ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Τροποποίηση της υλοποίησης διαδικτυακών πρωτοκόλλων λειτουργικού συστήματος για την εκμετάλλευση πολλαπλών επεξεργαστικών μονάδων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Άγγελου Ι.
Οικονομόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Αθήνα, Σεπτέμβριος 2009

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

Τροποποίηση της υλοποίησης διαδικτυακών πρωτοκόλλων λειτουργικού συστήματος για την εκμετάλλευση πολλαπλών επεξεργαστικών μονάδων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Άγγελου Ι.
Οικονομόπουλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29-09-2009.

.....	.....	.....
Νεκτάριος Κοζύρης Αναπληρωτής Καθηγητής	Νικόλαος Παπασπύρου Επίκουρος Καθηγητής	Δημήτριος Φωτάκης Λέκτορας

Αθήνα, Σεπτέμβριος 2009.

.....
Άγγελος Ι. Οικονομόπουλος
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

Περίληψη

Καθώς οι νέοι επεξεργαστές μετακινούνται σε ένα μοντέλο με πολλούς πυρήνες επεξεργασίας (ΠΕ), οι πυρήνες λειτουργικών συστημάτων πρέπει να είναι σε θέση να εκμεταλευτούν τις επιπλέον επεξεργαστικές μονάδες. Οι περισσότεροι πυρήνες επιλέγουν τη χρήση βραχύβιων (fine-grained) κλειδωμάτων για να επιτρέψουν παράλληλη εκτέλεση στον πυρήνα. Αντίθετα, ο πυρήνας του DragonFlyBSD διαμοιράζει τις κύριες δομές δεδομένων των διαδικτυακών πρωτοκόλλων και μεταφέρει την επεξεργασία πρωτοκόλλου σε ανεξάρτητα νήματα του πηρύνα (ένα ανά ΠΕ) τα οποία επικοινωνούν με τις εφαρμογές με ανταλλαγή μηνυμάτων. Η προσέγγιση αυτή έχει τη δυνατότητα για γραμμική κλιμάκωση με το πλήθος των διαθέσιμων ΠΕ, ενώ παρέχει ένα απλούστερο προγραμματιστικό μοντέλο. Η εργασία μας αφαιρεί την ανάγκη για συγχρονισμό ανάμεσα στα νήματα επεξεργασίας πρωτοκόλλου στα μονοπάτια ελέγχου από τα οποία εξαρτάται η επίδοση. Ακόμα σχεδιάζουμε και υλοποιούμε μια αρχιτεκτονική για έναν ενταμιευτή εισόδου / εξόδου για τη δομή socket ο οποίος χρησιμοποιείται από τα νήματα του πυρήνα χωρίς αμοιβαίο αποκλεισμό ή αναμονή. Οι πρώτες μετρήσεις δίνουν ενθαρρυντικά αποτελέσματα και διερευνούμε τις κατευθύνσεις για μελλοντικές προσπάθειες βελτίωσης.

Keywords: Πολλαπλοί πυρήνες επεξεργασίας, Διαδικτυακά πρωτόκολλα, TCP, UDP, Παραγωγός-καταναλωτής, Βραχύβια κλειδώματα, Ανταλλαγή μηνυμάτων, DragonFlyBSD

Περίληψη

As CPU manufacturers move to a multi-core model, OS kernels must be ready to take advantage of additional processing units. Most kernels have elected to use fine-grained locking in order to allow parallel execution in the kernel. Instead, the DragonFlyBSD kernel partitions the main network-related data structures onto each cpu and moves network protocol processing into independant per-cpu kernel threads that communicate with applications using message passing. This approach has the potential to scale linearly with the processor count while providing a simpler programming model. Our work eliminates the need for synchronization between the protocol processing threads on the performance sensitive control paths. We also design and implement a socket buffer architecture which is used by the kernel threads in a lock and wait free fashion. Our initial measurements give promising results and we investigate directions for further improvements.

Keywords: SMP, Multi-core, Internet protocol, TCP, UDP, Producer-consumer, Fine-grained locking, Message passing, DragonFlyBSD

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Κεφάλαιο 1

Διαδικτυακά πρωτόκολλα

Η δημιουργία των δικτυακών πρωτοκόλλων προέκυψε από την ανάγκη για διασύνδεση υπολογιστικών συστημάτων, πολλά από τα οποία ανήκουν σε διαφορετικές αρχιτεκτονικές υλικού (hardware) και χρησιμοποιούν ποικίλες πλατφόρμες λογισμικού (software). Στις μέρες μας, η διασύνδεση αυτή επιτυγχάνεται κυρίως μέσω των πρωτοκόλλων διαδικτύου (internet protocols) τα οποία χρησιμοποιούνται για την ανταλλαγή δεδομένων και στο παγκόσμιο διαδίκτυο (internet). Σε αυτό το κεφάλαιο θα περιγράψουμε τα βασικότερα των διαδικτυακών πρωτοκόλλων επικεντρώνοντας σε εκείνα που συνδέονται άμεσα με το αντικείμενο αυτής της εργασίας.

Τα διαδικτυακά πρωτόκολλα είναι προτυποποιημένα από την IETF (Internet Engineering Task Force) σε μια σειρά εγγράφων που ονομάζονται RFC (Request For Comment). Στο εξής, όταν αναφερόμαστε σε κάποιο συγκεκριμένο RFC θα το κάνουμε με την έκφραση RFCdddd (όπου τα d είναι ψηφία). Όπως είναι φυσικό, θα προσπαθήσουμε να περιοριστούμε στην ορολογία που καθορίζεται στα RFC. Σε κάποια σημεία όμως, θα είναι ίσως χρήσιμο να αναφερθούμε και στην ανταγωνιστική προτυποποίηση OSI (Open Systems Interconnect).

1.1 Δομή ενός δικτύου

Ένα δίκτυο μπορεί να αποτελείται από πολλά είδη μηχανημάτων και κάθε μηχάνημα είναι μέρος του δικτύου παίζοντας έναν ή περισσότερους ρόλους. Το RFC1122 ορίζει το ρόλο του υπολογιστή υπηρεσίας (host) που μπορεί να είναι

π.χ. ένας προσωπικός υπολογιστής, ένας διακομιστής ή μία διαδικτυακή συσκευή, σαν τον τελικό καταναλωτή των υπηρεσιών επικοινωνίας που παρέχει το δίκτυο. Σύμφωνα με το ίδιο έγγραφο, ένα μηχάνημα που συνδέει δύο ή περισσότερα δίκτυα καλείται δρομολογητής IP. Στη σύνθεση ενός δικτύου μπορεί να συμμετέχουν επίσης μεταγωγείς (switches), συγκεντρωτές (hubs) και γέφυρες (bridges).

1.2 Στοίβα πρωτοκόλλων διαδικτύου

Για να περιγράψουμε τα διαδικτυακά πρωτόκολλα που μας ενδιαφέρουν, θα θεωρήσουμε ότι καθένα από αυτά ανήκει σε ένα από τέσσερα λογικά στρώματα. Τα στρώματα αυτά είναι οργανωμένα σε μία στοίβα έτσι ώστε κάθε στρώμα να στηρίζεται στις υπηρεσίες που παρέχουν τα κατώτερα στρώματα της στοίβας.

Ξεκινώντας την περιγραφή από τη βάση, έχουμε το στρώμα ζεύξης (link layer). Η μετάδοση δεδομένων στο στρώμα ζεύξης γίνεται συνήθως με το πρωτόκολλο ethernet, αλλά πρωτόκολλα όπως το token bus (IEEE802.4), token ring (IEEE802.5), IEEE802.11, bluetooth (IEEE802.15.1), frame relay, myrinet και άλλα χρησιμοποιούνται ανάλογα με τις απαιτήσεις κάθε εφαρμογής. Εναλλακτικά, σε περιπτώσεις που απαιτείται πιστοποίηση της ταυτότητας των επικοινωνούντων μερών (π.χ. στην επικοινωνία ενός οικιακού χρήστη με τον παροχέα υπηρεσιών internet) ή συμπίεση/κρυπτογράφηση, χρησιμοποιείται το πρωτόκολλο PPP (Point-to-point protocol).

Στις περισσότερες περιπτώσεις χρησιμοποιείται το πρωτόκολλο ARP (Address Resolution Protocol, RFC826) για την αντιστοίχιση διευθύνσεων του στρώματος δικτύου σε διευθύνσεις του στρώματος ζεύξης.

Με τη χρήση του στρώματος ζεύξης είναι δυνατό να ανταλλάξουν δεδομένα δύο ή περισσότεροι υπολογιστές υπηρεσίας που είναι άμεσα συνδεδεμένοι. Όπως είναι εύλογο, θα θέλαμε να είναι δυνατό να επικοινωνήσουν υπολογιστές υπηρεσίας ανεξάρτητα από τη θέση τους στην τοπολογία ενός αυθαίρετα πολύπλοκου δικτύου. Για το σκοπό αυτό είναι απαραίτητη η δρομολόγηση πακέτων από τους ενδιάμεσους υπολογιστές υπηρεσίας που ανήκουν στο μονοπάτι που ενώνει τα επικοινωνούντα μέρη στο γράφο του δικτύου. Το RFC1122 ορίζει το στρώμα διαδικτύου (internet layer) στο οποίο ανήκουν τα πρωτόκολλα IP, ICMP και IGMP.

Echo Request
Echo Reply
Destination Unreachable
Time Exceeded
Redirect
Parameter Problem
Source Quench
Timestamp
Timestamp Reply

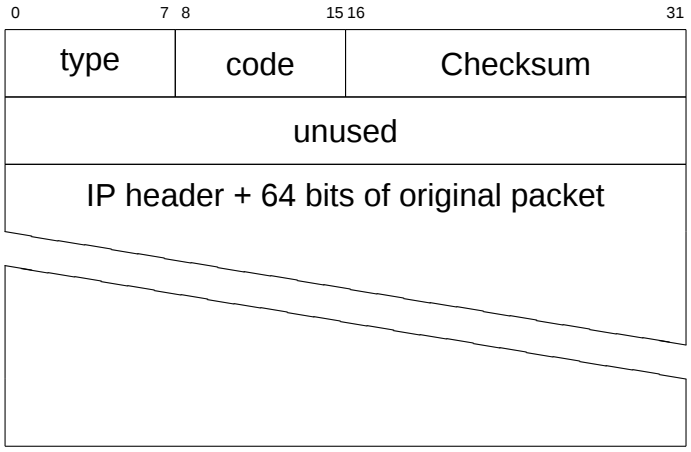
Πίνακας 1.1: Τύποι μηνυμάτων ICMP

1.3 ICMP

Το πρωτόκολλο ICMP (Internet Control Message Protocol, RFC792) χρησιμοποιείται για την ανταλλαγή μηνυμάτων λάθους ή διαγνωστικών μηνυμάτων που αφορούν τη λειτουργία του πρωτοκόλλου IP. Παρόλα αυτά, τα μηνύματα ICMP μεταδίδονται σε πακέτα IP. Η επικεφαλίδα των πακέτων του ICMP έχει διαφορετική μορφή ανάλογα με τον τύπο του μηνύματος. Ο τύπος του μηνύματος καθορίζεται από το πρώτο byte (στα κείμενα που ασχολούνται με δικτυακά πρωτόκολλα χρησιμοποιείται συχνά και ο όρος octet) της επικεφαλίδας. Οι πιο συνηθισμένοι τύποι μηνυμάτων ICMP φαίνονται στον πίνακα 1.1.

Ένα μήνυμα Destination Unreachable αποστέλλεται όταν λαμβάνεται ένα πακέτο για άγνωστο δίκτυο ή υπολογιστή υπηρεσίας, με άγνωστο αριθμό πρωτοκόλλου (στην επικεφαλίδα IP ή για μη ενεργή θύρα. Μήνυμα αυτό του τύπου μπορεί επίσης να σταλεί αν ένα πακέτο IP απορριφθεί είτε γιατί ο αποστολέας ζήτησε να μεταφερθεί χωρίς κατακερματισμό και αυτό δεν είναι δυνατό είτε γιατί προσδιόρισε λανθασμένα το μονοπάτι του πακέτου. Σε κάθε περίπτωση, η αιτία προσδιορίζεται στο πεδίο κώδικα της επικεφαλίδας αυτού του μηνύματος ICMP.

Όπως φαίνεται στο σχήμα, η επικεφαλίδα περιλαμβάνει ακόμα ένα άθροισμα ελέγχου (checksum) μήκους 16 bit που υπολογίζεται σαν το συμπλήρωμα ως προς 1 του αθροίσματος (με αριθμητική συμπληρώματος ως προς 1) του μηνύματος ICMP, θεωρώντας ότι το πεδίο του αθροίσματος ελέγχου είναι μηδέν. Τέλος, το μήνυμα περιλαμβάνει την επικεφαλίδα IP και τα πρώτα 8 bytes του πακέτου στο οποίο αναφέρεται το μήνυμα ώστε να είναι δυνατή η ταυτοποίησή του από τον αποστολέα.



Όμοια είναι και η μορφή της επικεφαλίδας για ένα μήνυμα τύπου Time Exceeded που αποστέλλεται όταν το πεδίο “διάρκεια ζωής” ενός ληφθέντος πακέτου είναι 0 ή αν εκπνεύσει το χρονικό όριο κατά την ανασύνθεση (reassembly) των τεμαχίων (fragments) IP.

Το μήνυμα Parameter Problem στέλνεται αν ληφθεί πακέτο με λανθασμένη επικεφαλίδα IP και έχει ICMP επικεφαλίδα ίδια με τον προηγούμενο τύπο.

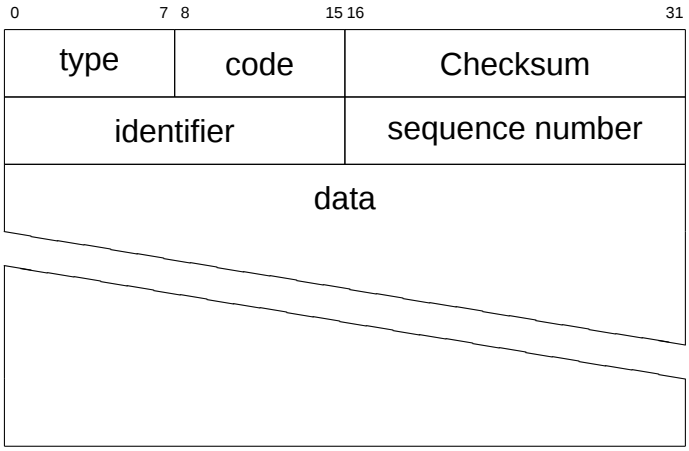
Αν ένας δρομολογητής δεν έχει αρκετή ελεύθερη μνήμη για να αποθηκεύσει ένα πακέτο, το απορρίπτει και μπορεί να στείλει ένα μήνυμα ICMP source quench με επικεφαλίδα IP ίδια με τον προηγούμενο τύπο.

Αν ένας δρομολογητής πρόκειται να προωθήσει ένα πακέτο IP σε δρομολογητή ο οποίος συνδέεται άμεσα με τον αποστολέα, μπορεί να ειδοποιήσει τον αποστολέα για την ύπαρξη αυτού του “καλύτερου” δρομολογητή με ένα μήνυμα ICMP redirect όπου η διεύθυνση του “καλύτερου” δρομολογητή περιέχεται στα bytes 5-8 της επικεφαλίδας ICMP που είναι κατά τα άλλα όμοια με τις παραπάνω.

Η επικεφαλίδα πακέτων τύπου ICMP echo και echo reply έχει τη μορφή που φαίνεται στο σχήμα.

Το πεδίο του κωδικού είναι πάντα μηδέν και το πεδίο identifier επιλέγεται αυθαίρετα από τον αποστολέα. Το πεδίο sequence number συνήθως αυξάνεται κατά ένα για κάθε νέο μήνυμα ICMP echo. Τα υπόλοιπα δεδομένα του πακέτου (αν υπάρχουν) επιλέγονται και αυτά από τον αποστολέα και επιστρέφονται αυτούσια, όπως και τα πεδία identifier και sequence number, στο αντίστοιχο μήνυμα echo reply. Τα μηνύματα αυτά χρησιμοποιούνται για διαγνωστικούς σκοπούς.

Τέλος υπάρχουν και τα ζεύγη μηνυμάτων timestamp/timestamp reply και information



request/information reply τα οποία χρησιμοποιούνται για τον συγχρονισμό ρολογιών και την ανάκτηση της διεύθυνσης δικτύου αντίστοιχα. Καθώς όμως υπάρχουν πλέον καταλληλότερα πρωτόκολλα για αυτές τις λειτουργίες, τα μηνύματα αυτά απαντώνται σπάνια στις μέρες μας.

1.4 IP

Το πρωτόκολλο IP (Internet Protocol, RFC791) καθορίζει τη διευθυνσιοδότηση επιπέδου δικτύου και τη δρομολόγηση των πακέτων σε κάθε δρομολογητή. Επιπλέον, φροντίζει για τον κατακερματισμό (fragmentation) των πακέτων όπου αυτό είναι απαραίτητο για να μεταδοθούν σε ζεύξεις με μέγιστη μονάδα μεταφοράς (maximum transmission unit ή MTU) μικρότερη από το μέγεθος του πακέτου.

Μια διεύθυνση IP έχει μέγεθος 4 byte και γράφεται συνήθως στη μορφή aa.bb.cc.dd, δηλαδή κάθε byte αναγράφεται ξεχωριστά. τα πρώτα n bits αποτελούν το τμήμα δικτύου και τα τελευταία (32 - n) bits αποτελούν το τμήμα υπολογιστή υπηρεσίας της διεύθυνσης. Η τιμή του n εξαρτάται από το δίκτυο στο οποίο ανήκει η διεύθυνση IP. Μια διεύθυνση IP δεν αντιστοιχεί σε ένα υπολογιστή υπηρεσίας αλλά σε μια (από τις πιθανόν πολλές) διασυνδέσεις (interfaces) του. Ακόμα, κάθε διασύνδεση μπορεί να έχει περισσότερες από μια διευθύνσεις IP χωρίς αυτές να ανήκουν απαραίτητα στο ίδιο δίκτυο.

Οι διευθύνσεις που καθορίζονται στο RFC3171 έχουν ειδική χρήση: αντί για τη διεύθυνση μιας διασύνδεσης, αναφέρονται σε ομάδες πολυεκπομπής (multicasting)

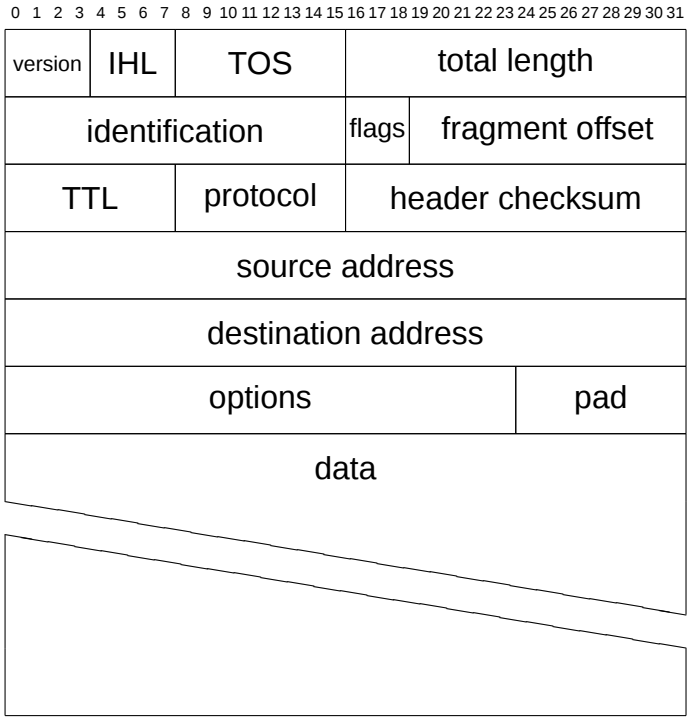
Δίκτυο προορισμού	Επόμενος δρομολογητής	Αριθμός διαδρομων
aa.bb.cc.dd/n	ee.ff.gg.hh	m

στην οποία συνήθως ανήκουν αρκετοί υπολογιστές υπηρεσίας. Με την πολυεκπομπή δεν θα ασχοληθούμε περισσότερο σε αυτό το κείμενο.

Ένας δρομολογητής IP διατηρεί ένα λογικό πίνακα δρομολόγησης για να είναι σε θέση να μεταφέρει σωστά τα πακέτα που δεν προορίζονται για αυτόν. Ο πίνακας αυτός περιέχει γραμμές της μορφής

Η διεύθυνση του πακέτου προς δρομολόγηση συγκρίνεται με τις διευθύνσεις δικτύων προορισμού και επιλέγεται η γραμμή με το μακρύτερο κοινό πρόθεμα. Φυσικά, αν ο δρομολογητής είναι άμεσα συνδεδεμένος με ένα δίκτυο προορισμού, δεν απαιτείται το πεδίο του επόμενου δρομολογητή.

Η μορφή ενός πακέτου IP είναι



Το πεδίο της έκδοσης (version) αναφέρεται στην έκδοση του πρωτοκόλλου IP. Σε αυτό το κείμενο ασχολούμαστε μόνο με την έκδοση 4. Το πεδίο IHL (internet header length) δίνει το μήκος της επικεφαλίδας σε λέξεις των τεσσάρων bytes.

Το πεδίο TOS (type of service, τύπος υπηρεσίας) καθορίζει το επίπεδο προτεραιότητας και αν το πακέτο έχει ιδιαίτερες απαιτήσεις για μικρή καθυστέρηση, υψηλή ρυθμαπόδοση ή αυξημένη αξιοπιστία (ή κάποιο συνδυασμό αυτών των απαιτήσεων).

Το 16-bit πεδίο total Length καθορίζει το συνολικό μήκος του IP πακέτου περιλαμβανομένης της επικεφαλίδας.

Το πεδίο identification είναι το ίδιο για κάθε τεμάχιο (fragment) IP ώστε ο παραλήπτης να είναι σε θέση να τα επανενώσει. Για το σκοπό αυτό χρησιμοποιεί και το τρίτο bit του πεδίου flags (που είναι 1 σε όλα τα τεμάχια πλην του τελευταίου).

Το πεδίο fragment offset (μετατόπιση τεμαχίου) καθορίζει σε ποια περιοχή του αρχικού πακέτου αντιστοιχεί το τεμάχιο (σε μονάδες των 8 bytes). Το πρώτο bit του πεδίου flags είναι πάντα 0 ενώ, αν το δεύτερο bit του είναι 1, τότε το πακέτο δεν επιτρέπεται να κατακερματιστεί.

Το πεδίο TTL (Time-to-live, χρόνος ζωής) μειώνεται τουλάχιστον κατά 1 κάθε φορά που το πακέτο δρομολογείται και το πακέτο απορρίπτεται όταν το πεδίο γίνει μηδέν. Με αυτό τον τρόπο αποφεύγεται η παγίδευση και ανακύκλωση πακέτων στο δίκτυο.

Το πεδίο του πρωτοκόλλου προσδιορίζει το πρωτόκολλο των δεδομένων του πακέτου (π.χ. UDP ή TCP).

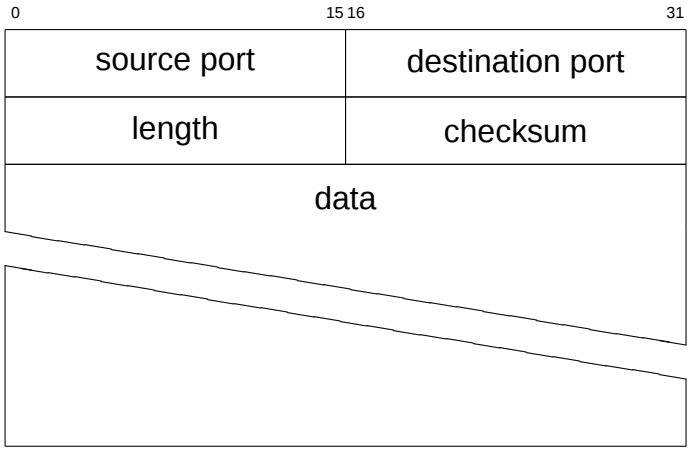
Το άθροισμα ελέγχου της επικεφαλίδας (header checksum) υπολογίζεται ακριβώς όπως και το άθροισμα ελέγχου της επικεφαλίδας ICMP (ενότητα 1.3).

Τέλος, το πρωτόκολλο IP επιτρέπει κάποιες προαιρετικές επιλογές. Η ύπαρξη αυτών των επιλογών υποδηλώνεται στο πεδίο του μήκους της επικεφαλίδας. Οι συνήθως χρησιμοποιούμενες επιλογές είναι η καταγραφή του χρόνου επεξεργασίας του πακέτου και ο προσδιορισμός ή η καταγραφή της διαδρομής του.

1.5 UDP

Το απλούστερο πρωτόκολλο που υλοποιείται πάνω στο IP είναι το User Datagram Protocol ή UDP (RFC768). Το UDP δεν δίνει καμία εγγύηση για την παράδοση των πακέτων, ούτε έστω για τη σειρά με την οποία θα παραδοθούν. Η επικεφαλίδα του είναι απλά αυτή που φαίνεται στο σχήμα 1.5.

