

Εργαστήριο Λειτουργικών Συστημάτων

Μάθημα 6^{ου} Εξαμήνου,

Τομέας Λογικού και Υπολογιστών

Ιωάννης Χατζηγιαννάκης

Σημειώσεις Μαθήματος
Ενότητα 5



Σημειώσεις Μαθήματος – 5^η Ενότητα

Σύστημα Αρχείων στο MINIX 3

Λειτουργίες Συστήματος Αρχείων

Δομή Συστήματος Αρχείων

Ανάγνωση Αρχείων

Διαχείριση Block

Διαχείριση i-Node

Διαχείριση Superblock, Πίνακα Ανοικτών Αρχείων, File Locks

Αρχικοποίηση Συστήματος Αρχείων

Άνοιγμα, Δημιουργία Ανάγνωση Αρχείων



Σύστημα Αρχείων στο Λ.Σ. MINIX 3

- ▶ Όπως όλα τα Λ.Σ. και το MINIX 3 προσφέρει ένα σύστημα αρχείων για την αποθήκευση πληροφοριών
- ▶ Μπορεί να δεσμεύει / αποδεσμεύει αποθηκευτικό χώρο για τα αρχεία
- ▶ Να διαχειρίζεται τα blocks του δίσκου και να απελευθερώνει αποθηκευτικό χώρο
- ▶ Να διασφαλίζει την ασφάλεια των δεδομένων
- ▶ Πρόκειται για ένα 'μεγάλο' πρόγραμμα γραμμένο σε C -- τρέχει εξ' ολοκλήρου στο user space
- ▶ Οι διεργασίες που θέλουν να διαβάσουν/γράψουν αρχεία στέλνουν μηνύματα στο σύστημα αρχείων (file system)
 - ▶ Το σύστημα αρχείων επεξεργάζεται το μήνυμα, εκτελεί τις απαραίτητες ενέργειες και επιστρέφει την απάντηση



Λίστα Μηνυμάτων (1)

Message	Input parameters	Reply Value
access	File name, access mode	Status
chdir	Name of new working directory	Status
chmod	File name, new mode	Status
chown	File name, new owner, group	Status
chroot	Name of new root directory	Status
close	File descriptor of file to close	Status
creat	Name of file to be created, mode	File descriptor
dup	File descriptor (for dup2, two fds)	New file descriptor
fcntl	File descriptor, function code, arg	Depends on function
fstat	Name of file, buffer	Status
lctl	File descriptor, function code, arg	Status



Λίστα Μηνυμάτων (2)

Message	Input parameters	Reply Value
link	Name of file to link to, name of link	Status
lseek	File descriptor, offset, whence	New position
mkdir	File name, mode	Status
mknod	Name of dir or special, mode, address	Status
mount	Special file, where to mount, ro flag	Status
open	Name of file to open, r/w flag	File descriptor
pipe	Pointer to 2 file descriptors (modified)	Status
read	File descriptor, buffer, how many bytes	# Bytes read
rename	File name, file name	Status

Λίστα Μηνυμάτων (3)

Message	Input parameters	Reply Value
rmdir	File name	Status
stat	File name, status buffer	Status
stime	Pointer to current time	Status
sync	(None)	Always OK
time	Pointer to place where current time goes	Status
times	Pointer to buffer for process and child times	Status
umask	Complement of mode mask	Always OK
umount	Name of special file to unmount	Status
unlink	Name of file to unlink	Status
utime	File name, file times	Always OK
write	File descriptor, buffer, how many bytes	# Bytes written

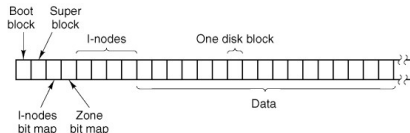
Λίστα Μηνυμάτων (4)

Message	Input parameters	Reply Value
exec	Pid	Status
exit	Pid	Status
fork	Parent pid, child pid	Status
setgid	Pid, real and effective gid	Status
setuid	Pid	Status
setuid	Pid, real and effective uid	Status
revive	Process to revive	(No reply)
unpause	Process to check	...

- ▶ Αυτά τα μηνύματα προέρχονται από τον διαχειριστή διεργασιών

Δομή Συστήματος Αρχείων

- ▶ Το Σύστημα Αρχείων του MINIX 3 αποτελείται από i-nodes, φακέλους και blocks δεδομένων
- ▶ Το μέγεθος των τμημάτων διαφοροποιείτε ανάλογα με το συνολικό μέγεθος του συστήματος και την υφιστάμενη τεχνολογία
- ▶ Όμως αυτά τα τμήματα βρίσκονται σε όλα τα συστήματα αρχείων τύπου MINIX 3



To superblock του MINIX 3

- ▶ Το superblock περιέχει πληροφορίες για τη δομή του συστήματος αρχείου
- ▶ Έχει πάντα μέγεθος 1024 bytes
- ▶ Βάση του πλήθους των inodes και το μέγεθος των blocks μπορούμε να υπολογίσουμε το μέγεθος του bitmap των inodes
- ▶ Ο αποθηκευτικός χώρος διαιρείται σε zones από $\log_2 n$ blocks

Present on disk and in memory

Present in memory but not on disk

Number of inodes
(unused)
Number of i-node bitmap blocks
Number of zone bitmap blocks
First data zone
Log. (block/zone)
Padding
Maximum file size
Number of zones
Magic number
padding
Block size (bytes)
FS sub-version
Pointer to i-node for root of mounted file system
Pointer to i-node mounted upon
i-nodes/block
Device number
Read-only flag
Native or byte-swapped flag
FS version
Direct zones/i-node
Indirect zones/indirect block
First free bit in i-node bitmap
First free bit in zone bitmap



Δομή super_block (1)

- ▶ Ορίζεται στο αρχείο `/usr/src/servers/fs/super.h`
- ▶ Κατά την εκκίνηση του συστήματος φορτώνεται στην μνήμη

```
EXTERN struct super_block {
    /* # usable inodes on the minor device */
    ino_t s_ninodes;
    /* total device size, including bit maps etc */
    zone1_t s_nzones;
    /* # of blocks used by inode bit map */
    short s_imap_blocks;
    /* # of blocks used by zone bit map */
    short s_zmap_blocks;
    /* number of first data zone */
    zone1_t s_firstdatazone;
```



Δομή super_block (2)

```
/* log2 of blocks/zone */
short s_log_zone_size;
/* try to avoid compiler-dependent padding */
short s_pad;
/* maximum file size on this device */
off_t s_max_size;
/* number of zones (replaces s_nzones in V2) */
zone_t s_zones;
/* magic number to recognize super-blocks */
short s_magic;
/* try to avoid compiler-dependent padding */
short s_pad2;
/* block size in bytes. */
unsigned short s_block_size;
```



