ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Διπλωματική Εργασία

Υλοποίηση ενός Γενικού Στρώματος Επικοινωνίας Πυρήνα προς Πυρήνα Linux μέσω Χώρου Χρήστη, Χρησιμοποιώντας Δίκτυα SCI και FastEthernet

Ευάγγελος Κούκης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης

Αθήνα, Νοέμβριος 2002

Πρόλογος

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου το ακαδημαϊκό έτος 2001-2002. Νιώθω την ανάγκη να εκφράσω τις θερμές ευχαριστίες μου στον επιβλέποντα καθηγητή κ. Νεκτάριο Κοζύρη για τη συμπαράσταση, την καθοδήγηση και το ενδιαφέρον του καθ' όλη τη διάρκεια της εκπόνησης της εργασίας αυτής. Θα ήθελα επίσης να ευχαριστήσω ιδιαίτερα τον υποψήφιο διδάκτορα Α. Σωτηρόπουλο και τον φοιτητή Γ. Τσουκαλά για τα εποικοδομητικά σχόλιά τους και τις εύστοχες παρατηρήσεις τους. Τέλος, ευχαριστώ τα μέλη της τριμελούς επιτροπής, καθηγητές του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών κ.κ. Γεώργιο Παπακωνσταντίνου και Παναγιώτη Τσανάκα, καθώς και τα υπόλοιπα μέλη του Εργαστηρίου Υπολογιστικών Συστημάτων.

Πίνακας Περιεχομένων

Εισαγωγή

Στις μέρες μας, οι συστοιχίες υπολογιστών (workstation clusters) κερδίζουν όλο και περισσότερο έδαφος ως σχεδιαστική επιλογή για την κατασκευή υπολογιστικών συστημάτων υψηλών επιδόσεων. Οι συστοιχίες υπολογιστών έχουν ορισμένα χαρακτηριστικά που τις κάνουν ιδιαίτερα ελκυστικές σε σύγκριση με άλλες τεχνολογίες συστημάτων υψηλών επιδόσεων, όπως είναι τα συστήματα συμμετρικής πολυεπεξεργασίας (Symmetric Multiprocessing Systems – SMP). Ανάμεσα σε αυτά θα μπορούσαμε να αναφέρουμε τη δυνατότητα κλιμάκωσης σε πολύ μεγαλύτερο αριθμό επεξεργαστών, τον πολύ καλύτερο λόγο κόστους υλικού προς απόδοση του συστήματος κ.ά.

Μια συστοιχία υπολογιστών θα μπορούσε να οριστεί ως ένας αριθμός υπολογιστών (*κόμβων*), που επικοινωνούν μεταξύ τους μέσω ενός δικτύου διασύνδεσης υψηλών επιδόσεων. Σε αντίθεση με ένα σύστημα συμμετρικής πολυεπεξεργασίας (*στενά συνδεδεμένο σύστημα*), όπου οι επεξεργαστές επικοινωνούν μεταξύ τους μέσω ενός διαδρόμου συστήματος (system bus) και απέχουν εξίσου από την κοινή μνήμη του συστήματος, μια συστοιχία υπολογιστών είναι ένα κατανεμημένο σύστημα (*χαλαρά συνδεδεμένο σύστημα*), όπου κάθε ένας από τα επιμέρους συστήματα που συμμετέχει σε αυτή διαθέτει τη δική του τοπική μνήμη και ο κοινός δίαυλος επικοινωνίας έχει αντικατασταθεί από ένα δίκτυο διασύνδεσης υψηλών επιδόσεων, το οποίο μπορεί να κλιμακωθεί σε πολύ μεγαλύτερο αριθμό επεξεργαστών.

Τα πλεονεκτήματα που προσφέρουν οι συστοιχίες υπολογιστών μπορούν να συνοψιστούν στα εξής:

- **Δυνατότητα κλιμάκωσης:** Σε αντίθεση με τα συστήματα συμμετρικής πολυεπεξεργασίας, οι συστοιχίες υπολογιστών δεν περιορίζονται από την ύπαρξη ενός μοναδικού διαδρόμου συστήματος και μπορούν να κλιμακωθούν σε εκατοντάδες ή και χιλιάδες επεξεργαστές.

- **Αξιοπιστία / υψηλή διαθεσιμότητα:** Στην περίπτωση που υπάρξει πρόβλημα στη λειτουργία ενός από τους κόμβους που αποτελούν τη συστοιχία, το υπολογιστικό φορτίο μπορεί να κατανεμηθεί στους υπόλοιπους κόμβους.

- **Καταμερισμός του φορτίου Ε/Ε:** Εκτός από το υπολογιστικό φορτίο, είναι δυνατό να καταμεριστεί και το φορτίο Εισόδου / Εξόδου σε μια συστοιχία υπολογιστών, χρησιμοποιώντας κατανεμημένα συστήματα αρχείων. Με τον τρόπο αυτό μια συστοιχία υπολογιστών μπορεί να αντεπεξέλθει σε μεγαλύτερο φόρτο αιτήσεων Ε/Ε συγκρινόμενη με ένα συγκεντρωμένο υπολογιστικό σύστημα.

- **Καλύτερος λόγος τιμής / απόδοσης:** Τα επιμέρους υπολογιστικά συστήματα που συνδυάζονται για τον σχηματισμό μιας συστοιχίας υπολογιστών είναι έτοιμες, χαμηλού κόστους, δοκιμασμένες λύσεις, ευρέως διαθέσιμες στην αγορά. Έτσι, καθίσταται ευκολότερη η αύξηση της υπολογιστικής ισχύος όταν αυτό γίνει αναγκαίο, ενώ διατηρείται σχεδόν σταθερός ο λόγος τιμής / απόδοσης ως συνάρτηση του αριθμού των επεξεργαστών στο σύστημα.

Γίνεται φανερό ότι η δυνατότητα για αποδοτική υλοποίηση συστοιχιών μεγάλου αριθμού επεξεργαστών είναι άμεσα συνδεδεμένη με την ανάπτυξη αποδοτικών δικτύων διασύνδεσης. αλλά και με την κατάλληλη προσαρμογή του λογισμικού συστήματος των επιμέρους υπολογιστικών συστημάτων.

Την πρώτη ανάγκη έρχονται να καλύψουν νέα εξελιγμένα δίκτυα διασύνδεσης, όπως είναι το SCI (Scalable Coherent Interface [9], [11]) και το Myrinet. Το SCI υλοποιεί λειτουργίες παρόμοιες με αυτές ενός διαδρόμου συστήματος, προσφέροντας ένα περιβάλλον κατανεμημένης μοιραζόμενης μνήμης μέσω απομακρυσμένων εντολών αποθήκευσης/ανάκλησης δεδομένων. Ο ρυθμός διαμεταγωγής δεδομένων που επιτυγχάνεται είναι ιδιαίτερα υψηλός, χρησιμοποιώντας συνδέσμους σημείου-προς-σημείο της τάξης του

1GB/sec ενώ αρκετά μικρός είναι και ο χρόνος αρχικής απόκρισης (latency), της τάξης των 5μsec.

Για την ικανοποίηση της δεύτερης ανάγκης, της ανάγκης για προσαρμογή του λογισμικού συστήματος έτσι ώστε να εκτελείται αποδοτικότερα πάνω σε μια συστοιχία υπολογιστών, χρειάζεται η κατασκευή επεκτάσεων στο Λειτουργικό Σύστημα (Λ.Σ.) των υπολογιστών που αποτελούν τη συστοιχία. Οι επεκτάσεις αυτές μπορούν για παράδειγμα να υλοποιούν κάποιο κατανεμημένο μοιραζόμενο σύστημα αρχείων.

Η ανάπτυξη αυτών των επεκτάσεων του Λ.Σ. είναι επιθυμητό να γίνει με τρόπο ανεξάρτητο από το χρησιμοποιούμενο δίκτυο διασύνδεσης σε κάθε περίπτωση. Για το σκοπό αυτό, είναι χρήσιμη η ύπαρξη ενός γενικού στρώματος επικοινωνίας, σχεδιασμένο για χρήση σε επίπεδο πυρήνα λειτουργικού συστήματος, το οποίο θα αναλαμβάνει την αξιόπιστη διακίνηση των δεδομένων ανάμεσα στους κόμβους της συστοιχίας.

Αυτός είναι και ο σκοπός της παρούσας διπλωματικής εργασίας: Ο σχεδιασμός και η υλοποίηση ενός γενικού στρώματος επικοινωνίας πυρήνα προς πυρήνα λειτουργικού συστήματος, το οποίο θα επιτρέπει την ασφαλή, αξιόπιστη αλλά και γρήγορη διακίνηση δεδομένων με τρόπο ανεξάρτητο του χρησιμοποιούμενου δικτύου διασύνδεσης. Το στρώμα αυτό ονομάζεται KSockets (Kernel Sockets).

Το λειτουργικό σύστημα για το οποίο σχεδιάστηκε και υλοποιήθηκε το στρώμα των KSockets είναι το Linux. Ένας από τους σημαντικότερους λόγους που συνηγορούν στην επιλογή του Linux, είναι το γεγονός ότι ο πηγαίος του κώδικας είναι ελεύθερα διαθέσιμος, κάνοντάς το ιδανικό για χρήστη στον ακαδημαϊκό χώρο.

Ένα ιδιαίτερο χαρακτηριστικό του στρώματος επικοινωνίας είναι ότι δεν υλοποιείται εξ ολοκλήρου ως τμήμα του πυρήνα του Linux, αλλά ως ένας συνδυασμός αρθρωμάτων του πυρήνα (Linux kernel modules) και μιας πολυνηματικής διεργασίας που τρέχει στον χώρο χρήστη (userspace). Η σχεδιαστική αυτή επιλογή υπαγορεύεται από συγκεκριμένες ανάγκες και έχει ορισμένα πλεονεκτήματα και μειονεκτήματα τα οποία αναλύονται διεξοδικά στη συνέχεια. Τα υποστηριζόμενα δίκτυα διασύνδεσης είναι το SCI (μέσω της βιβλιοθήκης SISCI – Software Infrastructure for SCI) και το Gigabit- ή Fast-Ethernet, μέσω των TCP/IP sockets. Ωστόσο, το τμήμα των KSockets που εξαρτάται από το χρησιμοποιούμενο δίκτυο διασύνδεσης είναι ανεξάρτητο από τα υπόλοιπα, κι έτσι είναι δυνατή η επέκταση τους ώστε να υποστηρίζονται και άλλα δίκτυα διασύνδεσης. Η επιλογή του δικτύου διασύνδεσης κάθε φορά

δεν επηρεάζει τον τρόπο με τον οποίο χρησιμοποιείται το στρώμα επικοινωνίας από άλλα τμήματα του Λ.Σ.

Η παρουσίαση του σχεδιασμού του στρώματος των KSockets είναι απαραίτητο να συνοδεύεται από την εξέταση μιας σειράς θεμάτων που αφορούν σε αυτόν. Έτσι, στο 1ο κεφάλαιο γίνεται σύντομη αναφορά στη δομή του πυρήνα ενός λειτουργικού συστήματος, εξετάζεται η διάκριση ανάμεσα στο χώρο πυρήνα και τον χώρο χρήστη και παρουσιάζονται οι τρόποι για την επικοινωνία ανάμεσα στο χώρο πυρήνα και στο χώρο χρήστη κάτω από το Linux. Επιπλέον, γίνεται σύντομη αναφορά στους μηχανισμούς που προσφέρει ο πυρήνας του Linux για εξασφάλιση της ακεραιότητας κατά την ταυτόχρονη πρόσβαση σε μοιραζόμενα δεδομένα.

Στο 2ο κεφάλαιο παρουσιάζεται το υποσύστημα οδηγών υλικού του πυρήνα του Linux και πιο συγκεκριμένα ο τρόπος αλληλεπίδρασης με τον πυρήνα ενός οδηγού συσκευής χαρακτήρων. Επιπλέον, γίνεται αναφορά στο εικονικό σύστημα αρχείων (VFS) του Linux και τη σημασία του κατά την κατασκευή οδηγών συσκευών χαρακτήρων.

Στο 3ο κεφάλαιο αναφέρονται οι ανάγκες που καλούνται να καλύψουν τα KSockets και προδιαγράφονται οι απαιτήσεις από το σύστημα επικοινωνίας. Παρουσιάζεται επίσης το σύνολο των λειτουργιών που προσφέρονται (*KSockets Programming Interface*) και περιγράφεται ο τρόπος χρήσης των λειτουργιών αυτών από άλλα τμήματα του πυρήνα.

Στο 4ο κεφάλαιο γίνεται περιγραφή του τμήματος των KSockets που εκτελείται σε χώρο πυρήνα. Αναλύεται οι δομές δεδομένων που χρησιμοποιούνται κατά τη λειτουργία του, περιγράφεται ο τρόπος επικοινωνίας με το τμήμα που εκτελείται σε χώρο χρήστη και παρουσιάζεται ο οδηγός της εικονικής συσκευής χαρακτήρων που χρησιμοποιείται για την επίτευξη αυτής της επικοινωνίας.

Το 5ο κεφάλαιο είναι αφιερωμένο στο τμήμα των KSockets που υλοποιείται ως διεργασία χώρου χρήστη. Γίνεται συνοπτική αναφορά στη βιβλιοθήκη δημιουργίας πολυνηματικών εφαρμογών POSIX Threads και παρουσιάζεται ο σχεδιασμός της διεργασίας χώρου χρήστη με τρόπο ανεξάρτητο του δικτύου διασύνδεσης. Επιπλέον, προδιαγράφεται η διαπροσωπεία προς το τμήμα της διεργασίας χώρου χρήστη που εξαρτάται από το δίκτυο διασύνδεσης.

Το 6ο κεφάλαιο πραγματεύεται την επέκταση της διεργασίας χώρου χρήστη έτσι ώστε να υποστηρίζει την επικοινωνία πάνω από δίκτυα που βασίζονται στο πρωτόκολλο TCP/IP.

Παρουσιάζεται η στοίβα πρωτοκόλλων του TCP/IP και αναλύεται η πρόσβαση σε αυτή μέσω του μηχανισμού των BSD Sockets.

Στο 7ο κεφάλαιο γίνεται παρουσίαση του προτύπου διασύνδεσης SCI και αναλύονται τα ιδιαίτερα πλεονεκτήματα που προσφέρει ως δίκτυο διασύνδεσης για συστοιχίες υπολογιστών. Στη συνέχεια, παρουσιάζεται συνοπτικά η βιβλιοθήκη χώρου χρήστη SISCI, μέσω της οποίας παρέχεται ομοιόμορφη πρόσβαση στις υπηρεσίες του SCI.

Το θέμα του 8ου κεφαλαίου είναι η επέκταση της διεργασίας χώρου χρήστη έτσι ώστε να υποστηρίζεται η μεταφορά των δεδομένων του στρώματος επικοινωνίας πάνω από δίκτυα SCI. Αναλύεται η διαχείριση των πόρων του δικτύου μέσω της βιβλιοθήκης SISCI καθώς και η διαδικασία συγχρονισμού και ελέγχου ροής ανάμεσα σε δύο υπολογιστικά συστήματα που ανταλλάσσουν δεδομένα.

Τέλος, στο 9ο κεφάλαιο παρουσιάζεται ένας απλός οδηγός συσκευής για τον πυρήνα του Linux, ο οποίος χρησιμοποιεί τις υπηρεσίες του στρώματος KSockets για να υλοποιήσει μια δικτυακή σωλήνωση (network pipe) ανάμεσα σε δύο υπολογιστές. Ο οδηγός αυτός χρησιμοποιείται για τον έλεγχο της σωστής λειτουργίας αλλά και τη διεξαγωγή μετρήσεων για τους βασικούς δείκτες επίδοσης κατά την επικοινωνία μέσω των KSockets.

Στο παράρτημα Α, παρατίθεται ολόκληρος ο πηγαίος κώδικας, σε γλώσσα C για τα διάφορα τμήματα του στρώματος επικοινωνίας καθώς και για τον δοκιμαστικό οδηγό συσκευής που χρησιμοποιήθηκε για τον έλεγχο της σωστής λειτουργίας του.

Κεφάλαιο 1

Χαρακτηριστικά του πυρήνα του Linux

1.1 Ο ρόλος του λειτουργικού συστήματος

Το *λειτουργικό σύστημα* (Λ.Σ.) είναι ένα πρόγραμμα το οποίο λειτουργεί ως σύνδεσμος ανάμεσα στο υλικό ενός υπολογιστικού συστήματος και το λογισμικό εφαρμογών, που χρησιμοποιείται από τους χρήστες του υπολογιστικού συστήματος. Σκοπός του είναι η διευκόλυνση των χρηστών αλλά και η αποδοτική χρήση των συσκευών που αποτελούν το υλικό του υπολογιστικού συστήματος.

Ένα σύγχρονο Λ.Σ. πολυπρογραμματισμού, που χρησιμοποιεί την αρχή του καταμερισμού χρόνου, είναι σχεδιασμένο ώστε να επιτρέπει σε πολλές διαφορετικές διεργασίες (μονάδες εκτέλεσης, προγράμματα που εκτελούνται) να εκτελούνται συγχρόνως από το υπολογιστικό σύστημα, μοιραζόμενες τους πόρους του. Για να καταστεί αυτό δυνατό, το Λ.Σ. επιτελεί μια σειρά λειτουργιών, οι σημαντικότερες από τις οποίες είναι οι εξής:

- **Διαχείριση ΚΜΕ:** Εφόσον πολλές διεργασίες συναγωνίζονται για εκτέλεση στην Κεντρική Μονάδα Επεξεργασίας, το λειτουργικό σύστημα αναλαμβάνει τη δίκαιη κατανομή του χρόνου της ΚΜΕ στις διάφορες διεργασίες. Κάθε διεργασία χρησιμοποιεί για πεπερασμένο χρονικό διάστημα την ΚΜΕ και στη συνέχεια η εκτέλεσή της διακόπτεται για να δοθεί η ευκαιρία και στις υπόλοιπες να

εκτελεστούν. Η διαδικασία επιλογής της πλέον κατάλληλης προς εκτέλεσης διεργασίας κάθε φορά ονομάζεται *χρονοδρομολόγηση* (CPU scheduling).

- **Διαχείριση των μονάδων Ε/Ε:** Καθώς πολλές διεργασίες ανταγωνίζονται για πρόσβαση στις μονάδες Εισόδου / Εξόδου του υπολογιστικού συστήματος, το Λ.Σ. συντονίζει και ελέγχει την πρόσβαση στις μονάδες αυτές. Οι διεργασίες δεν έχουν δικαίωμα απευθείας χρήσης των περιφερειακών συσκευών, αλλά εκμεταλλεύονται ειδικές υπηρεσίες Ε/Ε του Λ.Σ., το οποίο αναλαμβάνει την αποδοτική και ασφαλή εκτέλεση των λειτουργιών Ε/Ε εκ μέρους των διεργασιών.

- **Διαχείριση μνήμης:** Οι διεργασίες που εκτελούνται συγχρόνως είναι υποχρεωμένες να μοιράζονται μεταξύ τους την κεντρική μνήμη του συστήματος. Στα σύγχρονα Λ.Σ., κάθε μία από τις διεργασίες εκτελείται στο δικό της εικονικό χώρο διευθύνσεων, ο οποίος είναι ανεξάρτητος από αυτόν κάποιας άλλης διεργασίας. Το λειτουργικό σύστημα αναλαμβάνει την απεικόνιση του εικονικού χώρου μνήμης κάθε διεργασίας σε ένα τμήμα της κεντρικής μνήμης του υπολογιστικού συστήματος. Στην περίπτωση που αυτή δεν επαρκεί, τότε ανήκει στο Λ.Σ. η ευθύνη να αποφασίσει ποια τμήματα του εικονικού χώρου διευθύνσεων κάποιας διεργασίας θα αποθηκευτούν προσωρινά σε μονάδες περιφερειακής μνήμης, για να ανακληθούν στην κεντρική μνήμη όταν χρειαστεί να χρησιμοποιηθούν. Η πρόσβαση σε περιοχές μνήμης που έχουν κατανεμηθεί σε μια διεργασία δεν επιτρέπεται για τις υπόλοιπες. Με τον τρόπο αυτό, το σύστημα είναι προστατευμένο από προβληματικές διεργασίες ή από κακόβουλους χρήστες, οι οποίοι επιθυμούν πρόσβαση στα δεδομένα άλλων χρηστών.

- **Διαχείριση συστημάτων αρχείων:** Το λειτουργικό σύστημα αναλαμβάνει την αποθήκευση και ανάκληση των αρχείων στις διάφορες περιφερειακές μονάδες αποθήκευσης, οργανώνοντάς τα σε συστήματα αρχείων. Όταν ένα πρόγραμμα εφαρμογής χρειαστεί δεδομένα τα οποία είναι αποθηκευμένα σε κάποιο αρχείο υποβάλλει κατάλληλο αίτημα προς το Λ.Σ., το οποίο αναλαμβάνει να ελέγξει αν ο χρήστης του προγράμματος έχει δικαίωμα πρόσβασης στα δεδομένα που ζητούνται και στη συνέχεια να τα ανακτήσει και να τα παραδώσει στην εφαρμογή.

1.2 Χώρος πυρήνα και χώρος χρήστη

Όταν μιλάμε για τον *πυρήνα* ενός λειτουργικού συστήματος, εννοούμε το βασικό εκείνο τμήμα εκείνο του Λ.Σ. το οποίο φορτώνεται στην κεντρική μνήμη του υπολογιστικού συστήματος κατά την εκκίνησή του, παραμένει εκεί καθ' όλη τη διάρκεια της λειτουργίας του υπολογιστικού συστήματος και επιτελεί τις θεμελιώδεις λειτουργίες που αναφέρθηκαν προηγούμενα.

Γίνεται φανερό ότι υπάρχει διάκριση στον τρόπο κατά τον οποίο εκτελείται από την ΚΜΕ κώδικας που ανήκει στον πυρήνα και κώδικας που ανήκει σε προγράμματα εφαρμογής. Ο κώδικας του πυρήνα έχει τη δυνατότητα να αλληλεπιδρά άμεσα με το υλικό, έχει τον πλήρη έλεγχο των περιφερειακών μονάδων και μπορεί να προσπελάσει οποιαδήποτε διεύθυνση της κεντρικής μνήμης, ανεξάρτητα από τη διεργασία στην οποία ανήκει.

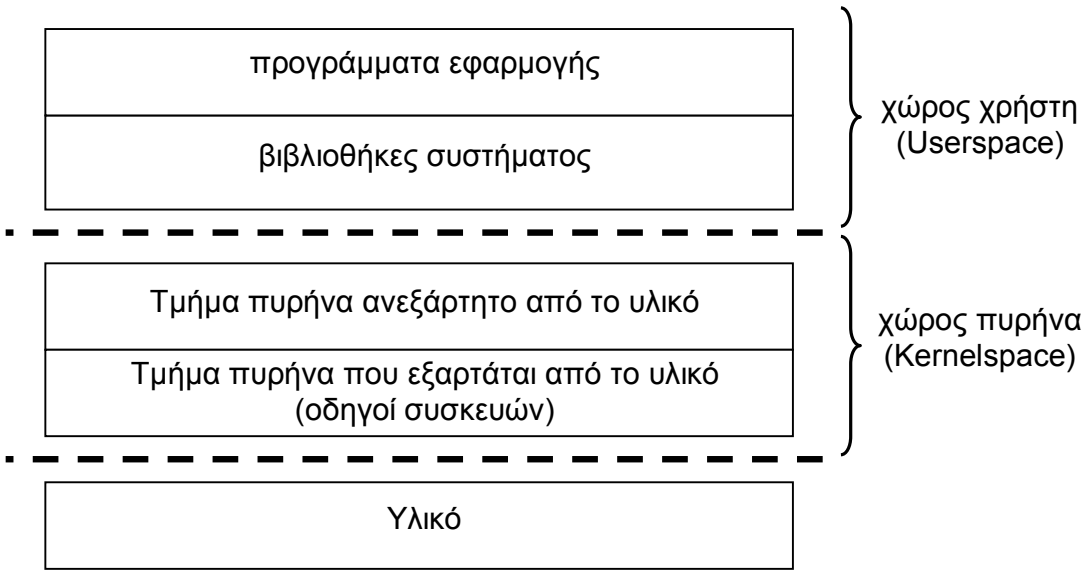
Σε αντίθεση με τον κώδικα του πυρήνα ενός Λ.Σ., ο κώδικας ενός προγράμματος εφαρμογής που εκτελείται ως διεργασία χρήστη δεν έχει τη δυνατότητα άμεσου ελέγχου του υλικού, έτσι ώστε να είναι δυνατή η ταυτόχρονη εκτέλεση πολλών διεργασιών και η εξασφάλιση της σωστής λειτουργίας του συστήματος. Επιπλέον, δεν του επιτρέπεται η πρόσβαση σε διευθύνσεις μνήμης οι οποίες δεν έχουν κατανεμηθεί στη διεργασία από το λειτουργικό σύστημα. και υποχρεώνεται να απελευθερώνει περιοδικά τον έλεγχο της ΚΜΕ ώστε να εκτελεστούν και οι υπόλοιπες διεργασίες.

Για να είναι δυνατή η διάκριση ανάμεσα στον κώδικα του πυρήνα και τον κώδικα των προγραμμάτων των χρηστών απαιτείται υποστήριξη από το υλικό του υπολογιστικού συστήματος. Έτσι, είναι απαραίτητες δύο (τουλάχιστον) διαφορετικές καταστάσεις λειτουργίας της ΚΜΕ: Η *κατάσταση επόπτη*, ή προνομιούχος κατάσταση (monitor ή supervisor ή privileged mode) και η *κατάσταση χρήστη* (user mode). Επιπρόσθετα, ορισμένες εντολές της ΚΜΕ χαρακτηρίζονται ως *προνομιούχες*, και επιτρέπεται η χρήση τους μόνο όταν η ΚΜΕ βρίσκεται στην κατάσταση επόπτη. Αν μια διεργασία προσπαθήσει να εκτελέσει μια τέτοια εντολή, ενώ το σύστημα βρίσκεται σε κατάσταση χρήστη, τότε συμβαίνει μια εξαίρεση υλικού (*hardware exception*) και ο έλεγχος μεταφέρεται στο λειτουργικό σύστημα, το οποίο αναλαμβάνει να τερματίσει τη διεργασία.

Για να εξασφαλιστούν οι απαιτήσεις απομόνωσης των προγραμμάτων χρήστη από το υλικό και η δίκαιη κατανομή του χρόνου της ΚΜΕ στις διάφορες διεργασίες, οι εντολές Ε/Ε

καθώς και οι εντολές που επηρεάζουν το σύστημα διακοπών (interrupts) της ΚΜΕ χαρακτηρίζονται προνομιούχες. Επιπλέον, για να εξασφαλιστεί η προστασία του χώρου μνήμης που έχει κατανεμηθεί σε κάθε διεργασία αλλά και η προστασία των δεδομένων που χρησιμοποιούνται από το ίδιο το Λ.Σ., χαρακτηρίζονται προνομιούχες οι εντολές που αλλάζουν τους καταχωρητές της ΚΜΕ που χρησιμοποιούνται για έλεγχο των ορίων των περιοχών μνήμης κατά την εκτέλεση εντολών αποθήκευσης / ανάκλησης από αυτή. Με τον τρόπο αυτό, μόνο ο πυρήνας του Λ.Σ., που εκτελείται σε κατάσταση επόπτη μπορεί να αλλάξει τα επιτρεπόμενα όρια κατά την πρόσβαση σε περιοχές της μνήμης και να μεταβάλλει τα δεδομένα οποιασδήποτε διεργασίας.

Στο ακόλουθο διάγραμμα παρουσιάζεται η ιεραρχία των προγραμμάτων που εκτελούνται σε ένα υπολογιστικό σύστημα, σύμφωνα με όσα έχουν αναφερθεί ως τώρα:



Σχήμα 1.1 – Ιεραρχία κώδικα πυρήνα και προγραμμάτων χρήστη

1.3 Ο πυρήνας του λειτουργικού συστήματος Linux

Το Linux είναι ένα σύγχρονο λειτουργικό σύστημα, βασισμένο στις σχεδιαστικές αρχές και την παράδοση του λειτουργικού συστήματος UNIX. Η ανάπτυξη του ξεκίνησε το 1991, από τον Φινλανδό φοιτητή Linus Torvalds, ο οποίος ανέπτυξε την πρώτη έκδοση του πυρήνα του Linux, η οποία ήταν σχεδιασμένη για να τρέχει σε υπολογιστικά συστήματα με τον επεξεργαστή 80386 της Intel. Το γεγονός ότι ο πηγαίος του κώδικας ήταν ανοικτός, διαθέσιμος στον καθένα μέσω του διαδικτύου για μελέτη και μεταβολή / επέκταση,

οδήγησε γρήγορα στην ανάπτυξη νέων, βελτιωμένων εκδόσεων που ενσωμάτωναν όλο και περισσότερες δυνατότητες.

Σήμερα, η ανάπτυξη του Linux συνεχίζεται ενεργά από μια ομάδα προγραμματιστών που συνεργάζονται μέσω του διαδικτύου και είναι μια διαδικασία ανοικτή σε οποιονδήποτε έχει την ικανότητα και επιθυμεί να συμμετάσχει. Ο πυρήνας του Λ.Σ. έχει μεταφερθεί σε πολλές διαφορετικές αρχιτεκτονικές υπολογιστικών συστημάτων και υποστηρίζει πολλά από τα χαρακτηριστικά που θα περίμενε κανείς από ένα σύγχρονο λειτουργικό σύστημα: Το Linux είναι ένα πολυχρηστικό σύστημα καταμερισμού χρόνου, με εικονική μνήμη, μοιραζόμενες βιβλιοθήκες, δυνατότητα δικτύωσης, υποστηρίζει πλήθος συστημάτων αρχείων, ενώ μπορεί να εκμεταλλευτεί αποδοτικά τις δυνατότητες συστημάτων συμμετρικής πολυεπεξεργασίας (SMP). Η ανάπτυξή του στοχεύει στην συμμόρφωση με το πρότυπο POSIX, ένα σύνολο προδιαγραφών για διάφορα τμήματα ενός λειτουργικού συστήματος που βασίζεται στη φιλοσοφία του UNIX.

1.3.1 Σχεδιαστικές αρχές του πυρήνα

Ο πυρήνας του Linux, ακολουθώντας την παραδοσιακή μέθοδο σχεδιασμού πυρήνων για λειτουργικά συστήματα που συμβαδίζουν με τη φιλοσοφία του UNIX, είναι ένας *μονολιθικός* πυρήνας. Είναι ένα μεγάλο, ενιαίο εκτελέσιμο, το οποίο παραμένει συνεχώς φορτωμένο στην κεντρική μνήμη, ο κώδικάς του εκτελείται στην προνομιούχο κατάσταση της ΚΜΕ και αναλαμβάνει την υλοποίηση όλων σχεδόν των λειτουργιών του Λ.Σ.: Από την χρονοδρομολόγηση στην ΚΜΕ, τη διαχείριση εικονικής μνήμης και την διαχείριση των περιφερειακών μονάδων, έως την υλοποίηση δικτυακών πρωτοκόλλων (πχ. στοίβα TCP/IP) και την υποστήριξη πολλών διαφορετικών συστημάτων αρχείων, όλες οι λειτουργίες του πυρήνα του Linux διεκπεραιώνονται από κώδικα που βρίσκεται στον ίδιο εικονικό χώρο διευθύνσεων, τον χώρο πυρήνα.

Αυτή η σχεδιαστική επιλογή έγινε με γνώμονα την ανάγκη για υψηλές επιδόσεις. Εφόσον ο κώδικας του λειτουργικού συστήματος και οι δομές δεδομένων που αυτός διαχειρίζεται είναι αποθηκευμένες στον ίδιο εικονικό χώρο διευθύνσεων, δεν απαιτείται *μεταγωγή περιεχομένου* (context switch) όταν συμβεί μια διακοπή υλικού ή μια διεργασία χρήστη ζητήσει τις υπηρεσίες του λειτουργικού συστήματος. Το κυριότερο μειονέκτημα αυτής της σχεδίασης είναι η μειωμένη ευελιξία που παρέχεται κατά την επέκταση του πυρήνα και την προσθήκη νέων δυνατοτήτων, εφόσον κάθε αλλαγή μπορεί να επηρεάσει δραστικά τη

λειτουργία του υπόλοιπου του πυρήνα και ακόμη και ένα μικρό τμήμα λανθασμένου κώδικα μπορεί να έχει οδυνηρές επιπτώσεις στη σταθερότητα του συστήματος.

Μια νεότερη σχεδιαστική αντίληψη για τον πυρήνα ενός λειτουργικού συστήματος, που ακολουθείται από αρκετά σύγχρονα Λ.Σ., είναι αυτή του *μικροπυρήνα*. Σύμφωνα με αυτή, τα τμήματα του Λ.Σ. που δεν είναι απαραίτητο να εκτελούνται σε προνομιούχο κατάσταση (όπως είναι η υλοποίηση δικτυακών πρωτοκόλλων και η υποστήριξη συστημάτων αρχείων) υλοποιούνται από ανεξάρτητες διεργασίες συστήματος, ενώ το τμήμα του Λ.Σ. που εκτελείται σε προνομιούχο κατάσταση συρρικνώνεται στην υποδομή που είναι απολύτως απαραίτητη ώστε να είναι δυνατή η εκτέλεση των διεργασιών αυτών και η επικοινωνία μεταξύ τους. Ο μικροπυρήνας που προκύπτει αναλαμβάνει τη διαχείριση των πόρων του υπολογιστικού συστήματος (χρονοδρομολόγηση ΚΜΕ, διαχείριση εικονικής μνήμης, διαχείριση διακοπών από περιφερειακές συσκευές) και την ανταλλαγή μηνυμάτων ανάμεσα στις διεργασίες, ενώ όλα τα άλλα χαρακτηριστικά του Λ.Σ. υλοποιούνται από διακεκριμένες διεργασίες που επικοινωνούν μεταξύ τους μέσω μηνυμάτων καθορισμένης μορφής.

Τα πλεονεκτήματα της σχεδίασης με τη μέθοδο του μικροπυρήνα (διάκριση και απομόνωση των λειτουργιών, ευκολότερος εντοπισμός και διόρθωση προγραμματιστικών λαθών στον κώδικα, ευελιξία κατά την προσθήκη νέων δυνατοτήτων στο Λ.Σ., ευκολία μεταφοράς σε νέες αρχιτεκτονικές) αντισταθμίζονται από τη μείωση των επιδόσεων και την καθυστέρηση που εισάγει η ανταλλαγή μηνυμάτων ανάμεσα στις διεργασίες που υλοποιούν το Λ.Σ., ενώ αυξάνεται και ο αριθμός των μεταγωγών περιεχομένου που απαιτούνται.

Το Linux προσφέρει αρκετά από τα πλεονεκτήματα μιας σχεδίασης μικροπυρήνα, διατηρώντας παράλληλα το χαρακτήρα του ως μονολιθικό σύστημα, μέσω του μηχανισμού των *αρθρωμάτων πυρήνα*, ή Linux kernel modules.

1.3.2 Linux Kernel Modules

Ο πυρήνας του Linux έχει τη δυνατότητα να φορτώνει και να αφαιρεί από το χώρο διευθύνσεων του πυρήνα τμήματα κώδικα, όποτε αυτό είναι αναγκαίο. Τα τμήματα κώδικα αυτά ονομάζονται αρθρώματα του πυρήνα του Linux, ή kernel modules. Ο κώδικας που περιέχεται σε κάποιο module του πυρήνα εισάγεται στον εικονικό χώρο διευθύνσεων ενός

πυρήνα που βρίσκεται ήδη σε λειτουργία και από τη στιγμή που συμβεί αυτό δεν υπάρχει καμία διάκριση ανάμεσα σε αυτόν και τον κώδικα του πυρήνα που προϋπήρχε.

Έτσι, εκτελείται επίσης στην προνομιούχο κατάσταση της ΚΜΕ και έχει πλήρη πρόσβαση σε κάθε εσωτερική δομή δεδομένων του πυρήνα και στο χώρο διευθύνσεων κάθε διεργασίας, όπως θα περίμενε κανείς από ένα μονολιθικό σύστημα. Παρόλο που δεν υπάρχει κανένας περιορισμός ως προς τι θα μπορούσε να υλοποιηθεί ως module, επιλέγεται να υλοποιούνται ως modules οδηγοί για συσκευές υλικού αλλά και εικονικές συσκευές, συστήματα αρχείων και πρωτόκολλα δικτύων.

Η ύπαρξη του μηχανισμού των modules στον πυρήνα του Linux παρέχει ορισμένα σημαντικά πλεονεκτήματα. Διευκολύνεται σε μεγάλο βαθμό η ανάπτυξη νέων επεκτάσεων για τον πυρήνα, εφόσον αυτές μπορούν να δοκιμάζονται σε έναν πυρήνα που ήδη εκτελείται, χωρίς να απαιτείται ένας χρονοβόρος κύκλος αλλαγής του κώδικα – εκ νέου μεταγλώττιση ολόκληρου του πυρήνα - επανεκκίνηση του συστήματος με τον νέο πυρήνα. Ο νέος κώδικας μεταγλωττίζεται χωριστά, εισάγεται ως module, ελέγχεται και αν χρειάζονται επιπλέον αλλαγές αφαιρείται και μεταβάλλεται χωρίς να επηρεαστεί η λειτουργία του υπόλοιπου συστήματος.

Επιπλέον, εξοικονομείται κεντρική μνήμη και μειώνεται το μέγεθος του πυρήνα, εφόσον η λειτουργία του συστήματος ξεκινά με τους απολύτως απαραίτητους οδηγούς συσκευών ενσωματωμένους στον πυρήνα, ενώ οι οδηγοί για συσκευές που χρησιμοποιούνται σπάνια μπορούν να φορτώνονται όποτε υπάρξει ανάγκη χρήσης τους. Οι οδηγοί αυτοί παραμένουν στη μνήμη προσωρινά, φορτωμένοι ως modules και αφαιρούνται όταν δεν χρησιμοποιούνται.

Γίνεται λοιπόν φανερό ότι ο μηχανισμός των modules προσφέρει στο Linux πλεονεκτήματα τα οποία χαρακτηρίζουν κυρίως τα συστήματα που ακολουθούν τη σχεδιαστική επιλογή του μικροπυρήνα, όπως είναι η διάκριση του κώδικα σε ξεχωριστά τμήματα και η δομημένη οργάνωση του, διατηρώντας όμως το κύριο πλεονέκτημα μιας μονολιθικής σχεδίασης, που είναι ο ενιαίος χώρος διευθύνσεων πυρήνα. Αυτό γίνεται δυνατό λόγω του τρόπο με τον οποίο φορτώνεται ο κώδικας ενός kernel module στη μνήμη.

Κατά τη φόρτωση ενός module, που είναι στην ουσία αντικειμενικός κώδικας με αναφορές σε άλλα τμήματα του πυρήνα, γίνεται δυναμική σύνδεσή του με τον υπόλοιπο κώδικα. Χρησιμοποιώντας έναν πίνακα συμβόλων, ο οποίος περιέχει τις διευθύνσεις όλων των

βασικών συναρτήσεων και δομών δεδομένων που είναι δυνατόν να χρησιμοποιηθούν από κάποιο module, επιλύονται οι αναφορές που γίνονται από το module σε σημεία έξω από αυτό. Η διαδικασία αυτή γίνεται μόνο μία φορά, κατά την εισαγωγή του στο χώρο πυρήνα. Στη συνέχεια, ο κώδικας του module εκτελεί απλές κλήσεις συναρτήσεων και μεταβάλλει απευθείας τις δομές του πυρήνα, χωρίς την ανάγκη για χρονοβόρα ανταλλαγή μηνυμάτων. Επιπλέον, υπάρχει η δυνατότητα το ίδιο το module να εξάγει κάποια από τα σύμβολά του ώστε ένα επόμενο να μπορεί να συνδεθεί με αυτά. Έτσι, υλοποιείται μια στοίβα, και κάθε ένα από τα αρθρώματα που περιέχονται σε αυτή χρησιμοποιεί υπηρεσίες αυτού που βρίσκεται σε χαμηλότερο επίπεδο.

Το γενικό στρώμα επικοινωνίας που είναι το αντικείμενο αυτής της εργασίας υλοποιείται ως άρθρωμα του πυρήνα, ώστε να είναι δυνατή η δυναμική φόρτωση και απομάκρυνσή του από τη μνήμη. Το ίδιο συμβαίνει και με τον δοκιμαστικό οδηγό συσκευής που αναπτύχθηκε για τον έλεγχο του. Ωστόσο, όπως και με σχεδόν όλα τα modules, είναι δυνατό αυτά να μεταγλωττιστούν μαζί με τον υπόλοιπο κώδικα του πυρήνα και να ενσωματωθούν στο τελικό εκτελέσιμο που προκύπτει.

Συνοπτικά, η δομή του πυρήνα του Linux και η διάκριση των τμημάτων του ανάλογα με τη λειτουργία που επιτελούν παρουσιάζεται στο παρακάτω διάγραμμα:

Σχήμα 1.2 – Δομή του πυρήνα του Linux

1.4 Μηχανισμοί του πυρήνα του Linux

1.4.1 Επικοινωνία ανάμεσα στους χώρους πυρήνα και χρήστη

Η επικοινωνία ανάμεσα στις διεργασίες, που εκτελούνται στο χώρο χρήστη, και τον πυρήνα γίνεται μέσω ενός καθορισμένου συνόλου *πρωτογενών κλήσεων*, ή κλήσεων συστήματος. Μια διεργασία εκτελεί μια κλήση προς τον πυρήνα ώστε να χρησιμοποιήσει μια από τις διαθέσιμες λειτουργίες του, για παράδειγμα όταν επιθυμεί να πραγματοποιήσει Είσοδο / Έξοδο από μια περιφερειακή συσκευή. Ο πυρήνας αρχικά ελέγχει αν η ζητούμενη ενέργεια μπορεί να επιτραπεί για τη συγκεκριμένη διεργασία και στη συνέχεια, αν όλα είναι εντάξει, την πραγματοποιεί εκ μέρους της, στον χώρο πυρήνα. Αν η εκτέλεση της διαδικασίας ολοκληρωθεί με επιτυχία τα αποτελέσματα επιστρέφονται στον χώρο χρήστη, αλλιώς επιστρέφεται κάποιος κωδικός σφάλματος[1].

Η διαδικασία που ακολουθείται για την πραγματοποίηση μιας κλήσης συστήματος στο Linux ποικίλλει σημαντικά ανάλογα με την αρχιτεκτονική του υπολογιστικού συστήματος. Στην περίπτωση της αρχιτεκτονικής Intel x86, η διεργασία αρχικά αποθηκεύει στον καταχωρητή `%eax` τον αύξοντα αριθμό της κλήσης συστήματος που επιθυμεί να καλέσει[2] και στους καταχωρητές `%ebx`, `%ecx` και `%edx` τυχόν ορίσματα για αυτή και στη συνέχεια προκαλεί τη διακοπή λογισμικού 80h. Η ΚΜΕ συλλαμβάνει τη διακοπή και περνά τον έλεγχο στη διαδικασία χειρισμού της, η οποία εκτελείται στον χώρο πυρήνα. Η διαδικασία χειρισμού της διακοπής καλεί ανάλογα με την τιμή του `%eax` την κατάλληλη ρουτίνα του πυρήνα και αποθηκεύει την τιμή επιστροφής στον καταχωρητή `%ebx`. Τέλος, ο έλεγχος επιστρέφει τελικά στη διεργασία, αφού πρώτα η ΚΜΕ περάσει στην κατάσταση χρήστη.

Κατά την εξυπηρέτηση μιας διεργασίας από τον πυρήνα απαιτείται πολλές φορές η πρόσβαση σε δεδομένα που βρίσκονται αποθηκευμένα στον χώρο χρήστη, πχ. κατά την εξυπηρέτηση μιας κλήσης συστήματος `read` ή `write`. Η διεργασία περνά ως όρισμα έναν δείκτη στη θέση του χώρου εικονικής μνήμης απ' όπου πρόκειται να διαβαστούν ή όπου πρόκειται να γραφούν τα δεδομένα. Για λόγους ασφάλειας δεν επιτρέπεται η απευθείας

---

[1] Οι κωδικοί σφάλματος που είναι δυνατό να επιστραφούν στο Linux ορίζονται στο αρχείο /usr/src/linux/include/asm/errno.h

[2] Οι αύξοντες αριθμοί των κλήσεων συστήματος περιέχονται στο /usr/src/linux/include/asm/unistd.h

αναφορά μέσω του δείκτη σε περιεχόμενα της μνήμης, εφόσον είναι δυνατόν η διεργασία είτε λόγω προγραμματιστικού σφάλματος είτε λόγω κακόβουλου χρήστη να αναφέρεται σε δεδομένα που δεν ανήκουν σε αυτή. Αντίθετα, προσφέρεται μια σειρά από ρουτίνες του πυρήνα, οι οποίες αναλαμβάνουν την ασφαλή μεταφορά δεδομένων από και προς το χώρο χρήστη. Οι σημαντικότερες από αυτές είναι οι:

```
int copy_to_user(void *user_p, void *kernel_p, int n)
int copy_from_user(void *kernel_p, void *user_p, int n)
```

οι οποίες πραγματοποιούν την αντιγραφή `n` bytes δεδομένων ανάμεσα στο χώρο πυρήνα (δείκτης `kernel_p`) και τον χώρο χρήστη (δείκτης `user_p`). Οι δύο αυτές διαδικασίες εντοπίζουν άκυρες αναφορές, έξω από τον επιτρεπτό χώρο διευθύνσεων της διεργασίας, όποτε η χρήση τους είναι ασφαλής. Επιπλέον, αναλαμβάνουν το χειρισμό των *σφαλμάτων σελίδας* (page faults), την περίπτωση δηλαδή κατά την οποία τμήματα της εικονικής μνήμης της διεργασίας δεν βρίσκονται στην κεντρική μνήμη και πρέπει πρώτα να μεταφερθούν εκεί από κάποια περιφερειακή μονάδα, ώστε να είναι δυνατή η επεξεργασία τους.

1.4.2 Συγχρονισμός των διαδικασιών πυρήνα

Ο πυρήνας του Linux αναλαμβάνει να δημιουργήσει το κατάλληλο περιβάλλον εκτέλεσης για ένα σύνολο διεργασιών, έτσι ώστε να είναι δυνατή η εκτέλεσή τους συγχρόνως, χωρίς οι ενέργειες της μιας να επηρεάζουν τη λειτουργία της άλλης, παρά το γεγονός ότι συναγωνίζονται για τον χρόνο εκτέλεσης στην ΚΜΕ και τη χρήση των περιφερειακών μονάδων. Αυτό σημαίνει ότι κάποιες από τις λειτουργίες του πυρήνα είναι δυνατό να εκτελούνται συγχρόνως[3], για διαφορετικές διεργασίες. Επιπλέον, ο πυρήνας περιέχει τις ρουτίνες εξυπηρέτησης των διακοπών υλικού, οι οποίες ενεργοποιούνται σε απρόβλεπτες χρονικές στιγμές, στις οποίες δεν αποκλείεται να εκτελείται ήδη κάποιο άλλο τμήμα του κώδικα του πυρήνα.

Γίνεται φανερό ότι υπάρχουν δύο διαφορετικά αίτια για τα οποία μπορεί να κληθεί κάποιο τμήμα του πυρήνα: Το πρώτο είναι να εκτελείται εκ μέρους μιας διεργασίας, η οποία

[3] Σε συστήματα που διαθέτουν μόνο μία ΚΜΕ, ‘συγχρόνως’ θα πει ότι μοιράζονται το χρόνο εξυπηρέτησης της μονάδας, εφόσον μόνο ένα από αυτά είναι δυνατό να εκτελείται κάθε φορά

ζήτησε τις υπηρεσίες του λειτουργικού συστήματος. Η περίπτωση αυτή ονομάζεται εκτέλεση σε *περιβάλλον διεργασίας* (process context). Το δεύτερο αίτιο είναι ο κώδικας να εκτελείται ως αποτέλεσμα μιας διακοπής υλικού. Τότε, έχουμε εκτέλεση σε *περιβάλλον διακοπής* (interrupt context) και δεν έχει νόημα η αναφορά σε συγκεκριμένη (τρέχουσα) διεργασία, εφόσον κατά την έναρξη της ρουτίνας εξυπηρέτησης οποιαδήποτε διεργασία μπορεί να εξυπηρετείται στην ΚΜΕ.

Η ταυτόχρονη χρήση τμημάτων του πυρήνα δημιουργεί την ανάγκη οι διαδικασίες του να είναι *επανεισερχόμενες* (reentrant), δηλαδή να μπορούν να εκτελούνται ταυτόχρονα, εκ μέρους διαφορετικών διεργασιών, χωρίς οι ενέργειες από την εκτέλεση κάποιας από αυτές να επηρεάζουν τα αποτελέσματα από την εκτέλεση των υπολοίπων. Αυτό είναι δυνατό, μόνο αν τα τμήματα κώδικα που εκτελούνται παράλληλα μεταβάλλουν δεδομένα τοπικά για το καθένα από αυτά. Ωστόσο, αυτό σπάνια συμβαίνει, εφόσον οι ενέργειες του πυρήνα επηρεάζουν τις περισσότερες φορές κοινές δομές δεδομένων, στις οποίες αποθηκεύεται η κατάσταση του συστήματος και των υπό εκτέλεση διεργασιών. Έτσι, είναι απαραίτητη η προστασία των *κρίσιμων τμημάτων* (critical sections), των τμημάτων δηλαδή εκείνων που δρουν σε μοιραζόμενα δεδομένα, έτσι μόνο ένα από αυτά να εκτελείται σε δεδομένη χρονική στιγμή.

Για την ικανοποίηση της ανάγκης αυτής έγιναν κάποιες σχεδιαστικές επιλογές όσον αφορά τον τρόπο με τον οποίο εκτελείται ο κώδικας πυρήνα στο Linux, ενώ προσφέρονται επίσης διάφοροι τρόποι συγχρονισμού των τμημάτων του πυρήνα. Η πρώτη από αυτές τις επιλογές είναι ότι ο κώδικας πυρήνα είναι *μη διακοπτός* (non-preemptible). Αυτό σημαίνει ότι ενώ εκτελείται κώδικας πυρήνα στην ΚΜΕ, δεν καλείται ποτέ ο χρονοδρομολογητής του συστήματος για να διακόψει την εκτέλεσή του και να επιτρέψει σε κάποια άλλη διεργασία να χρησιμοποιήσει την ΚΜΕ. Η ρουτίνα εξυπηρέτησης για τη διακοπή χρονιστή, που είναι το κύριο μέσο για την επιβολή του καταμερισμού χρόνου, ενεργοποιείται, αλλά δεν προκαλεί την άμεση χρονοδρομολόγηση μιας νέας διεργασίας. Αντίθετα, τίθεται η σημαία need_resched, έτσι χρονοδρομολογητής να ενεργοποιηθεί όταν ολοκληρωθεί η εκτέλεση του κώδικα πυρήνα, πριν επιστραφούν τα αποτελέσματα της κλήσης συστήματος στην καλούσα διεργασία.

Από τη στιγμή που θα ξεκινήσει η εκτέλεση κώδικα πυρήνα σε περιβάλλον διεργασίας, αυτή θα συνεχιστεί χωρίς διακοπή έως ότου:

- Συμβεί διακοπή υλικού
- Συμβεί σφάλμα σελίδας
- Κληθεί (άμεσα ή έμμεσα) ο χρονοδρομολογητής

Η πρώτη περίπτωση (διακοπή υλικού) δεν δημιουργεί προβλήματα συνδρομικότητας, αρκεί να μην περιέχονται κρίσιμα τμήματα στη ρουτίνα εξυπηρέτησης της διακοπής. Στην περίπτωση που περιέχεται κρίσιμο τμήμα, ο κώδικας πυρήνα μπορεί να απενεργοποιεί προσωρινά το σύστημα των διακοπών, πριν εισέλθει στο κρίσιμο τμήμα του.

Η δεύτερη περίπτωση (σφάλμα σελίδας) ωστόσο μπορεί να αποτελέσει πηγή προβλημάτων. Ο χώρος εικονικής μνήμης του πυρήνα παραμένει φορτωμένος στην κεντρική μνήμη συνεχώς κι έτσι αποκλείεται το ενδεχόμενο μια αναφορά σε διεύθυνση που ανήκει στον χώρο πυρήνα να προκαλέσει σφάλμα σελίδας. Ωστόσο, αυτό δεν συμβαίνει και για τον χώρο εικονικής μνήμης που διατίθεται στις διεργασίες χρήστη. Οι σελίδες που ανήκουν στον χώρο χρήστη είναι δυνατό να απομακρυνθούν από την κεντρική μνήμη και να αποθηκευτούν σε κάποιο αρχείο αντιμετάθεσης, που βρίσκεται σε δευτερεύουσα μνήμη. Έτσι, κάθε πρόσβαση σε διεύθυνση που ανήκει στον χώρο εικονικής μνήμης μιας διεργασίας μπορεί να δημιουργήσει σφάλμα σελίδας, οπότε η τρέχουσα διεργασία απομακρύνεται από την ΚΜΕ («κοιμάται») μέχρι να ολοκληρωθούν οι απαραίτητες ανταλλαγές σελίδων και να την επιλέξει αργότερα για εκτέλεση ο χρονοδρομολογητής. Έτσι, κάθε χρήση των `copy_to_user` και `copy_from_user` μπορεί να προκαλέσει διακοπή της εκτέλεσης.

Η τρίτη περίπτωση αφορά την άμεση κλήση του χρονοδρομολογητή (ο κώδικας πυρήνα αφήνει εθελοντικά την ΚΜΕ) αλλά και την έμμεση κλήση του χρονοδρομολογητή, π.χ. διότι ο πυρήνας χρειάζεται να περιμένει την ολοκλήρωση μιας διαδικασίας Ε/Ε.

Διαπιστώνουμε λοιπόν, ότι το πρόβλημα του κρίσιμου τμήματος λύνεται με την απαίτηση ο κώδικας του πυρήνα να είναι μη διακοπτός, ενώ θα πρέπει επίσης να μην κάνει κάποια ενέργεια που ίσως οδηγήσει στην εκτέλεση του χρονοδρομολογητή, ενώ βρίσκεται μέσα σε κρίσιμο τμήμα.

Πρέπει να σημειώσουμε ότι, αντίθετα με την εκτέλεση σε περιβάλλον διεργασίας, ο κώδικας που εκτελείται σε περιβάλλον διακοπής δεν επιτρέπεται σε καμία περίπτωση να

κοιμηθεί. Αυτό οφείλεται στο ότι ο κώδικας αυτός δεν έχει κληθεί εκ μέρους συγκεκριμένης διαδικασίας, δεν συνδέεται με κάποιο συγκεκριμένο περιβάλλον διεργασίας και έτσι δεν μπορεί να υποστεί χρονοδρομολόγηση. Κατά συνέπεια, κατά την εκτέλεση σε περιβάλλον διακοπής δεν επιτρέπεται η πρόσβαση στο χώρο χρήστη, η κλήση του χρονοδρομολογητή, ή η πράξη P επάνω σε κάποιο σηματοφορέα.

1.4.3 Κλείδωμα με χρήση σηματοφορέων

Είδαμε ότι η επιλογή να μην διακόπτεται η εκτέλεση μιας διεργασίας όταν μια κλήση συστήματός της εξυπηρετείται σε κατάσταση πυρήνα είναι ένα καλό μέτρο για την αποφυγή προβλημάτων συνδρομικότητας. Τι θα συμβεί ωστόσο αν κώδικας του πυρήνα που βρίσκεται σε κρίσιμο τμήμα, σταματήσει προσωρινά να εκτελείται στην ΚΜΕ, ίσως γιατί θέλησε να προσπελάσει δεδομένα στο χώρο χρήστη; Στην περίπτωση αυτή είναι δυνατό, για όσο διάστημα η πρώτη διεργασία κοιμάται, να δοθεί η ευκαιρία σε μια δεύτερη διεργασία να ζητήσει την εκτέλεση της ίδιας κλήσης συστήματος οπότε θα μπει και αυτή στο κρίσιμο τμήμα, κάτι που θα είχε ανεπιθύμητα αποτελέσματα όσον αφορά την ακεραιότητα των δεδομένων και τη σταθερότητα του συστήματος.

Επιπλέον, η μέχρι τώρα ανάλυση εφαρμόζεται σε υπολογιστικά συστήματα τα οποία διαθέτουν μόνο μια ΚΜΕ, οπότε μπορεί να εκτελείται μόνο μία διεργασία κάθε φορά. Οι υπόλοιπες περιμένουν να επιλεγούν από το χρονοδρομολογητή για εκτέλεση. Στην περίπτωση όμως ενός συστήματος συμμετρικής πολυεπεξεργασίας (SMP) είναι δυνατό να έχουμε δύο η περισσότερες ΚΜΕ, ή οποίες λειτουργούν παράλληλα. Το Linux υποστηρίζει τέτοιου είδους υπολογιστικά συστήματα και μάλιστα, περισσότερες από μία ΚΜΕ μπορούν να εκτελούν κώδικα σε κατάσταση πυρήνα κάθε φορά. Το αποτέλεσμα είναι ότι η απαίτηση για μη διακοπτόμενη εκτέλεση κώδικα πυρήνα δεν είναι πλέον αρκετή και υπάρχει ανάγκη για νέους μηχανισμούς συγχρονισμού.

Ο ένας από τους δύο κύριους μηχανισμούς συγχρονισμού στον πυρήνα είναι οι γνωστοί μας *σηματοφορείς* (semaphores). Κάθε μεταβλητή σηματοφορέα ανήκει στον αδιαφανή τύπο `struct semaphore` και επιτρέπεται ο χειρισμός της μόνο μέσω συγκεκριμένων ρουτινών του πυρήνα, όπως είναι οι `sema_init`, `down` και `up`, οι οποίες του δίνουν αρχική τιμή, εκτελούν επάνω του την πράξη P, εκτελούν επάνω του την πράξη V αντίστοιχα. Επιπλέον, σε κάθε σηματοφορέα αντιστοιχεί μια ουρά αναμονής (`wait_queue`), στην οποία προστίθενται οι διεργασίες που μπλοκάρονται στη λειτουργία P και περιμένουν έως ότου

εκτελεστεί η λειτουργία V για να συνεχίσουν. Ο μηχανισμός των σηματοφορέων εξασφαλίζει το σωστό συγχρονισμό των διεργασιών ακόμη και κατά την εκτέλεση κώδικα πυρήνα ταυτόχρονα σε διαφορετικές ΚΜΕ ενός συστήματος συμμετρικής πολυεπεξεργασίας.

1.4.4 Κλείδωμα με χρήση spinlocks

Ο μηχανισμός των *spinlocks* παρέχει μια λιγότερο δαπανηρή λύση από πλευράς χρόνου εκτέλεσης στο πρόβλημα συγχρονισμού των διαδικασιών του πυρήνα, όταν αυτός εκτελείται σε μηχανές συμμετρικής πολυεπεξεργασίας. Τα spinlocks μοιάζουν με τους σηματοφορείς στο ότι είναι μοιραζόμενες μεταβλητές, για τις οποίες ορίζονται ρουτίνες κλειδώματος (`spin_lock`) και απελευθέρωσης (`spin_unlock`). Ωστόσο, σε αντίθεση με τους σηματοφορείς, τα spinlocks υλοποιούν το σχήμα «*απασχόλησης-αναμονής*» (busy-wait): Όταν ένα τμήμα κώδικα δεν μπορεί προσωρινά να κλειδώσει ένα spinlock διότι αυτό είναι ήδη κλειδωμένο από κάποιο άλλο τμήμα, δεν περιμένει σε μια ουρά αναμονής αλλά συνεχίζει να εκτελείται στην ΚΜΕ, προσπαθώντας, έως ότου καταφέρει να πάρει το κλείδωμα.

Τα spinlocks υπερέχουν σε σχέση με τους σηματοφορείς όταν τα κρίσιμα τμήματα είναι σύντομης διάρκειας, οπότε αποφεύγεται η χρονοβόρα διαδικασία της προσθήκης σε ουρά αναμονής και της κλήσης του χρονοδρομολογητή. Ωστόσο, πρέπει να παραμένουν δεσμευμένα για όσο το δυνατό μικρότερο χρονικό διάστημα, ώστε να αποφεύγεται η σπατάλη κύκλων στις ΚΜΕ του συστήματος και σε καμία περίπτωση δεν πρέπει ένα τμήμα κώδικα να κοιμηθεί ενώ κρατά κάποιο spinlock. Κάτι τέτοιο θα μπορούσε να οδηγήσει το σύστημα σε αδιέξοδο[4]. Η χρήση των spinlocks δεν έχει νόημα σε συστήματα που διαθέτουν μόνο μια ΚΜΕ, εφόσον τα προβλήματα συγχρονισμού που επιλύουν δεν εμφανίζονται ποτέ σε τέτοια συστήματα. Για το λόγο αυτό, όταν μεταγλωττίζεται ο πυρήνας του Linux για συστήματα ενός επεξεργαστή κάθε κλήση προς τις ρουτίνες χειρισμού των spinlocks παραλείπεται.

---

[4] Θεωρούμε τρεις διεργασίες Α, Β, Γ σε σύστημα με δύο ΚΜΕ. Η Α κοιμάται κρατώντας ένα spinlock. Όταν οι Β και Γ ξεκινήσουν να εκτελούνται στις δύο ΚΜΕ και προσπαθήσουν να κλειδώσουν το spinlock θα έχουμε αδιέξοδο, αφού θα αποτύχουν και δεν θα αφήσουν πότε τις ΚΜΕ (ο κώδικας πυρήνα δεν διακόπτεται) ώστε να συνεχίσει η Α και να το απελευθερώσει.

Κεφάλαιο 2

Linux και συσκευές χαρακτήρων

2.1 Κατηγορίες οδηγών συσκευών

Το Linux, ακολουθώντας τη σχεδιαστική φιλοσοφία του UNIX διακρίνει τρεις βασικές κατηγορίες συσκευών: συσκευές *χαρακτήρων*, συσκευές *block* και συσκευές *δικτύου*. Η κατηγορία στην οποία κατατάσσεται κάθε συσκευή υλικού, καθορίζει τον τρόπο με τον οποίο αντιμετωπίζεται ο οδηγός της από τον πυρήνα καθώς και τις υπηρεσίες που πρέπει κατ' ελάχιστο να υλοποιεί ο οδηγός, ώστε η συσκευή να μπορεί να χρησιμοποιηθεί από το σύστημα. Είναι ευθύνη του οδηγού της συσκευής να αποκρύψει τις λεπτομέρειες της χρήσης της και να παρουσιάσει μία αφαιρετική εικόνα της, η οποία εμπίπτει σε μία από τις τρεις κατηγορίες συσκευών.

