Εργαστήριο Λειτουργικών Συστημάτων

Μάθημα $6^{ου}$ Εξαμήνου,

Τομέας Λογικού και Υπολογιστών

Ιωάννης Χατζηγιαννάκης

Σημειώσεις Μαθήματος

Ενότητα 6

Σημειώσεις Μαθήματος – $6^{η}$ Ενότητα

Μονάδες Εισόδου / Εξόδου στο MINIX 3

Συσκευές Εισόδου / Εξόδου

Οδηγοί Συσκευών Εισόδου / Εξόδου Ανεξάρτητοι Υλικού

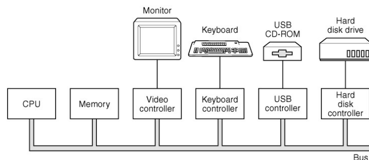
Οδηγοί Συσκευών Εισόδου / Εξόδου τύπου Block

Οδηγοί Συσκευών RAM Disk

Γενικά Θέματα συσκευών εισόδου / εξόδου

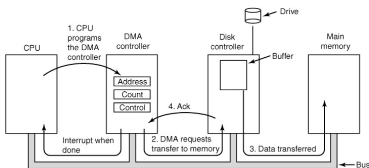
- Κάθε άνθρωπος αντιλαμβάνεται τις συσκευές εισόδου / εξόδου με διαφορετικό τρόπο
  - Ο ηλεκτρολόγος μηχ. τις βλέπει σε επίπεδο υλικού
  - Ο προγραμματιστής τις βλέπει σε επίπεδο διεπαφής με το λογισμικό
  - Ο χρήστης τις βλέπει σε επίπεδο εφαρμογής
  - ...
- Υπάρχουν διαφορετικών ειδών συσκευές – χωρίζονται (χοντρικά) σε δύο βασικές κατηγορίες
  - Μπλοκ συσκευές (block devices)
  - Συσκευές χαρακτήρων (character devices)
- Ο χειρισμός μιας συσκευής σε επίπεδο υλικού γίνεται με την χρήση ενός ελεγκτή (controller) ή προσαρμογέα (adaptor)

Συνδεσμολογία συσκευών εισόδου / εξόδου



- Η επικοινωνία των συσκευών με τους ελεγκτές γίνεται σε χαμηλό επίπεδο
- Οι ελεγκτές επικοινωνούν με τον επεξεργαστή με την χρήση interrupt (και ορισμένους registers)
- Η ανταλλαγή πληροφορίας γίνεται με την χρήση της μνήμης

Direct Memory Access (DMA)

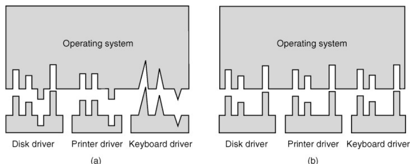


- Είναι απαραίτητο ο επεξεργαστής να επικοινωνεί με τον ελεκτή για την μεταφορά δεδομένων
- Η μεταφορά της πληροφορίας γίνεται με 1 byte κάθε φορά – αυτό καθυστερεί υπερβολικά το σύστημα
- Χρησιμοποιούμε τον ελεγκτή DMA για να επιταχύνουμε διαδικασία

Αρχές Λογισμικού Εισόδου / Εξόδου

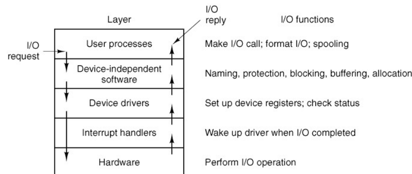
- Βασική αρχή για τον σχεδιασμό είναι η ανεξαρτησία του λογισμικού από την τεχνολογία του υλικού της συσκευής
- Θέλουμε να αναπτύξουμε εφαρμογές χωρίς να πρέπει να ορίσουμε εκ των προτέρων την συσκευή που θα συνδέσουμε στο τελικό σύστημα
- Χωρίς να περιοριζόμαστε σε συγκεκριμένο κατασκευαστή
- Ο χειρισμός των λαθών να είναι ανεξάρτητος του τύπου συσκευής
- Να μην γράψουμε επιπλέον κώδικα για να χειριστούμε συσκευές περιορισμένων δυνατοτήτων (π.χ. μικρή/αργή μνήμη)
- Ιεραρχούμε το σύστημα που σχετίζεται με τις συσκευές εισόδου / εξόδου σε 5 επίπεδα
  1. Υλικό συσκευής
  2. Χειριστές interrupt
  3. Οδηγοί Συσκευών (device drivers)
  4. Διαχειριστές Συσκευών (ανεξάρτητοι από τους οδηγούς)
  5. Λογισμικό Εισόδου / Εξόδου (user space)

Διαχειριστές Συσκευών (ανεξάρτητοι από τους οδηγούς)



- Ενιαία διεπαφή χειρισμού/επικοινωνίας συσκευών
- Ενιαίος τρόπος χειρισμός λαθών
- Ενιαίος τρόπος απαρίθμησης / ονομασίας συσκευών

