ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδίαση και Υλοποίηση Δικτυακής Συσκευής Block στο Λειτουργικό Σύστημα Linux

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αντώνιος - Κορνήλιος Β. Κούρτης

Επιβλέπων: Νεκτάριος Κοζύρης
Επίκουρος Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2004

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Σχεδίαση και Υλοποίηση Δικτυακής Συσκευής Block στο Λειτουργικό Σύστημα Linux

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αντώνιου - Κορνήλιου Β. Κούρτη

Επιβλέπων: Νεκτάριος Κοζύρης
Επίκουρος Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 19 Οκτωβρίου 2004

Ν. Κοζύρης Επ. Καθηγητής Ε.Μ.Π	Τ. Σελλής Καθηγητής Ε.Μ.Π	Π. Τσανάκας Καθηγητής Ε.Μ.Π

Αθήνα, 19 Οκτωβρίου 2004

Αντώνιος - Κορνήλιος Β. Κούρτης

Περίληψη

Ο σκοπός της εργασίας αυτής είναι η σχεδίαση και η υλοποίηση μιας δικτυακής συσκευής block στο λειτουργικό σύστημα Linux, η οποία θα αποτελείται από ανεξάρτητα μεταξύ τους στρώματα. Η συσκευή αυτή επιτρέπει σε ένα σύστημα (πελάτης) να έχει πρόσβαση στον αποθηκευτικό χώρο ενός άλλου συστήματος (εξυπηρετητής) με διαφανή τρόπο.

Ο σχεδιασμός σε πολλαπλά στρώματα επιτρέπει την υποστήριξη διαφορετικών τύπων δικτύου και την εφαρμογή πολιτικής χρονοδρομολόγησης Ε/Ε ανάλογα με το δικτυακό περιβάλλον που χρησιμοποιείται. Επιπρόσθετα, επιτρέπει την υλοποίηση πιο ανεπτυγμένων συστημάτων αποθήκευσης, τα οποία θα βασίζονται στη συσκευή αυτή, προσθέτοντας λειτουργικότητα στα επιμέρους στρώματα.

Ο οδηγός της συσκευής είναι υπεύθυνος για ένα σύνολο από συσκευές block και παρέχει μια διεπαφή στα διάφορα δικτυακά στρώματα ώστε να καταχωρηθούν σε αυτόν και να συσχετιστούν με μια συγκεκριμένη συσκευή. Τα διάφορα δικτυακά στρώματα είναι αυτά, τα οποία προσφέρουν την λειτουργικότητα για την ικανοποίηση των αιτήσεων block που εκδίδονται από το λειτουργικό σύστημα.

Το δικτυακό στρώμα που αναπτύχθηκε στην εργασία αυτή λειτουργεί προωθώντας τις αιτήσεις σε μια εφαρμογή, η οποία βρίσκεται στον χώρο χρήστη. Για λόγους αποδοτικότητας τα δεδομένα block δεν αντιγράφονται από τον χώρο πυρήνα στον χώρο χρήστη και αντίστροφα.

Η εφαρμογή χώρου χρήστη, η οποία είναι υπεύθυνη για την ικανοποίηση των block αιτήσεων χρησιμοποιεί μια απλή και γενική δικτυακή βιβλιοθήκη για την επικοινωνία με το απομακρυσμένο σύστημα. Η βιβλιοθήκη αυτή παρέχει μια κοινή διεπαφή, η οποία μπορεί να υλοποιηθεί από διαφορετικά δίκτυα. Υλοποιήσεις της διεπαφής αυτής αναπτύχθηκαν για δίκτυα TCP/IP χρησιμοποιώντας την διεπαφή sockets, για δίκτυα Myrinet χρησιμοποιώντας την διεπαφή GM και για δίκτυα SCI χρησιμοποιώντας την διεπαφή SISCI.

Λέξεις Κλειδιά: Συστήματα Αποθήκευσης, Συσκευές Αποθήκευσης, Network Attached Storage, Storage Area Network, Block Συσκευή, Δικτυακή Block Συσκευή, Linux, Linux Block Layer, Myrinet, GM, SCI, SISCI.

Abstract

The purpose of this thesis is the design and implementation of a modular network block device for the Linux operating system. This device allows a system (client) to use storage area exported by another system (server) in a transparent way.

The modular approach allows different network types to be supported by the device and makes it easier to apply different I/O scheduling policies, based on the underlying network environment. Also, it allows the implementation of more complex storage systems based on this device by adding functionality to individual modules.

The device driver is responsible for a number of block devices and provides an interface for various network modules to be registered and associated with each device. The network modules provide the actual functionality of handling the various block requests issued by the operating system.

The network module developed in this thesis handles the requests by forwarding them to a userspace application. For reasons of efficiency the block data are not copied from kernelspace to userspace and vice versa.

The userspace application that is responsible for handling the various block request uses a simple and general networking library for communication with the remote system. This library provides a common interface that can be implemented for different networks. Implementations of this interface were developed for TCP/IP networks, using the sockets interface, Myrinet networks using the GM interface and SCI networks using the SISCI interface.

Keywords: Storage Systems, Storage Devices, Network Attached Storage, Storage Area Network, Block Device, Network Block Device, Linux, Linux Block Layer, Myrinet, GM, SCI, SISCI.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Πρόλογος

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του επίκουρου καθηγητή Νεκτάριου Κοζύρη.

Θα ήθελα να ευχαριστήσω τον κ. Κοζύρη τόσο για την καθοδήγηση και την υποστήριξή του, όσο και για την δυνατότητα που μου έδωσε να να συμμετάσχω στο εργαστήριο κατά την διάρκεια των προπτυχιακών μου σπουδών κερδίζοντας πολύτιμες εμπειρίες.

Θα ήθελα να εκφράσω τις ευχαριστίες μου σε όλα τα μέλη του εργαστηρίου και ιδιαίτερα στον διδάκτορα του Εθνικού Μετσόβιου Πολυτεχνείου Αριστείδη Σωτηρόπουλο και στον προπτυχιακό φοιτητή Γιώργο Τσουκαλά, η συμβολή των οποίων στην επιστημονική και πνευματική μου ωρίμανση ήταν και παραμένει καθοριστική.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου για την στήριξή τους, ηθική και υλική, χωρίς την βοήθεια της οποίας θα ήταν αδύνατη η ολοκλήρωση των σπουδών μου.

Αθήνα, Οκτώβριος 2004

Αντώνιος - Κορνήλιος Β. Κούρτης

Κεφάλαιο 1

Συστήματα Αποθήκευσης

Το παρόν κεφάλαιο ως σκοπό έχει τη σύντομη παρουσίαση των βασικότερων και πιο διαδεδομένων τεχνολογιών για την αποθήκευση δεδομένων στα σύγχρονα υπολογιστικά συστήματα. θα περιοριστούμε στη μελέτη τεχνολογιών που προορίζονται για υπολογιστές προσωπικής χρήσης και εξυπηρετητές, μιας και αυτά τα συστήματα είναι που σχετίζονται με το αντικείμενο της εργασίας αυτής.

1.1 Εισαγωγή

Είναι κοινά αποδεκτό ότι η ανάπτυξη του υποσυστήματος εισόδου - εξόδου (Ε/Ε) υπολείπεται σε σχέση με την ανάπτυξη των επεξεργαστικών μονάδων και της μνήμης. Η ασυμμετρία αυτή έχει ως αποτέλεσμα, σύμφωνα με τον νόμο του Amdahl, να μην αξιοποιούνται οι δυνατότητες των επεξεργαστικών μονάδων και της μνήμης πλήρως, όσον αφορά τη συνολική επίδοση του υπολογιστικού συστήματος.

Άλλωστε, ενώ κάποτε οι υπολογιστές είχαν ως βασικό ρόλο την εκτέλεση υπολογισμών, στις μέρες μας ο βασικός τους ρόλος συνδέεται άμεσα με την έννοια της πληροφορίας. Οι σύγχρονοι υπολογιστές καλούνται να επεξεργαστούν, να αποθηκεύσουν και να μεταφέρουν όλο και μεγαλύτερες ποσότητες πληροφορίας.

Οι παραπάνω λόγοι οδήγησαν τα τελευταία χρόνια σε ανάπτυξη και καθιέρωση νέων τεχνολογιών στα συστήματα Ε/Ε γενικότερα, αλλά και στα συστήματα αποθήκευσης ειδικότερα.

1.2 Αρχιτεκτονική Ε/Ε για συσκευές αποθήκευσης

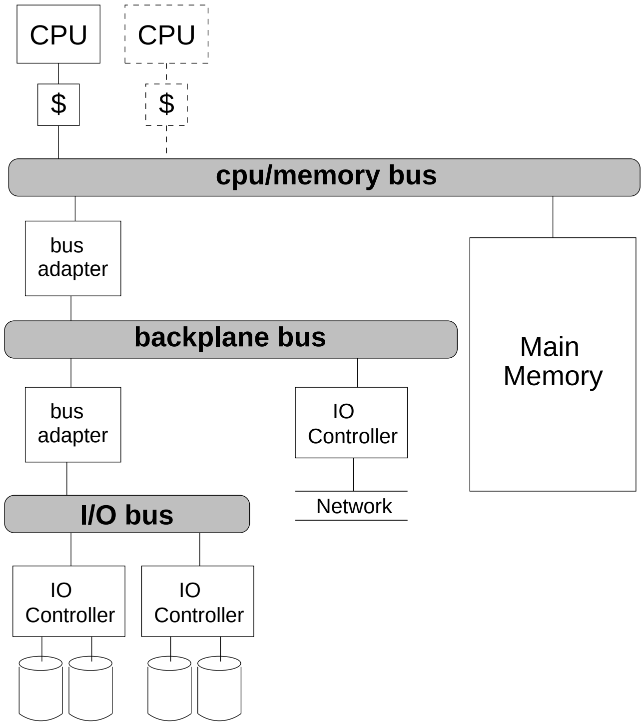
1.2.1 Αρχιτεκτονική Ε/Ε στους σύγχρονους υπολογιστές

Ένας από τους καθοριστικούς παράγοντες για την επίδοση των συστημάτων αποθήκευσης είναι το πως συνδέονται αυτά με τις επεξεργαστικές μονάδες και τη μνήμη. Η πιο συνηθισμένη περίπτωση είναι η χρήση της αρχιτεκτονικής διαδρόμου για την διασύνδεση των μονάδων Ε/Ε και του υπόλοιπου υπολογιστικού συστήματος.

Η διασύνδεση διαδρόμου, έχει ως πλεονεκτήματα το χαμηλό κόστος και την εύκολη υλοποίηση της από τις επιμέρους συσκευές που μετέχουν σε αυτή. Ωστόσο, η επίδοσή της σε περιπτώσεις που υπάρχουν πολλές διαφορετικές ροές δεδομένων ή υψηλές ταχύτητες μεταφοράς δεν είναι καλή.

Συχνά χρησιμοποιείται μια ιεραρχία διαφορετικών διαδρόμων. Δύο διάδρομοι συνδέονται μεταξύ τους με συσκευές, οι οποίες γνωρίζουν τα πρωτόκολλα τους και φροντίζουν για την διαφανή επικοινωνία όλων των συσκευών. Οι συσκευές αυτές ονομάζονται προσαρμογείς διαδρόμου (bus adapters).

Στο σχήμα 1.1 παρουσιάζεται το διάγραμμα ενός κλασικού υπολογιστικού συστήματος. Διακρίνουμε τρεις διαφορετικούς διαδρόμους:



Σχήμα 1.1: Αρχιτεκτονική Διασύνδεσης Υπολογιστικού συστήματος με τις μονάδες Ε/Ε

- **Διάδρομος μνήμης - επεξεργαστών (cpu/memory bus)**

  Ο διάδρομος αυτός επιτρέπει στις επεξεργαστικές μονάδες, την κεντρική μνήμη και τις περιφερειακές μονάδες να επικοινωνούν μεταξύ τους. Στις περισσότερες περιπτώσεις είναι εξειδικευμένος, ώστε να εκμεταλλευτεί όσο το δυνατόν καλύτερα τα χαρακτηριστικά των επεξεργαστών και της κεντρικής μνήμης. Ο διάδρομος αυτός είναι συνήθως σχεδιασμένος για την υποστήριξη μικρού αριθμού συσκευών.

- **Κεντρικός διάδρομος περιφερειακών συσκευών (backplane bus)**

  Ο διάδρομος αυτός χρησιμοποιείται σαν ενδιάμεσος διάδρομος για να συνδέει επιμέρους διαδρόμους μονάδων Ε/Ε και αυτόνομες μονάδες Ε/Ε με τον διάδρομο μνήμης - επεξεργαστών. Σε αντίθεση με τον διάδρομο μνήμης - επεξεργαστών οι συσκευές από τις οποίες θα αποτελείται δεν είναι γνωστές κατά τη σχεδίασή του. Για την εξασφάλιση της ομαλής του λειτουργίας, συνεπώς, είναι απαραίτητη η θέσπιση κάποιου γενικού πρωτοκόλλου.

  Η ύπαρξη αυτού του υβριδικού διαδρόμου επιτρέπει την επέκταση του υπολογιστικού συστήματος με διαφορετικά συστήματα διασύνδεσης συσκευών Ε/Ε, βοηθώντας στην προσαρμοστικότητα του υπολογιστικού συστήματος στις εκάστοτε απαιτήσεις. Με αυτόν τον τρόπο μπορούν να υιοθετηθούν μελλοντικά πρωτόκολλα διασύνδεσης, αφού αρκεί η δημιουργία ενός προσαρμογέα για την ένωση των δύο διαδρόμων.

  Η κυρίαρχη αρχιτεκτονική για τους διαδρόμους αυτούς στις μέρες μας είναι το PCI.

- **Διάδρομος Ε/Ε (I/O bus)**

  Ο διάδρομος Ε/Ε είναι ο τελευταίος διάδρομος στην ιεραρχία. Συνδέει ίδιου τύπου συσκευές που υλοποιούν ένα πιο εξειδικευμένο πρωτόκολλο από αυτό του κεντρικού διαδρόμου.

Παραδείγματα, τέτοιων πρωτοκόλλων είναι το ATA (Advanced Technology Attachment) και το SCSI (Small Computer System Interface). Τέτοιοι διάδρομοι μπορεί να συνδέονται ή στον διάδρομο περιφερειακών συσκευών (όπως φαίνεται στο σχήμα 1.1) ή κατευθείαν στον διάδρομο μνήμης - επεξεργαστών.

1.2.2 Επικοινωνία με τις μονάδες Ε/Ε

Ένα βασικό ζήτημα είναι το πως επιτυγχάνεται η μεταφορά δεδομένων από την μνήμη στις συσκευές Ε/Ε και το αντίστροφο. Ακολουθεί μια παρουσίαση των πιο συνηθισμένων μεθόδων.

1.2.2.1 Προγραμματιζόμενη Ε/Ε

Κατά την προγραμματιζόμενη Ε/Ε (Programmed I/O - PIO), ο επεξεργαστής είναι αυτός που αναλαμβάνει τη μεταφορά των δεδομένων. Είτε χρησιμοποιείται ένας ξεχωριστός χώρος διευθύνσεων Ε/Ε (I/O address space), είτε κομμάτια του χώρου διευθύνσεων της μνήμης παραχωρούνται στις συσκευές (memory mapped I/O). Στην πρώτη περίπτωση ο επεξεργαστής χρησιμοποιεί ειδικές εντολές Ε/Ε ενώ στην δεύτερη κοινές εντολές `LOAD` και `STORE`.

Η μέθοδος αυτή, όταν εφαρμόζεται σε συσκευές που προσανατολίζονται σε μεγάλες μεταφορές δεδομένων, όπως οι συσκευές αποθήκευσης, δεν είναι αποδοτική αφού καταναλώνει πολύ επεξεργαστική ισχύ. Για την αντιμετώπιση του προβλήματος αυτού αναπτύχθηκε μια τεχνική, η οποία αναφέρεται ως Απευθείας Πρόσβαση στη Μνήμη (Direct Memory Access - DMA).

1.2.2.2 Απευθείας πρόσβαση στη μνήμη

Η μέθοδος αυτή χρησιμοποιεί μονάδες υλικού, οι οποίες ονομάζονται μηχανές DMA (DMA engines). Ο επεξεργαστής αρκεί να προγραμματίσει τις μονάδες αυτές σχετικά με την μεταφορά δεδομένων που χρειάζεται να γίνει. Κατόπιν ο επεξεργαστής είναι ελεύθερος να συνεχίσει την λειτουργία του, ενώ πραγματοποιείται η μεταφορά από την μνήμη στη συσκευή ή το αντίστροφο. Μόλις η μεταφορά ολοκληρωθεί η μηχανή DMA, ειδοποιεί τον επεξεργαστή μέσω μιας διακοπής (interrupt). Σύγχρονες μηχανές DMA υλοποιούν και πιο προηγμένες λειτουργίες όπως λίστες Scatter/Gather, παρέχοντας υποστήριξη για μεταφορά δεδομένων από πολλαπλές διευθύνσεις.

1.2.3 Τα πρότυπα ATA και SCSI

1.2.3.1 ATA

Το πρωτόκολλο ATA που συχνά αναφέρεται και με άλλα ονόματα (IDE, ATAPI, UDMA), τα οποία προκύπτουν από διάφορες αναθεωρήσεις του προτύπου, είναι το πιο διαδεδομένο. Χρησιμοποιείται αποκλειστικά για την σύνδεση συσκευών αποθήκευσης στο υπολογιστικό σύστημα. Η μεγάλη χρήση του οφείλεται κυρίως στο χαμηλό κόστος του, που το κάνει κυρίαρχο στην αγορά των υπολογιστών προσωπικής χρήσης.

1.2.3.2 SCSI

Το SCSI, σε αντίθεση με το ATA είναι γενικό πρότυπο και μπορεί να χρησιμοποιηθεί για κάθε είδους περιφερειακή συσκευή. Ωστόσο, τις περισσότερες φορές και αυτό χρησιμοποιείται για συσκευές αποθήκευσης. Το SCSI θεωρείται πιο αξιόπιστο, πιο επεκτάσιμο και καλύτερα διαχειριζόμενο. Βρίσκει εφαρμογή κυρίως σε σταθμούς εργασίας υψηλής απόδοσης και εξυπηρετητές.

Στον πίνακα 1.1 παρουσιάζονται οι βασικότερες διαφορές των δύο προτύπων.

πρότυπο	ATA	SCSI
Μέγιστος αριθμός συσκευών	2	16
Μέγιστο μήκους καλωδίου (m)	0.4	12
Μέγιστος αριθμός εκκρεμών εντολών	32	256
Μέγιστος ρυθμός μεταφοράς δεδομένων	133 Mb/sec	640 Mb/sec
Έλεγχος εγκυρότητας δεδομένων	μερικός	πλήρης
Μήκος δεδομένων (bit)	16	8, 16, 32

Πίνακας 1.1: Συγκριτικός πίνακας για τα δύο βασικότερα πρότυπα για διαδρόμους Ε/Ε ΑΤΑ και SCSI.

1.2.4 Εξέλιξη των προτύπων ΑΤΑ και SCSI

Τα πρότυπα ΑΤΑ και SCSI, παρά τις διαφορές τους, έχουν βασικές ομοιότητες. Τα δύο πρότυπα βασίζονται σε παράλληλη μετάδοση δεδομένων και υλοποιούν τοπολογία διαμοιραζόμενου διαδρόμου για την σύνδεση των επιμέρους συσκευών. Τα δύο αυτά βασικά τους χαρακτηριστικά δεν τους δίνουν τη δυνατότητα να ανταποκριθούν στις σύγχρονες, αλλά και στις μελλοντικές ανάγκες.

Κατά την παράλληλη μετάδοση δεδομένων μεταφέρονται ταυτόχρονα περισσότερα από ένα bit, από το ένα άκρο στο άλλο, ενώ το μήκος του διαδρόμου είναι συνήθως πολλαπλάσιο των 8 bits. Αντίθετα στην σειριακή μετάδοση, τα δεδομένα μεταφέρονται ανά ένα bit τη φορά.

Διαισθητικά, θα περιμέναμε να επιτυγχάνονται υψηλότεροι ρυθμοί μεταφοράς δεδομένων με την παράλληλη μετάδοση. Ωστόσο, κάτι τέτοιο δεν ισχύει. Χρησιμοποιώντας σειριακή μετάδοση είναι δυνατή η επίτευξη μεγαλύτερων ταχυτήτων. Μερικοί από τους λόγους για τους οποίους συμβαίνει αυτό είναι ότι δεν υπάρχουν παρεμβολές από διαφορετικά κανάλια, δεν χρειάζεται συγχρονισμός μεταξύ τους και το μικρότερο μέγεθος των κυκλωμάτων επιτρέπει λειτουργία σε υψηλότερες συχνότητες.

Αντίστοιχα η διασύνδεση των συσκευών με τοπολογία διαδρόμου δημιουργεί αρκετά προβλήματα, αφού περιορίζονται σημαντικά οι δυνατότητές για υψηλούς ρυθμούς μετάδοσης δεδομένων και χαμηλό χρόνο απόκρισης.

Δεν είναι τυχαίο λοιπόν που τα δύο νέα πρότυπα που σκοπό έχουν να αντικαταστήσουν το ΑΤΑ και το SCSI, χρησιμοποιούν σειριακή μετάδοση δεδομένων και τοπολογία σημείο προς σημείο (point to point). Τα πρότυπα αυτά ονομάζονται Serial ATA (SATA) και Serial Attached SCSI (SAS) αντίστοιχα.

1.2.4.1 SATA

Το SATA ως αντικαταστάτης του ΑΤΑ διαθέτει νέα χαρακτηριστικά που σκοπό έχουν την καλύτερη απόδοση, επεκτασιμότητα και αξιοπιστία του συστήματος διατηρώντας το χαμηλό κόστος, που έκανε το ΑΤΑ τόσο δημοφιλές.

Μερικά από τα νέα χαρακτηριστικά είναι:

- Έλεγχος εγκυρότητας δεδομένων και ορθής λειτουργίας και δυνατότητες διόρθωσης λαθών.
- Δυνατότητα πρόσθεσης - αφαίρεσης συσκευής ενώ το σύστημα παραμένει σε λειτουργία (hot-plug).
- Καλύτερη καλωδίωση. Τα καλώδια είναι πλέον πιο λεπτά, προσφέροντας καλύτερες δυνατότητες ψύξης του συστήματος και μεγαλύτερου μήκους, διευκολύνοντας την εγκα-

τάσταση των δίσκων.

1.2.4.2 SAS

Το πρότυπο SAS είναι η εξέλιξη του παράλληλου SCSI για την υποστήριξη σειριακής μετάδοσης δεδομένων. Το SAS προβλέπει συσκευές που μπορούν να υποστηρίξουν ταυτόχρονη μετάδοση δεδομένων και στις δύο κατευθύνσεις (full duplex).

Το πρότυπο, επίσης, προβλέπει ειδικές συσκευές επέκτασης (expanders) για την υποστήριξη μεγάλου αριθμού SCSI συσκευών. Η κάθε συσκευή επέκτασης, επιτρέπει τη σύνδεση 64 συσκευών, είτε SAS, είτε επέκτασης. Συνολικά μπορούν να διευθυνσιοδοτηθούν 16,256 συσκευές.

Το πρότυπο SAS επιλέχτηκε να είναι συμβατό με το SATA χρησιμοποιώντας σηματοδότηση, που είναι υποσύνολο αυτής του SATA. Συνεπώς, με τη χρήση του πρωτοκόλλου STP (SATA Tunneled Protocol) υποστηρίζονται συσκευές SATA σε ένα σύστημα SAS.

1.3 Συσκευές αποθήκευσης

Υπάρχει ένα ευρύ φάσμα συσκευών αποθήκευσης για διάφορες εφαρμογές. Οι πιο συνηθισμένες είναι οι σκληροί δίσκοι, οι κάρτες μνήμης, οι οπτικοί δίσκοι (DVD-CD) και οι συσκευές βασισμένες σε μαγνητική ταινία. Θα περιοριστούμε στους σκληρούς δίσκους, δεδομένου ότι κυριαρχούν στον χώρο των προσωπικών υπολογιστών και εξυπηρετητών.

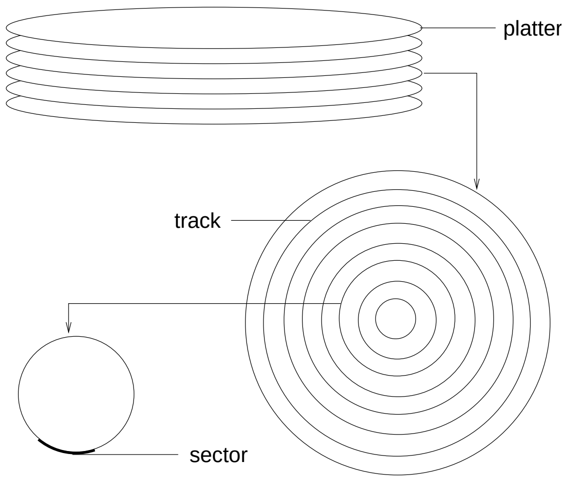
1.3.1 Σκληροί δίσκοι

Οι σκληροί δίσκοι αποτελούνται από ένα σύνολο δίσκων (platters) οι οποίοι περιστρέφονται. Μια συνηθισμένη ταχύτητα περιστροφής τους είναι 7200 στροφές ανά λεπτό. Στους δίσκους αυτούς μπορούν να γραφούν δεδομένα και στις δύο όψεις. Η κάθε επιφάνεια των δίσκων είναι χωρισμένη σε ομόκεντρους κύκλους που ονομάζονται ίχνη (tracks). Συνήθως, υπάρχουν από 3000 έως 5000 ίχνη σε κάθε επιφάνεια. Επιπλέον, κάθε ίχνος χωρίζεται σε τομείς (sectors), οι οποίοι αποτελούν και τη μικρότερη δυνατή μονάδα εγγραφής ή ανάγνωσης από τον δίσκο. Η πιο διαδεδομένη τιμή μεγέθους των τομέων είναι 512 bytes. Τα λειτουργικά συστήματα χρησιμοποιούν τη δική τους ελάχιστη μονάδα δεδομένων για την προσπέλαση του σκληρού δίσκου. Η μονάδα αυτή είναι πολλαπλάσια του τομέα και ονομάζεται block.

Για την ανάγνωση και την εγγραφή δεδομένων από τις επιφάνειες του σκληρού δίσκου, χρησιμοποιούνται μηχανικοί βραχίονες οι οποίοι ονομάζονται κεφαλές ανάγνωσης/εγγραφής (read/write heads). Όταν οι βραχίονες αυτοί δεν χρειάζεται να μετακινηθούν πολύ, η προσπέλαση στον δίσκο μπορεί να γίνει αρκετά γρήγορα. Σε αντίθετη περίπτωση εισάγεται σημαντική καθυστέρηση. Για την καλύτερη απόδοση των σκληρών δίσκων συχνά χρησιμοποιούνται προσωρινοί χώροι αποθήκευσης (buffers), μέσω των οποίων επιτυγχάνονται μεγαλύτερες ταχύτητες προσπέλασης.

1.4 Συστήματα RAID

Το ακρωνύμιο RAID προέρχεται από τα αρχικά των λέξεων Redundant Array of Inexpensive Disks, αν και πολλές φορές αναφέρεται και ως Redundant Array of Independent disks. Η βασική ιδέα του προτύπου RAID είναι η χρήση πολλαπλών σκληρών δίσκων, ώστε να επιτευχθεί αύξηση της ταχύτητας, του μεγέθους και της αξιοπιστίας του συστήματος, αίροντας έτσι τους περιορισμούς και τα προβλήματα των σκληρών δίσκων, όταν χρησιμοποιούνται σαν αυτόνομες αποθηκευτικές μονάδες.



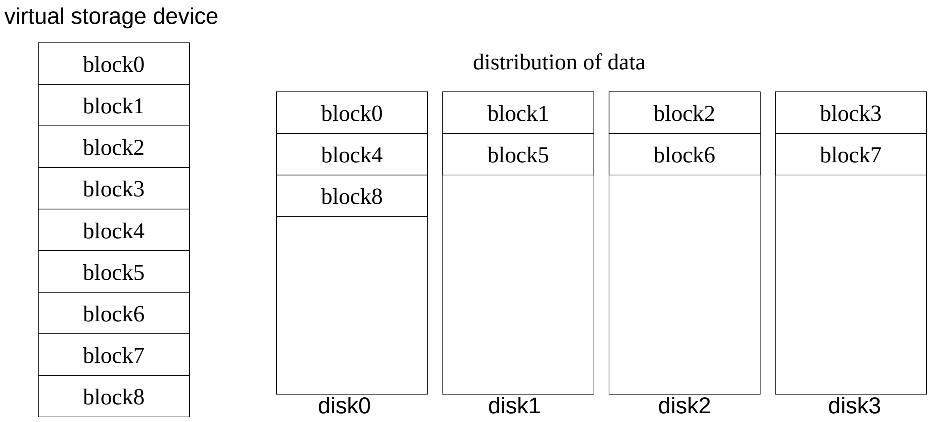
Σχήμα 1.2: Τα μέλη ενός σκληρού δίσκου

Βασικό στοιχείο της φιλοσοφίας του RAID είναι η αποθήκευση πλεονάζουσας (redundant) πληροφορίας. Με αυτόν τον τρόπο μπορούν να αντιμετωπιστούν προβλήματα που πιθανώς να παρουσιαστούν στους επιμέρους δίσκους.

Για την κατηγοριοποίηση των διαφορετικών τρόπων κατανομής των δεδομένων στους πολλαπλούς δίσκους χρησιμοποιείται ένας αριθμός, που υποδηλώνει το πρωτόκολλο της κατανομής αυτής. Ο αριθμός αυτός ονομάζεται και επίπεδο. Παρακάτω αναλύονται τα πρωτόκολλα αυτά.

1.4.1 RAID 0

Στο επίπεδο 0 τα block των δεδομένων προς αποθήκευση κατανέμονται διαδοχικά και συμμετρικά στο σύνολο των δίσκων όπως φαίνεται στο σχήμα 1.3.



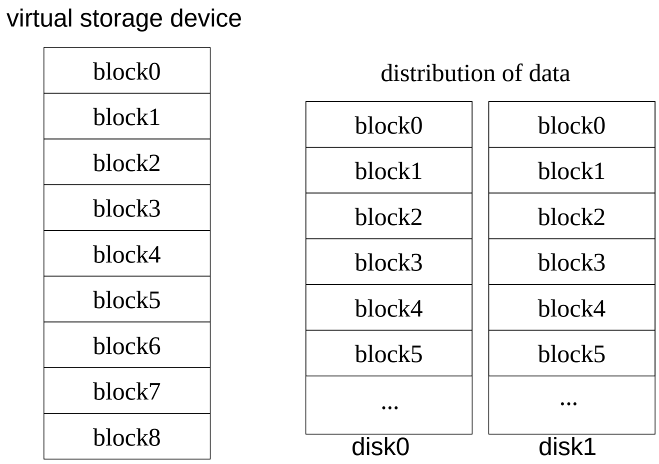
Σχήμα 1.3: Παράδειγμα κατανομής δεδομένων στο RAID 0

Το RAID 0 επιτυγχάνει τη μέγιστη δυνατή χωρητικότητα ενός συστήματος RAID. Επίσης, θεωρητικά, μπορεί να προσφέρει και αυξημένη ταχύτητα στις εγγραφές και στις αναγνώσεις, αφού μπορούν να παραλληλοποιηθούν οι αιτήσεις προς τους επιμέρους δίσκους. Πρακτικά, ωστόσο, η αύξηση αυτή εξαρτάται σε μεγάλο βαθμό από την υλοποίηση του προτύπου και του συστήματος Ε/Ε που χρησιμοποιείται. Η πιθανότητα απώλειας των δεδομένων που είναι αποθηκευμένα σε μια διάταξη RAID 0 ισούται με την πιθανότητα απώλειας των δεδομένων σε έναν δίσκο επί τον αριθμό των δίσκων και είναι αρκετά υψηλή.

Το RAID 0 χρησιμοποιείται σε εφαρμογές όπου δεν απαιτείται μεγάλη ασφάλεια για τα δεδομένα, αλλά μας ενδιαφέρει μόνο η όσο το δυνατόν μεγαλύτερη χωρητικότητα και η απόδοση.

1.4.2 RAID 1

Αντίθετα από το RAID 0 το RAID 1 προσφέρει την μεγαλύτερη δυνατή αξιοπιστία. Τα δεδομένα αντιγράφονται από κοινού σε όλους τους δίσκους της διάταξης, αν και στις περισσότερες περιπτώσεις χρησιμοποιούνται μόνο δύο δίσκοι. Ένα παράδειγμα RAID 1 παρουσιάζεται στο σχήμα 1.4.



Σχήμα 1.4: Παράδειγμα κατανομής δεδομένων στο RAID 1

Με τη χρήση του RAID 1 επιτυγχάνεται μεγάλη ασφάλεια για τα αποθηκευμένα δεδομένα, αφού αυτά υπάρχουν σε όλους τους δίσκους. Το συνολικό μέγεθος της διάταξης ισούται με το μέγεθος του ενός δίσκου. Η ταχύτητα εγγραφής είναι πιο μεγάλη από την περίπτωση του αυτόνομου δίσκου, αφού τα δεδομένα θα πρέπει κάθε φορά να αντιγράφονται σε όλους τους δίσκους. Αντίθετα, η ταχύτητα της ανάγνωσης είναι δυνατόν να αυξηθεί διαβάζοντας παράλληλα από διαφορετικούς δίσκους.

Το RAID 1 είναι ιδανικό για περιπτώσεις που απαιτούν την μέγιστη δυνατή αξιοπιστία. Επίσης, βρίσκει χρήση και σε περιπτώσεις όπου οι εγγραφές δεδομένων είναι σπάνιες σε σχέση με τις αναγνώσεις.

1.4.3 Κώδικες Διόρθωσης Σφαλμάτων

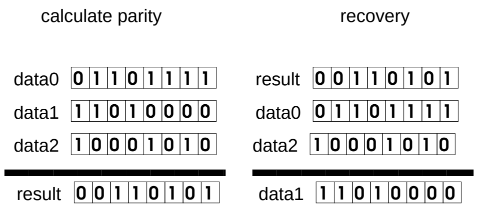
Το RAID 0 είναι έκδηλο ότι είναι ιδιαίτερα επιρρεπές στα σφάλματα, ενώ το RAID 1 εγγυάται συνέχιση της λειτουργίας σε περίπτωση βλάβης κάποιου δίσκου, θυσιάζοντας, όμως, μεγάλο χώρο αποθήκευσης.

Οι παραπάνω δύο διατάξεις είναι απλές εφαρμογές της αρχικής ιδέας, για την χρήση πολλαπλών δίσκων. Τα συστήματα RAID μπορούν να προσφέρουν πολλά περισσότερα χρησιμοποιώντας κώδικες διόρθωσης σφαλμάτων (Error Correcting Codes - ECC). Τέτοιοι κώδικες είναι ευρέως διαδεδομένοι στα ψηφιακά συστήματα επικοινωνίας.

Οι κώδικες διόρθωσης σφαλμάτων χρησιμοποιούν πλεονάζοντα bit που προστίθενται στην αρχική λέξη. ο αριθμός των bits που χρησιμοποιούνται καθορίζει και ποια σφάλματα μπορεί να ανιχνεύσει ο εκάστοτε αλγόριθμος και ποια μπορεί να διορθώσει. Τα bits που προστίθενται ονομάζονται bits ισοτιμίας (parity bits).

Μια απλή αλλά διαδεδομένη τεχνική είναι η προσθήκη στο τέλος της λέξης ενός bit που δηλώνει ότι τα εν λόγω δεδομένα έχουν άρτιο ή περιττό αριθμό από '1'. Μια άλλη δημοφιλής τεχνική είναι ο κώδικας Hamming.

Στα συστήματα RAID ο πιο συνηθισμένος τρόπος για την διόρθωση λαθών είναι η χρήση της λογικής πράξης XOR. Έστω ότι έχουμε n ακολουθίες bit: $d_1, d_2, \ldots, d_n$. Εφαρμόζοντας την XOR σε αυτές έχουμε: $p = d_1 \oplus d_2 \oplus \ldots \oplus d_n$. Οποιαδήποτε από τις αρχικές ακολουθίες μπορεί να κατασκευαστεί από τις υπόλοιπες και το προηγούμενο αποτέλεσμα εφαρμόζοντας πάλι την XOR. ένα παράδειγμα ανάκτησης δεδομένων με αυτόν τον τρόπο παρουσιάζεται στο σχήμα 1.5.



Σχήμα 1.5: Παράδειγμα ανάκτησης δεδομένων με τη λογική πράξη XOR

1.4.4 RAID 2

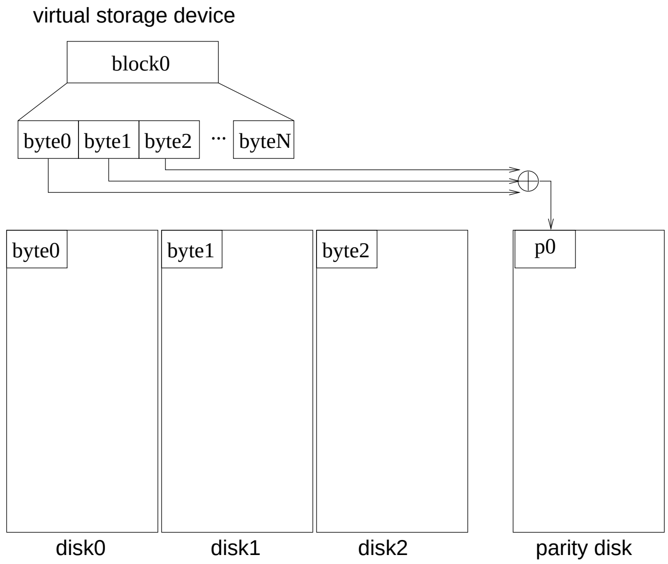
Το RAID 2 κατανέμει τα δεδομένα στους δίσκους σε επίπεδο bit χρησιμοποιώντας τον κώδικα Hamming. Υπάρχουν δύο κατηγορίες δίσκων, οι δίσκοι δεδομένων και οι δίσκοι ισοτιμίας. Ο αριθμός των δίσκων δεδομένων ισούται με το μήκος της λέξης και ο αριθμός των δίσκων ισοτιμίας ισούται με τον αριθμό bit ισοτιμίας που χρησιμοποιείται από τον κώδικα Hamming που υλοποιείται. Το RAID 2 δεν είναι αποδοτικό από πολλές απόψεις και δεν χρησιμοποιείται ποτέ στην πράξη.

1.4.5 RAID 3 και RAID 4

Τα δύο αυτά πρωτόκολλα λειτουργούν με τον ίδιο τρόπο. Υπάρχει ένας δίσκος για την αποθήκευση των δεδομένων ισοτιμίας, τα οποία προκύπτουν εφαρμόζοντας την πράξη XOR. Τα δεδομένα κατανέμονται διαδοχικά στους υπόλοιπους δίσκους. Η διαφορά τους έγκειται στο ότι το RAID 3 κατανέμει τα δεδομένα ανά byte ενώ το RAID 4 ανά block.

Στα σχήματα 1.6 και 1.7 παρουσιάζεται ένα παράδειγμα εφαρμογής των πρωτοκόλλων.

Σε αντίθεση με το RAID 2 είναι απαραίτητος μόνο ένας δίσκος για την αποθήκευση των δεδομένων ισοτιμίας. Τα συστήματα RAID 3 και 4 μπορούν να συνεχίσουν να λειτουργούν χωρίς πρόβλημα σε περίπτωση απώλειας ενός δίσκου, είτε ο δίσκος αυτός είναι δίσκος ισοτιμίας, είτε είναι δίσκος δεδομένων. Η χρήση ενός δίσκου για την αποθήκευση της πλεονάζουσας πληροφορίας, δίνει τη δυνατότητα να επιτύχουμε εύκολα μεγάλες χωρητικότητες στο σύστημα, διατηρώντας παράλληλα έναν αρκετά μεγάλο βαθμό αξιοπιστίας.



Σχήμα 1.6: Παράδειγμα κατανομής δεδομένων στο RAID 3

Στο RAID 3 κάθε εγγραφή ή ανάγνωση απαιτεί την συμμετοχή όλων των σκληρών δίσκων. Το γεγονός αυτό δημιουργεί πρόβλημα στην ταχύτητα απόκρισης του συστήματος αποθήκευσης. Αντίθετα το RAID 4 έχει καλύτερη επίδοση, αφού διανέμει τα δεδομένα στους δίσκους ανά block, δίνοντας έτσι τη δυνατότητα στο σύστημα να επεξεργαστεί και να αποκριθεί παράλληλα σε περισσότερες από μία αιτήσεις τη φορά.

1.4.6 RAID 5

Το RAID 5, σε αντίθεση με τα RAID 3 και 4, δεν χρησιμοποιεί έναν δίσκο για την αποθήκευση των δεδομένων ισοτιμίας. Τα δεδομένα ισοτιμίας κατανέμονται και αυτά διαδοχικά σε όλους τους δίσκους της διάταξης. Το RAID 5, όπως και το RAID 4 διανέμει τα δεδομένα ανά block. Ένα παράδειγμα λειτουργίας του παρουσιάζεται στο σχήμα 1.8.

Το RAID 5 επιτυγχάνει καλύτερη επίδοση από το RAID 4, αφού στο RAID 4 οι αιτήσεις για εγγραφές σειριοποιούνται πάντα στο δίσκο που αποθηκεύονται τα δεδομένα ισοτιμίας. Κατανέμοντας και τα δεδομένα ισοτιμίας στους δίσκους οι αιτήσεις για εγγραφές μπορούν να παραλληλοποιηθούν, οδηγώντας σε καλύτερους χρόνους προσπέλασης.

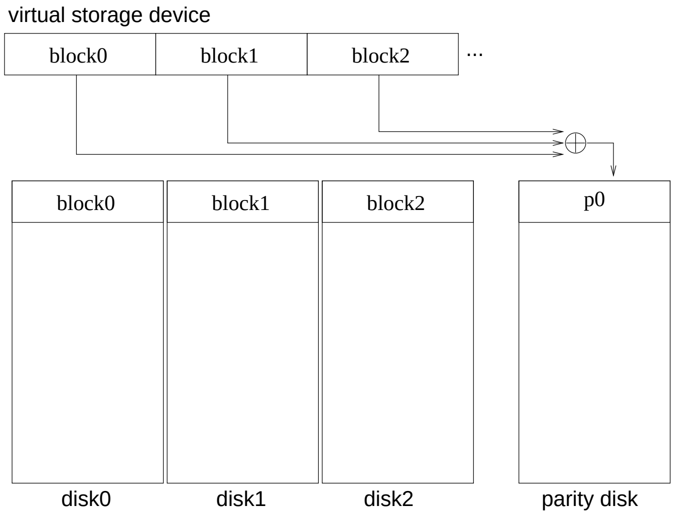
1.4.7 Ιεραρχική χρήση περισσοτέρων του ενός RAID επιπέδων

Δεν είναι λίγες οι φορές που επιλέγεται ένας συνδυασμός, επιπέδων RAID, ώστε το σύστημα αποθήκευσης να ανταποκριθεί στις ανάγκες της κάθε εφαρμογής. Ανάλογα με την διαστρωμάτωση των επιπέδων προκύπτει και η αντίστοιχη ονομασία του συστήματος.

