ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδιασμός και υλοποίηση fork σε rumprun unikernels

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΞΥΠΟΛΙΤΟΥ
ΓΕΩΡΓΙΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Σχεδιασμός και υλοποίηση fork σε rumprun unikernels

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Ξυπόλιτου Γεώργιου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 13 Ιουλίου, 2021.

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2021.

(Υπογραφή)

.....

ΓΕΩΡΓΙΟΣ ΞΥΠΟΛΙΤΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η εικονοποίηση είναι μια επίμαχη τεχνολογία της σύγχρονης εποχής, καθώς υποστηρίζει μεγάλο μέρος του cloud στο οποίο βασίζεται μεγάλο ποσοστό των υπηρεσιών στο internet. Με την τεχνολογία αυτή γίνεται δυνατό ένας υπολογιστής να φιλοξενεί πολλαπλά εικονικά μηχανήματα. Η εικονοποίηση, όμως, μιας ολόκληρης εικονικής μηχανής καταναναλώνει αρκετούς υπολογιστικούς πόρους. Μία τεχνική αντιμετώπισης αυτού του προβλήματος είναι τα containers. Αυτά βελτιώνουν πολύ την χρήση των πόρων του φιλοξενιτή, αφού πλέον οι εφαρμογές χωρίζονται σε ξεχωριστά userspaces χρησιμοποιώντας από κοινού τις λειτουργίες του υποβόσκοντος λειτουργικού συστήματος. Παρότι αυτή η τεχνική μείωσε την κατανάλωση πόρων, η έλλειψη απομόνωσης μεταξύ των containers οδήγησε στην χρήση τους μέσα σε εικονικές μηχανές που αναιρούσε το αρχικό πλεονέκτημα του σχεδιασμού τους. Σε αυτό το πρόβλημα βασίστηκε η προσέγγιση των unikernels, εικονικές μηχανές σε μορφή αυτοτελών εικονών μιας εφαρμογής. Δηλαδή, η εικόνα περιέχει τόσο την εφαρμογή, αλλά και ακριβώς τα κομμάτια του λειτουργικού συστήματος που θα χρησιμοποιήσει κατά την εκτέλεση της. Όλα τα κομμάτια ενός unikernel τρέχουν σε ένα μοναδικό χώρο διευθύνσεων σε αντίθεση με τα συμβατικά μονολιθικά συστήματα, δηλαδή οι λειτουργίες του πυρήνα πλέον εκτελούνται στον ίδιο χώρο διευθύνσεων με την εφαρμογή. Οι unikernel εικόνες μπορούν να τρέξουν είτε σε ένα μηχάνημα φιλοξενιτή επικοινωνόντας με αυτό μέσω ενός υπερεπόπτη, είτε απευθείας πάνω στο υλικό. Έτσι παραμένουν απομονωμένες τόσο οι εικονικές μηχανές μεταξύ τους, αλλά και με τον φιλοξενιτή. Με αυτό τον τρόπο μειώθηκαν οι ανάγκες σε πόρους για την εκτέλεση εικονικών μηχανών, διατηρώντας την απομόνωση που προσφέρουν.

Σημαντικό όμως μειονέκτημα των unikernels, είναι η έλλειψη υποστήριξης κλήσεων συστήματος και λειτουργικοτήτων που υποστηρίζονται στα λειτουργικά συστήματα που χρησιμοποιούνται στο cloud και λαμβάνονται ως δεδομένα κατά τον σχεδιασμό των εφαρμογών. Σκοπός αυτής της διπλωματικής είναι η υλοποίηση της fork λειτουργικότητας, ώστε πολυδιεργασιακές εφαρμογές που χρησιμοποιούν την κλήση συστήματος να μπορούν να τρέξουν σε Rumprun unikernels.

Λέξεις-Κλειδιά: εικονοποίηση, rumprun, unikernel, fork, πολυδιεργασιακές εφαρμογές, solo5

Περιεχόμενα

Λίστα Προγραμμάτων

Κεφάλαιο 1

Εισαγωγή

Στην εποχή μας μεγάλο μέρος των υπηρεσίων που κάποτε πρόσφερε ένας προσωπικός υπολογιστής, έχει μεταφερθεί στο cloud. Εφαρμογές που παλαιότερα τρέχαν σε κάποιο τοπικό σύστημα, καθώς και επιπρόσθετες εφαρμογές που δημιουργήθηκαν λόγω της ανόδου του cloud computing, τρέχουν σε κάποιο cloud center απομακρυσμένα από τον χρήστη. Έτσι, μειώνονται οι απαιτήσεις από τον προσωπικό υπολογιστή του. Αυτό, όμως, σημαίνει ότι οι απαιτήσεις εκτέλεσης και αποθήκευσης των εφαρμογών έχουν μεταφερθεί στα cloud centers, μεταβιβάζοντας σε αυτά το βάρος διαχείρισης τους. Αυτός είναι ένας από τους λόγους όπου το cloud computing και οι υπηρεσίες cloud είναι ένα από τα πιο επίκαιρα και ερευνημένα θέματα του κλάδου της επιστήμης των υπολογιστών στην σύγχρονη εποχή.

Κύριος παράγοντας της ανόδου του cloud computing είναι η τεχνολογία της εικονοποίησης. Αυτή επιτρέπει την δημιουργία και εκτέλεση περιβαλλόντων που λειτουργούν ως ανεξάρτητοι υπολογιστές. Η φιλοξένηση αυτών των μηχανών σε cloud centers και η απομακρυσμένη πρόσβαση που επιτρέπουν στους χρήστες δημιουργεί τις υπηρεσίες στο cloud, αφού ένας χρήστης μπορεί πλέον να τρέξει την εκάστοτε υπηρεσία σε ένα προσωπικό εικονικό μηχάνημα που φιλοξενείται σε κάποιο cloud center. Με αυτή την αύξηση στην χρήση υπηρεσιών cloud, αυξήθηκαν και οι απαιτήσεις σε υποδομές και λογισμικό που αξιοποιεί αποδοτικά το υποβόσκων υλικό, καθως αυξήθηκε ραγδαία και ο αριθμός εικονικών μηχανών που χρειάζεται να υποστηρίζει ένα cloud center.

Παρότι η κεντροποίηση του cloud μειώνει την σπατάλη υπολογιστικών πόρων, ακόμη είναι συχνό το φαινόμενο να γίνεται άσκοπη χρήση πόρων λόγω της χρήσης συμβατικών λειτουργικών συστημάτων. Τα λειτουργικά συστήματα που χρησιμοποιούνται στο cloud είναι συχνά γενικού σκοπού, δηλαδή έχουν ως σκοπό την υποστήριξη ευρείας γκάμας διαφορετικών εργασιών. Αντιθέτως, οι εικονικές μηχανές συχνά δημιουργούνται για την εξυπηρέτηση ενός εξειδικευμένου σκοπού. Λόγω αυτής της ασυμβα-

τότητας μεταξύ του σχεδιασμού των διαδεδομένων λειτουργικών συστημάτων και των σύγχρονων αναγκών, αφήνονται αναξιοποίητοι υπολογιστικοί πόροι ενώ δημιουργούνται και οικολογικά προβλήματα από την περιττή σπατάλη υλικού και ηλεκτρισμού στα cloud centers.

1.1 Σκοπός της εργασίας

Στην φιλοσοφία των γρήγορων και εξειδικευμένων μηχανών βασίστηκαν τα unikernels. Εξειδικευμένα μηχανήματα που τρέχουν μια εφαρμογή εις πέρας με ελάχιστες ανάγκες σε υλικό (κύρια μνήμη, κύκλους επεξεργαστή κ.ο.κ). Έτσι, είναι δυνατό να αυξηθεί ο αριθμός των εικονικών μηχανών σε κάθε μηχανήμα φιλοξενητή κατά μερικές τάξεις μεγέθους, καθώς και να μειωθεί ο χρόνος απόκρισης των υπηρεσιών του cloud.

Κύριος άξονας της φιλοσοφίας των unikernels είναι η εκτέλεση μίας και μοναδικής διεργασίας ανά εικονική μηχανή όπως συχνά παρατηρείται να γίνεται στο cloud. Αυτό, όμως, σημαίνει ότι κλήσεις όπως αυτή της fork και άλλων κλήσεων για την δημιουργία και διαχείριση διεργασιών, καθώς και άλλες διαδιεργασιακές κλήσεις συστήματος δεν υποστηρίζονται. Έτσι, για την εκτέλεσή της, μια εφαρμογή πρέπει να διαμορφωθεί κατάλληλα για να τρέξει στα νέα πλαίσια που ορίζει το εκάστοτε unikernel. Αυτό κοστίζει σε χρόνο για τους developers που θα πρέπει να συμμορφώνουν την εφαρμογή τους ανάλογα το unikernel framework με το οποίο θέλουν να είναι συμβατοί.

Για αυτό το λόγο, η εργασία αυτή καταπιάστηκε με την δημιουργία της κλήσης συστήματος fork στα πλαίσια ενός unikernel framework που έχει σκοπό την υποστήριξη εφαρμογών που ακολουθούν το πρότυπο POSIX. Παρότι η προσπάθεια υλοποίησης της fork έγινε με γνώμονα το πρότυπο POSIX, η ιδιαιτερότητα των Rumprun unikernels με solo5 hypervisor, δεν επέτρεψε την πλήρη συμμόρφωσή της με αυτό, όπως θα εξηγηθεί στην εργασία. Επίσης, υλοποιήθηκε και η υποστήριξη για την συνάρτηση wait που είναι σημαντική για την επικοινωνία και συγχρονισμό μεταξύ συγγενικών διεργασίων.

1.2 Οργάνωση της εργασίας

Αρχικά, γίνεται αναφορά στο απαραίτητο θεωρητικό υπόβαθρό στο κεφάλαιο 2. Σε αυτό το κεφάλαιο αναφέρονται οι έννοιες του cloud computing, της εικονοποιήσης και των λειτουργικών συστημάτων που είναι βασικές έννοιες σχετικές με το θέμα της παρούσας εργασίας.

Στο κεφάλαιο 3 θα γίνει μια ανασκόπηση μερικών από τα υπάρχοντα unikernel frameworks, καθώς και του Rumprun που είναι και το framework στο οποίο

υλοποιήθηκε αυτή η εργασία.

Στο κεφάλαιο 4 θα παρουσιαστούν ο σχεδιασμός και η υλοποίηση της fork και του wait. Τέλος, στο κεφάλαιο 5 θα αναφερθούν τα τελικά συμπεράσματα από την ασχολία με το συγκεκριμένο θέμα και διάφορες παρατηρήσεις πάνω σε αυτά.

- Κεφάλαιο 2: Αναφορά στις απαραίτητες θεωρητικές γνώσεις
- Κεφάλαιο 3: Ανασκόπηση των υπάρχοντων unikernel frameworks
- Κεφάλαιο 4: Σχεδιασμός και υλοποίηση fork και wait
- Κεφάλαιο 5: Συμπεράσματα και αναφορές σε περαιτέρω μελέτη

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Λειτουργικά Συστήματα

Ένα λειτουργικό σύστημα είναι ένα αυτοτελές λογισμικό το οποίο διαχειρίζεται ένα υπολογιστικό σύστημα είτε αυτό είναι φυσικό είτε εικονικό. Την διαχείριση του την πετυχαίνει χρησιμοποιώντας ένα σύνολο υποσυστημάτων, τα οποία επιτρέπουν την εκτέλεση εφαρμογών πάνω στο υπολογιστικό σύστημα. Δύο από τα κυριότερα υποσυστήματα ενός λειτουργικού συστήματος είναι αυτό της εικονικής μνήμης που διαχωρίζει την μνήμη που βλέπει το πρόγραμμα χρήστη από την φυσική μνήμη του υλικού και αυτό της εκτέλεσης διεργασιών το οποίο γνωρίζει πως να τρέξει ένα εκτελέσιμο πρόγραμμα.

Μια ακόμη σημαντική λειτουργία ενός λειτουργικού συστήματος είναι η διεπαφή του υλικού με το λογισμικό. Με εξαίρεση συστήματα όπως αυτό της μνήμης και του επεξεργαστή, οι συσκευές ενός υπολογιστή χρειάζονται οδηγούς συσκευών για να τρέξουν. Οι οδηγοί συσκευών λειτουργούν ως διεπαφή μεταξύ του λειτουργικού και κάποιας συσκευής υλικού, ώστε να γίνει κατάλληλη μετάφραση μεταξύ των δεδομένων του εκάστοτε υλικού για να τα παραλάβει το λειτουργικό σύστημα σε μορφή που μπορεί να επεξεργαστεί. Οι οδηγοί συσκευών ποικίλουν από οδηγούς για ασύρματα ποντίκια μέχρι την δευτερεύουσα μνήμη του σκληρού δίσκου. Αξιοσημείωτο είναι ότι το μεγαλύτερο μέρος της βάσης κώδικα ενός λειτουργικού συστήματος το αποτελούν οι οδηγοί συσκευών.

Τα παραπάνω κομμάτια ενός λειτουργικού συστήματος, μαζί με κάποια ακόμη που δεν αναφέρονται, αποτελούν τον πυρήνα του. Μια διάκριση που μπορεί να γίνει μεταξύ των λειτουργικών συστημάτων είναι ως προς την δομή του πυρήνα. Μία σημαντική κατηγορία είναι τα μονολιθικά λειτουργικά συστήματα, όπως είναι τα Linux τα οποία αποτέλουν τον πλέον διαδεδομένο πυρήνα για λειτουργικά συστήματα εξειδικευμένου

σκοπού, όπως είναι τα data centres, υπερυπολογιστές και ενσωματωμμένα συστήματα καθώς και για κινητά όπως είναι οι συσκευές με Android λειτουργικό.

Τα μονολιθικά λειτουργικά συστήματα (εικόνα 2.1) διαχωρίζουν την εκτέλεση τους σε δύο περιβάλλοντα· το περιβάλλον χρήστη (userspace) και το περιβάλλον πυρήνα (kernelspace). Με αυτό τον διαχωρισμό ο πυρήνας τρέχει όλες τις λειτουργίες του και τις υπηρεσίες που του ζητάει ο χρήστης στο δικό του χώρο διευθύνσεων, ενώ απομονώνει τις εφαρμογές χρήστη στο userspace. Το λειτουργικό σύστημα, λοιπόν, αποτελεί μια υψηλού επιπέδου διεπαφή με την οποία μπορεί να αλληλεπιδράσει ο χρήστης μέσω κλήσεων συστήματος. Οι κλήσεις συστήματος είναι προγραμματιστικές διεπαφές που επιτρέπουν στον χρήστη να ζητήσει από τον πυρήνα να εκτελέσει υπηρεσίες. Τέτοιες κλήσεις, όπως είναι η χρήση του συστήματος αρχείων ή κάποιας συσκευής υλικού μέσω ενός οδηγού συσκευής, λαμβάνονται από τον πυρήνα όπου και τρέχουν στο kernelspace για χάρη του χρήστη.

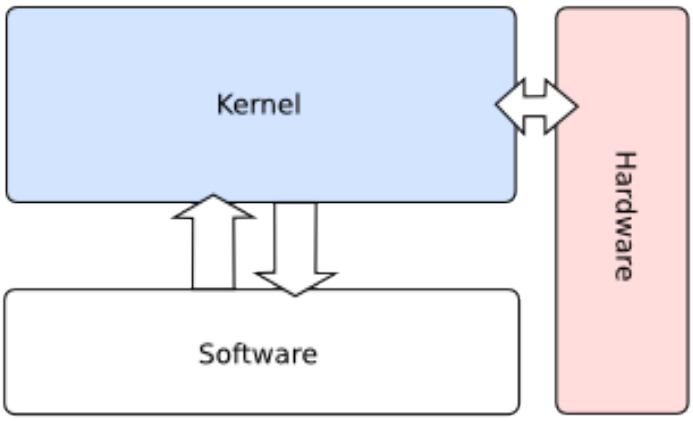

Figure 2.1: Μονολιθική δομή λειτουργικού συστήματος

Ο διαχωρισμός των διευθύνσεων σε user και kernel space απομονώνει το χώρο διευθύνσεων του χρήστη από αυτόν του πυρήνα απεμπλέκοντας την εκτελεσή τους. Το γεγονός, όμως, ότι όλες οι υπηρεσίες του πυρήνα τρέχουν στον ίδιο χώρο διευθύνσεων επιβάλει τον κίνδυνο ότι άμα μια υπηρεσία του πυρήνα αποτύχει, ολόκληρος ο πυρήνας μπορεί να βρεθεί σε κατάσταση πανικού αποτυγχάνοντας να ανακάμψει ή ανακάμπτοντας με μη προβλέψιμη συμπεριφορά.

Άλλες κατηγορίες λειτουργικών συστημάτων είναι:

- Πολυεπίπεδα: Τα λειτουργικά συστήματα τα οποία διατηρούν μια ιεραρχεία μεταξύ των επιμέρους κομματιών τους σε μορφή επιπέδων όπου το μηδενικό επίπεδο αντιστοιχεί στο υλικό, ενώ το μέγιστο επίπεδο στην διεπαφή χρήστη. Κάθε επίπεδο χρησιμοποιεί συναρτήσεις μόνο από επίπεδα κάτω από αυτό.

- Μικροπυρήνες: Όταν αφαιρεθούν όλα τα μη απαραίτητα κομμάτια από ένα μονολιθικό πύρηνα, δημιουργείται ένας μικρότερος πυρήνας που ονομάζεται μικροπυρήνας. Οι μη καίριες λειτουργίες του πυρήνα γράφονται ως κώδικας χρήστη ή συστήματος και εκτελούνται σε ξεχωριστό χώρο διευθύνσεων από αυτόν του πυρήνα ενώ επικοινωνούν με τον πυρήνα μέσω μηνυμάτων. Αυτό προσφέρει μεγαλύτερη ασφάλεια στον πυρήνα, καθώς ακόμη και κάποιες υπηρεσίες να αποτύχουν, είναι ευκολότερο να ανακάμψει.

- Υβριδικά: Αυτά τα λειτουργικά συστήματα είναι μια μίξη μεταξύ των μικροπυρήνων και των μονολιθικών λειτουργικών συστημάτων. Ένα παράδειγμα τέτοιου πυρήνα είναι ο Windows NT kernel των τελευταίων εκδόσεων των Windows, ο οποίος τρέχει ένας μέρος των υποσυστημάτων του σε user mode (αντίστοιχο userspace στα Linux), και κάποια στον χώρο διευθύνσεων του πυρήνα.

Μια επίσης ενδιαφέρουσα δομή πυρήνα είναι το anykernel. Σε αυτή την δομή, όλοι οι οδηγοί συσκευών του πυρήνα είναι γραμμένοι ως βιβλιοθήκες οι οποίες μπορούν να τρέξουν είτε ως μέρος του πυρήνα όπως στα μονολιθικά συστήμα, είτε ως μέρος του userspace ως βιβλιοθήκες που μπορούν να χρησιμοποιηθούν από όλες τις εφαρμογές. Αυτό έρχεται σε αντίθεση με τα μονολιθικά λειτουργικά συστήματα όπου οι οδηγοί συσκευών είναι αναπόσπαστο μέρος του πυρήνα. Αυτή η δομή, λοιπόν, επιτρέπει στο λειτουργικό σύστημα να έχει φορτωμένες στην μνήμη μόνο τις βιβλιοθήκες που χρειάζεται για τα εκάστοτε προγράμματα που τρέχει. Αντιθέτως, οι μονολιθικοί πυρήνες λόγω της δομής τους, φορτώνονται ολόκληροι, χωρίς απαραίτητα να αξιοποιούνται όλες οι λειτουργίες τους. Αυτή η δομή πυρήνα οδήγησε και στην δημιουργία των rump kernels που θα αναφερθούν στο κεφάλαιο 3.

2.2 Εικονοποίηση

Με τον όρο εικονοποίηση αναφερόμαστε στην δημιουργία ενός εικονικού αντικειμένου. Πιο συγκεκριμένα, στην επιστήμη των υπολογιστών, η εικονοποίηση αναφέρεται στην δημιουργία μιας εικόνας ενός συστήματος. Το σύστημα αυτό μπορεί να

ποικίλει από την εικόνα ενός μέρους του υλικού μέχρι ένα ολόκληρο δίκτυο υπολογιστών.

