ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μηχανισμός Διαδιεργασιακής Επικοινωνίας Μηδενικών Αντιγράφων για το ΛΣ Linux με Δωρεά Σελίδων Εικονικής Μνήμης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Δ. Τσιρώνης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μηχανισμός Διαδιεργασιακής Επικοινωνίας Μηδενικών Αντιγράφων για το ΛΣ Linux με Δωρεά Σελίδων Εικονικής Μνήμης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Δ. Τσιρώνης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2η Νοεμβρίου 2016.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αναπ. Καθηγητής ΕΜΠ	Επίκουρος Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2016

. . . . . .

Νικόλαος Δ. Τσιρώνης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Μια από τις πιο κοινές λειτουργίες των ταυτόχρονων διεργασιών, στα σύγχρονα πολυπύρηνα συστήματα, είναι η ανταλλαγή μηνυμάτων. Επιπρόσθετα, το λειτουργικό σύστημα και οι εφαρμογές χρήστη έχουν ανάγκη να επικοινωνήσουν αποτελεσματικά και με τις ελάχιστες δυνατές επιβαρύνσεις. Συνήθως, όταν μεταδίδεται ένα μήνυμα, δημιουργούνται ένα ή περισσότερα αντίγραφα αυτού. Η εξάλειψη αυτών των αντιγράφων έχει την προοπτική να αυξήσει το ρυθμό μετάδοσης των δεδομένων και να μειώσει την καθυστέρηση της μεταφοράς τους. Ωστόσο, οι υπάρχουσες προσεγγίσεις για επικοινωνία μηδενικών αντιγράφων, όπως η χρήση μοιραζόμενης μνήμης, προϋποθέτουν την ύπαρξη "εμπιστοσύνης" ανάμεσα στις διεργασίες και δεν μπορούν να χρησιμοποιηθούν για την επικοινωνία αυθαίρετων διεργασιών. Στην παρούσα εργασία σχεδιάζουμε και υλοποιούμε δύο μηχανισμούς διαδιεργασιακής επικοινωνίας μηδενικών αντιγράφων. Οι μηχανισμοί αυτοί βασίζονται στη δωρεά σελίδων εικονικής μνήμης, ανάμεσα στις εμπλεκόμενες διεργασίες, και μπορούν να χρησιμοποιηθούν για την επικοινωνία τυχαίων διεργασιών. Το περιβάλλον ανάπτυξης είναι ο πυρήνας του λειτουργικού συστήματος Linux. Καθ' όλη τη διάρκεια της εργασίας παρουσιάζονται τα προβλήματα σχεδιασμού, που προκύπτουν σε κάθε επιμέρους στάδιο, και αιτιολογούνται οι αντίστοιχες επιλογές. Τέλος, οι μηχανισμοί αξιολογούνται πειραματικά σε ένα πραγματικό σύστημα και εξάγονται χρήσιμα συμπεράσματα.

Λέξεις Κλειδιά: *Διαδιεργασιακή Επικοινωνία, Μηδενικά Αντίγραφα, Linux, Εικονική Μνήμη, Οδηγός Συσκευής, Μεταφορά Δεδομένων, Ρυθμός Μεταφοράς, Καθυστέρηση Μεταφοράς, σωληνώσεις, unix sockets, μοιραζόμενη μνήμη, vmsplice, page re-mapping, page flipping*

Abstract

In contemporary multi-core and many-core hardware explicit message-passing between concurrent processes running on different cores is a common operation. Additionally, operating system kernels need to efficiently communicate with user applications. During the transmission of a message, one or more copies of the transmitted data are usually made. Eliminating these copies has the potential to increase transmission throughput and decrease latency. However, current zero-copy communication schemes, such as those that employ shared memory segments, rely on endpoints trusting each other. Thus, they cannot be used as a communication mechanism between arbitrary processes. The objective of this study is the design and implementation of two zero-copy interprocess communication mechanisms. These mechanisms are based on virtual memory gifting and they make no assumptions about the communicating endpoints. Thus, arbitrary processes can use them to safely communicate. The development framework is the Linux kernel. Throughout the course of our work, the problems concerning the design will be presented in detail in any individual stage and the corresponding choices will be justified. Finally, an experimental evaluation of the mechanisms in a real system will occur, providing results not only for further evaluation, but also as a source of useful conclusions.

Keywords: *IPC, Inter Process Communication, zero copy, Linux, Virtual Memory, Device Driver, Data Transfer, Throughput, Latency, multi-core, pipe, unix sockets, shared memory, vmsplice, page re-mapping, page flipping*

στους γονείς μου
Δημήτρη και Άννα
στην αδερφή μου
Ιωάννα

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου υπό την επίβλεψη του Καθηγητή κ. Νεκτάριου Κοζύρη.

Θα ήθελα να ευχαριστήσω θερμά τον κύριο Κοζύρη για την ευκαιρία που μου έδωσε, με την εκπόνηση της παρούσας διπλωματικής εργασίας, να ασχοληθώ με τον τομέα της Επιστήμης των Υπολογιστικών Συστημάτων.

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

Θα ήθελα να ευχαριστήσω, επίσης, τον Ευάγγελο Φωτόπουλο, που με τις επισημάνσεις του συνέβαλε σημαντικά στην επιμέλεια της εργασίας αυτής.

Τέλος, ένα μεγάλο ευχαριστώ στους γονείς μου και στην οικογένειά μου, για τη συνεχή τους στήριξη καθ' όλη τη διάρκεια των σπουδών μου, καθώς και στους φίλους και συμφοιτητές για τη στήριξη και τις ωραίες αναμνήσεις, που μου προσέφεραν όλα αυτά τα χρόνια.

Νίκος Τσιρώνης
Νοέμβριος 2016

Περιεχόμενα

1 Εισαγωγή

Στην επιστήμη των υπολογιστών ο όρος διαδιεργασιακή επικοινωνία (inter-process communication - IPC) αναφέρεται στο σύνολο των μηχανισμών, που παρέχει ένα λειτουργικό σύστημα, οι οποίοι επιτρέπουν στις εκτελούμενες διεργασίες (processes) να επικοινωνήσουν μεταξύ τους. Τα λειτουργικά συστήματα, εδώ και αρκετές δεκαετίες, υποστηρίζουν μια ποικιλία μηχανισμών διαδιεργασιακής επικοινωνίας. Από την εποχή που οι πρώτες "παράλληλες" εφαρμογές εκτελούταν σε συστήματα διαμοιρασμού χρόνου (time-sharing systems), οι πυρήνες των λειτουργικών συστημάτων και οι διεργασίες είχαν την ανάγκη να επικοινωνήσουν αποτελεσματικά. Ταυτόχρονα, οι πρώτες δικτυακές εφαρμογές εισήγαγαν την έννοια της αποστολής ενός μηνύματος σε μια απομακρυσμένη διεργασία, με αποτέλεσμα την εκτέλεση μιας απομακρυσμένης ενέργειας.

1.1 Κίνητρο

Σήμερα, αυτοί οι μηχανισμοί είναι πιο επίκαιροι από ποτέ: η διάδοση των πολυ-πύρηνων αρχιτεκτονικών σημαίνει ότι η ρητή ανταλλαγή μηνυμάτων, ανάμεσα σε ταυτόχρονες διεργασίες εκτελούμενες σε διαφορετικούς πυρήνες, είναι πλέον μια συνηθισμένη λειτουργία. Τα μελλοντικά λειτουργικά συστήματα είναι πιθανό να βασίζονται σε μεγάλο βαθμό στους μηχανισμούς IPC [BBD+09, WGB+10]. Ταυτόχρονα, καθώς τα κέντρα δεδομένων (data centers) αποτελούνται όλο και περισσότερο από πολυ-πύρηνα και πολυ-επεξεργαστικά μηχανήματα, μεγαλώνει διαρκώς η ανάγκη να γεφυρωθεί το χάσμα ανάμεσα στον παραδοσιακό πολυ-νηματικό προγραμματισμό στα συμμετρικά πολυεπεξεργαστικά συστήματα (multithreaded SMP systems) και την κατανεμημένη πα-

ράλληλη επεξεργασία των δεδομένων. Επιπλέον, βιβλιοθήκες ανταλλαγής μηνυμάτων, όπως αυτές που βασίζονται στο πρότυπο MPI [mpi16], στηρίζονται σε μηχανισμούς IPC για την αποδοτική υλοποίηση τους.

Είναι, λοιπόν, φανερό ότι υπάρχει ανάγκη τόσο για αξιολόγηση των υπαρχόντων μηχανισμών διαδιεργασιακής επικοινωνίας [SMS+12], όσο και για ανάπτυξη νέων, πιο αποδοτικών μηχανισμών.

Παραδοσιακά υπάρχουν δύο μοντέλα διαδιεργασιακής επικοινωνίας. Το μοντέλο μοιραζόμενης μνήμης και το μοντέλο ανταλλαγής μηνυμάτων.

Κάθε διεργασία έχει το δικό της χώρο εικονικών διευθύνσεων. Στο μοντέλο μοιραζόμενης μνήμης εγκαθιδρύεται, μέσω κάποιας κλήσης συστήματος, ένα τμήμα μοιραζόμενης μνήμης, που απεικονίζεται στους χώρους διευθύνσεων των συνεργαζόμενων διεργασιών. Στη συνέχεια, οι διεργασίες επικοινωνούν απευθείας, με χρήση εντολών load/store, αποφεύγοντας τις κλήσεις συστήματος. Ωστόσο, η προσέγγιση αυτή απαιτεί συγχρονισμό της πρόσβασης στα μοιραζόμενα δεδομένα και είναι επιρρεπής σε λάθη (για παράδειγμα race conditions, deadlocks, κ.τ.λ.). Τέλος, απαιτεί την ύπαρξη "εμπιστοσύνης" ανάμεσα στης διεργασίες και δεν μπορεί να χρησιμοποιηθεί για την επικοινωνία αυθαίρετων διεργασιών.

Ο πιο συνηθισμένος τρόπος διαδιεργασιακής επικοινωνίας είναι η ανταλλαγή μηνυμάτων, η οποία παρέχει ένα ρητό κανάλι επικοινωνίας. Η αποστολή και λήψη δεδομένων πραγματοποιείται με κλήσεις συστήματος της μορφής send() / receive(). Η προσέγγιση αυτή επιτρέπει τον ρητό διαμοιρασμό των δεδομένων, ευνοεί τη δομημένη σχεδίαση και δεν απαιτεί την ύπαρξη εμπιστοσύνης ανάμεσα στις διεργασίες. Δηλαδή, μπορεί να χρησιμοποιηθεί για την επικοινωνία τυχαίων διεργασιών. Ωστόσο, εισάγει επιπρόσθετο κόστος εξαιτίας της χρήσης κλήσεων συστήματος και της δημιουργίας αντιγράφων των μεταδιδόμενων δεδομένων.

1.2 Συνεισφορά της εργασίας

Οι μηχανισμοί IPC, που βασίζονται στο μοντέλο ανταλλαγής μηνυμάτων, δημιουργούν ένα ή περισσότερα αντίγραφα των μεταδιδόμενων δεδομένων. Παράλληλα, η σημασιολογία των παραδοσιακών κλήσεων συστήματος των UNIX/Linux, για διαδιεργασιακή επικοινωνία και Ε/Ε, βασίζεται στη δημιουργία αντιγράφων (copy semantics): Άμεση

επαναχρησιμοποίηση των buffers, μόλις η αντίστοιχη κλήση συστήματος επιστρέψει, χωρίς να αλλοιώνονται τα μεταδιδόμενα δεδομένα. Ωστόσο, η αντιγραφή των δεδομένων εισάγει επιπρόσθετο κόστος κατά τη μετάδοση ενός μηνύματος. Ιδανικά, θα θέλαμε μετάδοση μηδενικών αντιγράφων (zero-copy): Η αποφυγή "περιττών" αντιγράφων έχει τη προοπτική να αυξήσει το ρυθμό μετάδοσης των δεδομένων (throughput) και να μειώσει την καθυστέρηση μεταφοράς (latency).

Οι υπάρχουσες προσεγγίσεις για διαδιεργασιακή επικοινωνία μηδενικών αντιγράφων βασίζονται στο μοντέλο μοιραζόμενης μνήμης. Ωστόσο, όπως αναφέραμε, αυτή η προσέγγιση φέρνει τους προγραμματιστές αντιμέτωπους με προβλήματα ταυτοχρονισμού, δε διατηρεί τα παραδοσιακά copy semantics και δεν μπορεί να χρησιμοποιηθεί για την επικοινωνία αυθαίρετων διεργασιών.

Στην παρούσα εργασία συνδυάζουμε τα δύο μοντέλα διαδιεργασιακής επικοινωνίας για τη δημιουργία ενός μηχανισμού με τα εξής χαρακτηριστικά:

- Μηδενικά Αντίγραφα κατά τη μετάδοση
- Σέβεται τα παραδοσιακά copy semantics των Unix/Linux
- Μπορεί να χρησιμοποιηθεί για την επικοινωνία τυχαίων διεργασιών

Για να το πετύχουμε αυτό εκμεταλλευόμαστε το υποσύστημα εικονικής μνήμης του Linux και δίνουμε στον πυρήνα το ρόλο του συντονιστή, ώστε να εξασφαλίσει, μεταξύ άλλων, απομόνωση ανάμεσα στις διεργασίες και προστασία μνήμης.

Σχεδιάζουμε και υλοποιούμε δύο μηχανισμούς διαδιεργασιακής επικοινωνίας με τα παραπάνω χαρακτηριστικά. Η λειτουργία των μηχανισμών, που αναπτύξαμε, βασίζεται στη δωρεά σελίδων εικονικής μνήμης από τον αποστολέα στον παραλήπτη. Για να το πετύχουμε αυτό χρησιμοποιούμε μια τεχνική επανα-απεικόνισης των σελίδων από το χώρο διευθύνσεων του αποστολέα στο χώρο διευθύνσεων του παραλήπτη (page re-mapping ή page-flipping).

Αναλύουμε, λεπτομερώς, τα χαρακτηριστικά του υλικού και του λογισμικού που απαιτούνται για την υλοποίηση ενός μηχανισμού page re-mapping και για την ασφαλή επικοινωνία τυχαίων διεργασιών. Επιπλέον, περιγράφουμε αναλυτικά τα προβλήματα που προέκυψαν κατά την υλοποίηση αυτής της λειτουργικότητας. Τέλος, εξετάζουμε τις

επιδόσεις των μηχανισμών που υλοποιήσαμε και τις συγκρίνουμε με τις επιδόσεις των διαθέσιμων στο Linux μηχανισμών IPC. Στα πλαίσια των μετρήσεων αυτών, τροποποιήσαμε και επεκτείναμε ένα εργαλείο αξιολόγησης μηχανισμών διαδιεργασιακής επικοινωνίας, το ipc-bench [SMS+12], για να καλύψει καλύτερα τις ανάγκες της παρούσας εργασίας.

1.3 Οργάνωση κειμένου

Το παρόν κείμενο είναι δομημένο ως εξής:

Στο κεφάλαιο 2 παρουσιάζεται το απαραίτητο θεωρητικό υπόβαθρο για την κατανόηση της υπόλοιπης εργασίας.

Στο κεφάλαιο 3 παρουσιάζεται ένα εργαλείο αξιολόγησης μηχανισμών διαδιεργασιακής επικοινωνίας, το οποίο χρησιμοποιήθηκε καθ' όλη τη διάρκεια της εργασίας για την αξιολόγηση και βελτίωση των μηχανισμών που αναπτύξαμε.

Στο κεφάλαιο 4 παρουσιάζεται αναλυτικά ο σχεδιασμός του πρώτου από τους δύο μηχανισμούς που αναπτύξαμε, του zipc.

Στο κεφάλαιο 5 περιγράφεται η υλοποίηση του zipc.

Στο κεφάλαιο 6 παρουσιάζεται αναλυτικά ο σχεδιασμός του δεύτερου μηχανισμού, του zipc-tmpfs.

Στο κεφάλαιο 7 περιγράφεται η υλοποίηση του zipc-tmpfs.

Στο κεφάλαιο 8 παρουσιάζεται η πειραματική αξιολόγηση των μηχανισμών zipc και zipc-tmpfs και με βάση αυτή εξάγονται χρήσιμα συμπεράσματα.

Στο κεφάλαιο 9 παρουσιάζονται παρεμφερείς εργασίες και στο Κεφάλαιο 10 παρουσιάζουμε πιθανές επεκτάσεις των μηχανισμών που υλοποιήσαμε.

Τέλος, παραθέτουμε τη βιβλιογραφία που μελετήθηκε για την εκπόνηση της παρούσας εργασίας.

2

Θεωρητικό Υπόβαθρο

Στο κεφάλαιο αυτό γίνεται μια εισαγωγή στις έννοιες και τις δομές λειτουργίας, στις οποίες βασίζεται η σχεδίαση και υλοποίηση των μηχανισμών διαδιεργασιακής επικοινωνίας που αναπτύξαμε. Σκοπός του κεφαλαίου είναι η παρουσίαση των απαραίτητων θεωρητικών στοιχείων, καθώς και του περιβάλλοντος εργασίας, που καθόρισαν τις επιλογές μας και τον τρόπο προγραμματισμού.

Το κεφάλαιο χωρίζεται σε τρία βασικά μέρη: στο πρώτο αναλύεται ο πυρήνας του Linux με έμφαση στα σημεία που συνδέονται άμεσα με το αντικείμενο της παρούσας εργασίας, στο δεύτερο αναλύεται το υποσύστημα διαχείρισης μνήμης του Linux και στο τρίτο γίνεται περιγραφή των υπαρχόντων μηχανισμών διαδιεργασιακής επικοινωνίας στον πυρήνα του Linux.

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

Η κατανόηση του υποσυστήματος διαχείρισης μνήμης του Linux αποτελεί απαραίτητη προϋπόθεση για την κατανόηση των μηχανισμών που αναπτύχθηκαν στην παρούσα εργασία. Για το λόγο αυτό αφιερώνουμε ένα ξεχωριστό κεφάλαιο για την περιγραφή των δομών και λειτουργιών που είναι απαραίτητες για την κατανόηση της σχεδίασης και υλοποίησης των μηχανισμών IPC που δημιουργήσαμε.

Τέλος, ιδιαίτερα σημαντική είναι η κατανόηση των υπαρχόντων μηχανισμών IPC, που είναι διαθέσιμοι στον πυρήνα του Linux. Η κατανόηση του τρόπου λειτουργίας και των επιδόσεων των ήδη υπαρχόντων μηχανισμών στο Linux αποτελεί τη βάση για την ανάπτυξη ενός νέου μηχανισμού. Για το λόγο αυτό εξετάζουμε αναλυτικά τις γενικές στρατηγικές που είναι διαθέσιμες σε έναν πυρήνα για τη μεταφορά δεδομένων ανάμεσα σε διεργασίες και τον τρόπο που ο πυρήνας του Linux υλοποιεί αυτές τις στρατηγικές.

2.1 Το Linux

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

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

Στη συνέχεια, ξεκινάμε με μια γενική περιγραφή της αρχιτεκτονικής του πυρήνα Linux, στην οποία μελετάμε τη δομή, τη λειτουργία και τους τρόπους διασύνδεσής του με το υπόλοιπο λειτουργικό σύστημα, τις εφαρμογές αλλά και το υλικό. Ακολουθεί λεπτομερής περιγραφή των υποσυστημάτων που μελετήθηκαν για το σχεδιασμό και την υλοποίηση της εργασίας αυτής.

2.1.1 Αρχιτεκτονική του πυρήνα

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

Στο Linux διάφορες ταυτόχρονες διεργασίες επιτελούν διαφορετικές λειτουργίες και είναι αναγκαίο να διασφαλίζεται η εύρυθμη εκτέλεση τους. Η κάθε διεργασία ζητά πόρους του συστήματος, όπως υπολογιστική δύναμη, μνήμη, συνδεσιμότητα δικτύου ή κάτι άλλο. Ο πυρήνας είναι ένα μεγάλο κομμάτι εκτελέσιμου κώδικα που διαχειρίζεται τους πόρους[CRKH05]. Παρόλο που ο διαχωρισμός των λειτουργιών του πυρήνα δεν είναι πάντα σαφώς ορισμένος, ο ρόλος του μπορεί να χωριστεί (όπως φαίνεται και στο Σχήμα 2.1) στα ακόλουθα κομμάτια:

- **Process Management** (διαχείριση διεργασιών)
  Ο πυρήνας δημιουργεί και καταστρέφει διεργασίες και χειρίζεται τη σύνδεσή τους με τον "έξω κόσμο" (είσοδος / έξοδος). Η επικοινωνία ανάμεσα στις διεργασίες (μέσω σημάτων, σωληνώσεων ή άλλων μηχανισμών διαδιεργασιακής επικοινωνίας) είναι βασική στη λειτουργικότητα του όλου συστήματος και ελέγχεται επίσης από τον πυρήνα. Επιπροσθέτως, ο χρονοδρομολογητής, ο οποίος ελέγχει πώς οι διεργασίες μοιράζονται τον επεξεργαστή, είναι κι αυτός κομμάτι της διαχείρισης διεργασιών. Γενικότερα, η διαχείριση διεργασιών που επιτελεί ο πυρήνας υλοποιεί ένα επίπεδο αφαίρεσης για διεργασίες που εκτελούνται σε έναν ή περισσότερους επεξεργαστές.

- **Memory Management** (διαχείριση μνήμης)
  Η μνήμη του υπολογιστικού συστήματος είναι ένας σημαντικός πόρος και η πολιτική που χρησιμοποιείται για τη διαχείρισή της είναι κρίσιμη για την απόδοση του συστήματος. Ο πυρήνας δημιουργεί ένα χώρο εικονικών διευθύνσεων για κάθε διεργασία πάνω από τους πεπερασμένους διαθέσιμους πόρους. Τα διαφορετικά κομμάτια του πυρήνα αλληλεπιδρούν με το υποσύστημα διαχείρισης μνήμης μέσω ενός συνόλου συναρτήσεων που ποικίλλουν από το απλό ζεύγος malloc()

/ free() μέχρι και πολύ πιο πολύπλοκες συναρτήσεις.

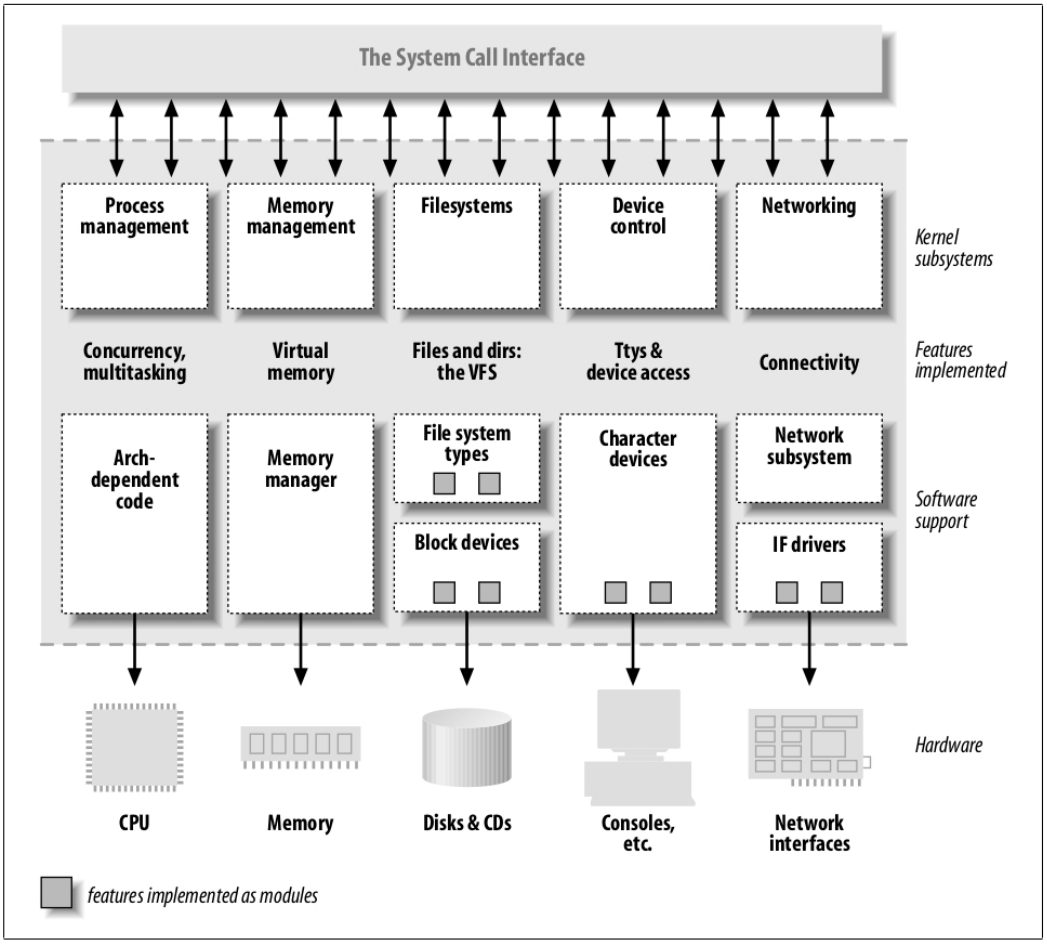
- **Filesystems** (συστήματα αρχείων)
  Το Linux βασίζεται πολύ στην έννοια του συστήματος αρχείων: μπορούμε να χειριστούμε σχεδόν τα πάντα σαν ένα αρχείο. Ο πυρήνας χτίζει ένα δομημένο σύστημα αρχείων πάνω από αδόμητο υλικό και το αποτέλεσμα, ένα μοντέλο αφαίρεσης με αρχεία (file abstraction), χρησιμοποιείται διαμέσου όλου του συστήματος. Επιπρόσθετα, το Linux υποστηρίζει διάφορους τύπους συστημάτων αρχείων, δηλαδή διαφορετικούς τρόπους οργάνωσης των δεδομένων στο φυσικό μέσο. Για παράδειγμα, οι σκληροί δίσκοι μπορούν να διαμορφωθούν με το πρότυπο σύστημα αρχείων ext4, το κοινώς χρησιμοποιούμενο FAT ή και αρκετά άλλα.

- **Device Control** (έλεγχος συσκευών)
  Όλες σχεδόν οι λειτουργίες συστήματος τελικά καταλήγουν σε μια φυσική συσκευή. Με εξαίρεση τον επεξεργαστή, τη μνήμη και λίγες άλλες οντότητες, όλος ο έλεγχος λειτουργιών μιας συσκευής επιτυγχάνεται με κώδικα συγκεκριμένο στην εκάστοτε συσκευή. Ο κώδικας αυτός ονομάζεται οδηγός συσκευής (device driver). Ο πυρήνας πρέπει να έχει ενσωματωμένο έναν οδηγό συσκευής για κάθε περιφερειακή συσκευή παρούσα στο σύστημα, από το σκληρό δίσκο μέχρι το πληκτρολόγιο.

- **Networking** (δικτύωση)
  Το λειτουργικό σύστημα πρέπει να διαχειρίζεται τη δικτύωση, γιατί οι περισσότερες λειτουργίες δικτύου δεν είναι συγκεκριμένες σε μια διεργασία: τα εισερχόμενα πακέτα είναι ασύγχρονα γεγονότα. Τα πακέτα πρέπει να συγκεντρωθούν, να αναγνωριστούν και να διεκπεραιωθούν, πριν τα αναλάβει μια διεργασία. Το σύστημα είναι υπεύθυνο να παραδίδει πακέτα δεδομένων μεταξύ εφαρμογών και διεπαφών δικτύου και να ελέγχει την εκτέλεση των εφαρμογών, ανάλογα με τη δικτυακή τους λειτουργία. Επιπρόσθετα, η δρομολόγηση των πακέτων και η μετάφραση των διευθύνσεων υλοποιούνται μέσα στον πυρήνα.

2.1.2 Ενότητες δυναμικής φόρτωσης (Modules)

Ένα από τα πολύ σημαντικά χαρακτηριστικά του Linux είναι η δυνατότητα που έχει ο πυρήνας να επεκτείνει το σύνολο των λειτουργιών που προσφέρει, κατά τη διάρκεια



Σχήμα 2.1: *Αρχιτεκτονική πυρήνα Linux*

που αυτός τρέχει. Αυτό σημαίνει ότι μπορούμε ανά πάσα στιγμή κατά τη διάρκεια εκτέλεσης του συστήματος να προσθέσουμε λειτουργικότητα στον πυρήνα (όπως και να αφαιρέσουμε). Κάθε κομμάτι κώδικα που μπορεί να προστεθεί στον πυρήνα τη στιγμή που αυτός τρέχει, ονομάζεται module. Ο πυρήνας του Linux υποστηρίζει αρκετούς διαφορετικούς τύπους (ή κλάσεις) τέτοιων ενοτήτων κώδικα (modules) και ένας από αυτούς είναι και οι οδηγοί συσκευών. Κάθε module απαρτίζεται από αντικειμενικό κώδικα (object code), που μπορεί να συνδεθεί στον πυρήνα και να αποσυνδεθεί από αυτόν δυναμικά, κατά το χρόνο εκτέλεσης. Η εισαγωγή και αφαίρεση γίνεται με χρήση των προγραμμάτων insmod και rmmod, αντίστοιχα. Το Σχήμα 2.1 περιγράφει διαφορετικούς τύπους από modules που είναι υπεύθυνα για συγκεκριμένες λειτουργίες: ο τύπος ενός module καθορίζεται από τη λειτουργικότητα που αυτό προσφέρει. Το Σχήμα 2.1 καλύπτει τους πιο βασικούς τύπους από modules, αλλά σε καμία περίπτωση δεν μπορεί να θεωρηθεί ολοκληρωμένο.

2.1.3 Κλάσεις συσκευών και modules

Από την οπτική γωνία του πυρήνα, οι συσκευές διακρίνονται σε τρείς τύπους. Κάθε module, συνήθως, υλοποιεί έναν από αυτούς τους τύπους κι έτσι μπορούν να χωριστούν σε char modules (για συσκευές χαρακτήρων), block modules (για συσκευές block) και σε network modules. Ο διαχωρισμός αυτός δεν είναι σαφής, αφού εξαρτάται από τον προγραμματιστή αν θα ενσωματώσει διαφορετικούς οδηγούς για διαφορετικές συσκευές σε ένα κοινό module. Προτιμότερο είναι, βέβαια, για λόγους επεκτασιμότητας και λειτουργικότητας, να δημιουργείται ένα καινούριο module για κάθε λειτουργία που προστίθεται στον πυρήνα.

Οι τρεις τύποι είναι:

- *Character Devices*
  Μια συσκευή χαρακτήρων μπορεί να προσπελαστεί σαν μια ροή από bytes (όπως ένα αρχείο). Ο οδηγός συσκευής χαρακτήρων υλοποιεί αυτή τη συμπεριφορά. Ένας τέτοιος οδηγός υλοποιεί συνήθως τις κλήσεις συστήματος open, close, read και write. Οι συσκευές χαρακτήρων είναι προσβάσιμες μέσω ειδικών αρχείων στο σύστημα αρχείων που ονομάζονται κόμβοι (για παράδειγμα /dev/tty1, /dev/lp0). Η σημαντικότερη διαφορά τους με ένα απλό αρχείο είναι ότι, ενώ στο αρχείο μπορούν να προσπελαστούν δεδομένα σε οποιαδήποτε θέση, στη συσκευή χαρακτήρων η πρόσβαση είναι ακολουθιακή. Υπάρχουν, ωστόσο, συσκευές (όπως τα frame grabbers) που μπορούν να προσπελαστούν συνολικά με χρήση των κλήσεων συστήματος mmap ή lseek.

- *Block Devices*
  Οι συσκευές block είναι προσβάσιμες μέσω αρχείων - κόμβων του συστήματος αρχείων στον κατάλογο /dev. Μια τέτοια συσκευή μπορεί να ενθυλακώσει ένα σύστημα αρχείων. Στα περισσότερα συστήματα Unix μια συσκευή block μπορεί να χειριστεί λειτουργίες εισόδου / εξόδου που μεταφέρουν ένα ή περισσότερα blocks, τα οποία συνήθως έχουν μέγεθος 512 bytes (ή κάποια μεγαλύτερη δύναμη του 2). Αντίθετα, στο Linux οι εφαρμογές μπορούν να διαβάσουν και να γράψουν σε μια συσκευή block όπως και σε μία συσκευή χαρακτήρων: επιτρέπεται η μεταφορά οποιουδήποτε αριθμού bytes. Αυτό έχει ως αποτέλεσμα οι συσκευές χαρακτήρων και οι συσκευές block να διαφέρουν μόνο στον τρόπο με τον

οποίο τις χειρίζεται εσωτερικά ο πυρήνας και η διαφορά αυτή είναι διαφανής στο χρήστη.

- *Network Interfaces*
  Οποιαδήποτε δικτυακή κίνηση γίνεται μέσω μιας διεπαφής, δηλαδή μιας συσκευής ικανής να ανταλλάξει δεδομένα με άλλα συστήματα. Συνήθως, μια διεπαφή είναι μία φυσική συσκευή, αλλά θα μπορούσε να είναι και μια εικονική συσκευή, όπως το loopback interface, που χρησιμοποιείται για την επικοινωνία ενός συστήματος με τον εαυτό του. Μια δικτυακή επαφή είναι υπεύθυνη για την αποστολή και λήψη πακέτων δεδομένων, χωρίς να γνωρίζει πώς οι κινήσεις αντιστοιχούνται στα μεταφερόμενα πακέτα, γιατί οδηγείται από το υποσύστημα δικτύου του πυρήνα. Πολλές συνδέσεις δικτύου είναι προσανατολισμένες σε ροή δεδομένων αλλά οι συσκευές δικτύου, συνήθως, είναι σχεδιασμένες με βάση τη μετάδοση πακέτων. Ένας οδηγός συσκευής δικτύου δε γνωρίζει τίποτα για τις διακριτές συνδέσεις, απλά χειρίζεται πακέτα.

  Αφού μια συσκευή δικτύου δεν είναι προσανατολισμένη σε ροή δεδομένων, η διεπαφή δικτύου δεν αντιστοιχίζεται σε έναν κόμβο στο σύστημα αρχείων, όπως μια συσκευή χαρακτήρων ή block. Τα συστήματα Unix χειρίζονται μια συσκευή δικτύου δίνοντάς της ένα μοναδικό αναγνωριστικό (όπως το eth0), που δεν έχει κάποια αντιστοίχιση στο σύστημα αρχείων. Ο τρόπος προσπέλασης αυτής της διεπαφής δεν είναι με τις κλήσεις read και write αλλά με κλήσεις σε συναρτήσεις που σχετίζονται με το χειρισμό και τη μετάδοση πακέτων.

Υπάρχουν και άλλοι τρόποι διαχωρισμού modules οδηγών συσκευών που είναι ορθογώνιοι (orthogonal) με τους παραπάνω τύπους. Γενικά, κάποιοι τύποι συσκευών λειτουργούν με επιπρόσθετα επίπεδα συναρτήσεων του πυρήνα για ένα δεδομένο τύπο συσκευών. Για παράδειγμα, υπάρχουν USB modules, serial modules, SCSI modules κ.τ.λ. Κάθε συσκευή USB οδηγείται από ένα USB module, που λειτουργεί με το υποσύστημα USB, αλλά η συσκευή εμφανίζεται στο σύστημα σαν μια συσκευή χαρακτήρων (όπως μια σειριακή πόρτα USB), μια συσκευή block (μια USB συσκευή που διαβάζει κάρτες μνήμης) ή μια συσκευή δικτύου (μια USB διεπαφή δικτύου Ethernet).

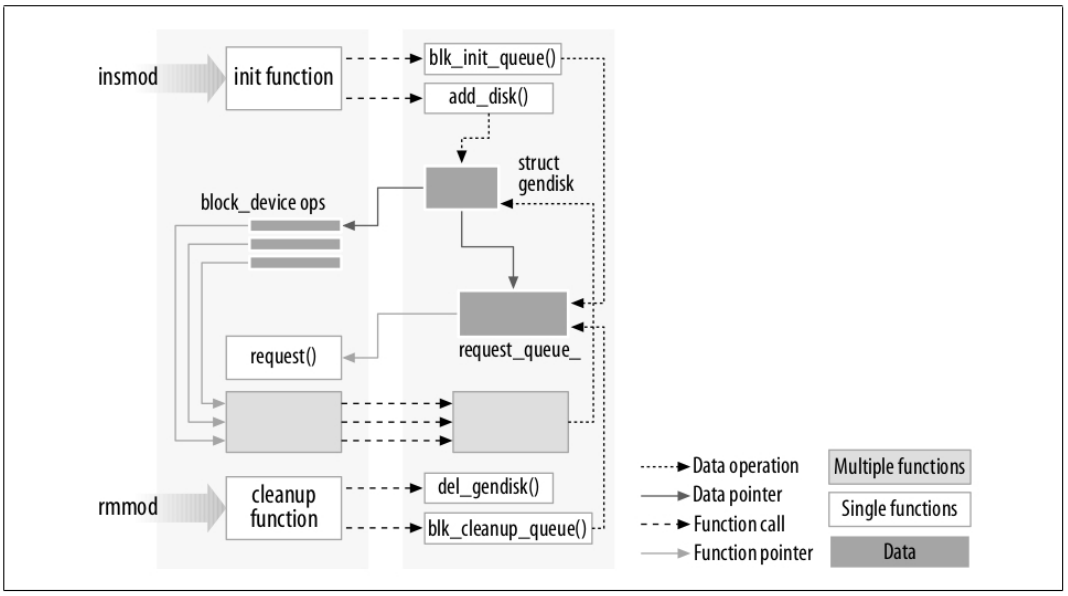
Παράλληλα με τους οδηγούς συσκευών, υπάρχουν και άλλες λειτουργίες, τόσο υλικού, όσο και λογισμικού, που έχουν τη μορφή ενοτήτων δυναμικής φόρτωσης στον πυρήνα. Ένα συνηθισμένο παράδειγμα είναι τα συστήματα αρχείων. Ο τύπος του συστήματος

αρχείων έχει να κάνει με τον τρόπο που οργανώνεται η πληροφορία σε μία συσκευή block, ώστε να παριστάνει ένα δένδρο από φακέλους και αρχεία. Αυτή η οντότητα δεν είναι οδηγός συσκευής, αλλά οδηγός λογισμικού, γιατί αντιστοιχεί τις χαμηλού επιπέδου δομές δεδομένων σε υψηλού επιπέδου δομές.

2.1.4 Διαφορές εφαρμογών και Kernel Modules

Πριν προχωρήσουμε παρακάτω αξίζει να αναφέρουμε μερικές διαφορές μεταξύ ενός kernel module και μιας εφαρμογής.

Ενώ οι περισσότερες μικρές και μεσαίες εφαρμογές εκτελούν μία μόνο εργασία από την αρχή έως το τέλος, κάθε kernel module απλά δηλώνεται στον πυρήνα, με σκοπό να εξυπηρετήσει μελλοντικά αιτήματα, και η συνάρτηση αρχικοποίησης του τερματίζει άμεσα μετά τη φόρτωσή του. Ο ρόλος, δηλαδή, της συνάρτησης αρχικοποίησης, είναι να προετοιμάσει τη μετέπειτα κλήση του συνόλου των συναρτήσεων που υλοποιούνται στο module. Η συνάρτηση εξόδου καλείται ακριβώς πριν αφαιρεθεί το module. Αυτός ο τρόπος προγραμματισμού είναι αντίστοιχος με τον προγραμματισμό υποκινούμενο από γεγονότα (event-driven programming), που δε συναντάται μεν σε όλες τις εφαρμογές, αλλά συναντάται σε όλα τα kernel modules. Μία άλλη μεγάλη διαφορά, μεταξύ των event-driven εφαρμογών και του κώδικα του πυρήνα, βρίσκεται στη συνάρτηση εξόδου. Ενώ μια εφαρμογή που τερματίζει μπορεί να μην ασχολείται με την αποδέσμευση των πόρων που έχει προηγουμένως δεσμεύσει, η συνάρτηση εξόδου ενός module πρέπει να αναιρέσει όλα όσα δημιούργησε η συνάρτηση αρχικοποίησης και ίσως και άλλες συναρτήσεις του σώματος του module. Σε αντίθετη περίπτωση, τα υπολείμματα παραμένουν παρόντα μέχρι την επόμενη επανεκκίνηση του συστήματος. Επίσης, οι εφαρμογές μπορούν να καλέσουν συναρτήσεις, τις οποίες δεν έχουν ορίσει οι ίδιες. Αυτό συμβαίνει γιατί μπορούν να γίνουν συνδέσεις με βιβλιοθήκες κώδικα, όπως για παράδειγμα η libc. Τα modules, αντίθετα, είναι συνδεδεμένα μόνο με τον πυρήνα και έτσι μπορούν να καλέσουν μόνο συναρτήσεις που αυτός εξάγει. Ακόμη, μεταξύ προγραμματισμού στον πυρήνα και προγραμματισμού εφαρμογών σημειώνεται διαφοροποίηση σχετικά με τον τρόπο που το κάθε περιβάλλον χειρίζεται τα λάθη. Ένα λάθος κατάτμησης (segmentation fault) κατά τη διάρκεια ανάπτυξης μιας εφαρμογής είναι ακίνδυνο, όχι όμως και ένα λάθος πυρήνα, που στην καλύτερη περίπτωση σκοτώνει την τρέχουσα διεργασία, αν όχι όλο το σύστημα.



Σχήμα 2.2: *Σύνδεση module στον πυρήνα του Linux*

Τέλος, το Σχήμα 2.2 δείχνει τη διαδικασία σύνδεσης ενός module στον πυρήνα.

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

Ένα module τρέχει στο "χώρο πυρήνα", ενώ οι εφαρμογές τρέχουν στο "χώρο χρήστη". Αυτή η διαφοροποίηση βρίσκεται στη βάση της θεωρίας των λειτουργικών συστημάτων. Ο ρόλος του λειτουργικού συστήματος είναι να παρέχει στις εφαρμογές μια συνεπή όψη του υλικού του υπολογιστικού συστήματος. Επιπροσθέτως, το λειτουργικό σύστημα πρέπει να εξασφαλίζει την ανεξάρτητη λειτουργία των προγραμμάτων και την προστασία από αναρμόδια πρόσβαση στους εκάστοτε πόρους. Αυτό το σημαντικό έργο είναι δυνατό να επιτελεστεί μόνο αν η CPU είναι ικανή να διαχωρίσει και να προστατέψει το λογισμικό συστήματος από τις εφαρμογές. Όλοι οι σύγχρονοι επεξεργαστές έχουν τη δυνατότητα να επιβάλλουν αυτή τη συμπεριφορά με τη μορφή διαφορετικών επιπέδων (levels) λειτουργίας. Κάθε επίπεδο λειτουργίας της CPU έχει διαφορετικό ρόλο και ορισμένες λειτουργίες δεν μπορούν να εκτελεστούν στα χαμηλότερα επίπεδα. Η εναλλαγή από το ένα επίπεδο στο άλλο γίνεται μέσω ενός περιορισμένου αριθμού πυλών. Τα συστήματα Unix έχουν σχεδιαστεί για να εκμεταλλεύονται αυτό το χαρακτηριστικό του υλικού, χρησιμοποιώντας δύο τέτοια επίπεδα. Ο πυρήνας εκτελείται στο υψηλότερο επίπεδο (supervisor mode), όπου τα πάντα επιτρέπονται, σε αντίθεση με τις εφαρμογές που τρέχουν στο χαμηλότερο επίπεδο (user mode), όπου ο επεξεργα-

στής ρυθμίζει την πρόσβαση στο υλικό και την αναρμόδια πρόσβαση στη μνήμη.

Συνήθως, αναφερόμαστε στους δύο αυτούς τρόπους εκτέλεσης σαν χώρο πυρήνα (kernel space) και χώρο χρήστη (user space). Οι όροι αυτοί δεν περικλείουν μόνο τα διαφορετικά επίπεδα δικαιωμάτων που είναι σύμφυτα με τους δύο τρόπους εκτέλεσης, αλλά και το γεγονός ότι κάθε τρόπος εκτέλεσης μπορεί να έχει το δικό του χώρο διευθύνσεων. Το Unix μεταφέρει την εκτέλεση από το χώρο χρήστη στο χώρο πυρήνα κάθε φορά που μία εφαρμογή εκτελεί μία κλήση συστήματος ή όταν λαμβάνει χώρα μία διακοπή υλικού (hardware interrupt). Ο κώδικας πυρήνα που εκτελεί μία κλήση συστήματος τρέχει στο πλαίσιο μιας διεργασίας. Εκτελείται εκ μέρους της διεργασίας που έκανε την κλήση και έχει τη δυνατότητα πρόσβασης στο δικό της χώρο μνήμης. Από την άλλη πλευρά, ο κώδικας που χειρίζεται διακοπές είναι ασύγχρονος, αντιμετωπίζει ισότιμα όλες τις διεργασίες και δε σχετίζεται με καμία συγκεκριμένη από αυτές.

Ο ρόλος του module, όπως προαναφέραμε, είναι να επεκτείνει τη λειτουργικότητα του πυρήνα. Έτσι, ο κώδικας των modules, όπως φαίνεται και από την προηγούμενη ανάλυση, τρέχει στο χώρο πυρήνα. Συνήθως, ένας οδηγός συσκευής εκτελεί και τις δύο εργασίες που περιγράφηκαν προηγουμένως: μερικές συναρτήσεις του module εκτελούνται ως μέρος κλήσεων συστήματος και κάποιες άλλες είναι υπεύθυνες για το χειρισμό διακοπών.

2.1.6 Ιδιαιτερότητες προγραμματισμού στον πυρήνα του Linux

Ο προγραμματισμός στο χώρο πυρήνα (kernel space) διαφέρει σε αρκετά σημεία από τον προγραμματισμό στο χώρο χρήστη (user space).

Μια σημαντική διαφορά είναι το θέμα του ταυτοχρονισμού. Οι περισσότερες εφαρμογές, με εξαίρεση τις πολυνηματικές, συνήθως εκτελούνται σειριακά, από την αρχή μέχρι το τέλος, χωρίς να χρειάζεται να ανησυχούν για ταυτόχρονες αλλαγές στο περιβάλλον τους. Ο κώδικας πυρήνα από την άλλη πλευρά, όσο απλός και να είναι, πρέπει πάντοτε να γράφεται λαμβάνοντας υπόψιν τα θέματα ταυτοχρονισμού. Υπάρχουν διάφορες πηγές ταυτοχρονισμού στον πυρήνα. Ένα σύστημα Linux τρέχει πολλαπλές διεργασίες και πολλές από αυτές μπορεί να καλούν τις ίδιες μεθόδους ενός module. Οι χειριστές διακοπών (interrupt handlers) τρέχουν ασύγχρονα και πιθανώς παράλληλα με τον κώδικα ενός οδηγού συσκευής. Επιπλέον, υπάρχουν κι άλλες δομές του πυρήνα που εκτελού-

νται ασύγχρονα, όπως για παράδειγμα kernel timers, tasklets, workqueues. Ακόμη, το Linux μπορεί να τρέχει σε ένα πολυεπεξεργαστικό σύστημα (SMP) με αποτέλεσμα ο ίδιος κώδικας να εκτελείται παράλληλα σε πολλούς επεξεργαστές. Τέλος, ο κώδικας του πυρήνα μπορεί να διακοπεί ανά πάσα στιγμή (preemptive kernel), που σημαίνει ότι ακόμα και τα μονοεπεξεργαστικά συστήματα αντιμετωπίζουν πολλά από τα προβλήματα ταυτοχρονισμού που εμφανίζονται στα πολυεπεξεργαστικά συστήματα.

Κάθε εφαρμογή διαθέτει ένα αρκετά μεγάλο μέρος του χώρου εικονικών διευθύνσεων της για τη στοίβα (stack area). Η στοίβα χρησιμοποιείται για την αποθήκευση του ιστορικού κλήσεων συναρτήσεων (function call history) και όλων των αυτόματων μεταβλητών (automatic variables) που δημιουργούνται από τις ενεργές συναρτήσεις. Ο πυρήνας, αντίθετα, διαθέτει στοίβα μικρού μεγέθους, την οποία ο κώδικας ενός οποιουδήποτε module πρέπει να τη μοιραστεί με όλο το kernel space.

Μια ακόμη διαφοροποίηση ανάμεσα στο χώρο χρήστη και στο χώρο πυρήνα είναι ότι ο τελευταίος δε μπορεί να εκτελέσει αριθμητική κινητής υποδιαστολής. Αυτό θα απαιτούσε να σώζεται η κατάσταση (state) του επεξεργαστή κινητής υποδιαστολής κατά την είσοδο και έξοδο από το kernel space, τουλάχιστον σε ορισμένες αρχιτεκτονικές. Δεδομένου ότι ουσιαστικά δεν υπάρχει ανάγκη για αριθμητική κινητής υποδιαστολής στον πυρήνα και δεδομένου του κόστους για την υποστήριξή της, έχει γίνει η επιλογή να μην υποστηρίζεται στο kernel space.

2.1.7 Οι οδηγοί συσκευών χαρακτήρων

Οι μηχανισμοί διαδιεργασιακής επικοινωνίας που αναπτύσσονται στην παρούσα διπλωματική εργασία υλοποιούνται ως οδηγοί συσκευών χαρακτήρων (char drivers). Για το λόγο αυτό θεωρούμε χρήσιμο να παραθέσουμε ορισμένες λεπτομέρειες που αφορούν τη σχεδίαση και υλοποίηση ενός οποιουδήποτε τέτοιου οδηγού.

Ο χειρισμός των συσκευών χαρακτήρων (character devices) γίνεται μέσω ονομάτων στο σύστημα αρχείων. Αυτά τα ονόματα καλούνται "ειδικά αρχεία" ή "αρχεία συσκευών" και βρίσκονται συνήθως στον κατάλογο /dev.

Οι οδηγοί χαρακτήρων χαρακτηρίζονται από το μείζονα και τον ελάσσονα αριθμό (major, minor number). Οι καινούργιες εκδόσεις του πυρήνα επιτρέπουν σε περισσότερες από μια συσκευές να μοιράζονται τον ίδιο μείζονα αριθμό, αλλά η συνηθισμένη οργάνωση

είναι κάθε οδηγός να έχει τον δικό του. Ο ελάσσων αριθμός χρησιμοποιείται από τον πυρήνα για να ξεχωρίζει σε ποια συσκευή γίνεται η αναφορά. Συνηθισμένο είναι, επίσης, ένας οδηγός να κρατά ένα δείκτη στη συσκευή χαρακτήρων που δημιουργεί ή ένα πίνακα συσκευών, όπου ο ελάσσων αριθμός αντιστοιχεί στη θέση της συσκευής μέσα στον πίνακα.

