

ΕΠΛ421 - Προγραμματισμός Συστημάτων



Διάλεξη 11: Είσοδος/Έξοδος Χαμηλού Επιπέδου (Low-Level I/O) Κεφάλαιο 3 (Stevens & Rago)

Κεφάλαιο 5 (Stevens & Rago για
επανάληψη)

Δημήτρης Ζεϊναλιπούρ



Περιεχόμενο Διάλεξης

- Διαχείριση Λαθών με την `<errno.h>`
- **Δίσκοι και RAID**
- Εισαγωγή στα **Αρχεία και Συστήματα Αρχείων** στο Unix (Τύποι Αρχείων, Partitions, i-nodes, blocks)
- **Μέθοδοι Επεξεργασίας Αρχείων:** Standard I/O vs. Χαμηλού Επιπέδου I/O
- **Χαμηλού Επιπέδου I/O (System Calls I/O):** Θεωρία και Πρακτική.
- Παραδείγματα Χρήσης



Διαχείριση Λαθών στην C

- **Πως διαχειρίζεστε τα λάθη στην C εάν αυτά είναι Run-time Errors? Με printf? debugging?**
- Είναι γενικά αποδεκτό ότι δεν μπορούμε να γνωρίζουμε την αιτία όλων των λαθών **εάν αυτά δεν οφείλονται στην λογική του προγράμματος μας.**
π.χ., ένα λάθος στο σύστημα αρχείων, ο δίσκος έχει χαλάσει, το αρχείο στο οποίο προσπαθούμε να γράψουμε είναι κλειδωμένο από άλλο χρήστη, κτλ...)

Nόμος του Murphy: “*If anything can go wrong, it will*”

- **Στόχος:** Εάν συμβεί κάποιο λάθος να δώσουμε στον χρήστη ένα μήνυμα λάθους το οποίο να αντικατοπτρίζει την πραγματική αιτία.

Διαχείριση Λαθών

#include <errno.h>



- Η βιβλιοθήκη **<errno.h>** : System Error Numbers (**/usr/include/errno.h**)
- Μετά από κάθε κλήση συνάρτησης, η errno περιέχει **ένα ακέραιο (errno)**, το οποίο εκφράζει τι λάθος έχει προηγηθεί.
- Υπάρχουν **περισσότερα από 100 errno** (ακέραιοι, δες “man errno”) οι οποίοι προσδιορίζουν τα διάφορα είδη λαθών.
- Το **errno** τίθεται από τις συναρτήσεις που καλούμε (π.χ., fopen).
- Εάν μια συνάρτηση δεν το θέτει ρητά τότε το **errno** διατηρεί την υφιστάμενη του τιμή μέχρι να το θέσει κάποια εντολή στο πρόγραμμα μας (errno=0;)
- Στην συνέχεια εκτυπώνουμε αυτά λάθη με την συνάρτηση:

void perror(char *str)

- Εκτυπώνει την συμβολοσειρά **str** και την περιγραφή του **errno**
- Εάν θέλετε να εκτυπώσετε κάποιο μορφοποιημένο str, χρησιμοποιήσετε την γνωστή sprintf(), π.χ.,
 - `char string[50]; int file_number = 0;`
 - `sprintf(string, "file.%d", file_number);`
 - `perror(string);`



Διαχείριση Λαθών

#include <errno.h>

- Τα ακόλουθα προγράμματα προσπαθούν να ανοίξουν ένα αρχείο σε write mode με την Standard I/O.
- Προφανώς υπάρχουν πάρα πολλά ενδεχόμενα λάθη: δικαιώματα πρόσβασης, προβλήματα πρόσβασης, το αρχείο δεν υπάρχει, κτλ.
- Το αριστερό πρόγραμμα δεν δίνει κάποιο κατατοπιστικό μήνυμα λάθους ενώ το δεξιά (με την perror) επιστρέφει την ακριβή αιτία.

```
#include <stdio.h>
int main () {
    FILE * pFile;
    char *filename=“/home/a.txt”;
    pFile = fopen (filename,”w”);
    if (pFile==NULL) {
        printf(“Unable to open %s”,
               filename),
        exit(1);
    }
    return 0;
}
```

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

int main () {
    FILE * pFile;
    char *filename=“/home/a.txt”;
    pFile = fopen (filename,”w”);
    if (pFile==NULL) {
        perror(filename);
        exit(1);
    }
    return 0;
}
```

Εάν δεν μπορεί να δημιουργηθεί το αρχείο τότε εκτυπώνει (κωδικό 13) :
\$./program
/home/a.txt: Permission Denied

Διαχείριση Λαθών

#include <errno.h>



- Μερικές, από τις πάρα πολλές, σταθερές που βρίσκονται μέσα στο αρχείο header **errno.h**.
 - EPERM: Operation not permitted
 - ENOENT (2): **No such file or directory**
 - EINTR: Interrupted system call
 - EIO: I/O Error
 - EBUSY: Device or resource busy
 - EEXIST: **File exists**
 - EINVAL: Invalid argument
 - EMFILE: **Too many open files**
 - ENODEV: No such device
 - EISDIR: **Is a directory**
 - ENOTDIR: Isn't a directory
 -

Διαχείριση Λαθών

Παράδειγμα Εκτέλεσης



```
/* File: errors_demo.c */  
#include <stdio.h> /* For fopen, printf */  
#include <errno.h> /* For errno variable */  
  
main()  
{  
    FILE *fp = NULL;  
    char *p = NULL;  
    int stat;  
  
    # Εντολή Α  
    fp = fopen("non_existent_file", "r");  
    if (fp == NULL) {  
        /* Check for error */  
        printf("errno = %d\n", errno);  
        perror("fopen");  
    }  
    Συνέχεια...  
}
```

Αποτέλεσμα Εκτέλεσης

errno = 2
fopen: No such file or directory

errno = 12
malloc: Not enough space

errno = 13
unlink: Permission denied

ΕΠΛ 421 – Προγρα

