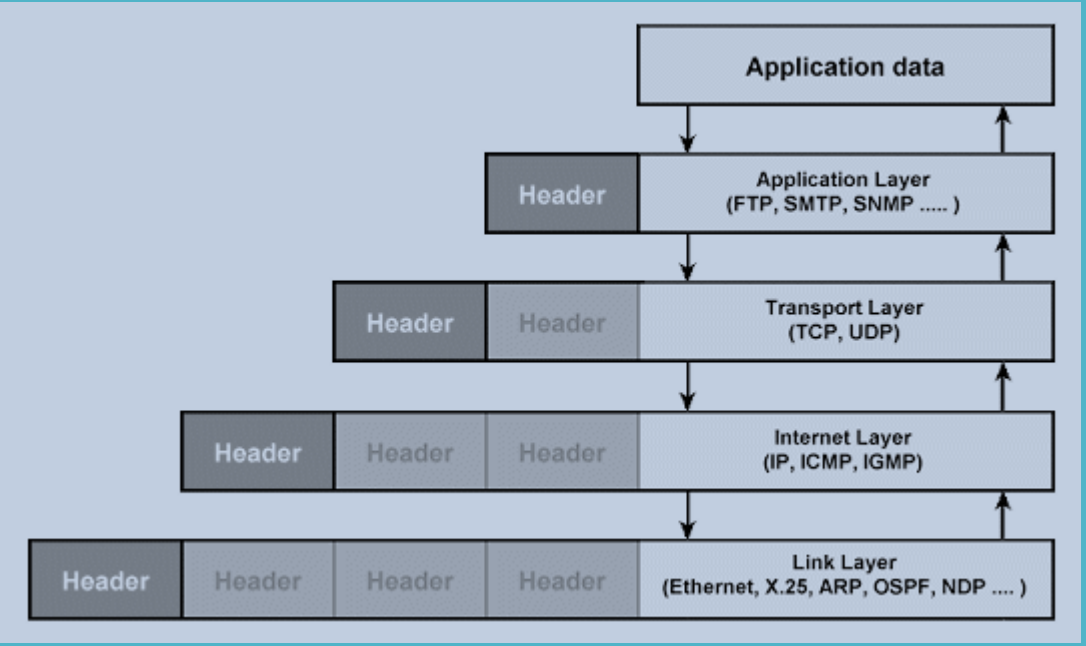


Υποδοχείς

Το μοντέλο αναφοράς TCP / IP

Το βασικό χαρακτηριστικό της ενθυλάκωσης δεδομένων είναι πως το κάθε επίπεδο δεν γνωρίζει την εσωτερική δομή που πακέτου που παραλαμβάνει – με άλλα λόγια το θεωρεί ως ένα ενιαίο πακέτο δεδομένων και απλά προσθέτει σε αυτό τη δική του επικεφαλίδα και τίποτε περισσότερο.

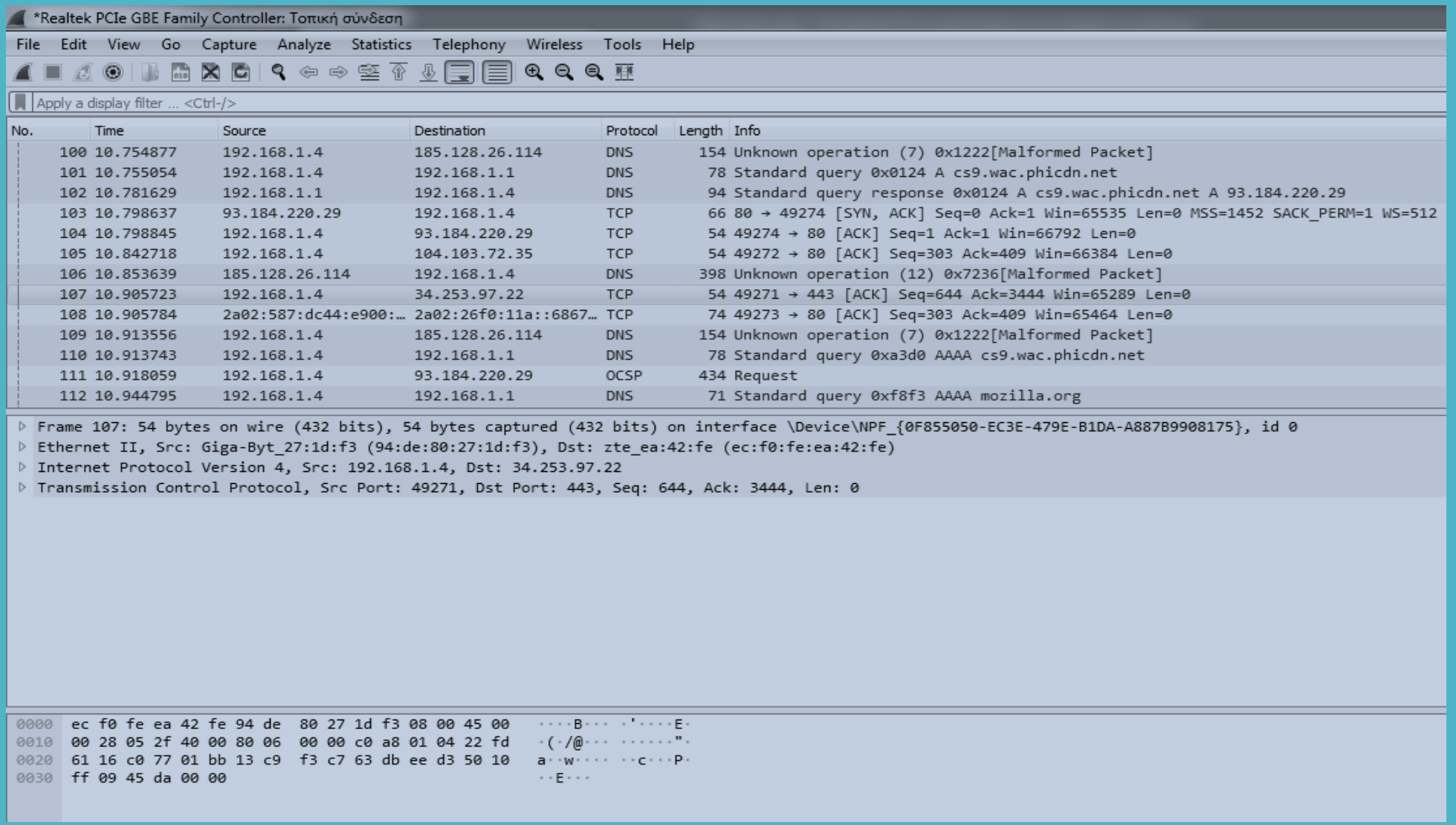




Για παράδειγμα, το IP δεν γνωρίζει πως μέσα στο TCP Segment υπάρχει TCP header - το μόνο που κάνει είναι να προσθέσει το δικό του header και να το προωθήσει στο επόμενο επίπεδο χωρίς να γνωρίζει (και στην πραγματικότητα, χωρίς να το ενδιαφέρει) τι υπάρχει μέσα στο TCP segment.

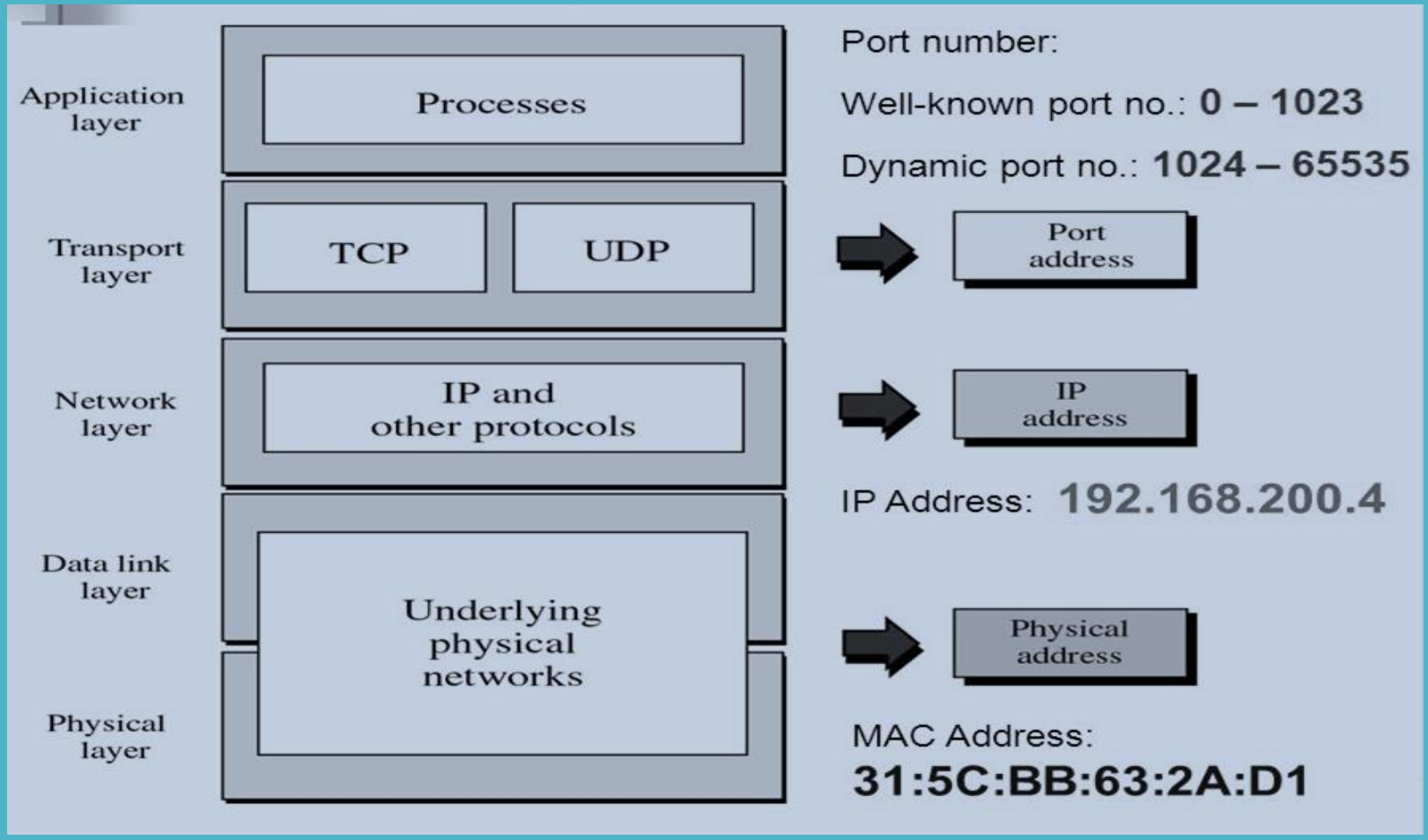
Υποδοχείς

Η βιβλιοθήκη npcap (παλαιότερα winpcap) και η εφαρμογή Wireshark



Υποδοχείς

Με πιο τρόπο είναι δυνατή η ταυτοποίηση ενός υπολογιστή σε κάθε επίπεδο? Αυτή η ταυτοποίηση απαιτείται στο επίπεδο μεταφοράς (port number), στο επίπεδο δικτύου (IP address) και στο επίπεδο πρόσβασης δικτύου (MAC address).



Υποδοχείς

Ανάκτηση των πληροφοριών διαμόρφωσης στο λειτουργικό σύστημα Linux – η εντολή ifconfig

```
amarg@amarg-vbox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::2079:b6d9:76c6:8708  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:59:2e:04  txqueuelen 1000  (Ethernet)
        RX packets 243  bytes 219625 (219.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 181  bytes 17403 (17.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 76  bytes 7294 (7.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 76  bytes 7294 (7.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

internet

loopback

Υποδοχείς

TCP / IP sockets vs Unix sockets

Ένα TCP / IP socket είναι ένας συνδυασμός της μορφής

IP Address : Port Number

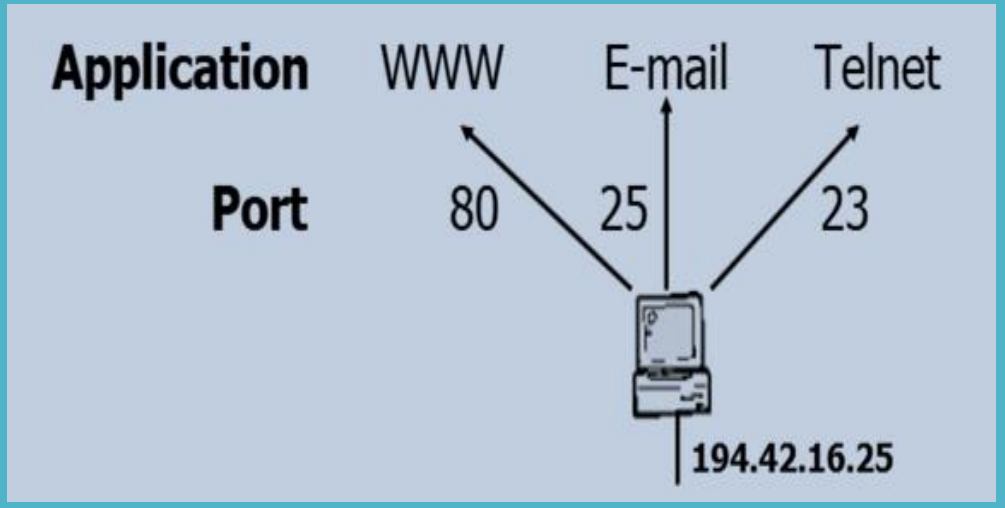
για παράδειγμα,

194.42.16.25 : 21

Στον παραπάνω συνδυασμό, η διεύθυνση IP προσδιορίζει με μοναδικό τρόπο έναν και μοναδικό υπολογιστή του παγκόσμιου διαδικτύου, ενώ ο αριθμός θύρας προσδιορίζει μία υπηρεσία σε αυτόν τον υπολογιστή.

Το σύστημα χρησιμοποιεί τα ports 0 – 1023 ενώ τα υπόλοιπα (με τιμές μέχρι 65536) χρησιμοποιούνται από τις εφαρμογές των χρηστών. Γνωστοί αριθμοί θύρας είναι

www → 80
ftp → 21
email → 25
ssh → 22
telnet → 23



Υποδοχείς

TCP / IP sockets vs Unix sockets

Ένα UNIX socket ή IPC (Interprocess communication) socket αποτελεί έναν μηχανισμό επικοινωνίας ανάμεσα σε διεργασίες που εκτελούνται στον ίδιο υπολογιστή που επιτρέπει αμφίδρομη επικοινωνία και ανταλλαγή δεδομένων.

Χρησιμοποιούνται με παρόμοιο τρόπο με τα TCP / IP sockets αλλά δεν στηρίζονται σε κάποιο υποκείμενο δικτυακό πρωτόκολλο που να επιτρέπει την επικοινωνία ανάμεσα σε δύο υπολογιστές – αντίθετα όλη η επικοινωνία πραγματοποιείται μέσα στον πυρήνα του λειτουργικού συστήματος.

Οι διεργασίες του συστήματος αντιλαμβάνονται τα UNIX sockets ως i-nodes και κατά συνέπεια, δύο διεργασίες μπορούν να επικοινωνήσουν μεταξύ τους , ανοίγοντας το ίδιο socket.

Τα UNIX sockets γνωρίζουν πως εκτελούνται στον ίδιο υπολογιστή και για το λόγο αυτό δεν καταφεύγουν σε πολύπλοκες διαδικασίες και ελέγχους όπως η δρομολόγηση και ο έλεγχος σφαλμάτων, γεγονός που τα καθιστά πιο αποτελεσματικά από τα TCP / IP sockets. Ωστόσο, υπόκεινται στον ίδιο έλεγχο πρόσβασης με τα συστήματα αρχείων, ενώ τα TCP / IP sockets υφίστανται έλεγχο σε επίπεδο φίλτρου πακέτου.

Εάν γνωρίζουμε πως η εφαρμογή μας θα εκτελεστεί σε έναν απλό υπολογιστή, είναι προτιμότερο να χρησιμοποιήσουμε UNIX sockets παρά TCP / IP sockets.

Υποδοχείς

TCP / IP sockets vs Unix sockets

Στα Unix sockets οι διευθύνσεις είναι ονόματα διαδρομής που δημιουργούνται στο σύστημα αρχείων όταν ένα socket συνδέεται στο όνομα διαδρομής. Αυτά τα αρχεία δεν ανοίγουν με τη συνάρτηση open αλλά με την κατάλληλη συνάρτηση της βιβλιοθήκης διαχείρισης των sockets.

Τα Unix sockets υποστηρίζουν τόσο STREAMS όσο και DGRAMS με τον πρώτο τύπο σύνδεσης να μοιάζει με τους αγωγούς (pipes) χωρίς όμως να ταυτίζεται μαζί τους.

Στην πραγματικότητα, τα Unix sockets πλεονεκτούν έναντι των αγωγών στο ότι είναι πλήρως αμφίδρομα και για το λόγο αυτό χρησιμοποιούνται για επικοινωνία μεταξύ διεργασιών (Interprocess Communication, IPC). Για τη διευκόλυνση του χρήστη προσφέρεται η συνάρτηση

int socketpair (int domain, int type, int protocol, int sockfds [2]);

με το τελευταίο όρισμά να περιέχει τους περιγραφείς αρχείων για τα δύο άκρα της σύνδεσης και τα υπόλοιπα ορίσματα να καθορίζουν τα χαρακτηριστικά του socket με τον τρόπο που θα δούμε σε λίγο.

Υποδοχείς

Δημιουργία socket

Η δημιουργία ενός socket γίνεται με την κλήση συστήματος socket η οποία δηλώνεται ως

int socket (int domain, int type, int protocol);

όπου domain η οικογένεια πρωτοκόλλων επικοινωνίας που θα χρησιμοποιηθεί στην εντολή:

- **AF_UNIX** Local communication
- **AF_LOCAL** Synonym for **AF_UNIX**
- **AF_INET** IPv4 Internet protocols
- **AF_AX25** Amateur radio AX.25 protocol
- **AF_IPX** IPX - Novell protocols
- **AF_APPLETALK** AppleTalk
- **AF_X25** ITU-T X.25 / ISO-8208 protocol
- **AF_INET6** IPv6 Internet protocols
- **AF_DECnet** DECet protocol sockets
- **AF_KEY** Key management protocol
- **AF_NETLINK** Kernel user interface device
- **AF_PACKET** Low-level packet interface
- **AF_RDS** Reliable Datagram Sockets (RDS
- **AF_PPPOX** Generic PPP transport layer
- **AF_LLC** Logical link control (IEEE 802.2 LLC)
- **AF_IB** InfiniBand native addressing
- **AF_MPLS** Multiprotocol Label Switching
- **AF_CAN** Controller Area Network automotive bus protocol
- **AF_TIPC** TIPC, "cluster domain sockets" protocol
- **AF_BLUETOOTH** Bluetooth low-level socket protocol
- **AF_ALG** Interface to kernel crypto API
- **AF_VSOCK** VSOCK (originally "VMWare VSockets")
- **AF_KCM** KCM (kernel connection multiplexer) interface
- **AF_XDP** XDP (express data path) interface

Υποδοχείς

Δημιουργία socket

Το όρισμα type περιγράφει τον τύπο της επικοινωνίας και μπορεί να λάβει τις τιμές

SOCK_STREAM → προσφέρει αξιόπιστη επικοινωνία με σύνδεση με ρεύματα από bytes **(read / write ή send / recv).**
SOCK_DGRAM → προσφέρει αναξιόπιστη επικοινωνία χωρίς σύνδεση **(sendto / recvfrom)**.
SOCK_SEQPACKET → προσφέρει αξιόπιστη επικοινωνία με σύνδεση με ακολουθίας από αυτοδύναμα πακέτα.

Το όρισμα protocol περιγράφει το πρωτόκολλο της οικογένειας πρωτοκόλλων επικοινωνίας που θα χρησιμοποιηθεί. Η τιμή 0 ορίζει πως θα χρησιμοποιηθεί το προεπιλεγμένο πρωτόκολλο επικοινωνίας που είναι το TCP για επικοινωνία με σύνδεση και το UDP για επικοινωνία χωρίς σύνδεση.

TCP → Transmission Control Protocol
UDP → User Datagram Protocol

Τα ονόματα και οι κωδικοί των πρωτοκόλλων που μπορούν να χρησιμοποιηθούν, περιλαμβάνονται στο αρχείο διαμόρφωσης /etc/protocols.

Η εντολή socket επιστρέφει τιμή μικρότερη του μηδενός σε περίπτωση αποτυχίας, ενώ εάν η κλήση της πραγματοποιηθεί χωρίς πρόβλημα επιστρέφει έναν περιγραφέα αρχείου (file descriptor) με τιμή μεγαλύτερη ή ίση του μηδενός. Αυτή η τιμή ταυτοποιεί το socket σε όλες τις μεταγενέστερες διαδικασίες στις οποίες αυτό συμμετέχει.

Υποδοχείς

Τα περιεχόμενα του αρχείου /etc/protocols

```
ip      0       IP              # internet protocol, pseudo protocol number
hopopt  0       HOPOPT          # IPv6 Hop-by-Hop Option [RFC1883]
icmp    1       ICMP            # internet control message protocol
igmp    2       IGMP            # Internet Group Management
ggp     3       GGP             # gateway-gateway protocol
ipencap 4       IP-ENCAP        # IP encapsulated in IP (officially ``IP'')
st      5       ST              # ST datagram mode
tcp     6       TCP             # transmission control protocol
egp     8       EGP             # exterior gateway protocol
igp     9       IGP             # any private interior gateway (Cisco)
pup     12      PUP             # PARC universal packet protocol
udp     17      UDP             # user datagram protocol
hmp     20      HMP             # host monitoring protocol
xns-idp 22      XNS-IDP         # Xerox NS IDP
rdp     27      RDP             # "reliable datagram" protocol
iso-tp4 29      ISO-TP4         # ISO Transport Protocol class 4 [RFC905]
dccp    33      DCCP            # Datagram Congestion Control Prot. [RFC4340]
xtp     36      XTP             # Xpress Transfer Protocol
ddp     37      DDP             # Datagram Delivery Protocol
idpr-cmtp 38    IDPR-CMTP       # IDPR Control Message Transport
ipv6    41      IPv6            # Internet Protocol, version 6
ipv6-route 43   IPv6-Route       # Routing Header for IPv6
ipv6-frag 44    IPv6-Frag       # Fragment Header for IPv6
idrp    45      IDRP            # Inter-Domain Routing Protocol
rsvp    46      RSVP            # Reservation Protocol
gre     47      GRE             # General Routing Encapsulation
esp     50      IPSEC-ESP       # Encap Security Payload [RFC2406]
ah      51      IPSEC-AH        # Authentication Header [RFC2402]
skip    57      SKIP            # SKIP
ipv6-icmp 58    IPv6-ICMP       # ICMP for IPv6
ipv6-nonxt 59   IPv6-NoNxt      # No Next Header for IPv6
ipv6-opts 60    IPv6-Opts       # Destination Options for IPv6
```

Υποδοχείς

Τα περιεχόμενα του αρχείου /etc/protocols