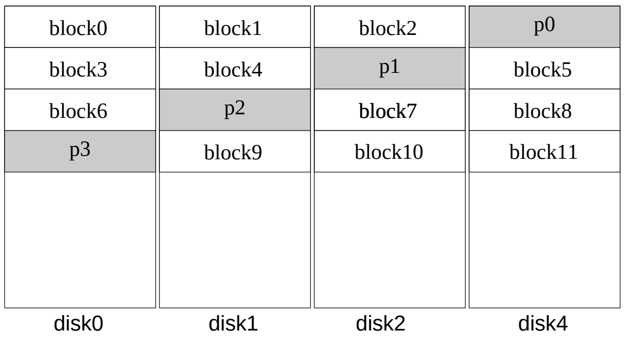
Στην αρχιτεκτονική αυτή, συνήθως χρησιμοποιούνται οι δύο ακραίες διατάξεις RAID, RAID 0 και 1. Οι δύο δυνατοί συνδυασμοί των αρχιτεκτονικών αυτών παρουσιάζονται στο σχήμα 1.9.

1.4.8 Υλοποίηση και απόδοση του RAID

Το πρότυπο RAID αν και καθορίζει σαφώς το πρωτόκολλο δεν αναφέρεται καθόλου στο πως θα υλοποιηθεί. Υπάρχουν δύο βασικοί τρόποι υλοποίησής του. Σε επίπεδο λογισμικού



Σχήμα 1.7: Παράδειγμα κατανομής δεδομένων στο RAID 4



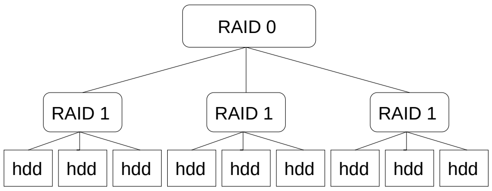
Σχήμα 1.8: Παράδειγμα κατανομής δεδομένων στο RAID 5

από το λειτουργικό σύστημα ή σε επίπεδο υλικού με τη βοήθεια ενός ελεγκτή RAID (RAID Controller). Η υλοποίηση του RAID σε επίπεδο υλικού είναι πιο αποδοτική από την υλοποίηση σε επίπεδο λογισμικού, αφού το λειτουργικό σύστημα δεν επιβαρύνεται με την υλοποίηση του πρωτοκόλλου.

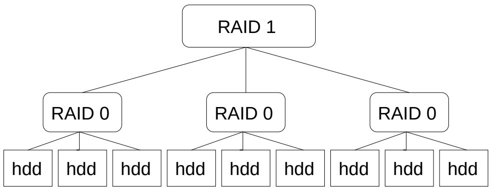
Ένας άλλος σημαντικός παράγοντας για την απόδοση του RAID είναι το υποσύστημα Ε/Ε που χρησιμοποιείται. Στην περίπτωση χρήσης διαμοιρασμένου διαδρόμου, η παράλληλη χρήση των επιμέρους δίσκων δεν μπορεί να γίνει αποδοτικά. Για αυτόν το λόγο είναι ιδιαίτερα δημοφιλής στις μέρες μας η χρήση δίσκων SATA, που χρησιμοποιούν συνδέσεις σημείο προς σημείο, για την δημιουργία διατάξεων RAID.

1.5 Αποθήκευση Δεδομένων

Ένα σημαντικό κομμάτι των τεχνολογιών αποθήκευσης δεδομένων είναι το πως αλληλεπιδρά ο χρήστης ή πιο συγκεκριμένα η εφαρμογή που χρησιμοποιεί ο χρήστης με τις συσκευές αποθήκευσης. Ο βασικός ρόλος των συσκευών αποθήκευσης είναι να προσφέρουν πρόσβαση



(a) RAID 1+0



(b) RAID 0+1

Σχήμα 1.9: Ιεραρχική χρήση της αρχιτεκτονικής RAID

σε έναν ενιαίο και γραμμικό χώρο. Από εκεί και πέρα ο χρήστης μπορεί να διανέμει τα δεδομένα του όπως κρίνει αυτός.

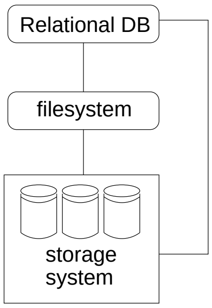
Στην πράξη, ωστόσο, ο χρήστης σπάνια αλληλεπιδρά απευθείας με τον γραμμικό χώρο της συσκευής αποθήκευσης. Τις περισσότερες φορές υλοποιείται ένα ενδιάμεσο επίπεδο, το οποίο αφενός προσφέρει στον χρήστη έναν πιο βολικό τρόπο για να αποθηκεύσει και να ανακτήσει τα δεδομένα του και αφετέρου φροντίζει την κατανομή των δεδομένων αυτών στον αποθηκευτικό χώρο.

Δύο από τους πιο διαδεδομένους τρόπους για την αλληλεπίδραση του χρήστη με τα δεδομένα είναι τα συστημάτων αρχείων (File Systems) και οι σχεσιακές βάσεις δεδομένων (Relational DataBases).

Τα συστήματα αρχείων έχουν δύο βασικές δομές αποθήκευσης, τους καταλόγους και τα αρχεία. Στα αρχεία είναι δυνατόν να τοποθετηθούν δεδομένα, με όποιον τρόπο επιλέξει ο χρήστης. Οι κατάλογοι αποτελούν σύνολα αρχείων και άλλων καταλόγων, προσφέροντας δυνατότητα για ιεραρχική ταξινόμηση των δεδομένων.

Στις σχεσιακές βάσεις δεδομένων τα δεδομένα οργανώνονται σε ένα σύνολο από πίνακες. Τα δεδομένα μπορούν να ανακτηθούν με πολλούς διαφορετικούς τρόπους χωρίς να χρειάζεται να αναδιοργανωθούν οι πίνακες. Για την αλληλεπίδραση του χρήστη με την βάση δεδομένων χρησιμοποιείται η γλώσσα SQL (Structured Query Language).

Τα συστήματα αρχείων συνήθως αποθηκεύουν τα δεδομένα απευθείας στη συσκευή αποθήκευσης. Οι βάσεις δεδομένων είτε χρησιμοποιούν ένα σύστημα αρχείων είτε, σε πιο σπάνιες περιπτώσεις, την ίδια την συσκευή αποθήκευσης για να τοποθετήσουν τα δεδομένα.



Σχήμα 1.10: Χρήση των αποθηκευτικών συστημάτων από συστήματα αρχείων και σχεσιακές βάσεις δεδομένων

Θα πρέπει να σημειωθεί, ότι τα συστήματα αυτά δεν αποθηκεύουν μόνο τα δεδομένα του χρήστη, αλλά και ορισμένα δεδομένα τα οποία σχετίζονται με το πως είναι οργανωμένα τα δεδομένα του χρήστη. Τα δεδομένα αυτά ονομάζονται μετα-δεδομένα (meta-data).

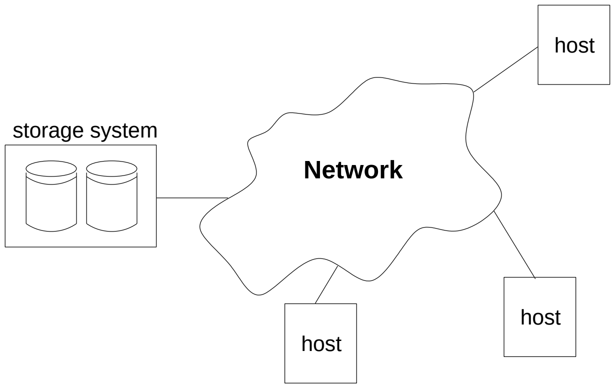
1.6 Συστήματα Αποθήκευσης και δικτυακές τεχνολογίες

Μέχρι στιγμής έχουμε αναφερθεί σε τεχνολογίες αποθήκευσης που εφαρμόζονται τοπικά, δηλαδή τεχνολογίες που αποτελούν κομμάτι του ίδιου του υπολογιστικού συστήματος που τις χρησιμοποιεί. Μια παρουσίαση των συστημάτων αποθήκευσης, όμως, σε καμία περίπτωση δεν είναι πλήρης αν δεν γίνει αναφορά στις δικτυακές τεχνολογίες που χρησιμοποιούνται για την βελτίωση και επέκταση των τοπικών τεχνολογιών αποθήκευσης.

Τον τελευταίο καιρό είναι ιδιαίτερα δημοφιλείς οι όροι Network Attached Storage (NAS) και Storage Area Network (SAN). Και οι δύο σχετίζονται με την χρησιμοποίηση δικτυακών τεχνολογιών στα συστήματα αποθήκευσης, σε διαφορετικό όμως επίπεδο.

1.6.1 NAS

Η βασική αρχή των συστημάτων NAS είναι η τοποθέτηση ενός συστήματος αποθήκευσης σε ένα δίκτυο, ώστε ο χώρος του να είναι προσπελάσιμος από όλα τα υπολογιστικά συστήματα που συνδέονται σε αυτό. Βασικός στόχος τους είναι να επιτρέψουν σε περισσότερα από ένα υπολογιστικά συστήματα και κατά συνέπεια χρήστες, να μοιράζονται δεδομένα, διευκολύνοντας έτσι την συνεργασία τους. Επιπρόσθετα, η παροχή ενός κεντρικού χώρου αποθήκευσης βοηθάει στην καλύτερη διαχείριση και αξιοποίηση του.



Σχήμα 1.11: Network Attached Storage

1.6.2 Δικτυακά συστήματα αρχείων

Μια αρκετά διαδεδομένη αρχιτεκτονική NAS είναι τα δικτυακά συστήματα αρχείων. Τα συστήματα αυτά ακολουθούν το μοντέλο εξυπηρετητή - πελάτη (server - client). Ο εξυπηρετητής, έχοντας δημιουργήσει ένα τοπικό σύστημα αρχείων σε μία συσκευή αποθήκευσης, το εξάγει στους πελάτες. Οι πελάτες, μέσω ενός κατάλληλου πρωτοκόλλου, προσφέρουν στο χρήστη ένα απομακρυσμένο σύστημα αρχείων για να αποθηκεύσει δεδομένα.

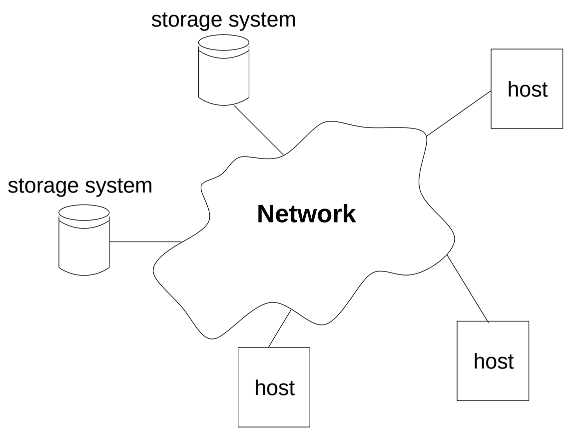
Ένα από τα πιο διαδεδομένα δικτυακά συστήματα αρχείων είναι το NFS το οποίο αναπτύχθηκε αρχικά από την εταιρία Sun Microsystems το 1984. Οι εκδόσεις 2, 3 του NFS χρησιμοποιούν το δικτυακό πρωτόκολλο UDP (User Datagram Protocol), αν και από την έκδοση

3 και μετά είναι δυνατή η χρήση και του TCP (Transmission Control Protocol). Μέχρι και την έκδοση 3 το πρωτόκολλο του NFS δεν χρειαζόταν να διατηρεί πληροφορίες για την κατάσταση (state) του κάθε πελάτη. Τα πρωτόκολλα αυτού του είδους ονομάζονται stateless. Η έκδοση 4 του NFS εισάγει ένα πρωτόκολλο που διατηρεί πληροφορίες για την κατάσταση για κάθε πελάτη (statefull). Το πρωτόκολλο NFS απευθύνεται κυρίως σε UNIX συστήματα.

Αντίθετα για συστήματα που χρησιμοποιούν το λειτουργικό σύστημα Windows της Microsoft, το πιο συνηθισμένο δικτυακό σύστημα αρχείων είναι το SMB (Server message block). Το SMB είναι ένα πρωτόκολλο για τον διαμοιρασμό (sharing) όχι μόνο αρχείων αλλά και άλλων πόρων, όπως εκτυπωτές και σειριακές θύρες. Αναπτύχθηκε αρχικά από την IBM, αλλά υιοθετήθηκε και επεκτάθηκε σε μεγάλο βαθμό από την Microsoft, η οποία και το μετονόμασε σε CIFS (Common Internet File System). Το SMB αρχικά χρησιμοποιούσε το NetBIOS (Network Basic Input/Output System) και αργότερα επανασχεδιάστηκε ώστε να μπορεί να χρησιμοποιεί κατευθείαν το TCP. Το SAMBA, είναι μια ελεύθερη υλοποίηση του SMB, ιδιαίτερα δημοφιλής στα UNIX συστήματα.

1.6.3 SAN

Ένα βασικό πρόβλημα των αρχιτεκτονικών NAS είναι ότι δεν είναι επεκτάσιμες (scalable). Αυτό συμβαίνει διότι υπάρχει μόνο ένα σημείο πρόσβασης τους μέσω του δικτύου. Ο παραπάνω περιορισμός οδήγησε στην δημιουργία των συστημάτων SAN. Στα συστήματα SAN οι ίδιες οι συσκευές αποθήκευσης συνδέονται με ένα δίκτυο διασύνδεσης, στο οποίο μπορούν να συνδεθούν ένα ή περισσότερα υπολογιστικά συστήματα. Τα υπολογιστικά συστήματα αυτά θα έχουν πρόσβαση στον αποθηκευτικό χώρο, είτε αυτός παρουσιάζεται ενιαίος, είτε όχι. Η κατασκευή ενός δικτύου για τις αποθηκευτικές συσκευές δημιουργεί τις κατάλληλες συνθήκες για την υποστήριξη μεγάλων αποθηκευτικών χώρων και για την επίτευξη καλύτερης απόδοσης, και επεκτασιμότητας. Θα πρέπει να αναφερθεί ότι το ακρωνύμιο SAN χρησιμοποιείται και για τον όρο System Area Network.



Σχήμα 1.12: Storage Area Network

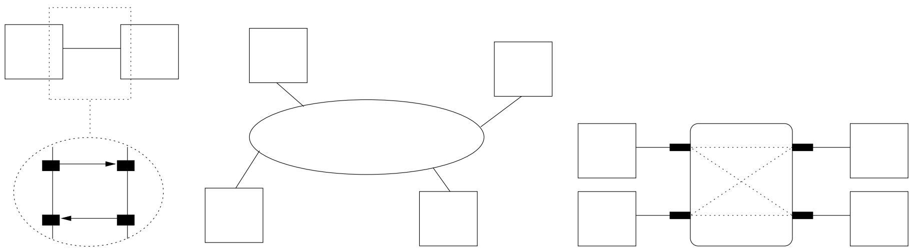
1.6.4 Fibre Channel

Το Fibre Channel (FC) είναι ένα σύνολο προτύπων για αποδοτική σειριακή μετάδοση δεδομένων σε ταχύτητες της τάξης Gbit/sec. Αν και τις περισσότερες φορές χρησιμοποιούνται οπτικά μέσα ως μέσο μετάδοσης των δεδομένων, το πρότυπο υποστηρίζει και μετάδοση σε κλασικές γραμμές μεταφοράς από χαλκό (twisted pair). Το Fibre Channel είναι ιδιαίτερα δημοφιλές στις αρχιτεκτονικές SAN, όπου χρησιμοποιείται για να συνδέσει τις συσκευές αποθήκευσης με τις υπολογιστικές μονάδες.

Οι συνδέσεις στο FC πραγματοποιούνται με δύο ζεύξεις, μία για την αποστολή δεδομένων και μια για την παραλαβή. Υπάρχουν τρεις βασικές τοπολογίες για την σύνδεση κόμβων σε ένα δίκτυο FC.

- **σημείο προς σημείο (point to point):**
  Η τοπολογία αυτή είναι η πιο απλή που μπορεί να υπάρξει και συνδέει δύο συσκευές μεταξύ τους. Στις περισσότερες των περιπτώσεων η σύνδεση αυτή είναι μεταξύ ενός εξυπηρετητή και μίας συσκευής αποθήκευσης.
- **Διαιτητεύσιμος δακτύλιος (arbitrary loop):**
  Η τοπολογία αυτή είναι ένας ένας δακτύλιος, στον οποίον συνδέονται συσκευές και μονάδες αποθήκευσης. Ο δακτύλιος χαρακτηρίζεται ως διαιτητεύσιμος διότι κάθε στιγμή μόνο δύο συσκευές μπορούν να επικοινωνούν, ενώ οι υπόλοιπες θα πρέπει να περιμένουν. Ισοδύναμη τοπολογία με αυτή του δακτυλίου είναι η σύνδεση όλων των συσκευών σε μία συσκευή hub, που επιτρέπει κεντρικοποιημένη διαχείριση.

  Δεν είναι λίγες οι φορές που χρησιμοποιούνται δύο δακτύλιοι για την σύνδεση των κόμβων, ώστε αν παρουσιαστεί πρόβλημα στον έναν να είναι δυνατή η συνέχιση της λειτουργίας. Ο δακτύλιος συχνά χρησιμοποιείται ως φθηνή λύση όταν ο αριθμός των κόμβων προς σύνδεση είναι μικρός.
- **πλέγμα μεταγωγής (switched fabric):**
  Η πιο επεκτάσιμη και αποδοτική, αλλά παράλληλα και υψηλή σε κόστος τοπολογία είναι το πλέγμα μεταγωγής. Η τοπολογία αυτή μπορεί να υποστηρίξει μέχρι και 16 εκατομμύρια συνδεδεμένους κόμβους και κάθε σύνδεση λειτουργεί αυτόνομα και με τον μέγιστο δυνατό ρυθμό μεταφοράς δεδομένων σαν να ήταν σύνδεση σημείο προς σημείο.



(a) σημείο προς σημείο (b) δακτύλιος (c) πλέγμα μεταγωγής

Σχήμα 1.13: Βασικές Τοπολογίες των δικτύων Fibre Channel

Βέβαια, οι τοπολογίες αυτές μπορούν να συνδυαστούν για την δημιουργία πιο σύνθετων τοπολογιών που καλύπτουν τις ανάγκες της κάθε εφαρμογής.

Το πρωτόκολλο του FC χωρίζεται σε πέντε στρώματα. Το πιο χαμηλό στρώμα είναι το FC0 και διαπραγματεύεται το φυσικό επίπεδο. Τα FC1 και FC2 υλοποιούν το επίπεδο ζεύξης δεδομένων και το επίπεδο δικτύου του μοντέλου OSI (Open Systems Interconnection) αντίστοιχα.

Υποστηρίζονται οι εξής διαφορετικές κλάσεις επικοινωνίας:

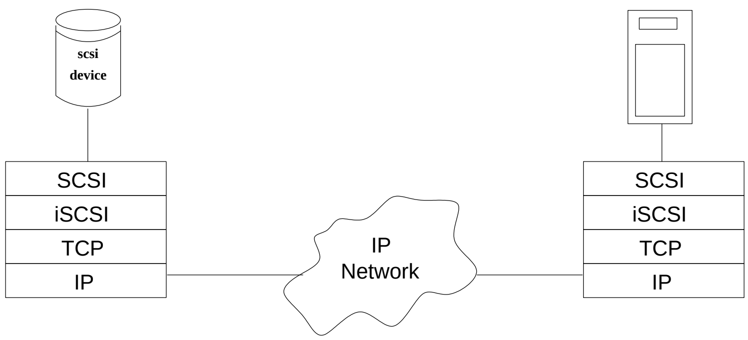
- επικοινωνία με σύνδεση και επιβεβαίωση λήψης (acknowledged connection oriented)
- επικοινωνία χωρίς σύνδεση με επιβεβαίωση λήψης (acknowledged connectionless)
- επικοινωνία χωρίς σύνδεση και χωρίς επιβεβαίωση λήψης (unacknowledged connectionless)

Υπάρχει ακόμα ένα λεπτό επίπεδο (FC3) που υλοποιεί βοηθητικές λειτουργίες, που σκοπό έχουν να εκμεταλλευτούν στο έπακρο τις δυνατότητες του Fibre Channel. Το υψηλότερο επίπεδο (FC4) χρησιμοποιείται για να απεικονίζονται υπάρχουσες τεχνολογίες, ώστε να χρησιμοποιούν το FC για την μεταφορά των δεδομένων τους. Ένα από τα πρωτόκολλα του FC4 επιτρέπει στο πρότυπο SCSI να μεταφέρει τα δεδομένα του μέσω ενός δικτύου FC, δημιουργώντας τις κατάλληλες προϋποθέσεις για την χρήση του ως SAN.

1.6.5 iSCSI

Αν και η χρήση Fibre Channel δικτύων είναι αρκετά αποδοτική και επεκτάσιμη, τις περισσότερες φορές προϋποθέτει πλήρη αναδιοργάνωση των υποδομών, ώστε να εγκατασταθεί. Ο παραπάνω λόγος σε συνδυασμό με το ότι το κόστος εγκατάστασης του παραμένει αρκετά υψηλό, οδήγησε στην ανάπτυξη ενός άλλου προτύπου.

Το πρότυπο iSCSI (Internet SCSI) σχεδιάστηκε για να προσφέρει την δυνατότητα επικοινωνίας με συσκευές Ε/Ε, χρησιμοποιώντας δίκτυα τύπου TCP/IP και το πρωτόκολλο SCSI. Με αυτόν τον τρόπο είναι δυνατόν να χρησιμοποιηθούν οι υπάρχουσες δικτυακές υποδομές στην δημιουργία ενός SAN. Οι αποθηκευτικές συσκευές συνδέονται κατευθείαν στο δίκτυο και απεικονίζονται σαν κόμβοι αυτού.

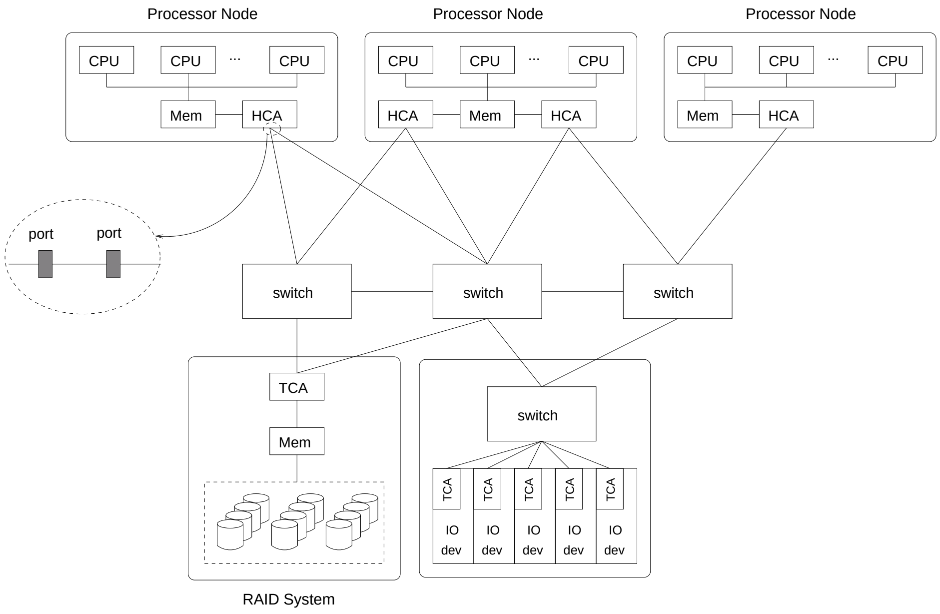


Σχήμα 1.14: Διαστρωμάτωση του πρωτοκόλλου iSCSI

Η χρήση του πρωτοκόλλου TCP/IP προσθέτει σημαντική επιβάρυνση στην επικοινωνία μεταξύ των υπολογιστικών μονάδων και των αποθηκευτικών συστημάτων. Με την διάδοση όμως του Gbit Ethernet, το οποίο υποστηρίζει ταχύτητα μετάδοσης δεδομένων 1Gbit/sec και την ανάπτυξη του 10Gbit Ethernet (10Gbit/sec) η επιβάρυνση αυτή γίνεται όλο και μικρότερη. Ένας άλλος τρόπος για να αρθεί η επιβάρυνση που προκαλεί το πρωτόκολλο TCP/IP είναι να αναλάβει ο ίδιος ο προσαρμογέας του iSCSI την επεξεργασία που απαιτείται, αντί για το λειτουργικό σύστημα. Η τεχνολογία αυτή ονομάζεται TCP Offload Engine (TOE) και είναι αρκετά διαδεδομένη σε αρχιτεκτονικές iSCSI.

1.6.6 Infiniband

Το Infiniband είναι ένα δίκτυο κατάλληλο για να συνδέσει ανεξάρτητες επεξεργαστικές πλατφόρμες, συστήματα αποθήκευσης και συσκευές αποθήκευσης. Χαρακτηρίζεται ως System Area Network, αλλά μπορεί να χρησιμοποιηθεί και ως Storage Area Network. Με το Infiniband μπορούν να κατασκευαστούν από μικρά έως πάρα πολύ μεγάλα δίκτυα, προσφέροντας μεγάλη επεκτασιμότητα.



Σχήμα 1.15: Παράδειγμα Infiniband Δικτύου

Το Infiniband χρησιμοποιεί τοπολογία πλέγματος μεταγωγής (switched fabric), επιτρέποντας σε πολλές συσκευές να επικοινωνούν ταυτόχρονα με χαμηλή καθυστέρηση και υψηλούς ρυθμούς μετάδοσης δεδομένων. Χρησιμοποιεί συνδέσεις σημείου προς σημείο και και σειριακή μετάδοση δεδομένων.

Κάθε μέλος του δικτύου συνδέεται σε αυτό με έναν προσαρμογέα διαύλου (channel adapter). Ο κάθε προσαρμογέας μπορεί να χρησιμοποιεί ένα ή περισσότερα σημεία εισόδου στο δίκτυο (ports). Επιπρόσθετα, ένα μέλος του δικτύου μπορεί να χρησιμοποιεί για τη σύνδεση του περισσότερους από έναν προσαρμογείς.

Η αρχιτεκτονική αυτή επιτρέπει την αποδοτική λειτουργία σε περίπτωση που υπάρχουν πολλαπλά νήματα επεξεργασίας (threads) σε έναν κόμβο, όπως επίσης και τη χρήση πολλαπλών μονοπατιών (multiple paths) για την καλύτερη αντιμετώπιση σφαλμάτων (fault tolerance) και αυξημένους ρυθμούς μετάδοσης. Οι προσαρμογείς ως βασική λειτουργία τους έχουν τη μεταφορά δεδομένων χρησιμοποιώντας απευθείας πρόσβαση στη μνήμη (Direct Memory Access - DMA).

Οι προσαρμογείς διαύλου χωρίζονται σε δύο βασικές κατηγορίες:

- Προσαρμογείς διαύλου κόμβου
  (Host Channel Adapters - HCA)
- Προσαρμογείς διαύλου προορισμού
  (Target Channel Adapters - TCA)

Οι HCA χρησιμοποιούνται για να εντάξουν στο δίκτυο επεξεργαστικές μονάδες, ενώ οι TCA για μονάδες ή συστήματα αποθήκευσης.

Το Infiniband χρησιμοποιεί ως βασική δομή τις ουρές (queues) για να επικοινωνεί με το υψηλότερο επίπεδο, το οποίο ονομάζεται και διαδικασία καταναλωτής (consumer process). Οι ουρές δημιουργούνται πάντα σε ζεύγη (Queue Pair QP), μία για λειτουργίες αποστολής, και μία για λειτουργίες λήψης.

Σε γενικές γραμμές η μεταφορά δεδομένων μπορεί να περιγραφεί ως εξής: η διαδικασία καταναλωτής εισάγει εντολές στην ουρά αποστολής για τη μεταφορά δεδομένων από τη μνήμη της στη μνήμη μιας άλλης διαδικασίας, ενώ στην ουρά λήψης εισάγει πληροφορίες για το που θα τοποθετηθούν τα δεδομένα που θα παραληφθούν.

Το πρότυπο του Infiniband ορίζει τις εντολές με τις οποίες επικοινωνεί μια διαδικασία καταναλωτής με τον HCA. Οι λειτουργίες αυτές ονομάζονται Verbs και ορίζουν ένα πλαίσιο για την δημιουργία ενός API (Application Programming Interface). Στην περίπτωση του HCA η πιο συνηθισμένη διαδικασία καταναλωτής είναι το λειτουργικό σύστημα. Αντίθετα, στην περίπτωση των TCA το πρότυπο δεν ορίζει κάποιον προκαθορισμένο τρόπο για το πως θα χρησιμοποιηθούν.

1.6.7 Αρχιτεκτονικές SAN και συστήματα αρχείων

Οι τεχνολογίες SAN παρέχουν τη δυνατότητα για διασύνδεση των επιμέρους συσκευών αποθήκευσης, ώστε να επιτευχθεί επεκτασιμότητα, υψηλή ταχύτητα προσπέλασης και αυξημένη χωρητικότητα. Όπως έχει αναφερθεί, οι εφαρμογές που χρησιμοποιούν τον αποθηκευτικό χώρο συνήθως έχουν πρόσβαση σε αυτόν με τη βοήθεια ενός ενδιάμεσου επιπέδου όπως ένα σύστημα αρχείων.

Στην περίπτωση που μόνο ένα υπολογιστικό σύστημα έχει πρόσβαση στις αποθηκευτικές μονάδες ενός SAN, τότε μπορεί να χρησιμοποιηθεί ένα συνηθισμένο σύστημα αρχείων. Λέγοντας συνηθισμένο σύστημα αρχείων αναφερόμαστε σε ένα που υποθέτει κατά την λειτουργία του ότι μόνο αυτό έχει πρόσβαση στον αποθηκευτικό χώρο. Ο περιορισμός αυτός, όμως, δημιουργεί προβλήματα επεκτασιμότητας και δεν επιτρέπει την πλήρη εκμετάλλευση του SAN.

Για την πλήρη εκμετάλλευσή των δυνατοτήτων που προσφέρει το SAN θα πρέπει να υπάρχει η προοπτική για την σύνδεση παραπάνω από ενός υπολογιστικού συστήματος σε αυτό. Η χρήση ενός κοινού συστήματος αρχείων ωστόσο θα δημιουργούσε διάφορα προβλήματα. Τα σημαντικότερα από αυτά είναι:

- **Απουσία Συγχρονισμού:**
  Δεδομένου ότι τα συστήματα αρχείων στους υπολογιστικούς κόμβους θα αποθηκεύουν και θα ανακτούν δεδομένα χωρίς συγχρονισμό υπάρχει η πιθανότητα να απολεσθούν δεδομένα ή μετα-δεδομένα.

- **Προσωρινή Αποθήκευση Δεδομένων στη μνήμη (Caching):**
  Μια συνήθης πρακτική στα συστήματα αρχείων είναι η αποθήκευση δεδομένων που προέρχονται ή προορίζονται για το αποθηκευτικό σύστημά στην μνήμη. Με αυτόν τον τρόπο αυξάνεται η επίδοση, αφού αντί για το αποθηκευτικό σύστημα χρησιμοποιείται η μνήμη, στην οποία η πρόσβαση είναι πιο γρήγορη. Στην περίπτωση όπου δύο συστήματα αρχείων δρουν ανεξάρτητα χρησιμοποιώντας την ίδια συσκευή είναι αδύνατο να διατηρηθεί η συνάφεια των μεταξύ των δεδομένων που βρίσκονται στην μνήμη και στην συσκευής αποθήκευσης.

Για να αντιμετωπιστούν αυτά και άλλα προβλήματα δημιουργήθηκαν συστήματα αρχείων, που επιτρέπουν σε υπολογιστικά συστήματα να έχουν πρόσβαση σε κοινό αποθηκευτικό χώρο με αποδοτικό και επεκτάσιμο τρόπο. Παρακάτω, παρουσιάζονται οι βασικές αρχές δύο τέτοιων συστημάτων, του GFS και του Lustre.

1.6.8 Global File System

Το GFS είναι ένα σύστημα αρχείων, το οποίο επιτρέπει ταυτόχρονη πρόσβαση από πολλά υπολογιστικά συστήματα σε μία ή περισσότερες μονάδες αποθήκευσης Αυτά τα συστήματα αρχείων ονομάζονται συστήματα αρχείων διαμοιραζόμενου δίσκου (shared disk file systems).

Το GFS, αρχικά αναπτύχθηκε για το λειτουργικό σύστημα IRIX, αν και στις μέρες μας χρησιμοποιείται σχεδόν αποκλειστικά στο λειτουργικό σύστημα Linux.

Το GFS είναι πλήρως κατανεμημένο, με την έννοια ότι όλοι οι κόμβοι που συμμετέχουν σε αυτό είναι ισότιμοι. Τα δεδομένα, αλλά και τα μετα-δεδομένα του συστήματος αρχείων κατανέμονται σε όλες τις αποθηκευτικές συσκευές.

Αρχικά ο συγχρονισμός μεταξύ των κόμβων γινόταν με ειδικές εντολές SCSI για τον συγχρονισμό των επιμέρους κόμβων κατά την εγγραφή μετα-δεδομένων στις αποθηκευτικές συσκευές. Παράλληλα, για συσκευές που δεν υλοποιούσαν τις εντολές αυτές είχε δημιουργηθεί ένα επίπεδο προσομοίωσης τους μέσω ενός εξυπηρετητή, το οποίο και επικράτησε, αφού οι εντολές αυτές δεν υιοθετήθηκαν από το σύνολο των κατασκευαστών.

Η λύση της προσομοίωσης μέσω ενός εξυπηρετητή ήταν κεντρικοποιημένη και αντίθετη με τον κατανεμημένο χαρακτήρα του GFS. Για να επιτευχθεί καλύτερη απόδοση και μεγαλύτερη αξιοπιστία αναπτύχθηκαν μηχανισμοί κατανεμημένου συγχρονισμού (distributed locking) που χρησιμοποιούνται για την προστασία των δεδομένων και των μετα-δεδομένων του συστήματος αρχείων. Οι μηχανισμοί αυτοί συχνά αναφέρονται και ως Distributed Locking Managers (DLM). Η επικοινωνία των κόμβων, για την υλοποίηση των μηχανισμών αυτών, συνήθως γίνεται μέσω ενός τυπικού LAN δικτύου.

Στην αρχική υλοποίηση του GFS, συμπεριλαμβανόταν και ένα επίπεδο λογισμικού που επέτρεπε τον συνδυασμό πολλών διαφορετικών μονάδων αποθήκευσης σε ομάδες και υποομάδες. Ωστόσο, το Linux πλέον παρέχει τέτοιες δυνατότητες (βλ. παράγραφο 2.4) , οπότε κάτι τέτοιο δεν είναι απαραίτητο να υλοποιηθεί στο ίδιο το GFS.

1.6.9 Lustre

Το Lustre είναι ένα σύστημα αρχείων για το λειτουργικό σύστημα Linux, σχεδιασμένο για υψηλή απόδοση και μεγάλη επεκτασιμότητα. Ο στόχος του είναι να διαχειρίζεται αποδοτικά αποθηκευτικούς χώρους της τάξης του petabyte υποστηρίζοντας χιλιάδες πελάτες. Το όνομα Lustre προέρχεται από τις λέξεις Linux και cluster.

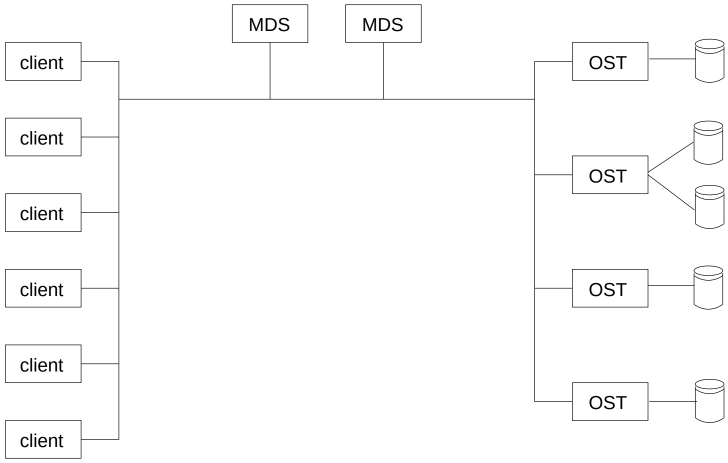
Το Lustre χρησιμοποιεί έναν δικό του κατανεμημένο μηχανισμό συγχρονισμού με ρόλο αντίστοιχο αυτού του GFS.

Υπάρχουν τρεις κατηγορίες οντοτήτων στα συστήματα Lustre:

- **Πελάτες (Clients):**
  Οι πελάτες είναι αυτοί στους οποίους εξάγεται ο αποθηκευτικός χώρος για αποθήκευση ή ανάκτηση δεδομένων.

- **Εξυπηρετητές μετα-δεδομένων (Metadata Servers - MDSs):**
  Οι MDS είναι οι οντότητες που είναι υπεύθυνες για την αποθήκευση και την διαχείριση των μετα-δεδομένων.

- **Κόμβοι αντικειμένων αποθήκευσης (Object Storage Targets - OSTs):**



Σχήμα 1.16: Αρχιτεκτονική του Lustre

> Οι OSTs είναι οι οντότητες που αλληλεπιδρούν με τις αποθηκευτικές συσκευές και φροντίζουν για την αποθήκευση και ανάκτηση των δεδομένων.

Βασική αρχή στον σχεδιασμό του Lustre είναι η χρήση αντικειμένων αποθήκευσης. Τα αντικείμενα αποθήκευσης είναι μια γενική αρχιτεκτονική, η οποία αναφέρεται και ως Object Store. Τα συνηθισμένα συστήματα αποθήκευσης προσφέρουν απλά πρόσβαση σε έναν ενιαίο γραμμικό χώρο. Αντίθετα, στις αρχιτεκτονικές αντικειμένων αποθήκευσης η πρόσβαση πραγματοποιείται μέσω αντικειμένων, αντίστοιχων με αυτών των αντικειμενοστραφών γλωσσών προγραμματισμού (Object oriented computer languages). Τα αντικείμενα αποτελούνται από ιδιότητες και μεθόδους, βάσει των οποίων πραγματοποιείται η πρόσβαση στον αποθηκευτικό χώρο. Η τεχνική αυτή έχει εμφανή προτερήματα σε σχέση με τις κλασικές προσεγγίσεις, αφού επιτρέπει στο αποθηκευτικό σύστημα να υλοποιήσει με όποιον τρόπο θέλει την αλληλεπίδραση με τις αποθηκευτικές συσκευές προσφέροντας πολλές δυνατότητες. Ένα άλλο παράδειγμα συστήματος αρχείων που βασίζεται σε object store αρχιτεκτονική είναι το zFS (βλ. [RT]).

Η επικοινωνία των κόμβων στο Lustre γίνεται με την χρήση ενός γενικού στρώματος επικοινωνίας Το στρώμα αυτό χρησιμοποιεί ένα γενικό API, το οποίο ονομάζεται Portals και υποστηρίζει πληθώρα δικτύων, όπως τα TCP/IP, Quadrics, Myrinet και SCI.

Η προσέγγιση του Lustre για την δημιουργία ενός επεκτάσιμου αποθηκευτικού συστήματος είναι διαφορετική από αυτή του GFS. Το Lustre, προϋποθέτει την ύπαρξη ενός OST, το οποίο θα είναι υπεύθυνο για ένα αποθηκευτικό σύστημα και κάθε συνδιαλλαγή των πελατών με αυτό. Ο παραπάνω περιορισμός φαίνεται να δημιουργεί πρόβλημα στην πλήρη αξιοποίηση των αρχιτεκτονικών SAN. Ωστόσο είναι δυνατό ένας πελάτης να επικοινωνήσει πρώτα με τον OST κόμβο και κατόπιν να μεταφέρει τα δεδομένα απευθείας από το SAN, εκμεταλλευόμενος έτσι πλήρως τις δυνατότητες που του παρέχει αυτό.

Κεφάλαιο 2

Συστήματα Αποθήκευσης στο Linux

Στόχος του κεφαλαίου αυτού είναι η ανάλυση του στρώματος του λειτουργικού συστήματος Linux, το οποίο είναι υπεύθυνο για την υποστήριξη συσκευών αποθήκευσης. Επιπρόσθετα παρουσιάζεται και η διεπαφή, η οποία χρησιμοποιείται για την ανάπτυξη οδηγών (drivers) για συσκευές αποθήκευσης.

2.1 Εισαγωγή

2.1.1 Γενικά για το Linux

Το Linux είναι ένα λειτουργικό σύστημα (ΛΣ) τύπου Unix, το οποίο αναπτύσσεται από μια κοινότητα προγραμματιστών. Ο κώδικας του διατίθεται ελεύθερα στο διαδίκτυο, σύμφωνα με την άδεια GPL. Ο πυρήνας του Linux είναι μονολιθικός (monolithic), αλλά υποστηρίζει αντικείμενα τα οποία μπορούν να προσκολληθούν και να αποκολληθούν δυναμικά στον και από τον πυρήνα (dynamic modules).

Το Linux, αρχικά δημιουργήθηκε από τον Linus Torvalds, ο οποίος δημοσίευσε τον πηγαίο κώδικα σε μια ομάδα συζητήσεων (newsgroup) το 1991. Ο κώδικας διαδόθηκε και επεκτάθηκε μέσω του διαδικτύου, για να εξελιχθεί σε ένα πλήρες και ανταγωνιστικό ΛΣ. Αν και αρχικά δημιουργήθηκε για τους επεξεργαστές 386 της Intel, στις μέρες μας το Linux υποστηρίζει μια πληθώρα αρχιτεκτονικών.

Το Linux είναι στενά συνδεδεμένο με το GNU project, του οποίου στόχος είναι η δημιουργία ελεύθερου λογισμικού. Το βασικότερο κομμάτι των εφαρμογών χώρου χρήστη σε ένα Linux σύστημα προέρχονται από το GNU project. Κυριότερο παράδειγμα τέτοιας εφαρμογής είναι ο gcc, ένας μεταγλωττιστής για τη γλώσσα C, που χρησιμοποιείται κατά κόρον για την μεταγλώττιση του πυρήνα.

2.1.2 Εικονικά αρχεία συσκευών

Στο Linux, όπως και στα άλλα ΛΣ τύπου UNIX κυριαρχεί το ρητό "everything is a file" (όλα είναι ένα αρχείο). Μια εφαρμογή του ρητού αυτού είναι η χρήση του αρχείου ως κυρίαρχη διεπαφή μεταξύ του χώρου χρήστη (userspace) και του χώρου πυρήνα (kernelspace). Παραδείγματα τέτοιας χρήσης είναι εικονικά συστήματα αρχείων, όπως το proc και αρχεία οδηγών συσκευών.