Μέσα στον πυρήνα ορίζεται ο τύπος *dev_t* που μπορεί να κρατά το μείζονα και τον ελάσσονα αριθμό. Επίσης, από τον πυρήνα γίνονται διαθέσιμες και οι συναρτήσεις χειρισμού μιας μεταβλητής αυτού του τύπου. Έτσι, από μια μεταβλητή τύπου *dev_t* μπορούμε να πάρουμε το μείζονα και τον ελάσσονα αριθμό με τις μακροεντολές:

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

Αντίστοιχα, από τους δύο αυτούς αριθμούς μπορούμε να πάρουμε και την τιμή της μεταβλητής τύπου *dev_t*:

MKDEV(int major, int minor);

Για να μπορεί να χρησιμοποιηθεί μια συσκευή χαρακτήρων πρέπει αρχικά να γίνει μια κλήση στη συνάρτηση *register_chrdev_region()*, που δηλώνεται στο αρχείο κεφαλίδα *<linux/fs.h>*:

*int register_chrdev_region(dev_t first, unsigned int count, char *name);*

Η παράμετρος *first* είναι το αρχικό ζεύγος αριθμών από το οποίο θα αρχίσει η δέσμευση των συσκευών χαρακτήρων. Συνήθως ο ελάσσων αριθμός είναι 0, αλλά δεν υπάρχει απαίτηση για κάτι τέτοιο. Η παράμετρος *count* είναι ο συνολικός αριθμός των συσκευών που θα χρησιμοποιηθούν. Αν το μέγεθος αυτό είναι μεγάλο, τότε ενδέχεται οι συσκευές που θα δεσμευτούν να εκτείνονται σε συνεχόμενους μείζονες αριθμούς. Τέλος, η παράμετρος *name* είναι το όνομα της συσκευής που σχετίζεται με τους αριθμούς αυτούς.

Εναλλακτικά, μπορεί να χρησιμοποιηθεί η *alloc_chrdev_region()*. Με τη συνάρτηση αυτή οι χαρακτηριστικοί αριθμοί μιας συσκευής χαρακτήρων αποδίδονται από τον πυρήνα. Έτσι, εξασφαλίζεται μεγαλύτερη φορητότητα στους οδηγούς, αφού δίνεται η δυνατότητα στον πυρήνα να διαθέσει ελεύθερους αριθμούς.

*int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);*

Στη συνάρτηση αυτή η παράμετρος *dev* θα πάρει την τιμή που θα αποδώσει ο πυρήνας έπειτα από μια επιτυχή δέσμευση. Η παράμετρος *firstminor* πρέπει να είναι ο πρώτος ελάσσων αριθμός που ζητείται και συνήθως έχει την τιμή 0. Τέλος, οι παράμετροι *count* και *name* έχουν την ίδια χρήση όπως και στη *register_chrdev_region()*.

Ανεξάρτητα από τον τρόπο δέσμευσης, η απελευθέρωση των χαρακτηριστικών αριθμών της συσκευής γίνεται με την κλήση της συνάρτησης:

void unregister_chrdev_region(dev_t first, unsigned int count);

Οι παραπάνω συναρτήσεις δεσμεύουν αριθμούς για τη χρήση σε έναν οδηγό, αλλά δεν πληροφορούν τον πυρήνα ποιες θα είναι οι λειτουργίες που θα υποστηρίζονται από αυτούς. Προτού μια εφαρμογή στο χώρο χρήστη μπορεί να έχει πρόσβαση σε αυτούς, ο οδηγός θα πρέπει να τους συνδέσει με κάποιες εσωτερικές συναρτήσεις που υλοποιούν τη λειτουργικότητα μιας συσκευής χαρακτήρων.

Οι πιο σημαντικές λειτουργίες σε μια συσκευή χαρακτήρων περιγράφονται από δομές του πυρήνα, κυρίαρχα τις *stuct file_operations*, *stuct file* και *stuct inode*.

Αν και οι οδηγοί χαρακτήρων είναι αρκετά απλοί, η πλήρης περιγραφή τους εκτείνεται πέρα από τα όρια του σκοπού αυτής της εργασίας. Για το λόγο αυτό, στη συνέχεια, θα αναφερθούν μόνο εκείνα τα στοιχεία τους που χρησιμοποιήθηκαν στην παρούσα εργασία.

Η δομή *stuct file_operations*

Με τη δομή *stuct file_operations* μια συσκευή χαρακτήρων δημιουργεί τη σύνδεση μεταξύ των αριθμών και των υποστηριζόμενων λειτουργιών της συσκευής. Η δομή αυτή ορίζεται στο αρχείο κεφαλίδα *<linux/fs.h>* και είναι μια συλλογή από δείκτες σε συναρτήσεις. Κάθε ανοιχτό αρχείο (που αναπαρίσταται εσωτερικά με μια δομή τύπου *struct file*) σχετίζεται με το δικό του σύνολο συναρτήσεων. Οι λειτουργίες αυτές είναι υπεύθυνες κατά κύριο λόγο για την υλοποίηση κλήσεων συστήματος και γι αυτό ονομάζονται open, close, κ.τ.λ. Το αρχείο αυτό μπορεί να θεωρηθεί ως ένα "αντικείμενο" και οι συναρτήσεις που σχετίζονται με αυτό ως οι "μέθοδοί" του, στην ορολογία του αντικειμενοστραφούς προγραμματισμού.

Παραδοσιακά μια μεταβλητή τύπου *struct file_operations* ή ένας δείκτης σε μια τέτοια μεταβλητή ονομάζεται *fops*. Κάθε πεδίο της υλοποιεί μια συγκεκριμένη λειτουργία ή

μπορεί να έχει τιμή NULL, για λειτουργίες που δεν υποστηρίζονται. Στην ακόλουθη λίστα περιγράφουμε τα πεδία που μας ενδιαφέρουν στην παρούσα εργασία:

- *struct module *owner*
  Το πρώτο πεδίο της δομής δεν είναι μια λειτουργία, αλλά ένας δείκτης στο module που του ανήκει η συσκευή χαρακτήρων. Η πιο συνηθισμένη τιμή που ανατίθεται στο πεδίο αυτό είναι η μακροεντολή *THIS_MODULE* που ορίζεται στο αρχείο κεφαλίδας <*linux/module.h*>.

- *ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)*
  Η συνάρτηση αυτή χρησιμοποιείται για την ανάκτηση δεδομένων από τη συσκευή. Μια τιμή NULL στο πεδίο αυτό θα προκαλέσει την επιστροφή της τιμής -EINVAL ("Invalid Argument") σε μια εφαρμογή που επιχειρεί να καλέσει την αντίστοιχη κλήση συστήματος read.

- *ssize_t (*read_iter) (struct kiocb *, struct iov_iter *)*
  Αντίστοιχη με την read με τη διαφορά ότι χρησιμοποιείται για την υλοποίηση της κλήσης συστήματος readv, η οποία υλοποιεί λειτουργικότητα scatter input.

- *ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)*
  Η συνάρτηση αυτή χρησιμοποιείται για την αποστολή δεδομένων προς τη συσκευή. Μια τιμή NULL στο πεδίο αυτό θα προκαλέσει αντίστοιχα αποτελέσματα όπως και στη read.

- *ssize_t (*write_iter) (struct kiocb *, struct iov_iter *)*
  Αντίστοιχη με την write με τη διαφορά ότι χρησιμοποιείται για την υλοποίηση της κλήσης συστήματος writev, η οποία υλοποιεί λειτουργικότητα gather output.

- *long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long)*
  Η κλήση συστήματος ioctl παρέχει έναν τρόπο να δημιουργούνται με τον έλεγχο του προγραμματιστή εντολές κατάλληλες για κάθε συσκευή. Με την ioctl, δηλαδή, υλοποιούνται σε μια συσκευή όλες εκείνες οι λειτουργίες που δεν είναι δυνατό να υλοποιηθούν με άλλες κλήσεις συστήματος. Υπάρχουν μερικές ioctl εντολές που αναγνωρίζονται από τον πυρήνα ακόμα και όταν η ioctl δεν ορίζεται στη δομή fops. Αν μια συσκευή δεν παρέχει μια κλήση συστήματος ioctl, τότε η τιμή που επιστρέφεται σε μια απόπειρα χρησιμοποίησης της είναι η -ENOTTY ("No such ioctl for device")

- *int (*open) (struct inode *, struct file *)*
  Η αντίστοιχη κλήση συστήματος που υλοποιείται από τη συνάρτηση αυτή είναι πάντα η πρώτη που χρησιμοποιείται στη συσκευή. Αν η τιμή της είναι NULL, τότε κάθε απόπειρα κλήσης της επιτυγχάνει, χωρίς όμως να εκτελείται κάποια ενέργεια από τον οδηγό.

- *int (*release) (struct inode *, struct file *)*
  Αυτή η συνάρτηση υλοποιεί τη λειτουργία που εκτελείται κατά την απελευθέρωση της δομής *struct file*. Όπως και η open έτσι και η release μπορεί να είναι κενή.

Για λόγους πληρότητας αναφέρονται όλα τα πεδία της δομής *stuct file_operations* χωρίς, όμως, να περιγράφονται:

- *struct module *owner*
- *loff_t (*llseek) (struct file *, loff_t, int)*
- *ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)*
- *ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)*
- *ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t)*
- *ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t)*
- *ssize_t (*read_iter) (struct kiocb *, struct iov_iter *)*
- *ssize_t (*write_iter) (struct kiocb *, struct iov_iter *)*
- *int (*iterate) (struct file *, struct dir_context *)*
- *unsigned int (*poll) (struct file *, struct poll_table_struct *)*
- *long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long)*
- *long (*compat_ioctl) (struct file *, unsigned int, unsigned long)*
- *int (*mmap) (struct file *, struct vm_area_struct *)*
- *int (*open) (struct inode *, struct file *)*

- *int (*flush) (struct file *, fl_owner_t id)*
- *int (*release) (struct inode *, struct file *)*
- *int (*fsync) (struct file *, loff_t, loff_t, int datasync)*
- *int (*aio_fsync) (struct kiocb *, int datasync)*
- *int (*fasync) (int, struct file *, int)*
- *int (*lock) (struct file *, int, struct file_lock *)*
- *ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int)*
- *unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long)*
- *int (*check_flags)(int)*
- *int (*flock) (struct file *, int, struct file_lock *)*
- *ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int)*
- *ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int)*
- *int (*setlease)(struct file *, long, struct file_lock **)*
- *long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len)*
- *int (*show_fdinfo)(struct seq_file *m, struct file *f)*

Η δομή *struct file*

Η δομή *struct file* ορίζεται στο αρχείο κεφαλίδα <*linux/fs.h*> και είναι η δεύτερη πιο σημαντική δομή σε μια συσκευή χαρακτήρων. Με τη δομή αυτή αναπαρίσταται ένα ανοιχτό αρχείο στο Linux και δεν αποτελεί ειδική δομή των συσκευών χαρακτήρων. Δημιουργείται από τον πυρήνα σε μία κλήση της open και περνά ως όρισμα σε κάθε συνάρτηση που επιτελεί κάποια ενέργεια στο αρχείο, μέχρι την κλήση της close. Όταν

όλα τα στιγμιότυπα του αρχείου κλείσουν, τότε ο πυρήνας ελευθερώνει το χώρο της δομής *struct file*.

Στη συνέχεια περιγράφονται τα πιο σημαντικά πεδία της δομής αυτής:

- *mode_t f_mode*
  Το πεδίο αυτό καθορίζει το αρχείο σαν αναγνώσιμο ή εγγράψιμο ή και τα δύο με βάση τα bits FMODE_READ και FMODE_WRITE. Αυτό το πεδίο μπορεί να ελεγχθεί για την υλοποίηση ελεγχομένης πρόσβασης σε μια κλήση open ή ioctl. Αυτόματα ελέγχεται σε μία κλήση της read ή της write.

- *loff_t f_pos*
  Κρατά την τρέχουσα θέση εγγραφής ή ανάγνωσης στο αρχείο. Ο τύπος *loff_t* έχει μέγεθος 64-bit σε όλες τις αρχιτεκτονικές. Συνήθως, η τιμή του πεδίου προορίζεται για ανάγνωση και ενημερώνεται αυτόματα με μια κλήση της read ή της write.

- *unsigned int f_flags*
  Αυτές είναι σημαίες με τιμές όπως *O_RDONLY*, *O_NONBLOCK* και *O_SYNC*. Ένας οδηγός πρέπει να ελέγξει τη σημαία *O_NONBLOCK* για να δει αν έχει ζητηθεί κάποια nonblocking λειτουργία Ε/Ε. Οι άλλες σημαίες χρησιμοποιούνται πολύ σπάνια.

- *struct file_operations *f_op*
  Το πεδίο αυτό είναι ένας δείκτης στις λειτουργίες που σχετίζονται με το αρχείο. Ο πυρήνας δίνει τιμή σε αυτό το δείκτη με την κλήση της open και διαβάζει από αυτόν το κατάλληλο πεδίο για κάθε κλήση συστήματος στο αρχείο. Αυτό δίνει στον οδηγό τη δυνατότητα να αλλάζει την υλοποίηση των λειτουργιών κατά το χρόνο εκτέλεσης, κάτι που είναι αντίστοιχο με την υπερφόρτωση συναρτήσεων (method overriding) του αντικειμενοστραφούς προγραμματισμού.

- *void *private_data*
  Το πεδίο αυτό παίρνει τιμή NULL πριν την κλήση της μεθόδου open του οδηγού. Ο προγραμματιστής μπορεί να το χρησιμοποιήσει για τους δικούς του σκοπούς ή να το αγνοήσει. Μπορεί, για παράδειγμα, να δεσμεύσει μνήμη και να δώσει τη διεύθυνση αυτής ως τιμή του δείκτη, φροντίζοντας, όμως, να απελευθερώσει αυτή τη μνήμη κατά την κλήση της μεθόδου release.

- *struct dentry *f_dentry*
  Μέσω της δομής αυτής μπορούμε να προσπελάσουμε τη δομή inode που σχετίζεται με το αντίστοιχο ανοικτό αρχείο.

Η δομή *struct inode*

Η δομή *struct inode* χρησιμοποιείται εσωτερικά από τον πυρήνα για την αναπαράσταση αρχείων. Είναι διαφορετική από τη δομή *struct file*, η οποία χρησιμοποιείται για την αναπαράσταση των ***ανοικτών*** αρχείων. Είναι δυνατόν να υπάρχουν πολλαπλές δομές *struct file* που αναπαριστούν διάφορους ανοικτούς περιγραφείς αρχείου (open file descriptors) σε ένα αρχείο, αλλά όλες αναφέρονται στην ίδια δομή *struct inode*. Η δομή αυτή περιέχει πολλές πληροφορίες που αφορούν ένα αρχείο αλλά γενικά δύο μόνο πεδία της έχουν ενδιαφέρον για τους οδηγούς συσκευών χαρακτήρων:

- *dev_t i_rdev*
  Για τα inodes που αναπαριστούν αρχεία συσκευών αυτό το πεδίο περιέχει τον αριθμό της συσκευής (major και minor number).

- *struct cdev *i_cdev*
  Η δομή *struct cdev* χρησιμοποιείται εσωτερικά από τον πυρήνα για την αναπαράσταση συσκευών χαρακτήρων. Το πεδίο *i_cdev* αποθηκεύει ένα δείκτη σε αυτή τη δομή όταν το inode αναπαριστά ένα αρχείο συσκευής.

2.2 Το υποσύστημα διαχείρισης μνήμης του Linux

Στην ενότητα αυτή δίνουμε μια εκτενή περιγραφή των δομών δεδομένων που χρησιμοποιεί ο πυρήνας για να διαχειριστεί τη μνήμη του συστήματος [Gor04, CRKH05, BC05].

2.2.1 Τύποι διευθύνσεων

Το Linux είναι ένα λειτουργικό σύστημα που χρησιμοποιεί εικονική μνήμη (virtual memory), δηλαδή οι διευθύνσεις που χειρίζονται οι εφαρμογές δεν έχουν άμεση αντιστοιχία με τις φυσικές διευθύνσεις που χρησιμοποιεί το υλικό. Η εικονική μνήμη εισάγει ένα

επίπεδο αφαίρεσης, το οποίο επιτρέπει την υλοποίηση διάφορων χρήσιμων λειτουργιών. Με την εικονική μνήμη, τα προγράμματα που τρέχουν σε ένα σύστημα μπορούν να δεσμεύσουν περισσότερη μνήμη από όση είναι φυσικά διαθέσιμη στο υλικό. Ακόμα και μία μεμονωμένη εφαρμογή μπορεί να έχει ένα χώρο εικονικών διευθύνσεων μεγαλύτερο από τη φυσική μνήμη του συστήματος. Επιπλέον, με την εικονική μνήμη είναι εφικτό να απεικονίσουμε μέρος της μνήμης μιας διεργασίας σε μνήμη μιας συσκευής, επιτρέποντας έτσι την πρόσβαση στη μνήμη αυτής της συσκευής.

Το Linux χρησιμοποιεί διάφορους τύπους διευθύνσεων, καθένας με τη δική του σημασιολογία. Δυστυχώς, ο κώδικας του πυρήνα δεν είναι πάντα ξεκάθαρος ποιος τύπος διεύθυνσης χρησιμοποιείται σε κάθε περίπτωση, γι αυτό απαιτείται προσοχή από τον προγραμματιστή.

Στα επόμενα αναφέρονται οι τύποι διευθύνσεων που χρησιμοποιούνται στον πυρήνα του Linux. Το Σχήμα 2.3 δείχνει πώς αυτές οι διευθύνσεις σχετίζονται με τη φυσική μνήμη του συστήματος.

- *Εικονικές διευθύνσεις χρήστη (User virtual addresses)*
  Αυτές είναι οι διευθύνσεις που χρησιμοποιούν οι εφαρμογές στο χώρο χρήστη. Οι διευθύνσεις αυτές έχουν μέγεθος 32 ή 64 bits, ανάλογα με την υποκείμενη αρχιτεκτονική του υλικού και κάθε εφαρμογή έχει το δικό της χώρο εικονικών διευθύνσεων.

- *Φυσικές διευθύνσεις (Physical addresses)*
  Αυτές είναι οι διευθύνσεις που χρησιμοποιούνται μεταξύ επεξεργαστή και φυσικής μνήμης. Έχουν μέγεθος 32 ή 64 bits. Ακόμα και συστήματα 32-bit μπορούν, σε ορισμένες περιπτώσεις, να χρησιμοποιήσουν φυσικές διευθύνσεις μεγαλύτερες των 32-bit (Physical Address Extension)

- *Διευθύνσεις διαύλου (Bus addresses)*
  Οι διευθύνσεις αυτές χρησιμοποιούνται μεταξύ των περιφερειακών διαύλων και της κύριας μνήμης. Συχνά, ταυτίζονται με τις φυσικές διευθύνσεις που χρησιμοποιούνται από τον επεξεργαστή αλλά αυτό δε συμβαίνει απαραίτητα. Κάποιες αρχιτεκτονικές παρέχουν μια μονάδα διαχείρισης μνήμης Ε/Ε (IOMMU), η οποία εκτελεί μια λειτουργία αντίστοιχη με αυτή της εικονικής μνήμης ανάμεσα σε έναν δίαυλο και τη κύρια μνήμη. Οι διευθύνσεις αυτές, προφανώς, εξαρτώνται πολύ

από την υποκείμενη αρχιτεκτονική.

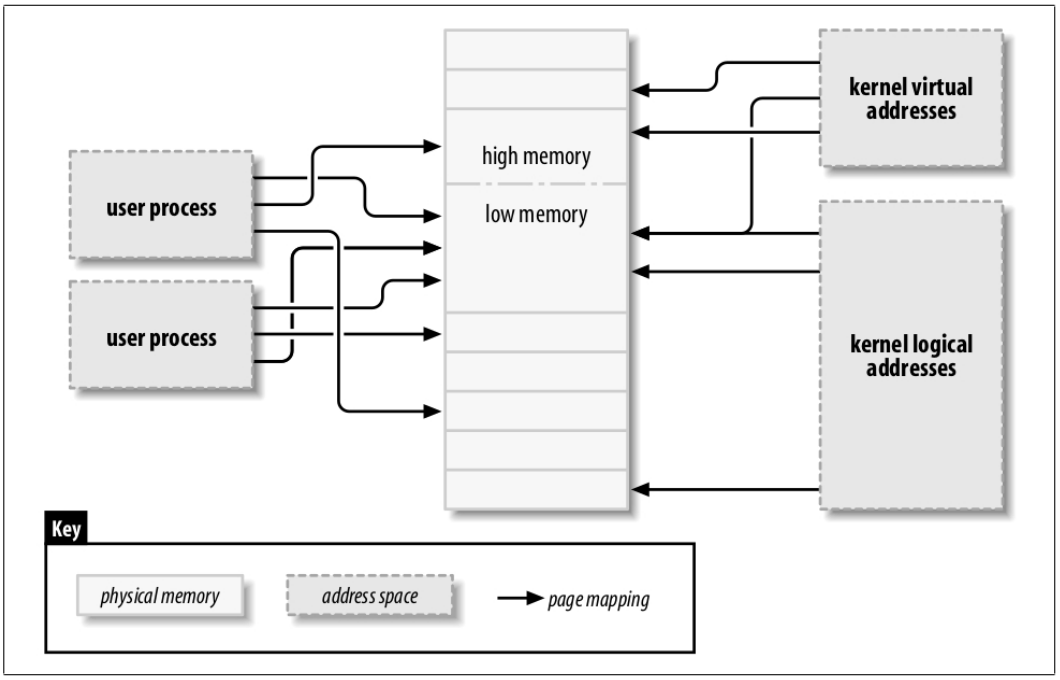
- *Λογικές διευθύνσεις πυρήνα (Kernel logical addresses)*
  Οι διευθύνσεις αυτές συνιστούν τον κανονικό χώρο διευθύνσεων του πυρήνα. Απεικονίζουν ένα μέρος ή πολλές φορές και όλη την κύρια μνήμη και συχνά αντιμετωπίζονται σα να ήταν φυσικές διευθύνσεις. Στις περισσότερες αρχιτεκτονικές οι λογικές διευθύνσεις και οι αντίστοιχες τους φυσικές διαφέρουν μόνο κατά έναν σταθερό παράγοντα (offset). Το μέγεθος των διευθύνσεων αυτών είναι το ίδιο με το μέγεθος του δείκτη του συστήματος, που υπαγορεύεται από το υλικό, και γι αυτό σε κάποια συστήματα 32-bit δεν είναι σε θέση να προσπελάσουν όλη τη φυσική μνήμη. Αποθηκεύονται, συνήθως, σε μεταβλητές τύπου *unsigned long* ή *void **.

- *Εικονικές διευθύνσεις πυρήνα (Kernel virtual addresses)*
  Οι εικονικές διευθύνσεις πυρήνα είναι παρόμοιες με τις λογικές διευθύνσεις, με την έννοια ότι και οι δύο αποτελούν μια απεικόνιση από το χώρο διευθύνσεων του πυρήνα σε φυσικές διευθύνσεις. Οι εικονικές διευθύνσεις, όμως, δεν έχουν απαραίτητα την ένα προς ένα, γραμμική αντιστοιχία με τις φυσικές διευθύνσεις που χαρακτηρίζει το λογικό χώρο διευθύνσεων. Όλες οι λογικές διευθύνσεις είναι και εικονικές διευθύνσεις, αλλά το αντίθετο δεν ισχύει πάντα. Αποθηκεύονται, συνήθως, σε μεταβλητές τύπου δείκτη.

2.2.2 Φυσικές διευθύνσεις και σελίδες

Η φυσική μνήμη του συστήματος χωρίζεται σε διακριτές μονάδες που λέγονται σελίδες. Ο χειρισμός της μνήμης από το λειτουργικό σύστημα γίνεται στο μεγαλύτερο βαθμό του με βάση τις σελίδες. Το μέγεθος μιας σελίδας κυμαίνεται ανάλογα με την αρχιτεκτονική, αλλά τα περισσότερα συστήματα χρησιμοποιούν σελίδες μεγέθους 4096 bytes. Η σταθερά *PAGE_SIZE*, που ορίζεται στο αρχείο κεφαλίδα *<asm/page.h>*, δίνει το τρέχον μέγεθος της σελίδας.

Μια διεύθυνση μνήμης, είτε φυσική είτε εικονική, διαιρείται σε έναν αριθμό σελίδας και σε ένα offset μέσα στη σελίδα. Αν χρησιμοποιούνται, για παράδειγμα, σελίδες 4096 bytes, τα 12 λιγότερο σημαντικά bits είναι το offset και τα υπόλοιπα περισσότερο σημαντικά bits είναι ο αριθμός της σελίδας. Αν αποκοπεί το offset και γίνει ολίσθηση του



Σχήμα 2.3: *Τύποι διευθύνσεων στο Linux*

αριθμού σελίδας προς τα δεξιά το αποτέλεσμα καλείται αριθμός πλαισίου σελίδας (page frame number - pfn). Η μετατροπή ανάμεσα σε διευθύνσεις και αριθμούς πλαισίων είναι μια συνηθισμένη πρακτική γι αυτό ο πυρήνας παρέχει τη μακροεντολή *PAGE_SHIFT*, η οποία καθορίζει πόσα bits πρέπει να ολισθήσουν για να γίνει η μετατροπή.

2.2.3 Μνήμη High και Low

Η διαφορά μεταξύ των λογικών και εικονικών διευθύνσεων του πυρήνα φαίνεται καθαρά σε συστήματα 32-bit εξοπλισμένα με μεγάλου μεγέθους μνήμη. Με 32 bits είναι δυνατόν να διευθυνσιοδοτηθούν 4 GB μνήμης. Το Linux σε τέτοια συστήματα περιοριζόταν σε σημαντικά λιγότερη μνήμη από αυτή, λόγω του τρόπου που δημιουργεί και διαχειρίζεται το χώρο εικονικών διευθύνσεων.

Ο πυρήνας (στην x86 αρχιτεκτονική) χωρίζει τον 4 GB χώρο εικονικών διευθύνσεων στο χώρο χρήστη και στο χώρο πυρήνα. Το ίδιο σύνολο απεικονίσεων χρησιμοποιείται και στις δύο περιπτώσεις. Ένα τυπικό χώρισμα αφιερώνει 3 GB στο χώρο χρήστη και 1 GB στο χώρο πυρήνα. Ο κώδικας και οι δομές του πυρήνα πρέπει να χωρέσουν εκεί, όμως το μεγαλύτερο χώρο καταλαμβάνουν οι εικονικές απεικονίσεις (virtual mappings) για την πραγματική μνήμη. Ο πυρήνας δεν μπορεί να διαχειριστεί μνήμη που δεν απει-

κονίζεται στο χώρο διευθύνσεων του. Με άλλα λόγια ο πυρήνας χρειάζεται μια δική του εικονική διεύθυνση για οποιοδήποτε κομμάτι μνήμης θέλει να προσπελάσει. Έτσι, για πολλά χρόνια, το μεγαλύτερο μέγεθος φυσικής (πραγματικής) μνήμης που μπορούσε να χειριστεί ο πυρήνας ήταν το μέγεθος που μπορούσε να απεικονιστεί στο χώρο εικονικών διευθύνσεων του, μειωμένο κατά το χώρο που καταλαμβάνει ο ίδιος ο πυρήνας.

Σε απάντηση της εμπορικής πίεσης για περισσότερη μνήμη, οι κατασκευαστές επεξεργαστών προσέθεσαν δυνατότητες "επέκτασης διευθύνσεων" στα προϊόντα τους, διατηρώντας τη συμβατότητα με τις υπάρχουσες 32-bit εφαρμογές. Το αποτέλεσμα είναι ότι σε πολλές περιπτώσεις ακόμα και 32-bit επεξεργαστές μπορούν να διευθυνσιοδοτήσουν περισσότερα από 4 GB φυσικής μνήμης. Βέβαια, παραμένει ο περιορισμός της ένα προς ένα απεικόνισης φυσικής μνήμης με τις λογικές διευθύνσεις. Μόνο το χαμηλότερο κομμάτι της μνήμης (μέχρι 1 ή 2 GB, ανάλογα το υλικό και τη παραμετροποίηση του πυρήνα) έχει λογικές διευθύνσεις, ενώ το υπόλοιπο (high memory) δεν έχει. Πριν την προσπέλαση μιας συγκεκριμένης high-memory σελίδας, ο πυρήνας πρέπει να κατασκευάσει μια εικονική απεικόνιση (virtual mapping), για να μπορεί να έχει τη σελίδα αυτή διαθέσιμη στο χώρο διευθύνσεων του. Έτσι αρκετές δομές δεδομένων του πυρήνα πρέπει να τοποθετηθούν στη low memory, ενώ η high memory δεσμεύεται κυρίως για εφαρμογές χώρου χρήστη.

Ακολούθως ορίζονται οι όροι High και Low memory:

- *Low memory*
  Μνήμη για την οποία υπάρχουν λογικές διευθύνσεις στο χώρο διευθύνσεων του πυρήνα. Στα περισσότερα συστήματα, συνήθως, είναι όλη η διαθέσιμη πραγματική μνήμη.

- *High memory*
  Μνήμη για την οποία δεν υπάρχουν λογικές διευθύνσεις, γιατί ξεπερνά το όριο του χώρου διευθύνσεων που έχει δεσμευτεί για τις εικονικές διευθύνσεις του πυρήνα.

Τέλος, αξίζει να σημειώσουμε ότι στα συστήματα *x86-64* [Kle04] όλη η μνήμη είναι low memory. Ο διαθέσιμος χώρος εικονικών διευθύνσεων στα 64-bit είναι πολύ μεγαλύτερος από τη μέγιστη φυσική μνήμη, με την οποία μπορεί να είναι εξοπλισμένο ένα

σύστημα, με αποτέλεσμα στο χώρο χρήστη και στο χώρο πυρήνα να μπορούν να αποδοθούν διαφορετικά κομμάτια του εικονικού χώρου διευθύνσεων, τα οποία επαρκούν για την άμεση απεικόνιση ολόκληρης της φυσικής μνήμης.

2.2.4 Η δομή *struct page* και ο Χάρτης Μνήμης του συστήματος

Ο πυρήνας πρέπει να γνωρίζει την τρέχουσα κατάσταση κάθε σελίδας φυσικής μνήμης. Για παράδειγμα, πρέπει να μπορεί να ξεχωρίσει τις σελίδες που περιέχουν δεδομένα μιας διεργασίας από εκείνες που περιέχουν κώδικα ή δεδομένα του πυρήνα. Όμοια, πρέπει να μπορεί να διακρίνει αν μία σελίδα φυσικής μνήμης στη δυναμική μνήμη είναι ελεύθερη ή όχι.

Οι πληροφορίες κατάστασης μιας σελίδας αποθηκεύονται σε μια δομή τύπου *struct page*. Για κάθε σελίδα φυσικής μνήμης του συστήματος υπάρχει μία δομή τύπου *struct page*, η οποία περιέχει ο,τιδήποτε χρειάζεται να γνωρίζει ο πυρήνας για τη σελίδα αυτή. Ορισμένα από τα πιο σημαντικά πεδία αυτής της δομής είναι:

- *atomic_t count*
  Ο αριθμός των αναφορών σε αυτή τη σελίδα φυσικής μνήμης. Όταν αυτός ο αριθμός πέσει στο 0, η σελίδα επιστρέφει στη λίστα με τις ελεύθερες σελίδες του συστήματος.

- *unsigned long flags*
  Ένα σύνολο από σημαίες που περιγράφουν την κατάσταση της σελίδας. Αυτές περιλαμβάνουν τη σημαία PG_locked, η οποία δηλώνει ότι η σελίδα είναι “κλειδωμένη” στη μνήμη, και τη σημαία PG_reserved, η οποία εμποδίζει το υποσύστημα διαχείρισης μνήμης να “ακουμπήσει” τη σελίδα με οποιονδήποτε τρόπο.

- *atomic_t _mapcount*
  Το πλήθος των καταχωρήσεων πινάκων σελίδων (page table entries, 2.2.5) που αναφέρονται σε αυτή τη σελίδα.

- *struct address_space *mapping*
  Χρησιμοποιείται όταν η σελίδα ανήκει στη κρυφή μνήμη σελίδων (page cache) ή όταν ανήκει σε μια ανώνυμη περιοχή μνήμης (2.2.6).

- *pgoff_t index*
  Προσδιορίζει τη θέση των δεδομένων, που είναι αποθηκευμένα στην αντίστοιχη φυσική σελίδα μνήμης, μέσα σε ένα αρχείο ή σε μια ανώνυμη περιοχή μνήμης (είναι το offset της σελίδας μέσα στο mapping).

Ο πυρήνας διατηρεί έναν ή περισσότερους πίνακες από δομές *struct page* που καλύπτουν ολόκληρη τη φυσική μνήμη του συστήματος. Ένας τέτοιος πίνακας καλείται "χάρτης μνήμης" (memory map). Το πλήθος των πινάκων αυτών εξαρτάται από το εκάστοτε σύστημα. Για παράδειγμα, στα συστήματα NUMA (Nonuniform memory access), όπου η μνήμη του συστήματος χωρίζεται σε πολλούς κόμβους (memory nodes), και σε αυτά με μη συνεχή χώρο φυσικών διευθύνσεων, μπορεί να υπάρχουν πολλοί "χάρτες μνήμης", ένας για κάθε κόμβο μνήμης. Ο πυρήνας αντιμετωπίζει τα συστήματα UMA (Uniform memory access) σαν ειδική περίπτωση συστήματος NUMA με έναν μόνο κόμβο μνήμης. Αυτή η προσέγγιση κάνει τον κώδικα χειρισμού της μνήμης πιο φορητό, αφού ο πυρήνας μπορεί να θεωρήσει ότι η φυσική μνήμη διαμερίζεται σε έναν ή περισσότερους κόμβους, ανεξαρτήτως αρχιτεκτονικής.

2.2.5 Πίνακες σελίδων (Page Tables)

Σε οποιοδήποτε σύγχρονο σύστημα, ο επεξεργαστής χρειάζεται έναν μηχανισμό για να μετατρέπει τις εικονικές διευθύνσεις στις αντίστοιχες τους φυσικές. Αυτός ο μηχανισμός καλείται πίνακας σελίδων (page table). Πρόκειται, ουσιαστικά, για έναν πολύεπίπεδο πίνακα με δομή δέντρου που περιέχει απεικονίσεις από εικονικές σε φυσικές διευθύνσεις και μερικά bits - σημαίες που χαρακτηρίζουν τη κατάσταση και τις ιδιότητες της κάθε καταχώρησης (page table entry).

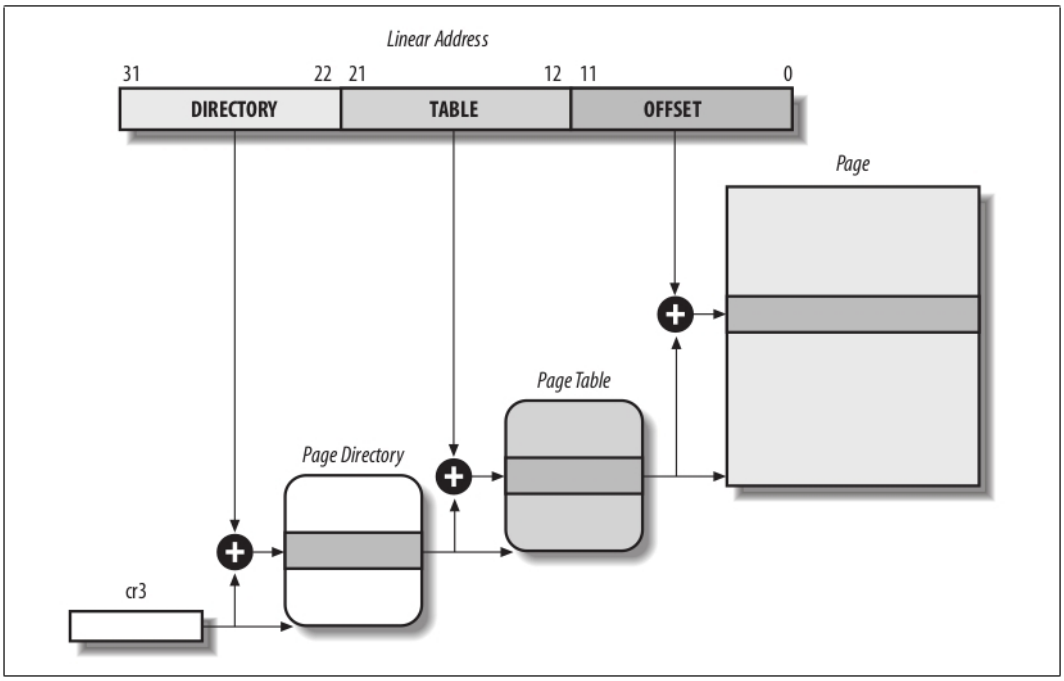
Για λόγους αποδοτικότητας οι εικονικές διευθύνσεις ομαδοποιούνται σε διαστήματα ορισμένου μεγέθους, τα οποία καλούνται σελίδες. Οι συνεχόμενες εικονικές διευθύνσεις μέσα σε μια σελίδα απεικονίζονται σε συνεχόμενες φυσικές διευθύνσεις. Κατά αυτό τον τρόπο ο πυρήνας μπορεί να προσδιορίσει τη φυσική διεύθυνση και τα δικαιώματα πρόσβασης μιας σελίδας, ως σύνολο, αντί κάθε εικονικής διεύθυνσης, μέσα στη σελίδα, χωριστά.

Προκειμένου να διακρίνουμε τις σελίδες φυσικής μνήμης (βλέπε 2.2.2) από τις σελίδες εικονικής μνήμης θα αναφερόμαστε στις πρώτες ως πλαίσια φυσικής μνήμης και στις

δεύτερες απλά ως σελίδες. Επιπρόσθετα, με τον όρο σελίδα θα αναφερόμαστε τόσο σε ένα σύνολο εικονικών διευθύνσεων, όσο και στα δεδομένα που περιέχονται σε αυτές τις διευθύνσεις. Είναι σημαντικό, στο σημείο αυτό, να κατανοήσουμε τη διαφορά ανάμεσα στις σελίδες και στα πλαίσια. Ένα πλαίσιο φυσικής μνήμης είναι μέρος της φυσικής μνήμης του συστήματος και επομένως αποτελεί ένα κομμάτι αποθηκευτικού χώρου. Αντίθετα, μια σελίδα εικονικής μνήμης είναι ένα σύνολο από δεδομένα, το οποίο μπορεί να αποθηκευτεί σε οποιαδήποτε πλαίσιο φυσικής μνήμης ή στο δίσκο.

Η μετάφραση μιας εικονικής διεύθυνσης σε φυσική ακολουθεί τα εξής βήματα: Η εικονική διεύθυνση χωρίζεται σε έναν αριθμό από πεδία (fields). Ο αριθμός των πεδίων εξαρτάται από τον αριθμό των επιπέδων του πίνακα σελίδων. Σε ένα σχήμα με δύο επίπεδα, για παράδειγμα, η διεύθυνση χωρίζεται σε τρία τμήματα. Το πρώτο τμήμα δεικτοδοτεί το πρώτο επίπεδο του πίνακα σελίδων, το δεύτερο τμήμα το δεύτερο επίπεδο και το τρίτο τμήμα χρησιμοποιείται ως σχετική θέση (offset) μέσα στο πλαίσιο φυσικής μνήμης. Οι καταχωρήσεις κάθε επιπέδου του πίνακα σελίδων περιέχουν είτε τη διεύθυνση ενός πίνακα του επόμενου επιπέδου, είτε του πλαισίου μνήμης που αντιστοιχεί στην εικονική διεύθυνση που γίνεται η αναφορά. Ο στόχος αυτού του ιεραρχικού σχήματος είναι να μειώσει την ποσότητα μνήμης που απαιτείται για τους πίνακες σελίδων κάθε διεργασίας, αφού δεσμεύεται μνήμη μόνο για τους πίνακες που αντιστοιχούν σε περιοχές του εικονικού χώρου διευθύνσεων που όντως χρησιμοποιούνται από τη διεργασία. Ο αριθμός των επιπέδων του πίνακα σελίδων εξαρτάται από την εκάστοτε αρχιτεκτονική. Για παράδειγμα, στους επεξεργαστές *x86* χωρίς PAE (Physical Address Extension) χρησιμοποιούνται δύο επίπεδα (Σχήμα 2.4), με PAE τρία επίπεδα και στους *x86-64* τέσσερα επίπεδα.

Πέρα από τη μετάφραση των εικονικών διευθύνσεων σε φυσικές, η μονάδα σελιδοποίησης (paging unit) ενός επεξεργαστή επιτελεί μία ακόμη, ιδιαίτερα σημαντική, λειτουργία: Ελέγχει αν ο αιτούμενος τύπος πρόσβασης στη μνήμη (για παράδειγμα εγγραφή, ανάγνωση ή εκτέλεση) συμφωνεί με τα δικαιώματα πρόσβασης της διεργασίας στην αντίστοιχη σελίδα εικονικής μνήμης. Αυτό είναι εφικτό, γιατί ο πίνακας σελίδων, όπως αναφέρθηκε και προηγουμένως, αποθηκεύει για κάθε καταχώρηση ορισμένα bit - σημαίες, τα οποία μας δίνουν διάφορες πληροφορίες για την αντίστοιχη σελίδα, μεταξύ των οποίων είναι και τα δικαιώματα πρόσβασης σε αυτή. Αν μία πρόσβαση στη μνήμη δεν είναι έγκυρη (ελλειπή δικαιώματα πρόσβασης ή η σελίδα δεν είναι παρούσα στη μνήμη), τότε η μονάδα σελιδοποίησης του επεξεργαστή προκαλεί μία εξαίρεση σφάλ-



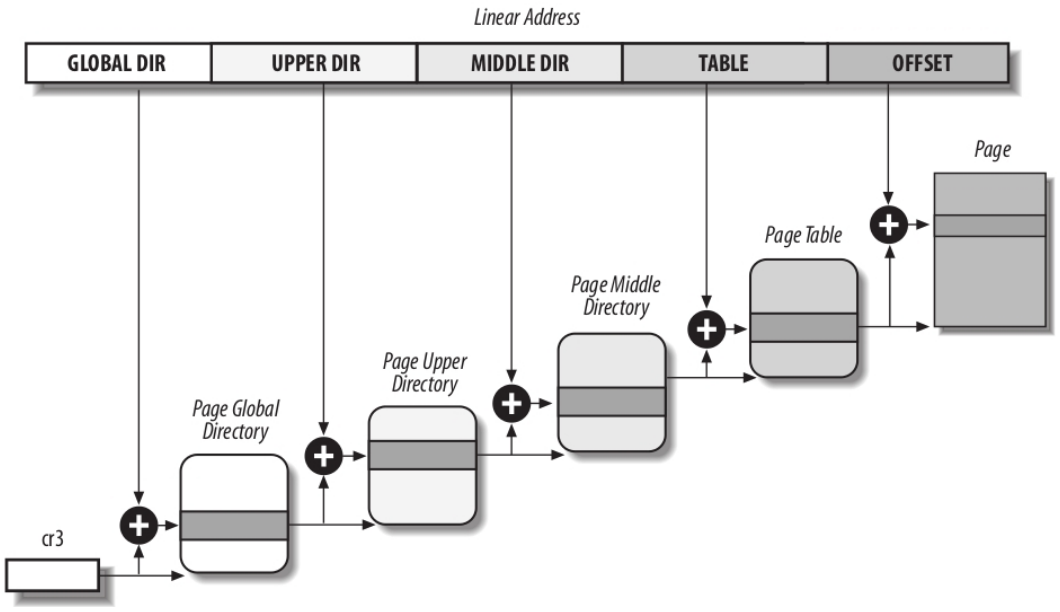
Σχήμα 2.4: *Πίνακας σελίδων επεξεργαστών x86*

ματος σελίδας (page fault exception), η οποία πυροδοτεί την εκτέλεση του χειριστή σφάλματος σελίδας (page fault handler) του λειτουργικού συστήματος.

Ο πυρήνας του Linux υιοθετεί ένα κοινό μοντέλο σελιδοποίησης τεσσάρων επιπέδων τόσο για αρχιτεκτονικές 32-bit, όσο και για αρχιτεκτονικές 64-bit. Τα τέσσερα είδη των πινάκων σελιδοποίησης (Σχήμα 2.5) είναι:

- *Καθολικός Κατάλογος Σελιδοποίησης (Page Global Directory)*
- *Ανώτερος Κατάλογος Σελιδοποίησης (Page Upper Directory)*
- *Μεσαίος Κατάλογος Σελιδοποίησης (Page Middle Directory)*
- *Πίνακας Σελίδων (Page Table)*

Ο Καθολικός Κατάλογος Σελιδοποίησης περιέχει τις διευθύνσεις αρκετών Ανώτερων Καταλόγων Σελιδοποίησης, οι οποίοι με τη σειρά τους περιέχουν τις διευθύνσεις αρκετών Μεσαίων Καταλόγων Σελιδοποίησης και αυτοί με τη σειρά τους τις διευθύνσεις αρκετών Πινάκων Σελίδων. Κάθε τιμή του Πίνακα Σελίδων είναι η διεύθυνση ενός πλαισίου φυσικής μνήμης. Έτσι, η εικονική διεύθυνση μπορεί να χωριστεί σε πέντε, το πολύ,



Σχήμα 2.5: *Το μοντέλο σελιδοποίησης του Linux*

μέρη. Στο σχήμα 2.5 φαίνεται αυτός ο διαχωρισμός, χωρίς όμως να φαίνεται το μέγεθος κάθε πεδίου, αφού αυτό εξαρτάται από την αρχιτεκτονική του υπολογιστή.

Για αρχιτεκτονικές 32-bit, χωρίς επέκταση φυσικών διευθύνσεων (PAE), δύο επίπεδα σελιδοποίησης είναι αρκετά. Ο πυρήνας του Linux ουσιαστικά καταργεί τον Ανώτερο και τον Μεσαίο Κατάλογο σελιδοποίησης, θεωρώντας ότι τα αντίστοιχα πεδία της εικονικής διεύθυνσης περιέχουν 0 bits. Ωστόσο, διατηρούνται οι θέσεις των δύο αυτών καταλόγων στη διαδοχή των δεικτών προκειμένου ο ίδιος κώδικας να δουλεύει τόσο σε αρχιτεκτονικές 32-bit, όσο και σε αρχιτεκτονικές 64-bit. Ο πυρήνας το καταφέρνει αυτό θέτοντας τον αριθμό των καταχωρήσεων για τους δύο αυτούς καταλόγους στη μία και απεικονίζοντας αυτές τις δύο καταχωρήσεις στην κατάλληλη καταχώρηση του Καθολικού Καταλόγου Σελιδοποίησης.

Για αρχιτεκτονικές 32-bit, με την επέκταση φυσικών διευθύνσεων (PAE) ενεργοποιημένη, απαιτούνται τρία επίπεδα σελιδοποίησης. Ο Καθολικός Κατάλογος Σελιδοποίησης του Linux αντιστοιχεί στον Πίνακα Δεικτών Καταλόγου Σελίδων (Page Directory Pointer Table) του *80x86*, ο Ανώτερος Κατάλογος Σελιδοποίησης καταργείται, ο Μεσαίος Κατάλογος Σελιδοποίησης αντιστοιχεί στον Κατάλογο Σελίδων (Page Directory) του *80x86* και ο Πίνακας Σελίδων του Linux αντιστοιχεί στον Πίνακα Σελίδων (Page Table) του *80x86*.

Τέλος, για αρχιτεκτονικές 64-bit απαιτούνται τρία ή τέσσερα επίπεδα σελιδοποίησης

Πίνακας 2.1: *Επίπεδα σελιδοποίησης σε ορισμένες αρχιτεκτονικές 64-bit.*

Όνομα Πλατφόρμας	Μέγεθος Σελίδας	Πλήθος bits διεύθυνσης	Πλήθος Επιπέδων Σελιδοποίησης	Διαμερισμός Εικονικής Διεύθυνσης
alpha	8KB	43	3	10 + 10 + 10 + 13
ia64	4KB	39	3	9 + 9 + 9 + 12
ppc64	4KB	41	3	10 + 10 + 9 + 12
sh64	4KB	41	3	10 + 10 + 9 + 12
x86-64	4KB	48	4	9 + 9 + 9 + 9 + 12

ανάλογα με το πώς το υλικό διαμερίζει σε πεδία την εικονική διεύθυνση (βλέπε Πίνακα 2.1).

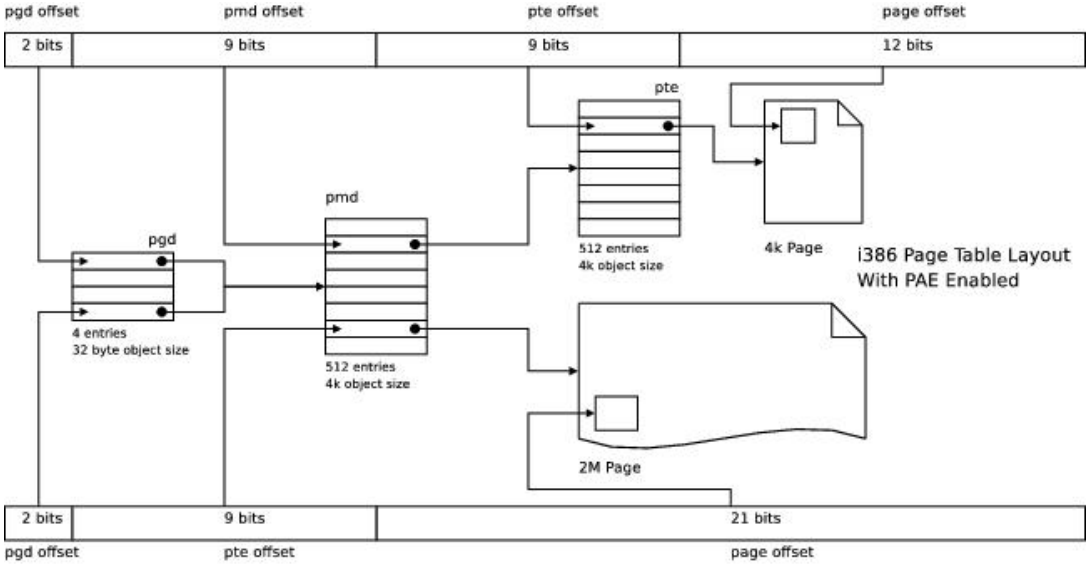
Το Linux βασίζεται πολύ στο μηχανισμό σελιδοποίησης για τη διαχείριση των διεργασιών. Κάθε διεργασία έχει τους δικούς της Πίνακες Σελίδων. Η αυτόματη μετάφραση των εικονικών διευθύνσεων σε φυσικές, από τη μονάδα σελιδοποίησης (paging unit), καθιστά εφικτούς του ακόλουθους σχεδιαστικούς στόχους:

- Ανάθεση διαφορετικού χώρου φυσικών διευθύνσεων σε κάθε διεργασία, εξασφαλίζοντας αποτελεσματική προστασία έναντι των σφαλμάτων διευθυνσιοδότησης.
- Διαφοροποίηση ανάμεσα στις σελίδες (σύνολο δεδομένων) και στα πλαίσια φυσικής μνήμης (φυσικές διευθύνσεις στη κύρια μνήμη του συστήματος). Η διάκριση αυτή επιτρέπει σε μια σελίδα να αποθηκευτεί σε ένα πλαίσιο μνήμης, στη συνέχεια να αποθηκευτεί στον δίσκο και αργότερα να φορτωθεί ξανά σε ένα διαφορετικό πλαίσιο μνήμης. Αυτή η δυνατότητα είναι το βασικό συστατικό του μηχανισμού εικονικής μνήμης.