Ροή Πρόσβασης στις Συσκευές Εισόδου / Εξόδου



- Η ροή εκτέλεσης μιας αίτησης εισόδου / εξόδου
- π.χ. για την εκτύπωση ενός μηνύματος στην οθόνη

Οδηγοί Συσκευών Εισόδου / Εξόδου στο Minix (1)

- Για κάθε τύπο συσκευής ΕΕ υπάρχει ένας ανεξάρτητος οδηγός (device diver)
- Οι οδηγοί συσκευών είναι κανονικές διεργασίες
- Οι οδηγοί επικοινωνούν με το σύστημα αρχείων με την χρήση του μηχανισμού ανταλλαγής μηνυμάτων
- Ο κώδικας των οδηγών βρίσκεται στον φάκελο */usr/src/drivers*
- Ένας απλός οδηγός υλοποιείται σε ένα μόνο αρχείο, π.χ. ο οδηγός του εκτυπωτή υλοποιείται στο αρχείο */usr/src/drivers/printer/printer.c*
- Για τους οδηγούς RAM disk, hard disk και floppy disk υπάρχει ένα αρχείο για κάθε τύπο συσκευής και ένα κεντρικό αρχείο που περιέχει τις βασικές/κοινές συναρτήσεις (φάκελος (*/usr/src/drivers/libdriver*, αρχεία *driver.c* και *drvlib.c*)
  - Πρόκειται για μια διεπαφή ανεξάρτητη από το τύπο της συσκευής

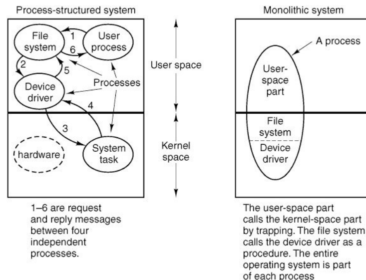
Οδηγοί Συσκευών Εισόδου / Εξόδου στο Minix (2)

- Ο κώδικας των οδηγών για το τερματικό οργανώνεται με ίδιο τρόπο
- Το αρχείο */usr/src/drivers/tty/tty.c* υλοποιεί έναν οδηγό ανεξάρτητο από τον τύπο της συσκευής
- Για κάθε τύπο συσκευής υπάρχει ένα επιπλέον αρχείο που υλοποιεί τις λειτουργίες που εξαρτώνται από το υλικό
- Οι οδηγοί επικοινωνούν με το σύστημα αρχείων με την χρήση του μηχανισμού ανταλλαγής μηνυμάτων
  - Ο οδηγός της κονσόλας υλοποιείται στο αρχείο */usr/src/drivers/tty/console.c*
  - Ο οδηγός του πληκτρολογίου υλοποιείται στο αρχείο */usr/src/drivers/tty/keyboard.c*
  - Ο οδηγός της θύρας RS232 υλοποιείται στο αρχείο */usr/src/drivers/tty/rs232.c*
- Στο αρχείο *tty.c* γίνεται ο έλεγχος των εισερχόμενων μηνυμάτων
- Στα αρχεία *console.c, keyboard.c, rs232.c* γίνεται ο χειρισμός της συσκευής

Θέματα Υλοποίησης Οδηγών Εισόδου / Εξόδου

- Ο τρόπος που υλοποιούνται οι οδηγοί των συσκευών ΕΕ στο Minix είναι χαρακτηριστικός
  - Διαφέρει ριζικά από τον τρόπο υλοποίησης άλλων UNIX/LINUX συστημάτων
  - Βασίζεται στην σχεδιαστική αρχή της εκτέλεσης των συνιστοσών του Λ.Σ. ως διαφορετικές διεργασίες στο user space
  - Είναι ιδιαίτερα ευέλικτος και επεκτάσιμος
  - Επιτυγχάνει μειωμένη απόδοση (σε σχέση με τα μονολιθικά συστήματα) λόγω της ανταλλαγής μηνυμάτων
- Μια διεργασία που θέλει να χρησιμοποιήσει μια μια συσκευή
  - Επικοινωνεί με το Σύστημα Αρχείων (αποστολή μηνύματος)
  - Το Σύστημα Αρχείων επικοινωνεί με τον Οδηγό της συσκευής (αποστολή μηνύματος)
  - Ο οδηγός επικοινωνεί με τον πυρήνα (αποστολή μηνύματος στο system task)
  - Ο πυρήνας επικοινωνεί με την συσκευή (επίπεδο υλικού)