Τα αρχεία οδηγών συσκευών είναι αρχεία, που χρησιμοποιούνται για την αλληλεπίδραση του χρήστη και των οδηγών περιφερειακών συσκευών του υπολογιστικού συστήματος.

Τα ειδικά αρχεία αυτά χαρακτηρίζονται από τον τύπο τους και δύο αριθμούς, οι οποίοι ονομάζονται major και minor. Οι αριθμοί αυτοί χρησιμοποιούνται για την διαφοροποίηση των συσκευών. Οι συσκευές ιδίου τύπου συνήθως έχουν τον ίδιο αριθμό major και ελέγχονται από τον ίδιο οδηγό, ο οποίος τις διαχωρίζει βάσει του αριθμού minor.

Υπάρχουν δύο είδη τέτοιων αρχείων, ανάλογα με τον τύπο της συσκευής που αναπαριστούν:

- **Αρχεία που αναπαριστούν συσκευές χαρακτήρων (character devices):**
  Οι συσκευές αυτές είτε δεν μπορούν να προσπελαστούν σε τυχαία σημεία, όπως για παράδειγμα μία κάρτα ήχου (sound card) ή μπορούν να προσπελαστούν σε τυχαία σημεία αλλά ο χρόνος απόκρισης τους εξαρτάται σε μεγάλο βαθμό από τη θέση στην οποία βρίσκονται τη στιγμή εκείνη, όπως στην περίπτωση των συσκευών μαγνητικής ταινίας (magnetic tape).

- **Αρχεία που αναπαριστούν συσκευές block (block devices):**
  Οι συσκευές αυτές μπορούν να προσπελαστούν σε τυχαία σημεία, και (τουλάχιστον από την πλευρά του χρήστη) ο χρόνος προσπέλασης σε διαφορετικά σημεία είναι περίπου ο ίδιος. Η προσπέλαση στις συσκευές αυτές μπορεί να γίνει σε συγκεκριμένου μεγέθους μονάδες δεδομένων που ονομάζονται block. Οι συσκευές αποθήκευσής είναι συσκευές block.

2.1.3 Συσκευές block από τη σκοπιά του χώρου χρήστη

Υπάρχουν δύο τρόποι με τους οποίους ο χρήστης μπορεί να έχει πρόσβαση στις συσκευές block.

- **Χρησιμοποιώντας το αρχείο της συσκευής:**
  Χρησιμοποιώντας το αρχείο της συσκευής block ο χρήστης μπορεί να έχει απευθείας πρόσβαση στον γραμμικό χώρο αποθήκευσης της συσκευής, χρησιμοποιώντας κλήσεις συστήματος όπως `open()`, `lseek()`, `read()`, `write()` και `close()`. Επίσης, μπορεί να χρησιμοποιήσει κλήσεις συστήματος `ioctl()` για να δώσει εντολές ελέγχου στη συσκευή.

- **Μέσω ενός συστήματος αρχείων:**
  Αρχικά ο χρήστης πρέπει να δημιουργήσει ένα σύστημα αρχείων σε μια συσκευή χρησιμοποιώντας την εντολή `mkfs`. Η εντολή αυτή γράφει στο αρχείο της συσκευής, το οποίο ορίζει ο χρήστης, πληροφορίες που αφορούν το σύστημα αρχείων, το οποίο θα χρησιμοποιηθεί.

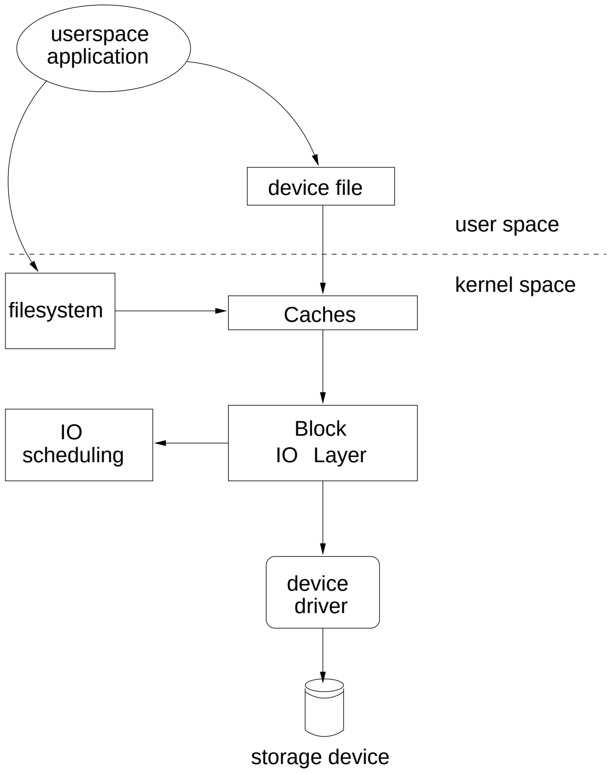
  Αφού αρχικοποιηθεί το σύστημα αρχείων στη συσκευή θα πρέπει να συσχετιστεί με κάποιον κατάλογο για να είναι δυνατή η πρόσβαση σε αυτό. Η συσχέτιση αυτή γίνεται με την εντολή `mount`. Μετά την επιτυχή εκτέλεση της εντολής αυτής, ο κατάλογος που προσδιορίστηκε αποτελεί τον αρχικό κατάλογο του συστήματος αρχείων που βρίσκεται στην συγκεκριμένη συσκευή.

2.1.4 Συσκευές block από τη σκοπιά του πυρήνα

Το Linux προσφέρει ένα γενικό στρώμα για την υποστήριξη των συσκευών block. Με αυτόν τον τρόπο δεν χρειάζεται κάθε οδηγός για μια συσκευή block να υλοποιεί πάλι βασικές λειτουργίες που είναι κοινές σε κάθε συσκευή. Επιπρόσθετα, δίνεται η δυνατότητα στο ΛΣ να βελτιστοποιήσει την απόδοση των συσκευών αυτών, χωρίς να χρειάζεται ο οδηγός να γνωρίζει την εσωτερική σχεδίαση του, παρά μόνο να χρησιμοποιεί την διεπαφή που του παρέχει αυτό. Οι πιο συνηθισμένες συσκευές αποθήκευσης είναι οι σκληροί δίσκοι που παρουσιάστηκαν στην παράγραφο 1.3.1.

Αν για κάθε εντολή ανάγνωσης ή εγγραφής χρησιμοποιούνταν απευθείας και άμεσα η συσκευή, η απόδοση δεν θα ήταν καλή. Για την βελτίωση της απόδοσης των συσκευών αποθήκευσης χρησιμοποιούνται δύο βασικές τεχνικές:

- **Χρήση ιεραρχίας μνήμης (caching):**



Σχήμα 2.1: Συσκευές αποθήκευσης στο Linux

Η πρόσβαση στις συσκευές αποθήκευσης είναι αργή σε σχέση με την πρόσβαση στη μνήμη. Για καλύτερη απόδοση και ταχύτητα απόκρισης χρησιμοποιούνται κρυφές μνήμες (caches) στην κυρία μνήμη για να κρατάνε τα δεδομένα των συσκευών αυτών.

Οι κρυφές μνήμες χρησιμοποιούνται και στις εγγραφές και στις αναγνώσεις. Η βελτίωση της απόδοσης εξασφαλίζεται από την τοπικότητας της αναφοράς (locality of reference), η οποία ισχύει και στα δεδομένα των αποθηκευτικών συσκευών.

- **Χρονοδρομολόγηση αιτήσεων Ε/Ε (I/O scheduling):**

  Όσο καλή και να είναι η χρήση των κρυφών μνημών, κάποια στιγμή θα πρέπει να εγγραφούν ή να αναγνωσθούν δεδομένα από την αποθηκευτική συσκευή. Οι λειτουργίες αυτές πραγματοποιούνται με την δημιουργία αιτήσεων Ε/Ε (I/O requests) στην συσκευή αυτή.

  Στις περισσότερες αποθηκευτικές συσκευές είναι αποδοτικότερο να μην ικανοποιούνται άμεσα οι αιτήσεις αυτές, αλλά μέσω κάποιου αλγόριθμου να συγκεντρώνεται ένα σύνολο αιτήσεων και να ικανοποιούνται μαζικά.

  Για παράδειγμα στους σκληρούς δίσκους είναι πιο αποδοτικό να συγκεντρώνονται και να ικανοποιούνται μαζί οι αιτήσεις που αφορούν γειτονικά block, ώστε να ελαχιστοποιηθεί η μετακίνηση του βραχίονα που χρησιμοποιείται.

  Απαραίτητη προϋπόθεση είναι ο παραπάνω αλγόριθμος να μην μειώνει σε αισθητό βαθμό την καθυστέρηση που βιώνει ο χρήστης όταν αλληλεπιδρά με τις συσκευές αποθήκευσης. Οι αλγόριθμοι αυτοί ονομάζονται αλγόριθμοι χρονοδρομολόγησης Ε/Ε (I/O scheduling alogorithms).

2.2 Βασικές δομές δεδομένων του στρώματος block

Παρακάτω θα παρουσιαστούν οι βασικότερες δομές δεδομένων που σχετίζονται με το στρώμα block του Linux.

2.2.1 Σελίδες

Το Linux, όπως και τα περισσότερα λειτουργικά συστήματα, αναλαμβάνει να δημιουργήσει έναν ενιαίο εικονικό χώρο μνήμης στις διεργασίες που εκτελούνται σε αυτό. Για να το πετύχει αυτό, ένας από τους μηχανισμούς που χρησιμοποιεί είναι αυτός της σελιδοποίησης, όπου η φυσική μνήμη του συστήματος χωρίζεται σε σελίδες ίδιου μεγέθους. Η τιμή του μεγέθους της σελίδας για την αρχιτεκτονική i386 είναι 4096 bytes. Η δομή δεδομένων που ορίζεται για την περιγραφή της κάθε φυσικής σελίδας είναι η `struct page`, η οποία όπως θα δούμε χρησιμοποιείται από το στρώμα block του Linux.

2.2.2 Block μονάδες Ε/Ε

Η βασική μονάδα Ε/Ε για τις συσκευές block είναι η δομή `struct bio`, η ονομασία της οποίας προέρχεται από τη φράση Block IO. Οποιοδήποτε κομμάτι του Linux θέλει να επικοινωνήσει με την συσκευή block θα χρησιμοποιήσει, έμμεσα ή άμεσα, την δομή αυτή.

Ένα αντικείμενο `bio` περιέχει ένα σύνολο σελίδων, οι οποίες σχετίζονται με την εν λόγω μεταφορά δεδομένων. Συγκεκριμένα, σε περίπτωση αποθήκευσης δεδομένων στην συσκευή οι σελίδες αυτές περιέχουν τα δεδομένα προς εγγραφή. Αντίθετα, σε περίπτωση ανάκτησης δεδομένων, οι σελίδες αυτές αποτελούν τον χώρο μνήμης, στον οποίο θα μεταφερθούν τα δεδομένα. Οι σελίδες που σχετίζονται με μια μονάδα `bio` περιέχονται σε έναν πίνακα με στοιχεία τύπου `struct bio_vec`.

Η δομή αυτή ορίζεται ως εξής:

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

Τα μέλη που την αποτελούν έχουν της εξής σημασία:

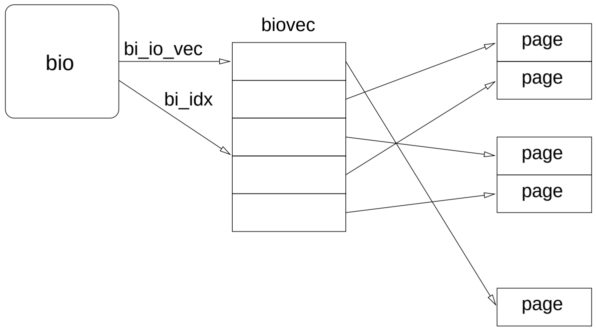
`bv_page`:	Δείκτης στην εν λόγω σελίδα.
`bv_len`:	Μέγεθος του χώρου που θα χρησιμοποιηθεί στη σελίδα.
`bv_offset`:	Σημείο έναρξης του χώρου που θα χρησιμοποιηθεί στη σελίδα.

Εκτός από τον ίδιο τον πίνακα η δομή `struct bio` περιέχει ορισμένα πεδία για την διαχείριση του πίνακα αυτού.

`unsigned short bi_cnt`:	Αριθμός στοιχείων πίνακα.
`unsigned short bi_idx`:	Τρέχουσα θέση στον πίνακα.

Το πεδίο `bi_idx` χρησιμοποιείται για να αποθηκευτεί η τρέχουσα σελίδα προς επεξεργασία σε περίπτωση που ένα `bio` δεν μπορεί να επεξεργαστεί πλήρως από τον οδηγό σε ένα μόνο πέρασμα.

Μερικά ακόμα πεδία της δομής `bio` παρουσιάζονται παρακάτω:



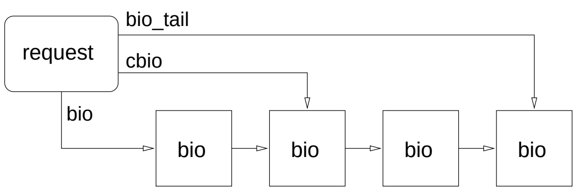
Σχήμα 2.2: Μονάδα Ε/Ε block

Τύπος	Όνομα	Χρήση
`sector_t`	`bi_sector`	Ο τομέας (βλ. παράγραφο 1.3.1) της συσκευής στον οποίο αντιστοιχεί η μονάδα `bio`.
`unsigned int`	`bi_size`	το μέγεθος των δεδομένων που θα αποθηκευτούν ή θα ανακτηθούν, μέσω του `bio` αυτού.
`struct bio *`	`bi_next`	Πεδίο για την κατασκευή συνδεδεμένης λίστας.
`unsigned long`	`bi_rw`	Προτεραιότητα και τύπος της αίτησης (εγγραφή ή ανάγνωση).

2.2.3 Block αιτήσεις Ε/Ε

Τις περισσότερες φορές οι οδηγοί συσκευών block δεν επεξεργάζονται αυτόνομες μονάδες `bio`, αλλά ένα σύνολο από αυτές, αφού όπως προαναφέρθηκε το Linux χρησιμοποιεί χρονοδρομολόγηση Ε/Ε ώστε να ομαδοποιεί τις αιτήσεις κατάλληλα. Οι μονάδες `bio` συλλέγονται σε δομές `struct request`, οι οποίες και μεταφέρονται στον οδηγό της συσκευής.

Κάθε αντικείμενο `request` περιέχει ένα σύνολο από αντικείμενα τύπου `bio`, τα οποία συνδέονται μέσω μια λίστας. Η λίστα αυτή υλοποιείται με την βοήθεια των πεδίων `struct bio *bio` και `struct bio *biotail` της δομής `request` και των πεδίων `bi_next` των δομών `bio`. Επιπρόσθετα, ορίζεται και ένα πεδίο `cbio`, το οποίο αντιστοιχεί στην τρέχουσα μονάδα `bio`.



Σχήμα 2.3: Αίτηση Ε/Ε block

Η δομή `request` περιέχει και κάποια ακόμα πεδία, που έχουν βοηθητικό ρόλο, και μπορούν να χρησιμοποιηθούν είτε από το στρώμα συσκευών block, είτε από τον χρονοδρομολογητή Ε/Ε, είτε ακόμα και από τον οδηγό της συσκευής.

2.2.4 Ουρές αιτήσεων

Η δομή δεδομένων που είναι υπεύθυνη για την συλλογή και αποθήκευση των αιτήσεων είναι η `struct request_queue` ή ισοδύναμα η `request_queue_t`. Η δομή αυτή περιέχει όλες τις πληροφορίες και συναρτήσεις που είναι απαραίτητες για κάθε πτυχή της υλοποίησης του στρώματος block στο Linux. Ο σχεδιασμός είναι τέτοιος, ώστε να μπορεί να προσαρμοστεί σε κάθε πιθανή ανάγκη μιας συσκευής block, αλλά και για την υποστήριξη πολλαπλών αλγορίθμων χρονοδρομολόγησης. Η δομή αυτή είναι ένα από τα πολλά παραδείγματα χρήσης αντικειμενοστραφών (object oriented) μεθόδων στον πυρήνα του Linux.

Ο αλγόριθμος που χρησιμοποιείται για την χρονοδρομολόγηση Ε/Ε, ονομάζεται και αλγόριθμος ανελκυστήρα (elevator algorithm). Για την υποστήριξη πολλαπλών αλγόριθμων ορίστηκε μια διεπαφή για την συνεργασία με το στρώμα block του Linux.

Κάθε τέτοιος αλγόριθμος, θα πρέπει να υλοποιήσει ένα σύνολο από συναρτήσεις με καθορισμένο σκοπό. Οι συναρτήσεις αυτές μαζί με κάποια άλλα βοηθητικά στοιχεία για την υλοποίηση του αλγορίθμου περιέχονται στη δομή `elevator_t`. Η δομή αυτή χρησιμοποιείται από την ουρά ώστε να εφαρμόσει με διάφανο τρόπο την πολιτική του κάθε αλγορίθμου. Σε κάθε ουρά υπάρχει ένα πεδίο `elevator` τύπου `elevator_t` για αυτόν τον σκοπό.

2.3 Διεπαφή οδηγού συσκευής block

Κατά την δημιουργία ενός οδηγού συσκευής block, χρησιμοποιείται μια συγκεκριμένη διεπαφή για την επικοινωνία με το στρώμα block του linux. Σε αυτή τη παράγραφο, θα σκιαγραφηθούν μερικές βασικές συναρτήσεις της διεπαφής αυτής.

2.3.1 Διεπαφή gendisk

Οι περισσότερες λειτουργίες για μια συσκευή block υλοποιούνται από το στρώμα `gendisk`, το οποίο προσφέρει ορισμένες συναρτήσεις για την αλληλεπίδραση με τους οδηγούς των συσκευών block. Για κάθε συσκευή ο οδηγός θα πρέπει να χρησιμοποιήσει μια μεταβλητή τύπου `struct gendisk`.

Για την δέσμευση των πόρων που χρειάζονται για μια τέτοια μεταβλητή χρησιμοποιείται η συνάρτηση `alloc_disk()`. Για την απελευθέρωση των πόρων αυτών θα πρέπει να κληθεί η συνάρτηση `put_disk()`.

Μερικά βασικά πεδία της δομής `struct gendisk` παρουσιάζονται σύντομα στον πίνακα 2.1.

Μετά την αρχικοποίηση ενός `gendisk` η συσκευή που αναπαριστά μπορεί να εισαχθεί στο σύστημα με την συνάρτηση `add_disk()`. Η αντίθετη διαδικασία πραγματοποιείται με τη συνάρτηση `del_gendisk()`.

2.3.2 Ουρές αιτήσεων

Κάθε οδηγός συσκευής block μπορεί να χρησιμοποιείται για περισσότερες από μια συσκευές. Για κάθε συσκευή ο οδηγός, στη συντριπτική πλειοψηφία των περιπτώσεων, δημιουργεί μια ουρά, η οποία απεικονίζεται μέσω του τύπου `struct request_queue`. Η δέσμευση μνήμης για τη δομή αυτή αλλά και η αρχικοποίηση της πραγματοποιείται μέσω της συνάρτησης `blk_init_queue()` η οποία ορίζεται ως εξής:

```
request_queue_t *blk_init_queue(request_fn_proc *request_fn, spinlock_t *lock);
```

Η συνάρτηση αυτή επιστρέφει την αρχικοποιημένη ουρά ή `NULL` σε περίπτωση κάποιου

Πεδίο	Σημασία
`int major`	Ο αριθμός major της συσκευής.
`struct request_queue *queue`	Ουρά αιτήσεων που σχετίζεται με τη συσκευή.
`struct block_device_operations *fops`	Σύνολο συναρτήσεων σχετικών με την συσκευή. Η δομή αυτή περιέχει έναν σύνολο συναρτήσεων, τις οποίες μπορεί να ορίσει ο οδηγός, παρέχοντας λειτουργικότητα που σχετίζεται με την συγκεκριμένη συσκευή. Παράδειγμα τέτοιας συνάρτησης είναι η συνάρτηση `ioctl()` μέσω της οποίας μπορούν να υλοποιηθούν εξειδικευμένες εντολές ελέγχου για κάθε συσκευή.
`void *private_data`	Πεδίο το οποίο παρέχεται στον οδηγό για δική του χρήση.
`char disk_name[32]`	Ένα όνομα για την συσκευή, το οποίο χρησιμοποιείται σε διάφορα σημεία του πυρήνα.
`sector_t capacity`	Η χωρητικότητα της συσκευής. Για τον ορισμό της ο οδηγός θα πρέπει να χρησιμοποιήσει την μακροεντολή `set_capacity()`.

Πίνακας 2.1: Βασικά πεδία της δομής `struct gendisk`

σφάλματος. Η συνάρτηση `request_fn` είναι η βασική συνάρτηση που χρειάζεται να υλοποιήσει ο οδηγός και ως σκοπό έχει την επεξεργασία των αιτήσεων. Η μεταβλητή `lock` είναι μια μεταβλητή συγχρονισμού, η οποία, προστατεύει την ουρά από ταυτόχρονη πρόσβαση.

Όταν ο οδηγός δεν χρειάζεται πλέον την ουρά μπορεί να την καταστρέψει χρησιμοποιώντας την συνάρτηση:

```
void blk_cleanup_queue(request_queue_t * q)
```

Ο τύπος της συνάρτησης `request_fn` ορίζεται ως εξής:

```
typedef void (request_fn_proc) (request_queue_t *q);
```

Η συνάρτηση αυτή θα κληθεί όταν το στρώμα block του Linux αποφανθεί ότι θα πρέπει να προωθηθούν ορισμένες αιτήσεις από την ουρά στην συσκευή. Η κλήση της συνάρτησης αυτής θα γίνει, αφού πρώτα η ουρά έχει κλειδωθεί μέσω της μεταβλητής `lock`.

2.3.3 Επεξεργασία αιτήσεων

Όταν κληθεί η συνάρτηση `request_fn`, για να εξάγει από την ουρά την επόμενη αίτηση θα πρέπει να καλέσει τη συνάρτηση:

```
struct request *elv_next_request(request_queue_t *q)
```

Η συνάρτηση αυτή θα επιστρέψει την επόμενη αίτηση προς επεξεργασία ή `NULL` αν δεν υπάρχει καμία τέτοια αίτηση.

Για την επεξεργασία της αίτησης ο οδηγός θα πρέπει να εξάγει διαδοχικά τις μονάδες `bio` που απαρτίζουν την αίτηση. Για την διαδικασία αυτή έχει οριστεί μια μακροεντολή ώστε να είναι δυνατή η αλλαγή του τρόπου αποθήκευσης των μονάδων σε μια αίτηση χωρίς να χρειαστεί να αλλάξουν όλοι οι χαμηλού επιπέδου οδηγοί.

Η μακροεντολή αυτή ορίζεται ως εξής:

```
rq_for_each_bio(struct bio *bio, struct request *rq)
```

και ένα παράδειγμα χρήσης της είναι:

```
struct bio *bio;

rq_for_each_bio(bio, req) {
        do_something_with_bio(bio);
        ...
}
```

Καθώς προχωράει η επεξεργασία της αίτησης είναι θεμιτό να ανανεώνεται η δομή της αίτησης, σχετικά με την κατάστασή της. Για αυτό το λόγο χρησιμοποιούνται οι συναρτήσεις:

```
int end_that_request_first(struct request *req, int uptodate, int nr_sectors)
int end_that_request_chunk(struct request *req, int uptodate, int nr_bytes)
```

Όπου:

`req`	Η αίτηση προς ανανέωση.
`uptodate`	Μεταβλητή για το αποτέλεσμα επεξεργασίας της αίτησης: (1: για επιτυχή ολοκλήρωση, ≤ 0: για ανεπιτυχή ολοκλήρωση).
`nr_sectors`	Αριθμός τομέων που επεξεργάστηκαν
`nr_bytes`	Αριθμός bytes που επεξεργάστηκαν.

Οι συναρτήσεις αυτές επιστρέφουν `0` σε περίπτωση που η επεξεργασία της αίτησης αυτής έχει ολοκληρωθεί και `1` σε περίπτωση που δεν έχει ολοκληρωθεί.

Όταν ολοκληρωθεί η επεξεργασία της αίτησης, ο οδηγός θα πρέπει να εξάγει την αίτηση από την ουρά και να την τερματίσει. Οι συναρτήσεις που χρησιμοποιούνται είναι οι:

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

Η πρώτη χρησιμοποιείται για την εξαγωγή της αίτησης από την ουρά, ενώ η δεύτερη για την ενημέρωση του στρώματος block για την περάτωση της επεξεργασίας της αίτησης, από τον οδηγό της συσκευής.

2.3.4 Επεξεργασία μονάδων bio

Οι μονάδες Ε/Ε bio συνοδεύονται και αυτές από ορισμένες συναρτήσεις, οι οποίες είτε χρησιμοποιούνται από το στρώμα block του Linux είτε από τον ίδιο τον οδηγό της συσκευής.

Οι οδηγοί πολλές φορές είναι απαραίτητο να επεξεργαστούν όλες τις σελίδες που περιέχονται σε μία μονάδα Ε/Ε `bio`. Για την διευκόλυνση τους έχει οριστεί η μακροεντολή `bio_for_each_segment()`, η οποία χρησιμοποιεί το πεδίο `bi_cnt` της μονάδας `bio` (βλ. και παράγραφο 2.2.2). Ένα παράδειγμα χρήσης της μακροεντολής αυτής είναι:

```
int i;
```

```
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, i){
        do_something_with_bvec(bvec);
}
```

Μια άλλη βασική συνάρτηση για τις μονάδες bio είναι η

```
void bio_endio(struct bio *bio, unsigned int bytes_done, int error)
```

Η συνάρτηση αυτή αναβαθμίζει τα κατάλληλα πεδία της μονάδας `bio` μετά από την μερική ή ολική επεξεργασία της. Σε περίπτωση που η επεξεργασία ολοκληρωθεί και δεν υπάρχουν άλλα δεδομένα η συνάρτηση αυτή φροντίζει για τον τερματισμό της μονάδας αυτής. Η `bio_endio()`, στις περισσότερες περιπτώσεις δεν χρησιμοποιείται απευθείας από τον οδηγό, αλλά καλείται από τις συναρτήσεις `end_that_request_first()` ή `end_that_request_chunk()`.

2.3.5 Παραμετροποίηση της ουράς αιτήσεων

Η διεπαφή του Linux για τους οδηγούς συσκευών προσφέρει πολλές δυνατότητες για ρύθμιση της λειτουργίας του στρώματος block. Οι ρυθμίσεις αυτές είναι ξεχωριστές για κάθε συσκευή και γίνονται στο επίπεδο της ουράς αιτήσεων. Παρακάτω παρουσιάζονται ορισμένα παραδείγματα τέτοιων ρυθμίσεων. Η παρουσίαση αυτή σε καμία περίπτωση δεν είναι πλήρης. Σκοπός της είναι η κατανόηση του τρόπου με τον οποίο είναι δυνατόν να ρυθμιστεί η ουρά αιτήσεων, ώστε να ανταποκρίνεται στις ανάγκες κάθε συσκευής.

Η κάθε συσκευή τις περισσότερες φορές μπορεί να επεξεργαστεί έναν περιορισμένο αριθμό από τομείς κάθε φορά. Συνεπώς, για να μην χρειαστεί να διασπαστούν οι αιτήσεις που παράγει το στρώμα block του Linux, ώστε να τις επεξεργαστεί η συσκευή, ο κάθε οδηγός μπορεί να ορίσει τον μέγιστο αριθμό τομέων που μπορεί να περιέχει μια αίτηση. Η λειτουργία αυτή γίνεται μέσω της συνάρτησης:

```
void blk_queue_max_sectors(request_queue_t *q, unsigned short max)
```

Στις περισσότερες περιπτώσεις οι συσκευές που χρησιμοποιούνται έχουν μέγεθος τομέα 512 bytes. Ωστόσο υπάρχουν ορισμένες συσκευές στις οποίες το μέγεθος του τομέα έχει άλλη τιμή. Η ρύθμιση της τιμής αυτής μπορεί να γίνει μέσω της συνάρτησης:

```
void blk_queue_hardsect_size(request_queue_t *q, unsigned short size)
```

Πολλές φορές είναι απαραίτητο ο οδηγός να κάνει ορισμένες λειτουργίες, ώστε να προετοιμάσει την κάθε αίτηση πριν την επεξεργαστεί μέσω της συνάρτησης `request_fn`. Η παρακάτω συνάρτηση, του επιτρέπει να ορίσει μια συνάρτηση για την προεπεξεργασία μιας αίτησης:

```
void blk_queue_prep_rq(request_queue_t *q, prep_rq_fn *pfn)
```

Ο τύπος `prep_rq_fn` ορίζεται ως:

```
typedef int (prep_rq_fn) (request_queue_t *, struct request *)
```

Η συνάρτηση `pfn` θα καλείται κάθε φορά πριν από την κλήση της συνάρτησης request_fn για την προετοιμασία της αίτησης. Ανάλογα με το τι επιστρέφει η συνάρτηση αυτή η αίτηση, είτε θα προωθηθεί για επεξεργασία, είτε θα αναβληθεί ή θα τερματιστεί εξαιτίας κάποιου λάθους.

Ένα παράδειγμα χρήσης της συνάρτησης αυτής υπάρχει στις συσκευές SCSI, όπου δημιουργείται από πριν η κατάλληλη εντολή που θα μεταφερθεί στην συσκευή. Σε περίπτωση που η δημιουργία της εντολής αποτύχει, αφού απαιτεί δέσμευση μνήμης, η αίτηση αναβάλλεται για αργότερα.

2.3.6 Υλοποίηση συσκευής χωρίς χρονοδρομολόγηση Ε/Ε

Αν και η πλειονότητα των συσκευών αποθήκευσης λειτουργούν αποδοτικότερα με τη χρήση χρονοδρομολόγησης Ε/Ε υπάρχουν και ορισμένες οι οποίες δεν την χρειάζονται. Τέτοιες συσκευές, για παράδειγμα, είναι αυτές που προσομοιώνουν συσκευές block στη κεντρική μνήμη (συσκευές ramdisk).

Κατά την υλοποίηση ενός οδηγού για μια τέτοια συσκευή θα πρέπει να οριστεί μια συνάρτηση, η οποία θα καλείτε κάθε φορά που μια μονάδα `bio` παραδίδεται στο στρώμα block για επεξεργασία. Η συνάρτηση αυτή είναι τύπου:

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

Αρχικά, θα πρέπει να δεσμευθεί μνήμη για μια ουρά αιτήσεων. Κατόπιν η προεπιλεγμένη συνάρτηση, η οποία αναλαμβάνει την επεξεργασία κάθε νέας μονάδας `bio` θα αντικατασταθεί από αυτή που ορίζει ο οδηγός. Οι συναρτήσεις που χρησιμοποιούνται για τις παραπάνω λειτουργίες είναι:

```
request_queue_t *blk_alloc_queue(int gfp_mask)
void blk_queue_make_request(request_queue_t * q, make_request_fn * mfn)
```

Η μεταβλητή `gfp` στην `blk_alloc_queue()` καθορίζει τον τρόπο με τον οποίο θα δεσμευθεί μνήμη από το ΛΣ.

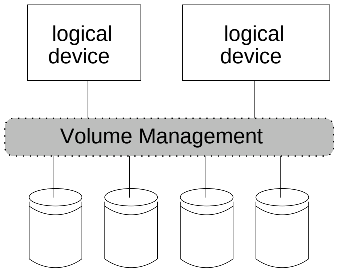
Όταν ο οδηγός ορίζει την δικιά του `make_request_fn` συνάρτηση είναι υπεύθυνος και για τον τερματισμό των αιτήσεων `bio` που δέχεται μέσω της συνάρτησης `bio_endio()`.Τέλος όταν ο οδηγός τερματίσει την λειτουργία του θα πρέπει να καλέσει την `blk_cleanup_queue()`, ώστε να αποδεσμευτούν οι πόροι που σχετίζονται με την ουρά.

2.4 Συστήματα αποθήκευσης πολλαπλών συσκευών

Το Linux προσφέρει υποστήριξη για συστήματα αποθήκευσης πολλαπλών συσκευών block. Ένα τέτοιο σύστημα για παράδειγμα είναι οι αρχιτεκτονικές RAID, που μελετήθηκαν στη παράγραφο 1.4. Υπάρχουν δύο διαφορετικά κομμάτια του πυρήνα που υποστηρίζουν τέτοιου είδους συστήματα

Το πιο παλιό ονομάζεται MD (Multiple Devices) και προσφέρει την δυνατότητα δημιουργίας RAID διατάξεων σε επίπεδο λογισμικού. Υποστηρίζει τις διατάξεις RAID 0, RAID 1, RAID 4 και RAID 5. Αν και το MD θεωρείται σταθερό και δοκιμασμένο περιορίζεται κυρίως στη δημιουργία RAID συστημάτων και δεν μπορεί να αντεπεξέλθει στις ανάγκες για γενικότερη αντιμετώπιση όλων των συστημάτων πολλαπλών συσκευών.

Την ανάγκη για ένα γενικό πλαίσιο δημιουργίας και διαχείρισης συστημάτων πολλαπλών συσκευών καλύπτει στο Linux το DM (Device Mapper). Το DM παρέχει τη δυνατότητα δημιουργίας τέτοιων συστημάτων, συνδυάζοντας με πολλούς διαφορετικούς τρόπους αποθηκευτικές συσκευές ή τμήματα αυτών. Αυτού του είδους η διαχείριση ονομάζεται και Volume Management. Ωστόσο, το DM δεν υποστηρίζει πολύπλοκες διατάξεις RAID, όπως το RAID 5 και συνεπώς



Σχήμα 2.4: Συστήματα αποθήκευσης πολλαπλών συσκευών (Volume Management)

κρίνεται απαραίτητη η συνύπαρξή του με το παλαιότερο MD.

Αρχικά, υπήρχαν δύο υποψήφια συστήματα για την υποστήριξη Volume Management στον πυρήνα του Linux, το DM και το EVMS. Όταν αποφασίστηκε από την κοινότητα να υιοθετηθεί το πρώτο οι προγραμματιστές του EVMS στήριξαν την απόφαση αυτή ανακοινώνοντας ότι θα μετατρέψουν τα προγράμματα χώρου χρήστη που είχαν αναπτύξει να υποστηρίζουν μεταξύ άλλων το DM, προσφέροντας ένα ολοκληρωμένο και πλήρες εργαλείο για Volume Management. Η απόφαση αυτή επιδοκιμάστηκε σε μεγάλο βαθμό από την κοινότητα του Linux.

2.5 Network Block Device

Στον πυρήνα του Linux συμπεριλαμβάνεται μια δικτυακή συσκευή block. Η συσκευή αυτή, χρησιμοποιώντας ένα μοντέλο εξυπηρετητή - πελάτη, επιτρέπει σε ένα σύστημα να χρησιμοποιήσει μια αποθηκευτική συσκευή ενός άλλου απομακρυσμένου συστήματος. Το πρώτο σύστημα αποτελεί τον πελάτη και το δεύτερο τον εξυπηρετητή. Η συσκευή αυτή ονομάζεται Network Block Device (nbd). Δεδομένου ότι η συσκευή nbd σχετίζεται άμεσα με το αντικείμενο της εργασίας αυτής θα παρουσιαστεί συνοπτικά η λειτουργία της.

Η συσκευή nbd, χρησιμοποιεί TCP/IP sockets για την μεταφορά δεδομένων, μεταξύ των δύο συστημάτων. Ο εξυπηρετητής, είναι ένα πρόγραμμα, το οποίο δέχεται αιτήσεις Ε/Ε και τις ικανοποιεί χρησιμοποιώντας ως χώρο αποθήκευσης ένα η περισσότερα αρχεία, από το τοπικό του σύστημα αρχείων. Τα αρχεία αυτά, μπορεί να είναι κανονικά αρχεία ή ειδικά αρχεία συσκευών block.

Ο πελάτης, αποτελείται από μια εφαρμογή χώρου χρήστη, και έναν οδηγό συσκευής block στον χώρο πυρήνα. Η εφαρμογή χώρου χρήστη αρχικά δημιουργεί ένα socket, επικοινωνεί με τον εξυπηρετητή και κατόπιν περνάει το socket που δημιούργησε στον χώρο πυρήνα.

Ο οδηγός που υλοποιείται στον χώρο πυρήνα μπορεί να χρησιμοποιήσει το socket αυτό για να στείλει αιτήσεις στον εξυπηρετητή μέσω ενός συγκεκριμένου πρωτοκόλλου. Οι αιτήσεις εισάγονται σε μία λίστα, ώστε να μπορούν να ανακτηθούν όταν έρθει η απάντηση από τον εξυπηρετητή. Σε περίπτωση αίτησης εγγραφής, εκτός από την αίτηση αποστέλλονται και τα δεδομένα προς εγγραφή.

Παράλληλα ο οδηγός περιμένει απαντήσεις από τον εξυπηρετητή στις αιτήσεις που έχει στείλει. Μόλις λάβει μια απάντηση, εξάγει την αίτηση στην οποία αντιστοιχεί η απάντηση από την λίστα. Αν η απάντηση αφορά ανάγνωση, τοποθετεί τα δεδομένα της ανάγνωσης στον κατάλληλο χώρο. Τέλος τερματίζει την αίτηση κατάλληλα.

Κεφάλαιο 3

Δίκτυα Διασύνδεσης Υπολογιστικών Συστημάτων

Στο κεφαλαίο αυτό θα παρουσιαστούν τα δίκτυα διασύνδεσης που χρησιμοποιήθηκαν στην εργασία αυτή. Η παρουσίαση αυτή αφορά όχι μόνο τα φυσικά χαρακτηριστικά τους, αλλά και τη διεπαφή που προσφέρουν για τον προγραμματισμό τους. Τα δίκτυα που θα αναλυθούν συνθέτουν μια αρκετά ολοκληρωμένη εικόνα των σύγχρονων τεχνολογιών διασύνδεσης υπολογιστικών συστημάτων.

3.1 Ethernet και TCP/IP

Τα πρωτόκολλα ethernet και TCP/IP είναι αρκετά διαδεδομένα και πολύπλοκα και δεν κρίθηκε απαραίτητη μια αναλυτική παρουσίασή τους. Θα περιοριστούμε σε μια απλή αναφορά ορισμένων χαρακτηριστικών τους που σχετίζονται με την εργασία αυτή. Μια ιδιαίτερα αναλυτική παρουσίαση των δικτύων TCP/IP και του προγραμματισμού τους υπάρχει στα [Ste93] και [Ste97].

3.1.1 Ethernet

Η οικογένεια πρωτοκόλλων ethernet αφορά τοπικά δίκτυα (LAN - Local Area Network) και έχει αναπτυχθεί από το IEEE (Institute of Electrical and Electronics Engineers). Το Ethernet στη γενική του μορφή χρησιμοποιεί ένα κοινό διαμοιραζόμενο μέσο για την μετάδοση δεδομένων, στο οποίο με κάποιο αλγόριθμο οι διάφοροι κόμβοι μπορούν να στείλουν και να λάβουν πακέτα δεδομένων. Το διαμοιραζόμενο μέσο μπορεί να είναι πολλών μορφών αν και στις περισσότερες εφαρμογές είναι διάδρομος.

Για μεγάλο αριθμό κόμβων η χρήση του κοινού μέσου δεν είναι αποδοτική. Για την αντιμετώπιση του προβλήματος αυτού χρησιμοποιούνται, συνήθως, μεταγωγείς (switches) σε τοπολογία αστέρα για τη διασύνδεση των διάφορων κόμβων στα δίκτυα αυτά. Οι μεταγωγείς, στους οποίους καταλήγουν όλες οι συνδέσεις, επιτρέπουν την ταυτόχρονη επικοινωνία πολλών κόμβων υποσκελίζοντας τους περιορισμούς του διαμοιρασμένου μέσου.

το Fast Ethernet είναι το πιο συχνά χρησιμοποιούμενο δίκτυο διασύνδεσης, κυρίως εξαιτίας του χαμηλού κόστους, αλλά και της απλότητας εγκατάστασής του. Το Fast Ethernet υποστηρίζει ταχύτητες 100 Mbit/sec και χρησιμοποιεί αντεστραμμένα ζεύγη χαλκού (twisted pair) ως καλωδίωση.

Για την υποστήριξη ταχυτήτων 1 Gbit/sec αναπτύχθηκαν διάφορα πρωτόκολλα Ethernet, τα οποία αναφέρονται στο σύνολό τους ως Gigabit Ethernet. Το Gigabit Ethernet υποστηρίζει αντεστραμμένα ζεύγη χαλκού, αλλά και οπτικές ίνες (optical fiber) ως μέσο μετάδοσης. Η όλο και μεγαλύτερη ανάγκη για υψηλότερες ταχύτητες μετάδοσης δεδομένων έχει ως αποτέλεσμα την αρκετά μεγάλη διάδοση του Gigabit Ethernet, κυρίως σε δίκτυα με μεγάλο αριθμό κόμβων.

Πρόσφατα αναπτύχθηκε και μια νέα οικογένεια προτύπων Ethernet, τα οποία μεταφέρουν δεδομένα σε ταχύτητα 10 Gbit/sec. Τα πρότυπα αυτά αναφέρονται ως 10 Gigabit Ethernet και αποτελούν το πιο γρήγορο πρωτόκολλο Ethernet.

3.1.2 TCP/IP

Στην συντριπτική πλειοψηφία των περιπτώσεων τα δίκτυα Ethernet χρησιμοποιούν τα πρωτόκολλα TCP/IP. Η χρήση είναι τόσο συχνή, που σε πολλές περιπτώσεις, όταν γίνεται αναφορά σε δίκτυο TCP/IP υπονοείται η χρήση ethernet και το αντίστροφο.

Το πρωτόκολλο στρώματος δικτύου IP είναι υπεύθυνο για την μεταφορά πακέτων στο δίκτυο από ένα άκρο σε ένα άλλο. Μπορεί να χρησιμοποιηθεί είτε σε μεγάλης κλίμακας δίκτυα, όπως το internet είτε σε μικρά τοπικά δίκτυα. Κάθε κόμβος ενός δικτύου IP, αναγνωρίζεται από μια μοναδική διεύθυνση, η οποία ονομάζεται διεύθυνση IP.

Το TCP επιτελεί το έργο της ασφαλής μεταφοράς δεδομένων, από το ένα άκρο στο άλλο. Για να μπορεί ο κάθε κόμβος να δημιουργήσει διαφορετικές συνδέσεις και να τις ξεχωρίσει το πρωτόκολλο TCP χρησιμοποιεί αριθμούς που ονομάζονται ports.

3.1.3 Sockets

Ο προγραμματισμός σε δίκτυα TCP/IP γίνεται με τη βοήθεια του socket API. Τα sockets είναι μία γενικού τύπου διεπαφή για επικοινωνία υπολογιστικών συστημάτων που υποστηρίζει διαφορετικές οικογένειες πρωτοκόλλων. Μια από αυτές τις οικογένειες αντιστοιχεί σε δίκτυα TCP/IP και είναι και η πιο συχνά χρησιμοποιούμενη.