```
Συνέχεια... # Εντολή Β  
p = (char *) malloc(400000000); // ~400MB  
if (p == NULL) {  
    /* Check for error */  
    printf("errno = %d\n", errno);  
    perror("malloc");  
}  
# Εντολή Γ  
***** BE CAREFUL: unlink tries to remove a  
file so do not run this under root*/  
stat = unlink("/etc/motd");  
if (stat == -1) {  
    /* Check for error */  
    printf("errno = %d\n", errno);  
    perror("unlink");  
}  
}
```

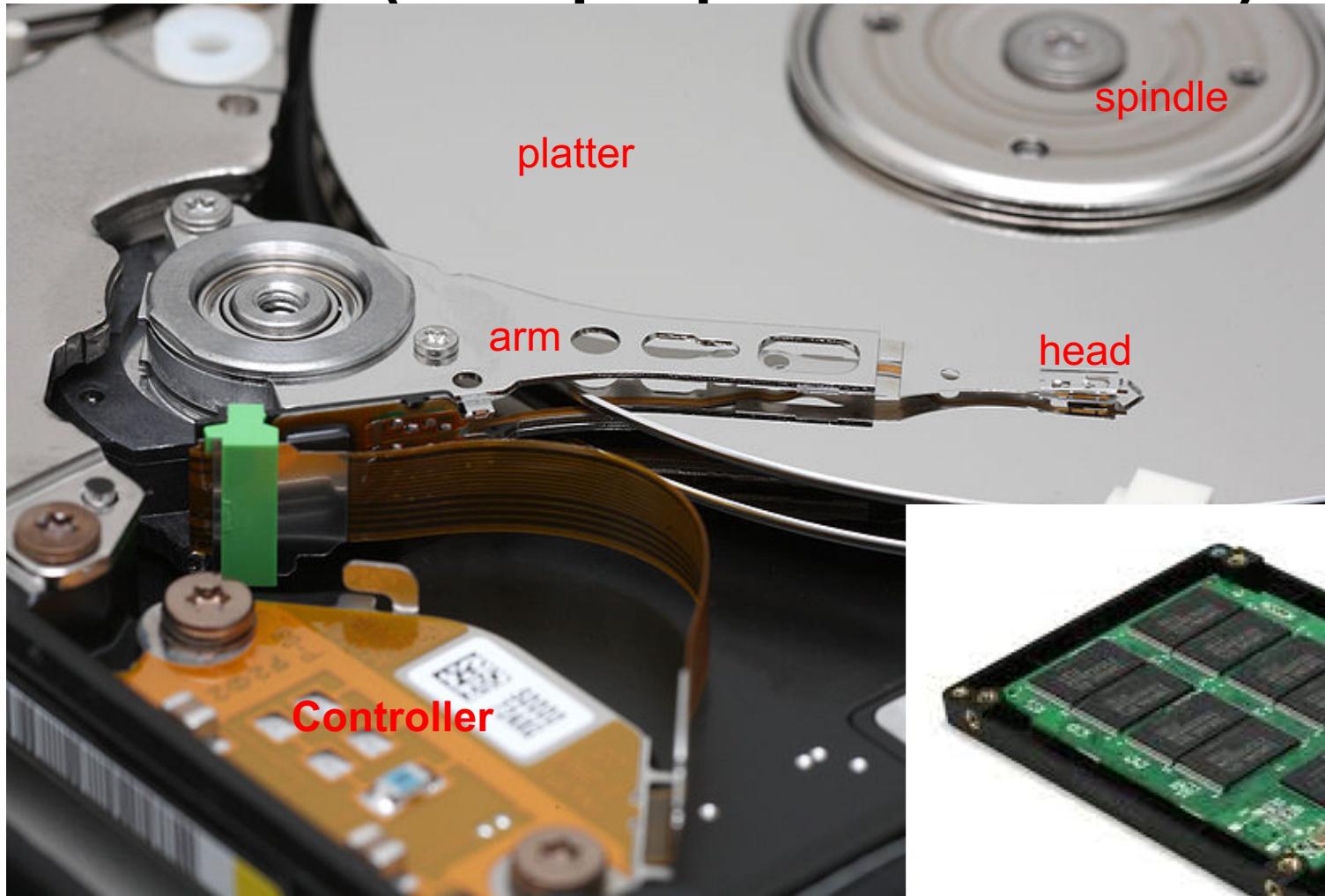
*/etc/motd: greeting displayed
whenever a user logs on to the system*

Αυτό ισχύει μόνο στον aias@, ενώ στις μηχανές
του εργαστηρίου δεν υπάρχει όριο στην malloc

Ιωπούρ ©



Magnetic Disks (Μαγνητικοί Δίσκοι)



HDD (Hard Disk Drive)

SSD (Solid State Disk)

RAID: Redundant Array of Independent* Disks (Εφεδρικές Συστοιχίες Ανεξαρτήτων Δίσκων)



- **Disk Array:** Arrangement of several disks that gives abstraction of a **Single, Large Disk!**
- **Goals:**
 - Increase **Performance (Επίδοση)**;
 - Why? Disk: a mechanical component that is inherently slow!
 - Increase **Reliability (Αξιοπιστία)**.
 - Why? Mechanical and Electronic Components tend to fail!



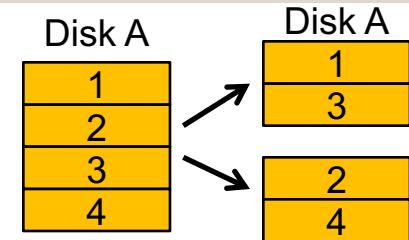
* Historically used to be **Inexpensive**



RAID: Key Concepts (RAID: Βασικές Αρχές)

A. **Striping (Διαχωρισμός)**: the splitting of data across more than one disk using a round-robin ($i \bmod \text{disks}$);

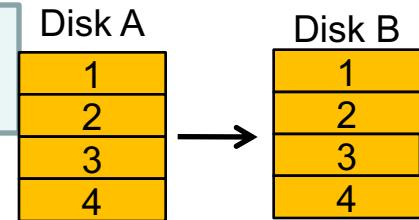
- Improving **Performance (Επίδοση)** and **Load Balancing** (εξισορρόπηση φόρτου)!
- **NOT** improving **Reliability (αξιοπιστία)**! (if one disk fails all data is useless)



A) Striping

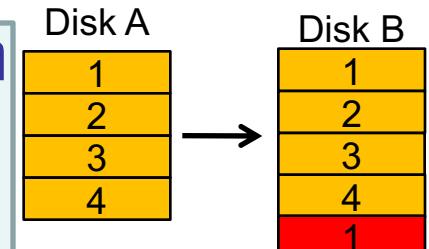
B. **Mirroring (Κατοπτρισμός) or Shadowing (Σκίαση)**: the copying of data to more than one disk

- Improving **Reliability (Αξιοπιστία)**!
- Improving **Read Performance** but **NOT Write Performance** (same as 1 disk!) / **Wasting space**



B) Mirroring

C. **Error Detection/Correction (Εντοπισμός/Διόρθωση Σφαλμάτων)**: the storage of additional information, either on same disks or on redundant disk, allowing the **detection (parity, CRC)** and/or **correction** (Hamming/Reed-Solomon) of failures.



C) Error Detection

□ RAID levels combine the above basic concepts: 0

(striping), 1 (mirroring), 4,5 (parity) □

Εισαγωγή στα Αρχεία και τα Συστήματα Αρχείων

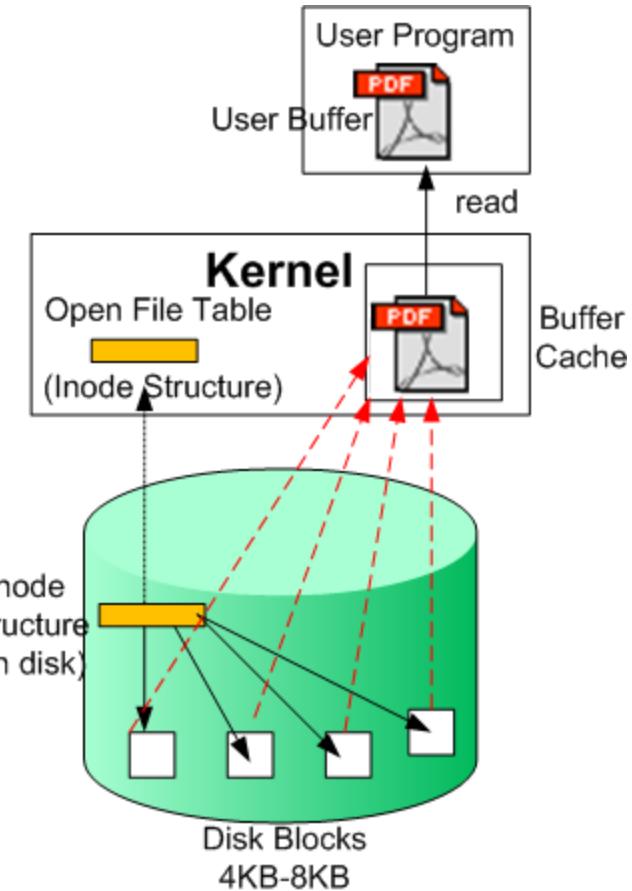


- Τώρα ας μελετήσουμε αναλυτικότερα τα **συστήματα αρχείων (filesystem)** και την **διαχείριση αρχείων στο UNIX**.
- **Αντίστοιχες** αρχές ισχύουν και σε άλλα λειτουργικά συστήματα, επομένως **MHN** θεωρήσετε ότι αυτά βρίσκουν εφαρμογή μόνο στο **UNIX**.



Τι είναι ένα Αρχείο;

- **Μια ακολουθία από bytes**, χωρίς καμιά δομή όσο αφορά το λειτουργικό σύστημα (δηλ., το ΛΣ δεν αντιλαμβάνεται την **σημασιολογία** αυτών των bytes).
- Οι βασικές λειτουργίες είναι αυτές της ανάγνωσης **read()** δεδομένων και της γραφής **write()** δεδομένων.
- **File Structure:** Η αναγνώριση της δομής των δεδομένων εναπόκειται αποκλειστικά στην εφαρμογή (π.χ. το Acrobat γνωρίζει την εσωτερική δομή ενός PDF αλλά όχι ενός DOC).



Τι είναι ένα Αρχείο: Παράδειγμα GIF



Ας δούμε πως **είναι κωδικοποιημένο** ένα αρχείο εικόνας PDF (Graphics Interchange Format).

\$ **hexdump -C filename.pdf # dump contents of file**