Σε έναν υπολογιστή η τεχνική της εικονοποίησης χρησιμοποιείται σε πολλαπλά επίπεδα. Ένα καίριο εικονικό σύστημα είναι αυτό της κύριας μνήμης. Η εικονική μνήμη επιτρέπει σε κάθε πρόγραμμα που τρέχει σε έναν υπολογιστή, να βρίσκεται σε περιβάλλον στο οποίο όλη η μνήμη του υπολογιστή να φαίνεται διαθέσιμη σε αυτό, ακόμη και περισσότερη από αυτή που πραγματικά έχει ο υπολογιστής. Προφανώς, η πραγματική μνήμη του υπολογιστή δεν ανήκει σε μια και μόνο διεργασία, καθώς ακόμη και ο πυρήνας που την διαχειρίζεται είναι φορτωμένος σε αυτή. Σε παλαιότερα λειτουργικά συστήματα, η έλλειψη της εικονικής μνήμης δυσχαίρενε τον προγραμματιστή, καθώς κατά την ανάπτυξη ενός προγράμματος έπρεπε να διαχειρίζεται την κοινή μνήμη μεταξύ των διεργασιών κατάλληλα. Έτσι, η εικονοποίηση επιτρέπει την δημιουργία επιπέδων αφαιρετικότητας με σκοπό την απλοποίηση ενός πολύπλοκου προβλήματος, όπως αυτό της διαχείρισης μνήμης. Κάθε επίπεδο εικονοποίησης έχει τα δικά του πλεονεκτήματα και μειονεκτήματα και η μελέτη τους είναι εκτός του σκοπού αυτής της εργασίας.

Στην εικόνα 2.2 φαίνονται οι διάφορες μορφές εικονοποίησης που μας ενδιαφέρουν στα πλαίσια της εργασίας. Παρακάτω θα εξηγηθεί η εικονοποίηση υλικού, καθώς και η εικονοποίηση σε επίπεδο λειτουργικού συστήματος.

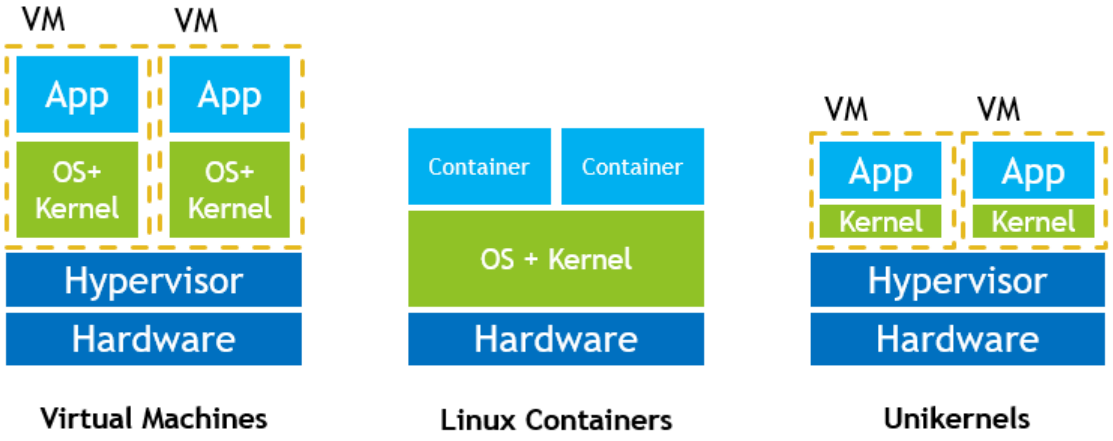

Figure 2.2: Μορφές εικονοποίησης

2.2.1 Εικονοποίηση υλικού

Η εικονοποίηση υλικού - ή αλλιώς εικονοποίηση πλατφόρμας - αφορά την δημιουργία μιας εικονικής μηχανής, η οποία προσομοιώνει μια πραγματική μηχανή πλήρως. Αυτό επιτρέπει σε έναν υπολογιστή να δημιουργεί πολλαπλές εικόνες υπολογιστών όπου η

κάθε μία έχει το δικό της λειτουργικό σύστημα και το δικό της υλικό, χωρίς απαραίτητα αυτά να έχουν την ίδια αρχιτεκτονική με το υλικό του φυσικού μηχανήματος.

Ο πραγματικός υπολογιστής που χρησιμοποιεί την εικονοποίηση για να δημιουργήσει τις εικονικές μηχανές λέγεται φιλοξενιτής ή host για συντομία, ενώ η εικονική μηχανή λέγεται guest machine ή απλώς guest.

Κάτω από το guest machine υπάρχει ο υπερεπόπτης (hypervisor ή virtual machine monitor - VMM), είτε ως λογισμικό στο userspace του host, το οποίο δημιουργεί και διαχειρίζεται εικονικές μηχανές για αυτόν, είτε απευθείας πάνω στο φυσικό υλικό. Επίσης, παρέχει το εικονικό υλικό στον guest και αναλαμβάνει την εκτέλεση των υπηρεσιών του λειτουργικού συστήματός του.

Υπάρχουν δύο τύποι hypervisor (εικόνα 2.3):

- Τύπου 1: Αναφέρονται και ως native/bare metal hypervisors καθώς τρέχουν απευθείας πάνω στο υλικό του host, χωρίς την ύπαρξη host λειτουργικού συστήματος.
- Τύπου 2: Τρέχουν ως διεργασία στο userspace του host. Γνωστά παραδείγματα τέτοιων hypervisors είναι το VMWare και το VirtualBox.

Μερικοί hypervisors μπορούν να λειτουργήσουν ως τύπου 1 και ως 2, όπως είναι ο QEMU/KVM που μπορεί να τρέξει είτε απευθείας πάνω στο υλικό, είτε ως διεργασία πάνω από ένα πλήρες λειτουργικό σύστημα.

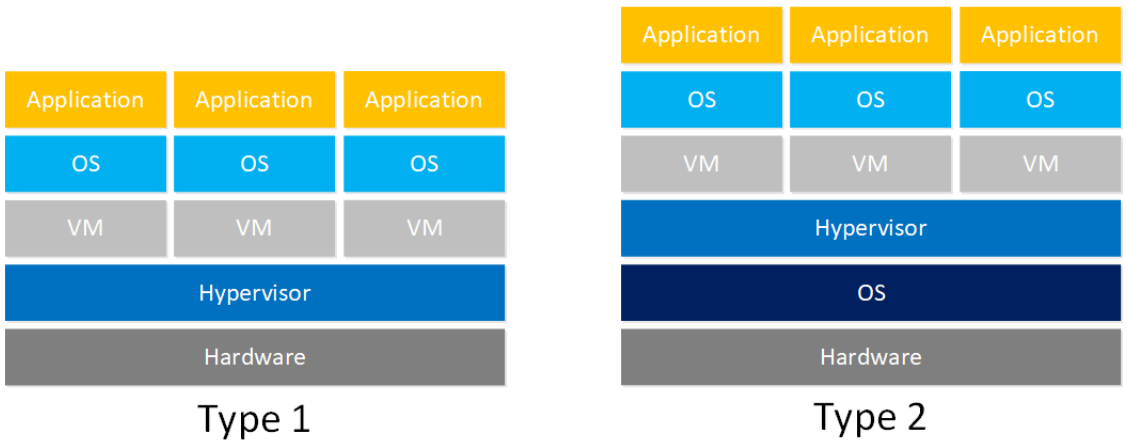

Figure 2.3: Κατηγορίες hypervisor

Με την άνοδο του ενδιαφέροντος και των αναγκών εικονοποίησης, δημιουργήθηκαν οι τεχνολογίες Intel VT-x και AMD-v από την Intel και την AMD αντίστοιχα. Αυτές υποστηρίζουν την εικονικοποίηση, μειώνοντας τους κύκλους επεξεργαστή που χρειάζονται οι εντόλες μιας εικονικής μηχανής, επιτυγχάνοντας έτσι καλύτερες επιδόσεις κατά την εκτέλεση τους, αλλά απαιτώντας και εξειδικευμένο υλικό για την επίτευξη εικονοποίησης.

Βασικές μορφές εικονοποίησης υλικού (με ή χωρίς επιτάχυνση) είναι:

- Full virtualization: Επιτρέπει σε ένα guest λειτουργικό σύστημα να τρέξει χωρίς τροποποιήσεις, καθώς όλα τα μέρη του υλικού προσομοιώνονται.
- Paravirtualization: Το guest λειτουργικό σύστημα γνωρίζει ότι τρέχει σε εικονικό περιβάλλον. Σε αυτή την μορφή εικονοποίησης δεν είναι απαραίτητη η προσομοίωση όλου του υλικού, αλλά οι κλήσεις του guest λειτουργικού προς το υλικό γίνονται μέσω διεπαφής του hypervisor. Για να επιτευχθεί το paravirtualization πρέπει να γίνουν κατάλληλες αλλάγες στο guest λειτουργικό.
- Hardware-assisted virtualization: Σε αυτή την μορφή εικονοποίησης, το υλικό δουλεύει μαζί με τον hypervisor για να προσφέρουν πιο γρήγορη εκτέλεση των λειτουργιών της εικονικής μηχανής.

Μερικά από τα πλεονεκτήματα και μειονεκτήματα αυτής της μεθόδου συνοψίζονται παρακάτω:

+ **Υψηλή απομόνωση:** Η απομόνωση αναφέρεται τόσο μεταξύ του host μηχανήματος με τους guests, αλλά και μεταξύ των εικονικών μηχανών. Αυτό προσφέρει ασφάλεια από κακόβουλη χρήση, καθώς άμα μια εικονική μηχανή αποτύχει, το host σύστημα δεν σφάλει και δεν επηρεάζεται το ίδιο ή κάποια από τις υπόλοιπες εικονικές μηχανές του.
+ **Φορητότητα:** Επιτρέπει την αποθήκευση και μεταφορά εικονικών μηχανών, οι οποίες μπορούν να τρέξουν σε οποιοδήποτε σύστημα έχει συμβατό hypervisor, καθώς δεν εξαρτώνται από το λειτουργικό σύστημα του host και την εκάστοτε αρχιτεκτονική του υλικου που χρησιμοποιεί.
+ **Περιβάλλον δοκιμής [1]:** Τα παραπάνω πλεονεκτήματα σε συνδυασμό με την δυνατότητα εύκολης δημιουργίας και διαγραφής εικονικών μηχανών, επιτρέπουν την γρήγορη κατασκευή περιβαλλόντων δοκιμής απομονωμένα από το υπόλοιπο λειτουργικό σύστημα. Έτσι, προσφέρουν ασφάλεια από σφάλματα του χρήστη ή του προγράμματος κατά την χρήση του στο περιβάλλον της εικονικής μηχανής.

- **Σπατάλη πόρων και ταχύτητας:** Η εικονοποίηση ολόκληρου του λογισμικού και του υλικού αποτελεί μεγάλη σπατάλη πόρων, καθώς τις περισσότερες φορές, οι εικονικές μηχανές τρέχουν μόνο μία διεργασία. Δηλαδή, μεγάλο μέρος του εικονοποιημένου λογισμικού και υλικού δεν χρησιμοποιείται για την λειτουργία της διεργασίας, μειώνοντας δραστικά τον αριθμό των εικονικών μηχανών που μπορεί να φιλοξενήσει ο host.

- **Χρόνος εκκίνησης:** Μια εικονική μηχανή κατά την εκκίνησή της εκτελεί όλη την ρουτίνα αρχικοποίησης του λειτουργικού συστήματός της. Αυτό δημιουργεί μεγάλους χρόνους μεταξύ της ζήτησης μιας cloud υπηρεσίας και της παροχής της, αποτρέποντας την χρήση πλήρους εικονικών μηχανών για την ικανοποίηση υπηρεσιών πραγματικού χρόνου.

- **Ασφάλεια εικονικής μηχανής:** Όπως αναφέρθηκε παραπάνω, αυτή η μορφή εικονοποίησης προϋποθέτει την ύπαρξη ολόκληρου λειτουργικού συστήματος. Από την μία, αυτό διευκολύνει την εκτέλεση εφαρμογών, καθώς δεν απαιτεί τροποποίηση στον κώδικά τους. Από την άλλη, όμως, αυξάνει τον αριθμό των υπηρεσίων που τρέχουν από προεπιλογή στο σύστημα. Αυτό αυξάνει την επιφάνεια επίθεσης που μπορεί να χρησιμοποιήσει ένας κακόβουλος χρήστης, μειώνοντας την ασφάλεια της εικονικής μηχανής.

2.2.2 Εικονοποίηση λειτουργικού συστήματος

Σε αυτή την μορφή εικονοποίησης ο πυρήνας επιτρέπει την ύπαρξη πολλάπλών userspaces. Τα userspaces μεταξύ τους είναι απομονωμένα, δηλαδή το καθένα βλέπει μόνο τους δικούς του πόρους, υπηρεσίες που του προσφέρει το λειτουργικού σύστημα και συσκευές που του έχουν ανατεθεί. Από την άλλη οπτική, ο πυρήνας βλέπει όλους τους πόρους και τις συσκευές που υπάρχουν, καθώς και όλα τα επιμέρους userspaces των εφαρμογών. Ο πυρήνας τρέχει στο kernelspace όλες τις λειτουργίες που του ζητάνε τα προγράμματα στα userspaces, όπως δηλαδή θα έτρεχε ένα μονολιθικό λειτουργικό σύστημα.

Αυτή η μορφή εικονοποίησης αρχικά υποστηρίχθηκε από τα Linux στην μορφή του LXC [1]. Στις μέρες μας η πιο διαδεδομένη μορφή εικονοποίησης λειτουργικού συστήματος είναι τα containers.

Τα containers είναι μια μορφή εικονοποίησης στην οποία γίνεται packaged ο κώδικας μιας εφαρμογής μαζί με τις βιβλιοθήκες και τις εξαρτήσεις της. Κάθε container τρέχει σε ένα δικό του userspace πάνω σε κάποιο host λειτουργικό σύστημα. Η απομόνωση μεταξύ των containers διασφαλίζεται από το λειτουργικό σύστημα μέσω των μηχανι-

[1] `https://linuxcontainers.org`

σμών απομόνωσης διεργασίων που προσφέρει. Σε αντίθεση με τις εικονικές μηχανές, σε αυτή την μορφή εικονοποίησης δεν εικονοποιείται το υποβόσκον υλικό, αλλά το guest λειτουργικό σύστημα. Έτσι, κάθε container μιας εφαρμογής δεν περιέχει δικό της λειτουργικό σύστημα, αλλά χρησιμοποιεί ένα κοινό λειτουργικό σύστημα που μοιράζεται με άλλα containers του ίδιου host μηχανήματος. Η έλλειψη λειτουργικού συστήματος στα containers καταφέρνει να μειώσει το μέγεθος της εικόνας τους από μερικά giga bytes σε μερικά mega bytes. Αυτό οδηγεί στην καλύτερη αξιοποίηση του εκάστοτε host των containers, αφού πλέον μπορεί να αυξήσει κατά πολύ τον αριθμό guests που μπορεί να εξυπηρετεί.

Συνήθως, για την εκκίνηση και διαχείριση των containers χρησιμοποιείται κάποιο container engine όπως φαίνεται στην εικόνα 2.4. Η διαφορά ενός container engine, όπως είναι το Docker, με έναν hypervisor είναι ότι ο δεύτερος μπορεί είτε να τρέξει απευθείας πάνω στο υλικό, είτε ως διεργασία στο userspace του host λειτουργικού συστήματος. Στην περίπτωση του hypervisor, η εφαρμογή χρήστη επικοινωνεί με το εικονοποιημένο guest λειτουργικό σύστημα, το οποίο με την σειρά του επικοινωνεί με τον hypervisor για να επικοινωνήσει με τον πυρήνα του host. Σε αντίθεση, τα containers επικοινωνούν απευθείας με τον πυρήνα του host λειτουργικού συστήματος μέσω του container engine.

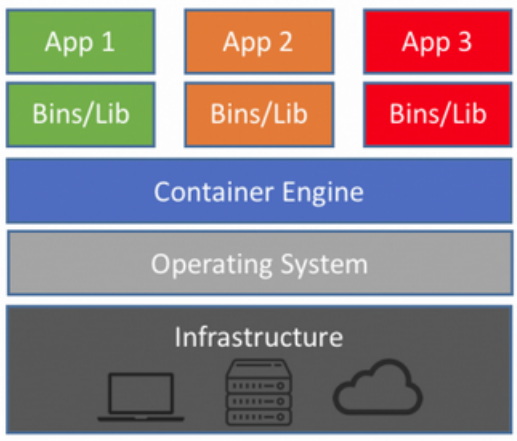

Figure 2.4: Δομή containers

Μερικά από τα πλεονεκτήματα και μειονεκτήματα αυτής της μεθόδου συνοψίζονται παρακάτω:

+ **Εξοικονόμηση πόρων:** Όλα τα containers χρησιμοποιούν ένα κοινό λειτουργικό σύστημα και σε κάθε ένα δίνονται μόνο οι πόροι που χρειάζεται. Έτσι, δεν χρειάζεται η αποθήκευση πολλαπλών ίδιων λειτουργικών συστημάτων, μειώνοντας τον καταναλισκόμενο χώρο. Αυτό αυξάνει δραστικά τον αριθμό των containers που μπορεί να φιλοξενήσει το host σύστημα.

+ **Χρόνος εκκίνησης:** Για να εκκινήσει ένα container, δεν χρειάζεται η αρχικοποίηση ενός λειτουργικού συστήματος κάθε φορά, παρά μόνο από τον host κατά την εκκίνηση του. Απαιτείται μόνο η αρχικοποίηση του userspace του container, μειώνοντας σημαντικά τον χρόνο εκκίνησης της εφαρμογής.

- **Μικρότερη απομόνωση:** Καθώς όλα τα containers χρησιμοποιούν κοινό λειτουργικό σύστημα για να τρέξουν, οι λειτουργίες του πυρήνα τρέχουν σε κοινό χώρο. Αυτό, όμως, προκαλεί προβλήματα απομόνωσης, καθώς ένας κακόβουλος χρήστης μπορεί να πάρει πρόσβαση ή να προκαλέσει σφάλματα στα υπόλοιπα containers του host ή ακόμη και στο ίδιο το host μηχάνημα.

- **Συμβατότητα με host:** Η χρήση του λειτουργικού συστήματος του host από τα containers σημαίνει ότι η εφαρμογή που τρέχει σε αυτά πρέπει να είναι φτιαγμένη για το λειτουργικό σύστημα του host. Δηλαδή, ένα container που περίεχει μια εφαρμογή που είναι φτιαγμένη για Linux δεν μπορεί να τρέξει σε host με λειτουργικό σύστημα Windows.

Παρότι τα containers μειώσαν πολύ την χρήση περιττών πόρων των host μηχανών, η έλλειψη πλήρους απομόνωσης μεταξύ των userspaces και η σύγχρονη χρήση του υλικού από τα containers οδήγησαν σε προβλήματα ασφάλειας. Έτσι, η χρήση τους τελικά γίνεται μέσα σε εικονικές μηχανές, αναιρόντας το αρχικό τους σχεδιαστικό πλεονέκτημα.

2.3 Cloud Computing

Ο υπολογισμός νέφους (cloud computing), αναφέρεται στην παροχή υπολογιστικών πόρων σε χρήστες, χωρίς να χρειάζεται να τις διαχειρίζονται οι ίδιοι. Αυτό επιτυγχάνεται από κέντρα δεδομένων τα οποία παρέχουν υπηρεσίες και υλικό σε διάφορες μορφές.

Καθώς η έννοια του cloud computing είναι κάπως αφηρημένη, δεν υπάρχει σαφής ορισμός. Για αυτό τον λόγο το National Institute of Standards and Technology (NIST) δημιούργησε έναν ορισμό που περιγράφει τα βασικά χαρακτηριστικά, τα μοντέλα υπηρεσιών και τα τέσσερα μοντέλα deployment του cloud computing. Σύμφωνα με το NIST[2]:

Ο υπολογισμός νέφους είναι ένα μοντέλο που επιτρέπει την εύκολη και κατά απαίτηση δικτυακή πρόσβαση ενός συνόλου από ρυθμιζόμενους υπολογιστικούς πόρους που μπορούν να παρασχεθούν και ελευθερωθούν με ελάχιστο διαχειριστικό κόπο ή αλληλεπίδραση με τον προμηθευτή της υπηρεσίας.

2.3.1 Βασικά Χαρακτηριστικά

Τα βασικά χαρακτηριστικά που προσδίδει το NIST στο cloud computing φαίνονται στην εικόνα 2.5. Τα χαρακτηριστικά αυτά είναι τα εξής:

- κατ'απαίτηση αυτοεξυπηρέτηση, δηλαδή ένας χρήστης μπορεί να παρασχεθεί πόρους χωρίς την διαμεσολάβηση ανθρώπινης επικοινωνίας με τον πάροχο,
- ευρεία δικτυακή πρόσβαση, έτσι ο χρήστης μπορεί να χρησιμοποιήσει τους παρασχεθέντες πόρους μέσω κλασσικών μέσων πρόσβασης του διαδικτύου (μέσω κινητού ή επιτραπέζιου υπολογιστή),
- συσσώρευση πόρων σε λίγα μηχανήματα χρησιμοποιόντας την τεχνολογία της εικονοποίησης που αναλύθηκε προηγουμένως στην εργασία,
- ταχεία ελαστικοτήτα, καθώς ένας χρήστης μπορεί να αλλάξει τις απαιτήσεις φόρτου από την υπηρεσία καθόλη την διάρκεια χρήσης της, είτε μειώνοντας ή αυξάνοντάς τους,
- μετρήσιμες υπηρεσίες, μέσω διαφόρων μετρικών που χρησιμοποιούνται από τον προμηθευτή για την καλύτερη ανάθεση και αξιοποίηση των υπολογιστικών του πόρων, καθώς και σε πολλές περιπτώσεις την διαφάνεια στην χρήση πόρων προς τον καταναλωτή.