Αναλυτικότερα, τα χαρακτηριστικά κάθε μίας από τις κύριες κατηγορίες συσκευών στο Linux είναι τα εξής:

- **Συσκευές χαρακτήρων:** Οι συσκευές χαρακτήρων μπορούν να προσπελαστούν ως ένα ρεύμα από χαρακτήρες. Ένας οδηγός συσκευής που υλοποιεί τη διαπροσωπεία για συσκευή χαρακτήρων, χειρίζεται κλήσεις συστήματος όπως οι `open`, `close`, `read` και `write`, οι οποίες επιτρέπουν τη μεταφορά ενός αριθμού από bytes από και προς τη συσκευή. Η αφηρημένη όψη της συσκευής χαρακτήρων ταιριάζει σε συσκευές υλικού όπως οι θύρες σειριακές και παράλληλης επικοινωνίας, το

τερματικό χρήστη, ή μια κάρτα ήχου. Οι συσκευές χαρακτήρα αναπαρίστανται από κόμβους στο σύστημα αρχείων (π.χ. `/dev/ttyS0` και `/dev/lp0` για την πρώτη σειριακή και παράλληλη θύρα, αντίστοιχα) και είναι δυνατή η προσπέλασή τους με τρόπο παρόμοιο με αυτόν που γίνεται προσπέλαση στα δεδομένα κανονικών αρχείων. Υπάρχει ωστόσο μια σημαντική διαφορά, ότι σε αντίθεση με ένα κανονικό αρχείο, το οποίο παρουσιάζεται ως μια περιοχή δεδομένων από την οποία μπορεί κανείς να ανακτήσει δεδομένα με τυχαία σειρά και να κινηθεί προς τα εμπρός η προς τα πίσω, μια συσκευή χαρακτήρων μπορεί τις περισσότερες φορές να προσπελαστεί μόνο σειριακά. Αυτό οφείλεται στο ότι ο τρόπος λειτουργίας της πραγματικής συσκευής υλικού δεν μπορεί να προσαρμοστεί σε μια αφαιρετική όψη που επιτρέπει την τυχαία προσπέλαση, οπότε ο οδηγός συσκευής δεν προσφέρει τις ανάλογες υπηρεσίες (υλοποίηση της κλήσης συστήματος `lseek`).

- **Συσκευές block:** Η βασική μονάδα πληροφορίας στις συσκευές block δεν είναι ο χαρακτήρας, το byte, αλλά το block, που συνήθως αποτελείται από έναν ορισμένο αριθμό χαρακτήρων. Έτσι η μεταφορά δεδομένων από και προς τον οδηγό που υλοποιεί τη διαπροσωπεία των συσκευών block γίνεται για ακέραιο αριθμό από blocks. Η βασική διαφορά ανάμεσα σε μια συσκευή χαρακτήρων και σε μια συσκευή block βρίσκεται στον τρόπο επικοινωνίας του οδηγού με τα υπόλοιπα στρώματα του πυρήνα. Ο οδηγός δεν υλοποιεί απευθείας λειτουργίες όπως οι `read` και `write`, αλλά ικανοποιεί αιτήσεις για εγγραφή ή ανάγνωση συγκεκριμένων block. Η ικανοποίηση των κλήσεων συστήματος `read` και `write` των διεργασιών γίνεται από ανώτερα στρώματα του πυρήνα, τα οποία αναλαμβάνουν την εξυπηρέτησή τους κάνοντας τις απαραίτητες αιτήσεις για block προς τον οδηγό συσκευής, χρησιμοποιώντας απομονωτές Ε/Ε και προσωρινή αποθήκευση των περιεχομένων των block για αύξηση της επίδοσης κατά την εκτέλεση λειτουργιών Ε/Ε. Όπως και οι συσκευές χαρακτήρα, έτσι και οι συσκευές block μπορούν να προσπελαστούν από διεργασίες χρήστη μέσω κόμβων του συστήματος αρχείων. Ωστόσο, μόνο οι συσκευές block παρέχουν τις κατάλληλες υπηρεσίες στον πυρήνα έτσι ώστε να είναι δυνατή η προσάρτηση (mount) ενός συστήματος αρχείων που περιέχουν σε κάποιο κατάλογο, έτσι ώστε τα αρχεία που περιέχει να γίνουν διαθέσιμα στις διεργασίες χρήστη. Οι συσκευές υλικού που μπορούν να προσαρμοστούν αποδοτικά στην αφαιρετική όψη μιας συσκευής block είναι κυρίως μονάδες

δευτερεύουσας μνήμης, όπως είναι οι μονάδες δισκέτας και οι σκληροί δίσκοι και τα CD-ROM.

- **Συσκευές δικτύου:** Αυτή η κατηγορία περιλαμβάνει συσκευές υλικού οι οποίες μπορούν να χρησιμοποιηθούν από το υποσύστημα δικτύου του πυρήνα και να ενσωματωθούν στη στοίβα πρωτοκόλλων δικτύου. Σε αντίθεση με τις δύο προηγούμενες κατηγορίες, δεν έχουν αναπαράσταση στο σύστημα αρχείων και δεν μπορούν να χρησιμοποιηθούν απευθείας από διεργασίες χρήστη. Οι οδηγοί συσκευών που ακολουθούν το μοντέλο των προσαρμογέων δικτύου παρέχουν τις κατάλληλες υπηρεσίες στα υπόλοιπα στρώματα του πυρήνα έτσι ώστε να είναι δυνατή η αποστολή και λήψη πακέτων πληροφορίας ανάμεσα στον πυρήνα και το υλικό. Η αφαιρετική όψη των συσκευών δικτύου ταιριάζει σε συσκευές υλικού όπως είναι οι προσαρμογείς E/E για δίκτυα Ethernet αλλά και σε εικονικές συσκευές όπως η συσκευή PPP, για ανταλλαγή πακέτων IP πάνω από σειριακούς συνδέσμους.

2.2 Διαπροσωπεία του πυρήνα προς οδηγούς συσκευών χαρακτήρων

2.2.1 Ο ρόλος του VFS

Κάθε οδηγός συσκευής χαρακτήρων δεν αλληλεπιδρά απευθείας με τις διεργασίες χρήστη, αλλά παρεμβάλλεται ανάμεσά τους το VFS, ένα στρώμα λογισμικού το οποίο χρησιμοποιείται για την πρόσβαση στους οδηγούς συσκευών μέσω ειδικών κόμβων στο σύστημα αρχείων. Το VFS αναλαμβάνει την προώθηση των αιτήσεων των διεργασιών στον οδηγό συσκευής. Όταν μιλάμε για διαπροσωπεία του πυρήνα προς οδηγούς συσκευών χαρακτήρων, στην ουσία εννοούμε διαπροσωπεία του VFS προς τους οδηγούς συσκευών. Για το λόγο αυτό, είναι χρήσιμη μια σύντομη παρουσίαση του ρόλου του VFS και του τρόπου λειτουργίας του.

Το VFS (Virtual File System - *εικονικό σύστημα αρχείων*), είναι ένα στρώμα λογισμικού το οποίο αποτελεί τμήμα του πυρήνα του Linux και διαχειρίζεται όλες τις κλήσεις συστήματος που αφορούν στο σύστημα αρχείων. Το Linux υποστηρίζει πρόσβαση σε πληθώρα συστημάτων αρχείων (μερικά από οποία είναι τα ext2, ReiserFS, VFAT, NTFS, ISO9660, UDF, NFS), τα οποία διαφέρουν σημαντικά μεταξύ τους ως προς τα ιδιαίτερα

χαρακτηριστικά τους, την εσωτερική δομή τους και τις δυνατότητες τους. Το VFS ορίζει μια κοινή διαπροσωπεία, την οποία οφείλουν να υλοποιούν όλα τα τμήματα κώδικα που υποστηρίζουν τα διάφορα συστήματα αρχείων, έτσι ώστε να είναι δυνατός ο ομοιόμορφος χειρισμός τους, αποκρύπτοντας τις ιδιαιτερότητες υλοποίησης του καθενός.

Η δομημένη αυτή σχεδίαση συμβάλλει στην ευελιξία και την επεκτασιμότητα του συστήματος. Για να προστεθεί υποστήριξη για ένα νέο σύστημα αρχείων χρειάζεται απλά να γραφεί κώδικας που αναλαμβάνει την πρόσβαση στις δομές δεδομένων του συστήματος αρχείων και τη μετάφρασή τους, την αναπαράστασή τους στις αφηρημένες δομές δεδομένων που χρησιμοποιούνται από το VFS. Όταν αυτό γίνει δυνατό, το νέο σύστημα αρχείων μπορεί να χρησιμοποιηθεί ακριβώς όπως τα προϋπάρχοντα, παρουσιάζοντας την ίδια εικόνα προς τις διεργασίες που εκτελούνται στο σύστημα.

Ο σχεδιασμός του VFS βασίζεται σε αντικειμενοστραφείς αρχές. Οι βασικές δομές - αντικείμενα που ορίζονται είναι οι δομές *inode* και *file*[5]. Κάθε μία από αυτές τις δομές χαρακτηρίζεται από ένα σύνολο υπηρεσιών, το οποίο πρέπει να υλοποιείται από τα κατώτερα στρώματα λογισμικού. Ένας πίνακας δεικτών που αποθηκεύεται σε κάθε αντικείμενο του VFS περιέχει τις διευθύνσεις των συναρτήσεων του πυρήνα οι οποίες αναλαμβάνουν τη διεκπεραίωση κάθε λειτουργίας. Χρησιμοποιώντας την καθιερωμένη ορολογία του αντικειμενοστραφούς προγραμματισμού, οι συναρτήσεις αυτές καλούνται *μέθοδοι* του αντικειμένου.

Η δομή inode χρησιμοποιείται για την αναπαράσταση ενός αρχείου ως σύνολο. Το VFS δεν ασχολείται με το αν το αρχείο είναι ένα κανονικό αρχείο αποθηκευμένο σε δευτερεύουσα μνήμη, ένα αρχείο προσβάσιμο μέσω ενός δικτυακού συστήματος αρχείων (π.χ. NFS), ή ίσως ένα ειδικό αρχείο συσκευής χαρακτήρων ή block. Σε κάθε περίπτωση καλεί την κατάλληλη συνάρτηση από τον πίνακα δεικτών του αντικειμένου, ανάλογα με το είδος της λειτουργίας που χρειάζεται να εκτελεστεί και ο έλεγχος περνά σε κάποιο κατώτερο στρώμα λογισμικού, το οποίο ικανοποιεί το αίτημα για το συγκεκριμένο είδος αρχείου, γνωρίζοντας τις λεπτομέρειες της υλοποίησης.

---

[5] Το VFS ορίζει αρκετά ακόμη αντικείμενα, όπως το αντικείμενο dentry, που αναπαριστά καταλόγους συστημάτων αρχείων, ωστόσο αυτά τα αντικείμενα δεν παρουσιάζονται εδώ εφόσον δεν χρησιμοποιούνται κατά την αλληλεπίδραση του VFS με τους οδηγούς συσκευών χαρακτήρων.

Η δομή file περιγράφει την αλληλεπίδραση μιας διεργασίας χρήστη με κάποιο *ανοιχτό* αρχείο. Ένα αρχείο είναι πιθανό να είναι ανοιχτό από πολλές διαφορετικές διεργασίες ταυτόχρονα, ή πολλές φορές από την ίδια διεργασία. Σε κάθε σημείο αλληλεπίδρασης διεργασίας - αρχείου αντιστοιχεί μια δομή file, η οποία περιγράφει τις ιδιότητες της πρόσβασης στο ανοιχτό αρχείο. Από τις σημαντικότερες πληροφορίες διαχείρισης που αποθηκεύονται σε μια δομή τύπου `struct file` είναι:

- Ο τρόπος πρόσβασης της διεργασίας στο ανοιχτό αρχείο (μόνο για ανάγνωση, μόνο για εγγραφή, πρόσβαση για ανάγνωση και εγγραφή)

- Η τρέχουσα θέση μέσα στο αρχείο, η θέση δηλαδή από την οποία ανακτώνται δεδομένα από μία κλήση read ή στην οποία αποθηκεύονται από μία κλήση write. Εάν επιτρέπεται η τυχαία πρόσβαση μέσα στο αρχείο, η θέση αυτή μπορεί να μεταβληθεί αυθαίρετα με χρήση της κλήσης συστήματος `lseek`.

- Ένας δείκτης σε δομή του τύπου `struct file_operations`, η οποία περιέχει δείκτες σε συναρτήσεις που υλοποιούν τις λειτουργίες Ε/Ε για το συγκεκριμένο αρχείο. Οι δείκτες αυτοί αποτελούν τα σημεία επικοινωνίας με τα στρώματα λογισμικού που βρίσκονται κάτω από το VFS, στα συμπεριλαμβάνονται οι οδηγοί συσκευών χαρακτήρων. Οι συναρτήσεις στις οποίες δείχνουν αυτοί οι δείκτες ονομάζονται μέθοδοι του αρχείου.

- Ένας δείκτης `private_data`, ο οποίος χρησιμοποιείται συνήθως για τη διατήρηση πληροφορίας κατάστασης (state information) από τα κατώτερα στρώματα λογισμικού. Ο τρόπος χρήσης του πεδίου αυτού και το είδος της πληροφορίας που αποθηκεύεται δεν προδιαγράφεται από το VFS, αλλά αφήνεται να επιλεγεί ελεύθερα από τις υλοποιήσεις των διάφορων συστημάτων αρχείων και τους οδηγούς των συσκευών.

Για κάθε διεργασία το VFS διατηρεί έναν πίνακα ανοικτών αρχείων, που αποτελείται από δομές τύπου `struct file`. Ο πίνακας αυτός αποθηκεύεται για λόγους ασφάλειας στο χώρο εικονικής μνήμης του πυρήνα και δεν είναι διαθέσιμος στον χώρο μνήμης της αντίστοιχης διεργασίας. Ο κώδικας του πυρήνα διαχειρίζεται τις δομές αυτές άμεσα, μέσω δεικτών, ενώ κάθε διεργασία χρήστη χρησιμοποιεί μη αρνητικούς ακεραίους για τη

δεικτοδότηση του πίνακα των ανοιχτών αρχείων της. Οι ακέραιοι αυτοί ονομάζονται *περιγραφητές αρχείων* (file descriptors).

2.2.2 Μείζων και ελάσσων αριθμός συσκευής – Το σύστημα αρχείων devfs

Όπως έχουμε ήδη αναφέρει, κάθε οδηγός συσκευής χαρακτήρων ή block είναι προσβάσιμος από τις διεργασίες χρήστη μέσω ειδικών κόμβων στο σύστημα αρχείων, οι οποίοι συνήθως βρίσκονται στον κατάλογο `/dev` (π.χ. `/dev/ttyS0` για την πρώτη σειριακή θύρα). Σε κάθε τέτοιο όνομα αρχείου αντιστοιχούν δύο θετικοί ακέραιοι, ο *μείζων* και ο *ελάσσων* αριθμός (major / minor number), οι οποίοι και καθορίζουν τον οδηγό συσκευής με τον οποίο συνδέεται το συγκεκριμένο ειδικό αρχείο και τη συσκευή στην οποία αυτό αντιστοιχεί. Χρησιμοποιώντας τον μείζονα αριθμό ο πυρήνας αναγνωρίζει αμέσως τον οδηγό συσκευής που αντιστοιχεί σε κάποιο ειδικό αρχείο, εφόσον κάθε μείζων αριθμός ανατίθεται σε μοναδικό οδηγό συσκευής[6]. Ο ελάσσων αριθμός δεν χρησιμοποιείται από το VFS αλλά περνά αμετάβλητος στον οδηγό συσκευής, ο οποίος συνήθως τον χρησιμοποιεί για να διακρίνει ανάμεσα σε πολλές διαφορετικές συσκευές του ίδιου τύπου που ίσως διαχειρίζεται. Για παράδειγμα ο μείζων αριθμός 7 όταν αναφέρεται σε ειδικό αρχείο τύπου χαρακτήρα αντιστοιχεί στον οδηγό των παράλληλων θυρών κάτω από το Linux. Όταν ο μείζων αριθμός είναι 7, ο ελάσσων αριθμός 0 καθορίζει την πρώτη παράλληλη θύρα, ο αριθμός 1 τη δεύτερη, κ.ο.κ.

Όταν χρησιμοποιούνται μείζονες για το διαχωρισμό ανάμεσα στους οδηγούς συσκευών, επιβάλλεται για κάθε συσκευή που πρόκειται να χρησιμοποιηθεί από διεργασίες χρήστη να έχει εκ τον προτέρων δημιουργηθεί ανάλογος κόμβος στο σύστημα αρχείων, που να χαρακτηρίζεται από τον κατάλληλο ζεύγος μείζονος - ελάσσονος αριθμού. Έτσι, δεν είναι σπάνιο ο κατάλογος /dev να περιέχει αρχεία τα οποία αντιστοιχούν σε συσκευές που δεν υπάρχουν στο σύστημα, αλλά δημιουργούνται κατά την εγκατάσταση του λειτουργικού συστήματος, έτσι ώστε να καλυφθούν οι πιο πιθανές περιπτώσεις συσκευών υλικού και οδηγών για τις συσκευές αυτές.

---

[6] Η διαδικασία με την οποία ανατίθενται μείζονες αριθμοί σε οδηγούς καθώς και ο κατάλογος των αριθμών που έχουν ήδη ανατεθεί περιέχεται στο αρχείο /usr/src/linux/Documentation/devices.txt

Στις τελευταίες εκδόσεις του πυρήνα προσφέρεται επιλεκτικά, ως εναλλακτικός του μηχανισμού που χρησιμοποιεί μείζονες αριθμούς, το σύστημα αρχείων συσκευών, ή device filesystem (devfs). Η χρήση του εικονικού αυτού συστήματος αρχείων και η προσάρτησή του κάτω από το σημείο `/dev` στο δέντρο των αρχείων επιτρέπει τη δυναμική δημιουργία και καταστροφή κόμβων στον κατάλογο `/dev`, έτσι ώστε οι διαθέσιμοι κόμβοι να ανταποκρίνονται ακριβώς στους οδηγούς συσκευών που είναι διαθέσιμοι ως τμήματα του πυρήνα. Κάθε φορά που κάποιος οδηγός συσκευής φορτώνεται στον πυρήνα ως module, δηλώνει τον εαυτό του στο devfs, έτσι ώστε να δημιουργηθούν κάτω από τον κατάλογο `/dev` οι κόμβοι που είναι απαραίτητοι για τη χρήση του από διεργασίες χρήστη. Κατά την απεγκατάσταση του module του οδηγού, όταν η χρήση του έχει ολοκληρωθεί, οι κόμβοι αυτοί αφαιρούνται από τον κατάλογο `/dev`. Οι περισσότεροι οδηγοί συσκευών υποστηρίζουν πλέον και τις δύο μεθόδους για τη δήλωσή και καταγραφή τους στον τον πυρήνα, έτσι ώστε να επιτρέπεται η χρήση τους χωρίς προβλήματα σε όλες τις περιπτώσεις.

Το devfs δεν σχετίζεται με το VFS. Το devfs αναλαμβάνει τη δυναμική δημιουργία των κόμβων που είναι απαραίτητοι για την επικοινωνία των διεργασιών με τους οδηγούς συσκευής ενώ το VFS αναλαμβάνει την προώθηση των κλήσεων συστήματος των διεργασιών στον κατάλληλο οδηγό όταν ο κόμβος στο σύστημα αρχείων έχει ήδη δημιουργηθεί και οι διεργασίες εκτελούν κλήσεις συστήματος που τον αφορούν.

2.2.3 Περιγραφή της δομής file_operations

Κάθε οδηγός συσκευής χαρακτήρα περιγράφεται μέσω μιας δομής τύπου `struct file_operations`. Η δομή αυτή αποτελείται από δείκτες στις συναρτήσεις εκείνες του οδηγού που υλοποιούν κλήσεις συστήματος όπως η `open`, η `read`, η `write` και άλλες που θα αναφερθούν στη συνέχεια. Έτσι, όταν το VFS κληθεί να εξυπηρετήσει μια κλήση συστήματος για ένα ανοιχτό αρχείο που αντιστοιχεί στον συγκεκριμένο οδηγό συσκευής, ακολουθεί το μέλος `f_ops` το οποίο περιέχεται στη δομή file και δείχνει προς την δομή file_operations του οδηγού. Έχοντας πλέον πρόσβαση στην δομή file_operations, το VFS βρίσκει τη διεύθυνση της κατάλληλης συνάρτησης και την καλεί. Εάν σε κάποιο πεδίο της δομής file_operations αποθηκεύεται η τιμή `NULL` της C, τότε ο οδηγός δεν παρέχει υλοποίηση της συγκεκριμένης κλήσης συστήματος και - ανάλογα με την κλήση - είτε αυτή αντικαθίσταται από κάποια γενική συνάρτηση του VFS, είτε δεν επιτρέπεται η χρήση της από διεργασίες. Τα σημαντικότερα πεδία του τύπου `struct file_operations` είναι τα εξής:

Δείκτης προς τη μέθοδο llseek: Η μέθοδος `llseek` αναλαμβάνει τη μετακίνηση του δείκτη ανάγνωσης / εγγραφής για το ανοιχτό αρχείο και υλοποιείται μόνο αν η μετακίνηση μέσα σε ανοιχτό αρχείο που αντιστοιχεί στον συγκεκριμένο οδηγό συσκευής χαρακτήρων έχει νόημα.

Δείκτης προς τη μέθοδο read: Η μέθοδος `read` αναλαμβάνει τη μεταφορά δεδομένων από τον οδηγό συσκευής στη καλούσα διεργασία χρήστη. Συνήθως χρησιμοποιεί μια ρουτίνα για αντιγραφή δεδομένων προς τον χώρο χρήστη όπως η `copy_to_user` για να πραγματοποιήσει τη μεταφορά.

Δείκτης προς τη μέθοδο write: Η μέθοδος `write` αναλαμβάνει τη μεταφορά δεδομένων από την καλούσα διεργασία στον οδηγό συσκευής. Συνήθως χρησιμοποιεί μια ρουτίνα για αντιγραφή δεδομένων από το χώρο χρήστη όπως η `copy_from_user` για να πραγματοποιήσει τη μεταφορά.

Δείκτης προς τη μέθοδο poll: Η μέθοδος `poll` χρησιμοποιείται ως το κατώτερο στρώμα λογισμικού για την εξυπηρέτηση των κλήσεων συστήματος select και poll από το VFS. Οι δύο αυτές κλήσεις συστήματος επιτρέπουν σε διεργασίες χρήστη να ελέγχουν την κατάσταση πολλών διαφορετικών συσκευών Ε/Ε ταυτόχρονα και να ενημερώνονται όταν είναι δυνατή η ανταλλαγή δεδομένων με κάποια από αυτές.

Δείκτης προς τη μέθοδο ioctl: Η μέθοδος `ioctl` χρησιμοποιείται για την εξυπηρέτηση της ομώνυμης κλήσης συστήματος. Η κλήση συστήματος `ioctl` χρησιμοποιείται ως το κύριο εργαλείο για τη αποστολή ειδικών εντολών προς τις συσκευές Ε/Ε από τις διεργασίες. Οι εντολές αυτές δεν είναι δυνατό να μεταφερθούν ως δεδομένα, μέσω των κλήσεων `read` και `write`, αλλά μάλλον μεταβάλλουν τον τρόπο με τον οποίο αντιμετωπίζονται τα δεδομένα αυτά από τον οδηγό της συσκευής. Για παράδειγμα, ένας οδηγός για μια κάρτα ήχου θα μπορούσε να επιτρέπει τον έλεγχο του ρυθμού δειγματοληψίας μέσω μιας εντολής ioctl. Στη συνέχεια, μια κλήση read θα είχε ως αποτέλεσμα είσοδο δεδομένων από κάποιον A/D μετατροπέα, ενώ μια κλήση `write` έξοδο δεδομένων σε κάποιο D/A μετατροπέα, με το ρυθμό που καθορίστηκε με χρήση της `ioctl`. Άλλο ένα παράδειγμα είναι ο οδηγός για μονάδες δισκέτας, ο οποίος προσφέρει τη δυνατότητα για μορφοποίηση μιας δισκέτας και σχηματισμό ιχνών επάνω της με χρήση ειδικών εντολών ioctl. Τέλος, ο οδηγός της σειριακής θύρας προσφέρει ειδικές ioctl εντολές για τον έλεγχο των παραμέτρων επικοινωνίας, όπως είναι η ταχύτητα επικοινωνίας ή η χρήση ψηφίου ισοτιμίας.

Δείκτης προς τη μέθοδο mmap: Η μέθοδος `mmap` χρησιμεύει για την απεικόνιση ενός χώρου μνήμης που ανήκει στον οδηγό συσκευής απευθείας στον χώρο εικονικής μνήμης της καλούσας διεργασίας. Έτσι είναι δυνατή η επικοινωνία της διεργασίας αυτής με τον οδηγό συσκευής μέσω εντολών ανάκλησης / αποθήκευσης σε θέσεις μνήμης του χώρου εικονικής μνήμης της.

Δείκτης προς τη μέθοδο open: Η μέθοδος `open` καλείται όταν μια διεργασία χρήστη επιχειρήσει να ανοίξει κάποιο ειδικό αρχείο που αντιστοιχεί στον οδηγό συσκευής. Συνήθως αναλαμβάνει την αρχικοποίηση της συσκευής υλικού και θέτει κατάλληλη τιμή στο πεδίο `private_data` της δομής file.

Δείκτης προς τη μέθοδο release: Η μέθοδος `release` καλείται όταν η αντίστοιχη δομή file πρόκειται να καταστραφεί από το VFS, εφόσον ζητήθηκε από τη διεργασία χρήστη το κλείσιμο του ανοιχτού αρχείου.

Ο ρόλος των παραπάνω πεδίων θα γίνει φανερός στα επόμενα κεφάλαια, όπου περιγράφεται η χρήση τους κατά την υλοποίηση του στρώματος επικοινωνίας που αποτελεί το αντικείμενο αυτής της εργασίας.

Επίσης, η δομή file_operations περιέχει κάποια ακόμα πεδία τα οποία δεν εξετάστηκαν παραπάνω, εφόσον πολύ σπάνια χρησιμεύουν κατά την ανάπτυξη ενός οδηγού συσκευής χαρακτήρων και δεν χρησιμοποιήθηκαν στην εργασία αυτή.

2.2.4 Ένα παράδειγμα χρήσης οδηγού συσκευής μέσω του VFS

Ολοκληρώνοντας την παρουσίαση του VFS και του τρόπου επικοινωνίας των οδηγών συσκευής με το υπόλοιπο του πυρήνα, μπορούμε να δούμε αναλυτικά πως μπορεί μια διεργασία που εκτελείται κάτω από το Linux να χρησιμοποιήσει το ειδικό αρχείο /dev/lp0 για να στείλει έναν αριθμό από χαρακτήρες σε εκτυπωτή που συνδέεται με τον υπολογιστή μέσω της πρώτης παράλληλης θύρας:

Αρχικά η διεργασία καλεί την κλήση συστήματος open για το ειδικό αρχείο `/dev/lp0`, ζητώντας πρόσβαση εγγραφής. Το VFS αναλαμβάνει την εξυπηρέτηση της κλήσης. Ως

πρώτο βήμα, αναζητά τη δομή inode που αντιστοιχεί στο αρχείο `/dev/lp0`[7]. Όταν την εντοπίσει, ξεκινά μια διαδικασία ελέγχου ασφάλειας, κατά την οποία επαληθεύεται ότι η διεργασία έχει δικαίωμα ανοίγματος του ειδικού αρχείου για εγγραφή. Αν το αποτέλεσμα του ελέγχου είναι θετικό, το VFS δημιουργεί μια νέα δομή file, που αναπαριστά το ανοιχτό αρχείο `/dev/lp0` για τη διεργασία. Εφόσον η δομή inode για το `/dev/lp0` περιέχει την πληροφορία ότι αυτό είναι ειδικό αρχείο με ζεύγος μείζονος - ελάσσονος αριθμού (7, 0), το VFS εντοπίζει τον οδηγό συσκευής χαρακτήρα με τον μείζονα αριθμό 7 (είναι ο οδηγός της παράλληλης θύρας) και θέτει τον κατάλληλο δείκτη της δομής file έτσι ώστε να οδηγεί στη δομή file_operations του οδηγού της παράλληλης θύρας. Τέλος, ακολουθείται ο δείκτης προς τη μέθοδο open, ο οποίος περιέχεται στη file_operations, και εκτελείται η ανάλογη συνάρτηση, έτσι ώστε να ενημερωθεί ο οδηγός συσκευής, να πάρει τον έλεγχο της πρώτης παράλληλης θύρας (εφόσον ο ελάσσων αριθμός είναι ο 0) και να αρχικοποιήσει το πεδίο `private_data`.

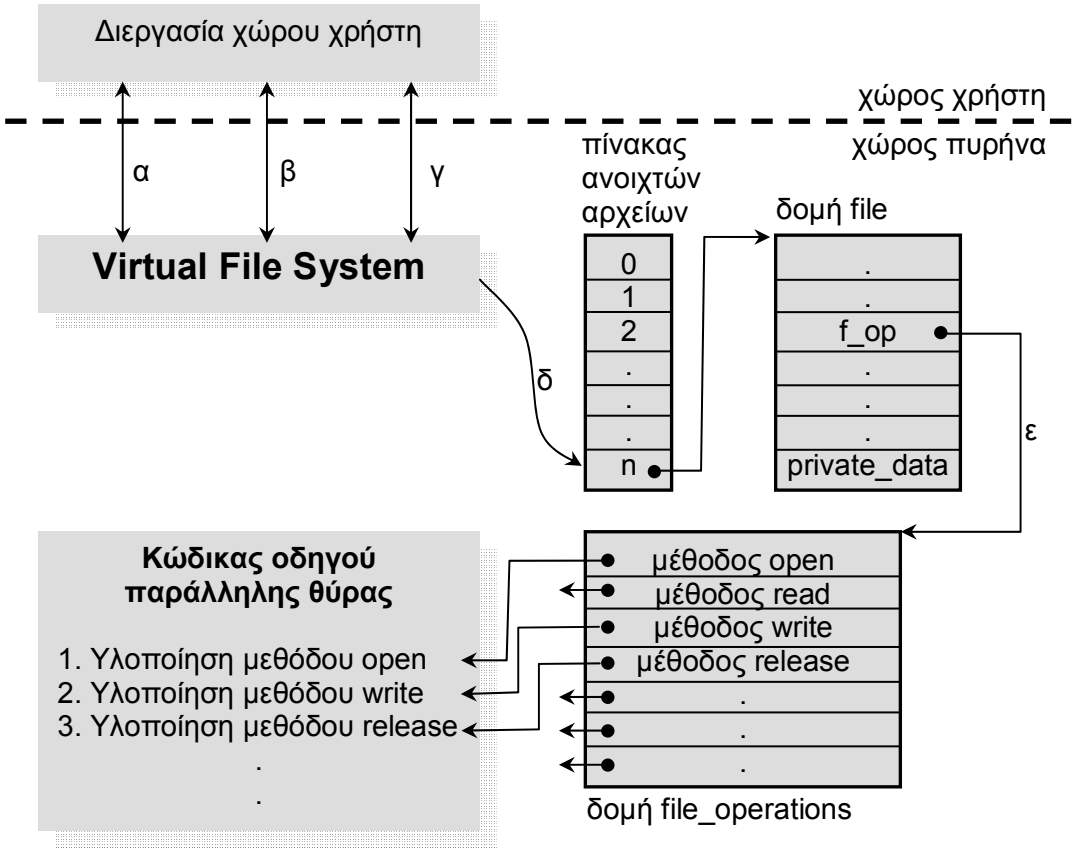
Στη συνέχεια η διεργασία καλεί την κλήση συστήματος `write` για να ζητήσει την εγγραφή των χαρακτήρων στο ανοιχτό αρχείο, άρα και την αποστολή τους στην παράλληλη θύρα. Το VFS εντοπίζει τη δομή file που αντιστοιχεί στο ανοιχτό αρχείο, αναζητώντας το στον πίνακα ανοιχτών αρχείων της διεργασίας, ακολουθεί το δείκτη στη δομή file_operations του οδηγού, εντοπίζει τη μέθοδο `write` και την καλεί. Η μέθοδος write του οδηγού αναλαμβάνει την αντιγραφή των χαρακτήρων από το χώρο εικονικής μνήμης της διεργασίας και την αποστολή της μέσω εντολών Ε/Ε στην παράλληλη θύρα.

Τέλος, η διεργασία καλεί την κλήση συστήματος close για να ζητήσει το κλείσιμο του ανοικτού αρχείου. Εφόσον η δομή file που αντιστοιχεί σε αυτό πρόκειται να καταστραφεί, εντοπίζεται και καλείται, όμοια με τα προηγούμενα, η μέθοδος `release` του οδηγού συσκευής. Όταν ολοκληρωθεί η εκτέλεσή της, το VFS καταστρέφει τη δομή file και την αφαιρεί από τον πίνακα ανοικτών αρχείων της διεργασίας.

Οι διαδικασίες που περιγράφηκαν παραπάνω, παρουσιάζονται σχηματικά στο παρακάτω διάγραμμα:

---

[7] Το αντίστοιχο αντικείμενο inode και οι ιδιότητές του επιστρέφονται από το σύστημα αρχείων στο οποίο περιέχονται τα αρχεία του /dev, άρα και το ειδικό αρχείο /dev/lp0



Σχήμα 2.1 – Κλήση μεθόδων οδηγού συσκευής από το VFS

Η σημασία των γραμμάτων που διακρίνουν τα βέλη του σχήματος είναι: **(α)(β)(γ)** χρήση των κλήσεων συστήματος `open`, `write` και `close` **(δ)** εύρεση της κατάλληλης δομής file από τον πίνακα ανοιχτών αρχείων της διεργασίας με βάση των περιγραφητή αρχείου **(ε)** το VFS ακολουθεί το δείκτη προς τη δομή file_operations που περιέχεται στο πεδίο `f_op` της δομής file ώστε να εντοπίσει και να καλέσει την κατάλληλη μέθοδο του οδηγού.

Κεφάλαιο 3

Σχεδιασμός του γενικού στρώματος επικοινωνίας KSockets

3.1 Εισαγωγή – Γενική περιγραφή

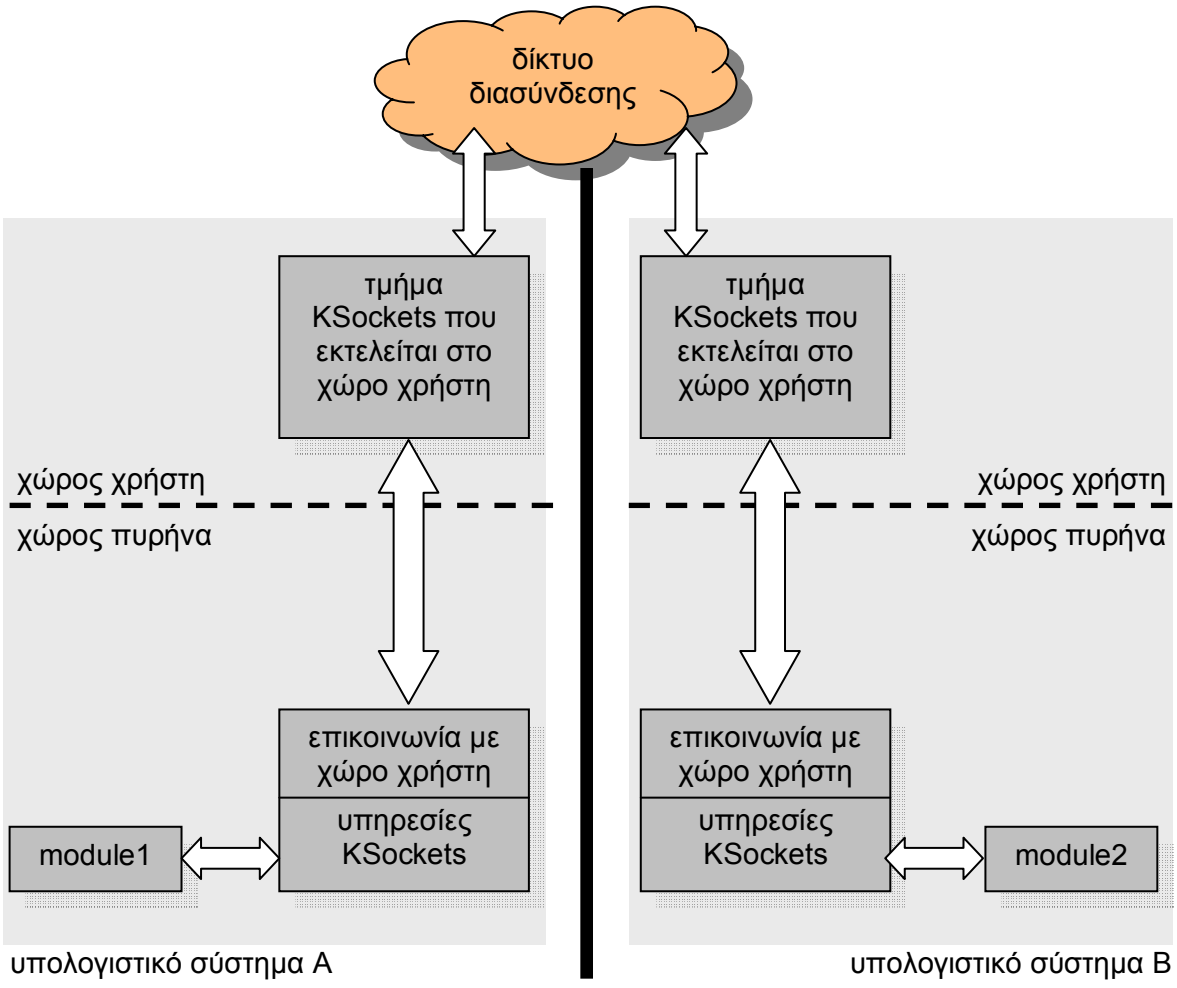
Έχοντας παρουσιάσει τα βασικά χαρακτηριστικά του πυρήνα του Linux και διάφορους από τους μηχανισμούς που προσφέρονται για προγραμματισμό σε επίπεδο πυρήνα, μπορούμε να προχωρήσουμε στην παρουσίαση του σχεδιασμού και της υλοποίησης του στρώματος επικοινωνίας KSockets.

Ο κύριος λόγος δημιουργίας των KSockets είναι η ανάγκη για επικοινωνία ανάμεσα σε πυρήνες του Linux που εκτελούνται σε υπολογιστικά συστήματα τα οποία αποτελούν μέρος μιας συστοιχίας υπολογιστών. Η επικοινωνία ανάμεσά τους γίνεται μέσω ενός εξελιγμένου δικτύου διασύνδεσης υψηλών επιδόσεων, όπως είναι το SCI. Οι χρήστες του γενικού στρώματος επικοινωνίας το οποίο σχεδιάζουμε είναι τμήματα του πυρήνα τα οποία συνήθως φορτώνονται στον εικονικό χώρο μνήμης του ως modules, ενώ και το ίδιο το στρώμα επικοινωνίας είναι επιθυμητό να έχει τη δυνατότητα να χρησιμοποιηθεί ως module.

Ο σχεδιασμός του στρώματος επικοινωνίας είναι προσαρμοσμένος στην ανάγκη χρήσης του από κώδικα που εκτελείται στο χώρο πυρήνα. Ωστόσο, το ιδιαίτερο χαρακτηριστικό του είναι ότι το ίδιο δεν υλοποιείται εξ ολοκλήρου ως τμήμα του πυρήνα του Linux, αλλά ως συνδυασμός κώδικα που εκτελείται σε χώρο πυρήνα και κώδικα που εκτελείται στο

χώρο χρήστη, ως μια πολυνηματική διεργασία. Οι ανάγκες που οδήγησαν σε αυτή τη σχεδιαστική επιλογή, καθώς και τα προτερήματα και τα μειονεκτήματα που συνεπάγεται, αναλύονται διεξοδικά στην παράγραφο 3.3.

Στο ακόλουθο διάγραμμα παρουσιάζεται συνοπτικά ο ρόλος των KSockets στην ανταλλαγή δεδομένων ανάμεσα σε δύο αρθρώματα module1 και module2, τα οποία αποτελούν τμήματα δύο πυρήνων Linux που εκτελούνται σε διαφορετικά υπολογιστικά συστήματα μιας συστοιχίας υπολογιστών. Η επικοινωνία ανάμεσά τους γίνεται με τρόπο ανεξάρτητο του χρησιμοποιούμενου δικτύου διασύνδεσης:



Σχήμα 3.1 - Ρόλος των KSockets στην επικοινωνία ανάμεσα σε δύο πυρήνες

Όπως φαίνεται και στο παραπάνω σχήμα, τα KSockets αποτελούνται από τρία διακριτά μεταξύ τους τμήματα:

- Το πρώτο τμήμα υλοποιεί τη διαπροσωπεία προς τα στρώματα του πυρήνα που χρησιμοποιούν τις υπηρεσίες των KSockets

- Το δεύτερο τμήμα αναλαμβάνει την επικοινωνία ανάμεσα στο χώρο πυρήνα και το χώρο χρήστη, έτσι ώστε να δημιουργηθεί μια οδός μεταφοράς δεδομένων μέσω του χώρου χρήστη

- Το τρίτο τμήμα εκτελείται ως διεργασία κάτω από το λειτουργικό σύστημα και διαχειρίζεται την επικοινωνία μέσω του δίκτυου διασύνδεσης

Τα δύο πρώτα τμήματα εκτελούνται στο χώρο πυρήνα, ενώ το τρίτο στο χώρο χρήστη.

3.2 Προδιαγραφές των απαιτήσεων από το στρώμα επικοινωνίας

Στη συνέχεια, προδιαγράφονται οι απαιτήσεις που οφείλει να ικανοποιεί το στρώμα επικοινωνίας KSockets, έτσι ώστε να είναι δυνατή η χρήση του από τα υπόλοιπα υποσυστήματα που εκτελούνται σε χώρο πυρήνα με αξιόπιστο και αποδοτικό τρόπο. Οι προδιαγραφές που παρουσιάζονται στη συνέχεια λαμβάνονται υπόψη κατά το σχεδιασμό των KSockets και εξηγούν αρκετές από τις επιλογές που έγιναν κατά τη διάρκειά του.

Προδιαγραφή 1: Τα KSockets θα είναι ένα γενικό στρώμα επικοινωνίας, προσαρμοσμένο για χρήση από υποσυστήματα του πυρήνα του Linux. Χρήστες του θα είναι τμήματα κώδικα που εκτελούνται στο χώρο πυρήνα.

Προδιαγραφή 2: Το στρώμα επικοινωνίας θα επιτρέπει την αξιόπιστη ανταλλαγή δεδομένων ανάμεσα σε υποσυστήματα πυρήνων που εκτελούνται σε διαφορετικά υπολογιστικά συστήματα. Η επικοινωνία θα είναι βασισμένη σε αμφίδρομες συνδέσεις, οι οποίες θα εγκαθίστανται και θα λύονται με χρήση κατάλληλων ρουτινών των KSockets. Στο κάθε άκρο της σύνδεσης θα βρίσκεται ακριβώς ένας πυρήνας. Όταν έχει εγκατασταθεί μία σύνδεση ανάμεσα σε δύο πυρήνες, τα δεδομένα που πρόκειται να ανταλλαγούν θα μεταφέρονται από και προς τον κάθε πυρήνα ως μια συνεχής ροή από bytes, έτσι ώστε η σύνδεση να μπορεί να χρησιμοποιηθεί ως ένα ρεύμα δεδομένων.

Προδιαγραφή 3: Το στρώμα επικοινωνίας θα επιτρέπει την ταυτόχρονη εγκατάσταση και λειτουργία πολλών διαφορετικών συνδέσεων ανάμεσα σε δύο πυρήνες. Κάθε σύνδεση πρέπει να είναι εντελώς ανεξάρτητη από τις υπόλοιπες. Το στρώμα επικοινωνίας πρέπει να παρέχει κατάλληλους μηχανισμούς ώστε να είναι δυνατή η διάκριση ανάμεσά τους και η αναφορά σε κάθε μία από αυτές ξεχωριστά. Επιπλέον, αναλαμβάνει την *πολυπλεξία* των συνδέσεων επάνω στο δίκτυο διασύνδεσης.

Για να είναι δυνατή η διάκριση ανάμεσα σε πολλές ταυτόχρονες συνδέσεις που είναι εγκατεστημένες προς τον ίδιο πυρήνα, χρησιμοποιείται ο μηχανισμός των *θυρών επικοινωνίας* (ports). Οι θύρες επικοινωνίας αποτελούν τα σημεία από τα οποία μπορεί ένας πυρήνας να ξεκινήσει μία νέα εξερχόμενη σύνδεση, ή τα σημεία στα οποία μπορεί να δεχθεί μια νέα εισερχόμενη σύνδεση. Είναι αριθμημένες και βρίσκονται στα άκρα των συνδέσεων που εγκαθίστανται με χρήση των KSockets. Χρησιμοποιούνται κατά την εγκατάσταση μιας σύνδεσης, έτσι ώστε να είναι δυνατή η διάκριση ανάμεσα σε διάφορα υποσυστήματα του ίδιου πυρήνα, τα οποία μπορεί να είναι έτοιμα να δεχτούν εισερχόμενες συνδέσεις. Ο αριθμός της θύρας επικοινωνίας καθορίζει το υποσύστημα με το οποίο πρόκειται να ανταλλαγούν δεδομένα.

Επιπλέον, η προδιαγραφή αυτή επιβάλλει τη χρήση κατάλληλων μηχανισμών συγχρονισμού κατά την υλοποίηση των KSockets, έτσι ώστε η ταυτόχρονη χρήση των συνδέσεων να είναι ασφαλής και να προστατεύεται η ακεραιότητα των μοιραζόμενων δεδομένων.

Προδιαγραφή 4: Κάθε ένας από τους πυρήνες που χρησιμοποιούν τις υπηρεσίες του στρώματος επικοινωνίας θα πρέπει να μπορεί να διακρίνεται από τους υπολοίπους.

Η διάκριση ανάμεσα τους επιτυγχάνεται μέσω της απόδοσης στον καθένα ενός *μοναδικού ονόματος*. Το όνομα αυτό είναι ανεξάρτητο από τον τρόπο διάκρισης των κόμβων στο επίπεδο του δικτύου διασύνδεσης. Έτσι, είναι απαραίτητο ένα ενδιάμεσο βήμα αντιστοίχισης του ονόματος κάθε πυρήνα σε κατάλληλη διεύθυνση για το δίκτυο διασύνδεσης που χρησιμοποιείται κάθε φορά. Η αντιστοιχία αυτή καθορίζεται από το διαχειριστή της συστοιχίας υπολογιστών και χρησιμοποιείται εσωτερικά από το στρώματος επικοινωνίας, ενώ δεν γίνεται αντιληπτή από τα τμήματα του πυρήνα που χρησιμοποιούν τις υπηρεσίες του.

Προδιαγραφή 5: Το στρώμα επικοινωνίας θα παρέχει μια καθορισμένη διαπροσωπεία στους χρήστες των υπηρεσιών του, μέσω της οποίας θα είναι δυνατή η εγκατάσταση και λύση συνδέσεων, καθώς και η ανταλλαγή δεδομένων χρησιμοποιώντας ήδη εγκατεστημένες συνδέσεις. Η διαπροσωπεία αυτή θα είναι ανεξάρτητη του δικτύου διασύνδεσης που χρησιμοποιείται για την επικοινωνία ανάμεσα στα υπολογιστικά συστήματα που αποτελούν τη συστοιχία υπολογιστών. Το σύνολο των συναρτήσεων που

προσφέρονται για χρήση από τα υπόλοιπα υποσυστήματα του πυρήνα παρουσιάζεται αναλυτικά στην παράγραφο 3.4.

Προδιαγραφή 6: Το στρώμα επικοινωνίας θα έχει τη δυνατότητα υποστήριξης πολλών διαφορετικών δικτύων διασύνδεσης, αποκρύπτοντας από τους χρήστες του τις ιδιαιτερότητες του καθενός. Η διατήρηση της αφηρημένης όψης του δικτύου επικοινωνίας, η οποία επιτρέπει εγκατάσταση αξιόπιστων συνδέσεων ανεξάρτητα από τις πραγματικές δυνατότητες του δικτύου διασύνδεσης και του μοντέλου επικοινωνίας που αυτό ακολουθεί, αποτελεί ευθύνη του στρώματος επικοινωνίας.

Η υλοποίηση που παρουσιάζεται αργότερα υποστηρίζει επικοινωνία με χρήση του δικτύου διασύνδεσης SCI, ή εναλλακτικά με χρήση δικτύων βασισμένων στο πρωτόκολλο TCP/IP. Στα δίκτυα αυτά συμπεριλαμβάνονται τα FastEthernet, GigabitEthernet αλλά και οποιοδήποτε άλλο δίκτυο προσφέρει υποστήριξη για TCP/IP, όπως είναι το Myrinet με τη χρήση του ανάλογου λογισμικού.

Ο σχεδιασμός του στρώματος επικοινωνίας διακρίνει το τμήμα εκείνο που εξαρτάται από το δίκτυο διασύνδεσης και επιτρέπει την αντικατάστασή του, ώστε να είναι δυνατή η επέκταση των KSockets και η υποστήριξη διαφορετικών δικτύων διασύνδεσης.

Προδιαγραφή 7: Σε περίπτωση αποτυχίας κατά την εξυπηρέτηση αίτησης προς το στρώμα επικοινωνίας, το υποσύστημα του πυρήνα που έκανε το αίτημα πρέπει να ενημερώνεται μέσω της επιστροφής κατάλληλου κωδικού σφάλματος. Οι κωδικοί σφάλματος που επιστρέφονται από την υλοποίηση ανήκουν στο σύνολο των καθιερωμένων κωδικών σφάλματος που υποστηρίζονται από το Linux[8]. Σε περίπτωση επιτυχίας επιστρέφεται η τιμή μηδέν.

Προδιαγραφή 8: Το στρώμα επικοινωνίας θα συμμορφώνεται με τις απαιτήσεις που θέτει το λειτουργικό σύστημα Linux για τον προγραμματισμό σε επίπεδο πυρήνα και θα συμβαδίζει με τις σχεδιαστικές αρχές του πυρήνα που παρουσιάστηκαν στα δύο προηγούμενα κεφάλαια.

---

[8] Όλοι οι κωδικοί σφάλματος, μαζί με σύντομη λεκτική περιγραφή τους βρίσκονται στο αρχείο /usr/src/linux/include/asm/errno.h

Προδιαγραφή 9: Το στρώμα επικοινωνίας θα διατηρεί τα ιδιαίτερα χαρακτηριστικά επίδοσης του χρησιμοποιούμενου δικτύου διασύνδεσης.

Τα σύγχρονα δίκτυα διασύνδεσης για συστοιχίες υπολογιστών χαρακτηρίζονται από υψηλούς ρυθμούς διαμεταγωγής και χαμηλούς χρόνους αρχικής απόκρισης (latency). Το στρώμα επικοινωνίας οφείλει να διατηρεί τα ιδιαίτερα αυτά χαρακτηριστικά κατά τη λειτουργία του και να ελαχιστοποιεί την καθυστέρηση και τη μείωση του ρυθμού διαμεταγωγής που εισάγεται από τη χρήση του ανάμεσα στα υπόλοιπα υποσυστήματα του πυρήνα και το δίκτυο διασύνδεσης.

3.3 Συνδυασμός κώδικα χώρου πυρήνα και κώδικα χώρου χρήστη

3.3.1 Πλεονεκτήματα της σχεδίασης

Η υλοποίηση του στρώματος επικοινωνίας ως συνδυασμού κώδικα χώρου πυρήνα και διεργασίας χώρου χρήστη και η κατανομή των λειτουργιών του ανάμεσα τους, προσφέρει ορισμένα πλεονεκτήματα σε σχέση με μια υλοποίηση που λειτουργεί εξ ολοκλήρου στο χώρο πυρήνα. Τα σημαντικότερα από αυτά είναι:

3.3.1.1 Προστασία μνήμης

Σε αντίθεση με κώδικα που εκτελείται στο χώρο πυρήνα, ένα πρόγραμμα που εκτελείται ως διεργασία χρήστη έχει πρόσβαση μόνο στο δικό του χώρο εικονικής μνήμης. Πιθανή δυσλειτουργία του δεν μεταβάλλει την κατάσταση των υπόλοιπων διεργασιών και δεν είναι δυνατό να επηρεάσει τη σταθερότητα του συστήματος. Η εκτέλεση μιας απαγορευμένης λειτουργίας στο τμήμα του στρώματος επικοινωνίας που εκτελείται στο χώρο χρήστη θα οδηγήσει στον ασφαλή τερματισμό του από τον πυρήνα. Στη συνέχεια, ο διαχειριστής του συστήματος μπορεί να διερευνήσει τους λόγους τερματισμού του και να επανεκκινήσει τη διεργασία χρήστη, ώστε να συνεχιστεί ομαλά η λειτουργία του στρώματος επικοινωνίας.

3.3.1.2 Ευελιξία στη φάση της ανάπτυξης

Κατά την ανάπτυξη του τμήματος που εκτελείται σε χώρο χρήστη, είναι διαθέσιμος μεγάλος αριθμός προγραμματιστικών εργαλείων που επιταχύνουν και διευκολύνουν την ανάπτυξη του κώδικα και την ανεύρεση προγραμματιστικών λαθών. Σε αυτά συμπεριλαμβάνονται debuggers, συστήματα ελέγχου ορίων κατά την πρόσβαση στη μνήμη

(memory bounds checkers) καθώς και αναλυτές της λειτουργίας του προγράμματος (profilers).

3.3.1.3 Δυνατότητα χρήσης βιβλιοθηκών χώρου χρήστη

Η υλοποίηση ενός τμήματος του στρώματος επικοινωνίας ως διεργασία χρήστη επιτρέπει τη χρήση μεγάλου αριθμού βιβλιοθηκών κώδικα, όπως ακριβώς θα έκανε οποιοδήποτε πρόγραμμα εκτελείται στο χώρο χρήστη. Ολόκληρη η καθιερωμένη βιβλιοθήκη της C είναι άμεσα διαθέσιμη, όπως και βιβλιοθήκες δημιουργίας πολυνηματικών εφαρμογών, αλλά και βιβλιοθήκες για κρυπτογράφηση και συμπίεση των δεδομένων. Η επέκταση του στρώματος επικοινωνίας και η προσθήκη νέων λειτουργιών σε αυτό διευκολύνεται, εφόσον πολλές από τις απαιτούμενες υπηρεσίες προσφέρονται ήδη από τις βιβλιοθήκες αυτές και δεν είναι απαραίτητη η προσαρμογή τους και ο εκτεταμένος έλεγχος της σωστής λειτουργίας τους έτσι ώστε να μπορούν να εκτελεστούν στο χώρο πυρήνα.

Επιπλέον, η διεργασία χρήστη μπορεί να χρησιμοποιήσει βιβλιοθήκες του δίκτυου διασύνδεσης οι οποίες προσφέρουν ένα απλούστερο μοντέλο πρόσβασης στο δίκτυο και διευκολύνουν σημαντικά την ανταλλαγή δεδομένων πάνω από αυτό. Αυτό είναι ιδιαίτερα σημαντικό για την περίπτωση του δικτύου διασύνδεσης SCI: Η βιβλιοθήκη *SISCI* (Software Infrastructure for SCI), προσφέρει σημαντική ευελιξία στις εφαρμογές χρήστη και απλοποιεί τη χρήση του δικτύου διασύνδεσης. Επιπλέον, οι υπηρεσίες της είναι διαθέσιμες στις εφαρμογές χρήστη μέσω καθορισμένης διαπροσωπείας, επαρκώς τεκμηριωμένης και σύμφωνης με συγκεκριμένες προδιαγραφές. Αντίθετα, δεν υπάρχει καθορισμένη διαπροσωπεία για πρόσβαση στο SCI από τον χώρο πυρήνα, αλλά απαιτείται σημαντική προσπάθεια και γνώση των λεπτομερειών της υλοποίησης. Ο λεπτομέρειες αυτές αποκρύπτονται από το SISCI, όταν η πρόσβαση γίνεται μέσω του χώρου χρήστη. Το γεγονός αυτό, σε συνδυασμό με την ανάγκη για υποστήριξη του SCI από το στρώμα επικοινωνίας ήταν ένας από τους σημαντικότερους λόγους υλοποίησης των KSockets μέσω χώρου χρήστη.

Σε αντίθεση με την ευελιξία που προσφέρεται κατά την σχεδίαση προγραμμάτων που εκτελούνται στο χώρο χρήστη, απαιτείται πολύ προσεκτική επιλογή των λειτουργιών που επιλέγονται να εκτελεστούν στο χώρο πυρήνα. Ο κώδικας πυρήνα πρέπει να είναι αυτοτελής και δεν έχει τη δυνατότητα χρήσης βιβλιοθηκών. Πολλές από τις υπηρεσίες που θεωρούνται δεδομένες για το χώρο χρήστη (π.χ. χειρισμός συμβολοσειρών (strings), μορφοποιημένη Ε/Ε αριθμητικών και άλλων τιμών) υλοποιούνται εξ αρχής σε επίπεδο

πυρήνα. Άλλες, όπως λειτουργίες κινητής υποδιαστολής δεν επιτρέπεται καν να χρησιμοποιηθούν[9], με αποτέλεσμα να δυσχεραίνεται η πραγματοποίηση διαδικασιών όπως η κρυπτογραφία και η συμπίεση, οι οποίες πρέπει να υλοποιούνται με χρήση αποκλειστικά της Αριθμητικής Λογικής Μονάδας. Το σημαντικότερο όμως στοιχείο που δυσχεραίνει την εισαγωγή νέων λειτουργιών σε κώδικα που εκτελείται σε χώρο χρήστη είναι η ανάγκη εξαντλητικού ελέγχου του, εφόσον ολόκληρο το σύστημα μπορεί να καταρρεύσει σε περίπτωση δυσλειτουργίας του.

3.3.2 Μειονεκτήματα της σχεδίασης

3.3.2.1 Συχνότερη αντιγραφή δεδομένων

Το τμήμα χώρου χρήστη μπορεί να προσπελάσει άμεσα όλες τις δομές δεδομένων που χρειάζεται να εξετάσει κατά τη λειτουργία του. Αντίθετα, ο περιορισμός των διεργασιών χρήστη σε ιδιωτικό χώρο εικονικής μνήμης και η αδυναμία προσπέλασης και μεταβολής δεδομένων που βρίσκονται στο χώρο πυρήνα επιβάλλει την ύπαρξη ενός καθορισμένου τρόπου ανταλλαγής μηνυμάτων ανάμεσα στους δύο χώρους και οδηγεί στην ανάγκη συχνότερης αντιγραφής δεδομένων από το χώρο πυρήνα στο χώρο χρήστη και αντίστροφα. Η δημιουργία αντιγράφων αφενός οδηγεί σε λιγότερο αποδοτική χρήση της μνήμης του συστήματος, αφετέρου αυξάνει το χρόνο απόκρισης κατά την επικοινωνία μέσω των KSockets.

3.3.2.2 Συχνές μεταγωγές περιεχομένου

Κατά τη χρήση ενός στρώματος επικοινωνίας του οποίου οι λειτουργίες κατανέμονται ανάμεσα στον χώρο πυρήνα και το χώρο χρήστη, απαιτούνται συχνές μεταγωγές περιεχομένου (context switches) από και προς τη διεργασία χρήστη που αναλαμβάνει τη διαχείριση του δικτύου διασύνδεσης. Ο χρόνος που απαιτείται για να ολοκληρωθεί η μεταγωγή περιεχομένου προσαυξάνει στο ακέραιο το χρόνο απόκρισης κατά την επικοινωνία μέσω των KSockets, εφόσον οι μεταγωγές περιεχομένου βρίσκονται επάνω

[9] Η χρήση της Μονάδας Κινητής Υποδιαστολής (FPU) στο χώρο πυρήνα δεν επιτρέπεται, για να εξασφαλιστεί η μεταφερσιμότητα του κώδικα, αφού ορισμένες αρχιτεκτονικές στις οποίες έχει μεταφερθεί ο πυρήνας το Linux δεν διαθέτουν ξεχωριστή FPU. Ακόμη όμως και σε αυτές που διαθέτουν FPU, δεν υπάρχει καν πρόβλεψη για αποθήκευση της κατάστασής της πριν ξεκινήσει η εκτέλεση κώδικα πυρήνα, ώστε να είναι δυνατή η ομαλή επιστροφή από την κλήση συστήματος στον κώδικα της διεργασίας χρήστη.

στο *κρίσιμο μονοπάτι* (critical path) που ακολουθείται κατά την ανταλλαγή μηνυμάτων από τα υποσυστήματα των πυρήνων που επικοινωνούν.

Αξίζει να σημειώσουμε ότι ο χρονοδρομολογητής του Linux προσφέρει έναν από τους μικρότερους χρόνους για την ολοκλήρωση μιας μεταγωγής περιεχομένου σε σύγκριση με άλλα λειτουργικά συστήματα που ακολουθούν το μοντέλο του Unix, της τάξης των 5-7μsec όπως προκύπτει από μετρήσεις στην αρχιτεκτονική x86 της Intel [12]. Ωστόσο, αυτό το χρονικό διάστημα είναι συγκρίσιμο με το χρόνο απόκρισης που προσφέρει το δίκτυο SCI για εγγραφή σε κατανεμημένη μνήμη, που είναι της τάξης των 5-15μsec.

3.3.3 Κατανομή λειτουργιών στους χώρου πυρήνα και χρήστη

Ο τρόπος κατανομής των λειτουργιών των KSockets ανάμεσα στα τμήματα χώρου πυρήνα και χώρου χρήστη επηρεάζει αποφασιστικά την απόδοση του στρώματος επικοινωνίας. Επιλέγουμε σε χώρο πυρήνα να γίνονται μόνο οι απαραίτητες λειτουργίες διαχείρισης των συνδέσεων και επικοινωνίας με το χώρο χρήστη, ενώ η διαχείριση των πόρων του δικτύου διασύνδεσης, η εγκατάσταση και λύση συνδέσεων και η πραγματική επικοινωνία ανάμεσα σε διαφορετικά υπολογιστικά συστήματα αποτελούν ευθύνη της διεργασίας χρήστη.

Αναλυτικότερα, το τμήμα που εκτελείται σε χώρο πυρήνα:

- Προδιαγράφει και υλοποιεί ένα συγκεκριμένο πρωτόκολλο ανταλλαγής δεδομένων με τη διεργασία χώρου χρήστη
- Δέχεται αιτήσεις για εξυπηρέτηση των χρηστών του στρώματος επικοινωνίας και τις προωθεί προς το χώρο χρήστη
- Διατηρεί πληροφορίες διαχείρισης των συνδέσεων από την πλευρά του πυρήνα
- Εξασφαλίζει την ακεραιότητα των δεδομένων κατά την ταυτόχρονη χρήση συνδέσεων από διαφορετικά τμήματα του πυρήνα, μέσω των κατάλληλων μηχανισμών συγχρονισμού

Το τμήμα που εκτελείται ως διεργασία χώρου χρήστη:

- Διαχειρίζεται τους πόρους του δικτύου διασύνδεσης

- Προβάλλει μια αφηρημένη εικόνα του δικτύου διασύνδεσης προς το χώρο πυρήνα, ανεξάρτητα από τις πραγματικές δυνατότητες που αυτό προσφέρει

- Ικανοποιεί αιτήσεις εγκατάσταση και λύση των συνδέσεων προς άλλα υπολογιστικά συστήματα

- Μεταφέρει δεδομένα πάνω από εγκατεστημένες συνδέσεις, από το χώρο πυρήνα προς το δίκτυο διασύνδεσης και αντίστροφα

3.4 Προσφερόμενες υπηρεσίες του στρώματος επικοινωνίας

Οι υπηρεσίες των KSockets προσφέρονται στα υπόλοιπα τμήματα του πυρήνα του Linux μέσω ενώ συνόλου συναρτήσεων της C. Οι συναρτήσεις αυτές αναλαμβάνουν τις διαδικασίες εγκατάστασης συνδέσεων, τερματισμού συνδέσεων και μεταφοράς δεδομένων από και προς τον πυρήνα. Κάθε μία από αυτές δίνει τη δυνατότητα στο καλούν τμήμα κώδικα να καθορίσει τι επιθυμεί να συμβεί κατά την περίπτωση στην οποία το αίτημά του δεν μπορεί για κάποιο λόγο να ικανοποιηθεί άμεσα, αλλά χρειάζεται να περιμένει έως ότου συμβεί ένα συγκεκριμένο γεγονός. Για παράδειγμα, μία αίτηση ανάγνωσης δεδομένων δεν είναι άμεσα ικανοποιήσιμη αν δεν υπάρχουν διαθέσιμα δεδομένα από την άλλη πλευρά. Κατά την κλήση της συνάρτησης για ανάγνωση δεδομένων πρέπει να καθορίζεται αν στην περίπτωση αυτή η συνάρτηση θα μπλοκάρει, οπότε ο έλεγχος θα επιστραφεί στο καλούν τμήμα κώδικα μόνο όταν ληφθούν νέα δεδομένα από την άλλη πλευρά, ή αν η συνάρτηση θα επιστρέψει αμέσως με τον κωδικό σφάλματος `EAGAIN`. Η τιμή επιστροφής `EAGAIN` σημαίνει ότι το αίτημα δεν είναι δυνατό να ικανοποιηθεί άμεσα από το στρώμα επικοινωνίας και πρέπει να επαναληφθεί σε κάποια μεταγενέστερη χρονική στιγμή.

