Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Σχεδίαση και Υλοποίηση Οδηγού Συσκευής για τη Χρήση Προσαρμογέα Myrinet ως Αποθηκευτικού Μέσου Υψηλής Επίδοσης στο Λειτουργικό Σύστημα Linux

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Χ. Βενετσανόπουλος

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπ. Καθηγητής ΕΜΠ

Αθήνα, Μάρτιος 2010

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Σχεδίαση και Υλοποίηση Οδηγού Συσκευής για τη Χρήση Προσαρμογέα Myrinet ως Αποθηκευτικού Μέσου Υψηλής Επίδοσης στο Λειτουργικό Σύστημα Linux

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Χ. Βενετσανόπουλος

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπ. Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31/3/2010.

....	....	....
Νεκτάριος Κοζύρης Αναπ. Καθηγητής ΕΜΠ	Παναγιώτης Τσανάκας Καθηγητής ΕΜΠ	Νικόλαος Παπασπύρου Επίκ. Καθηγητής ΕΜΠ

Αθήνα, Μάρτιος 2010

. . . .

Κωνσταντίνος Χ. Βενετσανόπουλος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Copyright © Κωνσταντίνος Χ. Βενετσανόπουλος, 2010
Με επιφύλαξη παντός δικαιώματος. All rights reserved.
Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Έκδοση 2.1 από `cven`

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.9995.

Χρησιμοποιήθηκαν οι γραμματοσειρές Minion Pro και `Consolas`.

Περιεχόμενα

Περίληψη **ix**

Abstract **xi**

Ευχαριστίες **xiii**

1 Εισαγωγή **1**

1.1 Σκοπός 2

1.2 Δομή Κειμένου 4

2 Θεωρητικό Υπόβαθρο **7**

2.1 Ο πυρήνας Linux 8

2.1.1 Αρχιτεκτονική του πυρήνα 8

2.1.2 Ενότητες δυναμικής φόρτωσης (Modules) 12

2.1.3 Κλειδώματα (Spinlocks) 19

2.1.4 Χειρισμός Διακοπών (Interrupt Handling) 23

2.1.5 Οδηγοί συκευών block (Block Drivers) 27

2.2 Η διεπαφή Myrinet 47

2.2.1 Δικτύωση σε επίπεδο χρήστη (User Level Networking) 47

2.2.2 Δικτύωση με το στρώμα Myrinet/GM 50

3 Σχεδιασμός της συσκευής MyriBLK **61**

3.1 Γενικά 62

3.2 Αρχικές επιλογές σχεδίασης 63

3.3 Σχεδίαση από την πλευρά του κόμβου (host) 65
3.3.1 Επικοινωνία πυρήνα Linux με οδηγό block 67
3.3.2 Επικοινωνία οδηγού block με τον προσαρμογέα δικτύου . . . 70
3.4 Σχεδίαση από την πλευρά του προσαρμογέα δικτύου 72
3.4.1 Οργάνωση μνήμης SRAM και βασικές περιοχές 73
3.4.2 Μετακίνηση δεδομένων και επικοινωνία με οδηγό 76
3.5 Επικοινωνία των δύο πλευρών (Host - NIC) και συγχρονισμός 78
3.5.1 Συγχρονισμός Περιοδικού Ελέγχου (Polling) 79
3.5.2 Συγχρονισμός με χρήση διακοπών συστήματος (Interrupt driven) 81

4 Υλοποίηση της συσκευής MyriBLK 85

4.1 Πλατφόρμα εργασίας 86
4.2 Συγχώνευση με το GM 87
4.3 Υλοποίηση από την πλευρά του κόμβου (Host) 90
4.3.1 Διεπαφή οδηγού (MyriBLK module) με το GM 91
4.3.2 Διεπαφή οδηγού με επίπεδο block πυρήνα 95
4.3.3 Διεπαφή οδηγού με το firmware (MCP) 99
4.4 Υλοποίηση από την πλευρά του προσαρμογέα (NIC) 102
4.4.1 Αναδιοργάνωση της μνήμης SRAM 102
4.4.2 Μετακίνηση δεδομένων και επικοινωνία με οδηγό 105
4.5 Συγχρονισμός των δύο πλευρών (Host - NIC) 108
4.5.1 Συγχρονισμός Περιοδικού Ελέγχου (Polling) 108
4.5.2 Συγχρονισμός με χρήση διακοπών συστήματος (Interrupts) . . 110

5 Πειραματική Αξιολόγηση 115

5.1 Πειραματική Διαδικασία 116
5.2 Συμπεράσματα - Μελλοντική Εργασία 119

Περίληψη

Στην παρούσα εργασία σχεδιάζεται και υλοποιείται μία μονάδα αποθήκευσης πάνω σε έναν προσαρμογέα δικτύου υψηλής επίδοσης. Το περιβάλλον ανάπτυξης είναι ο πυρήνας Linux, όσον αφορά στο λειτουργικό σύστημα που εκτελείται στον κόμβο (host), και ο προσαρμογέας δικτύου (NIC) Myrinet-2000, όσον αφορά στο δίκτυο διασύνδεσης υψηλής επίδοσης που θα χρησιμοποιηθεί.

Η συσκευή MyriBLK σχεδιάζεται και υλοποιείται, ώστε να διαχειρίζεται μέρος της μνήμης SRAM του προσαρμογέα δικτύου Myrinet ως πρόσθετο σκληρό δίσκο του συστήματος, διαφανώς προς το υπόλοιπο σύστημα. Η μελέτη χωρίζεται σε τρία σημεία. Το πρώτο μέρος της συσκευής είναι ένας οδηγός συσκευής block, που εξάγει μία μονάδα δίσκου στο σύστημα και ταυτόχρονα εξυπηρετεί όλες τις αιτήσεις που αναφέρονται σε αυτή. Τις αιτήσεις αυτές είναι επίσης υπεύθυνος να προωθήσει με τον κατάλληλο τρόπο στον προσαρμογέα δικτύου. Το υλικολογισμικό που εκτελείται στον προσαρμογέα δικτύου, αποτελεί το δεύτερο μέρος της συσκευής και αναλαμβάνει την εξυπηρέτηση των αιτήσεων που παράγει ο οδηγός. Σχεδιάζεται και υλοποιείται για την πραγματική μεταφορά δεδομένων από και προς τη μνήμη του προσαρμογέα. Το τρίτο μέρος ασχολείται με το ζήτημα της επικοινωνίας και συγχρονισμού των δύο πρώτων μερών και περιγράφει δύο διαφορετικές μεθόδους επίλυσης.

Καθ’ όλη τη διάρκεια της εργασίας παρουσιάζονται τα προβλήματα σχεδιασμού που προκύπτουν σε κάθε επιμέρους στάδιο και αιτιολογούνται οι αντίστοιχες επιλογές. Τέλος, αποτιμάται πειραματικά η συσκευή σε ένα πραγματικό σύστημα, αξιολογούνται τα αποτελέσματα και προκύπτουν αντίστοιχα συμπεράσματα.

Λέξεις Κλειδιά: *Αποθηκευτικό Μέσο, Σκληρός Δίσκος, Συσκευή Block, Επίπεδο Block Linux, Οδηγός Συσκευής, Αιτήσεις, Μεταφορά Δεδομένων, Δίκτυο Διασύνδεσης Υψηλής Επίδοσης, Προσαρμογέας Δικτύου, Υλικολογισμικό, DMA, Myrinet, GM, Συστοιχίες Υπολογιστών, Δίαυλος Επικοινωνίας, Συγχρονισμός.*

Abstract

The objective of this study is the design and implementation of a storage drive, residing on a high performance - low latency Network Interface Card (NIC). The development framework is the Linux kernel as an Operating System on the host and the Myrinet-2000 NIC as the high performance interconnect.

The designed and implemented MyriBLK device, controls part of the NIC's onboard SRAM like an additional hard drive, transparently to the rest of the system. The study is divided into three parts. The first one constitutes of a block device driver, which exports a disk drive to the system and simultanesously serves all corresponding requests for this drive. The driver is also responsible for passing properly these requests to the network adapter. The firmware running on the network adapter is the second part of the device and undertakes the servicing of requests generated by the driver. It is designed and implemented in a way that enables it to carry out the actual data transfer to and from the adapter's onboard memory. The third part deals with the issue of communication and synchronization of the first two parts, and describes two different methods of solution.

Throughout the course of our work, the problems concerning the design will be presented in detail in any individual stage and the corresponding choices will be justified. Finally, an experimental valuation of the device in a real system will occur, providing results not only for further evaluation, but also as a source of useful conclusions.

Keywords: *Storage Media, Hard Disk, Block Device, Linux Block Layer, Device Driver, Requests, Data Transfer, High Performance Interconnection Networks, NIC, Firmware, DMA, Myrinet, GM, Computer Cluster, Transfer Rate, Synchronization.*

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων (http://www.cslab.ece.ntua.gr) της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου υπό την επίβλεψη του Αναπληρωτή Καθηγητή Νεκτάριου Κοζύρη και για την αποπεράτωσή της χρησιμοποιήθηκε μέρος του εξοπλισμού του εργαστηρίου.

Θα ήθελα να ευχαριστήσω θερμά τον κύριο Κοζύρη για την προτροπή του στο συγκεκριμένο θέμα εργασίας, την ευκαιρία που μου έδωσε να ασχοληθώ με τον αντίστοιχο τομέα της επιστήμης των υπολογιστών στο Εργαστήριο Υπολογιστικών Συστημάτων, καθώς και για τις πολύτιμες συμβουλές του κατά τη διάρκεια εκπόνησης της εργασίας.

Το θέμα της διπλωματικής εργασίας αποτελεί έμπνευση του Διδάκτορα Ευάγγελου Κούκη. Θα ήθελα να τον ευχαριστήσω, για την ανεκτίμητη βοήθεια σε όλα τα στάδια της εργασίας, για τις τεχνικές γνώσεις που μου μετέδωσε και για την υπομονή και ανιδιοτέλεια που έδειξε, καθ’ όλη τη διάρκεια της συνεργασίας μας. Χωρίς τη συμβολή του η διπλωματική εργασία αυτή δε θα ήταν εφικτή.

1 Εισαγωγή

Στις μέρες μας, η υπολογιστική ισχύς αυξάνεται με ραγδαίους ρυθμούς. Κλιμακώνεται δε, σε ακόμη μεγαλύτερο βαθμό, με τη χρήση παράλληλων αρχιτεκτονικών, που προτείνουν αύξηση της επίδοσης εκμεταλλευόμενες το συνδυασμό πολλών επεξεργαστών. Τέτοιου είδους συστήματα αποτελούν οι αρχιτεκτονικές συμμετρικής πολυεπεξεργασίας (Symetric Multiprocessing Systems – SMP) και οι αρχιτεκτονικές συστοιχιών υπολογιστών (Cluster Systems). Σημαντικά χαρακτηριστικά των συστοιχιών υπολογιστών είναι η μεγάλη δυνατότητα κλιμάκωσης και ο καλός λόγος κόστους προς απόδοση, ιδιότητες που καθιστούν αυτού του είδους τις αρχιτεκτονικές ελκυστικές σε συστήματα υψηλών επιδόσεων. Οι συστοιχίες υπολογιστών χρησιμοποιούνται κατά κόρον σήμερα, γεγονός που συνεπάγεται και τη συνεχή εξέλιξή τους.

Ιστορικά, παράλληλα με τις συστοιχίες υπολογιστών εξελίσσονται και οι υποδομές δικτύωσής τους. Το πρόβλημα διασύνδεσης των κόμβων επεξεργασίας είναι δισεπίλυτο, και η πολυπλοκότητα αυξάνεται ακόμη περισσότερο λόγω του μεγάλου αριθμού των κόμβων, που φτάνουν τις εκατοντάδες ή χιλιάδες. Ο μεγάλος αριθμός κόμβων σε συνδυασμό με τις υψηλές ταχύτητες επεξεργαστών δημιουργεί καταστάσεις συμφόρησης (bottlenecks). Παρόλα αυτά, η εξέλιξη των δικτύων διασύνδεσης έχει ακολουθήσει τους ταχύτατους ρυθμούς αύξησης της υπολογιστικής ισχύος, αλλά και των συστοιχιών υπολογιστών, με αποτέλεσμα σήμερα να χρησιμοποιούνται σύγχρονα δίκτυα με σημαντικά υψηλούς ρυθμούς μεταφοράς δεδομένων. Χαρακτηριστικές τιμές αγγίζουν τα 10 - 20Gbps σε σύγχρονα συστήματα υψηλών επιδόσεων. Φυσικά,η περαιτέρω εξέλιξη των δικτύων διασύνδεσης κρίνεται απαραίτητη, αν επιθυμούμε να πετύχουμε μελλοντική σύγκλιση με τις ταχύτητες επεξεργασίας, και ρυθμούς επεξεργασίας και μεταφοράς

δεδομένων συνεχώς αυξανόμενους. Μελετώντας την επεξεργασία και την μεταφορά δεδομένων, παρατηρούμε ότι το σοβαρό πρόβλημα έγκειται στο ρυθμό με τον οποίο αποθηκεύουμε τα δεδομένα αυτά.

Το σύνολο, του μεγάλου πλέον, όγκου δεδομένων που διακινούνται, αποθηκεύονται σε συγκεκριμένα αποθηκευτικά μέσα όταν δε μεταφέρονται ή υπόκεινται επεξεργασία. Τα σύγχρονα αυτά αποθηκευτικά μέσα δεν έχουν τους αντίστοιχους ρυθμούς μεταφοράς προς και από, το ζητούμενο προορισμό με αυτούς των δικτύων διασύνδεσης, γεγονός που κοστίζει πολύ στην τελική απόδοση του συστήματος και πολλές φορές ακυρώνει την ταχύτητα του δικτύου. Για την αποδοτική λύση του προβλήματος έχουν γίνει διάφορες προσπάθειες και έχει προταθεί μία σειρά διαφορετικών αρχιτεκτονικών καταμερισμού και διανομής των μέσων αποθήκευσης στο δίκτυο και στους κόμβους αντίστοιχα, χωρίς ωστόσο να έχουμε καταλήξει σε μεμονωμένο και ασφαλές συμπέρασμα για τον τρόπο χειρισμού του προβλήματος. Όσον αφορά σε συγκεκριμένα φυσικά μέσα, για την αντιμετώπιση του προβλήματος, έχουν προταθεί οι διατάξεις RAID (Redundant array of inexpensive disks) πολλαπλών σκληρών δίσκων ή η χρήση ταχέων δίσκων στερεάς κατάστασης (Solid State Disks - SSDs). Συνεχείς είναι οι προσπάθειες που γίνονται για την ανάδειξη νέων, επαρκώς οικονομικών και αντίστοιχα πρακτικών λύσεων, που θα μπορούν να θεωρηθούν ενδεδειγμένες προσεγγίσεις στο πρόβλημα.

Η παρούσα εργασία ασχολείται με το συγκεκριμένο ζήτημα αποθηκευτικών μέσων υψηλής επίδοσης και τη μεταφορά δεδομένων από/προς αυτά και εφαρμόζει μία διαφορετική προσέγγιση στην υλοποιήσή τους, μελετώντας ταυτόχρονα μία περίπτωση διαφορετικής αρχιτεκτονικής στην οποία το αποθηκευτικό μέσο εδράζεται πολύ κοντά στο φυσικό δίκτυο διασύνδεσης.

1.1 Σκοπός

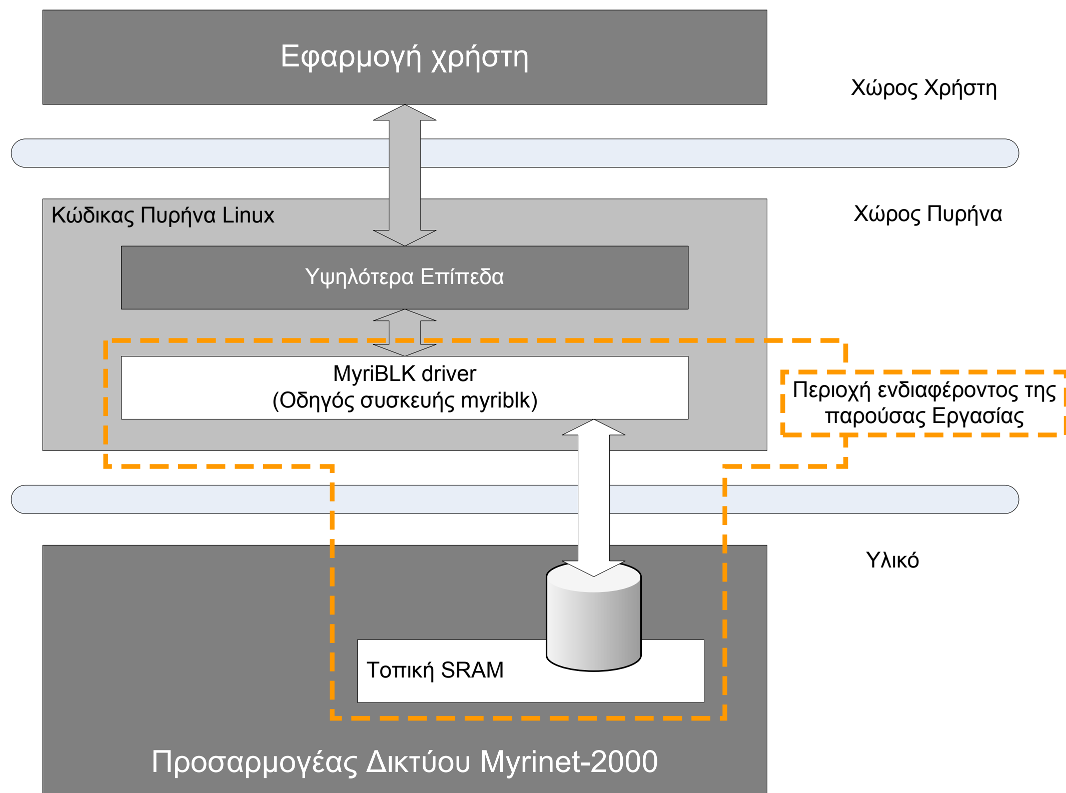
Στην παρούσα εργασία σχεδιάζουμε και υλοποιούμε μία μονάδα αποθήκευσης πάνω σε έναν προσαρμογέα δικτύου υψηλής επίδοσης Myrinet-2000. Πιο συγκεκριμένα θα εμφανίσουμε μέρος της τοπικής μνήμης SRAM του προσαρμογέα δικτύου σαν σκληρό δίσκο στο υπόλοιπο σύστημα, το οποίο θα μπορεί να αποθηκεύει εκεί δεδομένα, όπως ακριβώς θα έκανε και σε οποιοδήποτε άλλο πραγματικό δίσκο του συστήματος. Στόχος είναι ο σχεδιασμός και υλοποίηση ενός μηχανισμού που θα μας επιτρέπει να διαχειρι-

στούμε ένα αποθηκευτικό μέσο πολύ υψηλής επίδοσης, που θα έχει τη δυνατότητα να μεταφέρει δεδομένα με ταχύτητες που θα αγγίζουν τα όρια του ρυθμού μεταφοράς του διαύλου πάνω στον οποίο εδράζεται. Μία τέτοια υλοποίηση έχει ως σκοπό, τόσο την ενδελεχή μελέτη και κατανόηση διαφορετικών επιπέδων λειτουργιών του πυρήνα του λειτουργικού συστήματος και του δικτύου διασύνδεσης, όσο και την παρουσίαση μίας διαφορετικής αρχιτεκτονικής προσέγγισης αποθηκευτικού μέσου, στο πλαίσιο της γενικότερης έρευνας που λαμβάνει χώρα αυτή τη στιγμή γύρω από τα συστήματα συστοιχιών υπολογιστών υψηλής επίδοσης.

Το λειτουργικό σύστημα με το οποίο εργαζόμαστε είναι της οικογένειας UNIX και ο πυρήνας που τρέχει είναι ο Linux. Ο πυρήνας Linux αποτελεί ελεύθερο και ανοιχτό λογισμικό, που σημαίνει ότι μπορούμε να έχουμε πλήρη πρόσβαση σε κάθε κομμάτι του κώδικά του, αλλά και άδεια τροποποίησής του με κάθε τρόπο που εξυπηρετεί τις ανάγκες και το σκοπό μας. Παράλληλα καλύπτει όλες τις προδιαγραφές ενός μοντέρνου λειτουργικού συστήματος, όπως αυτές διαφαίνονται κατά την θεωρητική ανάλυση που θα ακολουθήσει, και έχει μεγάλη υποστήριξη από ανθρώπινο δυναμικό που το αναπτύσσει σε όλο τον κόσμο. Τα γεγονότα αυτά το καθιστούν ιδανικό ως πλατφόρμα για τη δική μας υλοποίηση.

Από την πλευρά του υλικού θα χρησιμποιήσουμε τον προσαρμογέα δικτύου υψηλής επίδοσης Myrinet-2000 της εταιρείας Myricom, ο οποίος είναι προγραμματιζόμενος και φέρει δική του αποκλειστική μνήμη. Για τον προσαρμογέα αυτόν έχουμε εξασφαλίσει πλήρη πρόσβαση στον κώδικα του οδηγού και των βιβλιοθηκών του (GM) αλλά και του υλικολογισμικού (firmware) που τρέχει στον μικροεπεξεργαστή του. Όπως θα παρουσιάσουμε στα επόμενα κεφάλαια, ο προσαρμογέας αυτός διαθέτει έναν μικροεπεξεργαστή RISC, μία μνήμη SRAM μεγέθους 2MBs και τρείς μηχανές DMA. Ακόμη συνεργάζεται με το υπόλοιπο σύστημα μέσω του διαύλου επικοινωνίας PCI/PCI-X. Τα χαρακτηριστικά αυτά προσπαθούμε να εκμεταλλευτούμε με τον αποδοτικότερο τρόπο κατά το σχεδιασμό, και μετέπειτα στην υλοποίηση.

Τελικά έχουμε ως στόχο, την μεταφορά δεδομένων από το σύστημα προς την μνήμη του προσαρμογέα και αντίστροφα, με αποδοτικούς ρυθμούς, κοντά σε αυτούς του διαύλου PCI. Το πλεονέκτημα μίας τέτοιας σχεδιαστικής προσέγγισης ενός αποθηκευτικού μέσου, είναι ότι η επίδοση κλιμακώνεται αναλογικά με την αύξηση του διαύλου, κάτι που δεν συναντούμε στις αρχιτεκτονικές που προαναφέρθηκαν. Στο Σχ. 1.1 φαίνεται



Σχήμα 1.1: *Περιοχή Ενδιαφέροντος Εργασίας*

το πλαίσιο στο οποίο εντάσσεται η παρούσα εργασία, καθώς και ο ακριβής χώρος με τον οποίο θα καταπιαστούμε.

1.2 Δομή Κειμένου

Για την υλοποίηση μίας πλήρως λειτουργικής συσκευής που θα ενσωματωθεί στο σύστημά μας και θα είναι σε θέση να εξυπηρετήσει όλες τις ανάγκες και απαιτήσεις που θα ορίσουμε, πρέπει να μελετήσουμε αρχικά μία σειρά ζητημάτων. Στο κεφάλαιο 2 γίνεται η περιγραφή των θεωρητικών εννοιών πάνω στις οποιές θα βασιστεί η σχεδίαση και υλοποίηση της συσκευής μας. Το όνομα της συσκευής είναι ”myriblk” και με αυτό το όνομα θα αναφερόμαστε σε αυτή στο εξής. Σκοπός του κεφαλαίου είναι η παρουσίαση των απαραίτητων θεωρητικών στοιχείων καθώς και του περιβάλλοντος εργασίας που καθόρισαν τις επιλογές μας και τον τρόπο προγραμματισμού. Το κεφάλαιο χωρίζεται σε δύο βασικές υποενότητες. Στην πρώτη αναλύεται ο πυρήνας Linux και δίνεται ιδιαίτερη έμφαση στα υποσυστήματα που συνδέονται άμεσα με το αντικείμενο ασχολίας μας,

και στη δεύτερη αναλύεται διεξοδικά η διεπαφή δικτύου Myrinet που θα αποτελέσει και μέρος του υλικού με το οποίο θα εργαστούμε.

Στο κεφάλαιο 3 παρουσιάζεται ο σχεδιασμός της συσκευής myriblk. Περιγράφεται η ακριβής λειτουργικότητα που θα προσφέρει η συσκευή, καθορίζονται οι απαιτήσεις με τις οποίες οφείλει να συμμορφώνεται και οι περιορισμοί που καλείται να αντιμετωπίσει. Επίσης σε κάθε περίπτωση περιγράφονται όλες οι σχεδιαστικές επιλογές και δικαιολογούνται χωριστά. Όπου κρίνεται αναγκαίο, παρουσιάζονται και οι εναλλακτικές τους. Καθ' όλη τη διάρκεια της διαδικασίας αυτής τίθενται και οι στόχοι που θα πρέπει να ικανοποιούν τα τελικά αποτελέσματα.

Στο κεφάλαιο 4 περιγράφεται η ακριβής υλοποίηση της συσκευής myriblk. Εδώ ερχόμαστε σε επαφή με τις συγκεκριμένες συναρτήσεις, δομές και μηχανισμούς που την υλοποιούν. Η υλοποίηση έχει ως στόχο την απόδειξη λειτουργίας του αρχικού μας μοντέλου και τη γενικότερη μελέτη της περίπτωσης μίας μονάδας δίσκου που βρίσκεται πάνω σε έναν προσαρμογέα δικτύου, με ό,τι πλεονεκτήματα ή δυσκολίες αυτή επιφέρει. Η υλοποίηση μας φέρνει επίσης άμεσα εκτεθειμένους σε προγραμματιστικά προβλήματα, που σχετίζονται με τα υποσυστήματα του πυρήνα αλλά και τους περιορισμούς του υλικού. Επιπροσθέτως, μας παρέχει τη δυνατότητα πειραματικής αξιολόγησης της εν λόγω σχεδίασης.

Η πειραματική αυτή αξιολόγηση λαμβάνει χώρα στο κεφάλαιο 5, όπου ελέγχεται η λειτουργία σε ένα πραγματικό σύστημα, παρουσιάζονται τα αποτελέσματα και καταλήγουμε σε χρήσιμα συμπεράσματα. Τέλος, στο κεφάλαιο 6 το οποίο αποτελεί τον επίλογο της εργασίας, προτείνουμε επεκτάσεις της εργασίας σε σημεία που θεωρούμε πως τις επιδέχεται και παρουσιάζουμε την βιβλιογραφία μελέτης. Με αυτόν τον τρόπο, θα γίνει μία προσπάθεια γενικότερης ένταξης της παρούσας εργασίας, στο πλαίσιο του αντικειμένου έρευνας, που επιτελείται το παρόν χρονικό διάστημα και ασχολείται με τις συστοιχίες υπολογιστών, τα δίκτυα διασύνδεσής τους και τα αποθηκευτικά μέσα αυτών.

2 Θεωρητικό Υπόβαθρο

Με το κεφάλαιο αυτό γίνεται μιά εισαγωγή στις έννοιες και τις δομές λειτουργίας στις οποιές βασίζεται η σχεδίαση και υλοποίηση της συσκευής myriblk. Σκοπός του κεφαλαίου είναι η παρουσίαση των απαραίτητων θεωρητικών στοιχείων, καθώς και του περιβάλλοντος εργασίας που καθόρισαν τις επιλογές μας και τον τρόπο προγραμματισμού. Το κεφάλαιο χωρίζεται σε δύο βασικά μέρη: στο πρώτο αναλύεται ο πυρήνας Linux με έμφαση στα σημεία που συνδέονται άμεσα με το αντικείμενο ασχολίας μας και στο δεύτερο γίνεται περιγραφή της διεπαφής δικτύου Myrinet.

Η εις βάθος κατανόηση του πυρήνα Linux αποτελεί προϋπόθεση για την υλοποίηση της συγκεκριμένης διπλωματικής εργασίας. Εξάλλου μεγάλο μέρος της θα αποτελεί κομμάτι αυτού. Για το λόγο αυτό, θα γίνει λεπτομερής περιγραφή των σημείων του πυρήνα που έρχονται σε άμεση επαφή με τα δικά μας κομμάτια κώδικα, όπως επίσης και μια γενικότερη περιγραφή των δομών λειτουργίας, των υποσυστημάτων και των επιπέδων διασύνδεσής του, που θα δίνουν μια ολοκληρωμένη εικόνα του προγραμματιστικού περιβάλλοντος πάνω στο οποίο καλούμαστε να σχεδιάσουμε.

Εξίσου σημαντική είναι και η κατανόηση της διεπαφής Myrinet ως δικτύου διασύνδεσης και του προγραμματιστικού περιβάλλοντος GM στο οποίο βασίζεται. Στο κεφάλαιο αυτό θα αναλυθεί τόσο το υλικό και η αρχιτεκτονική του, όσο το υλικολογισμικό που τρέχει στη διεπαφή αλλά και το καθαρά λογισμικό μέρος που αποτελεί το συνδετικό κρίκο με τον πυρήνα του λειτουργικού συστήματος. Αναγκαία είναι η πλήρης κατανόηση και των τριών μερών ξεχωριστά, όπως επίσης και οι τρόποι με τον οποίο αυτά συνδέονται. Έτσι γίνονται φανερές οι εξειδικευμένες δυνατότητες που προσφέρει το υλικό που έχουμε στη διάθεση μας και καλούμαστε να διαχειριστούμε, τις οποίες θα

προσπαθήσουμε με τη σειρά μας να αναπτύξουμε σε ακόμη μεγαλύτερο βαθμό κατά τη διάρκεια της εργασίας αυτής.

2.1 Ο πυρήνας Linux

Linux ονομάζεται ο πυρήνας μιας οικογένειας λειτουργικών συστημάτων που μοιάζουν με το λειτουργικό σύστημα AT&T Unix και ο όρος χρησιμοποιείται συχνά για να προσδιορίσει και το ίδιο το λειτουργικό σύστημα στο οποίο εκτελείται. Ο κώδικας του Linux έχει γραφτεί από την αρχή και αποτελεί ελεύθερο και ανοιχτό λογισμικό, το οποίο πλέον αναπτύσσεται από εθελοντές σε όλο τον κόσμο.Τη στιγμή που γράφεται η εργασία αυτή, τελεί υπό την άδεια GNU General Public License version 2 (GPLv2) και στοχεύει στα πρότυπα POSIX και Single Unix Specification.

Το μεγαλύτερο κομμάτι είναι γραμμένο στη γλώσσα προγραμματισμού C και έχει όλα τα χαρακτηριστικά ενός μοντέρνου συστήματος Unix: πολυεπεξεργασία, εικονική μνήμη, μοιραζόμενες βιβλιοθήκες, φόρτωση κατά απαίτηση, κανονική διαχείριση μνήμης και δικτύωση που ικανοποιεί τα πρότυπα IPv4 και IPv6. Αν και αρχικά αναπτύχθηκε σε αρχιτεκτονική 32-bit x86 PC σήμερα υποστηρίζει δεκάδες διαφορετικές αρχιτεκτονικές τόσο στα 32 όσο και στα 64-bit. Μερικές από αυτές είναι: Alpha AXP, Sun SPARC, Motorola 68000, PowerPC, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, AMD x86-64, AXIS CRIS, Renesas M32R. Οι διαφορές ανάμεσα στις αρχιτεκτονικές είναι εκτός του στόχου της παρούσας διπλωματικής εργασίας (η οποία υλοποιήθηκε σε 32-bit x86 PC) γι'αυτό και δεν θα αναλυθούν περαιτέρω.

Στη συνέχεια ξεκινάμε με μια γενική περιγραφή της αρχιτεκτονικής του πυρήνα Linux στην οποία μελετάμε τη δομή, τη λειτουργία και τους τρόπους διασύνδεσής του με το υπόλοιπο λειτουργικό σύστημα, τις εφαρμογές αλλά και το υλικό. Ακολουθεί λεπτομερής περιγραφή των υποσυστημάτων που μας ενδιαφέρουν κάθε ένα ξεχωριστά.

2.1.1 Αρχιτεκτονική του πυρήνα

Σε ένα σύστημα Unix, διάφορες ταυτόχρονες διεργασίες αναλαμβάνουν για διάφορες εργασίες. Κάθε διεργασία κάνει αιτήσεις για να εκμεταλλευτεί πόρους του συστήματος όπως: υπολογιστική ισχύ, μνήμη, δικτύωση, πρόσβαση σε κάποια συσκευή. Ο πυρήνας

είναι το μεγάλο κομμάτι του κώδικα που πρέπει να χειριστεί όλες αυτές τις αιτήσεις.

Ο πυρήνας Linux είναι ένας μονολιθικός πυρήνας (monolithic kernel) όπου όλο το λειτουργικό σύστημα τρέχει μόνο του στο χώρο πυρήνα (kernel space) με αυξημένα δικαιώματα.Σε αντίθεση με άλλου είδους αρχιτεκτονικές, ο μονολιθικός πυρήνας ορίζει από μόνος του μία υψηλού επιπέδου εικονική διεπαφή πάνω από το υλικό. Με αυτή την διεπαφή παρέχει όλες τις αναγκαίες υπηρεσίες ενός λειτουργικού συστήματος μέσα από ένα σύνολο κλήσεων συστήματος (system calls). Ο ρόλος του πυρήνα και οι υπηρεσίες που αυτός παρέχει, μπορούν να ομαδοποιηθούν στα παρακάτω μέρη:

Διαχείριση Διεργασιών Οι διεργασίες (processes) που βρίσκονται μέσα στον πυρήνα συνήθως ονομάζονται νήματα (threads) αλλά μιας και η υλοποιήση του Linux δεν διαχωρίζει τους όρους, στο εξής θα αναφερόμαστε και εμείς σε αυτές ως "διεργασίες (processes)". Ο πυρήνας είναι υπεύθυνος για τη δημιουργία και την καταστροφή διεργασιών. Επίσης είναι υπεύθυνος για την μεταξύ τους επικοινωνία και συγχρονισμό (π.χ. μέσω σημάτων, σωληνώσεων), κάτι πολύ βασικό για την ολική λειτουργικότητα του συστήματος. Αυτές οι λειτουργίες γίνονται διαθέσιμες και στις εφαρμογές που τρέχουν στο χώρο χρήστη (user space) μέσω μιας διεπαφής προγράμματος εφαρμογής (API) που χρησιμοποιεί τις κλήσεις συστήματος που προαναφέρθηκαν. Ακόμη με τον όρο διαχείριση διεργασιών αναφερόμαστε στη πολύ σημαντική λειτουργία που έχει να κάνει με τον τροπο καταμερισμού του χρόνου που αφιερώνει η κεντρική μονάδα επεξεργασιας (CPU) σε κάθε διεργασία. Ο πυρήνας υλοποιεί ένα καινοτόμο αλγόριθμο δρομολόγησης που καταμερίζει δίκαια το χρόνο σε όλες τις διεργασίες. Ο δρομολογητής (scheduler) αυτός, λειτουργεί και σε αρχιτεκτονικές που εμπλέκουν άνω του ενός επεξεργαστή (συστήματα συμμετρικής πολυεπεξεργασίας - SMPs). Τελικά, η διαχείριση διεργασιών του πυρήνα υλοποιεί ένα αφηρημένο επίπεδο διεργασιών πάνω από έναν ή περισσότερους επεξεργαστές.

Διαχείριση Μνήμης Η μνήμη του υπολογιστή αποτελεί μείζονα πόρο και η τακτική με την οποία αντιμετωπίζεται, είναι κρίσιμης σημασίας για την ολική επίδοση του συστήματος. Ο πυρήνας χτίζει ένα εικονικό χώρο μνήμης για να χρησιμοποιήσουν οι διεργασίες, πάνω από τον περιορισμένο φυσικό χώρο του συστήματος. Με αυτό τον τρόπο οι διεργασίες βλέπουν ως διαθέσιμο, μόνο τον εικονικό χώρο μνήμης και τα διάφορα μέρη του πυρήνα αλληλεπιδρούν με το υποσύστη-

μα διαχείρισης μνήμης μέσω συγκεκριμένου συνόλου συναρτήσεων. Για λόγους αποδοτικότητας η μνήμη διαχωρίζεται σε σελίδες (pages) διαφορετικού μεγέθους ανάλογα με την αρχιτεκτονική (συνήθως 4KB). Ο πυρήνας χρησιμοποιεί και πρόσθετα επίπεδα αφαίρεσης εκτός των 4KB buffers που με πιο σύνθετες μεθόδους (slab allocator) παρακολουθεί ποιές σελίδες είναι γεμάτες και ποιές όχι και αυξάνει ή μειώνει δυναμικά τους αντίστοιχους χώρους, ανάλογα με τις ανάγκες του ευρύτερου συστήματος. Ένα τελευταίο σημείο που αξίζει να σημειωθεί, είναι η λειτουργία ανταλλαγής (swapping), η οποία χρησιμοποιείται όταν τα αποθέματα μνήμης τελειώνουν, αφού υποστηρίζονται και πολλαπλοί χρήστες μνήμης. Στην περίπτωση αυτή, ο πυρήνας μεταφέρει ολόκληρες σελίδες από τη μνήμη στο δίσκο και αντίστροφα. Απο αυτή την αμφίδρομη διαδικασία παίρνει και το όνομά της η λειτουργία.

Συστήματα Αρχείων Το Unix βασίζεται σε μεγάλο βαθμό στην έννοια του συστήματος αρχείων. Σχεδόν τα πάντα μπορούν να μεταχειριστούν σαν αρχεία. Ο πυρήνας χτίζει ένα δομημένο σύστημα αρχείων πάνω από μη δομημένο υλικό και αυτό το επίπεδο αφαίρεσης χρησιμοποιείται βαρέως σε όλη την έκταση του συστήματος. Ο πυρήνας Linux υποστηρίζει πολλαπλούς τύπους συστημάτων αρχείων, δηλαδή διαφορετικούς τρόπους οργάνωσης δεδομένων σε ένα φυσικό μέσο. Το πρότυπο τη στιγμή της συγγραφής είναι το ext3, αλλά οι επιλογές είναι αρκετές ανάλογα με τη χρήση. Μια άλλη ενδιαφέρουσα άποψη του Linux, είναι το εικονικό σύστημα αρχείων (virtual filesystem VFS), γιατί παρέχει μια αφηρημένη κοινή διεπαφή για όλα τα συστήματα αρχείων. Το VFS βρίσκεται ανάμεσα στη διεπαφή κλήσεων συστήματος (system call interface) και στο πραγματικό σύστημα αρχείων. Με αυτό τον τρόπο, παρέχεται άλλο ένα επίπεδο αφαίρεσης στις εφαρμογές, με συναρτήσεις όπως οι: open, close, read, write και έτσι δεν είναι αναγκαίο οι εφαρμογές να γνωρίζουν τον συγκεκριμένο τύπο συστήματος αρχείων στο οποίο αναφέρονται και υπάρχει σε χαμηλότερο επίπεδο. Στο χαμηλότερο άκρο του συστήματος αρχείων, υπάρχει μια ενδιάμεση κρυφή μνήμη (page cache) που χωρίζει το σύστημα αρχείων από τον οδηγό συσκευής και είναι ανεξάρτητη από τον τύπο του συστήματος αρχείων. Στόχο έχει να βελτιστοποιεί την πρόσβαση στο φυσικό μέσο, κρατώντας δεδομένα κοντά στα παραπάνω επίπεδα, ή ακόμα και διαβάζοντας εκ των προτέρων δεδομένα που έχουν μεγάλη πιθανότητα να ζητηθούν σε μελλοντικό χρόνο.

Έλεγχος Συσκευών Το μεγαλύτερο ποσοστό όσον αφορά στην ποσότητα του πηγαίου κώδικα του πυρήνα Linux βρίσκεται αδιαμφισβήτητα στους οδηγούς συσκευών, που χρησιμεύουν για να κάνουν ένα συγκεκριμένο υλικό χρήσιμο. Σχεδόν κάθε λειτουργία συστήματος τελικά καθρεφτίζεται σε μία φυσική συσκευή. Με εξαίρεση τον επεξεργαστή, τη μνήμη και ελάχιστες ακόμη οντότητες του συστήματος, όλες οι λειτουργίες που ελέγχουν κάποια συσκευή εκτελούνται από κώδικα ο οποίος είναι ειδικός για τη συγκεκριμένη συσκευή και αναφέρεται σε αυτή. Ο κώδικας αυτός ονομάζεται οδηγός συσκευής (device driver) και ένα μεγάλο κομμάτι της δικής μας υλοποιήσης για τη διπλωματική εργασία, εμπλέκει τέτοιου είδους κώδικα, όπως θα φανεί στη συνέχεια. Ο πυρήνας έχει ενσωματωμένο ένα οδηγό συσκευής για κάθε περιφερειακό που βρίσκεται παρών στο σύστημα. Οι οδηγοί συσκευών διαφέρουν σε μεγάλο βαθμό μεταξύ τους, το οποίο είναι λογικό αν αναλογιστεί κανείς πόσα διαφορετικά είδη υλικού υπάρχουν και τις αντίστοιχες ανάγκες που καλούνται να ικανοποιήσουν οι οδηγοι συσκευών. Αυτό βέβαια δεν σημαίνει πως δεν υπάρχουν κάποιες γενικά αποδεκτές πρακτικές (που ακολουθεί και το περιβάλλον του Linux) κάτω από τις οποιές καλούμαστε να σχεδιάσουμε και να υλοποιήσουμε και οι οποίες θα παρουσιαστούν στα αντίστοιχα σημεία, δικαιολογώντας πολλές φορές τις επιλογές μας.

Δικτύωση Για τη διαχείριση της δικτύωσης υπεύθυνο είναι το λειτουργικό σύστημα, γιατί οι περισσότερες λειτουργίες δικτύου δεν αφορούν σε μία συγκεκριμένη διεργασία: τα εισερχόμενα πακέτα είναι γεγονότα που συμβαίνουν ασύγχρονα. Υπάρχουν πολλές διαδικασίες που πρέπει να γίνουν πριν τα πακέτα παραδοθούν σε μια διεργασία για να τα αναλάβει εκείνη μετέπειτα. Τέτοιες διαδικασίες είναι η συλλογή των πακέτων, η αναγνώρισή τους και η κατάλληλη αποστολή τους. Το σύστημα είναι υπεύθυνο ώστε να παραδοθούν σωστά όλα τα πακέτα στις κατάλληλες διεπαφές δικτύου ή εφαρμογών και ταυτόχρονα για τον έλεγχο της εκτέλεσης των προγραμμάτων ανάλογα με τη δραστηριότητα του δικτύου. Επιπροσθέτως, το σύνολο της δρομολόγησης και ανάλυσης διευθύνσεων υλοποιείται στον πυρήνα. Η διεπαφή του υποσυστήματος δικτύου με τα προγράμματα εφαρμογών είναι οι υποδοχείς δικτύου (sockets). Το επίπεδο αυτό βρίσκεται ακριβώς κάτω από το επίπεδο κλήσεων συστήματος και παρέχει στο χρήστη πρόσβαση σε πολλά είδη πρωτοκόλλων δικτύου, καθώς και έναν προτυποποιημένο τρόπο διαχείρισης συνδέσεων και μεταφοράς δεδομένων από ένα τελικό άκρο σε ένα

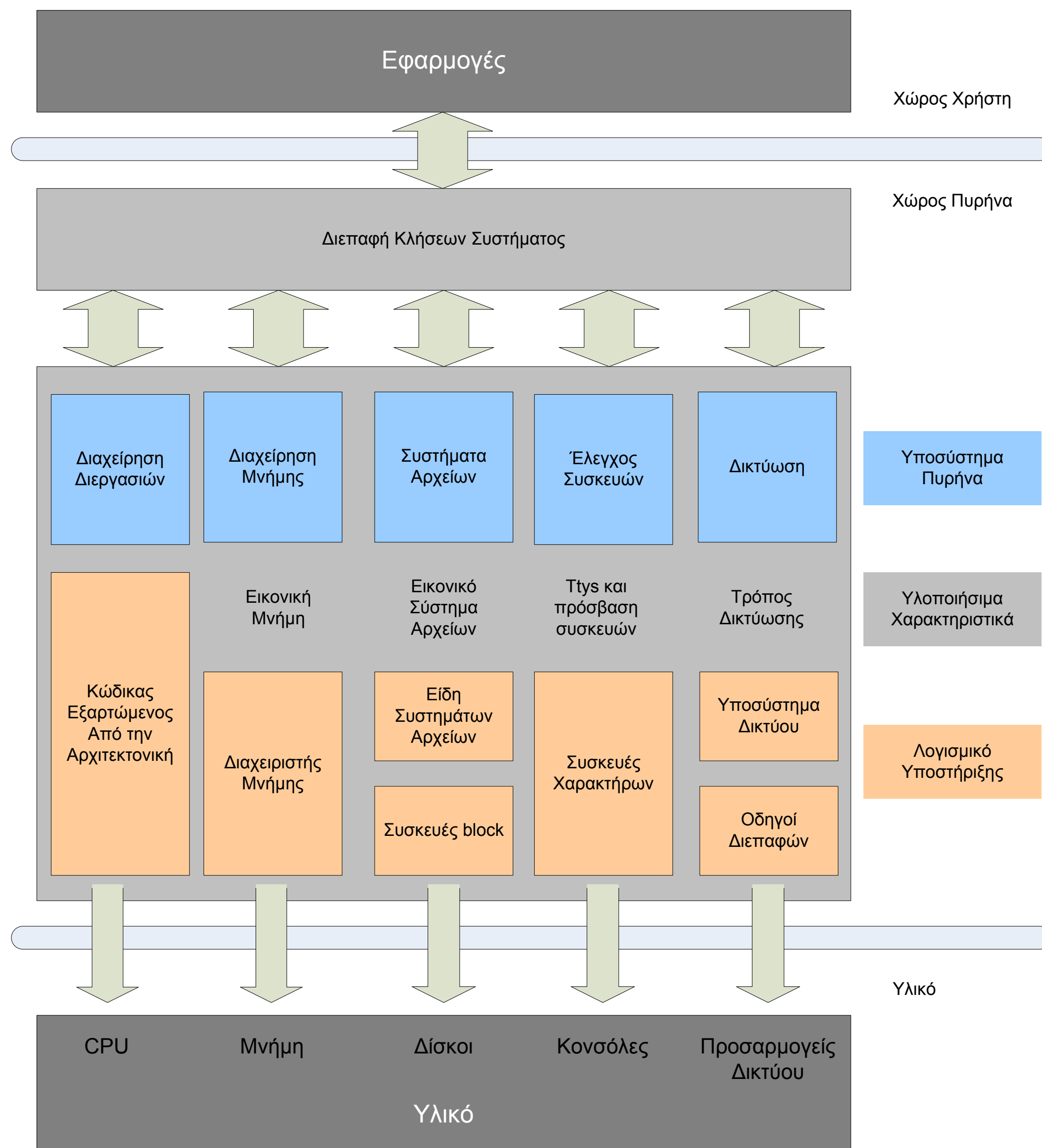
άλλο. Η ίδια η στοίβα δικτύου εξάλλου είναι υλοποιημένη με μία αρχιτεκτονική πολλαπλών επιπέδων, ακολουθώντας το μοντέλο σχεδίασης των πρωτοκόλλων.

Κώδικας εξαρτώμενος από την αρχιτεκτονική Κλείνοντας μια γενικότερη περιγραφή, να σημειώσουμε ότι αν και το μεγαλύτερο κομμάτι του πυρήνα Linux είναι ανεξάρτητο της αρχιτεκτονικής πάνω από την οποία τρέχει, υπάρχουν στοιχεία του πυρήνα που είναι αναγκαίο να λάβουν υπ' όψιν την υποκαθήμενη αρχιτεκτονική για να εξασφαλιστεί η ομαλή λειτουργία αλλά και η αποδοτικότητα. Τα κομμάτια του πηγαίου κώδικα που είναι υπεύθυνα για αυτές τις λειτουργίες βρίσκονται κάτω από τον υποφάκελο ./linux/arch του δένδρου του πηγαίου κώδικα και έχουν να κάνουν με χαμηλού επιπέδου λειτουργίες κυρίως σχετιζόμενες με συγκεκριμένο υλικό (εκκίνηση, επεξεργαστή, μνήμη).

2.1.2 Ενότητες δυναμικής φόρτωσης (Modules)

Ένα από τα πολύ σημαντικά χαρακτηριστικά του Linux, είναι η δυνατότητα που έχει ο πυρήνας να επεκτείνει το σύνολο των λειτουργιών που προσφέρει, κατά τη διάρκεια που αυτός τρέχει. Αυτό σημαίνει ότι μπορούμε ανά πάσα στιγμή που το σύστημα τρέχει να προσθέσουμε λειτουργικότητα στον πυρήνα (όπως και να αφαιρέσουμε). Κάθε κομμάτι κώδικα που μπορεί να προστεθεί στον πυρήνα τη στιγμή που αυτός τρέχει, ονομάζεται module. Ο πυρήνας Linux υποστηρίζει αρκετούς διαφορετικούς τύπους (ή κλάσεις) τέτοιων ενοτήτων κώδικα (modules) και ένας από αυτούς είναι και οι οδηγοί συσκευών. Κάθε module απαρτίζεται από αντικειμενικό κώδικα (object code), που μπορεί να συνδεθεί στον τρέχοντα πυρήνα και να αποσυνδεθεί από αυτόν δυναμικά, κατά τον χρόνο εκτέλεσης. Η εισαγωγή και αφαίρεσης γίνεται με χρήση των insmod και rmmod αντίστοιχα.