2 Τρόποι Επικοινωνίας Χειριστών–Συστήματος



Βασικός Τρόπος Διεπαφής Οδηγών Εισόδου / Εξόδου

- Η διεπαφή/επικοινωνία με τους οδηγούς ΕΕ είναι σχετικά απλή
  - Η αίτηση για την χρήση της συσκευής στέλνεται μέσω μηνύματος στον αντίστοιχο οδηγό της συσκευής
  - Το μήνυμα αίτησης περιέχει διάφορα πεδία που περιγράφουν τον τύπο της χρήσης (π.χ. αν πρόκειται για READ ή WRITE)
  - Ο οδηγός επιχειρεί να εκτελέσει την εντολή (δηλ. να κάνει χρήση της συσκευής)
  - Απαντάει με το αποτέλεσμα μέσω ενός μηνύματος απάντησης
- Οι οδηγοί των συσκευών λαμβάνουν αιτήσεις κυρίως από το Σύστημα Αρχείων
  - Όλες οι οδηγοί για συσκευές block υλοποιούν την ίδια διαδικασία: παραλαβή μηνύματος αίτησης, εκτέλεση εντολής, αποστολή μηνύματος απάντησης
- Η εκτέλεση των εντολών γίνεται σειριακά και δεν περιέχουν εσωτερικά τεχνικές multiprogramming
  - Για λόγους απλοποίησης του κώδικα

Περιεχόμενα Μηνυμάτων Αίτησης σε Συσκευές Block

Πεδίο	Τύπος	Περιγραφή
m.m_type	int	Αίτηση λειτουργίας
m.DEVICE	int	Αριθμός συσκευής
m.PROC_NR	int	Διεργασία που κάνει την αίτηση
m.COUNT	int	Πλήθος byte ή κώδικας ioctl
m.POSITION	long	Θέση συσκευής
m.ADDRESS	char *	Θέση μνήμης στην διεργασία που κάνει την αίτηση

- Όταν ένας οδηγός περιμένει την ολοκλήρωση της εντολης από την συσκευή μεταβαίνει σε θέση *RECEIVE*
  - Δεν δέχεται νέες αιτήσεις
- Ο οδηγός για το τερματικό είναι ελαφρά διαφορετικός
  - Επιτρέπει νέες αιτήσεις για είσοδο από το πληκτρολόγιο όσο επεξεργάζεται κάποια αίτηση για είσοδο από μια άλλη συσκευή εισόδου (π.χ. σειριακή θύρα)

Περιεχόμενα Μηνυμάτων Απάντησης από Συσκευές Block

Πεδίο	Τύπος	Περιγραφή
m.m_type	int	Πάντα DRIVER_REPLY
m.REP_PROC_NR	int	Ίδιο με το πεδίο PROC_NR της αίτησης
m.REP_STATUS	int	Πλήθος byte που μεταφέρθηκαν ή κωδικός λάθους

- Τα περιεχόμενα των μηνυμάτων αίτησης/απάντησης που αφορούν συσκευές χαρακτήρων (character devices) είναι παρόμοια

Εσωτερική Δομή Οδηγών Εισόδου / Εξόδου

- Η εσωτερική δομή του κώδικα των οδηγών Εισόδου / Εξόδου είναι σχεδόν ίδια για όλους τους οδηγούς
- Μοιάζει με την δομή του διαχειριστή διεργασιών, συστήματος αρχείων κλπ.
- Η συνάρτηση main αρχικοποιεί τις εσωτερικές δομές
- Αποτελείται από έναν ατέρμονο βρόγχο
  - Αναμένει για το επόμενο μήνυμα (receive)
  - Επεξεργάζεται το μήνυμα
  - Στέλνει την απάντηση
- Οι συναρτήσεις *do_XXX* διαχειρίζονται τις λειτουργίες που υλοποιεί ο οδηγός
  - Επιστρέφουν ένα κωδικό που περιγράφει την κατάσταση της λειτουργίας

Παράδειγμα Κώδικα Οδηγών Εισόδου / Εξόδου

```
message mess;
void io_driver() {
  initialize(); /* only done once */
  while (TRUE) {
    receive(ANY, &mess);  /* wait for a request */
    caller = mess.source; /* sender process */
    switch(mess.type) {
      case READ:  rcode = dev_read(&mess); break;
      case WRITE: rcode = dev_write(&mess); break;
      /* Other cases go here, e.g. OPEN, CLOSE, IOCTL*/
      default:    rcode = ERROR;
    }
    mess.type = DRIVER_REPLY;
    mess.status = rcode;  /* result code */
    send(caller,&mess);
  }
}
```

Συσκευές Εισόδου / Εξόδου τύπου Block

- Το Minix 3 υποστηρίζει πολλούς τύπους συσκευών block
  - RAM disk, Hard disk, Floppy disk . . .
- Κάθε τύπος συσκευής έχει τις ιδιαιτερότητες του
- Οι συσκευές τύπου RAM disk είναι ένα καλό παράδειγμα
  - σχετικά εύκολη υλοποίηση εφόσον δεν υπάρχει κάποιο ‘πραγματικό’ υλικό
- Οι συσκευές τύπου Hard disk είναι ένα πλήρες παράδειγμα
  - υλοποιεί όλες τις λειτουργίες
  - υπάρχει ‘πραγματικό’ υλικό, οπότε μπορούμε να δούμε θέματα υλοποίησης που σχετίζονται με το υλικό
- Οι συσκευές τύπου Floppy disk έχουν αρκετές ιδιαιτερότητες
  - Παραδόξως δεν είναι ποιο εύκολη περίπτωση από τις παραπάνω
  - Οι μονάδες μπορούν να αλλάξουν/αφαιρεθούν
  - Το υλικό δεν προσφέρει πολλές λειτουργίες