Η θύρα προορισμού (και, όταν απαιτείται, η θύρα προέλευσης) χρησιμοποιείται για την πολύπλεξη/αποπολύπλεξη των δεδομενογραφημάτων (πακέτων) ώστε

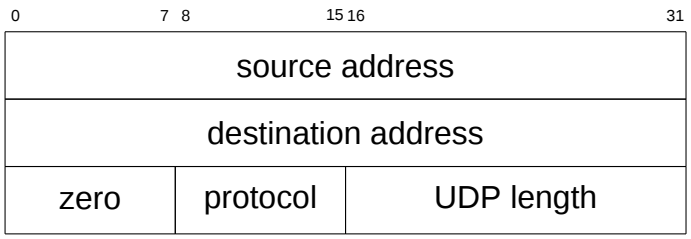


Σχήμα 1.1: Επικεφαλίδα UDP

να μπορούν να επικοινωνούν περισσότερα από ένα ζευγάρια εφαρμογών (που η μία εκτελείται στον αποστολέα και η άλλη στον παραλήπτη) ταυτόχρονα.

Το πεδίο length προσδιορίζει το μήκος (σε bytes) του δεδομενογραφήματος περιλαμβανομένης της επικεφαλίδας UDP.

Το πεδίο του αθροίσματος ελέγχου (checksum) υπολογίζεται πάνω σε ολόκληρο το δεδομενογράφημα με την προσθήκη της ιδεατής επικεφαλίδας που φαίνεται στο σχήμα 1.2.

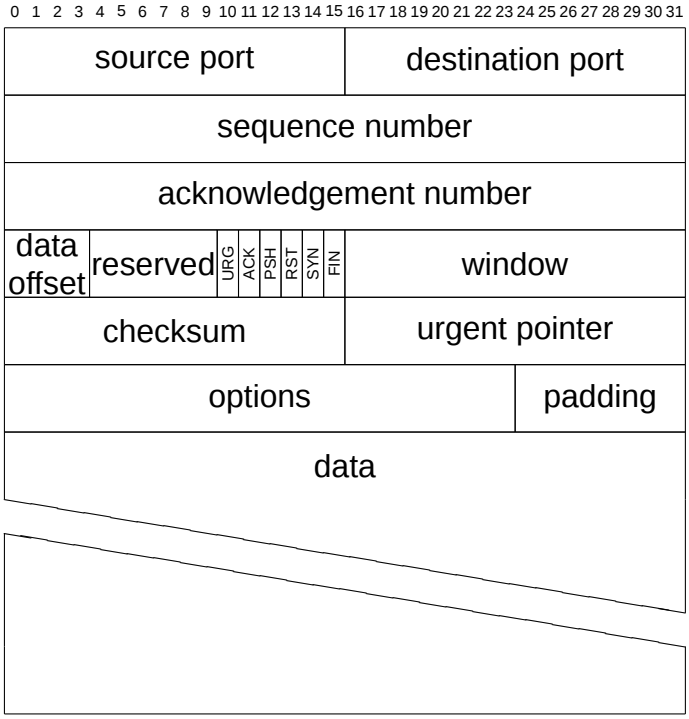


Σχήμα 1.2: Ψευδοεπικεφαλίδα

1.6 TCP

Το άλλο σημαντικό πρωτόκολλο που υλοποιείται πανω από το IP είναι το Transmission Control Protocol (TCP, RFC793). Το TCP με τη σειρά του είναι αυτό στο οποίο στηρίζονται τα περισσότερα πρωτόκολλα ανωτέρου επιπέδου όπως τα HTTP και FTP, το SMTP και το Bittorrent. Ο λόγος που προτιμάται το TCP είναι επειδή, στηριζόμενο μόνο στις βασικές λειτουργίες που παρέχει το IP, προσφέρει στα παραπάνω στρώματα μια υπηρεσία αξιόπιστης μεταφοράς δεδομένων. Η αξιοπιστία αυτή διατηρείται ακόμα και όταν υπάρχει απώλεια πακέτων ή πολλαπλή μετάδοση του ίδιου πακέτου, τροποποίηση των περιεχομένων του πακέτου κατά τη μετάδοση ή παράδοση πακέτων εκτός σειράς.

Η επικεφαλίδα ενός πακέτου TCP φαίνεται στο σχήμα 1.3.



Σχήμα 1.3: Επικεφαλίδα TCP

Η θύρα προέλευσης και η θύρα προορισμού χρησιμοποιούνται όπως και στο UDP για πολύπλεξη / αποπολύπλεξη.

Το πεδίο sequence number (αριθμός ακολουθίας προσδιορίζει τη θέση των δε-

δομένων του πακέτου στη ροή δεδομένων ενώ ο αριθμός επιβεβαίωσης (acknowledgement number) είναι ο επόμενος αριθμός ακολουθίας που αναμένεται από το άλλο άκρο.

Το πεδίο data offset είναι το πλήθος των λέξεων μήκους 32 bit που καταλαμβάνει η επικεφαλίδα.

Το πεδίο window προσδιορίζει τον διαθέσιμο χώρο στον ενταμιευτή λήψης, με άλλα λόγια την διαφορά του μέγιστου αριθμού ακολουθίας που πρόκειται να δεχθεί ο παραλήπτης από τον τρέχοντα αριθμό επιβεβαίωσης.

Το πεδίο του αθροίσματος ελέγχου υπολογίζεται σε όλο το πακέτο (περιλαμβανομένης της επικεφαλίδας και μιας ψευδοεπικεφαλίδας όπως αυτή του σχήματος 1.2), με τον ίδιο αλγόριθμο όπως για το ICMP.

Ο δείκτης στα επείγοντα δεδομένα προσδιορίζει το byte που θα παραδοθεί σαν επείγον.

Το πεδίο reserved πρέπει να είναι πάντα μηδέν.

Τέλος, υπάρχουν οι επιλογές μήκους ενός bit:

URG Το πεδίο urgent pointer είναι έγκυρο.

ACK Το πεδίο acknowledgement number είναι έγκυρο.

PSH Τα δεδομένα πρεπει να παραδοθούν χωρίς καθυστέρηση στη εφαρμογή.

RST Διακοπή της σύνδεσης

SYN Έναρξη της σύνδεσης, συγχρονισμός αριθμών ακολουθίας.

FIN Ο αποστολέας δηλώνει ότι δεν πρόκειται να στείλει άλλα δεδομένα.

Δεν είναι έγκυροι όλοι οι συνδυασμοί αυτών των επιλογών και ο χειρισμός άκυρων επιλογών μπορεί να διαφέρει σε κάθε υλοποίηση. Περισσότερες λεπτομέρειες μπορούν να βρεθούν στο RFC793 ή στο [STEVEN94].

Η επικεφαλίδα TCP μπορεί να ακολουθείται από επιλογές μεταβλητού μήκους.

Δεν μας ενδιαφέρουν άμεσα οι περισσότερες πλευρές της δυναμικής συμπεριφοράς του TCP για αυτή την εργασία, θα αναφερθούμε ωστόσο στους χρονομετρητές του TCP οι οποίοι αλληλεπιδρούν ασύγχρονα με την επεξεργασία των πακέτων.

1.6.1 Χρονομετρητής καθυστερημένης επιβεβαίωσης

Παρόλο που το TCP οφείλει να αποστέλει μια επιβεβαίωση όταν λαμβάνει δεδομένα, η ενδεικνύμενη συμπεριφορά είναι να καθυστερήσει την αποστολή της επιβεβαίωσης για ένα μικρό χρονικό διάστημα με την ελπίδα ότι θα ληφθούν νέα δεδομένα οπότε θα μπορέσει να στείλει μια σωρευτική επιβεβαίωση. Για αυτό, χρησιμοποιεί των χρονομετρητή καθυστερημένης επιβεβαίωσης κατά την εκπνοή του οποίου αποστέλεται ένα πακέτο επιβεβαίωσης ώστε να συνεχίσει απρόσκοπτα η ροή δεδομένων από τον αποστολέα.

1.6.2 Χρονομετρητής επιμονής

Όταν το ένα άκρο του TCP δεν έχει τη δυνατότητα να δεχτεί επιπλέον πακέτα, προσδιορίζει μηδενικό μέγεθος παραθύρου. Πιθανώς σε ένα επόμενο πακέτο επιβεβαίωσης θα ανοίξει πάλι το παράθυρο, όμως οι επιβεβαιώσεις είναι δυνατό να χαθούν. Για να εξασφαλίσει ομαλή λειτουργία σε αυτή την περίπτωση, το TCP του αποστολέα χρησιμοποιεί τον χρονομετρητή επιμονής ώστε να στέλνει ένα πακέτο διερεύνησης παραθύρου (window probe) σε τακτά χρονικά διαστήματα.

1.6.3 Χρονομετρητής επαναμετάδοσης

Μετά την αποστολή δεδομένων,το TCP εκκινεί έναν χρονομετρητή, ώστε να επαναμεταδώσει τα δεδομένα αν αυτά δεν επιβεβαιωθούν ύστερα από ένα εύλογο χρονικό διάστημα. Το διάστημα αυτό πρέπει σίγουρα να είναι μεγαλύτερο από το χρόνο διαδρομής μετ'επιστροφής ανάμεσα στα επικοινωνούντα άκρα και η εκτίμηση του τελευταίου είναι ένα σημαντικό ζήτημα σε κάθε υλοποίηση.

1.6.4 Χρονομετρητής 2MSL

Ο χρόνος MSL (Maximum Segment Lifetime) είναι ο μέγιστος χρόνος κατά τον οποίο ένα τεμάχιο TCP μπορεί να κυκλοφορεί στο δίκτυο και συνήθως επιλέγεται αυθαίρετα από κάθε υλοποίηση. Στην περίπτωση που το ένα άκρο TCP έχει κλείσει το δικό του άκρο της σύνδεσης (δηλαδή δεν πρόκειται να στείλει άλλα δεδομένα) και λάβει ένα πακέτο με το οποίο το άλλο άκρο κλείνει το άλλο μέρος (ένα πακέτο με την επιλογή FIN), τότε στέλνει αμέσως μία επιβεβαίωση. Σε αυτό το σημείο η σύνδεση είναι ουσιαστικά κλειστή, το TCP όμως οφείλει να κρατήσει δεσμευμένη

την τοπική πόρτα για χρόνο ίσο με 2 * MSL. Έτσι, αν χαθεί η επιβεβαίωση του πακέτου με την επιλογή FIN, το άλλο άκρο θα το ξαναστείλει και έτσι το TCP θα έχει την ευκαιρία να επαναμεταδώσει την επιβεβαίωσή του.

1.6.5 Χρονομετρητής διατήρησης σύνδεσης

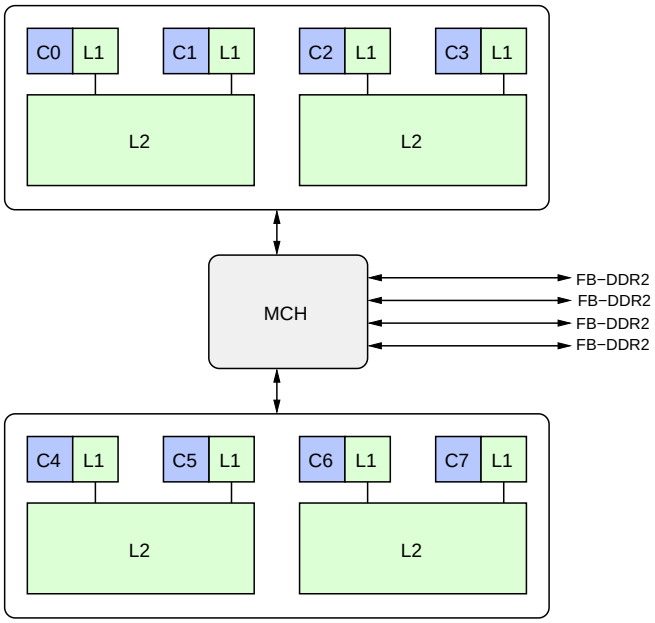
Το RFC793 δεν απαιτεί ροή δεδομένων για να θεωρείται μια σύνδεση έγκυρη. Ωστόσο, αν δεν ρέουν δεδομένα, δεν είναι δυνατή η ανίχνευση συνθηκών σφάλματος, όπως π.χ. ο ανώμαλος τερματισμός κάποιου από τα άκρα. Κάποιες υλοποιήσεις επιλέγουν να απαλλάξουν τις εφαρμογές από την ευθύνη παρακολούθησης της κατάστασης της σύνδεσης και για αυτό χρησιμοποιούν το χρονομετρητή διατήρησης σύνδεσης, στην εκπνοή του οποίου, αν η σύνδεση έχει μείνει ανενεργή για μεγάλο χρονικό διάστημα, στέλνουν ένα πακέτο ανίχνευσης στο άλλο άκρο και τερματίζουν τη σύνδεση αν δε λάβουν απάντηση. Το DragonFlyBSD διαθέτει έναν τέτοιο χρονομετρητή.

Κεφάλαιο 2

Μηχανισμοί εκμετάλλευσης πολλών επεξεργαστών

Παρότι ο αριθμός των τρανζίστορ που είναι οικονομικά εφικτό να τοποθετηθούν σε ένα chip συνεχίζει να διπλασιάζεται κάθε δύο χρόνια περίπου, σε συμφωνία με το νόμο του Moure, η βελτίωση της επίδοσης μέσω της αύξησης της συχνότητας ρολογιού του επεξεργαστή έχει πάψει πλέον να είναι εφικτή. Αντί για αυτό, οι κατασκευαστές παράγουν πλέον επεξεργαστές με πολλούς πυρήνες επεξεργασίας. Μια άλλη τεχνική που χρησιμοποιείται είναι η πολυνημάτωση (Simultaneous Multi Threading), κατά την οποία το ο επεξεργαστής παρουσιάζει στο λογισμικό περισσότερους λογικούς πυρήνες και εναλλάσει την εκτέλεση των διαθέσιμων νημάτων στους φυσικούς πυρήνες. Η πολυνημάτωση εκμεταλλεύεται την καθυστέρηση ενός νήματος κατά την πρόσβαση στη μνήμη για να εκτελέσει ένα άλλο νήμα στον επεξεργαστικό πυρήνα. Τελικά, ένα σύστημα μπορεί να υποστηρίζει περισσότερους από έναν επεξεργαστές, καθένας από τους οποίους μπορεί να παρέχει πολλούς πυρήνες επεξεργασίας, σε κάθε έναν από τους οποίους μπορεί να εκτελούνται με τη χρήση της πολυνημάτωσης δύο ή περισσότερα νήματα.

Στο σχήμα 2.1 φαίνεται η αρχιτεκτονική ενός μοντέρνου συστήματος. Το σύστημα έχει δύο επεξεργαστές, με τέσσερεις πυρήνες επεξεργασίας στον καθένα. Υπάρχουν δύο επίπεδα κρυφής μνήμης. Το πρώτο (L1) είναι τοπικό σε κάθε πυρήνα, ενώ το δεύτερο (L2) είναι κοινό σε κάθε ζευγάρι πυρήνων. Οι επεξεργαστές έχουν πρόσβαση στην κεντρική μνήμη μέσω του Memory Controller Hub (MCH).



Σχήμα 2.1: Αρχιτεκτονική ενός συστήματος με πολλούς πυρήνες επεξεργασίας

2.1 Ζητήματα ορθότητας

2.1.1 Αμοιβαίος αποκλεισμός

Σε ένα σύστημα με πολλούς επεξεργαστές ανακύπτουν προβλήματα τα οποία δεν υπάρχουν ή έχουν περιορισμένες επιπτώσεις σε συστήματα με έναν επεξεργαστή. Ας υποθέσουμε δύο ή περισσότερα προγράμματα με πρόσβαση σε κοινή μνήμη όπου βρίσκονται αποθηκευμένες δομές δεδομένων (συνθήκη που ικανοποιούν τα προγράμματα που εκτελούν κλήσεις συστήματος αφού στο DragonFlyBSD, όπως στα περισσότερα λειτουργικά συστήματα γενικής χρήσης, όλη η περιοχή διευθύνσεων (address space) του πυρήνα είναι προσβάσιμη στη διάρκεια μιας κλήσης συστήματος). Αν δύο νήματα προσπαθήσουν π.χ. να αυξήσουν την τιμή της μεταβλητής x κατά ένα ταυτόχρονα, τότε είναι δυνατόν να χαθεί η μία αύξηση.

Για περίπλοκες δομές δεδομένων, οι πιθανότητες να καταλήξει η δομή σε μη συνεπή κατάσταση είναι πολύ μεγαλύτερες. Αυτό είναι ένα πρόβλημα που απαντάται και σε συστήματα με έναν επεξεργαστή όταν ρουτίνες εξυπηρέτησης διακοπής μοιράζονται δεδομένα με τον υπόλοιπο πυρήνα. Η προσωρινή απενεργοποίηση

Νήμα Α	Νήμα Β
ανάκτηση x, r r = r + 1 αποθήκευση r, x	ανάκτηση x, r διακοπή r = r +1 αποθήκευση r, x

των διακοπών (που ήταν για χρόνια η ενδεδειγμένη αντιμετώπιση) δεν είναι αποδεκτή στις μέρες μας γιατί έχει αρνητική επίπτωση στην απόδοση του συστήματος.

Η πιο διαδεδομένη προσέγγιση στα μοντέρνα λειτουργικά συστήματα είναι η χρήση κλειδωμάτων (locks) για αμοιβαίο αποκλεισμό. Ένα κλείδωμα μπορεί να βρίσκεται σε μια από τις εξής δύο καταστάσεις: κλειδωμένο ή ελεύθερο. Λέμε ότι ένα νήμα κατέχει το κλείδωμα ή, ισοδύναμα, ότι το κλείδωμα ανήκει στο νήμα,αν το νήμα ήταν εκείνο που το κλείδωσε. Η ενέργεια του κλειδώματος υλοποιείται με την χρήση ειδικών εντολών που παρέχουν όλες οι σύγχρονες αρχιτεκτονικές και οι οποίες εξασφαλίζουν ότι ο έλεγχος και η τροποποίηση μιας θέσης μνήμης θα γίνουν αδιαίρετα (ατομικά). Έτσι όταν λέμε ότι μια δομή δεδομένων προστατεύεται από ένα κλείδωμα, αυτό σημαίνει ότι απαιτούμε ένα νήμα να τροποποιεί ή να προσπελαύνει εκείνη τη δομή μόνο εφόσον του ανήκει το κλείδωμα το οποίο την προστατεύει.

Το νήμα μπορεί να κατέχει το κλείδωμα μόνο για ανάγνωση, οπότε δεν δικαιούται να τροποποιήσει τα δεδομένα της δομής, ή για εγγραφή και ανάγνωση. Κάποιες υλοποιήσεις κλειδωμάτων επιτρέπουν την αναβάθμιση από ανάγνωση σε εγγραφή/ανάγνωση. Όταν ένα νήμα κατέχει ένα κλείδωμα για ανάγνωση, επιτρέπεται και σε άλλα νήματα να κάνουν το ίδιο. Αυτό εγείρει σοβαρά ζητήματα για το πως θα επιτευχθεί η δικαιοσύνη απέναντι στα νήματα που ζητούν το κλείδωμα για εγγραφή και κάθε πραγματική υλοποίηση θα πρέπει να φροντίζει οι εγγραφές να μην καθυστερούν επ'αόριστο όταν υπάρχουν πολλά νήματα που κάνουν ανάγνωση.

Τύποι κλειδωμάτων

Ένα ζήτημα που τίθεται, είναι η συμπεριφορά ενός νήματος που χρειάζεται πρόσβαση σε ένα κλείδωμα που ανήκει(για ανάγνωση/εγγραφή)σε άλλο νήμα. Υπάρχουν δύο βασικές επιλογές: είτε το νήμα θα συνεχίσει να προσπαθεί να απο-

κτήσει το κλείδωμα έως ότου το επιτύχει, είτε θα μπλοκάρει για να το αφυπνίσει ο πυρήνας όταν το κλείδωμα ελευθερωθεί. Στην πρώτη περίπτωση, το νήμα σπαταλά τον χρόνο του επεξεργαστή ο οποίος θα μπορούσε στο μεταξύ να τρέξει κάποιο άλλο νήμα που θα εκτελούσε χρήσιμη εργασία. Από την άλλη, αν το νήμα μπλοκάρει και το κλείδωμα ελευθερωθεί αμέσως μετά, τότε το νήμα θα καθυστερήσει αχρείαστα για ένα μεγάλο συγκριτικά χρονικό διάστημα. Έγκειται στον προγραμματιστή να αποφασίσει ποια κλειδώματα θα επιβάλλουν την πρώτη συμπεριφορά (κλειδώματα τύπου spin) και ποια τη δεύτερη (κλειδώματα τύπου sleep). Σε ορισμένα λειτουργικά συστήματα, όπως και στο DragonFlyBSD, υπάρχει η υβριδική προσέγγιση στην οποία κώδικας που ζητά ένα κλείδωμα που ανήκει σε άλλο νήμα εξακολουθεί την προσπάθεια να το αποκτήσει για ένα μικρό χρονικό διάστημα πριν μπλοκάρει.

Κατά καιρούς έχουν προταθεί εξειδικευμένοι τύποι κλειδωμάτων που είναι καταλληλότεροι για συγκεκριμένες περιστάσεις. Για παράδειγμα, ο τύπος κλειδώματος seqlock υποθέτει ότι οι αναγνώσεις είναι τάξεις μεγέθους πιο συχνές από τις εγγραφές. Ένα νήμα που κάνει εγγραφή αυξάνει ένα συσχετισμένο μετρητή και οι αναγνώστες απλά ελέγχουν αν ο μετρητής έχει την ίδια τιμή πριν και μετά την ανάγνωση. Αν οι τιμές διαφέρουν τότε ο αναγνώστης απλά προσπαθεί ξανά μέχρι να επιτύχει.

Big Giant Lock

Όταν τα συστήματα πολλών επεξεργαστών (και κοινής μνήμης) άρχισαν να γίνονται προσιτά, ήταν επιτακτικό τα τότε λειτουργικά συστήματα να τροποποιηθούν για να εκμεταλλευτούν πολλούς επεξεργαστές, έτσι ώστε να μπορούν να χρησιμοποιηθούν οι υπάρχουσες εφαρμογές. Η πρώτη λύση που δόθηκε ήταν η δημιουργία ενός μόνο κλειδώματος το οποίο προστάτευε όλο τον πυρήνα. Το πρώτο βήμα σε κάθε κλήση συστήματος ήταν η απόκτηση αυτού του κλειδώματος. Αυτό είχε σαν συνέπεια ότι μόνο ένα νήμα κάθε φορά θα μπορούσε να εκτελεί κώδικα του πυρήνα. Από την άλλη, εφαρμογές που δεν χρειάζονταν συχνά τις υπηρεσίες του πυρήνα π.χ. επειδή εκτελούν κυρίως αριθμητικούς υπολογισμούς, θα μπορούσαν να επιταχυνθούν με τη χρήση των επιπλέον επεξεργαστών. Η έξοδος από τον πυρήνα ή το μπλοκάρισμα συνεπαγόταν την αυτόματη ελευθέρωση αυτού του κεντρικού κλειδώματος. Αυτή ήταν η προσέγγιση που ακολουθούνταν μέχρι και την τέταρτη έκδοση του λειτουργικού συστήματος FreeBSD και αυτή εν

πολλοίς κληρονόμησε το DragonFlyBSD. Στον πυρήνα του DragonFlyBSD αυτό το κλείδωμα ονομάζεται Big Giant Lock (BGL) ή πιο απλά Giant. Είναι προφανές ότι για να παρατηρήσουμε κάποια επιτάχυνση σε εφαρμογές που εισέρχονται συχνά στον πυρήνα (π.χ. για είσοδο/έξοδο δεδομένων) πρέπει να καταργήσουμε ή να περιορίσουμε την χρήση της BGL, που είναι και το αντικείμενο αυτής της εργασίας.

Προβλήματα των κλειδωμάτων

Ενώ τα κλειδώματα επιτρέπουν αρκετά γρήγορη πρόσβαση σε μοιραζόμενες δομές δεδομένων, έχουν και αυτά τα δικά τους προβλήματα. Ένα βασικό πρόβλημα είναι το πρόβλημα του αδιεξόδου(deadlock) όταν κάποιο νήμα(A) χρειάζεται να κατέχει περισσότερα από ένα κλειδώματα για να κάνει πρόοδο και κατέχει μόνο κάποια από αυτά. Αν κάποιο νήμα (B) κατέχει κάποια από τα υπολειπόμενα κλειδώματα, αλλά χρειάζεται κάποιο από τα κλειδώματα που κατέχει το A για να κάνει πρόοδο, τα νήματα θα μείνουν μπλοκαρισμένα για πάντα. Ακόμα το πρόβλημα μπορεί να συμβεί με περισσότερα από δυο νήματα. Σε λειτουργικά συστήματα με χιλιάδες στιγμιότυπα κλειδωμάτων, όπως συμβαίνει στις μέρες μας, δεν είναι καθόλου εύκολο να αποδειχθεί ότι δεν υπάρχει το ενδεχόμενο του αδιεξόδου.

Ένα άλλο πρόβλημα είναι ότι ενώ το μεγάλο πλήθος κλειδωμάτων διευκολύνει την κλιμάκωση προς τα άνω, παρεμποδίζει την κλιμάκωση προς τα κάτω. Πράγματι, σε ένα σύστημα με δυο επεξεργαστές, η ανάγκη για την κατοχή δεκάδων κλειδωμάτων αποτελεί καθαρή επιβάρυνση, γιατί μπορεί να υπάρχει μόνο ένα άλλο νήμα σε εκτέλεση και η πιθανότητα να υπάρχει σύγκρουση στα δεδομένα που προσπελαύνονται είναι αρκετά μικρή. Οπότε θα βελτιώναμε ίσως την συνολική απόδοση του συστήματος χρησιμοποιώντας έναν μικρότερο αριθμό κλειδωμάτων.