3.4.1 Ο τύπος δεδομένων ksocket

Ο τύπος δεδομένων *ksocket* αναπαριστά ένα σημείο αλληλεπίδρασης με το στρώμα επικοινωνίας. Χρησιμοποιείται ως όρισμα των συναρτήσεων που παρουσιάζονται στη συνέχεια, ώστε να είναι δυνατή η ύπαρξη πολλών ταυτόχρονων συνδέσεων με διάφορους πυρήνες, μία για κάθε δομή ksocket. Η δομή του αναλύεται στο επόμενο κεφάλαιο, όπου παρουσιάζεται αναλυτικά ο σχεδιασμός και η υλοποίηση του τμήματος των KSockets που εκτελείται στο χώρο πυρήνα.

3.4.2 Η συνάρτηση ksocket_create

Η συνάρτηση `ksocket_create` χρησιμοποιείται για τη δημιουργία μιας νέας δομής ksocket, ώστε αυτή να μπορεί να χρησιμοποιηθεί κατά την αλληλεπίδραση με το στρώμα επικοινωνίας. Δεσμεύει χώρο στη μνήμη του συστήματος για τη νέα δομή και την αρχικοποιεί. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_create(ksocket **p);
```

Εάν η συνάρτηση επιτύχει στη δημιουργία της νέας δομής τότε επιστρέφεται η τιμή μηδέν και η τιμή του `p` μεταβάλλεται ώστε να δείχνει στη δομή που μόλις δημιουργήθηκε, αλλιώς επιστρέφεται κατάλληλος κωδικός σφάλματος (συνήθως `ENOMEM`, αποτυχία δέσμευσης χώρου στη μνήμη).

3.4.3 Η συνάρτηση ksocket_open

Η συνάρτηση `ksocket_open` ανοίγει ένα νέο κανάλι επικοινωνίας με τη διεργασία χώρου χρήστη για χρήση με τη δομή ksocket που δέχεται ως όρισμα, έτσι ώστε να είναι δυνατή αργότερα η εγκατάσταση νέας σύνδεσης. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_open(ksocket *p, int nonblock);
```

όπου η δομή στην οποία δείχνει ο `p` πρέπει να έχει εκ των προτέρων δημιουργηθεί με χρήση της `ksocket_create`. Αν το όρισμα `nonblock` είναι μη μηδενικό, δεν επιτρέπεται στη διαδικασία να μπλοκάρει αν το αίτημα δεν είναι δυνατό να ικανοποιηθεί άμεσα.

3.4.4 Η συνάρτηση ksocket_bind

Η συνάρτηση `ksocket_bind` δεσμεύει μια θύρα επικοινωνίας του στρώματος KSockets για χρήση με τη δομή ksocket που δέχεται ως όρισμα. Η χρήση της είναι απαραίτητη ώστε να μπορεί ο πυρήνας να δεχτεί νέα σύνδεση στη συγκεκριμένη θύρα επικοινωνίας, ή να εγκαταστήσει μια νέα σύνδεση από τη θύρα αυτή. Η δήλωσή της είναι:

```
int ksocket_bind(ksocket *p, int port, int nonblock);
```

όπου port η επιθυμητή θύρα επικοινωνίας. Στην περίπτωση που αυτή δεν είναι διαθέσιμη, διότι έχει ήδη δεσμευθεί για χρήση με κάποια άλλη δομή ksocket, επιστρέφεται η τιμή σφάλματος `EBUSY` (Device or resource busy), ενώ αν είναι εκτός ορίων η τιμή `EINVAL` (Input value out of range). Για το όρισμα `nonblock`, όπως συμβαίνει με όλες τις

συναρτήσεις του στρώματος επικοινωνίας, ισχύουν όσα αναφέρθηκαν προηγούμενα. Η δομή ksocket που χρησιμοποιείται ως όρισμα πρέπει προηγουμένως να έχει συνδεθεί με το χώρο χρήστη μέσω της συνάρτησης `ksocket_open`. Αν αυτό δεν έχει συμβεί επιστρέφεται η τιμή `EINVAL`.

3.4.5 Η συνάρτηση ksocket_accept

Η συνάρτηση `ksocket_accept` χρησιμοποιεί τη δομή ksocket που δέχεται ως όρισμα για να δεχτεί μια σύνδεση στη θύρα επικοινωνίας που πρέπει προηγουμένως να έχει δεσμευτεί για χρήση μέσω της `ksocket_bind`. Αν δεν έχει ήδη κληθεί η `ksocket_bind` επιστρέφεται η τιμή `EINVAL`. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_accept(ksocket *p, int nonblock);
```

Αν το όρισμα `nonblock` είναι μηδενικό, τότε η συνάρτηση μπλοκάρει και επιστρέφει μόνο όταν γίνει δεκτή μια νέα εισερχόμενη σύνδεση ή αν συμβεί κάποιο σφάλμα. Αν το όρισμα nonblock είναι μη μηδενικό, τότε η συνάρτηση επιστρέφει άμεσα, με την τιμή επιστροφής τον κωδικό σφάλματος `EINPROGRESS` (Operation in progress) και είναι ευθύνη του καλούντος να ελέγξει την κατάσταση της δομής ksocket σε κάποια μεταγενέστερη χρονική στιγμή για διαπιστώσει αν έχει γίνει δεκτή μια νέα σύνδεση, ή συνέβη κάποιο σφάλμα.

3.4.6 Η συνάρτηση ksocket_connect

Η συνάρτηση `ksocket_connect` χρησιμοποιεί τη δομή ksocket που δέχεται ως όρισμα για να εγκαταστήσει μια νέα εξερχόμενη σύνδεση μέσω της θύρας επικοινωνίας που πρέπει προηγουμένως να έχει δεσμευτεί για χρήση μέσω της `ksocket_bind`. Αν δεν έχει ήδη κληθεί η `ksocket_bind` επιστρέφεται η τιμή `EINVAL`. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_connect(ksocket *p, const char *kernel_name, int
remote_port, int nonblock);
```

όπου το όρισμα `kernel_name` είναι το όνομα του απομακρυσμένου πυρήνα προς τον οποίο επιθυμούμε να εγκαταστήσουμε τη νέα σύνδεση και `remote_port` η θύρα επικοινωνίας του. Αν ο απομακρυσμένος πυρήνας δεν περιμένει ήδη μια νέα σύνδεση στη θύρα επικοινωνίας `remote_port` έχοντας εκτελέσει την κλήση `ksocket_accept`, επιστρέφεται η τιμή σφάλματος `ECONNREFUSED` (Connection refused). Αν το όρισμα nonblock είναι μη μηδενικό, η συνάρτηση `ksocket_connect` επιστρέφει αμέσως με την τιμή σφάλματος `EINPROGRESS`

και είναι ευθύνη του καλούντος να ελέγξει την κατάσταση της δομής ksocket αργότερα για να διαπιστώσει αν η σύνδεση εγκαταστάθηκε με επιτυχία.

3.4.7 Η συνάρτηση ksocket_read

Η συνάρτηση `ksocket_read` χρησιμοποιείται για την λήψη δεδομένων από έναν απομακρυσμένο πυρήνα μέσω μιας σύνδεσης προς αυτόν που πρέπει να έχει ήδη εγκατασταθεί με χρήση της `ksocket_connect` ή της `ksocket_accept`. Αν δεν υπάρχει εγκατεστημένη σύνδεση επιστρέφεται ο κωδικός σφάλματος `ENOTCONN` (Transport endpoint not connected). Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
ssize_t ksocket_read(ksocket *p, char *buf, int is_userspace_buf,
size_t n, int nonblock);
```

Η συνάρτηση `ksocket_read` ακολουθεί τη σημασιολογία της κλήσης συστήματος `read` των Linux/Unix:

- Αν υπάρχουν διαθέσιμα δεδομένα, τότε αυτά επιστρέφονται στο χώρο μνήμης που δείχνει ο δείκτης `buf`. Το όρισμα `n` αποτελεί τον μέγιστο αριθμό bytes που το καλούν τμήμα είναι διατεθειμένο να δεχτεί στο χώρο μνήμης `buf`. Έτσι λαμβάνονται το πολύ `n` bytes, ωστόσο η συνάρτηση είναι ελεύθερη να επιστρέψει οποιοδήποτε αριθμό από bytes από 1 έως και `n`. Αν επιστραφούν λιγότερα από `n` bytes και το τμήμα του πυρήνα που έκανε την κλήση χρειάζεται να λάβει ακριβώς `n` bytes, τότε μπορεί να επαναλάβει την κλήση, ώστε να λάβει και τα υπόλοιπα. Η τιμή επιστροφής της συνάρτησης είναι ο αριθμός των bytes που έλαβε και αποθήκευσε στο `buf`.

- Αν δεν υπάρχουν διαθέσιμα δεδομένα κατά τη χρονική στιγμή εκτέλεσης της `ksocket_read` τότε, αν το όρισμα `nonblock` είναι μηδενικό, η συνάρτηση μπλοκάρει έως ότου ληφθούν νέα δεδομένα. Αν το όρισμα `nonblock` είναι μη μηδενικό, τότε η συνάρτηση επιστρέφει αμέσως με τον κωδικό σφάλματος `EAGAIN`, οπότε το τμήμα κώδικα που έκανε την κλήση πρέπει να ξαναπροσπαθήσει αργότερα, οπότε ίσως έχουν αποσταλεί δεδομένα από τον απομακρυσμένο πυρήνα.

- Αν δεν υπάρχουν διαθέσιμα δεδομένα και ο απομακρυσμένος πυρήνας έχει ζητήσει τον τερματισμό της σύνδεσης, τότε επιστρέφεται η τιμή μηδέν.

Το όρισμα `is_userspace_buf` χρησιμεύει ώστε να είναι δυνατή λήψη δεδομένων σε θέση μνήμης που βρίσκεται στο χώρο χρήστη. Πολλές φορές το τμήμα κώδικα που καλεί την `ksocket_read` χρειάζεται να αποθηκεύσει τα δεδομένα που λαμβάνονται σε χώρο μνήμης που έχει διατεθεί σε διεργασία χρήστη. Στην περίπτωση αυτή, για να αποφευχθεί η άσκοπη αντιγραφή των δεδομένων πρώτα στο χώρο πυρήνα και στη συνέχεια στο χώρο χρήστη, τίθεται το όρισμα `is_userspace_buf` σε μη μηδενική τιμή και τα δεδομένα αποθηκεύονται απευθείας στο χώρο χρήστη. Αυτό επιτυγχάνεται με χρήση από την ksocket_read της ρουτίνας `copy_to_user`.

3.4.8 Η συνάρτηση ksocket_write

Η συνάρτηση `ksocket_write` χρησιμοποιείται για την αποστολή δεδομένων προς ένα απομακρυσμένο πυρήνα μέσω μιας σύνδεσης προς αυτόν, η οποία πρέπει να έχει ήδη εγκατασταθεί με χρήση της `ksocket_connect` ή της `ksocket_accept`. Αν δεν υπάρχει εγκατεστημένη σύνδεση επιστρέφεται ο κωδικός σφάλματος `ENOTCONN` (Transport endpoint not connected). Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
ssize_t ksocket_write(ksocket *p, const char *buf, int
is_userspace_buf, size_t n, int nonblock);
```

Η συνάρτηση `ksocket_write` ακολουθεί τη σημασιολογία της κλήσης συστήματος `write` των Linux/Unix:

- Αν είναι δυνατή η αποστολή δεδομένων τη συγκεκριμένη χρονική στιγμή, η συνάρτηση είναι ελεύθερη να αποστείλει οποιονδήποτε αριθμό από 1 έως `n` bytes, ξεκινώντας από τη θέση μνήμης στην οποία δείχνει ο δείκτης `buf`. Αν το καλούν τμήμα κώδικα του πυρήνα επιθυμεί την αποστολή ακριβώς `n` bytes, τότε είναι υποχρεωμένο να επαναλάβει την κλήση για τα υπόλοιπα. Η τιμή επιστροφής της `ksocket_write` είναι ο αριθμός των bytes που απέστειλε.

- Αν δεν είναι δυνατή η αποστολή δεδομένων, π.χ. διότι έχει γεμίσει κάποιος απομονωτής εξόδου, τότε η συνάρτηση μπλοκάρει, αν το όρισμα `nonblock` είναι μηδενικό, αλλιώς επιστρέφει την τιμή σφάλματος `EAGAIN`.

- Αν ο απομακρυσμένος πυρήνας έχει ζητήσει τον τερματισμό της σύνδεσης και δεν είναι πλέον δυνατή η αποστολή δεδομένων προς αυτόν επιστρέφεται ο κωδικός σφάλματος `EPIPE` (Broken pipe).

Για το όρισμα `is_userspace_buf` ισχύουν όσα αναφέρθηκαν προηγούμενα για τη συνάρτηση `ksocket_read`.

3.4.9 Η συνάρτηση ksocket_close

Η συνάρτηση `ksocket_close` χρησιμοποιείται όταν η επικοινωνία με τη χρήση ενός ksocket έχει ολοκληρωθεί και επιθυμείται ο τερματισμός της σύνδεσης. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_close(ksocket *p, int nonblock);
```

Η χρήση της επιτρέπεται και όταν δεν υπάρχει ήδη εγκατεστημένη σύνδεση με χρήση του ksocket, οπότε απλά κλείνει το κανάλι επικοινωνίας με το χώρο χρήστη.

3.4.10 Η συνάρτηση ksocket_release

Η συνάρτηση `ksocket_release` καλείται όταν δεν πρόκειται να χρησιμοποιηθεί ξανά η δομή ksocket που δέχεται ως όρισμα, οπότε πρέπει να καταστραφεί και να χαρακτηριστεί ως διαθέσιμος ο χώρος μνήμης που κατείχε. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_release(ksocket *p);
```

Είναι ευθύνη του καλούντος προγράμματος να καλέσει την `ksocket_release` μόνο όταν δεν υπάρχει περίπτωση να χρησιμοποιηθεί πλέον η συγκεκριμένη δομή ksocket. Η χρήση της δομής ksocket έπειτα από κλήση της `ksocket_release` αποτελεί προγραμματιστικό σφάλμα και έχει απρόβλεπτες συνέπειες στη σταθερότητα του πυρήνα.

3.4.11 Η συνάρτηση ksocket_reset

Η συνάρτηση `ksocket_reset` επιτρέπει εκ νέου την χρήση μιας δομής ksocket που έχει ήδη χρησιμοποιηθεί για την επικοινωνία με απομακρυσμένους πυρήνες. Η κλήση της γίνεται σύμφωνα με τη δήλωση:

```
int ksocket_reset(ksocket *p);
```

Η συνάρτηση `ksocket_reset` επαναφέρει τη δομή ksocket στην αρχική κατάσταση, όπως ακριβώς ήταν όταν κλήθηκε αρχικά η `ksocket_create`. Η χρήση της είναι ισοδύναμη με διαδοχική κλήση των `ksocket_release` και `ksocket_create`, ωστόσο απαιτεί λιγότερο χρόνο. Είναι ευθύνη του καλούντος να εξασφαλίσει ότι κατά την κλήση της `ksocket_release` η δομή ksocket δεν χρησιμοποιείται από κανένα τμήμα του πυρήνα.

3.4.12 Σύνοψη

Οι υπηρεσίες που προσφέρονται από το στρώμα επικοινωνίας στους χρήστες του, δηλαδή στα υποσυστήματα του πυρήνα που επιθυμούν να ανταλλάξουν δεδομένα, συνοψίζονται στον ακόλουθο πίνακα:

υπηρεσία	λειτουργία υπηρεσίας
`ksocket_create`	δημιουργεί και αρχικοποιεί μια νέα δομή ksocket
`ksocket_open`	ανοίγει ένα νέο κανάλι επικοινωνίας με το χώρο χρήστη
`ksocket_bind`	δεσμεύει μια τοπική θύρα επικοινωνίας των KSockets
`ksocket_connect`	εγκαθιστά σύνδεση προς απομακρυσμένο πυρήνα
`ksocket_accept`	αναμένει σύνδεση από απομακρυσμένο πυρήνα
`ksocket_read`	λαμβάνει δεδομένα από τον απομακρυσμένο πυρήνα
`ksocket_write`	αποστέλλει δεδομένα στον απομακρυσμένο πυρήνα
`ksocket_close`	τερματίζει τη σύνδεση και τη χρήση της δομής ksocket
`ksocket_reset`	θέτει τα πεδία της δομής ksocket στις αρχικές τους τιμές
`ksocket_release`	απελευθερώνει το χώρο μνήμης μιας δομής ksocket

Πίνακας 3.1 – Προσφερόμενες υπηρεσίες του στρώματος επικοινωνίας

Έχοντας ολοκληρώσει την παρουσίαση των υπηρεσιών του στρώματος επικοινωνίας που προσφέρονται στα υπόλοιπα τμήματα του πυρήνα του Linux μπορούμε να προχωρήσουμε στην παρουσίαση της υλοποίησης των παραπάνω συναρτήσεων στο χώρο πυρήνα του Linux, καθώς και του τρόπου επικοινωνίας με τη διεργασία χώρου χρήστη. Τα θέματα αυτά αποτελούν το αντικείμενο του επόμενου κεφαλαίου.

Κεφάλαιο 4

Σχεδιασμός και υλοποίηση από την πλευρά του πυρήνα

Από την πλευρά του χώρου πυρήνα το στρώμα επικοινωνίας ορίζει τη βασική διαπροσωπεία για τη χρήση του από τα υπόλοιπα υποσυστήματα του πυρήνα, όπως αυτή περιγράφηκε στο προηγούμενο κεφάλαιο, υποβάλλει αιτήσεις στο τμήμα χώρου χρήστη για την εγκατάσταση και τον τερματισμό νέων συνδέσεων, ανταλλάσσει τα δεδομένα της επικοινωνίας με τη διεργασία χώρου χρήστη και επιβάλλει τους κατάλληλους μηχανισμούς συγχρονισμού κατά τη χρήση του.

4.1 Γενικά για το μοντέλο επικοινωνίας με το χώρο χρήστη

Η επικοινωνία με τη διεργασία χώρου χρήστη γίνεται μέσω μιας εικονικής συσκευής χαρακτήρων, ο οδηγός της οποίας υλοποιείται από το τμήμα πυρήνα των KSockets. Από την πλευρά του χώρου χρήστη χρησιμοποιούνται οι συνηθισμένες κλήσεις συστήματος για χειρισμό αρχείων (`open`, `close`, `read`, `write`) επάνω σε συγκεκριμένο ειδικό αρχείο συσκευής[10], το οποίο αποτελεί την αναπαράσταση της εικονικής συσκευής χαρακτήρων στο σύστημα αρχείων του Unix. Από την πλευρά του χώρου πυρήνα, ο οδηγός καταχωρείται ως διαθέσιμος για την ικανοποίηση των αιτήσεων προς την εικονική

[10] Το όνομα του αρχείου επιλέγεται αυθαίρετα κατά τη μεταγλώττιση, μια επιλογή είναι /dev/ksocket0

συσκευή και παρέχει στο VFS τη δομή `file_operations` η οποία περιέχει δείκτες προς τις μεθόδους του, όπως περιγράφεται στο κεφάλαιο 2.

Έχοντας ανοίξει ένα κανάλι επικοινωνίας με την εικονική συσκευή χαρακτήρων, χρησιμοποιώντας την κλήση open για το ειδικό αρχείο συσκευής, η διεργασία χώρου χρήστη μπορεί να δέχεται αιτήσεις από το χώρο πυρήνα, χρησιμοποιώντας την κλήση συστήματος `read` και να τις ικανοποιεί εκτελώντας τις απαραίτητες ενέργειες στο δίκτυο διασύνδεσης. Οι αιτήσεις που υποβάλλονται από τις συναρτήσεις που προσφέρει το στρώμα επικοινωνίας αποθηκεύονται προσωρινά σε μία *ουρά αιτήσεων*, έως ότου τις προσπελάσει η διεργασία χώρου χρήστη χρησιμοποιώντας την κλήση συστήματος read. Η εκτέλεση της `read` από τη διεργασία χώρου χρήστη συνεπάγεται την κλήση της ανάλογης μεθόδου του οδηγού εικονικής συσκευής, στη συγκεκριμένη περίπτωση της `ksocket_fops_read`, η οποία και αναλαμβάνει το χειρισμό της ουράς αιτήσεων και την αντιγραφή της αίτησης στο χώρο χρήστη, εκ μέρους της διεργασίας. Η μορφή των αιτήσεων, τα τμήματα που αποτελούν την κάθε μία και η σημασία τους, αναλύονται διεξοδικά στη συνέχεια.

Η βασική μονάδα διαχειριστικής πληροφορίας από την πλευρά του πυρήνα είναι η δομή ksocket, με την οποία αναπαρίσταται κάθε σύνδεση με απομακρυσμένο πυρήνα. Εφόσον όμως οι δομές ksocket αποθηκεύονται στο χώρο πυρήνα, η διεργασία χώρου χρήστη δεν μπορεί να τις προσπελάσει άμεσα. Έτσι, χρησιμοποιεί υπηρεσίες του πυρήνα, συγκεκριμένα την κλήση συστήματος `ioctl` του οδηγού εικονικής συσκευής, ώστε να ενημερώσει τον πυρήνα για αλλαγές στην κατάσταση των συνδέσεων και να ζητήσει την ανανέωση των πληροφοριών που τηρούνται στις δομές ksocket. Αυτό συνεπάγεται ότι η εξυπηρέτηση ενός αιτήματος επικοινωνίας του τμήματος χώρου πυρήνα από το τμήμα χώρου χρήστη γίνεται σε δύο ξεχωριστά βήματα: Αρχικά το αίτημα του τμήματος πυρήνα εισάγεται σε κατάλληλη ουρά αιτήσεων, απ' όπου το τμήμα χώρου χρήστη το ανακτά με χρήση της `read` και στη συνέχεια, αφού έχουν τελειώσει όλες οι διαδικασίες διαχείρισης του δικτύου διασύνδεσης ώστε να ικανοποιηθεί το αίτημα, χρησιμοποιείται η κλήση συστήματος `ioctl` για την ανανέωση της κατάστασης των ksockets και την ενημέρωση του χώρου πυρήνα για την ολοκλήρωση του αιτήματός του.

Το αρχικό κανάλι επικοινωνίας χώρου πυρήνα - χώρου χρήστη που δημιουργείται από την κλήση της `open` αναλαμβάνει τη μεταφορά αιτήσεων που αφορούν τον έλεγχο των συνδέσεων, όχι όμως και τη μεταφορά των πραγματικών δεδομένων που ανταλλάσσονται

ανάμεσα σε δύο πυρήνες που επικοινωνούν. Αντίθετα, για κάθε ksocket που συνδέεται με το χώρο χρήστη (συνάρτηση `ksocket_open`) δημιουργείται ένα νέο κανάλι επικοινωνίας, από το οποίο θα ανταλλαγούν δεδομένα όταν εγκατασταθεί μια σύνδεση με κάποιο απομακρυσμένο πυρήνα. Το νέο κανάλι επικοινωνίας χώρου πυρήνα - χώρου χρήστη είναι ουσιαστικά ένα επιπλέον ανοιχτό αρχείο για τη διεργασία χώρου χρήστη, εντελώς ανεξάρτητο από τα υπόλοιπα, που κατασκευάζεται με εκ νέου χρήση της κλήσης συστήματος open. Η προσπέλαση του αρχείου γίνεται μέσω ενός νέου περιγραφητή αρχείων από την πλευρά της διεργασίας, ενώ από την πλευρά του πυρήνα κατασκευάζεται μια ακόμη δομή τύπου `struct file`, η οποία τοποθετείται στον πίνακα των ανοιχτών αρχείων της διεργασίας. Η παραπάνω διαδικασία περιγράφεται αναλυτικά στη συνέχεια, όπου παρουσιάζεται το σύνολο των αιτήσεων του πυρήνα προς το χώρο χρήστη και ο τρόπος που αυτά εξυπηρετούνται.

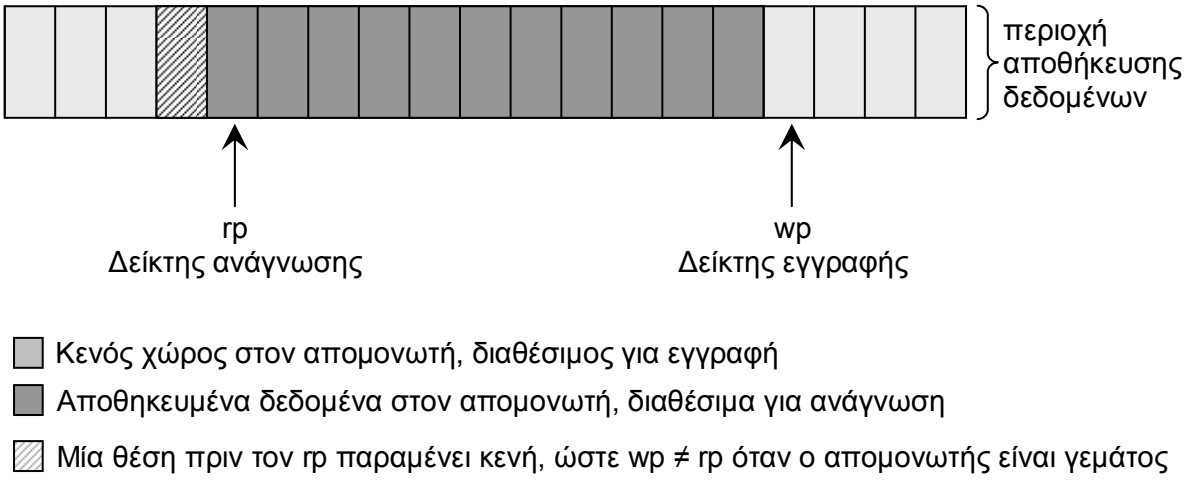
4.2 Υλοποίηση απομονωτών Εισόδου / Εξόδου

Κατά την επικοινωνία μέσω των KSockets χρησιμοποιούνται *απομονωτές* Εισόδου / Εξόδου (I/O buffers). Σε απομονωτή εξόδου αποθηκεύονται προσωρινά τα δεδομένα προς αποστολή σε απομακρυσμένο πυρήνα, μέχρι να ανακτηθούν και να αποσταλούν από τη διεργασία χώρου χρήστη, ενώ σε απομονωτή εισόδου αποθηκεύονται τα δεδομένα που λαμβάνονται από το δίκτυο διασύνδεσης και παραδίδονται μέσω του οδηγού εικονικής συσκευής στον πυρήνα, μέχρι να ανακτηθούν από τους χρήστες του στρώματος επικοινωνίας. Κάθε δομή ksocket περιέχει δύο τέτοιους απομονωτές Ε / Ε.

4.2.1 Δομή των απομονωτών Ε / Ε

Οι απομονωτές Ε / Ε είναι κυκλικοί και υλοποιούνται από τον τύπο `circ_buf` στο τμήμα πυρήνα των KSockets. Κάθε απομονωτής αποτελείται από μια περιοχή αποθήκευσης των δεδομένων που βρίσκονται αποθηκευμένα σε αυτόν. Το μέγεθος της περιοχής (έστω `BUF_SIZE`) είναι παράμετρος που καθορίζεται κατά τη μεταγλώττιση του στρώματος επικοινωνίας. Επιπλέον χρησιμοποιούνται δύο δείκτες για τη διαχείριση του απομονωτή: Ο πρώτος δείκτης (write pointer) χρησιμοποιείται από τους παραγωγούς των δεδομένων που εισάγονται στον απομονωτή και δείχνει στο σημείο όπου υπάρχει διαθέσιμος χώρος για την εισαγωγή νέων δεδομένων, ενώ ο δεύτερος χρησιμοποιείται από τους καταναλωτές των

δεδομένων και δείχνει στο σημείο απ' όπου μπορούν να αναγνωσθούν δεδομένα. Σχηματικά, η δομή ενός κυκλικού απομονωτή είναι:



Σχήμα 4.1 – Δομή κυκλικού απομονωτή E / E

Η τιμή του δείκτη ανάγνωσης μεταβάλλεται μόνο από τους καταναλωτές των δεδομένων, ενώ η τιμή του δείκτη εγγραφής μόνο από τους παραγωγούς. Τα έγκυρα δεδομένα που περιέχονται στον απομονωτή είναι αποθηκευμένα από το σημείο που δείχνει ο `rp`, έως και μία θέση πριν το σημείο που δείχνει ο `wp`. Καθώς αφαιρούνται και προστίθενται δεδομένα στον απομονωτή, οι δείκτες αυξάνονται και όταν φτάσουν στο όριο της περιοχής αποθήκευσης επαναφέρονται στην αρχή της. Έτσι, ο αριθμός των bytes που περιέχει ο απομονωτής δίνονται από τη σχέση:

```
nstored = (wp - rp + BUF_SIZE) mod BUF_SIZE
```

όπου λαμβάνεται υπόψη το ενδεχόμενο rp > wp.

Ανάλογα, ο διαθέσιμος χώρος στον απομονωτή είναι:

```
nfree = (rp - wp + BUF_SIZE - 1) mod BUF_SIZE
```

Αρχικά είναι `rp` = `wp`, δείχνουν στην αρχή της περιοχής αποθήκευσης, οπότε ο απομονωτής είναι άδειος. Αντίθετα, όταν ο απομονωτής είναι γεμάτος ο `wp` βρίσκεται στην ακριβώς προηγούμενη θέση από τον `rp`. Έτσι, ο μέγιστος αριθμός bytes που μπορούν να αποθηκευτούν είναι `BUF_SIZE - 1`.

4.2.2 Συγχρονισμός κατά την πρόσβαση σε απομονωτές

Η ταυτόχρονη πρόσβαση στους απομονωτές επιβάλλει την ύπαρξη κατάλληλων μηχανισμού συγχρονισμού. Τα περιεχόμενα κάθε απομονωτή προστατεύονται από αντίστοιχο σηματοφορέα. Εφόσον γίνεται ανταλλαγή δεδομένων ανάμεσα στους απομονωτές Ε / Ε και το χώρο χρήστη, δεν επιτρέπεται η χρήση spinlocks, αφού οι συναρτήσεις πρόσβασης στο χώρο χρήστη μπορούν να προκαλέσουν διακοπή της εκτέλεσης του κώδικα πυρήνα (βλ. §1.4.2). Έτσι, χρησιμοποιούμε σηματοφορείς, για τους οποίους επιτρέπεται ενώ κρατείται το κλείδωμα να γίνει πρόσβαση στο χώρο χρήστη και να «κοιμηθεί» ο κώδικας πυρήνα.

Επιπλέον, εκτός από την προστασία κατά την ταυτόχρονη πρόσβαση είναι αναγκαία η ύπαρξη μηχανισμών συγχρονισμού και για να αντιμετωπιστούν δύο ακόμη περιπτώσεις:

- Ζητείται ανάγνωση δεδομένων όταν ο απομονωτής είναι άδειος
- Ζητείται εγγραφή δεδομένων όταν ο απομονωτής είναι πλήρης

Στις περιπτώσεις αυτές, οι συναρτήσεις διαχείρισης των απομονωτών δίνουν τη δυνατότητα είτε να επιστρέψουν αμέσως με τον κωδικό σφάλματος `EAGAIN` (Try again - Operation would block), είτε να μπλοκάρουν έως ότου γίνουν διαθέσιμα δεδομένα ή ελευθερωθεί χώρος για αποθήκευση νέων δεδομένων, ανάλογα με την περίπτωση. Για το σκοπό αυτό αντιστοιχίζονται σε κάθε απομονωτή δύο ουρές αναμονής (τύπου `wait_queue`, υλοποιούνται από τον κώδικα χρονοδρομολόγησης του πυρήνα), μία για τους παραγωγούς και μία για τους καταναλωτές των δεδομένων. Κάθε φορά που αποθηκεύονται νέα δεδομένα στον απομονωτή, ενεργοποιούνται οι διεργασίες από την ουρά αναμονής αυτών που περιμένουν για ανάγνωση. Αντίστοιχα, κάθε φορά που γίνεται ανάγνωση, ενεργοποιούνται οι διεργασίες από την ουρά αναμονής αυτών που περιμένουν για εγγραφή.

4.2.3 Ρουτίνες διαχείρισης των απομονωτών

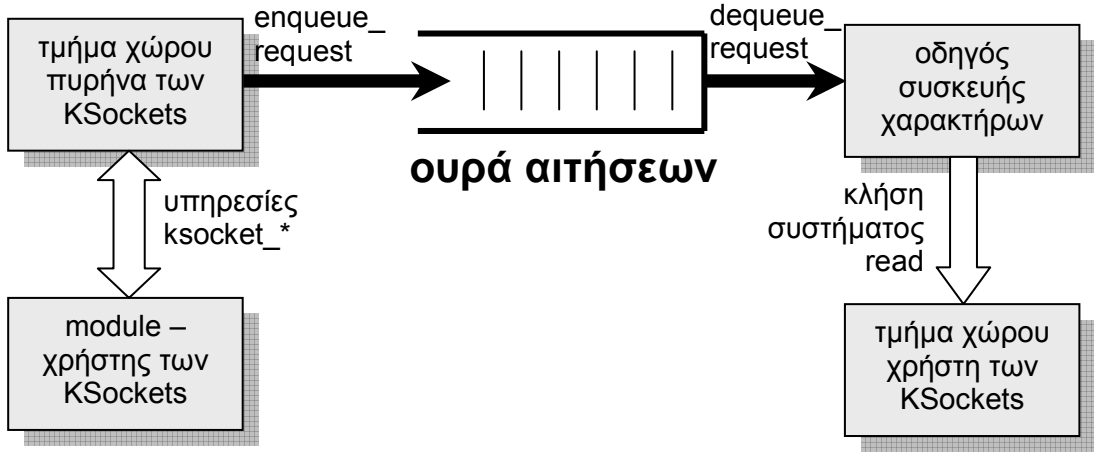
Το τμήμα χώρου πυρήνα του στρώματος επικοινωνίας προσφέρει κατάλληλες ρουτίνες για τη διαχείριση των απομονωτών Ε / Ε:

Η συνάρτηση `buf_init` δεσμεύει χώρο στην κεντρική μνήμη για μια νέα δομή κυκλικού απομονωτή και αρχικοποιεί τις τιμές των πεδίων της, ενώ η συνάρτηση `buf_destroy` χρησιμοποιείται από την `ksocket_release` για την αποδέσμευση του χώρου μνήμης ενός

απομονωτή και την καταχώρησή του ως διαθέσιμου. Οι συναρτήσεις `buf_read` και `buf_write` αναλαμβάνουν την αποθήκευση και ανάκτηση δεδομένων από τον απομονωτή. Ακολουθούν τη σημασιολογία των κλήσεων συστήματος `read` / `write` κατά τη χρήση τους και ισχύουν όσα είχαν ειπωθεί προηγούμενα για τις `ksocket_read` και `ksocket_write` (§3.4.7, §3.4.8). Τέλος, η `buf_free` επιστρέφει το μέγεθος του χώρου που είναι διαθέσιμος σε έναν απομονωτή για αποθήκευση δεδομένων.

4.3 Η ουρά αιτήσεων προς το χώρο χρήστη

Οι αιτήσεις ελέγχου των συνδέσεων που υποβάλλονται από το τμήμα πυρήνα προς τη διεργασία χώρου χρήστη, η οποία και διαχειρίζεται το δίκτυο διασύνδεσης, αποθηκεύονται προσωρινά στην ουρά αιτήσεων, μία δομή FIFO (First In, First Out). Η διεργασία χώρου χρήστη λαμβάνει αιτήσεις από την ουρά αιτήσεων, εκτελώντας την κλήση συστήματος `read` του οδηγού εικονικής συσκευής χαρακτήρων και τις εξυπηρετεί.



Σχήμα 4.2 – Ο ρόλος της ουράς αιτήσεων στην ικανοποίηση αιτημάτων του πυρήνα

Η δομή της ουράς αιτήσεων δεν διαφέρει σημαντικά από αυτή των απομονωτών Ε / Ε. Ωστόσο, βασική μονάδα αποθήκευσης σε αυτή δεν είναι το byte, όπως στους απομονωτές, αλλά η αίτηση προς το χώρο χρήστη, που αναπαρίσταται από τον τύπο `ksocket_req_t`. Η ουρά αιτήσεων ρησιμοποιεί ένα σηματοφορέα για εξασφάλιση της ακεραιότητας των δεδομένων κατά την ταυτόχρονη πρόσβαση και διαθέτει ουρές αναμονής για τους παραγωγούς (τις συναρτήσεις που υλοποιούν υπηρεσίες του στρώματος επικοινωνίας) και τον καταναλωτή (τη διεργασία χώρου χρήστη). Το μέγεθος της ουράς αιτήσεων, δηλαδή το πλήθος των αιτήσεων που είναι δυνατό να περιέχει χωρίς αυτές να έχουν παραληφθεί από τη διεργασία χώρου χρήστη καθορίζεται κατά τη μεταγλώττιση.

4.3.1 Συναρτήσεις διαχείρισης της ουράς αιτήσεων

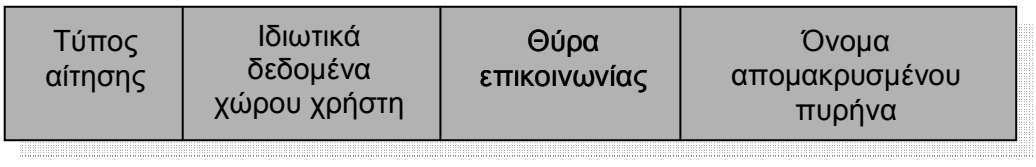
Η συνάρτηση `init_request_queue` δεσμεύει χώρο στην κεντρική μνήμη για τη δημιουργία της ουράς αιτήσεων και τον αρχικοποιεί. Χρησιμοποιείται κατά την εκκίνηση του πυρήνα του Linux, στην περίπτωση που το στρώμα KSockets είναι μεταγλωττισμένο ως μόνιμο τμήμα του πυρήνα. Στην περίπτωση που τα KSockets φορτώνονται ως module για να συνδεθούν με ένα πυρήνα που ήδη εκτελείται, η συνάρτηση init_request_queue καλείται από τη ρουτίνα αρχικοποίησής του.

Η συνάρτηση `destroy_request_queue` αποδεσμεύει το χώρο μνήμης που κατείχε η ουρά αιτήσεων και χρησιμοποιείται όταν αφαιρείται το module των KSockets από τον πυρήνα του Linux.

Η συνάρτηση `enqueue_request` εισάγει μια νέα αίτηση στην ουρά αιτήσεων, ενώ η `dequeue_request_to_user` εξάγει την πρώτη αίτηση από την ουρά αιτήσεων και την προωθεί στο χώρο χρήστη. Η πρώτη από τις δύο συναρτήσεις χρησιμεύει στις ρουτίνες που υλοποιούν υπηρεσίες των KSockets (`ksocket_open`, `ksocket_bind` κλπ), ενώ η δεύτερη χρησιμοποιείται από τη μέθοδο `ksocket_fops_read` του οδηγού εικονικής συσκευής χαρακτήρων.

4.3.2 Περιγραφή της δομής των αιτήσεων

Κάθε αίτηση που προωθείται από το χώρο πυρήνα στο χώρο χρήστη μέσω της ουράς αιτήσεων έχει συγκεκριμένο μέγεθος και δομή. Όπως φαίνεται στο σχήμα, μια αίτηση αποτελείται από τέσσερα κύρια πεδία: Το είδος της αίτησης, ένα δείκτη προς ιδιωτικά δεδομένα της διεργασίας χώρου χρήστη (`user_data`), τον αριθμό της θύρας επικοινωνίας και το όνομα του απομακρυσμένου πυρήνα που αφορά η αίτηση.



Σχήμα 4.3 – Δομή αίτησης προς το τμήμα χώρου χρήστη

Ο δείκτης προς ιδιωτικά δεδομένα της διεργασίας χρήστη υποδηλώνει τη δομή ksocket που αφορά το συγκεκριμένο αίτημα του πυρήνα. Σε κάθε δομή ksocket του χώρου πυρήνα υπάρχει ένα πεδίο `user_data`, του οποίου η χρήση δεν αφορά το τμήμα πυρήνα, αλλά

καθορίζεται αποκλειστικά από το τμήμα χρήστη. Όταν ανοίγει ένα νέο κανάλι επικοινωνίας προς το χώρο χρήστη που σχετίζεται με μια δομή ksocket, το πεδίο `user_data` της δομής αρχικοποιείται ώστε να δείχνει στις αντίστοιχες πληροφορίες κατάστασης της διεργασίας χώρου χρήστη. Είναι δηλαδή ένα μέσο διάκρισης των ksockets από την πλευρά της διεργασίας, η οποία χρησιμοποιεί το πεδίο `user_data` της αίτησης ώστε να εντοπίσει τις διαχειριστικές πληροφορίες του δικτύου διασύνδεσης που αφορούν το συγκεκριμένο ksocket.

4.3.3 Τύποι αιτήσεων προς το χώρο χρήστη

Στη συνέχεια παρουσιάζονται οι πέντε διαφορετικοί τύποι αιτήσεων που προβλέπονται προς το χώρο χρήστη και αναλύεται η σημασία των πεδίων της αίτησης για καθένα από αυτούς:

- **τύπος KSOCKET_REQ_OPEN:** Ζητά από τη διεργασία χώρου χρήστη τη δημιουργία ενός νέου καναλιού επικοινωνίας και τη συσχέτισή του με κάποιο ksocket. Είναι ο μόνος τύπος αίτησης για τον οποίο δεν χρησιμοποιείται το πεδίο `user_data`, αφού η διεργασία χώρου χρήστη δεν έχει ακόμη ενημερωθεί για την ύπαρξη του νέου ksocket και δεν έχει δημιουργήσει κατάλληλες δομές για την αποθήκευση πληροφοριών κατάστασης. Αιτήσεις `KSOCKET_REQ_OPEN` δρομολογούνται από τη συνάρτηση `ksocket_open`.

- **τύπος KSOCKET_REQ_BIND:** Ζητά τη δέσμευση μιας θύρας επικοινωνίας των KSockets για χρήση με συγκεκριμένη δομή ksocket. Το πεδίο `user_data` της αίτησης παίρνει τιμή από το πεδίο `user_data` της δομής, ενώ στο πεδίο `port` αποθηκεύεται ο αριθμός της επιθυμητής θύρας επικοινωνίας.

- **τύπος KSOCKET_REQ_CONNECT:** Ζητά την εγκατάσταση μιας σύνδεσης προς συγκεκριμένο απομακρυσμένο πυρήνα και θύρα επικοινωνίας. Τα στοιχεία αυτά αποθηκεύονται στα πεδία `remote_name` και `port` αντίστοιχα. Η δομή ksocket μέσω της οποίας θα είναι προσβάσιμη η σύνδεση για ανταλλαγή δεδομένων καθορίζεται μέσω του πεδίου `user_data`.

- **τύπος KSOCKET_REQ_ACCEPT:** Ζητά την αναμονή για σύνδεση από κάποιο απομακρυσμένο πυρήνα, με χρήση του ksocket που καθορίζεται από το πεδίο `user_data`.

- **τύπος KSOCKET_REQ_CLOSE:** Ζητά το κλείσιμο ενός καναλιού επικοινωνίας προς το χώρο χρήστη.

Οι συναρτήσεις που υλοποιούν υπηρεσίες του στρώματος επικοινωνίας στο χώρο πυρήνα αναλαμβάνουν το σωστό έλεγχο των παραμέτρων και εξασφαλίζουν ότι οι αιτήσεις που προωθούνται στο χώρο χρήστη περιέχουν έγκυρα δεδομένα και ανταποκρίνονται στην τρέχουσα κατάσταση του ksocket. Έτσι, η διεργασία χώρου χρήστη μπορεί να χρησιμοποιήσει άμεσα τις πληροφορίες που λαμβάνει, χωρίς να απαιτείται έλεγχος ορθότητας των παραμέτρων.

4.4 Η δομή ksocket

Η δομή ksocket περιγράφει ένα σημείο αλληλεπίδρασης των χρηστών του στρώματος επικοινωνίας KSockets με αυτό. Αναπαριστά το τοπικό άκρο μιας σύνδεσης ανάμεσα σε πυρήνες Linux και χρησιμοποιείται για την αποθήκευση των διαχειριστικών πληροφοριών της σύνδεσης στο χώρο πυρήνα. Ο ορισμός της δομής που παρουσιάζεται στη συνέχεια περιέχεται στο αρχείο `ksocket.h`, το αρχείο επικεφαλίδας που περιγράφει τις βασικότερες δομές δεδομένων των KSockets:

πεδία	σημασία πεδίων
`int port, remote port`	τοπική και απομακρυσμένη θύρα επικοινωνίας KSockets
`char remote_name[]`	όνομα απομακρυσμένου πυρήνα
`ksocket_state_t state`	πεδίο κατάστασης για τη δομή ksocket
`volatile int *error_p`	δείκτης για την επιστροφή τιμών σφάλματος
`wait_queue_head_t evqueue`	ουρά αναμονής για αλλαγή κατάστασης του ksocket
`circ_buf rdbuf, wrbuf`	κυκλικοί απομονωτές Ε/Ε
`spinlock_t slock`	κλείδωμα spinlock για προστασία της δομής
`void *user_data`	δείκτης προς ιδιωτικά δεδομένα χώρου χρήστη
`struct list_head list`	πεδίο για την κατασκευή συνδ. λίστας από ksockets

Πίνακας 4.1 – Πεδία της δομής ksocket

Τα πεδία που αναφέρονται στον παραπάνω πίνακα αναλύονται διεξοδικά στη συνέχεια.

Όπως βλέπουμε, κάθε ksocket σχετίζεται με δύο κυκλικούς απομονωτές, τον απομονωτή εισόδου `rdbuf` και τον απομονωτή εξόδου `wrbuf`. Ο πρώτος χρησιμοποιείται για την αποθήκευση των δεδομένων που γίνονται διαθέσιμα από το χώρο χρήστη και είναι προσπελάσιμα μέσω της υπηρεσίας `ksocket_read` του στρώματος επικοινωνίας, ενώ ο

δεύτερος χρησιμοποιείται για την αποθήκευση των δεδομένων που είναι έτοιμα για προώθηση προς το χώρο χρήστη και προέρχονται από κλήση της υπηρεσίας `ksocket_write`.

Επιπλέον, σε κάθε ksocket αποθηκεύεται δείκτης προς τα ιδιωτικά δεδομένα του χώρου χρήστη που σχετίζονται με αυτό (`user_data`). Επίσης υπάρχουν πληροφορίες που αφορούν την τοπική θύρα επικοινωνίας που έχει δεσμευτεί για χρήση με το συγκεκριμένο ksocket (`port`), το όνομα του απομακρυσμένου πυρήνα, στην περίπτωση που υπάρχει εγκατεστημένη σύνδεση, όπως και ο αριθμός της απομακρυσμένης θύρας επικοινωνίας (`remote_name` και `remote_port` αντίστοιχα).

Από τα πιο σημαντικά πεδία της δομής ksocket είναι το πεδίο κατάστασης `state`, το οποίο καθορίζει το είδος των λειτουργιών που είναι δυνατό να πραγματοποιηθούν κάθε φορά. Προβλέπονται δύο διαφορετικά είδη καταστάσεων, οι *μόνιμες καταστάσεις*, όταν δεν εκκρεμεί κάποιο αίτημα προς το χώρο χρήστη που αφορά το ksocket και οι *μεταβατικές καταστάσεις*, στις οποίες βρίσκεται ένα ksocket όταν έχει ήδη εισαχθεί στην ουρά αιτήσεων κάποιο αίτημα που το αφορά. Η αλλαγή σε μεταβατική κατάσταση γίνεται από τις ρουτίνες που υλοποιούν τις υπηρεσίες του στρώματος επικοινωνίας, ενώ η μετάβαση σε μόνιμη κατάσταση γίνεται από τη μέθοδο ksocket_fops_ioctl, που καλείται όταν η διεργασία χώρου χρήστη ανανεώνει την κατάσταση του ksocket ώστε να αντανακλά τις ενέργειες της και τα αποτελέσματά τους στο δίκτυο διασύνδεσης.

Οι δυνατές μόνιμες καταστάσεις και η σημασία τους είναι:

- **KSOCKET_OPEN:** Υπάρχει ανοιχτό κανάλι επικοινωνίας προς το χώρο χρήστη, συσχετισμένο με τη δομή αυτή.

- **KSOCKET_BOUND:** Υπάρχει ανοιχτό κανάλι επικοινωνίας, έχει δεσμευτεί θύρα επικοινωνίας και η δομή μπορεί να χρησιμοποιηθεί για εγκατάσταση εξερχόμενης σύνδεσης ή αποδοχή εισερχόμενης σύνδεσης.

- **KSOCKET_CONNECTED**: Μια σύνδεση έχει εγκατασταθεί και είναι δυνατή η αμφίδρομη ανταλλαγή δεδομένων με χρήση των `ksocket_read`, `ksocket_write`.

- **KSOCKET_CLOSED_LINKED:** Ο τοπικός πυρήνας έχει ζητήσει τερματισμό της σύνδεσης με χρήση της `ksocket_close` και δεν επιτρέπεται να προσπελάσει τη

δομή με χρήση των `ksocket_read` και `ksocket_write`. Ωστόσο ο σύνδεσμος με το χώρο χρήστη παραμένει ανοιχτός, έως ότου ολοκληρωθεί η αποστολή δεδομένων που βρίσκονται προσωρινά στον απομονωτή εξόδου.

- **KSOCKET_OPEN_UNLINKED:** Το κανάλι επικοινωνίας με το χώρο χρήστη έχει κλείσει, διότι ο απομακρυσμένος πυρήνας τερμάτισε τη σύνδεση και έτσι δεν έχει πλέον νόημα η μεταφορά δεδομένων από και προς το χώρο χρήστη. Ωστόσο, τα δεδομένα που βρίσκονται στον απομονωτή εισόδου είναι διαθέσιμα στα υπόλοιπα τμήματα του πυρήνα με χρήση της `ksocket_read`.

- **KSOCKET_CLOSED_UNLINKED:** Το κανάλι επικοινωνίας με το χώρο χρήστη έχει κλείσει και οι απομονωτές Ε / Ε δεν περιέχουν δεδομένα. Το ksocket μπορεί να χρησιμοποιηθεί για τη δημιουργία νέων συνδέσεων αρκεί να αρχικοποιηθούν όλα τα πεδία του με χρήση της `ksocket_reset`. Η `KSOCKET_CLOSED_UNLINKED` είναι επίσης η κατάσταση ενός ksocket μόλις αυτό έχει δημιουργηθεί με χρήση της `ksocket_create`.

Επίσης υπάρχουν τέσσερις μεταβατικές καταστάσεις:

- **KSOCKET_OPEN_INPROGRESS:** Έχει προωθηθεί αίτημα `KSOCKET_REQ_OPEN` για τη δημιουργία καναλιού επικοινωνίας με το χώρο χρήστη

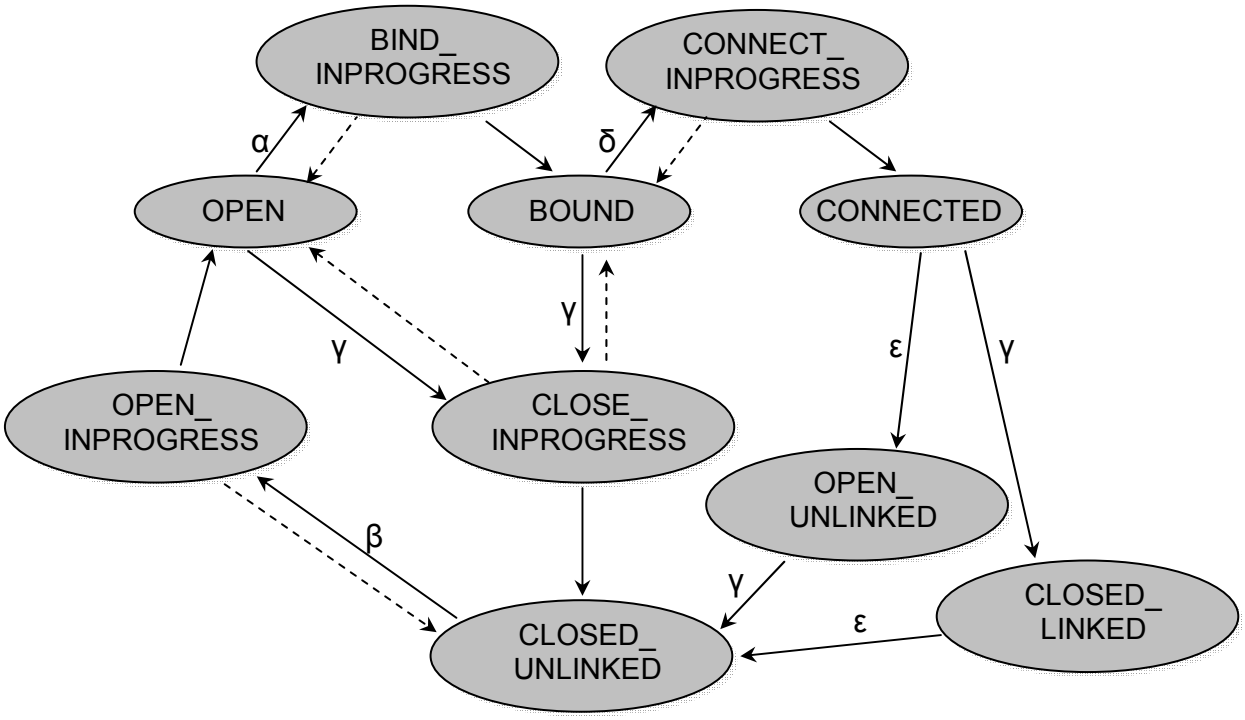
- **KSOCKET_BIND_INPROGRESS:** Έχει προωθηθεί αίτημα `KSOCKET_REQ_BIND` για τη δέσμευση τοπικής θύρας επικοινωνίας

- **KSOCKET_CONNECT_INPROGRESS:** Έχει προωθηθεί αίτημα `KSOCKET_REQ_ACCEPT` ή `KSOCKET_REQ_CONNECT` και αναμένεται η εγκατάσταση σύνδεσης με απομακρυσμένο πυρήνα

- **KSOCKET_CLOSE_INPROGRESS:** Έχει προωθηθεί αίτημα `KSOCKET_REQ_CLOSE` και αναμένεται η χρήση της κλήσης συστήματος close από τη διεργασία

Το διάγραμμα καταστάσεων για μια δομή ksocket παρουσιάζεται στο παρακάτω σχήμα. Κάθε βέλος αναπαριστά μια δυνατή μετάβαση. Τα βέλη με διακεκομμένη γραμμή υποδεικνύουν μεταβάσεις που οφείλονται σε αποτυχία ικανοποίησης του αιτήματος του

πυρήνα από το χώρο χρήστη. Για συντομία έχει παραληφθεί η επιγραφή “KSOCKET_” από το όνομα κάθε κατάστασης.



Σχήμα 4.4 – Διάγραμμα καταστάσεων δομής ksocket

Η περιγραφή των μεταβάσεων είναι ως εξής: **(α)** κλήση `ksocket_bind`, **(β)** κλήση `ksocket_open`, **(γ)** κλήση `ksocket_close`, **(δ)** κλήση `ksocket_connect` ή `ksocket_accept`, (ε) κλείσιμο ανοιχτού αρχείου από διεργασία χρήση με χρήση της κλήσης συστήματος `close`. Οι μεταβάσεις που δεν έχουν ετικέτα είναι αυτές που προκαλούνται όταν η διεργασία χώρου χρήστη εκτελέσει εντολή ioctl για να θέσει την κατάσταση ενός ksocket.

Ο έλεγχος ταυτόχρονης πρόσβασης σε δομή ksocket πραγματοποιείται με ένα spinlock. Ο έλεγχος της κατάστασης ενός ksocket και η μεταβολή της γίνονται μόνο αφού προηγουμένως έχει κλειδωθεί το αντίστοιχο spinlock. Η λύση του spinlock προτιμάται σε σύγκριση με τους σηματοφορείς, αφού ο χρόνος κράτησης του κλειδώματος είναι πολύ μικρός και είναι προτιμότερος ένας σύντομος βρόχος «απασχόλησης-αναμονής» από την προσθήκη σε ουρά αναμονής και την κλήση του χρονοδρομολογητή.

4.5 Κανάλια επικοινωνίας χώρου πυρήνα – χώρου χρήστη

Έχουμε ήδη αναφέρει ότι σε κάθε ksocket που βρίσκεται στην κατάσταση ksocket_open αντιστοιχεί ένα *κανάλι επικοινωνίας* με τη διεργασία χώρου χρήστη. Με τον όρο «κανάλι επικοινωνίας» εννοούμε απλά ένα ανοιχτό αρχείο για τη διεργασία χώρου χρήστη, που έχει προκύψει από την εκτέλεση της κλήσης συστήματος open.

Έχοντας καλέσει μία φορά, αρχικά, την open για το ειδικό αρχείο της εικονικής συσκευής χαρακτήρων ώστε να αποκτήσει πρόσβαση στην ουρά αιτήσεων, η διεργασία καλεί την open εκ νέου, κάθε φορά που δέχεται αίτημα τύπου `KSOCKET_REQ_OPEN`. Αυτό έχει σαν αποτέλεσμα τη δημιουργία από το VFS μιας νέας δομής τύπου `struct file`, η οποία αποτελεί την αναπαράσταση κάθε ανοιχτού αρχείου στο χώρο πυρήνα. Η δομή αυτή, παρόλο που προέκυψε από κλήση της open για τον ίδιο κόμβο του συστήματος αρχείων (το ειδικό αρχείο συσκευής), είναι εντελώς ανεξάρτητη από τις υπόλοιπες. Από την πλευρά του χώρου χρήστη, η διεργασία μπορεί να προσπελάσει το ανοιχτό αρχείο μέσω ενός νέου περιγραφητή αρχείου που της επιστρέφεται από το σύστημα.

Η διάκριση των δομών τύπου `struct file` μεταξύ τους και η εύρεση της δομής ksocket με την οποία σχετίζονται γίνεται μέσω του πεδίου `private_data` που περιέχουν (§2.2.1). Ο οδηγός της εικονικής συσκευής χαρακτήρων χρησιμοποιεί το πεδίο αυτό ως δείκτη στην κατάλληλη δομή ksocket. Επιπλέον, αν η δομή file αφορά το αρχικό κανάλι επικοινωνίας, για πρόσβαση στην ουρά αιτήσεων, το πεδίο `private_data` έχει την τιμή `NULL`. Έτσι η συμπεριφορά των μεθόδων του οδηγού συσκευής μεταβάλλεται ανάλογα με το αν το ανοιχτό αρχείο αφορά την ουρά αιτήσεων ή κάποιο ksocket: Στην πρώτη περίπτωση γίνεται πρόσβαση στην ουρά αιτήσεων, στη δεύτερη γίνεται μεταφορά δεδομένων από και προς τους απομονωτές Ε / Ε του αντίστοιχου ksocket.

Η λύση του ξεχωριστού ανοιχτού αρχείου για κάθε ksocket απλοποιεί σημαντικά τη σχεδίαση των μεθόδων του οδηγού συσκευής και αυξάνει την απόδοση του στρώματος επικοινωνίας. Τα δεδομένα πολλών διαφορετικών συνδέσεων θα μπορούσαν, εναλλακτικά, να μεταφέρονται μαζί με τις αιτήσεις ελέγχου μέσω του ίδιου ανοιχτού αρχείου. Ωστόσο, μια τέτοια σχεδίαση θα απαιτούσε τον χωρισμό των δεδομένων προς μεταφορά σε πακέτα καθορισμένης μορφής και τη σαφή διάκριση του καθενός από το προηγούμενο και το επόμενο, με την εισαγωγή στοιχείων επικεφαλίδας. Κάτι τέτοιο θα οδηγούσε σε αυξημένο κόστος επεξεργασίας τόσο στο χώρο πυρήνα όσο και στο χώρο χρήστη, ώστε να είναι

δυνατή η πολυπλεξία και ο διαχωρισμός των πληροφοριών κατά τη μεταφορά τους ανάμεσα στους δύο χώρους.

Ο μοναδικός περιορισμός που εισάγεται από την παρούσα σχεδίαση προκύπτει από το μέγιστο αριθμό αρχείων που είναι δυνατό να έχει μια διεργασία ανοιχτά κάτω από το Linux, εφόσον έτσι περιορίζεται ο μέγιστος αριθμός ταυτόχρονων συνδέσεων. Στις τελευταίες εκδόσεις του Linux το όριο αυτό δεν είναι στατικό, αλλά μπορεί να ρυθμιστεί δυναμικά κατά τη λειτουργία του συστήματος. Επιπλέον, η προκαθορισμένη του τιμή είναι 8.192, οπότε δεν ενδέχεται να δημιουργήσει προβλήματα στη δημιουργία του στρώματος επικοινωνίας.

4.6 Υλοποίηση των υπηρεσιών του στρώματος επικοινωνίας

Ο τρόπος λειτουργίας των `ksocket_bind`, `ksocket_connect`, `ksocket_accept`, `ksocket_close` μπορεί να συνοψιστεί στα εξής βήματα:

- Έλεγχος της ορθής κατάστασης του ksocket, ανάλογα με τη λειτουργία που ζητείται
- Αλλαγή της κατάστασής του σε μία από τις μεταβατικές καταστάσεις
- Προώθηση του κατάλληλου αιτήματος στο χώρο χρήστη
- Αναμονή για τιμή επιστροφής από το χώρο χρήστη, όπως περιγράφηκε προηγούμενα

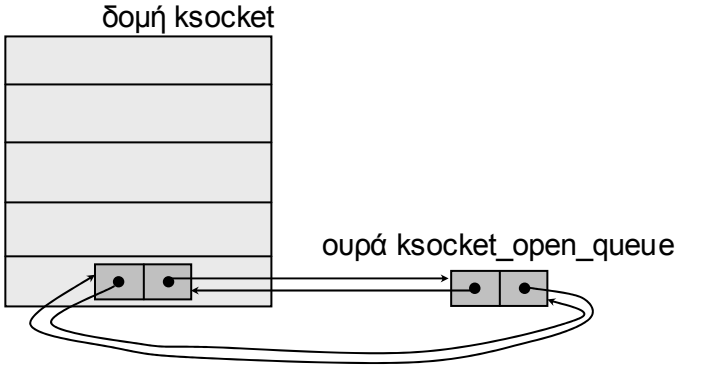
Οι συναρτήσεις `ksocket_read` και `ksocket_write` ελέγχουν την κατάσταση του ksocket, ώστε να εξασφαλιστεί ότι είναι δυνατή η ανταλλαγή δεδομένων και στη ανταλλάσσουν δεδομένα με τους αντίστοιχους απομονωτές E / E.