Δομή Οδηγών Συσκευών Εισόδου / Εξόδου τύπου Block

- Για κάθε τύπο συσκευής υπάρχει και διαφορετικός οδηγός
- Μας ενδιαφέρει
  - Η εσωτερική δομή του κώδικα να είναι ίδια για όλους τους οδηγούς
  - Να υπάρχει μια κοινή διεπαφή ανεξάρτητη από τον τύπο της συσκευής
- Ο κώδικας της κοινής διεπαφής βρίσκεται στον φάκελο */usr/src/drivers/libdriver*
  - αρχεία *driver.c* και *drvlib.c*
- Ο οδηγός για μία συγκεκριμένη συσκευή προκύπτει από την ένωση
  - Της κοινής διεπαφής που είναι ανεξάρτητη του τύπου της συσκευής
  - Των υλοποιήσεων των λειτουργιών που εξαρτώνται από τον τύπο της συσκευής

Παράδειγμα Κώδικα Οδηγών Εισόδου / Εξόδου τύπου Block

```
message mess;
...
while (TRUE) {
  receive(ANY, &mess);  /* wait for a request */
  caller = mess.source; /* sender process */
  switch(mess.type) {
    case READ:
      rcode = (*entry_points->dev_read)(&mess); break;
    case WRITE:
      rcode = (*entry_points->dev_write)(&mess); break;
    /* Other cases go here, e.g. OPEN, CLOSE, IOCTL*/
    default:    rcode = ERROR;
  }
  mess.type = DRIVER_REPLY;
  mess.status = rcode;  /* result code */
  send(caller,&mess);
}
```

Κλήση Συναρτήσεων με Δείκτες

- Με την παραλαβή ενός μηνύματος (αίτησης) η κοινή διεπαφή ελέγχει τον τύπο της αίτησης και επιλέγει την κατάλληλη συνάρτηση
- Η κοινή διεπαφή δεν γνωρίζει ποια είναι η τελική συνάρτηση που πρέπει να εκτελεστεί
  - Γνωρίζει απλά 'τι πρέπει να κάνει' η συνάρτηση
- Η εκτέλεση της συγκεκριμένης κλήσης γίνεται με την χρήση δεικτών στην σωστή συνάρτηση
- Για τον κώδικα του παραδείγματος:
  `(*entry_points->dev_read)(&mess);`
- Είναι ένας έμμεσος/δυναμικός τρόπος να καλέσουμε μια συνάρτηση
  - Δεν χρειάζεται να 'γνωρίζουμε' το όνομα της συνάρτησης όταν κάνουμε compile
  - Διαφορετική συνάρτηση `dev_read` καλείται από διαφορετικούς οδηγούς συσκευών block

Βασικές Λειτουργίες Οδηγών συσκευών τύπου Block (1)

- Οι συσκευές τύπου block υποστηρίζουν τις ακόλουθες λειτουργίες
  1. *OPEN*
  2. *CLOSE*
  3. *READ*
  4. *WRITE*
  5. *IOCTL*
  6. *SCATTERED_IO*
- Οι λειτουργίες *READ* / *WRITE* αφορούν την μεταφορά δεδομένων από/προς την συσκευή προς/από τις θέσεις μνήμης της διεργασίας
- Η λειτουργία *READ* 'μπλοκάρει' έως ότου όλα τα δεδομένα μεταφερθούν
- Η λειτουργία *WRITE* επιστρέφει αμέσως και τα δεδομένα μεταφέρονται σε δεύτερο χρόνο

Βασικές Λειτουργίες Οδηγών συσκευών τύπου Block (2)

- Οι λειτουργίες *OPEN* / *CLOSE* αφορούν την ενεργοποίηση και απενεργοποίηση της συσκευής και προετοιμασία για χρήση
- Η λειτουργία *OPEN* ελέγχει κατά πόσο η συσκευή είναι προσβάσιμη
- Η λειτουργία *CLOSE* βεβαιώνεται ότι όλα τα δεδομένα έχουν μεταφερθεί και είναι ασφαλές να απενεργοποιηθεί η συσκευή
- Η λειτουργία *IOCTL* επιτρέπει τον έλεγχο των παραμέτρων λειτουργίας της συσκευής και την τροποποίηση τους – π.χ. αλλαγή στα partitions της συσκευής
- Η λειτουργία *SCATTERED_IO* αφορά τις συσκευές που είναι 'αργές' στην εκτέλεση των λειτουργιών (π.χ. οι συσκευές RAM disk δεν την χρειάζονται)
  - Επιτρέπει την ανάγνωση πολλαπλών block
  - Επιτρέπει την ανάγνωση υπό προυποθέσεις -- best effort