Το socket API λειτουργεί παρέχοντας στον προγραμματιστή έναν περιγραφητή αρχείου (file descriptor), ο οποίος ονομάζεται socket. Όταν η σύνδεση μεταξύ των δύο κόμβων έχει αρχικοποιηθεί, ο κάθε κόμβος μπορεί να στείλει δεδομένα στον άλλο γράφοντας στον περιγραφητή και να λάβει δεδομένα διαβάζοντας από αυτόν. Η εγγραφή και η ανάγνωση από περιγραφητές αρχείων συνήθως γίνεται μέσω των κλήσεων συστήματος (system calls) `read()` και `write()`. Βέβαια, υπάρχουν και πιο εξελιγμένες δυνατότητες για τη λήψη και αποστολή δεδομένων χρησιμοποιώντας τις κλήσεις συστήματος `recvmsg()` και `sendmsg()`, που έχουν δημιουργηθεί αποκλειστικά για sockets.

Κατά την αρχικοποίηση μιας σύνδεσης το ένα άκρο δεσμεύει μια προκαθορισμένη διεύθυνση και το άλλο άκρο συνδέεται σε αυτή. Η διεύθυνση ενός άκρου σύνδεσης αποτελείται από την IP διεύθυνση του κόμβου του δικτύου και ενός αριθμού port.

Οι βασικές συναρτήσεις για την αρχικοποίηση μιας σύνδεσης παρουσιάζονται στον πίνακα 3.1.

Συνάρτηση	Σημασία
`socket()`	Δημιουργία ενός περιγραφητή αρχείου τύπου socket.
`bind()`	Δέσμευση μιας διεύθυνσης σε έναν κόμβο.
`listen()`	Αναμονή σύνδεσης από απομακρυσμένο κόμβο σε μία δεσμευμένη διεύθυνση.
`connect()`	Αίτηση σύνδεσης σε απομακρυσμένο άκρο.
`accept()`	Αποδοχή αίτησης σύνδεσης και ολοκλήρωση της.

Πίνακας 3.1: βασικές συναρτήσεις του socket API

3.2 Δίκτυα διασύνδεσης υψηλών επιδόσεων

Τα πρωτόκολλα TCP/IP παρουσιάζουν προβλήματα όταν χρησιμοποιούνται σε τοπικά δίκτυα, των οποίων στόχος είναι η υψηλή επίδοση. Ο βασικός λόγος, για τον οποίο συμβαίνει αυτό είναι η εισαγωγή μεγάλης καθυστέρησης (latency) κατά την επεξεργασία των πακέτων, που ανταλλάσσουν τα δύο άκρα της σύνδεσης.

Ένας από τους λόγους που δημιουργείται η καθυστέρηση αυτή είναι ότι στα δίκτυα TCP/IP απαιτείται η μεσολάβηση του λειτουργικού συστήματος μεταξύ του προσαρμογέα δικτύου και της εφαρμογής. Η μεσολάβηση αυτή, η οποία συνοδεύεται από αντιγραφές δεδομένων σε ενδιάμεσους χώρους αποθήκευσης, μπορεί να αρθεί δίνοντας στην εφαρμογή τη δυνατότητα να προσπελάσει απευθείας τον προσαρμογέα και κατά συνέπεια το δίκτυο. Τα μοντέρνα δίκτυα διασύνδεσης χρησιμοποιούν την τεχνική αυτή σε συνδυασμό με άλλες μειώνοντας σημαντικά το κόστος επικοινωνίας μεταξύ δύο υπολογιστικών συστημάτων.

Ακολουθεί η ανάλυση δύο δικτύων υψηλής επίδοσης.

3.3 Myrinet

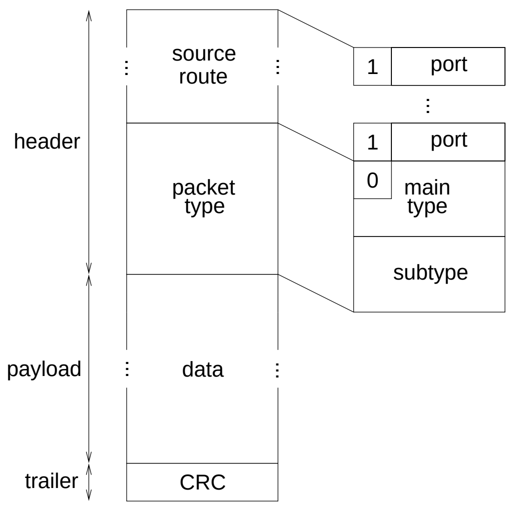
3.3.1 Γενικά

Το Myrinet είναι ένα υψηλής επίδοσης δίκτυο και βασίζεται σε τεχνολογίες που αναπτύχθηκαν για τη διασύνδεση πολλαπλών επεξεργαστών σε αρχιτεκτονικές MMP (Massively Parallel Processors). Προσφέρει συνδέσεις ταχύτητας 1.28 Gbit/sec ή 2 Gbit/sec, δυνατότητα ταυτόχρονης μετάδοσης δεδομένων και στις δύο κατευθύνσεις (full duplex) και μικρούς χρόνους μετάδοσης μηνυμάτων. Το Myrinet βρίσκει εφαρμογή σε συστοιχίες υπολογιστών (computer clusters) και γενικά χρησιμοποιείται συχνά ως SAN (System Area Network). Το Myrinet υποστηρίζει διάφορα μέσα μετάδοσης αν και στις περισσότερες εφαρμογές χρησιμοποιούνται οπτικές ίνες.

Ένα τυπικό δίκτυο Myrinet αποτελείται από υπολογιστικούς κόμβους και μεταγωγείς Myrinet, που φροντίζουν για τη διασύνδεση των κόμβων. Για την επικοινωνία των κόμβων χρησιμοποιούνται πακέτα μεταβλητού μήκους.

3.3.2 Πακέτα Myrinet

Η δομή ενός πακέτου Myrinet παρουσιάζεται στο σχήμα 3.1.



Σχήμα 3.1: Η δομή ενός πακέτου Myrinet

Το πακέτο Myrinet μπορεί να χωριστεί σε 3 κομμάτια:

1. **Header:**
   Η επικεφαλίδα του πακέτου αποτελείται με τη σειρά της από δύο μέλη:

   - **source route :**
     Στην αρχή του κάθε πακέτου υπάρχει ένας μεταβλητός αριθμός από πεδία μήκους ενός byte. Το MSB (Most Significant Bit) κάθε τέτοιου πεδίου είναι 1. Τα πεδία αυτά χρησιμοποιούνται από τους μεταγωγείς και περιέχουν την πόρτα (port) που πρέπει να προωθηθεί το πακέτο.
   - **packet type :**
     Ακολουθεί ένα πεδίο μήκους 4 bytes που καθορίζει τον τύπο του πακέτου. Το MSB του πεδίου αυτού είναι 0 και προορίζεται για χρήση από τους κόμβους του δικτύου. Τα δύο πρώτα bytes του πεδίου αυτού καθορίζουν τον βασικό τύπο του πακέτου ενώ τα επόμενα τον δευτερεύοντα τύπο (subtype). Μερικοί τύποι πακέτων Myrinet υπάρχουν στον πίνακα 3.2.

   Αν και η source route περιοχή ενός πακέτου είναι μεταβλητού μεγέθους είναι εύκολο να βρεθεί που αρχίζει το πεδίο τύπου του πακέτου από τα MSB. Αν όλες οι διαδρομές ήταν προκαθορισμένες δεν θα υπήρχε ανάγκη για μια τέτοια λειτουργία. Κατά την χαρτογράφηση (mapping) του δικτύου, ωστόσο, δεν γνωρίζουμε την τοπολογία του, οπότε η παραπάνω λειτουργικότητα είναι απαραίτητη.

2. **Payload:**
   Σε αυτό το κομμάτι του πακέτου αποθηκεύονται τα δεδομένα. Υπεύθυνο για τα δεδομένα είναι το υψηλότερο στρώμα, το οποίο και τα διαχειρίζεται.

3. **Trailer:**
   Για τον έλεγχο της ακεραιότητας του πακέτου στο τέλος του αποθηκεύεται ένα byte CRC (Cyclic Redundancy Check) όλων των προηγούμενων δεδομένων.

Βασικός Τύπος	Χρήση
`0x0004`	Data Packet
`0x0008`	GM
`0x000F`	GM-1 Mapping
`0x0100`	Active Messages
`0x0700`	VIA (Virtual Interface Architecture)
`0x0800`	Portals

Πίνακας 3.2: Παραδείγματα τύπων πακέτων Myrinet

3.3.3 Σύμβολα ελέγχου

Στα δίκτυα Myrinet εκτός από τα 256 (2^8) κλασικά σύμβολα δεδομένων χρησιμοποιούνται και σύμβολα ελέγχου. Τα βασικότερα από αυτά είναι:

- `GAP`:
  Το σύμβολο αυτό χρησιμοποιείται ως διαχωριστικό μεταξύ των πακέτων. Σε αντίθεση με τα υπόλοιπα σύμβολα ελέγχου το GAP μπορεί να σταλεί μόνο μεταξύ πακέτων.

- `GO` και `STOP`:
  Τα σύμβολα αυτά χρησιμοποιούνται για έλεγχο της ροής δεδομένων μεταξύ δύο άκρων μιας σύνδεσης.

3.3.4 Δρομολόγηση

Η δρομολόγηση στο Myrinet είναι δρομολόγηση πηγής (source routing). Ο κάθε κόμβος γνωρίζει τη τοπολογία του δικτύου και είναι υπεύθυνος να καθορίσει την διαδρομή που θα ακολουθήσει το κάθε πακέτο.

Ο κάθε μεταγωγέας όταν δέχεται ένα πακέτο εκτελεί τα παρακάτω βήματα:

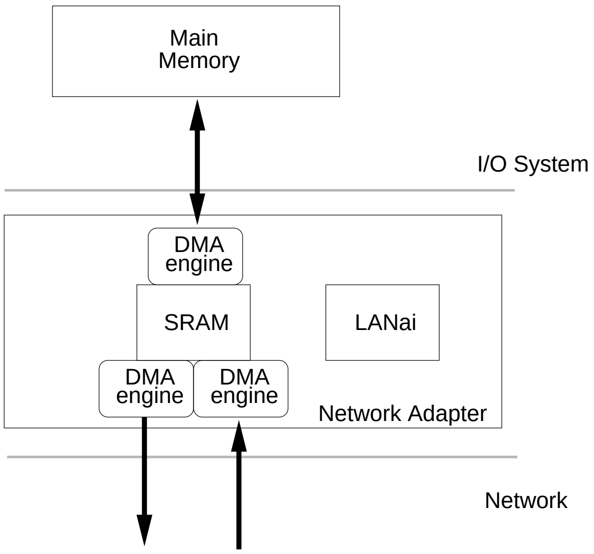
- βρίσκει την πόρτα που θα το προωθήσει από το πρώτο byte του πακέτου.
- Αφαιρεί το byte αυτό και ανανεώνει το CRC πεδίο.

Σε περίπτωση που η πόρτα εξόδου του πακέτου είναι κατειλημμένη το πακέτο περιμένει έως ότου αυτή απελευθερωθεί. Θα πρέπει να σημειωθεί πως η ακεραιότητα του πακέτου δεν ελέγχεται στους μεταγωγείς, αλλά μόνο στον τελικό προορισμό του.

Η τεχνική δρομολόγησης αυτή ονομάζεται cut-through routing και μειώνει σημαντικά την καθυστέρηση μεταγωγής του πακέτου.

3.3.5 Προσαρμογέας Δικτύου Myrinet

Ο προσαρμογέας δικτύου (network adapter) συνδέεται στο υπολογιστικό σύστημα ως μια τυπική συσκευή Ε/Ε και προσφέρει πρόσβαση στο δίκτυο.



Σχήμα 3.2: Διάγραμμα του προσαρμογέα δικτύου Του Myrinet

Τα βασικά στοιχεία του προσαρμογέα είναι όπως φαίνεται και στο σχήμα 3.2:

- Ένας επεξεργαστής LANai τύπου RISC που λειτουργεί στα 33, 66, 133 ή 200 MHz.

- Γρήγορη μνήμη SRAM μεγέθους 1-4 MB.
- 3 μηχανές DMA για τη μεταφορά δεδομένων στην SRAM. Οι δύο χρησιμοποιούνται για την μεταφορά δεδομένων προς και από το δίκτυο και η τρίτη για την μεταφορά δεδομένων προς και από την κεντρική μνήμη του συστήματος.

Ο προγραμματιζόμενος επεξεργαστής LANai προσφέρει δυνατότητες επικοινωνίας στις εφαρμογές χωρίς την παρέμβαση του λειτουργικού συστήματος. Επίσης, ο επεξεργαστής αυτός έχει κάνει ιδιαίτερα δημοφιλές το Myrinet σε ερευνητικούς φορείς αφού μπορεί να προγραμματιστεί και να προσφέρει διάφορες υπηρεσίες στην εκάστοτε εφαρμογή που χρησιμοποιεί το Myrinet.

3.3.6 Υποδομή Λογισμικού

Υπάρχουν δύο βασικά επίπεδα λογισμικού που χρησιμοποιούνται στο Myrinet. Το πρώτο εκτελείται στον επεξεργαστή που βρίσκεται στον προσαρμογέα δικτύου και ονομάζεται MCP (Myrinet Control Program). Το δεύτερο εκτελείται στο κυρίως σύστημα και αποτελείται από μια βιβλιοθήκη και έναν οδηγό που εκτελείται στον χώρο πυρήνα. Η βιβλιοθήκη αυτή προσφέρει μία διεπαφή, την οποία που μπορούν να χρησιμοποιήσουν οι διάφορες εφαρμογές. Το σύστημα αυτό στο σύνολο του αναφέρεται ως GM.

Ο οδηγός που βρίσκεται στο χώρο πυρήνα έχει σε γενικές γραμμές περιορισμένο ρόλο, αφού η εκάστοτε εφαρμογή επικοινωνεί απευθείας με τον προσαρμογέα, ώστε να επιτευχθεί μικρή καθυστέρηση κατά την αποστολή και λήψη μηνυμάτων. Μερικές από τις λειτουργίες που επιτελεί είναι:

- Αρχικοποίηση της συσκευής, μεταφορά του MCP στην κάρτα Myrinet και έναρξη του.
- Αρχικοποίηση των δομών που χρειάζεται μια εφαρμογή για να χρησιμοποιήσει τον προσαρμογέα.
- Αναστολή του τρεξίματος μιας διεργασίας μέχρι να συμβεί κάποιο γεγονός ώστε να μην καταναλώνει επεξεργαστική ισχύ (sleep). Η λειτουργία αυτή είναι απαραίτητη σε συναρτήσεις, οποίες περιμένουν σύγχρονα κάποιο γεγονός (blocking).
- Επιβολή συγκεκριμένων περιορισμών στους χώρους μνήμης που χρησιμοποιεί η διεργασία για να επικοινωνεί με τον προσαρμογέα. Συγκεκριμένα, η μνήμη αυτή θα πρέπει να είναι "αγκιστρωμένη" (pinned down), ώστε οι εικονικές διευθύνσεις των χώρων αυτών να απεικονίζονται στη φυσική μνήμη και όχι σε δευτερεύοντα αποθηκευτικά μέσα (σκληροί δίσκοι).

Αντίστοιχα μερικές από τις λειτουργίες του MCP είναι:

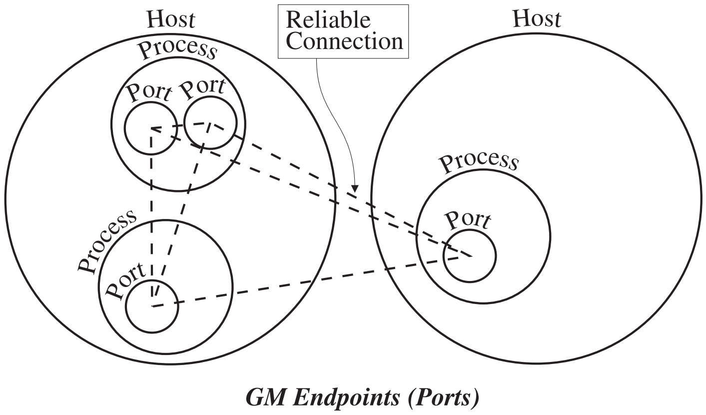
- Χαρτογράφηση του δικτύου.
- Δρομολόγηση των πακέτων Myrinet στο δίκτυο.
- Μεταφορά δεδομένων από το δίκτυο στην εσωτερική μνήμη της κάρτας και το αντίστροφο.
- Μεταφορά δεδομένων από την κυρία μνήμη του συστήματος στην εσωτερική μνήμη της κάρτας και το αντίστροφο.
- Υλοποίηση πρωτοκόλλου που εξασφαλίζει εγγυημένη αποστολή δεδομένων και ανίχνευση σφαλμάτων.

Η βιβλιοθήκη προσφέρει ένα API, για να μπορούν οι εφαρμογές να χρησιμοποιήσουν όλες τις δυνατότητες του Myrinet με έναν διαφανή και ενιαίο τρόπο. Υπάρχουν δύο βασικές εκδόσεις του GM, οι οποίες αναφέρονται ως GM-1 και GM-2. Παρακάτω θα μελετηθεί η διεπαφή του GM-2 (βλ. [Myr])

3.3.7 Διεπαφή του GM

3.3.7.1 Ports

Το port (`gm_port_t`) είναι η βασική δομή του GM. Αναπαριστά ένα εικονικό άκρο μιας σύνδεσης και επιτρέπει στην διεργασία να χρησιμοποιεί το δίκτυο, επικοινωνώντας με την κάρτα Myrinet. το GM παρέχει εγγυημένη μεταφορά δεδομένων χωρίς σύνδεση μεταξύ δύο ports.



Σχήμα 3.3: GM Ports

Η αρχικοποίηση ενός port πραγματοποιείται με τη συνάρτηση:

```
gm_status_t gm_open(gm_port_t **port_p, unsigned unit, unsigned port_id,
                    const char *client_type, enum gm_api_version api_version);
```

Όταν η σύνδεση δεν χρειάζεται πλέον το port μπορεί να αποδεσμευτεί χρησιμοποιώντας τη συνάρτηση:

```
void gm_close(gm_port_t *p)
```

3.3.7.2 Δέσμευση χώρου μνήμης για μηνύματα

Για να επιτευχθεί υψηλός ρυθμός μετάδοσης δεδομένων, αλλά και η μικρή καθυστέρηση του GM πρέπει να τηρούνται ορισμένες προϋποθέσεις, που αφορούν τον χώρο μνήμης, ο οποίος χρησιμοποιείται για τις μεταφορές.

Η βασικότερη από αυτές είναι ότι η μνήμη που χρησιμοποιείται για την αποθήκευση των μηνυμάτων λήψης και αποστολής θα πρέπει να μπορεί να προσπελαστεί με μεθόδους DMA και συνεπώς θα πρέπει να είναι αγκιστρωμένη.

Το GM παρέχει συναρτήσεις για την δέσμευση και απελευθέρωση τέτοιου είδους μνήμης:

```
void *gm_dma_malloc(gm_port_t *p, gm_size_t length);
```

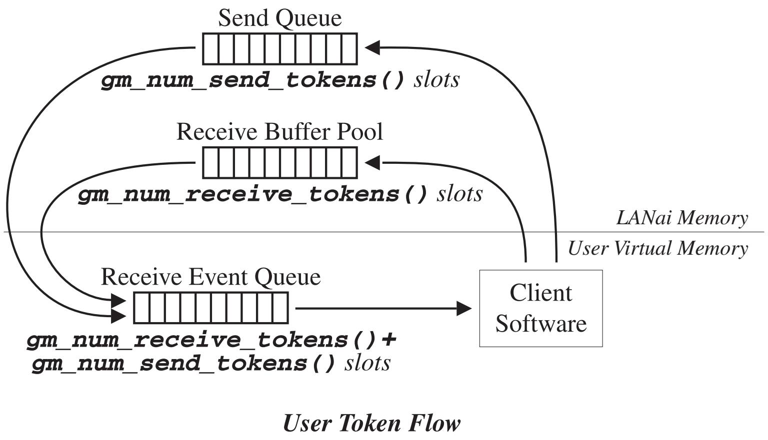
```
void *gm_dma_calloc(gm_port_t *p, gm_size_t count, gm_size_t length);
void gm_dma_free(gm_port_t *p, void *addr);
```

Το GM παράλληλα παρέχει και τη δυνατότητα χρησιμοποίησης μνήμης που έχει ήδη δεσμευθεί με κλασικές μεθόδους (`malloc()` , `calloc()` κλπ). Η λειτουργία αυτή επιτελείται με την βοήθεια των παρακάτω συναρτήσεων:

```
gm_status_t gm_register_memory(gm_port_t *p, void *ptr, gm_size_t length);
gm_status_t gm_deregister_memory(gm_port_t *p, void *ptr, gm_size_t length);
```

3.3.7.3 Έλεγχος ροής μεταξύ του προσαρμογέα και της εφαρμογής

Τα δεδομένα που μπορούν να αποθηκευτούν στη μνήμη της κάρτας είναι περιορισμένα. Απαιτείται, συνεπώς, έλεγχος ροής που θα εξασφαλίσει την ορθή λειτουργία του GM. Αυτόν τον σκοπό εξυπηρετεί η έννοια των tokens.



Σχήμα 3.4: GM Tokens

Η εφαρμογή κατά τη διάρκεια της εκτέλεσης έχει στην κατοχή της έναν αριθμό από tokens αποστολής και από tokens λήψης. Υπάρχουν ορισμένες συναρτήσεις που είτε απαιτούν την κατανάλωση ενός token είτε επιστρέφουν ένα token. Τα tokens εκφράζουν τις ελεύθερες θέσεις που υπάρχουν στις ουρές αποστολής και λήψης στο MCP. Κάθε πρόγραμμα που χρησιμοποιεί το GM θα πρέπει από μόνο του να ελέγχει τον αριθμό των tokens που διαθέτει σε κάθε χρονική στιγμή.

Τα tokens που αρχικά παρέχονται σε μία εφαρμογή για κάθε port που χρησιμοποιεί ορίζονται από τις συναρτήσεις:

```
unsigned int gm_num_send_tokens(gm_port_t *p);
unsigned int gm_num_receive_tokens(gm_port_t *p);
```

Τα tokens καταναλώνονται με την κλήση συγκεκριμένων συναρτήσεων. Σε καμία περίπτωση η εφαρμογή δεν θα πρέπει να καλέσει μια συνάρτηση, η οποία καταναλώνει ένα token χωρίς να το διαθέτει. Αντίστοιχα υπάρχουν γεγονότα, τα οποία σηματοδοτούν την επιστροφή ενός token στην εφαρμογή.

3.3.7.4 Αποστολή Μηνυμάτων

Όπως προαναφέρθηκε η αποστολή μηνυμάτων είναι βασισμένη στα tokens αποστολής. Ένα μήνυμα μπορεί να σταλεί μόνο όταν η εφαρμογή διαθέτει τουλάχιστον ένα τέτοιο token.

Η βασική συνάρτηση που χρησιμοποιείται για αποστολή μηνυμάτων στο GM είναι η:

```
gm_send_with_callback(gm_port_t *port, void *message, unsigned int size,
                      gm_size_t len, unsigned int priority,
                      unsigned int target_node_id,
                      unsigned int target_port_id,
                      gm_send_completion_callback_t callback,
                      void *context);
```

Τα ορίσματα της παρουσιάζονται στον πίνακα 3.3.

Όρισμα	Σημασία
`port`	Το port που θα χρησιμοποιηθεί για την επικοινωνία με την κάρτα Myrinet.
`message`	Τα δεδομένα προς αποστολή.
`size`	Η τάξη του μεγέθους του χώρου μνήμης του παραλήπτη, στον οποίο θα τοποθετηθούν τα δεδομένα.
`len`	Το μέγεθος των δεδομένων προς αποστολή.
`priority`	Η προτεραιότητα του μηνύματος.
`target_node_id`	Ένα αναγνωριστικό του κόμβου στον οποίο θα σταλούν τα δεδομένα.
`target_port_id`	Ένα αναγνωριστικό του απομακρυσμένου port στο οποίο θα σταλούν τα δεδομένα.
`callback`	Συνάρτηση η οποία θα κληθεί για την επιβεβαίωση της αποστολής των δεδομένων.
`context`	Όρισμα με το οποίο θα κληθεί η συνάρτηση `callback`.

Πίνακας 3.3: Ορίσματα της `gm_send_with_callback`

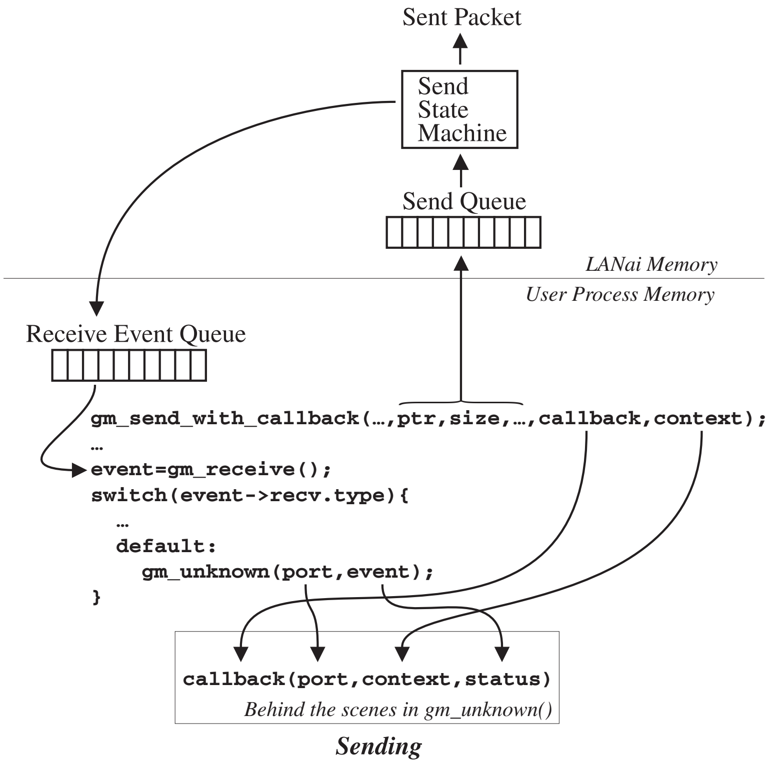
Κατά την κλήση της συνάρτησης αυτής προσδιορίζεται μια συνάρτηση, η οποία είναι τύπου `gm_send_completion_callback_t`. Η συνάρτηση αυτή χρησιμοποιείται για την υλοποίηση ασύγχρονης επικοινωνίας και ονομάζεται συνάρτηση callback. Όταν τα δεδομένα σταλούν ή προκύψει ένα λάθος, το οποίο δεν μπορεί να αντιμετωπιστεί θα κληθεί η συνάρτηση αυτή ώστε να ενημερωθεί η εφαρμογή σχετικά με την κατάσταση της αποστολής.

Τις περισσότερες φορές η συνάρτηση αυτή θα χρειαστεί κάποια δεδομένα σχετικά είτε με την συγκεκριμένη αποστολή την οποία αφορά ή με εσωτερικές δομές της εφαρμογής που χρησιμοποιεί το GM. Για αυτόν το λόγο ορίζεται επίσης και ένας διαφανής δείκτης, ο οποίος θα μεταβιβαστεί στη συνάρτηση callback κατά την εκτέλεση της και μπορεί να περιέχει τέτοια δεδομένα.

Ο τύπος `gm_send_completion_callback_t` ορίζεται ως εξής:

```
typedef void (*gm_send_completion_callback_t) (gm_port_t *p,
                                                void *context,
                                                gm_status_t status);
```

Η μεταβλητή `status` περιέχει την κατάσταση της αποστολής. Σε περίπτωση που η αποστολή ολοκληρώθηκε επιτυχώς η τιμή της είναι `GM_SUCCESS`. Σε αντίθετη περίπτωση η μεταβλητή αυτή προσδιορίζει το λάθος, το οποίο εμπόδισε την αποστολή των δεδομένων. Η συνάρτηση callback εκτελείται στο πλαίσιο της συνάρτησης `gm_unknown()` η οποία θα παρουσιαστεί παρακάτω. Στο σχήμα 3.5 παρουσιάζεται ένα διάγραμμα της λειτουργίας της συνάρτησης `gm_send_with_callback()`.



Σχήμα 3.5: Αποστολή μηνύματος στο GM

Η συνάρτηση αυτή όπως και κάθε άλλη συνάρτηση αποστολής απαιτεί ένα token αποστολής. Με την κλήση της συνάρτησης callback το token επιστρέφεται στην εφαρμογή και ο χώρος, στον οποίο ήταν αποθηκευμένα τα δεδομένα είναι πλέον ελεύθερος για χρήση.

Το GM παρέχει δύο διαφορετικές κλάσεις μηνυμάτων, οι οποίες καθορίζονται από την προτεραιότητα τους. Οι δύο δυνατές τιμές για την προτεραιότητα ενός μηνύματος είναι `GM_LOW_PRIORITY` και `GM_HIGH_PRIORITY`. Τα μηνύματα διαφορετικής προτεραιότητας δεν εμποδίζουν το ένα το άλλο. Η παραπάνω ιδιότητα έχει ως αποτέλεσμα να είναι δυνατόν να σταλεί ένα μήνυμα χαμηλής προτεραιότητας, ενώ υπάρχουν μηνύματα υψηλής προτεραιότητας στην ουρά. Η χρήση διαφορετικών προτεραιοτήτων επιτρέπει την αποστολή μηνυμάτων ελέγχου χωρίς να χρειάζεται να αποσταλούν πρώτα όλα τα μηνύματα που βρίσκονται στην ουρά.

3.3.7.5 Λήψη Μηνυμάτων

Για να μπορεί να λάβει μηνύματα η εφαρμογή θα πρέπει να δεσμεύσει χώρο μνήμης και να τον παραδώσει στο GM, ώστε σε περίπτωση που φθάσουν δεδομένα από το δίκτυο να μεταφερθούν στον χώρο αυτό. Η συνάρτηση που χρησιμοποιείται για αυτόν το σκοπό είναι η:

gm_provide_receive_buffer_with_tag(gm_port_t * port, **void** *ptr, **unsigned** size,

unsigned priority, **unsigned int** tag);

Όρισμα	Σημασία
`port`	Το port που θα χρησιμοποιηθεί για την επικοινωνία με την κάρτα Myrinet.
`ptr`	Εικονική διεύθυνση του χώρου μνήμης που παραχωρείται στο GM.
`size`	Τάξη μεγέθους του χώρου μνήμης.
`priority`	Προτεραιότητα μηνυμάτων για τα οποία προορίζεται ο χώρος μνήμης.
`tag`	Ετικέτα του χώρου μνήμης που προσφέρεται.

Πίνακας 3.4: Ορίσματα της `gm_provide_receive_buffer_with_tag`

Κάθε κλήση της συνάρτησης αυτής καταναλώνει ένα token λήψης. Τα ορίσματά της παρουσιάζονται σύντομα στον πίνακα 3.4.

Το GM χρησιμοποιεί την έννοια της τάξης μεγέθους. Η τάξη μεγέθους ενός μηνύματος είναι κάθε θετικός ακέραιος αριθμός μεγαλύτερος η ίσος του: $\log_2(len + 8)$, όπου len είναι το μέγεθος του μηνύματος. Αντίστοιχα η τάξη ενός χώρου μνήμης είναι κάθε θετικός ακέραιος αριθμός μικρότερος ή ίσος από $\log_2(len + 8)$, όπου len είναι το μέγεθος του χώρου αυτού.

Κατά την αποστολή μηνυμάτων ή την προσφορά ενός χώρου μνήμης για την τοποθέτηση εισερχόμενων μηνυμάτων η εφαρμογή ορίζει και τη τάξη μεγέθους. Για ένα εισερχόμενο μήνυμα θα χρησιμοποιηθούν μόνο χώροι μνήμης που έχουν ακριβώς την ίδια τάξη μεγέθους. Συνεπώς, για κάθε τάξη μεγέθους που χρησιμοποιεί ο αποστολέας θα πρέπει να υπάρχουν χώροι μνήμης της ίδιας τάξης στον παραλήπτη.

Επιπρόσθετα, θα πρέπει να σημειωθεί πως ο χώρος, στον οποίο αποθηκεύεται η ετικέτα (tag) είναι μεγέθους 8 bit. Συνεπώς η ετικέτα που θα οριστεί θα πρέπει να είναι μεταξύ των αριθμών 0 και 255.

Το GM για να ενημερώσει την εφαρμογή για την άφιξη νέων μηνυμάτων χρησιμοποιεί γεγονότα (events). Η εφαρμογή θα πρέπει να καλέσει μια συνάρτηση η οποία θα επιστρέψει το γεγονός που εκκρεμεί. Υπάρχουν διάφορες προσεγγίσεις για το πως η εφαρμογή περιμένει ένα γεγονός, ανάλογα με την συνάρτηση που χρησιμοποιείται.

Ένα γεγονός αναπαρίσταται με τη δομή `gm_recv_event_t`. Υπάρχουν διάφορα γεγονότα τα οποία μπορεί να προκύψουν και όχι μόνο γεγονότα που αφορούν άφιξη νέων μηνυμάτων. Για αυτόν τον λόγο κάθε γεγονός χαρακτηρίζεται από τον τύπο του, ο οποίος είναι προσβάσιμος με χρήση της μακροεντολής `GM_RECV_EVENT_TYPE()`. Η παραπάνω δομή είναι τύπου `union` αφού θα πρέπει να μπορεί να αναπαριστά διαφορετικούς τύπους γεγονότων που έχουν διαφορετικά πεδία. Τα γεγονότα, τα οποία δεν μπορεί να επεξεργαστεί η εφαρμογή τα προωθεί στη συνάρτηση:

void gm_unknown (gm_port_t *port, gm_recv_event_t *event);

Για παράδειγμα σε περίπτωση που ολοκληρωθεί η αποστολή ενός μηνύματος δημιουργείται ένα γεγονός, το οποίο η εφαρμογή δεν μπορεί να το επεξεργαστεί άμεσα. Συνεπώς το προωθεί στην `gm_unknown()`. Η συνάρτηση αυτή είναι υπεύθυνη για την κλήση της callback συνάρτησης που ορίστηκε κατά την αποστολή του μηνύματος.

Οι βασικές συναρτήσεις που χρησιμοποιούνται για τον έλεγχο για εκκρεμή γεγονότα είναι οι εξής:

```
gm_recv_event_t *gm_receive(gm_port_t *p);
gm_recv_event_t *gm_blocking_receive_no_spin (gm_port_t *p);
gm_recv_event_t *gm_blocking_receive(gm_port_t *p);
```

Ακολουθεί μια περιγραφή της λειτουργικότητας για κάθε συνάρτηση.

- `gm_receive()`
  Η συνάρτηση αυτή σε περίπτωση που υπάρχει κάποιο γεγονός το επιστρέφει στον χρήστη. Σε αντίθετη περίπτωση επιστρέφει ένα εικονικό γεγονός του οποίου ο τύπος είναι `GM_NO_RECV_EVENT`.
- `gm_blocking_receive_no_spin()`
  Η συνάρτηση αυτή σε περίπτωση που δεν εκκρεμεί κάποιο γεγονός, αναστέλλει την λειτουργία της διεργασίας που την καλεί έως ότου προκύψει κάποιο γεγονός. Βέβαια, στην πραγματικότητα η συνάρτηση αυτή επιστρέφει ένα γεγονός τύπου `_GM_SLEEP_EVENT`, το οποίο προωθείται στην `gm_unknown()`. Η τελευταία αναλαμβάνει την αναστολή της εκτέλεσης της διεργασίας, έως ότου προκύψει ένα νέο γεγονός.
- `gm_blocking_receive()`
  Η λειτουργία την συνάρτησης αυτής είναι αντίστοιχή με αυτή της προηγούμενης. Ωστόσο, σε περίπτωση που δεν υπάρχει κάποιο γεγονός που να εκκρεμεί δεν αναστέλλει την λειτουργία της διεργασίας αμέσως, αλλά περιμένει ένα μικρό χρονικό διάστημα ενεργή ελέγχοντας συνεχώς για νέα γεγονότα (polling).

Τα γεγονότα του GM που αντιστοιχούν σε μηνύματα χαμηλής προτεραιότητας έχουν τύπο `GM_RECV_EVENT`, ενώ αυτά υψηλής προτεραιότητας `GM_HIGH_RECV_EVENT`. Αν και ακόμα και αυτά τα γεγονότα μπορούν να προωθηθούν στην `gm_unknown()` στις περισσότερες περιπτώσεις θα πρέπει να τα επεξεργαστεί η ίδια η εφαρμογή. Όταν τέτοια γεγονότα παραδίδονται στην εφαρμογή επιστρέφεται και ένα token λήψης, το οποίο όμως χάνεται αν προωθηθούν στην `gm_unknown()`.

Ένα γεγονός λήψης είναι τύπου `gm_recv_t` και στο `gm_recv_event_t` ορίζεται ως εξής:

```
typedef union gm_recv_event
{
  ...
  gm_recv_t recv;
  ...
}
gm_recv_event_t;
```

Μερικά από τα βασικά πεδία ενός γεγονότος λήψης (`gm_recv_t`) παρουσιάζονται στον πίνακα 3.5

Βέβαια η προσπέλαση των πεδίων αυτών έχει ορισμένες ιδιαιτερότητες. Για λόγους που σχετίζονται με την καλύτερη επίδοση του Myrinet τα πεδία αυτά είναι αποθηκευμένα σε byte order δικτύου (network byte order). Για αυτό το λόγο θα πρέπει να μετατραπούν καταλλήλως πριν χρησιμοποιηθούν οι τιμές τους. Το GM παρέχει συναρτήσεις που επιτελούν το έργο αυτό. Για παράδειγμα για τη μετατροπή ενός πεδίου `gm_u16_n_t` χρησιμοποιείται η συνάρτηση `gm_ntoh_u16()`. Ένα απλό παράδειγμα χειρισμού ενός γεγονότος λήψης παρουσιάζεται παρακάτω:

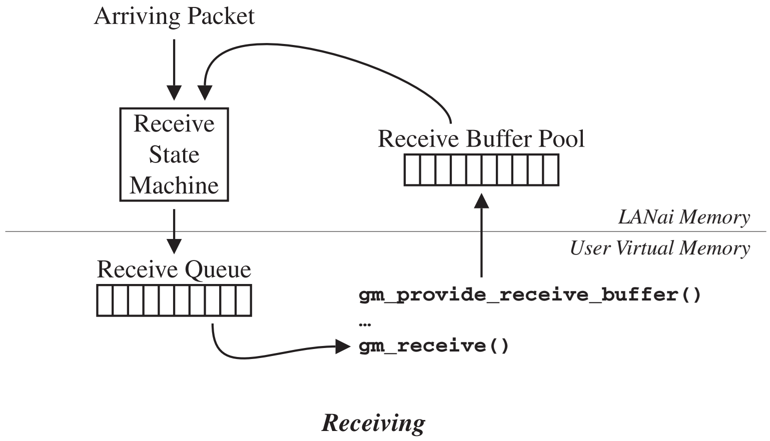
```
void handle_gm_recv_event(gm_recv_event_t *event)
{
        unsigned int len;
        void *data;
```

Τύπος	Όνομα	Σημασία
gm_u8_n_t	tag	Η ετικέτα που καθορίστηκε κατά την κλήση της συνάρτησης gm_provide_receive_buffer_with_tag.
gm_u16_n_t	sender_node_id	Αναγνωριστικό του κόμβου που έστειλε το μήνυμα.
gm_u8_n_t	sender_port_id	Αναγνωριστικό του απομακρυσμένου port από το οποίο εστάλη το μήνυμα.
gm_up_n_t	buffer	Δείκτης στον χώρο μνήμης που βρίσκονται τα δεδομένα.
gm_u32_n_t	length	Μέγεθος των δεδομένων που παρελήφθησαν
gm_u8_n_t	size	Τάξη μεγέθους του μηνύματος που παραλήφθηκε.

Πίνακας 3.5: Βασικά πεδία του τύπου gm_recv_t

```
        data = gm_ntohp(event->recv.buffer);
        len = gm_ntoh_u32(event->recv.length);
        do_something_with_data(data, len);
}
```

Τέλος στο σχήμα 3.6 παρουσιάζεται ένα διάγραμμα της λειτουργίας λήψης μηνυμάτων στο GM.



Σχήμα 3.6: Λήψη μηνύματος στο GM

3.4 Scalable Coherent Interface

3.4.1 Εισαγωγή

Το πρότυπο Scalable Coherent Interface (SCI) αναπτύχθηκε από το IEEE με στόχο τη δημιουργία ενός γενικού προτύπου, το οποίο θα επιτρέπει τη σύνδεση κάθε είδους υπολογιστικού στοιχείου, όπως:

- Επεξεργαστές.
- Μονάδες μνήμης.

- Συσκευές Ε/Ε.
- Αυτόνομα επεξεργαστικά συστήματα.

Βασικοί στόχοι του SCI κατά την ανάπτυξή του ήταν:

- Ο μεγάλος ρυθμός μετάδοσης δεδομένων.
- Η μικρή καθυστέρηση.
- Η μικρή κατανάλωση επεξεργαστικής ισχύος για την επικοινωνία
- Η υποστήριξη μεγάλου αριθμού κόμβων, χωρίς να επηρεάζεται η υψηλή επίδοση του.
- Η δυνατότητα δημιουργίας διασυνδέσεων τόσο σε μικρές, όσο και σε μεγάλες αποστάσεις.

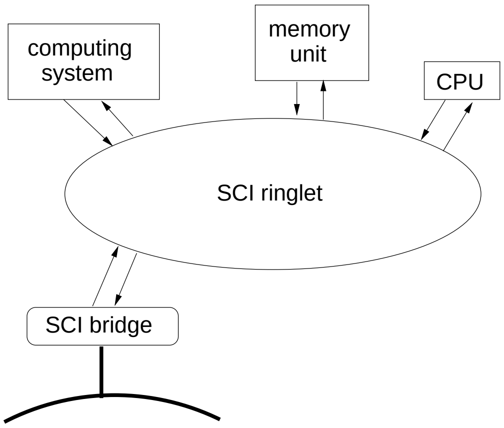
Ο αρχικός προσανατολισμός ήταν η χρήση της αρχιτεκτονικής διαδρόμου για την υλοποίηση του SCI, η οποία όμως δεν μπορούσε να ανταποκριθεί στις απαιτήσεις και έτσι εγκαταλείφθηκε.

3.4.2 Βασικά Χαρακτηριστικά του SCI

3.4.2.1 Τοπολογία δικτύων SCI

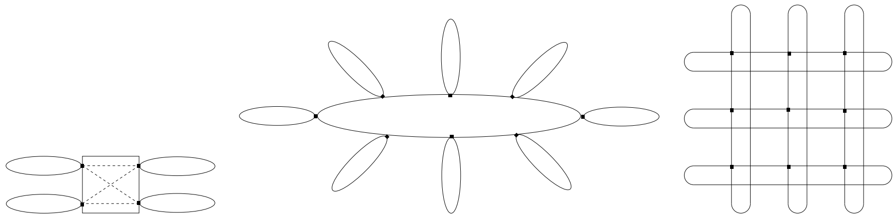
Ένα δίκτυο SCI απαρτίζεται από ζεύξεις σημείου προς σημείο. Οι ζεύξεις αυτές μπορούν να λειτουργούν ταυτόχρονα αίροντας τους περιορισμούς ενός διαμοιρασμένου μέσου όπως ο διάδρομος.

