ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΉΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και υλοποίηση μηχανισμών fork και pipe σε unikernels

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΜΑΙΝΑ ΧΑΡΑΛΑΜΠΟΥ

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2019

ΕΘΝΙΚΌ ΜΕΤΣΌΒΙΟ ΠΟΛΥΤΕΧΝΕΊΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και υλοποίηση μηχανισμών fork και pipe σε unikernels

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΜΑΙΝΑ ΧΑΡΑΛΑΜΠΟΥ

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20η Μαρτίου.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2019

.....
(Μάινας Χαράλαμπος)
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα τελευταία χρόνια το cloud computing αποτελεί ένα σημαντικό κεφάλαιο στη σύγχρονη επιστήμη υπολογιστών. Η κύρια τεχνολογία που χρησιμοποιείται, προκειμένου να μπορεί να υποστηριχθεί το cloud computing είναι αυτή της εικονικοποίησης. Με αυτό τον τρόπο ένα φυσικό μηχάνημα, μπορεί να φιλοξενήσει πολλά εικονικά μηχανήματα, κάθε ένα από τα οποία αποτελεί έναν αυτοδύναμο υπολογιστή. Ωστόσο, αποτελεί συχνό φαινόμενο οι εικονικές αυτές μηχανές να χρησιμοποιούνται για την εκτέλεση μίας και μόνο εφαρμογής. Αυτό έχει ως αποτέλεσμα, να χαραμείζονται πόροι σε ενέργειες που δε χρειάζονται από την εφαρμογή, αλλά είναι απαραίτητες για το λειτουργικό σύστημα στο οποίο εκτελούνται αυτές οι εφαρμογές.

Μία νεότερη τάση για την υποστήριξη του cloud computing είναι τα containers, που προσφέρουν ελαφρύτερη εικονικοποίηση με γρηγορους χρόνους εκτέλεσης, μικρή κατανάλωση μνήμης και άλλα πλεονεκτήματα. Από την άλλη, παρουσιάζουν αρκετά σημαντικά ζητήματα που αφορούν την ασφάλεια. Ένα από τα ζητήματα αυτά είναι, εκείνο της απομόνωσης το οποίο αναγκάζει σε αρκετές περιπτώσεις να οδηγεί στη χρήση εικονικών μηχανών για τη φιλοξενία των containers, χάνοντας αρκετά από τα πλεονεκτήματα τους.

Μία ακόμη προσέγγιση στο θέμα είναι τα unikernels. Πρόκειται για μία εικόνα μηχανής, με ένα μόνο address space το οποίο κατασκευάζεται από library operating systems και είναι ειδικευμένο για μία συγκεκριμένη εφαρμογή. Πιο απλά, περιέχει τον κώδικα της εφαρμογής και ακριβώς ό,τι κομμάτι του λειτουργικού συστήματος χρειάζεται η εφαρμογή για να λειτουργήσει η διεργασία (drivers, βιβλιοθήκες, κ.λ.π.), ενοποιημένα σαν ένα αυτόνομο πρόγραμμα που μπορεί να τρέξει ως εικονική μηχανή, ή ακόμα και bare metal. Τα unikernels καταφέρνουν να έχουν γρήγορους χρόνους εκκίνησης και μικρή κατανάλωση μνήμης, χωρίς να θυσιάζεται η ασφάλεια Εν τούτοις, ένα πρόβλημα είναι ότι τα unikernels υποστηρίζουν μία και μόνο διεργασία, με αποτέλεσμα να μην μπορούν εφαρμογές με παραπάνω από μία διεργασίες να εκτελεστούν σε unikernels.

Σκοπός, λοιπόν, αυτής της εργασίας είναι η υλοποίηση ενός μηχανισμού που θα επιτρέπει σε εφαρμογές με περισσότερες από μία διεργασίες να μπορούν να εκτελεστούν και σε unikernels. Επιπλέον, υλοποιείται και ένας απλός μηχανισμός για επικοινωνία μεταξύ των εικονικών μηχανών, στα πρότυπα του pipe.

Λέξεις-Κλειδιά: εικονικοποίηση, εικονικές μηχανές, ενδοεπικοινωνία εικονικών μηχανών, unikernel, kvm, QEMU, rumprun, λειτουργικά συστήματα

Abstract

In recent years cloud computing is one important chapter of modern commputer science. The main technology used in order to support cloud computing is virtualization. Virtualization allows a physical machine to host many virtual machines, each one of which is a self-sufficient computer. However, virtual machines are often used to execute a single application. As a result, resources are devoted to actions that are not needed by the application, but they are necessary for the operating system in which the applications run.

An another technology to support cloud computing is containers which offer lightweight virtualization, fast instantiation times and small per-instance memory footprints among other features. On the other hand, containers have several important security issues. Isolation is one of these issues , which in several cases leads to the use of virtual machines to host the containers, losing several of their advantages.

A further approach in cloud computing is unikernerls. Unikernels are specialised, single-address-space machine images constructed by using library operating systems and are specialised for one application. In somewhat simplified terms, unikernels consist of the application's source code and the parts of an opperating system that are necessary for the proccess to run (drivers, libraries, etc.) consolidated as a stand-alone virtual machine, or even an app that can be executed bare metal. The Unikernels manage to have fast instantiation times, small memory footprints, without sacrificing security. However, one of the problems of unikernels is that they are single-proccess and as a result multi-proccess applcations are not able to run on unikernels.

The purpose of this thesis is to implement a mechanism that will enable the execution of multi-proccess applications on unikernels. Furthermore, a pipe-like mechanism for inter-vm communication is impemented.

Keywords: virtualization, virtual machines, inter-vm communication, unikernel, kvm, QEMU, rumprun, operating systems

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ.Γεώργιο Γκούμα για την ευκαιρία που μου έδωσε να εκπονήσω την παρούσα εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων του Ε.Μ.Π.

Ιδιαίτερες ευχαριστίες οφείλω στον Ερευνητή Αναστάσιο Νάνο, για τη συνεχή του καθοδήγηση και συνεργασία σε όλη τη διάρκεια της διπλωματικής αυτής εργασίας, καθώς και για την υποστήριξή και πίστη που μου έδειχνε. Θα ήθελα επίσης να ευχαριστήσω τους Στράτο Ψωμαδάκη, Κωστή Παπαζαφειρόπουλο και Στέφανο Γεράγγελο για τη βοήθεια που μου παρείχαν.

Χαράλαμπος Μάινας

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

Στις μέρες μας το cloud computing έχει γίνει ένα από τα ταχύτερα αναπτυσσόμενα και ενδιαφέροντα θέματα στην επιστήμη των υπολογιστών. Το cloud computing δίνει τη δυνατότητα σε απομακρυσμένους χρήστες να αποκτάνε πρόσβαση σε υπολογιστικούς πόρους (αποθηκευτικός χώρος, εφαρμογές, υπηρεσίες κ.λ.π.) όταν χρειαστούν από τους χρήστες. Ιδιαίτερα τα τελευταία χρόνια το cloud έχει μπει και στις ζωές των απλών χρηστών. Η χρήση των προσωπικών υπολογιστών αρχίζει να αλλάζει σημαντικά, καθώς πλέον προγράμματα και δεδομένα απομακρύνονται από τους προσωπικούς υπολογιστές για να εκτελεστούν και να αποθηκευτούν στο λεγόμενο cloud.

Μία από τις βασικές τεχνολογίες που κρύβονται πίσω από το cloud είναι αυτή της εικονικοποίησης. Χάρις την εικονικοποίηση, μπορούμε σε ένα φυσικό μηχάνημα να φιλοξενήσουμε πολλά εικονικά μηχανήματα κάθε ένα από τα οποία είναι ανεξάρτητο. Με αυτό τον τρόπο, μπορούμε να αξιοποιήσουμε καλύτερα τους φυσικούς πόρους του μηχανήματος και να τους διαμοιράσουμε όπως επιθυμούμε μεταξύ των εικονικών μηχανών. Ακόμα, η εικονικοποίηση δημιούργησε και κάποιες νέες δυνατότητες, όπως αυτή της μεταφοράς εικονικών μηχανών σε διαφορετικό φυσικό μηχάνημα, η δημιουργία αντιγράφων εικονικών μηχανών, αλλά και περισσότερη ασφάλεια, αφού ένα πρόβλημα σε ένα εικονικό μηχάνημα δε θα επηρεάσει ούτε το φυσικό ούτε τα υπόλοιπα εικονικά μηχανήματα.

Ένα συχνό φαινόμενο κατά τη χρήση της εικονικοποίησης, είναι η χρήση μία εικονικής μηχανής με συμβατικά λειτουργικά συστήματα για την υποστήριξη μίας και μόνο υπηρεσίας. Όπως είναι φυσικό το λειτουργικό σύστημα μέσα στην εικονική μηχανή χρειάζεται κάποιους πόρους για να λειτουργήσει, ενώ συχνά μπορεί να εκτελεί λειτουργίες οι οποίες δε χρειάζονται από την εφαρμογή. Ακόμα, ο κώδικας όλου του λειτουργικού συστήματος αυξάνει αρκετά και το μέγεθος σε μνήμη της εικονικής μηχανής. Γίνεται λοιπόν, κατανοητό ότι σπαταλούνται πόροι, οι οποίοι θα μπορούσα να διατεθούν είτε στην ίδια την υπηρεσία είτε σε κάποια άλλη.

Για την επίλυση του παραπάνω προβλήμαατος αναζητήθηκαν λύσεις για να γίνει η εικονικοποίηση πιο ελαφρυά, αλλά και να μη σπαταλούνται πόροι σε αχρείαστες λειτουργίες. Μία από αυτές τις λύσεις, ήταν η εικονικοποίηση σε επίπεδο λειτουργικού συστήματος ή διαφορετικά containerization. Με τη συγκεκριμένη μέθοδο ο πυρήνας επιτρέπει την ύπαρξη πολλαπλών απομονωμένων user-space instances, που ονομάζονται containers. Τα containers μοιράζονται μεταξύ τους το λειτουργικό σύστημα στα οποία εκτελούνται, ενώ ταυτόχρονα παρέχουν ένα απομονωμένο περιβάλλον για τις διεργασίες μέσα σε αυτό.

Η τεχνολογία των containers, όπως κάθε άλλη τεχνολογία δε θα μπορούσε να μην έχει και κάποια μειονεκτήματα. Ένα από τα κύρια μειονεκτήματα, είναι αυτό της ασφάλειας. Τα containers μοιράζονταai τον ίδιο πυρήνα του host, ενώ οι μηχανισμοί απομόνωσης δεν είναι το ίδιο ισχυροί με αυτούς στα εικονικά μηχανήματα. Μάλιστα, έχουν αναφερθεί περιπτώσεις όπου από ένα container μπορούσαν να παρθούν πληροφορίες και δεδομένα τόσο για το host, όσο και για άλλα containers [9]. Όλα αυτά έχουν οδηγήσει σε περιπτώσεις όπου τα containers χρησιμοποιούνται πάνω από ένα πλήρη λειτουργικό σύστημα το οποίο τρέχει μέσα σε μία εικονική μηχανή.

Μία ιδέα που αρχίζει να αποκτά όλο και περισσότερο ενδιαφέρον και προσοχή είναι αυτή των unikernels. Η βάση της ιδέας, είναι η κατασκευή ειδικευμένων εικονικών μηχανών για κάθε ξεχωριστή υπηρεσία. Στην εικονική αυτή μηχανή δε χρειάζεται να υπάρχει ένα πλήρη λειτουργικό σύστημα, αλλά μόνο τα κομμάτια αυτού τα οποία είναι απαραίτητα για την εκτέλεση της υπηρεσίας. Με αυτό τον τρόπο, μειώνεται σημαντικά το μέγεθος των εικονικών μηχανών, ενώ πλέον οι πόροι χρησιμοποιούνται ακριβώς για τις λειτουργίες που χρειάζεται η υπηρεσία. Τέλος, εφόσον αναφερόμαστε σε εικονικές μηχανές, η απομόνωση τους είναι κάτι το οποίο προσφέρουν ήδη οι ελεγκτές.

1.1 Σκοπός

Η φιλοσοφία των unikernels είναι ότι κάθε εικονική μηχανή θα έχει ένα συγκεκριμένο σκοπό. Για το λόγο αυτό τα unikernels δεν υποστηρίζουν παραπάνω από μία διεργασία σε κάθε εικονική μηχανή. Από την άλλη, υπάρχουν unikernel frameworks τα οποία επιτρέπουν την υποστήριξη περισσότερων από ένα νήμα. Ωστόσο οι περισσότερες εφαρμογές και υπηρεσίες στο cloud είναι φτιαγμένες για ένα πλήρη λειτουργικό σύστημα. Επομένως, προκειμένου να μπορούν να τρέξουν σε ένα unikernel θα πρέπει να αλλάξει άλλοτε περισσότερο και άλλοτε λιγότερο ο σχεδιασμός τους. Ιδιαίτερα, οι εφαρμογές που χρησιμοποιούν παραπάνω από μία διεργασίες θα πρέπει να αλλάξουν

σε ένα μοντέλο με μία μόνο διεργασία.

Ο σκοπός της συγκεκριμένης εργασίας είναι να υλοποιήσει, κρατώντας το single-proccess χαρακτηριστικό των unikernels, ένα μηχανισμό για την υποστήριξη των εφαρμογών που απαιτούν παραπάνω από μία διεργασία.

Αρχικά έγινε μία μελέτη γύρω από τα υπάρχοντα unikernel frameworks, παρατηρώντας τα χαρακτηριστικά του κάθε ενός. Στη συνέχεια σε ένα από αυτά τα unikernels (rumprun) υλοποιήθηκαν οι δύο παρακάτω λειτουργίες:

- ένας μηχανισμός για επικοινωνία μεταξύ των εικονικών μηχανών, ο οποίος είναι στα πρότυπα της κλήσης συστήματος pipe (POSIX). Ουσιαστικά πρόκειται για την υλοποίηση της pipe σε επίπεδο εικονικών μηχανών.
- ένας μηχανισμός για την υποστήριξη της κλήσης συστήματος fork από τα unikernels. Όταν μία εφαρμογή θα χρησιμοποιεί τη συγκεκριμένη κλήση συστήματος, θα δημιουργείται μία νέα εικονική μηχανή-κλώνο της αρχικής και θα ξεκινά την εκτέλεση της, ακριβώς μετά την κλήση συστήματος fork.

1.2 Οργάνωση Κειμένου

Στη συνέχεια παρουσιάζεται αναλυτικά η περιγραφή του σχεδιασμού και της υλοποίησης των δύο λειτουργιών pipe, fork, γίνεται μία ανασκόπηση στα ήδη υπάρχοντα unikernel frameworks και περιγράφεται το απαραίτητο θεωρητικό υπόβαθρο.

Πιο συγκεκριμένα, στο Κεφάλαιο 2 καλύπτεται το απαραίτητο θεωρητικό υπόβαθρο, κάνοντας μία μικρή αναφορά για το cloud computing, μία εισαγωγή στην εικονικοποίηση και στα λειτουργικά συστήματα.

Στο Κεφάλαιο 3 παρουσιάζονται τα unikernel frameworks που μελετήθηκαν, τα χαρακτηριστικά τους, η φιλοσοφία τους κ.λ.π.

Στο Κεφάλαιο 4 γίνεται αναλυτική περιγραφή του σχεδιασμού και της υλοποίησης των δύο λειτουργιών που αναφέρθηκαν προηγουμένως στο σκοπό της εργασίας (pipe,fork).

Τέλος, στο Κεφάλαιο 5 αναφέρονται τα τελικά συμπεράσματα της παρούσας μελέτης όπως επίσης και πιθανές μελλοντικές επεκτάσεις αυτής της διπλωματικής εργασίας.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Cloud computing

Εδώ και αρκετό καιρό γίνεται συχνά αναφορά στο ”Cloud”, αλλά και σε αυτή την εργασία έχει αναφερθεί αρκετές φορές ο όρος ”cloud computing”. Παρά τη δημοφιλία του όρου, δεν υπάρχει ακριβής ορισμός. Εντούτοις, θα χρησιμοποιηθεί ο ορισμός που έχει δοθεί από το NIST (National Institute of Standards and Technology of USA) [20]. Το cloud computing (υπολογιστικό νέφος με ελληνικούς όρους) είναι η κατ’ αίτηση διαδικτυακή κεντρική διάθεση υπολογιστικών πόρων (όπως δίκτυο, εξυπηρετητές, εφαρμογές και υπηρεσίες) με υψηλή ευελιξία, ελάχιστη προσπάθεια από τον χρήστη και υψηλή αυτοματοποίηση. Ωστόσο, είναι συχνό φαινόμενο να αναφερόμαστε με το συγκεκριμένο όρο τόσο στις υπηρεσίες που είναι διαθέσιμες μέσω διαδικτύου με τη μορφή μίας υπηρεσίας ιστού όσο και στο υλικό και λογισμικό που απαρτίζουν την υποδομή που προσφέρει αυτές τις υπηρεσίες.

2.1.1 Χαρακτηριστικά του cloud computing

Πέρα από τον ορισμό το NIST [20], περιγράφει και τα ακόλουθα βασικά χαρακτηριστικά του cloud computing:

- παροχή υπηρεσίας κατ’ απαίτηση: Οι χρήστες έχουν τη δυνατότητα να τροποποιήσουν τους πόρους που προσφέρονται, χωρίς την ανθρώπινη διαμεσολάβηση από την πλευρά του παρόχου του cloud.
- ευρυζωνική δικτυακή πρόσβαση: Οι υπηρεσίες είναι διαθέσιμες από το διαδίκτυο.
- Διάθεση πόρων σε περισσότερους χρήστες: Οι πόροι μπορούν

να διατεθούν ανάμεσα σε πολλούς χρήστες χωρίς να δημιουργούνται προβλήματα.

- ταχεία ελαστικότητα/επεκτασιμότητα: Οι πόροι μπορούν να ανακατεμηθούν, είτε αυτόματα είτε κατ' απαίτηση, χωρίς περιορισμούς και χωρίς να επηρεάζεται η ομαλή λειτουργία άλλων υπηρεσιών.
- μετρήσιμη υπηρεσία: Η χρήση των πόρων καταγράφεται και παρουσιάζεται τόσο στον πάροχο όσο και στον πελάτη.

2.1.2 Μοντέλα υπηρεσιών

Σύμφωνα με το NIST [20], τα μοντέλα υπηρεσιών του cloud είναι τα εξής:

Λογισμικό ως Υπηρεσία (SaaS)

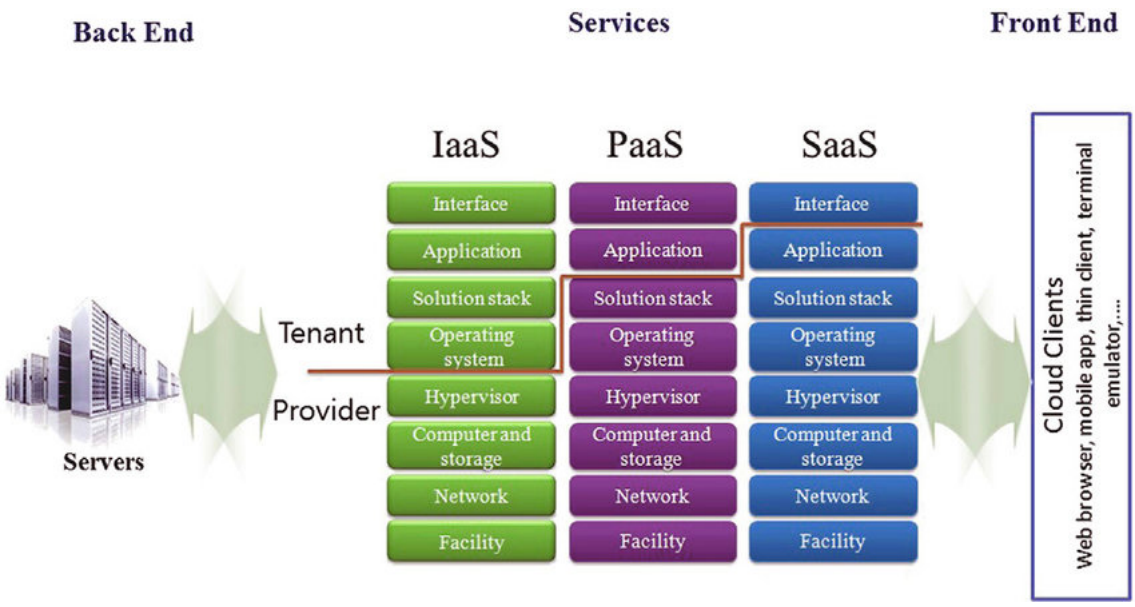
Η υπηρεσία που παρέχεται στο χρήστη είναι αυτή μία εφαρμογής που τρέχει στην υποδομή του παρόχου. Ο χρήστης δε χρειάζεται να γνωρίζει τίποτα σχετικά με την υποδομή που υποστηρίζει την εφαρμογή, ούτε έχει τη δυνατότητα να τροποποιήσει πολλά πράγματα πέρα ίσως από κάποιες ρυθμίσεις που προσφέρει η εφαρμογή. Οι εφαρμογές αυτές είναι προσβάσιμες μέσα από κάποιο λογιμικό πελάτη που προσφέρει ο πάροχος ή ακόμα και από ένα περιηγητή ιστού (web browser).

Πλατφόρμα ως υπηρεσία (PaaS)

Σε αυτή την περίπτωση ο χρήστης έχει τη δυνατότητα της δημιουργίας εφαρμογών ή/και περιβάλλοντων εφαρμογών, χρησιμοποιώντας προγραμματιστικά εργαλεία και υπηρεσίες που παρέχονται από τον πάροχο. Ο χρήστης δεν μπορεί να ελέγξει και να διαχειριστεί την υποδομή του cloud, ωστόσο έχει τον έλεγχο των αναπτυγμένων εφαρμογών και ενδεχομένως των ρυθμίσεων διαμόρφωσης για το περιβάλλον φιλοξενίας αυτών των εφαρμογών.

Υποδομή ως υπηρεσία (IaaS)

Οι παροχές προς το χρήστη είναι υπολογιστικοί πόροι όπως επεξεργαστές, αποθηκευτικός χώρος και δίκτυο στα οποία μπορεί αναπτύξει και να τρέξει λειτουργικά συστήματα και εφαρμογές. Παρά τη δυνατότητα να τροποποιήσει την ποσότητα των πόρων, ο χρήστης δεν έχει παραπάνω έλεγχο ως προς την υποδομή του cloud.



Σχήμα 2.1: Cloud services

Όπως βλέπουμε και από το σχήμα 2.1, όσο περισσότερη ευελιξία προσφέρει ένα μοντέλα τόσο περισσότερο έλεγχο έχει και ο χρήστης στις υπηρεσίες που λαμβάνει. Για παράδειγμα στην περίπτωση SaaS, ο χρήστης έχει το λιγότερο έλεγχο, σε αντίθεση με την περίπτωση IaaS όπου ο χρήστης μπορεί πλέον να αποκτήσει έλεγχο ακόμα και στο λειτυργικό σύστημα.

2.2 Εικονικοποίηση

Στην επιστήμη των υπολογιστών ο όρος εικονικοποίηση (virtualization) περιγράφει ένα μηχανισμό αφαίρεσης, που με τη χρήση ”εικονικών υπολογιστικών πόρων”, έχει ως σκοπό την απόκρυψη λεπτομερειών για την υλοποίηση ή την κατάσταση των φυσικών πόρων. Με το μηχανισμό αυτό ένας φυσικός πόρος μπορεί να παρουσιαστεί ως μία πλειάδα εικονικών φυσικών πόρων, ή αντίστροφα μία πλειάδα φυσικών πόρων να παρουσιαστούν ως ένας ενιαίος εικονικός πόρος. Ανάλογα σε ποιο επίπεδο στη στοίβα του λογισμικού υλοποιείται η εικονικοποίηση, υπάρχουν και τα αντίστοιχα πλεονεκτήματα, αλλά σε γενικές γραμμές κάποια από τα πλεονεκτήματα της εικονικοποίησης είναι:

- Καλύτερη αξιοποίηση και διαμοιρασμός των πόρων.
- Ασφάλεια μεσω της απομόνωσης των υπηρεσιών.

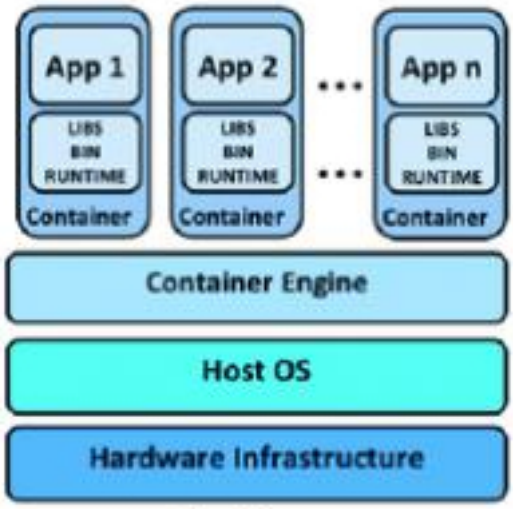
- Δυνατότητα προσομοίωσης περιβαλλόντων που δεν είναι φυσικά διαθέσιμα.
- Δυνατότητα αποθήκευσης, μεταφοράς και επαναφοράς της κατάστασης των υπηρεσιών που προσφέρονται μέσω της εικονικοποίησης.

Η εικονικοποίηση μπορεί να υλοποιηθεί σε διάφορα επίπεδα, όπως στο δίκτυο, στον αποθηκευτικό χώρο, στο hardware, στο λειτουργικό σύστημα κ.α. Η συγκεκριμένη εργασία ασχολείται κυρίως με την εικονικοποίηση σε επίπεδο υλικού, ωστόσο είναι συχνό φαινόμενο να συγκρίνονται τα unikernels με τα containers. Για το λόγο αυτό θα γίνει μία σύντομη παρουσίαση της εικονικοποίησης σε επίπεδο λειτουργικού συστήματος, ενώ μετά από αυτή την περιγραφή, όταν χρησιμοποιείται ο όρος εικονικοποίηση θα εννοείται εικονικοποίηση σε επίπεδο υλικού.