2.1.2 RCU

Μια επιλογή εναλλακτική στα κλειδώματα που εξασφαλίζει την συνέπεια μιας δομής δεδομένων χωρίς να επιβάλλει αμοιβαίο αποκλεισμό ανάμεσα στα νήματα που διαβάζουν τα δεδομένα της δομής και σε αυτά που τα τροποποιούν, είναι η τεχνική Read – Copy update (ή RCU) [MCKENN04]. Μια δομή που προσπελαύνεται με τους κανόνες που ορίζει το RCU μπορεί να χρησιμοποιείται ταυτόχρονα για ανάγνωση από οποιοδήποτε αριθμό νημάτων και ένα νήμα το οποίο κάνει εγ-

γραφή.

Τα δεδομένα που προστατεύονται από το RCU προσπελαύνονται αποκλειστικά μέσω ενός δείκτη. Ένα νήμα που επιθυμεί να τα τροποποιήσει, δημιουργεί ένα αντίγραφο με τις αλλαγές του και απλά αντικαθιστά (ατομικά) τον δείκτη. Έτσι, οι αναγνώστες που χρησιμοποιούν τα παλαιά δεδομένα μπορούν να συνεχίσουν να εργάζονται με ασφάλεια. Το RCU θέτει τον περιορισμό ότι οι αναγνώστες δεν μπορούν να μπλοκάρουν όσο προσπελαύνουν τη δομή. Ο περιορισμός αυτός σημαίνει ότι όταν κάθε επεξεργαστής του συστήματος έχει αλλάξει το τρέχον νήμα του, είναι βέβαιο ότι όλες οι αναγνώσεις έχουν περατωθεί και είναι πλέον δυνατό να αποδεσμευτούν τα παλαιά δεδομένα. Η υλοποίηση του RCU πρέπει να κάνει εκτεταμένη χρήση φραγών μνήμης (memory barriers) για να εξασφαλίσει την ορθότητα των ενεργειών.

Η σύντομη περιγραφή που δώσαμε ανταποκρίνεται στην πιο διαδεδομένη μορφή RCU. Υπάρχουν επίσης παραλλαγές που στοχεύουν σε συστήματα με χιλιάδες επεξεργαστές (Hierarchical RCU), σε συστήματα πραγματικού χρόνου (real – time) ή που χαλαρώνουν τον περιορισμό που αφορά το μπλοκάρισμα των αναγνωστών (SRCU [MCKENN08] και preemptable RCU [MCKENN07]).

2.1.3 Τοπικά δεδομένα

Ένας άλλος τρόπος για την αποφυγή του αμοιβαίου αποκλεισμού (ο οποίος μειώνει την δυνατότητα παράλληλης εκτέλεσης όταν υπάρχει ανταγωνισμός για ένα κλείδωμα) είναι η χρήση τοπικών δεδομένων. Τα τοπικά δεδομένα προσπελαύνονται μόνο από έναν επεξεργαστή οπότε αρκεί ένα κρίσιμο τμήμα το οποίο εξασφαλίζει ότι ο επεξεργαστής δεν θα διακόψει την εκτέλεση του νήματος που προσπελαύνει τα τοπικά δεδομένα για να τρέξει κάποιο άλλο νήμα.

Συνήθως πραγματοποιείται η τεμάχιση μιας δομής σε τοπικά κομμάτια δεδομένων που μπορούν να προσπελαστούν αποδοτικά και παρέχεται ένας λιγότερο αποδοτικός μηχανισμός για συγχρονισμένη πρόσβαση σε αυτά.

Παραδείγματος χάριν, στο Dragonfly BSD ένας επεξεργαστής που χρειάζεται να τροποποιήσει δεδομένα που είναι τοπικά («ανήκουν») σε κάποιον άλλον επεξεργαστή, συνήθως το επιτυγχάνει με το να προκαλέσει μια διακοπή η οποία θα εκτελέσει την ενέργεια που απαιτείται τρέχοντας στον επεξεργαστή στον οποίο ανήκουν τα δεδομένα.

2.2 Ζητήματα επίδοσης

Γνωρίζουμε ότι οι επεξεργαστές χρησιμοποιούν πολυεπίπεδες κρυφές μνήμες για να γεφυρώσουν τη διαφορά στην ταχύτητα ανάμεσα στους επεξεργαστές και την κεντρική μνήμη (RAM). Η πρόσβαση στην κρυφή μνήμη γίνεται σε ομάδες λέξεων που καλούνται γραμμές κρυφής μνήμης (cache lines). Σε ένα σύστημα με πολλούς επεξεργαστές, είναι απαραίτητα τα περιεχόμενα της κρυφής μνήμης κάθε επεξεργαστή να βρίσκονται σε συμφωνία με τους υπολοίπους επεξεργαστές του συστήματος. Όταν υπάρχουν μόνο αναγνώσεις πολλοί επεξεργαστές μπορεί να έχουν μια γραμμή κρυφής μνήμης με τα περιεχόμενα της ίδιας διεύθυνσης κεντρικής μνήμης. Όταν όμως κάποιος επεξεργαστής γράφει στη δική του γραμμή πρέπει να σιγουρευτούμε ότι οι αναγνώσεις των υπόλοιπών επεξεργαστών του συστήματος για την αντίστοιχη διεύθυνση κεντρικής μνήμης θα επιστρέψουν τα νέα δεδομένα και όχι τα περιεχόμενα της δικής του γραμμής κρυφής μνήμης για αυτή την διεύθυνση.

Οι επεξεργαστές, που συνδέονται με την κύρια μνήμη μέσω διαδρόμου, χρησιμοποιούν συνήθως πρωτόκολλα κατασκοπείας (snooping) για την υλοποίηση των μηχανισμών συνοχής (coherency). Το πιο συνηθισμένο τέτοιο πρωτόκολλο είναι το MESI (Modified, Exclusive, Shared, Invalid), στο οποίο μια γραμμή κρυφής μνήμης μπορεί να βρίσκεται σε μια από τις τέσσερις καταστάσεις που περιγράφονται στον πίνακα 2.2.

Οι γραμμές κρυφής μνήμης ξεκινάνε στη κατάσταση Invalid (άκυρη). Αν κάποιος επεξεργαστής (A) κάνει μια ανάγνωση από μια διεύθυνση μνήμης και δεν υπάρχει γραμμή κρυφής μνήμης γι' αυτό τη διεύθυνση σε άλλον επεξεργαστή γραμμή της κρυφής μνήμης του A φορτώνεται με τα περιεχόμενα της κεντρικής διεύθυνσης μνήμης και μεταβαίνει στην κατάσταση Exclusive. Αν υπάρχει γραμμή κρυφής μνήμης σε άλλον επεξεργαστή στην κατάσταση Shared, τότε και στον A η γραμμή φορτώνεται στην ίδια κατάσταση. Όταν ο επεξεργαστής A θέλει να γράφει σε μια διεύθυνση μνήμης στέλνει ένα μήνυμα Request for Ownership (RFO) στους υπόλοιπους επεξεργαστές και φορτώνει τη μνήμη στην κατάσταση Modified. Στους υπόλοιπους επεξεργαστές οι αντίστοιχες γραμμές κρυφής μνήμης μεταβαίνουν αναγκαστικά στην κατάσταση Invalid. Αν, ακολούθως κάποιος επεξεργαστής B, χρειάζεται να διαβάσει από την ίδια διεύθυνση μνήμης, τα δεδομένα πρέπει να σταλούν από τον A, και η γραμμή μεταβαίνει στην κατάσταση Shared τόσο στον A όσο και στον B, κάτι που συνεπάγεται μετρήσιμη καθυστέρηση. Είναι φανερό,

Κατάσταση	Περιγραφή
Modified (Τροποποιημένη)	Αν η γραμμή είναι παρούσα στην κρυφή μνήμη του επεξεργαστή και διαφέρει από τα περιεχόμενα της αντίστοιχης διεύθυνσης κεντρικής μνήμης. Σε όλους τους άλλους επεξεργαστές η κατάσταση της αντίστοιχης γραμμής πρέπει να είναι Invalid.
Exclusive (Αποκλειστική)	Αν η γραμμή είναι παρούσα στην κρυφή μνήμη μόνο αυτού του επεξεργαστή και τα περιεχόμενά της δεν διαφέρουν από αυτά της αντίστοιχης διεύθυνσης μνήμης.
Shared (Μοιρασμένη)	Αν τα περιεχόμενα της γραμμής ταυτίζονται με αυτά της αντίστοιχης διεύθυνσης κεντρικής μνήμης και η γραμμή πιθανώς υπάρχει και στην κρυφή μνήμη άλλου επεξεργαστή (αν υπάρχει, θα είναι επίσης στην κατάσταση Shared.
Invalid (Άκυρη)	Αν η γραμμή δεν είναι πλέον αντιστοιχισμένη με κάποια διεύθυνση μνήμης.

ότι αν έχουμε μια γραμμή μνήμης την οποία ο Α, γράφει και ο Β διαβάζει, τότε τα δεδομένα της θα πρέπει να μεταφέρονται συνεχώς από τον Α στον Β (κάτι που θα προσπαθήσουμε να αποφύγουμε για τη δομή sockbuf, βλ). Ακόμα χειρότερη είναι η περίπτωση στην οποία ο Α και ο Β χρειάζονται πρόσβαση σε διαφορετικές λέξεις της γραμμόης, γιατί τότε η γραμμή θα πρέπει να μεταφέρεται συνεχώς μεταξύ τους (cacheline ping – pong) ενώ με καλύτερη διευθέτηση στη μνήμη θα μπορούσαμε να αποφύγουμε εντελώς το πρόβλημα (που είναι γνωστό σαν False shaving). Στο πρωτόκολλο MESI, ο ελεγκτής μνήμης παρακολουθεί τις μεταβιβάσεις καταστάσεων και φροντίζει να κρατά επίκαιρα τα δεδομένα της κεντρικής μνήμης. Νεότεροι επεξεργαστές χρησιμοποιούν επιπλέον την κατάσταση. Νεότεροι επεξεργαστές χρησιμοποιούν επιπλέον την κατάσταση Owned στην οποία ο ελεγκτής μνήμης δεν υποχρεούται να ενημερώσει την κεντρική μνήμη. Αν ένας επεξεργαστής έχει μια γραμμή κρυφής μνήμης στην κατάσταση Owned, αντίγραφά της σε άλλους επεξεργαστές, αν υπάρχουν θα πρέπει να είναι στην κατάσταση Shared.

Κεφάλαιο 3

Υλοποίηση διαδικτυακών πρωτοκόλλων

Θα δώσουμε τώρα μια σύντομη περιγραφή της υλοποίησης των πρωτοκόλλων με τα οποία ασχοληθήκαμε στο πλαίσιο αυτής της εργασίας. Η περιγραφή μας θα επικεντρωθεί στις κύριες δομές δεδομένων που χρησιμοποιεί η στοίβα των διαδικτυακών πρωτοκόλλων. Ακόμα, θα εξετάσουμε κάποια από τα βασικά μονοπάτια κώδικα για την αποστολή και λήψη δεδομένων ώστε να γίνουν ορατά τα κίνητρα πίσω από τις αλλαγές που πραγματοποιήσαμε.

Το λειτουργικό σύστημα DragonFlyBSD είναι μετεξέλιξη του λειτουργικού συστήματος 4.4BSD [MCKUSI96]. Η υλοποίηση των πρωτοκόλλων TCP/IP από το 4.4BSD αποτελεί κώδικα αναφοράς και περιγράφεται αναλυτικά στο βιβλίο [WRIGHT95]. Η στοίβα πρωτοκόλλων (network stack) διαδικτύου στο DragonFlyBSD έχει επεκταθεί με νέους αλγόριθμους και επιπλέον λειτουργικότητα, αλλά η υλοποίηση της βασικής λειτουργίας TCP/IP παραμένει σε μεγάλο βαθμό παρόμοια με αυτή που αναλύεται στο παραπάνω βιβλίο. Πριν περιγράψουμε την ίδια τη στοίβα πρωτοκόλλων, είναι χρήσιμο να εξετάσουμε της υπηρεσίες του πυρήνα που αυτή χρησιμοποιεί.

3.1 Ανταλλαγή μηνυμάτων

Ο πυρήνας του DragonFlyBSD και κατ'εξοχήν η στοίβα πρωτοκόλλων κάνουν χρήση της υποδομής για την ανταλλαγή μηνυμάτων. Η κύρια έννοια με την οποία

πρέπει να ασχοληθούμε είναι η θύρα μηνυμάτων (message port). Θύρα μηνυμάτων ονομάζεται η δομή στην οποία ένας παραλήπτης αναμένει μηνύματα και στην οποία οι αποστολείς παραδίδουν τα μηνύματά τους. Υπάρχουν αρκετοί τύποι θύρας μηνυμάτων στον πυρήνα, αλλά η στοίβα πρωτοκόλλων χρησιμοποιεί αποκλειστικά θύρες μηνυμάτων που ανήκουν η καθεμία σε ένα νήμα.

Ο κανόνας αποκλεισμού για μια τέτοια θύρα είναι ότι επιτρέπεται να τροποποιηθεί μόνο από κώδικα ο οποίος εκτελείται στον ίδιο επεξεργαστή με το νήμα στο οποίο ανήκει η θύρα. Αρκεί έτσι η είσοδος σε ένα κρίσιμο τμήμα (critical section, κάτι που στο DragonFlyBSD χρειάζεται ελάχιστους κύκλους ρολογιού), ώστε τυχόν διακοπές (interrupts) να μην είναι σε θέση να επιτρέψουν σε άλλο κώδικα να εκτελεστεί στον ίδιο επεξεργαστή, για να τροποποιηθεί ορθά η θύρα μηνυμάτων.

Αν ένα νήμα το οποίο εκτελείται σε άλλο επεξεργαστή επιθυμεί να παραδώσει ένα μήνυμα σε μία θύρα, οφείλει να ζητήσει την αποστολή ενός Inter-processor Interrupt (IPI, μια διακοπή που ζητά ένας επεξεργαστής από έναν άλλο), το οποίο θα εκτελέσει τον κώδικα που είναι υπεύθυνος για την παράδοση του μηνύματος στον επεξεργαστή που εκτελείται το νήμα της θύρας μηνυμάτων.

Όπως είναι γνωστό (βλ. π.χ. [SCHUEP08]), τα IPI παραδίδονται με σημαντική καθυστέρηση της τάξης των εκατοντάδων κύκλων ρολογιού. Για το λόγο αυτό, είναι σημαντικό να γίνεται μαζική παράδοση των IPI. Ο πυρήνας του DragonFlyBSD φροντίζει να μην ζητήσει μια πραγματική διακοπή IPI αν ο επεξεργαστής-παραλήπτης τύχει να εξυπηρετεί την ουρά με τα IPI που προορίζονται για αυτόν τη στιγμή που ο κώδικας σε έναν άλλο επεξεργαστή προσπαθήσει να στείλει ένα IPI. Επιπλέον, όπως θα δούμε πιο κάτω (ενότητα 3.9, τα κατώτερα στρώματα της στοίβας καταβάλλουν σημαντική προσπάθεια να αποφύγουν τη δημιουργία IPI ομαδοποιώντας τα πακέτα πριν την παράδοσή τους στο ανώτερο στρώμα.

3.2 mbufs

Όλα τα στρώματα της στοίβας πρωτοκόλλων του πυρήνα, όπως και οι οδηγοί των καρτών δικτύου, χρησιμοποιούν τις δομές mbuf (memory buffer) για να αποθηκεύουν τα δεδομένα και τις επικεφαλίδες των πακέτων. Η ιδιαιτερότητα που έχουν τα δεδομένα των πακέτων είναι ότι κάθε στρώμα προσθέτει, αφαιρεί ή τροποποιεί τις επικεφαλίδες. Εφόσον γνωρίζουμε ότι σε ένα πακέτο που πρόκειται να αποσταλεί θα επικολληθούν νέα δεδομένα στην αρχή του, είναι λογικό να φροντί-

σουμε ώστε να υπάρχει διαθέσιμος χώρος για να γίνει αυτό χωρίς να χρειαστεί να μετακινήσουμε τα δεδομένα. Το ίδιο ισχύει και όταν χρειάζεται να αφαιρεθούν δεδομένα από την αρχή του πακέτου.

Επιπροσθέτως, για να αποφύγουμε μια επιπλέον αντιγραφή των δεδομένων, είναι επιθυμητό να προγραμματίσουμε την κάρτα δικτύου ώστε να προσπελαύνει τα δεδομένα απευθείας μέσα στα mbuf. Ακόμα και οι μοντέρνες κάρτες δικτύου ωστόσο, μπορεί να έχουν περιορισμούς στις διευθύνσεις μνήμης που μπορούν να δεχθούν για την είσοδο/έξοδο δεδομένων με απευθείας προσπέλαση μνήμης (direct memory access, DMA). Επιπλέον, για μεγάλες MTU ίσως είναι βολικό να παρέχουμε στην κάρτα δικτύου αρκετούς μικρότερους ενταμιευτές (buffers). Μερικά στρώματα επίσης πιθανόν να μεγαλώσουν το πακέτο πέραν του χώρου που αρχικά έχει δεσμευτεί για αυτό. Σε αυτή την περίπτωση, καθώς και σε όλες τις προηγούμενες, είναι εξαιρετικά χρήσιμη η δυνατότητα να συνδέσουμε δυναμικά τις δομές mbuf ώστε ένα πακέτο να περιέχεται σε μία μονά συνδεδεμένη λίστα από mbuf.

Listing 3.1: struct mbuf

```
struct mbuf {
        struct  m_hdr m_hdr;
        union {
                struct {
                        struct  pkthdr MH_pkthdr;       /*
                           M_PKTHDR set */
                        union {
                                struct  m_ext MH_ext;   /* M_EXT
                                    set */
                                char    MH_databuf[MHLEN];
                        } MH_dat;
                } MH;
                char    M_databuf[MLEN];                /*
                    !M_PKTHDR, !M_EXT */
        } M_dat;
};
```

Listing 3.2: struct m_hdr

```
/*
 * Header present at the beginning of every mbuf.
```

Τύπος mbuf	Περιγραφή
MT_FREE	Μη δεσμευμένο
MT_DATA	Περιέχει δεδομένα
MT_HEADER	Περιέχει επικεφαλίδα πακέτου
MT_SONAME	Περιέχει ονομασία socket
MT_CONTROL	Περιέχει δεδομένα ελέγχου
MT_OOB	Περιέχει επισπευσμένα δεδομένα

Πίνακας 3.1: Τύποι mbuf

```
 */
struct m_hdr {
        struct  mbuf *mh_next;          /* next buffer in chain
            */
        struct  mbuf *mh_nextpkt;       /* next chain in
            queue/record */
        caddr_t mh_data;                /* location of data */
        int     mh_len;                 /* amount of data in
            this mbuf */
        int     mh_flags;               /* flags; see below */
        short   mh_type;                /* type of data in this
            mbuf */
        short   mh_pad;                 /* padding */
        struct netmsg_packet mh_netmsg; /* hardware >proto stack
             msg */
};
```

Η ανάγκη να εξυπηρετηθούν όλες αυτές οι απαιτήσεις οδήγησε στη σημερινή μορφή της δομής mbuf που φαίνεται στο απόσπασμα 3.1. Κάθε δομή mbuf περιέχει στην αρχή της την επικεφαλίδα m_hdr που φαίνεται στο απόσπασμα 3.2. Το πεδίο mh_next χρησιμοποιείται για να συνδέσει τα mbufs ενός πακέτου (σε μία όπως καλείται, “αλυσίδα mbuf”), όπως περιγράψαμε παραπάνω, ενώ με το πεδίο mh_nextpkt συνδέονται σε μια μονά συνδεδεμένη λίστα αλυσίδες από mbuf.

Το πεδίο mh_len περιέχει τον αριθμό των έγκυρων bytes δεδομένων που περιέχονται στο mbuf.

Τα δεδομένα που περιέχονται σε ένα mbuf έχουν έναν από τους τύπους που φαίνονται στον πίνακα 3.1.

Το πεδίο mh_flags μπορεί να περιέχει ένα συνδυασμό επιλογών κάθε μια από

Bit	Περιγραφή
M_EXT	Συνδέεται με εξωτερικό ενταμιευτή
M_PKTHDR	Αρχή πακέτου
M_EOR	Τέλος πακέτου
M_CLCACHE	Δεσμεύτηκε από την περιοχή CLCACHE
M_PHCACHE	Δεσμεύτηκε από την περιοχή PHCACHE
M_EXT_CLUSTER	Συνδέεται με εξωτερικό ενταμιευτή προκαθορισμένου τύπου
M_HASH	Το πεδίο hash της δομής pkthdr είναι έγκυρο

Πίνακας 3.2: Bit πεδίου mh_flags

M_PKTHDR	M_EXT	Έγκυρα πεδία
0	0	M_databuf[MLEN]
1	0	MH_pkthdr; MH_databuf[MHLEN];
0	1	MH_ext
1	1	MH_pkthdr MH_ext

Πίνακας 3.3: Συνδυασμοί M_PKTHDR και M_EXT

τις οποίες αντιστοιχεί σε ένα bit του πεδίου. Οι πιο σημαντικές επιλογές και κάποιες που σχετίζονται με τα προβλήματα που αντιμετωπίσαμε σε αυτή την εργασία φαίνονται στον πίνακα 3.2.

Η επικεφαλίδα του mbuf περιέχει ενσωματωμένο ένα μήνυμα (mh_netmsg) το οποίο αποστέλλεται σε ένα νήμα πρωτοκόλλου για να επεξεργαστεί το πακέτο, όπως θα εξηγηθεί αργότερα.

Το πεδίο mh_data είναι δείκτης στην αρχή των έγκυρων δεδομένων του mbuf.

Τέλος, υπάρχει ένα πεδίο mh_pad για λόγους ευθυγράμμισης του mh_netmsg.

Τα περιεχόμενα του mbuf μετά την επικεφαλίδα εξαρτώνται από τις τιμές των επιλογών M_PKTHDR και M_EXT. Η μορφή των υπόλοιπων πεδίων φαίνεται στον πίνακα 3.2 για κάθε συνδυασμό αυτών των bit.

Στη δομή pkthdr (απόσπασμα 3.3) περιέχονται οι πληροφορίες για ένα ολόκληρο πακέτο, όπως το μήκος του πακέτου (πεδίο len) και η διεπαφή στην οποία αυτό μεταδόθηκε. Για τους δικούς μας σκοπούς, ενδιαφέρον παρουσιάζει επίσης και το πεδίο hash, ο ρόλος του οποίου θα γίνει ξεκάθαρος στην ενότητα 3.3.

Listing 3.3: struct pkthdr

```
struct pkthdr {
        struct  ifnet *rcvif;           /* rcv interface */
        int     len;                    /* total packet length
            */
        struct packet_tags tags;        /* list of packet tags
            */

        /* variables for ip and tcp reassembly */
        void    *header;                /* pointer to packet
            header */

        /* variables for hardware checksum */
        int     csum_flags;             /* flags regarding
            checksum */
        int     csum_data;              /* data field used by
            csum routines */

        /* firewall flags */
        uint32_t fw_flags;              /* flags for PF */

        /* variables for PF processing */
        uint16_t pf_tag;                /* PF tag id */
        uint8_t pf_routed;              /* PF routing counter */

        /* variables for ALTQ processing */
        uint8_t ecn_af;                 /* address family for
            ECN */
        uint32_t altq_qid;              /* queue id */
        uint32_t altq_state_hash;       /* identifies
            'connections' */

        uint16_t ether_vlantag;         /* ethernet 802.1p+q
            vlan tag */
        uint16_t hash;                  /* packet hash */
};
```

Listing 3.4: struct m_ext

```
struct m_ext {
```

```
        caddr_t ext_buf;                /* start of buffer */
        void    (*ext_free)(void *);
        u_int   ext_size;               /* size of buffer, for
                ext_free */
        void    (*ext_ref)(void *);
        void    *ext_arg;
};
```

3.3 Νήματα πρωτοκόλλων

Ένα χαρακτηριστικό της στοίβας πρωτοκόλλων που υπήρχε από την πρώτη της υλοποίηση στο 4.2BSD και εξακολουθεί να χρησιμοποιείται στις μοντέρνες εκδοχές των FreeBSD και NetBSD είναι ότι όλες οι λειτουργίες των πρωτοκόλλων λαμβάνουν χώρα στο πλαίσιο μιας κλήσης συστήματος ή σε διακοπές λογισμικού (software interrupts) και χρονομετρητές.

Αυτό σημαίνει ότι κατά την αποστολή δεδομένων, το μονοπάτι ελέγχου ξεκινά από το στρώμα των sockets, περνά από τα διαδοχικά στρώματα πρωτοκόλλων και καταλήγει στην τοποθέτηση των mbuf με τα δεδομένα στην ουρά αποστολής της κατάλληλης κάρτας δικτύου. Ομοίως, για την παροχή μίας υπηρεσίας όπως η αποδοχή σύνδεσης ή ρύθμιση παραμέτρων, η κλήση συστήματος καταλήγει να εκτελεί κώδικα της στοίβας πρωτοκόλλων.