```
$ hexdump -C short-math-guide.pdf | more
```

00000000	25 50 44 46 2d 31 2e 33	0a 33 20 30 20 6f 62 6a	%PDF-1.3.3 0 obj
00000010	20 3c 3c 0a 2f 4c 65 6e	67 74 68 20 31 30 32 35	<<./Length 1025
00000020	35 20 20 20 20 20 0a 3e	3e 0a 73 74 72 65 61 6d	15 .>>.stream
00000030	0a 31 20 30 20 30 20 31	20 31 37 38 2e 33 35 37	.1 0 0 1 178.357
00000040	20 36 37 36 2e 32 33 31	20 63 6d 0a 42 54 0a 2f	676.231 cm.BT./

Magic number



Το σχήμα δείχνει ότι οι εικόνες (όπως και ΌΛΑ τα αρχεία) είναι ουσιαστικά μια σειρά από bytes αποθηκευμένα στον μαγνητικό δίσκο.



ΤÚΠΟΙ Αρχείο στο Unix

“Everything in Unix is a File”

- **Regular Files:** Αρχεία Δεδομένων (txt, pdf, jpg,...)
 - \$ ls -al ~/.profile
-rwxr----- 1 dzeina faculty 281 Jan 19 12:46 profile
- **Directory Files:** Αρχεία συστήματος για την διατήρηση της δομής του συστήματος αρχείων.
 - \$ ls -al ~ | grep public_html
drwxr-sr-x 8 dzeina faculty 4096 Feb 19 22:53 public_html
- **Character-Special Files:** Χρησιμοποιούνται για να αναπαραστήσουν Serial I/O συσκευές (terminals, printers και networks)
 - \$ ls -al `tty`
crw--w---- 1 dzeina tty 136, 1 Feb 21 11:36 /dev/pts/1
- **Block Special Files :** Χρησιμοποιούνται για να αναπαραστήσουν Μαγνητικούς Δίσκους, Cdroms, flash drives στα οποία η ανάγνωση/γραφή δεδομένων γίνεται σε Blocks (π.χ. 512 - 8192 bytes)
 - \$ ls -al /dev/hdc # το /dev/cdrom
brw-rw---- 1 root disk 22, 0 Feb 21 10:09 /dev/hdc
- **Symbolic link Files:** Το αρχείο είναι ένας σύνδεσμος που δείχνει σε κάποιο άλλο αρχείο (hard links are regular files)
 - \$ ln -s public_html/index.html index.lnk ; ls -al index.lnk
lrwxrwxrwx 1 dzeina faculty 22 Feb 21 12:07 index.lnk -> public_html/index.html
- **Named Pipes (FIFO):** Χρησιμοποιείται για επικοινωνία μεταξύ διεργασιών Θα τα δούμε αργότερα στο μάθημα.
 - \$ mkfifo pipef; ls -al | grep pipef
prw-r--r-- 1 dzeina faculty 0 Feb 21 11:42 pipef

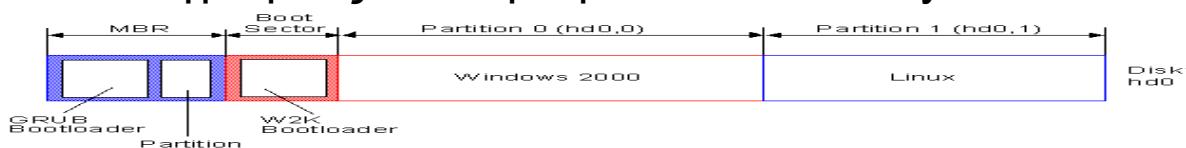


Πως αποθηκεύονται τα αρχεία;

- Τα αρχεία αποθηκεύονται πάνω στις συσκευές αποθήκευσης (Μαγνητικούς Δίσκους, FLASH, Ταινίες, Cdrom, DVDs, etc).
- File-System (Σύστημα Αρχείων):**

Μέρος του πυρήνα το οποίο υλοποιεί ένα σύνολο από δομές δεδομένων **κυρίας** και **δευτερεύουσας** μνήμης για την αποθήκευση, ιεραρχική οργάνωση, ανάκτηση και επεξεργασία δεδομένων.

- To file system σε μερικά Λ.Σ
 - Windows NT,2000,XP,Vista,7: FAT, FAT32, NTFS, Windows Future Storage (WinFS: File Org. with DBs!)
 - Linux Local Ext2, Ext3, Ext4, Linux Distributed: NFS, AFS, MacOS: Hierarchical FS+, Google's Non-kernel: MacFuse
 - CDROMs: ISO9960,
 - Φωτογραφικές & Κινητά με Flash Memory : FAT



ΕΙΔΑΤΑ - Ι ιρυγματισμός ζυστημάτων, Γιαν. Κυπριου - Διημήτρης Ζεϊναλίπούρ ©
|512 bytes|: Master Boot Record (or Volume Boot Record)





Disk Partitions

- Ο δίσκος χωρίζεται σε partitions
- Για να βρούμε τα partitions στο UNIX χρησιμοποιούμε την εντολή **df**

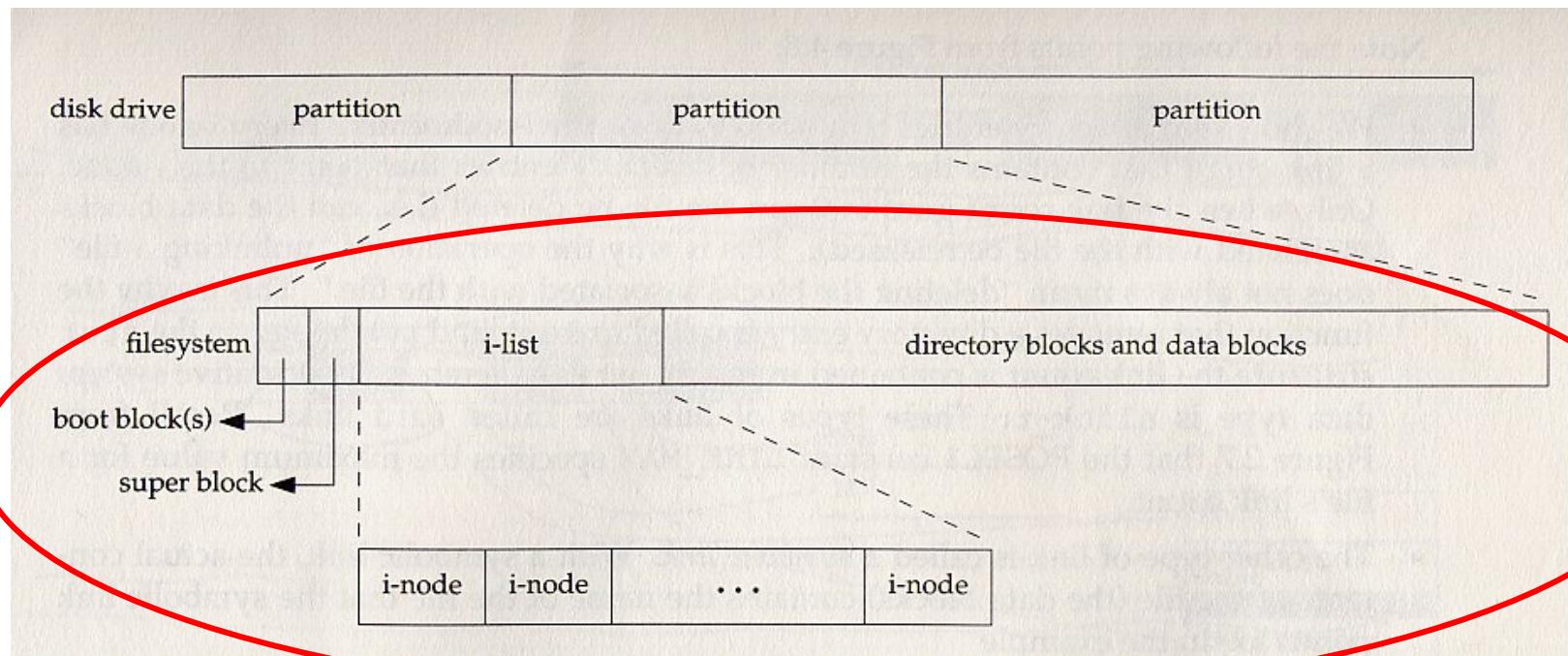
\$ df

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/mapper/VolGroup00-LogVolRoot	2031440	609380	1317204	32%	/
/dev/mapper/VolGroup00-LogVolTmp	1015704	94252	869024	10%	/tmp
/dev/mapper/VolGroup00-LogVolVar	2031440	391220	1535364	21%	/var
/dev/mapper/VolGroup00-LogVolUsr	5078656	3409680	1406832	71%	/usr
/dev/mapper/VolGroup00-LogVolOpt	2031440	1378252	548332	72%	/opt
/dev/mapper/VolGroup00-LogVolUsrLocal	507748	36197	445337	8%	/usr/local
/dev/sda1	101086	23619	72248	25%	/boot
tmpfs	517108	0	517108	0%	/dev/shm
csfs7.cs.ncy.ac.cy:/home/students	164288512	129185280	35103232	79%	/home/students
csfs1.cs.ncy.ac.cy:/home/faculty	278673504	242194208	27323520	90%	/home/faculty



Partitions

- Διαφορετικά partitions μπορούν να έχουν διαφορετικά συστήματα αρχείων.
- Ένα partition στο UNIX έχει την πιο κάτω μορφή:



Super block (Filesystem Metadata)

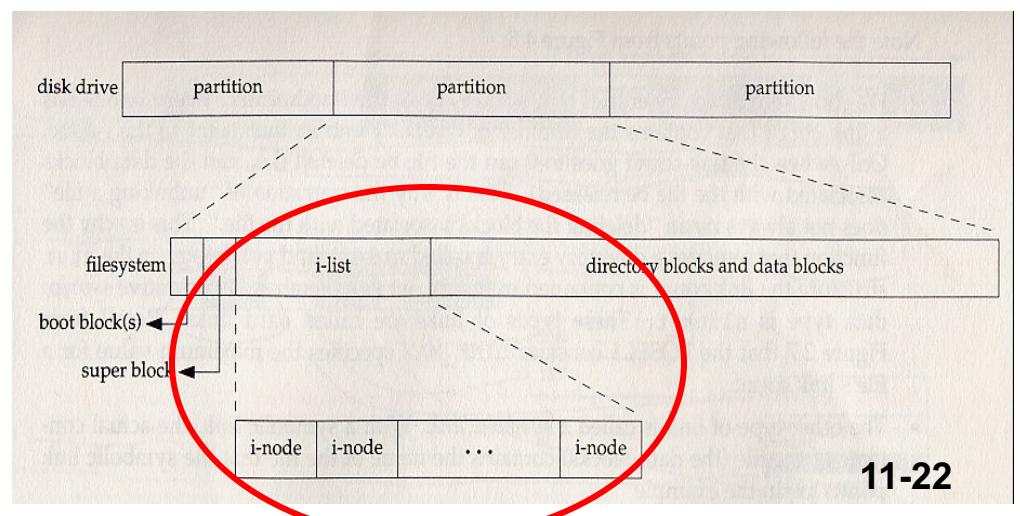
File system type, Size, Status

Information about other metadata structures



i-nodes

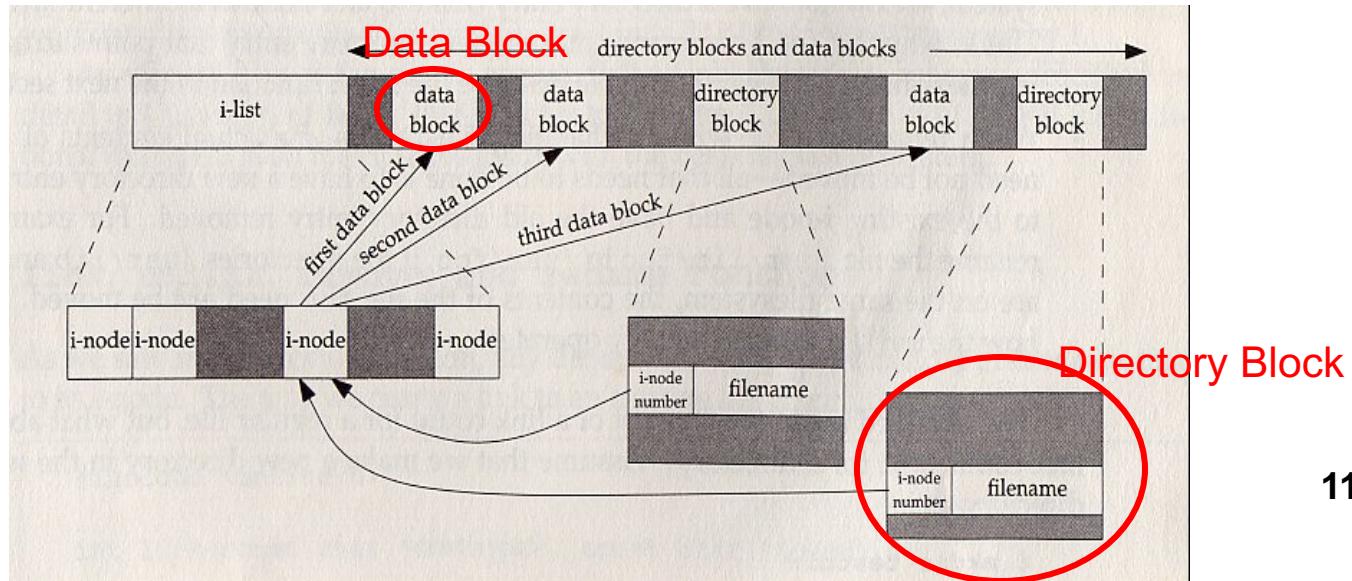
- Στην αρχή κάθε partition βρίσκεται μια λίστα με i-nodes, τα οποία αναγνωρίζουν μοναδικά τα αρχεία του συγκεκριμένου partition.
- Ένα **i-node** περιέχει τις πληροφορίες (μια συλλογή από διευθύνσεις δίσκου) έτσι ώστε να γνωρίζει ο πυρήνας από πού να ανακτήσει τα data/directory blocks κάποιου αρχείου