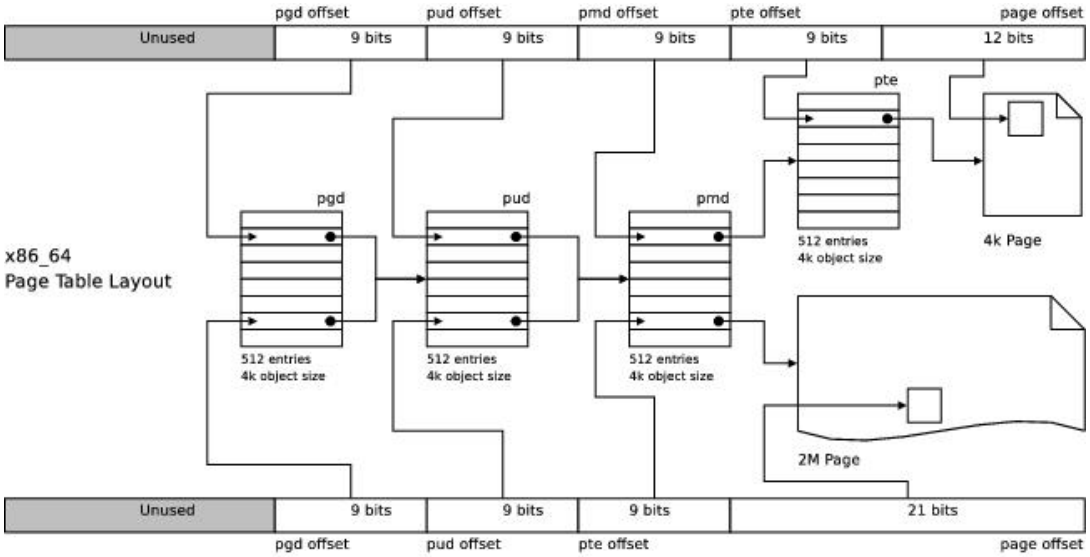
Τα σχήματα 2.6, 2.7, 2.8 και 2.9 αποτελούν παραδείγματα του συστήματος σελιδοποίησης του Linux για κάποιες ευρέως διαδεδομένες αρχιτεκτονικές.

2.2.6 Περιοχές εικονικής μνήμης (Virtual Memory Areas)

Η περιοχή εικονικής μνήμης (Virtual Memory Area ή VMA) είναι η δομή που χρησιμοποιεί ο πυρήνας για να αναπαραστήσει και να χειριστεί διακριτές περιοχές του χώρου



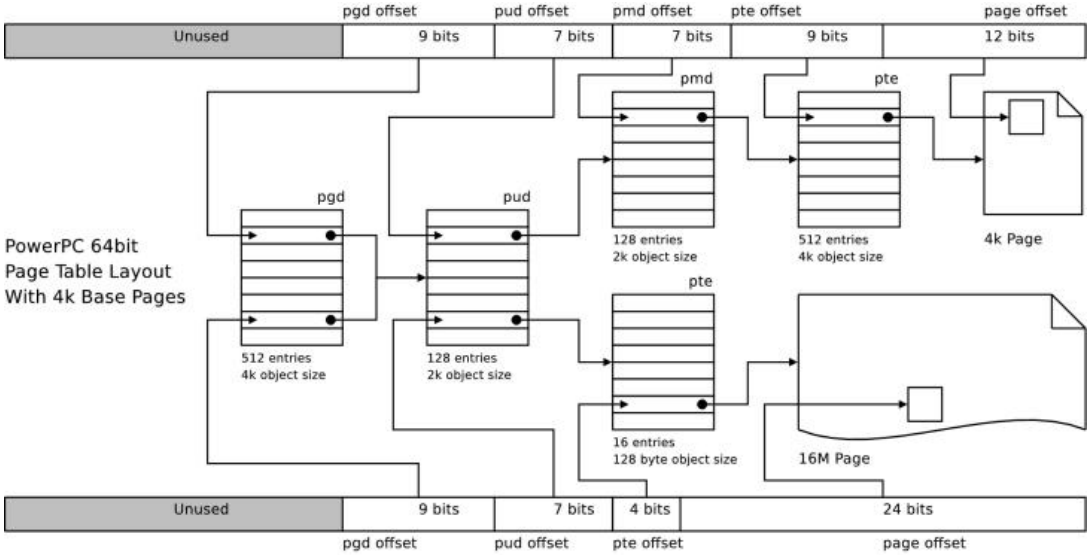
Σχήμα 2.6: *Το μοντέλο σελιδοποίησης του Linux στην αρχιτεκτονική x86 με PAE*



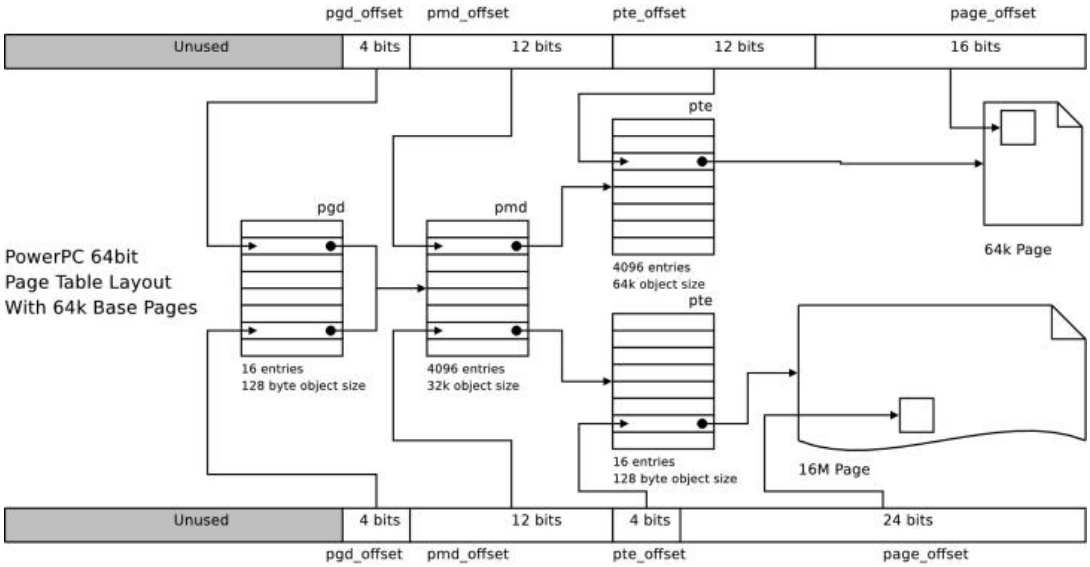
Σχήμα 2.7: *Το μοντέλο σελιδοποίησης του Linux στην αρχιτεκτονική x86-64*

διευθύνσεων μιας διεργασίας. Μια VMA αναπαριστά μια ομογενή περιοχή της εικονικής μνήμης της διεργασίας: ένα συνεχές εύρος εικονικών διευθύνσεων, οι οποίες έχουν τα ίδια δικαιώματα πρόσβασης (access rights) και υποστηρίζονται από το ίδιο αντικείμενο (ένα αρχείο, για παράδειγμα, ή τη περιοχή swap).

Οι περιοχές εικονικής μνήμης μιας διεργασίας δεν έχουν ποτέ επικαλύψεις και ο πυρήνας προσπαθεί να συγχωνεύει περιοχές, όταν μια καινούρια περιοχή προστίθεται ακριβώς δίπλα σε μία υπάρχουσα. Δύο γειτονικές περιοχές μπορούν να συγχωνευθούν εάν έχουν τα ίδια δικαιώματα πρόσβασης.



Σχήμα 2.8: *Το μοντέλο σελιδοποίησης του Linux στην αρχιτεκτονική PowerPC με μέγεθος σελίδας 4KB*



Σχήμα 2.9: *Το μοντέλο σελιδοποίησης του Linux στην αρχιτεκτονική PowerPC με μέγεθος σελίδας 64KB*

Κάθε διεργασία έχει, τουλάχιστον, τις εξής περιοχές εικονικής μνήμης:

- Μια περιοχή με τον εκτελέσιμο κώδικα της εφαρμογής (που συνήθως ονομάζεται *text*)
- Διάφορες περιοχές για δεδομένα, που περιλαμβάνουν τα αρχικοποιημένα δεδομένα (δηλαδή αυτά που έχουν συγκεκριμένη τιμή κατά την έναρξη της εκτέλεσης του προγράμματος), τα μη αρχικοποιημένα δεδομένα (BSS - Ο πυρήνας απεικο-

νίζει σελίδες γεμάτες μηδενικά σε αυτό το εύρος διευθύνσεων) και τη στοίβα του προγράμματος

- Μια περιοχή για κάθε ενεργή απεικόνιση (mapping) της διεργασίας

Ο πυρήνας αναπαριστά εσωτερικά μια περιοχή εικονικής μνήμης με τη δομή *struct vm_area_struct*. Τα κυριότερα πεδία της δομής αυτής είναι:

- *unsigned long vm_start*, *unsigned long vm_end*
  Η περιοχή εικονικών διευθύνσεων που καλύπτεται από αυτή τη VMA.

- *struct file *vm_file*
  Ένας δείκτης στη δομή *struct file* που σχετίζεται με αυτή τη περιοχή ή NULL αν πρόκειται για ανώνυμη περιοχή μνήμης (anonymoys memory region).

- *unsigned long vm_pgoff*
  Η σχετική θέση (offset) της περιοχής μέσα στο *vm_file*, σε σελίδες.

- *unsigned long vm_flags*
  Ένα σύνολο από σημαίες που περιγράφουν την περιοχή.

- *pgprot_t vm_page_prot*
  Τα δικαιώματα πρόσβασης (access permisions) για τις σελίδες που ανήκουν σε αυτή την περιοχή.

- *struct mm_struct *vm_mm*
  Δείκτης στο χώρο διευθύνσεων στον οποίο ανήκει αυτή η περιοχή (βλέπε 2.2.8).

- *struct vm_operations_struct *vm_ops*
  Ένα σύνολο από συναρτήσεις που καλεί ο πυρήνας για να χειριστεί την περιοχή εικονικής μνήμης.

- *void *vm_private_data*
  Δείκτης σε ιδιωτικά δεδομένα της περιοχής.

2.2.7 Ζώνες Μνήμης (Memory Zones)

Σε ένα ιδανικό υπολογιστικό σύστημα ένα πλαίσιο μνήμης είναι μια μονάδα αποθηκευτικού χώρου που μπορεί να χρησιμοποιηθεί για οτιδήποτε: να αποθηκεύσει δεδομένα χώρου χρήστη ή χώρου πυρήνα, δεδομένα ενός δίσκου (disk buffer) κ.τ.λ. Οποιαδήποτε σελίδα δεδομένων μπορεί να αποθηκευτεί σε οποιοδήποτε πλαίσιο μνήμης χωρίς περιορισμούς.

Παρόλα αυτά, οι πραγματικές αρχιτεκτονικές μπορεί να θέτουν περιορισμούς στους τρόπους με τους οποίους μπορεί να χρησιμοποιηθεί ένα πλαίσιο μνήμης. Για παράδειγμα, στην αρχιτεκτονική *x86*, υπάρχουν δύο σημαντικοί περιορισμοί:

- Οι επεξεργαστές άμεσης προσπέλασης μνήμης (Direct Memory Access processors) για τους παλιούς διαύλους ISA μπορούν να προσπελάσουν μόνο τα πρώτα 16 MB της κύριας μνήμης.
- Σε συστήματα 32-bit, με μεγάλη ποσότητα κύριας μνήμης, ο επεξεργαστής δεν μπορεί να προσπελάσει απευθείας όλη τη φυσική μνήμη, επειδή ο χώρος εικονικών διευθύνσεων είναι μικρός.

Για να αντιμετωπίσει αυτούς τους περιορισμούς ο πυρήνας του Linux διαμερίζει τη φυσική μνήμη κάθε κόμβου μνήμης (memory node) σε ζώνες.

Σε ένα *x86* σύστημα UMA (Uniform memory access) οι ζώνες είναι:

- *ZONE_DMA*
  Περιέχει πλαίσια μνήμης για τη μνήμη κάτω από τα 16 MB.
- *ZONE_NORMAL*
  Περιέχει πλαίσια μνήμης για τη μνήμη από 16 MB μέχρι κάτω από τα 896 MB.
- *ZONE_HIGHMEM*
  Περιέχει πλαίσια μνήμης για τη μνήμη από τα 896 MB και πάνω.

Η ζώνη ZONE_DMA περιλαμβάνει πλαίσια μνήμης που μπορούν να χρησιμοποιηθούν από παλιότερες ISA συσκευές μέσω του μηχανισμού DMA. Οι ζώνες ZONE_DMA και ZONE_NORMAL περιλαμβάνουν τα πλαίσια μνήμης που μπορούν να απεικονιστούν

απευθείας στο χώρο διευθύνσεων του πυρήνα και άρα να προσπελαστούν άμεσα από αυτόν. Αντίθετα, η ζώνη ZONE_HIGHMEM περιλαμβάνει πλαίσια μνήμης που δεν μπορούν να προσπελαστούν άμεσα από τον πυρήνα (βλέπε 2.2.3).

Στις 64-bit αρχιτεκτονικές η ζώνη ZONE_HIGHMEM είναι πάντα κενή, αφού μπορούν να προσπελάσουν άμεσα όλη τη διαθέσιμη φυσική μνήμη. Συγκεκριμένα σε αυτές τις αρχιτεκτονικές, οι ζώνες μνήμης είναι:

- *ZONE_DMA*
  Περιέχει πλαίσια μνήμης για τη μνήμη κάτω από τα 16 MB.
- *ZONE_DMA32*
  Περιέχει πλαίσια μνήμης για τη μνήμη από 16 MB μέχρι κάτω από τα 4 GB.
- *ZONE_NORMAL*
  Περιέχει πλαίσια μνήμης για τη μνήμη από τα 4 GB και πάνω.

Η ζώνη ZONE_DMA έχει την ίδια σημασία όπως και στις 32-bit αρχιτεκτονικές. Η ζώνη ZONE_DMA32 περιλαμβάνει πλαίσια μνήμης που έχουν φυσικές διευθύνσεις των 32-bit και μπορεί να χρησιμοποιηθεί από συσκευές που δεν μπορούν να προσπελάσουν, μέσω DMA, μνήμη που απαιτεί πάνω από 32-bit για τη διευθυνσιοδότηση της. Η ζώνη ZONE_NORMAL περιλαμβάνει όλα τα υπόλοιπα πλαίσια φυσικής μνήμης, αν υπάρχουν διαθέσιμα πάνω από 4 GB μνήμης στο σύστημα.

Όταν ο πυρήνας καλεί μια συνάρτηση δέσμευσης μνήμης (memory allocation function) προσδιορίζει από ποιες ζώνες επιθυμεί να προέρχεται η μνήμη.

2.2.8 Χώρος διευθύνσεων διεργασίας

Ο χώρος διευθύνσεων μιας διεργασίας είναι το σύνολο των εικονικών διευθύνσεων, τις οποίες μπορεί να χρησιμοποιήσει η διεργασία. Κάθε διεργασία έχει το δικό της χώρο διευθύνσεων, ο οποίος είναι ανεξάρτητος από τους χώρους διευθύνσεων των άλλων διεργασιών του συστήματος. Δηλαδή, μια διεύθυνση σε μια διεργασία δεν έχει καμία συσχέτιση με την ίδια διεύθυνση σε άλλες διεργασίες. Εξαίρεση σε αυτόν τον κανόνα αποτελούν ορισμένα νήματα πυρήνα (kernel threads).

Ο χώρος διευθύνσεων μιας διεργασίας αναπαρίσταται, στον πυρήνα του Linux, από τη δομή *struct mm_struct*. Η δομή αυτή περιέχει τη λίστα με τις περιοχές εικονικής μνήμης (VMAs) της διεργασίας, τους πίνακες σελίδων της διεργασίας και διάφορες άλλες πληροφορίες σχετικές με τη διαχείριση του χώρου διευθύνσεων της διεργασίας. Τη δομή αυτή μπορεί να τη μοιράζονται περισσότερες από μία διεργασίες. Με τον τρόπο αυτό το Linux υλοποιεί τα νήματα (threads).

2.2.9 Χειρισμός της κρυφής μνήμης TLB

Οι περισσότεροι σύγχρονοι επεξεργαστές, συμπεριλαμβανομένων αυτών της οικογένειας 80x86, περιλαμβάνουν μία κρυφή μνήμη που ονομάζεται Translation Lookaside Buffers (TLB). Στόχος του TLB είναι η επιτάχυνση της μετάφρασης των εικονικών διευθύνσεων σε φυσικές. Κατά την πρώτη πρόσβαση σε μία εικονική διεύθυνση, η αντίστοιχη της φυσική διεύθυνση εντοπίζεται με χρήση των Πινάκων Σελίδων (Page Tables) που βρίσκονται στη κύρια μνήμη του συστήματος. Η φυσική διεύθυνση στη συνέχεια αποθηκεύεται στην κρυφή μνήμη TLB, με αποτέλεσμα, κατά τις επόμενες προσπελάσεις στην ίδια εικονική διεύθυνση (και γενικότερα σε όσες εικονικές διευθύνσεις ανήκουν στην ίδια σελίδα εικονικής μνήμης), η μετάφραση να μπορεί να γίνει πολύ γρηγορότερα. Σε ένα πολύ-επεξεργαστικό σύστημα κάθε επεξεργαστής έχει τη δική του κρυφή μνήμη TLB, που αποκαλείται τοπικός TLB (local TLB).

Οι επεξεργαστές δεν μπορούν να συγχρονίσουν αυτόματα την κρυφή τους μνήμη TLB διότι είναι ο πυρήνας, όχι το υλικό, που αποφασίζει πότε μια απεικόνιση, μεταξύ μιας εικονικής και μιας φυσικής διεύθυνσης, δεν είναι πλέον έγκυρη. Ο πυρήνας του Linux προσφέρει διάφορες συναρτήσεις για την ακύρωση όλων ή μεμονωμένων καταχωρήσεων της κρυφής μνήμης TLB (TLB flushing). Στην περίπτωση πολύ-επεξεργαστικών συστημάτων οι συναρτήσεις ακύρωσης των καταχωρήσεων του TLB επεκτείνονται ως εξής: στέλνεται μια δια-επεξεργαστική διακοπή (Interprocessor Interrupt) που αναγκάζει τους υπόλοιπους επεξεργαστές να εκτελέσουν την κατάλληλη λειτουργία ακύρωσης.

Ως γενικός κανόνας, όταν συμβαίνει μια αλλαγή διεργασίας (process switch) σε έναν επεξεργαστή, αλλάζουν οι ενεργοί Πίνακες Σελίδων και επομένως απαιτείται να καθαριστεί (flushed) ο τοπικός TLB. Εξαίρεση σε αυτόν τον κανόνα αποτελεί η περίπτωση, όπου η νέα διεργασία χρησιμοποιεί τους ίδιους πίνακες σελίδων, όπως συμβαίνει στην

περίπτωση των διαφορετικών νημάτων μιας πολυνηματικής διεργασίας ή ενός νήματος πυρήνα (kernel thread). Υπάρχουν, βέβαια, και άλλες περιπτώσεις που απαιτείται η ακύρωση καταχωρήσεων του TLB. Για παράδειγμα, όταν ο πυρήνας καταργεί μια απεικόνιση από τους πίνακες σελίδων μιας διεργασίας, πρέπει να ακυρώσει την αντίστοιχη καταχώρηση από τον τοπικό TLB. Στα πολύ-επεξεργαστικά συστήματα, ο πυρήνας πρέπει να καταργήσει την ίδια καταχώρηση από τις κρυφές μνήμες TLB όλων των επεξεργαστών που χρησιμοποιούν τους ίδιους πίνακες σελίδων (στέλνοντας ένα IPI, όπως αναφέρθηκε προηγουμένως).

Σε ορισμένες περιπτώσεις, στα πολύ-επεξεργαστικά συστήματα, είναι δυνατόν να αποφευχθεί το καθάρισμα του TLB. Για να το πετύχει αυτό, ο πυρήνας, χρησιμοποιεί μία τεχνική που αποκαλείται *lazy TLB mode*. Η βασική ιδέα είναι η εξής: Αν διάφοροι επεξεργαστές χρησιμοποιούν τους ίδιους Πίνακες Σελίδων και μια εγγραφή TLB (TLB entry) πρέπει να καταργηθεί από όλους, τότε είναι δυνατόν, σε ορισμένες περιπτώσεις, το καθάρισμα του TLB να καθυστερήσει, αν η αντίστοιχη CPU τρέχει ένα νήμα πυρήνα. Τα νήματα πυρήνα δεν έχουν δικούς τους Πίνακες Σελίδων, αλλά χρησιμοποιούν τους Πίνακες Σελίδων της διεργασίας που έτρεχε πριν από αυτά στον επεξεργαστή. Παρόλα αυτά, δεν υπάρχει λόγος να καταργήσουμε εγγραφές TLB που αφορούν διευθύνσεις του χώρου χρήστη, αφού τα νήματα πυρήνα δεν θα προσπελάσουν ποτέ διευθύνσεις στο χώρο χρήστη.

Όταν κάποιος επεξεργαστής αρχίζει να εκτελεί ένα νήμα πυρήνα, ο πυρήνας τον θέτει σε κατάσταση *lazy TLB mode*. Όταν αυτός ο επεξεργαστής λάβει, μέσω ενός IPI, αίτημα να καταργήσει κάποιες εγγραφές TLB, δεν προχωράει άμεσα στην κατάργηση τους. Ωστόσο, θυμάται ότι η τρέχουσα διεργασία χρησιμοποιεί Πίνακες Σελίδων, για τους οποίους οι αντίστοιχες εγγραφές TLB είναι πλέον άκυρες. Εάν η επόμενη διεργασία που θα δρομολογηθεί στον επεξεργαστή χρησιμοποιεί διαφορετικούς πίνακες σελίδων, τότε απαιτείται έτσι κι αλλιώς το καθάρισμα του TLB και ο επεξεργαστής βγαίνει από την κατάσταση *lazy TLB mode*. Αν, όμως, η επόμενη διεργασία που θα δρομολογηθεί σε έναν επεξεργαστή που βρίσκεται σε *lazy TLB mode* χρησιμοποιεί τους ίδιους πίνακες σελίδων με το νήμα πυρήνα, που εκτελούταν πριν σε αυτόν, τότε πρέπει να πραγματοποιηθεί η κατάργηση των εγγραφών TLB, που είχε αναβληθεί προηγουμένως.

2.3 Μηχανισμοί Διαδιεργασιακής Επικοινωνίας στο Linux

Ο πυρήνας του Linux προσφέρει μία πληθώρα μηχανισμών διαδιεργασιακής επικοινωνίας και στο παρόν κεφάλαιο εξετάζουμε ορισμένους από αυτούς. Πριν προχωρήσουμε, όμως, στην περιγραφή των μηχανισμών IPC του Linux, θεωρούμε σημαντικό να δώσουμε μία περιγραφή των γενικών στρατηγικών που είναι διαθέσιμες στον πυρήνα ενός λειτουργικού συστήματος για τη μεταφορά δεδομένων ανάμεσα σε διεργασίες [OSD16].

Ο αριθμός των αντιγράφων δεδομένων (data copies) που απαιτούνται από οποιονδήποτε μηχανισμό IPC εξαρτάται από την υποκείμενη τεχνική μεταφοράς δεδομένων που αυτός χρησιμοποιεί. Διακρίνουμε τις εξής περιπτώσεις:

- Πολλαπλά Αντίγραφα (Multi-copy): "Kernel-relayed buffers" ή "Kernel-buffered".
- Ένα Αντίγραφο (Single-copy): "Map-shared and copy" ή "map and copy".
- Μηδενικά Αντίγραφα (Zero-copy): "Map-shared and read" ("map and read") και "Page-table displacement".

Kernel-buffered Σε αυτή την τεχνική, η διεργασία αποστολέας ξεκινάει μια συναλλαγή και δίνει στον πυρήνα τη διεύθυνση των δεδομένων, τα οποία επιθυμεί να μεταφερθούν σε άλλη διεργασία. Ο πυρήνας αντιγράφει τα δεδομένα αυτά στο δικό του χώρο διευθύνσεων, ειδοποιεί με κάποιον τρόπο τον παραλήπτη και επιστρέφει στον αποστολέα. Όταν ο παραλήπτης λάβει την "ειδοποίηση" ζητάει από τον πυρήνα να αντιγράψει τα δεδομένα σε μια δική του περιοχή μνήμης.

Αυτή η προσέγγιση δεν απαιτεί από τον πυρήνα να μπλοκάρει τον αποστολέα, αφού τυχούσες αλλαγές στα δεδομένα του αποστολέα δεν επηρεάζουν την ακεραιότητα των δεδομένων που θα λάβει ο παραλήπτης. Δεν απαιτείται η ενημέρωση του αποστολέα για την επιτυχή παραλαβή των δεδομένων από τον παραλήπτη, αλλά κάποια συγκεκριμένη υλοποίηση θα μπορούσε να επιλέξει να στείλει μια τέτοια επιβεβαίωση στα πλαίσια μιας "χειραψίας" μεταξύ των δύο διεργασιών.

Όπως είναι φανερό, αυτή η τεχνική απαιτεί δύο αντίγραφα των μεταδιδόμενων δεδομένων, δύο πιθανές εναλλαγές του χώρου διευθύνσεων (μετάβαση στο χώρο διευθύνσεων του παραλήπτη και μετά πίσω στο χώρο διευθύνσεων του αποστολέα) και έναν

αριθμό από μεταβάσεις ανάμεσα στο χώρο χρήστη και στο χώρο πυρήνα, ανάλογα με τη συγκεκριμένη υλοποίηση.

Map and copy Αυτή η τεχνική εκμεταλλεύεται τη δυνατότητα απεικόνισης μνήμης (memory mapping) του συστήματος εικονικής μνήμης (βλέπε 2.2.1) προκειμένου να μειώσει τον αριθμό των αντιγράφων.

Εν συντομία, ο αποστολέας ζητάει από τον πυρήνα να στείλει ένα πακέτο δεδομένων στον παραλήπτη. Ο πυρήνας πρέπει στο σημείο αυτό να μπλοκάρει τον αποστολέα (συνήθως βάζοντας τον να κοιμηθεί) και στη συνέχεια ειδοποιεί τον παραλήπτη.

Ο αποστολέας πρέπει να μπλοκάρει, διότι ο πυρήνας δε δημιουργεί αντίγραφο των μεταδιδόμενων δεδομένων και, ως εκ τούτου, αν επέτρεπε στον αποστολέα να συνεχίσει την εκτέλεση του, θα ήταν δυνατό να αλλοιωθούν τα μεταδιδόμενα δεδομένα πριν τα διαβάσει ο παραλήπτης. Ωστόσο, αυτό δεν είναι πάντα αρκετό. Ας θεωρήσουμε, για παράδειγμα, την περίπτωση μιας πολύ-νηματικής διεργασίας. Το μπλοκάρισμα του νήματος - αποστολέα δεν είναι αρκετό, διότι ένα άλλο νήμα θα μπορούσε να αλλοιώσει τα δεδομένα πριν προλάβει να τα αντιγράψει ο παραλήπτης. Προφανώς, το μπλοκάρισμα όλων των νημάτων της διεργασίας - αποστολέα, για την αποστολή ενός μηνύματος, δεν είναι ικανοποιητική λύση για τις περισσότερες εφαρμογές. Επομένως, μπορεί να απαιτηθεί κάποιο πιο περίτεχνο σχήμα συγχρονισμού για να εξασφαλιστεί η σειριοποίηση των προσβάσεων στα μεταδιδόμενα δεδομένα.

Όταν ο παραλήπτης λάβει την "ειδοποίηση" ότι υπάρχουν διαθέσιμα δεδομένα γι αυτόν ζητάει από τον πυρήνα να τα αντιγράψει σε δική του μνήμη. Μόλις ο πυρήνας ολοκληρώσει την αντιγραφή θα ξυπνήσει τον αποστολέα.

Σε σχέση με την προηγούμενη τεχνική (Kernel-buffered) έχει τα εξής πλεονεκτήματα:

- Η μεταφορά μεγάλων ποσοτήτων δεδομένων δεν εξαντλεί το χώρο εικονικών διευθύνσεων του πυρήνα.
- Η μεταφορά των δεδομένων γίνεται απευθείας από τον αποστολέα στον παραλήπτη, χωρίς τη δημιουργία ενός ενδιάμεσου αντιγράφου στον πυρήνα.
- Για μεταφορά μεγάλων ποσοτήτων δεδομένων, η μείωση των αντιγράφων από τα δύο στο ένα και η εξάλειψη της ανάγκης για δέσμευση μνήμης στον πυρήνα έχουν ως αποτέλεσμα καλύτερες επιδόσεις.

Map and read Αυτή η τεχνική μπορεί να χρησιμοποιηθεί μόνο εάν ο αποστολέας μπορεί να εγγυηθεί ότι δε θα γράψει στα δεδομένα που θέλει να στείλει μέχρι ο παραλήπτης να τα διαβάσει και να σταματήσει να τα χρησιμοποιεί.

Το πλεονέκτημα αυτής της προσέγγισης έναντι της προηγούμενης (map and copy) είναι ότι δεν απαιτούνται αντίγραφα των δεδομένων. Τα δεδομένα απεικονίζονται από το χώρο διευθύνσεων του αποστολέα στο χώρο διευθύνσεων του παραλήπτη και ο τελευταίος μπορεί να τα χρησιμοποιήσει απευθείας. Τα βήματα, σε αυτή τη τεχνική, είναι αντίστοιχα με την προηγούμενη:

Ο αποστολέας ετοιμάζει ένα πακέτο δεδομένων μόνο για ανάγνωση και ζητάει από τον πυρήνα να τα μεταφέρει στον παραλήπτη. Ο πυρήνας "κοιμίζει" τον αποστολέα και ειδοποιεί τον παραλήπτη. Μόλις ο παραλήπτης λάβει την ειδοποίηση ότι υπάρχουν διαθέσιμα δεδομένα γι αυτόν, ζητάει από τον πυρήνα να κατασκευάσει μια απεικόνιση, γι αυτά τα δεδομένα, στο χώρο διευθύνσεων του. Ο πυρήνας κατασκευάζει την απεικόνιση και στη συνέχεια ο παραλήπτης χρησιμοποιεί τα δεδομένα, λαμβάνοντας υπόψη ότι είναι μόνο για ανάγνωση και δεν πρέπει να τα αλλοιώσει. Όταν ο παραλήπτης τελειώσει πρέπει να ειδοποιήσει τον πυρήνα, ο οποίος καταργεί την απεικόνιση από το χώρο διευθύνσεων του παραλήπτη και ξυπνάει τον αποστολέα.

Page-table displacement Αυτή η τεχνική δεν αντιγράφει αλλά μεταφέρει τα δεδομένα από τη μία διεργασία στην άλλη. Πιο συγκεκριμένα μεταφέρει τα δεδομένα από την εμβέλεια της μιας διεργασίας στην εμβέλεια μιας άλλης, καταργώντας τις αντίστοιχες απεικονίσεις από το χώρο διευθύνσεων του αποστολέα (unmapping) και φτιάχνοντας νέες απεικονίσεις για τα δεδομένα στο χώρο διευθύνσεων του παραλήπτη.

Δε δημιουργείται κανένα αντίγραφο των δεδομένων και αλλάζουν μόνο οι Πίνακες Σελίδων των δύο διεργασιών, κάνοντας αυτή τη μέθοδο τόσο γρήγορη, όσο ο χρόνος που απαιτείται για να αλλάξουν οι Πίνακες Σελίδων των εμπλεκόμενων διεργασιών και να καταργηθούν οι παλιές απεικονίσεις από τη μονάδα διαχείρισης μνήμης (βλέπε TLB flushing κεφάλαιο 2.2.9).

Εν ολίγοις, οι περισσότεροι μηχανισμοί IPC υλοποιούν κάποια από τις προαναφερθείσες τεχνικές ή παραλλαγές αυτών. Για παράδειγμα, οι τεχνικές map-and-copy και map-and-read απαγορεύουν στον αποστολέα να αλλάξει τα δεδομένα του μηνύματος ενώ

αυτό μεταδίδεται. Ωστόσο, υπάρχουν πυρήνες που επιτρέπουν αυτή τη συμπεριφορά και τη θεωρούν επιθυμητή (πχ. ο πυρήνας του Linux).

Ο πυρήνας του Linux υλοποιεί ένα υποσύνολο των προηγούμενων τεχνικών καθώς και παραλλαγές αυτών.

Οι κυριότεροι μηχανισμοί διαδιεργασιακής επικοινωνίας που παρέχει ο πυρήνας του Linux είναι:

- *Σήματα (Signals)*: Χρησιμοποιούνται για να ειδοποιήσουν μια διεργασία ότι συνέβη ένα γεγονός (event).

- *Σωληνώσεις (Anonymous Pipes / Named Pipes)*: Είναι μονόδρομα κανάλια επικοινωνίας που μεταφέρουν δεδομένα ανάμεσα σε δύο ή περισσότερες διεργασίες.

- *Unix Sockets*: Είναι αμφίδρομα κανάλια επικοινωνίας που μεταφέρουν δεδομένα ανάμεσα σε δύο ή περισσότερες διεργασίες.

- *Ουρές μηνυμάτων (Message queues)*: Πρόκειται για ένα ασύγχρονο πρωτόκολλο επικοινωνίας που χρησιμοποιείται για την ανταλλαγή μηνυμάτων ανάμεσα σε διεργασίες.

- *Σημαφόροι (Semaphores)*: Μια απλή δομή που χρησιμοποιείται κυρίως για τον έλεγχο πρόσβασης σε μοιραζόμενους πόρους.

- *Μοιραζόμενη Μνήμη (Shared memory)*: Πολλαπλές διεργασίες αποκτούν πρόσβαση στο ίδιο "κομμάτι" μνήμης, το οποίο μπορούν να χρησιμοποιήσουν για να ανταλλάξουν δεδομένα.

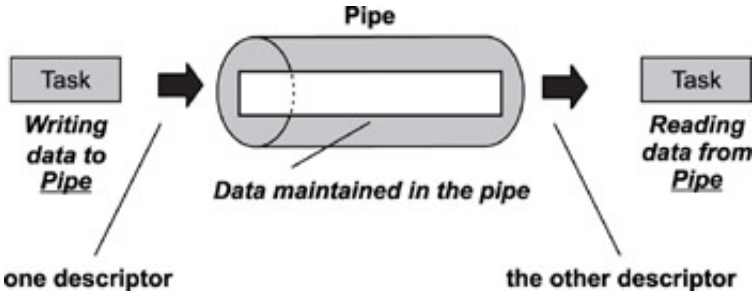
- *vmsplice*: Μηχανισμός βασισμένος στο pipe που δημιουργεί ένα αντί για δύο αντίγραφα των μεταδιδόμενων δεδομένων.

Στη συνέχεια αναλύουμε τους μηχανισμούς που μας ενδιαφέρουν στην παρούσα εργασία.

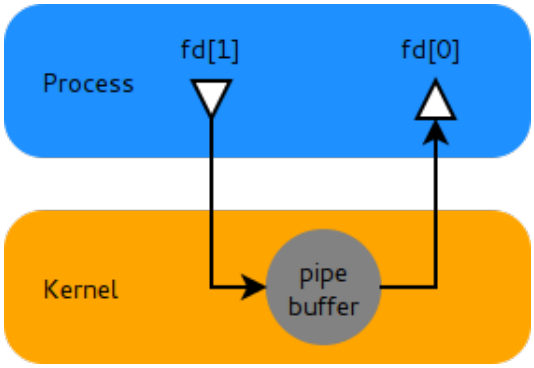
2.3.1 Σωληνώσεις (Pipes)

Ανώνυμες Σωληνώσεις (Anonymous pipes)

Ένα pipe παρέχει ένα μονόδρομο κανάλι επικοινωνίας (half duplex) ανάμεσα σε συγγενικές διεργασίες (related processes). Το pipe έχει ένα άκρο ανάγνωσης και ένα άκρο εγγραφής. Τα δεδομένα που γράφονται στο άκρο εγγραφής του pipe μπορούν να διαβαστούν από το άκρο ανάγνωσης. Όταν δημιουργείται ένα καινούριο pipe (με την κλήση συστήματος pipe), ο πυρήνας επιστρέφει δύο περιγραφείς αρχείου (file descriptors). Ο ένας χρησιμοποιείται για πρόσβαση στο άκρο εγγραφής του pipe και ο άλλος για πρόσβαση στο άκρο ανάγνωσης (Σχήματα 2.10 και 2.11). Το κανάλι επικοινωνίας που παρέχεται από το pipe είναι μια αδόμητη ροή από bytes (byte stream) και τα δεδομένα διαβάζονται από αυτό σε διάταξη FIFO (First In First Out).



Σχήμα 2.10: *Ένα pipe*



Σχήμα 2.11: *Pipe και χώρος χρήστη/πυρήνα*

Αν μία διεργασία προσπαθήσει να διαβάσει από ένα άδειο pipe, τότε η κλήση συστήματος read θα μπλοκάρει μέχρι να υπάρξουν διαθέσιμα δεδομένα. Αν μία διεγασία προσπαθήσει να γράψει σε ένα γεμάτο pipe, τότε η κλήση συστήματος write θα μπλοκάρει μέχρι να ελευθερωθεί επαρκής χώρος για να μπορέσει να πραγματοποιηθεί η εγγραφή. Είναι δυνατόν να πραγματοποιηθεί Nonblocking E/E (βλέπε στα επόμενα).

Αν όλοι οι file descriptors που αναφέρονται στο άκρο εγγραφής του pipe κλείσουν, τότε μια απόπειρα ανάγνωσης από το pipe θα δει end-of-file (η κλήση συστήματος read θα επιστρέψει 0). Αν όλοι οι file descriptors που αναφέρονται στο άκρο ανάγνωσης του pipe κλείσουν, τότε μια απόπειρα εγγραφής στο pipe θα έχει ως αποτέλεσμα να σταλεί στη διεργασία το σήμα (signal) SIGPIPE. Αν η διεργασία αγνοεί αυτό το σήμα, τότε η κλήση συστήματος write θα αποτύχει με κωδικό σφάλματος EPIPE.

Το pipe έχει περιορισμένη χωρητικότητα, η οποία εξαρτάται από την εκάστοτε υλοποίηση. Στο Linux η χωρητικότητα αυτή είναι 65536 bytes και μπορεί να αλλάξει με κατάλληλες κλήσεις συστήματος. Το πρότυπο POSIX.1-2001 ορίζει ότι εγγραφές μικρότερες από PIPE_BUF bytes πρέπει να είναι ατομικές, δηλαδή τα δεδομένα γράφονται στο pipe ως μια συνεχή ακολουθία. Εγγραφές μεγαλύτερες από PIPE_BUF bytes μπορεί να είναι μη ατομικές: ο πυρήνας μπορεί να παρεμβάλλει τα δεδομένα που γράφει η διεργασία με δεδομένα που γράφουν άλλες διεργασίες. Το POSIX.1-2001 απαιτεί το PIPE_BUF να είναι τουλάχιστον 512 bytes (στο Linux είναι 4096 bytes). Η ακριβής συμπεριφορά εξαρτάται από το αν ο file descriptor είναι nonblocking (O_NONBLOCK), από το αν υπάρχουν πολλαπλοί εγγραφείς στο pipe και από τον αριθμό n των bytes που γράφονται:

- Blocking συμπεριφορά, n <= PIPE_BUF
  Όλα τα n bytes γράφονται ατομικά. Η κλήση συστήματος write μπορεί να μπλοκάρει, αν δεν υπάρχει ελεύθερος χώρος για να γραφτούν άμεσα τα n bytes.

- Blocking συμπεριφορά, n > PIPE_BUF
  Τα δεδομένα που δίνονται στην κλήση συστήματος write μπορούν να παρεμβληθούν με δεδομένα από άλλες διεργασίες. Η write μπλοκάρει μέχρι να γραφτούν και τα n bytes.

- Nonblocking συμπεριφορά, n <= PIPE_BUF
  Αν υπάρχει ελεύθερος χώρος στο pipe για να γραφούν n bytes, τότε η κλήση συστήματος write επιτυγχάνει άμεσα γράφοντας όλα τα n bytes. Διαφορετικά, η write αποτυγχάνει με τον κωδικό λάθους EAGAIN (προσπαθήστε ξανά).

- Nonblocking συμπεριφορά, n > PIPE_BUF
  Αν το pipe είναι γεμάτο τότε η κλήση συστήματος write αποτυγχάνει με τον κωδικό λάθους EAGAIN. Διαφορετικά, από 1 μέχρι n bytes μπορεί να γραφτούν

στο pipe (με άλλα λόγια μπορεί να συμβεί μια μερική εγγραφή και η διεργασία πρέπει να ελέγξει τη τιμή που επιστρέφει η write για να διαπιστώσει πόσα bytes γράφτηκαν τελικά στο pipe), τα οποία μπορεί να παρεμβληθούν με δεδομένα άλλων διεργασιών που γράφουν παράλληλα στο pipe.

Οι σωληνώσεις (pipes) βασίζονται στην πρώτη από τις τεχνικές μεταφοράς δεδομένων που εξετάσαμε νωρίτερα (Kernel-buffered - 2.3). Αυτό σημαίνει ότι κατά τη μετάδοση δεδομένων από τον αποστολέα στον παραλήπτη δημιουργούνται δύο αντίγραφα. Ένα από μνήμη του αποστολέα σε μνήμη του πυρήνα (στο pipe) και ένα από τη μνήμη του πυρήνα (pipe) σε μνήμη του παραλήπτη.

Επώνυμες Σωληνώσεις (Named Pipes ή FIFOs)

Πρόκειται για ειδικά αρχεία που προσπελάζονται μέσω του συστήματος αρχείων. Μοιάζουν με τις σωληνώσεις και μπορούν να ανοιχτούν από πολλαπλές διεργασίες για ανάγνωση ή εγγραφή. Όταν δύο ή περισσότερες διεργασίες ανταλλάσσουν δεδομένα μέσω ενός αρχείου FIFO, ο πυρήνας μεταφέρει όλα τα δεδομένα εσωτερικά χωρίς να τα γράφει στο σύστημα αρχείων. Επομένως, ένα αρχείο FIFO δεν έχει περιεχόμενα στο σύστημα αρχείων και απλά εξυπηρετεί ως σημείο αναφοράς, ούτως ώστε οι διεργασίες να μπορούν να αποκτήσουν πρόσβαση σε μία σωλήνωση (pipe), χρησιμοποιώντας ένα όνομα στο σύστημα αρχείων. Τα δικαιώματα πρόσβασης των αρχείων και των καταλόγων περιορίζουν ποιες διεργασίες μπορούν να ανοίξουν ένα αρχείο τύπου FIFO και, επομένως, περιορίζουν ποιες διεργασίες μπορούν να επικοινωνήσουν μεταξύ τους.

Η μόνη διαφορά ανάμεσα στις ανώνυμες και στις επώνυμες σωληνώσεις είναι ο τρόπος που δημιουργούνται και ανοίγονται. Μόλις αυτές οι εργασίες ολοκληρωθούν, οι λειτουργίες Ε/Ε στα pipes και στα FIFOs έχουν ακριβώς την ίδια σημασιολογία.

Ο πυρήνας διατηρεί ακριβώς ένα pipe για κάθε αρχείο FIFO, που έχει ανοιχτεί από τουλάχιστον μία διεργασία. Ένα αρχείο FIFO πρέπει να ανοιχτεί τόσο για ανάγνωση, όσο και για εγγραφή για να είναι εφικτή η μεταφορά δεδομένων μέσω αυτού. Συνήθως, το άνοιγμα του αρχείου, θα μπλοκάρει τη διεργασία μέχρι να ανοιχτεί και το άλλο άκρο (εγγραφής ή ανάγνωσης).

Μια διεργασία μπορεί να ανοίξει ένα αρχείο FIFO σε κατάσταση nonblocking. Σε αυτή την περίπτωση, το άνοιγμα του αρχείου για ανάγνωση θα επιτύχει, ακόμα κι αν δεν έχει

ανοίξει κανείς το αρχείο για εγγραφή. Αντίθετα, το άνοιγμα του αρχείου για εγγραφή θα αποτύχει με κωδικό λάθους ENXIO (no such device or address), εκτός κι αν το άκρο ανάγνωσης έχει ήδη ανοίξει.

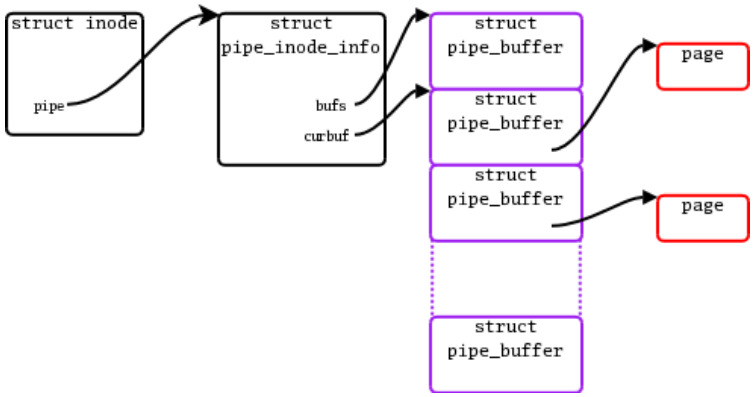
Στο Linux, το άνοιγμα ενός αρχείου FIFO για ανάγνωση **και** εγγραφή πετυχαίνει πάντα, τόσο σε κατάσταση blocking, όσο και σε κατάσταση nonblocking. Αυτή η συμπεριφορά επιτρέπει σε μια διεργασία να γράψει στο pipe, παρόλο που δεν υπάρχουν ακόμα αναγνώστες, καθώς και να επικοινωνήσει με τον εαυτό της.

Υλοποίηση σωληνώσεων στον πυρήνα του Linux

Για κάθε σωλήνωση (ανώνυμη ή επώνυμη) ο πυρήνας του Linux δημιουργεί ένα αντικείμενο τύπου *struct inode*, το οποίο περιέχει ένα δείκτη σε μια δομή τύπου *struct pipe_inode_info* που αναπαριστά το pipe. Για τις ανώνυμες σωληνώσεις το αντικείμενο inode δημιουργείται κατά την κλήση της κλήσης συστήματος pipe και προσαρτάται στο εικονικό σύστημα αρχείων *pipefs*. Επειδή το *pipefs* δεν προσαρτάται σε κάποιον κατάλογο του συστήματος, οι χρήστες δεν το βλέπουν ποτέ. Για τις επώνυμες σωληνώσεις το αντικείμενο inode δημιουργείται κατά τη δημιουργία του ειδικού αρχείου FIFO και προσαρτάται σε κάποιον κατάλογο του συστήματος. Έτσι, είναι προσβάσιμο από το σύστημα αρχείων.

Για τις ανώνυμες σωληνώσεις δημιουργούνται αυτόματα, από τον πυρήνα και δύο αντικείμενα τύπου *struct file*, το ένα χρησιμοποιείται για ανάγνωση από το pipe και το άλλο για εγγραφή στο pipe. Η διεργασία πρέπει να χρησιμοποιήσει τον κατάλληλο file descriptor, από αυτούς που επιστρέφονται από την κλήση συστήματος pipe, για να αποκτήσει πρόσβαση στο αντίστοιχο αντικείμενο τύπου *struct file*. Στην περίπτωση των επώνυμων σωληνώσεων, το άνοιγμα του αρχείου FIFO για ανάγνωση ή/και εγγραφή έχει ως άμεσο αποτέλεσμα τη δημιουργία του αντίστοιχου αντικειμένου τύπου *struct file*.

Στην καρδιά της δομής τύπου *struct pipe_inode_info* βρίσκεται ένα σύνολο από pipe buffers. Κάθε pipe buffer περιέχει έναν δείκτη σε ένα πλαίσιο μνήμης, το οποίο περιέχει τα δεδομένα που έχουν γραφτεί στο pipe και είναι διαθέσιμα προς ανάγνωση. Τα pipe buffers είναι οργανωμένα σε έναν κυκλικό buffer, όπως φαίνεται απλοϊκά στο Σχήμα 2.12 [Cor05].



Σχήμα 2.12: *Σωλήνωση (pipe) στον πυρήνα του Linux*

2.3.2 Unix Sockets

Τα Unix Sockets παρέχουν έναν μηχανισμό διαδιεργασιακής επικοινωνίας βασισμένο στο μοντέλο πελάτη-εξυπηρετητή (Client - Server Model). Υπάρχουν τρεις τύποι unix socket, που καθορίζουν τον τρόπο επικοινωνίας των διεργασιών:

- *SOCK_STREAM*: Παρέχει μια σειριακή, αξιόπιστη, αμφίδρομη, συνδεσμοστρεφή ροή από byte.
- *SOCK_DGRAM*: Παρέχει υποστήριξη για δεδομενογράμματα: ασυνδεσμικά, αναξιόπιστα μηνύματα με καθορισμένο μέγιστο μέγεθος (Στις περισσότερες υλοποιήσεις, ωστόσο, η μετάδοση των δεδομενογραμμάτων είναι αξιόπιστη και διατηρείται η σειρά μετάδοσης τους).
- *SOCK_SEQPACKET*: Παρέχει ένα σειριακό, αξιόπιστο, αμφίδρομο, συνδεσμοστρεφή κανάλι επικοινωνίας για δεδομενογράμματα καθορισμένου μέγιστου μεγέθους.

Στην περίπτωση των συνδεσμοστρεφών (connection oriented) sockets (*SOCK_STREAM* και *SOCK_SEQPACKET*) η διεργασία server και η διεργασία client ακολουθούν διαφορετικά βήματα για τη μεταξύ τους επικοινωνία.

Ο server πρέπει να ακολουθήσει τα εξής βήματα:

1. Δημιουργεί ένα socket με την κλήση συστήματος socket. Η τελευταία επιστρέφει έναν file descriptor που δίνει πρόσβαση στο socket.

2. Συσχετίζει αυτό το socket με μία διεύθυνση (βλέπε στα επόμενα) μέσω της κλήσης συστήματος bind.

3. Ακούει για εισερχόμενες συνδέσεις και τις αποδέχεται μέσω των κλήσεων συστήματος listen και accept.

Αξίζει να σημειωθεί ότι για κάθε εισερχόμενη σύνδεση, που αποδέχεται ο server, δημιουργείται ένα καινούριο socket για την επικοινωνία με τον συγκεκριμένο client. Το αρχικό socket εξακολουθεί να ακούει για νέες συνδέσεις. Αυτό έχει ως αποτέλεσμα να υπάρχει ένα ξεχωριστό κανάλι επικοινωνίας για κάθε ζεύγος διεργασιών (client - server) που επικοινωνούν.

Ο client πρέπει να ακολουθήσει τα εξής βήματα:

1. Δημιουργεί ένα socket με την κλήση συστήματος socket. Η τελευταία επιστρέφει έναν file descriptor που δίνει πρόσβαση στο socket.