```
rspf    73      RSPF CPHB       # Radio Shortest Path First (officially CPHB)
vmtp    81      VMTP            # Versatile Message Transport
eigrp   88      EIGRP           # Enhanced Interior Routing Protocol (Cisco)
ospf    89      OSPFIGP         # Open Shortest Path First IGP
ax.25   93      AX.25           # AX.25 frames
ipip    94      IPIP            # IP-within-IP Encapsulation Protocol
etherip 97      ETHERIP         # Ethernet-within-IP Encapsulation [RFC3378]
encap   98      ENCAP           # Yet Another IP encapsulation [RFC1241]
#       99                      # any private encryption scheme
pim     103     PIM             # Protocol Independent Multicast
ipcomp  108     IPCOMP          # IP Payload Compression Protocol
vrrp    112     VRRP            # Virtual Router Redundancy Protocol [RFC5798]
l2tp    115     L2TP            # Layer Two Tunneling Protocol [RFC2661]
isis    124     ISIS            # IS-IS over IPv4
sctp    132     SCTP            # Stream Control Transmission Protocol
fc      133     FC              # Fibre Channel
mobility-header 135 Mobility-Header # Mobility Support for IPv6 [RFC3775]
udplite 136     UDPLite         # UDP-Lite [RFC3828]
mpls-in-ip 137  MPLS-in-IP      # MPLS-in-IP [RFC4023]
manet   138                     # MANET Protocols [RFC5498]
hip     139     HIP             # Host Identity Protocol
shim6   140     Shim6           # Shim6 Protocol [RFC5533]
wesp    141     WESP            # Wrapped Encapsulating Security Payload
rohc    142     ROHC            # Robust Header Compression
```

Υποδοχείς

Ορισμός και ανάκτηση επιλογών για sockets

Ο ορισμός και η ανάκτηση των τιμών για τις παραμέτρους λειτουργίας ενός socket, πραγματοποιείται με τις συναρτήσεις setsockopt και getsockopt που δηλώνονται στο αρχείο <sys/socket.h>.

int **setsockopt** (int *s*, int *level*, int *optname*, const void* *optval*, int *optlen*);

int **getsockopt** (int *s*, int *level*, int *optname*, void* *optval*, int* *optlen*);

s: the socket the system call applies to

level: layer which handle the option

optname: identifies the option we are setting/getting

optval: the value taken by the option

optlen: the size of data structure *optval*

Οι τιμές του level είναι οι:

SOL_SOCKET (default)
IPROTO_IP (Πρωτόκολλο IP)
IPROTO_IPV6 (Πρωτόκολλο IPv6)
IPROTO_ICMP (Πρωτόκολλο ICMP)
IPROTO_RAW (RAW data protocol)
IPROTO_TCP (Πρωτόκολλο TCP)
IPROTO_UDP (Πρωτόκολλο UDP)

Για το καθένα από τα παραπάνω πρωτόκολλα υπάρχει ξεχωριστή λίστα επιλογών και τιμών με τις λεπτομέρειες και τις σχετικές πληροφορίες να μπορούν να βρεθούν στα αρχεία τεκμηρίωσης.

Υποδοχείς

Ορισμός και ανάκτηση επιλογών για sockets

level	*optname*	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•		Get pending error and clear		int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size		int
	SO_SNDBUF	•	•	Send buffer size		int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	Send buffer low-water mark		int
	SO_RCVTIMEO	•	•	Receive timeout		timeval{}
	SO_SNDTIMEO	•	•	Send timeout		timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•		Get socket type		int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback		u_char
	IP_{ADD,DROP}_MEMBERSHIP		•	Join or leave multicast group		ip_mreq{}
	IP_{BLOCK,UNBLOCK}_SOURCE		•	Block or unblock multicast source		ip_mreq_source{}
	IP_{ADD,DROP}_SOURCE_MEMBERSHIP		•	Join or leave source-specific multicast		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass		icmp6_filter{}

Υποδοχείς

Ορισμός και ανάκτηση επιλογών για sockets

IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets		int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address		sockaddr_in6()
	IPV6_PATHMTU	•		Retrieve current path MTU		ip6_mtuinfo()
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit		int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_XXX	•	•	Sticky ancillary data		(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface		u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit		int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP		•	Join multicast group		ipv6_mreq()
	IPV6_LEAVE_GROUP		•	Leave multicast group		ipv6_mreq()
IPPROTO_IP *or* IPPROTO_IPV6	MCAST_JOIN_GROUP		•	Join multicast group		group_req()
	MCAST_LEAVE_GROUP		•	Leave multicast group		group_source_req()
	MCAST_BLOCK_SOURCE		•	Block multicast source		group_source_req()
	MCAST_UNBLOCK_SOURCE		•	Unblock multicast source		group_source_req()
	MCAST_JOIN_SOURCE_GROUP		•	Join source-specific multicast		group_source_req()
	MCAST_LEAVE_SOURCE_GROUP		•	Leave source-specific multicast		group_source_req()

Υποδοχείς

Ορισμός και ανάκτηση επιλογών για sockets

level	*optname*	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	Disable Nagle algorithm	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Adaption layer indication		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examine and set association info		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Default send parameters		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP fragmentation	•	int
	SCTP_EVENTS	•	•	Notification events of interest		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Retrieve peer address status		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Mapped v4 addresses	•	int
	SCTP_INITMSG	•	•	Default INIT parameters		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Maximum burst size		int
	SCTP_MAXSEG	•	•	Maximum fragmentation size		int
	SCTP_NODELAY	•	•	Disable Nagle algorithm	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Peer address parameters		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Primary destination address		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO information		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Peer primary destination address		sctp_setpeerprim{}
	SCTP_STATUS	†		Get association status		sctp_status{}

Υποδοχείς

Αρχιτεκτονικές Client - Server

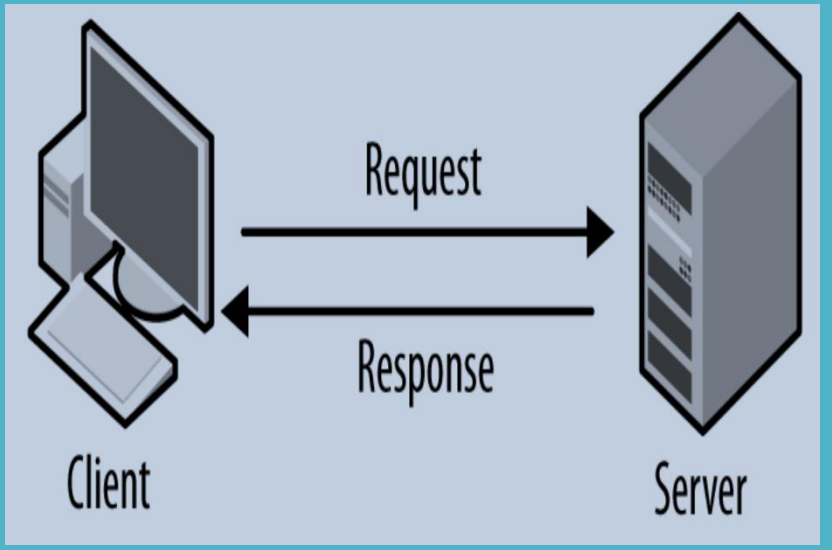
Στις αρχιτεκτονικές client – server υπάρχει ένας κεντρικός υπολογιστής (server) ο οποίος εξυπηρετεί αιτήσεις άλλων υπολογιστών (clients) που συνδέονται στον server μέσω δικτύου.

Ο client υποβάλλει στον server αιτήματα προς εξυπηρέτηση τα οποία διεκπεραιώνονται από τον server. Το αίτημα είτε εξυπηρετείται άμεσα είτε μπαίνει σε ουρά εκκρεμών μηνυμάτων εάν ο server είναι απασχολημένος.

Η αρχιτεκτονική client server έχει πολλά πλεονεκτήματα όπως:

1. Αποτελεσματική χρήση της υπολογιστικής ισχύος.
2. Μείωση του κόστους συντήρησης.
3. Αύξηση της παραγωγικότητας και της ευελιξίας.

Η επικοινωνία των client με τον server γίνεται μέσω socket.



ΠΑΡΑΔΕΙΓΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΩΝ CLIENT SERVER

1. Web Client – Web Server
2. FTP Client – FTP Server
3. Email Client – Email Server.

Υποδοχείς

Αρχιτεκτονικές Client - Server

Για την αποκατάσταση της σύνδεσης ο client και ο server λειτουργούν με διαφορετικό τρόπο:

Ειδικότερα:

Ο Client:

1. Δημιουργεί το socket καλώντας τη συνάρτηση socket.
2. Ενημερώνει το σύστημα σε ποια διεύθυνση θα συνδεθεί καλώντας τη συνάρτηση connect.

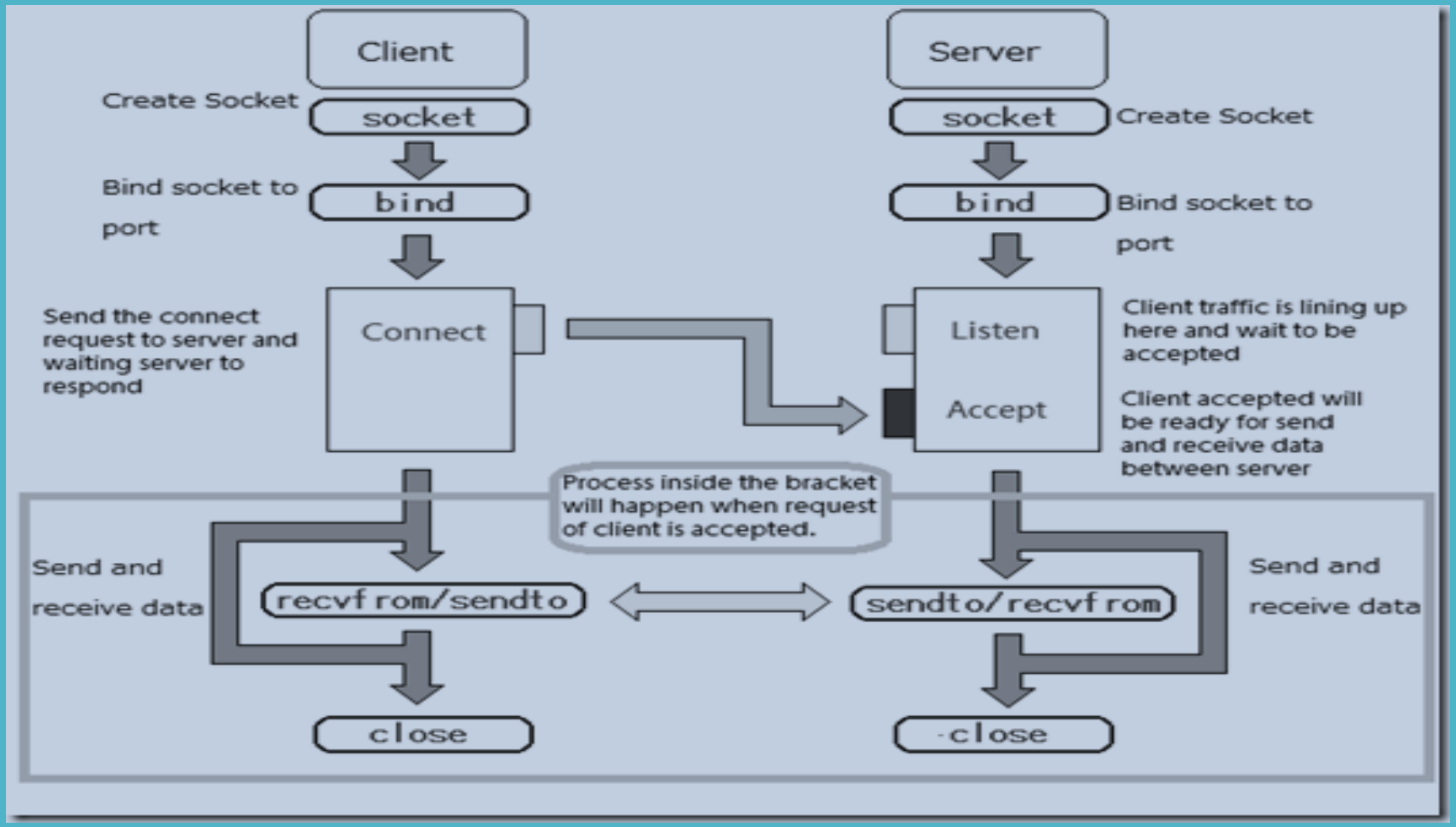
Ο Server:

1. Δημιουργεί το socket καλώντας τη συνάρτηση socket.
2. Συσχετίζει το socket που δημιούργησε με τη διεύθυνση στην οποία θα αναμένει αιτήματα σύνδεσης καλώντας τη συνάρτηση bind.
3. Αναμένει αιτήματα σύνδεσης καλώντας τη συνάρτηση listen.
4. Αποδέχεται αιτήματα σύνδεσης καλώντας τη συνάρτηση accept.

Υποδοχείς

Αρχιτεκτονικές Client - Server

Η αλληλεπίδραση του client με τον server με διαγραμματικό τρόπο απεικονίζεται στη συνέχεια.



Υποδοχείς

SOCKET PROGRAMMING

Η βασική δομή δεδομένων που χρησιμοποιείται με τις παραπάνω συναρτήσεις είναι η

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

(MAXSOCKADDRDATA=14) όπου sa_family η οικογένεια πρωτοκόλλων (π.χ. AF_UNIX, AF_INET) και sa_data δεδομένα που σχετίζονται με τη διεύθυνση του πρωτοκόλλου.

Η παραπάνω δομή είναι εντελώς γενική και χρησιμοποιείται με κάθε οικογένεια πρωτοκόλλων, ενώ για την ειδική περίπτωση των οικογενειών AF_UNIX και AF_INET μπορούν να χρησιμοποιηθούν οι δομές

UNIX SOCKET

```
struct sockaddr_un{
 uint8_t  sun_len;
 sa_family_t  sun_family;
 char  sun_path[104];   };
```

TCP/IP SOCKET

```
struct sockaddr_in {
   uint8_t          sin_len;
   sa_family_t      sin_family;
   in_port_t        sin_port;
   };  /* included in <netinet/in.h> */
```

όπου

```
struct in_addr { unsigned long s_addr; };
```

Υποδοχείς

SOCKET PROGRAMMING

Συσχέτιση του socket που δημιουργήθηκε, με την τοπική διεύθυνση για την επικοινωνία.

int bind (int sock, const struct sockaddr * addr, int addrlen);

Η bind συσχετίζει την τοπική διεύθυνση addr (η οποία θα πρέπει προηγουμένως να έχει αρχικοποιηθεί) με την ακέραια μεταβλητή sock που έχει επιστραφεί από την κλήση συστήματος socket.

Η bind καλείται υποχρεωτικά από την εφαρμογή του server προκειμένου αυτός να είναι σε θέση να δεχθεί αιτήματα σύνδεσης από την εφαρμογή του client ενώ αν και μπορεί να κληθεί και από την εφαρμογή του client αυτό δεν γίνεται σχεδόν ποτέ (γιατί δεν εξυπηρετεί σε τίποτε).

Εάν δεν συσχετίσουμε ένα socket μέσω της bind το λειτουργικό του εκχωρεί προσωρινά και για πολύ μικρό χρονικό διάστημα μία θύρα για την επίτευξη της επικοινωνίας.

Συνήθως δεν επιτρέπεται η συσχέτιση ενός socket με μία θύρα που ήδη χρησιμοποιείται.

Αν και οι δομές που χρησιμοποιούνται με AF_UNIX και AF_INET είναι αντίστοιχα οι sockaddr_un και sockaddr_in η bind (και η connect) απαιτούν δομή sockaddr – απαιτείται λοιπόν type casting.

Υποδοχείς

SOCKET PROGRAMMING

Αναμονή για αίτημα σύνδεσης πελάτη

int listen (int sock, int backlog);

Στην παραπάνω συνάρτηση που επίσης καλείται από τον server, το socket που περιγράφεται από το όρισμα sock που επέστρεψε η συνάρτηση socket, είναι ένα passive socket αφού το μόνο που κάνει είναι να αναμένει αιτήματα σύνδεσης πελατών προκειμένου να τα διεκπεραιώσει.

Τα αιτήματα σύνδεσης πελατών που δεν εξυπηρετούνται εν τη γενέσει τους χαρακτηρίζονται ως εκκρεμή και τοποθετούνται στο τέλος μιας ουράς αναμονής εξυπηρέτησης.

Αυτή η ουρά δεν είναι δυνατόν να αυξάνει απεριόριστα αλλά έχει πεπερασμένο μήκος το οποίο προσδιορίζεται από το δεύτερο όρισμα backlog η τυπική τιμή του οποίου είναι ίση με 5.

Εάν η ουρά αναμονής είναι πλήρης, ο πελάτης λαμβάνει το μήνυμα σφάλματος ECONNREFUSED ενώ σε περίπτωση επιτυχούς συσχέτισης η συνάρτηση επιστρέφει μηδενική τιμή.

Υποδοχείς

SOCKET PROGRAMMING

Υποβολή αιτήματος σύνδεσης από πελάτη

int connect (int sock, const struct sockaddr * addr, int addrlen);

Αυτή η συνάρτηση χρησιμοποιείται για την υποβολή ενός αιτήματος σύνδεσης του πελάτη και ένα άλλο TCP socket το οποίο βρίσκεται στην πλευρά του server.

Στα connection based sockets (SOCK_STREAM) αυτή η σύνδεση πραγματοποιείται μία και μοναδική φορά, ενώ στα connectionless sockets (SOCK_DGRAM) η συνάρτηση connect μπορεί να κληθεί πολλές φορές για να συνδεθεί κάθε φορά με διαφορετικό socket.

Όταν ο πελάτης υποβάλλει ένα αίτημα σύνδεσης, αυτό τοποθετείται στην ουρά αναμονής αιτημάτων εξυπηρέτησης που διατηρείται στην πλευρά του server όπου και παραμένει μέχρι ο server να κάνει αποδεκτό το αίτημα σύνδεσης.

Η connect αναστέλλει τη λειτουργία της μέχρι να γίνει αποδεκτό το αίτημα της σύνδεσης που υπέβαλλε. Η συνάρτηση επιστρέφει μηδέν σε περίπτωση επιτυχία και -1 σε περίπτωση αποτυχίας, αρχικοποιώντας με τον κατάλληλο τρόπο και τον κωδικό σφάλματος.

Υποδοχείς

SOCKET PROGRAMMING

Εξυπηρέτηση αιτήματος σύνδεσης πελάτη

int accept (int sock, struct sockaddr * addr, int addrlen);

Αυτή η συνάρτηση χρησιμοποιείται με connection based sockets (SOCK_STREAM) και δημιουργεί μία σύνδεση εξυπηρέτησης για το εκκρεμές αίτημα σύνδεσης που βρίσκεται στην κορυφή της ουράς αναμονής και σχετίζεται με το ενεργό socket που περιγράφεται από το πρώτο όρισμα.

Η accept αναστέλλει τη λειτουργία της μέχρι να φτάσει αίτημα σύνδεσης με κλήση της connect από κάποιο πελάτη.

Όταν λάβει αυτό ένα τέτοιο αίτημα:

Δημιουργεί ένα νέο active socket που συνομιλεί με το active socket του πελάτη (που είναι διαφορετικό από το αρχικό passive socket που χρησιμοποιήθηκε ως όρισμα στη listen).

Επιστρέφει τον file descriptor που αναφέρεται στο νέο αυτό socket.

Υποδοχείς

SOCKET PROGRAMMING

Ανταλλαγή μηνυμάτων στα Unix sockets

Στην περίπτωση κατά την οποία χρησιμοποιούνται για την επικοινωνία Unix sockets, η αποστολή και η ανταλλαγή δεδομένων πραγματοποιούνται με τις συναρτήσεις read και write αλλά και τις

int sendmsg (int fd, struct msghdr * msg, unsigned int flags);

int recvmsg (int fd, struct msghdr * msg, unsigned int flags);

όπου fd ο περιγραφέας αρχείου μέσω του οποίου αποστέλλεται ή παραλαμβάνεται η πληροφορία, msghdr ειδική δομή δεδομένων που περιγράφει το μήνυμα, ενώ η τιμή του flags είναι συνήθως 0.

```
struct msghdr
        {
        void            *msg_name;      // optional address
        socklen_t       msg_namelen;    // size of address
        struct iovec    *msg_iov;       // scatter/gather array
        int             msg_iovlen;     // no. of members
        void            *msg_control;   // ancillary data buffer
        int             msg_controllen; // ancillary buffer length
        int             flags;          // flags on received message
        };
```

Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης Unix socket A

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.

SERVER

```c
int main(void) {
    struct sockaddr_un address;
    int sock, conn;
    socklen_t addrLength;
    if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
         perror("Message from socket:"); exit(1); }
    unlink("./sample-socket");
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "./sample-socket");
    addrLength = sizeof(address.sun_family) + strlen(address.sun_path);

    if (bind(sock, (struct sockaddr *) &address, addrLength)){
        perror("Message from bind:"); exit(1); }
    if (listen(sock, 5)){
        perror("Message from listen:"); exit(1); }
    while ((conn = accept(sock, (struct sockaddr *) &address,
                          &addrLength)) >= 0) {
        printf("---- getting data\n");
        copyData(conn, 1);
        printf("---- done\n");
        close(conn); }
    if (conn < 0) {
        perror("Message from accept:"); exit(1); }
    close(sock);
    return 0; }
```

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

Η συνάρτηση copyData αντιγράφει δεδομένα από τον περιγραφέα from στον περιγραφέα to μέχρι να μην υπάρχει τίποτε άλλο να αντιγραφεί.