4.6.1 Υλοποίηση της ksocket_open

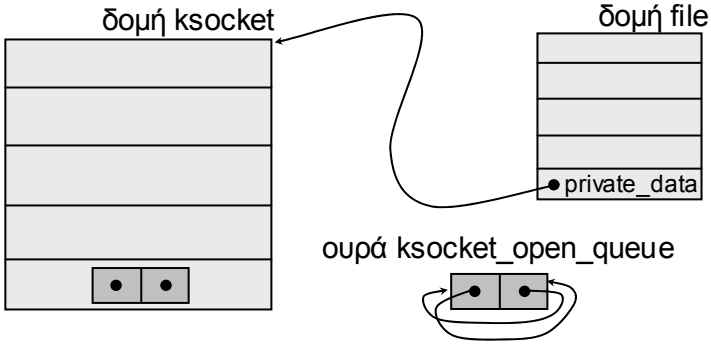
Ιδιαίτερη αναφορά χρειάζεται στην συνάρτηση `ksocket_open`, η οποία αναλαμβάνει τη δημιουργία ενός καναλιού επικοινωνίας με τη διεργασία χώρου χρήστη και τη συσχέτισή του με κάποιο ksocket. Σε αντίθεση με τις υπόλοιπες συναρτήσεις, όταν εκτελείται η `ksocket_open` δεν υπάρχει κανάλι επικοινωνίας συσχετισμένο με τη δομή ksocket και το πεδίο `user_data` δεν έχει έγκυρη τιμή.

Για να είναι δυνατή η συσχέτιση των ksockets με τα νέα αρχεία τα οποία ανοίγονται από τη διεργασία χώρου χρήστη διατηρείται από το χώρο πυρήνα μια ουρά από ksockets για τα οποία εκκρεμούν αιτήσεις `KSOCKET_REQ_OPEN`, που ονομάζεται `ksocket_open_queue` και υλοποιείται ως διπλά συνδεδεμένη λίστα. Πριν την εισαγωγή του αιτήματος `KSOCKET_REQ_OPEN` στην ουρά αιτήσεων, η `ksocket_open` εισάγει το ksocket στην `ksocket_open_queue`. Έτσι, όταν η διεργασία χώρου χρήστη δημιουργήσει ένα επιπλέον κανάλι επικοινωνίας με χρήση της open και ζητήσει την αντιστοίχισή του με κάποιο ksocket, μέσω κατάλληλης εντολής ioctl, η μέθοδος `ksocket_fops_ioctl` του οδηγού εικονικής συσκευής αφαιρεί ένα ksocket από την `ksocket_open_queue` και θέτει ανάλογα το πεδίο `private_data` της δομής file που αντιστοιχεί στο ανοιχτό αρχείο.

Η διαδικασία αυτή παρουσιάζεται σχηματικά ως εξής:



(α) Η δομή ksocket βρίσκεται στην ουρά ksocket_open_queue

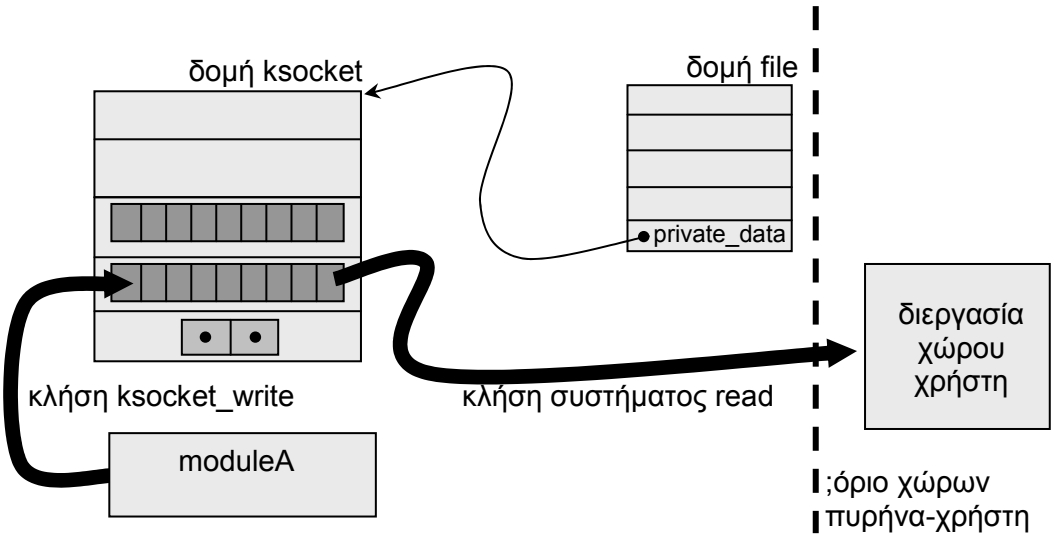


(β) Μια δομή file (ανοιχτό αρχείο) έχει συσχετιστεί με το ksocket

Σχήμα 4.5 – Ένα ksocket συσχετίζεται με κανάλι επικοινωνίας χώρου χρήστη

4.7 Παράδειγμα χρήσης του στρώματος επικοινωνίας

Ολοκληρώνοντας την παρουσίαση του τμήματος των KSockets που εκτελείται σε χώρο πυρήνα, μπορούμε να δούμε ένα παράδειγμα στο οποίο περιγράφονται οι διαδικασίες που συμβαίνουν ώστε να εγκατασταθεί μια σύνδεση ανάμεσα σε δύο τμήματα των απομακρυσμένων πυρήνων Α, Β με πρωτοβουλία του Α και να αποσταλεί μια σειρά από δεδομένα από τον Α στον Β.



Σχήμα 4.6 – Εγγραφή δεδομένων στον απομονωτή εξόδου και παραλαβή τους από τη διεργασία χώρου χρήστη

Οι διάφορες φάσεις της επικοινωνίας, από την πλευρά του τμήματος moduleA του πυρήνα Α έχουν ως εξής:

Βήμα 1: Το τμήμα moduleA του πυρήναΑ δημιουργεί μια νέα δομή ksocket με κλήση της `ksocket_create`

Βήμα 2: Καλείται η `ksocket_open` ώστε το ksocket να συνδεθεί με το χώρο χρήστη. Η `ksocket_open` το εισάγει στην `ksocket_open_queue` και προωθεί το αίτημα `KSOCKET_REQ_OPEN` στο χώρο χρήστη. Στη συνέχεια, κοιμάται σε ουρά αναμονής που αντιστοιχεί στο ksocket. Η διεργασία χώρου χρήστη δέχεται την αίτηση, εκτελεί την κλήση συστήματος `open`, οπότε δημιουργείται ένα νέο ανοιχτό αρχείο. Έπειτα εκτελεί την κλήση συστήματος `ioctl`, έτσι ώστε να συσχετιστεί το νέο αρχείο με το ksocket που βρίσκεται στην `ksocket_open_queue`. Τέλος εκτελεί για δεύτερη φορά την κλήση `ioctl` ώστε να

μεταβάλλει την κατάσταση του ksocket σε `KSOCKET_OPEN` και να ξυπνήσει την `ksocket_open`, που περιμένει σε αντίστοιχη ουρά αναμονής. Η `ksocket_open` επιστρέφει.

Βήμα 3: Το moduleA καλεί την συνάρτηση `ksocket_bind` ώστε να δεσμεύσει μια θύρα επικοινωνίας. Η `ksocket_bind` προωθεί κατάλληλο αίτημα στη διεργασία χώρου χρήστη η οποία δεσμεύει τη ζητούμενη θύρα επικοινωνίας. Οι λεπτομέρειες της δέσμευσης στο δίκτυο διασύνδεσης δεν γίνονται αντιληπτές από το χώρο πυρήνα. Τέλος, η διεργασία χώρου χρήστη εκτελεί την κλήση `ioctl` ώστε να μεταβάλλει την κατάσταση του ksocket σε `KSOCKET_BOUND` και να ξυπνήσει την `ksocket_bind`. Η `ksocket_bind` επιστρέφει.

Βήμα 4: Το moduleA καλεί την συνάρτηση `ksocket_connect`, ώστε να ζητήσει εξερχόμενη σύνδεση. Δρομολογείται το κατάλληλο αίτημα προς το χώρο χρήστη και όταν η σύνδεση εγκατασταθεί, μεταβάλλεται η κατάσταση του ksocket σε `KSOCKET_CONNECTED` με κλήση της `ioctl` από τη διεργασία χώρου χρήστη.

Βήμα 5: Καλείται η `ksocket_write` για αποστολή μιας ομάδας από bytes. Η `ksocket_write` αποθηκεύει τα δεδομένα στον απομονωτή εξόδου με χρήση της `buf_write`. Η διεργασία χώρου χρήστη, η οποία περιμένουμε ότι παρακολουθούσε την κατάσταση του απομονωτή εκτελώντας την κλήση συστήματος `poll` στο ανοιχτό αρχείο, διαβάζει τα δεδομένα με χρήση της κλήσης `read` (οπότε εκτελείται η `ksocket_fops_read` και τα αφαιρεί από τον απομονωτή) και τα αποστέλλει μέσω του δικτύου διασύνδεσης.

Αντίστοιχα είναι τα βήματα της επικοινωνίας και από την πλευρά του πυρήνα Β, για την αποδοχή της εισερχόμενης σύνδεσης και τη λήψη των δεδομένων.

Κεφάλαιο 5

Σχεδιασμός του στρώματος επικοινωνίας από την πλευρά του χώρου χρήστη

Έχοντας ολοκληρώσει την περιγραφή του σχεδιασμού του τμήματος του στρώματος επικοινωνίας που εκτελείται σε χώρο πυρήνα, προχωράμε στην περιγραφή του σχεδιασμού του τμήματος που εκτελείται στο χώρο χρήστη. Το τμήμα αυτό υλοποιείται ως διεργασία χώρου χρήστη και αναλαμβάνει την επικοινωνία πάνω από το δίκτυο διασύνδεσης. Στα καθήκοντά του συμπεριλαμβάνονται η επικοινωνία μέσω ενός καθορισμένου πρωτοκόλλου με το τμήμα χώρου πυρήνα, η αποδοχή αιτήσεων για εγκατάστασή και λύση συνδέσεων προς άλλα υπολογιστικά συστήματα και η μεταφορά των δεδομένων που μεταφέρονται πάνω από τις συνδέσεις αυτές, από τον τοπικό πυρήνα προς το δίκτυο διασύνδεσης και αντίστροφα.

5.1 Γενική περιγραφή της διεργασίας χώρου χρήστη

Η διεργασία χώρου χρήστη, με την οποία υλοποιείται το τμήμα των KSockets που εκτελείται σε χώρο χρήστη, είναι ένας *δαίμονας* (daemon) του λειτουργικού συστήματος Linux. Με τον όρο δαίμονας, εννοούμε μια διεργασία η οποία δεν αλληλεπιδρά με τους χρήστες του υπολογιστικού συστήματος, δεν σχετίζεται με κάποιο τερματικό, άρα δεν ελέγχεται από κάποιο τερματικό, αλλά εκτελείται μονίμως στο παρασκήνιο. Συνήθως, οι δαίμονες που εκτελούνται σε κάποιο υπολογιστικό σύστημα ξεκινούν αυτόματα, μόλις

ολοκληρωθεί η εκκίνηση του λειτουργικού συστήματος και τηρούν κατάλληλο *αρχείο καταγραφής* των ενεργειών τους (log file), έτσι ώστε ο διαχειριστής του υπολογιστικού συστήματος να μπορεί να ενημερώνεται για τυχόν προβλήματα που προκύπτουν κατά τη λειτουργία τους.

Ο δαίμονας του στρώματος επικοινωνίας, όπως προκύπτει από τις προδιαγραφές του συστήματος, οφείλει να επιτρέπει την ταυτόχρονη επικοινωνία επάνω από πολλές διαφορετικές εγκατεστημένες συνδέσεις. Για το λόγο αυτό, επιλέγουμε η διεργασία χώρου χρήστη που σχεδιάζουμε να είναι μια *πολυνηματική διεργασία*, που υλοποιείται με τη βοήθεια της βιβλιοθήκης *POSIX Threads*. Οι λόγοι που οδηγούν σε αυτή τη σχεδιαστική επιλογή παρουσιάζονται αναλυτικά στη συνέχεια, όπου γίνεται μια σύντομη περιγραφή της βιβλιοθήκης POSIX Threads και παρουσιάζονται τα πλεονεκτήματα και τα μειονεκτήματα της κατασκευής πολυνηματικών εφαρμογών.

Η διεργασία χώρου χρήστη των KSockets, αποτελείται από δύο διακριτά τμήματα, που συνεργάζονται:

- **Ένα τμήμα που εξαρτάται από το δίκτυο διασύνδεσης:** Το τμήμα αυτό είναι συγκεκριμένο ανάλογα με το δίκτυο διασύνδεσης που χρησιμοποιείται κάθε φορά. Διαχειρίζεται τους πόρους του δικτύου και αναλαμβάνει την μεταφορά των δεδομένων επάνω από αυτό. Προβάλλει μια αφαιρετική εικόνα του δικτύου, η οποία επιτρέπει την εγκατάσταση και λύση αξιόπιστων συνδέσεων.

- **Ένα τμήμα ανεξάρτητο του χρησιμοποιούμενου δικτύου διασύνδεσης:** Το τμήμα αυτό αναλαμβάνει την επικοινωνία με το χώρο πυρήνα, μέσω του καθορισμένου πρωτοκόλλου που περιγράφηκε στο προηγούμενο κεφάλαιο. Δέχεται αιτήσεις από τον χώρο πυρήνα και χρησιμοποιεί τις υπηρεσίες του τμήματος που εξαρτάται από το δίκτυο διασύνδεσης για να τις ικανοποιήσει. Επιπλέον, είναι υπεύθυνο για τη δημιουργία και καταστροφή των νημάτων εκτέλεσης, από τα οποία αποτελείται η διεργασία.

5.2 Πολυνηματική σχεδίαση της διεργασίας χώρου χρήστη

5.2.1 Διεργασίες και νήματα

Τα περισσότερα λειτουργικά συστήματα υποστηρίζουν, συμπληρωματικά προς τις διεργασίες ως μονάδες χρονοδρομολόγησης, τα *νήματα εκτέλεσης* (threads). Η σημασία των νημάτων μπορεί να γίνει περισσότερο κατανοητή, συγκρίνοντας την έννοια της διεργασίας με αυτή του νήματος.

Με τον όρο διεργασία εννοούμε ένα υπό εκτέλεση πρόγραμμα. Κάθε διεργασία εκτελείται σε ιδιωτικό χώρο μνήμης και μπορεί να προσπελάσει έναν ιδιωτικό χώρο διευθύνσεων. Ο χώρος εικονικής μνήμης που έχει διατεθεί σε κάθε διεργασία είναι διαφορετικός και ανεξάρτητος από τον χώρο των υπολοίπων διεργασιών, έτσι ώστε να εξασφαλίζεται η απομόνωση της από αυτές κατά τη λειτουργία της. Οι διεργασίες που εκτελούνται ταυτόχρονα κάτω από το λειτουργικό σύστημα συναγωνίζονται για τη χρήση των ΚΜΕ του υπολογιστικού συστήματος και υφίστανται χρονοδρομολόγηση. Επιπλέον, για κάθε διεργασία το λειτουργικό σύστημα τηρεί κατάλληλες διαχειριστικές πληροφορίες που αφορούν τους πόρους που έχουν διατεθεί σε αυτή την κατάσταση στην οποία βρίσκεται. Στις πληροφορίες αυτές συμπεριλαμβάνονται:

- Η *ταυτότητα διεργασίας* (Process ID), ένας αριθμός μοναδικός για τη διεργασία, ώστε αυτή να διακρίνεται από όλες τις υπόλοιπες
- Ο αριθμός χρήστη με τα δικαιώματα του οποίου εκτελείται η διεργασία, ο οποίος καθορίζει τα δεδομένα τα οποία επιτρέπεται να προσπελάσει και τις λειτουργίες που επιτρέπεται να εκτελέσει
- Ο πίνακας των ανοιχτών αρχείων της διεργασίας
- Οι τιμές των καταχωρητών και του μετρητή προγράμματος ώστε να είναι δυνατή η περιοδική διακοπή και επανεκκίνηση της εκτέλεσής της κατά τη χρονοδρομολόγηση

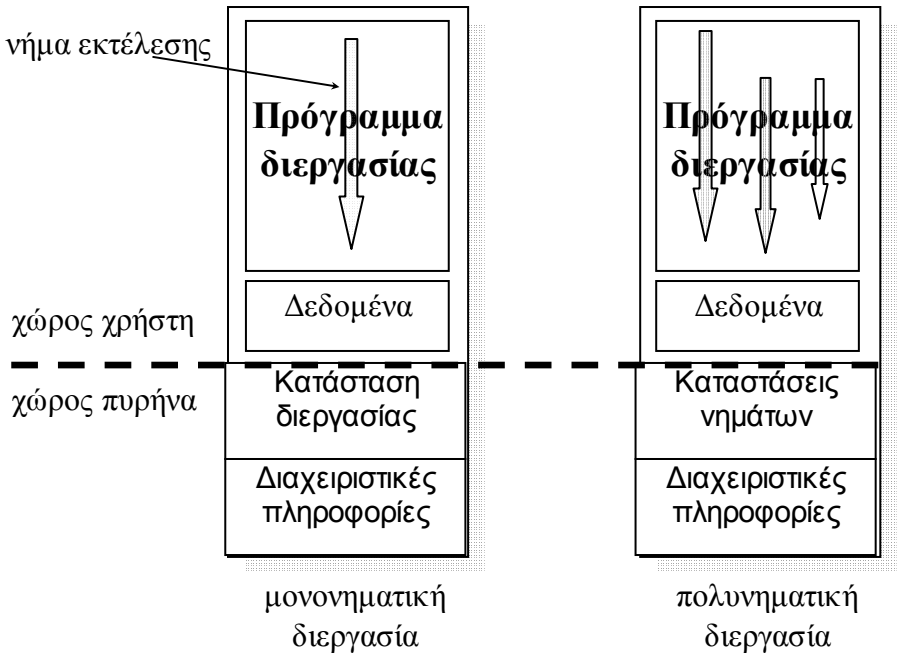
Στην περίπτωση μιας συνηθισμένης διεργασίας, ένα μόνο τμήμα του προγράμματός της εκτελείται κάθε φορά, υπάρχει δηλαδή μόνο μία *ροή εκτέλεσης* (flow of execution). Αν χρειάζεται δύο τμήματα ενός προγράμματος να εκτελούνται ταυτόχρονα, τότε μπορούν να

δημιουργηθούν περισσότερες από μία διεργασίες, οι οποίες όμως παραμένουν ανεξάρτητες και για κάθε μία διατηρούνται χωριστά διαχειριστικές πληροφορίες από το Λ.Σ.

Τα νήματα αναπαριστούν *διαφορετικές* ροές εκτέλεσης, μέσα στην *ίδια* διεργασία. Είναι διακριτά, ανεξάρτητα περιβάλλοντα εκτέλεσης, τα οποία όμως ζουν μέσα στην ίδια διεργασία και ακολουθούν εντολές του ίδιου προγράμματος. Έτσι, δύο νήματα τα οποία ανήκουν στην ίδια διεργασία μοιράζονται τις διαχειριστικές πληροφορίες που αντιστοιχούν σε αυτή: Χαρακτηρίζονται από την ίδια ταυτότητα διεργασίας (διακρίνονται με βάση την ταυτότητα νήματος), έχουν τα ίδια δικαιώματα κατά την πρόσβαση στους πόρους του υπολογιστικού συστήματος, διαθέτουν κοινό πίνακα ανοιχτών αρχείων και μπορούν να προσπελάσουν τον ίδιο χώρο εικονικής μνήμης. Εφόσον όμως κάθε νήμα αποτελεί μια ανεξάρτητη ροή εκτέλεσης, είναι αναγκαίο να διατηρείται η πληροφορία κατάστασής του (μετρητής προγράμματος, τιμές καταχωρητών, στοίβα) ξεχωριστή από αυτή των υπόλοιπων νημάτων.

Μια πολυνηματική διεργασία (multithreaded process) αποτελεί το διαχειριστικό πλαίσιο μέσα στο οποίο δραστηριοποιούνται πολλά διαφορετικά νήματα εκτέλεσης, τα οποία πλέον αποτελούν τις βασικές μονάδες χρονοδρομολόγησης για το λειτουργικό σύστημα[11]. Το γεγονός αυτό παρουσιάζεται στο ακόλουθο σχήμα:

[11] Αυτό δεν είναι απόλυτα ακριβές στην – σπάνια – περίπτωση κατά την οποία η υλοποίηση των νημάτων εκτέλεσης γίνεται αποκλειστικά στο χώρο χρήστη, χωρίς την παρέμβαση του Λ.Σ., οπότε η χρονοδρομολόγησή τους γίνεται από την ίδια τη διεργασία και τα νήματα δεν μπορούν να διακριθούν από το Λ.Σ.



Σχήμα 5.1 – Τα νήματα ως ανεξάρτητες ροές εκτέλεσης μέσα σε μία διεργασία

5.2.2 Πλεονεκτήματα και μειονεκτήματα μιας πολυνηματικής σχεδίασης

Η σχεδίαση μιας εφαρμογής έτσι ώστε να εκτελείται ως πολυνηματική διεργασία παρέχει ορισμένα σημαντικά πλεονεκτήματα σε σχέση με την σειριακή εκτέλεση ενός προγράμματος από μία μόνο διεργασία, ή την επίτευξη παραλληλίας με χρήση πολλών ταυτόχρονων διεργασιών:

- Στα περισσότερα λειτουργικά συστήματα, η διαχείριση νημάτων εκτέλεσης έχει πολύ μικρότερες απαιτήσεις σε χρόνο από τη διαχείριση διεργασιών. Η δημιουργία και ο τερματισμός νημάτων δεσμεύει λιγότερους πόρους συστήματος και έχει μικρότερη επίπτωση στο χρόνο εκτέλεσης του προγράμματος. Αυτό συμβαίνει διότι μόνο ένα πολύ μικρό τμήμα των πληροφοριών που τηρεί το λειτουργικό σύστημα για κάθε διεργασία, αυτό που απαιτείται για την υποστήριξη χρονοδρομολόγησης, χρειάζεται να αντιγραφεί και να καταστραφεί κατά τη δημιουργία και τον τερματισμό νημάτων εκτέλεσης. Οι πληροφορίες διαχείρισης παραμένουν αμετάβλητες, εφόσον τις μοιράζονται όλα τα νήματα εκτέλεσης.

- Η μεταγωγή περιεχομένου (context switch) ανάμεσα σε νήματα εκτέλεσης της ίδιας διεργασίας εισάγει μικρότερη καθυστέρηση σε σύγκριση με την μεταγωγή περιεχομένου ανάμεσα σε δύο ανεξάρτητες, απομονωμένες διεργασίες. Αυτό

οφείλεται στο γεγονός ότι εφόσον τα νήματα έχουν προσπέλαση στον ίδιο χώρο εικονικής μνήμης, δεν χρειάζονται ενέργειες που αφορούν στη διαχείριση μνήμης κατά τη μεταγωγή περιεχομένου και δεν γίνεται καμία αλλαγή στους πίνακες σελίδων του υποσυστήματος εικονικής μνήμης.

- Η επικοινωνία και η ανταλλαγή δεδομένων ανάμεσα σε νήματα της ίδιας διεργασίας είναι σαφώς απλούστερη και αποδοτικότερη όσον αφορά την καθυστέρηση που εισάγεται, σε σχέση με την επικοινωνία ανάμεσα σε δύο διεργασίες. Τα δεδομένα που είναι αποθηκευμένα στο χώρο εικονικής μνήμης μιας διεργασίας δεν είναι προσβάσιμα από τις υπόλοιπες διεργασίες άμεσα, αλλά πρέπει να χρησιμοποιηθούν συγκεκριμένοι μηχανισμοί επικοινωνίας ανάμεσα τους (Inter-Process Communication mechanisms) που παρέχονται από το λειτουργικό σύστημα, όπως είναι οι *σωληνώσεις* (pipes) και οι *ουρές μηνυμάτων*. Έτσι, η ανταλλαγή δεδομένων ανάμεσα σε διεργασίες επιτυγχάνεται με χρήση κλήσεων του λειτουργικού συστήματος, που εισάγουν κάποιο κόστος επικοινωνίας. Αντίθετα, δύο νήματα της ίδιας διεργασίες εκτελούνται στον ίδιο χώρο εικονικής μνήμης, οπότε έχουν άμεση πρόσβαση σε όλες τις δομές δεδομένων που χρειάζονται κατά τη λειτουργία τους. Αν απαιτείται ανταλλαγή μηνυμάτων ανάμεσα σε νήματα, αυτή μπορεί να πραγματοποιηθεί με εντολές αποθήκευσης / ανάγνωσης από την ίδια περιοχή μνήμης, χρησιμοποιούνται δηλαδή μηχανισμοί μοιραζόμενης μνήμης, χωρίς την ανάγκη να παρέμβει το λειτουργικό σύστημα.

- Δύο νήματα εκτέλεσης μπορούν να μοιράζονται εντελώς φυσικά όχι μόνο το χώρο εικονικής μνήμης αλλά και τους υπόλοιπους πόρους του συστήματος. Για παράδειγμα, ένα αρχείο που ανοίγεται από κάποιο νήμα είναι άμεσα προσβάσιμο από όλα τα υπόλοιπα χωρίς καμία ιδιαίτερη ενέργεια, με τον ίδιο περιγραφητή αρχείου, αφού τα νήματα μοιράζονται τον πίνακα ανοιχτών αρχείων της διεργασίας.

Ωστόσο, η ύπαρξη ενός ενιαίου χώρου εικονικής μνήμης για όλα τα νήματα εκτέλεσης μιας διεργασίας εισάγει ορισμένα μειονεκτήματα:

- Σε αντίθεση με τις διεργασίες, κάθε νήμα δεν είναι προστατευμένο από τις ενέργειες των υπολοίπων. Πιθανή δυσλειτουργία μιας διεργασίας, π.χ. η εγγραφή άκυρων δεδομένων στο χώρο εικονικής μνήμης της δεν είναι δυνατό να επηρεάσει τη λειτουργία των υπολοίπων, ωστόσο αυτό δεν ισχύει στην περίπτωση των

νημάτων. Προβληματική λειτουργία ενός νήματος και ανεξέλεγκτη αλλαγή δεδομένων στον χώρο εικονικής μνήμης του είναι δυνατό να επηρεάσει και τη λειτουργία των υπολοίπων.

- Ο κοινός χώρος εικονικής μνήμης επιβάλλει τη χρήση κατάλληλων μηχανισμών συγχρονισμού (π.χ. σηματοφορείς) από τα νήματα, έτσι ώστε να εξασφαλίζεται η ακεραιότητα των μοιραζόμενων δεδομένων κατά την ταυτόχρονη πρόσβαση σε αυτά.

5.2.2.1 Ανάγκη για πολυνηματική σχεδίαση των KSockets

Στην περίπτωση της διεργασίας χώρου χρήστη των KSockets, οδηγούμαστε στη σχεδίαση μιας πολυνηματικής εφαρμογής από την ανάγκη για υποστήριξη πολλών ταυτόχρονων συνδέσεων, διατηρώντας όμως όσο το δυνατόν μικρότερη την καθυστέρηση (latency) που εισάγεται κατά την επικοινωνία. Επιπλέον, είναι επιθυμητό το στρώμα επικοινωνίας να μπορεί να εκμεταλλευτεί τις δυνατότητες των συστημάτων συμμετρικής πολυεπεξεργασίας, τα οποία χρησιμοποιούνται όλο και συχνότερα ως οι δομικές μονάδες για συστοιχίες υπολογιστών.

Μια σχεδίαση βασισμένη στην ύπαρξη μοναδικής, μονονηματικής διεργασίας είναι φανερό πως δεν πληρεί τις προδιαγραφές του στρώματος επικοινωνίας. Θα μπορούσε να υποστηρίζει πολλές ταυτόχρονες συνδέσεις, εξυπηρετώντας περιοδικά την κάθε μία από αυτές, ωστόσο αυτό θα είχε σημαντική επίπτωση στην εισαγόμενη καθυστέρηση. Επιπλέον, δεν μπορεί να εκμεταλλευτεί την ύπαρξη πολλών ΚΜΕ, εφόσον μόνο μία από αυτές θα μπορούσε να εκτελεί σε κάθε στιγμή κώδικα της διεργασίας χώρου χρήστη.

Η ανάγκη για παραλληλία ίσως μας οδηγούσε σε σχεδίαση βασισμένη σε πολλές, ανεξάρτητες διεργασίες, που θα υλοποιούσαν από κοινού τις λειτουργίες του στρώματος επικοινωνίας στο χώρο χρήστη. Οι διεργασίες αυτές μπορούν να εκτελούνται ταυτόχρονα, σε διαφορετικές ΚΜΕ. Ωστόσο, παραμένει το μειονέκτημα του σημαντικού κόστους επικοινωνίας ανάμεσα στις διεργασίες, αφού απαιτείται η παρέμβαση του λειτουργικού συστήματος ώστε να πραγματοποιηθεί.

Έτσι, οδηγούμαστε στη λύση της μοναδικής διεργασίας που αποτελείται από πολλά, παράλληλα νήματα εκτέλεσης. Τα νήματα μπορούν να χρονοδρομολογηθούν σε διαφορετικές ΚΜΕ από το λειτουργικό σύστημα, οπότε εκτελούνται αποδοτικά σε

συστήματα SMP. Επιπλέον, επικοινωνούν αποδοτικά μεταξύ τους μέσω του κοινού χώρου εικονικής μνήμης.

5.2.3 Υποστήριξη του Linux για POSIX Threads

Από την πρώτη στιγμή που έγινε φανερή η ανάγκη για την υποστήριξη πολυνηματικών εφαρμογών, δημιουργήθηκαν πολλές διαφορετικές, ασύμβατες μεταξύ τους και εξαρτώμενες από το λειτουργικό σύστημα βιβλιοθήκες για τη διαχείριση νημάτων εκτέλεσης. Κάθε βιβλιοθήκη νημάτων διέφερε σημαντικά ως προς τις προσφερόμενες δυνατότητες και τον τρόπο χρήσης σε σχέση με τις υπόλοιπες, δυσχεραίνοντας έτσι σημαντικά τη συγγραφή εφαρμογών με μεταφέρσιμο, ανεξάρτητο από την αρχιτεκτονική ή το λειτουργικό σύστημα τρόπο. Το γεγονός αυτό οδήγησε το 1995 στην καθιέρωση του προτύπου IEEE POSIX 1003.1c, το οποίο προδιαγράφει βιβλιοθήκες δημιουργίες πολυνηματικών εφαρμογών κάτω από λειτουργικά συστήματα που ακολουθούν τη φιλοσοφία του UNIX. Οι υλοποιήσεις του προτύπου αυτού αναφέρονται συνήθως ως βιβλιοθήκες "POSIX Threads".

Κάτω από το Linux το πρότυπο των POSIX Threads υποστηρίζεται μέσω της βιβλιοθήκης *LinuxThreads* [8], η οποία και χρησιμοποιήθηκε για την ανάπτυξη της διεργασίας χώρου χρήστη των KSockets. Από την πλευρά του πυρήνα του Linux, διεργασίες και νήματα αντιμετωπίζονται σχεδόν με τον ίδιο τρόπο: Τα νήματα υλοποιούνται ως διαφορετικές διεργασίες που μοιράζονται όμως τον ίδιο χώρο εικονικής μνήμης και τις ίδιες πληροφορίες διαχείρισης. Συνηθισμένες, απομονωμένες διεργασίες κατασκευάζονται με χρήση της κλήσης συστήματος `fork`. Αντίθετα, η κλήση συστήματος `clone` κατασκευάζει μια νέα διεργασία η οποία μοιράζεται το χώρο μνήμης της γονικής διεργασίας, τον πίνακα ανοιχτών αρχείων, κλπ. Έτσι, χρησιμοποιείται εσωτερικά από τη βιβλιοθήκη LinuxThreads για τη δημιουργία νέων νημάτων. Ωστόσο αυτή η ιδιαιτερότητα της υλοποίησης δεν γίνεται αντιληπτή κατά τη χρήση της βιβλιοθήκης LinuxThreads, αφού τότε καλούνται οι συναρτήσεις που προδιαγράφονται από το πρότυπο των POSIX Threads και ποτέ απευθείας η `clone`.

5.2.4 Προσφερόμενες υπηρεσίες των POSIX Threads

Το πρότυπο POSIX Threads ορίζει μια σειρά από υπηρεσίες που προσφέρονται για τη δημιουργία πολυνηματικών εφαρμογών. Οι υπηρεσίες αυτές μπορούν να χρησιμοποιηθούν

μέσω της κλήσης κατάλληλων συναρτήσεων από τη γλώσσα C, που είναι και η μόνη γλώσσα προγραμματισμού για την οποία γίνεται προς το παρόν πρόβλεψη στο πρότυπο. Μπορούμε να διακρίνουμε τρεις κατηγορίες υπηρεσιών:

- **Υπηρεσίες που αφορούν τη διαχείριση των νημάτων εκτέλεσης:** Στην κατηγορία αυτή συμπεριλαμβάνονται συναρτήσεις για τη δημιουργία και τον τερματισμό νημάτων, όπως είναι οι `pthread_create`, `pthread_join`, `pthread_cancel` και άλλες.

- **Υπηρεσίες σηματοφορέων:** Για το συγχρονισμό ανάμεσα σε νήματα εκτέλεσης και την επίλυση του προβλήματος του κρίσιμου τμήματος, το πρότυπο προβλέπει την ύπαρξη σηματοφορέων. Προσφέρονται συναρτήσεις για τη δημιουργία (`pthread_mutex_init`) και την καταστροφή (`pthread_mutex_destroy`) σηματοφορέων, καθώς και συναρτήσεις για την εκτέλεση της πράξης P (`pthread_mutex_lock`) και της πράξης V (`pthread_mutex_unlock`) σε κάποιο σηματοφορέα.

- **Υπηρεσίες για ανταλλαγή ειδοποιήσεων ανάμεσα σε νήματα:** Το πρότυπο των POSIX Threads προβλέπει μια ειδική κατηγορία μοιραζόμενων μεταβλητών, τις *μεταβλητές συνθήκης* (condition variables). Μέσω των μεταβλητών συνθήκης ένα νήμα εκτέλεσης μπορεί να διακόψει προσωρινά την εκτέλεσή του και να περιμένει σε κάποια ουρά αναμονής, έως ότου ειδοποιηθεί από κάποιο άλλο νήμα ότι μπορεί να συνεχίσει, έως ότου δηλαδή ικανοποιηθεί κάποια συνθήκη. Προσφέρονται υπηρεσίες για αρχικοποίηση και καταστροφή των μεταβλητών συνθήκης (`pthread_cond_create`, `pthread_cond_destroy`) καθώς και υπηρεσίες για προσθήκη σε ουρά αναμονής που σχετίζεται με κάποια μεταβλητή συνθήκης (`pthread_cond_wait`) και ειδοποίηση νημάτων που περιμένουν σε κάποια ουρά (`pthread_cond_signal`, `pthread_cond_broadcast`).

5.3 Τμήμα ανεξάρτητο από το δίκτυο διασύνδεσης

Το τμήμα της διεργασίας χρήστη που είναι ανεξάρτητο του χρησιμοποιούμενου δικτύου διασύνδεσης έχει τις εξής αρμοδιότητες:

- Επικοινωνεί με το τμήμα χώρου πυρήνα μέσω καθορισμένου πρωτοκόλλου

- Δέχεται αιτήσεις από το χώρο πυρήνα και τις ικανοποιεί χρησιμοποιώντας τις υπηρεσίες του τμήματος που εξαρτάται από το δίκτυο διασύνδεσης

- Διαχειρίζεται τα νήματα εκτέλεσης, δημιουργώντας και τερματίζοντας νήματα όποτε αυτό είναι απαραίτητο

Ο τρόπος με τον οποίο πραγματοποιούνται οι παραπάνω λειτουργίες παρουσιάζεται αναλυτικά στις επόμενες παραγράφους.

5.3.1 Κανάλια επικοινωνίας με το χώρο πυρήνα

Στο προηγούμενο κεφάλαιο έγινε παρουσίαση του μοντέλου επικοινωνίας ανάμεσα στο χώρο πυρήνα και το χώρο χρήστη από την πλευρά όμως του πυρήνα (§4.1, §4.5). Στην παράγραφο αυτή, παρουσιάζεται η επικοινωνία από την πλευρά του χώρου χρήστη.

Η διεργασία χώρου χρήστη επικοινωνεί με το χώρο πυρήνα μέσω της ειδικής εικονικής συσκευής χαρακτήρων. Η συσκευή αυτή μπορεί να προσπελαστεί από το χώρο χρήστη, εκτελώντας τις συνηθισμένες κλήσεις συστήματος για πρόσβαση σε αρχεία (`open`, `close`, `read`, `write` κ.ά.) στο ειδικό αρχείο συσκευής, π.χ. το `/dev/ksocket0`, που αποτελεί την αναπαράσταση της εικονικής συσκευής χαρακτήρων στο σύστημα αρχείων. (§2.1)

Έχοντας ανοίξει ένα κανάλι επικοινωνίας με το χώρο πυρήνα εκτελώντας την κλήση συστήματος `open` στο ειδικό αρχείο συσκευής, η διεργασία χώρου χρήστη μπορεί να δεχθεί αιτήσεις από τον χώρο πυρήνα μέσω της κλήσης συστήματος `read`, η οποία τις ανακτά από την ουρά των αιτήσεων. Όταν χρειάζεται η δημιουργία ενός νέου καναλιού επικοινωνίας με το χώρο πυρήνα, ώστε να είναι δυνατή η σύνδεση μιας δομής ksocket με το χώρο χρήστη και η εγκατάσταση νέας σύνδεσης, η διεργασία χώρου χρήστη εκτελεί εκ νέου την κλήση συστήματος `open` στο ειδικό αρχείο συσκευής, ώστε να αποκτήσει ένα νέο ανοιχτό αρχείο, προσβάσιμο μέσω ενός διαφορετικού περιγραφητή αρχείων.

Η συσχέτιση του νέου ανοιχτού αρχείου με μια δομή ksocket του χώρου πυρήνα γίνεται μέσω κατάλληλης εντολής `ioctl`. Η εντολή ioctl `KSOCKET_IOC_SETCOCKET`, ζητά από τον οδηγό της εικονικής συσκευής χαρακτήρων τη συσχέτιση του νέου ανοιχτού αρχείου με κάποια από τις δομές ksocket για την οποία έχει υποβληθεί αίτημα `KSOCKET_REQ_OPEN` και βρίσκεται στην ουρά αναμονής για σύνδεση με το χώρο χρήστη. Μαζί με την εντολή `KSOCKET_IOC_SETSOCKET`, η διεργασία χώρου χρήστη περνά στο χώρο πυρήνα ένα δείκτη

προς τα ιδιωτικά δεδομένα της, τα οποία αφορούν τη συγκεκριμένη σύνδεση. Έτσι, κάθε ένα από τα επόμενα αιτήματα του πυρήνα που αφορά το συγκεκριμένο ksocket, θα συμπεριλαμβάνει την τιμή αυτού του δείκτη στο πεδίο `user_data` της αίτησης, ώστε η διεργασία χώρου χρήστη να μπορεί να ανακτήσει τις ανάλογες πληροφορίες διαχείρισης του δικτύου διασύνδεσης (§4.3.2).

Όταν ένα ανοιχτό αρχείο έχει συσχετιστεί με μια δομή ksocket, η διεργασία χώρου χρήστη μπορεί να χρησιμοποιήσει την εντολή ioctl `KSOCKET_IOC_SETSTATE` σε αυτό, έτσι ώστε να ενημερώσει το χώρο πυρήνα για μεταβολές της κατάστασης του ksocket, ανταποκρινόμενη σε αιτήματα του πυρήνα αλλά και σε γεγονότα που συμβαίνουν στο δίκτυο διασύνδεσης.

5.3.2 Η δομή uksocket

Το αντίστοιχο της δομής ksocket για τη διεργασία χώρου χρήστη είναι η *δομή uksocket* (Userspace ksocket). Σε κάθε δομή uksocket του χώρου χρήστη αντιστοιχεί μια δομή ksocket του χώρου πυρήνα. Η σύνδεση ανάμεσά τους γίνεται όταν εκτελείται η εντολή ioctl `KSOCKET_IOC_SETSOCKET`, για τη συσχέτιση ενός ανοιχτού αρχείου της διεργασίας χώρου χρήστη με μια δομή ksocket. Η δομή uksocket περιέχει πεδία ανάλογα με τα πεδία της δομής ksocket, εφόσον οι δομές δεδομένων που αποθηκεύονται στο χώρο πυρήνα δεν είναι άμεσα προσπελάσιμες από το χώρο χρήστη. Επιπλέον όμως αυτών, περιέχει και πεδία που εξαρτώνται από το δίκτυο διασύνδεσης που χρησιμοποιείται κάθε φορά. Αυτά μεταβάλλονται μόνο από το τμήμα το οποίο εξαρτάται από το δίκτυο διασύνδεσης και περιέχουν διαχειριστικές πληροφορίες που αφορούν τους πόρους του δικτύου διασύνδεσης που έχουν διατεθεί σε κάθε εγκατεστημένη σύνδεση.

πεδία	σημασία πεδίων
`int port, remote port`	τοπική και απομακρυσμένη θύρα επικοινωνίας KSockets
`char remote_name[]`	όνομα απομακρυσμένου πυρήνα
`ksocket_state_t state`	πεδίο κατάστασης για τη δομή uksocket
`int fd`	περιγραφητής αρχείου για επικοινωνία με χώρο πυρήνα
`πεδία εξαρτώμενα από το δίκτυο διασύνδεσης`	Τα πεδία αυτά καθορίζονται από το δίκτυο διασύνδεσης που χρησιμοποιείται για τη μεταφορά των δεδομένων

Πίνακας 5.1 – Πεδία της δομής uksocket

Τα σημαντικότερα από τα πεδία της δομής uksocket είναι:

- Το πεδίο `state`, στο οποίο καταγράφεται η τρέχουσα κατάσταση της αντίστοιχης δομής ksocket. Η σημασία του και οι τιμές του είναι όμοιες με αυτές του πεδίου state της δομής ksocket.

- Το πεδίο `fd`, στο οποίο αποθηκεύεται ο περιγραφητής του ανοιχτού αρχείου για πρόσβαση στη αντίστοιχη δομή ksocket. Ο περιγραφητής αρχείου `fd` καθορίζει το ανοιχτό αρχείο για το οποίο εκτελείται η κλήση συστήματος `ioctl` όταν η διεργασία χώρου χρήστη χρειαστεί να ενημερώσει τον πυρήνα για αλλαγή της κατάστασης του ksocket.

- Τα πεδία `port`, `remote_port`, `remote_name`, στα οποία αποθηκεύεται ο αριθμός της τοπικής και της απομακρυσμένης θύρας επικοινωνίας των KSockets όταν είναι εγκατεστημένη σύνδεση επάνω από το αντίστοιχο ksocket, καθώς και το όνομα του απομακρυσμένου πυρήνα.

- Ένα σύνολο πεδίων που εξαρτώνται από το δίκτυο διασύνδεσης. Τα πεδία αυτά περιγράφονται αναλυτικότερα στα επόμενα κεφάλαια, όπου παρουσιάζεται η υποστήριξη του στρώματος επικοινωνίας για το δίκτυο διασύνδεσης SCI και δίκτυα διασύνδεσης βασισμένα στο πρωτόκολλο TCP/IP.

5.3.3 Διαπροσωπεία του τμήματος διαχείρισης του δίκτυου διασύνδεσης

Το τμήμα της διεργασίας χρήστη που εξαρτάται από το δίκτυο διασύνδεσης υλοποιεί μια καθορισμένη διαπροσωπεία και προσφέρει συγκεκριμένες υπηρεσίες στο τμήμα που δεν εξαρτάται από το δίκτυο διασύνδεσης, έτσι ώστε να είναι δυνατή η αντικατάστασή του χωρίς να μεταβληθεί ο τρόπος λειτουργίας του στρώματος επικοινωνίας. Οι υπηρεσίες αυτές αφορούν μια αφαιρετική όψη του δικτύου διασύνδεσης, η οποία υποστηρίζει αξιόπιστες συνδέσεις για τη μεταφορά δεδομένων. Είναι ευθύνη του τμήματος της διεργασίας χώρου χρήστη που διαχειρίζεται το δίκτυο διασύνδεσης να προσαρμόσει το δίκτυο διασύνδεσης σε αυτή την αφαιρετική όψη, αποκρύπτοντας τις λεπτομέρειες της επικοινωνίας πάνω από αυτό.

Οι υπηρεσίες που προσφέρονται από το τμήμα της διεργασίας χώρου χρήστη που εξαρτάται από το δίκτυο διασύνδεσης παρουσιάζονται συνοπτικά στον παρακάτω πίνακα:

υπηρεσία	λειτουργία υπηρεσίας
`protocol_start`	αρχικοποιεί το δίκτυο διασύνδεσης
`protocol_stop`	τερματίζει τη χρήση του δικτύου διασύνδεσης
`protocol_init`	αρχικοποιεί τα πεδία της δομής uksocket που εξαρτώνται από το δίκτυο διασύνδεσης
`protocol_destroy`	απελευθερώνει πόρους που δέσμευσε η protocol_init
`protocol_open`	δεσμεύει πόρους του δικτύου για νέο uksocket
`protocol_close`	τερματίζει τη χρήση μιας δομής uksocket
`protocol_bind`	δεσμεύει πόρους του δικτύου για νέα θύρα επικοινωνίας
`protocol_connect`	εγκαθιστά σύνδεση με απομακρυσμένο πυρήνα
`protocol_accept`	αναμένει αίτηση σύνδεσης από απομακρυσμένο πυρήνα
`protocol_main_loop`	αναλαμβάνει την ανταλλαγή δεδομένων ανάμεσα στον τοπικό πυρήνα και το δίκτυο διασύνδεσης

Πίνακας 5.2 – Υπηρεσίες που υλοποιούνται για την υποστήριξη ενός δικτύου διασύνδεσης από το στρώμα επικοινωνίας

Η ύπαρξη καθορισμένης διαπροσωπείας επιτρέπει την επέκταση του στρώματος επικοινωνίας ώστε να υποστηρίζει διαφορετικά μεταξύ τους δίκτυα διασύνδεσης, αρκεί για κάθε ένα από αυτά να γραφεί κώδικας που προσαρμόζει τη λειτουργία τους στην αφαιρετική όψη των υπηρεσιών που προαναφέραμε.

5.3.4 Ικανοποίηση των αιτήσεων του πυρήνα

Η διεργασία χώρου χρήστη διατηρεί κατά τη λειτουργία της ένα νήμα εκτέλεσης το οποίο λαμβάνει αιτήσεις από το χώρο πυρήνα και δημιουργεί νέα νήματα, όποτε αυτό είναι αναγκαίο, τα οποία καλούν τις υπηρεσίες `protocol_*` του τμήματος που διαχειρίζεται το δίκτυο διασύνδεσης. Στη συνέχεια, περιγράφονται συνοπτικά οι ενέργειες που απαιτούνται για την ικανοποίηση καθενός από τους δυνατούς τύπους αιτημάτων του χώρου πυρήνα:

- **τύπος KSOCKET_REQ_OPEN:** Η διεργασία δημιουργεί ένα ακόμη ανοικτό αρχείο με χρήση της κλήσης συστήματος `open`, καθώς και μια νέα δομή uksocket. Τα πεδία της δομής που εξαρτώνται από το δίκτυο διασύνδεσης αρχικοποιούνται με κλήση της `protocol_init`. Στη συνέχεια εκτελείται η κλήση `ioctl KSOCKET_IOC_SETSOCKET` για τη συσχέτιση του νέου ανοικτού αρχείου με τη δομή ksocket του χώρου πυρήνα και τη δομή uksocket του χώρου χρήστη. Τέλος, καλείται η υπηρεσία `protocol_open` για τη δέσμευση πόρων του δικτύου διασύνδεσης.

- **τύπος KSOCKET_REQ_BIND:** Χρησιμοποιείται η υπηρεσία `protocol_bind` για τη δέσμευση των πόρων του δικτύου διασύνδεσης που αντιστοιχούν σε συγκεκριμένη θύρα επικοινωνίας των KSockets.

- **τύποι KSOCKET_REQ_ACCEPT, KSOCKET_REQ_CONNECT:** Δημιουργείται ένα νέο νήμα εκτέλεσης, ώστε η σύνδεση με κάποιον απομακρυσμένο πυρήνα να εξελίσσεται παράλληλα με την επεξεργασία των αιτήσεων του τοπικού πυρήνα. Στο νέο νήμα καλείται η υπηρεσία `protocol_accept` ή η υπηρεσία `protocol_connect` ανάλογα με το είδος του αιτήματος και όταν εγκατασταθεί μία νέα σύνδεση χρησιμοποιείται η υπηρεσία `protocol_main_loop` η οποία και αναλαμβάνει τη μεταφορά δεδομένων από και προς το δίκτυο διασύνδεσης

- **τύπος KSOCKET_REQ_CLOSE:** Χρησιμοποιούνται οι υπηρεσίες `protocol_close` και `protocol_destroy` για την αποδέσμευση των πόρων του δικτύου διασύνδεσης και την καταστροφή των πεδίων της δομής uksocket που εξαρτώνται από το δίκτυο διασύνδεσης. Τέλος, ο χώρος που κατέχει η δομή uksocket απελευθερώνεται και επιστρέφεται στο λειτουργικό σύστημα.

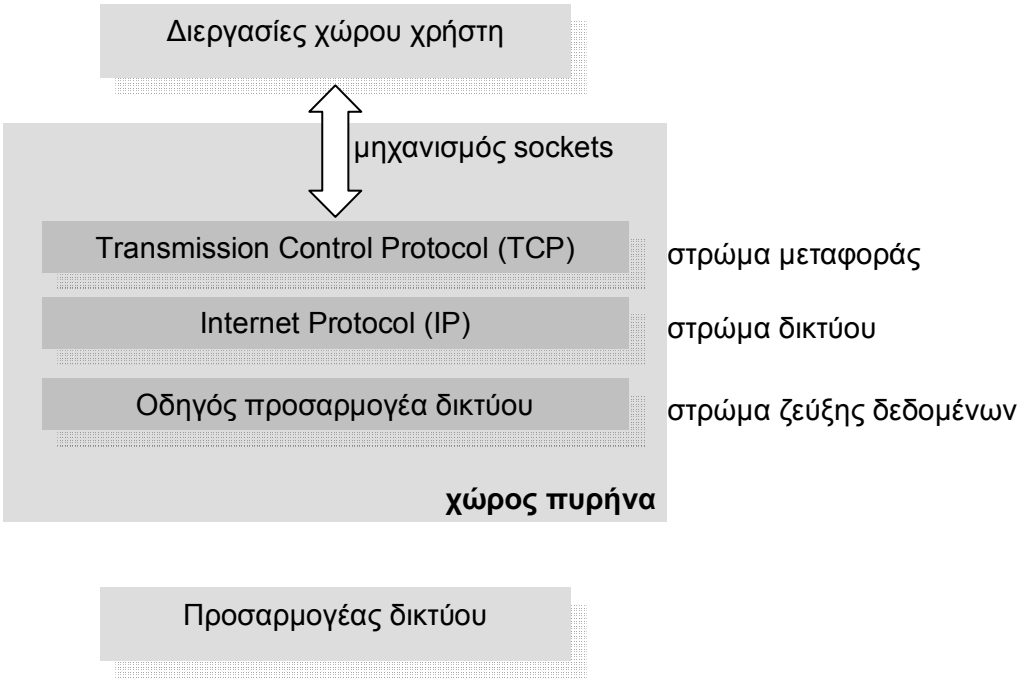
Κεφάλαιο 6

Υποστήριξη του στρώματος επικοινωνίας για δίκτυα TCP/IP

Στο κεφάλαιο αυτό περιγράφεται η επέκταση της διεργασίας χώρου χρήστη των KSockets έτσι ώστε να υποστηρίζονται δίκτυα διασύνδεσης βασισμένα στο πρωτόκολλο TCP/IP (Transmission Control Protocol / Internet Protocol). Η επέκταση του στρώματος επικοινωνίας έτσι ώστε να είναι δυνατή η ανταλλαγή δεδομένων με χρήση του πρωτοκόλλου TCP/IP γίνεται με την υλοποίηση των υπηρεσιών που περιγράφονται στην §5.3.3 (`protocol_start`, `protocol_open`, ...), έτσι ώστε κάθε δίκτυο TCP/IP να μπορεί να προσαρμοστεί στην αφαιρετική όψη του δικτύου διασύνδεσης που χρησιμοποιείται από τα υπόλοιπα τμήματα του στρώματος επικοινωνίας.

6.1 Στοίβα πρωτοκόλλων TCP/IP

Το πρωτόκολλο TCP/IP είναι δομημένο σε διάφορα στρώματα, τα οποία σχηματίζουν τη *στοίβα πρωτοκόλλων* του TCP/IP (TCP/IP protocol stack), όπως αυτή παρουσιάζεται στο ακόλουθο σχήμα:



Σχήμα 6.1 – Στοίβα πρωτοκόλλων TCP/IP

Κάθε στρώμα της στοίβας πρωτοκόλλων χρησιμοποιεί υπηρεσίες που παρέχει το αμέσως χαμηλότερο και παρέχει υπηρεσίες στο αμέσως υψηλότερο στρώμα. Τα τέσσερα στρώματα της στοίβας πρωτοκόλλων, από το χαμηλότερο προς το υψηλότερο, είναι:

- **Στρώμα προσαρμογέα δικτύου (Network interface layer):** Το στρώμα προσαρμογέα δικτύου, ή στρώμα ζεύξης δεδομένων, είναι το χαμηλότερο της στοίβας πρωτοκόλλων. Στο επίπεδο αυτό γίνεται άμεση πρόσβαση στο δίκτυο διασύνδεσης για μεταφορά δεδομένων. Οι λεπτομέρειες της επικοινωνίας (για παράδειγμα, αν παρέχεται αξιοπιστία κατά τη μεταφορά των δεδομένων) δεν αφορούν την εφαρμογή που κάνει χρήση του πρωτοκόλλου TCP/IP για μεταφορά δεδομένων, εφόσον αυτές αποκρύπτονται από τα ανώτερα στρώματα. Στο επίπεδο του προσαρμογέα δικτύου είναι δυνατό να υπάρχουν πολλά διαφορετικά δίκτυα, όπως FastEthernet ή GigabitEthernet, τα οποία δίνουν τη δυνατότητα μεταφοράς *πλαισίων* (Ethernet frames), χωρίς να εγγυώνται την ασφαλή μεταφορά τους, δίκτυα οπτικών ινών FDDI, ένα δίκτυο που βασίζεται στην τεχνολογία ATM (Asynchronous Transfer Mode), κ.ά.

- **Στρώμα δικτύου (Network layer):** Στο στρώμα αυτό βρίσκεται το *πρωτόκολλο IP* (Internet Protocol), το ένα από τα δύο μέρη του πρωτοκόλλου TCP/IP. Το πρωτόκολλο IP παρέχει στα υψηλότερα στρώματα μια αφαιρετική όψη του

πραγματικού δικτύου επικοινωνίας που βρίσκεται στο χαμηλότερο στρώμα. Είναι ένα πρωτόκολλο χωρίς σύνδεση, το οποίο αναλαμβάνει τη μεταφορά πακέτων από και προς υπολογιστικά συστήματα που συμμετέχουν στο δίκτυο επικοινωνίας του χαμηλότερου στρώματος, χωρίς να προϋποθέτει ότι αυτό προσφέρει τη δυνατότητα αξιόπιστης μεταφοράς των δεδομένων, αλλά και χωρίς να εγγυάται το ίδιο την ασφάλεια των δεδομένων που μεταφέρονται. Επιπλέον, το πρωτόκολλο IP δεν πραγματοποιεί ούτε έλεγχο ροής (flow control). Αυτές οι λειτουργίες πρέπει να παρέχονται από τα ανώτερα στρώματα. Το IP αναλαμβάνει μόνο τη δρομολόγηση πακέτων πληροφορίας προς το υπολογιστικό σύστημα για το οποίο προορίζονται. Για να είναι δυνατή η διάκριση ανάμεσα στους κόμβους του δίκτυου, ανατίθεται σε κάθε υπολογιστικό σύστημα που συμμετέχει σε ένα δίκτυο IP ένας μοναδικός αριθμός των 32-bit, ο οποίος ονομάζεται *διεύθυνση IP* (IP address) και χρησιμοποιείται κατά τη δρομολόγηση των πακέτων.

- **Στρώμα μεταφοράς (Transport layer):** Στο στρώμα μεταφοράς βρίσκεται το *πρωτόκολλο TCP* (Transmission Control Protocol). Σκοπός του είναι να προσφέρει τη δυνατότητα εγκατάστασης αξιόπιστων αμφίδρομων συνδέσεων σημείο-προς-σημείο στο ανώτερο στρώμα της στοίβας πρωτοκόλλων. Το TCP χρησιμοποιεί για τη μεταφορά των δεδομένων τις υπηρεσίες του πρωτοκόλλου IP, ωστόσο προσφέρει επιπλέον λειτουργίες ελέγχου λαθών, επανάληψης της μετάδοσης στην περίπτωση που εντοπιστεί σφάλμα και ελέγχου ροής. Οι συνδέσεις TCP παρουσιάζονται στο στρώμα εφαρμογής την εικόνα ενός συνεχούς, αξιόπιστου ρεύματος δεδομένων. Είναι ευθύνη του TCP/IP να χωρίσει τα δεδομένα προς μεταφορά σε πακέτα και να ζητήσει τη δρομολόγησή τους προς το απομακρυσμένο υπολογιστικό σύστημα από το στρώμα δικτύου. Στην περίπτωση που υπάρξουν προβλήματα κατά τη μεταφορά των πακέτων (πακέτα που φτάνουν με λανθασμένη σειρά, αλλοιωμένα, χαμένα, ή ίσως και διπλασιασμένα) το TCP θα αναλάβει να ζητήσει την εκ νέου αποστολή τους, χωρίς αυτή η διαδικασία να γίνει αντιληπτή από το στρώμα εφαρμογής.

- **Στρώμα εφαρμογής (Application layer):** Στο επίπεδο αυτό βρίσκονται οι εφαρμογές που κάνουν χρήση των υπηρεσιών του TCP/IP. Εδώ βρίσκεται η διεργασία χώρου χρήστη του στρώματος επικοινωνίας των KSockets όταν το

στρώμα επικοινωνίας πυρήνα-προς-πυρήνα χρησιμοποιεί τις υπηρεσίες ενός δικτύου διασύνδεσης βασισμένου στο TCP/IP για τη μεταφορά δεδομένων.

6.2 Υποστήριξη TCP/IP στο Linux - Η βιβλιοθήκη BSD Sockets

Το Linux παρέχει πλήρη υποστήριξη του πρωτοκόλλου TCP/IP στις εφαρμογές που εκτελούνται σε χώρο χρήστη. Η στοίβα πρωτοκόλλων του TCP/IP υλοποιείται σε επίπεδο πυρήνα. Στο κατώτερο επίπεδο, χρησιμοποιούνται οι οδηγοί συσκευών δικτύου για την αποστολή και λήψη πακέτων από το δίκτυο διασύνδεσης που χρησιμοποιείται κάθε φορά.

Η πρόσβαση στη στοίβα πρωτοκόλλων TCP/IP από τις εφαρμογές χώρου χρήστη γίνεται μέσω του μηχανισμού των *BSD Sockets*[12]. Τα sockets ορίζουν σημεία αλληλεπίδρασης των εφαρμογών με τη στοίβα πρωτοκόλλων TCP/IP που υλοποιείται στον πυρήνα. Προσφέρονται κατάλληλες κλήσεις συστήματος, μέσω των οποίων η εφαρμογή μπορεί να ζητήσει την εγκατάσταση αξιόπιστης σύνδεσης με κάποιο απομακρυσμένο υπολογιστικό σύστημα, την ανταλλαγή δεδομένων με αυτό και τον τερματισμό της σύνδεσης.

Στο επίπεδο εφαρμογής, τα sockets που αντιστοιχούν σε συνδέσεις TCP/IP εμφανίζονται ως ανοιχτά αρχεία για τη διεργασία που επιθυμεί να επικοινωνήσει χρησιμοποιώντας το TCP/IP. Ένα νέο socket δημιουργείται με χρήση της κλήσης συστήματος `socket`, η οποία επιστρέφει έναν περιγραφητή αρχείου που μπορεί να χρησιμοποιηθεί από τη διεργασία ακριβώς όπως και κάθε άλλο ανοιχτό αρχείο. Όταν έχει εγκατασταθεί μία σύνδεση προς κάποιο απομακρυσμένο υπολογιστικό σύστημα, με χρήση της κλήσης συστήματος `connect`, ή έχει γίνει αποδεκτή μια σύνδεση από κάποιο απομακρυσμένο σύστημα με χρήση της κλήσης `accept`, είναι δυνατή η ανταλλαγή δεδομένων ανάμεσα στα δύο άκρα της σύνδεσης με χρήση των συνηθισμένων κλήσεων συστήματος `read` και `write` στον περιγραφητή αρχείου. Τέλος, η κλήση συστήματος `close` τερματίζει τη σύνδεση.

6.3 Επέκταση των KSockets για χρήση με δίκτυα TCP/IP

Πολλά δίκτυα διασύνδεσης, προσφέρουν κατάλληλους οδηγούς συσκευών δικτύου στον πυρήνα του Linux, έτσι ώστε να μπορούν να συμμετέχουν στο κατώτερο επίπεδο της

[12] Ονομάζονται BSD Sockets διότι η πρώτη υλοποίησή τους ήταν μέρος του BSD Unix έκδοση 4.1c

στοίβας πρωτοκόλλων TCP/IP για τη μεταφορά πακέτων του πρωτοκόλλου IP. Αυτό σημαίνει ότι μπορούν να χρησιμοποιηθούν άμεσα από τις εφαρμογές χρήστη με χρήση του πρωτοκόλλου TCP/IP. Στα δίκτυα διασύνδεσης στα οποία μπορεί μια διεργασία να έχει πρόσβαση με χρήση του πρωτοκόλλου TCP/IP συμπεριλαμβάνονται τα Fast- και Gigabit-Ethernet, αλλά και το δίκτυο διασύνδεσης υψηλών επιδόσεων Myrinet, το οποίο προσφέρει κατάλληλο στρώμα λογισμικού για τον πυρήνα του Linux που επιτρέπει την μετάδοση πακέτων IP πάνω από το GM, το δικό του χαμηλού επιπέδου στρώμα για πέρασμα μηνυμάτων.