- Για να βρείτε το i-node number ενός αρχείου δώστε την εντολή "**ls -i**".
 - Από ένα πρόγραμμα μπορούμε να βρούμε το i-node μέσω του system call **stat()**...αργότερα....





Data/Directory Blocks

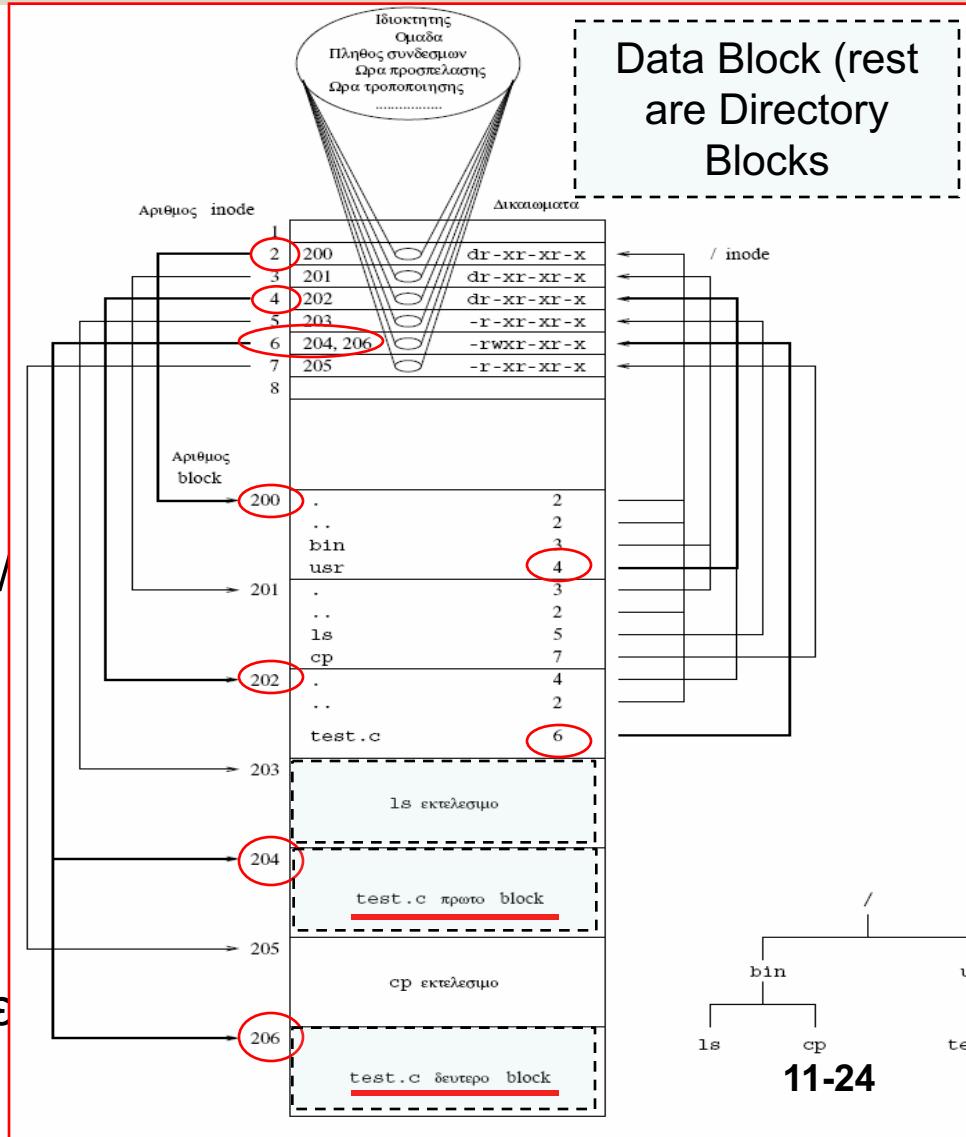
- Ένα αρχείο (regular ή directory) αποτελείται από μια σειρά "blocks" και ονομάζεται "**data block**" και "**directory block**" αντίστοιχα.
- To **data-block** αποθηκεύει
 - Μέρος των πραγματικών δεδομένων ενός αρχείου.
- To **directory-block** αποθηκεύει
 - I-node number και filename για κάθε αρχείο ή υποκατάλογο.





Παράδειγμα Εύρεσης Αρχείου με i-nodes

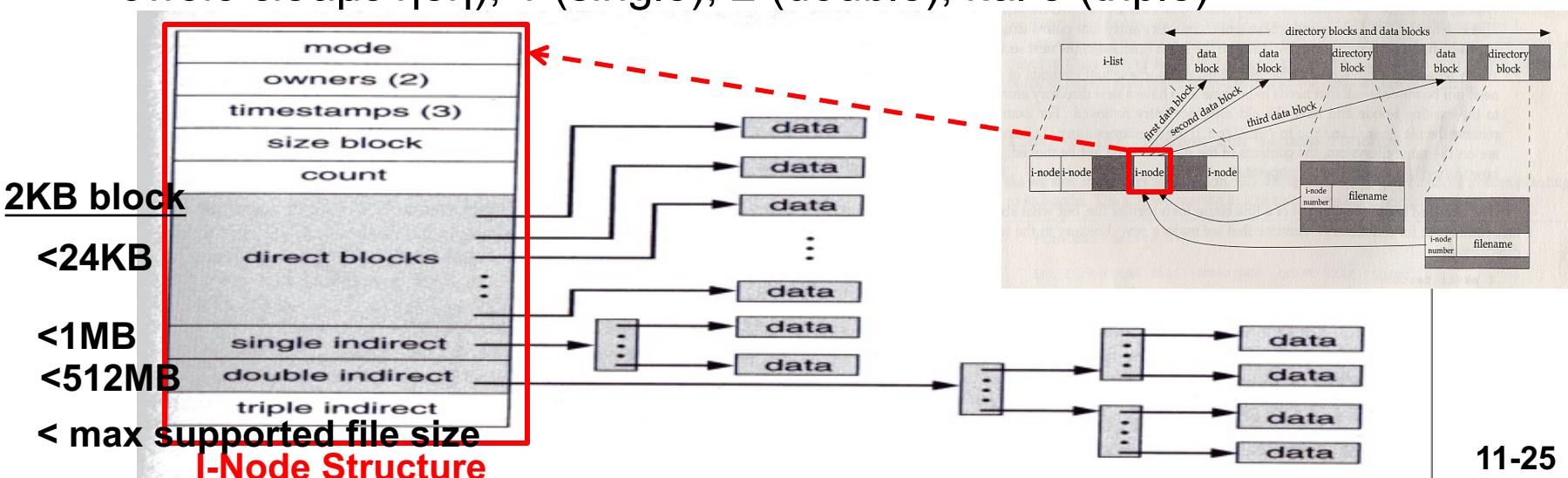
- Πως βρίσκει το λειτουργικό σύστημα (το υποσύστημα αρχείων) τα περιεχόμενα ενός αρχείου /usr/test.c ?
- Ξεκινά από το directory block με i-node 2 (πρώτο διαθέσιμο i-node στο i-list, το οποίο αντιστοιχεί στο / folder)
- Από εκεί βρίσκει ότι το usr έχει ως i-node to 200.
- Στην συνέχεια
=> 4 => 202 => 6 =>
- Τέλος βρίσκει ότι τα data blocks με i-nodes 204 και 206 περιέχουν τα δεδομένα του αρχείου.





Παράδειγμα Εύρεσης Αρχείου με i-nodes

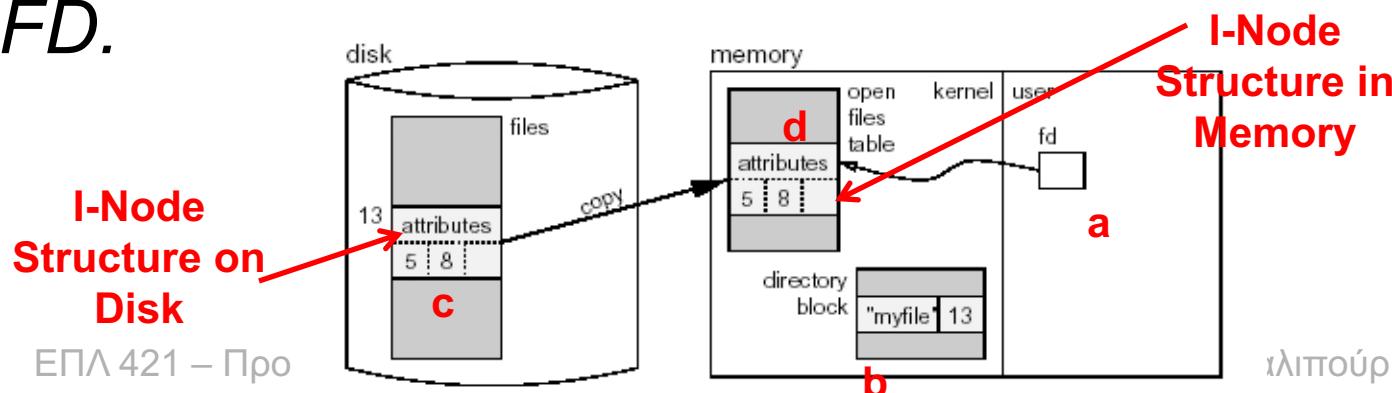
- Στο προηγούμενο παράδειγμα το αρχείο **/usr/test.c** με i-node 6 περιείχε μονάχα **2 blocks** (με i-node 204 και 206).
- Τι γίνεται εάν ένα αρχείο έχει πολλά blocks;**
- Υπάρχει αρκετός χώρος για να αποθηκευτούν **όλα τα i-nodes** των blocks που συσχετίζονται με το αρχείο;
- Το Υποσύστημα Αρχείων χρησιμοποιεί ένα **ιεραρχικό σχήμα** το οποίο αποτελείται από δένδρα δεικτών βάθους 0 (direct, αυτό το οποίο είδαμε ήδη), 1 (single), 2 (double), και 3 (triple)



(περισσότερα για την δομή αυτή στην επόμενη διάλεξη)

Μέθοδοι Επεξεργασίας Αρχείων

- Τώρα θα δούμε μεθόδους επεξεργασίας του περιεχομένου των αρχείων.
- Το χαμηλού επιπέδου I/O διεκπεραιώνεται εκτελώντας **system calls (κλήσεις συστήματος)**.
- I/O System Call:** κλήσεις προς τον πυρήνα του λειτουργικού συστήματος ο οποίος μας επιστρέφει ένα **File Descriptor (FD)** (ορολογία Unix) ή **File Handle** (ορολογία Windows).
- Όλη η επεξεργασία στην συνέχεια γίνεται μέσω του FD.





Διεργασίες και Αρχεία

Μέσα στην Διεργασία

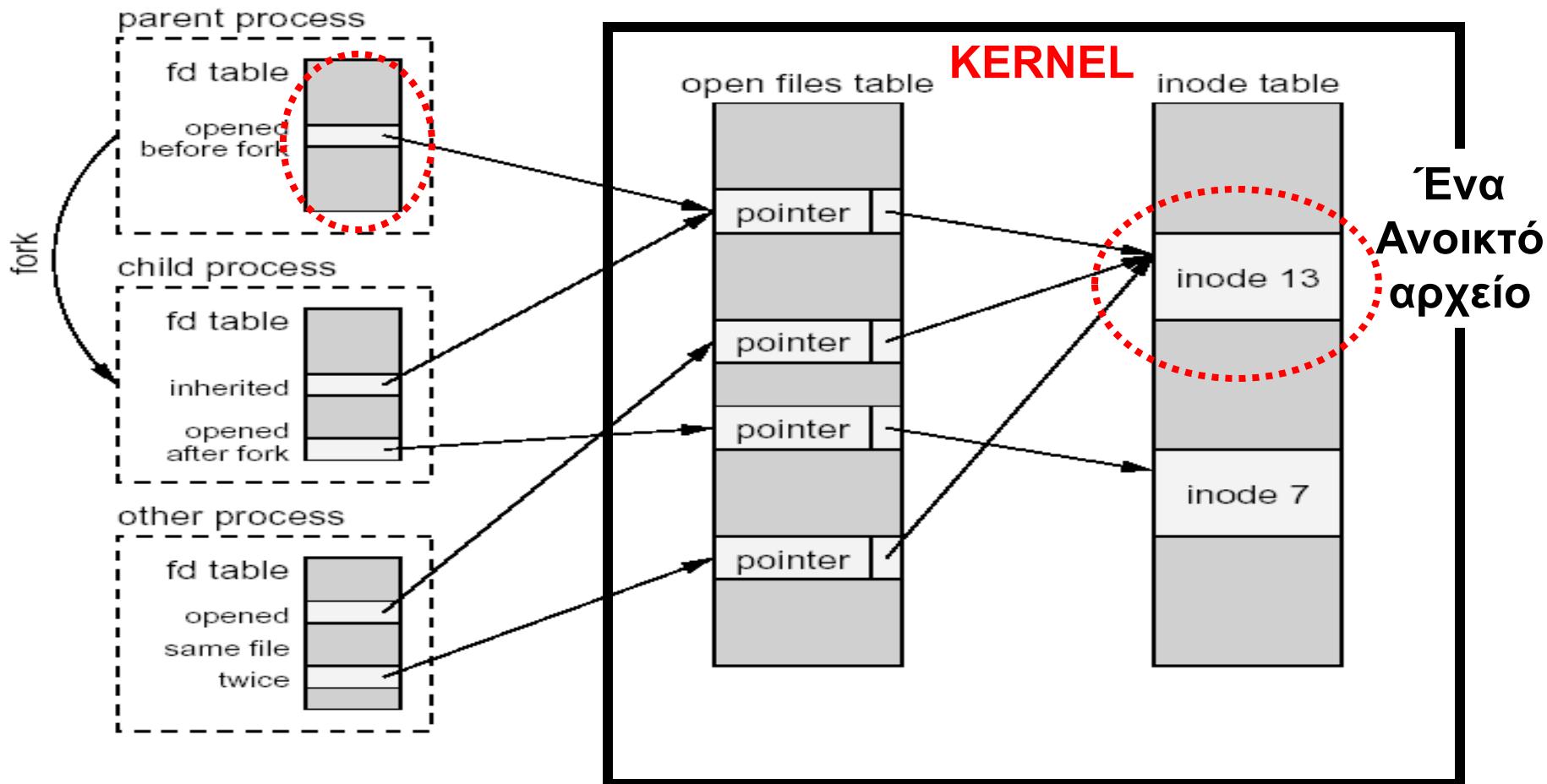
- **File Descriptor Table:** Ένα για κάθε διεργασία. Κάθε εγγραφή του, ή οποία αναγνωρίζεται από το fd ($0, 1, \dots$), περιέχει ένα δείκτη στο open files table. (ελέγξετε το `/proc/$$/fd/`)

Μέσα στον Πυρήνα

- **Open Files Table:** Μια εγγραφή δημιουργείται όποτε εκτελέσουμε `open()`. Η εγγραφή περιέχει i) **δείκτη στο inode table**, ii) **offset** μέσα στο αρχείο (το επόμενο `read` εκτελείται από το offset); iii) **status** (π.χ. `read`, `write`, `append`) με το οποίο ανοίξαμε το αρχείο.
- **Inode Table:** Κάθε αρχείο εμφανίζεται μόνο μια φορά σε αυτό τον πίνακα.



Διεργασίες και Αρχεία



Θα μελετηθεί αργότερα ξανά



Μέθοδοι Επεξεργασίας Αρχείων

Από τα προγράμματα εφαρμογών μπορούμε να ανακτήσουμε τα δεδομένα των αρχείων με δύο τρόπους:

α) Standard I/O Library (Procedure) Calls

fopen, fclose, fread, putc, readc, etc.

... αυτά τα οποία χρησιμοποιούσατε μέχρι τώρα στα διάφορα μαθήματα προγραμματισμού με χρήση δομής FILE. Αυτή η μέθοδος συνιστάται μόνο για τα regular αρχεία (όχι άλλους τύπους)

β) Κλήσεις Συστήματος (System Calls)

(e.g., open, close, read, write, lseek) μέσω File Descrip. (int)

- Τα οποία μοιάζουν με **library calls (κλήσεις συναρτήσεων βιβλιοθήκης)**, με την διαφορά ότι τα system calls εκτελούνται μέσα στο **kernel space** αντί στο **user space**.
- Επομένως κλήση σε ένα system call δεν σημαίνει απλά ότι θα κάνουμε **jump σε μια άλλη εντολή** ορισμένη αλλού στο πρόγραμμα αλλά ότι θα γίνει **context switching** (εναλλαγή διεργασιών), έτσι ώστε να εκτελέσει το kernel την αίτηση σας.
- Βέβαια στο τέλος όλες οι **λειτουργίες επεξεργασίας** ενός αρχείου εκτελούνται από τον πυρήνα, ωστόσο με τις κλήσεις βιβλιοθήκης μπορεί να γίνεται αυτό με κάποια καθυστέρηση (δηλ., buffering)



Standard IO vs. Systems Calls

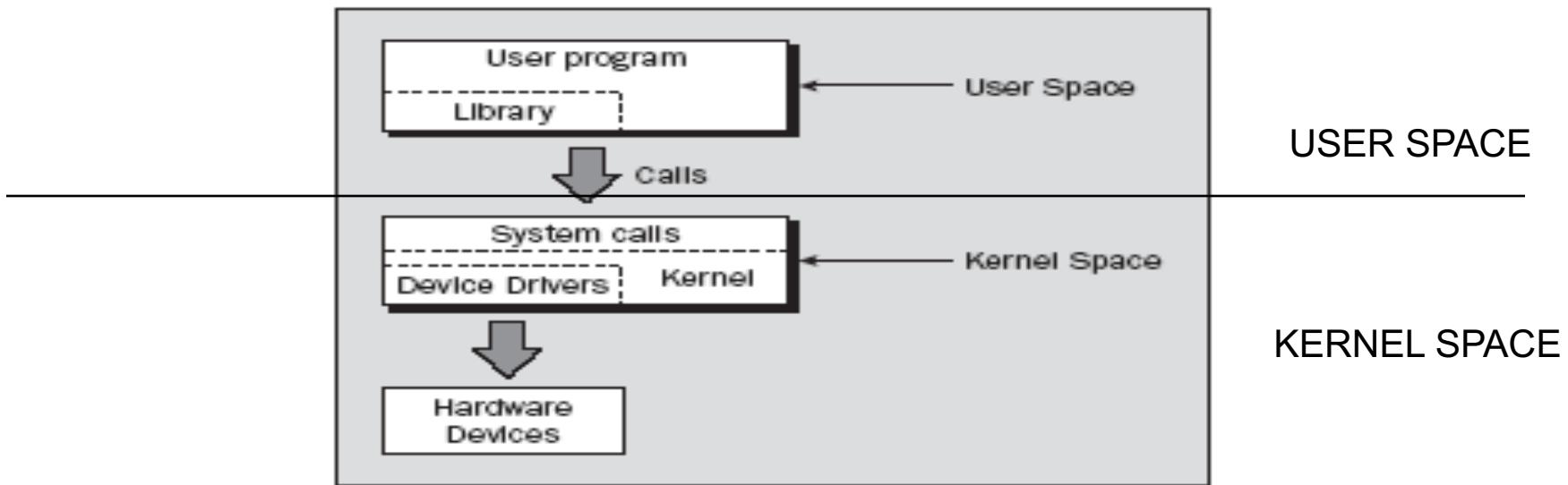
- Τα Standard I/O `stdio.h` (ANSI C99) είναι μια βιβλιοθήκη συναρτήσεων για πρόσβαση σε αρχεία από προγράμματα C (χρησιμοποιείται πέρα από το UNIX)
 - Τα Standard I/O κάνουν **κλήση συνάρτησης της ίδιας της διεργασίας** ενώ τα System Calls καλούν λειτουργίες του πυρήνα.