Οι διαφορετικές κλάσεις στις οποίες κατηγοριοποιούνται αυτές οι ενότητες κώδικα αντικατοπτρίζουν ουσιαστικά τις διαφορετικές λειτουργίες που έχουν σκοπό να επιτελέσουν. Η φιλοσοφία προγραμματισμού σήμερα, τείνει να μετατρέπει όλο και μεγαλύτερο κομμάτι της λειτουργικότητας του Linux σε τέτοιου είδους κώδικα. Αυτή η τάση έχει να κάνει σε μεγάλο βαθμό με το γεγονός ότι η αποσύνθεση σε μικρότερες δομικές μονάδες, αποτελεί στοιχείο κλειδί για καλύτερη κλιμάκωση και μεγαλύτερη επεκτασιμότητα. Παρόλα αυτά, μπορούμε να κάνουμε ένα βασικό διαχωρισμό στον τρόπο με

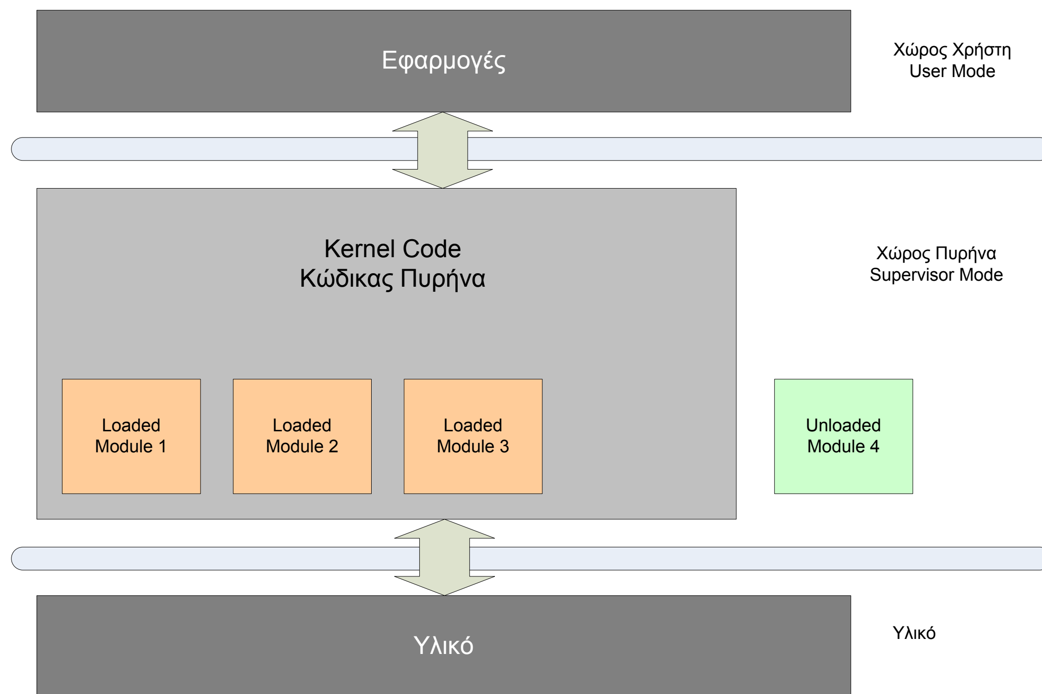


Σχήμα 2.1: *Αρχιτεκτονική Πυρήνα Linux*

τον οποίο ο πυρήνας βλέπει τις συσκευές και αφού συνήθως κάθε module παρέχει τα μέσα για μία τέτοια συσκευή, να διαχωρίσουμε αντίστοιχα και τα modules. Έτσι λοιπόν έχουμε συσκευές χαρακτήρων, συσκευές block και συσκευές δικτύου. Αντίστοιχα χαρακτηρίζουμε τις ενότητες κώδικα σε char modules, block modules και network modules. Φυσικά παράλληλα με τους οδηγούς συσκευών υπάρχουν και άλλες λειτουργίες, τόσο υλικού όσο και λογισμικού που έχουν τη μορφή ενοτήτων δυναμικής φόρτωσης στον πυρήνα. Ένα κοινό παράδειγμα είναι τα συστήματα αρχείων. Ο τύπος του συστήματος αρχείων έχει να κάνει με τον τρόπο που οργανώνεται η πληροφορία σε μία block συσκευή, ώστε να παριστάνει ένα δένδρο από φακέλους και αρχεία. Αυτή η οντότητα

δεν είναι οδηγός συσκευής, αλλά οδηγός λογισμικού, γιατί παρουσιάζει τις χαμηλού επιπέδου δομές δεδομένων σε υψηλού επιπέδου δομές.

Πριν προχωρήσουμε στην περιγραφή κάποιων βασικών δομών και τρόπων λειτουργιάς που ισχύουν για όλες τις ενότητες δυναμικής φόρτωσης, θα ανφέρουμε μερικές διαφορές μεταξύ ενός kernel module και μιας εφαρμογής. Ενώ οι περισσότερες μικρές και μεσαίες εφαρμογές εκτελούν ένα μόνο έργο από την αρχή έως το τέλος, κάθε kernel module απλά δηλώνεται στον πυρήνα, με σκοπό να εξυπηρετήσει μελλοντικά αιτήματα και η συνάρτηση αρχικοποίησής του τερματίζει άμεσα μετά την φόρτωσή του. Ο ρόλος δηλαδή της συνάρτησης αρχικοποίησης, είναι να προετοιμάσει την μετέπειτα κλήση του συνόλου των συναρτήσεων που υλοποιούνται στο module. Η συνάρτηση εξόδου καλείται ακριβώς πριν αποφορτωθεί το module. Αυτός ο τρόπος προσέγγισης στον προγραμματισμό είναι παρόμοιος με τον προγραμματισμό υποκινούμενο από γεγονότα (event-driven programming), που δεν συναντάται σε όλες τις εφαρμογές, αλλά σε όλα τα kernel modules. Μία άλλη μεγάλη διαφορά μεταξύ των event-driven εφαρμογών και του κώδικα του πυρήνα, βρίσκεται στην συνάρτηση εξόδου. Ενώ η εφαρμογή που τερματίζει μπορεί να δίνει λίγη ή καθόλου σημασία στην αποδέσμευση των πόρων που έχει δεσμεύσει, η συνάρτηση εξόδου ενός module πρέπει πολύ προσεκτικά να αναιρέσει όλα αυτά που δημιούργησε η συνάρτηση αρχικοποίησης και ίσως και άλλες συναρτήσεις του σώματος του module. Σε αντίθετη περίπτωση τα υπολείμματα παραμένουν παρόντα, μέχρι την επόμενη επανεκκίνηση του συστήματος. Επίσης, οι εφαρμογές μπορούν να καλέσουν συναρτήσεις τις οποίες δεν έχουν ορίσει. Αυτό συμβαίνει γιατί μπορούν να γίνουν συνδέσεις με βιβλιοθήκες συναρτήσεων. Τα modules αντίθετα, είναι συνδεδεμένα μόνο με τον πυρήνα και έτσι μπορούν να καλέσουν μόνο συναρτήσεις που αυτός εξάγει. Ακόμη, μεταξύ προγραμματισμού στον πυρήνα και προγραμματισμού εφαρμογών σημειώνεται διαφοροποιήση σχετικά με τον τρόπο που το κάθε περιβάλλον χειρίζεται τα λάθη. Ένα λάθος κατάτμησης (segmentation fault) κατά τη διάρκεια ανάπτυξης μιας εφαρμογής είναι ακίνδυνο, όχι όμως και ένα λάθος πυρήνα που στην καλύτερη περίπτωση σκοτώνει την τρέχουσα διεργασία, αν όχι όλο το σύστημα.



Σχήμα 2.2: *Modules Πυρήνα*

Χώρος χρήστη και Χώρος πυρήνα (User space and Kernel Space)

Ένα module τρέχει στο χώρο πυρήνα, ενώ οι εφαρμογές τρέχουν στο χώρο χρήστη. Αυτή η ιδέα βρίσκεται στη βάση της θεωρίας λειτουργικών συστημάτων. Ο ρόλος του λειτουργικού συστήματος είναι να παρέχει στις εφαρμογές μια συνεπή άποψη του υλικού του υπολογιστικού συστήματος. Επιπροσθέτως το λειτουργικό σύστημα πρέπει να λαμβάνει υπ' όψιν τις ανεξάρτητες λειτουργίες των προγραμμάτων και την προστασία από αναρμόδια πρόσβαση στους εκάστοτε πόρους. Αυτό το σημαντικό έργο είναι δυνατό να επιτελεστεί, μόνο αν η CPU μπορεί να επιβάλει προστασία του λογισμικού συστήματος από αυτό των εφαρμογών. Κάθε μοντέρνος επεξεργαστής όμως έχει τη δυνατότητα να επιβάλει αυτή τη συμπεριφορά. Με αυτόν τον τρόποξ έχουμε διαφορετικά επίπεδα λειτουργίας μέσα στον ίδιο τον επεξεργαστή. Τα συστήματα Unix έχουν σχεδιαστεί για να εκμεταλλεύονται αυτό το χαρακτηριστικό του υλικού, χρησιμοποιώντας δύο τέτοια επίπεδα. Ο πυρήνας εκτελείται στο υψηλότερο επίπεδο (supervisor mode), όπου τα πάντα επιτρέπονται, σε αντίθεση με τις εφαρμογές που τρέχουν στο χαμηλότερο επίπεδο (user mode), όπου ο επεξεργαστής ρυθμίζει την άμεση πρόσβαση στο υλικό και την αναρμόδια πρόσβαση στη μνήμη. Συνήθως, αναφερόμαστε στους δύο αυτούς

τρόπους εκτέλεσης σαν ”χώρο πυρήνα” (kernel space) και ”χώρο χρήστη” (user space). Οι όροι αυτοί δεν περικλείουν μόνο τα διαφορετικά επίπεδα δικαιωμάτων που είναι σύμφυτα με τους δύο τρόπους εκτέλεσης, αλλά και το γεγονός ότι κάθε τρόπος έχει τη δική του απεικόνιση μνήμης και το δικό του χώρο διευθύνσεων. Το Unix μεταφέρει την εκτέλεση από το χώρο χρήστη στο χώρο πυρήνα κάθε φορά που μία εφαρμογή κάνει μία κλήση συστήματος ή όταν λαμβάνει χώρα μία διακοπή υλικού (με τις διακοπές υλικού θα ασχοληθούμε αναλυτικά σε επόμενη ενότητα). Ο κώδικας πυρήνα που εκτελεί μία κλήση συστήματος δουλεύει στο πλαίσιο της διεργασίας. Λειτουργεί εκ μέρους της διεργασίας που έκανε την κλήση και έχει τη δυνατότητα πρόσβασης στο δικό της χώρο μνήμης. Από την άλλη πλευρά, ο κώδικας που χειρίζεται διακοπές είναι ασύγχρονος, αντιμετωπίζει ισότιμα τις υπόλοιπες διεργασίες και δεν σχετίζεται με καμία συγκεκριμένη από αυτές.

Ο ρόλος του module όπως προαναφέραμε είναι να επεκτείνει την λειτουργικότητα του πυρήνα. Έτσι ο κώδικας των ενοτήτων αυτών δυναμικής φόρτωσης, όπως φαίνεται και από την παραπάνω ανάλυση, τρέχει στο χώρο πυρήνα. Συνήθως, ένας οδηγός (όπως και αυτός που θα υλοποιήσουμε) εκτελεί και τα δύο έργα που περιγράφηκαν παραπάνω: μερικές συναρτήσεις του module εκτελούνται ως μέρος κλήσεων συστήματος και κάποιες άλλες είναι υπεύθυνες για τον χειρισμό των διακοπών. Στη συνέχεια παραθέτουμε μερικές βασικές συναρτήσεις και λειτουργίες που συναντώνται σε όλα τα modules και έχουν χρησιμοποιηθεί και στο δικό μας σχεδιασμό και υλοποίηση.

Αρχικοποίηση και Τερματισμός

Όπως προαναφέραμε, κατά τη φόρτωση του module στον πυρήνα (insmod), η συνάρτηση αρχικοποίησης που είναι η πρώτη που εκτελείται δηλώνει το module και καταχωρεί κάθε παροχή που προσφέρει αυτό.Με τον όρο παροχή, εννοούμε κάθε καινούρια λειτουργικότητα στην οποία μπορούν να έχουν πρόσβαση οι εφαρμογές. Ο ορισμός της συνάρτησης αρχικοποίησης έχει την παρακάτω μορφή:

```
static int __init initialization_function(void)
{
          /*
           *   Κώδικας αρχικοποίησης
           */
}
module_init(initialization_function);
```

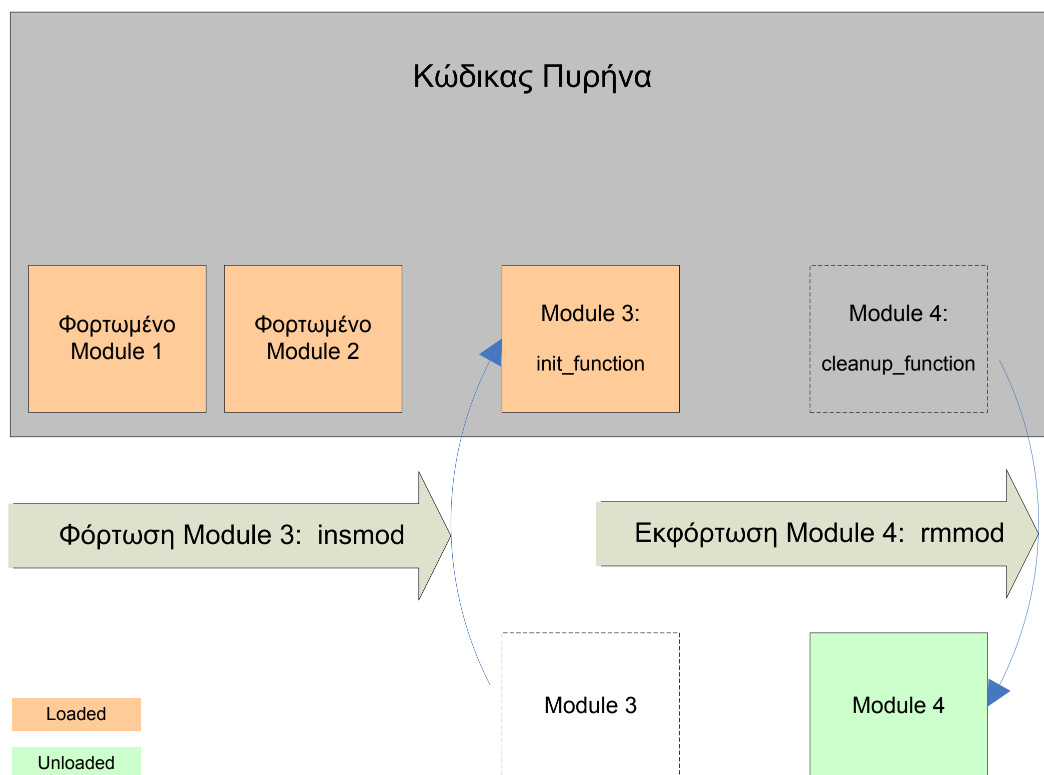
Στον ορισμό αυτό η ένδειξη `__init` γνωστοποιεί στον πυρήνα πως η συνάρτηση που

ακολουθεί χρησιμοποιείται μόνο κατά τη διάρκεια αρχικοποίησης. Η χρήση της δεν είναι απαραίτητη, αλλά προτιμάται γιατί με αυτό τον τρόπο ο φορτωτής του module την διαγράφει μετά την πλήρη φόρτωση του module και η μνήμη που δέσμευε γίνεται διαθέσιμη για άλλες χρήσεις. Η χρήση της `module_init()` είναι υποχρεωτική. Αυτό το macro προσθέτει μία ειδική ενότητα στο object code του module που δηλώνει που βρίσκεται η συνάρτηση αρχικοποίησης. Χωρίς αυτό τον ορισμό, η συνάρτηση αρχικοποίησης δεν καλείται ποτέ. Αντίστοιχα έχουμε και τη συνάρτηση εκκαθάρισης:

```
static void __exit cleanup_function(void)
{
          /*
           *   Κώδικας εκκαθάρισης
           */
}
module_exit(cleanup_function);
```

Η συνάρτηση εκκαθάρισης καταργεί τις εγγραφές διεπαφών και επιστρέφει όλους τους πόρους που δέσμευε το module πίσω στο σύστημα, πρίν αυτό εκφορτωθεί. Η ένδειξη `__exit` σημειώνει ότι ο κώδικας αυτός χρησιμοποιείται μόνο κατά την εκφόρτωση και έτσι ο compiler τοποθετεί τον κώδικα σε συγκεκριμένο σημείο (ELF section). Ο ορισμός της `module_exit()` είναι και πάλι απαραίτητος και δηλώνει στον πυρήνα που βρίσκεται η συνάρτηση εκκαθάρισης. Στην περίπτωση που δεν οριστεί συνάρτηση εκκαθάρισης, ο πυρήνας δεν επιτρέπει στο module να αφαιρεθεί.

Στο σημείο αυτό, πρέπει να σημειώσουμε μία άλλη πολύ σημαντική πτυχή που έχει να κάνει με τη φόρτωση module: κατάσταση συναγωνισμού (race condition). Αν η συνάρτηση αρχικοποίησης δεν γραφτεί με απόλυτη προσοχή, υπάρχει περίπτωση να δημιουργηθούν καταστάσεις που διακινδυνεύεται η σταθερότητα ολόκληρου του συστήματος. Αρχικά πρέπει να λαμβάνεται υπ' όψιν το γεγονός, ότι ο πυρήνας μπορεί να κάνει χρήση οποιασδήποτε παροχής τη στιγμή που ολοκληρώνεται η δήλωσή της. Αυτό σημαίνει ότι υπάρχει περίπτωση να έχουμε κλήσεις από τον πυρήνα χωρίς να έχει προλάβει να ολοκληρωθεί η συνάρτηση αρχικοποίησης. Ο κώδικας θα πρέπει να είναι σε θέση να ικανοποιήσει κάθε τέτοια κλήση, επομένως θα πρέπει να είναι απολύτως λειτουργικός μετά από κάθε δήλωση. Καμία παροχή δε δηλώνεται, αν δεν έχει ολοκληρωθεί πρώτα όλη η αρχικοποίηση που υποστηρίζει την συγκεκριμένη παροχή. Ένα άλλο θέμα, έχει να κάνει με τη διαδικασία που ακολουθείται σε περίπτωση που η συνάρτηση αρχικοποίησης αποφασίσει πως πρέπει να εκπέσει. Στο σημείο αυτό οι πιθανότητες είναι πολλές, ο πυρήνας να χρησιμοποιεί μια παροχή που είχε δηλωθεί νωρίτερα και είναι λειτουργι-



Σχήμα 2.3: *Εισαγωγή και Αφαίρεση Modules*

κή. Σε αυτή την περίπτωση η συνάρτηση αρχικοποίησης δεν πρέπει να εκπέσει, μέχρι να ολοκληρωθούν οι υπόλοιπες λειτουργίες που λαμβάνουν χώρα σε άλλο σημείο του πυρήνα.

Παραμετροποίηση του Module

Αν ένα module είναι σωστά σχεδιασμένο, μπορεί να ρυθμιστεί την στιγμή που φορτώνεται στον πυρήνα. Η στιγμή φόρτωσης δίνει πολύ μεγαλύτερη ευελιξία στο χρήστη από τη ρύθμιση τη στιγμή της μεταγλώττισης, η οποία χρησιμοποιείται ακόμη σε μερικές περιπτώσεις. Πολλοί παράμετροι που χρειάζεται να γνωρίζει ένας οδηγός αλλάζουν από σύστημα σε σύστημα. Ο πυρήνας δίνει τη δυνατότητα στον οδηγό να προσδιορίσει αυτές τις παραμέτρους κατά τη φόρτωση του module. Οι τιμές των παραμέτρων αναθέτονται κατά τη φόρτωση από το πρόγραμμα insmod (ή το modprobe). Για να είναι όμως αυτό δυνατό και να μπορεί το insmod να αλλάζει τις τιμές των παραμέτρων, πρέπει πρώτα να τις κάνει διαθέσιμες το ίδιο το module. Αυτό γίνεται με τη χρήση του `macro module_param`. Οι παράμετροι δηλώνονται με το `module_param` το οποίο παίρ-

νει με τη σειρά του τρεις παραμέτρους: το όνομα της μεταβλητής, τον τύπο της και την μάσκα δικαιωμάτων (permission mask). Τα macro τοποθετούνται έξω από κάθε συνάρτηση και παραδοσιακά βρίσκονται στην αρχή κάθε αρχείου πηγαίου κώδικα. Μια τυπική χρήση macro είναι η εξής:

```
static int example_param = 1;
module_param(example_param, int, S_IRUGO);
```

Στην πρώτη γραμμή αρχικοποιούμε την μεταβλητή `example_param` στην τιμή 1, η οποία θα χρησιμοποιηθεί στην περίπτωση που δεν δώσουμε τιμή κατά την φόρτωση. Στη δεύτερη γραμμή έχουμε τον πραγματικό ορισμό της μεταβλητής ως `module parameter`. Το πρώτο πεδίο είναι το όνομα της μεταβλητής που θέλουμε να παραμετροποιήσουμε, το δεύτερο πεδίο είναι ο τύπος της και η τιμή του τρίτου πεδίου ελέγχει ποιός έχει πρόσβαση στην απεικόνιση της παραμέτρου στο sysfs. Η παράμετρος παίρνει τιμή 20 στην γραμμή εντολών δίνοντας:

```
insmod module_name example_param=20
```

2.1.3 Κλειδώματα (Spinlocks)

Στην ενότητα αυτή θα ασχοληθούμε με το μηχανισμό των κλειδωμάτων (spinlocks), θα εξηγήσουμε το ρόλο που επιτελούν, πως ορίζονται και πως λειτουργούν, μια περιγραφή που κρίνεται αναγκαία αφού θα τα συναντούμε καθ' όλη τη διάρκεια της εργασίας. Η διαχείριση του συγχρονισμού (τι γίνεται όταν πολλές ταυτόχρονες διεργασίες επιθυμούν πρόσβαση σε κοινούς πόρους του συστήματος) είναι ένα από τα θεμελιώδη προβλήματα κατά των προγραμματισμό λειτουργικών συστημάτων. Ο πυρήνας Linux έχει εξελιχθεί σε σημείο που να επιτρέπει πάρα πολλά πράγματα να γίνονται ταυτόχρονα, για να μπορέσει να ανταποκριθεί στο μοντέρνο υλικό, τις πολυάριθμες εφαρμογές και τις απαιτήσεις αυτών. Η εξέλιξη αυτή έχει οδηγήσει σε μέγιστη βελτίωση σε θέματα επίδοσης και κλιμάκωσης.

Σε ένα μοντέρνο σύστημα Linux, υπάρχουν πολυάριθμες πηγές συγχρονισμού, με αποτέλεσμα να δημιουργούνται καταστάσεις συναγωνισμού (race conditions). Πολλαπλές διεργασίες χώρου-χρήστη τρέχουν ταυτόχρονα και έχουν πρόσβαση σε κοινό κώδικα χώρου-πυρήνα με πολλαπλούς τρόπους. Συστήματα SMP μπορεί να εκτελούν επίσης αυτό τον κώδικα ταυτόχρονα σε διαφορετικούς επεξεργαστές. Ακόμη, κώδικας οδη-

γού μπορεί να χάσει οποιαδήποτε στιγμή τον επεξεργαστή και τη θέση του να πάρει κώδικας διεργασίας, που τρέχει και αυτή μέσα στον ίδιο οδηγό. Οι διακοπές συσκευών είναι και αυτές ασύγχρονα γεγονότα, που μπορούν να προκαλέσουν ταυτόχρονη εκτέλεση κώδικα οδηγού. Γίνεται φανερό πως υπάρχουν πολυάριθμοι τρόποι (με αυτούς που αναφέρθηκαν να αποτελούν ένα μικρό υποσύνολο) που μπορούν να οδηγήσουν σε κατάσταση συναγωνισμού. Οι καταστάσεις συναγωνισμού είναι σε μεγάλο βαθμό αποτέλεσμα μοιραζόμενης πρόσβασης σε κοινούς πόρους. Όταν δύο διαφορετικά νήματα έχουν λόγο να δουλέψουν με τις ίδιες δομές δεδομένων (ή πόρους υλικού αντίστοιχα), η ενδεχόμενη σύγχυση έχει μεγάλη πιθανότητα να συμβεί. Για τους λόγους αυτούς ο πυρήνας παρέχει διάφορα εργαλεία για να διαχειριστούμε καταστάσεις συναγωνισμού, που το καθένα καλύπτει διαφορετικές ανάγκες.

Από τα διάφορα εργαλεία, στη συγκεκριμένη εργασία θα επικεντρωθούμε στα spinlocks γιατί συναντώνται πιο συχνά στον πυρήνα, καταλήγουν σε καλύτερες επιδόσεις αν χρησιμοποιηθούν σωστά και τελικά είναι αυτά που θα χρησιμοποιήσουμε και στη δική μας υλοποίηση. Ένας επίσης σημαντικός λόγος είναι ότι τα spinlocks, σε αντίθεση με άλλα εργαλεία, μπορούν να χρησιμοποιηθούν σε κώδικα ο οποίος δεν κοιμάται, όπως είναι οι χειριστές διακοπών. Ο μηχανισμός των spinlocks είναι πολύ απλός σαν έννοια. Το spinlock είναι μία συσκευή αμοιβαίου αποκλεισμού (mutual exclusion) που μπορεί να πάρει δύο τιμές: κλειδωμένη (locked)" και "ξεκλείδωτη (unlocked)". Συνήθως υλοποιείται σαν το μοναδικό bit μιας ακέραιας τιμής. Ο κώδικας που θέλει να πάρει ένα συγκεκριμένο lock εξετάζει το σχετικό bit. Αν το lock είναι διαθέσιμο, τότε τίθεται το bit "locked" και ο κώδικας μπορεί να συνεχίσει στο κρίσιμο τμήμα. Αντίθετα, αν το lock έχει τεθεί από κάποιον άλλο, τότε ο κώδικας μπαίνει σε ένα σφιχτό βρόγχο που ελέγχει επανειλημμένως το lock, μέχρι αυτό να γίνει διαθέσιμο. Αυτός ο βρόγχος είναι και το "spin" κομμάτι του spinlock. Φυσικά, η πραγματική υλοποίηση του spinlock είναι αρκετά πιο πολύπλοκη από την παραπάνω περιγραφή, αλλά είναι πέραν του σκοπού αυτής της εργασίας, αφού η γενική ιδέα του μηχανισμού και οι αρχές λειτουργίας του αρκούν για τη σωστή επιλογή και χρήση του στον κώδικα οδηγών.

Το αρχείο που ορίζει τα spinlocks είναι το `<linux/spinlock.h>` και το lock είναι τύπου `spinlock_t`. Όπως και κάθε άλλη δομή δεδομένων έτσι και το spinlock πρέπει να αρχικοποιηθεί σωστά για να μπορεί να χρησιμοποιηθεί. Αυτό μπορεί να γίνει είτε κατά τη μεταγλώττιση:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

είτε κατά την εκτέλεση:

```
void spin_lock_init(spinlock_t *lock);
```

Όταν έχουμε ένα κώδικα που θέλει να εισέλθει στο κρίσιμο τμήμα, τότε πρέπει να εξασφαλίσει το απαιτούμενο lock με:

```
void spin_lock(spinlock_t *lock);
```

Για να αποδεσμεύσει ο κώδικας το lock καλεί την:

```
void spin_unlock(spinlock_t *lock);
```

Να σημειώσουμε ότι όλες οι αναμονές που γίνονται με spinlocks δεν διακόπτονται. Αν κληθεί η `spin_lock` τότε ο κώδικας τρέχει (spins) μέχρι να γίνει διαθέσιμο το lock. Υπάρχουν και άλλες συναρτήσεις spinlock, μερικές από τις οποίες θα αναφέρουμε στη συνέχεια, αλλά καμία δεν αποκλίνει από τη βασική ιδέα, που παρουσιάστηκε στις παραπάνω συναρτήσεις. Εξάλλου, περιορίζονται σε ελάχιστα αυτά που μπορεί να κάνει κάποιος με ένα spinlock, εκτός από το να το κλειδώσει και να το αποδεσμεύσει. Συνολικά υπάρχουν τέσσερις παραλλαγές που κλειδώνουν ένα lock:

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
```

Η συνάρτηση `spin_lock` είδαμε πως λειτουργεί.Η `spin_lock_irqsave` απενεργοποιεί τις διακοπές (μόνο στον τοπικό επεξεργαστή) πριν δεσμεύσει το lock και η προηγούμενη κατάσταση διακοπών αποθηκεύεται στο flags. Αν είμαστε απολύτως σίγουροι ότι κανένας άλλος δεν έχει απενεργοποιήσει τις διακοπές στον επεξεργαστή πριν από μας (δηλαδή θα χρειαστεί να τις επαναφέρουμε μετά την αποδέσμευση του spinlock), μπορούμε να χρησιμοποιήσουμε την `spin_lock_irq`. Τέλος η `spin_lock_bh` απενεργοποιεί τις διακοπές λογισμικού πριν τη δέσμευση και αφήνει τις διακοπές υλικού ενεργοποιημένες. Για να αποφευχθεί πλήρες αδιέξοδο (deadlock), πρέπει να χρησιμοποιούνται οι συναρτήσεις που απενεργοποιούν τις διακοπές, για κάθε spinlock το οποίο μπορεί να δεσμευτεί και από κώδικα που τρέχει σε πλαίσιο διακοπών (interrupt context). Αντίστοιχα με την συνάρτηση που χρησιμοποιήσαμε για τη δέσμευση, πρέπει να χρησιμοποιήσουμε μία από τις παρακάτω συναρτήσεις για την αποδέσμευση του lock:

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Κάθε μορφή της `spin_unlock` αναιρεί τις ενέργειες της αντίστοιχης `spin_lock`. Η μεταβλητή `flags` που περνιέται στην `spin_unlock_irqrestore` πρέπει να είναι η ίδια με αυτή που έχει περαστεί προηγουμένως στη `spin_lock_irqsave`. Πρέπει επίσης, οι συναρτήσεις `spin_lock_irqsave` και `spin_unlock_irqrestore` να καλούνται μέσα στην ίδια συνάρτηση, σε διαφορετική περίπτωση υπάρχει περίπτωση ο κώδικας να σπάσει (break) σε κάποιες αρχιτεκτονικές. Ο πυρήνας παρέχει επιπροσθέτως, μία μορφή spinlocks τύπου ανάγνωσης/γραφής (reader/writer). Τα locks αυτά επιτρέπουν σε έναν αριθμό αναγνωστών να εισέλθουν στο κρίσιμο τμήμα ταυτόχρονα, αλλά όχι και σε γραφείς. Οποιοσδήποτε writer θέλει να εισέλθει πρέπει να έχει αποκλειστική πρόσβαση. Τα reader/writer locks είναι τύπου `rwlock_t`, ο οποίος ορίζεται στο <linux/spinlock.h>. Δηλώνονται και αρχικοποιούνται στατικά ή δυναμικά, όπως φαίνεται παρακάτω:

```
 rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* static */


rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* dynamic */
```

Οι συναρτήσεις είναι παρόμοιες με τις γενικές που παρουσιάστηκαν παραπάνω, το ίδιο και η λειτουργία τους γιαυτό και θα γίνει μόνο απλή αναφορά τους. Για τους αναγνώστες (readers) έχουμε:

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

και για τους γραφείς (writers):

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);

void wrtie_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

2.1.4 Χειρισμός Διακοπών (Interrupt Handling)

Οι περισσότερες πραγματικές συσκευές, όταν θέλουν να επικοινωνήσουν με τον επεξεργαστή το κάνουν μέσω του μηχανισμού διακοπών. Η διακοπή είναι ένα σήμα που στέλνει το υλικό για να λάβει την προσοχή του επεξεργαστή. Τις περισσότερες φορές, το μόνο που έχει να κάνει ένας οδηγός, είναι να εγγράψει ένα χειριστή στον πυρήνα για τις διακοπές της συσκευής που ελέγχει, και να αρχίσει να τις αντιμετωπίζει όταν αυτές αρχίσουν να φτάνουν. Ο μηχανισμός διακοπών παρέχει ένα τρόπο για να εκτρέπεται ο επεξεργαστής σε κώδικα εκτός της κανονικής ροής ελέγχου. Έτσι όταν φτάνει ένα σήμα διακοπής, ο επεξεργαστής διακόπτει την εκτέλεση της τρέχουσας διεργασίας και εκτελείται μία καινούρια ενέργεια. Εδώ πρέπει να σημειώσουμε, ότι ο κώδικας που εκτελείται από μία συνάρτηση χειρισμού διακοπής (Interrupt Service Routine - ISR) δεν αποτελεί διεργασία. Αντίθετα είναι ένα μονοπάτι ελέγχου μέσα στον πυρήνα, που τρέχει με κόστος της ίδιας διεργασίας την οποία διέκοψε.

Εγκατάσταση ενός Χειριστή Διακοπών (Interrupt Handler)

Όπως προαναφέραμε για να μπορούν να εξυπηρετηθούν διακοπές, αρχικά πρέπει να εγγραφεί ένας χειριστής στον πυρήνα. Σε διαφορετική περίπτωση, ο πυρήνας Linux αγνοεί οποιοδήποτε σήμα φτάσει σε κάποια γραμμή διακοπών. Οι γραμμές διακοπών από την πλευρά τους είναι ένας πολύτιμος και συχνά περιορισμένος πόρος του συστήματος, ειδικά σε περιπτώσεις που ο αριθμός τους δεν ξεπερνά τις 15 ή 16. Ο πυρήνας διατηρεί ένα μητρώο των γραμμών διακοπών και κάθε module αναμένεται να αιτηθεί για ένα κανάλι διακοπών πριν το χρησιμοποιήσει και τελικά να το αποδεσμεύσει όταν

ολοκληρώσει. Στη συνέχεια, θα δούμε πώς σε πολλές περιπτώσεις τα modules μοιράζονται γραμμές διακοπών με άλλους οδηγούς. Οι παρακάτω συναρτήσεις ορίζονται στο `<linux/interrupt.h>` και υλοποιούν τη διεπαφή εγγραφής μίας διακοπής:

```
int request_irq(unsigned int irq,
                   irqreturn_t (*handler)(int, void *, struct pt_regs *),
                   unsigned long flags,
                   const char *dev_name,
                   void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

Η τιμή που επιστρέφει η `request_irq` είναι 0 για επιτυχία και αρνητική για κωδικό λάθους. Είναι σύνηθες να έχουμε τιμή επιστροφής `-EBUSY`, κάτι που μας ενημερώνει πως κάποιος άλλος οδηγός έχει ήδη καταλάβει τη γραμμή διακοπών που αιτηθήκαμε.Οι μεταβλητές των συναρτήσεων είναι:

`unsigned int irq` Το νούμερο της διακοπής που αιτούμαστε

`irqreturn_t (*handler)(int, void *, struct pt_regs *)` Δείκτης στη συνάρτηση χειρισμού που εγκαθιστούμε.

`unsigned long flags` Μάσκα επιλογών που σχετίζεται με τη διαχείριση της διακοπής

`const char *dev_name` Το string που περνιέται στη `request_irq` και χρησιμοποιείται στο `/proc/interrupts` για να εμφανίσει τον κάτοχο της διακοπής

`void *dev_id` Δείκτης που χρησιμοποιείται για μοιραζόμενες γραμμές διακοπών. Είναι ένα μοναδικό αναγνωριστικό, που χρησιμοποιείται όταν ελευθερώνεται μια γραμμή διακοπής. Μπορεί ακόμη να χρησιμοποιηθεί από τον οδηγό, για να δείξει στον δικό του ιδιωτικό χώρο (για να αναγνωρίσει ποια συγκεκριμένη συσκευή διακόπτει). Αν η διακοπή δεν είναι μοιραζόμενη, το `dev_id` μπορεί να έχει την τιμή `NULL`.

Η μεταβλητή `flags` που είδαμε αμέσως πριν, μπορεί να πάρει τις παρακάτω τιμές:

`SA_INTERRUPT` Όταν οριστεί, καταδεικνύει γρήγορο χειριστή διακοπής. Οι γρήγοροι χειριστές εκτελούνται με απενεργοποιημένες τις διακοπές στον τρέχοντα επεξεργαστή

SA_SHIRQ Το bit αυτό σηματοδοτεί ότι η διακοπή μπορεί να είναι μοιραζόμενη μεταξύ πολλών συσκευών.

SA_SAMPLE_RANDOM Όταν οριστεί, καταδεικνύει ότι οι διακοπές που παράγονται συνεισφέρουν στην εντροπία του συστήματος. Αν η συσκευή παράγει διακοπές σε πραγματικά τυχαίο χρόνο (που δεν μπορούν να επηρεαστούν από εξωτερικούς παράγοντες), τότε ορίζεται αυτο το bit έτσι ώστε να συμβάλουν στην παραγωγή τυχαίων αριθμών του συστήματος (`/dev/random, /dev/urandom`). Να σημειώσουμε σε αυτό το σημείο πως τα παραπάνω `flags` συναντώνται στον πυρήνα 2.6.10. Στη δική μας υλοποίηση που γίνεται στον τρέχοντα πυρήνα (2.6.24) έχουν αλλάξει κάποια ονόματα αλλά οι λειτουργίες παραμένουν οι ίδιες.

Ένας χειριστής διακοπών εγκαθίσταται είτε κατά την αρχικοποίηση του οδηγού, είτε την πρώτη φορά που ανοίγεται η συσκευή. Επειδή ο αριθμός των γραμμών διακοπών είναι περιορισμένος, κρίνεται πιο σωστό η συνάρτηση `request_irq` να καλείται όταν η συσκευή ανοίγεται για πρώτη φορά. Πριν δηλαδή ανατεθεί στο υλικό η παραγωγή διακοπών. Αντίστοιχα, η `free_irq()` καλείται κατά το τελευταίο κλείσιμο της συσκευής, αμέσως μετά την ενημέρωση του υλικού ότι δεν έχει πλέον τη δυνατότητα να διακόπτει τον επεξεργαστή. Το μειονέκτημα αυτής της μεθόδου είναι η ανάγκη διατήρησης ενός μετρητή ανά άνοιγμα συσκευής, έτσι ώστε να γνωρίζουμε πότε μπορούν να απενεργοποιηθούν οι διακοπές. Να προσθέσουμε επίσης, ότι οι αρχιτεκτονικές i386 και x86_64, ορίζουν μια συνάρτηση για την εξέταση της διαθεσιμότητας μίας γραμμής διακοπής:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

Υλοποίηση χειριστή διακοπών

Η υλοποίηση ενός χειριστή διακοπών δεν είναι τίποτα άλλο από συνήθη κώδικα C με κάποιες όμως ιδιομορφίες. Οι ιδιομορφίες αυτές έχουν να κάνουν με το γεγονός ότι ο κώδικας χειριστή τρέχει σε χρόνο διακοπής (interrupt time), περιορίζοντας τις λειτουργίες που μπορεί να επιτελέσει. Έτσι ένας χειριστής διακοπής δεν μπορεί να μεταφέρει δεδομένα από και προς το χώρο χρήστη γιατί δεν εκτελείται στο πλαίσιο μιας διεργασίας. Επίσης δεν μπορεί να κάνει οτιδήποτε έχει σχέση με κατάσταση ύπνου (sleep), όπως: να καλέσει τη συνάρτηση wait_event, να δεσμεύσει μνήμη με οτιδήποτε διαφορετικό του `GFP_ATOMIC`, ή να κλειδώσει ένα σηματοφορέα. Τέλος οι χειριστές διακοπής

δεν μπορούν να καλέσουν τον δρομολογητή, δηλαδή τη συνάρτηση schedule.

Ένα ουσιώδες πρόβλημα που πρέπει να αντιμετωπιστεί κατά το χειρισμό διακοπών, είναι ο τρόπος που θα εκτελεστούν χρονοβόρες εργασίες μέσα στον ίδιο το χειριστή. Συχνά σε απάντηση μίας διακοπής, χρειάζεται να γίνει σημαντική ποσότητα εργασίας. Ταυτόχρονα οι χειριστές διακοπών πρέπει να ολοκληρώνουν γρήγορα και να μην εμποδίζουν τις διακοπές για μεγάλο χρονικό διάστημα. Έιναι φανερό πως οι δύο ανάγκες εργασίας και ταχύτητας έρχονται σε σύγκρουση η μία με την άλλη. Ο πυρήνας Linux (όπως και άλλα συστήματα) αντιμετωπίζουν το πρόβλημα χωρίζοντας τον χειριστή διακοπής σε δύο μέρη (halves). Το λεγόμενο ”άνω μέρος (top half)” είναι η ρουτίνα που απαντά ουσιαστικά στη διακοπή. Το ”κάτω μέρος (bottom half)” είναι η ρουτίνα που δρομολογείται από το άνω μέρος να εκτελεστεί σε μελλοντικό - ασφαλή χρόνο. Η μεγάλη διαφορά των δύο μερών έγκειται στο γεγονός ότι κατά τη διάρκεια εκτέλεσης του κάτω μέρους όλες οι διακοπές είναι ενεργοποιημένες, γιαυτό και εκτελείται σε ασφαλέστερο χρόνο. Σε ένα τυπικό σενάριο, το άνω μέρος αποθηκεύει τα δεδομένα της συσκευής σε ειδική για τη συσκευή μνήμη, δρομολογεί το κάτω μέρος και επιστρέφει. Αυτή η λειτουργία είναι ταχύτατη. Στη συνέχεια, αναλαμβάνει το κάτω μέρος να εκτελέσει οποιαδήποτε άλλη εργασία απαιτείται, όπως ξύπνημα διεργασιών, αρχή άλλης λειτουργίας Ε/Ε, κτλ. Αυτός ο διαχωρισμός επιτρέπει στο άνω μέρος να εξυπηρετήσει μία νέα διακοπή, τη στιγμή που το κάτω μέρος ασχολείται ακόμη με την προηγούμενη.

Μοιραζόμενες διακοπές (Shared IRQ)

Όπως προαναφέραμε στην αρχή της ενότητας, ο περιορισμένος αριθμός γραμμών διακοπής που επιβάλλουν οι διάφορες αρχιτεκτονικές, επιφέρει σοβαρούς περιορισμούς στην εύρυθμη λειτουργία του συστήματος, από τη στιγμή που οι συσκευές πολλαπλασιάζονται συνεχώς. Για το λόγο αυτό το μοντέρνο υλικό είναι σχεδιασμένο να επιτρέπει το μοίρασμα των διακοπών. Συνεπώς και ο πυρήνας Linux υποστηρίζει μοιραζόμενες διακοπές σε όλους τους διαδρόμους συστήματος, ακόμη και σε αυτούς (π.χ. ISA bus) που δεν υποστηρίζονταν παραδοσιακά. Οι μοιραζόμενες διακοπές εγκαθιστώνται μέσω της `request_irq`, όπως και οι μη μοιραζόμενες αλλά με δύο διαφορές:

- Το SA_SHIRQ πρέπει να προσδιοριστεί στη μεταβλητή flags κατά την αίτηση της γραμμής.

- Η μεταβλητή `dev_id` πρέπει να είναι μοναδική. Οποιοσδήποτε δείκτης στο χώρο διευθύνσεων του module είναι έγκυρος, αλλά το `dev_id` δεν μπορεί σε καμία περίπτωση να οριστεί `NULL`.

Ο πυρήνας διατηρεί μία λίστα με όλους τους μοιραζόμενους χειριστές που σχετίζονται με μία διακοπή, και το `dev_id` μπορεί να θεωρηθεί σαν μία υπογραφή που τους διαφοροποιεί μεταξύ τους. Όταν γίνει αίτηση για μία μοιραζόμενη διακοπή, η `request_irq` επιτυγχάνει στην περίπτωση που ισχύει ένα από τα παρακάτω:

- Η γραμμή διακοπής είναι ελεύθερη.
- Όλοι οι χειριστές που έχουν ήδη εγγραφεί για αυτή τη γραμμή, την έχουν δηλώσει ως μοιραζόμενη.

Όταν δύο ή περισσότεροι οδηγοί μοιράζονται μία γραμμή διακοπής και το υλικό διακόψει τον επεξεργαστή στη γραμμή αυτή, ο πυρήνας καλεί όλους τους χειριστές (που είναι εγγεγραμμένοι για αυτή τη διακοπή) και περνά στον καθένα το αντίστοιχο `dev_id`. Έτσι κάθε μοιραζόμενος χειριστής πρέπει να είναι σε θέση να αναγνωρίσει τις δικές του διακοπές και να επιστρέψει γρήγορα, στην περίπτωση που η διακοπή δεν προέρχεται από δική του συσκευή. Στην τελευταία περίπτωση, πρέπει ο χειριστής να επιστρέψει `IRQ_NONE`. Η αποδέσμευση του χειριστή γίνεται με το φυσιολογικό τρόπο χρησιμοποιώντας την `free_irq`. Εδώ η μεταβλητή `dev_id` χρειάζεται για να επιλεγεί ο σωστός χειριστής που θα απαλειφεί από τη λίστα με τους μοιραζόμενους χειριστές για τη συγκεκριμένη διακοπή. Αυτός είναι και ο λόγος που ο δείκτης `dev_id` πρέπει να είναι μοναδικός.

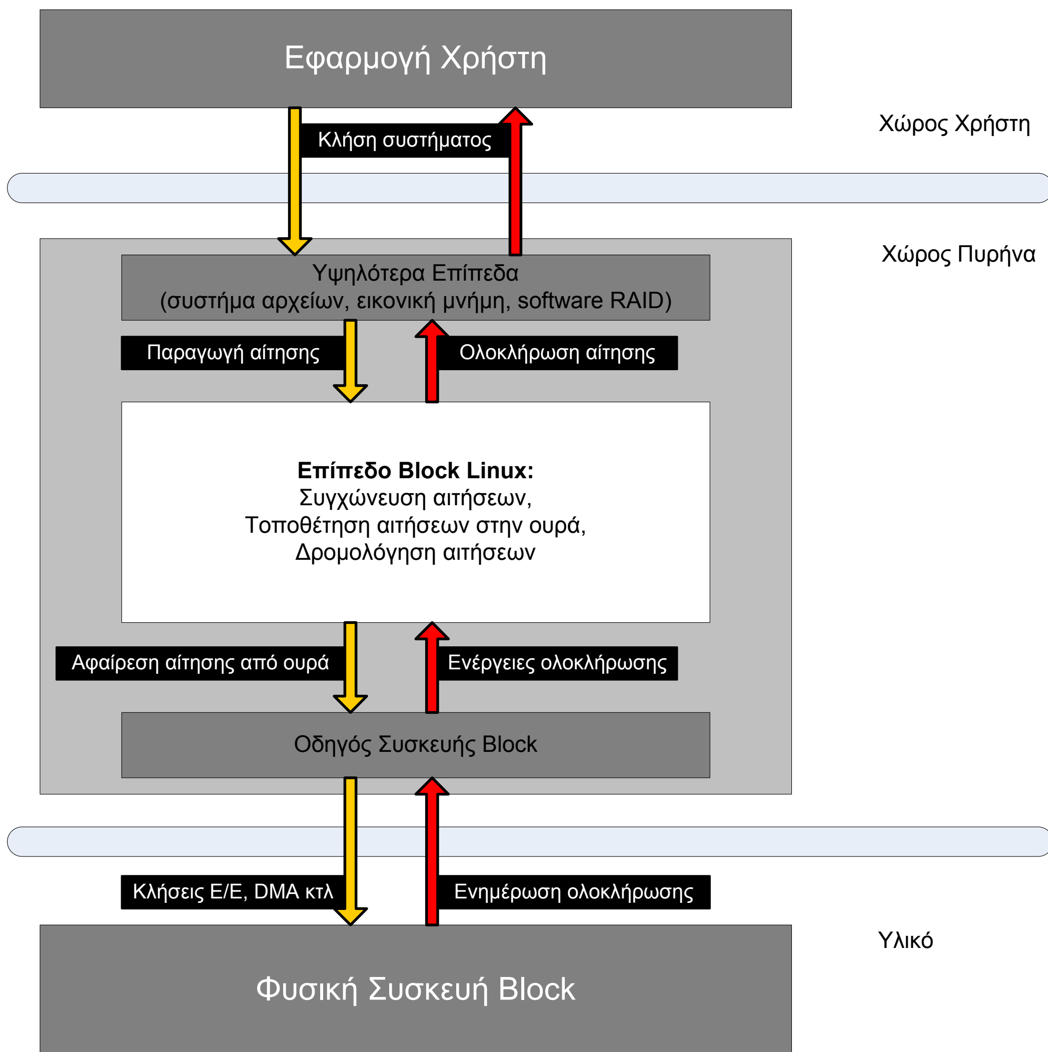
2.1.5 Οδηγοί συκευών block (Block Drivers)

Όπως είδαμε στην ενότητα 2.1.2, υπάρχουν block modules που υλοποιούν οδηγούς συσκευών block (block device drivers). Ο οδηγός block παρέχει πρόσβαση σε συσκευές που μεταφέρουν με τυχαίο ρυθμό δεδομένα καθορισμένου μεγέθους, κυρίως σε δίσκους. Ο πυρήνας Linux αντιμετωπίζει τις συσκευές block ως θεμελιώδες είδος συσκευών, διαφορετικό από τις υπόλοιπες συσκευές, κάτι που έχει σαν αποτέλεσμα οι οδηγοί block να έχουν μία ξεχωριστή διεπαφή και ιδιαίτερες προκλήσεις. Οι αποτελεσματικοί οδηγοί block αποτελούν κρίσιμο παράγοντα επίδοσης, κάτι που δεν περιορίζεται μόνο στις ρητές αναγνώσεις και εγγραφές των εφαρμογών χρήστη. Τα μοντέρνα

συστήματα με εικονική μνήμη δουλεύουν προωθώντας μη αναγκαία δεδομένα σε δευτερεύοντα μέσα αποθήκευσης, συνήθως δίσκους. Οι οδηγοί block αποτελούν τον αγωγό μεταξύ της κεντρικής μνήμης και των δευτερεύοντων μέσων αποθήκευσης, γι'αυτό και πολλές φορές μπορεί να θεωρηθούν ώς κομμάτι του υποσυστήματος της εικονικής μνήμης. Για το λόγο αυτό, οποιοσδήποτε έχει ανάγκη από έναν οδηγό block υψηλής επίδοσης πρέπει να είναι οικείος με τις βασικές έννοιες οργάνωσης και διαχείρισης μνήμης. Το μεγαλύτερο μέρος του σχεδιασμού του επιπέδου block επικεντρώνεται σε θέματα επίδοσης. Πολλές άλλες συσκευές π.χ. χαρακτήρων, μπορούν να μη τρέχουν στο μέγιστο των δυνατοτήτων τους και η συνολική επίδοση του συστήματος να μένει ανέπαφη. Το σύστημα δεν μπορεί όμως να τρέχει σωστά, αν το υποσύστημα των Ε/Ε (Εισόδων/Εξόδων) block δεν είναι καλά συντονισμένο. Έτσι η διεπαφή του Linux block driver επιτρέπει την αξιοποίηση στο έπακρο των δυνατοτήτων της συσκευής, υπαγορεύοντας βέβαια ταυτόχρονα, ένα μεγαλύτερο βαθμό πολυπλοκότητας όσον αφορά στη σχεδίαση και υλοποίηση.