Χειρισμός Σημάτων

- Οι διεργασίες των οδηγών δεν θέλουμε να σταματάνε την λειτουργία τους όταν προκύπτει ένα σήμα
- Στα προηγούμενα μαθήματα είδαμε την επιλογή της αποστολής σημάτων με την μορφή μηνυμάτων
- 'Οταν πρέπει να σταλεί ένα σήμα σε κάποιον οδηγό, αυτό μετατρέπεται σε ένα μήνυμα με τους εξής τρεις τύπους
  1. *HARD_INT*
  2. *SYS_SIG*
  3. *SYN_ALARM*
- Επομένως η ροή εκτέλεσης του οδηγού δεν διακόπτεται
- Το εισερχόμενο σήμα/διακόπτης τοποθετείτε στην ουρά εισερχόμενων μηνυμάτων
- Ο χειρισμός είναι αντίστοιχος με αυτόν των βασικών λειτουργιών

Ορισμός Συναρτήσεων Χειρισμού Μηνυμάτων (1)

- Στο αρχείο */usr/src/drivers/libdriver/driver.h* ορίζεται η δομή *driver*

```
struct driver {
  _PROTOTYPE( char *(*dr_name), (void) );
  _PROTOTYPE( int (*dr_open),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_close),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_ioctl),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( struct device *(*dr_prepare),
              (int device) );
  _PROTOTYPE( int (*dr_transfer),
              (int proc_nr, int opcode, off_t position,
               iovec_t *iov, unsigned nr_req) );
  _PROTOTYPE( void (*dr_cleanup), (void) );
  ...
```

Ορισμός Συναρτήσεων Χειρισμού Μηνυμάτων (2)

```
  ...
  _PROTOTYPE( void (*dr_geometry),
              (struct partition *entry) );
  _PROTOTYPE( void (*dr_signal),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( void (*dr_alarm),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_cancel),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_select),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_other),
              (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_hw_int),
              (struct driver *dp, message *m_ptr) );
};
```

Παράδειγμα Ορισμού Συναρτήσεων

```
/* Entry points to this driver. */
PRIVATE struct driver m_dtab = {
  w_name, /* current device's name */
  w_do_open, /* open or mount request, initialize device
  w_do_close, /* release device */
  do_diocntl, /* get or set a partition's geometry */
  w_prepare, /* prepare for I/O on a given minor device
  w_transfer, /* do the I/O */
  nop_cleanup, /* nothing to clean up */
  w_geometry, /* tell the geometry of the disk */
  nop_signal, /* no cleanup needed on shutdown */
  nop_alarm, /* ignore leftover alarms */
  nop_cancel, /* ignore CANCELs */
  nop_select, /* ignore selects */
  w_other, /* catch-all for unrecognized commands and io
  w_hw_int /* leftover hardware interrupts */
};
```

Βασική Συνάρτηση Οδηγών Συσκευών τύπου Block

- Στο αρχείο */usr/src/drivers/libdriver/driver.h* ορίζεται η συνάρτηση *driver_task*

```
PUBLIC void driver_task(struct driver *dp) {
  /* Main program of any device driver task. */
  int r, proc_nr;
  message mess;

  init_buffer();  /* Get a DMA buffer. */

  while (TRUE) {
    /* Here is the main loop of the disk task.
     * It waits for a message, carries
     * it out, and sends a reply. */
  }
}
```

Παράδειγμα Οδηγού Συσκευής τύπου Block

```
PUBLIC int main(void)
{
  /* Install signal handlers.
   * Ask PM to transform signal into message. */
  struct sigaction sa;

  sa.sa_handler = SIG_MESS;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  if (sigaction(SIGTERM,&sa,NULL)<0)
    panic("AT","sigaction failed", errno);

  m_init();       /* Set special disk parameters */
  driver_task(&m_dtab);  /* start the main loop */
  return(OK);
}
```

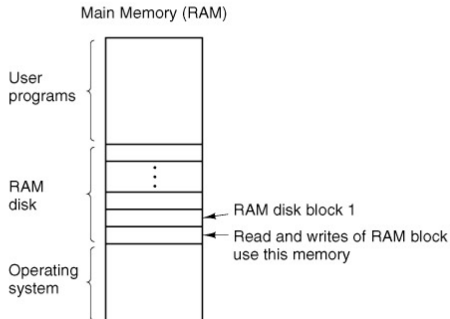
Συσκευές RAM Disk

- Θα εξετάσουμε τον οδηγό για συσκευές τύπου RAM disk
- Πρόκειται για εικονικούς δίσκους που χρησιμοποιούν την κεντρική μνήμη για προσωρινή αποθήκευση δεδομένων
- Συνήθως χρησιμοποιούμε RAM disk
  - για την αποθήκευση προσωρινών αρχείων (π.χ. φάκελος /tmp)
  - για την αντιγραφή των αρχείων ρυθμίσεων και άλλων αρχείων που χρησιμοποιούνται συχνά (π.χ. φάκελος /etc)
  - κατά την εγκατάσταση του συστήματος για την δημιουργία ενός προσωρινού συστήματος
- Βασικό πλεονέκτημα είναι η ταχύτητα προσπέλασης των δεδομένων και η ευκολία δημιουργίας
- Το πλήθος και το μέγεθος των RAM disk ορίζεται κατά την εκκίνηση του συστήματος
- Υπάγεται στον οδηγό μνήμης που παρέχει και άλλες λειτουργίες