 - Τα Standard I/O calls **υλοποιούνται και αυτά μέσω** των System Calls!
 - Τα Standard I/O **είναι πιο εύκολα**
 - Η Standard I/O δημιουργεί **ένα Stream (FILE *)** ενώ τα System Calls ένα **File Descriptor**
 - Η Standard I/O κάνει buffering (BUFSIZE την stdio.h).

.... αλλά δίδουν λιγότερη ελευθερία ☺, ή οποία χρειάζεται στο Systems Programming
- **System Call Standard I/O call**

		Επανάληψη Standard I/O Calls: Κεφάλαιο 5 – Stevens & Rago Man stdio
open	<code>fopen</code>	
close	<code>fclose</code>	
read/write	<code>fread/fwrite</code>	// used for binary I/O chunks of bytes rather char or line
	<code>getchar/putchar</code>	// read a character from STDIN
	<code>getc/putc</code>	// ...>.....>.....from file
	<code>fgetc/fputc</code>	// identical to getc/putc
	<code>gets/puts</code>	// read a string from STDIN
	<code>fgets/fputs</code>	// >> > > > > from file
	<code>scanf/printf</code>	// read formatted input from STDIN
	<code>fscanf/fprintf</code>	// >> > > > > from file
lseek	<code>fseek</code>	// move to a specific location in a file



Πλεονέκτημα των System Calls



Πλεονέκτημα System Call: Πλήρες Έλεγχος!

- Υλοποίηση Λειτουργιών που δεν προσφέρονται από τα Standard I/O (locking, buffering, IPC)
- **Έλεγχος του Buffering:** Το οποίο είναι πολύ σημαντικό σε συστήματα βάσεων δεδομένων όπου θέλουμε να χειριστούμε το buffering μέσα στην ίδια την εφαρμογή και όχι το standard I/O.
- **Interprocess Communication:** Όταν προγραμματίζεται επικοινωνία μεταξύ διεργασιών με χρήση σωλήνων και υποδοχών (θα μας απασχολήσουν αργότερα στο μάθημα ...)



Μειονεκτήματα των System Calls

- **Είναι Πολύ Ακριβά!** : Ενώ ένα procedure call θέλει μόνο μερικές εντολές μηχανής ενώ ένα system call θα κάνει τα εξής:
 - Η υπό εκτέλεση διεργασία διακόπτεται και το state της φυλάγεται.
 - To Kernel (ΛΣ) παίρνει τον έλεγχο του CPU και εκτελεί το system call.
 - To Kernel (ΛΣ) διακόπτεται και το state της φυλάγεται ενώ ταυτόχρονα δίδει τον έλεγχο πίσω στην διεργασία.
- **Όμως, η ορθολογιστική χρήση των system calls οδηγεί σε αυξημένη Απόδοση κάτι το οποίο είναι πολύ σημαντικό.**
- **System Dependent:** Διαφορετικές Υλοποιήσεις Λ.Σ υποστηρίζουν διαφορετικές λειτουργίες. Οπόταν τίθεται θέμα **μεταφερσιμότητας** (Portability) του πηγαίου κώδικα με τα system calls.
- **Δυσκολότερος Προγραμματισμός:** Με την standard I/O, η βιβλιοθήκη οργανώνει τις κλήσεις προς τον πυρήνα με έτσι τρόπο που τα δεδομένα θα μεταφέρονται σε **πλήρη blocks** από τον δίσκο στην μνήμη, ενώ με τα System Calls πρέπει να το χειριστούμε μόνοι μας.



System Calls for I/O

- Υπάρχουν 5 βασικά system calls τα οποία παρέχει το Unix για file I/O (ελέγξτε το man –s 2 open)
 - int open(char *path, int flags [, int mode]) ;
 - int close(int fd) ;
 - int read(int fd, void *buf, int size) ;
 - int write(int fd, void *buf, int size) ;
 - off_t lseek(int fd, off_t offset, int whence) ;
- Για να έχουμε πρόσβαση σε αυτές τις συναρτήσεις μέσω των προγραμμάτων εφαρμογών χρειάζεται να συμπεριλάβουμε τις ακόλουθες βιβλιοθήκες

// file control options:

#include <fcntl.h>

// standard symbolic constants and types for unix

#include <unistd.h>



1) System Calls: Open()/Close()

Άνοιγμα Αρχείου για ανάγνωση/γραφή δεδομένων

Returns: File Descriptor if OK, -1 on error

int open(const char *path, int flags [, int mode])

Returns: 0 if OK, -1 on error

int close(int filedes);

- 'path ': Απόλυτο η σχετικό μονοπάτι προς το αρχείο
- Τα **flags** και **mode** δηλώνουν πως θέλουμε να χρησιμοποιήσουμε το αρχείο (επόμενη διαφάνεια)
- Εάν ο πυρήνας αποδεχθεί την αίτηση μας, τότε δημιουργείται ο μικρότερος δυνατό ``**file descriptor (FD)**'' (το οποίο είναι μια ακέραια τιμή για αρχείο). Οποιεσδήποτε μελλοντικές αναφορές στο αρχείο γίνονται μέσω του FD. (Θα εξηγηθεί αναλυτικότερα στην συνέχεια)
- Το Linux 2.4.22 θέτει ένα hard limit από 1,048,576 FDs ανά διεργασία. Αυτό ορίζεται από την σταθερά OPEN_MAX στην βιβλιοθήκη limits.h (όπως την εντολή \$ulimit -n)
- Εάν η open επιστρέψει -1, τότε δεν ικανοποιήθηκε η αίτηση και μπορούμε να ελέγξουμε την τιμή της μεταβλητής "errno" για να βρούμε γιατί (με την perror()).
- Θυμηθείτε ότι όταν ανοίγει ένα πρόγραμμα τότε τρεις FDs είναι ήδη ανοικτοί (0:stdin, 1:stdout, 2:stderr).



Open/close: Απλό Παράδειγμα