2. Συνδέει αυτό το socket με το socket του server, που βρίσκεται στη διεύθυνση που προσδιόρισε προηγουμένως ο server. Ο client πρέπει, προφανώς, να γνωρίζει ποια είναι αυτή η διεύθυνση για να μπορέσει να βρει τον server. Για το σκοπό αυτό χρησιμοποιεί την κλήση συστήματος connect.

Μόλις εγκαθιδρυθεί η σύνδεση ανάμεσα σε client και server, οι διεργασίες μπορούν να επικοινωνήσουν μεταξύ τους είτε με τις κλήσεις συστήματος write / read, είτε με τις ειδικές για sockets κλήσης συστήματος send - sendto - sendmsg / recv - recvfrom - recvmsg.

Στην περίπτωση των ασυνδεσμικών (connectionless) sockets (*SOCK_DGRAM*) τα βήματα και για τις δύο διεργασίες (client / server) είναι τα ίδια:

1. Δημιουργία socket με την κλήση συστήματος socket. Η τελευταία επιστρέφει έναν file descriptor που δίνει πρόσβαση στο socket.

2. Συσχετισμός αυτού του socket με μία διεύθυνση (βλέπε στα επόμενα) μέσω της κλήσης συστήματος bind. Αυτό το βήμα είναι προαιρετικό, εν γένει, για τον client. Ωστόσο, αν θέλουμε ο server να μπορεί να απαντήσει στον client, τότε απαιτείται και από τον client να συσχετίσει το socket του με μία διεύθυνση.

3. Χρήση των κλήσεων συστήματος sendto / sendmsg για αποστολή δεδομένων, προσδιορίζοντας ρητά τη διεύθυνση του αποστολέα, και των κλήσεων συστήματος recvfrom / recvmsg για την παραλαβή δεδομένων, από οποιονδήποτε αποστολέα, η διεύθυνση του οποίου παρέχεται από την αντίστοιχη κλήση συστήματος.

Ένα unix socket είναι επί της ουσίας ένα άκρο επικοινωνίας (communication endpoint) που χρησιμοποιείται από μία διεργασία για να επικοινωνήσει με μία άλλη. Τα unix sockets παρέχουν τρεις τρόπους διευθυνσιοδότησης, δηλαδή τρεις τρόπους με τους οποίους μια διεργασία μπορεί να βρει μια άλλη:

- Ένα όνομα στο σύστημα αρχείων: Πρόκειται για ένα ειδικό αρχείο τύπου socket προσβάσιμο από το σύστημα αρχείων. Τα δικαιώματα πρόσβασης των αρχείων και των καταλόγων περιορίζουν ποιες διεργασίες μπορούν να ανοίξουν ένα αρχείο τύπου socket και επομένως περιορίζουν ποιος μπορεί να επικοινωνήσει με ποιον. Επίσης, το αρχείο αυτό, όπως και στην περίπτωση των FIFOs (βλέπε 2.3.1), παραμένει στο σύστημα αρχείων ακόμη κι αν δεν υπάρχουν διεργασίες που το χρησιμοποιούν.

- Ανώνυμα sockets. Δημιουργούνται, συνήθως, από την κλήση συστήματος socketpair, η οποία επιστρέφει ένα ζεύγος συνδεδεμένων socket για επικοινωνία μεταξύ συγγενικών διεργασιών.

- Ένα όνομα σε ένα αφηρημένο χώρο ονομάτων, ανεξάρτητο από το σύστημα αρχείων: Πρόκειται για επέκταση του Linux. Παρέχει αντίστοιχη λειτουργικότητα με την πρώτη περίπτωση διευθυνσιοδότησης, με τις εξής διαφορές:

    1. Τα δικαιώματα πρόσβασης των αρχείων και καταλόγων δεν έχουν νόημα για τα αφηρημένα socket.
    2. Τα αφηρημένα socket, όταν σταματήσουν να χρησιμοποιούνται, αφαιρούνται αυτόματα από το χώρο ονομάτων.

Σε σχέση με τις σωληνώσεις (pipes), που εξετάστηκαν προηγουμένως (2.3.1), τα unix sockets έχουν τα εξής πλεονεκτήματα:

- Παρέχουν ένα αμφίδρομο κανάλι επικοινωνίας (full duplex), ενώ οι σωληνώσεις παρέχουν ένα μονόδρομο κανάλι επικοινωνίας (half duplex).

- Διατηρούν ένα διακριτό κανάλι επικοινωνίας για κάθε ζεύγος διεργασιών που επικοινωνεί. Στην περίπτωση των σωληνώσεων, αν και πολλές διεργασίες μπορούν να γράψουν στη σωλήνωση, ο server δεν μπορεί να ξεχωρίσει από ποιον προέρχονται τα εκάστοτε δεδομένα. Επίσης, αν μία διεργασία γράψει περισσότερα από *PIPE_BUF* bytes στο pipe, τότε τα bytes αυτά μπορούν να παρεμβληθούν με bytes προερχόμενα από άλλες διεργασίες που γράφουν ταυτόχρονα στο pipe. Αν απαιτείται, επομένως, η διάκριση των διεργασιών - clients μεταξύ τους ή/και υπάρχει το ενδεχόμενο της αποστολής μεγάλων μηνυμάτων, τότε πρέπει να χρησιμοποιηθούν unix sockets αντί για pipes.

- Δίνουν τη δυνατότητα στη διεργασία - παραλήπτη να δει (peek) εισερχόμενα δεδομένα χωρίς αυτά να αφαιρεθούν από την ουρά των εισερχόμενων μηνυμάτων.

Επιπρόσθετα, τα unix sockets παρέχουν δύο ακόμη ενδιαφέρουσες δυνατότητες:

1. Δυνατότητα αποστολής ανοικτών file descriptors από μία διεργασία σε οποιαδήποτε άλλη. Με τον τρόπο αυτό, η διεργασία παραλήπτης μπορεί να αποκτήσει πρόσβαση σε πόρους που, υπό κανονικές συνθήκες, δεν θα μπορούσε να προσπελάσει.

2. Δυνατότητα αποστολής των διαπιστευτηρίων μιας διεργασίας (process credentials) σε μια άλλη, κάτω από τον έλεγχο του πυρήνα. Αυτό επιτρέπει, για παράδειγμα, σε έναν server να πιστοποιήσει ποιος επικοινωνεί μαζί του, χωρίς να χρειάζεται κάποιο πιο περίπλοκο σχήμα πιστοποίησης.

Υλοποίηση unix sockets στον πυρήνα του Linux

Στην ενότητα αυτή δίνουμε μια σύντομη περιγραφή για το πώς ο πυρήνας του Linux υλοποιεί τα unix sockets. Κάθε unix socket έχει μια ουρά εισερχόμενων μηνυμάτων. Κάθε μήνυμα αναπαρίσταται στον πυρήνα από μία δομή τύπου *struct sk_buff*. Στην περίπτωση της αποστολής δεδομενογραμμάτων (socket τύπου *SOCK_DGRAM* ή

SOCK_SEQPACKET) κάθε δεδομενόγραμμα αποτελεί ξεχωριστό μήνυμα, με παραμετροποιήσιμο μέγιστο μέγεθος. Στην περίπτωση ροής από byte (byte stream - socket τύπου *SOCK_STREAM*) η ροή των bytes χωρίζεται σε πολλαπλά μηνύματα και αυτά είναι που τελικά αποστέλλονται.

Κατά την αποστολή ενός μηνύματος από ένα unix socket σε ένα άλλο, η διεργασία - αποστολέας τοποθετεί το μήνυμα απευθείας στην ουρά εισερχομένων μηνυμάτων του socket της διεργασίας παραλήπτη. Για τη λήψη ενός μηνύματος η διεργασία - παραλήπτης ελέγχει την ουρά εισερχομένων μηνυμάτων του δικού της socket για να δει αν υπάρχουν διαθέσιμα μηνύματα προς ανάγνωση.

Τα unix sockets βασίζονται στην πρώτη από τις τεχνικές μεταφοράς δεδομένων που εξετάσαμε νωρίτερα (Kernel-buffered - 2.3). Αυτό σημαίνει ότι κατά τη μετάδοση δεδομένων από τον αποστολέα στον παραλήπτη δημιουργούνται δύο αντίγραφα. Ένα από μνήμη του αποστολέα σε μνήμη του πυρήνα (ουρά εισερχομένων μηνυμάτων του socket του παραλήπτη) και ένα από τη μνήμη του πυρήνα (socket) σε μνήμη του παραλήπτη.

2.3.3 vmsplice

Το Linux παρέχει την κλήση συστήματος vmsplice, η οποία μπορεί να χρησιμοποιηθεί για διαδιεργασιακή επικοινωνία ενός αντιγράφου. Το πρότυπο της συνάρτησης vmsplice είναι το εξής:

*ssize_t vmsplice(int fd, const struct iovec *iov, unsigned long nr_segs, unsigned int flags);*

Το vmsplice απεικονίζει *nr_segs* περιοχές εικονικής μνήμης της διεργασίας - αποστολέα, που περιγράφονται από τη δομή *iov*, σε ένα pipe, που προσδιορίζεται από τον file descriptor *fd*.

Ο υποκείμενος μηχανισμός μεταφοράς είναι το pipe, επομένως ισχύουν όλα όσα αναφέρθηκαν για τις σωληνώσεις στο κεφάλαιο 2.3.1. Για να πετύχει, όμως, η διεργασία - αποστολέας μεταφορά των δεδομένων με ένα μόνο αντίγραφο, πρέπει να χρησιμοποιήσει την κλήση συστήματος vmsplice στη θέση της write για την “εγγραφή” των δεδομένων στο άκρο εγγραφής του pipe. Ο παραλήπτης εξακολουθεί να χρησιμοποιεί την κλήση συστήματος read για την αντιγραφή των δεδομένων από το άκρο ανάγνωσης του pipe στο χώρο διευθύνσεων του.

Όπως είδαμε στο σχήμα 2.12 του κεφαλαίου 2.3.1, ένα pipe υλοποιείται ως ένα σύνολο από δείκτες σε πλαίσια φυσικής μνήμης. Όταν μια διεργασία στέλνει δεδομένα μέσω ενός pipe, χρησιμοποιώντας την κλήση συστήματος write, ο πυρήνας αντιγράφει τα δεδομένα σε καινούρια πλαίσια μνήμης, που δεσμεύονται εκείνη τη στιγμή και δείκτες σε αυτά τα πλαίσια τοποθετούνται στους pipe buffers του pipe. Όταν χρησιμοποιούμε την κλήση συστήματος vmsplice, τα δεδομένα δεν αντιγράφονται σε νέα πλαίσια μνήμης, αλλά τα pipe buffers του pipe αναφέρονται στα πλαίσια μνήμης του αποστολέα, που περιέχουν τα δεδομένα προς αποστολή, και αυξάνονται οι αντίστοιχοι μετρητές αναφορών των πλαισίων αυτών (βλέπε πεδίο *count* κεφάλαιο 2.2.4), για να δηλώσουν το γεγονός ότι το pipe αναφέρεται σε αυτά. Άρα, αυτό που τελικά αντιγράφεται είναι δείκτες προς τα κατάλληλα πλαίσια μνήμης (συγκεκριμένα δείκτες προς τις δομές *struct page* που περιγράφουν τα πλαίσια αυτά) και όχι τα ίδια τα δεδομένα που περιέχονται σε αυτά. Όταν η διεργασία - παραλήπτης καλέσει την κλήση συστήματος read τα δεδομένα αντιγράφονται από το χώρο διευθύνσεων του αποστολέα απευθείας στο χώρο διευθύνσεων του παραλήπτη.

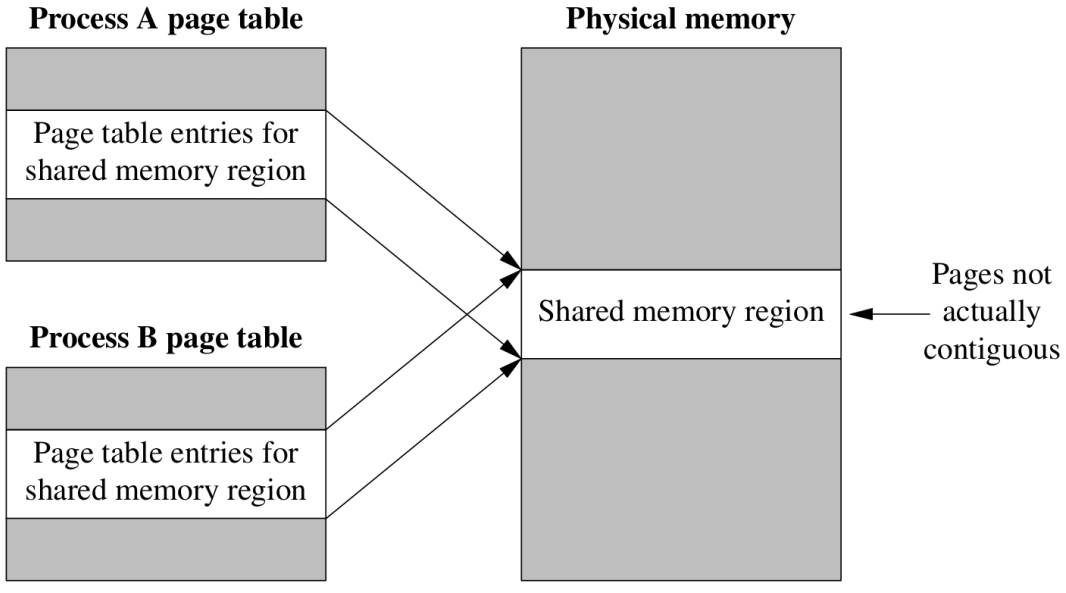
Ένα σημαντικό μειονέκτημα του vmsplice είναι ότι καθιστά αδύνατον για τον αποστολέα να προσδιορίσει πότε οι σελίδες που απεικονίστηκαν στο pipe και περιέχουν τα δεδομένα προς αποστολή, έχουν καταναλωθεί από τον παραλήπτη και άρα είναι ασφαλές να χρησιμοποιήσει ξανά αυτή τη μνήμη. Αν ο αποστολέας γράψει στις αντίστοιχες σελίδες μνήμης, πριν προλάβει να τις διαβάσει ο παραλήπτης, τότε θα αλλοιώσει το μήνυμα που είχε στείλει προηγουμένως, αφού ο παραλήπτης θα δει αυτές τις αλλαγές.

Η μετάδοση δεδομένων από τον αποστολέα στον παραλήπτη με χρήση του vmsplice αποτελεί παραλλαγή της τεχνικής *map and copy*, που είδαμε στο κεφάλαιο 2.3.

Σημείωση: Η κλήση συστήματος vmsplice είναι μέλος μιας οικογένειας τριών κλήσεων συστήματος του Linux, των splice, vmsplice, tee. Αυτές οι κλήσεις συστήματος παρέχουν στα προγράμματα χρήστη πλήρη έλεγχο ενός buffer που βρίσκεται στο χώρο πυρήνα, ο οποίος υλοποιείται χρησιμοποιώντας τον ίδιο τύπο buffer που χρησιμοποιείται για τις σωληνώσεις. Η χρήση αυτών των κλήσεων συστήματος δεν περιορίζεται στη διαδιεργασιακή επικοινωνία αλλά επεκτείνεται σε λειτουργίες Ε/Ε με αρχεία, sockets, κτλ. Ωστόσο, μια λεπτομερέστερη περιγραφή αυτών των κλήσεων συστήματος και της χρήσης τους ξεφεύγει από το αντικείμενο της παρούσας εργασίας.

2.3.4 Μοιραζόμενη μνήμη (Shared Memory)

Σε αυτόν τον μηχανισμό διαδιεργασιακής επικοινωνίας οι διεργασίες ανταλλάσσουν δεδομένα μέσω μιας περιοχής μνήμης, η οποία απεικονίζεται στο χώρο εικονικών διευθύνσεων τους, όπως φαίνεται στο σχήμα 2.13. Αυτός ο μηχανισμός εκμεταλλεύεται τη δυνατότητα απεικόνισης μνήμης (memory mapping) του συστήματος εικονικής μνήμης (βλέπε 2.2.1). Σε αντίθεση με τους προηγούμενους μηχανισμούς, στην περίπτωση αυτή, ο πυρήνας δε μεσολαβεί για τη μεταφορά των δεδομένων. Μόλις μια διεργασία γράψει κάποια δεδομένα στο κομμάτι μοιραζόμενης μνήμης αυτά γίνονται άμεσα διαθέσιμα σε όλες τις διεργασίες που το απεικονίζουν στο χώρο διευθύνσεων τους. Το γεγονός αυτό κάνει τη μοιραζόμενη μνήμη γρηγορότερη από τους άλλους μηχανισμούς και είναι μάλιστα ο γρηγορότερος τρόπος ανταλλαγής δεδομένων ανάμεσα σε διεργασίες που εκτελούνται στο ίδιο μηχάνημα. Ωστόσο, η μοιραζόμενη μνήμη αποτελεί πόρο που τον μοιράζονται παράλληλα εκτελούμενες διεργασίες και επομένως υπάρχει ανάγκη συγχρονισμού των προσβάσεων σε αυτή, προκειμένου να αποτρέψουμε τις ταυτόχρονες ενημερώσεις ή/και αναγνώσεις δεδομένων, των οποίων δεν έχει ολοκληρωθεί ακόμη η εγγραφή.



Σχήμα 2.13: *Μοιραζόμενη Μνήμη (Shared Memory)*

Το Linux παρέχει διάφορες προγραμματιστικές διεπαφές (APIs) για τη μοιραζόμενη μνήμη:

- *System V shared memory*
  - Παλαιότερος μηχανισμός μοιραζόμενης μνήμης.
  - Διαμοιρασμός ανάμεσα σε ασυσχέτιστες διεργασίες.
- *Shared mappings* - κλήση συστήματος mmap
  - *Shared anonymous mappings*
    Διαμοιρασμός ανάμεσα σε συγγενικές διεργασίες (που σχετίζονται μέσω της κλήσης συστήματος fork)
  - *Shared file mappings*
    Διαμοιρασμός ανάμεσα σε ασυσχέτιστες διεργασίες, που απεικονίζουν το ίδιο αρχείο του συστήματος αρχείων
- *POSIX shared memory*
  - Διαμοιρασμός ανάμεσα σε ασυσχέτιστες διεργασίες, χωρίς το κόστος των λειτουργιών Ε/Ε του συστήματος αρχείων.
  - Ανανεωμένο API με στόχο να αντικαταστήσει το παλαιότερο *System V shared memory* API.

Ένα αντικείμενο μοιραζόμενης μνήμης (κομμάτι μοιραζόμενης μνήμης), στην περίπτωση του POSIX shared memory API, υλοποιείται, στο Linux, ως ένα αρχείο σε ένα ειδικό σύστημα αρχείων που καλείται tmpfs και είναι προσαρτημένο στον κατάλογο */dev/shm*. Τα αρχεία του *tmpfs* δεν αντιστοιχούν σε πραγματικά αρχεία στον δίσκο, αλλά βρίσκονται εξ' ολοκλήρου σε πλαίσια φυσικής μνήμης. Ωστόσο, αν ο πυρήνας χρειαστεί να ανακτήσει μνήμη, οι σελίδες αυτές μπορούν να αποθηκευτούν στην περιοχή swap (swap area) στο δίσκο και αργότερα, αν και όταν χρειαστούν, να φορτωθούν πάλι στην κύρια μνήμη. Τα αντικείμενα μοιραζόμενης μνήμης εξακολουθούν να υπάρχουν μέχρι είτε να διαγραφούν ρητά, είτε να γίνει επανεκκίνηση του συστήματος. Για παράδειγμα, μια διεργασία μπορεί να απεικονίσει ένα τέτοιο αντικείμενο στο χώρο διευθύνσεων της, να αλλάξει τα περιεχόμενά του και έπειτα να καταργήσει την απεικόνιση. Οι αλλαγές αυτές θα είναι ορατές στην επόμενη διεργασία που θα απεικονίσει το αντικείμενο στο δικό της χώρο διευθύνσεων. Τέλος, τα δικαιώματα πρόσβασης των αρχείων και των καταλόγων περιορίζουν ποιες διεργασίες μπορούν να ανοίξουν και να απεικονίσουν ένα αρχείο

στον κατάλογο */dev/shm* και επομένως περιορίζουν ποιος μπορεί να επικοινωνήσει με ποιον.

Αυτός ο μηχανισμός IPC αποτελεί παραλλαγή της τεχνικής map and read, που είδαμε στην αρχή του κεφαλαίου (βλέπε 2.3), και πρόκειται για μηχανισμό μηδενικών αντιγράφων. Ωστόσο, όπως αναφέρθηκε και στα προηγούμενα, έχει το μειονέκτημα ότι απαιτεί από τις διεργασίες να υλοποιούν κάποιο ρητό σχήμα συγχρονισμού για την πρόσβαση στα μοιραζόμενα δεδομένα.

3

Αξιολόγηση μηχανισμών διαδιεργασιακής επικοινωνίας

Ένα σημαντικό και ουσιώδες βήμα, κατά την ανάπτυξη ενός μηχανισμού διαδιεργασιακής επικοινωνίας, είναι η αξιολόγηση των επιδόσεων του και η σύγκριση τους με τις επιδόσεις των ήδη διαθέσιμων μηχανισμών. Με τον τρόπο αυτό μπορούμε να αξιολογήσουμε αποτελεσματικά τη σχεδίαση και υλοποίηση του μηχανισμού που αναπτύσσουμε. Για να είναι, όμως, συγκρίσιμες οι μετρήσεις των επιδόσεων διαφορετικών μηχανισμών θα πρέπει να πραγματοποιηθούν με έναν ενιαίο και συνεπή τρόπο. Είναι, λοιπόν, φανερό ότι υπάρχει ανάγκη για ένα εργαλείο αξιολόγησης μηχανισμών IPC, το οποίο θα παρέχει ένα κοινό πλαίσιο αξιολόγησης και θα επιτρέπει την εύκολη και γρήγορη ανάπτυξη νέων benchmarks.

3.1 Το ipc-bench

Το ipc-bench είναι ένα τέτοιο εργαλείο, που έχει αναπτυχθεί από το εργαστήριο Υπολογιστικών Συστημάτων του Πανεπιστημίου του Cambridge [oCCL12]. Το ipc-bench δίνει τη δυνατότητα δημιουργίας benchmarks που βασίζονται στο μοντέλο παραγωγού - καταναλωτή και μετρούν είτε το ρυθμό μεταφοράς δεδομένων (throughput), είτε την καθυστέρηση μεταφοράς (ping - pong latency). Στη συνέχεια δίνουμε μια σύντομη περιγραφή του τρόπου λειτουργίας του ipc-bench και του πώς μπορούμε να υλοποιήσουμε νέα benchmarks σε αυτό.

Για κάθε benchmark το ipc-bench δημιουργεί ένα ζεύγος διεργασιών. Η μία διεργασία

είναι ο αποστολέας - παραγωγός και η άλλη διεργασία ο παραλήπτης - καταναλωτής. Πριν τη δημιουργία αυτών των διεργασιών εκτελείται μια συνάρτηση αρχικοποίησης του υποκείμενου μηχανισμού ipc (*init_test*). Στη συνέχεια κάθε διεργασία καρφιτσώνεται (pin) σε έναν επεξεργαστή και αρχίζει να εκτελείται.

Τα βήματα εκτέλεσης του αποστολέα είναι:

1. Εκτέλεση μιας συνάρτησης αρχικοποίησης του αποστολέα (*init_parent*).
2. Ξεκινάει ένα χρονόμετρο.
3. Εκτελεί τα παρακάτω για έναν, καθορισμένο από το χρήστη, αριθμό επαναλήψεων:
   - *Throughput Benchmark*
     (a) Ζητάει από τον υποκείμενο μηχανισμό ipc έναν buffer για να τοποθετήσει τα δεδομένα του μηνύματος προς αποστολή (*get_write_buffer*).
     (b) Αν έχει ενεργοποιηθεί η επιλογή *write-in-place* από το χρήστη παράγει απευθείας στον προηγούμενο buffer τα δεδομένα προς αποστολή. Διαφορετικά τα παράγει σε έναν ιδιωτικό buffer, τον οποίο στη συνέχεια αντιγράφει στον buffer που του δόθηκε στο προηγούμενο βήμα.
     (c) Ζητάει από τον υποκείμενο μηχανισμό ipc να στείλει το μήνυμα στον παραλήπτη (*release_write_buffer*).
   - *ping - pong Latency Benchmark*
     (a) Ζητάει από τον υποκείμενο μηχανισμό ipc να στείλει ένα μήνυμα (ping) στον παραλήπτη (*parent_ping*).
4. Εκτέλεση μιας συνάρτησης τερματισμού του αποστολέα (*finish_parent*).
5. Σταματάει το χρονόμετρο και ανάλογα με το είδος του benchmark υπολογίζει είτε τον ρυθμό μεταφοράς (throughput) σε Mbps, είτε την καθυστέρηση μεταφοράς (latency) σε sec.

Τα βήματα εκτέλεσης του παραλήπτη είναι:

1. Εκτέλεση μιας συνάρτησης αρχικοποίησης του παραλήπτη (*init_child*).

2. Εκτελεί τα παρακάτω για έναν, καθορισμένο από το χρήστη, αριθμό επαναλήψεων:

   - *Throughput Benchmark*

     (a) Ζητάει από τον υποκείμενο μηχανισμό ipc να του δώσει έναν buffer, ο οποίος θα περιέχει το επόμενο μήνυμα (*get_read_buffer*).

     (b) Αν δεν έχει ενεργοποιηθεί η επιλογή *read-in-place* από το χρήστη αντιγράφει το μήνυμα σε έναν ιδιωτικό buffer.

     (c) Αν έχει ενεργοποιηθεί η επιλογή *verify* από το χρήστη διαβάζει το μήνυμα και ελέγχει την εγκυρότητα του.

     (d) Ενημερώνει τον υποκείμενο μηχανισμό ipc ότι τελείωσε με το μήνυμα και αποδεσμεύει έτσι τον αντίστοιχο buffer (*release_read_buffer*).

   - *ping - pong Latency Benchmark*

     (a) Ζητάει από τον υποκείμενο μηχανισμό ipc να στείλει ένα μήνυμα (pong) στον παραλήπτη (*child_ping*).

3. Εκτέλεση μιας συνάρτησης τερματισμού του παραλήπτη (*finish_child*).

Το ipc-bench επιτρέπει την παραμετροποίηση της εκτέλεσης ενός benchmark, παρέχοντας ένα σύνολο από παραμέτρους γραμμής εντολών (command line arguments):

- *-a <cpuid>*
  Καθορίζει τον επεξεργαστή στον οποίο θα καρφιτσωθεί η διεργασία - παραλήπτης.

- *-b <cpuid>*
  Καθορίζει τον επεξεργαστή στον οποίο θα καρφιτσωθεί η διεργασία - αποστολέας.

- *-s <bytes>*
  Το μέγεθος κάθε μηνύματος, που στέλνει ο αποστολέας στον παραλήπτη, σε bytes.

- *-c <num>*
  Ο αριθμός των επαναλήψεων που θα εκτελεστεί το benchmark.

- *-n <node>*
  Ο κόμβος μνήμης (NUMA node) στον οποίο θα πρέπει να τοποθετηθεί ένα κομμάτι μοιραζόμενης μνήμης, για benchmarks που βασίζονται σε μοιραζόμενη μνήμη.

- *-r*
  Ενεργοποιεί την επιλογή *read-in-place* για τον παραλήπτη.

- *-w*
  Ενεργοποιεί την επιλογή *write-in-place* για τον αποστολέα.

- *-v*
  Ενεργοποιεί την επιλογή *verify* για τον παραλήπτη.

- *-o <directory>*
  Καθορίζει τον κατάλογο στον οποίο θα αποθηκευτούν τα αποτελέσματα.

Για τη δημιουργία ενός benchmark, για την αξιολόγηση ενός μηχανισμού IPC, αρκεί να υλοποιήσουμε ορισμένες συναρτήσεις, τις οποίες καλεί το ipc-bench κατά την εκτέλεσή του και οι οποίες αναφέρθηκαν και προηγούμενα στα βήματα εκτέλεσης του αποστολέα και του παραλήπτη. Συγκεκριμένα οι συναρτήσεις αυτές είναι:

- *init_test*
  Γενική συνάρτηση αρχικοποίησης του μηχανισμού IPC, που καλείται πριν δημιουργηθούν οι διεργασίες του αποστολέα και του παραλήπτη. Όποιες δομές δημιουργηθούν εδώ θα κληρονομηθούν, μέσω του fork, και στις δύο διεργασίες.

- *init_parent*
  Συνάρτηση αρχικοποίησης που καλείται από τη διεργασία - αποστολέα, πριν ξεκινήσει την αποστολή μηνυμάτων.

- *finish_parent*
  Συνάρτηση τερματισμού που καλείται από τη διεργασία - αποστολέα, όταν τελειώσει με την αποστολή των μηνυμάτων.

- *init_child*
  Συνάρτηση αρχικοποίησης που καλείται από τη διεργασία - παραλήπτη, πριν ξεκινήσει τη λήψη μηνυμάτων.

- *finish_child*
  Συνάρτηση τερματισμού που καλείται από τη διεργασία - παραλήπτη, όταν τελειώσει με την επεξεργασία των εισερχόμενων μηνυμάτων.

- Συναρτήσεις για *Throughput Benchmark*

  - *get_write_buffer*
    Η συνάρτηση αυτή καλείται από τον αποστολέα για να ζητήσει από τον υποκείμενο μηχανισμό IPC έναν buffer για να τοποθετήσει τα δεδομένα του μηνύματος προς αποστολή.

  - *release_write_buffer*
    Η συνάρτηση αυτή καλείται από τον αποστολέα για να ζητήσει από τον υποκείμενο μηχανισμό IPC να στείλει στον παραλήπτη το μήνυμα που βρίσκεται στον buffer που επέστρεψε η *get_write_buffer*.

  - *get_read_buffer*
    Η συνάρτηση αυτή καλείται από τον παραλήπτη για να ζητήσει από τον υποκείμενο μηχανισμό IPC να του δώσει το επόμενο μήνυμα προς ανάγνωση.

  - *release_read_buffer*
    Η συνάρτηση αυτή καλείται από τον παραλήπτη για να ενημερώσει τον υποκείμενο μηχανισμό IPC ότι τελείωσε με ένα μήνυμα.

- Συναρτήσεις για *ping - pong Latency Benchmark*

  - *parent_ping*
    Καλείται από τον αποστολέα για να στείλει ένα μήνυμα (ping) στον παραλήπτη.

  - *child_ping*
    Καλείται από τον παραλήπτη για να στείλει ένα μήνυμα (pong) στον αποστολέα.

Από την περιγραφή αυτή γίνεται φανερό ότι το ipc-bench πετυχαίνει σε μεγάλο βαθμό τους στόχους που θέσαμε στην αρχή του κεφαλαίου, δηλαδή παρέχει ένα κοινό πλαίσιο αξιολόγησης των μηχανισμών IPC, που επιτρέπει τη σύγκριση των επιδόσεων τους, και

είναι σχετικά εύκολο να γράψουμε καινούρια benchmarks για την αξιολόγηση νέων ή υπαρχόντων μηχανισμών.

3.2 Επέκταση του ipc-bench

Το ipc-bench παρέχει ένα ικανοποιητικό πλαίσιο για τη συγγραφή benchmarks για μηχανισμούς διαδιεργασιακής επικοινωνίας. Ωστόσο, δε μας δίνει τη δυνατότητα να εξετάσουμε πώς κλιμακώνει ένας μηχανισμός IPC στην περίπτωση ενός πολυ-νηματικού αποστολέα και παραλήπτη. Για το λόγο αυτό και λαμβάνοντας υπόψη τη διάδοση των πολυ-πύρηνων αρχιτεκτονικών και των πολυ-νηματικών εφαρμογών, θεωρήσαμε αναγκαίο να επεκτείνουμε το ipc-bench για να καλύψουμε το κενό αυτό. Ονομάσαμε την επέκταση ipc-bench-mt. Το ipc-bench-mt συνεχίζει να υποστηρίζει όλη τη λειτουργικότητα του ipc-bench και επιπλέον παρέχει την επιλογή οι διεργασίες του αποστολέα και του παραλήπτη να είναι πολυ-νηματικές. Στη συνέχεια, δίνουμε μια σύντομη περιγραφή της πολυ-νηματικής λειτουργίας του ipc-bench-mt και του πώς μπορούμε να υλοποιήσουμε benchmarks που εκμεταλλεύονται αυτό τον τρόπο λειτουργίας. Σημειώνουμε ότι η πολυ-νηματική λειτουργία υποστηρίζεται μόνο για benchmarks που μετράνε τον ρυθμό μεταφοράς (throughput), μιας και δεν έχει νόημα ένα πολυ-νηματικό benchmark για τη μέτρηση της καθυστέρησης μεταφοράς (latency).

Για κάθε benchmark, που χρησιμοποιεί τον πολυ-νηματικό τρόπο λειτουργίας, το ipc-bench-mt δημιουργεί ένα ζεύγος διεργασιών. Η μία διεργασία είναι ο αποστολέας - παραγωγός και η άλλη διεργασία ο παραλήπτης - καταναλωτής. Πριν τη δημιουργία αυτών των διεργασιών εκτελείται μια συνάρτηση αρχικοποίησης του υποκείμενου μηχανισμού ipc (*init_test*). Σε αντίθεση με το ipc-bench οι διεργασίες δεν καρφιτσώνονται σε κάποιον επεξεργαστή. Το λειτουργικό σύστημα είναι ελεύθερο να επιλέξει σε ποιον επεξεργαστή θα δρομολογήσει την κάθε διεργασία και τα νήματα της.

Τα βήματα εκτέλεσης του αποστολέα είναι:

1. Εκτέλεση μιας συνάρτησης αρχικοποίησης του αποστολέα. (*init_parent*).
2. Ξεκινάει ένα χρονόμετρο.
3. Δημιουργεί ένα σύνολο νημάτων, με χρήση της βιβλιοθήκης pthreads. Το πλήθος των νημάτων καθορίζεται από το χρήστη. Ο συνολικός αριθμός μηνυμάτων,

που πρέπει να σταλούν στον παραλήπτη (ο οποίος δηλώνεται από την παράμετρο του αριθμού επαναλήψεων του benchmark, που είδαμε νωρίτερα), μοιράζεται ισόποσα σε όλα τα threads. Δηλαδή κάθε thread στέλνει στον παραλήπτη ένα μέρος από τα συνολικά μηνύματα. Η εκτέλεση κάθε νήματος περιλαμβάνει τα εξής βήματα:

(a) Εκτέλεση μιας συνάρτησης αρχικοποίησης ανά νήμα του αποστολέα (*init_parent_thread*).

(b) Ζητάει από τον υποκείμενο μηχανισμό ipc έναν buffer για να τοποθετήσει τα δεδομένα του μηνύματος προς αποστολή (*get_write_buffer*).

(c) Αν έχει ενεργοποιηθεί η επιλογή *write-in-place* από το χρήστη παράγει απευθείας στον προηγούμενο buffer τα δεδομένα προς αποστολή. Διαφορετικά τα παράγει σε έναν ιδιωτικό buffer, τον οποίο στη συνέχεια αντιγράφει στον buffer που του δόθηκε στο προηγούμενο βήμα.

(d) Ζητάει από τον υποκείμενο μηχανισμό ipc να στείλει το μήνυμα στον παραλήπτη (*release_write_buffer*).

(e) Εκτέλεση μιας συνάρτησης τερματισμού ανά νήμα του αποστολέα (*finish_parent_thread*).

4. Αναμονή να ολοκληρώσουν όλα τα νήματα την εκτέλεση τους (*pthread_join*).

5. Εκτέλεση μιας συνάρτησης τερματισμού του αποστολέα (*finish_parent*).

6. Σταματάει το χρονόμετρο και υπολογίζει τον ρυθμό μεταφοράς σε Mbps.

Τα βήματα εκτέλεσης του παραλήπτη είναι:

1. Εκτέλεση μιας συνάρτησης αρχικοποίησης του παραλήπτη (*init_child*).

2. Δημιουργεί ένα σύνολο νημάτων, με χρήση της βιβλιοθήκης pthreads. Το πλήθος των νημάτων καθορίζεται από το χρήστη. Ο συνολικός αριθμός μηνυμάτων, που πρέπει να επεξεργαστεί ο παραλήπτης (ο οποίος δηλώνεται από την παράμετρο του αριθμού επαναλήψεων του benchmark, που είδαμε νωρίτερα), μοιράζεται ισόποσα σε όλα τα threads. Δηλαδή κάθε thread επεξεργάζεται ένα μέρος από τα συνολικά μηνύματα. Η εκτέλεση κάθε νήματος περιλαμβάνει τα εξής βήματα:

   (a) Εκτέλεση μιας συνάρτησης αρχικοποίησης ανά νήμα του παραλήπτη (*init_child_thread*).

   (b) Ζητάει από τον υποκείμενο μηχανισμό ipc να του δώσει έναν buffer, ο οποίος θα περιέχει το επόμενο μήνυμα (*get_read_buffer*).

   (c) Αν δεν έχει ενεργοποιηθεί η επιλογή *read-in-place* από το χρήστη, αντιγράφει το μήνυμα σε έναν ιδιωτικό buffer.

   (d) Αν έχει ενεργοποιηθεί η επιλογή *verify* από το χρήστη διαβάζει το μήνυμα και ελέγχει την εγκυρότητα του.

   (e) Ενημερώνει τον υποκείμενο μηχανισμό ipc ότι τελείωσε με το μήνυμα και αποδεσμεύει έτσι τον αντίστοιχο buffer (*release_read_buffer*).

   (f) Εκτέλεση μιας συνάρτησης τερματισμού ανά νήμα του παραλήπτη (*finish_child_thread*).

3. Αναμονή να ολοκληρώσουν όλα τα νήματα την εκτέλεση τους (*pthread_join*).

4. Εκτέλεση μιας συνάρτησης τερματισμού του παραλήπτη (*finish_child*).

Για τη παραμετροποίηση της εκτέλεσης ενός benchmark προσθέσαμε δύο επιπλέον παραμέτρους γραμμής εντολών (command line arguments):

- *-d <num>*
  Καθορίζει το πλήθος των threads της διεργασίας - αποστολέα.

- *-e <num>*
  Καθορίζει το πλήθος των threads της διεργασίας - παραλήπτη.

Για τη δημιουργία ενός benchmark, που θα χρησιμοποιεί τη πολυ-νηματική λειτουργία του ipc-bench-mt, πρέπει να υλοποιήσουμε τις ίδιες συναρτήσεις με πριν και τις παρακάτω καινούριες:

- *init_parent_thread*
  Συνάρτηση αρχικοποίησης που καλείται από κάθε νήμα της διεργασίας - αποστολέα, πριν ξεκινήσει την αποστολή μηνυμάτων.

- *finish_parent_thread*
  Συνάρτηση τερματισμού που καλείται από κάθε νήμα της διεργασίας - αποστολέα, όταν τελειώσει με την αποστολή των μηνυμάτων.

- *init_child_thread*
  Συνάρτηση αρχικοποίησης που καλείται από κάθε νήμα της διεργασίας - παραλήπτη, πριν ξεκινήσει τη λήψη μηνυμάτων.

- *finish_child_thread*
  Συνάρτηση τερματισμού που καλείται από κάθε νήμα της διεργασίας - παραλήπτη, όταν τελειώσει με την επεξεργασία των εισερχόμενων μηνυμάτων.

Τέλος, για τη σωστή λειτουργία του ipc-bench-mt, όταν οι διεργασίες του αποστολέα και του παραλήπτη είναι πολυ-νηματικές, απαιτείται οι ακόλουθες καινούριες συναρτήσεις να είναι thread-safe:

- *init_parent_thread*
- *finish_parent_thread*
- *init_child_thread*
- *finish_child_thread*
- *get_write_buffer*
- *release_write_buffer*
- *get_read_buffer*
- *release_read_buffer*

4 Σχεδιασμός του zipc

Στο κεφάλαιο αυτό θα ασχοληθούμε με τον σχεδιασμό του zipc, ενός μηχανισμού διαδιεργασιακής επικοινωνίας μηδενικών αντιγράφων. Όπως έχει ήδη αναφερθεί, ο μηχανισμός αυτός βασίζεται στη δωρεά σελίδων εικονικής μνήμης από τον αποστολέα στον παραλήπτη (page re-mapping). Πρόκειται, ουσιαστικά, για την τελευταία από τις τεχνικές μεταφοράς δεδομένων που αναλύσαμε στο κεφάλαιο 2.3.

Σε υψηλό επίπεδο ο μηχανισμός αυτός μεταφέρει τα δεδομένα από την εμβέλεια της μίας διεργασίας στην εμβέλεια της άλλης, καταργώντας τις αντίστοιχες απεικονίσεις από τους πίνακες σελίδων του αποστολέα και φτιάχνοντας νέες απεικονίσεις για τα δεδομένα στους πίνακες σελίδων του παραλήπτη. Κατά την υλοποίηση αυτής της λειτουργικότητας προκύπτουν διάφορες δυσκολίες, οι οποίες προέρχονται από περιορισμούς που θέτει τόσο το υλικό, όσο και ο πυρήνας του Linux.

Στη συνέχεια παρουσιάζουμε όλες τις δομές και λειτουργίες που απαιτούνται για την υλοποίηση του μηχανισμού αυτού. Επίσης, παρουσιάζουμε αναλυτικά όλες τις δυσκολίες και τις αποφάσεις που χρειάστηκε να λάβουμε κατά τη διάρκεια του σχεδιασμού του zipc.

4.1 Γενικά

Μία από τις πρώτες επιλογές που χρειάζεται να πραγματοποιηθούν, κατά το σχεδιασμό ενός νέου μηχανισμού διαδιεργασιακής επικοινωνίας, είναι αν αυτός θα υλοποιηθεί στο χώρο χρήστη (user space) ή στο χώρο πυρήνα (kernel space). Κάθε επιλογή έχει τα

πλεονεκτήματα και τα μειονεκτήματά της (βλέπε κεφάλαια 2.1.4, 2.1.5 και 2.1.6). Στην προκειμένη περίπτωση, λόγω της φύσης του μηχανισμού που σχεδιάσαμε και της αναγκαιότητας να μεταχειριζόμαστε τους πίνακες σελίδων των εμπλεκόμενων διεργασιών, η υλοποίηση στο χώρο πυρήνα ήταν μονόδρομος.

Ωστόσο, υπάρχει ένας σημαντικότερος λόγος πίσω από αυτή την απόφαση. Ο καθιερωμένος τρόπος για διαδιεργασιακή επικοινωνία μηδενικών αντιγράφων είναι με χρήση μοιραζόμενης μνήμης. Η μοιραζόμενη μνήμη προσφέρει υψηλό ρυθμό μεταφοράς των δεδομένων (throughput) και χαμηλή καθυστέρηση μεταφοράς (latency). Όπως είδαμε στο κεφάλαιο 2.3.4, ο τρόπος δημιουργίας ενός κομματιού μοιραζόμενης μνήμης είναι προδιαγεγραμμένος και αυστηρά ορισμένος από τα πρότυπα POSIX. Αντίθετα, δεν υπάρχει κάποιο πρότυπο για την επικοινωνία με χρήση μοιραζόμενης μνήμης, οπότε κάθε εφαρμογή πρέπει να υλοποιήσει το δικό της μηχανισμό. Ένας τέτοιος μηχανισμός πρέπει, μεταξύ άλλων, να θέτει κανόνες για το πότε και πώς κάθε διεργασία θα διαβάζει / γράφει στη μοιραζόμενη μνήμη, για την οργάνωση των δεδομένων και για το συγχρονισμό των εμπλεκόμενων διεργασιών. Ταυτόχρονα, απαιτείται ιδιαίτερη προσοχή για την αποφυγή καταστάσεων συναγωνισμού (race conditions) και αδιεξόδων (deadlocks). Τέλος, το σημαντικότερο, ίσως, μειονέκτημα αυτής της προσέγγισης είναι ότι απαιτεί να υπάρχει εμπιστοσύνη μεταξύ των διεργασιών που επικοινωνούν. Εφόσον η υλοποίηση αυτών των μηχανισμών γίνεται στο χώρο χρήστη, δεν υπάρχει τίποτα που να εξασφαλίζει ότι οι διεργασίες θα σεβαστούν τους κανόνες επικοινωνίας. Ως εκ τούτου οι μηχανισμοί αυτοί, σε αντίθεση με μηχανισμούς όπως τα unix sockets και τα fifos, δεν μπορούν να χρησιμοποιηθούν για την επικοινωνία αυθαίρετων διεργασιών. Ένας από τους στόχους μας ήταν η δημιουργία ενός μηχανισμού μηδενικών αντιγράφων που να μπορεί να χρησιμοποιηθεί με ασφάλεια για επικοινωνία τυχαίων διεργασιών. Για το λόγο αυτό, λοιπόν, αποφασίσαμε να αναθέσουμε στον πυρήνα το ρόλο του συντονιστή, ώστε να εξασφαλίσει απομόνωση ανάμεσα στις διεργασίες και προστασία μνήμης.

θα μπορούσαμε να υλοποιήσουμε τον μηχανισμό ως ένα σύνολο από καινούριες κλήσεις συστήματος (system calls). Ωστόσο, η λύση αυτή είναι παρεμβατική αφού θα απαιτούσε να τροποποιήσουμε τον πυρήνα του Linux. Επιπρόσθετα, θα έκανε την αποσφαλμάτωση και τη δοκιμή νέων λειτουργιών πιο δύσκολη αφού κάθε αλλαγή στον πηγαίο κώδικα θα απαιτούσε εκ᾽ νέου μεταγλώττιση του πυρήνα και επανεκκίνηση του μηχανήματος. Επιλέξαμε, λοιπόν, να υλοποιήσουμε τον zipc ως μια συσκευή χαρακτήρων (character device), αφού αυτό το μοντέλο συσκευής είναι το πιο κοντινό στη λειτουρ-

γικότητα που θα προσφέρουμε (βλέπε 2.1.7). Οι λειτουργίες του μηχανισμού παρέχονται με την υλοποίηση των κατάλληλων μεθόδων της συσκευής αυτής (βλέπε 2.1.7). Η επιλογή αυτή έχει το πλεονέκτημα ότι δεν απαιτείται μεταγλώττιση του πυρήνα και, επιπλέον, μπορούμε να προσθέτουμε και να αφαιρούμε δυναμικά τον οδηγό συσκευής, γεγονός που κάνει την αποσφαλμάτωση και τη δοκιμή νέων λειτουργιών σημαντικά ευκολότερη. Ωστόσο, η προσέγγιση αυτή έχει και ορισμένα μειονεκτήματα. Συγκεκριμένα, αρκετή από τη λειτουργικότητα που παρέχει ο πυρήνας σχετικά με τον χειρισμό των Πινάκων Σελίδων μιας διεργασίας δεν εξάγεται για χρήση από τα modules. Ως εκ τούτου χρειάστηκε να βρούμε εναλλακτικούς τρόπους για να μπορέσουμε να χρησιμοποιήσουμε αυτή τη λειτουργικότητα. Περισσότερες λεπτομέρειες δίνονται στο κεφάλαιο 5, που έχει ως θέμα την υλοποίηση του zipc.

4.2 Κατηγορίες πλαισίων μνήμης

Ο πυρήνας του Linux διακρίνει τα πλαίσια μνήμης στους παρακάτω τύπους:

- **Mapped Pages**
  Ένα πλαίσιο μνήμης είναι απεικονισμένο (mapped) εάν απεικονίζει μέρος ενός αρχείου. Για παράδειγμα, όλα τα πλαίσια, στο χώρο διευθύνσεων κάποιας διεργασίας, που ανήκουν σε μια απεικόνιση αρχείου (file memory mapping) είναι mapped. Επίσης, όλα τα πλαίσια που περιέχονται στην page cache είναι mapped.

- **Anonymous Pages**
  Ένα πλαίσιο καλείται ανώνυμο (anonymous) εάν ανήκει σε μια ανώνυμη περιοχή εικονικής μνήμης (anonymous vma) μιας διεργασίας, δηλαδή σε μια περιοχή εικονικής μνήμης που δεν απεικονίζει κάποιο αρχείο. Για παράδειγμα, όλα τα πλαίσια στο σωρό και στη στοίβα μιας διεργασίας είναι ανώνυμα.

Ειδικά για πλαίσια που απεικονίζονται στους Πίνακες Σελίδων κάποιας διεργασίας διακρίνουμε τις εξής δύο περιπτώσεις:

- **Μοιραζόμενα Πλαίσια (Shared page frames)**
  Ένα μοιραζόμενο πλαίσιο μνήμης απεικονίζεται στους Πίνακες Σελίδων πολλών

διεργασιών. Μοιραζόμενα πλαίσια δημιουργούνται όταν μια διεργασία δημιουργεί μια καινούρια διεργασία - παιδί, μέσω της κλήσης συστήματος *fork*. Μια άλλη συνηθισμένη περίπτωση είναι όταν δύο ή περισσότερες διεργασίες προσπελάσουν το ίδιο αρχείο μέσω μιας μοιραζόμενης απεικόνισης μνήμης (shared memory mapping).

- **Ιδιωτικά Πλαίσια (non-shared page frames)**
  Ένα ιδιωτικό (μη μοιραζόμενο) πλαίσιο μνήμης απεικονίζεται στους Πίνακες Σελίδων μίας μόνο διεργασίας. Ωστόσο, ένα ιδιωτικό πλαίσιο μνήμης μπορεί να ανήκει σε πολλαπλά νήματα (threads ή lightweight processes), που μοιράζονται τον ίδιο χώρο διευθύνσεων.

Στη πραγματικότητα ο πυρήνας του Linux, στα πλαίσια του αλγόριθμου ανάκτησης μνήμης, διακρίνει τα πλαίσια μνήμης (page frames) σε περισσότερες κατηγορίες. Ωστόσο, στην παρούσα εργασία μάς ενδιαφέρουν μόνο οι προηγούμενοι τύποι πλαισίων.

4.3 Προέλευση των δεδομένων - Μνήμη Αποστολέα

Σε όλους τους μηχανισμούς διαδιεργασιακής επικοινωνίας, που εξετάσαμε στο κεφάλαιο 2.3, τα δεδομένα του αποστολέα μπορούσαν να προέρχονται από οπουδήποτε. Αρκεί η αντίστοιχη μνήμη να είναι έγκυρη και η διεργασία του αποστολέα να έχει δικαιώματα ανάγνωσης ή/και εγγραφής σε αυτή. Αυτό είναι εφικτό επειδή οι μηχανισμοί αυτοί δημιουργούν τουλάχιστον ένα αντίγραφο των δεδομένων. Ο μηχανισμός που υλοποιήσαμε, ωστόσο, μεταφέρει ένα πλαίσιο μνήμης, που πρέπει να είναι απεικονισμένο στους Πίνακες Σελίδων του αποστολέα, από τον αποστολέα στον παραλήπτη. Η λειτουργικότητα αυτή θέτει ορισμένους περιορισμούς στη προέλευση των δεδομένων. Στη συνέχεια εξετάζουμε τους περιορισμούς αυτούς αναλυτικά.