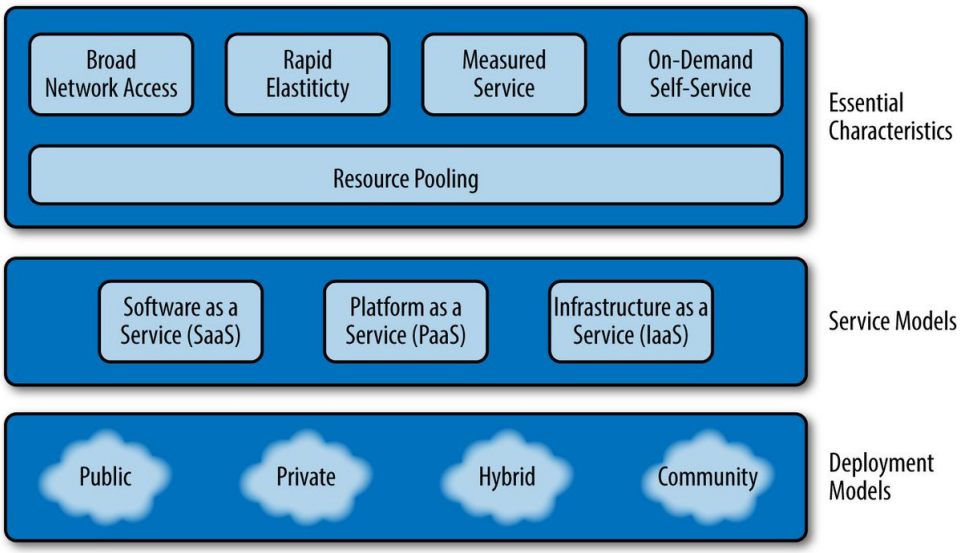

Figure 2.5: Χαρακτηριστικά υπολογιστικού νέφους

2.3.2 Μοντέλα υπηρεσιών

Τα μοντέλα υπηρεσιών που προσφέρει το cloud computing σύμφωνα με το NIST είναι τα εξής:

- **Software as a service(SaaS)**: Σε αυτή την μορφή υπηρεσίας παρέχεται στον χρήστη η δυνατότητα χρήσης κάποιας εφαρμογής που φιλοξενεί ο πάροχος, χωρίς να χρειάζεται να διαχειριστεί απευθείας τους υποβόσκοντες πόρους όπως η κύρια μνήμη, το ποσοστό χρήσης του επεξεργαστή κ.ο.κ. Η χρήση της υπηρεσίας γίνεται μέσω κάποιου προγράμματος ή διεπαφής, ενώ μπορεί να προσφέρονται κάποιες περιορισμένες ρυθμίσεις εξατομικευμένες στον χρήστη. Παραδείγματα τέτοιων υπηρεσιών είναι το Slack και το Dropbox.
- **Platform as a service(PaaS)**: Ομοίως με πριν, σε αυτή την μορφή υπηρεσίας, ο χρήστης δεν έχει έλεγχο τον υποβόσκοντων πόρων, αλλά ο προμηθευτής του παρέχει ένα σύνολο υποστηριζόμενων εργαλείων όπως γλώσσες προγραμματισμού, βιβλιοθήκες και άλλα, τα οποία ο καταναλωτής μπορεί να χρησιμοποιήσει για την δημιουργία και εκτέλεση εφαρμογών στο περιβάλλον του νέφους. Οι ρυθμίσεις που μπορεί να κάνει ο χρήστης αφορούν το περιβάλλον εκτέλεσης της εφαρμογής και ρυθμίσεις που αφορούν το deployment της. Παραδείγμα τέτοιων υπηρεσιών είναι το Heroku.
- **Infrastructure as a service(IaaS)**: Αυτή η μορφή υπηρεσίας επιτρέπει στον χρήστη την διάθεση πόρων για την εκτέλεση αυθαίρετων προγραμμάτων όπως λειτουργικά συστήματα ή εφαρμογές. Παρότι ο χρήστης δεν έχει έλεγχο των υποδομών, έχει τον έλεγχο του λειτουργικού συστήματος, του διαθέσιμου αποθηκευτικού χώρου και των deployed εφαρμογών. Σε περιπτώσεις, μπορεί να του παρέχονται περισσότερες επιλογές, όπως η περιορισμένη ρύθμιση του firewall του φιλοξενιτή. Παραδείγματα τέτοιων υπηρεσιών είναι το Amazon Web Service (AWS) και το Google Cloud Engine (GCE).

2.3.3 Μοντέλα deployment

Τα deployment μοντέλα που ορίζει το NIST είναι τα εξής τέσσερα:

- Ιδιωτικό νέφος: Οι υποδομές του νέφους χρησιμοποιούνται αποκλειστικά από ένα πρόσωπο, όπως μια εταιρεία. Οι υποδομές μπορεί να ανήκουν και να διαχειρίζονται από κάποιο τρίτο πρόσωπο είτε από την ίδια την οργάνωση που το χρησιμοποιεί, είτε από ένα συνδυασμό αυτών.
- Κοινοτικό νέφος: Η διαφορά με το προηγούμενο μοντέλο, είναι ότι σε αυτή την περίπτωση το νέφος μπορεί να χρησιμοποιείται από μια κοινότητα που μοιράζεται κάποιο σκοπό.
- Δημόσιο νέφος: Σε αυτό το μοντέλο, οι υποδομές του νέφους φτιάχτηκαν για να χρησιμοποιούνται από το γενικό κοινό. Η παροχή τους μπορεί να γίνεται από διάφορους φορείς, όπως πανεπιστημία.

- Υβριδικό νέφος: Η δημιουργία νέφων τα οποία είναι ένας συνδυασμός δύο ή περισσοτέρων από τα προηγούμενα μοντέλα, δημιουργεί υβριδικά νέφη. Σε αυτή την μορφή τα επιμέρους νέφη παραμένουν ανεξάρτητα, αλλά υπάρχει κάποια μορφή φορητότητας των εφαρμογών και των δεδομένων μεταξύ τους.

Κεφάλαιο 3

Unikernels

Όπως είδαμε στο κεφάλαιο 2, η κύρια τεχνολογία που οδηγάει το cloud computing είναι η εικονοποίηση. Συγκεκριμένα στην εποχή μας, η κύρια μορφή εικονοποίησης είναι τα containers, μια μορφή εικονοποίησης σε επίπεδο λειτουργικού συστήματος. Παρότι τα containers επιτρέπουν σε ένα φυσικό μηχάνημα να τρέχει μεγάλο αριθμό guest εφαρμογών, η εξειδίκευση των εικόνων τόσο των υποβόσκουσων λειτουργικών συστημάτων όσο και των προγραμμάτων των εικονικών μηχανών είναι περιορισμένη. Για αυτό το λόγο, συχνά παρατηρείται το φαινόμενο όπου ένα εικονικό σύστημα περιέχει λειτουργίες οι οποίες μένουν αναξιοποίητες, καθώς οι μηχανές είναι μοναδικού σκοπου, δηλαδή τρέχουν στο περιβάλλον τους μία μοναδική εφαρμογή με συγκεκριμένες ανάγκες. Αυτό έχει ως αποτέλεσμα, ανάμεσα σε άλλα, να μένει αναξιοποίητο μέρος του καταναλισκούμενου χώρου, καθώς και να μειώνει την ασφάλεια των συστημάτων με εφαρμογές οι οποίες ναι μεν υπάρχουν στην εικονική μηχανή λόγω του λειτουργικού συστήματος γενικού σκοπού, αλλά δεν είναι κατάλληλα ρυθμισμένες. Για την λύση των παραπάνω προβλημάτων προτάθηκαν τα unikernels [3].

Σύμφωνα με το `http://unikernel.org/` τα unikernels είναι:

Εξειδεικευμένες εικόνες μηχανής με μοναδικό χώρο διευθύνσεων, που δημιουργούνται από λειτουργικά συστήματα οργανωμένα σε βιβλιοθήκες.

Αποδομώντας αυτό τον ορισμό, τα unikernels έχουν τα εξής χαρακτηριστικά:

- **Εικόνες ειδικού σκοπού:** Τα unikernels φτιάχνονται εξειδικευμένα για ένα σκοπό. Αυτό σημαίνει ότι τρέχουν μία εφαρμογή, για την οποία και έχουν κατάλληλα δομηθεί.
- **Μοναδικός χώρος διευθύνσεων**: Σε αντίθεση με τα μονολιθικά λει-

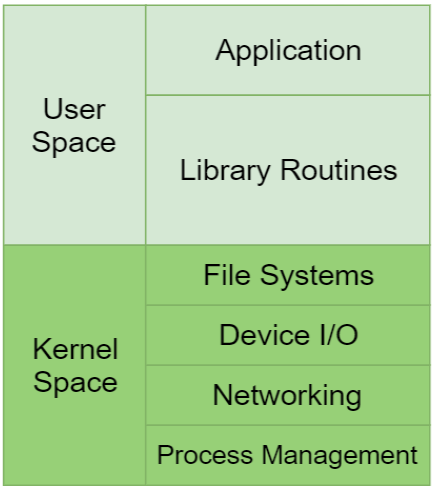

Figure 3.1: Μονολιθικό stack

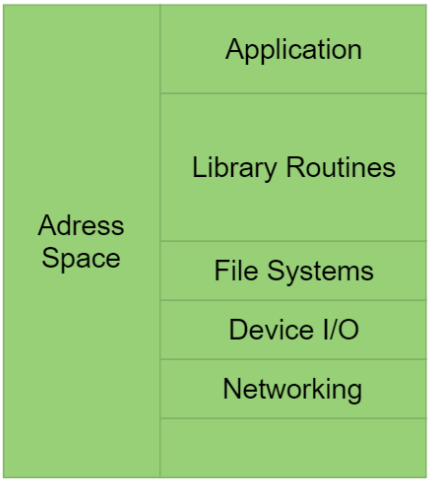

Figure 3.2: Unikernel stack

τουργικά συστήματα ο χώρος διευθύνσεων ενός unikernel δεν διαχωρίζεται σε userspace και kernelspace. Αντίθετα, όλος ο χώρος διευθύνσεων είναι κοινός και προσβάσιμος τόσο από τον πυρήνα, όσο και από την εφαρμογή χρήστη όπως φαίνεται στην εικόνα 3.2 σε αντίθεση με το μονολοιθικό μοντέλο δόμησης του λειτουργικού συστήματος (εικόνα 3.1).

- **Library Operating System (libOS)**: Η δομή αυτών των λειτουργικών συστημάτων επιτρέπει την επιλογή μόνο των απαραίτητων συστατικών τους που χρειάζεται μια εφαρμογή για να τρέξει.

Αρχικά αυτό σημαίνει ότι η εφαρμογή και το λειτουργικό σύστημα γίνονται packaged σε μία εικόνα. Δηλαδή μια εικόνα στην οποία υπάρχει μια πλήρης εφαρμογή και μόνο οι υπηρεσίες του λειτουργικού συστήματος που χρειάζονται για την εκτέλεσή της.

Δεύτερον, ολες οι διεργασίες που τρέχουν μοιράζονται μία κοινή μνήμη μεταξύ τους. Η έλλειψη διαχωρισμού οφείλεται στην τάση να έχουμε μονοδιεργασιακές μηχανές (single purpose machines) στα cloud centers και άρα η ύπαρξη διαφορετικών χώρων διευθύνσεων δεν χρειάζεται. Η έλλειψη διαχωρισμού μεταξύ kernelspace και userspace σημαίνει ότι οι υπηρεσίες του guest λειτουργικού συστήματος δεν καλούνται πλέον μέσω κλήσεων συστήματος, αλλά απευθείας ως κλήσεις συνάρτησεων. Αυτό σημαίνει ότι οποιαδήποτε εφαρμογή τρέχει σε ένα unikernel μπορεί να αγγίξει τον χώρο διευθύνσεων της ίδιας της εφαρμογής, αλλά και του πυρήνα, αφού τον μοιράζονται.

Τέλος, για να επιτευχθεί η κατάλληλη τμηματοποίηση των υπηρεσιών, πυρήνες όπως αυτός του NetBSD ξαναγράφθηκαν έτσι ώστε οι οδηγοί συσκευών τους να έχουν την μορφή βιβλιοθηκών. Καταλήγουμε έτσι στα unikernels, ένα μονοπυρήνα, ο οποίος μειώνοντας το μέγεθος του και εξειδικεύοντας την λειτουργία του, μειώνει τον χρόνο

εκκίνησής του, αυξάνει την ασφάλεια του αφού μειώνεται η επιφάνεια επίθεσης του και απομονώνεται από τις υπόλοιπες εικονικές μηχανές και τον host πλήρως όπως μια πλήρως εικονικοποιημένη σε επίπεδο υλικού μηχανή. Ακόμη και να μπορούσε κάποιος να εκτελέσει μια κακόβουλη επίθεση σε ένα unikernel, η έλλειψη υπηρεσιών και η πλήρης απομόνωση μέσω του hypervisor από τις υπόλοιπες μηχανές δεν θα πρόσφεραν περαίτερω δυνατότητες επίθεσης εκτός του unikernel.

Τα κύρια πλεονεκτήματα που προσφέρουν τα unikernels είναι:

- **Μέγεθος:** Κατά την μεταγλώττιση μιας εφαρμογής ως unikernel, συνδέονται στο εκτελέσιμο μόνο οι υπηρεσίες του πυρήνα που απαιτούνται. Αυτό έχεις ως αποτέλεσμα ο περιττός κώδικας που υπάρχει σε μια εικονική μηχανή να μην συμπεριλαμβάνεται μειώνοντας πολύ το μέγεθος ενός unikernel μέχρι και σε τάξεις μικρότερες του megabyte.

- **Ταχύτητα:** Όπως αναφέρεται προηγουμένως, τα unikernels δεν περιέχουν ολόκληρο πυρήνα, αλλά μόνο τα απαραίτητα κομμάτια του. Επειδή κατά την εκκίνησή τους δεν χρειάζεται να αρχικοποιηθεί κάθε λειτουργία του πυρήνα, ο χρόνος εκκίνησης τους είναι στις τάξεις των milliseconds.

- **Ασφάλεια:** Ομοίως με από πάνω, η μη ύπαρξη κώδικα στην εικονική μηχανή που είναι άσχετος με την εφαρμογή αυξάνει κατά πολύ την ασφάλεια. Αυτό προκύπτει από το γεγονός ότι υπηρεσίες του πυρήνα και proprietary εφαρμογές του λειτουργικού συστήματος άσχετες με την εφαρμογή δεν αφήνονται πλέον εκτεθειμένες, αφού δεν συμπεριλαμβάνονται καν στην εικόνα που τρέχει η μηχανή.

Βέβαια, όλα τα προαναφερθέντα σημαίνουν αλλαγές στην δομή των λειτουργικών συστημάτων, ώστε να απομακρυνθούνε περαιτέρω από την μονολιθική δομή και να μπορούν να χρησιμοποιηθούν ως βιβλιοθήκες. Επίσης, οι εφαρμογές να ξαναγραφούν λαμβάνοντας υπόψη το γεγονός ότι πλέον είναι μονοδιεργασιακές και ο χώρος διευθύνσεων είναι κοινός με τον πυρήνα. Παρακάτω θα αναφερθούν μερικά από τα unikernel frameworks που χρησιμοποιούνται για την δημιουργία και εκτέλεση εικόνων unikernels.

3.1 Rump kernels και Rumprun

Το Rumprun είναι ένα unikernel framework χτισμένο πάνω στο rump kernel του NetBSD [4]. Τα rump kernels είναι μια δομή πυρήνα, η οποία ακολουθεί την λογική των anykernels, δηλαδή οι οδηγοί συσκευών έχουν γραφεί, ώστε να μην εξαρτώνται

από το υποβόσκων λειτουργικό σύστημα. Η δομή των rump kernels (εικόνα 3.3) αποτελείται από την βάση, τα factions και τους οδηγούς συσκευών.

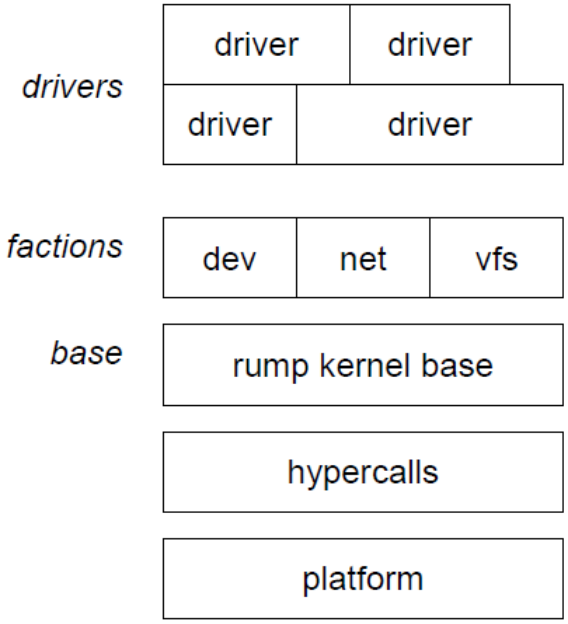

Figure 3.3: Rump kernel stack

Τα rump kernels δεν είναι ανεξάρτητες οντότητες, δηλαδή δεν μπορούν να τρέξουν απευθείας πάνω στο υλικό. Πιο συγκεκριμένα, τα rump kernels φτιάχτηκαν για να τρέχουν στο userspace του NetBSD, καθώς αποτελούν ένα εργαλείο αποσφαλμάτωσης των οδηγών συσκευών του. Αυτό σημαίνει ότι χρειάζονται υποστήριξη χαμηλού επιπέδου για να τρέξουν. Η υποστήριξη αυτή μπορεί είτε να είναι ένας bare metal hypervisor φτιαγμένος ειδικά για τον σκοπό που εξυπηρετεί, είτε ένα λειτουργικό σύστημα γενικού σκοπού.

Η βάση των rump kernels, η οποία είναι υπεύθυνη για την επικοινωνία με τον hypervisor είναι ανεξάρτητη των ανώτερων στρωμάτων και περίεχει την βασική υποστήριξη όπως την κατανομή μνήμης για τις διάφορες λειτουργίες του πυρήνα.

Πάνω από την βάση του rump kernel βρίσκονται τα factions. Τα factions dev, net και vfs αποτελούν τις συσκευές, το υποσύστημα δικτύου και το εικονικό σύστημα αρχείων. Αυτά μεταξύ τους είναι ανεξάρτητα, ώστε η ύπαρξη του ενός να μην απαιτεί την ύπαρξη των άλλων.

Το μεγαλύτερο μέρος ενός λειτουργικού συστήματος είναι οι οδηγοί συσκευών. Σε αυτή την δομή, οι οδηγοί συσκευών επιλέγονται από το NetBSD ως έχουν. Σε ειδικές περιπτώσεις όμως μπορεί να χρησιμοποιηθούν τροποποιημένοι, ώστε να μην χρειάζεται κάποιο faction. Για παράδειγμα, ένας οδηγός συσκευής συνήθως έχει πρόσβαση στην

συσκευή μέσω του /dev, το οποίο υπονοεί την ύπαρξη ενός εικονικού συστήματος αρχείων. Σε περιπτώσεις, όμως, όπου οι πόροι είναι περιορισμένοι, ο οδηγός συσκευής μπορεί να γραφεί ώστε να χρησιμοποιεί απευθείας τις διεπαφές της συσκευής, αφαιρόντας έτσι την ανάγκη για το vfs faction.

Αυτή η τμηματοποίηση, από την οποία πήρε και το όνομά του, επιτρέπει την επιλογή μόνο των factions και των οδηγών που απαιτεί η εφαρμογή προς εκτέλεση, δημιουργώντας ένα πολύ μικρό πυρήνα.

Αφού εξηγήθηκαν τα rump kernels, επανερχόμαστε στο Rumprun. Το Rumprun είναι ένα unikernel framework, με κύριο χαρακτηριστικό ότι επιτρέπει σε εφαρμογές που συμμορφώνονται στο POSIX να τρέχουν χωρίς αλλαγές. Όπως και τα rump kernels για να τρέξει χρειάζεται έναν hypervisor, με τους πιο σύνηθες να είναι ο Xen και ο KVM, οι οποίοι είναι hypervisors τύπου 1, δηλαδή πάνω σε bare metal.