Πρίν περάσουμε σε περισσότερες λεπτομέρειες σχετικά με το επίπεδο block και τους οδηγούς συσκευών block, κρίνεται σκόπιμο να διευκρινήσουμε κάποιους όρους και να ορίσουμε με ακρίβεια κάποιους άλλους, που θα συναντήσουμε στην πορεία της ανάλυσης. Έχουμε αναφερθεί κατ' επανάληψη στον όρο "block" χωρίς να τον μεταφράζουμε στα ελληνικά. Αυτό συμβαίνει γιατί ως block ορίζουμε ένα μεγάλο τεμάχιο δεδομένων καθορισμένου μεγέθους, με το μέγεθος να καθορίζεται από τον πυρήνα. Συνήθως τα blocks είναι 4096 bytes, αλλά αυτή η τιμή διαφέρει ανάλογα με την αρχιτεκτονική και το ακριβές σύστημα αρχείων που χρησιμοποιείται. Ένας τομέας (sector), σε αντίθεση, είναι ένα μικρό block, του οποίου το μέγεθος συνήθως καθορίζεται από το υποκαθήμενο υλικό. Ο πυρήνας αναμένει να αντιμετωπίσει συσκευές που υλοποιούν sectors μεγέθους 512 bytes. Αν η συσκευή χρησιμοποιεί διαφορετικό μέγεθος, ο πυρήνας προσαρμόζεται και αποφεύγει τη παραγωγή αιτήσεων Ε/Ε που δεν μπορεί να χειριστεί το υλικό. Επίσης, οποιαδήποτε υποβολή νούμερων σε sectors κάνει ο πυρήνας, εννοείται πως γίνεται στη βάση των 512 bytes και γι'αυτό στην περίπτωση που δουλεύουμαι με υλικό που έχει διαφορετικό μέγεθος sector υποχρεούμαστε να κάνουμε εμείς εσωτερικά τη μετατροπή στην κατάλληλη κλίμακα.

Στη συνέχεια παρουσιάζουμε τη διεπαφή του οδηγού block με τον πυρήνα, τις κύριες δομές που χρησιμοποιούνται και τις λειτουργίες που επιτελούνται μέσα σε αυτόν.



Σχήμα 2.4: *Block Layer*

Δήλωση

Οι οδηγοί block χρησιμποιούν ένα σύνολο διεπαφών δήλωσης, για να κάνουν διαθέσιμες τις συσκευές στον πυρήνα. Για το λόγο αυτό, υπάρχει μια ομάδα από δομές δεδομένων και λειτουργίες που δίνουν αυτή τη δυνατότητα στον οδηγό. Οι συσκευές block γίνονται προσβάσιμες μέσω ονομάτων στο σύστημα αρχείων. Αυτά τα ονόματα χαρακτηρίζονται ειδικά αρχεία (special files) ή αρχεία συσκευών και βρίσκονται συμβατικά κάτω από τον κατάλογο /dev. Τα ειδικά αρχεία που αναφέρονται σε οδηγούς block, προσδιορίζονται από ένα "b" στην πρώτη στήλη της εξόδου της εντολής `ls -l`. Στην ίδια έξοδο φαίνονται και δύο αριθμοί: ο μείζων και ο ελάσσων αριθμός (major, minor number). Οι δύο αυτοί αριθμοί χαρακτηρίζουν τη συσκευή. Παραδοσιακά, ο μείζων

αριθμός προσδιορίζει ποιός οδηγός αντιστοιχεί σε ποιά συσκευή. Οι μοντέρνοι πυρήνες Linux επιτρέπουν σε πολλαπλούς οδηγούς να μοιράζονται τον ίδιο μείζονα αριθμό, αλλά οι περισσότερες συσκευές ακόμη είναι οργανωμένες γύρω από την αρχή ενός μείζονος σε ένα οδηγό. Ο ελάσσων αριθμός χρησιμοποιείται από τον πυρήνα για τον ακριβή καθορισμό της συσκευής στην οποία θέλει να αναφερθεί. Αναλόγως με τον τρόπο που είναι γραμμένος ο οδηγός, μπορούμε να έχουμε ένα δείκτη από τον πυρήνα απευθείας στη συσκευή, ή μπορούμε να χρησιμοποιήσουμε τον ελάσσονα αριθμό σαν δείκτη σε ένα εσωτερικό πίνακα από συσκευές. Με τον ένα ή τον άλλο τρόπο, ο πυρήνας δεν γνωρίζει τίποτα για τους ελάσσονες αριθμούς, εκτός απο το γεγονός ότι αναφέρονται σε συσκευές που υλοποιούνται εσωτερικά στον οδηγό.

Εσωτερικά στον πυρήνα ο τύπος `dev_t` (που ορίζεται στο `<linux/types.h>`) διατηρεί τους μείζονες και ελάσσονες αριθμούς συσκευών. Σε επαφή με τους αριθμούς αυτούς δεν ερχόμαστε άμεσα, αλλά διαμέσου ενός συνόλου macros που βρίσκονται στο `<linux/kdev.h>`. Για να αποκτήσουμε των μείζονα ή ελάσσονα αριθμό από ένα `dev_t` χρησιμποιούμε:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Αντίθετα αν έχουμε και τους δύο αριθμούς και θέλουμε να τους μετατρέψουμε σε ένα `dev_t` χρησιμοποιούμε:

```
MKDEV(int major, int minor);
```

Έτσι λοιπόν το πρώτο πράγμα που κάνει η συντριπτική πλειοψηφία των οδηγών block είναι να δηλωθεί στον πυρήνα. Η συνάρτηση που δίνει αυτή τη δυνατότητα είναι η `register_blkdev`, η οποία ορίζεται στο `<linux/fs.h>` :

```
int register_blkdev(unsigned int major, const char *name);
```

Οι μεταβλητές της συνάρτησης είναι ο μείζων αριθμός που θα χρησιμοποιεί η συσκευή και το συναφές της όνομα (το οποίο θα εμφανίζει ο πυρήνας στο `/proc/devices`). Αν ο μείζων αριθμός οριστεί 0 (μηδέν), τότε ο πυρήνας δεσμεύει μόνος του έναν αριθμό και τον επιστρέφει σε αυτόν που κάλεσε τη συνάρτηση. Οπως πάντα, αρνητική τιμή επιστροφής από την `register_blkdev` υποδηλώνει την εμφάνιση λάθους κατά την εκτέλεση. Η αντίστοιχη συνάρτηση που ακυρώνει την εγγραφή ενός οδηγού block είναι η:

```
int unregister_blkdev(unsigned int major, const char *name);
```

Εδώ, οι μεταβλητές πρέπει να ταιριάζουν με αυτές που περάστηκαν στη `register_blkdev`, γιατί σε διαφορετική περίπτωση η συνάρτηση επιστρέφει `-EINVAL` και δεν απεγγράφει τίποτα.

Η δομή block device operations

Οι συσκευές block κάνουν διαθέσιμες στο σύστημα τις λειτουργίες τους μέσω της δομής `block_device_operations`. Η δομή αυτή ορίζεται στο αρχείο `<linux/fs.h>` και είναι μία συλλογή από δείκτες σε συναρτήσεις. Σε κάθε πεδίο της δομής ορίζεται μία λειτουργία, η οποία στις περισσότερες περιπτώσεις είναι υπεύθυνη για την υλοποιήση μίας κλήσης συστήματος. Ακολουθεί μία σύντομη επισκόπηση όλων των πεδίων της δομής ενώ αυτά που έχουν κύριο ενδιαφέρον για τη δική μας υλοποίηση θα επανεξεταστούν με λεπτομέρεια στη συνέχεια:

`int (*open)(struct inode *inode, struct file *filp)` Με τη μέθοδο open ο οδηγός κάνει οποιαδήποτε αρχικοποίηση χρειάζεται για να προετοιμαστεί η συσκευή για μετέπειτα χρήση. Στη συνάρτηση αυτή οι οδηγοί συνήθως ελέγχουν για τυχόν λάθη της συσκευής (όπως ανέτοιμη συσκευή, πρόβλημα στο υλικό), κάνουν τις κατάλληλες ενέργειες που χρειάζονται στην περίπτωση που η συσκευή ανοίγεται για πρώτη φορά, δεσμεύουν και γεμίζουν δομές δεδομένων.

`int (*release)(struct inode *inode, struct file *filp)` Ο ρόλος της μεθόδου release είναι ο αντίστροφος της open. Αποδεσμεύει οτιδήποτε δέσμευσε η open, ξεκλειδώνει τα μέσα (αποσπώμενα μέσα που έχει κλειδώσει η open) και τερματίζει με ασφάλεια τη συσκευή.

`int (*ioctl)(struct inode, struct file, unsinged int, unsigned long)` Η μέθοδος αυτή υλοποιεί την κλήση συστήματος ioctl. Η κλήση αυτή προσφέρει ένα τρόπο κατεύθυνσης εντολών ειδικών για τη συσκευή (όπως τη μορφοποίηση ενός track μιας δισκέττας floppy ή την ανάγνωριση των στοιχείων γεωμετρίας ενός δίσκου). Αν η συσκευή δεν παρέχει αυτή τη μέθοδο, τότε οι κλήσεις συστήματος επιστρέφουν λάθος για οποιαδήποτε μη προκαθορισμένη αίτηση.

int (*media_changed)(struct gendisk *gd) Η μέθοδος καλείται από τον πυρήνα για να ελέγξει αν κάποιος χρήστης έχει αλλάξει το μέσο από τη φυσική συσκευή. Σε αυτή την περίπτωση επιστρέφει μη μηδενική τιμή. Φυσικά αυτή η μέθοδος είναι εφαρμόσιμη μόνο σε συσκευές που υποστηρίζουν αποσπώμενα μέσα και ταυτόχρονα έχουν προβλέψει να παρέχουν ένα ”media changed” flag στον οδηγό (π.χ cdrom drive). Στις υπόλοιπες περιπτώσεις η μέθοδος αυτή μπορεί να παραλειφθεί.Η δομή gendisk που φαίνεται ως μεταβλητή θα αναλυθεί αμέσως παρακάτω.

int (*revalidate_disk)(struct gendisk *gd) Η μέθοδος αυτή καλείται σε απάντηση μιας αλλαγής μέσου. Έτσι δίνεται η δυνατότητα στον οδηγό να εκτελέσει τις κατάλληλες απαιτούμενες ενέργειες για να κάνει το νέο μέσο έτοιμο για χρήση, μόλις αντιλειφθεί πως αυτό έχει αλλάξει. Η συνάρτηση επιστρέφει μια ακέραια τιμή, η οποία όμως αγνοείται από τον πυρήνα.

struct module *owner Αυτός είναι ο δείκτης στο module που ανήκει αυτή η δομή. Συνήθως αρχικοποιείται με την τιμή `THIS_MODULE`.

Η δομή gendisk

Η δομή gendisk (ορίζεται στο `<linux/genhd.h>`) αποτελεί την αναπαράσταση του πυρήνα μίας μεμονωμένης συσκευής δίσκου. Στην πραγματικότητα, ο πυρήνας χρησιμοποιεί επίσης τη gendisk για να αναπαραστήσει διαμερίσματα (partitions), αλλά αυτή η λειτουργία είναι πέραν του θέματος της παρούσας εργασίας. Υπάρχουν αρκετά πεδία της δομής gendisk που πρέπει να αρχικοποιηθούν από τον οδηγό block:

int major, int first_major, int minors Τα πεδία αυτά περιγράφουν τον αριθμό ή τους αριθμούς συσκευής που χρησιμοποιεί ο δίσκος. Κατ’ ελάχιστο μια φυσική συσκευή πρέπει να χρησιμοποιεί το λιγότερο έναν ελλάσονα αριθμό. Αν βέβαια η φυσική συσκευή πρόκειται να είναι διαμερίσιμη, πρέπει επιπροσθέτως να κατανεμηθεί ένας ελλάσονας αριθμός για κάθε πιθανή διαμέριση. Μία κοινή τιμή για ελλάσονες είναι 16, οι οποίοι επιτρέπουν τη συσκευή πλήρους δίσκου και 15 διαμερίσματα. Κάποιοι οδηγοί δίσκων χρησιμοποιούν και 64 ελλάσονες αριθμούς για κάθε συσκευή.

char disk_name[32] Πεδίο στο οποίο πρέπει να τεθεί το όνομα του δίσκου της συσκευής. Αυτό εμφανίζεται στο /proc/partitions και sysfs.

struct block_device_operations *fops Δείκτης στη δομή block_device_operations που περιγράψαμε προηγουμένως. Με αυτό τον τρόπο γίνεται προσβάσιμο το σύνολο των συναρτήσεων που περιέχει η δομή αυτή.

struct request_queue *queue Δομή που χρησιμοποιεί ο πυρήνας για να διαχειριστεί αιτήσεις Ε/Ε για τη συγκεκριμένη συσκευή. Το πεδίο αυτό θα εξεταστεί αναλυτικά στην πορεία του κειμένου όπου θα αναλυθεί ο τρόπος αιτήσεων.

int flags Ένα ελάχιστα χρησιμοποιούμενο σύνολο σημαίων που περιγράφουν την κατάσταση της φυσικής συσκευής. Αν η συσκευή έχει αποσπώμενα μέσα ορίζεται GENHD_FL_REMOVABLE. Φυσικές συσκευές CD-ROM θέτουν GENHD_FL_CD. Σε περίπτωση που δεν θέλουμε να φαίνονται πληροφορίες για τη διαμέριση στο /proc/partitions θέτουμε GENHD_FL_SUPPRESS_PARTITION_INFO.

sector_t capacity Η χωρητικότητα της φυσικής συσκευής, σε sectors μεγέθους 512-bytes. Ο τύπος sector_t μπορεί να έχει εύρος 64 bits. Οι οδηγοί δεν πρέπει να θέτουν αυτό το πεδίο άμεσα, αλλά διαμέσου του set_capacity.

void *private_data Οι οδηγοί block χρησιμοποιούν αυτό το πεδίο για να δείξουν στα εσωτερικά τους δεδομένα.

Ο πυρήνας παρέχει ένα μικρό σύνολο από συναρτήσεις για να μεταχειριστεί κανείς τις δομές gendisk. Θα τις αναφέρουμε στο σημείο αυτό, αλλά η απόλυτη λειτουργικότητά τους θα φανεί κατά τη χρήση τους σε μετέπειτα κεφάλαιο που θα παρουσιαστεί η πραγματική υλοποίηση του οδηγού. Η δομή gendisk δεσμεύεται δυναμικά και απαιτεί ειδική διαχείριση από τον πυρήνα για να αρχικοποιηθεί. Οι οδηγοί δεν μπορούν να δεσμεύσουν τη δομή από μόνοι τους. Αντίθετα καλούν την:

```
struct gendisk *alloc_disk(int minors);
```

Η μεταβλητή minors πρέπει να είναι ο αριθμός των ελάσσονων αριθμών που χρησιμοποιεί ο συγκεκριμένος δίσκος. Όταν ένας δίσκος δεν είναι πλέον αναγκαίος πρέπει να απελευθερώνεται με την:

```
void del_gendisk(struct gendisk *gd);
```

Η del_gendisk πρέπει να καλείται μετά το τελευταίο release ή μέσα στη συνάρτηση εκκαθάρισης του module για να είναι βέβαιο ότι θα αφαιρέσει τη δομή. Σε διαφορετική περίπτωση υπάρχει πιθανότητα η δομή να μην αφαιρεθεί. Ο λόγοι έχουν να κάνουν με μετρητές αναφορών που υπάρχουν για τη δομή αυτή, η περιγραφή των οποίων είναι πέραν του σκοπού αυτής της εργασίας και γι' αυτό το λόγο δεν θα επεκταθούμε περισσότερο. Παρατηρούμε ότι η δέσμευση της δομής gendisk δεν συνεπάγεται ότι ο δίσκος γίνεται διαθέσιμος στο σύστημα. Για να γίνει αυτό πρέπει να ενεργοποιήσουμε τη δομή και να καλέσουμε την:

```
void add_disk(struct gendisk *gd);
```

Εδώ χρειάζεται σημαντική προσοχή, γιατί από τη στιγμή που καλούμε την add_disk ο δίσκος είναι πλέον παρών στο σύστημα και οι μέθοδοί του καλούνται ανά πάσα στιγμή. Στην πραγματικότητα, τέτοιες κλήσεις στις μεθόδους του θα γίνουν πιθανότατα πριν προλάβει να επιστρέψει η ίδια η add_disk. Ο πυρήνας θα προσπαθήσει να διαβάσει τα πρώτα blocks σε μια προσπάθεια να βρει τον πίνακα διαμερίσεων (partition table). Για το λόγο αυτό η add_disk καλείται αυστηρά αφού έχει ολοκληρωθεί κάθε είδους αρχικοποίηση και ενεργοποίηση και ο οδηγός είναι πλέον έτοιμος να ανταποκριθεί σε αιτήσεις που αφορούν στο συγκεκριμένο δίσκο. Μέχρι αυτή τη στιγμή θα έχει παρατηρήσει κανείς, ότι δεν έχουμε αναφερθεί σε καμία συνάρτηση που πραγματικά να διαβάζει ή να γράφει δεδομένα. Στο block υποσύστημα Ε/Ε τις λειτουργίες αυτές χειρίζεται η συνάρτηση request.

Επεξεργασία Αιτήσεων (Request Processing)

Ο πυρήνας κάθε οδηγού block είναι η συνάρτηση request. Με τη συνάρτηση αυτή γίνεται η πραγματική δουλειά του οδηγού, ή τουλάχιστον ξεκινά μέσα σε αυτή. Η επίδοση ενός οδηγού δίσκου μπορεί να αποτελεί ένα πολύ κρίσιμο κομμάτι της ολικής επίδοσης του συστήματος. Ως εκ τούτου, το block υποσύστημα του πυρήνα έχει γραφτεί σε μεγάλο βαθμό με γνώμονα την επίδοση και έτσι κάνει οτιδήποτε είναι δυνατό για να ενεργοποιήσει τους οδηγούς να αξιοποιήσουν στο έπακρο τα χαρακτηριστικά των συσκευών που ελέγχουν. Έτσι το υποσύστημα block εκθέτει μία πολύπλοκη διεπαφή με τον οδηγό συσκευής, που για κάποιους οδηγούς δεν είναι αναγκαία. Για το λόγο αυτό, υπάρχει η δυνατότητα προγραμματισμού τόσο μίας σχετικά απλής συνάρτησης request, όσο και μιας υψηλού επιπέδου συνάρτησης που θα εκτελείται σε πολύπλοκο

υλικό. Παρακάτω, θα γίνει μία περιγραφή συναρτήσεων `request` που θα αναφέρονται σε διαβαθμισμένα επίπεδα πολυπλοκότητας και θα καλύπτουν ένα ευρύ σύνολο απαιτήσεων.

Η μέθοδος request του οδηγού block έχει το παρακάτω πρωτότυπο:

```
void request(request_queue_t *queue);
```

Η συνάρτηση αυτή καλείται οποτεδήποτε θεωρεί ο πυρήνας ότι έχει έρθει η στιγμή που ο οδηγός μας πρέπει να επεξεργαστεί κάποια reads, writes ή άλλες λειτουργίες πάνω στη συσκευή. Η συνάρτηση `request` δεν χρειάζεται να ολοκληρώσει όλες τις αιτήσεις στην ουρά πριν επιστρέψει. Στην πραγματικότητα πιθανότατα δεν ολοκληρώνει καμία στις περισσότερες πραγματικές συσκευές. Πρέπει ωστόσο να ξεκινήσει με κάποιες από τις αιτήσεις και να διασφαλίσει ότι τελικά όλες θα ικανοποιηθούν από τον οδηγό. Κάθε συσκευή έχει μία ουρά αιτήσεων (request queue). Αυτό συμβαίνει γιατί οι πραγματικές μεταφορές από και προς το δίσκο λαμβάνουν χώρα σε πολύ διαφορετική χρονική στιγμή από αυτή που τις ζητά ο πυρήνας, και επίσης επειδή ο πυρήνας χρειάζεται την ευελιξία να δρομολογεί κάθε μεταφορά την πιο ευνοϊκή στιγμή (ομαδοποιώντας για παράδειγμα αιτήσεις που επηρεάζουν πολύ κοντινά sectors του δίσκου). Η ουρά αυτή συνδέεται με μία συνάρτηση `request` κατά τη δημιουργία της request queue. Συνήθως, αυτή η δημιουργία και σύνδεση γίνεται με έναν τρόπο που μοιάζει με τον παρακάτω:

```
dev->queue = blk_init_queue(our_request_function, &dev->lock);
```

Όπως φαίνεται παραπάνω, όταν δημιουργείται η ουρά της συσκευής (με την `blk_init_queue`) συνδέεται ταυτόχρονα και με τη συνάρτηση `request` (`our_request_function`). Επίσης, βλέπουμε ότι παρέχεται και ένα spinlock ώς μέρος της διαδικασίας δημιουργίας της ουράς. Όποτε καλείται η συνάρτηση request του οδηγού, κρατείται το lock από τον πυρήνα. Ως αποτέλεσμα, η συνάρτηση request τρέχει σε ατομικό πλαίσιο (atomic context), με ότι αυτό συνεπάγεται. Το κλείδωμα (lock) της ουράς αποτρέπει τον πυρήνα από το να προσθέσει και άλλες αιτήσεις στην ουρά για τη συσκευή, καθόλη τη διάρκεια που η συνάρτηση request κρατά το lock. Κάτω από ορισμένες συνθήκες, μπορούμε να αποδεσμεύσουμε το κλείδωμα ενώ η συνάρτηση request τρέχει. Στην περίπτωση βέβαια αυτή, πρέπει να είμαστε σίγουροι ότι δεν προσεγγίζουμε την ουρά αιτήσεων, ούτε καμία άλλη δομή που προστατευόταν προηγουμένως από το κλείδωμα. Φυσικά, πρέπει να ξαναποκτηθεί το κλείδωμα πριν την επιστροφή της συνάρτησης request. Τέλος, η κλήση της συνάρτησης request γίνεται ασύγχρονα και με σεβασμό

προς τις λειτουργίες των συνυπαρχόντων διεργασιών του χώρου χρήστη. Δέν μπορούμε να υποθέσουμε ότι ο πυρήνας τρέχει στο πλαίσιο της διεργασίας που εισήγαγε την τρέχουσα αίτηση. Δεν ξέρουμε αν ο buffer Ε/Ε που παρέχεται από την αίτηση βρίσκεται στο χώρο χρήστη ή στον χώρο πυρήνα. Έτσι οποιαδήποτε λειτουργία προσχωρεί ρητά στο χώρο χρήστη, αποτελεί λάθος και σίγουρα οδηγεί σε προβληματική κατάσταση. Όπως θα φανεί και στη συνέχεια, οτιδήποτε χρειάζεται να ξέρει ο οδηγός για την αίτηση βρίσκεται μέσα στις δομές τις οποίες λαμβάνει μέσω της ουράς αιτήσεων.

Μία απλή μέθοδος request (έστω `simple_request`) φαίνεται παρακάτω:

```
static void simple_request(request_queue_t *q)
{
	struct request *req;

	while ((req = elv_next_request(q)) != NULL) {
		struct myriblk_dev *dev = req->rq_disk->private_data;
		if (! blk_fs_request(req)) {
			printk (KERN_NOTICE "Skip non-fs request\n");
			end_request(req, 0);
			continue;
		}
		myriblk_transfer(dev, req->sector, req->current_nr_sectors,
				req->buffer, rq_data_dir(req));
		end_request(req, 1);
	}
}
```

Το πρώτο πράγμα που βλέπουμε στη συνάρτηση αυτή, είναι μία δομή `request`. Η δομή αυτή θα εξεταστεί προσεχώς, αλλά στο συγκεκριμένο σημείο θα αρκεστούμε στο γεγονός ότι παριστάνει μία αίτηση Ε/Ε block την οποία πρέπει να εκτελέσει ο οδηγός μας. Ο πυρήνας παρέχει την συνάρτηση `elv_next_request` για να αποκτήσουμε την πρώτη εκκρεμή αίτηση της ουράς (στην περίπτωση που δεν υπάρχουν αιτήσεις στην ουρά η συνάρτηση επιστρέφει `NULL`). Να σημειώσουμε ότι η συνάρτηση `elv_next_request` δεν αφαιρεί την αίτηση από την ουρά. Σε αυτή την απλή υλοποίηση οι αιτήσεις δεν αφαιρούνται από την ουρά παρά μόνο όταν ολοκληρωθούν.

Μία ουρά αιτήσεων block μπορεί να περιέχει αιτήσεις που να μην έχουν ώς σκοπό την μεταφορά blocks από και προς τη συσκευή. Τέτοιες αιτήσεις περιέχουν χαμηλού επιπέδου, ειδικές για τον κατασκευαστή διαγνωστικές λειτουργίες ή οδηγίες σχετικές με εξειδικευμένες επιλογές της συσκευής (όπως η επιλογη packet writing mode για εγγράψιμα μέσα). Οι περισσότεροι οδηγοί δεν γνωρίζουν πως να αντιμετωπίσουν τις αιτήσεις αυτές και εκπίπτουν. Στο συγκεκριμένο παράδειγμα ο οδηγός τις αντιμετωπίζει. Η κλήση στη `blk_fs_request` μας ενημερώνει κατά πόσον η αίτηση είναι αίτηση συστήματος αρχείων (filesystem request) και έχει σκοπό τη μεταφορά blocks δεδομένων. Άν η αίτηση δεν είναι τέτοιου τύπου την περνάμε στην `end_request`:

```
void end_request(struct request *req, int succeeded);
```

Στην περίπτωση που απορρίψουμε μια αίτηση που δεν είναι filesystem request περνάμε στη μεταβλητή `succeeded` το 0 για να υποδηλώσουμε ότι η αίτηση δεν ολοκληρώθηκε με επιτυχία. Διαφορετικά καλούμε την `myriblk_transfer` για να κάνει την πραγματική μετακίνηση δεδομένων.Η συνάρτηση αυτή παίρνει ώς μεταβλητές δεδομένα που παρέχονται από τη δομή request (`struct request`):

`sector_t sector` : Ο δείκτης στο αρχικό sector της συσκευής μας από/προς το οποίο θα ξεκινήσει η μετακίνηση δεδομένων. Να θυμίσουμε ότι αυτός ο αριθμός, όπως και κάθε άλλος που περνιέται μεταξύ πυρήνα και οδηγού, είναι εκφρασμένος σε sectors μεγέθους 512-bytes. Αν το υλικό χρησιμοποιεί διαφορετικό μέγεθος sector, επιβαρυνόμαστε εμείς με την ανάλογη μετατροπή στη σωστή κλίμακα.

`usigned long nr_sectors` : Ο αριθμός των sectors που θα μεταφερθούν (μεγέθους 512-bytes).

`char *buffer` : Δείκτης στον buffer από/στον οποίο θα γίνει η μεταφορά των δεδομένων. Αυτός ο δείκτης είναι μία εικονική μνήμη πυρήνα και μπορεί να αποκωδικοποιηθεί κατευθείαν από τον οδηγό αν χρειαστεί.

`rq_data_dir(struct request *req)` : Αυτό το macro εξάγει από την αίτηση την κατεύθυνση που θα έχει η μεταφορά των δεδομένων. Η επιστροφή τιμής 0 (μηδεν) δηλώνει διάβασμα (`read`) από τη συσκευή, ενώ μη μηδενική τιμή δηλώνει γράψιμο στη συσκευή (`write`).

Με τις πληροφορίες αυτές ο οδηγός υλοποιεί την πραγματική μεταφορά δεδομένων μέσα στη συνάρτηση transfer. Ο τρόπος, έχει να κάνει με το υλικό που θέλουμε να χειριστούμε και τις συγκεκριμένες λειτουργίες που θέλουμε να εφαρμόσουμε. Παρόλα αυτά, ο οδηγός αυτός δεν είναι ρεαλιστικός για πολλών ειδών συσκευές. Ο πρώτος λόγος που συμβαίνει αυτό είναι γιατί ένας τέτοιος οδηγός εκτελεί τις αιτήσεις σύγχρονα, μία κάθε φορά. Συσκευές δίσκων υψηλής επίδοσης είναι ικανές να έχουν μεγάλο αριθμό αιτήσεων οι οποίες εκκρεμούν την ίδια χρονική στιγμή και ο ελεγκτής δίσκου διαλέγει να τις εκτελέσει με τη βέλτιστη σειρά. Από τη στιγμή που εμείς ασχολούμαστε μόνο με την αίτηση που βρίσκεται πρώτη στην ουρά, δεν μπορούμε ποτέ να έχουμε πολλαπλές αιτήσεις που να εκπληρώνονται σε μια δεδομένη στιγμή. Ένας δεύτερος λόγος έχει να

κάνει με το γεγονός ότι η καλύτερη επίδοση πετυχαίνεται όταν το σύστημα εκτελεί μεγάλες μεταφορές στις οποίες εμπλέκονται πολλαπλά sectors που γειτνιάζουν πάνω στο δίσκο. Το μεγαλύτερο κόστος στις λειτουργίες δίσκων εξαρτάται από την τοποθέτηση της κεφαλής στα κατάλληλα σημεία για `read` και `write`. Από τη στιγμή που θα γίνει αυτό, ο χρόνος που χρειάζεται από κει και πέρα για να γίνει η ανάγνωση ή η εγγραφή είναι ασήμαντος. Γι' αυτό και όσοι ασχολούνται με υλοποίηση συστημάτων αρχείων και υποσυστημάτων εικονικής μνήμης, προσπαθούν όσο γίνεται σε μεγαλύτερο βαθμό, να τοποθετούν δεδομένα που σχετίζονται μεταξύ τους σε συνεχόμενες θέσεις στο δίσκο και να μεταφέρουν όσο το δυνατόν περισσότερα sectors σε κάθε αίτηση. Συμπερασματικά για να μπορέσει ένας οδηγός να εκμεταλλευτεί όλα αυτά που μόλις αναφέραμε χρειάζεται μία πολύ βαθύτερη κατανόηση των ουρών αιτήσεων, των δομών `request` και των δομών `bio`, πάνω στις οποίες φτιάχνονται οι αιτήσεις. Με αυτά θα ασχοληθούμε αμέσως παρακάτω.

Ουρές αιτήσεων

Στην απλή μορφή τους, που συναντήσαμε και παραπάνω, οι ουρές αιτήσεων block είναι απλά μία σειρά από αιτήσεις Ε/Ε στο επίπεδο block. Αν δούμε όμως βαθύτερα, κάτι που δεν χρειάζεται βέβαια να γνωρίζουν οι οδηγοί, τα πράγματα αποδεικνύονται αρκετά πιο πολύπλοκα. Οι ουρές αιτήσεων παρακολουθούν και συντονίζουν ένα τεράστιο αριθμό από αιτήσεις Ε/Ε block, αλλά ταυτόχρονα παίζουν και κρίσιμο ρόλο στην δημιουργία αυτών των αιτήσεων. Η ουρά αιτήσεων αποθηκεύει παραμέτρους που περιγράφουν το είδος των αιτήσεων που είναι ικανή να εξυπηρετήσει η συσκευή. Μερικές από αυτές τις παραμέτρους είναι: το μέγιστο μέγεθος των αιτήσεων, ο αριθμός των διαφορετικών τμημάτων (segments) που περιέχει μία αίτηση, το μέγεθος sector του υλικού, απαιτήσεις ευθυγράμμισης (alignment). Αν η ουρά αιτήσεων είναι ρυθμισμένη κατάλληλα, δεν υποβάλλει ποτέ μία αίτηση την οποία δεν μπορεί να χειριστεί η συσκευή. Επίσης οι ουρές αιτήσεων υλοποιούν μία ενσωματώσιμη διεπαφή που επιτρέπει τη χρήση πολλαπλών δρομολογητών (schedulers) ή ανυψωτών (elevators) Ε/Ε. Η δουλειά ενός δρομολογητή Ε/Ε είναι να υποβάλλει αιτήσεις Ε/Ε στον οδηγό με τρόπο που να μεγιστοποιεί την απόδοση. Έτσι οι περισσότεροι δρομολογητές Ε/Ε συσσωρεύουν μία παρτίδα αιτήσεων, τις ταξινομούν έτσι ώστε οι δείκτες σε block να έχουν αύξουσα (ή φθίνουσα) σειρά, και τις υποβάλλουν με αυτή τη σειρά στον οδηγό. Αυτό έχει ως αποτέ-

λεσμα η κεφαλή των δίσκων να κάνει λιγότερες κινήσεις συνολικά για να ικανοποιήσει μία παρτίδα αιτήσεων. Τη στιγμή της συγγραφής αυτής της εργασίας, ο προεπιλεγμένος δρομολογητής του Linux είναι ο λεγόμενος προνοητικός (anticipatory scheduler), ο οποίος φαίνεται να παράγει την καλύτερη επίδοση σε ένα διαδραστικό σύστημα. Ο δρομολογητής αυτός παύει τη συσκευή βραχέως μετά από μία αίτηση read γιατί περιμένει (προνοεί) ότι και άλλο ένα παρακείμενο read θα φτάσει άμεσα. Μία άλλη λειτουργία που επιτελούν οι δρομολογητές Ε/Ε είναι η συγχώνευση παρακείμενων αιτήσεων. Όταν μία νέα αίτηση παραδίδεται στο δρομολογητή, αυτός ψάχνει την ουρά για αιτήσεις που αφορούν γειτονικά sectors. Αν βρεθεί τέτοια αίτηση και η τελική αίτηση δεν θα υπερβεί καθορισμένα μεγέθη τότε έχουμε συγχώνευση των δύο αιτήσεων. Οι ουρές αιτήσεων είναι τύπου `struct request_queue` ή `request_queue_t`. Ο τύπος αυτός ορίζεται στο `<linux/blkdev.h>` και το μεγαλύτερο μέρος της υλοποίησής τους γίνεται στα αρχεία `drivers/block/ll_rw_block.c` και `elevator.c`. Όπως είδαμε και στον κώδικα του παραδείγματος, η ουρά αιτήσεων είναι μία δυναμική δομή δεδομένων που πρέπει να δημιουργηθεί από το υποσύστημα Ε/Ε block. Η συνάρτηση που δημιουργεί και εκκινεί την ουρά είναι:

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock *lock);
```

Οι μεταβλητές είναι: η συνάρτηση request της ουράς και το spinlock που ελέγχει την πρόσβαση στην ουρά. Η συνάρτηση αυτή δεσμεύει ένα μεγάλο μέρος μνήμης, γι'αυτό και είναι πολύ πιθανό να εκπέσει. Ο έλεγχος της τιμής επιστροφής κρίνεται απαραίτητος πριν από οποιαδήποτε απόπειρα χρήσης της ουράς. Αντίστοιχα, για να επιστρέψουμε μία ουρά αιτήσεων στο σύστημα (συνήθως κατά την αποφόρτωση του module) χρησιμοποιούμε την:

```
void blk_cleanup_queue(request_queue_t *queue);
```

Μετά από αυτή την κλήση ο οδηγός δεν βλέπει περαιτέρω αιτήσεις από τη συγκεκριμένη ουρά και δεν πρέπει και ο ίδιος να αναφερθεί ξανά σε αυτήν. Εκτός από τις συναρτήσεις δημιουργίας και εκκαθάρισης, υπάρχει και ένα μικρό σύνολο από συναρτήσεις χειρισμού των αιτήσεων στις ουρές. Για να χρησιμοποιηθούν αυτές οι συναρτήσεις πρέπει πρώτα να δεσμεύσουμε το κλείδωμα (lock) της ουράς πριν της καλέσουμε:

```
struct request *elv_next_request(request_queue_t *queue);
```

Η συνάρτηση αυτή επιστρέφει την επόμενη αίτηση προς επεξεργασία. Όπως είδαμε και

στο παράδειγμα, επιστρέφει ένα δείκτη στην αίτηση που βρίσκεται πρώτη στην ουρά (όπως αυτή καθορίζεται από τον δρομολογητή Ε/Ε) ή `NULL` στην περίπτωση που δεν απομένουν άλλες αιτήσεις προς επεξεργασία. Η `elv_next_request` αφήνει την αίτηση στην ουρά, αλλά τη σημειώνει ώς ενεργή, αποτρέποντας το δρομολογητή Ε/Ε να την συγχωνεύσει με άλλες αιτήσεις. Για να αφαιρεθεί τελείως μια αίτηση από την ουρά χρησιμοποιούμε την:

```
void blkdev_dequeue_request(struct request *req);
```

Αν έχουμε ένα οδηγό που χειρίζεται πολλαπλές αιτήσεις της ίδιας ουράς ταυτόχρονα, πρέπει να τις αφαιρεί μόνο με αυτόν τον τρόπο. Σε περίπτωση που χρειαστεί να επανατοποθετηθεί στην ουρά μία αίτηση που έχει προηγουμένως αφαιρεθεί, τότε γίνεται κλήση στην:

```
void elv_requeue_request(request_queue_t *queue, struct request* req);
```

Εκτός από τις συναρτήσεις που χειρίζονται τις αιτήσεις των ουρών, το επίπεδο block εξάγει και ένα σύνολο συναρτήσεων που μπορεί να χρησιμοποιήσει ο οδηγός για να χειριστεί τη συμπεριφορά των ίδιων των ουρών αιτήσεων. Οι σημαντικότερες αυτών είναι:

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

Στην περίπτωση που η συσκευή φτάσει σε σημείο που δεν μπορεί να χειριστεί πλέον άλλες αιτήσεις που εκκρεμούν, μπορεί να καλέσει την `blk_stop_queue` και να ενημερώσει το επίπεδο block. Μετά από αυτή την κλήση η συνάρτηση request δεν θα ξανακαλεστεί, παρά μόνον όταν προηγηθεί κλήση στην`blk_start_queue`. Οι παραπάνω ενέργειες γίνονται πάντα με το κλείδωμα κρατημένο.

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

Η συνάρτηση αυτή ενημερώνει τον πυρήνα για το ποιά είναι η υψηλότερη φυσική διεύθυνση στην οποία μπορεί να εκτελέσει DMA η συσκευή. Αν εμφανιστεί αίτηση που αναφέρεται σε διεύθυνση μνήμης άνω του δοθέντος ορίου, ένας buffer αναπήδησης θα χρησιμοποιηθεί για την ενέργεια. Αυτός είναι ένας τρόπος λειτουργίας με μεγάλο κόστος επίδοσης και τις περισσότερες φορές αποφεύγεται.

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short
```

```
max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short
max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int
max);
```

Οι συναρτήσεις αυτές ορίζουν παράμετρους που περιγράφουν το είδος αιτήσεων που μπορεί να ικανοποιήσει η συσκευή. Η `blk_queue_max_sectors` ορίζει το μέγιστο μέγεθος οποιασδήποτε αίτησης (σε 512-bytes sectors). Το προεπιλεγμένο είναι 255. Η `blk_queue_max_phys_segments` και η `blk_queue_max_hw_segments` ελέγχουν πόσα φυσικά τμήματα (μη γειτονικές περιοχές στη μνήμη συστήματος) μπορούν να περιέχονται σε κάθε ξεχωριστή αίτηση. Η πρώτη αναφέρεται στον αριθμό των διαφορετικών τμημάτων που μπορεί να χειριστεί ο οδηγός, ενώ η δεύτερη στον αριθμό που μπορεί να χειριστεί η ίδια η συσκευή. Η προεπιλογή και για τις δύο είναι 128. Η `blk_queue_max_segment_size` αναφέρει στον πυρήνα πόσο μεγάλο μπορεί να είναι κάθε μεμονωμένο τμήμα. Η τιμή εκφράζεται σε bytes και προεπιλέγεται ίση με 65.536 bytes.

```
blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
```

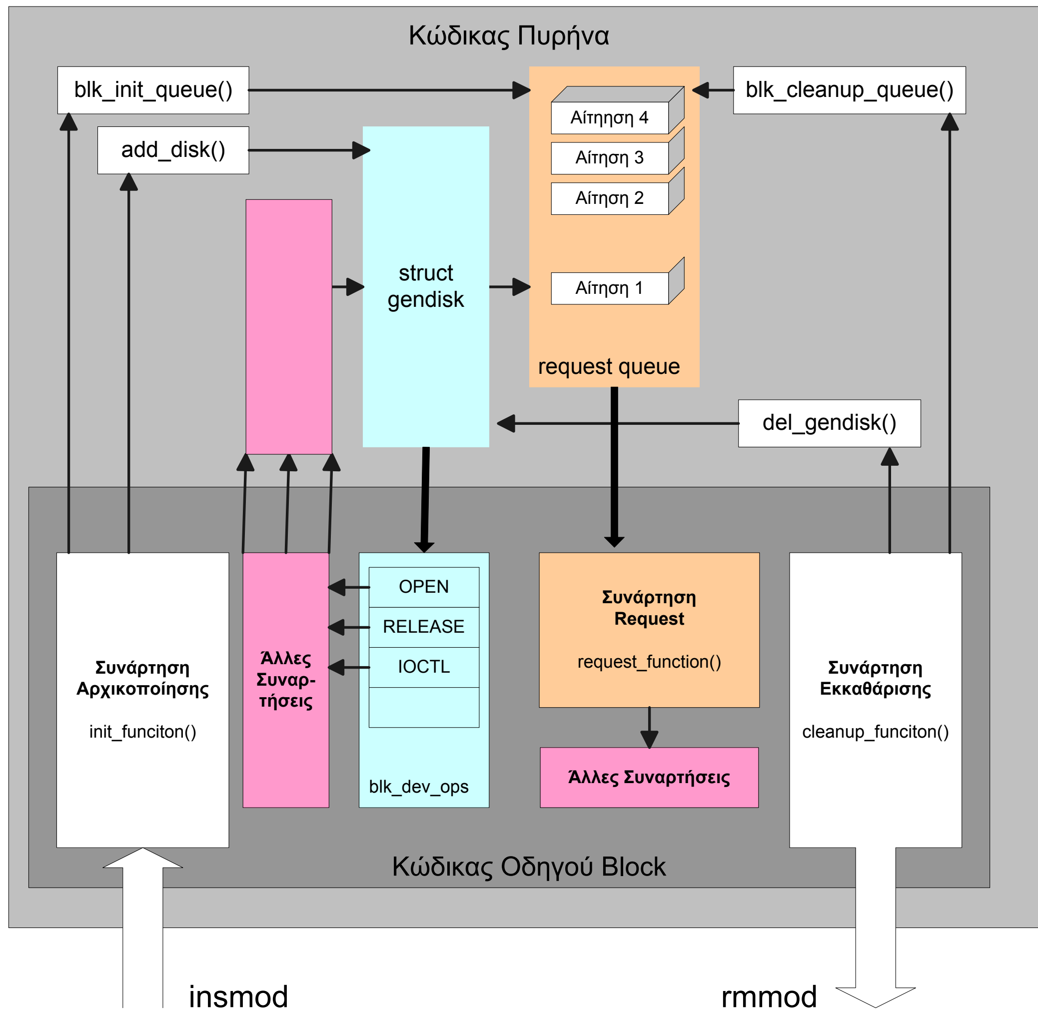
Υπάρχουν συσκευές που δεν μπορούν να χειριστούν αιτήσεις που ξεπερνούν ένα συγκεκριμένο όριο μεγέθους μνήμης και με αυτόν τον τρόπο ενημερώνουν τον πυρήνα για το όριο αυτό. Έτσι μία συσκευή που έχει πρόβλημα με αιτήσεις που ξεπερνούν το όριο των 4MB δίνει μάσκα `0x3fffff`. Η προεπιλογή είναι `0xffffffff`.

```
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
```

Η συνάρτηση αυτή ενημερώνει τον πυρήνα για τους περιορισμούς στην ευθυγράμμιση μνήμης που υπαγορεύει η συσκευή κατά τις μεταφορές DMA. Όλες οι αιτήσεις δημιουργούνται βάση αυτών των ευθυγραμμίσεων και το τελικό μήκος της αίτησης επίσης ταιριάζει με αυτές. Η προεπιλεγμένη μάσκα είναι η `0x1ff`, η οποία παράγει αιτήσεις ευθυγραμμισμένες σε όρια 512 bytes.

```
void blk_queue_hardsect_size(reqeust_queue_t *queue, unsigned short
max);
```

Ενημερώνει τον πυρήνα για το μέγεθος sector του υλικού. Όλες οι αιτήσεις που παράγει ο πυρήνας έχουν μέγεθος πολλαπλάσιο αυτού του μεγέθους και είναι κατάλληλα ευθυγραμμισμένες. Παρόλα αυτά, όλη η επικοινωνία μεταξύ πυρήνα και επιπέδου block



Σχήμα 2.5: *Διεπαφή του Block Layer*

συνεχίζει να εκφράζεται σε sectors μεγέθους 512-bytes.

Η δομή `request`

Στο παράδειγμα που είδαμε, συναντήσαμε τη δομή request αλλά δεν φάνηκε η πολύπλοκη δομή της. Κάθε δομή `request` αντιπροσωπεύει μία αίτηση Ε/Ε στο επίπεδο block, ακόμη και αν έχει προκύψει από τη συνένωση αρκετών ανεξάρτητων αιτήσεων σε υψηλότερο επίπεδο. Τα sectors που πρόκειται να μεταφερθούν σε καθεμία αίτηση, μπορεί να είναι κατανεμημένα καθ' όλη την έκταση της κεντρικής μνήμης, αλλά πάντα αντιστοιχούν σε ένα σύνολο συναπτών και συνεχόμενων sectors πάνω στην συσκευή block. Η αίτηση παρουσιάζεται σαν σύνολο κάποιων τμημάτων (segments), καθένα από τα οποία αντιστοιχεί σε ένα μοναδικό χώρο μέσα στη μνήμη (in-memory buffer). Ο πυ-

ρήνας μπορεί να ενώνει πολλαπλές αιτήσεις που αφορούν διαδοχικά sectors στο δίσκο, αλλά ποτέ δεν συνδυάζει λειτουργίες read και write μέσα στην ίδια δομή request. Όπως σημειώσαμε και νωρίτερα, ο πυρήνας δεν συνδυάζει επίσης αιτήσεις που θα καταλήξουν να παραβιάζουν οποιοδήποτε όριο της ουράς αιτήσεων. Η δομή request υλοποιείται κατ' ουσίαν ως συνδεδεμένη λίστα δομών bio, συνδυασμένη με πληροφορίες που βοηθούν τον οδηγό να την χειριστεί.

Όταν ο πυρήνας αποφασίσει πως ένα σύνολο blocks πρέπει να μεταφερθεί από ή προς μία συσκευή Ε/Ε block, συναρμολογεί μία δομή `bio`, για να περιγράψει αυτή τη λειτουργία. Στη συνέχεια παραδίδει τη δομή αυτή στον κώδικα που είναι υπεύθυνος για την Ε/Ε block, ο οποίος την συνενώνει σε μία προϋπάρχουσα δομή `request` ή αν χρειαστεί δημιουργεί μία καινούρια. Η δομή `bio` περιέχει όλη την πληροφορία που χρειάζεται ο οδηγός block για να διενεργήσει την αίτηση, αδιαφορώντας για οποιαδήποτε αναφορά σε διεργασία του χώρου χρήστη που μπορεί να την προκάλεσε. Η δομή `bio` (που ορίζεται στο `<linux/bio.h>`) περιέχει έναν αριθμό από πεδία που χρησιμοποιούνται από τους οδηγούς:

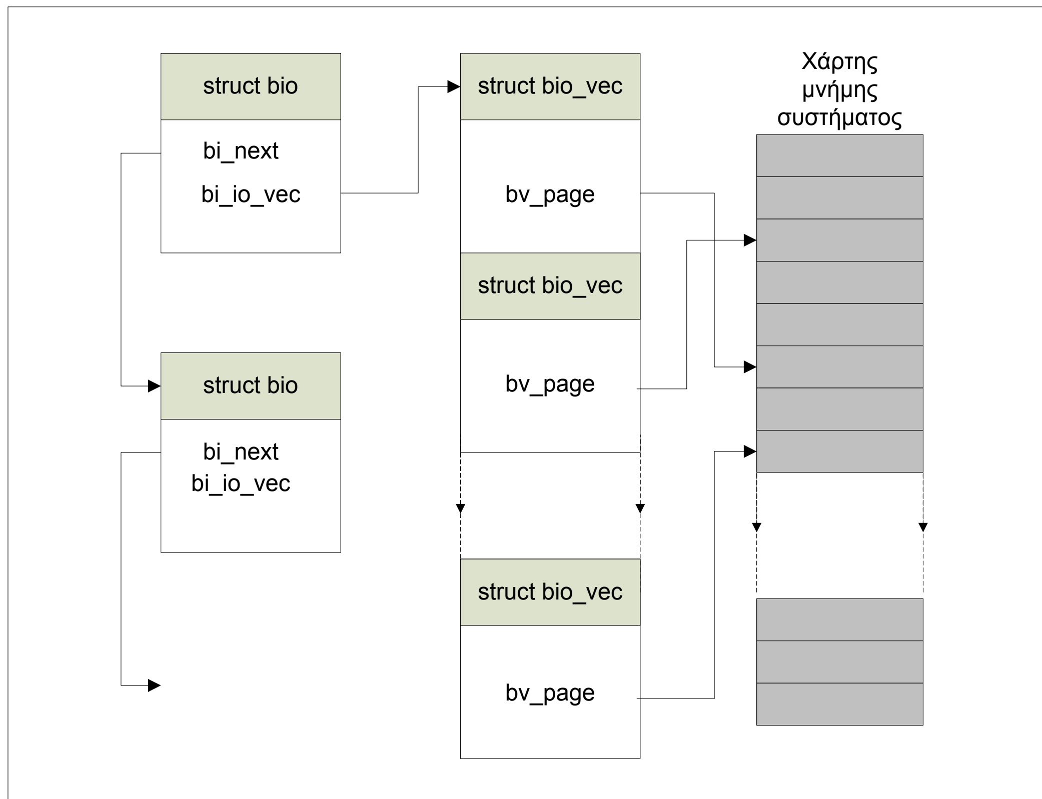
```
sector_t bi_sector;
unsigned int bi_size;
unsigned long bi_flags;
unsigned short bio_phys_segments;
unsigned short bio_hw_segments;
```

Ο πυρήνας όμως του κάθε `bio` είναι ένας πίνακας που ονομάζεται `bi_io_vec` και έχει ώς στοιχεία του τις παρακάτω δομές:

```
struct bio_vec {
        struct page         *bv_page;
        unsigned int        bv_len;
        unsigned int        bv_offset;
};
```

Όπως φαίνεται στο Σχ. 2.6, από τη στιγμή που μία αίτηση Ε/Ε block μετατραπεί σε μία δομή `bio`, έχει ήδη αποσυντεθεί σε μεμονωμένες σελίδες (pages) φυσικής μνήμης. Το μόνο που χρειάζεται να κάνει ο οδηγός από εκείνο το σημείο και μετά είναι διασχίσει κάθε δομή του πίνακα (υπάρχουν `bi_vcnt` δομές) και να μεταφέρει τα δεδομένα μέσα στην κάθε σελίδα.