Τα δεδομένα προς αποστολή πρέπει να είναι ευθυγραμμισμένα στη μνήμη στο όριο μιας σελίδας, δηλαδή η διεύθυνση έναρξης ενός buffer πρέπει να είναι πολλαπλάσιο του μεγέθους μιας σελίδας (*PAGE_SIZE*). Επίσης, το μέγεθος των buffers που αποστέλλονται πρέπει να είναι πολλαπλάσιο του μεγέθους μιας σελίδας. Οι περιορισμοί αυτοί προκύπτουν από τη φύση του μηχανισμού zipc και από τον τρόπο που χειρίζεται τη μνήμη το υποσύστημα διαχείρισης μνήμης του Linux. Όπως ήδη αναφέραμε, ο zipc μεταφέ-

ρει τα δεδομένα από την εμβέλεια της μιας διεργασίας στην εμβέλεια μιας άλλης, καταργώντας τις αντίστοιχες απεικονίσεις από τους πίνακες σελίδων του αποστολέα και φτιάχνοντας νέες απεικονίσεις για τα δεδομένα στους πίνακες σελίδων του παραλήπτη. Ωστόσο, όπως έχουμε εξετάσει στα κεφάλαια 2.2.2 και 2.2.5, ο χειρισμός της μνήμης από το λειτουργικό σύστημα γίνεται με βάση τις σελίδες. Οι πίνακες σελίδων των διεργασιών, όπως έχουμε αναφέρει, ομαδοποιούν τις εικονικές διευθύνσεις σε σελίδες, ίδιου μεγέθους με τις φυσικές, και προσδιορίζουν τη φυσική διεύθυνση μιας σελίδας ως σύνολο, αντί κάθε εικονικής διεύθυνσης μέσα στη σελίδα χωριστά. Αυτό σημαίνει ότι η ελάχιστη μονάδα μνήμης που μπορούμε να μεταφέρουμε από μία διεργασία σε μία άλλη είναι η σελίδα. Δεδομένου ότι οι σελίδες δεν αντιγράφονται αλλά μεταφέρονται, δεν είναι δυνατή η μεταφορά μόνο ενός κομματιού μιας σελίδας, αλλά αντίθετα πρέπει να μεταφερθεί ολόκληρη.

Όταν μια διεργασία στέλνει δεδομένα σε μια άλλη, χρησιμοποιώντας το μηχανισμό μας, δωρίζει ουσιαστικά τις υποκείμενες σελίδες μνήμης στον παραλήπτη. Για να μπορεί να δωρίσει, όμως, αυτές τις σελίδες θα πρέπει να είναι ο αποκλειστικός κάτοχός τους. Θα ήταν σημασιολογικά λάθος να επιτρέπαμε στον αποστολέα να δωρίσει σελίδες που δεν ανήκουν μόνο σε αυτόν. Η απαίτηση αυτή θέτει περιορισμούς στο είδος των πλαισίων μνήμης που μπορούν να φιλοξενούν τα δεδομένα προς αποστολή:

- *Anonymous pages*
  - *Ιδιωτικά πλαίσια (non-shared page frames)*
    Τα πλαίσια αυτά ανήκουν αποκλειστικά σε μια διεργασία και επομένως έχει δικαίωμα να τα δωρίσει σε κάποια άλλη.
  - *Μοιραζόμενα πλαίσια (Shared page frames)*
    Διάφορες διεργασίες μπορεί να μοιράζονται μια ανώνυμη σελίδα. Η πιο συνηθισμένη περίπτωση είναι κατά τη δημιουργία μιας διεργασίας - παιδιού μέσω της κλήσης συστήματος *fork*: όλα τα πλαίσια μνήμης που ανήκουν στο γονιό ανατίθενται και στο παιδί. Μια άλλη, λιγότερο συνηθισμένη, περίπτωση εμφανίζεται όταν μια διεργασία δημιουργεί μια περιοχή εικονικής μνήμης με την κλήση συστήματος mmap και προσδιορίζει τις σημαίες *MAP_ANONYMOUS* και *MAP_SHARED* - όλα τα πλαίσια που ανήκουν σε αυτή την περιοχή θα μοιράζονται από αυτή τη διεργασία και όλους τους απογόνους της.

Τα πλαίσια αυτής της κατηγορίας δεν μπορούν να δωριστούν, αφού δεν ανήκουν αποκλειστικά σε μία διεργασία.

- *Mapped pages*

  - *Ιδιωτικά πλαίσια (non-shared page frames)*
    Τα πλαίσια αυτά μπορεί να ανήκουν είτε σε μια ιδιωτική απεικόνιση αρχείου (private file memory mapping) είτε σε μια μοιραζόμενη απεικόνιση αρχείου (shared file memory mapping). Αν και τα πλαίσια μνήμης αυτής της κατηγορίας είναι απεικονισμένα στους πίνακες σελίδων μίας μόνο διεργασίας, δεν επιτρέπουμε τη μεταφορά τους. Και στις δύο προηγούμενες περιπτώσεις τα πλαίσια ανήκουν και στην page cache και περιέχουν δεδομένα ενός αρχείου, τα οποία μπορεί να μην είναι συγχρονισμένα με το αρχείο στο δίσκο. Άρα, για να δωρίσει μια διεργασία ένα τέτοιο πλαίσιο, πρέπει να το "κλέψει" από την page cache, συγχρονίζοντάς το ταυτόχρονα με τον δίσκο, αν χρειάζεται. Αυτό μπορεί να αποδειχθεί ιδιαίτερα χρονοβόρο και για το λόγο αυτό αποφασίσαμε να μην επιτρέπουμε τα δεδομένα προς αποστολή να περιέχονται σε τέτοιου είδους πλαίσια. Σημειώνουμε ότι στην περίπτωση της ιδιωτικής απεικόνισης αρχείου χρησιμοποιείται ο μηχανισμός COW (Copy On Write). Δηλαδή, αν η διεργασία επιχειρήσει να γράψει σε ένα πλαίσιο που ανήκει σε μια ιδιωτική απεικόνιση θα δημιουργηθεί ένα αντίγραφο αυτού του πλαισίου και η εγγραφή θα πραγματοποιηθεί στο αντίγραφο. Το νέο πλαίσιο δεν εμπίπτει πλέον στην κατηγορία των *Mapped pages*, οπότε γι αυτό ισχύουν όσα αναφέρθηκαν για τα *Anonymous pages*.

  - *Μοιραζόμενα πλαίσια (Shared page frames)*
    Ο διαμοιρασμός των *mapped pages* είναι ένα συχνό φαινόμενο. Για παράδειγμα, πολλές από τις διεγασίες του συστήματος μοιράζονται τις σελίδες που περιέχουν τον κώδικα της βιβλιοθήκης της C (standard C library). Τα πλαίσια αυτής της κατηγορίας δεν μπορούν να δωριστούν αφού δεν ανήκουν αποκλειστικά σε μία διεργασία.

Σημειώνουμε ότι υπάρχει τρόπος να χειριστούμε τους τύπους πλαισίων, που απορρίψαμε. Θα μπορούσαμε να ακολουθήσουμε το παράδειγμα του αλγόριθμου ανάκτησης

μνήμης του Linux και να καταργήσουμε τις απεικονίσεις των πλαισίων αυτών από τους πίνακες σελίδων όλων των διεργασιών στις οποίες είναι απεικονισμένα, γράφοντας παράλληλα τα περιεχόμενα των πλαισίων είτε στο δίσκο (αν απαιτείται) είτε στη περιοχή swap (ανάλογα με το αν πρόκειται για anonymous ή mapped pages). Η διαδικασία αυτή, όμως, είναι ιδιαίτερα χρονοβόρα και υπάρχει ενδεχόμενο να αποτύχει, δηλαδή μπορεί να μην καταφέρουμε να βρούμε όλες τις διεργασίες που έχουν καταχώρηση στους πίνακες σελίδων τους για κάποιο πλαίσιο. Ο στόχος του μηχανισμού που αναπτύσσουμε είναι να αυξήσει την απόδοση της διαδιεργασιακής επικοινωνίας αποφεύγοντας τα περιττά αντίγραφα. Το γράψιμο (flush) ενός πλαισίου μνήμης στο δίσκο είναι αντίθετο με αυτό το στόχο, οπότε αποφασίσαμε να μην χειριζόμαστε τα πλαίσια των αντίστοιχων κατηγοριών.

Τέλος, δε χειριζόμαστε πλαίσια που ανήκουν σε ορισμένες ειδικές κατηγορίες περιοχών εικονικής μνήμης: Περιοχές μνήμης που απεικονίζουν μνήμη μιας συσκευής, περιοχές μνήμης "κλειδωμένες" στη φυσική μνήμη κατά απαίτηση της διεργασίας (δηλαδή τα πλαίσια μνήμης που ανήκουν σε μια τέτοια περιοχή δεν πρέπει να φύγουν από τη κύρια μνήμη) και περιοχές μνήμης που χρησιμοποιούν "μεγάλες σελίδες" (huge pages), δηλαδή σελίδες μεγαλύτερου μεγέθους από το "κανονικό", για τις οποίες απαιτείται ειδικός χειρισμός των πινάκων σελίδων.

4.4 Προορισμός των δεδομένων - Μνήμη παραλήπτη

Οι buffers που θα φιλοξενήσουν τις εισερχόμενες σελίδες πρέπει να είναι ευθυγραμμισμένοι στη μνήμη στο όριο μιας σελίδας, δηλαδή να έχουν διεύθυνση έναρξης που να είναι πολλαπλάσιο του μεγέθους μιας σελίδας, και να έχουν μέγεθος πολλαπλάσιο του μεγέθους μιας σελίδας. Ο λόγος γι' αυτούς τους περιορισμούς είναι ο ίδιος με αυτόν που αναφέραμε στην προηγούμενη ενότητα 4.3 για τους buffers του αποστολέα.

Στους μηχανισμούς IPC που εξετάσαμε στο κεφάλαιο 2.3 τα εισερχόμενα δεδομένα αντιγράφονταν στους buffers προορισμού και κατά επέκταση στις υποκείμενες σελίδες φυσικής μνήμης. Αυτό επέτρεπε η μνήμη προορισμού να είναι οποιοδήποτε έγκυρη περιοχή εικονικής μνήμης για την οποία ο παραλήπτης έχει δικαίωμα εγγραφής. Θυμίζουμε ότι υπάρχουν δύο είδη περιοχών εικονικής μνήμη:

1. *Ανώνυμες περιοχές εικονικής μνήμης (Anonymous VMAs)*: Είναι οι περιοχές που

δεν απεικονίζουν κάποιο αρχείο ή συσκευή και υποστηρίζονται από κάποιο swap area.

2. *Απεικονισμένες περιοχές εικονικής μνήμης (mapped VMAs)*: Είναι οι περιοχές που απεικονίζουν κάποιο αρχείο ή συσκευή.

Όταν μια διεργασία λαμβάνει δεδομένα με χρήστη του zipc οι υποκείμενες σελίδες φυσικής μνήμης απεικονίζονται στο χώρο διευθύνσεών της. Αν η μνήμη προορισμού απεικονίζει κάποιο αρχείο, τότε δεν αρκεί να απεικονίσουμε τις εισερχόμενες σελίδες στους πίνακες σελίδων της διεργασίας. Πρέπει να καταργήσουμε τις παλιές σελίδες από την page cache, να προσθέσουμε τις καινούριες σε αυτή και να ενημερώσουμε τους πίνακες σελίδων όλων των διεργασιών που πιθανώς απεικονίζουν το ίδιο αρχείο στους χώρους διευθύνσεών τους. Αυτή είναι μια χρονοβόρα διαδικασία, που μπορεί και να αποτύχει, οπότε αποφασίσαμε να μην υποστηρίζουμε τις περιοχές εικονικής μνήμης, που απεικονίζουν αρχεία, ως πιθανούς προορισμούς για τις σελίδες που μεταφέρει ο μηχανισμός μας.

Επομένως, περιοριζόμαστε στις ανώνυμες περιοχές εικονικής μνήμης και μάλιστα μόνο στις ιδιωτικές (private anonymous mappings). Μια μοιραζόμενη ανώνυμη περιοχή εικονικής μνήμης (shared anonymous mapping) δημιουργείται με την κλήση συστήματος *mmap*, με χρήση των σημαιών *MAP_ANONYMOUS* και *MAP_SHARED*, και έχει ως σκοπό τον διαμοιρασμό δεδομένων ανάμεσα σε συγγενικές διεργασίες (βλέπε Μοιραζόμενη Μνήμη κεφάλαιο 2.3.4). Και σε αυτή την περίπτωση θα ήταν απαραίτητη η ενημέρωση των αντίστοιχων απεικονίσεων στους πίνακες σελίδων όλων των εμπλεκόμενων διεργασιών και επιλέγουμε να μην την υποστηρίξουμε.

Τέλος, όπως και προηγουμένως, απορρίπτουμε ως buffers προορισμού τις εξής κατηγορίες περιοχών εικονικής μνήμης: Περιοχές μνήμης που απεικονίζουν μνήμη μιας συσκευής, περιοχές μνήμης "κλειδωμένες" στη φυσική μνήμη κατά απαίτηση της διεργασίας (δηλαδή τα πλαίσια μνήμης που ανήκουν σε μια τέτοια περιοχή δεν πρέπει να φύγουν από τη κύρια μνήμη) και περιοχές μνήμης που χρησιμοποιούν "μεγάλες σελίδες" (huge pages), δηλαδή σελίδες μεγαλύτερου μεγέθους από το "κανονικό" για τις οποίες απαιτείται ειδικός χειρισμός των πινάκων σελίδων.

4.5 Write / Read Semantics

Επιλέξαμε να διατηρήσουμε τη σημασιολογία των κλήσεων συστήματος write και read για την αποστολή και λήψη δεδομένων. Αυτές οι κλήσεις συστήματος χρησιμοποιούνται από τη πλειοψηφία των εφαρμογών τόσο για διαδιεργασιακή επικοινωνία, χρησιμοποιώντας κάποιους από τους μηχανισμούς του κεφαλαίου 2.3, όσο και για εκτέλεση λειτουργιών Εισόδου / Εξόδου. Θεωρήσαμε, ως εκ τούτου, σημαντικό να παρέχουμε αντίστοιχη συμπεριφορά στο μηχανισμό που αναπτύξαμε.

write semantics Όταν μια διεργασία χρησιμοποιεί την κλήση συστήματος write αναμένει ότι θα είναι σε θέση να χρησιμοποιήσει ξανά τον buffer, μόλις η κλήση της write επιστρέψει (copy semantics), χωρίς να αλλοιώσει τα μεταδιδόμενα δεδομένα. Για τη διατήρηση της σημασιολογίας της κλήσης συστήματος write, και λαμβάνοντας υπόψη τη φύση του μηχανισμού μας (μεταφορά σελίδων από μια διεργασία σε μια άλλη), υπάρχουν οι εξής επιλογές:

- *Χρήση του μηχανισμού COW για την προστασία των μεταδιδόμενων δεδομένων.* Αν η διεργασία - αποστολέας επιχειρήσει να γράψει στους απεσταλμένους buffers, πριν ο παραλήπτης τελειώσει με αυτούς, θα προκύψει page fault και ο πυρήνας θα αντιγράψει τα δεδομένα σε καινούριες σελίδες. Η εγγραφή θα πραγματοποιηθεί τελικά στις καινούριες σελίδες. Αυτή η προσέγγιση έχει το μειονέκτημα ότι αν ο αποστολέας προσπαθήσει να χρησιμοποιήσει τους buffers, πριν τελειώσει ο παραλήπτης με αυτούς, θα δημιουργηθεί ένα αντίγραφο των δεδομένων. Αυτό καταργεί το στόχο του μηχανισμού μας, που είναι η αποφυγή της δημιουργίας αντιγράφων των μεταδιδόμενων δεδομένων. Επιπρόσθετα, το κόστος θα είναι τελικά μεγαλύτερο σε σχέση με την εξ' αρχής δημιουργία ενός αντιγράφου. Ωστόσο, η προσέγγιση αυτή έχει το πλεονέκτημα ότι είναι πιο κοντά στα copy semantics της κλήσης συστήματος write. Επίσης, αν αποστολέας και παραλήπτης καταφέρουν να συγχρονιστούν (κάτι που δεν μπορεί να εξασφαλιστεί από τον μηχανισμό μας), έχει τη προοπτική να επιτύχει καλές επιδόσεις κατά τη μετάδοση των δεδομένων. Παρόλα αυτά, η υλοποίηση αυτής της προσέγγισης δεν είναι εφικτή λόγω περιορισμών που θέτει ο πυρήνας του Linux. Συγκεκριμένα, δεν προβλέπεται ένα ανώνυμο πλαίσιο να είναι απεικονισμένο ταυτόχρονα σε μη συγγενικές διεργασίες. Κάθε δομή τύπου *struct page* (βλέπε κεφάλαιο 2.2.4) έχει ένα πεδίο

index που είναι το offset του πλαισίου μνήμης μέσα στην ανώνυμη περιοχή εικονικής μνήμης. Για συγγενικές (μέσω του fork) διεργασίες κάθε μοιραζόμενο ανώνυμο πλαίσιο θα έχει το ίδιο offset μέσα σε όλες τις ανώνυμες περιοχές εικονικής μνήμης των εμπλεκόμενων διεργασιών, αφού ο χώρος διευθύνσεων μιας διεργασίας - παιδιού είναι αντίγραφο του χώρου διευθύνσεων του γονιού του. Όταν ο μηχανισμός μας απεικονίζει ένα πλαίσιο σε μια περιοχή εικονικής μνήμης ενός παραλήπτη, που δεν σχετίζεται μέσω του fork με τον αποστολέα, δεν μπορεί να εξασφαλίσει ότι το offset του πλαισίου θα είναι το ίδιο και στις δύο περιοχές εικονικής μνήμης που το φιλοξενούν. Θα μπορούσαμε μεν να απεικονίσουμε το πλαίσιο στον παραλήπτη, παρόλο που πιθανώς θα είχε λάθος offset, αλλά αυτό θα είχε ως συνέπεια ο μηχανισμός ανάκτησης μνήμης του πυρήνα να μην μπορεί να το ανακτήσει. Ο λόγος γιαυτό είναι ότι ο αλγόριθμος ανάκτησης μνήμης χρησιμοποιεί το πεδίο *index* για την εύρεση της αντίστοιχης καταχώρησης στους πίνακες σελίδων της διεργασίας. Κάτι τέτοιο είναι προφανώς ανεπιθύμητο.

- *Κατάργηση των αντίστοιχων απεικονίσεων από το χώρο διευθύνσεων του αποστολέα.*

  Καταργούνται τόσο οι αντίστοιχες καταχωρήσεις στους πίνακες σελίδων της διεργασίας - αποστολέα, όσο και οι αντίστοιχες περιοχές εικονικής μνήμης. Αυτό έχει ως συνέπεια επόμενη πρόσβαση σε buffers που έχουν σταλεί να προκαλεί σφάλμα κατάτμησης (segmentation fault). Αυτή η προσέγγιση διασφαλίζει μεν ότι ο αποστολέας δεν μπορεί να αλλοιώσει τα μεταδιδόμενα δεδομένα, αλλά δεν ικανοποιεί την απαίτηση για επαναχρησιμοποίηση των buffers, οπότε δεν την επιλέξαμε.

- *Κατάργηση των αντίστοιχων απεικονίσεων από τους πίνακες σελίδων του αποστολέα.*

  Σε αυτή την περίπτωση καταργούνται μόνο οι αντίστοιχες καταχωρήσεις στους πίνακες σελίδων της διεργασίας - αποστολέα και δεν πειράζονται οι σχετικές περιοχές εικονικής μνήμης. Εφόσον πρόκειται για ανώνυμες περιοχές μνήμης επόμενη πρόσβαση σε αυτές, από τον αποστολέα, θα προκαλέσει page fault και ο πυρήνας θα απεικονίσει σε αυτές νέα πλαίσια γεμάτα μηδενικά (demand paging). Αυτή η προσέγγιση είναι σημασιολογικά η κοντινότερη στο write λύση. Διαφέρει από τη τυπική κλήση συστήματος write στο ότι τα παλιά δεδομένα χάνονται

αλλά αυτό είναι σύμφωνο με το τρόπο λειτουργίας του μηχανισμού μας, ο οποίος βασίζεται στη δωρεά σελίδων εικονικής μνήμης από μια διεργασία σε μια άλλη. Για τους προηγούμενους λόγους επιλέξαμε να ακολουθήσουμε αυτή τη προσέγγιση.

read semantics Όταν μια διεργασία χρησιμοποιεί την κλήση συστήματος read δίνει στον πυρήνα τη διεύθυνση ενός buffer και αναμένει από αυτόν να τοποθετήσει εκεί τα δεδομένα. Δεδομένου ότι στον μηχανισμό μας οι σελίδες μεταφέρονται και δεν αντιγράφονται, ο μόνος τρόπος για να διατηρήσουμε αυτή τη συμπεριφορά είναι να απεικονίσουμε στον buffer, που δίνει ο χρήστης, τις εισερχόμενες σελίδες, καταργώντας παράλληλα τις απεικονίσεις σελίδων που μπορεί να ήταν ήδη απεικονισμένες σε αυτόν. Η συμπεριφορά αυτή είναι ακριβώς η αναμενόμενη από την κλήση συστήματος read.

4.6 Η βασική δομή δεδομένων του zipc

Η βασική δομή του zipc είναι μία σωλήνωση (pipe), αντίστοιχη αυτής που είδαμε στο σχήμα 2.12 του κεφαλαίου 2.3.1. Έτσι, ο zipc παρέχει ένα μονόδρομο κανάλι επικοινωνίας (half duplex) που παραδίδει τις σελίδες από τον αποστολέα στον παραλήπτη σε διάταξη FIFO, όπως ακριβώς και οι σωληνώσεις του Linux. Στη συνέχεια θα χρησιμοποιούμε τον όρο pipe για να αναφερόμαστε στη δομή σωλήνωσης που χρησιμοποιεί εσωτερικά ο μηχανισμός μας. Όπου χρειάζεται να αναφερθούμε στις σωληνώσεις του Linux θα το προσδιορίζουμε ρητά.

Το pipe έχει ένα άκρο εγγραφής και ένα άκρο ανάγνωσης. Οι σελίδες που τοποθετούνται στο άκρο εγγραφής μπορούν να διαβαστούν από το άκρο ανάγνωσης. Το pipe, σε αντίθεση με τις σωληνώσεις του Linux, δε χρησιμοποιείται για να αποθηκεύσει δεδομένα, αλλά για να μεταφέρει δείκτες σε πλαίσια μνήμης. Ο βασικός λόγος χρήσης ενός pipe είναι για να επιβάλουμε διάταξη FIFO στη μετάδοση των δεδομένων - σελίδων και για να έχουμε έναν εύκολο τρόπο συγχρονισμού των διεργασιών που επικοινωνούν. Η χωρητικότητα του pipe είναι περιορισμένη και μπορεί να μεταβληθεί κατά το χρόνο φόρτωσης του module. Η προεπιλεγμένη τιμή της είναι δεκαέξι (16) δείκτες σε πλαίσια μνήμης, όση και η προεπιλεγμένη χωρητικότητα των σωληνώσεων του Linux. Η επιλογή αυτή έγινε για να είναι άμεσα συγκρίσιμοι οι μηχανισμοί.

4.7 zipc API

Οι διεργασίες που επιθυμούν να επικοινωνήσουν με χρήση του μηχανισμού zipc πρέπει να ανοίξουν το ίδιο αρχείο συσκευής (device file). Αυτό γίνεται με χρήση της κλήσης συστήματος open. Όταν τελειώσουν με τη μεταξύ τους επικοινωνία, κλείνουν το αρχείο με χρήση της κλήσης συστήματος close. Για την επικοινωνία των διεργασιών υλοποιήσαμε τις κλήσεις συστήματος writev και readv. Η επιλογή αυτή έγινε προκειμένου να παρέχουμε στις εφαρμογές μια ήδη γνωστή προγραμματιστική διεπαφή (programming interface). Στη συνέχεια αναλύουμε τη λειτουργικότητα όλων των μεθόδων που υλοποιήσαμε.

4.7.1 Άνοιγμα του zipc

Επιτελείται με την κλήση συστήματος open, δίνοντάς της ως όρισμα το μονοπάτι, στο σύστημα αρχείων, ενός αρχείου συσκευής του zipc. Η open επιστρέφει έναν file descriptor, ο οποίος χρησιμοποιείται σε όλες τις υπόλοιπες κλήσεις συστήματος και αποτελεί, κατά μία έννοια, τον τρόπο πρόσβασης στο στιγμιότυπο του μηχανισμού (pipe) που χρησιμοποιεί η διεργασία για να επικοινωνήσει με άλλες. Για πρόσβαση στο άκρο ανάγνωσης του pipe πρέπει να χρησιμοποιηθεί η σημαία *O_RDONLY*, ενώ για πρόσβαση στο άκρο εγγραφής η σημαία *O_WRONLY*. Ένα αρχείο του zipc μπορεί να ανοιχτεί από πολλαπλές διεργασίες για ανάγνωση ή εγγραφή. Ο zipc διατηρεί ακριβώς ένα pipe για κάθε ειδικό αρχείο (πιο σωστά για κάθε minor number), που έχει ανοιχτεί από τουλάχιστον μία διεργασία. Ένα αρχείο zipc πρέπει να ανοιχτεί τόσο για ανάγνωση, όσο και για εγγραφή, για να είναι εφικτή η μεταφορά δεδομένων μέσω αυτού. Συνήθως, το άνοιγμα του αρχείου, θα μπλοκάρει τη διεργασία μέχρι να ανοιχτεί και το άλλο άκρο (εγγραφής ή ανάγνωσης). Σημειώνουμε, εδώ, ότι αν και μιλάμε για αρχεία στο σύστημα αρχείων όλες οι δομές του μηχανισμού διατηρούνται στη φυσική μνήμη του συστήματος. Το ειδικό αρχείο του zipc εξυπηρετεί απλώς ως σημείο πρόσβασης στο pipe και στη λειτουργικότητα του μηχανισμού.

Μια διεργασία μπορεί να ανοίξει ένα αρχείο του zipc σε κατάσταση nonblocking. Σε αυτή την περίπτωση, το άνοιγμα του αρχείου για ανάγνωση θα επιτύχει ακόμα κι αν δεν έχει ανοίξει κανείς το αρχείο για εγγραφή. Αντίθετα, το άνοιγμα του αρχείου για εγγραφή θα αποτύχει με κωδικό λάθους ENXIO (no such device or address), εκτός κι αν το

άκρο ανάγνωσης έχει ήδη ανοίξει. Το άνοιγμα του αρχείου για ανάγνωση **και** εγγραφή πετυχαίνει πάντα, τόσο σε κατάσταση blocking, όσο και σε κατάσταση nonblocking. Αυτή η συμπεριφορά επιτρέπει σε μια διεργασία να "γράψει" στο pipe, παρόλο που δεν υπάρχουν ακόμα αναγνώστες, καθώς και να επικοινωνήσει με τον εαυτό της.

4.7.2 Αποστολή δεδομένων

Πραγματοποιείται με την κλήση συστήματος writev:

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Το όρισμα *fd* πρέπει να είναι ο file descriptor που επιστράφηκε προηγουμένως από την open. Ο δείκτης *iov* δείχνει σε έναν πίνακα από *iovcnt* το πλήθος δομές τύπου *struct iovec*, οι οποίες ορίζονται ως εξής:

```
struct iovec {
  void *iov_base;     /* Starting address */
  size_t iov_len;     /* Number of bytes to transfer */
};
```

Η κλήση συστήματος writev καρφιτσώνει (pin) στη μνήμη τις σελίδες που περιέχουν τα δεδομένα του κάθε iovec. Στη συνέχεια απεικονίζει τις σελίδες αυτές στο pipe, στο οποίο αναφέρεται ο file descriptor *fd* (gather output) και καταργεί τις αντίστοιχες καταχωρήσεις στους πίνακες σελίδων του αποστολέα. Επόμενες προσβάσεις σε αυτές θα έχουν ως αποτέλεσμα τη δέσμευση νέων πλαισίων από τον πυρήνα, τα οποία θα περιέχουν μηδενικά. Οι buffers μεταφέρονται με τη σειρά που εμφανίζονται στον πίνακα. Δηλαδή, η writev απεικονίζει στο pipe πρώτα τις σελίδες του buffer *iov[0]*, μετά του *iov[1]*, κοκ.

Αν το pipe είναι γεμάτο, τότε η writev θα μπλοκάρει μέχρι να ελευθερωθεί επαρκής χώρος για να μπορέσει να πραγματοποιηθεί η αποστολή των σελίδων. Αν το αρχείο του zipc έχει ανοιχτεί με το *O_NONBLOCK* flag (non-blocking mode) τότε η συμπεριφορά της writev εξαρτάται από το πλήθος των ταυτόχρονων διεργασιών που στέλνουν στο

pipe και το πλήθος των σελίδων προς μεταφορά, όπως εξηγείται αναλυτικά στα επόμενα.

Σε περίπτωση επιτυχίας η writev επιστρέφει τον αριθμό των bytes που απεικονίστηκαν στο pipe, ο οποίος μπορεί να είναι μικρότερος απ' ότι ζητήθηκε από το χρήστη. Σε περίπτωση σφάλματος επιστρέφεται -1 και η μεταβλητή *errno* περιέχει τον κωδικό σφάλματος. Αν όλοι οι file descriptors που αναφέρονται στο άκρο ανάγνωσης του pipe κλείσουν, τότε μια απόπειρα αποστολής στο pipe θα έχει ως αποτέλεσμα να σταλεί στη διεργασία το σήμα (signal) SIGPIPE. Αν η διεργασία αγνοεί αυτό το σήμα, τότε η κλήση συστήματος writev θα αποτύχει με κωδικό σφάλματος EPIPE.

Atomicity Όπως οι σωληνώσεις του Linux έτσι και ο μηχανισμός μας ακολουθεί τις οδηγίες του προτύπου POSIX.1-2001 όσον αφορά την ατομικότητα της μεταφοράς δεδομένων (data transfer).

Συγκεκριμένα, μεταφορές δεδομένων μικρότερες από PIPE_BUF bytes πρέπει να είναι ατομικές, δηλαδή οι αντίστοιχες σελίδες απεικονίζονται στο pipe ως μια συνεχής ακολουθία. Μεταφορές δεδομένων μεγαλύτερες από PIPE_BUF bytes μπορεί να είναι μη ατομικές: ο πυρήνας μπορεί να παρεμβάλλει τις σελίδες που στέλνει η διεργασία με σελίδες που στέλνουν άλλες διεργασίες. Η ακριβής συμπεριφορά εξαρτάται από το αν ο file descriptor είναι nonblocking (O_NONBLOCK), από το αν υπάρχουν πολλαπλοί αποστολείς (senders) στο pipe και από τον αριθμό *n* των bytes που στέλνονται:

- Blocking συμπεριφορά, *n* <= PIPE_BUF
  Όλα τα *n* bytes αποστέλλονται ατομικά. Η κλήση συστήματος writev μπορεί να μπλοκάρει, αν δεν υπάρχει ελεύθερος χώρος για να απεικονιστούν άμεσα τα *n* bytes στο pipe.

- Blocking συμπεριφορά, *n* > PIPE_BUF
  Τα δεδομένα που δίδονται στην κλήση συστήματος writev μπορούν να παρεμβληθούν με δεδομένα από άλλες διεργασίες. Η writev μπλοκάρει μέχρι να απεικονιστούν και τα *n* bytes στο pipe.

- Nonblocking συμπεριφορά, *n* <= PIPE_BUF
  Αν υπάρχει ελεύθερος χώρος στο pipe για να απεικονιστούν *n* bytes, τότε η κλήση

συστήματος writev επιτυγχάνει άμεσα απεικονίζοντας όλα τα *n* bytes. Διαφορετικά η writev αποτυγχάνει με τον κωδικό σφάλματος EAGAIN (προσπαθήστε ξανά).

- Nonblocking συμπεριφορά, *n* > PIPE_BUF
  Αν το pipe είναι γεμάτο τότε η κλήση συστήματος writev αποτυγχάνει με τον κωδικό σφάλματος EAGAIN. Διαφορετικά, από 1 μέχρι *n* bytes μπορεί να απεικονιστούν στο pipe (με άλλα λόγια μπορεί να συμβεί μια μερική αποστολή και η διεργασία πρέπει να ελέγξει την τιμή που επιστρέφει η writev για να διαπιστώσει πόσα bytes απεικονίστηκαν τελικά στο pipe), τα οποία μπορεί να παρεμβληθούν με δεδομένα άλλων διεργασιών που στέλνουν παράλληλα στο pipe.

Η τιμή του PIPE_BUF μπορεί να καθοριστεί κατά το χρόνο φόρτωσης του module και πρέπει να είναι πολλαπλάσιο του μεγέθους μια σελίδας. Η προεπιλεγμένη τιμή του είναι μία σελίδα (4096 bytes), όπως και στις σωληνώσεις του Linux.

Οι περιοχές μνήμης, που περιγράφονται από τις δομές τύπου *struct iovec*, πρέπει να ξεκινάνε σε διευθύνσεις που είναι πολλαπλάσια του μεγέθους μιας σελίδας (PAGE_SIZE aligned) και να έχουν μέγεθος που είναι επίσης πολλαπλάσιο του μεγέθους της σελίδας. Διαφορετικά η writev αποτυγχάνει με κωδικό σφάλματος EINVAL.

Σημείωση Για την αποστολή δεδομένων μπορεί να χρησιμοποιηθεί και η κλήση συστήματος write, η οποία μεταφράζεται αυτόματα από τον πυρήνα σε κλήση της μεθόδου writev, που υλοποιεί ο zipc. Δηλαδή μια κλήση της μορφής *write(fd, buf, count)* είναι ισοδύναμη με τη *writev(fd, iov, 1)*, όπου *iov->iov_base = buf* και *iov->iov_len = count*.

4.7.3 Λήψη δεδομένων

Πραγματοποιείται με την κλήση συστήματος readv:

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

Το όρισμα *fd* πρέπει να είναι ο file descriptor που επιστράφηκε προηγουμένως από την open. Ο δείκτης *iov* δείχνει σε έναν πίνακα από *iovcnt* το πλήθος δομές τύπου *struct iovec*, οι οποίες ορίζονται ως εξής:

```
struct iovec {
  void *iov_base;      /* Starting address */
  size_t iov_len;      /* Number of bytes to transfer */
};
```

Η κλήση συστήματος readv διαβάζει από το pipe, στο οποίο αναφέρεται ο file descriptor *fd*, τις διαθέσιμες σελίδες και τις απεικονίζει στους buffers που περιγράφονται από τις δομές *iov* (scatter input). Ταυτόχρονα οι σελίδες αυτές αφαιρούνται από το pipe. Οι buffers γεμίζουν με τη σειρά που εμφανίζονται στον πίνακα. Δηλαδή, η readv γεμίζει πρώτα τον buffer *iov[0]*, μετά τον *iov[1]*, κοκ. Αν δεν υπάρχουν επαρκή δεδομένα στο pipe τότε μπορεί να μην γεμίσουν όλοι οι buffers. Η μεταφορά δεδομένων που πραγματοποιείται από τη readv είναι ατομική, δηλαδή διαβάζεται μια συνεχής ακολουθία σελίδων από το pipe, ανεξάρτητα από ταυτόχρονα reads άλλων διεργασιών ή threads, που έχουν έναν file descriptor, ο οποίος αναφέρεται στο ίδιο pipe.

Αν το pipe είναι άδειο, τότε η readv μπλοκάρει μέχρι να υπάρξουν διαθέσιμες σελίδες. Αν το αρχείο του zipc έχει ανοιχτεί με το *O_NONBLOCK* flag (non-blocking mode) τότε η readv επιστρέφει άμεσα -1 και κωδικό σφάλματος EAGAIN (προσπαθήστε ξανά).

Σε περίπτωση επιτυχίας η readv επιστρέφει τον αριθμό των bytes που διαβάστηκαν από το pipe και απεικονίστηκαν στο χώρο διευθύνσεων του παραλήπτη, ο οποίος μπορεί να είναι μικρότερος απ' ότι ζητήθηκε από το χρήστη. Σε περίπτωση σφάλματος επιστρέφεται -1 και η μεταβλητή *errno* περιέχει τον κωδικό σφάλματος. Αν όλοι οι file descriptors που αναφέρονται στο άκρο εγγραφής του pipe έχουν κλείσει, τότε η readv επιστρέφει 0 (end-of-file).

Παρατηρούμε ότι η κλήση συστήματος readv χρησιμοποιείται με τον ίδιο τρόπο που θα χρησιμοποιούνταν για ανάγνωση δεδομένων από οποιοδήποτε αρχείο ή συσκευή. Ωστόσο, υπάρχει μια σημαντική διαφορά. Οι περιοχές μνήμης, που περιγράφονται από τις δομές τύπου *struct iovec*, πρέπει να ξεκινάνε σε διευθύνσεις που είναι πολλαπλάσια του μεγέθους μιας σελίδας (PAGE_SIZE aligned) και να έχουν μέγεθος που είναι επίσης

πολλαπλάσιο του μεγέθους της σελίδας. Διαφορετικά, η readv αποτυγχάνει με κωδικό σφάλματος EINVAL. Ο λόγος ύπαρξης αυτού του περιορισμού εξηγήθηκε στην ενότητα 4.4.

Σημείωση Για τη λήψη δεδομένων μπορεί να χρησιμοποιηθεί και η κλήση συστήματος read, η οποία μεταφράζεται αυτόματα από τον πυρήνα σε κλήση της μεθόδου readv, που υλοποιεί ο zipc. Δηλαδή μια κλήση της μορφής *read(fd, buf, count)* είναι ισοδύναμη με τη *readv(fd, iov, 1)*, όπου *iov->iov_base = buf* και *iov->iov_len = count*.

4.7.4 Κλείσιμο του zipc

Όταν μια διεργασία δεν επιθυμεί πλέον να επικοινωνήσει μέσω του zipc μπορεί να καλέσει την κλήση συστήματος close για να κλείσει τον αντίστοιχο file descriptor. Αυτό, βέβαια, γίνεται αυτόματα από τον πυρήνα όταν η διεργασία τερματίσει. Όταν όλες οι διεργασίες κλείσουν τους file descriptors, που αναφέρονται σε ένα στιγμιότυπο του μηχανισμού, τότε ο zipc απελευθερώνει όλες τις δομές που χρησιμοποιούσε εσωτερικά γι' αυτό το στιγμιότυπο.

4.8 Διευθυνσιοδότηση και Ασφάλεια

Ο προορισμός ενός μηνύματος δεν καθορίζεται ρητά από τον αποστολέα. Οποιαδήποτε διεργασία έχει ανοίξει το ίδιο αρχείο συσκευής του zipc για ανάγνωση μπορεί να λάβει τα δεδομένα που στέλνει ένας αποστολέας. Εν γένει μπορεί να υπάρχουν πολλές διεργασίες - αποστολείς και πολλές διεργασίες - παραλήπτες. Ο συγχρονισμός τους ακολουθεί τους ίδιους κανόνες με τις σωληνώσεις του Linux και έχει περιγραφεί στις προηγούμενες ενότητες. Ωστόσο, δεν μπορεί οποιαδήποτε διεργασία να ανοίξει ένα αρχείο του zipc για ανάγνωση ή/και εγγραφή. Τα δικαιώματα πρόσβασης των αρχείων και των καταλόγων περιορίζουν ποιες διεργασίες μπορούν να ανοίξουν ένα αρχείο συσκευής του zipc και επομένως περιορίζουν ποιος μπορεί να επικοινωνήσει με ποιον.

4.9 Παράδειγμα χρήσης του zipc

Στα ακόλουθα σχήματα φαίνεται ένα παράδειγμα χρήσης του μηχανισμού zipc. Στο σχήμα 4.1 φαίνεται ο χώρος διευθύνσεων του αποστολέα, το pipe και ο χώρος διευθύνσεων του παραλήπτη, πριν την αποστολή των δεδομένων. Ο αποστολέας έχει ετοιμάσει δύο περιοχές μνήμης προς αποστολή, που περιγράφονται από δομές τύπου *struct iovec*. Στο σχήμα 4.2 βλέπουμε την εικόνα μετά την εκτέλεση της writev. Οι σελίδες μνήμης, με τα δεδομένα προς αποστολή, έχουν απεικονιστεί στο pipe (gather output), ενώ έχουν καταργηθεί οι απεικονίσεις τους από το χώρο διευθύνσεων του αποστολέα. Στα σχήματα 4.3, 4.4 και 4.5 βλέπουμε τρεις διαφορετικές περιπτώσεις εκτέλεσης της readv στον παραλήπτη. Οι σελίδες με τα δεδομένα έχουν μεταφερθεί από το pipe και έχουν απεικονιστεί στο χώρο διευθύνσεων του παραλήπτη (scatter input). Οι περιοχές μνήμης, στις οποίες θα απεικονιστούν οι εισερχόμενες σελίδες, περιγράφονται από τις δομές τύπου *struct iovec*, που έδωσε ο παραλήπτης ως όρισμα στη readv. Όπως φαίνεται και στα σχήματα, δε δημιουργείται κανένα αντίγραφο των μεταδιδόμενων δεδομένων.

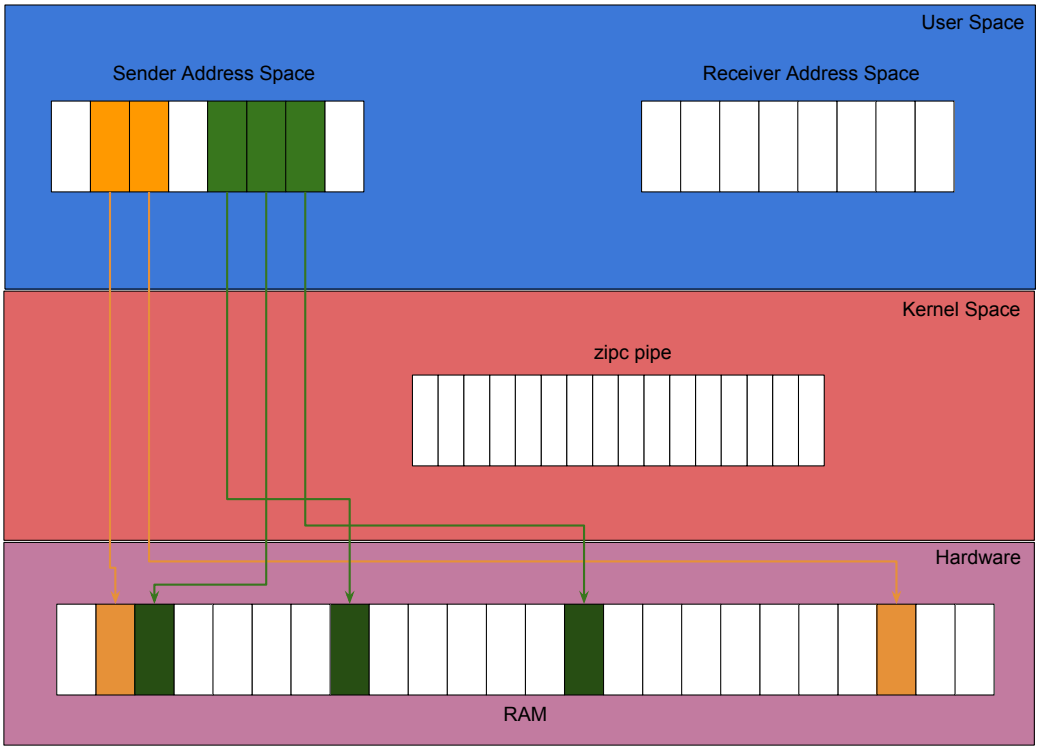
Τέλος, για καλύτερη κατανόηση των παραδειγμάτων, δίνεται ενδεικτικός ψευδοκώδικας.

```
// Sender

#define PAGE_SIZE 4096
struct iovec iov[2];
int fd, ret;

fd = open("zipc-device-file-path", O_WRONLY);

iov[0].iov_base = mmap(NULL, PAGE_SIZE * 2, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov[0].iov_len = PAGE_SIZE * 2;

iov[1].iov_base = mmap(NULL, PAGE_SIZE * 3, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov[1].iov_len = PAGE_SIZE * 3;

// fill iovecs with data

ret = writev(fd, &iov[0], 2);
```

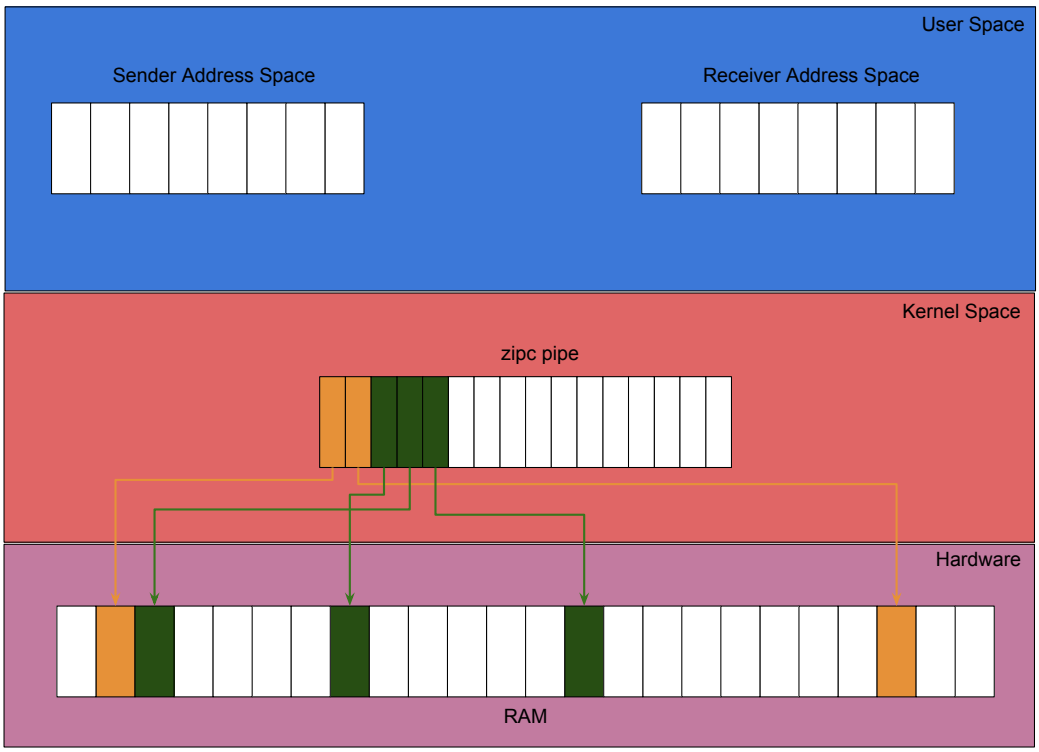
```
// Receiver

#define PAGE_SIZE 4096
int fd, ret;

fd = open("zipc-device-file-path", O_RDONLY);

// case (a)

struct iovec iov[2];

iov[0].iov_base = mmap(NULL, PAGE_SIZE * 2, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov[0].iov_len = PAGE_SIZE * 2;

iov[1].iov_base = mmap(NULL, PAGE_SIZE * 3, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov[1].iov_len = PAGE_SIZE * 3;

ret = readv(fd, &iov[0], 2);

// case (b)

struct iovec iov[2];

iov[0].iov_base = mmap(NULL, PAGE_SIZE * 3, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov[0].iov_len = PAGE_SIZE * 3;

iov[1].iov_base = mmap(NULL, PAGE_SIZE * 2, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov[1].iov_len = PAGE_SIZE * 2;

ret = readv(fd, &iov[0], 2);

// case (c)

struct iovec iov;
```

```
iov.iov_base = mmap(NULL, PAGE_SIZE * 5, PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
iov.iov_len = PAGE_SIZE * 5;

ret = readv(fd, &iov, 1);
```

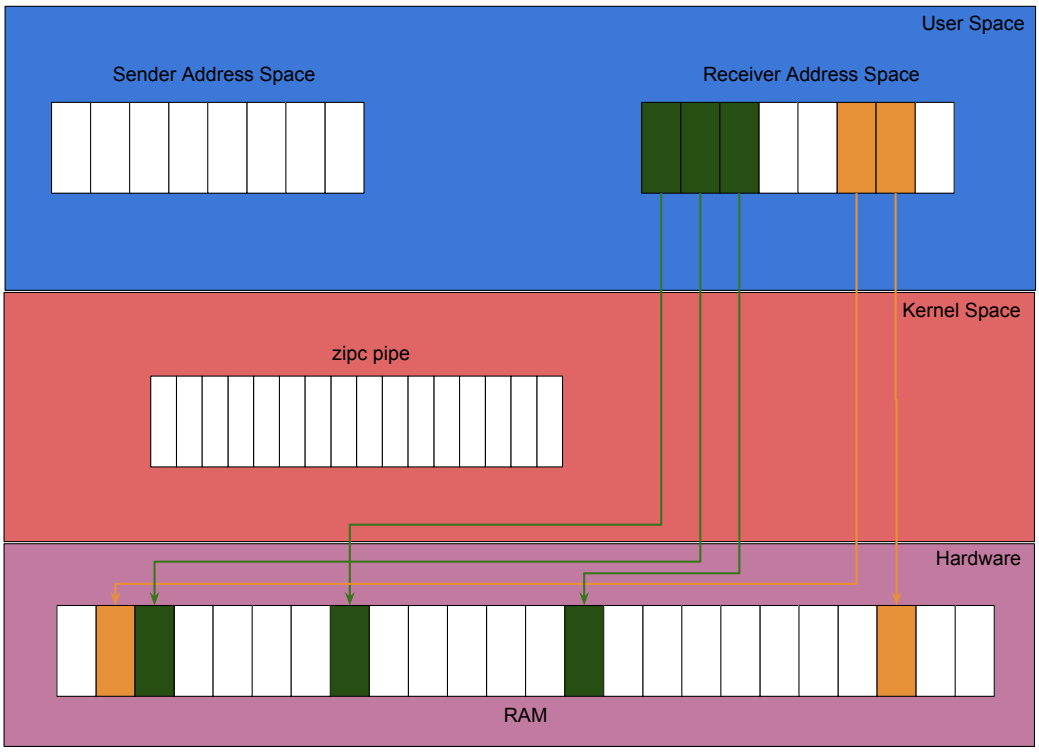
Σχήμα 4.1: *Εικόνα πριν την εκτέλεση της writev*



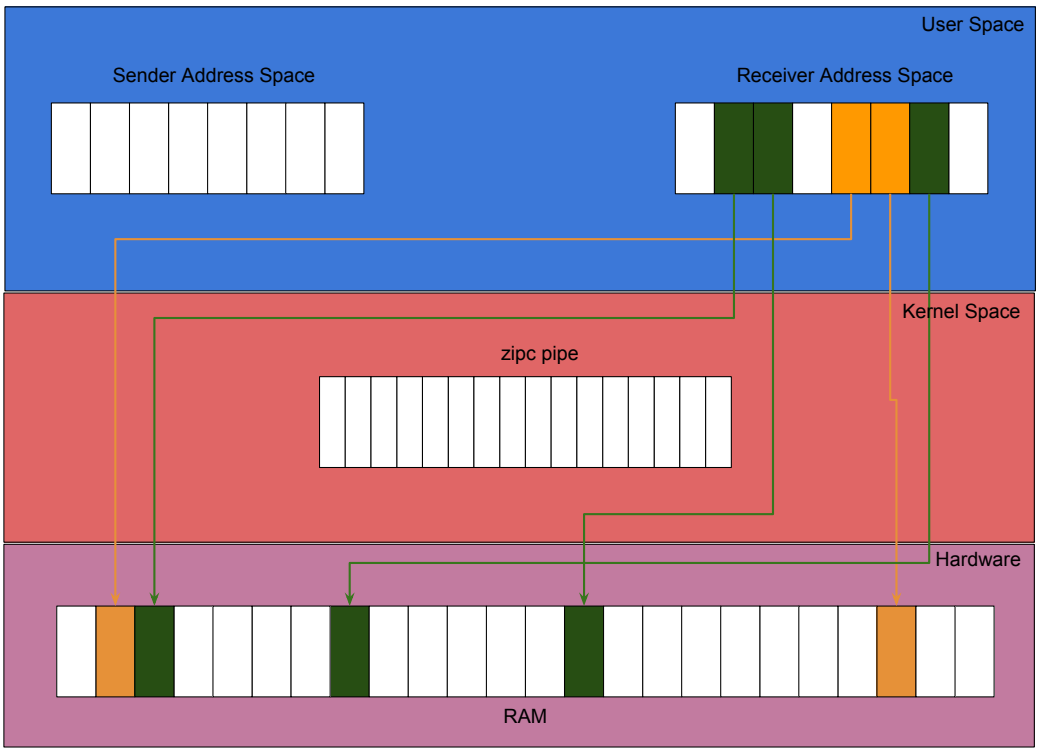
Σχήμα 4.2: *Εικόνα μετά την εκτέλεση της writev και πριν την εκτέλεση της readv*



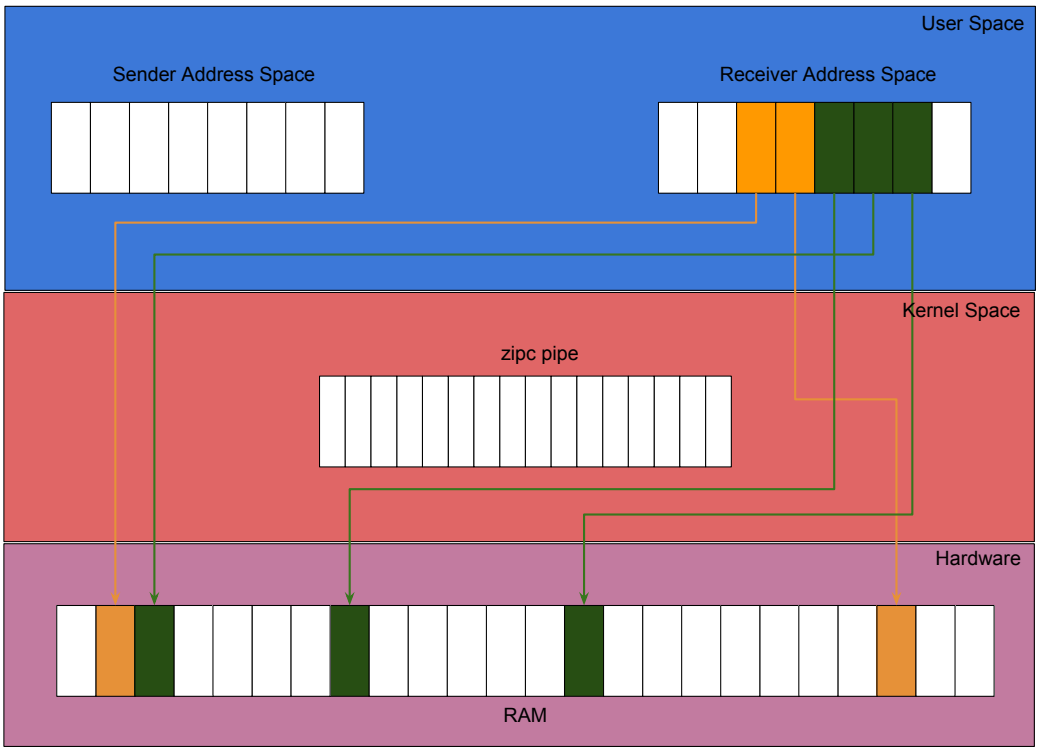
Σχήμα 4.3: *Εικόνα μετά την εκτέλεση της readv (a)*



Σχήμα 4.4: *Εικόνα μετά την εκτέλεση της readv (b)*



Σχήμα 4.5: *Εικόνα μετά την εκτέλεση της readv (c)*



5

Υλοποίηση του zipc

Στο κεφάλαιο αυτό περιγράφεται η υλοποίηση του μηχανισμού διαδιεργασιακής επικοινωνίας zipc, η σχεδίαση του οποίου εξετάστηκε λεπτομερώς στο προηγούμενο κεφάλαιο. Η υλοποίηση έχει ως στόχο την πειραματική αξιολόγηση της σχεδίασης, μέσα από τη διεξαγωγή μετρήσεων. Επίσης, μας φέρνει αντιμέτωπους με προγραμματιστικά προβλήματα, που σχετίζονται με τα υποσυστήματα του πυρήνα, αλλά και τους περιορισμούς του υλικού.