Η επέκταση του στρώματος επικοινωνίας KSockets έτσι ώστε να χρησιμοποιεί το πρωτόκολλο TCP/IP για την πρόσβαση στο δίκτυο διασύνδεσης θα προσέφερε τη δυνατότητα για υποστήριξη πλήθους διαφορετικών δικτύων διασύνδεσης χωρίς να απαιτείται καμία αλλαγή στον κώδικά του, ή στον τρόπο χρήσης του από τα διάφορα υποσυστήματα του πυρήνα. Για να είναι δυνατή η χρήση του πρωτοκόλλου TCP/IP από το στρώμα επικοινωνίας είναι αναγκαία η γραφή κατάλληλου τμήματος της διεργασία χώρου χρήστη το οποίο από τη μία πλευρά προσφέρει τις υπηρεσίες του στο τμήμα της διεργασίας χώρου χρήστη που είναι ανεξάρτητο από το δίκτυο διασύνδεσης (υπηρεσίες `protocol_*`) και από την άλλη χρησιμοποιεί το μηχανισμό των BSD Sockets για πρόσβαση στη στοίβα πρωτοκόλλων του TCP/IP.

Συγκρίνοντας τις ιδιότητες των συνδέσεων που εγκαθίστανται με χρήση του πρωτοκόλλου TCP/IP με αυτές που εγκαθίστανται από πυρήνα σε πυρήνα με χρήση των KSockets, συμπεραίνουμε ότι οι συνδέσεις TCP/IP ικανοποιούν ήδη την αφαιρετική εικόνα ενός αξιόπιστου, αμφίδρομου ρεύματος δεδομένων που απαιτείται να παρέχεται από συνδέσεις των KSockets. Έτσι, διευκολύνεται η σχεδίαση του νέου τμήματος της διεργασίας χώρου χρήστη για υποστήριξη του TCP/IP, εφόσον αυτό δεν χρειάζεται να αναλάβει την εξασφάλιση της αξιοπιστίας κατά τη μεταφορά των δεδομένων.

Στις ακόλουθες παραγράφους περιγράφεται συνοπτικά η υλοποίηση των υπηρεσιών `protocol_*`, έτσι ώστε να είναι δυνατή η χρήση δικτύων βασισμένων στο TCP/IP από το στρώμα επικοινωνίας.

6.3.1 Πεδία της δομής uksocket που εξαρτώνται από το δίκτυο διασύνδεσης

Η δομή uksocket περιλαμβάνει ορισμένα επιπλέον πεδία στα οποία αποθηκεύονται διαχειριστικές πληροφορίες του χρησιμοποιούμενου δικτύου διασύνδεσης. Στην περίπτωση του πρωτοκόλλου TCP/IP το μόνο επιπλέον πεδίο που απαιτείται ονομάζεται `sockfd`. Σε αυτό καταγράφεται ο περιγραφητής αρχείου του socket που χρησιμοποιείται για την αλληλεπίδραση με τη στοίβα πρωτοκόλλων TCP/IP.

6.3.2 Υλοποίηση υπηρεσιών protocol_start, protocol_init

Η συνάρτηση `protocol_start` καλείται κατά την αρχικοποίηση του δικτύου διασύνδεσης. Η βιβλιοθήκη των BSD Sockets δεν χρειάζεται ιδιαίτερη διαδικασία αρχικοποίησης πριν από τη χρήση της, οπότε η συνάρτηση `protocol_start` υπάρχει απλώς ώστε να εξασφαλιστεί η πληρότητα της υλοποίησης. Το ίδιο ισχύει και για την υπηρεσία `protocol_init`, η οποία αρχικοποιεί τα πεδία της δομής uksocket που εξαρτώνται από το δίκτυο διασύνδεσης.

6.3.3 Υλοποίηση υπηρεσίας protocol_open

Η συνάρτηση `protocol_open` στην περίπτωση του πρωτοκόλλου TCP/IP κατασκευάζει ένα νέο socket με χρήση της κλήσης συστήματος `socket` και επιστρέφει τον περιγραφητή του στο πεδίο `sockfd` της δομής uksocket.

6.3.4 Υλοποίηση υπηρεσίας protocol_bind

Η συνάρτηση `protocol_bind` χρησιμοποιείται για τη δέσμευση κατάλληλων πόρων του δικτύου διασύνδεσης, έτσι ώστε να μπορεί να χρησιμοποιηθεί μια συγκεκριμένη θύρα επικοινωνίας των KSockets. Το ίδιο το πρωτόκολλο TCP χρησιμοποιεί την έννοια των *θυρών επικοινωνίας TCP* (TCP ports), έτσι ώστε να μπορεί να πραγματοποιεί πολυπλεξία πολλών διαφορετικών συνδέσεων ανάμεσα σε δύο υπολογιστικά συστήματα. Για διευκόλυνση της υλοποίησης, επιλέγουμε να αντιστοιχίζουμε κάθε θύρα επικοινωνίας των KSockets σε συγκεκριμένη θύρα επικοινωνίας του πρωτοκόλλου TCP. Έτσι, η θύρα επικοινωνίας `n` των KSockets, αντιστοιχίζεται στη θύρα `30000 + n`, του TCP. Η υλοποίηση της υπηρεσίας `protocol_bind` χρησιμοποιεί την κλήση συστήματος `bind`, ώστε να δεσμευτεί το αντίστοιχο TCP port.

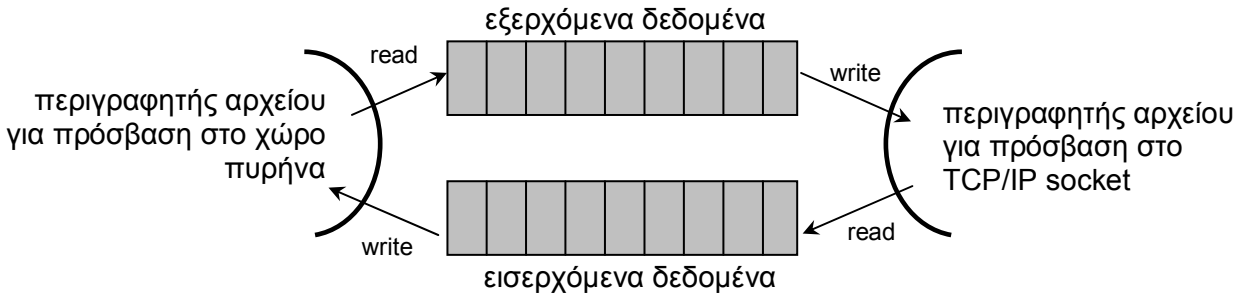
6.3.5 Υλοποίηση υπηρεσιών protocol_connect, protocol_accept

Οι υπηρεσίες `protocol_connect` και `protocol_accept` υλοποιούνται με χρήση των κλήσεων συστήματος `connect` και `accept` που παρέχονται από το μηχανισμό των BSD sockets. Εφόσον κάθε πυρήνας που επικοινωνεί μέσω των KSockets διακρίνεται από τα υπόλοιπα μέσω μοναδικού ονόματος, απαιτείται ένας μηχανισμός αντιστοίχισης των ονομάτων των πυρήνων με τις διευθύνσεις IP των υπολογιστικών συστημάτων στα οποία εκτελούνται. Για το σκοπό αυτό διατηρείται από το διαχειριστή κατάλληλο αρχείο στο οποίο αποθηκεύονται ζεύγη {όνομα πυρήνα, διεύθυνση IP}. Στα δεδομένα του αρχείου αυτού γίνεται αναζήτηση κάθε φορά που ζητείται από τον τοπικό πυρήνα εγκατάσταση ή αποδοχή σύνδεσης, όταν η πρόσβαση στο δίκτυο διασύνδεσης γίνεται μέσω του πρωτοκόλλου TCP/IP.

6.3.6 Υλοποίηση υπηρεσίας protocol_main_loop

Η υπηρεσία `protocol_main_loop` αναλαμβάνει την ανταλλαγή των δεδομένων ανάμεσα στον τοπικό πυρήνα από τη μία πλευρά και τη σύνδεση TCP/IP από την άλλη. Το κανάλι επικοινωνίας με τον τοπικό πυρήνα αντιμετωπίζεται από την διεργασία χώρου χρήστη ως ανοιχτό αρχείο και η πρόσβαση σε αυτό γίνεται μέσω των κλήσεων `read` και `write`. Ομοίως, η πρόσβαση στο δίκτυο TCP/IP γίνεται μέσω κλήσεων συστήματος `read` και `write` στον περιγραφητή αρχείου `sockfd` που αντιστοιχεί στο TCP/IP socket.

Έτσι, η υπηρεσία `protocol_main_loop` καλείται όταν διαβάζει δεδομένα από τον περιγραφητή αρχείου που χρησιμοποιείται για την επικοινωνία με τον τοπικό πυρήνα (πεδίο `fd` της δομής uksocket), να τα γράφει στον περιγραφητή του socket και αντίστροφα. Η διαδικασία αυτή παρουσιάζεται στο παρακάτω διάγραμμα:



Σχήμα 6.2 – Ανταλλαγή δεδομένων ανάμεσα στο χώρο πυρήνα και το TCP/IP socket

Τα δεδομένα που διαβάζονται από κάποιον από τους δύο περιγραφητές αποθηκεύονται προσωρινά σε κάποιον απομονωτή Ε / Ε, έως ότου είναι δυνατή η εγγραφή τους στον άλλο. Οι απομονωτές Ε / Ε που χρησιμοποιούνται είναι κυκλικοί και δεν διαφέρουν σημαντικά από αυτούς που χρησιμοποιούνται για ενδιάμεση αποθήκευση των δεδομένων από το τμήμα του στρώματος επικοινωνίας που εκτελείται σε χώρο πυρήνα (§4.2).

Κεφάλαιο 7

Το δίκτυο διασύνδεσης SCI και η βιβλιοθήκη SISCI

Η υλοποίηση του στρώματος επικοινωνίας KSockets που μελετούμε, υποστηρίζει ως δίκτυο διασύνδεσης για την μεταφορά δεδομένων εκτός από το TCP/IP και το SCI, ένα εξελιγμένο δίκτυο διασύνδεσης υψηλών επιδόσεων. Το SCI διαθέτει χαρακτηριστικά τα οποία είναι ιδιαίτερα επιθυμητά κατά την επιλογή ενός δικτύου διασύνδεσης για συστοιχίες υπολογιστών, όπως μικρός χρόνος αρχικής απόκρισης (latency) και μεγάλος ρυθμός διαμεταγωγής (bandwidth).

Στο κεφάλαιο αυτό γίνεται μια σύντομη παρουσίαση του SCI καθώς και της βιβλιοθήκης SISCI, η οποία προσφέρει στις διεργασίες χρήστη μια ομοιόμορφη προγραμματιστική διαπροσωπεία για τη χρήση των υπηρεσιών του δικτύου διασύνδεσης.

7.1 Το πρότυπο Scalable Coherent Interface

7.1.1 Χαρακτηριστικά του SCI

Το SCI (*Scalable Coherent Interface*) είναι ένα εξελιγμένο πρότυπο διασύνδεσης (ANSI/IEEE 1596-1992 [11]) που αφορά το σχεδιασμό υπολογιστών και δικτύων υψηλών επιδόσεων. Είναι ένα εξαιρετικά ευέλικτο πρότυπο διασύνδεσης, που μπορεί να βρει εφαρμογή σε διάφορους τομείς και να χρησιμοποιηθεί [9]:

- Ως δίκτυο διασύνδεσης υψηλών επιδόσεων για συστοιχίες υπολογιστών

- Για τη διασύνδεση της μνήμης σε συστήματα πολυεπεξεργασίας με συνάφεια κρυφής μνήμης (cache-coherent multiprocessors)

- Ως δίκτυο διασύνδεσης κατανεμημένων συστημάτων Ε / Ε υψηλών επιδόσεων

Οι κόμβοι που συμμετέχουν σε ένα δίκτυο SCI μπορεί να είναι πλήρη υπολογιστικά συστήματα, ακόμη και συστήματα συμμετρικής πολυεπεξεργασίας, ή μόνο ένας επεξεργαστής και η κρυφή μνήμη του, μονάδες μνήμης, συσκευές Ε / Ε, αλλά και γέφυρες προς διαδρόμους ή άλλα δίκτυα διασύνδεσης. Κάθε κόμβος διαθέτει κατάλληλο *προσαρμογέα* για την προσάρτησή του στο δίκτυο SCI.

Στην παρούσα διπλωματική εργασία ασχολούμαστε ιδιαίτερα με τη χρήση του SCI ως δικτύου διασύνδεσης για συστοιχίες υπολογιστών (System Area Network - SAN).

Η ευελιξία του SCI προκύπτει από ορισμένα ιδιαίτερα χαρακτηριστικά, τα οποία αποτέλεσαν βασικούς στόχους κατά τη σχεδίασή του. Τα χαρακτηριστικά αυτά αναλύονται στις επόμενες παραγράφους.

7.1.1.1 Δυνατότητα κλιμάκωσης

Το SCI προσφέρει υπηρεσίες παρόμοιες με αυτές ενός διαδρόμου συστήματος (system bus). Ωστόσο, σε αντίθεση με ένα διάδρομο συστήματος, ο οποίο λόγω της σχεδίασής του ως *μέσο εκπομπής* (broadcast medium) δεν μπορεί να κλιμακωθεί σε μεγάλο αριθμό επεξεργαστών, το SCI σχεδιάστηκε εξ αρχής ως ένα πρότυπο διασύνδεσης με δυνατότητα κλιμάκωσης σε πολύ μεγάλο αριθμό κόμβων, χωρίς ανάλογη μείωση των επιδόσεων. Για το λόγο αυτό χαρακτηρίζεται «scalable».

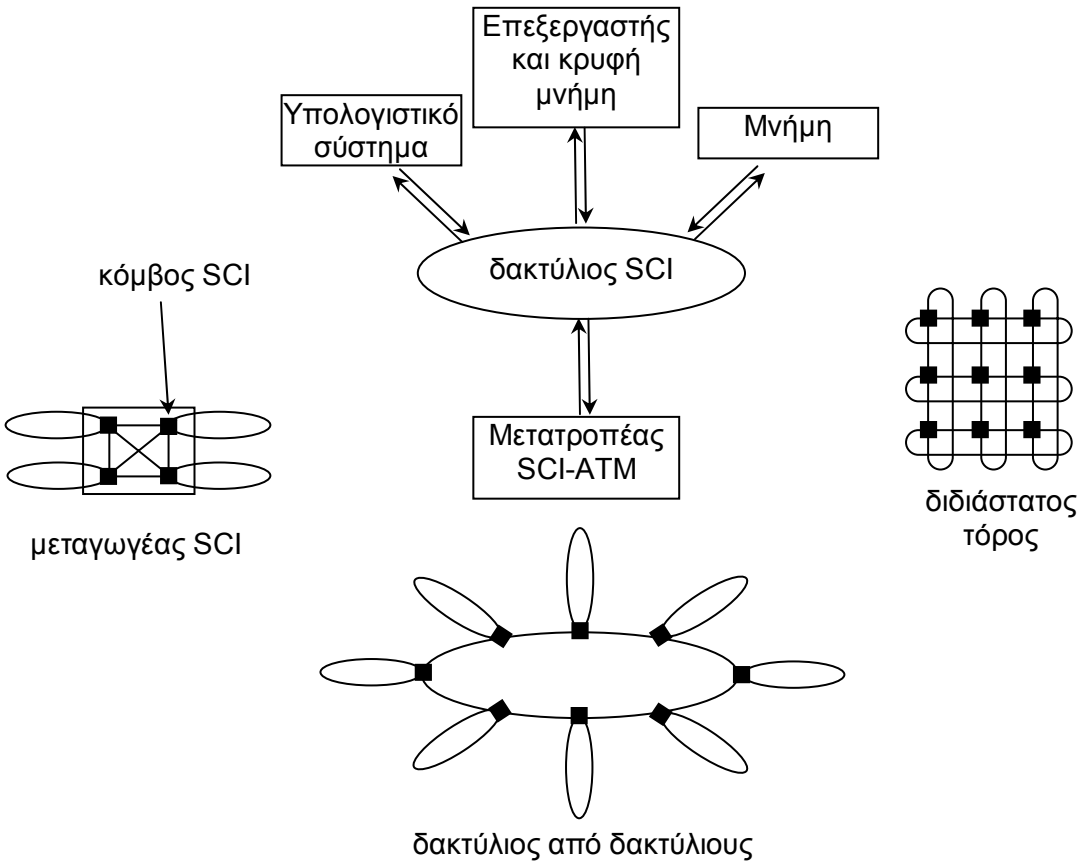
Οι συνηθισμένοι διάδρομοι συστήματος δεν έχουν τη δυνατότητα κλιμάκωσης σε μεγάλο αριθμό επεξεργαστών και αδυνατούν να καλύψουν τις σύγχρονες ανάγκες διασύνδεσης. Δύο είναι οι κυριότερες αιτίες:

- Η φύση του διαδρόμου, ως μέσο εκπομπής, τον καθιστά ακατάλληλο για τη διασύνδεση μεγάλου αριθμού επεξεργαστών, εφόσον μόνο μία δοσοληψία είναι δυνατό να βρίσκεται σε εξέλιξη επάνω στο διάδρομο κάθε φορά. Ο διάδρομος γίνεται η στένωση του συστήματος και περιορίζει δραστικά τον αριθμό των επεξεργαστών που μπορούν να τον χρησιμοποιήσουν αποδοτικά για τη διασύνδεση

με την κεντρική μνήμη, ιδιαίτερα καθώς οι ταχύτητες των μικροεπεξεργαστών αυξάνονται με τη βελτίωση της τεχνολογίας.

- Οι τεχνικές σηματοδοσίας σε διαδρόμους συστήματος φτάνουν στα όριά τους, που καθορίζονται από την ταχύτητα διάδοσης (ταχύτητα του φωτός) καθώς η συχνότητα ρολογιού αυξάνεται. Το μήκος του διαδρόμου είναι ανάγκη να παραμένει πολύ μικρό, καθώς η αύξησή του αυξάνει απαγορευτικά το χρόνο διάδοσης, άρα και το χρόνο απόκρισης κατά τη χρήση του.

Το SCI καταργεί τη χρήση ενός μέσου εκπομπής για τη διασύνδεση των κόμβων του δικτύου και εισάγει ανάμεσά τους μονόδρομους συνδέσμους *σημείο-προς-σημείο*. Περισσότεροι από ένας σύνδεσμοι μπορούν να χρησιμοποιηθούν παράλληλα ανά πάσα στιγμή, έτσι ώστε περισσότεροι από δύο κόμβοι να είναι σε θέση να επικοινωνούν για την ανταλλαγή δεδομένων. Ανάλογα με το μέγεθος του δικτύου διασύνδεσης είναι δυνατές διάφορες τοπολογίες, που περιλαμβάνουν απλούς δακτύλιους (SCI ringlets), δακτύλιους συνδεδεμένους με μεταγωγέα SCI, διδιάστατους ή πολυδιάστατους τόρους.



Σχήμα 7.1 – Τοπολογίες δικτύων SCI

Επειδή ο αριθμός των συνδέσμων αυξάνεται καθώς αυξάνεται ο αριθμός των κόμβων που συμμετέχουν στο δίκτυο διασύνδεσης, αυξάνεται ανάλογα και ο συνολικά προσφερόμενος ρυθμός διαμεταγωγής. Έτσι, αύξηση του αριθμού των κόμβων δεν συνεπάγεται μείωση των επιδόσεων του δικτύου και γίνεται δυνατή η ανάπτυξη του σε πάρα πολύ μεγάλη κλίμακα. Θεωρητικά, το πρότυπο υποστηρίζει την διασύνδεση έως 65536 κόμβων.

7.1.1.2 Υψηλές επιδόσεις

Ένας από τους κυριότερους στόχους του SCI, όπως και κάθε προτύπου διασύνδεσης, είναι η εξασφάλιση σε παράλληλες και κατανεμημένες εφαρμογές όσο το δυνατόν υψηλότερων επιδόσεων, άρα και χαμηλού κόστους επικοινωνίας. Δύο είναι οι κυριότεροι παράγοντες που καθορίζουν το κόστος επικοινωνίας:

- Ο ρυθμός διαμεταγωγής κατά τη μεταφορά δεδομένων (bandwidth)
- Ο χρόνος αρχικής απόκρισης (latency)

Το SCI προβλέπει συνδέσμους υψηλού ρυθμού διαμεταγωγής, της τάξης του 1Gb/sec και προσφέρει ιδιαίτερα ικανοποιητικούς χρόνους αρχικής απόκρισης, μεγέθους λίγων μsec.

7.1.1.3 Υποστήριξη κατανεμημένης μοιραζόμενης μνήμης

Οι πολύ μικροί χρόνοι αρχικής απόκρισης που προσφέρει το SCI οφείλονται σε μεγάλο βαθμό στο μοντέλο επικοινωνίας που ακολουθείται κατά τη χρήση του. Η επικοινωνία πάνω από το SCI βασίζεται στο μοντέλο της *κατανεμημένης μοιραζόμενης μνήμης* (Distributed Shared Memory - DSM). Το πρότυπο του SCI προβλέπει την ύπαρξη ενός ενιαίου, προσβάσιμου από όλους τους κόμβους που συμμετέχουν στο δίκτυο διασύνδεσης, χώρου διευθύνσεων εύρους 64-bit, που ονομάζεται *χώρος διευθύνσεων του SCI*. Ο χώρος διευθύνσεων του SCI κατανέμεται στις μονάδες μνήμης του δικτύου με τρόπο εντελώς αδιαφανή προς το λογισμικό, αλλά ακόμη και τους επεξεργαστές που εκτελούν μεταφορές δεδομένων από και προς το χώρο αυτό.

Η κατανομή της μνήμης γίνεται σε επίπεδο υλικού: Όταν ένας από τους επεξεργαστές επιχειρήσει να προσπελάσει μια διεύθυνση που ανήκει στο χώρο διευθύνσεων του SCI, ο προσαρμογέας του δικτύου εξυπηρετεί την αίτηση αναλαμβάνοντας τη δρομολόγησή της στον ανάλογο κόμβο, όπου βρίσκεται η μνήμη στην οποία αντιστοιχεί η συγκεκριμένη διεύθυνση. Η επικοινωνία με τον κόμβο αυτό γίνεται αποστέλλοντας κατάλληλα πακέτα - αιτήσεις και λαμβάνοντας πακέτα - απαντήσεις επάνω από τους συνδέσμους του δικτύου,

ωστόσο αυτή η λεπτομέρεια αποκρύπτεται από τους επεξεργαστές. Σε λογικό επίπεδο, η μνήμη είναι μοιραζόμενη, ακριβώς όπως θα περίμενε κανείς από ένα σύστημα που χρησιμοποιεί κεντρικό διάδρομο για την πρόσβαση σε μοιραζόμενη μνήμη.

Η υλοποίηση ενός συστήματος κατανεμημένης μοιραζόμενης μνήμης σε επίπεδο υλικού, συνεπάγεται ότι πλέον είναι δυνατή η επικοινωνία ανάμεσα σε κόμβους του δικτύου μέσω της εκτέλεσης απλών εντολών ανάκλησης και αποθήκευσης δεδομένων από και προς την μοιραζόμενη μνήμη. Κατά τη λειτουργία μιας κατανεμημένης εφαρμογής, η πρόσβαση σε απομακρυσμένη μνήμη επιτυγχάνεται με χρήση *ακριβώς* των ίδιων εντολών που χρησιμοποιούνται και για την πρόσβαση σε τοπική μνήμη. Οι εντολές αυτές εκτελούνται απευθείας από την εφαρμογή, στο χώρο χρήστη, χωρίς να είναι αναγκαία η παρέμβαση του λειτουργικού συστήματος για την πραγματοποίηση της επικοινωνίας. Εφόσον δεν ενεργοποιείται κάποια πολυεπίπεδη στοίβα πρωτοκόλλων για την επεξεργασία των δεδομένων προς μεταφορά, ο χρόνος απόκρισης ελαττώνεται σημαντικά και είναι της τάξης των μsec.

Ωστόσο, προκύπτει τώρα το πρόβλημα της ένταξης του δικτύου SCI, άρα και της πρόσβασης στην κατανεμημένη μοιραζόμενη μνήμη, στην αρχιτεκτονική μνήμης ενός υπολογιστικού συστήματος που διαθέτει έναν ή περισσότερους επεξεργαστές. Μια λύση είναι η προσαρμογή του δικτύου διασύνδεσης στον διάδρομο μνήμης ή στον διάδρομο Ε/Ε του υπολογιστικού συστήματος. Η μέθοδος αυτή περιγράφεται στη συνέχεια, όπου παρουσιάζεται η εφαρμογή του SCI ως δίκτυο διασύνδεσης υψηλών επιδόσεων για συστοιχίες υπολογιστών.

7.1.1.4 Συνάφεια κρυφής μνήμης

Τις περισσότερες φορές οι επεξεργαστές που συμμετέχουν στο δίκτυο SCI και έχουν πρόσβαση στην κατανεμημένη μοιραζόμενη μνήμη χρησιμοποιούν *κρυφές μνήμες* (cache memories) για τη μείωση του μέσου χρόνου πρόσβασης στην κεντρική μνήμη. Η χρήση των κρυφών μνημών εισάγει το πρόβλημα της *συνάφειας μνήμης* (cache coherency problem): Κατά την ταυτόχρονη πρόσβαση σε μοιραζόμενα δεδομένα, είναι δυνατό αν δεν ληφθούν κατάλληλα μέτρα τα αντίγραφα των δεδομένων που βρίσκονται στις κρυφές μνήμες να μην είναι πλέον έγκυρα και έτσι να μην εξασφαλίζεται η ακεραιότητα των μοιραζόμενων δεδομένων.

Τα πρωτόκολλα που χρησιμοποιούνται σε συστήματα συμμετρικής πολυεπεξεργασίας, όπου οι επεξεργαστές συνδέονται μέσω διαδρόμου με τη μοιραζόμενη μνήμη δεν είναι δυνατό να χρησιμοποιηθούν στην περίπτωση του SCI, όπου η επικοινωνία με τη μοιραζόμενη μνήμη δεν γίνεται πλέον μέσω ενός μέσου εκπομπής, όπως είναι ο διάδρομος, αλλά κατανεμημένα, μέσω ενός συνόλου συνδέσμων σημείο-προς-σημείο. Έτσι, το πρότυπο ορίζει κατανεμημένα πρωτόκολλα συνάφειας γρήγορης μνήμης, τα οποία βασίζονται στην προσέγγιση του κατανεμημένου καταλόγου [9]. Τα πρωτόκολλα αυτά είναι σχεδιασμένα ώστε να υλοποιούνται σε επίπεδο υλικού, στον προσαρμογέα του δικτύου SCI. Στην ύπαρξη πρωτοκόλλων συνάφειας κρυφής μνήμης, οφείλει το SCI το χαρακτηρισμό του ως «coherent».

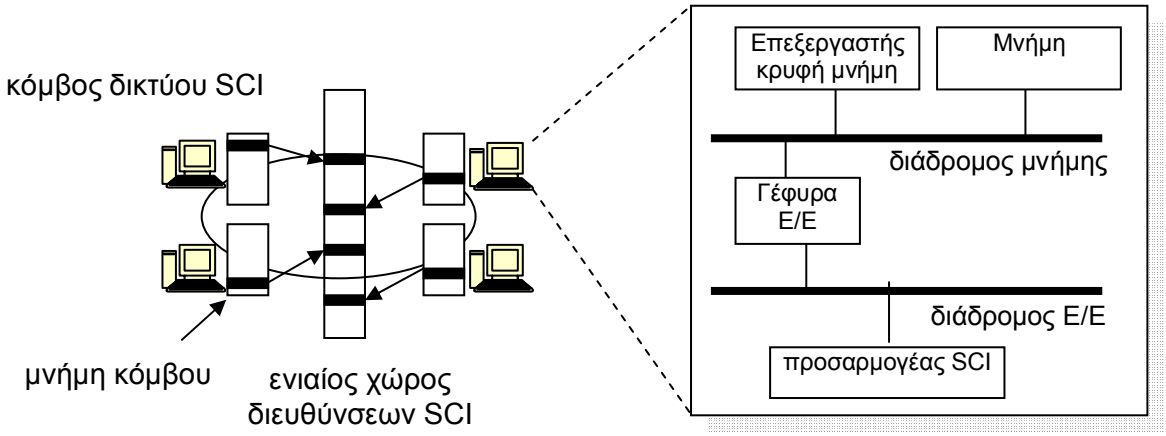
Η συμμόρφωση με τα πρωτόκολλα συνάφειας είναι ιδιαίτερα δύσκολη, λόγω της μεγάλης πολυπλοκότητάς τους, ωστόσο δεν αποτελεί προϋπόθεση για να ακολουθεί μια υλοποίηση το πρότυπο του SCI. Η συνάφεια γρήγορης μνήμης είναι προαιρετικό χαρακτηριστικό και δεν επιβάλλεται από το πρότυπο. Μάλιστα, όπως θα δούμε στη συνέχεια, στην περίπτωση κατά την οποία το SCI χρησιμοποιείται ως δίκτυο διασύνδεσης για συστοιχίες υπολογιστών είναι τις περισσότερες φορές αδύνατη, αφού ο προσαρμογέας του δικτύου SCI είναι εγκατεστημένος στον διάδρομο E/E του υπολογιστικού συστήματος και δεν μπορεί να συμμετάσχει στο πρωτόκολλο συνάφειας που ακολουθείται στο διάδρομο μνήμης, ανάμεσα στους επεξεργαστές και την κεντρική μνήμη.

7.1.2 Το SCI ως δίκτυο διασύνδεσης για συστοιχίες υπολογιστών

Ένα δίκτυο διασύνδεσης βασισμένο στο πρότυπο SCI παρέχει τη δυνατότητα για αποδοτική επικοινωνία ανάμεσα στα υπολογιστικά συστήματα μιας συστοιχίας υπολογιστών. Η χρήση του SCI ως δικτύου διασύνδεσης γίνεται δυνατή μέσω της προσαρμογής του στον διάδρομο E/E των συστημάτων που συμμετέχουν στη συστοιχία. Την προσαρμογή αναλαμβάνει μια κάρτα επέκτασης, η οποία από τη μία πλευρά συνδέεται με το δίκτυο SCI, και από την άλλη επικοινωνεί με το διάδρομο E/E (π.χ. με το διάδρομο PCI στην περίπτωση της αρχιτεκτονικής Intel x86).

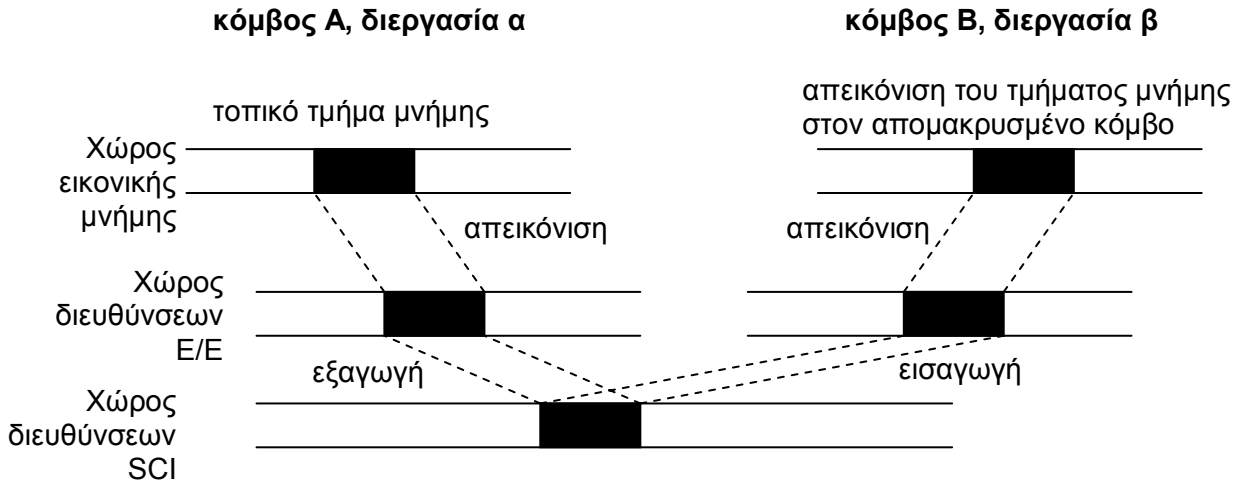
Το δίκτυο διασύνδεσης, χρησιμοποιώντας τον ενιαίο χώρο διευθύνσεων του SCI, παρέχει ένα σύστημα κατανεμημένης μοιραζόμενης μνήμης που υλοποιείται σε επίπεδο υλικού. Η μοιραζόμενη μνήμη σχηματίζεται από τμήματα της κεντρικής μνήμης των υπολογιστικών συστημάτων που συμμετέχουν στη συστοιχία. Έτσι, μια συστοιχία υπολογιστών βασισμένη

στο SCI είναι στενότερα συνδεδεμένη απ' ότι αν είχε χρησιμοποιηθεί ένα τοπικό δίκτυο για τη διασύνδεση των συστημάτων και εμφανίζει χαρακτηριστικά μιας παράλληλης μηχανής *NUMA* (Non-uniform Memory Access), εφόσον κάθε επεξεργαστής μπορεί να έχει πρόσβαση με μικρότερο κόστος επικοινωνίας σε ορισμένα τμήματα της μοιραζόμενης μνήμης, σε σχέση με τα υπόλοιπα.



Σχήμα 7.2 – Το SCI ως δίκτυο διασύνδεσης για συστοιχία υπολογιστών

Η επικοινωνία ανάμεσα σε δύο υπολογιστικά συστήματα γίνεται χρησιμοποιώντας το μοντέλο της μοιραζόμενης μνήμης. Κάποιο υπολογιστικό σύστημα, έστω Α, μοιράζεται ένα τμήμα της κεντρικής, φυσικής του μνήμης με τους υπόλοιπους κόμβους, *εξάγοντάς* το στον χώρο διευθύνσεων του SCI. Ένα άλλο σύστημα, έστω Β, *εισάγει* το συγκεκριμένο τμήμα της κατανεμημένης μοιραζόμενης μνήμης στον χώρο διευθύνσεων Ε/Ε του. Η απεικόνιση του χώρου διευθύνσεων του SCI στον χώρο φυσικών διευθύνσεων Ε/Ε του υπολογιστικού συστήματος γίνεται σε επίπεδο υλικού, από τον προσαρμογέα SCI, μέσω *πινάκων μετάφρασης διευθύνσεων* (Address Translation Tables - ATT). Δύο διεργασίες α και β, που εκτελούνται στους κόμβους Α και Β αντίστοιχα, μπορούν πλέον να επικοινωνήσουν μέσω του δικτύου διασύνδεσης ζητώντας από το λειτουργικό σύστημα την απεικόνιση των συγκεκριμένων τμημάτων φυσικής μνήμης Ε/Ε στο χώρο εικονικής μνήμης τους. Η απεικόνιση αναλαμβάνεται από τη μονάδα διαχείρισης μνήμης (Memory Management Unit - MMU) της ΚΜΕ.



Σχήμα 7.3 – Χώροι διευθύνσεων και απεικόνιση τμημάτων μνήμης στο SCI

Όταν η απεικόνιση των τμημάτων μνήμης έχει ολοκληρωθεί, η διεργασία α (αντίστροφα, η διεργασία β) μπορεί να μεταβάλλει τις τιμές που αποθηκεύονται στο συγκεκριμένο τμήμα της κεντρικής μνήμης του Β (του Α) μέσω απλών εντολών ανάκλησης / αποθήκευσης. Οι δοσοληψίες Ε/Ε (I/O transactions) που προκαλούνται κατά την πρόσβαση των επεξεργαστών σε απομακρυσμένες θέσεις μνήμης, μετατρέπονται από τους προσαρμογείς SCI σε δοσοληψίες επάνω στο δίκτυο SCI, για την ανάκληση η αποθήκευση τιμών.

Ο προσαρμογέας SCI επικοινωνεί μόνο με το διάδρομο Ε/Ε και δεν μπορεί να συμμετάσχει στο πρωτόκολλο συνάφειας κρυφής μνήμης που ακολουθείται στο διάδρομο μνήμης. Για το λόγο αυτό, δεδομένα που ανήκουν σε απομακρυσμένη μνήμη δεν αποθηκεύονται ποτέ σε τοπική κρυφή μνήμη, αλλά κάθε πρόσβαση σε αυτά ικανοποιείται πάντοτε με δοσοληψίες του δικτύου διασύνδεσης.

7.2 Η βιβλιοθήκη χώρου χρήστη SISCI

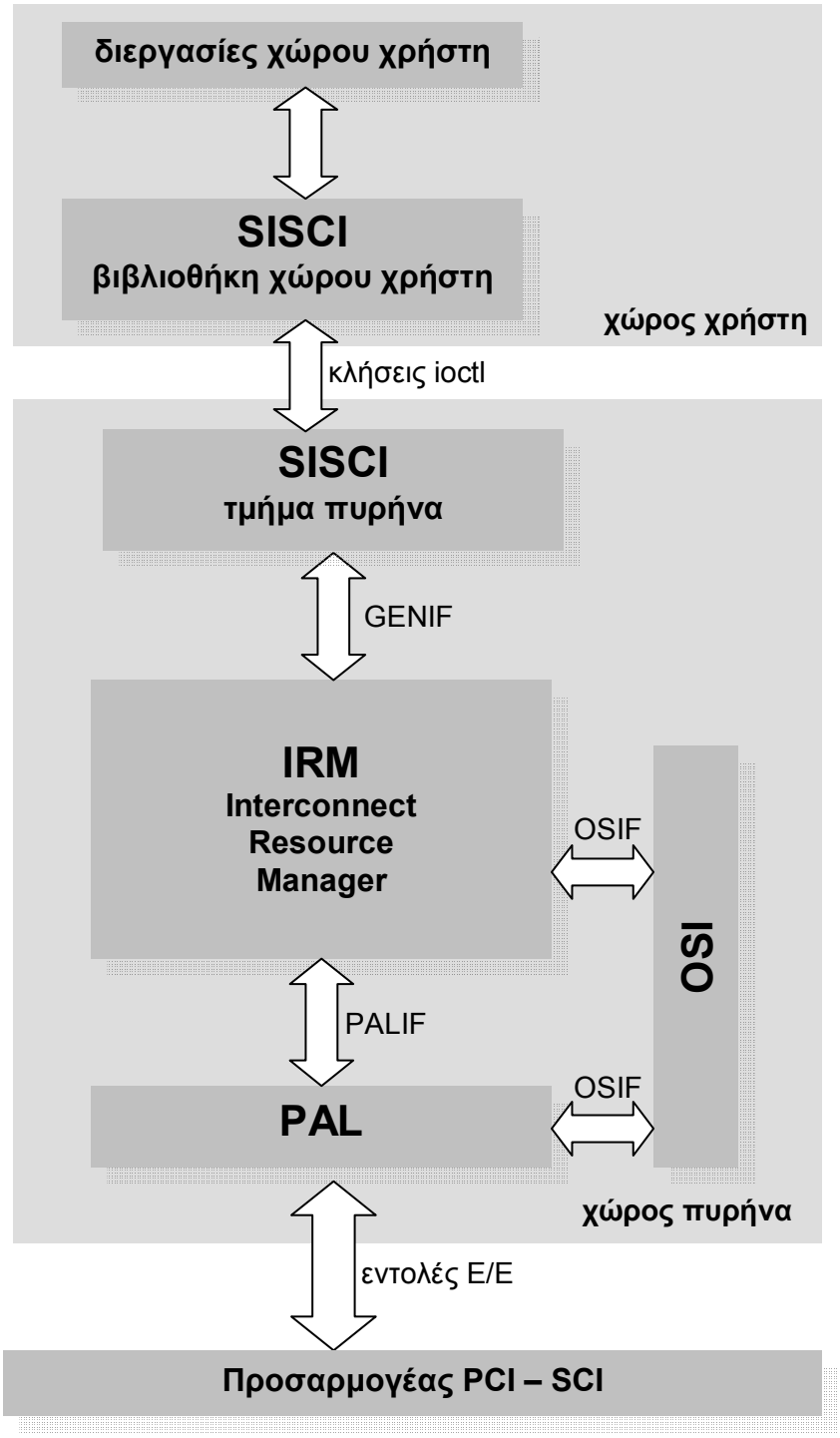
7.2.1 Υποστήριξη του SCI κάτω από το Linux

Μία εμπορική υλοποίηση προσαρμογέα του δικτύου SCI στον διάδρομο PCI, ο οποίος χρησιμοποιείται στην αρχιτεκτονική Intel x86 είναι η κάρτα επέκτασης PCI-SCI της Dolphin Interconnect Solutions[13]. Η Dolphin έχει αναπτύξει διάφορες εκδόσεις του

[13] Προς το παρόν, η υλοποίηση αυτή παραμένει και η μοναδική εμπορική υλοποίηση γέφυρας PCI-SCI

προσαρμογέα αυτού, που υποστηρίζουν σύνδεση σε διάδρομο PCI εύρους 32 ή 64-bit, με συχνότητα ρολογιού 33MHz και 66MHz.

Ο προσαρμογέας PCI-SCI συνοδεύεται από κατάλληλο σύστημα λογισμικού, το οποίο υλοποιείται ως συνδυασμός από αρθρώματα του πυρήνα του Linux και βιβλιοθήκες χώρου χρήστη, έτσι ώστε να είναι δυνατή η χρήση του δικτύου SCI ως δίκτυο διασύνδεσης για συστοιχίες βασισμένες στο Linux.



Σχήμα 7.4 – Δομή του λογισμικού διαχείρισης για τον προσαρμογέα SCI

Τα δομικά στοιχεία του λογισμικού για τη διαχείριση των πόρων του δικτύου SCI, όπως αυτά παρουσιάζονται στο παραπάνω σχήμα, είναι:

- **Το στρώμα PAL (Physical Abstraction Layer - Στρώμα Αφαίρεσης Υλικού):** Το στρώμα αυτό βρίσκεται στο κατώτερο επίπεδο της στοίβας λογισμικού. Παρέχει συγκεκριμένη προγραμματιστική διαπροσωπεία στο ανώτερο στρώμα, το IRM, έτσι ώστε να επιτρέπεται ο προγραμματισμός του υλικού ανεξάρτητα από τις λεπτομέρειες της υλοποίησης για το συγκεκριμένο τύπο προσαρμογέα PCI-SCI που χρησιμοποιείται κατά την πρόσβαση στο δίκτυο διασύνδεσης.

- **Το στρώμα IRM (Interconnect Resource Manager - Διαχειριστής Πόρων Δικτύου Διασύνδεσης):** Το στρώμα IRM χρησιμοποιεί τις υπηρεσίες που παρέχονται από το στρώμα PAL, μέσω της διαπροσωπείας PALIF (PAL InterFace) για τον προγραμματισμό του προσαρμογέα PCI-SCI και τη διαχείριση των πόρων του. Οι υπηρεσίες του IRM προσφέρονται σε οδηγούς που εκτελούνται στο χώρο πυρήνα μέσω της προγραμματιστικής διαπροσωπείας GENIF (GENeral InterFace). Η χρήση των υπηρεσιών του IRM μέσω του GENIF επιτρέπει πρόσβαση χαμηλού επιπέδου στις υπηρεσίες του δικτύου SCI, με τρόπο εντελώς ανεξάρτητο του υλικό. Επιπλέον, ο IRM επιτρέπει την ύπαρξη πολλών διαφορετικών χρηστών της διαπροσωπείας GENIF στο ίδιο υπολογιστικό σύστημα και συντονίζει την ταυτόχρονη πρόσβασή τους στον προσαρμογέα SCI.

- **Το στρώμα OSI:** Το στρώμα OSI περικλείει κώδικα που εξαρτάται από το λειτουργικό σύστημα. Προσφέρει στον IRM τη διαπροσωπεία OSIF (Operating System IFace), η οποία αποκρύπτει πολλές από τις λεπτομέρειες της αλληλεπίδρασης με το λειτουργικό σύστημα. Η ύπαρξή του στρώματος OSI συμβάλλει στη μεταφερσιμότητα του κώδικα του IRM, εφόσον το μεγαλύτερο τμήμα κώδικα που εξαρτάται από το λειτουργικό σύστημα εμπεριέχεται στο στρώμα OSIF και είναι προσβάσιμο μέσω καθορισμένης διαπροσωπείας. Ο κώδικας του IRM έχει μεταφερθεί σε αρκετά λειτουργικά συστήματα, όπως τα Windows NT, το Linux, το Solaris και άλλα. Για κάθε ένα από αυτά γράφτηκε το ανάλογο στρώμα OSI.

Τα υπόλοιπα δύο στρώματα αποτελούν τμήματα του λογισμικού SISCI και αναλύονται διεξοδικά στη συνέχεια.

7.2.2 Γενική περιγραφή της βιβλιοθήκης SISCI

Το πρότυπο SISCI (*Software Infrastructure for SCI*) αναπτύσσεται στα πλαίσια του προγράμματος ESPRIT [10] και σκοπεύει στον καθορισμό συγκεκριμένης προγραμματιστικής διαπροσωπείας για την πρόσβαση στο δίκτυο διασύνδεσης SCI από διεργασίες χώρου χρήστη. Η Dolphin έχει αναπτύξει μια υλοποίησή του, η οποία είναι διαθέσιμη για διάφορα λειτουργικά συστήματα, στα οποία συμπεριλαμβάνεται και το Linux.

Σκοπός του στρώματος SISCI είναι η παροχή υπηρεσιών στις διεργασίες χρήστη, ώστε να είναι δυνατή η αποδοτική χρήση του SCI ως δικτύου διασύνδεσης σε συστοιχίες υπολογιστών και να απλοποιηθεί η δημιουργία κατανεμημένων εφαρμογών οι οποίες βασίζονται απευθείας στο μοντέλο κατανεμημένης μοιραζόμενης μνήμης που προσφέρει το SCI. Ορισμένες από τις λειτουργίες που αφορά η διαπροσωπεία που ορίζεται από το στρώμα SISCI είναι:

- Η κατασκευή τοπικών τμημάτων μνήμης, τα οποία είναι προσβάσιμα από απομακρυσμένους κόμβους μέσω του δικτύου SCI
- Η απεικόνιση τοπικών και απομακρυσμένων τμημάτων μνήμης στον χώρο εικονικής μνήμης της διεργασίας χρήστη
- Η δημιουργία τοπικών διακοπών και η ενεργοποίηση απομακρυσμένων διακοπών του SCI

Οι λειτουργίες αυτές παρουσιάζονται αναλυτικότερα στη συνέχεια.

Η υλοποίηση του στρώματος SISCI, όπως φαίνεται και στο προηγούμενο σχήμα αποτελείται από δύο τμήματα: Το πρώτο από αυτά εκτελείται σε χώρο πυρήνα και χρησιμοποιεί τις υπηρεσίες του IRM μέσω της διαπροσωπείας GENIF. Το δεύτερο εκτελείται στο χώρο χρήστη, χρησιμοποιεί τις υπηρεσίες του πρώτου μέσω του μηχανισμού των κλήσεων συστήματος του λειτουργικού συστήματος (κυρίως της κλήσης `ioctl`) και συνδέεται με τον κώδικα των διεργασιών χώρου χρήστη ως βιβλιοθήκη. Το τμήμα αυτό αναλαμβάνει την υλοποίηση της διαπροσωπείας προς τις διεργασίες χρήστη.

7.2.3 Προσφερόμενη προγραμματιστική διαπροσωπεία

Η βιβλιοθήκη χώρου χρήστη του SISCI ορίζει ένα σύνολο αντικειμένων τα οποία αναπαριστούν πόρους του δικτύου διασύνδεσης και μια σειρά από συναρτήσεις για την κατασκευή, τη διαχείριση και την καταστροφή τους. Οι συναρτήσεις αυτές μπορούν να κληθούν άμεσα από προγράμματα σε γλώσσα C. Στη συνέχεια παρουσιάζονται συνοπτικά οι σημαντικότερες από τις συναρτήσεις του SISCI μαζί με τα αντικείμενα τα οποία διαχειρίζονται.

7.2.3.1 Διαχείριση εικονικών συσκευών

Το SISCI ορίζει το αντικείμενο `sci_desc`, το οποίο αναπαριστά μια εικονική συσκευή SCI, δηλαδή ένα κανάλι επικοινωνίας ανάμεσα στη διεργασία χρήστη και τον οδηγό του SISCI σε επίπεδο πυρήνα. Μια διεργασία χρήστη μπορεί να ανοίξει μία ή και περισσότερες εικονικές συσκευές, δημιουργώντας νέα κανάλια επικοινωνίας με τον οδηγό χώρου πυρήνα, χρησιμοποιώντας την κλήση `SCIOpen`. Όταν η χρήση κάποιας εικονικής συσκευής έχει ολοκληρωθεί, αυτή μπορεί να καταστραφεί με χρήση της κλήσης `SCIClose`.

7.2.3.2 Διαχείριση τμημάτων μνήμης

Ορίζονται τα αντικείμενα `sci_local_segment` και `sci_remote_segment` τα οποία αναπαριστούν ένα τοπικό και ένα απομακρυσμένο τμήμα μνήμης αντίστοιχα. Κάθε τοπικό τμήμα μνήμης σχετίζεται με ένα αναγνωριστικό τμήματος (μη αρνητικό ακέραιο), ο οποίος χρησιμοποιείται για τη διάκρισή του από τα υπόλοιπα τοπικά τμήματα και οφείλει να είναι μοναδικό ανάμεσα στα αναγνωριστικά των τμημάτων του τοπικού κόμβου.

Ένα τοπικό τμήμα μνήμης δημιουργείται με χρήση της κλήσης `SCICreateSegment`. Μετά τη δημιουργία του μπορεί να προετοιμαστεί (κλήση `SCIPrepareSegment`) ώστε να γίνει διαθέσιμο στους υπόλοιπους κόμβους του δικτύου SCI με χρήση της κλήσης `SCISetSegmentAvailable`. Όταν ένα τοπικό τμήμα μνήμης είναι διαθέσιμο, είναι δυνατή η σύνδεση με αυτό και η μεταβολή του από απομακρυσμένους κόμβους μέσω εντολών ανάκλησης / αποθήκευσης. Όταν δεν πρόκειται να γίνουν δεκτές νέες συνδέσεις προς αυτό από απομακρυσμένους κόμβους του δικτύου, τότε μπορεί να σταματήσει να είναι διαθέσιμο, μέσω της κλήσης `SCISetSegmentUnavailable`. Τέλος, όταν δεν πρόκειται να χρησιμοποιηθεί πλέον, μπορεί να καταστραφεί με χρήση της κλήσης `SCIRemoveSegment`.

Για τη σύνδεση με ένα απομακρυσμένο τμήμα μοιραζόμενης μνήμης προσφέρεται η κλήση `SCIConnectSegment`. Για να είναι επιτυχημένη η σύνδεση πρέπει προηγουμένως το τμήμα προορισμού να έχει προετοιμαστεί και να έχει γίνει διαθέσιμο μέσω των `SCIPrepareSegment` και `SCISetSegmentAvailable`. Μετά την ολοκλήρωση της πρόσβαση σε απομακρυσμένο τμήμα μνήμης, η σύνδεση προς αυτό μπορεί να διακοπεί με χρήση της κλήσης `SCIDisconnectSegment`.

7.2.3.3 Διαχείριση απεικονίσεων στον εικονικό χώρο μνήμης

Για να είναι δυνατή η ανάκτηση και η μεταβολή των δεδομένων που περιέχονται σε κάποιο τοπικό ή απομακρυσμένο τμήμα μνήμης, προσβάσιμο μέσω του δικτύου διασύνδεσης, πρέπει προηγουμένως αυτό να έχει απεικονιστεί σε σελίδες του χώρου εικονικής μνήμης της διεργασίας. Μια απεικόνιση τοπικού ή απομακρυσμένου τμήματος μνήμης στο χώρο εικονικής μνήμης της διεργασίας αναπαρίσταται από τον τύπο `sci_map`. Στην περίπτωση τοπικού τμήματος η απεικόνιση δημιουργείται με χρήση της κλήσης `SCIMapLocalSegment`, ενώ στην περίπτωση απομακρυσμένου τμήματος, με χρήση της `SCIMapRemoteSegment`. Η απεικόνιση απομακρυσμένου τμήματος στο χώρο εικονικής μνήμης επιτρέπεται μόνο όταν υπάρχει ήδη εγκατεστημένη σύνδεση προς αυτό (κλήση `SCIConnectSegment`).

7.2.3.4 Διαχείριση τοπικών και απομακρυσμένων διακοπών SCI

Το δίκτυο SCI προσφέρει το μηχανισμό των *διακοπών SCI* ως μέσο για την ειδοποίηση ενός απομακρυσμένου κόμβου για ορισμένο γεγονός, π.χ. ότι υπάρχουν δεδομένα προς αυτόν σε κάποιο τμήμα μοιραζόμενης μνήμης. Το SISCI προσφέρει τα αντικείμενα `sci_local_interrupt` και `sci_remote_interrupt` τα οποία αναπαριστούν μία τοπική και μία απομακρυσμένη διακοπή SCI, αντίστοιχα.

Μια τοπική διακοπή δημιουργείται με χρήση της συνάρτησης `SCICreateInterrupt`. Η εφαρμογή μπορεί να περιμένει για ενεργοποίηση της διακοπής από κάποιο απομακρυσμένο κόμβο με χρήση της κλήσης `WaitForInterrupt`, ή να δηλώσει στη βιβλιοθήκη SISCI μια συνάρτηση η οποία επιθυμεί να εκτελείται *ασύγχρονα* (asynchronous interrupt callback), κάθε φορά που ενεργοποιείται η συγκεκριμένη τοπική διακοπή. Η υποστήριξη ασύγχρονων συναρτήσεων χειρισμού διακοπών διατίθεται από τη βιβλιοθήκη SISCI μόνο για την περίπτωση πολυνηματικών διεργασιών, οπότε η λειτουργία της συνάρτησης χειρισμού διακοπής γίνεται σε ένα διαφορετικό νήμα εκτέλεσης από την υπόλοιπη εφαρμογή.

Για να είναι δυνατή η ενεργοποίηση μιας διακοπής σε απομακρυσμένο κόμβο του δικτύου SCI πρέπει προηγουμένως να έχει εγκατασταθεί σύνδεση προς αυτή, με χρήση της κλήσης `SCIConnectInterrupt`. Όταν η σύνδεση έχει ολοκληρωθεί, μπορεί να προκληθεί απομακρυσμένη διακοπή με χρήση της κλήσης `SCITriggerInterrupt`. Η αποσύνδεση από απομακρυσμένη διακοπή, όταν δεν υπάρχει πλέον ανάγκη ενεργοποίησής της, γίνεται με χρήση της κλήσης `SCIDisconnectInterrupt`.

Κεφάλαιο 8

Υποστήριξη του στρώματος επικοινωνίας για το δίκτυο SCI

Στο κεφάλαιο αυτό παρουσιάζεται η επέκταση του στρώματος επικοινωνίας KSockets ώστε να υποστηρίζεται το δίκτυο διασύνδεσης SCI για την ανταλλαγή δεδομένων ανάμεσα σε πυρήνες Linux που επικοινωνούν. Η επέκταση του στρώματος επικοινωνίας γίνεται μέσω της υλοποίησης κατάλληλου τμήματος της διεργασίας χώρου χρήστη. Το τμήμα αυτό υλοποιεί την προγραμματιστική διαπροσωπεία που απαιτείται για την υποστήριξη ενός νέου δικτύου διασύνδεσης από το στρώμα επικοινωνίας (§5.3.3). Η υλοποίηση των υπηρεσιών αυτών βασίζεται στη χρήση της βιβλιοθήκης χώρου χρήστη SISCI, όπως περιγράφηκε στο προηγούμενο κεφάλαιο, για την εξαγωγή τοπικών τμημάτων μνήμης και την απομακρυσμένη πρόσβαση σε μοιραζόμενα τμήματα μνήμης που ανήκουν σε διαφορετικούς κόμβους της συστοιχίας.

8.1 Γενικά για το μοντέλο επικοινωνίας

Το τμήμα της διεργασίας χρήστη που σχεδιάζουμε οφείλει να παρέχει μια αφαιρετική όψη του δικτύου διασύνδεσης που επιτρέπει την εγκατάσταση αξιόπιστων συνδέσεων σημείο-προς-σημείο και την αντιμετώπισή τους ως ένα συνεχές ρεύμα δεδομένων. Ωστόσο, το πραγματικό προγραμματιστικό μοντέλο του SCI διαφέρει αρκετά από αυτή την αφαιρετική όψη.

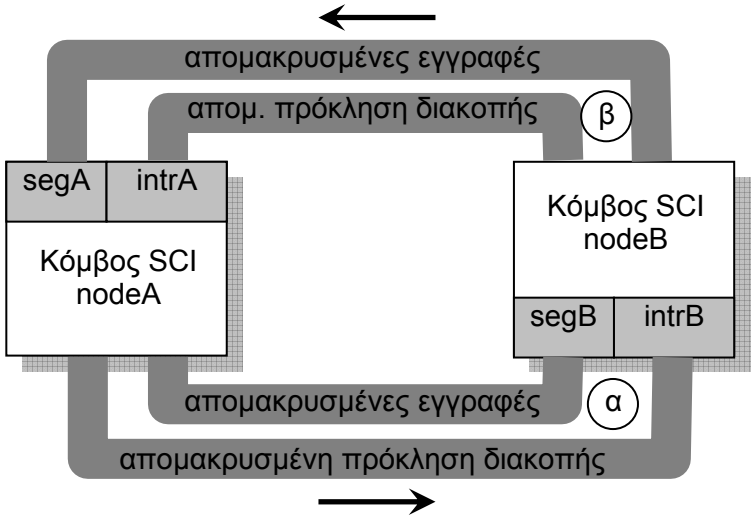
Οι συνδέσεις που παρέχει το SCI δεν είναι αξιόπιστα σειριακά ρεύματα δεδομένων, αλλά επιτρέπουν γρήγορη και αξιόπιστη πρόσβαση σε απομακρυσμένα τμήματα μνήμης. Επιπλέον, παρέχονται μηχανισμοί διακοπής, μέσω των οποίων είναι δυνατή η ειδοποίηση απομακρυσμένων κόμβων για γεγονότα της επικοινωνίας, με μικρή επιβάρυνση όσον αφορά το χρόνο καθυστέρησης. Οι δύο αυτοί μηχανισμοί, ο μηχανισμός της απομακρυσμένης πρόσβασης σε μνήμη και ο μηχανισμός των διακοπών μπορούν να συνδυαστούν για τη δημιουργία της ζητούμενης αφαιρετικής όψης.

Για το σκοπό αυτό, αντιστοιχίζεται σε κάθε ιδεατή τοπική θύρα επικοινωνίας των KSockets ένα ζεύγος τοπικού τμήματος μνήμης και τοπικής διακοπής SCI. Η αντιστοίχιση ανάμεσα σε αριθμούς θυρών επικοινωνίας KSockets και ακέραιους- αναγνωριστικά τοπικών τμημάτων μνήμης και διακοπών είναι αυθαίρετη. Στην παρούσα υλοποίηση, η θύρα επικοινωνίας των KSockets n αντιστοιχίζεται στο τοπικό τμήμα μνήμης με αναγνωριστικό `4*n` και στην διακοπή `4*n+1`.

Η δέσμευση μιας τοπικής θύρας επικοινωνίας συνεπάγεται τη δημιουργία ενός τοπικού τμήματος μνήμης και την αντιστοίχισή του στον εικονικό χώρο μνήμης της διεργασίας χώρου χρήστη. Η αποδοχή μιας σύνδεσης υλοποιείται με εξαγωγή του τμήματος στον ενιαίο χώρο διευθύνσεων του SCI και αναμονή για σύνδεση σε αυτό από κάποιον απομακρυσμένο κόμβο του δικτύου. Οι συνδέσεις με χρήση των KSockets είναι αμφίδρομες, οπότε όταν γίνει μια σύνδεση σε τοπικό τμήμα μνήμης από κάποιο απομακρυσμένο κόμβο είναι αναγκαία η πραγματοποίηση της αντίστροφης σύνδεσης: Από τον τοπικό κόμβο προς το απομακρυσμένο τμήμα μνήμης. Το αναγνωριστικό του απομακρυσμένου τμήματος μνήμης στο οποίο γίνεται η σύνδεση εξαρτάται από την ιδεατή θύρα επικοινωνίας των KSockets που χρησιμοποιείται από την πλευρά του απομακρυσμένου συστήματος.

Παράλληλα με τις συνδέσεις σε τμήματα μνήμης, οφείλουν να ολοκληρωθούν με επιτυχία και οι συνδέσεις στις αντίστοιχες διακοπές του SCI. Μόνο τότε, όταν έχουν εγκατασταθεί όλες οι απαραίτητες συνδέσεις πάνω από το δίκτυο SCI και προς τις δύο κατευθύνσεις μπορεί να θεωρηθεί επιτυχημένη η εγκατάσταση της σύνδεσης πάνω από το στρώμα επικοινωνίας.

Η υλοποίηση μιας σύνδεσης των KSockets με χρήση του δικτύου SCI και τα βήματα που ακολουθούνται κατά την εγκατάστασή της παρουσιάζονται στο παρακάτω σχήμα:



(α) Ο κόμβος nodeA εισάγει το μοιραζόμενο τμήμα μνήμης segB και συνδέεται στην απομακρυσμένη διακοπή intrB

(β) Ο κόμβος nodeB εισάγει το μοιραζόμενο τμήμα μνήμης segA και συνδέεται στην απομακρυσμένη διακοπή intrA, οπότε ξεκινά αμφίδρομη επικοινωνία

Σχήμα 8.1 – Εγκατάσταση αμφίδρομης σύνδεσης με χρήση του SCI

Σε περίπτωση που οποιοδήποτε τμήμα της διαδικασίας αποτύχει για κάποιο λόγο, η διεργασία χώρου χρήστη οφείλει να αναιρέσει τις ενέργειες που έχουν γίνει έως αυτό το σημείο, καλώντας τις κατάλληλες συναρτήσεις καταστροφής αντικειμένων που παρέχονται από τη βιβλιοθήκη SISCI. Έτσι, είναι αναγκαία η παρακολούθηση της εξέλιξης της διαδικασίας και η καταγραφή των μέχρι τότε επιτυχημένων ενεργειών, ώστε να είναι δυνατή η πλήρης αναίρεσή τους στην περίπτωση που συμβεί σφάλμα κατά την εγκατάσταση της σύνδεσης.

Η λύση μιας σύνδεσης γίνεται ακολουθώντας την αντίστροφη σειρά. Όταν ένας από τους δύο κόμβους του δικτύου SCI που βρίσκονται στα άκρα της σύνδεσης αντιληφθεί ότι ο απομακρυσμένος κόμβος αποσυνδέθηκε από τους τοπικούς πόρους του που γίνονται διαθέσιμοι στο δίκτυο (την τοπική διακοπή και το τοπικό τμήμα μνήμης), οφείλει να αποσυνδεθεί και αυτός με τη σειρά του από τους απομακρυσμένους πόρους, ώστε η σύνδεση να τερματιστεί.