Η σχέση του rump kernel και του Rumprun φαίνεται στην εικόνα 3.4. Για μια συγκεκριμένη εφαρμογή διαλέγονται μόνο τα factions και οι drivers, από τον κώδικα των rump kernels στο NetBSD, που χρειάζονται για την λειτουργία της και δημιουργείται ο πυρήνας (rump kernel) του Rumprun.

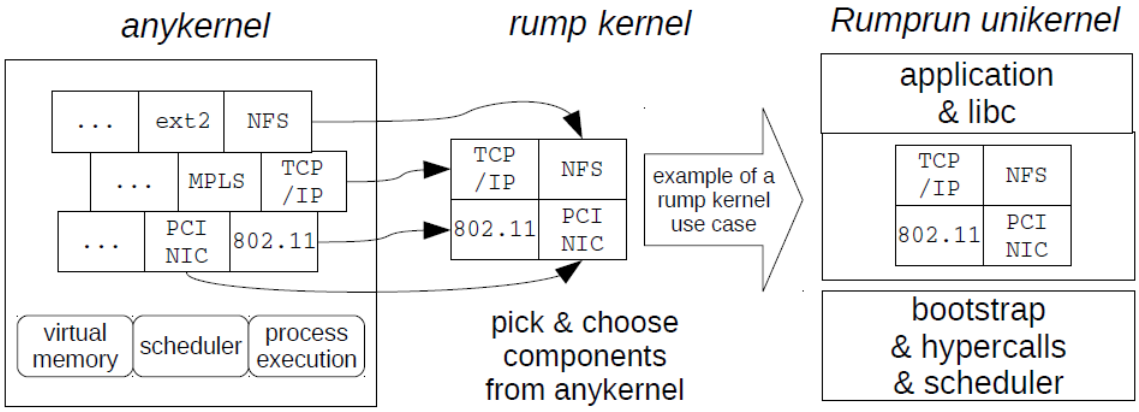

Figure 3.4: Η επιλογή των κατάλληλων κομματιών από το NetBSD δημιουργεί τον rump kernel του Rumprun

Το Rumprun, όπως και όλα τα unikernels, δεν υποστηρίζει παραπάνω της μίας διεργασίας στην λογική που αναφέρθηκε περί cloud computing, αλλά υποστηρίζει POSIX-νήματα τα οποία εκτελεί σε σειρά με βάση τον χρονοδρομολογητή του. Ο χρονοδρομολογητής που διαθέτει είναι συνεργατικός, δηλαδή ένα νήμα τρέχει μέχρι να σταματήσει είτε γιατί τελείωσε την εργασία του είτε επειδή έκανε κλήση οι οποία είναι blocking, όπως είναι το read/write. Σε αντίθεση δηλαδή με τους προληπτικούς χρονοδρομολογητές, οι οποίοι επιπρόσθετα σταματάνε τα νήματα μετά το πέρας κάποιου χρόνου ή άμα γίνει κάποια διακοπή. Το παραπάνω έρχεται σε συνεργία με το γεγονός ότι

το Rumprun δεν υποστηρίζει διακοπές, καθώς δεν δύναται να αλλάξει το περιβάλλον εκτέλεσης του νήματος με το περιβάλλον εκτέλεσης μιας διακοπής. Συγκεκριμένα, ένα νήμα ανατίθεται σε μια εικονική μονάδα επεξεργασίας (CPU) και τρέχει σε αυτή μέχρι να την αφήσει για τους λόγους που αναφέρθηκαν. Αφού ένα νήμα πάρει μια εικονική μονάδα επεξεργασίας, την καταλαμβάνει μόνιμα και δεν αλλάζει μέχρι να τερματίσει. Οποιοδήποτε άλλο νήμα και να δημιουργηθεί δεν μπορεί να τρέξει, εκτός αν υπάρχουν ελεύθερες εικονικές μονάδες επεξεργασίας, όπως σε πολυεπεξεργαστικά συστήματα. Επειδή όμως κάθε νήμα κατά την εκτέλεσή του δεν επιτρέπει μέσω μηχανισμού κλειδώματος άλλα νήματα να τρέξουν, όλα τα άλλα νήματα πρέπει να περιμένουν να τελειώσει την εκτέλεσή του ή να την παύσει οικειοθελώς.

Επίσης, το Rumprun δεν διαθέτει εικονική μνήμη. Αυτό ήταν σχεδιαστική επιλογή των δημιουργών καθώς η ύπαρξη εικονικής μνήμης θα μείωνε πολύ την απόδοση των μηχανών, θα αύξανε την χρήση πόρων και θα αύξανε και την πολυπλοκότητα της υλοποίησής του. Ακόμη, η ύπαρξη εικονικής μνήμης δεν θα επέτρεπε την δημιουργία εξειδικευμένων οδηγών συσκευών, καθώς δεν θα είχαν απευθείας και ελεύθερη πρόσβαση στην μνήμη του πυρήνα.

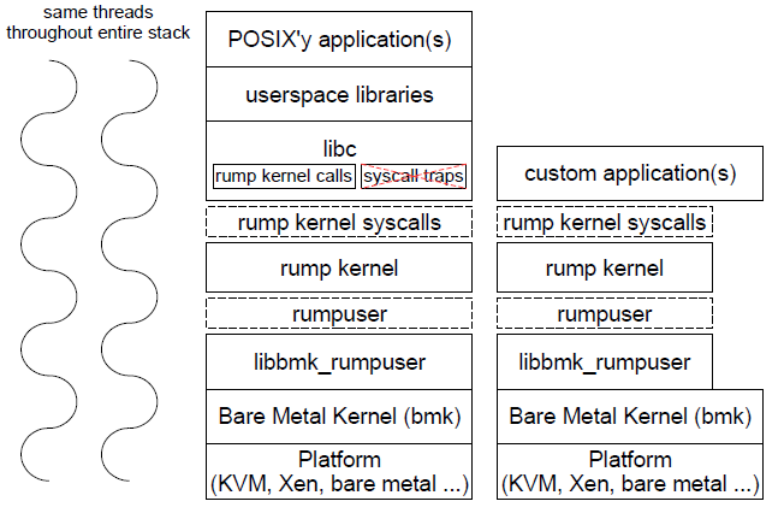

Figure 3.5: Αριστερά φαίνεται η δομή του Rumprun για POSIX εφαρμογές που περιέχει την τροποποιημένη libc. Δεξιά φαίνεται το stack για custom εφαρμογές, όπου έχουν την δυνατότητα να καλέσουν συναρτήσεις του πυρήνα απευθείας.

Η δομή του Rumprun φαίνεται στην είκονα 3.5. Οι κλήσεις συστήματος έχουν αντικατασταθεί από τις rump kernel κλήσεις, οι οποίες χρησιμοποιούνται για να προσπελαστεί ο πυρήνας. Βέβαια, δεν υπάρχει διαχωρισμός μεταξύ kernel και userspace. Αυτό σημαίνει ότι δεν υπάρχει πραγματική αλλαγή σε εκτέλεση πυρήνα καθώς η εκτέλεση σε πλαίσιο πυρήνα ισοδυναμεί με την κλήση μιας απλής συνάρτησης, το οποίο επιταχύνει

την συνολική εκτέλεση των προγραμμάτων. Επίσης, τα rump kernels που εκτελούν τις κλήσεις συστήματος δεν αποτελούν πραγματικό πυρήνα, για αυτό κάποιες λειτουργίες λείπουν. Η κάλυψη αυτών των λειτουργιών γίνεται σε κατώτερο στρώμα από τον bare metal kernel, ο οποίος είναι υπεύθυνος και για την επικοινωνία με τον hypervisor.

3.2 ClickOS

Επειδή το κόστος διαχείρισης και αναβάθμισης hardware-based middleboxes είναι μεγάλο, καθώς και η δημιουργία εφαρμογών που λειτουργούν ως middleboxes είναι πολύπλοκη, ενώ απαιτεί μικρό μέρος των υπηρεσιών του λειτουργικού, δημιουργήθηκε το ClickOS. Το ClickOS [5], [6] είναι ένα unikernel framework φτιαγμένο για λειτουργία ως middlebox. Συγκεκριμένα, είναι μια πλατφόρμα λογισμικού πάνω στον hypervisor Xen που τρέχει εφαρμογές που χρησιμοποιούν την αρχιτεκτονική Click.

Το ClickOS εκμεταλλεύεται την τεχνική του para-virtualization, αλλάζοντας το guest λειτουργικό σύστημα, για να μειώσει τις καθυστερήσεις που δημιουργούνται από το full virtualization. Επίσης, οι προσβάσεις σε όλες τις κάρτες δικτύου γίνονται μέσω του οδηγού συσκευής που παρέχει ο Xen, ο οποίος είναι ανεξάρτητος υλικού, μεταφέροντας την επικοινωνία με την πραγματική κάρτα δικτύου στο υποβόσκων λειτουργικό σύστημα λύνοντας το πρόβλημα του ποιου οδηγού συσκευής, ανάμεσα σε μεγάλο εύρος, πρέπει να χρησιμοποιηθεί. Όσον αφορά τις ανάγκες της μηχανής για διαχείριση μνήμης και χρονοδρομολόγηση, χρησιμοποιείται το μινιμαλιστικό λειτουργικό σύστημα που, επίσης, παρέχει ο Xen, το MiniOS. Το MiniOS έχει μοναδικό χώρο διευθύνσεων και συνεργατικό χρονοδρομολογητή.

Κάθε εικονικό μηχάνημα του ClickOS αποτελείται από μια Click εφαρμογή που τρέχει πάνω στο MiniOS. Κατά την εκκίνηση της εικονικής μηχανής, αρχικοποιείται ένα νήμα ελέγχου, το οποίο έπειτα παρακολουθεί για αλλαγές και όταν χρειαστεί δημιουργεί στην ίδια μηχανή ένα νήμα το οποίο τρέχει την Click εφαρμογή.

Τέλος, η ανάπτυξη εφαρμογών για το Click είναι αρκετά απλή, καθώς το MiniOS αποτελεί ένα χαμηλού επιπέδου περιβάλλον και για αυτό η δημιουργία των Click εφαρμογών μπορεί να γίνει σε ένα userspace που το υποστηρίζει και έπειτα να εισαχθούν στην εικόνα της εικονικής μηχανής κατά το build.

3.3 IncludeOS

Το IncludeOS [7] είναι ένα library operating system για υπηρεσίες cloud. Είναι γραμμένο από το μηδέν σε C++ και είναι φτιαγμένο για να εκτελεί μία μοναδική εργασία. Φτιάχτηκε για να επιτρέπει την εύκολη δημιουργία εικονικών μηχανών για

την εκτέλεση προγραμμάτων σε C++, τα οποία ενσωματώνονται στην εικονική μηχανή κατά την δημιουργία της.

Όσον αφορά την αρχιτεκτονική του, το IncludeOS ακουλουθεί την αρχή μηδενικού πλεονάσματος. Δηλαδή, κατά την δημιουργία εικονικών μηχανών, καμία υπηρεσία του λειτουργικού συστήματος δεν συμπεριλαμβάνεται άμα δεν είναι απαραίτητη για το πρόγραμμα που θα εκτελεστεί. Για να το πετύχει αυτό χρησιμοποιεί τον μηχανισμό των linkers που χρησιμοποιείτε έτσι κι αλλιώς από τους καινούριους μεταγλωττιστές. Όλες οι βιβλιοθήκες του λειτουργικού συστήματος αρχικά μετατρέπονται σε αρχεία αντικειμένου και έπειτα ο linker χρησιμοποιεί μόνο αυτά τα οποία χρειάζονται για να δημιουργήσει το τελικό εκτελέσιμο.

Τον μόνο οδηγό που έχει το IncludeOS είναι ένας VirtioNet οδηγός συσκευής. Έτσι, ο hypervisor δεν χρειάζεται να εικονοποιήσει κάποια συγκεκριμένη συσκευή, αλλά τοποθετεί δεδομένα απευθείας σε μια ουρά στην μνήμη που μοιράζεται ο guest με τον host, ο οποίος με την σειρά του τα επεξεργάζεται.

Το IncludeOS χειρίζεται τις διακοπές σε δεύτερο χρόνο. Αυτό σημαίνει ότι οι χειριστές των διακοπών αυξάνουν ατομικά ένα μετρητή και έπειτα αφήνουν τον περαιτέρω χειρισμό στην κύρια event-loop, η οποία τις εκτελεί όταν μείνει αδρανής. Επίσης, οι συναρτήσεις εισόδου/εξόδου γίνονται ασύγχρονα, ώστε να μην κάνουν block. Και στις δύο περιπτώσεις αποφεύγεται η αλλαγή πλαισίου εκτέλεσης.

3.4 MirageOS

Το MirageOS είναι ένα unikernel framework φτιαγμένο εξολοκλήρου σε OCaml εκτός των πολύ χαμηλών επιπέδων του και δημιουργεί εκτελέσιμες εικόνες εικονικών μηχανών Xen από προγράμματα γραμμένα σε OCaml.

Η OCaml είναι μια υψηλού επιπέδου γλώσσα προγραμματισμού που συνδυάζει functional, object-oriented και imperative προγραμματισμό, ενώ προσφέρει ένα ασφαλή στατικό σύστημα τύπων. Το στατικό σύστημα τύπων βοηθάει στην αποφυγών σφαλμάτων τύπου κατά την εκτέλεση μιας εφαρμογής και την μείωση των διαρροών μνήμης, αφού μειώνονται τα runtime errors που θα δημιουργούνταν κατά την ασυμφωνία τύπων των μεταβλητών κατά την εκτέλεση μιας εφαρμογής.

Παρότι η OCaml είναι μια ασφαλή γλώσσα προγραμματισμού, δεν συνηθίζεται να χρησιμοποιείται για προγραμματισμό συστημάτων. Αυτό σημαίνει ότι κατά την επιλογή της, πάρθηκε και η απόφαση να ξαναγραφεί σε OCaml μεγάλο μέρος των συστημάτων ενός λειτουργικού συστήματος. Επίσης, επειδή η σειριακή εκτέλεση νήματος της OCaml είναι γρήγορη, πάρθηκε η απόφαση κάθε εικονική μηχανή να τρέχει σε μία εικονική μονάδα επεξεργασίας με την πολυπύρηνη επεξεργασία να γίνεται με πολλαπλές

μηχανές πάνω σε ένα Xen hypervisor.

Το MirageOS έχει ένα 64-bit χώρο διευθύνσεων, δεν υποστηρίζει πολλαπλές διεργασίες και δεν χρησιμοποιεί προληπτικό χρονοδρομολογητή.

3.5 OSᵛ

Το OSᵛ [8] είναι ένα λειτουργικό σύστημα φτιαγμένο για την δημιουργία εικονικών μηχανών στο cloud. Ο σκοπός του είναι η επιτάχυνση και η μείωση της περιττότητας στις εικονικές μηχανές, είτε για υπάρχουσες εφαρμογές, είτε για εφαρμογές φτιαγμένες πάνω στο API του OSᵛ.

Τα κύρια κριτήρια με τα οποία φτιάχτηκε είναι:

- να τρέχει υπάρχουσες cloud εφαρμογές,
- να τις τρέχει πιο γρήγορα από συμβατικά λειτουργικά συστήματα,
- να δημιουργεί αρκετά μικρές εικόνες μηχανών και αυτές να εκκινούν αρκετά γρήγορα, ώστε να μπορούν να χρησιμοποιηθούν ως IaaS,
- να γίνει χρήση του API του OSᵛ για να επιτευχθεί καλύτερη απόδοση των εφαρμογών,
- να είναι μια πλατφόρμα για συνεχή έρευνα πάνω στα λειτουργικά συστήματα εικονικών μηχανών, καθώς το OSᵛ είναι ανοιχτού κώδικα λειτουργικό σύστημα γραμμένο σε C++11.

Στην φιλοσοφία των unikernels, το OSᵛ τρέχει μία εφαρμογή και έχει ένα κοινό χώρο διευθύνσεων και πίνακα σελιδών που μοιράζονται όλα τα νήματα και ο πυρήνας, που επιταχύνει την αλλαγή πλαισίου και την κλήση συναρτήσεων πυρήνα.

Για να πετύχει συμβατότητα με ήδη υπάρχουσες εφαρμογές, το OSᵛ παρέχει μεγάλο μέρος των προγραμματιστικών διεπαφών των Linux. Επίσης, η σύνδεση του ABI γίνεται μέσω ενός δυναμικού ELF linker που βρίσκεται στον πυρήνα του OSᵛ και όποτε γίνεται μια κλήση στο ABI - από π.χ. την βιβλιοθήκη C των Linux -, καλείται η κατάλληλη συνάρτηση που έχει υλοποιηθεί στο OSᵛ. Δεν έχουν υλοποιηθεί παρόλαυτα κλήσεις όπως το fork και execv, καθώς το μοντέλο του OSᵛ είναι μονοδιεργασιακό.

Για τους οδηγούς συσκευών, το OSᵛ χρησιμοποιεί ένα μικρό σετ που υποστηρίζεται από διαδεδομένους hypervisors. Μάλιστα, η 64-bit x86 αρχιτεκτονική υποστηρίζεται στους KVM, Xen, VMware και VirtualBox hypervisors, καθώς και στο cloud των Amazon EC2 (Xen) και Google GCE (KVM).

Συνήθως τα library λειτουργικά συστήματα χρησιμοποιούν ένα απλό mapping του χώρου διευθύνσεων της φυσικής μνήμης. Το OSᵛ από την άλλη χρησιμοποιεί

εικονική μνήμη όπως τα λειτουργικά συστήματα γενικού σκοπού. Αυτό το χαρακτηριστικό οφείλεται στην απαίτηση των αρχιτεκτονικών όπως η x86_64, καθώς και στην ανάγκη του προτύπου POSIX για πολλές λειτουργίες του. Όμως, επειδή υποστηρίζει εικονικές μηχανές με μία διεργασία, η διαγραφή σελιδών δεν υποστηρίζεται.

Ένα πρόβλημα που παρατηρείται να μειώνει την απόδοση των εικονικών μηχανών είναι τα spinlocks. Σε αντίθεση με μια φυσική μονάδα επεξεργασίας, μια εικονική μονάδα επεξεργασίας μπορεί να σταματήσει να τρέχει οποιαδήποτε στιγμή, καθώς μπορεί για παράδειγμα ο hypervisor να αλλάξει τον guest που τρέχει σε αυτή. Αυτό σημαίνει ότι άμα μια CPU σταματήσει με κάποιο spinlock κρατημένο, οι άλλες εικονικές CPU που μπορεί να είναι ενεργές και περιμένουν το lock της σταματημένης CPU σπαταλάνε κύκλους. Για να αποφύγει αυτό το πρόβλημα, το OS^{v} δεν χρησιμοποιεί καθόλου spinlocks. Για να το πετύχει αυτό, αρχικά μεταφέρει όλη την δουλειά του πυρήνα σε νήματα, τα οποία χρησιμοποιούν mutexes, τα οποία τα βάζουν σε κατάσταση sleep αντί για busy wait. Επίσης, υλοποιήθηκε ο ίδιος ο μηχανισμός mutex, ώστε να μην χρησιμοποιεί γενικά spinlocks. Τέλος, επειδή ο χρονοδρομολογητής δεν μπορεί να τρέξει σε νήμα, το OS^{v} χρησιμοποιεί αλγόριθμους χωρίς κλείδωμα και ουρές εκτέλεσης ανά επεξεργαστική μονάδα.

Ο δρομολογητής του OS^{v} είναι προληπτικός, δηλαδή τα νήματα αλλάζουν είτε μόνα τους, όπως άμα περιμένουν σε κάποια κλήση ή ξυπνόντας ένα άλλο νήμα, είτε λόγω κάποιου timer ή διακοπής μεταξύ των CPU. Για την αποφυγή διακοπής τους, τα νήματα μπορούν να απενεργοποιήσουν την προληπτική δρομολόγησή τους. Σε αυτή την περίπτωση, αν ένα νήμα ζητήσει να δρομολογηθεί, τότε περιμένει μέχρι το νήμα εκτέλεσης να ενεργοποιήσει ξανά την δρομολόγηση και δρομολογείται τότε. Ο δρομολογητής του OS^{v} χρησιμοποιεί και ένα ισορροπιστή φόρτου που τρέχει ανά 10 δευτερόλεπτα. Ο ισσοροπιστής φόρτου μπορεί να αλλάξει την ουρά εκτέλεσης στην οποία περιμένει ένα νήμα, τοποθετόντας το στην ουρά εκτέλεσης μιας CPU στην οποία περιμένουν λιγότερα νήματα.

Ένα ακόμη σημαντικό χαρακτηριστικό του δρομολογητή είναι ότι δεν αναγνωρίζει τα νήματα τα οποία δεν είναι εκτελέσιμα, επομένως δεν έχει κόστος η διατήρηση νημάτων που τρέχουν σπάνια, όπως ο ισορροπιστής φόρτου που αναφέρθηκε παραπάνω.