Δομή super_block (3)

```
/* filesystem format sub-version */
char s_disk_version;

} super_block[NR_SUPERS];
```

- ▶ Ορίζεται ως πίνακας από NR_SUPERS συσκευές
- ▶ ... το μέγιστο πλήθος συσκευών που μπορούμε να χειριζόμαστε ανά πάσα στιγμή



Δομή super_block (4)

- ▶ Για να επιταχυνθεί η απόδοση του συστήματος, οι υπολογισμοί του μεγέθους του inode bitmap, το inode που περιέχει τη ρίζα των φακέλων κλπ αποθηκεύονται στη δομή super_block
- ▶ Αυτές όμως οι πληροφορίες δεν αποθηκεύονται στον αποθηκευτικό χώρο
- ▶ Υπάρχουν μόνο στη μνήμη

```
/* inode for root dir of mounted file sys */
struct inode *s_isup;
/* inode mounted on */
struct inode *s_imount;
/* precalculated from magic number */
unsigned s_inodes_per_block;
/* whose super block is this? */
dev_t s_dev;
```



Δομή super_block (5)

```
/* set to 1 iff file sys mounted read only */
int s_rd_only;
/* set to 1 iff not byte swapped file system */
int s_native;
/* file system version, zero means bad magic */
int s_version;
/* # direct zones in an inode */
int s_ndzones;
/* # indirect zones per indirect block */
int s_nindirs;
/* inodes below this bit number are in use */
bit_t s_isearch;
/* all zones below this bit number are in use*/
bit_t s_zsearch;
```



Αρχικοποίηση Συστήματος Αρχείων

- ▶ Η αρχικοποίηση ενός αποθηκευτικού χώρου γίνεται με την χρήση της mkfs

```
mkfs /dev/fd1 1440
```

- ▶ Δημιουργεί το superblock της συσκευής
- ▶ Το superblock διαβάζεται όταν καλούμε την mount (κλήση συστήματος)
 - ▶ Βασίζεται στην συνάρτηση read_super

```
_PROTOTYPE(int read_super, (struct super_block *s
```



Επικοινωνία με Οδηγό Συσκευών

- ▶ Η χρήση του αποθηκευτικού χώρου (συσκευή Εισόδου/Εξόδου) γίνεται μέσω του οδηγού της συσκευής (device driver)
- ▶ Στο αρχείο /usr/src/servers/fs/driver.c ορίζονται όλες οι συναρτήσεις για την χρήση της συσκευής
 1. dev_open -- FS opens a device
 2. dev_close -- FS closes a device
 3. dev_lo -- FS does a read or write on a device
 4. dev_status -- FS processes callback request alert
 5. gen_opcl -- generic call to a task to perform an open/close
 6. gen_lo -- generic call to a task to perform an I/O operation
 7. do_ioctl -- perform the IOCTL system call
 8. do_setsid -- perform the SETSID system call (FS side)
 9. ...
- ▶ Οι συναρτήσεις αναλαμβάνουν να στείλουν μηνύματα στον κατάλληλο οδηγό της συσκευής (system call)



- ▶ Η καταγραφή του ελεύθερου χώρου γίνεται με την χρήση 2 bitmaps
- ▶ Όταν ένα αρχείο διαγράφεται ενημερώνεται το bitmap θέτοντας το bit που αντιστοιχεί στο block σε 0
- ▶ Όταν όλα τα blocks μιας ζώνης είναι ελεύθερα, με τον ίδιο τρόπο απελευθερώνεται και η ζώνη
- ▶ Για την δημιουργία ενός αρχείου, ανατρέχουμε στο bitmap για να εντοπίσουμε κενό χώρο και να τον δεσμεύσουμε
- ▶ Σχετικά με το i-node του καινούργιου αρχείου – για να επιταχυνθεί η διαδικασία, το superblock έχει ένα δείκτη στο πρώτο ελεύθερο i-node
 - ▶ Όταν το αρχείο διαγραφεί, το superblock ενημερώνεται για να δείχνει στο i-node του αρχείου

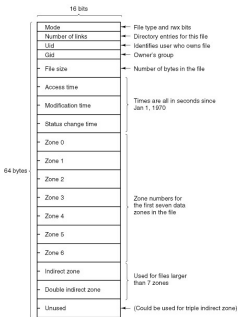


- ▶ Με βάση την πληροφορία που διατηρεί το superblock μπορούμε να υπολογίσουμε το πλήθος των blocks
- ▶ Αναλόγως το πλήθος, υπολογίζουμε πόσα blocks απαιτούνται για να αποθηκεύουμε το bitmap
 - ▶ Αν το μέγεθος των block είναι 1KB -- χωράει ένα bitmap με 8192 εγγραφές
 - ▶ ... και για ένα bitmap από 10000 εγγραφές χρειαζόμαστε 2 blocks
- ▶ Zones
 - ▶ Το σύστημα αρχείων του MINIX 3 αποθηκεύει ζώνες blocks
 - ▶ ... για την αποθήκευση των blocks σε συνεχόμενα blocks
 - ▶ Στην πράξη δεν χρησιμοποιείται



Το i-node του MINIX 3

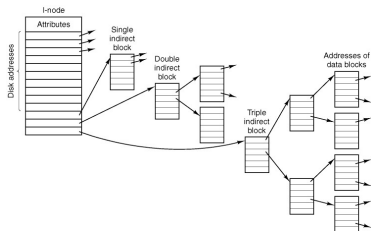
- ▶ Τα i-nodes διατηρούν τις διευθύνσεις των blocks όπου αποθηκεύονται τα δεδομένα του αρχείου
- ▶ Αποθηκεύουν μεταδεδομένα που περιγράφουν το αρχείο
- ▶ Το μέγεθος ενός i-node είναι 64 bytes
- ▶ Η δομή επιτρέπει την αποθήκευση αρχείων έως 4 GB όταν το block size είναι 4 KB



Παράδειγμα i-node

- ▶ 32-bit Zone Number (address)
- ▶ Block size = Zone size = 1KB
 - ▶ Με την χρήση των 7 διευθύνσεων zones -- 7 KB πληροφορίας
 - ▶ Με την χρήση του indirect zone μπορούμε να αποθηκεύουμε σε 1 block, 256 διευθύνσεις zones ($\frac{1024 \times 8}{32}$) -- 256 KB
 - ▶ Με την χρήση των second indirect zone μπορούμε να αποθηκεύουμε σε 1 block, 256 διευθύνσεις από indirect zones, όπου κάθε indirect zone διατηρεί 256 διευθύνσεις zones -- 64 MB
- ▶ Block size = Zone size = 4KB
 - ▶ Με την χρήση των 7 διευθύνσεων zones -- 28 KB πληροφορίας
 - ▶ Με την χρήση του indirect zone μπορούμε να αποθηκεύουμε σε 1 block, 1024 διευθύνσεις zones ($\frac{4096 \times 8}{32}$) -- 4 MB 1024*
 - ▶ Με την χρήση των second indirect zone μπορούμε να αποθηκεύουμε σε 1 block, 1024 διευθύνσεις από indirect zones, όπου κάθε indirect zone διατηρεί 1024 διευθύνσεις zones -- 4 GB