8.2 Επιπλέον πεδία της δομής uksocket

Η δομή uksocket χρησιμοποιείται από τη διεργασία χώρου χρήστη για την αποθήκευση της κατάστασης μιας εγκατεστημένης σύνδεσης από πυρήνα σε πυρήνα. Ορισμένα από τα πεδία που περιλαμβάνει εξαρτώνται από το δίκτυο διασύνδεσης που χρησιμοποιείται κάθε φορά (§5.3.2). Στην περίπτωση κατά την οποία χρησιμοποιείται το δίκτυο διασύνδεσης SCI για τη μεταφορά των δεδομένων, η δομή uksocket περιλαμβάνει επιπλέον πεδία τα οποία ορίζουν την τοπική και την απομακρυσμένη θύρα επικοινωνίας. Έτσι ορίζονται πεδία τύπου `sci_local_interrupt` και `sci_remote_interrupt` τα οποία περιγράφουν τον τοπικό και τον απομακρυσμένο πόρο διακοπής SCI, πεδία τύπου `sci_local_segment` και `sci_remote_segment` τα οποία περιγράφουν το τοπικό και το απομακρυσμένο τμήμα μνήμης που χρησιμοποιούνται για την ανταλλαγή των δεδομένων καθώς και δύο πεδία `sci_map`, τα οποία αντιστοιχούν στις απεικονίσεις του τοπικού και του απομακρυσμένου τμήματος μνήμης στο χώρο εικονικής μνήμης της διεργασίας χρήστη.

8.3 Υλοποίηση προγραμματιστικής διαπροσωπείας

Στις ακόλουθες παραγράφους περιγράφεται η υλοποίηση των σημαντικότερων υπηρεσιών προς το τμήμα της διεργασίας χώρου χρήστη που δεν εξαρτάται από το δίκτυο διασύνδεσης. Οι υπηρεσίες που υλοποιούνται ορίζονται στην §5.3.3.

8.3.1 Υλοποίηση υπηρεσίας protocol_start

Η υπηρεσία `protocol_start` καλείται κατά την έναρξη της λειτουργίας του στρώματος επικοινωνίας, ώστε να αρχικοποιηθεί το δίκτυο διασύνδεσης που χρησιμοποιείται. Στην περίπτωση του δικτύου SCI, η `protocol_start` καλεί τη διαδικασία `SCIInitialize` της βιβλιοθήκης SISCI, η οποία πρέπει οπωσδήποτε να χρησιμοποιηθεί πριν την πραγματοποίηση οποιασδήποτε άλλης κλήσης προς τη βιβλιοθήκη.

8.3.2 Υλοποίηση υπηρεσίας protocol_open

Η υπηρεσία `protocol_open` καλείται έπειτα από τη δημιουργία μιας νέας δομής uksocket και ενός νέου καναλιού επικοινωνίας με το χώρο πυρήνα, έτσι ώστε να δεσμευτούν οι απαραίτητοι πόροι για την εγκατάσταση, αργότερα, μιας νέας σύνδεσης πάνω από το δίκτυο SCI. Η υλοποίησή της χρησιμοποιεί την κλήση `SCIOpen` της βιβλιοθήκης SISCI για

την δημιουργία μιας νέας εικονικής συσκευής, μέσω της οποίας αργότερα θα δημιουργηθούν και θα εξαχθούν στους υπόλοιπους κόμβους του δικτύου οι πόροι του SCI που χρησιμοποιούνται κατά τη μεταφορά δεδομένων.

8.3.3 Υλοποίηση υπηρεσίας protocol_bind

Η υπηρεσία `protocol_bind` χρησιμοποιείται για τη δέσμευση μιας θύρας επικοινωνίας των KSockets. Η υλοποίησή της για το δίκτυο SCI χρησιμοποιεί τις κλήσεις `SCICreateInterrupt`, `SCICreateSegment`, `SCIPrepareSegment` και `SCIMapLocalSegment` για την δημιουργία μιας τοπικής διακοπής του SCI και τη δημιουργία, προετοιμασία για μοίρασμα σε απομακρυσμένους κόμβους και απεικόνιση στο χώρο εικονικής μνήμης ενός νέου τοπικού τμήματος. Στην περίπτωση που η θύρα επικοινωνίας που ζητείται χρησιμοποιείται ήδη, η δημιουργία των τοπικών πόρων αποτυγχάνει και επιστρέφεται κατάλληλο μήνυμα λάθους στο υποσύστημα του πυρήνα που ζήτησε τη δέσμευση της συγκεκριμένης θύρας επικοινωνίας.

8.3.4 Υλοποίηση υπηρεσίας protocol_connect

Η υπηρεσία `protocol_connect` αναλαμβάνει την εγκατάσταση μιας νέας σύνδεσης με κάποιο απομακρυσμένο σύστημα. Οι πυρήνες που συμμετέχουν στην επικοινωνία με χρήση των KSockets αναγνωρίζονται και διακρίνονται με χρήση μοναδικού ονόματος, ανεξάρτητου από το χρησιμοποιούμενο δίκτυο διασύνδεσης. Όμως, στην περίπτωση του δικτύου SCI, οι κόμβοι του δικτύου διακρίνονται μέσω ενός μοναδικού ακεραίου, που ονομάζεται *αναγνωριστικό κόμβου* (node identifier). Έτσι, είναι απαραίτητη η ύπαρξη ενός αρχείου το οποίο περιέχει ζεύγη της μορφής { όνομα πυρήνα, αναγνωριστικό κόμβου στο δίκτυο SCI }, το οποίο διατηρείται από το διαχειριστή της συστοιχίας υπολογιστών. Στο αρχείο αυτό ανατρέχει η υλοποίηση της protocol_connect, έτσι ώστε να μεταφράσει το όνομα του πυρήνα προς τον οποίο ζητείται εγκατάσταση σύνδεσης σε αναγνωριστικό κόμβου για χρήση στο δίκτυο SCI.

Για την εγκατάσταση μιας σύνδεσης προς απομακρυσμένο πυρήνα, ακολουθούνται τα εξής βήματα:

Βήμα 1: Το όνομα του απομακρυσμένου πυρήνα αντιστοιχίζεται, μέσω του αρχείου που αναφέραμε προηγούμενα, σε αναγνωριστικό κόμβου SCI

Βήμα 2: Με χρήση της συνάρτησης `SCIConnectSegment` εγκαθίσταται σύνδεση προς συγκεκριμένο μοιραζόμενο τμήμα μνήμης του απομακρυσμένου κόμβου. Το αναγνωριστικό του απομακρυσμένου τμήματος προκύπτει από τον αριθμό της απομακρυσμένης θύρας επικοινωνίας των KSockets που αφορά το αίτημα σύνδεσης. Επιπλέον, γίνεται σύνδεση με χρήση της `SCIConnectInterrupt` στην αντίστοιχη απομακρυσμένη διακοπή SCI. Η σύνδεση με τους απομακρυσμένους πόρους είναι επιτυχής μόνο αν αυτοί έχουν γίνει εκ των προτέρων διαθέσιμοι, αν δηλαδή έχει εκτελεστεί η κλήση `protocol_accept` στο απομακρυσμένο σύστημα.

Βήμα 3: Το αναγνωριστικό του τοπικού κόμβου SCI και ο τοπικός αριθμός θύρας επικοινωνίας αποθηκεύονται σε συγκεκριμένη θέση του απομακρυσμένου τμήματος μνήμης. Στη συνέχεια προκαλείται απομακρυσμένη διακοπή, έτσι ώστε να ενημερωθεί ο απομακρυσμένος κόμβος για τη νέα σύνδεση.

Βήμα 4: Στο σημείο αυτό έχει εγκατασταθεί η μία από τις δύο κατευθύνσεις της σύνδεσης ανάμεσα στα δύο υπολογιστικά συστήματα. Ο απομακρυσμένος κόμβος, όταν ενεργοποιηθεί η διακοπή οφείλει να πραγματοποιήσει τα βήματα 1 - 3 προς την κατεύθυνση του τοπικού κόμβου. Αν η διαδικασία αυτή δεν ολοκληρωθεί σε συγκεκριμένο χρονικό διάστημα, τότε η εγκατάσταση της σύνδεσης θεωρείται ότι απέτυχε και ο τοπικός κόμβος προχωρά σε αποσύνδεσή του από τους απομακρυσμένους πόρους. Σε αντίθετη περίπτωση, η σύνδεση έχει εγκατασταθεί με επιτυχία και είναι διαθέσιμη για την επικοινωνία ανάμεσα στον τοπικό και τον απομακρυσμένο πυρήνα.

8.3.5 Υλοποίηση υπηρεσίας protocol_accept

Η υπηρεσία `protocol_accept` αναλαμβάνει την αποδοχή μιας σύνδεσης σε τοπική θύρα επικοινωνίας των KSockets από απομακρυσμένο πυρήνα. Τα βήματα που ακολουθούνται είναι ανάλογα με αυτά της `protocol_connect`.

Αρχικά οι τοπικοί πόροι που αντιστοιχούν στη θύρα επικοινωνίας γίνονται διαθέσιμοι στους υπόλοιπους κόμβους της συστοιχίας και στη συνέχεια η διαδικασία αναμένει την εγκατάσταση σύνδεσης προς αυτούς και την πρόκληση τοπικής διακοπής. Όταν η προκληθεί τοπική διακοπή, γίνεται ανάγνωση των στοιχείων του απομακρυσμένου κόμβου από το τοπικό τμήμα μνήμης, πραγματοποιείται η σύνδεση προς τους απομακρυσμένους πόρους και ξεκινά η ανταλλαγή δεδομένων.

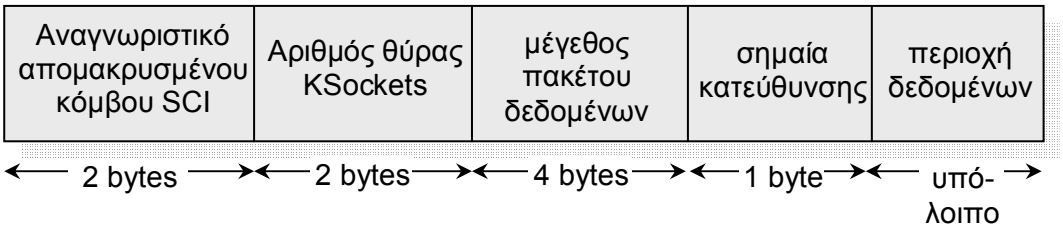
8.3.6 Υλοποίηση υπηρεσίας protocol_main_loop

Η υπηρεσία `protocol_main_loop` αναλαμβάνει την ανταλλαγή των δεδομένων ανάμεσα στον τοπικό πυρήνα και το δίκτυο διασύνδεσης. Το κανάλι επικοινωνίας με τον τοπικό πυρήνα είναι προσβάσιμο από τη διεργασία χώρου χρήστη ως ανοιχτό αρχείο. Τα δεδομένα που προέρχονται από τον πυρήνα και πρόκειται να μεταφερθούν μέσω του δικτύου διασύνδεσης παρέχονται στο χώρο χρήστη μέσω της κλήσης συστήματος `read` σε κατάλληλο περιγραφητή αρχείου (§5.3.1), όμοια, δεδομένα που φτάνουν από το δίκτυο διασύνδεσης προωθούνται στο χώρο πυρήνα μέσω της κλήσης συστήματος `write`.

Η επικοινωνία με τον απομακρυσμένο κόμβο του SCI πάνω από το δίκτυο διασύνδεσης, γίνεται με την αποθήκευση των εξερχόμενων δεδομένων στο απομακρυσμένο τμήμα μνήμης και την ανάγνωση των εισερχόμενων δεδομένων από το τοπικό τμήμα μνήμης. Αυτός ο τρόπος επικοινωνίας ευνοεί τις *απομακρυσμένες* εγγραφές και τις *τοπικές* αναγνώσεις. Αντίστροφα, θα μπορούσαμε να διαβάζουμε τα εισερχόμενα δεδομένα από το απομακρυσμένο τμήμα μνήμης και να γράφουμε τα εξερχόμενα δεδομένα στο τοπικό τμήμα μνήμης, ευνοώντας τις απομακρυσμένες αναγνώσεις και τις τοπικές εγγραφές.

Ο τρόπος επικοινωνίας που τελικά επιλέγεται είναι αυτός που ευνοεί τις απομακρυσμένες εγγραφές και αποφεύγει τις απομακρυσμένες αναγνώσεις δεδομένων, λόγω του μεγαλύτερου κόστους επικοινωνίας κατά την πραγματοποίηση απομακρυσμένων αναγνώσεων. Το μεγαλύτερο κόστος επικοινωνίας μπορεί να εξηγηθεί ως εξής: Μια δοσοληψία απομακρυσμένης ανάγνωσης υλοποιείται σε επίπεδο υλικού είναι διαδικασία δύο βημάτων: Πρώτα αποστέλλεται κατάλληλη αίτηση στον απομακρυσμένο κόμβο από τον τοπικό προσαρμογέα SCI και στη συνέχεια αναμένονται οι τιμές των δεδομένων που ζητήθηκαν. Κατά τη διάρκεια αυτής της διαδικασίας ο τοπικός επεξεργαστής που εκτέλεσε την εντολή ανάκλησης δεδομένων από τη μνήμη αναγκάζεται να καθυστερεί την εκτέλεση των επόμενων εντολών (pipeline stall). Αντίθετα, κατά την εκτέλεση απομακρυσμένων εγγραφών, η μεγαλύτερη καθυστέρηση που συνεπάγεται η πρόσβαση σε απομακρυσμένη μνήμη, σε σύγκριση με την πρόσβαση στην τοπική μνήμη δεν γίνεται αντιληπτή από τον τοπικό επεξεργαστή, εφόσον κατάλληλου μεγέθους απομονωτές που βρίσκονται στον προσαρμογέα SCI επιτρέπουν στον επεξεργαστή να συνεχίσει άμεσα με την εκτέλεση των επόμενων εντολών και αποκρύπτουν την καθυστέρηση κατά την πρόσβαση σε απομακρυσμένη μνήμη.

Τα δεδομένα μεταφέρονται ως ένα συνεχές ρεύμα από και προς τον τοπικό πυρήνα, ωστόσο η ανάγνωσή τους από το τοπικό τμήμα μνήμης και η εγγραφή τους στο απομακρυσμένο τμήμα μνήμης γίνεται σε ομάδες, το μέγιστο μέγεθος των οποίων καθορίζεται από το μήκος των μοιραζόμενων τμημάτων μνήμης. Η μορφή που έχει κάθε τμήμα μοιραζόμενης μνήμης κατά τη μεταφορά δεδομένων είναι:



Σχήμα 8.2 – Μορφή μοιραζόμενου τμήματος μνήμης κατά την επικοινωνία μέσω SCI

Στα πρώτα 2 bytes του τοπικού τμήματος μνήμης αποθηκεύεται το 16-bit αναγνωριστικό του απομακρυσμένου κόμβου SCI, ενώ στα επόμενα 2 bytes η απομακρυσμένη θύρα επικοινωνίας των KSockets. Τα δύο αυτά στοιχεία καθορίζουν τα αναγνωριστικά του απομακρυσμένου μοιραζόμενου τμήματος μνήμης και της απομακρυσμένης διακοπής SCI.

Στα επόμενα 4 bytes καθορίζεται το μήκος των δεδομένων που είναι αποθηκευμένα στο χώρο αποθήκευσης δεδομένων του μοιραζόμενου τμήματος μνήμης. Τα δεδομένα που βρίσκονται εκεί προωθούνται προς τον τοπικό πυρήνα με χρήση της κλήσης `write`. Τέλος, η *σημαία κατεύθυνσης* καταλαμβάνει ένα byte και χρησιμοποιείται για το συγχρονισμό των δύο πλευρών κατά την πρόσβαση σε μοιραζόμενο τμήμα μνήμης: Όταν η σημαία έχει την τιμή `READ_FLAG` γίνεται ανάγνωση των δεδομένων από το συγκεκριμένο τμήμα μνήμης και αποστολή τους με χρήση της `write` στο τμήμα πυρήνα. Αντίθετα, όταν η σημαία έχει την τιμή `WRITE_FLAG`, γίνεται εγγραφή νέων εισερχόμενων δεδομένων από τον απομακρυσμένο κόμβο SCI.

Έτσι, ο βασικός βρόχος ανταλλαγής δεδομένων κατά την επικοινωνία με χρήση του SCI ως δικτύου διασύνδεσης αποτελείται από τα παρακάτω βήματα:

Βήμα 1: Στο βήμα αυτό η διαδικασία `protocol_main_loop` περιμένει έως ότου ικανοποιηθεί μία ή περισσότερες από τις παρακάτω συνθήκες: **(α)** η τιμή της σημαίας κατεύθυνσης στο τοπικό τμήμα μνήμης είναι `READ_FLAG` και νέα δεδομένα μπορούν να προωθηθούν προς το χώρο πυρήνα μέσω της κλήσης `write`, **(β)** η τιμή της σημαίας κατεύθυνσης στο απομακρυσμένο τμήμα μνήμης είναι `WRITE_FLAG` και να νέα δεδομένα

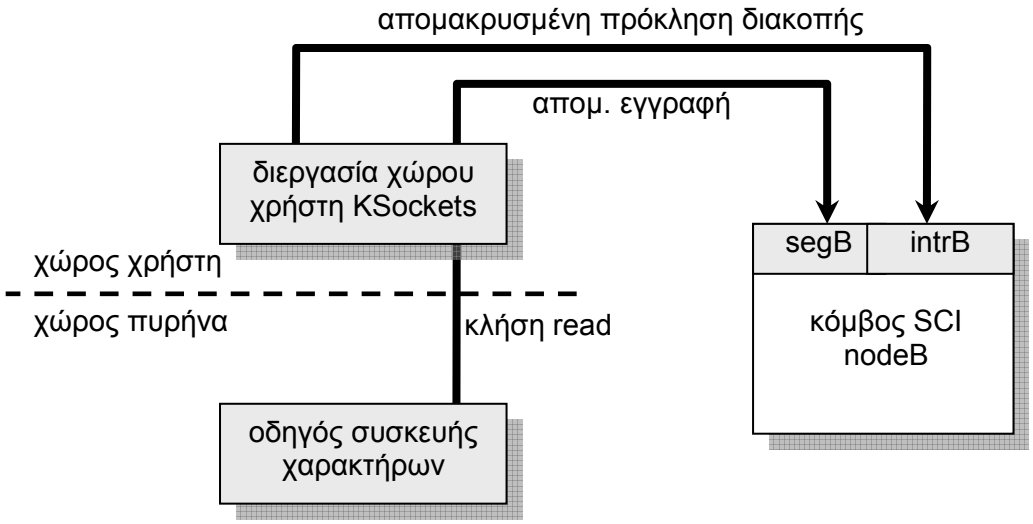
είναι διαθέσιμα από το χώρο πυρήνα μέσω της κλήσης `read`, **(γ)** προκληθεί τοπική διακοπή SCI

Βήμα 2: Αν νέα δεδομένα από το χώρο πυρήνα είναι διαθέσιμα, τότε αυτά αντιγράφονται απευθείας στο απομακρυσμένο τμήμα μνήμης με χρήση της κλήσης συστήματος `read`, στη συνέχεια μεταβάλλεται η τιμή της απομακρυσμένης σημαίας κατεύθυνσης και προκαλείται απομακρυσμένη διακοπή.

Βήμα 3: Αν είναι δυνατή η προώθηση δεδομένων προς τον τοπικό πυρήνα και υπάρχουν δεδομένα στο τοπικό τμήμα μνήμης, αυτά προωθούνται με χρήση της `write`. Όταν ολοκληρωθεί η αποστολή των δεδομένων στον τοπικό πυρήνα και δεν απομένουν εισερχόμενα δεδομένα στο τοπικό τμήμα μνήμης, μεταβάλλεται η τιμή της τοπικής σημαίας κατεύθυνσης και προκαλείται απομακρυσμένη διακοπή.

Η παραπάνω διαδικασία εφαρμόζεται στο ακόλουθο παράδειγμα, όπου ο πυρήνας του υπολογιστικού συστήματος Α (τοπικός κόμβος) αποστέλλει μια ομάδα από bytes στον πυρήνα του υπολογιστικού συστήματος Β (απομακρυσμένος κόμβος):

Αρχικά η συνάρτηση `protocol_main_loop` στον κόμβο Α αντιλαμβάνεται, διαβάζοντας τη σημαία κατεύθυνσης στο απομακρυσμένο τμήμα μνήμης (έχει την τιμή `WRITE_FLAG`) ότι μπορεί να αποστείλει δεδομένα. Η αποστολή των δεδομένων γίνεται μέσω της κλήσης `read`, με την οποία ζητά από τον πυρήνα την προώθηση δεδομένων από το χώρο πυρήνα σε συγκεκριμένη διεύθυνση του χώρου εικονική μνήμης της. Εφόσον όμως εκεί έχει ήδη απεικονιστεί το απομακρυσμένο τμήμα μνήμης, η αντιγραφή των δεδομένων από το χώρο πυρήνα στο χώρο εικονικής μνήμης της διεργασίας μεταφράζεται από τον προσαρμογέα SCI σε απομακρυσμένες εγγραφές. Όταν η κλήση read ολοκληρωθεί, μεταβάλλεται η απομακρυσμένη σημαία κατεύθυνσης σε `READ_FLAG` και προκαλείται απομακρυσμένη διακοπή, ώστε να ενημερωθεί η απομακρυσμένη διεργασία και να προωθήσει τα δεδομένα που μόλις έλαβε προς στον απομακρυσμένο χώρο πυρήνα.



Σχήμα 8.3 – Προώθηση δεδομένων του τοπικού πυρήνα σε απομακρυσμένο κόμβο SCI

Κεφάλαιο 9

Έλεγχος καλής λειτουργίας και μετρήσεις επίδοσης

9.1 Έλεγχος καλής λειτουργίας του στρώματος επικοινωνίας

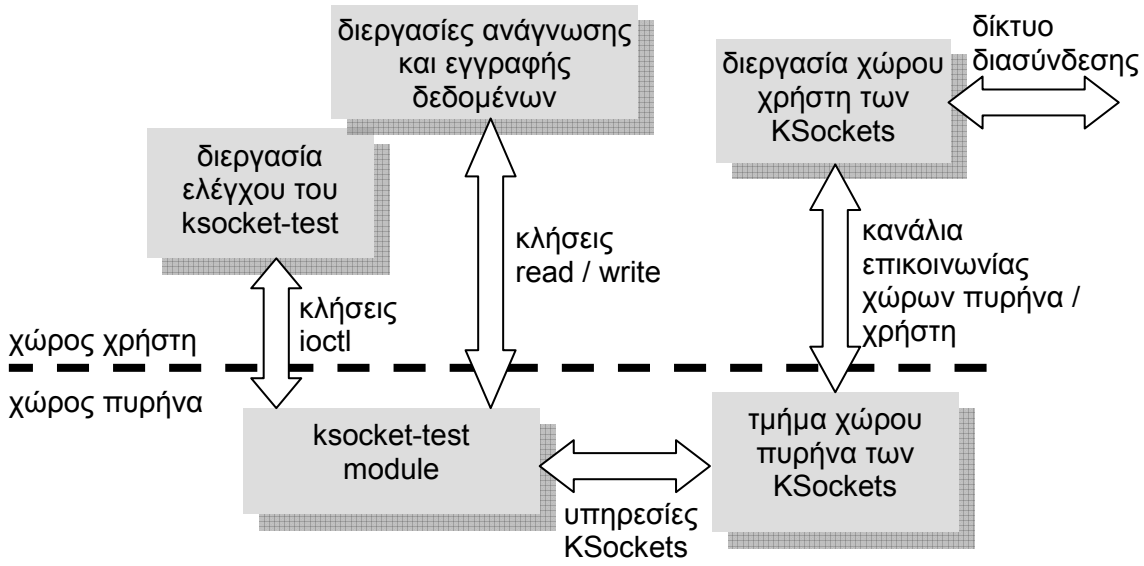
Έχοντας επεκτείνει τη διεργασία χώρου χρήστη έτσι ώστε να υποστηρίζονται διάφορα δίκτυα διασύνδεσης για τη μεταφορά δεδομένων, η υλοποίηση του στρώματος επικοινωνίας έχει ολοκληρωθεί και μπορούμε να το χρησιμοποιήσουμε για την εγκατάσταση αξιόπιστων συνδέσεων ανάμεσα σε πυρήνες Linux. Η υλοποίηση που παρουσιάστηκε μπορεί να βασιστεί για την πραγματική ανταλλαγή δεδομένων είτε σε δίκτυο SCI είτε σε δίκτυα τα οποία ενσωματώνονται στη στοίβα πρωτοκόλλων του TCP/IP.

Για τη λειτουργία του στρώματος επικοινωνίας KSockets απαιτείται η γραφή επιπλέον κώδικα χώρου πυρήνα του Linux, ο οποίος κάνει χρήση των υπηρεσιών επικοινωνίας που προσφέρονται. Το νέο τμήμα του Linux περιμένουμε να αποτελεί επέκταση των δυνατοτήτων του πυρήνα, με σκοπό την αποδοτικότερη εκτέλεσή του σε περιβάλλον συστοιχίας υπολογιστών και την καλύτερη υποστήριξη κατανεμημένων εφαρμογών. Το στρώμα επικοινωνίας που σχεδιάστηκε και υλοποιήθηκε έρχεται να καλύψει τις ανάγκες αυτού του νέου τμήματος για επικοινωνία από το επίπεδο του πυρήνα με ομοιόμορφο τρόπο, ανεξάρτητο του χρησιμοποιούμενου δικτύου διασύνδεσης.

Ένα τέτοιο υποσύστημα του πυρήνα του Linux, το οποίο υλοποιήθηκε και χρησιμοποιήθηκε στη φάση ανάπτυξης του στρώματος επικοινωνίας για τον έλεγχο της

σωστής λειτουργίας του και τη διεξαγωγή μετρήσεων για τους βασικούς δείκτες επίδοσης είναι ο οδηγός συσκευής χαρακτήρων ksocket-test. Το υποσύστημα ksocket-test προσφέρει στις διεργασίες χρήστη τη δυνατότητα δημιουργίας δικτυακών σωληνώσεων (network pipes), προσβάσιμων μέσω ειδικών αρχείων στο σύστημα αρχείων (π.χ. της μορφής `/dev/ksocket/test0`) και τη μεταφορά δεδομένων επάνω από αυτές, ανάμεσα σε διαφορετικά υπολογιστικά συστήματα.

Το τμήμα ksocket-test υλοποιείται ως module του πυρήνα και βασίζεται αποκλειστικά στις υπηρεσίες των KSockets για την κάλυψη των επικοινωνιακών του αναγκών. Στο στρώμα επικοινωνίας παρουσιάζεται ως χρήστης των υπηρεσιών του, ενώ από τον υπόλοιπο πυρήνα του Linux αντιμετωπίζεται ως ένας οδηγός συσκευής χαρακτήρων (§2.1). Η αλληλεπίδρασή του με τα υπόλοιπα τμήματα λογισμικού μπορεί να παρουσιαστεί σχηματικά ως εξής:



Η ανταλλαγή των δεδομένων που μεταφέρονται μέσω της υλοποιούμενης δικτυακής σωλήνωσης γίνεται από τις διεργασίες χρήστη με χρήση των συνηθισμένων κλήσεων συστήματος `read` και `write` στο ειδικό αρχείο συσκευής, έστω `/dev/ksocket/test0`. Η καθοδήγηση του υποσυστήματος ksocket-test για την εγκατάσταση συνδέσεων και την πραγματοποίηση μετρήσεων για τους βασικούς δείκτες επίδοσης, αναλαμβάνεται από ειδική διεργασία ελέγχου στον χώρο χρήστη, η οποία συνοδεύει τον κώδικα πυρήνα του ksocket-test. Η διεργασία ελέγχου που εκτελείται σε χώρο χρήστη, κατευθύνει τον κώδικα πυρήνα στις επιθυμητές ενέργειες εκτελώντας κατάλληλες κλήσεις συστήματος `ioctl`. Για

την εξυπηρέτησή τους, ο κώδικας πυρήνα καλεί απευθείας τις υπηρεσίες `ksocket_*`, όπως αυτές παρουσιάστηκαν στο κεφ. 3.

9.2 Μέτρηση των βασικών δεικτών επίδοσης

9.2.1 Μεθοδολογία των μετρήσεων

Το υποσύστημα ksocket-test χρησιμοποιήθηκε εκτός από τον έλεγχο της καλής λειτουργίας των KSockets και για την πραγματοποίηση μετρήσεων των βασικών δεικτών επίδοσης που αφορούν τις συνδέσεις που δημιουργούνται. Οι δείκτες που μετρήθηκαν είναι ο ρυθμός διαμεταγωγής δεδομένων (bandwidth), και ο χρόνος αρχικής απόκρισης από επίπεδο πυρήνα σε επίπεδο πυρήνα (kernel-to-kernel latency), τόσο στην περίπτωση κατά την οποία χρησιμοποιείται το SCI ως δίκτυο διασύνδεσης, όσο και στην περίπτωση που χρησιμοποιείται δίκτυο βασισμένο στην στοίβα του TCP/IP (συγκεκριμένα το FastEthernet).

Η μέτρηση του ρυθμού διαμεταγωγής έγινε απευθείας από χώρο χρήστη σε χώρο χρήστη, αποστέλλοντας μεγάλη ποσότητα πληροφορίας (της τάξης του 1GB) μέσω του network pipe που υλοποιεί το ksocket-test και μετρώντας τον απαιτούμενο χρόνο για την μεταφοράς τους, για διάφορα μήκη πακέτων πληροφορίας.

Αντίθετα, η μέτρηση του χρόνου απόκρισης γίνεται σε επίπεδο πυρήνα και αφορά το χρόνο που απαιτείται για μετάδοση πακέτων δεδομένων από τον τοπικό κώδικα χώρου πυρήνα ksocket-test στον απομακρυσμένο. Για το σκοπό αυτό, το ένα από τα δύο άκρα της σύνδεσης ρυθμίζεται έτσι ώστε να στέλνει το ταχύτερο δυνατό προς την αντίθετη κατεύθυνση το σύνολο των δεδομένων που λαμβάνονται. Το άλλο άκρο, χρησιμοποιεί το στρώμα επικοινωνίας για την αποστολή πακέτων προοδευτικά αυξανόμενου μεγέθους και μετρά το χρόνο που απαιτείται από την αποστολή έως την επιστροφή τους (χρόνος ανακύκλωσης, round-trip time). Το χρονικό διάστημα που απαιτήθηκε για τη μεταφορά των δεδομένων μόνο ως προς τη μία κατεύθυνση μπορεί να προκύψει ως το μισό του μετρούμενου χρόνου ανακύκλωσης.

Ο χρόνος ανακύκλωσης που μετράται είναι της τάξης των μsec. Για την ακριβή μέτρησή του χρησιμοποιείται ένας ειδικός καταχωρητής μέτρησης του χρόνου, ο οποίος προσφέρεται από ορισμένες αρχιτεκτονικές. Στην περίπτωση της αρχιτεκτονικής Intel x86,

στην οποία βασίζονται οι μετρήσεις που ακολουθούν, ο καταχωρητής αυτός είναι διαθέσιμος στον επεξεργαστή Pentium και τους μεταγενέστερούς του και ονομάζεται TSC (timestamp counter). Κατά τη λειτουργία του επεξεργαστή, η τιμή του καταχωρητή TSC αυξάνεται ακριβώς μία φορά σε κάθε χτύπο ρολογιού. Εφόσον οι συχνότητες ρολογιού που παρέχονται σήμερα είναι της τάξης του 1GHz, βλέπουμε πως η χρήση του TSC επιτρέπει αρκετά μεγάλη ακρίβεια κατά τη μέτρηση μικρών χρονικών διαστημάτων.

Η τιμή του καταχωρητή είναι διαθέσιμη μέσω της εντολής `rdtsc` (read TSC) του επεξεργαστή. Κατά τη διεξαγωγή των μετρήσεων γίνονται δύο αναγνώσεις των τιμών του για κάθε πακέτο πληροφορίας που αποστέλλεται και λαμβάνεται: Μία ακριβώς πριν την αποστολή και μία αμέσως μετά τη λήψη του πακέτου, οπότε ο χρόνος ανακύκλωσης προκύπτει ως η διαφορά των δύο μετρήσεων.

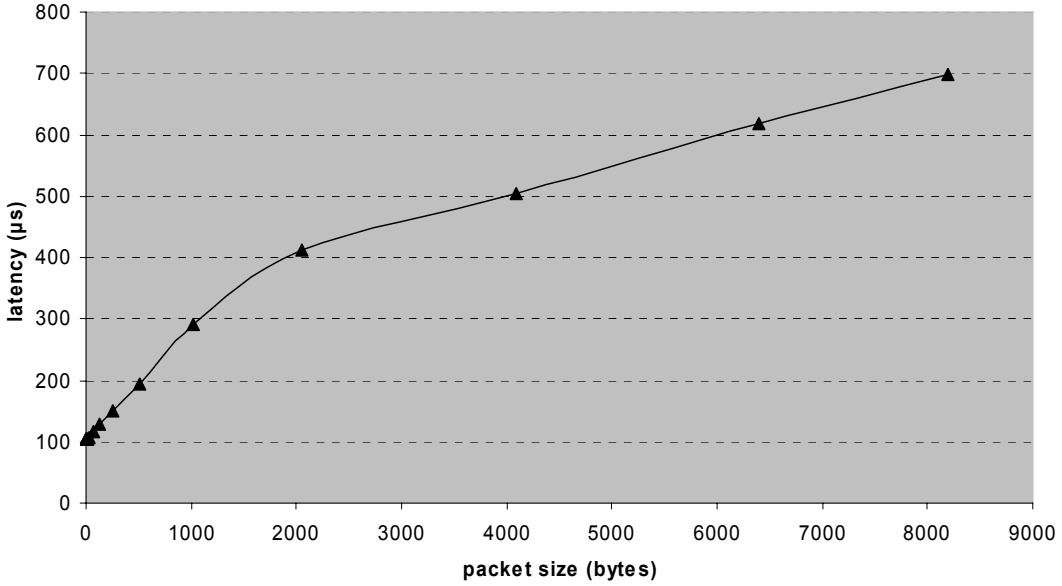
9.2.2 Περιβάλλον διεξαγωγής των μετρήσεων

Οι μετρήσεις που ακολουθούν έγιναν σε συστοιχία υπολογιστών του Εργαστηρίου Υπολογιστικών Συστημάτων. Η σύνθεση των υπολογιστικών συστημάτων συμμετρικής πολυεπεξεργασίας που αποτελούν τη συστοιχία και το λογισμικό που χρησιμοποιήθηκε είναι:

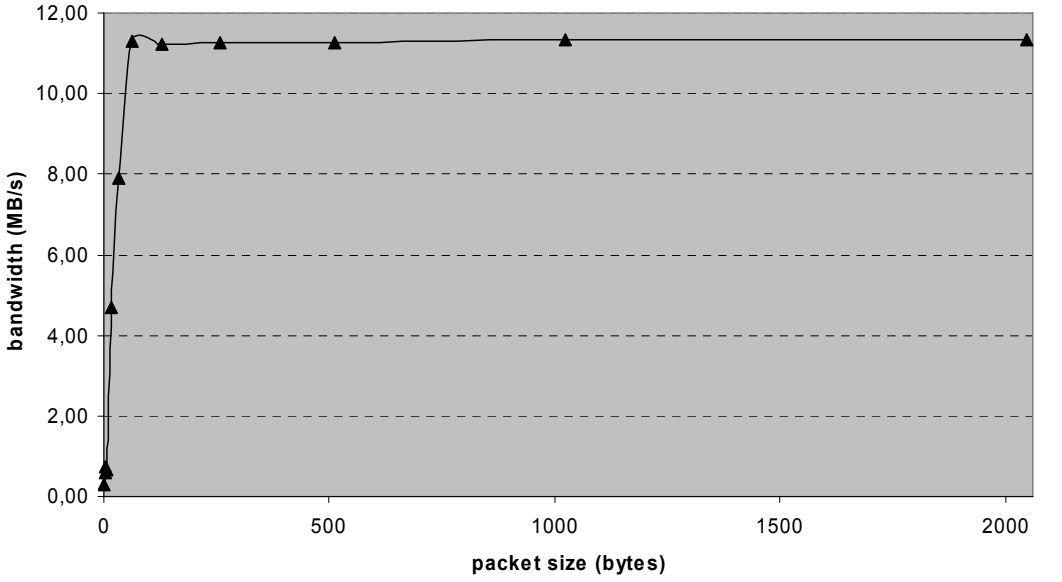
Επεξεργαστές	Δύο Intel Pentium III (Coppermine) 800MHz, 256KB cache
Μητρική πλακέτα	ASUS CUR-DLS
Κεντρική Μνήμη	128MB registered SDRAM, συχνότητα ρολογιού 133MHz
Προσαρμογείς δικτύου	Intel Ethernet Pro 100 για το δίκτυο FastEthernet και Dolphin D330 PSB66 SCI-PCI adapter για το δίκτυο SCI
Λειτουργικό Σύστημα	Linux, πυρήνας έκδοσης 2.4.18
Μεταγλωττιστής C	GNU C Compiler, έκδοση 2.95.4

9.2.3 Αποτελέσματα

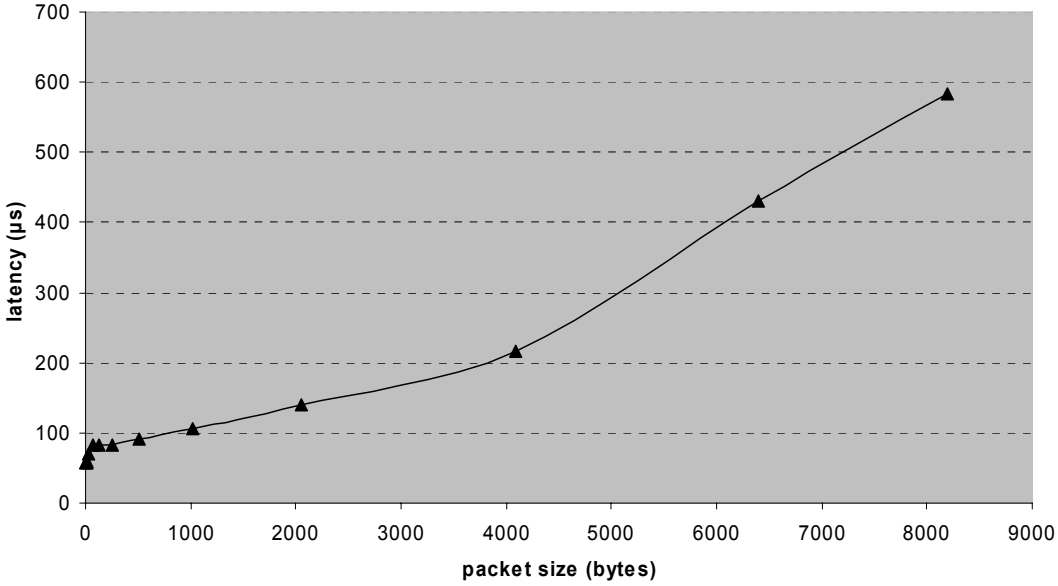
Στα παρακάτω διαγράμματα παρουσιάζονται τα αποτελέσματα των μετρήσεων τόσο στην περίπτωση του δικτύου FastEthernet όσο και στην περίπτωση του δικτύου SCI.



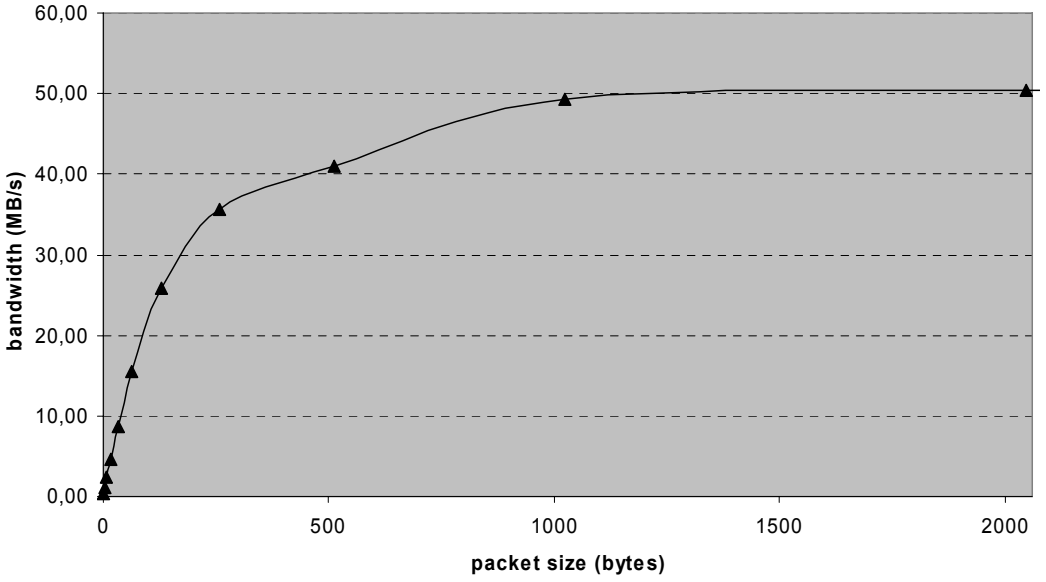
Σχήμα 9.1 – Latency με χρήση του FastEthernet ως δικτύου διασύνδεσης



Σχήμα 9.2 – Ρυθμός διαμεταγωγής με χρήση του FastEthernet ως δικτύου διασύνδεσης



Σχήμα 9.3 – Latency με χρήση του SCI ως δικτύου διασύνδεσης



Σχήμα 9.4 – Ρυθμός διαμεταγωγής με χρήση του SCI ως δικτύου διασύνδεσης

Από τη μελέτη των παραπάνω αποτελεσμάτων των μετρήσεων προκύπτει η υπεροχή του SCI όσον αφορά το latency που εισάγεται στην επικοινωνία, κάτι που ήταν αναμενόμενο

λόγω του μοντέλου επικοινωνίας που χρησιμοποιείται, το οποίο δεν κάνει χρήση στοίβας πρωτοκόλλων για την επεξεργασία των δεδομένων προς μεταφορά.

Το μέγεθος του χρόνου αρχικής απόκρισης στην περίπτωση του SCI δεν οφείλεται τόσο στην πραγματική αναμονή για την ολοκλήρωση των εγγραφών στην απομακρυσμένη μοιραζόμενη μνήμη, μέσω του δικτύου διασύνδεσης, αλλά στην καθυστέρηση που εισάγουν οι απαιτούμενες μεταγωγές περιεχομένου ανάμεσα στο χώρο πυρήνα και το χώρο χρήστη, εφόσον οι λειτουργίες του στρώματος επικοινωνίας είναι κατανεμημένες ανάμεσα στους δύο αυτούς χώρους. Η βελτίωση της απόδοσης του χρονοδρομολογητή του Linux, καθώς και η χρήση νέων, εξελιγμένων βιβλιοθηκών δημιουργίας πολυνηματικών διεργασιών κάτω από το Linux, οι οποίες είναι υπό ανάπτυξη αυτή την περίοδο, περιμένουμε ότι θα συμβάλλουν σημαντικά στη μείωση του latency κατά την επικοινωνία πάνω από το SCI.

Τέλος, για τη μέτρηση του ρυθμού διαμεταγωγής ήταν αναμενόμενη η υπεροχή του SCI λόγω του πολύ υψηλότερου ρυθμού διαμεταγωγής των συνδέσμων σημείο-προς-σημείο που χρησιμοποιεί σε σχέση με το FastEthernet. Παρατηρούμε επίσης πως ο ρυθμός διαμεταγωγής στην περίπτωση του FastEthernet προσεγγίζει το θεωρητικό μέγιστο των 12.5MB/s, αν αυξηθεί αρκετά το μέγεθος του πακέτου δεδομένων.

Βιβλιογραφία – Αναφορές

[1] Γ. Κ. Παπακωνσταντίνου, Ν. Α. Μπιλάλης, Π. Δ. Τσανάκας, *Λειτουργικά Συστήματα Μέρος Ι: Αρχές Λειτουργίας*, Εκδόσεις Συμμετρία, 1999

[2] A. Silberschatz, P. B. Galvin, *Operating System Concepts*, John Wiley and Sons, 5η έκδοση, 1999

[3] Alessandro Rubini, Jonathan Corbet, *Linux Device Drivers*, O' Reilly, 2η έκδοση, 2001

[4] Daniel P. Bovet, Marco Cesati, *Understanding The Linux Kernel*, O' Reilly, 2001

[5] Tigran Aivazian, *Linux Kernel 2.4 Internals*, διαθέσιμο στο διαδίκτυο στη διεύθυνση http://www.moses.uklinux.net/patches/lki.sgml, 2001

[6] Jeremy Elson, *FUSD – A Linux Framework for User-space Drivers*, διαθέσιμο στο διαδίκτυο στη διεύθυνση http://www.circlemud.org/~jelson/software/fusd

[7] David R. Butenhof, *Programming with POSIX Threads*, Addison – Wesley, 1997

[8] Xavier Leroy, *The LinuxThreads Library*, διαθέσιμο στο διαδίκτυο στη διεύθυνση http://pauillac.inria.fr/~xleroy/linuxthreads

[9] Hermann Hellwagner, Alexander Reinefeld, *SCI: Scalable Coherent Interface*, Springer, 1999

[10] ESPRIT Project No. 23174, *SISCI: Standard Software Infrastructure for SCI-based Parallel Systems*, διαθέσιμο στο διαδίκτυο στη διεύθυνση http://www.parallab.uib.no/projects/sisci

[11] IEEE Std. 1596-1992, *IEEE Standard for Scalable Coherent Interface*, The Institute of Electrical and Electronics Engineers, Inc., 1993

[12] The Linux Benchmark website, *Context Switch Latency*, διαθέσιμο στο διαδίκτυο στη διεύθυνση http://cs.nmu.edu/~benchmark/index.php

Παράρτημα Α

Κώδικας του στρώματος επικοινωνίας σε γλώσσα C

Στις ακόλουθες σελίδες περιέχεται ο κώδικας σε γλώσσα C του στρώματος επικοινωνίας KSockets, τόσο του τμήματος που εκτελείται σε χώρο πυρήνα, όσο και του τμήματος που εκτελείται σε χώρο χρήστη. Επιπλέον, περιέχονται ο κώδικας για το υποσύστημα δοκιμής ksocket-test καθώς και τα απαραίτητα Makefiles για την μεταγλώττιση όλων των αρχείων.

αρχείο ksocket.h:

```
/*
 * ksocket.h
 *
 * Definitions used by the KSockets kernel module
 *
 * Vangelis Koukis, 2002
 */

#ifndef _KSOCKET_H
#define _KSOCKET_H

#define KSOCKET_VERSION_STRING     "0.0.1alpha"
#define KSOCKET_DIR_NAME     "ksocket"
#define KSOCKET_DEV_NAME     "ksocket0"
#define KSOCKET_MAJOR        60    /* Reserved for local / experimental use */

/* Compile-time parameters */
#define KSOCKET_MAX_PORTNR   63
#define KSOCKET_BUF_SIZE     2000000
#define KSOCKET_KERNEL_NAME_LEN    16
#define KSOCKET_REQ_QUEUE_SIZE     16

/*
 * Enumeration of possible ksocket states
 * Note:
 * CLOSED = the ksocket has been closed by the ksocket API
 * UNLINKED = the connection to userspace has been broken
 */
typedef enum ksocket_state_enum {
     KSOCKET_OPEN_INPROGRESS, KSOCKET_OPEN,
     KSOCKET_BIND_INPROGRESS, KSOCKET_BOUND,
     KSOCKET_CONNECT_INPROGRESS, KSOCKET_CONNECTED,
     KSOCKET_CLOSE_INPROGRESS, KSOCKET_CLOSED_UNLINKED,
     KSOCKET_CLOSED_LINKED, KSOCKET_OPEN_UNLINKED
} ksocket_state_t;

/*
 * The kernel queues requests of type ksocket_req to the userspace daemon
*/
typedef enum ksocket_req_type_enum {
     KSOCKET_REQ_BIND,
     KSOCKET_REQ_SETOPT,
     KSOCKET_REQ_GETOPT,
     KSOCKET_REQ_OPEN,
     KSOCKET_REQ_CONNECT,
     KSOCKET_REQ_ACCEPT,
     KSOCKET_REQ_CLOSE
} ksocket_req_type_t;

typedef struct ksocket_req_struct {
     ksocket_req_type_t type;
     void *user_data;
     int port;
     char data[KSOCKET_KERNEL_NAME_LEN + 1];
} ksocket_req_t;

#ifdef __KERNEL__
```

```
#include <linux/fs.h>
#include <linux/poll.h>
#include <linux/list.h>
#include <linux/spinlock.h>
#include <linux/devfs_fs_kernel.h>

#include <asm/semaphore.h>

/* Let's define some handy macros */
#ifndef min
#define min(a,b) ((a) < (b) ? (a) : (b))
#endif

#ifdef KSOCKET_DEBUG
#define P_DEBUG(fmt,arg...)  printk(KERN_DEBUG fmt,##arg)
#else
#define P_DEBUG(fmt,arg...)  do { } while(0)
#endif

/*
 * Type definitions
 */

typedef struct circ_buf_struct {
     int size;                /* Size of buffer, in bytes */
     char *start, *end;            /* pointers to start / end of buffer */
     char *rp, *wp;                /* Read and write pointers in buffer */
     struct semaphore sem;         /* Mutual exclusion semaphore */
     wait_queue_head_t rdq, wrq;   /* Wait queues for reads and writes */
     int eof;                 /* Flag to signal EOF on buffer */
} circ_buf;

typedef struct ksocket_struct {
     int port, remote_port;
     char remote_name[KSOCKET_KERNEL_NAME_LEN + 1];
     ksocket_state_t state;
     volatile int *error_p;        /* Pointer to where an error value is expected */
     wait_queue_head_t evqueue;    /* Wait queue for ksocket state changes */
     circ_buf rdbuf, wrbuf;        /* Circular socket I/O buffers */
     spinlock_t slock;        /* Protects the ksocket state */
     void *user_data;         /* Pointer to private userspace data */
     struct list_head list;        /* A ksocket structure can be part of a linked list
*/
} ksocket;

typedef struct ksocket_req_queue_struct {
     int size;                /* Size of request queue, in requests */
     ksocket_req_t *start, *end;
     ksocket_req_t *rp, *wp;
     struct semaphore sem;
     wait_queue_head_t rdq, wrq;
} ksocket_req_queue_t;

/*
 * Function prototypes
 */

inline int request_queue_full(ksocket_req_queue_t *);
inline int init_request_queue(ksocket_req_queue_t *, int);
inline void reset_request_queue(ksocket_req_queue_t *, int);
inline void destroy_request_queue(ksocket_req_queue_t *);
inline int dequeue_request(ksocket_req_queue_t *, ksocket_req_t *, int);
```

```
inline int dequeue_request_to_user(ksocket_req_queue_t *, ksocket_req_t *, int);
inline int enqueue_requeets(ksocket_req_queue_t *, ksocket_req_t *, int);

inline int buf_init(circ_buf *, int);
inline void buf_reset(circ_buf *);
inline void buf_destroy(circ_buf *);
inline void buf_set_eof(circ_buf *);
inline int buf_free(circ_buf *);
inline int buf_read(circ_buf *, char *, int, int, int);
inline int buf_write(circ_buf *, const char *, int, int, int);

int ksocket_fops_open(struct inode *, struct file *);
int ksocket_fops_release(struct inode *, struct file *);
loff_t ksocket_fops_llseek(struct file *filp, loff_t off, int whence);
ssize_t ksocket_fops_read(struct file *, char *, size_t, loff_t *);
ssize_t ksocket_fops_write(struct file *, const char *, size_t, loff_t *);
int ksocket_fops_ioctl(struct inode *, struct file *, unsigned int, unsigned long);
unsigned int ksocket_fops_poll(struct file *, poll_table *);

inline void ksocket_set_error(ksocket *, int);
inline void ksocket_set_eof(ksocket *);

int ksocket_create(ksocket **);
int ksocket_release(ksocket *);
int ksocket_reset(ksocket *);
int ksocket_open(ksocket *, int);
int ksocket_bind(ksocket *, int, int);
int ksocket_connect(ksocket *, const char *, int, int);
ssize_t ksocket_read(ksocket *, char *, int, size_t, int);
ssize_t ksocket_write(ksocket *, const char *, int, size_t, int);
int ksocket_accept(ksocket *, int);
int ksocket_close(ksocket *, int);

#endif     /* __KERNEL__ */

#include <linux/ioctl.h>

struct ioc_setstate {
     int error;
     ksocket_state_t state;
     int port, remote_port;
     char remote_name[KSOCKET_KERNEL_NAME_LEN + 1];
};

/*
 * Definition of ioctl commands
 */
#define KSOCKET_IOC_MAGIC          KSOCKET_MAJOR
#define KSOCKET_IOC_SETSOCKET             _IO(KSOCKET_IOC_MAGIC, 0)
#define KSOCKET_IOC_SETSTATE       _IOW(KSOCKET_IOC_MAGIC, 1, struct ioc_setstate)
#define KSOCKET_IOC_RESETQUEUE           _IO(KSOCKET_IOC_MAGIC, 2)

#define KSOCKET_IOC_MAXNR          2

#endif     /* _KSOCKET_H */
```