Ένα σημαντικό πρόβλημα όλων των unikernels είναι η ανάγκη αλλαγής των υπάρχοντων προγραμμάτων, ώστε να μπορούν υπάρχουσες εφαρμογές να τρέχουν αποδοτικά πάνω τους. Το OS^{v} εστίασε σε δύο σημεία, την βελτίωση εκτέλεσης της Java Virtual Machine (JVM), καθώς αυτό σημαίνει ότι όλες οι εφαρμογές που χρησιμοποιούν αυτό το περιβάλλον θα δεχτούν βελτίωση στην εκτέλεσή τους, και την μείωση του overhead του Linux API που συνήθως οφείλεται στον διαχωρισμό του χώρου πυρήνα από τον χώρο χρήστη, ο οποίος δεν υπάρχει στο OS^{v}.

Ένα χαρακτηριστικό του OS^v είναι το shrinker API. Το shrinker API επιτρέπει στην εφαρμογή ή ένα κομμάτι του λειτουργικού συστήματος να καταχωρήσει συναρτήσεις που θα κληθούν όταν το σύστημα είναι χαμηλό σε μνήμη. Έτσι, αντί να υπάρχει διαχειριστής μνήμης, κάθε κομμάτι του συστήματος μπορεί να διαχειριστεί την έλλειψη μνήμης μόνο του, καθώς σε ένα μονοδιεργασιακό σύστημα δεν υπάρχει conflict για την μνήμη μεταξύ εφαρμογών.

Ένας άλλος μηχανισμός του OS^v είναι το JVM Ballon, ο οποίος υπολογίζει αυτόματα την διαθέσιμη μνήμη που θα δωθεί στο JVM. Η προσέγγιση του OS^v είναι η ανάθεση σχεδόν όλης της διαθέσιμης μνήμης στο JVM. Μόλις χρειαστεί το JVM παραπάνω μνήμη, το OS^v την παρέχει στο heap του μέσω του JNI και η αναφορά στο αντικείμενο της μνήμης γίνεται από το JNI, ώστε να μην χαθεί σε κάποια συλλογή σκουπιδιών.

3.6 Graphene OS

Το Graphene OS [9] είναι ένα library λειτουργικό σύστημα (libOS), το οποίο είναι συμβατό με τα Linux. Σχεδιάστηκε για την υποστήριξη πολυδιεργασιακών εφαρμογών, καθώς τα περισσότερα unikernel frameworks υποστηρίζουν μόνο την μονοδιεργασιακή εκτέλεση εφαρμογών. Στο Graphene πολλαπλά στιγμιότυπά libOS λειτουργούν συνεργατικά για την υποστήριξη των κλήσεων POSIX, ενώ στην εφαρμογή φαίνεται ένα μοναδικό λειτουργικό σύστημα.

Οι κύριες σχεδιαστικές επιλογές του Graphene OS είναι η εφαρμογή της έννοιας των διεργασιών και της δρομολόγησης σε libOS περιβάλλον. Για την υποστήριξη μιας εφαρμογής το Graphene OS δημιουργεί πολλαπλά στιγμιότυπα libOS, σε καθένα από τα οποία φέρεται σαν ξεχωριστή διεργασία. Μια συλλογή από στιγμιότυπα που λειτουργούν συνεργατικά υλοποιούν και το πρότυπο POSIX για τις ανάγκες μιας εφαρμογής. Η επικοινωνία μεταξύ των συνεργαζόμενων στιγμιοτύπων γίνεται μέσω remote procedure call (RPC) και επεκτείνοντας στο ελάχιστο δυνατό το ABI του host. Στην εικόνα 3.6 φαίνεται η δομή libOS στιγμιοτύπων πάνω σε ένα host λειτουργικό σύστημα.

Το Graphene OS πετυχαίνει απομόνωση μεταξύ των libOS στιγμιοτύπων μέσω ενός έμπιστου reference monitor. Ο reference monitor είναι υπεύθυνος για την εκτέλεση των στιγμιοτύπων του libOS, καθώς και παρεμβαίνει σε κάθε κλήση συστήματος του στιγμιοτύπου, η οποία επηρεάζει κάτι που βρίσκεται εκτός του δικού του χώρου διευθύνσεων. Για να πετύχει την επικοινωνία το Graphene ορίζει τα sandboxes. Κάθε sandbox είναι ένα σύνολο libOS στιγμιοτύπων που θεωρούνται μεταξύ τους trusted και μπορούν να επικοινωνήσουν μέσω RPC. Στιγμιότυπα σε διαφορετικά sanboxes δεν μπορούν να επικοινωνήσουν με κάποιο τρόπο.

Η επικοινωνία μεταξύ στιγμιοτύπων του libOS στο Graphene γίνεται μέσω του

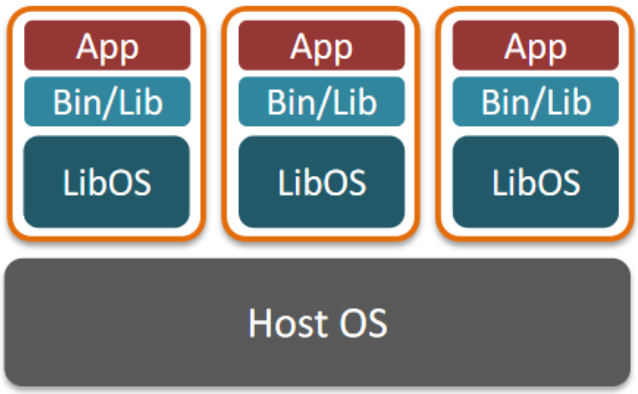

Figure 3.6: Δομή libOS στιγμιοτύπων

host ABI, το οποίο περιέχει 43 συναρτήσεις που μεταφράζονται μέσω του πίνακα PAL. Στην περίπτωση των Linux οι συναρτήσεις αυτές μεταφράζονται σε κλήσεις συστήματος, στις οποίες περιλαμβάνονται κλήσεις σχετικές με signals και δημιουργία αντιγράφων του στιγμιοτύπου όπως η fork. Αυτές οι κλήσεις επιτρέπουν την επικοινωνία του guest με τον host μέσα από συγκεκριμένες διόδους. Οι κλήσεις που τροποποιούν σημεία εξωτερικά της μηχανής γίνονται μέσω του έμπιστου reference monitor. Αυτός τρέχει τις εφαρμογές και κατά την αρχικοποίησή τους τοποθετεί ένα φίλτρο κλήσεων συστήματος, ο οποίος παρεμβάλλει στις επιτρεπόμενες κλήσεις, ώστε να διασφαλίζει την απομόνωσή. Ο reference monitor περιέχει ένα σύνολο από seccomp κανόνες. Όταν ένα στιγμιότυπο καλεί μια κλήση συστήματος, όπως για παράδειγμα κατά την αποστολή ενός σήματος, τότε παρεμβαίνει και ελέγχει αν η κλήση που εκτελείται γίνεται προς trusted στιγμιότυπο.

Αυτό το σύνολο ABI προσπαθεί να μειώσει τον όγκο του state που διατηρεί ο πυρήνας για την εφαρμογή αποσυμπλέκοντάς τα. Παρόλαυτά, μια εφαρμογή που τρέχει σε Graphene μπορεί να τροποποιήσει το state του λειτουργικού της, να αντιγράψει και να φορτώσει το state ενός άλλου μηχανήματος σε αυτή, καθώς και να δημιουργήσει ένα νέο unikernel με state όμοιο με το δικό του. Όμως, δεν μπορεί να επηρεάσει το state ενός άλλου unikernel.

3.7 Intel MPK

Το Intel MPK είναι μια τεχνολογία υλικού της Intel που επιτρέπει permission control ανά νήμα σε ένα σύνολο από σελίδες. Αυτή η τεχνολογία προσφέρει, δηλαδή, τον έλεγχο των δικαιωμάτων ενός νήματος εκτέλεσης στις σελίδες ενός χώρου διευθύνσεων.

Όπως έχει αναφερθεί τα unikernels συνήθως τρέχουν σε ένα μοναδικό χώρο διευ-

θύνσεων, ο οποίος περιέχει τόσο το πρόγραμμα χρήστη όσο και τον κώδικα πυρήνα χωρίς προστασία. Τα unikernels μπορεί να δομούνται από κομμάτια που μεταξύ τους δεν εμπιστεύονται το ένα το άλλο όσον αφορά την ασφάλεια τους ενάντια σε επιθέσεις. Αυτό δημιουργείται από το πρόβλημα ότι άμα ένα κομμάτι ενός unikernel είναι ευάλωτο σε επίθεση, τότε ο επιτιθέμενος θα έπαιρνε έλεγχο ολόκληρου του unikernel. Το Intel MPK μπορεί να διαχωρίσει τις σελίδες του χώρου διευθύνσεων σε ομάδες μέσω εξειδικευμένου υλικού. Έτσι, κάθε νήμα που τρέχει σε κάποιον πυρήνα μπορεί να έχει συγκεκριμένα permissions για ομάδες σελιδών. Συγκεκριμένα, αυτή η τεχνολογία έχει χρησιμοποιηθεί για την απομόνωση του κώδικα πυρήνα από τον κώδικα χρήστη στο RustyHermit [10].

Το MPK χρησιμοποιεί 4 bits κάθε καταχώρισης στον πίνακα σελιδών για να ορίσει το αναγνωριστικό τους όσον αφορά την ομάδα προστατευμένων σελιδών που ανήκουν. Επειδή χρησιμοποιούνται 4 bits, οι ομάδες σελιδών είναι 15 (με το 0 να μην ανήκει σε ομάδα). Με αυτή την τεχνολογία κάθε πυρήνας έχει PKRU registers (32 bits) που δείχνουν τις τιμές των permissions των προστατευμένων σελιδών που έχει ένα νήμα που τρέχει την δεδομένη στιγμή στον πυρήνα. Οι τιμές των permissions είναι read/write, read-only ή no-access. Επομένως, κατά την εκτέλεσή των νημάτων, κάθε φορά που γίνεται πρόσβαση κάποιας σελίδας σε προστατευμένη ομάδα ελέγχεται το permission που δείχνει ο αντίστοιχος PKRU register για τη συγκεκριμένη ομάδα στον πυρήνα που τρέχει το νήμα. Σημαντικό είναι ότι το κόστος αλλαγής της τιμής του PKRU είναι αμελητέο [11].

3.8 Solo5

Το solo5 δεν είναι unikernel framework για την δημιουργία εικονικών μηχανών unikernel, αλλά ένας hypervisor σχεδιασμένος να τρέχει unikernels βασισμένος στον τύπου 1 hypervisor ukvm. Το solo5 προσφέρει υποστήριξη για μια σειρά από library λειτουργικά συστήματα όπως το MirageOS (OCaml), IncludeOS (C++) και το Rumprun (NetBSD). Επίσης, επιτρέπει την γρήγορη εκκίνηση και εκτέλεση unikernels, ενώ προσφέρει απομόνωση μεταξύ των εικονικών μηχανών.

Μια τυπική αρχιτεκτονική για την εκτέλεση unikernels φαίνεται στην εικόνα 3.7. Σε αυτή την δομή, ο πυρήνας των Linux εκθέτει την διεπαφή του εικονοποιημένου υλικού στο userspace μέσω του KVM. Για την εκκίνηση, αρχικοποίηση και διαγραφή της εικονικής μηχανής χρησιμοποιεί έναν hypervisor για unikernels όπως ο ukvm. Ο ukvm προσφέρει υποστήριξη για τα προαναφερθέντα library λειτουργικά συστήματα πάνω σε συστήματα όπως το Linux, NetBSD και OpenBSD σε αρχιτεκτονικές x86_64 και ARM64. Για να φορτώσει ο hypervisor και να εκκινήσει την εικονική μηχανή, ο ukvm επικοινωνεί με το KVM μέσω κλήσεων συστήματος, καθώς και όποτε το

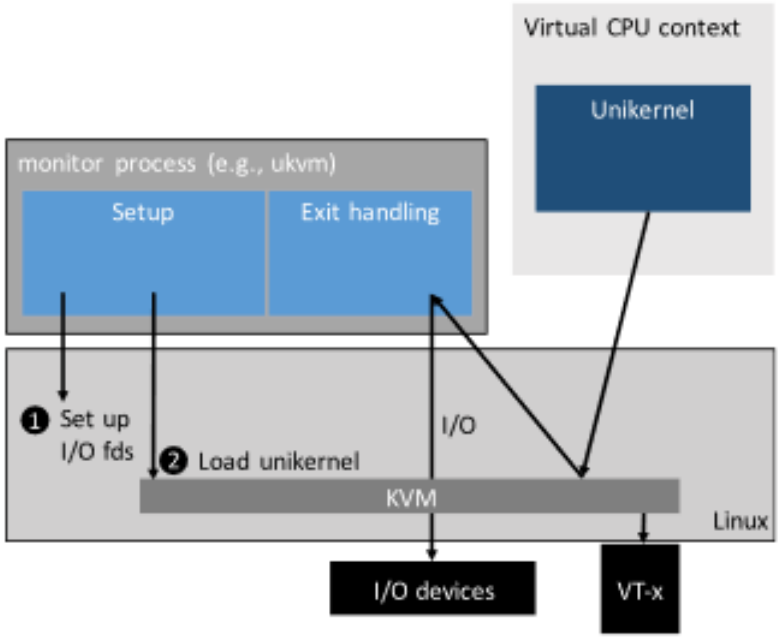

Figure 3.7: Δομή unikernel με ukvm hypervisor

unikernel κάνει exit μέσω κάποιας hypercall. Συγκεκριμένα, το ukvm προσφέρει μόλις δέκα hypercalls στο unikernel, ένα μικρό αριθμό κλήσεων που προσφέρει καλή απομόνωση της εικονικής μηχανής από το πραγματικό σύστημα.

Σε αντίθεση με την τυπική εκτέλση των unikernels, το solo5 χρησιμοποιεί tenders για την εκτέλεση των unikernels ως διεργασίες. Τα tenders του solo5 λειτουργούν ως hypervisors, δηλαδή έχουν τις ίδιες αρμοδιότητες. Η διαφορά είναι ότι το tender του solo5, κατά την εκκίνηση της εικονικής μηχανής, φορτώνει τον κώδικα του unikernel στον δικό του χώρο διευθύνσεων. Η εκκίνηση του unikernel, δηλαδή, είναι απλώς μια κλήση συνάρτησης στην συνάρτηση εισόδου του, παρακάμπτοντας τις κλήσεις συστήματος του προηγούμενου παραδείγματος. Από την πλευρά του, το unikernel πλέον αντί να κάνει VM exits κάθε φορά που καλεί μια hypercall, χρησιμοποιεί την υλοποίησης της αντίστοιχης hypercall στο tender, καθώς αυτές βρίσκονται στον χώρο διευθύνσεών του.

Σε αυτή την δομή το ίδιο το tender τρέχει τον κώδικα της εικονικής μηχανής, αφού πρώτα ορίσει seccomp κανόνες. Οι seccomp κανόνες επιτρέπουν την απόρριψη ή αποδοχή συγκεκριμένων κλήσεων συστήματος με συγκεκριμένα ορίσματα, τα οποία μπορούν να είναι γνωστά πριν την εκτέλεση της εικονικής μηχανής, όπως για παράδειγμα ένας δείκτης αρχείου μιας συσκευής δικτύου που θα χρησιμοποιήσει η εικονική μηχανή. Έτσι πετυχαίνεται απομόνωση σε επίπεδα εικονικής μηχανής, ενώ στην πραγματικότητα η εικονική μηχανή είναι μια ακόμη διεργασία του συστήματος.

Όπως φαίνεται στην εικόνα 3.8, το tender αρχικοποιεί τους κατάλληλους δείκτες αρχείων για τις συσκευές εισόδου/εξόδου. Έπειτα φορτώνει την εικόνα του unikernel

στην μνήμη του, αρχικοποιεί τους seccomp κανόνες για να δημιουργήσει την απομόνωση από το υπόλοιπο σύστημα και τέλος μεταβαίνει στην εκτέλεση του κώδικα της εικονικής μηχανής. Αφού ξεκινήσει η εκτέλεση της μηχανής, ο ρόλος του tender ανάγεται στην διαχείριση των exits της και την επικοινωνία με τις συσκευές και τον πυρήνα.

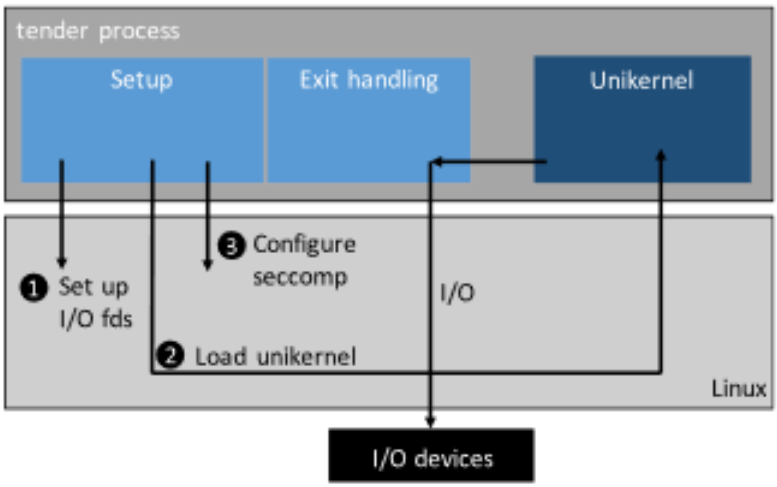

Figure 3.8: Δομή unikernel με solo5 tender

Ένα σημαντικό πλεονέκτημα που προσφέρει αυτή η προσέγγιση είναι ότι εργαλεία που προηγουμένως λείπανε, λόγω της έλλειψης μιας πλήρους πλατφόρμας μέσα στα πλαίσια των unikernels, πλέον μπορούν να χρησιμοποιηθούνε. Ένα σημαντικό παράδειγμα είναι το gdb που χρησιμοποιείται για την αποσφαλμάτωση προγραμμάτων. Επίσης, επιτρέπει την εκτέλεση εικονικών μηχανών εμφωλευμένα σε άλλες εικονικές μηχανές, αφού δεν χρειάζεται η ύπαρξη υποστήριξης εικονοποίησης από το υλικό.

Συνοψίζοντας, το solo5 προσφέρει μινιμαλισμό, έχοντας μικρή βάση κώδικα, μικρούς χρόνους εκκίνησης και αφού έχει σαν βάση το ukvm, οικειοποιείται τον μικρό αριθμό hypercalls, το οποίο του προσφέρει καλή απομόνωση. Σε αυτή συνυπολογίζεται και η έλλειψη δυνατότητας του guest συστήματος να τροποποιήσει τον host, όπως για παράδειγμα ζητώντας περισσότερους πόρους. Το ίδιο ισχύει και από την πλευρά του host, καθώς το solo5 δεν υποστηρίζει τις διακοπές. Τέλος, η υποστήριξη πολλών frameworks και πυρήνων επιτρέπει την εύκολη μεταφορά του.

Κεφάλαιο 4

Σχεδιασμός και Υλοποίηση

Στο κεφάλαιο 3 είδαμε ότι κανένα από τα unikernel frameworks δεν υποστηρίζει την κλήση συστήματος fork τοπικά στην ίδια εικονική μηχανή. Επειδή, όμως, πολλές υπάρχουσες εφαρμογές χρησιμοποιούν αυτή την κλήση συστήματος, η συγκεκριμένη εργασία καταπιάνεται με την υλοποίησή της. Το unikernel framework στο οποίο υλοποιήθηκε η κλήση συστήματος fork είναι το Rumprun όπως εξηγήθηκε στο κεφάλαιο 3.1. Επίσης, ο hypervisor πάνω στον οποίο δοκιμάστηκε η υλοποίηση είναι ο solo5 που περιγράφεται στο κεφάλαιο 3.8.

Η προσπάθεια της συγκεκριμένης διπλωματικής είναι να παραβιάσει τον κανόνα που λέει ότι τα unikernels είναι μονοδιεργασιακά. Δηλαδή, η διεργασία που δημιουργείται, σε αντίθεση με το συνηθισμένο μοντέλο (εικόνα 4.1), δεν είναι ξεχωριστό unikernel, αλλά διεργασία στην ίδια εικονική μηχανή. Αυτό απλοποιεί την επικοινωνία μεταξύ των διεργασιών, αφού με την τοπική δημιουργία διεργασιών, αυτές μπορούν να επικοινωνήσουν μέσω κλασσικών μηχανισμών όπως τα pipes, χωρίς την εκ νεού γραφής τους για το Rumprun.