```c
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); } }
```

Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης Unix socket A

CLIENT

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

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); } }

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; }
```

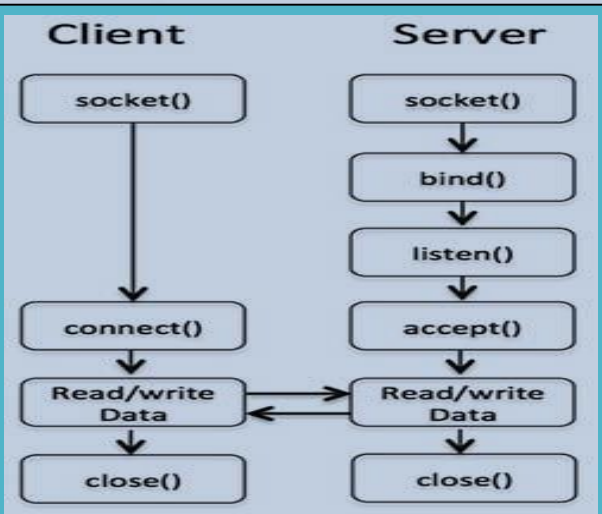
Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης Unix socket A

CLIENT

```
int main(void) {
    struct sockaddr_un address;
    int sock;
    socklen_t addrLength;
    if ((sock = socket(PF_UNIX, SOCK_
         perror("Message from socket:"
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "./sample
    addrLength = sizeof(address.sun_fa
    if (connect(sock, (struct sockaddr
         perror("Message from connect:
    copyData(0, sock);
    close(sock);
    return 0; }
```

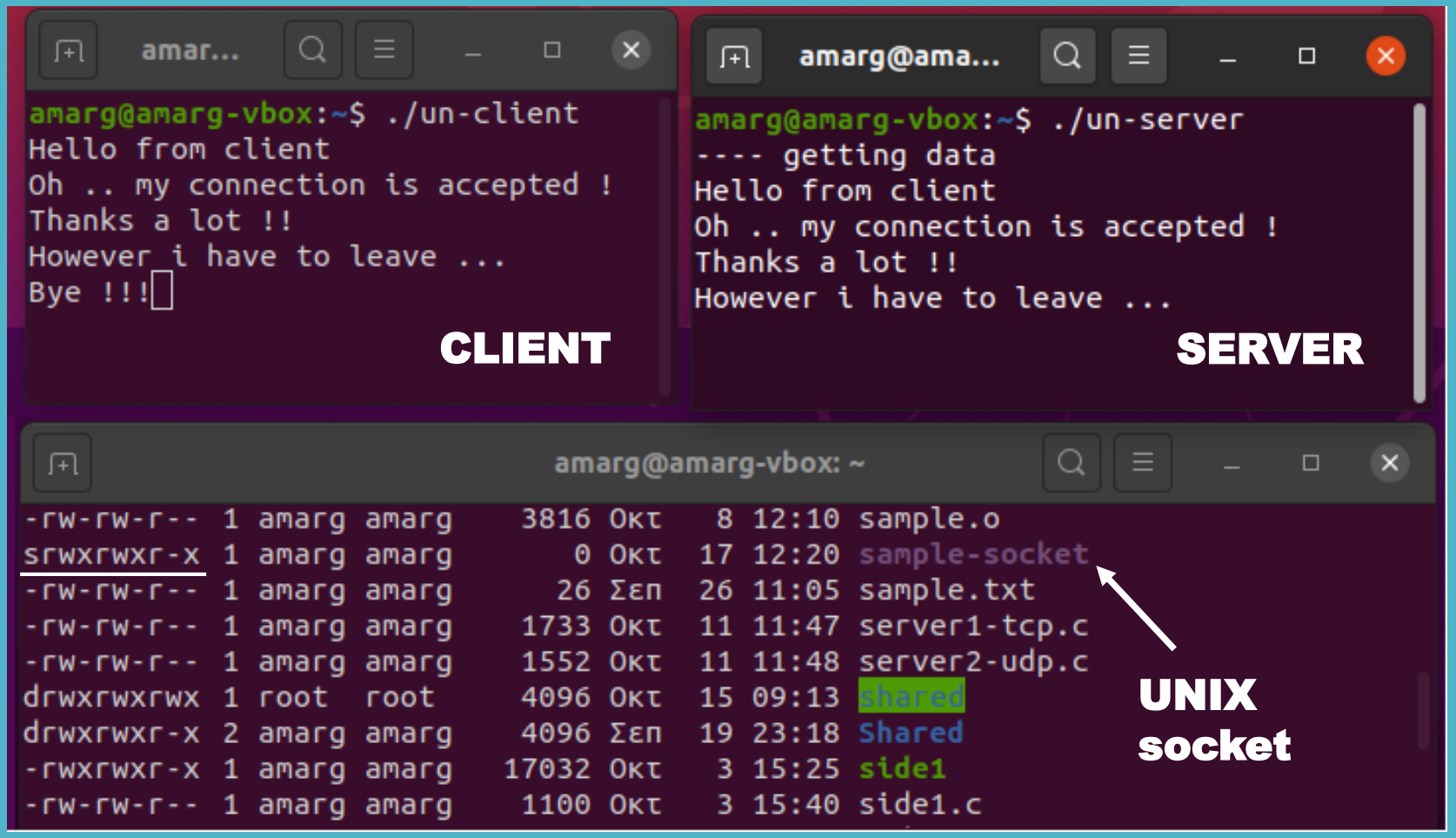


SERVER

```
int main(void) {
    struct sockaddr_un address;
    int sock, conn;
    socklen_t addrLength;
    if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
         perror("Message from socket:"); exit(1); }
    unlink("./sample-socket");
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "./sample-socket");
    addrLength = sizeof(address.sun_family) + strlen(address.sun_path);

    if (bind(sock, (struct sockaddr *) &address, addrLength)){
        perror("Message from bind:"); exit(1); }
    if (listen(sock, 5)){
        perror("Message from listen:"); exit(1); }
    while ((conn = accept(sock, (struct sockaddr *) &address,
                          &addrLength)) >= 0) {
        printf("---- getting data\n");
        copyData(conn, 1);
        printf("---- done\n");
        close(conn); }
    if (conn < 0) {
        perror("Message from accept:"); exit(1); }
    close(sock);
    return 0; }
```

Υποδοχείς

SOCKET PROGRAMMING



Το UNIX socket αποτελεί ένα αρχείο του σκληρού δίσκου !!

Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης Unix socket B

Ο server και ο client συνδέονται με μία σχέση πατέρα (server) – παιδιού (client) που δημιουργείται μέσω της fork και επικοινωνούν στέλνοντας ένα μήνυμα ο ένας στον άλλο.

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

#define SOCKETNAME "MySocket"

int main(void) {
    struct sockaddr_un sa;
    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    int fd_skt, fd_client; char buf[100];
    int written; ssize_t readb;
```

CLIENT

```c
if (fork() == 0) {
    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);
    while (connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) == -1) {
        if (errno == ENOENT) {
            sleep(1); continue; }
        else {
         perror("Message from connect [client]");
         exit(-2); }}
    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);
    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);
    close(fd_skt);
    exit(0); }
```

Υποδοχείς

SOCKET PROGRAMMING

SERVER

```
else {
    if ((fd_skt = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
         perror("Message from socket [server]");
    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");
    if ((fd_client = accept(fd_skt, NULL, 0))<0) {
        perror("Message from accept [server]");
        exit(-4); }
    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);
    printf ("[Server] ==> let us write to client\n");
    if (write(fd_client, "Goodbye!", 9)==-1) {
         perror("Message from write [server]");
         exit(-6); }
    close(fd_skt);
    close(fd_client);
    exit(0); }}
```

Ο server και ο client συνδέονται με μία σχέση πατέρα (server) – παιδιού (client) που δημιουργείται μέσω της fork και επικοινωνούν στέλνοντας ένα μήνυμα ο ένας στον άλλο.

```
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
[Client] ==> Connection has been established .. let us write to server
[Server] ==> let us read from client via socket 4
[Client] ==> 7 bytes written to server
Server got Hello! ==> 7 bytes read from client
[Server] ==> let us write to client
amarg@amarg-vbox:~$ [Client] ==> Now let us read from server
[Client] ==> 9 bytes read from server
Client got Goodbye!
```

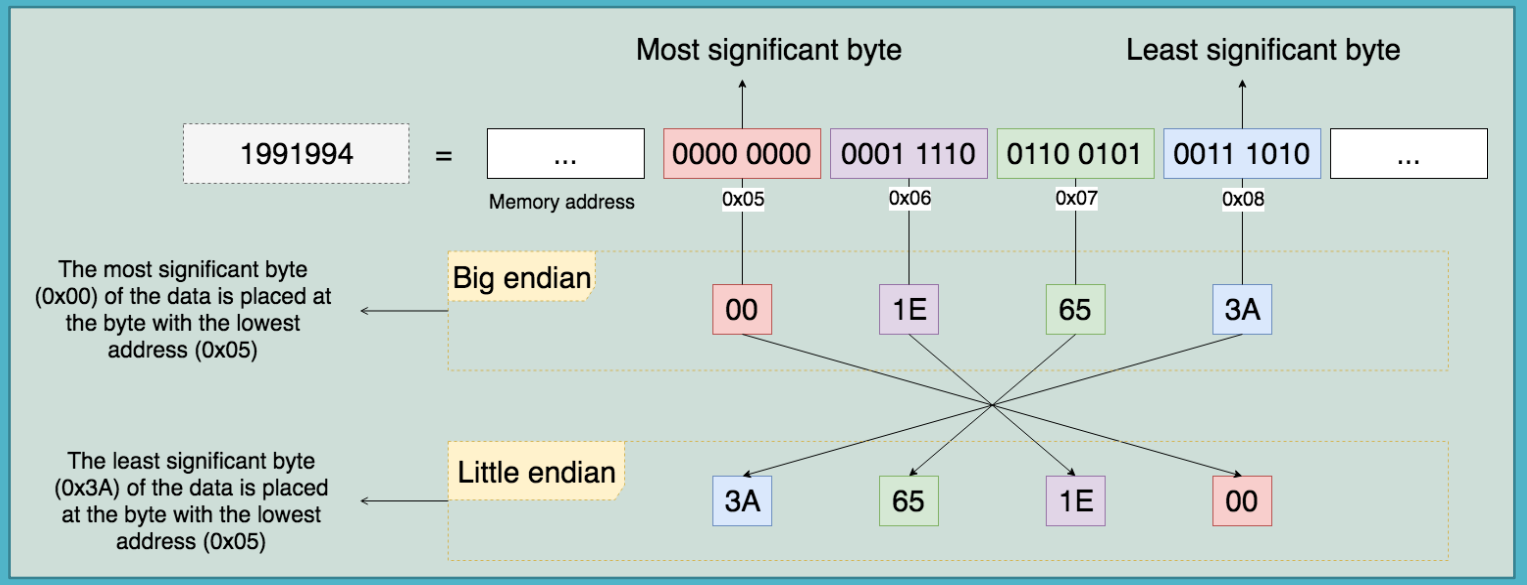
Υποδοχείς

SOCKET PROGRAMMING

Χρησιμοποιώντας TCP / IP sockets

Το βασικό χαρακτηριστικό των δικτύων που χρησιμοποιούν το πρωτόκολλο TCP / IP είναι πως αποτελούν ετερογενή δίκτυα, υπό την έννοια πως οι υπολογιστές που επικοινωνούν μέσα από αυτά μπορεί να διαφέρουν τόσο ως προς την αρχιτεκτονική όσο και ως προς το λειτουργικό σύστημα.

Μία βασική διαφορά εντοπίζεται στον τρόπο αποθήκευσης των bytes ενός αριθμού των 32 bits (4 bytes). Οι δύο βασικοί τρόποι αποθήκευσης είναι ο big endian (80x86) και ο little endian (SUN SPARC, Motorola, Power PC) που αποθηκεύουν τα bytes με την αντίστροφη σειρά και δεν είναι συμβατοί μεταξύ τους.



Υποδοχείς

SOCKET PROGRAMMING

Χρησιμοποιώντας TCP / IP sockets

Σε περιπτώσεις κατά τις οποίες οι υπολογιστές που επικοινωνούν χρησιμοποιούν διαφορετικό τρόπο αναπαράστασης (δηλαδή ο ένας χρησιμοποιεί big-endian [κωδικοποίηση πού είναι γνωστή και ως network byte order] και ο άλλος little-endian) τα δικτυακά πρωτόκολλα είναι υπεύθυνα για την πραγματοποίηση των κατάλληλων μετασχηματισμών δεδομένων έτσι ώστε οι δύο υπολογιστές να είναι σε θέση να επικοινωνήσουν μεταξύ τους.

Η μετατροπή από τον έναν τύπο κωδικοποίησης στον άλλο γίνεται από τις συναρτήσεις της γλώσσας C με ονόματα htonl, htons, ntohl και ntohs (netinet/in.h) και με ορίσματα αριθμούς μήκους 32 bits

```
unsigned int htonl (unsigned int hostlong);
unsigned short htons (unsigned short hostshort);
unsigned int ntohl (unsigned int netlong);
unsigned short ntohs (unsigned short netshort);
```

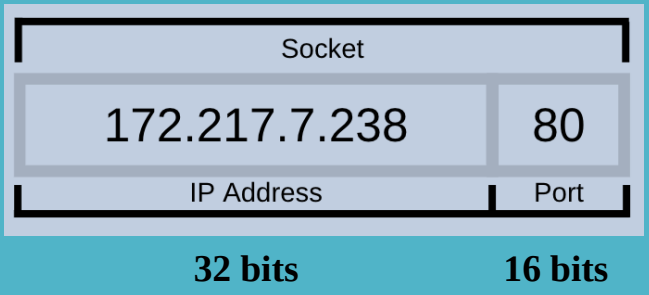
htonl → host to network, long
htons → host to network, short

ntohl → network to host, long
ntohs → network to host, short

Υποδοχείς

SOCKET PROGRAMMING

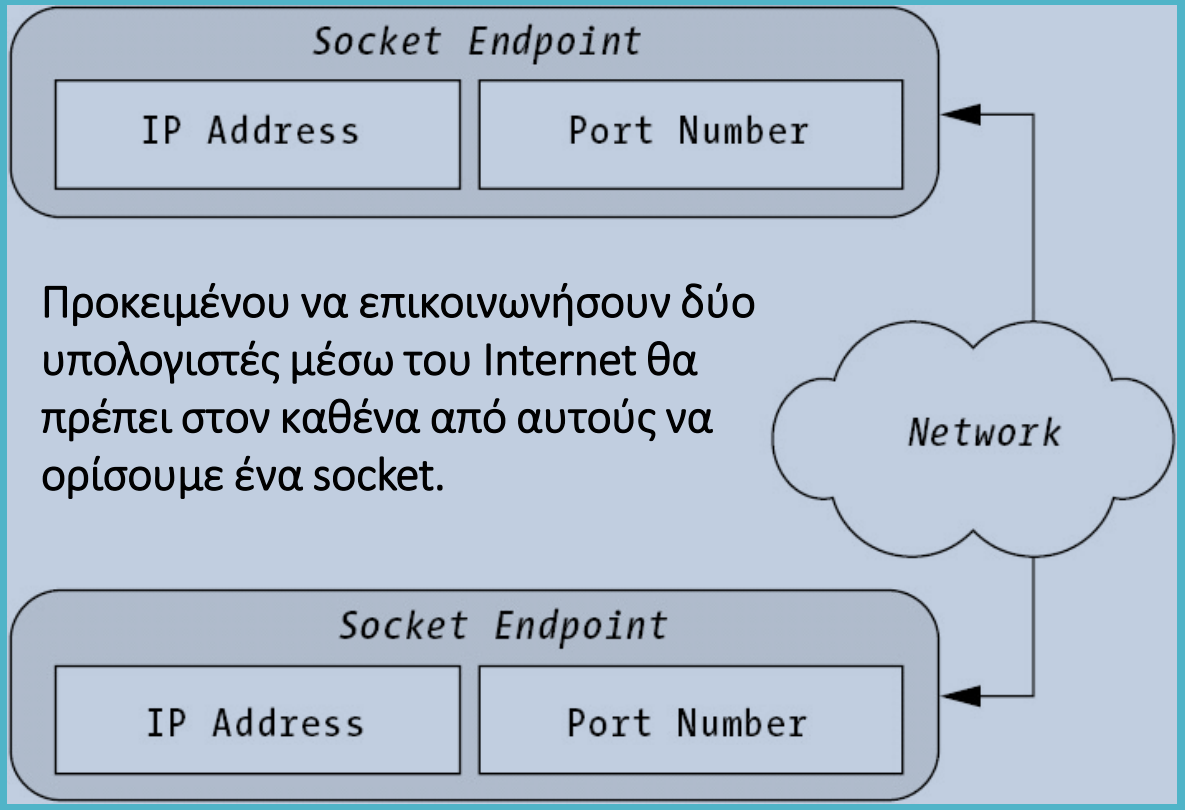
Ένας υπολογιστής του διαδικτύου που προσφέρει κάποια υπηρεσία, προσδιορίζεται πλήρως από το συνδυασμό IP Address : Port Number ο οποίος είναι γνωστός ως TCP / IP socket.



Δομή για TCP / IP socket

```
struct sockaddr_in {
        short           sin_family;
        unsigned short  sin_port;
        struct in_addr  sin_addr;  };
```

sin_family → AF_INET
port → port number
sin_addr → IP Address



Προκειμένου να επικοινωνήσουν δύο υπολογιστές μέσω του Internet θα πρέπει στον καθένα από αυτούς να ορίσουμε ένα socket.

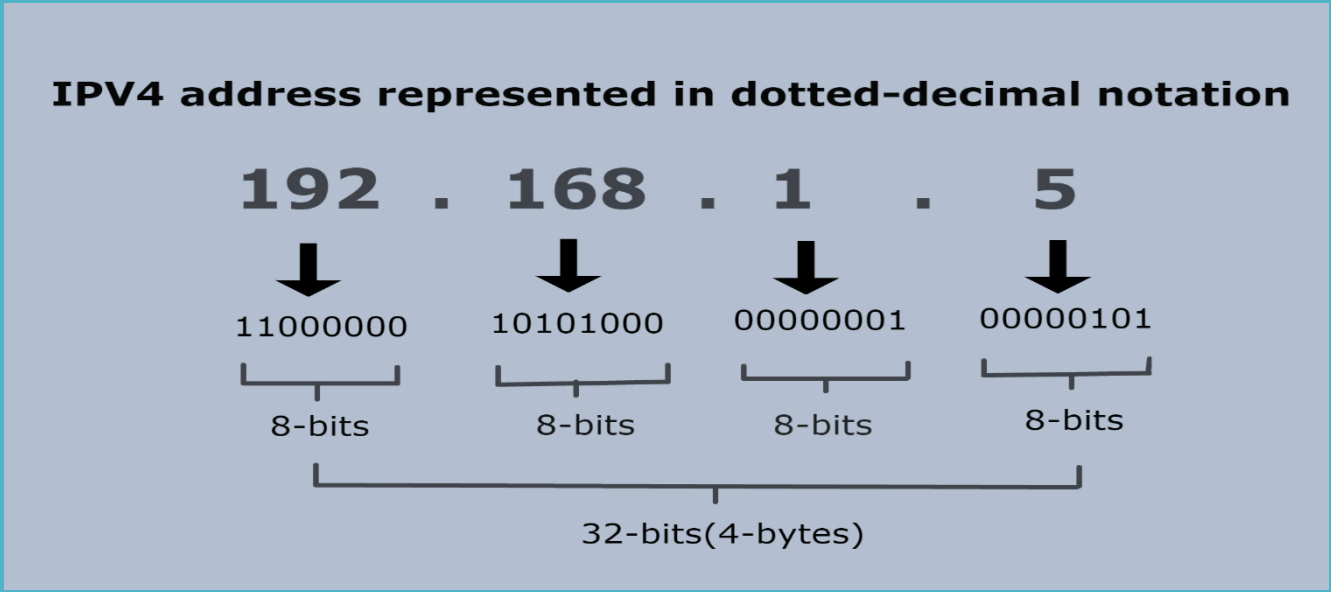
Υποδοχείς

SOCKET PROGRAMMING

Μία διεύθυνση IP μπορεί να εμφανιστεί σε δύο διαφορετικές μορφές και πιο συγκεκριμένα:

Εστιγμένη δεκαδική αναπαράσταση (dotted decimal notation) → η διεύθυνση IP αποτελείται από τέσσερις αριθμούς με τιμές ανάμεσα στο 0 και στο 255 οι οποίοι είναι χωρισμένοι με τελεία.

Δυαδική αναπαράσταση (binary notation) → προκύπτει από τη μετατροπή της παραπάνω αναπαράστασης στο δυαδικό σύστημα. Οι τέσσερις αριθμοί μετατρέπονται στο δυαδικό σύστημα με μήκος 8 bits και στη συνέχεια συνενώνονται. Οι τελείες δεν λαμβάνονται υπόψη.



Υποδοχείς

SOCKET PROGRAMMING

Μετατροπή IP διεύθυνσης από δυαδικό συμβολισμό σε εστιγμένο δεκαδικό συμβολισμό.

char * inet_ntoa (struct in_addr address)

Μετατροπή IP διεύθυνσης από εστιγμένο δεκαδικό συμβολισμό σε δυαδικό συμβολισμό.

unsigned long int inet_addr (const char * ddaddress);

Αν και αυτή η συνάρτηση ήταν ιστορικά η πρώτη που χρησιμοποιήθηκε για αυτό το σκοπό, ωστόσο αποδείχθηκε προβληματική και για το λόγο αυτό αντικαταστάθηκε από την πιο κατάλληλη συνάρτηση

unsigned long int inet_atton
(const char * ddaddress, struct in_addr * address);

Για τη χρήση των παραπάνω συναρτήσεων θα πρέπει να γίνουν include τα header files

<netinet/in.h> και <arpa/inet.h>

Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης TCP / IP socket (Πρωτόκολλο TCP)

SERVER

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

#define PORT 8080
```

```
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";
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE); }
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
                   &opt, sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE); }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    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; }
```

Ο client αποστέλλει στον server ένα μήνυμα που παραλαμβάνεται και εκτυπώνεται από αυτόν.

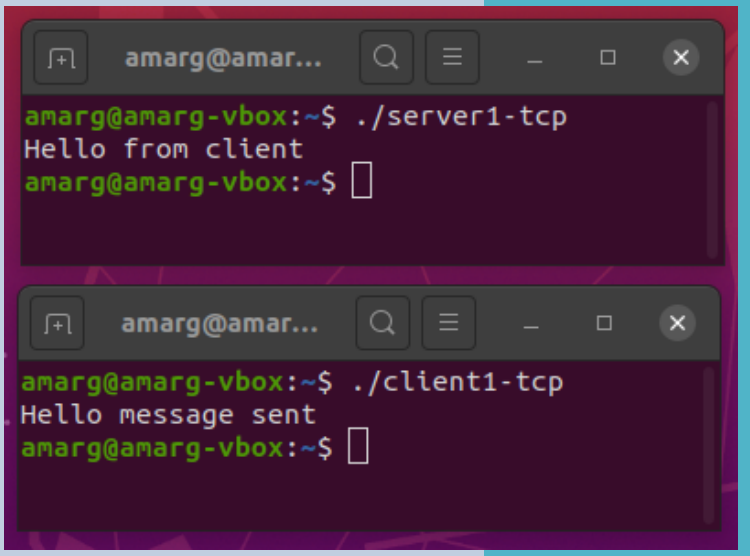
Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης TCP / IP socket (Πρωτόκολλο 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);
    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; }
```

CLIENT



Output

Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης TCP / IP socket (Πρωτόκολλο TCP)

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.

```c
int main(int argc, char * argv[]) {
    struct sockaddr_in sa;
    int fd_skt;
    char buf[1000];
    ssize_t nread;
    if (argc!=2) {
       printf ("Usage: inetExample xxx.xxx.xxx.xxx\n");
       exit (-1); }
    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr =  inet_addr(argv[1]);
    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; }
    write(fd_skt, REQUEST, strlen(REQUEST));
    nread = read(fd_skt, buf, sizeof(buf));
    (void) write(STDOUT_FILENO, buf, nread);
    close(fd_skt);
    exit(0); }
```

port 80 (http)

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

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

Το πρωτόκολλο HTTP (HyperText Transfer Protocol) απαιτεί τη χρήση της θύρας 80 για την εξυπηρέτηση αιτημάτων σύνδεσης από το διακομιστή.

Υποδοχείς

SOCKET PROGRAMMING

Παράδειγμα χρήσης TCP / IP socket (Πρωτόκολλο TCP)

Αρχικά καλούμε την εντολή ping για να ανακτήσουμε τη διεύθυνση IP του δικτυακού τόπου στον οποίο επιθυμούμε να συνδεθούμε (π.χ. www.google.com)

```
amarg@amarg-vbox:~$ ping www.google.com
PING www.google.com (172.217.19.100) 56(84) bytes of data.
64 bytes from bud02s27-in-f4.1e100.net (172.217.19.100): icmp_seq=1 ttl=111 time=68.8 ms
```

και στη συνέχεια χρησιμοποιούμε αυτή τη διεύθυνση (172.217.19.100) ως όρισμα στην εφαρμογή μας.