Η υλοποίηση έλαβε χώρα σε ένα ευρέως χρησιμοποιούμενο σύστημα αρχιτεκτονικής *x86-64*. Το λειτουργικό σύστημα που χρησιμοποιήθηκε είναι το Debian GNU/Linux με έκδοση πυρήνα τη *3.16.0-4-amd64*. Όλες οι συναρτήσεις, που θα δούμε στη συνέχεια της ενότητας αυτής, βασίζονται στις διεπαφές που ορίζει αυτή η έκδοση πυρήνα.

Για τη διευκόλυνση της αποσφαλμάτωσης του κώδικα και την επιτάχυνση του προγραμματισμού, κατά τα πρώτα στάδια ανάπτυξης του μηχανισμού, χρησιμοποιήσαμε μια εικονική μηχανή, στην οποία εγκαταστήσαμε το ίδιο λειτουργικό σύστημα και τον ίδιο πυρήνα με το host σύστημα. Ως hypervisor χρησιμοποιήθηκε ο qemu-kvm. Για την αποσφαλμάτωση χρησιμοποιήσαμε τον debugger Kgdb, που επιτρέπει την απομακρυσμένη αποσφαλμάτωση κώδικα πυρήνα σαν να ήταν μια απλή εφαρμογή χρήστη. Για τη χρήση του Kgdb απαιτούνται δύο μηχανήματα. Το ένα από αυτά είναι το μηχάνημα ανάπτυξης (development machine), δηλαδή το σύστημα στο οποίο γράφουμε τον κώδικα και το δεύτερο είναι το μηχάνημα - στόχος (target machine), δηλαδή το σύστημα στο οποίο θα τρέξει ο κώδικας μας. Στην προκειμένη περίπτωση το φυσικό μηχάνημα είναι το development machine και η εικονική μηχανή είναι το target machine. Χρησιμοποιώντας τον gdb στο development machine μπορούμε να συνδεθούμε στον Kgdb

στο target machine. Στη συνέχεια μπορούμε να χρησιμοποιήσουμε τον gdb, όπως και σε οποιαδήποτε εφαρμογή χρήστη (επισκόπηση μνήμης, μεταβλητών και στοίβας κλήσης, χρήση breakpoints, κτλ.), για την αποσφαλμάτωση του module μας.

5.1 Χρήση μη εξαχθέντων συναρτήσεων του πυρήνα

Κατά την υλοποίηση του μηχανισμού χρειάστηκε να χρησιμοποιήσουμε αρκετές από τις συναρτήσεις του πυρήνα που σχετίζονται με το χειρισμό της μνήμης και των πινάκων σελίδων μιας διεργασίας. Ωστόσο, όπως αναφέρθηκε και στο κεφάλαιο του σχεδιασμού, ο πυρήνας του Linux δεν εξάγει τις περισσότερες από αυτές τις συναρτήσεις για χρήση από τα modules, διότι δεν αναμένεται από ένα module να χρειαστεί αυτή τη λειτουργικότητα. Ως εκ τούτου, χρειάστηκε να βρούμε έναν τρόπο για να μπορέσουμε να χρησιμοποιήσουμε αυτές τις συναρτήσεις. Ο πυρήνας του Linux παρέχει μια συνάρτηση, την *kallsyms_lookup_name*, η οποία επιστρέφει τη διεύθυνση, στη μνήμη του πυρήνα, του αντίστοιχου συμβόλου. Η *kallsyms_lookup_name* το πετυχαίνει αυτό ψάχνοντας στον πίνακα συμβόλων του πυρήνα για τη διεύθυνση του αντίστοιχου συμβόλου. Με χρήση αυτής της συνάρτησης μπορούμε να βρούμε τη διεύθυνση οποιασδήποτε συνάρτησης του πυρήνα επιθυμούμε, ανεξάρτητα από το αν εξάγεται για χρήση από τον έξω κόσμο, δηλαδή από τα modules, ή όχι. Δίνουμε ένα παράδειγμα:

```
void (*lru_add_drain)(void);

lru_add_drain = (void *) kallsyms_lookup_name("lru_add_drain");

lru_add_drain();
```

Το μειονέκτημα της χρήσης μη εξαχθέντων συναρτήσεων είναι ότι υπάρχει ο κίνδυνος οι διεπαφές αυτές να αλλάξουν σε επόμενες εκδόσεις του πυρήνα, με αποτέλεσμα να σταματήσει να μεταγλωττίζεται ή/και να δουλεύει σωστά ο κώδικας μας.

5.2 Buffers Αποστολέα και Παραλήπτη

Οι buffers προέλευσης και προορισμού των δεδομένων πρέπει να είναι ευθυγραμμισμένοι στη μνήμη στο όριο μιας σελίδας, δηλαδή να έχουν διεύθυνση έναρξης πολλαπλάσιο του μεγέθους μιας σελίδας. Επίσης, το μέγεθός τους πρέπει να είναι και αυτό πολλαπλάσιο του μεγέθους μιας σελίδας. Οι λόγοι γι᾽ αυτούς του περιορισμούς εξηγήθηκαν στο κεφάλαιο σχεδιασμού του zipc. Στη συνέχεια παραθέτουμε τον κώδικα με τον οποίο ο zipc ελέγχει ότι ένα σύνολο από buffers, που περιγράφονται από έναν πίνακα δομών τύπου *struct iovec*, ικανοποιεί τους περιορισμούς:

```
/*
   buffers_aligned ()
    Returns 1 if the buffers are page aligned both in memory and length, 0 otherwise
*/

static int buffers_aligned (const struct iovec *iov, unsigned long nr_segs)
{
  while (nr_segs) {
    /*
      Alignment restriction : offset must be zero and length a multiple of the PAGE_SIZE
    */

    if ((unsigned long) iov->iov_base & ~PAGE_MASK || iov->iov_len & ~PAGE_MASK)
      return 0;

    iov++;
    nr_segs--;
  }

  return 1;
}
```

5.3 Προέλευση των δεδομένων - Μνήμη Αποστολέα

Όπως εξηγήσαμε στο κεφάλαιο σχεδιασμού του zipc, τα μόνα πλαίσια μνήμης, που δεχόμαστε ως μνήμη προέλευσης των δεδομένων, είναι τα ιδιωτικά ανώνυμα πλαίσια. Ο

έλεγχος για το κατά πόσο ένα πλαίσιο μνήμης εμπίπτει σε αυτή τη κατηγορία γίνεται με τον ακόλουθο κώδικα:

```
static inline int check_page(struct page *page)
{
  if ( unlikely (!PageAnon(page) || (page_mapcount(page) != 1) || PageReserved(page) ||
      PageMlocked(page) || PageCompound(page)))
    return 0;

  return 1;
}
```

5.4 Προορισμός των δεδομένων - Μνήμη παραλήπτη

Οι μόνες έγκυρες περιοχές μνήμης που μπορούν να φιλοξενούν τους buffers του παραλήπτη, όπως είδαμε στο προηγούμενο κεφάλαιο, είναι οι ιδιωτικές ανώνυμες περιοχές εικονικής μνήμης. Ο έλεγχος για την εγκυρότητα μιας περιοχής εικονικής μνήμης γίνεται με τον παρακάτω κώδικα:

```
#define VM_FLAGS_BAD (VM_SPECIAL | VM_LOCKED | VM_SHARED | VM_HUGEPAGE)

static int verify_vma(struct vm_area_struct *vma)
{
  if (vma->vm_flags & VM_FLAGS_BAD || vma->vm_file || is_vm_hugetlb_page(vma))
    return -EINVAL;

  return 0;
}
```

5.5 Υλοποίηση του pipe

Το pipe υλοποιείται ως ένας κυκλικός buffer με προκαθορισμένο μέγεθος δεκαέξι καταχωρήσεις, κάθε μία από τις οποίες περιέχει ένα δείκτη προς μια σελίδα φυσικής μνήμης. Ακολουθεί ο σχετικός κώδικας:

```
#define PIPE_DEF_BUFFERS 16

/*
 * struct pipe_buffer - a pipe buffer
 * @page: the page containing the data for the pipe buffer
 * @ops: operations associated with this buffer - NULL if buffer is empty
 */

struct pipe_buffer {
  struct page *page;
  const struct pipe_buf_operations *ops;
};

/*
 * struct pipe_info - a pipe
 * @mutex: mutex protecting the pipe
 * @wait: reader/writer wait point in case of empty/full pipe
 * @nrbufs: the number of non-empty pipe buffers in this pipe
 * @buffers: total number of buffers (power of 2)
 * @curbuf: the current pipe buffer entry
 * @readers: number of current readers of this pipe
 * @writers: number of current writers of this pipe
 * @r_counter: used for syncing with writers in zipc_open()
 * @w_counter: used for syncing with readers in zipc_open()
 * @files: number of struct file referring this pipe (protected by inode->i_lock)
 * @waiting_writers: number of writers blocked waiting for room
 * @bufs: the circular array of pipe buffers
 */

struct pipe_info {
  struct mutex mutex;
  wait_queue_head_t wait;
  unsigned int nrbufs, curbuf, buffers;
  unsigned int readers;
  unsigned int writers;
  unsigned int r_counter;
  unsigned int w_counter;
  unsigned int files;
  unsigned int waiting_writers;
  struct pipe_buffer *bufs;
```

```
};

struct pipe_buf_operations {
  /*
    When the contents of this pipe buffer has been consumed by a
    reader, ->release() is called.
  */
  void (* release )( struct pipe_info *, struct pipe_buffer *);

  /*
    Get a reference to the pipe buffer.
  */
  void (* get )( struct pipe_info *, struct pipe_buffer *);
};
```

5.6 Υλοποίηση λειτουργικότητας Αποστολέα

Όπως αναφέρθηκε στο κεφάλαιο του σχεδιασμού, προκειμένου να διατηρήσουμε τη σημασιολογία της κλήσης συστήματος write (copy semantics), όταν ο αποστολέας στέλνει κάποια δεδομένα με χρήση του zipc, καταργούνται από τους πίνακες σελίδων του οι καταχωρήσεις για τις σελίδες φυσικής μνήμης που περιέχουν τα δεδομένα αυτά. Στη συνέχεια οι σελίδες αυτές απεικονίζονται στο pipe. Οι σχετικές περιοχές εικονικής μνήμης του αποστολέα δεν πειράζονται.

Η αρχική μας προσέγγιση ήταν να υλοποιήσουμε ακριβώς αυτή τη λειτουργικότητα. Ωστόσο, στην πλειοψηφία των περιπτώσεων, ο αποστολέας χρησιμοποιεί ξανά τους buffers που περιείχαν τα δεδομένα προς αποστολή, συνήθως για την αποστολή ενός νέου μηνύματος. Αυτό έχει ως αποτέλεσμα κάθε επόμενη πρόσβαση στους buffers να προκαλεί ένα σφάλμα σελίδας (page fault). Όταν συμβεί ένα σφάλμα σελίδας ο έλεγχος μεταφέρεται στη συνάρτηση χειρισμού σφαλμάτων σελίδας (page fault handler) του πυρήνα, η οποία δεσμεύει ένα καινούριο πλαίσιο μνήμης και το απεικονίζει στους πίνακες σελίδων της διεργασίας. Άρα, θεωρώντας ότι ο αποστολέας επαναχρησιμοποιεί τους buffers αποστολής, συμβαίνουν τόσα page fault, όσο το πλήθος των σελίδων που στάλθηκαν προηγουμένως. Κάθε page fault, πέραν του κόστους της δέσμευσης μνήμης

και της ενημέρωσης των πινάκων σελίδων, εμπεριέχει το κόστος της μετάβασης από το χώρο χρήστη στο χώρο πυρήνα και αντίστροφα, το οποίο δεν είναι αμελητέο.

Αποφασίσαμε, λοιπόν, να βελτιώσουμε την υλοποίησή μας και να δεσμεύουμε προκαταβολικά ένα καινούριο πλαίσιο μνήμης για κάθε πλαίσιο που μεταφέρεται από τον αποστολέα στο pipe. Το καινούριο πλαίσιο γεμίζεται με μηδενικά και απεικονίζεται στους πίνακες σελίδων του αποστολέα. Δεν μπορούμε, προφανώς, να αποφύγουμε το κόστος της δέσμευσης μνήμης και της ενημέρωσης των πινάκων σελίδων, αλλά γλιτώνουμε τη μετάβαση από το χώρο χρήστη στο χώρο πυρήνα και πίσω, για εκείνες τις εφαρμογές που επαναχρησιμοποιούν τους buffers αποστολής. Αυτό βελτιώνει τις επιδόσεις της υλοποίησης μας. Επομένως, η αποστολή κάποιων σελίδων, με χρήση του zipc, περιλαμβάνει τα εξής βήματα:

1. Απεικόνιση των σελίδων στο pipe.
2. Κατάργηση των απεικονίσεων για τις σελίδες αυτές από τους πίνακες σελίδων του αποτολέα.
3. Δέσμευση νέων πλαισίων μνήμης και απεικόνιση τους στους πίνακες σελίδων του αποτολέα.

Ωστόσο, και αυτή η προσέγγιση έχει ένα σημαντικό μειονέκτημα. Διατρέχουμε δύο φορές τους πίνακες σελίδων της διεργασίας - αποστολέα. Μία για την κατάργηση των απεικονίσεων των σελίδων που μεταφέρθηκαν στο pipe και μία για την εγκατάσταση των απεικονίσεων των νέων πλαισίων. Προκειμένου να βελτιώσουμε την υλοποίηση μας συνδυάσαμε αυτές τις δύο ενέργειες και τις πραγματοποιούμε με ένα πέρασμα των πινάκων σελίδων της διεργασίας - αποστολέα.

Για την υλοποίηση αυτής της λειτουργικότητας χρειάστηκε να τροποποιήσουμε κάποιες από τις συναρτήσεις που παρέχει ο πυρήνας για χειρισμό των πινάκων σελίδων. Συγκεκριμένα, ο πυρήνας παρέχει τη συνάρτηση *zap_page_range*, η οποία καταργεί τις εγγραφές στους πίνακες σελίδων μιας διεργασίας για ένα εύρος εικονικών διευθύνσεων. Τροποποιήσαμε τη συνάρτηση αυτή ούτως ώστε να δέχεται ως όρισμα μία άλλη συνάρτηση τύπου *pte_fn_t*. Ο τύπος αυτός ορίζεται ως εξής:

```
typedef int (*pte_fn_t)(pte_t *pte, pgtable_t token, unsigned long addr, void *data);
```

Μια συνάρτηση τύπου *pte_fn_t* δέχεται ως όρισμα, μεταξύ άλλων, έναν δείκτη σε μια καταχώρηση ενός πίνακα σελίδων (page table entry). Μπορεί, επομένως, να ενημερώσει αυτή την καταχώρηση. Ως εκ τούτου, τροποποιήσαμε την *zap_page_range* ούτως ώστε μόλις καταργεί μια καταχώρηση σε έναν πίνακα σελίδων να καλεί μια συνάρτηση τύπου *pte_fn_t*, με όρισμα το page table entry που μόλις καταργήθηκε. Ονομάσαμε την τροποποιημένη συνάρτηση *zap_reset_page_range*. Παρέχοντας, λοιπόν, στην *zap_reset_page_range* μια κατάλληλη συνάρτηση μπορούμε να απεικονίζουμε τα νέα πλαίσια αμέσως μόλις καταργηθούν οι απεικονίσεις για τα παλιά. Έτσι, όχι μόνο γλιτώνουμε το δεύτερο πέρασμα των πινάκων σελίδων, βελτιώνουμε και τη χρησιμοποίηση της κρυφής μνήμης του επεξεργαστή (temporal locality).

Ο κώδικας της *zap_reset_page_range* έχει μεγάλη έκταση γι αυτό δεν τον παραθέτουμε εδώ. Παραθέτουμε μόνο τον κώδικα της συνάρτησης τύπου *pte_fn_t* που αναλαμβάνει τη δέσμευση νέων πλαισίων γεμάτων μηδενικά και την απεικόνιση τους στη θέση των παλιών:

```
/*
   pte_fn_prealloc () - callback function to use with zap_reset_page_range ()

   It maps new zeroed pages back to the writer's address space in order to avoid subsequent
       page faults

   zap_reset_page_range () takes the page table lock before calling pte_fn_prealloc ()

   A return value of 0 means zap_reset_page_range () continues walking the page tables.
       -errno means zap_reset_page_range () stops immediately
*/

int pte_fn_prealloc(pte_t *pte, pgtable_t token, unsigned long addr, void *data)
{
   struct vm_area_struct *vma;
   struct page *page;
   pte_t entry;

   /*
     Find the address of unexported kernel functions
   */

   int (*mem_cgroup_charge_anon)(struct page *, struct mm_struct *, gfp_t);
```

```
int (*anon_vma_prepare)(struct vm_area_struct *);
void (*add_mm_counter_fast)(struct mm_struct *, int, int);
void (*page_add_new_anon_rmap)(struct page *, struct vm_area_struct *, unsigned long);

mem_cgroup_charge_anon = (void *) mem_cgroup_charge_anon_addr;
anon_vma_prepare = (void *) anon_vma_prepare_addr;
add_mm_counter_fast = (void *) add_mm_counter_fast_addr;
page_add_new_anon_rmap = (void *) page_add_new_anon_rmap_addr;


vma = ((struct ptable_walk_info *) data)->vma;

/*
  do_anonymous_page()
*/

/*
  Check if we need to add a guard page to the stack
*/
if (check_stack_guard_page(vma, addr) < 0)
  return 0;

if ( unlikely (anon_vma_prepare(vma)))
  return 0;

page = alloc_page(__GFP_ZERO | GFP_HIGHUSER_MOVABLE);

if (!page)
  return 0;

/*
  The memory barrier inside __SetPageUptodate makes sure that preceeding stores to the
      page contents become visible before the set_pte_at() write.
*/

__SetPageUptodate(page);

if (mem_cgroup_charge_anon(page, vma->vm_mm, GFP_KERNEL)) {
  __free_page(page);
  return 0;
```

```
  }

  entry = mk_pte(page, vma->vm_page_prot);
  if (vma->vm_flags & VM_WRITE)
    entry = pte_mkwrite(pte_mkdirty(entry));

  add_mm_counter_fast(vma->vm_mm, MM_ANONPAGES, 1);

  /*
    page_add_new_anon_rmap()
  */

  page_add_new_anon_rmap(page, vma, addr);

  /*
    end of page_add_new_anon_rmap()
  */

  set_pte_at (vma->vm_mm, addr, pte, entry);

  /* No need to invalidate - it was non-present before */
  update_mmu_cache(vma, addr, pte);

  /*
    end of do_anonymous_page()
  */

  return 0;
}
```

Ένα άλλο σημείο, άξιο αναφοράς, είναι το πώς μπορούμε να βρούμε τα πλαίσια μνήμης που αντιστοιχούν σε ένα εύρος εικονικών διευθύνσεων, προκειμένου να τα απεικονίσουμε στο pipe. Εδώ απαιτείται προσοχή διότι οι σελίδες που περιέχουν τα δεδομένα προς αποστολή μπορεί να μη βρίσκονται στη φυσική μνήμη του συστήματος, αλλά σε μια περιοχή swap (swap area) στο δίσκο. Επειδή αυτή είναι μια συνηθισμένη λειτουργία για τους οδηγούς πολλών συσκευών, ο πυρήνας παρέχει για το σκοπό αυτό τη συνάρτηση *get_user_pages*, η οποία καρφιτσώνει (pin) ένα σύνολο σελίδων μιας διεργασίας.

Το πρότυπο αυτής είναι:

```
long get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned long start,
    unsigned long nr_pages, int write, int force, struct page **pages, struct
    vm_area_struct **vmas);
```

Η παράμετρος *start* περιέχει την αρχική διεύθυνση, στο χώρο εικονικών διευθύνσεων της διεργασίας και η παράμετρος *nr_pages* προσδιορίζει τον αριθμό των σελίδων που θέλουμε να καρφιτσώσουμε (pin). Μετά την επιτυχή ολοκλήρωση αυτής της συνάρτησης, ο πίνακας *pages* περιέχει δείκτες στις δομές *struct page*, που περιγράφουν αυτό το εύρος διευθύνσεων. Όταν τελειώσουμε με τις σελίδες και δεν τις χρειαζόμαστε πλέον, απαιτείται να τις απελευθερώσουμε με κλήση της συνάρτησης *page_cache_release*.

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση *get_user_pages_fast*, η οποία παρέχει την ίδια λειτουργικότητα με την *get_user_pages*, αλλά είναι βελτιστοποιημένη για τη συνηθισμένη περίπτωση, όπου οι σελίδες μνήμης είναι παρόντες στους πίνακες σελίδων και δεν χρειάζεται να φορτωθούν από το δίσκο.

Με βάση αυτές τις συναρτήσεις υλοποιήσαμε τη συνάρτηση *get_iovec_page_array*, η οποία καρφιτσώνει τις σελίδες ενός συνόλου από buffers, που περιγράφεται από έναν πίνακα δομών *struct iovec*:

```
/*
 get_iovec_page_array() — Map an iovec array into an array of pages.
 @iov: pointer to the iovec array
 @nr_vecs: number of iovecs in @iov
 @pages: array that receives pointers to the pages pinned
         Should be at least max_nr_pages long
 @max_nr_pages: maximum number of pages to pin
 Returns number of pages pinned or a negative error code
*/

static int get_iovec_page_array(const struct iovec *iov, unsigned long nr_vecs, struct
    page **pages, unsigned int max_nr_pages)
{
  int page_cnt, error;

  page_cnt = 0;
  error = 0;
```

```
while (nr_vecs) {
  unsigned long npages;
  unsigned long base;
  size_t len;

  base = (unsigned long) iov->iov_base;
  len = iov->iov_len;

  /*
    Alignment restriction : offset must be zero and length a multiple of the PAGE_SIZE
  */

  error = -EINVAL;
  if (base & ~PAGE_MASK || len & ~PAGE_MASK)
    break;

  npages = len >> PAGE_SHIFT;

  if ( unlikely (npages > max_nr_pages - page_cnt))
    npages = max_nr_pages - page_cnt;

  error = get_user_pages_fast (base, npages, 1,
            &pages[page_cnt]);

  if ( unlikely ( error <= 0))
    break;

  page_cnt += error;

  /*
    Don't continue if we mapped fewer pages than we asked for, or if we mapped the max
        number of pages that we have room for
  */

  if (error < npages || page_cnt == max_nr_pages)
    break;

  nr_vecs--;
  iov++;
```

```
  }

  if ( likely (page_cnt))
    return page_cnt;

  return error;
}
```

5.7 Υλοποίηση λειτουργικότητας Παραλήπτη

Όπως αναφέρθηκε στο κεφάλαιο του σχεδιασμού, προκειμένου να διατηρήσουμε τη σημασιολογία της κλήσης συστήματος read (copy semantics), οι εισερχόμενες σελίδες φυσικής μνήμης απεικονίζονται στο χώρο διευθύνσεων του παραλήπτη. Ταυτόχρονα καταργούνται οι απεικονίσεις σελίδων που μπορεί να ήταν ήδη απεικονισμένες εκεί. Επομένως, η κλήση συστήματος readv περιλαμβάνει τα εξής βήματα:

1. Κατάργηση των απεικονίσεων "παλαιών" σελίδων που είναι απεικονισμένες στους buffers που θα τοποθετηθούν οι εισερχόμενες σελίδες.

2. Απεικόνιση των εισερχόμενων σελίδων στις διευθύνσεις που προσδιορίζουν οι buffers, που παρέχει ο χρήστης στην κλήση συστήματος readv.

3. Αφαίρεση των εισερχόμενων σελίδων, που απεικονίστηκαν στον παραλήπτη, από το pipe.

Όπως και στην περίπτωση του αποστολέα, έτσι κι εδώ, συνδυάζουμε τα δύο πρώτα βήματα, ούτως ώστε να εκτελέσουμε ένα μόνο πέρασμα των πινάκων σελίδων του παραλήπτη. Στην περίπτωση αυτή δίνουμε ως όρισμα στην *zap_reset_page_range* τη συνάρτηση *pte_fn*, η οποία απεικονίζει στο page table entry, που της δίνεται ως όρισμα, την επόμενη διαθέσιμη σελίδα στο pipe:

```
/*
  pte_fn () — callback function to use with zap_reset_page_range ()
```

```
   zap_reset_page_range () must be called with the pipe lock held in order to use this
        callback .

   zap_reset_page_range () takes the page table lock before calling pte_fn ()

   A return value of 0 means zap_reset_page_range () continues walking the page tables .
        -errno means zap_reset_page_range () stops immediately
*/

static int pte_fn(pte_t *pte, pgtable_t token, unsigned long addr, void *data)
{
   struct ptable_walk_info *ptw_info;
   struct pipe_info *pipe;
   struct vm_area_struct *vma;
   struct pipe_buffer *buf;
   pte_t entry;
   struct anon_vma *anon_vma;
   pgoff_t pgoff;

   /*
     Find the address of unexported kernel functions
   */

   int (*mem_cgroup_charge_anon)(struct page *, struct mm_struct *, gfp_t);
   int (*anon_vma_prepare)(struct vm_area_struct *);
   void (*add_mm_counter_fast)(struct mm_struct *, int, int);

   mem_cgroup_charge_anon = (void *) mem_cgroup_charge_anon_addr;
   anon_vma_prepare = (void *) anon_vma_prepare_addr;
   add_mm_counter_fast = (void *) add_mm_counter_fast_addr;


   ptw_info = data;
   pipe = ptw_info->pipe;
   vma = ptw_info->vma;

   buf = pipe->bufs + pipe->curbuf;

   /*
     do_anonymous_page()
```

```
*/

/*
  Check if we need to add a guard page to the stack
*/
if (check_stack_guard_page(vma, addr) < 0)
  return -EFAULT;

if ( unlikely (anon_vma_prepare(vma)))
  return -ENOMEM;

/*
  The memory barrier inside __SetPageUptodate makes sure that preceeding stores to the
      page contents become visible before the set_pte_at () write .
*/
__SetPageUptodate(buf->page);

if (mem_cgroup_charge_anon(buf->page, vma->vm_mm, GFP_KERNEL))
  return -ENOMEM;

entry = mk_pte(buf->page, vma->vm_page_prot);
if (vma->vm_flags & VM_WRITE)
  entry = pte_mkwrite(pte_mkdirty(entry));

add_mm_counter_fast(vma->vm_mm, MM_ANONPAGES, 1);

/*
  page_add_new_anon_rmap()
*/

SetPageSwapBacked(buf->page);
atomic_set(&buf->page->_mapcount, 0); /* increment count ( starts at -1) */

__mod_zone_page_state(page_zone(buf->page), NR_ANON_PAGES,
      hpage_nr_pages(buf->page));

/*
  __page_set_anon_rmap()
*/
```

```
    anon_vma = (void *) vma->anon_vma + PAGE_MAPPING_ANON;
    buf->page->mapping = (struct address_space *) anon_vma;

    pgoff = (addr - vma->vm_start) >> PAGE_SHIFT;
    pgoff += vma->vm_pgoff;
    buf->page->index = pgoff >> (PAGE_CACHE_SHIFT - PAGE_SHIFT);

    /*
      end of __page_set_anon_rmap()
    */

    /*
      end of page_add_new_anon_rmap()
    */

    set_pte_at (vma->vm_mm, addr, pte, entry);

    /* No need to invalidate - it was non-present before */
    update_mmu_cache(vma, addr, pte);

    /*
      end of do_anonymous_page()
    */

    buf->ops = NULL;
    pipe->curbuf = (pipe->curbuf + 1) & (pipe->buffers - 1);
    pipe->nrbufs--;

    ptw_info->pages_mapped++;

    return 0;
}
```

6 Σχεδιασμός του zipc-tmpfs

Στο κεφάλαιο αυτό θα ασχοληθούμε με το σχεδιασμό του zipc-tmpfs, ενός ακόμα μηχανισμού διαδιεργασιακής επικοινωνίας μηδενικών αντιγράφων. Κι αυτός ο μηχανισμός βασίζεται στη δωρεά σελίδων εικονικής μνήμης από τον αποστολέα στον παραλήπτη. Ωστόσο, έχει ουσιώδεις διαφορές από τον zipc. Πρόκειται, ουσιαστικά, για παραλλαγή της τεχνικής μεταφοράς δεδομένων map-and-read που αναλύσαμε στο κεφάλαιο 2.3.

Σε υψηλό επίπεδο ο μηχανισμός αυτός μεταφέρει τα δεδομένα ενός μηνύματος από την εμβέλεια μίας διεργασίας στην εμβέλεια μιας άλλης, καταργώντας τις αντίστοιχες απεικονίσεις από το χώρο διευθύνσεων του αποστολέα και φτιάχνοντας νέες απεικονίσεις για τα δεδομένα στο χώρο διευθύνσεων του παραλήπτη. Κατά την υλοποίηση αυτής της λειτουργικότητας προκύπτουν διάφορες δυσκολίες, οι οποίες προέρχονται από περιορισμούς που θέτει τόσο το υλικό, όσο και ο πυρήνας του Linux.

Στη συνέχεια παρουσιάζουμε όλες τις δομές και λειτουργίες που απαιτούνται για την υλοποίηση του μηχανισμού αυτού. Επίσης, παρουσιάζουμε αναλυτικά όλες τις δυσκολίες και τις αποφάσεις που χρειάστηκε να λάβουμε κατά τη διάρκεια του σχεδιασμού του zipc-tmpfs.

6.1 Γενικά

Ο zipc πετυχαίνει τη μεταφορά των δεδομένων καταργώντας τις απεικονίσεις, για τα υποκείμενα πλαίσια μνήμης, από τους πίνακες σελίδων του αποστολέα και εγκαθιστώντας νέες απεικονίσεις γι᾽ αυτά στους πίνακες σελίδων του παραλήπτη. Ωστόσο, δεν

είναι πάντα επιθυμητό να ενημερώνουμε τους πίνακες σελίδων μιας διεργασίας, για να πετύχουμε τη μετάδοση των δεδομένων. Ενδέχεται, για παράδειγμα, μια διεργασία - παραλήπτης να μην επιθυμεί να διαβάσει ή/και τροποποιήσει τα δεδομένα που λαμβάνει αλλά να έχει σκοπό να τα προωθήσει, στο σύνολο τους ή μέρη από αυτά, σε κάποια άλλη διεργασία. Η απόφαση, για το ποιο κομμάτι των δεδομένων θα προωθηθεί και σε ποιον, θα μπορούσε να λαμβάνεται με χρήση κάποιας ξεχωριστής κεφαλίδας (header), η οποία θα συνοδεύει τα δεδομένα. Στην περίπτωση αυτή, η ενημέρωση των πινάκων σελίδων του παραλήπτη είναι περιττή, μιας και η διεργασία δε θα χρειαστεί ποτέ να προσπελάσει τις αντίστοιχες σελίδες. Θα ήταν, λοιπόν, επιθυμητό να αποφύγουμε το κόστος της ενημέρωσης των πινάκων σελίδων σε τέτοιες περιπτώσεις. Ταυτόχρονα, όμως, θα πρέπει να υπάρχει η δυνατότητα μια διεργασία - παραλήπτης να μπορεί να διαβάσει ή/και τροποποιήσει τα δεδομένα που λαμβάνει. Χρειαζόμαστε, λοιπόν, έναν τρόπο να απεικονίζουμε πλαίσια μνήμης στο χώρο διευθύνσεων μιας διεργασίας, χωρίς να ενημερώνουμε εξ' αρχής τους πίνακες σελίδων. Αυτό μπορεί να επιτευχθεί με χρήση του μηχανισμού απεικόνισης αρχείων (file memory mapping) του Linux, όπως εξηγείται αναλυτικά στη συνέχεια του κεφαλαίου.

Επιλέξαμε να υλοποιήσουμε το zipc-tmpfs ως μια συσκευή χαρακτήρων. Οι λόγοι πίσω από αυτή την επιλογή, καθώς και τα πλεονέκτημα και τα μειονεκτήματα της, είναι ίδιοι με την περίπτωση του zipc και περιγράφηκαν στην ενότητα 4.1.

6.2 Προέλευση των δεδομένων - Μνήμη Αποστολέα

Όπως αναφέρθηκε στη προηγούμενη ενότητα, επιθυμούμε να απεικονίζουμε δεδομένα στο χώρο διευθύνσεων μιας διεργασίας, χωρίς να ενημερώνουμε εξ' αρχής τους αντίστοιχους πίνακες σελίδων. Ο μόνος τρόπος για να επιτευχθεί αυτό είναι με χρήση του μηχανισμού απεικόνισης αρχείων του Linux. Όταν απεικονίζουμε ένα αρχείο στο χώρο διευθύνσεων μιας διεργασίας, οι αντίστοιχοι πίνακες σελίδων ενημερώνονται κατά απαίτηση (on demand), όταν η διεργασία προσπελάσει το αρχείο. Τότε δεσμεύεται ένα πλαίσιο μνήμης, γεμίζεται με τα αντίστοιχα δεδομένα από το δίσκο και ενημερώνεται τόσο η page cache του αρχείου, όσο και οι πίνακες σελίδων της διεργασίας που προσπέλασε τα δεδομένα. Ωστόσο, η προσπέλαση δεδομένων στο δίσκο είναι ιδιαίτερα χρονοβόρα και στην περίπτωση ενός μηχανισμού διαδιεργασιακής επικοινωνίας ούτε απαιτείται

ούτε είναι επιθυμητή. Η μόνη λύση, επομένως, είναι η χρήση του ειδικού συστήματος αρχείων *tmpfs*. Όπως εξηγήθηκε στο κεφάλαιο 2.3.4, τα αρχεία του *tmpfs* δεν αντιστοιχούν σε πραγματικά αρχεία στον δίσκο, αλλά βρίσκονται εξ' ολοκλήρου σε πλαίσια φυσικής μνήμης.

Όταν μια διεργασία στέλνει δεδομένα σε μια άλλη, χρησιμοποιώντας τον μηχανισμό zipc-tmpfs, δωρίζει, ουσιαστικά, τις υποκείμενες σελίδες μνήμης στον παραλήπτη. Για να μπορεί να δωρίσει, όμως, αυτές τις σελίδες θα πρέπει να είναι ο αποκλειστικός κάτοχος τους. Θα ήταν σημασιολογικά λάθος να επιτρέπαμε στον αποστολέα να δωρίσει σελίδες που δεν ανήκουν μόνο σε αυτόν. Τα αρχεία στο *tmpfs*, όμως, μπορούν να προσπελαστούν και να απεικονιστούν από οποιονδήποτε έχει τα κατάλληλα διακαιώματα. Πρέπει να εξασφαλίσουμε ότι ένα αρχείο στο *tmpfs*, που χρησιμοποιείται ως buffer αποστολής με τον μηχανισμό μας, δεν μπορεί να είναι απεικονισμένο σε πολλαπλές διεργασίες. Αποφασίσαμε, λοιπόν, να επιβάλουμε στις διεργασίες να χρησιμοποιούν μια ειδική συνάρτηση δέσμευσης μνήμης. Η υλοποίηση αυτής της συνάρτησης θα γίνει μέσω της κλήσης συστήματος ioctl. Η συνάρτηση αυτή θα δημιουργεί ένα μη ορατό αρχείο (unlinked file) στο *tmpfs* και θα το απεικονίζει στο χώρο διευθύνσεων της διεργασίας που την καλεί. Ως εκ τούτου, καμία άλλη διεργασία δε θα μπορεί να το ανοίξει και να το απεικονίσει στο χώρο διευθύνσεων της. Για τον ίδιο λόγο, φροντίζουμε η μνήμη που γίνεται allocate με τη συνάρτησή μας να μη μπορεί να κληρονομηθεί μέσω της κλήσης συστήματος fork. Διαφορετικά, μπορεί περισσότερες από μία διεργασίες να μοιράζονται τα υποκείμενα πλαίσια μνήμης. Κατά τη χρήση του zipc-tmpfs, για αποστολή δεδομένων, ελέγχουμε ότι ο buffer αποστολής υποστηρίζεται από ένα αρχείο που έχει δημιουργηθεί από τη συνάρτησή μας. Σε διαφορετική περίπτωση αρνούμαστε να μεταφέρουμε τις αντίστοιχες σελίδες.

Για τους ίδιους λόγους που αναφέρθηκαν στο κεφάλαιο 4.3, η ελάχιστη μονάδα μνήμης που μπορούμε να μεταφέρουμε είναι μία σελίδα. Συνεπώς, το μέγεθος του buffer - αρχείου πρέπει να είναι πολλαπλάσιο του μεγέθους της σελίδας.

6.3 Προορισμός των δεδομένων - Μνήμη παραλήπτη

Όπως εξηγήσαμε στη προηγούμενη ενότητα, τα δεδομένα που αποτελούν ένα μήνυμα υποστηρίζονται από ένα μη ορατό αρχείο στο *tmpfs*. Προκειμένου να αποκτήσει πρό-

σβαση στις αντίστοιχες σελίδες ο παραλήπτης, χωρίς να ενημερώσει τους πίνακες σελίδων του, πρέπει να απεικονίσει στο χώρο διευθύνσεων του τα αντίστοιχα κομμάτια του αρχείου αυτού. Για το σκοπό αυτό παρέχεται μια ειδική συνάρτηση, που υλοποιείται με χρήση της ioctl, η οποία αναλαμβάνει το έργο αυτό. Δηλαδή, δεσμεύει ένα εύρος εικονικών διευθύνσεων στο χώρο διευθύνσεων του παραλήπτη και απεικονίζει εκεί τα κομμάτια του αρχείου, τα οποία προσδιορίζει το εισερχόμενο μήνυμα. Η διαδικασία αυτή εκτελείται αυτόματα από τον μηχανισμό κατά τη λήψη ενός μηνύματος, επομένως δεν υπάρχει κάποια επιπλέον απαίτηση από τον παραλήπτη. Η συνάρτηση που υλοποιεί αυτή τη λειτουργικότητα αναλύεται εκτενέστερα στην ενότητα 6.6.4.

6.4 Write / Read Semantics

Προσπαθήσαμε και σε αυτόν τον μηχανισμό να διατηρήσουμε, όσο είναι εφικτό, τη σημασιολογία των κλήσεων συστήματος write και read.

write semantics Όταν μια διεργασία χρησιμοποιεί την κλήση συστήματος write αναμένει ότι θα είναι σε θέση να χρησιμοποιήσει ξανά τον buffer, μόλις η κλήση της write επιστρέψει (copy semantics), χωρίς να αλλοιώσει τα μεταδιδόμενα δεδομένα. Για τη διατήρηση της σημασιολογίας της κλήσης συστήματος write και λαμβάνοντας υπόψη τη φύση του μηχανισμού zipc-tmpfs, εξετάσαμε τις ακόλουθες προσεγγίσεις:

- *Χρήση του μηχανισμού COW για τη προστασία των μεταδιδόμενων δεδομένων.*
  Ο μηχανισμός COW δεν είναι εφαρμόσιμος στην περίπτωση του zipc-tmpfs. Όπως έχουμε ήδη αναφέρει, ένας από τους στόχους του μηχανισμού είναι να παρέχει στις διεργασίες τη δυνατότητα να προωθούν μηνύματα, χωρίς να ενημερώνουν τους πίνακες σελίδων τους. Ο μηχανισμός COW, όμως, υλοποιείται με χρήση κατάλληλων σημαιών στις καταχωρήσεις των πινάκων σελίδων (page table entry flags).

- *Αλλαγή των δικαιωμάτων πρόσβασης της εικονικής περιοχής μνήμης που απεικονίζει το αρχείο με τα δεδομένα προς αποστολή.*
  Κατά την αποστολή ενός μηνύματος αλλάζουν τα δικαιώματα πρόσβασης της διεργασίας - αποστολέα στα αντίστοιχα κομμάτια του αρχείου στο *tmpfs*. Συγκεκριμένα, αφαιρούμε από τον αποστολέα το δικαίωμα εγγραφής στις αντίστοιχες

περιοχές εικονικής μνήμης. Η προσέγγιση αυτή έχει ορισμένα σημαντικά μειονεκτήματα. Η διεργασία - αποστολέας δεν μπορεί να χρησιμοποιήσει άμεσα τον buffer αλλά πρέπει να περιμένει να τελειώσουν με αυτόν όλοι οι πιθανοί παραλήπτες. Θυμίζουμε ότι μια διεργασία - παραλήπτης μπορεί να επιλέξει να προωθήσει ένα εισερχόμενο μήνυμα σε κάποια άλλη. Αυτό παραβιάζει την απαίτηση για άμεση επαναχρησιμοποίηση του buffer. Επίσης, επαφίεται στις διεργασίες - παραλήπτες να αποδεσμεύσουν ρητά τις αντίστοιχες περιοχές του αρχείου, για να μπορέσει ο μηχανισμός μας να επαναφέρει τα δικαιώματα εγγραφής στην αρχική διεργασία - αποστολέα. Αυτή η συμπεριφορά βασίζεται στην ύπαρξη εμπιστοσύνης ανάμεσα στις διεργασίες που επικοινωνούν. Ωστόσο, θέλοντας να υλοποιήσουμε έναν γενικό μηχανισμό, που θα επιτρέπει την επικοινωνία ανάμεσα σε αυθαίρετες διεργασίες, αποφασίσαμε να μην κάνουμε μια τέτοια υπόθεση και ως εκ τούτου απορρίψαμε αυτή τη προσέγγιση.

- *Κατάργηση των αντίστοιχων απεικονίσεων από το χώρο διευθύνσεων του αποστολέα.*

  Καταργούνται τόσο οι αντίστοιχες καταχωρήσεις στους πίνακες σελίδων της διεργασίας - αποστολέα, όσο και οι αντίστοιχες περιοχές εικονικής μνήμης. Αυτό έχει ως συνέπεια επόμενη πρόσβαση σε buffers που έχουν σταλεί να προκαλεί σφάλμα κατάτμησης (segmentation fault). Αυτή η προσέγγιση διασφαλίζει μεν ότι ο αποστολέας δεν μπορεί να αλλοιώσει τα μεταδιδόμενα δεδομένα, αλλά δεν ικανοποιεί την απαίτηση για επαναχρησιμοποίηση των buffers. Ωστόσο, είναι το καλύτερο που μπορούμε να πετύχουμε, χωρίς να κάνουμε κάποια υπόθεση για τις εμπλεκόμενες διεργασίες, οπότε αποφασίσαμε να ακολουθήσουμε αυτή τη λύση.

read semantics Όταν μια διεργασία χρησιμοποιεί την κλήση συστήματος read δίνει στον πυρήνα τη διεύθυνση ενός buffer και αναμένει από αυτόν να τοποθετήσει εκεί τα δεδομένα. Στην περίπτωση του μηχανισμού μας, ωστόσο, δε δίνεται η δυνατότητα στο χρήστη να προσδιορίσει που θα τοποθετηθούν τα δεδομένα. Δεσμεύεται αυτόματα ένα εύρος εικονικών διευθύνσεων στο χώρο διευθύνσεων του παραλήπτη και απεικονίζονται εκεί τα εισερχόμενα δεδομένα (πιο σωστά απεικονίζονται εκεί τα κομμάτια του αρχείου που περιέχουν τα εισερχόμενα δεδομένα). Με εξαίρεση αυτή τη διαφορά, η συμπεριφορά του μηχανισμού μας, κατά τη λήψη δεδομένων, διατηρεί τη σημασιολογία της κλήσης συστήματος read.

6.5 Βασικές δομές δεδομένων του zipc-tmpfs

Η βασική δομή του zipc-tmpfs είναι μία σωλήνωση (pipe), αντίστοιχη αυτής που είδαμε στο σχήμα 2.12 του κεφαλαίου 2.3.1. Έτσι, ο zipc-tmpfs παρέχει ένα μονόδρομο κανάλι επικοινωνίας (half duplex) που παραδίδει μηνύματα από τον αποστολέα στον παραλήπτη σε διάταξη FIFO.

Το pipe έχει ένα άκρο εγγραφής και ένα άκρο ανάγνωσης. Τα μηνύματα που τοποθετούνται στο άκρο εγγραφής μπορούν να διαβαστούν από το άκρο ανάγνωσης. Το pipe του zipc-tmpfs, σε αντίθεση με αυτό του zipc, δε μεταφέρει δείκτες σε πλαίσια μνήμης αλλά μηνύματα. Ένα μήνυμα είναι ένας πίνακας από εγγραφές της μορφής *(struct file *file, unsigned long offset, unsigned long len)*. Κάθε τέτοια εγγραφή περιγράφει ένα κομμάτι ενός αρχείου στο *tmpfs*, το οποίο περιέχει δεδομένα προς αποστολή. Ως εκ τούτου, ένα μήνυμα μπορεί να μεταφέρει πολλές σελίδες μνήμης ατομικά. Σε αντιδιαστολή το pipe του zipc μπορεί να μεταφέρει, συνολικά, δείκτες για το πολύ 16 πλαίσια μνήμης, πριν γεμίσει (το προκαθορισμένο μέγεθος μπορεί φυσικά να αλλάξει, όπως έχουμε αναφέρει).

Η χωρητικότητα του pipe είναι περιορισμένη και μπορεί να μεταβληθεί κατά το χρόνο φόρτωσης του module. Η προεπιλεγμένη τιμή της είναι δεκαέξι (16) μηνύματα. Το μέγεθος του pipe ορίζει, ουσιαστικά, πόσα μηνύματα μπορούν να είναι εκκρεμή ανά πάσα στιγμή, δηλαδή να μην έχουν διαβαστεί από κάποια διεργασία.

6.6 zipc-tmpfs API

Οι διεργασίες που επιθυμούν να επικοινωνήσουν με χρήση του μηχανισμού zipc-tmpfs πρέπει να ανοίξουν το ίδιο αρχείο συσκευής (device file). Αυτό γίνεται με χρήση της κλήσης συστήματος open. Όταν τελειώσουν με τη μεταξύ τους επικοινωνία, κλείνουν το αρχείο με χρήση της κλήσης συστήματος close. Για την επικοινωνία των διεργασιών υλοποιήσαμε την κλήση συστήματος writev και τη συνάρτηση receive_msg. Η δέσμευση (allocation) ενός νέου buffer, που θα φιλοξενήσει τα δεδομένα προς αποστολή, γίνεται με τη συνάρτηση allocate_buffer. Στη συνέχεια αναλύουμε τη λειτουργικότητα όλων των μεθόδων που υλοποιήσαμε.