2.2.1 Εικονικοποίηση σε επίπεδο λειτουργικού συστήματος

Η εικονικοποίηση σε επίπεδο λειτουργικού συστήματος είναι νεότερη σε σχέση με αυτή σε επίπεδο υλικού, ωστόσο έχει καταφέρει να συγκεντρώσει αρκετή προσοχή τα τελευταία χρόνια. Ο πυρήνας του λειτουργικού συστήματος επιτρέπει την ύπαρξη παραπάνω από ένα, απομονωμέα userspace instances τα οποία ονομάζονται containers, virtualization engines ή jails. Όπως φαίνεται στο σχήμα 2.2 όλα τα containers μοιράζονται τον ίδιο πυρήνα πάνω από τον οποίο βρίσκεται το στρώμα που είναι υπεύθυνο για την εικονικοποίηση.

Το γεγονός ότι τα containers μοιράζονται τον ίδιο πυρήνα κάνει το μέγεθο τους αρκετά μικρό, αφού δε χρειάζεται να συμπεριλάβουν κάποιο λειτουργικό σύστημα. Επιπλέον, οι χρόνοι εκκίνησης τους είναι αρκετά μικροί, χωρίς να χρειάζεται να καταναλωθεί πολλή μνήμη για τη λειτουργία των containers. Επιπροσθέτως, τα containers είναι ανεξάρτητα από το φυσικό μηχάνημα, καθώς η εικονικοποίηση γίνεται με τη χρήση του κατάλληλου λογισμικού χωρίς να χρειάζεται κάποια υποστήριξη από το hardware. Τέλος, τα containers επιτρέπουν το εύκολο πακετάρισμα τόσο των εφαρμογών όσο και των κατάλληλων ρυθμίσεων, που σε συνδυασμό με την ανεξαρτησία από το hardware, δίνουν τη δυνατότητα για την εύκολη μεταφορά τους σε διαφορετικά μηχανήματα.



Σχήμα 2.2: Εικονικοποίηση σε επίπεδο λειτουργικού συστήματος

Δυστυχώς, όλα αυτά τα πλεονεκτήματα έρχονται με κάποιο κόστος, με το μεγαλύτερο να είναι αυτό της ασφάλειας. Αν και οι υποστηρικτές των containers θεωρούν ότι με τις κατάλληλες ρυθμίσεις τα containers μπορούν να γίνουν ασφαλή, δεν προσφέρουν την ίδια ασφάλεια με την εικονικοποίηση σε επίπεδο υλικού [10]. Τα containers δεν έχουν τη δυνατότητα να προσφέρουν τόσο ισχυρή απομόνωση όσο προσφέρει η εικονικοποίηση σε επίπεδο υλικού. Τα αποτελέσματα από την παραβίαση της απομόνωσης μπορεί να είναι από την απόκτηση πληροφοριών σχετικά με το host μηχάνημα ή άλλων containers, μέχρι την απόκτηση ελέγχου του λειτουργικού συστήματος στον host [9].

Η σύγκριση τψν δύο ειδών εικονικοποίησης είναι ένα ανοιχτό ζήτημα το οποίο δύσκολα θα λυθεί σύντομα [8]. Άλλωστε και τα δύο είδη, αναπτύσσονται ταχύτατα, προσφέροντας συνεχώς νέες δυνατότητες και μειώνοντας τα προβλήματα που υπάρχουν. Σε μία προσπάθεια να ενοποιηθούν τα θετικα των δύο ειδών, είναι συχνή η χρήση εικονικών μηχανών μέσα στις οποίες τρέχουν containers. Με αυτό τον τρόπο βελτιώνεται η ασφάλεια σε βάρος όμως της απόδοσης, καθώς πλέον χρειάζονται πόροι και για την εκτέλεση του απαραίτητου λογισμικού που θα επιτρέψει την υπάρξη εικονικών μηχανών.

2.2.2 Εικονικοποίηση σε επίπεδο υλικού

Η εικονικοποίηση σε επίπεδο υλικού, ή απλά εικονικοποίηση για το υπόλοιπο της συγκεκριμένης εργασίας, επιτρέπει τη λειτουργία ενός ολόκληρου υπολογιστικού συστήματος μέσα από έναν άλλον.

Τα εμφωλευμένα υπολογιστικά συστήματα αναφέρονται ως guests, ενώ το υπολογιστικό σύστημα στο οποίο πραγματοποιείται η εικονικοποίηση αναφέρεται ως host. Ένας guest εκτελείται μέσα σε μία εικονική μηχανή (virtual machine - VM), την οποία οι Popek και Goldberg [21] ορίζουν ως εξής: "ένα αποδοτικό και απομονωμένο αντίγραφο μιας πραγματικής μηχανής". Κάθε εικονική μηχανή είναι ανεξάρτητη τόσο από το host, όσο και άπο άλλες τυχόν εικονικές μηχανές που μπορεί να υπάρχουν. Με αυτού του είδους την εικονικοποίηση, κάθε εικονική μηχανή έχει την ψαυδαίσθηση ότι κατέχει αποκλειστικά τους πόρους που της έχουν διατεθεί.

Απαραίτητο συστατικό για να επιτευχθούν όλα τα παραπάνω είναι ο επόπτης (hypervisor). Ο επόπτης, σύμφωνα με τους Popek και Goldberg έχει τρία συγκεκριμένα χαρακτηριστικά [21]:

1. ο επόπτης παρέχει ένα περιβάλλον όμοιο με το πραγματικό υπολογιστικό σύστημα,

2. τα προγράμματα που τρέχουν στο περιβάλλον που δημιουργείται θα πρέπει να έχουν στη χειρότερη περίπτωση μικρές μειώσεις στην ταχύτητα εκτέλεσης,

3. ο επόπτης έχει πλήρη έλεγχο των πόρων.

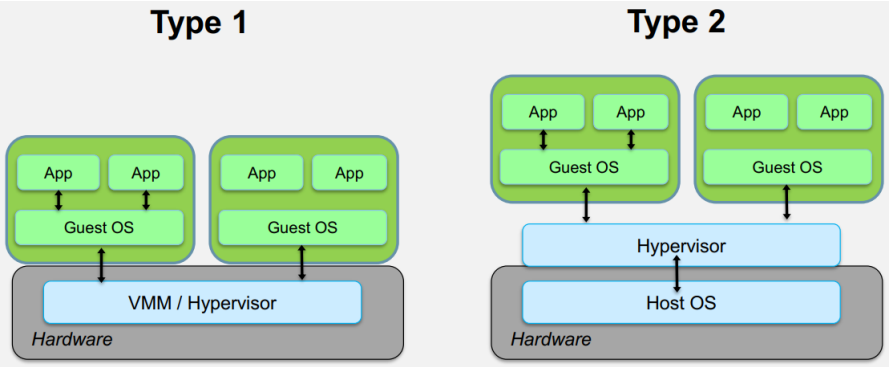
Ο ρόλος του επόπτη μοιάζει με το ρόλο του λειτουργικού συστήματος, με τις εικονικές μηχανές να έχουν το ρόλο των διεργασιών. Ο επόπτης χρονοδρομολογεί τις εικονικές μηχανές στη CPU, μοιράζει γενικότερα τους διαθέσιμους πόρους στα εικονικά μηχανήματα, εκτελεί εκ μέρους των εικονικών μηχανών "προνομιούχες εντολές" και είναι αυτός που διαχειρίζεται τις εικονικές μηχανές (δημιουργία, εκκίνηση, τερματισμός κ.λ.π).

Όπως βλέπουμε στο σχήμα 2.3 υπάρχουν δύο διαφορετικά είδη εποπτών, ανάλογα με το που βρίσκονται στο σύστημα.

- **Type 1 - Bare metal/native**
  Σε αυτή την περίπτωση ο επόπτης εκτελείται πάνω από το υλικό του συστήματος, χωρίς να παρεμβάλλεται κάποιο λειτουργικό σύστημα. Είναι υπεύθυνος για τη χρονοδρομολόγηση των εικονικών μηχανών, τη διαχείριση της μνήμης και των πόρων καθώς και άλλων λειτουργιών. Χρησιμοποιεί δικούς του drivers για τη διαχείριση του υλικού και για την εξυπηρέτηση των I/O λειτουργιών των guest.

- **Type 2 - Hosted**
  Σε αυτή την περίπτωση ο επόπτης εκτελείται πάνω από ένα λειτουργικό σύστημα, σαν ένα συνηθισμένο πρόγραμμα στο user space. Αποτελούν ένα ενδιάμεσο στρώμα μεταξύ του host και του guest, δημιουργώντας το κατάλληλο περιβάλλον για τις εικονικές μηχανές. Δε χρειάζονται ειδικοί drivers για το υλικό,

καθώς αυτοί παρέχονται από το λειτουργικό σύστημα του host. Από την άλλη, οι I/O λειτουργίες και διάφορες άλλες ”προνομιούχες εντολές” του guest θα πρέπει να περάσουν μέσα από τον hypervisor και μετά να εξυπηρετηθούν από τον host, δημιουργώντας επιπλέον κόστος στην εκτέλεση τους.



Σχήμα 2.3: Εικονικοποίηση σε επίπεδο υλικού

Εκτός από το διαχωρισμό των εποπτών, γίνεται και διαχωρισμός και στις τεχνικές που χρησιμοποιούνται για να επιτευχθεί η εικονικοποίηση. Οι τεχνικές αυτές βασίζονται τόσο στη δυνατότητα που προσφέρει ειδικό υλικό για την υποστήριξη της εικονικοποίησης, όσο και στο βαθμό συνεργασίας μεταξύ του επόπτη με την εικονική μηχανή. Θα αναφερθούμε σε τρεις από αυτές τις τεχνικές, την πλήρη εικονικοποίηση (full virtualization), την παραεικονικοποίηση (paravirtualization) και τέλος την εικονικοποίηση υποστηριζόμενη από το υλικό (hardware-assisted virtualization).

Full virtualization

Στην πλήρη εικονικοποίηση ο guest δεν έχει υποστεί καμία αλλαγή και εκτελείται όπως και στην περίπτωση που εκτελείται απευθείας σε ένα φυσικό υπολογιστικό σύστημα. Απο τη μεριά του ελεγκτή, θα πρέπει να γίνει πλήρης προσομοίωση του υλικού ώστε να δίνει την ψευδαίσθηση στον guest ότι επικοινωνεί απευθείας με το υλικό. Ωστόσο δημιουργείται ένα πρόβλημα, το λειτουργικό σύστημα του guest εκτελείται σε unprivileged mode, οπότε δεν μπορεί να εκτελέσει προνομιούχες εντολές. Η λύση σε αυτό το πρόβλημα είναι η τεχνική trap-and-emulate [4], κατά την οποία όταν ο guest προσπαθεί να εκτελέσει μία προνομιούχα εντολή, δημιουργείται μία εξαίρεση (trap) στον hypervisor, καθώς ο guest βρίσκεται σε unprivileged mode. Ύστερα, ο hypervisor παίρνει τον έλεγχο και προσομοιώνει ή εκτελεί την προνομοιούχα εντολή για λογαριασμό του guest.

Η συγκεκριμένη λύση δεν είναι αρκετή για όλες τις αρχιτεκτονικές. Μία από αυτές τις αρχιτεκτονικές είναι και η x86. Στην

x86 αρχιτεκτονική ορισμένες εντολές δεν προκαλούν trap όταν εκτελούνται σε unprivileged mode, παρά το γεγονός ότι για την εκτέλεση τους είναι ανάγκη να βρίσκονται σε privileged mode (non-virtualizable instructions). Μία από αυτές τις εντολές είναι η popf, η οποία ενώ αλλάζει τόσο τα ALU flags όσο και τα flags του επεξεργαστή δεν προκαλεί κάποιο trap με αποτέλεσμα να αγνοούνται οι αλλαγές στα flags του επεξεργαστή. Για την αντιμετώπιση του συγκεκριμένου προβλήματος χρησιμοποείται από τους επόπτες η τεχνική του binary translation [4]. Όταν ο guest βρίσκεται σε unprivileged mode όλες οι εντολές εκτελούνται κανονικά, ενώ όταν μεταβεί στον πυρήνα και σε privileged mode τότε ο hypervisor μεταφράζει τις εντολές που δεν προκαλούν trap σε ένα νέο σύνολο εντολών που θα επιφέρουν τις επιθυμητές αλλαγές στην εικονική μηχανή.

Μπορούμε να παρατηρήσουμε εύκολα ότι και στις δύο περιπτώσεις η απόδοση θα μειωθεί σημαντικά, λόγω της ανάμειξης του hypervisor. Ιδιαίτερα στην περίπτωση του binary translation το κόστος της παρακολούθησης και μετάφρασης των εντολών του guest είναι αρκετά μεγάλο.

Paravirtualization

Στην παραεικονικοποίηση ο guest γνωρίζει ότι δεν εκτελείται απευθείας στο υλικό και ”συνεργάζεται” κατάλληλα με τον hypervisor. Απαιτείται λοιπόν, η τροποποίηση του guest, ώστε να αντικατασταθούν οι non-virtualizable εντολές, με κατάλληλες εντολές (hypercalls) που δίνουν τον έλεγχο στο hypervisor και αυτός με τη σειρά του διεκπαιρεώνει την αντίστοιχη λειτουργία. Επιπλέον ο hypervisor μπορεί να υποστηρίξει και κάποιες άλλες εντολές, που χρησιμοποιούνται για άλλες σημαντικές λειτουργίες του πυρήνα, όπως διαχείρηση μνήμης, διαχείριση διακοπών κ.α.

Με αυτό τον τρόπο μειώνεται σημαντικά το κόστος της εικονικοποίησης, αφού ο guest γνωρίζει ότι εκτελείται σε εικονικό περιβάλλον και μπορεί να βελτιστοποιηθεί για τις συγκεκριμένες συνθήκες. Παράλληλα γίνεται δυνατόν να εικονικοποιηθούν αρχιτεκτονικές όπως η x86 χωρίς σημαντική μείωση της απόδοσης αποφεύγοντας το binary translation. Από την άλλη, ο guest τροποποιείται αρκετά για να μπορέσει να ”συνεργαστεί” με τον επόπτη. Συνεπώς, μειώνεται η δυνατότητα μεταφοράς του σε διαφορετικά συστήματα (π.χ. διαφορετικό επόπτη), ενώ ταυτόχρονα αυξάνει το κόστος συντήρησης του καθώς οποιεσδήποτε αλλαγές ή αναβαθμίσεις στον πυρήνα του guest θα πρέπει να τροποποιηθούν κατάλληλα για την υποστήριξη της παραεικονικοποίησης.

Hardware-assisted virtualization

Η συγκεκριμένη μέθοδος ουσιαστικά έχει τα ίδια χαρακτηριστικά με τη μέθοδο της πλήρης εικονικοποίησης. Ο guest δε χρειάζεται να

υποστεί καμία αλλαγή, ενώ εισάγεται κατάλληλο hardware το οποίο διευκολύνει την εικονικοποίηση. Πρόκειται για επεκτάσεις των επεξεργαστών με στόχο την αύξηση της απόδοσης και της ασφάλειας της εικονικοποίησης. Αυτές οι επεκτάσεις είναι γνωστές ως επεκτάσεις εικονικοποίησης (virtualization extensions).

Ένα από τα βασικά στοιχεία αυτών των επεκτάσεων είναι η δημιουργία ενός νέου επιπέδου εκτέλεσης, αυτό του guest στο οποίο μπορούν να εκτελεστούν τόσο privileged όσο και unprivileged εντολές. Συνεπώς, δε χρειάζεται πλέον να γίνονται trap οι privileged εντολές του guest, αλλά αυτές μπορούν να εκτελεστούν απευθείας, χωρίς μάλιστα να επηρεάζεται ο host από την εκτέλεση τους. Οι επεκτάσεις αφορούν ακόμα τη διαχείριση μνήμης, αλλά και των διακοπών. Πλέον η εκτέλεση του guest συνεχίζεται αδιάκοπα μέχρι την ικανοποίηση κάποιας συνθήκης, όπως για παράδειγμα ένα page fault, όπου τότε ο έλεγχος μεταφέρεται στον host (VM exit).

Με αυτό τον τρόπο επιτυγχάνεται υψηλή απόδοση για την εικονικοποίηση χωρίς να είναι απαραίτητες οι αλλαγές στον guest. Ο πιο σημαντικός παράγοντας απόδοσης για την εν λόγω τεχνική είναι το πλήθος των VM exits, καθώς πρόκειται για μία χρονοβόρα διαδικασία [5]. Μάλιστα πολλές από αυτές τις επεκτάσεις έχουν δημιουργηθεί για ακριβώς αυτό το σκοπό, δηλαδή τη μείωση των VM exits.

2.2.3 QEMU/KVM

Το Qemu [6] είναι ένας επόπτης τύπου 2 και πρόκειται για έναν από τους πιο γνωστούς επόπτες. Με την τεχνική του full virtualization και συγκεκριμένα με τη μέθοδο του dynamic binary translation, υποστηρίζει τη φιλοξενεία πολλών λειτουργικών συστημάτων χωρίς να απαιτείται κάποια αλλαγή σε αυτά. Πέρα από τη λειτουργία του επόπτη, μπορεί να λειτουργήσει και ως προσομοιωτής υλικού. Το Qemu έχει τη δυνατότητα να προσομοιώσει διαφορετικές αρχιτεκτονικές από αυτή που διαθέτει ο host, πληθώρα από κάρτες δικτύου, σκληρούς δίσκους, περιφερειακές συσκευές κ.λ.π.. Με αυτό τον τρόπο μπορεί να χρησιμοποιηθεί και για εφαρμογές που έχουν υλοποιηθεί για διαφορετική αρχιτεκτονική να εκτελεστούν στην αρχιτεκτονική που διαθέτει ο host.

Οι εικονικές μηχανές που εκτελούνται στον ίδιο host, συχνά επικοινωνούν μεταξύ τους. Συνήθως πρόκειται για πλήρη υπολογιστικά συστήματα, συνεπώς για την επικοινωνία μεταξύ τους θα μπορούσε να χρησιμοποιηθεί ό,τι χρησιμοποιειταί μεταξύ φυσικών υπολογιστικών συστημάτων (π.χ. TCP/IP sockets). Μάλιστα ειδικά στη συγκεκριμένη περίπτωση οι επόπτες μπορούν να αντιληφθούν ότι πρόκειται για επικοινωνία μεταξύ εικονικών μηχανών που ελέγχουν οπότε και χρησιμοποιούν διάφορες βελτιστοποιήσεις, προκειμένου να μη χρειαστεί να περάσουν τα πακέτα πέρα από το host. Εντούτοις, συ-

χνά πολλές εικονικές μηχανές δε χρειάζονται δίκτυο και δημιουργείται η ανάγκη για επικοινωνία μεταξύ τους ή ακόμα και με το host. Επιπλεόν πολλές φορές είναι χρήσιμο εικονικές μηχανές να μοιράζονται μνήμη μεταξύ τους. Έτσι έχουν δημιουργηθεί αρκετές μέθοδοι για τον διαμοιρασμό μνήμης και ενδοεπικοινωνίας μεταξύ εικονικών μηχανών, αλλά και μεταξύ του host και των εικονικών μηχανών [24].

Ένας από αυτούς τους μηχανισμούς είναι ο nahanni ή ivshmem (inter-VM shared memory) [17]. Το ivshmem είναι ένας μηχανισμός για το διαμοιρασμό μνήμης του host με τις εικονικές μηχανές που εκτελούνται στο host. Μπορεί να χρησιμοποιηθεί τόσο για διαμοιρασμό μνήμης μεταξύ host και guest, όσο και μεταξύ των guests. Το Qemu επιτρέπει τη χρήση αυτού τού μηχανισμού μέσω μίας PCI συσκευής την οποία πρέπει να υποστηρίξουν οι guests για να χρησιμοποιήσουν τη συγκεκριμένη περιοχή μνήμης.

Πολλές φορές το Qemu χρησιμοποιείται από άλλους επόπτες για την εξομοίωση του υλικού, όπως για παράδειγμα γίνεται από το Xen. Επιπροσθέτως συχνά χρησιμοποιείται σε συνδυασμό με το kvm (στην περίπτωση υποστήριξης hardware-assisted virtualization), πετυχαίνοντας υψηλή απόδοση. Το kvm είναι ένα kernel module του Linux, που μετατρέπει τον πυρήνα σε εναν επόπτη.Ωστόσο, από μόνο του το kvm δε λειτουργεί ως επόπτης, αλλά ανοίγει μία διεπαφή (/dev/kvm) μέσω της οποίας μπορούν να χρησιμοποιηθούν οι διάφορες λειτουργίες που προσφέρει. Σε αυτό τον συνδυασμό (Qemu/Kvm) κάθε εικονική μηχανή εκτελείται ως μία διεργασία.

2.3 Λειτουργικά συστήματα

Το λειτουργικό σύστημα είναι ένα πρόγραμμα το οποίο εκτελείται σε ένα υπολογιστικό σύστημα. Η ειδοποιός διαφορά του με οποιοδήποτε άλλο πρόγραμμα είναι ότι το πρώτο υπάρχει για να εξυπηρετεί το δεύτερο. Το λειτουργικό σύστημα διαχειρίζεται το υλικό του υπολογιστικού συστήματος και παρέχει στους πργαμματιστές των εφαρμογών ένα σαφές αφηρημένο σύνολο πόρων, αντί για το μπερδεμένο σύνολο που υπάρχει στο υπολογιστικό σύστημα. Χωρίς αυτό, ο προγραμματιστής μίας οποιασδήποτε εφαρμογής θα έπρεπε να γράψει και τον κατάλληλο κώδικα για το υλικό που χρησιμοποιεί η συγκεκριμένη εφαρμογή (π.χ. αποθήκευση σε σκληρό δίσκο). Ως εκ τούτου θα σπαταλούσε αρκετό χρόνο σε μη βασικές λειτουργίες του προγράμματος. Επιπροσθέτως πολλές από αυτές τις λειτουργίες που αφορούν το υλικό μπορεί να έχουν ήδη υλοποιηθεί από άλλα προγράμματα, σπαταλώντας έτσι χρόνο για ήδη υπάρχουσες λειτουργίες.

Η πληθώρα και οι σημαντικές διαφορές στο σκοπό και στις δυνατότητες των υπολογιστικών συστημάτων οδήγησε στη δημιουργί-

α διαφορετικών ειδών λειτουργικών συστημάτων. Για παράδειγμα, ένα λειτουργικό σύστημα που έχει δημιουργηθεί με σκοπό την εκτέλεση του σε προσωπικούς υπολογιστές διαφέρει αρκετά από ένα λειτουργικό σύστημα που χρησιμοποιείται σε ενσωματωμένα συστήματα. Το πιο γνωστό είδος λειτουργικών συστημάτων είναι αυτά του γενικού σκοπού, τα οποία έχουν ως στόχο να μπορούν να εκτελεστούν σε διάφορα υπολογιστικά συστήματα προσφέροντας όσο το δυνατόν περισσότερες λειτουργίες.

Ένας διαφορετικός τρόπος κατηγοριοποίησης των λειτουργικών συστημάτων γίνεται με τη δομή τους. Αναφέρουμε τις κυριότερες από αυτές τις δομές, οι οποίες είναι τα μονολιθικά συστήματα, τα πολυεπίπεδα συστήματα και οι μικροπυρήνες.

- μονολιθικά συστήματα: η πιο διαδεδομένη οργάνωση, στην οποία ολόκληρο το λειτουργικό συστημα εκτελείται ως ένα μοναδικό πρόγραμμα σε κατάσταση πυρήνα. Ουσιαστικά πρόκειται για ένα μεγάλο εκτελέσιμο δυαδικό πρόγραμμα. Κάθε διαδικασία του συστήματος μπορεί να καλέσει οποιαδήποτε άλλη.

- πολυεπίπεδα συστήματα: το λειτουργικό σύστημα οργανώνεται σε μία ιεραρχία επιπέδων, όπου το καθένα δομείται πάνω στο αμέσως κατώτερο του. Κάθε επίπεδο αναλαμβάνει μία διαφορετική λειτουργία και μπορεί να χρησιμοποιήσει μόνο λειτουργίες που υποστηρίζουν κατώτερα επίπεδα.

- μικροπυρήνες: το λειτουργικό σύστημα διαιρείται σε μικρές καλά ορισμένες υπομονάδες, από τις οποίες μόνο μία, ο μικροπυρήνας, εκτελείται σε κατάσταση πυρήνα ενώ οι υπόλοιπες εκτελούνται σε σχετικά λιγότερο ισχυρές κοινές διεργασίες χρήστη.

Σε ένα μονολιθικό λειτουργικό συτημα διακρίνονται δύο ξεχωριστά επίπεδα εκτέλεσης κώδικα. Κάθε επίπεδο έχει διαφορετικούς χώρους μνήμης και διαφορετικά δικαιώματα πρόσβασης στο υλικό. Τα δύο αυτά επίπεδα είναι ο χώρος χρήστη (user space), στο οποίο εκτελούνται οι διεργασίες του χρήστη και ο χώρος πυρήνα (kernel space), στο οποίο εκτελείται ο κώδικας του πυρήνα. Προκειμένου μία διεργασία να εκτελέσει μία λειτουργία για την οποία δε διαθέτει τα ανάλογα δικαιώματα, υπάρχει ένας μηχανισμός μέσω του οποίου η διεργασία ζητά από τον πυρήνα να εκτελέσει τη συγκεκριμένη λειτουργία για λογαριασμό της. Ο μηχανισμός αυτός ονομάζεται κλήση συτήματος (system call).

Ένα από τα πρώτα και πιο ιστορικά λειτουργικά συστήματα είναι το UNIX. Το UNIX επιλέχτηκε ως η βάση για μία τυποποιημένη διεπαφή συστήματος, ωστόσο υπήρχαν και δημιουργήθηκαν πολλές παραλλαγές του. Συνεπώς, δημιουργήθηκε η ανάγκη για τη δημιουργία ενός κοινού standard, ώστε να διατηρείται η συμβατότητα μεταξύ αυτών

των διάφορων εκδόσεων του UNIX. Το standard που δημιουργήθηκε ήταν το POSIX, το οποίο ορίζει μία ελάχιστη διασύνδεση κλήσεων συστήματος που πρέπει να υποστηρίζουν τα συμβατά συστήματα UNIX. Μάλιστα το POSIX χρησιμοποιείται ακόμα και στις μέρες μας, ενώ έχουν δημιουργηθεί και κάποιες παραλλαγές του. Το UNIX επηρέασε σε μεγάλο βαθμό και κάποια από τα πιο δημοφιλη σύγχρονα λειτουργικά συστήματα όπως το Linux και αυτά της οικονγένειας BSD. Ένα από αυτά τα λειτουργικά συστήματα είναι και το NetBSD, το οποίο είναι ένα γενικού σκοπού λειτουργικό σύστημα. Χρησιμοποιείται συνήθως σε servers και λιγότερο σε προσωπικούς υπολογιστές, ενώ υποστηρίζει αρκετές αριτεκτονικές.