Εκτός από τους τυπικούς κόμβους που μπορούν να συνδεθούν σε ένα δίκτυο SCI, προβλέπεται και η σύνδεση προσαρμογέων που να επιτρέπουν τη συμμετοχή διαφορετικού τύπου δικτύων. Οι προσαρμογείς αυτοί ονομάζονται γέφυρες (bridges).



Σχήμα 3.7: Δακτύλιος SCI

Η βασική τοπολογία των δικτύων SCI είναι μικροί δακτύλιοι, οι οποίοι ονομάζονται ringlets. Πολύπλοκα δίκτυα μπορούν να δημιουργηθούν χρησιμοποιώντας ως δομικά στοιχεία τέτοιους δακτυλίους ή μεταγωγείς. Μερικές από τις πιο διαδεδομένες τοπολογίες δικτύων παρουσιάζονται στο σχήμα 3.8.



(a) Switch (b) Ring (c) 2D torus

Σχήμα 3.8: Διάφορες τοπολογίες SCI δικτύων

3.4.2.2 Ενιαίος χώρος διευθύνσεων

Το SCI παρέχει έναν ενιαίο χώρο διευθύνσεων που επεκτείνεται στο σύνολο των κόμβων που αποτελούν το δίκτυο, υλοποιώντας μια αρχιτεκτονική DSM (Distributed Shared Memory). Ο χώρος αυτός χρησιμοποιεί διευθύνσεις μήκους 64 bit. Τα 16 πρώτα bits χρησιμοποιούνται για τον καθορισμό του κόμβου ενώ τα υπόλοιπα 48 χρησιμοποιούνται για εσωτερική διευθυνσιοδότηση. Τα δίκτυα SCI υποστηρίζουν έως και $2^{16} = 65536 = 64k$ κόμβους.

Η υλοποίηση της DSM αρχιτεκτονικής ολοκληρώνεται με τον ορισμό ενεργειών για την ανάγνωση, εγγραφή και κλείδωμα θέσεων μνήμης. Επίσης υποστηρίζονται ενέργειες για μετάδοση σε όλους τους κόμβους (broadcast) και για την ενεργοποίηση απομακρυσμένων διακοπών (remote interrupts).

Η πρόσβαση σε πόρους του δικτύου μπορεί να πραγματοποιηθεί με απλές εντολές `LOAD` και `STORE`. Το υλικό του SCI κατόπιν θα φροντίσει να μεταφέρει τα δεδομένα που χρειάζονται. Η μέθοδος πρόσβασης αυτή έχει ως αποτέλεσμα την επίτευξη ιδιαίτερα μικρών τιμών καθυστέρησης στην επικοινωνία, αφού δεν χρειάζεται η παρέμβαση κάποιου στρώματος λογισμικού. Επιπρόσθετα, δεν απαιτεί υπολογιστική ισχύ, αφού το μόνο που χρειάζεται από τον επεξεργαστή είναι η εκτέλεση μιας εντολής και μπορεί να χρησιμοποιηθεί απευθείας από τον χώρο χρήστη.

3.4.2.3 Πρωτόκολλο Επικοινωνίας

Η λειτουργία του SCI βασίζεται σε αιτήσεις και απαντήσεις. Ένας κόμβος στέλνει μια αίτηση και περιμένει ασύγχρονα την απάντηση. Ο τρόπος λειτουργίας αυτός ονομάζεται split transactions και έχει το πλεονέκτημα ότι δεν χρειάζεται ο δίαυλος να είναι κατειλημμένος κατά την διάρκεια ικανοποίησης μιας ενέργειας. Το πρότυπο του SCI, ορίζει ότι είναι δυνατό να υπάρχουν έως και 64 εκκρεμείς αιτήσεις σε κάθε κόμβο.

Για την επικοινωνία χρησιμοποιούνται πακέτα μικρού μεγέθους. Η επικεφαλίδα και ένα πεδίο ελέγχου CRC που προστίθεται στο τέλος έχουν συνολικό μέγεθος 16 bytes, ενώ το κάθε πακέτο μπορεί να περιέχει 0, 16, 64 ή 256 bytes δεδομένων.

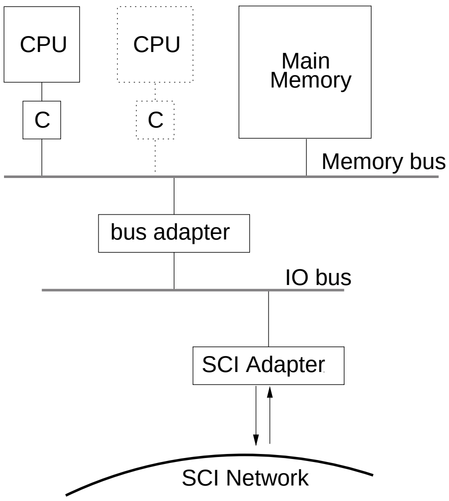
3.4.2.4 Συνάφεια κρυφής μνήμης

Το πρότυπο του SCI προβλέπει προαιρετικά και την ύπαρξη ενός στρώματος, το οποίο θα αναλαμβάνει τη διατήρηση της συνάφειας των κρυφών μνημών που βρίσκονται στους επεξεργαστές (Cache Coherence Layer). Ένα σύστημα μνημών είναι συναφές (coherent) αν, θεωρώντας μια διατεταγμένη σειρά εντολών σε μια θέση μνήμης, κάθε ανάγνωση επιστρέφει την τιμή της τελευταίας εγγραφής.

Δεδομένου ότι δεν υπάρχει κεντρικοποιημένος δίαυλος και δεν μπορούν να χρησιμοποιηθούν snooping τεχνικές, το πρωτόκολλο που χρησιμοποιείται είναι καταλόγου (directory based).

3.4.3 το SCI ως δίκτυο διασύνδεσης υπολογιστικών συστημάτων

Το πρότυπο του SCI ορίζει ένα ευέλικτο δίκτυο διασύνδεσης, το οποίο μπορεί να χρησιμοποιηθεί σε πολλά διαφορετικά επίπεδα ενός υπολογιστικού συστήματος. Ιδιαίτερα συχνή είναι η χρήση του ως τοπικού δικτύου (LAN) υψηλών επιδόσεων. Βέβαια σε αντίθεση με τα συνηθισμένα LANs το SCI δεν προσφέρει απλά τη δυνατότητα μεταφοράς δεδομένων από έναν κόμβο σε ένα άλλο. Αντίθετα προσφέρει μια DSM όψη του δικτύου, μέσα από τον ενιαίο χώρο διευθύνσεων, προσεγγίζοντας αρχιτεκτονικές NUMA (NonUniform Memory Access).



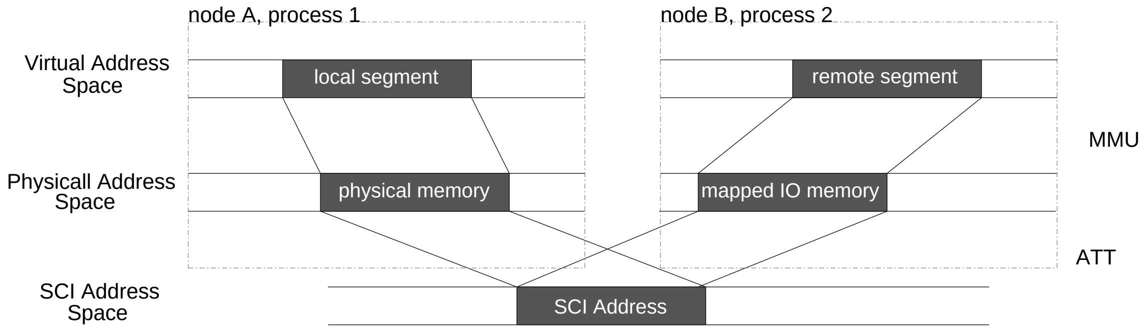
Σχήμα 3.9: Χρήση του SCI ως δίκτυο διασύνδεσης τυπικών υπολογιστικών συστημάτων

Για να χρησιμοποιηθεί το SCI σαν τοπικό δίκτυο θα πρέπει με κάποιο τρόπο να ενσωματωθεί στο υπολογιστικό σύστημα. Ο καταλληλότερος τρόπος είναι η χρήση του συστήματος Ε/Ε. Ωστόσο, η σημασιολογία των τυπικών συσκευών Ε/Ε, οι οποίες συνήθως συνδέονται σε έναν δίαυλο όπως το PCI, είναι διαφορετική από αυτή ενός δικτύου SCI. Για αυτό τον λόγο είναι απαραίτητη η χρήση ενός προσαρμογέα που να μετατρέπει εντολές του δικτύου σε εντολές στον δίαυλο Ε/Ε και το αντίστροφο.

Τέτοιες κάρτες έχουν δημιουργηθεί από την εταιρία Dolphin. Θα περιοριστούμε στην μελέτη τους και ειδικότερα σε κάρτες που χρησιμοποιούνται σε συστήματα Ε/Εi PCI, δεδομένου ότι αυτές χρησιμοποιήθηκαν για την εργασία αυτή.

Οι προσαρμογείς αυτοί σε συνδυασμό με το λογισμικό που τους συνοδεύει επιτρέπουν στις εφαρμογές να ενσωματώσουν στον εικονικό χώρο διευθύνσεων τους απομακρυσμένα τμήματα μνήμης, όπως παρουσιάζεται στο σχήμα 3.10. Τα απομακρυσμένα τμήματα αυτά μπορούν να χρησιμοποιηθούν για αναγνώσεις ή εγγραφές με κλασικές εντολές `LOAD` και `STORE`. Οι εντολές αυτές, μέσω μιας σειράς απεικονίσεων, θα εκτελεστούν στο απομακρυσμένο τμήμα μνήμης.

Για την καλύτερη κατανόηση της λειτουργίας των καρτών της Dolphin παρουσιάζονται συνοπτικά τα βήματα που ακολουθούνται για την εκτέλεση μια εντολής σε ένα απομακρυσμένο τμήμα μνήμης:



Σχήμα 3.10: Απεικόνιση απομακρυσμένων τμημάτων μνήμης στον εικονικό χώρο διευθύνσεων μέσω του SCI

- Αρχικά εκτελείται μια εντολή τύπου `LOAD` ή `STORE` από την διεργασία 2 σε μία διεύθυνση του εικονικού της χώρου.

- Η διεύθυνση αυτή, μέσω του MMU (Memory Management Unit), απεικονίζεται σε μία φυσική διεύθυνση.

- Η φυσική διεύθυνση αυτή αντιστοιχεί σε διεύθυνση συσκευής Ε/Ε, οπότε προωθείται στο δίαυλο Ε/Ε μέσω του προσαρμογέα διαδρόμου. (βλ. σχήμα 3.9)

- Ο προσαρμογέας SCI, ο οποίος είναι υπεύθυνος για τη διεύθυνση αυτή, βρίσκει την διεύθυνση SCI στην οποία αντιστοιχεί και δημιουργεί τα κατάλληλα πακέτα για την εκτέλεση της εντολής. Για την εύρεση της διεύθυνσης SCI, χρησιμοποιούνται κατάλληλοι πίνακες ATT (Address Translation Tables).

- Όταν ο προσαρμογέας του κόμβου Α δεχτεί την απομακρυσμένη αίτηση, χρησιμοποιεί DMA για να εκτελέσει την εντολή αυτή στη φυσική μνήμη, της οποίας τη διεύθυνση βρίσκει από τους δικούς του πίνακες ATT.

- Ο προσαρμογέας του κόμβου Β, τέλος, στέλνει την κατάλληλη απάντηση στην αίτηση του κόμβου Α.

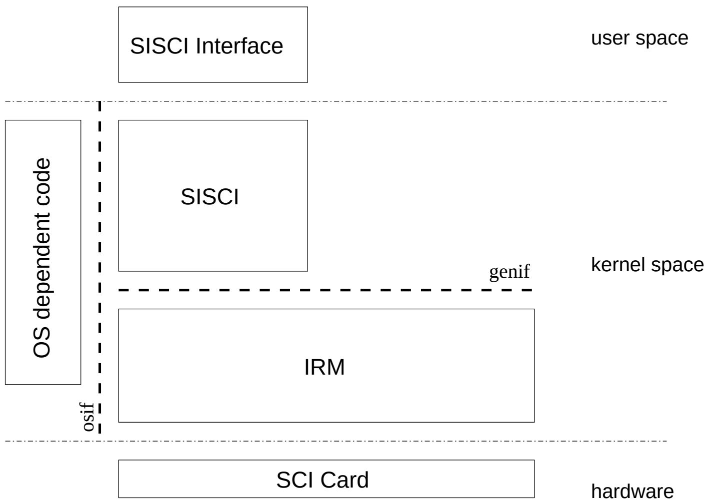
Απαιτείται φυσικά κατάλληλη αρχικοποίηση από τον κάθε κόμβο για τα τοπικά και απομακρυσμένα τμήματα μνήμης, ώστε να τοποθετηθούν τα απαραίτητα δεδομένα στους πίνακες ATT.

Η χρήση του SCI για διασύνδεση κοινών υπολογιστικών συστημάτων, έχει το πλεονέκτημα ότι μπορεί να εφαρμοστεί άμεσα, απλά και με μικρό κόστος. Άλλωστε, δεν είναι τυχαίο ότι το SCI είναι ένα αρκετά δημοφιλές μέσο διασύνδεσης για συστοιχίες υπολογιστών (computer clusters).

Ένα από τα μειονεκτήματα αυτής της πρακτικής, όμως, είναι ότι δεν μπορεί να επιτευχθεί συνάφεια κρυφής μνήμης, όπως προβλέπεται προαιρετικά από το πρότυπο του SCI. Η κάρτα SCI βρίσκεται σε διαφορετικό διάδρομο από αυτόν που χρησιμοποιούν η μνήμη και οι επεξεργαστές για να επικοινωνήσουν και συνεπώς δεν μπορεί να συμμετέχει στα πρωτόκολλα που χρησιμοποιούνται για την διατήρηση της συνάφειας. Επιπρόσθετα, αφού χρησιμοποιείται το PCI για την επικοινωνία της κάρτας με το υπόλοιπο υπολογιστικό σύστημα είναι αδύνατον να αξιοποιηθούν πλήρως οι δυνατότητες που παρέχει το SCI.

3.4.4 Διαστρωμάτωση λογισμικού του SCI

Για την λειτουργία των καρτών SCI απαιτείται και το κατάλληλο λογισμικό, που θα τις ελέγχει και θα παρέχει μια ενοποιημένη διεπαφή για τον προγραμματισμό τους σε διαφορετικά επίπεδα. Επιπρόσθετα, είναι απαραίτητο να υποστηριχτούν οι κάρτες αυτές από ένα σύνολο διαφορετικών λειτουργικών συστημάτων.



Σχήμα 3.11: Διαστρωμάτωση του Λογισμικού για τον έλεγχο των καρτών SCI

Υπάρχουν, όπως φαίνεται στο σχήμα 3.11, δύο βασικά στρώματα στο λογισμικό αυτό:

- **Interconnect Resource Manager (IRM):**
  Μερικές από τις βασικές λειτουργίες του στρώματος αυτού είναι να αρχικοποιεί τους προσαρμογείς SCI, να διαχειρίζεται τους πίνακες ATT και να ελέγχει τις μηχανές DMA της κάρτας. Παρέχει τη γενική διεπαφή `genif`, για τον προγραμματισμό δικτύων SCI σε επίπεδο πυρήνα. Το IRM έχει σχεδιαστεί ώστε να είναι ανεξάρτητο του λειτουργικού συστήματος, στο οποίο ενσωματώνεται. Οι συναρτήσεις που σχετίζονται με το λειτουργικό σύστημα παρέχονται από την διεπαφή `osif` και υλοποιούνται ξεχωριστά για κάθε λειτουργικό σύστημα.

- **Software Infrastructure for SCI (SISCI)**
  Ο βασικότερος χρήστης των υπηρεσιών που προσφέρει το IRM μέσα από το `genif` είναι το SISCI, το οποίο προσφέρει μια ενοποιημένη διεπαφή για τον εύκολο και συνάμα αποδοτικό προγραμματισμό δικτύων SCI σε χώρο χρήστη. Το SISCI προσφέρει στην εφαρμογή που το χρησιμοποιεί τη δυνατότητα επικοινωνίας με άλλους κόμβους, χωρίς την μεσολάβηση του λειτουργικού συστήματος και διατηρώντας τα χαρακτηριστικά της αρχιτεκτονικής DSM που υλοποιεί το δίκτυο SCI.

3.4.5 Διεπαφή του SISCI

Σε αυτή τη παράγραφο θα παρουσιαστούν σύντομα κάποια από τα βασικά στοιχεία της διεπαφή του SISCI. Για μια πιό αναλυτική παρουσίασή του βλ. [Dol01].

3.4.5.1 Βασικά χαρακτηριστικά

Αρχικά θα παρουσιαστούν ορισμένα χαρακτηριστικά που διέπουν στο σύνολο της τη διεπαφή του SISCI.

- **Σφάλματα:**
  Κατά την εκτέλεση των διάφορων συναρτήσεων είναι πιθανό να προκύψουν σφάλματα, που ως αποτέλεσμα έχουν την ανεπιτυχή ολοκλήρωση τους. Η εφαρμογή θα πρέπει πάντα να ελέγχει για την παρουσία τέτοιων σφαλμάτων. Για τον σκοπό αυτό υπάρχει η δομή `sci_error_t`, η οποία ή θα είναι ίση με `SCI_ERR_OK`, αν η συνάρτηση ολοκληρώθηκε επιτυχώς είτε θα έχει μια άλλη τιμή η οποία θα περιγράφει το ακριβές σφάλμα που συνέβη Οι περισσότερες συναρτήσεις δέχονται ως όρισμα έναν δείκτη σε μία τέτοια δομή, ώστε να ενημερώσουν κατάλληλα την εφαρμογή που τις χρησιμοποιεί.

- **Διαφανείς περιγραφητές (opaque descriptors)**:
  Βασικό στοιχείο της διεπαφής του SISCI είναι η έννοια του πόρου (resource). Πόροι για παράδειγμα θεωρούνται τοπικά ή απομακρυσμένα τμήματα μνήμης. Για την διαχείριση του κάθε πόρου χρησιμοποιούνται διαφανείς περιγραφητές, που είναι δομές, των οποίων τα πεδία τους δεν είναι απευθείας προσβάσιμα στην εφαρμογή και η πρόσβαση πραγματοποιείται μόνο μέσα από τις κατάλληλες συναρτήσεις.

- **Γεγονότα (events)**:
  Κατά την διάρκεια εκτέλεσης εφαρμογών που χρησιμοποιούν το SISCI είναι πιθανό να συμβούν ορισμένα γεγονότα που σχετίζονται με τους διάφορους πόρους που έχουν αρχικοποιηθεί. Τα γεγονότα αυτά μπορούν να αναγνωριστούν και να αντιμετωπιστούν από την εφαρμογή είτε σύγχρονα είτε ασύγχρονα. Ο ασύγχρονος χειρισμός τους πραγματοποιείται μέσω μιας συνάρτησης callback, η οποία θα καλείται κάθε φορά που συμβαίνει ένα γεγονός. Η συνάρτηση αυτή ορίζεται κατά την αρχικοποίηση του συγκεκριμένου πόρου.

- **Flags:**
  Για καλύτερο έλεγχο στον τρόπο λειτουργίας και μεγαλύτερη ευελιξία στις περισσότερες συναρτήσεις υπάρχει μια παράμετρος `flags`, η οποία χρησιμοποιείται από την εφαρμογή. Μέσω της παραμέτρου αυτής η εφαρμογή παραμετροποιεί τις λειτουργίες που επιτελεί η συνάρτηση ανάλογα με τις ανάγκες της.

3.4.5.2 Αρχικοποίηση και τερματισμός

Για την αρχικοποίηση και τον τερματισμό του περιβάλλοντος SISCI χρησιμοποιούνται οι συναρτήσεις:

```
void SCIInitialize(unsigned int flags, sci_error_t *error);
void SCITerminate(void);
```

Επιπρόσθετα, είναι απαραίτητο να αρχικοποιηθεί μια εικονική συσκευή, μέσω της οποίας θα κληθούν οι υπόλοιπες συναρτήσεις του SISCI. Η συσκευή αυτή απεικονίζεται σε μία δομή `sci_desc_t` και περιέχει τα κατάλληλα στοιχεία για την επικοινωνία με τους διάφορους εσωτερικούς μηχανισμούς του SISCI. Παρέχονται δύο συναρτήσεις για το "άνοιγμα" και "κλείσιμο" της συσκευής αυτής:

```
void SCIOpen(sci_desc_t *sd, unsigned int flags, sci_error_t *error);
void SCIClose(sci_desc_t sd, unsigned int flags, sci_error_t *error);
```

3.4.5.3 Τοπικά τμήματα μνήμης

Για την δημιουργία ενός τοπικού τμήματος μνήμης χρησιμοποιείται η συνάρτηση:

```
void SCICreateSegment(sci_desc_t sd,
                      sci_local_segment_t *segment,
                      unsigned int segmentId,
                      unsigned int size,
                      sci_cb_local_segment_t callback,
                      void *callbackArg,
                      unsigned int flags,
                      sci_error_t *error);
```

Παράμετρος	Σημασία
`sci_desc_t sd`	Περιγραφητής της εικονικής συσκευής που χρησιμοποιείται.
`sci_local_segment_t *segment`	Περιγραφητής του τοπικού τμήματος μνήμης που θα δημιουργηθεί.
`unsigned int segmentId`	Αναγνωριστικό του τμήματος. Ο αριθμός αυτός είναι μοναδικός για κάθε τοπικό τμήμα του κόμβου.
`unsigned int size`	Μέγεθος του τμήματος.
`sci_cb_local_segment_t callback`	Συνάρτηση callback του τμήματος για ασύγχρονο χειρισμό των γεγονότων που συμβαίνουν σε αυτό.
`void *callbackArg`	Όρισμα με το οποίο θα κληθεί η συνάρτηση callback.
`unsigned int flags`	Το σημαντικότερο flag της συνάρτησης αυτής είναι το `SCI_FLAG_USE_CALLBACK` που χρησιμοποιείται για την ενεργοποίηση της callback συνάρτησης στο τμήμα.
`sci_error_t *error`	Σφάλμα που τυχόν παρουσιάστηκε κατά την εκτέλεση της συνάρτησης.

Πίνακας 3.6: Παράμετροι της συνάρτησης `SCICreateSegment`

Η συνάρτηση αυτή χρησιμοποιείται για την δέσμευση μνήμης και την ενημέρωση των πιο χαμηλών στρωμάτων για την δημιουργία του τμήματος. Κοινές συναρτήσεις σαν τη `malloc()` δεν μπορούν να χρησιμοποιηθούν διότι θα πρέπει το τμήμα μνήμης να είναι συνεχόμενο και αγκιστρωμένο (pinned down) στη φυσική μνήμη. Οι παράμετροι της συνάρτησης παρουσιάζονται στον πίνακα 3.6.

Βασικά γεγονότα που σχετίζονται με ένα τοπικό τμήμα μνήμης είναι τα `SCI_CB_CONNECT` και `SCI_CB_DISCONNECT` που ενημερώνουν για τη σύνδεση και αποσύνδεση ενός απομακρυσμένου κόμβου στο τμήμα αυτό.

Τα γεγονότα αυτά μπορούν να αναγνωριστούν σύγχρονα χρησιμοποιώντας την συνάρτηση `SCIWaitForLocalSegmentEvent()`. Για ασύγχρονο χειρισμό των γεγονότων η συνάρτηση callback, που θα πρέπει να οριστεί, είναι τύπου:

```
typedef sci_callback_action_t
        (*sci_cb_local_segment_t)(void *arg,
                                  sci_local_segment_t segment,
                                  sci_segment_cb_reason_t reason,
                                  unsigned int nodeId,
                                  unsigned int localAdapterNo,
```

```
                    sci_error_t error);
```

Όταν το τμήμα αυτό δεν χρειάζεται πλέον μπορεί να απελευθερωθεί με τη συνάρτηση `SCIRemoveSegment()`.

Για να γίνει διαθέσιμο ένα τοπικό τμήμα στο δίκτυο SCI, ώστε να συνδεθούν και να το χρησιμοποιήσουν απομακρυσμένοι κόμβοι, είναι απαραίτητη η κλήση των συναρτήσεων:

```
void SCIPrepareSegment(sci_local_segment_t segment,
                       unsigned int localAdapterNo,
                       unsigned int flags,
                       sci_error_t *error);

void SCISetSegmentAvailable(sci_local_segment_t segment,
                            unsigned int localAdapterNo,
                            unsigned int flags,
                            sci_error_t * error);
```

Η πρώτη συνάρτηση εκτελεί ορισμένους ελέγχους και απεικονίζει το τμήμα στον 64 bit χώρο διευθύνσεων του SCI. Η δεύτερη συνάρτηση κάνει το τμήμα ορατό στους υπόλοιπους κόμβους του δικτύου.

Η συνάρτηση `SCISetSegmentUnavailable()`, αντίθετα, αποτρέπει νέες συνδέσεις στο εν λόγω τμήμα μνήμης. Οι ήδη υπάρχουσες συνδέσεις δεν επηρεάζονται.

Για να έχει η εφαρμογή πρόσβαση σε ένα τοπικό τμήμα μνήμης θα πρέπει πρώτα να το απεικονίσει στον εικονικό χώρο διευθύνσεών της. Η λειτουργία αυτή επιτελείται από την συνάρτηση:

```
void *SCIMapLocalSegment(sci_local_segment_t segment,
                         sci_map_t *map,
                         unsigned int offset,
                         unsigned int size,
                         void *addr,
                         unsigned int flags,
                         sci_error_t *error);
```

Οι απαραίτητες πληροφορίες σχετικά με την απεικόνιση αποθηκεύονται σε μία μεταβλητή τύπου `sci_map_t`. Οι παράμετροί `offset` και `size` εκφράζουν την απόσταση από την αρχή του τοπικού τμήματος μνήμης και το μέγεθος της απεικονιζόμενης περιοχής. Η τιμή επιστροφής της συνάρτησης είναι ένας δείκτης στην αρχή της περιοχής της εικονικής μνήμης που έγινε η απεικόνιση. Η θέση αυτή μπορεί να χρησιμοποιηθεί για εγγραφές και αναγνώσεις από την εφαρμογή.

3.4.5.4 Απομακρυσμένα τμήματα μνήμης

Για να μπορέσει ένα πρόγραμμα να χρησιμοποιήσει ένα απομακρυσμένο τμήμα μνήμης θα πρέπει πρώτα να συνδεθεί σε αυτό με τη βοήθεια της συνάρτησης:

```
void SCIConnectSegment(sci_desc_t sd,
                       sci_remote_segment_t *segment,
                       unsigned int nodeId,
                       unsigned int segmentId,
                       unsigned int localAdapterNo,
```

```
                    sci_cb_remote_segment_t callback,
                    void *callbackArg,
                    unsigned int timeout,
                    unsigned int flags,
                    sci_error_t *error);
```

Η λειτουργία της αποσύνδεσης από ένα απομακρυσμένο τμήμα πραγματοποιείτα με την συνάρτηση `SCIDisconnectSegment()`.

Αντίστοιχα με τα τοπικά τμήματα μνήμης, έτσι και τα απομακρυσμένα πρέπει να απεικονιστούν στον εικονικό χώρο διευθύνσεων της διεργασίας, ώστε να έχει η εφαρμογή πρόσβαση σε αυτά. Η συνάρτηση που χρησιμοποιείται είναι η :

```
volatile void *SCIMapRemoteSegment(sci_remote_segment_t segment,
                                   sci_map_t *map,
                                   unsigned int offset,
                                   unsigned int size,
                                   void *addr,
                                   unsigned int flags,
                                   sci_error_t *error);
```

Ο δείκτης που επιστρέφεται από την συνάρτηση είναι τύπου `volatile`, ώστε να μην θεωρήσει ο μεταγλωττιστής ότι μόνο η εφαρμογή που μεταγλωττίζει έχει πρόσβαση σε αυτή τη θέση της μνήμης. Η θέση της μνήμης αυτή αντιστοιχεί σε μνήμη ενός απομακρυσμένου κόμβου και τα περιεχόμενα της μπορούν να αλλάξουν ανά πάσα στιγμή.

3.4.5.5 Μεταφορές DMA

Με την απεικόνιση ενός απομακρυσμένου τμήματος μνήμης στον εικονικό χώρο διευθύνσεων μιας διεργασίας η μεταφορά δεδομένων μεταξύ δύο κόμβων μπορεί απλά να επιτευχθεί με εντολές `LOAD` και `STORE`. Ένα πρόβλημα που παρουσιάζεται είναι ότι κατά τη μεταφορά μεγάλου όγκου δεδομένων, χρησιμοποιώντας την παραπάνω μέθοδο, ο επεξεργαστής θα παραμένει κατειλημμένος για μεγάλο χρονικό διάστημα. Το SISCI παρέχει δυνατότητες μεταφοράς των δεδομένων με DMA, ώστε ο επεξεργαστής να παραμένει ελεύθερος να εκτελέσει άλλες λειτουργίες κατά τη διάρκεια της μεταφοράς (βλ και παράγραφο 1.2.2).

Το SISCI χρησιμοποιεί ουρές DMA (DMA queues) για τον σκοπό αυτό. Οι ουρές DMA έχουν 6 διαφορετικές καταστάσεις στις οποίες μπορούν να βρίσκονται:

- `IDLE`: Η ουρά έχει δημιουργηθεί επιτυχώς και είναι άδεια.
- `GATHER`: Η ουρά δέχεται δεδομένα προς αποστολή.
- `POSTED`: Τα δεδομένα αποστέλλονται στον απομακρυσμένο κόμβο.
- `DONE`: Όλες οι μεταφορές των δεδομένων έχουν ολοκληρωθεί επιτυχώς.
- `ERROR`: Παρουσιάστηκε κάποιο πρόβλημα κατά τη μεταφορά δεδομένων.
- `ABORTED`: Η εφαρμογή διέκοψε την DMA μεταφορά.

Μια ουρά DMA δημιουργείται με την κλήση της συνάρτησης:

```
void SCICreateDMAQueue(sci_desc_t sd,
```

```
sci_dma_queue_t *dq,
unsigned int localAdapterNo,
unsigned int maxEntries,
unsigned int flags,
sci_error_t *error);
```

Όταν η ουρά δεν χρειάζεται πλέον μπορεί να απελευθερωθεί με την κλήση της συνάρτησης `SCIRemoveDMAQueue()`.

Η εφαρμογή προσθέτει μεταφορές στην ουρά μέσω της συνάρτησης

```
sci_dma_queue_state_t SCIEnqueueDMATransfer(sci_dma_queue_t dq,
                                            sci_local_segment_t localSegment,
                                            sci_remote_segment_t remoteSegment,
                                            unsigned int localOffset,
                                            unsigned int remoteOffset,
                                            unsigned int size,
                                            unsigned int flags,
                                            sci_error_t *error);
```

Στην γενική περίπτωση μεταφέρονται δεδομένα μήκους `size` από το τοπικό τμήμα αρχίζοντας από τη θέση `localOffset` στο απομακρυσμένο τμήμα στη θέση `remoteOffset`. Σε περίπτωση που έχει ενεργοποιηθεί το flag `SCI_FLAG_DMA_READ` τα δεδομένα μεταφέρονται από το απομακρυσμένο τμήμα στο τοπικό.

Για να προωθηθεί η ουρά στον προσαρμογέα SCI, ο οποίος θα φροντίσει για την μεταφορά των δεδομένων, θα πρέπει να κληθεί η συνάρτηση:

```
void SCIPostDMAQueue(sci_dma_queue_t dq,
                     sci_cb_dma_t callback,
                     void *callbackArg,
                     unsigned int flags,
                     sci_error_t *error);
```

Υπάρχουν τρεις τρόποι για να αναγνωρίσει η εφαρμογή αν και πότε ολοκληρώθηκαν οι μεταφορές των δεδομένων σε μια ουρά DMA:

- **σύγχρονα:**
  Η εφαρμογή μπορεί να περιμένει το αποτέλεσμα της μεταφοράς των δεδομένων μιας ουράς με την συνάρτηση `SCIWaitForDMAQueue()`.
- **ελέγχοντας:**
  Η εφαρμογή μπορεί να ελέγχει τη κατάσταση της ουράς χρησιμοποιώντας την συνάρτηση `SCIDMAQueueState()` για να καθορίσει αν ολοκληρώθηκαν επιτυχώς οι μεταφορές.
- **ασύγχρονα:**
  Τέλος η εφαρμογή μπορεί να ενημερωθεί ασύγχρονα για το αποτέλεσμα των μεταφορών μιας ουράς ορίζοντας μια συνάρτηση callback.

Δύο ακόμα συναρτήσεις που μπορούν να χρησιμοποιηθούν είναι οι:

- `SCIDAbortDMAQueue()` που τερματίζει την επεξεργασία των DMA μεταφορών μιας ουράς.

- `SCIResetDMAQueue()` που αρχικοποιεί μια DMA ουρά, της οποίας οι μεταφορές ολοκληρώθηκαν.

Θα πρέπει να σημειωθεί πως για τον απομακρυσμένο κόμβο δεν υπάρχει καμία διαφορά είτε χρησιμοποιηθεί μια κλασική μέθοδος όπως η `memcpy()` είτε χρησιμοποιηθούν ουρές DMA. Η διαφορά έγκειται στο πως θα μεταφερθούν τα δεδομένα στον κόμβο που κάνει τη μεταφορά από τη μνήμη στον προσαρμογέα SCI.

3.4.5.6 Διακοπές

Οι διακοπές (interrupts) είναι ένας πόρος που προσφέρει το SISCI για την ειδοποίηση απομακρυσμένων διεργασιών ότι συνέβη κάποιο προκαθορισμένο γεγονός. Κάθε διακοπή αναγνωρίζεται από έναν μοναδικό θετικό ακέραιο αριθμό. Η εφαρμογή που θέλει να ειδοποιηθεί για ένα γεγονός χρησιμοποιεί μια τοπική διακοπή (local interrupt), ενώ η εφαρμογή που θέλει να ειδοποιήσει για ένα γεγονός χρησιμοποιεί μια απομακρυσμένη διακοπή (remote interrupt).

Για την δημιουργία μιας τοπικής διακοπής χρησιμοποιείται η συνάρτηση:

```
void SCICreateInterrupt(sci_desc_t sd,
                        sci_local_interrupt_t *interrupt,
                        unsigned int localAdapterNo,
                        unsigned int *interruptNo,
                        sci_cb_interrupt_t callback,
                        void *callbackArg,
                        unsigned int flags,
                        sci_error_t *error);
```

Ο χειρισμός μιας τοπικής διακοπής μπορεί να γίνει είτε σύγχρονα, μέσω της `SCIWaitForInterrupt()`, είτε ασύγχρονα μέσω του ορισμού μιας callback συνάρτησης.

Όταν η τοπική διακοπή δεν είναι πλέον απαραίτητη μπορεί να ελευθερωθεί με τη συνάρτηση `SCIRemoveInterrupt()`

Η αρχικοποίηση μιας απομακρυσμένης διακοπής πραγματοποιείται με την σύνδεση σε μία τοπική διακοπή ενός άλλου κόμβου. Για την λειτουργία αυτή χρησιμοποιείται η συνάρτηση `SCIConnectInterrupt()`.

Η ενεργοποίηση μιας απομακρυσμένης διακοπής, η οποία θα δημιουργήσει ένα γεγονός στην τοπική διακοπή ενός άλλο κόμβου γίνεται μέσω της συνάρτησης `SCITriggerInterrupt()`.

Τέλος για την αποσύνδεση μιας απομακρυσμένης διακοπής χρησιμοποιείται η συνάρτηση `SCIDisconnectInterrupt()`.

Κεφάλαιο 4

Σχεδιασμός της συσκευής GeNBD

Σε αυτό το κεφάλαιο θα γίνει μια παρουσίαση της δομής και των αρχών λειτουργίας της δικτυακής συσκευής block που υλοποιήθηκε στα πλαίσια της εργασίας αυτής. Η συσκευή αυτή ονομάστηκε GeNBD (General Network Block Device) και αναπτύχθηκε στο λειτουργικό σύστημα Linux.

4.1 Γενικά

Η συσκευή αυτή δημιουργήθηκε ώστε να παρέχει αντίστοιχη λειτουργικότητα με αυτή της συσκευής nbd (βλ. παράγραφο 2.5). Συγκεκριμένα, προσφέρει δυνατότητες απομακρυσμένης αποθήκευσης δημιουργώντας μια εικονική συσκευή block, η οποία προωθεί τις αιτήσεις σε ένα άλλο υπολογιστικό σύστημα όπου και ικανοποιούνται.

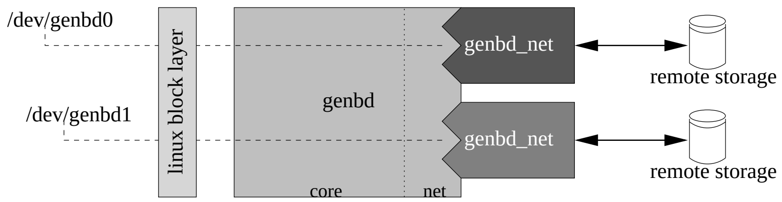
Σε αντίθεση με τη συσκευή nbd, όμως προτιμήθηκε μια πιο γενική προσέγγιση στη σχεδίασή της. Η συσκευή χωρίζεται σε δύο βασικά επίπεδα. Στο `genbd` που αποτελεί τον βασικό πυρήνα της συσκευής και σε διάφορα επιμέρους κομμάτια τα οποία υλοποιούν την απομακρυσμένη αποθήκευση. Τα κομμάτια αυτά ονομάζονται `genbd_net`.

Με αυτόν τον τρόπο μπορούν να υποστηριχτούν πολλοί διαφορετικοί τρόποι απομακρυσμένης αποθήκευσης, χωρίς να χρειάζεται να υλοποιούνται κάθε φορά τα κοινά κομμάτια.

Επιπρόσθετα, ο σχεδιασμός ήταν τέτοιος, ώστε να μπορεί να προστεθεί επιπλέον λειτουργικότητα, χωρίς να χρειαστούν μεγάλες αλλαγές. Τέτοια λειτουργικότητα θα μπορούσε για παράδειγμα να είναι η χρήση πολλαπλών εξυπηρετητών, ώστε το σύστημα αποθήκευσης να είναι πιο επεκτάσιμο.

Παρακάτω αναλύεται πιο διεξοδικά η σχεδίαση της συσκευής.

4.2 Το επίπεδο genbd



Σχήμα 4.1: Λειτουργία του στρώματος genbd

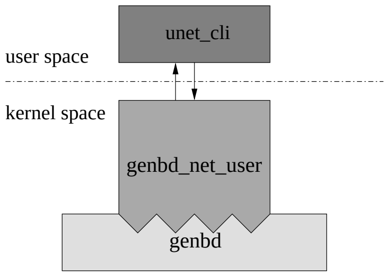
Βασικός σκοπός του `genbd` είναι η αλληλεπίδραση με το στρώμα block του Linux και η υποστήριξη ενός συνόλου από `genbd_net` υλοποιήσεων. Την πρώτη λειτουργία την αναλαμβάνει το `core` μέρος, ενώ τη δεύτερη το `net` μέρος. Το genbd αποτελεί τον οδηγό block για μια σειρά από συσκευές, κάθε μία από τις οποίες χρησιμοποιεί και ένα `genbd_net` για την απομακρυσμένη αποθήκευση και ανάκτηση των δεδομένων.

Το `genbd` δεν παρέχει μεγάλη λειτουργικότητα και βασικά αποτελεί μια διεπαφή μεταξύ του στρώματος block του Linux και των διαφόρων `genbd_net` υλοποιήσεων. Η διεπαφή αυτή λειτουργεί προωθώντας μονάδες `bio` (βλ. παράγραφο 2.2.2) στο `genbd_net` που σχετίζεται με τη συσκευή, την οποία και αφορούν.

Βέβαια η αρχιτεκτονική του `genbd` είναι τέτοια ώστε να μπορεί να επεκταθεί αποκτώντας μεγαλύτερη λειτουργικότητα, την οποία δυνητικά θα μπορούν να χρησιμοποιήσουν τα διάφορα `genbd_net` που προσκολλούνται σε αυτό.

4.3 Το genbd_net_user

Το `genbd_net_user` είναι μια υλοποίηση ενός `genbd_net`. Βασικός στόχος της υλοποίησης αυτής είναι η δημιουργία ενός γενικού επιπέδου που απλώς θα προωθεί τις αιτήσεις που δέχεται σε μια εφαρμογή χώρου χρήστη. Η εφαρμογή αυτή θα έχει πλέον ως ρόλο την επικοινωνία με τον εξυπηρετητή και την επεξεργασία των αιτήσεων.



Σχήμα 4.2: Λειτουργία του genbd_net_user

Η προσέγγιση αυτή έχει ορισμένα πλεονεκτήματα και μειονεκτήματα. Το βασικό μειονέκτημα της είναι ότι η επικοινωνία με τον χώρο χρήστη, όσο αποδοτική και να είναι, αποτελεί μια επιπλέον καθυστέρηση στην ολοκλήρωση των αιτήσεων. Επιπρόσθετα, ο προγραμματισμός στον χώρο πυρήνα προσφέρει πολύ περισσότερες δυνατότητες από ότι στον χώρο χρήστη, αφού υπάρχει ολοκληρωτικός έλεγχος σε κάθε επίπεδο.

Βέβαια, η μέθοδος που χρησιμοποιήθηκε επιτρέπει τη δημιουργία ενός πλαισίου, στο οποίο θα μπορούν πολύ εύκολα να ενταχθούν νέοι τύποι δικτύων. Επίσης, ο προγραμματισμός στον χώρο χρήστη είναι πιο εύκολος, δεδομένου ότι οι διεπαφές είναι σταθερές και σαφώς ορισμένες ενώ, το λειτουργικό σύστημα προσφέρει λειτουργίες, όπως η προστασία μνήμης, οι οποίες διευκολύνουν σε μεγάλο βαθμό τον προγραμματιστή.

Βασικό κομμάτι του `genbd_net_user` είναι η επικοινωνία με την εφαρμογή χώρου χρήστη. Για να γίνει αποδοτική η επικοινωνία αυτή τα δεδομένα block δεν αντιγράφονται σε χώρους μνήμης της διεργασίας χρήστη, μιας και η διαδικασία αυτή θα προκαλούσε επιπλέον καθυστέρηση. Αντίθετα τα δεδομένα απεικονίζονται στον εικονικό χώρο διευθύνσεων της διεργασίας, ώστε να περιοριστεί όσο το δυνατόν η καθυστέρηση. Για την απεικόνιση αυτή χρησιμοποιείται μια συσκευή χαρακτήρων (character device).