```
amarg@amarg-vbox:~$ ./inetExample 172.217.19.100
HTTP/1.0 200 OK
Date: Sun, 18 Oct 2020 08:08:43 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=204=tlcVs7M6fZxpWYf0Hs3wj7YRWWnVBvVL7Zd4PhlEW5TGt_ytDx8dhf6AMv7QyDDM9RNDgDFZ91kR03PKBRf3
E6jcKM37W39p7uX_TqQmtUdfqW6t1z379yf2ohRE5XUxzHxPsEDJ9dqGfSVAuzhPzIZzj4qd5itapKsmW2LaHaU; expires=Mon, 19
-Apr-2021 08:08:43 GMT; path=/; domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding

<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="el"><head><meta content="te
xt/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/googleg/1x/googleg_sta
ndard_color_128dp.png" itemprop="image"><title>Google</title><script nonce="ecDPX8dwpviLT4NfLtA7Fg==">(f
unction(){window.google={kEI:'C_iLX4uhFI2oa_2KiYAJ',kEXPI:'0,18168,183994,1151585,5662,731,22
```

Υποδοχείς

ΑΝΑΚΤΗΣΗ ΟΝΟΜΑΤΟΣ ΚΕΝΤΡΙΚΟΥ ΥΠΟΛΟΓΙΣΤΗ (HOSTNAME)

Η χρήση διευθύνσεων IP των 32 bits είναι ακατάλληλη για τους ανθρώπους οι οποίοι προτιμούν να χρησιμοποιούν ένα απλό όνομα υπολογιστή (hostname). Η μετατροπή ενός hostname στην διεύθυνση IP του υπολογιστή και αντίστροφα, προσφέρεται από την υπηρεσία DNS (Domain Name Service).

Η ανάκτηση του hostname του τοπικού υπολογιστή γίνεται καλώντας την εντολή hostname ή εκτυπώνοντας την τιμή της μεταβλητής περιβάλλοντος HOSTNAME

```
amarg@amarg-vbox:~$ hostname
amarg-vbox
amarg@amarg-vbox:~$ echo $HOSTNAME
amarg-vbox
```

Ένας τρόπος για να δούμε στην πράξη την υπηρεσία DNS είναι να εκτελέσουμε την εντολή ping η οποία χρησιμοποιεί το πρωτόκολλο ICMP για να ελέγξει εάν ένας υπολογιστής είναι προσπελάσιμος ή όχι.

```
amarg@amarg-vbox:~$ ping www.google.com
PING www.google.com (172.217.169.132) 56(84) bytes of data.
64 bytes from sof02s32-in-f4.1e100.net (172.217.169.132): icmp_seq=1 ttl=111 time=87.2 ms
64 bytes from sof02s32-in-f4.1e100.net (172.217.169.132): icmp_seq=2 ttl=111 time=86.7 ms
64 bytes from sof02s32-in-f4.1e100.net (172.217.169.132): icmp_seq=3 ttl=111 time=89.1 ms
```

Η εντολή ping ανατρέχοντας στον DNS server που είναι δηλωμένος στο σύστημα, αναφέρει πως η διεύθυνση IP του υπολογιστή με hostname www.google.com είναι η 172.217.169.132.

Υποδοχείς

ΑΝΑΚΤΗΣΗ ΟΝΟΜΑΤΟΣ ΚΕΝΤΡΙΚΟΥ ΥΠΟΛΟΓΙΣΤΗ (HOSTNAME)

Σε ένα hostname μπορούν να αντιστοιχούν περισσότερες από μία διευθύνσεις IP και αντίστοιχα, σε μία διεύθυνση IP μπορεί να αντιστοιχούν περισσότερα από ένα hostnames.

```
struct hostent {
   char* h_name;         /* official name of host */
   char** h_aliases;     /* NULL-terminated alias list */
   int h_addrtype        /* address type (AF_INET) */
   int h_length;         /* length of addresses (4B) */
   char** h_addr_list; /* NULL-terminated address list */
};
```

Σε επίπεδο κώδικα η μετατροπή **hostname ↔ IP address** πραγματοποιείται από τις συναρτήσεις

struct hostent * gethostbyname (const char * name);

struct hostent * gethostbyaddr (const char * addr, int len, int type);

εκ των οποίων η πρώτη δέχεται ένα hostname ενώ η δεύτερη μία IP address. Αμφότερες οι συναρτήσεις επιστρέφουν έναν δείκτη σε μία δομή hostent συμπληρωμένη με τις κατάλληλες πληροφορίες.

Υποδοχείς

Αριθμοί θύρας (port numbers)

Κατά τη σύνδεση ενός χρήστη σε έναν υπολογιστή που αντιστοιχεί σε μία συγκεκριμένη διεύθυνση IP, το είδος της εξυπηρέτησης που θα πραγματοποιηθεί, προσδιορίζεται από τον αριθμό θύρας. Γνωστοί αριθμοί θύρας είναι οι 21 (ftp), 22 (ssh), 23 (telnet), 25 (email) και 80 (www).

Στο λειτουργικό σύστημα Linux η αντιστοίχιση των δικτυακών πρωτοκόλλων στους διάφορους αριθμούς θύρας περιλαμβάνεται στο αρχείο /etc/services.

```
amarg@amarg-vbox:~$ cat /etc/services
tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
whois           43/tcp          nicname
tacacs          49/tcp                          # Login Host Protocol (TACACS)
tacacs          49/udp
```

Υποδοχείς

Αριθμοί θύρας (port numbers)

Η ανάκτηση των σχετικών πληροφοριών γίνεται με τη συνάρτηση getservbyname του αρχείου netdb.h

struct servent * getservbyname (const char * name, const char * protocol);

όπου name το όνομα της υπηρεσίας για την οποία η εφαρμογή χρειάζεται πληροφορίες και protocol το πρωτόκολλο που θα χρησιμοποιηθεί.

Η getservbyname επιστρέφει έναν δείκτη σε μία δομή servent που ορίζεται ως

```
struct servent  {
        char *s_name;           /* service name */
        char **s_aliases;       /* pointer to alternate service name array */
        int s_port;             /* port number */
        char *s_proto;          /* name of protocol */
};
```

Η κάθε υπηρεσία μπορεί να έχει πολλά ονόματα αλλά έναν και μοναδικό αριθμό θύρας.

Υποδοχείς

Η συνάρτηση getaddrinfo

Εάν ο χρήστης επιθυμεί να συνδεθεί σε ένα υπολογιστή με συγκεκριμένο όνομα (nodename ή hostname) και να χρησιμοποιήσει μία συγκεκριμένη υπηρεσία (service), μπορεί να κατασκευάσει εύκολα μία socket address που θα χρησιμοποιηθεί ως όρισμα στις συναρτήσεις bind και connect καλώντας τη συνάρτηση getaddrinfo ως

```
int getaddrinfo (const char * node, const char * service,
                 const struct addrinfo * hints, struct addrinfo ** res);
```

όπου node του υπολογιστή (που μπορεί να αναζητηθεί σε DNS Server, στο αρχείο /etc/hosts ή να είναι κάποια διεύθυνση IPv4 ή IPv6), service το όνομα της υπηρεσίας και hints ένας δείκτης σε δομή addrinfo που ορίζει κριτήρια για την επιλογή των δομών socket address που επιστρέφονται στη λίστα res.

```
struct addrinfo {
  int          ai_flags;
  int          ai_family;
  int          socktype;
  int          ai_protocol;
  size_t       ai_addrlen;
  struct sockaddr * ai_addr;
  char *       ai_cannonname;
  struct addrinfo * ai_next; }
```

Με την ολοκλήρωση της σύνδεσης απαιτείται αποδέσμευση μνήμης.

```
void freeaddrinfo
     (struct addrinfo * res)
```

Εάν ανακύψει σφάλμα ανακτάται καλώντας τη συνάρτηση

```
const char * gai_strerror
     (int code);
```

Υποδοχείς

Παράδειγμα χρήσης της συνάρτησης getaddrinfo

```
// Source: https://basepath.com/aup/ex/adi_8c-source.html
// Rochkind's Book, page 574

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

int main (int argc, char * argv[]) {
    int retVal; char err[50];
    struct addrinfo *infop = NULL, hint;
    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    if (argc!=2) {
       printf ("Usage: inetExample xxx.xxx.xxx.xxx\n");
       exit (-1); }
    retVal = getaddrinfo(argv[1], "80", &hint, &infop);
    if (retVal!=0) {
        strcpy (err,gai_strerror(retVal));
        printf ("Error found ==> %s\n", err);
        exit (-1); }
    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);}
```

```
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:~$
```

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.

Υποδοχείς

Ασυνδεσμικές επικοινωνίες (SOCKET_DGRAM)

Δεν χρησιμοποιείται σύνδεση οπότε δεν χρησιμοποιούνται οι listen και accept.

Ο αποστολέας καθορίζει τη διεύθυνση στην οποία επιθυμεί να στείλει ένα πακέτο. Η διεύθυνση αποστολής προστίθεται στο πακέτο το οποίο αποστέλλεται χωρίς να ελέγχεται και να υπάρχει εγγύηση πως αυτό έφτασε σωστά στον παραλήπτη.

Ο παραλήπτης καθορίζει τη διεύθυνση από την οποία επιθυμεί να παραλάβει ή παραλαμβάνει από όλους και ενημερώνεται για τη διεύθυνση του αποστολέα.

Συνήθως δεν χρησιμοποιούνται αρχιτεκτονικές client – server αλλά όλοι οι σταθμοί είναι ομότιμοι μεταξύ τους (peer to peer).

```
ssize_t recvfrom(int sockfd, void *buff, size_t nbyte, int flag,
                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbyte, int flag,
               const struct sockaddr *to, socklen_t addrlen);
```

Υποδοχείς

Παράδειγμα ασυνδεσμικής επικοινωνίας

SERVER

```c
int main() {
    int sockfd, len, n;
    char buffer[MAXLINE];
    char *hello = "Hello from server";
    struct sockaddr_in servaddr, cliaddr;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE); }
    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); }
    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; }
```

```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
```

Ανταλλαγή ενός μηνύματος μεταξύ σταθμών οι οποίοι συνήθως είναι ομότιμοι.

Υποδοχείς

Παράδειγμα ασυνδεσμικής επικοινωνίας

```c
int main() {
    int sockfd, len, n;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in servaddr;
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE); }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;
    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; }
```

CLIENT

```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
```

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

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

Output

Υποδοχείς

Σύνδεση πολλών client σε έναν server

Στην περίπτωση κατά την οποία ένας server δέχεται αιτήσεις εξυπηρέτησης από πολλούς clients, ανακύπτουν ζητήματα που σχετίζονται με τη διαχείριση τερματικών (terminals).

Αυτά σχετίζονται με τα αρχεία συστήματος /dev/tty για τα οποία χρησιμοποιούνται οι γνωστές κλήσεις συστήματος open, close, read και write – ωστόσο μπορούν να χρησιμοποιηθούν και τα γνωστά αρχεία stdin, stdout και stderr.

Κάθε προσπάθεια ανοίγματος ενός τερματικού μπλοκάρεται (αναστέλλει τη λειτουργία της και τίθεται σε αναμονή) μέχρι η συσκευή να συνδεθεί σε κάποιο τερματικό, εκτός εάν χρησιμοποιηθεί το flag O_NONBLOCK.

ΣΥΝΔΕΣΗ ΧΡΗΣΤΗ ΣΤΟ ΣΥΣΤΗΜΑ → Η open επιστρέφει μη μηδενική τιμή στη διεργασία που επιχειρεί τη σύνδεση – στη συνέχεια εκτελείται το login process για την είσοδο του χρήστη και στο τέλος καλείται το login shell.

Η write είναι σχετικά απλή → όσοι χαρακτήρες προωθούνται για έξοδο, μπαίνουν σε μία σειρά προκειμένου να αποσταλούν στο τερματικό.

Η close λειτουργεί ακριβώς όπως στα αρχεία.

Υποδοχείς

Σύνδεση πολλών client σε έναν server

Ωστόσο, η read λειτουργεί διαφορετικά! ΔΕΝ επιστρέφει, παρά μόνο όταν ολοκληρωθεί η γραμμή, δηλαδή όταν ο χρήστης πατήσει το ENTER και αυτό, επειδή ο χρήστης μπορεί να αναθεωρήσει (←) . Επίσης, επιστρέφει μία μόνο γραμμή κάθε φορά.

Με άλλα λόγια, η συνάρτηση μπλοκάρει ... ωστόσο, στην περίπτωση κατά την οποία συνδέονται στον server πολλά τερματικά (π.χ μετρητικές διατάξεις) είναι επιθυμητή η κατάργηση του μπλοκαρίσματος.

Για ποιο λόγο συμβαίνει αυτό?

Διότι μπορούμε να περιμένουμε άσκοπα από ένα τερματικό να στείλει δεδομένα χωρίς αυτό να έχει κάτι να στείλει και ταυτόχρονα να αγνοούμε άλλα τερματικά που έχουν δεδομένα για αποστολή και τα οποία δεν μπορούν να στείλουν.

Αναζητούμε μία συνάρτηση η οποία να παραμένει αδρανής μέχρι να εμφανιστεί διαθέσιμος χαρακτήρας εισόδου για οποιονδήποτε περιγραφέα αρχείου. Αυτή η συνάρτηση είναι η

```
int select (int nfds, fd_set * readfds, fd_set * writefds,
                      fd_set * errorfds, struct timeval * timeout);
```

και επιτρέπει την ταυτόχρονη παρακολούθηση πολλών περιγραφέων αρχείων αναμένοντας μέχρι ένας ή περισσότερους από αυτούς να γίνει έτοιμος για την πραγματοποίηση κάποιας λειτουργίας I/O.

Υποδοχείς

Σύνδεση πολλών client σε έναν server

Τα ορίσματα readfds, writefds και errorfds είναι δείκτες σε σύνολα περιγραφέων αρχείων που ορίζουν ποιοι περιγραφείς αρχείων θα παρακολουθεί η συνάρτηση.

Το όρισμα nfds συνήθως είναι ίσο με max {fds}+1 όπου max {fds} το μέγιστο πλήθος περιγραφέων που παρακολουθεί η select.

Το όρισμα timeout είναι δείκτης σε μία δομή timeval που ορίζεται ως

```
struct timeval{
        int tv_sec;   /* seconds*/
        int tv_usec; /* microseconds*/
        };
```

και καθορίζει το χρονικό διάστημα που θα περιμένει η select προκειμένου να συμβεί κάτι. Εάν η τιμή αυτής της παραμέτρου είναι NULL η select θα διακοπεί μέχρι την εκδήλωση κάποιου συμβάντος.

Η select επιστρέφει το συνολικό πλήθος των στοιχείων που εξετάζονται στα τρία σύνολα περιγραφέων αρχείου, την τιμή 0 εάν έχει λήξει ο χρόνος για την κλήση και την τιμή -1 εάν έχει εκδηλωθεί σφάλμα.

Υποδοχείς

Σύνδεση πολλών client σε έναν server

Η select διαχειρίζεται τα σύνολα περιγραφέων αρχείων μέσω των επόμενων μακροεντολών.

FD_ZERO (fd_set * fds)

Αρχικοποιεί το σύνολο fd_set απομακρύνοντας από αυτό όλους τους περιγραφείς αρχείων.

FD_SET (int fd, fd_set * fds)

Προσθέτει τον περιγραφέα αρχείου fd στο σύνολο fds. Για παράδειγμα, μετά την κλήση της FD_ZERO εάν θέλουμε να εντάξουμε στο σύνολο fds τους περιγραφείς fd1 και fd2 θα εκτελέσουμε τις εντολές

```
FD_SET (fd1, &set);          FD_SET (df2, &set);
```

FD_CLR (int fd, fd_set * fds)

Απομακρύνει τον περιγραφέα αρχείου fd από το σύνολο fds.

FD_ISSET (int fd, fd_set * fds)

Ελέγχει εάν ο περιγραφέας αρχείου fd ανήκει στο σύνολο fds.

Υποδοχείς

Σύνδεση πολλών client σε έναν server

Έστω πως μία διεργασία θέλει να διαβάσει δεδομένα από 3 sockets, από μία διάταξη SCSI και από ένα αρχείο και έστω πως οι τιμές των περιγραφέων αρχείου είναι οι 3, 7, 10, 12 και 67. Στην περίπτωση αυτή, το όρισμα readfds έχει τιμή

```
00010001.00101000:00000000.00000000:00000000.00000000:00000000.00000000   00010000.00000000
   ^   ^   ^ ^                                                                 ^
   0   0   1 1                                                                 6
   3   7   0 2                                                                 7
```

FD_SET (fd1, &set); (όπου fd1=03)
FD_SET (fd2, &set); (όπου fd2=07)
FD_SET (fd3, &set); (όπου fd3=10)
FD_SET (fd4, &set); (όπου fd4=12)
FD_SET (fd5, &set); (όπου fd5=67)

Η select τροποποιεί τα σύνολα που δέχεται και προκειμένου να ελέγξουμε εάν ένας περιγραφέας αρχείου είναι ενεργός ή όχι, καλούμε τη μακροεντολή ISSET.

Μία ανταγωνιστική παραλλαγή της select είναι η poll που αντί για μάσκες bits ενημερώνει ολόκληρες δομές , ενώ εκτός από επιλογή πραγματοποιεί και σταθμοσκόμηση (polling).

Υποδοχείς

Σύνδεση πολλών client σε έναν server

Η συνάρτηση run_server μέσα από ένα infinite loop καλεί συνεχώς την accept για να συλλάβει αιτήματα σύνδεσης από τέσσερις διεργασίες – πελάτες, χρησιμοποιώντας τη select. Στη συνέχεια επικοινωνεί με τον καθέναν από τους πελάτες ανταλλάσσοντας με αυτόν ένα μήνυμα.

Οι πελάτες αποτελούν θυγατρικές διεργασίες που δημιουργούνται με κλήση της fork μέσα στην αντίστοιχη συνάρτηση run_client.

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

#define SOCKETNAME "MySocket"
```

SERVER

```c
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;
    printf ("Server pid is %d\n", getpid());
    if ((fd_skt = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n"); return -1; }
    if (bind(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) < 0) {
        perror("Bind"); exit(EXIT_FAILURE); }
    if (listen(fd_skt, SOMAXCONN)<0) {
        perror("Listen"); exit(EXIT_FAILURE); }
    if (fd_skt > fd_hwm) fd_hwm = fd_skt;
    FD_ZERO(&set);
    FD_SET(fd_skt, &set);
    while (1) {
        read_set = set;
        if (select(fd_hwm+1, &read_set, NULL, NULL, NULL)==-1) {
            perror("Select"); exit(EXIT_FAILURE); }
        for (fd = 0; fd <= fd_hwm; fd++)
            if (FD_ISSET(fd, &read_set)) {
                if (fd == fd_skt) {
                    if ((fd_client = accept(fd_skt, NULL, 0))<0) {
                        perror("Select"); exit(EXIT_FAILURE); }
                    FD_SET(fd_client, &set);
                    if (fd_client > fd_hwm) fd_hwm = fd_client; }
                else {
                    if ((nread = read(fd, buf, sizeof(buf)))<0) {
                        perror("Read"); exit(EXIT_FAILURE); }
                    if (nread == 0) {
                        FD_CLR(fd, &set);
                        if (fd == fd_hwm) fd_hwm--;
                        close(fd); }
                    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; }
```

Υποδοχείς

Σύνδεση πολλών client σε έναν server

CLIENT

```
static int run_client(struct sockaddr_un *sap) {
    if (fork() == 0) {
        int fd_skt;
        char buf[100];
        if ((fd_skt = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
             printf("\n Socket creation error \n"); return -1; }
        while (connect(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) == -1) {
            if (errno == ENOENT) {
                sleep(1);
                continue; }
            else {
             perror("Message from connect [client]");
             exit(-2); }}
        snprintf (buf, sizeof(buf), "Hello from %ld!", (long)getpid());
        if (write (fd_skt, buf, strlen(buf)+1) < 0) {
            perror("Write"); exit(EXIT_FAILURE); }
        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; }
```

Output

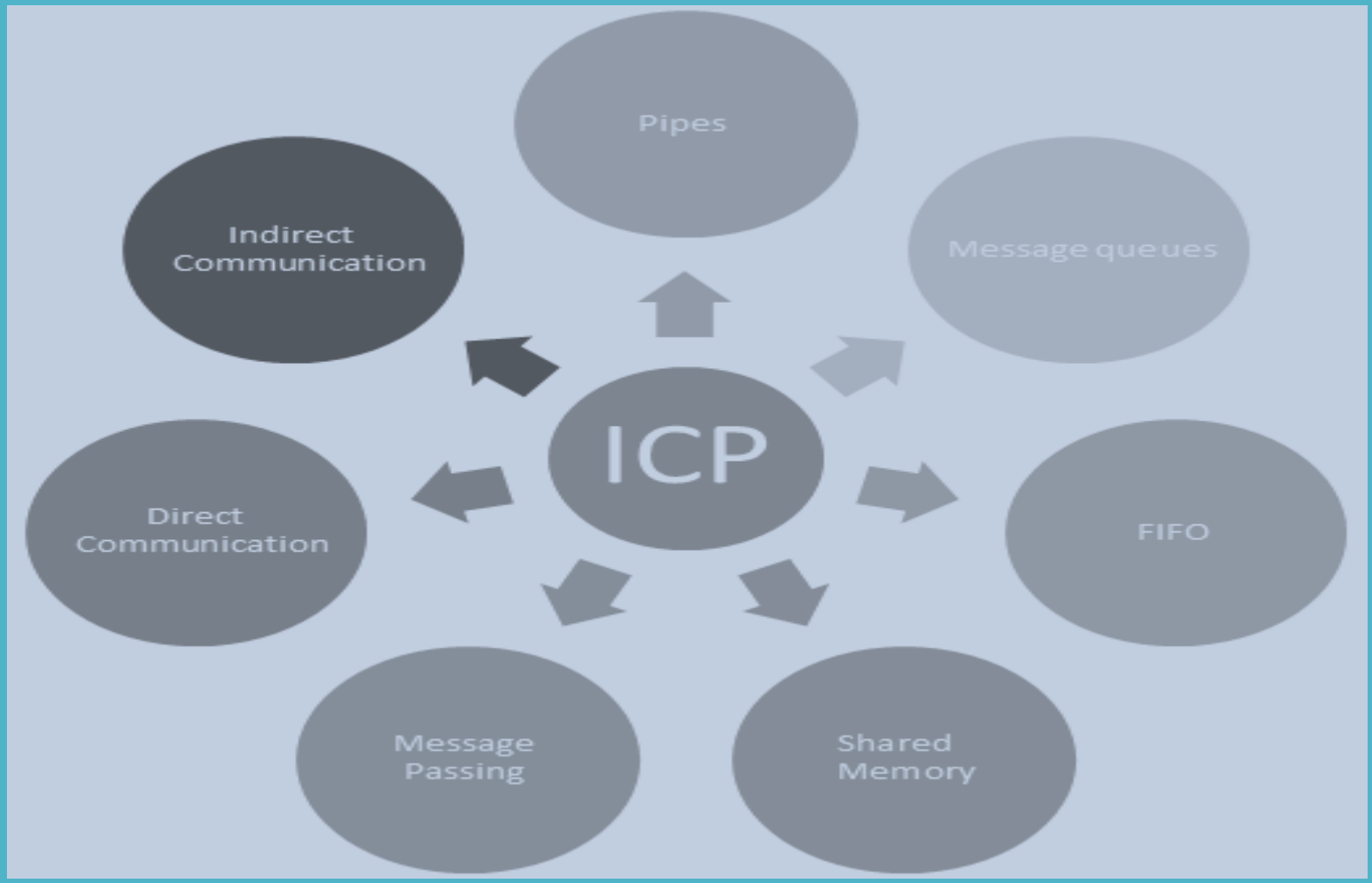
```
Server pid is 3082
Server got "Hello from 3086!"
Server got "Hello from 3085!"
Server got "Hello from 3087!"
Client 3086 got Goodbye!
Client 3086 terminates
Client 3087 got Goodbye!
Client 3085 got Goodbye!
Server got "Hello from 3084!"
Client 3087 terminates
Client 3085 terminates
Client 3084 got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

Main