Βασική Λειτουργία RAM Disk

- Μια συσκευή block είναι ένα αποθηκευτικό μέσο με δύο εντολές: ανάγνωση block και εγγραφή block
- Συνήθως αυτά τα block αποθηκεύονται σε κινούμενους μαγνητικούς δίσκους, π.χ. hard disks, cd
- Για την περίπτωση των RAM disk η αποθήκευση γίνεται απλά σε ένα μέρος της μνήμης που έχουμε ορίσει εκ των προτέρων
- Τα RAM disk έχουν το πλεονέκτημα της άμεσης πρόσβασης – δεν υπάρχει απώλεια χρόνου για τον εντοπισμό του block, την περιστροφή των δίσκων κλπ.
- Κατά την αρχικοποίηση ενός RAM disk διαιρούμε την διαθέσιμη μνήμη σε *n* blocks
  - κάθε block έχει μέγεθος αντίστοιχο με τους πραγματικούς δίσκους
- Όταν θέλουμε να γράψουμε σε ένα block, ο οδηγός εντοπίζει τη θέση μνήμης και αντιγράφει τα δεδομένα με την χρήση της *phys_copy* (υλοποιείται σε επίπεδο assembly)

Παράδειγμα ενός RAM Disk



Βασικές Λειτουργίες RAM disk (1)

- Ο οδηγός του RAM disk υλοποιεί 6 συσκευές
  1. */dev/ram*
  2. */dev/mem*
  3. */dev/kmem*
  4. */dev/null*
  5. */dev/boot*
  6. */dev/zero*
- Η συσκευή */dev/ram* είναι ένα πραγματικό RAM disk
- Το μέγεθος και η αρχική θέση μνήμης δεν ορίζονται στον οδηγό αλλά δίνονται ως παράμετροι κατά την εκτέλεση του οδηγού (εκκίνηση συστήματος)
  - Μπορούμε να αλλάξουμε το μέγεθος του RAM disk χωρίς να χρειαστεί να κάνουμε recompile το σύστημα
- Η συσκευή */dev/mem* επιτρέπει την εγγραφή/ανάγνωση στην κύρια μνήμη του συστήματος

Βασικές Λειτουργίες RAM disk (2)

- Η συσκευή */dev/kmem* επιτρέπει την εγγραφή/ανάγνωση στην κύρια μνήμη του συστήματος που αφορά τον πυρήνα του ΛΣ
- Οι συσκευές */dev/mem* και */dev/kmem* χρησιμοποιούνται για λόγους ελέγχου και αποσφαλμάτωσης
- Για τη συσκευή */dev/kmem* η θέση 0 αφορά την θέση 0 της μνήμης του πυρήνα – και το μέγεθος της συσκευής είναι ανάλογο με το μέγεθος του πυρήνα
- Μπορούμε να αποκτήσουμε πρόσβαση στη μνήμη του πυρήνα και με την χρήση της συσκευής */dev/mem* -- αρκεί να ξέρουμε το σημείο που ξεκινάει η μνήμη του πυρήνα
- Η συσκευή */dev/null* δέχεται δεδομένα και τα σβήνει αμέσως – είναι μονίμως άδεια (αν προσπαθήσουμε να διαβάσουμε θα πάρουμε μήνυμα λάθους EOF)
- Η συσκευή */dev/boot* δεν χρησιμοποιείτε στο Minix 3

Βασικές Λειτουργίες RAM disk (3)

- Η συσκευή */dev/zero* είναι αντίστοιχη με την συσκευή */dev/null* όταν πρόκειται να κάνουμε κάποια εγγραφή – σβήνει αμέσως τα δεδομένα
  - Όταν όμως κάνουμε ανάγνωση επιστρέφει ένα απεριόριστο πλήθος μηδενικών
  - Δεν πρόκειται ποτέ να επιστρέψει μήνυμα λάθους EOF
- Σε επίπεδο οδηγού συσκευής, οι συσκευές */dev/ram*, */dev/mem*, */dev/kmem* και */dev/boot* είναι πανομοιότυπες
  - Η μόνη διαφορά έχει να κάνει με την με την θέση μνήμης που πρέπει να διαβάσουν/γράψουν
  - Ορίζονται από τους πίνακες *ram_origin* και *ram_limit*
- Οι συσκευές */dev/null* και */dev/zero* `εμφανίζονται' ως συσκευές χαρακτήρων