► Τα triple indirect zone δεν υλοποιούνται στο MINIX 3

```

EXTERN struct inode {
    /* file type, protection, etc. */
    mode_t i_mode;
    /* how many links to this file */
    nlink_t i_nlinks;
    /* user id of the file's owner */
    uid_t i_uid;
    /* group number */
    gid_t i_gid;
    /* current file size in bytes */
    off_t i_size;

```

```

/* time of last access (V2 only) */
time_t i_atime;
/* when was file data last changed */
time_t i_mtime;
/* when was inode itself changed (V2 only)*/
time_t i_ctime;
/* zone numbers for direct, ind, and dbl ind */
zone_t i_zone[V2_NR_TZONES];
} inode[NR_INODES];

```

- Για να επιταχυνθεί η απόδοση του συστήματος, ορισμένες πληροφορίες διατηρούνται μόνο στη μνήμη
- Δεν αποθηκεύονται στον αποθηκευτικό χώρο – υπάρχουν μόνο στη μνήμη

```

/* which device is the inode on */
dev_t i_dev;
/* inode number on its (minor) device */
ino_t i_num;
/* # times inode used; 0 means slot is free */
int i_count;
/* # direct zones (Vx_NR_DZONES) */
int i_ndzones;
/* # indirect zones per indirect block */
int i_nindirs;

```

Δομή i-node (4)

```
/* pointer to super block for inode's device */
struct super_block *i_sp;
/* CLEAN or DIRTY */
char i_dirt;
/* set to I_PIPE if pipe */
char i_pipe;
/* this bit is set if file mounted on */
char i_mount;
/* set on LSEEK, cleared on READ/WRITE */
char i_seek;
/* the ATIME, CTIME, and MTIME bits are here */
char i_update;
```



Block Cache – Προσωρινή Αποθήκευση

- ▶ Στα μοντέρνα συστήματα οι συσκευές E/E είναι οι πιο 'αργές' σε σύγκριση με τον επεξεργαστή / μνήμη
- ▶ Η προσωρινής αποθήκευση των blocks στην μνήμη στοχεύει στην επιτάχυνση της ανάγνωσης δεδομένων από τις συσκευές E/E
- ▶ Σε μια κλήση συστήματος read (για την ανάγνωση δεδομένων από μια συσκευή E/E)
 - ▶ Ελέγχουμε αν τα block είναι αποθηκευμένα στην προσωρινή μνήμη (cache)
 - ▶ Όταν ένα block δεν υπάρχει στην cache ανατρέχουμε στην συσκευή E/E και τοποθετούμε στην cache
 - ▶ Πως αδειάζει η cache ?
- ▶ Βασίζονται σε κάποιο αλγόριθμο διαχείρισης προσωρινής αποθήκευσης – στρατηγική caching

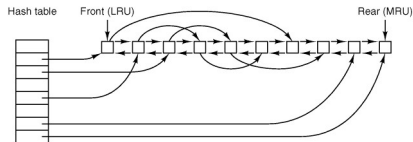


Λειτουργία Block Cache στο MINIX 3

- ▶ Οι δίσκοι μεγαλώνουν σε μέγεθος → αυξάνεται το πλήθος των blocks
- ▶ Η μνήμη μεγαλώνει σε μέγεθος → αυξάνεται το μέγεθος της προσωρινής μνήμης (cache)
- ▶ Πρέπει να μπορούμε γρήγορα να εντοπίσουμε αν ένα block βρίσκεται στην cache
 - ▶ Ο έλεγχος γίνεται σε κάθε read (κλήση συστήματος)
- ▶ Χρησιμοποιούμε μια hash function
 - ▶ Βασίζεται στα τελευταία n bits της διεύθυνσης ενός block
 - ▶ Η συνάρτηση δεν είναι 1:1 – για περισσότερα του ενός blocks επιστρέφει την ίδια τιμή (hash value)
- ▶ Όλα τα blocks με την ίδια τιμή hash αποθηκεύονται μαζί σε μια συνδεδεμένη λίστα (linked list)



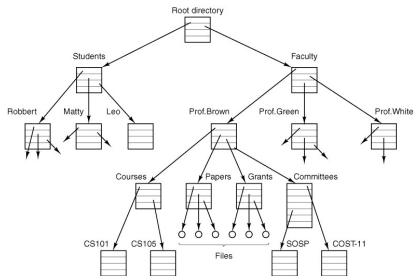
Δομή Block Cache στο MINIX 3



1. Πίνακας που περιέχει όλα τα hash values
 - ▶ Σε κάθε γραμμή αναθέτουμε μια συνδεδεμένη λίστα με τα blocks που έχουν την ίδια hash value
2. Συνδεδεμένη λίστα που περιέχει όλα τα blocks
 - ▶ Ταξινομημένη με βάση την παλαιότητα του block



Φάκελοι και Μονοπάτια



Διαχείριση Ανοικτών Αρχείων – File Descriptors

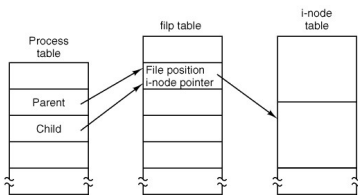
- ▶ Για να χρησιμοποιήσουμε ένα αρχείο πρέπει πρώτα να το 'ανοίξουμε'
 - ▶ Να προετοιμάσουμε το σύστημα αρχείων με την κλήση `open`
- ▶ Στο τμήμα του πίνακα των διεργασιών που διατηρεί το σύστημα αρχείων αποθηκεύουμε τα αρχεία που έχει ανοίξει μια διεργασία

```
EXTERN struct fproc {  
    ...  
    struct inode *fp_workdir; /* pointer to working direct  
    struct inode *fp_rootdir; /* pointer to current root d  
    struct filp *fp_filp[OPEN_MAX]; /* the file descriptor  
    fd_set fp_filp_inuse; /* which fd's are in use? */  
    ...  
} fproc[NR_PROCS];
```