4.1 Προηγούμενες Προσεγγίσεις

Προηγούμενη προσέγγιση για υποστήριξη της fork κλήσης σε Rumprun unikernels έχει γίνει σε προηγούμενη διπλωματική εργασία [12]. Στην διπλωματική αυτή διατηρείται η μονοδιεργασιακή φύση των unikernels. Δηλαδή, αντί μια διεργασία να δημιουργείται μέσα στο ίδιο unikernel που κάλεσε την fork, δημιουργείται ένα νέο unikernel στο host μηχάνημα που λειτουργεί πλέον ως παιδί.

Σε αυτή την υλοποίηση, η διαδιεργασιακή επικοινωνία γίνεται με την χρήση κοινής μνήμης στον host, είτε μέσω δικτύου. Αυτό σημαίνει ότι κλήσεις συστήματος με σκοπό την διαδιεργασιακή επικοινωνία όπως το pipe υλοποιήθηκαν εκ νέου, ώστε να λειτουργούν με τα semantics που ορίζει το POSIX, ενώ χρησιμοποιούν την κοινή μνήμη ή το δίκτυο για την επικοινωνία τους. Παρότι αυτό λύνει το πρόβλημα της υποστήριξης της κλήσης συστήματος fork, προϋποθέτει την προσομοίωση μιας συσκευής δικτύου και αντίστοιχου οδηγού συσκευής, καθώς και την ύπαρξη δικτύου ή επιπρόσθετου χώρου μνήμης για την επικοινωνία μεταξύ των εικονικών μηχανών.

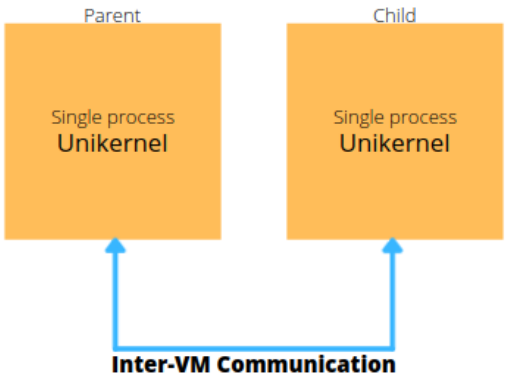

Figure 4.1: Η εικονική μηχανή γονέας καλεί την fork η οποία μέσω του hypervisor και κάποιου hypercall δημιουργεί τον κλώνο της και η επικοινωνία τους γίνεται είτε μέσω κοινής μνήμης είτε μέσω δικτύου με κάποιο πρωτόκολλο όπως TCP/UDP.

Οι προσεγγίσεις της υλοποίησης του pipe στην συγκεκριμένη εργασία για διαδιεργασιακή επικοινωνία είναι τρεις. Αρχικά υλοποιήθηκε η κλήση συστήματος pipe μέσω TCP και UDP sockets. Σε αυτές τις προσεγγίσεις προϋποτίθεται η ύπαρξη συσκευής δικτύου. Η τρίτη υλοποίηση έγινε με χρήση κοινής μνήμης στον host μεταξύ των unikernels. Έτσι δεν είναι απαραίτητη η ύπαρξη συσκευής δικτύου στα unikernels, αλλά η επικοινωνία γίνεται μέσω του ivshmem [13].

Η προσέγγιση για την υλοποίηση της fork, όμως, χρησιμοποιεί μηχανισμούς του hypervisor QEMU/KVM. Μέσω αυτών γίνεται cloning του γονικού μηχανήματος για την δημιουργία της εικονικής μηχανής που παιδιού. Η υλοποίηση αυτής της προσέγγισης δεν δημιουργεί διεργασία μέσα στην ίδια εικονική μηχανή που καλεί την fork. Με αυτή την προσέγγιση καταπιάνεται η παρούσα εργασία. Δηλαδή, η χρήση POSIX νημάτων αφαιρετικά ως διεργασίες για την επίτευξη της μίμησης της κλήσης fork στο περιβάλλον ενός Rumprun unikernel.

4.2 Διαχείριση μνήμης Rumprun

Ο τρόπος που δημιουργεί τα sections[1] της μνήμης το Rumprun είναι ως εξής: text, data, heap από τις χαμηλότερες διευθύνσεις προς τις ψηλότερες. Το Rumprun, όμως, ακολουθεί το flat μοντέλο μνήμης, δηλάδη η μνήμη χωρίζεται μόνο νοηματικά σε κομμάτια και όχι μέσω κάποιας προστασίας μνήμης.

Η μνήμη του text section περιέχει το κώδικα της εικονικής μηχανής και είναι ο χώρος που κινείται ο δείκτης εντολών, ο οποίος βρίσκεται σε χαμηλές διευθύνσεις της μνήμης. Το data section αναφέρεται στον χώρο που αποθηκεύονται οι global μεταβλητές καθώς και οι static μεταβλητές της εικονικής μηχανής, δηλαδή οι μεταβλητές που είναι ενεργές καθόλη την διάρκεια εκτέλεσης της. Από την άλλη, στο heap αποθηκεύονται μεταβλητές που ανατίθονται δυναμικά κατά την εκτέλεση του προγράμματος μέσω κλήσεων σε συναρτήσεις όπως την malloc. Το section που δεν αναφέρεται ρητά παραπάνω είναι το stack, καθώς στο Rumprun το stack κάθε νήματος αρχικοποιείται ως σελίδα σε κάποιο σημείο του heap. Επίσης, στο stack αποθηκεύονται οι automatic μεταβλητές και το call stack. Με τον όρο automatic μεταβλητές αναφερόμαστε στις τοπικές μεταβλητές που ορίζει η εκάστοτε συνάρτηση του προγράμματος και των οποίων το scope είναι εντός του scope της συνάρτησης που ορίστηκαν. Το call stack αναφέρεται στα διαδοχικά πλαίσια από κλήσεις συναρτήσεων που γίνονται κατά την ροή εκτέλεσης του νήματος. Σε αυτό αποθηκεύονται οι διευθύνσεις (pass-by-reference) ή οι τιμές (pass-by-value) των μεταβλητών που δίνονται ως ορίσματα στην εκάστοτε κλήση μιας συνάρτησης. Στο πλαίσιο μιας κλήσης συνάρτησης βρίσκονται επίσης η διεύθυνση μνήμης στην οποία πρέπει να επιστρέψει η ροή εκτέλεσης κατά την έξοδο από αυτή, καθώς και η τιμή επιστροφής - αν υπάρχει - της συνάρτησης.

Για την εκτέλεση προγραμμάτων, το Rumprun χρησιμοποιεί αποκλειστικά νήματα. Τα νήματα που χρησιμοποιεί το Rumprun είναι POSIX νήματα ή pthreads. Τα pthreads είναι νήματα εκτέλεσης μιας διεργασίας που μοιράζονται μεταξύ τους το heap και το data section, ενώ διατηρούν ξεχωριστό stack. Η λογική αυτή συνάδει και με το Rumprun, όπου η heap μνήμη δεν ανατίθεται σε κάποιο νήμα ή διεργασία συγκεκριμένα, αλλά διαχωρίζεται μόνο ως προς το ποιος ανέθεσε την μνήμη, δηλαδή είτε ο πυρήνας είτε τα εκτελέσιμα χρήστη. Επομένως, πέραν του διαχωρισμού μεταξύ του αν ένα μέρος της heap μνήμης το ανέθεσε ο πυρήνας ή κάποιο πρόγραμμα χρήστη, η μνήμη φαίνεται απλώς ως χρησιμοποιούμενη ή όχι. Οπότε, η αδυναμία διαχώρισης μεταξύ της ανάθεσης μνήμης στο heap από διαφορετικές διεργασίες, καθώς και η τοποθέτησή των global αναθέσεων μεταβλητών στο data section, οδήγησε στην μερική αδυναμία υλοποίησης της fork κατά POSIX όπως θα εξηγηθεί αργότερα.

[1] Η χρήση της λέξης section έναντι του segment γίνεται για να διευκρινιστεί η ιδέα ότι σε ένα unikernel όπως το Rumprun, δεν υπάρχει πραγματικός διαχωρισμός μεταξύ της μνήμης, παρά νοηματικά.

Αρχικά, το data section δεν γίνεται να αντιγραφεί, αφού η πρόσβαση σε αυτό δεν μπορεί να γίνει μέσω του Rumprun. Μια προσέγγιση θα ήταν η πρόσβαση του data section μέσω hypercall, αλλά οι ρυθμίσεις του seccomp του solo5 επιτρέπουν παρά ένα μικρό σύνολο κλήσεων συστήματος. Το seccomp είναι μια λειτουργία του Linux kernel και σημαίνει secure computing mode. Με αυτή την λειτουργία επιτρέπει σε μια εφαρμογή να κάνει μια μετάβαση μονής κατεύθυνσης, στην οποία δεν της επιτρέπεται να εκτελέσει κλήσεις συστήματος πέραν ενός μικρού αριθμού. Με τον ίδιο τρόπο το solo5 απομονώνει και την εικονική μηχανή επιτρέποντάς της μόνο 12 hypercalls που αντιστοιχούν σε κλήσεις συστήματος στον host. Αυτό γίνεται για την αποφυγή παραβίασης της απομόνωσης των εικονικών μηχανών[2].

Για να γίνει πιο κατανοητή και η δυσκολία αντιγραφής του heap, πρέπει αρχικά να εξηγηθεί το πως αναθέτει μνήμη στις εφαρμογές το Rumprun. Το Rumprun χωρίζει τις αναθέσεις μνήμης που κάνει σε δύο κατηγορίες, αυτές που είναι μικρότερες ή ίσες των 2MB και αυτές που είναι μεγαλύτερες μέχρι το όριο των 4GB, όπου είναι και το όριο ανάθεσης που υποστηρίζει το Rumprun. Οι αναθέσεις σε αυτές τις δύο κατηγορίες γίνονται σε buckets στην περίπτωση που η ανάθεση είναι μικρότερη ή ίση των 2MB και σε chunks στην περίπτωση όπου η ανάθεση είναι μεγαλύτερη των 2MB.

Αυτά τοποθετούνται σε δύο λίστες - `freebuckets` και `freelist` για buckets και chunks αντίστοιχα - από τις οποίες μπορούν να εξαχθούν κατά την ανάθεση μνήμης. Επανατοποθετούνται κατά την ελευθέρωση τους, ώστε να μπορούν να χρησιμοποιηθούν ξανά. Στο Rumprun σε solo5 με αρχιτεκτονική x86_64, ο αριθμός των buckets είναι 6. Αυτές οι τιμές αντιστοιχούν σε buckets μεγέθους από 32 bytes εώς 2MB, για αριθμούς bucket 0 και 6 αντίστοιχα. Για τιμές μεγαλύτερες του 6, η ανάθεση γίνεται σε chunks με κατάλληλο μέγεθος. Οι σελίδες που αποτελούν την ελάχιστη ανάθεση chunk έχουν μέγεθος 4MB στην συγκεκριμένη αρχιτεκτονική και χρησιμοποιούνται για αναθέσεις άνω των 2MB.

Κάθε κλήση για ανάθεση μνήμης από τον χρήστη, όπως πχ η `malloc`, καταλήγει να καλεί την συνάρτηση του Rumprun `bmk_memalloc`. Κατά την εκτέλεση της, το Rumprun ελέγχει το μέγεθος της ανάθεσης που ζητήθηκε. Στην περίπτωση που ένα bucket επαρκεί, δηλαδή η ανάθεση είναι μεγέθους μικρότερη ή ίση των 2MB, τότε το Rumprun καλεί την `bucketalloc`, η οποία βρίσκει το πρώτο διαθέσιμο bucket κατάλληλου μεγέθους και γυρνάει το δείκτη στο πρόγραμμα που ζήτησε την ανάθεση. Στην περίπτωση που ένα bucket δεν επαρκεί, τότε καλείται η `bmk_pgalloc`, η οποία αναθέτει κατάλληλου μεγέθους chunk. Το Rumprun χρησιμοποεί chunks για την ανάθεση μνήμης σε σελίδες. Το μέγεθος των chunks παίρνει τιμές από μέγεθος σελίδας - 4MB στην συγκεκριμένη αρχιτεκτονική όπως αναφέρθηκε παραπάνω - μέχρι τιμές που ορίζονται από την αρχιτεκτονική της εικονικής μηχανής. Ο τρόπος που συνδέει μεταξύ

[2]https://nabla-containers.github.io/2018/10/29/go/

```
struct memalloc_hdr {
        uint32_t        mh_alignpad;    /* padding for alignment */
        uint16_t        mh_magic;       /* magic number */
        uint8_t         mh_index;       /* bucket # */
        uint8_t         mh_who;         /* who allocated */
};
bmk_ctassert(sizeof(struct memalloc_hdr) == 8);
```

Figure 4.2: Header δείκτη ανατιθέμενης μνήμης

τους τις σελίδες για να δημιουργήσει τα chunks αποτελεί την μέθοδο binary page allocator [14]. Σε αυτή την μέθοδο, η μνήμη χωρίζεται σε όσο το δυνατό μεγαλύτερα κομμάτια. Δηλαδή, η μνήμη αρχικοποιείται χωρισμένη σε όσο το δυνατόν μεγαλύτερα chunks. Όταν ζητείται μια ανάθεση για την οποία δεν υπάρχει κατάλληλη θέση μνήμης, καθώς ένα αρχικό chunk μπορεί να έχει χώρο διευθύνσεων της τάξης των 4GB, ενώ η ανάθεση είναι της τάξης των μερικών kB, τότε ένα μεγάλο chunk σπάει στην μέση. Το πρώτο μέρος χρησιμοποείται για την συνέχεια εύρεσης κατάλληλου μεγέθους chunk για την ανάθεση, ενώ το άλλο είναι ελεύθερο και τοποθετείται στην αντίστοιχη θέση της λίστας ελεύθερων chunks. Αυτό επαναλαμβάνεται μέχρι να βρεθεί κατάλληλο μέγεθος, μέχρι δηλαδή η τάξη μεγέθους της ανάθεσης να συμπίπτει με την τάξη μεγέθους του chunk σε δυνάμεις του 2. Τα ελεύθερα chunks, πλέον, αποτελούν buddy των ανατεθιμένων και κατά την ελευθέρωση των δεύτερων, ελέγχεται άμα τα πρώτα είναι ελεύθερα. Στην περίπτωση που είναι, τότε τα επιμέρους buddy chunks ενώνονται για να σχηματίσουν πάλι το μεγαλύτερο chunk και τοποθετούνται στην αντίστοιχη θέση της λίστας ελεύθερων chunks.

Η δομή για τις πληροφορίες ανάθεσης της μνήμης, η οποία φαίνεται στην εικόνα 4.2, αποθηκεύονται πάνω από τον δείκτη της ανάθεσης και είναι ακριβώς 8 bytes σε μέγεθος. Το `mh_alignpad` είναι μια `uint32_t` μεταβλητή, η οποία κρατάει την απόσταση του δείκτη που δόθηκε κατά την ανάθεση της μνήμης από τα πραγματικά δεδομένα της ανάθεσης μέσα στο χώρο του chunk ή του bucket. Το `mh_index` χρησιμοποιείται για τον υπολογισμό του μεγέθους της ανάθεσης με τις τιμές μικρότερες ή ίσες του 6 να αποτελούν buckets, ενώ τιμές μεγαλύτερες του 6 σε chunks όπως εξηγήθηκε παραπάνω. Το `mh_magic` χρησιμοποιείται για sanity ελέγχους κατά την ανάθεση και ελεύθερωση μνήμης. Τέλος, το `mh_who` κρατάει το ποιος έκανε την ανάθεση με το αναγνωριστικό `BMK_MEMWHO_USER` να αντιστοιχεί σε ανάθεση από εφαρμογή χρήστη, χωρίς κάποιο άλλο τρόπο διάκρισης μεταξύ των εφαρμογών. Από αυτό μπορούμε να καταλάβουμε ότι η μνήμη δεν έχει κάποιο τρόπο να ξεχωρίσει το ποια εφαρμογή και κατ'επέκταση διεργασία έκανε την ανάθεση.

4.3 Σχεδιασμός

Σε αυτή την ενότητα αρχικά θα αναφερθούμε στις δομές του Rumprun που μας ενδιαφέρουν για την υλοποίηση της fork. Έπειτα θα εξηγηθεί πως αυτές αρχικοποιούνται στην διεργασία παιδί και αντιγράφονται σε αυτό από τον γονέα, καθώς και η ροή εκτέλεσης των δύο διεργασιών.

4.3.1 Light-weight Processes

Σε ένα συμβατικό μοντέλο λειτουργικό συστήματος μια διεργασία είναι μια δομή που περιέχει τους διαθέσιμους πόρους ενός προγράμματος, ενώ ένα light-weight process (LWP) είναι το πλαίσιο εκτέλεσής του. Κάθε LWP είναι συσχετισμένο με ακριβώς μια διεργασία, ενώ μια διεργασία μπορεί να συσχετίζεται με ένα ή περισσότερα LWP. Κατά την εκτέλεση ενός προγράμματος το τρέχων LWP υποδηλώνει την διεργασία η οποία τρέχει καθώς και προσδιορίζει τους πόρους της, όπως τον πίνακα δεικτών σε αρχεία που είναι διαθέσιμα την δεδομένη στιγμή.

Συγκεκριμένα στους rump kernels, όταν ένα pthread νήμα του Rumprun θέλει να χρησιμοποιήσει λειτουργίες του rump kernel χρησιμοποιεί το συσχετισμένο σε αυτό LWP. Στην περίπτωση που δεν έχει συσχετιστεί ρητά κάποιο LWP με το pthread νήμα, τότε χρησιμοποιείται ανταυτού το έμμεσο LWP που υπάρχει για την διεργασία με αναγνωριστικό 1 ονόματι `lwp0`. Αυτό σημαίνει ότι κάθε φορά που ένα pthread του Rumprun χωρίς ορισμένο LWP εισερχεται στον rump kernel, τότε το `lwp0` χρησιμοποιείται για να αρχικοποιηθεί ένα προσωρινό LWP που θα λειτουργήσει ως πλαίσιο νήματος κατά την ροή εκτέλεσης στο rump kernel. Αυτό διαγράφεται κατά την έξοδο της ροής εκτέλεσης από το rump kernel.

4.3.2 Bare metal kernel threads

Τα bare metal kernel threads (bmk threads) αποτελούν τα νήματα που τρέχουν στον εικονικό επεξεργαστή ενός Rumprun unikernel. Στην εικόνα 4.3 φαίνεται η δομή την οποία κρατάει το Rumprun και η οποία χρησιμοποιείται για την αποθήκευση της κατάστασης ενός νήματος. Αυτή η δομή διατηρεί τα χαρακτηριστικά του νήματος για την σωστή διατήρηση της κατάστασής του κατά την διακοπή και επαναφορά της εκτέλεσής του. Παρακάτω θα εξηγηθούν τα χαρακτηριστικά της δομής `bmk_thread` που αφορούν την παρούσα εργασία. Για να γίνει ξεκάθαρο, είναι σημαντικό να αναφερθεί ότι το solo5 δεν χρησιμοποιεί thread local storage (TLS), επομένως οι μεταβλητές `bt_lwp` και `bt_lwp_me` δεν αναφέρονται περαιτέρω.

- `void *bt_stackbase`: Ο δείκτης `bt_stackbase` δείχνει στην αρχή του stack

```
struct bmk_thread {
        char bt_name[NAME_MAXLEN];

        bmk_time_t bt_wakeup_time;

        int bt_flags;
        int bt_errno;

        void *bt_stackbase;

        void *bt_cookie;

#ifndef RR_USE_TLS
        /* for rumpuser_synch.c: __thread lwp */
        struct lwp *bt_lwp;

        /* for _lwp: __thread me */
        struct rumprun_lwp *bt_lwp_me;
#endif

        /* MD thread control block */
        struct bmk_tcb bt_tcb;

        TAILQ_ENTRY(bmk_thread) bt_schedq;
        TAILQ_ENTRY(bmk_thread) bt_threadq;
};
#ifdef RR_USE_TLS
__thread struct bmk_thread *bmk_current;
#else
struct bmk_thread *bmk_current;
```

Figure 4.3: Δομή BMK thread

του νήματος, με το stack να μεγαλώνει προς τις μεγαλύτερες διευθύνσεις. Αυτή η μεταβλητή ορίζει το stack του νήματος εκτέλεσης και χρησιμοποιείται για automatic μεταβλητές και το call stack όπως αναφέρθηκε προηγουμένως.

- `void *bt_cookie`: Ο δείκτης `bt_cookie` δείχνει στην δομή `struct rumprun_lwp`, το οποίο είναι με την σειρά του μια δομή η οποία κρατάει το Rumprun σαν wrapper για τα LWP νήματα που χρησιμοποιεί κατά την είσοδο της ροής εκτέλεσης στον rump kernel. Στο `bmk_thread` ο δείκτης στο `struct rumprun_lwp` χρησιμοποιείται για να κρατιέται το αναγνωριστικό του επεξεργαστή στον οποίο θα τρέξει το νήμα, καθώς και αν η εκτέλεσή του νήματος έχει σταματήσει. Αυτές οι πληροφορίες περιέχονται στο πεδίο `struct lwpctl` του `rumprun_lwp`, η οποία περιέχει πληροφορίες σχετικά με την εκτέλεση του LWP.