Κεφάλαιο 3

Unikernel frameworks

Σε αυτό το κεφάλαιο θα γίνει μία εισαγωγή στην έννοια των unikernels και το σκοπό τους, ενώ θα παρουσιαστούν ορισμένα unikernel frameworks, τα οποία μελετήθηκαν κατά τη διάρκεια αυτής της εργασίας.

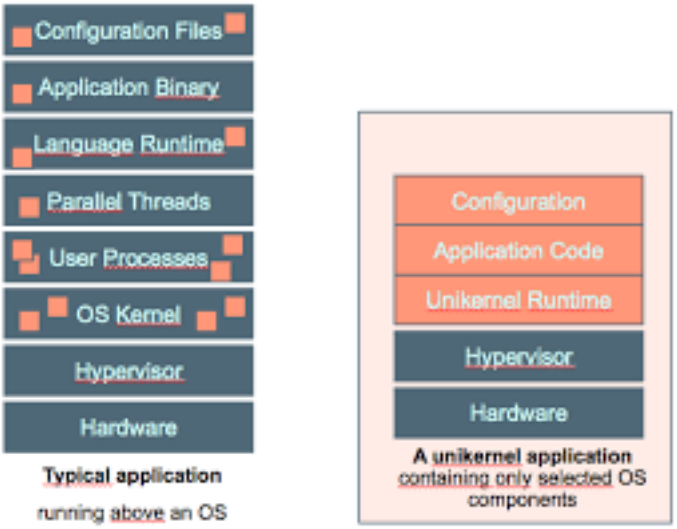
Ένας τυπικός ορισμός των unikernels αναφέρεται στη σελίδα unikernel.org και είναι ο εξής:

Τα Unikernels είναι εξειδικευμένες εικόνες μηχανής, με ένα μοναδικό χώρο διευθύνσεων, τα οποία κατασκευάζονται χρησιμοποιώντας library operating systems

Αναλύοντας τον παραπάνω ορισμό μπορούμε εύκολα να συμπεράνουμε τα βασικά χαρακτηριστικά των unikernels:

- εξειδικευμένες εικονικές μηχανής: κάθε unikernel μπορεί να είναι διαφορετικός από τους υπόλοιπους. Κάθε ένας αφορά μία συγκεκριμένη εφαρμογή και φτιάχνεται γύρω από αυτήν.
- ένας μοναδικός χώρος διευθύνσεων: στην περίπτωση των unikernels δεν υπάρχει ο διαχωρισμός μεταξύ user space και kernel space, όπως στα συμβατικά λειτουργικά συστήματα.
- library operating systems [22]: πρόκειται για λειτουργικά συστήματα τα οποία παρέχουν τις υπηρεσίες που υποστηρίζουν (networking κ.λ.π.) στη μορφή βιβλιοθηκών, οι οποίες στη συνέχεια μπορούν να χρησιμοποιηθούν (link) από τις εφαρμογές.

Με λίγα λόγια ένας unikernel αποτελείται από τον κώδικα της εφαρμογής την οποία θα εκτελέσει, τα απαραίτητα κομμάτια του λειτουργικού συστήματος που η συγκεκριμένη εφαρμογή χρειάζεται και τις κατάλληλες παραμετροποιήσεις. Τα unikernels δεδομένου ότι αφορούν μία και μόνο εφαρμογή δεν υποστηρίζουν περισσότερες από μία διεργασίες, ούτε περισσότερους από έναν χρήστες. Αυτόματα



Σχήμα 3.1: Unikernel vs OS software stack

όλο το κομμάτι των συμβατικών λειτουργικών συστημάτων που αφορούν τη διαχείρηση και υποστήριξη πολλών διεργασίων και χρηστών καθίσταται άχρηστο και δεν συμπεριλαμβάνεται. Τα πράγματα είναι διαφορετικά όσον αφορά τα νήματα, καθώς άλλα unikernel frameworks τα υποστηρίζουν ενώ άλλα όχι.

Στην εικόνα 3.1 φαίνεται η διαφορά μεταξύ ενός συμβατικού λειτουργικού συστήματος και ενός unikernel. Τα παραπάνω ενοποιούνται σε μία εικόνα μηχανής που μπορεί να εκτελεστεί από έναν επόπτη, ή ακόμα και στο υλικό. Γίνεται εύκολα αντιληπτό ότι το μέγεθος της τελικής εικόνας θα είναι πολύ μικρότερο από αυτή ενός συμβατικού λειτουργικού συστήματος. Μία ακόμη συνέπεια αυτού είναι ότι οι χρόνοι εκκίνησης θα είναι αρκετά μικρότεροι, αφού πλέον δε χρειάζεται να φορτωθούν και να εκκινήσουν όλες οι υπηρεσίες που προσφέρει ένα λειτουργικό σύστημα.

Από άποψη ασφάλειας, ο σχεδιασμός των unikernels από μόνος του προσφέρει αρκετά πλεονεκτήματα. Αρχικά το μικρό μέγεθος των unikernels συνεπάγεται αυτόματα και μικρότερο εύρος επίθεσης από κάποιον κακόβουλο χρήστη. Επιπλέον πρόκειται για εξειδικευμένες εικόνες που αφορούν μία και μόνο εφαρμογή, οπότε στην περίπτωση που αποκτηθεί πρόσβαση στην εικονική μηχανή, το περιθώριο εκμετάλλευσης θα είναι αρκετά μικρό. Τέλος, υπάρχει απομόνωση μεταξύ των εικονικών μηχανών η οποία προσφέρεται από τον επόπτη.

Η έλλειψη διαχωρισμού kernel space και user space, μειώνει το κόστος μετάβασης από τον ένα χώρο διευθύνσεων στον άλλο, κάνοντας έτσι τις κλήσεις συστήματος να λειτουργούν σαν απλές κλήσεις συναρτήσεων. Ως αποτέλεσμα, η ταχύτητα εκτέλεσης αυξάνει σημαντικά, καθώς δε χρειάζονται πλέον οι μεταβάσεις από τον έ-

να χώρο διευθύνσεων στον άλλο, μία αρκετά δαπανηρή λειτουργία. Ένας όμως, από τους βασικούς ρόλους ύπαρξης ττου διαχωρισμού αυτού ήταν η ασφάλεια, μεταξύ των διεργασιών και γενικότερα του συστήματος. Ωστόσο, στην περίπτωση των unikernels υπάρχει μία και μόνο διεργασία, η οποία μάλιστα δεν είναι κακόβουλη.

Η έννοια των library operating systems δεν είναι καινούρια, ωστόσο δεν μπορούσαν να χρησιμοποιηθούν λόγω του μεγάλου πλήθους των συσκευών και συνεπώς των drivers που θα έπρεπε να υποστηρίζουν. Χάρις την εικονικοποίηση η διαχείριση των συσκευών γίνεται από τους επόπτες, κάνοντας εφικτή τη χρήση των συγκεκριμένων λειτουργικών συστημάτων.

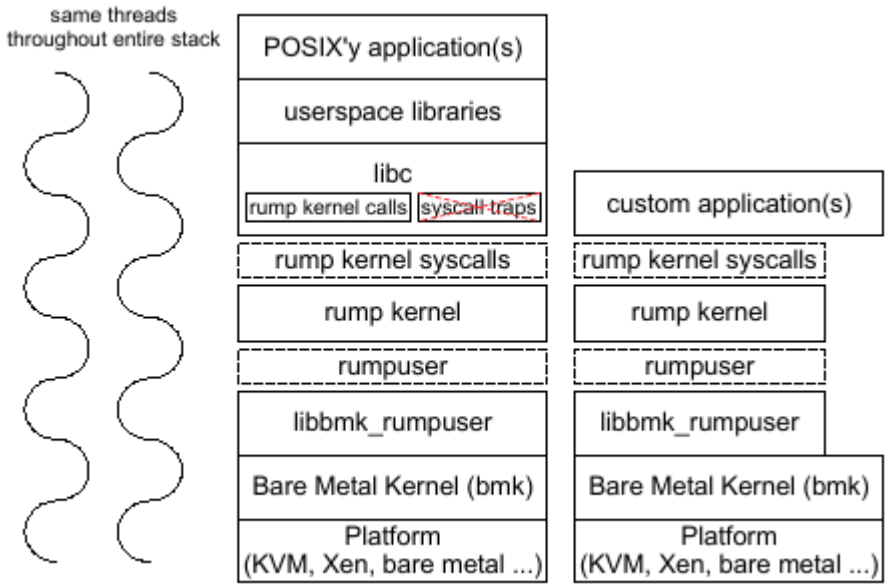
Αφ' ετέρου η απομάκρυνση όλων αυτών των υπηρεσιών από το λειτουργικό σύστημα, δημιουργεί αρκετούς περιορισμούς. Οι περισσότερες εφαρμογές έχουν φτιαχτεί για συμβατικά λειτυργικά συστήματα λαμβάνοντας υπόψιν και χρησιμοποιοώντας τις συγκεκριμένες υπηρεσίες. Απαιτείται, λοιπόν, να γίνουν σημαντικές αλλαγές στις εφαρμογές για να μπορέσουν να εκτελεστούν σε unikernels. Επιπλέον γίνεται πιο δύσκολη η αποσφαλμάτωση των unikernels δεδομένου των λίγοστών υπηρεσιών που προσφέρουν.

Ακολουθεί μία συνοπτική περιγραφή ορισμένων από τα unikernel frameworks που μελετήθηκαν κατά τη διάρκεια της εργασίας.

3.1 Rumprun

Το rumprun είναι ένα unikernel framework, το οποίο έχει χτιστεί με βάση τα συστατικά των drivers που προσφέρουν οι rump kernels. Το rumprun μπορεί να προσφέρει ένα POSIX-like περιβάλλον, που μπορεί να χρησιμοποιηθεί από POSIX εφαρμογές ώστε να μετατραπούν χωρίς καμία αλλαγή σε unikernel εικόνες. Αυτός ήταν και ένας από τους βασικούς στόχους του εγχειρήματος, η δυνατότητα δηλαδή σε POSIX κώδικα να μπορεί να εκτελεστεί χωρίς να υποστεί καμία αλλαγή. Στην εικόνα 3.2 φαίνεται η δομή του rumprun για POSIX εφαρμογές (αριστερά) και προσαρμοσμένες εφαρμογές (δεξιά). Προφανώς το δεξί μοντέλο φαίνεται αρκετά μικρότερο, αλλά είναι απαραίτητες να γίνουν συγκεκριμένες αλλαγές στην εφαρμογή. Ο συγκεκριμένος unikernel μπορεί να εκτελεστεί χρησιμοποιώντας Xen ή KVM, ενώ υποστηρίζει arm και x86 αρχιτεκτονικές. Ο κώδικας του rumprun είναι διαθέσιμος στο `http://repo.rumpkernel.org/rumprun`.

Όπως κάθε unikernel framework το rumprun υποστηρίζει μία και μόνο διεργασία. Μολαταύτα υποστηρίζει POSIX threads, δίνοντας τη δυνατότητα για την ύπαρξη περισσοτέρων από ένα νήματα. Βέβαια ο χρονοδρομολογητής που διαθέτει είναι cooperative, οπότε αν ένα νήμα αποκτήσει τον έλεγχο δεν πρόκειται να διακοπεί από το



Σχήμα 3.2: Rumprun software stack

χρονδορομοληγτή αλλά θα αφήσει το έλεγχο όταν εκείνο τερματίσει ή πρόκειται να μπλοκάρει. Κάποιοι ακόμα περιορισμοί, είναι η μη υποστήριξη εικονικής μνήμης (virtual memory) και σημάτων (signals), Συνεπώς εφαρμογές που χρησιμοποιούν σήματα ή κλήσεις συστήματος όπως η mmap() δε θα λειτουργούν χωρίς αλλαγές στον κώδικα τους.

Ένας rumprun unikernel αποτελείται από τα απαραίτητα κομμάτια των rump kernels που χρειάζεται η εφαρμογή και την ίδια την εφαρμογή. Για τη δημιουργία του unikernel γίνεται πάντα cross-compilation, οπότε δε χρειάζεται κάποιος ήδη ΄ετοιμος rumprun unikernels, αλλά μία σειρά από εργαλεία. Η διαδικασία έχει ως εξής:

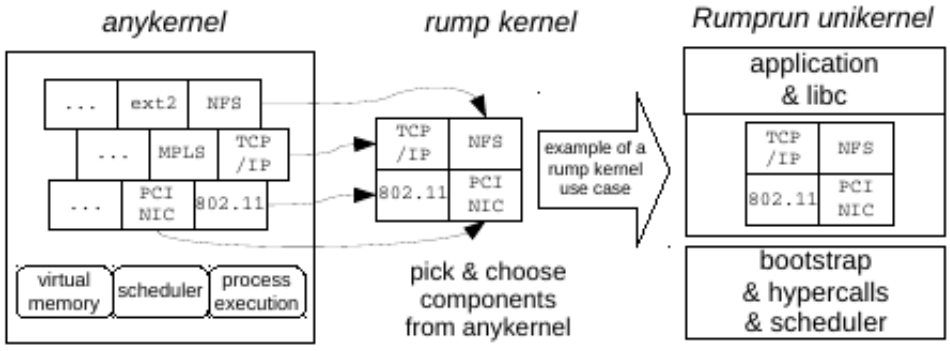
1. Αρχικά γίνεται η μεταγλώττιση του κώδικα της εφαρμογής χρησιμοποιώντας έναν cross-compiler.

2. Ύστερα, αντί για linking γίνεται pseudo-linking όπως ονομάζουν τη διαδικασία κατά την οποία απλά ελέγχονται αν ικανοποιούνται όλες οι εξαρτήσεις συμβόλων, χωρίς να γίνεται κάποια σύνδεση με κάποιο συστατικό του λειτουργικού συστήματος.

3. Τέλος, το παράγωγο του προηγούμενου βήματος γίνεται ”bake”, όπου εισάγονται και τα κομμάτια του λειτουργικού συστήματος που απαιτούνται, όπως έχουν οριστεί κατά την εκτέλεση της εντολής.

Όπως έχει αναφερθεί το rumprun unikernel βασίζεται στα rump kernels [13]. Τα rump kernels παρέχουν drivers του NetBSD ως φορητά εξαρτήματα με τα οποία μπορούμε να εκτελέσουμε εφαρμογές χωρίς να ναι απαραίτητη η ύπαρξη λειτουργικού συστήματος [14]. Η αρχική ιδέα ήταν να υπάρχει η δυνατότητα να εκτελούνται αμετάβλητοι drivers του NetBSD ως απλά προγράμματα σε userspace, ώστε να είναι πιο εύκολος ο έλεγχος και η ανάπτυξη NetBSD drivers. Ένας rump kernel είναι ένας πυρήνας χρονομερισμού (timesharing) από τον οποίο έχουν αφαιρεθεί ορισμένα κομμάτια [12]. Τα κομμάτια που κρατήθηκαν είναι οι drivers και οι απαραίτητες ρουτίνες που χρειάζονται οι συγκεκριμένοι drivers για να λειτουργήσουν (συγχρονισμός, κατανομή μνήμης κ.λ.π.). Τα κομμάτια που αφαιρέθηκαν είναι αυτά που αφορούν τις διεργασίες, την εικονική μνήμη κ.α.. Επιπλέον τα rump kernels έχουν και ένα καλώς ορισμένο στρώμα φορητότητας, ώστε να είναι εύκολο να ενσωματωθούν σε διάφορα περιβάλλοντα.

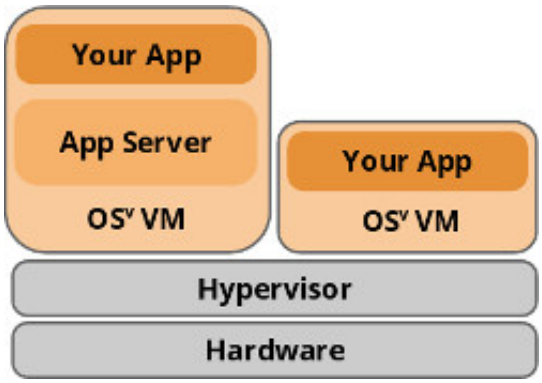
Πίσω από τα rump kernels υπάρχει μία ακόμα έννοια αυτή του anykernel [12]. Ο συγκεκριμένος όρος δημιουργήθηκε ως απάντηση στην όλο και αυξανόμενο αριθμό μοντέλων λειτουργικών συστημάτων (monolithic, exokernel, mikrokernel, unikernel κ.λ.π.). Ένα πολύ μεγάλο ποσοστό ενός λειτουργικού συστήματος ανεξάρτητα από το μοντέλο που χρησιμοποιεί αποτελείται από drivers. Ο όρος anykernel περιγράφει ένα κώδικα βάσης (codebase) τύπου κώδικα πυρήνα από τον οποίο οι οδηγοί μπορούν να εξαχθούν και να ενσωματωθούν σε οποιοδήποτε μοντέλο λειτουργικού συστήματος, χωρίς να χρειάζονται αλλαγές και συντήρηση. Στην εικόνα 3.3 φαίνεται η σχέση μεταξύ των εννοιών anykernel, rump kernels και rumprun.



Σχήμα 3.3: Σχέση μεταξύ των ενοιών anykernel, rump kernel και rumprun

3.2 OSv

Το OSv δημιουργήθηκε από το μηδέν, με σκοπό να μπορεί να εκτελέσει μία εφαρμογή πάνω από έναν επόπτη. Ως γλώσσα προγραμματισμού χρησιμοποιεί τη C++ και ο κώδικας του είναι διαθέσιμος στο `https://github.com/cloudius-systems/osv`. Υποστηρίζει διάφορους επόπτες και επεξεργαστές, όπως arm και x86, ενώ στην x86 αρχιτεκτονική υποστηρίζει τους επόπτες Xen, KVM, VMWare, Virtualbox. Παρέχει τη δυνατότητα να μπορεί να υποστηρίξει σχεδόν οποιοδήποτε πρόγραμμα, που δε χρειάζεται παραπάνω από μία διεργασία. Μάλιστα παρέχει υποστήριξη για αρκετές γλώσσες προγραμματισμού όπως C, Java, Ruby, Node.js, Perl και άλλες.



Σχήμα 3.4: Osv unikernel stack

Η επιλογή των δημιουργών του OSv να δημιουργήσουν από την αρχή ένα νέο λειτουργικό σύστημα τους έδωσε τη δυνατότητα να εισάγουν ορισμένες νέες τεχνικές [15]. Σε αυτό το πλαίσιο στο OSv δεν υπάρχουν spinlocks. Στα περισσότερα SMP (Symmetric multiprocessing) λειτουργικά συστήματα χρησιμοποιούνται spinlocks για την προστασία κοινών δεδομένων από πολλά νήματα. Όταν ένα νήμα κατέχει το spinlock για μία δομή δεδομένων τα υπόλοιπα νήματα δεν μπορούν να αποκτήσουν πρόσβαση στη συγκεκριμένη δομή και περιμένουν, κάνοντας συνεχώς ερωτήματα αν το κλείδωμα είναι ελεύθερο. Τα spinlocks χρησιμοποιούνται κυρίως για νήματα που δεν μπορούν να μπλοκάρουν (κυρίως νήματα του πυρήνα). Ωστόσο, στην περίπτωση της εικονικοποίησης τα spinlocks δημιουργούν το πρόβλημα του ”lock-holder preemption” [26]. Αν μία εικονική CPU σταματήσει να εκτελείται και κρατάει ένα spinlock, οι άλλες CPU που χρειάζονται το συγκεκριμένο spinlock, είναι αναγκασμένες να περιμένουν σπαταλώντας κύκλους της CPU. Υπό αυτές τις συνθήκες στο OSv αποφάσισαν όλη η δουλειά του πυρήνα να γίνεται από νήματα που μπορούν να ”κοιμηθούν”. Τα παραπάνω δεν ισχύουν για το χρονοδρομολογητή, ο οποίος δεν μπορεί να ”κοιμηθεί”, οπότε

χρησιμοποιεί ουρές εκτέλεσης για κάθε επεξεργαστή, ωστε να μην απαιτείται συννενόηση μεταξύ των επεξεργαστών για τη χρονοδρομολόγηση, ενώ αν ένα νήμα αλλάξει επεξεργαστή χρησιμοποιούνται lock-free αλγόριθμοι.

Όπως αναφέρθηκε προηγουμένως το OSv διαθέτει χρονοδρομολογητή. Μερικά από τα χαρακτηριστικά του είναι: διακοπτόμενη χρονοδρομολόγηση (preemptive scheduling), η μη χρήση spinlocks και timer interrupts, ενώ τέλος δε γίνεται κανένας διαχωρισμός μεταξύ των νημάτων του πυρήνα και της εφαρμογής. Ένα εξίσου ενδιαφέρον κομμάτι στο σχεδιασμό του OSv είναι το network stack. Λαμβάνοντας υπόψιν ότι πρόκειται για ένα λειτουργικό συστημα για το cloud, το OSv έχει δώσει μεγάλη βαρύτητα στην υποστήριξη εφαρμογών που χρησιμοποιούν πολύ το δίκτυο. Το network stack λοιπόν, είναι σχεδιασμένο με βάση τις ιδέες του Van Jacobson [11], με βάση τις οποίες το OSv υποστηρίζει ότι πετυχαίνει πολύ υψηλότερη απόδοση από τα network stacks των συμβατικών λειτουργικών συστημάτων.

Η εκτέλεση μίας εφαρμογής στο OSv δε διαφέρει αρκετά από την εκτέλεση της σε ένα συμβατικό λειτουργικό σύστημα, όπως π.χ. το Linux. Η διαφορά είναι ότι στο OSv δεν εκτελούνται τα συνηθισμένα fixed-position εκτελέσιμα αλλά απαιτούνται shared objects. Για την εκτέλεση του το OSv αναζητά την main συνάρτηση της εφαρμογής και την εκτελεί. Για το πως θα τοποθετηθεί η εφαρμογή στο unikernel, προσφέρονται δύο τρόποι. Εϊτε όταν χτίζεται το unikernel ορίζουμε να συμπεριληφθεί και το αντίστοιχο shred object, είτε μπρορούμε να το φορτώσουμε στο unikernel μεταγενέστερα, μέσω του REST API που προσφέρει το OSv. Μάλιστα, υπάρχει η επιλογή για διαχείρηση του unikernel μέσω μία ιστοσελίδας, ενώ υπάρχει η δυνατότητα να υπάρχει και ένα πολύ βασικό κέλυφος (shell). Αξιοσημείωτη είναι και η προσπάθεια δημιουργίας ενός εργαλείου για τη δημιουργία unikernels με OSv. Το εργαλείο αυτό ονομάζεται Capstan και μοιάζει αρκετά στη χρήση του με το Docker.

Σαν γενική εικόνα το OSv μοιάζει με ένα library operating system, το οποίο υποστηρίζει πολλές λειτουργίες των συμβατικών λειτυργικών συστημάτων. Ενδεικτικό είναι το γεγονός ότι διαθέτει εικονική μνήμη, καθώς και ένα εικονικό σύστημα αρχείων. Χάρις όλα αυατά τα χαρακτηριστικά μπορεί να υποστηρίξει ένα πολύ μεγάλο εύρος εφαρμογών ακόμα και αυτές που κάνουν χρήση της mmap(). Ο μόνος περιορισμός που υπάρχει είναι ότι δεν υποστηρίζεται η fork(), καθώς παραβιάζει το single-process χαρακτήρα. Από την άλλη μεριά, όλες αυτές οι λειτουργίες που προσφέρονται αυξάνουν το μέγεθος του unikernel, κάνοντας το OSv ένα από τα πιο ”βαριά” unikernel frameworks.

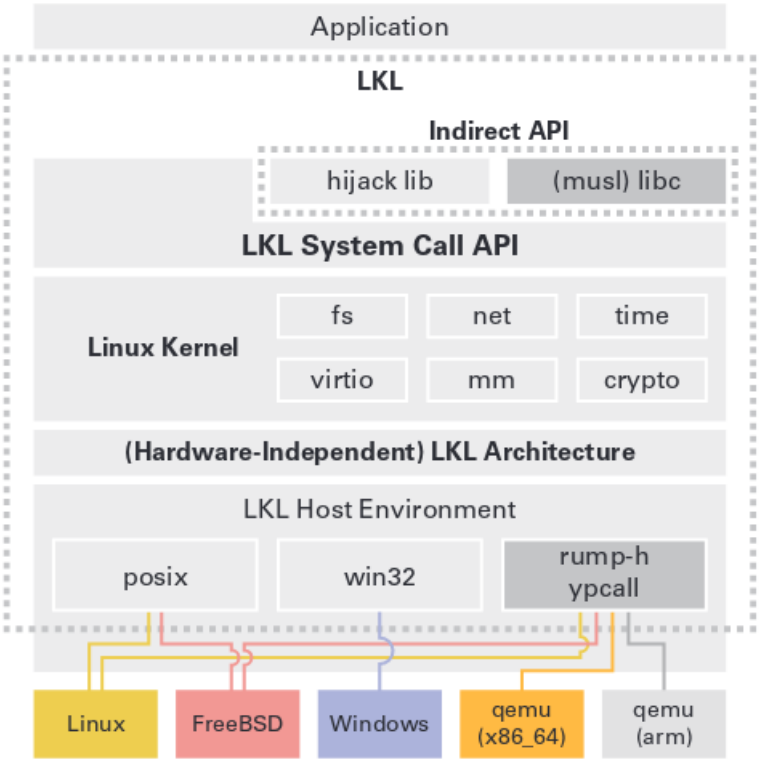
3.3 Linux Kernel Library - lkl

Το lkl [23] φέρνει τις ιδέες του rumpkernel και του anykernel στο Linux. Βασικός στόχος του είναι η επαναχρησιμοποίηση του κώδικα του Linux με όσο το δυνατόν λιγότερη προσπάθεια και μειωμένο κόστος συντήρησης. Έχει υλοποιηθεί ως ένα port του Linux σε μία εικονική αρχιτεκτονική (lkl), προκειμένου να μειωθούν οι ενοχλητικές τροποποιήσεις του πυρήνα. Το lkl μπορεί να χρησιμποιηθεί για τρεις διαφορετικούς λόγους, α) στιγμιαία παράκαμψη του πυρήνα, β) επαναχρησιμοποίηση κώδικα πυρήνα στο χώρο χρήστη και τέλος γ) ως unikernel. Βασικά χρησιμοποιείται περισσότερο για τη χρήση ορισμένων λειτουργιών του πυρήνα του Linux σε άλλα λειτουργικά συστήματα ή στο χώρο χρήστη, παρά ως unikernel.