```
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); }
```

Η συνάρτηση main δημιουργεί τέσσερις πελάτες και στη συνέχεια δημιουργεί έναν διακομιστή για να τους εξυπηρετήσει.

Διαδιεργασιακή επικοινωνία



Διαδιεργασιακή επικοινωνία

Στα μοντέρνα λειτουργικά συστήματα, η διαχείριση πολλαπλών διεργασιών μπορεί να αφορά σε

- Ένα σύστημα απλού επεξεργαστή (πολυπρογραμματισμός)
- Ένα σύστημα πολυεπεξεργαστή (πολυεπεξεργασία)
- Ένα σύστημα κατανεμημένης επεξεργασίας

Η ταυτόχρονη εκτέλεση πολλών διεργασιών οι οποίες μάλιστα μπορούν να αλληλεπιδρούν μεταξύ τους, είναι γνωστή ως ταυτοχρονισμός (concurrency) και σχετίζεται με πολύ σημαντικά ζητήματα σχεδίασης όπως είναι ο διαμοιρασμός πόρων και ο ανταγωνισμός.

Υπάρχουν δύο τρόποι εκκίνησης διεργασιών οι οποίες επικοινωνούν μεταξύ τους:

- Η μία διεργασία ξεκινά τη fork για να ξεκινήσει την άλλη διεργασία (πατέρας - παιδί).
- Η κάθε διεργασία εκτελείται από το δικό της τερματικό (ανεξάρτητες διεργασίες).

Αναγκαία προϋπόθεση για την υποστήριξη του ταυτοχρονισμού είναι ο αμοιβαίος αποκλεισμός, δηλαδή ο αποκλεισμός όλων των υπόλοιπων διεργασιών από μία ροή ενεργειών η οποία εκτελείται σε κάθε χρονική στιγμή από κάποια συγκεκριμένη διεργασία.

Οι μηχανισμοί υποστήριξης ταυτοχρονισμού περιλαμβάνουν μεταξύ άλλων τους σημαφόρους (semaphores), τους παρακολουθητές (watchers) και τη μεταβίβαση μηνυμάτων (message passing).

Διαδιεργασιακή επικοινωνία

Οι πιο σημαντικοί όροι που σχετίζονται με τον ταυτοχρονισμό διεργασιών, είναι οι ακόλουθοι:

- Ατομική λειτουργία ή ατομικότητα: ενέργεια που υλοποιείται ως ακολουθία εντολών οι οποίες εμφανίζονται αδιαίρετες. Αυτή η ακολουθία είτε εκτελείται στο σύνολό της είτε δεν εκτελείται καθόλου.

- Κρίσιμο τμήμα: τμήμα κώδικα που σχετίζεται με τη λειτουργία κοινόχρηστου πόρου και δεν πρέπει να εκτελεστεί τη στιγμή που μία άλλη διεργασία εκτελεί ένα αντίστοιχο τμήμα κώδικα.

- Αδιέξοδο: κατάσταση που ανακύπτει όταν δύο ή περισσότερες διεργασίες δεν μπορούν να προχωρήσουν επειδή η καθεμία από αυτές περιμένει την ολοκλήρωση της λειτουργίας κάποιας άλλης.

- Αλληλεξάρτηση: κατάσταση που προκύπτει όταν δύο ή περισσότερες διεργασίες αλλάζουν συνεχώς τις καταστάσεις τους ως αντίδραση σε μεταβολές που πραγματοποιούνται σε άλλες διεργασίες.

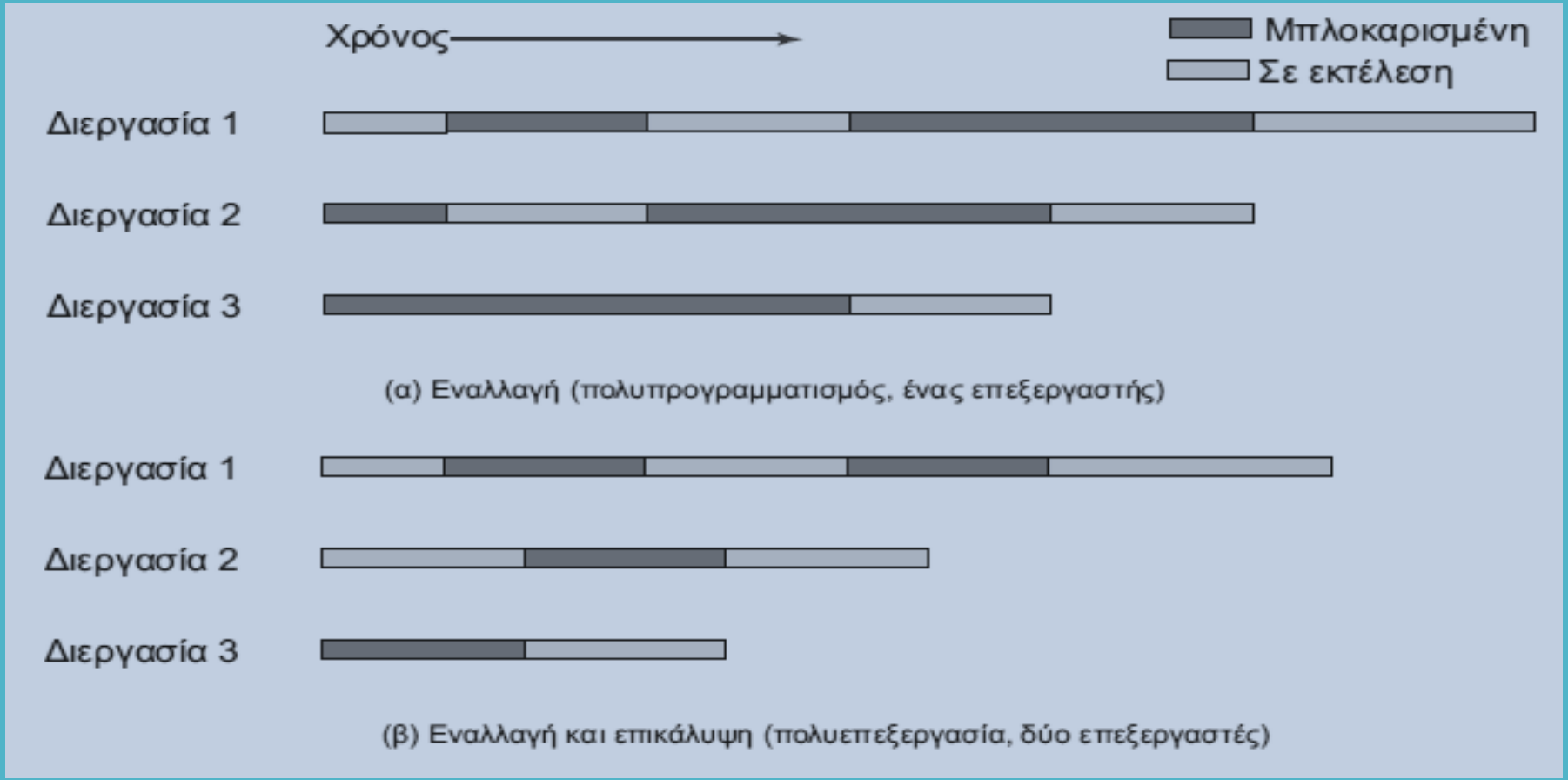
- Αμοιβαίος αποκλεισμός: προϋποθέτει πως όταν μία διεργασία βρίσκεται στο κρίσιμο τμήμα της, καμία άλλη διεργασία δεν μπορεί να βρίσκεται στο δικό της κρίσιμο τμήμα (αλγόριθμοι Decker και Peterson).

- Συνθήκη ανταγωνισμού: κατάσταση στην οποία πολλαπλές διεργασίες διαβάσουν ή γράφουν ένα διαμοιραζόμενο αρχείο και το τελικό αποτέλεσμα εξαρτάται από τη σειρά της εκτέλεσής τους.

- Λιμοκτονία: κατάσταση κατά την οποία μία διεργασία αγνοείται επ’ αόριστον από τον επεξεργαστή.

Διαδιεργασιακή επικοινωνία

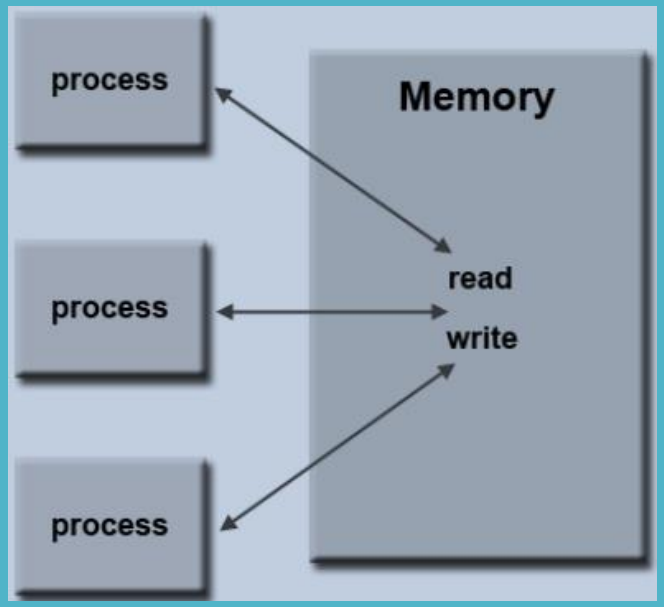
Σε συστήματα απλού επεξεργαστή οι διεργασίες εναλλάσσονται χρονικά έτσι ώστε να δίδεται η εντύπωση ταυτόχρονης εκτέλεσης. Στα συστήματα των πολυεπεξεργαστών εκτός από χρονική εναλλαγή υποστηρίζεται και η χρονική επικάλυψη.

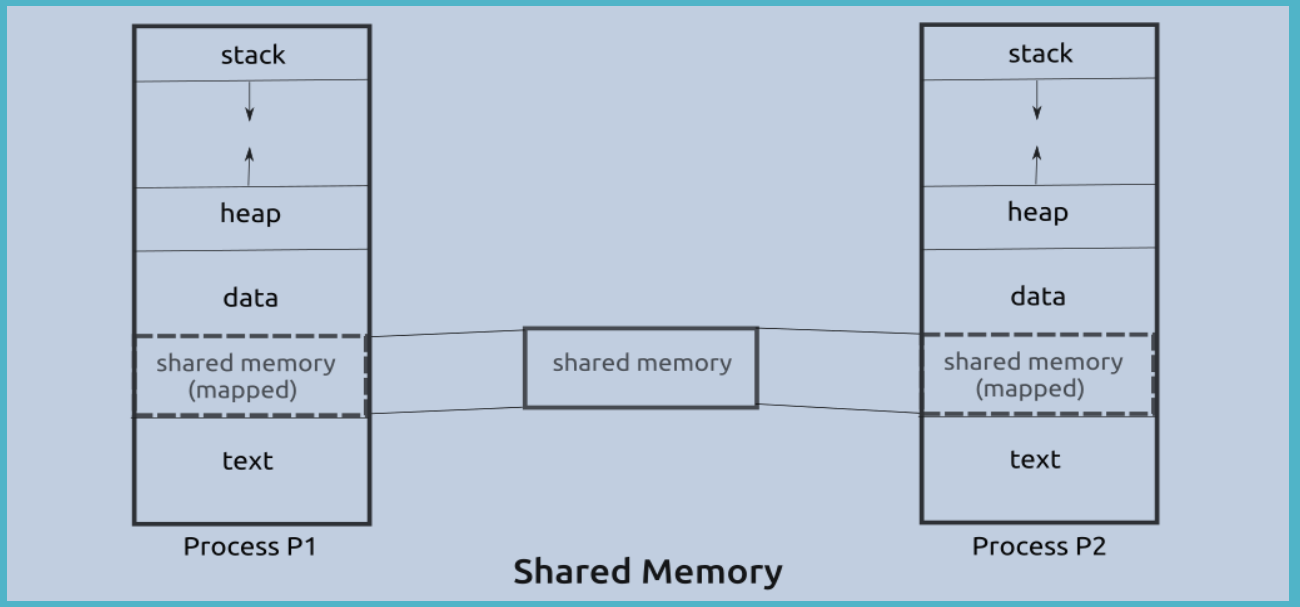


Διαδιεργασιακή επικοινωνία

Στην επιστήμη της πληροφορικής χρησιμοποιούμε τον όρο κοινόχρηστος πόρος για να περιγράψουμε κάθε είδος πόρου (συνήθως περιοχές μνήμης και αρχεία) τα οποία μπορούν να προσπελαστούν από πολλές διεργασίες ταυτόχρονα.

Κοινόχρηστη μνήμη (shared memory, shm) --> δημιουργείται από τον πυρήνα του λειτουργικού ο οποίος την αντιστοιχεί σε κάποια περιοχή του τμήματος δεδομένων της κάθε διεργασίας.





Κοινόχρηστο αρχείο --> επιτρέπει την ταυτόχρονη προσπέλασή του από πολλές διεργασίες για διαδικασίες ανάγνωσης ή εγγραφής.

Διαδιεργασιακή επικοινωνία

Η χρήση κοινόχρηστων πόρων ενδέχεται να οδηγήσει σε προβλήματα απροσδιόριστης έκβασης και συνοχής δεδομένων. Τι γίνεται όταν δύο διεργασίες προσπελαύνουν ταυτόχρονα την ίδια θέση μνήμης η μία για ανάγνωση και η άλλη για εγγραφή?

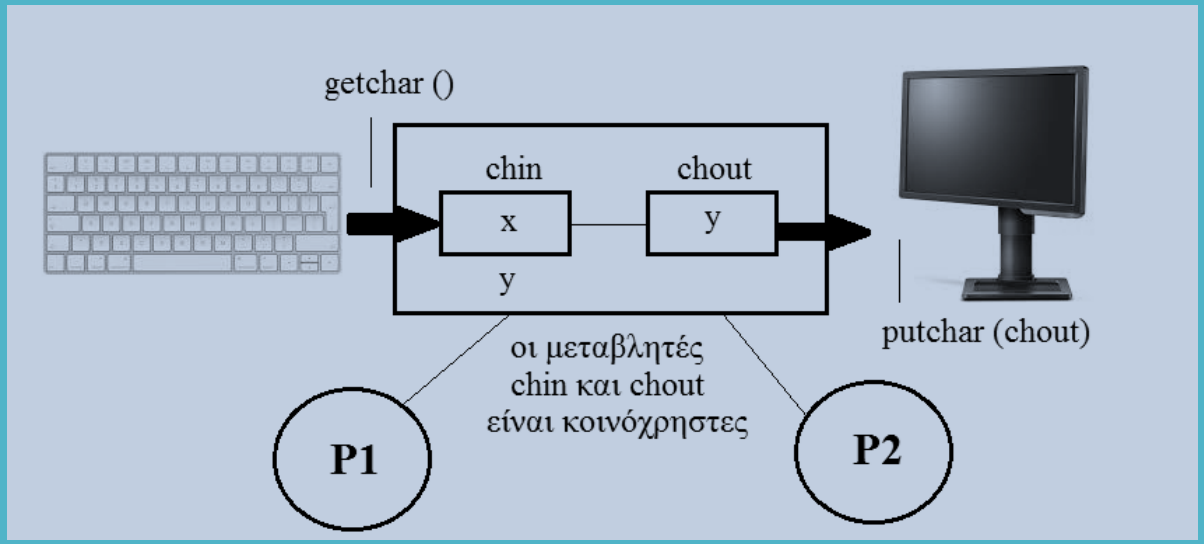
Έστω δύο διεργασίες P1 και P2 που καλούν την εντολή echo.
Οι μεταβλητές chin και chout είναι κοινόχρηστες στις P1 και P2.

```
void echo(){
   chin = getchar();
   chout = chin;
   putchar(chout); }
```

Η P1 καλεί την echo και διακόπτεται όταν εκτελείται η εντολή chin = getchar () με chin = x.

Η P2 ενεργοποιείται και καλεί την echo η οποία ολοκληρώνεται με την εκχώρηση chout = chin = y.

Όταν ενεργοποιείται η P2, η chin έχει την τιμή y η οποία εκτυπώνεται για δεύτερη φορά, ενώ η τιμή x έχει χαθεί οριστικά !!



Διαδιεργασιακή επικοινωνία

ΛΥΣΗ → Επιτρέπουμε σε ΜΙΑ ΜΟΝΟ διεργασία να καλέσει τη συνάρτηση echo. Στην περίπτωση αυτή:

- Η διεργασία P1 εκτελεί την πρώτη εντολή της echo και στη συνέχεια αναστέλλεται.
- Η διεργασία P2 ενεργοποιείται και καλεί την echo. Ωστόσο, επειδή η διεργασία P1 βρίσκεται μέσα στην echo (παρά το γεγονός πως είναι ανεσταλμένη) η διεργασία P2 μπλοκάρεται.
- Η διεργασία P1 συνεχίζει τη λειτουργία της και ολοκληρώνει την κλήση της echo.
- Η διεργασία P2 ξεμπλοκάρει και καλεί με τη σειρά της την echo την οποία και εκτελεί.

Συνθήκη ανταγωνισμού → το τελικό αποτέλεσμα εξαρτάται από τη σειρά εκτέλεσης.

- Έστω διεργασίες P1 και P2 και κοινόχρηστες μεταβλητές b=1 και c=2.
- Η P1 εκτελεί την εντολή b = b + c και η P2 την εντολή c = b + c.
- Εάν εκτελεστεί πρώτη η P1 και μετά η P2 οι νέες τιμές θα είναι b = 3 και c = 5.
- Εάν εκτελεστεί πρώτη η P2 και μετά η P1 οι νέες τιμές θα είναι c = 3 και b = 4.

Η ΑΝΑΓΚΑΙΟΤΗΤΑ ΑΜΟΙΒΑΙΟΥ ΑΠΟΚΛΕΙΣΜΟΥ ΕΠΙΦΕΡΕΙ ΑΔΙΕΞΟΔΑ ΚΑΙ ΛΙΜΟΚΤΟΝΙΕΣ !!

Διαδιεργασιακή επικοινωνία

Σημαφόροι και μεταβλητές αμοιβαίου αποκλεισμού (Dijkstra, 1965)

Θεμελιώδης κανόνας → Δύο ή περισσότερες διεργασίες μπορούν να συνεργάζονται μεταξύ τους μέσω απλών μηνυμάτων με τέτοιο τρόπο ώστε μία διεργασία να εξαναγκάζεται να σταματήσει σε μία συγκεκριμένη θέση όταν λάβει ένα τέτοιο σήμα.

Η σηματοδοσία πραγματοποιείται με τη βοήθεια ειδικών μεταβλητών που ονομάζονται σημαφόροι (semaphores, sem) και οι οποίες αρχικοποιούνται σε μη αρνητικές ακέραιες τιμές.

Πώς χρησιμοποιούνται οι σημαφόροι?

Η συνάρτηση semWait (s) μειώνει την τιμή του σημαφόρου s κατά μία μονάδα. Αν η τιμή του s γίνει αρνητική, τότε μπλοκάρεται η διεργασία που κάλεσε την semWait, διαφορετικά συνεχίζει κανονικά.

Η συνάρτηση semSignal (s) αυξάνει την τιμή του σημαφόρου s κατά μία μονάδα. Αν η τιμή του s μικρότερη ή ίση με το μηδέν, τότε ξεμπλοκάρεται μία διεργασία (εάν υπάρχει) που έχει μπλοκαριστεί από μία συνάρτηση semWait.

Διαδιεργασιακή επικοινωνία

Σημαφόροι και μεταβλητές αμοιβαίου αποκλεισμού (Dijkstra, 1965)

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* τοποθέτηση της διεργασίας αυτής στην s.queue */;
            /* μπλοκάρισμα της διεργασίας αυτής */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* αφαίρεση μίας διεργασίας P από την s.queue */;
            /* τοποθέτηση της διεργασίας P στην έτοιμη λίστα */;
      }
}
```

Κανείς δεν μπορεί να γνωρίζει αν μία διεργασία θα μπλοκαριστεί η όχι, πριν αυτή μειώσει έναν σημαφόρο.

Εάν η αύξηση του σημαφόρου ενεργοποιήσει κάποια άλλη διεργασία, αμφότερες εκτελούνται ταυτόχρονα.

Όταν αποστέλλεται σήμα σε σημαφόρο κανείς δεν γνωρίζει εάν απαραίτητα κάποια διεργασία βρίσκεται σε αναμονή και κατά συνέπεια το πλήθος των διεργασιών που ενεργοποιούνται μπορεί να είναι 0 ή 1.

Διαδιεργασιακή επικοινωνία

Σημαφόροι και μεταβλητές αμοιβαίου αποκλεισμού (Dijkstra, 1965)

Αρχικά η τιμή του σημαφόρου (γενικός σημαφόρος ή σημαφόρος μέτρησης) είναι θετική ή μηδενική.

- Θετική τιμή → ίση με το πλήθος των διεργασιών που μπορούν να εκδώσουν αίτημα αναμονής και αμέσως μετά να συνεχίσουν να εκτελούνται.
- Μηδενική τιμή → το λειτουργικό μπλοκάρει την επόμενη διεργασία που θα ζητήσει αίτημα αναμονής και η τιμή του σημαφόρου γίνεται αρνητική. Κάθε επακόλουθη αναμονή καθιστά την τιμή του σημαφόρου ακόμη πιο αρνητική – στην περίπτωση αυτή η αρνητική τιμή εκφράζει το πλήθος των διεργασιών που αναμένουν να ξεμλοκαριστούν.

Δυαδικοί σημαφόροι (binary semaphores) – παίρνουν μόνο τις τιμές 0 ή 1

Η λειτουργία τους ελέγχεται από τη συνάρτηση semWaitB. Εάν είναι s = 0 μπλοκάρεται η διεργασία που καλεί την semWaitB. Αν είναι s = 1 τότε θέτουμε s = 0 και η διεργασία συνεχίζει να εκτελείται.