Τρέχουσα Θέση Ανοικτού Αρχείου -- File Position

- ▶ Πρέπει να αποθηκεύουμε την 'τρέχουσα θέση' σε ένα ανοικτό αρχείο
 - ▶ Σε ποιο σημείο 'σταμάτησε' η τελευταία κλήση `read`
 - ▶ Χρησιμοποιούμε την κλήση `lseek` για να αλλάξουμε την θέση
- ▶ Στο τμήμα του πίνακα των διεργασιών που διατηρεί το σύστημα αρχείων αποθηκεύουμε τα αρχεία που έχει ανοίξει μια διεργασία
- ▶ Πού αποθηκεύουμε την τρέχουσα θέση;
 - ▶ Στον πίνακα των διεργασιών υπάρχει πρόβλημα όταν καλέσουμε τη `fork`
 - ▶ Θα αντιγραφούν τα ανοικτά αρχεία – σωστό
 - ▶ Θα αντιγραφούν οι θέσεις – λάθος
 - ▶ Στο `i-node` του αρχείου
 - ▶ ... χειρότερα

Πίνακας Ανοικτών Αρχείων Θέσεων



- ▶ Σε ξεχωριστό 'διαμοιραζόμενο' πίνακα `filp`
- ▶ Ο πίνακας είναι κοινός για όλες τις διεργασίες

Κλήση Συστήματος read

- ▶ Ο κώδικας του συστήματος αρχείου επικεντρώνεται στην εκτέλεση των κλήσεων του συστήματος
- ▶ Ας εξετάσουμε την περίπτωση όπου ένα πρόγραμμα εκτελεί:
`n = read(fd, buffer, nbytes);`
- ▶ Μόλις παραλάβει το σύστημα αρχείων το μήνυμα
 - ▶ Ανατρέχει στον πίνακα filp σύμφωνα με την παράμετρο fd
 - ▶ Ελέγχει την τρέχουσα θέση στο αρχείο
 - ▶ Υπολογίζει το πλήθος των blocks που πρέπει να διαβαστούν – και τις διευθύνσεις τους
 - ▶ Για κάθε block ανατρέχει στην προσωρινή μνήμη ή μονάδα E/E
 - ▶ Συγχωνεύει τα blocks στον buffer
 - ▶ Αντιγράφει τον buffer στην μνήμη της διεργασίας
- ▶ Αφού ολοκληρωθεί η εκτέλεση της κλήσης, το σύστημα αρχείων διαβάζει ορισμένα block ακολουθώντας τεχνικές read-ahead



Βοηθητική συνάρτηση read (2)

```
PUBLIC ssize_t read(fd, buffer, nbytes)
int fd;
void *buffer;
size_t nbytes;
{
    message m;

    m.m1_i1 = fd;
    m.m1_i2 = nbytes;
    m.m1_p1 = (char *) buffer;
    return(_syscall(FS, READ, &m));
}
```



Διαχείριση Block

- ▶ Η δομή της προσωρινής μνήμης block ορίζεται στο αρχείο `/usr/src/servers/fs/buf.h`
- ▶ Ο πίνακας `buf` αποθηκεύει τα δεδομένα των block μαζί με όλες τις μετα-πληροφορίες
 - ▶ Μία εγγραφή χωρίζεται σε 2 τμήματα: data και header
 - ▶ Το τμήμα data αποτελείται από τα περιεχόμενα του block
 - ▶ Το τμήμα header περιέχει pointers, counters, flags που χαρακτηρίζουν την κατάσταση του block
- ▶ Το τμήμα δεδομένων ορίζεται ως union 7 διαφορετικών τύπων
 - ▶ Μερικές φορές βολεύει να προσπελάσουμε τα δεδομένα με την χρήση ειδικού τύπου πίνακα
 - ▶ π.χ., πίνακας χαρακτήρων, πίνακας bit ...
- ▶ Ο πίνακας με τις πημές hash ονομάζεται `buf_hash`
- ▶ Οι συνδεδεμένες λίστες LRU, MRU τηρούνται από τους δείκτες `front rear`



Εσωτερικές Δομές για Διαχείριση block (1)

```
EXTERN struct buf {
    /* Data portion of the buffer. */
    union { ... } b;

    /* Header portion of the buffer. */
    struct buf *b_next; /* link all free bufs in a chain
    struct buf *b_prev; /* link all free bufs the other way
    struct buf *b_hash; /* link bufs on hash chains *
    block_t b_blocknr; /* block number of its (minor
    dev_t b_dev; /* major | minor device where block
    char b_dirt; /* CLEAN or DIRTY */
    char b_count; /* number of users of this buffer *
} buf[NR_BUFS];
```



Εσωτερικές Δομές για Διαχείριση block (2)

```
union {
    char b__data[_MAX_BLOCK_SIZE]; /* ordinary user
    /* directory block */
    struct direct b__dir[NR_DIR_ENTRIES(_MAX_BLOCK_SIZE)];
    /* V1 indirect block */
    zone1_t b__v1_ind[V1_INDIRECTS];
    /* V2 indirect block */
    zone_t b__v2_ind[V2_INDIRECTS(_MAX_BLOCK_SIZE)];
    /* V1 inode block */
    d1_inode b__v1_ino[V1_INODES_PER_BLOCK];
    /* V2 inode block */
    d2_inode b__v2_ino[V2_INODES_PER_BLOCK(_MAX_BLOCK_SIZE)];
    /* bit map block */
    b1_chunk_t b__bitmap[FS_BITMAP_CHUNKS(_MAX_BLOCK_SIZE)];
} b;
```

Εσωτερικές Δομές για Διαχείριση block (3)

```
/* These defs make it possible to use to
   bp->b_data instead of bp->b.b_data */
#define b_data b.b_data
#define b_dir b.b_dir
#define b_v1_ind b.b_v1_ind
#define b_v2_ind b.b_v2_ind
#define b_v1_ino b.b_v1_ino
#define b_v2_ino b.b_v2_ino
#define b_bitmap b.b_bitmap
```

Εσωτερικές Δομές για Διαχείριση block (4)

```
/* the buffer hash table */
EXTERN struct buf *buf_hash[NR_BUF_HASH];

/* points to least recently used free block */
EXTERN struct buf *front;

/* points to most recently used free block */
EXTERN struct buf *rear;

/* # bufs currently in use (not on free list)*/
EXTERN int bufs_in_use;
```

Λειτουργίες Διαχείρισης Block (1)