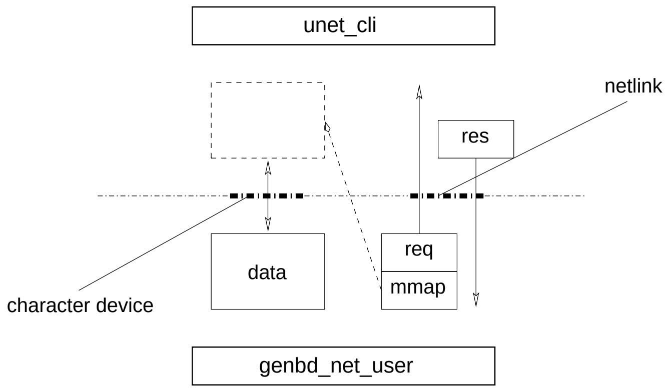
Συνεπώς οι αιτήσεις που στέλνει το `genbd_net_user` προς την εφαρμογή του χώρου χρήστη, η οποία ονομάζεται `unet_cli` θα πρέπει να έχουν δύο ειδών πληροφορίες:

- Πληροφορίες σχετικά με την εντολή block που πρέπει να πραγματοποιηθεί. Συγκεκριμένα, θα πρέπει να καθοριστεί το μέγεθος, η θέση και ο τύπος της (ανάγνωση ή εγγραφή).

- Πληροφορίες σχετικά με το πως το `unet_cli` θα απεικονίσει τους χώρους των δεδομένων block στον εικονικό χώρο διευθύνσεών του. Οι χώροι αυτοί ή περιέχουν τα δεδομένα προς εγγραφή στη συσκευή ή αποτελούν το σημείο που πρέπει να τοποθετηθούν τα δεδομένα προς ανάγνωση.

Για την αποστολή των αιτήσεων και των απαντήσεων χρησιμοποιείται το `netlink`, μέσω του οποίου μπορούν να επικοινωνήσουν ο πυρήνας και εφαρμογές χώρου χρήστη, χρησιμοποιώντας την διεπαφή των `sockets`.

Στο σχήμα 4.3 παρουσιάζεται το μοντέλο επικοινωνίας μεταξύ του `genbd_net_user` και της εφαρμογής `unet_cli`.



Σχήμα 4.3: Επικοινωνία μεταξύ `genbd_net_user` *και* `unet_cli`

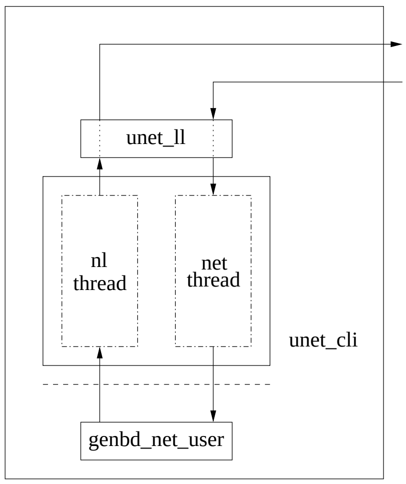
4.4 Η εφαρμογή unet_cli

Η βασική λειτουργία του `unet_cli` προγράμματος έχει ως εξής:

- Το `unet_cli` δέχεται αιτήσεις από τον χώρο πυρήνα, για αποθήκευση ή ανάκτηση δεδομένων.
- Μετά από κάποια κατάλληλη επεξεργασία οι αιτήσεις αυτές, αφού μετασχηματισθούν κατάλληλα, προωθούνται στην απομακρυσμένη εφαρμογή `unet_srv`.
- Μόλις ολοκληρωθεί η ικανοποίηση μιας αίτησης από το `unet_srv` αποστέλλεται η αντίστοιχη απάντηση στο `unet_cli`.
- Μόλις ληφθεί η απάντηση, ειδοποιείται το `genbd_net_user` για την ολοκλήρωση της αρχικής αίτησης.

Το `unet_cli` συνεπώς, θα πρέπει να επικοινωνεί ταυτόχρονα με δύο διαφορετικές οντότητες: το `genbd_net_user` και το `unet_srv`. Για να επιτευχθεί ασύγχρονη επικοινωνία είναι απαραίτητη η χρήση δύο διαφορετικών νημάτων εκτέλεσης (threads). Κατά την ασύγχρονη επικοινωνία μεταξύ του εξυπηρετητή και του πελάτη δεν χρειάζεται να ολοκληρωθεί η επεξεργασία μιας αίτησης από τον εξυπηρετητή για να σταλεί η επόμενή της και συνεπώς η απόκριση της συσκευής είναι καλύτερη.

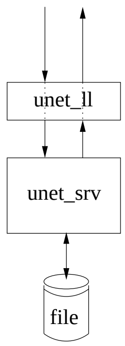
Τα δύο νήματα εκτέλεσης που αποτελούν την εφαρμογή `unet_cli` είναι τα εξής:



Σχήμα 4.4: Λειτουργία της unet_cli εφαρμογής

- **nl thread:**
  Το νήμα εκτέλεσης αυτό είναι υπεύθυνο για να δέχεται τις αιτήσεις από τον χώρο πυρήνα. Το όνομα του προέρχεται από την διεπαφή επικοινωνίας που χρησιμοποιείται (`netlink`). Μόλις ληφθεί μια αίτηση από τον χώρο πυρήνα και γίνουν οι κατάλληλες λειτουργίες, αποστέλνεται μια αίτηση στον εξυπηρετητή, μέσω του `unet_ll`.
- **net thread:**
  Το νήμα αυτό περιμένει δεδομένα που προέρχονται από το δίκτυο, μέσω του οποίου πραγματοποιείται η επικοινωνία με τον εξυπηρετητή. Μόλις ληφθεί μια απάντηση σχετικά με μια αίτηση, εκτελούνται κάποιες απαραίτητες λειτουργίες και ενημερώνεται ο χώρος πυρήνα για το αποτέλεσμα της αρχικής αίτησης του `genbd_net_user`.

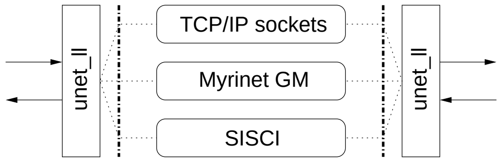
4.5 Η εφαρμογή unet_srv



Σχήμα 4.5: Λειτουργία της unet_srv εφαρμογής

Η εφαρμογή `unet_srv` η οποία αποτελεί τον εξυπηρετητή είναι αρκετά απλή. Λειτουργεί σύγχρονα και ικανοποιεί τις αιτήσεις που αποστέλλονται από το `unet_cli` χρησιμοποιώντας ένα αρχείο, το οποίο μπορεί να είναι είτε ένα απλό αρχείο, είτε ένα αρχείο συσκευής block.

4.6 Η βιβλιοθήκη unet_ll



Σχήμα 4.6: Λειτουργία της unet_ll βιβλιοθήκης

Η βιβλιοθήκη `unet_ll` παρέχει μια ιδιαίτερα απλή διεπαφή για την επικοινωνία δύο κόμβων, μέσω ενός δικτύου διασύνδεσης. Η επικοινωνία πραγματοποιείται με συναρτήσεις `send` και `recv` για αποστολή και λήψη δεδομένων. Την διεπαφή αυτή χρησιμοποιούν για την επικοινωνία τους οι εφαρμογές `unet_cli` και `unet_srv`, αν και η χρήση της δεν περιορίζεται σε αυτές.

Η διεπαφή αυτή μπορεί να υλοποιηθεί με πολλούς διαφορετικούς τρόπους και με πολλά διαφορετικά δίκτυα διασύνδεσης. Προσφέρει, έτσι, ένα διαφανή τρόπο για την υποστήριξη πολλαπλών δικτύων και υλοποιήσεων. Για την εργασία αυτή έγιναν απλές υλοποιήσεις της `unet_ll` για τα δίκτυα διασύνδεσης που περιγράφηκαν στο κεφάλαιο 3.

4.7 Λειτουργία της συσκευής

Παρακάτω θα παρουσιαστεί πως ικανοποιούνται τυχόν εγγραφές ή αναγνώσεις σε μία συσκευή `genbd` η οποία χρησιμοποιεί το `genbd_net_user`.

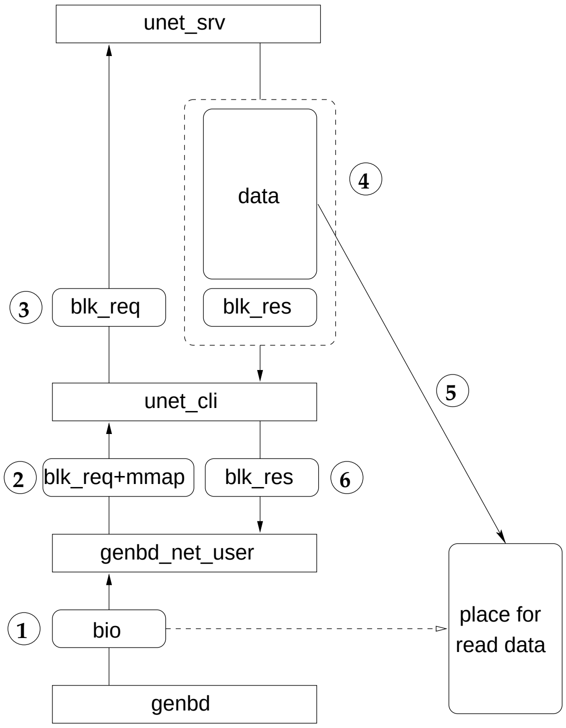
4.7.1 Ανάγνωση Δεδομένων

Η διαδικασία που ακολουθείται για την ανάγνωση δεδομένων είναι η εξής (βλ. και σχήμα 4.7)

1. Το επίπεδο `genbd` προωθεί μια μονάδα `bio` στο `genbd_net_user`.
2. Το επίπεδο `genbd_net_user` δημιουργεί μια αίτηση προς την εφαρμογή `unet_cli`. Η αίτηση αυτή αποτελείται από ένα κομμάτι με πληροφορίες για την λειτουργία block προς εκτέλεση (blk_req) και ένα κομμάτι για την απεικόνιση του μέρους, στο οποίο θα τοποθετηθούν τα δεδομένα, στον εικονικό χώρο διευθύνσεων (mmap).
3. Οι πληροφορίες που σχετίζονται με τη λειτουργία block μεταφέρονται στον εξυπηρετητή.
4. Ο εξυπηρετητής εκτελεί τη λειτουργία ανάγνωσης και επιστρέφει την απάντηση (blk_res) και τα δεδομένα (data) που ζητήθηκαν
5. Τα δεδομένα αντιγράφονται στον προκαθορισμένο χώρο.
6. Η απάντηση του εξυπηρετητή μεταφέρεται στο `genbd_net_user`, το οποίο και τερματίζει το bio κατάλληλα.

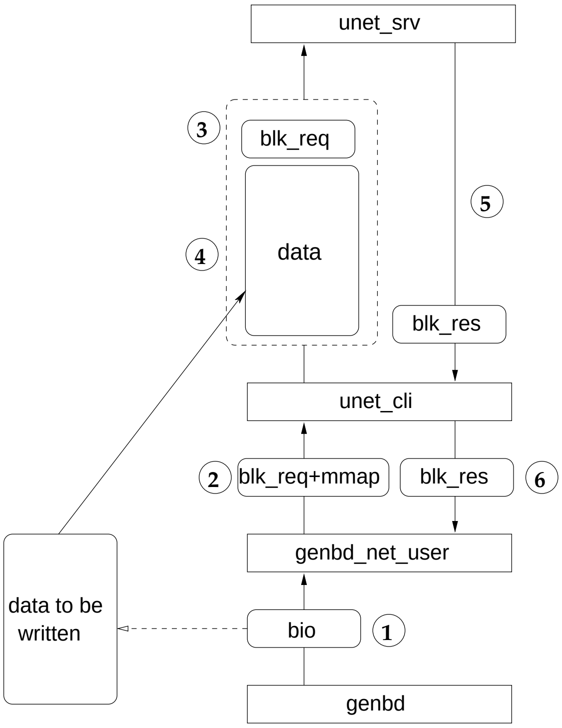
4.7.2 Εγγραφή Δεδομένων

Αντίστοιχη είναι και η διαδικασία που ακολουθείται για την εγγραφή δεδομένων στην συσκευή:



Σχήμα 4.7: Ανάγνωση δεδομένων από μία genbd συσκευή

1. Το επίπεδο `genbd` προωθεί μια μονάδα `bio` στο `genbd_net_user`.
2. Το επίπεδο `genbd_net_user` δημιουργεί μια αίτηση προς την εφαρμογή `unet_cli`. Η αίτηση αυτή αποτελείται από ένα κομμάτι με πληροφορίες για την λειτουργία block προς εκτέλεση (blk_req) και ένα κομμάτι για την απεικόνιση του μέρους, στο οποίο υπάρχουν τα δεδομένα που προορίζονται για εγγραφή στην συσκευή.
3. Οι πληροφορίες που σχετίζονται με τη λειτουργία block μεταφέρονται στον εξυπηρετητή.
4. Τα δεδομένα προς εγγραφή μεταφέρονται και αυτά στον εξυπηρετητή από τον προκαθορισμένο χώρο.
5. Ο εξυπηρετητής εκτελεί τη λειτουργία εγγραφής και επιστρέφει την απάντηση (blk_res).
6. Η απάντηση του εξυπηρετητή μεταφέρεται στο `genbd_net_user`, το οποίο και τερματίζει το bio κατάλληλα.



Σχήμα 4.8: Εγγραφή δεδομένων σε μία genbd συσκευή

Κεφάλαιο 5

Υλοποίηση της συσκευής GeNBD

Στο κεφάλαιο αυτό θα γίνει μια λεπτομερής ανάλυση της υλοποίησης της συσκευής genbd. Θα παρουσιαστούν ιεραρχικά όλα τα επίπεδα και οι διεπαφές οι οποίες χρησιμοποιήθηκαν, τόσο στον χώρο χρήστη, όσο και στον χώρο του πυρήνα.

Υπάρχουν πέντε ξεχωριστά στρώματα που χρησιμοποιούνται από την συσκευή αυτή:

- Το `genbd` και το `genbd_net_user`, τα οποία υλοποιούνται σαν modules για τον πυρήνα του Linux.
- Τα `unet_cli` και `unet_srv` που υλοποιούνται σαν τυπικές εφαρμογές χώρου χρήστη.
- Το `unet_ll`, το οποίο υλοποιείται ως δυναμική βιβλιοθήκη.

5.1 Το genbd module

Το module αυτό υλοποιεί ένα σύνολο από συσκευές block. Η κάθε συσκευή σχετίζεται με ένα `genbd_net` το οποίο φροντίζει για την απομακρυσμένη αποθήκευση δεδομένων.

Δεδομένου ότι οι συσκευές δεν σχετίζονται με τυπικές συσκευές αποθήκευσης, κατά την αρχικοποίηση τους απενεργοποιείται η χρονοδρομολόγηση Ε/Ε (βλ. και παράγραφο 2.3.6). Κάθε μονάδα `bio` προωθείται στο `genbd`, το οποίο με τη σειρά του το προωθεί στο κατάλληλο `genbd_net`. Με αυτόν τον τρόπο για κάθε δίκτυο που χρησιμοποιείται μπορεί να υλοποιηθεί ξεχωριστή χρονοδρομολόγηση για τις αιτήσεις, προσαρμοσμένη κατάλληλα στα χαρακτηριστικά του.

Σημαντικό ρόλο στην υλοποίηση της συσκευή παίζουν οι διάφορες δομές δεδομένων που χρησιμοποιούνται, οι οποίες παρουσιάζονται στη συνέχεια.

Η κεντρική δομή δεδομένων του `genbd` είναι η:

```
struct genbd_handle {
        int                      major;
        struct genbd_device      *devs;
        int                      max_devs;
        spinlock_t               nets_lock;
        struct list_head         nets;
};
```

Τα πεδία της δομής αυτής επεξηγούνται στον πίνακα 5.1.

Σε κάθε συσκευή genbd αντιστοιχεί μια μεταβλητή, η οποία είναι τύπου:

Πεδίο	Σημασία
`major`	Ο αριθμός major των genbd συσκευών block, ο οποίος μπορεί να οριστεί ως παράμετρος κατά το φόρτωμα του module.
`devs`	Ένας πίνακας από περιγραφητές για κάθε συσκευή που ελέγχεται από τον οδηγό genbd.
`max_devs`	Το μέγεθος του πίνακα `devs`.
`nets`	Μια διπλά συνδεδεμένη λίστα στην οποία τοποθετούνται οι περιγραφητές των `genbd_net` που χρησιμοποιούνται.
`nets_lock`	Μεταβλητή τύπου spinlock που χρησιμοποιείται για τον συγχρονισμό των προσπελάσεων στη λίστα `nets`.

Πίνακας 5.1: Σημασία των πεδίων του τύπου `struct genbd_handle`

```
struct genbd_device {
        unsigned long           size;
        unsigned int            blksize;
        int                     status;
        int                     flags;
        struct gendisk          *disk;
        request_queue_t         *queue;
        struct genbd_network    *net;
        spinlock_t              lock;
};
```

Το νόημα των πεδίων της εξηγείται σύντομα στον πίνακα 5.2.

Πεδίο	Σημασία
`size`	Χωρητικότητα της συσκευής σε bytes.
`blksize`	Χωρητικότητα της συσκευής σε blocks.
`status`	Κατάσταση της συσκευής.
`flags`	Παράμετροι που σχετίζονται με την συσκευή.
`disk`	Δείκτης στη μεταβλητή τύπου `gendisk`, που χρησιμοποιείται για την συσκευή αυτή.
`queue`	Ουρά αιτήσεων της συσκευής.
`net`	`genbd_net` που έχει σχετιστεί με την συσκευή αυτή.
`lock`	Μεταβλητή τύπου spinlock ώστε να συγχρονίζεται η πρόσβαση στη δομή.

Πίνακας 5.2: Σημασία των πεδίων του τύπου `struct genbd_device`

Αντίστοιχα ο τύπος `struct genbd_network` χρησιμοποιείται για να αποθηκεύσει όλες τις απαραίτητες πληροφορίες σχετικά με ένα `genbd_net`:

```
struct genbd_network {
        struct list_head                netlist;
        unsigned int                    type;
        unsigned int                    id;
        char                            name[GENBD_NET_NAMESIZE];
        struct genbd_net_operations     *ops;
        void                            *net_handle;
        struct genbd_device             *dev;
        unsigned int                    status;
};
```

Πεδίο	Σημασία
netlist	Πεδίο για την δημιουργία διπλά συνδεδεμένης λίστας.
type	Τύπος του δικτύου που χρησιμοποιείται.
id	Μοναδικό αναγνωριστικό που χαρακτηρίζει το συγκεκριμένο genbd_net.
name	Όνομα του δικτύου που χρησιμοποιείται.
ops	Συναρτήσεις μέσω των οποίων το κάθε genbd_net υλοποιεί την λειτουργικότητα που απαιτείται για τον χειρισμό των αιτήσεων block.
net_handle	Δείκτης γενικού τύπου που ορίζεται από το genbd_net και χρησιμοποιείται σαν παράμετρος στην κλήση των συναρτήσεων που ορίζονται στο πεδίο ops.
dev	Συσκευή genbd που έχει με την οποία έχει σχετιστεί το συγκεκριμένο genbd_net.
status	Πεδίο για την κατάσταση του genbd_net.

Πίνακας 5.3: Σημασία των πεδίων του τύπου `struct genbd_network`

Η λειτουργικότητα ενός genbd_net υλοποιείται κυρίως μέσω των συναρτήσεων που περιέχονται στη δομή:

```
struct genbd_net_operations {
        int (*init) (void *net_handle);
        int (*shut) (void *net_handle);
        int (*handle_bio) (void *net_handle, struct bio *bio);
};
```

Οι δύο πρώτες συναρτήσεις χρησιμοποιούνται για την αρχικοποίηση και τον τερματισμό λειτουργίας του genbd_net. Η τελευταία συνάρτηση χρησιμοποιείται για των χειρισμό των μονάδων Ε/Ε bio.

Και οι τρεις συναρτήσεις δέχονται ένα όρισμα γενικού τύπου net_handle, το οποίο το ορίζει το genbd_net και χρησιμοποιείται μόνο από αυτό. Με αυτόν τον τρόπο τα genbd_net ορίζουν την πληροφορία που χρειάζονται κατά την κλήση των συναρτήσεων τους με διαφανή τρόπο ως προς το genbd.

Κάθε συσκευή genbd και κάθε genbd_net έχει τρεις δυνατές καταστάσεις στις οποίες μπορεί να βρίσκεται.

Οι μεταβάσεις μεταξύ των καταστάσεων αυτών πραγματοποιούνται με κλήσεις συστήματος ioctl() στο ειδικό αρχείο της συσκευής block.

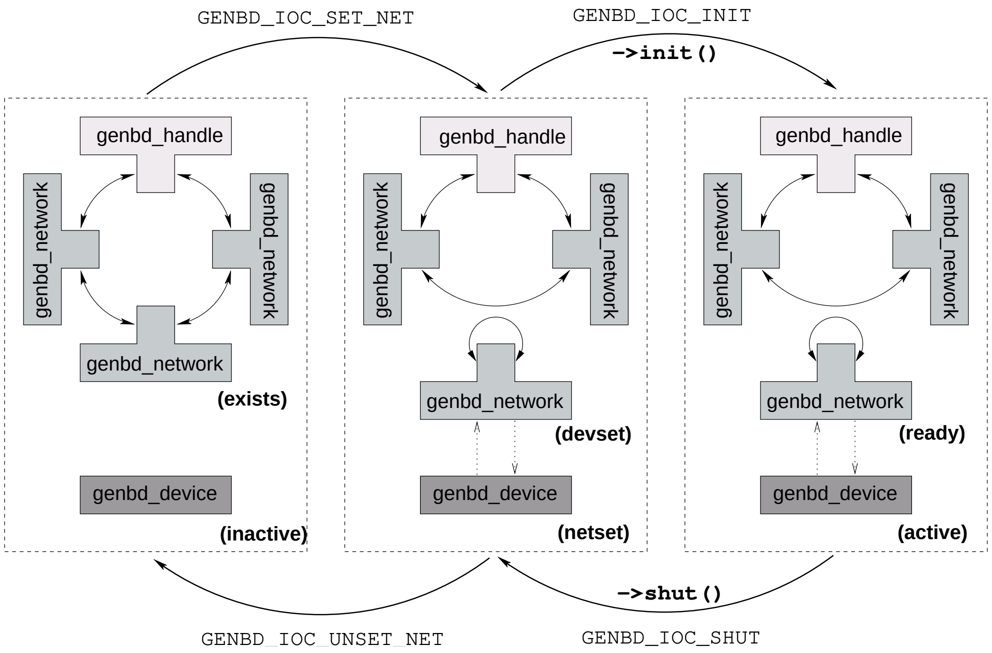
Κάθε συσκευή genbd ξεκινά από την κατάσταση GENBD_DEV_INACTIVE και κάθε genbd_net από την κατάσταση GENBD_NETWORK_EXISTS.

Κατόπιν η συσκευή πρέπει να συσχετιστεί με ένα υπάρχον genbd_net, κάτι που πραγματοποιείται με την κλήση συστήματος ioctl() τύπου GENBD_IOC_SET_NET. Η κλήση αυτή έχει ως όρισμα ένα αναγνωριστικό, ώστε να βρεθεί η κατάλληλη δομή genbd_network από την λίστα που υπάρχει στη δομή genbd_handle. Μόλις πραγματοποιηθεί η διαδικασία αυτή η κατάσταση της συσκευής γίνεται GENBD_DEV_NETSET και του genbd_net GENBD_NETWORK_DEVSET

Η ενεργοποίηση της συσκευής πραγματοποιείται με μια ioctl() κλήση τύπου GENBD_IOC_INIT, όπου και αρχικοποιείται το genbd_net μέσω της init() συνάρτησης που το ίδιο έχει ορίσει. Οι καταστάσεις της συσκευής και του genbd_net γίνονται GENBD_DEV_ACTIVE και GENBD_NETWORK_READY αντίστοιχα.

Οι κλήσεις `ioctl()` `GENBD_IOC_UNSET_NET` και `GENBD_IOC_SHUT` χρησιμοποιούνται για τις αντίθετες λειτουργίες.

Το διάγραμμα καταστάσεων των `genbd` και `genbd_net` παρουσιάζεται στο σχήμα 5.1.



Σχήμα 5.1: Οι αλλαγές στην κατάσταση μιας συσκευής `genbd` *και του* `genbd_net` *που σχετίζεται με αυτή*

5.2 Η διεπαφή μεταξύ genbd και genbd_net

Ένα βασικό κομμάτι της διεπαφής μεταξύ των δύο αυτών στρωμάτων, είναι η δομή `struct genbd_net_operations`, βάσει της οποίας το `genbd` μπορεί να χρησιμοποιήσει την λειτουργικότητα που του παρέχει κάθε `genbd_net`.

Ωστόσο, πριν από αυτό θα πρέπει να υπάρχει κάποιος τρόπος, ώστε το `genbd_net` να ενημερώσει το `genbd` για την ύπαρξή του. Η ενημέρωση αυτή γίνεται με την χρήση της παρακάτω συνάρτησης:

```
int genbd_register_net(unsigned int net_type,
                       unsigned int net_id,
                       const char *net_name,
                       struct genbd_net_operations *net_ops,
                       void *net_handle,
                       void **genbd_handle,
                       genbd_ctl_net_fn_t **genbd_ctl_fn);
```

Στον πίνακα 5.2 εξηγούνται σύντομα τα ορίσματα της.

Η συνάρτηση που χρησιμοποιείται για τον έλεγχο του `genbd` ορίζεται ως:

Όρισμα	Σημασία
net_type	Τύπος του δικτύου που θα χρησιμοποιηθεί.
net_id	Αναγνωριστικό του genbd_net.
net_name	Όνομα του δικτύου που χρησιμοποιείται.
net_ops	Συναρτήσεις που υλοποιούν την κατάλληλη λειτουργικότητα του genbd_net για το genbd.
net_handle	Γενικού τύπου μεταβλητή, η οποία θα χρησιμοποιηθεί σαν όρισμα στις συναρτήσεις net_ops.
genbd_handle	Γενικού τύπου μεταβλητή του genbd που χρησιμοποιείται στην κλήση της συνάρτησης ελέγχου genbd_ctl_fn.
genbd_ctl_fn	Μεταβλητή στην οποία θα τοποθετηθεί ένας δείκτης σε μία συνάρτηση για τον έλεγχο του genbd. Με αυτή τη συνάρτηση το genbd_net μπορεί να ελέγξει παραμέτρους του genbd με έναν γενικό τρόπο.

Πίνακας 5.4: Ορίσματα της συνάρτησης genbd_net_register()

```
typedef int genbd_ctl_net_fn_t(void *genbd_handle,
                               unsigned long cmd,
                               unsigned long arg);
```

Οι παράμετροί της περιλαμβάνουν μια γενικού τύπου μεταβλητή (genbd_handle), της οποίας η τιμή ορίζεται από το αντίστοιχο όρισμα της συνάρτησης genbd_register_net(), την εντολή προς εκτέλεση (cmd) και μια τιμή γενικού τύπου (arg).

Προς το παρών υπάρχει μόνο μία εντολή για τη συνάρτηση ελέγχου, ώστε να μπορεί να θέσει το κάθε genbd_net το μέγεθος της συσκευής αποθήκευσης. Η μεταβλητή cmd παίρνει την τιμή GENBD_CTL_CMD_SET_SIZE_BLOCKS και η μεταβλητή arg το μέγεθος της συσκευής σε blocks.

Για να ειδοποιήσει το genbd_net το genbd ότι δεν είναι πια διαθέσιμο χρησιμοποιείται η συνάρτηση: genbd_unregister_net().

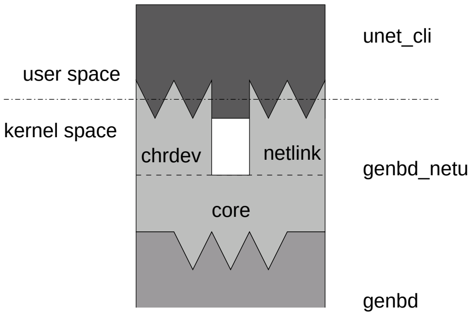
5.3 Το genbd_netu module

Το genbd_netu module υλοποιεί το genbd_net_user στρώμα, το οποίο περιγράφηκε σύντομα στην παράγραφο 4.3. Το genbd_netu αλληλεπιδρά με δύο στρώματα, το genbd και το unet_cli. Για την επικοινωνία με το unet_cli χρησιμοποιούνται δύο διεπαφές: ένα netlink socket και μια εικονική συσκευή χαρακτήρων Έτσι το genbd_netu μπορεί να χωριστεί σε τρία επιμέρους κομμάτια, όπως φαίνεται και στο σχήμα 5.2:

1. **core**: Το κομμάτι αυτό υλοποιεί την βασική λειτουργικότητα, που απαιτείται από ένα genbd_net στρώμα.

2. **netlink**: Το κομμάτι αυτό προσφέρει στο core μια διεπαφή για την χρήση του netlink socket

3. **chrdev**: Το κομμάτι αυτό προσφέρει στο core μια διεπαφή για την χρήση της συσκευής χαρακτήρων.



Σχήμα 5.2: Διαστρωμάτωση του `genbd_netu` *module*

5.3.1 Επικοινωνία με τον χώρο χρήστη

5.3.1.1 Η διεπαφή netlink

Η `genbd_netu` και το `unet_cli` χρησιμοποιούν για την αλληλεπίδραση τους ένα πρωτόκολλο που βασίζεται σε αιτήσεις και απαντήσεις οι οποίες μεταφέρονται μέσα από netlink sockets.

Το netlink είναι μια υλοποίηση μιας socket διεπαφής για την επικοινωνία μεταξύ modules που βρίσκονται στον χώρο πυρήνα και εφαρμογές που βρίσκονται στον χώρο χρήστη. Η πιο διαδεδομένη εφαρμογή που χρησιμοποιεί το netlink είναι το iproute2, το οποίο χρησιμοποιείται για την υποστήριξη προηγμένων τεχνικών δρομολόγησης (routing) και έλεγχου κίνησης (traffic control) στο Linux.

Η δημιουργία ενός netlink socket στον πυρήνα πραγματοποιείται με τη συνάρτηση:

```
struct sock *
netlink_kernel_create(int unit, void (*input)(struct sock *sk, int len))
```

Το πρώτο όρισμα αφορά τον τύπο του netlink socket που θα δημιουργηθεί και το δεύτερο είναι μια συνάρτηση, η οποία θα καλείται κάθε φορά που υπάρχουν εισερχόμενα δεδομένα στο socket.

Η επικεφαλίδα ενός netlink μηνύματος απεικονίζεται στον τύπο:

```
struct nlmsghdr
{
        __u32        nlmsg_len;
        __u16        nlmsg_type;
        __u16        nlmsg_flags;
        __u32        nlmsg_seq;
        __u32        nlmsg_pid;
};
```

Το πεδίο `nlmsg_len` περιέχει το μέγεθος του μηνύματος, το `nlmsg_type` τον τύπο του μηνύματος, το `nlmsg_flags` ορισμένες παραμέτρους που μπορούν να οριστούν, το `nlmsg_seq` έναν αύξοντα αριθμό για τα μηνύματα και το `nlmsg_pid` το pid της διεργασίας που χρησιμοποιεί το socket.

Επιπρόσθετα ορίζεται και ο τύπος `struct nlmsgerr`, ο οποίος χρησιμοποιείται για την επαλήθευση netlink μηνυμάτων:

```
struct nlmsgerr
{
        int error;
        struct nlmsghdr msg;
};
```

Εκτός από τη συνάρτηση που χρησιμοποιείται για την παραλαβή δεδομένων και ορίζεται στην αρχικοποίηση του netlink socket, υπάρχουν και συναρτήσεις για την αποστολή δεδομένων. Οι βασικότερες από αυτές είναι:

- `netlink_broadcast()`: Αποστολή μηνύματος σε ένα σύνολο διεργασιών.
- `netlink_unicast()`: Αποστολή μηνύματος σε μόνο μια διεργασία.
- `netlink_ack()`: Αποστολή μηνύματος επαλήθευσης.

Η διεπαφή μεταξύ του `core` και του `netlink` επιπέδου στο `genbd_netu` σχεδιάστηκε με τέτοιο τρόπο ώστε να μπορεί να αντικατασταθεί ή να προστεθεί νέα λειτουργικότητα, χωρίς να χρειαστούν αλλαγές στο `core` κομμάτι.

- **Λήψη μηνυμάτων:**
  Για την λήψη μηνυμάτων το `core` ορίζει τη συνάρτηση

  ```
  int genbd_netu_ureq(int req_type, void *req_data, void *req_handle);
  ```

  Τα δύο πρώτα ορίσματα περιέχουν τον τύπο του μηνύματος και έναν δείκτη στα δεδομένα του. Το τρίτο όρισμα χρησιμοποιείται σε περίπτωση που το `core` χρειαστεί να στείλει απάντηση σε μία αίτηση. Το όρισμα αυτό είναι απαραίτητο διότι η απάντηση πρέπει να έχει το ίδιο `nlmsg_seq` με το αρχικό μήνυμα.

- **Αποστολή μηνυμάτων:**
  Για την αποστολή μηνυμάτων αρχικά πρέπει να δεσμευτεί χώρος για το μήνυμα και την επικεφαλίδα του με την χρήση των συναρτήσεων:

  ```
  void *genbd_netu_nl_alloc(void **send_handle,int size, int type, int flags);
  void *genbd_netu_nl_alloc_reply(void **send_handle,int size, int flags,
                                  void *req_handle);
  ```

  Η πρώτη συνάρτηση χρησιμοποιείται για την αποστολή αιτήσεων, ενώ η δεύτερη για την αποστολή απαντήσεων και για αυτό χρησιμοποιεί και ένα επιπλέον όρισμα. Το όρισμα αυτό είναι το `req_handle` της συνάρτησης `genbd_netu_ureq` που παρουσιάστηκε παραπάνω.

  Το πρώτο όρισμα των συναρτήσεων αυτών είναι μια θέση για να τοποθετηθεί ένας δείκτης γενικού τύπου. Ο δείκτης αυτός θα χρησιμοποιηθεί κατά την αποστολή του μηνύματος.

  Οι συναρτήσεις αυτές χρησιμοποιούν την `alloc_skb()` για να δεσμεύσουν μνήμη για ένα socket buffer, του οποίου ο τύπος είναι `struct sk_buff`. Το socket buffer είναι η βασική δομή για μεταφορές μέσω sockets. Κατόπιν τοποθετούν τα στοιχεία της επικεφαλίδας και επιστρέφουν τη θέση στην οποία το `core` μπορεί να τοποθετήσει τα δεδομένα του μηνύματος. Ωστόσο, η θέση αυτή δεν αρκεί για να εκτελεστεί η αποστολή αργότερα. Απαραίτητο για την αποστολή είναι το αρχικό socket buffer που αποθηκεύεται στην θέση που ορίζεται από την παράμετρο `send_handle`.

  Αφού τοποθετηθούν τα δεδομένα, το μήνυμα μπορεί να σταλεί με χρήση της συνάρτησης:

```
int genbd_netu_nl_send(void *send_handle);
```

Το πρωτόκολλο επικοινωνίας που χρησιμοποιείται μεταξύ του `genbd_netu` και της εφαρμογής `unet_cli` βασίζεται σε αιτήσεις και απαντήσεις. Αιτήσεις μπορούν να εκδοθούν και από τα δύο άκρα.

Οι αιτήσεις που εκδίδει το `unet_cli` παρουσιάζονται σύντομα στον πίνακα 5.5, ενώ αυτές που εκδίδει το `genbd_netu` στον πίνακα 5.6.

Αίτηση	Σημασία
GENBD_NETU_UREQ_CONF_SET	Αρχικοποίηση του `genbd_unet`.
GENBD_NETU_UREQ_BLK_DONE	Ολοκλήρωση αίτησης block.
GENBD_NETU_UREQ_SIZE_BLOCKS_SET	θέσπιση μεγέθους της συσκευής σε blocks.
GENBD_NETU_UREQ_SHUT	Τερματισμός λειτουργίας.

Πίνακας 5.5: Αιτήσεις που εκδίδει το `unet_cli` *προς το* `genbd_netu`

Αίτηση	Σημασία
GENBD_NETU_REQ_BLK	Αίτηση για την ικανοποίηση μιας λειτουργίας block (εγγραφή ή ανάγνωση).
GENBD_NETU_REQ_SHUT	Τερματισμός λειτουργίας.

Πίνακας 5.6: Αιτήσεις που εκδίδει `genbd_netu` *το προς το* `unet_cli`

5.3.1.2 Συσκευή χαρακτήρων

Οι συσκευές χαρακτήρων είναι ένας αρκετά δημοφιλής και απλός τρόπος για την αλληλεπίδραση του χώρου πυρήνα και του χώρου χρήστη. Ένα module μπορεί να δημιουργήσει μια τέτοια συσκευή και να ορίσει ένα σύνολο συναρτήσεων, οι οποίες θα αντιστοιχούν στις κλήσεις συστήματος που θα χρησιμοποιηθούν κατά την πρόσβαση του αρχείου της συσκευής.

Τυπικό παράδειγμα είναι η χρήση των κλήσεων συστήματος `read` και `write`, όπου εφαρμογή διαβάζοντας από το αρχείο μπορεί να λάβει δεδομένα από τον πυρήνα και γράφοντας στο αρχείο να στείλει δεδομένα στον πυρήνα.

Στη περίπτωση του `genbd_netu` η συσκευή χαρακτήρων χρησιμοποιείται για να απεικονιστούν οι σελίδες που περιέχονται στις μονάδες Ε/Ε `bio` στον εικονικό χώρο διευθύνσεων της εφαρμογής χώρου χρήστη. Η λειτουργία αυτή υλοποιείται μέσω της κλήσης συστήματος `mmap`.

Η κλήση συστήματος `mmap`, όταν σχετίζεται με ένα αρχείο, χρησιμοποιείται για να απεικονιστούν τα περιεχόμενα του αρχείου στην εικονική μνήμη της διαδικασίας. Με αυτόν τον τρόπο η πρόσβαση στο αρχείο μπορεί επιτευχθεί με τυπικές εντολές `LOAD` και `STORE`.

Υλοποιώντας την κλήση `mmap` για μια συσκευή χαρακτήρων το module, μπορεί να απεικονίσει στον χώρο χρήστη σελίδες. Με αυτόν τον τρόπο τα δεδομένα των αιτήσεων block δεν χρειάζεται να αντιγράφονται μεταξύ του χώρου χρήστη και του χώρου πυρήνα.