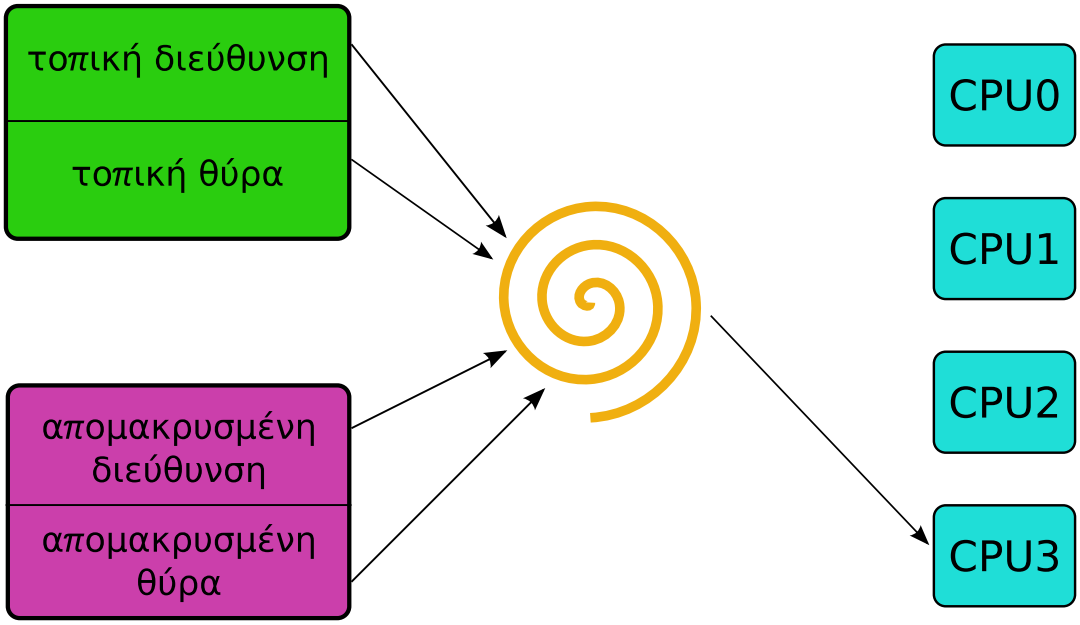
Αντίστροφα, όταν το στρώμα της κάρτας δικτύου ειδοποιηθεί για ένα εισερχόμενο πακέτο, το επεξεργάζεται κατάλληλα και φροντίζει για την εκτέλεση μιας διακοπής λογισμικού στην οποία θα εκτελεστεί ο κώδικας της στοίβας πρωτοκόλλων μέχρι την παράδοση των δεδομένων στο κατάλληλο socket. Από εκεί θα τα παραλάβει η εφαρμογή με την εκτέλεση της κατάλληλης κλήσης συστήματος για την ανάγνωση δεδομένων. Η κλήση αυτή θα ικανοποιηθεί στο στρώμα socket.

Αντίθετα, στο DragonFlyBSD η οργάνωση είναι αρκετά διαφορετική. Ο κώδικας των διαδικτυακών πρωτοκόλλων εκτελείται στο πλαίσιο νημάτων επεξεργασίας του πυρήνα. Τα πρωτόκολλα UDP και TCP δημιουργούν κατά την εκκίνησή τους ένα νήμα για κάθε επεξεργαστικό πυρήνα του συστήματος. Τούτο σημαίνει ότι οι κλήσεις συστήματος σταματούν τώρα στο στρώμα socket. Για να ζητήσει τις υπηρεσίες της στοίβας πρωτοκόλλων, μια κλήση συστήματος στέλνει ένα μήνυμα στο αντίστοιχο νήμα πρωτοκόλλου. Οι τύποι των μηνυμάτων είναι αυτοί που εμφανίζονται στον πίνακα 3.4.

Μηνύματα
abort
accept
attach
bind
connect
connect2
control
detach
disconnect
listen
peeraddr
notify
rcvd
rcvoob
send
sense
shutdown
sockaddr
poll

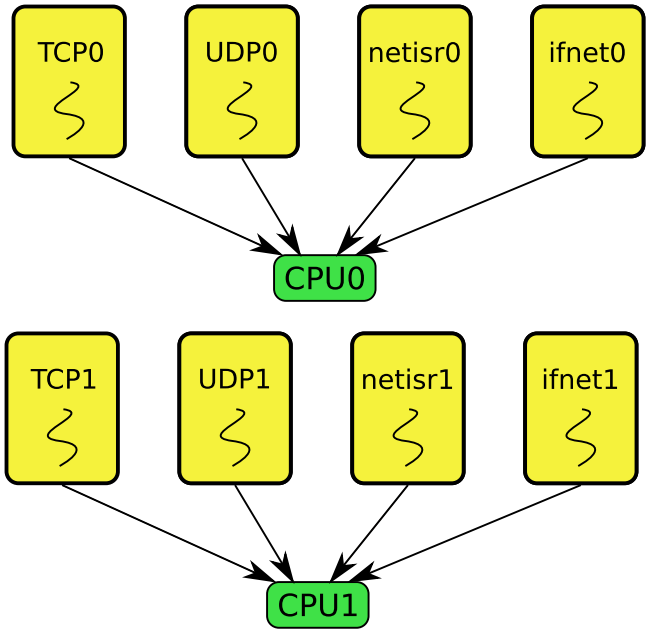
Πίνακας 3.4: Μηνύματα από κλήσεις συστήματος προς τα νήματα πρωτοκόλλου

Στην περίπτωση του πρωτοκόλλου TCP, κάθε σύνδεση διαχειρίζεται μόνο από ένα νήμα πρωτοκόλλου. Μία σύνδεση TCP αντιστοιχεί σε μία μοναδική (για το σύστημα) τετράδα <απομακρυσμένη διεύθυνση, απομακρυσμένη θύρα, τοπική διεύθυνση, τοπική θύρα>. Αναθέτουμε έναν επεξεργαστικό πυρήνα σε μια σύνδεση υπολογίζοντας μια συνάρτηση κατακερματισμού με είσοδο τα στοιχεία της τετράδας που αναφέραμε (σχήμα 3.1). Κατά αυτό τον τρόπο είναι εύκολο να βρούμε σε ποιόν επεξεργαστικό πυρήνα πρέπει να επεξεργαστεί ένα πακέτο που μόλις παραλήφθηκε. Αντίστροφα, κώδικας ο οποίος έχει πρόσβαση στα στοιχεία της σύνδεσης μπορεί εύκολα να βρει σε ποιο νήμα πρέπει να στείλει μήνυμα για να εξυπηρετηθεί.



Σχήμα 3.1: Αντιστοίχιση πακέτων σε επεξεργαστές με συνάρτηση κατακερματισμού

Πέρα από τα νήματα των πρωτοκόλλων UDP και TCP υπάρχει επίσης ένα νήμα (ονομάζεται netisr) ανά επεξεργαστικό πυρήνα που αναλαμβάνει τα υπόλοιπα πρωτόκολλα της στοίβας. Έτσι τα νήματα στα οποία τρέχει ο κώδικας της στοίβας πρωτοκόλλων για ένα σύστημα δύο επεξεργαστών είναι αυτά που φαίνονται στην εικόνα 3.2 (περιλαμβάνονται και τα νήματα ifnet του στρώματος της κάρτας δικτύου που εξετάζουμε στις ενότητες 3.8 και 3.9).



Σχήμα 3.2: Νήματα πρωτοκόλλων σε σύστημα με δύο επεξεργαστές

3.4 Internet Control Block

Κάθε πρωτόκολλο κρατά τα δεδομένα που χρειάζεται σε μία δομή που ανήκει στην οικογένεια των Protocol Control Blocks (δομές ελέγχου πρωτοκόλλων, PCB) και συνδέεται αμφίδρομα με τη δομή socket [STEVEN98]. Τόσο το πρωτόκολλο UDP όσο και το TCP χρησιμοποιούν τη δομή inpcb (InterNet Protocol Control Block) για να κρατήσουν τα δικά τους δεδομένα. Η δομή φαίνεται στο σχήμα 3.5.

Listing 3.5: struct in_pcb

```
struct inpcb {
        LIST_ENTRY(inpcb) inp_hash; /* hash list */
        LIST_ENTRY(inpcb) inp_list; /* list for all PCBs of this
             proto */
        u_int32_t       inp_flow;

        /* local and foreign ports, local and foreign addr */
        struct  in_conninfo inp_inc;

        void    *inp_ppcb;              /* pointer to
```

```
    per protocol pcb */
struct  inpcbinfo *inp_pcbinfo; /* PCB list info */
struct  inpcbinfo *inp_cpcbinfo;/* back pointer for
    connection table */
struct  socket *inp_socket;     /* back pointer to
    socket */
                                /* list for this PCB's
                                   local port */
int     inp_flags;              /* generic IP/datagram
    flags */

struct  inpcbpolicy *inp_sp; /* for IPSEC */
u_char  inp_vflag;
u_char  inp_ip_ttl;             /* time to live proto */
u_char  inp_ip_p;               /* protocol proto */
u_char  inp_ip_minttl;          /* minimum TTL or drop
    */

/* protocol dependent part; options */
struct {
        u_char  inp4_ip_tos;            /* type of
            service proto */
        struct  mbuf *inp4_options;     /* IP options */
        struct  ip_moptions *inp4_moptions; /* IP
            multicast options */
} inp_depend4;
struct {
        /* IP options */
        struct  mbuf *inp6_options;
        /* IP6 options for outgoing packets */
        struct  ip6_pktopts *inp6_outputopts;
        /* IP multicast options */
        struct  ip6_moptions *inp6_moptions;
        /* ICMPv6 code type filter */
        struct  icmp6_filter *inp6_icmp6filt;
        /* IPV6_CHECKSUM setsockopt */
        int     inp6_cksum;
        u_short inp6_ifindex;
        short   inp6_hops;
```

```
                u_int8_t        inp6_hlim;
        } inp_depend6;
        LIST_ENTRY(inpcb) inp_portlist;
        struct  inpcbport *inp_phd;     /* head of this list */
        inp_gen_t       inp_gencnt;     /* generation count of
           this instance */
};
```

Οι βασικές πληροφορίες είναι αποθηκευμένες στο πεδίο inp_inc που έχει τύπο δομής in_conninfo (βλ. σχ. 3.6). Εκεί περιέχονται η διεύθυνση και η θύρα του άλλου άκρου (αν το socket είναι συνδεδεμένο) και η τοπική διεύθυνση (αν είναι συγκεκριμένη) και η τοπική θύρα (το πεδίο αυτό είναι πάντα έγκυρο).

Listing 3.6: struct in_conninfo

```
struct in_conninfo {
        u_int8_t        inc_flags;
        u_int8_t        inc_len;
        u_int16_t       inc_pad;        /* XXX alignment for
           in_endpoints */
        struct  in_endpoints inc_ie;
        union {
                /* placeholder for routing entry */
                struct  route inc4_route;
                struct  route_in6 inc6_route;
        } inc_dependroute;
};
```

Listing 3.7: struct in_endpoints

```
struct in_endpoints {
        u_int16_t       ie_fport;               /* foreign port
           */
        u_int16_t       ie_lport;               /* local port */
        /* protocol dependent part, local and foreign addr */
        union {
                /* foreign host table entry */
                struct  in_addr_4in6 ie46_foreign;
                struct  in6_addr ie6_foreign;
        } ie_dependfaddr;
        union {
```

```
                /* local host table entry */
                struct  in_addr_4in6 ie46_local;
                struct  in6_addr ie6_local;
        } ie_dependladdr;
};
```

Listing 3.8: struct in_addr_4in6

```
struct in_addr_4in6 {
        u_int32_t       ia46_pad32[3];
        struct  in_addr ia46_addr4;
};
```

Η αποθήκευση των διευθύνσεων είναι περίπλοκη γιατί η δομή inpcb χρησιμοποιείται και από το πρωτόκολλο IP έκδοσης 6 στο οποίο οι διευθύνσεις έχουν μεγαλύτερο μήκος. Η αποθηκευμένη διεύθυνση είναι τύπου IP έκδοσης 4 αν το πεδίο inc_flags της δομής in_conninfo είναι μηδέν.

Το πεδίο inc_dependroute.inc4_route προσδιορίζει τη δρομολόγηση των πακέτων προς αποστολή.

Το πεδίο inp_ppcb είναι δείκτης σε ένα επιπλέον PCB αν αυτό απαιτείται (π.χ. για το TCP).

Όλες οι δομές inpcb ενός πρωτοκόλλου συνδέονται σε μία λίστα μέσω του πεδίου inp_list ώστε τα προγράμματα διαχείρισης να μπορούν να παρουσιάσουν πληροφορίες για όλες τις δομές socket. Επιπλέον, οι δομές inpcb τοποθετούνται σε πίνακες κατακερματισμού. Σε αυτή την περίπτωση, από κάθε πεδίο του πίνακα κατακερματισμού ξεκινά μια μονά συνδεδεμένη λίστα από δομές inpcb που συνδέονται με το πεδίο inp_hash. Το πεδίο του πίνακα κατακερματισμού στο οποίο ανήκει ένα PCB προκύπτει από τον υπολογισμό μιας συνάρτησης κατακερματισμού με είσοδο τα στοιχεία της σύνδεσης. Θα ασχοληθούμε περισσότερο με αυτό το θέμα σε επόμενη ενότητα όπου θα περιγράψουμε τα προβλήματα που υπάρχουν όταν το σύστημα διαθέτει πολλούς πυρήνες επεξεργασίας.

Επιπλέον, οι δομές inpcb ανήκουν σε ένα ακόμα πίνακα κατακερματισμού ο οποίος κάνει εύκολη την αναζήτησή τους με βάση την τοπική θύρα.

Τα υπόλοιπα πεδία δεν μας απασχόλησαν στο πλαίσιο αυτής της εργασίας.

3.5 TCP Control Block

Όπως είδαμε (ενότητα 1.6), το πρωτόκολλο TCP προσφέρει πολλές περισσότερες υπηρεσίες από το UDP και για αυτό η υλοποίησή του είναι αρκετά πιο πολύπλοκη. Το TCP κρατά σημαντικά περισσότερα δεδομένα για κάθε σύνδεση που συνάπτει. Τα δεδομένα αυτά είναι αποθηκευμένα σε μία δομή τύπου tcpcb (TCP Control Block) η οποία είναι αμφίδρομα συνδεδεμένη με την συμπληρωματική δομή τύπου inpcb μέσω των πεδίων t_inpcb και inp_ppcb. Τα πεδία της δομής tcpcb φαίνονται στο απόσπασμα 3.9.

Listing 3.9: struct tcpcb

```
struct tcpcb {
        struct  tsegqe_head t_segq;
        int     t_dupacks;              /* consecutive dup acks
           recd */
        int     tt_cpu;                 /* sanity check the cpu
           */

        struct  tcp_callout *tt_rexmt;  /* retransmit timer */
        struct  tcp_callout *tt_persist;/* retransmit
           persistence */
        struct  tcp_callout *tt_keep;   /* keepalive */
        struct  tcp_callout *tt_2msl;   /* 2*msl TIME_WAIT timer
            */
        struct  tcp_callout *tt_delack; /* delayed ACK timer */
        struct  netmsg_tcp_timer *tt_msg; /* timer message */

        struct  inpcb *t_inpcb;         /* back pointer to
           internet pcb */
        int     t_state;                /* state of this
           connection */
        u_int   t_flags;
        tcp_seq snd_up;                 /* send urgent pointer
           */

        tcp_seq snd_una;                /* send unacknowledged
           */
        tcp_seq snd_recover;            /* for use with NewReno
           Fast Recovery */
```

```
tcp_seq snd_max;                /* highest sequence
    number sent;
                                 * used to recognize
                                    retransmits */
tcp_seq snd_nxt;                /* send next */

tcp_seq snd_wl1;                /* window update seg seq
     number */
tcp_seq snd_wl2;                /* window update seg ack
     number */
tcp_seq iss;                    /* initial send sequence
     number */
tcp_seq irs;                    /* initial receive
    sequence number */

tcp_seq rcv_nxt;                /* receive next */
tcp_seq rcv_adv;                /* advertised window */
u_long  rcv_wnd;                /* receive window */
tcp_seq rcv_up;                 /* receive urgent
    pointer */

u_long  snd_wnd;                /* send window */
u_long  snd_cwnd;               /* congestion controlled
     window */
u_long  snd_wacked;             /* bytes acked in one
    send window */
u_long  snd_ssthresh;           /* snd_cwnd size
    threshold for
                                 * for slow start
                                    exponential to
                                 * linear switch */

int     t_rxtcur;               /* current retransmit
    value (ticks) */
u_int   t_maxseg;               /* maximum segment size
    */
int     t_srtt;                 /* smoothed round trip
    time */
int     t_rttvar;               /* variance in
```

```
            round trip time */

        u_int   t_maxopd;               /* mss plus options */

        u_long  t_rcvtime;              /* inactivity time */
        u_long  t_starttime;            /* time connection was
            established */
        int     t_rtttime;              /* round trip time */
        tcp_seq t_rtseq;                /* sequence number being
             timed */

        int     t_rxtshift;             /* log(2) of rexmt exp.
            backoff */
        u_int   t_rttmin;               /* minimum rtt allowed
            */
        u_int   t_rttbest;              /* best rtt we've seen
            */
        u_long  t_rttupdated;           /* number of times rtt
            sampled */
        u_long  max_sndwnd;             /* largest window peer
            has offered */

        int     t_softerror;            /* possible error not
            yet reported */
/* out of band data */
        char    t_oobflags;             /* have some */
        char    t_iobc;                 /* input character */
#define TCPOOB_HAVEDATA 0x01
#define TCPOOB_HADDATA  0x02

/* RFC 1323 variables */
        u_char  snd_scale;              /* window scaling for
            send window */
        u_char  rcv_scale;              /* window scaling for
            recv window */
        u_char  request_r_scale;        /* pending window
            scaling */
        u_char  requested_s_scale;
        u_long  ts_recent;              /* timestamp echo data
```

```
            */

        u_long  ts_recent_age;          /* when last updated */
        tcp_seq last_ack_sent;

/* RFC 1644 variables */
        tcp_cc  cc_send;                /* send connection count
             */
        tcp_cc  cc_recv;                /* receive connection
            count */

/* experimental */
        u_long  snd_cwnd_prev;          /* cwnd prior to
            retransmit */
        u_long  snd_wacked_prev;        /* prior bytes acked in
            send window */
        u_long  snd_ssthresh_prev;      /* ssthresh prior to
            retransmit */
        tcp_seq snd_recover_prev;       /* snd_recover prior to
            retransmit */
        u_long  t_badrxtwin;            /* window for retransmit
             recovery */
        u_long  t_rexmtTS;              /* timestamp of last
            retransmit */
        u_char  snd_limited;            /* segments limited
            transmitted */

        tcp_seq rexmt_high;             /* highest seq #
            retransmitted + 1 */
        tcp_seq snd_max_rexmt;          /* snd_max when rexmting
             snd_una */
        struct scoreboard scb;          /* sack scoreboard */
        struct raw_sackblock reportblk; /* incoming segment or
            D SACK block */
        struct raw_sackblock encloseblk;
        int     nsackhistory;
        struct raw_sackblock
            sackhistory[MAX_SACK_REPORT_BLOCKS]; /* reported */
        TAILQ_ENTRY(tcpcb) t_outputq;   /* tcp_output needed
```

```
            list */

        /* bandwith limitation */
        u_long  snd_bandwidth;          /* calculated bandwidth
            or 0 */
        u_long  snd_bwnd;               /* bandwidth controlled
            window */
        int     t_bw_rtttime;           /* used for bandwidth
            calculation */
        tcp_seq t_bw_rtseq;             /* used for bandwidth
            calculation */
};
```

Η υλοποίηση του TCP στον πυρήνα βασίζεται σε μια μηχανή καταστάσεων, το διάγραμμα μεταβάσεων της οποίας παρουσιάζεται στο RFC793. Η τρέχουσα κατάσταση μιας σύνδεσης TCP βρίσκεται αποθηκευμένη στο πεδίο t_state. Ακόμα, για τον συντονισμό μεταξύ των χρονομετρητών και του μεγάλου αριθμού τοποθεσιών του κώδικα που εκτελείται από το νήμα πρωτοκόλλου στα οποία μπορεί να αλλάξει η ροή ελέγχου, χρησιμοποιείται μια πληθώρα επιλογών που προσδιορίζουν ενέργειες που έχουν γίνει ή ενέργειες που πρόκειται να γίνουν. Η επιλογές αυτές αποθηκεύονται στα bit του πεδίου t_flags.

Το πεδίο tt_cpu περιέχει τον αριθμό του πυρήνα επεξεργασίας στον οποίο τρέχει το νήμα πρωτοκόλλου TCP το οποίο εξυπηρετεί αυτή την σύνδεση.

Τα πεδία με τύπο δείκτη σε δομή tcp_callout, δηλαδή τα tt_rexmit, tt_persist, tt_keep, tt_2msl και το tt_delack επιτρέπουν την εύρεση των αντίστοιχων χρονομετρητών του TCP, δηλαδή του χρονομετρητή για την αναμετάδοση (πιθανώς) χαμένου πακέτου, του χρονομετρητή “επιμονής” όταν το παράθυρο λήψης του άλλου άκρου έχει κλείσει εντελώς, του χρονομετρητή για την ανίχνευση απώλειας της σύνδεσης, εκείνου που φροντίζει να μην κλείσει η σύνδεση πριν επιβεβαιωθεί το πακέτο FIN του άλλου μέρους και εκείνου που φροντίζει για την αποστολή τυχόν καθυστερημένων πακέτων ACK.

Τα πεδία snd_una, snd_max, snd_nxt, snd_wl1, snd_wl2, iss, irs, rcv_next, rcv_adv, rcv_wnd, snd_wnd, snd_cwnd, snd_wacked, snd_ssthresh, t_rxtcur, t_srtt, t_rttvar, t_maxopd, t_rcvtime, t_starttime, t_rtttiem, t_rtseq, t_rxtshift, t_rttmin, t_rttbest, t_rttupdated, max_sndwnd χρησιμοποιούνται για την εκτίμηση του χρόνου διαδρομής μετ’επιστροφής, για τον έλεγχο ροής και την αποφυγή συμφόρησης και για την αρχικοποίηση της σύνδεσης κατά την εκκίνηση.

Τα πεδία snd_up, rcv_up, t_oobflags, t_iobc χρησιμοποιούνται για την ανταλλαγή δεδομένων (ένα byte τη φορά) εκτός της κανονικής ροής δεδομένων της σύνδεσης.

Στο πεδίο t_softerror μπορεί να αποθηκευθεί ένας κωδικός λάθους που δεν είναι ακόμα δυνατό να παραδοθεί στην εφαρμογή.

Τα πεδία snd_scale, ts_recent, ts_recent_age και last_ack_sent χρησιμοποιούνται για την κλιμάκωση παραθύρου, την καλύτερη εκτίμηση του χρόνου διαδρομής μετ'επιστροφής και την προστασία από την υπερχείλιση των αριθμών ακολουθίας (βλ. RFC1323, Jacobson, Braden, Borman: Tcp Extensions for High Performance).

Τα πεδία cc_send και cc_recv χρησιμοποιούνται για την υλοποίηση των επεκτάσεων TCP for Transactions (T/TCP) που περιγράφονται στο RFC1644.

Το πεδίο t_rexmtTS περιέχει τη χρονοσφραγίδα του τελευταίου μεταδοθέντος πακέτου για τη υλοποίηση του αλγορίθμου ανίχνευσης Eiffel (RFC3522).

Τα πεδία snd_wacked_prev, snd_bandwidth, snd_bwnd, t_bw_rtttime και t_bt_rtseq είναι μέρος της υλοποίησης του RFC3465.

Το snd_limited χρησιμοποιείται για την τροποποίηση που προτείνει το RFC3042.

Τα rexmt_high, snd_max_rexmt, scb, reportblk, encloseblk, nsackhistory, χρειάζονται για την υποστήριξη επιλεκτικών επιβεβαιώσεων (RFC2018).

Τα πεδία rcv_second, rcv_pps, rcv_byps, rfbuf_ts, rfbuf_cnt κρατούν στατιστικά για να γίνεται με ασφάλεια η αυτόματη μεταβολή μεγέθους του socket buffer.

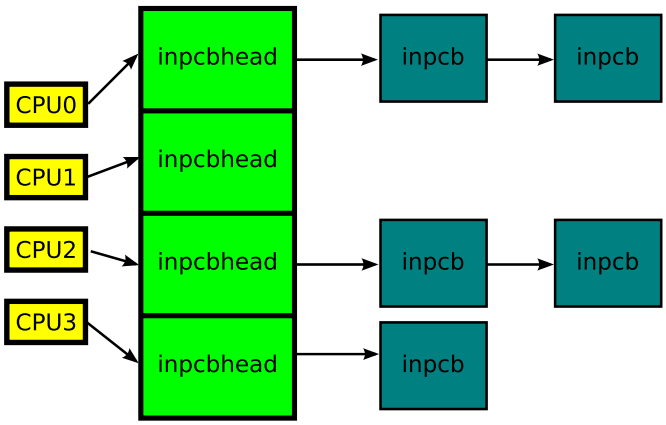
Το πεδίο t_outputq χρησιμεύει στο να συνδέονται σε μία λίστα τα tcpcb για τα οποία πρέπει να σταλεί καθυστερημένο ACK έτσι ώστε αντί για τη συνηθισμένη συμπεριφορά του TCP (ACK για κάθε δεύτερο ληφθέν πακέτο) να στέλνονται λιγότερα ACK σε γρήγορα δίκτυα).