αρχείο ksocket-core.c:

```
/*
 * ksocket-core.c
 *
 * Implementation of the kernel-to-user communication mechanism
 * and the KSockets kernel API
 *
 * Vangelis Koukis, 2002
 */

#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/list.h>
#include <linux/poll.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/vmalloc.h>
#include <linux/spinlock.h>
#include <linux/devfs_fs_kernel.h>

#include <asm/uaccess.h>

#include "ksocket.h"

/* Module parameters */
int ksocket_major = KSOCKET_MAJOR;

/* Global variables used for integration with devfs */
devfs_handle_t ksocket_devfs_dir, ksocket_devfs_handle;

/* File operations vector */
struct file_operations ksocket_fops = {
     llseek:           ksocket_fops_llseek,
     read:       ksocket_fops_read,
     write:            ksocket_fops_write,
     readdir:    NULL,
     poll:       ksocket_fops_poll,
     ioctl:            ksocket_fops_ioctl,
     mmap:       NULL,
     open:       ksocket_fops_open,
     flush:            NULL,
     release:    ksocket_fops_release,
     fsync:            NULL,
     fasync:           NULL,
     lock:       NULL,
     readv:            NULL,
     writev:           NULL,
     owner:            THIS_MODULE,
};

/* Queue of requests to userspace daemon */
ksocket_req_queue_t ksocket_req_queue;

/* Queue of ksockets waiting to be linked to userspace, implemented as a linked list */
```

```
LIST_HEAD(ksocket_open_queue);
spinlock_t ksocket_open_slock = SPIN_LOCK_UNLOCKED;  /* Protects ksocket_open_list */

/*
 * Manipulation of circular buffers
 */

inline int buf_free(circ_buf *bufp)
{
     /* Never fill the buffer completely, maximum free space is size - 1 */
     if (bufp->rp == bufp->wp)
           return bufp->size - 1;
     return ((bufp->rp + bufp->size - bufp->wp) % bufp->size) - 1;
}

inline int buf_init(circ_buf *bufp, int size)
{
     /* Allocate buffer space, initialize buffer locks and queues */
     if ( !(bufp->start = vmalloc(size))) {
           printk(KERN_ERR "%s: vmalloc cannot allocate %d bytes\n",
                 __FUNCTION__, size);
           return -ENOMEM;
     }

     bufp->size = size;
     bufp->end = bufp->start + size;

     buf_reset(bufp);
     return 0;
}

inline void buf_reset(circ_buf *bufp)
{
     bufp->rp = bufp->wp = bufp->start;
     bufp->eof = 0;

     init_waitqueue_head(&bufp->rdq);
     init_waitqueue_head(&bufp->wrq);
     sema_init(&bufp->sem, 1);
}

inline void buf_destroy(circ_buf *bufp)
{
     /* Deallocate buffer space */
     vfree(bufp->start);
}

inline void buf_set_eof(circ_buf *bufp)
{
     P_DEBUG("%s: Setting the EOF flag on buffer 0x%p\n",
           __FUNCTION__, bufp);

     /* Set the End-of-file flag on a buffer */
     down(&bufp->sem);
     ++bufp->eof;
     up(&bufp->sem);

     /* Both readers and writers get notified */
     wake_up_interruptible(&bufp->rdq);
     wake_up_interruptible(&bufp->wrq);
}
```

```
inline int buf_read(circ_buf *bufp, char *dest, int is_user_buf,
          int count, int nonblock)
{
     if (down_interruptible(&bufp->sem))
          return -ERESTARTSYS;

     /* If no data is available yet */
     while (bufp->rp == bufp->wp) {
          if (bufp->eof) {
               up(&bufp->sem);
               return 0;
          }
          up(&bufp->sem);               /* Release the lock before sleeping */
          if (nonblock)
               return -EAGAIN;
          if (wait_event_interruptible(bufp->rdq,
                                  bufp->rp != bufp->wp || bufp->eof))
               return -ERESTARTSYS;    /* Got signal, let caller handle it */

          /* Reacquire the lock and loop */
          if (down_interruptible(&bufp->sem))
               return -ERESTARTSYS;
     }

     /* Data is there, check if wp has wrapped */
     if (bufp->wp > bufp->rp)
          count = min(count, bufp->wp - bufp->rp);
     else
          count = min(count, bufp->end - bufp->rp);

     /* Copy data to destination pointer */
     if (is_user_buf) {
          if (copy_to_user(dest, bufp->rp, count)) {
               up(&bufp->sem);
               return -EFAULT;
          }
     } else
          memcpy(dest, bufp->rp, count);

     bufp->rp += count;
     if (bufp->rp == bufp->end)
          bufp->rp = bufp->start;

     up(&bufp->sem);
     wake_up_interruptible(&bufp->wrq);
     P_DEBUG("%s: Buf 0x%p, %d bytes %s, rp is now 0x%p\n",
          __FUNCTION__,bufp,count,is_user_buf ? "to user":"to kernel",bufp->rp);
     return count;
}

inline int buf_write(circ_buf *bufp, const char *source, int is_user_buf,
          int count, int nonblock)
{
     if (down_interruptible(&bufp->sem))
          return -ERESTARTSYS;

     /* If the buffer is full */
     while (!buf_free(bufp)) {
          if (bufp->eof) {
               up(&bufp->sem);
               return -EPIPE;
          }
```

```
            up(&bufp->sem);               /* Release the lock before sleeping */
            if (nonblock)
                  return -EAGAIN;
            if (wait_event_interruptible(bufp->wrq, buf_free(bufp) > 0 || bufp->eof))
                  return -ERESTARTSYS;    /* Got signal, let caller handle it */

            /* Reacquire the lock and loop */
            if (down_interruptible(&bufp->sem))
                  return -ERESTARTSYS;
      }

      /* Check if EOF has been signalled */
      if (bufp->eof) {
            up(&bufp->sem);
            return -EPIPE;
      }

      /* Space is there, fill it up to the end of buffer */
      count = min(count, buf_free(bufp));
      if (bufp->wp >= bufp->rp)
            count = min(count, bufp->end - bufp->wp);

      /* Copy data from source pointer, increase wp */
      if (is_user_buf) {
            if (copy_from_user(bufp->wp, source, count)) {
                  up(&bufp->sem);
                  return -EFAULT;
            }
      } else
            memcpy(bufp->wp, source, count);

      bufp->wp += count;
      if (bufp->wp == bufp->end)
            bufp->wp = bufp->start;

      up(&bufp->sem);
      wake_up_interruptible(&bufp->rdq);
      P_DEBUG("%s: Buf 0x%p, %d bytes %s, wp is now 0x%p\n",
            __FUNCTION__,bufp,count,is_user_buf ? "from user":"from kernel",bufp->wp);
      return count;
}

/*
 * Manipulation of request queue
 */

inline int request_queue_full(ksocket_req_queue_t *q)
{
      int left;

      left = (q->rp + q->size - q->wp) % q->size;
      return (left == 1);
}

inline int init_request_queue(ksocket_req_queue_t *q, int size)
{
      /* Allocate buffer space, initialize buffer locks and queues */
      if ( !(q->start = vmalloc(size * sizeof(ksocket_req_t)))) {
            printk(KERN_ERR "%s: vmalloc cannot allocate %d bytes\n",
                  __FUNCTION__, size);
            return -ENOMEM;
      }
```

```
     reset_request_queue(q, size);
     return 0;
}

inline void reset_request_queue(ksocket_req_queue_t *q, int size)
{
     /* Reset buffer locks and queues */
     q->size = size;
     q->rp = q->wp = q->start;
     q->end = q->start + size;

     init_waitqueue_head(&q->rdq);
     init_waitqueue_head(&q->wrq);
     sema_init(&q->sem, 1);
}

inline void destroy_request_queue(ksocket_req_queue_t *q)
{
     /* Deallocate buffer space */
     if (q->start)
          vfree(q->start);
}

inline int dequeue_request(ksocket_req_queue_t *q, ksocket_req_t *req, int nonblock)
{
     if (down_interruptible(&q->sem))
          return -ERESTARTSYS;

     /* If no data is available yet */
     while (q->rp == q->wp) {
          up(&q->sem);                  /* Release the lock before sleeping */
          if (nonblock)
               return -EAGAIN;
          if (wait_event_interruptible(q->rdq, q->rp != q->wp))
               return -ERESTARTSYS;    /* Got signal, let caller handle it */

          /* Reacquire the lock and loop */
          if (down_interruptible(&q->sem))
               return -ERESTARTSYS;
     }

     /* Dequeue a request and increase rp, taking care of wrapping */
     memcpy(req, q->rp, sizeof(ksocket_req_t));
     if (++q->rp == q->end)
          q->rp = q->start;

     up(&q->sem);
     wake_up_interruptible(&q->wrq);
     P_DEBUG("%s: rp is now 0x%p\n", __FUNCTION__, q->rp);

     return sizeof(ksocket_req_t);
}

inline int dequeue_request_to_user(ksocket_req_queue_t *q, ksocket_req_t *req, int
nonblock)
{
     if (down_interruptible(&q->sem))
          return -ERESTARTSYS;

     /* If no data is available yet */
     while (q->rp == q->wp) {
```

```
            up(&q->sem);                    /* Release the lock before sleeping */
            if (nonblock)
                  return -EAGAIN;
            if (wait_event_interruptible(q->rdq, q->rp != q->wp))
                  return -ERESTARTSYS;    /* Got signal, let caller handle it */

            /* Reacquire the lock and loop */
            if (down_interruptible(&q->sem))
                  return -ERESTARTSYS;
      }

      /* Dequeue a request and increase rp, taking care of wrapping */
      if (copy_to_user(req, q->rp, sizeof(ksocket_req_t)))
            return -EFAULT;
      if (++q->rp == q->end)
            q->rp = q->start;

      up(&q->sem);
      wake_up_interruptible(&q->wrq);
      P_DEBUG("%s: rp is now 0x%p\n", __FUNCTION__, q->rp);

      return sizeof(ksocket_req_t);
}

inline int enqueue_request(ksocket_req_queue_t *q, ksocket_req_t *req, int nonblock)
{
      if (down_interruptible(&q->sem))
            return -ERESTARTSYS;

      /* If the buffer is full */
      while (request_queue_full(q)) {
            up(&q->sem);                    /* Release the lock before sleeping */
            if (nonblock)
                  return -EAGAIN;
            if (wait_event_interruptible(q->wrq, !request_queue_full(q)))
                  return -ERESTARTSYS;    /* Got signal, let caller handle it */

            /* Reacquire the lock and loop */
            if (down_interruptible(&q->sem))
                  return -ERESTARTSYS;
      }

      /* Enqueue a request, increase wp taking care of wrapping */
      memcpy(q->wp, req, sizeof(ksocket_req_t));
      if (++q->wp == q->end)
            q->wp = q->start;

      up(&q->sem);
      wake_up_interruptible(&q->rdq);
      P_DEBUG("%s: wp is now 0x%p\n", __FUNCTION__, q->wp);

      return sizeof(ksocket_req_t);
}

/*
 * Module init and cleanup functions
 */

void __exit ksocket_module_cleanup(void)
{
      P_DEBUG("%s: Unloading module\n", __FUNCTION__);
```

```
#ifdef CONFIG_DEVFS_FS
     P_DEBUG("%s: Performing devfs unregistration\n", __FUNCTION__);
     devfs_unregister(ksocket_devfs_handle);
     devfs_unregister(ksocket_devfs_dir);
#else
     P_DEBUG("%s: Un-registering major number\n", __FUNCTION__);
     if (unregister_chrdev(ksocket_major, "ksocket") < 0)
           printk(KERN_ERR "%s: Could not unregister major %d\n",
                 __FUNCTION__, ksocket_major);
#endif

     destroy_request_queue(&ksocket_req_queue);
}

int __init ksocket_module_init(void)
{
     int res;

     printk(KERN_INFO "KSockets driver version %s\n", KSOCKET_VERSION_STRING);

#ifdef CONFIG_DEVFS_FS
     P_DEBUG("%s: Registering with devfs\n", __FUNCTION__);

     /* Create /dev/ksocket, which contains device files */
     if ( !(ksocket_devfs_dir = devfs_mk_dir(NULL, KSOCKET_DIR_NAME, NULL)))
           return -EBUSY;

     ksocket_devfs_handle = devfs_register(
           ksocket_devfs_dir, KSOCKET_DEV_NAME,
           DEVFS_FL_DEFAULT,
           ksocket_major, 0, S_IFCHR | S_IRUGO | S_IWUGO,
           &ksocket_fops, NULL);
     if (!ksocket_devfs_handle) {
                 printk(KERN_ERR "%s: Could not register device %s\n",
                        __FUNCTION__, KSOCKET_DEV_NAME);
                 return -EBUSY;
     }
#else
     P_DEBUG("%s: Registering major number\n", __FUNCTION__);

     /* Register character device driver */
     if ((res = register_chrdev(ksocket_major, "ksocket", &ksocket_fops)) < 0) {
           printk(KERN_ERR "%s: Could not get major %d\n", __FUNCTION__, ksocket_major);
           return res;
     } else
           P_DEBUG("%s: Using major %d\n", __FUNCTION__, ksocket_major);
#endif

     /* Initialize the request queue */
     if ((res = init_request_queue(&ksocket_req_queue, KSOCKET_REQ_QUEUE_SIZE))) {
           ksocket_module_cleanup();     /* Unregister character device */
           return res;
     }

     return 0;
}

/*
 * Implementation of character device operations
 */

loff_t ksocket_fops_llseek(struct file *filp, loff_t off, int whence)
```

```
{
     /* The request queue and the ksockets are unseekable */
     return -ESPIPE;
}

int ksocket_fops_ioctl(struct inode *inode, struct file *filp,
       unsigned int cmd, unsigned long arg)
{
     ksocket *ksockp;
     int err = 0;
     struct ioc_setstate setstate;

     /*
      * Extract the type and number bitfields, check
      * for appropriate values
      */
     if (_IOC_TYPE(cmd) != KSOCKET_IOC_MAGIC)
           return -ENOTTY;
     if (_IOC_NR(cmd) > KSOCKET_IOC_MAXNR)
           return -ENOTTY;

     /* Ensure that the userspace pointer is valid */
     if (_IOC_DIR(cmd) & _IOC_READ)
           err = !access_ok(VERIFY_WRITE, (void *) arg, _IOC_SIZE(cmd));
     else if (_IOC_DIR(cmd) & _IOC_WRITE)
           err = !access_ok(VERIFY_READ, (void *) arg, _IOC_SIZE(cmd));
     if (err)
           return -EFAULT;

     switch (cmd) {
           case KSOCKET_IOC_SETSOCKET:
                 if (filp->private_data)
                       return -ENOTTY;

                 /* Get the next ksocket from the ksocket_open_queue */
                 spin_lock(&ksocket_open_slock);
                 if (list_empty(&ksocket_open_queue)) {
                       spin_unlock(&ksocket_open_slock);
                       printk(KERN_ERR "%s: ksocket_open_queue found empty\n",
                             __FUNCTION__);
                       return -ENOTTY;
                 }
                 ksockp = list_entry(ksocket_open_queue.next, ksocket, list);
                 list_del(&ksockp->list);
                 spin_unlock(&ksocket_open_slock);

                 /* Set private_data and user_data pointers */
                 filp->private_data = ksockp;
                 ksockp->user_data = (void *) arg;

                 P_DEBUG("%s: IOC_SETSOCKET says ksockp is 0x%p, user_data is "
                        "0x%p\n", __FUNCTION__, ksockp, ksockp->user_data);
                 break;

           case KSOCKET_IOC_SETSTATE:
                 if ( !(ksockp = (ksocket *) filp->private_data))
                       return -ENOTTY;

                 /* Set ksocket state and connection information */
                 if (copy_from_user(&setstate, (struct ioc_setstate *)arg,
                     sizeof(struct ioc_setstate)))
                       return -EFAULT;
```

```
                spin_lock(&ksockp->slock);
                ksockp->state = setstate.state;
                ksockp->port = setstate.port;
                ksockp->remote_port = setstate.remote_port;
                strncpy(ksockp->remote_name, setstate.remote_name,
                      KSOCKET_KERNEL_NAME_LEN + 1);

                /* Ensure string is null-terminated */
                ksockp->remote_name[KSOCKET_KERNEL_NAME_LEN] = '\0';

                /* Return the error value to a function that may be waiting */
                if (ksockp->error_p) {
                      *ksockp->error_p = setstate.error;
                      ksockp->error_p = NULL;
                }
                spin_unlock(&ksockp->slock);

                /* Wake up any processes waiting for the ksocket state to change */
                wake_up_interruptible(&ksockp->evqueue);

                P_DEBUG("%s: IOC_SETSTATE for 0x%p to {%d, %d, %d, \"%s\", %d}\n",
                       __FUNCTION__, ksockp, setstate.state, setstate.error,
                       setstate.port, setstate.remote_name, setstate.remote_port);
                break;

          case KSOCKET_IOC_RESETQUEUE:
                if (filp->private_data)
                      return -ENOTTY;

                /*
                 * Clear the request queue. Used by userspace when the userspace
                 * daemon (re-)starts, possibly after abnormal termination.
                 * There is no danger of old ksockets queueing faulty requests to
                 * userspace, because they will all have been closed by
                 * ksocket_fops_release, as the file descriptors of the userspace
                 * daemon were being closed on process termination.
                 */
                reset_request_queue(&ksocket_req_queue, KSOCKET_REQ_QUEUE_SIZE);

                P_DEBUG("%s: IOC_RESETQUEUE for 0x%p\n",
                      __FUNCTION__, &ksocket_req_queue);
                break;

          default:
                return -ENOTTY;
     }

     return 0;
}

unsigned int ksocket_fops_poll(struct file *filp, poll_table *wait)
{
     ksocket *ksockp;
     unsigned int mask = 0;

      /* If polling the request queue */
     if (!filp->private_data) {
           /* This wait queue is woken up when a new request arrives */
           poll_wait(filp, &ksocket_req_queue.rdq, wait);

           /* If the request queue is not empty, then it is readable */
```

```
            if (ksocket_req_queue.rp != ksocket_req_queue.wp)
                  mask |= POLLIN | POLLRDNORM;

            /* The request queue is not writable from userspace */
            return mask;
      }

      ksockp = (ksocket *) filp->private_data;

      poll_wait(filp, &ksockp->wrbuf.rdq, wait);
      poll_wait(filp, &ksockp->rdbuf.wrq, wait);

      /* Does the kernel output buffer contain any data? */
      if (ksockp->wrbuf.rp != ksockp->wrbuf.wp)
            mask |= POLLIN | POLLRDNORM;
      else {
            /* If EOF is set, set the POLLHUP bit */
            if (ksockp->wrbuf.eof)
                  mask |= POLLHUP;

            /* If not connected, read() will not block */
            if (ksockp->state != KSOCKET_CONNECTED)
                  mask |= POLLIN;
      }

      /*
       * write() only blocks if the kernel input buffer is full
       * and the ksocket is in the KSOCKET_CONNECTED state
       */
      if (ksockp->state != KSOCKET_CONNECTED || buf_free(&ksockp->rdbuf))
            mask |= POLLOUT | POLLWRNORM;

      return mask;
}

int ksocket_fops_open(struct inode *inode, struct file *filp)
{
      /* Make sure that the value of filp->private_data is set to NULL */

#ifndef CONFIG_DEVFS_FS
      int minor = MINOR(inode->i_rdev);

      P_DEBUG("%s: Called for minor %d\n", __FUNCTION__, minor);
      if (minor)
            return -ENODEV;

      filp->private_data = NULL;
#else
      if (filp->private_data) {
            printk(KERN_ERR "%s: Using devfs, but private_data is not NULL!\n",
                  __FUNCTION__);
            return -EINVAL;
      } else
            P_DEBUG("%s: Called using devfs, private_data is already NULL\n",
                  __FUNCTION__);
#endif

      MOD_INC_USE_COUNT;
      return 0;               /* Return success */
}

int ksocket_fops_release(struct inode *inode, struct file *filp)
```

```
{
     int oldstate;
     ksocket *ksockp;

     /* Nothing needs to be done if the control fd is being closed */
     if (!filp->private_data)
           goto out;

     ksockp = (ksocket *) filp->private_data;
     oldstate = ksockp->state;

     spin_lock(&ksockp->slock);
     switch (ksockp->state) {
           case KSOCKET_OPEN_UNLINKED:
           case KSOCKET_CLOSED_UNLINKED:
                 spin_unlock(&ksockp->slock);
                 printk(KERN_ERR "%s: Failed assertion: ksocket state is %d "
                       "(UNLINKED)\n", __FUNCTION__, ksockp->state);
                 break;
           case KSOCKET_OPEN_INPROGRESS:
           case KSOCKET_BIND_INPROGRESS:
           case KSOCKET_CONNECT_INPROGRESS:
                 /* Userspace has not acknowledged a request, yet it closes the fd? */
                 ksockp->state = KSOCKET_CLOSED_UNLINKED;
                 ksocket_set_error(ksockp, ECONNABORTED);
                 spin_unlock(&ksockp->slock);
                 ksocket_set_eof(ksockp);
                 printk(KERN_ERR "%s: ksocket request was in progress. State was "
                       "%d (INPROGRESS)\n", __FUNCTION__, ksockp->state);
                 break;
           case KSOCKET_CONNECTED:
                 ksockp->state = KSOCKET_OPEN_UNLINKED;
                 ksocket_set_error(ksockp, ECONNRESET);    /* The remote disconnected */
                 spin_unlock(&ksockp->slock);
                 ksocket_set_eof(ksockp);
                 break;
           case KSOCKET_CLOSED_LINKED:
                 ksockp->state = KSOCKET_CLOSED_UNLINKED; /* The local kernel
disconnected */
                 spin_unlock(&ksockp->slock);
                 break;
           case KSOCKET_OPEN:
           case KSOCKET_BOUND:
                 ksockp->state = KSOCKET_CLOSED_UNLINKED;
                 ksocket_set_error(ksockp, 0);
                 spin_unlock(&ksockp->slock);
                 printk(KERN_ERR "%s: ksocket fd closed unexpectedly, state was "
                       "%d (NO REQUEST)\n", __FUNCTION__, ksockp->state);
                 /* fallthrough */
           case KSOCKET_CLOSE_INPROGRESS:
                 ksockp->state = KSOCKET_CLOSED_UNLINKED; /* ksocket_close was called */
                 ksocket_set_error(ksockp, 0);
                 spin_unlock(&ksockp->slock);
                 break;
           default:
                 spin_unlock(&ksockp->slock);
                 printk(KERN_ERR "%s: ksocket is in invalid state %d!\n",
                       __FUNCTION__, ksockp->state);
     }
     wake_up_interruptible(&ksockp->evqueue);
     P_DEBUG("%s: Socket state was %d, now is %d\n",
           __FUNCTION__, oldstate, ksockp->state);
```

```
out:
     MOD_DEC_USE_COUNT;
     return 0;
}

ssize_t ksocket_fops_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
     ksocket *ksockp;

     if (f_pos != &filp->f_pos)
           return -ESPIPE;

     /*
      * if reading from the control connection, then
      * dequeue a request from ksocket_req_queue and pass it to userspace
      */
     if (!filp->private_data) {
           if (count != sizeof(ksocket_req_t)) {
                 P_DEBUG("%s: Cannot return %d != %d bytes\n",
                       __FUNCTION__, count, sizeof(ksocket_req_t));
                 return -EINVAL;
           }

           return dequeue_request_to_user(&ksocket_req_queue, (ksocket_req_t *) buf,
                 filp->f_flags & O_NONBLOCK);
     }

     /* Read from a ksocket connection */
     ksockp = (ksocket *) filp->private_data;
     P_DEBUG("%s: ksocket at 0x%p, buf = 0x%p, count = %ld\n",
           __FUNCTION__, ksockp, buf, (long)count);

     spin_lock(&ksockp->slock);
     if (ksockp->state != KSOCKET_CONNECTED &&
         ksockp->state != KSOCKET_CLOSED_LINKED) {
           spin_unlock(&ksockp->slock);
           return -EINVAL;
     }
     spin_unlock(&ksockp->slock);

     return buf_read(&ksockp->wrbuf, buf, 1, count, filp->f_flags & O_NONBLOCK);
}

ssize_t ksocket_fops_write(struct file *filp, const char *buf, size_t count, loff_t
*f_pos)
{
     int err;
     ksocket *ksockp;

     if (f_pos != &filp->f_pos)
           return -ESPIPE;

      /* Refuse to write to the request queue from userspace */
     if (!filp->private_data)
           return -EINVAL;

     ksockp = (ksocket *) filp->private_data;
     P_DEBUG("%s: ksocket at 0x%p, buf = 0x%p, count = %ld\n",
           __FUNCTION__, ksockp, buf, (long)count);

     spin_lock(&ksockp->slock);
```

```
    if (ksockp->state != KSOCKET_CONNECTED &&
        ksockp->state != KSOCKET_CLOSED_LINKED) {
          spin_unlock(&ksockp->slock);
          return -EINVAL;
    }
    spin_unlock(&ksockp->slock);

    err = buf_write(&ksockp->rdbuf, buf, 1, count, filp->f_flags & O_NONBLOCK);

    /* The ksocket has been closed, send SIGPIPE to the calling process */
    if (err == -EPIPE)
          send_sig(SIGPIPE, current, 0);
    return err;
}

/*
 * Implementation of KSockets API
 */

inline void ksocket_set_error(ksocket *ksockp, int value)
{
    if (ksockp->error_p) {
          *ksockp->error_p = value;
          ksockp->error_p = NULL;
    }
}

inline void ksocket_set_eof(ksocket *ksockp)
{
    buf_set_eof(&ksockp->rdbuf);
    buf_set_eof(&ksockp->wrbuf);
}

/*
 * ksocket_create:    Allocate memory for a new ksocket structure,
 *               initialize ksocket I/O buffers
 */
int ksocket_create(ksocket **ksockp)
{
    int res;
    ksocket *p;

    /* Allocate a new ksocket structure */
    if ((p = kmalloc(sizeof(ksocket), GFP_KERNEL)) == NULL)
          return -ENOMEM;

    /* Init lock and wait queue */
    init_waitqueue_head(&p->evqueue);
    spin_lock_init(&p->slock);

    /* Init ksocket buffers */
    if ((res = buf_init(&p->wrbuf, KSOCKET_BUF_SIZE)) < 0) {
          kfree(p);
          return res;
    }
    if ((res = buf_init(&p->rdbuf, KSOCKET_BUF_SIZE)) < 0) {
          buf_destroy(&p->wrbuf);
          kfree(p);
          return res;
    }

    ksocket_reset(p); /* Initialize structure fields */
```

```
     *ksockp = p;

     P_DEBUG("%s: new ksocket at 0x%p\n", __FUNCTION__, p);
     return 0;
}

/*
 * ksocket_reset:      Reset a ksocket structure to its original
 *               condition so that it can be used again
 */
int ksocket_reset(ksocket *ksockp)
{
     /* Init structure fields */
     ksockp->error_p = NULL;
     ksockp->port = 0;
     ksockp->remote_port = 0;
     ksockp->remote_name[0] = '\0';
     ksockp->state = KSOCKET_CLOSED_UNLINKED;

     /* Reset I/O buffers, clear any remaining data */
     buf_reset(&ksockp->rdbuf);
     buf_reset(&ksockp->wrbuf);

     return 0;
}

/*
 * ksocket_release:    Release memory allocated for a ksocket structure
 *               and its I/O buffers
 */
int ksocket_release(ksocket *ksockp)
{
     P_DEBUG("%s: ksocket at 0x%p\n", __FUNCTION__, ksockp);

     buf_destroy(&ksockp->wrbuf);
     buf_destroy(&ksockp->rdbuf);
     kfree(ksockp);

     return 0;
}

/*
 * ksocket_open:       Connect a ksocket structure to the userspace daemon
 */
int ksocket_open(ksocket *ksockp, int nonblock)
{
     int res;
     volatile int userspace_res = 1;
     ksocket_req_t req;

     P_DEBUG("%s: ksocket at 0x%p\n", __FUNCTION__, ksockp);

     /* Sanity checks */
     if (!ksockp)
           return -EINVAL;

     spin_lock(&ksockp->slock);
     if (ksockp->state != KSOCKET_CLOSED_UNLINKED) {
           spin_unlock(&ksockp->slock);
           return -EINVAL;
     }
     ksockp->state = KSOCKET_OPEN_INPROGRESS;
```

```
	ksockp->error_p = &userspace_res;
	spin_unlock(&ksockp->slock);

	/* Insert this ksocket in the ksocket_open_queue */
	spin_lock(&ksocket_open_slock);
	list_add_tail(&ksockp->list, &ksocket_open_queue);
	spin_unlock(&ksocket_open_slock);

	/* Insert an OPEN request in request queue */
	req.type = KSOCKET_REQ_OPEN;

	if ((res = enqueue_request(&ksocket_req_queue, &req, nonblock)) < 0) {
		spin_lock(&ksockp->slock);		/* Restore ksocket state */
		ksockp->state = KSOCKET_CLOSED_UNLINKED;

		/* This ksocket must be removed from the ksocket_open_queue */
		spin_lock(&ksocket_open_slock);
		list_del(&ksockp->list);
		spin_unlock(&ksocket_open_slock);

		goto out;
	}

	/* Wait until the open request is satisfied */
	spin_lock(&ksockp->slock);
	while (userspace_res == 1) {
		if (nonblock) {
			res = -EINPROGRESS;
			goto out;
		}
		spin_unlock(&ksockp->slock);
		if (wait_event_interruptible(ksockp->evqueue, userspace_res != 1)) {
			spin_lock(&ksockp->slock);
			res = -EINPROGRESS;
			goto out;
		}
		spin_lock(&ksockp->slock);
	}

	/* Lock held, an error value has been returned */
	res = userspace_res;
out:
	ksockp->error_p = NULL;
	spin_unlock(&ksockp->slock);
	return res;
}

/*
 * ksocket_bind: Bind a ksocket to a specific communication port
 */
int ksocket_bind(ksocket *ksockp, int port, int nonblock)
{
	int res;
	volatile int userspace_res = 1;
	ksocket_req_t req;

	P_DEBUG("%s: ksocket at 0x%p, port %d\n", __FUNCTION__, ksockp, port);

	/* Sanity checks */
	if (port < 0 || port > KSOCKET_MAX_PORTNR)
		return -EINVAL;
	if (!ksockp)
```

```
        return -EINVAL;

    spin_lock(&ksockp->slock);
    if (ksockp->state != KSOCKET_OPEN) {
        spin_unlock(&ksockp->slock);
        return -EINVAL;
    }
    ksockp->state = KSOCKET_BIND_INPROGRESS;
    ksockp->error_p = &userspace_res;
    spin_unlock(&ksockp->slock);

    /* Insert request in request queue */
    req.type = KSOCKET_REQ_BIND;
    req.port = port;
    req.user_data = ksockp->user_data;

    if ((res = enqueue_request(&ksocket_req_queue, &req, nonblock)) < 0) {
        spin_lock(&ksockp->slock);          /* Restore ksocket state */
        ksockp->state = KSOCKET_OPEN;
        goto out;
    }

    /* Wait until the bind request is satisfied */
    spin_lock(&ksockp->slock);
    while (userspace_res == 1) {
        if (nonblock) {
            res = -EINPROGRESS;
            goto out;
        }
        spin_unlock(&ksockp->slock);
        if (wait_event_interruptible(ksockp->evqueue, userspace_res != 1)) {
            spin_lock(&ksockp->slock);
            res = -EINPROGRESS;
            goto out;
        }
        spin_lock(&ksockp->slock);
    }

    /* Lock held, an error value has been returned */
    res = userspace_res;
out:
    ksockp->error_p = NULL;
    spin_unlock(&ksockp->slock);
    return res;
}

/*
 * ksocket_connect:    Establish connection to a remote kernel
 */
int ksocket_connect(ksocket *ksockp, const char *dest, int port, int nonblock)
{
    int res;
    volatile int userspace_res = 1;
    ksocket_req_t req;

    P_DEBUG("%s: ksocket at 0x%p, remote = {\"%s\", %d}\n",
        __FUNCTION__, ksockp, dest, port);

    /* Sanity checks */
    if (!ksockp || !dest ||
        port < 0 || port > KSOCKET_MAX_PORTNR ||
        strlen(dest) > KSOCKET_KERNEL_NAME_LEN)
```

```
        return -EINVAL;

    spin_lock(&ksockp->slock);
    if (ksockp->state != KSOCKET_BOUND) {
        spin_unlock(&ksockp->slock);
        return -EINVAL;
    }
    ksockp->state = KSOCKET_CONNECT_INPROGRESS;
    ksockp->error_p = &userspace_res;
    spin_unlock(&ksockp->slock);

    /* Insert request in request queue */
    req.type = KSOCKET_REQ_CONNECT;
    req.user_data = ksockp->user_data;
    req.port = port;
    strcpy(req.data, dest);

    if ((res = enqueue_request(&ksocket_req_queue, &req, nonblock)) < 0) {
        spin_lock(&ksockp->slock);          /* Restore ksocket state */
        ksockp->state = KSOCKET_BOUND;
        goto out;
    }

    /* Wait until the connect request is satisfied */
    spin_lock(&ksockp->slock);
    while (userspace_res == 1) {
        if (nonblock) {
            res = -EINPROGRESS;
            goto out;
        }
        spin_unlock(&ksockp->slock);
        if (wait_event_interruptible(ksockp->evqueue, userspace_res != 1)) {
            spin_lock(&ksockp->slock);
            res = -EINPROGRESS;
            goto out;
        }
        spin_lock(&ksockp->slock);
    }

    /* Lock held, an error value has been returned */
    res = userspace_res;
out:
    ksockp->error_p = NULL;
    spin_unlock(&ksockp->slock);
    return res;
}

/*
 * ksocket_accept:     Mark ksocket as ready to accept remote connections
 */
int ksocket_accept(ksocket *ksockp, int nonblock)
{
    int res;
    volatile int userspace_res = 1;
    ksocket_req_t req;

    P_DEBUG("%s: ksocket at 0x%p\n", __FUNCTION__, ksockp);

    /* Sanity checks */
    if (!ksockp)
        return -EINVAL;
```

```
	spin_lock(&ksockp->slock);
	if (ksockp->state != KSOCKET_BOUND) {
		spin_unlock(&ksockp->slock);
		return -EINVAL;
	}
	ksockp->state = KSOCKET_CONNECT_INPROGRESS;
	ksockp->error_p = &userspace_res;
	spin_unlock(&ksockp->slock);

	/* Insert request in request queue */
	req.type = KSOCKET_REQ_ACCEPT;
	req.user_data = ksockp->user_data;

	if ((res = enqueue_request(&ksocket_req_queue, &req, nonblock)) < 0) {
		spin_lock(&ksockp->slock);	/* Restore ksocket state */
		ksockp->state = KSOCKET_BOUND;
		goto out;
	}

	/* Wait until the accept request is satisfied */
	spin_lock(&ksockp->slock);
	while (userspace_res == 1) {
		if (nonblock) {
			res = -EINPROGRESS;
			goto out;
		}
		spin_unlock(&ksockp->slock);
		if (wait_event_interruptible(ksockp->evqueue, userspace_res != 1)) {
			spin_lock(&ksockp->slock);
			res = -EINPROGRESS;
			goto out;
		}
		spin_lock(&ksockp->slock);
	}

	/* Lock held, an error value has been returned */
	res = userspace_res;
out:
	ksockp->error_p = NULL;
	spin_unlock(&ksockp->slock);
	return res;
}

/*
 * ksocket_close:	Request disconnection from remote kernel and userspace
 */
int ksocket_close(ksocket *ksockp, int nonblock)
{
	int res;
	volatile int *old_error_p, userspace_res = 1;
	ksocket_req_t req;
	ksocket_state_t old_state;

	P_DEBUG("%s: ksocket at 0x%p\n", __FUNCTION__, ksockp);

	/* Sanity checks */
	if (!ksockp)
		return -EINVAL;

	spin_lock(&ksockp->slock);
	switch (ksockp->state) {
		case KSOCKET_OPEN_INPROGRESS:
```

```
        case KSOCKET_BIND_INPROGRESS:
        case KSOCKET_CONNECT_INPROGRESS:
        case KSOCKET_CLOSED_LINKED:
        case KSOCKET_CLOSED_UNLINKED:
        case KSOCKET_CLOSE_INPROGRESS:
                /*
                 * Refuse to close the ksocket when a request is pending,
                 * or when it is already closed. Really bad things could
                 * happen when userspace decides to set the state of the
                 * device using the IOC_SETSTATE ioctl
                 */
                spin_unlock(&ksockp->slock);
                return -EINVAL;
        case KSOCKET_OPEN_UNLINKED:
                /* Connection to userspace has already been broken */
                ksockp->state = KSOCKET_CLOSED_UNLINKED;
                wake_up_interruptible(&ksockp->evqueue);
                spin_unlock(&ksockp->slock);
                return 0;
        case KSOCKET_CONNECTED:
                /* Only userspace can now access the ksocket buffers */
                ksockp->state = KSOCKET_CLOSED_LINKED;
                ksocket_set_eof(ksockp);
                wake_up_interruptible(&ksockp->evqueue);
                spin_unlock(&ksockp->slock);
                return 0;
        case KSOCKET_OPEN:
        case KSOCKET_BOUND:
                /* Continue, enqueue a REQ_CLOSE to userspace */
                break;
        default:
                printk(KERN_ERR "%s: ksocket is in invalid state %d!\n",
                        __FUNCTION__, ksockp->state);
                spin_unlock(&ksockp->slock);
                return -EINVAL;
}

/* Save old ksocket state and error pointer */
old_state = ksockp->state;
ksockp->state = KSOCKET_CLOSE_INPROGRESS;
old_error_p = ksockp->error_p;
ksockp->error_p = &userspace_res;
spin_unlock(&ksockp->slock);

/* Insert request in request queue */
req.type = KSOCKET_REQ_CLOSE;
req.user_data = ksockp->user_data;

if ((res = enqueue_request(&ksocket_req_queue, &req, nonblock)) < 0) {
        spin_lock(&ksockp->slock);          /* Restore ksocket state */
        ksockp->state = old_state;
        ksockp->error_p = old_error_p;
        spin_unlock(&ksockp->slock);
        return res;
}

/* Wait until the close request is satisfied */
spin_lock(&ksockp->slock);
while (userspace_res == 1) {
        if (nonblock) {
                res = -EINPROGRESS;
                goto out;
```

```
            }
            spin_unlock(&ksockp->slock);
            if (wait_event_interruptible(ksockp->evqueue, userspace_res != 1)) {
                  spin_lock(&ksockp->slock);
                  res = -EINPROGRESS;
                  goto out;
            }
            spin_lock(&ksockp->slock);
      }

      /* Lock held, an error value has been returned */
      res = userspace_res;
out:
      ksockp->error_p = NULL;
      spin_unlock(&ksockp->slock);
      return res;
}

ssize_t ksocket_read(ksocket *ksockp, char *buf, int is_user_buf,
      size_t size, int nonblock)
{
      P_DEBUG("%s: ksocket at 0x%p, param = { 0x%p, %ld }\n",
            __FUNCTION__, ksockp, buf, (long)size);

      /* Sanity check */
      if (!ksockp)
            return -EINVAL;

      spin_lock(&ksockp->slock);
      if (ksockp->state != KSOCKET_CONNECTED &&
          ksockp->state != KSOCKET_OPEN_UNLINKED) {
            spin_unlock(&ksockp->slock);
            return -ENOTCONN;
      }
      spin_unlock(&ksockp->slock);

      /* Return data from the ksocket input buffer, used for userspace writes */
      return buf_read(&ksockp->rdbuf, buf, is_user_buf, size, nonblock);
}

ssize_t ksocket_write(ksocket *ksockp, const char *buf, int is_user_buf,
            size_t size, int nonblock)
{
      P_DEBUG("%s: ksocket at 0x%p, param = { 0x%p, %ld }\n",
            __FUNCTION__, ksockp, buf, (long)size);

      /* Sanity check */
      if (!ksockp)
            return -EINVAL;

      spin_lock(&ksockp->slock);
      if (ksockp->state != KSOCKET_CONNECTED &&
          ksockp->state != KSOCKET_OPEN_UNLINKED) {
            spin_unlock(&ksockp->slock);
            return -ENOTCONN;
      }
      spin_unlock(&ksockp->slock);

      /* Write data to the ksocket output buffer, used for userspace reads */
      return buf_write(&ksockp->wrbuf, buf, is_user_buf, size, nonblock);
}
```

```
/*
 * Module information section
 */

MODULE_AUTHOR("Vangelis Koukis");
MODULE_LICENSE("GPL");

MODULE_PARM(ksocket_major, "i");
MODULE_PARM_DESC(ksocket_major, "The major number to be used by the ksocket driver");

module_init(ksocket_module_init);
module_exit(ksocket_module_cleanup);
```

αρχείο ksocket-test.h:

```
/*
 * ksocket.h
 *
 * Definitions used by the KSockets kernel module
 *
 * Vangelis Koukis, 2002
 */

#ifndef _KSOCKET_TEST_H
#define _KSOCKET_TEST_H

#include <linux/ioctl.h>

#include "ksocket.h"

#ifdef __KERNEL__
#include <asm/atomic.h>

#define KSOCKET_TEST_VERSION_STRING      "0.1alpha"
#define KSOCKET_TEST_DEV_NAME            "test0"
#define KSOCKET_TEST_MAJOR         61
#define KSOCKET_TEST_DEVCOUNT            4     /* Default number of devices */
#define KSOCKET_TEST_MAXDEVCOUNT   50    /* Maximum number of devices */

/* Parameters for benchmarking functions */
#define KSOCKET_TEST_BENCH_BUF           2000000
#define KSOCKET_TEST_BENCH_CNT           50

/* Definition of KSocket_Dev type */
typedef struct ksocket_test_dev_struct {
     atomic_t users;               /* Counter of device users */
     ksocket *ksockp;
     devfs_handle_t handle;        /* Only used if devfs is there */
} test_dev;

/*
 * Function prototypes
 */
int ksocket_test_ioctl(struct inode *, struct file *, unsigned int, unsigned long);
int ksocket_test_open(struct inode *, struct file *);
int ksocket_test_release(struct inode *, struct file *);
ssize_t ksocket_test_read(struct file *, char *, size_t, loff_t *);
ssize_t ksocket_test_write(struct file *, const char *, size_t, loff_t *);

#endif  /* __KERNEL__ */

/*
 * Definition of ioctl commands
 */
struct ioc_remotedata {
     int port;
     char name[KSOCKET_KERNEL_NAME_LEN + 1];
};

#define KSOCKET_TEST_IOC_MAGIC           61
#define KSOCKET_TEST_IOC_OPEN            _IO(KSOCKET_TEST_IOC_MAGIC, 0)
#define KSOCKET_TEST_IOC_BIND            _IOW(KSOCKET_TEST_IOC_MAGIC, 1, int *)
```

```
#define KSOCKET_TEST_IOC_CONNECT   _IOW(KSOCKET_TEST_IOC_MAGIC, 2, struct ioc_remotedata
*)
#define KSOCKET_TEST_IOC_ACCEPT          _IO(KSOCKET_TEST_IOC_MAGIC, 3)
#define KSOCKET_TEST_IOC_CLOSE           _IO(KSOCKET_TEST_IOC_MAGIC, 4)
#define KSOCKET_TEST_IOC_RESET           _IO(KSOCKET_TEST_IOC_MAGIC, 5)
#define KSOCKET_TEST_IOC_STATUS          _IOR(KSOCKET_TEST_IOC_MAGIC, 6, struct
ioc_setstate *)

/* Used for benchmarking */
#define KSOCKET_TEST_IOC_BENCH_MIRROR    _IO(KSOCKET_TEST_IOC_MAGIC, 7)
#define KSOCKET_TEST_IOC_BENCH_LATENCY   _IO(KSOCKET_TEST_IOC_MAGIC, 8)

#define KSOCKET_TEST_IOC_MAXNR           8

#endif     /* _KSOCKET_TEST_H */
```

αρχείο ksocket-test.c:

```
/*
 * ksocket-test.c
 *
 * KSockets test module
 *
 * Vangelis Koukis, 2002
 */

#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/ioctl.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/vmalloc.h>
#include <linux/devfs_fs_kernel.h>

#include <asm/atomic.h>
#include <asm/uaccess.h>

#include "ksocket-test.h"

/* Module parameters */
int ksocket_test_major = KSOCKET_TEST_MAJOR;
int ksocket_test_devcount = KSOCKET_TEST_DEVCOUNT;

/* Pointer to dynamically allocated array of test devices */
test_dev *devices;

/* File operations vector */
struct file_operations ksocket_test_fops = {
     llseek:           NULL,
     read:       ksocket_test_read,
     write:            ksocket_test_write,
     readdir:    NULL,
     poll:       NULL,
     ioctl:            ksocket_test_ioctl,
     mmap:       NULL,
     open:       ksocket_test_open,
     flush:            NULL,
     release:    ksocket_test_release,
     fsync:            NULL,
     fasync:           NULL,
     lock:       NULL,
     readv:            NULL,
     writev:           NULL,
     owner:            THIS_MODULE,
};

/*
 * Functions used for benchmarking
 */

inline int ksocket_test_bench_read(ksocket *ksockp, char *buf, size_t n)
{
```

```
	int tmp;

	/* Receive data from ksocket, insist on partial reads */
	while (n > 0) {
		tmp = ksocket_read(ksockp, buf, 0, n, 0);
		if (tmp <= 0)
			return tmp;
		n -= tmp;
		buf += tmp;
	}

	if (n < 0)
		printk(KERN_ERR "%s: This shouldn't have happened, n = %d\n",
			__FUNCTION__, n);
	return 0;
}

inline int ksocket_test_bench_write(ksocket *ksockp, char *buf, size_t n)
{
	int tmp;

	/* Send data to ksocket, insist on partial writes */
	while (n > 0) {
		tmp = ksocket_write(ksockp, buf, 0, n, 0);
		if (tmp < 0)
			return tmp;
		n -= tmp;
		buf += tmp;
	}

	if (n < 0)
		printk(KERN_ERR "%s: This shouldn't have happened, n = %d\n",
			__FUNCTION__, n);

	return 0;
}

/* Measure latency, by computing the round-trip time */
inline int ksocket_test_bench_latency(ksocket *ksockp, size_t n)
{
	int i, res;
	char *lbuf;
	cycles_t t0, t1;
	cycles_t min = INT_MAX, max = 0, avg = 0;

	if (! (lbuf = kmalloc(n, GFP_KERNEL)))
		return -ENOMEM;

	for (i = 1; i <= KSOCKET_TEST_BENCH_CNT; i++) {

	t0 = get_cycles();	/* Read the Pentium timestamp counter */
	if ((res = ksocket_test_bench_write(ksockp, lbuf, n)) < 0) /* Ping */
		goto bail_out;
	if ((res = ksocket_test_bench_read(ksockp, lbuf, n)) < 0)  /* Pong */
		goto bail_out;
	t1 = get_cycles();

	/* Sum up the cycles used for this packet */
	t1 -= t0;
	if (t1 < min)
		min = t1;
	if (t1 > max)
```

```
            max = t1;
      avg += t1;

      }

      /* These are round-trip times and must be divided in half */
      min >>= 1;
      max >>= 1;
      avg = (long)avg / KSOCKET_TEST_BENCH_CNT / 2;

      printk(KERN_INFO "%s: n = %d, cnt = %d, min = %d, max = %d, avg = %d\n",
            __FUNCTION__, n, KSOCKET_TEST_BENCH_CNT,
            (int)min, (int)max, (int)avg);

bail_out:
      kfree(lbuf);
      return res;
}

/* Mirrors input to output, used for measuring latencies */
int ksocket_test_bench_mirror(ksocket *ksockp)
{
      int n, tmp;
      char *lbuf;

      if ( !(lbuf = vmalloc(KSOCKET_TEST_BENCH_BUF)))
            return -ENOMEM;

      /* Keep reading until connection termination */
      while ((n = ksocket_read(ksockp, lbuf, 0, KSOCKET_TEST_BENCH_BUF, 0)) > 0)
            if ((tmp = ksocket_test_bench_write(ksockp, lbuf, n)) < 0) {
                  vfree(lbuf);
                  return tmp;
            }

      vfree(lbuf);
      return n;
}

/*
 * Module init and cleanup functions
 */

void __exit ksocket_test_cleanup(void)
{
      int i;

      P_DEBUG("%s: Unloading KSockets test module\n", __FUNCTION__);

#ifdef CONFIG_DEVFS_FS
      P_DEBUG("%s: Un-registering devfs entries\n", __FUNCTION__);
      for (i = 0; i < ksocket_test_devcount; i++)
            if (devices[i].handle)
                  devfs_unregister(devices[i].handle);
#else
      P_DEBUG("%s: Un-registering major number\n", __FUNCTION__);
      if (unregister_chrdev(ksocket_test_major, "ksocket_test") < 0)
            printk(KERN_ERR "%s: Could not unregister major %d\n",
                  __FUNCTION__, ksocket_test_major);
#endif

      /* Release ksocket structures */
```

```
     for (i = 0; i < ksocket_test_devcount; i++)
           if (devices[i].ksockp)
                 ksocket_release(devices[i].ksockp);
}

int __init ksocket_test_init(void)
{
     int i, result;
     char devname[10];

     printk(KERN_INFO "KSockets test module version %s\n", KSOCKET_TEST_VERSION_STRING);

     /* Verify module parameters */
     if (ksocket_test_devcount < 1 || ksocket_test_devcount > KSOCKET_TEST_MAXDEVCOUNT)
{
           printk(KERN_ERR "%s: Invalid number of devices specified\n",
                 __FUNCTION__);
           return -EINVAL;
     }

     /* Register the ksocket-test character device driver */
#ifndef CONFIG_DEVFS_FS
     P_DEBUG("%s: Registering major number\n", __FUNCTION__);
     if ((result = register_chrdev(ksocket_test_major, "ksocket_test",
&ksocket_test_fops)) < 0) {
           printk(KERN_ERR __FUNCTION__ ": Could not get major %d\n",
ksocket_test_major);
           return result;
     } else
           P_DEBUG("%s: Using major %d\n", __FUNCTION__, ksocket_test_major);
#endif

     /* Allocate memory for the device array */
     if ( !(devices = kmalloc(ksocket_test_devcount * sizeof(test_dev), GFP_KERNEL))) {
           result = -ENOMEM;
           goto fail;
     }

     /* Initialize the entries of the device array */
     for (i = 0; i < ksocket_test_devcount; i++) {
           devices[i].handle = NULL;
           devices[i].ksockp = NULL;
           atomic_set(&devices[i].users, 0);
     }

     /* Register each test device with devfs and create the corresponding ksocket */
     for (i = 0; i < ksocket_test_devcount; i++) {
#ifdef CONFIG_DEVFS_FS
           /* devfs entries are created inside the ksocket directory */
           sprintf(devname, "%s/test%d", KSOCKET_DIR_NAME, i);
           devices[i].handle =
                 devfs_register(NULL, devname,
                       DEVFS_FL_DEFAULT,
                       ksocket_test_major, i, S_IFCHR | S_IRUGO | S_IWUGO,
                       &ksocket_test_fops, &devices[i]);
           if (!devices[i].handle) {
                 printk(KERN_ERR "%s: Could not register device %s\n",
                       __FUNCTION__, KSOCKET_TEST_DEV_NAME);
                 result = -EBUSY;
                 goto fail;
           }
#endif
```

```
            if ((result = ksocket_create(&devices[i].ksockp)) < 0)
                  goto fail;
      }

      return 0;
fail:
      /* If module initialization failed, cleanup our mess */
      ksocket_test_cleanup();
      return result;
}

/*
 * Implementation of character device operations
 */

int ksocket_test_ioctl(struct inode *inode, struct file *filp,
        unsigned int cmd, unsigned long arg)
{
      int ret, nonblock;
      int i, n, port;
      struct ioc_remotedata remotedata;
      struct ioc_setstate st;

      /* Packet sizes to be used for latency measurements */
      int latency_sizes[] = { 1, 4, 8, 16, 32, 64, 128, 256 };

      test_dev *devp = (test_dev *) filp->private_data;
      nonblock = filp->f_flags & O_NONBLOCK;

      /*
       * Extract the type and number bitfields, check
       * for appropriate values
       */
      if (_IOC_TYPE(cmd) != KSOCKET_TEST_IOC_MAGIC)
            return -ENOTTY;
      if (_IOC_NR(cmd) > KSOCKET_TEST_IOC_MAXNR)
            return -ENOTTY;

      P_DEBUG("%s: Received ioctl request, cmd = 0x%x\n", __FUNCTION__, cmd);

      switch (cmd) {
            case KSOCKET_TEST_IOC_OPEN:
                  ret = ksocket_open(devp->ksockp, nonblock);
                  break;

            case KSOCKET_TEST_IOC_ACCEPT:
                  ret = ksocket_accept(devp->ksockp, nonblock);
                  break;

            case KSOCKET_TEST_IOC_CLOSE:
                  ret = ksocket_close(devp->ksockp, nonblock);
                  break;

            case KSOCKET_TEST_IOC_BIND:
                  if (get_user(port, (int *)arg)) {
                        ret = -EFAULT;
                        break;
                  }
                  ret = ksocket_bind(devp->ksockp, port, nonblock);
                  break;

            case KSOCKET_TEST_IOC_CONNECT:
```

```
        if (copy_from_user(&remotedata, (struct ioc_remotedata *) arg,
            sizeof(struct ioc_remotedata))) {
              ret = -EFAULT;
              break;
        }

        /* Check if the kernel name actually is null-terminated */
        if (strnlen(remotedata.name, KSOCKET_KERNEL_NAME_LEN + 1) >
            KSOCKET_KERNEL_NAME_LEN) {
              ret = -EINVAL;
              break;
        }

        ret = ksocket_connect(devp->ksockp, remotedata.name,
              remotedata.port, nonblock);
        break;

    case KSOCKET_TEST_IOC_STATUS:
        /* Fill in an ioc_setstate structure */
        spin_lock(&devp->ksockp->slock);
        st.state = devp->ksockp->state;
        st.port = devp->ksockp->port;
        st.remote_port = devp->ksockp->remote_port;
        strcpy(st.remote_name, devp->ksockp->remote_name);
        spin_unlock(&devp->ksockp->slock);

        /* and pass it to userspace */
        if (copy_to_user((struct ioc_setstate *)arg, &st,
            sizeof(struct ioc_setstate))) {
              ret = -EFAULT;
              break;
        }

        ret = 0;
        break;

    case KSOCKET_TEST_IOC_RESET:
        /* There must be no other users of the ksocket */
        if (atomic_read(&devp->users) != 1)
              return -EBUSY;

        /* Make sure that the ksocket is KSOCKET_CLOSED_UNLINKED */
        spin_lock(&devp->ksockp->slock);
        if (devp->ksockp->state != KSOCKET_CLOSED_UNLINKED) {
              spin_unlock(&devp->ksockp->slock);
              ret = -EBUSY;
              break;
        }
        spin_unlock(&devp->ksockp->slock);

        ret = ksocket_reset(devp->ksockp);
        break;

    case KSOCKET_TEST_IOC_BENCH_MIRROR:
        /* Used for measuring latencies, mirrors input to output */
        ret = ksocket_test_bench_mirror(devp->ksockp);
        break;

    case KSOCKET_TEST_IOC_BENCH_LATENCY:
        /* Perform latency measurements */
        n = sizeof(latency_sizes) / sizeof(latency_sizes[0]);
        for (i = 0; i < n; i++) {
```

```
                        ret = ksocket_test_bench_latency(devp->ksockp,
                              latency_sizes[i]);
                        if (ret < 0)
                              break;
                  }
                  break;

            default:
                  P_DEBUG("%s: Ignoring invalid ioctl cmd %d\n", __FUNCTION__, cmd);
                  return -ENOTTY;
      }

      return ret;
}

int ksocket_test_open(struct inode *inode, struct file *filp)
{
#ifndef CONFIG_DEVFS_FS
      /* Set the filp->private_data pointer if not using devfs */

      int minor = MINOR(inode->i_rdev);

      P_DEBUG("%s: Called for minor number %d\n", __FUNCTION__, minor);

      if (minor >= ksocket_test_devcount) {
            P_DEBUG("%s: Invalid request for minor %d\n",
                  __FUNCTION__, minor);
            return -ENODEV;
      }

      /* Set the private_data pointer to the appropriate device */
      filp->private_data = &devices[minor];
#endif
      /* The filp->private_data is already set by devfs */

      atomic_inc(&((test_dev *)filp->private_data)->users);
      MOD_INC_USE_COUNT;
      return 0;               /* Return success */
}

int ksocket_test_release(struct inode *inode, struct file *filp)
{
      atomic_dec(&((test_dev *)filp->private_data)->users);
      MOD_DEC_USE_COUNT;
      return 0;
}

ssize_t ksocket_test_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
      test_dev *devp = (test_dev *) filp->private_data;

      /*
       * Use ksocket_read to read from the corresponding ksocket
       * to the userspace buffer buf
       */
      return ksocket_read(devp->ksockp, buf, 1, count, filp->f_flags & O_NONBLOCK);
}

ssize_t ksocket_test_write(struct file *filp, const char *buf, size_t count, loff_t
*f_pos)
{
      test_dev *devp = (test_dev *) filp->private_data;
```

```
     /*
      * Use ksocket_write to write to the corresponding ksocket
      * from the userspace buffer buf
      */
     return ksocket_write(devp->ksockp, buf, 1, count, filp->f_flags & O_NONBLOCK);
}

/*
 * Module information section
 */

MODULE_AUTHOR("Vangelis Koukis");
MODULE_LICENSE("GPL");

MODULE_PARM(ksocket_test_major, "i");
MODULE_PARM_DESC(ksocket_test_major, "The major number used by the ksocket test
device");

MODULE_PARM(ksocket_test_devcount, "i");
MODULE_PARM_DESC(ksocket_test_devcount, "The required number of ksocket-test devices");

module_init(ksocket_test_init);
module_exit(ksocket_test_cleanup);
```

αρχείο Makefile:

```
############################################################
#                                                          #
# Makefile                                                 #
#                                                          #
# Makefile for the KSockets library                        #
#                                                          #
# Vangelis Koukis, 2002                                    #
#                                                          #
############################################################

CC = gcc

# Find out the directories where the kernel lives

KERNELDIR := /lib/modules/$(shell uname -r)/build
INSTALLDIR := /lib/modules/$(shell uname -r)/kernel/drivers/char

include $(KERNELDIR)/.config

# Get the architecture-specific options to the C compiler
ARCH_CFLAGS := $(shell $(MAKE) -fMakefile.arch show_cflags)
export ARCH_CFLAGS

CFLAGS = $(ARCH_CFLAGS)
CFLAGS += -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include
CFLAGS += -O2 -finline-functions -fexpensive-optimizations -fomit-frame-pointer
CFLAGS += -Wall

# Uncomment following line to enable debugging messages in syslog
# CFLAGS += -DKSOCKET_DEBUG

# Take care of compiling for an SMP enabled kernel
ifdef CONFIG_SMP
     CFLAGS += -D__SMP__ -DSMP
endif

# Prepend modversions.h if compiling for a kernel using module versioning
ifdef CONFIG_MODVERSIONS
     CFLAGS += -DMODVERSIONS -include $(KERNELDIR)/include/linux/modversions.h
endif

OBJS = ksocket-core.o ksocket-test.o

all: $(OBJS) usrmode_make testprog_make

install: $(OBJS)
     install -d $(INSTALLDIR)
     install -c -m 644 $(OBJS) $(INSTALLDIR)
     depmod -a

clean:
     rm -f $(OBJS) *~ core
     make -C usrmode clean
     make -C testprog clean

ksocket-core.o: ksocket-core.c ksocket.h
ksocket-test.o: ksocket-test.c ksocket.h ksocket-test.h
```

```
usrmode_make:
	$(MAKE) -C usrmode

testprog_make:
	$(MAKE) -C testprog
```

αρχείο Makefile.arch:

```
############################################################
#                                                          #
# Makefile.arch                                            #
#                                                          #
# Makefile used to get architecture-specific CFLAGS        #
#                                                          #
# Vangelis Koukis, 2002                                    #
#                                                          #
############################################################

# Output the kernel architecture-specific C compiler options

CC = gcc

.PHONY: show_cflags

show_cflags:
     @echo $(CFLAGS)

# Get architecture information and kernel directory

ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm/ -e
s/sa110/arm/)
KERNELDIR := /lib/modules/$(shell uname -r)/build
INSTALLDIR := /lib/modules/$(shell uname -r)/kernel/drivers/char

# Include kernel configuration and architecture-specific makefile
include $(KERNELDIR)/.config
include $(KERNELDIR)/arch/$(ARCH)/Makefile
```

αρχείο usrmode/usrmode.h:

```
/*
 * usrmode.h
 *
 * Definitions used by the KSockets usermode daemon
 *
 * Vangelis Koukis, 2002
 */

#ifndef _USRMODE_H
#define _USRMODE_H

#include "sisci_api.h"

#define KSOCKET_DAEMON_VERSION     "0.8beta"
#define KSOCKET_DAEMON_LOGFILE     "usrmode.log"

/*
 * Communication over SCI
 */
#if defined(USRMODE_USE_SCI)
#define NO_CALLBACK    NULL
#define NO_FLAGS 0

#define SCI_CONNECT_TMOUT    4000  /* In milliseconds */
#define SCI_DISCONNECT_WAIT  1000  /* In milliseconds */
#define SCI_POLL_TMOUT       750   /* In milliseconds */

#define SCI_MAX_NODE_ID            65536
#define SCI_SEGMENT_SIZE     1000000
#define SCI_ADAPTER_NO       0

#define SCI_READ_FLAG        'R'
#define SCI_WRITE_FLAG       'W'

#define SEG_DATA(segment, offset, type)        \
     *(volatile type *)((segment) + (offset))

/* Offsets of various fields in an SCI segment */
#define SCI_REMOTEID_OFF     0
#define SCI_PORT_OFF         SCI_REMOTEID_OFF + 2
#define SCI_PACKETLEN_OFF    SCI_PORT_OFF + 2
#define SCI_FLAG_OFF         SCI_PACKETLEN_OFF + 4
#define SCI_DATA_OFF         SCI_FLAG_OFF + 1

#define SCI_PACKET_MAX_SIZE  SCI_SEGMENT_SIZE - SCI_DATA_OFF

#define SCI_PIPE_BUF_SIZE    100
#define KERNEL_LIST_FILE     "kernels.sci"

#endif     /* defined(USRMODE_USE_SCI) */

/*
 * Communication over TCP/IP
 */
#if defined(USRMODE_USE_TCPIP)
#define TCPIP_BUF_SIZE       536870
#define KERNEL_LIST_FILE     "kernels.tcpip"
```

```
#endif     /* defined(USRMODE_USE_TCPIP) */

typedef struct uksocket_struct {
     int fd;                                    /* fd used to communicate with
kernelspace */
     ksocket_state_t state;
     char remote_name[KSOCKET_KERNEL_NAME_LEN + 1];  /* Remote kernel name */
     uint16_t port, remote_port;                /* Local, remote port */
     pthread_t thread;                   /* POSIX Thread for data exchange */

#if defined(USRMODE_USE_TCPIP)
     int sockfd;
#elif defined(USRMODE_USE_SCI)
     enum {
           SEGMENT_INVALID,
           SEGMENT_VALID,
           SEGMENT_MAPPED,
           INTR_VALID,
           PIPE_VALID
     } states[2];

     sci_desc_t sd;                 /* SCI virtual device descriptor */
     sci_local_segment_t lsegment; /* Local shared segment descriptor */
     sci_remote_segment_t rsegment;      /* Remote shared segment descriptor */
     sci_local_interrupt_t lintr;  /* Local interrupt descriptor */
     sci_remote_interrupt_t rintr; /* Remote interrupt descriptor */
     sci_map_t lseg_map, rseg_map; /* SCI descriptors of maps to userspace */
     volatile char *lsegp, *rsegp; /* Virtual addresses where segments are mapped */

     sci_segment_cb_reason_t lstate;     /* Segment states */
     sci_segment_cb_reason_t rstate;

     int pipefds[2];                /* Pipe used for callback / main thread communication
*/

     pthread_mutex_t   mutex;            /* Mutex used by main thread and callbacks */
     pthread_cond_t cond;           /* Condition is signaled when the state of a seg
changes */
#endif
} uksocket;

/* Structure to hold a { kernel name, protocol address } pair */
typedef struct name_entry_struct {
     char kernel_name[KSOCKET_KERNEL_NAME_LEN + 1];

#if defined(USRMODE_USE_TCPIP)
     struct in_addr addr;
#elif defined(USRMODE_USE_SCI)
     uint16_t addr;
#endif
} name_entry;

/*
 * Let's define some useful macros
 */
#define min(a,b)       ((a) < (b) ? (a) : (b))
#define STR_EQ(a, b)         (!strcmp((a), (b)))

#ifdef KSOCKET_DEBUG
#define P_DEBUG(stream,arg...)     fprintf(stream,##arg)
#else
#define P_DEBUG(arg...)            do { } while (0)
```

```
#endif

#ifdef KSOCKET_DEBUG
#define ASSERT(condition)                                       \
     if (!(condition))                                  \
           perr(1, 0, "%s:%d: Assertion failed: " # condition,   \
                 __FILE__, __LINE__);
#else
#define ASSERT(condition)    do { } while (0)
#endif

#define PERR_BUF_SIZE        500
#define LINE_BUF_SIZE        1024

#endif     /* _USRMODE_H */
```