Η συνάρτηση semSignalB ελέγχει αν υπάρχουν μποκαρισμένες διεργασίες σε αυτόν τον σημαφόρο – δηλαδή να έχουν s = 0. Εάν υπάρχουν, κάποια από αυτές ξεμπλοκάρεται (με πολιτική FIFO για τους ισχυρούς σημαφόρους και χωρίς πολιτική για τους ασθενείς σημαφόρους) , εάν όχι θέτουμε s = 1.

Οι δυαδικοί σημαφόροι έχουν την ίδια εκφραστική δύναμη με τους γενικούς σημαφόρους.

Διαδιεργασιακή επικοινωνία

Κλειδαριές αμοιβαίου αποκλεισμού

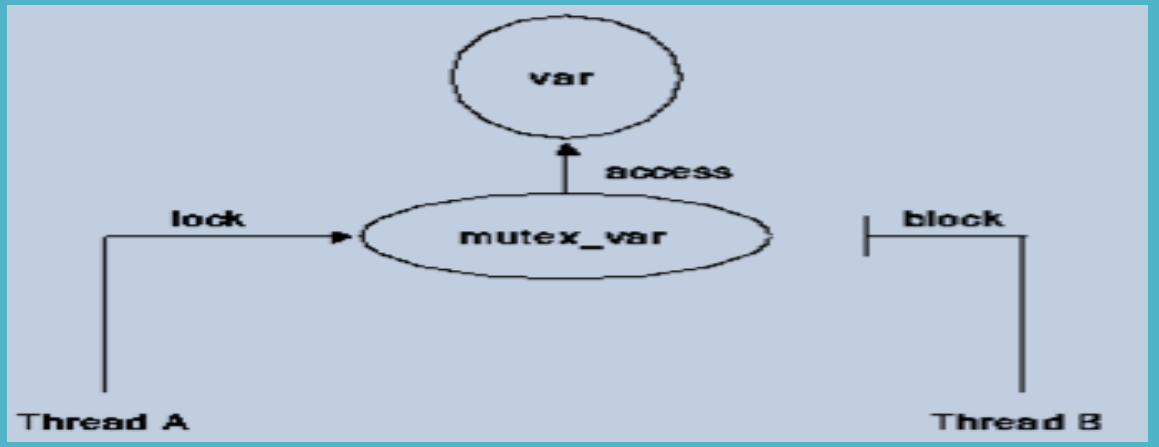
Οι κλειδαριές αμοιβαίου αποκλεισμού (mutual exclusion locks, mutex) είναι παρόμοιες με τους δυαδικούς σημαφόρους που χρησιμοποιούνται για τη δέσμευση και την αποδέσμευση αντικειμένων.

Είναι παρόμοιες με τους δυαδικούς σημαφόρους που χρησιμοποιούνται για τη δέσμευση και την αποδέσμευση αντικειμένων.

Όταν αποκτώνται δεδομένα που δεν μπορούν να αποτελέσουν αντικείμενο διαμοιρασμού ή εκκινείται επεξεργασία η οποία δεν μπορεί να εκτελεστεί κάπου αλλού στο σύστημα, τότε η mutex τίθεται σε κατάσταση κλειδώματος, λαμβάνοντας την τιμή 0.

Όταν δεν απαιτούνται πλέον αυτά τα δεδομένα ή όταν έχει τερματιστεί η παραπάνω επεξεργασία, τότε η mutex τίθεται σε κατάσταση ξεκλειδώματος, λαμβάνοντας την τιμή 1.

Η διαφορά ανάμεσα σε μία mutex και σε έναν δυαδικό σημαφόρο, είναι πως η διεργασία που κλειδώνει τη mutex πρέπει να είναι και αυτή που θα την ξεκλειδώσει, ενώ ένας δυαδικός σημαφόρος μπορεί να κλειδωθεί από μία διεργασία και να ξεκλειδωθεί από μία άλλη.



Διαδιεργασιακή επικοινωνία

Στο λειτουργικό σύστημα Linux η επικοινωνία μεταξύ των διεργασιών μπορεί να πραγματοποιηθεί χρησιμοποιώντας τις επόμενες προσεγγίσεις:

- Αγωγοί (pipes) ανώνυμοι και επώνυμοι.
- Σήματα (signals) που ανταλλάσσονται ανάμεσα στις διεργασίες και στον πυρήνα.
- Υποδοχείς (sockets) που επιτρέπουν την επικοινωνία διεργασιών οι οποίες εκτελούνται στον ίδιο ή σε διαφορετικούς υπολογιστές.
- **Ουρές μηνυμάτων (message queues, msg).**
- **Σημαφόρους (semaphores, sem) & κοινόχρηστα αρχεία (shared files).**
- **Κοινόχρηστες περιοχές μνήμης (shared memory, shm).**

Εκτός από τις παραπάνω υπάρχουν και άλλες τεχνικές όπως είναι η μεταβίβαση μηνυμάτων (message passing) που σχετίζεται με τα παράλληλα και κατανεμημένα συστήματα.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία με δομές FIFO (επώνυμους αγωγούς) χαρακτηρίζεται από τα ακόλουθα πλεονεκτήματα και μειονεκτήματα:

Πλεονεκτήματα FIFO:

- Είναι εύκολοι στην κατανόηση και στη χρήση τους.
- Είναι διαθέσιμοι σε όλες τις εκδοχές και εκδόσεις του λειτουργικού συστήματος UNIX.
- Είναι αρκετά αποδοτικοί.
- Λειτουργούν καλά τόσο με ρεύματα δεδομένων όσο και με απλά μηνύματα.

Μειονεκτήματα FIFO:

- Μία δομή FIFO δεν μπορεί να έχει πολλούς αναγνώστες.
- Απαιτείται προσωρινή αποθήκευση η οποία είναι αρκετά δαπανηρή.
- Αν το μέγεθος του μηνύματος είναι πολύ μεγάλο, ο συγγραφέας μπορεί να μπλοκάρει.

Το πρόβλημα παραγωγού καταναλωτή → θα πρέπει να διασφαλιστεί πως ο παραγωγός (συγγραφέας) δεν θα πρέπει να προσθέσει δεδομένα στον αγωγό εάν αυτός είναι γεμάτος και πως ο καταναλωτής (αναγνώστης) δεν θα προσπαθήσει να διαβάσει δεδομένα από έναν άδειο αγωγό.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία με σήματα (signals) χαρακτηρίζεται από τα ακόλουθα πλεονεκτήματα και μειονεκτήματα:

Πλεονεκτήματα σημάτων

- Επιτρέπουν ασύγχρονη λειτουργία (το σήμα στέλνεται όταν απαιτείται κάτι τέτοιο).
- Ως αποτέλεσμα, οδηγούν σε μεγαλύτερη απόδοση της εφαρμογής.
- Προσφέρουν εύκολο χειρισμό διεργασιών μέσω της συνάρτησης αποστολής σημάτων kill.

Μειονεκτήματα σημάτων

- Οδηγούν σε καταστάσεις ανταγωνισμού υπό την έννοια πως ίσως η έκβαση μιας διεργασίας να εξαρτάται από τη σειρά αποστολής των σημάτων.
- Οι εφαρμογές που χρησιμοποιούν σήματα είναι δύσκολες στην αποσφαλμάτωση.
- Ανακύπτουν προβληματικές καταστάσεις όταν οι διεργασίες παραλαμβάνουν ένα σήμα τη στιγμή που έκτελούν τη συνάρτηση του χειρισμού τους.
- Οι διεργασίες δεν χαρακτηρίζονται από μεγάλο βαθμό απομόνωσης.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία με υποδοχείς (sockets) χαρακτηρίζεται από τα ακόλουθα πλεονεκτήματα και μειονεκτήματα:

Πλεονεκτήματα υποδοχών

- Υποστηρίζουν πλήρως αμφίδρομη επικοινωνία, σε αντίθεση με τους αγωγούς που επιτρέπουν τη μετακίνηση δεδομένων μόνο προς τη μία κατεύθυνση. Ωστόσο είναι πιο δύσχρηστα από τους αγωγούς στο να διασφαλίσουν την χωρίς σφάλματα διακίνηση της διακινούμενης πληροφορίας.
- Η διαδικασία αποσφαλμάτωσης είναι σχετικά απλή ενώ διασφαλίζεται η φορητότητα.
- Επιτρέπουν την εύκολη υλοποίηση εφαρμογών client – server.

Μειονεκτήματα υποδοχών

- Η επιβάρυνση του συστήματος είναι μεγαλύτερη και όταν είναι γνωστό πως η εφαρμογή θα εκτελεστεί σε έναν απλό υπολογιστή είναι προτιμότερη η λύση της κοινόχρηστης μνήμης.
- Απαιτείται ιδιαίτερη προσοχή για τη διασφάλιση της εμπιστευτικότητας των πληροφοριών ειδικά σε περιπτώσεις στις οποίες η εφαρμογή μπορεί να προσπελαστεί και από τρίτους.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία (Interprocess Communication, IPC) που στηρίζεται στη χρήση σημαφόρων, ουρών μηνυμάτων και κοινόχρηστης μνήμης, πραγματοποιείται με τη βοήθεια δύο διαφορετικών ομάδων κλήσεων συναρτήσεων:

System V IPC και POSIX IPC

Στο υποσύστημα IPC του System V υπάρχουν και τα τρία παραπάνω είδη αντικειμένων, δηλαδή

- Ουρές μηνυμάτων (Message Queues, msg)
- Σημαφόροι (Semaphores, sem)
- Κοινόχρηστη μνήμη (Shared Memory, shm)

Για τη διαχείριση αυτών των αντικειμένων υπάρχουν δύο οικογένειες συναρτήσεων με ονόματα Xget και Xctl όπου το X μπορεί να είναι ένα από τα msg, sem, shm. Υπάρχουν λοιπόν έξι συναρτήσεις, οι

msgget, msgctl, shmget, shmctl, semget, semctl

ενώ επιπλέον υπάρχουν άλλες πέντε συναρτήσεις με ονόματα

msgsnd, msgrcv, shmat, shmdt, semop

Διαδιεργασιακή επικοινωνία

Ιδιότητες των αντικειμένων του System V IPC

- Υφίστανται μόνο σε ένα απλό υπολογιστή.
- Η διάρκεια ζωής τους είναι η ίδια με αυτή του πυρήνα και κατά συνέπεια καταστρέφονται κατά την επανεκκίνηση του υπολογιστή.
- Προσπελαύνονται με τη βοήθεια ενός ακέραιου αναγνωριστικού το οποίο είναι σταθερό σε όλη τη διάρκεια της ζωής του αντικειμένου. Όποια διεργασία γνωρίζει αυτό το αναγνωριστικό μπορεί να προσπελάσει το αντικείμενο χωρίς να χρειαστεί να το ανοίξει.
- Η τιμή του αναγνωριστικού για το κάθε αντικείμενο είναι διαφορετική σε κάθε επανεκκίνηση ενώ η διαδικασία λήψης του αναγνωριστικού διευκολύνεται από τη χρήση ενός μόνιμου κλειδιού.
- Δεν υπάρχουν i-nodes ή pathnames και κατά συνέπεια δεν μπορούν να χρησιμοποιηθούν οι παραδοσιακές κλήσεις συστήματος όπως είναι η unlink, stat, read και write.
- Το κάθε αντικείμενο χαρακτηρίζεται από άδειες πρόσβασης για ανάγνωση, εγγραφή και εκτέλεση παρόμοιες με αυτές των αρχείων αν και το bit εκτέλεσης δεν χρησιμοποιείται, αφού τα αντικείμενα αυτά δεν εκτελούνται.
- Το κάθε αντικείμενο διαθέτει επίσης αναγνωριστικά για τον κάτοχο και την ομάδα του κατόχου τα οποία προσπελαύνονται και τροποποιούνται με τις συναρτήσεις Xctl.

Διαδιεργασιακή επικοινωνία

POSIX IPC

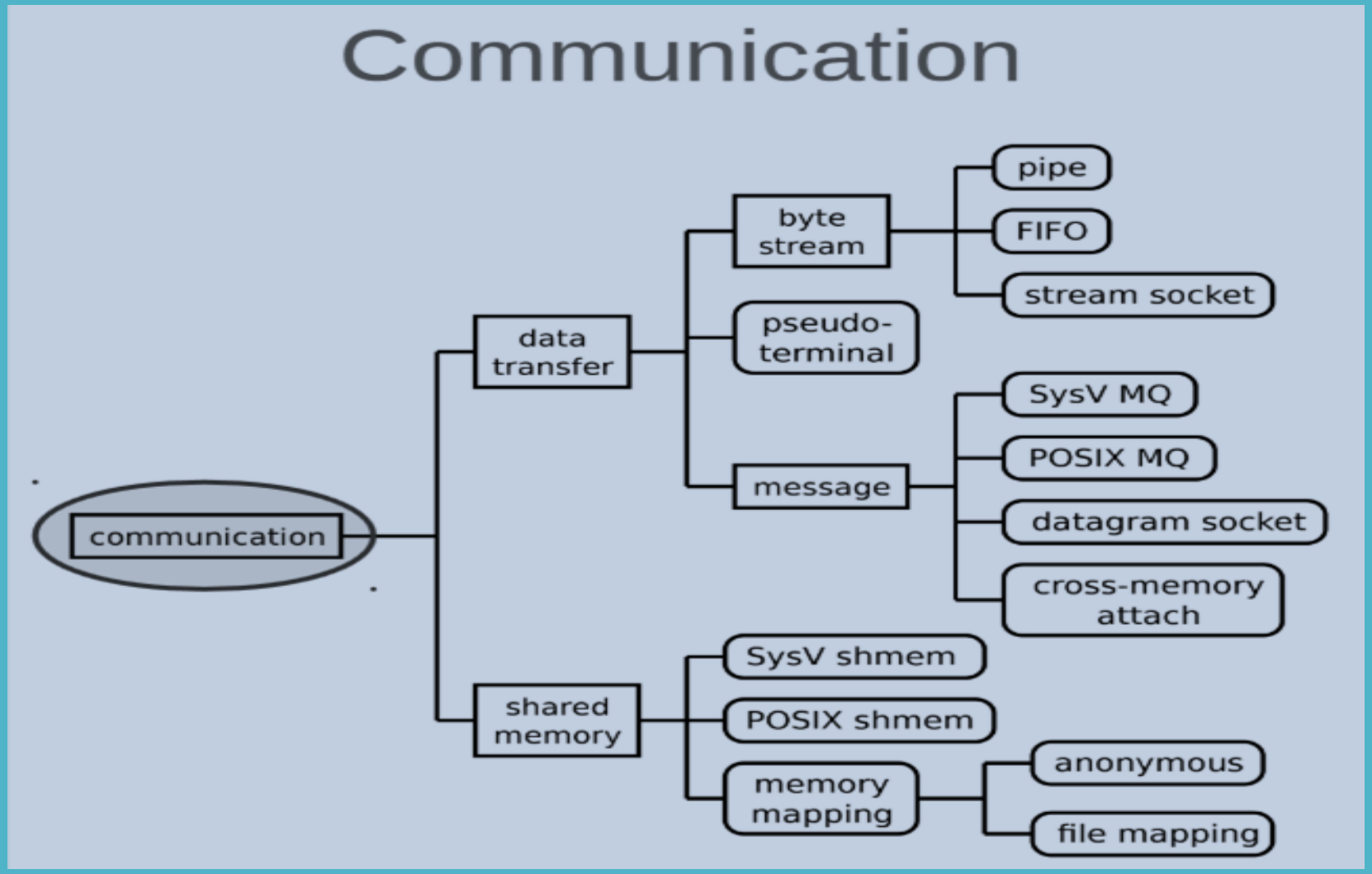
- Στο POSIX τη θέση των κλειδιών παίρνουν κατάλληλα ορισμένα αλφαριθμητικά.
- Τα ονόματα των αντικειμένων ακολουθούν τους ίδιους κανόνες με τα ονόματα των διαδρομών (εάν το όνομα ξεκινά με κάθετο, όλες οι αναφορές σε αυτό σχετίζονται με το ίδιο αντικείμενο).
- Τα αντικείμενα του POSIX IPC (όπως και του System V IPC) δεν είναι αρχεία.

Στο λειτουργικό σύστημα Linux, το πλήρες σύστημα IPC περιλαμβάνει αντικείμενα για πραγματοποίηση δύο θεμελιωδών λειτουργιών

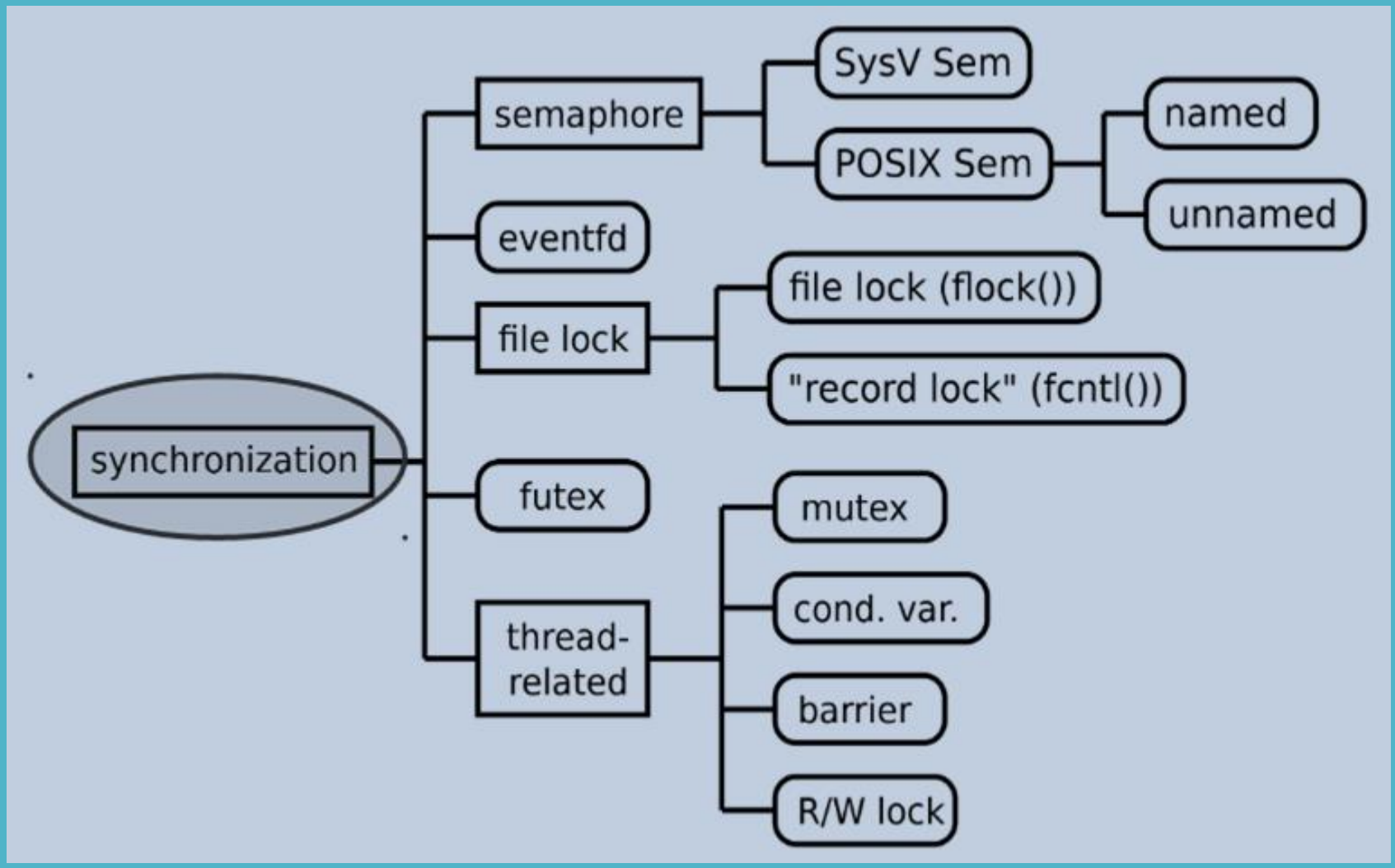
Επικοινωνία & Συγχρονισμός

Τα πιο σημαντικά αντικείμενα της πρώτης κατηγορίας είναι η κοινόχρηστη μνήμη, οι αγωγοί, οι υποδοχείς και οι ουρές μηνυμάτων, ενώ στη δεύτερη κατηγορία ανήκουν οι σημαφόροι και οι μεταβλητές αμοιβαίου αποκλεισμού.

Διαδιεργασιακή επικοινωνία



Διαδιεργασιακή επικοινωνία



Διαδιεργασιακή επικοινωνία

Κοινόχρηστα αρχεία

Δύο διεργασίες ανοίγουν ταυτόχρονα το ίδιο αρχείο, η μία για εγγραφή και η άλλη για ανάγνωση.

Εάν η διεργασία - συγγραφέας προσπελάσει το αρχείο ακριβώς την ίδια στιγμή με τη διεργασία – αναγνώστης, το αποτέλεσμα που θα προκύψει μπορεί να είναι απροσδιόριστο.

Προκειμένου να αποφύγουμε τέτοια προβλήματα, η διεργασία – συγγραφέας θα πρέπει να κλειδώσει το αρχείο πριν την έναρξη της διαδικασίας εγγραφής δεδομένων σε αυτό, έτσι ώστε κατά τη χρονική διάρκεια της εγγραφής να μην είναι δυνατή η προσπέλαση του αρχείου από άλλη διεργασία για ανάγνωση ή εγγραφή. Όταν η εγγραφή ολοκληρωθεί, το αρχείο ξεκλειδώνει και μπορεί πλέον να χρησιμοποιηθεί και από άλλες διεργασίες **(exclusive lock)**.

Από την άλλη πλευρά, η ανάγνωση δεν τροποποιεί τα δεδομένα του αρχείου και γενικά είναι επιτρεπτή η ταυτόχρονη ανάγνωση του αρχείου από πολλές διεργασίες. Για μία ακόμη φορά απαιτείται κλείδωμα του αρχείου αλλά αυτή τη φορά δεν είναι αποκλειστικό αλλά κοινόχρηστο **(shared lock)** έτσι ώστε η ανάγνωση να μπορεί να πραγματοποιηθεί από πολλές διεργασίες. Ωστόσο για όσο χρονικό διάστημα ένας αναγνώστης έχει ένα κοινόχρηστο κλείδωμα, η εγγραφή στο αρχείο δεν επιτρέπεται.

Η διαχείριση των shared και exclusive locks γίνεται από τη συνάρτηση **fcntl** .