Το lkl στην ουσία είναι βιβλιοθήκες τις οποίες οι εφαρμογές μπορούν να χρησιμοποιήσουν. Για τη χρήση του, λοιπόν, προσφέρει ένα δικό του API, που αποτελεί ένα υποσύνολο των κλήσεων συστήματος του Linux. Στην εικόνα 3.7 φαίνονται όλα τα συστατικά του lkl. Όπως φαίνεται υπάρχουν δύο APIs 1) lkl system call API, που είναι βασισμένο στις κλήσεις συστήματος του Linux και 2) το API που χρησιμοποιεί το πρώτο έμμεσα. Το δεύτερο API αποτελείται από τη hijack library και μία βασική υλοποίηση βιβλιοθήκης και συγκεκριμένα μία τροποποίηση της musl libc ώστε να χρησιμοποιεί το lkl system call API. Η hijack library χρησιμοποιείται για την επί τόπου αντικατάσταση των κλήσεων συστήματος που εκτελεί μία εφαρμογή, ώστε να μπορεί η εφαρμογή να χρησιμοποιεί το lkl αντί για τον πυρήνα του host.

Τα υπόλοιπα κομμάτια του lkl έχουν να κάνουν με το περιβάλλον του host και την generic lkl architecture. Δεδομένου, ότι το lkl δε στοχεύει μόνο σε ένα περιβάλλον, δε χρησιμοποιεί κώδικα που εξαρτάται από την πλατφόρμα. Αντ' αυτού η εφαρμογή θα πρέπει να υλοποιεί ορισμένα θεμελιακά στοιχεία που εξαρτόνται από το περιβάλλον. Αυτά τα θεμελιακά στοιχεία χρησιμοποιούνται από την lkl generic architecture για τη δημιουργία της εικονικής μηχανής πάνω στην οποία θα εκετελεστεί ο πυρήνας του Linux. Ωστόσο, υπάρχουν ορισμένα περιβάλλοντα για τα οποία το lkl παρέχει αυτά τα θεμελιακά στοιχεία. Τα περιβάλλοντα αυτά είναι το POSIX, το NT (windows userspace), το NTK (Windows kernel), το Apache Portable Runtime, ενώ υπάρχει υποστήριξη και για το rumprun.

Το lkl επιτρέπει την εκτέλεση μίας και μόνο διεργασίας, ενώ παρέχει υποστήριξη για πολλαπλά νήματα με ορισμένους περιορισμούς. Αρχικά δεν υπάρχει υποστήριξη για SMP και επιπλέον μία εφαρμογή περιορίζεται απλά στη δημιουργία και τον τερματισμό ένός νήματος. Όσον αφορά τη διαχείρηση της μνήμης, αφού δε χρειάζεται κάποια προστασία μνήμης το lkl απλά χρειάζεται ένα χώρο μνήμης από τον οποίο μπορεί ο πυρήνας του Linux να κατανείμει μνήμη που χρειάζε-



Σχήμα 3.5: Η αρχιτεκτονική του lkl

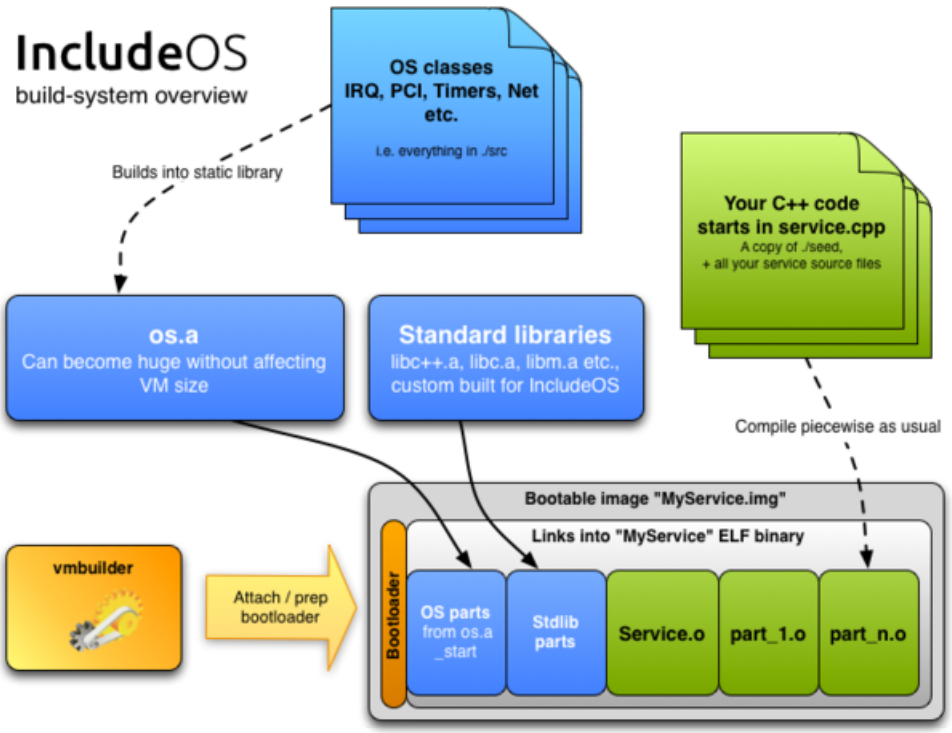
ται. Επιπλέον χάρις τη δυνατότητα του Linux να παρέχει εικονική μνήμη ακόμα και για non-MMU αρχιτεκτονικές μπορεί να υποστηριχθεί και εικονική μνήμη.

3.4 IncludeOS

Το IncludeOS [7] είναι ένα από τα νεότερα Unikernel frameworks. Είναι γραμμένο σε C++ και υποστηρίζει εφαρμογές που χρησιμοποιούν την ίδια γλώσσα προγραμματισμού. Βασικός στόχος για το συγκεκριμένο unikernel framework είναι να έχει τόσο απλή χρήση όσο η χρήση μίας βιβλιοθήκης στη C++. Πιο συγκεκριμένα μία εντολή της μορφής #include <os> στην αρχή μίας εφαρμογής θα έχει ως αποτέλεσμα το χτίσιμο μίας εικόνας δίσκου (disk image) που μπορεί να εκτελεστεί από τους περισσότερους επόπτες (kvm, virtualbox). Για να υπάρχει αυτή η δυνατότητα οι δημιουργοί του IncludeOS, έχουν αναπτύξει και μία σειρά εργαλείων. Χρησιμοποιώντας τα, η διαδικασία του ”χτισίματος” περιλαμβάνει τρία στάδια. Αρχικά, γίνεται η σύνδεση της εφαρμογής με τα μέρη του λειτουργικού συστήματος που χρειάζονται. Στη συνέχεια, προσάπτεται ο boot loader και τέλος όλα αυτά ενοποιούνται σε ένα disk image. Η διαδικασία του χτισίματος φαίνεται και στην εικόνα 3.6.

Αρκετά ενδιαφέρουσες είναι και ορισμένες σχεδιαστικές επιλογές των δημιουργών που αφορούν κυρίως το network stack. Χρησιμοποιεί έναν virtio driver για την επικοινωνία με τον επόπτη, συνεπώς δε χρειάζεται ο τελευταίος να προσομοιώνει διάφορες συσκευές αλλά απλά να χρησιμοποιεί το virtio. Επιπροσθέτως, η επιλογή να φτιαχτεί από την αρχή το συγκεκριμένο framework, έδωσε τη δυνατότητα στη δημιουργία ενός νέου TCP/IP stack, χρησιμοποιώντας σε μεγάλο βαθμό τις δυνατότητες που έδινε η C++. Τέλος, το IncludeOS δεν έχει κάποιο τρόπο να φορτώσει ένα πρόγραμμα, συνεπώς δεν υπάρχει η κλασσική main συνάρτηση. Αντ’ αυτού, υπάρχει η κλάση Service και ο προγραμματιστής πρέπει να υλοποιήσει την Service::start η οποία θα κληθεί μετά την αρχικοποίηση του λειτουργικού συστήματος.

Όπως έχει αναφερθεί προηγουμένως, μόνο C++ προγράμματα μπορούν να εκτελεστούν στο IncludeOS, ενώ μπορεί να χρησιμοποιηθεί και η standard βιβλιοθήκη της C. Με αυτό τον τρόπο υπάρχει μία μερική υποστήριξη για POSIX εφαρμογές. Μολαταύτα βασικά χαρακτηριστικά του POSIX που αφορούν την είσοδο/έξοδο (I/O) και τα νήματα δεν υποστηρίζονται. Στο IncludeOS όλη η εργασία γίνεται από ένα και μόνο νήμα, χωρίς I/O μπλοκάρισμα ή context switching μέσα στο guest, με το συγκεκριμένο μοντέλο να θυμίζει εκείνο του Node.js. Όπως και στα υπόλοιπα unikernel frameworks μόνο μία διεργασία μπορεί να υπάρχει ανά unikernel, ενώ δεν υπάρχει και εικονική μνήμη.



Σχήμα 3.6: IncludeOS building process

3.5 ClickOS

Το ClickOS είναι ένα unikernel framework το οποίο δημιουργήθηκε με γνώμονα την υποστήριξη εφαρμογών Network Function Virtualization (NFV). Η ιδέα του NFV είναι η μεταφορά της εργασίας που γίνεται στα middleboxes από συσκευές υλικού σε ειδικό λογισμικό που μπορεί να εκτελεστεί σε οποιοδήποτε υλικό σαν μία εφαρμογή. Τα middleboxes είναι συσκευές δικτύου, οι οποίες μετατρέπουν, ελέγχουν, φιλτράρουν και γενικότερα χειρίζονται διαδικτυακή κίνηση για σκοπούς διαφορετικούς από την προώθηση πακέτων. Οι συσκευές αυτές συνήθως είναι ακριβές, δύσκολα διαχειρίσιμες και είναι αρκετά δύσκολο να αλλάξει η λειτουργία τους. Χρησιμοποιώντας τις τεχνολογίες εικονικοποίησης το NFV αναπτύσει τις δικτυακές λειτουργίες σε εικονικές μηχανές. Με αυτό τον τρόπο αντιμετωπίζονται τα παραπάνω προβλήματα των middleboxes, ενώ παράλληλα επιτρέπεται σε παραπάνω από μία λειτουργεία να εκτελεστεί στο ίδιο υλικό.

Το ClickOS μετατρέπει το Click modular router [16] σε μία εικονική μηχανή που μπορεί να εκετελεστεί στο Xen. Για να γίνει αυτό εφικτό χρησιμοποείται το Mini-OS (βλέπε ενότητα 3.8). Η δομή του ClickOS είναι αρκετά απλή με το click να εκτελείται στο Mini-OS όπως φαίνεται και στην εικόνα 3.7. Το Mini-OS διαθέτει τα βασικά χαρακτηριστικά που χρειάζονται για να μπορέσει να εκετελεστεί το Click, όπως διαχείρηση μνήμης, τους απαραίτητους drivers και έναν απλό χρονοδρομολογητή νημάτων. Έτσι σε κάθε εικόνα του ClickOS μπορούν αν συνυπάρχουν περισσότερα από ένα click modular routers, κάθε ένα από τα οποία εκτελείται σε διαφορετικό νήμα. Το configuration για κάθε router γράφεται στο xenstore και ένα είδικό νήμα είναι υπεύθυνσ για την παρακολούθηση τυχόν αλλαγών σε αυτό και την εφαρμογή του configuration στο αντίστοιχο router. Ωστόσο κάθε στιγμή μόνο ένα νήμα εκετελείται, ενώ η χρονοδρομολόγηση είναι non-preemptive.

Το ClickOS παρουσιάζει αρκετά υψηλές δικτυακές επιδόσεις, ενώ παράλληλα διαθέτει και άλλα θετικά στοιχεία των unikernels, όπως μικρό μέγεθος, γρήγορη εκκίνηση της εικονικής μηχανής και μικρή κατανάλωση μνήμης. Σημαντικό ρόλο στις επιδόσεις του δικτύου είχαν οι αλλαγές που έγιναν σε όλη την αρχιτεκτονική του δικτύου στο Xen. Το Xen ακολουθεί το μοντέλο του split driver για το δίκτυο, μέ ένα driver (netback) να υπάρχει στο driver domain και έναν άλλο driver (netfront) στο παραεικονικοποιημένο λειτουργικό σύστημα, οι οποίοι drivers επικοινωνούν μεταξύ τους μέσω κοινής μνήμης. Προκειμένου λοιπόν, να πετύχει τόσο υψηλή απόδοση το ClickOS, έγιναν σημαντικές αλλαγές τόσο στο netfront και netback driver όσο και σε αλλα κομμάτια του Xen που εμπλέκονται για την εξυπυρέτηση των δικτυακών λειτουργιών όπως το switch [19].

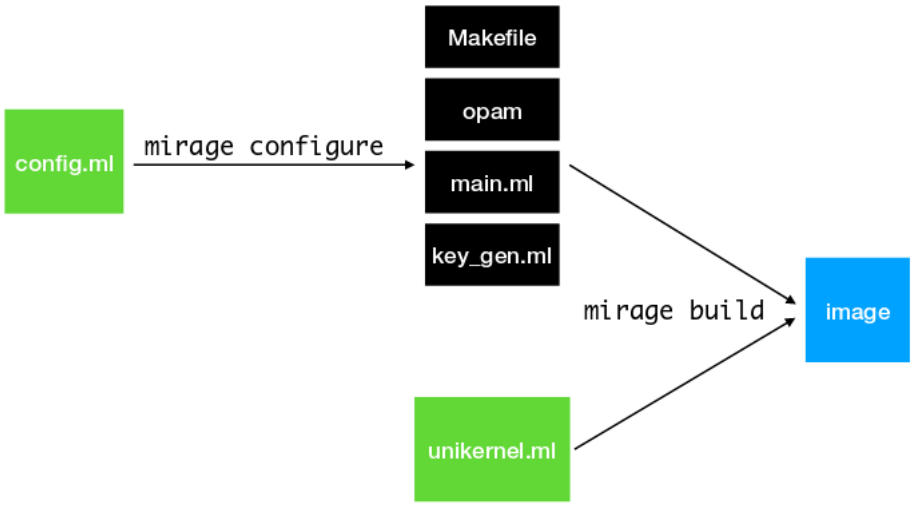
Σχήμα 3.7: Clickos design

3.6 MirageOS

Το MirageOS είναι ένα από τα παλιότερα και πιο διαδεδομένα unikernel frameworks. Σε αντίθεση με όσα είδαμε προηγουμένως, το MirageOS είναι γραμμένο σε μία υψηλού επιπέδου γλώσσα την OCaml. Μερικά από τα βασικά πλεονεκτήματα που προσφέρει η χρήση μία τέτοιας γλώσσας είναι η δημιουργία type safe προγραμμάτων και η μειώση των memory leaks. Η έννοια του type safety αφορά την αποθάρρυνση ή αποτροπή σφαλμάτων τύπων (type errors). Τα σφάλματα τύπων δημιουργούνται όταν υπάρχει ασυμφωνία μεταξύ των τύπων δεδομένων σε μία εντολή και μπορεί να οδηγήσουν σε λανθασμένη ή ανεπιθύμητη συμπεριφορά του προγράμματος. Η OCaml κάνει στατικό έλεγχο τύπων (static type checking) οπότε βρίσκει σφάλματα τύπων κατά τη διάρκεια της μεταγλώττισης και όχι κατά τη διάρκεια της εκτέλεσης. Επιπλέον διαχειρίζεται μόνη της τη μνήμη ενώ ο συλλέκτης σκουπιδιών (garbage collector) έχει σχεδιαστεί με τέτοιο τρόπο ώστε να μειώνονται οι διακοπές.

Η επιλογή της γλώσσας είχε ως συνέπεια και τη δημιουργία από την αρχή πολλών συστατικών ενός λειτουργικού συστήματος. Με αυτό τον τρόπο δημιουργήθηκαν βιβλιοθήκες για το network και storage stack, αλλά και για διάφορα πρωτόκολλα όπως τα TCP/IP, SSH, HTTP, DNS, Openflow, XMPP και Xen inter-VM transports. Γενικότερα η αρχιτεκτονική του MirageOS είναι αρθρωτή και κάθε κομμάτι του είναι σε μορφή βιβιλιοθηκών και κάθε φορά οι απαραίτητες βιβλιοθήες συνδέονται μεταξύ τους για τη δημιουργία μία συγκεκριμένης εφαρμογής (π.χ. μία ιστοσελίδα). Η διαδικασία για τη δημιουργία και εκτέλεση εφαρμογών sto MirageOS είναι αρκετά απλή. Αρχικά, η δημιουργία και ο έλεγχος της εφαρμογής μπορεί να γίνει χρησιμοποιώντας τον mirage compiler, ο οποίος μπορεί να δημιουργήσει εκτελέσιμα τόσο για unix συστήματα όσο και για το MirageOS. Συνεπώς, αρχικά μπορεί η εφαρμογή να εκτελεστεί σε UNIX συστήματα, ώστε να γίνει πιο εύκολα και η αποσφαλμάτωση της και στη συνέχεια να ξαναμεταγλωττιστεί για να αποτελέσει ένα πλήρη unikernel. Η εικόνα αυτή περιέχει την εφαρμογή, το configuration αυτής, τις κατάλληλες βιβλιοθήκες, υποστήριξη για την εκκίνηση του συστήματος

και τον garbage collector. Το MirageOS μπορεί να εκτελεστεί σε Xen, KVM αλλά και σε ARM αρχιτεκτονικές με τη χρήση του Solo5.



Σχήμα 3.8: mirage compiler

Σε μετρήσεις που έχουν πραγματοποιήσει οι δημιουργοί του, δείχνουν ότι παρά την επιλογή μίας υψηλού επιπέδου γλώσσας, το MirageOS παρουσιάζει υψηλή απόδοση τόσο σε θέματα δικτύου όσο και σε μνήμη και χρόνους εκκίνησης [18]. Μάλιστα παρουσιάζει αρκετά καλή συμπεριφορά και στη χρήση περισσότερων του ενός νήματος. Η λειτουργεία αυτή γίνεται εφικτή με τη χρήση μίας ξεχωριστής βιβλιοθήκης η οποία διαχειρίζεται τα νήματα. Επιπλέον, λόγω της γλώσσας που χρησιμοποιείται, σε αντίθεση με τα περισσότερα λειτουργικά συστήματα το μέγεθος του κώδικα είναι αρκετά μικρότερο. Από την άλλη βέβαια η επιλογή της συγκεκριμένης γλώσσας επιτρέπει την υποστήριξη εφαρμογών που είναι γραμμένες σε αυτήν και μόνο. Ωστόσο, οι περισσότερες εφαρμογές που υπάρχουν δεν είναι γραμμένες σε OCaml και συνεπώς η εκτέλεση τους στο MirageOS απαιτεί την ολική επανασχεδίαση τους και μεταφορά τους στην OCaml.

3.7 Solo5

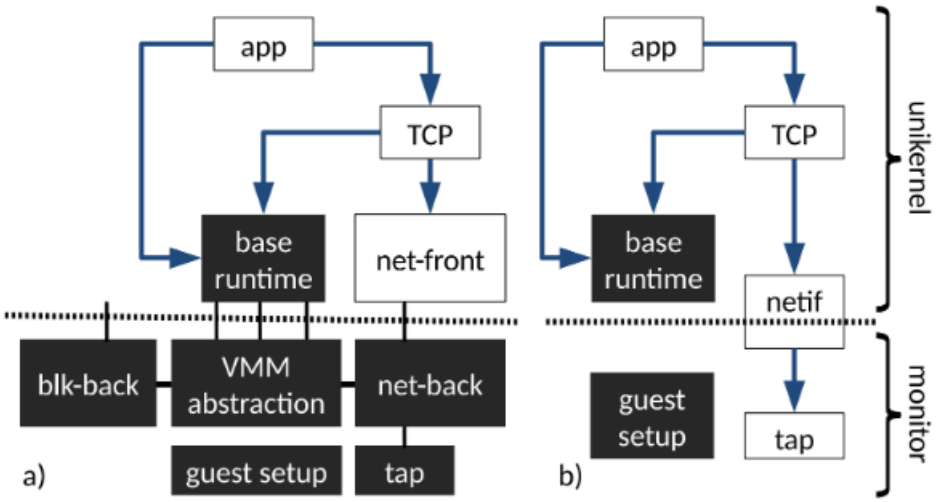
Το Solo5 δεν είναι ένα πλήρες unikernel framework, αλλά μία βάση πάνω στην οποία μπορεί να φτιαχτεί ένα unikernel. Αρχικά σχεδιάστηκε με σκοπό την εκτέλεση του MirageOS στο KVM, αλλά σύντομα εξελίχθηκε σε ένα γενικότερο περιβάλλον εκτέλεσης που προσφέρει απομόνωση (sandbox), κατάλληλο για την εκτέλεση εφαρμο-

γών που έχουν χτιστεί χρησιμοποιώντας διάφορα unikernel frameworks, με στόχο διάφορες τεχνολογίες sandboxing σε διάφορα host λειτουργικά συστήματα και hypervisors [2]. Σκεφτόμενοι τα unikernels ως διεργασίες, αυτό που κάνει το Solo5 είναι να επανασχεδιάζει τη διεπαφή μεταξύ της διεργασίας και του πυρήνα ή του hypervisor [3] με στόχο η νέα διεπαφή να είναι:

- όσο το δυνατόν πιο legacy-free και ”λεπτή” σε σχέση με τις υπάρχουσες υλοποιήσεις. Δηλαδή να εκθέτει όσο το δυνατόν λιγότερα σε σχέση με τον πυρήνα ή με έναν hypervisor.
- να είναι πιο εύκολη η υποστήριξη νέων τεχνολογιών sandboxing, λειτουργικά συστήματα και hypervisors.

Μερικά από τα unikernels που χρησιμοποιούν το Solo5 είναι το MirageOS και το IncludeOS.

Σε συνδυασμό με το ukvm [27] το Solo5 λειτουργεί ως ένα πρόγραμμα φύλακα (tender) αποτελώντας ένα εξειδικευμένο monitor για unikernels. Ο ρόλος αυτός μοιάζει με το ρόλο του QEMU στην περίπτωση του συνδυασμού QEMU/KVM. Ωστόσο το QEMU είναι ένα monitor γενικού σκοπού σε αντίθεση με το Solo5 που είναι μόνο για unikernels. Η λογική πίσω απο αυτό είναι ίδια με αυτή των unikernels, τα οποία αποτελούνται μόνο από τα απαραίτητα στοιχεία που χρειάζεται η εφαρμογή. το Solo5 αντίστοιχα εκθέτει μόνο τα απαραίτητα στοιχεία που χρειάζεται ένα unikernel για να εκτελεστεί. Τα πλεονεκτήματα που προσφέρονται είναι περισσότερη ασφάλεια, καθώς μειώνεται το attack surface αλλά και ταχύτεροι χρόνοι εκκίνησης καθώς δε χρειάζεται να αρχικοποιηθούν συσκευές που δε θα χρειαστούν. Στην παρακάτω εικόνα 3.9 βλέπουμε τις διαφορές μεταξύ ενός ειδικού και γενικού σκοπού monitor. Το Solo5 ως tender μπορεί να εκετελεστεί σε x86_64 και ARm 64-bit (aarch64) αρχιτεκτονικές μαζί με Linux/KVM. Έτσι τα unikernels που χρησιμοποιούν ως βάση το solo5 έχουν τη δυνατότητα να εκτελούνται σε αυτές τις αρχιτεκτονικές.



Σχήμα 3.9: a) γενικού σκοπού monitor b) ειδικού σκοπού monitor

3.8 Mini-OS

Το Mini-OS είναι ένας μικρός πυρήνας που παρέχεται μαζί με το Xen και μπορεί να αποτελέσει τη βάση για τη δημιουργία παραεικονικοποιημένων λειτουργικών συστημάτων. Μία από τις βασικές του χρήσεις είναι η δημιουργία stub domains στο xen. Ένα stub domain είναι ένα ελαφρύ service ή driver domain που έχει ως στόχο να μειώσει το φόρτο εργασίας του QEMU από το dom0 σε ένα διαφορετικό domain. Με αυτό τον τρόπο επιτυγχάνεται καλύτερη απόδοση και περισσότερη ασφάλεια. Μπορεί να εκτελεστεί τόσο σε x86 αρχιτεκτονικές όσο και σε arm. Πολλά από τα γνωστά λειτουργικά συστήματα χρησιμοποιούν το Mini-OS για να μπορούν να είναι είναι συμβατά με το Xen.

Εκτός από λειτουργικά συστήματα, πολλά unikernel frameworks χρησιμοποιούν το Mini-OS ως βάση ώστε να μπορούν να εκτελεστούν στο Xen, όπως το MirageOS, το Rumprun, το ClickOS και άλλα. Ωστόσο μπορεί να χρησιμοποιηθεί και μόνο του για τη δημιουργία unikernels. Διαθέτει TCP/IP stack χρησιμοποιώντας την LWIP βιβλιοθήκη και τη newlib ως βιβλιοθήκη για την C. Επιπλέον διαθέτει υποστήριξη για περισσότερα από ένα νήματα. Από την άλλη, το μικρο εύρος δυνατοτήτων που παρέχει κάνει δύσκολη τη μεταφορά μιας σύνθετης εφαρμογής σε Mini-OS και κατά συνέπεια τη δημιουργία unikernels.

3.9 Και πολλά ακόμα

Εκτός από τα παραπάνω υπάρχουν πολλά ακόμα unikernel frameworks. Μερικά ακολουθώντας το παράδειγμα του MirageOS, χρησιμοποιούν γλώσσες υψηλού επιπέδου όπως το Clive που χρησιμοποιεί την GO, το Runtime.js που χρησιμοποιεί Javascript, το HalVM που χρησιμοποιεί Haskell και το LING που χρησιμοποιεί Erlang. Επιπλέον υπάρχει και το Drawbridge που αποτελεί ένα library operating system βασισμένο στο windows.