- ▶ Οι συναρτήσεις που διαχειρίζονται τις δομές της προσωρινής μνήμης block ορίζονται στο αρχείο `/usr/src/servers/fs/cache.h`
- ▶ 9 βασικές συναρτήσεις
- ▶ `get_block`: οποτεδήποτε κάποια συνάρτηση του συστήματος αρχείου χρειάζεται να διαβάσει ένα block
 - ▶ πρώτα ελέγχει την προσωρινή μνήμη: εξετάζει τον πίνακα με τις τιμές hash
 - ▶ αν δεν υπάρχει το block το μεταφέρει από την συσκευή Ε/Ε
 - ▶ πρέπει να επιλέξει ποιο block θα αφαιρέσει από τη cache (για να χωρέσει αυτό που μόλις μεταφέρθηκε)
- ▶ `put_block`: τοποθετεί ένα block στην προσωρινή μνήμη και σε ορισμένες περιπτώσεις την εγγραφή στην συσκευή Ε/Ε
 - ▶ ενημερώνει την λίστα LRU

Λειτουργίες Διαχείρισης Block (2)

- ▶ `alloc_zone`: Δέσμευση νέας ζώνης (για να μεγαλώσει ένα αρχείο)
 - ▶ αν ζητηθεί μόνο 1 block ελέγχει το πεδίο `s_zsearch` του `superblock`
 - ▶ αλλιώς ελέγχει το `zone bitmap` για την πρώτη ελεύθερη θέση
 - ▶ προσπαθεί να βρει μια ζώνη 'κοντά' στην τελευταία ζώνη που χρησιμοποιεί το αρχείο
 - ▶ αυτό γίνεται ξεκινώντας την αναζήτηση ανατρέχοντας στον πίνακα στην θέση της τελευταίας ζώνης
- ▶ `free_zone`: Αποδέσμευση ζώνης (κατά την καταγραφή αρχείου)
- ▶ `rw_block`: Μεταφορά block από/προς αποθηκευτική μονάδα προς/από cache
 - ▶ βασικότερη λειτουργία



Λειτουργίες Διαχείρισης Block (3)

- ▶ `invalidate`: Απομάκρυνση αποθηκευμένων block από προσωρινή μνήμη
 - ▶ χρησιμοποιείται όταν πρόκειται να κάνουμε unmount την συσκευή E/E
 - ▶ 'αδειάζει' την προσωρινή μνήμη
- ▶ `flushall`: Αποθήκευση όλων των block που έχουν αλλαγές
 - ▶ χρησιμοποιείται όταν την `get_block` όταν κάποιο block πρέπει να αφαιρεθεί από την προσωρινή μνήμη για να προκύψει χώρος για το νέο block
 - ▶ επίσης καλείται από την κλήση συστήματος SYNC – καλείται περιοδικά
- ▶ `rw_scattered`: Μεταφορά πολλαπλών block από/προς αποθηκευτική μονάδα προς/από cache
- ▶ `rm_lru`: Διαγραφή του πρώτου block από την λίστα LRU



Λειτουργίες Διαχείρισης i-Node (1)

- ▶ Η δομή του i-node ορίζεται στο αρχείο `/usr/src/servers/fs/inode.h`
- ▶ Οι συναρτήσεις που διαχειρίζονται τα inodes ορίζονται στο αρχείο `/usr/src/servers/fs/inode.c`
- ▶ 10 βασικές συναρτήσεις
- ▶ Ορισμένες ακολουθούν την ίδια λογική με την διαχείριση block
- ▶ `get_inode`: οποτεδήποτε κάποια συνάρτηση του συστήματος αρχείου χρειάζεται να διαβάσει ένα i-node
 - ▶ πρώτα ελέγχει τον πίνακα των inodes
 - ▶ αν βρεθεί αυξάνει τον μετρητή και επιστρέφει έναν δείκτη στο i-node
 - ▶ αν δεν υπάρχει το i-node το μεταφέρει από την συσκευή E/E χρησιμοποιώντας τη μέθοδο `rw_inode`



Λειτουργίες Διαχείρισης i-Node (2)

- ▶ `put_inode`: όταν ολοκληρωθεί η συνάρτηση που χρησιμοποίησε την `get_inode` το inode 'επιστρέφεται' με αυτή την συνάρτηση
 - ▶ μειώνει τον μετρητή `i_count`
 - ▶ αν ο μετρητής πάρει την τιμή 0 τότε το αρχείο που περιγράφει το i-node δεν χρησιμοποιείται
 - ▶ αν χρειαστεί μπορούμε να αφαιρέσουμε το i-node από τον πίνακα
 - ▶ αν το i-node έχει αλλαγές (λέμε ότι είναι "dirty") μεταφέρεται στην μονάδα E/E
 - ▶ αν το πεδίο `i_link` είναι 0 τότε το αρχείο δεν περιέχεται σε κάποιο φάκελο – άρα μπορούμε να απελευθερώσουμε όλες τις ζώνες του αρχείου
 - ▶ Η μείωση του μετρητή `i_count` έχει διαφορετικές επιπτώσεις από την μείωση του μετρητή `i_link`



Λειτουργίες Διαχείρισης i-Node (3)

- ▶ `alloc_inode`: Δέσμευση ενός i-node όταν δημιουργείται ένα αρχείο
 - ▶ ελέγχει το `superblock` κατά πόσο η συσκευή επιτρέπει εγγραφή
 - ▶ ελέγχει το πεδίο `s_isearch` του `superblock`
 - ▶ σε αντίθεση με την `alloc_block` δεν προσπαθεί να βρει ένα κενό i-node κοντά σε κάποιο άλλο
- ▶ `free_inode`: Αποδέσμευση i-node (κατά την καταγραφή αρχείου)
 1. ενημερώνει το `superblock` (πεδίο `s_isearch`)
 2. ενημερώνει την αντίστοιχη θέση του `bitmap` θέτοντας την σε 0
- ▶ `update_times`: Ενημερώνει τα πεδία ημερ/νιας - ώρας του i-node
 - ▶ επικοινωνεί με τον πυρήνα – εκεί υλοποιούνται οι υπηρεσίες ρολογιού
 - ▶ χρησιμοποιείται από τις κλήσεις συστήματος `STAT` και `FSTAT`



Λειτουργίες Διαχείρισης i-Node (4)

- ▶ `rw_inode`: Μεταφορά i-node από/προς αποθηκευτική μονάδα προς/από cache
 - ▶ αντιστοιχη με την `rw_inode`
 - ▶ υπολογίζει σε ποιο block είναι αποθηκευμένο το i-node
 - ▶ διαβάζει το block από την μονάδα E/E
 - ▶ εντοπίζει το i-node μέσα στο block
 - ▶ διαβάζει το i-node και το τοποθετεί στον πίνακα των inodes
 - ▶ καλεί την `put_block`
- ▶ `old_icopy`: Χρησιμοποιείται για την μετατροπή των i-nodes της έκδοσης V1 του συστήματος
- ▶ `new_icopy`: Χρησιμοποιείται για την μετατροπή των i-nodes της έκδοσης V2 του συστήματος
- ▶ `dup_icopy`: Αυξάνει τον μετρητή του i-node κάθε φορά που καλείται η `OPEN`