Υλοποίηση Οδηγού RAM disk

- Όπως όλες οι συσκευές block ο κεντρικός βρόγχος ορίζεται στο αρχείο *driver.c*
- Οι συναρτήσεις που αφορούν τις λειτουργίες του οδηγού υλοποιούνται στο αρχείο */usr/src/drivers/memory/memory.c*
- Το εκτελέσιμο πρόγραμμα προκύπτει από το compilation των δύο αρχείων και το linking των επιμέρους αρχείων object
- Η δομή *device* που χρησιμοποιεί η συνάρτηση *driver_task* ορίζεται στο αρχείο *memory.c*
- Η δομή της κάθε συσκευής ορίζεται με την χρήση της δομής *device* ορίζεται στο αρχείο *driver.h*

```
struct device {
  u64_t dv_base; /* Base of a partition in bytes */
  u64_t dv_size; /* Size of a partition in bytes */
};
```

Ορισμός Συναρτήσεων RAM disk

```
/* Entry points to this driver. */
PRIVATE struct driver m_dtab = {
  m_name, /* current device's name */
  m_do_open, /* open or mount */
  do_nop, /* nothing on a close */
  m_ioctl, /* specify ram disk geometry */
  m_prepare, /* prepare for I/O on a given minor device
  m_transfer, /* do the I/O */
  nop_cleanup, /* no need to clean up */
  m_geometry, /* memory device "geometry" */
  nop_signal, /* system signals */
  nop_alarm,
  nop_cancel,
  nop_select,
  NULL,
  NULL
};
```

Κεντρική Συνάρτηση Οδηγού RAM Disk

```
PUBLIC int main(void)
{
  /* Disable signals -- convert them to messages */
  struct sigaction sa;

  sa.sa_handler = SIG_MESS;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  if (sigaction(SIGTERM,&sa,NULL)<0)
    panic("MEM","sigaction failed", errno);

  m_init();      /* Initialize the memory driver */
  driver_task(&m_dtab);  /* start the main loop. */
  return(OK);
}
```

Συνάρτηση *m_init* (1)

```
PRIVATE void m_init() {
  /* Initialize this task.
   * All minor devices are initialized one by one. */
  phys_bytes ramdev_size;
  phys_bytes ramdev_base;
  message m;
  int i, s;
  if (OK != (s=sys_getkinfo(&kinfo))) {
      panic("MEM","Couldn't get kernel information.",s);
  }
  /* Install remote segment for /dev/kmem memory. */
  m_geom[KMEM_DEV].dv_base = cvul64(kinfo.kmem_base);
  m_geom[KMEM_DEV].dv_size = cvul64(kinfo.kmem_size);
  if (OK != (s=sys_segctl(&m_seg[KMEM_DEV],
              (u16_t *) &s, (vir_bytes *) &s,
              kinfo.kmem_base, kinfo.kmem_size))) {
      panic("MEM","Couldn't install remote segment.",s);
  }
```

Συνάρτηση *m_init* (2)

```
  /* Install remote segment for
   * /dev/boot memory, if enabled. */
  m_geom[BOOT_DEV].dv_base = cvul64(kinfo.bootdev_base);
  m_geom[BOOT_DEV].dv_size = cvul64(kinfo.bootdev_size);
  if (kinfo.bootdev_base > 0) {
      if (OK != (s=sys_segctl(&m_seg[BOOT_DEV],
              (u16_t *) &s, (vir_bytes *) &s,
              kinfo.bootdev_base, kinfo.bootdev_size)))
          panic("MEM","Couldn't install remote segment."
      }
  }
```

Συνάρτηση *m_init* (3)

```
/* See if there are already RAM disk details
 * at the Data Store server. */
m.DS_KEY = MEMORY_MAJOR;
if (OK == (s = _taskcall(DS_PROC_NR, DS_RETRIEVE, &m))
  ramdev_size = m.DS_VAL_L1;
  ramdev_base = m.DS_VAL_L2;
  printf("MEM retrieved size %u and base %u from DS, s
          ramdev_size, ramdev_base, s);
  if (OK != (s=sys_segctl(&m_seg[RAM_DEV], (u16_t *) &
                (vir_bytes *) &s, ramdev_base, ramdev_s
      panic("MEM","Couldn't install remote segment.",s
  }
  m_geom[RAM_DEV].dv_base = cvul64(ramdev_base);
  m_geom[RAM_DEV].dv_size = cvul64(ramdev_size);
  printf("MEM stored retrieved details as new RAM disk
}
```

Συνάρτηση *m_init* (4)

```
/* Ramdisk image built into the memory driver */
m_geom[IMGRD_DEV].dv_base= cvul64(0);
m_geom[IMGRD_DEV].dv_size= cvul64(imgrd_size);

/* Initialize /dev/zero. Simply write zeros
 * into the buffer. */
for (i=0; i<ZERO_BUF_SIZE; i++) {
     dev_zero[i] = '\0';
}
```