Πέρα από τα unikernel frameworks, υπάρχουν και πολλά projects τα οποία έχουν άμμεση σχέση με αυτά. Μερικά τέτοια παραδείγματα είναι το Unik, το οποίο αποτελεί ένα εργαλείο που βοηθάει στη μεταγλώτιση και στην ενορχήστρωση των unikernels. Ουσιαστικά πρόκειται για κάτι ανάλογο με το docker αλλά για unikernels. Ένα πολύ ενδιαφέρον project είναι το Jitsu, ένα forwarding DNS server που εκκινεί αυτόματα unikernels όταν χρειαστεί.

Κεφάλαιο 4

Σχεδιασμός και υλοποίηση

Όπως είδαμε στο προηγούμενο κεφάλαιο κανένα unikernel framework δεν υποστηρίζει την κλήση συστήματος fork και επομένως εφαρμογές που χρησιμοποιούν τη συγκεκριμένη κλήση δεν μπορούν να υποστηριχτούν από αυτά. Στο πλαίσιο, λοιπόν, της παρούσας διπλωματικής εργασίας δημιουργήσαμε ένα μηχανισμό που υλοποιεί τις κλήσεις συστήματος fork και pipe σε QEMU/KVM hypervisor. Ο στόχος ήταν να διατηρηθεί το single process χαρακτηριστικό των unikernels. Για το λόγο αυτό το αποτέλεσμα της κλησης fork, δεν είναι η δημιουργία μία νέας διεργασίας στο υπάρχον unikernel, αλλά η εκκίνηση ενός unikernel ίδιου με το αρχικό.

Επιπλέον, υλοποιήθηκε και ένας inter-vm communication μηχανισμός, στα πρότυπα της κλήσης συστήματος pipe. Δηλαδή δύο εικονικά μηχανήματα μπορούν να επικοινωνούν μεταξύ τους, όπως δύο διεργασίες επικοινωνούν μεταξύ τους, μέσω της κλήσης συστήματος pipe, σε ένα λειτουργικό σύστημα γενικού σκοπού.

Στο παρών κεφάλαιο περιγράφουμε αναλυτικά τους δύο αυτούς μηχανισμούς. Το κεφάλαιο χωρίζεται σε δύο μέρη, ένα για το μηχανισμό pipe και ένα για το μηχανισμό fork. Σε κάθε μέρος περιγράφεται αναλυτικά τόσο ο μηχανισμός, όσο και η διαδικασία και τα στάδια μέχρι την τελική υλοποίηση τους.

4.1 Μηχανισμός pipe

Σύμφωνα με το POSIX ο ορισμός της κλήσης συστήματος pipe έχει ως εξής:

```
int pipe(int fildes[2]);
```

Ο μηχανισμός pipe που υλοποιήθηκε ακολουθεί τον ορισμό του POSIX. Πιο συγκεκριμένα μία κλήση στη συγκεκριμένη συνάρτηση δημιουργεί ένα pipe μεταξύ των δύο εικονικών μηχανημών και δημιουργεί δύο νέους file descriptors. Οι δύο αυτοι file descriptors αποθηκεύονται στις παραμέτρους fildes[0] και fildes[1]. Ο πρώτος file descriptor αφορά το read κομμάτι του pipe, ενώ ο δεύτερος το write κομμάτι του pipe. Η τιμή που επιστρέφει η κλήση συστήματος είναι 0, αν η δημιουργία του pipe ήταν επιτυχής, ενώ διαφορετικά θα επιστραφεί -1 και η μεταβλητή errno περιέχει την τιμή του σφάλματος.

Η ανάγνωση από το pipe γίνεται χρησιμοποιώντας τον πρώτο file descriptor από τις παραμέτρους της κλήσης. Τα δεδομένα διαβάζονται με FIFO (first in first out) σειρά. Ενώ αν δεν υπάρχουν δεδομένα στο pipe τότε η κλήση ”μπλοκάρει” μέχρι να προκύψουν δεδομένα από το write άκρο του pipe. Η εγγραφή στο pipe γίνεται χρησιμοποιώντας το δεύτερο file descriptor από τις παραμέτρους της κλήσης. Η εγγραφή στο pipe μπορεί να μπλοκάρει αν δεν υπάρχει χώρος στο pipe. Επιπλέον μπορεί να αποτύχει αν όλα τα file descriptos για ανάγνωση από το pipe έχουν κλείσει με κωδικό σφάλματος EPIPE.

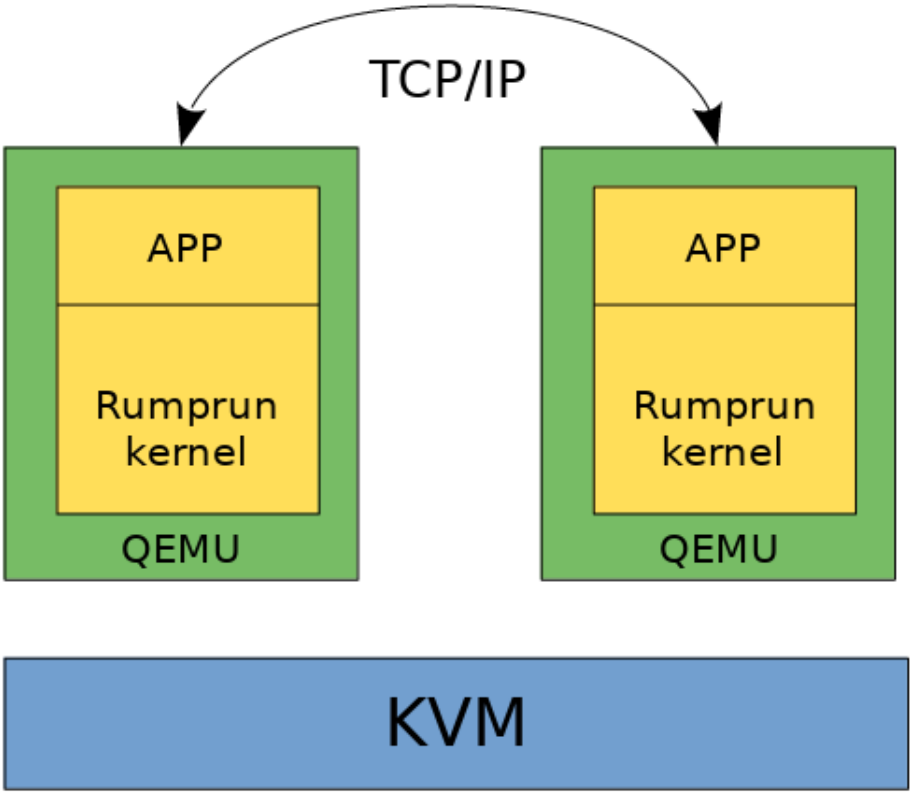
4.1.1 Στάδια υλοποίησης

Η υλοποίηση του συγκεκριμένου μηχανισμού έγινε σε 3 στάδια, τα οποία αναλύονται παρακάτω.

4.1.1.1 Στάδιο 1 - επίπεδο εφαρμογής

Στο πρώτο στάδιο, υλοποιήσαμε το pipe ως μία συνάρτηση που καλείται από την εφαμρογή. Η συνάρτηση αυτή χρησιμοποιεί TCP/IP sockets για να εγκαθιδρύσει την επικοινωνία μεταξύ των δύο εφαρμογών. Συγκεκριμένα χρησιμοποιούνται δύο sockets, ένα για την αποστολή δεδομένων και ένα για την παραλαβή. Οι file descriptors των των δύο αυτών sockets, είναι και οι τιμές που αποθηκεύονται στις μεταβλητές fildes[0] και fildes[1]. Στην παρακάτω εικόνα 4.1 φαίνεται σχηματικά η υλοποίηση.

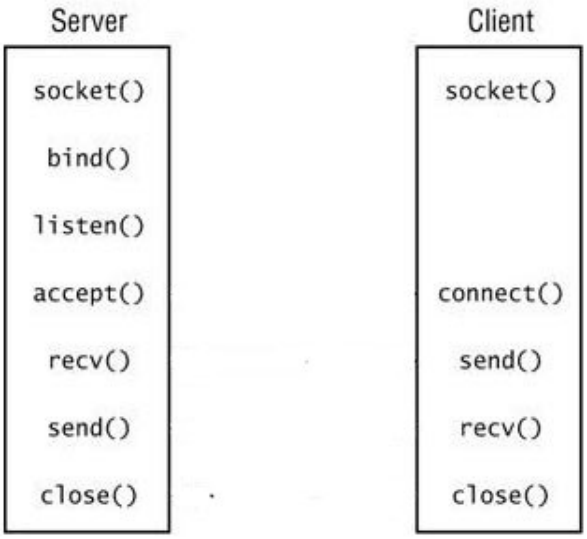
Όπως γίνεται εύκολα κατανοητό η συγκεκριμένη υλοποίηση απαιτεί την ύπαρξη δικτύου ανάμεσα στα δύο εικονικά μηχανήματα. Επιπλέον, λόγω της χρήσης των sockets, δεν υλοποιήθηκαν όλα τα



Σχήμα 4.1: Πρώτο στάδιο υλοποίησης μηχανισμού pipe

semantics του pipe. Έτσι ακόμα και αν δεν υπάρχει ανοιχτό read άκρο στο pipe, οποιαδήποτε εγγραφή στο pipe θα είναι επιτυχής. Ακόμα, οποιαδήποτε εγγραφή μπορεί να μπλοκάρει μόνο αν ο παραλήπτης δεν μπορεί να δεχτεί άλλα δεδομένα. Παρομοίως τα semantics του άκρου ανάγνωσης του pipe διατηρήθηκαν εν μέρει καθώς και στην περίπτωση των sockets, αν δεν υπάρχουν δεδομένα προς ανάγνωση η αντίστοιχη κλήση ”μπλοκάρει”. Ωστόσο, αν δεν υπάρχουν ανοιχτά άκρα εγγραφής στο pipe, η κλήση read θα μπλοκάρει αντί να επιστρέψει EOF.

Για να υλοποιηθεί η συνάρτηση pipe, ακολουθήθηκε η ίδια διαδικασία που ακολουθείται από μία εφαρμογή για να επικοινωνήσει με κάποια άλλη μέσω δικτύου. Η διαφορά είναι ότι η εφαρμογή έχει το ρόλο του server και του client ταυτόχρονα. Για αυτό το λόγο χρησιμοποιήθηκαν δύο sockets, ένα για κάθε ρόλο. Στην παρακάτω εικόνα 4.2 φαίνεται η αλληλουχία κλήσεων συστήματος για την εγκαθίδρυση της επικοινωνίας (server/client).



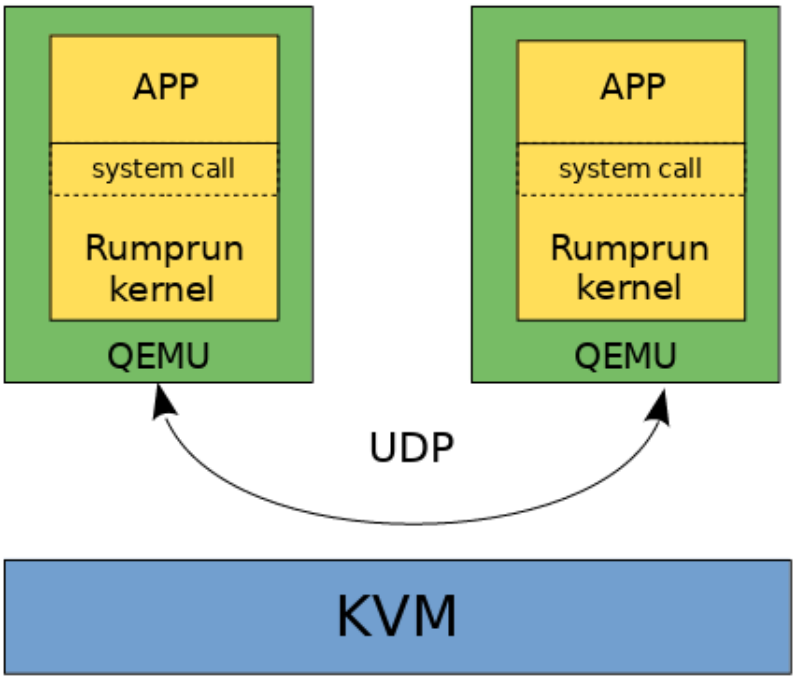
Σχήμα 4.2: Αλληλουχία κλήσεων για επικοινωνία μέσω TCP/IP sockets

Η συνάρτηση pipe, λοιπόν ακολουθεί και τις δύο διαδικασίες που φαίνονται (server/client). Αυτό όμως προξενεί ένα πρόβλημα κατά τη δημιουργία της επικοινωνίας. Αν και οι δύο εφαρμογές ακολουθήσουν την παραπάνω ακολουθία κλήσεων τότε θα δημιουργηθεί αδιέξοδο, καθώς και οι 2 θα έχουν κολλήσει στην κλήση accept, περιμένοντας κάποια σύνδεση. Ο τρόπος με τον οποίο επιλύθηκε το παραπάνω πρόβλημα είναι ως εξής. Μία από τις δύο εφαρμογές να εκτελεί πρώτα τις κλήσεις που αφορούν το client κομμάτι και μετά αυτές που αφορούν το server. Αντίθετα η άλλη εφαρμογή, πρώτα

εκτελεί τις κλήσεις που αφορούν το server και ύστερα αυτές που αφορούν το client μερος.

4.1.1.2 Στάδιο 2 - pipe ως system call με UDP sockets

Στο δεύτερο στάδιο, υλοποιήσαμε το μηχανισμό του pipe μέσα στον πυρήνα του rumprun. Ουσιαστικά μετατρέψαμε το function call του προηγούμενου σταδίου σε system call. Σε αντίθεση με πριν, η επικοινωνία γίνεται μέσω UDP sockets, οπότε και είναι απαραίτητη η ύπαρξη δικτύου. Η επιλογή του πρωτοκόλλου UDP έναντι του TCP, έγινε για την πιο εύκολη δημιουργία και χρήση sockets στον πυρήνα του rumprun. Ο συγκεκριμένος μηχανισμός δίνει τη δυνατότητα να υπάρχει επικοινωνία μεταξύ δύο ξεχωριστών unikernels, ακόμα και αν αυτά δε μοιράζονται τον ίδιο host, μέσω μίας κλήσης συστήματος παρόμοια με την pipe(). Με αυτό τον τρόπο, οι εφαρμογές δε χρειάζονται να τροποποιηθούν σε μεγάλο βαθμό, από την αρχική τους έκδοση. Στην παρακάτω εικόνα 4.3 φαίνεται σχηματικά η υλοποίηση.



Σχήμα 4.3: Δεύτερο στάδιο υλοποίησης μηχανισμού pipe

Όσον αφορά τη χρήση του μηχανισμού, αυτή γίνεται ακολουθώντας την ίδια διαδικασία με τη χρήση της κλήσης pipe σε ένα UNIX σύστημα. Στις παραμέτρους fildes[0], fildes[1], αποθηκεύονται οι δύο file descriptors που θα χρησιμοποιηθούν για την ανάγνωση και εγ-

γραφή αντίστοιχα. Σε περίπτωση επιτυχίας επιστρέφεται η τιμή 0, ενώ σε αντίθετη περίπτωση, στη μεταβλητή errno αποθηκεύεται ο κωδικός του σφάλματος. Δεδομένου, ότι η επικοινωνία γίνεται με UDP sockets, τα semantics είναι ίδια με το προηγούμενο στάδιο. Μία σημαντική διαφορά, είναι ότι πρέπει να καθοριστεί η διεύθυνση IP του unikernel με το οποίο θα εγκαθιδρυθεί η επικοινωνία. Για το λόγο αυτό, μετά την κλήση της pipe() θα πρέπει μέσω της ioctl στο file descriptor της εγγραφής να οριστεί η διεύθυνση IP του παραλήπτη.

```
ioctl(fildes[1], SETIPADDR, htonl(IP_ADDRESS));
```

Η μορφή της IP διεύθυνσης θα πρέπει να είναι σε network byte order και για αυτό το σκοπό μπορεί να χρησιμοποιηθεί η συνάρτηση htonl. Ο τρόπος που λειτουργεί ο μηχανισμός έχει ως εξής:

1. Αρχικά καλώντας την pipe, επιστρέφονται τα δύο απαραίτητα file descriptors, ένα για εγγραφή και ένα για ανάγνωση.
2. Στη συνέχεια, πρεπει να οριστεί η διεύθυνση IP, στην οποία θέλουμε να στείλουμε τα δεδομένα. Αυτό γίνεται μέσω της προαναφερθείσας ioctl κλήσης.
3. Η εφαρμογή στέλνει δεδομένα χρησιμοποιώντας την κλήση write() και το file descriptor που αντιπροσωπεύει το άκρο εγγραφής του pipe.
4. Η εφαρμογή λαμβάνει δεδομένα χρησιμοποιώντας την κλήση read() και το file descriptor που αντιπροσωπεύει το άκρο ανάγνωσης του pipe.

Η μεταφορά του μηχανισμού στον πυρήνα του rumprun απαιτούσε και τις ανάλογες αλλαγές στις συναρτήσεις που χρησιμοποιήθηκαν, καθώς πλέον ο προγραμματισμός γινόταν εντός του πυρήνα. Ο πυρήνας που χρησιμοποιεί το rumprun είναι αυτός του NetBSD και σε σχέση με τον τρόπο που χρησιμοποιούνται τα sockets σε userspace, διαφέρει με τον αντίστοιχο σε kernelspace. Ο συγκεκριμένος μηχανισμός χρησιμοποιεί ένα struct, στο οποίο αποθηκεύονται το socket, το file descriptor και η διεύθυνση IP του παραλήπτη.

```
struct pipe_data {
        struct socket *so;
        uint32_t ip;
        int fd;
};
```

Παρακάτω, περιγράφεται τι γίνεται μέσα στον πυρήνα όταν εκτελούνται οι παραπάνω κλήσεις.

1. pipe: Αρχικά, δημιουργούνται δύο sockets με την socreate, ένα για αποστολή και ένα για παραλαβή δεδομένων και αποθηκεύονται στα αντίστοιχα πεδία του struct pipe_data. Το socket που θα χρησιμοποιηθεί για ανάγνωση γίνεται bind στη θύρα 23456 με την sobind, επιτρέποντας συνδέσεις από κάθε IP. Στη συνέχεια, δημιουργούνται τα 2 file descriptors που θα χρησιμοποιούνται από την εφαρμογή. Τα file descriptors, από τη μεριά του πυρήνα αντιπροσωπεύονται από τις δομές file_t. Στις δομές αυτές αποθηκεύεται και το struct pipe_data που αναφέρθηκε προηγουμένως. Αν όλα έχουν πάει καλά, προστίθονται τα δύο file descriptors στην εφαρμογή και επιστρέφει με επιτυχία η pipe.

2. ioctl: Όταν καλείται η ioctl με το command SETIPADDR, αποθηκεύεται στο πεδίο ip του pipe_data η τιμή που έχει περαστεί ως τρίτη παράμετρος στην ioctl.

3. write: Αρχικά αρχικοποιείται το struct sockaddr_in, το οποίο λαμβάνει την IP του παραλήπτη από το πεδίο ip του pipe_data. Στη συνέχεια χρησιμοποιώντας τη sosend, στέλνονται τα δεδομένα μέσα από το socket.

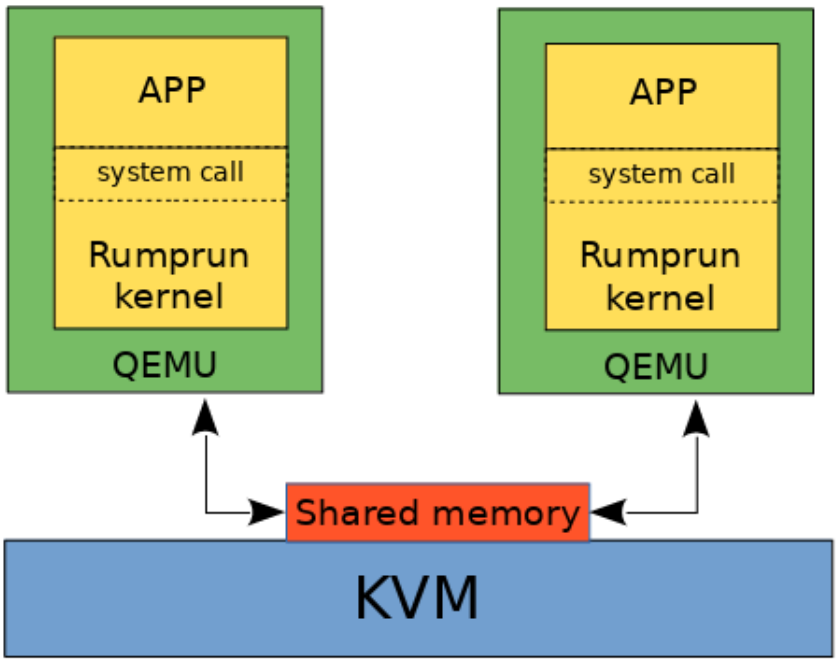
4. read: Με τη χρήση της soreceive, διαβάζονται τυχόν δεδομένα στο socket. Αν δεν υπάρχουν δεδομένα η soreceive μπλοκάρει μέχρι να προκύψουν.

Στον πυρήνα του NetBSD, τυχόν δεδομένα που μεταφέρονται από userspace σε kernelspace αποθηκεύονται στο struct uio. Τόσο η soreceive, όσο και η sosend, δίνουν τη δυνατότητα να χρησιμοποιηθεί το συγκεκριμένο struct και συνεπώς δεν υπάρχει ανάγκη για παραπάνω αντιγραφές των δεδομένων. Η διαδικασία για την εισαγωγή μία νέας κλήσης συστήματος στο rumprun, είναι ακριβώς ίδια με αυτή που θα χρησιμοποιηθεί για την εισαγωγή μία κλήσης συστήματος στον πυρήνα του NetBSD, με ένα επιπλέον βήμα. Αφού εισαχθεί η κλήση συστήματος στο NetBSD, πρέπει να εισαχθεί στο αρχείο src-netbsd/sys/rump/librump/rumpkern η κλήση συστήματος. Σε αυτό το σημείο, καλό είναι να ξανααναφερθεί ότι στα unikernels δεν υπάρχει διαχωρισμός μεταξύ userspace και kernelspace, οπότε οποιαδήποτε system call λειτουργεί ως ένα function call. Ωστόσο οι συγκεκριμένοι όροι χρησιμοποιούνται για να γίνει διαχωρισμός μεταξύ του κώδικα μίας οποιασδήποτε εφαρμογής και του κώδικα του πυρήνα του unikernel.

4.1.1.3 Στάδιο 3 - pipe ως system call με shared memory

Στο τελευταίο στάδιο, υλοποιήσαμε το μηχανισμό pipe, πάλι ως system call με τη διαφορά ότι πλέον δε χρησιμοποιούνται sockets,

αλλά κοινή μνήμη μεταξύ των εικονικών μηχανημάτων. Πλέον δεν υπάρχει ανάγκη για ύπαρξη δικτύου μεταξύ των εικονικών μηχανών, ωστόσο η συγκεκριμένη υλοποίηση μπορεί να χρησιμοποιηθεί μόνο για unikernels, που μοιράζονται τον ίδιο host. Για την κοινή μνήμη χρησιμοποιήθηκε το ivshmem [17]. Όπως έχει ήδη αναφερθεί πρόκειται για ένα μηχανισμό, που επιτρέπει το διαμοιρασμό μνήμης μεταξύ του host και των εικονικών μηχανών που εκτελούνται στο host. Η χρήση του γίνεται μέσω μίας PCI συσκευής. Συνεπώς έπρεπε να δημιουργήσουμε τον κατάλληλο driver που θα μας επιστρέψει να το χρησιμοποιήσουμε. Στην παρακάτω εικόνα 4.4 φαίνεται σχηματικά η υλοποίηση.



Σχήμα 4.4: Τρίτο στάδιο υλοποίησης μηχανισμού pipe

Η χρήση του pipe, από την εφαρμογή γίνεται πλέον όπως σε κάθε UNIX λειτουργικό σύστημα. Καλώντας την pipe, αν όλα πάνε καλά αποθηκεύονται στις παραμέτρους fildes[0], fildes[1], οι δύο file descriptors που θα χρησιμοποιηθούν για την ανάγνωση και εγγραφή αντίστοιχα. Σε περίπτωση επιτυχίας επιστρέφεται η τιμή 0, ενώ σε αντίθετη περίπτωση, στη μεταβλητή errno αποθηκεύεται ο κωδικός του σφάλματος. Επιπλέον έχουν υλοποιηθεί και όσα semantics δεν είχαν υλοποιηθεί στα προηγούμενα στάδια. Οπότε, σε περίπτωση που δεν υπάρχουν ανοιχτά άκρα ανάγνωσης στο pipe, η εγγραφή δε θα είναι επιτυχής. Επίσης, αν δεν υπάρχει αρκετός χώρος για την

εγγραφή δεδομένων, τότε η write ”μπλοκάρει”.

Θα ξεκινήσουμε την περιγραφή της υλοποίησης από τον PCI driver για το ivshmem. Αρχικά πρέπει να ενημερώσουμε το qemu για τη χρήση του συγκεκριμένου μηχανισμού. Για να γίνει αυτό χρησιμοποιούμε τις ακόλουθες παραμέτρους για το qemu.

```
-device ivshmem-plain,memdev=hostmem -object
   memory-backend-file,size=1M,share,
   mem-path=/dev/shm/ivshmem,id=hostmem
```

Στην παράμετρο size, ορίζουμε τη χωρητικότητα του pipe και για κάθε pipe πρέπει να ορίσουμε διαφορετικό mem-path. Κάθε unikernel που ξεκινάει με τις συγκεκριμένες παραμέτρους μπορεί να χρησιμοποιήσει το pipe.

Από τη μεριά του unikernel, χρειάζεται να κατασκευάσουμε έναν PCI driver για το μηχανισμό ivshmem. Όπως και στην περίπτωση του system call έτσι και στην περίπτωση του driver, αρχικά πρέπει να φτιάξουμε το driver για το NetBSD. Ακολουθώντας το Kernel Development Manual του NetBSD [1], για τον pci device driver υλοποιήσαμε τρεις συναρτήσεις:

1. match: Είναι η συνάρτηση που καλείται από το autoconfiguration framework του NetBSD, όταν το σύστημα εκκινεί, ώστε να ταιριαστεί ο driver με τη συσκευή. Οπότε στην περίπτωση του δικού μας driver η συγκεκιμένη συνάρτηση ελέγχει αν πρόκειται για ivshmem pci device, ελέγχοντας τις τιμές του κατασκευαστή και το id της συσκευής.