Λειτουργίες Διαχείρισης Superblock (1)

- ▶ Η δομή του `superblock` ορίζεται στο αρχείο `/usr/src/servers/fs/super.h`
- ▶ Οι συναρτήσεις που διαχειρίζονται το `superblock` ορίζονται στο αρχείο `/usr/src/servers/fs/super.c`
- ▶ 6 βασικές συναρτήσεις
- ▶ `alloc_bit`: χρησιμοποιείται όταν καλούμε την `alloc_inode` ή την `alloc_zone`
 - ▶ διαβάζει τον αντίστοιχο `bitmap` για να εντοπίσει μια κενή θέση
 - ▶ πρόκειται για ένα loop 3ων επιπέδων
 - ▶ Το πρώτο επίπεδο ανατρέχει στα blocks όπου έχει αποθηκευτεί το `bitmap`
 - ▶ Το δεύτερο επίπεδο ανατρέχει στα words του κάθε block
 - ▶ Το τρίτο επίπεδο ανατρέχει τα bit του κάθε word



Λειτουργίες Διαχείρισης Superblock (2)

- ▶ `free_bit`: χρησιμοποιείται όταν καλούμε την `free_inode` ή την `free_zone`
 - ▶ ποιο απλή από την `alloc_bit`
 - ▶ Εντοπίζει το block που περιέχει τη θέση του πίνακα
 - ▶ Ανατρέχει στο word και θέτει το αντίστοιχο bit σε 0
 - ▶ Για να διαβάσει το block καλεί την `get_block`
 - ▶ Για να αποθηκεύσει την αλλαγή καλεί την `put_block`
- ▶ `get_super`: ελέγχει τον πίνακα με τα `superblock` και επιστρέφει αυτό που αντιστοιχεί στην συγκεκριμένη συσκευή E/E
 - ▶ Την χρησιμοποιούμε για να ελέγξουμε κατά πόσο η συσκευή έχει ήδη γίνει mount
- ▶ `get_block_size`: εντοπίζει το μέγεθος των block στην συγκεκριμένη συσκευή E/E
 - ▶ Εντοπίζει το `superblock` της συσκευής και διαβάζει το συγκεκριμένο πεδίο



Λειτουργίες Διαχείρισης Superblock (3)

- ▶ `mounted`: αναφέρει κατά πόσο ένα συγκεκριμένο `inode` βρίσκεται σε μια `mounted` συσκευή `E/E`
 - ▶ χρησιμοποιείται μόνο όταν μια συσκευή είναι κλειστή
 - ▶ Επιστρέφει `TRUE` αν η συσκευή είναι η ρίζα του συστήματος αρχείων
 - ▶ ή αν η συσκευή είναι `mounted`
- ▶ `read_super`: διαβάζει το `superblock` από μια συσκευή `E/E`
 - ▶ ανάλογη με την `read_block` και `read_inode`
 - ▶ δεν διαβάζει από την προσωρινή μνήμη – διαβάζει απ' ευθείας από την συσκευή `E/E`
 - ▶ ελέγχει την έκδοση του συστήματος αρχείων της συσκευής και κάνει τις απαραίτητες μετατροπές στην δομή `superblock` αμέσως μετά την ανάγνωση
 - ▶ ανεξάρτητα από την έκδοση του συστήματος αρχείων που έχει κάθε συσκευή `E/E`, τα `superblock` που διατηρούνται στην μνήμη έχουν την ίδια δομή



Λειτουργίες Διαχείρισης Πίνακα Ανοικτών Αρχείων (1)

- ▶ Η δομή του πίνακα των ανοικτών αρχείων ορίζεται στο αρχείο `/usr/src/servers/fs/file.h`

```
EXTERN struct filp {
    mode_t filp_mode; /* RW bits - how file is opened
    int filp_flags; /* flags from open and fcntl */
    int filp_count; /* file descriptors share this sl
    struct inode *filp_ino; /* pointer to the inode *
    off_t filp_pos; /* file position */

    /* the following fields are for select() */
    int filp_selectors; /* select()ing processes bloc
    int filp_select_ops; /* interested in these SEL_*
    int filp_pipe_select_ops; /* fd-type-specific se
} filp[NR_FILPS];
```



Λειτουργίες Διαχείρισης Πίνακα Ανοικτών Αρχείων (2)

- ▶ Οι συναρτήσεις που διαχειρίζονται τον πίνακα των ανοικτών αρχείων ορίζονται στο αρχείο `/usr/src/servers/fs/filedes.c`
- ▶ 4 βασικές συναρτήσεις
- ▶ `get_fd`: επιστρέφει ένα νέο `file descriptor` ενός συγκεκριμένου αρχείου
 - ▶ ανατρέπει τον πίνακα και εντοπίζει μια κενή εγγραφή
 - ▶ χρησιμοποιείται από τις κλήσεις συστήματος `OPEN` και `CREAT`
- ▶ `get_filp`: εντοπίζει ένα `file descriptor`
 - ▶ ανατρέπει τον πίνακα και εντοπίζει το `file descriptor`
- ▶ `find_filp`: εντοπίζει όλες τις διεργασίες που έχουν ένα `file descriptor` για ένα συγκεκριμένο αρχείο
- ▶ `inval_filp`: κλείνει ένα `file descriptor`



Λειτουργίες Διαχείρισης File Locks (1)

- ▶ Το `MINIX 3` επιτρέπει να κάνουμε `lock` ένα μέρος ενός αρχείου για ανάγνωση, εγγραφή ή και τα δύο
- ▶ Το κλειδωμα ενός (μέρους) αρχείου γίνεται με την κλήση συστήματος `FCNTL`
- ▶ Η δομή του πίνακα των `file locks` ορίζεται στο αρχείο `/usr/src/servers/fs/lock.h`

```
EXTERN struct file_lock {
    short lock_type; /* F_RDLCK or F_WRLCK; 0 means
    pid_t lock_pid; /* pid of the process holding the
    struct inode *lock_inode; /* pointer to the inode
    off_t lock_first; /* offset of first byte locked
    off_t lock_last; /* offset of last byte locked */
} file_lock[NR_LOCKS];
```



Λειτουργίες Διαχείρισης File Locks (2)