```
#include <fcntl.h>
#include <stdio.h> // for printf
int main() {
    int fd1, fd2;

    if(( fd1 = open("foo.txt", O_RDONLY) ) < 0) {
        perror("opening file");
        exit(1);
    }

    printf("The File Descriptor is : %d", fd1);

    if (close(fd1) < 0) {
        perror("closing file");
        exit(1);
    }
}
```

ΕΠΛ 421 – Προγραμματισμός Συστημάτων, Παν. Κύπρου - Δημήτρης Ζεϊναλίπούρ ©



1) System Calls: Open()/Close()

Flags (σταθερές μέσα στη fcntl.h) : Υποχρεωτικές και προαιρετικές επιλογές.

Υποχρεωτικές: **O_RDONLY** (open for reading only), **O_WRONLY** (writing), **O_RDWR** (read/write)

Προαιρετικές: **O_APPEND** (append to the end of file), **O_TRUNC** (if file exists in WR or RDWR then truncate its size to 0), **O_CREAT** (create if file does not exist – we have to set MODE variables – see below),

Mode (σταθερές sys/stat.h) : Προσδιορίζει τα δικαιώματα εάν δημιουργούμε αρχείο.

To mode είναι ακέραιος (ιδία λογική με Unix file permissions).

Mode

S_IRUSR: Read permission, owner
S_IWUSR: Write permission, owner
S_IXUSR: Execute permission, owner
S_IRGRP: Read permission, group
S_IWGRP: Write permission, group
S_IXGRP: Execute permission, group
S_IROTH: Read permission, others
S_IWOTH: Write permission, others
S_IXOTH: Execute permission, others

Σημείωση: Τα **Flags** και το **Mode** εκφράζονται μπορούν να δοθούν σαν διάζευξη συμβολικών ονομάτων, π.χ.,

`open("test.txt", O_WRONLY | O_CREAT,
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);`

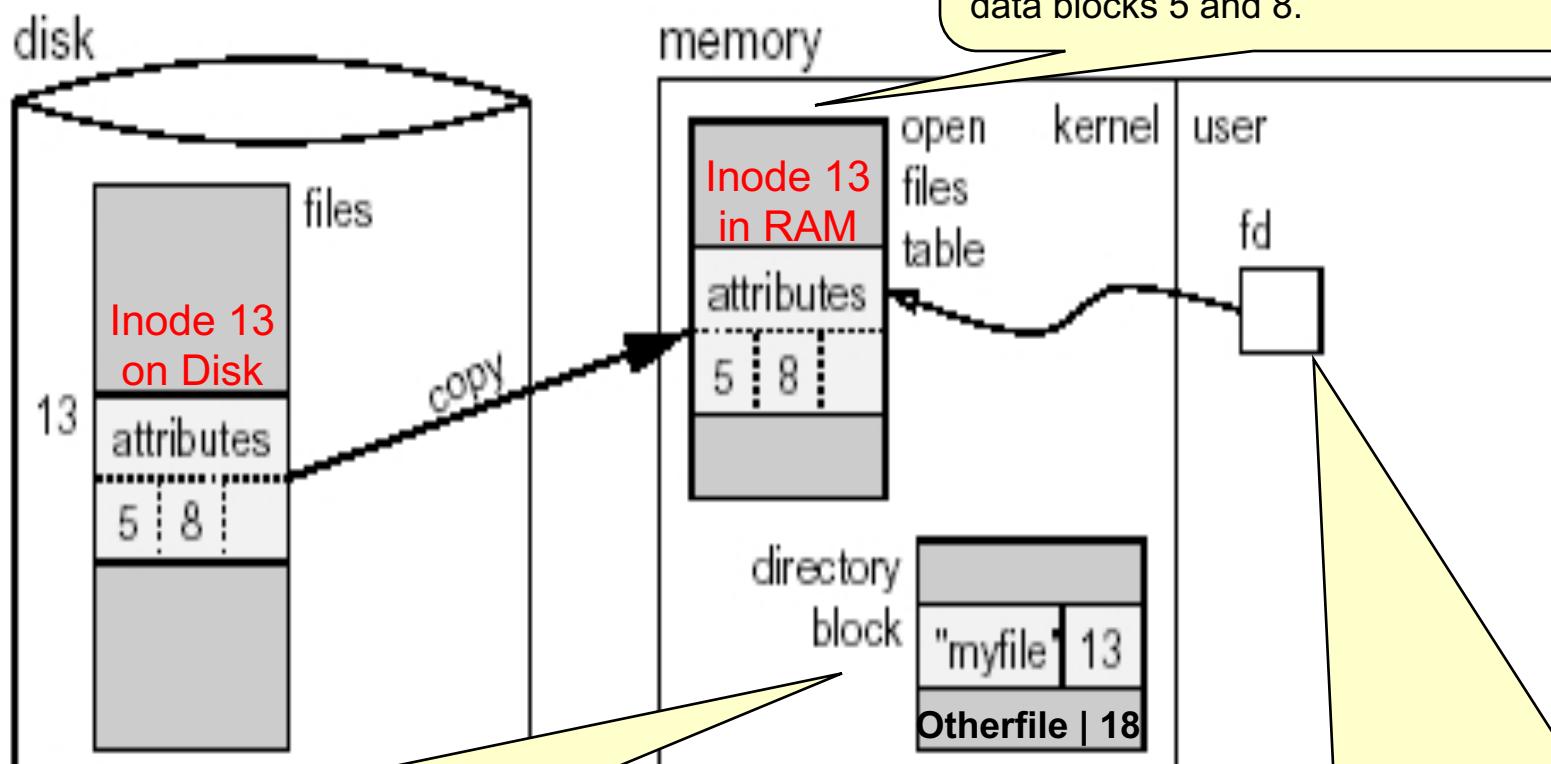
Εξήγηση: To ORing θετει τα bits :
`rwxrwxrwx = 110100100`
`(δηλ. 100000000 | 010000000 | ...)`

1) System Calls: Open()/Close()

Μηχανισμός Εκτέλεσης



Opening “myfile” (Inode:13)



A) The file system reads the current directory, and finds that “myfile” is represented internally by **entry 13** in the list of files maintained on the disk.

C) User’s **access rights** are checked (through Inode attributes) and the user’s variable **fd** is made to point to the allocated entry in the open files table

2) System Call: Creat()

Δημιουργία Αρχείου