2. attach: Καλείται μόνο αν η ανίσχνευση της συσκευής ήταν επιτυχής (αν η match επέστρεψε 1). Ο ρόλος της είναι να αρχιοποιήσει τη συσκευή. Δεδομένου, ότι εμείς χρησιμοποιούμε το συγκεκριμένο μηχανισμό απλά για διαμοιρασμό μνήμης, αρκεί να κάνουμε map το PCI_BAR(2) του ivshmem και να αποθηκεύσουμε τις απαραίτητες μεταβλητές για τη χρήση του μηχανισμού.

3. detach: Καλείται σε περίπτωση που η συσκευή αποσυνδεθεί. Στη δική μας περίπτωση είναι αρκετό να κάνουμε unmap το bus_space που χρησιμοποιούμε και να μηδενίσουμε τις μεταβλητές που χρησιμοποιούμε για τη συσκευή.

Προκειμένου να μπορεί να χρησιμοποιηθεί η συσκευή από τον υπόλοιπο πυρήνα του NetBSD, είναι απαραίτητο να αποθηκευτούν οι απαραίτητες μεταβλητές που αναφέρθηκαν στις συναρτήσεις attach και detach. Για το λόγο αυτό δημιουργήθηκε το struct ivshm, που έχει ως πεδία τις μεταβλητές αυτές.

```
struct ivshm {
        bus_size_t              data_s; /* size of shared
           memory */
        bus_addr_t              data_b; /* base address
           of shared memory */
        bus_space_tag_t         data_t; /* bus tag for
           shared memory */
        bus_space_handle_t      data_h; /* bus handle for
           shared memory */
};
```

Ιδανικά θα θέλαμε να κάνουμε map όλη την κοινή μνήμη του ivshmem, ώστε να μπορεί να χρησιμοποιηθεί σαν μνήμη του πυρήνα. Ωστόσο, το rumprun δεν υποστηρίζει το maping μνήμης μέσω του bus. Ως εκ τούτου, η πρόσβαση στην κοινή μνήμη γίνεται κάθε φορά με εγγραφή και ανάγνωση bytes από αυτήν χρησιμοποιώντας το bus. Υπό αυτές τις συνθήκες, υπήρχαν τέσσερις μεταβλητές που ήταν απαραίτητες:

1. το μέγεθος της κοινής μνήμης.
2. η διεύθυνση βάσης της κοινής μνήμης
3. το bus tag της κοινής μνήμης
4. και το bus handle της κοινής μνήμης.

Οι 2 τελευταίες μεταβλητές, είναι αυτές που χρησιμοποιούνται από τις συναρτήσεις που παρέχει το Rumprun για να μπορέσει χρησιμοποιώντας το bus, να επικοινωνήσει με τη συσκευή. Ο ρόλος των υπόλοιπων δύο μεταβλητών θα γίνει πιο ξεκάθαρος παρακάτω.

Αφού λοιπόν, δημιουργήσαμε το driver για το NetBSD και ενημερώσαμε το autoconfiguration framework για την ύπαρξη του, έπρεπε να κάνουμε γνωστή την ύπαρξη του και στο rumprun. Για να γίνει αυτό απαιτούνται δύο ενέργεις. Πρώτον προσθέτουμε το pci driver στο src-netbsd/sys/rump/dev/Makefile.rumpdevcomp. Δεύτερον, να δημιουργηθεί ένα νέο directory στο src-netbsd/sys/rump/dev/lib/ για τη νέα συσκευή. Το directory υατό περιέχει το ioconf και το απαραίτητο Makefile για να προστεθεί ο οδηγός μας στο build σύστημα του rumprun.

```
1 ioconf ivshmem
2
3 include "conf/files"
4 include "dev/pci/files.pci"
5 include "rump/dev/files.rump"
6
7 pseudo-root pci*
8
9 ivshmem* at pci? dev ? function ?
```

Code 4.1: IVSHMEM.ioconf

```
1 RUMPTOP=${TOPRUMP}
2
3 .PATH:  ${RUMPTOP}/../dev/pci
4
5 LIB=    rumpdev_ivshmem
6 COMMENT=ivshmem
7
8 IOCONF= IVSHMEM.ioconf
9 RUMP_COMPONENT=ioconf
10
11 SRCS+=  ivshmem.c
12 ##SRCS+=        ivshmem_component.c
13
14 CPPFLAGS+= -I${RUMPTOP}/librump/rumpkern
15
16 .include "${RUMPTOP}/Makefile.rump"
17 .include <bsd.lib.mk>
18 .include <bsd.klinks.mk>
```

Code 4.2: rumprun Makefile για το ivshmem

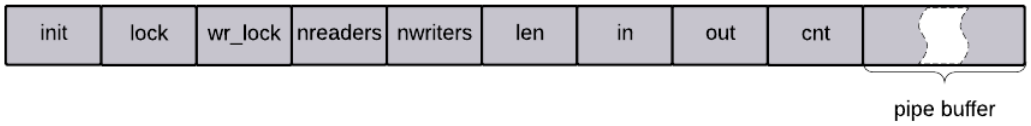
Η αρθρωτή οργάνωση των unikernels, δίνει τη δυνατότητα κάθε φορά να επιλέγονται μόνο τα απαραίτητα συστατικά για κάθε unikernel. Σε αυτό αποσκοπεί και η δημιουργία του παραπάνω directory, ώστε να μπορεί ο συγκεκιμένος driver να ενσωματωθεί στο unikernel, κατά τη διάρκεια του baking. Για τη χρήση του συγκεκριμένου οδηγού είναι απαραίτητη η εισαγωγή του -lrumpdev_ivshmem στο config με το οποίο θα γίνει bake η εφαρμογή.

Αφού δημιουργήσαμε τον driver, το επόμενο βήμα ήταν η αλλαγή του system call, ώστε αντί για sockets να χρησιμοποιείται ο μηχανισμός του ivshmem. Αρχικά, δε χρειαζόμαστε την ioctl κλήση και δημιουργούμε δύο structs, το struct pipe και το struct pipe_op. Οι ορισμοί τους φαίνονται παρακάτω:

```
 1 struct pipe {
 2         bus_size_t      init;           /* is shared
              memory initalized? */
 3         bus_size_t      lock;           /* pipe lock */
 4         bus_size_t      wr_lock;        /* writers lock */
 5         bus_size_t      nreaders;       /* number of
              readers in pipe */
 6         bus_size_t      nwriters;       /* number of
              writers in pipe */
 7         bus_size_t      len;            /* size of pipe
              buffer */
 8         bus_size_t      in;             /* pointer for
              next write */
 9         bus_size_t      out;            /* pointer for
              next read */
10         bus_size_t      cnt;            /* number of
              bytes in pipe */
11         bus_size_t      buf;            /* pipe buffer */
12         int             pr_readers;     /* readers from
              this process */
13         int             pr_writers;     /* writers from
              curr process */
14 };
15
16 struct pipe_op {
17         int             oper;           /* operation in
              pipe 0 for read,
18                                            1 for write*/
19         struct pipe     *pipe;
20 };
```

Το struct pipe_op συνδέεται με τα file descriptors που δημιουργεί η κλήση pipe, ένα για κάθε file descriptor. Η μεταβλητή oper, καθορίζει αν πρόκειται για το file descriptor που αφορά την εγγραφή στο pipe ή αυτό που αφορά στην ανάγνωση. η άλλη τιμή είναι το struct pipe που είναι κοινό και για τα δύο άκρα του pipe και περιέχει βοηθητικές μεταβλητές για τη διαχείρηση του pipe. Όπως έχει ήδη αναφερθεί η χρήση της κοινής μνήμης γίνεται μέσω του bus και οι συναρτήσεις εγγραφής ή ανάγνωσης από το bus απαιτούν ως παράμετρο τη διεύθυνση βάσης από την οποία θα διαβαστούν τα δεδομένα. Έτσι για να αναφερθούμε σε διαφορετική μεταβλητή μέσα σε αυτή τη μνήμη θα πρέπει να χρησιμοποιήσουμε διαφορετική διεύθυνση βάσης. Συνεπώς, το struct pipe περιέχει τις διευθύνσεις των κοινών

μεταβλητών στην κοινή μνήμη. Επιπλέον, χρησιμοποιούνται και οι μεταβλητές pr_readers και pr_writers που αφορούν τα ανοιχτά άκρα στο pipe από τη συγκεκριμένη διεργασία. Οι δύο τελευταίες μεταβλητές είναι απαραίτητες για τη διαχείρηση του pipe όταν η διεργασία που έχει δημιουργήσει το pipe κάνει fork.



Σχήμα 4.5: shared memory layout

Αν θεωρήσουμε την κοινή μνήμη σαν ένα μεγάλο συνεχόμενο κομμάτι μνήμης, τότε στην παραπάνω εικόνα 4.5 φαίνεται πως την έχουμε διαχωρίσει εσωτερικά, με ένα κομμάτι της να αφορά τις μοιραζόμενες μεταβλητές και ένα άλλο τον buffer του pipe. Οι αρχικές μεταβλητές, έχουν μικρότερο μέγεθος από τις τελευταίες και η λειτουργία της κάθε μίας έχει ώς εξής:

- init: Πρόκειται για μία μεταβλητή ελέγχου, που καθορίζει αν έχουν αρχικοποιηθεί οι κοινές μεταβλητές του pipe. Αλλάζει μόνο δύο φορές, όταν πρωτοδημιουργείται το pipe και όταν αυτό καταστρέφεται. Ουσιαστικά δηλώνει αν οι υπόλοιπες μεταβλητές περιέχουν ”σκουπίδια”, ή αν έχουν χρήσιμες τιμές.

- lock: Πρόκειται για το γενικό lock του pipe και προστατεύει όλες τις κοινές μεταβλητές του pipe.

- wr_lock: Πρόκειται για ένα lock που κάθε φορά επιτρέπει σε ένα και μόνο unikernel ννα γράφει στο pipe.

- nreaders: Ο αριθμός των ανοιχτών άκρων ανάγνωσης για το pipe

- wreaders: Ο αριθμός των ανοιχτών άκρων εγγραφής για το pipe

- len: Το μέγεθος του pipe buffer.

- in: Ο δείκτης που καθορίζει σε ποιο σημείο θα γραφτούν τα νέα δεδομένα στο pipe buffer.

- out: Ο δείτης που καθορίζει από ποιο σημείο θα διβαστούν δεδομένα από το pipe buffer

- cnt: Μετρητής των bytes που υπάρχουν στο pipe, κάθε στιγμή

Για την πρόσβαση στις παραπάνω μεταβλητές και γενικά για την κοινή μνήμη χρησιμοποιήθηκαν οι συναρτήσεις bus_space_read_1, bus_space_read_4, bus_space_write_1, bus_space_write_4. Ο αριθμός στο τέλος της κάθε συνάρτησης υποδηλώνει το μέγεθος των δεδομένων που θα διαβαστούν ή θα γραφτούν. Για την πιο εύκολη χρήση της κοινής μνήμης δημιουργήθηκαν 4 βοηθητικές συναρτήσεις, που φαίνονται παρακάτω.

```
 1 /*
 2  * Read a region of bytes in bus (data is 1 byte)
 3  */
 4 void read_region_1(bus_size_t offset, uint8_t *datap,
      bus_size_t count)
 5 {
 6         int i;
 7         for (i=0; i<count; i++) {
 8                 datap[i] =
                      bus_space_read_1(sharme.data_t,
                      sharme.data_h,
 9                                   offset + i);
10         }
11         return;
12 }
13
14 /*
15  * Read a region of bytes in bus (data is 4 byte)
16  */
17 void read_region_4(bus_size_t offset, uint32_t *datap,
      bus_size_t count)
18 {
19         int i;
20         for (i=0; i<count; i++) {
21                 datap[i] =
                      bus_space_read_4(sharme.data_t,
                      sharme.data_h,
22                                   offset + i*4);
23         }
24         return;
25 }
26
27 /*
28  * Write in a region of bytes in bus (data is 1 byte)
29  */
30 void write_region_1(bus_size_t offset, uint8_t *datap,
      bus_size_t count)
```

```
31 {
32         int i;
33         for (i=0; i<count; i++) {
34                 bus_space_write_1(sharme.data_t,
                      sharme.data_h, offset + i,
35                                 datap[i]);
36         }
37         return;
38 }
39
40 /*
41  * Write in a region of bytes in bus (data is 4 byte)
42  */
43 void write_region_4(bus_size_t offset, uint32_t *datap,
       bus_size_t count)
44 {
45         int i;
46         for (i=0; i<count; i++) {
47                 bus_space_write_4(sharme.data_t,
                      sharme.data_h, offset + i*4,
48                                 datap[i]);
49         }
50         return;
51 }
```

Οι παραπάνω συναρτήσεις διαβάζουν ή γράφουν σε μία περιοχή μνήμης, είτε 1 είτε 4 bytes δεδομένα. Το μέγεθος της περιοχής που θα γίνει η πρόσβαση καθορίζεται από την παράμετρο count και το μέγεθος των bytes (count * bytes_of_data). Οι άλλες 2 παράμετροι των συναρτήσεων είναι η διεύθυνση βάσης για το bus και ένας πίνακας , στον οποίο αποθηκεύονται τα δεδομένα που διαβάστηκαν από μνήμη είτε τα δεδομένα που θα γραφτούν στην περίπτωση, στις λειτουργίες read και write αντίστοιχα. Οι συναρτήσεις bus_space, χρησιμοποιούν τη δομή struct ivshm, για τις μεταβλητές bus tag και bus handle. Όλες οι αναφορές στην κοινή μνήμη, πέρα από αυτές των locks, γίνονται χρησιμοποιώντας αυτές τις συναρτήσεις.

Η ύπαρξη κοινής μνήμης μεταξύ διαφορετικών unikernels, απαιτεί και την ύπαρξη κάποιου συγχρονισμού μεταξύ αυτών. Για το σκοπό αυτό χρησιμοποιούνται τα δύο locks στο struct pipe (lock, wr_lock). Τα δύο αυτά locks, βρίσκονται εντός της κοινής μνήμης και απαιτούν ειδική μεταχείριση. Ο μηχανισμός ivshmem μπορεί να υποστηρίξει τη χρήση ατομικών εντολών του gcc. Για το λόγο, αυτό δημιουργήσαμε δύο ακόμα συναρτήσεις για το locking, μία για την απόκτηση του κλειδώματος και μία για την απελευθέρωση του. Οι

συναρτήσεις αυτές φαίνονται παρακάτω.

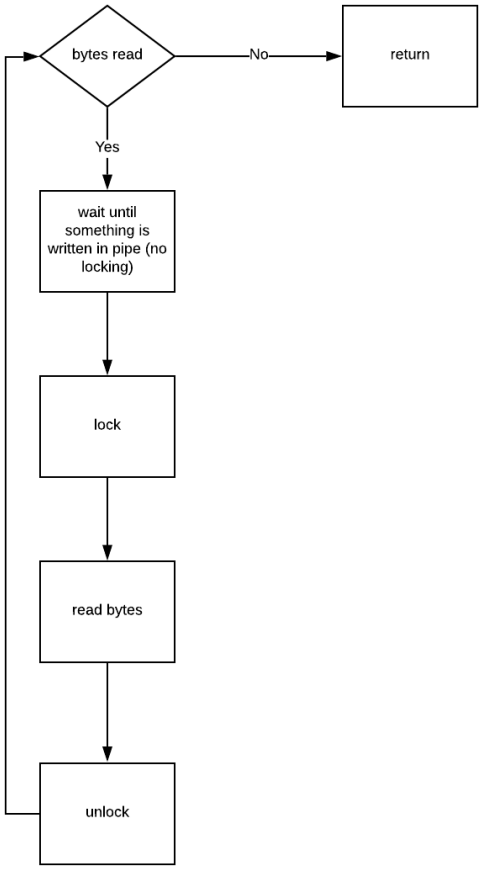
```
1 /*
2  * Spinlock for pipe
3  */
4 void pipe_lock(bus_size_t lock)
5 {
6         while(__sync_val_compare_and_swap((uint8_t
             *)sharme.data_b + lock, 0, 1) == 1)
7                 /* do nothing */;
8         return;
9 }
10
11 /*
12  * Release the lock
13  */
14 void pipe_unlock(bus_size_t lock)
15 {
16         __sync_lock_release((uint8_t *)sharme.data_b +
             lock);
17         return;
18 }
```

Οι ατομικές εντολές που χρησιμοποιήθηκαν για τη χρήση του lock, είναι οι __sync_val_compare_and_swap και __sync_lock_release. Οι εντολές αυτές δέχονται ως παράμετρο τη διεύθυνση της μνήμης στην οποία θα γίνει η πρόσβαση. Η διεύθυνση αυτή καθορίζεται από τη διεύθυνση βασης των δεδομένων (data_b) στο struct ivshmem και το offset του lock μέσα στην κοινή μνήμη που είναι αποθηκευμένο στο struct pipe. Οποιαδήποτε αλλαγή σε κάποιο κομμάτι της κοινής μνήμης απαιτεί την απόκτηση του γενικού lock, ενώ οποιαδήποτε εγγραφή στο pipe θα πρέπει να έχει πάρει τον έλεγχο του wr_lock.

Όταν λοιπόν, η εφαρμογή που εκτελείται σε ένα unikernel εκτελέσει την κλήση pipe, από τη μεριά του πυρήνα πέρα από τη δημιουργία των file descriptors και τη σύνδεση τους με το εκάστοτε struct pipe_op συμβαίνουν τα εξής. Αρχικά, δημιουργείται το struct pipe και αρχικοποιούνται όλες οι μεταβλητές τους με το offset της κάθε μίας στην κοινή μνήμη. Ύστερα, ξεκινά ο έλεγχος της κοινής μνήμης. Ελέγχεται η τιμή της init στην κοινή μνήμη και ανάλογα με την τιμή της η κοινή μνήμη, είτε θα αρχικοποιηθεί είτε όχι. Στη συνέχεια, αφού η περιοχή μνήμης έχει αρχικοποιηθεί, αυξάνονται οι μετρητές των άκρων του pipe και τέλος συνδέεται το κοινό struct pipe με τα struct pipe_op. Η κλήση επιστρέφει τους file descriptors στην εφαρμογή και το pipe είναι πλέον έτοιμο να χρησιμοποιηθεί. Ειδική σημείωση πρέ-

πει να γίνει για τον έλεγχο της τιμής init στην κοινή μνήμη. Η κοινή μνήμη πρέπει να καταστραφεί, αν δεν υπάρχουν pipes που τη χρησιμοποιούν καθώς σε διαφορετική περίπτωση μπορεί λόγω του τρόπου ελέγχου της init, να μην αρχικοποιηθεί.

Σημαντικό ρόλο στον παραπάνω σκοπό έχει η κλήση close σε ένα file descriptor του pipe. Στην απλή περίπτωση που δεν πρόκειται για το τελευταίο file descriptor του pipe, απλά ενημερώνονται οι μετρητές των ανοιχτών άκρων του pipe. Αν όμως, πρόκειται να κλείσει το τελευταίο file descriptor που αφορά το συγκεκριμένο pipe τότε μηδενίζεται όλη η κοινή μνήμη που χρησιμοποιήθηκε. Με αυτό τον τρόπο την επόμενη φορά που θα χρησιμοποιηθεί το συγκεκριμένο κομμάτι μνήμης η τιμή init θα υποδηλώνει ότι πρέπει η κοινή μνήμη να αρχικοποιηθεί.



Σχήμα 4.6: pipe read flow chart

Στην παραπάνω εικόνα 4.6 φαίνεται το διάγραμμα ροής, όταν εκτελείται η κλήση read σε άκρο ανάγνωσης του pipe. Ουσιαστικά πρόκειται για ένα while loop, το οποίο τερματίζει υπό τέσσερις ορι-

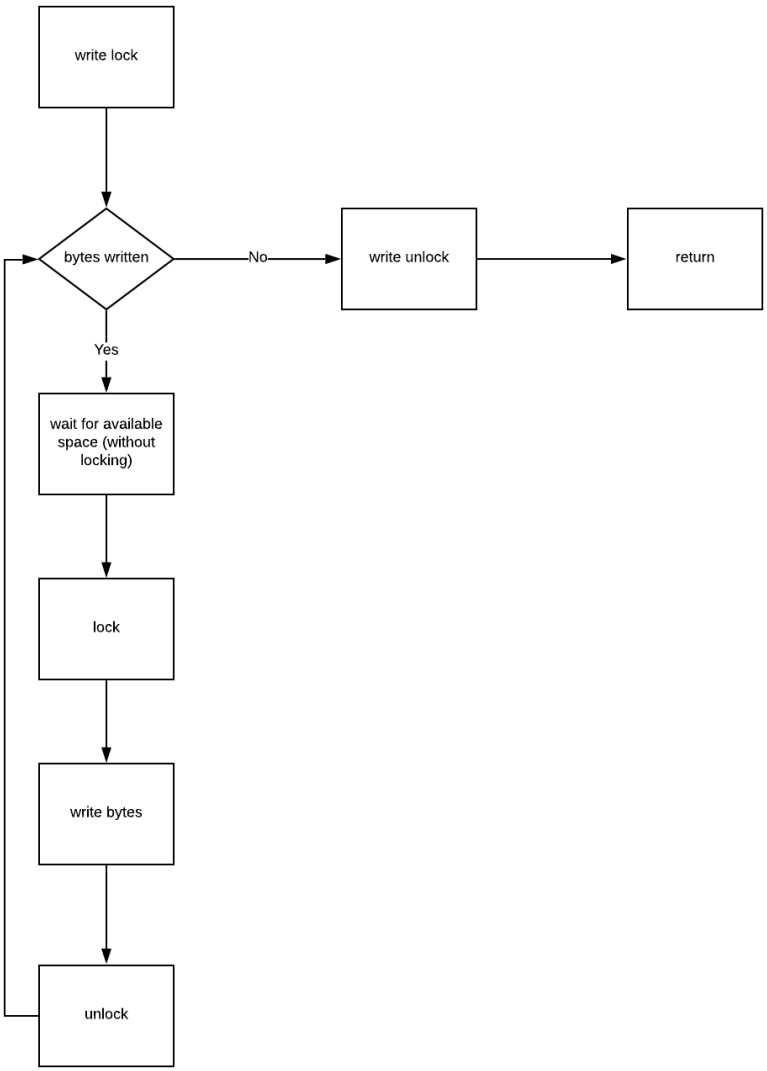
σμένες συνθήκες:

1. Αν έχουν διαβαστεί όσα bytes ζήτησε η εφαρμογή
2. Αν έχουν διαβαστεί όλα τα bytes από το pipe (ανεξάρτητα από τα πόσα ζήτησε η εφαρμογή)
3. Αν δεν υπάρχουν δεδομένα να διαβαστούν και όλα τα άκρα εγγραφής έχουν κλείσει
4. Αν κάτι πάει στραβά κατά τη διαδικασία μεταφοράς δεδομένων από την κοινή μνήμη στην εφαρμογή.

Για την αποφυγή deadlocks, αρχικά γίνεται έλεγχος για τη διαθεσιμότητα δεδομένων στο pipe χωρίς την απόκτηση του γενικού κλειδώματος. Σε περίπτωση, που υπάρχουν δεδομένα, τότε αποκτάται το κλείδωμα και επαναλαμβάνεται ο έλεγχος με την κατοχή του κλειδώματος. Αφού λοιπόν μεταφερθούν τα δεδομένα από την κοινή μνήμη στην εφαρμογή, τότε ενημερώνονται οι κοινές μεταβλητές του pipe (out, cnt) και ελευθερώνεται το lock. Καθώς ο buffer του pipe είναι κυκλικός, πρέπει να λαμβάνουμε υπόψιν την περίπτωση που έχουμε φτάσει στο τέλος του buffer, οπότε πρέπει να ξεκινήσουμε να διαβάζουμε τα υπόλοιπα δεδομένα από την αρχή. Τέλος, τόσο κατά τη διάρκεια του ελέγχου δεδομένων χωρίς αλλά και με το κλείδωμα, ελέγχονται και τα ανοιχτά άκρα εγγραφής στο pipe. Αν όλα έχουν κλείσει τότε πρέπει να επιστρέψει η κλήση pipe καθώς δεν πρόκειται να εγγραφούν νέα δεδομένα.

Στην παραπάνω εικόνα 4.7 φαίνεται το διάγραμμα ροής της κλήσης write του pipe. Όπως φαίνεται δεν έχει κάποια ιδιαίτερη διαφορά με το διάγραμμα ροής της κλήσης read, εκτός από το γεγονός ότι από την αρχή μέχρι το τέλος της εκτέλεσης της, δεσμεύεται το wr_lock. Ο λόγος ύπαρξης ενός τέτοιου κλειδώματος, είναι για να επιτευχθεί η ατομικότητα της κλήσης write σε ένα pipe. Από εκεί και πέρα, όπως και πριν αρχικά ελέγχουμε αν υπάρχει διαθέσιμος χώρος στην κοινή μνήμη για την εισαγωγή νέων δεδομένων. Έπειτα αποκτάται το κλείδωμα και επαναλαμβάνεται ο έλεγχος διαθέσιμου χώρου. Αν είναι επιτυχής, γίνεται η εγγραφή δεδομένων στην κοινή μνήμη και ενημερώνονται οι αντίστοιχες μεταβλητές (in, cnt). Καθώς ο buffer του pipe είναι κυκλικός πρέπει να λαμβάνουμε υπόψιν την περίπτωση που ο διαθέσιμος χώρος βρίσκεται την αρχή του buffer. Τόσο κατά τη διαδικασία ελέγχου διαθέσιμου χώρου χωρίς κλείδωμα όσο και με κλείδωμα, ελέγχονται και τα ανοιχτά άκρας ανάγνωσης του pipe. Αν όλα έχουν κλείσει τότε πρέπει να επιστρέψει η κλήση pipe με το error EPIPE, καθώς δεν υπάρχουν αναγνώστες να διαβάσουν τα δεδομένα. Τέλος, όλα τα παραπάνω βρίσκονται μέσα σε ένα while loop το οποίο μπορεί να τερματίσει υπό τις τρις παρακάτω συνθήκες.