Τα υπόλοιπα πεδία χρησιμοποιούνται για την υλοποίηση άλλων πειραματικών μετατροπών στο TCP.

3.6 Πίνακες Κατακερματισμού

Όπως είδαμε παραπάνω, οι δομές inpcb κάθε πρωτοκόλλου τοποθετούνται σε ένα πίνακα κατακερματισμού στον οποίο μπορούν να αναζητηθούν με βάση την τετράδα <απομακρυσμένη διεύθυνση, απομακρυσμένη θύρα, τοπική διεύθυνση, τοπική θύρα>.

Για το πρωτόκολλο TCP, ο πίνακας κατακερματισμού είναι σπασμένος σε τόσα



Σχήμα 3.3: “Σπασμένος” ανά επεξεργαστή πίνακας κατακερματισμού για τις δομές inpcb που αντιστοιχούν σε συνδέσεις TCP

το πλήθος κομμάτια, όσα και ο αριθμός των επεξεργαστικών πυρήνων. Κάθε κομμάτι του πίνακα ανήκει σε έναν επεξεργαστικό πυρήνα ή, ισοδύναμα, σε ένα νήμα πρωτοκόλλου TCP. Αυτό σημαίνει, με λίγες εξαιρέσεις τις οποίες έπρεπε να απαλείψουμε, ότι μόνο το αντίστοιχο νήμα TCP έχει δικαίωμα να τροποποιήσει κάποιο από τα κομμάτια του πίνακα. Επομένως, δεν χρειάζεται κανενός είδους συγχρονισμός για την πρόσβαση στα κομμάτια του πίνακα. Προφανώς, η αντιστοίχιση των δομών inpcb στα κομμάτια του πίνακα πρέπει να γίνεται με την ίδια συνάρτηση κατακερματισμού που χρησιμοποιούμε για την αντιστοίχιση των πακέτων σε επεξεργαστικούς πυρήνες.

Αντίθετα, για το UDP, οι δομές inpcb τοποθετούνται σε ένα μοναδικό πίνακα κατακερματισμού και ο κώδικας που τα τροποποιεί είναι απαραίτητο να κατέχει την BGL. Η αίρεση αυτής της απαίτησης ανακινεί αρκετά ενδιαφέροντα ζητήματα, τα οποία θα εξετάσουμε στη συνέχεια.

3.7 Στρώμα socket

Η προγραμματική διεπαφή (API) socket που περιγράφεται στο [STEVEN98] καθρεφτίζει την υλοποίηση των sockets στον πυρήνα. Σύμφωνα με τις οδηγίες της εφαρμογής, όταν αυτή ζήτηση τη δημιουργία ενός socket, αυτό έχει διαφορετικό τύπο (π.χ. SOCK_STREAM ή SOCK_DGRAM που συνήθως αντιστοιχούν στο πρωτόκολλο TCP ή UDP αντίστοιχα), ο οποίος αποθηκεύεται στο πεδίο so_type

(απόσπασμα 3.10).

Listing 3.10: struct socket

```
struct socket {
        short   so_type;                /* generic type, see
                socket.h */
        short   so_options;             /* from socket call, see
                socket.h */
        short   so_linger;              /* time to linger while
                closing */
        short   so_state;               /* internal state flags
                SS_*, below */
        void    *so_pcb;                /* protocol control
                block */
        struct  protosw *so_proto;      /* protocol handle */
        struct  socket *so_head;        /* back pointer to
                accept socket */

        /*
         * These fields are used to manage sockets capable of
                accepting
         * new connections.
         */
        TAILQ_HEAD(, socket) so_incomp; /* in progress,
                incomplete */
        TAILQ_HEAD(, socket) so_comp;   /* completed but not yet
                accepted */
        TAILQ_ENTRY(socket) so_list;    /* list of unaccepted
                connections */
        short   so_qlen;                /* so_comp count */
        short   so_incqlen;             /* so_incomp count */
        short   so_qlimit;              /* max number queued
                connections */

        /*
         * Misc socket support
         */
        short   so_timeo;               /* connection timeout */
        u_short so_error;               /* error affecting
                connection */
```

```
        struct  sigio *so_sigio;        /* information for async
            I/O or
                                           out of band data
                                              (SIGURG) */
        u_long  so_oobmark;             /* chars to oob mark */
        TAILQ_HEAD(, aiocblist) so_aiojobq; /* AIO ops waiting
           on socket */
        struct signalsockbuf so_rcv;
        struct signalsockbuf so_snd;

        void    (*so_upcall) (struct socket *, void *, int);
        void    *so_upcallarg;
        struct  ucred *so_cred;         /* user credentials */
        void    *so_emuldata;           /* private data for
           emulators */
        struct  so_accf {
                struct  accept_filter *so_accept_filter;
                void    *so_accept_filter_arg;  /* saved filter
                   args */
                char    *so_accept_filter_str;  /* saved user
                   args */
        } *so_accf;
};
```

Το πεδίο so_options περιέχει επιλογές που μπορεί να ζητήσει η εφαρμογή μέσω της κλήσης συστήματος setsockopt() ([STEVEN98]) καθώς και η επιλογή SO_ACCEPTCONN που ενεργοποιείται όταν η εφαρμογή εκτελέσει την κλήση συστήματος listen() ώστε το socket να μπορεί να δέχεται συνδέσεις.

Το πεδίο so_linger μπορεί να καθοριστεί από την εφαρμογή κατά την ενεργοποίηση της επιλογή SO_LINGER (βλ. [STEVEN98]).

Για διαδικτυακά πρωτόκολλα, το πεδίο so_pcb συνδέει τη δομή socket με μια δομή τύπου inpcb όπως αναφέρθηκε παραπάνω.

Η διεύθυνση του πεδίου so_timeo χρησιμοποιείται σαν κανάλι αναμονής (wait channel, βλ. [MCKUSI04]) στο οποίο μπλοκάρει η απόπειρα κλεισίματος της σύνδεσης περιμένοντας την επιβεβαίωση του πρωτοκόλλου πριν ειδοποιήσει την εφαρμογή για την επιτυχία της ενέργειας.

Στο πεδίο so_error μπορεί να αποθηκευτεί προσωρινά ένας κωδικός σφάλματος ο οποίος μπορεί να ερωτηθεί από την εφαρμογή ([STEVEN98]).

Το πεδίο so_sigio χρησιμοποιείται μόνο αν η εφαρμογή έχει ζητήσει ενημέρωση με σήματα (signals) για συμβάντα εισόδου/εξόδου ([STEVEN98]) και δεν μας ενδιαφέρει άμεσα.

Το πεδίο so_oobmark αποθηκεύει την απόσταση του χαρακτήρα που πρέπει να παραδοθεί εκτός της κανονικής ροής δεδομένων.

Το πεδίο so_aiojobq υπάρχει για την υποστήριξη ασύγχρονης εισόδου/εξόδου και ούτε και αυτό μας ενδιαφέρει άμεσα.

Το πεδίο so_cred είναι δείκτης σε πληροφορίες για την ταυτότητα και τις άδειες του χρήστη στον οποίο ανήκει το socket.

Το πεδίο so_proto είναι δείκτης σε δομή τύπου protosw που περιλαμβάνει δείκτες σε συναρτήσεις που υλοποιούν τις υπηρεσίες που προσφέρει το συνδεδεμένο πρωτόκολλο.

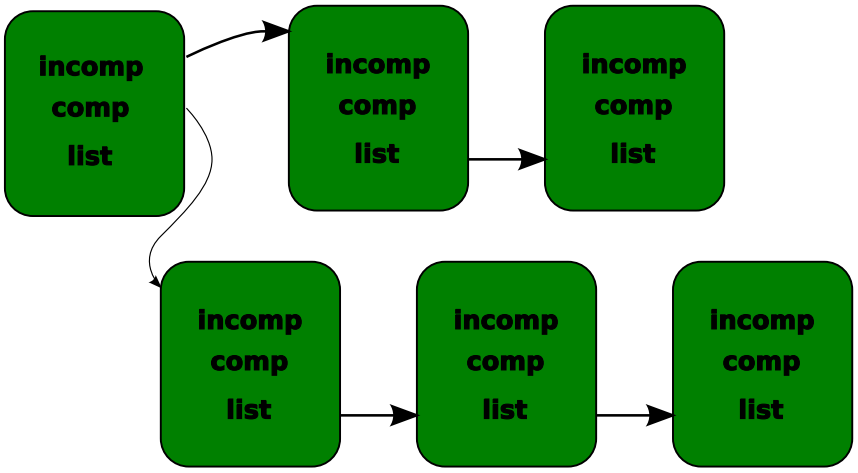
Το πεδίο so_upcall είναι δείκτης σε συνάρτηση η οποία καλείται όταν ξεκινήσει μία σύνδεση και στην οποία δίνεται σαν μία από τις παραμέτρους το περιεχόμενο του πεδίου so_upcallarg. Ο δείκτης αυτός αρχικοποιείται από το υποσύστημα netgraph (το οποίο είναι ένα εναλλακτικό υποσύστημα διασύνδεσης αντικειμένων που εκτελούν δικτυακές λειτουργίες σε αυθαίρετα συνδεδεμένους γράφους και από τα φίλτρα αποδοχής (accept filters) τα οποία επιτρέπουν την ειδοποίηση της εφαρμογής για το ότι δημιουργήθηκε μια νέα σύνδεση μόνο όταν ληφθούν δεδομένα σε αυτή (με το φίλτρο αποδοχής HTTP πρέπει μάλιστα αυτά να είναι δεδομένα πρωτοκόλλου HTTP). Ο τύπος του φίλτρου, μαζί με τα υπόλοιπα δεδομένα του, αποθηκεύεται στο πεδίο so_accf.

Το πεδίο so_emuldata είναι δείκτης σε δεδομένα που τυχόν υπάρχουν αν είναι ενεργός ο κώδικας εξομοίωσης πυρήνα άλλου λειτουργικού συστήματος (π.χ. του Linux).

Όταν σε ένα socket που μπορεί να δέχεται συνδέσεις (στον κώδικα καλείται accept socket) η εφαρμογή κάνει την κλήση συστήματος accept, τότε ο πυρήνας δημιουργεί ένα νέο socket για αυτή τη σύνδεση και επιστρέφει στην εφαρμογή μια αναφορά στο socket που μόλις δημιουργήθηκε. Αυτό μπορεί να επαναληφθεί όσο υπάρχουν διαθέσιμοι πόροι για νέες συνδέσεις και όσο το επιτρέπουν τα όρια του συστήματος. Τα socket που αντιστοιχούν σε συνδέσεις μπορούν να συνδεθούν σε λίστα μέσω του πεδίου so_list.

Από ένα accept socket ξεκινούν δύο τέτοιες λίστες από socket συνδέσεων (σχήμα 3.7). Αυτή που ξεκινά από το πεδίο so_incomp περιλαμβάνει τα socket για τα οποία η σύνδεση βρίσκεται σε εξέλιξη, ώστε να μπορούν να διαγραφούν αν

η εφαρμογή κλείσει το accept socket από το οποίο δημιουργήθηκαν.



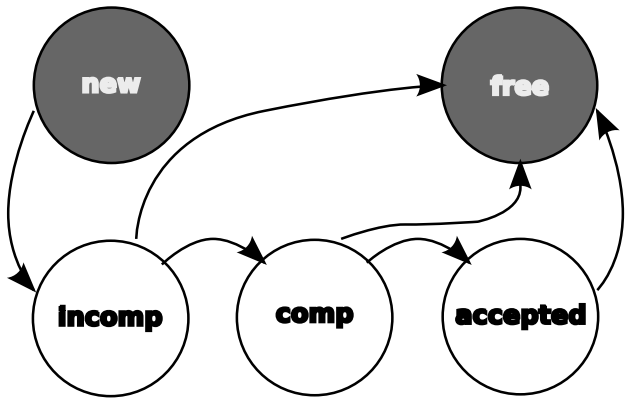
Σχήμα 3.4: Λίστες που ξεκινούν από ένα accept socket

Η άλλη λίστα ξεκινά από το πεδίο so_comp και περιέχει τις ολοκληρωμένες συνδέσεις ώστε αυτές να μπορούν να επιστραφούν στην εφαρμογή κατά την τρέχουσα ή επόμενη κλήση της accept().

Τα πεδία so_qlen, so_inqlen περιέχουν το πλήθος των στοιχείων των λιστών so_comp και so_incomp αντίστοιχα. Το πλήθος των συνδέσεων στη λίστα so_incomp δεν μπορεί να ξεπεράσει την τιμή του πεδίου so_qlen (το οποίο αρχικοποιείται από την εφαρμογή) ενώ το μήκος της λίστας so_comp δεν μπορεί να είναι μεγαλύτερο από το 150% αυτής της τιμής. Στην πρώτη περίπτωση απορρίπτεται η πιο παλιά σύνδεση που βρίσκεται σε εξέλιξη για να εισαχθεί η καινούργια, ενώ στη δεύτερη παύουν να γίνονται αποδεκτές νέες συνδέσεις έως ότου μειωθεί το μήκος της λίστας.

Φυσικά τα πεδία αυτά μένουν ανενεργά αν το πρωτόκολλο δεν υποστηρίζει συνδέσεις.

Τέλος η δομή socket περιλαμβάνει έναν ενταμιευτή για τα δεδομένα προς αποστολή που προέρχονται από την εφαρμογή και έναν για τα δεδομένα που παραλήφθηκαν από το δίκτυο και δεν έχουν παραδοθεί ακόμα στην εφαρμογή. Ο πρώτος περιέχεται στο πεδίο so_snd και ο δεύτερος είναι ενσωματωμένος στο πεδίο so_rcv. Η υλοποίηση του sockbuf θα μας απασχολήσει σε λίγο. Η δομή signalsockbuf (βλ. απόσπασμα 3.11) περιλαμβάνει επίσης το πεδίο so_timeo το οποίο καθορίζει πότε θα εκπνεύσει μια προσπάθεια ανάγνωσης ή εγγραφής. Το



Σχήμα 3.5: Μετάβαση καταστάσεων ενός socket που αντιστοιχεί σε σύνδεση

πεδίο ssb_lowat στον ενταμιευτή so_rcv είναι ο ελάχιστος αριθμός bytes που πρέπει να είναι διαθέσιμος για να επιτύχει μία ανάγνωση από το socket ενώ στον so_snd είναι ο ελεύθερος χώρος που πρέπει να υπάρχει στον so_snd για να πετύχει μια προσπαθεί αποστολής δεδομένων. Τα πεδία αυτά μπορούν να καθοριστούν από την εφαρμογή ([STEVEN98]).

Listing 3.11: struct signalsockbuf

```
/*
 * Signaling socket buffers contain additional elements for
    locking
 * and signaling conditions.  These are used primarily by
    sockets.
 */
struct signalsockbuf {
        struct sockbuf sb;
        struct selinfo ssb_sel; /* process selecting read/write
            */
        short   ssb_flags;      /* flags, see below */
        short   ssb_timeo;      /* timeout for read/write */
        long    ssb_lowat;      /* low water mark */
        u_long  ssb_hiwat;      /* high water mark / max actual
            char count */
        u_long  ssb_mbmax;      /* max chars of mbufs to use */
};
```

Η τιμή του πεδίου ssb_mbmax είναι το άνω όριο στο συνολικό μέγεθος των δομών mbuf που μπορεί να χρησιμοποιηθούν στο sockbuf της δομής signalsockbuf.

Το πεδίο ssb_hiwat συγκρατεί το μέγιστο αριθμό bytes που περιέχει ο ενταμιευτής.

Τέλος, στα bits του πεδίου ssb_flags κωδικοποιούνται οι επιλογές SSB_LOCK, SSB_WANT, SSB_WAIT με τις οποίες υλοποιείται ο αμοιβαίος αποκλεισμός στα πεδία της δομής.

Ακόμα, η επιλογή SSB_SEL που υποδηλώνει ότι υπάρχει ενεργή κλήση συστήματος select ([STEVEN99]) για αυτόν τον ενταμιευτή.

Η επιλογή SSB_ASYNC που δηλώνει ότι η εφαρμογή έχει ζητήσει είσοδο/έξοδο χωρίς μπλοκάρισμα ([STEVEN98]).

Η επιλογή SSB_UPCALL εξασφαλίζει την κλήση του δείκτη so_upcall (βλ. παραπάνω) του socket που περιέχει αυτή τη δομή signalsockbuf.

Το μπλοκάρισμα σε αυτόν τον ενταμιευτή δεν πρόκειται να διακοπεί για την παράδοση ενός σήματος αν υπάρχει η επιλογή SSB_NOINTR.

Η επιλογή SSB_AIO είναι ενεργή αν υπάρχουν σε εξέλιξη ενέργειες ασύγχρονης εισόδου/εξόδου (AIO).

Αν υπάρχει η επιλογή SSB_MEVENT, τότε σε κάθε συμβάν που επηρεάζει την δυνατότητα εισόδου/εξόδου πρέπει να αποσταλεί ένα μήνυμα στους αποδέκτες που έχουν εκδηλώσει ενδιαφέρον (γεγονός το οποίο καταγράφεται στη δομή selinfo του πεδίου ssb_sel.

Η επιλογή SSB_KNOTE δηλώνει ότι υπάρχει ειδοποίηση τύπου knote που χρησιμοποιείται από το υποσύστημα kqueue [LEMON].

Τέλος, η επιλογή SSB_STOP χρησιμοποιείται για τον έλεγχο ροής σε socket τύπου Unix Domain [STEVEN99].

3.7.1 Socket buffers

Οι δομές τύπου signalsockbuf περιέχουν κυρίως πεδία τα οποία είναι συνοδευτικά του βασικού τους πεδίου (sb) που έχει τύπο δομής sockbuf (socket buffer, βλ. απόσπασμα 3.12). Με την εξαίρεση της μεταφοράς ορισμένων πεδίων στη δομή signalsockbuf, η δομή sockbuf έχει μείνει ίδια με αυτή του 4.4BSD και παρόμοια με τη μορφή της δομής sockbuf στο 4.3BSD.

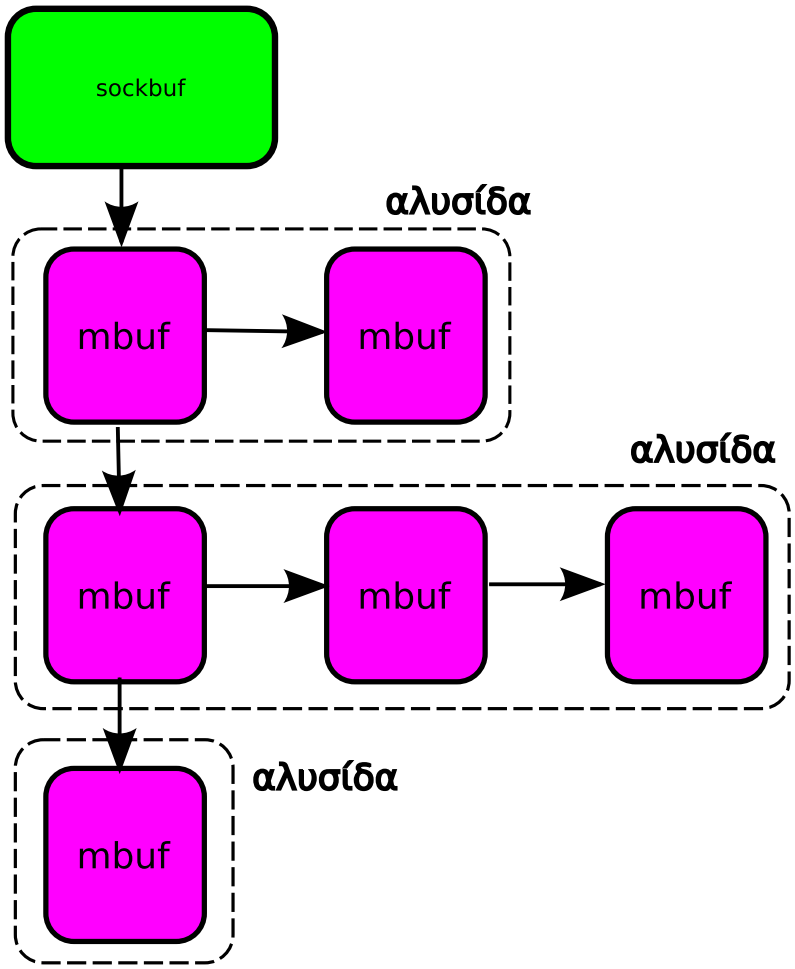
Listing 3.12: struct sockbuf

```
/*
 * Generic socket buffer for keeping track of mbuf chains.
    These
 * are used primarily to manipulate mbuf chains in standalone
    pieces
 * of code.
 */
struct sockbuf {
        u_long  sb_cc;          /* actual chars in buffer */
        u_long  sb_mbcnt;       /* chars of mbufs used */
        u_long  sb_climit;      /* data limit when used for I/O
            */
        struct  mbuf *sb_mb;    /* the mbuf chain */
        struct  mbuf *sb_lastmbuf;      /* last mbuf in sb_mb */
        struct  mbuf *sb_lastrecord;    /* last record in sb_mb
                                         * valid <=> sb_mb
                                            non NULL */
};
```

Ο βασικός ρόλος της δομής sockbuf είναι να αποθηκεύει αλυσίδες από mbuf (κάθε αλυσίδα αντιστοιχεί σε ένα πακέτο). Το πεδίο sb_mb είναι δείκτης στο πρώτο στοιχείο της πρώτης αλυσίδας mbuf. Οι αλυσίδες mbuf συνδέονται μεταξύ τους μέσω των πεδίων m_nextpkt κάθε πρώτου στοιχείου αλυσίδας. Όταν η δομή sockbuf δεν περιέχει πακέτα, το πεδίο sb_mb είναι μηδέν. Η αποθήκευση των πακέτων υπακούει στον κανόνα FIFO (first in, first out). Έτσι τα νεότερα πακέτα καταλήγουν στο τέλος της αλυσίδας. Για το λόγο αυτό, ένας δείκτης στο τελευταίο πακέτο της λίστας αποθηκεύεται στο πεδίο sb_lastmbuf ώστε να γίνεται γρήγορα η σύνδεση του νέου πακέτου. Αντίστροφα, τα πακέτα αφαιρούνται από την αρχή της λίστας (δηλαδή το πρώτο πακέτο που θα “διαβαστεί” είναι αυτό στο οποίο δείχνει ο δείκτης sb_mb.

Κατ’αναλογία με το sb_lastmbuf που μόλις είδαμε, το πεδίο sb_lastrecord περιέχει ένα δείκτη στο πρώτο στοιχείο της τελευταίας αλυσίδας που περιέχεται στο sockbuf, έτσι ώστε να είναι εύκολο να συνδεθεί μια νέα αλυσίδα χωρίς να χρειάζεται να διατρέξουμε όλα τα στοιχεία της λίστας.

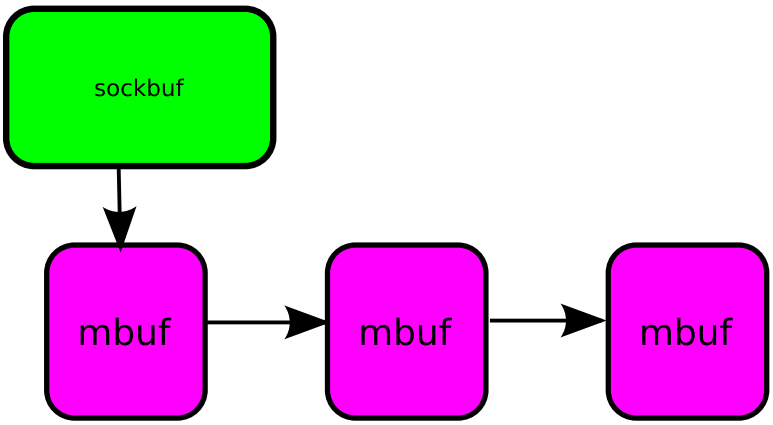
Το sb_lastrecord χρησιμοποιείται αν το πρωτόκολλο υποστηρίζει την έννοια των εγγραφών (records) δηλαδή αν κάθε πακέτο παραδίδεται ξεχωριστά. Αν, όπως το TCP, έχει την έννοια μιας ακολουθίας από bytes τα mbuf αποθηκεύονται γραμ-



Σχήμα 3.6: Δομή sockbuf με εγγραφές

μικά σε μία λίστα και τ΄οτε το πεδίο sb_lastrecord είναι περιττό.

Στα πεδία sb_cc, sb_mbcnt αποθηκεύεται, αντίστοιχα, ο αριθμός των bytes δεδομένων που περιέχονται στο sockbuf και το συνολικό μέγεθος των mbuf που είναι συνδεδεμένα σε αυτό. Το πεδίο sb_climit παίζει ρόλο μόνο όταν το sockbuf χρησιμοποιείται αυτόνομα (χωρίς δηλαδή να αποτελεί μέρος ενός socket) και δεν θα μας απασχολήσει.



Σχήμα 3.7: Δομή sockbuf χωρίς εγγραφές

3.8 Στρώμα διεπαφής

Είναι χρήσιμο να δώσουμε μια σύντομη περιγραφή του στρώματος διεπαφής το οποίο παρέχει τις βοηθητικές λειτουργίες για την επικοινωνία του οδηγού μιας κάρτας δίκτυο με το κατώτερο στρώμα της στοίβας πρωτοκόλλων. Χωρίς μεγάλη απώλεια γενικότητας, θα ασχοληθούμε μόνο με κάρτες τύπου Ethernet.