6.6.1 Άνοιγμα του zipc-tmpfs

Ισχύουν ακριβώς τα ίδια με το zipc. Βλέπε κεφάλαιο 4.7.1.

6.6.2 Δέσμευση του buffer αποστολής

Πραγματοποιείται με τη συνάρτηση allocate_buffer:

```
void * allocate_buffer (int fd, size_t size);
```

Το όρισμα *fd* πρέπει να είναι ο file descriptor που επιστράφηκε προηγουμένως από την open. Το όρισμα *size* είναι το επιθυμητό μέγεθος του buffer, που πρέπει να είναι πολλαπλάσιο του μεγέθους μιας σελίδας. Σε περίπτωση επιτυχίας η allocate_buffer επιστρέφει τη διεύθυνση του buffer, στον εικονικό χώρο διευθύνσεων της διεργασίας που την κάλεσε. Σε περίπτωση αποτυχίας επιστρέφεται NULL. Η allocate_buffer είναι συνάρτηση - περιτύλιγμα για την εντολή ioctl ZIPC_IOC_GETBUFFER. Παρέχεται για διευκόλυνση του προγραμματισμού στα προγράμματα χρήστη.

Όπως έχει σκιαγραφηθεί και στις προηγούμενες ενότητες, η allocate_buffer δημιουργεί ένα μη ορατό αρχείο (unlinked file) στο ειδικό σύστημα αρχείων *tmpfs*. Ως εκ τούτου, καμία άλλη διεργασία δεν μπορεί να ανοίξει και να απεικονίσει το αρχείο αυτό. Το μέγεθος του αρχείου είναι αυτό που προσδιορίζεται από το όρισμα *size*. Στη συνέχεια η allocate_buffer απεικονίζει το αρχείο στο χώρο διευθύνσεων της διεργασίας, καλώντας εσωτερικά στον πυρήνα την mmap, και επιστρέφει τη διεύθυνση αυτής της περιοχής εικονικής μνήμης ως αποτέλεσμα. Επιπρόσθετα, τίθεται κατάλληλη σημαία για την περιοχή μνήμης, ούτως ώστε να εξασφαλίσουμε ότι δε θα κληρονομηθεί μέσω της κλήσης συστήματος fork. Έτσι, εξασφαλίζουμε ότι τα υποκείμενα πλαίσια μνήμης είναι ιδιωτικά και μπορούν να μεταφερθούν μέσω του zipc-tmpfs.

6.6.3 Αποστολή δεδομένων

Πραγματοποιείται με την κλήση συστήματος writev:

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Το όρισμα *fd* πρέπει να είναι ο file descriptor που επιστράφηκε προηγουμένως από την open. Ο δείκτης *iov* δείχνει σε έναν πίνακα από *iovcnt* το πλήθος δομές τύπου *struct iovec*, οι οποίες ορίζονται ως εξής:

```
struct iovec {
  void *iov_base;     /* Starting address */
   size_t iov_len;    /* Number of bytes to transfer */
};
```

Η writev οργανώνει τις σελίδες μνήμης που περιγράφονται από τους buffers *iov* σε ένα μήνυμα. Εντοπίζει τα κομμάτια των αρχείων στο *tmpfs* στα οποία ανήκουν αυτές οι σελίδες και κατασκευάζει έναν πίνακα από εγγραφές της μορφής *(struct file *file, unsigned long offset, unsigned long len)*. Αυτός ο πίνακας αποτελεί ένα μήνυμα που στη συνέχεια τοποθετείται στο pipe. Ταυτόχρονα, οι αντίστοιχες περιοχές μνήμης του αποστολέα, που απεικόνιζαν αυτά τα κομμάτια του αρχείου, καταστρέφονται. Έτσι, επόμενη πρόσβαση σε αυτές θα προκαλέσει σφάλμα κατάτμησης (segmentation fault). Θυμίζουμε ότι οι περιοχές μνήμης που περιγράφονται από κάθε δομή τύπου *struct iovec* πρέπει να ξεκινάνε σε διεύθυνση πολλαπλάσιο του μεγέθους της σελίδας και να έχουν μέγεθος επίσης πολλαπλάσιο μιας σελίδας. Επιπρόσθετα, πρέπει να χρησιμοποιηθεί μόνο μνήμη που έχει δεσμευτεί μέσω της allocate_buffer, που εξετάσαμε προηγουμένως. Τέλος, οι δομές *iov* δεν πρέπει να έχουν επικαλύψεις. Είναι λάθος μια διεργασία να δωρίσει δύο ή περισσότερες φορές την ίδια σελίδα μνήμης. Αν η writev συναντήσει buffer που δεν πληροί τις παραπάνω προϋποθέσεις σταματάει άμεσα. Αν υπήρχαν έγκυρα δεδομένα μέχρι εκείνο το σημείο, τότε στέλνονται αυτά, δηλαδή μπορεί να συμβεί μια μερική αποστολή. Διαφορετικά η writev επιστρέφει -1 και κωδικό σφάλματος EINVAL.

Για την κατασκευή του μηνύματος οι buffers χρησιμοποιούνται με τη σειρά που εμφανίζονται στον πίνακα. Δηλαδή, η writev τοποθετεί στο μήνυμα πρώτο το κομμάτι αρχείου που περιγράφει ο buffer *iov[0]*, μετά αυτό που περιγράφει ο *iov[1]*, κοκ.

Αν το pipe είναι γεμάτο, τότε η writev θα μπλοκάρει μέχρι να ελευθερωθεί επαρκής χώρος για να μπορέσει να πραγματοποιηθεί η αποστολή του μηνύματος. Αν το αρχείο

του zipc έχει ανοιχτεί με το *O_NONBLOCK* flag (non-blocking mode) τότε η writev επιστρέφει άμεσα -1 και κωδικό σφάλματος EAGAIN.

Σε περίπτωση επιτυχίας η writev επιστρέφει τον αριθμό των bytes που μεταφέρει το μήνυμα, ο οποίος μπορεί να είναι μικρότερος απ' ότι ζητήθηκε από το χρήστη. Σε περίπτωση σφάλματος επιστρέφεται -1 και η μεταβλητή *errno* περιέχει τον κωδικό σφάλματος. Αν όλοι οι file descriptors που αναφέρονται στο άκρο ανάγνωσης του pipe κλείσουν, τότε μια απόπειρα αποστολής στο pipe θα έχει ως αποτέλεσμα να σταλεί στη διεργασία το σήμα (signal) SIGPIPE. Αν η διεργασία αγνοεί αυτό το σήμα τότε η κλήση συστήματος writev θα αποτύχει με κωδικό σφάλματος EPIPE.

Σημείωση Για την αποστολή δεδομένων μπορεί να χρησιμοποιηθεί και η κλήση συστήματος write, η οποία μεταφράζεται αυτόματα από τον πυρήνα σε κλήση της μεθόδου writev, που υλοποιεί ο zipc-tmpfs. Δηλαδή μια κλήση της μορφής *write(fd, buf, count)* είναι ισοδύναμη με τη *writev(fd, iov, 1)*, όπου *iov->iov_base = buf* και *iov->iov_len = count*.

6.6.4 Λήψη δεδομένων

Πραγματοποιείται με τη συνάρτηση receive_msg:

```
int receive_msg(int fd, struct iovec *iov)
```

Το όρισμα *fd* πρέπει να είναι ο file descriptor που επιστράφηκε προηγουμένως από την open. Ο δείκτης *iov* δείχνει σε μια δομή τύπου *struct iovec*, η οποία ορίζεται ως εξής:

```
struct iovec {
  void *iov_base;      /* Starting address */
  size_t iov_len;      /* Number of bytes to transfer */
};
```

Η συνάρτηση receive_msg είναι συνάρτηση - περιτύλιγμα για την εντολή ioctl ZIPC_IOC_RCVMSG. Παρέχεται για διευκόλυνση του προγραμματισμού στα προγράμματα χρήστη.

Η receive_msg διαβάζει το επόμενο διαθέσιμο μήνυμα από το pipe. Στη συνέχεια καλεί εσωτερικά την mmap, όσες φορές χρειαστεί, για να απεικονίσει σε γραμμικές διευθύνσεις του χώρου διευθύνσεων του παραλήπτη το μήνυμα. Ένα μήνυμα μεταφέρει μια σειρά από εγγραφές της μορφής *(struct file *file, unsigned long offset, unsigned long len)*. Κάθε εγγραφή, όπως έχουμε αναφέρει, περιγράφει ένα κομμάτι κάποιου αρχείου στο *tmpfs*, που έχει δημιουργηθεί με την allocate_buffer. Τα διαδοχικά κομμάτια μπορεί να μην αναφέρονται σε διαδοχικές θέσεις μέσα στο αρχείο. Επιπλέον, μπορεί να αναφέρονται σε διαφορετικά αρχεία. Από την προηγούμενη περιγραφή γίνεται φανερό ότι υπάρχει η ανάγκη για μια μη γραμμική περιοχή εικονικής μνήμης, δηλαδή μια περιοχή μνήμης, η οποία απεικονίζει τις σελίδες του αρχείου σε μη γραμμική σειρά στη μνήμη. Δυστυχώς ο πυρήνας του Linux δεν παρέχει ουσιαστική υποστήριξη για μη γραμμικές περιοχές εικονικής μνήμης. Έτσι, το καλύτερο που μπορούμε να κάνουμε είναι να δημιουργήσουμε πολλαπλές διαδοχικές περιοχές εικονικής μνήμης, που απεικονίζουν διαφορετικά κομμάτια του αρχείου ή/και διαφορετικά αρχεία. Σημειώνουμε ότι απλώς κατασκευάζονται οι αντίστοιχες περιοχές εικονικής μνήμης χωρίς να ενημερωθούν οι πίνακες σελίδων του παραλήπτη.

Αν το pipe είναι άδειο, τότε η receive_msg μπλοκάρει μέχρι να υπάρξει κάποιο διαθέσιμο μήνυμα. Αν το αρχείο του zipc-tmpfs έχει ανοιχτεί με το *O_NONBLOCK* flag (non-blocking mode), τότε η receive_msg επιστρέφει άμεσα -1 και κωδικό σφάλματος EAGAIN.

Εάν στο χώρο διευθύνσεων του παραλήπτη δεν υπάρχει ελεύθερο αρκετά μεγάλο εύρος διευθύνσεων για να απεικονιστεί το μήνυμα, η receive_msg επιστρέφει -1 και κωδικό σφάλματος ENOMEM. Στην περίπτωση αυτή το μήνυμα παραμένει στο pipe.

Σε περίπτωση επιτυχίας η receive_msg τοποθετεί στο πεδίο *iov->iov_base* τη διεύθυνση, στη μνήμη του παραλήπτη, στην οποία απεικονίστηκε το μήνυμα και στο πεδίο *iov->iov_len* το μέγεθος του. Σε περίπτωση σφάλματος επιστρέφεται -1 και η μεταβλητή *errno* περιέχει τον κωδικό σφάλματος. Αν όλοι οι file descriptors που αναφέρονται στο άκρο εγγραφής του pipe έχουν κλείσει τότε το πεδίο *iov->iov_len* είναι ίσο με μηδέν (end-of-file).

6.6.5 Κλείσιμο του zipc-tmpfs

Ισχύουν ακριβώς τα ίδια με το zipc. Βλέπε κεφάλαιο 4.7.4.

6.7 Προώθηση μηνύματος

Όπως έχει αναφερθεί σε διάφορα σημεία του κεφαλαίου, ο zipc-tmpfs παρέχει τη δυνατότητα σε έναν παραλήπτη να προωθήσει ένα εισερχόμενο μήνυμα. Αυτό γίνεται με χρήση της κλήσης συστήματος writev, δηλαδή ακριβώς όπως και η αποστολή μηνυμάτων. Οι περιοχές μνήμης που δημιουργεί η receive_msg είναι κατάλληλες για χρήση με τη writev. Μάλιστα ενεργοποιούνται για τις περιοχές αυτές κατάλληλες σημαίες για να μην μπορούν να κληρονομηθούν μέσω του fork και για να μη μπορεί να επεκταθεί το μέγεθός τους μέσω της mremap. Αυτό εξασφαλίζει ότι τα υποκείμενα πλαίσια μνήμης παραμένουν ιδιωτικά και ότι ο παραλήπτης δεν μπορεί να δει κομμάτια ενός αρχείου που δεν του στάλθηκαν μέσω ενός μηνύματος. Αν ο παραλήπτης δεν έχει διαβάσει ή/και γράψει στα κομμάτια του μηνύματος που θα προωθήσει, τότε δεν ενημερώνονται ποτέ οι αντίστοιχες καταχωρήσεις στους πίνακες σελίδων του και ως εκ τούτου δε χρειάζεται να καταστραφούν κατά την προώθηση.

Οποιαδήποτε κομμάτια ενός εισερχόμενου μηνύματος μπορούν να προωθηθούν με χρήση της writev. Επιπλέον, ο παραλήπτης θα μπορούσε να συνδυάσει στο ίδιο εξερχόμενο μήνυμα κομμάτια από διάφορα εισερχόμενα μηνύματα ή/και buffers που έχει κάνει ο ίδιος allocate με την allocate_buffer. Παρέχεται, κατά αυτόν τον τρόπο, μια δυνατότητα gather output σε μια διεργασία, εφόσον τα δεδομένα βρίσκονται σε περιοχές μνήμης που χειρίζεται ο zipc-tmpfs.

6.8 Διευθυνσιοδότηση και Ασφάλεια

Ισχύουν ακριβώς τα ίδια με το zipc. Βλέπε κεφάλαιο 4.8.

6.9 Παράδειγμα χρήσης του zipc-tmpfs

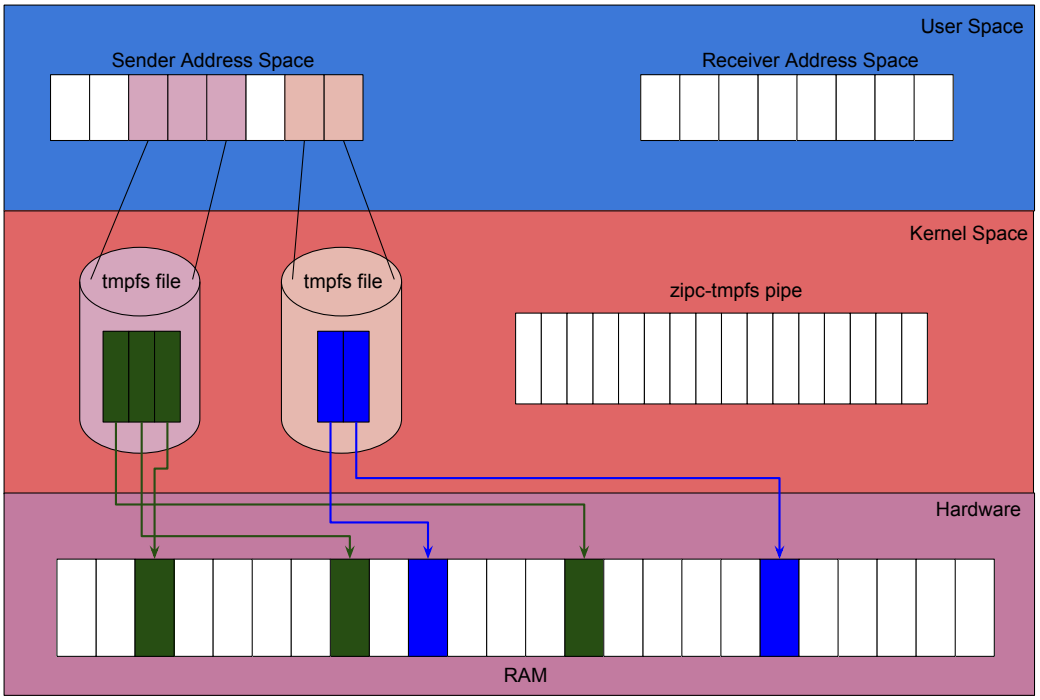
Στα ακόλουθα σχήματα φαίνεται ένα παράδειγμα χρήσης του μηχανισμού zipc-tmpfs. Στο σχήμα 6.1 φαίνεται ο χώρος διευθύνσεων του αποστολέα, το pipe και ο χώρος διευθύνσεων του παραλήπτη, πριν την αποστολή των δεδομένων. Ο αποστολέας έχει ετοιμάσει δύο περιοχές μνήμης προς αποστολή, που περιγράφονται από δομές τύπου *struct iovec*. Η δέσμευση των περιοχών αυτών έγινε με χρήση της συνάρτησης allocate_buffer και ως εκ τούτου απεικονίζουν αρχεία στο σύστημα αρχείων tmpfs. Στο σχήμα 6.2 βλέπουμε την εικόνα μετά την εκτέλεση της writev. Οι σελίδες μνήμης, με τα δεδομένα προς αποστολή, έχουν οργανωθεί σε ένα μήνυμα, το οποίο έχει τοποθετηθεί στο pipe, ενώ έχουν καταργηθεί οι απεικονίσεις τους από το χώρο διευθύνσεων του αποστολέα. Στο σχήμα 6.3 βλέπουμε την εικόνα μετά την εκτέλεση της receive_msg στον παραλήπτη. Το μήνυμα έχει αφαιρεθεί από το pipe και το περιεχόμενο του έχει απεικονιστεί στο χώρο διευθύνσεων του παραλήπτη. Όπως φαίνεται και στα σχήματα, δε δημιουργείται κανένα αντίγραφο των μεταδιδόμενων δεδομένων.

Τέλος, για καλύτερη κατανόηση των παραδειγμάτων, δίνεται ενδεικτικός ψευδοκώδικας.

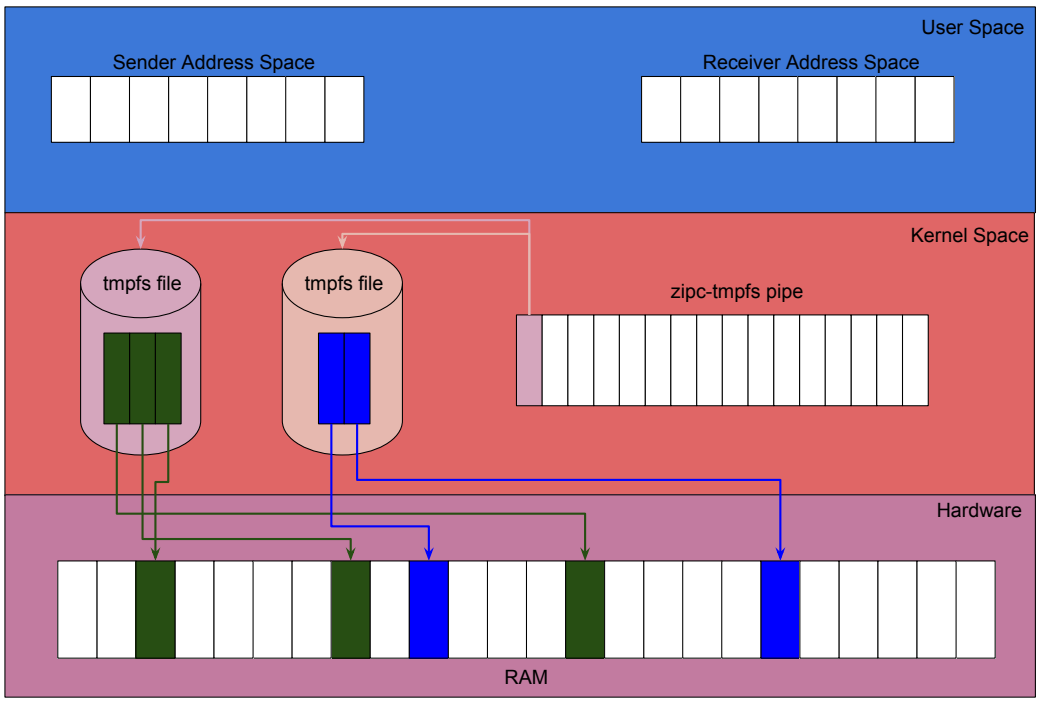
```
// Sender

#define PAGE_SIZE 4096
int fd, ret;
struct iovec iov[2];

fd = open("zipc-tmpfs-device-file-path", O_WRONLY);

iov[0].iov_base = allocate_buffer(fd, PAGE_SIZE * 3);
iov[0].iov_len = PAGE_SIZE * 3;

iov[1].iov_base = allocate_buffer(fd, PAGE_SIZE * 2);
iov[1].iov_len = PAGE_SIZE * 2;

// fill iovecs with data

ret = writev(fd, &iov[0], 2);

// Receiver
```

```
#define PAGE_SIZE 4096
int fd, ret;
struct iovec iov;

fd = open("zipc-tmpfs-device-file-path", O_RDONLY);

ret = receive_msg(fd, &iov);
```

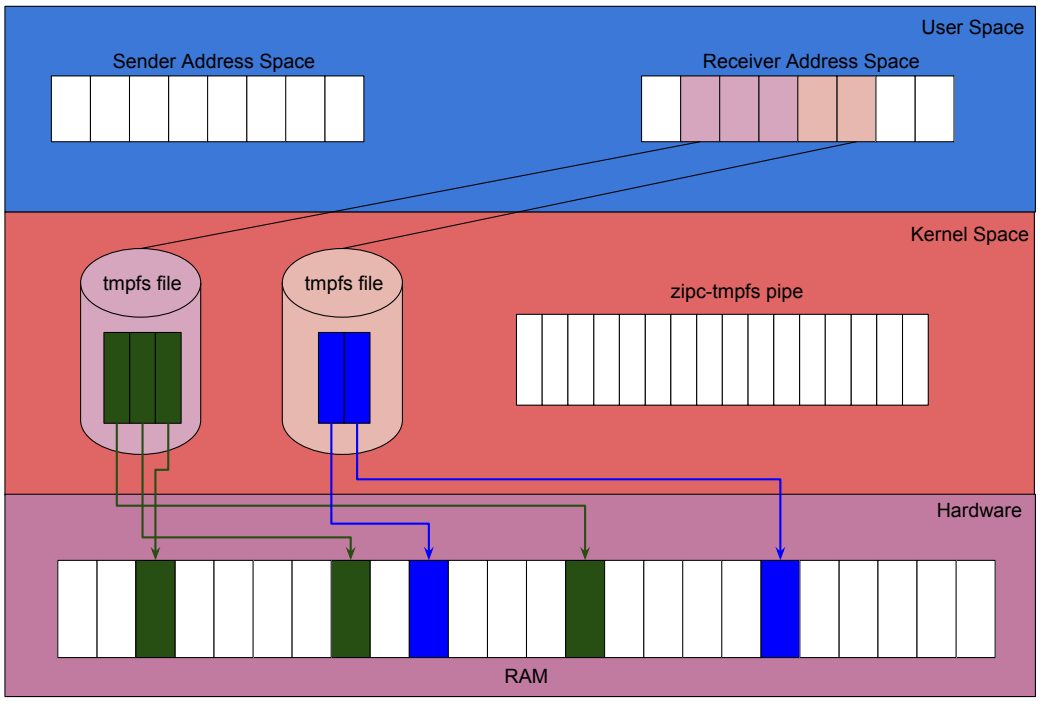
Σχήμα 6.1: *Εικόνα μετά την εκτέλεση της allocate_buffer και πριν την εκτέλεση της writev*



Σχήμα 6.2: *Εικόνα μετά την εκτέλεση της writev και πριν την εκτέλεση της receive_msg*



Σχήμα 6.3: *Εικόνα μετά την εκτέλεση της receive_msg*



7

Υλοποίηση του zipc-tmpfs

Στο κεφάλαιο αυτό περιγράφεται η υλοποίηση του μηχανισμού διαδιεργασιακής επικοινωνίας zipc-tmpfs, η σχεδίαση του οποίου εξετάστηκε λεπτομερώς στο προηγούμενο κεφάλαιο. Η υλοποίηση έχει ως στόχο την πειραματική αξιολόγηση της σχεδίασης, μέσα από τη διεξαγωγή μετρήσεων. Επίσης, μας φέρνει αντιμέτωπους με προγραμματιστικά προβλήματα, που σχετίζονται με τα υποσυστήματα του πυρήνα, αλλά και τους περιορισμούς του υλικού.

Η υλοποίηση έλαβε χώρα σε ένα ευρέως χρησιμοποιούμενο σύστημα αρχιτεκτονικής *x86-64*. Το λειτουργικό σύστημα που χρησιμοποιήθηκε είναι το Debian GNU/Linux με έκδοση πυρήνα τη *3.16.0-4-amd64*. Όλες οι συναρτήσεις, που θα δούμε στη συνέχεια της ενότητας αυτής, βασίζονται στις διεπαφές που ορίζει αυτή η έκδοση πυρήνα.

Για τη διευκόλυνση της αποσφαλμάτωσης του κώδικα και την επιτάχυνση του προγραμματισμού, κατά τα πρώτα στάδια ανάπτυξης του μηχανισμού, χρησιμοποιήσαμε μια εικονική μηχανή, ακριβώς όπως και στην περίπτωση του zipc.

7.1 Χρήση μη εξαχθέντων συναρτήσεων του πυρήνα

Ισχύουν ακριβώς τα ίδια με το zipc. Βλέπε κεφάλαιο 5.1.

7.2 Υλοποίηση βασικών δομών δεδομένων του zipc-tmpfs

Το pipe υλοποιείται ως ένας κυκλικός buffer με προκαθορισμένο μέγεθος δεκαέξι καταχωρήσεις, κάθε μία από τις οποίες περιέχει ένα δείκτη προς ένα μήνυμα. Ακολουθεί ο σχετικός κώδικας:

```
#define PIPE_DEF_BUFFERS 16

/*
  struct pipe_buffer - a pipe buffer
  @msg: the message this buffer is carrying
  @ops: operations associated with this buffer - NULL if buffer is empty
*/

struct pipe_buffer {
  struct msg msg;
  const struct pipe_buf_operations *ops;
};

/*
  struct pipe_info - a pipe
  @mutex: mutex protecting the pipe
  @wait: reader/writer wait point in case of empty/full pipe
  @nrbufs: the number of non-empty pipe buffers in this pipe
  @buffers: total number of buffers (power of 2)
  @curbuf: the current pipe buffer entry
  @readers: number of current readers of this pipe
  @writers: number of current writers of this pipe
  @r_counter: used for syncing with writers in zipc_open()
  @w_counter: used for syncing with readers in zipc_open()
  @files: number of struct file referring this pipe (protected by inode->i_lock)
  @waiting_writers: number of writers blocked waiting for room
  @bufs: the circular array of pipe buffers
*/

struct pipe_info {
  struct mutex mutex;
  wait_queue_head_t wait;
  unsigned int nrbufs, curbuf, buffers;
  unsigned int readers;
```

```
  unsigned int  writers ;
  unsigned int  r_counter;
  unsigned int  w_counter;
  unsigned int  files ;
  unsigned int  waiting_writers ;
  struct  pipe_buffer  *bufs;
};

struct  pipe_buf_operations  {
  /*
    When the contents  of  this  pipe  buffer  has  been  consumed by a
    reader ,  ->release ()  is  called .
  */
  void  (* release )( struct  pipe_info  *,  struct  pipe_buffer  *);
};
```

Ένα μήνυμα είναι ένας πίνακας από εγγραφές της μορφής *(struct file *file, unsigned long offset, unsigned long len)*. Κάθε τέτοια εγγραφή περιγράφει ένα κομμάτι ενός αρχείου στο *tmpfs*, το οποίο κομμάτι περιέχει δεδομένα προς αποστολή. Ως εκ τούτου, ένα μήνυμα μπορεί να μεταφέρει πολλές σελίδες μνήμης. Τα διαδοχικά κομμάτια μπορεί να μην αναφέρονται σε διαδοχικές θέσεις μέσα στο αρχείο. Επιπλέον, μπορεί να αναφέρονται σε διαφορετικά αρχεία. Ακολουθεί ο σχετικός κώδικας:

```
/*
  struct  msg_part - a message  part
  @file :  pointer  to  file  descriptor  this  message  part  refers  to
  @offset :  offset  in  @file
  @len:  length  (page  aligned )  of  the  data  in  @file  we  refer  to
*/

struct  msg_part {
  struct  file  * file ;
  unsigned long  offset ;
  unsigned long  len;
};

/*
  struct  msg - a message
```

```
  @msg_parts: array storing the parts of the message
  @nr_parts: number of elements in @msg_parts array
  @count: total size of message in bytes

  A message can have multiple parts. Each part refers to a possibly
   different open file.
*/

struct msg {
  struct msg_part *msg_parts;
  size_t nr_parts;
  size_t count;
};
```

7.3 Υλοποίηση κλήσης συστήματος ioctl

Χρησιμοποιούμε την κλήση συστήματος ioctl για την υλοποίηση δύο εντολών: δέσμευση ενός νέου buffer αποστολής και λήψη ενός μηνύματος. Ο κώδικας που υλοποιεί την ioctl και καλεί τις αντίστοιχες συναρτήσεις δίνεται ακολούθως:

```
/* ioctl definitions */

#define ZIPC_IOC_MAGIC 0xFC

#define ZIPC_IOC_RCVMSG _IOR(ZIPC_IOC_MAGIC, 1, struct iovec)
#define ZIPC_IOC_GETBUFFER _IOWR(ZIPC_IOC_MAGIC, 2, struct iovec)

#define ZIPC_IOC_MAXNR 2

long zipc_ioctl (struct file * filp, unsigned int cmd, unsigned long arg)
{
  int err;
  struct iovec iov = {NULL, 0};
  unsigned long addr;

  err = 0;
```

```
if (_IOC_TYPE(cmd) != ZIPC_IOC_MAGIC)
  return -ENOTTY;

if (_IOC_NR(cmd) > ZIPC_IOC_MAXNR)
  return -ENOTTY;

if (_IOC_DIR(cmd) & _IOC_READ)
  err = !access_ok(VERIFY_WRITE, (void __user *) arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
  err = !access_ok(VERIFY_READ, (void __user *) arg, _IOC_SIZE(cmd));

if (err)
  return -EFAULT;

switch (cmd) {

  case ZIPC_IOC_RCVMSG:
    err = receive_msg(filp, (unsigned long *) &iov.iov_base, &iov.iov_len);
    if (err)
      return err;
    if (__copy_to_user((void __user *) arg, &iov, sizeof iov))
      return -EFAULT;
    break;

  case ZIPC_IOC_GETBUFFER:
    if (__copy_from_user(&iov, (void __user *) arg, sizeof iov))
      return -EFAULT;

    addr = get_buffer(iov.iov_len);

    if (IS_ERR_VALUE(addr))
      return addr;

    iov.iov_base = (void *) addr;

    if (__copy_to_user((void __user *) arg, &iov, sizeof iov))
      return -EFAULT;

    break;
```

```
    default:
      return -ENOTTY;
  }

  return 0;
}
```

7.4 Δέσμευση buffer αποστολής

Υλοποιείται με τη συνάρτηση get_buffer που καλείται από την κλήση συστήματος ioctl. Η λειτουργία αυτής της συνάρτησης περιγράφηκε αναλυτικά στο κεφάλαιο του σχεδιασμού. Εδώ δίνουμε τον αντίστοιχο κώδικα:

```
/*
  get_buffer () - Get a memory buffer of size @size
  @size: the requested size of the buffer (must be PAGE ALIGNED)

  Returns the user virtual address of the buffer or error code in case something went
      wrong (check with IS_ERR_VALUE())

  The buffer is backed by an internal tmpfs file mapped to the address space of the process
*/

unsigned long get_buffer(unsigned long size)
{
  struct file * file;
  unsigned long addr;
  struct mm_struct *mm = current->mm;
  unsigned long populate;

  unsigned long (*do_mmap_pgoff)(struct file *, unsigned long, unsigned long, unsigned
      long, unsigned long, unsigned long *);

  do_mmap_pgoff = (void *) do_mmap_pgoff_addr;


  if (size & ~PAGE_MASK)
```

```
    return -EINVAL;

  file = shmem_file_setup("zipc-buf", size, 0);

  if (unlikely(IS_ERR(file)))
    return PTR_ERR(file);

  down_write(&mm->mmap_sem);

  addr = do_mmap_pgoff(file, 0, size, PROT_READ | PROT_WRITE, MAP_SHARED, 0,
      &populate);

  if (likely(!IS_ERR_VALUE(addr))) {
    struct vm_area_struct *vma;

    vma = find_vma(mm, addr);

    BUG_ON(!vma || !(vma->vm_start <= addr && vma->vm_end >= addr + size));

    vma->vm_flags |= VM_DONTCOPY;
  }

  up_write(&mm->mmap_sem);

  fput(file);

  return addr;
}
```

Ο κώδικας της συνάρτησης - περιτύλιγμα allocate_buffer έχει ως εξής:

```
/*
 * allocate_buffer() - allocate a buffer of size @size
 * @fd: file descriptor of zipc-tmpfs device file
 * @size: buffer size, multiple of PAGE_SIZE
 *
 * Allocate a buffer for use with the zipc-tmpfs module
 *
 * Returns the user space address of the buffer or NULL if allocation failed (check errno)
```

```
*/

void * allocate_buffer (int fd, size_t size)
{
  struct iovec iov = {NULL, 0};

  iov.iov_len = size;

  if (ioctl(fd, ZIPC_IOC_GETBUFFER, &iov))
    return NULL;

  return iov.iov_base;
}
```

7.5 Υλοποίηση λειτουργικότητας Αποστολέα

Η λειτουργικότητα του αποστολέα υλοποιείται μέσω δύο συναρτήσεων:

1. *make_msg*: Δημιουργεί ένα μήνυμα από τις σελίδες που δίνονται από το χρήστη μέσω της κλήσης συστήματος writev ή write.
2. *zipc_write_iter*: Υλοποιεί την κλήση συστήματος writev και αναλαμβάνει τον συγχρονισμό της πρόσβασης στο pipe και την επισύναψη του μηνύματος σε αυτό.

Η ακριβής συμπεριφορά του αποστολέα έχει αναλυθεί εκτενώς στο κεφάλαιο του σχεδιασμού. Για οικονομία χώρου δεν παρατίθεται ο κώδικας των παραπάνω συναρτήσεων.

7.6 Υλοποίηση λειτουργικότητας Παραλήπτη

Η λειτουργικότητα του παραλήπτη υλοποιείται μέσω δύο συναρτήσεων:

1. *map_msg*: Απεικονίζει ένα μήνυμα στο χώρο διευθύνσεων του παραλήπτη.

2. *receive_msg*: Καλείται από την κλήση συστήματος ioctl για τη λήψη ενός νέου μηνύματος. Αναλαμβάνει τον συγχρονισμό της πρόσβασης στο pipe και το διάβασμα του επόμενου διαθέσιμου μηνύματος από αυτό.

Η ακριβής συμπεριφορά του παραλήπτη έχει αναλυθεί εκτενώς στο κεφάλαιο του σχεδιασμού. Για οικονομία χώρου δεν παρατίθεται ο κώδικας των παραπάνω συναρτήσεων.

Ο κώδικας της συνάρτησης - περιτύλιγμα receive_msg έχει ως εξής:

```
/*
  receive_msg () — receive a message using zipc—tmpfs module
  @fd: file descriptor of zipc—tmpfs device file
  @iov: on success the iovec is filled with the address and size of the received message.
        iov—>iov_len == 0 means EOF.

  Returns 0 on success and —1 on failure (check errno)
*/

int receive_msg(int fd, struct iovec *iov)
{
  return ioctl (fd, ZIPC_IOC_RCVMSG, iov);
}
```

8 Πειραματική Αξιολόγηση

Στο κεφάλαιο αυτό γίνεται μία πειραματική αποτίμηση των μηχανισμών διαδιεργασιακής επικοινωνίας zipc και zipc-tmpfs, που υλοποιήσαμε στα πλαίσια της παρούσας διπλωματικής εργασίας. Στόχος είναι η αξιολόγηση των μηχανισμών σε ένα πραγματικό σύστημα και η εξαγωγή χρήσιμων συμπερασμάτων. Για το σκοπό αυτό συγκρίνουμε τις επιδόσεις που επιτυγχάνουν οι μηχανισμοί αυτοί με τις επιδόσεις υπαρχόντων μηχανισμών IPC του Linux.

8.1 Πλατφόρμα αξιολόγησης

Η πειραματική αξιολόγηση έλαβε χώρα σε ένα ευρέως χρησιμοποιούμενο σύστημα αρχιτεκτονικής *x86-64*. Το λειτουργικό σύστημα που χρησιμοποιήθηκε είναι το Debian GNU/Linux με έκδοση πυρήνα τη *3.16.0-4-amd64*. Τα χαρακτηριστικά του συστήματος αυτού παρουσιάζονται αναλυτικά στους πίνακες 8.1 και 8.2.

Για τη διεξαγωγή των μετρήσεων χρησιμοποιήθηκε το ipc-bench-mt, το οποίο παρουσιάσθηκε αναλυτικά στο κεφάλαιο 3.

Επεξεργαστής	Intel Core i7-980X [Int16]
Συχνότητα	3333 MHz
Ταχύτητα Διαύλου	3200 MHz
Αρχιτεκτονική	x86-64
Αριθμός πυρήνων	6
Αριθμός νημάτων (Hyper-Threading)	12
Level 1 Instruction Cache	6 x 32KB 4-way set associative
Level 1 Data Cache	6 x 32KB 8-way set associative
Level 2 Unified Cache	6 x 256KB 8-way set associative
Level 3 Unified Cache	12 MB shared cache 16-way set associative
Μέγιστο Εύρος Ζώνης Μνήμης	25.6 GB/s
Διαθέσιμη RAM	12 GB DDR3 1066

Πίνακας 8.1: *Hardware specifications*

Λειτουργικό Σύστημα	Debian GNU/Linux 8
Πυρήνας	3.16.0-4-amd64
Μεταγλωττιστής	gcc version 4.9.2 (Debian 4.9.2-10)

Πίνακας 8.2: *Software specifications*

8.2 Micro-benchmarks

Εκτελέσαμε μετρήσεις του ρυθμού μεταφοράς (throughput) και της καθυστέρησης μεταφοράς (latency) για όλους τους μηχανισμούς διαδιεργασιακής επικοινωνίας που περιγράψαμε στο κεφάλαιο 2.3. Η μέτρηση του ρυθμού μεταφοράς μας επιτρέπει να αξιολογήσουμε τη διεκπεραιωτική ικανότητα του εκάστοτε μηχανισμού, δηλαδή πόσο γρήγορα μπορεί να μεταφέρει ένα σύνολο από δεδομένα από τη διεργασία - αποστολέα στη διεργασία - παραλήπτη. Η μέτρηση της καθυστέρησης μεταφοράς (ping-pong latency) μας επιτρέπει να αξιολογήσουμε πόσο χρόνο απαιτεί κάθε μηχανισμός για τη μεταφορά μιας μικρής ποσότητας δεδομένων από τη διεργασία - αποστολέα στη διεργασία - παραλήπτη και αντίστροφα και είναι μια ένδειξη του πόσο γρήγορα απαντάει ο παραλήπτης σε ένα αίτημα.

Όλα τα benchmarks δημιουργήθηκαν με τη βοήθεια του ipc-bench-mt προκειμένου να έχουμε ένα κοινό πλαίσιο αξιολόγησης των μηχανισμών. Η βασική δομή ενός benchmark περιγράφηκε στο κεφάλαιο 3, στο οποίο αναλύσαμε το τρόπο λειτουργίας του εργαλείου αυτού.

Εκτελέσαμε δύο εκδοχές των benchmarks για μέτρηση του ρυθμού μεταφοράς. Στη πρώτη εκδοχή (touch), ο παραλήπτης επαληθεύει την εγκυρότητα των εισερχόμενων δεδομένων (συγκρίνοντας τα με μια αναμενόμενη τιμή), ούτως ώστε να βεβαιωθεί ότι δεν αλλοιώθηκαν κατά τη μετάδοση. Αυτό έχει, επιπρόσθετα, τη συνέπεια να εισαχθούν τα δεδομένα στην κρυφή μνήμη (cache) του επεξεργαστή. Στη δεύτερη εκδοχή, ο παραλήπτης λαμβάνει τα δεδομένα χωρίς να τα διαβάσει (notouch). Οι διεργασίες του αποστολέα και του παραλήπτη καρφιτσώθηκαν (pin) σε κάποιον επεξεργαστή για να εξασφαλίσουμε ότι δε θα μετακινηθούν από το λειτουργικό σύστημα σε άλλον επεξεργαστή, κατά την εκτελέση τους. Έτσι, λάβαμε μετρήσεις κατά τις οποίες οι δύο διεργασίες βρίσκονται στον ίδιο πυρήνα, στον ίδιο πυρήνα αλλά σε διαφορετικά νήματα (hyper-threading) και σε διαφορετικούς πυρήνες. Όλα τα benchmarks εκτελέστηκαν για τουλάχιστον 10000 επαναλήψεις.

Στη συνέχεια δίνουμε μια σύντομη περιγραφή των mirco-benchmarks που εκτελέσαμε:

8.2.1 pipe benchmarks

Το benchmark αυτό είναι ένα από τα προκαθορισμένα (default) benchmarks της σουίτας του ipc-bench. Δημιουργείται ένα pipe με την κλήση συστήματος pipe για τη μετάδοση των δεδομένων από τον αποστολέα στον παραλήπτη. Ισχύουν όσα αναφέρθηκαν στο κεφάλαιο 2.3.1 για τις σωληνώσεις. Οι buffers αποστολής και λήψης έχουν το μέγεθος ενός μηνύματος και επαναχρησιμοποιούνται σε κάθε αποστολή/λήψη.

Όσον αφορά το latency μετράμε την καθυστέρηση μεταφοράς μιας σελίδας δεδομένων από τον αποστολέα στον παραλήπτη και αντίστροφα. Για το σκοπό αυτό απαιτούνται δύο κανάλια επικοινωνίας, ένα από τον αποστολέα στον παραλήπτη και ένα από τον παραλήπτη στον αποστολέα. Ως εκ τούτου, δημιουργούνται δύο pipes.

8.2.2 Unix Sockets benchmarks

Κι αυτό το benchmark είναι ένα από τα προκαθορισμένα (default) benchmarks της σουίτας του ipc-bench. Δημιουργείται ένα ζεύγος συνδεδεμένων unix sockets, τα οποία χρησιμοποιούνται για την επικοινωνία των δύο διεργασιών. Ισχύουν όσα αναφέρθηκαν στο κεφάλαιο 2.3.2 για τα unix sockets. Οι buffers αποστολής και λήψης έχουν το μέγεθος ενός μηνύματος και επαναχρησιμοποιούνται σε κάθε αποστολή/λήψη.

Όσον αφορά το latency μετράμε την καθυστέρηση μεταφοράς μιας σελίδας δεδομένων από τον αποστολέα στον παραλήπτη και αντίστροφα.

8.2.3 vmsplice benchmarks

Κι αυτό το benchmark είναι ένα από τα προκαθορισμένα (default) benchmarks της σουίτας του ipc-bench. Τα δεδομένα μεταφέρονται μέσω μιας σωλήνωσης που δημιουργείται με την κλήση συστήματος pipe. Ισχύουν όσα αναφέρθηκαν στο κεφάλαιο 2.3.3 για το vmsplice.

Παρέχονται δύο διαφορετικά benchmarks για τη μέτρηση του throughput. Όπως έχουμε αναφέρει, όταν χρησιμοποιούμε το vmsplice δεν είναι εφικτό να μάθουμε πότε τα απεσταλμένα δεδομένα καταναλώθηκαν από τον παραλήπτη και είναι ασφαλές να επαναχρησιμοποιήσουμε τους buffers αποστολής. Το πρώτο benchmark δεσμεύει ένα κομμάτι μνήμης 2 MB με την mmap, το οποίο χρησιμοποιεί για να τοποθετεί τα μηνύματα προς αποστολή. Όταν η μνήμη αυτή εξαντληθεί καλεί την munmap για την αποδέσμευσή της και δεσμεύει νέα μνήμη με την mmap. Αυτή η διαδικασία επαναλαμβάνεται όσες φορές χρειαστεί για τη μετάδοση του συνόλου των δεδομένων. Με τον τρόπο αυτό εξασφαλίζεται ότι δε θα αλλοιωθούν τα δεδομένα που έχουν σταλεί. Το δεύτερο benchmark χρησιμοποιεί ένα δεύτερο, βοηθητικό pipe, μέσω του οποίου ο παραλήπτης ενημερώνει τον αποστολέα πότε έχει τελειώσει με την επεξεργασία των εισερχόμενων σελίδων. Με τον τρόπο αυτό επιτυγχάνεται η ασφαλής επαναχρησιμοποίηση των buffers και αποφεύγεται η συνεχής δέσμευση και αποδέσμευση μνήμης.

Όσον αφορά το latency μετράμε την καθυστέρηση μεταφοράς μιας σελίδας δεδομένων από τον αποστολέα στον παραλήπτη και αντίστροφα. Για το σκοπό αυτό απαιτούνται δύο κανάλια επικοινωνίας, ένα από τον αποστολέα στον παραλήπτη και ένα από τον παραλήπτη στον αποστολέα. Ως εκ τούτου, δημιουργούνται δύο pipes.

8.2.4 Shared Memory benchmarks

Κι αυτό το benchmark είναι ένα από τα προκαθορισμένα (default) benchmarks της σουίτας του ipc-bench και στη συνέχεια θα αναφέρεται με το όνομα mempipe. Δημιουργείται ένα κομμάτι μοιραζόμενης μνήμης 2 MB, με χρήση του POSIX shared memory API, το οποίο χρησιμοποιούν οι διεργασίες για να επικοινωνήσουν. Το συγκεκριμένο benchmark χειρίζεται τη μοιραζόμενη μνήμη ως έναν κυκλικό buffer που αποθηκεύει μηνύματα μεταβλητού μήκους. Κάθε μήνυμα φέρει μια κεφαλίδα με δύο πεδία: μία σημαία που καθορίζει αν το μήνυμα είναι έτοιμο προς κατανάλωση και έναν ακέραιο που δίνει το μήκος του μηνύματος. Τα δεδομένα του μηνύματος ακολουθούν αμέσως μετά την κεφαλίδα. Στις περιπτώσεις που ο buffer είναι γεμάτος/άδειος ο αποστολέας/παραλήπτης μπλοκάρει περιμένοντας να ελευθερωθεί χώρος ή να τοποθετηθεί κάποιο μήνυμα αντίστοιχα. Για το σκοπό αυτό χρησιμοποιείται η κλήση συστήματος futex του Linux.

Όσον αφορά το latency, το μετράμε θέτοντας τιμές σε σημαίες που βρίσκονται στο κομμάτι της μοιραζόμενης μνήμης. Η πρώτη διεργασία θέτει την τιμή μιας σημαίας, ενώ η άλλη εκτελεί έναν βρόγχο (spins) μέχρι να αλλάξει τιμή η σημαία και στη συνέχεια αλλάζει, με τη σειρά της, την τιμή μιας άλλης σημαίας, που διαβάζεται από την πρώτη διεργασία. Δεν πραγματοποιείται μεταφορά δεδομένων.

8.2.5 zipc benchmarks

Αποστολέας και παραλήπτης ανοίγουν το ίδιο αρχείο συσκευής του zipc, ο ένας για εγγραφή και ο άλλος για ανάγνωση, αντίστοιχα. Ισχύουν όσα αναφέρθηκαν για το zipc στα κεφάλαια 4 και 5.

Όσον αφορά τη μέτρηση του throughput, και οι δύο δεσμεύουν με την κλήση συστήματος mmap μνήμη ικανή να χωρέσει ένα μήνυμα. Στη συνέχεια ο αποστολέας μεταφέρει τα δεδομένα στον παραλήπτη με την κλήση συστήματος writev και ο παραλήπτης τα λαμβάνει με την κλήση συστήματος readv.

Όσον αφορά το latency μετράμε την καθυστέρηση μεταφοράς μιας σελίδας δεδομένων από τον αποστολέα στον παραλήπτη και αντίστροφα. Για το σκοπό αυτό απαιτούνται δύο κανάλια επικοινωνίας, ένα από τον αποστολέα στον παραλήπτη και ένα από τον παραλήπτη στον αποστολέα. Ως εκ τούτου, ανοίγονται δύο αρχεία συσκευής του zipc.

8.2.6 zipc-tmpfs benchmarks

Αποστολέας και παραλήπτης ανοίγουν το ίδιο αρχείο συσκευής του zipc-tmpfs, ο ένας για εγγραφή και ο άλλος για ανάγνωση αντίστοιχα. Ισχύουν όσα αναφέρθηκαν για το zipc-tmpfs στα κεφάλαια 6 και 7.

Όσον αφορά τη μέτρηση του throughput, ο αποστολέας δεσμεύει με την allocate_buffer έναν buffer 2 MB, ικανό να αποθηκεύσει πολλαπλά μηνύματα. Τα μηνύματα στέλνονται με χρήση της κλήσης συστήματος writev. Μόλις ο buffer αποστολής εξαντληθεί δεσμεύεται νέος και η διαδικασία αυτή επαναλαμβάνεται όσες φορές χρειαστεί για τη μετάδοση του συνόλου των δεδομένων. Ο παραλήπτης λαμβάνει δεδομένα με χρήση της συνάρτησης receive_msg. Μόλις καταναλώσει ένα μήνυμα αποδεσμεύει την αντίστοιχη μνήμη με χρήση της κλήσης συστήματος munmap.

Όσον αφορά το latency μετράμε τη καθυστέρηση μεταφοράς μιας σελίδας δεδομένων από τον αποστολέα στον παραλήπτη και αντίστροφα. Για το σκοπό αυτό απαιτούνται δύο κανάλια επικοινωνίας, ένα από τον αποστολέα στον παραλήπτη και ένα από τον παραλήπτη στον αποστολέα. Ως εκ τούτου, ανοίγονται δύο αρχεία συσκευής του zipc-tmpfs.