Αναφέρουμε στη συνέχεια ένα σύνολο δεικτών που διατηρεί το επίπεδο block από τη

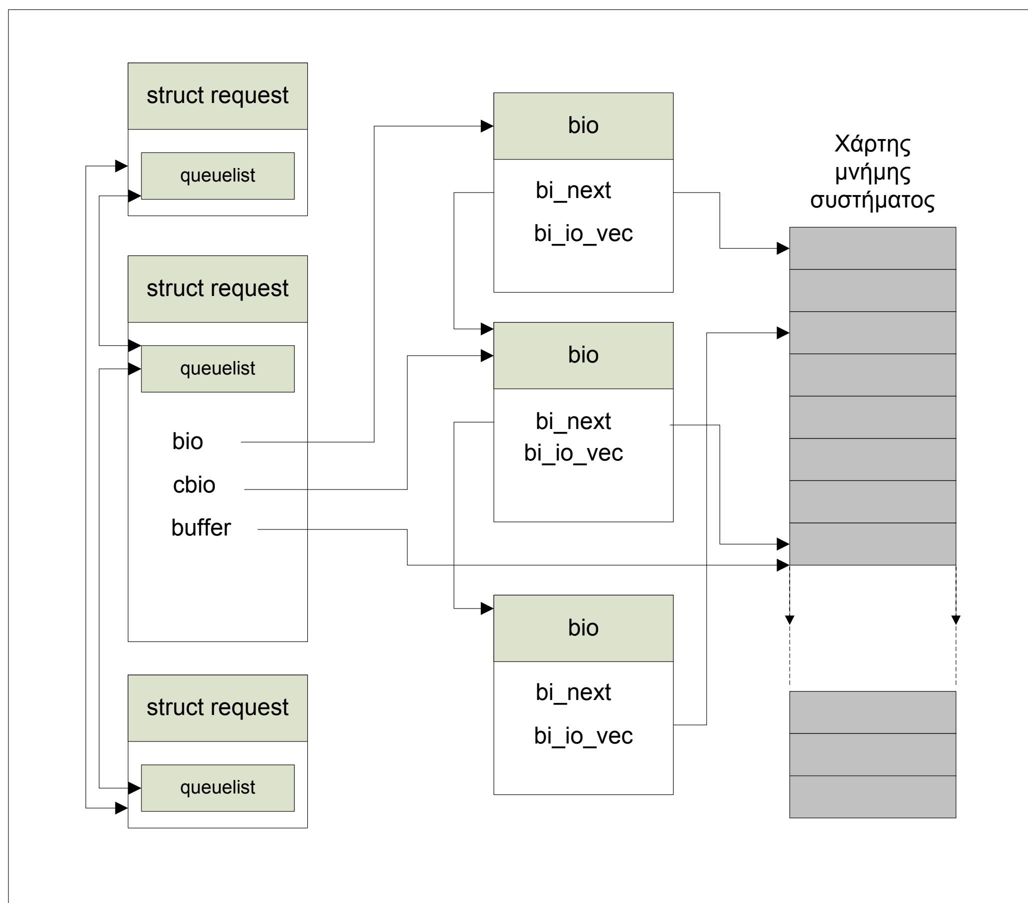


Σχήμα 2.6: *Δομή bio*

μεριά του και δείχνουν μέσα στη δομή bio. Οι δείκτες αυτοί χρησιμεύουν κυρίως για να μπορεί το επίπεδο block να παρακολουθεί την τρέχουσα κατάσταση της επεξεργασίας αιτήσεων και πρακτικά δεν χρησιμοποιούνται σχεδόν ποτέ από τους οδηγούς. Αυτός είναι και ο λόγος που δεν κρίνεται απαραίτητη η εις βάθος ανάλυσή αυτών, αλλά μία σύντομη αναφορά τους:

```
struct page *bio_page(struct bio *bio);
int bio_offset(struct bio *bio);
int bio_cur_sectors(struct bio *bio);
char *bio_data(struct bio *bio);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

Μετά την παρουσίαση της δομής bio και γνωρίζοντας πως αυτή λειτουργεί, μπορούμε να εμβαθύνουμε και στη δομή request και να δούμε πώς δουλεύει στην πραγματικότητα η επεξεργασία αιτήσεων. Θα ξεκινήσουμε με τα κυριότερα πεδία της δομής.



Σχήμα 2.7: *Δομή request*

`sector_t hard_sector, unsigned long hard_nr_sectors, unsigned int hard_cur_sectors` : Τα πεδία αυτά παρακολουθούν τα sectors που δεν έχουν ακόμη ολοκληρωθεί από τον οδηγό. Το πρώτο sector που δεν έχει ακόμη μεταφερθεί αποθηκεύεται στο`hard_sector`. Ο συνολικός αριθμός sector που απομένουν να μεταφερθούν αποθηκεύεται στο `hard_nr_sectors` και ο αριθμός των sectors που υπολείπονται στο τρέχων `bio` αποθηκεύεται στο `hard_cur_sectors`. Τα πεδία αυτά προορίζονται για χρήση μόνο από το block υποσύστημα και όχι από τους οδηγούς.

`struct bio *bio` : Το bio είναι η συνδεδεμένη λίστα των δομών `bio` για αυτή την αίτηση.

`char *buffer` : Το πεδίο αυτό, αν και χρησιμοποιείται από το παράδειγμα για να βρούμε τον `buffer` από/στον οποίο θα γίνει η μεταφορά, στην πραγματικότητα είναι το αποτέλεσμα της κλήσης του `bio_data` στο τρέχον `bio`. Είναι δηλαδή μία λογική διεύθυνση πυρήνα, που δείχνει στα δεδομένα που θα μεταφερθούν. Να επιση-

μάνουμε εδώ, ότι η διεύθυνση αυτή είναι διαθέσιμη μόνο αν η σελίδα βρίσκεται στην υψηλή μνήμη (high memory).

`unsigned short nr_phys_segments` : Ο αριθμός διακριτών τμημάτων στη φυσική μνήμη που κατέχει η αίτηση, μετά την συνένωση διαδοχικών σελίδων.

`struct list_head queuelist` : Η δομή της συνδεδεμένης λίστας που συνδέει την αίτηση στην ουρά αιτήσεων. Μόνο στην περίπτωση που αφαιρεθεί μία αίτηση από την ουρά με την `blkdev_dequeue_request`, μπορεί ο οδηγός να χρησιμοποιήσει αυτόν το δείκτη για να ανιχνεύσει την αίτηση σε εσωτερική λίστα αυτού.

Συναρτήσεις αποπεράτωσης/ολοκλήρωσης αιτήσεων

Για να ολοκληρώσουμε την εργασία μας με μία δομή `request` υπάρχουν διάφοροι τρόποι, όλοι τους όμως χρησιμοποιούν μερικές κοινές συναρτήσεις, οι οποίες χειρίζονται κατάλληλα την αποπεράτωση μίας αίτησης Ε/Ε ή μέρος αυτής. Όταν η συσκευή έχει ολοκληρώσει τη μεταφορά μερικών ή όλων των sectors μίας αίτησης Ε/Ε, πρέπει να ενημερώσει το επίπεδο block με τη συνάρτηση:

```
int end_that_request_first(struct request *req, int success, int count);
```

Η συνάρτηση αυτή ενημερώνει το επίπεδο block, πως ο οδηγός έχει ολοκληρώσει τη μεταφορά "`count`" sectors έχοντας ξεκινήσει από εκεί που σταμάτησε την τελευταία φορά. Αν η Ε/Ε ήταν επιτυχής η μεταβλητή `success` περνιέται ως 1, σε αντίθετη περίπτωση 0. Η τιμή επιστροφής της συνάρτησης `end_that_request_first` αποτελεί ένδειξη για το κατά πόσο όλα τα sectors της αίτησης έχουν μεταφερθεί ή όχι. Τιμή επιστροφής 0 σημαίνει ότι όλα τα sectors μεταφέρθηκαν και η αίτηση ολοκληρώθηκε. Στο σημείο αυτό η αίτηση πρέπει να αφαιρεθεί από την ουρά (αν δεν έχει αφαιρεθεί ήδη) με την `blkdev_dequeue_request` και να περαστεί στη:

```
void end_that_request_last(struct request *req);
```

Η συνάρτηση αυτή ενημερώνει οποιονδήποτε βρίσκεται σε αναμονή για αυτή την αίτηση πως αποπερατώθηκε και έπειτα την ανακυκλώνει. Η συνάρτηση αυτή καλείται πάντα με κρατημένο το κλείδωμα της ουράς. Στο παράδειγμα μας, αντί των δύο αυτών συναρτήσεων χρησιμοποιήσαμε την `end_request` που στην ουσία είναι ο συνδυασμός τους (υλοποίηση στον πυρήνα 2.6.10):

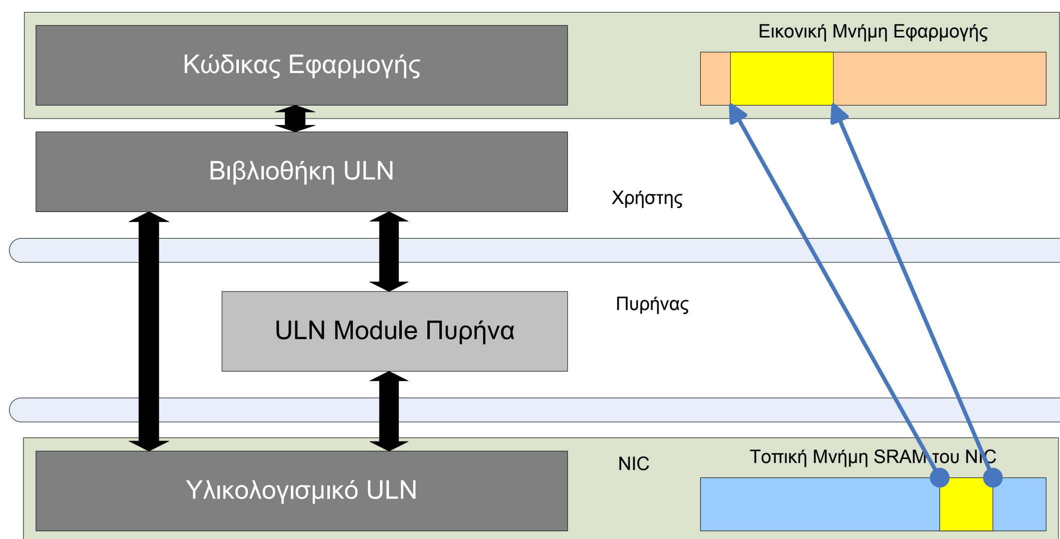
```
void end_request(struct request *req, int uptodate)
{
        if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
                add_disk_randomness(req->rq_disk);
                blkdev_dequeue_request(req);
                end_that_request_last(req);
        }
}
```

Η συνάρτηση `add_disk_randomness` δεν μας ενδιαφέρει αφού έχει να κάνει με την βοήθεια στην παραγωγή τυχαίων αριθμών του συστήματος.

2.2 Η διεπαφή Myrinet

2.2.1 Δικτύωση σε επίπεδο χρήστη (User Level Networking)

Οι πρόσφατες εξελίξεις στον τομέα των δικτύων υψηλής επίδοσης, επέφεραν μία σημαντικότατη αύξηση του διαθέσιμου έυρους ζώνης (bandwidth) στο φυσικό επίπεδο. Οι εφαρμογές σε υλικό προσφέρουν εύρος ζώνης της τάξης των 2-40Gbps και ταυτόχρονα ο αρχικός χρόνος απόκρισης (latency) στο υλικό και στο καλώδιο έχει μειωθεί στην κλίμακα των 0.3-1.0μs ανάλογα με το μέγεθος του δικτύου. Όλη όμως αυτή η ανάπτυξη δεν μπορεί εύκολα να μεταφερθεί στο επίπεδο εφαρμογών, κυρίως λόγω πολύπλοκων δομών που εδράζουν στον πυρήνα, όπως είναι οι στοίβες πρωτοκόλλων. Στην περίπτωση αυτή, εισάγονται μεγάλες καθυστερήσεις λογισμικού, αφού οι εφαρμογές αναγκάζονται να κάνουν κλήσεις συστήματος, κάτι που εμπλέκει το λειτουργικό σύστημα στο κρίσιμο μονοπάτι επικοινωνίας μεταξύ εφαρμογής και δικτύου διασύνδεσης. Για να μειωθεί το κόστος επικοινωνίας, τα περισσότερα μοντέρνα δίκτυα διασύνδεσης συστοιχιών υπολογιστών (clusters) προσπαθούν να παρακάμψουν, σε όσο το δυνατόν μεγαλύτερο βαθμό, το λειτουργικό σύστημα από το κρίσιμο μονοπάτι, χρησιμοποιώντας τεχνικές δικτύωσης σε επίπεδο χρήστη (user level networking). Σε αυτό το μοντέλο η διεργασία της εφαρμογής έχει τη δυνατότητα να ελέγχει τη διεπαφή δικτύου (network interface - NI) άμεσα. Από τη στιγμή που το λειτουργικό σύστημα δεν εμπλέκεται στην επικοινωνία, το αντικαθιστά ένας συνδυασμός από βιβλιοθήκες επιπέδου εφαρμογής και ένα υλικολογισμικό που τρέχει στον προσαρμογέα δικτύου. Ο κώδικας που δίνει τη δυνατότητα στις δύο αυτές οντότητες να ανταλλάσουν δεδομένα είναι προνομιούχος (privileged) και βρίσκεται σε ένα module πυρήνα. Η εφαρμογή επιτυγχάνει άμεσο έλεγχο στη διεπαφή δικτύου, αντικατοπτρίζοντας (map) μέρος των καταχωρητών ή



Σχήμα 2.8: *Διαστρωμάτωση ULN*

του χώρου μνήμης της διεπαφής, μέσα στο δικό της χώρο εικονικής μνήμης. Με τον τρόπο αυτό, μπορεί πλέον να χρησιμοποιήσει μη προνομιούχες (unprivileged) εντολές load/store στα σχετικά τμήματα (segments) εικονικής μνήμης για να πετύχει άμεση επικοινωνία.

Για να γίνει οποιαδήποτε απαλοιφή του λειτουργικού συστήματος ή της CPU από το κρίσιμο μονοπάτι, πρέπει μέρος της λειτουργικότητας να υλοποιηθεί τοπικά πάνω στον ίδιο τον προσαρμογέα δικτύου. Αυτό υποδηλώνει έναν ”έξυπνο” προσαρμογέα δικτύου με κάποια δυνατότητα προγραμματισμού. Βέβαια το μέγεθος προγραμματισιμότητας είναι ζήτημα προς συζήτηση και σχετίζεται κυρίως με το σχεδιασμό του πρωτοκόλλου δικτύωσης επιπέδου χρήστη (ULN Protocol). Ακόμη, σχεδιάζοντας ένα πρωτόκολλο που υποστηρίζει λειτουργίες στο χώρο χρήστη, ερχόμαστε αντιμέτωποι με δύο βασικά ζητήματα, που είναι η κίνηση δεδομένων και η μετάφραση διευθύνσεων.

Όπως προαναφέραμε, για να έχουμε επικοινωνία, γίνεται μεταφορά δεδομένων από τη φυσική RAM (μνήμη εφαρμογής) σε κάποιο μέρος της τοπικής μνήμης της διεπαφής δικτύου. Αυτή η διαδικασία μπορεί να γίνει με διάφορους τρόπους. Ένας από αυτούς είναι η προγραμματισμένη Ε/Ε που αναφέραμε, δηλαδή με loads/stores που χρησιμοποιεί η εφαρμογή για να μεταφέρει τα μηνύματα σε διευθύνσεις που αντικατοπτρίζουν τη μνήμη της διεπαφής δικτύου. Ο τρόπος αυτός βελτιστοποιείται με τη μέθοδο write-combining, σύμφωνα με την οποία όλα τα stores ομαδοποιούνται και γράφονται στη μνήμη με μία συνδιαλλαγή, γλιτώνοντας έτσι κύκλους CPU αλλά και πολλές συνδιαλ-

λαγές που επιβαρύνουν τους διαδρόμους συστήματος.

Μία άλλη μέθοδος που χρησιμοποιούν οι περισσότερες μοντέρνες κάρτες διασύνδεσης συστοιχιών υπολογιστών, είναι η άμεση πρόσβαση στη μνήμη (direct memory access - DMA). Πάνω στον προσαρμογέα δικτύου υπάρχουν μηχανές DMA που έχουν πρόσβαση στην κύρια μνήμη του συστήματος και μεταφέρουν από αυτή μεγάλου μεγέθους δεδομένα που περικλείουν μηνύματα, κατευθείαν στην τοπική μνήμη της κάρτας. Αυτό συνεπάγεται πλήρη αφαίρεση της CPU από το κρίσιμο μονοπάτι επικοινωνίας και επιτρέπει την επικάλυψη υπολογισμού και επικοινωνίας. Το πρόβλημα που προκύπτει σε αυτή την περίπτωση είναι η συνέπεια της μνήμης. Από τη στιγμή που τα μηνύματα επικοινωνίας αποθηκεύονται στον εικονικό χώρο μνήμης της εφαρμογής και το λειτουργικό σύστημα δεν γνωρίζει τίποτα για αυτά, μπορεί να τα έχει ανταλλάξει με άλλες θέσεις μνήμης (swap). Έτσι τη στιγμή που η εφαρμογή θέλει να στείλει ένα μήνυμα ή την ώρα που φτάνει κάποιο άλλο, στη μνήμη δεν θα υπάρχουν τα κατάλληλα δεδομένα με αποτέλεσμα να έχουμε σύγκρουση (conflict). Η άμεση πρόσβαση στη μνήμη αναφέρεται σε φυσικές διευθύνσεις. Αυτό μπορεί να καταλήξει μέχρι και σε αλλοίωση μνήμης αν π.χ. ο πυρήνας (που δεν βρίσκεται πλέον στο κρίσιμο μονοπάτι) αποφασίσει να ανταλλάξει μία σελίδα μνήμης κατά τη διάρκεια μιας μεταφοράς δεδομένων. Τελικά το πρόβλημα αντιμετωπίζεται με διάφορες μεθόδους, η ανάλυση των οποίων ξεπερνά το σκοπό αυτής της εργασίας. Παραδειγματικά αναφέρουμε το μαρκάρισμα κάποιων σελίδων ώς unswappable ή τη χρήση προκαθορισμένων περιοχών μνήμης που δεσμεύονται για DMA.

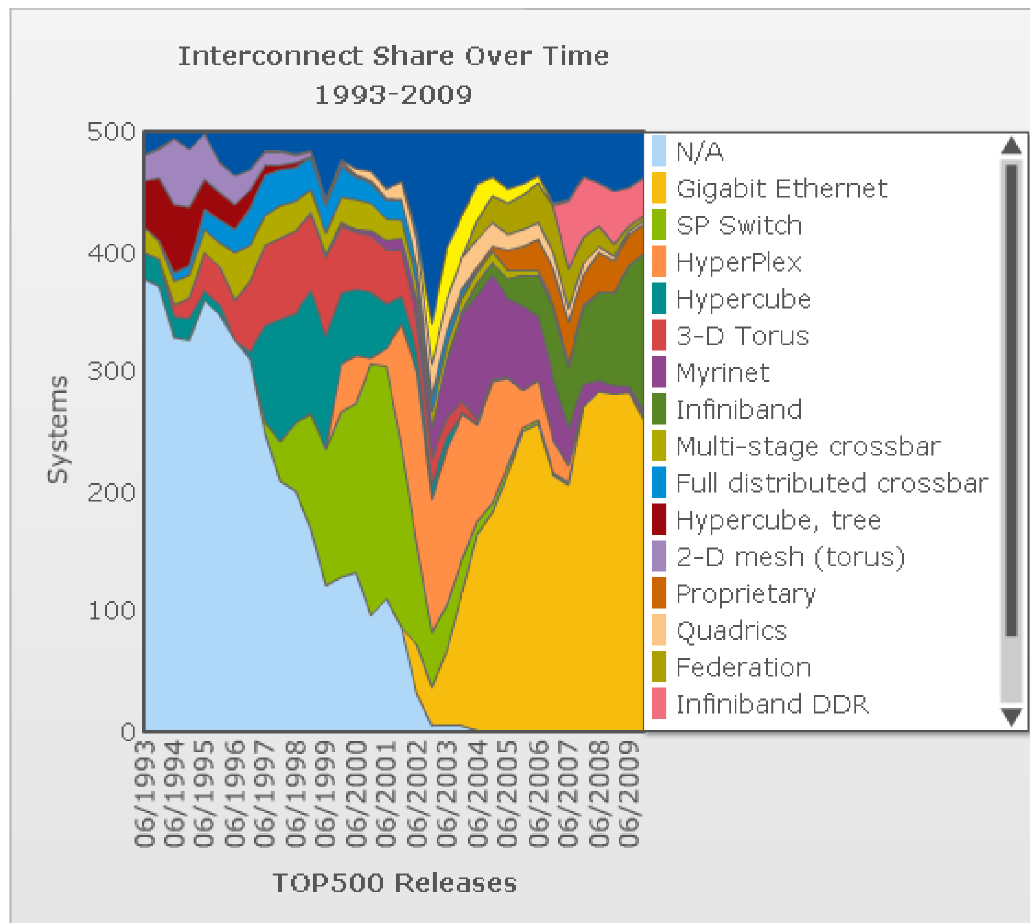
Το δεύτερο ζήτημα, που έχει να κάνει με τη μετάφραση διευθύνσεων είναι επίσης σημαντικό. Με τον ένα ή τον άλλο τρόπο, τελικά θα χρειαστεί κάποια στιγμή να γίνει μετάφραση κάποιας εικονικής διεύθυνσης του χώρου μνήμης της εφαρμογής, σε μία φυσική διεύθυνση για να μπορέσει να γίνει η πραγματική μεταφορά των δεδομένων. Στην περίπτωση που έχουμε προγραμματισμένη Ε/Ε το ζήτημα είναι σχετικά απλό, αφού τη μετάφραση την κάνει η μονάδα διαχείρισης μνήμης του επεξεργαστή (memory management unit - MMU). Όταν έχουμε όμως DMA και ο πυρήνας δεν λαμβάνει μέρος, δεν υπάρχει κανένας μηχανισμός που να επιτελεί το συγκεκριμένο έργο. Ο μηχανισμός πρέπει να υλοποιηθεί από την αρχή από το πρωτόκολλο δικτύωσης σε επίπεδο χρήστη. Για να γίνει αυτό υπάρχουν πολλές εναλλακτικές υλοποιήσεις, που σχετίζονται με το διαθέσιμο υλικό στην διεπαφή δικτύου και τις επιλογές μεταφοράς δεδομένων που έχουν γίνει κατά τη σχεδίαση του πρωτοκόλλου. Τα κύρια ερωτήματα έχουν να κάνουν με την

επιλογή της πλευράς που θα κάνει τη μετάφραση και της πλευράς που θα επιβαρυνθεί να χειριστεί τις αστοχίες μνήμης. Με τον όρο πλευρά αναφερόμαστε είτε στη CPU είτε στην ίδια την διεπαφή δικτύου. Η μετάφραση από την πλευρά της CPU γίνεται με αναζήτηση διευθύνσεων από διεπαφή εκτεθειμένη στον πυρήνα και μετά με μεταφορά της φυσικής διεύθυνσης στην κάρτα δικτύου. Αν διαλέξουμε την εργασία να την κάνει η ίδια η διεπαφή δικτύου πρέπει να δεσμεύσουμε ένα μικρό κομμάτι της δικής της φυσικής μνήμης στην οποία θα φαίνεται η αντιστοιχία φυσικών σε εικονικών θέσεων μνήμης. Στη συνέχεια η προγραμματιζόμενη μονάδα που έχει η διεπαφή θα κάνει την αναζήτηση στη δική της μνήμη. Αντίστοιχα για την αντιμετώπιση αστοχιών πάλι μπορεί να επιβαρυνθεί είτε η μία είτε η άλλη πλευρά, με τη διαφορά ότι σε αυτό το κομμάτι εργασίας υπάρχουν υλοποιήσεις που συνδυάζουν και τις δύο. Τελικά κάθε επιλογή έχει άμεσο αντίκρυσμα και σε συγκεκριμένα σημεία επίδοσης που θέλει να επικεντρωθεί ο σχεδιαστής του πρωτοκόλλου.

2.2.2 Δικτύωση με το στρώμα Myrinet/GM

Το Myrinet είναι μία δικτυακή υποδομή για συστοιχίες υπολογιστών υψηλής επίδοσης και βασίζεται σε τεχνολογίες που αναπτύχθηκαν για τη διασύνδεση πολλαπλών επεξεργαστών σε αρχιτεκτονικές MPP (Massivily Parallel Processors). Προσφέρει ταχύτητες σύνδεσης μέχρι και 10Gbps, δυνατότητα μετάδοσης δεδομένων και στις δύο κατευθύνσεις (full duplex) και μικρούς χρόνους αρχικής απόκρισης. Το Myrinet βρίσκει εφαρμογή σε συστοιχίες υπολογιστών και γενικότερα χρησιμοποιείται ως δίκτυο περιοχής συστήματος (System Area Network - SAN). Δύο γενιές Myrinet είναι διαθέσιμες αυτή τη στιγμή: το Myrinet-2000 και το Myri-10G. Το φυσικό επίπεδο του Myrinet-2000 χρησιμοποιεί οπτικές συνδέσεις σημείου–προς–σημείο 2+2Gbps, και μεταδίδει δεδομένα και προς τις δύο κατευθύνσεις (full duplex). Το Myri-10G βασίζεται στο ίδιο φυσικό επίπεδο όπως και το 10G Ethernet, αυξάνοντας έτσι το εύρος διαύλου στα 10Gbps και μπορεί να χρησιμοποιήσει ως επίπεδο συνδέσμου είτε το 10G Ethernet, είτε το Myrinet με δρομολόγηση από την πηγή. Στην παρούσα εργασία θα παρουσιάσουμε διεξοδικά το Myrinet-2000 μιας και αυτό αποτέλεσε την πλατφόρμα της δικής μας υλοποίησης.

Οι κόμβοι του δικτύου Myrinet διασυνδέονται με μεταγωγείς τύπου Crossbar, σε μια τοπολογία Clos. Ένα από τα πιο σημαντικά χαρακτηριστικά του δικτύου είναι η δρομολόγηση από την πηγή: το δίκτυο χαρτογραφείται έτσι ώστε κάθε κόμβος που συμ-

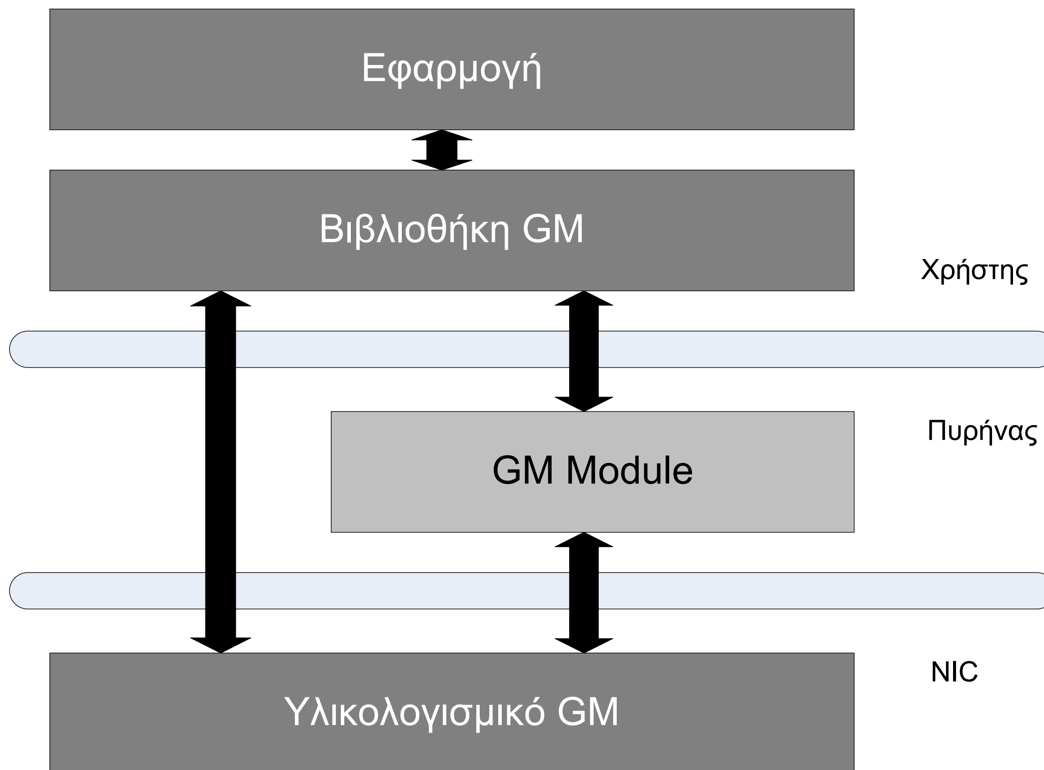


Σχήμα 2.9: *Στατιστικά Δικτύων Διασύνδεσης*

μετέχει να γνωρίζει τη διαδρομή μέχρι κάποιον άλλο, χρησιμοποιώντας δρομολόγηση up/down. Κάθε πακέτο περιέχει όλη την πληροφορία για τη διαδρομή μέχρι τον προορισμό του. Αυτό υλοποιείται ως μια σειρά από πόρτες που πρέπει να διασχίσουν κάθε μεταγωγέα. Για την επικοινωνία των κόμβων χρησιμοποιούνται πακέτα μεταβλητού μήκους. Με αυτόν τον τρόπο επιτυγχάνεται πολύ μικρός χρόνος αρχικής απόκρισης κατά τις μεταγωγές.

Για να μειωθεί η επιβάρυνση από τη συμμετοχή του λειτουργικού συστήματος στην επικοινωνία, το Myrinet εφαρμόζει τεχνικές πρωτοκόλλου δικτύωσης σε επίπεδο χρήστη, σαν αυτές που παρουσιάστηκαν παραπάνω. Χρησιμοποιώντας αυτό το μοντέλο, απεμπλέκεται τελείως ο πυρήνας και μια εφαρμογή μπορεί να ελέγχει τη διεπαφή του δικτύου απευθείας. Όπως είδαμε αμέσως πριν, καθώς ο πυρήνας δεν εμπλέκεται στην επικοινωνία, το ρόλο του αναλαμβάνουν μια βιβλιοθήκη στο χώρο χρήστη, καθώς και το υλικολογισμικό (firmware) στη διεπαφή δικτύου. Η ανταλλαγή δεδομένων μεταξύ

της εφαρμογής και του προσαρμογέα δικτύου γίνεται με ένα μηχανισμό που προετοιμάζεται από κώδικα με αυξημένα δικαιώματα μέσα σε ένα module. Η εφαρμογή ελέγχει τον προσαρμογέα δικτύου με προγραμματισμένη Ε/Ε και αντιστοίχιση εικονικού και φυσικού χώρου μνήμης όπως αναλύσαμε προηγούμενα (VM mappings). Στο Σχ. 2.10 φαίνεται αυτή η δομή λειτουργίας:



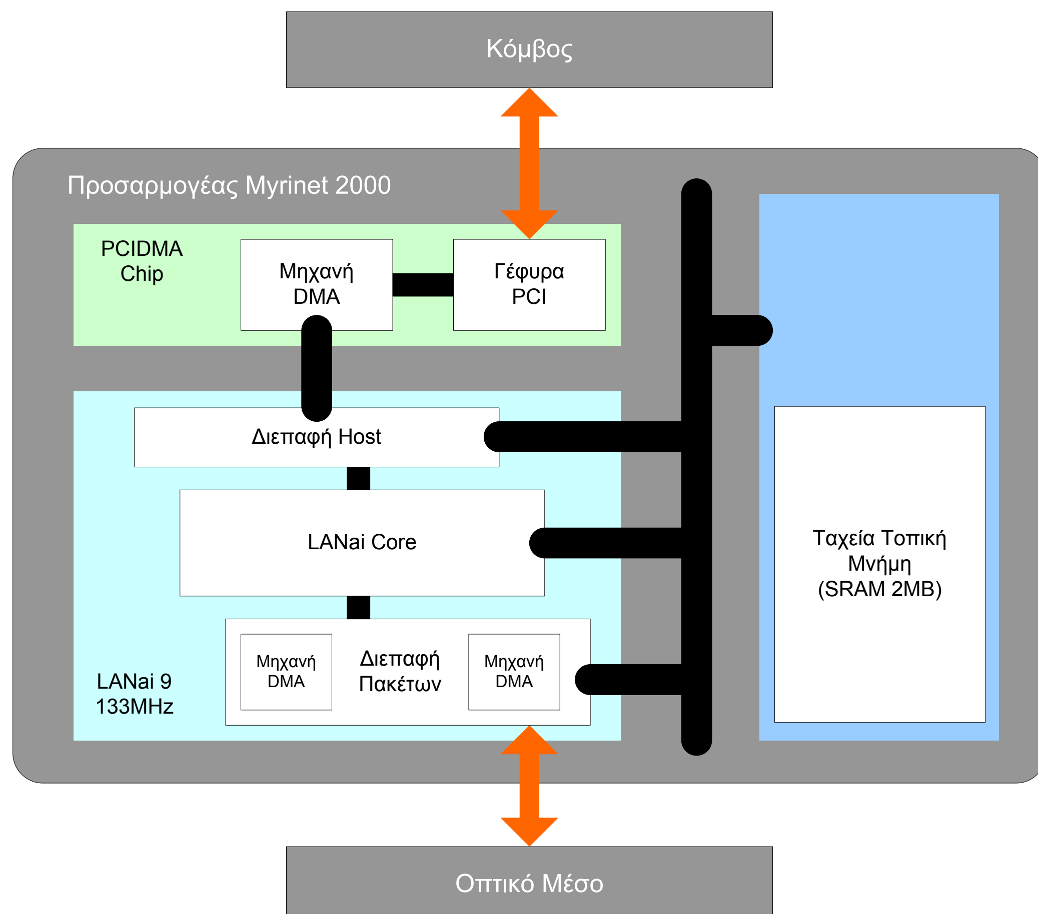
Σχήμα 2.10: *Διεπαφή GM*

Οι ελεγκτές διεπαφών δικτύου (Network Interface Controller - NIC) εδρεύουν σε ένα περιφερειακό δίαυλο, πάνω σε κάθε κόμβο της συστοιχίας, στη περίπτωση του Myrinet-2000 με το οποίο δουλεύουμε είναι το PCI/PCI-X. Ο ελεγκτής ενσωματώνει έναν μικροεπεξεργαστή RISC, τον Lanai, ο οποίος αναλαμβάνει το μεγαλύτερο μέρος της επεξεργασίας του πρωτοκόλλου του δικτύου, μια μικρή μνήμη SRAM (2MB) για τις ανάγκες του Lanai και τρεις διαφορετικές μηχανές DMA. Η πρώτη, είναι υπεύθυνη για τις μεταφορές DMA των δεδομένων που περικλείουν μηνύματα, μεταξύ της κύριας μνήμης του συστήματος και της SRAM του Lanai, πάνω από ένα half-duplex PCI/PCI-X δίαυλο. Οι άλλες δύο χρεώνονται όλη τη μεταφορά δεδομένων, μεταξύ της μνήμης SRAM του Lanai και του full-duplex 2+2Gbps οπτικού συνδέσμου. Για την παροχή όλων των υπηρεσιών του δικτύου σε επίπεδο χρήστη στις εφαρμογές, χρησιμοποιείται το σύστη-

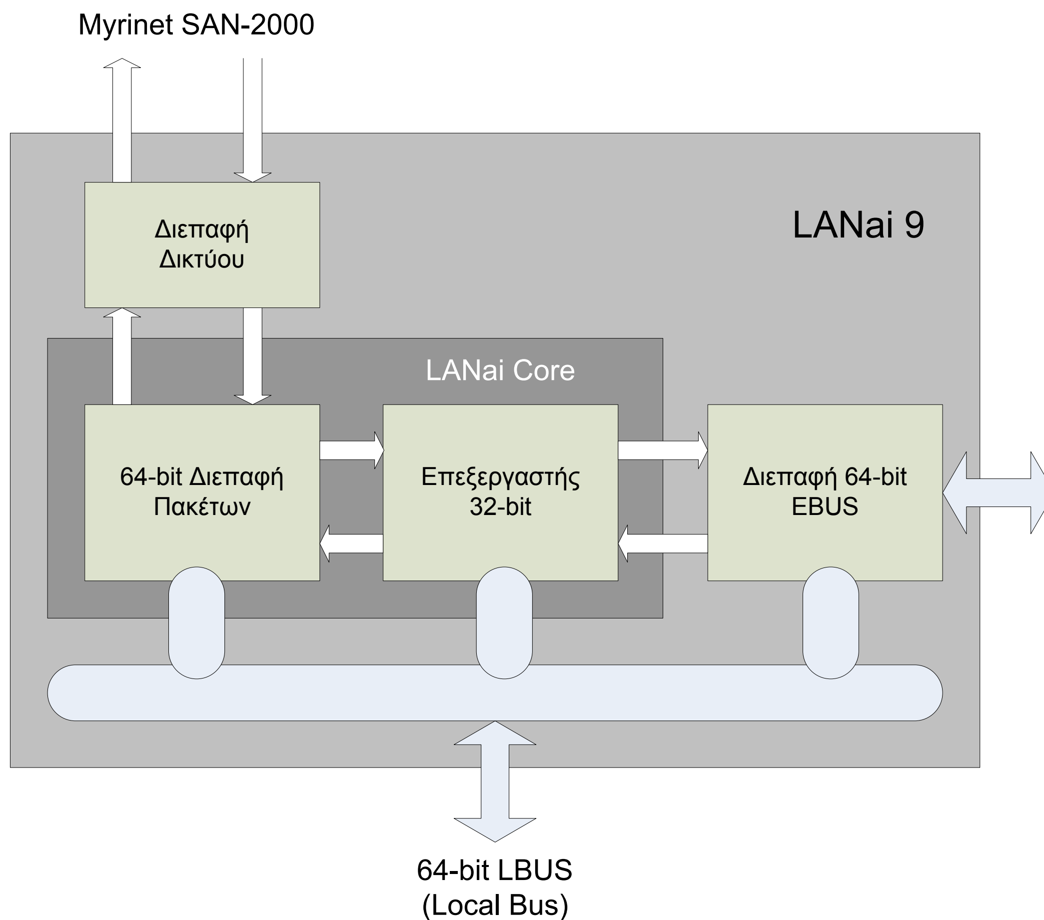
μα μετάδοσης μηνυμάτων GM. Το GM αποτελείται απο:

- Το υλικολογισμικό (firmware) που τρέχει στον Lanai.
- Ένα module πυρήνα του λειτουργικού συστήματος.
- Μια βιβλιοθήκη στο χώρο χρήστη που χρησιμοποιείται από τις εφαρμογές

Τα τρία αυτά μέρη συνεργάζονται και συντονίζονται έτσι ώστε να επιτρέπουν άμεση πρόσβαση στο NIC από το χώρο χρήστη, χωρίς να χρειάζεται να εμπλακεί ο πυρήνας μέσω κλήσεων συστήματος. Επίσης διατηρείται η ακεραιότητα του συστήματος, εξασφαλίζεται η απομόνωση των διεργασιών και διασφαλίζεται η προστασία της μνήμης. Υλοποιείται δηλαδή πλήρως ένα πρωτόκολλο δικτύωσης σε επίπεδο χρήστη.Τα τρία μέρη και η λειτουργία τους φαίνονται καθαρά στο Σχ. 2.10.



Σχήμα 2.11: *Προσαρμογέας δικτύου Myrinet-2000*



Σχήμα 2.12: *Μικροεπεξεργαστής LANai9*

Στα Σχ. 2.11 και Σχ. 2.12 παρουσιάζονται: τα βασικά μέρη που βρίσκονται πάνω σε μία κάρτα Myrinet M3F-PCI64B-2 και το εσωτερικό του Lanai9. Η έκδοση αυτή του ελεγκτή Myrinet-2000 χρησιμοποιεί διακριτές ψηφίδες και παρουσιάζεται για μεγαλύτερη σαφήνεια. Έτσι διακρίνουμε μία μηχανή DMA στο PCIDMA chip που είναι υπεύθυνο για την επικοινωνία με τον εσωτερικό δίαυλο PCI και άλλες δύο στο Packet Interface. Επίσης φαίνεται ο Lanai και η SRAM.

Η επικοινωνία με μηδενικά αντίγραφα στο χώρο χρήστη επιτυγχάνεται με αντικατοπτρισμό (mapping) μερών της SRAM του Lanai (ονομάζονται GM ports) σε διευθύνσεις στον εικονικό χώρο μνήμης μίας εφαρμογής. Αυτή είναι προνομιούχος λειτουργία και γίνεται μέσω κλήσεων συστήματος στο GM module κατά τη φάση αρχικοποίησης της εφαρμογής. Κάθε port έχει το ρόλο ενός άκρου επικοινωνίας για την εφαρμογή. Το GM firmware ελέγχει περιοδικά όλα τα ports για να εντοπίσει οποιαδήποτε καινούρια αλλαγή. Στην περίπτωση που υπάρχει μία αίτηση για μία λειτουργία send, τότε χρησι-

μοποιεί τη μηχανή DMA για να μεταφέρει τα δεδομένα που απαιτούνται από τη κύρια μνήμη RAM στη μνήμη SRAM του Lanai και στη συνέχεια από την SRAM στον ελεγκτή του κόμβου παραλήπτη. Η ίδια διαδικασία αλλά με ανάποδη σειρά ακολουθείται στην περίπτωση που αντί για send, έχουμε receive. Το πρωτόκολλο που χρησιμοποιείται για τη μετάδοση δεδομένων μεταξύ των κόμβων είναι το "Go back N", το οποίο στοχεύει στην ελαχιστοποίηση των αρχικών χρόνων απόκρισης και του φόρτου του λογισμικού, σε βάρος μερικές φορές του εύρους ζώνης.

Το firmware του GM, που ονομάζεται και MCP (Myrinet Control Program), είναι οργανωμένο σε τέσσερις μηχανές καταστάσεων. Αυτές είναι οι: SDMA, SEND, RECV και RDMA, καθεμια από τις οποίες είναι υπεύθυνη για ένα συγκεκριμένο κομμάτι της επεξεργασίας του πρωτοκόλλου. Οι μηχανές SDMA και RDMA έχουν να κάνουν με την εσωτερική επικοινωνία, δηλαδή την επικοινωνία και μεταφορά δεδομένων μεταξύ εφαρμογής και NIC του ίδιου κόμβου, ενώ οι μηχανές SEND και RECV υλοποιούν την επικοινωνία μεταξύ διαφορετικών κόμβων, δηλαδή διαφορετικών NIC του δικτύου.

SDMA Η μηχανή αυτής της κατάστασης ελέγχει κυκλικά όλα τα ανοιχτά ports σε αναζήτηση καινούριου γεγονότος send που έχει προκληθεί από την εφαρμογή. Μόλις το εντοπίσει, κάνει όλες τις απαραίτητες εργασίες για να προετοιμάσει τα δεδομένα βάσει του κόμβου στον οποίο πρέπει να σταλούν. Επίσης αντιλαμβάνεται πότε μία εφαρμογή ζητά να δεχτεί εισερχόμενα δεδομένα και δημιουργεί τις κατάλληλες ενδείξεις. Στη συνέχεια, η μηχανή SDMA ξεκινά ένα `read DMA` έτσι ώστε να μεταφερθούν τα δεδομένα από την κεντρική μνήμη RAM στους κατάλληλους buffers στη μνήμη SRAM του Lanai. Τελικά προωθεί τα δεδομένα στη μηχανή κατάστασης SEND ώστε να εγχυθούν στο δίκτυο. Τέλος η SDMA κρατά κάποιο ιστορικό των πακέτων προς αποστολή για να διασφαλίσει την σωστή αποστολή τους.

RDMA Η μηχανή αυτή κάνει ουσιαστικά το αντίστροφο έργο της SDMA. Δέχεται εισερχόμενα πακέτα που έχουν φτάσει από κάποιο άλλο κόμβο του δικτύου και τις τα έχει προωθήσει η RECV. Τότε προσπαθεί να τα ταιριάξει με κάποια ένδειξη που εκρεμμεί (γιατί κάποια εφαρμογή ζήτησε δεδομένα και η SDMA δημιούργησε την ένδειξη). Αν επιτύχει ξεκινά μια διαδικασία `write DMA` για να μεταφέρει τα δεδομένα από τη μνήμη του Lanai στην κεντρική μνήμη του συστήματος. Όταν η διαδικασία ολοκληρωθεί επιτυχώς και τα δεδομένα βρίσκονται στο προβλεπό-

μενο σημείο, ειδοποιεί την εφαρμογή που τα είχε αιτηθεί. Τέλος η RDMA είναι υπεύθυνη να δημιουργήσει τα κατάλληλα (N)ACK πακέτα ελέγχου, αφού πρώτα έχει κάνει την πιστοποίηση που απαιτείται.

SEND Η μηχανή κατάστασης SEND λαμβάνει τα πακέτα που έχει προετοιμάσει η SDMA και της έχει προωθήσει, καθώς επίσης και τα πακέτα (N)ACK από την RDMA και τα προωθεί με τη σειρά της στο δίκτυο. Αυτό το κάνει προγραμματίζοντας κατάλληλα τη μηχανή SEND DMA της διεπαφής πακέτων που βρίσκεται στην ψηφίδα.

RECV Όπως αναφέρει και το όνομα της η μηχανή κατάστασης RECV (Receive) παραλαμβάνει όλα τα εισερχόμενα πακέτα που έρχονται από τους διάφορους κόμβους του δικτύου. Στην περίπτωση που τα πακέτα περιέχουν δεδομένα εισερχόμενων μηνυμάτων, προωθούνται στη μηχανή RDMA για να φτάσουν από εκεί στην εφαρμογή. Αν πρόκειται για πακέτα ελέγχου τότε η ίδια η RECV τροποποιεί το ιστορικό κατάλληλα. Για acknowlegment το ιστορικό του εισερχόμενου πακέτου διαγράφεται και ενημερώνεται η εφαρμογή στην περίπτωση τελικού πακέτου (send complete). Για αρνητικό acknowlegment το ιστορικό διαγράφεται, αλλά επανατροποποιούνται οι δείκτες που το αφορούν έτσι ώστε το πακέτο να ξανασταλεί.

Από τα παραπάνω φαίνεται καθαρά ότι, είτε λαμβάνουμε κάποιο μήνυμα είτε στέλνουμε, μέσω της διεπαφής GM η διαδικασία ολοκληρώνεται σε δύο φάσεις.

Για λειτουργία send:

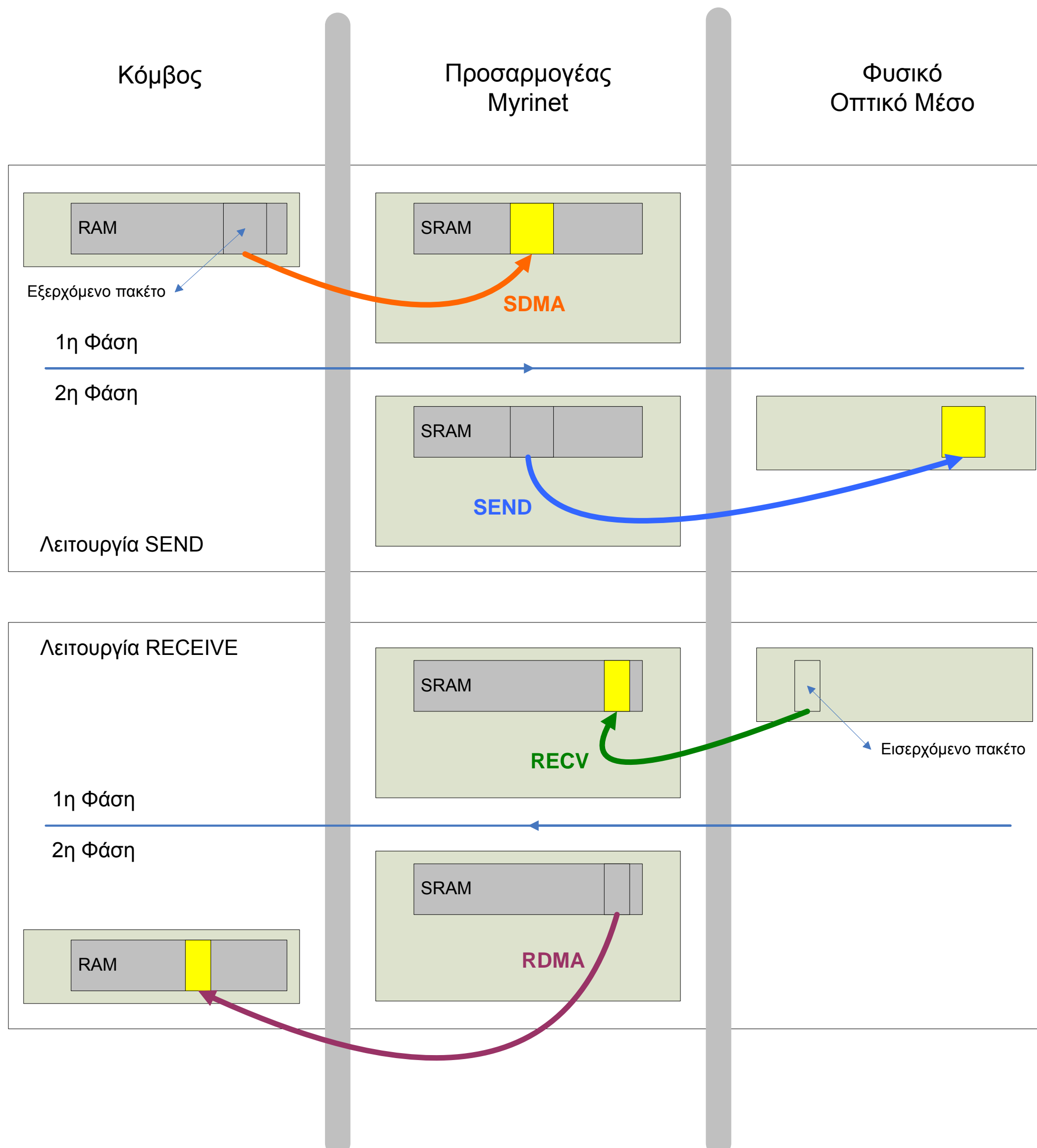
1. DMA από RAM σε Lanai :

- μετάφραση από εικονική μνήμη σε φυσική
- χρήση της PCIDMA μηχανής
- μεταφορά δεδομένων από RAM συστήματος σε Lanai SRAM

2. DMA από Lanai στο φυσικό μέσο :

- τα δεδομένα ανακτώνται από την SRAM
- χρήση της SEND DMA μηχανής

- μεταφορά δεδομένων πάνω στο φυσικό μέσο προς άλλο NIC



Σχήμα 2.13: *Μηχανισμοί αποστολής και λήψης πακέτου*

Για λειτουργία receive:

1. DMA από φυσικό μέσο στο Lanai :

- εισερχόμενα δεδομένα στο φυσικό μέσο από άλλο NIC
- χρήση της RECEIVE DMA μηχανής
- μεταφορά των δεδομένων στην SRAM

2. DMA από Lanai στη RAM :

- μετάφραση από φυσική μνήμη σε εικονική
- χρήση της PCIDMA μηχανής
- μεταφορά δεδομένων από Lanai SRAM στη RAM συστήματος

Τελικά, για να έχουμε μία γενικότερη εικόνα των χαρακτηριστικών που προσφέρει το GM, θα μπορούσαμε να τα συνοψίσουμε στα παρακάτω:

- Πλήρης υλοποίηση ενός πρωτοκόλλου δικτύωσης σε επίπεδο χρήστη.
- Προστατευόμενη και ανεξάρτητη πρόσβαση στις δικτυακές λειτουργίες για τις εφαρμογές.
- Χαμηλός χρόνος πρώτης απόκρισης (latency).
- Κόστος υπολογισμών στο NIC και όχι στο λογισμικό του host
- Επικάλυψη της επικοινωνίας και των υπολογισμών
- Ασύγχρονες λειτουργίες επικοινωνίας.
- Διαφανής διαχείριση της μνήμης.
- Αποκατάσταση λαθών στο δίκτυο και υψηλή διαθεσιμότητα.
- Υποστήριξη για δρομολόγηση διασποράς.
- Βασικός μηχανισμός ταυτοποίησης για κάθε μήνυμα.
- Μηχανισμός γενικευμένης ταυτοποίησης μηνυμάτων.
- Υποστήριξη αποστολής και λήψης μηνυμάτων από και προς διάσπαρτες θέσεις στη μνήμη.
- Αποδοτική υποστήριξη για μη αναμενόμενα μηνύματα
- Για κάθε μήνυμα ή για κάθε άκρο συναρτήσεις που κάνουν περιοδικό έλεγχο (polling) ή σταματούν τη ροή ελέγχου μέχρι την ολοκλήρωσή τους (blocking)
- Βιβλιοθήκες τόσο ενός νήματος αλλά και ασφαλών νημάτων.

- Υποστήριξη ακύρωσης των αιτήσεων σε αναμονή.
- Ενσωματωμένη υποστήριξη χαρτογράφησης του Myrinet υλικού.

Γίνεται φανερό ότι το περιβάλλον Myrinet/GM υλοποιεί ένα πλήθος λειτουργιών και αποτελεί μία ολοκληρωμένη πλατφόρμα ενός πολύπλευρου δικτύου διασύνδεσης. Αυτός είναι και ο λόγος άλλωστε που χρησιμοποιείται σε πολυσύνθετες εφαρμογές μεγάλης πολυπλοκότητας αλλά και υψηλών απαιτήσεων, όπως είναι οι πολύ μεγάλης κλίμακας συστοιχίες υπολογιστών (large scale clustering systems). Στη συγκεκριμένη εργασία δεν θα χρειαστεί να επηρεάσουμε όλα τα υποσυστήματα του Myrinet/GM αλλά θα προσπαθήσουμε να σχεδιάσουμε και να υλοποιήσουμε με τη μικρότερη δυνατή τροποποίηση του αρχικού κώδικα. Μιας και η εργασία αποτελεί ουσιαστικά μία μελέτη περίπτωσης, θα προσπαθήσουμε να εκμεταλλευτούμε στο μέγιστο όλες τις δυνατότητες που μας δίνει το υλικό, αλλά και το πρωτότυπο λογισμικό, ενώ ταυτόχρονα θα επικεντρωθούμε στο θέμα μας και θα κάνουμε κάθε δυνατή προσπάθεια να κρατηθεί ο κώδικας καθαρός (clean), αυτοδύναμος (robust) και διαμορφώσιμος (modular). Όλα τα παραπάνω υποδηλώνουν την ανάγκη ένος ολοκληρωμένου και προσεκτικού σχεδιασμού μετά από κατανόηση του θεωρητικού υποβάθρου, τόσο όσον αφορά στη διαστρωμάτωση των επιπέδων του πυρήνα Linux και τις λειτουργίες τους, όσο και στην αντίστοιχη διαστρωμάτωση και λειτουργίες του περιβάλλοντος Myrinet/GM. Απαραίτητη προϋπόθεση είναι η κατανόηση της σχέσης των δύο και η αλληλοεπίδρασή τους. Έχοντας πλέον μία συνολική εικόνα των θεωρητικών εννοιών που θα εμπλακούν κατά την εκπόνηση της εργασίας, προχωρούμε στην παρουσίαση του σχεδιαστικού μέρους.

3

Σχεδιασμός της συσκευής MyriBLK

Στο κεφάλαιο αυτό, θα ασχοληθούμε με το σχεδιασμό της συσκευής myriblk και θα παρουσιάσουμε όλες τις δομές και λειτουργίες που χρειάζονται να υλοποιηθούν, για να καταλήξουμε σε μία πλήρως λειτουργική συσκευή. Το γεγονός ότι η υλοποίηση μίας τέτοιας συσκευής εμπλέκει την εργασία σε πολλά επίπεδα του λειτουργικού συστήματος αλλά και σε εξειδικευμένο υλικό, μας υποχρεώνει να προχωρήσουμε σε έναν ολοκληρωμένο σχεδιασμό πριν κάνουμε οποιαδήποτε προσπάθεια υλοποίησης. Αρχικά, στο κεφάλαιο αυτό θα περιγράψουμε με ακρίβεια το είδος της συσκευής που καλούμαστε να υλοποιήσουμε και τις λειτουργίες που πρέπει να επιτελεί. Θα κάνουμε κάποιες αρχικές επιλογές που θα μας βοηθήσουν να αποκτήσουμε μία βάση για να ξεκινήσουμε, στη συνέχεια θα περάσουμε στον κυρίως σχεδιασμό και τελικά θα παρουσιάσουμε τις τελικές επιλογές στις οποίες καταλήγουμε μετά την ανάλυση και πάνω στις οποίες θα υλοποιήσουμε.

Ο κυρίως σχεδιασμός χωρίζεται σε τρία τμήματα όπως θα δούμε στη συνέχεια. Στο πρώτο μέρος σχεδιάζουμε το κομμάτι της συσκευής από την πλευρά του κόμβου. Αναλύουμε τα προβλήματα που προκύπτουν από τις διεπαφές με τον πυρήνα και το λειτουργικό σύστημα και προσανατολίζουμε το σχεδιασμό έτσι ώστε να καλύπτονται οι ανάγκες λειτουργικότητας της συσκευής, προσαρμόζοντας παράλληλα τη σχεδίαση έτσι ώστε να υπάρχει μία απρόσκοπτη και διαφανής επικοινωνία με το υπάρχον σύστημα, χωρίς να επηρεάζονται οι υπόλοιπες ενέργειές του. Στο δεύτερο μέρος ασχολούμαστε με το κομμάτι της συσκευής από την πλευρά του προσαρμογέα δικτύου. Μελετάμε την αρχιτεκτονική του υλικού, για να εκμεταλλευτούμε όσο το δυνατόν περισσότερες από τις δυνατότητες που μας προσφέρει. Μελετάμε επίσης το υλικολογισμικό και τη διαδικα-

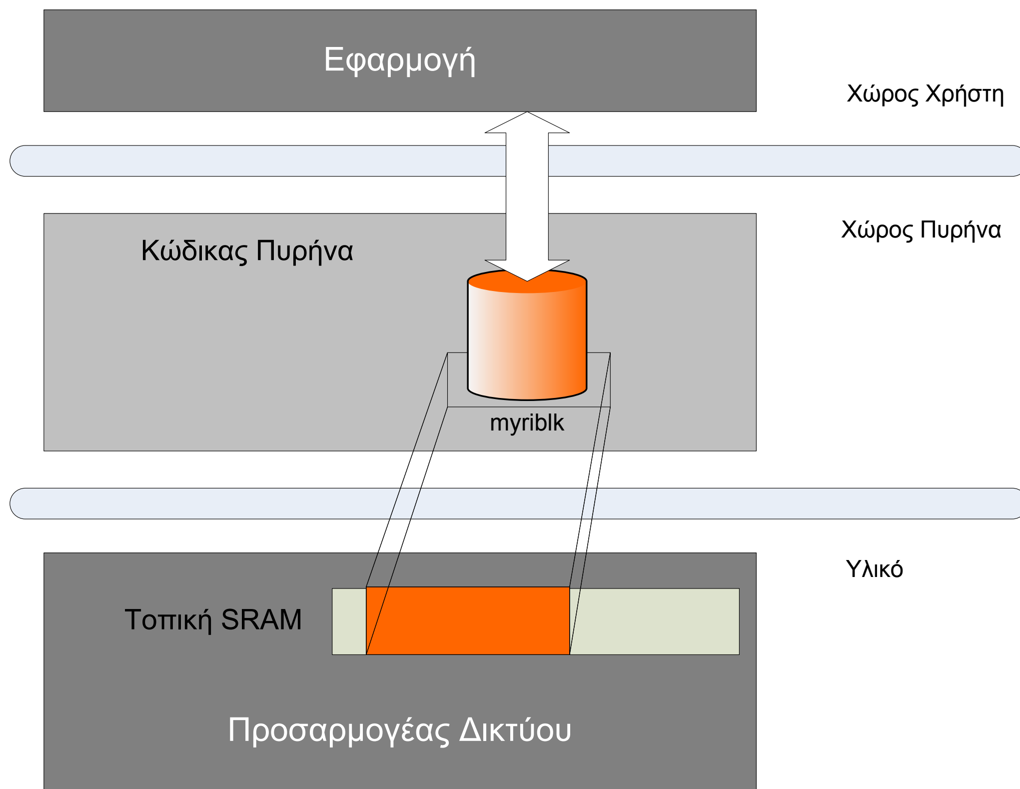
σία με την οποία μπορούμε να το τροποποιήσουμε, για την ικανοποίηση των αναγκών μας. Τέλος, στο τρίτο κομμάτι φτάνουμε να αντιμετωπίσουμε το πρόβλημα της επικοινωνίας των δύο παραπάνω πλευρών και του συγχρονισμού τους. Παρουσιάζουμε τους διαφορετικούς τρόπους συγχρονισμού και καταλήγουμε στον τελικό, που ταιριάζει στις απαιτήσεις μας. Να σημειώσουμε, ότι καθ' όλη τη διάρκεια σχεδιασμού, έχουμε συνεχώς υπ' όψιν το θέμα της επίδοσης και μεγάλο μέρος των επιλογών μας γίνεται με αυτό τον γνώμονα. Τις επιλογές αυτές θα τονίσουμε στα σημεία που θα τις συναντήσουμε, κατά τη διάρκεια του κεφαλαίου αυτού.

3.1 Γενικά

Στην παρούσα εργασία, σχεδιάζουμε και να υλοποιούμε μία συσκευή block πάνω στην ενσωματωμένη μνήμη ενός προσαρμογέα δικτύου υψηλής επίδοσης. Ουσιαστικά θέλουμε έναν τρόπο που να επιτρέπει στις εφαρμογές να βλέπουν μέρος της μνήμης του προσαρμογέα δικτύου σαν έναν αυτόνομο και ανεξάρτητο σκληρό δίσκο. Η υλοποίηση πρέπει να γίνει αφαιρετικά, έτσι ώστε το επίπεδο εφαρμογών να χρησιμοποιεί ήδη υπάρχουσες μεθόδους για να προσπελάσει τα δεδομένα, ίδιες με αυτές που χρησιμοποιεί για να προσπελάσει τους πραγματικούς σκληρούς δίσκους του συστήματος. Τελικά, μετά την υλοποίηση, οι εφαρμογές αλλά και το λειτουργικό σύστημα δεν θα έχουν τη δυνατότητα να διαχωρίσουν το μέρος της μνήμης του προσαρμογέα, από τους υπόλοιπους δίσκους του συστήματος και θα το αντιμετωπίζουν σαν άλλον έναν από αυτούς.

Αυτό σημαίνει ότι θα μπορούν να επιτελέσουν πάνω του και όλες τις αντίστοιχες λειτουργίες. Θα μπορούν να διαβάσουν δεδομένα (`read`), να γράψουν (`write`), να κάνουν ενέργειες ελέγχου και να ζητήσουν πληροφορίες (`ioctl`), θα μπορεί να εγκατασταθεί στη μνήμη ένα ολοκληρωμένο σύστημα αρχείων (filesystem), θα μπορεί να διαμεριστεί (partitioning) και θα έχει τη δυνατότητα να φορτωθεί ώς μέρος του κεντρικού δένδρου αρχείων του συστήματος (mount).

Για να γίνουν όλα αυτά δυνατά, το πρώτο στο οποίο πρέπει να καταλήξουμε είναι το υλικό και το λογισμικό που θα χρησιμοποιήσουμε. Επίσης πρέπει να επιλέξουμε τα χαρακτηριστικά και την αρχιτεκτονική της πλατφόρμας δοκιμών που ταιριάζει στις ανάγκες μας και πάνω στην οποία καλούμαστε να υλοποιήσουμε.



Σχήμα 3.1: *Αναπαράσταση Λειτουργίας Συσκευής MyriBLK*

3.2 Αρχικές επιλογές σχεδίασης

Όπως είδαμε στην εισαγωγή, το λειτουργικό σύστημα με το οποίο θα εργαστούμε είναι της οικογένειας UNIX και ο πυρήνας που τρέχει είναι ο Linux. Από τα διάφορα υποσυστήματα του πυρήνα, θα επικεντρωθούμε στο επίπεδο block (block layer) και θα δούμε πώς αυτό άλληλεπιδρά με τα υπόλοιπα υποσυστήματα, αλλά και τις διεργασίες του επιπέδου εφαρμογών. Πιο συγκεκριμένα θα χρειαστεί να σχεδιάσουμε εξ' ολοκλήρου έναν οδηγό συσκευής block, ο οποίος θα τρέχει στον πυρήνα και θα υλοποιεί όλες τις block λειτουργίες που αναμένουν από αυτόν, τόσο το λειτουργικό όσο και οι εφαρμογές. Ο οδηγός αυτός θα πρέπει να μεταφράζει τις αιτήσεις των παραπάνω επιπέδων, σε ενέργειες που πρέπει να επιτελεστούν στον προσαρμογέα δικτύου. Θα πρέπει δηλαδή μεταξύ άλλων να είναι σε θέση να ικανοποιήσει οποιοδήποτε request από τον πυρήνα, αφού πρώτα συνεργαστεί επιτυχώς με τον προσαρμογέα δικτύου.

Από την άλλη πλευρά, από τη στιγμή που ο οδηγός block που καλούμαστε να σχεδιάσουμε εμφανίζει ώς σκληρό δίσκο ένα μέρος μνήμης που βρίσκεται πάνω σε έναν

προσαρμογέα δικτύου, προκύπτει το θέμα επιλογής του προσαρμογέα αυτού. Ο προσαρμογέας εκτός από την μνήμη πρέπει να διαθέτει και δυνατότητα προγραμματισμού έως ένα βαθμό, για να γίνει δυνατή, τόσο η διαχείρηση της μνήμης που ενσωματώνει, όσο και η επικοινωνία με τον οδηγό που θα συνεργάζεται. Ο προσαρμογέας δικτύου υψηλής επίδοσης, προγραμματιζόμενος και με δική του αποκλειστική μνήμη που θα χρησιμοποιήσουμε είναι ο Myrinet-2000 της εταιρείας Myricom. Όπως παρουσιάσαμε στο προηγούμενο κεφάλαιο, ο προσαρμογέας αυτός διαθέτει έναν μικροεπεξεργαστή RISC, τον Lanai, μία μνήμη SRAM μεγέθους 2MBs, τρείς μηχανές DMA και συνεργάζεται με το υπόλοιπο σύστημα μέσω του διαύλου επικοινωνίας PCI. Αυτά τα χαρακτηριστικά θα προσπαθήσουμε να εκμεταλευτούμε με τον αποδοτικότερο τρόπο κατά το σχεδιασμό για να πετύχουμε τελικά μία άρτια υλοποίηση. Επιλέγουμε να χωρίσουμε τη σχεδίασή μας σε τρία μέρη:

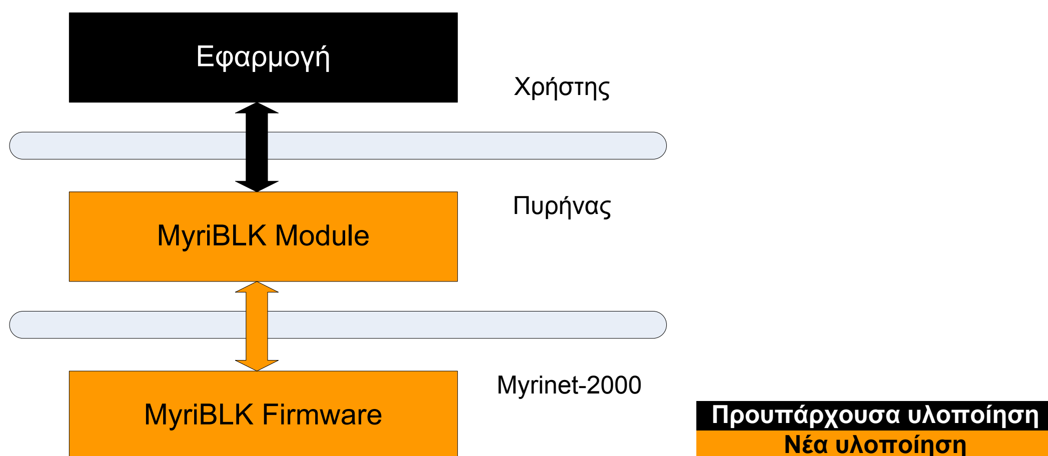
Σχεδίαση από την πλευρά του κόμβου (host) Στο μέρος αυτό σχεδιάζουμε έναν block οδηγό που θα πρέπει να τρέχει ώς μέρος του πυρήνα και να είναι υπεύθυνος για την εξυπηρέτηση οποιασδήποτε αίτησης από την πλευρά του λειτουργικού. Οι αιτήσεις αυτές γίνονται συνήθως μέσω κλήσεων συστήματος που παράγουν οι εφαρμογές και τις κλήσεις αυτές μεταφέρει με τη σειρά του ο πυρήνας στον οδηγό. Αυτό σημαίνει ότι ο οδηγός αντιλαμβάνεται οποιαδήποτε αίτηση σχετίζεται με το επίπεδο block και ταυτόχρονα επεξεργάζεται με κατάλληλο τρόπο τις αιτήσεις αυτές για να τις παραδώσει με τη σειρά του στον προσαρμογέα δικτύου. Επιπροσθέτως, είναι υπεύθυνος και για την ενημέρωση των παραπάνω επιπέδων όταν ο προσαρμογέας ολοκληρώσει (επιτυχώς ή ανεπιτυχώς) το δικό του κομμάτι εργασίας.

Σχεδίαση από την πλευρά του προσαρμογέα δικτύου Στο μέρος αυτό τροποποιούμε κατάλληλα το υλικολογισμικό που εκτελείται στον μικροεπεξεργαστή του προσαρμογέα δικτύου για να μπορεί να ικανοποιήσει τις αιτήσεις που δέχεται από τον οδηγό block. Αυτό σημαίνει ότι το υλικολογισμικό θα πρέπει να οργανώσει κατάλληλα τη μνήμη SRAM, θα μπορεί να διαβάσει και να στείλει δεδομένα, θα έχει τη δυνατότητα να ελέγξει τυχόν λάθη, θα χρησιμοποιεί σωστά τις μηχανές DMA και τελικά θα ενημερώνει τον οδηγό για κάθε πιθανό αποτέλεσμα των ενεργειών του. Φυσικά όλα αυτά πρέπει να γίνουν τροποποιώντας μεν το υλικολογισμικό (firmware ή MCP) σε πολύ μεγάλο βαθμό, κρατώντας δε λειτουργικότητες που

είναι ήδη υλοποιημένες και θα μας χρειαστούν π.χ. διαχείρηση του διαύλου επικοινωνίας PCI.

Επικοινωνία των δύο πλευρών (Host - NIC) και συγχρονισμός Το τρίτο μέρος, αναφέρεται τελευταίο αλλά καθόλα ασήμαντο δεν μπορεί να χαρακτηριστεί, αφού αποτελεί τον συνδετικό κρίκο των δύο προηγούμενων μερών. Εδώ έχουμε να κάνουμε με την καθεαυτή επικοινωνία και συγχρονισμό του οδηγού με τον προσαρμογέα. Στο μέρος αυτό θα δούμε με ποιούς τρόπους γίνονται τα παραπάνω εφικτά, ποιά τα πλεονεκτήματα και τα μειονεκτήματα καθενός και τελικά θα σχεδιάσουμε ένα απλό πρωτόκολλο επικοινωνίας που θα καλύπτει τις απαιτήσεις που θα θέσουμε. Για να φτάσουμε στο σημείο σχεδιασμού αυτού του μέρους χρειάζεται μία πλήρης κατανόηση της λειτουργίας των δύο προηγούμενων και επίσης στο σημείο αυτό θα διαφανούν έντονα (και στη συνέχεια θα τεθούν επί του πρακτέου) θέματα επίδοσης.

Έχοντας έτσι κάνει μία γενική περιγραφή του πλαισίου στο οποίο θα κινηθεί η σχεδίασή μας, ακολουθούν τα τρία κυρίως μέρη:



Σχήμα 3.2: *Τρία Μέρη Σχεδιασμού*

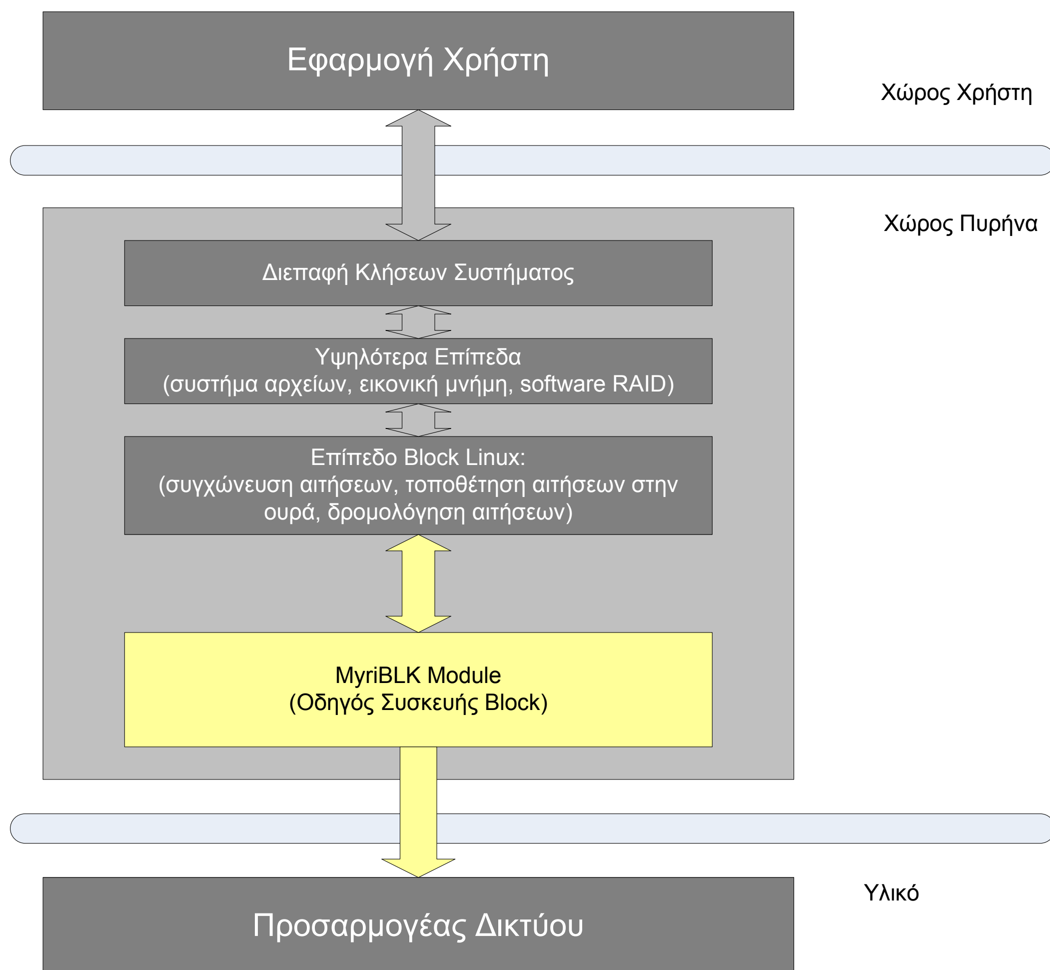
3.3 Σχεδίαση από την πλευρά του κόμβου (host)

Με την ενότητα αυτή προχωρούμε στην κυρίως σχεδίαση του αντικειμένου της εργασίας. Από την πλευρά του κόμβου χρειαζόμαστε μία συσκευή block που να μπορεί να

ικανοποιήσει αιτήσεις του συστήματος που αφορούν block Ε/Ε δεδομένων. Πρέπει λοιπόν, να υλοποιήσουμε έναν πλήρη οδηγό block που να τρέχει ώς μέρος του πυρήνα. Με τον τρόπο αυτό, οι διεργασίες επιπέδου χρήστη παράγουν αρχικά τις αιτήσεις Ε/Ε δεδομένων. Οι αιτήσεις αυτές προωθούνται στον πυρήνα μέσω κλήσεων συστήματος και στη συνέχεια ο πυρήνας, αφού περάσει από κάποια υψηλότερα επίπεδα (VFS, σύστημα αρχείων, block layer), τελικά τις κατευθύνει στον οδηγό block έχοντας απομονώσει τις μεταβλητές που του είνα αναγκαίες. Από το σημείο εκείνο υπεύθυνος είνα πλέον ο οδηγός, που καλούμαστε να υλοποιήσουμε, για να τις χειριστεί και να επιστρέψει στον πυρήνα τα κατάλληλα αποτελέσματα.

Ο οδηγός αυτός σχεδιάζεται και υλοποιείται στη συνέχεια ώς ένα τμήμα δυναμικής φόρτωσης στον πυρήνα (kernel module). Ως kernel module αποτελεί μέρος του πυρήνα και φυσικά εκτελείται σε χώρο πυρήνα. Για το λόγο αυτό, κάθε λειτουργία και συνάρτηση που θα υλοποιηθεί μέσα σε αυτόν, διέπεται από τους βασικούς κανόνες αλλά και υφίσταται τους αντίστοιχους περιορισμούς του χώρου πυρήνα που αναφέρθηκαν στο προηγούμενο κεφάλαιο.

Όπως φαίνεται και από το Σχ. 3.3 ο οδηγός block συνδέει δύο άκρα. Το ένα άκρο είναι η διεπαφή με το επίπεδο block του πυρήνα και το άλλο άκρο είναι το υλικό μας, δηλαδή ο προσαρμογέας δικτύου. Ο οδηγός δέχεται την αίτηση από την πρώτη διεπαφή, την επεξεργάζεται κατάλληλα και την παραδίδει στη δεύτερη διεπαφή (προσαρμογέα δικτύου) που στην προκειμένη περίπτωση είναι ο Lanai. Σύμφωνα με αυτή τη δομή ο οδηγός μπορεί να χωριστεί σε δύο μέρη. Το πρώτο μέρος είναι η επικοινωνία πυρήνα Linux (για την ακρίβεια στρώματος block) με τον οδηγό block και το δεύτερο μέρος είναι η επικοινωνία του οδηγού block με τον προσαρμογέα δικτύου (για την ακρίβεια τον Lanai). Οι δύο αυτές λειτουργίες είναι ουσιαστικά ανεξάρτητες μεταξύ τους, αφού υπονοούν δύο διαφορετικές μετακινήσεις δεδομένων προς δύο διαφορετικές κατευθύνσεις. Για το λόγο αυτό κάθε περίπτωση θα εξεταστεί χωριστά αμέσως παρακάτω. Πριν όμως προχωρήσουμε στη σχεδίαση των δύο αυτών μερών να σημειώσουμε κάποια σημαντικά ζητήματα. Αρχικά, η σειρά με την οποία αναφέρονται τα δύο μέρη δεν καταδεικνύει και τη μοναδική σειρά ροής των δεδομένων. Είναι φυσικό ότι η ροή δεδομένων είναι αμφίδρομη και όχι μόνο από τον πυρήνα Linux προς τον προσαρμογέα αλλά και προς την αντίθετη κατεύθυνση, κάτι που συμβαίνει κάθε φορά που η επεξεργασία από την πλευρά του προσαρμογέα δικτύου, ολοκληρωθεί και πρέπει να ενημερωθούν τα ανώτερα στρώματα. Επίσης, ακόμη και κατά τη ροή των δεδομένων προς μία κατεύθυνση



Σχήμα 3.3: *Σχεδίαση από την πλευρά του κόμβου*

(π.χ. από τον πυρήνα στον προσαρμογέα), κανείς δεν εγγυάται ότι τα δύο μέρη εναλλάσσονται κυκλικά ή σειριακά.

3.3.1 Επικοινωνία πυρήνα Linux με οδηγό block

Στην υποενότητα αυτή θα καταπιαστούμε με το πρώτο μέρος της επικοινωνίας του οδηγού μας. Θα μελετήσουμε τις απαιτήσεις της διεπαφής που υλοποιεί ο πυρήνας Linux για να έρχεται σε επαφή με όλους τους οδηγούς block του συστήματος και θα περιγράψουμε τις λειτουργίες που πρέπει να υποστηρίζει ο οδηγός μας για να ικανοποιήσει τις απαιτήσεις αυτές. Στο συγκεκριμένο σημείο όπως είναι φυσικό δεν μπορούμε να έχουμε σημαντικές πρωτοβουλίες σχεδίασης, αφού η διεπαφή με τον πυρήνα είναι καθορισμένη, κάτι που έχει όμως νόημα, γιατί μόνο έτσι πετυχαίνουμε την επιθυμητή

αφαίρεση και διαστρωμάτωση.

Στο κεφάλαιο 2 είδαμε πως ο πρωταρχικός σκοπός ενός οδηγού block είναι να μεταφέρει δεδομένα από και προς μία συσκευή block (στην πλειονότητα των περιπτώσεων μία μονάδα δίσκου).Οι δύο κύριες λειτουργίες δηλαδή που επιτελεί είναι η εγγραφή των δεδομένων αυτών στο δίσκο ή η ανάγνωσή τους από αυτόν. Στην πρώτη περίπτωση, τα δεδομένα του τα παρέχει ο πυρήνας και του υποδεικνύει την ακριβή τους τοποθέτηση στο δίσκο, ενώ στη δεύτερη του παρέχει τη σωστή θέση και περιμένει να τα παραλάβει. Η διεπαφή αυτή με τον πυρήνα υλοποιείται όπως είδαμε με ένα σύνολο συναρτήσεων αλλά και με ειδικά διαμορφωμένες δομές που προυπάρχουν στον πυρήνα και αποτελούν μέρος της υλοποίησης του επιπέδου block του Linux (linux block layer).

Έτσι ο οδηγός καλείται να αλληλεπιδράσει με τη διεπαφή του linux block layer σε τρία επίπεδα: πρώτον με τις μεθόδους της δομής `block_device_operations` για την υλοποίηση των αναγκαίων κλήσεων συστήματος, δεύτερον με τη δομή `gendisk` για να αναπαραστήσει στον πυρήνα την μονάδα δίσκου την οποία διαχειρίζεται, τρίτον με την ουρά αιτήσεων η οποία θα τον φέρει σε επαφή με τις πραγματικές αιτήσεις δεδομένων του συστήματος.

Τρεις μεθόδους επιβάλλεται να ορίσει ο οδηγός μας που θα υλοποιούν τις αντίστοιχες κλήσεις συστήματος. Αυτές είναι οι: `open`, `release`, `ioctl`. Με αυτές πρέπει να είναι σε θέση να διαχειριστεί το άνοιγμα και κλείσιμο της συσκευής, όπως και την άμεση κατεύθυνση εντολών προς αυτήν μέσω της `ioctl`. Στη συγκεκριμένη εργασία, ο οδηγός block δεν διαχειρίζεται έναν πραγματικό σκληρό δίσκο που περιλαμβάνει μηχανικά μέρη. Ο σχεδιασμός βρίσκεται σε αντιστοιχία με αυτόν που θα κάναμε για έναν δίσκο στερεάς κατάστασης (SSD). Ο οδηγός από την πλευρά του βλέπει ένα συνεχόμενο χώρο αποθήκευσης μίας μνήμης SRAM. Για το λόγο αυτό, από τη στιγμή που γίνουν οι κατάλληλες αρχικοποιήσεις κατά την εισαγωγή του module στον πυρήνα, δεν έχει λόγο να προβεί σε ενέργειες διαχείρισης κάθε φορά που ανοίγει ή κλείνει η συσκευή. Αντίθετα σε έναν παραδοσιακό σκληρό δίσκο θα έπρεπε να ελέγξει ότι ο δίσκος δουλεύει, να ξεκινήσει ή να σταματήσει την περιστροφή του κτλ. Η μέθοδος `ioctl` δεν είναι επίσης αναγκαία, επειδή δεν υπάρχουν εξειδικευμένες απαιτήσεις για έλεγχο του φυσικού υλικού, αφού πρόκειται απλά για μία μνήμη SRAM. Παρόλα αυτά, θα υλοποιήσουμε ένα μέρος της, που αναφέρεται στα γεωμετρικά χαρακτηριστικά ενός φυσικού δίσκου, παρουσιάζοντας αντίστοιχα εικονικά (sectors, tracks, cylinders), που συμφω-

νούν όμως με τη πραγματική χωρητικότητα της φυσικής μας μνήμης. Αυτό το κάνουμε για λόγους πληρότητας και συμβατότητας, επειδή υπάρχουν ακόμη εφαρμογές χώρου χρήστη, όπως είναι το εργαλείο fdisk, που εξαρτώνται από πληροφορίες κυλίνδρων και δεν λειτουργούν σωστά αν οι πληροφορίες αυτές δεν είναι διαθέσιμες από τη συσκευή (ο πυρήνας δεν ενδιαφέρεται για γεωμετρία, αφού αντιμετωπίζει κάθε συσκευή block σαν ένα συνεχόμενο πίνακα από sectors). Τέλος, όπως είναι ξεκάθαρο οι μέθοδοι `media_changed` καιrevalidate_disk` που αναφέραμε σε προηγούμενο κεφάλαιο, δεν αντικατοπτρίζουν καμία φυσική ενέργεια στο υλικό με το οποίο ασχολούμαστε, γι' αυτό και δεν θα υλοποιηθούν.

Η δομή `gendisk` αποτελεί την αναπαράσταση μίας μονάδας δίσκου στον πυρήνα. Ο κύριος ρόλος του οδηγού block που σχεδιάζουμε είναι να αναπαραστίσει τη μνήμη SRAM του προσαρμογέα δικτύου σαν μονάδα δίσκου στο σύστημα. Για το λόγο αυτό, πρέπει να ορίσει κατάλληλα μία δομή `gendisk`. Κάθε φορά που ο πυρήνας θα αναφέρεται σε αυτή τη δομή, όλες οι ενέργειες θα καθρεφτίζονται στην SRAM του προσαρμογέα. Έτσι ο οδηγός δημιουργεί και στη συνέχεια θέτει όλες τις μεταβλητές και τα πεδία μίας τέτοιας δομής. Αυτά έχουν να κάνουν με μία πλειάδα χαρακτηριστικών της μονάδας δίσκου. Μερικά από αυτά είναι το όνομα του δίσκου ("`myriblk`"), η χωρητικότητά του, η δομή μεθόδων block που υλοποιεί τις κλήσεις συστήματος και φυσικά η ουρά αιτήσεων Ε/Ε που θα χρησιμοποιήσει ο πυρήνας για την συγκεκριμένη μονάδα δίσκου. Το τελευταίο αυτό πεδίο είναι και το πιο σημαντικό, γιατί συνδέει τη συσκευή που διαχειριζόμαστε με τον μηχανισμό αιτήσεων Ε/Ε block του πυρήνα.

Αφού έχουμε συνδεθεί με τις αιτήσεις Ε/Ε του πυρήνα, ορίζοντας μία ουρά αιτήσεων, ο οδηγός μας έχει εξασφαλίσει εισερχόμενες αιτήσεις από τον πυρήνα. Αυτό που υπολοίπεται είναι να τις ικανοποιήσει. Φτάνουμε έτσι στο σχεδιασμό της κύριας συνάρτησης κάθε οδηγού block, που είναι η συνάρτηση request. Όπως είδαμε και κατά την περιγραφή των θεωρητικών στοιχείων η συνάρτηση request μπορεί να είναι από σχετικά απλή, μέχρι εξαιρετικά πολύπλοκη ανάλογα με τις απαιτήσεις του υλικού και κυρίως τις απαιτήσεις επίδοσης. Στο πλαίσιο της εργασίας αυτής δεν έχουμε ως μοναδική μέριμνα την επίδοση, ούτε επικεντρωνόμαστε μόνο σε αυτή. Αντίθετα, διεξάγουμε μία μελέτη περίπτωσης για να εξακριβώσουμε αν μπορούμε τελικά να πετύχουμε τη συγκεκριμένη υλοποίηση, να την αξιολογήσουμε πειραματικά, να κάνουμε μετέπειτα υποθέσεις σχετικές με τη διαφορετική αρχιτεκτονική που παρουσιάζει και να καταλήξουμε σε χρήσιμα συμπεράσματα. Φυσικά στοχεύουμε σε μία υψηλή επίδοση, η οποία όμως θέλουμε να

προκύψει κυρίως από το διαφορετικό υλικό και αρχιτεκτονική που επιλέξαμε. Για το λόγο αυτό, δεν επιλέγουμε αρχικά υπερβολικά σύνθετη μέθοδο request αν και η πολυπλοκότητα θα αυξηθεί (όπως θα διαπιστώσουμε στη συνέχεια) κατά την υλοποίηση αποδοτικού πρωτοκόλλου επικοινωνίας. Έχοντας υπ’ όψιν τα προαναφερθέντα, καταλήγουμε σε μία συνάρτηση request που θα ικανοποιεί μία αίτηση κάθε φορά και δεν θα συνεχίζει στην επόμενη αν δεν έχει ολοκληρωθεί πλήρως η προηγούμενη. Αυτό έχει ώς αποτέλεσμα να ικανοποιούνται σειριακά όλες οι αιτήσεις που αναμένουν στην ουρά αιτήσεων. Τέλος, κάθε μεμονωμένη αίτηση θα περιέχει ένα φυσικό τμήμα (physical segment).

3.3.2 Επικοινωνία οδηγού block με τον προσαρμογέα δικτύου

Εδώ θα ασχοληθούμε με τη δεύτερη διεπαφή επικοινωνίας του οδηγού. Με τη συνάρτηση request υλοποιημένη, ο οδηγός έχει πλέον στη διάθεση του τις αιτήσεις Ε/Ε block του συστήματος. Αυτές περιέχουν τα ακριβή sectors που πρέπει να μεταφερθούν. Να θυμίσουμε, οτι ανεξάρτητα από την προέλευση των sectors αυτών, τα sectors που φτάνουν στον οδηγό είναι συνεχόμενα πάνω στο δίσκο (όχι απαραίτητα και στη μνήμη του συστήματος). Επίσης η αίτηση δηλώνει αν πρόκειται για λειτουργία ανάγνωσης ή εγγραφής. Στην περίπτωση ανάγνωσης η αίτηση περιέχει τη θέση μνήμης στην οποία πρέπει να μεταφερθούν τα δεδομένα, ενώ στην περίπτωση εγγραφής, τη θέση μνήμης από την οποία θα μεταφερθούν. Έτσι ο οδηγός έχει στη διάθεσή του τέσσερις παραμέτρους, που προέρχονται από τη δομή request, και είναι αρκετοί για οποιαδήποτε μεταφορά δεδομένων:

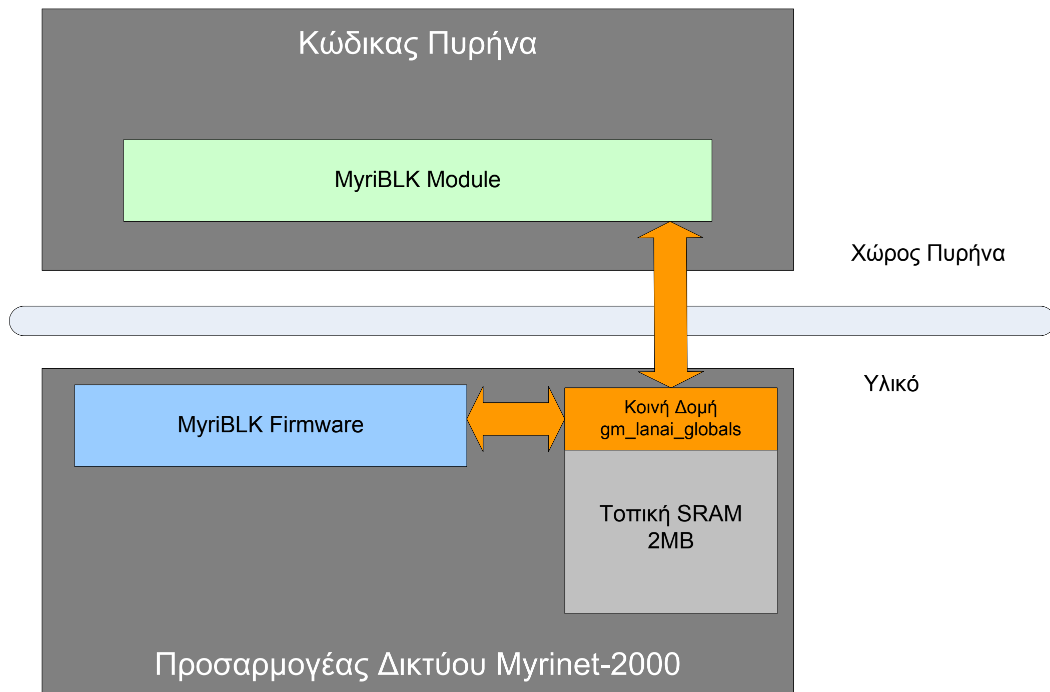
`rq_data_dir(req)`: ανάλογα με την τιμή έχουμε εγγραφή ή ανάγνωση

`buffer`: θέση μνήμης από/προς την οποία θα γίνει η εγγραφή/ανάγνωση

`sector`: το πρώτο sector από το οποίο θα ξεκινήσει η μεταφορά

`current_nr_sectors`: ο συνολικός αριθμός sectors που θα μεταφερθούν

Χρησιμοποιώντας τις τέσσερις αυτές παραμέτρους ο οδηγός καλείται να επικοινωνήσει με τον προσαρμογέα δικτύου και να του υποβάλλει το αντίστοιχο αίτημα δεδομένων. Για την επικοινωνία του οδηγού με τον προσαρμογέα δικτύου, θα εκμεταλλευτούμε



Σχήμα 3.4: *Κοινή Δομή Επικοινωνίας*

ένα μηχανισμό που χρησιμοποιεί το λογισμικό GM. Σύμφωνα με αυτόν τον μηχανισμό, υπάρχει μία δομή δεδομένων που αποθηκεύεται στην μνήμη SRAM του προσαρμογέα κατά την αρχικοποίηση της συσκευής και είναι κοινή τόσο για τον κώδικα που τρέχει το module, όσο και για τον κώδικα του firmware (Σχ. 3.4). Με αυτόν τον τρόπο καταφέρνει το GM να περνά μηνύματα ελέγχου από τον κώδικα του πυρήνα, στον κώδικα που τρέχει πάνω στον Lanai του προσαρμογέα και αντίστροφα. Αυτό καθίσταται εφικτό, αποθηκεύοντας μεταβλητές στη συγκεκριμένη δομή, οι οποίες στη συνέχεια προσπελαύνονται από κάθε άκρο ξεχωριστά. Ακολουθώντας τις μεθόδους αυτού του μηχανισμού επικοινωνίας, θα ορίσουμε δικές μας μεταβλητές στη δομή αυτή και αφού επεξεργαστούμε (όπου είναι αναγκαίο) τις παραπάνω παραμέτρους θα τις θέσουμε ώς τιμές στις προηγούμενα ορισμένες μεταβλητές. Από εκείνη τη στιγμή, οι παράμετροι γίνονται άμεσα διαθέσιμες στο υλικολογισμικό, που βαραίνεται και με την καθεαυτή μεταφορά.

Παρατηρούμε ότι με τον παραπάνω σχεδιασμό, ο οδηγός του συστήματος δεν επιβαρύνεται με την ουσιαστική μεταφορά δεδομένων. Δηλώνει απλά στο υλικολογισμικό τα ακριβή δεδομένα που χρειάζεται να μεταφερθούν και από τη στιγμή εκείνη μπορεί η CPU του συστήματος να αναλωθεί σε διαφορετικές εργασίες. Με τη μεταφορά δεδο-

μένων θα ασχοληθεί αποκλειστικά ο προσαρμογέας δικτύου, όπως θα δούμε αναλυτικά στη συνέχεια. Ο κώδικας του οδηγού θα χρειαστεί να επέμβει μόνο όταν η μεταφορά ολοκληρωθεί και ενημερωθεί γι' αυτό από το υλικολογισμικό. Με αυτή την εκφόρτωση (offload) εργασίας από τη CPU προς τον προσαρμογέα ωφελούμαστε σε μεγάλο βαθμό σε θέματα απόδοσης λόγω παραλληλίας ενεργειών. Ταυτόχρονα όμως, προκύπτει το πρόβλημα συγχρονισμού των δύο παραπάνω μερών που περιγράψαμε (κώδικας οδηγού με κώδικα υλικολογισμικού), το οποίο θα εξεταστεί σε μετέπειτα στάδιο της ανάλυσης.

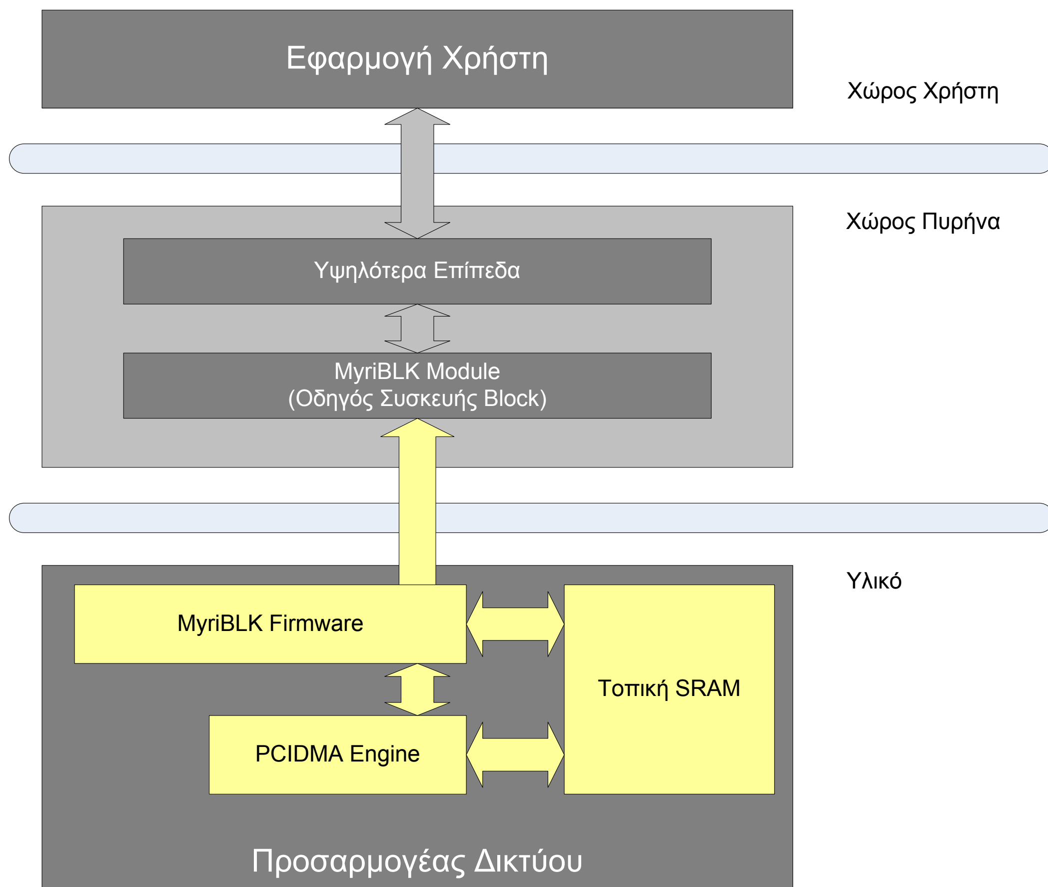
3.4 Σχεδίαση από την πλευρά του προσαρμογέα δικτύου

Από την πλευρά του προσαρμογέα δικτύου χρειαζόμαστε ένα μηχανισμό, ο οποίος λαμβάνει μία αίτηση δεδομένων από τον οδηγό, προσπελάζει την τοπική μνήμη SRAM και στη συνέχεια εκτελεί την επιθυμητή ενέργεια. Ταυτόχρονα, όπως είδαμε και κατά το σχεδιασμό του οδηγού, θέλουμε αυτή η πλευρά να επιβαρυνθεί με το κόστος της πραγματικής μεταφοράς δεδομένων. Αυτό καθίσταται δυνατό από το γεγονός ότι ο προσαρμογέας που χρησιμοποιούμε είναι εφοδιασμένος με τρεις μηχανές DMA (από τις οποίες θα χρησιμοποιήσουμε τη μία για τη μεταφορά). Επομένως, ο προσαρμογέας, και συγκεκριμένα το υλικολογισμικό που εκτελείται στον Lanai, πρέπει να λάβει τις τέσσερις παραμέτρους που συναντήσαμε παραπάνω (μαζί με τυχόν μεταβλητές ελέγχου), να τις μεταφράσει σωστά, να αναλάβει τη μεταφορά δεδομένων από/προς τη κεντρική μνήμη προς/από την τοπική SRAM, να ενημερώσει τον οδηγό για την επιτυχία ή αποτυχία της μεταφοράς και στη συνέχεια να επαναλάβει τη διαδικασία για επόμενη αίτηση.

Όπως και στον οδηγό έτσι και στο υλικολογισμικό θα χρησιμοποιήσουμε μηχανισμούς που ορίζει το GM. Για να επωφεληθούμε όμως από αυτό χρειάζεται κατανόηση της λειτουργίας του GM και από τις δύο πλευρές, ώστε να σχεδιάσουμε με τις λιγότερες δυνατές τροποποιήσεις και παράλληλα να εκμεταλλευτούμε στο μέγιστο δυνατό βαθμό τη λειτουργικότητα και τους μηχανισμούς που προσφέρει. Διεξοδική ανάλυση του περιβάλλοντος GM και του τρόπου με τον οποίο υλοποιείται, θα δούμε στο επόμενο κεφάλαιο που ασχολούμαστε με την υλοποίηση. Παρόλα αυτά, εδώ θα περιγράψουμε τις βασικές σχεδιαστικές τεχνικές που ακολουθεί το GM, και θα αποφασίσουμε ποιές πρέπει να τροποποιηθούν και με ποιόν τρόπο, όπως επίσης και με ποιες συμφέρει να

συμβαδίσουμε.

Θα χωρίσουμε την ανάλυση σε δύο μέρη. Στο πρώτο θα παρατηρήσουμε, πώς οργανώνει την τοπική μνήμη SRAM το GM και ποιές είναι οι βασικές δομές που εδράζονται πάνω σε αυτή. Στο δεύτερο, θα ασχοληθούμε με τους μηχανισμούς μετακίνησης δεδομένων του GM καθώς και την τελική επικοινωνία του προσαρμογέα με τον οδηγό, μετά το πέρας μίας επιτυχούς ή ανεπιτυχούς μετακίνησης. Σε κάθε μέρος θα κάνουμε τις αντίστοιχες επιλογές που απαιτούνται για την υλοποίηση της μονάδας δίσκου που καλούμαστε να προσομοιώσουμε.



Σχήμα 3.5: *Σχεδίαση από την πλευρά του προσαρμογέα δικτύου*

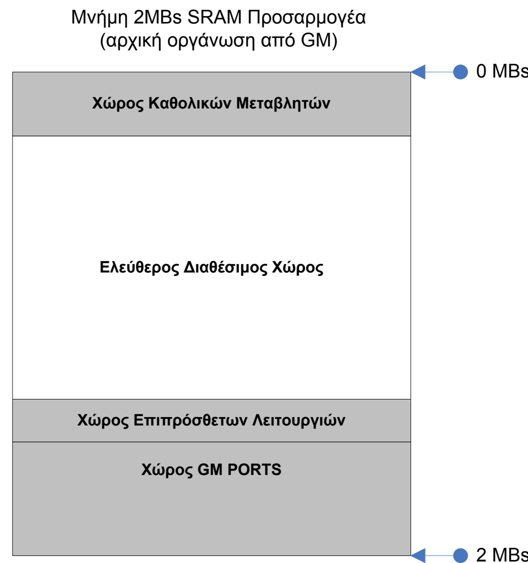
3.4.1 Οργάνωση μνήμης SRAM και βασικές περιοχές

Όπως αναλύεται διεξοδικά στην Εν. 2.2.2, η τοπική ενσωματωμένη μνήμη του προσαργογέα δικτύου που θα χρησιμοποιήσουμε είναι τύπου SRAM και έχει μέγεθος 2MBs.

Κατά την αρχικοποιήση του GM, πρώτα απ’ όλα φορτώνεται το GM module στον πυρήνα. Το module με τη σειρά του φορτώνει τον κώδικα του υλικολογισμικού στον Lanai, ο οποίος μηδενίζει αρχικά όλη την SRAM. Μετά την χωρίζει σε καθορισμένες περιοχές οι οποίες επιτελούν συγκεκριμένες λειτουργίες. Τελικά, απομένει ένας ελεύθερος χώρος μνήμης, ο οποίος προορίζεται για γενική χρήση όταν θα ξεκινήσει η μεταφορά πακέτων από και πρός το υπόλοιπο δίκτυο.

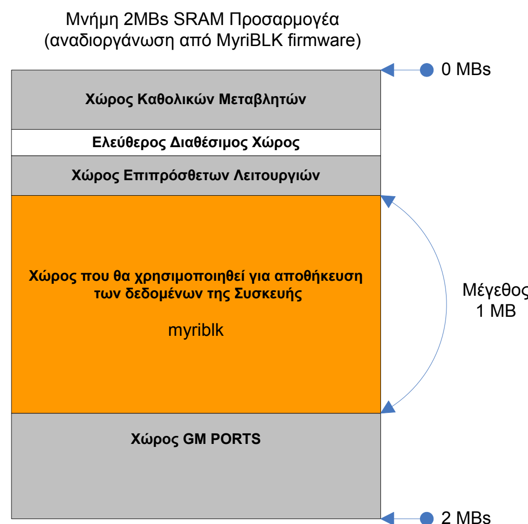
Σκοπός μας είναι να αναδιοργανώσουμε με τέτοιο τρόπο την μνήμη, ώστε να μας επιτραπεί να δεσμεύσουμε και μία ακόμη περιοχή κατά την αρχικοποίηση, η οποία θα αποτελέσει το χώρο πάνω στον οποίο θα προσομοιώσουμε την μονάδα δίσκου μας. Με αυτόν τον τρόπο κρατούμε ανέπαφη, όλη τη διεπαφή που έχει το GM module με το GM firmware (MCP). Αυτό μας παρέχει τη δυνατότητα να χρησιμοποιήσουμε στη συνέχεια ήδη υλοποιημένες μεθόδους επικοινωνίας για λογαριασμό της συσκευής myriblk. Επίσης με τέτοιου είδους υλοποίηση, η συσκευή myriblk αποτελεί επιπρόσθετο χαρακτηριστικό του συστήματος του προσαρμογέα, χωρίς να ακυρώνεται καμία προηγούμενη λειτουργικότητά του. Αυτή η προσέγγιση μας ευνοεί φυσικά, αλλά παράλληλα προξενεί νέες δυσκολίες σε θέματα υλοποίησης. Αυτό είναι προφανές, αφού μπαίνουμε σε μία διαδικασία, κατά την οποία πρέπει μεν να τροποποιήσουμε την οργάνωση μνήμης, με καθόλου τυχαίο τρόπο δε, γιατί η οποιαδήποτε τροποποίηση επιβάλλεται να είναι αόρατη για το υπόλοιπο GM. Ουσιαστικά θέλουμε να εξαφανίσουμε ένα κομμάτι της ήδη υπάρχουσας μνήμης, με απόλυτα διαφανή τρόπο προς το υπόλοιπο σύστημα, το οποίο θα είναι προσβάσιμο μόνο από τον οδηγό block που θα υλοποιήσουμε. Ο ακριβής τρόπος που το πετυχαίνουμε δεν θα περιγραφεί εδώ, αλλά στο κεφάλαιο της υλοποίησής. Παρόλα αυτά παρουσιάζουμε την γενική σχεδιαστική ιδέα. Η αρχική οργάνωση της SRAM όπως καθορίζεται από το GM φαίνεται στο Σχ. 3.6.

Αρχικά η μνήμη είναι γεμάτη μηδενικά (zeroed). Η πρώτη περιοχή που δεσμεύει το GM είναι ο χώρος που θα αποθηκεύσει τις καθολικές μεταβλητές. Πιο συγκεκριμένα ξεκινά από την αρχή της μνήμης SRAM και δεσμεύει χώρο, ίσο με το μέγεθος της δομής κοινών μεταβλητών. Η δομή αυτή αποτελεί την κοινή δομή μεταξύ οδηγού και υλικολογισμικού που συναντήσαμε σε προηγούμενη ενότητα και χρησιμοποιούμε για την επικοινωνία των δύο μερών. Οτιδήποτε αποθηκεύεται στη δομή αυτή είναι προσβάσιμο και από τα δύο μέρη. Γι’ αυτό και η δομή αποθηκεύεται στην αρχή της SRAM ώστε να ξέρουν ακριβώς που θα την βρούν οι αντίστοιχοι κώδικες. Η δεύτερη περιοχή που δεσμεύεται ξεκινά από το τέλος της μνήμης SRAM και εκτείνεται έως το σημείο που



Σχήμα 3.6: *Αρχική Οργάνωση Μνήμης SRAM*

χρειάζεται για να καλυφθεί το σύνολο των ports που έχει τη δυνατότητα να εξυπηρετήσει το GM. Αυτός ο χώρος είναι το μέρος της SRAM που θα αντικατοπτριστεί στη συνέχεια σε διευθύνσεις εικονικής μνήμης των εφαρμογών. Τέλος, από το σημείο που ξεκινούν τα ports και προς τα πάνω, υπάρχει περίπτωση να δεσμευτεί και μία τρίτη περιοχή που εξυπηρετεί επιπρόσθετες λειτουργίες. Τελικά ο χώρος που απομένει αποτελεί την διαθέσιμη μνήμη του προσαρμογέα (`available memory`) που προορίζεται για πάσα χρήση.



Σχήμα 3.7: *Αναδιοργάνωση Μνήμης SRAM από MyriBLK Firmware*

Στη σχεδίασή μας, θα δεσμεύσουμε έναν χώρο μνήμης που δεν θα επηρεάζει τους προη-

γούμενους τρεις, αλλά θα μειώνει την διαθέσιμη μνήμη. Ένα κομμάτι διαθέσιμης μνήμης θα παραμείνει αφιερωμένο στο GM για να μην καταλήξουμε σε σύγκρουση. Μέσα σ' αυτό το χώρο θα προσομοιώσουμε τη μονάδα δίσκου μας και θα είναι ο μοναδικός που θα είναι εμφανής στον οδηγό block. Επιλέγουμε το μέγεθός του περίπου στο 1MB (το ακριβές μέγεθος αλλά και το σημείο στη μνήμη θα καθοριστεί κατά την υλοποίηση, λαμβάνοντας υπ' όψιν όλους τους περιορισμούς που προκύπτουν). Τελικά η μνήμη SRAM θα αναδιοργανωθεί στη μορφή του Σχ. 3.7

3.4.2 Μετακίνηση δεδομένων και επικοινωνία με οδηγό

Στην ενότητα που αφορούσε στη σχεδίαση από την πλευρά του οδηγού, είδαμε πως ο οδηγός έχει αποθηκεύσει στην κοινή δομή τις μεταβλητές που είναι απαραίτητες για την πλήρη μεταφορά δεδομένων. Με αυτόν τον τρόπο, και σε συνδυασμό με την αναδιοργάνωση της μνήμης, έχουμε πετύχει την πρόσβαση στις μεταβλητές αυτές από το υλικολογισμικό, αλλά και την πρόσβαση σε έναν ανεξάρτητο χώρο μνήμης πάνω στη μνήμη SRAM που χρησιμοποιείται αποκλειστικά για τις ανάγκες της block συσκευής μας. Έτσι το υλικολογισμικό βρίσκεται σε θέση να διαχειριστεί κατά βούληση τόσο τις μεταβλητές αυτές, όσο και τον αντίστοιχο χώρο μνήμης που είναι αφιερωμένος στη συσκευή. Έχοντας αυτές τις δύο δυνατότητες, έχει φτάσει η στιγμή που το υλικολογισμικό πρέπει πλέον να πραγματοποιήσει την πραγματική μεταφορά των δεδομένων.

Στο σημείο αυτό, θα φανεί η σημασία του γεγονότος ότι ο σχεδιασμός μας συμβαδίζει έως τώρα πλήρως με τον σχεδιασμό και την υλοποίηση του GM, αφού θα μας επιτρέψει να χρησιμοποιήσουμε ήδη υλοποιημένες μεθόδους για την πραγματική μεταφορά δεδομένων. Με αυτό τον τρόπο κερδίζουμε σε συμβατότητα, αλλά και απόδοση, γιατί δεν χρειάζεται να υλοποιήσουμε από την αρχή μεθόδους που στοχεύουν σε επίδοση, οι οποίες είναι σύνθετες και εμπλέκουν ένα σοβαρό βαθμό πολυπλοκότητας. Για την μεταφορά δεδομένων θα χρησιμοποιήσουμε τη μία από τις τρεις μηχανές DMA του προσαρμογέα, επιτυγχάνοντας μηδενική εμπλοκή του κώδικα του οδηγού, καθώς και της CPU του συστήματος. Το χειρισμό της μηχανής DMA θα τον πραγματοποιήσουμε εκμεταλλευόμενοι τους μηχανισμούς καταστάσεων SDMA και RDMA που περιγράψαμε στο κεφάλαιο 2 (Σχήμα 2.2.2.Ε).

Πιο συγκεκριμένα, θα περιγράψουμε αρχικά την περίπτωση που αναγνωρίζουμε ένα αίτημα εγγραφής από τον οδηγό. Στο αίτημα εγγραφής, πρέπει να γίνει μία μεταφο-

ρά δεδομένων από την κεντρική μνήμη του συστήματος στη μονάδα δίσκου μας, που είναι η μνήμη SRAM του προσαρμογέα. Παρατηρούμε επίσης, ότι κατά την αποστολή ενός πακέτου στην αρχική υλοποίηση του GM καλείται ο μηχανισμός SDMA που εκτός από τις υπόλοιπες λειτουργίες που επιτελεί (και περιγράφηκαν στην ενότητα 2.2.2), ξεκινά ένα `read` DMA για τη μεταφορά των δεδομένων από την κεντρική μνήμη RAM στους κατάλληλους buffers της μνήμης SRAM του προσαρμογέα. Αυτό το συγκεκριμένο κομμάτι λειτουργίας της μηχανής κατάστασης SDMA θα χρησιμοποιήσουμε και στη δική μας υλοποίηση, όταν θα έχουμε αίτημα εγγραφής. Αφού διαβάσουμε τις παραμέτρους από την κοινή δομή μεταβλητών και διαπιστώσουμε πως το αίτημα που μας έχει μεταβιβαστεί αποτελεί αίτημα εγγραφής, θα περάσουμε με τον κατάλληλο τρόπο τις διευθύνσεις από και προς τις οποίες πρέπει να μεταφερθούν τα δεδομένα, στη μηχανή DMA του προσαρμογέα, μέσω τις μηχανής κατάστασης SDMA και της λειτουργίας που προαναφέραμε. Αντίστοιχα, στην περίπτωση που έχουμε αίτημα ανάγνωσης από τη μονάδα δίσκου, θα χρησιμοποιήσουμε την αντίστοιχη λειτουργία της μηχανής κατάστασης RDMA, που μεταφέρει δεδομένα από τη μνήμη SRAM στην κεντρική RAM κατά τη διαδικασία συλλογής των εισερχόμενων πακέτων. Και σε αυτή την περίπτωση θα περάσουμε τις κατάλληλες διευθύνσεις (καθώς και όποια άλλη μεταβλητή είναι αναγκαία) στη μηχανή RDMA, αφού πρώτα τις διαβάσουμε από την κοινή δομή μεταβλητών. Με αυτό το σχεδιασμό, ολοκληρώνουμε την πραγματική μεταφορά δεδομένων με τις ελάχιστες τροποποιήσεις στο υλικολογισμικό, αλλά ταυτόχρονα με τον πιο αποδοτικό τρόπο. Βέβαια, μόνο η σωστή χρήση της μηχανής DMA δεν εγγυάται τη μέγιστη απόδοση, όπως θα φανεί ξεκάθαρα στη συνέχεια.

Ο ρόλος του υλικολογισμικού δεν τελειώνει μετά το πέρας της μεταφοράς, ούτε καν κατά τη διάρκεια αυτής. Καθόλη τη διάρκεια της μεταφοράς αλλά και μετά την ολοκλήρωσή της με επιτυχία ή όχι, το υλικολογισμικό πρέπει να παρέχει τα κατάλληλα σήματα ελέγχου στον οδηγό, έτσι ώστε αυτός να παραμένει ενήμερος για την ακριβή κατάσταση στην οποία βρίσκεται η εργασία στον προσαρμογέα. Τέτοια σήματα περνώνται με τον ίδιο τρόπο επικοινωνίας που χρησιμοποιούσε και ο οδηγός, δηλαδή μέσω της κοινής δομής επικοινωνίας. Τελικά, η τελευταία ενέργεια που κάνει το υλικολογισμικό είναι να ενημερώσει τον οδηγό για το πέρας της μεταφοράς δεδομένων στα οποία αναφερόταν η τελευταία αίτηση και επίσης να του γνωστοποιήσει ότι είναι διαθέσιμο για την επόμενη αίτηση. Αυτό το σημείο της επικοινωνίας, όπως και ο ακριβής συγχρονισμός του κώδικα οδηγού με το υλικολογισμικό (επισημαίνουμε πως πρόκειται

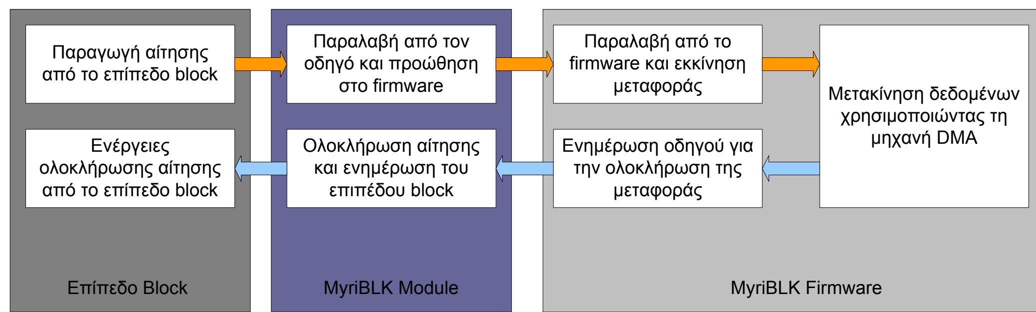
για κώδικες που εκτελούνται σε διαφορετικούς επεξεργαστές: CPU - Lanai) μπορεί να γίνει με διάφορους τρόπους, καθένας από τους οποίους παρέχει διαφορετικά πλεονεκτήματα. Αυτούς τους τρόπους επικοινωνίας αλλά και το κρίσιμο ζήτημα του συγχρονισμού θα αναλύσουμε αμέσως παρακάτω.

3.5 Επικοινωνία των δύο πλευρών (Host - NIC) και συγχρονισμός

Στο σημείο αυτό, έχουμε στη διάθεσή μας δύο διαφορετικά κομμάτια κώδικα, που εκτελούνται σε δύο διαφορετικούς επεξεργαστές, έχουν τη δυνατότητα ανταλλαγής μυνημάτων, και το καθένα είναι πλήρως λειτουργικό όσον αφορά στην εργασία που του έχει ανατεθεί. Το ερώτημα που προκύπτει είναι: πώς αυτά τα δύο μέρη θα επικοινωνήσουν μεταξύ τους αποδοτικά και πώς θα συγχρονιστούν για να έχουμε την επιθυμητή λογική ροή προγράμματος που καλούμαστε να υλοποιήσουμε; Η απάντηση στο ερώτημα αυτό θα δοθεί στο τέλος της ενότητας αυτής, αφού πρώτα μελετήσουμε το πρόβλημα, εξετάσουμε τις μεθόδους που μπορούμε να εφαρμόσουμε και καταλήξουμε στις τελικές επιλογές σχεδίασης που ταιριάζουν στη μελέτη της περίπτωσής μας.

Αρχικά επικεντρωνόμαστε στο πρόβλημα που πρέπει να αντιμετωπίσουμε. Έχοντας σχεδιάσει τη συσκευή μας και από τις δύο πλευρές, καταλήγουμε με ένα κομμάτι κώδικα που εκτελείται στην CPU του συστήματος ως module πυρήνα και ένα κομμάτι κώδικα να εκτελείται στον Lanai ως μέρος του υλικολογισμικού του προσαρμογέα. Επίσης έχουμε σχεδιάσει μία κοινή δομή μέσω της οποίας έχουν τη δυνατότητα τα δύο κομμάτια κώδικα να επικοινωνούν έχοντας πρόσβαση σε κοινές μεταβλητές. Παρόλο που οι λειτουργίες που επιτελούν οι δύο πλευρές είναι εμφανώς διαχωρισμένες και ανεξάρτητες μεταξύ τους, η χρονική σειρά με την οποία πρέπει να εκτελεστούν είναι σαφώς καθορισμένη. Αυτός ο περιορισμός είναι αλληλένδετος με το υποσύστημα block. Όπως φαίνεται και στο Σχ. 3.8 για να ολοκληρωθεί μία αίτηση block με επιτυχία, πρέπει να επιτελεστεί μία αλληλουχία λειτουργιών οι οποίες έχουν σειριακή ροή.

Η αίτηση που παράγει μία διεργασία (συνήθως από το χώρο εφαρμογών) για την συσκευή myriblk, φτάνει μέσω του υποσυστήματος block του πυρήνα, στον οδηγό block που καλούμαστε να υλοποιήσουμε. Σύμφωνα με το σχεδιασμό μας, ο οδηγός block ελέγχει αν ο προσαρμογέας είναι διαθέσιμος και σε αυτή την περίπτωση του προω-



Σχήμα 3.8: *Σειρά ενεργειών για ολοκλήρωση αίτησης block*

θεί την αίτηση. Αφού του προωθήσει την αίτηση, πρέπει να λάβει απάντηση από τον προσαρμογέα και τελικά να ολοκληρώσει την διαδικασία της αίτησης, ενημερώνοντας ταυτόχρονα το υποσύστημα block. Παράλληλα το υλικολογισμικό μόλις λάβει την αίτηση επιβαρύνεται με τη μεταφορά των δεδομένων και την ενημέρωση του οδηγού. Οι δύο αυτές διαφορετικές ενέργειες πρέπει να συγχρονιστούν. Στην παρούσα εργασία σχεδιάσαμε και υλοποιήσαμε δύο μεθόδους συγχρονισμού του κώδικα οδηγού με το υλικολογισμικό. Ο σχεδιασμός των δύο αυτών μεθόδων περιγράφεται παρακάτω.

3.5.1 Συγχρονισμός Περιοδικού Ελέγχου (Polling)

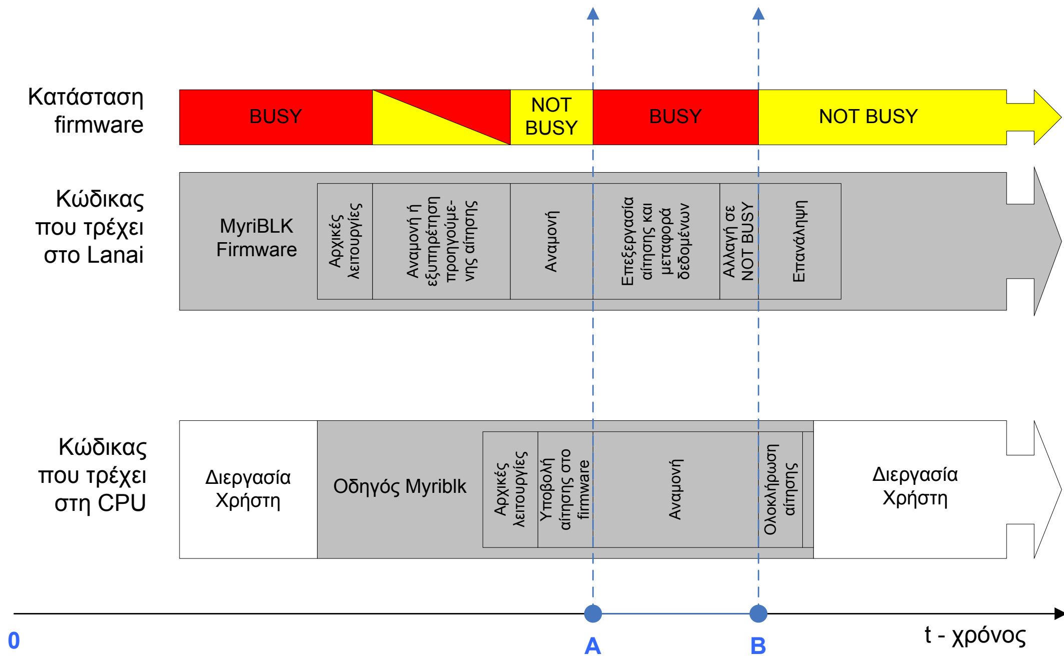
Σύμφωνα με το σχεδιασμό μας μέχρι αυτό το σημείο, το υλικολογισμικό βρίσκεται σε μία κατάσταση αναμονής (ουσιαστικά σε έναν επαναληπτικό βρόγχο), ελέγχοντας συνέχεια τις κοινές μεταβλητές, μέχρι να διαπιστώσει μία υποβολή αίτησης από τον οδηγό. Μόλις τη διαπιστώσει, ξεκινά τη διαδικάσια μεταφοράς ανάλογα με το είδος της αίτησης. Από τη στιγμή αυτή, και μέχρι να ολοκληρωθεί η μεταφορά, ο οδηγός δεν μπορεί να προχωρήσει σε καμία άλλη ενέργεια. Σε περίπτωση που το κάνει, θα έχουμε ανώμαλη ροή προγράμματος και πιθανό αδιέξοδο.

Ο πρώτος τρόπος που θα παρουσιάσουμε για να ξεπεράσουμε το πρόβλημα και να συγχρονίσουμε τα δύο κομμάτια κώδικα, είναι αυτός του περιοδικού ελέγχου (polling). Σύμφωνα με αυτόν τον τρόπο, ορίζουμε αρχικά δύο καταστάσεις, στις οποίες μπορεί να βρίσκεται το υλικολογισμικό. Η πρώτη είναι η κατάσταση `BUSY` και η δεύτερη η `NOT BUSY`. Στην κατάσταση `BUSY`, όπως υποδηλώνει και ο όρος, το υλικολογισμικό είναι απασχολημένο με κάποια εργασία και δεν βρίσκεται σε θέση να ικανοποιήσει εισερχόμενες αιτήσεις. Αυτό μπορεί να συμβαίνει, είτε επειδή ικανοποιεί ήδη μία αίτηση (και

βρίσκεται π.χ. στη μέση μίας μεταφοράς δεδομένων), είτε επειδή το υλικολογισμικό κάνει κάποιου είδους αρχικοποίηση πριν εισέλθει στην κατάσταση αναμονής αιτήσεων. Αντίθετα στην κατάσταση `NOT BUSY` το υλικολογισμικό είναι διαθέσιμο να ικανοποιήσει καινούρια εισερχόμενη αίτηση από τον οδηγό. Οι δύο καταστάσεις αυτές απεικονίζονται με μία μεταβλητή που παίρνει τις ανάλογες δυαδικές τιμές στην κοινή δομή μεταβλητών. Με αυτόν τον τρόπο, ο οδηγός διαβάζοντας αυτή τη μεταβλητή, γνωρίζει κάθε στιγμή την κατάσταση στην οποία βρίσκεται το υλικολογισμικό.

Από τη στιγμή που ο οδηγός είναι σε θέση να γνωρίζει την κατάσταση στην οποία βρίσκεται το υλικολογισμικό, συνεπάγεται πως αντιλαμβάνεται πότε μπορεί να υποβάλλει καινούρια αίτηση και πότε όχι. Ο οδηγός έχει τη δυνατότητα να υποβάλλει αίτηση, όταν το υλικολογισμικό δηλώνει κατάσταση `NOT BUSY`. Επιπροσθέτως ο οδηγός, αφού έχει υποβάλλει μία αίτηση δεν μπορεί να προχωρήσει και να ολοκληρώσει την διαδικασία αυτής της αιτήσης, αν προηγουμένως δεν ενημερωθεί από το υλικολογισμικό, ότι η μεταφορά δεδομένων για τη συγκεκριμένη αίτηση έχει ολοκληρωθεί. Αυτό σημαίνει, ότι ο οδηγός πρέπει να περιμένει μέχρι νέα κατάσταση `NOT BUSY` του υλικολογισμικού, ώστε να καταλάβει πως η προηγούμενη αίτηση ολοκληρώθηκε. Γίνεται έτσι φανερό, ότι ο κώδικας οδηγού, κάθε φορά που υποβάλλει μία αίτηση στο υλικολογισμικό, πρέπει να μπει σε μία δική του κατάσταση αναμονής μέχρι να ολοκληρωθεί η εργασία που πρέπει να επιτελέσει το υλικολογισμικό. Στην κατάσταση αυτή αναμονής το μόνο που κάνει είναι να ελέγχει περιοδικά τη μεταβλητή κατάστασης του υλικολογισμικού, μέχρι να διαπιστώσει αλλαγή από την κατάσταση `BUSY` στην `NOT BUSY` και να συμπεράνει ολοκλήρωση της παρούσας αίτησης. Από αυτή την κατάσταση αναμονής και ελέγχου, που υλοποιείται με ένα είδος επαναληπτικού βρόγχου, παίρνει και το όνομα της η μέθοδος polling. Μόλις ο οδηγός αντιληφθεί την ολοκλήρωση της αίτησης, μόνο τότε συνεχίζει στις υπόλοιπες ενέργειες που χρειάζονται για να την ολοκληρώσει και να προχωρήσει σε πιθανή επόμενη αίτηση. Όλη αυτή η διαδικασία προυποθέτει και τον σωστό συγχρονισμό από την πλευρά του υλικολογισμικού, την εναλλαγή δηλαδή των καταστάσεων `BUSY` και `NOT BUSY` σε ακριβή χρονικά σημεία. Ο ακριβής καθορισμός των σημείων αυτών παίζει πολύ σημαντικό ρόλο στην αποφυγή καταστάσεων συναγωνισμού (race conditions), αλλά αυτός αποτελεί αμιγώς θέμα υλοποίησης και θα περιγραφεί στο αντίστοιχο κεφάλαιο.

Ο συγχρονισμός των δύο μερών του κώδικα χρησιμοποιώντας τον παραπάνω τρόπο περιοδικού ελέγχου γίνεται πιο σαφής αν παρατηρήσουμε το σύνολο των καταστάσεων



Σχήμα 3.9: *Συγχρονισμός Περιοδικού Ελέγχου*

και την ακολουθία τους στο Σχ. 3.9

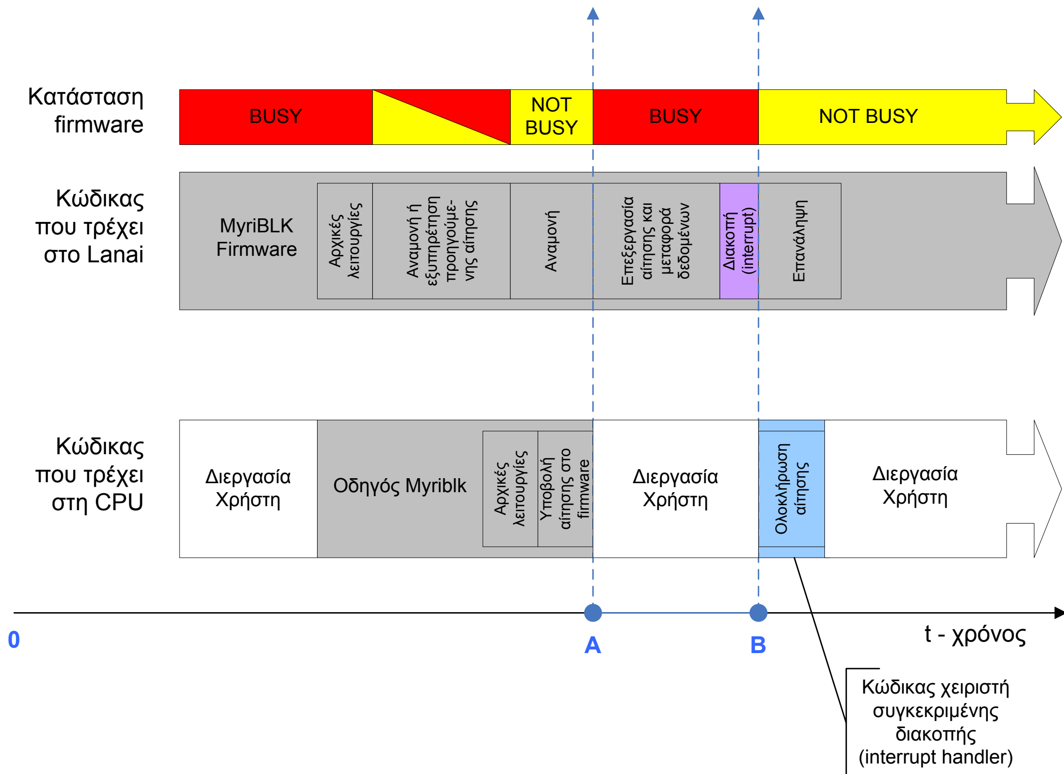
3.5.2 Συγχρονισμός με χρήση διακοπών συστήματος (Interrupt driven)

Μελετώντας προσεκτικά το Σχ. 3.9, παρατηρούμε πως το χρονικό διάστημα από τη στιγμή που ο οδηγός υποβάλλει μία αίτηση, μέχρι να διαπιστώσει ότι το υλικολογισμικό ολοκλήρωσε τις ενέργειες που έπρεπε να επιτελέσει από μέρους του (από το σημείο A στο B του Σχ. 3.9), ο οδηγός είναι αναγκασμένος να βρίσκεται σε αναμονή. Κατά τη διάρκεια αυτής της αναμονής, παραμένει σε έναν βρόγχο ελέγχοντας της μεταβλητής κατάστασης του υλικολογισμικού. Με αυτή την προσέγγιση είναι φανερό πως για όσο διαρκεί αυτή η αναμονή, επιβαρύνεται η CPU του συστήματος με κώδικα ο οποίος δεν παράγει τίποτα, ενώ θα μπορούσε να επιτελεί κάποια άλλη λειτουργία ή σε ακόμη καλύτερη περίπτωση, να έχει δρομολογηθεί στη θέση του μία εντελώς διαφορετική διεργασία. Αναλογιζόμενοι τον τεράστιο αριθμό διαφορετικών αιτήσεων, που συνεπάγονται τον αντίστοιχο πολλαπλασιασμό αυτού του χρονικού διαστήματος, κατανοούμε το μεγάλο κόστος που έχει ένας τέτοιος σχεδιασμός στην απόδοση του συστήματος. Λαμβάνοντας υπ' όψιν το γεγονός αυτό, και στοχεύοντας στη βελτίωση της απόδοσης και επίδοσης αναπροσαρμόζουμε το σχεδιασμό μας ανάλογα.

Το πρόβλημα απόδοσης που προκύπτει στην προηγούμενη περίπτωση περιοδικού ελέγ-

χου μπορεί να ξεπεραστεί αν εκμεταλλευτούμε τον μηχανισμό διακοπών συστήματος που προσφέρει ο πυρήνας Linux και περιγράψαμε στο Κεφ. 2. Σύμφωνα με αυτόν το μηχανισμό, μόλις ο οδηγός υποβάλλει την αίτηση στο υλικολογισμικό η συνάρτηση request επιστρέφει, χωρίς να έχει ολοκληρώσει την διαδικασία χειρισμού της αίτησης που μόλις υπέβαλε. Η αίτηση αυτή παραμένει εκκρεμής στο σύστημα. Από τη στιγμή αυτή, ο κώδικας του οδηγού έχει εγκαταλείψει την CPU, η οποία είναι πλέον ελεύθερη να εξυπηρετήσει οποιαδήποτε άλλη διεργασία. Ενώ το κεντρικό σύστημα δρομολογεί άλλες διεργασίες στην κεντρική CPU, το υλικολογισμικό έχει δεχθεί την αίτηση και αναλαμβάνει την μεταφορά των δεδομένων. Μόλις το υλικολογισμικό ολοκληρώσει όλες τις λειτουργίες που χρειάζονται, παράγει τελικά μία διακοπή συστήματος (interrupt) για να σηματοδοτήσει το γεγονός. Η διακοπή αυτή συστήματος γίνεται στο σημείο που με τον προηγούμενο σχεδιασμό θα άλλαζε η κατάσταση από `BUSY` σε `NOT BUSY`. Ο πυρήνας με τη σειρά του αντιλαμβάνεται τη διακοπή και καλείται να την χειριστεί ανάλογα.

Για να επιτευχθεί αυτό, πρέπει να έχουμε ήδη εγκαταστήσει τον κατάλληλο χειριστή διακοπών. Στην περίπτωση υλοποίησης με συγχρονισμό τέτοιου είδους, ο χειριστής διακοπών έχει εγκατασταθεί κατά την αρχικοποίηση του οδηγού και είναι διαθέσιμος στο σύστημα από την αρχή. Ο ρόλος του χειριστή περιορίζεται στην ολοκλήρωση της εκκρεμούς αίτησης και στην ενημέρωση του οδηγού, ότι μπορεί να συνεχίσει στην επόμενη αίτηση. Έτσι ο χειριστής από τη στιγμή που αρχίζει να εκτελείται, γνωρίζει (λόγω σχεδιασμού) ότι η αίτηση που εκκρεμούσε μέχρι πρότεινος έχει ολοκληρωθεί από την πλευρά του υλικολογισμικού, και πρέπει να γίνουν οι κατάλληλες ενέργειες για την πλήρη ολοκλήρωσή της και από την πλευρά του οδηγού. Ουσιαστικά ο χειριστής εκτελεί τις ενέργειες ολοκλήρωσης (με τη διεπαφή του υποσυστήματος block του πυρήνα) που στην προηγούμενη περίπτωση θα έκανε ο οδηγός. Αφού τις πραγματοποιήσει επανενεργοποιεί την ουρά αιτήσεων (έτσι ώστε να καλεστεί ξανά η συνάρτηση request του οδηγού για να εξυπηρετήσει την επόμενη αίτηση) και επιστρέφει. Η διαδικασία αυτή επαναλαμβάνεται για κάθε αίτηση. Με αυτή τη λειτουργικότητα του χειριστή συμμορφωνόμαστε και με τη γενικότερη αρχή που διέπει τους κώδικες χειριστών και επιβάλλει τον ελάχιστο δυνατό, μη απαιτητικό σε πόρους, κώδικα στο σώμα τους. Στο Σχ. 3.10 φαίνεται σαφέστερα η διαδικασία που ακολουθούμε και ο τρόπος συγχρονισμού και επικοινωνίας των δύο μερών κώδικα (οδηγού και υλικολογισμικού), με την επιπλέον διάκριση του κώδικα χειριστή:



Σχήμα 3.10: *Συγχρονισμός με τη χρήση Διακοπών Συστήματος*

Με μία απλή σύγκριση των δύο σχημάτων (Σχ. 3.9, Σχ. 3.10) που προηγήθηκαν, γίνεται φανερή η σημασία του δεύτερου τρόπου συγχρονισμού. Παρατηρούμε ότι εξαλοίφεται όλος ο χρόνος αναμονής από την πλευρά του οδηγού και χρησιμοποιείται εξ' ολοκλήρου για την εξυπηρέτηση διαφορετικών διεργασιών. Το κέρδος σε απόδοση καταλήγει να είναι πολλαπλάσιο, και ο σχεδιασμός μας πιο σωστά δομημένος. Φυσικά, όταν φτάνουμε στο σημείο να υλοποιήσουμε (όπως θα δούμε στο επόμενο κεφάλαιο), η μετάβαση από τον έναν τρόπο συγχρονισμού στον άλλο δεν είναι απολύτως προφανής και εγείρονται νέα προβλήματα που καλούμαστε να αντιμετωπίσουμε, τα οποία όμως θα αναλύσουμε τη στιγμή που θα τα συναντήσουμε.

4 Υλοποίηση της συσκευής MyriBLK

Στο κεφάλαιο αυτό, περιγράφεται η ακριβής υλοποίηση της συσκευής block `/dev/myriblk`. Η υλοποίηση έχει ως στόχο την απόδειξη της αρχικής μας υπόθεσης και τη γενικότερη μελέτη της περίπτωσης μίας μονάδας δίσκου που βρίσκεται πάνω σε έναν προσαρμογέα δικτύου. Η υλοποίηση μας φέρνει επίσης άμεσα εκτεθειμένους σε προγραμματιστικά προβλήματα, που σχετίζονται με τα υποσυστήματα του πυρήνα, αλλά και τους περιορισμούς του υλικού. Τέλος, μας παρέχει τη δυνατότητα πειραματικής αξιολόγησης της σχεδίασής μας.

Κατά τη διάρκεια της σχεδίασης, χωρίσαμε τη συσκευή σε επιμέρους κομμάτια που επιτελούν διαφορετικές λειτουργίες και περιγράψαμε διαφορετικούς μηχανισμούς που επικοινωνούν μεταξύ τους μέσω κάποιων καλώς ορισμένων διεπαφών. Την ίδια αντικειμενοστρεφή προσέγγιση θα ακολουθήσουμε και κατά τη διαδικασία υλοποίησης. Θα χωρίσουμε την υλοποίηση σε υποενότητες, που θα ακολουθούν το πρότυπο του σχεδιαστικού μοντέλου που έχουμε ορίσει προηγουμένως. Με αυτόν τον τρόπο, η ίδια η υλοποίηση θα αποτελείται από ανεξάρτητους μηχανισμούς, που θα συνδυαστούν κατάλληλα μεταξύ τους για να παράγουν τελικά το επιθυμητό αποτέλεσμα.

Το κεφάλαιο θα ξεκινήσει με την ακριβή περιγραφή του περιβάλλοντος εργασίας και του υλικού. Θα εξετάσουμε με ποιόν τρόπο μπορεί να συγχωνευθεί η υλοποίηση μας με το πρωτότυπο GM και σε ποιά σημεία χρειάζονται τροποποιήσεις και νέος κώδικας. Συνεχίζοντας θα προχωρήσουμε στην υλοποίηση από την πλευρά του κόμβου. Στο μέρος αυτό θα περιγράψουμε την πλήρη διεπαφή του οδηγού με το υποσύστημα block του πυρήνα, καθώς και τη διεπαφή που πρέπει να υλοποιήσουμε για την επικοινωνία με το υλικολογισμικό. Στη συνέχεια, θα ασχοληθούμε με την πλευρά του προσαρμογέα και

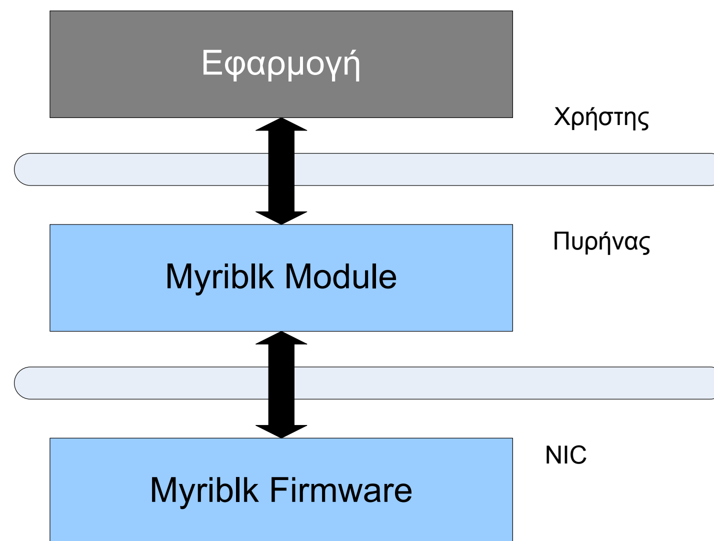
θα τροποποιήσουμε το υλικολογισμικό έτσι ώστε να αναδιοργανώσει αρχικά τη μνήμη και να δεσμεύσει το χώρο που θα χρησιμοποιηθεί ως μονάδα δίσκου. Επίσης θα υλοποιήσουμε τον μηχανισμό μεταφοράς δεδομένων και τον μηχανισμό επικοινωνίας με τον οδηγό. Το κεφάλαιο θα κλείσει παρουσιάζοντας τους δύο τρόπους συγχρονισμού των δύο ανεξάρτητων μερών κώδικα, που συναντήσαμε κατά το σχεδιασμό.

4.1 Πλατφόρμα εργασίας

Η υλοποίησή θα λάβει χώρα σε ένα ευρέως χρησιμοποιούμενο σύστημα (commodity hardware), με την ειδική προσθήκη του προσαρμογέα δικτύου που έχουμε περιγράψει στη σχεδίασή μας. Συγκεκριμένα, το σύστημα ανήκει στην κατηγορία αρχιτεκτονικής i386 και το λειτουργικό σύστημα που τρέχει είναι το Debian GNU/Linux. Η διανομή αυτή παρέχει ένα ολοκληρωμένο λειτουργικό σύστημα που χρησιμοποιεί ώς πυρήνα τον Linux και πρόσθετες εφαρμογές διαχείρησης. Κατά τη διάρκεια εκπόνησης της εργασίας, χρησιμοποιήσαμε τον πιο πρόσφατο σταθερό πυρήνα που ήταν διαθέσιμος την περίοδο της υλοποίησης, ο οποίος είναι ο linux kernel 2.6.24. Όλες οι συναρτήσεις που θα δούμε στη συνέχεια της ενότητας αυτής, βασίζονται στις διεπαφές που ορίζει αυτή η έκδοση πυρήνα. Βέβαια, ο τρόπος με τον οποίο έχουμε σχεδιάσει, μας επιτρέπει να προσαρμόσουμε τον κώδικα και σε επόμενες εκδόσεις, αφού οι αλλαγές που θα χρειαστούν είναι ελάχιστες σε σχέση με το σύνολο της υλοποίησης. Σε αυτό συντείνει σε μεγάλο βαθμό και το γεγονός του αντικειμενοστρεφούς μοντέλου που ακολουθούμε.

Στο σύστημα αυτό προσθέτουμε τον προσαρμογέα δικτύου Myrinet-2000 M3F-PCI64B-2 της εταιρείας Myricom. Ο προσαρμογέας αυτός συνδέεται στο δίαυλο επικοινωνίας PCI/PCI-X του συστήματος και παρέχει όλη τη λειτουργικότητα που χρειαζόμαστε για να υλοποιήσουμε τη συσκευή μας. Είναι πλήρως προγραμματιζόμενος με δικό του μικροεπεξεργαστή, είναι εφοδιασμένος με τοπική μνήμη και μηχανές DMA. Τα συνολικά χαρακτηριστικά του προσαρμογέα όπως επίσης και του μικροεπεξεργαστή της οικογένειας RISC που ονομάζεται Lanai παρουσιάστηκαν στα Σχ. 2.11 και Σχ. 2.12.

Γνωρίζοντας πλέον το υλικό μπορούμε να προχωρήσουμε στην καθεαυτή υλοποίηση. Στην ενότητα του σχεδιασμού, είδαμε πως ο οδηγός block, που θα εξυπηρετήσει τη συσκευή μας, χωρίζεται σε δύο κομμάτια. Ένα εκτελείται στον χώρο πυρήνα και ένα εκτελείται στον προσαρμογέα δικτύου. Σύμφωνα με αυτή την αρχιτεκτονική δύο μερών,



Σχήμα 4.1: *Δομή Συσκευής Myriblk*

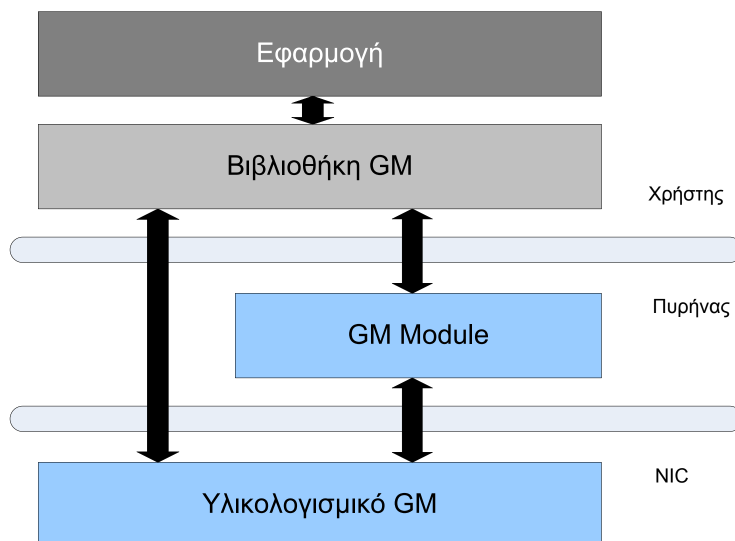
καταλήγουμε στην δομή του Σχ. 4.1

4.2 Συγχώνευση με το GM

Ο προσαρμογέας δικτύου που θα χρησιμοποιήσουμε, συνδυάζεται με το λογισμικό GM για να λειτουργήσει ώς μέρος του συστήματος στο οποίο εγκαθίσταται. Ουσιαστικά το GM αποτελεί τον οδηγό του συγκεκριμένου προσαρμογέα δικτύου. Όπως αναλύσαμε στην Εν. 2.2.2 το GM υλοποιείται σε τρία βασικά μέρη που φαίνονται στο Σχ. 4.2:

- Μια βιβλιοθήκη στο χώρο χρήστη που αναφέρεται στις εφαρμογές
- Ένα module πυρήνα του λειτουργικού συστήματος
- Το υλικολογισμικό (firmware) που τρέχει στον Lanai

Παρατηρώντας τα Σχ. 4.1 και Σχ. 4.2, γίνεται φανερό πως η δομή πάνω στην οποία έχουμε επιλέξει να υλοποιήσουμε μοιάζει σε μεγάλο βαθμό με αυτή του GM. Αυτό δεν συμβαίνει τυχαία, αφού ο καταμερισμός αυτός από την πλευρά της συσκευής myriblk είναι αποτέλεσμα του σχεδιασμού μας που έγινε με βάση το GM. Πιο συγκεκριμένα αν εξαιρέσουμε το μέρος της βιβλιοθήκης στο χώρο χρήστη το υπόλοιπο κομμάτι ταυτίζεται. Έχοντας αυτήν την παρατήρηση υπ' όψιν, στην υλοποίηση μας θα προσπαθήσουμε



Σχήμα 4.2: *Δομή Myrinet/GM*

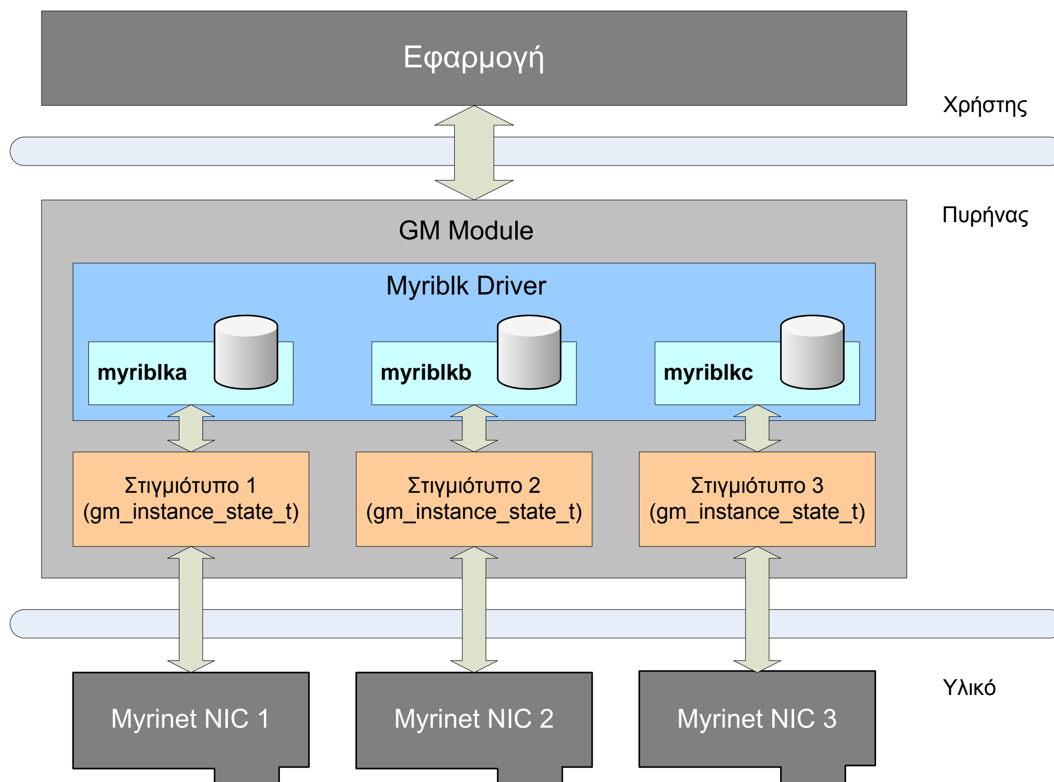
να συγχωνεύσουμε κάθε μέρος της συσκευής με το αντίστοιχο μέρος του GM. Με αυτόν τον τρόπο θα επωφεληθούμε με όλα τα πλεονεκτήματα μίας τέτοιας υλοποίησης που προαναφέρθηκαν σε προηγούμενα κεφάλαια. Βασιζόμενοι σε αυτή τη λογική, θα υλοποιήσουμε το κομμάτι της συσκευής block που αναφέρεται στην πλευρά του κόμβου τροποποιώντας κατάλληλα το module πυρήνα του GM και το κομμάτι της συσκευής block που αναφέρεται στον προσαρμογέα δικτύου τροποποιώντας το υλικολογισμικό που υλοποιεί το GM και τρέχει στον Lanai. Η βιβλιοθήκη στο χώρο χρήστη δεν θα μας απασχολήσει στην παρούσα εργασία (θα παραλείπεται στο εξής όταν αναφερόμαστε στο GM), αφού η διεπαφή με το χώρο χρήστη είναι ήδη υλοποιημένη και γίνεται μέσω των κλήσεων συστήματος για συσκευές block σε ό,τι αφορά στη δική μας λειτουργικότητα.

Για να καταφέρουμε να συγχωνεύσουμε διαφανώς τον κώδικα μας μέσα στον κώδικα του GM, να χρησιμοποιήσουμε ήδη υλοποιημένες συναρτήσεις σε όποιο σημείο μας εξυπηρετεί και να παραμείνει ο κώδικας αρκετά ανεξάρτητος και επεκτάσιμος, κρίνεται απολύτως αναγκαία η μελέτη της ακριβούς υλοποίησης του ίδιου του GM. Για το λόγο αυτό, θα περιγράψουμε αρχικά τους βασικούς μηχανισμούς και λειτουργίες που το διέπουν και πάνω στις οποίες θα στηριχτεί και η δική μας υλοποίηση.

Το GM φορτώνεται στο σύστημα όπως κάθε άλλο module. Γίνεται `insmod` στον πυρήνα και ξεκινά την αρχικοποίηση, με το δεύτερο μέρος του GM (GM module) να εκτελείται πρώτο. Κατά την αρχικοποίησή του, το GM module συντονίζει τα υπόλοιπα δύο

μέρη. Εγκαθιστά τη βιβλιοθήκη (GM library) στο χώρο χρήστη και καλεί έναν εσωτερικό φορτωτή που φορτώνει το μεταγλωττισμένο υλικολογισμικό (GM firmware - MCP) στον Lanai. To GM module συντονίζεται επίσης κατάλληλα με το GM firmware, μέχρι το δεύτερο να ολοκληρώσει και την δική του αρχικοποίηση, από την πλευρά του προσαρμογέα πλέον. Τελικά, μετά το περάς της αρχικοποίησης του GM module, έχουν εγκατασταθεί σωστά και τα τρία μέρη στο σύστημα, είναι πλήρως λειτουργικά και βρίσκονται σε αναμονή για να το εξυπηρετήσουν. Αντίστοιχη είναι και η διαδικασία εκφόρτωσης του GM. Η διαδικασία ξεκινά πάλι από το GM module, το οποίο συντονίζει τα άλλα δύο μέρη και τελικά επιτελεί όλες τις λειτουργίες εκκαθάρισης που απαιτούνται, έτσι ώστε να εκφορτωθεί το GM ασφαλώς και χωρίς εκκρεμότητες από το σύστημα. Παρατηρούμε ότι για την πλήρη εγκατάσταση και απεγκατάσταση του GM στο σύστημα, συντονιστικό ρόλο παίζει το GM module, το οποίο εκκινεί και τερματίζει κάθε διαδικασία που έχει να κάνει με τον προσαρμογέα δικτύου. Ακολουθώντας αυτή τη φιλοσοφία, όλες οι συντονιστικές ενέργειες της συσκευής myriblk θα επιτελούνται από το κομμάτι του οδηγού που θα τρέχει στο χώρο πυρήνα.

Ένα σημαντικό σημείο της διαδικασίας αρχικοποίησης (που αποτελεί και σχεδιαστική επιλογή της υλοποίησης του GM), είναι ο τρόπος με τον οποίο απεικονίζει εσωτερικά το GM module τον κάθε φυσικό προσαρμογέα δικτύου. Το GM module ορίζει μία δομή που την ονομάζει `gm_instance_state_t` (στιγμιότυπο) και με αυτή απεικονίζει κάθε φυσικό προσαρμογέα δικτύου που βρίσκεται στο σύστημα. Η `gm_instance_state_t` έχει ένα πλήθος μεταβλητών που χαρακτηρίζουν το κάθε στιγμιότυπο. Ο λόγος που γίνεται αυτό, μπορεί να μην είναι προφανής στο συγκεκριμένο σημείο, αλλά πρέπει να αναλογιστούμε πως το GM έχει υλοποιηθεί με τέτοιο τρόπο, ώστε να μπορεί να εξυπηρετήσει άνω του ενός προσαρμογέα δικτύου. Στη δική μας περίπτωση δεν συναντάται κάτι τέτοιο, αλλά σε άλλα συστήματα μπορεί να υπάρχουν πολλαπλοί προσαρμογείς δικτύου. Το GM πρέπει να είναι σε θέση να τους διακρίνει εσωτερικά, για να μπορέσει στην συνέχεια να τους ελέγξει. Για να έχει ο κώδικάς μας τη δυνατότητα υποστήριξης πολλών προσαρμογέων, αποφασίζουμε να διατηρήσουμε αυτή τη δομή.



Σχήμα 4.3: *Συγχώνευση οδηγού MyriBLK με το GM*

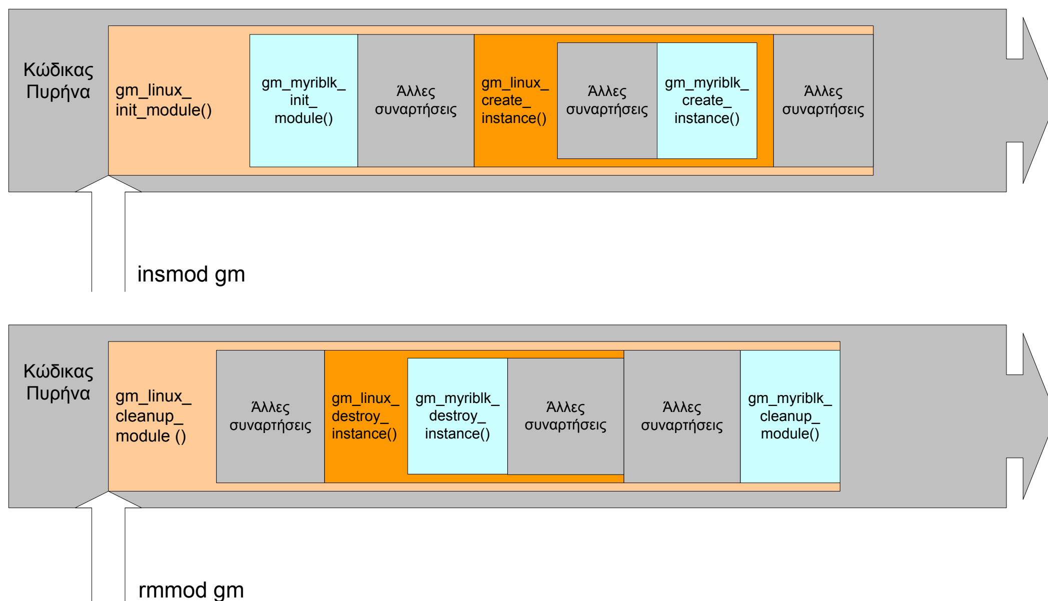
4.3 Υλοποίηση από την πλευρά του κόμβου (Host)

Η πραγματική υλοποίηση της συσκευής myriblk ξεκινά σε αυτή την ενότητα, από την πλευρά του κόμβου. Αναφέραμε ότι η υλοποίησή μας θα έχει τη δυνατότητα να υποστηρίξει πολλαπλούς προσαρμογείς. Αυτό σημαίνει, ότι στο σύστημα θα εμφανίζεται μία ξεχωριστή μονάδα δίσκου για κάθε προσαρμογέα. Η κάθε μονάδα δίσκου θα αντικατοπτρίζει τον αντίστοιχο χώρο μνήμης SRAM κάθε προσαρμογέα. Για παράδειγμα, σε ένα σύστημα με τρεις προσαρμογείς δικτύου πρέπει να καταλήξουμε με τρεις block συσκευές myriblk, έστω `myriblka`, `myriblkb`, `myriblkc`, η κάθε μία από τις οποίες θα διαχειρίζεται την SRAM του αντίστοιχου προσαρμογέα δικτύου. Για να πετύχουμε αυτή τη δομή, πρέπει να δουλέψουμε στη βάση των στιγμιοτύπων που ορίζει το GM. Είδαμε πως κάθε στιγμιότυπο συνδέεται με έναν φυσικό προσαρμογέα δικτύου και αποτελεί ταυτόχρονα την αναπαράστασή του στο σύστημα. Εκμεταλλευόμενοι αυτό το γεγονός, πρέπει με τη σειρά μας να αντιστοιχίσουμε μία block συσκευή σε κάθε στιγμιότυπο. Η συσκευή αυτή θα διαχειρίζεται τον προσαρμογέα δικτύου (ουσιαστικά την μνήμη SRAM του) διαμέσου του στιγμιοτύπου του, το οποίο έχει δημιουργήσει το GM. Τελικά,

καταλήγουμε σε μία πλήρως οργανωμένη δομή που μοιάζει με αυτή του Σχ. 4.3.

4.3.1 Διεπαφή οδηγού (MyriBLK module) με το GM

Από τη στιγμή αυτή, θα αναφερόμαστε στο μέρος του κώδικα που έχουμε υλοποιήσει εμείς και τρέχει στην πλευρά του κόμβου ώς MyriBLK module σε αντιστοιχία με το GM module. Στην τελική υλοποίηση το MyriBLK module θα έχει συγχωνευτεί με το GM module και θα τρέχει παράλληλα με αυτό, προσθέτοντας στο σύστημα τη λειτουργικότητα που έχουμε σχεδιάσει. Είναι φανερό πως μέσα στο MyriBLK module θα υλοποιηθούν όλες οι συναρτήσεις που έχουν να κάνουν με τη συσκευή block `myriblk` αλλά και όλοι οι μηχανισμοί επικοινωνίας με το firmware. Πριν προχωρήσουμε στην αναλυτική περιγραφή αυτών των συναρτήσεων και μεθόδων, πρέπει να εξετάσουμε τον τρόπο με τον οποίο θα καταστεί δυνατή η επικοινωνία του δικού μας κώδικα με τον κώδικα του GM. Το GM module, όταν φορτωθεί στον πυρήνα ξεκινά μία διαδικασία αρχικοποίησης, όπως περιγράψαμε παραπάνω. Σε κάποια συγκεκριμένα σημεία της αρχικοποίησης αυτής (Σχ. 4.4), καλούμε δικές μας συναρτήσεις οι οποίες υλοποιούνται στο MyriBLK module και θα κάνουν την αντίστοιχη αρχικοποίηση της συσκευής `myriblk`. Η ίδια διαδικασία ακολουθείται και κατά την εκφόρτωση του GM.



Σχήμα 4.4: *Αρχικοποίηση και Εκκαθάριση MyriBLK Module*

Συγκεκριμένα, όπως φαίνεται και στο Σχ. 4.4, μόλις φορτωθεί στον πυρήνα το GM

module η πρώτη συνάρτηση που εκτελείται είναι η `gm_linux_init_module (void)` που αποτελεί και τη συνάρτηση εκκίνησης του module. Η `gm_linux_init_module` είναι υπεύθυνη για την αρχικοποίηση του GM και για το λόγο αυτό κάνει μία μεγάλη σειρά απαραίτητων ενεργειών και καλεί τις αντίστοιχες συναρτήσεις για να ικανοποιήσουν κάθε ανάγκη. Σε κατάλληλο σημείο της `gm_linux_init_module` καλούμε την δική μας συνάρτηση `gm_myriblk_init_module()`. Σε αυτό το σημείο να σημειώσουμε ένα θέμα που έχει να κάνει με την ονοματολογία και θα βοηθήσει τον αναγνώστη στην ευκολότερη ανάγνωση της εργασίας. Οτιδήποτε νέο έχουμε ορίσει κατά τη διάρκεια της υλοποίησης και τροποποίησης του GM και αφορά στη συσκευή myriblk ξεκινά με το χαρακτηριστικό "`gm_myriblk`" (μεταβλητές, δομές, συναρτήσεις κτλ.). Έτσι στο εξής οποιοδήποτε όνομα ξεκινά με αυτό το χαρακτηριστικό υποδηλώνει δική μας υλοποίηση ή τροποποίηση, που αφορά στη συσκευή `myriblk`. Η `gm_myriblk_init_module()` είναι η πρώτη συνάρτηση που θα εκτελέσει ο οδηγός μας, περιορισμένης βέβαια ευθύνης, αφού δεν είναι σε θέση να δημιουργήσει σε αυτό το χρονικό σημείο πραγματικές συσκευές. Ο ρόλος που επιτελεί όμως η `gm_myriblk_init_module` είναι να δηλώσει τον οδηγό ως οδηγό block στον πυρήνα και να του εξασφαλίσει έναν μείζονα αριθμό. Η υλοποίηση της είναι η εξής:

```
int
gm_myriblk_init_module (void)
{
	gm_myriblk_major = register_blkdev(gm_myriblk_major, "myriblk");
	if (gm_myriblk_major <= 0) {
		printk(KERN_WARNING "myriblk: unable to get major number\n");
		return -EBUSY;
	}
	return 0;
}
```

Αφού καλέσει η `gm_linux_init_module` την `gm_myriblk_init_module` και μία σειρά άλλων συναρτήσεων καλεί σε επόμενο σημείο τη συνάρτηση `gm_linux_create_instance()`. Η συνάρτηση αυτή είναι υπεύθυνη να αρχικοποιήσει έναν συγκεκριμένο φυσικό προσαρμογέα δικτύου του συστήματος και να τον αναπαραστήσει κατάλληλα στο GM, δημιουργώντας το αφιερωμένο σε αυτόν στιγμιότυπο. Στην περίπτωση επιτυχούς ολοκλήρωσής της, εκτός των πολλαπλών λειτουργιών που έχει επιτελέσει κατά την εκτέλεση της, δημιουργεί μία δομή `gm_instance_state_t` και θέτει μέρος των μεταβλητών της. Διαπιστώνουμε πως αυτή είναι η συνάρτηση που ουσιαστικά κάνει διαθέσιμο έναν προσαρμογέα στο σύστημα. Από τη στιγμή που επιθυμούμε να συνδέσουμε μία συσκευή myriblk με ένα στιγμιότυπο (και έναν φυσικό προσαρμογέα στη συνέχεια),

μέσα στη συγκεκριμένη συνάρτηση είναι το κατάλληλο σημείο να το κάνουμε. Για το λόγο αυτό, μέσα στην συνάρτηση `gm_linux_create_instance` καλούμε τη δική μας `gm_myriblk_create_instance()` η οποία θα κάνει την πραγματική αρχικοποίηση της συσκευής `myriblk` για το συγκεκριμένο στιγμιότυπο. Η συνάρτηση αυτή υλοποιείται, όπως και όλες οι υπόλοιπες, στο MyriBLK module και έχει ώς εξής:

```
int
gm_myriblk_create_instance (gm_instance_state_t *is)
{
          gm_myriblk_devs[gm_myriblk_num_instances] = kmalloc(sizeof(gm_myriblk_dev_t), \
                                                             GFP_KERNEL);
          if (gm_myriblk_devs[gm_myriblk_num_instances] == NULL) {
                     goto out_noDevice;
          }
          is->gm_myriblk_dev_pnt = gm_myriblk_devs[gm_myriblk_num_instances];
          gm_myriblk_setup_device(gm_myriblk_devs[gm_myriblk_num_instances], \
                                  gm_myriblk_num_instances, is);
          return 0;

 out_noDevice:
          printk (KERN_EMERG "myriblk: Could not allocate the gm_myriblk_dev_t  \
                  for the device\n");
          return 1;
}
```

Παρατηρούμε ότι έχουμε έναν πίνακα από δείκτες σε συσκευές myriblk και κάθε στιγμιότυπο (instance) ανάλογα με τον αύξοντα αριθμό του (ο οποίος έχει οριστεί από το GM και έχει γίνει ίσος με `gm_myriblk_num_instances`) αντιστοιχίζεται μέσω του πίνακα σε μία συσκευή `myriblk`. Επίσης, έχοντας τροποποιήσει κατάλληλα τη δομή `gm_instance_state_t` θέτουμε και έναν δείκτη που δείχνει κατευθείαν στην καινούρια συσκευή, δίνοντας έτσι τη δυνατότητα στο στιγμιότυπο να την προσπελάζει απευθείας. Τελικά, δημιουργούμε και εγκαθιστούμε τη συσκευή (ο ακριβής τρόπος που το πετυχαίνουμε θα αναλυθεί στην συνέχεια, στην ενότητα της διεπαφής του οδηγού με το επίπεδο block του πυρήνα) με τη συνάρτηση `gm_myriblk_setup_device()`. Αν η `gm_myriblk_create_instance` επιστρέψει επιτυχώς, έχουμε ολοκληρώσει την επικοινωνία μας με το GM module και έχουμε συγχωνεύσει πλήρως τη συσκευή μας μέσα σε αυτό. Οι υπόλοιπες block λειτουργίες της συσκευής είναι ανεξάρτητες από το GM module και δεν το εμπλέκουν όπως θα δούμε στη συνέχεια. Επισημαίνουμε πως αναφέραμε ”block” λειτουργίες, γιατί όταν φτάσουμε στο σημείο επικοινωνίας με το firmware θα επανέλθουμε στο GM.

Παρόμοια και αντίστροφη διαδικασία ακολουθούμε κατά την εκφόρτωση του GM. Όταν χρειαστεί να γίνει εκφόρτωση, η πρώτη συνάρτηση που εκτελεί το GM είναι η `gm_linux_cleanup_module()` που αποτελεί και τη συνάρτηση εκκαθάρισης του GM module.

Εκτός των άλλων συναρτήσεων που καλεί για να πραγματοποιήσει την εκκαθάριση η `gm_linux_cleanup_module()` σε κάποιο σημείο καλεί την `gm_linux_destroy_instance()` για κάθε στιγμιότυπο. Αυτή είναι υπεύθυνη για την ασφαλή διαγραφή ενός στιγμιοτύπου, που συνεπάγεται την ακύρωση της παρουσίας του αντίστοιχου προσαρμογέα δικτύου από το σύστημα. Από τη στιγμή που αυτή η συνάρτηση παύει την αναπαράσταση του προσαρμογέα, σε αυτή τη χρονική στιγμή πρέπει και εμείς να καταργήσουμε τη συσκευή `myriblk` που αναφέρεται στον συγκεκριμένο προσαρμογέα. Έτσι στην αρχή της `gm_linux_destroy_instance` καλούμε την δική μας `gm_myriblk_destroy_instance()`, η οποία είναι υπεύθυνη να καταργήσει από το σύστημα τη συσκευή `myriblk` που αναφέρεται στο συγκεκριμένο στιγμιότυπο και προσαρμογέα δικτύου. Η `gm_myriblk_destroy_instance()` υλοποιείται με τον παρακάτω τρόπο:

```
int
gm_myriblk_destroy_instance (gm_instance_state_t *is)
{
          gm_myriblk_dev_t *dev = is->gm_myriblk_dev_pnt;

          if (dev->gd) {
                    del_gendisk(dev->gd);
                    put_disk(dev->gd);
          }
          if (dev->queue)
                    blk_cleanup_queue(dev->queue);

          kfree(dev);
          return 0;
}
```

Βλέπουμε, πως προσπελάζουμε τη συσκευή που αναφέρεται στον προσαρμογέα δικτύου, που θέλουμε να καταργήσουμε, μέσω δύο δεικτών. Έναν που καταδεικνύει το στιγμιότυπο του προσαρμογέα (`*is`) και έναν που ξεκινά από το στιγμιότυπο και καταδεικνύει τη συσκευή (`*gm_myriblk_dev_pnt`). Από τη στιγμή που έχουμε φτάσει στην κατάλληλη συσκευή, οι υπόλοιπες ενέργειες καταργούν δομές και αποδεσμεύουν τη μνήμη ώστε να απαλοιφεί ολοκληρωτικά η μνήμη από το σύστημα. Ο ακριβής ρόλος τους θα φανεί στην συνέχεια, που θα περιγράψουμε τις διαδικασίες δημιουργίας της συσκευής στο επίπεδο block. Στην περίπτωση που η συνάρτηση `gm_myriblk_destroy_instance` αλλά και η `gm_linux_destroy_instance` ολοκληρώσουν επιτυχώς, ο έλεγχος επιστρέφει στην `gm_linux_cleanup_module` που επιβαρύνεται με τις τελικές ενέργειες εκκαθάρισης. Αντίστοιχα με τη λογική της διαδικασίας φόρτωσης, όταν η `gm_linux_cleanup_module` ολοκληρώσει τις λειτουργίες που εκτελεί για λογαριασμό του GM module (και αμέσως πριν επιστρέψει) καλούμε την `gm_myriblk_cleanup_module()` που θα πραγματοποιήσει τις τελικές λειτουργίες ολοκλήρωσης και

εκκαθάρισης για λογαριασμό του MyriBLK module. Όταν όλες οι συσκευές έχουν καταργηθεί με επιτυχία από την `gm_myriblk_destroy_instance`, ο ρόλος της `gm_myriblk_cleanup_module()` περιορίζεται στην απεγγραφή του οδηγού block από τον πυρήνα:

```
int
gm_myriblk_cleanup_module (void)
{
          unregister_blkdev(gm_myriblk_major, "myriblk");
          return 0;
}
```

Με τις τέσσερις αυτές συναρτήσεις έχουμε συγχωνεύσει πλήρως τη διαδικασία δημιουργίας και κατάργησης των συσκευών μας με το GM. Στο εξής όλες οι υπόλοιπες λειτουργίες που θα υλοποιήσουν ουσιαστικά τη συσκευή θα γίνουν ανεξάρτητα, μέσα στο MyriBLK module. Όπως είδαμε και κατά τη διάρκεια του σχεδιασμού, η υλοποίηση από την πλευρά του κόμβου χωρίζεται σε δύο μέρη. Το ένα έχει να κάνει με την επικοινωνία του πυρήνα Linux με τον οδηγό (διεπαφή με το επίπεδο block) και το δεύτερο με την επικοινωνία του οδηγού με το υλικολογισμικό (διεπαφή με το firmware). Την ανάλυση των δύο αυτών μερών θα περιγράψουμε στη συνέχεια.

4.3.2 Διεπαφή οδηγού με επίπεδο block πυρήνα

Όπως είδαμε και στην ενότητα σχεδίασης 3.3.1 που αναφερόταν στην επικοινωνία του πυρήνα Linux με τον οδηγό μας, για να πραγματοποιηθεί μία πλήρως λειτουργική διεπαφή μεταξύ του οδηγού και του επιπέδου block του πυρήνα χρειάζεται να γίνει υλοποίηση σε τρία επίπεδα. Αρχικά πρέπει να υλοποιηθούν οι μέθοδοι της δομής `block_device_operations` για να έχει τη δυνατότητα η συσκευή να ικανανοποιήσει τις απαιτούμενες κλήσεις συστήματος. Μετά πρέπει να αναπαρασταθεί η μονάδα δίσκου στο σύστημα και να συνδεθεί με μία ουρά αιτήσεων. Τέλος πρέπει να υλοποιηθεί η μέθοδος request που θα είναι υπεύθυνη να εξυπηρετήσει τις εισερχόμενες αιτήσεις. Οι μέθοδοι της δομής `block_device_operations` φαίνονται και υλοποιούνται από τον κώδικα:

```
static struct block_device_operations gm_myriblk_ops = {
       .owner  = THIS_MODULE,
       .open   = gm_myriblk_open,
       .release= gm_myriblk_release,
       .ioctl  = gm_myriblk_ioctl
};
```

```
static int
gm_myriblk_open(struct inode *inode, struct file *filp)
{
          gm_myriblk_dev_t *dev = inode->i_bdev->bd_disk->private_data;

          spin_lock(&dev->lock);
          filp->private_data = dev;
          spin_unlock(&dev->lock);
          return 0;
}
```

```
static int
gm_myriblk_release(struct inode *inode, struct file *filp)
{
          return 0;
}
```

```
static int
gm_myriblk_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, \
                 unsigned long arg)
{
          long nkern_sects; /* number of sectors we would have if we were
                               Counting in a GM_MYRIBLK_KERNEL_SECTOR_SIZE basis */
          struct hd_geometry geo;

          switch (cmd){
                    case HDIO_GETGEO:
                              nkern_sects = gm_myriblk_nsectors*            \
                                            (gm_myriblk_hardsect_size/       \
                                             GM_MYRIBLK_KERNEL_SECTOR_SIZE);
                              geo.cylinders = (nkern_sects & ~0x3f) >> 6;
                              geo.heads = 4;
                              geo.sectors = 16;
                              geo.start = 0;
                              if (copy_to_user((void __user *) arg, &geo,   \
                                  sizeof(geo)))
                                        return -EFAULT;
                              return 0;
                    }
          return -ENOTTY;                 /* unknown command */
}
```

Παρατηρούμε πως η ύπαρξη των μεθόδων `open` και `release` είναι αναγκαία, αλλά στην πραγματικότητα δεν επιτελούν καμία λειτουργία. Αυτό συμβαίνει, όπως αναλύθηκε διεξοδικά κατά το σχεδιασμό (εν. 3.3.1), γιατί η συσκευή `myriblk` δεν χειρίζεται έναν πραγματικό σκληρό δίσκο, αλλά την μνήμη SRAM του προσαρμογέα, τον οποίο αρχικοποιεί και τερματίζει το GM για λογαριασμό μας. Στην περίπτωση της `ioctl`, υλοποιούμε υποθετικά μία γεωμετρία της μονάδας δίσκου, για να είμαστε συμβατοί με κάποιες εφαρμογές που το απαιτούν (όπως αναλύσαμε επίσης στην εν. 3.3.1). Στην προηγούμενη υποενότητα (4.3.1) περιγράψαμε τη συνάρτηση `gm_myriblk_create_instance`, η οποία έχει ως σκοπό να δημιουργήσει και να εγκαταστήσει στο σύστημα την συσκευή myriblk και το κάνει καλώντας τη συνάρτηση `gm_myriblk_setup_device`. Η συνάρτηση αυτή δημιουργεί και εγκαθιστά μία συσκευή `myriblk`. Εκτός

των άλλων, μέσα στην συνάρτηση αυτή αναπαρίσταται η μονάδα δίσκου με την δομή gendisk και συνδέεται με μία ουρά αιτήσεων:

```
static void
gm_myriblk_setup_device(gm_myriblk_dev_t *dev, int which, gm_instance_state_t *is)
{
        memset(dev, 0, sizeof(gm_myriblk_dev_t));
        spin_lock_init(&dev->lock);
        /* general variables */
        dev->dev_to_is = is;
        /* Initially, no outstanding request exists */
        dev->myriblk_req = NULL;
        dev->id = gm_myriblk_num_instances;
        /* values passed from gm_myriblk_glb */
        dev->size = gm_read_lanai_global_u32(is, gm_myriblk_glb.dev_size);

        /* queue setup */
        dev->queue = blk_init_queue(gm_myriblk_request, &dev->lock);
        if (dev->queue == NULL)
                goto out_vfree;
        blk_queue_hardsect_size(dev->queue, gm_myriblk_hardsect_size);
        blk_queue_max_phys_segments(dev->queue, 1); /* just 1 segment */
        blk_queue_max_segment_size(dev->queue, 4096); /* segment size = 1 page */
        blk_queue_max_sectors(dev->queue, 8);      /* maximum 8 sectors(1 page) */
        dev->queue->queuedata = dev;

        /* gendisk setup */
        dev->gd = alloc_disk(GM_MYRIBLK_MINORS);
        if (! dev->gd) {
                printk(KERN_NOTICE "alloc_disk failure\n");
                goto out_vfree;
        }
        dev->gd->major = gm_myriblk_major;
        dev->gd->first_minor = which*GM_MYRIBLK_MINORS;
        dev->gd->fops = &gm_myriblk_ops;
        dev->gd->queue = dev->queue;
        dev->gd->private_data = dev;
        snprintf(dev->gd->disk_name, 32, "myriblk%c", which + 'a');

        set_capacity(dev->gd, (dev->size)/512 );    /* has to be in sectors */
        add_disk(dev->gd);
        return;

  out_vfree:
        printk (KERN_EMERG "myriblk: problem with queue or gendisk\n");
}
```

Αρχικά μηδενίζουμε όλη τη δομή που αναπαριστά την συσκευή μας (`dev`). Θέτουμε κάποιες αρχικές τιμές που αφορούν στο μέγεθος (η συνάρτηση `gm_read_lanai_global_u32` θα αναλυθεί στη συνέχεια), την ταυτότητα και δείκτες χειρισμού. Στη συνέχεια δημιουργούμε και αρχικοποιούμε την ουρά αιτήσεων. Την συνδέουμε με τη συνάρτηση request που θα την εξυπηρετεί και είναι η `gm_myriblk_request`, και ορίζουμε πως κάθε αίτηση υποχρεούται να περιέχει μόνο ένα φυσικό τμήμα, όπως ακριβώς έχουμε προαποφασίσει κατά τη σχεδίαση. Αφού ολοκληρώσουμε με την ουρά, δεσμεύουμε και αρχικοποιούμε τη δομή `gendisk`. Θέτουμε βασικές μεταβλητές που τη χαρακτηρίζουν (μείζονες και ελάσσονες αριθμούς, σύνδεση με μεθόδους κλήσεων συστήματος και ουρά αιτήσεων, όνομα) και ορίζουμε το μέγεθός της (σε sectors) με τη συνάρτηση `set_-`

`capacity`. Τελικά κάνουμε διαθέσιμη τη μονάδα δίσκου, που αναπαριστάται από αυτή τη δομή `gendisk`, στο σύστημα εκτελώντας τη συνάρτηση `add_disk(dev->gd)`. Έχοντας υλοποιήσει και το δεύτερο επίπεδο της διεπαφής οδηγού - επιπέδου block, περνάμε στο τρίτο που αποτελεί και τον πυρήνα κάθε συσκευής block, που είναι η συνάρτηση request:

```
static void
gm_myriblk_request(struct request_queue *q)
{
	struct request *req;

	while ((req = elv_next_request(q)) != NULL) {
		gm_myriblk_dev_t *dev = req->rq_disk->private_data;

		/*
		 * Only a single outstanding request
		 * allowed at any given time.
		 */
		if (dev->myriblk_req) {
			printk(KERN_EMERG "%s:%d: -- bailing out", \
				__func__, __LINE__);
			break;
		}
		if (! blk_fs_request(req)) {
			printk(KERN_NOTICE "Skip non-fs request\n");
			end_request(req, 0);
			continue;
		}

		blkdev_dequeue_request(req); /* we are using an
						interrupt handler */
		dev->myriblk_req = req;
		gm_myriblk_submit_req(dev, req->sector, \
				req->current_nr_sectors, \
				req->buffer, rq_data_dir(req));

		/*
		 * A request is outstanding, no need to loop.
		 */
		break;
	}
}
```

Η συνάρτηση αυτή εξυπηρετεί όλες τις εισερχόμενες αιτήσεις που καταφθάνουν από το επίπεδο block, μία κάθε φορά. Αρχικά προμηθεύεται την αίτηση που βρίσκεται στην κορυφή της ουράς με τη συνάρτηση `elv_next_request`. Ελέγχει ότι δεν εξυπηρετείται ήδη άλλη αίτηση και πως η αίτηση αφορά μεταφορά blocks (`blk_fs_request`). Αν ισχύουν τα παραπάνω, διαβάζει τις τέσσερις παραμέτρους από τη δομή `request` και αναφέρονται στα δεδομένα που πρέπει να μεταφερθούν (εν. 3.3.2) και τις προωθεί στην `gm_myriblk_submit_req`, η οποία θα επιβαρυνθεί να τις μεταφέρει (με τις κατάλληλες τροποποιήσεις) στον προσαρμογέα. Στο σημείο αυτό παρατηρούμε δύο σημαντικές λεπτομέρειες, τις οποίες απλά αναφέρουμε και ο ρόλος τους θα γίνει κατανοητός στην επερχόμενη ενότητα συγχρονισμού. Η πρώτη είναι πως αφαιρούμε την αίτηση από την

ουρά πριν την `gm_myriblk_submit_req` και η δεύτερη ότι δεν ολοκληρώνουμε την αίτηση με καμία συνάρτηση ολοκλήρωσης (σαν αυτές που παρουσιάσαμε στην εν. 2.1.3), αλλά αφού επιστρέψει η `gm_myriblk_submit_req` επιστρέφει και η συνάρτηση request αφήνοντας την αίτηση να εκκρεμεί στο σύστημα. Στο σημείο αυτό, ολοκληρώνεται η υλοποίηση της διεπαφής του οδηγού μας με το επίπεδο block του πυρήνα και είμαστε σε θέση να ανταποκριθούμε σε οποιοδήποτε αίτημα.

4.3.3 Διεπαφή οδηγού με το firmware (MCP)

Έχοντας ολοκληρώσει την διεπαφή πυρήνα - οδηγού, προχωρούμε στην διεπαφή του οδηγού με το firmware (GM MCP). Στο σχεδιασμό αναφέραμε ότι έχοντας τις τέσσερις παραμέτρους από τη δομή `request` διαθέσιμες, ο οδηγός γνωρίζει πλέον όλα όσα χρειάζονται για την ακριβή μεταφορά των δεδομένων. Πράγματι, είδαμε ότι η συνάρτηση `gm_myriblk_request` περνάει τις τέσσερις παραμέτρους αυτές στη συνάρτηση `gm_myriblk_submit_req` που είναι πλέον υπεύθυνη να τις προωθήσει με τον κατάλληλο τρόπο στον προσαρμογέα. Η συνάρτηση `gm_myriblk_submit_req` φαίνεται παρακάτω.

Στην ενότητα 3.3.2 περιγράψαμε την υπάρξη μίας κοινής δομής μεταξύ GM module και GM firmware. Η δομή αυτή ονομάζεται `gm_lanai_globals`, εδράζεται στην αρχή της μνήμης SRAM του προσαρμογέα και είναι προσβάσιμη και από τις δύο πλευρές, μέσω ειδικών συναρτήσεων που υλοποιεί το περιβάλλον GM. Για να έχουμε τη δυνατότητα να χρησιμοποιήσουμε αυτή τη λειτουργικότητα που προσφέρει το GM, έχουμε τροποποιήσει τη δομή `gm_lanai_globals`, προσθέτοντας στο εσωτερικό της μία δομή που ονομάζουμε `gm_myriblk_glb`. Η δομή αυτή είναι εφοδιασμένη με μεταβλητές που αφορούν στην επικοινωνία του οδηγού MyriBLK με το firmware. Οι μεταβλητές αυτές έχουν να κάνουν τόσο με μεταφορά συγκεκριμένων δεδομένων προς τον προσαρμογέα (όπως είναι οι τέσσερις προαναφερθέντες παράμετροι), όσο και με σημάτων επικοινωνίας και συγχρονισμού μεταξύ των δύο πλευρών.

```
static void
gm_myriblk_submit_req(gm_myriblk_dev_t *dev, unsigned long sector, \
                      unsigned long nsect, char *buffer, int write)
{
           unsigned long offset = sector*GM_MYRIBLK_KERNEL_SECTOR_SIZE;
           unsigned long nbytes = nsect*GM_MYRIBLK_KERNEL_SECTOR_SIZE;

           if (( offset + nbytes) > dev->size) {
                      printk(KERN_NOTICE "myriblk: Trying to access          \
                             beyond end of device                            \
                             [ %ld(offset) %ld(nbytes) ] size is:            \
                             %d\n", offset, nbytes, dev->size);
                      return;
           }
            if (write) {
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.start_sector, \
                                                sector);
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.num_sectors,  \
                                                nsect);
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.trf_buffer,   \
                                                virt_to_phys(buffer));
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.cur_status,   \
                                                0);
                      GM_STBAR();
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                 gm_myriblk_glb.cmd, 1);
           /* EDW XREIAZETAI NA PERIMENOUME MEXRI NA TELEIWSEI TO MCP
           K NA KANEI TO cur_status = 1 = NOT_BUSY */
                      //while ( gm_read_lanai_global_u32(dev->dev_to_is,     \
                                 gm_myriblk_glb.cur_status) == 0 ) {}
            } else {
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.start_sector, \
                                                sector);
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.num_sectors,  \
                                                nsect);
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.trf_buffer,   \
                                                virt_to_phys(buffer));
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.cur_status,   \
                                                0);
                      GM_STBAR();
                      gm_write_lanai_global_u32(dev->dev_to_is,              \
                                                gm_myriblk_glb.cmd, 2);
           /* EDW XREIAZETAI NA PERIMENOUME MEXRI NA TELEIWSEI TO MCP
            K NA KANEI TO cur_status = 1 = NOT_BUSY */
                      //while ( gm_read_lanai_global_u32(dev->dev_to_is,     \
                                  gm_myriblk_glb.cur_status) == 0 ) {}
            }
}
```

Κατά την υλοποίηση, αρχικά ελέγχουμε αν τα δεδομένα στα οποία αναφέρεται η αίτηση βρίσκονται εντός των ορίων της συσκευής μας. Σε διαφορετική περίπτωση επιστρέφουμε με μήνυμα λάθους. Επισημαίνουμε ότι δουλεύουμε σε μία σχεδιαστική φιλοσοφία κατά την οποία όλοι οι έλεγχοι και μετατροπές των δεδομένων σε κατάλληλες κλίμακες γίνονται από την πλευρά του οδηγού. Τη στιγμή που η αίτηση φτάσει στο firmware

ξεκινά η μεταφορά δεδομένων, χωρίς να χρειαστεί το firmware να κάνει οποιαδήποτε άλλη ενέργεια. Το firmware θεωρεί ότι η αίτηση που έλαβε ικανοποιεί όλες τις αναγκαίες προδιαγραφές. Αφού λοιπόν ελέγξουμε τα προβλεπόμενα όρια της συσκευής και διαπιστώσουμε ότι η αίτηση μπορεί να ικανοποιηθεί, περνάμε αρχικά τις τρεις από τις τέσσερις παραμέτρους στην δομή `gm_lanai_globals`. Οι παράμετροι θέτονται στην δομή έμμεσα με τη χρήση της συνάρτησης `gm_write_lanai_global_u32`, την οποία υλοποιεί το GM και πραγματοποιεί όλες τις απαραίτητες ενέργειες ώστε να μεταφερθούν σωστά οι μεταβλητές στη συγκεκριμένη δομή (π.χ. μετατροπή από little σε big endian, alignment κτλ). Οι παράμετροι αυτοί είναι: το πρώτο sector προς μεταφορά, ο συνολικός αριθμός sectors και η θέση μνήμης από/στην οποία θα διαβάσει/γράψει η συσκευή. Οι δύο πρώτες παράμετροι προωθούνται όπως λαμβάνονται, αφού δεν χρειάζεται καμία μετατροπή. Δεν μπορεί να γίνει το ίδιο και με την τρίτη παράμετρο που αφορά στη διεύθυνση μεταφοράς. Η διεύθυνση αυτή (που έχουμε εξάγει από τη δομή request) είναι μία εικονική διεύθυνση, την οποία δεν μπορεί να χρησιμοποιήσει το υλικολογισμικό, από τη στιγμή που η μεταφορά των δεδομένων θα γίνει με μέθοδο DMA. Για το λόγο αυτό, πριν τη μεταφέρουμε στην πλευρά του προσαρμογέα, την μετατρέπουμε στην αντίστοιχη φυσική διεύθυνση με τη συνάρτηση `virt_to_phys(buffer)`. Μόλις βεβαιωθούμε ότι η μεταφορά των παραμέτρων αυτών ολοκληρώθηκε (`GM_STBAR()`), μεταφέρουμε και την τελευταία παράμετρο που ορίζει το είδος της μεταφοράς. Σε περίπτωση εγγραφής η μεταβλητή `cmd` που έχουμε ορίσει στη δομή παίρνει την τιμή 1, διαφορετικά την τιμή 2. Η σύμβαση αυτή είναι γνωστή στο firmware, το οποίο αναγνωρίζοντας την μεταβλητή θα πράξει τις αντίστοιχες ενέργειες μεταφοράς.

Τέλος, ο ρόλος της μεταβλητής `cur_status` αλλά και ο κώδικας που έχει χαρακτηριστεί ώς σχόλια, θα γίνει ξεκάθαρος στην ενότητα 4.5 που αναλύεται το θέμα του συγχρονισμού. Ο κώδικας που παρουσιάζεται κατα την υλοποίηση είναι ο τελικός κώδικας, με τον συγχρονισμό των δύο πλευρών να γίνεται με τη χρήση διακοπών συστήματος. Έτσι, από τη στιγμή που μεταφέρεται και η τελευταία μεταβλητή στη δομή `gm_lanai_globals`, η συνάρτηση `gm_myriblk_submit_req` επιστρέφει και η αίτηση έχει μεταφερθεί επιτυχώς στην πλευρά του προσαρμογέα. Παρατηρώντας και την προηγούμενη ενότητα, διαπιστώνουμε ότι μετά την `gm_myriblk_submit_req`, επιστρέφει και η συνάρτηση `gm_myriblk_request` και ο οδηγός τερματίζει προσωρινά την αλληλεπίδραση του με τον πυρήνα. Τον έλεγχο της συγκεκριμένης εκκρεμούς αίτησης αναλαμβάνει από το σημείο αυτό το υλικολογισμικό (GM MCP).

4.4 Υλοποίηση από την πλευρά του προσαρμογέα (NIC)

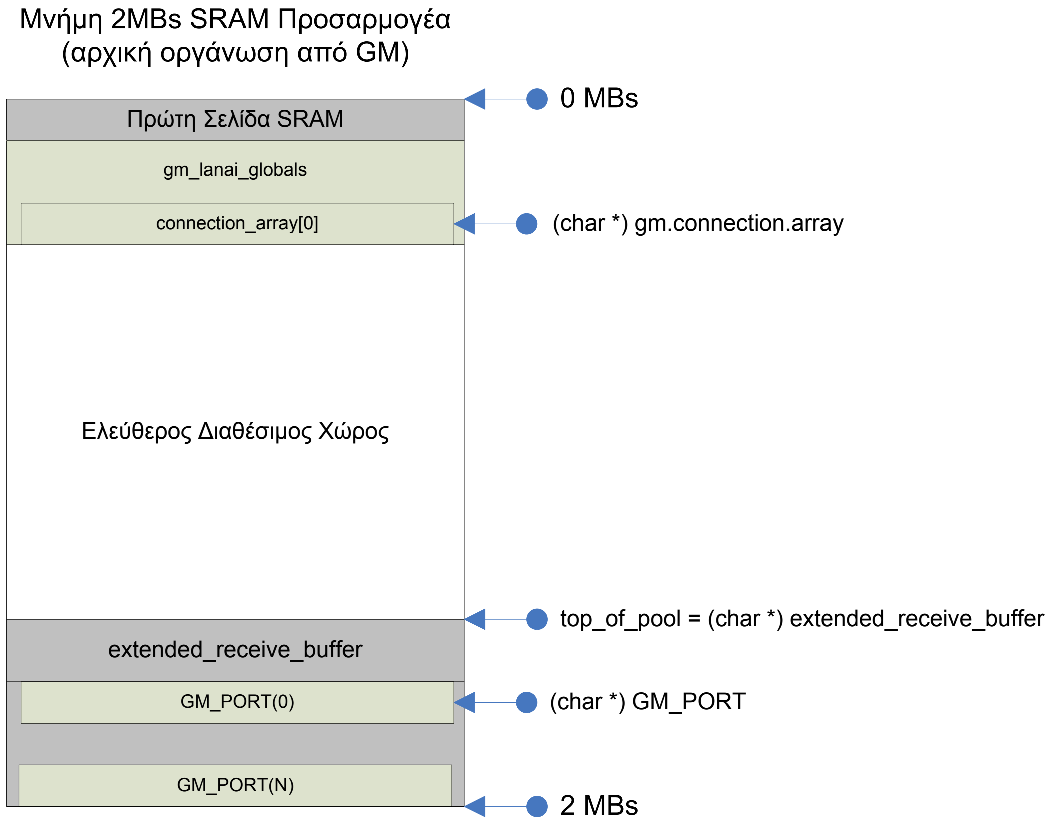
Στην ενότητα αυτή, τροποποιούμε κατάλληλα το υλικολογισμικό (firmware ή GM MCP) που εκτελείται στον μικροεπεξεργαστή Lanai του προσαρμογέα, με τέτοιον τρόπο ώστε να του δίνεται η δυνατότητα να εξυπηρετήσει τις εισερχομένες αιτήσεις και να προσομοιώσει ταυτόχρονα μία μονάδα δίσκου πάνω στη μνήμη SRAM του προσαρμογέα. Στην αντίστοιχη ενότητα σχεδιασμού, χωρίσαμε τις ενέργειες που πρέπει να επιτελέσει το firmware σε δύο μέρη. Την αναδιοργάνωση της τοπικής μνήμης ώστε να δεσμευτεί ο χώρος ο οποίος θα προσομοιώσει τη μονάδα δίσκου και την καθεαυτή διαδικασία μετακίνησης δεδομένων και επικοινωνίας με τον οδηγό. Τη λογική αυτή ακολουθούμε και κατά την υλοποίηση.

4.4.1 Αναδιοργάνωση της μνήμης SRAM

Η μνήμη SRAM πρέπει να αναδιοργανωθεί με πολύ συγκεκριμένο τρόπο, όπως επιβάλλει η σχεδίαση μας στην Εν. 3.4.1. Κατά την υλοποίηση θα διατηρήσουμε όλους τους προκαθορισμένους χώρους που υλοποιεί το GM MCP και θα δεσμεύσουμε το μεγαλύτερο μέρος της ελεύθερης μνήμης που απομένει για την μονάδα δίσκου που θα υλοποιήσουμε. Όλα τα παραπάνω πρέπει να γίνουν διαφανώς για το GM, έτσι ώστε ο χώρος της συσκευής να είναι ουσιαστικά αόρατος στο GM και ορατός μόνο στον οδηγό μας (MyriBLK module). Η αρχική οργάνωση μνήμης του GM περιγράφηκε αναλυτικά στην Εν. 3.4.1. Την ξαναπαρουσιάζουμε με παρόντες όλους τους δείκτες που υλοποιεί το GM στο Σχ. 4.5, για να γίνει κατανοητός ο κώδικας που ακολουθεί.

Παρατηρούμε τα εξής: τελευταίο στοιχείο της δομής `gm_lanai_globals` είναι ένας πίνακας ενός στοιχείου τύπου `gm_connection_t (gm_connection_t array[1])`. Αυτό συμβαίνει για να έχει ένα συγκεκριμένο δείκτη το GM που να δείχνει στην αρχή της ελεύθερης διαθέσιμης μνήμης. Αντίστοιχα ο δείκτης `top_of_pool` υποδεικνύει το τέλος αυτής. Ο δείκτης `GM_PORT` δείχνει στην αρχή των `ports` και ο δείκτης `extended_-receive_buffer` υποδηλώνει την αρχή της αντίστοιχης περιοχής μνήμης. Τελικά το GM αντιλαμβάνεται την διαθέσιμη ελεύθερη μνήμη ώς τον χώρο που βρίσκεται μεταξύ `array[0]` και `top_of_pool` δηλαδή:

```
available_memory = top_of_pool - (char *) gm.connection.array;
```

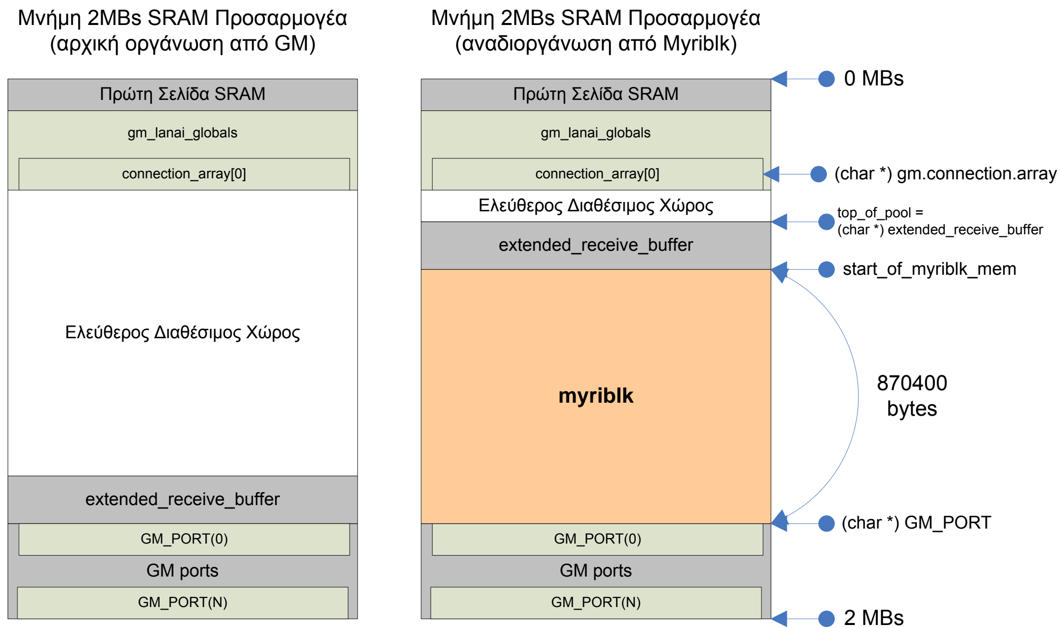


Σχήμα 4.5: *Αρχική Οργάνωση Μνήμης SRAM*

Οι δείκτες `top_of_pool` και `extended_receive_buffer` συμπίπτουν. Με τον κώδικα που ακολουθεί αναδιοργανώνουμε τη μνήμη έτσι ώστε να δεσμεύσουμε τον επιθυμητό χώρο μεταξύ `extended_receive_buffer` και `GM_PORT`. Μετά την αναδιοργάνωση οι δείκτες `top_of_pool` και `extended_receive_buffer` θα συνεχίσουν να συμπίπτουν και η νέα ελεύθερη διαθέσιμη μνήμη την οποία θα διαχειρίζεται το GM θα εξακολουθήσει να προσδιορίζεται με τον παραπάνω τύπο. Ο δείκτης `GM_PORT` θα παραμείνει ανέπαφος και έτσι οι αλλαγές θα είναι διαφανείς στο υπόλοιπο σύστημα. Ο νέος δείκτης που εισάγεται ονομάζεται `start_of_myriblk_mem` και δείχνει στην αρχή του χώρου μνήμης της συσκευής `myriblk`, δηλαδή του νέου χώρου που έχουμε δεσμεύσει και θα προσομοιώσει τη μονάδα δίσκου μας. Ο δείκτης αυτός είναι απαραίτητος, όπως θα δούμε στη συνέχεια, για τη διαχείρηση της συγκεκριμένης περιοχής μνήμης. Το τροποποιημένο κομμάτι κώδικα του MCP που αναδιοργανώνει κατάλληλα τη μνήμη SRAM παρουσιάζεται παρακάτω:

```
{
    char *top_of_pool;
    unsigned long gm_myriblk_memory_size = 870400;     /* cven  512 * 1700  */
    unsigned long available_myriblk_memory;           /* cven - part */
    unsigned long available_memory;
    unsigned long bitmap_len;
    unsigned long cached_pte_cnt;
    unsigned long connection_cnt;

    top_of_pool = (char *) GM_PORT;

    /*********************************************************************
     *
     * cven-part
     * memory allocation for the myriblk device to operate
     *
     *********************************************************************/

    top_of_pool -= gm_myriblk_memory_size;
    start_of_myriblk_mem = top_of_pool;
    available_myriblk_memory = (char *) GM_PORT - start_of_myriblk_mem;
    gm.gm_myriblk_glb.dev_size = available_myriblk_memory;

    /*********************************************************************/

#if LX
    /* Allocate space for the extended receive buffers from the top of
       the pool used to allocate gm.connection.array and
       gm.page_hash.cache.entry[], below. */

    extended_receive_buffer
      = &(((struct gm_lx_extended_receive_buffer (*)[2]) top_of_pool)
            [-gm.num_packet_ifcs]);
    if ((char *) extended_receive_buffer < (char *) gm.connection.array)
      {
           status = GM_OUT_OF_MEMORY;
           goto abort;
      }
    top_of_pool = (char *) extended_receive_buffer;
#endif

    available_memory = top_of_pool - (char *) gm.connection.array;
    gm_highmem_compute_layout (available_memory, &connection_cnt,
                                    &cached_pte_cnt, &bitmap_len, 1<<16);

    *(char **)&gm.page_hash.cache.entry
      = ((char *) &gm.connection.array[connection_cnt]
            + bitmap_len);
    GM_STBAR();
    gm.connection.max_id = connection_cnt - 1;
    GM_MAX_PAGE_HASH_CACHE_INDEX = cached_pte_cnt - 1;
}
```

Κατά τη σχεδίαση επιλέξαμε το μέγεθος της μονάδας δίσκου περίπου στο 1MB. Μετά από πειραματική εφαρμογή του παραπάνω κώδικα, καταλήξαμε ότι ο μεγαλύτερος χώρος που μπορούμε να δεσμεύσουμε, στην περίπτωση που θέλουμε να διατηρήσουμε και τις υπόλοιπες περιοχές που υλοποιεί το GM είναι 870400 bytes (0.9MB), πολύ κοντά δηλαδή στην αρχική μας υπόθεση. Ο αριθμός αυτός προκύπτει ως ακέραιο πολλαπλάσιο του μεγέθους sector της συσκευής μας, το οποίο επιλέξαμε στα 512 bytes. Τελικά η μονάδα δίσκου θα αποτελείται από 1700 τέτοια sectors, συνεχόμενα πάνω στη μνήμη SRAM του προσαρμογέα, που ξεκινούν από το σημείο που δείχνει ο δείκτης `start_-`



Σχήμα 4.6: *Σύγκριση της SRAM πριν και μετά την αναδιοργάνωση από MyriBLK*

`of_myriblk_mem`. Επίσης το ακριβές αυτό μέγεθος, αφού υπολογιστεί τίθεται και στην αντίστοιχη μεταβλητή της δομής `gm_lanai_globals`. Η τελική διαμόρφωση της μνήμης SRAM παρουσιάζεται στο Σχ. 4.6 και παραλληλίζεται ταυτόχρονα με την αρχική, ώστε να διαφανούν καθαρά όλες οι περιοχές.

4.4.2 Μετακίνηση δεδομένων και επικοινωνία με οδηγό

Έχοντας αναδιοργανώσει τη μνήμη SRAM, το firmware έχει αποκτήσει τον ελεύθερο χώρο που χρειάζεται, για να τον χρησιμοποιήσει ώς μονάδα αποθήκευσης. Αφού κάνει όλες τις αναγκαίες αρχικοποιήσεις, πρέπει στο κατάλληλο σημείο να τροποποιηθεί ο κώδικάς του, έτσι ώστε να μπορεί να αντιμετωπίσει τις εισερχόμενες αιτήσεις. Γνωρίζουμε μέχρι στιγμής ότι το firmware επικοινωνεί με τον οδηγό μέσω της δομής `gm_lanai_globals`, η οποία είναι αποθηκευμένη στην αρχή της SRAM. Όλα τα στοιχεία της αίτησης βρίσκονται στις ήδη ορισμένες μεταβλητές της δομής και το firmware το μόνο που καλείται να κάνει είναι να τις διαβάσει. Βέβαια, στο σημείο αυτό ερχόμαστε αντιμέτωποι με ένα πρόβλημα υλοποίησης. Χρειαζόμαστε ένα μηχανισμό, τον οποίο θα χρησιμοποιήσει το υλικολογισμικό για να διαπιστώσει πότε οι μεταβλητές που διαβάζει από τη δομή `gm_lanai_globals` αναφέρονται σε καινούρια αίτηση. Στην περίπτωση που το υλικολογισμικό δεν μπορεί να ξεχωρίσει αν οι μεταβλητές που προ-

σπελαύνει αναφέρονται σε αίτηση που ήδη έχει ικανοποιήσει ή όχι, τότε οδηγούμαστε σε αδιέξοδο και λανθασμένη λειτουργία. Για να ξεπεράσουμε το πρόβλημα χρησιμοποιούμε την ήδη ορισμένη μεταβλητή `gm_lanai_globals.gm_myriblk_glb.cmd`. Στην Εν. 4.3.3 σημειώσαμε πως η μεταβλητή αυτή παίρνει 2 τιμές ανάλογα με την λειτουργία εγγραφής ή ανάγνωσης. Ορίζουμε τρείς ενέργειες στις οποίες μπορεί να επιδοθεί το υλικολογισμικό: εγγραφή, ανάγνωση και αδράνεια. Για να αποφασίσει το υλικολογισμικό σε ποιά από τις τρείς θα λειτουργήσει ελέγχει την μεταβλητή `cmd`. Είδαμε στην Εν. 4.3.3 πως ο οδηγός θέτει την τιμή 1 για εγγραφή και την τιμή 2 για ανάγνωση. Προσθέτουμε την τιμή 0 για αδράνεια. Με αυτόν τον τρόπο το υλικολογισμικό ελέγχει συνεχώς την τιμή αυτής της μεταβλητής και μόλις διαπιστώσει αλλαγή από `0` σε `1` ή `2` επιτελεί την αντίστοιχη ενέργεια. Μόλις ολοκληρώσει, επαναφέρει την τιμή της `cmd` στο `0`. Έτσι, ικανοποιεί πάντα την σωστή τρέχουσα αίτηση. Ο κώδικας που υλοποιεί την πραγματική μεταφορά δεδομένων ακολουθεί στο τέλος της υποενότητας.

Παρατηρούμε ότι έχουμε έναν επαναληπτικό βρόγχο που ελέγχει την τιμή της μεταβλητής `cmd` και ανάλογα με την τιμή προχωρούμε στην αντίστοιχη ενέργεια, ακριβώς όπως περιγράψαμε προηγουμένως. Αρχικά σηματοδοτούμε την τρέχουσα κατάσταση στην οποία βρίσκεται το firmware και θέτουμε κάποιες μεταβλητές ελέγχου. Στην περίπτωση εγγραφής καλούμε την συνάρτηση `synchronous_SDMA` η οποία υλοποιείται από το MCP και χειρίζεται την μία από τις τρεις μηχανές DMA του προσαρμογέα. Η συνάρτηση αυτή χρησιμοποιείται από το υλικολογισμικό κατά την διαδικασία αποστολής ενός πακέτου, για να μετακινήσει δεδομένα από την κεντρική μνήμη του συστήματος στην τοπική SRAM. Συγκεκριμένα η συνάρτηση λαμβάνει τρεις μεταβλητές: ένα δείκτη στη διεύθυνση κεντρικής μνήμης απ᾽ όπου θα λάβει τα δεδομένα (`dma_pointer`), ένα δείκτη στο σημείο που πρέπει να μεταφερθούν τα δεδομένα στην τοπική SRAM και το συνολικό μέγεθος των δεδομένων. Η συνάρτηση αυτή μπορεί να χρησιμοποιηθεί μόνο στην περίπτωση που η κάθε αίτηση περιλαμβάνει ένα φυσικό τμήμα, δηλαδή όλα τα δεδομένα βρίσκονται σε συνεχόμενες θέσεις στην κεντρική μνήμη. Αυτό όμως είναι ήδη εξασφαλισμένο, κατά την αρχικοποίηση της ουράς αιτήσεων από τον οδηγό μας. Αντίστοιχη διαδικασία ακολουθείται και στην περίπτωση της ανάγνωσης. Η συνάρτηση που χρησιμοποιούμε είναι η `synchronous_RDMA` που χρησιμοποιείται από το MCP κατά την παραλαβή πακέτων και επιτελεί την αντίστροφη λειτουργία, μεταφέρει δηλαδή δεδομένα από την μνήμη SRAM στην κεντρική μνήμη συστήματος, χρησιμοποιώντας τις αντίστοιχες μεταβλητές. Αφού ολοκληρωθεί η μεταφορά των δεδομένων,

η μεταβλητή `cmd` τίθεται στην τιμή `0`, ώστε να αδρανοποιηθεί η εξυπηρέτηση αιτήσεων μέχρι την εμφάνιση κάποιας νέας. Ο κώδικας που ακολουθεί μετά από αυτό το σημείο, και μέχρι να τερματιστεί ο βρόγχος αφορά στον συγχρονισμό του υλικολογισμικού (μετά την ολοκλήρωση της μεταφοράς), με τον οδηγό και θα αναλυθεί στην ενότητα που ακολουθεί.

```
switch (gm.gm_myriblk_glb.cmd) {
          case 0:    /* idle */
                     break;
          case 1:    /* write */
                     gm.gm_myriblk_glb.cur_status = 0;   /* 0 = BUSY */
                     gm.gm_myriblk_glb.test1 = 91;
                     /* edw ginetai to actual transfer (dma) */
                     // synchronous_SDMA
                     //(gm_dp_t ear, gm_lp_t lar, gm_u32_t len)
                     // synchronous_SDMA
                     //(apo edw    , ekei       , length)
                     //myriblk_dma = gm.gm_myriblk_glb.trf_buffer;
                     gm.gm_myriblk_glb.dma_pointer = gm.gm_myriblk_glb.trf_buffer;
                     synchronous_SDMA(gm.gm_myriblk_glb.dma_pointer,       \
                                      gm.gm_myriblk_glb.mem_start +        \
                                      (512*gm.gm_myriblk_glb.start_sector), \
                                      512 * gm.gm_myriblk_glb.num_sectors);
                     gm.gm_myriblk_glb.cmd = 0;
                     /* the interrupt */
                     {
                     union gm_interrupt_descriptor d;
                     d.type = GM_MYRIBLK_INTERRUPT;
                     gm_interrupt (d);
                     }
                     gm.gm_myriblk_glb.cur_status = 1;  /* 1 = NOT_BUSY */
                     break;
          case 2:    /* read */
                     gm.gm_myriblk_glb.cur_status = 0;
                     gm.gm_myriblk_glb.test2 = 92;
                     /* edw ginetai to actual transfer (dma) */
                     // synchronous_RDMA
                     //(gm_lp_t lar, gm_dp_t ear, gm_u32_t len)
                     // synchronous_RDMA
                     //(apo edw  , ekei          , len)
                     //myriblk_dma = gm.gm_myriblk_glb.trf_buffer;
                     gm.gm_myriblk_glb.dma_pointer = gm.gm_myriblk_glb.trf_buffer;
                     synchronous_RDMA(gm.gm_myriblk_glb.mem_start +        \
                                      (512*gm.gm_myriblk_glb.start_sector), \
                                      gm.gm_myriblk_glb.dma_pointer,       \
                                      512 * gm.gm_myriblk_glb.num_sectors);
                     gm.gm_myriblk_glb.cmd = 0;
                     /* the interrupt */
                     {
                      union gm_interrupt_descriptor d;
                      d.type = GM_MYRIBLK_INTERRUPT;
                      gm_interrupt (d);
                     }
                     gm.gm_myriblk_glb.cur_status = 1;
                     break;
          default:
                     gm.gm_myriblk_glb.test2 = 95;
                     gm_assert (0);
                     break;
}
```

4.5 Συγχρονισμός των δύο πλευρών (Host - NIC)

Φτάνοντας στο σημείο αυτό, η αίτηση που έχει παραχθεί από κάποια εφαρμογή, έχει περάσει μέσω του επιπέδου block του Linux στον οδηγό μας (Myriblk module), αυτός την έχει προωθήσει με τον κατάλληλο τρόπο στον προσαρμογέα δικτύου και ο προσαρμογέας με τη σειρά του έχει πραγματοποιήσει επιτυχώς τη μεταφορά των δεδομένων. Κάθε είδος block αίτησης έχει στην ουσία εξυπηρετηθεί. Το μόνο που απομένει είναι οι κατάλληλες λειτουργίες ολοκλήρωσης από την πλευρά του οδηγού, έτσι ώστε να ενημερωθεί το επιπέδο block για την εξέλιξη της μεταφοράς, που θα του επιτρέψει να προωθήσει την επόμενη αίτηση. Όπως αναλύσαμε και κατά τη σχεδίαση, αυτό μπορεί να πραγματοποιηθεί με δύο τρόπους. Στην τελική υλοποίηση, από την οποία προέρχονται και όλα τα κομμάτια κώδικα που παρουσιάσαμε στο κεφάλαιο αυτό, χρησιμοποιούμε συγχρονισμό με τη χρήση διακοπών συστήματος. Πριν όμως περιγράψουμε αυτή τη μέθοδο, θα αναφερθούμε με συντομία στη μέθοδο περιοδικού ελέγχου, αφού αυτή ήταν η πρώτη που υλοποιήσαμε κατά την πειραματική μας εξέταση.

4.5.1 Συγχρονισμός Περιοδικού Ελέγχου (Polling)

Για να υλοποιηθεί αυτή η μέθοδος χρειάζεται όπως είδαμε στο σχεδιασμό, να προωθήσει ο οδηγός την αίτηση στον προσαρμογέα και να αναμείνει μέχρι να λάβει απάντηση από το υλικολογισμικό για την ολοκλήρωση της μεταφοράς. Αφού λάβει την απάντηση ολοκληρώνει την αίτηση και η συνάρτηση request επιστρέφει. Για να πραγματοποιηθούν τα παραπάνω αρχικά ορίζουμε τις δύο καταστάσεις `BUSY` και `NOT BUSY` που αναφέραμε κατά το σχεδιασμό. Αυτό γίνεται μέσω της μεταβλητής `gm_lanai_globals.gm_myriblk_glb.cur_status` της κοινής δομής `gm_lanai_globals`. Όταν το υλικολογισμικό είναι απασχολημένο και όχι διαθέσιμο να εξυπηρετήσει αιτήσεις (`BUSY`) η μεταβλητή παίρνει την τιμή `0`. Στην αντίθετη περίπτωση (`NOT BUSY`) παίρνει την τιμή `1`. Έτσι η συνάρτηση `gm_myriblk_submit_req`, αφού ορίσει τις μεταβλητές των τεσσάρων παραμέτρων, εισέρχεται σε έναν βρόγχο που ελέγχει την τιμή της μεταβλητής `cur_status`, και αναμένει μέχρι να αλλάξει η συγκεκριμένη τιμή. Μόλις η τιμή αλλάξει (από το υλικολογισμικό), σημαίνει πως η μεταφορά ολοκληρώθηκε και μπορεί να τερματίσει την αίτηση. Ο κώδικας που υλοποιεί αυτή τη λειτουργία φαίνεται ως σχόλιο στην συνάρτηση `gm_myriblk_submit_req` της εν. 4.3.3 και είναι ο εξής:

```
while ( gm_read_lanai_global_u32(dev->dev_to_is,
gm_myriblk_glb.cur_status) == 0 ) {}
```

για κάθε περίπτωση (εγγραφής ή ανάγνωσης) ξεχωριστά. Αφού επιστρέψει η `gm_myriblk_-submit_req`, πρέπει η συνάρτηση `gm_myriblk_request` να ολοκληρώσει την αίτηση και να προχωρήσει στην επόμενη. Αυτό γίνεται με μία ελαφρώς τροποποιημένη μέθοδο request που φαίνεται παρακάτω:

```
static void
gm_myriblk_request(struct request_queue *q)
{
		struct request *req;

		while ((req = elv_next_request(q)) != NULL) {
				gm_myriblk_dev_t *dev = req->rq_disk->private_data;
				if (! blk_fs_request(req)) {
						printk(KERN_NOTICE "Skip non-fs request\n");
						end_request(req, 0);
						continue;
				}
				gm_myriblk_transfer(dev, req->sector, req->current_nr_sectors,
				req->buffer, rq_data_dir(req));
				end_request(req, 1);
		}
}
```

Η μόνη ουσιαστική διαφορά από τη μέθοδο που παρουσιάστηκε στην εν. 4.3.2 είναι η ύπαρξη της `end_request(req, 1)` στο τέλος, και η επανάληψη του βρόγχου. Η `end_-request` κάνει όλες τις διαδικασίες τερματισμού της αίτησης και ενημερώνει το επίπεδο block. Επίσης παραλείπεται η αφαίρεση της αίτησης από την ουρά με την `blkdev_-dequeue_request(req)` η ύπαρξη της οποίας χρησιμεύει μόνο κατά τη μέθοδο των διακοπών συστήματος, όπως θα παρακολουθήσουμε στη συνέχεια. Από την πλευρά του υλικολογισμικού ο κώδικας παραμένει ο ίδιος με αυτόν της ενότητας 4.4.2, μόνο που παραλείπεται το κομμάτι που πυροδοτεί τη διακοπή συστήματος (στον κώδικα φαίνονται τα ακριβή σημεία στα οποία αλλάζει τιμή η μεταβλητή `cur_status` για να συγχρονίσει τη διαδικασία):

```
/* the interrupt */
{
		union gm_interrupt_descriptor d;
		d.type = GM_MYRIBLK_INTERRUPT;
		gm_interrupt (d);
}
```

4.5.2 Συγχρονισμός με χρήση διακοπών συστήματος (Interrupts)

Ο κώδικας που έχει παρουσιαστεί καθόλη τη διάρκεια του κεφαλαίου αυτού, είναι μέρος της τελικής υλοποίησης της συσκευής myriblk. Γι' αυτό το λόγο είναι δομημένος με τέτοιο τρόπο, ώστε να συγχρονίσει τις δύο πλευρές με τη χρήση διακοπών συστήματος (interrupts). Παρατηρήσαμε, σε αντίθεση με τη μέθοδο polling, ότι η συνάρτηση request προμηθεύεται την τρέχουσα αίτηση από την ουρά αιτήσεων, την προωθεί στο υλικολογισμικό μέσω της συνάρτησης `gm_myriblk_submit_req` και μόλις αυτή ολοκληρώσει την εγγραφή των παραμέτρων στην δομή `gm_lanai_globals`, επιστρέφουν και οι δύο συναρτήσεις. Έτσι το module MyriBLK απομακρύνεται από τη CPU (εξάλλου αυτός είναι ο αρχικός σκοπός του σχεδιασμού μας) και τη θέση του παίρνει μία άλλη διεργασία. Αποτέλεσμα αυτού του σχεδιασμού, είναι να παραμείνει εκκρεμής στο σύστημα η αίτηση που μόλις προωθήθηκε στο υλικολογισμικό. Η αίτηση θα ολοκληρωθεί όταν το υλικολογισμικό διακόψει τον επεξεργαστή για να τον ενημερώσει για το πέρας της μεταφοράς δεδομένων.

Όπως περιγράψαμε και στην εν. 2.1.5 για να εξυπηρετηθεί μια διακοπή συστήματος, η πρώτη απαίτηση είναι να δηλωθεί ο κατάλληλος χειριστής διακοπών στο σύστημα. Ο χειριστής αυτός θα κληθεί από τον πυρήνα να εξυπηρετήσει τη διακοπή του υλικού, μόλις το σύστημα την αντιληφθεί. Στην υλοποίησή μας, θα τροποποιήσουμε κατάλληλα τον υπάρχοντα μηχανισμό εξυπηρέτησης διακοπών του GM, προσθέτοντάς του τη λειτουργικότητα που απαιτείται για τη σωστή λειτουργία της συσκευής `myriblk`.

Το GM υλοποιεί ένα μηχανισμό διακοπών που εξυπηρετεί πολλά διαφορετικά είδη διακοπών με ένα μοναδικό χειριστή (`void gm_handle_claimed_interrupt()`). Αρχικά ορίζει όλους τους διαφορετικούς τύπους διακοπών που μπορεί να εξυπηρετήσει. Μετά ορίζει μία δομή τύπου union (`union gm_interrupt_descriptor`) η οποία είναι ορατή από τον χειριστή διακοπών και προσβάσιμη από το υλικολογισμικό, στην οποία αποθηκεύεται κάθε φορά ο τρέχων τύπος διακοπής. Όταν το υλικολογισμικό αποφασίσει πως πρέπει να στείλει μία διακοπή στο σύστημα, θέτει πρώτα το είδος της διακοπής στο union και στη συνέχεια διακόπτει με ένα σήμα τον επεξεργαστή. Ο επεξεργαστής αντιλαμβάνεται τη διακοπή, τη συνδέει με το GM και τρέχει τον χειριστή του GM. Ο χειριστής ελέγχει τον τύπο διακοπής που πρέπει να εξυπηρετήσει προσπελάζοντας τον `gm_interrupt_descriptor` και αφού τον έχει αναγνωρίσει, εκτελεί την κατάλληλη λειτουργία εξυπηρέτησης.

Για να ενσωματώσουμε τον χειρισμό της δικής μας διακοπής στον ήδη υλοποιημένο μηχανισμό, ορίζουμε αρχικά έναν πρόσθετο τύπο διακοπής (`GM_MYRIBLK_INTERRUPT`). Τροποποιούμε το `union gm_interrupt_descriptor` έτσι ώστε να συμπεριλάβει αυτόν τον τύπο και τελικά τροποποιούμε τον χειριστή διακοπών ώστε να μπορεί να αναγνωρίσει αυτό το είδος της διακοπής και να το χειριστεί κατάλληλα:

```
enum gm_interrupt_type
{
  GM_NO_INTERRUPT,
  GM_INTERRUPTS_UNINITIALIZED,
  GM_PAUSE_INTERRUPT,
  GM_COMMAND_COMPLETE_INTERRUPT,
  GM_WAKE_INTERRUPT,
  GM_PRINT_INTERRUPT,
  GM_FAILED_ASSERTION_INTERRUPT,
  GM_WRITE_INTERRUPT,
  GM_WAIT_INTERRUPT,
  GM_BOGUS_RECV_INTERRUPT,
  GM_BOGUS_SEND_INTERRUPT,
  GM_YP_WAKE_INTERRUPT,
  GM_NODE_ID_TRANSLATION_INTERRUPT,
  GM_MYRIBLK_INTERRUPT  /* cven - part */
};
```

```
union gm_interrupt_descriptor
{
  volatile gm_u32_n_t type;
  struct
  {
    volatile gm_u32_n_t type;
    volatile gm_u32_n_t port;
  }
  wake;

           *
           *
           *
   OTHER GM STRUCTS HERE
           *
           *

  struct                         /* cven - part */
  {
    volatile gm_u32_n_t type;    /* | */
    gm_u32_n_t status;           /* | */
  }
  gm_myriblk;                    /* cven - part */
  struct
  {
    gm_u8_n_t _reserved[23]; /* make size of union be 24 bytes */
    gm_u8_n_t ready;
  }
  last_byte;
};
```

```
void
gm_handle_claimed_interrupt (gm_instance_state_t * is)
{
  enum gm_interrupt_type type;

  gm_assert (is);
  gm_assert (is->lanai.running);
  gm_assert (is->lanai.special_regs);

  type = (enum gm_interrupt_type) gm_ntoh_u32 (is->interrupt->type);

  GM_PRINT (GM_INTR_PRINT,
              ("*** got an interrupt to handler %d  type = %d\n",
               is->id, type));

  switch (type)
    {

    case GM_COMMAND_COMPLETE_INTERRUPT:
      is->command_done = 1;
      gm_arch_wake (&is->command_sync);
      break;

           *
           *
     case  FOR EVERY SINGLE INTERRUPT TYPE
           DEFINED IN gm_interrupt_type
           *
           *

    /****************************************************************************
     * cven - part
     *
     * the myriblk interrupt handling
     ****************************************************************************/

     case GM_MYRIBLK_INTERRUPT:
      {
           gm_myriblk_handle_interrupt(is);
           break;
     }

    /*****************************************************************************/

    default:
      GM_WARN (("gm_intr: unit=%d got an unrecognized interrupt of type 0x%x\n",
                      is->id, type));
      gm_disable_lanai (is);
      break;
    }

  /* clear last_byte.ready before we allow the MCP to begin to
     interrupt again by DMA'ing last_byte.ready to the host */

  is->interrupt->last_byte.ready = gm_hton_u8 (0);
  GM_STBAR ();

  /* Tell the LANai we are ready for the next interrupt. */

  GM_PRINT (GM_INTR_PRINT, ("gm_intr: done with handling \n"));
  gm_write_lanai_global_u32 (is, interrupt.type, GM_NO_INTERRUPT);

  GM_STBAR ();

  GM_PRINT (GM_INTR_PRINT, ("gm_intr: return from interrupt \n"));
}
```

Στον παραπάνω κώδικα αρχικά προσθέτουμε τον τύπο `GM_MYRIBLK_INTERRUPT` στο `enum gm_interrupt_type`. Στη συνέχεια τροποποιούμε το `union gm_interrupt_descriptor` προσθέτοντάς του την δομή `gm_myriblk`, αφού πρώτα έχουν προηγηθεί

όλα τα structs που ορίζει από μόνο του το GM (π.χ. το wake που φαίνεται στην αρχή). Αυτό το struct θα διαβάσει ο χειριστής για να αναγνωρίσει ότι η τρέχουσα διακοπή έχει προκληθεί από τη συσκευή myriblk. Τελικά, τροποποιούμε τον ίδιο τον χειριστή για να εξυπηρετηθεί η διακοπή.Ο χειριστής αφού αναγνωρίσει τον τύπο της διακοπής καλεί την συνάρτηση gm_myriblk_handle_interrupt(is), την οποία έχουμε υλοποιήσει μέσα στο MyriBLK module για να εξυπηρετήσει την διακοπή. Η συνάρτηση gm_myriblk_handle_interrupt(is) που χειρίζεται στην πραγματικότητα τη διακοπή φαίνεται στη συνέχεια:

```
void
gm_myriblk_handle_interrupt(gm_instance_state_t *is)
{
   unsigned long flags;

   if (is->gm_myriblk_dev_pnt == 0) {
        printk(KERN_EMERG "Woops! gm_myriblk_dev_pnt is NULL\n");
   } else {
   /* Lock the queue before manipulating it */
   spin_lock_irqsave(&is->gm_myriblk_dev_pnt->lock, flags);
   if (!end_that_request_first(is->gm_myriblk_dev_pnt->myriblk_req, 1, \
                              is->gm_myriblk_dev_pnt->myriblk_req->nr_sectors)) {
         end_that_request_last(is->gm_myriblk_dev_pnt->myriblk_req, 1); }
   is->gm_myriblk_dev_pnt->myriblk_req = NULL;
   spin_unlock_irqrestore(&is->gm_myriblk_dev_pnt->lock, flags); /*unlock the queue*/
   blk_run_queue(is->gm_myriblk_dev_pnt->queue);     /* rerun the request function */
                                                     /* to queue more requests */
   }
}
```

Για να φτάσουμε στην εκτέλεση αυτής της συνάρτησης, σημαίνει πως η μεταφορά των δεδομένων για κάποια αίτηση έχει ολοκληρωθεί και το υλικολογισμικό μας ενημέρωσε για το γεγονός. Εμείς άρα καλούμαστε να τερματίσουμε την αίτηση που μέχρι τώρα εκκρεμούσε και να προετοιμάσουμε τον οδηγό να δεχθεί μία νέα. Για να το πετύχουμε αυτό, κλειδώνουμε αρχικά το lock της συσκευής μας. Προσπελάζουμε την αίτηση που εκκρεμεί μέσω του αντίστοιχου στιγμιότυπου και του δείκτη myriblk_req (αφού ο χειριστής τρέχει σε interrupt context και δεν γνωρίζει ποιά είναι η εκκρεμής αίτηση) και την ολοκληρώνουμε εκτελώντας τις συναρτήσεις end_that_request_first και end_that_request_last. Σημειώνουμε την αίτηση ώς ολοκληρωμένη, θέτοντας τον δείκτη myriblk_req NULL και ξεκλειδώνουμε το lock. Τέλος, με τη συνάρτηση blk_run_queue επανεκτελούμε την συνάρτηση request για να εξυπηρετήσει νέα αίτηση της ουράς στην περίπτωση που αυτή υπάρχει. Να σημειώσουμε στο σημείο αυτό πως για να εκτελέσουμε την blk_run_queue πρέπει να έχει αφαιρεθεί πρώτα η ολοκληρωμένη αίτηση από την κορυφή της ουράς, ώστε να μην την επαναχρησιμοποιήσει η συνάρτηση request. Γι’ αυτό ακριβώς το λόγο η συνάρτηση gm_myriblk_request

τρέχει την `blkdev_dequeue_request(req)` που είδαμε στην εν. 4.3.2 και αποφύγαμε να εξηγήσουμε το ρόλο της σε εκείνο το σημείο της ανάλυσης. Με αυτόν τον τρόπο ολοκληρώνεται η διαδικασία επεξεργασίας μίας αίτησης με την μέθοδο των διακοπών συστήματος. Αναφέρουμε επίσης, ότι ο τρόπος με τον οποίο προκαλείται η διακοπή από το firmware, μόλις αυτό ολοκληρώσει τη μεταφορά, έχει ήδη φανεί στον κώδικα της εν. 4.4.2:

```
/* the interrupt */
{
   union gm_interrupt_descriptor d;
   d.type = GM_MYRIBLK_INTERRUPT;
   gm_interrupt (d);
}
```

Μετά το πέρας και του συγχρονισμού των δύο πλευρών, καταλήγουμε σε μία πλήρως λειτουργική συσκευή block που προσομοιώνει στο σύστημα μία μονάδα σκληρού δίσκού η οποία όμως εδράζεται στην τοπική μνήμη SRAM ενός προσαρμογέα δικτύου υψηλής επίδοσης. Στο επόμενο κεφάλαιο ακολουθεί η πειραματική αποτίμηση της συσκευής.

5

Πειραματική Αξιολόγηση

Στο κεφάλαιο αυτό γίνεται μία πειραματική αποτίμηση της συσκευής MyriBLK, που υλοποιήθηκε στα πλαίσια της παρούσας εργασίας. Σκοπός είναι η αξιολόγηση της συσκευής σε ένα πραγματικό σύστημα, η επισήμανση πιθανών λαθών υλοποίησης και η εξαγωγή χρήσιμων συμπερασμάτων. Υποβάλοντας επίσης τη συσκευή σε πειραματική μελέτη, γίνεται δυνατή η αποκάλυψη του πραγματικού ρυθμού μεταφοράς δεδομένων από και προς τη συσκευή, καθώς και τα σημεία που χρείζουν βελτιστοποίησης.

5.1 Πειραματική Διαδικασία

Η πειραματική πλατφόρμα πάνω στην οποία πραγματοποιήθηκαν οι μετρήσεις είναι ένα σύστημα αρχιτεκτονικής x86, που ως λειτουργικό σύστημα χρησιμοποιεί το Debian GNU/Linux και είναι επίσης εφοδιασμένο με τον προσαρμογέα δικτύου Myrinet-2000. Τα αναλυτικά χαρακτηριστικά του συστήματος φαίνονται στο Σχ. 5.1. Έχοντας υλοποιήσει τη συσκευή MyriBLK και μεταγλωττίσει τον κώδικά μας, φτάνουμε στο σημείο που η συσκευή πρέπει να γίνει διαθέσιμη στο σύστημα, ώστε να μπορέσει να αξιολογηθεί πειραματικά. Το σύστημα θα αναφέρεται στη συσκευή MyriBLK μέσω του ειδικού αρχείου `/dev/myriblk`. Συγκεκριμένα, ακολουθούνται τα παρακάτω βήματα:

- Εισάγεται το τροποποιημένο GM module στον πυρήνα Linux
- Εγκαθιστάται ένα σύστημα αρχείων (ext2) στη συσκευή `/dev/myriblka`
- Το σύστημα αρχείων της `/dev/myriblka` γίνεται μέρος της κεντρικής δενδρικής δομής του συστήματος
- Αξιολογούμε το σύστημα αρχείων χρησιμοποιώντας το εργαλείο IOzone

Χαρακτηριστικά Πειραματικής Πλατφόρμας	
Επεξεργαστής	Pentium III @ 500MHz
Motherboard	ASUS P3B-F
RAM	512MB PC100 SDRAM
I/O Bus	PCI 32-bit @ 33MHz
NIC	Myrinet-2000 M3F-PCI64B-2

Λογισμικό	
Λειτουργικό Σύστημα	Debian GNU/Linux 4.1.1-21
Πυρήνας	2.6.24-myri (προερχόμενος από: 2.6.24-6~etchnhalf.7)
Μεταγλωττιστής	gcc version 4.1.2 20061115
Myrinet/GM	gm-2.1.29_Linux
Lanai Tools	lanai-tools-3.3.2..2

Σχήμα 5.1: *Τεχνικά χαρακτηριστικά πειραματικής πλατφόρμας*

Το IOzone είναι ένα εργαλείο αξιολόγησης συστημάτων αρχείων. Η αξιολόγηση γίνεται δυνατή δημιουργώντας και μετρώντας ένα πλήθος λειτουργιών αρχείων (file operations). Αποτελεί ένα πολύ χρήσιμο εργαλείο για την εκτέλεση μίας ευρείας ανάλυσης ενός συστήματος αρχείων που υποστηρίζουν διάφορες πλατφόρμες και χρησιμοποιείται κάτω

από ένα μεγάλο σύνολο διαφορετικών λειτουργικών συστημάτων. Στην πειραματική αξιολόγηση της συσκευής MyriBLK δεν θα εκτελέσουμε όλες τις διαφορετικές λειτουργίες αρχείων που υποστηρίζει το IOzone για έλεγχο της επίδοσης της Ε/Ε ενός αρχείου, αλλά θα επικεντρωθούμε στις: read, write, re-read, re-write.

`read:` Γίνεται μέτρηση της επίδοσης της λειτουργίας διαβάσματος ενός υπάρχοντος αρχείου.

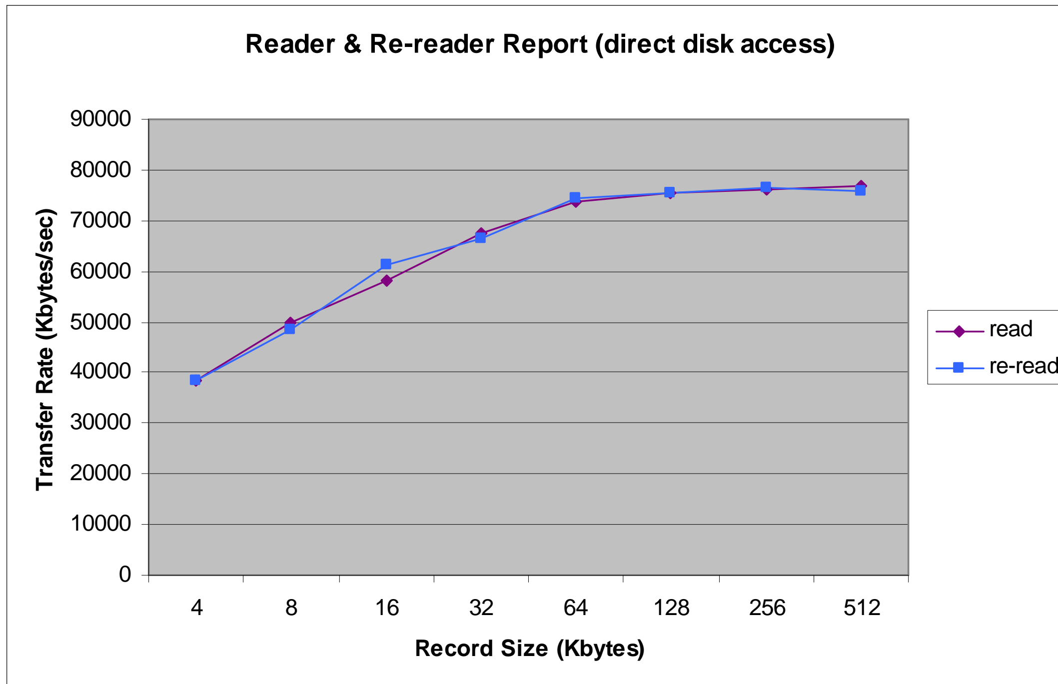
`re-read:` Γίνεται μέτρηση της επίδοσης της λειτουργίας διαβάσματος ενός αρχείου που έχει διαβαστεί πρόσφατα.

`write:` Γίνεται μέτρηση της επίδοσης της λειτουργίας εγγραφής ενός νέου αρχείου. Κατά τη διάρκεια εγγραφής ενός νέου αρχείου πρέπει να εγγραφούν τόσο τα δεδομένα στο μέσο αποθήκευσης (η πραγματική πληροφορία του αρχείου), όσο και όλη η πληροφορία που αφορά στην τοποθεσία τους πάνω σε αυτό (metadata).

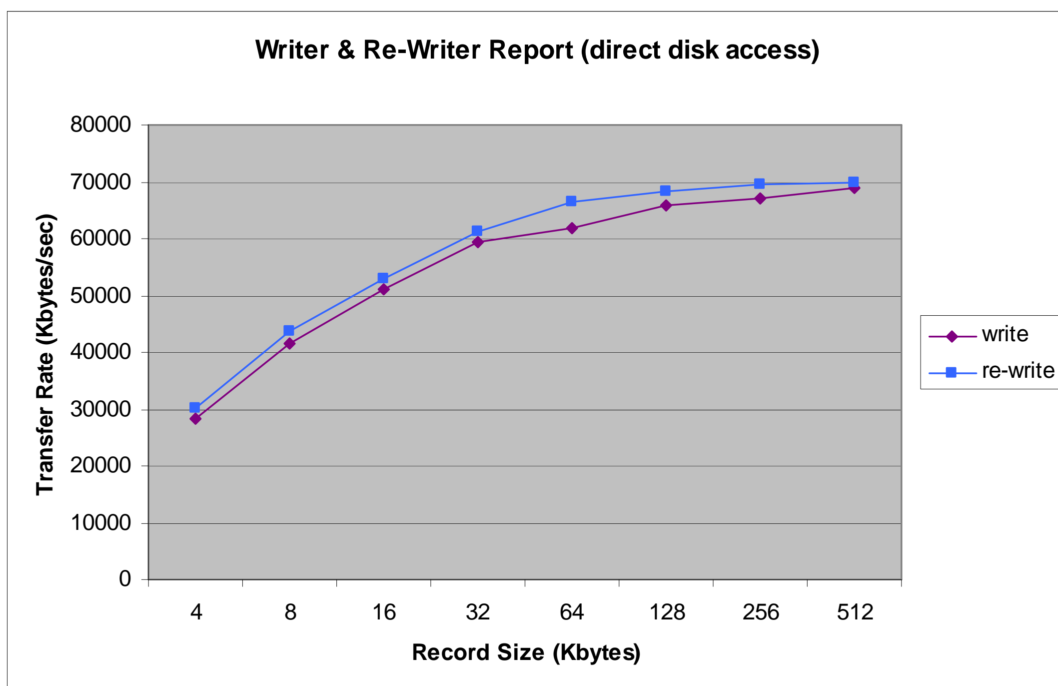
`re-write:` Γίνεται μέτρηση της επίδοσης της λειτουργίας εγγραφής ενός υπάρχοντος αρχείου. Όταν ένα αρχείο προϋπάρχει, τότε η διαδικασία εγγραφής απαιτεί λιγότερη εργασία, αφού η πρόσθετη πληροφορία (metadata) υπάρχει ήδη.

Κατά την πειραματική διαδικασία εκτελούμε το IOzone για αρχείο μεγέθους 750KB (σχεδόν το σύνολο της διαθέσιμης μνήμης της συσκευής `/dev/myriblk`). Όλες οι τιμές που παρουσιάζονται προκύπτουν ως ο μέσος όρος 10 διαφορετικών επαναλήψεων τις διαδικασίας. Τα αποτελέσματα φαίνονται στα παρακάτω διαγράμματα.

Στα Διαγ. 5.2 και Διαγ. 5.3 παρουσιάζονται οι λειτουργίες read, re-read και write, re-write, οι οποίες παρακάμπτουν την κρυφή μνήμη του συστήματος (buffer cache) και απευθύνονται κατευθείαν στο αποθηκευτικό μέσο. Στον άξονα x φαίνονται τα διαφορετικά μεγέθη record και στον άξονα y οι αντίστοιχοι ρυθμοί μετάδοσης δεδομένων εκφρασμένοι σε Kbytes/sec. Στη συνέχεια (Διαγ. 5.4) επαναλαμβάνονται οι ίδιες μετρήσεις για τις λειτουργίες read και write, αλλά αυτή τη φορά σε τυχαία σημεία μέσα στο αρχείο (random read/write), για να διαπιστωθούν οι καθυστερήσεις σε σχέση με την τοποθεσία των δεδομένων στη μονάδα δίσκου (seek latencies).



Σχήμα 5.2: *Λειτουργίες read και re-read με άμεση πρόσβαση στο μέσο*



Σχήμα 5.3: *Λειτουργίες write και re-write με άμεση πρόσβαση στο μέσο*

Σχήμα 5.4: *Λειτουργίες random read/write με άμεση πρόσβαση στο μέσο*

5.2 Συμπεράσματα - Μελλοντική Εργασία

Στην πρώτη περίπτωση παρατηρούμε ταχύτητες που ξεκινούν περίπου από τα 40MBytes/sec (38367) και φτάνουν τα 80MBytes/sec (77016) για μέγεθος record 512Kbytes, όσον αφορά στη λειτουργία read. Από τη στιγμή που παρακάμπτουμε την memory cache, η λειτουργία re-read ανοίγει από την αρχή το αρχείο και έτσι συμπίπτει ουσιαστικά με τη read, γι' αυτό και στο Διαγ. 5.2 οι καμπύλες μετρήσεων σχεδόν αλληλοκαλύπτονται. Για να έχουμε ένα μέτρο σύγκρισης της απόδοσης read, αλλά και όλων των υπόλοιπων μετρήσεων που πραγματοποιήθηκαν κατά την πειραματική αποτίμηση, σε αυτό το σημείο υπολογίζουμε κάποια θεωρητικά μέγιστα. Τα θεωρητικά μέγιστα αναφέρονται στο μέγιστο ρυθμό μετάδοσης δεδομένων που θα μπορούσε να επιτύχει η υλοποίηση αν συνυπολογίσουμε τα όρια που θέτει το υλικό, αλλά και η αρχιτεκτονική που χρησιμοποιούμε. Στην περίπτωσή μας, ο ρυθμός μετάδοσης δεδομένων περιορίζεται από τα όρια του φυσικού διαύλου επικοινωνίας που συνδέει τον προσαρμογέα δικτύου Myrinet με το υπόλοιπο σύστημα. Ο δίαυλος αυτός είναι ο διάδρομος PCI. Μετρώντας την ικανότητα μετάδοσης δεδομένων του διαύλου PCI στο συγκεκριμένο σύστημα, καταλήγουμε σε: 121MBytes/sec για λειτουργία read και 120MBytes/sec για λειτουργία write.

Έχοντας υπ’ όψιν αυτά τα μεγέθη, συμπεραίνουμε ότι για άμεση λειτουργία διαβάσματος από τη μονάδα δίσκου (direct read) πετυχαίνουμε ταχύτητες έως και 64% της ταχύτητας του διαύλου. Αντίστοιχα παρατηρώντας το Διαγ. 5.3 έχουμε τιμές που ξεκινούν από 30MBytes/sec (30083) και φτάνουν τα 70MBytes/sec (69775). Αν ανάγουμε αυτές τις τιμές ρυθμού μεταφοράς δεδομένων για άμεση λειτουργία write στη μονάδα δίσκου, καταλήγουμε σε ταχύτητες έως και 58% της ταχύτητας του διαύλου. Όπως και κατά τη λειτουργία re-read το αρχείο ανοίγεται από την αρχή, το ίδιο συμβαίνει και κατά τη λειτουργία re-write γι’ αυτό και δεν παρουσιάζονται ουσιαστικές διαφορές απόδοσης από την λειτουργία write. Η μικρή αύξηση της απόδοσης της re-write σε σχέση με την write οφείλεται στην απουσία της πρόσθετης πληροφορίας (metadata) που αναφέραμε στην προηγούμενη ενότητα, αφού το αρχείο προϋπάρχει.

Στη δεύτερη περίπτωση χρησιμοποιούμε το IOzone για να προσπελάσουμε το αρχείο σε διαφορετικά σημεία, τυχαία επιλεγμένα. Οι επαναλαμβανόμενες λειτουργίες εγγραφής και ανάγνωσης σε τυχαία σημεία αποκαλύπτουν πιθανές αδυναμίες του υλικού και συνήθως επιφέρουν πτώση της τελικής επίδοσης σε αποθηκευτικά μέσα με κινητά φυσικά μέρη, όπως είναι οι σκληροί δίσκοι. Στο Διαγ. 5.4 γίνεται φανερό πως κάτι τέτοιο δεν συμβαίνει στην περίπτωση της συσκευής MyriBLK, αφού οι μετρήσεις σχεδόν συμπίπτουν με αυτές των προηγούμενων διαγραμμάτων. Τα αποτελέσματα αυτά ήταν αναμενόμενα σε ένα βαθμό, επειδή η μνήμη SRAM του προσαρμογέα Myrinet προσομοιώνει τη συμπεριφορά ενός SSD δίσκου, στον οποίο η θέση των δεδομένων δεν επηρεάζει τον ρυθμό με τον οποίο αυτά ανακτώνται ή εγγράφονται.

Οι παραπάνω επιδόσεις μπορούν να βελτιωθούν με επεκτάσεις της υλοποίησης της συσκευής MyriBLK. Μελλοντική εργασία πάνω στον σχεδιασμό και την υλοποιήση της παρούσας διπλωματικής εργασίας, μπορεί να επιφέρει αύξηση των προαναφερθέντων ρυθμών μετάδοσης δεδομένων, μεγαλύτερη αξιοπιστία σε σημεία που πιθανόν την επιδέχονται, επέκταση σε διαφορετικές πλατφόρμες, αλλά και παρουσίαση διαφορετικών προτάσεων χρήσης μίας παρόμοιας αρχιτεκτονικής προσέγγισης, όπου το αποθηκευτικό μέσο βρίσκεται πολύ κοντά στο φυσικό δίκτυο διασύνδεσης. Συγκεκριμένα, ένα σημείο που θα θεωρούσαμε κρίσιμης σημασίας, είναι η τροποποίηση της μεθόδου request, έτσι ώστε ο οδηγός αλλά και το υλικολογισμικό να μπορούν να επεξεργαστούν και να εξυπηρετήσουν πολλαπλές αιτήσεις ασύγχρονα, χρησιμοποιώντας αποδοτικούς εσωτερικούς αλγόριθμους δρομολόγησης. Μια τέτοια υλοποίηση θα είχε ώς αποτέλεσμα καλύτερη διαχείριση των αιτήσεων του υποσυστήματος block του Linux, με άμεση συ-

νέπεια την βελτίωση της τελικής επίδοσης. Επίσης, ενδιαφέρον θα παρουσίαζε η επέκταση της υλοποίησης σε μια πλατφόρμα με καλύτερα τεχνικά χαρακτηριστικά, όπως η επόμενη γενιά Myrinet, η Myri-10G, που υποστηρίζει σύνδεση PCI Express μεγαλύτερου ρυθμού μετάδοσης δεδομένων και εφοδιάζεται με ενσωματωμένο επεξεργαστή υψηλότερης συχνότητας ρολογιού.