- `struct bmk_tcb bt_tcb`: Η δομή `bmk_tcb` είναι μια δομή που χρησιμοποιείται κατά το ξετύλιγμα του νήματος στον επεξεργαστή ως σημείο εκκίνησης της εκτέλεσης. Σε αυτό υπάρχουν ο δείκτης εντολής του νήματος, δηλαδή το σημείο στην ροή εκτέλεσης από το οποίο πρέπει να συνεχίσει το νήμα. Επίσης, κρατάει την κορυφή της στοίβας από την οποία θα ξεκινήσει κατά την επανεκκίνησή του να ξεδιπλώνει το call stack. Τέλος, κρατάει και το TLS χώρο ενός νήματος. Αφού, όμως, το solo5 δεν χρησιμοποιεί TLS, αυτή η περιοχή δεν χρησιμοποιείται παρά μόνο για την αποθήκευση της διεύθυνσης μνήμης του ίδιου του `bmk_thread`.

4.3.3 Rumprunners

Κατά την εκκίνηση της εικονικής μηχανής το solo5 δημιουργεί ένα `bmk_thread` που εκτελεί την συνάρτηση `bmk_mainbouncer`. Αυτή η συνάρτηση λειτουργεί ως το σημείο εισόδου στον κώδικα του unikernel και το νήμα που την εκτελεί λειτουργεί νοηματικά όπως το init νήμα στα Linux. Αυτό, λοιπόν, τρέχει την συνάρτηση `rumprun_boot` που αρχικοποιεί το περιβάλλον του Rumprun. Έπειτα, κάνει κλήσεις στην συνάρτηση `rumprun` με όρισμα τα εκτελέσιμα χρήστη, για να δημιουργήσει τις δομές που χρησιμοποιεί το Rumprun για να μιμηθεί τις διεργασίες που θα υπήρχαν σε έναν συμβατικό πυρήνα.

Αυτή η δομή φαίνεται στην εικόνα 4.4. Στην δομή κρατούνται στο `rr_mainfun` η συνάρτηση εισόδου του εκτελέσιμου, δηλαδή η main συνάρτηση μιας εφαρμογής. Επίσης, κρατούνται τα `rr_argc`, `rr_argv` τα οποία αντιστοιχούν στον αριθμό των ορισμάτων που δόθηκαν στο εκτελέσιμο και των τιμών που έχουν. Σε αυτή την δομή κρατιέται και το `rr_mainthread` που είναι το αναγνωριστικό του pthread το οποίο θα λειτουργήσει σαν κύριο νήμα κατά την εκτέλεση του προγράμματος `rr_mainfun`.

```
struct rumprunner {
        int (*rr_mainfun)(int, char *[]);
        int rr_argc;
        char **rr_argv;

        pthread_t rr_mainthread;
        struct lwp *rr_lwp;

        int rr_flags;

        LIST_ENTRY(rumprunner) rr_entries;
};
static LIST_HEAD(,rumprunner) rumprunners = LIST_HEAD_INITIALIZER(&rumprunners);
```

Figure 4.4: Δομή rumprunner

Επίσης, υπάρχει το `rr_lwp` που είναι το LWP που αντιστοιχεί στον `rumprunner`. Τέλος, τα `rr_flags` αποτελούν αναγνωριστικά που χρησιμοποιούνται για την ενημέρωση του init νήματος ότι ένας rumprunner έχει ολοκληρώσει την εκτέλεσή του. Την ολοκλήρωση τους την ελέγχει μέσω της `rumprun_get_finished`. Κάθε φορά που έρχεται η σειρά εκτέλεσής του, τρέχει την συνάρτηση επαναλαμβανόμενα μέχρι να βρεθεί κάποιο νήμα με την μεταβλητή rr_flags τεθειμένη σε `RUMPRUNNER_DONE`. Μόλις την βρει τρέχει τις συναρτήσεις καθαρισμού του pthread. Τέλος, οι `rumprunners` τοποθετούνται στην ομότιτλη λίστα κατά την δημιουργία τους και αφαιρούνται από αυτή κατά την εκτέλεση των ρουτινών καθαρισμού του init νήματος. Αφού όλα τα προγράμματα ολοκληρώσουν την εκτέλεση τους, το init νήμα εκτελεί την διαδικασία απενεργοποίησης του Rumprun μέσω της συνάρτησης `rumprun_reboot`.

4.3.4 Προβλήματα κατά τον σχεδιασμό

Η υλοποίηση του fork σε ένα περιβάλλον με μοναδικό χώρο διευθύνσεων και έλλειψη εικονικής μνήμης - όπως του Rumprun και άλλων unikernels - δεν είναι χωρίς δυσκολίες. Αρχικά, η έλλειψη εικονικής μνήμης σημαίνει ότι η προσέγγιση της fork όπως την εκτελούν τα Linux δεν λειτουργεί. Στην περίπτωση των Linux, για να κάνει μια διεργασία fork, αντιγράφεται εξολοκλήρου η διεργασία γονέας και τοποθετείται σε ένα νέο περιβάλλον με νέο χώρο διευθύνσεων. Αυτό δεν μπορεί να λειτουργήσει σε ένα unikernel περιβάλλον, αφού υπάρχει μόνο ένας χώρος διευθύνσεων για όλη την μηχανή και το περιβάλλον της και δεν υπάρχει υποστήριξη για δημιουργία καινούριου. Αυτό για να γίνει θα προϋπόθετε την υποστήριξη εικονικής μνήμης, κάτι που δεν έχει νόημα στο μονοδιεργασιακό περιβάλλον του Rumprun, αφού η πολυπλοκότητα υλοποίσης της εικονικής μνήμης, καθώς και η επιβάρυνση στον χρόνο εκτέλεσης του ίδιου του unikernel δεν βρέθηκε σκόπιμη [4].

Εφόσον η προσέγγιση των Linux δεν λειτουργεί στο Rumprun, για να αντιγραφεί

κατάλληλα μια διεργασία πρέπει να αντιγραφούν το stack, το heap και το data section του γονέα, καθώς και να διατηρηθούν ανέγγιχτες οι δομές οι οποίες χρησιμοποιούνται για την διαχώριση νοηματικά γονέα και παιδιού. Δομές όπως είναι το pthread που λειτουργεί ως κύριο νήμα της εφαρμογής και το `bmk_thread` που είναι το νήμα εκτέλεσης του επεξεργαστή, πρέπει να τεθούν σωστά στο παιδί, χωρίς όμως να τα μοιράζεται με τον γονέα του. Το ίδιο συμβαίνει και στα δεδομένα του προγράμματος εκτέλεσης που ανατίθενται δυναμικά κατά την εκτέλεση της εφαρμογής.

Δεδομένου της έλλειψης πολλαπλών χώρων διευθύνσεων από τα Rumprun unikernels, η υλοποίηση της fork σε αυτή την εργασία διαφέρει από το πρότυπο POSIX στο εξής. Όταν μια διεργασία παιδί δημιουργηθεί οι μεταβλητές που έχουν global scope και γενικά δεν βρίσκονται στο stack μοιράζονται με τον γονέα. Αυτές συμπεριλαμβάνουν τις static μεταβλητές του data section. Αυτό οφείλεται στην αδυναμία αναφοράς στις διευθύνσεις των μεταβλητών που δεν βρίσκονται στο stack κατά την εκτέλεση ενός προγράμματος. Δηλαδή, κατά το fork αντιγράφεται η ροή εκτέλεσης του γονέα, καθώς και αντιγράφεται το stack του γονέα όπως θα περιγραφεί στην επόμενη ενότητα, αλλά οι μεταβλητές που ανατίθονται στο heap και στο data section, δεν γίνεται να αντιγραφούν χωρίς να γίνεται αντιγραφή όλου του χώρου διευθύνσεων. Επίσης, ο hypervisor που χρησιμοποιήθηκε, δηλαδή ο solo5, δεν επιτρέπει την αλλαγή των πόρων που χρησιμοποιεί μια εικονική μηχανή κατά την εκτέλεσή της, όπως να αυξηθεί η μνήμη που χρησιμοποιεί.

4.4 Υλοποίηση

Μέχρις στιγμής η fork στο Rumprun δεν έχει υλοποιηθεί. Για τις απαιτήσεις του συστήματος έχει δωθεί ένα weak alias στην κλήση, η οποία γυρνάει ένα μήνυμα στον χρήστη ότι δεν έχει γίνει υλοποίησή της.

Για τις κλήσεις συστήματος, το Rumprun χρησιμοποιεί strong και weak aliases. Τα strong aliases είναι για κλήσεις εσωτερικά του συστήματος και δεν μπορούν να γίνουν override, ενώ τα weak aliases είναι κλήσεις εκτεθιμένες στον χρήστη και μπορούν να γίνουν override με την απλή υλοποίηση της κλήσης που αντιστοιχούν. Έτσι έγινε override η user exposed κλήση fork στο αρχείο lib/librumprun_base/fork.c. Το οποίο προστέθηκε και στο Makefile του αντίστοιχου υποφακέλου για την εντόπιση του κατά το build.

Για την προσομοίωση της fork, λοιπόν, δημιουργείται ένας `rumprunner`, ο οποίος θα ξεκινάει την εκτέλεση του από το σημείο που έγινε η κλήση της fork από τον γονέα. Ο `rumprunner` δημιουργείται, ώστε το Rumprun να αναγνωρίσει το παιδί ως ένα εκτελέσιμο νήμα χρήστη. Έτσι, αφού το νήμα εκτελέσει την έξοδο του, το init νήμα κάνει `pthread_join` με το τερματισμένο πλέον νήμα του παιδιού και το αφαιρέσει από

την λίστα των `rumprunners`, όπως δηλαδή θα εκτελούσε για οποιοδήποτε `rumprunner` είχε φτιάξει κατά την εκκίνηση του unikernel. Η συνάρτηση `rumprun` είναι, επίσης, αυτή η οποία αναλαμβάνει και την δημιουργία του LWP του παιδιού και την αντιγραφή πόρων που σχετίζονται με αυτό από τον γονέα, όπως είναι ο πίνακας δεικτών σε αρχεία του γονέα. Έτσι διατηρείται η ομοιομορφία του περιβάλλοντος, καθώς μετά την δημιουργία του `rumprunner` παιδιού και την τοποθέτησή του στην λίστα των `rumprunners`, δεν χρειάζεται καμία αλλαγή στην λογική διαχείρισης των `rumprunners` του Rumprun για να εκτελεστεί κανονικά ο καινούριος `rumprunner`.

Η αντιγραφή της ροής εκτέλεσης γίνεται με δύο τρόπους. Από την μία αντιγράφεται το `bt_tcb` του `rumprunner` του γονέα, το οποίο ορίζει και το σημείο εκκίνησης για τον επεξεργαστή όταν το νήμα επαναδρομολογηθεί σε αυτόν, συγκεκριμένα μέσω του `bt_tcb.btcb_ip` πεδίου του. Αντιγράφεται επίσης το chunk του stack, δηλαδή ο χώρος που ορίζεται από το `bt_stackbase` εώς και την διεύθυνση που ορίζει το `bt_tcb.btcb_sp`, το οποίο ορίζει την κορυφή της στοίβας και από το οποίο ξεκινάει και το call stack του νήματος. Πέρα από το call stack, κατά την αντιγραφή του stack του νήματος του γονέα, δημιουργούνται στο stack του παιδιού οι μεταβλητές που έχουν οριστεί στο stack του γονέα και αρχικοποιούνται στις τιμές που έχει την δεδομένη στιγμή ο γονέας.

4.4.1 Εκτέλεση στο πλαίσιο γονέα

Κατά την κλήση της, η fork κάνει τα εξής:

1. αρχικοποιεί δύο μηχανισμούς, ένα pthread mutex και ένα pthread condition
2. καλεί την συνάρτηση `rumprun`, η οποία δημιουργεί έναν `rumprunner` που θα λειτουργήσει ως διεργασία παιδί,
3. περιμένει την αρχικοποίηση του νήματος παιδιού
4. και, τέλος, αφού το παιδί αρχικοποιηθεί γυρνάει στο σημείο που κλήθηκε η fork από το πρόγραμμα χρήστη με κατάλληλη τιμή επιστροφής.

Μόλις η ροή εκτέλεσης του γονέα μεταβεί στην κλήση της fork δύο μηχανισμοί των POSIX threads αρχικοποιούνται για τον συγχρονισμό της δημιουργίας του παιδιού και της αντιγραφής του γονέα σε αυτό. Αυτοί οι δύο μηχανισμοί είναι ένα mutex και ένα condition variable όπως ορίζονται από τα pthreads. Αφού αρχικοποιηθούν, ο γονέας κλειδώνει το mutex, δημιουργεί το παιδί μέσω της κλήσης στην συνάρτηση `rumprun`, και περιμένει το condition variable - αφού ελευθερώσει το mutex - που θα του ανακοινωθεί από το παιδί όταν η αρχικοποίησή του ολοκληρωθεί.

Για την δημιουργία του παιδιού χρησιμοποιήθηκε η συνάρτηση `rumprun` στο lib/librumprun_base/rumprun.c, η οποία καλείται για την δημιουργία του κύριου νήματος που τρέχει ένα πρόγραμμα. Η συνάρτηση `rumprun` δημιουργεί, επίσης, και το πλαίσιο που θα εκτελεστεί το `rumprunner` μέσω της `rump_pub_lwproc_rfork`, η οποία είναι υπεύθυνη για την δημιουργία ενός νέου LWP. Αυτή η συνάρτηση είναι μια user exposed κλήση του rump kernel, η οποία δημιουργεί μια διεργασία στον rump kernel με ένα LWP με τα χαρακτηριστικά του LWP που είναι ενεργό κατά την εκτέλεσή της. Η `rump_pub_lwproc_rfork` παίρνει ως όρισμα μια σημαία η οποία παίρνει τιμές `RUMP_RFFDG` και `RUMP_RFCFDG`, με την πρώτη να σημαίνει ότι η διεργασία/LWP που θα δημιουργηθεί μοιράζεται τους δείκτες αρχείων με τον γονέα, η οποία είναι και η τιμή που χρησιμοποιεί η συνάρτηση `rumprun`.

Καθώς ο δρομολογητής του Rumprun είναι συνεργατικός, όταν ο γονέας μπει σε αναμονή για την αρχικοποίηση του παιδιού, παραχωρεί την θέση του στην εκτέλεση των υπόλοιπων νημάτων, με ένα από αυτά να είναι του παιδιού.

Η τιμή επιστροφής της fork για τον γονέα ενημερώνεται, αφού δημιουργηθεί το παιδί και κοινοποιηθεί το αναγνωριστικό διεργασίας του στον γονέα. Η τιμή αυτή είναι και η τιμή επιστροφής του γονέα κατά την έξοδο του από την κλήση της fork.

4.4.2 Εκτέλεση στο πλαίσιο παιδιού

Η συνάρτηση `rumprun` που καλεί ο γονέας, παίρνει ως όρισμα, ανάμεσα σε άλλα, την συνάρτηση από την οποία θα ξεκινήσει η εκτέλεση του `rumprunner` του παιδιού. Η συνάρτηση που δίνεται ως όρισμα κατά την κλήση της fork είναι η `init_forkthread`, η οποία έχει ως σκοπό την σωστή αρχικοποίηση του παιδιού, ώστε στην επόμενη επαναδρομολόγηση του να συνεχίσει την ροή από το σημείο που την άφησε ο γονέας. Η συνάρτηση `init_forkthread` που τρέχει το παιδί εκτελεί τα εξής:

1. βρίσκει το αναγνωριστικό της διεργασίας του,
2. δημιουργεί ένα νέο νήμα με εργασία την αντιγραφή του γονέα,
3. και παραχωρεί την εκτέλεση του στο νήμα εργάτη που έφτιαξε.

Αρχικά, η διεργασία καλεί την `rump_sys_getpid` για να μάθει το αναγνωριστικό διεργασίας της. Αυτό αποθηκεύεται σε μια global μεταβλητή και θα χρησιμοποιηθεί από τον γονέα για την επιστροφή της σωστής τιμής επιστροφής στο πρόγραμμα που κάλεσε την fork. Έπειτα δημιουργεί ένα pthread νήμα εργάτη το οποίο έχει ως εργασία την αντιγραφή του γονέα, όπως θα εξηγηθεί παρακάτω. Αφού το δημιουργήσει, το παιδί παραχωρεί την εκτέλεσή του, η οποία κατά την επαναδρομολόγηση του νήματος παιδιού, θα ξεκινήσει πλέον από το ίδιο σημείο που σταμάτησε την εκτέλεση του ο

γονέας, αφού το νήμα εργάτης που δημιούργησε θα έχει αντιγράψει την ροή εκτέλεσης του γονέα στο παιδί. Η ροή εκτέλεσης βρίσκεται στην στοίβα του γονέα, η οποία και αντιγράφεται κατά την εκτέλεση του νήματος εργάτη του παιδιού.

Για την σωστή αντιγραφή του νήματος του γονέα, πρέπει να αντιγραφεί το `bmk_thread` του Rumprun. Για να γίνει αυτό, το νήμα εργάτης του παιδιού όταν εκτελεστεί κάνει τα εξής:

1. κλειδώνει το mutex που μοιράζεται το παιδί με τον γονέα,
2. καλεί την συνάρτηση `mk_sched_cpy` για την αντιγραφή του `bmk_thread` του γονέα,
3. κάνει broadcast το condition variable που μοιράζεται με τον γονέα,
4. και ελευθερώνει το mutex ολοκληρώνοντας την εκτέλεσή του.

Όπως προαναφέρθηκε, για την εξασφάλιση του συγχρονισμού, έγινε η χρήση mutex και condition variables από την βιβλιοθήκη των pthreads. Επίσης, απαραίτητη ήταν η υλοποίηση της `bmk_sched_cpy` για την αντιγραφή του `bmk_thread` του γονέα, η οποία βρίσκεται στο αρχείο lib/libbmk_core/sched.c, το οποίο περιέχει διάφορες συναρτήσεις σχετικές με την δομή του `bmk_thread`.

Η συνάρτηση `bmk_sched_cpy` παίρνει δύο `bmk_thread` ως ορίσματα και αντιγράφει το πρώτο στο δεύτερο, επιστρέφοντας το δεύτερο κατά την επιστροφή της. Στην συγκεκριμένη περίπτωση αντιγράφει το νήμα του γονέα στο νήμα του παιδιού. Κατά την εκτέλεσή της αντιγράφεται ο δείκτης εντολών που έχει το `bmk_thread` του γονέα στο παιδί, ώστε να έχει την σωστή συνάρτηση εισόδου κατά την επαναδρομολόγησή του. Ο δείκτης εντολών ενός νήματος βρίσκεται στην τιμή `btcb_ip` του `bt_tcb` πεδίου του. Έπειτα καλεί την `stackcpy`, η οποία κάνει την αντιγραφή των stacks μεταξύ των δύο νημάτων.

Η συνάρτηση `stackcpy` κάνει τα εξής:

1. ευθυγραμμίζει το μέγεθος των stacks γονέα και παιδιού, ώστε να έχουν το ίδιο μέγεθος,
2. αντιγράφει τις θέσεις μνήμης που δείχνει ο δείκτης του πάτου του stack και έπειτα, μέχρι το τέλος της σελίδας του,
3. και τέλος, αντιγράφει το `bmk_thread` του παιδιού ξανά στην κορυφή της στοίβας του.

Πρέπει, επίσης, να διευκρινιστεί ότι υπάρχουν τρία είδη δεδομένων που μας ενδιαφέρουν στην συγκεκριμένη υλοποίηση. Αυτές είναι:

- μεταβλητές που βρίσκονται στο heap σε κουβάδες και σελίδες,
- μεταβλητές που αποθηκεύονται στο stack όπως είναι οι τοπικές μεταβλητές,
- μεταβλητές που αποθηκεύονται στο data section,
- και τέλος, τα δεδομένα που μένουν αμετάβλητα μεταξύ της εκτέλεσης των δύο διεργασιών όπως διευθύνσεις μνήμης συναρτήσεων του text section.

Όπως έχει ήδη αναφερθεί, η αντιγραφή των heap και data sections προκάλεσε δυσκολία, καθώς δεν υπάρχει αναφορά στο stack ενός νήματος για κάθε μεταβλητή που αποθηκεύεται σε αυτά τα sections. Αυτές είναι μεταβλητές που ορίζονται κατά την μεταγλώττιση του προγράμματος και οι διευθύνσεις μνήμης τους γράφονται είτε απευθείας στον κώδικα, είτε στο data section της εικονικής μηχανής που είναι απροσπέλαστο μέσα από το Rumprun λόγω των σχεδιαστικών επιλογών του solo5. Και για τα δύο sections, δηλαδή, δεν υπάρχει συστηματικός τρόπος για την αντιγραφή τους.