- ▶ Η κλήση συστήματος FCNTL προσφέρει 3 λειτουργίες σχετικά με το κλειδωμα μέρους ενός αρχείου
 1. F_SETLK – Κλειδώνει μια περιοχή ενός αρχείου για ανάγνωση και εγγραφή
 2. F_SETLKW – Κλειδώνει μια περιοχή ενός αρχείου για εγγραφή
 3. F_GETLK – Ελέγχει αν μια περιοχή ενός αρχείου είναι κλειδωμένη
- ▶ Οι συναρτήσεις που διαχειρίζονται τον πίνακα των file locks ορίζονται στο αρχείο `/usr/src/servers/fs/lock.c`
- ▶ 4 βασικές συναρτήσεις
- ▶ `lock_op`: καλείται όταν θέλουμε να κλειδώσουμε ένα (μέρος) αρχείου με την κλήση συστήματος FCNTL
 - ▶ ελέγχει αν το μέρος του αρχείου που θέλουμε να κλειδώσουμε υπάρχει
 - ▶ ελέγχει αν το νέο κλειδωμα καλύπτει περιοχές που είναι ήδη κλειδωμένες



Λειτουργίες Διαχείρισης File Locks (3)

- ▶ `lock_revive`: καλείται όταν θέλουμε να ξεκλειδώσουμε ένα μέρος ενός αρχείου
 - ▶ ενεργοποιεί όλες τις διεργασίες που περιμένουν να διαβάσουν το αρχείο (ή το μέρος του αρχείου)
 - ▶ ο εντοπισμός των διεργασιών δεν είναι εύκολος
 - ▶ για να αποφύγουμε σύνθετα τμήματα κώδικα, υποθέτουμε ότι το κλειδωμα των αρχείων δεν είναι σύνθετος
 - ▶ στην ουσία οι διεργασίες που έχουν μπλοκαριστεί, περιοδικά ξανά-τρέχουν. Αν το αρχείο είναι ακόμα κλειδωμένο, θα ξανά μπλοκαρουν
 - ▶ δεν είναι αποδοτικός τρόπος για να υλοποιήσουμε βάσεις δεδομένων (για πολλούς χρήστες)
 - ▶ η μέθοδος καλείται και όταν θέλουμε να κλείσουμε ένα αρχείο – για να διαγραφούν τα κλειδώματα



Αρχικοποίηση Συστήματος Αρχείων

- ▶ Η κεντρική συνάρτηση του συστήματος αρχείων ορίζεται στο αρχείο `/usr/src/servers/fs/main.c`
- ▶ Η αρχικοποίηση του συστήματος γίνεται από την συνάρτηση `fs_init`
- ▶ Το πρώτο βήμα της αρχικοποίησης είναι η δημιουργία του πίνακα των διεργασιών
 - ▶ Το σύστημα αρχείων αρχικοποιεί τον πίνακα των διεργασιών σε συνεργασία με τον Διαχειριστή Διεργασιών
 - ▶ Ο Διαχειριστής Διεργασιών στέλνει τις εγγραφές του πίνακα υπο την μορφή μηνυμάτων
 - ▶ Για κάθε διεργασία στέλνει ένα μήνυμα με το PID και τα UID, GID
 - ▶ Αμέσως μετά το μήνυμα που περιγράφει την τελευταία διεργασία, ο Διαχειριστής Διεργασιών στέλνει ένα τελευταίο μήνυμα για να ενημερώσει ότι δεν υπάρχουν άλλες διεργασίες
 - ▶ Το Σύστημα Αρχείων απαντάει στέλνοντας ένα μήνυμα OK



Αρχικοποίηση Πίνακα Διεργασιών

```
do {
    if (OK != (s=receive(PM_PROC_NR, &mess)))
        panic(__FILE__, "FS couldn't receive from PM", s);
    if (NONE == mess.PR_ENDPT) break;
    rfp = &fproc[mess.PR_SLOT];
    rfp->fp_pid = mess.PR_PID;
    rfp->fp_endpoint = mess.PR_ENDPT;
    rfp->fp_realmid = (uid_t) SYS_UID;
    rfp->fp_effuid = (uid_t) SYS_UID;
    rfp->fp_realgid = (gid_t) SYS_GID;
    rfp->fp_effgid = (gid_t) SYS_GID;
    rfp->fp_umask = ~0;
} while (TRUE);
mess.m_type = OK;
s=send(PM_PROC_NR, &mess);
```



Εσωτερικοί Έλεγχοι

- ▶ Αμέσως μετά την αρχικοποίηση του πίνακα διεργασιών γίνονται ορισμένοι εσωτερικοί έλεγχοι

```
if (OPEN_MAX > 127)
    panic(__FILE__, "OPEN_MAX > 127", NO_NUM);
if (NR_BUFS < 6)
    panic(__FILE__, "NR_BUFS < 6", NO_NUM);
if (V1_INODE_SIZE != 32)
    panic(__FILE__, "V1 inode size != 32", NO_NUM);
if (V2_INODE_SIZE != 64)
    panic(__FILE__, "V2 inode size != 64", NO_NUM);
if (OPEN_MAX > 8 * sizeof(long))
    panic(__FILE__, "Too few bits in fp_cloexec", NO_N
```



Αρχικοποίηση Προσωρινής Μνήμης (1)

- ▶ Μετά απο τους εσωτερικούς ελέγχους γίνεται η αρχικοποίηση της προσωρινής μνήμης
- ▶ Χρησιμοποιείται η συνάρτηση `buf_pool()`
 - ▶ Ορίζεται στο αρχείο `main.c`
- ▶ Αρχικοποιεί την συνδεδεμένη λίστα των block
- ▶ Αρχικοποιεί την πίνακα με τις τιμές hash
- ▶ Αρχικοποιεί τις δύο ουρές (`front`, `read`)

```
bufs_in_use = 0;
front = &buf[0];
rear = &buf[NR_BUFS - 1];
```



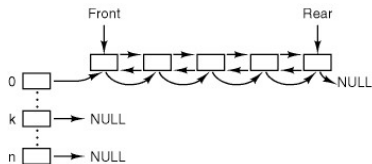
Αρχικοποίηση Προσωρινής Μνήμης (2)

```
for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
    bp->b_blocknr = NO_BLOCK;
    bp->b_dev = NO_DEV;
    bp->b_next = bp + 1;
    bp->b_prev = bp - 1;
}
buf[0].b_prev = NIL_BUF;
buf[NR_BUFS - 1].b_next = NIL_BUF;

for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
    bp->b_hash = bp->b_next;
buf_hash[0] = front;
```