Συνάρτηση *m_init* (5)

```
  /* Set up memory ranges for /dev/mem. */
#if (CHIP == INTEL)
  if (OK != (s=sys_getmachine(&machine))) {
    panic("MEM","Couldn't get machine information.",s);
  }
  if (! machine.prot) { /* 1M for 8086 systems */
    m_geom[MEM_DEV].dv_size = cvul64(0x100000);
  } else {
#if _WORD_SIZE == 2  /* 16M for 286 systems */
    m_geom[MEM_DEV].dv_size =  cvul64(0x1000000);
#else  /* 4G-1 for 386 systems */
    m_geom[MEM_DEV].dv_size = cvul64(0xFFFFFFFF);
#endif
  }
```

Συνάρτηση *m_name* και *m_prepare*

```
PRIVATE char *m_name() {
/* Return a name for the current device. */
  static char name[] = "memory";
  return name;
}
```

```
PRIVATE struct device *m_prepare(int device) {
/* Prepare for I/O on a device:
 * check if the minor device number is ok. */
  if (device < 0 || device >= NR_DEVS)
    return(NIL_DEV);
  m_device = device;
  return(&m_geom[device]);
}
```

Συνάρτηση *do_open*

```
PRIVATE int m_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
  int r;
/* Check device number on open. */
  if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO)
  if (m_device == MEM_DEV) {
    r = sys_enable_iop(m_ptr->IO_ENDPT);
    if (r != OK) {
      printf("m_do_open: sys_enable_iop failed for %d: %
      m_ptr->IO_ENDPT, r);
      return r;
    }
  }
  return(OK);
}
```

Συνάρτηση *m_geometry*

```
PRIVATE void m_geometry(entry)
struct partition *entry;
{
  /* Memory devices don't have a geometry,
   * but the outside world insists. */
  entry->cylinders =
       div64u(m_geom[m_device].dv_size, SECTOR_SIZE)
       / (64 * 32);
  entry->heads = 64;
  entry->sectors = 32;
}
```

Συνάρτηση *m_transfer* (1)

```
PRIVATE int m_transfer(proc_nr,opcode,position,iov,nr_re
int proc_nr;    /* process doing the request */
int opcode;     /* DEV_GATHER or DEV_SCATTER */
off_t position;/* offset on device to read or write */
iovec_t *iov;   /* pointer to read or write request vecto
unsigned nr_req;/* length of request vector */
{
/* Read or write one the driver's minor devices. */
  phys_bytes mem_phys;
  int seg;
  unsigned count, left, chunk;
  vir_bytes user_vir;
  struct device *dv;
  unsigned long dv_size;
  int s;
```

Συνάρτηση *m_transfer* (2)

```
  /* Get minor device number and check for /dev/null. */
  dv = &m_geom[m_device];
  dv_size = cv64ul(dv->dv_size);
  while (nr_req > 0) {
    /* How much to transfer and where to / from. */
    count = iov->iov_size;
    user_vir = iov->iov_addr;
    switch (m_device) {
    ...
    }
    /* Book the number of bytes transferred. */
    position += count;
    iov->iov_addr += count;
    if ((iov->iov_size -= count) == 0) { iov++; nr_req--
  }
  return(OK);
}
```

Συνάρτηση *m_transfer* (3)

```
/* No copying; ignore request. */
case NULL_DEV:
  if (opcode == DEV_GATHER) return(OK); /* always at EOF
  break;
/* Virtual copying. For RAM disk, kernel memory and boot
case RAM_DEV:
case KMEM_DEV:
case BOOT_DEV:
  if (position >= dv_size) return(OK);  /* check for EOF
  if (position + count > dv_size) count = dv_size - posi
  seg = m_seg[m_device];
  if (opcode == DEV_GATHER) { /* copy actual data */
    sys_vircopy(SELF,seg,position, proc_nr,D,user_vir, c
  } else {
    sys_vircopy(proc_nr,D,user_vir, SELF,seg,position, c
  }
  break;
```

Συνάρτηση *m_transfer* (4)

```
/* Physical copying. Only used to access entire memory.
case MEM_DEV:
    if (position >= dv_size) return(OK);  /* check for E
    if (position + count > dv_size) count = dv_size - po
    mem_phys = cv64ul(dv->dv_base) + position;

    if (opcode == DEV_GATHER) { /* copy data */
        sys_physcopy(NONE, PHYS_SEG, mem_phys,
         proc_nr, D, user_vir, count);
    } else {
        sys_physcopy(proc_nr, D, user_vir,
         NONE, PHYS_SEG, mem_phys, count);
    }
    break;
```

Συνάρτηση *m_transfer* (5)

```
/* Null byte stream generator. */
case ZERO_DEV:
    if (opcode == DEV_GATHER) {
        left = count;
     while (left > 0) {
         chunk = (left > ZERO_BUF_SIZE) ?
              ZERO_BUF_SIZE : left;
         if (OK != (s=sys_vircopy(SELF, D,
            (vir_bytes) dev_zero,
                 proc_nr, D, user_vir, chunk)))
             report("MEM","sys_vircopy failed", s);
         left -= chunk;
              user_vir += chunk;
     }
    }
    break;
```