Για την δημιουργία μιας συσκευής χαρακτήρων χρησιμοποιείται η συνάρτηση:

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops)
```

Η παράμετρος `major` είναι ο αριθμός major της συσκευής και η `name` είναι ένα όνομα της συσκευής, το οποίο χρησιμοποιείται σε διάφορα μέρη του πυρήνα. Η βασική λειτουργικότητα της συσκευής, παρέχεται από τη παράμετρο `fops`. Η παράμετρος αυτή περιέχει ένα σύνολο συναρτήσεων, οι οποίες αντιστοιχούν σε κλήσεις συστήματος που εκτελούνται στο αρχείο της συσκευής.

Το `genbd_netu` χρειάζεται να ορίσει μόνο τις: `open`, `release` και `mmap` που αντιστοιχούν στις κλήσεις συστήματος `open()`, `close()` και `mmap()`.

Η συνάρτηση που πρέπει να υλοποιηθεί για να υποστηριχτεί η κλήση συστήματος `mmap` στο αρχείο της συσκευής είναι τύπου:

```
int (*mmap) (struct file *filep, struct vm_area_struct *vma);
```

Ο τύπος `struct file` είναι ένας γενικός τύπος του πυρήνα, ο οποίος χρησιμοποιείται για να παραστήσει οποιουδήποτε τύπου ανοιχτό αρχείο. Στην συγκεκριμένη περίπτωση το όρισμα `filep` αποτελεί τη δομή `struct file` που σχετίζεται με το αρχείο της συσκευής.

Ο τύπος `struct vm_area_struct` είναι και αυτός ένα γενικός τύπος που χρησιμοποιείται για να παραστήσει μια ομοιογενής περιοχή διευθύνσεων στον εικονικό χώρο της διεργασίας. Κατά την κλήση της `mmap` δημιουργείται μια τέτοια δομή για την περιοχή που θα απεικονιστεί στην εικονική μνήμη της διεργασίας.

Ένα βασικό πεδίο της δομής αυτής είναι το:

```
struct vm_area_struct {
        ...
        struct vm_operations_struct *vm_ops;
        ...
};
```

Η δομή `struct vm_operations_struct` ορίζει ένα σύνολο από συναρτήσεις. Η συνάρτηση που χρησιμοποιείται από το `genbd_netu` είναι η:

```
struct vm_operations_struct {
        ...
        struct page * (*nopage)(
                struct vm_area_struct * area,
                unsigned long address,
                int *type);
        ...
};
```

Η συνάρτηση `nopage` θα κληθεί σε περίπτωση που η διεργασία θα προσπαθήσει να προσπελάσει εικονικές διευθύνσεις στην περιοχή που ορίζει η δομή `vm_area_struct`, χωρίς ωστόσο να υπάρχουν φυσικές σελίδες μνήμης για να ικανοποιηθεί η προσπέλαση. Αν έχει οριστεί η συνάρτηση αυτή και επιστρέψει μια σελίδα, τότε η σελίδα αυτή θα χρησιμοποιηθεί για την προσπέλαση.

Η λειτουργικότητα αυτή χρησιμοποιείται από το `genbd_netu` ώστε να απεικονιστούν οι κατάλληλες σελίδες στον εικονικό χώρο διευθύνσεων του `unet_cli`. Κατά την κλήση της `mmap` το `genbd_netu` θέτει την `nopage` κατάλληλα ώστε να επιλεχτεί η σελίδα που αντιστοιχεί στη συγκεκριμένη θέση που έγινε η πρόσβαση.

5.3.2 Αρχικοποίηση

Η αίτηση τύπου `GENBD_NETU_UREQ_CONF_SET` χρησιμοποιείται για αρχικοποίηση. Συγκεκριμένα το `unet_cli` ενημερώνει το `genbd_netu` ότι είναι έτοιμο προς λειτουργία και του μεταφέρει τις κατάλληλες παραμέτρους ώστε να μπορέσει να επικοινωνήσει με το `genbd`.

Τα δεδομένα της αίτησης είναι:

```
struct genbd_netu_ureq_conf {
        unsigned int    net_type;
        unsigned int    net_id;
        char            net_name[GENBD_NET_NAMESIZE];
};
```

Όταν το `genbd_netu` δεχτεί την αίτηση αυτή δεσμεύει μνήμη για τη βασική δομή που χρησιμοποιεί και καλεί την `genbd_register_net()`.

Η δομή αυτή είναι η :

```
struct genbd_netu_dev {
        void *genbd_handle;
        genbd_ctl_net_fn_t *genbd_ctl;

        unsigned int status;

        spinlock_t lock;

        struct bio *bio_pend, *bio_pend_tail;
        unsigned int bio_pend_nr;

        struct bio **bio_r, **bio_r_tail;
        struct bio **bio_w, **bio_w_tail;
        unsigned int bio_rw_nr;

        struct workqueue_struct *genbd_workqueue;
        struct work_struct work_pending;

        unsigned int chrdev_users;
};
```

Για να είναι δυνατή η υποστήριξη πολλαπλών εφαρμογών `unet_cli` στέλνεται ως απάντηση ένα αναγνωριστικό τύπου `unsigned long`. Το αναγνωριστικό αυτό χρησιμοποιείται στις επόμενες αιτήσεις του `unet_cli` ώστε να μπορεί το `genbd_netu` να ξεχωρίσει τα διάφορα `unet_cli`. Το κάθε αναγνωριστικό αυτό αντιστοιχεί σε μια μεταβλητή τύπου `struct genbd_netu_dev`.

Η εφαρμογή θα πρέπει να θέσει το μέγεθος της συσκευής. Η λειτουργία αυτή γίνεται μέσω μιας αίτησης τύπου `GENBD_NETU_UREQ_SIZE_BLOCKS_SET`. Τα δεδομένα της αίτησης απεικονίζονται στον τύπο:

```
struct genbd_netu_ureq_size_blocks {
        unsigned long   dev_handle;
        unsigned long   size_blocks;
};
```

Όπου `dev_handle` είναι το αναγνωριστικό που αναφέρθηκε πριν και `size_blocks` το μέγεθος σε blocks.

5.3.3 Βασική λειτουργία

5.3.3.1 Η συνάρτηση genbd_netu_handle_bio

Η συνάρτηση `handle_bio` που υλοποιεί το `genbd_netu` για τον χειρισμό των `bio` που δέχεται το `genbd` ονομάζεται `genbd_netu_handle_bio` και απλά τοποθετεί τα `bio` σε μια ουρά και αναβάλλει την επεξεργασία τους για αργότερα. Με αυτόν τον τρόπο η συνάρτηση `handle_bio` έχει μικρή καθυστέρηση μειώνοντας τον χρόνο απόκρισης της συσκευής.

Για την μεταθετημένη χρονικά επεξεργασία των αιτήσεων χρησιμοποιείται ο μηχανισμός των `workqueues`. Ο μηχανισμός αυτός υλοποιεί μια ουρά, η οποία περιέχει μια σειρά εργασιών προς εκτέλεση. Η εκτέλεση των εργασιών αυτών πραγματοποιείται μέσω ενός η περισσοτέρων νημάτων εκτέλεσης χώρου πυρήνα (kernel threads).

Η δημιουργία μιας ουράς μπορεί να γίνει με τις συναρτήσεις:

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

Η πρώτη συνάρτηση δημιουργεί ένα kernel thread για κάθε επεξεργαστή του συστήματος, ενώ η δεύτερη δημιουργεί μόνο ένα. Η κάθε εργασία προς εκτέλεση απεικονίζεται σε μία δομή `struct work_struct` η οποία μπορεί να αρχικοποιηθεί με την βοήθεια των μακροεντολών:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

Η `DECLARE_WORK` χρησιμοποιείται για την αρχικοποίηση στον χρόνο μεταγλώττισης (compile time), ενώ η `INIT_WORK` για αρχικοποίηση στο χρόνο εκτέλεσης (run time). Απαραίτητος είναι ο ορισμός μιας συνάρτησης προς εκτέλεση και μιας παραμέτρου γενικού τύπου με την οποία θα κληθεί η συνάρτηση αυτή.

Για την εισαγωγή μιας εργασίας σε μία ουρά χρησιμοποιείται η συνάρτηση:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
```

Η ουρά και η εργασία που χρησιμοποιούνται δημιουργούνται μετά από μια αίτηση τύπου `GENBD_NETU_UREQ_CONF_SET` και τοποθετούνται στα πεδία `genbd_workqueue` και `work_pending` της μεταβλητής τύπου `struct genbd_netu_dev` κατά την αρχικοποίηση της:

```
static void genbd_netu_dev_init(struct genbd_netu_dev *dev)
{
        ...
        dev->genbd_workqueue = create_singlethread_workqueue("genbdnetud");
        INIT_WORK(&dev->work_pending, genbd_netu_work,dev);
        ...
}
```

Τα μέλη `bio_pend`, `bio_pend_tail` και `bio_pend_nr` χρησιμοποιούνται για την υλοποίηση της ουράς για την τοποθέτηση των αιτήσεων που δεν έχουν επεξεργαστεί ακόμα. Τα `bio_pend` και `bio_pend_tail` είναι η αρχή και το τέλος μιας λίστας από `bio` και το `bio_pend_nr` ο αριθμός των στοιχείων της λίστας. Για τον χειρισμό της ουράς αυτής έχουν οριστεί οι συναρτήσεις:

```
void enqueue_pend_bio(struct genbd_netu_dev *dev, struct bio *bio);
struct bio *dequeue_pend_bio(struct genbd_netu_dev *dev);
```

Η συνάρτηση `genbd_netu_handle_bio`, λοιπόν, υλοποιείται ως εξής:

```
static int genbd_netu_handle_bio(void *net_handle, struct bio *bio)
{

        struct genbd_netu_dev *dev;

        dev = (struct genbd_netu_dev *)net_handle;

        enqueue_pend_bio(dev,bio);

        queue_work(dev->genbd_workqueue, &dev->work_pending);

        return 0;
}
```

5.3.3.2 Επεξεργασία των αιτήσεων

Η συνάρτηση, η οποία αναλαμβάνει την επεξεργασία των μονάδων `bio` είναι η:

```
static void genbd_netu_work(void *data)
{
        struct genbd_netu_dev *dev;
        struct bio *bio;

        dev = (struct genbd_netu_dev *)data;

        for (;;){

                bio = dequeue_pend_bio(dev);
                if (!bio) break;

                do_genbd_netu_work(dev,bio);
        }
}
```

Η συνάρτηση αυτή απλώς εξάγει ένα `bio` από την ουρά και χρησιμοποιεί την `do_genbd_netu_work` για την επεξεργασία του.

Η συνάρτηση αυτή πρέπει να επεξεργαστεί κατάλληλα το `bio` και να εκδώσει μια αίτηση τύπου `GENBD_NETU_REQ_BLK` προς την εφαρμογή του χώρου χρήστη.

Η αίτηση αυτή έχει δύο μέρη:

- Το πρώτο σχετίζεται με την λειτουργία block που πρέπει να πραγματοποιηθεί και αναπαρίσταται από την δομή:

  ```
  struct genbd_netu_req_blk {
          unsigned int    type;
          unsigned long   start;
          unsigned long   len;
          unsigned long   handle;
  };
  ```

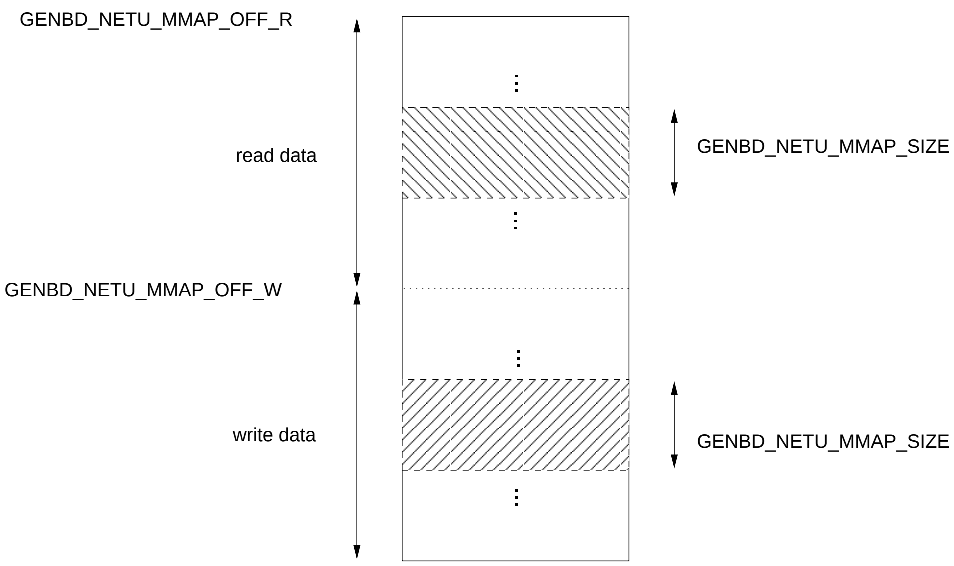
  Το πεδίο `type` αφορά τον τύπο της αίτησης (εγγραφή ή ανάγνωση), τα πεδία `start` και `len` την αρχή και το μέγεθος του τμήματος του χώρου αποθήκευσης με το οποίο σχετίζεται η αίτηση. Τέλος το `handle` είναι ένα μοναδικό αναγνωριστικό της αίτησης αυτής.

- Το δεύτερο σχετίζεται με το τις περιοχές του αρχείου της συσκευής χαρακτήρων που πρέπει να απεικονιστούν στην εικονική μνήμη της εφαρμογής `unet_cli` για να έχει πρόσβαση στον χώρο των δεδομένων block. Η δομή που χρησιμοποιείται για αυτήν την πληροφορία είναι ένας πίνακας από:

```
struct genbd_netu_mmap {
        unsigned long offset;
        unsigned long length;
};
```

  Στο `offset` τοποθετείται η θέση του τμήματος στο αρχείο της συσκευής χαρακτήρων και στο `length` το μήκος του. Το μέγεθος πίνακα μπορεί να βρεθεί εύκολα από το `nlmsg_len` πεδίο του μηνύματος.

Για την απεικόνιση των δεδομένων των διαφόρων αιτήσεων στην συσκευή χαρακτήρων χρησιμοποιείται μια στατική μέθοδος. Ο χώρος της συσκευής χωρίζεται σε σταθερά τμήματα, τα οποία έχουν μήκος το μέγιστο δυνατό μέγεθος δεδομένων ενός `bio` (`GENBD_NETU_MMAP_SIZE`). Το πρώτο μισό του διαθέσιμου χώρου χρησιμοποιείται για αιτήσεις ανάγνωσης, ενώ το δεύτερο μισό για αιτήσεις εγγραφής.



Σχήμα 5.3: Ο χωρισμός του χώρου στη συσκευή χαρακτήρων που χρησιμοποιεί το `genbd_netu`

Χρησιμοποιούνται `GENBD_NETU_MMAP_HASH_SIZE` τμήματα για τις αναγνώσεις και αλλά τόσα για τις εγγραφές. Κάθε `bio` αντιστοιχίζεται σε ένα τμήμα βάσει της συνάρτησης κατακερματισμού (hash function):

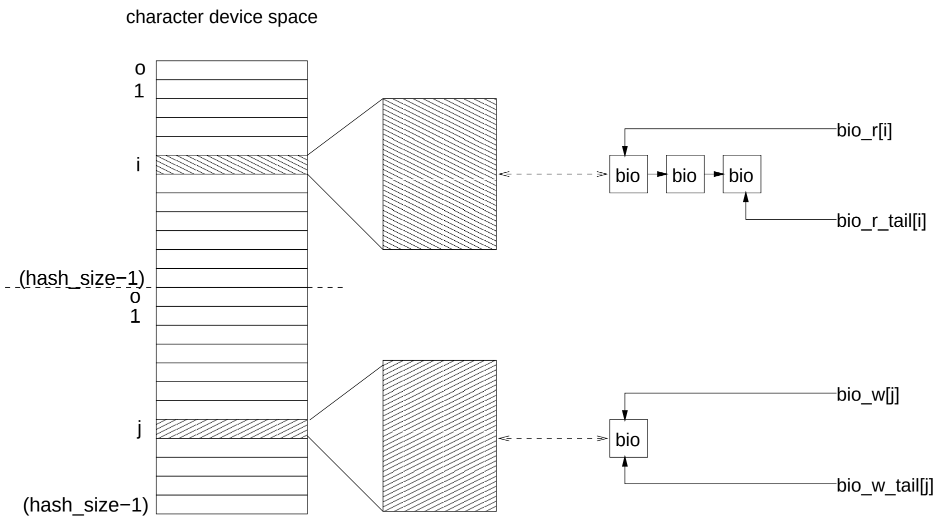
```
int genbd_netu_bio_hashval(struct bio *b)
```

Η συνάρτηση αυτή χρησιμοποιεί την συνάρτηση `hash_ptr()` του linux (βλ. [Lev00]) και τη διεύθυνση του `bio` για να υπολογίσει την ζητούμενη τιμή.

Βέβαια παρουσιάζεται πρόβλημα σε περίπτωση που η θέση ενός νέου `bio` είναι ήδη κατειλημμένη από ένα άλλο. Για την αντιμετώπιση του προβλήματος αυτού δημιουργείται μια ουρά για κάθε τμήμα στον χώρο του αρχείου της συσκευής χαρακτήρων. Το πρώτο στοιχείο της ουράς είναι αυτό για το οποίο ο αντίστοιχος χώρος στο αρχείο της συσκευής χαρακτήρων έχει δεσμευθεί.

Οι ουρές αυτές υλοποιούνται με τη βοήθεια των πεδίων `bio_r`, `bio_r_tail` και `bio_w`, `bio_w_tail` που είναι τύπου `struct bio **`.

Για παράδειγμα η ουρά του τμήματος ανάγνωσης `i` μπορεί να προσπελαστεί από τα `->bio_r[i]` και `->bio_r_tail[i]`. Αντίστοιχα η ουρά του τμήματος εγγραφής `j` μπορεί να προσπελαστεί από τα `->bio_w[j]` και `->bio_w_tail[j]` όπως φαίνεται και στο σχήμα: 5.4



Σχήμα 5.4: Χρήση ουρών για αποθήκευση των μονάδων Ε/Ε `bio`

Η λειτουργία της `do_genbd_netu_work()` έχει ως εξής:

1. Εύρεση του τμήματος, της ουράς και της θέσης στο αρχείο της συσκευής χαρακτήρων που αντιστοιχούν στο προς επεξεργασία `bio`.
2. Τοποθέτηση του `bio` στην κατάλληλη ουρά.
3. Αν η ουρά ήταν άδεια κλήση της `forward_bio_user()` για να σταλεί η κατάλληλη αίτηση στην εφαρμογή στον χώρο χρήστη.

Η `forward_bio_user()` χρησιμοποιεί την `genbd_netu_nl_alloc()` για να δεσμεύσει χώρο για το μήνυμα, τις `fill_req_blk()` και `fill_mmaps()` για να τοποθετήσει τα κατάλληλα δεδομένα στον χώρο αυτό και την `genbd_netu_nl_send()` για να στείλει το μήνυμα.

Για κάθε σελίδα που περιέχεται στο `bio` η `fill_mmaps()` δημιουργεί και ένα `genbd_netu_mmap` στοιχείο, στο οποίο προσαρμόζονται κατάλληλα τα πεδία `offset` και `length`. Η προσαρμογή αυτή γίνεται βάσει των πεδίων `bv_offset` και `bv_len` του `bio_vec` που περιέχει την σελίδα.

Συγκεκριμένα η `fill_mmaps()` ορίζεται ως εξής:

```
static void
fill_mmaps(struct bio *bio, struct genbd_netu_mmap *mmaps, unsigned long off)
{
        int i;
```

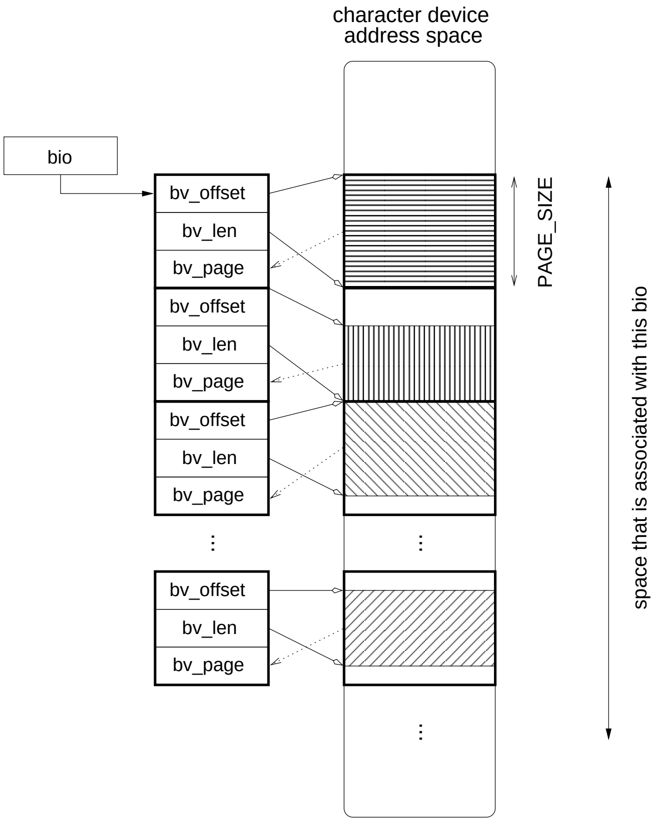
```
        struct bio_vec *bvec;

        bio_for_each_segment(bvec, bio, i){
                mmaps[i].offset = start_off + bvec->bv_offset + i*PAGE_SIZE;
                mmaps[i].length = bvec->bv_len;
        }
}
```

Όπου `off` είναι η θέση της αρχής του τμήματος του αρχείου της συσκευής, στο οποίο αντιστοιχεί το συγκεκριμένο `bio`.

5.3.3.3 Απεικόνιση των δεδομένων των αιτήσεων

Συνεπώς η απεικόνιση των δεδομένων ενός `bio` στον χώρο γίνεται όπως παρουσιάζεται στο σχήμα 5.5



Σχήμα 5.5: Απεικόνιση ενός `bio` *στον χώρο της συσκευή χαρακτήρων που χρησιμοποιεί το* `genbd_netu`

Ο στατικός αυτός τρόπος απεικόνισης των δεδομένων έχει το μειονέκτημα ότι πολύς χώρος της συσκευής είναι πιθανό να μην χρησιμοποιείται καθόλου. Ωστόσο διευκολύνει πολύ τον υπολογισμό της σελίδας που αντιστοιχεί σε μια θέση του αρχείου της συσκευής χαρακτήρων.

Η λειτουργία αυτή είναι απαραίτητη κατά την υλοποίηση της μεθόδου `nopage` που χρησιμοποιείται για την απεικόνιση των κατάλληλων σελίδων στον εικονικό χώρο της διεργασίας `unet_cli`.

Η συνάρτηση που υλοποιεί την `nopage` στα τμήματα εικονικής μνήμης (`struct vm_area_struct`) είναι η `genbd_netu_nopage()`. Η συνάρτηση αρχικά υπολογίζει την θέση στον χώρο του αρχείου της συσκευής στον οποίο μετά από μια κλήση συστήματος `mmap` έγινε η προσπέλαση και κατόπιν καλεί την `page_from_offset()`, η οποία και υπολογίζει την σελίδα που αντιστοιχεί σε αυτή τη θέση.

5.3.3.4 Ολοκλήρωση των αιτήσεων

Η εφαρμογή `unet_cli` ενημερώνει το `genbd_net` module για την ολοκλήρωση (επιτυχή ή όχι) μιας αίτησης block με ένα μήνυμα τύπου `GENBD_NETU_UREQ_BLK_DONE`. Το μήνυμα αυτό περιέχει τα εξής δεδομένα:

```
struct genbd_netu_ureq_blk_done {
        unsigned long dev_handle;
        unsigned long blkreq_handle;
        unsigned int  bytes_done;
        int           error;
};
```

Το πεδίο `dev_handle` είναι το αναγνωριστικό της εφαρμογής `unet_cli` και το `blkreq_handle` το αναγνωριστικό της αίτησης `GENBD_NETU_REQ_BLK` που ολοκληρώθηκε. Το πεδίο `bytes_done` περιέχει τον αριθμό των bytes, τα οποία αποθηκεύτηκαν ή ανακτήθηκαν χωρίς σφάλμα και το πεδίο `error` χρησιμοποιείται για να ενημερώσει το `genbd_netu` για τυχόν παρουσία σφάλματος.

Η συνάρτηση που είναι υπεύθυνη για την επεξεργασία των αιτήσεων αυτών είναι η `genbd_netu_bio_endio()`. Η συνάρτηση αυτή λειτουργεί ως εξής:

1. Εύρεση του `bio` που αντιστοιχεί στην αίτηση αυτή.
2. Εύρεση της ουράς στην οποία έχει τοποθετηθεί και εξαγωγή του από αυτήν.
3. Κλήση της `bio_endio()` για τον τερματισμό του `bio` αυτού.
4. Σε περίπτωση που η ουρά περιείχε και άλλο `bio` κλήση της `forward_bio_user()` ώστε να σταλεί η κατάλληλη αίτηση στην εφαρμογή `unet_cli`.

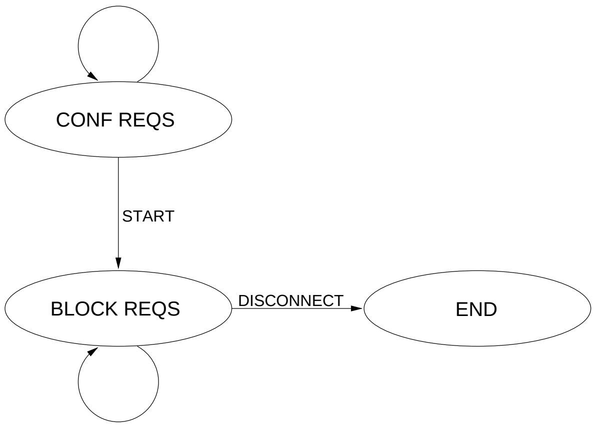
5.4 Οι εφαρμογές unet_cli και unet_srv

5.4.1 Πρωτόκολλο επικοινωνίας

Οι εφαρμογές `unet_cli` και `unet_srv` είναι αυτές που υλοποιούν το πρωτόκολλο επικοινωνίας, το οποίο επιτρέπει την υλοποίηση της απομακρυσμένης αποθήκευσης και ανάκτησης δεδομένων. Το πρωτόκολλο αυτό βασίζεται σε αιτήσεις που εκδίδει το `unet_cli` και απαντήσεις που επιστρέφει το `unet_srv`.

Υπάρχουν δύο ειδών αιτήσεων: οι αιτήσεις συντονισμού (configure requests) και οι αιτήσεις για λειτουργίες block (block requests). Μετά την πραγματοποίηση της σύνδεσης μεταξύ των δύο κόμβων ανταλλάσσονται αιτήσεις συντονισμού.

Η `UNET_CONF_REQ_CMD_START` είναι η τελευταία αίτηση συντονισμού που εκδίδεται. Μετά από αυτήν ανταλλάσσονται αιτήσεις block, έως ότου εκδοθεί μια αίτηση `UNET_REQ_TYPE_DISCONNECT`, η οποία και τερματίζει την επικοινωνία



Σχήμα 5.6: Διάγραμμα καταστάσεων του πρωτοκόλλου μεταξύ του `unet_srv` *και του* `unet_cli`

5.4.1.1 Αιτήσεις συντονισμού

Οι αιτήσεις συντονισμού είναι της μορφής:

```
struct unet_conf_req {
        unsigned int cmd;
        unsigned long arg;
};
```

Οι αιτήσεις αυτές αποτελούνται από μια παράμετρο (πεδίο `cmd`) και από μια τιμή για αυτήν την παράμετρο (πεδίο `val`). Ο εξυπηρετητής (`unet_srv`) επιστρέφει μια απάντηση ίδιου τύπου με την αίτηση, η οποία περιέχει την τιμή που τελικά έθεσε στην παράμετρο.

Οι παράμετροι που υποστηρίζονται είναι οι εξής:

- `UNET_CONF_REQ_CMD_BLOCK_SIZE`:
  Η παράμετρος αυτή είναι το μέγεθος του block που θα χρησιμοποιηθεί στο πρωτόκολλο και ορίζεται από το `unet_cli`.

- `UNET_CONF_REQ_CMD_SIZE_BLOCKS`:
  Η παράμετρος αυτή είναι το μέγεθος του χώρου, που προσφέρει ο εξυπηρετητής και τίθεται από αυτόν.

- `UNET_CONF_REQ_CMD_START`:
  Η παράμετρος αυτή απλώς σηματοδοτεί την αρχή της κανονικής λειτουργίας του πρωτοκόλλου όπου χρησιμοποιούνται αιτήσεις block.

5.4.1.2 Αιτήσεις block

Η έκδοση και η ικανοποίηση αιτήσεων block αποτελεί και τη βάση της απομακρυσμένης αποθήκευσης. Για λόγους ευκολίας για μια αίτηση block χρησιμοποιείται ο τύπος `struct genbd_netu_req_blk` που παρουσιάστηκε στην παράγραφο 5.3.3.2.

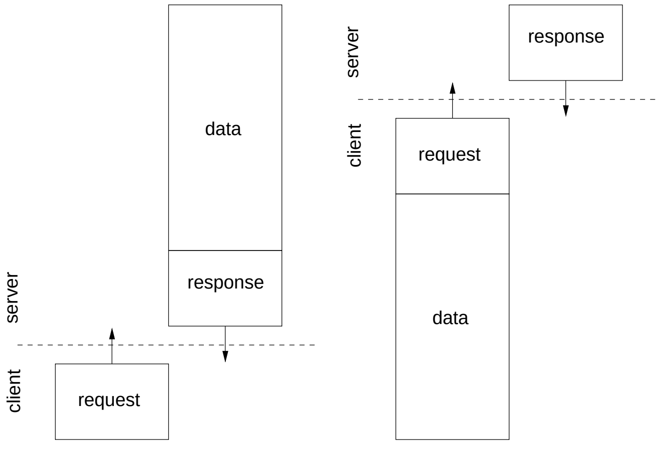
Αντίστοιχα η απάντηση στην αίτηση αυτή είναι τύπου:

```
struct unet_srv_res {
        int           error;
        unsigned long handle;
        unsigned long bytes_done;
};
```

Όπου το πεδίο `error` ενημερώνει για την παρουσία σφάλματος κατά την προσπάθεια ικανοποίησης της αίτησης, το `handle` είναι το αναγνωριστικό της αίτησης και το `bytes_done` περιέχει τον αριθμό των bytes τα οποία αποθηκεύτηκαν ή ανακτήθηκαν χωρίς πρόβλημα. Σε περίπτωση που δεν υπήρξε σφάλμα το πεδίο `error` έχει την τιμή `UNET_RES_ERR_NONE`.

Οι αιτήσεις ανάγνωσης είναι τύπου `UNET_REQ_TYPE_READ` ενώ, οι αιτήσεις εγγραφής είναι τύπου `UNET_REQ_TYPE_WRITE`.

Οι απαντήσεις ανάγνωσης και οι αιτήσεις εγγραφής ακολουθούνται από τα αντίστοιχα δεδομένα που σχετίζονται με την αίτηση, όπως φαίνεται και στο σχήμα 5.7.



Σχήμα 5.7: Αιτήσεις block και απαντήσεις του πρωτοκόλλου μεταξύ των `unet_cli` και `unet_srv`

e Το πρωτόκολλο επικοινωνίας τερματίζεται όταν ο πελάτης εκδόσει μια αίτηση τύπου `UNET_REQ_TYPE_DISCONNECT`. Σε περίπτωση που ο εξυπηρετητής θέλει να τερματίσει για κάποιον λόγο την επικοινωνία μπορεί να στείλει μια απάντηση, στην οποία το πεδίο `error` να έχει την τιμή `UNET_RES_ERR_DISCONNECT`.

5.4.2 Η διεπαφή του unet_ll

Για την μεταφορά δεδομένων χρησιμοποιείται μια απλή διεπαφή, η οποία είναι σχεδιασμένη για την υποστήριξη πολλαπλών δικτύων. Η διεπαφή χρησιμοποιεί ένα σύνολο συναρτήσεων για την αρχικοποίηση, τον τερματισμό και την αποστολή και την λήψη δεδομένων. Οι συναρτήσεις αυτές υλοποιούνται διαφορετικά για κάθε τύπο δικτύου.

Οι τύποι των συναρτήσεων αυτών είναι:

```
typedef void *unet_ll_init_fn_t(char *conf);
typedef int unet_ll_shut_fn_t(void *handle);
```

```
typedef int unet_ll_send_fn_t(void *handle, void *buf, unsigned int count);
typedef int unet_ll_recv_fn_t(void *handle, void *buf, unsigned int count);
```

Η βασική δομή του `unet_ll` είναι η:

```
struct unet_ll {
        void                *ll_handle;
        void                *dl_handle;
        unet_ll_send_fn_t   *send;
        unet_ll_recv_fn_t   *recv;
        unet_ll_init_fn_t   *init;
        unet_ll_shut_fn_t   *shut;
};
```

Το πεδίο `ll_handle` περιέχει κάποια δεδομένα γενικού τύπου για χρήση από την κάθε υλοποίηση. Η τιμή του πεδίου αυτού είναι η τιμή που επιστρέφει η συνάρτηση `init()` και συνοδεύει όλες τις υπόλοιπες κλήσεις των διάφορων συναρτήσεων.

Η συνάρτηση `init()` έχει ως σκοπό την αρχικοποίηση του δικτύου και την εγκατάσταση μιας σύνδεσης μεταξύ δύο κόμβων. Το όρισμά της είναι ένα όρισμα γενικού τύπου, το οποίο χρησιμοποιείται για να περάσει παραμέτρους σχετικά με την επικείμενη σύνδεση στην κάθε υλοποίηση. Στις υλοποιήσεις που έγιναν σε αυτή την εργασία το όρισμα αυτό είναι μια συμβολοσειρά της μορφής:

```
par1=val1 par2=val2 ...  parn=valn
```

Οι παράμετροι που χρειάζεται η κάθε υλοποίηση παρουσιάζονται στον πίνακα 5.7.

inet_cli	
server	IP του εξυπηρετητή
port	TCP port στο οποίο θα γίνει η σύνδεση.
inet_srv	
port	TCP port στο οποίο θα γίνει η σύνδεση.
gmnet	
lport_id	Αναγνωριστικό τοπικού GM port.
rport_id	Αναγνωριστικό απομακρυσμένου GM port.
unit	Αναγνωριστικό συσκευής.
rhost	Απομακρυσμένος κόμβος.
scinet	
rnode_id	Αναγνωριστικό απομακρυσμένου κόμβου.
adap_no	Αναγωνσιστικό συσκευής.

Πίνακας 5.7: Παράμετροι που χρησιμοποιούνται στην συνάρτηση αρχικοποίησης των διάφορων υλοποιήσεων του `unet_ll`.

Το πεδίο `dl_handle` περιέχει έναν διαφανή περιγραφητή που σχετίζεται με το δυναμικό φόρτωμα μιας υλοποίησης του `unet_ll` και δεν χρησιμοποιείται από τις διάφορες εφαρμογές.

Οι συναρτήσεις `send()` και `recv()` χρησιμοποιούνται για την αποστολή και την λήψη δεδομένων αντίστοιχα.

Τέλος η συνάρτηση `shut()` χρησιμοποιείται για τον τερματισμό της σύνδεσης μεταξύ των δύο κόμβων.

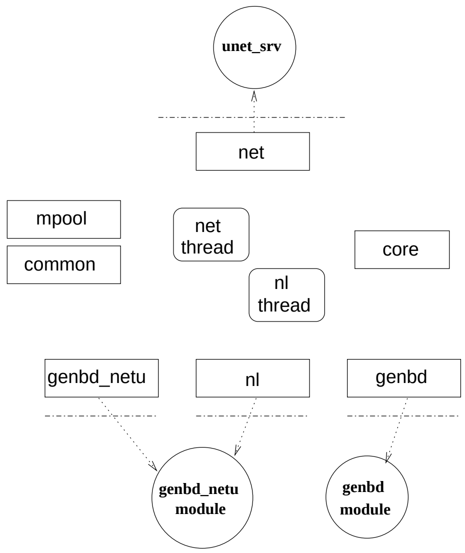
Το δυναμικό φόρτωμα και η εκφόρτωση ενός `unet_ll` πραγματοποιείται με τις συναρτήσεις:

```
struct unet_ll *unet_ll_init(char *libname, char *fprefix);
void unet_ll_shut(struct unet_ll *);
```

Όπου η μεταβλητή `libname` περιέχει το όνομα του αρχείου της βιβλιοθήκης και η `fprefix` μια καθορισμένη συμβολοσειρά για κάθε υλοποίηση.

5.4.3 Το unet_cli

5.4.3.1 Βασική δομή του unet_cli



Σχήμα 5.8: Τμήματα από τα οποία αποτελείται το `unet_cli`

Όπως παρουσιάζεται στο σχήμα 5.8 Το `unet_cli` αποτελείται από διάφορα τμήματα, το καθένα από τα οποία έχει διαφορετικές λειτουργίες:

- **core**: Το κεντρικό κομμάτι του `unet_cli`, το οποίο είναι υπεύθυνο κυρίως για την αρχικοποίηση.
- **net**: Το κομμάτι που είναι υπεύθυνο για την χρήση του `unet_ll` και την επικοινωνία με τον απομακρυσμένο κόμβο.
- **nl**: Το κομμάτι που είναι υπεύθυνο για την επικοινωνία με το module `genbd_netu` μέσω της διεπαφής netlink.
- **genbd_netu**: Το κομμάτι που είναι υπεύθυνο για την επικοινωνία με το module `genbd_netu` μέσω του αρχείου της συσκευής χαρακτήρων.
- **common**: Μερικές συναρτήσεις γενικού σκοπού.
- **mpool**: Το κομμάτι αυτό βοηθάει στην δέσμευση μνήμης για ορισμένα δεδομένα, τα οποία χρησιμοποιούνται συχνά. Αντί κάθε φορά να χρησιμοποιούνται κλήσεις `malloc()` και `free()` για την δέσμευση και αποδέσμευση μνήμης τα δεδομένα δεσμεύονται στην αρχή και τοποθετούνται σε μια λίστα. Η εφαρμογή χρησιμοποιεί τις λίστες αυτές για να δεσμεύσει και να αποδεσμεύσει τα δεδομένα.

Για να είναι δυνατή η ασύγχρονη επικοινωνία μεταξύ του εξυπηρετητή και του πελάτη χρησιμοποιούνται δύο νήματα εκτέλεσης (threads). Το πρώτο thread (`nl`) χρησιμοποιείται για την επεξεργασία αιτήσεων από τον `genbd_netu`, ενώ το δεύτερο (`net`) για την επεξεργασία απαντήσεων από τον εξυπηρετητή.

5.4.3.2 Αρχικοποίηση

Η αρχικοποίηση που κάνει η εφαρμογή `unet_cli`, είναι αρκετά πολύπλοκη αφού απαιτείται ο συντονισμός πολλών διαφορετικών επιπέδων. Οι λειτουργίες που επιτελούνται είναι οι εξής:

1. Αρχικοποίηση του `unet_ll`:
   (a) Φόρτωμα της βιβλιοθήκης μέσω της συνάρτησης `unet_ll_init()`.
   (b) Κλήση της `->init()` συνάρτησης για να εγκατασταθεί η σύνδεση μεταξύ των δύο κόμβων.
2. Αρχικοποίηση του netlink socket.
3. Αποστολή `GENBD_NETU_UREQ_CONF_SET` αίτησης στο `genbd_netu`.
4. `ioctl()` κλήση τύπου `GENBD_IOC_SET_NET` στο αρχείο της συσκευής block.
5. Άνοιγμα της συσκευής χαρακτήρων του `genbd_netu`.
6. Συντονισμός με τον εξυπηρετητή:
   (a) ορισμός του μεγέθους του block από τον πελάτη
   (b) ορισμός του μεγέθους του χώρου αποθήκευσης από τον εξυπηρετητή
   (c) έκδοση `UNET_CONF_REQ_CMD_START` αίτησης.
7. Έκδοση `GENBD_NETU_UREQ_SIZE_BLOCKS_SET` αίτησης στο `genbd_netu` για τον ορισμό του μεγέθους της συσκευής block.
8. `ioctl()` κλήση τύπου `GENBD_IOC_INIT` στο αρχείο της συσκευής block.
9. Δημιουργία των δύο νημάτων εκτέλεσης.
   Η υποστήριξη πολυνηματικών (multithreading) λειτουργιών στην εφαρμογή `unet_cli` πραγματοποιείται με την βοήθεια των Posix Threads (pthreads). Τα pthreads στις τελευταίες εκδόσεις του linux υλοποιούνται από την βιβλιοθήκη NPTL (Native Posix Thread Library) (βλ. και [DM03].
   Νέα νήματα εκτέλεσης δημιουργούνται μέσω της `pthread_create()` συνάρτησης.

5.4.3.3 Δομές δεδομένων του unet_cli

Ο βασικός τύπος που χρησιμοποιεί το `unet_cli` είναι ο:

```
struct unet_cli_handle {
        unsigned int block_size;
        unsigned long size_blocks;

        unsigned int net_type;
        unsigned int net_id;
        char net_name[GENBD_NET_NAMESIZE];

        int genbd_fd;
```

```
        int genbd_netu_fd;

        int nl_sock;
        struct sockaddr_nl nl_local;
        __u32 nl_seq;
        pthread_t nl_thread;

        struct unet_ll *ll;
        pthread_t ll_thread;

        struct blist_node send_list;
        pthread_spinlock_t send_list_lock;

        unsigned long genbd_net_handle;

        struct blist_node mpool_cli_reqs;
        struct unet_cli_mmap *mpool_cli_mmaps;
};
```

Για κάθε αίτηση που παραλαμβάνεται από το `genbd_netu` δημιουργείται η δομή:

```
struct unet_cli_blkreq {
        struct genbd_netu_req_blk kreq;

        void *mmap_addr;
        struct unet_cli_mmap *mmap_data;
        off_t mmap_len;

        unsigned int bytes_done;
        int error;

        struct blist_node req_list;
};
```

Η δομή `struct unet_cli_mmap` ορίζεται ως εξής:

```
struct unet_cli_mmap {
        struct genbd_netu_mmap map;
        struct unet_cli_mmap *next;
};
```

Για τις δύο παραπάνω δομές χρησιμοποιούνται συναρτήσεις `mpool` για την δέσμευση και αποδέσμευση μνήμης.

5.4.3.4 Λειτουργία νήματος nl

Η συνάρτηση η οποία εκτελείται κατά την λειτουργία του νήματος `nl` είναι η `unet_cli_nl_read_loop()`, η οποία είναι ένας ατέρμον βρόγχος. Διαβάζει δεδομένα από το netlink socket, το οποίου ο περιγραφητής είναι αποθηκευμένος στο πεδίο `nl_sock` του `struct unet_cli_handle` τύπου και χρησιμοποιεί την συνάρτηση `unet_cli_nl_handle_msg()` για την επεξεργασία των αιτήσεων.

Η συνάρτηση `unet_cli_nl_handle_msg()` είναι αρκετά απλή και ο μόνος ρόλος της είναι να προωθήσει την αίτηση στην κατάλληλη συνάρτηση ανάλογα με τον τύπο της. Η συνάρτηση που χρησιμοποιείται για την επεξεργασία των αιτήσεων τύπου `GENBD_NETU_REQ_BLK` είναι η `unet_cli_nl_handle_blkreq()`.

Η συνάρτηση `unet_cli_nl_handle_blkreq()` εκτελεί τις παρακάτω λειτουργίες:

1. Χρησιμοποιεί την συνάρτηση `unet_cli_mpool_req_get()` για την δέσμευση χώρου για μία μεταβλητή `req` τύπου `struct unet_cli_blkreq`

2. Αντιγράφει τα δεδομένα της αίτησης που έλαβε από το `genbd_netu` στο πεδίο `kreq` της `req`.

3. Χρησιμοποιεί την `unet_cli_mmap_data_fill()` για να δημιουργήσει τη λίστα με τα δεδομένα της απεικόνισης, η οποία τοποθετείται στο πεδίο `mmap_data` της `req`.

   Η συνάρτηση αυτή δημιουργεί μια λίστα από περιοχές του αρχείου της συσκευής χαρακτήρων συγχωνεύοντας τις περιοχές που στάλθηκαν από το `genbd_netu`. Οι ανάγκες για νέες δομές τύπου `struct unet_cli_mmap` ικανοποιούνται από την συνάρτηση `unet_cli_mpool_mmap_get()`.

4. Τοποθετεί την `req` σε μια λίστα, η οποία περιέχει τις αιτήσεις οι οποίες έχουν σταλεί στον εξυπηρετητή καλώντας τη συνάρτηση `unet_cli_sendlist_add()`. Η λίστα αυτή υλοποιείται με τη βοήθεια του πεδίου `req_list`.

   Βέβαια επειδή αυτή η λίστα χρησιμοποιείται και από τα δύο νήματα θα πρέπει να προστατευθεί από ταυτόχρονη προσπέλασή. Ο συγχρονισμός επιτυγχάνεται με την βοήθεια μιας μεταβλητής τύπου `pthread_spinlock_t`. Οι εντολές προσπέλασης στη λίστα τοποθετούνται ανάμεσα στις συναρτήσεις `pthread_spin_lock()` και `pthread_spin_unlock()`, ώστε να σειριοποιηθεί η πρόσβαση σε αυτή.

5. Στέλνει την κατάλληλη αίτηση στον εξυπηρετητή χρησιμοποιώντας την συνάρτηση `unet_cli_net_send_req()`. Η συνάρτηση αυτή απλώς δημιουργεί μια αίτηση τύπου `struct genbd_netu_req_blk` την οποία και αποστέλλει στον εξυπηρετητή μέσω της βιβλιοθήκης `unet_ll`.

6. Σε περίπτωση που η αίτηση αφορά εγγραφή δεδομένων στον απομακρυσμένο αποθηκευτικό χώρο θα πρέπει να σταλούν και τα δεδομένα:

   - Αρχικά τα δεδομένα απεικονίζονται στον εικονικό χώρο της διεργασίας μέσω της συνάρτησης `unet_cli_genbd_netu_mmap()`. Η συνάρτηση αυτή χρησιμοποιεί την κλήση συστήματος `mmap` και θέτει το πεδίο `mmap_addr` της `req` στην διεύθυνση, στην οποία αρχίζει ο χώρος των δεδομένων.
   - Κατόπιν τα δεδομένα, που έχουν πλέον απεικονιστεί στον εικονικό χώρο της διεργασίας, στέλνονται στον εξυπηρετητή μέσω της `unet_cli_net_send_blkdata()`.
   - Τέλος αφαιρείται η απεικόνιση των δεδομένων με την βοήθεια της συνάρτησης `unet_cli_genbd_netu_munmap()`, η οποία χρησιμοποιεί την κλήση συστήματος `munmap()`.