```
int creat(const char * filename, int mode);  
ή εναλλακτικά open(filename, O_WRONLY | O_CREAT |  
O_TRUNC, mode);  
# Returns: File Descriptor for O_WRONLY if OK, -1 on error
```

- Ο Dennis Ritchie ερωτήθηκε κάποτε πιο είναι το μοναδικό πράγμα που έχει μετανιώσει για την C. Αυτός απάντησε “leaving off the ‘e’ on creat()”.
- Μειονέκτημα της creat() είναι ότι δεν μπορούμε να διαβάσουμε από το αρχείο που δημιουργήσαμε.
- Για να το διορθώσουμε και να έχουμε Δημιουργία + Read/Write χρησιμοποιήστε το ακόλουθο:
**open(filename, O_RDWR | O_CREAT | O_TRUNC,
mode);**

Creating Files on command line (Creating inode vs. inode+data)



\$ touch test.txt # Creating an inode

\$ stat test.txt # print test.txt inode info - will see in a while

File: 'test.txt'

Size: 0 Blocks: 0 IO Block: 1048576 regular empty file

Device: 3fh/63d Inode: 6670939443 Links: 1

...

\$ echo "" > test2.txt # Creating an inode+data

\$ stat test2.txt # print test2.txt inode info - will see in a while

File: 'test2.txt'

Size: 1 Blocks: 8 IO Block: 1048576 regular file

total size of all files in the list (measured in 512 B) => 4KB block size

...

\$ hexdump -C test.txt

\$ hexdump -C test2.txt

00000000 0a

ΕΠΛ 421 – Προγραμματισμός Συστημάτων, Παν. Κύπρου - Δημήτρης Ζεϊναλίπούρ ©

00000001

..

3) System Call: Read()

Ανάγνωση από κάποιο FD



*int read(int fd, void *buf, int size)*

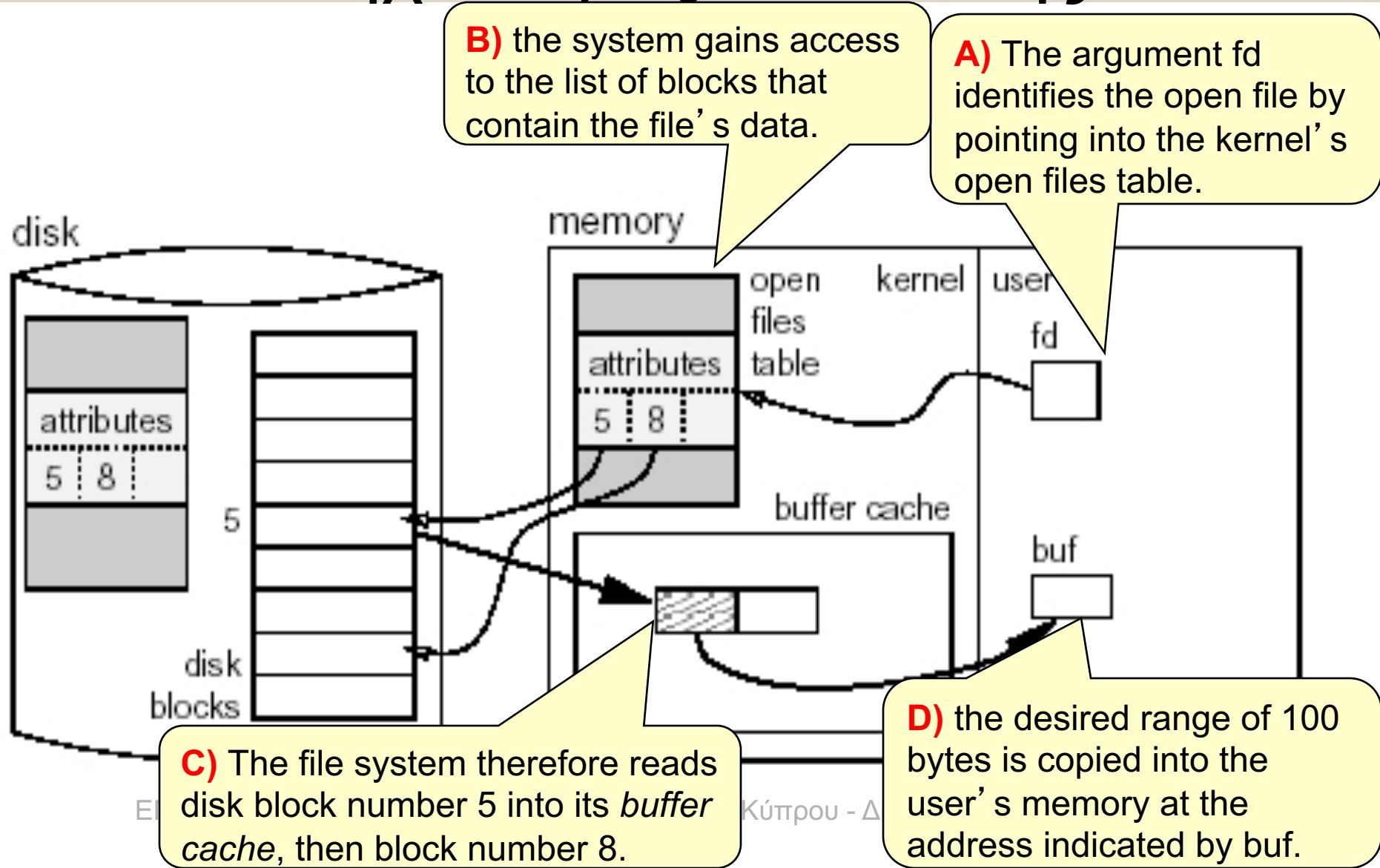
Returns: αριθμό bytes που διαβάστηκαν, 0:EOF, -1:ERROR

- Διαβάζει `size` bytes από την οντότητα (αρχείο, συσκευή, άκρο σωλήνα ή υποδοχή) που αντιστοιχεί στον περιγράφει `fd` και τα αντιγραφεί στο `buf`.
- Περιμένει (blocking wait) μέχρι συμπληρωθεί το `buf` η μέχρι φτάσει το `EOF`.
- Προϋποθέτει ότι η εφαρμογή σας έχει δεσμεύσει αρκετό χώρο (malloc) στο `buf`. Τι θα συμβεί στην αντίθετη περίπτωση;
- Σημειώστε ότι το `buf` είναι *Posix-compliant (void *)* παρά τον παραδοσιακό ορισμό `char *buf`)



3) System Call: Read()

Μηχανισμός Εκτέλεσης



4) System Call: Write()

Ανάγνωση από κάποιο FD



*int write(int fd, void *buf, int size)*

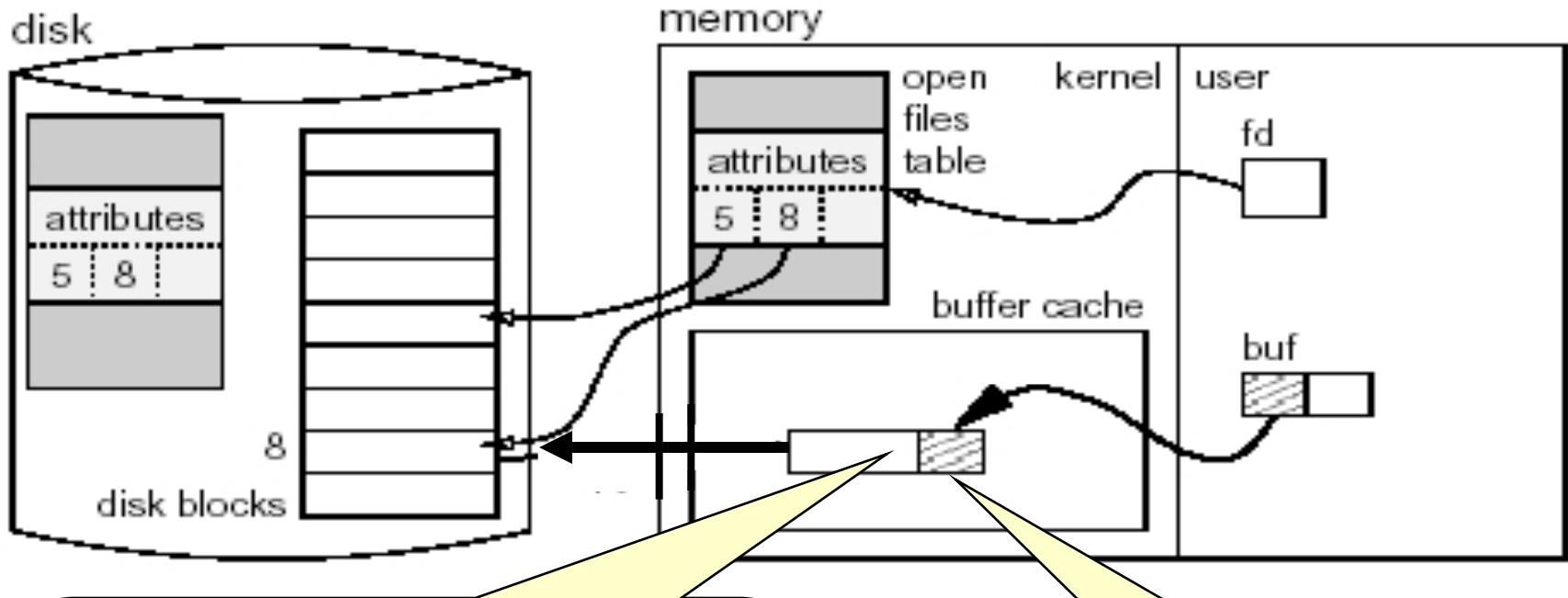
Returns: αριθμό bytes που γράφτηκαν, -1:ERROR

- **Γράφει** size bytes από το buf στην οντότητα (αρχείο, συσκευή, áκρο σωλήνα ή υποδοχή) που αντιστοιχεί στον περιγράφει fd.
- **Επιστρέφει** τον αριθμό από bytes που γραφτήκαν.
- Αυτό μπορεί να είναι λιγότερο από την αίτηση (**size**) εάν υπήρξε κάποιο πρόβλημα με τον file descriptor
 - Π.χ., size=100 αλλά η write καταφέρνει να γράψει μόνο 10.
- Εάν επιστρέψει 0, τότε δεν γράφτηκε τίποτα.
- Εάν επιστρέψει -1, τότε ελέγξετε τον κωδικό λάθους **errno**.



4) System Call: Write()

Μηχανισμός Εκτέλεσης



β) The kernel then might flush (best to be done by programmer) the newly written data to disk (assuming that block#8 has adequate space). Else a new block is allocated and the data is written there as well.

α) Our program calls `write(fd,buf, sizeof(buf));` and the data is copied to kernel space



Παράδειγμα 1: Υλοποίηση του **cat**

Να υλοποιήσετε σε C και με χρήση κλήσεων συστήματος, ένα απλό πρόγραμμα το οποίο να προσομοιώνει την εντολή **cat** (χωρίς παραμέτρους)

π.χ.

./mycat <filename

ls | ./mycat

./mycat (επαναλαμβάνει ότι γράφουμε)



Παράδειγμα 1: Υλοποίηση του cat

// Η βιβλιοθήκη stdio.h περιλαμβάνεται μόνο για την μεταβλητή BUFSIZE (αν κάναμε και printf/sccanf επίσης), η οποία παίρνει την πιο καλή τιμή (με βάση το υπάρχων σύστημα), παρά να την θέταμε μόνοι μας.