Οι κάρτες δικτύου χρησιμοποιούν συνήθως έναν κυκλικό ενταμιευτή (ring buffer) για την αποθήκευση εγγραφών σταθερού μήκους που έχουν στοιχεία για τις περιοχές της μνήμης που περιέχουν δεδομένα των πακέτων. Υπάρχει ένας (ή στις μοντέρνες κάρτες περισσότεροι) τέτοιοι κυκλικοί ενταμιευτές για την αποστολή δεδομένων, στους οποίους ο οδηγός της κάρτας αποθηκεύει εγγραφές που “δείχνουν” στα περιεχόμενα των πακέτων που του δόθηκαν για αποστολή. Ο τρόπος με τον οποίο ο οδηγός ειδοποιεί την κάρτα για τις νέες εγγραφές διαφέρει σε κάθε κάρτα.

Το στρώμα διεπαφής δημιουργεί για κάθε διεπαφή μια ουρά αποστολής δεδομένων στην οποία οι αποστολείς τοποθετούν τα εξερχόμενα mbuf. Κατόπιν, καλούν τη ρουτίνα του οδηγού που αναλαμβάνει τη δημιουργία κατάλληλων εγγραφών στους κυκλικούς ενταμιευτές αποστολής και την ειδοποίηση της κάρτας ότι υπάρχουν πακέτα προς αποστολή. Η ρουτίνα αυτή εκτελείται, αν είναι δυνατό, στο πλαίσιο του τρέχοντος νήματος (αυτού δηλαδή που αποστέλει τα δεδομενα ή στο πλαίσιο ενός ειδικού νήματος (ifnet).

Αντίστοιχα, υπάρχει ένα σύνολο κυκλικών ενταμιευτών (στις παλαιότερες κάρ-

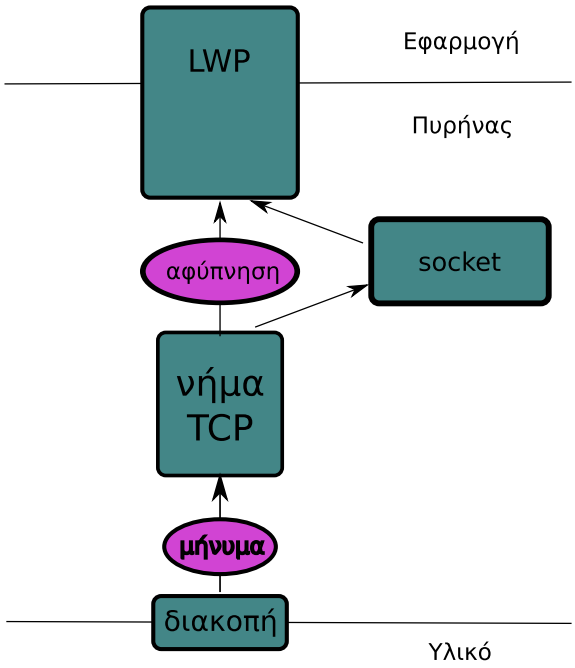
τες δικτύου μόνο ένας) όπου ο οδηγός τοποθετεί εγγραφές που “δείχνουν” σε άδεια mbuf στα οποία η κάρτα αποθηκεύει τα πακέτα που λαμβάνει. Τυπικά, η κάρτα δημιουργεί μία διακοπή (interrupt) για κάθε n πακέτα που λαμβάνει ή κάθε t μs όσο υπάρχουν νέα πακέτα. Τα χαρακτηριστικά και η δυναμική συμπεριφορά διαφέρουν πολύ από κάρτα σε κάρτα, αλλά το απλοποιημένο μοντέλου που δώσαμε καλύπτει τη μεγάλη πλειοψηφία των καρτών που βρίσκονται σε χρήση και των νέων μοντέλων που αναμένονται.

Αρκετές κάρτες είναι σε θέση να εκτελέσουν λειτουργίες της στοίβας πρωτοκόλλων όπως π.χ. ο υπολογισμός του αθροίσματος ελέγχου για τα δημοφιλή πρωτόκολλα και η τεμάχιση TCP (TCP Segmentation Offloading). Παρότι αυτές οι επιπλέον δυνατότητες βελτιώνουν την απόδοση, δεν επηρεάζουν σε μεγάλο βαθμό τις επιλογές μας για την υλοποίηση της στοίβας πρωτοκόλλων και για αυτό θα τις αγνοήσουμε σε αυτό το κείμενο.

3.9 Ροή δεδομένων

Ας δώσουμε τώρα ένα ολοκληρωμένο παράδειγμα που θα βοηθήσει τον αναγνώστη να κατανοήσει πως συντίθενται μεταξύ τους τα υποσυστήματα που περιγράψαμε καθώς και την δυναμική συμπεριφορά του συστήματος.

Αρχικά, ας υποθέσουμε ότι λαμβάνεται ένα πακέτο σε μια κάρτα δικτύου του συστήματος. Τότε, όπως είδαμε παραπάνω, αυτό αποθηκεύεται σε ένα από τα mbuf που έχει δεσμεύσει για αυτό το σκοπό ο οδηγός της διεπαφής. Όταν μαζευτούν αρκετά πακέτα ή έχει περάσει ένα συγκεκριμένο χρονικό διάστημα, η κάρτα προκαλεί μια διακοπή. Αντιδρώντας στη διακοπή ο επεξεργαστικός πυρήνας εκτελεί τη ρουτίνα εξυπηρέτησης διακοπής (interrupt service routine) που έχει καταχωρίσει ο οδηγός. Η ρουτίνα συνδέει τα εισερχόμενα mbuf σε λίστες και προκαλεί ένα IPI σε κάθε επεξεργαστικό πυρήνα για τον οποίο προορίζονται πακέτα. Ο κώδικας που τρέχει σε απόκριση του IPI στέλνει ένα μήνυμα με το κάθε πακέτο στο νήμα πρωτοκόλλου. Αν έχει καθοριστεί ότι πρόκειται π.χ. για πακέτο TCP, το mbuf, θα αποσταλεί απευθείας στο νήμα TCP και σε αυτό το νήμα θα εκτελεστεί ο κώδικας τόσο για το στρώμα Ethernet όσο και για το στρώμα IP και τελικά για το TCP χωρίς επιπλέον context switches και ανταλλαγές μηνυμάτων. Στην συνέχεια το στρώμα TCP θα παραδώσει το πακέτο στο ενταμιευτή λήψης του socket που αντιστοιχεί στη σύνδεση και θα ενημερώσει τους αναγνώστες που τυχόν περιμένουν για δε-



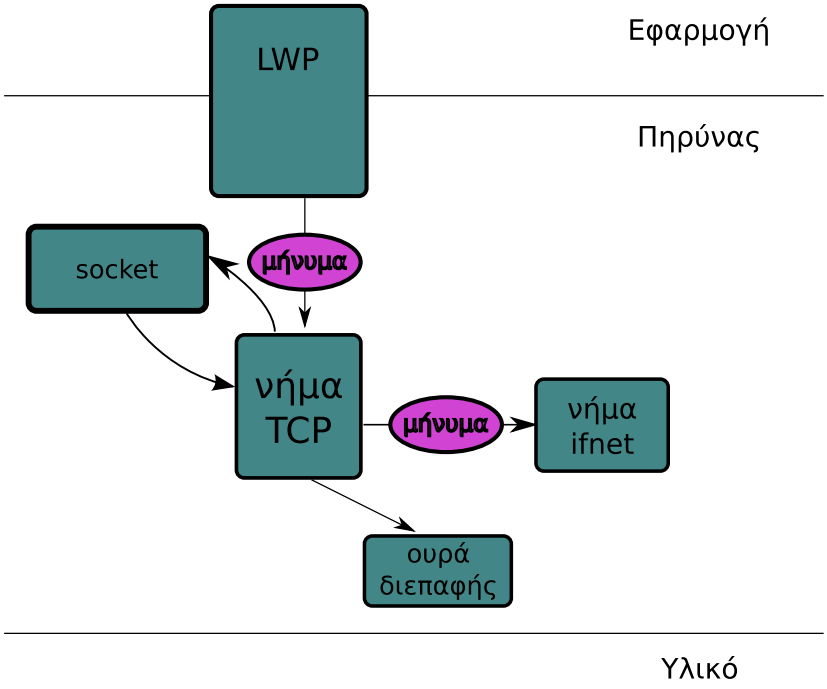
Σχήμα 3.8: Λήψη δεδομένων

δομένα.

Όταν ένας αναγνώστης προσπαθήσει να διαβάσει δεδομένα ή να διαπιστώσει αν υπάρχουν διαθέσιμα δεδομένα και έχει δηλώσει την πρόθεση να μπλοκάρει περιμένοντάς τα, τότε αν δεν υπάρχουν δεδομένα θα μπλοκάρει περιμένοντας σε ένα κανάλι αναμονής, να τον ξυπνήσει το νήμα πρωτοκόλλου όταν παραδοθούν νέα δεδομένα ή όταν αλλάξει η κατάσταση της σύνδεσης (π.χ. αν διακοπεί) ή όταν ο αποστολέας κάνει γνωστό ότι δεν πρόκειται να στείλει περαιτέρω δεδομένα.

Θεωρούμε τώρα την περίπτωση όπου μια εφαρμογή προσπαθεί να στείλει δεδομένα. Αν υποθέσουμε ότι υπάρχει χώρος στον ενταμιευτή αποστολής τους socket, τότε κατά την κλήση συστήματος τα δεδομένα προς αποστολή αντιγράφονται σε δομές mbuf και στέλνεται ένα μήνυμα που ζητά από το νήμα πρωτοκόλλου να τοποθετήσει τα δεδομένα στον ενταμιευτή αποστολής για να τα αποστείλει. Όταν το νήμα πρωτοκόλλου λάβει το μήνυμα και αποθηκεύσει τα δεδομένα προχωρεί στην αποστολή τους, καλώντας τελικά τη ρουτίνα αποστολής του κατώτερου πρωτοκόλλου στη στοίβα και ούτως καθεξής έως ότου το τελευταίο στρώμα συν-

δέσει το mbuf στην ουρά αποστολής της σωστής διεπαφής (που την είδαμε στην ενότητα 3.8) και φροντίσει να τρέξει την ρουτίνα αποστολής δεδομένων του οδηγού της διεπαφής. Αν η ουρά της διεπαφής είναι κλειδωμένη από κάποιο άλλο νήμα, τότε στέλνεται ένα μήνυμα στο νήμα ifnet που αντιστοιχεί στη διεπαφή.



Σχήμα 3.9: Αποστολή δεδομένων

Κεφάλαιο 4

Τροποποίηση υλοποίησης διαδικτυακών πρωτοκόλλων

4.1 Εισαγωγή

Οι αλλαγές που πραγματοποιήσαμε στον πυρήνα του DragonFlyBSD αποσκοπούσαν σε δύο κύριους στόχους. Από τη μία, έπρεπε να επιθεωρήσουμε όλα τα σημεία στα οποία τμήματα κώδικα που θα μπορούσαν να τρέχουν ταυτόχρονα σε διαφορετικούς επεξεργαστικούς πυρήνες τροποποιούσαν ή διάβαζαν το ίδιο στιγμιότυπο μίας δομής δεδομένων, αν δεν απαιτούνταν η κατοχή της BGL. Κατόπιν, έπρεπε να βρούμε τον τρόπο να αφαιρέσουμε την απαίτηση για την κατοχή της BGL, πιθανώς αντικαθιστώντας τη με μια άλλη λύση αμοιβαίου αποκλεισμού ή ίσως και δείχνοντας ότι οι ταυτόχρονες αλλαγές δεν αποτελούν πρόβλημα με οποιαδήποτε χρονική σειρά και αν γίνουν (το τελευταίο δεν είναι καθόλου προφανές). Από την άλλη, χρειάστηκαν και κάποιες αλγοριθμικές βελτιώσεις εκεί όπου υπήρχαν εμφανή προβλήματα απόδοσης.

Στην εργασία αυτή, επικεντρωθήκαμε στα μονοπάτια κώδικα τα οποία εκτελούνται πιο συχνά (τις περισσότερες φορές τάξεις μεγέθους πιο συχνά) γιατί οι βελτιώσεις σε αυτά τα μονοπάτια είναι που μπορούν να επιφέρουν βελτίωση στην απόδοση του συστήματος. Σύμφωνα με το νόμο του Amdahl, αν P είναι το ποσοστό του χρόνου εκτέλεσης μιας εργασίας του κομματιού που βελτιώνουμε και S η επιτάχυνση που μπορούμε να πετύχουμε για αυτό το κομμάτι (όπου η επιτάχυνση ορίζεται ως ο λόγος του χρόνου εκτέλεσης του κομματιού πριν τις βελτιώσεις προς

το χρόνο εκτέλεσης του κομματιού μετά τις βελτιώσεις), τότε η συνολική επιτάχυνση (δηλαδή ο λόγος του χρόνου εκτέλεσης πριν τις βελτιώσεις προς τον χρόνο εκτέλεσης μετά τις βελτιώσεις) είναι

$$\frac{1}{1 \quad P + P/S}$$

Επιλέγοντας συχνά εκτελούμενα (“θερμά”) μονοπάτια, αποσκοπούμε στο να μεγιστοποιήσουμε το P, άρα και τη συνολική επιτάχυνση.

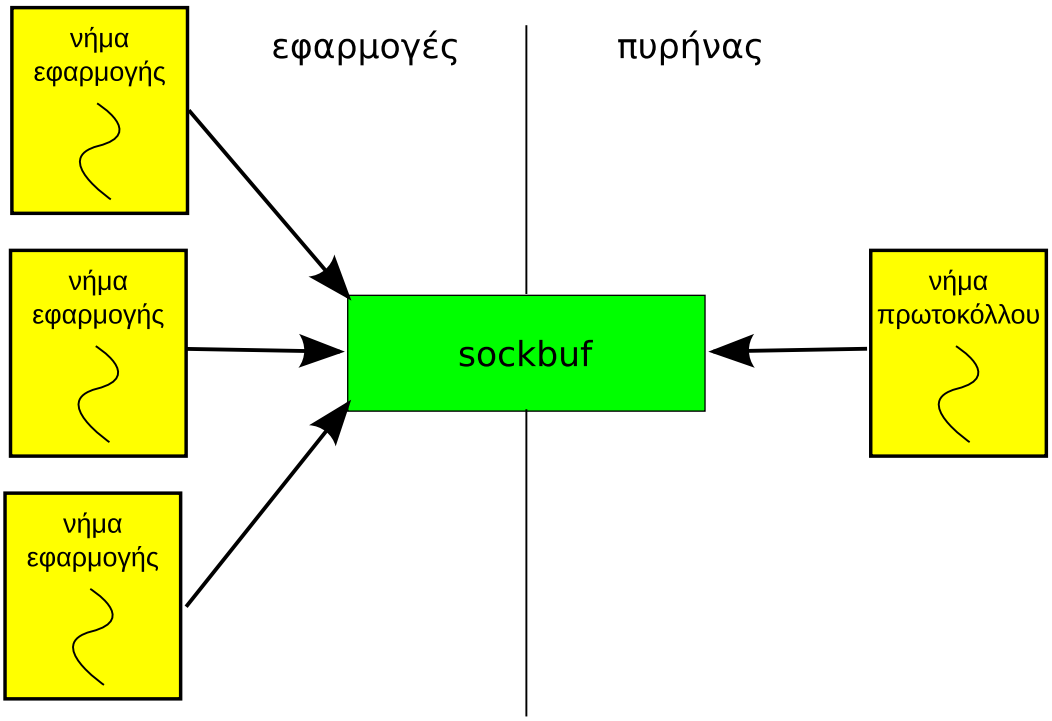
Το μονοπάτι που ακολουθείται για την αποστολή δεδομένων, για παράδειγμα, είναι πολύ πιο σημαντικό για την απόδοση του συστήματος από ο,τι το μονοπάτι για τον πρόωρο τερματισμό μίας σύνδεσης.

4.2 Η δομή sockbuf

Η πιο ενδιαφέρουσα από τις δομές δεδομένων που προσπελαύνονται / τροποποιούνται ταυτόχρονα τόσο από τα νήματα πρωτοκόλλου του πυρήνα όσο και από τις εφαρμογές όταν αυτές τρέχουν στον πυρήνα εκτελώντας μια κλήση συστήματος είναι χωρίς αμφιβολία η δομή sockbuf (3.7.1).

Όπως είδαμε, κάθε δομή socket περιέχει δύο δομές sockbuf (ενσωματωμένες σε δομές τύπου signalsockbuf), μια για την αποστολή (so_snd) και μια για τη λήψη (so_rcv) δεδομένων. Εξετάσαμε προηγουμένως (3.9) το μονοπάτι εισόδου και το μονοπάτι εξόδου δεδομένων και από τις περιγραφές αυτές πρέπει να είναι φανερό ότι ο ενταμιευτής αποστολής προσπελαύνεται διαφορετικά από τον ενταμιευτή λήψης. Πράγματι, μόνο ένα νήμα πρωτοκόλλου αφαιρεί δεδομένα από τον ενταμιευτή αποστολής αλλά σε αυτόν μπορεί να γράφουν πολλά νήματα ή διεργασίες που τυχόν έχουν αποκτήσει μια αναφορά στο socket. Έχουμε λοιπόν ένα πρόβλημα πολλών παραγωγών - ενός καταναλωτή (MPSC, multiple producers, single consumer). Συμμετρικά, στον ενταμιευτή λήψης μπορεί να προσθέτει δεδομένα μόνο το αντίστοιχο νήμα πρωτοκόλλου, όμως και πάλι μπορούν να αφαιρούν δεδομένα πολλά νήματα και διεργασίες (σχ. 4.1). Ο ενταμιευτή λήψης δηλαδή αποτελεί ένα πρόβλημα ενός παραγωγού - πολλών καταναλωτών (SPMC, single producer, multiple consumers). Τα προβλήματα αυτά έχουν μελετηθεί αρκετά και έχουν προταθεί διάφορες λύσεις. Δυστυχώς, καμία τους δεν αποτελεί προφανή επιλογή, είτε γιατί είναι πολύ περίπλοκες είτε γιατί είναι πολύ αργές για τις απαιτή-

σεις μας είτε γιατί υποθέτουν την ύπαρξη εντολών οι οποίες δεν είναι διαθέσιμες στις τρέχουσες αρχιτεκτονικές του εμπορίου.



Σχήμα 4.1: Πρόσβαση στη δομή sockbuf

Παρατηρούμε ωστόσο ότι οι πολλοί παραγωγοί / καταναλωτές εμφανίζονται πάντα από την πλευρά των κλήσεων συστήματος ενώ η πλευρά του νήματος πρωτοκόλλου είναι πάντα η “μονή” πλευρά. Συνεπώς, αν απαιτήσουμε οι κλήσεις συστήματος να αποκτούν ένα κλείδωμα προτού τροποποιήσουν τη δομή sockbuf, τότε τόσο το πρόβλημα MPSC (στον ενταμιευτή αποστολής), όσο και το πρόβλημα SPMC (στον ενταμιευτή λήψης) μετατρέπονται στο πολύ απλούστερο πρόβλημα ενός παραγωγού - ενός καταναλωτή (SPSC, single producer, single consumer). Υπάρχουν αρκετοί προφανείς αλγόριθμοι για το πρόβλημα SPSC οι οποίοι δεν απαιτούν μηχανισμούς αμοιβαίου αποκλεισμού και στους οποίους καμία από τις δύο πλευρές δεν χρειάζεται ποτέ να περιμένει. Αυτή η ιδιότητα είναι το βασικό πλεονέκτημα της ιδέας να μετατρέψουμε το πρόβλημα σε ενός παραγωγού - ενός καταναλωτή. Αφού η πλευρά του πρωτοκόλλου δεν χρειάζεται να αποκτήσει κανένα κλείδωμα δεν πρόκειται ποτέ να μπλοκάρει περιμένοντας μια κλήση συστήματος

όσο υπάρχουν ακόμα ενέργειες που πρέπει να γίνουν.

Με κάποια δυσκολία, θα ήταν ίσως δυνατό να απαλείψουμε την ανάγκη για την κατοχή του κλειδώματος όταν υπάρχει μόνο μια αναφορά στο socket, όμως δεν ακολουθήσαμε αυτή την κατεύθυνση.

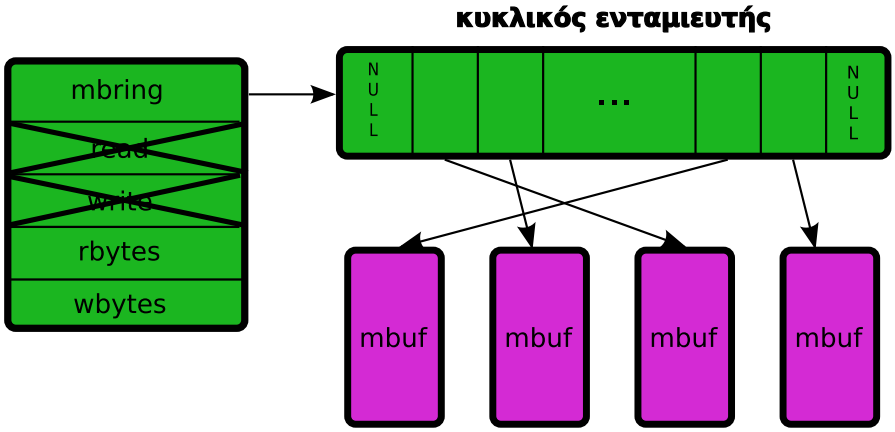
4.2.1 Επίλυση του προβλήματος SPSC

Υλοποιήσαμε τρεις διαφορετικές προσεγγίσεις για την επίλυση του προβλήματος SPSC στους ενταμιευτές sockbuf.

Κυκλικός ενταμιευτής: Μια πολύ συνηθισμένη επιλογή για το πρόβλημα SPSC είναι αυτή του κυκλικού ενταμιευτή. Η "κλασσική" υλοποίηση του Lamport χρησιμοποιεί έναν κυκλικό ενταμιευτή σταθερού μεγέθους και δείκτες στο επόμενο στοιχείο για ανάγνωση και το επόμενο για εγγραφή. Δυστυχώς, αυτό συνεπάγεται ότι ο παραγωγός τροποποιεί συνεχώς το πεδίο που διαβάζει ο καταναλωτής και το αντίστροφο. Έτσι, παρόλο που αυτός είναι ένας αλγόριθμος χωρίς αμοιβαίο αποκλεισμό ή αναμονή, έχει κακή συμπεριφορά σε ένα σύστημα με πολλούς επεξεργαστές γιατί οι γραμμές της κρυφής μνήμης (cache lines) που περιέχουν τους δείκτες πρέπει να μεταφέρονται συνεχώς από τον επεξεργαστή του καταναλωτή σε αυτόν του παραγωγού και το αντίθετο.

Μια καλύτερη και εξίσου απλή λύση είναι αυτή που προτείνεται στο [GIACOM08]. Σε αυτόν τον αλγόριθμο (FastForward), ο παραγωγός και ο καταναλωτής μπορούν να ανιχνεύσουν τις οριακές συνθήκες (γεμάτος ή άδειος ενταμιευτής αντίστοιχα) απλά διαβάζοντας τις ίδιες τις εγγραφές του κυκλικού ενταμιευτή. Όσο ο ενταμιευτής περιέχει κάποιες εγγραφές, δεν θα υπάρχει συνεχής μεταφορά γραμμών της κρυφής μνήμης.

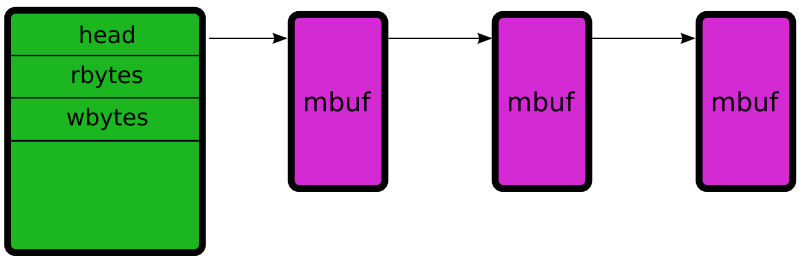
Η αρχική μας προσέγγιση υλοποιούσε τον αλγόριθμο FastForward, όμως παρουσίαζε ορισμένα προβλήματα. Εν πρώτοις, ο υπάρχων κώδικας χρησιμοποιούσε τα πεδία που κρατούν τον αριθμό των bytes στον ενταμιευτή για να ελέγξει τις οριακές συνθήκες (κάτι που ακυρώνει τα οφέλη του αλγορίθμου FastForward) και η αλλαγή αυτής της συμπεριφοράς απαιτούσε τροποποιήσεις που δεν ήταν ασήμαντες. Το σημαντικότερο πρόβλημα όμως ήταν η μνήμη που σπαταλούνταν σε ένα sockbuf. Για να καλύψουμε το μέγιστο παράθυρο του παραδοσιακού TCP θα θέλαμε 8192 bytes σε αρχιτεκτο-



Σχήμα 4.2: Sockbuf με κυκλικό ενταμιευτή

νικές 32bit ή 16384 bytes σε αρχιτεκτονικές 64bit, ενώ με την επιλογή TCP για κλιμάκωση παραθύρου αυτό μπορεί να είναι ακόμα μεγαλύτερο. Αυτή η σπατάλη μάλιστα θα υπήρχε ακόμα και αν το sockbuf έμενε αχρησιμοποίητο.