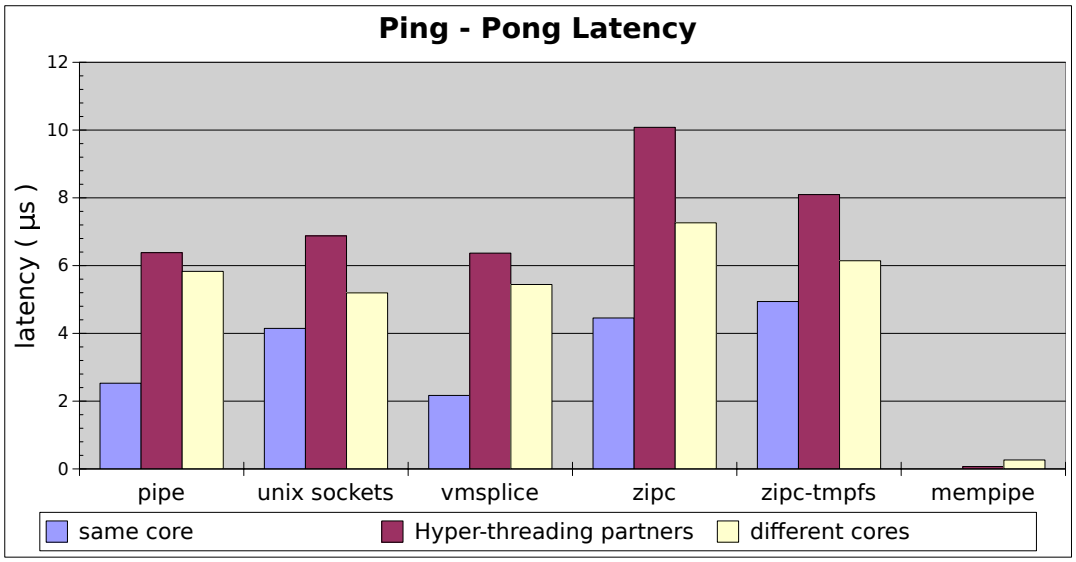
8.3 Macro-benchmark

Τα benchmarks που εξετάσαμε στην προηγούμενη ενότητα μας δίνουν μια εκτίμηση των επιδόσεων των αντίστοιχων μηχανισμών IPC στην περίπτωση που μόνο δύο πυρήνες επικοινωνούν μεταξύ τους. Ωστόσο, τα αποτελέσματα αυτών των μετρήσεων δεν είναι απολύτως αντιπροσωπευτικά των πραγματικών επιδόσεων των εφαρμογών, ειδικά κατά την παρουσία κορεσμού στο υποσύστημα μνήμης. Για το λόγο αυτό, και για να εκτιμήσουμε καλύτερα τις επιδόσεις των μηχανισμών κάτω από πραγματικές συνθήκες, εκτελέσαμε τις μετρήσεις για το throughput υπό συνθήκες κορεσμού μνήμης (memory contention), δηλαδή τρία ζεύγη πυρήνων επικοινωνούν ταυτόχρονα ανταλλάσσοντας μηνύματα με χρήση του εκάστοτε μηχανισμού.

8.4 Πειραματικά Αποτελέσματα

8.4.1 Latency

Στο σχήμα 8.1 φαίνονται τα αποτελέσματα των μετρήσεων για την καθυστέρηση μεταφοράς (latency) σε μsec. Σημειώνουμε ότι για την περίπτωση του mempipe δε δίνουμε το latency για την εκτέλεση των δύο διεργασιών στον ίδιο πυρήνα. Λόγω της φύσης του συγκεκριμένου benchmark και του τρόπου που οι διεργασίες περιμένουν για αλλαγή στις σημαίες (spinning: βλέπε 8.2.4) το αποτέλεσμα δεν έχει αξία για εκτέλεση στον ίδιο πυρήνα.

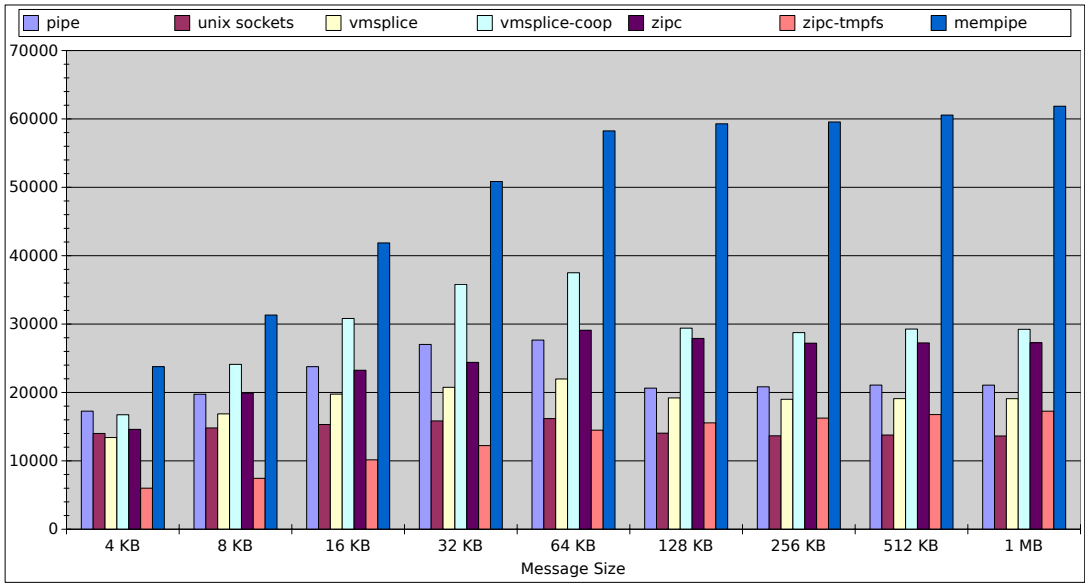


Σχήμα 8.1: *Latency*

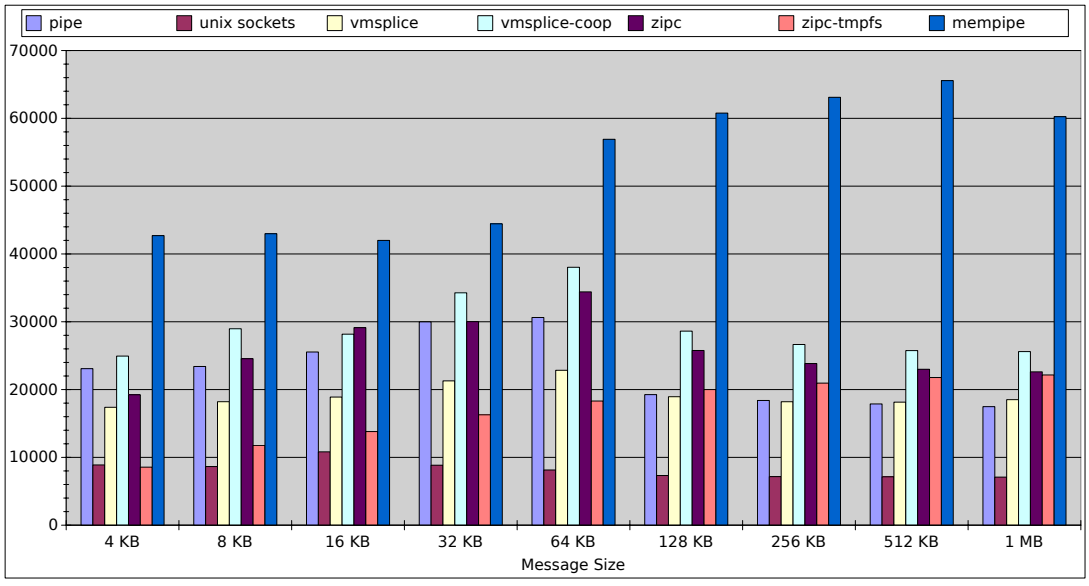
8.4.2 Throughput

Στη συνέχεια δίνονται τα αποτελέσματα των μετρήσεων για το ρυθμό μεταφοράς, για τις δύο εκδοχές touch και notouch, σε Mbps. Επίσης, δίνονται τα αποτελέσματα για την εκτέλεση υπό συνθήκες κορεσμού του υποσυστήματος μνήμης.

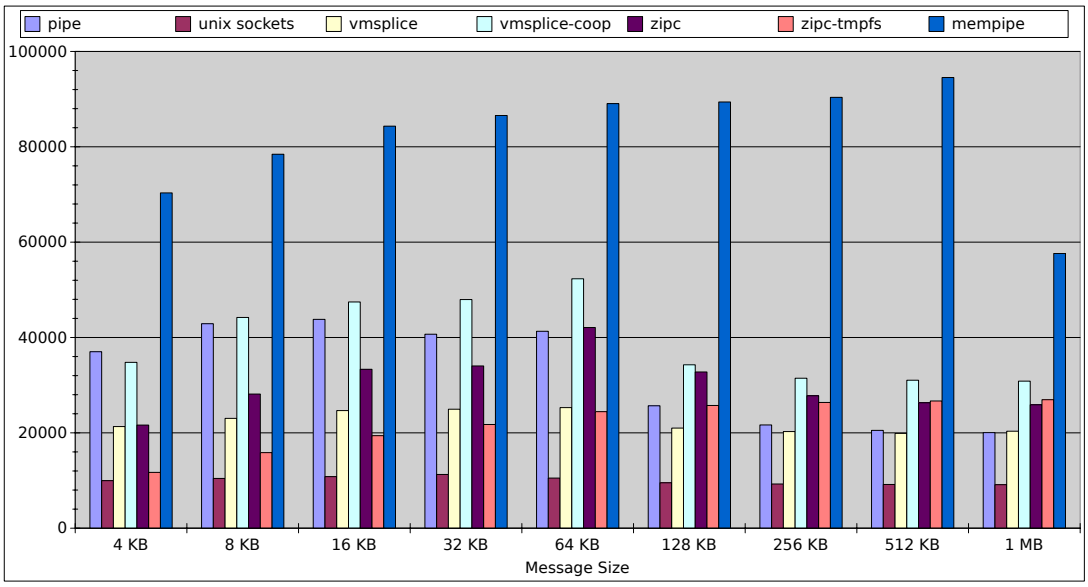
Touch



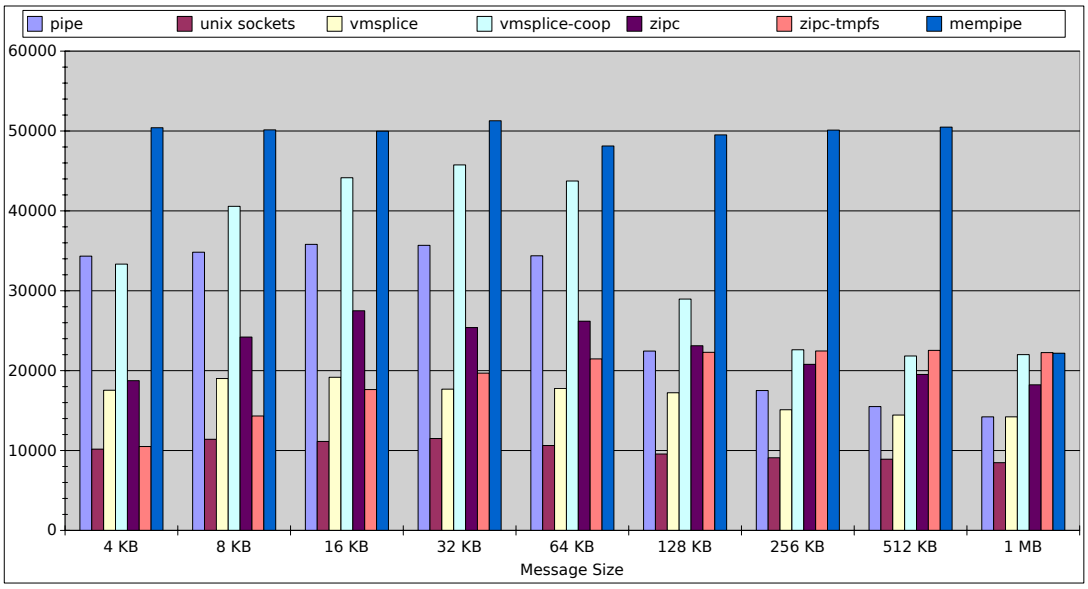
Σχήμα 8.2: *Throughput (Mbps) - Εκτέλεση στον ίδιο πυρήνα*



Σχήμα 8.3: *Throughput (Mbps) - Hyper-Threading Partners*

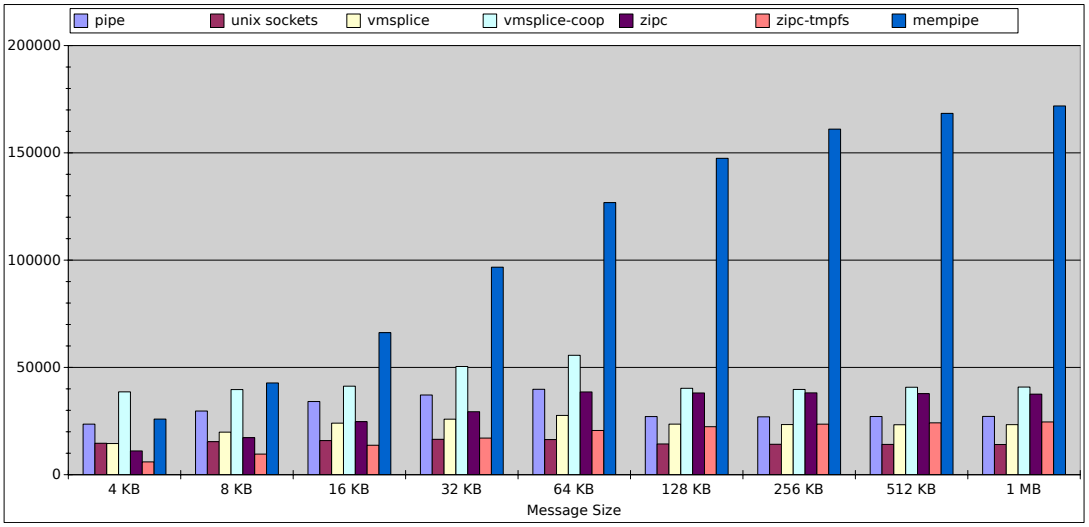


Σχήμα 8.4: *Throughput (Mbps) - Εκτέλεση σε διαφορετικούς πυρήνες*

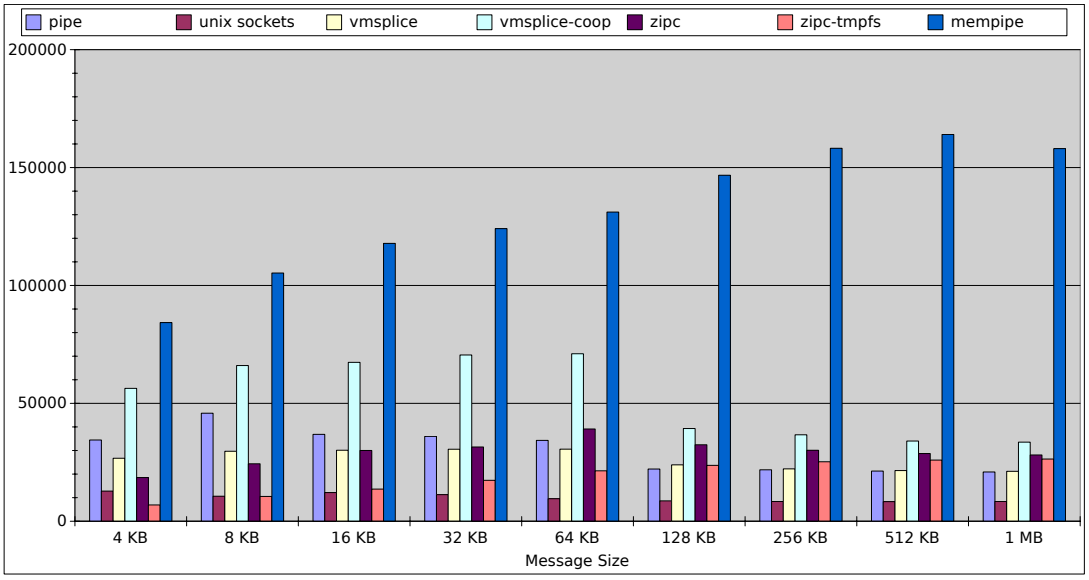


Σχήμα 8.5: *Throughput (Mbps) - Τρία ζεύγη πυρήνων (Memory Contention)*

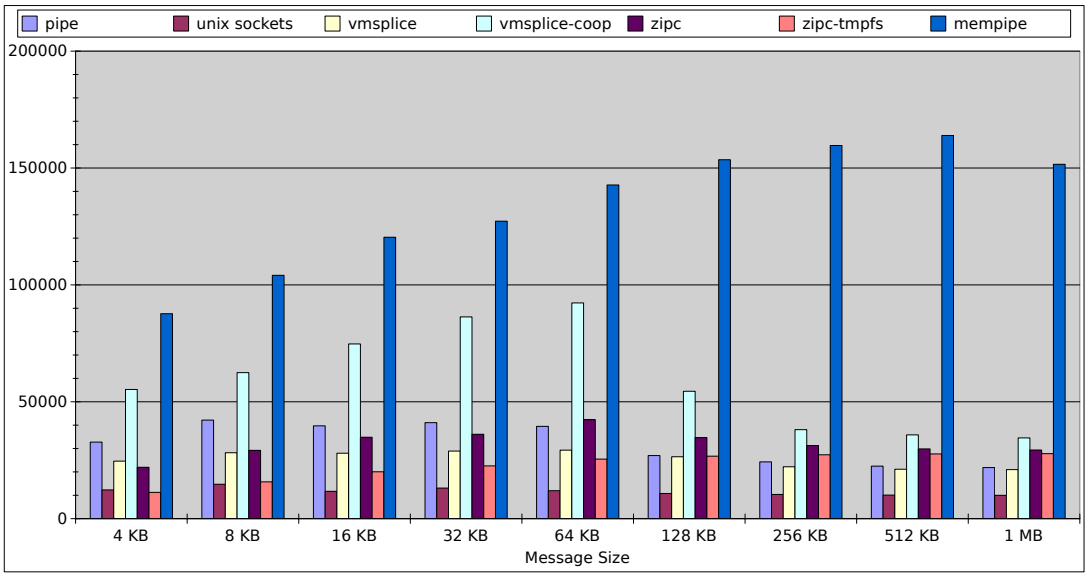
Notouch



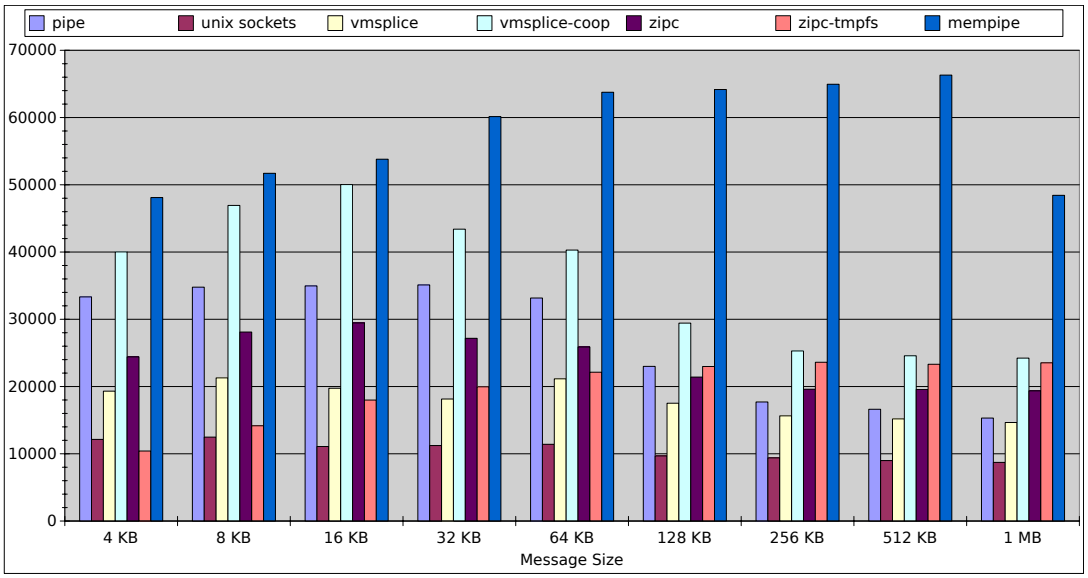
Σχήμα 8.6: *Throughput (Mbps) - Εκτέλεση στον ίδιο πυρήνα*



Σχήμα 8.7: *Throughput (Mbps) - Hyper-Threading Partners*



Σχήμα 8.8: *Throughput (Mbps) - Εκτέλεση σε διαφορετικούς πυρήνες*



Σχήμα 8.9: *Throughput (Mbps) - Τρία ζεύγη πυρήνων (Memory Contention)*

8.5 Σχολιασμός Αποτελεσμάτων

8.5.1 Latency

Το χειρότερο latency παρατηρείται στην περίπτωση που οι διεργασίες εκτελούνται στον ίδιο πυρήνα με χρήση του μηχανισμού Hyper-Threading. Κάτι τέτοιο είναι λογικό και αναμενόμενο. Πρόκειται για πανομοιότυπες διεργασίες που χρησιμοποιούν τις ίδιες

λειτουργικές μονάδες (execution units) του επεξεργαστή. Αυτό έχει ως αποτέλεσμα να υπάρχει έντονος συναγωνισμός για τους πόρους του επεξεργαστή, γεγονός που ρίχνει αισθητά τις επιδόσεις. Εξαίρεση αποτελεί το mempipe που παρουσιάζει καλύτερο latency στην περίπτωση αυτή. Το benchmark αυτό είναι ιδιαίτερα απλό και συνίσταται από λίγες εντολές που αλλάζουν κάποιες θέσεις μνήμης. Οπότε οι πόροι του επεξεργαστή επαρκούν για τη ταυτόχρονη εκτέλεση των δύο διεργασιών, οι οποίες επωφελούνται από το γεγονός ότι χρησιμοποιούν την ίδια ιεραρχία κρυφής μνήμης.

Το καλύτερο latency για όλα τα benchmarks, με εξαίρεση το mempipe, παρατηρείται στην περίπτωση της εκτέλεσης στον ίδιο πυρήνα. Στα benchmarks αυτά τα δεδομένα που μεταφέρονται έχουν μικρό μέγεθος και ολόκληρο το data set χωράει στη level 1 cache του επεξεργαστή. Αυτό έχει ως αποτέλεσμα την επίτευξη πολύ καλών επιδόσεων.

Η εκτέλεση σε διαφορετικούς πυρήνες επιτυγχάνει επιδόσεις ανάμεσα στις δύο προηγούμενες περιπτώσεις. Εδώ, για τη μεταφορά μιας σελίδας από τη μια διεργασία στην άλλη απαιτούνται περισσότερες προσβάσεις στη κύρια μνήμη, μιας και οι δύο επεξεργαστές έχουν διαφορετική ιεραρχία κρυφών μνημών.

Το mempipe πετυχαίνει το χαμηλότερο latency. Αντίθετα, οι μηχανισμοί μας παρουσιάζουν το μεγαλύτερο latency. Το γεγονός αυτό αποτελεί ένδειξη ότι το κόστος των μηχανισμών αυτών για τη μεταφορά μιας σελίδας (στην περίπτωση του zipc καρφίτσωμα των σελίδων και τροποποίηση των πινάκων σελίδων, ενώ στην περίπτωση του zipc-tmpfs δημιουργία και καταστροφή περιοχών εικονικής μνήμης) είναι μεγάλο και δεν αξίζει για μεταφορά μικρών ποσοτήτων δεδομένων.

8.5.2 Throughput

Micro-benchmarks

Σε όλα τα σχήματα της ενότητας 8.4.2, παρατηρούμε ότι τα benchmarks pipe, vmsplice, vmsplice-coop και zipc έχουν μια πτώση στο ρυθμό μεταφοράς για μηνύματα μεγαλύτερα των 64 KB. Όλοι αυτοί οι μηχανισμοί χρησιμοποιούν μια σωλήνωση χωρητικότητας 64 KB για τη μετάδοση των δεδομένων. Επομένως, μετά τα 64 KB, κάθε μήνυμα γεμίζει πολλές φορές την υποκείμενη σωλήνωση μέχρι να μεταδοθεί ολόκληρο. Ως εκ τούτου, ο αποστολέας ξοδεύει αρκετό χρόνο μπλοκαρισμένος, περιμένοντας τον παραλήπτη να αδειάσει χώρο στη σωλήνωση.

Εκτέλεση σε διαφορετικούς πυρήνες Όπως ήταν αναμενόμενο το mempipe εμφανίζει τις καλύτερες επιδόσεις. Τα unix sockets, αν και έχουν σχεδόν σταθερό ρυθμό μεταφοράς για τα διάφορα μεγέθη του μηνύματος, παρουσιάζουν το χειρότερο throughput. Το vmsplice-coop (single copy) είναι το δεύτερο καλύτερο, μετά το mempipe, εκτός από την περίπτωση μηνυμάτων των 4 KB. Για τα μηνύματα αυτού του μεγέθους το pipe έχει ελαφρώς καλύτερο throughput. Ο λόγος γι᾽ αυτό είναι ότι ο buffer αποστολής του pipe είναι μικρότερος από αυτόν του vmsplice-coop και επομένως αξιοποιείται καλύτερα η κρυφή μνήμη του επεξεργαστή. Όσον αφορά τους μηχανισμούς μας, δεν παρουσιάζουν τις αναμενόμενες επιδόσεις. Το zipc ξεπερνάει το pipe (που δημιουργεί δύο αντίγραφα) για μηνύματα μεγέθους 64 KB και πάνω, ενώ το zipc-tmpfs για μηνύματα μεγέθους 256 KB και πάνω. Κανένα από τα δύο, όμως, δεν φτάνει το ρυθμό μεταφοράς του vmsplice-coop, που δημιουργεί ένα αντίγραφο.

Hyper-Threading Partners Τα αποτελέσματα είναι αντίστοιχα με αυτά της προηγούμενης περίπτωσης για εκτέλεση σε διαφορετικούς πυρήνες. Σε σχέση, βέβαια, με την προηγούμενη περίπτωση παρατηρούμε μειωμένο ρυθμό μετάδοσης σε όλα τα benchmarks. Κάτι τέτοιο είναι λογικό και αναμενόμενο, αφού η εκτέλεση με χρήση του μηχανισμού hyper-threading είναι εν γένει υποδεέστερη σε σχέση με την εκτέλεση σε διαφορετικούς φυσικούς πυρήνες.

Εκτέλεση στον ίδιο πυρήνα Παρατηρούμε ορισμένες διαφοροποιήσεις σε σχέση με την εκτέλεση σε διαφορετικούς πυρήνες. Τα unix sockets παρουσιάζουν καλύτερες επιδόσεις από το zipc-tmpfs για μηνύματα μεγέθους μέχρι και 64 KB. Το zipc-tmpfs, από την άλλη, δεν καταφέρνει να ξεπεράσει το throughput του pipe και συνολικά εμφανίζει τη χειρότερη συμπεριφορά. Ο λόγος γι᾽ αυτό είναι η κακή αξιοποίηση της κρυφής μνήμης του επεξεργαστή από το zipc-tmpfs, αφού σε κάθε αποστολή δεσμεύονται καινούρια πλαίσια φυσικής μνήμης. Στην περίπτωση της εκτέλεσης στον ίδιο πυρήνα η κακή αυτή αξιοποίηση της cache γίνεται πιο εμφανής, αφού αποστολέας και παραλήπτης μοιράζονται την ίδια ιεραρχία κρυφών μνημών.

Στην περίπτωση των notouch benchmarks παρατηρούμε όμοια συμπεριφορά με τα touch benchmarks, με τη διαφορά ότι ο ρυθμός μεταφοράς για όλα τα benchmarks είναι αυξημένος. Αυτό ήταν αναμενόμενο, αφού στην περίπτωση αυτή ο παραλήπτης δε διαβάζει και δεν επαληθεύει τα εισερχόμενα δεδομένα.

Macro-benchmark

Στην περίπτωση αυτή όλα τα benchmarks εμφανίζουν μειωμένο ρυθμό μετάδοσης, λόγω του κορεσμού του υποσυστήματος μνήμης. Η συμπεριφορά των benchmarks είναι ανάλογη με την περίπτωση της εκτέλεσης σε διαφορετικούς πυρήνες, που εξετάσαμε πριν, με τις ακόλουθες διαφορές. Το zipc ξεπερνάει το pipe μετά τα 128 KB. Το zipc-tmpfs, για μηνύματα μεγέθους 512 KB και 1 MB, ξεπερνάει ελάχιστα το vmsplice-coop. Τέλος, παρατηρούμε ότι για μηνύματα μεγέθους 1 MB τα benchmarks vmsplice-coop, zipc-tmpfs και mempipe εμφανίζουν περίπου τον ίδιο ρυθμό μετάδοσης. Η απότομη πτώση του ρυθμού μετάδοσης του mempipe, στην περίπτωση touch μόνο, εξηγείται ως εξής: Για μηνύματα αυτού του μεγέθους ο κυκλικός buffer, που μοιράζονται αποστολέας και παραλήπτης, χωράει δύο μηνύματα. Παράλληλα, λόγω του κορεσμού του υποσυστήματος μνήμης, μειώνεται ο ρυθμός ανάγνωσης του παραλήπτη (read throughput). Επομένως, ο αποστολέας θα μπλοκάρει συχνότερα (με χρήση του μηχανισμού futex) περιμένοντας τον παραλήπτη να ελευθερώσει χώρο στον buffer. Σε δοκιμή που πραγματοποιήσαμε, με μεγαλύτερο μέγεθος για το κομμάτι μοιραζόμενης μνήμης, διαπιστώσαμε ότι πράγματι το mempipe παρουσιάζει μεγαλύτερο throughput από τα υπόλοιπα benchmarks για μηνύματα του 1 MB.

8.6 Συμπεράσματα

Παρατηρούμε ότι το mempipe παρουσιάζει τις καλύτερες επιδόσεις σε θέμα latency και throughput. Αυτό ήταν αναμενόμενο μιας και η μοιραζόμενη μνήμη δίνει στις διεργασίες τη δυνατότητα να επικοινωνήσουν χωρίς τη δημιουργία αντιγράφων και με τις ελάχιστες δυνατές επιβαρύνσεις.

Ο δεύτερος καλύτερος μηχανισμός διαδιεργασιακής επικοινωνίας ήταν το vmsplice. Το vmsplice επιτρέπει την απευθείας αντιγραφή των δεδομένων από μνήμη του αποστολέα σε μνήμη του παραλήπτη, δημιουργώντας έτσι ένα μόνο αντίγραφο. Η επιβάρυνση που εισάγει, για να πετύχει την εξάλειψη του ενός από τα δύο αντίγραφα, συνίσταται στο καρφίτσωμα στη μνήμη των σελίδων, που περιέχουν τα δεδομένα προς μετάδοση. Αυτό απαιτεί τη διάσχιση των πινάκων σελίδων της διεργασίας, για το αντίστοιχο εύρος εικονικών διευθύνσεων, και την εκτέλεση ατομικών εντολών. Όπως αποδεικνύεται από τις μετρήσεις, το κόστος αυτής της λειτουργίας είναι τελικά μικρότερο από τη δη-

μιουργία ενός αντιγράφου.

Οι μηχανισμοί zipc και zipc-tmpfs, που αναπτύξαμε στην παρούσα εργασία, δεν είχαν τις αναμενόμενες επιδόσεις. Παρόλο που δεν δημιουργούν αντίγραφα, κατά τη μετάδοση των δεδομένων, δεν κατάφεραν να ξεπεράσουν το vmpslice, που δημιουργεί ένα αντίγραφο. Για μεγάλα μόνο μηνύματα (πάνω από 64 KB, 128 KB ή 256 KB κατά περίπτωση) κατάφεραν να ξεπεράσουν το pipe, που δημιουργεί δύο αντίγραφα.

Στις μετρήσεις του throughput το zipc αποδείχθηκε καλύτερο από το zipc-tmpfs, με εξαίρεση την περίπτωση μεγάλων μηνυμάτων (πάνω από 256 KB) για εκτέλεση σε διαφορετικούς πυρήνες και υπό συνθήκες κορεσμού μνήμης. Όσον αφορά το latency, το zipc-tmfps ήταν καλύτερο από το zipc, με εξαίρεση την εκτέλεση στον ίδιο πυρήνα.

Η αποφυγή των αντιγράφων αναμέναμε να επιτύχει πολύ καλύτερες επιδόσεις. Τουλάχιστον καλύτερες από τις περιπτώσεις των δύο και του ενός αντιγράφου. Οι λόγοι για το αντιφατικό αυτό αποτέλεσμα φαίνεται να είναι η κακή διαχείριση της κρυφής μνήμης και η επιβάρυνση που εισάγουν οι μηχανισμοί αυτοί σε κάθε αποστολή και λήψη ενός μηνύματος.

Το zipc επιβαρύνει την αποστολή ενός μηνύματος με τα εξής:

1. Καρφίτσωμα στη μνήμη των σελίδων που περιέχουν τα δεδομένα προς μετάδοση. Αυτό απαιτεί τη διάσχιση των πινάκων σελίδων της διεργασίας, για το αντίστοιχο εύρος εικονικών διευθύνσεων, και την εκτέλεση ατομικών εντολών.

2. Διάσχιση των πινάκων σελίδων του αποστολέα, για το αντίστοιχο εύρος εικονικών διευθύνσεων, με στόχο την καταστροφή των καταχωρήσεων για τα απεσταλμένα πλαίσια και την απεικόνιση νέων πλαισίων στις θέσεις αυτές. Αυτό περιλαμβάνει, μεταξύ άλλων, ορισμένες ιδιαίτερα χρονοβόρες λειτουργίες: Απόκτηση των σχετικών κλειδωμάτων, που προστατεύουν τις δομές που αλλάζουμε, εκκαθάριση της κρυφής μνήμης TLB (TLB flush) και δέσμευση φυσικής μνήμης.

Το zipc επιβαρύνει τη λήψη ενός μηνύματος με τα εξής:

- Διάσχιση των πινάκων σελίδων του παραλήπτη, για το αντίστοιχο εύρος εικονικών διευθύνσεων, με στόχο την καταστροφή των καταχωρήσεων για τυχόν πλαίσια που βρίσκονται ήδη εκεί και την απεικόνιση των εισερχόμενων πλαισίων στις

θέσεις αυτές. Αυτό περιλαμβάνει, μεταξύ άλλων, ορισμένες ιδιαίτερα χρονοβόρες λειτουργίες: Απόκτηση των σχετικών κλειδωμάτων, που προστατεύουν τις δομές που αλλάζουμε, και εκκαθάριση της κρυφής μνήμης TLB (TLB flush).

Επιπρόσθετα, κατά την αποστολή και λήψη μηνυμάτων, επιβαρυνόμαστε με cache και tlb misses. Τα πλαίσια φυσικής μνήμης που αποτελούν τους buffers αποστολής και λήψης αλλάζουν συνεχώς, οπότε δεν αξιοποιούμε σχεδόν καθόλου την κρυφή μνήμη. Η εκκαθάριση του TLB και η δημιουργία νέων απεικονίσεων (λόγω της δέσμευσης νέων πλαισίων) είναι υπεύθυνες για τα tlb misses, τα οποία είναι ιδιαίτερα κοστοβόρα. Ειδικά στην περίπτωση πολυ-πύρηνων μηχανημάτων, αν η διεργασία είναι πολυ-νηματική και νήματα της εκτελούνται σε διάφορους πυρήνες, απαιτείται η εκκαθάριση των tlb caches όλων αυτών των πυρήνων. Αυτό εμπεριέχει την αποστολή δια-επεξεργαστικών διακοπών (IPI), όπως εξηγήσαμε στο κεφάλαιο 2.2.9.

Το zipc-tmpfs επιβαρύνει την αποστολή ενός μηνύματος με τα εξής:

1. Δέσμευση του buffer αποστολής, που υποστηρίζεται από ένα αρχείο στο tmpfs. Αυτό περιλαμβάνει τη δημιουργία του αρχείου στο tmpfs, την απόκτηση των σχετικών κλειδωμάτων και την απεικόνιση του αρχείου στο χώρο διευθύνσεων του αποστολέα.

2. Διάσχιση των περιοχών εικονικής μνήμης, που απεικονίζουν τα δεδομένα προς αποστολή, με στόχο την κατασκευή ενός μηνύματος.

3. Καταστροφή αυτών των περιοχών εικονικής μνήμης, κατά την αποστολή ενός μηνύματος. Αν η διεργασία είχε διαβάσει/γράψει στις υποκείμενες σελίδες μνήμης τότε απαιτείται, επιπλέον, διάσχιση των πινάκων σελίδων και κατάργηση των αντίστοιχων απεικονίσεων από εκεί. Αυτές οι εργασίες περιλαμβάνουν, μεταξύ άλλων, ορισμένες ιδιαίτερα χρονοβόρες λειτουργίες: Απόκτηση των σχετικών κλειδωμάτων, που προστατεύουν τις δομές που αλλάζουμε, και πιθανή εκκαθάριση της κρυφής μνήμης TLB.

Το zipc-tmpfs επιβαρύνει τη λήψη ενός μηνύματος με τα εξής:

1. Απόκτηση των κλειδωμάτων, που προστατεύουν τις δομές που θα αλλάξουμε.

2. Εύρεση και δέσμευση μιας περιοχής διευθύνσεων, στο χώρο εικονικών διευθύνσεων του παραλήπτη, ικανής να χωρέσει το εισερχόμενο μήνυμα.

3. Δημιουργία διαδοχικών περιοχών εικονικής μνήμης, που καλύπτουν τις διευθύνσεις που δεσμεύσαμε στο προηγούμενο βήμα, και απεικόνιση σε αυτές των διαδοχικών μερών του μηνύματος.

Επιπρόσθετα, κατά την αποστολή και λήψη μηνυμάτων, επιβαρυνόμαστε με cache και tlb misses. Οι λόγοι είναι οι ίδιοι με την περίπτωση του zipc, που αναλύσαμε προηγουμένως.

Οι παραδοσιακές διεπαφές Εισόδου / Εξόδου του Linux, όπως οι κλήσεις συστήματος read και write, βασίζονται στην αντιγραφή των δεδομένων (copy semantics - βλέπε κεφάλαια 4.5 και 6.4). Η απόφαση να διατηρήσουμε αυτή τη σημασιολογία στους μηχανισμούς που αναπτύξαμε, μας αναγκάζει να κατασκευάζουμε και να καταστρέφουμε απεικονίσεις εικονικής μνήμης, σε κάθε λειτουργία αποστολής/λήψης. Το κόστος της απόκτησης των κλειδωμάτων, που σχετίζονται με τις δομές του υποσυστήματος μνήμης, της αλλαγής των σχετικών απεικονίσεων στον πυρήνα και της εκτέλεσης ενεργειών σχετικών με τη συνέπεια της κρυφής μνήμης και του TLB, περιορίζει τις επιδόσεις των μηχανισμών μας και καταργεί τα πλεονεκτήματα των μηδενικών αντιγράφων.

9

Παρεμφερείς Εργασίες

Η μείωση του κόστους που οφείλεται στην αντιγραφή των δεδομένων έχει αποτελέσει το θέμα αρκετών ερευνητικών εργασιών και έχει ένα ευρύ φάσμα εφαρμογών: Διαδιεργασιακή επικοινωνία, δικτυακές στοίβες, συστήματα Εισόδου / Εξόδου. Οπουδήποτε υπάρχει ανάγκη για μεταφορά δεδομένων ανάμεσα σε δύο οντότητες, η μείωση των αντιγράφων είναι ένα σημαντικό βήμα για την επίτευξη υψηλών επιδόσεων.

Στην [RBV04] γίνεται μια προσπάθεια υλοποίησης ενός μονοπατιού μηδενικών αντιγράφων (zero-copy path) ανάμεσα σε μια δικτυακή διεπαφή και το χώρο χρήστη. Για το σκοπό αυτό χρησιμοποιείται μια τεχνική page-flipping που μεταφέρει ένα πλαίσιο μνήμης από το χώρο πυρήνα στο χώρο χρήστη μέσω τροποποίησης των πινάκων σελίδων. Οι συγγραφείς καταλήγουν στο συμπέρασμα ότι η δημιουργία ενός αντιγράφου είναι πιο ωφέλιμη από τη τροποποίηση των πινάκων σελίδων.

Ιδιαίτερο ενδιαφέρον παρουσιάζουν οι *fbufs* [KT95, DP93]. Στις εργασίες αυτές γίνεται μια προσπάθεια επέκτασης των παραδοσιακών προγραμματιστικών διεπαφών του Unix, οι οποίες βασίζονται στην αντιγραφή των δεδομένων. Οι *fbufs* εισάγουν την έννοια της ανταλλαγής buffers: Αντί να μεταφέρονται δεδομένα, ανάμεσα στο χώρο χρήστη και στο χώρο πυρήνα, μεταφέρονται buffers. Στόχος των *fbufs* είναι η δημιουργία ενός συστήματος Εισόδου / Εξόδου για το Unix, το οποίο δε θα περιορίζεται μόνο στην περίπτωση των δικτυακών πρωτοκόλλων αλλά θα μπορεί να χρησιμοποιηθεί για οποιαδήποτε λειτουργία Εισόδου / Εξόδου. Οι *fbufs* καταφέρνουν, υπό ορισμένες συνθήκες, να βελτιώσουν τις επιδόσεις των εφαρμογών που τους χρησιμοποιούν αλλά κάνουν αρκετές υποθέσεις και θέτουν περιορισμούς που δεν είναι πάντα αποδεκτοί από τις εφαρμογές.

Στην [SSCZ12] εξετάζεται η περίπτωση των εφαρμογών web-caching. Οι εφαρμογές αυτές στέλνουν δεδομένα τα οποία μεταβάλλονται σπάνια. Αντίθετα, αυτό που μεταβάλλεται είναι τα μετα-δεδομένα, όπως κεφαλίδες, που τα συνοδεύουν. Στην εργασία αυτή προτείνεται η χρήση ενός δεύτερου memory allocator για τη δέσμευση μνήμης για τα δεδομένα που μεταβάλλονται σπάνια. Με κατάλληλη τροποποίηση της δικτυακής στοίβας είναι εφικτή η μετάδοση των δεδομένων αυτών χωρίς να δημιουργούνται αντίγραφα. Ταυτόχρονα τα δεδομένα προστατεύονται με χρήση του μηχανισμού COW προκειμένου να διατηρηθεί η σημασιολογία των αντίστοιχων κλήσεων συστήματος. Η χρήση του μηχανισμού αυτού σε εφαρμογές web-caching, όπως για παράδειγμα το Memcached, είχε ως αποτέλεσμα τη βελτίωση του ρυθμού μετάδοσης.

10

Επεκτάσεις - Μελλοντική Εργασία

Αρχικά, επιθυμούμε να αναλύσουμε την εκτέλεση των μηχανισμών zipc και zipc-tmpfs με κάποιο εργαλείο όπως το perf. Το perf παρέχει, μεταξύ άλλων, πρόσβαση στους performace counters του επεξεργαστή. Η ανάλυση αυτών των στοιχείων θα επιτρέψει την καλύτερη σκιαγράφηση και κατανόηση του προφίλ εκτέλεσης των μηχανισμών αυτών. Ως εκ τούτου, θα μπορέσουμε να εντοπίσουμε τα σημεία στα οποία χάνεται ο περισσότερος χρόνος (bottlenecks) και να κατανοήσουμε τους λόγους αυτής της συμπεριφοράς. Αυτό θα μας επιτρέψει να βελτιστοποιήσουμε τους μηχανισμούς μας στο έπακρο.

Μια πιθανή επέκταση είναι η άρση των περιορισμών στη μνήμη προέλευσης και προορισμού των δεδομένων (buffer independence). Αυτό θα επιτρέψει στους μηχανισμούς μας να μεταφέρουν δεδομένα ανεξάρτητα από πού προέρχονται ή πού πηγαίνουν και ως εκ τούτου θα επεκτείνει το πεδίο εφαρμογών.

Η προσπάθεια διατήρησης της σημασιολογίας των παραδοσιακών διεπαφών του Linux για Είσοδο / Έξοδο μας ανάγκασε να χρησιμοποιήσουμε μια τεχνική page-remapping για τη μεταφορά των σελίδων από τον αποστολέα στον παραλήπτη. Το κόστος του page-remapping είναι τελικά μεγαλύτερο από το να δημιουργήσουμε ένα αντίγραφο των δεδομένων. Ως επόμενο βήμα, θα ελαττώσουμε τις απαιτήσεις μας για χρήση των copy semantics. Αντ' αυτού θα σχεδιάσουμε έναν μηχανισμό που θα υλοποιεί μια λογική παρόμοια με το vmsplice ή τις κλήσεις συστήματος του Linux για ασύγχρονη Είσοδο / Έξοδο. Βασιζόμαστε στον αποστολέα να μη γράψει στους buffers αποστολής, όσο το μήνυμα μεταδίδεται. Αν τελικά γράψει, πριν την ολοκλήρωση της αποστολής, θα αλλοιώσει τα δεδομένα που διαβάζει ο παραλήπτης. Για την ασφαλή επαναχρησιμο-

ποίηση των buffers αποστολής υπάρχουν δύο πιθανές προσεγγίσεις: Βασιζόμαστε στις εφαρμογές να υλοποιήσουν κάποιον μηχανισμό, με τον οποίο ο παραλήπτης θα ειδοποιεί τον αποστολέα ότι τελείωσε με τους αντίστοιχους buffers (όπως στην περίπτωση του benchmark vmsplice-coop). Εναλλακτικά, υλοποιούμε κάποιον τέτοιο μηχανισμό στο module μας, για να εξασφαλίσουμε την ασφαλή επαναχρησιμοποίηση των buffers. Με τον τρόπο αυτό αποφεύγουμε το κόστος του page-remapping, τουλάχιστον στην πλευρά του αποστολέα. Επίσης, αποφεύγουμε την εκκαθάριση του TLB (TLB flush), ενώ η επαναχρησιμοποίηση των buffers αποστολής θα έχει ως αποτέλεσμα την καλύτερη χρησιμοποίηση της κρυφής μνήμης του επεξεργαστή.

Η προσέγγιση αυτή έχει την προοπτική να εξαλείψει το κόστος του page-remapping στην πλευρά του αποστολέα. Στην πλευρά του παραλήπτη, και δεδομένου ότι επιθυμούμε να αποφύγουμε τα αντίγραφα, κάθε λήψη μηνύματος εξακολουθεί να απαιτεί την απεικόνιση των εισερχόμενων σελίδων στο χώρο διευθύνσεων της αντίστοιχης διεργασίας. Όταν μια διεργασία επιθυμεί να στείλει ένα μεγάλο όγκο δεδομένων σε κάποια άλλη, χρησιμοποιεί συνήθως ένα μεγάλο αριθμό μηνυμάτων. Επιπρόσθετα, στις περισσότερες περιπτώσεις, δύο διεργασίες που επικοινώνησαν μια φορά θα επικοινωνήσουν ξανά κάποια στιγμή στο άμεσο μέλλον. Θα μπορούσαμε να εκμεταλλευτούμε τις παρατηρήσεις αυτές εισάγοντας την έννοια ενός buffer "τοπικού" σε ένα κανάλι επικοινωνίας. Δηλαδή, ένας buffer θα χρησιμοποιείται για ανταλλαγή δεδομένων αποκλειστικά ανάμεσα σε δύο συγκεκριμένες διεργασίες. Αν κάποια από τις δύο θελήσει να επικοινωνήσει με μία τρίτη διεργασία θα πρέπει να δεσμεύσει έναν καινούριο buffer, "τοπικό" στο νέο κανάλι επικοινωνίας. Ο ίδιος buffer χρησιμοποιείται για την ανταλλαγή πολλαπλών μηνυμάτων ανάμεσα στις δύο αυτές διεργασίες. Όταν ο αποστολέας επιθυμεί να στείλει ένα μήνυμα στον παραλήπτη θα ζητάει έναν buffer από το μηχανισμό μας. Αν υπάρχει διαθέσιμος buffer, που είναι ήδη απεικονισμένος στο χώρο διευθύνσεων των δύο διεργασιών, θα επιστρέφεται η διεύθυνση αυτού. Αλλιώς θα δεσμεύεται ένας νέος buffer, θα απεικονίζεται στο χώρο διευθύνσεων των δύο διεργασιών και θα προσαρτάται στο σύνολο των buffers (buffer pool) αυτού του καναλιού επικοινωνίας. Όταν ο παραλήπτης τελειώσει με τα δεδομένα ενός buffer ειδοποιεί τον μηχανισμό μας. Αντί να καταργούμε τις απεικονίσεις για τις σελίδες του buffer από το χώρο διευθύνσεων του παραλήπτη, τον προσθέτουμε στο σύνολο των διαθέσιμων buffer του καναλιού επικοινωνίας. Αυτό επιτρέπει την επαναχρησιμοποίηση του για την αποστολή μελλοντικών μηνυμάτων. Η παραπάνω προσέγγιση θα μας επιτρέψει να αποφύγουμε το μεγαλύτερο

μέρος της επιβάρυνσης του page-remapping στη πλευρά του παραλήπτη, πετυχαίνοντας ταυτόχρονα το στόχο μας για επικοινωνία μηδενικών αντιγράφων.

Βιβλιογραφία

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania, *The multikernel: A new os architecture for scalable multicore systems*, Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (New York, NY, USA), SOSP '09, ACM, 2009, pp. 29–44.

[BC05] Daniel Bovet and Marco Cesati, *Understanding the linux kernel*, Oreilly & Associates Inc, 2005.

[Cor05] Jonathan Corbet, *Circular pipes*, 2005, `http://lwn.net/Articles/118750/` [Online; accessed 23-September-2016].

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux device drivers, 3rd edition*, O'Reilly Media, Inc., 2005.

[DP93] Peter Druschel and Larry L. Peterson, *Fbufs: A high-bandwidth cross-domain transfer facility*, SIGOPS Oper. Syst. Rev. **27** (1993), no. 5, 189–202.

[Gor04] Mel Gorman, *Understanding the linux virtual memory manager*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[Int16] Intel, *Intel core i7-980x extreme edition specifications*, 2016, `http://ark.intel.com/products/47932/`

Intel-Core-i7-980X-Processor-Extreme-Edition-12M-Cache-3_33-GHz-6_40-GTs-Intel-QPI/ [Online; accessed 5-October-2016].

[Kle04] Andi Kleen, *Virtual memory map with 4 level page tables*, 2004, http://lxr.free-electrons.com/source/Documentation/x86/x86_64/mm.txt [Online; accessed 20-September-2016].

[KT95] Yousef A. Khalidi and Moti N. Thadani, *An efficient zero-copy i/o framework for unix*, Tech. report, Mountain View, CA, USA, 1995.

[mpi16] *Mpi forum*, 2016, http://mpi-forum.org/ [Online; accessed 16-September-2016].

[oCCL12] University of Cambridge Computer Laboratory, *ipc-bench*, 2012, https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/ [Online; accessed 26-September-2016].

[OSD16] OSDEV.org, *Ipc data copying methods*, 2016, http://wiki.osdev.org/IPC_Data_Copying_methods [Online; accessed 22-September-2016].

[RBV04] John A. Ronciak, Jesse Brandeburg, and Ganesh Venkatesan, *Page-flip technology for use within the linux networking stack*, Proceedings of the Linux Symposium, vol. 2, 2004, pp. 175–180.

[SMS+12] Steven Smith, Anil Madhavapeddy, Christopher Smowton, Malte Schwarzkopf, Richard Mortier, Steven H, and Robert M. Watson, *Draft: Have you checked your ipc performance lately?*, 2012.

[SSCZ12] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang, *Revisiting software zero-copy for web-caching applications with twin memory allocation*, Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12) (Boston, MA), USENIX, 2012, pp. 355–360.

[WGB+10] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal, *An operating system for multicore and clouds: Mechanisms and implementation*, Proceedings of the 1st ACM Symposium on Cloud Computing (New York, NY, USA), SoCC '10, ACM, 2010, pp. 3–14.