Από την άλλη, μεταβλητές που υπάρχει αναφορά τους στο stack ενός νήματος όπως τοπικές μεταβλητές που μπορεί να βρίσκονται είτε σε chunk είτε σε bucket, μπορούν να αναγνωριστούν μέσω της συνάρτησης `addr_is_managed`. Η συνάρτηση `addr_is_managed` παίρνει σαν όρισμα μια διεύθυνση μνήμης και γυρνάει σαν αποτέλεσμα μια boolean μεταβλητή που δηλώνει αν η διεύθυνση ανήκει στο heap section και αν έχει ανατεθεί. Όταν μια τέτοια μεταβλητή βρεθεί κατά την αντιγραφή του stack του γονέα, τότε αντιγράφεται ολόκληρο το αντίστοιχο bucket ή chunk στο οποίο περιέχεται σε μια νέα θέση στην μνήμη και τοποθετείται στο stack του παιδιού.

Τέλος, τα δεδομένα που δεν εμπίμπτουν στις managed addresses αποτελούνται από κομμάτια του text section και του data section. Όσα εμπίπτουν στο text section είναι διευθύνσεις συναρτήσεων και αντιγράφονται απευθείας στο stack του παιδιου, ώστε να αρχικοποιηθεί σωστά το call stack του. Τα κομμάτια του data section που υπάρχουν στο stack του γονέα, αντιγράφονται επίσης απευθείας ως διευθύνσεις στο stack του παιδιού, χάριν καλύτερης διαχείρισης. Αυτό, δηλαδή, σημαίνει ότι ακόμη και οι μεταβλητές του data section που βρίσκονται στο stack, δεν αντιγράφονται, αλλά τις μοιράζεται ο γονέας με το παιδί.

Αφού τρέξουν, λοιπόν, το νήμα του εργάτη και αρχικοποιηθεί το παιδί, το σημείο από το οποίο ξεκινάει την εκτέλεσή του το παιδί είναι το σημείο που σταμάτησε την εκτέλεσή του ο γονέας, δηλαδή στην κλήση της `pthread_cond_wait`. Επειδή το νήμα εργάτη του παιδιού όμως έχει κάνει κάνει κλήση στην `pthread_cond_broadcast` και τα δύο νήματα συνεχίζουν την εκτέλεσή τους κανονικά, χωρίς άλλες αναμονές. Σε αυτό το σημείο τοποθετείται ένας έλεγχος, ο οποίος ελέγχει αν το `bmk_thread` που τρέχει ανήκει στον γονέα ή στο παιδί και γυρνάει τις κατάλληλες τιμές κατά την έξοδό τους από την fork.

4.5 Υλοποίηση wait

Η wait ορίζεται ως η συνάρτηση που καλείται από τον γονέα και επιστρέφει αφού κάποιο από τα παιδιά του αλλάξει κατάσταση. Οι αλλαγές κατάστασεις ορίζονται ως εξής, το παιδί είτε τερματίστηκε, είτε ένα σήμα το σταμάτησε είτε συνέχισε την εκτέλεσή του εξαιτίας κάποιου σήματος. Στην περίπτωση όμως του Rumprun, τα σήματα δεν υποστηρίζονται. Αυτό σημαίνει ότι μόνο όταν ένα παιδί τερματίσει θα επιστρέφει η εκτέλεση στον γονέα. Για να επιτευχθεί, λοιπόν, η λειτουργία της wait αρκεί ο επανειλλημένος έλεγχος της κατάστασης του `rumprunner` του παιδιού από τον γονέα μέσω της σημαίας της δομής του `rumprunner`, η οποία όταν το νήμα ολοκληρώσει την εκτέλεσή του γίνεται `RUMPRUNNER_DONE`. Αν αυτό ακόμη είναι ενεργό, τότε ο γονέας παραχωρεί πάλι την σειρά του, ώστε να τρέξει το παιδί ξανά. Όταν πλέον το παιδί τερματίσει, επιστρέφεται στον γονέα το αναγωνριστικό της διεργασίας του παιδιού.

Το override της wait, όπως και της fork γίνεται με την απλή υλοποίησή της, καθώς της έχει δωθεί και αυτής ένα weak alias. Η υλοποίησή της ορίζεται στο ίδιο αρχείο με την fork.

4.6 Αξιολόγηση

Για την χρονική αξιολόγηση της fork συγκρίθηκαν οι χρόνοι εκτέλεσης του αποσπάσματος που φάινεται στο 4.1 τρέχοντάς το απευθείας πάνω σε ένα συμβατικό λειτουργικό σύστημα ενάντια στους χρόνους εκτέλεσης μέσα σε ένα unikernel στο ίδιο σύστημα. Το λειτουργικό σύστημα που χρησιμοποιήθηκε για τον host είναι το Ubuntu 18.04 LTS με πυρήνα Linux 5.4.0-72-generic #80∼18.04.1-Ubuntu SMP, ενώ το unikernel έτρεξε στο ίδιο host λειτουργικό σύστημα πάνω σε hypervisor solo5 και guest λειτουργικό σύστημα το Rumprun πάνω σε NetBSD rump kernel [15].

Η σύγκριση των δύο γίνεται καθώς όπως φαίνεται ο χρόνος εκτέλεσης της fork είναι ίδιας τάξης μεγέθους, καθώς και επειδή το solo5 τρέχει τις εικονικές μηχανές ως απλές διεργασίες στον φιλοξενητή μειώνοντας κατά πολύ τα traps που εκτελούνται κατά την εκτέλεση της μηχανής, άρα και τον χρόνο εκτέλεσής της. Άρα κρίθηκε και πιο λογική η σύγκριση αυτών των δύο περιβάλλοντων. Στην περίπτωση της εκτέλεσης απευθείας στο λειτουργικό σύστημα του host, ο μέσος χρόνος εκτέλεσης σε 10 εκτελέσεις διαρκούσε 0.000118 δευτερόλεπτα. Ενώ στην περίπτωση του unikernel, ο μέσος χρόνος εκτέλεσης σε 10 εκτελέσεις διαρκούσε 0.000882 δευτερόλεπτα.

Παρότι ο χρόνος εκτέλεσης της fork φαίνεται να είναι αρκετά ικανοποιητικός, η συγκεκριμένη υλοποίηση της fork, δηλαδή με την χρήση νημάτων μασκαρεμένα ως διεργασίες δεν καλύπτει όλες τις χρήσεις της fork. Συγκεκριμένα, όπως έχει αναφερθεί, η συγκεκριμένη υλοποίηση της fork δεν καλύπτει το πρότυπο POSIX. Αυτό σημαίνει

ότι εφαρμογές που θέλουν να κάνουν χρήση μιας τέτοιας υλοποίησης θα πρέπει να τροποποιηθούν κατάλληλα. Παρότι αυτό περιορίζει τις χρήσεις της συγκεκριμένης υλοποίησης, διεργασίες που δημιουργούνται με σκοπό την εκτέλεση της κλήσης exec μπορούν να καλυφθούν. Αυτό σημαίνει, όμως, ότι πρέπει πρώτα να γίνει υλοποίηση της `exec` που την στιγμή που γράφηκε αυτή η διπλωματική, δεν υποστηρίζεται από το Rumprun.

Τέλος, πρέπει να αναφερθεί ότι η συγκεκριμένη υλοποίηση της fork καλύπτει εφαρμογές που χρησιμοποιούν διαδιεργασιακούς μηχανισμούς όπως το pipe χωρίς αλλαγές στην λογική της χρήσης τους όπως φαίνεται και στο απόσπασμα 4.2. Αυτό οφείλεται στην διαχείριση των δεικτών σε αρχεία από τα LWP και τον rump kernel, ο οποίος κατά την fork αντιγράφει τον πίνακα δεικτών σε αρχεία του γονέα και το τοποθετεί στο παιδί.

Παρότι το πρόγραμμα του αποσπάσματος 4.2 είναι αρκετά απλό, λειτουργεί ικανοποιητικά ως ένα proof of concept της υλοποίησης, καθώς καλεί διαδοχικές κλήσεις στο read και write των δεικτών του pipe και τυπώνει αυτά που προβλέπονται.

```c
#include <unistd.h>
#include <stdio.h>

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

int main() {
  struct timeval t1, t2;

  gettimeofday(&t1, 0);
  pid_t pid = fork();
  gettimeofday(&t2, 0);

  fprintf(stdout, "fork took: %lfs\n",
          (double)((t2.tv_sec - t1.tv_sec) * USEC +
          t2.tv_usec - t1.tv_usec) / USEC);

  if (pid < 0) {
    perror("fork");
    return -1;
  } else if (pid > 0) {
    // This is the parent "process"
    fprintf(stdout, "parent\n");
  } else {
    // This is the child "process"
    fprintf(stdout, "child\n");
  }

  return 0;
}
```

Απόσπασμα 4.1: Fork Test Programme

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

void server(int fd[2])
{
  int n;
  char buf[30], buf1[30];

  close(fd[1]);
  printf("wait to read...\n");
  n = read(fd[0], buf, 30);
  if (n < 0)
    perror("read");

  printf("parent: read %d bytes\n", n);
  printf("parent: Got message: %s\n", buf);

  n = read(fd[0], buf, 30);
  if(n < 0)
    perror("read");
  printf("parent: read %d bytes\n", n);
  printf("parent: Got message: %s\n", buf);
  printf("wait toread...\n");
  n = read(fd[0], buf1, 30);
  if (n < 0)
    perror("read");
  printf("parent: read %d bytes\n", n);
  printf("parent: Got message: %s\n", buf1);
  close(fd[0]);
  return;
}

void client(int fd[2])
{
  int n;
  char *buf = "Hello from another unikernel!\n";

  close(fd[0]);
  n = write(fd[1], buf, 30);
  if(n < 0)
    perror("write");

```

```
46    printf("child: wrote %d bytes\n", n);
47    fflush(stdout);
48    buf[0] = 'S';
49    n = write(fd[1], buf, 30);
50    if (n < 0)
51      perror("write");
52
53    printf("child: wrote %d bytes\n", n);
54    buf[0] = 'Q';
55    n = write(fd[1], buf, 20);
56    if(n < 0)
57      perror("write");
58    printf("child: wrote %d bytes\n", n);
59    close(fd[1]);
60    return;
61  }
62
63  int main()
64  {
65    int fd[2], n;
66
67    if(pipe(fd) < 0)
68      perror("pipe");
69
70    printf("parent: fd: %d - %d\n", fd[0], fd[1]);
71    n = fork();
72    printf("parent: fork returned: %d\n", n);
73    if (n > 0) {
74        /* parent process */
75      server(fd);
76    } else if (n == 0) {
77        /* child process */
78        client(fd);
79    } else {
80        /* fork failed */
81      perror("fork");
82      exit(1);
83    }
84    return 0;
85  }
```

Απόσπασμα 4.2: IPC Test Programme

Κεφάλαιο 5

Κατακλείδα

Το cloud computing είναι αναπόφευκτα το κέντρο της προσοχής στις μέρες μας. Καθώς όλο και περισσότερες υπηρεσίες προσφέρονται online, οι απαιτήσεις προς το υλικό και το λογισμικό των cloud centres αυξήθηκε. Η έρευνα πάνω σε αυτό τον τομέα έχει οδηγήσει σε τεχνολογίες που βελτιώνουν όλο και περισσότερο την αξιοποίηση των πόρων τους. Σε αυτές έχουν προστεθεί και τα library λειτουργικά συστήματα ως ένας νέος τρόπος δόμησης των λειτουργικών συστημάτων.

Η αύξηση των πόρων που απαιτούνται οδήγησε στην αναθεώρηση του συμβατικού μοντέλου των λειτουργικών συστημάτων από μονολιθικό σε library λειτουργικά συστήματα. Το συμβατικό μοντέλο βασίζεται στην ανάγκη ενός λειτουργικού συστήματος να καλύψει μια ευρεία γκάμα αναγκών. Αυτό προέρχεται από μια εποχή όπου ένας υπολογιστής χρησιμοποιούταν από πολλαπλούς παράλληλους χρήστες όπως σε μοιραζόμενους υπολογιστές πανεπιστημίων. Στην σύγχρονη εποχή, όμως, πολλές υπηρεσίες έχουν μεταφερθεί από τους προσωπικές υπολογιστές σε κέντρα υπολογιστών με κάθε εικονική μηχανή να δημιουργείται σε πραγματικό χρόνο και να εξυπηρετεί ένα συγκεκριμένο σκοπό. Παρότι οι ανάγκες αλλάξανε, η χρήση συμβατικών λειτουργικών συστημάτων στα cloud centers αφήνει αναξιοποίητο μεγάλο ποσοστό των καταναλισκόμενων πόρων. Αυτή την ανάγκη προσπαθούν να λύσουν τα library λειτουργικά συστήματα, όπου κάθε κομμάτι του λειτουργικού μπορεί να χρησιμοποιηθεί ανεξάρτητα, αλλάζοντας τον τρόπο προσέγγισης της δόμησης των εφαρμογών. Δηλαδή, από μια εφαρμογή που προσπαθεί να προσαρμοστεί στο περιβάλλον του λειτουργικού συστήματος, στην δόμηση ενός λειτουργικού συστήματος στα μέτρα της εφαρμογής.

Αποτέλεσμα της εφαρμογής των library λειτουργικών συστημάτων είναι τα unikernels. Από την μία τα unikernels προσφέρουν γρήγορους χρόνους εκκίνησης, μικρό αποτύπωμα μνήμης και πλήρη απομόνωση από τις άλλες εικονικές μηχανές και τον φιλοξενητή με κατάλληλο hypervisor, όπως ο solo5. Από την άλλη, η τροποποίηση

των εφαρμογών για την εκτέλεσή τους σε unikernels είναι δύσκολη, καθώς κλήσεις συστήματος που προϋποθέτουν πολλές εφαρμογές δεν υποστηρίζονται στο περιβάλλον ενός unikernel.

Μια τέτοια κλήση συστήματος είναι η fork με την οποία καταπιάστηκε η εργασία με σκοπό την φιλοξένηση περισσότερων υπάρχοντων εφαρμογών σε unikernels με τις ελάχιστες τροποποιήσεις. Επίσης, ότι με τις συμβάσεις που έγιναν στα semantics της fork, είδαμε ότι ο χρόνος δημιουργίας μιας διεργασίας στο Rumprun unikernel περιβάλλον σε σχέση με την δημιουργία μιας διεργασίας σε εικονική μηχανή με συμβατικό λειτουργικό Linux είναι ίδια τάξη μεγέθους, καθώς και ότι IPC μηχανισμοί όπως το pipe λειτουργούν χωρίς αλλαγές.

5.1 Συμπεράσματα

Μέσα από την μελέτη των unikernel frameworks, παρατήρηθηκε η έλλειψη ασφάλειας της μνήμης μεταξύ των διεργασιών που δημιουργούνται. Αυτό μπορεί να δημιουργήσει κενά ασφαλείας, το οποίο μπορεί να λυθεί ακόμη και μέσα από την επέκταση του υλικού, όπως είδαμε στο 3.7. Επίσης, παρότι τα unikernels αποτελούν μια πολύ καλή μελλοντική προσέγγιση για το deployment εφαρμογών στο cloud, παραμένει ακόμη η δυσκολία στην αποσυμπλοκή των συμβατικών λειτουργικών συστημάτων σε ανεξάρτητες βιβλιοθήκες, και η δημιουργία περισσότερων υποστηριζόμενων λειτουργιών που καλύπτουν τα συμβατικά λειτουργικά συστήματα που χρησιμοποιούνται ήδη σε cloud centers.

Η ενασχόληση με την υλοποίηση της fork και γενικά με τους μηχανισμούς του Rumprun και του solo5 οδήγησε στα εξής συμπεράσματα. Αρχικά, όπως αναφέρθηκε το Rumprun δεν υποστηρίζει memory virtualization. Μία προσέγγιση για την περαιτέρω ανάπτυξη του παρόντος σχεδιασμού θα ήταν η υλοποίηση ενός memory virtualization συστήματος, για την υποστήριξη πολλαπλών χώρων διευθύνσεων. Αυτό βέβαια θα πρόσθετε πολυπλοκότητα στο Rumprun και θα επιβράδυνε την εκτέλεση των μηχανών unikernels. Από την άλλη θα έκανε πιο εύκολη την υλοποίηση περισσότερων κλήσεων συστήματος, που με την σειρά του θα επέτρεπε σε περισσότερες POSIX εφαρμογές να τρέξουν στο Rumprun.

Μια ακόμη παρατήρηση που έγινε είναι ότι ο τρόπος που κρατάει τα μεταδεδομένα για την μνήμη το Rumprun δεν έχει προσχεδιαστεί για την υποστήριξη πολλαπλών διεργασιών. Δηλαδή, ο τρόπος που αναθέτει την μνήμη δεν διακρίνει μεταξύ διαφορετικών διεργασιών. Αυτό είναι απόρροια της χρήσης POSIX νημάτων για την εκτέλεση του, το οποίο σε συνδυασμό με την μονοδιεργασιακή προσέγγιση των μηχανών, οδήγησε στην έλλειψη διαχωρισμού των heap sections των νημάτων. Επομένως, η επέκταση των μεταδεδομένων για να χρησιμοποιούνται σε συνδυασμό με την δημιουργία

ενός rumprunner θα μπορούσε να είναι μια προσέγγιση για την λύση της έλλειψης αντιγραφής του heap κατά την κλήση της fork.

Ακόμη, όμως, και να απευθυνθεί το παραπάνω πρόβλημα, το data section παραμένει ένα σημείο της μνήμης που δεν είναι προσβάσιμο μέσω του Rumprun. Αυτό οφείλεται στην έλλειψη αναφοράς στο data section μέσα από την εικονική μηχανή. Για την πρόσβαση και αντιγραφή του data section θα πρέπει να γραφεί assembly κώδικας για την αποθήκευσή του και επαναφορά του κατά την δρομολόγιση. Από την άλλη, θα πρέπει να επεκταθεί η δομή του νήματος του εικονικού επεξεργαστή, για να διατηρείτε η αναφορά στα ξεχωριστά data sections των νημάτων καθώς και να γίνεται σωστή αρχικοποίηση τους κατά την δημιουργία τους.

Το Rumprun υποστηρίζει την ύπαρξη πολλαπλών εκτελέσιμων στην τελική του εικόνα και την εκτέλεσή τους ως rumprunners. Επομένως ένα επόμενο βήμα μετά την υλοποίηση της fork με τα semantics της παρούσας εργασίας, θα ήταν η υποστήριξη της κλήσης exec. Παρότι το heap και το data segment δεν αντιγράφονται, η ύπαρξη διαφορετικού namespace θα απέτρεπε την ακούσια παρεμβολή μεταξύ των εκτελέσιμων. Με αυτό τον τρόπο εφαρμογές που χρησιμοποιούν τον συνδυασμό fork και exec για την εκτέλεση προγραμμάτων στα παιδιά τους, θα μπορούσαν να υποστηριχθούν στο Rumprun.

Βιβλιογραφία

[1] Michal Bližňák, Jiri Vojtesek, Radek Matušů, and Tomáš Dulík. Virtualization as a teaching tool. pages 214–217, 09 2008.

[2] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, USA, 2011.

[3] 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 *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Aalto University, 2012.

[5] 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*, NSDI'14, page 459–473, USA, 2014. USENIX Association.

[6] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling fast, dynamic network processing with clickos. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 67–72, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 2015 IEEE 7th International Conference on*

Cloud Computing Technology and Science (CloudCom), CLOUDCOM '15, page 250–257, USA, 2015. IEEE Computer Society.

[8] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Os^{v}: Optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 61–72, USA, 2014. USENIX Association.

[9] 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*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[10] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys (MPK). *arXiv e-prints*, page arXiv:1801.06822, January 2018.

[12] Χαράλαμπος Μαϊνάς. Σχεδιασμός και υλοποίηση μηχανισμών fork και pipe σε unikernels. Master's thesis, Εθνικό Μετσόβειο Πολυτεχνείο, 2019.

[13] A. Cameron Macdonell. *Shared-Memory Optimizations for Virtual Machines*. PhD thesis, CAN, 2011. AAINR89468.

[14] Mel gorman. chapter 6: Understanding the linux virtual memory manager. `https://www.kernel.org/doc/gorman/html/understand/understand009.html`. [Online; Last Accessed 2021-04-21].

[15] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 199–211, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*,

SoCC '18, page 199–211, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Antti Kantee. The rise and fall of the operating system. *login Usenix Mag.*, 40(5), 2015.

[18] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11(11):30–44, December 2013.

[19] Antti Kantee and Justin Cormack. Rump kernels: No os? no problem! *login Usenix Mag.*, 39(5), 2014.

[20] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, page 71–76, USA, 2016. USENIX Association.