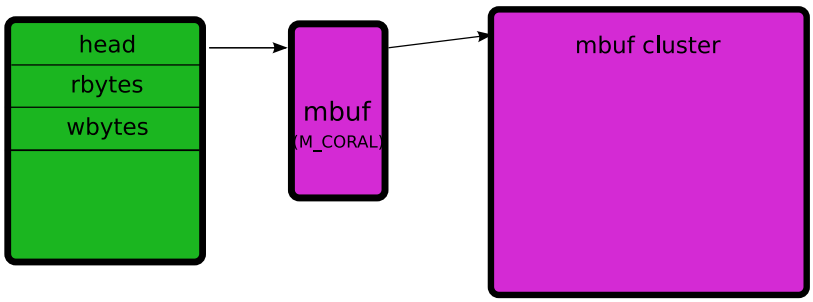
M_CORAL: Η επόμενη προσέγγιση που υλοποιήσαμε ήταν να χρησιμοποιήσουμε μια απλά συνδεδεμένη λίστα (σχ. 4.3 με την επιπλέον συνθήκη ότι ο καταναλωτής ποτέ δεν θα αφαιρούσε το τελευταίο στοιχείο της λίστας. Το πρόβλημα με μια μονά συνδεδεμένη λίστα είναι ότι ο καταναλωτής μπορεί να αφαιρέσει το τελευταίο στοιχείο ενώ ο παραγωγός ετοιμάζεται να συνδέσει ένα νέο στοιχείο. Η συνθήκη μας εξασφαλίζει ότι αυτό δεν θα συμβεί ποτέ, με το κόστος ότι κάνει τον αλγόριθμο πιο περίπλοκο.



Σχήμα 4.3: Sockbuf με συνδεδεμένη λίστα από mbuf

Ο παραγωγός παραμένει απλός γιατί το μόνο που έχει να κάνει είναι να

συνδέσει το νέο στοιχείο στο τελευταίο στοιχείο της λίστας. Ο καταναλωτής όμως ελέγχει αν το στοιχείο που αφαιρεί είναι το τελευταίο και, αν είναι έτσι, δεν το αφαιρεί αλλά θέτει μια επιλογή (που την ονομάσαμε M_CORAL) στο mbuf ώστε την επόμενη φορά θα αγνοήσει τα περιεχόμενα του και, αν υπάρχουν πλέον άλλα στοιχεία στη λίστα θα το ελευθερώσει.



Σχήμα 4.4: Sockbuf με συνδεδεμένο ένα mbuf με την επιλογή M_CORAL

Μαζί με την ανάγκη να υπάρχει πάντα ένας δείκτης στο τελευταίο στοιχείο της λίστας, ο κώδικας του καταναλωτή έγινε αρκετά πιο πολύπλοκος από ο,τι οι άλλες μας προσεγγίσεις. Επιπλέον, υπήρχε το ζήτημα ότι, στην πράξη, τα περισσότερα mbuf είχαν δεσμευτεί μαζί με εξωτερικό ενταμιευτή ο οποίος θα έπρεπε να αποδεσμευτεί μαζί τους. Αφού το mbuf δεν μπορούσε να αποδεσμευτεί θα έπρεπε να κρατάμε άσκοπα και τον (μεγάλου μεγέθους) εξωτερικό ενταμιευτή (σχ. 4.4. Αυτό είναι ένα θέμα που ελπίζουμε να λύσουμε σε μια επανεξέταση του προβλήματος.

cupholders: Η τρίτη προσέγγιση ξεπερνά το πρόβλημα προσθέτοντας ένα επίπεδο ανακατεύθυνσης. Αντί να κρατάμε στη δομή sockbuf μια συνδεδεμένη λίστα από δομές mbuf, κρατάμε μια συνδεδεμένη λίστα από δομές cupholder που περιέχουν ένα δείκτη σε δομή mbuf. Έτσι όπως και παραπάνω, η τελευταία δομή cupholder της λίστας δεν μπορεί ποτέ να αφαιρεθεί, αλλά μπορούμε να αποδεσμεύσουμε τη δομή mbuf στην οποία δείχνει.

Όπως αναφέραμε (1.6)για πρωτόκολλα με επαναμεταδόσεις (όπως το TCP) τα δεδομένα παραμένουν στον ενταμιευτή αποστολής μέχρι να επιβεβαιωθεί η ορθή παραλαβή τους. Για να διαφοροποιήσουμε τα νέα δεδομένα από αυτά που περιμένουν επιβεβαίωση αναγκαστήκαμε να απωλέσουμε μέρος της απλότητας του παραπάνω αλγορίθμου στην πραγματική υλοποίηση.

4.2.2 Απροσδιοριστία των δεδομένων του sockbuf

Ανεξάρτητα από την επιλεχθείσα προσέγγιση, ένα sockbuf που προσπελαύνεται χωρίς αμοιβαίο αποκλεισμό έχει μια θεμελιώδη διαφορά απ ό το παραδοσιακό sockbuf: πλέον είναι αδύνατο να αποφανθούμε για τον ακριβή αριθμό των bytes που είναι αποθηκευμένα στο sockbuf. Πράγματι, αμέσως αφότου υπολογίσουμε το πλήθος των bytes είναι δυνατό η άλλη πλευρά να προσθέσει ή να αφαιρέσει δεδομένα ακυρώνοντας τον υπολογισμό μας.

Στην πράξη όμως, αυτό δεν παρουσιάζει σημαντικά προβλήματα όσο είμαστε αρκετά προσεκτικοί στη χρήση της προσεγγιστικά υπολογισμένης τιμής. Πράγματι, το νούμερο που υπολογίζουμε από την πλευρά του καταναλωτή είναι ένα κάτω όριο για τον πραγματικό αριθμό των bytes στο sockbuf. Αφού υπάρχει ένας μόνο καταναλωτής, τα bytes στο sockbuf μπορούν μόνο να αυξηθούν, αν ο παραγωγός προσθέσει δεδομένα. Αντίθετα, το νούμερο που υπολογίζει η πλευρά του παραγωγού αποτελεί ένα άνω όριο στον αριθμό των bytes. Έτσι, αν ο καταναλωτής υπολογίσει ότι τα περιεχόμενα του sockbuf είναι αρκετά για να τα διαβάσει, μπορεί να είναι σίγουρος ότι αυτό δεν πρόκειται να αλλάξει μετά τον έλεγχο. Παρόμοια, αν ο παραγωγός βρει ότι υπάρχει χώρος στο sockbuf για να αποθηκεύσει δεδομένα, το sockbuf δεν πρόκειται να γεμίσει από κάποιον άλλο παραγωγό. Δημιουργήσαμε μια νέα προγραμματιστική διεπαφή (API) για την πρόσβαση στο sockbuf ώστε ο χρήστης να “βλέπει” ένα σταθερό νούμερο για τα περιεχόμενα του sockbuf στη διάρκεια μιας ενέργειας.

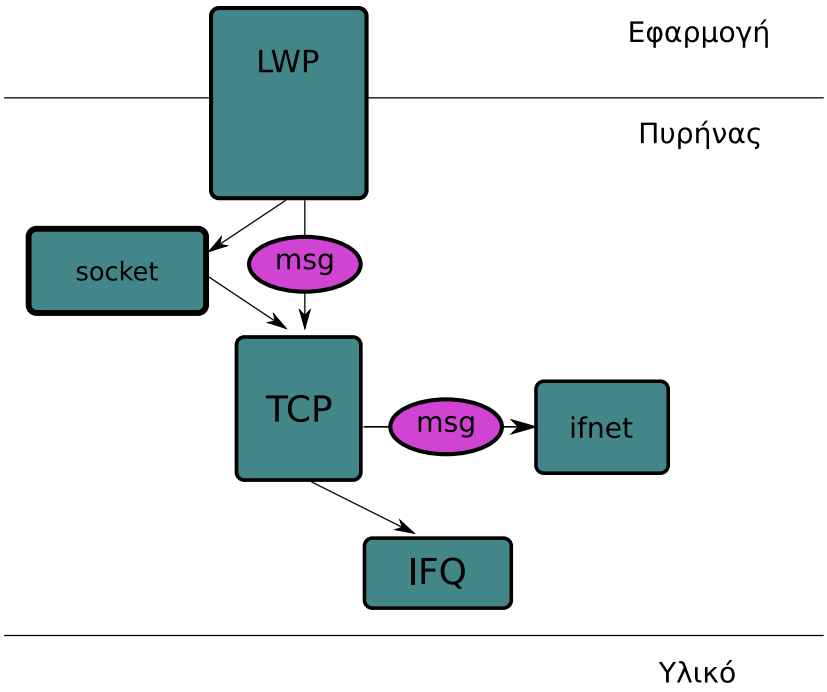
4.2.3 Χαμένη αφύπνιση

Τί συμβαίνει όμως όταν ο καταναλωτής δεν βρει αρκετά δεδομένα στο sockbuf; Τότε, εκτός αν η εφαρμογή έχει ζητήσει κάτι διαφορετικό, η κλήση συστήματος θα μπλοκάρει περιμένοντας ειδοποίηση όταν παραληφθούν νέα δεδομένα ή αλλάξει κάτι στην κατάσταση της σύνδεσης. Αν δεν είμαστε προσεκτικοί, μπορεί τα δεδομένα που περιμένει ο καταναλωτής να φτάσουν σε ακατάλληλη στιγμή έτσι ώστε η αφύπνιση να γίνει πριν το μπλοκάρισμα αλλά μετά τον έλεγχο για την ποσότητα των δεδομένων. Σε αυτή την περίπτωση, αν δεν πρόκειται να παραληφθούν περισσότερα δεδομένα χωρίς να πράξει κάτι ο καταναλωτής, το μπλοκάρισμα θα διαρκέσει για πάντα.

Ο τρόπος που λύσαμε το πρόβλημα της χαμένης αφύπνισης (lost wakeup)

ήταν, αντί για να μπλοκάρει περιμένοντας αφύπνιση, ο καταναλωτής να στέλνει ένα μήνυμα και να μπλοκάρει περιμένοντας απάντηση. Το μήνυμα ζητά από το νήμα πρωτοκόλλου να ελέγξει αν ικανοποιούνται οι συνθήκες για αφύπνιση και μόνο τότε να απαντήσει σε αυτό. Με αυτή τη διαδικασία, δεν υπάρχει πιθανότητα χαμένης αφύπνισης.

4.3 Τοποθέτηση των δεδομένων στον ενταμιευτή αποστολής



Σχήμα 4.5: Αποστολή δεδομένων (τροποποιημένη)

Σε μία κλήση αποστολής δεδομένων, ο αποστολέας έστελνε ένα μήνυμα στο νήμα πρωτοκόλλου για να τοποθετήσει εκείνο τα δεδομένα προς αποστολή στον ενταμιευτή sockbuf. Πλέον, είναι δυνατόν για τον αποστολέα να τοποθετεί τα δεδομένα στο sockbuf απευθείας (φροντίζοντας μόνο για αμοιβαίο αποκλεισμό με άλλους αποστολείς). Κατόπιν, στέλνει ένα ασύγχρονο μήνυμα για να ενημερώσει το νήμα πρωτοκόλλου για την ύπαρξη νέων δεδομένων στο sockbuf αποστολής.

Όταν το νήμα παραλάβει το μήνυμα, φροντίζει για την αποστολή όσων δεδομένων υπάρχουν στον ενταμιευτή.

4.4 Παράλληλη πρόσβαση στον πίνακα κατακερματισμού για το UDP

Στην ενότητα 3.6 είδαμε ότι ο πίνακας κατακερματισμού για τις δομές inpcb του TCP είναι ξεχωριστός για κάθε επεξεργαστή, ενώ το UDP χρησιμοποιεί ένα μοναδικό πίνακα κατακερματισμού. Η πρόσβαση στον πίνακα αυτόν απαιτεί την κατοχή του BGL. Για να επιτρέψουμε στα νήματα του UDP να τρέχουν χωρίς την BGL χρειαζόταν να καταργήσουμε αυτή την απαίτηση.

Ιδανικά θα θέλαμε να ακολουθήσουμε τη ίδια προσέγγιση με το TCP, χρησιμοποιώντας διαφορετικούς πίνακες σύμφωνα με μια συνάρτηση κατακερματισμού. Αλλά τι θα μπορούσαμε να χρησιμοποιήσουμε σαν είσοδο της συνάρτησης; Η δομή inpcb που αντιστοιχεί σε ένα socket UDP ίσως έχει ένα έγκυρο ζευγάρι <απομακρυσμένη διεύθυνση, απομακρυσμένη θύρα>, ίσως και όχι, αν η εφαρμογή δεν έχει κάνει κλήση connect ([STEVEN98]). Ακόμα χειρότερα, το ζευγάρι αυτό μπορεί να αλλάξει αυθαίρετα με μια νέα κλήση connect. Η τοπική διεύθυνση είναι επίσης μεταβλητή. Αν αποφασίσουμε να χρησιμοποιήσουμε κάποιο από αυτά τα τρία πεδία σαν είσοδο στη συνάρτηση κατακερματισμού, θα έπρεπε να είμαστε προετοιμασμένοι να μετακινήσουμε το inpcb μεταξύ επεξεργαστικών πυρήνων, κάθε φορά που αλλάζαμε κάποιο από αυτά. Δεδομένου ότι οι εφαρμογές που χρησιμοποιούν UDP πρέπει να είναι προετοιμασμένες για την απώλεια πακέτων, αυτό θα μπορούσαμε να το κατορθώσουμε σημειώνοντας στο παλιό inpcb ότι πρόκειται να καταστραφεί (όποτε το σύστημα θα το αγνοούσε χάνοντας πακέτα) και αποδεσμεύοντας το αφού εγκαταστήσουμε το καινούργιο.

Η λύση που επιλέξαμε προς το παρόν είναι να χρησιμοποιήσουμε μόνο την τοπική θύρα. Αυτή είναι μια απλή λύση (που επιτρέπει σε πολλές εφαρμογές UDP να χρησιμοποιήσουν ταυτόχρονα τις υπηρεσίες του πυρήνα) με κάποιους σημαντικούς περιορισμούς. Ας θεωρήσουμε ένα τυπικό εξυπηρετητή DNS. Σε αυτή την εφαρμογή δεν υπάρχει απομακρυσμένη διεύθυνση και θύρα, ούτε τοπική διεύθυνση. Έτσι τα πακέτα DNS θα επεξεργάζονται πάντα σε έναν μόνο από τους επεξεργαστές του συστήματος.

Μία άλλη προσέγγιση που θέλουμε να δοκιμάσουμε είναι η αντιγραφή των

inpcd στα οποία κάποια από τα πεδία που αναφέραμε δεν είναι ορισμένα, σε περισσότερους επεξεργαστές. Εκμεταλλευόμενοι το ότι το UDP δεν εγγυάται την παράδοση των πακέτων με τη σωστή σειρά, θα μπορούσαμε να έχουμε στη δομή socket ένα sockbuf για κάθε επεξεργαστικό πυρήνα του συστήματος. Τότε, για παράδειγμα, η κλήση συστήματος για ανάγνωση δεδομένων θα μπορούσε να διαβάζει δεδομένα από όλα τα socket με τη σειρά. Επειδή όμως αυτό δεν συμβαίνει πολύ συχνά στο διαδίκτυο, μόνο η δοκιμασία στην πράξη θα μπορέσει να μας πει αν οι υπάρχουσες εφαρμογές είναι έτοιμες να αντιμετωπίσουν ροές δεδομένων που είναι τυπικά αναδιαταγμένες.

4.5 Αλλαγές στη δομή socket

Η άλλη σημαντική και συχνά προσπελασσόμενη δομή που βασιζόταν στην BGL για να είναι συνεπής ήταν η δομή socket. Η δομή socket εξυπηρετεί, όπως είδαμε (3.7) πολλές λειτουργίες και για αυτό έπρεπε να εξετάσουμε κάθε πεδίο της χωριστά. Για κάθε πεδίο φτιάξαμε ένα δέντρο με τις συναρτήσεις που το τροποποιούν ή το διαβάζουν, τις συναρτήσεις που καλούν αυτές τις συναρτήσεις και ούτως καθεξής. Έτσι, μπορέσαμε να καθορίσουμε ποια πεδία προσπελαύνονται τόσο από νήματα πρωτοκόλλου όσο και από κλήσεις συστήματος και να εξετάσουμε πως αυτή θα μπορούσε να γίνει με ασφάλεια.

Για πεδία όπως τα so_type, so_proto που αρχικοποιούνται μαζί με το socket και μένουν αμετάβλητα δεν υπάρχει κανένα πρόβλημα. Ακόμα, αφού ο στόχος μας περιορίζεται στην βελτίωση της στοίβας πρωτοκόλλων, δεν ασχοληθήκαμε με τα πεδία που χρησιμοποιούνται για ασύγχρονη είσοδο/έξοδο (AIO) και για εξομείωση άλλων πυρήνων. Αυτά τα υποσυστήματα απαιτούν την BGL, οπότε δεν θα υπήρχε μετρήσιμο κέρδος αν αφαιρούσαμε αυτή την απαίτηση μόνο για το πεδίο της δομής socket.

Τα μονοπάτια τροποποίησης των επιλογών του socket ελέγχθηκαν επίσης και αφαιρέσαμε όσες υποθέσεις δεν ανταποκρίνοντας στη νέα πραγματικότητα.

Στην ενότητα 3.7 είδαμε τις λίστες που διατηρεί ο πυρήνας σε κάθε δομή socket η οποία δέχεται συνδέσεις. Όπως είναι φανερό, αν διαφορετικοί επεξεργαστές προσπαθούν να τροποποιήσουν τις λίστες ταυτόχρονα, αργά ή γρήγορα οι λίστες θα βρεθούν σε μη συνεκτική κατάσταση. Εδώ ακολουθήσαμε τη συνηθισμένη πρακτική και προσθέσαμε ένα κλείδωμα τύπου spin την οποία πρέπει

να κατέχει ο κώδικας που τροποποιεί ή διατρέχει τις λίστες και τα στοιχεία τους. Η πρόσβαση στις λίστες ολοκληρώνεται σε λίγους κύκλους ρολογιού, οπότε είναι προτιμητέο ένας επεξεργαστής να σπαταλήσει λίγες εντολές περιμένοντας παρά να μπλοκάρει και να εξυπηρετηθεί κάποια μικροδευτερόλεπτα (δηλαδή, ένα μεγάλο διάστημα) αργότερα. Με το ίδιο κλείδωμα προστατεύονται και τα πεδία στα οποία αποθηκεύεται το μήκος κάθε λίστας.

4.6 Επεξεργασία μηνυμάτων λάθους ICMP

Άλλη μια ενδιαφέρουσα επιπλοκή που αντιμετωπίσαμε ήταν η επεξεργασία των μηνυμάτων λάθους ICMP. Ένα μήνυμα ICMP για μια σύνδεση TCP μπορεί να παραδοθεί σε διαφορετικό επεξεργαστικό πυρήνα από αυτόν ο οποίος χειρίζεται τη σύνδεση που αφορά το πακέτο. Η λύση που δώσαμε ήταν να στέλνουμε ένα μήνυμα στο κατάλληλο νήμα πρωτοκόλλου για να εκτελέσει την διαδικασία εισαγωγής δεδομένων ελέγχου για το πακέτο ICMP.

Όταν πλέον είχε ολοκληρωθεί ένα μεγάλο μέρος των αλλαγών, αναδύθηκαν δευτερεύοντα σημεία ανταγωνισμού ανάμεσα στα νήματα πρωτοκόλλου. Για παράδειγμα, η ανάκτηση τυχαίων δεδομένων για το πεδίο identification της επικεφαλίδας IP απαιτούσε την κατοχή ενός κλειδώματος τύπου spin και αυτό έγινε σημείο στραγγαλισμού (bottleneck) για τις δοκιμές μας με μεγάλο αριθμό συνδέσεων.

Κεφάλαιο 5

Λεπτομέρειες της Υλοποίησης

Σε αυτό το κεφάλαιο θα παραθέσουμε αποσπάσματα από τον κώδικα της υλοποίησης των αλλαγών που περιγράψαμε. Στην πλειονότητα των περιπτώσεων, απαιτήθηκε τροποποίηση του υπάρχοντος κώδικα που δεν είναι εύκολο να παρουσιαστεί αυτόνομα. Ειδικά για τη δομή sockbuf όμως, χρειάστηκε να ξεκινήσουμε την υλοποίηση από την αρχή.

5.1 Η νέα δομή sockbuf

Η υλοποίηση στην οποία καταλήξαμε προς το παρόν για τη δομή sockbuf χρησιμοποιεί, όπως αναλύσαμε, τη δομή τύπου cupholder που παρουσιάζεται στο απόσπασμα 5.1.

Listing 5.1: struct cupholder

```
struct cupholder {
        struct cupholder *next;
        struct mbuf *m;
};

typedef struct cupholder *cupholder_t;
```

Το πεδίο m είναι ο δείκτης στην δομή τύπου mbuf που είναι το πραγματικό περιεχόμενο του sockbuf και το πεδίο next χρησιμεύει για να συνδέονται οι δομές cupholder σε μια μονά συνδεδεμένη λίστα.

Η νέα δομή sockbuf παρουσιάζεται στο απόσπασμα 5.2.

Listing 5.2: struct sockbuf

```
struct sockbuf {
        struct cupholder *head; /* consumer   never NULL */
        struct cupholder *note; /* consumer   track new appends.
            Can be NULL */
        int      rbytes;        /* consumer   field can
           roll over */
        int      rmbufs;        /* consumer   field can
           roll over */
#ifdef __amd64__
        int      dummy[2];      /* cache line alignment */
#endif
        struct cupholder *tail; /* producer   never NULL */
        int      wbytes;        /* producer   field can
           roll over */
        int      wmbufs;        /* producer   field can
           roll over */
};
```

Είναι ξεκάθαρη η φροντίδα τα πεδία που τροποποιεί ο παραγωγός να ανήκουν σε διαφορετική γραμμή κρυφής μνήμης από αυτά που τροποποιεί ο καταναλωτής. Για το λόγο αυτό δεν αποθηκεύουμε το πλήθος των αποθηκευμένων bytes σε ένα πεδίο της δομής, αλλά όταν το χρειαζόμαστε το συνθέτουμε αφαιρώντας τα περιεχόμενα του πεδίου wbytes από αυτά του rbytes (βλ. απόσπασμα 5.3).

Listing 5.3: Συνάρτηση sb_cc_est

```
static __inline
int
sb_cc_est(struct sockbuf *sb)
{
        int len;

        len = (int)(sb >wbytes   sb >rbytes);
        KKASSERT(len >= 0);
        return (len);
}
```

Αφού πλέον δεν υπάρχει εγγύηση ότι το πλήθος των αποθηκευμένων δεδομένων θα μείνει σταθερό μετά τον υπολογισμό του, κάθε κομμάτι κώδικα που

το χρησιμοποιεί πρέπει είναι έτοιμο να αντιμετωπίσει μια τέτοια αλλαγή. Επειδή υπάρχουν αρκετά και πολύπλοκα σημεία τα οποία έχουν γραφτεί για μια πολύ πιο στατική κατάσταση, αλλά και για να είναι δυνατό να δουλέψει ο προγραμματιστής ευκολότερα και με περισσότερη ασφάλεια, αποφασίσαμε να εισάγουμε μια νέα προγραμματιστική διεπαφη που στηρίζεται στη δομή sb_reader (απόσπασμα 5.4

Listing 5.4: struct sb_reader

```
struct sb_reader {      /* sockbuf iterator (ro) */
        struct sockbuf *sb;
        struct cupholder *ch;
        struct mbuf *m;
        int flags;
        int mflags;
        int max_len;
        int len;
};
```

Αφού αρχικοποιηθεί η δομή με τη χρήση της συνάρτησης

```
void sb_reader_init(struct sb_reader *, struct sockbuf *);
```

συγκρατεί (στο πεδίο max_len) τον αριθμό των bytes που ήταν αποθηκευμένα στο sockbuf εκείνη τη στιγμή και συμπεριφέρεται σαν το sockbuf με το οποίο είναι συσχετισμένο να έχει παραμείνει σε εκείνη τη στιγμή. Ο χρήστης της διεπαφής, διαβάζει δεδομένα με τη χρήση της συνάρτησης

```
struct mbuf *sb_reader_next(struct sb_reader *, int *);
```

η οποία προσδιορίζει αν πρόκειται για το τέλος μιας εγγραφής γράφοντας στον ακέραιο η διεύθυνση του οποίου περάστηκε σαν παράμετρος (αν είναι μη μηδενική). Ο αριθμός των bytes που έχουν διαβαστεί μέχρι τώρα αποθηκεύεται στο πεδίο len της δομής sb_reader, ώστε σε κάθε κλήση της sb_reader_next() αυξάνεται, έως ότου φτάσει την τιμή του πεδίου max_len. Όταν συμβεί αυτό, το sockbuf θεωρείται άδειο. Το θετικό είναι ότι οι χρήστες της διεπαφής πρέπει ούτως ή άλλως να χειρίζονται σωστά την περίπτωση στην οποία νέα δεδομένα καταφθάνουν

αμέσως μόλις τελειώσουν με την πρόσβασή τους στο sockbuf, οπότε η αλλαγή μας δεν επιβάλει επιπλέον απαιτήσεις στον υπάρχοντα κώδικα.