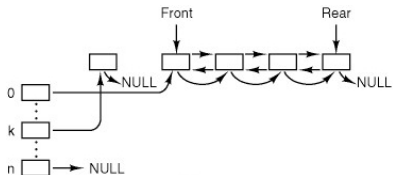
Παράδειγμα Προσωρινής Μνήμης



- ▶ Όλα τα block βρίσκονται στις δύο ουρές LRU, MRU
- ▶ Όλα τα block είναι τοποθετημένα στην πρώτη ουρά hash
- ▶ Όλα τα block σχηματίζουν την συνδεδεμένη λίστα (δύο κατευθύνσεων)



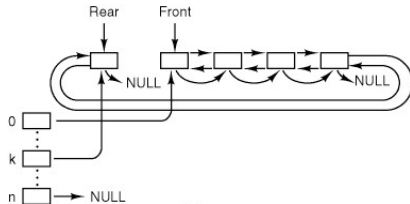
Χρήση ενός block από την Προσωρινή Μνήμη



- ▶ Όταν ζητηθεί ένα block για χρήση με την `get_block`
- ▶ Το πρώτο διαθέσιμο block της LRU αφαιρείται
- ▶ Αντιγράφουμε σε αυτό την τιμή από την μονάδα E/E
- ▶ Τοποθετείτε στην σωστή ουρά hash



Επιστροφή ενός block στην Προσωρινή Μνήμη



- ▶ Όταν επισταθεί το block αμέσως μετά την χρήση του με την `put_block`
- ▶ Παραμένει στις ουρές για μελλοντική χρήση



Ολοκλήρωση Αρχικοποίησης Συστήματος Αρχείων

- ▶ Μετά από την αρχικοποίηση της προσωρινής μνήμης
- ▶ Γίνεται σύνδεση/αναγνώριση των συσκευών E/E με την χρήση της συνάρτησης `build_dmap()`
- ▶ Γίνεται αρχικοποίηση του ramdisk με την χρήση της συνάρτησης `init_root()`
 - ▶ Χρησιμοποιείται από το υπόλοιπο ΛΣ σε διάφορες λειτουργίες
- ▶ Γίνεται ανάγνωση του superblock της κεντρικής συσκευής E/E με την χρήση της συνάρτησης `init_root()`



Δημιουργία, Άνοιγμα Κλείσιμο Αρχείων

- ▶ Οι λειτουργίες που υλοποιούν το άνοιγμα των αρχείων για χρήση βρίσκονται στο αρχείο `open.c`
- ▶ Πρόκειται για 6 κλήσεις συστήματος
 1. CREAT
 2. OPEN
 3. MKNOD
 4. MKDIR
 5. CLOSE
 6. LSEEK
- ▶ Σε παλαιότερες εκδόσεις συστημάτων UNIX οι κλήσεις συστήματος CREAT και OPEN ήταν διαφορετικές
- ▶ Πλέον δεν υπάρχει λόγος:
 - ▶ Αν το αρχείο που πρέπει να ανοίξει δεν υπάρχει, δημιουργείται καινούργιο

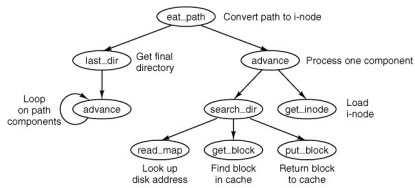


Μετατροπή Path σε i-node

- ▶ Ορισμένες κλήσεις συστήματος εκτελούνται βάση ενός path που υποδηλώνει την θέση του αρχείου στο σύστημα
- ▶ Οι κλήσεις του συστήματος πρέπει να εντοπίσουν το i-node που αντιστοιχεί στο αρχείο
- ▶ Η υλοποίηση της μετατροπής του path στο αντίστοιχο i-node γίνεται στο αρχείο `path.c`
- ▶ Η βασική συνάρτηση είναι η `eat_path()`
 - ▶ Ξεκινάει από τον τελικό φάκελο που ορίζει το path χρησιμοποιώντας την συνάρτηση `last_dir`
 - ▶ Εξετάζει το path αναδρομικά με την χρήση της συνάρτησης `advance`
 - ▶ Αν δεν εντοπίσει το i-node τότε επιστρέφει το `NIL_INODE`
 - ▶ Αν εντοπιστεί το i-node το τοποθετεί στον πίνακα των i-node



Μετατροπή Path σε i-node

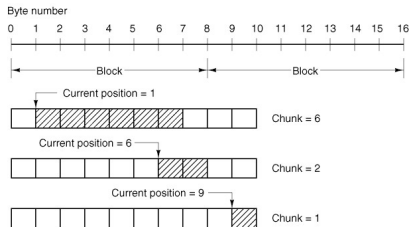


Ανάγνωση Αρχείων (1)

- ▶ Εφόσον έχουμε ανοίξει ένα αρχείο μπορούμε να διαβάσουμε τα δεδομένα
- ▶ Η υλοποίηση της ανάγνωσης αρχείων γίνεται στο αρχείο `read.c`
- ▶ Γίνονται βασικοί έλεγχοι για τις παραμέτρους που έχει κληθεί η λειτουργία του συστήματος
- ▶ Αρχικοποιούνται οι μεταβλητές όπου θα αποθηκευτούν τα δεδομένα που διαβάσκων
- ▶ Στην συναίχεια ξεκινάει ένας βρόγχος για να διαβαστούν όλα τα block που περιέχουν τα δεδομένα που ζητήθηκαν
- ▶ Τα δεδομένα διαβάζονται σε chunks
 - ▶ Κάθε chunk χωράει σε ένα disk block
- ▶ Ο βρόγχος συναίζεται έως ότου
 1. Αναγνωστούν όλα τα δεδομένα που ζητήθηκαν
 2. Έχει διαβαστεί όλο το block
 3. Τελειώσει το αρχείο
- ▶ Η βασική λειτουργία της ανάγνωσης ενός chunk γίνεται με την χρήση της `rw_chunk`



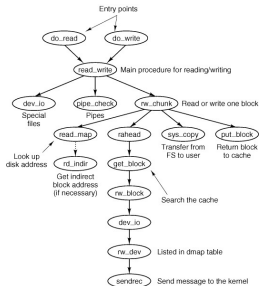
Ανάγνωση Αρχείων (2)



- ▶ Παράδειγμα για την ανάγνωση chunk από ένα αρχείο 10 byte όταν ζητάμε να διαβάσουμε 6 bytes



Ανάγνωση Αρχείων (3)



Εγγραφή Αρχείων

(a)

24

 Free zones: 12 20 31 36...

(b)

24	25
----	----

(c)

24	25	40
----	----	----

(d)

24	25	40	41
----	----	----	----

(e)

24	25	40	41	62
----	----	----	----	----

(f)

24	25	40	41	62	63
----	----	----	----	----	----

Block number

- ▶ Ανάθεση block σε ένα αρχείο, όταν το ΣΑ έχει 1 Kbyte blocks και 2 Kbyte zones