Διαδιεργασιακή επικοινωνία

Κοινόχρηστα αρχεία (lock & unclock)

Το πρωτότυπο της συνάρτησης fcntl (unistd.h και fcntl.h) έχει τη μορφή

int fcntl (int fd, int cmd, [opt arg]);

Η συνάρτηση επιτρέπει το χειρισμό ενός ανοικτού αρχείου που περιγράφεται από το πρώτο όρισμα fd, σύμφωνα με το δεύτερο όρισμα cmd ενώ η συμπεριφορά της καθορίζεται από το τρίτο όρισμα που είναι μία δομή τύπου flock της μορφής

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;   /* offset in bytes, relative to l_whence */
  off_t  l_len;     /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;     /* returned with F_GETLK */
};
```

Σχετικά με το πρώτο όρισμα αυτό μπορεί να λάβει κάποια από τις τιμές F_SETLK που ενεργοποιεί ένα κλείδωμα όταν το πεδίο l_type έχει μία από τις τιμές F_RDLCK ή F_WRLCK) ή καταργεί ένα κλείδωμα όταν το πεδίο l_type έχει την τιμή F_UNLCK για τα bytes του αρχείου που ορίζονται από τα πεδία l_start, l_len και l_whence της δομής flock).

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστου αρχείου

```
int main() {
  struct flock lock;
  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 */
  if ((fd = open(FileName, O_RDWR | O_CREAT, 0666)) < 0)  {
       perror("open failed..."); exit(-1); }
  if (fcntl(fd, F_SETLK, &lock) < 0) {
       perror("fcntl failed to get lock..."); exit(-2); }
  else {
    write(fd, DataString, strlen(DataString)); /* populate data file */
    fprintf(stderr, "Process %d has written to data file...\n", lock.l_pid); }

  /* Now release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0) {
       perror("explicit unlocking failed..."); exit(-1); }

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

Η διεργασία – συγγραφέας ανοίγει το αρχείο data.dat υπό συνθήκες exclusive lock και αποθηκεύει σε αυτό το περιεχόμενο μιας συμβολοσειράς.

lock.l_type = F_WRLCK
exclusive lock

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

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

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστου αρχείου

```c
int main() {
  struct flock lock;
  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 */
  if ((fd = open(FileName, O_RDONLY)) < 0) {
       perror("open to read failed..."); exit(-1); }

  /* If the file is write-locked, we can't continue. */
  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); }

  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); }

  /* Read the bytes (they happen to be ASCII codes) one at a time. */
  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 */

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

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

Η διεργασία – αναγνώστης ανοίγει το αρχείο data.dat υπό συνθήκες shared lock και διαβάζει από αυτό το περιεχόμενο της συμβολοσειράς.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define FileName "data.dat"
```

lock.l_type = F_RDLCK
shared lock

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστου αρχείου

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

Το κλείδωμα του αρχείου αναιρείται με δύο τρόπους: είτε άμεσα καλώντας την fnctl

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

είτε έμμεσα, απλά κλείνοντας το αρχείο, αφού προφανώς εάν μία διεργασία κλείσει ένα αρχείο αυτό σημαίνει πως δεν το χρειάζεται πλέον, οπότε δεν έχει λόγο να παραμείνει κλειδωμένο.

Η χρήση του flag F_SETLK στην fcntl

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

προκαλεί την άμεση επιστροφή της συνάρτησης είτε καταφέρει να κλειδώσει το αρχείο είτε όχι, ενώ εάν χρησιμοποιήσουμε το flag F_SETLKW η συνάρτηση μπλοκάρει μέχρι να κλειδωθεί το αρχείο.

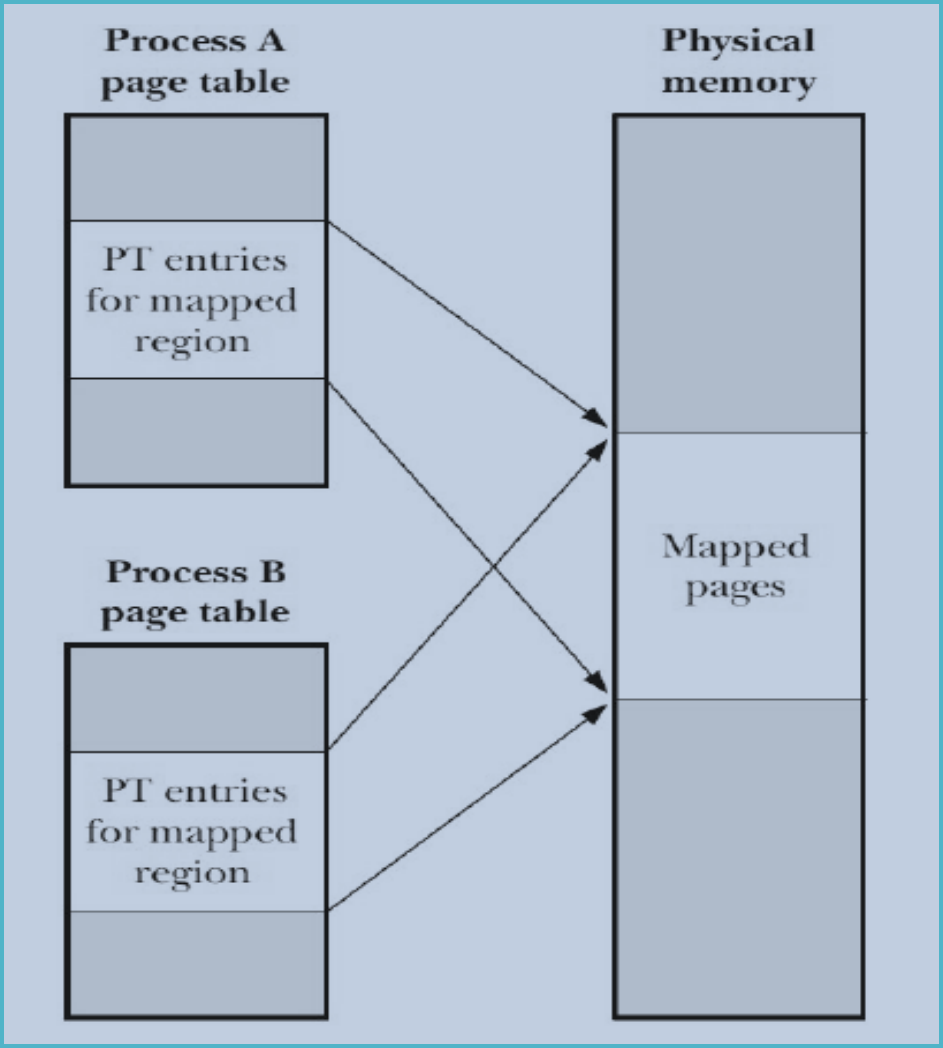
Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Στην προσέγγιση της κοινόχρηστης μνήμης, οι διεργασίες διαμοιράζονται τις ίδιες σελίδες φυσικής μνήμης και η επικοινωνία μεταξύ τους πραγματοποιείται γράφοντας δεδομένα σε αυτή την περιοχή.

Αυτή η προσέγγιση είναι ιδιαίτερα αποδοτική αφού η διαδικασία της αντιγραφής δεδομένων περιορίζεται στο χώρο του χρήστη χωρίς την εμπλοκή του πυρήνα.

Ωστόσο απαιτείται συγχρονισμός της διαδικασίας προσπέλασης των διεργασιών μέσω σημαφόρων για να μην ανακύπτουν συνθήκες ανταγωνισμού.



Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Το Linux προσφέρει δύο διαφορετικά API (Application Programming Interface) για τη χρήση κοινόχρηστης μνήμης, το API του System V και το POSIX API τα οποία δεν είναι συμβατά μεταξύ τους και σε καμία περίπτωση δεν θα πρέπει να συνδυάζονται.

Το POSIX API τελεί ακόμη υπό καθεστώς ανάπτυξης και εξαρτάται από την έκδοση του πυρήνα, κάτι που επηρεάζει και τη φορητότητα.

Το POSIX API υλοποιεί την κοινόχρηστη μνήμη με τη βοήθεια ενός αρχείου που απεικονίζεται στη μνήμη (backing file) και το οποίο περιέχει τα περιεχόμενα του κοινόχρηστου τμήματος μνήμης. Η χρήση αυτού του αρχείου επιτρέπει την επικοινωνία μεταξύ διεργασιών που δεν σχετίζονται μεταξύ τους.

Οι εφαρμογές προσπελαύνουν μόνο το τμήμα της κοινόχρηστης μνήμης και όχι το βοηθητικό αρχείο, ενώ ο συγχρονισμός του αρχείου με την κοινόχρηστη μνήμη γίνεται από το λειτουργικό σύστημα.

Υπάρχουν τρεις τρόποι χρήσης κοινόχρηστης μνήμης και αντίστοιχης επικοινωνίας διεργασιών:

- Μη σχετιζόμενες μεταξύ τους διεργασίες με χρήση βοηθητικού αρχείου.
- Μη σχετιζόμενες μεταξύ τους διεργασίες χωρίς τη χρήση βοηθητικού αρχείου.
- Σχετιζόμενες μεταξύ τους διεργασίες (πατέρας - παιδί) χωρίς τη χρήση βοηθητικού αρχείου.

Διαδιεργασιακή επικοινωνία

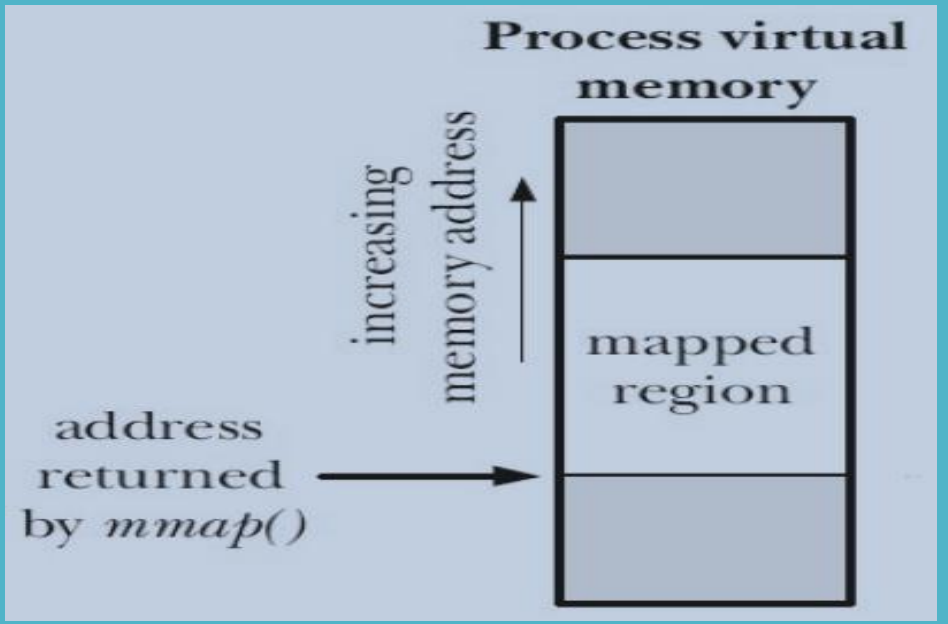
Κοινόχρηστη μνήμη

Η βασική συνάρτηση που πραγματοποιεί την παραπάνω διαδικασία είναι η συνάρτηση

void * mmap (void * daddr, size_t len, int prot,
int flags, int fd, off_t offset)

με τα ορίσματα που περιέχει να έχουν την ακόλουθη σημασία:

- daddr → ορίζει πού θα τοποθετηθεί η απεικόνιση (εάν αυτό το όρισμα λάβει τιμή NULL αυτή η απόφαση λαμβάνεται από τον πυρήνα).
- len → το μήκος της απεικόνισης σε bytes.
- prot → προστασία μνήμης (read, write, execute)
- flags → MAP_SHARED, MAP_ANONYMOUS
- fd → περιγραφέας βοηθητικού αρχείου
- offset → μετατόπιση μέσα στο βοηθητικό αρχείο.

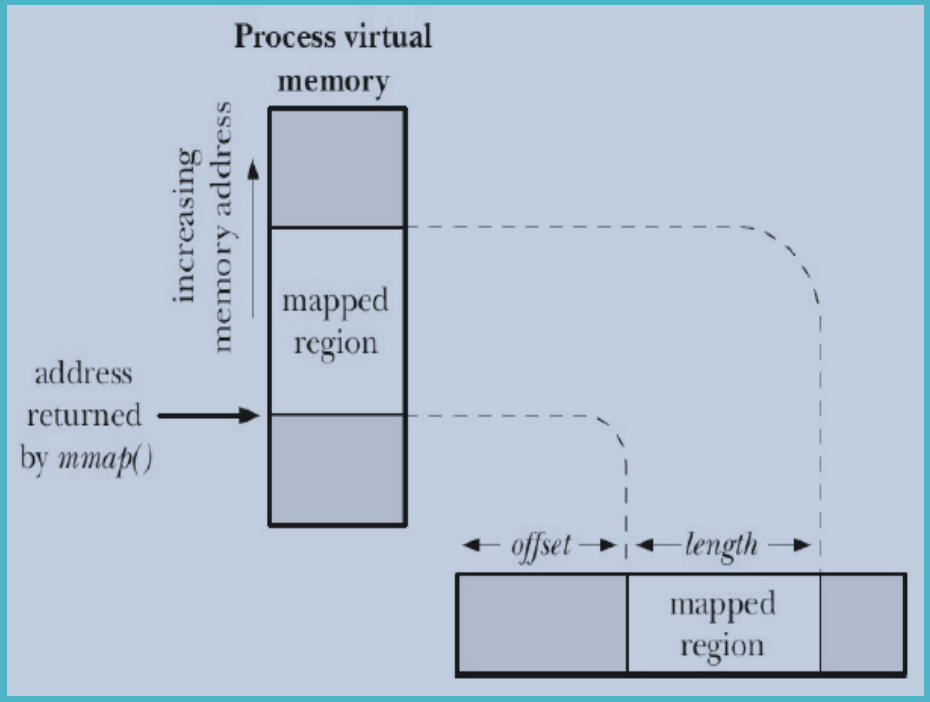


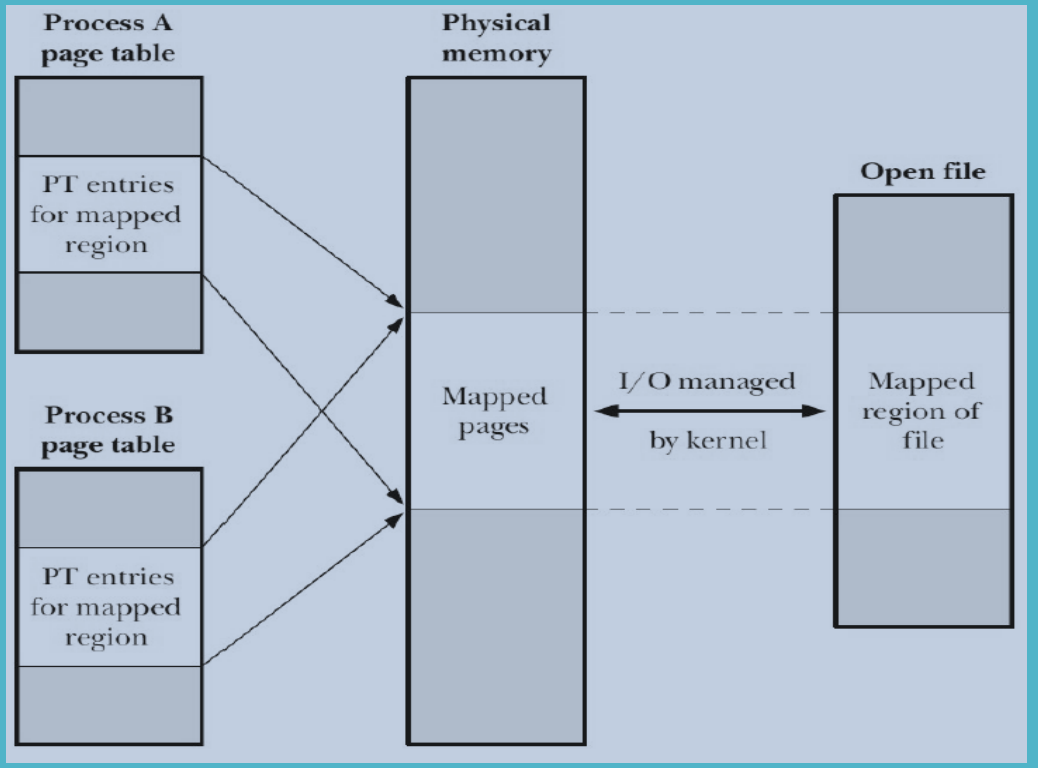
Η επιστρεφόμενη τιμή της συνάρτησης mmap είναι ένας δείκτης που προσδιορίζει το κοινόχρηστο τμήμα μνήμης που ορίζεται με τον παραπάνω τρόπο.

MAP_SHARED → ορατό στις άλλες διεργασίες …

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη





Εκτός από τη χρήση κοινόχρηστης μνήμης από σχετιζόμενες διεργασίες, αυτή μπορεί να χρησιμοποιηθεί και από διεργασίες που σχετίζονται μεταξύ τους με σχέση πατέρα – παιδιού.

Ωστόσο, σε αυτή την περίπτωση δεν χρησιμοποιείται βοηθητικό αρχείο και κατά συνέπεια, τα δύο τελευταία ορίσματα στη συνάρτηση nmap έχουν τις τιμές -1 και 0 – επομένως, η συνάρτηση καλείται ως

addr = mmap (NULL, len, prot, flags, -1, 0);

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Στο μοντέλο κοινόχρηστης μνήμης του POSIX το σύστημα επιτρέπει την επικοινωνία μη σχετιζόμενων μεταξύ τους διεργασιών χωρίς τη χρήση βοηθητικού αρχείου, με αποτέλεσμα την αύξησης της απόδοσης αφού δεν έχουμε πρόσθετη επιβάρυνση λόγω διαδικασιών εισόδου / εξόδου.

Η δημιουργία / άνοιγμα ενός νέου αντικειμένου ή το άνοιγμα ενός υπάρχοντος αντικειμένου γίνεται με τη βοήθεια της συνάρτησης (απαιτείται η χρήση των αρχείων sys/mman.h, sys/stat.h και fcntl.h)

int shm_open (const char * name, int oflags, mode_t mode)

η οποία επιστρέφει έναν ακέραιο file descriptor (fd) στο νέο αντικείμενο. Στην παραπάνω σύνταξη:

- το όρισμα name (που θα πρέπει να ξεκινά με /) αναφέρεται στο όνομα του αντικειμένου.
- τα oflags είναι παρόμοια με εκείνα της open (O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_TRUNK)
- το όρισμα mode εκφράζει τα δικαιώματα πρόσβασης.

ΣΗΜΑΝΤΙΚΟ → το βοηθητικό αρχείο (όταν χρησιμοποιείται) δημιουργείται στο σύστημα αρχείων /dev/shm το οποίο ΔΕΝ χρησιμοποιεί το δίσκο αλλά τη μνήμη του υπολογιστή (ramdisk). Αν και αυτό το σύστημα αρχείων είναι τύπου tmpfs (temporary file system) ωστόσο αυτά τα αρχεία ΔΕΝ θα πρέπει να θεωρηθούν της ίδιας φύσης με τους καταλόγους /tmp και /var/tmp που υπάρχουν στο σκληρό δίσκο.

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Δημιουργία (O_CREAT) και άνοιγμα νέου αντικειμένου shm γνωστού μεγέθους ίσο με size bytes.

```
fd = shm_open("/myshm", O_CREAT | O_EXCL | O_RDWR, 0600);
ftruncate(fd, size);    // Set size of object
addr = mmap(NULL, size,
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Δημιουργία και άνοιγμα υπάρχοντος αντικειμένου shm άγνωστου μεγέθους.

```
fd = shm_open("/myshm", O_RDWR, 0); // No O_CREAT

// Use object size as length for mmap()

struct stat sb;
fstat(fd, &sb);

addr = mmap(NULL, sb.st_size,
            PROT_READ | PROT_WRITE,MAP_SHARED, fd, 0);
```

Κατάργηση ονόματος με την **int shm_unlink (const char * name);**

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Ο συγχρονισμός των διεργασιών έτσι ώστε να αποτρέπεται η ταυτόχρονη ενημέρωση της κοινόχρηστης μνήμης από τις διεργασίες του συστήματος, πραγματοποιείται με τη βοήθεια σημαφόρων.

Ένας σημαφόρος είναι μία ακέραια μεταβλητή που παρακολουθείται από τον πυρήνα και ο οποίος μπορεί να λάβει μηδενική τιμή και θετικές τιμές, όχι όμως και αρνητική τιμή.

Υπάρχουν δυο είδη σημαφόρων, οι ανώνυμοι που είναι ενσωματωμένοι στην κοινόχρηστη μνήμη και οι επώνυμοι που είναι ανεξάρτητα αντικείμενα.

Εάν υπάρχουν Ν ταυτόσημοι κοινόχρηστοι πόροι, τότε η αρχική τιμή του σημαφόρου είναι Ν ενώ η μεταβολή της πραγματοποιείται με τη βοήθεια των επόμενων συναρτήσεων (του αρχείου semaphore.h)

int sem_post (sem_t * semp); αύξηση της τιμής κατά 1 (unlock)

int sem_wait (sem_t * semp); μείωση της τιμής κατά 1 (lock)

Εκτός από τους παραπάνω σημαφόρους αρίθμησης ή γενικούς σημαφόρους, το σύστημα υποστηρίζει και δυαδικούς σημαφόρους με τιμές 0 και 1.

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Για τους ανώνυμους σημαφόρους χρησιμοποιούνται οι συναρτήσεις

```
int sem_init (sem_t * semp, int pshared, unsigned int value);
int sem_destroy (sem_t * semp);
```

Η συνάρτηση sem_init αρχικοποιεί τον σημαφόρο semp στην τιμή value. Το όρισμα pshared παίρνει την τιμή 0 εάν η κοινόχρηστη μνήμη χρησιμοποιηθεί από threads και την τιμή 1 εάν η κοινόχρηστη μνήμη χρησιμοποιηθεί από διεργασίες.

Η συνάρτηση sem_destroy διαγράφει έναν ανώνυμο σημαφόρο που είχε δημιουργηθεί με την sem_init.

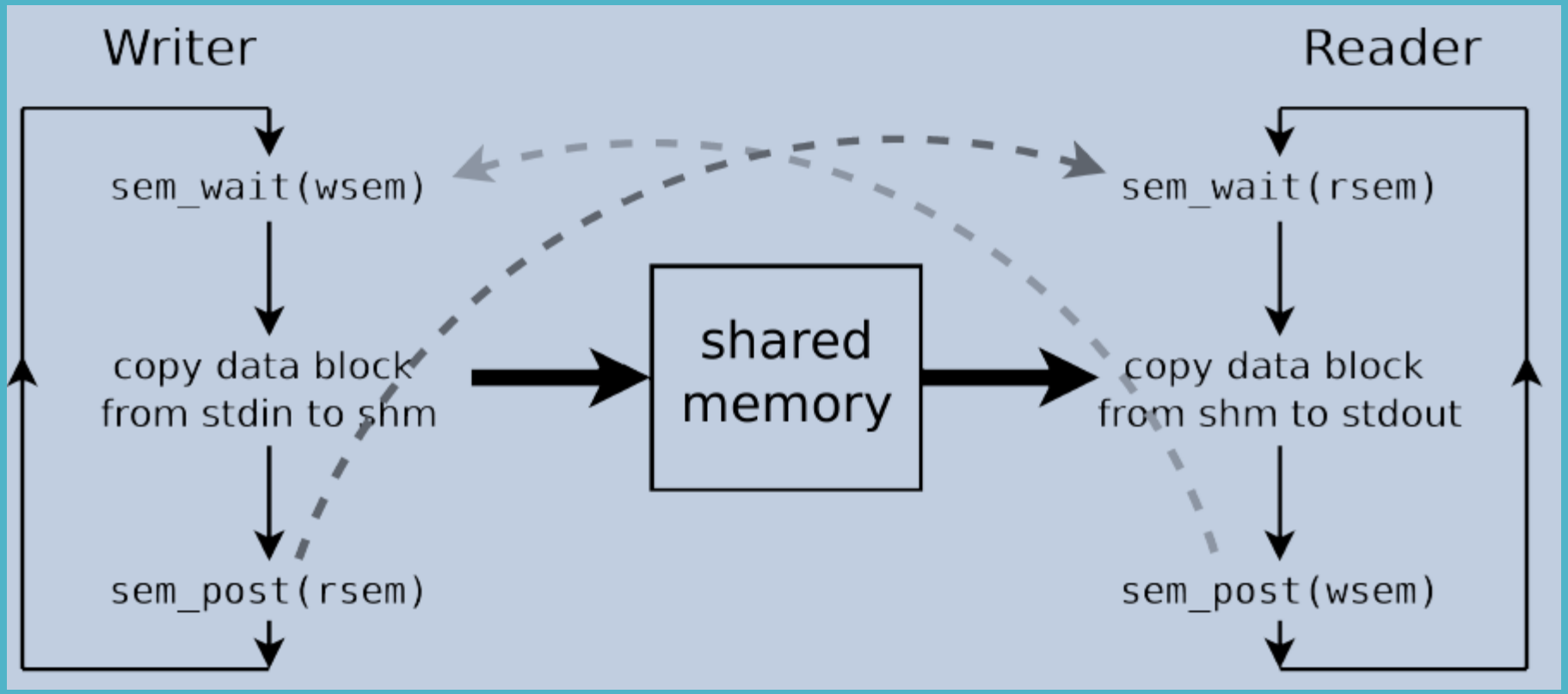
Για τους επώνυμους σημαφόρους χρησιμοποιούνται οι συναρτήσεις

```
int sem_open (const char * name, int oflag,
              mode_t mode, unsigned int value);
int sem_unlink (const char * name);
```

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Παράδειγμα χρήσης ανώνυμων σημαφόρων



Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστης μνήμης και σημαφόρων

Memwriter.c

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

int main() {

  int fd = shm_open("/shMemEx", O_RDWR | O_CREAT, 0644);
  if (fd < 0) { perror ("Can't open shared mem segment..."); exit (-1); }
  ftruncate(fd, ByteSize);

  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); }

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

  strcpy(memptr, MemContents);

  /* increment the semaphore so that memreader can read */
  if (sem_post(semptr) < 0) { perror ("sem_post"); exit (-3); }

  sleep(12); /* give reader a chance */

  munmap(memptr, ByteSize); /* unmap the storage */
  close(fd);
  sem_close(semptr);
  shm_unlink("/shMemEx");
  return 0; }
```

```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>
```

Αυτή η εφαρμογή γράφει στην κοινόχρηστη μνήμη χρησιμοποιώντας βοηθητικό αρχείο και αναμένει για 12 δευτερόλεπτα για να δώσει τη δυνατότητα στην εφαρμογή ανάγνωσης να διαβάσει τα περιεχόμενα από την κοινόχρηστη μνήμη.