Τέλος, θα παρουσιάσουμε τις συναρτήσεις για την προσθήκη / αφαίρεση δεδομένων από το sockbuf με την προσέγγιση που επιλέξαμε τελικά (απόσπασμα 5.5) και με την προσέγγιση του κυκλικού ενταμιευτή (απόσπασμα 5.6).

Listing 5.5: Συναρτήσεις _sb_enq, _sb_deq της υλοποίησης cupholder

```
/*
 * Append a new cupholder and mbuf.  If the existing tail
    cupholder is empty
 * we can just reuse it.  The mbuf may be chained via m_next but
     represents
 * a record bounded entity only if M_EOR is set in the first
    mbuf.
 *
 * The caller may be breaking up records separated by m_nextpkt.
      Once
 * in a sockbuf m_nextpkt must be NULL.  Set m_nextpkt to NULL
    here.
 */
static inline
int
_sb_enq(struct sockbuf *sb, struct mbuf *m)
{
        struct cupholder *ch;
        int len;

        KKASSERT((m->m_flags & M_SOCKBUF) == 0);
        m_set_flagsm(m, M_SOCKBUF);
        len = m_lengthm(m, NULL);
        m->m_nextpkt = NULL;    /* obsolete? */

        /*
         * Note race in new ch case.  The linkage is created
            before the
         * tail is moved.  Consumers should consume based on
            sb_cc_est(),
```

```
         * not based on ch->next, though I think the worst that
           happens
         * is you end up with an empty cupholder in the list,
           which we
         * allow.
         *
         * When optimizing re use of the last cupholder we have
           to determine
         * if the other side temporarily chained some mbufs.
           Checking whether
         * the sockbuf has a 0 count is sufficient.
         */
        if (sb->tail->m == NULL && sb->rbytes == sb->wbytes) {
                sb->tail->m = m;
        } else {
                ch = sb_allocate_ch(sb);
                ch->m = m;
                sb->tail->next = ch;
                sb->tail = ch;
        }

        /*
         * Currently the code checks for the existence of
         * data by testing the byte count, so make sure
         * that before we update the byte counters, the
         * data has already been added.  Changing the
         * network code to test for mbuf availability
         * should be doable but is not straightforward.
         */
        _sb_produce(sb, len, 1);

        return 0;
}

/*
 * Remove the head and return the mbuf.  The head must exist.
 *
 * The returned mbuf may be chained via m_next but represents a
   whole
```

```
 * record only if M_EOR is set in the first mbuf.
 */
static inline
struct mbuf *
_sb_deq(struct sockbuf *sb)
{
        struct cupholder *ch;
        struct mbuf *m;
        int len;
        static struct krate kr = { .freq = 3 };

        /*
         * Locate the head, dispose of any empty cupholders
           along
         * the way.  The tail cupholder is never disposed of.
         *
         * If note is pointing at the current head we must set
           it to NULL,
         * which will cause sb_next_note() to return the new
           head.
         */
        for (;;) {
                ch = sb >head;
                if (ch >m || ch >next == NULL)
                        break;
                cpu_lfence();           /* SMP race against tail
                    append */
                if (ch >m)
                        break;
                if (sb >note == ch)
                        sb >note = NULL;
                sb >head = ch >next;
                sb_dispose_ch(sb, ch);
        }

        /*
         * We must have found a mbuf
         */
        m = ch >m;
```

```
        KKASSERT(m != NULL);
        if ((m->m_flags & M_SOCKBUF) != 0) {
                kprintf(&kr, "m->m_flags & M_SOCKBUF != 0 "
                    "(%d)\n", m->m_flags);
        }
        len = m_lengthm(m, NULL);
        _sb_consume(sb, len, 1);

        /*
         * We can dispose of ch if it is not the tail.  Note the
             SMP race
         * here.  Once we set ch->m to NULL the producer can
             instantly
         * load it with a new mbuf and possibly also chain
             another mbuf,
         * so we can't clear the mbuf until after we've checked
             the tail.
         */
        if (sb->note == ch)
                sb->note = NULL;
        if (ch->next) {
                cpu_ccfence();
                ch->m = NULL;
                sb->head = ch->next;
                sb_dispose_ch(sb, ch);
        } else {
                cpu_ccfence();
                ch->m = NULL;
        }
        m_clear_flagsm(m, M_SOCKBUF);
        return (m);
}
```

Listing 5.6: Συναρτήσεις _sb_enq, _sb_deq της υλοποίησης κυκλικού εντα-μιευτή

```
static inline int
_sb_enq(struct sockbuf *sb, struct mbuf *m)
{
        int len;
```

```
        KKASSERT(!(m->m_flags & M_SOCKBUF));
        if (sb->mbring[sb->write]) {
                panic("buffer overflow\n");
                return !0;
        }
        m->m_nextpkt = NULL;
        len = m_lengthm(m, NULL);
        m_set_flagsm(m, M_SOCKBUF);
        sb->mbring[sb->write] = m;
        sb->write = _sb_next(sb->write);
        /*
         * Currently the code checks for the existence of
         * data by testing the byte count, so make sure
         * that before we update the byte counters, the
         * data has already been added. Changing the
         * network code to test for mbuf availability
         * should be doable but is not straightforward.
         */
        _sb_produce(sb, len);

        return 0;
}

static inline struct mbuf *
_sb_deq(struct sockbuf *sb)
{
        struct mbuf *m;
        int len;

        if (!(m = sb->mbring[sb->read])) {
                panic("buffer underflow\n");
                return m;
        }
        KKASSERT(!!(m->m_flags & M_SOCKBUF));
        len = m_lengthm(m, NULL);
        _sb_consume(sb, len);
        sb->mbring[sb->read] = NULL;
        sb->read = _sb_next(sb->read);
```

```
        m_clear_flagsm(m, M_SOCKBUF);

        return m;
}
```

5.2 Χαμένη αφύπνιση

Κατά τον έλεγχο για το αν υπάρχουν δεδομένα στον ενταμιευτή λήψης, η κλήση συστήματος μπλοκάρει στέλνοντας ένα σύγχρονο μήνυμα στο νήμα πρωτοκόλλου (απόσπασμα 5.7). Το μήνυμα προσδιορίζει ότι το νήμα πρέπει να απαντήσει όταν η συνάρτηση socanreceive_predicate (απόσπασμα 5.8) επιστρέψει “αληθές”.

Listing 5.7: Συνάρτηση soreceive

```
port = so >so_proto >pr_mport(so, NULL, NULL, PRU_PRED);
netmsg_init_abortable(&msg.nm_netmsg, &curthread >td_msgport,
                      0, netmsg_so_notify,
                      netmsg_so_notify_doabort);
msg.nm_predicate = socanreceive_predicate;
msg.nm_so = so;
msg.nm_etype = NM_REVENT;
msg.nm_rbytes = sb_reader_len(&sr);
if (!m) {
        ++msg.nm_rbytes;
} else {
        /*
         * TBD: try to be smart here
         */
        ++msg.nm_rbytes;
}
error = lwkt_domsg(port, &msg.nm_netmsg.nm_lmsg, PINTERLOCKED);
```

Listing 5.8: Συνάρτηση socanreceive_predicate

```
static boolean_t
socanreceive_predicate(struct netmsg *msg0)
{
        struct netmsg_so_notify *msg = (struct netmsg_so_notify
            *)msg0;
```

```
        struct socket *so = msg >nm_so;

        msg >nm_netmsg.nm_lmsg.ms_error = 0;
        if (so >so_state & SS_CANTRCVMORE)
                return TRUE;
        if (sb_cc_est(&so >so_rcv.sb) >= msg >nm_rbytes)
                return TRUE;
        if (!(so >so_state & (SS_ISCONNECTING|SS_ISCONNECTED)))
                return TRUE;
        return FALSE;
}
```

5.3 Πρόσβαση στον ενταμιευτή αποστολής δεδομένων

Στο απόσπασμα 5.9 φαίνεται η τοποθέτηση των δεδομένων στον ενταμιευτή αποστολής. Σε αυτό το σημείο, τα δεδομένα προς αποστολή έχουν ήδη αντιγραφεί σε δομές mbuf. Αν το πρωτόκολλο δεν απαιτεί την ύπαρξη σύνδεσης, πιθανόν ο αποστολέας να χρειάζεται να προσδιορίσει και τη διεύθυνση του παραλήπτη. Αν είναι έτσι, τότε αναγκαζόμαστε να στείλουμε ένα μήνυμα με τη χρήση της συνάρτησης so_pru_send() στο νήμα πρωτοκόλλου, το οποίο και θα τοποθετήσει τα δεδομένα στον ενταμιευτή αποστολής. Το ίδιο θα συμβεί και αν χρειάζεται να στείλουμε επείγοντα δεδομένα. Για κανονικά δεδομένα ωστόσο, ο αποστολέας πλέον τοποθετεί τις δομές mbuf στον ενταμιευτή αποστολής και απλά ειδοποιεί το νήμα πρωτοκόλλου ότι υπάρχουν δεδομένα προς αποστολή. Έτσι, το νήμα μπορεί να στείλει τα δεδομένα κατά την εξυπηρέτηση ενός προηγούμενου μηνύματος για αποστολή μηνυμάτων.

Listing 5.9: Συνάρτηση sosend

```
if (addr) {
        error = so_pru_send(so, pru_flags, top,
                            addr, control, td);
} else if ((pru_flags & M_OOB) ||
           pr >pr_usrreqs >pru_notify == NULL) {
        error = so_pru_send(so, pru_flags, top,
                            addr, control, td);
```

```
} else if (control) {
        control >m_flags |= pru_flags;
        sb_append_control(&so >so_snd.sb, top, control);
        error = so_pru_notify(so);
} else {
        top >m_flags |= pru_flags;
        sb_append(&so >so_snd.sb, top);
        error = so_pru_notify(so);
}
```

Κεφάλαιο 6

Επίδοση

6.1 Υπέρβαση των προβλημάτων στην πραγματοποίηση των μετρήσεων

Αφού είχε ολοκληρωθεί ένα μεγάλο μέρος των αλλαγών, ξεκινήσαμε την προσπάθεια προσδιορισμού της επίδοσης του συστήματος. Ο βασικός στόχος ήταν η επαλήθευση ότι η ρυθμαπόδοση του συστήματος αυξάνει με την χρησιμοποίηση περισσότερων επεξεργαστικών πυρήνων και ότι η BGL δεν αποτελεί πλέον αντικείμενο ανταγωνισμού ανάμεσα στα νήματα πρωτοκόλλων καθώς και τις κλήσεις συστήματος που διενεργούν είσοδο/έξοδο από/προς το δίκτυο. Δευτερευόντως, θέλαμε να αποκτήσουμε μία εικόνα της δυναμικής συμπεριφοράς του συστήματος και να εντοπίσουμε τα μέρη του συστήματος που εμποδίζουν την κλιμάκωση σε περισσότερους επεξεργαστές.

Για τον πρώτο στόχο θα θέλαμε, ιδανικά, να διενεργήσουμε μια σειρά μετρήσεων της ρυθμαπόδοσης με διαφορετικούς βαθμούς παραλληλισμού των προγραμμάτων αποστολής και ληψης δεδομένων σε μηχανήματα με αρκετούς επεξεργαστικούς πυρήνες. Δυστυχώς, έγινε φανερό πολύ νωρίς ότι δεν θα ήταν δυνατό να παρουμε χρήσιμες μετρήσεις στο διαθέσιμο υλικό. Το πρόβλημα ήταν ότι ένα γραμμικό πρόγραμμα (δηλαδή, ένα πρόγραμμα που δεν εκμεταλλεύεται παρά μόνο έναν επεξεργαστή) ήταν ικανό να κορέσει μια ζεύξη με ρυθμό μετάδοσης ενός Gigabit ανά δευτερόλεπτο. Συνεπώς, τόσο ο πυρήνας του DragonFlyBSD χωρίς τροποποιήσεις (θα αναφερόμαστε σε αυτόν σαν “master”, που είναι το όνομα του κλαδιού του στο σύστημα διαχείρισης εκδόσεων του DragonFlyBSD) όσο και ο

πυρήνας με τις τροποποιήσεις μας (θα τον αναφέρουμε σαν “netmp”, για τον ίδιο λόγο) έφταναν στο σημείο κορεσμού χωρίς να υπάρχει μεγάλη διαφοροποίηση με τον βαθμό παραλληλισμού.

Για να υπερβούμε αυτόν τον περιορισμό, έπρεπε να χρησιμοποιήσουμε πολλαπλές διεπαφές ή διεπαφές με ταχύτερο ρυθμό μετάδοσης. Η χρήση της διεπαφής επανακύκλωσης πακέτων (loopback), η οποία είναι αρκετά πιο γρήγορη, δεν ανταποκρίνεται σε κάποιο ρεαλιστικό σενάριο χρήσης και επιπλέον τα νήματα του αποστολέα και του παραλήπτη θα ανταγωνίζονταν για χρόνο εκτέλεσης στους επεξεργαστές του συστήματος κάτι το οποίο θα άλλαζε σημαντικά τα χαρακτηριστικά της δοκιμής.

Καταλήξαμε να χρησιμοποιήσουμε τις κάρτες 10G-PCIE-8B-C+E της εταιρείας Myricom οι οποίες έχουν ρυθμό μετάδοσης 10 Gigabit ανά δευτερόλεπτο. Ωστόσο, ο πυρήνας του DragonFlyBSD δεν είχε υποστήριξη για αυτές τις κάρτες, οπότε αναγκαστήκαμε να μεταφέρουμε τον κατάλληλο οδηγό από το λειτουργικό σύστημα FreeBSD. Η μεταφορά ολοκληρώθηκε σχετικά γρήγορα, με το μειονέκτημα ότι δεν ενεργοποιήσαμε τις δυνατότητες του οδηγού για πολλαπλούς κυκλικούς ενταμιευτές για αποστολή / λήψη (βλ. 3.8), τη δυνατότητα τεμάχισης των πακέτων TCP από τη διεπαφή (TSO) και την υποστήριξη μεγαλύτερης μέγιστης μονάδας μεταφοράς (MTU). Παρά τη νεότητα του οδηγού, μπορέσαμε να επιτύχουμε ικανοποιητική, προς το παρόν, απόδοση.

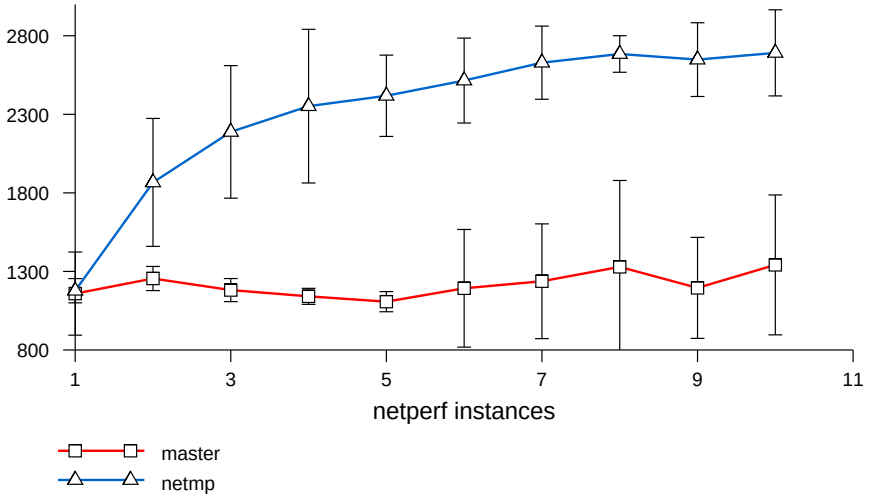
6.2 Μέθοδος μετρήσεων

Τα μηχανήματα που χρησιμοποιήσαμε για τις μετρήσεις είχαν όμοιο υλικό, η περιγραφή του οποίου δίνεται στον πίνακα 6.2. Το σχήμα 2.1 δίνει μια εικόνα της αρχιτεκτονικής του συστήματος που χρησιμοποιήσαμε. Οι κάρτες δικτύου ήταν συνδεδεμένες απευθείας (χωρίς τη μεσολάβηση μεταγωγέα) με χάλκινα καλώδια με βύσματα τύπου CX4.

Για τις μετρήσεις μας χρησιμοποιήσαμε την έκδοση 2.3.1 του προγράμματος netperf. Ξεκινήσαμε παράλληλα n τον αριθμό διεργασίες netperf οι οποίες εκτελούσαν τη λειτουργία TCP_STREAM από το ένα μηχάνημα στο άλλο. Κατά τη λειτουργία TCP_STREAM, το netperf δημιουργεί μία σύνδεση TCP μέσω της οποίας αποστέλει δεδομένα με το μέγιστο δυνατό ρυθμό. Στις δοκιμές μας χρησιμοποιήσαμε μέγεθος ενταμιευτή ίσο με 1048576 bytes. Ο μέσος χρόνος διαδρομής

Υλικό	
Αριθμός επεξεργαστών	2
Τύπος επεξεργαστή	Intel® 64-bit Xeon® Quad-Core
Πυρήνες/Επεξεργαστή	4
Μέγεθος κεντρικής μνήμης	2GB
Κάρτα δικτύου 10Gb	Myricom 10G-PCIE-8B-C+E

μετ'επιστροφής ήταν 0,363 μικροδευτερόλεπτα με τυπική απόκλιση 0,034. Το γινόμενο ρυθμαπόδοσης - καθυστέρησης είναι $363ns \quad 1\ 25GB/s = 453bytes$, επομένως το μέγεθος αυτό είναι αρκετά μεγάλο. Στο τέλος της δοκιμής προσθέταμε τη ρυθμαπόδοση καθε νήματος netperf για να υπολογίσουμε τη συνολική ρυθμαπόδοση.



Σχήμα 6.1: Γράφημα των μετρήσεων με το πρόγραμμα netperf

6.3 Σχολιασμός μετρήσεων

Τα αποτελέσματα των μετρήσεων που διενεργήσαμε παρουσιάζονται γραφικά στο σχήμα 6.1. Καταρχήν, παρατηρούμε τη μεγάλη τιμή της τυπικής απόκλισης σε ορισμένες από τις μετρήσεις. Αυτό είναι σε κάποιο βαθμό αναμενόμενο, αφού

με βάση τον αριθμό της θύρας TCP στον αποστολέα και τον παραλήπτη (νούμερα τα οποία επιλέγονται τυχαία), υπολογίζεται το νήμα πρωτοκόλλου που θα χειριστεί τη σύνδεση. Σε ορισμένες περιπτώσεις, ένα νήμα πρωτοκόλλου θα καλείται να χειριστεί περισσότερες από μία συνδέσεις TCP παρά το ότι υπάρχουν νήματα σε αδράνεια. Αυτό είναι ένα εγγενές μειονέκτημα της προσέγγισής μας το οποίο ωστόσο αναμένουμε να εξασθενεί όσο αυξάνεται ο αριθμός των ταυτόχρονων συνδέσεων περα από το πλήθος των διαθέσιμων επεξεργαστικών πυρήνων. Ακόμα, η απόδοση εξαρτάται και από το αν ο πυρήνας θα αποφασίσει να εκτελέσει το νήμα του netperf στον ίδιο επεξεργαστικό πυρήνα ή σε πυρήνα του ίδιου επεξεργαστή με το νήμα πρωτοκόλλου που χειρίζεται τη σύνδεση. Ο δεύτερος παράγοντας προσθέτει επιπλέον τυχαιότητα στα αποτελέσματα των μετρήσεών μας. Από το γεγονός ότι η τυπική απόκλιση δεν μεταβάλλεται ομαλά, συμπεραίνουμε ότι επηρεάζεται και απο κάποιον τρίτο (ή ίσως και άλλους) παράγοντα, πιθανώς αλληλεπίδραση με το χρονοδρομολογητή του πηρύνα. Το ενδεχόμενο αυτό χρήζει περαιτέρω έρευνας για να διαπιστώσουμε αν υπάρχει δυνατότητα βελτιστοποίησης.

Όπως αναμέναμε, ο πυρήνας master δεν επιτρέπει στα επιπλέον νήματα netperf να βελτιώσουν την απόδοση του συστήματος, καθώς όλα συναγωνίζονται για την απόκτηση της BGL. Η αποστολή / λήψη δεδομένων σε ένα λειτουργικό σύστημα στο οποίο η στοίβα πρωτοκόλλων είναι υλοποιημένη στον πυρήνα δεν μπορεί παρά να γραμμικοποιείται από τον αμοιβαίο αποκλεισμό που επιβάλλει η BGL. Αντίθετα, περιμένουμε η περαιτέρω αύξηση του ανταγωνισμού να χειροτερέψει την συνολική ρυθμαπόδοση.

Με τον πυρήνα netmp παρατηρούμε ότι η συνολική ρυθμαπόδοση του συστήματος αυξάνεται με τον αριθμό (n) των νημάτων netperf που χρησιμοποιούμε όσο αυτός ο αριθμός δεν ξεπερνά το πλήθος των επεξεργαστικών πυρήνων. Είναι φανερό ότι η αυξητική τάση μειώνεται με το n. Αυτό υποδηλώνει ότι υπάρχουν ακόμα παράγοντες που περιορίζουν τον παραλληλισμό ανάμεσα στα νήματα πρωτοκόλλου.

6.4 Περαιτέρω εργασία

Ένας τέτοιος παράγοντας είναι ο αμοιβαίος αποκλεισμός για την πρόσβαση στην ουρά αποστολής δεδομένων του οδηγού της κάρτας δικτύου. Αυτό το πρό-

βλημα μπορεί να παρακαμφθεί με την προσθήκη υποστηριξης για περισσότερες από μια ουρές αποστολής, εφόσον το υποστηρίζει ο οδηγός της κάρτας δικτύου. Η ενεργοποίηση αυτής της επιλογής στον οδηγό για τις κάρτες που χρησιμοποιήσαμε είναι ένα από τα επόμενα βήματα για την επίτευξη γραμμικής κλιμάκωσης της ρυθμαπόδοσης με τον αριθμό των επεξεργαστικών πυρήνων του συστήματος.

Ακόμα, χρήζει διερεύνησης αν ο οδηγός της κάρτας ή και το ίδιο το υλικό μπορούν να μας δώσουν υψηλότερη ρυθμαπόδοση, χωρίς την ενεργοποίηση των πολλαπλών κυκλικών ενταμιευτών λήψης / αποστολής, της μεγαλύτερης μέγιστης μονάδας μετάδοσης και την ενεργοποίηση μεταφοράς μέρους της επεξεργασίας πρωτοκόλλου στο υλικό της κάρτας.

Επιπλέον, κάποια μικρά μέρη του μονοπατιού αποστολής/λήψης δεδομένων απαιτούν ακόμα την κατοχή της BGL. Για παράδειγμα, η BGL απαιτείται για την αφύπνιση ενός νήματος και η αίρεση αυτού του περιορισμού απαιτεί σημαντική τροποποίηση του υποσυστήματος που χειρίζεται τις ειδοποιήσεις συμβάντων.

Τέλος, κρίνεται απαραίτητη η καταγραφή των ακριβών χρόνων γεγονότων συστήματος όπως η λήψη μηνυμάτων, το μπλοκάρισμα και η εκκίνηση ενός νήματος και η δημιουργία διακοπής από την κάρτα δικτύου. Αυτή η δυνατότητα υπάρχει για πολλά από τα συμβάντα που μας ενδιαφέρουν και την επεκτείναμε όπου δεν ήταν επαρκής. Προσθέσαμε επίσης κώδικα για λεπτομερή καταγραφή των ενεργειών του οδηγού των καρτών 10 Gigabit που χρησιμοποιήσαμε. Αυτό που μένει είναι η συγγραφή των βοηθητικών προγραμμάτων που θα μας επιτρέψουν να αναλύσουμε τον μεγάλο όγκο δεδομένων και να εντοπίσουμε τους παράγοντες που εμποδίζουν την πλήρως παράλληλη εκτέλεση της επεξεργασίας πρωτοκόλλου. Επιπλέον, θα είχε ιδιαίτερο ενδιαφέρον να εξάγουμε συμπεράσματα για το πως ο αλγόριθμος χρονοδρομολόγησης νημάτων του πυρήνα επηρεάζει τη ρυθμαπόδοση και την καθυστέρηση και να αναπτύξουμε τρόπους για τη βελτίωσή του.

Βιβλιογραφία

[SCHUEP08] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs: Embracing diversity in the Barrelfish manycore operating system. Boston, MA, USA, June 2008

[MCKUSI96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman: The Design and Implementation of the 4.4BSD Operating System

[MCKUSI04] Marshall Kirk McKusick, George V. Neville-Neil: The Design and Implementation of the FreeBSD Operating System

[STEVEN94] W. Richard Stevens: TCP/IP Illustrated, Volume 1: The Protocols

[WRIGHT95] Gary R. Wrighh, W. Richard Stevens: TCP/IP Illustrated, Volume 2: The Implementation

[STEVEN98] W. Richard Stevens: UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI

[STEVEN99] W. Richard Stevens: UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications

[LEMON] Jonathan Lemon: Kqueue: A generic and scalable event notification facility

[GIACOM08] John Giacomoni, Tipp Moseley, Manish Vachharajani: FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue

[MCKENN04] Paul E. McKenney: Exploiting Deferred Destruction: An analysis of Read-Copy-Update Techniques in Operating System Kernels

[MCKENN08] Paul E. McKenney: Sleepable Read-Copy Update

[MCKENN07] Paul E. McKenney: The design of preemptible read-copy-update