```
#include <unistd.h> // STDIN_FILENO, STDOUT_FILENO
#include <stdio.h> // BUFSIZE
#include <error.h> // perror
int main(int argc, char * argv []) {
    char buf[BUFSIZ]; // BUFSIZE 8192
    int n;

    while((n = read(STDIN_FILENO, buf, sizeof(buf))) > 0)
        if ( write(STDOUT_FILENO, buf, n) != n )
            perror ("write error");

    if (n < 0)
        perror("read error");
    return 0;
}
```

- Ερώτηση: Γιατί δεν χρειαστήκαμε να ανοίξουμε τους περιγραφείς FD#0 και FD#1;



Παράδειγμα 2: Read/Write Demo

Να υλοποιήσετε σε C και με χρήση κλήσεων συστήματος, ένα απλό πρόγραμμα το οποίο
i) δημιουργεί ένα, αρχείο, **ii) γράφει** κάποια συμβολοσειρά, **iii) κλείνει** το αρχείο,
iv) ανοίγει και γράφει κάποια άλλη συμβολοσειρά, **v) κλείνει** το αρχείο και τέλος **vi) διαβάζει** το αρχείο και το **εκτυπώνει** στην οθόνη. Κάθε βήμα να τυπώνει τον αριθμό των bytes.



Παράδειγμα 2: Read/Write Demo

```
/* File: io_demo.c */  
#include <stdio.h> /* For printf, BUFSIZE */  
#include <fcntl.h> /* For O_RDONLY, O_WRONLY,O_CREAT, O_APPEND */  
#include <string.h> /* strlen */  
main() {  
    int fd, bytes;  
    char buf[BUFSIZ];  
    char *filename = "temp.txt";  
    char *line1 = "First write. \n";  
    char *line2 = "Second write. \n";  
  
    // 0600 <==> S_IRUSR | S_IWUSR <==> -,rw-,---,---  
    if ((fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0600)) == -1) {  
        perror("open");  
        exit(1);  
    }  
    // Προσοχή: Γράφουμε μόνο το strlen(line1), αντί strlen(line1)+1 για να  
    // αποφύγουμε τα ανεπιθύμητα \0 ( “First write. \n\0 Second write. \n\0” )  
    bytes = write(fd, line1, strlen(line1)); /* Data out */  
    printf("%d bytes were written\n", bytes);  
    close(fd);  
}
```

Create file and enable us to write to the file

Ανεπιθύμητο



Παράδειγμα 2: Read/Write Demo

```
if ((fd = open(filename, O_WRONLY | O_APPEND)) == -1) {
    perror("open");
    exit(1);
}
bytes = write(fd, line2, strlen(line2)); /* Data out */
printf("%d bytes were written\n", bytes);
close(fd);

if ((fd = open(filename, O_RDONLY)) == -1) {
    perror("open");
    exit(1);
}
bytes = read(fd, buf, sizeof(buf)); /* Data in */
printf("%d bytes were read\n", bytes);
close(fd);
buf[bytes] = '\0';
printf("%s", buf);
}
```

```
$ ./a.exe
14 bytes were written
15 bytes were written
29 bytes were read
First write.
Second write.
```



Παράδειγμα 3: Υλοποίηση Concat Αρχείων

Να υλοποιήσετε σε C και με χρήση κλήσεων συστήματος, ένα απλό πρόγραμμα το οποίο να προσομοιώνει την λειτουργία της πιο κάτω εντολής κελύφους `cat file1 >> file2`

Το πρόγραμμα σας εκτελείται με τον ακόλουθο τρόπο.

`$concat file1 file2`



Παράδειγμα 3: Υλοποίηση Concat Αρχείων

```
/* File: append_file.c */
#include <fcntl.h> /* For O_RDONLY, O_WRONLY, O_CREAT, O_APPEND */
#include <unistd.h> // STDERR_FILENO
#include <string.h> // strlen

main(int argc, char *argv[])
{
    int n, fromFD, toFD;
    char buf[1024]={};

    if (argc != 3) {
        /* Check for proper usage */

        sprintf(buf, "Usage: %s from-file to-file", argv[0]) ;
        write(STDERR_FILENO, buf, strlen(buf));
        exit(1);
    }
}
```



Παράδειγμα 3: Υλοποίηση Concat Αρχείων

```
if ((fromFD = open(argv[1], O_RDONLY)) < 0) {
    /* Open from-file */
    perror("open");
    exit(1);
}

if ((toFD = open(argv[2], O_WRONLY | O_CREAT | O_APPEND, 0660)) < 0) {
    /* Open to-file */
    perror("open");
    exit(1);
}

while ((n = read(fromFD, buf, sizeof(buf))) > 0)
    if (write(toFD, buf, n) != n) { /* Copy data */
        perror("copy error");
    }

close(fromFD); /* Close from-file */
close(toFD); /* Close to-file */

}
```

Append mode

```
$ cat file1
First write.
$cat file 2
Second write.
$concat file1 file2
$cat file2
First write.
Second write.
```



5) System Call: lseek()

Τυχαία (Random) Μετακίνηση μέσα στο Αρχείο

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

Returns: new file offset if OK, -1 on error

- **off_t** δηλώνεται μέσα στο [sys/types.h](#)
- **Offset (από πού):** Ορίζει την θέση (position)
- **Whence :** Ορίζει πως χρησιμοποιείται το offset.
 - SEEK_SET: relative to beginning of file
 - SEEK_CUR: relative to the current position
 - SEEK_END: relative to end of file

- Η συνάρτηση lseek μας επιτρέπει να μετακινηθούμε μέσα στο αρχείο με τυχαίο τρόπο. (Random Access).
- Κάθε αρχείο έχει ένα «current file offset» το οποίο είναι ένας θετικός ακέραιος ο οποίος μετρά τον αριθμό από bytes από την αρχή του αρχείου.
 - Εάν ανοίξει το αρχείο το offset είναι 0.
 - Μετά από ανάγνωση m bytes, το offset γίνεται m.
- Το “l” seek προέρχεται από το “long” seek διότι το offset είναι ένας long integer.



5) System Call: `Iseek()`

Τυχαία (Random) Μετακίνηση μέσα στο Αρχείο

- Το `Iseek` δηλώνει το offset μέσα στο kernel χωρίς να προκαλέσει οποιονδήποτε I/O. Το I/O θα γίνει μόνο στο επόμενο write.
- Εάν εκτελέσουμε το πιο κάτω εντολή τότε η επόμενη γραφή (ή ανάγνωση) θα γίνει 100 bytes δεξιότερα από την παρούσα θέση:

`Iseek(filedesc, 100, SET_CUR);`

- Επομένως ένα αρχείο μπορεί να έχει «τρύπες» με κενά.
- Αυτές οι τρύπες είναι ουσιαστικά συνεχόμενα 0 (NULs) και καταλαμβάνουν 1 byte ανά Nul.
- Το offset μπορεί να πάρει θετικές και αρνητικές τιμές, εάν και εφόσον οι συσκευές το υποστηρίζουν (π.χ. σε ένα αρχείο γίνεται άλλα όχι σε ένα socket).



Παράδειγμα 4: Υλοποίηση Απλού Database

Να υλοποιήσετε σε C και με χρήση κλήσεων συστήματος, μια απλή δυαδική βάση δεδομένων ή οποία γράφει πέντε εγγραφές της μορφής

```
typedef struct person{  
    int id;  
    char sex;  
    char name[40];  
} __attribute__((packed)) PERSON;
```

To __attribute__((packed)) του σημαίνει ότι δεν θέλουμε ο GCC compiler να προσθέσει padding στην δομή για να την κάνει align στην μνήμη.

Εάν δεν το χρησιμοποιήσουμε πρέπει να κάνουμε τις εγγραφές των στοιχείων μια-μια

συνεχόμενα σε ένα αρχείο user.db, στην συνέχεια αφήνει 3 εγγραφές κενές και μετά γράφει ακόμη 5 εγγραφές. Τέλος διαβάζει από το αρχείο όλες τις εγγραφές και τις τυπώνει στην οθόνη.



Παράδειγμα 4: Υλοποίηση Απλού Database

```
main(int argc, char *argv[])
{
    int n, fd, i;
    PERSON p;
    char *filename="users.db";
    int offset;

    if ((fd = open(filename, O_RDWR | O_CREAT, 0660)) < 0) {
        /* Open to-file */
        perror("open db");
        exit(1);
    }

    // set the values in the p structure
    p.sex = 'm';
    strcpy(p.name, "Costas");

    // write 5 tuples to the file
    for (i=0; i<5; i++) {
        p.id = i;
        write(fd, (void *) &p, sizeof(p));
    }
}
```

Παράδειγμα 4: Υλοποίηση Απλού Database



```
// seek the file descriptor three sizeof(p) positions to the right (from SEEK_CUR)
Iseek(fd, 3*sizeof(p), SEEK_CUR);

// write another 5 tuples to the file
for (i=0; i<5; i++) {
    p.id = i;
    write(fd, (void *) &p, sizeof(p));
}

// rewind the file descriptor to the beginning of the file
Iseek(fd, 0, SEEK_SET);

// Now read and print the respective tuples
while((n = read(fd, (void *) &p, sizeof(p))) > 0) {
    if ( sizeof(p) != n )
        perror ("read error");
        printf("<%d,%c,%s>\n", p.id, p.sex, p.name);
}

close(fd);
```

Παράδειγμα 4: Υλοποίηση Απλού Database



```
$dbtest  
<0,m,Costas>  
<1,m,Costas>  
<2,m,Costas>  
<3,m,Costas>  
<4,m,Costas>  
<0, ,>  
<0, ,>  
<0, ,>  
<0,m,Costas>  
<1,m,Costas>  
<2,m,Costas>  
<3,m,Costas>  
<4,m,Costas>
```

**Το μέγεθος του αρχείου
είναι 585 bytes
(13 records * 45 bytes)**

Παράδειγμα 5: Δημιουργία Άδειου Αρχείου Μεγάλου μεγέθους



```
#include <fcntl.h> /* For O_RDONLY, O_WRONLY, O_CREAT, O_APPEND */  
#include <unistd.h> // STDERR_FILENO
```

```
typedef struct person{  
    int id;  
    char sex;  
    char name[40];  
} __attribute__((packed)) PERSON;
```

```
int main(int argc, char *argv[]) {  
    int fd;    PERSON p;  
    char *filename="users.db";  
    if ((fd = open(filename, O_RDWR | O_CREAT, 0660)) < 0) {  
        perror("open db"); return 1;  
    }  
    lseek(fd, 10000*sizeof(p), SEEK_CUR);  
    write(fd, "", 1);  
    close(fd);  
    return 0;  
}
```

```
$ stat users.db  
File: 'users.db'  
Size: 45001      Blocks:  
8 IO Block: 1048576 ...
```

Δημιουργία Άδειου Αρχείου Μεγάλου Μεγέθους στο Shell



- A) dd - convert and copy a file
 - Create inode + initialize with '\0' the data blocks

```
$ df
```

# Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/home/faculty	1091260416	876950528	214309888	81%	/home/faculty

```
$ dd if=/dev/zero of=./test.img bs=4k iflag=fullblock,count_bytes  
count=1000000000 # 1GB
```

244140+1 records in

244140+1 records out

1000000000 bytes (1.0 GB) copied, **12.6773 s, 78.9 MB/s**

```
$ stat test.img #
```

File: 'test.img'

The disk has shrunk by 1GB!

Size: **1000000000** Blocks: **1953128**

```
$ df
```

/home/faculty	1091260416	877926400	213334016	81%	/home/faculty
---------------	------------	-----------	------------------	-----	---------------

Δημιουργία Άδειου Αρχείου Μεγάλου Μεγέθους στο Shell



- **B) truncate – make a file smaller/larger**
 - Make/Modify i-node size but doesn't allocate disk pages on disk (slim file)

```
$ df
```

```
home/faculty      1091260416 876950528 214309888  81% /home/faculty
...
```

```
$ truncate -s 10G test.img
```

```
stat test.img    File: 'test.img'   Size: 10737418240  Blocks: 2048
```

```
$ df
```

```
home/faculty      1091260416 876950528 214309888  81% /home/faculty
...
```