5.4.3.5 Λειτουργία νήματος net

Η συνάρτηση, η οποία αναλαμβάνει την επεξεργασία των απαντήσεων του εξυπηρετητή είναι η `unet_cli_net_read_loop()`. Για κάθε τέτοια απάντηση που λαμβάνει τα βήματα που ακολουθεί είναι τα εξής:

1. Σε περίπτωση που η αίτηση αφορά ανάγνωση:

- Καλείται η συνάρτηση `unet_cli_genbd_netu_mmap()` για να απεικονιστεί ο χώρος αποθήκευσης των δεδομένων στον χώρο διευθύνσεων της διεργασίας.
- Τα δεδομένα αφού ανακτηθούν από το δίκτυο τοποθετούνται με κατάλληλο τρόπο στον χώρο αυτό.
- Η συνάρτηση `unet_cli_genbd_netu_munmap()` αφαιρεί την απεικόνιση που δημιουργήθηκε

2. Η αίτηση, την οποία αφορά η απάντηση αυτή αφαιρείται από την λίστα με τις αιτήσεις που έχουν σταλεί στον εξυπηρετητή μέσω της συνάρτησης `unet_cli_sendlist_cut()`.
3. Ακολουθεί η αποστολή ενός μηνύματος `GENBD_NETU_UREQ_BLK_DONE` στο `genbd_netu`, ώστε να ενημερωθεί για την ολοκλήρωση της αίτησης. Για τον σκοπό αυτό χρησιμοποιείται η συνάρτηση `unet_cli_nl_sendreq_done()`.
4. Αποδεσμεύονται τα στοιχεία τύπου `struct unet_cli_mmap` και `struct unet_cli_blkreq` που είχαν δεσμευτεί.

   Η λειτουργία αυτή πραγματοποιείται από την `unet_cli_blkreq_free()`, η οποία καλεί τις `unet_cli_mpool_mmap_put()` και `unet_cli_mpool_req_put()`.

5.4.4 To unet_srv

Η εφαρμογή `unet_srv` είναι αρκετά απλή και δεν θα αναλυθεί διεξοδικά. Χρησιμοποιεί δύο βασικές συναρτήσεις:

1. `unet_srv_confloop()`
   Η συνάρτηση αυτή αναλαμβάνει την ικανοποίηση των αιτήσεων συντονισμού του `unet_cli`. Μόλις εκδοθεί η αίτηση `UNET_CONF_REQ_CMD_START` η συνάρτηση αυτή επιστρέφει.
2. `unet_srv_mainloop()`
   Η συνάρτηση αυτή αναλαμβάνει την επεξεργασία των αιτήσεων block. Για την ικανοποίηση τους χρησιμοποιείται ένα αρχείο το οποίο προσφέρει και τον αποθηκευτικό χώρο.

   Οι αιτήσεις ανάγνωσης ικανοποιούνται από την `unet_srv_handle_rreq()` και οι αιτήσεις εγγραφής από την `unet_srv_handle_wreq()`. Σε περίπτωση που σταλεί αίτηση τύπου `UNET_REQ_TYPE_DISCONNECT` η συνάρτηση τερματίζεται.

Οι αιτήσεις ικανοποιούνται σε κάποιο τοπικό αρχείο, που ορίζεται σαν παράμετρος του προγράμματος, ως εξής:

- Αρχικά με την βοήθεια της συνάρτησης `lseek()` ορίζεται κατάλληλα η θέση του αρχείου, στην οποία πρέπει θα επιτελεστεί η λειτουργία που ορίζεται από την αίτηση.
- Κατόπιν με κλήση των συναρτήσεων `read()` ή `write()` ανακτώνται ή αποθηκεύονται κατάλληλα τα δεδομένα.

5.5 Η υλοποίηση του unet_ll

Το `unet_ll` υλοποιείται σαν ένα σύνολο δυναμικών βιβλιοθηκών. Κάθε υλοποίησή του θα πρέπει να δημιουργήσει ένα σύνολο από συναρτήσεις, οι οποίες θα περιέχονται σε μια τέτοια βιβλιοθήκη. Κατόπιν η συνάρτηση `unet_ll_do_init()` φροντίζει για το δυναμικό φόρτωμα τους χρησιμοποιώντας τις συναρτήσεις `dlopen()` και `dlsym()` και την τοποθέτηση τους σε μία `struct unet_ll` δομή. Η μακροεντολή `unet_ll_init()` απλά δίνει μια πιο απλή διεπαφή στην `unet_ll_do_init()`. Αντίστοιχα η συνάρτηση `unet_ll_shut()` χρησιμοποιείται για την αντίθετη διαδικασία.

Ακολουθεί μια ανάλυση για τις διάφορες υλοποιήσεις της `unet_ll` που αναπτύχθηκαν κατά την εργασία αυτή

5.5.1 inet

Η υλοποίηση αυτή αφορά TCP/IP δίκτυα, των οποίων ο προγραμματισμός γίνεται μέσω της διεπαφής των sockets. (βλ. παράγραφο 3.1.3).

Επειδή τα δύο άκρα μιας TCP/IP σύνδεσης έχουν διαφορετικούς ρόλους κατά την αρχικοποίηση αυτής δημιουργήθηκαν δύο διαφορετικά σύνολα συναρτήσεων, ένα για τον εξυπηρετητή (server) και ένα για τον πελάτη (client).

Οι συναρτήσεις που υλοποιήθηκαν για τον εξυπηρετητή είναι οι:

- `inet_srv_init()`
- `inet_srv_shut()`
- `inet_srv_send()`
- `inet_srv_recv()`

Αντίστοιχα για τον πελάτη οι συναρτήσεις είναι οι:

- `inet_cli_init()`
- `inet_cli_shut()`
- `inet_cli_send()`
- `inet_cli_recv()`

Οι συναρτήσεις λήψης και αποστολής δεδομένων υλοποιούνται με την χρήση κλήσεων συστήματος `read()` και `write()` για την μεταφορά δεδομένων από τους χώρους μνήμης στον απομακρυσμένο κόμβο.

5.5.2 gmnet

Το `gmnet` είναι μια απλή υλοποίηση του `unet_ll` που χρησιμοποιεί το Myrinet δίκτυο και την διεπαφή GM (βλ. παράγραφο 3.3.7).

Το `gmnet` χρησιμοποιεί ένα ξεχωριστό νήμα εκτέλεσης για την επεξεργασία των γεγονότων που λαμβάνονται από το GM. Το νήμα αυτό ονομάζεται `worker`. Η χρήση του ξεχωριστού νήματος εκτέλεσης είναι απαραίτητη διότι το GM λειτουργεί ασύγχρονα. Βέβαια, η προσπάθεια απεικόνισης της πιο εξελιγμένης και με μεγαλύτερες δυνατότητες διεπαφής του GM στην απλή διεπαφή του `unet_ll` έχει και επιπτώσεις στην επίδοση, οι οποίες οφείλονται κατά μεγάλο βαθμό στο επιπλέον αυτό νήμα εκτέλεσης.

Οι συναρτήσεις του GM δεν μπορούν να εκτελεστούν με ασφαλή τρόπο από διαφορετικά νήματα και συνεπώς χρησιμοποιείται μια μεταβλητή τύπου `pthread_mutex_t` ώστε να σειριοποιηθεί η εκτέλεσή τους. Επιπρόσθετα η μεταβλητή αυτή χρησιμοποιείται και για την προστασία κοινών πόρων μεταξύ των διαφορετικών νημάτων εκτέλεσης που χρησιμοποιούνται.

Η βασική δομή που χρησιμοποιείται στο `gmnet` είναι η:

```
struct gmnet_handle {
        struct gm_port *gm_port;
        gm_u32_t gm_r_nid;
        gm_u32_t gm_r_pid;
        gm_u32_t gm_l_pid;

        pthread_mutex_t lock;

        bq_buf_t *recv_bufs;
        bq_buf_t *send_bufs;

        int send_tokens;

        struct blist_node recv_filled;
        unsigned int recv_filled_nr;

        struct blist_node send_free;
        unsigned int send_free_nr;

        struct blist_node send_ready;
        unsigned int send_ready_nr;
        unsigned int send_ready_rem;

        pthread_cond_t cond_recv;
        pthread_cond_t cond_send;

        pthread_t worker;
};
```

5.5.2.1 Προσωρινοί χώροι αποθήκευσης

Το `gmnet` χρησιμοποιεί ένα σύνολο από προσωρινούς χώρους αποθήκευσης (buffers) για τις αποστολές και για τις λήψεις δεδομένων. Οι χώροι αυτοί αναπαρίστανται μέσω του τύπου:

```
struct bq_buffer {
        struct blist_node list;
        char *data;
        unsigned long len;
        unsigned long off;
        unsigned long private_data;
};

typedef struct bq_buffer bq_buf_t;
```

Τα πεδία του τύπου αυτού παρουσιάζονται σύντομα στον πίνακα 5.8

Κατά την αρχικοποίηση δεσμεύεται χώρος μνήμης για ένα σύνολο μεταβλητών τέτοιου τύπου, ο οποίος είναι προσβάσιμος από τα πεδία `recv_bufs` και `send_bufs` του `gmnet_handle`, που αντιστοιχούν στους χώρους για την αποστολή και την λήψη δεδομένων. Ο αριθμός των χώρων αυτών είναι `GMNET_SENDBUF_NR` και `GMNET_RECVBUF_NR` αντίστοιχα.

Ο χώρος μνήμης που χρησιμοποιείται για την αποθήκευση δεδομένων και είναι προσβάσιμος από το πεδίο `data` του ενός `bq_buf_t` θα πρέπει να πληρεί τις κατάλληλες προϋποθέσεις, ώστε να

Πεδίο	Σημασία
list	Πεδίο για την δημιουργία διπλά συνδεδεμένων λιστών.
data	Δείκτης στη διεύθυνση στην του προσωρινού χώρου αποθήκευσης.
len	Μήκος των αποθηκευμένων δεδομένων.
off	Τρέχουσα θέση στον χώρο αποθήκευσης.
private_data	Πεδίο για γενική χρήση.

Πίνακας 5.8: Σημασία των πεδίων του τύπου `bq_buf_t`

μπορεί να χρησιμοποιηθεί από το GM. Για αυτόν τον λόγο η δέσμευση μνήμης για αυτόν γίνεται με την συνάρτηση `gm_dma_calloc()`. Ο χώρος που δεσμεύεται έχει μέγεθος `GMNET_BUF_LEN`.

Οι λειτουργίες αρχικοποίησης και δέσμευσης μνήμης για τα παραπάνω δεδομένα πραγματοποιούνται μέσω της συνάρτησης `gmnet_alloc_bufs()` η οποία καλεί τις `gmnet_alloc_bufs_send()` και `gmnet_alloc_bufs_recv()`. Οι συναρτήσεις αυτές εκτελούν και ορισμένες άλλες λειτουργίες, οι οποίες παρουσιάζονται παρακάτω:

Οι μεταβλητές `bq_buf_t` αποστολής που δημιουργήθηκαν τοποθετούνται σε μία λίστα, η οποία είναι προσβάσιμη από το πεδίο `send_free` του `gmnet_handle`. Η λίστα αυτή περιέχει τους χώρους, στους οποίους δεν έχουν ακόμα τοποθετηθεί δεδομένα προς αποστολή. Ο αριθμός των στοιχείων της είναι αποθηκευμένος στο πεδίο `send_free_nr` του `gmnet_handle`. Τα πεδία `len` και `off` του `bq_buf_t` αναπαριστούν το μέγεθος του διαθέσιμο χώρου και την αρχή του, οπότε αρχικοποιούνται με τις τιμές `GMNET_BUF_LEN` και 0 αντίστοιχα. Τέλος στο πεδίο `private_data` τοποθετείται η διεύθυνση της μεταβλητής `gmnet_handle`.

Το πεδίο `private_data` του κάθε `bq_buf_t` λήψης τίθεται ίσο με τον αύξοντα αριθμό του. Οι χώροι αποθήκευσης για τα δεδομένα λήψης παραδίδονται στο GM, μέσω της συνάρτησης `gm_provide_receive_buffer_with_tag()`, ώστε να μπορέσει να τους χρησιμοποιήσει. Ως ετικέτα (`tag`) ορίζεται ο αύξων αριθμός του κάθε χώρου αποθήκευσης.

5.5.2.2 Αποστολή δεδομένων

Εκτός από την λίστα με τους άδειους χώρους αποστολής, υπάρχει και μια λίστα η οποία περιέχει χώρους αποστολής οι οποίοι αν και έχουν δεδομένα δεν έχουν σταλεί ακόμα. Τα πεδία του `gmnet_handle` που σχετίζονται με τη λίστα αυτή είναι το `send_ready`, στο οποίο τοποθετούνται τα στοιχεία της, το `send_ready_nr`, στο οποίο αποθηκεύεται ο αριθμός των στοιχείων της λίστας και το `send_ready_rem`, στο οποίο αποθηκεύεται το μέγεθος του ελεύθερου χώρου στο τελευταίο στοιχείο. Επίσης το πεδίο `send_tokens` περιέχει τον αριθμό των tokens αποστολής, τα οποία είναι διαθέσιμα στην εφαρμογή.

Η συνάρτηση που υλοποιεί την αποστολή δεδομένων είναι η `gmnet_send()`. Η συνάρτηση αυτή ακολουθεί τα εξής βήματα:

1. Αν υπάρχουν στοιχεία στη λίστα `send_ready` και αν το τελευταίο της στοιχείο έχει ελεύθερο χώρο τα δεδομένα αντιγράφονται σε αυτό.

2. Αν υπάρχουν ακόμα δεδομένα προς αποστολή, αντιγράφονται σε χώρους αποθήκευσης από την λίστα `send_free`. και τοποθετούνται στην λίστα `send_ready`. Αν η λίστα `send_free` δεν έχει στοιχεία η λειτουργία της συνάρτησης αναστέλλεται έως ότου τοποθετηθούν νέα στοιχεία Η διαδικασία αυτή υλοποιείται μέσω του πεδίου `cond_send` του `gmnet_handle`.

   Το πεδίο αυτό είναι τύπου `pthread_cond_t` και χρησιμοποιείται από συναρτήσεις όπως η `pthread_cond_wait()` και η `pthread_cond_signal()`. Οι συναρτήσεις αυτές είναι μέρος της βιβλιοθήκης `pthread` και προσφέρουν έναν μηχανισμό για την αναστολή της λειτουργίας ενός νήματος εκτέλεσης έως ότου συμβεί κάποιο προκαθορισμένο γεγονός.

3. Καλείται η συνάρτηση `gmnet_send_bufs_try()` για να στείλει τα δεδομένα που βρίσκονται στην λίστα `send_ready`. Η συνάρτηση αυτή θα στείλει τα δεδομένα των στοιχείων της λίστας μόνο αν τα tokens αποστολής δεν είναι λιγότερα από ένα συγκεκριμένο κατώφλι (threshold). Σε αντίθετη περίπτωση τα στοιχεία παραμένουν στη λίστα.

   Το πρόβλημα που προσπαθεί να αντιμετωπίσει η μέθοδος αυτή είναι ότι σε περίπτωση που γίνει μεγάλος αριθμός αποστολών, με λίγα δεδομένα οι ελεύθεροι χώροι αποθήκευσης τελειώνουν γρήγορα και πιθανώς η εφαρμογή να χρειαστεί να περιμένει.

   Για την αποστολή των δεδομένων χρησιμοποιείται η `gmnet_buf_send()`, η οποία μειώνει τον αριθμό των διαθέσιμων tokens, όπως αυτά απεικονίζονται στο πεδίο `send_tokens`, κατά ένα και καλεί την συνάρτηση `gm_send_with_callback()` με τα κατάλληλα ορίσματα.

   Η συνάρτηση callback που χρησιμοποιείται είναι η `gmnet_gm_send_cb()`, η οποία ως όρισμα δέχεται την μεταβλητή τύπου `bq_buf_t` από την οποία στάλθηκαν τα δεδομένα. Το όρισμα αυτό βοηθάει να αναγνωρίσει η συνάρτηση το συγκεκριμένο προσωρινό χώρο αποθήκευσης, ο οποίος είναι πλέον διαθέσιμος. Παράλληλα το πεδίο `private_data` αυτού περιέχει την μεταβλητή τύπου `struct gmnet_handle`, στην οποία και περιέχονται όλα τα δεδομένα που χρειάζονται.

   Τα βήματα που ακολουθεί η συνάρτηση callback είναι τα εξής

   (a) Αύξηση του πεδίου `send_tokens` κατά ένα.

   (b) Τοποθέτηση της `bq_buf_t` μεταβλητής στην λίστα `send_free`.

   (c) Αποστολή δεδομένων ενός στοιχείου της λίστας `send_ready` σε περίπτωση που αυτή δεν είναι άδεια.

   Το τελευταίο βήμα είναι απαραίτητο ώστε να μην υπάρχει περίπτωση να μείνουν δεδομένα στην λίστα `send_ready` που να μην σταλούν ποτέ. Θα πρέπει επίσης να σημειωθεί πως η συνάρτηση callback καλείται από το νήμα εκτέλεσης `worker`, το οποίο αναλαμβάνει την επεξεργασία των γεγονότων του GM.

5.5.2.3 Λήψη δεδομένων

Η λήψη δεδομένων, όπως και η αποστολή βασίζεται σε ένα μοντέλο παραγωγού - καταναλωτή (producer - consumer). ο καταναλωτής είναι η συνάρτηση `gmnet_recv()` η οποία και υλοποιεί την συνάρτηση λήψης του gmnet και ο παραγωγός είναι το νήμα εκτέλεσης που είναι υπεύθυνο για τον χειρισμό των γεγονότων που προκύπτουν από το GM. Τα πεδία `recv_filled` και `recv_filled_nr` χρησιμοποιούνται για την δημιουργία μιας λίστας για δεδομένα που έχουν σταλεί από τον απομακρυσμένο κόμβο.

Αρχικά όλοι οι χώροι αποθήκευσης των δεδομένων λήψης παραδίδονται στο GM. Στην περίπτωση που σταλούν δεδομένα από τον απομακρυσμένο κόμβο δημιουργούνται και τα αντίστοιχα γεγονότα λήψης, τα οποία τα χειρίζεται η συνάρτηση `gmnet_handle_gm_event_recv()`.

Η συνάρτηση αυτή αρχικά βρίσκει τη μεταβλητή τύπου `bq_buf_t` που σχετίζεται με τον χώρο στον οποίο έχουν αποθηκευτεί τα δεδομένα, χρησιμοποιώντας την ετικέτα (tag). Κατόπιν θέτει τα πεδία `len` και `off` της μεταβλητής αυτής ίσα με το μήκος των δεδομένων και μηδέν αντίστοιχα. Τέλος ο χώρος αυτός τοποθετείται στο τέλος της λίστας `recv_filled`.

Η συνάρτηση η οποία χρησιμοποιείται από την εφαρμογή για την λήψη δεδομένων είναι η `gmnet_recv()`. Η λειτουργία της συνάρτησης αυτής έχει ως εξής:

1. Αποκοπή του πρώτου στοιχείου από την λίστα `recv_filled`. Σε περίπτωση που η λίστα δεν έχει στοιχεία, αναστέλλεται η λειτουργία της συνάρτησης με την βοήθεια του πεδίου `cond_recv` της δομής `struct gmnet_handle`. Η συνάρτηση θα συνεχίσει την λειτουργία της μόλις εισαχθούν νέα στοιχεία στη λίστα.
2. Αντιγραφή των δεδομένων του στοιχείου αυτού στην κατάλληλη θέση μνήμης.
3. αναπροσαρμογή του πεδίου `off` του στοιχείου, ώστε να είναι ίσο με το μέγεθος των δεδομένων που έχουν διαβαστεί από αυτό.
4. Αν το πεδίο `off` είναι ίσο με το πεδίο `len`, έχουν διαβαστεί όλα τα δεδομένα που περιέχει το στοιχείο αυτό και συνεπώς ο σχετικός χώρος επιστρέφεται στο GM μέσω της συνάρτησης `gm_provide_receive_buffer_with_tag()`. Σε αντίθετη περίπτωση τοποθετείται στην αρχή της λίστας.

5.5.3 scinet

Η τελευταία υλοποίηση του `unet_ll` αφορά το δίκτυο SCI και την διεπαφή `SISCI` (βλ. 3.4.5).

5.5.3.1 Γενικά

Οι μηχανισμοί που χρησιμοποιούνται στο `scinet` είναι αντίστοιχοι με αυτούς που χρησιμοποιούνται στο `gmnet`. Συγκεκριμένα και σε αυτήν την περίπτωση χρησιμοποιούνται προσωρινοί χώροι αποθήκευσης, οι οποίοι περιγράφονται από μια δομή `bq_buf_t`. Ωστόσο, οι χώροι αυτοί χρησιμοποιούνται μόνο για τα δεδομένα λήψης και όχι και για τα δεδομένα αποστολής, όπως στο `gmnet`. ο αριθμός των προσωρινών χώρων αποθήκευσης είναι `SCINET_BUF_NR` και το μέγεθός τους `SCINET_BUF_LEN`.

Επιπρόσθετα εκτός από τον τύπο `bq_buf_t` χρησιμοποιείται και ο τύπος `bq_buf_q_t` ο οποίος αναπαριστά μια ουρά από χώρους αποθήκευσης. Οι συναρτήσεις που χρησιμοποιούνται από το `scinet` για τον χειρισμό τέτοιων ουρών παρουσιάζονται στον πίνακα 5.9.

`bq_buffer_queue_init()`	Αρχικοποίηση της ουράς
`bq_buffer_queue_add_tail()`	Προσθήκη στοιχείου στο τέλος της ουράς
`bq_buffer_queue_add_head()`	Προσθήκη στοιχείου στην αρχή της ουράς
`bq_buffer_queue_cut_head_wait()`	Αποκοπή στοιχείου από την αρχή της ουράς. Σε περίπτωση που η ουρά είναι άδεια η συνάρτηση αναστέλλει την λειτουργία της έως ότου τοποθετηθούν νέα στοιχεία

Πίνακας 5.9: Συναρτήσεις για τον χειρισμό ουρών τύπου `bq_buf_q_t`

Οι συναρτήσεις για των χειρισμό των ουρών αυτών έχουν σχεδιαστεί ώστε να μπορούν να κληθούν από πολλαπλά νήματα εκτέλεσης μιας διεργασίας, βασιζόμενες σε μηχανισμούς που παρέχονται από την βιβλιοθήκη pthread.

Όλα τα δεδομένα που χρειάζονται για την υλοποίηση του `scinet` έχουν τοποθετηθεί στον τύπο:

```
struct scinet_handle {

    sci_desc_t sci_dev;
    unsigned int sci_adapter_no;
    unsigned int sci_r_nid;

    sci_local_segment_t sci_lseg;
```

```
        sci_remote_segment_t sci_rseg;
        sci_map_t sci_lseg_map;
        sci_map_t sci_rseg_map;

        sci_local_interrupt_t sci_lint;
        sci_local_interrupt_t sci_lint_ack;
        sci_remote_interrupt_t sci_rint;
        sci_remote_interrupt_t sci_rint_ack;

        bq_buf_t *bufs;
        bq_buf_q_t bufs_ready;
        bq_buf_q_t bufs_empty;

        volatile char *lseg_data;
        volatile char *rseg_data;

        atomic_t connections;
        atomic_t status;
};
```

Τα πεδίο `bufs` αποτελεί έναν πίνακα, στον οποίο αποθηκεύονται οι περιγραφητές των χώρων αποθήκευσης που χρησιμοποιούνται. το πεδίο `bufs_empty`, αποτελεί μια ουρά για τους χώρους αποθήκευσης που δεν έχουν δεδομένα, ενώ το `bufs_ready` αυτούς που έχουν διαθέσιμα δεδομένα.

Για την μεταφορά δεδομένων μεταξύ των δύο κόμβων χρησιμοποιείται ένα τοπικό τμήμα μνήμης για κάθε κόμβο το οποίο εξάγεται στον χώρο του SCI. Με αυτόν τον τρόπο κάθε κόμβος έχει ένα τοπικό και ένα απομακρυσμένο τμήμα μνήμης.

Το τοπικό τμήμα μνήμης περιγράφεται από το πεδίο `sci_lseg` του τύπου `scinet_handle`. Τα πεδία `sci_lseg_map` και `lseg_data` αποτελούν τον περιγραφητή της απεικόνισης και την εικονική διεύθυνση μνήμης στην οποία έχει απεικονιστεί το τμήμα αυτό. Το μέγεθος του τοπικού τμήματος (και κατά συνέπεια του απομακρυσμένου) για κάθε κόμβο είναι `SCINET_SCIBUF_LEN`

Αντίστοιχα τα πεδία `sci_rseg`, `sci_rseg_map` και `rseg_data` χρησιμοποιούνται για το απομακρυσμένο τμήμα.

Απαραίτητο στοιχείο για την επικοινωνία των δύο κόμβων είναι η χρήση διακοπών. Στον τύπο `scinet_handle` ορίζονται δύο τοπικές διακοπές: η `sci_lint` και η `sci_lint_ack`. Ο χειρισμός της πρώτη διακοπής γίνεται ασύγχρονα μέσω μιας συνάρτησης callback ενώ της δεύτερης σύγχρονα με την χρήση της συνάρτησης `SCIWaitForInterrupt()`. Για τις αντίστοιχες απομακρυσμένες διακοπές χρησιμοποιούνται τα πεδία `sci_rint` και sci_rint_ack.

5.5.3.2 Αποστολή δεδομένων

Η αποστολή δεδομένων επιτυγχάνεται με την βοήθεια της συνάρτησης `scinet_send()`. Η συνάρτηση αυτή ακολουθεί τα παρακάτω βήματα:

1. Τοποθετεί στην αρχή του απομακρυσμένου τμήματος μνήμης πρώτα το μέγεθος των δεδομένων και μετά τα δεδομένα.
2. Ενεργοποιεί την απομακρυσμένη διακοπή `sci_rint`.
3. Περιμένει σύγχρονα την ενεργοποίηση τοπικής διακοπής `sci_lint_ack` από τον απομακρυσμένο κόμβο. Η ενεργοποίηση της διακοπής αυτής σημαίνει ότι το απομακρυσμένο τμήμα μπορεί να χρησιμοποιηθεί και πάλι για την μεταφορά δεδομένων.

5.5.3.3 Λήψη δεδομένων

Για την λήψη δεδομένων από την εφαρμογή χρησιμοποιείται η συνάρτηση `scinet_recv()`. Ωστόσο βασικό ρόλο παίζει και η συνάρτηση callback του τοπικού τμήματος `sci_lint`. Η συνάρτηση αυτή είναι η `scinet_lint_cb()`

Η `scinet_lint_cb()` εκτελεί τις παρακάτω λειτουργίες:

1. Ανάκτηση του μεγέθους των δεδομένων που αναγράφηκαν, το οποίο έχει αποθηκευτεί στην αρχή του τμήματος.
2. Κλήση της `bq_buffer_queue_cut_head_wait()` στην ουρά `bufs_empty` για την απόκτηση ενός άδειου χώρου αποθήκευσης.
3. Αντιγραφή των δεδομένων στον παραπάνω χώρο.
4. Εισαγωγή του στην ουρά `bufs_ready`.
5. Ενεργοποίηση της απομακρυσμένης διακοπής `sci_rint_ack`.

Η `scinet_recv()` με τη σειρά της ακολουθεί τα παρακάτω βήματα:

1. χρήση της `bq_buffer_queue_cut_head_wait()` στην ουρά `bufs_ready` για την απόκτηση ενός χώρου αποθήκευσης στον οποίο περιέχονται δεδομένα.
2. Αντιγραφή των δεδομένων του στην θέση μνήμης που ορίστηκε ως παράμετρος.
3. Τοποθέτηση του παραπάνω χώρου είτε στην αρχή της ουράς `bufs_ready` είτε στην ουρά `bufs_empty` ανάλογα με το αν υπάρχουν ή όχι και άλλα διαθέσιμα δεδομένα.

Κεφάλαιο 6

Επίλογος

6.1 Περίληψη

Στην εργασία αυτή αναπτύχθηκε μια δικτυακή συσκευή block, η οποία επιτρέπει σε ένα υπολογιστικό σύστημα (πελάτης) να χρησιμοποιήσει με διαφανή τρόπο αποθηκευτικό χώρο ενός άλλου συστήματος (εξυπηρετητής). Η συσκευή σχεδιάστηκε σε πολλαπλά στρώματα ώστε να μπορεί να επεκταθεί η λειτουργικότητά της.

Για την πρώτη υλοποίηση επιλέχτηκε ο προγραμματισμός του δικτύου, το οποίο αναλαμβάνει τις μεταφορές των δεδομένων μεταξύ των κόμβων, να γίνει στον χώρο χρήστη. Το βασικό μειονέκτημα της επιλογής αυτής είναι ότι εισάγεται επιπλέον καθυστέρηση στην ικανοποίηση των αιτήσεων block.

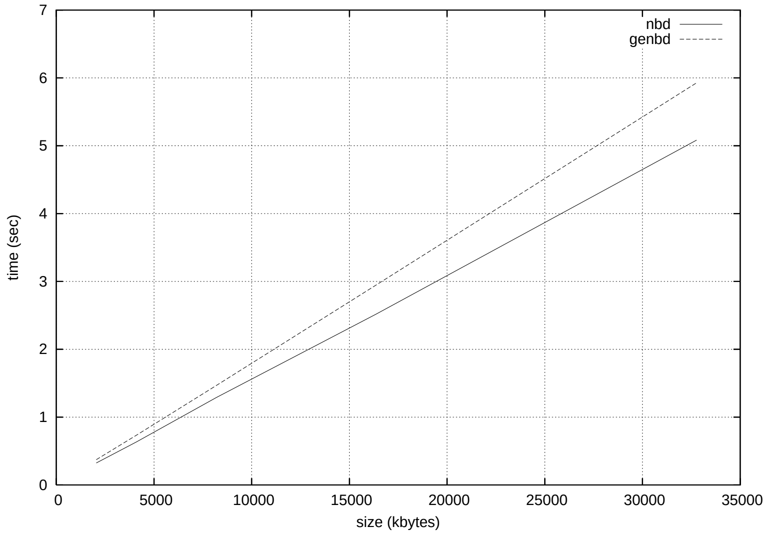
Συνεπώς, ήταν απαραίτητη η ανάπτυξη ενός αποδοτικού πρωτοκόλλου επικοινωνίας μεταξύ του module, το οποίο ανήκει στον χώρο πυρήνα και στην εφαρμογή, η οποία θα αναλάμβανε τις δικτυακές μεταφορές και ανήκει στον χώρο χρήστη. Ένας σημαντικός παράγοντας καθυστέρησης, στην επικοινωνία μεταξύ χώρου χρήστη και χώρου πυρήνα, είναι η αντιγραφή των δεδομένων. Για την εξάλειψη της καθυστέρησης αυτής, τα δεδομένα block δεν αντιγράφονται μεταξύ του χώρου χρήστη και του χώρου πυρήνα, αλλά οι χώροι στους οποίους περιέχονται ή πρέπει να τοποθετηθούν αυτά απεικονίζονται στον χώρο εικονικών διευθύνσεων της διεργασίας με την βοήθεια μιας συσκευής χαρακτήρων.

Μια άλλη σχεδιαστική επιλογή ήταν η υλοποίηση μιας γενικής και απλής βιβλιοθήκης, η οποία θα αναλάμβανε τις απαραίτητες δικτυακές μεταφορές για την εφαρμογή. Η βιβλιοθήκη αυτή επιτρέπει να προσαρμοστεί η συσκευή με εύκολο τρόπο σε διαφορετικές δικτυακές πλατφόρμες. Στα πλαίσια της εργασίας αυτής αναπτύχθηκαν απλές υλοποιήσεις της βιβλιοθήκης για τρία διαφορετικά δικτυακά περιβάλλοντα.

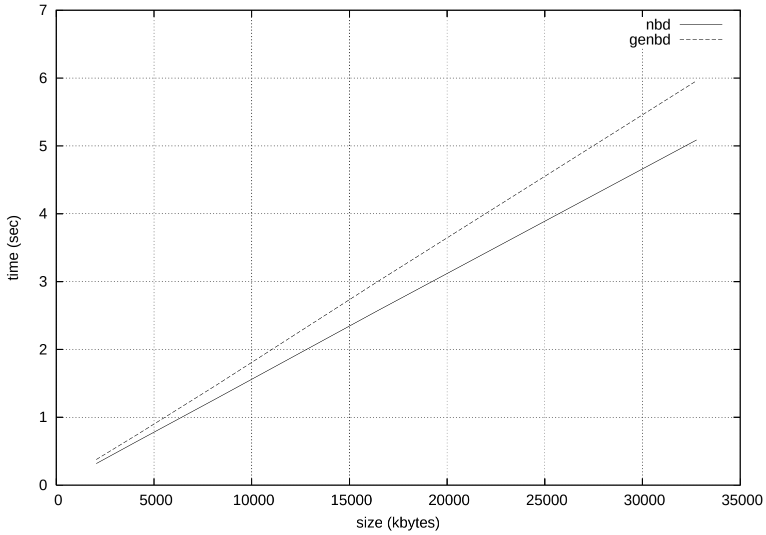
6.2 Επίδοση της Συσκευής

Για να συγκρίνουμε την επίδοση της συσκευής, χρησιμοποιήσαμε αναγνώσεις και εγγραφές διαφορετικού μεγέθους στο αρχείο της. Χρησιμοποιήθηκε η σημαία `O_DIRECT` κατά το άνοιγμα του αρχείου, ώστε να απενεργοποιηθεί η χρήση των κρυφών μνημών κατά την πρόσβαση στη συσκευή.

Αρχικά, συγκρίνουμε την επίδοση της συσκευής, όταν χρησιμοποιείται δίκτυο TCP/IP και FastEthernet με την συσκευή `nbd`, η οποία περιέχεται στον πυρήνα του Linux, κάτω από το ίδιο δικτυακό περιβάλλον. Τα αποτελέσματα παρουσιάζονται στο σχήμα 6.1. Συγκρίθηκε, ακόμα, η επίδοση της συσκευής για τις διάφορες υλοποιήσεις της βιβλιοθήκης `unet_ll`, που αναπτύχθηκαν και τα αποτελέσματα παρουσιάζονται στο σχήμα 6.2. Τέλος, τα χαρακτηριστικά των συστημάτων, στα οποία πραγματοποιήθηκαν οι μετρήσεις υπάρχουν στον πίνακα 6.1.

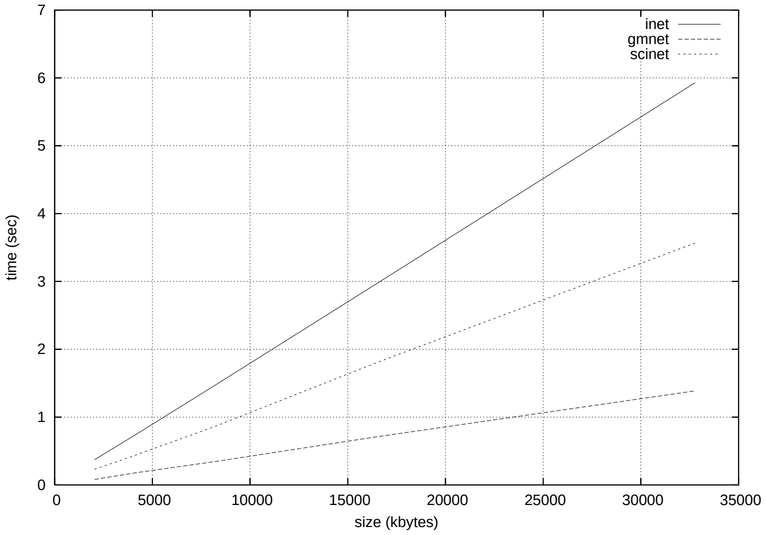


(a) Ανάγνωση Δεδομένων

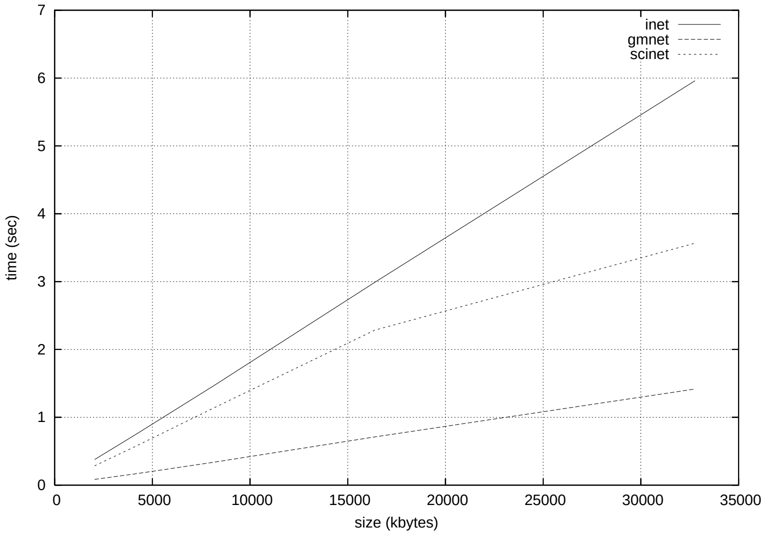


(b) Εγγραφή Δεδομένων

Σχήμα 6.1: Σύγκριση επίδοσης συσκευής genbd και nbd, κάτω από δικτυακό περιβάλλον TCP/IP και FastEthernet.



(a) Ανάγνωση Δεδομένων



(b) Εγγραφή Δεδομένων

Σχήμα 6.2: Σύγκρισή επίδοσης συσκευής genbd για τις διάφορες υλοποιήσεις της βιβλιοθήκης unet_ll

Υλικό (Hardware)	
Αριθμός Επεξεργαστών	2
Μοντέλο Επεξεργαστή	Pentium III, 1266MHz
Μέγεθος μνήμης	1011 Mbytes
Σύστημα Ε/Ε	PCI bus 64 bits 64 MHz
Προσαρμογέας FastEthernet	Intel Ethernet Pro 100
Προσαρμογέας SCI	Dolphin D330 PSB66 SCI-PCI
Προσαρμογέας Myrinet	LANai 9 133 MHz, 2 MB SRAM
Λογισμικό (Software)	
Έκδοση πυρήνα Linux	2.6.8.1
Έκδοση μεταγλωττιστή gcc	3.3.5
Έκδοση βιβλιοθήκης GM	2.0.13
Έκδοση βιβλιοθήκης SISCI (DIS)	2.5.0

Πίνακας 6.1: Χαρακτηριστικά υπολογιστικών συστημάτων που χρησιμοποιήθηκαν στις μετρήσεις.

6.3 Συμπεράσματα - Προοπτικές

Στα διαγράμματα του σχήματος 6.1 παρατηρούμε, ότι η συσκευή nbd επιτυγχάνει καλύτερη επίδοση από ότι η συσκευή genbd. Βέβαια, η διαφορά στην επίδοση είναι σχετικά μικρή και υπάρχουν αρκετά σημεία στη συσκευή genbd, τα οποία μπορούν να υλοποιηθούν με πιο αποδοτικό τρόπο. Θα πρέπει να σημειώσουμε, επίσης, ότι η πολυνηματική δομή της εφαρμογής `unet_cli` ευνοήθηκε σημαντικά από την αρχιτεκτονική SMP των υπολογιστικών συστημάτων που χρησιμοποιήθικαν.

Επιπρόσθετα, είναι φανερό, από τα διαγράμματα του σχήματος 6.2, ότι η χρήση μοντέρνων δικτυακών τεχνολογιών έχει ως αποτέλεσμα την σαφή βελτίωση στην επίδοση της συσκευής genbd, αφού οι δικτυακές μεταφορές πραγματοποιούνται πιο γρήγορα και με μικρότερη καθυστέρηση. Παρατηρούμε, επίσης, ότι η υλοποίηση του scinet έχει ιδιαίτερα χαμηλή επίδοση σε σχέση με αυτήν του gmnet γεγονός που υποδεικνύει ότι υπάρχουν πολλά περιθώρια βελτίωσης του.

Σε γενικές γραμμές υπάρχουν πολλά σημεία σε όλα τα επιμέρους στρώματα του genbd, τα οποία επιδέχονται βελτιώσεις, ώστε να αυξηθεί η συνολική επίδοση της συσκευής. Άλλωστε, η προσανατολισμός της σχεδίασης του genbd ήταν η επεκτασιμότητα και όχι η άμεση επίτευξη υψηλής επίδοσης.

Ενδεικτικά παρουσιάζονται ορισμένες από τις πιθανές επεκτάσεις που μπορούν να υλοποιηθούν:

- Δημιουργία πιο εξελιγμένης διεπαφής unet_II, η οποία να εκμεταλεύται πλήρως τα χαρακτηριστικά των μοντέρνων δικτύων διασύνδεσης.
- Δημιουργία δικτυακής υποδομής στον χώρο πυρήνα.
- Υποστήριξη πολλαπλών πελατών από τον εξυπηρετητή.
- Χρήση πολλαπλών εξυπηρετητών από τον πελάτη.

Βιβλιογραφία

[Ale01] Alessandro Rubini and Jonathan Corbet. Linux Device Drivers, 2nd Edition. O'Reilly, 2001.

[Axbda] Jens Axboe. Linux Block IO - present and future, Linux Symposium, July 21-24, 2004, Ottawa, Ontario, Canada.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. IEEE Micro, 15(1):29--36, Feb 1995.

[Bra02] Peter J. Braam. The Lustre Storage Architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, 2002.

[Cor] Jonathan Corbet. Porting drivers to the 2.5 kernel. Linux Symposium, July 23-26, 2003, Ottawa, Ontario, Canada.

[Dan03] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel, 2nd Edition. O'Reilly, 2003.

[DM03] Ulrich Drepper and Ingo Molnar. The Native Posix Thread Library for Linux, 2003.

[Dol01] Dolpin Interconnect Solutions. SISCI API Users Guide, May 2001. http://www.dolphinics.com.

[Gus92] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. IEEE Micro, 12(1):10--22, Jan/Feb 1992.

[Hel99] H. Hellwagner. SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Computer Clusters. Springer, 1999.

[HP02] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, 3d Edition. Morgan Kaufmann, 2002.

[inf02] InfiniBand Architecture Specification, Volume 1, Release 1.1, Infiniband Trade Association, 2002.

[Lev00] Chuck Lever. Linux Kernel Hash Table Behavior: Analysis and Imporovements, 2000.

[Myr] Myricom. The GM-2 Message Passing System. http://www.myri.com.

[Per] Kenneth W. Perslan. A 64 Bit, Shared Disk File System for Linux. IEEE Mass Storage Systems Symposium, March 15-18, 1999, San Diego, California.

[PH97] David A. Patterson and John L. Hennessy. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 1997.

[Rec03] Renato John Recio. Server i/o networks past, present, and future. In Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence, pages 163--178. ACM Press, 2003.

[RT] Ohad Rodeh and Avi Teperman. zfs - A Scalable Distributed File System Using Object Disks. 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies April 7-10, 2003, Paradise Point Resort, San Diego, California, USA.

[Schda] Philip Schwan. Lustre: Building a File System for 1.000-node Clusters, Linux Symposium, July 23-26, 2003, Ottawa, Ontario, Canada.

[SEP+] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart. The Global File System: A File System for Shared Disk Storage. submitted to the IEEE Transactions on Parallel and Distributed Systems, October 1997.

[Ste93] W. Richard Stevens. TCP/IP illustrated (vol. 1): the protocols. Addison-Wesley Longman Publishing Co., Inc., 1993.

[Ste97] W. Richard Stevens. UNIX Network Programming: Networking APIs: Sockets and XTI. Prentice Hall PTR, 1997.

[VS] K. Voruganti and P. Sarkar. An Analysis of Three Gigabit Networking Protocols for Storage Area Networks. 20th IEEE International Performance, Computing, and Communications Conference, April 2001.