αρχείο usrmode/usrmode.c:

```
/*
 * usrmode.c
 *
 * KSockets userspace daemon process
 *
 * Implementation of kernel-to-user communication mechanism
 * Communication over SCI shared segments and TCP/IP sockets
 *
 * Vangelis Koukis, 2002
 */

#include <time.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdarg.h>
#include <unistd.h>
#include <pthread.h>

#include <sys/time.h>
#include <sys/poll.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/socket.h>

#include <arpa/inet.h>
#include <netinet/in.h>

#include "sisci_api.h"

#include "ksocket.h"
#include "usrmode.h"

/*
 * perr:   Main error reporting function
 */
void perr(int fatal, int want_errno, char *fmt, ...)
{
     va_list ap;
     int err_number = errno;       /* We need errno NOW */
     char buf[PERR_BUF_SIZE], *p = buf;

     va_start(ap, fmt);
     p += sprintf(p, fatal ? "Fatal error: " : "Error: ");
     p += sprintf(p, "Thread %ld, LWP %ld: ",
           (long)pthread_self(), (long)getpid());
     p += vsprintf(p, fmt, ap);
     p += sprintf(p, "\n");
     va_end(ap);

     if (want_errno == 1) {
           /* Print last error returned from system call */
           p += sprintf(p, "Errno was: %d - %s\n",
```

```
                err_number, sys_errlist[err_number]);
     } else if (want_errno & SCI_ERR_MASK) {
           /* want_errno is interpreted as an SCI error value */
           /* If the error is SCI_ERR_SYSTEM, then errno must be consulted */
           if (want_errno == SCI_ERR_SYSTEM)
                 p += sprintf(p, "SCI system error: Errno was %d - %s\n",
                       err_number, sys_errlist[err_number]);
           else
                 p += sprintf(p, "SCI Errno was 0x%x\n", (int)want_errno);
     }

     /*
      * Output the buffer to stderr with a single call to fprintf,
      * which is thread-safe and locks the stderr semaphore
      */
     fprintf(stderr, "%s", buf);

     if (fatal)
           exit(1);
}

void credits(void)
{
     fprintf(stderr, "KSockets User Daemon version %s\n", KSOCKET_DAEMON_VERSION);
     fprintf(stderr, "Vangelis Koukis, 2002\n\n");
}

void *xmalloc(size_t size)
{
     void *p;

     if ((p = malloc(size)) == NULL)
           perr(1, 1, "%s: Could not allocate %ld bytes", __FUNCTION__, (long)size);
     return p;
}

/*
 * Setup the communication network to be used by the userspace daemon
 */
#if defined(USRMODE_USE_TCPIP)
     #define protocol_start        tcpip_start
     #define protocol_stop         tcpip_stop
     #define protocol_init         tcpip_init
     #define protocol_destroy      tcpip_destroy
     #define protocol_open         tcpip_open
     #define protocol_close        tcpip_close
     #define protocol_bind         tcpip_bind
     #define protocol_connect      tcpip_connect
     #define protocol_accept       tcpip_accept
     #define protocol_main_loop    tcpip_main_loop
#elif defined(USRMODE_USE_SCI)
     #define protocol_start        sisci_start
     #define protocol_stop         sisci_stop
     #define protocol_init         sisci_init
     #define protocol_destroy      sisci_destroy
     #define protocol_open         sisci_open
     #define protocol_close        sisci_close
     #define protocol_bind         sisci_bind
     #define protocol_connect      sisci_connect
     #define protocol_accept       sisci_accept
     #define protocol_main_loop    sisci_main_loop
#endif
```

```
/*
 * Global data
 */
name_entry *names;
uksocket *ports[KSOCKET_MAX_PORTNR + 1];

/*
 * Function prototypes
 */
static inline name_entry *addr_to_name(void *, name_entry *);

#if defined(USRMODE_USE_SCI)
/*
 * Implementation of communication over SCI
 */

/* Local segment asynchronous callback function */
sci_callback_action_t sisci_lsegment_callback(
     void *arg, sci_local_segment_t lsegment,
     sci_segment_cb_reason_t reason, unsigned int nodeId,
     unsigned int localAdapterNo, sci_error_t status)
{
     sci_error_t sci_error;
     uksocket *uksockp = arg;

     ASSERT(uksockp);
     P_DEBUG(stderr, "%s: Called for lsegment 0x%p, reason %d, nodeId %u\n",
           __FUNCTION__, (void *)lsegment, (int)reason, nodeId);

     /*
      * This callback function is called whenever there
      * is a change in the state of a local segment
      */
     pthread_mutex_lock(&uksockp->mutex);
     switch (reason) {
           case SCI_CB_CONNECT:
                 /*
                  * A new connection has been established.
                  * No new connections to this SCI segment are allowed
                  */
                 SCISetSegmentUnavailable(lsegment, SCI_ADAPTER_NO,
                                          NO_FLAGS, &sci_error);
                 if (sci_error != SCI_ERR_OK)
                       perr(0, sci_error, "%s: SCISetSegmentUnavailable failed",
                             __FUNCTION__);
                 uksockp->lstate = SCI_CB_CONNECT;
                 break;
           case SCI_CB_DISCONNECT:       /* The remote node has disconnected */
           case SCI_CB_LOST:       /* The remote disconnected unexpectedly */
           case SCI_CB_NOT_OPERATIONAL:  /* The link is temporarily not operational */
           case SCI_CB_OPERATIONAL:      /* The link has become operational again */
                 uksockp->lstate = reason;
                 break;
           default:
                 /* Should never happen */
                 perr(0, 0, "%s: Unknown callback reason %d specified",
                       __FUNCTION__, (int)reason);
     }

     /* Signal at most one waiting thread */
     pthread_cond_signal(&uksockp->cond);
```

```
	pthread_mutex_unlock(&uksockp->mutex);
	return SCI_CALLBACK_CONTINUE;
}

/* Remote segment asynchronous callback function */
sci_callback_action_t sisci_rsegment_callback(
	void *arg, sci_remote_segment_t rsegment,
	sci_segment_cb_reason_t reason, sci_error_t status)
{
	uksocket *uksockp = arg;

	ASSERT(uksockp);
	P_DEBUG(stderr, "%s: Called for rsegment 0x%p, reason %d, error status %u\n",
		__FUNCTION__, (void *)rsegment, (int)reason, (unsigned int)status);

	/*
	 * This callback function is called whenever there
	 * is a change in the state of a remote segment
	 */
	pthread_mutex_lock(&uksockp->mutex);
	switch (reason) {
		case SCI_CB_CONNECT:
			/*
			 * This shouldn't happen, as only synchronous connections
			 * to remote segments are requested by other threads
			 */
			perr(0, 0, "%s: Called with SCI_CB_CONNECT for segment 0x%p",
				__FUNCTION__, rsegment);
			break;
		case SCI_CB_DISCONNECT:		/* The remote node has disconnected */
		case SCI_CB_LOST:		/* The remote disconnected unexpectedly */
		case SCI_CB_NOT_OPERATIONAL:	/* The link is temporarily not operational */
			/*
			 * Actually, this is never going to happen: The SCI_FLAG_NOTIFY and
			 * SCI_FLAG_FORCE_DISCONNECT flags are not yet implemented by SISCI
			 */
			uksockp->rstate = SCI_CB_DISCONNECT;
			break;
		case SCI_CB_OPERATIONAL:	/* The link has become operational again */
			/* It is ignored, the connection is already considered broken */
			break;
		default:
			/* Should never happen */
			perr(0, 0, "%s: Unknown callback reason %d specified",
				__FUNCTION__, (int)reason);
	}

	/* Signal at most one waiting thread */
	pthread_cond_signal(&uksockp->cond);
	pthread_mutex_unlock(&uksockp->mutex);
	return SCI_CALLBACK_CONTINUE;
}

/* Local interrupt asynchronous callback function */
sci_callback_action_t sisci_lintr_callback(
	void *arg, sci_local_interrupt_t interrupt, sci_error_t status)
{
	int err;
	char *dummy;
	uksocket *uksockp = arg;

	ASSERT(uksockp);
```

```
    P_DEBUG(stderr, "%s: Called for local interrupt 0x%p, error status %u\n",
          __FUNCTION__, (void *)interrupt, (unsigned int)status);

    /*
     * Write one byte to the pipe used for communication with the
     * main thread, to signal that the remote has triggered the local interrupt
     */
    if ((err = write(uksockp->pipefds[1], &dummy, 1)) < 0)
          perr(0, 1, "%s: write to the pipe fd failed", __FUNCTION__);
    else if (!err)
          perr(0, 1, "%s: write to the pipe fd returned 0?!", __FUNCTION__);

    return SCI_CALLBACK_CONTINUE;
}

static inline unsigned int sisci_segment_id(uint16_t port)
{
    return port << 2;
}

static inline unsigned int sisci_interrupt_no(uint16_t port)
{
    return (port << 2) + 1;
}

static inline int sisci_destroy_local_resources(uksocket *uksockp)
{
    sci_error_t sci_error;

    /*
     * Destroy the allocated local resources, in reverse order.
     * If some of them are still in use, insist until all are freed
     */
    while (uksockp->states[0] != SEGMENT_INVALID) {
          switch (uksockp->states[0]) {
          case PIPE_VALID:
                if (close(uksockp->pipefds[0]) < 0 ||
                    close(uksockp->pipefds[1]) < 0) {
                      perr(0, 1, "%s: Could not close pipe file descriptors",
                            __FUNCTION__);
                      break;
                } else
                      uksockp->states[0] = INTR_VALID;
                /* fallthru */
          case INTR_VALID:
                SCIRemoveInterrupt(uksockp->lintr, NO_FLAGS, &sci_error);
                if (sci_error != SCI_ERR_OK) {
                      perr(0, sci_error, "%s: SCIRemoveInterrupt failed",
                            __FUNCTION__);
                      break;
                } else
                      uksockp->states[0] = SEGMENT_MAPPED;
                /* fallthru */
          case SEGMENT_MAPPED:
                SCIUnmapSegment(uksockp->lseg_map, NO_FLAGS, &sci_error);
                if (sci_error != SCI_ERR_OK) {
                      perr(0, sci_error, "%s: SCIUnmapSegment failed",
                            __FUNCTION__);
                      break;
                } else
                      uksockp->states[0] = SEGMENT_VALID;
                /* fallthru */
```

```
        case SEGMENT_VALID:
                SCIRemoveSegment(uksockp->lsegment, NO_FLAGS, &sci_error);
                if (sci_error != SCI_ERR_OK) {
                        perr(0, sci_error, "%s: SCIRemoveSegment failed",
                                __FUNCTION__);
                        break;
                } else
                        uksockp->states[0] = SEGMENT_INVALID;
                /* fallthru */
        case SEGMENT_INVALID:
                break;
        default:
                /* shouldn't happen */
                perr(1, 0, "%s: Invalid state %d for local resources",
                        __FUNCTION__, uksockp->states[0]);
                break;
        }
        if (uksockp->states[0] != SEGMENT_INVALID)
                P_DEBUG(stderr, "%s: State is still %d, will retry...\n",
                        __FUNCTION__, uksockp->states[0]);
        usleep(SCI_DISCONNECT_WAIT * 1000);
    }

    return 0;
}

static inline int sisci_create_local_resources(uksocket *uksockp, uint16_t port)
{
    int res;
    unsigned int interrupt;
    sci_error_t sci_error;

    ASSERT(uksockp->states[0] == SEGMENT_INVALID);

    /* Create a new SCI local segment... */
    SCICreateSegment(uksockp->sd, &uksockp->lsegment,
                 sisci_segment_id(port), SCI_SEGMENT_SIZE,
                 sisci_lsegment_callback, uksockp,
                 SCI_FLAG_USE_CALLBACK, &sci_error);
    if (sci_error != SCI_ERR_OK) {
        if (sci_error != SCI_ERR_SEGMENTID_USED) {
                perr(0, sci_error, "%s: SCICreateSegment failed", __FUNCTION__);
                res = -EIO;
        } else
                res = -EADDRINUSE;
        goto destroy_before_out;
    }
    uksockp->states[0] = SEGMENT_VALID;

    /* ...map it to a virtual address in userspace... */
    uksockp->lsegp = SCIMapLocalSegment(uksockp->lsegment,
        &uksockp->lseg_map, 0, SCI_SEGMENT_SIZE, NULL,
        NO_FLAGS, &sci_error);
    if (sci_error != SCI_ERR_OK) {
        perr(0, sci_error, "%s: SCIMapLocalSegment failed", __FUNCTION__);
        res = -EIO;
        goto destroy_before_out;
    }
    uksockp->states[0] = SEGMENT_MAPPED;

    /* Initialize local segment fields */
    SEG_DATA(uksockp->lsegp, SCI_REMOTEID_OFF, uint16_t) = 0;
```

```
    SEG_DATA(uksockp->lsegp, SCI_PORT_OFF, uint16_t) = 0;
    SEG_DATA(uksockp->lsegp, SCI_PACKETLEN_OFF, uint32_t) = 0;
    SEG_DATA(uksockp->lsegp, SCI_FLAG_OFF, char) = SCI_WRITE_FLAG;

    /* ...and prepare it for use by the local SCI adapter */
    SCIPrepareSegment(uksockp->lsegment, SCI_ADAPTER_NO,
                NO_FLAGS, &sci_error);
    if (sci_error != SCI_ERR_OK) {
          perr(0, sci_error, "%s: SCIPrepareSegment failed", __FUNCTION__);
          res = -EIO;
          goto destroy_before_out;
    }

    /* Create a new local interrupt */
    interrupt = sisci_interrupt_no(port);
    SCICreateInterrupt(uksockp->sd, &uksockp->lintr, SCI_ADAPTER_NO,
                &interrupt, sisci_lintr_callback, uksockp,
                SCI_FLAG_FIXED_INTNO | SCI_FLAG_USE_CALLBACK, &sci_error);
    if (sci_error != SCI_ERR_OK) {
          perr(0, sci_error, "%s: SCICreateInterrupt failed", __FUNCTION__);
          res = -EIO;
          goto destroy_before_out;
    }
    uksockp->states[0] = INTR_VALID;

    /* Create a pipe fd pair, for callback / main thread communication */
    if (pipe(uksockp->pipefds) < 0) {
          res = -errno;
          perr(0, 1, "%s: Pipe creation failed", __FUNCTION__);
          goto destroy_before_out;
    }
    uksockp->states[0] = PIPE_VALID;

    return 0;

destroy_before_out:
    /* An error has occured, destroy the resources already created */
    sisci_destroy_local_resources(uksockp);
    return res;
}

static inline int sisci_disconnect_remote_resources(uksocket *uksockp)
{
    sci_error_t sci_error;

    P_DEBUG(stderr, "%s: Called for uksockp 0x%p\n", __FUNCTION__, uksockp);

    /* Disconnect from remote resources, in reverse order */
    switch (uksockp->states[1]) {
          case INTR_VALID:
                SCIDisconnectInterrupt(uksockp->rintr, NO_FLAGS, &sci_error);
                if (sci_error != SCI_ERR_OK)
                      perr(0, sci_error, "%s: SCIDisconnectInterrupt failed",
                            __FUNCTION__);
                /* fallthru */
          case SEGMENT_MAPPED:
                SCIUnmapSegment(uksockp->rseg_map, NO_FLAGS, &sci_error);
                if (sci_error != SCI_ERR_OK)
                      perr(0, sci_error, "%s: SCIUnmapSegment failed",
                            __FUNCTION__);
                /* fallthru */
          case SEGMENT_VALID:
```

```
                SCIDisconnectSegment(uksockp->rsegment, NO_FLAGS, &sci_error);
                if (sci_error != SCI_ERR_OK)
                        perr(0, sci_error, "%s: SCIDisconnectSegment failed",
                                __FUNCTION__);
                /* fallthru */
        case SEGMENT_INVALID:
                break;
        default:
                /* shouldn't happen */
                perr(1, 0, "%s: Invalid state %d for remote resources",
                        __FUNCTION__, uksockp->states[1]);
    }

    uksockp->states[1] = SEGMENT_INVALID;
    return 0;
}

static inline int sisci_connect_remote_resources(uksocket *uksockp, int nodeId, uint16_t
port)
{
    int res;
    sci_error_t sci_error;

    ASSERT(uksockp->states[1] == SEGMENT_INVALID);
    P_DEBUG(stderr, "%s: Called for %d: %d\n", __FUNCTION__, nodeId, port);

    /* Connect to the remote segment... */
    SCIConnectSegment(uksockp->sd, &uksockp->rsegment,
                  nodeId, sisci_segment_id(port), SCI_ADAPTER_NO,
                  sisci_rsegment_callback, uksockp, SCI_CONNECT_TMOUT,
                  SCI_FLAG_USE_CALLBACK, &sci_error);
    if (sci_error != SCI_ERR_OK) {
        if (sci_error != SCI_ERR_CONNECTION_REFUSED &&
            sci_error != SCI_ERR_NO_SUCH_SEGMENT) {
                perr(0, sci_error, "%s: Could not connect to SCI segment %d:%d\n",
                        __FUNCTION__, nodeId, sisci_segment_id(port));
                res = -EIO;
        } else
                res = -ECONNREFUSED;
        goto destroy_before_out;
    }
    uksockp->rstate = SCI_CB_CONNECT;   /* Synch. connect, the callback was not called
*/
    uksockp->states[1] = SEGMENT_VALID;

    /* ...and map it to userspace */
    uksockp->rsegp = SCIMapRemoteSegment(uksockp->rsegment,
        &uksockp->rseg_map, 0, SCI_SEGMENT_SIZE, NULL,
        NO_FLAGS, &sci_error);
    if (sci_error != SCI_ERR_OK) {
        perr(0, sci_error, "%s: SCIMapRemoteSegment failed", __FUNCTION__);
        res = -EIO;
        goto destroy_before_out;
    }
    uksockp->states[1] = SEGMENT_MAPPED;

    /* Connect to the remote interrupt */
    SCIConnectInterrupt(uksockp->sd, &uksockp->rintr,
                    nodeId, SCI_ADAPTER_NO, sisci_interrupt_no(port),
                    SCI_CONNECT_TMOUT, NO_FLAGS, &sci_error);
    if (sci_error != SCI_ERR_OK) {
        perr(0, sci_error, "%s: SCIConnectInterrupt failed", __FUNCTION__);
```

```
            res = -EIO;
            goto destroy_before_out;
      }
      uksockp->states[1] = INTR_VALID;

      return 0;

destroy_before_out:
      /* An error has occured, disconnect from remote resources */
      sisci_disconnect_remote_resources(uksockp);
      return res;
}

static inline int sisci_waitfor_lintr_trigger(uksocket *uksockp, int tmout_msec)
{
      int err;
      struct pollfd pollfds[1];
      char dummy[SCI_PIPE_BUF_SIZE];

      /*
       * Poll the pipe used by the callback function, until the
       * local interrupt has been triggered, or tmout_msecs have passed.
       * A tmout_msec value of -1 means infinite timeout
       */
      pollfds[0].fd = uksockp->pipefds[0];
      pollfds[0].events = POLLIN;

      if ((err = poll(pollfds, 1, tmout_msec)) < 0) {
            err = -errno;
            perr(0, 1, "%s: Could not poll for fd %d", __FUNCTION__, pollfds[0].fd);
            return err;
      } else if (!err)
            return -ETIMEDOUT;
      else {
            /* An interrupt has been triggered, flush the pipe */
            if ((err = read(pollfds[0].fd, dummy, sizeof(dummy))) < 0)
                  perr(0, 1, "%s: Read from pipe failed", __FUNCTION__);
            else if (!err)
                  perr(0, 0, "%s: Read from pipe return EOF?!", __FUNCTION__);
      }

      return 0;
}

static inline int sisci_waitfor_lsegment_connect(uksocket *uksockp, int tmout_msec)
{
      int err = 0;
      struct timeval now;
      struct timespec tmout;

      P_DEBUG(stderr, "%s: Waiting for uksocket 0x%p to connect...\n",
            __FUNCTION__, uksockp);

      /*
       * Wait until a connection to the local segment has been established,
       * or tmout_msec have passed. If tmout_msec is < 0, it is ignored
       */
      pthread_mutex_lock(&uksockp->mutex);
      if (tmout_msec >= 0) {
            gettimeofday(&now, NULL);
            tmout.tv_sec = now.tv_sec + SCI_CONNECT_TMOUT;
            tmout.tv_nsec = now.tv_usec * 1000;
```

```
            while (!err && uksockp->lstate != SCI_CB_CONNECT)
                  err = pthread_cond_timedwait(&uksockp->cond,&uksockp->mutex,&tmout);
      } else
            while (uksockp->lstate != SCI_CB_CONNECT)
                  pthread_cond_wait(&uksockp->cond, &uksockp->mutex);
      pthread_mutex_unlock(&uksockp->mutex);

      P_DEBUG(stderr, "%s: Returning for uksocket 0x%p, err = %d\n",
            __FUNCTION__, uksockp, err);

      if (err)
            return (err == ETIMEDOUT) ? -ETIMEDOUT : -EIO;
      else
            return 0;
}

static inline int sisci_start(void)
{
      sci_error_t sci_error;

      /* Initialize the SISCI library */
      SCIInitialize(NO_FLAGS, &sci_error);
      if (sci_error != SCI_ERR_OK)
            perr(0, sci_error, "%s: Initialization of SISCI library failed",
                  __FUNCTION__);

      return (sci_error == SCI_ERR_OK) ? 0 : -1;
}

static inline int sisci_stop(void)
{
      SCITerminate();
      return 0;
}

static inline void sisci_init(uksocket *uksockp)
{
      int err;

      uksockp->lsegp = uksockp->rsegp = NULL;
      uksockp->lstate = uksockp->rstate = SCI_CB_DISCONNECT;
      uksockp->states[0] = uksockp->states[1] = SEGMENT_INVALID;

      if ((err = pthread_mutex_init(&uksockp->mutex, NULL))) {
            errno = err;
            perr(1, 1, "%s: Mutex initialization failed", __FUNCTION__);
      }

      if ((err = pthread_cond_init(&uksockp->cond, NULL))) {
            errno = err;
            perr(1, 1, "%s: Condition initialization failed", __FUNCTION__);
      }
}

static inline void sisci_destroy(uksocket *uksockp)
{
      int err;

      if ((err = pthread_mutex_destroy(&uksockp->mutex))) {
            errno = err;
            perr(0, 1, "%s: Mutex could not be destroyed", __FUNCTION__);
      }
```

```
     if ((err = pthread_cond_destroy(&uksockp->cond))) {
           errno = err;
           perr(0, 1, "%s: Condition could not be destroyed", __FUNCTION__);
     }
}

static inline int sisci_open(uksocket *uksockp)
{
     sci_error_t sci_error;

     /* Open an SCI virtual device */
     /* SCI_FLAG_THREAD_SAFE not yet implemented by SISCI... */
     SCIOpen(&uksockp->sd, NO_FLAGS, &sci_error);
     if (sci_error != SCI_ERR_OK)
           perr(0, sci_error, "%s: Could not open SCI virtual device",
                 __FUNCTION__);

     return (sci_error == SCI_ERR_OK) ? 0: -EIO;
}

static inline int sisci_bind(uksocket *uksockp, uint16_t port)
{
     int err;

     /* Allocate the necessary local SCI resources */
     err = sisci_create_local_resources(uksockp, port);

     if (!err)
           P_DEBUG(stderr, "%s: uksocket at 0x%p bound to segment %d\n",
                 __FUNCTION__, uksockp, sisci_segment_id(port));
     return err;
}

static inline int sisci_close(uksocket *uksockp)
{
     sci_error_t sci_error;

     /*
      * This function is called only when the state is
      * KSOCKET_BOUND or KSOCKET_OPEN
      */

     /* Deallocate the local SCI resources associated with this ksocket */
     if (uksockp->state == KSOCKET_BOUND)
           sisci_destroy_local_resources(uksockp);

     /* Close the SCI virtual device */
     SCIClose(uksockp->sd, NO_FLAGS, &sci_error);
     if (sci_error != SCI_ERR_OK)
           perr(0, sci_error, "%s: Failed to close SCI virtual device",
                 __FUNCTION__);

     return 0;
}

static inline int sisci_connect(uksocket *uksockp, name_entry *np)
{
     int res;
     unsigned int nodeId;
     sci_error_t sci_error;
     sci_query_adapter_t sq;
```

```
	/* Find out our local SCI node id */
	sq.localAdapterNo = SCI_ADAPTER_NO;
	sq.subcommand = SCI_Q_ADAPTER_NODEID;
	sq.data = &nodeId;
	SCIQuery(SCI_Q_ADAPTER, &sq, NO_FLAGS, &sci_error);
	if (sci_error != SCI_ERR_OK) {
		perr(0, sci_error, "%s: SCIQuery failed", __FUNCTION__);
		return -EIO;
	}

	ASSERT(uksockp->states[0] == PIPE_VALID);

	/* Step 1. Make the local segment available, so that the remote can connect */
	SCISetSegmentAvailable(uksockp->lsegment, SCI_ADAPTER_NO,
			       NO_FLAGS, &sci_error);
	if (sci_error != SCI_ERR_OK) {
		perr(0, sci_error, "%s: SCISetSegmentAvailable failed",
			__FUNCTION__);
		return -EIO;
	}

	/* Step 2. Connect to remote segment */
	res = sisci_connect_remote_resources(uksockp, np->addr, uksockp->remote_port);
	if (res < 0)
		goto set_unavailable;

	/* Step 3. Send local identification information to remote */
	SEG_DATA(uksockp->rsegp, SCI_REMOTEID_OFF, uint16_t) = htons((uint16_t) nodeId);
	SEG_DATA(uksockp->rsegp, SCI_PORT_OFF, uint16_t) = htons(uksockp->port);

	/* Step 4. Data transferred, trigger the remote interrupt */
	SCITriggerInterrupt(uksockp->rintr, NO_FLAGS, &sci_error);
	if (sci_error != SCI_ERR_OK) {
		perr(0, sci_error, "%s: SCITriggerInterrupt failed", __FUNCTION__);
		sisci_disconnect_remote_resources(uksockp);
		res = -EIO;
		goto set_unavailable;
	}

	/* Step 5. Wait until the remote has connected to the local segment */
	if ((res = sisci_waitfor_lsegment_connect(uksockp, SCI_CONNECT_TMOUT)) < 0) {
		sisci_disconnect_remote_resources(uksockp);
		goto set_unavailable;
	}

	/* The connection has been established, return success */
	return 0;

set_unavailable:
	/* An error has occured, make the local SCI segment unavailable */
	P_DEBUG(stderr, "%s: Connect unsuccessful, res = %d\n",
		__FUNCTION__, res);
	/* Flag SCI_FLAG_FORCE_DISCONNECT is not yet implemented by SISCI... */
	SCISetSegmentUnavailable(uksockp->lsegment,
		SCI_ADAPTER_NO, NO_FLAGS, &sci_error);
	if (sci_error != SCI_ERR_OK)
		perr(0, sci_error, "%s: SCISetSegmentUnavalailable failed",
			__FUNCTION__);
	return res;
}
```

```
static inline int sisci_accept(uksocket *uksockp)
{
     int res;
     name_entry *p;
     uint16_t rport;
     unsigned int nodeId;
     sci_error_t sci_error;

     ASSERT(uksockp->states[0] == PIPE_VALID);

     /* Step 1. Make the local segment available, so that the remote can connect */
     SCISetSegmentAvailable(uksockp->lsegment, SCI_ADAPTER_NO,
                         NO_FLAGS, &sci_error);
     if (sci_error != SCI_ERR_OK) {
           perr(0, sci_error, "%s: SCISetSegmentAvailable failed",
                 __FUNCTION__);
           return -EIO;
     }

     /* Step 2. Wait until a connection has been established */
     if ((res = sisci_waitfor_lsegment_connect(uksockp, -1)))
           goto set_unavailable;
     P_DEBUG(stderr, "%s: Connection to local segment established\n", __FUNCTION__);

     /* Step 3. Wait until an interrupt is triggered by the remote */
     if ((res = sisci_waitfor_lintr_trigger(uksockp, SCI_CONNECT_TMOUT)) < 0)
           goto set_unavailable;
     P_DEBUG(stderr, "%s: Local interrupt triggered\n", __FUNCTION__);

     /* Step 4. Retrieve remote identification information */
     nodeId = ntohs(SEG_DATA(uksockp->lsegp, SCI_REMOTEID_OFF, uint16_t));
     rport = ntohs(SEG_DATA(uksockp->lsegp, SCI_PORT_OFF, uint16_t));

     P_DEBUG(stderr, "%s: Remote id info { %d, %d } retrieved\n",
           __FUNCTION__, nodeId, rport);

     /* Step 5. Establish connection to remote */
     res = sisci_connect_remote_resources(uksockp, nodeId, rport);
     if (res < 0)
           goto set_unavailable;

     /* The connection has been established, return success */
     if ( !(p = addr_to_name(&nodeId, names)))
           strcpy(uksockp->remote_name, "unknown");
     else
           strcpy(uksockp->remote_name, p->kernel_name);
     uksockp->remote_port = rport;
     P_DEBUG(stderr, "%s: After reverse lookup, remote is %s:%d\n",
           __FUNCTION__, uksockp->remote_name, uksockp->remote_port);

     return 0;

set_unavailable:
     /* An error has occured, make the local SCI segment unavailable */

     /* Flag SCI_FLAG_FORCE_DISCONNECT is not yet implemented by SISCI... */
     SCISetSegmentUnavailable(uksockp->lsegment,
           SCI_ADAPTER_NO, NO_FLAGS, &sci_error);
     if (sci_error != SCI_ERR_OK)
           perr(0, sci_error, "%s: SCISetSegmentUnavalailable failed",
                 __FUNCTION__);
     return res;
```

```
}

static inline int can_read_from_lseg(uksocket *uksockp)
{
     /* Checks if there is data to read from the local SCI segment */
     char flag = SEG_DATA(uksockp->lsegp, SCI_FLAG_OFF,  char);

     ASSERT(flag == SCI_READ_FLAG || flag == SCI_WRITE_FLAG);
     return (flag == SCI_READ_FLAG);
}

static inline int can_write_to_rseg(uksocket *uksockp)
{
     /* Checks if data can be written to remote SCI segment */
     char flag = SEG_DATA(uksockp->rsegp, SCI_FLAG_OFF, char);

     ASSERT(flag == SCI_READ_FLAG || flag == SCI_WRITE_FLAG);
     return (flag == SCI_WRITE_FLAG);
}

static inline int sisci_main_loop(uksocket *uksockp)
{
     int res;
     unsigned int remcount;
     struct pollfd pollfds[2];
     char dummy[SCI_PIPE_BUF_SIZE];
     volatile char *lseg_readp, *rseg_writep;
     sci_error_t sci_error;

     P_DEBUG(stderr, "%s: Entering main data transfer loop\n"
                 "\tuksockp = 0x%p\n\tfd = %d\n\tlsegp = 0x%p\n\trsegp = 0x%p\n",
                 __FUNCTION__, uksockp, uksockp->fd, uksockp->lsegp, uksockp->rsegp);

     /* Initialization */
     pollfds[0].fd = uksockp->fd;        /* fd to communicate with the local kernel */
     pollfds[1].fd = uksockp->pipefds[0];      /* fd to wake up on SCI interrupt */

     remcount = 0;                       /* Remaining number of bytes in local segment
*/
     lseg_readp = uksockp->lsegp + SCI_DATA_OFF;
     rseg_writep = uksockp->rsegp + SCI_DATA_OFF;

     /* Main data transfer loop */
     for (;;) {
           /* Setup pollfd structures */
           pollfds[1].events = POLLIN;
           pollfds[0].events = 0;
           if (can_read_from_lseg(uksockp))
                 pollfds[0].events |= POLLOUT;
           if (can_write_to_rseg(uksockp))
                 pollfds[0].events |= POLLIN;

           if (poll(pollfds, 2, SCI_POLL_TMOUT) < 0) {
                 if (errno == EINTR)
                       continue;
                 else
                       perr(0, 1, "%s: poll failed for fdset { %d, %d }",
                             __FUNCTION__, pollfds[0].fd, pollfds[1].fd);
           }

           /* If the local SCI interrupt was triggered, flush the pipe */
           if (pollfds[1].revents & POLLIN) {
```

```
        if ((res = read(pollfds[1].fd, dummy, sizeof(dummy))) < 0)
                perr(0, 1, "%s: read from pipe failed", __FUNCTION__);
        else if (!res)
                perr(0, 0, "%s: read from pipe returned 0?!", __FUNCTION__);
}

pthread_mutex_lock(&uksockp->mutex);
/* If the link is temporarily not operational */
while (uksockp->lstate == SCI_CB_NOT_OPERATIONAL)
        pthread_cond_wait(&uksockp->cond, &uksockp->mutex);
if (uksockp->lstate!=SCI_CB_CONNECT && uksockp->lstate!=SCI_CB_OPERATIONAL)
        goto remote_disconnect;
pthread_mutex_unlock(&uksockp->mutex);

if (pollfds[0].revents & POLLHUP)
        goto local_disconnect;

/* If we can write to the remote SCI segment */
if ((pollfds[0].revents & POLLIN) && can_write_to_rseg(uksockp)) {
        /* Write packet data to remote segment */
        res = read(pollfds[0].fd, (void *)rseg_writep, SCI_PACKET_MAX_SIZE);
        if (res <= 0) {
                if (res < 0)
                        perr(0, 1, "%s: read from kernel fd failed",
                                __FUNCTION__);

                /* Local kernel disconnected or an error occured */
                goto local_disconnect;
        }

        /* Set the message length and flag, then trigger interrupt */
        SEG_DATA(uksockp->rsegp, SCI_PACKETLEN_OFF, uint32_t) = htonl(res);
        SEG_DATA(uksockp->rsegp, SCI_FLAG_OFF, char) = SCI_READ_FLAG;
        SCITriggerInterrupt(uksockp->rintr, NO_FLAGS, &sci_error);
        if (sci_error != SCI_ERR_OK)
                goto remote_disconnect;
}

/* If we can read from the local SCI segment */
if ((pollfds[0].revents & POLLOUT) && can_read_from_lseg(uksockp)) {
        if (!remcount) {
                /* Reset the read pointer if a new packet was received */
                remcount = SEG_DATA(uksockp->lsegp,SCI_PACKETLEN_OFF,uint32_t);
                remcount = ntohl(remcount);
                lseg_readp = uksockp->lsegp + SCI_DATA_OFF;
        }
        if (!remcount)
                perr(0, 0, "%s: Zero-sized packet received", __FUNCTION__);

        res = write(pollfds[0].fd, (void *)lseg_readp, remcount);
        if (res < 0) {
                if (errno != EPIPE)
                        perr(0, 0, "%s: write to kernel failed", __FUNCTION__);
                goto remote_disconnect;
        } else if (!res)
                perr(0, 0, "%s: write to kernel fd returned 0", __FUNCTION__);

        remcount -= res;
        lseg_readp += res;

        /* If all of the packet has been read, signal the remote */
        if (!remcount) {
```

```
                        SEG_DATA(uksockp->lsegp, SCI_FLAG_OFF, char) = SCI_WRITE_FLAG;
                        SCITriggerInterrupt(uksockp->rintr, NO_FLAGS, &sci_error);
                        if (sci_error != SCI_ERR_OK)
                              goto remote_disconnect;
                  }
            }
      }

remote_disconnect:
      P_DEBUG(stderr, "%s: remote_disconnect reached for uksockp = 0x%p, fd = %d\n",
            __FUNCTION__, uksockp, uksockp->fd);

      /* An error has occured, or the remote has disconnected */
      /* Flush any data waiting in the local segment to the kernel fd */
      if (can_read_from_lseg(uksockp))
            while (remcount) {
                  res = write(uksockp->fd, (void *)lseg_readp, remcount);
                  if (res < 0) {
                        if (errno != EPIPE)
                              perr(0, 1, "%s: write to kernel failed", __FUNCTION__);
                        break;
                  } else if (!res)
                        perr(0, 0, "%s: write to kernel returned 0", __FUNCTION__);

                  remcount -= res;
                  lseg_readp += res;
            }

local_disconnect:
      P_DEBUG(stderr, "%s: local_disconnect reached for uksockp = 0x%p, fd = %d\n",
            __FUNCTION__, uksockp, uksockp->fd);

      /* Close the fd to the local kernel and free SCI resources */
      if (close(uksockp->fd) < 0)
            perr(0, 1, "%s: Could not close kernel fd", __FUNCTION__);

      sisci_disconnect_remote_resources(uksockp);
      sisci_destroy_local_resources(uksockp);

      return 0;
}
#endif      /* defined(USRMODE_USE_SCI) */

#if defined(USRMODE_USE_TCPIP)
/*
 * Implementation of circular buffers
 */

typedef struct circ_buf_struct {
      int size;                  /* Size in bytes */
      char *start, *end;               /* Pointers to start / end of buffer */
      char *rp, *wp;                   /* Read and write pointers in buffer */
} circ_buf;

static inline void buf_init(circ_buf *bufp, int size)
{
      /* Allocate memory for buffer, initialize data pointers */
      bufp->start = xmalloc(size);
      bufp->size = size;
      bufp->end = bufp->start + size;
      bufp->rp = bufp->wp = bufp->start;
}
```

```
static inline void buf_destroy(circ_buf *bufp)
{
     /* Free memory allocated for buffer data */
     free(bufp->start);
}

static inline int buf_free(circ_buf *bufp)
{
     /* Returns the amount of available space in buffer */
     /* The buffer is never filled completely, maximum free space is size - 1 */
     if (bufp->rp == bufp->wp)
           return bufp->size - 1;
     return ((bufp->rp + bufp->size - bufp->wp) % bufp->size) - 1;
}

static inline int buf_empty(circ_buf *bufp)
{
     /* Returns true if the circular buffer contains no data */
     return bufp->rp == bufp->wp;
}

static inline int buf_read_count(circ_buf *bufp)
{
     /* Returns the amount of data that can be read without wrapping rp */
     return ((bufp->wp >= bufp->rp) ? bufp->wp : bufp->end) - bufp->rp;
}

static inline int buf_write_count(circ_buf *bufp)
{
     /* Returns the amount of data that can be written without wrapping wp */
     if (bufp->wp >= bufp->rp)
           return min(bufp->end - bufp->wp, buf_free(bufp));
     else
           return bufp->rp - bufp->wp - 1;
}

static inline void buf_advance_pointer(circ_buf *bufp, char **p, int num)
{
     /* Advance the char pointer by num bytes in the circular buffer bufp */
     *p += num;
     if (*p >= bufp->end)
           *p -= bufp->size;
}

/*
 * Implementation of communication over TCP/IP
 */
static inline int tcpip_tcp_port(uint16_t port)
{
     return 30000 + port;
}

static inline int tcpip_ksocket_port(int tcp_port)
{
     return tcp_port - 30000;
}

static inline int tcpip_start(void)
{
     return 0;
}
```

```
static inline int tcpip_stop(void)
{
     return 0;
}

static inline void tcpip_init(uksocket *uksockp)
{
     uksockp->sockfd = 0;
}

static inline void tcpip_destroy(uksocket *uksockp)
{
     return;
}

static inline int tcpip_open(uksocket *uksockp)
{
     int fd, err;

     if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
           err = -errno;           /* Get the value of errno now */
           perr(0, 1, "%s: TCP/IP socket creation failed", __FUNCTION__);
           return err;
     }

     uksockp->sockfd = fd;
     return 0;
}

static inline int tcpip_bind(uksocket *uksockp, uint16_t port)
{
     int err;
     struct sockaddr_in sa;

     sa.sin_family = AF_INET;
     sa.sin_port = htons(tcpip_tcp_port(port));
     sa.sin_addr.s_addr = INADDR_ANY;

     if ((err = bind(uksockp->sockfd, (struct sockaddr *)&sa, sizeof(sa))) < 0) {
           err = -errno;
           /* Anything else than EADDRINUSE (port busy) must be reported */
           if (errno != EADDRINUSE)
                 perr(0, 1, "%s: TCP/IP bind operation failed with errno != EADDRINUSE",
                       __FUNCTION__);
     }

     return err;
}

static inline int tcpip_connect(uksocket *uksockp, name_entry *np)
{
     struct sockaddr_in sa;

     sa.sin_family = AF_INET;
     sa.sin_addr = np->addr;
     sa.sin_port = htons(tcpip_tcp_port(uksockp->remote_port));

     if (connect(uksockp->sockfd, (struct sockaddr *)&sa, sizeof(sa)) < 0)
           return -errno;
     else
           return 0;
```

```
}

static inline int tcpip_accept(uksocket *uksockp)
{
     name_entry *p;
     int newfd, size;
     struct sockaddr_in sa;

     /* Listen for connections on this socket */
     if (listen(uksockp->sockfd, 1) < 0)
           return -errno;

     /* This will block until a new connection is accepted */
     size = sizeof(sa);
     if ((newfd = accept(uksockp->sockfd, (struct sockaddr *)&sa, &size)) < 0)
           return -errno;

     /* Close the original fd, since no new connections will be accepted */
     if (close(uksockp->sockfd) < 0)
           perr(0, 1, "%s: Could not close original fd after accept()",
                 __FUNCTION__);

     uksockp->sockfd = newfd;

     /* Get remote identification information */
     size = sizeof(sa);
     if (getpeername(uksockp->sockfd, (struct sockaddr *)&sa, &size) < 0) {
           perr(0, 1, "%s: getpeername failed for fd %d",
                 __FUNCTION__, uksockp->sockfd);
           strcpy(uksockp->remote_name, "unknown");
           uksockp->remote_port = 0;
     } else {
           if ( !(p = addr_to_name(&sa.sin_addr, names))) {
                 strcpy(uksockp->remote_name, "unknown");
                 uksockp->remote_port = 0;
           } else {
                 strcpy(uksockp->remote_name, p->kernel_name);
                 uksockp->remote_port = tcpip_ksocket_port(ntohs(sa.sin_port));
           }
     }
     P_DEBUG(stderr, "%s: After reverse lookup, remote is %s:%d\n",
           __FUNCTION__, uksockp->remote_name, uksockp->remote_port);

     return 0;
}

static inline void tcpip_close(uksocket *uksockp)
{
     if (close(uksockp->sockfd) < 0)
           perr(0, 1, "%s: Could not close socket file descriptor",
                 __FUNCTION__);
}

static inline int tcpip_main_loop(uksocket *uksockp)
{
     int i, n, can_write[2];
     circ_buf bufs[2];       /* I/O buffers */
     struct pollfd pollfds[2];

     P_DEBUG(stderr, "%s: Entering main data transfer loop\n"
                 "\tuksockp = 0x%p,\n\tfd = %d,\n\tsockfd = %d\n",
                 __FUNCTION__, uksockp, uksockp->fd, uksockp->sockfd);
```

```
	/* Initialize poll structures and circular buffers */
	pollfds[0].fd = uksockp->fd;
	pollfds[1].fd = uksockp->sockfd;
	buf_init(&bufs[0], TCPIP_BUF_SIZE);
	buf_init(&bufs[1], TCPIP_BUF_SIZE);
	can_write[0] = can_write[1] = 1;

	for (;;) {
		/* Setup pollfd structures */
		pollfds[0].events = pollfds[1].events = 0;
		for (i = 0; i <= 1; i++) {
			/* If data can be read into the buffer */
			if (buf_free(&bufs[i]))
				pollfds[i].events |= POLLIN;
			/* If data can be written from the buffer */
			if (!buf_empty(&bufs[i]) && can_write[1 - i])
				pollfds[1 - i].events |= POLLOUT;
		}

		if (poll(pollfds, 2, -1) < 0) {
			if (errno == EINTR)
				continue;
			else
				perr(0, 1, "%s: Could not poll for fd set {%d, %d}",
					__FUNCTION__, uksockp->fd, uksockp->fd);
		}

		/* Perform data exchange between the two file descriptors */
		for (i = 0; i <= 1; i++) {
			/* If there is any new data or the remote has disconnected */
			if (pollfds[i].revents & POLLIN ||
			    pollfds[i].revents & POLLHUP) {
				n = read(pollfds[i].fd, bufs[i].wp, buf_write_count(&bufs[i]));
				if (n > 0)
					buf_advance_pointer(&bufs[i], &bufs[i].wp, n);
				if (n <= 0) {
					if (n < 0)
						perr(0, 1, "%s: read from fd %d failed",
							__FUNCTION__, pollfds[i].fd);

					/* Disconnected or an error occured, flush buffers */
					while (!buf_empty(&bufs[i])) {
						n = write(pollfds[1 - i].fd, bufs[i].rp,
							  buf_read_count(&bufs[i]));
						if (n > 0)
							buf_advance_pointer(&bufs[i], &bufs[i].rp, n);
						if (!n) {
							perr(0, 0, "%s: write on fd %d returned 0!",
								__FUNCTION__, pollfds[i].fd);
							break;
						}
						if (n < 0) {
							perr(0, 1, "%s: write on fd %d failed",
								__FUNCTION__, pollfds[i].fd);
							break;
						}
					}

					/* Closing the ksocket fd sets the ksocket state to
UNLINKED */
					if (close(pollfds[i].fd) < 0)
```

```
                                    perr(0, 1, "%s: Could not close fd %d\n",
                                          __FUNCTION__, pollfds[i].fd);
                              if (close(pollfds[1 - i].fd) < 0)
                                    perr(0, 1, "%s: Could not close fd %d\n",
                                          __FUNCTION__, pollfds[1 - i].fd);

                              P_DEBUG(stderr, "%s: Leaving main data transfer loop\n",
                                    __FUNCTION__);
                              return 0;
                        }
                  }

                  /* To minimize latency, first check if we can write to the other fd */
                  if ((pollfds[1 - i].revents & POLLOUT) && can_write[1 - i]) {
                        n = write(pollfds[1 - i].fd, bufs[i].rp,
buf_read_count(&bufs[i]));
                        if (n > 0)
                              buf_advance_pointer(&bufs[i], &bufs[i].rp, n);
                        else if (n < 0) {
                              /*
                               * The connection has probably been closed (EPIPE),
                               * but we let the read() section above detect and handle
it/
                               */
                              if (errno != EPIPE)
                                    perr(0, 1, "%s: write to fd %d failed",
                                          __FUNCTION__, pollfds[1 - i].fd);
                              can_write[1 - i] = 0;
                        } else
                              perr(0, 0, "%s: write to fd %d returned 0!",
                                    __FUNCTION__, pollfds[1 - i].fd);
                  }
            }
      }
}

#endif      /* defined(USRMODE_USE_TCPIP) */

/*
 * Implementation of kernelspace - userspace communication
 * Functions that process requests from userspace
 */

static inline name_entry *name_to_addr(char *name, name_entry *names)
{
      /* The table ends with a blank entry */
      while (names->kernel_name[0] != '\0' && !STR_EQ(names->kernel_name, name))
            ++names;

      /* Return NULL if the lookup was unsuccessful */
      return (names->kernel_name[0] != '\0') ? names : NULL;
}

static inline name_entry *addr_to_name(void *addrp, name_entry *names)
{
      /* The table ends with a blank entry */
      while (names->kernel_name[0] != '\0' &&
             memcmp(addrp, &names->addr, sizeof(names->addr)))
            ++names;

      /* Return NULL if the lookup was unsuccessful */
      return (names->kernel_name[0] != '\0') ? names : NULL;
```

```
}

static inline long getlong(char *s)
{
     long val;
     char *endptr;

     val = strtol(s, &endptr, 10);
     if (*s != '\0' && *endptr =='\0')
           return val;
     else
           return -1;
}

static void init_name_entries(char *filename, name_entry **names)
{
     FILE *fp;
     long tmp;
     int n, line, pass;
     struct hostent *hp;
     name_entry *entryp = NULL;
     char buf[LINE_BUF_SIZE], *p, *commap;

     if ((fp = fopen(filename, "r")) == NULL)
           perr(1, 1, "Cannot open %s", filename);

     /*
      * Read the configuration file in two passes
      */
     for (pass = 0; pass <= 1; ++pass) {
           for (n = 0, line = 1; fgets(buf, sizeof(buf), fp); line++) {
                 if (buf[strlen(buf) - 1] != '\n')
                       perr(1, 0, "Line %d of %s too long or unterminated",
                             line, filename);
                 else
                       buf[strlen(buf) - 1] = '\0';

                 /* Get first non-whitespace character */
                 for (p = buf; *p == ' ' || *p == '\t'; ++p)
                       ;

                 /* Ignore empty lines and comments */
                 if (*p == '\0' || *p == '#')
                       continue;
                 else
                       ++n;

                 /* In the second pass, fill in memory structures */
                 if (pass) {
                       if ( !(commap = strchr(p, ',')))
                             perr(1, 0, "Syntax error on line %d of %s",
                                   line, filename);
                       if (commap - buf > KSOCKET_KERNEL_NAME_LEN)
                             perr(1, 0, "Kernel name on line %d of %s too long",
                                   line, filename);

                       *commap = '\0';
                       strcpy(entryp->kernel_name, buf);

#if defined(USRMODE_USE_TCPIP)
                       /* Lookup the hostname (the remainder of the line) on DNS */
                       if (! (hp = gethostbyname(commap + 1)))
```

```
                              perr(1, 0, "DNS lookup of %s on line %d of %s failed",
                                    commap + 1, line, filename);
                        else
                              memcpy(&(entryp++)->addr, hp->h_addr, sizeof(struct
in_addr));
#elif defined(USRMODE_USE_SCI)
                        /* Verify the specified SCI node id */
                        if ((tmp = getlong(commap + 1)) < 0 ||
                            tmp > SCI_MAX_NODE_ID)
                              perr(1, 0, "Invalid SCI id '%s' on line %d of %s",
                                    commap + 1, line, filename);
                        else
                              (entryp++)->addr = tmp;
#endif
                  }

            }

            /* After the first pass, setup data structures and start over */
            if (pass == 0) {
                  if (!n)
                        perr(1, 0, "File %s contains no entries!", filename);
                  *names = entryp = xmalloc((n + 1) * sizeof(name_entry));
                  rewind(fp);
            } else
                  /* Mark the last table entry, using an empty kernel name */
                  entryp->kernel_name[0] = '\0';
      }

      if (fclose(fp))
            perr(1, 1, "Could not close file %s");
}

/*
 * set_socket_state:   Use ioctl to update the state information of a ksocket
 */
static inline void set_socket_state(uksocket *uksockp, ksocket_state_t state, int error)
{
      struct ioc_setstate st;

      /* Update userspace structures */
      uksockp->state = state;

      /* Update kernelspace structures */
      st.state = state;
      st.error = error;
      st.port = uksockp->port;
      st.remote_port = uksockp->remote_port;
      strcpy(st.remote_name, uksockp->remote_name);

      if (ioctl(uksockp->fd, KSOCKET_IOC_SETSTATE, &st) < 0)
            perr(0, 1, "%s: KSOCKET_IOC_SETSTATE failed for 0x%p",
                  __FUNCTION__, &st);
}

/*
 * connect_thread:     Establish connection to a remote kernel and
 *                enter the main data transfer loop
 */
void *connect_thread(void *arg)
{
      int err;
```

```
	name_entry *np;
	uksocket *uksockp = arg;

	/* Lookup the address of the remote kernel */
	if ( !(np = name_to_addr(uksockp->remote_name, names))) {
		set_socket_state(uksockp, KSOCKET_BOUND, -EHOSTUNREACH);
		return (void *)0;
	}

	/* Establish connection using protocol specific function */
	if ((err = protocol_connect(uksockp, np))) {
		set_socket_state(uksockp, KSOCKET_BOUND, err);
		return (void *)0;
	} else {
		set_socket_state(uksockp, KSOCKET_CONNECTED, 0);
		protocol_main_loop(uksockp);

		ports[uksockp->port] = NULL;
		free(uksockp);
		return (void *)0;
	}
}

/*
 * accept_thread:	Wait for connection request from a remote kernel
 *			and enter the main data transfer loop
 */
void *accept_thread(void *arg)
{
	int err;
	uksocket *uksockp = arg;

	/* Establish connection using protocol specific function */
	if ((err = protocol_accept(uksockp))) {
		set_socket_state(uksockp, KSOCKET_BOUND, err);
		return (void *)0;
	} else {
		set_socket_state(uksockp, KSOCKET_CONNECTED, 0);
		protocol_main_loop(uksockp);

		ports[uksockp->port] = NULL;
		free(uksockp);
		return (void *)0;
	}
}

/*
 * process_request:	Process a request from kernelspace
 */
static inline int process_request(ksocket_req_t *req, char *control_device)
{
	int fd, err;
	uint16_t port;
	uksocket *uksockp;
	pthread_attr_t attr;

	P_DEBUG(stderr, "%s: Processing a request:\n", __FUNCTION__);
	P_DEBUG(stderr, "\ttype is %d, user_data pointer is 0x%p\n",
		(int) req->type, req->user_data);

	/* A kernel request has been read, process it according to its type field */
	switch (req->type) {
```

```
case KSOCKET_REQ_OPEN:
      /* Reopen the ksocket char device, get a new fd */
      if ((fd = open(control_device, O_RDWR)) < 0) {
            perr(0, 1, "%s: opening %s failed",
                  control_device, __FUNCTION__);
            break;
      }

      /* Allocate a new usermode ksocket structure */
      uksockp = xmalloc(sizeof(uksocket));
      uksockp->fd = fd;
      uksockp->port = 0;
      uksockp->remote_port = 0;
      uksockp->remote_name[0] = '\0';
      protocol_init(uksockp);

      /*
       * Perform the KSOCKET_IOC_SETSOCKET ioctl on fd to associate
       * the underlying file structure with the kernel and userspace
       * ksocket structures
       */
      if (ioctl(fd, KSOCKET_IOC_SETSOCKET, uksockp) < 0) {
            perr(0, 1, "%s: KSOCKET_IOC_SETSOCKET failed for 0x%p",
                  __FUNCTION__, uksockp);
            break;
      }

      /* Call the protocol specific open function */
      if ((err = protocol_open(uksockp)) < 0) {
            set_socket_state(uksockp, KSOCKET_CLOSE_INPROGRESS, err);
            if (close(fd) < 0)
                  perr(0, 1, "%s: Close fd failed", __FUNCTION__);
            protocol_destroy(uksockp);
            free(uksockp);
      } else
            /* Update state information for this ksocket */
            set_socket_state(uksockp, KSOCKET_OPEN, 0);
      break;
case KSOCKET_REQ_BIND:
      /* Decode fields of request: User data pointer and port number */
      uksockp = req->user_data;
      port = req->port;

      /* Sanity check */
      if (port > KSOCKET_MAX_PORTNR) {
            /* This request should have been denied by the kernel */
            perr(0, 0, "%s: Invalid bind request for port %d",
                  __FUNCTION__, port);
            break;
      }

      /* If another ksocket is already bound to the requested port, fail */
      if (ports[port]) {
            set_socket_state(uksockp, KSOCKET_OPEN, -EADDRINUSE);
            break;
      }

      /* Call the protocol specific bind function */
      if ((err = protocol_bind(uksockp, port)) < 0) {
            set_socket_state(uksockp, KSOCKET_OPEN, err);
            break;
      }
```

```
                uksockp->port = port;
                ports[port] = uksockp;
                set_socket_state(uksockp, KSOCKET_BOUND, 0);
                break;

        case KSOCKET_REQ_CONNECT:
                /*
                 * Decode fields of request: User data pointer,
                 * remote kernel name and port number
                 */
                uksockp = req->user_data;
                port = req->port;

                /* Sanity check */
                if (port > KSOCKET_MAX_PORTNR) {
                        perr(0, 0, "%s: Invalid connect request for port %d",
                                __FUNCTION__, port);
                        break;
                }

                uksockp->remote_port = port;
                strncpy(uksockp->remote_name, req->data, KSOCKET_KERNEL_NAME_LEN + 1);

                /*
                 * Create a new thread of execution to establish the connection
                 * and take over data transfer between local and remote kernel
                 */
                pthread_attr_init(&attr);
                pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
                if ((err = pthread_create(&uksockp->thread, &attr, connect_thread,
uksockp))) {
                        errno = err;
                        perr(1, 1, "%s: pthread_create for connect_thread failed",
                                __FUNCTION__);
                }
                pthread_attr_destroy(&attr);

                break;
        case KSOCKET_REQ_ACCEPT:
                /*
                 * The request has only one field, the user data pointer
                 */
                uksockp = req->user_data;

                /*
                 * Create a new thread of execution, which will block until a
                 * connection request from a remote kernel has been received.
                 * The new thread will perform the data transfer between local
                 * and remote kernel
                 */
                pthread_attr_init(&attr);
                pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
                if ((err = pthread_create(&uksockp->thread, &attr, accept_thread,
uksockp))) {
                        errno = err;
                        perr(1, 1, "%s: pthread_create for accept_thread failed",
                                __FUNCTION__);
                }
                pthread_attr_destroy(&attr);

                break;
```

```
            case KSOCKET_REQ_CLOSE:
                  /*
                   * A close request can only be serviced if the ksocket is in
                   * the KSOCKET_OPEN or KSOCKET_BOUND state. If it is in the
                   * KSOCKET_CONNECTED state, then the request is meaningless.
                   * If the ksocket is connected, the data transfer thread
                   * detects that it has been closed by examining the return value
                   * of read()/write().
                   */
                  uksockp = req->user_data;
                  switch (uksockp->state) {
                        case KSOCKET_BOUND:
                              ports[uksockp->port] = NULL;
                              /* fallthru */
                        case KSOCKET_OPEN:
                              /* Closing the ksocket fd sets the ksocket state to
UNLINKED */
                              close(uksockp->fd);

                              /* Call the protocol specific close function */
                              protocol_close(uksockp);
                              protocol_destroy(uksockp);
                              free(uksockp);
                              break;
                        default:
                             perr(0, 0, "%s: Invalid close request for ksocket state %d",
                                    __FUNCTION__, uksockp->state);
                  }
                  break;
            default:
                  perr(0, 0, "%s: Unknown kernelspace request type %d",
                        __FUNCTION__, req->type);
      }

      return 0;
}

int open_control_device(char *filename)
{
      int fd;

      /* Open the control device in read-only, blocking mode */
      if ((fd = open(filename, O_RDONLY)) < 0)
            perr(1, 1, "Cannot open character device %s", filename);

      /* Reset the request queue */
      if (ioctl(fd, KSOCKET_IOC_RESETQUEUE) < 0)
            perr(1, 1, "ioctl IOC_RESETQUEUE failed for fd %d", fd);

      return fd;
}

/*
 * Signal handling functions
 * These handlers are process-wide, affecting all threads of execution
 */
void fatal_signal_handler(int signum)
{
      perr(1, 0, "%s: Signal %d caught: %s\n",
            __FUNCTION__, signum, sys_siglist[signum]);
}
```

```
void set_signal_handlers(void)
{
     sigset_t set;
     struct sigaction act;

     /*
      * SIGPIPE: Ignore it, write() returns EPIPE on broken pipes instead
      */
     sigemptyset(&set);
     act.sa_mask = set;
     act.sa_handler = SIG_IGN;
     act.sa_flags = SA_RESTART;
     if (sigaction(SIGPIPE, &act, NULL) < 0)
           perr(1, 1, "Setting SIGPIPE handler failed");

     /*
      * SIGSEGV: Oops, something has gone awfully wrong... Just exit gracefully
      */
     sigemptyset(&set);
     act.sa_mask = set;
     act.sa_flags = 0;
     act.sa_handler = fatal_signal_handler;
     if (sigaction(SIGSEGV, &act, NULL) < 0)
           perr(1, 1, "Setting SIGSEGV handler failed");

     /*
      * SIGFPE handler
      */
     sigemptyset(&set);
     act.sa_mask = set;
     act.sa_flags = 0;
     act.sa_handler = fatal_signal_handler;
     if (sigaction(SIGFPE, &act, NULL) < 0)
           perr(1, 1, "Setting SIGFPE handler failed");
}

void daemonize(void)
{
     pid_t pid;

     P_DEBUG(stderr, "%s: Called\n", __FUNCTION__);

     /* Change working directory to '/', to avoid keeping any dirs busy */
     // FIXME: chdir("/");

     /* Become a daemon by detaching from controlling terminal */
     if ((pid = fork()) > 0)
           exit(0);
     else if (pid < 0)
           perr(1, 1, "unable to fork new process");

     /* Have the child perform a setsid */
     if (setsid() < 0)
           perr(1, 1, "setsid failed");

     /* Close stdin, stdout,stderr and re-open them */
     if (freopen("/dev/null", "r", stdin) < 0)
           perr(1, 1, "could not re-open stdin");

     if (freopen("/dev/null", "w", stdout) < 0)
           perr(1, 1, "could not re-open stdout");
```

```
    if (freopen(KSOCKET_DAEMON_LOGFILE, "w", stderr) < 0)
        perr(1, 1, "could not re-open stderr to %s", KSOCKET_DAEMON_LOGFILE);

    P_DEBUG(stderr, "%s: Leaving, completed successfully\n", __FUNCTION__);
}

int main(int argc, char *argv[])
{
    int i, fd;
    int become_daemon;
    ksocket_req_t req;
    char *control_device;

    credits();
    set_signal_handlers();
    init_name_entries(KERNEL_LIST_FILE, &names);

    /* Process command-line options */
    become_daemon = 1;
    control_device = NULL;
    for (i = 1; i < argc; i++) {
        if (STR_EQ(argv[i], "-nodetach")) {
            become_daemon = 0;
        } else {
            if (control_device != NULL)
                perr(1, 0, "Too many command-line arguments");
            else
                control_device = argv[i];
        }
    }

    if (become_daemon)
        daemonize();

    if (!control_device)
        perr(1, 0, "The name of the control device (eg. /dev/ksocket0) is required");

    fd = open_control_device(control_device);

    /* Initialize the communication library */
    if (protocol_start() < 0)
        perr(1, 0, "Initialization of communication library failed");

    /*
     * Start the main processing loop:
     *    1. read() a kernel request from the control char device
     *    2. process the request (eg. open or close a connection to a remote ksocket)
     *    3. update the state of the ksocket that was affected by the request
     */
    for (;;) {
        /* Dequeue a request from the request queue, blocking system call */
        if (read(fd, &req, sizeof(req)) < 0)
            perr(1, 1, "Read from control fd failed");
        else
            /* Decode and process kernel request */
            if (process_request(&req, control_device) < 0)
                perr(0, 0, "process_request failed");
    }

    if (close(fd) < 0)
        perr(1, 1, "close failed");
```

```
    return 0;
}
```

αρχείο usrmode/Makefile:

```
############################################################
#                                                          #
# Makefile                                                 #
#                                                          #
############################################################

SCIDIR = /usr/src/DIS

CC = gcc

INCLUDES =  -I$(SCIDIR)/src/SISCI/api -I$(SCIDIR)/src/SISCI/src -I..

# Get the architecture-specific options to the C compiler
CFLAGS = $(ARCH_CFLAGS)

CFLAGS += $(INCLUDES) -Wall
CFLAGS += -O2 -finline-functions -fexpensive-optimizations -fomit-frame-pointer

# Uncomment to include symbol information and enable debugging messages
# CFLAGS += -DKSOCKET_DEBUG -g

# Uncomment the following line to use TCP/IP sockets
# CFLAGS += -DUSRMODE_USE_TCPIP

# Uncomment the following block to use the SISCI library
CFLAGS += -DUSRMODE_USE_SCI
LIBS += -L$(SCIDIR)/src/SISCI/api -lsisci

# Multi-threading support by the POSIX compliant LinuxThreads library
CFLAGS += -D_REENTRANT
LIBS += -lpthread

OBJS = usrmode.o
BIN = usrmode

all: $(BIN)

$(BIN): $(OBJS) usrmode.h
     $(CC) $(CFLAGS) -o $(BIN) $(OBJS) $(LIBS)

clean:
     rm -f $(OBJS) $(BIN) *~ core
```

αρχείο testprog/testprog.h:

```
/*
 * testprog.h
 *
 * Definitions used by the usermode ksocket-test client
 *
 * Vangelis Koukis, 2002
 */

#ifndef _TESTPROG_H
#define _TESTPROG_H

#define TEST_PROGRAM_VERSION "ksocket-test01"

/*
 * Some useful macros...
 */
#define STR_EQ(p,q)    (!strcmp((p),(q)))

#endif     /* _TESTPROG_H */
```

αρχείο testprog/testprog.c:

```
/*
 * testprog.c
 *
 * A program that issues ioctl commands to setup
 * a ksocket-test device
 *
 * Vangelis Koukis, 2002
 */

#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "ksocket-test.h"
#include "testprog.h"

/*
 * perr:   Main error reporting function
 */

void perr(int fatal, int want_errno, char *fmt, ...)
{
     va_list ap;
     int err_number = errno;       /* We need errno NOW */

     va_start(ap, fmt);
     fprintf(stderr, fatal ? "Fatal error: " : "Error: ");
     vfprintf(stderr, fmt, ap);
     fprintf(stderr, "\n");
     va_end(ap);

     if (want_errno)
           fprintf(stderr, "Errno was: %d - %s\n",
                 err_number, sys_errlist[err_number]);

     if (fatal)
           exit(1);
}

void credits(void)
{
     fprintf(stderr, "KSocket library test program version %s\n", TEST_PROGRAM_VERSION);
     fprintf(stderr, "Vangelis Koukis, 2002\n\n");
}

void usage(char *s)
{
     fprintf(stderr, "Usage: %s device nonblock cmd [arg...]\n\n", s);
     fprintf(stderr, "device:\t\tThe name of the device, eg. /dev/ksocket/test0\n");
     fprintf(stderr, "nonblock:\tDetermines whether the O_NONBLOCK flag is used\n");
     fprintf(stderr, "cmd:\t\tOne of { open, bind, connect, accept, close, reset,
status }\n\n");
}
```

```
const char *ksocket_state_descr(ksocket_state_t state)
{
     char *p;
     switch(state) {
           case KSOCKET_OPEN_INPROGRESS:
                 p = "KSOCKET_OPEN_INPROGRESS"; break;
           case KSOCKET_BIND_INPROGRESS:
                 p = "KSOCKET_BIND_INPROGRESS"; break;
           case KSOCKET_CONNECT_INPROGRESS:
                 p = "KSOCKET_CONNECT_INPROGRESS"; break;
           case KSOCKET_CLOSE_INPROGRESS:
                 p = "KSOCKET_CLOSE_INPROGRESS"; break;
           case KSOCKET_OPEN:
                 p = "KSOCKET_OPEN"; break;
           case KSOCKET_BOUND:
                 p = "KSOCKET_BOUND"; break;
           case KSOCKET_CONNECTED:
                 p = "KSOCKET_CONNECTED"; break;
           case KSOCKET_CLOSED_LINKED:
                 p = "KSOCKET_CLOSED_LINKED"; break;
           case KSOCKET_OPEN_UNLINKED:
                 p = "KSOCKET_OPEN_UNLINKED"; break;
           case KSOCKET_CLOSED_UNLINKED:
                 p = "KSOCKET_CLOSED_UNLINKED"; break;
           default:
                 p ="Unknown ksocket state";
     }

     return p;
}

int main(int argc, char *argv[])
{
     int fd, flags = 0;
     int port;
     struct ioc_remotedata remotedata;
     struct ioc_setstate st;

     credits();

     if (argc < 4) {
           usage(argv[0]);
           perr(1, 0, "At least three arguments required");
     }

     /* Use non-blocking mode? */
     if (STR_EQ(argv[2], "nonblocking"))
           flags = O_NONBLOCK;
     else if (!STR_EQ(argv[2], "blocking")) {
           usage(argv[0]);
           perr(1, 0, "The nonblock argument must be 'nonblocking' or 'blocking'");
     }

     /* Open the ksocket-test device in read-only mode */
     if ((fd = open(argv[1], O_RDONLY | flags)) < 0)
           perr(1, 1, "Cannot open character device %s", argv[1]);

     /*
      * Examine the 'cmd' argument and issue the appropriate
      * ioctl() call to the ksocket-test driver
      */
```

```
if (STR_EQ(argv[3], "open"))
      if (ioctl(fd, KSOCKET_TEST_IOC_OPEN) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_OPEN ioctl failed");

if (STR_EQ(argv[3], "close"))
      if (ioctl(fd, KSOCKET_TEST_IOC_CLOSE) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_CLOSE ioctl failed");

if (STR_EQ(argv[3], "accept"))
      if (ioctl(fd, KSOCKET_TEST_IOC_ACCEPT) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_ACCEPT ioctl failed");

if (STR_EQ(argv[3], "reset"))
      if (ioctl(fd, KSOCKET_TEST_IOC_RESET) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_RESET ioctl failed");

if (STR_EQ(argv[3], "bind")) {
      if (argc != 5)
            perr(1, 0, "One arg required: local port number");
      else {
            port = atoi(argv[4]);
            if (ioctl(fd, KSOCKET_TEST_IOC_BIND, &port) < 0)
                  perr(1, 1, "KSOCKET_TEST_IOC_BIND ioctl failed");
      }
}

if (STR_EQ(argv[3], "connect")) {
      if (argc != 6)
            perr(1, 0, "Two args required: remote kernel name, remote port");
      else {
            remotedata.port = atoi(argv[5]);
            strncpy(remotedata.name, argv[4], KSOCKET_KERNEL_NAME_LEN + 1);
            if (ioctl(fd, KSOCKET_TEST_IOC_CONNECT, &remotedata) < 0)
                  perr(1, 1, "KSOCKET_TEST_IOC_CONNECT ioctl failed");
      }
}

if (STR_EQ(argv[3], "status")) {
      if (ioctl(fd, KSOCKET_TEST_IOC_STATUS, &st) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_STATUS ioctl failed");

      printf("Ksocket status information:\n\tState = %s,\n\tport = %d,\n\t"
             "remote port = %d, remote name = \"%s\"\n\n",
             ksocket_state_descr(st.state), st.port,
             st.remote_port, st.remote_name);
}

if (STR_EQ(argv[3], "bench_mirror"))
      if (ioctl(fd, KSOCKET_TEST_IOC_BENCH_MIRROR) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_BENCH_MIRROR ioctl failed");

if (STR_EQ(argv[3], "bench_latency"))
      if (ioctl(fd, KSOCKET_TEST_IOC_BENCH_LATENCY) < 0)
            perr(1, 1, "KSOCKET_TEST_IOC_BENCH_LATENCY ioctl failed");

if (!STR_EQ(argv[3], "open") && !STR_EQ(argv[3], "close") &&
    !STR_EQ(argv[3], "bind") && !STR_EQ(argv[3], "connect") &&
    !STR_EQ(argv[3], "accept") && !STR_EQ(argv[3], "status") &&
    !STR_EQ(argv[3], "reset") && !STR_EQ(argv[3], "bench_mirror") &&
    !STR_EQ(argv[3], "bench_latency")) {
      usage(argv[0]);
      perr(1, 0, "Specified value for 'cmd' is invalid");
```

```
	} else
		printf("ioctl() call completed\n\n");

	if (close(fd) < 0)
		perr(1, 1, "Cannot close character device");

	return 0;
}
```

αρχείο testprog/Makefile:

```
############################################################
#                                                          #
# Makefile                                                 #
#                                                          #
############################################################

CC = gcc

# Get the architecture-specific options to the C compiler
CFLAGS = $(ARCH_CFLAGS)

CFLAGS +=  -I..
CFLAGS += -O2 -finline-functions -fexpensive-optimizations -fomit-frame-pointer
CFLAGS += -Wall

# Uncomment to include symbol information
# CFLAGS += -g

OBJS = testprog.o
BIN = testprog

all: $(BIN)

$(BIN): $(OBJS) testprog.h
     $(CC) -o $(BIN) $(OBJS)

clean:
     rm -f $(OBJS) $(BIN) *~ core
```