1. Αν έχουν γραφτεί όσα bytes ζήτησε η εφαρμογή



Σχήμα 4.7: pipe write flow chart

2. Αν δεν υπάρχει κανένα ανοιχτό άκρο ανάγνωσης στο pipe.
3. Αν κάτι πάει στραβά κατά τη διαδικασία μεταφοράς δεδομένων από την την εφαρμογή στην κοινή μνήμη.

4.2 Μηχανισμός fork

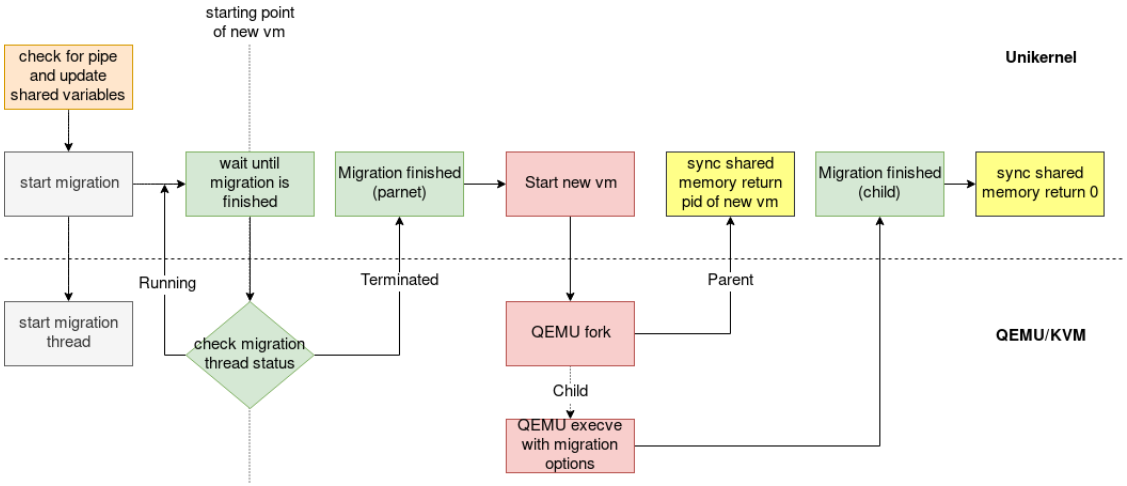
Η κλήση fork δημιουργεί μία νέα διεργασία (παιδί), η οποία είναι όμοια με τη διεργασία που την κάλεσε (γονέας). Και στις δύο διεργασίες ο κώδικας θα συνεχίσει να εκτελείται μετά την κλήση της fork. Στη διεργασία γονέας, η κλήση επιστρέφει το process id του παιδιού, ενώ στη διεργασία παιδί επιστρέφει 0. Αντιθέτως αν έχει υπάρξει κάποιο πρόβλημα κατά την κλήση της τότε επιστρέφεται η τιμή -1 και ο κωδικός του σφάλματος αποθηκεύεται στη μεταβλητή errno. Η διεργασία παιδί υιοθετεί από τη διεργασία γονέα τα ανοιχτά file descriptors. Μία συνηθισμένη πρακτική είναι να δημιουργούνται διεργασίες παιδιά τα οποία εκτελούν συγκεκριμένες ενέργειες και επικοινωνούν με το γονέα μέσω κάποιου μηχανισμού διαδεργασιακής επικοινωνίας όπως για παράδειγμα το pipe. Ένα εξίσου συχνό φαινόμενο είναι οι διεργασίες παιδιά να εκτελούν μία διαφορετική εφαρμογή, κάνοντας χρήση μίας exec κλήσης.

Οι παραπάνω λειτουργίες υλοποιούνται στα περισσότερα συμβατικά λειτουργικά συστήματα αλλά όχι στα unikernel frameworks, μέχρι τη στιγμή εγγραφής αυτής της εργασίας. Εξαίρεση αποτελεί το library operating system Graphene [25] που επιτρέπει την εκτέλεση multi-process εφαρμογών. Τα περισσότερα unikernel frameworks, στοχεύουν στη δημιουργία unikernels που εκτελούν μία και μόνο εφαρμογή, ως μία διεργασία. Προσπαθώντας να κρατήσουμε αυτό το βασικό χαρακτηριστικό τους, σχεδιάσαμε και υλοποιήσαμε ένα μηχανισμό fork, που δίνει τη δυνατότητα σε unikernels να μπορούν να χρησιμοποιήσουν τη συγκεκριμένη κλήση. Η διαφορά είναι ότι η νέα διεργασία δε θα δημιουργηθεί μέσα στο υπάρχον unikernel, αλλά θα δημιουργηθεί ένα νέο unikernel όμοιο με το αρχικό. Αφαιρετικά θεωρούμε κάθε unikernel ως μία διεργασία και το hypervisor, ως το λειτουργικό σύστημα που θα δημιουργήσει και θα διαχειριστεί τις διεργασίες - εικονικές μηχανές. Η υλοποίηση, δεν παρέχει όσες δυνατότητες θα παρείχε ένα συμβατικό λειτουργικό σύστημα, παρά μόνο τη δημιουργία μίας νέας εικονικής μηχανής που μπορεί να επικοινωνεί με το γονέα μέσω του μηχανισμού pipe, ο οποίος αναλύθηκε προηγουμένως. Σε αυτή την περίπτωση, είναι απαραίτητο να να προστεθεί η παράμετρος master=on στο option device του ivshmem-plain.

Η υλοποίηση έγινε για το Rumprun, πάνω από το QEMU/KVM (συγκεκριμένα την έκδοση 2.11.2 https://download.qemu.org/qemu-2.11.2.tar.xz), ωστόσο οποιοδήποτε unikernel framework, μπορεί να εκτελεστεί στο QEMU/KVM μπορεί να χρησιμοποιήσει το συγκεκριμένο μηχανισμό. Σε γενικές γραμμές, όταν μία εφαρμογή κάνει χρήση της κλήσης fork, σε ένα unikernel αυτή η κλήση μετατρέπεται σε hypercalls από τον πυρήνα του Rumprun προς το QEMU/KVM. Στη συνέχεια το QEMU?KVM είναι αυτό που ξεκινά τη διαδικασία δημιουργίας μίας νέας εικονικής μηχανής, η οποία είναι ίδια με την

αρχική (γονέα). Όταν δημιουργηθεί αυτή η εικονική μηχανή, επιστρέφεται ο έλεγχος στην εικονική μηχανή γονέα, η οποία πλέον μπορεί να συνεχίσει την εκτέλεση της.

Στην παρακάτω εικόνα 4.8, φαίνεται όλη η διαδικασία, από τη στιγμή [που μία εφαρμογή καλεί την fork, μέχρι να επιστρέψει ο έλεγχος σε αυτή. Έπειτα ακολουθεί η αναλυτική περιγραφή του μηχανισμού, δίνοντας ιδιαίτερη έμφαση στα σημαντικά σημεία. Η περιγραφή θα ξεκινήσει από την εφαρμογή που καλεί την fork και θα συνεχίζει βήμα βήμα. Ο χρωματισμός προσπαθεί να διευκολύνει την κατηγοριοποίηση της κάθε λειτουργίας στο αντίστοιχο βήμα. Η διαδικασία εισαγωγής κλήσης συστήματος στο Rumprun δε θα αναλυθεί, καθώς αυτό έχει γίνει ήδη στην περίπτωση του μηχανισμού pipe.



Σχήμα 4.8: fork flow chart

4.2.1 Ενημέρωση του pipe

Το πρώτο πράγμα που πρέπει να γίνει από την κλήση συστήματος, είναι ο έλεγχος αν υπάρχει αν η εφαμρογή που καλεί τη fork, χρησιμοποιεί και το μηχανισμό pipe. Όπως έχουμε αναφέρει παραπάνω, αποτελεί συχνό φαινόμενο η δημιουργία ενός pipe πριν την κλήση fork, ώστε να μπορούν οι δύο διεργασίες (γονιός, παιδί) να επικοινωνούν χρημοποιώντας το pipe που χει δημιουργήσει ο γονέας. Δεδομένου ότι πρόκειται για ένα νέο μηχανισμό pipe, θα πρέπει να εκτελεστούν οι ανάλογες ενέργεις για τη σωστή διαχείρηση του. Όπως και σε ένα συμβατικό λειτουργικό συτημα η διεργασία παιδί υιοθετεί τα ανοιχτά file descriptors της διεργασίας γονέας. Στη δικιά μας περίπτωση, η εικονική μηχανή παιδί είναι κλώνος αυτής του γονέα συνεπώς, γνωρίζουμε ότι θα διατηρηθούν τα file descriptors και στο unikernel παιδί. Εν τέλει, απομένει μία μόνο ενέργεια, να

ενημερώσουμε το pipe για την ύπαρξη περισσότερων ανοιχτών και κλειστών άκρων.

Ο τρόπος με τον οποίο ενημερωνόμαστε για την ύπαρξη ανοιχτού pipe, γίνεται από τα file descriptors της εφαρμογής. Ελέγχοντας, λοιπόν, όλα τα file descriptors της εφαρμογής, βρίσκουμε αν υπάρχουν file descriptors για το δικό μας μηχανισμό pipe. Από τη στιγμή που βρεθεί κάποιο, θα πρέπει να ενημερώσουμε το pipe ότι θα αυξηθούν τα ανοιχτά άκρα ανάγνωσης και εγγραφής σε αυτό. Χρησιμοποιώντας τη δομή file_t, στην οποία ο πυρήνας του NetBSD αποθηκεύει τα απαραίτητα στοιχεία του file descriptor, μπορούμε να αποκτήσουμε πρόσβαση στη δομή struct pipe_op και κατ' επέκταση στη δομή struct pipe του μηχανισμού pipe. Η δομή pipe_op, μας πληροφορεί για το αν πρόκειται για άκρο εγγραφής ή για άκρο ανάγνωσης στο pipe. Χρησιμοποιώντας τη δομή pipe, μπορούμε να χρησιμοποιήσουμε την κοινή μνήμη και να ενημερώσουμε τις αντίστοιχες μεταβλητές. Η ενημέρωση γίνεται φυσικά, δεσμεύοντας το γενικό lock του pipe και αυξάνοντας κατά ένα τις μεταβλητές nreaders και nwriters, αν το άκρο ανάγνωσης είναι ανοιχτό ή το άκρο εγγραφής είναι ανοιχτά αντίστοιχα.

Από τη μεριά του pipe δε χρειάζεται να πειράξουμε κάτι άλλο. Η εικονική μηχανή παιδί, όντας κλώνος της εικονικής μηχανής γονέα θα έχει πρόσβαση στην μοιραζόμενη μνήμη. Αν δεν υπάρχει κάποιο pipe από το unikernel γονέα, τότε καμία ενέργεια δε συμβαίνει στο παρών στάδιο.

4.2.2 Hypercall εκκίνησης migration

Από αυτό το στάδιο ξεκινάει η διαδικασία για τη δημιουργία μία εικονικής μηχανής που θα ναι κλώνος με την αρχική. Την κλωνοποίηση και την εκκίνηση της νέας εικονικής μηχανής αναλαμβάνει εξ' ολοκλήρου το QEMU/KVM. Από τη μεριά του το unikernel, περιμένει την ολοκλήρωση αυτής της διαδικασίας, χωρίς να επιστρέφει τον έλεγχο στην εφαρμογή, αλλά παραμένοντας εντός του πυρήνα. Με λίγα λόγια, η διαδικασία κλωνοποίησης της εικονικής μηχανής - γονέα, γίνεται χρησιμοποιώντας τους μηχανισμούς migration που προσφέρει το QEMU.

Με βάση αυτά τα δεδομένα, το πρώτο στάδιο για τη δημιουργία της εικονικής μηχανής - παιδί είναι η δημιουργία του migration file, με βάση το οποίο θα εκκινήσει το QEMU αργότερα, τη νέα εικονική μηχανή. Όλη η επικοινωνία μεταξύ του unikernel και του QEMU, γίνεται μέσω hypercalls. Δεδομένου, ότι απαιτείται η μεταφορά κάποιων δεδομένων από το QEMU στο unikernel, τα hypercalls αυτά είναι στην ουσία vm exits, που γίνονται χρησιμοποιοώντας την εντολή in, της assembly σε κάποιο συγκεκριμένο I/O port.

Για την εκκίνηση του migration το unikernel εκτελεί ένα hypercall,

ζητώντας δεδομένα από το I/O port 0x77dd. Αυτό έχει ως αποτέλεσμα, να γίνει vm exit και ο έλεγχος να βρίσκεται πλέον στην πλευρά του QEMU. Το unikernel αναμένει να λάβει απάντηση από το I/O port, την οποία απάντηση θα δώσει το QEMU. Από τη μεριά του QEMU, αφού διαπιστωθεί ότι πρόκειται για το συγκεκριμένο vm exit, ξεκινάει τη διαδικασία του migration και επιστρέφει την τιμή 0 στο unikernel. Η συγκεκριμένη τιμή απλά υποδηλώνει στο unikernel, ότι έχει ξεκινήσει η διαδικασία δημιουργίας του migration file. Η τοποθεσία του migration file, είναι προκαθορισμένη και είναι η εξής /tmp/vm_migration.out.

Η δημιουργία του migration file, γίνεται με τον ίδιο τρόπο που εκτελεί τη συγκεκριμένη λειτουργία το QEMU, όταν του δοθεί η συγκεκριμένη εντολή. Δηλαδή, χρησιμοποιείται ο ίδιος ο μηχανισμός του QEMU, με μία μικρή αλλαγή. Το QEMU, αφού περατώσει τη διαδικασία του migration, σταματά την αρχική εικονική μηχανή. Από τη δικιά μας μεριά, αυτό δεν είναι επιθυμητό αφού το unikernel γονέας, θέλουμε να μπορεί να συνεχίσει την εκτέλεση του. Συνεπώς, απαιτούνται κάποιες αλλαγές στον κώδικα του migration από το QEMU, ώστε μετά το πέρας του migration, να μην σταματά η εκτέλεση της αρχικής εικονικής μηχανής. Για την υλοποίηση, αυτής της αλλαγής, αφού έχει ολοκληρωθεί η δημιουργία του migration file, πριν επιστρέψει η συνάρτηση που χειρίζεται το migration, καλείται η qmp_cont, η οποία ξαναθέτει σε λειτουργία την εικονική μηχανή. Η συγκεκριμένη συνάρτηση, καλείται από το QEMU, όταν του δοθεί η εντολή να συνεχίσει τη λειτουργία μία εικονικής μηχανής της οποίας η λειτουργία έχει παύσει.

Εν κατακλείδι, η διαδικασία της δημιουργίας του migration file, γίνεται χρησιμοποιώντας τις λειτουργίες που προσφέρει ήδη το QEMU. Ωστόσο, για να μην αλλοιωθεί η κανονική λειτουργία του migration, από την αλλαγή που προσθέσαμε, αντιγράψαμε τον αρχικό κώδικα και τον προσθέσαμε εισάγοντας στο αντίγραφο την παραπάνω αλλαγή. Με αυτό τον τρόπο, διατηρείται η αρχική μορφή του migration από το QEMU και ικανοποιούμε την απαίτηση του μηχανισμού μας, η εικονική μηχανή γονέας να συνεχίζει την εκτέλεση της.

4.2.3 Hypercall ελέγχου migration

Ο τρόπος με τον οποίο εκτελεί το QEMU τη διαδικασία του migration, είναι δημιουργώντας ένα νήμα, που έχει ένα και μόνο ρόλο, να φέρει εις πέρας την όλη διαδικασία του migration. Επιπλέον είναι αναγκαίο η εικονική μηχανή να εκτελείται κατά τη διάρκεια του migration. Προκειμένου να ενημερωθεί ο πυρήνας του Rumprun για το τέλος της διαδικασίας του migration, αλλά και να εκτελείται η εικονική μηχανή, χρησιμοποιούμε ένα δεύτερο hypercall. Όπως και πριν το hypercall υλοποιείται με vm exit για I/O request, αλλά στη

θύρα 0x77db αυτή τη φορά.

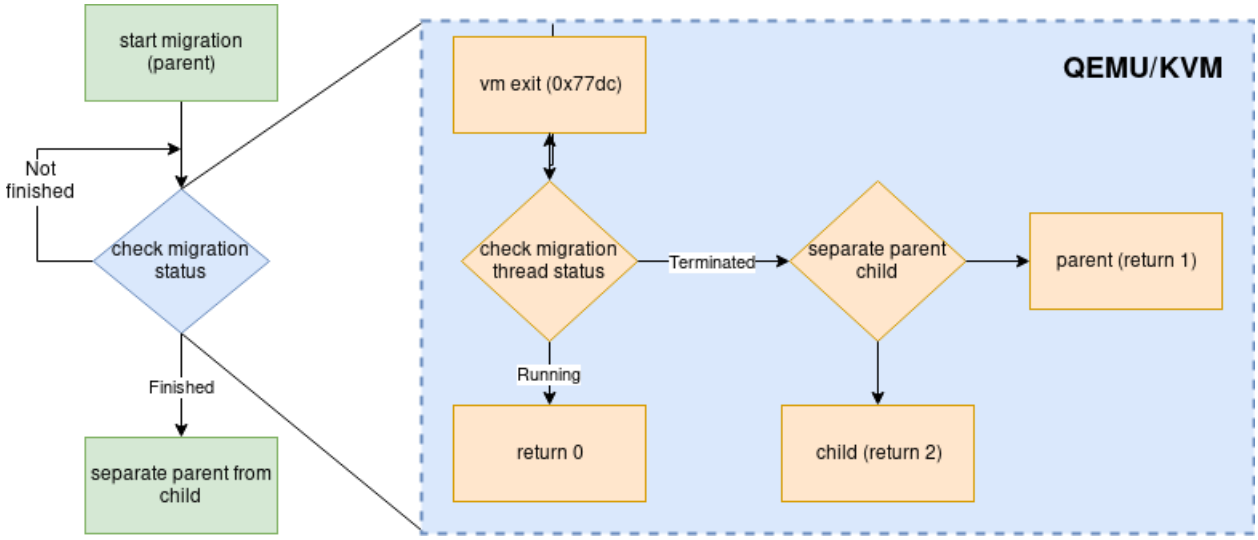
Αφού, λοιπόν, ξεκινήσει η διαδικασία του migration και δοθεί ο έλεγχος στην εικονική μηχανή, εκείνη παραμένοντας μέσα στο system call της fork, περιμένει την ολοκλήρωση του migration. Η αναμονή αυτή έχει υλοποιηθεί με busy wait, χρησιμοποιώντας ένα loop που κάνει συνεχώς hypercalls, για να ελέγξει την κατάσταση του migration. Το loop αυτό διακόπτεται όταν και μόνο επιστραφεί από το QEMU τιμή διαφορετική από το 0, που σημαίνει ότι έχει ολοκληρωθεί το migration. Αυτό το στάδιο χρήζει ιδιαίτερης προσοχής μιας και πρόκειται για το σημείο που θα ξεκινήσει και η νέα εικονική μηχανή. Επειδή, λοιπόν το migration απαιτεί η εικονική μηχανή να εκτελείται, χρειαζόμαστε ένα σταθερό σημείο από το οποίο θα ξέρουμε ότι θα ξεκινήσει και η νέα εικονική μηχανή.

Από τη μεριά του QEMU, μόλις αναγνωριστεί το συγκεκριμένο αίτημα, πρέπει να ελέγξει σε τι φάση βρίσκεται το migration. Αφού η συγκεκιρμένη λειτουργία ελέγχεται από ένα συγκεκριμένο νήμα, μπορούμε απλά να ελέγξουμε αν το νήμα του migration, εκτελείται ακόμα ή όχι. Σε περίπτωση που εκτελείται επιστρέφουμε στην εικονική μηχανή την τιμή 0 και της επιστρέφουμε τον έλεγχο να συνεχίσει την εκτέλεση της. Αν αντιθέτως, το νήμα έχει ολοκληρώσει την εργασία του, επιστρέφουμε την τιμή 1 ή 2, στην εικονική μηχανή και της επιστρέφουμε τον έλεγχο για να συνεχίσει τη λειτουργία της.

Όπως είπαμε και πριν το συγκεκριμένο στάδιο, είναι και το σημείο στο οποίο θα ξεκινήσει την εκτέλεση του η νέα εικονική μηχανή. Συνεπώς, θα πρέπει να διακρίνουμε με κάποιο τρόπο το γονέα από το παιδί. Για το σκοπό αυτό, το QEMU, επιστρέφει την τιμή 1 στο γονέα και την τιμή 2 στο παιδί. Ακόμα δεν έχει δημιουργηθεί το παιδί, αλλά από δω και πέρα ότι κώδικα εκτελέσει ο γονέας θα εκτελεστεί και από το παιδί όταν αυτό δημιουργηθεί. Για να επιστρέχουμε τη σωστή τιμή θα πρέπει το QEMU, να μπορεί να διαχωρίσει, ποιο unikernel έκανε το τελευταίο vm exit ελέγχου του migration.

Για το σκοπό αυτό χρησιμοποιούμε ένα μετρητή για το πόσα hypercalls, έχει κάνει η συγκεκριμένη εικονική μηχανή. Εκμεταλλευόμενοι το γεγονός, ότι κάθε εικονική μηχανή είναι διαφορετική διεργασία, κάθε εικονική μηχανή θα έχει διαφορετικό μετρητή. Επιπλέον, μία σημαντική διαφορά μεταξύ των δύο εικονικών μηχανών, είναι ότι η εικονική μηχανή γονέας που έχει εκκινήσει τη διαδικασία του migration και εκτελεί ένα busy wait, περιμένοντας την ολοκλήρωση του θα κάνει πολλά hypercalls, που θα επιστρέψουν 0, γιατί το migration είναι μία χρονοβόρα διαδικασία. Αντιθέτως η εικονική μηχανή παιδί, όταν θα ελέγξει αν το migration έχει ολοκληρωθεί, αυτό σίγουρα θα έχει ολοκληρωθεί, καθώς διαφορετικά δε θα είχε δημιουργηθεί. Συνεπώς, ο μετρητής των hypercalls του θα είναι σίγουρα 0, οπότε το QEMU, γνωρίζει ότι πρέπει να επιστρέψει την τιμή 2 στην εικονική μηχανή. Το παρακάτω σχήμα 4.9 προσπαθεί να κάνει πιο

κατανοητή την παραπάνω διαδικασία, δείχνοντας τη ροή εκτέλεσης των δύο εικονικών μηχανών.



Σχήμα 4.9: check migration status

4.2.4 Δημιουργία νέας εικονικής μηχανής

Έχουμε φτάσει στο σημείο, που έχει τελειώσει η δημιουργία του migration file από το QEMU και έχει επιστραφεί η τιμή 1 στην εικονική μηχανή αφού πρόκειται για το γονέα. Το επόμενο στάδιο είναι η δημιουργία του νέου unikernel. Από το γονέα unikernel γίνεται άλλο ένα hypercall, αλλά σε διαφορετική θύρα την 0x77dc. Το QEMU λαμβάνοντας το συγκεκριμένο vm exit, ξεκινάει τη διαδικασία για τη δημιουργία της νέας εικονικής μηχανής.

Στην περίπτωση του QEMU/KVM κάθε εικονική μηχανή εκτελείται σε ξεχωριστή διεργασία. Συνεπώς, το πρώτο πράγμα που πρέπει να γίνει είναι η δημιουργία μίας νέας διεργασίας, η οποία προφανώς γίνεται με το system call fork. Στη νέα διεργασία (παιδί) θα εκτελείται το εικονικό μηχάνημα παιδί, ενώ στην προϋπάρχουσα διεργασία θα συνεχίσει να εκτελείται η εικονική μηχανή γονέας.

Συνεπώς, από τη μεριά της η διεργασία πατέρας, επιστρέφει στην εικονική μηχανή το process id της νέας διεργασίας και η εικονική μηχανή μπορεί να συνεχίσει την εκτέλεση της. Από την άλλη η διεργασία παιδί θα πρέπει να ξεκινήσει μία νέα εικονική μηχανή χρησιμοποιώντας το migration file που δημιουργήθηκε στο προηγούμενο στάδιο. Η δημιουργία της νέας εικονικής μηχανής γίνεται χρησιμοποιώντας την κλήση συστήματος execve. Ο μηχανισμός του migration που διαθέτει το QEMU/KVM απαιτεί, η νέα εικονική μηχανή που θα χρησιμοποιεί το migration file, να έχει τις ίδιες ακριβώς παραμέτρους

με τις οποίες ξεκίνησε η αρχική εικονική μηχανή. Συνεπώς, οι παράμετροι της execve, είναι ίδιοι με τις παραμέτρους που ξεκίνησε η αρχική εικονική μηχανή με την προσθήκη των παρακάτω επιπλέον επιλογών, που υποδηλώνουν ότι θα χρησιμοποιηθεί το migration file που δημιουργήθηκε στο προηγούμενο στάδιο.

```
--incoming  exec: cat /tmp/vm_migration.out
```

Ακόμα, προκειμένου να μην υπάρξει πρόβλημα με την έξοδο του νέου και παλιού unikernel, ανακτευθύνουμε το stderr και το stdout, της νέας διεργασίας στο αρχείο ”/tmp/my_server.out”. Προφανώς η παραπάνω ενέργεια πρέπει να προηγηθεί της κλησης execve.

Το νέο unikernel παιδί θα ξεκινήσει την εκτέλεση του από το προηγούμενο στάδιο, δηλαδη θα εκτελέσει ένα hypercall για να ελέγξει αν το migration έχει τελειώσει. Γίνεται εύκολα κατανοητό, ότι ο μετρητής των hypercalls της διεργασίας παιδί θα είναι μηδενικός, συνεπώς μόλις το QEMU αναγνωρίσει ότι πρόκειται για το πρώτο hypercall και ότι δεν εκτελείται το νήμα του migration, θα επιστραφεί η τιμή 2, στην εικονική μηχανή. Πλέον υπάρχουν δύο εικονικές μηχανές που κάθε μία γνωρίζει το ρόλο της (γονέας, παιδί) και μπορούν να συνεχίσουν την κανονική τους λειτουργία.

4.2.5 Συγχρονισμός μοιραζόμενης μνήμης

Στην περίπτωση που χρησιμοποιείται ο μηχανισμός pipe και κατ’ επέκταση το ivshmem, παρατηρήσαμε ότι υπάρχει το εξής πρόβλημα. Αν το unikernel παιδί δε γράψει κάτι στη μοιραζόμενη μνήμη, τότε τα δεδομένα που γράφονται από το unikernel γονιός δεν μπορούν να διαβαστούν από το unikernel παιδί. Συνεπώς, απαιτείται κάποιος συγχρονισμός των δύο unikernels, όσον αφορά τη μοιραζόμενη μνήμη.