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστης μνήμης και σημαφόρων

Memreader.c

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

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); }

  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); }

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

```
#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>
```

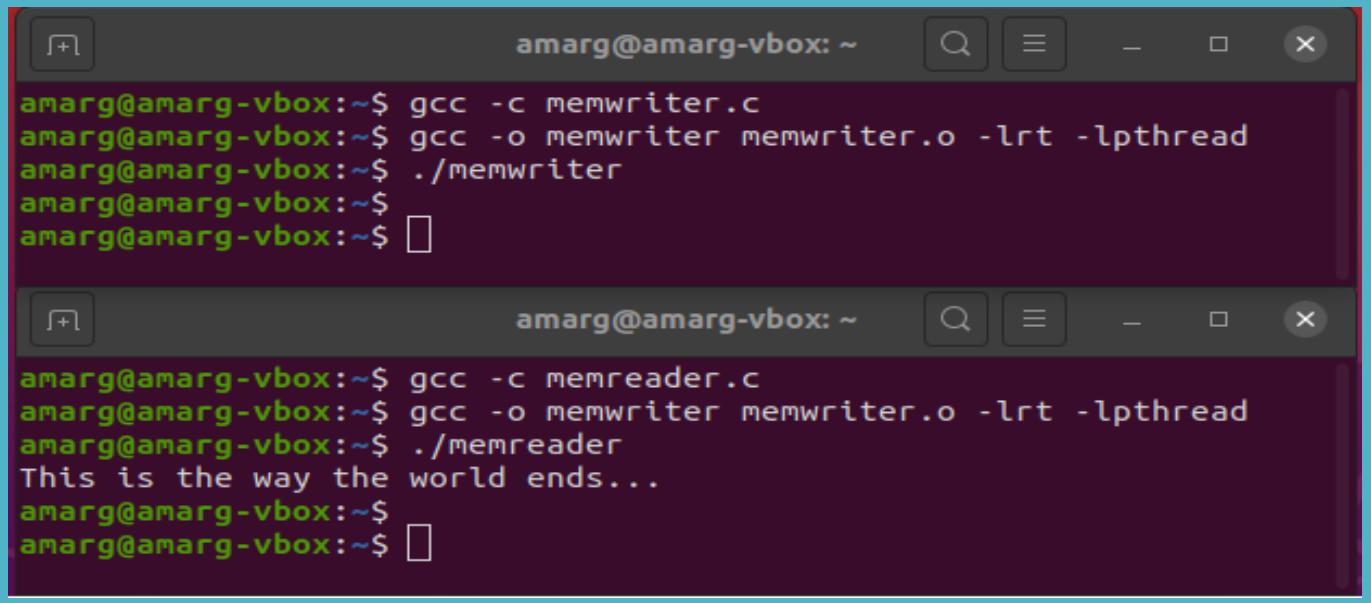
Αυτή η εφαρμογή εντός δώδεκα δευτερολέπτων από την έναρξη της προηγούμενης, διαβάζει από την κοινόχρηστη μνήμη το περιεχόμενό της και το εκτυπώνει στην οθόνη

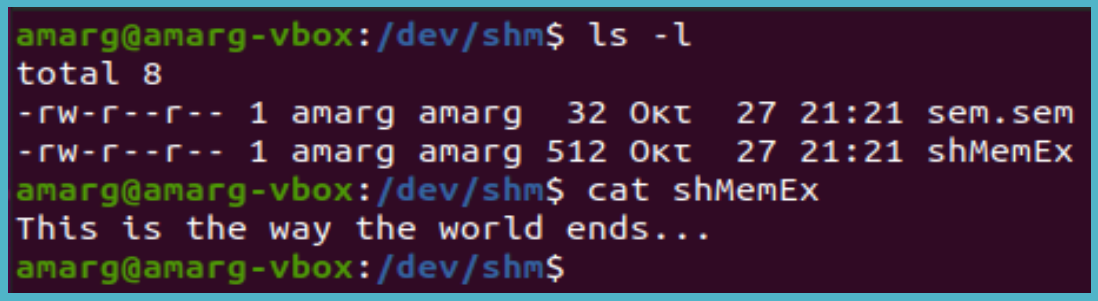
Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστης μνήμης και σημαφόρων

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.



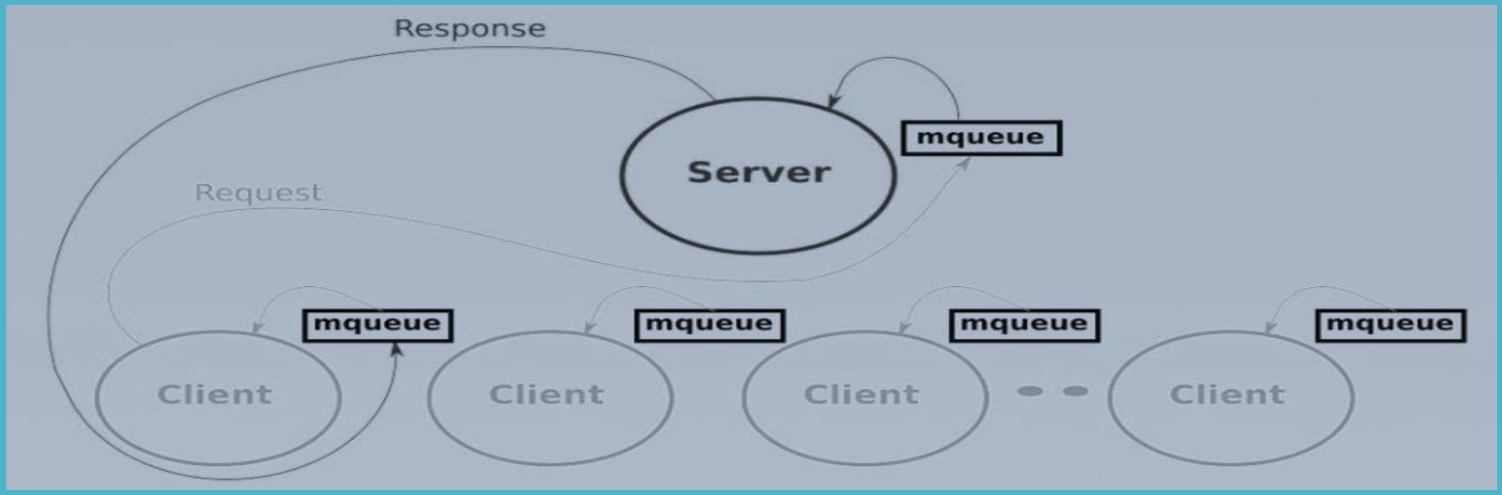


Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Οι ουρές μηνυμάτων (Message Queues, MQs) αποτελούν έναν μηχανισμό επικοινωνίας μεταξύ διεργασιών, ο οποίος χαρακτηρίζεται από τις επόμενες ιδιότητες:

- Επιτρέπει την επικοινωνία μεταξύ διεργασιών χρησιμοποιώντας κατάλληλα διαμορφωμένα μηνύματα με το ανάλογο περιεχόμενο.
- Ο παραλήπτης διαβάζει ένα μήνυμα κάθε φορά χωρίς να επιτρέπεται η ημιτελής ανάγνωση μηνύματος ή η πολλαπλή ανάγνωση του ιδίου μηνύματος.
- Υποστηρίζεται η περίπτωση πολλαπλών αναγνωστών και πολλαπλών συγγραφέων.
- Υποστηρίζεται η χρήση τιμών προτεραιότητας για τα μηνύματα.
- Υποστηρίζονται μηνύματα ειδοποίησης (notification messages).



Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Διαφορές μεταξύ αγωγών και ουρών μηνυμάτων

- Οι ουρές μηνυμάτων έχουν εσωτερική δομή ενώ στους αγωγούς ο συγγραφέας απλά γράφει bits. Για έναν αναγνώστη, οι διαφορετικές κλήσεις της write από διάφορους συγγραφείς είναι ταυτόσημες και ο προγραμματιστής πρέπει να διασφαλίσει πως γράφεται ή διαβάζεται ο σωστός αριθμός από bits, κάτι που καθιστά δύσκολο τον προγραμματισμό εφαρμογών για μηνύματα διαφορετικού μεγέθους.

- Στις ουρές μηνυμάτων έχουμε τιμές προτεραιότητας και τα μηνύματα είναι ταξινομημένα έτσι ώστε το παλαιότερο μήνυμα με την υψηλότερη τιμή προτεραιότητας να είναι πάντοτε πρώτο.

- Ο προγραμματιστής μπορεί να ορίσει τόσο το μέγιστο πλήθος μηνυμάτων που τοποθετούνται στην ουρά όσο και το μέγεθος κάθε τέτοιου μηνύματος.

- Σε αντίθεση με τους αγωγούς όπου η κατάστασή τους δεν είναι γνωστή, στις ουρές μηνυμάτων η εφαρμογή μπορεί να ανακτήσει τέτοιου είδους πληροφορίες, όπως το πλήθος των μηνυμάτων μιας ουράς, οι μέγιστες τιμές των παραμέτρων που έχουν οριστεί, καθώς και το πλήθος των διεργασιών που έχουν μπλοκάρει κατά την πραγματοποίηση μιας διαδικασίας αποστολής ή παραλαβής.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

- Στις ουρές μηνυμάτων μπορούν να χρησιμοποιηθούν όλες οι παραδοσιακές συναρτήσεις εισόδου / εξόδου που χρησιμοποιούνται και στα αρχεία (read, write, open, select, κ.τ.λ) αν και διαθέτουν και δικές του συναρτήσεις οι οποίες ξεκινούν με το πρόθεμα mq_ (από το message queue) και είναι δηλωμένες στο αρχείο **mqueue.h**.

- Οι συναρτήσεις διαχείρισης μίας ουράς μηνυμάτων στο POSIX ομαδοποποιούνται σε τρεις διαφορετικές κατηγορίες:

  - Συναρτήσεις διαχείρισης της ουράς μηνυμάτων:

    mq_open, mq_close, mq_unlink

  - Συναρτήσεις εισόδου / εξόδου:

    mq_send, mq_receive

  - Βοηθητικές συναρτήσεις:

    mq_setattr, mq_getattr, mq_notify

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

- Η δημιουργία και το άνοιγμα μίας νέας ουράς μηνυμάτων ή το άνοιγμα μιας υπάρχουσας ουράς μηνυμάτων γίνεται με τη συνάρτηση

mqd_t mq_open (const char * name, in oflag,
mode_t mode, struct mq_attr * attr)

ή την πιο απλή εκδοχή της **mqd_t mq_open (const char * name, in oflag)**

Το όρισμα name όπως και στην περίπτωση της κοινόχρηστης μνήμης έχει τη μορφή /somename, το oflag έχει υποχρεωτικά ακριβώς μία από τις τιμές O_RDONLY, O_WRONLY ή O_RDWR και προαιρετικά κάποια από τις τιμές O_CLOEXEC, O_CREAT και O_NONBLOCK.

Το όρισμα mode αναφέρεται στα δικαιώματα πρόσβασης π.χ. 0644, ενώ attr είναι μία δομή τύπου mq_attr – εάν λάβει τιμή NULL χρησιμοποιούνται προεπιλεγμένες τιμές για τις ιδιότητες της ουράς.

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Διαπιστώνουμε λοιπόν πως η ουρά μηνυμάτων έχει περιορισμένη χωρητικότητα η τιμή της οποίας ελέγχεται από τις συναρτήσεις διαχείρισης των ιδιοτήτων της ουράς.

Παράδειγμα χρήσης της mq_open

```
// Create new MQ, exclusive,for writing
mqd = mq_open("/mymq",O_CREAT| O_EXCL | O_WRONLY,0600, NULL);
// Open existing queue for reading
mqd = mq_open("/mymq", O_RDONLY);
```

Απομάκρυνση του ονόματος name της ουράς μυνημάτων

int mq_unlink (const char * name)

Το όνομα απομακρύνεται όταν η χρήση της ουράς μηνυμάτων έχει ολοκληρωθεί από όλους τους χρήστες που τη χρησιμοποιούν για επικοινωνία δεδομένων.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Αποστολή μηνύματος στην ουρά μηνυμάτων → Πραγματοποιείται με τη συνάρτηση

int mq_send (mqd_t mqdes, const char * msg_ptr,
size_t msg_len, unsigned int msg_prio);

όπου mqdes ο περιγραφέας της ουράς μηνυμάτων, msg_ptr ένας δείκτης στα bytes που σχηματίζουν το μήνυμα, msg_len το μέγεθος του μηνύματος και msg_prio η προτεραιότητα του μηνύματος η οποία λαμβάνει μη αρνητική ακέραια τιμή, με την τιμή 0 να περιγράφει τη μικρότερη δυνατή προτεραιότητα.

Το μέγεθος του μηνύματος msg_len πρέπει να είναι μικρότερο ή ίσο της τιμής της παραμέτρου mq_msgsize της ουράς. Η αποστολή μηνύματος μηδενικού μεγέθους, είναι επιτρεπτή.

Η συνάρτηση mq_send μπλοκάρει όταν η ουρά μηνυμάτων είναι γεμάτη και δεν μπορεί να φιλοξενήσει άλλα μηνύματα. Εάν θέλουμε να λειτουργήσει η συνάρτηση σε non-blocking mode θα πρέπει στη συνάρτηση mq_open να χρησιμοποιήσουμε το flag O_NONBLOCK.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Παραλαβή μηνύματος από την ουρά μηνυμάτων → Πραγματοποιείται με τη συνάρτηση

ssize_t mq_receive (mqd_t mqdes, char * msg_ptr,
size_t msg_len, unsigned int * msg_prio);

η οποία απομακρύνει από την ουρά μηνυμάτων το παλαιότερο μήνυμα με την υψηλότερη προτεραιότητα και το τοποθετεί στην περιοχή μνήμης που προσδιορίζεται από τον msg_ptr, το μέγεθος της οποίας θα πρέπει να είναι μεγαλύτερο ή ίσο της τιμής της παραμέτρου mq_msgsize της ουράς.

Εάν η τιμή της προτεραιότητας δεν είναι NULL, αυτή επιστρέφεται στο όρισμα msg_prio. Η συνάρτηση mq_receive μπλοκάρει όταν επιχειρεί να παραλάβει μήνυμα από μία κενή ουρά. Για λειτουργία σε non blocking mode χρησιμοποιείται όπως και πριν το flag O_NONBLOCK.

Σε περίπτωση επιτυχούς εκτέλεσης, η συνάρτηση mq_receive επιστρέφει το πλήθος των bytes που υπάρχουν στο μήνυμα που παρέλαβε, ενώ αντίθετα η συνάρτηση mq_send επιστρέφει μηδενική τιμή. Σε περίπτωση αποτυχίας, αμφότερες οι συναρτήσεις επιστρέφουν την τιμή -1 με την μεταβλητή errno να τίθεται στην κατάλληλη τιμή σφάλματος.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Παραδείγματα χρήσης των συναρτήσεων mq_send και mq_receive

```
mqd_t mqd;
mqd = mq_open("/mymq",O_CREAT | O_WRONLY, 0600, NULL);
char *msg = "hello world";
mq_send(mqd, msg, strlen(msg), 0);
```

```
const int BUF_SIZE = 1000;
char buf[BUF_SIZE];
unsigned int prio;
...
mqd = mq_open("/mymq", O_RDONLY);
nbytes = mq_receive(mqd, buf,BUF_LEN, &prio);
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Ειδοποίηση για την άφιξη μηνύματος προς παραλαβή

int mq_notify (mqd_t mqdes, const struct sigevent * sevp);

Μέσω της συνάρτησης mq_notify η διεργασία γνωστοποιεί πως επιθυμεί να ειδοποιηθεί όταν ένα νέο μήνυμα φτάσει σε μία κενή ουρά μηνυμάτων. Σε αυτή τη συνάρτηση χρησιμοποιείται η δομή

```
union sigval {          /* Data passed with notification */
    int     sival_int;         /* Integer value */
    void   *sival_ptr;         /* Pointer value */
};

struct sigevent {
    int          sigev_notify; /* Notification method */
    int          sigev_signo;  /* Notification signal */
    union sigval sigev_value;  /* Data passed with notification */

    void       (*sigev_notify_function) (union sigval); /* Function used for thread notification (SIGEV_THREAD) */
    void        *sigev_notify_attributes; /* Attributes for notification thread (SIGEV THREAD) */
    pid_t        sigev_notify_thread_id;/* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Ορισμός και λήψη τιμών ιδιοτήτων

int mq_getattr (mqd_t mqdes, struct mq_attr * attr);

int mq_setattr (mqd_t mqdes,
const struct mq_attr * newattr, struct mq_attr * newattr);

Οι παραπάνω συναρτήσεις επιστρέφουν 0 στην περίπτωση επιτυχούς εκτέλεσης και -1 σε περίπτωση αποτυχίας, αρχικοποιώντας κατάλληλα και τη μεταβλητή errno.

Η δομή mq_attr όπως σχολιάσαμε και προγουμένως, έχει τη μορφή

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

```
#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
```

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

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;
  mqdes = mq_open ("/mqueue1", O_RDWR | O_CREAT, 0664, NULL);
  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);
  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); }
  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); }
  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;}
  mq_close (mqdes);
  return (0); }
```

A

B

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Αρχικά η ουρά μηνυμάτων είναι κενή και το τμήμα κώδικα **A** δεν εκτελείται, αφού δεν υπάρχουν μηνύματα προς παραλαβή.

Εκτελείται λοιπόν το τμήμα κώδικα **B**, μέσω του οποίου αποστέλλονται στην ουρά μηνυμάτων το μέγιστο πλήθος μηνυμάτων που αυτή μπορεί να φιλοξενήσει που είναι ίσο με 10 (η προεπιλεγμένη τιμή του πεδίου mq_maxmsg της δομής mq_attr). Η έξοδος της εφαρμογής σε αυτή την πρώτη εκτέλεση του κώδικα έχει τη μορφή

```
Max number of messages on the queue ==> 10 messages.
Size of messages on the queue ==> 8192 bytes.
0 messages are currently on the queue.
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.
```

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)
  if (errno == EBUSY)
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Αυτή η διαδικασία δημιουργεί στον κατάλογο /dev/mqueue το αρχείο mqueue1 τα περιεχόμενα του οποίου ακολουθούν στη συνέχεια.

```
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$
```

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.

Προκειμένου να διαβάσουμε τα 10 μηνύματα που στάλθηκαν στην ουρά μηνυμάτων, εκτελούμε την εφαρμογή για δεύτερη φορά. Σε αυτή τη δεύτερη εκτέλεση, επειδή υπάρχουν μηνύματα για παραλαβή, θα εκτελεστεί αρχικά το τμήμα Α του κώδικα προκειμένου η διεργασία να διαβάσει τα 10 αυτά μηνύματα και στη συνέχεια το τμήμα Β προκειμένου η διεργασία να αποστείλει στην ουρά 10 νέα μηνύματα.

Κατά συνέπεια, η έξοδος της εφαρμογής σε αυτή τη δεύτερη εκτέλεση θα έχει τη μορφή

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

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.

Μετά την παραλαβή των μηνυμάτων που υπάρχουν στην ουρά, η διεργασία αποστέλλει στην ουρά μηνυμάτων 10 νέα μηνύματα και κατά συνέπεια το αρχείο /dev/mqueue/mqueue1 είναι το ίδιο με πριν.

Σε αυτό το παράδειγμα για λόγους απλότητας το όρισμα attr τέθηκε στην τιμή NULL έτσι ώστε να χρησιμοποιηθούν οι προεπιλεγμένες τιμές (μέγιστο πλήθος μηνυμάτων ίσο με 10 με μέγεθος μηνύματος ίσο με 8192 bytes αλλά σε κάθε περίπτωση μπορούμε μέσω του ορίσματος attr να ορίσουμε τις τιμές των παραμέτρων της ουράς).

```
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:~$
```