Για την αντιμετόπιση του παραπάνω προβλήματος, αφού το unikernel γονιός εκτελέσει το hypercall για τη δημιουργία της νέας εικονικής μηχανής, αν χρησιμοποιείται ο μηχανισμός pipe, εκτελεί ακόμα ένα busy wait. Σε αυτό το σημείο περιμένει να γραφτεί η τιμή 77, σε μία προκαθορισμένη διεύθυνση της μοιραζόμενης μνήμης. Μόλις αυτή η τιμή διαβαστεί, τότε η κλήση fork επιστρέφει το process id της νέας διεργασίας του QEMU και η διαδικασία του fork, έχει ολοκληρωθεί από τη μεριά του γονέα.

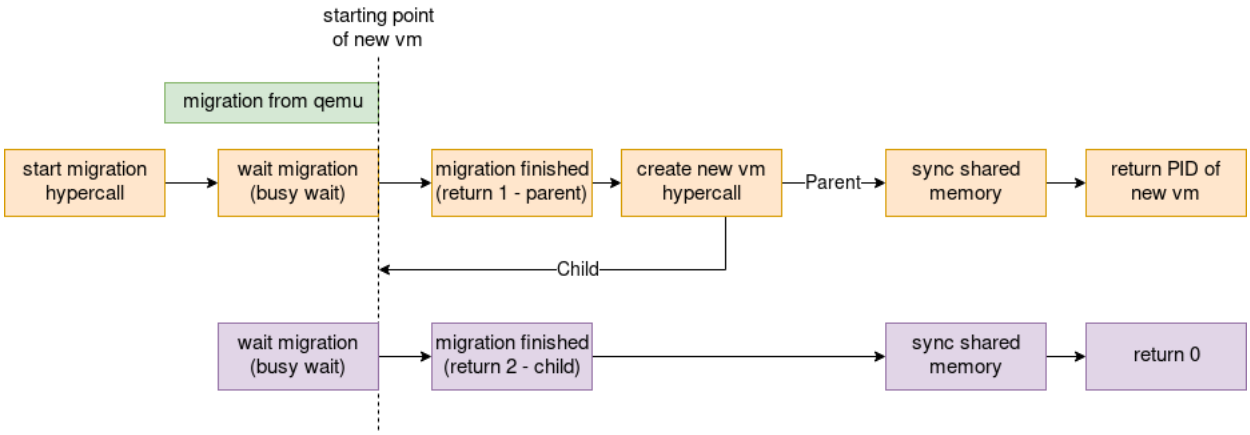
Αντίστοιχα, το unikernel παιδί, αφού δημιουργηθεί και εκτελέσει το hypercall για τον έλεγχο του migration θα πρέπει να γράψει την τιμή 77 στη συγκεκριμένη θέση μνήμης. Προφανώς, η παραπάνω ενέργεια θα εκτελεστεί μόνο στην περίπτωση που χρησιμοποιείται ο

μηχανισμός pipe. Από εκεί και πέρα και το unikernel παιδί, έχει ολοκληρώσει και αυτό τη διαδικασία του fork και επιστρέφοντας την τιμή 0, δίνει τον έλεγχο στην εφαρμογή να συνεχίσει την εκτέλεση της.

4.2.6 Σύνοψη

Στην παρακάτω εικόνα 4.10 φαίνονται με χρονολογική σειρά τα στάδια για την υλοποίηση του μηχανισμού fork. Προφανώς τα βήματα μετά το fork, μπορεί να μην έχουν ακριβώς αυτή τη σειρά, καθώς αυτή εξαρτάται από τη χρονοδρομολόγηση των εικονικών μηχανών και του QEMU. Οι λειτουργίες κάθε unikernel διαχωρίζωνται από τα διαφορετικά χρώματα. Συνοψίζοντας τα βήματα έχουν ως εξής:

1. Το unikernel γονέας, κάνει hypercall στο QEMU για την εκκίνηση της δημιουργίας του migration file
2. Το unikernel γονέας, περιμένει την ολοκλήσρωση του migration. Όταν ολοκληρωθεί, θα του επιστραφεί η τιμή 1.
3. Το unikernel γονέας, κάνει νέο hypercall στο QEMU για τη δημιουργία νέας εικονικής μηχανής.
4. Το qemu κάνει fork. Στο γονέα επιστρέφει το process id της νέας διεργασίας qemu. Η διεργασία παιδί εκτελέι execve για την εκκίνηση της νέας εικονικής μηχανής χρησιμοποιώντας το migration file που δημιουργήθηκε.
5. Το unikernel γονέας, περιμένει να συγχρονίσει την μοιραζόμενη μνήμη, αν υπάρχει.
6. Το unikernel παιδί, ελέγχει αν έχει τελειώσει το migration και του επιστρέφεται η τιμή 2.
7. Το unikernel παιδί συγχρονίζει την κοινή μνήμη.



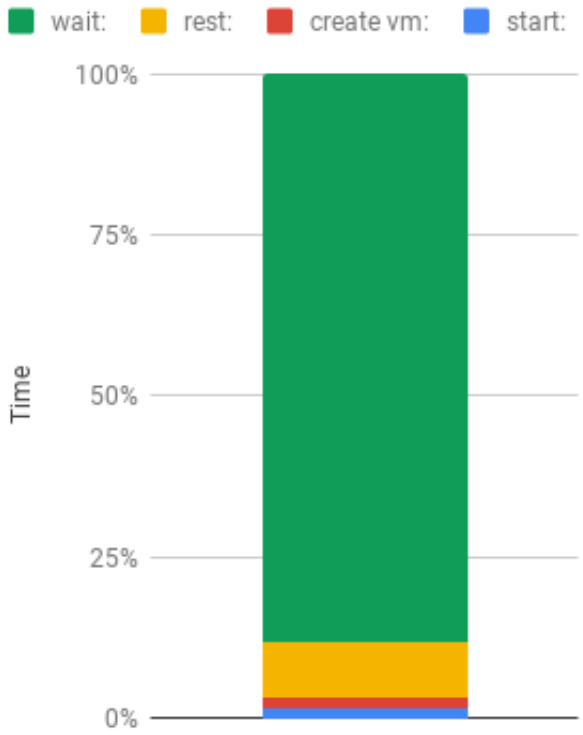
Σχήμα 4.10: fork steps

4.2.7 Αξιολόγηση

Προκειμένου να αξιολογήσουμε τη συγκεκριμένη υλοποίηση, θα συγκρίνουμε τους χρόνους που χρειάζεται η ίδια εφαρμογή για να κάνει fork σε λειτουργικό συτημα linux και στο Rumprun unikernel. Το λειτουργικό συστημα που χρησιμοποιήθηκε ως guest είχε τον εξής πυρήνα Linux 4.20.6-arch1-1-ARCH #1 SMP. Και στις δύο περιπτώσεις, τα λειτουργικά συστήματα εκτελούνται πάνω στην ίδια έκδοση qemu. Το πρόγραμμα που χρησιμοποιήθηκε και στις δύο περιπτώσεις φαίνεται στο τέλος του κεφαλαίου 4.2.7 και δεν κάνει χρήση του μηχανισμού pipe. Οι μετρήσεις έγιναν σε υπολογιστή με λειτουργικό σύστημα Linux 4.9.0-7-amd64 #1 SMP Debian 4.9.110-3+deb9u2, με επεξεργαστή Intel(R) Core(TM)2 Duo CPU P8400. Στην περίπτωση του linux η εφαρμογή για να κάνει fork, χρειάζεται 0.000541s, ενώ στην περίπτωση του rumprun, χρειάζεται 0.137423s.

Παρατηρούμε μία πολύ μεγάλη διαφορά μεταξύ των δύο περιπτώσεων, ωστόσο δεν αντικατοπτρίζει πλήρως την αλήθεια. Αρχικά, στη μία περίπτωση δημιουργείται μία νέα διεργασία ενώ στην άλλη περίπτωση μία ολόκληρη εικονική μηχανή. Επιπλέον, στη δικιά μας υλοποίηση του fork γίνεται πλήρης αντιγραφή του unikernel, σε αντίθεση με την περίπτωση των διεργασιών που έχουν αρκετούς μηχανισμούς (COW, π.χ.) ώστε να μην αντιγράφεται ολόκληρη η διεργασία. Προκειμένου, να δούμε γιατί είναι τόσο αργό το fork, στην περίπτωση του QEMU, μετρήσαμε κάθε ένα από τα βήματα που εκτελούνται από το μηχανισμό. Στο παρακάτω σχήμα 4.12 φαίνεται, το ποσοστό του χρόνου που λαμβάνει κάθε βήμα.

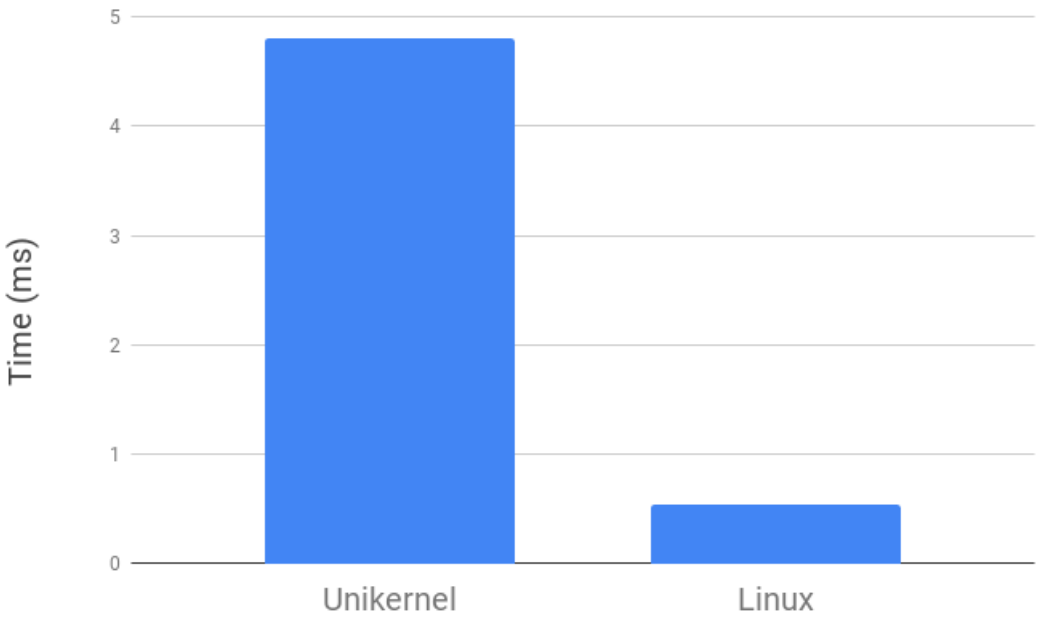
Γίνεται εύκολα εμφανές ότι η πιο χρονοβόρα διαδικασία είναι αυτή της αναμονής για τη δημιουργία του migration file. Το συγκεκριμένο στάδιο λαμβάνει μάλιστα περίπου το 88,191% του συνολικού χρόνου. Αυτός ο χρόνος αντιστοιχεί στο χρόνο που χρειάζεται



Σχήμα 4.11: Time for each step in fork mechanism

το migration thread για να ολοκληρώσει την εργασία του. Σε αυτό το πλαίσιο, μπορούν να εξεταστούν και άλλοι μηχανισμοί για το migration της εικονικής μηχανής. Ωστόσο, στην παρούσα διπλωματική εργασία εξετάζουμε τη δυνατότητα να μπορεί ένα unikernel να εκτελεί την κλήση fork, δημιουργώντας ένα νέο unikernel.

Συνεπώς, τα δύο πιο χρονοβόρα στάδια (δημιουργία migration file και έλεγχοι σφαλμάτων - ορθής λειτουργίας) μπορούν να βελτιστοποιηθούν σε μεγάλο βαθμό. Το μόνο σίγουρο είναι ο χρόνος που χρειάζονται τα δύο hypercals, για την εκκίνηση του migration και την εκκίνηση της εικονικής μηχανής. Αυτοί οι χρόνοι δεν εξαρτώνται από το μηχανισμό και συνεπώς δεν μπορούμε να τους βελτιστοποιήσουμε . Συνεπώς, θεωρούμε ότι πιο δίκαιο θα ήταν να συγκρίνουμε τους δύο αυτούς χρόνους με την κανονική κλήση fork. Με αυτό το σκεπτικό παρουσιάζεται το παρακάτω γράφημα που κάνει ακριβώς αυτή τη σύγκριση.



Σχήμα 4.12: ideal comparison between fork mechanisms

Εκτός από το θέμα των επιδόσεων, υπάρχουν και κάποιοι περιορισμοί που δημιουργούνται, λόγω των χαρακτηριστικών των unikernels. Ο συγκεκριμένος σχεδιασμός δεν καλύπτει όλες τις περιπτώσεις χρήσης της κλήσης fork. Πολλές εφαρμογές χρησιμοποιούν τη fork, για να χρησιμοποιήσουν άλλες εφαρμογές. Σε πολλές περιπτώσεις οι εφαμρογές χρησιμοποιούν εργαλεία που προσφέρει το εκάστοτε σύστημα για να παραμετροποιήσουν κατάλληλα κάποιες λειτουργίες που θα χρησιμοποιήσουν. Για παράδειγμα, πολλές εφαρμογές χρη-

σιμοποιούν την εντολή ifconfig, για να διαμορφώσουν τις δικτυακές επαφές του συστήματος. Γενικότερα πολλές εφαρμογές όταν χρησιμοποιούν τη fork, είναι σχεδιασμένες με γνώμονα ότι η διεργασία παιδί θα εκτελεστεί στο ίδιο λειτουργικό σύστημα.

Από την άλλη μεριά με το συγκεκριμένο μηχανισμό δίνεται η δυνατότητα στις εφαρμογές να μπορούν κάνουν autoscale, χωρίς να χρειάζονται εξωτερική βοήθεια. Άλλωστε αυτό είναι ένα ακόμη use case της κλήσης fork. Πολλές εφαρμογές όταν έχουν μεγάλο φόρτο εργασίας, δημιουργούν νέες διεργασίες για να μοιράσουν αυτό το φόρτο. Ένα τυπικό παράδειγμα θα μπορούσε να είναι ένας web server, ο οποίος για κάθε σύνδεση θα μπορεί να δημιουργεί ένα νέο unikernel, αποκλειστικά για τη συγκεκριμένη σύνδεση, έχοντας την ασφάλεια που προσφέρει το isolation.

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#include <sys/time.h>
#define USEC            1000000

int main()
{
        int fd[2], n;
        struct timeval t1, t2;
        gettimeofday(&t1, 0);
        n = fork();
        if (n == 0) {
                /* child */
                gettimeofday(&t2, 0);
                printf("child: fork took: %lfs\n",
                   (double)((t2.tv_sec - t1.tv_sec) *
                   USEC + t2.tv_usec - t1.tv_usec) /
                   USEC);
        } else if (n < 0) {
                perror("fork");
                exit(1);
        } else {
                gettimeofday(&t2, 0);
                printf("parent: fork took: %lfs\n",
                   (double)((t2.tv_sec - t1.tv_sec) *
                   USEC + t2.tv_usec - t1.tv_usec) /
                   USEC);
        }
        return 0;
}
```

Κεφάλαιο 5

Επίλογος

Το cloud computing, είτε αυτό αφορά σε κατανεμημένα - γεωγραφικά - περιβάλλοντα (grid computing) είτε αφορά υπολογιστικά κέντρα με συστοιχίες (clusters) υπολογιστών, βρίσκεται αναπόφευκτα στο επίκεντρο του ενδιαφέροντος στις μέρες μας. Η εικονικοποίηση, ως η βασική τεχνολογία πίσω από το cloud computing, δίνει τη δυνατότητα να αξιοποιήσουμε όσο το δυνατόν καλύτερα τους πόρους του συστήματος και επιτρέπει τη μεταφορά ενός ολόκληρου συστήματος από ένα φυσικό υπολογιστικό σύστημα σε ένα άλλο. Χάρις αυτές τις τεχνολογίες, πετυχαίνουμε την εκτέλεση περισσότερων εφαρμογών και με περισσότερη ασφάλεια στο ίδιο υπολογιστικό σύστημα. Σε αυτό το πλαίσιο αναπτύχθηκαν νέες τεχνολογίες εικονικοποίησης, όπως η εικονικοποίηση σε επίπεδο λειτουργικού συτήματος (containers), που αφαιρούν το κόστος της εκτέλεσης ενός hypervisor, αλλά μειώνοντας την απομόνωση μεταξύ των containers.

Ακολουθώντας μία διαφορετική προσέγγιση, που θέλει το λειτουργικό σύστημα να είναι tailored στις ανάγκες μίας εφαρμογής και όχι το ανάποδο, μπορούμε να επανασχεδιάσουμε τα περιβάλλοντα εκτέλεσης των εφαρμογών ώστε να περιλαμβάνουν μόνο ό,τι πραγματικά χρειάζεται η εκτέλεσή της πάνω από το υλικό. Η προσέγγιση αυτή καλύπτεται από τους unikernels, οι οποίοι αποτελούν ανεξάρτητες εικόνες μηχανές εννιαίου χώρου διευθύνσεων ικανές να εκτελεστούν αυτόνομα, περιλαμβάνοντας όλα και μόνο τα στοιχεία που χρειάζονται για να το πετύχουν αυτό. Ακόμη και σε αυτή τη προσέγγιση υπάρχουν διαφορετικές οπτικές. Η επαναδιαμόρφωση του νεφοϋπολογιστικού περιβάλλοντος εκτέλεσης της εφαρμογής μπορεί είτε να σημαίνει την εξαρχής επανασχεδίαση και ανάπτυξη των στοιχείων που παραδοσιακά παρείχε το λειτουργικό σύστημα είτε την επαναχρησιμοποίηση ήδη υπαρκτών.

Η δεύτερη περίπτωση φέρει το πλεονέκτημα του ευκολότερου porting των υπαρκτών εφαρμογών που ακολουθούν τα παραδοσιακά πρότυπα (POSIX) καθώς και την προφανή ανάγκη επαναπρογραμματισμού των βιβλιοθηκών και των drivers. Ωστόσο ο κατακερματισμός

ενός λειτουργικού συστήματος σε αυτόνομες επαναχρησιμοποιούμενες μονάδες δεν είναι εύκολη υπόθεση, ιδιαίτερα αν αναλογιστούμε ότι κατά τον αρχικό σχεδιασμό τους δεν υπήρχε μάλλον η ανάγκη και το κριτήριο της αποφυγής (κατά το δυνατόν) των πολλαπλών αλληλοεξαρτήσεων. Ακόμα όμως και σε αυτό το πλαίσιο το porting των εφαμρογών σε unikernels, δεν αποτελεί εύκολη διαδικασία, καθώς πολλές λειτουργίες που προσφέρονται από τα συμβατικά λειτουργικά συστήματα δεν υπάρχουν στα unikernel frameworks. Χαρακτηριστικό παράδειγμα αποτελούν οι multi-process εφαρμογές, οι οποίες δεν μπορούν να εκτελεστούν σε κάποιο unikernel, καθώς κανένα δεν υποστηρίζει περισσότερες από μία διεργασίες.

Θέλοντας να κάνουμε τα unikernel frameworks, περισσότερο φιλόξενα για τις ήδη υπάρχουσες εφαρμογές, σχεδιάσαμε και υλοποιήσαμε δύο μηχανσιμούς. Έναν για την επικοινωνία μεταξύ unikernels στα πλαίσια της κλήσης συστήματος pipe και ένα για τη δημιουργία νέων unikernels, όπως δημιουργούνται νέες διεργασίες με την κλήση fork. Σκεφτόμενοι τα unikernels ως διεργασίες και το hypervisor ως λειτουργικό σύτημα, δημιουργήσαμε τους δύο παραπάνω μηχανισμούς, με στόχο να μοιάζουν με τους αντίστοιχους μηχανισμούς στα συμβατικά λειτουργικά συστήματα.

Αρχικά παρουσιάστηκε στον αναγνώστη το θεωρητικό υπόβαθρο που ήταν απαραίτητο για την κατανόηση των εννοιών που χρησιμοποιήθηκαν, αλλά και για την κατανόηση της σχεδίασης και υλοποίησης των μηχανισμών. Στη συνέχεια, έγινε μία σύντομη περιγραφή των unikernel frameworks που μελετήθηκαν σε αυτή την εργασία και παρουσιάζονται κάποια κύρια χαρακτηριστικά τους. Έπειτα, παρουσιάσαμε το σχεδιασμό των δύο μηχανισμών που υλοποιήσαμε.

Αν και υπάρχουν library operating systems, που μπορούν να υποστηρίξουν multi-process εφαρμογές, όπως το Graphene [25] δεν υπήρχε κάποιος ανάλογος μηχανισμός, που μεταχειρίζεται τα unikernels ως διεργασίες. Ωστόσο κατά τη διάρκεια της εκπόνησης της εργασίας, παρουσιάστηκε το Kylinx [29], που εισάγει την έννοια του pVM (process-like VM). Το συγκεκριμένο λειτουργικό συτημα τρέχει μόνο πάνω από Xen hypervisor και πέρα από την υποστήριξη της fork, παρέχει και ένα inter-pVM communication API, που περιλαμβάνει τους μηχανισμούς pipe, signal, message queue και shared memory. Επιπροσθέτως η αντιμετώπιση των unikernels ως διεργασίες παρουσιάστηκε και σε ένα νέο paper το Νοέμβριο του 2018 [28].

5.1 Μελλοντικές κατευθύνσεις

Ξεκινώνοντας από το μηχανισμό pipe, θα μπορούσαμε να τον επεκτείνουμε ώστε να μπορεί να χρησιμοποιεί TCP sockets αντί γοα UDP, στο δεύτερο στάδιο υλοποίησης του UDP. Επιπλέον, όσον αφορά το τρίτο στάδιο υλοποίησης, ο μηχανισμός ivshmem παρέχει υποστήριξη για την αποστολή σημάτων μεταξύ των εικονικών μηχανών. Αυτό το χαρακτηριστικό θα μπορούσε να χρησιμοποιηθεί για την καλύτερη υλοποίηση του μηχανισμού, ώστε να μη βασίζεται τόσο στην κοινή μνήμη. Επιπλέον το συγκεκριμένο χαρακτηριστικό μπορεί να βοηθήσει στην υλοποίηση ενός μηχανσιμού signaling, παρόμοιο με αυτό που παρέχουν τα συμβατικά λειτουργικά συστήματα. Γενικότερα, θα μπορούσαμε να μεταφέρουμε τους περισσότερους inter-process μηχανισμός επικοινωνίας σε επίπεδο unikernels. Τέλος αρκετά ενδιαφέρον θα ήταν να συγκρίνουμε το συγκεκριμένο μηχανισμό και να τον βελτιστοποιήσουμε, ώστε να πλησιάζει, όσο γίνεται, στον κλασσικό μηχανισμό pipe.

Από τη μεριά του μηχανισμού fork, θεωρούμε ότι χωράει αρκετές βελτιστοποιήσεις. Ιδανικά θα θέλαμε ο χρόνος που απαιτείται, να πλησιάζει όσο το δυνατόν περισσότερο σε αυτόν του fork σε ένα συμβατικό λειτουργικό σύστημα. Η ύπαρξη δύο busy waits στην υλοποίηση δημιουργεί αυτόματα πεδίο για εξερεύνηση εναλλακτικών επιλογών που θα απαλείφουν την καθυστέρηση αυτή. Επιπλέον, η χρήση του μηχανισμού migration του QEMU, ίσως να μπορεί να αντικατασταθεί από ένα ταχύτερο και με περισσότερες λειτουργίες μηχανισμό. Τέλος, ο συγκεκριμένος μηχανισμός είναι υλοποιημένος για το QEMU και θα μπορούσε να υλοποιηθεί και σε άλλους hypervisors, ή ακόμα και σε tenders όπως το solo5.

Βιβλιογραφία

[1] Netbsd kernel developer's manual driver(9). http://netbsd.gw.com/cgi-bin/man-cgi?driver+9+NetBSD-current, Last accessed on 08-01-2019.

[2] Solo5 on github. https://github.com/Solo5/solo5, Last accessed on 08-01-2019.

[3] Solo5 technical overview. https://github.com/Solo5/solo5/blob/master/docs/architecture.md, Last accessed on 08-01-2019.

[4] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGARCH Computer Architecture News*, 34(5):2–13, 2006.

[5] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference*, pages 373–385, 2012.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[7] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 250–257. IEEE, 2015.

[8] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[9] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. Containerleaks: emerging security threats of information leakages in container clouds. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 237–248. IEEE, 2017.

[10] Major Hayden and Richard Carbone. Securing linux containers. *GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License*, 19, 2015.

[11] Van Jacobson and Bob Felderman. Speeding up networking. In *Linux Conference Au, http://www. lemis. com/grog/Documentation/vj/lca06vj. pdf*, 2006.

[12] Antti Kantee. *The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, doctoral dissertation. Department of Computer Science and Engineering, Aalto University, 2012.

[13] Antti Kantee. On rump kernels and the rumprun unikernel, 2015. `https://blog.xenproject.org/2015/08/06/on-rump-kernels-and-the-rumprun-unikernel/`, Last accessed on 08-01-2019.

[14] Antti Kantee and Justin Cormack. Rump kernels no os? no problem! *USENIX; login: magazine*, 2014.

[15] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. Os v—optimizing the operating system for virtual machines. In *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*, page 61, 2014.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[17] A Cameron Macdonell et al. *Shared-memory optimizations for virtual machines*. University of Alberta Edmonton, Canada, 2011.

[18] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Acm Sigplan Notices*, volume 48, pages 461–472. ACM, 2013.

[19] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[20] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[21] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[22] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library os from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304. ACM, 2011.

[23] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. Lkl: The linux kernel library. In *Roedunet International Conference (RoEduNet), 2010 9th*, pages 328–333. IEEE, 2010.

[24] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. Shared-memory optimizations for inter-virtual-machine communication. *ACM Computing Surveys (CSUR)*, 48(4):49, 2016.

[25] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, page 9. ACM, 2014.

[26] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium*, pages 43–56, 2004.

[27] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *HotCloud*, 2016.

[28] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 199–211. ACM, 2018.

[29] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx: a dynamic library operating system for simplified and efficient cloud virtualization. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 173–186, 2018.